目　　录

第 34 章　进程组、会话和作业控制 ····· 573

34.1　概述 ····· 573
34.2　进程组 ····· 575
34.3　会话 ····· 577
34.4　控制终端和控制进程 ····· 578
34.5　前台和后台进程组 ····· 580
34.6　SIGHUP 信号 ····· 581
34.6.1　在 shell 中处理 SIGHUP 信号 ····· 581
34.6.2　SIGHUP 和控制进程的终止 ····· 583
34.7　作业控制 ····· 585
34.7.1　在 shell 中使用作业控制 ····· 585
34.7.2　实现作业控制 ····· 587
34.7.3　处理作业控制信号 ····· 591
34.7.4　孤儿进程组（SIGHUP 回顾） ····· 594
34.8　总结 ····· 598
34.9　习题 ····· 599

第 35 章　进程优先级和调度 ····· 600

35.1　进程优先级（nice 值） ····· 600
35.2　实时进程调度概述 ····· 603
35.2.1　SCHED_RR 策略 ····· 604
35.2.2　SCHED_FIFO 策略 ····· 605
35.2.3　SCHED_BATCH 和 SCHED_IDLE 策略 ····· 605
35.3　实时进程调用 API ····· 605
35.3.1　实时优先级范围 ····· 606
35.3.2　修改和获取策略和优先级 ····· 606
35.3.3　释放 CPU ····· 611

更多资源请访问稀酷客(www.ckook.com)

35.3.4 SCHED_RR 时间片 …… 611
35.4 CPU 亲和力 …… 612
35.5 总结 …… 614
35.6 习题 …… 615

第 36 章 进程资源 …… 617

36.1 进程资源使用 …… 617
36.2 进程资源限制 …… 619
36.3 特定资源限制细节 …… 623
36.4 总结 …… 627
36.5 习题 …… 627

第 37 章 DAEMON …… 628

37.1 概述 …… 628
37.2 创建一个 daemon …… 629
37.3 编写 daemon 指南 …… 632
37.4 使用 SIGHUP 重新初始化一个 daemon …… 632
37.5 使用 syslog 记录消息和错误 …… 635
37.5.1 概述 …… 635
37.5.2 syslog API …… 636
37.5.3 /etc/syslog.conf 文件 …… 640
37.6 总结 …… 641
37.7 习题 …… 641

第 38 章 编写安全的特权程序 …… 642

38.1 是否需要一个 Set-User-ID 或 Set-Group-ID 程序? …… 642
38.2 以最小权限操作 …… 643
38.3 小心执行程序 …… 645
38.4 避免暴露敏感信息 …… 646
38.5 确定进程的边界 …… 647
38.6 小心信号和竞争条件 …… 647
38.7 执行文件操作和文件 I/O 的缺陷 …… 648
38.8 不要完全相信输入和环境 …… 648
38.9 小心缓冲区溢出 …… 649
38.10 小心拒绝服务攻击 …… 650
38.11 检查返回状态和安全地处理失败情况 …… 651
38.12 总结 …… 651
38.13 习题 …… 652

第 39 章 能力 …… 653

39.1 能力基本原理 …… 653

更多资源请访问稀酷客(www.ckook.com)

39.2 Linux 能力 ····· 654
39.3 进程和文件能力 ····· 654
39.3.1 进程能力 ····· 654
39.3.2 文件能力 ····· 655
39.3.3 进程许可和有效能力集的目的 ····· 657
39.3.4 文件许可和有效能力集的目的 ····· 657
39.3.5 进程和文件可继承集的目的 ····· 658
39.3.6 在 shell 中给文件赋予能力和查看文件能力 ····· 658
39.4 现代能力实现 ····· 659
39.5 在 exec()中转变进程能力 ····· 659
39.5.1 能力边界集 ····· 660
39.5.2 保持 root 语义 ····· 660
39.6 改变用户 ID 对进程能力的影响 ····· 661
39.7 用编程的方式改变进程能力 ····· 661
39.8 创建仅包含能力的环境 ····· 665
39.9 发现程序所需的能力 ····· 667
39.10 不具备文件能力的老式内核和系统 ····· 667
39.11 总结 ····· 669
39.12 习题 ····· 669

第 40 章 登录记账 ····· 670

40.1 utmp 和 wtmp 文件概述 ····· 670
40.2 utmpx API ····· 671
40.3 utmpx 结构 ····· 671
40.4 从 utmp 和 wtmp 文件中检索信息 ····· 673
40.5 获取登录名称：getlogin() ····· 676
40.6 为登录会话更新 utmp 和 wtmp 文件 ····· 677
40.7 lastlog 文件 ····· 681
40.8 总结 ····· 683
40.9 习题 ····· 683

第 41 章 共享库基础 ····· 684

41.1 目标库 ····· 684
41.2 静态库 ····· 685
41.3 共享库概述 ····· 686
41.4 创建和使用共享库——首回合 ····· 687
41.4.1 创建一个共享库 ····· 687
41.4.2 位置独立的代码 ····· 687
41.4.3 使用一个共享库 ····· 688
41.4.4 共享库 soname ····· 689

更多资源请访问稀酷客(www.ckook.com)

41.5 使用共享库的有用工具 …… 691
41.6 共享库版本和命名规则 …… 692
41.7 安装共享库 …… 694
41.8 兼容与不兼容库比较 …… 696
41.9 升级共享库 …… 697
41.10 在目标文件中指定库搜索目录 …… 698
41.11 在运行时找出共享库 …… 700
41.12 运行时符号解析 …… 700
41.13 使用静态库取代共享库 …… 701
41.14 总结 …… 702
41.15 习题 …… 703

第 42 章 共享库高级特性 …… 704

42.1 动态加载库 …… 704
42.1.1 打开共享库：dlopen() …… 705
42.1.2 错误诊断：dlerror() …… 706
42.1.3 获取符号的地址：dlsym() …… 707
42.1.4 关闭共享库：dlclose() …… 709
42.1.5 获取与加载的符号相关的信息：dladdr() …… 710
42.1.6 在主程序中访问符号 …… 710
42.2 控制符号的可见性 …… 710
42.3 链接器版本脚本 …… 711
42.3.1 使用版本脚本控制符号的可见性 …… 712
42.3.2 符号版本化 …… 713
42.4 初始化和终止函数 …… 715
42.5 预加载共享库 …… 716
42.6 监控动态链接器：LD_DEBUG …… 716
42.7 总结 …… 717
42.8 习题 …… 718

第 43 章 进程间通信简介 …… 719

43.1 IPC 工具分类 …… 719
43.2 通信工具 …… 720
43.3 同步工具 …… 721
43.4 IPC 工具比较 …… 723
43.5 总结 …… 727
43.6 习题 …… 727

第 44 章 管道和 FIFO …… 728

44.1 概述 …… 728

更多资源请访问稀酷客(www.ckook.com)

44.2 创建和使用管道……730
44.3 将管道作为一种进程同步的方法……735
44.4 使用管道连接过滤器……737
44.5 通过管道与 Shell 命令进行通信：popen()……739
44.6 管道和 stdio 缓冲……743
44.7 FIFO……743
44.8 使用管道实现一个客户端/服务器应用程序……745
44.9 非阻塞 I/O……751
44.10 管道和 FIFO 中 read()和 write()的语义……752
44.11 总结……753
44.12 习题……754

第 45 章 System V IPC 介绍……756

45.1 概述……757
45.2 IPC Key……759
45.3 关联数据结构和对象权限……761
45.4 IPC 标识符和客户端/服务器应用程序……763
45.5 System V IPC get 调用使用的算法……764
45.6 ipcs 和 ipcrm 命令……766
45.7 获取所有 IPC 对象列表……767
45.8 IPC 限制……767
45.9 总结……768
45.10 习题……768

第 46 章 System V 消息队列……769

46.1 创建或打开一个消息队列……769
46.2 交换消息……771
46.2.1 发送消息……772
46.2.2 接收消息……774
46.3 消息队列控制操作……777
46.4 消息队列关联数据结构……778
46.5 消息队列的限制……780
46.6 显示系统中所有消息队列……781
46.7 使用消息队列实现客户端/服务器应用程序……783
46.8 使用消息队列实现文件服务器应用程序……784
46.9 System V 消息队列的缺点……790
46.10 总结……790
46.11 习题……791

第 47 章 System V 信号量……792

47.1 概述……793

更多资源请访问稀酷客(www.ckook.com)

47.2 创建或打开一个信号量集 …… 795
47.3 信号量控制操作 …… 796
47.4 信号量关联数据结构 …… 798
47.5 信号量初始化 …… 801
47.6 信号量操作 …… 803
47.7 多个阻塞信号量操作的处理 …… 809
47.8 信号量撤销值 …… 810
47.9 实现一个二元信号量协议 …… 811
47.10 信号量限制 …… 814
47.11 System V 信号量的缺点 …… 815
47.12 总结 …… 816
47.13 习题 …… 817

第 48 章 System V 共享内存 …… 818

48.1 概述 …… 818
48.2 创建或打开一个共享内存段 …… 819
48.3 使用共享内存 …… 820
48.4 示例：通过共享内存传输数据 …… 821
48.5 共享内存在虚拟内存中的位置 …… 825
48.6 在共享内存中存储指针 …… 828
48.7 共享内存控制操作 …… 829
48.8 共享内存关联数据结构 …… 830
48.9 共享内存的限制 …… 832
48.10 总结 …… 833
48.11 习题 …… 833

第 49 章 内存映射 …… 835

49.1 概述 …… 835
49.2 创建一个映射：mmap() …… 837
49.3 解除映射区域：munmap() …… 840
49.4 文件映射 …… 840
49.4.1 私有文件映射 …… 841
49.4.2 共享文件映射 …… 842
49.4.3 边界情况 …… 845
49.4.4 内存保护和文件访问模式交互 …… 846
49.5 同步映射区域：msync() …… 847
49.6 其他 mmap()标记 …… 848
49.7 匿名映射 …… 849
49.8 重新映射一个映射区域：mremap() …… 852
49.9 MAP_NORESERVE 和过度利用交换空间 …… 853
49.10 MAP_FIXED 标记 …… 854

更多资源请访问稀酷客(www.ckook.com)

49.11 非线性映射：remap_file_pages() 855
49.12 总结 857
49.13 习题 858

第 50 章 虚拟内存操作 859

50.1 改变内存保护：mprotect() 859
50.2 内存锁：mlock()和 mlockatt() 861
50.3 确定内存驻留性：mincore() 864
50.4 建议后续的内存使用模式：madvise() 866
50.5 小结 868
50.6 习题 868

第 51 章 POSIX IPC 介绍 869

51.1 API 概述 869
51.2 System V IPC 与 POSIX IPC 比较 872
51.3 总结 873

第 52 章 POSIX 消息队列 874

52.1 概述 874
52.2 打开、关闭和断开链接消息队列 875
52.3 描述符和消息队列之间的关系 877
52.4 消息队列特性 878
52.5 交换消息 882
52.5.1 发送消息 882
52.5.2 接收消息 883
52.5.3 在发送和接收消息时设置超时时间 885
52.6 消息通知 886
52.6.1 通过信号接收通知 887
52.6.2 通过线程接收通知 889
52.7 Linux 特有的特性 891
52.8 消息队列限制 892
52.9 POSIX 和 System V 消息队列比较 893
52.10 总结 894
52.11 习题 894

第 53 章 POSIX 信号量 895

53.1 概述 895
53.2 命名信号量 895
53.2.1 打开一个命名信号量 896
53.2.2 关闭一个信号量 898
53.2.3 删除一个命名信号量 898

更多资源请访问稀酷客(www.ckook.com)

53.3 信号量操作 …… 899
53.3.1 等待一个信号量 …… 899
53.3.2 发布一个信号量 …… 901
53.3.3 获取信号量的当前值 …… 901
53.4 未命名信号量 …… 903
53.4.1 初始化一个未命名信号量 …… 904
53.4.2 销毁一个未命名信号量 …… 906
53.5 与其他同步技术比较 …… 906
53.6 信号量的限制 …… 907
53.7 总结 …… 908
53.8 习题 …… 908

第 54 章 POSIX 共享内存 …… 909

54.1 概述 …… 909
54.2 创建共享内存对象 …… 910
54.3 使用共享内存对象 …… 913
54.4 删除共享内存对象 …… 915
54.5 共享内存 APIs 比较 …… 915
54.6 总结 …… 916
54.7 习题 …… 917

第 55 章 文件加锁 …… 918

55.1 概述 …… 918
55.2 使用 flock()给文件加锁 …… 920
55.2.1 锁继承与释放的语义 …… 922
55.2.2 flock()的限制 …… 923
55.3 使用 fcntl()给记录加锁 …… 923
55.3.1 死锁 …… 928
55.3.2 示例：一个交互式加锁程序 …… 928
55.3.3 示例：一个加锁函数库 …… 931
55.3.4 锁的限制和性能 …… 933
55.3.5 锁继承和释放的语义 …… 934
55.3.6 锁定饿死和排队加锁请求的优先级 …… 935
55.4 强制加锁 …… 935
55.5 /proc/locks 文件 …… 938
55.6 仅运行一个程序的单个实例 …… 939
55.7 老式加锁技术 …… 941
55.8 总结 …… 942
55.9 习题 …… 943

第 56 章 SOCKET：介绍 …… 945

更多资源请访问稀酷客(www.ckook.com)

56.1 概述……945
56.2 创建一个 socket：socket()……948
56.3 将 socket 绑定到地址：bind()……948
56.4 通用 socket 地址结构：struct sockaddr……949
56.5 流 socket……950
56.5.1 监听接入连接：listen()……951
56.5.2 接受连接：accept()……952
56.5.3 连接到对等 socket：connect()……952
56.5.4 流 socket I/O……953
56.5.5 连接终止：close()……953
56.6 数据报 socket……953
56.6.1 交换数据报：recvfrom 和 sendto()……954
56.6.2 在数据报 socket 上使用 connect()……955
56.7 总结……956

第 57 章 SOCKET：UNIX DOMAIN……957

57.1 UNIX domain socket 地址：struct sockaddr_un……957
57.2 UNIX domain 中的流 socket……959
57.3 UNIX domain 中的数据报 socket……962
57.4 UNIX domain socket 权限……965
57.5 创建互联 socket 对：socketpair()……965
57.6 Linux 抽象 socket 名空间……966
57.7 总结……967
57.8 习题……967

第 58 章 SOCKET：TCP/IP 网络基础……968

58.1 因特网……968
58.2 联网协议和层……969
58.3 数据链路层……971
58.4 网络层：IP……971
58.5 IP 地址……973
58.6 传输层……975
58.6.1 端口号……975
58.6.2 用户数据报协议（UDP）……976
58.6.3 传输控制协议（TCP）……977
58.7 请求注解（RFC）……979
58.8 总结……980

第 59 章 SOCKET：Internet DOMAIN……982

59.1 Internet domain socket……982

更多资源请访问稀酷客(www.ckook.com)

59.2 网络字节序 …… 982
59.3 数据表示 …… 984
59.4 Internet socket 地址 …… 986
59.5 主机和服务转换函数概述 …… 988
59.6 inet_pton()和 inet_ntop()函数 …… 989
59.7 客户端-服务器示例（数据报 socket） …… 990
59.8 域名系统（DNS） …… 992
59.9 /etc/services 文件 …… 994
59.10 独立于协议的主机和服务转换 …… 995
59.10.1 getaddrinfo()函数 …… 996
59.10.2 释放 addrinfo 列表：freeaddrinfo() …… 998
59.10.3 错误诊断：gai_strerror() …… 999
59.10.4 getnameinfo()函数 …… 999
59.11 客户端-服务器示例（流式 socket） …… 1000
59.12 Internet domain socket 库 …… 1006
59.13 过时的主机和服务转换 API …… 1010
59.13.1 inet_aton()和 inet_ntoa()函数 …… 1010
59.13.2 gethostbyname()和 gethostbyaddr()函数 …… 1010
59.13.3 getserverbyname()和 getserverbyport()函数 …… 1012
59.14 UNIX 与 Internet domain socket 比较 …… 1013
59.15 更多信息 …… 1014
59.16 总结 …… 1014
59.17 习题 …… 1015

第 60 章 SOCKET：服务器设计 …… 1016

60.1 迭代型和并发型服务器 …… 1016
60.2 迭代型 UDP echo 服务器 …… 1016
60.3 并发型 TCP echo 服务器 …… 1019
60.4 并发型服务器的其他设计方案 …… 1021
60.5 inetd（Internet 超级服务器）守护进程 …… 1023
60.6 总结 …… 1027
60.7 练习 …… 1027

第 61 章 SOCKET：高级主题 …… 1028

61.1 流式套接字上的部分读和部分写 …… 1028
61.2 shutdown()系统调用 …… 1030
61.3 专用于套接字的 I/O 系统调用：recv()和 send() …… 1033
61.4 sendfile()系统调用 …… 1034
61.5 获取套接字地址 …… 1036
61.6 深入探讨 TCP 协议 …… 1039

更多资源请访问稀酷客(www.ckook.com)

61.6.1 TCP 报文的格式 …… 1039
61.6.2 TCP 序列号和确认机制 …… 1041
61.6.3 TCP 协议状态机以及状态迁移图 …… 1041
61.6.4 TCP 连接的建立 …… 1043
61.6.5 TCP 连接的终止 …… 1044
61.6.6 在 TCP 套接字上调用 shutdown() …… 1045
61.6.7 TIME_WAIT 状态 …… 1045
61.7 监视套接字：netstat …… 1047
61.8 使用 tcpdump 来监视 TCP 流量 …… 1048
61.9 套接字选项 …… 1049
61.10 SO_REUSEADDR 套接字选项 …… 1050
61.11 在 accept()中继承标记和选项 …… 1051
61.12 TCP vs UDP …… 1052
61.13 高级功能 …… 1053
61.13.1 带外数据 …… 1053
61.13.2 sendmsg()和 recvmsg()系统调用 …… 1053
61.13.3 传递文件描述符 …… 1054
61.13.4 接收发送端的凭据 …… 1054
61.13.5 顺序数据包套接字 …… 1055
61.13.6 SCTP 以及 DCCP 传输层协议 …… 1055
61.14 总结 …… 1056
61.15 练习 …… 1056

第 62 章 终端 …… 1058

62.1 整体概览 …… 1059
62.2 获取和修改终端属性 …… 1060
62.3 stty 命令 …… 1062
62.4 终端特殊字符 …… 1063
62.5 终端标志 …… 1068
62.6 终端的 I/O 模式 …… 1073
62.6.1 规范模式 …… 1073
62.6.2 非规范模式 …… 1074
62.6.3 加工模式、cbreak 模式以及原始模式 …… 1075
62.7 终端线速（比特率） …… 1081
62.8 终端的行控制 …… 1082
62.9 终端窗口大小 …… 1084
62.10 终端标识 …… 1085
62.11 总结 …… 1086
62.12 练习 …… 1087

更多资源请访问稀酷客(www.ckook.com)

第 63 章　其他备选的 I/O 模型 …… 1088

63.1　整体概览 …… 1088
63.1.1　水平触发和边缘触发 …… 1091
63.1.2　在备选的 I/O 模型中采用非阻塞 I/O …… 1092
63.2　I/O 多路复用 …… 1092
63.2.1　select()系统调用 …… 1092
63.2.2　poll()系统调用 …… 1097
63.2.3　文件描述符何时就绪？ …… 1101
63.2.4　比较 select()和 poll() …… 1103
63.2.5　select()和 poll()存在的问题 …… 1105
63.3　信号驱动 I/O …… 1105
63.3.1　何时发送“I/O 就绪”信号 …… 1109
63.3.2　优化信号驱动 I/O 的使用 …… 1110
63.4　epoll 编程接口 …… 1113
63.4.1　创建 epoll 实例：epoll_create() …… 1113
63.4.2　修改 epoll 的兴趣列表：epoll_ctl() …… 1114
63.4.3　事件等待：epoll_wait() …… 1115
63.4.4　深入探究 epoll 的语义 …… 1120
63.4.5　epoll 同 I/O 多路复用的性能对比 …… 1121
63.4.6　边缘触发通知 …… 1122
63.5　在信号和文件描述符上等待 …… 1124
63.5.1　pselect()系统调用 …… 1125
63.5.2　self-pipe 技巧 …… 1126
63.6　总结 …… 1128
63.7　练习 …… 1129

第 64 章　伪终端 …… 1130

64.1　整体概览 …… 1130
64.2　UNIX98 伪终端 …… 1133
64.2.1　打开未使用的主设备：posix_openpt() …… 1134
64.2.2　修改从设备属主和权限：grantpt() …… 1135
64.2.3　解锁从设备：unlockpt() …… 1135
64.2.4　获取从设备名称：ptsname() …… 1136
64.3　打开主设备：ptyMasterOpen() …… 1136
64.4　将进程连接到伪终端：ptyFork() …… 1138
64.5　伪终端 I/O …… 1140
64.6　实现 script(1)程序 …… 1142
64.7　终端属性和窗口大小 …… 1146
64.8　BSD 风格的伪终端 …… 1146

更多资源请访问稀酷客(www.ckook.com)

64.9 总结 …… 1148
64.10 练习 …… 1149

附录 A 跟踪系统调用 …… 1151

附录 B 解析命令行选项 …… 1153

附录 C 对 NULL 指针做转型 …… 1159

附录 D 内核配置 …… 1161

附录 E 更多信息源 …… 1162

附录 F 部分习题解答 …… 1167

更多资源请访问稀酷客(www.ckook.com)

第34章

进程组、会话和作业控制

进程组和会话在进程之间形成了一种两级层次关系：进程组是一组相关进程的集合，会话是一组相关进程组的集合。读者通过本章的学习就能弄清楚两个术语中“相关”的含义。

进程组和会话是为支持 shell 作业控制而定义的抽象概念，用户通过 shell 能够交互式地在前台或后台运行命令。术语“作业”通常与术语“进程组”作为同义词来看待。

本章将介绍进程组、会话和作业控制。

34.1 概述

进程组由一个或多个共享同一进程组标识符（PGID）的进程组成。进程组 ID 是一个数字，其类型与进程 ID 一样（pid_t）。一个进程组拥有一个进程组首进程，该进程是创建该组的进程，其进程 ID 为该进程组的 ID，新进程会继承其父进程所属的进程组 ID。

进程组拥有一个生命周期，其开始时间为首进程创建组的时刻，结束时间为最后一个成员进程退出组的时刻。一个进程可能会因为终止而退出进程组，也可能会因为加入了另外一个进程组而退出进程组。进程组首进程无需是最后一个离开进程组的成员。

会话是一组进程组的集合。进程的会话成员关系是由其会话标识符（SID）确定的，会话标识符与进程组 ID 一样，是一个类型为 pid_t 的数字。会话首进程是创建该新会话的进程，其进程 ID 会成为会话 ID。新进程会继承其父进程的会话 ID。

一个会话中的所有进程共享单个控制终端。控制终端会在会话首进程首次打开一个终端设备时被建立。一个终端最多可能会成为一个会话的控制终端。

在任一时刻，会话中的其中一个进程组会成为终端的前台进程组，其他进程组会成为后台进程组。只有前台进程组中的进程才能从控制终端中读取输入。当用户在控制终端中输入其中一个信号生成终端字符之后，该信号会被发送到前台进程组中的所有成员。这些字符包括生成 SIGINT 的中断字符（通常是 Control-C）、生成 SIGQUIT 的退出字符（通常是 Control-\）、生成 SIGSTP 的挂起字符（通常是 Control-Z）。

当到控制终端的连接建立起来（即打开）之后，会话首进程会成为该终端的控制进程。成为控制进程的主要标志是当断开与终端之间的连接时内核会向该进程发送一个 SIGHUP 信号。

更多资源请访问稀酷客(www.ckook.com)

通过检查 Linux 特有的/proc/PID/stat 文件，就能确定任意进程的进程组 ID 和会话 ID。此外，还能确定进程的控制终端的设备 ID（一个十进制数字，包含主 ID 和辅 ID）和控制该终端的控制进程的进程 ID。更多细节信息请参考 proc(5)手册。

会话和进程组的主要用途是用于 shell 作业控制。读者通过一个具体的例子就能够弄清楚这些概念了。如对于交互式登录来讲，控制终端是用户登录的途径。登录 shell 是会话首进程和终端的控制进程，也是其自身进程组的唯一成员。从 shell 中发出的每个命令或通过管道连接的一组命令都会导致一个或多个进程的创建，并且 shell 会把所有这些进程都放在一个新进程组中。（这些进程在一开始是其进程组中的唯一成员，它们创建的所有子进程会成为该组中的成员。）当命令或以管道连接的一组命令以&符号结束时会在后台进程组中运行这些命令，否则就会在前台进程组中运行这些命令。在登录会话中创建的所有进程都会成为该会话的一部分。

在窗口环境中，控制终端是一个伪终端。每个终端窗口都有一个独立的会话，窗口的启动 shell 是会话首进程和终端的控制进程。

在除任务控制之外的其他场景中也有可能用到进程组，因为进程组具备两个有用的属性：在特定的进程组中父进程能够等待任意子进程（参见 26.1.2 节）和信号能够被发送给进程组中的所有成员（参见 20.5 节）。

图 34.1 给出了执行下面的命令之后各个进程之间的进程组和会话关系。

```
$ echo $$                          Display the PID of the shell
400
$ find / 2> /dev/null | wc -l &    Creates 2 processes in background group
[1] 659
$ sort < longlist | uniq -c        Creates 2 processes in foreground group
```

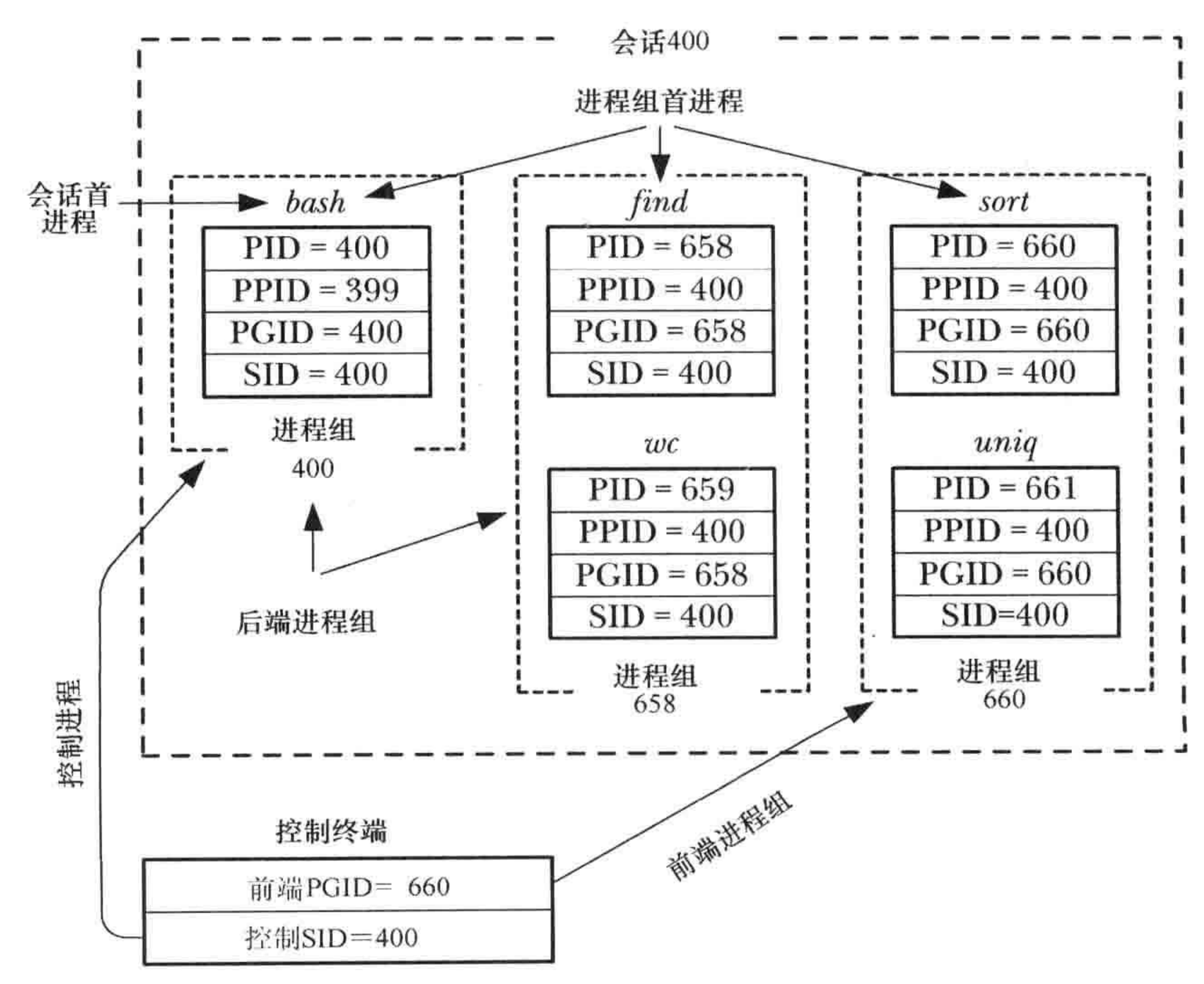


图 34.1　进程组、会话和控制终端之间的关系

34.2 进程组

每个进程都拥有一个以数字表示的进程组 ID，表示该进程所属的进程组。新进程会继承其父进程的进程组 ID，使用 getpgrp()能够获取一个进程的进程组 ID。

```
#include <unistd.h>

pid_t getpgrp(void);
```

Always successfully returns process group ID of calling process

如果 getpgrp()的返回值与调用进程的进程 ID 匹配的话就说明该调用进程是其进程组的首进程。

setpgid()系统调用将进程 ID 为 pid 的进程的进程组 ID 修改为 pgid。

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Returns 0 on success, or −1 on error

如果将 pid 的值设置为 0，那么调用进程的进程组 ID 就会被改变。如果将 pgid 的值设置为 0，那么 ID 为 pid 的进程的进程组 ID 会被设置成 pid 的值。因此，下面的 setpgid()调用是等价的。

```
setpgid(0, 0);
setpgid(getpid(), 0);
setpgid(getpid(), getpid());
```

如果 pid 和 pgid 参数指定了同一个进程（即 pgid 是 0 或者与 ID 为 pid 的进程的进程 ID 匹配），那么就会创建一个新进程组，并且指定的进程会成为这个新组的首进程（即进程的进程组 ID 与进程 ID 是一样的）。如果两个参数的值不同（即 pgid 不是 0 或者与 ID 为 pid 的进程的进程 ID 不匹配），那么 setpgid()调用会将一个进程从一个进程组中移到另一个进程组中。

通常调用 setpgid()（以及 34.3 节中介绍的 setsid()）函数的是 shell 和 login(1)。在 37.2 节中将会看到一个程序在使自己变成 daemon 的过程中也会调用 setsid()。

在调用 setpgid()时存在以下限制。

- pid 参数可以仅指定调用进程或其中一个子进程。违反这条规则会导致 ESRCH 错误。
- 在组之间移动进程时，调用进程、由 pid 指定的进程（可能是另外一个进程，也可能就是调用进程）以及目标进程组必须要属于同一个会话。违反这条规则会导致 EPERM 错误。
- pid 参数所指定的进程不能是会话首进程。违反这条规则会导致 EPERM 错误。
- 一个进程在其子进程已经执行 exec()后就无法修改该子进程的进程组 ID 了。违反这条规则会导致 EACCES 错误。之所以会有这条约束条件的原因是在一个进程开始执行之后再修改其进程组 ID 的话会使程序变得混乱。

在作业控制 shell 中使用 setpgid()

一个进程在其子进程已经执行 exec()之后就无法修改该子进程的进程组 ID 的约束条件会影响到基于 shell 的作业控制程序设计，即需要满足下列条件。

- 一个任务（即一个命令或一组以管道符连接的命令）中的所有进程必须被放置在一个进程组中。（通过图 34-1 中 bash 创建的两个进程组就能看出。）这一步允许 shell 使用

更多资源请访问稀酷客(www.ckook.com)

killpg()（或使用负的 pid 值来调用 kill()）来同时向进程组中的所有成员发送作业控制信号。一般来讲，这一步需要在发送任意作业控制信号前完成。

- 每个子进程在执行程序之前必须要被分配到进程组中，因为程序本身是不清楚如何操作进程组 ID 的。

对于任务中的各个进程来讲，父进程和子进程都可以使用 setpgid()来修改子进程的进程组 ID。但是，由于在父进程执行 fork()（参见 24.4 节）之后父进程与子进程之间的调度顺序是无法确定的，因此无法依靠父进程在子进程执行 exec()之前来改变子进程的进程组 ID，同样也无法依靠子进程在父进程向其发送任意作业控制信号之前修改其进程组 ID。（依赖这些行为中的任意一个行为都会导致竞争条件。）因此，在编写作业控制 shell 程序时需要让父进程和子进程在 fork()调用之后立即调用 setpgid()来将子进程的进程组 ID 设置为同样的值，并且父进程需要忽略在 setpgid()调用中出现的所有 EACCES 错误。换句话说，在一个作业控制 shell 程序中可能会出现像程序清单 34-1 中给出的代码。

程序清单 34-1：作业控制 shell 程序如何设置子进程的进程组 ID

```
pid_t childPid;
pid_t pipelinePgid;          /* PGID to which processes in a pipeline
                                are to be assigned */
/* Other code */

childPid = fork();
switch (childPid) {
case -1: /* fork() failed */
    /* Handle error */

case 0: /* Child */
    if (setpgid(0, pipelinePgid) == -1)
        /* Handle error */
    /* Child carries on to exec the required program */

default: /* Parent (shell) */
    if (setpgid(childPid, pipelinePgid) == -1 && errno != EACCES)
        /* Handle error */
    /* Parent carries on to do other things */
}
```

在处理由管道符连接起来的命令时事情会变得比程序清单 34-1 更加复杂一点，父 shell 需要记录管道中第一个进程的进程 ID 并使用这个值作为该组中所有进程的进程组 ID（pipelinePgid）。

获取和修改进程组 ID 的其他（过时的）接口

这里需要解释一下为何 getpgrp()和 setpgid()两个系统调用名称中的后缀不同。

在一开始，4.2BSD 提供了一个 getprgp(pid)系统调用来返回进程 ID 为 pid 的进程的进程组 ID。在实践中，pid 几乎总是用来表示调用进程。结果，POSIX 委员会认为这个系统调用过于复杂了，因此他们采纳了 System V getpgrp()系统调用，这个系统调用不接收任何参数并返回调用进程的进程组 ID。

为了修改进程组 ID，4.2BSD 提供了 setpgrp(pid,pgid)系统调用，它与 setpgid()的行为是相似的。这两个系统调用之间最主要的差别在于 BSD setpgrp()能够用来将进程组 ID 设置为任意值。（前面曾经提及过不能使用 setpgid()将一个进程迁移至其他会话中的进程组。）这会引起一

更多资源请访问稀酷客(www.ckook.com)

些安全问题，但在实现任务控制程序时也更加灵活。结果，POSIX 委员会决定给这个函数增加额外的限制条件并将其命名为 setpgid()。

更复杂的事情是 SUSv3 指定了一个 getpgid(pid)系统调用，它与老式的 BSD getpgrp()的功能是一样的。此外，它还定义了一个从 System V 演化而来的 setpgrp()，它不接受任何参数，与 setpgid(0, 0)调用几乎是等价的。

尽管对于实现 shell 作业控制来讲，利用前面介绍的 setpgid()和 getpgrp()系统调用已经足够了。但与其他大多数 UNIX 实现一样，Linux 也提供了 getpgid(pid)和 setpgrp(void)。为了向后兼容，很多从 BSD 演化而来的实现仍然提供了 setprgp(pid, pgid)，它与 setpgid(pid, pgid)是一样的。

在编译程序时如果显式地定义_BSD_SOURCE 特性测试宏的话，glibc 会使用从 BSD 演化而来的 setpgrp()和 getpgrp()来取代默认版本。

34.3 会话

会话是一组进程组集合。一个进程的会话成员关系是由其会话 ID 来定义的，会话 ID 是一个数字。新进程会继承其父进程的会话 ID。getsid()系统调用会返回 pid 指定的进程的会话 ID。

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

pid_t getsid(pid_t pid);
```

Returns session ID of specified process, or *(pid_t)* −1 on error

如果 pid 参数的值为 0，那么 getsid()会返回调用进程的会话 ID。

在一些 UNIX 实现中（如 HP-UX 11），只有当调用进程与 pid 指定的进程属于同一个会话时才能使用 getsid()来获取进程的会话 ID。（SUSv3 无此限制。）换句话说，只能通过这个调用的结果，即成功或失败（EPERM），来弄清楚指定进程与调用进程是否属于同一个会话。而在 Linux 和大多数其他实现中并不存在这一限制。

如果调用进程不是进程组首进程，那么 setsid()会创建一个新会话。

```
#include <unistd.h>

pid_t setsid(void);
```

Returns session ID of new session, or *(pid_t)* −1 on error

setsid()系统调用会按照下列步骤创建一个新会话。

- 调用进程成为新会话的首进程和该会话中新进程组的首进程。调用进程的进程组 ID 和会话 ID 会被设置成该进程的进程 ID。
- 调用进程没有控制终端。所有之前到控制终端的连接都会被断开。

如果调用进程是一个进程组首进程，那么 setsid()调用会报出 EPERM 错误。避免这个错误发生的最简单的方式是执行一个 fork()并让父进程终止以及让子进程调用 setsid()。由于子进程会继承其父进程的进程组 ID 并接收属于自己的唯一的进程 ID，因此它无法成为进程组首进程。

约束进程组首进程对 setsid()的调用是有必要的。因为如果没有这个约束的话，进程组组长就能够将其自身迁移至另一个（新的）会话中了，而该进程组的其他成员则仍然位于原来的会话中。（不会创建一个新进程组，因为根据定义，进程组首进程的进程组 ID 已经与其进程 ID 一样了。）

更多资源请访问稀酷客(www.ckook.com)

这会破坏会话和进程组之间严格的两级层次，因此一个进程组的所有成员必须属于同一个会话。

> 当使用 fork()创建一个新进程时，内核会确保它拥有一个唯一的进程 ID，并且该进程 ID 不会与任意已有进程的进程组 ID 和会话 ID 相同。这样，即使进程组或会话首进程退出之后，新进程也无法复用首进程的进程 ID，从而也无法成为既有会话和进程组的首进程。

程序清单 34-2 演示了使用 setsid()来创建一个新会话。为了检查该进程已经不再拥有控制终端了，这个程序尝试打开一个特殊文件/dev/tty（下一节将予以介绍）。当运行这个程序时会看到下面的结果：

```
$ ps -p $$ -o 'pid pgid sid command'          $$ is PID of shell
  PID  PGID   SID COMMAND
12243 12243 12243 bash                        PID, PGID, and SID of shell
$ ./t_setsid
$ PID=12352, PGID=12352, SID=12352
ERROR [ENXIO Device not configured] open /dev/tty
```

从输出中可以看出，进程成功地将其自身迁移至了新会话中的一个新进程组中。由于这个会话没有控制终端，因此 open()调用会失败。（从上面程序输出的倒数第二行中可以看出，hell 提示符与程序输出混杂在一起了，因为 shell 注意到父进程在 fork()调用之后就退出了，因此在子进程结束之前就输出了下一个提示符。）

程序清单 34-2　创建一个新会话

pgsjc/t_setsid.c

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (fork() != 0)            /* Exit if parent, or on error */
        _exit(EXIT_SUCCESS);

    if (setsid() == -1)
        errExit("setsid");

    printf("PID=%ld, PGID=%ld, SID=%ld\n", (long) getpid(),
            (long) getpgrp(), (long) getsid(0));

    if (open("/dev/tty", O_RDWR) == -1)
        errExit("open /dev/tty");
    exit(EXIT_SUCCESS);
}
```

pgsjc/t_setsid.c

34.4　控制终端和控制进程

一个会话中的所有进程可能会拥有一个（单个）控制终端。会话在被创建出来的时候是没有控制终端的，当会话首进程首次打开一个还没有成为某个会话的控制终端的终端时会建立控制终端，除非在调用 open()时指定 O_NOCTTY 标记。一个终端至多只能成为一个会话的控制终端。

更多资源请访问稀酷客(www.ckook.com)

SUSv3 定义了函数 tcgetsid(int fd)（在<termios.h>头文件中进行定义），它返回与由 fd 指定的控制终端相关联的会话的 ID。glibc 提供了这个函数（它是使用 ioctl() TIOCGSID 操作实现的）。

控制终端会被由 fork()创建的子进程继承并且在 exec()调用中得到保持。

当会话首进程打开了一个控制终端之后它同时也成为了该终端的控制进程。在发生终端断开之后，内核会向控制进程发送一个 SIGHUP 信号来通知这一事件的发生。在 34.6.2 节中将会介绍更多有关这一方面的细节信息。

如果一个进程拥有一个控制终端，那么打开特殊文件/dev/tty 就能够获取该终端的文件描述符。这对于一个程序在标准输入和输出被重定向之后需要确保自己确实在与控制终端进行通信是很有用的。如在 8.5 节中介绍的 getpass()函数会因此而打开/dev/tty。如果进程没有控制终端，那么在打开/dev/tty 时会报出 ENXIO 的错误。

删除进程与控制终端之间的关联关系

使用 ioctl(fd, TIOCNOTTY)操作能够删除进程与文件描述符 fd 指定的控制终端之间的关联关系。在调用这个函数之后再试图打开/dev/tty 文件的话就会失败。（尽管 SUSv3 没有指定这个操作，但大多数 UNIX 实现都支持 TIOCNOTTY 操作。）

如果调用进程是终端的控制进程，那么在控制进程终止时（参见 34.6.2）会发生下列事情。

1. 会话中的所有进程将会失去与控制终端之间的关联关系。
2. 控制终端失去了与该会话之间的关联关系，因此另一个会话首进程就能够获取该终端以成为控制进程。
3. 内核会向前台进程组的所有成员发送一个 SIGHUP 信号（和一个 SIGCONT 信号）来通知它们控制终端的丢失。

在 BSD 上建立一个控制终端

SUSv3 并不支持一个会话获取未指定的控制终端，在打开终端时仅指定 O_NOCTTY 标记的话只能确保该终端不会成为会话的控制终端。上面描述的 Linux 语义源自 System V 系统。

在 BSD 系统上，在会话首进程中打开一个终端不会导致该终端成为控制终端，不管是否指定了 O_NOCTTY 标记。相反，会话首进程需要使用 ioctl() TIOCSCTTY 操作来显式地将文件描述符 fd 指定的终端建立为控制终端。

```
if (ioctl(fd, TIOCSCTTY) == -1)
    errExit("ioctl");
```

只有在会话没有控制终端时才能执行这个操作。

Linux 系统上也有 TIOCSCTTY 操作，但在其他（非 BSD）实现中用得并不多。

获取表示控制终端的路径名：ctermid()

ctermid()函数返回表示控制终端的路径名。

```
#include <stdio.h>              /* Defines L_ctermid constant */

char *ctermid(char *ttyname);
```

Returns pointer to string containing pathname of controlling terminal, or NULL if pathname could not be determined

更多资源请访问稀酷客(www.ckook.com)

ctermid()函数以两种不同的方式返回控制终端的路径名：通过函数结果和通过 ttyname 指向的缓冲区。

如果 ttyname 不为 NULL，那么它是一个大小至少为 L_ctermid 字节的缓冲区，并且路径名会被复制进这个数组中。这里函数的返回值也是一个指向该缓冲区的指针。如果 ttyname 为 NULL，那么 ctermid()返回一个指向静态分配的缓冲区的指针，缓冲区中包含了路径名。当 ttyname 为 NULL 时，ctermid()是不可重入的。

在 Linux 和其他 UNIX 实现中，ctermid()通常会生成字符串/dev/tty。引入这个函数的目的是为了能更加容易地将程序移植到非 UNIX 系统上。

34.5 前台和后台进程组

控制终端保留了前台进程组的概念。在一个会话中，在同一时刻只有一个进程能成为前台进程，会话中的其他所有进程都是后台进程组。前台进程组是唯一能够自由地读取和写入控制终端的进程组。当在控制终端中输入其中一个信号生成终端字符之后，终端驱动器会将相应的信号发送给前台进程组的成员。34.7 节将会对此进行深入介绍。

> 从理论上来讲，可能会出现一个会话没有前台进程组的情况。如当前台进程组中的所有进程都终止并且没有其他进程注意到这个事实而将自己移动到前台时就会出现这种情况。但在实践中这种情况是比较少见的。通常 shell 进程会监控前台进程组的状态，当它注意到前台进程组结束之后（通过 wait()）会将自己移动到前台。

tcgetpgrp()和 tcsetpgrp()函数分别获取和修改一个终端的进程组。这些函数主要供任务控制 shell 使用。

```
#include <unistd.h>

pid_t tcgetpgrp(int fd);
                Returns process group ID of terminal's foreground process group,
                                                                or -1 on error

int tcsetpgrp(int fd, pid_t pgid);
                                        Returns 0 on success, or -1 on error
```

tcgetpgrp()函数返回文件描述符 fd 所指定的终端的前台进程组的进程组 ID，该终端必须是调用进程的控制终端。

> 如果这个终端没有前台进程组，那么 tcgetpgrp()返回一个大于 1 并且与所有既有进程组 ID 都不匹配的值。（SUSv3 规定了这种行为。）

tcsetpgrp()函数修改一个终端的前台进程组。如果调用进程拥有一个控制终端，那么文件描述符 fd 引用的就是那个终端，接着 tcsetpgrp()会将终端的前台进程组设置为 pgid 参数指定的进程组，该参数必须与调用进程所属的会话中的一个进程的进程组 ID 匹配。

tcgetpgrp() 和 tcsetpgrp()在 SUSv3 中都被标准化了。在 Linux 上，与很多其他 UNIX 实现一样，这些函数是通过两个非标准的 ioctl()操作来实现的，即 TIOCGPGRP 和 TIOCSPGRP。

更多资源请访问稀酷客(www.ckook.com)

34.6 SIGHUP 信号

当一个控制进程失去其终端连接之后，内核会向其发送一个 SIGHUP 信号来通知它这一事实。（还会发送一个 SIGCONT 信号以确保当该进程之前被一个信号停止时重新开始该进程。）一般来讲，这种情况可能会在下面两个场景中出现。

- 当终端驱动器检测到连接断开后，表明调制解调器或终端行上信号的丢失。
- 当工作站上的终端窗口被关闭时。发生这种情况是因为最近打开的与终端窗口关联的伪终端的主侧的文件描述符被关闭了。

SIGHUP 信号的默认处理方式是终止进程。如果控制进程处理了或忽略了这个信号，那么后续尝试从终端中读取数据的请求就会返回文件结束的错误。

> SUSv3 声称如果终端断开发生的同时还满足调用 read()时抛出 EIO 错误的条件的话，那么调用 read()既有可能返回文件结束，也有可能返回 EIO 错误。可移植的程序必须要处理好这两种情况。在 34.7.2 节和 34.7.4 节中将介绍在哪些情况下调用 read()会发生 EIO 错误。

向控制进程发送 SIGHUP 信号会引起一种链式反应，从而导致将 SIGHUP 信号发送给很多其他进程。这个过程可能会以下列两种方式发生。

- 控制进程通常是一个 shell。shell 建立了一个 SIGHUP 信号的处理器，这样在进程终止之前，它能够将 SIGHUP 信号发送给由它所创建的各个任务。在默认情况下，这个信号会终止那些任务，但如果它们捕获了这个信号，就能知道 shell 进程已经终止了。
- 在终止终端的控制进程时，内核会解除会话中所有进程与该控制终端之间的关联关系以及控制终端与该会话的关联关系（因此另一个会话首进程可以请求该终端成为控制终端了），并且通过向该终端的前台进程组的成员发送 SIGHUP 信号来通知它们控制终端的丢失。

下一节将深入介绍这两种方式的细节信息。

> SIGHUP 信号也可以用作他用。在 34.7.4 节中可以看到当一个进程组成为孤儿进程组时会生成 SIGHUP 信号。此外，手工发送 SIGHUP 信号通常用来触发 daemon 进程重新初始化自身或重新读取其配置文件。（根据定义，daemon 进程没有控制终端，因此无法从内核接收 SIGHUP 信号。）37.4 节将会介绍如何配合使用 SIGHUP 信号和 daemon 进程。

34.6.1 在 shell 中处理 SIGHUP 信号

在登录会话中，shell 通常是终端的控制进程。大多数 shell 程序在交互式运行时会为 SIGHUP 信号建立一个处理器。这个处理器会终止 shell，但在终止之前会向由 shell 创建的各个进程组（包括前台和后台进程组）发送一个 SIGHUP 信号。（在 SIGHUP 信号之后可能会发送一个 SIGCONT 信号，这依赖于 shell 本身以及任务当前是否处于停止状态。）至于这些组中的进程如何响应 SIGHUP 信号则需要根据应用程序的具体需求，如果不采取特殊的动作，那么默认情况下将会终止进程。

更多资源请访问稀酷客(www.ckook.com)

一些任务控制 shell 在正常退出（如登出或在 shell 窗口中接下 Control-D）时也会发送 SIGHUP 信号来停止后台任务。bash 和 Korn shell 都采取了这种处理方式（在首次登出尝试时打印出一条消息之后）。

nohup(1)命令可以用来使一个命令对 SIGHUP 信号免疫——即执行命令时将 SIGHUP 信号的处理设置为 SIG_IGN。bash 内置的命令 disown 提供了类似的功能，它从 shell 的任务列表中删除一个任务，这样在 shell 终止时就不会向该任务发送 SIGHUP 信号了。

程序清单 34-3 演示了 shell 接收 SIGHUP 信号并向其创建的任务发送 SIGHUP 信号的过程。这个程序的主要任务是创建一个子进程，然后让父进程和子进程暂停执行以捕获 SIGHUP 信号并在收到该信号时打印一条消息。如果在执行程序时使用了一个可选的命令行参数（它可以是任意字符串），那么子进程会将其自身放置在一个不同的进程组中（在同一个会话中）。这个功能对于说明 shell 不会向不是由它创建的进程组发送 SIGHUP 信号，即使该进程组与 shell 位于同一个会话中来讲是非常有用的。（由于程序中最后一个 for 循环是一个无限循环，因此这个程序使用了 alarm()设置一个定时器来发送 SIGALRM 信号。如果一个进程没有终止的话，那么当它接收到 SIGALRM 信号而不做处理时会导致进程终止。）

程序清单 34-3：捕获 SIGHUP 信号

pgsjc/catch_SIGHUP.c

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void
handler(int sig)
{
}
int
main(int argc, char *argv[])
{
    pid_t childPid;
    struct sigaction sa;

    setbuf(stdout, NULL);       /* Make stdout unbuffered */

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGHUP, &sa, NULL) == -1)
        errExit("sigaction");

    childPid = fork();
    if (childPid == -1)
        errExit("fork");

    if (childPid == 0 && argc > 1)
        if (setpgid(0, 0) == -1)        /* Move to new process group */
            errExit("setpgid");

    printf("PID=%ld; PPID=%ld; PGID=%ld; SID=%ld\n", (long) getpid(),
            (long) getppid(), (long) getpgrp(), (long) getsid(0));

    alarm(60);                  /* An unhandled SIGALRM ensures this process
```


更多资源请访问稀酷客(www.ckook.com)

```
                                    will die if nothing else terminates it */
    for(;;) {                    /* Wait for signals */
        pause();
        printf("%ld: caught SIGHUP\n", (long) getpid());
    }
}
```

pgsjc/catch_SIGHUP.c

假设在一个终端窗口中输入了下面的命令来运行程序清单 34-3 中的程序的两个实例，接着关闭终端窗口。

```
$ echo $$                                    PID of shell is ID of session
5533
$ ./catch_SIGHUP > samegroup.log 2>&1 &
$ ./catch_SIGHUP x > diffgroup.log 2>&1
```

第一个命令会导致创建两个进程，这两个进程属于由 shell 创建的进程组。第二个命令创建了一个子进程，子进程将自身放置在了一个不同的进程组中。

当查看 samegroup.log 时会发现其中包含了下面的输出，表明两个进程组的成员都收到了 shell 发送的信号。

```
$ cat samegroup.log
PID=5612; PPID=5611; PGID=5611; SID=5533     Child
PID=5611; PPID=5533; PGID=5611; SID=5533     Parent
5611: caught SIGHUP
5612: caught SIGHUP
```

当查看 diffgroup.log 时会发现下面的输出，表明 shell 在收到 SIGHUP 时不会向不是由它创建的进程组发送信号。

```
$ cat diffgroup.log
PID=5614; PPID=5613; PGID=5614; SID=5533     Child
PID=5613; PPID=5533; PGID=5613; SID=5533     Parent
5613: caught SIGHUP                          Parent was signaled, but not child
```

34.6.2 SIGHUP 和控制进程的终止

如果因为终端断开引起的向控制进程发送的 SIGHUP 信号会导致控制进程终止，那么 SIGHUP 信号会被发送给终端的前台进程组中的所有成员（见 25.2 节）。这个行为是控制进程终止的结果，而不是专门与 SIGHUP 信号关联的行为。如果控制进程出于任何原因终止，那么前台进程组就会收到 SIGHUP 信号。

在 Linux 上，SIGHUP 信号后面会跟着一个 SIGCONT 信号以确保在进程组之前被一个信号停止的情况下恢复该进程组。但 SUSv3 并没有指定这种行为，并且在这种情况下大多数其他 UNIX 实现不会发送 SIGCONT 信号。

程序清单 34-4 演示了控制进程的终止导致向终端的前台进程组的所有成员发送 SIGHUP 信号。这个程序为每个命令行参数都创建了一个子进程②。如果相应的命令行参数是 d，那么子进程会将自身放置在自己的（不同的）进程组中③；否则的话子进程加入到父进程所在的进程组中。（这里使用了字母 s 来指定后面这种处理方式，尽管可以使用除 d 之外的任意字母。）接着各个子进程设置了 SIGHUP 信号处理器④。为确保它们能够在进程终止事件不发生的情况下正常终止，父进程和子进程都调用了 alarm()设置一个定时器以在 60 秒之后发送一个 SIGALRM 信号⑤。最后所有进程（包括父进程）打印出了它们的进程 ID 和进程组 ID⑥，接着循环等待信号的到达⑦。当发出信号之后，处理器会打印出进程的进程 ID 和信号数值①。

更多资源请访问稀酷客(www.ckook.com)

程序清单 34-4　在终端断开发生时捕获 SIGHUP 信号

pgsjc/disc_SIGHUP.c

```
#define _GNU_SOURCE     /* Get strsignal() declaration from <string.h> */
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void             /* Handler for SIGHUP */
handler(int sig)
{
①   printf("PID %ld: caught signal %2d (%s)\n", (long) getpid(),
            sig, strsignal(sig));
                        /* UNSAFE (see Section 21.1.2) */
}
int
main(int argc, char *argv[])
{
    pid_t parentPid, childPid;
    int j;
    struct sigaction sa;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s {d|s}... [ > sig.log 2>&1 ]\n", argv[0]);

    setbuf(stdout, NULL);               /* Make stdout unbuffered */

    parentPid = getpid();
    printf("PID of parent process is:       %ld\n", (long) parentPid);
    printf("Foreground process group ID is: %ld\n",
            (long) tcgetpgrp(STDIN_FILENO));

②   for (j = 1; j < argc; j++) {        /* Create child processes */
        childPid = fork();
        if (childPid == -1)
            errExit("fork");

        if (childPid == 0) {            /* If child... */
③           if (argv[j][0] == 'd')     /* 'd' --> to different pgrp */
                if (setpgid(0, 0) == -1)
                    errExit("setpgid");

            sigemptyset(&sa.sa_mask);
            sa.sa_flags = 0;
            sa.sa_handler = handler;
④           if (sigaction(SIGHUP, &sa, NULL) == -1)
                errExit("sigaction");
            break;                      /* Child exits loop */
        }
    }

    /* All processes fall through to here */

⑤   alarm(60);          /* Ensure each process eventually terminates */

⑥   printf("PID=%ld PGID=%ld\n", (long) getpid(), (long) getpgrp());
    for (;;)
⑦       pause();        /* Wait for signals */
}
```

pgsjc/disc_SIGHUP.c

假设使用下面的命令在一个终端窗口中运行了程序清单 34-4 中的程序。

```
$ exec ./disc_SIGHUP d s s > sig.log 2>&1
```

exec 命令时一个 shell 内置命令，它会导致 shell 执行一个 exec()来使用指定的程序取代自己。由于 shell 是终端的控制进程，因此现在这个程序已经成为了控制进程并且在终端窗口被关闭时会收到 SIGHUP 信号。在关闭终端窗口之后，在 sig.log 文件中会看到下面的输出。

```
PID of parent process is:       12733
Foreground process group ID is: 12733
PID=12755 PGID=12755                    First child is in a different process group
PID=12756 PGID=12733                    Remaining children are in same PG as parent
PID=12757 PGID=12733
PID=12733 PGID=12733                    This is the parent process
PID 12756: caught signal  1 (Hangup)
PID 12757: caught signal  1 (Hangup)
```

关闭终端窗口会导致 SIGHUP 信号被发送给控制进程（父进程），进而导致该进程的终止。从上面可以看出，两个子进程与父进程位于同一个进程组中（终端的前台进程组），它们都收到了 SIGHUP 信号，但位于另一个进程组（后台）中的子进程并没有收到这个信号。

34.7 作业控制

作业控制是在 1980 年左右由 BSD 系统上的 C shell 首次推出的特性。作业控制允许一个 shell 用户同时执行多个命令（作业），其中一个命令在前台运行，其余的命令在后台运行。作业可以被停止和恢复以及在前后台之间移动，下面会对此予以详细介绍。

在初始的 POSIX.1 标准中，对作业的支持是可选的。后面的 UNIX 标准使这个功能成为了必备功能。

在基于字符的哑终端盛行的年代（物理终端设备只能显示 ASCII 字符），很多 shell 用户都知道如何使用 shell 作业控制命令。在运行 X Window System 的位图显示器出现之后，熟悉 shell 作业控制的人就越来越少了，但作业控制仍然是一项非常有用的特性。使用作业控制管理多个同时执行的命令比在几个窗口之间来回切换更快速和简单。对于那些不熟悉作业控制的读者来讲，可以看一下下面这个简短的入门指南。在介绍完入门指南之后将会介绍作业控制实现方面的细节信息并考虑作业控制对应用程序设计的约束。

34.7.1 在 shell 中使用作业控制

当输入的命令以&符号结束时，该命令会作为后台任务运行，如下面的示例所示。

```
$ grep -r SIGHUP /usr/src/linux >x &
[1] 18932                               Job 1: process running grep has PID 18932
$ sleep 60 &
[2] 18934                               Job 2: process running sleep has PID 18934
```

shell 会为后台的每个进程赋一个唯一的作业号。当作业在后台运行之后以及在使用各种作业控制命令操作或监控作业时作业号会显示在方括号中。作业号后面的数字是执行这个命令的进程的进程 ID 或管道中最后一个进程的进程 ID。在后面几个段落中介绍的命令中会使用%num 来引用作业，其中 num 是 shell 赋给作业的作业号。

在很多情况下是可以省略%num 的，当省略%num 时默认指当前作业。当前作业是在前

更多资源请访问稀酷客(www.ckook.com)

台最新被停止的作业（使用下面介绍的挂起字符）或者如果没有这样的作业的话，最新作业是在后台启动的任务。（不同 shell 确定哪个后台作业为当前作业的细节方面稍微有些不同。）另外，%%和%+符号指的是当前作业，%-符号指的是上一个当前作业。在 jobs 命令的输出中，当前的和上一个当前作业分别用加号（+）和减号（-）标记，稍后就会对此予以介绍。

jobs 是 shell 内置的一个命令，它会列出所有后台作业。

```
$ jobs
[1]- Running          grep -r SIGHUP /usr/src/linux >x &
[2]+ Running          sleep 60 &
```

在这个时刻，shell 是终端的前台进程。由于仅有一个前台进程能够从控制终端读取输入和接收终端生成的信号，因此有时候需要将后台作业移动到前台。这是通过 fg 这个 shell 内置命令来完成的。

```
$ fg %1
grep -r SIGHUP /usr/src/linux >x
```

从上面的示例中可以看出，当在前后台之间移动作业时 shell 会重新打印出该作业的命令行。读者通过阅读下面的内容就会发现，当作业在后台的状态发生变化时，shell 也会重新打印该作业的命令行。

当作业在前台运行时可以使用终端挂起字符（通常是 Control-Z）来挂起作业，它会向终端的前台进程组发送一个 SIGTSTP 信号。

```
Type Control-Z
[1]+ Stopped          grep -r SIGHUP /usr/src/linux >x
```

在按下 Control-Z 之后，shell 会打印出在后台被停止的命令。如果需要的话，可以使用 fg 命令在前台恢复这个作业或使用 bg 命令在后台恢复这个命令。不管使用哪个命令恢复作业，shell 都会通过向任务发送一个 SIGCONT 信号来恢复被停止的作业。

```
$ bg %1
[1]+ grep -r SIGHUP /usr/src/linux >x &
```

通过向后台作业发送一个 SIGSTOP 信号能够停止该后台作业。

```
$ kill -STOP %1
[1]+ Stopped          grep -r SIGHUP /usr/src/linux >x
$ jobs
[1]+ Stopped          grep -r SIGHUP /usr/src/linux >x
[2]- Running          sleep 60 &
$ bg %1                                   Restart job in background
[1]+ grep -r SIGHUP /usr/src/linux >x &
```

Korn 和 C shell 提供了一个命令 stop 作为 kill-stop 快捷方式。

当后台作业最后执行结束之后，shell 会在打印下一个 shell 提示符之前先打印一条消息。

```
Press Enter to see a further shell prompt
[1]- Done             grep -r SIGHUP /usr/src/linux >x
[2]+ Done             sleep 60
$
```

只有前台作业中的进程才能够从控制终端中读取输入。这个限制条件避免了多个作业竞争读取终端输入。如果后台作业尝试从终端中读取输入，就会接收到一个 SIGTTIN 信号。SIGTTIN 信号的默认处理动作是停止作业。

在上一个例子以及后面的几个例子中可能不需要按下回车键就能看到作业状态变更信息。根据内核的调度决策，shell 可能会在打印下一个 shell 提示符之前接收到有关后台作业

更多资源请访问稀酷客(www.ckook.com)

状态变更的通知。

现在必须要将作业移到前台来（fg）并向其提供所需的输入了。如果需要的话，可以通过先挂起该作业后在后台恢复该作业（bg）的方式继续该作业的执行。（当然，在这个特定的例子中，cat 将会再次立即被停止，因为它会再次尝试从终端中读取输入。）

在默认情况下，后台作业是被允许向控制终端输入内容的。但如果终端设置了 TOSTOP 标记（终端输出停止，参见 62.5 节），那么当后台作业尝试在终端上输出时会导致 SIGTTOU 信号的产生。（使用 stty 命令能够设置 TOSTOP 标志，62.3 节将会对此予以介绍。）与 SIGTTIN 信号一样，SIGTTOU 信号会停止作业。

```
$ stty tostop                    Enable TOSTOP flag for this terminal
$ date &
[1] 19023
$
Press Enter once more to see job state changes displayed prior to next shell prompt
[1]+ Stopped          date
```

可以通过将作业移到前台来查看作业的输出。

```
$ fg
date
Tue Dec 28 16:20:51 CEST 2010
```

作业具备多种状态，作业控制以及 shell 命令和终端字符（以及相应的信号）可以使作业在不同状态之间迁移，图 34-2 对作业的状态进行了总结。这些作业可以通过向作业发送各种信号来到达，如 SIGINT 和 SIGQUIT 信号，而这些信号可以通过键盘来生成。

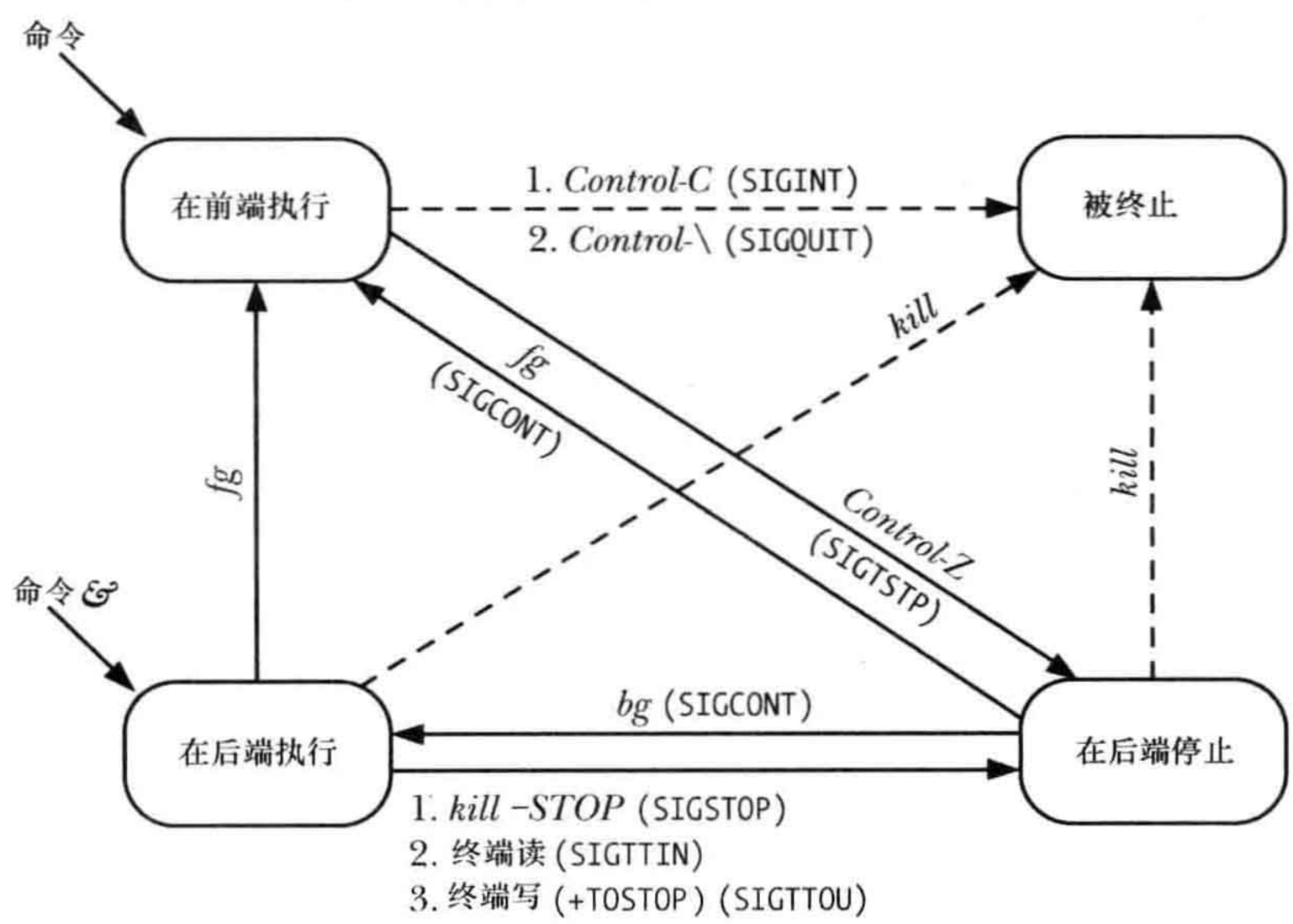


图 34-2　作业控制状态

34.7.2　实现作业控制

本节将先介绍与实现作业控制有关的各个方面，最后介绍一个能使作业控制操作更加透明的示例程序。

尽管作业控制一开始在 POSIX.1 标准中是可选的，但在后面的标准中，包括 SUSv3，则要求实现必须要支持作业控制。这种支持所需的条件如下。

- 实现必须要提供特定的作业控制信号：SIGTSTP、SIGSTOP、SIGCONT、SIGTTOU 以及 SIGTTIN。此外，SIGCHLD 信号（参见 26.3 节）也是必需的，因为它允许 shell（所有任务的父进程）找出其子进程何时执行终止或被停止了。
- 终端驱动器必须要支持作业控制信号的生成，这样当输入特定的字符或进行终端 I/O 以及在后台作业中执行特定的其他终端操作（下面将会予以介绍）时需要将恰当的信号（如图 34-2 所示）发送到相关的进程组。为了能够完成这些动作，终端驱动器必须要记录与终端相关联的会话 ID（控制进程）和前台进程组 ID（图 34-1）。
- shell 必须要支持作业控制（大多数现代 shell 都具备这个功能）。这种支持是通过前面介绍的将作业在前台和后台之间迁移以及监控作业的状态的命令的形式来完成的。其中某些命令会向作业发送信号（如图 34-2 所示）。此外，在执行将作业从前台运行的状态迁移至其他状态的操作中，shell 使用 tcsetpgrp()调用来调整终端驱动器中与前台进程组有关的记录信息。

在 20.5 节中曾经讲过，信号一般只有在发送进程的真实或有效用户 ID 与接收进程的真实用户 ID 或保存的 set-user-ID 匹配时才会被发送给进程，但 SIGCONT 是这个规则的一个例外。内核允许一个进程（如 shell）向同一会话中的任意进程发送 SIGCONT 信号，不管进程的验证信息是什么。在 SIGCONT 信号上放宽这个规则是有必要的，这样当用户开始一个会修改自身的验证信息（特别是真实的用户 ID）的 set-user-ID 程序时，仍然能够在程序被停止时通过 SIGCONT 信号来恢复这个程序的运行。

SIGTTIN 和 SIGTTOU 信号

SUSv3 对后台进程的 SIGTTIN 和 SIGTTOU 信号的产生规定了一些特殊情况（Linux 实现了这些规定）。

- 当进程当前处于阻塞状态或忽视 SIGTTIN 信号的状态时则不发送 SIGTTIN 信号，这时试图从控制终端发起 read()调用会失败，errno 会被设置成 EIO。这种行为的逻辑是没有这种行为的话进程就无法知道不允许进行 read()操作。
- 即使终端被设置了 TOSTOP 标记，当进程当前处于阻塞状态或忽视 SIGTTIN 信号的状态时也不发送 SIGTTOU 信号。这时从控制终端发起 write()调用是允许的（即 TOSTOP 标记被忽视了）。
- 不管是否设置了 TOSTOP 标记，当后台进程试图在控制终端上调用会修改终端驱动器数据结构的特定函数时会生成 SIGTTOU 信号。这些函数包括 tcsetpgrp()、tcsetattr()、tcflush()、tcflow()、tcsendbreak()以及 tcdrain()。（第 62 章将会介绍这些函数。）如果 SIGTTOU 信号被阻塞或被忽视了，那么这些调用就会成功。

示例程序：演示作业控制的操作

通过程序清单 34-5 给出的程序能够看出 shell 是如何将命令以管道连接的形式组织进一个作业的（进程组）。此外，通过这个程序还能监控发送的特定信号以及在作业控制中对终端的前台进程组设置所做的变更。读者可以以管道的形式运行这个程序的多个实例，如下面的例子所示。

```
$ ./job_mon | ./job_mon | ./job_mon
```

程序清单 34-5 中的程序执行了下面的操作。

- 在启动的时候，程序为 SIGINT、SIGTSTP 和 SIGTSTP 信号④安装了一个处理器，该

更多资源请访问稀酷客(www.ckook.com)

处理器执行下面的动作。

-显示终端的前台进程组①。为避免在输出中出现多行相同的内容，只有进程组首进程才能执行这个动作。

-显示进程的 ID、进程在管道中的位置以及接收到的信号②。

-当处理器捕获到 SIGTSTP 信号时必须要做一些额外的处理工作，因为捕获到这个信号不会停止进程。这样要停止进程的话处理器就需要发出一个 SIGSTOP 信号③，因为这个信号总是会停止进程的执行。（在 34.7.3 节中将会优化 SIGTSTP 信号的处理方式。）

- 如果程序是管道中的第一个进程，那么它就会打印出所有进程的输出的标题⑥。为了检测进程本身是否是管道中的第一个进程（或最后一个进程），程序使用了 isatty()函数（62.10 节中将会予以介绍）来检查其标准输入（或输出）是否是一个终端⑤。如果指定的文件描述符是一个管道，那么 isatty()返回 false（0）。
- 程序构建了一个消息并将消息传递给了管道中的下一个命令。这个消息是一个表明进程在管道中的位置的整数。因此，对于第一个进程来讲，消息中包含数字 1。如果程序是管道中的第一个进程，那么消息被初始化为 0。如果程序不是管道中的第一个进程，那么程序首先会从其前面的进程中读取这个消息⑦。程序在将控制权传递给下一个进程之前会递增消息值⑧。
- 不管程序在管道中所处的位置如何，它都会输出一行包含其在管道中的位置、进程 ID、父进程 ID、进程组 ID 以及会话 ID 的文本⑨。
- 除非程序是管道中的最后一个命令，否则就会写入一个整数消息以将其传递给管道中的下一个命令。
- 最后，程序会无限循环并使用 pause()等待信号⑪。

程序清单 34-5：观察作业控制中的进程处理

pgsjc/job_mon.c

```
#define _GNU_SOURCE     /* Get declaration of strsignal() from <string.h> */
#include <string.h>
#include <signal.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

static int cmdNum;              /* Our position in pipeline */

static void                     /* Handler for various signals */
handler(int sig)
{
    /* UNSAFE: This handler uses non-async-signal-safe functions
       (fprintf(), strsignal(); see Section 21.1.2) */
①     if (getpid() == getpgrp())          /* If process group leader */
          fprintf(stderr, "Terminal FG process group: %ld\n",
                  (long) tcgetpgrp(STDERR_FILENO));
②     fprintf(stderr, "Process %ld (%d) received signal %d (%s)\n",
                  (long) getpid(), cmdNum, sig, strsignal(sig));
      /* If we catch SIGTSTP, it won't actually stop us. Therefore we
         raise SIGSTOP so we actually get stopped. */

③     if (sig == SIGTSTP)
          raise(SIGSTOP);
}

int
```

```
main(int argc, char *argv[])
{
    struct sigaction sa;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = handler;
④   if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");
    if (sigaction(SIGTSTP, &sa, NULL) == -1)
        errExit("sigaction");
    if (sigaction(SIGCONT, &sa, NULL) == -1)
        errExit("sigaction");

    /* If stdin is a terminal, this is the first process in pipeline:
       print a heading and initialize message to be sent down pipe */

⑤   if (isatty(STDIN_FILENO)) {
        fprintf(stderr, "Terminal FG process group: %ld\n",
                (long) tcgetpgrp(STDIN_FILENO));
⑥       fprintf(stderr, "Command   PID  PPID  PGRP   SID\n");
        cmdNum = 0;

    } else {            /* Not first in pipeline, so read message from pipe */
⑦       if (read(STDIN_FILENO, &cmdNum, sizeof(cmdNum)) <= 0)
            fatal("read got EOF or error");
    }

⑧   cmdNum++;
⑨   fprintf(stderr, "%4d    %5ld %5ld %5ld %5ld\n", cmdNum,
            (long) getpid(), (long) getppid(),
            (long) getpgrp(), (long) getsid(0));

    /* If not the last process, pass a message to the next process */

    if (!isatty(STDOUT_FILENO))   /* If not tty, then should be pipe */
⑩       if (write(STDOUT_FILENO, &cmdNum, sizeof(cmdNum)) == -1)
            errMsg("write");

⑪   for(;;)             /* Wait for signals */
        pause();
}
```

pgsjc/job_mon.c

下面的 shell 会话演示了程序清单 34-5 中的程序的用法。它首先打印出了 shell 的进程 ID（它是会话首进程和进程组首进程，尽管它是进程组中的唯一成员），接着创建了一个包含两个进程的后台作业。

从上面的输出可以看出，shell 仍然是终端的前台进程，并且新作业与 shell 位于同一个会话中，所有进程都位于同一个进程组中。从进程 ID 可以看出，作业中进程的创建顺序与命令在命令行中出现的顺序是一致的。（大多数 shell 是这样处理的，但有些 shell 实现创建进程的顺序与命令在命令行中出现的顺序不一致。）

```
$ echo $$                                  Show PID of the shell
1204
$ ./job_mon | ./job_mon &                  Start a job containing 2 processes
[1] 1227
Terminal FG process group: 1204
Command   PID  PPID  PGRP   SID
   1     1226  1204  1226  1204
   2     1227  1204  1226  1204
```

下面继续创建第二个包含三个进程的后台作业。

```
$ ./job_mon | ./job_mon | ./job_mon &
[2] 1230
Terminal FG process group: 1204
Command   PID  PPID  PGRP   SID
   1     1228  1204  1228  1204
   2     1229  1204  1228  1204
   3     1230  1204  1228  1204
```

从上面可以看出，shell 仍然是终端的前台进程组，新任务中的进程与 shell 位于同一个会话中，但所处的进程组则与第一个任务中的进程所处的进程组不同。下面将第二个任务迁移至前台并向其发送一个 SIGINT 信号。

```
$ fg
./job_mon | ./job_mon | ./job_mon
Type Control-C to generate SIGINT (signal 2)
Process 1230 (3) received signal 2 (Interrupt)
Process 1229 (2) received signal 2 (Interrupt)
Terminal FG process group: 1228
Process 1228 (1) received signal 2 (Interrupt)
```

从上面的输出可以看出，SIGINT 信号被发送给了前台进程组中的所有进程，并且这个作业现在已经成为了终端的前台进程组。接着向这个作业发送一个 SIGTSTP 信号。

```
Type Control-Z to generate SIGTSTP (signal 20 on Linux/x86-32).
Process 1230 (3) received signal 20 (Stopped)
Process 1229 (2) received signal 20 (Stopped)
Terminal FG process group: 1228

Process 1228 (1) received signal 20 (Stopped)

[2]+  Stopped        ./job_mon | ./job_mon | ./job_mon
```

现在进程组中的所有成员都被停止了。从输出中可以看出进程组 1228 是前台作业，但当这个作业被停止之后，shell 变成了前台进程组，虽然这一点无法从输出中看出。

接着使用 bg 命令重新开始这个作业，该命令会向作业中的进程发送一个 SIGCONT 信号。

```
$ bg                                          Resume job in background
[2]+ ./job_mon | ./job_mon | ./job_mon &
Process 1230 (3) received signal 18 (Continued)
Process 1229 (2) received signal 18 (Continued)
Terminal FG process group: 1204               The shell is in the foreground
Process 1228 (1) received signal 18 (Continued)
$ kill %1 %2                                  We've finished: clean up
[1]-  Terminated     ./job_mon | ./job_mon
[2]+  Terminated     ./job_mon | ./job_mon | ./job_mon
```

34.7.3 处理作业控制信号

由于对于大多数应用程序来讲作业控制的操作是透明的，因此它们无需对作业控制信号采取特殊的动作，但像 vi 和 less 之类的进行屏幕处理的程序则是例外，因为它们需要控制文本在终端上的布局和修改各种终端设置，包括允许在某一时刻从终端输入中读取一个字符（不是一行）的设置。（第 62 章将会介绍各种终端设置。）

屏幕处理程序需要处理终端停止信号（SIGTSTP）。信号处理器应该将终端重置为规范（每次一行）输入模式并将光标放在终端的左下角。当进程恢复之后，程序会将终端设置回所需的模式，检查终端窗口大小（窗口大小同时可能会被用户改掉）以及使用所需的内容重新绘制屏幕。

更多资源请访问稀酷客(www.ckook.com)

当挂起或退出诸如 vi、xterm 或其他终端处理程序时通常会看到程序使用启动之前的可见文本来绘制终端。这些终端处理程序是通过捕获两个字符序列来取得这种效果的，所有使用 terminfo 或 termcap 包的程序在取得和释放终端布局的控制时都需要输出这两个字符序列。第一个字符序列称为 smcup（通常是 Escape 后面跟着[?1049h），它会导致终端处理程序切换至其“预备”屏幕。第二个序列称为 rmcup（通常是 Escape 后面跟着[?1049l），它会导致终端处理程序恢复到默认屏幕，从而导致在显示器上重现屏幕处理程序在获取终端的控制权之前的初始文本。

在处理 SIGTSTP 信号时需要清楚一些细节问题。第一个问题是在 34.7.2 节中提及过的：如果 SIGTSTP 信号被捕获了，那么就不会执行默认的停止进程的动作。在程序清单 34-5 中是通过让 SIGTSTP 信号的处理器生成一个 SIGSTOP 信号来解决这个问题的。由于 SIGSTOP 信号是无法被捕获、阻塞和忽略的，因此能确保立即停止进程，但这种方式不是非常准确。在 26.1.3 节中曾经介绍过父进程可以使用 wait()或 waitpid()返回的等待状态值来确定哪个信号导致了其子进程的停止。如果在 SIGTSTP 信号处理器中生成了 SIGSTOP 信号，那么对于父进程来讲，其子进程是被 SIGSTOP 信号停止的，这就会产生误导。

在这种情况下，恰当的处理方式是让 SIGTSTP 信号处理器再生成一个 SIGTSTP 信号来停止进程，如下所示。

1. 处理器将 SIGTSTP 信号的处理重置为默认值（SIG_DFL）。
2. 处理器生成 SIGTSTP 信号。
3. 由于 SIGTSTP 信号会被阻塞进入处理器（除非指定了 SA_NODEFER 标记），因此处理器会接触该信号的阻塞。这时，在上一个步骤中生成的 SIGTSTP 信号会导致默认动作的执行：进程会立即被挂起。
4. 在后面的某个时刻，当进程接收到 SIGCONT 信号时会恢复。这时，处理器的执行就会继续。
5. 在返回之前，处理器会重新阻塞 SIGTSTP 信号并重新注册本身来处理下一个 SIGTSTP 信号。

执行重新阻塞 SIGTSTP 信号这一步是因为防止在处理器重新注册本身之后和返回之前接收到另一个 SIGTSTP 信号而导致处理器被递归调用的情况。在 22.7 节中曾经提及过在快速发送信号时递归调用一个信号处理器会导致栈溢出。阻塞信号还避免了信号处理器在重新注册本身和返回之前需要执行其他动作（如保存和还原全局变量）时存在的问题。

示例程序

程序清单 34-6 中的处理器实现了上面描述的步骤，从而能够正确地处理 SIGTSTP。（在程序清单 62-4 中给出了另一个处理 SIGTSTP 信号的例子）。在注册了 SIGTSTP 信号处理器之后，这个程序的 main()函数开始循环等待信号。下面是运行这个程序之后的输出。

```
$ ./handling_SIGTSTP
Type Control-Z, sending SIGTSTP
Caught SIGTSTP                    This message is printed by SIGTSTP handler

[1]+  Stopped        ./handling_SIGTSTP
$ fg                              Sends SIGCONT
./handling_SIGTSTP
Exiting SIGTSTP handler           Execution of handler continues; handler returns
Main                              pause() call in main() was interrupted by handler
Type Control-C to terminate the program
```

在诸如 vi 之类的屏幕处理程序中，程序清单 34-6 中的信号处理器中的 printf()调用将会被前面概括过的能修改终端模式并重新绘制终端显示的程序所取代。（由于需要避免调用非同步信号

更多资源请访问棒酷客(www.ckook.com)

安全函数，参见 21.1.2 节，处理器应该通过设置一个标记通知主程序重新绘制屏幕。）

注意 SIGTSTP 处理器可能会中断特定的阻塞式系统调用（如 21.5 节中描述的那样）。从上面执行程序的输出中也可以看出这一点，在 pause()调用被中断之后，主程序打印出了消息 Main。

程序清单 34-6：处理 SIGTSTP

pgsjc/handling_SIGTSTP.c

```
#include <signal.h>
#include "tlpi_hdr.h"

static void                             /* Handler for SIGTSTP */
tstpHandler(int sig)
{
    sigset_t tstpMask, prevMask;
    int savedErrno;
    struct sigaction sa;

    savedErrno = errno;                 /* In case we change 'errno' here */

    printf("Caught SIGTSTP\n");         /* UNSAFE (see Section 21.1.2) */

    if (signal(SIGTSTP, SIG_DFL) == SIG_ERR)
        errExit("signal");              /* Set handling to default */

    raise(SIGTSTP);                     /* Generate a further SIGTSTP */

    /* Unblock SIGTSTP; the pending SIGTSTP immediately suspends the program */
    sigemptyset(&tstpMask);
    sigaddset(&tstpMask, SIGTSTP);
    if (sigprocmask(SIG_UNBLOCK, &tstpMask, &prevMask) == -1)
        errExit("sigprocmask");

    /* Execution resumes here after SIGCONT */

    if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
        errExit("sigprocmask");         /* Reblock SIGTSTP */

    sigemptyset(&sa.sa_mask);           /* Reestablish handler */
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = tstpHandler;
    if (sigaction(SIGTSTP, &sa, NULL) == -1)
        errExit("sigaction");

    printf("Exiting SIGTSTP handler\n");
    errno = savedErrno;
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    /* Only establish handler for SIGTSTP if it is not being ignored */

    if (sigaction(SIGTSTP, NULL, &sa) == -1)
        errExit("sigaction");

    if (sa.sa_handler != SIG_IGN) {
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = SA_RESTART;
```


更多资源请访问稀酷客(www.ckook.com)

```
        sa.sa_handler = tstpHandler;
        if (sigaction(SIGTSTP, &sa, NULL) == -1)
            errExit("sigaction");
    }
    for (;;) {                          /* Wait for signals */
        pause();
        printf("Main\n");
    }
}
```

pgsjc/handling_SIGTSTP.c

处理被忽略的任务控制和终端生成的信号

程序清单 34-6 中给出的程序只有在 SIGTSTP 信号不被忽略的情况下才会为该信号建立一个信号处理器。这里其实是遵循了一个常规规则，即应用程序应该在作业控制和终端生成信号不被忽略的时候才处理这些信号。对于作业控制信号（SIGTSTP、SIGTTIN 以及 SIGTTOU）来讲，这个规则防止应用程序试图处理那些从非作业控制 shell（如传统的 Bourne shell）发出的信号。在非作业控制 shell 中，这些信号的处理被设置成了 SIG_IGN，只有作业控制 shell 将这些信号的处理设置成了 SIG_DFL。

一个类似的规则同样适用于其他由终端生成的信号：SIGINT、SIGQUIT 以及 SIGHUP。对于 SIGINT 和 SIGQUIT 来讲，其原因是当一个命令在非作业控制 shell 的后台执行时，结果进程不会被放置在一个单独的进程组中，而是与 shell 位于同一个进程组中，而 shell 会在执行命令之前将 SIGINT 和 SIGQUIT 的处理设置为忽略。这样就能确保当用户输入终端中断或退出字符（它们应该只会影响到在前台运行的作业）时进程不会被杀死。如果进程在后面取消了 shell 对这些信号的处理动作，那么会更容易受到这些信号的影响。

当命令通过 nohup(1)被执行时会忽略 SIGHUP 信号，这样就防止了当终端被挂断时命令被杀死的情况的发生，因此应用程序不应该在该信号被忽略时试图改变这个信号的处理动作。

34.7.4 孤儿进程组（SIGHUP 回顾）

在 26.2 节中曾经讲过孤儿进程是那些在父进程终止之后被 init 进程（进程 ID 为 1）收养的进程。在程序中可以使用下面的代码创建一个孤儿进程。

```
if (fork() != 0)                /* Exit if parent (or on error) */
    exit(EXIT_SUCCESS);
```

假设在 shell 中执行一个包含上面这段代码的程序，图 34-3 给出了父进程终止前后该进程的状态。

从图 34-3 中可以看出，在父进程终止之后，子进程不仅是一个孤儿进程，同时也是孤儿进程组的一个成员。SUSv3 认为当一个进程组满足“每个成员的父进程本身是组的一个成员或不是组会话的一个成员”时就变成了一个孤儿进程组。换句话说，如果一个进程组中至少有一个成员拥有一个位于同一会话但不同进程组中的父进程，就不是孤儿进程组。图 34-3 中包含子进程的进程组是孤儿进程组，因为进程组中的子进程是唯一进程，其父进程（init）位于不同的会话中。

> 根据定义，会话首进程位于孤儿进程组中。这是因为 setsid()在新会话中创建了一个新进程组，而会话首进程的父进程则位于不同的会话中。

从 shell 作业控制的角度来讲，孤儿进程组是非常重要的。根据图 34-3 考虑下面的场景。

1. 在父进程退出之前，子进程被停止了（可能是由于父进程向子进程发送了一个停止信号）。

更多资源请访问稀酷客(www.ckook.com)

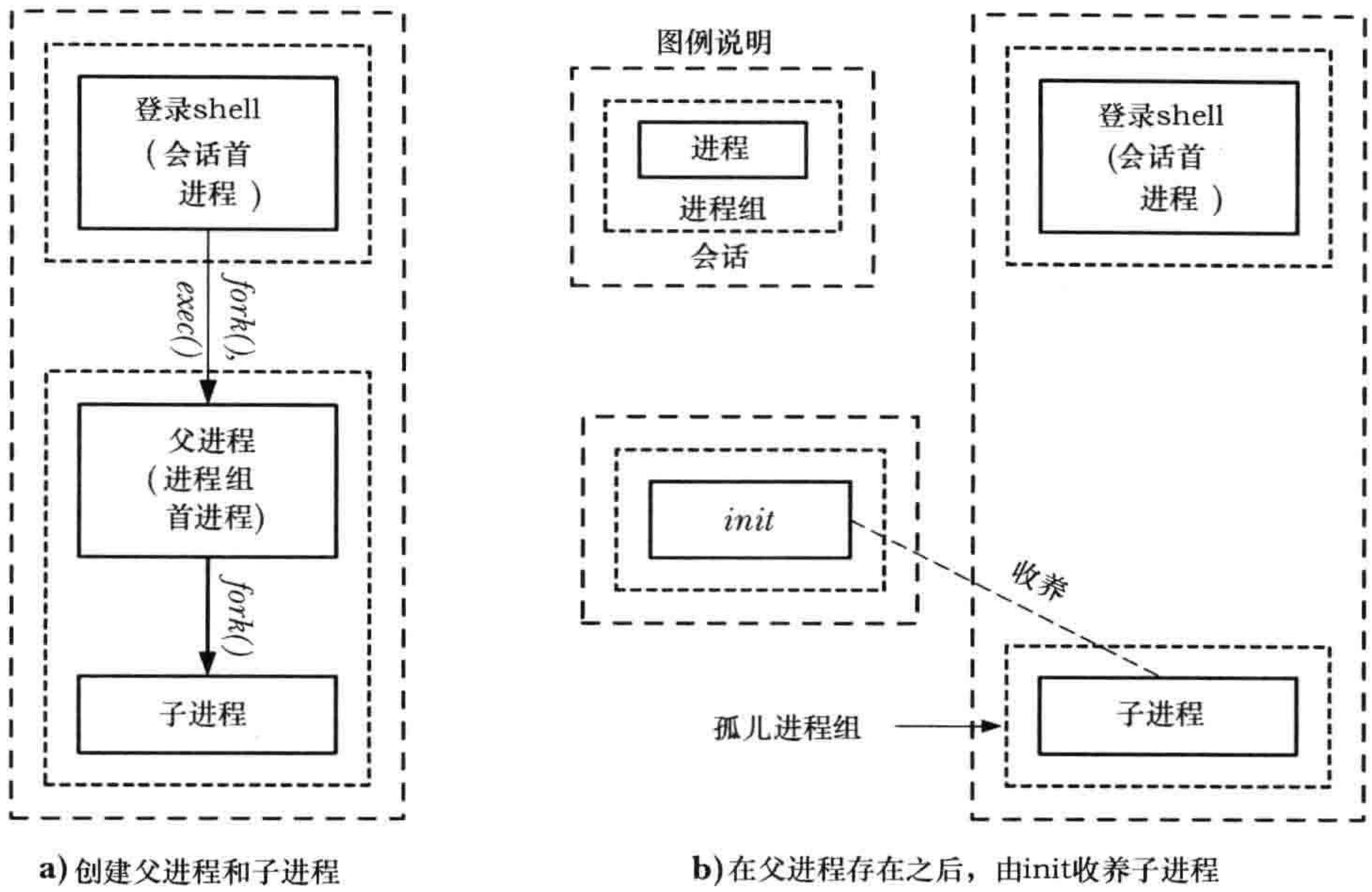


图 34-3　创建孤儿进程组的步骤

2. 当父进程退出时 shell 从作业列表中删除了父进程的进程组。子进程由 init 收养并变成了终端的一个后台进程，包含该子进程的进程组变成了孤儿进程组。
3. 这时没有进程会通过 wait()监控被停止的子进程的状态。

由于 shell 并没有创建子进程，因此它不清楚子进程是否存在以及子进程与已经退出的父进程位于同一个进程组中。此外，init 进程只会检查被终止的子进程并清理该僵尸进程，从而导致被停止的子进程可能会永远残留在系统中，因为没有进程知道要向其发送一个 SIGCONT 信号来恢复它的执行。

即使孤儿进程组中一个被停止的进程拥有一个仍然存活但位于不同会话中的父进程，也无法保证父进程能够向这个被停止的子进程发送 SIGCONT 信号。一个进程可以向同一会话中的任意其他进程发送 SIGCONT 信号，但如果子进程位于不同的会话中，发送信号的标准规则就开始起作用了（参见 20.5 节），因此如果子进程是一个修改了自身的验证信息的特权进程，父进程可能就无法向子进程发送信号。

为防止上面所描述的情况的发生，SUSv3 规定，如果一个进程组变成了孤儿进程组并且拥有很多已停止执行的成员，那么系统会向进程组中的所有成员发送一个 SIGHUP 信号通知它们已经与会话断开连接了，之后再发送一个 SIGCONT 信号确保它们恢复执行。如果孤儿进程组不包含被停止的成员，那么就不会发送任何信号。

一个进程组变成孤儿进程组的原因可能是因为最后一个位于不同进程组但属于同一会话的父进程终止了，也可能是因为父进程位于另一个进程组中的进程组中最后一个进程终止了。（图 34-3 展示了后一种情况。）不管是何种原因引起的，对包含被停止的子进程的新孤儿进程组的处理是一样的。

向包含被停止的成员的新孤儿进程组发送 SIGHUP 和 SIGCONT 信号是为了消除任务控制框架中的特定漏洞，因为没有任何措施能够防止一个进程（拥有合适的权限）向孤儿进程组中的成员发送停止信号来停止它们。这样，进程就会保持在停止的状态，直到一些进程（同样需

要拥有合适的权限）向它们发送一个 SIGCONT 信号。

孤儿进程组中的成员在调用 tcsetpgrp()函数（参见 34.5 节）时会得到 ENOTTY 的错误，在调用 tcsetattr()、tcflush()、tcflow()、tcsendbreak()和 tcdrain()函数时（参见第 62 章）会得到 EIO 的错误。

示例程序

程序清单 34-7 演示了前面描述的对孤儿进程的处理。在为 SIGHUP 和 SIGCONT 信号建立了处理器之后②，程序为每个命令行参数创建了一个子进程③。接着每个子进程停止了自己（通过发出 SIGSTOP 信号）④或等待信号（使用 pause()）⑤。至于子进程到底选择何种动作则取决于相应的命令行参数是否以字母 s（表示 stop）打头。（这里使用了以字母 p 打头的命令行参数来表示相反的动作，即调用 pause()，尽管可以使用除字母 s 之外的任何字母。）

在创建完所有子进程之后，父进程会睡眠一段时间以允许设置子进程时间⑥。（在 24.2 节中曾经提及过以这种方式使用 sleep()不是一个完美的方案，但有时候确实是达成这一目标的可行方法。）接着父进程会退出⑦，这时包含子进程的进程组就会变成孤儿进程组。如果有子进程因为进程组变成孤儿进程组而收到信号，就会调用信号处理器，信号处理器会显示出子进程的进程 ID 和信号编号①。

程序清单 34-7　SIGHUP 和孤儿进程组

pgsjc/orphaned_pgrp_SIGHUP.c

```
#define _GNU_SOURCE     /* Get declaration of strsignal() from <string.h> */
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void             /* Signal handler */
handler(int sig)
{
①   printf("PID=%ld: caught signal %d (%s)\n", (long) getpid(),
            sig, strsignal(sig));     /* UNSAFE (see Section 21.1.2) */
}

int
main(int argc, char *argv[])
{
    int j;
    struct sigaction sa;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s {s|p} ...\n", argv[0]);

    setbuf(stdout, NULL);               /* Make stdout unbuffered */

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
②   if (sigaction(SIGHUP, &sa, NULL) == -1)
        errExit("sigaction");
    if (sigaction(SIGCONT, &sa, NULL) == -1)
        errExit("sigaction");

    printf("parent: PID=%ld, PPID=%ld, PGID=%ld, SID=%ld\n",
            (long) getpid(), (long) getppid(),
            (long) getpgrp(), (long) getsid(0));
```

```
    /* Create one child for each command-line argument */
③   for (j = 1; j < argc; j++) {
        switch (fork()) {
        case -1:
            errExit("fork");
        case 0:         /* Child */
            printf("child:  PID=%ld, PPID=%ld, PGID=%ld, SID=%ld\n",
                    (long) getpid(), (long) getppid(),
                    (long) getpgrp(), (long) getsid(0));

            if (argv[j][0] == 's') {    /* Stop via signal */
                printf("PID=%ld stopping\n", (long) getpid());
④               raise(SIGSTOP);
            } else {                    /* Wait for signal */
                alarm(60);              /* So we die if not SIGHUPed */
                printf("PID=%ld pausing\n", (long) getpid());
⑤               pause();
            }
            _exit(EXIT_SUCCESS);

        default:        /* Parent carries on round loop */
            break;
        }
    }

    /* Parent falls through to here after creating all children */

⑥   sleep(3);                           /* Give children a chance to start */
    printf("parent exiting\n");
⑦   exit(EXIT_SUCCESS);                 /* And orphan them and their group */
}
```

pgsjc/orphaned_pgrp_SIGHUP.c

下面的 shell 会话日志给出了两次运行程序清单 34-7 中的程序的结果。

```
$ echo $$                          Display PID of shell, which is also the session ID
4785
$ ./orphaned_pgrp_SIGHUP s p
parent: PID=4827, PPID=4785, PGID=4827, SID=4785
child:  PID=4828, PPID=4827, PGID=4827, SID=4785
PID=4828 stopping
child:  PID=4829, PPID=4827, PGID=4827, SID=4785
PID=4829 pausing
parent exiting
$ PID=4828: caught signal 18 (Continued)
PID=4828: caught signal 1 (Hangup)
PID=4829: caught signal 18 (Continued)
PID=4829: caught signal 1 (Hangup)
Press Enter to get another shell prompt
$ ./orphaned_pgrp_SIGHUP p p
parent: PID=4830, PPID=4785, PGID=4830, SID=4785
child:  PID=4831, PPID=4830, PGID=4830, SID=4785
PID=4831 pausing
child:  PID=4832, PPID=4830, PGID=4830, SID=4785
PID=4832 pausing
parent exiting
```

第一次运行时在即将变为孤儿进程组的进程组中创建了两个子进程：一个进程停止了自己，另一个则暂停了。（在这次运行中，shell 提示符出现在子进程的输出的中间，这是因为 shell

更多资源请访问稀酷客(www.ckook.com)

注意到父进程已经退出了。）从输出中可以看出，两个子进程在父进程退出之后都收到了 SIGCONT 和 SIGHUP 信号。在第二次运行中创建了两个子进程，但它们都没有停止自身，因此当父进程退出之后不会发送任何信号。

孤儿进程组和 SIGTSTP、SIGTTIN、以及 SIGTTOU 信号

孤儿进程组还对 SIGTSTP、SIGTTIN 以及 SIGTTOU 信号的传输有影响。

在 34.7.1 节中讲过，当后台进程试图从控制终端中调用 read()时将会收到 SIGTTIN 信号，当后台进程试图向设置了 TOSTOP 标记的控制终端调用 write()时会收到 SIGTTOU 信号。但向一个孤儿进程组发送这些信号毫无意义，因为一旦被停止之后，它将再也无法恢复了。基于此，在进行 read()和 write()调用时内核会返回 EIO 的错误，而不是发送 SIGTTIN 或 SIGTTOU 信号。

基于类似的原因，如果 SIGTSTP、SIGTTIN 以及 SIGTTOU 信号的分送会导致停止孤儿进程组中的成员，那么这个信号会被毫无征兆地丢弃。（如果信号正在被处理，那么信号已经被分送给了进程。）这种行为不会因为信号发送方式（如信号可能是由终端驱动器产生的或由显式地调用 kill()而发送）的改变而改变。

34.8 总结

会话和进程组（也称为作业）形成了进程的双层结构：会话是一组进程组的集合，进程组是一组进程的集合。会话首进程是使用 setsid()创建会话的进程。类似的，进程组首进程是使用 setpgid()创建进程组的进程。进程组中的所有成员共享同样的进程组 ID（与进程组首进程的进程组 ID 一样），进程组中所有构成一个会话的进程拥有同样的会话 ID（与会话首进程的 ID 一样）。每个会话可以拥有一个控制终端（/dev/tty），这个关系是在会话首进程打开一个终端设备时建立的。打开控制终端还会导致会话首进程成为终端的控制进程。

会话和进程组是用来支持 shell 作业控制的（尽管有时候在应用程序中会另作他用）。在作业控制中，shell 是会话首进程和运行该 shell 的终端的控制进程。shell 会为执行的每个作业（一个简单的命令或以管道连接起来的一组命令）创建一个独立的进程组，并且提供了将作业在 3 个状态之间迁移的命令。这三个状态分别是在前台运行、在后台运行和在后台停止。

为了支持作业控制，终端驱动器维护了包含控制终端的前台进程组（作业）相关信息的记录。当输入特定的字符时，终端驱动器会向前台作业发送作业控制信号。这些信号会终止或停止前台作业。

终端的前台作业的概念还用于仲裁终端 I/O 请求。只有前台作业中的进程才能从控制终端中读取数据。系统通过 SIGTTIN 信号的分送来防止后台作业读取数据，这个信号的默认动作是停止作业。如果设置了终端的 TOSTOP 标记，那么系统会通过 SIGTTOU 信号的发送来防止后台任务向控制终端写入数据，这个信号的默认动作是停止作业。

当发生终端断开时，内核会向控制进程发送一个 SIGHUP 信号通知它这件事情。这样的事件可能会导致一个链式反应，即向很多其他进程发送一个 SIGHUP 信号。首先，如果控制进程是一个 shell（通常是这种情况），那么在终止之前，进程会向所有由其创建的进程组发送一个 SIGHUP 信号。第二，如果 SIGHUP 信号的分送导致了控制进程的终止，那么内核还会向该控制终端的前台进程组中的所有成员发送一个 SIGHUP 信号。

更多资源请访问稀酷客(www.ckook.com)

一般来讲，应用程序无需弄清楚作业控制信号，但执行屏幕处理操作的程序则是一种例外。这种程序需要正确处理 SIGTSTP 信号，在进程被挂起之前需要将终端特性重置为正确的值，而当应用程序在接收到 SIGCONT 信号而再次恢复时需要还原正确（特定于应用程序）的终端特性。

当一个进程组中没有一个成员进程拥有位于同一会话但不同进程组中的父进程时，就成了孤儿进程组。孤儿进程组是非常重要的，因为在这个组外没有任何进程能够监控组中所有被停止的进程的状态并总是能够向这些被停止的进程发送 SIGCONT 信号来重启它们。这样就可能导致这种被停止的进程永远残留在系统中。为了避免这种情况的发生，当一个拥有被停止的成员进程的进程组变成孤儿进程组时，进程组中的所有成员都会收到一个 SIGHUP 信号，后面跟着一个 SIGCONT 信号，这样就能通知它们变成了孤儿进程并确保重启它们。

更多信息

[Stevens & Rago，2005]的第 9 章介绍了与本章类似的内容，并描述了在登录期间与登录 shell 建立会话时所发生的步骤。glibc 手册对于作业控制相关的函数和作业控制在 shell 中的实现进行了详细的描述。SUSv3 对会话、进程组和作业控制进行了广泛的讨论。

34.9 习题

34-1. 假设一个父进程执行了下面的步骤。

```
/* Call fork() to create a number of child processes, each of which
   remains in same process group as the parent */

/* Sometime later... */
signal(SIGUSR1, SIG_IGN);     /* Parent makes itself immune to SIGUSR1 */

killpg(getpgrp(), SIGUSR1);   /* Send signal to children created earlier */
```

这个应用程序设计可能会碰到什么问题？（考虑 shell 管道。）如何避免此类问题的发生？

34-2. 编写一个程序来验证父进程能够在子进程执行 exec()之前修改子进程的进程组 ID，但无法在执行 exec()之后修改子进程的进程组 ID。

34-3. 编写一个程序来验证在进程组首进程中调用 setsid()会失败。

34-4. 修改程序清单 34-4（disc_SIGHUP.c）来验证当控制进程在收到 SIGHUP 信号而不终止时，内核不会向前台进程组中的成员发送 SIGHUP 信号。

34-5. 假设将程序清单 34-6 中的信号处理器中解除阻塞 SIGTSTP 信号的代码移动到处理器的开头部分。这样做会导致何种竞争条件？

34-6. 编写一个程序来验证当位于孤儿进程组中的一个进程试图从控制终端调用 read()时会得到 EIO 的错误。

34-7. 编写一个程序来验证当 SIGTTIN、SIGTTOU 或 SIGTSTP 三个信号中的一个信号被发送给孤儿进程组中的一个成员时，如果这个信号会停止该进程（即处理方式为 SIG_DFL），那么这个信号就会被丢弃（即不产生任何效果），但如果该信号存在处理器，就会发送该信号。

更多资源请访问稀酷客(www.ckook.com)

第 35 章

进程优先级和调度

本章介绍确定何　时以及哪个进程能够取得 CPU 的使用权的各种系统调用和进程特性。首先会介绍表示进程特点的 nice 值，这个值会影响内核调度器分配给进程的 CPU 时间。接着会介绍 POSIX 实时调度 API，这个 API 允许定义调度进程的策略和优先级，从而更好地控制如何给 CPU 分配进程。最后会讨论用于设置进程的 CPU 亲和力掩码的系统调用，CPU 亲和力掩码能够确定一个运行在多处理器系统上的进程在哪组 CPU 上运行。

35.1　进程优先级（nice 值）

Linux 与大多数其他 UNIX 实现一样，调度进程使用 CPU 的默认模型是循环时间共享。在这种模型中，每个进程轮流使用 CPU 一段时间，这段时间被称为时间片或量子。循环时间共享满足了交互式多任务系统的两个重要需求。

- 公平性：每个进程都有机会用到 CPU。
- 响应度：一个进程在使用 CPU 之前无需等待太长的时间。

在循环时间共享算法中，进程无法直接控制何时使用 CPU 以及使用 CPU 的时间。在默认情况下，每个进程轮流使用 CPU 直至时间片被用光或自己自动放弃 CPU（如进行睡眠或执行一个磁盘读取操作）。如果所有进程都试图尽可能多地使用 CPU（即没有进程会睡眠或被 I/O 操作阻塞），那么它们使用 CPU 的时间差不多是相等的。

进程特性 nice 值允许进程间接地影响内核的调度算法。每个进程都拥有一个 nice 值，其取值范围为-20（高优先级）～19（低优先级），默认值为 0（参见图 35-1）。在传统的 UNIX 实现中，只有特权进程才能够赋给自己（或其他进程）一个负（高）优先级。（在 35.3.2 节中将会解释一些 Linux 上的差别。）非特权进程只能降低自己的优先级，即赋一个大于默认值 0 的 nice 值。这样做之后它们就对其他进程“友好（nice）”了，这个特性的名称也由此而来。

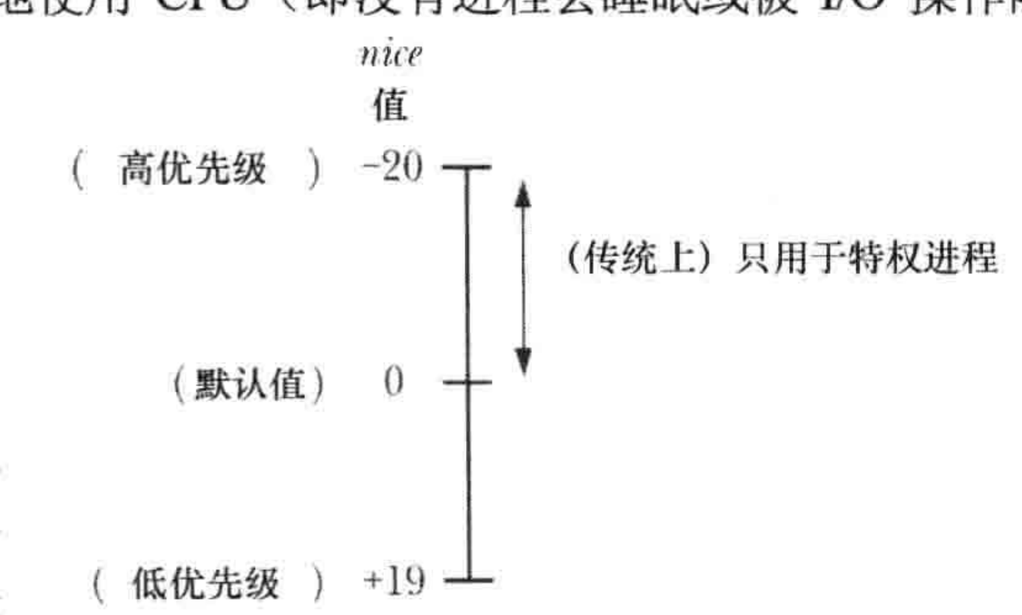


图 35-1　进程 nice 值的范围和解释

使用 fork()创建子进程时会继承 nice 值并且该值会在 exec()调用中得到保持。

getpriority()系统调用服务例程不会返回实际的 nice 值，相反，它会返回一个范围在 1（低优先级）～40（高优先级）之间的数字，这个数字是通过公式 unice=20-knice 计算得来的。这样做是为了避免让系统调用服务例程返回一个负值，因为负值一般都表示错误。（参见 3.1 节中系统调用服务例程的描述。）应用程序是不清楚系统调用服务例程对返回值所做的处理的，因为 C 库函数 getpriority()做了相反的计算操作，它将 20-unice 值返回给了调用程序。

nice 值的影响

进程的调度不是严格按照 nice 值的层次进行的，相反，nice 值是一个权重因素，它导致内核调度器倾向于调度拥有高优先级的进程。给一个进程赋一个低优先级（即高 nice 值）并不会导致它完全无法用到 CPU，但会导致它使用 CPU 的时间变少。nice 值对进程调度的影响程度则依据 Linux 内核版本的不同而不同，同时在不同 UNIX 系统之间也是不同的。

从版本号为 2.6.23 的内核开始，nice 值之间的差别对新内核调度算法的影响比对之前的内核中的调度算法的影响要强。因此，低 nice 值的进程使用 CPU 的时间将比以前少，高 nice 值的进程占用 CPU 的时间将大大提高。

获取和修改优先级

getpriority()和 setpriority()系统调用允许一个进程获取和修改自身或其他进程的 nice 值。

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

int getpriority(int which, id_t who);
        Returns (possibly negative) nice value of specified process on success,
                                                        or -1 on error

int setpriority(int which, id_t who, int prio);
                                        Returns 0 on success, or -1 on error
```

两个系统调用都接收参数 which 和 who，这两个参数用于标识需读取或修改优先级的进程。which 参数确定 who 参数如何被解释。这个参数的取值为下面这些值中的一个。

PRIO_PROCESS

操作进程 ID 为 who 的进程。如果 who 为 0，那么使用调用者的进程 ID。

PRIO_PGRP

操作进程组 ID 为 who 的进程组中的所有成员。如果 who 为 0，那么使用调用者的进程组。

PRIO_USER

操作所有真实用户 ID 为 who 的进程。如果 who 为 0，那么使用调用者的真实用户 ID。

who 参数的类型 id_t 是一个大小能容纳进程 ID 或用户 ID 的整型。

getpriority()系统调用返回由 which 和 who 指定的进程的 nice 值。如果有多个进程符合指定的标准（当 which 为 PRIO_PGRP 或 PRIO_USER 时会出现这种情况），那么将会返回优先级最高的进程的 nice 值（即最小的数值）。由于 getpriority()可能会在成功时返回-1，因此在调用这个函数之前必须要将 errno 设置为 0，接着在调用之后检查返回值为-1 以及 errno 不为 0 才能确认调用成功。

更多资源请访问稀酷客(www.ckook.com)

setpriority()系统调用会将由 which 和 who 指定的进程的 nice 值设置为 prio。试图将 nice 值设置为一个超出允许范围的值（-20～+19）时会直接将 nice 值设置为边界值。

> 以前 nice 值是通过调用 nice(incr)来完成的，这个函数会将调用进程的 nice 值加上 incr。现在这个函数仍然是可用的，但已经被更通用的 setpriority()系统调用所取代了。
>
> 在命令行中与 setpriority()系统调用实现类似功能的命令是 nice(1)，非特权用户可以使用这个命令来运行一个优先级更低的命令，特权用户则可以运行一个优先级更高的命令，超级用户则可以使用 renice(8)来修改既有进程的 nice 值。

特权进程（CAP_SYS_NICE）能够修改任意进程的优先级。非特权进程可以修改自己的优先级（将 which 设为 PRIO_PROCESS，who 设为 0）和其他（目标）进程的优先级，前提是自己的有效用户 ID 与目标进程的真实或有效用户 ID 匹配。Linux 中 setpriority()的权限规则与 SUSv3 中的规则不同，它规定当非特权进程的真实或有效用户 ID 与目标进程的有效用户 ID 匹配时，该进程就能修改目标进程的优先级。UNIX 实现在这一点上与 Linux 有些不同。一些实现遵循的 SUSv3 的规则，而另一些——特别是 BSD 系列——与 Linux 的行为方式一样。

> 版本号小于 2.6.12 的 Linux 内核与之后的内核对非特权进程调用 setpriority()时使用的权限规则不同（也与 SUSv3 不同）。当非特权进程的真实或有效用户 ID 与目标进程的真实用户 ID 匹配时，该进程就能修改目标进程的优先级。从 Linux 2.6.12 开始，权限检查变得与 Linux 中类似的 API 一致了，如 sched_setscheduler()和 sched_setaffinity()。

在版本号小于 2.6.12 的 Linux 内核中，非特权进程只能使用 setpriority()来降低（不可逆的）自己或其他进程的 nice 值。特权进程（CAP_SYS_NICE）可以使用 setpriority()来提高 nice 值。

从版本号为 2.6.12 的内核开始，Linux 提供了 RLIMIT_NICE 资源限制，即允许非特权进程提升 nice 值。非特权进程能够将自己的 nice 值最高提高到公式 20−rlim_cur 指定的值，其中 rlim_cur 是当前的 RLIMIT_NICE 软资源限制。如假设一个进程的 RLIMIT_NICE 软限制是 25，那么其 nice 值可以被提高到−5。根据这个公式以及 nice 值的取值范围为+19（低）～−20（高）的事实可以得出 RLIMIT_NICE 的有效范围为 1（低）～40（高）的结论。（RLIMIT_NICE 没有使用范围为+19～−20 之间的值，因为一些负的资源限制值具有特殊含义——如 RLIM_INFINITY 可以为−1。）

非特权进程能够通过 setpriority()调用来修改其他（目标）进程的 nice 值，前提是调用 setpriority()的进程的有效用户 ID 与目标进程的真实或有效用户 ID 匹配并且对 nice 值的修改符合目标进程的 RLIMIT_NIC 限制。

程序清单 35-1 中的程序使用 setpriority()来修改通过命令行参数（对应于 setpriority()函数的参数）指定的进程的 nice 值，接着调用 getpriority()来验证变更是否生效。

程序清单 35-1：修改和获取进程的 nice 值

procpri/t_setpriority.c

```
#include <sys/time.h>
#include <sys/resource.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int which, prio;
    id_t who;
```

```
    if (argc != 4 || strchr("pgu", argv[1][0]) == NULL)
        usageErr("%s {p|g|u} who priority\n"
                 "    set priority of: p=process; g=process group; "
                 "u=processes for user\n", argv[0]);

    /* Set nice value according to command-line arguments */

    which = (argv[1][0] == 'p') ? PRIO_PROCESS :
                (argv[1][0] == 'g') ? PRIO_PGRP : PRIO_USER;
    who = getLong(argv[2], 0, "who");
    prio = getInt(argv[3], 0, "prio");

    if (setpriority(which, who, prio) == -1)
        errExit("getpriority");

    /* Retrieve nice value to check the change */

    errno = 0;                      /* Because successful call may return -1 */
    prio = getpriority(which, who);
    if (prio == -1 && errno != 0)
        errExit("getpriority");

    printf("Nice value = %d\n", prio);

    exit(EXIT_SUCCESS);
}
```

procpri/t_setpriority.c

35.2 实时进程调度概述

在一个系统上一般会同时运行交互式进程和后台进程，标准的内核调度算法一般能够为这些进程提供足够的性能和响应度。但实时应用对调度器有更加严格的要求，如下所示。

- 实时应用必须要为外部输入提供担保最大响应时间。在很多情况下，这些担保最大响应时间必须非常短（如低于秒级）。如交通导航系统的慢速响应可能会使一个灾难。为了满足这种要求，内核必须要提供工具让高优先级进程能快速地取得 CPU 的控制权，抢占当前运行的所有进程。

> 一些时间关键的应用程序可能需要采取其他措施来避免不可接受的延迟。如为了避免由于页面错误而引起的延迟，应用程序可能会使用 mlock()或 mlockall()（50.2 节中将予以介绍）将其所有虚拟内存锁在 RAM 中。

- 高优先级进程应该能够保持互斥地访问 CPU 直至它完成或自动释放 CPU。
- 实时应用应该能够精确地控制其组件进程的调度顺序。

SUSv3 规定的实时进程调度 API（原先在 POSIX.1b 中定义）部分满足了这些要求。这个 API 提供了两个实时调度策略：SCHED_RR 和 SCHED_FIFO。使用这两种策略中任意一种策略进行调度的进程的优先级要高于使用 35.1 中介绍的标准循环时间分享策略来调度的进程，实时调度 API 使用常量 SCHED_OTHER 来标识这种循环时间分享策略。

每个实时策略允许一个优先级范围。SUSv3 要求实现至少要为实时策略提供 32 个离散的优先级。在每个调度策略中，拥有高优先级的可运行进程在尝试访问 CPU 时总是优先于优先级较低的进程。

更多资源请访问稀酷客(www.ckook.com)

对于多处理器 Linux 系统（包括超线程系统）来讲，高优先级的可运行进程总是优先于优先级较低的进程的规则并不适用。在多处理器系统中，各个 CPU 拥有独立的运行队列（这种方式比使用一个系统层面的运行队列的性能要好），并且每个 CPU 的运行队列中的进程的优先级都局限于该队列。如假设一个双处理器系统中运行着三个进程，进程 A 的实时优先级为 20，并且它位于 CPU 0 的等待队列中，而该 CPU 当前正在运行优先级为 30 的进程 B，即使 CPU 1 正在运行优先级为 10 的进程 C，进程 A 还是需要等待 CPU 0。

包含多个进程的实时应用可以使用 35.4 节中描述的 CPU 亲和力 API 来避免这种调度行为可能引起的问题。如在一个四处理器系统中，所有非关键的进程可以被分配到一个 CPU 中，让其他三个 CPU 处理实时应用。

Linux 提供了 99 个实时优先级，其数值从 1（最低）～99（最高），并且这个取值范围同时适用于两个实时调度策略。每个策略中的优先级是等价的。这意味着如果两个进程拥有同样的优先级，一个进程采用了 SCHED_RR 的调度策略，另一个进程采用了 SCHED_FIFO 的调度策略，那么两个都符合运行的条件，至于到底运行哪个则取决于它们被调度的顺序了。实际上，每个优先级级别都维护着一个可运行的进程队列，下一个运行的进程是从优先级最高的非空队列的队头选取出来的。

POSIX 实时与硬实时对比

满足本节开头处列出的所有要求的应用程序有时候被称为硬实时应用程序。但 POSIX 实时进程调度 API 无法满足这些要求。特别是它没有为应用程序提供一种机制来确保处理输入的响应时间，而这种机制需要操作系统的提供相应的特性，但 Linux 内核并没有提供这种特性（大多数其他标准的操作系统也没有提供这种特性）。POSIX API 仅仅提供了所谓的软实时，允许控制调度哪个进程使用 CPU。

在不给系统增加额外开销的情况下增加对硬实时应用程序的支持是非常困难的，这种新增的开销通常与时间分享应用程序的性能要求是存在冲突的，而典型的桌面和服务器系统上运行的应用程序大部分都是时间分享应用程序。这就是为何大多数 UNIX 内核——包括原来的 Linux——并没有为实时应用程序提供原生支持的原因。但从版本 2.6.18 开始，各种特性都被添加到了 Linux 内核中，从而允许 Linux 为硬实时应用程序提供了完全的原生支持，同时不会给时间分享应用程序增加前面提及到的开销。

35.2.1 SCHED_RR 策略

在 SCHED_RR（循环）策略中，优先级相同的进程以循环时间分享的方式执行。进程每次使用 CPU 的时间为一个固定长度的时间片。一旦被调度执行之后，使用 SCHED_RR 策略的进程会保持对 CPU 的控制直到下列条件中的一个得到满足：

- 达到时间片的终点了；
- 自愿放弃 CPU，这可能是由于执行了一个阻塞式系统调用或调用了 sched_yield()系统调用（35.3.3 节将予以介绍）；
- 终止了；
- 被一个优先级更高的进程抢占了。

对于上面列出的前两个事件，当运行在 SCHED_RR 策略下的进程丢掉 CPU 之后将会被放置在与其优先级级别对应的队列的队尾。在最后一种情况中，当优先级更高的进程执行结

更多资源请访问稀酷客(www.ckook.com)

束之后，被抢占的进程会继续执行直到其时间片的剩余部分被消耗完（即被抢占的进程仍然位于与其优先级级别对应的队列的队头）。

在 SCHED_RR 和 SCHED_FIFO 两种策略中，当前运行的进程可能会因为下面某个原因而被抢占：

- 之前被阻塞的高优先级进程解除阻塞了（如它所等待的 I/O 操作完成了）；
- 另一个进程的优先级被提到了一个级别高于当前运行的进程的优先级的优先级；
- 当前运行的进程的优先级被降低到低于其他可运行的进程的优先级了。

SCHED_RR 策略与标准的循环时间分享调度算法（SCHED_OTHER）类似，即它也允许优先级相同的一组进程分享 CPU 时间。它们之间最重要的差别在于 SCHED_RR 策略存在严格的优先级级别，高优先级的进程总是优先于优先级较低的进程。而在 SCHED_OTHER 策略中，低 nice 值（即高优先级）的进程不会独占 CPU，它仅仅在调度决策时为进程提供了一个较大的权重。前面 35.1 节中曾经讲过，一个优先级较低的进程（即高 nice 值）总是至少会用到一些 CPU 时间的。它们之间另一个重要的差别是 SCHED_RR 策略允许精确控制进程被调用的顺序。

35.2.2 SCHED_FIFO 策略

SCHED_FIFO（先入先出，first-in，first-out）策略与 SCHED_RR 策略类似，它们之间最主要的差别在于在 SCHED_FIFO 策略中不存在时间片。一旦一个 SCHED_FIFO 进程获得了 CPU 的控制权之后，它就会一直执行直到下面某个条件被满足：

- 自动放弃 CPU（采用的方式与前面描述的 SCHED_FIFO 策略中的方式一样）；
- 终止了；
- 被一个优先级更高的进程抢占了（场景与前面描述的 SCHED_FIFO 策略中场景一样）。

在第一种情况中，进程会被放置在与其优先级级别对应的队列的队尾。在最后一种情况中，当高优先级进程执行结束之后（被阻塞或终止了），被抢占的进程会继续执行（即被抢占的进程位于与其优先级级别对应的队列的队头）。

35.2.3 SCHED_BATCH 和 SCHED_IDLE 策略

Linux 2.6 系列的内核添加了两个非标准调度策略：SCHED_BATCH 和 SCHED_IDLE。尽管这些策略是通过 POSIX 实时调度 API 来设置的，但实际上它们并不是实时策略。

SCHED_BATCH 策略是在版本为 2.6.16 的内核中加入的，它与默认的 SCHED_OTHER 策略类似，两个之间的差别在于 SCHED_BATCH 策略会导致频繁被唤醒的任务被调度的次数较少。这种策略用于进程的批量式执行。

SCHED_IDLE 策略是在版本为 2.6.23 的内核中加入的，它也与 SCHED_OTHER 类似，但提供的功能等价于一个非常低的 nice 值（即低于+19）。在这个策略中，进程的 nice 值毫无意义。它用于运行低优先级的任务，这些任务在系统中没有其他任务需要使用 CPU 时才会大量使用 CPU。

35.3 实时进程调用 API

下面开始介绍构成实时进程调度 API 的各个系统调用。这些系统调用允许控制进程调度策略和优先级。

更多资源请访问稀酷客(www.ckook.com)

虽然从 2.0 内核开始实时调度已经是 Linux 的一部分了，但在实现中几个问题存在了很长时间。在 2.2 内核的实现中一些特性仍然无法工作，甚至在 2.4 内核的早期版本中也是同样的情况。其中大多数问题直到 2.4.20 内核才得以修正。

35.3.1 实时优先级范围

sched_get_priority_min()和 sched_get_priority_max()系统调用返回一个调度策略的优先级取值范围。

```
#include <sched.h>

int sched_get_priority_min(int policy);
int sched_get_priority_max(int policy);
```

Both return nonnegative integer priority on success, or −1 on error

在两个系统调用中，policy 指定了需获取哪种调度策略的信息。这个参数的取值一般是 SCHED_RR 或 SCHED_FIFO。sched_get_priority_min()系统调用返回指定策略的最小优先级，sched_get_priority_max()返回最大优先级。在 Linux 上，这些系统调用为 SCHED_RR 和 SCHED_FIFO 策略分别返回范围为 1 到 99 的数字。换句话说，两个实时策略的优先级取值范围是完全一样的，并且优先级相同的 SCHED_RR 和 SCHED_FIFO 进程都具备被调度的资格。（至于哪个进程先被调度则取决于它们在优先级级别队列中的顺序。）

不同 UNIX 实现中的实时策略的取值范围是不同的。因此不能在应用程序中硬编码优先级值，相反，需要根据两个函数的返回值来指定优先级。因此，SCHED_RR 策略中最低的优先级应该是 sched_get_priority_min(SCHED_FIFO)，比它高一级的优先级是 sched_ get_ priority_min（SCHED_FIFO）+1，依此类推。

SUSv3 并不要求 SCHED_RR 和 SCHED_FIFO 策略使用同样的优先级范围，但在大多数 UNIX 实现中都是这样做的。如在 Solaris 8 中两种策略的优先级范围是 0～59，而在 FreeBSD 6.1 中的优先级范围是 0～31。

35.3.2 修改和获取策略和优先级

本节将介绍修改和获取调度策略和优先级的系统调用。

修改调度策略和优先级

sched_setscheduler()系统调用修改进程 ID 为 pid 的进程的调度策略和优先级。如果 pid 为 0，那么将会修改调用进程的特性。

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);
```

Returns 0 on success, or −1 on error

param 参数是一个指向下面这种结构的指针。

更多资源请访问稀酷客(www.ckook.com)

```
struct sched_param {
    int sched_priority;        /* Scheduling priority */
};
```

SUSv3 将 param 参数定义成一个结构以允许实现包含额外的特定于实现的字段，当实现提供了额外的调度策略时这些字段可能会变得有用。但与大多数 UNIX 实现一样，Linux 提供了 sched_priority 字段，该字段指定了调度策略。对于 SCHED_RR 和 SCHED_FIFO 来讲，这个字段的取值必须位于 sched_get_priority_min()和 sched_get_priority_max()规定的范围内；对于其他策略来讲，优先级必须是 0。

policy 参数确定了进程的调度策略，它的取值为表 35-1 中的一个。

表 35-1　　Linux 实时和非实时调度策略

策　略	描　述	SUSv3
SCHED_FIFO	实时先入先出	●
SCHED_RR	实时循环	●
SCHED_OTHER	标准的循环时间分享	●
SCHED_BATCH	与 SCHED_OTHER 类似，但用于批量执行（自 Linux 2.6.16 起）	
SCHED_IDLE	与 SCHED_OTHER 类似，但优先级比最大的 nice 值（+19）还要低（自 Linux 2.6.23 起）	

成功调用 sched_setscheduler()会将 pid 指定的进程移到与其优先级级别对应的队列的队尾。

SUSv3 规定成功调用 sched_setscheduler()时其返回值应该是上一种调度策略。但 Linux 并没有遵循这个规则，在成功调用时该函数会返回 0。一个可移植的应用程序应该通过检查返回值是否不为-1 来判断调用是否成功。

通过 fork()创建的子进程会继承父进程的调度策略和优先级，并且在 exec()调用中会保持这些信息。

sched_setparam()系统调用提供了 sched_setscheduler()函数的一个功能子集。它修改一个进程的调度策略，但不会修改其优先级。

```
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *param);
```

Returns 0 on success, or -1 on error

pid 和 param 参数与 sched_setscheduler()中相应的参数是一样的。

成功调用 sched_setparam()会将 pid 指定的进程移到与其优先级级别对应的队列的队尾。

程序清单 35-2 使用 sched_setscheduler()来设置由命令行参数指定的进程的策略和优先级。第一个参数是一个指定调度策略的字母，第二个参数是一个整数优先级，剩下的参数是需修改调度特性的进程的进程 ID。

程序清单 35-2　修改进程的调度策略和优先级

procpri/sched_set.c

```
#include <sched.h>
#include "tlpi_hdr.h"

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


更多资源请访问稀酷客(www.ckook.com)

```
{
    int j, pol;
    struct sched_param sp;

    if (argc < 3 || strchr("rfo", argv[1][0]) == NULL)
        usageErr("%s policy priority [pid...]\n"
                "    policy is 'r' (RR), 'f' (FIFO), "
#ifdef SCHED_BATCH              /* Linux-specific */
                "'b' (BATCH), "
#endif
#ifdef SCHED_IDLE               /* Linux-specific */
                "'i' (IDLE), "
#endif
                "or 'o' (OTHER)\n",
                argv[0]);
    pol = (argv[1][0] == 'r') ? SCHED_RR :
                (argv[1][0] == 'f') ? SCHED_FIFO :
#ifdef SCHED_BATCH
                (argv[1][0] == 'b') ? SCHED_BATCH :
#endif
#ifdef SCHED_IDLE
                (argv[1][0] == 'i') ? SCHED_IDLE :
#endif
                SCHED_OTHER;
    sp.sched_priority = getInt(argv[2], 0, "priority");

    for (j = 3; j < argc; j++)
        if (sched_setscheduler(getLong(argv[j], 0, "pid"), pol, &sp) == -1)
            errExit("sched_setscheduler");

    exit(EXIT_SUCCESS);
}
```

procpri/sched_set.c

权限和资源限制会影响对调度参数的变更

在 2.6.12 之前的内核中，进程必须要先变成特权进程（CAP_SYS_NICE）才能够修改调度策略和优先级。这个规则的一个例外情况是非特权进程在调用者的有效用户 ID 与目标进程的真实或有效用户 ID 匹配时就能将该进程的调度策略修改为 SCHED_ OTHER。

从 2.6.12 的内核开始，设置实时调度策略和优先级的规则发生了变动，即引入了一个全新的非标准的资源限制 RLIMIT_RTPRIO。在老式内核中，特权（CAP_SYS_NICE）进程能够随意修改任意进程的调度策略和优先级。同时，非特权进程也能够根据下列规则修改调度策略和优先级。

- 如果进程拥有非零的 RLIMIT_RTPRIO 软限制，那么它就能随意修改自己的调度策略和优先级，只要符合实时优先级的上限为其当前实时优先级（如果该进程当前运行于一个实时策略下）的最大值及其 RLIMIT_RTPRIO 软限制值的约束即可。
- 如果进程的 RLIMIT_RTPRIO 软限制值为 0，那么进程只能降低自己的实时调度优先级或从实时策略切换非实时策略。
- SCHED_IDLE 策略是一种特殊的策略。运行在这个策略下的进程无法修改自己的策略，不管 RLIMIT_RTPRIO 资源限制的值是什么。
- 在其他非特权进程中也能执行策略和优先级的修改工作，只要该进程的有效用户 ID 与目标进程的真实或有效用户 ID 匹配即可。

- 进程的软 RLIMIT_RTPRIO 限制值只能确定可以对自己的调度策略和优先级做出哪些变更，这些变更可以由进程自己发起，也可以由其他非特权进程发起。拥有非零限制值的非特权进程无法修改其他进程的调度策略和优先级。

从 2.6.25 的内核开始，Linux 增加了实时调度组的概念。它通过 CONFIG_ RT_ GROUP_ SCHED 内核参数进行配置，会影响到在设置实时调度策略时能够做出哪些变更，具体可参见内核源文件 Documentation/scheduler/sched-rt-group.txt。

获取调度策略和优先级

sched_getscheduler()和 sched_getparam()系统调用获取进程的调度策略和优先级。

```
#include <sched.h>

int sched_getscheduler(pid_t pid);
                                    Returns scheduling policy, or -1 on error
int sched_getparam(pid_t pid, struct sched_param *param);
                                    Returns 0 on success, or -1 on error
```

在这两个系统调用中，pid 指定了需查询信息的进程 ID。如果 pid 为 0，那么就会查询调用进程的信息。两个系统调用都可被非特权进程用来获取任意进程的信息，而不管进程的验证信息是什么。

sched_getparam()系统调用返回由 param 指向的 sched_param 结构中 sched_priority 字段指定的进程的实时优先级。

如果执行成功，sched_getscheduler()将会返回前面表 35-1 中列出的一个策略。

程序清单 35-3 使用了 sched_getscheduler()和 sched_getparam()来获取进程 ID 为命令行参数指定的数值的进程的策略和优先级。下面的 shell 会话演示了这个程序的使用以及程序清单 35-2 的使用。

```
$ su                        Assume privilege so we can set realtime policies
Password:
# sleep 100 &               Create a process
[1] 2006
# ./sched_view 2006         View initial policy and priority of sleep process
2006: OTHER  0
# ./sched_set f 25 2006     Switch process to SCHED_FIFO policy, priority 25
# ./sched_view 2006         Verify change
2006: FIFO  25
```

程序清单 35-3　获取进程的调度策略和优先级

procpri/sched_view.c

```
#include <sched.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int j, pol;
```


更多资源请访问稀酷客(www.ckook.com)

```
    struct sched_param sp;

    for (j = 1; j < argc; j++) {
        pol = sched_getscheduler(getLong(argv[j], 0, "pid"));
        if (pol == -1)
            errExit("sched_getscheduler");

        if (sched_getparam(getLong(argv[j], 0, "pid"), &sp) == -1)
            errExit("sched_getparam");

        printf("%s: %-5s %2d\n", argv[j],
                (pol == SCHED_OTHER) ? "OTHER" :
                (pol == SCHED_RR) ? "RR" :
                (pol == SCHED_FIFO) ? "FIFO" :
#ifdef SCHED_BATCH              /* Linux-specific */
                (pol == SCHED_BATCH) ? "BATCH" :
#endif
#ifdef SCHED_IDLE               /* Linux-specific */
                (pol == SCHED_IDLE) ? "IDLE" :
#endif
                "???", sp.sched_priority);
    }

    exit(EXIT_SUCCESS);
}
```

procpri/sched_view.c

防止实时进程锁住系统

由于 SCHED_RR 和 SCHED_FIFO 进程会抢占所有低优先级的进程（如运行这个程序的 shell），因此在开发使用这些策略的应用程序时需要小心可能会发生失控的实时进程因一直占住 CPU 而导致锁住系统的情况。在程序中可以通过一些方法来避免这种情况的发生。

- 使用 setrlimit()设置一个合理的低软 CPU 时间组员限制（在 36.3 节中描述了 RLIMIT_CPU）。如果进程消耗了太多的 CPU 时间，那么它将会收到一个 SIGXCPU 信号，该信号在默认情况下会杀死该进程。
- 使用 alarm()设置一个警报定时器。如果进程的运行时间超出了由 alarm()调用指定的秒数，那么该进程会被 SIGALRM 信号杀死。
- 创建一个拥有高实时优先级的看门狗进程。这个进程可以进行无限循环，每次循环都睡眠指定的时间间隔，然后醒来并监控其他进程的状态。这种监控可以包含对每个进程消耗的 CPU 时间的度量（参见 23.5.3 节中对 clock_getcpuclockid()函数的讨论）并使用 sched_getscheduler()和 sched_getparam()来检查进程的调度策略和优先级。如果一个进程看起来行为异常，那么看门狗线线程可以降低该进程的优先级或向其发送合适的信号来停止或终止该进程。
- 从 2.6.25 的内核开始，Linux 提供了一个非标准的资源限制 RLIMIT_RTTIME 用于控制一个运行在实时调度策略下的进程在单次运行中能够消耗的 CPU 时间。RLIMIT_RTTIME 的单位是毫秒，它限制了一个进程在不执行阻塞式系统调用时能够消耗的 CPU 时间。当进程执行了这样的系统调用时，累积消耗的 CPU 时间将会被重置为 0。当这个进程被一个优先级更高的进程抢占时，累积消耗的 CPU 时间不会被重置。当进程的时间片被耗完或调用 sched_yield()（参见 35.3.3 节）时进程会放弃 CPU。当进程达到了 CPU 时间限制 RLIMIT_CPU 之后，系统会向其发送一个 SIGXCPU 信号，

更多资源请访问稀酷客(www.ckook.com)

该信号在默认情况下会杀死这个进程。

版本号为 2.6.25 的内核中做出的这个变更还有助于避免失控的实时进程锁住系统，详细信息可参考内核源文件 Documentation/scheduler/sched-rt-group.txt。

避免子进程进程特权调度策略

Linux 2.6.32 增加了一个 SCHED_RESET_ON_FORK，在调用 sched_setscheduler()时可以将 policy 参数的值设置为该常量。系统会将这个标记值与表 35-1 中列出的其中一个策略取 OR。如果设置了这个标记，那么由这个进程使用 fork()创建的子进程就不会继承特权调度策略和优先级了。其规则如下。

- 如果调用进程拥有一个实时调度策略（SCHED_RR 或 SCHED_FIFO），那么子进程的策略会被重置为标准的循环时间分享策略 SCHED_OTHER。
- 如果进程的 nice 值为负值（即高优先级），那么子进程的 nice 值会被重置为 0。

SCHED_RESET_ON_FORK 标记用于媒体回放应用程序，它允许创建单个拥有实时调度策略但不会将该策略传递给子进程的进程。使用 SCHED_RESET_ON_FORK 标记能够通过创建多个运行于实时调度策略下的子进程来防止创建试图超出 RLIMIT_RTTIME 资源限制的子进程。

一旦进程启用了 SCHED_RESET_ON_FORK 标记，那么只有特权进程（CAP_SYS_ NICE）才能够禁用该标记。当子进程被创建出来之后，它的 reset-on-fork 标记会被禁用。

35.3.3 释放 CPU

实时进程可以通过两种方式自愿释放 CPU：通过调用一个阻塞进程的系统调用（如从终端中 read()）或调用 sched_yield()。

```
#include <sched.h>

int sched_yield(void);
```

Returns 0 on success, or −1 on error

sched_yield()的操作是比较简单的。如果存在与调用进程的优先级相同的其他排队的可运行进程，那么调用进程会被放在队列的队尾，队列中队头的进程将会被调度使用 CPU。如果在该优先级队列中不存在可运行的进程，那么 sched_yield()不会做任何事情，调用进程会继续使用 CPU。

虽然 SUSv3 允许 sched_yield()返回一个错误，但在 Linux 或很多其他 UNIX 实现上这个系统调用总会成功。可移植的应用程序应该总是检查这个系统调用是否返回错误。

非实时进程使用 sched_yield()的结果是未定义的。

35.3.4 SCHED_RR 时间片

通过 sched_rr_get_interval()系统调用能够找出 SCHED_RR 进程在每次被授权使用 CPU 时分配到的时间片的长度。

```
#include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *tp);
```

Returns 0 on success, or −1 on error

更多资源请访问稀酷客(www.ckook.com)

与其他进程调度系统调用一样，pid 标识出了需查询信息的进程，当 pid 为 0 时表示调用进程。返回的时间片是由 tp 指向的 timespec 结构。

```
struct timespec {
    time_t tv_sec;          /* Seconds */
    long   tv_nsec;         /* Nanoseconds */
};
```

在最新的 2.6 内核中，实时循环时间片是 0.1 秒。

35.4 CPU 亲和力

当一个进程在一个多处理器系统上被重新调度时无需在上一次执行的 CPU 上运行。之所以会在另一个 CPU 上运行的原因是原来的 CPU 处于忙碌状态。

进程切换 CPU 时对性能会有一定的影响：如果在原来的 CPU 的高速缓冲器中存在进程的数据，那么为了将进程的一行数据加载进新 CPU 的高速缓冲器中，首先必须使这行数据失效（即在没被修改的情况下丢弃数据，在被修改的情况下将数据写入内存）。（为防止高速缓冲器不一致，多处理器架构在某个时刻只允许数据被存放在一个 CPU 的高速缓冲器中。）这个使数据失效的过程会消耗时间。由于存在这个性能影响，Linux（2.6）内核尝试了给进程保证软 CPU 亲和力——在条件允许的情况下进程重新被调度到原来的 CPU 上运行。

高速缓冲器中的一行与虚拟内存管理系统中的一页是类似的。它是 CPU 高速缓冲器和内存之间传输数据的单位。通常行大小的范围为 32～128 字节，更多信息请参考[Schimmel, 1994]和[Drepper, 2007]。

Linux 特有的/proc/PID/stat 文件中的一个字段显示了进程当前执行或上一次执行时所在的 CPU 编号。具体请参见 proc(5)手册。

有时候需要为进程设置硬 CPU 亲和力，这样就能显式地将其限制在可用 CPU 中的一个或一组 CPU 上运行。之所以需要这样做，原因如下。

- 可以避免由使高速缓冲器中的数据失效所带来的性能影响。
- 如果多个线程（或进程）访问同样的数据，那么当将它们限制在同样的 CPU 上的话可能会带来性能提升，因为它们无需竞争数据并且也不存在由此而产生的高速缓冲器未命中。
- 对于时间关键的应用程序来讲，可能需要为此应用程序预留一个或更多 CPU，而将系统中大多数进程限制在其他 CPU 上。

使用 isolcpus 内核启动参数能够将一个或更多 CPU 分离出常规的内核调度算法。将一个进程移到或移出被分离出来的 CPU 的唯一方式是使用本节介绍的 CPU 亲和力系统调用。isolcpus 启动参数是实现上面列出的最后一种场景的首选方式，具体可参考内核源文件 Documentation/kernel-parameters.txt。

Linux 还提供了一个 cpuset 内核参数，该参数可用于包含大量 CPU 的系统以实现如何给进程分配 CPU 和内存的复杂控制，具体可参考内核源文件 Documentation/cpusets.txt。

更多资源请访问稀酷客(www.ckook.com)

Linux 2.6 提供了一对非标准的系统调用来修改和获取进程的硬 CPU 亲和力：sched_setaffinity()和 sched_getaffinity()。

> 很多其他UNIX实现提供了控制CPU亲和力的接口，如HP-UX和Solaris提供了pset_bind()系统调用。

sched_setaffinity()系统调用设置了 pid 指定的进程的 CPU 亲和力。如果 pid 为 0，那么调用进程的 CPU 亲和力就会被改变。

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

int sched_setaffinity(pid_t pid, size_t len, cpu_set_t *set);
```

Returns 0 on success, or −1 on error

赋给进程的 CPU 亲和力由 set 指向的 cpu_set_t 结构来指定。

> 实际上 CPU 亲和力是一个线程级特性，可以调整线程组中各个进程的 CPU 亲和力。如果需要修改一个多线程进程中某个特定线程的 CPU 亲和力的话，可以将 pid 设定为线程中 gettid()调用返回的值。将 pid 设为 0 表示调用线程。

虽然 cpu_set_t 数据类型实现为一个位掩码，但应该将其看成是一个不透明的结构。所有对这个结构的操作都应该使用宏 CPU_ZERO()、CPU_SET()、CPU_CLR()和 CPU_ISSET()来完成。

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

void CPU_ZERO(cpu_set_t *set);
void CPU_SET(int cpu, cpu_set_t *set);
void CPU_CLR(int cpu, cpu_set_t *set);

int CPU_ISSET(int cpu, cpu_set_t *set);
```

Returns true (1) if *cpu* is in *set*, or false (0) otherwise

下面这些宏操作 set 指向的 CPU 集合：

- CPU_ZERO()将 set 初始化为空。
- CPU_SET()将 CPU cpu 添加到 set 中。
- CPU_CLR()从 set 中删除 CPU cpu。
- CPU_ISSET()在 CPU cpu 是 set 的一个成员时返回 true。

> GNU C 库还提供了其他一些宏来操作 CPU 集合，具体可参见 CPU_SET(3)手册。

CPU 集合中的 CPU 从 0 开始编号。<sched.h>头文件定义了常量 CPU_SETSIZE，它是比 cpu_set_t 变量能够表示的最大 CPU 编号还要大的一个数字。CPU_SETSIZE 的值为 1024。

传递给 sched_setaffinity()的 len 参数应该指定 set 参数中字节数（即 sizeof(cpu_set_t)）。

下面的代码将 pid 标识出的进程限制在四处理器系统上除第一个 CPU 之外的任意 CPU 上

运行。

```
cpu_set_t set;

CPU_ZERO(&set);
CPU_SET(1, &set);
CPU_SET(2, &set);
CPU_SET(3, &set);

sched_setaffinity(pid, CPU_SETSIZE, &set);
```

如果 set 中指定的 CPU 与系统中的所有 CPU 都不匹配，那么 sched_setaffinity()调用就会返回 EINVAL 错误。

如果运行调用进程的 CPU 不包含在 set 中，那么进程会被迁移到 set 中的一个 CPU 上。

非特权进程只有在其有效用户 ID 与目标进程的真实或有效用户 ID 匹配时才能够设置目标进程的 CPU 亲和力。特权（CAP_SYS_NICE）进程可以设置任意进程的 CPU 亲和力。

sched_getaffinity()系统调用获取 pid 指定的进程的 CPU 亲和力掩码。如果 pid 为 0，那么就返回调用进程的 CPU 亲和力掩码。

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

int sched_getaffinity(pid_t pid, size_t len, cpu_set_t *set);
```

Returns 0 on success, or −1 on error

返回的 CPU 亲和力掩码位于 set 指向的 cpu_set_t 结构中，同时应该将 len 参数设置为结构中包含的字节数，即 sizeof(cpu_set_t)。使用 CPU_ISSET()宏能够确定哪些 CPU 位于 set 中。

如果目标进程的 CPU 亲和力掩码并没有被修改过，那么 sched_getaffinity()返回包含系统中所有 CPU 的集合。

sched_getaffinity()执行时不会进行权限检查，非特权进程能够获取系统上所有进程的 CPU 亲和力掩码。

通过 fork()创建的子进程会继承其父进程的 CPU 亲和力掩码并且在 exec()调用之间掩码会得以保留。

sched_setaffinity()和 sched_getaffinity()系统调用是 Linux 特有的。

本书源代码中 procpri 子目录下 t_sched_setaffinity.c 和 t_sched_getaffinity.c 程序展示了 sched_setaffinity()和 sched_getaffinity()的使用。

35.5 总结

默认的内核调度算法采用的是循环时间分享策略。默认情况下，在这一策略下的所有进程都能平等地使用 CPU，但可以将进程的 nice 值设置为一个范围从−20（高优先级）～+19（低优先级）的数字来影响调度器对进程的调度。但即使给一个进程设置了一个最低的优先级，它仍然有机会用到 CPU。

Linux 还实现了 POSIX 实时调度扩展。这些扩展允许应用程序精确地控制如何分配 CPU

更多资源请访问稀酷客(www.ckook.com)

给进程。运作在两个实时调度策略 SCHED_RR（循环）和 SCHED_FIFO（先入先出）下的进程的优先级总是高于运作在非实时策略下的进程。实时进程优先级的取值范围为 1（低）～99（高）。只有进程处于可运行状态，那么优先级更高的进程就会完全将优先级低的进程排除在 CPU 之外。运作在 SCHED_FIFO 策略下的进程会互斥地访问 CPU 直到它执行终止或自动释放 CPU 或被进入可运行状态的优先级更高的进程抢占。类似的规则同样适用于 SCHED_RR 策略，但在该策略下，如果存在多个进程运行于同样的优先级下，那么 CPU 就会以循环的方式被这些进程共享。

进程的 CPU 亲和力掩码可以用来将进程限制在多处理器系统上可用 CPU 的子集中运行。这样就可以提高特定类型的应用程序的性能。

更多信息

[Love，2010]提供了 Linux 上进程优先级和调度的背景资料。[Gallmeister, 1995]提供了 POSIX 实时调度 API 的更多信息。虽然[Butenhof, 1996]中很多有关实时调度 API 的讨论都是针对 POSIX 线程的，但它也为本章中有关实时调度的讨论提供了有用的背景资料。

更多有关 CPU 亲和力以及控制多处理器系统上给线程分配 CPU 和内存节点的信息可以参见内核源文件 Documentation/cpusets.txt、mbind(2)、set_mempolicy(2)以及 cpuset(7)手册。

35.6 习题

35-1. 实现 nice(1)命令。

35-2. 编写一个与 nice(1)命令类似的实时调度程序 set-user-ID-root 程序。这个程序的命令行界面如下所示：

```
# ./rtsched policy priority command arg...
```

在上面的命令中，policy 中 r 表示 SCHED_RR，f 表示 SCHED_FIFO。基于在 9.7.1 节和 38.3 节中描述的原因，这个程序在执行命令前应该丢弃自己的特权 ID。

35-3. 编写一个运行于 SCHED_FIFO 调度策略下的程序，然后创建一个子进程。在两个进程中都执行一个能导致进程最多消耗 3 秒 CPU 时间的函数。（这可以通过使用一个循环并在循环中不断使用 times()系统调用来确定累积消耗的 CPU 时间来完成。）每当消耗了 1/4 秒的 CPU 时间之后，函数应该打印出一条显示进程 ID 和迄今消耗的 CPU 时间的消息。每当消耗了 1 秒的 CPU 时间之后，函数应该调用 sched_yield()来将 CPU 释放给其他进程。（另一种方法是进程使用 sched_setparam()提升对方的调度策略。）从程序的输出中应该能够看出两个进程交替消耗了 1 秒的 CPU 时间。（注意在 35.3.2 节中给出的有关防止失控实时进程占住 CPU 的建议。）

35-4. 如果两个进程在一个多处理器系统上使用管道来交换大量数据，那么两个进程运行在同一个 CPU 上的通信速度应该要快于两个进程运行在不同的 CPU 上，其原因是当两个进程运行在同一个 CPU 上时能够快速地访问管道数据，因为管道数据可以保留在 CPU 的高速缓冲器中。相反，当两个进程运行在不同的

更多资源请访问稀酷客(www.ckook.com)

CPU 上时将无法享受 CPU 高速缓冲器带来的优势。读者如果拥有多处理器系统，可以编写一个使用 sched_setaffinity()强制将两个进程运行在同一个 CPU 上或运行在两个不同的 CPU 上的程序来演示这种效果。（第 44 章描述了管道的使用。）

更多资源请访问稀酷客(www.ckook.com)

第36章

进程资源

每个进程都会消耗诸如内存和 CPU 时间之类的系统资源。本章将介绍与资源相关的系统调用，首先会介绍 getrusage()系统调用，该函数允许一个进程监控自己及其子进程已经用掉的资源。接着会介绍 setrlimit()和 getrlimit()系统调用，它们可以用来修改和获取调用进程对各类资源的消耗限值。

36.1 进程资源使用

getrusage()系统调用返回调用进程或其子进程用掉的各类系统资源的统计信息。

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

int getrusage(int who, struct rusage *res_usage);
```

Returns 0 on success, or -1 on error

who 参数指定了需查询资源使用信息的进程，其取值为下列几个值中的一个。

RUSAGE_SELF

返回调用进程相关的信息。

RUSAGE_CHILDREN

返回调用进程的所有被终止和处于等待状态的子进程相关的信息。

RUSAGE_THREAD（自 Linux 2.6.26 起）

返回调用线程相关的信息。这个值是 Linux 特有的。

res_usage 参数是一个指向 rusage 结构的指针，其定义如程序清单 36-1 所示。

程序清单 36-1：rusage 结构的定义

```
struct rusage {
    struct timeval ru_utime;      /* User CPU time used */
    struct timeval ru_stime;      /* System CPU time used */
```


更多资源请访问稀酷客(www.ckook.com)

```
    long            ru_maxrss;      /* Maximum size of resident set (kilobytes)
                                       [used since Linux 2.6.32] */
    long            ru_ixrss;       /* Integral (shared) text memory size
                                       (kilobyte-seconds) [unused] */
    long            ru_idrss;       /* Integral (unshared) data memory used
                                       (kilobyte-seconds) [unused] */
    long            ru_isrss;       /* Integral (unshared) stack memory used
                                       (kilobyte-seconds) [unused] */
    long            ru_minflt;      /* Soft page faults (I/O not required) */
    long            ru_majflt;      /* Hard page faults (I/O required) */
    long            ru_nswap;       /* Swaps out of physical memory [unused] */
    long            ru_inblock;     /* Block input operations via file
                                       system [used since Linux 2.6.22] */
    long            ru_oublock;     /* Block output operations via file
                                       system [used since Linux 2.6.22] */
    long            ru_msgsnd;      /* IPC messages sent [unused] */
    long            ru_msgrcv;      /* IPC messages received [unused] */
    long            ru_nsignals;    /* Signals received [unused] */
    long            ru_nvcsw;       /* Voluntary context switches (process
                                       relinquished CPU before its time slice
                                       expired) [used since Linux 2.6] */
    long            ru_nivcsw;      /* Involuntary context switches (higher
                                       priority process became runnable or time
                                       slice ran out) [used since Linux 2.6] */
};
```

从程序清单 36-1 中的注释中可以看出，在 Linux 上，在调用 getrusage()（或 wait3()以及 wait4()）时，rusage 结构中的很多字段都不会被填充，只有最新的内核才会填充这些字段。其中一些字段在 Linux 中并没有用到，只有 UNIX 实现用到了这些字段。而 Linux 系统之所以也提供了这些字段是为了防止以后扩展时需要修改 rusage 结构而破坏既有的应用程序库。

虽然大多数 UNIX 实现都提供了 getrusage()，但 SUSv3 并没有全面规范这个系统调用（仅规定了 ru_utime 和 ru_stime 字段），这样做的部分原因是因为 rusage 结构中的很多字段的含义是依赖于实现的。

ru_utime 和 ru_stime 字段的类型是 timeval 结构（参见 10.1 节），它分别表示一个进程在用户模式和内核模式下消耗的 CPU 的秒数和毫秒数。（10.7 节中介绍的 times()系统调用也会返回类似的信息。）

Linux 特有的/proc/PID/stat 文件提供了系统中所有进程的某些资源使用信息（CPU 时间和页面错误），更多信息可参考 proc(5)手册。

getrusage() RUSAGE_CHILDREN 操作返回的 rusage 结构中包含了调用进程的所有子孙进程的资源使用统计信息。如假设三个进程之间的关系为父进程、子进程和孙子进程，那么当子进程在 wait()孙子进程时，孙子进程的资源使用值就会被加到子进程的 RUSAGE_CHILDREN 值上，当父进程执行了一个 wait()子进程的操作时，子进程和孙子进程的资源使用信息就会被加到父进程的 RUSAGE_CHILDREN 值上。而如果子进程没有 wait()孙子进程的话，孙子进程的资源使用就不会被记录到父进程的 RUSAGE_CHILDREN 值中。

在 RUSAGE_CHILDREN 操作中，ru_maxrss 字段返回调用进程的所有子孙进程中最大驻留集大小（不是所有子孙进程之和）。

更多资源请访问稀酷客(www.ckook.com)

SUSv3 规定当 SIGCHLD 被忽略时（这样子进程就不会变成可等待的僵死进程了），子进程的统计信息不应该被加到 RUSAGE_CHILDREN 的返回值中。但在 26.3.3 节中曾经指出过在版本号早于 2.6.9 的内核中，Linux 的行为与这个规则不同——当 SIGCHLD 被忽略时，已经死去的子进程的资源使用值会被加到 RUSAGE_CHILDREN 的返回值中。

36.2 进程资源限制

每个进程都用一组资源限值，它们可以用来限制进程能够消耗的各种系统资源。如在执行任意一个程序之前如果不想让它消耗太多资源，则可以设置该进程的资源限制。使用 shell 的内置命令 ulimit 可以设置 shell 的资源限制（在 C shell 中是 limit）。shell 创建用来执行用户命令的进程会继承这些限制。

从 2.6.24 的内核开始，Linux 特有的/proc/PID/limits 文件可以用来查看任意进程的所有资源限制。这个文件由相应进程的真实用户 ID 所拥有，并且只有进程 ID 为用户 ID 的进程（或特权进程）才能够读取这个文件。

getrlimit()和 setrlimit()系统调用允许一个进程读取和修改自己的资源限制。

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

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

Both return 0 on success, or −1 on error

resource 参数标识出了需读取或修改的资源限制。rlim 参数用来返回限制值（getrlimit()）或指定新的资源限制值（(setrlimit()），它是一个指向包含两个字段的结构的指针。

```
struct rlimit {
    rlim_t rlim_cur;        /* Soft limit (actual process limit) */
    rlim_t rlim_max;        /* Hard limit (ceiling for rlim_cur) */
};
```

这两个字段对应于一种资源的两个关联限制：软限制（rlim_cur）和硬限制（rlim_max）。（rlim_t 数据类型是一个整数类型。）软限制规定了进程能够消耗的资源数量。一个进程可以将软限制调整为从 0 到硬限制之间的值。对于大多数资源来讲，硬限制的唯一作用是为软限制设定了上限。特权（CAP_SYS_RESOURCE）进程能够增大和缩小硬限制（只要其值仍然大于软限制），但非特权进程则只能缩小硬限制（这个行为是不可逆的）。在 getrlimit()和 setrlimit()调用中，rlim_cur 和 rlim_max 取值为 RLIM_INFINITY 表示没有限制（不限制资源的使用）。

在大多数情况下，特权进程和非特权进程在使用资源时都会受到限制。通过 fork()创建的子进程会继承这些限制并且在 exec()调用之间不得到保持。

表 36-1 列出了 getrlimit()和 setrlimit()两个函数中 resource 参数的可取值，详细信息可参见 36.3 节。

虽然资源限制是一个进程级别的特性，但在某些情况下，不仅需要度量一个进程对相关资源的消耗情况，还需要度量同一个真实用户 ID 下所有进程对资源的消耗总和情况。限制能

更多资源请访问稀酷客(www.ckook.com)

创建的进程数目的 RLIMIT_NPROC 就较好地遵循了这个规则。仅仅将这个限制施加于进程本身所创建的子进程的数量的做法不是非常有效，因为由该进程创建的每个子进程都可以创建自己的子进程，而这些子进程还能够创建更多的子进程，以此类推。因此，这个限制是根据同一真实用户 ID 下所有的进程数来度量的。注意只有在设置了资源限制的进程中（即进程本身及继承了限制值的子孙进程）才会对资源使用情况进行检查。如果同一真实用户 ID 下存在一个没有设置限制（即限制值为无限）或设置了一个不同的限制值的进程，那么就会根据它所设置的限制值来检查其创建的子进程的数量。

下面在介绍每类资源的限制值时都会指出此类资源限制值是指同一真实用户 ID 下所有进程累积能够消耗的资源限制值。如果没有特别指出，那么一个资源限制值就是指进程本身能够消耗的资源限制值。

记住，在很多情况下，获取和设置资源限制的 shell 命令（bash 和 Korn shell 中是 ulimit，C shell 中是 limit）使用的单位与 getrlimit()和 setrlimit()使用的单位不同。如 shell 命令在限制各种内存段的大小时通常以千字节为单位。

表 36-1：getrlimit()和 setrlimit()中的资源值

资　源	限　制	SUSv3
RLIMIT_AS	进程虚拟内存限制大小（字节数）	●
RLIMIT_CORE	核心文件大小（字节数）	●
RLIMIT_CPU	CPU 时间（秒数）	●
RLIMIT_DATA	进程数据段（字节数）	●
RLIMIT_FSIZE	文件大小（字节数）	●
RLIMIT_MEMLOCK	锁住的内存（字节数）	
RLIMIT_MSGQUEUE	为真实用户 ID 分配的 POSIX 消息队列的字节数（自 Linux 2.6.8 起）	
RLIMIT_NICE	nice 值（自 Linux 2.6.12 起）	
RLIMIT_NOFILE	最大的文件描述符数量加 1	●
RLIMIT_NPROC	真实用户 ID 下的进程数量	
RLIMIT_RSS	驻留集大小（字节数；没有实现）	
RLIMIT_RTPRIO	实时调度策略（自 Linux 2.6.12 起）	
RLIMIT_RTTIME	实时 CPU 时间（微秒；自 Linux 2.6.25 起）	
RLIMIT_SIGPENDING	真实用户 ID 信号队列中的信号数（自 Linux 2.6.8 起）	
RLIMIT_STACK	栈段的大小（字节数）	●

示例程序

在开始介绍各种资源限制的具体内容之前，首先来看一个使用了资源限制的简单示例。程序清单 36-2 定义了函数 printRlimit()，该函数会显示一条消息以及指定资源的软限制和硬限制。

rlim_t 数据类型与 off_t 通常是一样的，用来处理文件大小资源限制 RLIMIT_FSIZE 的表示。基于这个原因，在打印 rlim_t 值时（如在程序清单 36-2 中），需要像 5.10 节所说的

更多资源请访问稀酷客(www.ckook.com)

那样将它们转换成 long long 并使用%lld printf()修饰符。

程序清单 36-3 调用了 setrlimit()来设置一个用户能够创建的进程数量的软限制和硬限制（RLIMIT_NPROC），同时使用了程序清单 36-2 中的函数 printRlimit()来输出变更之前和之后的资源限制，最后根据资源限制创建了尽可能多的进程。在运行这个程序时，如果将软限制设置为 30，硬限制设置为 100，那么就能看到下面的输出。

```
$ ./rlimit_nproc 30 100
Initial maximum process limits:  soft=1024; hard=1024
New maximum process limits:      soft=30; hard=100
Child 1 (PID=15674) started
Child 2 (PID=15675) started
Child 3 (PID=15676) started
Child 4 (PID=15677) started
ERROR [EAGAIN Resource temporarily unavailable] fork
```

在这个例子中，程序只创建了 4 个新进程，因为在该用户下已经运行着 26 个进程了。

程序清单 36-2：显示进程资源限制

procres/print_rlimit.c

```
#include <sys/resource.h>
#include "print_rlimit.h"          /* Declares function defined here */
#include "tlpi_hdr.h"

int                        /* Print 'msg' followed by limits for 'resource' */
printRlimit(const char *msg, int resource)
{
    struct rlimit rlim;

    if (getrlimit(resource, &rlim) == -1)
        return -1;

    printf("%s soft=", msg);
    if (rlim.rlim_cur == RLIM_INFINITY)
        printf("infinite");
#ifdef RLIM_SAVED_CUR              /* Not defined on some implementations */
    else if (rlim.rlim_cur == RLIM_SAVED_CUR)
        printf("unrepresentable");
#endif
    else
        printf("%lld", (long long) rlim.rlim_cur);

    printf("; hard=");
    if (rlim.rlim_max == RLIM_INFINITY)
        printf("infinite\n");
#ifdef RLIM_SAVED_MAX              /* Not defined on some implementations */
    else if (rlim.rlim_max == RLIM_SAVED_MAX)
        printf("unrepresentable");
#endif
    else
        printf("%lld\n", (long long) rlim.rlim_max);

    return 0;
}
```

procres/print_rlimit.c

程序清单 36-3：设置 RLIMIT_NPROC 资源限制

procres/rlimit_nproc.c

```
#include <sys/resource.h>
#include "print_rlimit.h"              /* Declaration of printRlimit() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct rlimit rl;
    int j;
    pid_t childPid;
    if (argc < 2 || argc > 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s soft-limit [hard-limit]\n", argv[0]);

    printRlimit("Initial maximum process limits: ", RLIMIT_NPROC);

    /* Set new process limits (hard == soft if not specified) */

    rl.rlim_cur = (argv[1][0] == 'i') ? RLIM_INFINITY :
                                getInt(argv[1], 0, "soft-limit");
    rl.rlim_max = (argc == 2) ? rl.rlim_cur :
                (argv[2][0] == 'i') ? RLIM_INFINITY :
                                getInt(argv[2], 0, "hard-limit");
    if (setrlimit(RLIMIT_NPROC, &rl) == -1)
        errExit("setrlimit");

    printRlimit("New maximum process limits:     ", RLIMIT_NPROC);

    /* Create as many children as possible */

    for (j = 1; ; j++) {
        switch (childPid = fork()) {
        case -1: errExit("fork");

        case 0: _exit(EXIT_SUCCESS);            /* Child */

        default:        /* Parent: display message about each new child
                           and let the resulting zombies accumulate */
            printf("Child %d (PID=%ld) started\n", j, (long) childPid);
            break;
        }
    }
}
```

procres/rlimit_nproc.c

无法表示的限制值

在某些程序设计环境中，rlim_t 数据类型可能无法表示某个特定资源限制的所有可取值，这是因为一个系统可能提供了多个程序设计环境，而在这些程序设计环境中 rlim_t 数据类型的大小是不同的。如当一个 off_t 为 64 位的大型文件编译环境被添加到 off_t 为 32 位的系统中时就会出现这种情况。（在每种环境中，rlim_t 和 off_t 的大小是一样的。）这就会导致出现这样一种情况，即一个 off_t 为 64 位的程序能够创建一个子进程来执行一个 rlim_t 值较小的程序，这样子进程就会继承父进程的资源限制（如文件大小限制），但该资源限制超过了最大的 rlim_t 值。

更多资源请访问稀酷客(www.ckook.com)

为了帮助可移植应用程序处理可能出现的无法标识资源限制的情况，SUSv3 规定了两个常量来标记无法表示的限制值：RLIM_SAVED_CUR 和 RLIM_SAVED_MAX。如果一个软资源限制无法用 rlim_t 表示，那么 getrlimit()将会在 rlim_cur 字段返回 RLIM_SAVED_CUR。而 RLIM_SAVED_MAX 的功能类似，即当碰到无法表示的硬限制时在 rlim_max 字段返回该值。

SUSv3 允许实现在 rlim_t 能够表示资源限制的所有可取值时将 RLIM_SAVED_CUR 和 RLIM_SAVED_MAX 定义成与 RLIM_INFINITY 一样的值。在 Linux 上，这两个常量值就是这样定义的，这样 rlim_t 能够表示资源限制的所有可取值，但在像 x86-32 这样的 32 位架构上这种做法是不对的。在那些架构上，在一个大文件编译环境中，glibc 将 rlim_t 定义为 64 位，但内核中表示资源限制的数据类型是 unsigned long，它只有 32 位。当前版本的 glibc 是这样处理这种情况的：如果一个设置了_FILE_OFFSET_BITS=64 编译选项的程序试图将一个资源限制值设置为一个超出 32 位 unsigned long 表示范围的值，那么 glibc 中 setrlimit()的包装函数会毫无征兆地将这个值转换成 RLIM_INFINITY。换句话说，要求完成的资源限制值的设置并没有如实地被完成。

由于在很多 x86-32 发行版中，处理文件的实用程序在编译时通常都设置了_FILE_OFFSET_BITS=64 参数，因此当资源限制值超出 32 位的表示范围时系统不如实地设置资源限制值的做法不仅仅会影响到应用程序开发人员，还会影响到最终的用户。

有些人可能会认为 glibc setrlimit()包装函数的做法要比在请求的资源限制超出 32 位 unsigned long 表示范围时返回一个错误要好，而这个问题的本质是内核的限制，glibc 的开发人员在处理这个问题时则采用了前面正文中介绍的方法。

36.3 特定资源限制细节

本节将详细介绍Linux上可用的各个资源限制，特别需要注意那些Linux特有的资源限制。

RLIMIT_AS

RLIMIT_AS 限制规定了进程的虚拟内存（地址空间）的最大字节数，试图（brk()、sbrk()、mmap()、mremap()以及 shmat()）超出这个限制会得到 ENOMEM 错误。在实践中，程序中会超出这个限制的最常见的地方是在调用 malloc 包中的函数时，因为它们会使用 sbrk()和 mmap()。当碰到这个限制时，栈增长操作也会失败，进而会出现下面 RLIMIT_STACK 限制中列出的情况。

RLIMIT_CORE

RLIMIT_CORE 限制规定了当进程被特定信号（参见 22.1 节）终止时产生的核心 dump 文件的最大字节数。当达到这个限制时，核心 dump 文件就不会再产生了。将这个限制指定为 0 会阻止核心 dump 文件的创建，这种做法有时候是比较有用的，因为核心 dump 文件可能会变得非常大，而最终用户通常又不知道如何处理这些文件。另一个禁用核心 dump 文件的原因是安全性——防止程序占用的内存中的内容输出到磁盘上。如果 RLIMIT_FSIZE 限制值低于这个限制值，那么核心 dump 文件的最大大小会被限制为 RLIMIT_FSIZE 字节。

RLIMIT_CPU

RLIMIT_CPU 限制规定了进程最多使用的 CPU 时间（包括系统模式和用户模式）。SUSv3 要求当达到软限制值时需要向进程发送一个 SIGXCPU 信号，但并没有规定其他的细节。

更多资源请访问稀酷客(www.ckook.com)

（SIGXCPU 信号的默认动作是终止一个进程并输出一个核心 dump。）此外，也可以为 SIGXCPU 信号建立一个处理器来完成期望的处理工作，然后将控制返回给主程序。在达到软限制值之后，内核（在 Linux 上）会在进程每消耗一秒钟的 CPU 时间后向其发送一个 SIGXCPU 信号。当进程持续执行直至达到硬 CPU 限制时，内核会向其发送一个 SIGKILL 信号，该信号总是会终止进程。

不同的 UNIX 实现对进程处理完 SIGXCPU 信号之后继续消耗 CPU 时间这种情况的处理方式不同。大多数会每隔固定时间间隔向进程发送一个 SIGXCPU 信号。读者如果想要编写使用这个信号的可移植应用程序，那么应该在首次收到这个信号之后就完成必要的清理工作，然后终止执行。（或者，程序也可以在收到这个信号之后修改资源限制。）

RLIMIT_DATA

RLIMIT_DATA 限制规定了进程的数据段的最大字节数（在 6.3 节中介绍的初始化数据、非初始化数据、堆段的总和）。试图（sbrk()和 brk()）访问这个限制之外的数据段会得到 ENOMEM 的错误。与 RLIMIT_AS 一样，程序中会超出这个限制的最常见的地方是在调用 malloc 包中的函数时。

RLIMIT_FSIZE

RLIMIT_FSIZE 限制规定了进程能够创建的文件的最大字节数。如果进程试图扩充一个文件使之超出软限制值，那么内核就会向其发送一个 SIGXFSZ 信号，并且系统调用（如 write()或 truncate()）会返回 EFBIG 错误。SIGXFSZ 信号的默认动作是终止进程并产生一个核心 dump。此外，也可以捕获这个信号并将控制返回给主程序。不管怎样，后续视图扩充该文件的操作都会得到同样的信号和错误。

RLIMIT_MEMLOCK

RLIMIT_MEMLOCK 限制（源自 BSD，在 SUSv3 中并没有此限制，只有 Linux 和 BSD 系统提供了这个限制）规定了一个进程最多能够将多少字节的虚拟内存锁进物理内存以防止内存被交换出去。这个限制会影响 mlock()和 mlockall()系统调用以及 mmap()和 shmctl()系统调用的加锁参数，后面 50.2 节中将会介绍其中的细节信息。

如果在调用 mlockall()时指定了 MCL_FUTURE 标记，那么 RLIMIT_MEMLOCK 限制也会导致后续的 brk()、sbrk()、mmap()和 mremap()调用失败。

RLIMIT_MSGQUEUE

RLIMIT_MSGQUEUE 限制（Linux 特有的，自 Linux 2.6.8 起）规定了能够为调用进程的真实用户 ID 的 POSIX 消息队列分配的最大字节数。当使用 mq_open()创建了一个 POSIX 消息队列后会根据下面的公式将字节数与这个限制值进行比较。

```
bytes = attr.mq_maxmsg * sizeof(struct msg_msg *) +
        attr.mq_maxmsg * attr.mq_msgsize;
```

在这个公式中，attr 是传给 mq_open()的第四个参数 mq_attr 结构。加数中包含 sizeof(struct msg_msg *)确保了用户无法在队列中无止境地加入长度为零的消息。（msg_msg 结构是内核内部使用的一个数据类型。）这样做是有必要的，因为虽然长度为零的消息不包含数据，但它们需要消耗一些系统内存以供簿记。

RLIMIT_MSGQUEUE 限制只会影响调用进程。这个用户下的其他进程不会受到影响，因为它们也会设置这个限制或继承这个限制。

更多资源请访问稀酷客(www.ckook.com)

RLIMIT_NICE

RLIMIT_NICE 限制（Linux 特有的，自 Linux 2.6.12 起）规定了使用 sched_setscheduler()和 nice()能够为进程设置的最大 nice 值。这个最大值是通过公式 20 – rlim_cur 计算得来的，其中 rlim_cur 是当前的 RLIMIT_NICE 软资源限制，更多细节信息可参见 35.1 节。

RLIMIT_NOFILE

RLIMIT_NOFILE 限制规定了一个数值，该数值等于一个进程能够分配的最大文件描述符数量加 1。试图（如 open()、pipe()、socket()、accept()、shm_open()、dup()、dup2()、fcntl(F_DUPFD)和 epoll_create()）分配的文件描述符数量超出这个限制时会失败。在大多数情况，失败的错误是 EMFILE，但在 dup2(fd, newfd)调用中，失败的错误是 EBADF，在 fcntl(fd, F_DUPFD, newfd)调用中当 newfd 大于或等于这个限制时，失败的错误是 EINVAL。

对 RLIMIT_NOFILE 限制的变更会通过 sysconf(_SC_OPEN_MAX)的返回值反应出来。SUSv3 允许但不强制实现在修改 RLIMIT_NOFILE 限制前后调用 sysconf(_SC_OPEN_MAX)返回不同的值，在这一点上其他实现的行为与 Linux 可能并不相同。

> SUSv3 声称如果一个应用程序将进程的软或硬 RLIMIT_NOFILE 限制设置为一个小于或等于进程当前打开的最大文件描述符数量的值时会出现预期之外的行为。
>
> 在 Linux 上可以通过使用 readdir()扫描/proc/PID/fd 目录下的内容来检查一个进程当前打开的文件描述符，这个目录包含了进程当前打开的每个文件描述符的符号链接。

内核为 RLIMIT_NOFILE 限制规定了一个最大值。在 2.6.25 之前的内核中，这个最大值是一个由内核常量 NR_OPEN 定义的硬编码值，其值为 1048576。（提高这个最大值需要重建内核。）从 2.6.25 的版本开始，这个限制由 Linux 特有的/proc/sys/fs/nr_open 文件定义。这个文件中的默认值是 1048576，超级用户可以修改这个值。试图将软或硬 RLIMIT_NOFILE 限制设置为一个大于最大值的值会产生 EPERM 错误。

还存在一个系统级别的限制，它规定了系统中所有进程能够打开的文件数量，通过 Linux 特有的/proc/sys/fs/file-max 文件能够获取和修改这个限制。（可以将 file-max 更加精确地定义为系统中所能打开的文件描述符数量限制，具体可参考 5.4 节。）只有特权（CAP_SYS_ADMIN）进程才能够超出 file-max 的限制。在非特权进程中，当系统调用碰到 file-max 限制时会返回 ENFILE 错误。

RLIMIT_NPROC

RLIMIT_NPROC 限制（源自 BSD，在 SUSv3 中并没有此限制，只有 Linux 和 BSD 系统提供了这个限制）规定了调用进程的真实用户 ID 下最多能够创建的进程数量。试图（fork()、vfork()和 clone()）超出这个限制会得到 EAGAIN 错误。

RLIMIT_NPROC 限制只影响调用进程。这个用户下的其他进程不会受到影响，除非它们也设置或继承了这个限制。这个限制不适用于特权（CAP_SYS_ADMIN 和 CAP_SYS_RESOURCE）进程。

> Linux 还提供了系统层面的限制来规定所有用户能够创建的进程数量。在 Linux 2.4 以及之后的版本中，可以使用 Linux 特有的/proc/sys/kernel/threads-max 文件来获取和修改这个限制。
>
> 准确地说，RLIMIT_NPROC 资源限制和 threads-max 文件实际上限制的是所能创建的线程数量，而不是进程的数量。

更多资源请访问稀酷客(www.ckook.com)

不同版本的内核为 RLIMIT_NPROC 资源限制设置的默认值不同。在 Linux 2.2 中，该值是根据一个固定的公式计算得来的。在 Linux 2.4 和之后的版本中，该值是使用一条公式根据可用的物理内存数量计算得来的。

SUSv3 没有规定 RLIMIT_NPROC 资源限制，但它规定了通过 sysconf(_SC_CHILD_MAX)调用来获取（不是修改）一个用户 ID 最多能够创建的进程数量。Linux 也支持这个 sysconf()调用，但只有 2.6.23 前的内核才支持这个调用。这个调用不会返回精确的信息——它总是返回值 999。自 Linux 2.6.23 起（以及 glibc 2.4 和之后的版本），这个调用会正确地返回限制（通过检查 RLIMIT_NPROC 资源限制值）。

不存在一种统一的方法能够在不同系统中找出某个特定用户 ID 已经创建的进程数。在 Linux 中可以通过扫描系统中的所有/proc/PID/status 文件并检查 Uid 条目（它会按顺序列出四个进程用户 ID：真实、有效、保留集和文件系统）下的信息来估算一个用户当前拥有的进程，但有一点需要记住，即当完成扫描之后信息可能已经发生了改变。

RLIMIT_RSS

RLIMIT_RSS 限制（源自 BSD，在 SUSv3 并没有此限制，但该限制是被广泛使用的）规定了进程驻留集中的最大页面数，即当前位于物理内存中的虚拟内存页面总数。Linux 也提供了这个限制，但当前并没有起任何作用。

在 Linux 2.4 之前的内核中（早于以及包括 2.4.29），RLIMIT_RSS 会影响到 madvise() MADV_WILLNEED 操作的行为（参见 50.4 节）。如果这个操作因达到 RLIMIT_RSS 限制而无法执行，那么 errno 中会存储 EIO 错误。

RLIMIT_RTPRIO

RLIMIT_RTPRIO 限制（Linux 特有的，自 Linux 2.6.12 起）规定了使用 sched_ setscheduler() 和 sched_setparam()能够为进程设置的最高实时优先级，具体细节请参考 35.3.2 节。

RLIMIT_RTTIME

RLIMIT_RTTIME 限制（Linux 特有的，自 Linux 2.6.25 起）规定了一个进程在实时调度策略中不睡眠（即执行一个阻塞系统调用）的情况下最大能消耗的 CPU 秒数。当达到这个限制时系统的行为与达到 RLIMIT_CPU 限制时的行为是一样的：如果进程达到了软限制，那么内核会向进程发送一个 SIGXCPU 信号，之后进程每消耗一秒的 CPU 时间都会收到一个 SIGXCPU 信号。在达到硬限制时，内核会向进程发送一个 SIGKILL 信号。更多细节请参考 35.3.2 节。

RLIMIT_SIGPENDING

RLIMIT_SIGPENDING 限制（Linux 特有的，自 Linux 2.6.8 起）规定了调用进程的真实用户 ID 下信号队列中最多能容纳的信号数量。试图（sigqueue()）超出这个限制会得到 EAGAIN 错误。

RLIMIT_SIGPENDING 只影响调用进程。这个用户下的其他进程不会受到影响，除非它们也设置或继承了这个限制。

在最初的实现中，RLIMIT_SIGPENDING 限制的默认值为 1024。自内核 2.6.12 起，这个限制的默认值被改成了与 RLIMIT_NPROC 的默认值一样的值。

更多资源请访问稀酷客(www.ckook.com)

在检查 RLIMIT_SIGPENDING 限制时统计的队列中的信号包括实时信号和标准信号。(一个进程的标准信号只能进入队列一次。) 但这个限制只适用于 sigqueue()。即使这个实时用户 ID 下的进程的信号队列中包含的信号数量已经达到了这个限制，仍然可以使用 kill()来将不在进程的信号队列中的各个信号（包括实时信号）的一个实例添加到队列中。

在 2.6.12 之前的内核中，Linux 特有的/proc/PID/status 文件中的 SigQ 字段显示了进程的真实用户 ID 的信号队列中当前存储的信号数量以及最多存储的信号数量。

RLIMIT_STACK

RLIMIT_STACK 限制规定了进程栈的最大字节数。试图扩展栈大小以至于超出这个限制会导致内核向该进程发送一个 SIGSEGV 信号。由于栈空间已经被用光了，因此捕获这个信号的唯一方式是建立另外一个备用的信号栈，具体可参考 21.3 节。

> 自 Linux 2.6.23 起，RLIMIT_STACK 限制还确定了存储进程的命令行参数和环境变量的最大空间，具体可参考 execve(2)手册。

36.4 总结

进程会消耗各种系统资源。getrusage()系统调用允许一个进程监控自己及其子进程所消耗的各种资源。

setrlimit()和 getrlimit()系统调用允许一个进程设置和获取自己在各种资源上的消耗限制。每个资源限制有两个组成部分：一个是软限制，内核在检查进程的资源消耗时会应用这个限制；另外一个是硬限制，它是软限制可取的最大值。非特权进程能够将一个资源的软限制设置为 0 到硬限制之间的任意一个值，但只能降低硬限制值。特权进程能够随意修改这两个限制值，只要软限制值小于或等于硬限制值即可。当一个进程达到软限制时通常会通过接收一个信号或在调用试图超出这个限制的系统调用时得到一个错误来得知这个事实。

36.5 习题

36-1. 编写一个程序使用 getrusage() RUSAGE_CHILDREN 标记获取 wait()调用所等待的子进程相关的信息。(让程序创建一个子进程并使子进程消耗一些 CPU 时间，接着让父进程在调用 wait()前后都调用 getrusage()。)

36-2. 编写一个程序来执行一个命令，接着显示其当前的资源使用。这个程序与 time(1)命令的功能类似，因此可以像下面这样使用这个程序：

```
$ ./rusage command arg...
```

36-3. 编写一个程序来确定当进程所消耗的各种资源超出通过 setrlimit()调用设置的软限制时会发生什么事情。

更多资源请访问稀酷客(www.ckook.com)

第37章

DAEMON

本章介绍 daemon 进程的特征和将一个进程变成一个 daemon 所需完成的步骤。此外，还会介绍如何在 daemon 中使用 syslog 工具记录消息。

37.1 概述

daemon 是一种具备下列特征的进程。

- 它的生命周期很长。通常，一个 daemon 会在系统启动的时候被创建并一直运行直至系统被关闭。
- 它在后台运行并且不拥有控制终端。控制终端的缺失确保了内核永远不会为 daemon 自动生成任何任务控制信号以及终端相关的信号（如 SIGINT、SIGTSTP 和 SIGHUP）。

daemon 是用来执行特殊任务的，如下面的示例所示。

- cron：一个在规定时间执行命令的 daemon。
- sshd：安全 shell daemon，允许在远程主机上使用一个安全的通信协议登录系统。
- httpd：HTTP 服务器 daemon（Apache），它用于服务 Web 页面。
- inetd：Internet 超级服务器 daemon（参见 60.5 节），它监听从指定的 TCP/IP 端口上进入的网络连接并启动相应的服务器程序来处理这些连接。

很多标准的 daemon 会作为特权进程运行（即有效用户 ID 为 0），因此在编写 daemon 程序时应该遵循第 38 章中给出的指南。

通常会将 daemon 程序的名称以字母 d 结尾（但并不是所有人都遵循这个惯例）。

在 Linux 上，特定的 daemon 会作为内核线程运行。实现此类 daemon 的代码是内核的一部分，它们通常在系统启动的时候被创建。当使用 ps(1)列出线程时，这些 daemon 的名称会用方括号（[]）括起来。其中一个内核线程是 pdflush，它会定期将脏页面（即高速缓冲区中的页面）写入磁盘。

更多资源请访问稀酷客(www.ckook.com)

37.2 创建一个 daemon

要变成 daemon，一个程序需要完成下面的步骤。

1. 执行一个 fork()，之后父进程退出，子进程继续执行。（结果是 daemon 成为了 init 进程的子进程。）之所以要做这一步是因为下面两个原因。
   - 假设 daemon 是从命令行启动的，父进程的终止会被 shell 发现，shell 在发现之后会显示出另一个 shell 提示符并让子进程继续在后台运行。
   - 子进程被确保不会成为一个进程组首进程，因为它从其父进程那里继承了进程组 ID 并且拥有了自己的唯一的进程 ID，而这个进程 ID 与继承而来的进程组 ID 是不同的，这样才能够成功地执行下面一个步骤。
2. 子进程调用 setsid()（参见 34.3 节）开启一个新会话并释放它与控制终端之间的所有关联关系。
3. 如果 daemon 从来没有打开过终端设备，那么就无需担心 daemon 会重新请求一个控制终端了。如果 daemon 后面可能会打开一个终端设备，那么必须要采取措施来确保这个设备不会成为控制终端。这可以通过下面两种方式实现。
   - 在所有可能应用到一个终端设备上的 open()调用中指定 O_NOCTTY 标记。
   - 或者更简单地说，在 setsid()调用之后执行第二个 fork()，然后再次让父进程退出并让孙子进程继续执行。这样就确保了子进程不会成为会话组长，因此根据 System V 中获取终端的规则（Linux 也遵循了这个规则），进程永远不会重新请求一个控制终端（参见 34.4 节）。

> 在遵循 BSD 规则的实现中，一个进程只能通过一个显式的 ioctl() TIOCSCTTY 操作来获取一个控制终端，因此第二个 fork()调用对控制终端的获取并没有任何影响，但多一个 fork()调用不会带来任何坏处。

4. 清除进程的 umask（参见 15.4.6 节）以确保当 daemon 创建文件和目录时拥有所需的权限。
5. 修改进程的当前工作目录，通常会改为根目录（/）。这样做是有必要的，因为 daemon 通常会一直运行直至系统关闭为止。如果 daemon 的当前工作目录为不包含/的文件系统，那么就无法卸载该文件系统（参见 14.8.2 节）。或者 daemon 可以将工作目录改为完成任务时所在的目录或在配置文件中定义的一个目录，只要包含这个目录的文件系统永远不会被卸载即可。如 cron 会将自身放在/var/spool/cron 目录下。
6. 关闭 daemon 从其父进程继承而来的所有打开着的文件描述符。（daemon 可能需要保持继承而来的文件描述的打开状态，因此这一步是可选的或者是可变更的。）之所以需要这样做的原因有很多。由于 daemon 失去了控制终端并且是在后台运行的，因此让 daemon 保持文件描述符 0、1 和 2 的打开状态毫无意义，因为它们指向的就是控制终端。此外，无法卸载长时间运行的 daemon 打开的文件所在的文件系统。因此，通常的做法是关闭所有无用的打开着的文件描述符，因为文件描述符是一种有限的资源。

> 一些 UNIX 实现（如 Solaris 9 和一些最新的 BSD 发行版）提供了一个名为 closefrom(n)（或类似的名称）的函数，它关闭所有大于或等于 n 的文件描述符。Linux 上并不存在这个

更多资源请访问稀酷客(www.ckook.com)

函数。

7. 在关闭了文件描述符 0、1 和 2 之后，daemon 通常会打开/dev/null 并使用 dup2()（或类似的函数）使所有这些描述符指向这个设备。之所以要这样做是因为下面两个原因。
   - 它确保了当 daemon 调用了在这些描述符上执行 I/O 的库函数时不会出乎意料地失败。
   - 它防止了 daemon 后面使用描述符 1 或 2 打开一个文件的情况，因为库函数会将这些描述符当做标准输出和标准错误来写入数据（进而破坏了原有的数据）。

/dev/null 是一个虚拟设备，它总会将写入的数据丢弃。当需要删除一个 shell 命令的标准输出和错误时可以将它们重定向到这个文件。从这个设备中读取数据总是会返回文件结束的错误。

下面是 becomeDaemon()函数的实现，它完成了上面描述的步骤以将调用者变成一个 daemon。

```
#include <syslog.h>

int becomeDaemon(int flags);
```

Returns 0 on success, or −1 on error

becomeDaeomon()函数接收一个位掩码参数 flags，它允许调用者有选择地执行其中的步骤，具体可参考程序清单 37-1 中列出的头文件中的注释。

程序清单 37-1：become_daemon.c 的头文件

daemons/become_daemon.h

```
#ifndef BECOME_DAEMON_H              /* Prevent double inclusion */
#define BECOME_DAEMON_H

/* Bit-mask values for 'flags' argument of becomeDaemon() */

#define BD_NO_CHDIR           01     /* Don't chdir("/") */
#define BD_NO_CLOSE_FILES     02     /* Don't close all open files */
#define BD_NO_REOPEN_STD_FDS  04     /* Don't reopen stdin, stdout, and
                                        stderr to /dev/null */
#define BD_NO_UMASK0         010     /* Don't do a umask(0) */

#define BD_MAX_CLOSE  8192           /* Maximum file descriptors to close if
                                        sysconf(_SC_OPEN_MAX) is indeterminate */

int becomeDaemon(int flags);

#endif
```

daemons/become_daemon.h

程序清单 37-2 给出了 becomeDaemon()函数的实现。

GNU C 库提供了一个非标准的 daemon()函数，它将调用者变成一个 daemon。glibc daemon()函数与这里的 becomeDaemon()函数不同，它并没有定义一个与 flags 参数等价的参数。

更多资源请访问稀酷客(www.ckook.com)

程序清单 37-2：创建一个 daemon 进程

daemons/become_daemon.c

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

int                             /* Returns 0 on success, -1 on error */
becomeDaemon(int flags)
{
    int maxfd, fd;

    switch (fork()) {                   /* Become background process */
    case -1: return -1;
    case 0:  break;                     /* Child falls through... */
    default: _exit(EXIT_SUCCESS);       /* while parent terminates */
    }

    if (setsid() == -1)                 /* Become leader of new session */
        return -1;

    switch (fork()) {                   /* Ensure we are not session leader */
    case -1: return -1;
    case 0:  break;
    default: _exit(EXIT_SUCCESS);
    }
    if (!(flags & BD_NO_UMASK0))
        umask(0);                       /* Clear file mode creation mask */

    if (!(flags & BD_NO_CHDIR))
        chdir("/");                     /* Change to root directory */

    if (!(flags & BD_NO_CLOSE_FILES)) { /* Close all open files */
        maxfd = sysconf(_SC_OPEN_MAX);
        if (maxfd == -1)                /* Limit is indeterminate... */
            maxfd = BD_MAX_CLOSE;       /* so take a guess */

        for (fd = 0; fd < maxfd; fd++)
            close(fd);
    }

    if (!(flags & BD_NO_REOPEN_STD_FDS)) {
        close(STDIN_FILENO);            /* Reopen standard fd's to /dev/null */

        fd = open("/dev/null", O_RDWR);

        if (fd != STDIN_FILENO)         /* 'fd' should be 0 */
            return -1;
        if (dup2(STDIN_FILENO, STDOUT_FILENO) != STDOUT_FILENO)
            return -1;
        if (dup2(STDIN_FILENO, STDERR_FILENO) != STDERR_FILENO)
            return -1;
    }

    return 0;
}
```

daemons/become_daemon.c

假设编写一个程序调用 becomeDaemon(0)，之后睡眠一段时间，那么可以使用 ps(1)来查看结果进程的一些特性。

```
$ ./test_become_daemon
$ ps -C test_become_daemon -o "pid ppid pgid sid tty command"
  PID  PPID  PGID   SID TT       COMMAND
24731     1 24730 24730 ?        ./test_become_daemon
```

由于代码比较简单，因此这里并没有给出 daemons/test_become_daemon.c 的源代码，本书的源代码包中提供了这个程序的代码。

在 ps 的输出中，TT 标题下的？表示进程没有控制终端。从进程 ID 与会话 ID（SID）不同的事实也可以看出进程不是会话首进程，因此在打开终端设备时不会重新获得控制终端，这就是 daemon 应该具备的特性。

37.3 编写 daemon 指南

前面曾经提及过，一个 daemon 通常只有在系统关闭的时候才会终止。很多标准的 daemon 是通过在系统关闭时执行特定于应用程序的脚本来停止的。而那些不以这种方式终止的 daemon 会收到一个 SIGTERM 信号，因为在系统关闭的时候 init 进程会向所有其子进程发送这个信号。在默认情况下，SIGTERM 信号会终止一个进程。如果 daemon 在终止之前需要做些清理工作，那么就需要为这个信号建立一个处理器。这个处理器必须能快速地完成清理工作，因为 init 在发完 SIGTERM 信号的 5 秒之后会发送一个 SIGKILL 信号。（这并不意味着这个 daemon 能够执行 5 秒的 CPU 时间，因为 init 会同时向系统中的所有进程发送信号，而它们可能都试图在 5 秒内完成清理工作。）

由于 daemon 是长时间运行的，因此要特别小心潜在的内存泄露问题（参见 7.1.3 节）和文件描述符泄露（即应用程序没有关闭所有打开着的文件描述符）。如果此类 bug 影响到了 daemon 的运行，那么唯一的解决方案是杀死它，之后（修复了 bug）再重新启动它。

很多 daemon 需要确保同一时刻只有一个实例处于活跃状态。如让两个 cron daemon 都试图实行计划任务毫无意义。在 55.6 节中将会介绍完成这个任务的技术。

37.4 使用 SIGHUP 重新初始化一个 daemon

由于很多 daemon 需要持续运行，因此在设计 daemon 程序时需要克服一些障碍。

- 通常 daemon 会在启动时从相关的配置文件中读取操作参数，但有些时候需要在不重启 daemon 的情况下快速修改这些参数。
- 一些 daemon 会产生日志文件。如果 daemon 永远不关闭日志文件的话，那么日志文件就会无限制地增长，最终会阻塞文件系统。（在 18.3 节中曾经提到过即使删除了一个文件的文件名，只要有进程还打开着这个文件，那么这个文件就会一直存在下去。）这里需要有一种机制来告诉 daemon 关闭其日志文件并打开一个新文件，这样就能够在需要的时候旋转日志文件了。

解决这两个问题的方案是让 daemon 为 SIGHUP 建立一个处理器，并在收到这个信号时采取所需的措施。在 34.4 节中曾经讲到，当控制进程与控制终端断开连接之后就会生成 SIGHUP 信号。由于 daemon 没有控制终端，因此内核永远不会向 daemon 发送这个信号。这样 daemon

更多资源请访问稀酷客(www.ckook.com)

就可以使用 SIGHUP 信号来达到目的。

logrotate 程序可以用来自动旋转 daemon 的日志文件，具体可参考 logrotate(8)手册。

程序清单 37-3 提供了 daemon 如何使用 SIGHUP 的一个示例。这个程序为 SIGHUP 建立了一个处理器②，然后变成 daemon ③，接着打开日志文件④，最后读取其配置文件⑤。SIGHUP 处理器①只设置了一个全局标记变量 hupReceived，主程序会检查这个变量。主程序位于一个循环中，它每隔 15 秒向日志文件输出一条消息⑧。循环中对 sleep()的调用⑥用来模拟真实应用程序中的某些处理工作。在循环中每次 sleep()返回之后，程序会检查 hupReceived 变量是否被设置⑦，如果该变量被设置了，那么程序就会重新打开日志文件和重新读取配置文件以及清除 hupReceived 标记。

限于篇幅，程序清单 37-3 中并没有给出 logOpen()、logClose()、logMessage()和 readConfigFile()函数的实现，但本书的源代码分发包中提供了这些函数的源代码。其中前面三个函数所做的工作从其名称中就能看出，readConfigFile()函数只是简单地从配置文件中读取一行数据并将这行数据输出到日志文件中。

一些 daemon 在收到 SIGHUP 信号时会使用其他方法来重新初始化自身：它们会关闭所有文件，然后使用 exec()重新启动自身。

下面是运行程序清单 37-3 时可能看到的输出，这里首先创建一个哑配置文件，然后启动这个 daemon。

```
$ echo START > /tmp/ds.conf
$ ./daemon_SIGHUP
$ cat /tmp/ds.log                                   View log file
2011-01-17 11:18:34: Opened log file
2011-01-17 11:18:34: Read config file: START
```

现在修改这个配置文件并在向 daemon 发送 SIGHUP 信号之前重命名日志文件。

```
$ echo CHANGED > /tmp/ds.conf
$ date +'%F %X'; mv /tmp/ds.log /tmp/old_ds.log
2011-01-17 11:19:03 AM
$ date +'%F %X'; killall -HUP daemon_SIGHUP
2011-01-17 11:19:23 AM
$ ls /tmp/*ds.log                                   Log file was reopened
/tmp/ds.log  /tmp/old_ds.log
$ cat /tmp/old_ds.log                               View old log file
2011-01-17 11:18:34: Opened log file
2011-01-17 11:18:34: Read config file: START
2011-01-17 11:18:49: Main: 1
2011-01-17 11:19:04: Main: 2
2011-01-17 11:19:19: Main: 3
2011-01-17 11:19:23: Closing log file
```

ls 的输出表明新旧日志文件同时存在。当使用 cat 查看旧的日志文件中的内容时可以看出，即使使用了 mv 命令来重命名这个文件，daemon 仍然会将日志信息记录到那个文件中。这时如果不再需要这个旧日志文件，就可以删除这个旧日志文件了。在查看新日志文件时会发现配置文件被重新读取了。

```
$ cat /tmp/ds.log
2011-01-17 11:19:23: Opened log file
2011-01-17 11:19:23: Read config file: CHANGED
2011-01-17 11:19:34: Main: 4
$ killall daemon_SIGHUP                             Kill our daemon
```


更多资源请访问稀酷客(www.ckook.com)

注意 daemon 的日志和配置文件通常会像程序清单 37-3 所做的那样被放置在标准目录中，而不是/tmp 目录中。按照惯例，配置文件会被放在/etc 或它的一个子目录中，日志文件会被放在/var/log 中。Daemon 程序通常会提供命令行参数来指定其他存放位置以替换默认的存放位置。

程序清单 37-3：使用 SIGHUP 重新初始化一个 daemon

daemons/daemon_SIGHUP.c

```
#include <sys/stat.h>
#include <signal.h>
#include "become_daemon.h"
#include "tlpi_hdr.h"

static const char *LOG_FILE = "/tmp/ds.log";
static const char *CONFIG_FILE = "/tmp/ds.conf";

/* Definitions of logMessage(), logOpen(), logClose(), and
   readConfigFile() are omitted from this listing */

static volatile sig_atomic_t hupReceived = 0;
                                        /* Set nonzero on receipt of SIGHUP */
 from
static void
sighupHandler(int sig)
{
①    hupReceived = 1;
}

int
main(int argc, char *argv[])
{
    const int SLEEP_TIME = 15;      /* Time to sleep between messages */
    int count = 0;                  /* Number of completed SLEEP_TIME intervals */
    int unslept;                    /* Time remaining in sleep interval */
    struct sigaction sa;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = sighupHandler;
②   if (sigaction(SIGHUP, &sa, NULL) == -1)
        errExit("sigaction");

③   if (becomeDaemon(0) == -1)
        errExit("becomeDaemon");

④   logOpen(LOG_FILE);
⑤   readConfigFile(CONFIG_FILE);

    unslept = SLEEP_TIME;

    for (;;) {
⑥       unslept = sleep(unslept);       /* Returns > 0 if interrupted */

⑦       if (hupReceived) {              /* If we got SIGHUP... */
            logClose();
```

```
            logOpen(LOG_FILE);
            readConfigFile(CONFIG_FILE);
            hupReceived = 0;            /* Get ready for next SIGHUP */
        }

        if (unslept == 0) {             /* On completed interval */
            count++;
⑧           logMessage("Main: %d", count);
            unslept = SLEEP_TIME;       /* Reset interval */
        }
    }
}
```

daemons/daemon_SIGHUP.c

37.5 使用 syslog 记录消息和错误

在编写 daemon 时碰到的一个问题是如何显示错误消息。由于 daemon 是在后台运行的，因此通常无法像其他程序那样将消息输出到关联终端上。这个问题的一种解决方式是将消息写入到一个特定于应用程序的日志文件中，就像程序清单 37-3 所做的那样。这种方式存在的一个主要问题是让系统管理员管理多个应用程序日志文件和监控其中是否存在错误消息比较困难，syslog 工具就用于解决这个问题。

37.5.1 概述

syslog 工具提供了一个集中式日志工具，系统中的所有应用程序都可以使用这个工具来记录日志消息。图 37-1 提供了这个工具的一个概览。

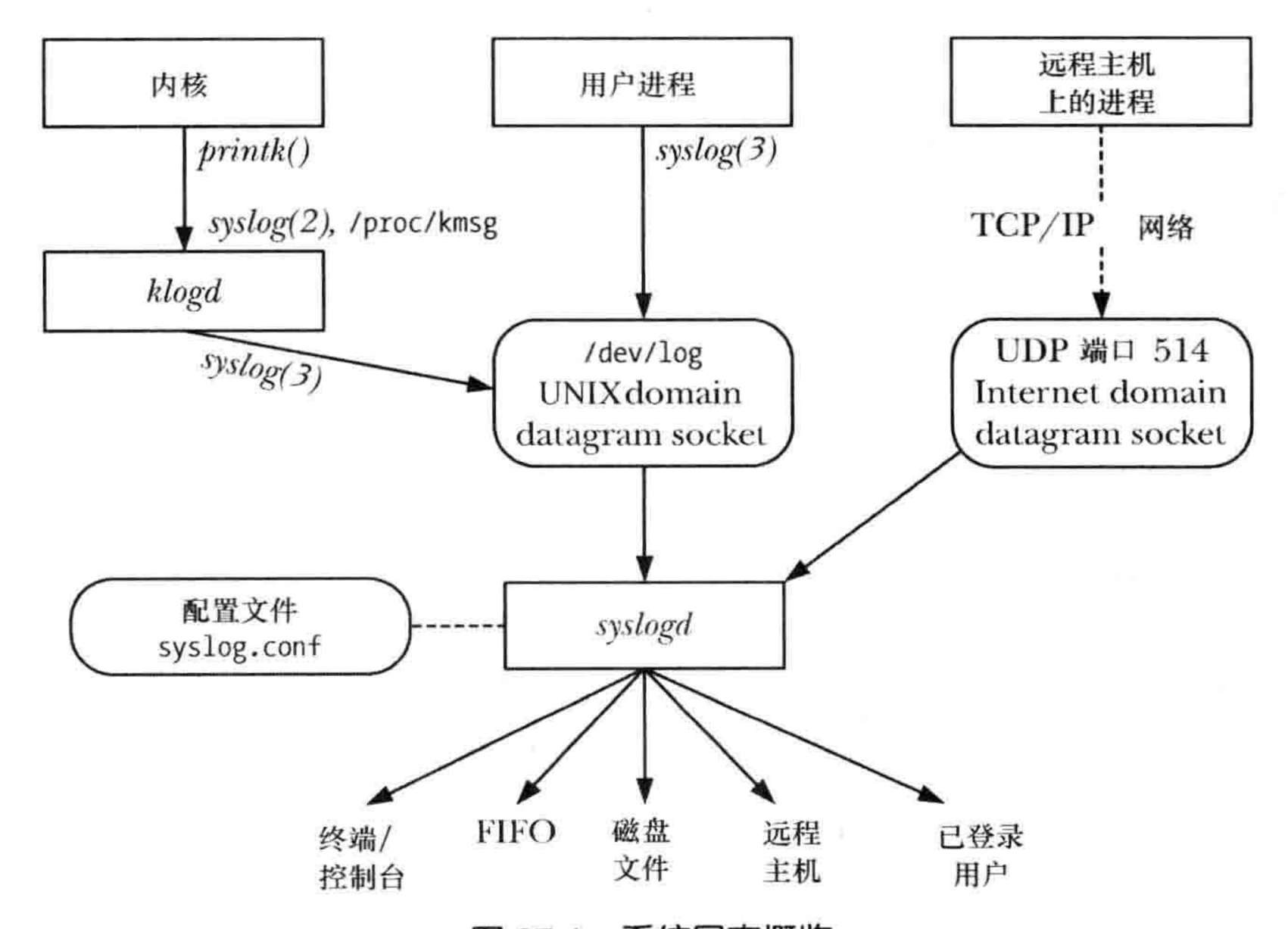


图 37-1：系统日志概览

syslog 工具有两个主要组件：syslogd daemon 和 syslog(3)库函数。

System Log daemon syslogd 从两个不同的源接收日志消息：一个是 UNIX domain socket

/dev/log，它保存本地产生的消息；另一个是 Internet domain socket（UNP 端口 514，如果启用的话），它保存通过 TCP/IP 网络发送的消息。（在其他一些 UNIX 实现中，syslog socket 位于/var/run/log。）

每条由 syslogd 处理的消息都具备几个特性，其中包括一个 facility，它指定了产生消息的程序类型；还有一个是 level，它指定了消息的严重程度（优先级）。syslogd daemon 会检查每条消息的 facility 和 level，然后根据一个相关配置文件/etc/syslog.conf 中的指令将消息传递到几个可能目的地中的一个。可能的目的地包括终端或虚拟控制台、磁盘文件、FIFO、一个或多个（或所有）登录过的用户以及位于另一个系统上的通过 TCP/IP 网络连接的进程（通常是另一个 syslogd daemon）。（将消息发送到另一个系统上的进程有助于通过将多个系统中的日志信息集中到一个位置以降低管理负担。）一条消息可以被发送到多个目的地（或不发送到任何目的地），具备不同的 facility 和 level 组合的消息可以被发送到不同的目的地或不同的目的地实例（即不同的控制台、不同的磁盘文件等）。

通过 TCP/IP 网络将 syslog 消息发送到另一个系统还有助于发现系统非法入侵。非法入侵通常会在系统日志中留下踪迹，但攻击者通常会删除日志记录以掩盖他们的行为。有了远程日志记录之后，攻击者就需要侵入另一个系统才能删除日志记录。

通常，任意进程都可以使用 syslog(3)库函数来记录消息。这个函数会使用传入的参数以标准的格式构建一条消息，然后将这条消息写入/dev/log socket 以供 syslogd 读取，本章稍后就会介绍这个函数。

/dev/log 中的消息的另一个来源是 Kernel Log daemon klogd，它会收集内核日志消息（内核使用 printk()函数生成的消息）。这些消息的收集可以通过两个等价的 Linux 特有的接口中的一个来完成（即/proc/kmsg 文件和 syslog(2)系统调用），然后使用 syslog(3)库函数将它们写入/dev/log。

尽管 syslog(2)和 syslog(3)的名称相同，但它们执行的任务是不同的。glibc 提供了一个调用 syslog(2)的接口，其名称为 klogctl()。除非特别指出，本节中的 syslog()指的是 syslog(3)。

syslog 工具原先出现在 4.2BSD 中，但现在几乎所有的 UNIX 实现都提供了这个工具。SUSv3 对 syslog(3)和相关函数进行了标准化，但并没有规定 syslogd 的实现和操作以及 syslog.conf 文件的格式。Linux 中 syslogd 的实现与它原先在 BSD 的实现的不同之处在于 Linux 允许对在 syslog.conf 中指定的消息处理规则进行一些扩展。

37.5.2 syslog API

syslog API 由以下三个主要函数构成。

- openlog()函数为后续的的 syslog()调用建立了默认设置。syslog()的调用是可选的，如果省略了这个调用，那么就会使用首次调用 syslog()时采用的默认设置来建立到日志记录工具的连接。
- syslog()函数记录一条日志消息。
- 当完成日志记录消息之后需要调用 closelog()函数拆除与日志之间的连接。

所有这些函数都不会返回一个状态值，这是因为系统日志服务应该总是处于可用状态（系统管理员应该在服务不可用时立即能发现这个问题）。此外，如果在系统记录日志的过程中发生了一个错误，应用程序通常也无法做更多的事情来报告这个错误。

更多资源请访问稀酷客(www.ckook.com)

GNU C 库还提供了函数 void vsyslog(int priority, const char*format, va_list args)。这个函数所做的工作与 syslog()一样，但接收之前由 stdarg(3) API 处理的一个参数列表。（因此 vsyslog()之于 syslog()就像 vprintf()之于 printf()。）SUSv3 并没有规定 vsyslog()，并且所有的 UNIX 实现都没有提供这个函数。

建立一个到系统日志的连接

openlog()函数的调用是可选的，它建立一个到系统日志工具的连接并为后续的 syslog()调用设置默认设置。

```
#include <syslog.h>

void openlog(const char *ident, int log_options, int facility);
```

ident 参数是一个指向字符串的指针，syslog()输出的每条消息都会包含这个字符串，这个参数的取值通常是程序名。注意 openlog()仅仅是复制了这个指针的值。只要应用程序后面会继续调用 syslog()，那么就应该确保不会修改所引用的字符串。

如果 ident 的值为 NULL，那么与其他一些实现一样，glibc syslog 实现会自动将程序名作为 ident 的值。但 SUSv3 并没有要求实现这个功能，一些实现也没有提供这个功能。可移植的应用程不应该依赖于这个功能。

传入 openlog()的 log_options 参数是一个位掩码，它是下面几个常量之间的 OR 值。

LOG_CONS

当向系统日志发送消息发生错误时将消息写入到系统控制台（/dev/console）。

LOG_NDELAY

立即打开到日志系统的连接（即底层的 UNIX domain socket, /dev/log）。在默认情况下（LOG_ODELAY），只有在首次使用 syslog()记录消息的时候才会打开连接。O_NDELAY 标记对于那些需要精确控制何时为/dev/log 分配文件描述符的程序来讲是比较有用的，如调用 chroot()的程序就有这样的要求。在调用 chroot()之后，/dev/log 路径名将不再可见，因此在 chroot()之前需要调用一个指定了 LOG_NDELAY 的 openlog()。tftpd daemon（Trivial File Transfer）就因为上述的原因而使用了 LOG_NDELAY。

LOG_NOWAIT

不要 wait()被创建来记录日志消息的子进程。在那些创建子进程来记录日志消息的实现上，当调用者创建并等待子进程时就需要使用 LOG_NOWAIT 了，这样 syslog()就不会试图等待已经被调用者销毁的子进程。在 Linux 上，LOG_NOWAIT 不起任何作用，因为在记录日志消息时不会创建子进程。

LOG_ODELAY

这个标记的作用与 LOG_NDELAY 相反——连接到日志系统的操作会被延迟至记录第一条消息时。这是默认行为，因此无需指定这个标记。

LOG_PERROR

将消息写入标准错误和系统日志。通常，daemon 进程会关闭标准错误或将其重定向到

更多资源请访问稀酷客(www.ckook.com)

/dev/null，这样 LOG_PERROR 就没有用了。

LOG_PID

在每条消息中加上调用者的进程 ID。在一个创建多个子进程的服务器中使用 LOG_PID 有助于区分哪个进程记录了某条特定的消息。

SUSv3 规定了上面除 LOG_PERROR 之前的所有常量，但很多其他（不是全部）UNIX 实现都定义了 LOG_PERROR 常量。

传入 openlog()的 facility 参数指定了后续的 syslog()调用中使用的默认的 facility 值。表 37-1 列出了这个参数的可取值。

表 37-1：openlog()的 facility 值和 syslog()的 priority 参数

值	描　　述	SUSv3
LOG_AUTH	安全和验证消息（如 su）	●
LOG_AUTHPRIV	私有的安全和验证消息	
LOG_CRON	来自 cron 和 at daemons 的消息	●
LOG_DAEMON	来自其他系统 daemon 的消息	●
LOG_FTP	来自 ftp daemon 的消息（ftpd）	
LOG_KERN	内核消息（用户进程无法生成此类消息）	●
LOG_LOCAL0	保留给本地使用（包括 LOG_LOCAL1 到 LOG_LOCAL7）	●
LOG_LPR	来自行打印机系统的消息（lpr、lpd、lpc）	●
LOG_MAIL	来自邮件系统的消息	●
LOG_NEWS	与 Usenet 网络新闻相关的消息	●
LOG_SYSLOG	来自 syslogd daemon 的消息	
LOG_USER	用户进程（默认值）生成的消息	●
LOG_UUCP	来自 UUCP 系统的消息	●

表 37-1 中列出的 facility 值的大部分都在 SUSv3 中进行了定义，如表中的 SUSv3 列所示，但 LOG_AUTHPRIV 和 LOG_FTP 只出现在了一些 UNIX 实现中，LOG_SYSLOG 则在大多数实现中都存在。当需要将包含密码或其他敏感信息的日志消息记录到一个与 LOG_AUTH 指定的位置不同的位置上时，LOG_AUTHPRIV 值是比较有用的。

LOG_KERN facility 值用于内核消息。用户空间的程序是无法用这个工具记录日志消息的。LOG_KERN 常量的值为 0。如果在 syslog()调用中使用了这个常量，那么 0 被翻译成了“使用默认的级别”。

记录一条日志消息

要写入一条日志消息可以调用 syslog()。

```
#include <syslog.h>

void syslog(int priority, const char *format, ...);
```

priority 参数是 facility 值和 level 值的 OR 值。facility 表示记录日志消息的应用程序的类别，其取值为表 37-1 中列出的值中的一个。如果省略了这个参数，那么 facility 的默认值为前面一个 openlog()调用中指定的 facility 值，或者当那个调用中也省略了 facility 值的话为

更多资源请访问稀酷客(www.ckook.com)

LOG_USER。level 表示消息的严重程度，其取值为表 37-2 中列出的值中的一个。这张表中列出的所有值都在 SUSv3 进行了定义。

表 37-2：syslog()中 priority 参数的 level 值（严重性从最高到最低）

值	描　述
LOG_EMERG	紧急或令人恐慌的情况（系统不可用了）
LOG_ALERT	需要立即处理的情况（如破坏了系统数据库）
LOG_CRIT	关键情况（如磁盘设备发生错误）
LOG_ERR	常规错误情况
LOG_WARNING	警告
LOG_NOTICE	可能需要特殊处理的普通情况
LOG_INFO	情报性消息
LOG_DEBUG	调试消息

另一个传入 syslog()的参数是一个格式字符串以及相应的参数，它们与传入 printf()中的参数是一样的，但与 printf()不同的是这里的格式字符串不需要包含一个换行字符。此外，格式字符串还可以包含双字符序列%m，在调用的时候这个序列会被与当前的 errno 值对应的错误字符串（即等价于 strerror(errno)）所替换。

下面的代码演示了 openlog()和 syslog()的用法。

```
openlog(argv[0], LOG_PID | LOG_CONS | LOG_NOWAIT, LOG_LOCAL0);
syslog(LOG_ERROR, "Bad argument: %s", argv[1]);
syslog(LOG_USER | LOG_INFO, "Exiting");
```

由于在第一个 syslog()调用中并没有指定 facility，因此将会使用 openlog()调用中的默认值（LOG_LOCAL0）。在第二个 syslog()调用中显式地指定了 LOG_USER 标记来覆盖 openlog()调用中设置的默认值。

在 shell 中可以使用 logger(1)命令来向系统日志中添加条目。这个命令允许指定与日志消息相关的 level（priority）和 ident（tag），更多细节可参考 logger(1)手册。SUSv3 规定了 logger 命令（并没有进行全面定义），大多数 UNIX 实现都实现了这个命令。

像下面这样使用 syslog()写入一些用户提供的字符串是错误的。

```
syslog(priority, user_supplied_string);
```

上面这段代码存在的问题是应用程序会面临所谓的格式字符串攻击。如果用户提供的字符串中包含格式指示符（如%s），那么结果将是不可预测的，从安全的角度来讲，这种结果可能是具有破坏性的。（这个结论也同样适用于传统的 printf()函数。）因此需要将上面的调用重写为下面这样。

```
syslog(priority, "%s", user_supplied_string);
```

关闭日志

当完成日志记录之后可以调用 closelog()来释放分配给/dev/log socket 的文件描述符。

```
#include <syslog.h>

void closelog(void);
```


更多资源请访问稀酷客(www.ckook.com)

由于 daemon 通常会持续保持与系统日志之间的连接的打开状态，因此通常会省略对 closelog()的调用。

过滤日志消息

setlogmask()函数设置了一个能过滤由 syslog()写入的消息的掩码。

```
#include <syslog.h>

int setlogmask(int mask_priority);
```

Returns previous log priority mask

所有 level 不在当前的掩码设置中的消息都会被丢弃。默认的掩码值允许记录所有的严重性级别。

宏 LOG_MASK()（在<syslog.h>中定义）会将表 37-2 中的 level 值转换成适合传入 setlogmask()的位值。如要丢弃除优先级为 LOG_ERR 以及以上之外的消息时可以使用下面的调用。

```
setlogmask(LOG_MASK(LOG_EMERG) | LOG_MASK(LOG_ALERT) |
           LOG_MASK(LOG_CRIT) | LOG_MASK(LOG_ERR));
```

SUSv3 规定了 LOG_MASK()宏。大多数 UNIX 实现（包括 Linux）还提供了标准中未规定的 LOG_UPTO()宏。它创建一个能过滤特定级别以及以上的所有消息的位掩码。使用这个宏能够将前面的 setlogmask()调用简化成下面这个。

```
setlogmask(LOG_UPTO(LOG_ERR));
```

37.5.3 /etc/syslog.conf 文件

/etc/syslog.conf 配置文件控制 syslogd daemon 的操作。这个文件由规则和注释（以#字符打头）构成。规则的形式如下所示。

facility.level	*action*

facility 和 level 组合在一起被称为选择器，因为它们选择了需应用规则的消息。这个字段是与表 37-1 和表 37-2 中的值对应的字符串。action 指定了与选择器匹配的消息被发送到何处。选择器和 action 之间用空白字符隔开，下面是一些示例。

```
*.err                          /dev/tty10
auth.notice                    root
*.debug;mail.none;news.none    -/var/log/messages
```

第一条规则表示来自所有工具（*）的 level 为 err（(LOG_ERR）或更高的消息应该被发送到/dev/tty10 控制台设备上。第二条规则表示来自验证工具（LOG_AUTH）的 level 为 notice（LOG_NOTICE）或更高的消息应该被发送到 root 登录的所有控制台和终端。如这个特别的规则允许一个登录的 root 用户立即看到失败的 su 尝试。

最后一条规则演示了规则语法中的几个高级特性。一个规则可以包含多个选择器，选择器之间用分号隔开。第一个选择器指定了所有的消息，它使用*通配符表示 facility 并将 level 的值指定为 debug，这意味着所有级别为 debug（最低的级别）以及更高的消息都会被记录下来。（在

更多资源请访问稀酷客(www.ckook.com)

Linux 以及其他一些 UNIX 实现中，可以将 level 指定为*，其含义与 debug 是一样的。但不是所有的 syslog 实现都支持这个特性。）通常，一个包含多个选择器的规则会匹配与其中任意一个选择器对应的消息，但当将 level 设置为 none 时则表示排除所有属于相应的 facility 的消息。因此这条规则将除来自 mail 和 news 工具的消息之外的所有消息发送到/var/log/messages 文件中。文件名前面的连接符（-）表示无需每次写入文件时都将文件同步到磁盘（参见 13.3 节）。这意味着写入操作将变得更快，但如果系统在写入之后崩溃的话可能会丢失一些数据。

每次修改 syslog.conf 文件之后都需要使用下面的方式让 daemon 根据这个文件重新初始化自身。

```
$ killall -HUP syslogd                    Send SIGHUP to syslogd
```

syslog.conf 规则语法的高级特性允许编写比前面介绍的更加强大的规则，更多细节可参考 syslog.conf(5)手册。

37.6 总结

daemon 是一个长时间运行并且没有控制终端的进程（即它运行在后台）。daemon 执行特定的任务，如提供一个网络登录工具或服务 Web 页面。一个程序要成为 daemon 需要按序执行一组步骤，包括调用 fork()和 setsid()。

daemon 应该在合适的地方正确地处理 SIGTERM 和 SIGHUP 信号。SIGTERM 信号的处理方式应该是按序关闭这个 daemon，而 SIGHUP 信号则提供了一种机制让 daemon 通过读取器配置文件并重新打开所使用的所有日志文件来重新初始化自身。

syslog 工具为 daemon（以及其他应用程序）提供了一种便捷的方式来将错误和其他消息记录到一个中心位置。这些消息由 syslogd daemon 处理，syslogd 会根据 syslogd.conf 配置文件中的指令来重新分发消息。可以将消息重新分发到几个目标上，包括终端、磁盘文件、登录的用户以及通过 TCP/IP 网络分发到远程主机上的进程中（通常是其他 syslogd daemon）。

更多信息

有关编写 daemon 的更多信息的最佳来源可能就是各种既有 daemon 的源代码。

37.7 习题

37-1： 编写一个使用 syslog(3)的程序（与 logger(1)类似）来将任意的消息写入到系统日志文件中。程序应该接收包含如记录到日志中的消息的命令行参数，同时应该允许指定消息的 level。

更多资源请访问稀酷客(www.ckook.com)

第38章

编写安全的特权程序

特权程序能够访问普通用户无法访问的特性和资源（文件设备等）。一个程序可以通过下面两种方式以特权方式运行。

- 程序在一个特权用户 ID 下启动，很多 daemon 和网络服务器通常以 root 身份运行，它们就属于这种类别。
- 程序设置了 set-user-ID 或 set-group-ID 权限位。当一个 set-user-ID（set-group-ID）程序被执行之后，它会将进程的有效用户（组）ID 修改为与程序文件的所有者（组）一样的 ID。（在 9.3 节中首次对 set-user-ID 和 set-group-ID 程序进行了介绍。）在本章中有时候会使用术语 set-user-ID-root 区分将超级用户权限赋给进程的 set-user-ID 程序与赋给进程另一个有效身份的程序。

如果一个特权程序包含 bug 或可以被恶意用户破坏，那么系统或应用程序的安全性就会受到影响。从安全的角度来讲，在编写程序的时候应该将系统受到安全威胁的可能性以及受到安全威胁时产生的损失降到最小。本章将对这些课题进行讨论，并提供了一组编写安全程序的推荐实践，同时介绍了在编写特权程序时应该避免的各种陷阱。

38.1 是否需要一个 Set-User-ID 或 Set-Group-ID 程序

有关编写 set-user-ID 和 set-group-ID 程序的最佳建议中的一条就是尽量避免编写这种程序。在执行一个任务时如果存在无需赋给程序权限的方法，那么一般来讲应该采用这种方法，因为这样就消除了发生安全性问题的可能。

有时候可以将需要权限才能完成的功能拆分到一个只执行单个任务的程序中，然后在需要的时候在子进程中执行这个程序。对于库来讲，这项技术是特别有用的。64.2.2 节中介绍的 pt_chown 程序就采用了这种技术。

即使有时候需要 set-user-ID 或 set-group-ID 权限，对于一个 set-user-ID 程序来讲也并不总是需要赋给进程 root 身份。如果赋给进程其他一些身份已经足够，那么就应该采用这种方法，因为以 root 权限运行可能会引起安全性问题。

假设一个 set-user-ID 程序需要允许用户更新一个它没有写权限的文件，那么解决这个问

更多资源请访问稀酷客(www.ckook.com)

题的一种更加安全的方式是为这个程序创建一个专用组账号（组 ID），然后将文件所属的组修改为那个组（即使得该组中的成员能够写入该文件），接着编写一个将进程的有效组 ID 设置为该专用组 ID 的 set-user-ID 程序。由于这个专用的组 ID 没有其他权限，因此能够极大地限制程序包含 Bug 或被破坏时所造成的损失。

38.2 以最小权限操作

set-user-ID（或 set-group-ID）程序通常只有在执行特定操作的时候才需要权限，因此在程序（特别是那些拥有超级用户权限的程序）执行其他工作时应该禁用这些权限，同时如果之后永远也不会请求这项权限时就应该永久删除这项权限。换句话说，程序应该总是使用完成当前所执行的任务所需的最小权限来操作，saved 的 set-user-ID 工具就是为此而设计的（参见 9.4 节）。

按需拥有权限

在 set-user-ID 程序中可以使用下面的 seteuid()调用序列来临时删除并在之后重新获取权限。

```
uid_t orig_euid;

orig_euid = geteuid();
if (seteuid(getuid()) == -1)            /* Drop privileges */
    errExit("seteuid");

/* Do unprivileged work */

if (seteuid(orig_euid) == -1)           /* Reacquire privileges */
    errExit("seteuid");

/* Do privileged work */
```

第一个调用使调用进程的有效用户 ID 变成其真实 ID。第二个调用将有效用户 ID 还原成 saved set-user-ID 程序中保存的值。

对于 set-group-ID 程序来讲，saved set-group-ID 会保存程序的初始有效组 ID，并且 setegid()可以用来删除和重新获取权限。第 9 章对在下面的建议中提到的 seteuid()、setegid()、以及类似的系统调用进行了介绍，表 9-1 对它们进行了总结。

最安全的作法是在程序启动的时候立即删除权限，然后在后面需要的时候临时重新获得这些权限。如果在某个特定的时刻之后永远不会再次请求权限时，那么程序应该删除这些权限，并通过确保 saved set-user-ID 的变更来保证程序无法再请求这些权限。这样就消除了通过第 38.9 节中介绍的栈崩溃技术来让程序重新要求权限的可能。

在无需使用权限时永久地删除权限

如果 set-user-ID 或 set-group-ID 程序完成了所有需要权限的任务，那么它应该永久地删除这些权限以消除任何由于程序中包含 bug 或其他意料之外的行为而可能引起的安全风险。永久删除权限是通过将所有进程用户（组）ID 重置为真实（组）ID 来完成的。

对于一个当前有效用户 ID 为 0 的 set-user-ID-root 程序来讲，可以使用下面的代码来重置所有的用户 ID。

更多资源请访问稀酷客(www.ckook.com)

```
if (setuid(getuid()) == -1)
    errExit("setuid");
```

当调用进程的当前有效用户 ID 为非零时，上面的代码不会重置 saved set-user-ID：当在一个有效用户 ID 不为零的程序中发起调用时，setuid()只会修改有效用户 ID（参见 9.7.1 节）。换句话说，在一个 set-user-ID-root 程序中，下面的调用序列不会永久地删除用户 ID 0。

```
/* Initial UIDs:    real=1000 effective=0 saved=0 */

/* 1. Usual call to temporarily drop privilege */

orig_euid = geteuid();
if (seteuid(getuid() == -1)
    errExit("seteuid");

/* UIDs changed to: real=1000 effective=1000 saved=0 */

/* 2. Looks like the right way to permanently drop privilege (WRONG!) */

if (setuid(getuid() == -1)
    errExit("setuid");

/* UIDs unchanged:  real=1000 effective=1000 saved=0 */
```

相反，在永久删除权限之前必须要重新获取权限，这可以通过将下面的调用插入到前面的第 1 步和第 2 步之间即可。

```
if (seteuid(orig_euid) == -1)
    errExit("seteuid");
```

另一方面，如果一个非 root 用户拥有了 set-user-ID 程序，那么由于 setuid()不足以修改 set-user-ID 标识符，因此必须要使用 setreuid()或 setresuid()来永久地删除特权标识符。例如，可以使用 setreuid()来获得预期的结果，如下所示。

```
if (setreuid(getuid(), getuid()) == -1)
    errExit("setreuid");
```

上面的代码依赖于 Linux 实现中的 setreuid()特性：如果第一个参数（ruid）不是−1，那么 saved set-user-ID 也会被设置成（新）有效用户 ID。SUSv3 并没有规定这个特性，但很多其他实现的行为方式与 Linux 是一样的。

在 set-group-ID 程序中永久地删除一个特权组 ID 同样必须要使用 setregid()或 setresgid()系统调用，因为当程序的有效用户 ID 不为零时，setgid()只会修改调用进程的有效组 ID。

修改进程身份信息的注意事项

前面几小节介绍了临时和永久删除权限的技术。下面介绍有关使用这些技术时的一些注意事项。

- 一些修改进程身份信息的系统调用在不同系统上的语义是不同的。此外，此类系统调用中的一些在调用者是特权进程时（有效用户 ID 为 0）与非特权进程时表现出来的语义是不同的。更多细节信息可参考第 9 章，特别是 9.7.4 节。由于存在这些差异，[Tsafrir et al., 2008]建议应用程序应该使用系统特有的非标准系统调用来修改进程的身份信息，因为在很多情况下，这些非标准系统调用比与其对应的标准系统调用提供了更加简单和一致的语义。在 Linux 上，这表示需要使用 setresuid()和 setresgid()来修改用户和组身份信息。尽管并不是所有的系统都提供了这些系统调用，但使用它们会降低发生错

更多资源请访问稀酷客(www.ckook.com)

误的可能性。（[Tsafrir et al., 2008]提供了一个函数库，其中的函数会使用各个平台上可用的最佳接口来修改身份信息。）

- 在 Linux 上，即使调用者的有效用户 ID 为 0，修改身份信息的系统调用在程序显式地操作其能力时也可能会表现出意料之外的行为。如，如果禁用了 CAP_SETUID 能力，那么修改进程用户 ID 将会失败，甚至更糟的是，它会毫无征兆地只修改其中一些需修改的用户 ID。
- 由于存在前面两种可能性，因此强烈建议在实践中（参见[Tsafrir et al., 2008]）不仅需要检查一个修改身份信息的系统调用是否成功，还需要验证修改行为是否如预期的那样。例如如果使用 seteuid()临时删除或重新请求一个特权用户 ID，那么接着应该使用一个 geteuid()调用来验证有效用户 ID 是否为预期值。类似地，如果永久删除了一个特权用户 ID，那么接着应该验证真实用户 ID、有效用户 ID 以及 saved set-user-ID 已经被成功地修改为非特权用户 ID。遗憾的是，虽然存在获取真实和有效 ID 的标准系统调用，但不存在获取 saved set IDs 的标准系统调用。Linux 和其他一些系统提供了 getresuid()和 getresgid()来解决这个问题，在其他一些系统上则可能需要使用诸如解析/proc 文件中的信息之类的技术来解决这个问题。
- 一些身份信息的变更只能由有效用户 ID 为 0 的进程来完成。因此在修改多个 ID 时——辅助组 ID、组 ID 和用户 ID——先删除特权 ID，最后再删除特权有效用户 ID。相应地，在提升特权 ID 时应该先提升特权有效用户 ID。

38.3 小心执行程序

当一个特权程序通过 exec()直接或通过 system()、popen()以及类似的库函数间接地执行另一个程序时就需要小心处理了。

在执行另一个程序之前永久地删除权限

如果一个 set-user-ID（或 set-group-ID）程序执行了另外一个程序，那么就应该要确保所有的进程用户（组）ID 被重置为真实用户（组）ID，这样新程序在启动时就不会拥有权限，并且也无法重新请求这些权限。完成这一任务的一种方式是在执行 exec()之前使用第 38.2 节中介绍的技术来重置所有的 ID。

在 exec()调用之前调用 setuid(getuid())能够取得同样的结果。虽然 setuid()调用只修改了那些有效用户 ID 不为零的进程的有效用户 ID，但权限还是会被删除，因为成功的 exec()调用会将有效用户 ID 复制到 saved set-user-ID。（如果 exec()执行失败了，那么 saved set-user-ID 不会发生改变。这对于在 exec()失败时需要执行其他特权工作的程序来讲是比较有用的。）

在 set-group-ID 程序中也可以采用类似的方式（即 setgid(getgid()），因为成功的 exec()调用也会将有效组 ID 复制到 saved-setgroup-ID。

现在假设用户 ID 200 拥有一个 set-user-ID 程序，当 ID 为 1000 的用户执行这个程序时，结果进程的用户 ID 如下所示。

```
real=1000 effective=200 saved=200
```

如果这个程序执行了 setuid(getuid())调用，那么进程的用户 ID 将会变成如下。

```
real=1000 effective=1000 saved=200
```

当进程执行一个非特权程序时，进程的有效用户 ID 会被复制到 saved set-user-ID，从而导

更多资源请访问稀酷客(www.ckook.com)

致进程的用户 ID 变为：

```
real=1000 effective=1000 saved=1000
```

避免执行一个拥有权限的 shell（或其他解释器）

运行于用户控制之下的特权程序永远都不该直接或间接（通过 system(), popen(), execlp(), execvp()或其他类似的库函数）地执行 shell。shell（以及其他不受限的解释器，如 awk）的复杂性和强大功能意味着几乎不可能消除所有的安全漏洞，即使被执行的 shell 不允许交互式访问。其可能引起的风险是用户可能能够在进程的有效用户 ID 下执行任意的 shell 命令。如果必须要执行 shell，那么就需要确保在执行之前永久地删除权限。

在 27.6 节有关 system()的讨论中介绍的安全漏洞就是执行 shell 时可能引起的一个漏洞。

一些 UNIX 实现在将权限位应用于解释器脚本时会采用 set-user-ID 和 set-group-ID 的权限位（参见 27.3），因此当运行脚本时，执行脚本的进程的身份是其他（特权）用户。由于存在前面描述的安全风险，Linux 与其他一些 UNIX 实现一样，在执行脚本时会毫无征兆地忽略 set-user-ID 和 set-group-ID 权限位。即使在允许 set-user-ID 和 set-group-ID 脚本的实现上也应该避免使用它们。

在 exec()之前关闭所有用不到的文件描述符

在 27.4 节中提到过在默认情况下，在 exec()调用之间文件描述符会保持在打开状态。特权进程可能会打开普通进程无法访问的文件，这种打开的文件描述符表示一种特权资源。在调用 exec()之前应该关闭这种文件描述符，这样被执行的程序就无法访问相关的文件了。完成这个任务既可以通过显式关闭文件描述符，也可以设置程序的 close-on-exec 标记（参见 27.4）。

38.4 避免暴露敏感信息

当一个程序读取密码或其他敏感信息时应该在执行完所需的处理之后立即从内存中删除这些信息。（在 8.5 节中给出了一个这样的例子。）在内存中保留这些信息是一种安全隐患，其原因如下。

- 包含这些数据的虚拟内存页面可能会被换出（除非使用 mlock()或类似的函数将它们锁在内存中），这样交换区域中的数据可能会被一个特权程序读取。
- 如果进程接收到了一个能导致它产生一个核心 dump 文件的信号，那么就有可能会从该文件中获取这类信息。

从上面的最后一点来讲，编写程序时应遵循的一个通用原则是安全程序应该避免产生核心 dump。一个程序可以使用 setrlimit()将 RLIMIT_CORE 资源限制设置为 0 来防止核心 dump 文件的创建（参见 36.3 节）。

在默认情况下，Linux 不允许 set-user-ID 程序在收到信号时执行一个核心 dump（参见 22.1 节），即使程序已经删除了所有的权限。但其他 UNIX 实现可能并没有提供这个安全特性。

更多资源请访问稀酷客(www.ckook.com)

38.5 确定进程的边界

本节将介绍各种限制程序以控制程序发生安全问题时所造成的损失的方法。

考虑使用能力

Linux 能力模型将传统的 all-or-nothing UNIX 权限模型划分为一个个被称为能力的单元。一个进程能够独立地启用或禁用单个能力。通过只启用进程所需的能力使得程序能够在不拥有完整的 root 权限的情况下运行。这样就降低了程序发生安全问题时造成损失的可能性。

此外，使用能力和 securebits 标记可以创建只拥有有限的一组权限但无需属于 root 的进程（即进程的所有用户 ID 都不为零）。这样的进程无法使用 exec()来重新获取所有能力。在第 39 章中将会介绍能力和 securebits 标记。

考虑使用一个 chroot 监牢

在特定情况下一项有用的安全技术是建立一个 chroot 监牢来限制程序能够访问的一组目录和文件。（还需确保调用 chdir()来将进程的当前工作目录改为监牢中的一个位置。）但 chroot 监牢不足以限制一个 set-user-ID-root 程序（参见 18.12 节）。

> 除了使用 chroot 监牢之外还可以使用一个虚拟服务器，它是实现于虚拟内核之上的一个服务器。由于每个虚拟内核与运行于同一硬件上的其他虚拟内核是相互隔离的，因此虚拟服务器比 chroot 监牢更加安全和灵活。（其他一些现代操作系统还提供了自己的虚拟服务器实现。）Linux 上最早的虚拟化实现是 User-Mode Linux（UML），它是 Linux 2.6 内核的标准组成部分。在 http://user-mode-linux.sourceforge.net/上可以找到有关 UML 的更多信息。最新的虚拟内核项目包括 Xen（http://www.cl.cam.ac.uk/Research/SRG/netos/xen/）和 KVM（http://kvm.qumranet.com/）。

38.6 小心信号和竞争条件

用户可以向他启动的 set-user-ID 程序发送任意信号，其发送时间和发送频率也是任意的。当信号在程序执行过程中的任意时刻发送时需要考虑可能出现的竞争条件。在程序中合适的地方应该捕获、阻塞或忽略信号以防止可能存在的安全性问题。此外，信号处理器的设计应该尽可能简单以降低无意中创建竞争条件的风险。

这个问题与停止进程的信号（如 SIGTSTP 和 SIGSTOP）特别相关。存在问题的场景如下所示。

1. 一个 set-user-ID 程序确定与其运行时环境有关的一些信息。
2. 用户需要停止运行程序的进程和修改运行时环境的细节。这样的变更可能包括修改文件的权限、改变符号链接的目标以及删除程序所依赖的文件。
3. 用户使用 SIGCONT 信号恢复进程。这时程序会假设原先的运行时环境没有发生变化并继续执行，但其实运行时环境已经发生了变化，因此在这种假设可能会破坏系统的安全性。

这里描述的情况确实只是检查时间（time-of-check）和使用时间（time-of-use）竞争条件的一种特殊情况。特权进程应该避免执行依赖于之前成立但现在已经不再成立的条件的操作（具体示例可参考第 15.4.4 节中对 access()系统调用的讨论）。即使当用户无法向进程发送信号时也应该遵循这个指南。停止一个进程的能力仅仅允许一个用户扩大检查时间和使用时间之

更多资源请访问稀酷客(www.ckook.com)

间的时间间隔。

> 虽然通过一次尝试就在检查时间和使用时间之间停止一个进程是比较困难的，但恶意用户可以重复地执行一个 set-user-ID 程序并使用另一个程序或一个 shell 脚本重复地向 set-user-ID 程序发送停止信号并修改运行时环境。

38.7 执行文件操作和文件 I/O 的缺陷

如果一个特权进程需要创建一个文件，那么必须要小心处理那个文件的所有权和权限以确保文件不存在被恶意操作攻击的风险点，不管这个风险点有多小。因此需遵循下列指南。

- 需要将进程的 umask（参见 15.4.6 节）设置为一个能确保进程永远无法创建公共可写的文件的值，否则恶意用户就能修改这些文件了。
- 由于文件的所有权是根据创建进程的有效用户 ID 来确定的，因此可能需要使用 seteuid()或 setreuid()来临时地修改进程的身份信息以确保新创建的文件不会属于错误的用户。由于文件的组所有权可能会根据进程的有效组 ID（参见 15.3.1 节）来确定，因此类似的规则也同样适用于 set-group-ID 程序，并且可以使用相应的组 ID 调用来避免此类问题的发生。（严格来讲，在 Linux 上，新文件的所有者是由进程的文件系统用户 ID 来确定的，这个 ID 值通常与进程的有效用户 ID 值是一样的，具体可参见 9.5 节）。
- 如果一个 set-user-ID-root 程序必须要创建一个一开始由其自己拥有但最终由另一个用户拥有的文件，那么所创建的文件在一开始应该不对其他用户开放写权限，这可以通过向 open()传入一个合适的 mode 参数或在调用 open()之前设置进程的 umask 完成。之后，程序可以使用 fchown()修改文件的所有权，然后根据需要使用 fchmod()修改文件的权限。这里的关键点是 set-user-ID 程序应该确保它永远不会创建一个由程序所有者拥有但允许其他用户写入（即使这项权限只开放了一瞬间）的文件。
- 在打开的文件描述符上检查文件的特性（如在 open()之后调用 fstat()），而不是检查与一个路径名相关联的特性后再打开文件（如在 stat()之后调用 open()）。后一种方法存在使用时间和检查时间的问题。
- 如果一个程序必须要确保它自己是文件的创建者，那么在调用 open()时应该使用 O_EXCL 标记。
- 特权进程应该避免创建或依赖像/tmp 这样的公共可写的目录，因为这样程序就容易受到那些试图创建文件名与特权程序预期一致的非授权文件的恶意攻击。一个必须要在某个公共可写的目录中创建文件的程序应该至少要使用诸如 mkstemp()之类的函数（参见 5.12 节）确保这个文件的文件名不会不可预测。

38.8 不要完全相信输入和环境

特权程序应该避免完全信任输入和它们所运行的环境。

不要信任环境列表

set-user-ID 和 set-group-ID 程序不应该假设环境变量的值是可靠的，特别是 PATH 和 IFS

更多资源请访问稀酷客(www.ckook.com)

两个变量。

PATH 确定了 shell（和 system() and popen()）以及 execlp()和 execvp()在何处搜索程序。恶意用户可以改变PATH的值以欺骗set-user-ID程序使它在使用其中一个函数时会导致在拥有权限的情况下执行任意一个程序。在使用这些函数时应该将 PATH 值设置为一个可信的目录列表（更好的做法是在执行程序时指定绝对路径名）。但正如之前已经提过的，在执行 shell 或使用前面提到的函数之前最好先删除权限。

IFS 指定了 shell 解释器用来分隔命令行中的单词的分隔符。应该将这个变量设置为任意一个空字符串，表示 shell 只会把空白字符当成单词分隔符。一些 shell 在启动的时候总是会这样设置 IFS 的值。（27.6 节描述了老式 Bourne shell 中与 IFS 相关的一个漏洞。）

在某些情况中，特别是在执行其他程序或调用可能会受到环境变量设置影响的库时，最安全的方式是删除整个环境列表（参见 6.7 节），然后使用已知的安全值来还原所选中的环境变量。

防御性地处理不可信用户的输入

特权程序应该在根据来自不可信源的输入采取动作之前小心地验证这些输入。这种验证包括校验数字是否位于接受范围之内、字符串的长度是否位于接受范围之内以及是否由允许的字符构成等。需要采取此类验证措施的输入包括用户创建的文件、命令行参数、交互式输入、CGI 输入、电子邮件消息、环境变量、不可信用户能够访问的进程间通信通道（FIFO、共享内存等）以及网络包。

避免对进程的运行时环境进行可靠性假设

set-user-ID 程序应该避免假设其初始的运行环境是可靠的。如标准输入、输出或错误可能会被关闭。（这些描述符可能是在执行这个 set-user-ID 程序的程序中被关闭的。）这样，当打开一个文件时可能会无意中复用描述符 1（假设），从而导致程序认为正在往标准输出中输出数据，但实际上是在往一个由其打开的文件中写入数据。

还有很多情况需要考虑。如一个进程可能会耗光各种资源限制，如能够创建的进程数限制、CPU 时间资源限制或文件大小资源限制，从而导致各种系统调用会失败或各种信号的生成。恶意用户可能会故意攻击系统使得资源耗尽以便破坏程序。

38.9 小心缓冲区溢出

当输入值或复制的字符串超出分配的缓冲区空间时就需要小心缓冲区溢出了。永远不要使用 gets()，在使用诸如 scanf()、sprintf()、strcpy()以及 strcat()时需要谨慎（如使用 if 语句防止在使用这些函数时造成缓冲区溢出）。

恶意用户可以通过诸如缓冲区溢出（也被称为栈粉碎）之类的技术将精心编写的字节放入一个栈帧中以强制特权程序执行任意代码。（网上的几个资源站点解释了栈粉碎的细节，读者还可以参考[Erickson, 2008]和[Anley, 2007]）。从 CERT（http://www.cert.org/）和 Bugtraq（http://www.securityfocus.com/）上发表的咨询报告的频率上明显可以看出在一个计算机系统上，缓冲区溢出可能是引起安全性问题的最常见的原因了。缓冲区溢出对于网络服务器来讲特别具有危害性，因为它们使得系统向网络上任意地方的远程攻击打开了大门。

更多资源请访问稀酷客(www.ckook.com)

为了使栈崩溃变得更加困难——特别是使得在远程主机上使用此类攻击手段攻击网络服务器时更加耗时——从内核 2.6.12 开始，Linux 实现了地址空间随机化。这项技术使得栈的位置能够在虚拟内存最前面的 8M 空间内随机变动。此外，如果软 RLIMIT_STACK 限制有限并且 Linux 特有的/proc/sys/vm/legacy_va_layout 不包含 0，那么内存映射的位置也可以随机化。

最新的 x86-32 架构为将页表变成 NX（“no execute”）提供了硬件支持。这个特性用来防止执行栈上的代码，从而使得栈奔溃变得更加困难。

上面提到的很多函数都存在安全的版本——如 snprintf()、strncpy()以及 strncat()——允许调用者指定需复制的最大字符数。这些函数考虑到了最大数量以防止目标缓冲区的溢出。一般来讲，最好使用这些函数，但使用时仍然需要小心，特别需要注意下列事项。

- 对于其中的大多数函数来讲，如果到达了指定的最大值，那么源字符串的截断部分会被放到目标缓冲区中。由于这样的截断字符串对于程序来讲可能是毫无意义的，因此调用者必须要检查字符串是否发生了截断（如使用 snprintf()的返回值）并且在发生截断的时候采取必要的措施。
- 使用 strncpy()对性能会有影响。如在 strncpy(s1, s2, n)调用中，如果 s2 指向的字符串的长度小于 n 字节，那么补全的 null 字节就会被写入 s1 以确保总共写入了 n 字节。
- 如果传入 strncpy()的最大大小不足以容纳结尾的 null 字符，那么目标字符串就会成为不以 null 结尾的字符串。

一些 UNIX 实现提供了 strlcpy()函数，它接收长度参数 n，最多将 n – 1 个字节复制到目标缓冲区中并且总是会在缓冲区的结尾加上 null 字符。但 SUSv3 并没有规定这个函数，并且 glibc 也没有实现这个函数。此外，如果调用者没有小心检查字符串长度，那么这个函数在解决一个问题（缓冲区溢出）的同时又引入了另一个问题（毫无征兆地丢弃数据）。

38.10 小心拒绝服务攻击

随着基于 Internet 服务的增长，系统相应受到远程拒绝服务（DoS）的攻击的可能性也在增长。这些攻击通过向服务器发送能导致其崩溃的错误数据或使用虚假请求给服务器增加过量的负载使得系统无法向合法用户提供正常的服务。

本地的拒绝服务攻击也是有可能的。最知名的例子是当用户运行一个简单的 fork 炸弹（一个重复执行 fork 的程序，因此会消耗系统中的所有进程槽）。但引起本地拒绝服务的源头更加容易确定，因此一般来讲可以通过合适的物理和密码安全措施来避免。

处理错误的请求是比较直观的——服务器应该严格地像上面描述的那样检查其输入以避免缓冲区溢出。

超负荷攻击更加难以处理。由于服务器无法控制远程客户端的行为以及它们提交请求的速率，因此这样的攻击几乎无法防止。（服务器甚至无法确定攻击的真正源头，因为网络包的源 IP 地址是可以伪造的。此外，分布式攻击可能会募集不知情的中间主机向目标系统发起攻击。）不过，仍然可以采取各种措施把超负荷攻击的风险和损失降到最小。

更多资源请访问稀酷客(www.ckook.com)

- 服务器应该执行负载控制，当负载超过预先设定的限制之后就丢弃请求。这可能会导致丢弃合法的请求，但能够防止服务器和主机机器的负载过大。资源限制和磁盘限额的使用也有助于限制过量的负载。（更多有关磁盘限额的信息可参考 http://sourceforge.net/projects/linuxquota/。）
- 服务器应该为与客户端的通信设置超时时间，这样如果客户端不响应（可能是故意的），那么服务器也不会永远地等待客户端。
- 在发生超负荷时，服务器应该记录下合适的信息以便系统管理员得知这个问题。（但日志记录应该也是有限制的，这样日志记录本身不会给系统增加过量的负载。）
- 服务器程序在碰到预期之外的负载时不应该崩溃。如应该严格进行边界检查以确保过多的请求不会造成数据结构溢出。
- 设计的数据结构应该能够避免算法复杂度攻击。如二叉树应该是平衡的，并且在常规负载下应该能提供可接受的性能。但攻击者可能会构造一组会导致树不平衡（在最坏的情况下等价于一个链表）的输入，从而降低性能。[Crosby & Wallach, 2003]详细描述了此类攻击的性质并讨论了能够用来避免此类攻击的数据结构技术。

38.11 检查返回状态和安全地处理失败情况

特权程序应该总是检查系统调用和库函数调用是否成功以及它们是否返回了预期的值。（当然所有程序都应该这么做，但这一点对于特权程序来讲特别重要。）各种各样的系统调用都可能会失败，即使程序以 root 的身份运行。如当达到系统的进程数限制时 fork()调用就会失败，对只读文件系统调用 open()以获取写权限时会失败，或者当目标目录不存在时调用 chdir()会失败。

即使系统调用成功了也有必要检查其结果。如，在需要的时候，特权程序应该检查成功的 open()调用没有返回三个标准文件描述符 0、1 或 2 中的某个。

最后，如果特权程序碰到了未知情形，那么恰当的处理方式通常是终止执行或如果是服务器的话就丢弃客户端请求。试图修复未知的问题通常要求满足一些前提条件，但并不是所有的场景都满足这些前提条件，当不满足时就可能会产生安全漏洞。在这种情况下，更安全的做法是让程序终止或让服务器把信息记录下来并丢弃客户端的请求。

38.12 总结

特权程序能够访问普通进程无法访问的系统资源。如果这种程序被破坏了，那么系统的安全性就会受到影响。本章给出了编写特权程序的一组指南，这些指南的目标包括两个方面：将特权程序被破坏的可能性降到最低和当特权程序被破坏时所造成的损失降到最小。

更多信息

[Viega & McGraw, 2002]介绍了与设计和实现安全软件相关的各项课题。[Garfinkel et al., 2003]介绍了与 UNIX 系统的安全以及安全程序设计技术有关的信息。[Bishop,2005]深入介绍了计算机安全，同时该书的作者在[Bishop, 2003]中更加深入地剖析了计算机安全。[Peikari & Chuvakin, 2004]也介绍了计算机安全，但关注点是攻击系统的各种方式。[Erickson, 2008]和[Anley, 2007]详尽讨论了各种安全陷阱并为聪明的程序员提供了详尽的信息来避免这些陷阱。

更多资源请访问稀酷客(www.ckook.com)

[Chen et al., 2002]这篇论文描述和分析了 UNIX set-user-ID 模型。[Tsafrir et al., 2008]对在[Chen et al., 2002]中讨论的各个课题进行了优化和增强。[Drepper, 2009]为 Linux 上的安全和防御性程序设计提供了有价值的建议。

网上存在几个编写安全程序的信息源，如下所示。

- Matt Bishop 撰写了很多与安全相关的文章，读者可以在 http://nob.cs.ucdavis.edu/~bishop/secprog 上找到这些文章。其中最有趣的一篇是《How to Write a Setuid Program》（原先发表于；login: 12(1) Jan/Feb 1986）。尽管这篇文章的年代有些久远，但其中包含了很多有价值的建议。
- David Wheeler 撰写的 *Secure Programming for Linux and Unix HOWTO* 可以在 http://www.dwheeler.com/secure-programs/上找到。
- http://www.homeport.org/~adam/setuid.7.html 为编写 set-user-ID 程序提供了一个有用的检查清单。

38.13 习题

38-1. 用一个普通的非特权用户登录系统，创建一个可执行文件（或复制一个既有文件，如/bin/sleep），然后启用该文件的 set-user-ID 权限位（chmod u+s）。尝试修改这个文件（如 cat >> file）。当使用(ls –l)时文件的权限会发生什么情况呢？为何会发生这种情况？

38-2. 编写一个与 sudo(8)程序类似的 set-user-ID-root 程序。这个程序应该像下面这样接收命令行选项和参数：

```
$ ./douser [-u user ] program-file arg1 arg2 ...
```

douser 程序使用给定的参数执行 program-file，就像是被 user 运行一样。（如果省略了–u 选项，那么 user 默认为 root。）在执行 program-file 之前，douser 应该请求 user 的密码并将密码与标准密码文件进行比较（参见程序清单 8-2），接着将进程的用户和组 ID 设置为与该用户对应的值。

更多资源请访问稀酷客(www.ckook.com)

第39章

能力

本章描述了 Linux 能力模型，它将传统的 all-or-nothing UNIX 权限模型划分成一个个能单独启用或禁用的能力。使用能力允许程序在执行一些特权操作的同时防止它们执行其他未经允许的操作。

39.1 能力基本原理

传统的 UNIX 权限模型将进程分为两类：能通过所有权限检测的有效用户 ID 为 0（超级用户）的进程和其他所有需要根据用户和组 ID 进行权限检测的进程。

这个模型的粗粒度划分是一个问题。如果需要允许一个进程执行一些只有超级用户才能执行的操作——如修改系统时间——那么就必须要让进程的有效用户 ID 为 0。（如果一个非特权用户需要执行这样的操作，那么通常需要使用 set-user-ID-root 程序来完成。）但这种做法同时也赋予了进程执行其他操作的权限——如在访问文件时会通过所有的权限检测——从而为程序表现异常（可能是由于未预见的环境或由于恶意用户的有意操作）时引起安全性问题埋下了隐患。第 38 章列出了处理此类问题的传统方法：删除有效权限（如将有效用户 ID 改为非零，同时将零保存在 saved set-user-ID 中）并只在需要的时候临时请求这些权限。

Linux 能力模型优化了对这个问题的处理方式，即在内核中执行安全性检测时不再使用单个权限（即有效用户 ID 为 0），超级用户权限被划分成了不同的的单元，这个单元成为能力。每个特权操作与一个特定的能力相关联，进程只有在拥有相应的能力的时候（不管其有效用户 ID 是什么）才能执行相应的操作。换句话说，本书中所讨论的 Linux 中的特权进程其实是指拥有相应的能力来执行特定操作的进程。

在大多数时候，Linux 能力模型对程序员来讲是不可见的，其原因是当一个对能力一无所知的应用程序的有效用户 ID 为 0 时，内核会赋予该进程所有能力。

Linux 能力是基于 POSIX 1003.1e 标准草案（http://wt.tuxomania.net/publications/posix.1e/）实现的。虽然这一项标准化工作在 20 世纪 90 年代末完成之前被放弃了，但各种能力实现仍然是根据这个标准草案来实施的。（表 39-1 列出了 POSIX.1e 草案中定义的一些能力，但大多数能力是 Linux 上的扩展。）

更多资源请访问稀酷客(www.ckook.com)

一些 UNIX 实现也提供的能力模型，如 Sun's Solaris 10 以及早期的 Trusted Solaris 发行版、SGI's Trusted Irix、以及作为 FreeBSD 一部分的 TrustedBSD 项目（([Watson, 2000])。其他一些操作系统中也存在类似的模型，如 Digital's VMS 系统中的特权机制。

39.2 Linux 能力

表 39-1 列出了 Linux 能力并为各个能力应用的操作提供了一个简短的（以及不完整的）指南。

39.3 进程和文件能力

每个进程拥有 3 个相关能力集——术语称作“许可的”、“有效的”以及“可继承的”——每个能力集都包含表 39-1 中列出的零个或多个能力。同样，每个文件也可以拥有 3 个相关能力集，其名称与进程的能力集名称一样。（其原因是非常明显的，文件的有效能力集其实就是一个可以被启用或禁用的位。）下面几节将会深入介绍各个能力集的细节信息。

39.3.1 进程能力

内核会为每个进程都维护 3 个能力集（实现为位掩码），每个能力集中都包含表 39-1 中列出的已经启用的零个或多个能力。这 3 个能力集如下所示。

- 许可的——这些是一个进程可能使用的能力。许可的集合是能够被添加到有效的和可继承的集合中的能力的受限超集。如果一个进程从其许可集中删除了一个能力，那么将永远也无法再重新获取该能力（除非它执行了一个再次授予该能力的程序）。
- 有效的——内核会使用这些能力来对进程执行权限检测。只有进程在其许可集中维护着一个能力，那么进程才能通过从有效集中删除这个能力来临时禁用该能力，之后再将该能力还原到这个集合中。
- 可继承的——当这个进程执行一个程序时可以将这些权限带入许可集中。

通过 Linux 特有的/proc/PID/status 文件中的 CapInh、CapPrm 以及 CapEff 三个字段能够查看任意进程的 3 个能力集的十六进制表示。

可以使用 getpcap 程序（第 39.7 节中介绍的 libcap 包的一部分）以更易阅读的格式显示一个进程的能力。

通过 fork()创建的子进程会继承其父进程的能力集的副本。在 39.5 节中描述了在 exec()调用中能力集的处理方式。

实际上，能力是一个线程级的特性，进程中的每个线程的能力都可以单独进程调整。在/proc/PID/task/TID/status 文件中可以查看一个多线程进程中某个具体线程的能力。/proc/PID/status 文件显示了主线程的能力。

在 2.6.25 之前的内核中，Linux 使用 32 位来表示能力集，而在 2.6.25 内核中因为加入了更多的能力导致需要 64 位来表示能力集。

更多资源请访问稀酷客(www.ckook.com)

39.3.2 文件能力

如果一个文件拥有相关的能力集，那么这些集合会被用来确定赋给执行这个文件的进程的能力。文件能力集包括下面 3 个。

- 许可的：在 exec()调用中可以将这组能力添加到进程的许可集中，不管进程的既有能力是什么。
- 有效的：这个只有一位。如果被启用了，那么在 exec()调用中，进程的新许可集中启用的能力在进程的新有效集中也会被启用。如果文件有效位被禁用了，那么在 exec()执行完之后，进程的新有效集在一开始是空的。
- 可继承的：这个集合将与进程的可继承集取掩码来确定在执行 exec()之后进程的许可集中启用的能力集。

第 39.5 节详细描述了在 exec()调用中如何使用文件能力。

许可和可继承文件能力原来称为强制的能力和允许的能力。现在那些术语已经过时了，但它们仍然能够提供一些有用的信息。许可的文件能力是那些在 exec()调用中被强制添加到进程的许可集中的能力，不管进程的既有能力是什么。可继承的文件能力是那些在 exec()调用中允许进入进程的许可集中的能力，前提是在进程的可继承能力集中也启用了那些能力。

与文件相关的能力是存储在名为 security.capability 的安全扩展特性（参见 16.1 节）中的，更新这个扩展特性需要具备 CAP_SETFCAP 能力。

表 39-1：各个 Linux 能力允许的操作

能　力	允 许 进 程
CAP_AUDIT_CONTROL	（自 Linux 2.6.11 起）启用和禁用内核审计日志、修改审计的过滤规则、读取审计状态和过滤规则
CAP_AUDIT_WRITE	（自 Linux 2.6.11 起）向内核审计日志写入记录
CAP_CHOWN	修改文件的用户 ID（所有者）或将文件的组 ID 修改为不包含进程的一个组（chown()）
CAP_DAC_OVERRIDE	绕过文件读取、写入和执行权限检查（DAC 是 discretionary access control 的缩写）；读取/proc/PID 中 cwd、exe 和 root 符号链接的内容
CAP_DAC_READ_SEARCH	绕过文件读取权限检查以及目录读取和执行的权限检查
CAP_FOWNER	忽略那些平时要求进程的文件系统用户 ID 与文件的用户 ID 匹配的操作（chmod(), utime()）的权限检查；设置任意文件的 i-node 标记；设置和修改任意文件的 ACL；在删除文件（unlink(), rmdir(), rename()）时忽略目录粘滞位的效果；在 open()和 fcntl(F_SETFL)中为任意文件指定 O_NOATIME 标记
CAP_FSETID	修改文件时使内核不关闭 set-user-ID 和 set-group-ID 位（write(), truncate()）；为那些组 ID 与进程的文件系统组 ID 或补充组 ID 不匹配的文件启用 set-group-ID 位
CAP_IPC_LOCK	覆盖内存加锁限制（mlock(), mlockall(), shmctl(SHM_LOCK), shmctl(SHM_UNLOCK)）；使用 shmget() SHM_HUGETLB 标记和 mmap() MAP_HUGETLB 标记

更多资源请访问稀酷客(www.ckook.com)

续表

能　　力	允 许 进 程
CAP_IPC_OWNER	绕过操作 System V IPC 对象的权限检查
CAP_KILL	绕过发送信号（kill(), sigqueue()）的权限检查
CAP_LEASE	（自 Linux 2.4 起）在任意文件上建立租赁关系（fcntl(F_SETLEASE)）
CAP_LINUX_IMMUTABLE	设置附加和不可变的 i-node 标记
CAP_MAC_ADMIN	（自 Linux 2.6.25 起）配置或修改强制访问控制（MAC）的状态（一些 Linux 安全模块实现了这个能力）
CAP_MAC_OVERRIDE	（自 Linux 2.6.25 起）覆盖 MAC（一些 Linux 安全模块实现了这个能力）
CAP_MKNOD	（自 Linux 2.4 起）使用 mknod()创建设备
CAP_NET_ADMIN	执行各种网络相关的操作（如设置特权 socket 选项、启用组播、配置网络接口、修改路由表）
CAP_NET_BIND_SERVICE	绑定到特权 socket 端口
CAP_NET_BROADCAST	（未使用）执行 socket 广播和监听组播
CAP_NET_RAW	使用原始和包 socket
CAP_SETGID	随意修改进程组 ID（setgid(), setegid(), setregid(),setresgid(), setfsgid(), setgroups(), initgroups()）；在通过 UNIX domain socket（SCM_CREDENTIALS）传递验证信息时伪造组 ID
CAP_SETFCAP	（自 Linux 2.6.24 起）设置文件能力
CAP_SETPCAP	在不支持文件能力时将进程的许可集中的能力授予其他进程（包括自己）或删除其他进程（包括自己）许可集中的能力；在支持文件能力时将进程的能力边界集中的所有能力都添加到自己的可继承集中，删除边界集中的能力以及修改 securebits 标记
CAP_SETUID	随意修改进程用户 ID（setuid(), seteuid(), setreuid(),setresuid(), setfsuid()）；在通过 UNIX domain socket（SCM_CREDENTIALS）传递验证信息时伪造用户 ID
CAP_SYS_ADMIN	在打开文件的系统调用中（如 open(),shm_open(), pipe(), socket(), accept(), exec(), acct(), epoll_create()）超出/proc/sys/fs/file-max 限制；执行各种系统管理操作，包括 quotactl()（控制磁盘限额）、mount()和 umount(), swapon() 和 swapoff(), pivot_root(), sethostname() 和 setdomainname()；执行各种 syslog(2)操作；覆盖 RLIMIT_NPROC 资源限制（fork()）；调用 lookup_dcookie()；设置 trusted 和 security 扩展特性；在任意 System V IPC 对象上执行 IPC_SET 和 IPC_RMID 操作；在通过 UNIX domain socket（SCM_CREDENTIALS）传递验证信息时伪造进程 ID；使用 ioprio_set()来分配 IOPRIO_CLASS_RT 调度类；使用 TIOCCONS ioctl()；在 clone()和 unshare()中使用 CLONE_NEWNS 标记；执行 KEYCTL_CHOWN 和 KEYCTL_SETPERM keyctl()操作；管理 random(4)设备；各种特定于设备的操作
CAP_SYS_BOOT	使用 reboot()重启系统；调用 kexec_load()
CAP_SYS_CHROOT	使用 chroot()设置进程根目录
CAP_SYS_MODULE	加载和卸载内核模块（init_module(), delete_module(), create_module()）

更多资源请访问棒酷客(www.ckook.com)

续表

能　力	允 许 进 程
CAP_SYS_NICE	提高 nice 值（nice(), setpriority()）；修改任意进程的 nice 值（setpriority()）；设置调用进程的 SCHED_RR 和 SCHED_FIFO 实时调度策略；重置 SCHED_RESET_ON_FORK 标记；设置任意进程的调度策略和优先级（sched_setscheduler(), sched_setparam()）；设置任意进程的 I/O 调度类和优先级（ioprio_set()）；设置任意进程的 CPU 亲和力（sched_setaffinity()）；使用 migrate_pages()将任意进程迁移到任意节点以及允许进程被迁移到任意节点；对任意进程应用 move_pages()；在 mbind() 和 move_pages()中使用 MPOL_MF_MOVE_ALL 标记
CAP_SYS_PACCT	使用 acct()启用或禁用进程记账
CAP_SYS_PTRACE	使用 ptrace()跟踪任意进程；访问任意进程的/proc/PID/environ；对任意进程应用 get_robust_list()
CAP_SYS_RAWIO	使用 iopl() 和 ioperm()在 I/O 端口上执行操作；访问/proc/kcore；打开/dev/mem 和/dev/kmem
CAP_SYS_RESOURCE	使用文件系统上的预留空间；使用 ioctl()调用控制 ext3 journaling；覆盖磁盘限额限制；提高硬资源限制（setrlimit()）；覆盖 RLIMIT_NPROC 资源限制（fork()）；将 System V 消息队列的/proc/sys/kernel/msgmnb 限制提高 msg_qbytes；绕过由/proc/sys/fs/mqueue 下各个文件定义的各种 POSIX 消息队列限制
CAP_SYS_TIME	修改系统时钟（settimeofday(), stime(), adjtime(), adjtimex()）；设置硬件时钟
CAP_SYS_TTY_CONFIG	使用 vhangup()执行终端或伪终端的虚拟挂起

39.3.3 进程许可和有效能力集的目的

进程的许可集定义了进程能够使用的能力。进程的有效能力集定义了进程当前使用的能力——即内核会使用这组能力来检查进程是否拥有足够的权限执行某个特定的操作。

许可能力集为有效能力集定义了一个上限。进程只能将其许可能力集中的能力上升到有效集中。（上升有时候也被称为添加或设置，与之相反的操作是丢弃或删除或清除。）

有效能力集和许可能力集之间的关系与一个 set-user-ID-root 程序中的有效用户 ID 和 set-user-ID 之间的关系类似。从有效集中删除一个能力与临时删除一个有效用户 ID 0 同时在 saved set-user-ID 中维持 0 是类似的。从有效能力集和许可能力集中删除一个能力与通过将有效用户 ID 和 saved set-user-ID 设置为非零值来永久删除超级用户权限类似。

39.3.4 文件许可和有效能力集的目的

文件许可能力集为可执行文件向进程赋予能力提供了一种机制，它指定了在 exec()调用中被赋给进程的许可能力集的一组能力。

文件有效文件集是一个可以被启用或禁用的标记（位）。要理解为何这个集合只由一个位构成就需要考虑在程序被执行时会发生的两种情况。

- 程序可能是一个能力哑元，表示它对能力一无所知（即传统的 set-user-ID-root 程序）。这种程序不知道需要在其有效集中提升能力以便能够执行特权操作。对于这样的程序

更多资源请访问稀酷客(www.ckook.com)

来讲，exec()应该将进程的新许可集中的所有能力自动加到其有效集中，这是通过启用文件有效位来完成的。

- 程序可能是知道能力的，表示在设计程序的时候使用了能力框架，并且会使用合适的系统调用（稍后讨论）来在其有效集中提升和删除能力。对于这样的程序来讲，最小权限表示在 exec()调用之后，进程的有效能力集中的所有能力一开始都是被禁用的，这是通过禁用文件有效能力位来完成的。

39.3.5 进程和文件可继承集的目的

乍一看，对于能力系统来讲，有了进程和文件的许可集和有效集看起来已经足够了，但还是存在一些这两个集合无法满足要求的情形。如当一个执行 exec()的进程想要在 exec()调用期间保存其当前能力时该如何做呢？看起来，能力实现可以通过简单地在 exec()调用之间保存进程的许可能力来实现这个特性，但这种方式无法处理下列情形。

- 执行 exec()可能需要特定的权限（如 CAP_DAC_OVERRIDE），但在 exec()调用之间可能不想要保存这种权限。
- 假设显式地删除了一些无需在 exec()调用之间保存的许可能力，但 exec()调用失败了。在这种情况下，程序可能需要知道这些已经被删除（不可逆的）的许可能力。

基于上述原因，在 exec()之间是不会保持进程的许可能力的。相反，在这种情况下会适应另一种能力集：可继承集。可继承集为进程在 exec()调用之间保持其部分能力提供了一种机制。

进程的可继承集指定了一组在 exec()调用之间可被赋给进程的许可能力集的能力。相应文件的可继承集会根据进程的可继承集取掩码（AND）来确定在 exec()之间被添加到进程的许可能力集中的能力。

> 在 exec()之间不是简单保持进程的许可能力集还存在深层次的哲学原因。能力系统的主要思想是赋给进程的所有权限都是由进程所执行的文件来授予或控制的。虽然进程的可继承集指定了在 exec()之间传入能力，但这些能力会根据文件的可继承集来取掩码。

39.3.6 在 shell 中给文件赋予能力和查看文件能力

在 39.7 节中介绍的 libcap 包中包含的 setcap(8)和 getcap(8)命令可以用来操作文件能力集。下面通过一个使用了标准的 date(1)程序的简短示例来演示这些命令的用法。（根据 39.3.4 节中给出的定义，这个程序就是一种能力哑元应用程序。）当在具备权限的情况下运行这个程序时，date(1)可以用来修改系统时间。date 程序不是一个 set-user-ID-root，因此通常使用权限运行这个程序的唯一方式是变成超级用户。

下面首先显示当前的系统时间，然后尝试以一个非特权用户的身份来修改时间。

```
$ date
Tue Dec 28 15:54:08 CET 2010
$ date -s '2018-02-01 21:39'
date: cannot set date: Operation not permitted
Thu Feb  1 21:39:00 CET 2018
```

从上面可以看出 date 命令没有能够修改系统时间，但它仍然以标准格式显示了传入的参数。

更多资源请访问稀酷客(www.ckook.com)

接下来变成超级用户，这样就能够成功地修改系统时间了。

```
$ sudo date -s '2018-02-01 21:39'
root's password:
Thu Feb  1 21:39:00 CET 2018
$ date
Thu Feb  1 21:39:02 CET 2018
```

现在复制一份 date 程序的副本并赋予该副本所需的能力。

```
$ whereis -b date                                  Find location of date binary
date: /bin/date
$ cp /bin/date .
$ sudo setcap "cap_sys_time=pe" date
root's password:
$ getcap date
date = cap_sys_time+ep
```

上面的 setcap 命令将 CAP_SYS_TIME 能力赋给了可执行文件的许可能力集（p）和有效能力集（e）。接着使用了 getcap 命令来验证能力确实被赋给了文件。（libcap 包中的 cap_from_text(3)手册描述了 setcap 和 getcap 中用来表示能力集的语法。）

date 程序的副本的文件能力允许非特权用户使用这个程序来设置系统时间。

```
$ ./date -s '2010-12-28 15:55'
Tue Dec 28 15:55:00 CET 2010
$ date
Tue Dec 28 15:55:02 CET 2010
```

39.4 现代能力实现

能力的完整实现要求如下。

- 对于每个特权操作，内核应该检查进程是否拥有相应的能力，而不是检查有效（或文件系统）用户 ID 是否为 0。
- 内核必须要提供允许获取和修改进程能力的系统调用。
- 内核必须要支持将能力附加给可执行文件的概念，这样当文件被执行时进程会获取相应的能力。这与 set-user-ID 位是类似的，但允许单独地设置可执行文件上的各个能力。此外，系统必须要提供一组编程接口和命令来设置和查看附加给可执行完文件的能力。

在 2.6.23 以及之前的内核中，Linux 只满足了前两条要求。自 2.6.24 内核开始就可以将能力附加到文件上了。在 2.6.25 和 2.6.26 内核中新增了很多其他特性以完善能力的实现。

这里针对有关能力的大多数讨论关注的都是现代实现。在 39.10 节中将介绍在引入文件能力之前实现之间存在的不一致性。此外，文件能力是现代内核的一个可选内核组件，但本次讨论的主要部分假设在内核中启用了这个组件。接着将会介绍文件能力被启用与被禁用之间存在的差别。（从几个方面来看，其行为与还未实现文件能力的 2.6.24 之前的 Linux 内核中行为类似。）

在下面几个小节中将深入介绍 Linux 能力实现的细节。

39.5 在 exec()中转变进程能力

在 exec()执行期间，内核会根据进程的当前能力以及被执行的文件的能力集来设置进程的

更多资源请访问稀酷客(www.ckook.com)

新能力。内核会使用下面的规则来计算进程的新能力。

```
P'(permitted) = (P(inheritable) & F(inheritable)) | (F(permitted) & cap_bset)

P'(effective) = F(effective) ? P'(permitted) : 0

P'(inheritable) = P(inheritable)
```

在上面的规则中，P 表示在调用 exec()之前进程的能力集的取值，P'表示在调用 exec()之后进程的能力集的取值，F 表示文件能力集。标识符 cap_bset 表示能力边界集的取值。注意 exec()调用不会改变进程的可继承能力集。

39.5.1 能力边界集

能力边界集是一种用于限制进程在 exec()调用中能够获取的能力的安全机制，其用法如下。

- 在 exec()调用中，能力边界集会与文件许可能力取 AND 来确定将被授予新程序的许可能力。换句话说，当一个可执行文件的某个许可能力不在边界能力集中时就无法向进程授予该项能力。
- 能力边界集是一个可以被添加到进程的可继承集中的能力的受限超集。这表示除非能力位于边界集中，否则进程就无法将其许可能力集中的某个能力添加到其可继承集中并——通过上面介绍的第一条能力转换规则——在进程执行一个可继承集中包含该项能力的文件时将该项能力保留在进程的许可集中。

能力边界集是一个进程级特性，通过 fork()创建的子进程会继承这个特性，并且在 exec()调用中会保持这个特性。在支持文件能力的内核中，init（所有进程的祖先）在启动时会使用一个包含了所有能力的能力边界集。

如果一个进程具备了 CAP_SETPCAP 能力，那么它就可以使用 prctl() PR_CAPBSET_DROP 操作从其边界集中删除能力（不可逆的）。（从边界集中删除一个能力不会对进程的许可、有效和可继承能力集产生影响。）一个进程使用 prctl() PR_CAPBSET_READ 操作能够确定一个能力是否位于其边界集中。

> 更准确地讲，能力边界集是一个线程级的特性。从 Linux 2.6.26 开始，这个特性在 Linux 特有的/proc/PID/task/TID/status 文件中的 CapBnd 字段予以显示。/proc/PID/status 文件显示了进程主线程的边界集。

39.5.2 保持 root 语义

在执行一个文件时为了保持 root 用户的传统语义（即 root 拥有所有的权限），与该文件相关联的所有能力集都会被忽略。但为了满足在 39.5 节中给出的算法的要求，在 exec()期间文件能力集的定义如下。

- 如果执行了一个 set-user-ID-root 程序或调用 exec()的进程的真实或有效用户 ID 为 0，那么文件的可继承和许可集被定义为包含所有能力。
- 如果执行了一个 set-user-ID-root 程序或调用 exec()的进程的有效用户 ID 为 0，那么文件有效位被定义成设置状态。

假设现在正在执行一个 set-user-ID-root 程序，那么这些文件能力集的概念定义表示在 39.5

更多资源请访问稀酷客(www.ckook.com)

节中给出的进程的新许可和有效能力集的计算被简化成了：

```
P'(permitted) = P(inheritable) | cap_bset
P'(effective) = P'(permitted)
```

39.6 改变用户 ID 对进程能力的影响

为了与用户 ID 在 0 与非 0 之间切换的传统含义保持兼容，在改变进程的用户 ID（使用 setuid()等）时，内核会完成下列操作。

1. 如果真实用户 ID、有效用户 ID 或 saved set-user-ID 之前的值为 0，那么修改了用户 ID 之后，所有这三个 ID 的值都会变成非 0，并且进程的许可和有效能力集会被清除（即所有的能力都被永久地删除了）。
2. 如果有效用户 ID 从 0 变成了非 0，那么有效能力集会被清除（即有效能力被删除了，但那些位于许可集中的能力会被再次提升）。
3. 如果有效用户 ID 从非 0 变成了 0，那么许可能力集会被复制到有效能力集中（即所有的许可能力变成了有效）。
4. 如果文件系统用户 ID 从 0 变成了非 0，那么会从有效能力集中清除这些文件相关的能力：CAP_CHOWN、CAP_DAC_OVERRIDE、CAP_DAC_READ_SEARCH、CAP_FOWNER、CAP_FSETID、CAP_LINUX_IMMUTABLE（自 Linux 2.6.30 起）、CAP_MAC_OVERRIDE 和 CAP_MKNOD（自 Linux 2.6.30 起）。相应地，如果文件系统用户 ID 从非 0 变成了 0，那么上面这些能力中所有在许可集中被启用的能力会在有效集中被启用。完成这些操作之后，对 Linux 特有的文件系统用户 ID 的操作的传统语义将会得到保持。

39.7 用编程的方式改变进程能力

一个进程可以使用 capset()系统调用或稍后介绍的 libcap API（首选方法）在其能力集中提升能力或删除能力。修改进程能力需要遵循下列规则。

1. 如果进程的有效集中没有 CAP_SETPCAP 能力，那么新的可继承集必须是既有可继承集合许可集组合的一个子集。
2. 新的可继承集必须是既有可继承集合能力边界集组合的一个子集。
3. 新许可集必须是既有许可集的一个子集。换句话说，一个进程无法授予自身不属于其许可集中的能力。换一种表述方法就是，在从许可集中删除了一个能力之后就无法再获取这个能力了。
4. 新的有效集只能包含位于新许可集中的能力。

libcap API

本章到现在还不介绍 capset()系统调用以及相应的获取进程能力的 capget()系统调用的原型，是因为应该避免使用这两个系统调用。相反，应该使用 libcap 库中的相关函数。这些函数提供了一个与 POSIX 1003.1e 标准草案一致的接口以及一些 Linux 扩展。

限于篇幅，本章不会详细介绍 libcap API。总的来说，使用这些函数的程序通常会执行下列步骤。

1. 使用 cap_get_proc()函数从内核中获取进程的当前能力集的一个副本并将其放置到这

更多资源请访问稀酷客(www.ckook.com)

个函数在用户空间分配的一个结构中。（或者可以使用 cap_init()函数来创建一个全新的空能力集结构。）在 libcap API 中，cap_t 数据类型是用来引用此类结构的一个指针。

2. 使用 cap_set_flag()函数更新用户空间的结构以便在上一步骤中返回的存储于用户空间中的结构中的许可、有效和可继承集中提升（CAP_SET）和删除（CAP_CLEAR）能力。
3. 使用 cap_set_proc()函数将用户空间的结构传回内核以修改进程的能力。
4. 使用 cap_free()函数释放在第一步中由 libcap API 分配的结构。

libcap-ng 是一个全新改良过的能力库 API。在撰写本书的时候，有关 libcap-ng 的开发工作仍在进行中，详细信息可参考 http://freshmeat.net/projects/libcap-ng。

示例程序

程序清单 8-2 会根据标准的口令数据库来验证用户名和口令。注意程序在读取影像口令文件时需要具备相应的权限，而只有 root 和 shadow 组中的成员才能够读取这个文件。给这个程序赋予它所需的权限的传统方式是在 root 用户下运行这个程序或将程序变成一个 set-user-ID-root 程序。下面将修改这个程序使之使用能力和 libcap API。

为了能够以普通用户的身份读取影像口令文件就需要绕过标准的文件权限检查。从表 39-1 中列出的能力可以看出相应的能力应该是 CAP_DAC_READ_SEARCH。程序清单 39-1 给出了修改过的口令校验程序。这个程序在正好需要访问影像口令文件之前使用了 libcap API 来在其有效能力集中提升 CAP_DAC_READ_SEARCH，然后在访问文件之后立即删除了这个能力。为了让非特权用户能够使用这个程序，必须要在文件的许可能力集中设置这个能力，如下面的 shell 会话所示。

```
$ sudo setcap "cap_dac_read_search=p" check_password_caps
root's password:
$ getcap check_password_caps
check_password_caps = cap_dac_read_search+p
$ ./check_password_caps
Username: mtk
Password:
Successfully authenticated: UID=1000
```

程序清单 39-1：使用能力来验证用户的程序

cap/check_password_caps.c

```
#define _BSD_SOURCE         /* Get getpass() declaration from <unistd.h> */
#define _XOPEN_SOURCE       /* Get crypt() declaration from <unistd.h> */
#include <sys/capability.h>
#include <unistd.h>
#include <limits.h>
#include <pwd.h>
#include <shadow.h>
#include "tlpi_hdr.h"

/* Change setting of capability in caller's effective capabilities */

static int
modifyCap(int capability, int setting)
```


更多资源请访问稀酷客(www.ckook.com)

```
{
    cap_t caps;
    cap_value_t capList[1];

    /* Retrieve caller's current capabilities */

    caps = cap_get_proc();
    if (caps == NULL)
        return -1;

    /* Change setting of 'capability' in the effective set of 'caps'. The
       third argument, 1, is the number of items in the array 'capList'. */

    capList[0] = capability;
    if (cap_set_flag(caps, CAP_EFFECTIVE, 1, capList, setting) == -1) {
        cap_free(caps);
        return -1;
    }

    /* Push modified capability sets back to kernel, to change
       caller's capabilities */

    if (cap_set_proc(caps) == -1) {
        cap_free(caps);
        return -1;
    }

    /* Free the structure that was allocated by libcap */

    if (cap_free(caps) == -1)
        return -1;

    return 0;
}

static int              /* Raise capability in caller's effective set */
raiseCap(int capability)
{
    return modifyCap(capability, CAP_SET);
}

/* An analogous dropCap() (unneeded in this program), could be
   defined as: modifyCap(capability, CAP_CLEAR); */

static int              /* Drop all capabilities from all sets */
dropAllCaps(void)
{
    cap_t empty;
    int s;

    empty = cap_init();
    if (empty == NULL)
        return -1;

    s = cap_set_proc(empty);
    if (cap_free(empty) == -1)
        return -1;

    return s;
```

```
}

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");

    /* Only raise CAP_DAC_READ_SEARCH for as long as we need it */

    if (raiseCap(CAP_DAC_READ_SEARCH) == -1)
        fatal("raiseCap() failed");

    spwd = getspnam(username);
    if (spwd == NULL && errno == EACCES)
        fatal("no permission to read shadow password file");

    /* At this point, we won't need any more capabilities,
       so drop all capabilities from all sets */

    if (dropAllCaps() == -1)
        fatal("dropAllCaps() failed");

    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);
}
```

cap/check_password_caps.c

39.8 创建仅包含能力的环境

在前面几页中介绍了在能力方面对用户 ID 为 0（root）的进程进行特殊处理的各种方式。

- 当一个或多个用户 ID 等于 0 的进程将其所有的用户 ID 设置为非 0 值时，进程的许可和有效能力集会被清除。（参见 39.6 节）。
- 当有效用户 ID 为 0 的进程将用户 ID 修改为非 0 值时会失去其有效能力。当做方向相反的变动时，许可能力集会被复制到有效集中。当进程的文件系统 ID 在 0 和非 0 值之间切换时会对能力子集执行一个类似的步骤。
- 当真实或有效用户 ID 为 root 的进程执行了一个程序或任意进程执行了一个 set-user-ID-root 程序，那么文件的可继承和许可集会被定义成包含所有能力。如果进程的有效用户 ID 为 0 或者它正在执行一个 set-user-ID-root 程序，那么文件的有效位被定义成 1。（参见 39.5.2 节）在通常情况下（即真实和有效用户 ID 都是 root 或正在执行一个 set-user-ID-root 程序），这表示进程的许可和有效集中包含了所有能力。

在一个完全基于能力的系统中，内核无需对 root 用户执行这些特殊的处理，因为不存在 set-user-ID-root 程序并且只会使用文件能力赋给程序执行所需的最小能力。

由于既有应用程序不会使用文件能力基础架构，因此内核必须要维持对用户 ID 为 0 的进程的传统处理，但可以要求应用程序在一个完全基于能力的环境中运行，在这样的环境中，不会对 root 做上述的特殊处理。从 2.6.26 的内核开始，当在内核中启用了文件能力时，Linux 会提供 securebits 机制，它可以控制一组进程级别的标记，通过这组标记可以分别启用或禁用前面针对 root 的三种特殊处理中的各种特殊处理。（更准确地讲，securebits 标记实际上是一个线程级别的特性。）

securebits 机制控制着表 39-2 中列出的标记，每个标记由一对相关的 base 标记和相应的 locked 标记表示。每个 base 标记控制上面描述的针对 root 的一种特殊处理。设置相应的 locked 标记是一个一次性操作，用于防止对相关联的 base 标记的后续变更——一旦设置之后就无法重置 locked 标记了。

更多资源请访问稀酷客(www.ckook.com)

表 39-2：securebits 标记

标　　记	设置之后的含义
SECBIT_KEEP_CAPS	当一个或多个用户 ID 为 0 的进程将其所有的用户 ID 设置为非 0 值时不要删除许可权限。只有在没有设置 SECBIT_NO_SETUID_FIXUP 标记的情况下这个标记才会起作用。在 exec()中这个标记会被清除
SECBIT_NO_SETUID_FIXUP	当有效或文件系统用户 ID 在 0 和非 0 之间切换时不要改变能力
SECBIT_NOROOT	在一个真实或有效用户 ID 为 0 的进程调用了 exec()或执行了一个 set-user-ID-root 程序时不要赋予其能力（除非可执行文件拥有文件能力）
SECBIT_KEEP_CAPS_LOCKED	锁住 SECBIT_KEEP_CAPS
SECBIT_NO_SETUID_FIXUP_LOCKED	锁住 SECBIT_NO_SETUID_FIXUP
SECBIT_NOROOT_LOCKED	锁住 SECBIT_NOROOT

fork()创建子进程会继承 securebits 标记设置。在调用 exec()期间，除 SECBIT_KEEP_ CAPS 之外的所有标记设置都会得到保留，之所以清除 SECBIT_KEEP_CAPS 标记是为了与下面描述的 PR_SET_KEEPCAPS 设置保持兼容。

进程可以使用 prctl() PR_GET_SECUREBITS 操作来获取 securebits 标记。一个进程如果拥有 CAP_SETPCAP 能力，那么它就可以使用 prctl() PR_SET_SECUREBITS 操作修改 securebits 标记。一个完全基于能力的应用程序能够使用下面的调用不可逆地禁用调用进程及其所有子孙进程对 root 用户的特殊处理。

```
if (prctl(PR_SET_SECUREBITS,
          /* SECBIT_KEEP_CAPS off */
          SECBIT_NO_SETUID_FIXUP | SECBIT_NO_SETUID_FIXUP_LOCKED |
          SECBIT_NOROOT | SECBIT_NOROOT_LOCKED)
        == -1)
    errExit("prctl");
```

在执行完这个调用之后，这个进程及其所有子孙进程获取能力的唯一方式是执行拥有文件能力的程序。

SECBIT_KEEP_CAPS 和 prctl() PR_SET_KEEPCAPS 操作

SECBIT_KEEP_CAPS 标记能够防止能力在一个或多个用户 ID 为 0 的进程将其所有的用户 ID 值设置为非 0 值时被删除。粗略地讲，SECBIT_KEEP_CAPS 提供了 SECBIT_NO_SETUID_FIXUP 标记的一半功能。（从表 39-2 中可以看出，只有在 SECBIT_NO_SETUID_FIXUP 没有被设置的情况下，SECBIT_KEEP_CAPS 才会起作用。）这个标记的存在是为了提供一个实现更古老的 prctl() PR_SET_KEEPCAPS 操作的 securebits 标记，它控制着同样的特性。（这两种机制之间的一个差别是进程在使用 prctl() PR_SET_KEEPCAPS 操作时无需具备 CAP_SETPCAP 能力。）

> 之前曾经提过在 exec()调用期间会保持除 SECBIT_KEEP_CAPS 之外的所有 securebits 标记。对 SECBIT_KEEP_CAPS 位的设置与其他 securebits 设置相反是为了与通过 prctl() PR_SET_KEEPCAPS 操作设置的对特性的处理保持一致。

更多资源请访问稀酷客(www.ckook.com)

prctl() PR_SET_KEEPCAPS 操作由运行于老式的不支持文件能力的内核上的 set-user-ID-root 程序使用。此类程序可以通过在程序中删除能力并在需要的时候提升能力（参见 39.10 节）来提高安全性。

即使此类 set-user-ID-root 程序删除了除所需的权限之外的所有其他权限，它仍然会保留两个重要的权限：访问由 root 用户拥有的文件的权限以及通过执行程序重新获取能力的权限（参见 39.5.2 节）。永久删除这些权限的唯一方式是将进程的所有用户 ID 值设置为非 0 值，但这样做通常会导致清除许可和有效能力集（参见 39.6 节中有关用户 ID 的变动对能力造成的四点影响）。这就产生了矛盾，即在保持一些能力的同时永久地删除用户 ID 0。为了允许这样的情况发生，可以使用 prctl() PR_SET_KEEPCAPS 操作来设置进程特性以防止在所有的用户 ID 变成非 0 值时许可能力集被清除。（在这种情况下总是会清除进程的有效能力集，不管是否设置了“keep capabilities”特性。）

39.9 发现程序所需的能力

假设现在有一个对能力一无所知的程序并且只有这个程序的二进制文件，或者假设程序的代码太多了以至于无法很容易地确定运行这个程序需要具备那些能力。如果这个程序需要特权，但又不是一个 set-user-ID-root 程序，那么如何确定将哪些许可能力使用 setcap(8)赋给这个可执行文件呢？解答这个问题的答案有两个。

- 使用 strace(1)（附录 A）检查哪个系统调用的错误号是 EPERM，因为这个错误号是用来标示缺乏所需的能力的。通过查阅系统调用的手册或内核的源代码可以推断出程序需要哪些能力。但这个方法不是很完美，因为偶尔会因为其他原因而引起 EPERM 错误，其中一些原因与程序缺乏相应的能力这个问题毫无关系。此外，程序可能会正常调用一个需要权限的系统调用，然后在确定没有权限执行某个特定操作之后改变自身的行为。而有些时候在试图确定一个可执行文件实际所需的能力时是难以区分这种“积极响应错误”的情况的。
- 使用一个内核探针在内核被要求执行能力检查时产生监控输出。[Hallyn, 2007]（由其中一个文件能力模块的开发者撰写的一篇文章）提供了如何完成这个任务的一个示例。对于每个能力检查请求，文章中所指的探针都会记录被调用的内核函数、被请求的能力以及请求程序的名称。虽然这个方法比使用 strace(1)需要做更多的工作，但它有助于更加精确地确定一个程序所需的能力。

39.10 不具备文件能力的老式内核和系统

本节将介绍之前各种版本的内核中有关能力实现方面的差异以及碰到不支持文件能力的内核时所发生的行为差异。Linux 在下面两个场景中是不支持文件能力的。

- 在 Linux 2.6.24 之前的版本中没有实现文件能力。
- 自 Linux 2.6.24 起，当在构建内核时不指定 CONFIG_SECURITY_FILE_CAPABILITIES 选项时文件能力将会被禁用。

虽然从 2.2 内核开始，Linux 就已经引入了能力并允许将能力附加到进程中，但文件能力的实现则推后了好几年。之所以未实现文件能力的原因不是因为技术上的困难，而是因

更多资源请访问稀酷客(www.ckook.com)

为政策的原因。（第 16 章介绍的扩展特性被用来实现文件能力，但它直到 2.6 内核才可用。）大部分内核开发人员要求系统管理员为各个特权程序分别设置和监控能力集的意见会使得管理任务变得复杂和难以管理——虽然有些意见是合理的，但很难做到。相反，系统管理员对于现有的 UNIX 权限模型比较熟悉，他们知道如何小心处理 set-user-ID 程序并且能够使用简单的 find 命令找出系统中的 set-user-ID 和 set-group-ID 程序。不过，文件能力模块的开发人员使得文件能力的应用在管理上变得可行，最终为将文件能力集成进内核提供了足够的令人信服的论据。

CAP_SETPCAP 能力

在不支持文件能力的内核中（即所有 2.6.24 之前的内核以及自 2.6.24 起文件能力被禁用的内核），CAP_SETPCAP 能力的语义是不同的。根据与 39.7 节中描述的规则类似的规则，从理论上来讲，一个在有效集中包含 CAP_SETPCAP 能力的进程能够修改除自身之外的其他进程的能力。换句话说，可以修改另一个进程的能力、指定进程组中所有成员的能力以及系统中除 init 和调用者本身之外的所有进程的能力。之所以将 init 排除在外是因为它对于系统的运作起着基础性的作用。之所以还将调用者本身排除在外是因为调用者可能会试图删除系统中其他进程的能力，但这里并不希望调用进程能删除自己的能力。

修改其他进程的能力只是在理论上可行，在较早的内核以及禁用了文件能力的现代内核中，能力边界集（稍后讨论）总是会隐藏掉 CAP_SETPCAP 能力。

能力边界集

自 Linux 2.6.25 起，能力边界集就是一个进程级的特性了。但在较早的内核中，能力边界集是一个系统级别的特性，它会影响系统中的所有进程。在初始化系统级别的能力边界集时总是会隐藏 CAP_SETPCAP（参见前面的介绍）。

在 2.6.25 之后的内核中，只有当在内核中启用了文件能力时才支持从各个进程的边界集中删除能力。在那种情况下，所有进程的祖先进程 init 在启动的时候会包含所有的能力，系统中所有其他进程会继承该边界集的一个副本。如果文件能力被禁用了，那么由于上面描述的 CAP_SETPCAP 能力的语义存在差别，因此 init 在启动的时候会包含除 CAP_SETPCAP 之外的所有能力。

在 Linux 2.6.25 中对能力边界集的语义还做了另一个变更。在之前（39.5.1 节）曾经提过，在 Linux 2.6.25 以及之后的版本中，各个进程的能力边界集是作为能够被添加到进程的可继承集中的能力的一个受限超集来处理的。在 Linux 2.6.24 以及之前的版本中，系统级别的能力边界集并没有这种掩码效果（不需要这种效果，因为这些内核不支持文件能力）。

通过 Linux 特有的/proc/sys/kernel/cap-bound 文件能够访问系统级别的能力边界集。进程必须要具备 CAP_SYS_MODULE 能力才能修改 cap-bound 文件的内容。但只有 init 进程才能够开启这个掩码中的位，其他特权进程只能关闭掩码中的位。这些限制的结果就是在不支持文件能力的系统上永远都无法将 CAP_SETPCAP 能力赋给进程。这种做法是合理的，因为这个能力可以用来破坏整个内核权限检查系统。（当需要修改这个限制时必须要加载一个修改集合中的值的内核模块并修改 init 程序的源代码或者在内核源代码中修改能力边界集的初始化过程并重建内核。）

更多资源请访问稀酷客(www.ckook.com)

令人迷惑的是，虽然是一个位掩码，但在 cap-bound 文件中系统级别的掩码值显示为一个带符号的十进制数字。如文件中的初始值是-257，它是除(1 << 8)之外所有位都被开启的位掩码的二的补码表示（即二进制格式为 11111111 11111111 11111110 11111111）；CAP_SETPCAP 的值为 8。

在运行于无文件能力的系统上的程序中使用能力

即使在不支持文件能力的系统上仍然可以使用能力来提升程序的安全性。这是通过下列步骤来完成的。

1. 在一个有效用户 ID 为 0 的进程中运行这个程序（通常是一个 set-user-ID-root 程序）。此类进程的许可和有效能力集中包含了所有的能力(前面提过,除了 CAP_SETPCAP 能力)。
2. 在程序启动的时候使用 libcap API 删除有效集中的所有能力和许可集中除后面会用到的能力之外的其他所有能力。
3. 设置 SECBIT_KEEP_CAPS 标记（或使用 prctl() PR_SET_KEEPCAPS 操作达到同样的效果），这样在下一步中就不会删除能力了。
4. 将所有用户 ID 设为非 0 值以防止进程访问由 root 拥有的文件或在 exec()中获取能力。

如果需要防止进程在 exec()中重新获取权限但同时要允许它访问由 root 拥有的文件的话则可以使用 SECBIT_NOROOT 标记这一步来取代前面的两步。（当然，允许进程访问由 root 拥有的文件为一些安全性风险打开了大门。）

5. 在程序的后续生命周期中根据需要使用 libcap API 在有效集中提升或删除剩余的许可能力以便执行特权任务。

一些基于 2.6.24 之前的 Linux 内核的应用程序采用了这种方法。

在所有反对为可执行文件实现能力的内核开发者所提出的反对理由中，反对使用正文中所描述的方法的最充分的理由之一是应用程序的开发人员通常知道可执行程序需要用到哪些能力，而系统管理员可能无法轻易地确定此类信息。

39.11 总结

Linux 能力模型将特权操作划分成不同的种类并允许一个进程在被授予一些能力的同时被禁止使用其他能力。这个模型对传统的一个进程要么拥有权限执行所有的操作（用户 ID 为 0）或没有权限（用户 ID 非 0）执行操作的 all-or-nothing 权限机制进行了优化。自 2.6.24 内核起，Linux 支持将能力附加到文件上，这样进程可以通过执行程序来获取所选中的能力。

39.12 习题

39-1. 修改程序清单 35-2 中的程序（sched_set.c）使它使用文件能力，这样非特权用户就也能使用这个程序了。

更多资源请访问稀酷客(www.ckook.com)

第40章

登录记账

登录记账关注的是哪些用户当前登录进了系统以及记录过去的登录和登出行为。本章将介绍登录记账文件以及用来获取和更新这些文件中所包含的信息的库函数。此外，本章还将介绍提供登录服务的应用程序应该采取的措施以便在用户登录和登出时更新这些文件。

40.1 utmp 和 wtmp 文件概述

UNIX 系统维护着两个包含与用户登录和登出系统有关的信息的数据文件。

- utmp 文件维护着当前登录进系统的用户记录（以及其他一些信息，稍后将会介绍）。每一个用户登录进系统时都会向 utmp 文件写入一条记录。在这条记录中包含一个 ut_user 字段，它记录着用户的登录名。当用户登出的时候该条记录会被删除。像 who(1)之类的程序会使用 utmp 文件中的信息来显示当前登录进系统的用户列表。
- wtmp 文件包含着所有用户登录和登出行为的留痕信息以供审计之用（以及其他一些信息，稍后将会介绍）。每一个用户登录进系统时，写入 utmp 文件中的记录同时会被附加到 wtmp 文件中。当用户登出系统的时候还会向这个文件附加一条记录。这条记录包含的信息与登录记录相同，但 ut_user 字段会被置零。last(1)命令可以用来显示和过滤 wtmp 文件中的内容。

在 Linux 上，utmp 文件位于/var/run/utmp，wtmp 文件位于/var/log/wtmp。一般来讲，应用程序无需知道这些路径名，因为这些路径名是编译进 glibc 的。需要引用这些文件的存储位置的程序应该使用在<paths.h> (and <utmpx.h>)中定义的_PATH_UTMP 和_PATH_WTM 路径名常量，而不是在代码中硬编码路径名。

SUSv3 并没有为 utmp 和 wtmp 文件的路径名提供标准化的符号名。Linux 和 BSD 使用了_PATH_UTMP 和_PATH_WTMP。而其他很多 UNIX 实现定义了 UTMP_FILE 和 WTMP_FILE 常量来表示这两个路径名。Linux 还在<utmp.h>中定义了这些名词，但并没有在<utmpx.h>和<paths.h>中对它们进行定义。

更多资源请访问稀酷客(www.ckook.com)

40.2 utmpx API

utmp 和 wtmp 文件很早就出现在了 UNIX 系统上了，但随着系统的演化，不同 UNIX 实现之间的分歧开始出现了，特别是 BSD 与 System V 之间的差别。System V Release 4 对 API 进行了大量的扩展，包括创建了一个全新的（并行的）utmpx 结构以及相关的 utmpx 和 wtmpx 文件。同样，处理这些新文件的函数名以及相关的头文件名中也包含了字母 x。很多其他 UNIX 实现也在 API 中增加了自己的扩展。

本章将介绍 Linux utmpx API，它是 BSD 和 System V 实现的一个混合体。Linux 并没有像 System V 那样创建并行的 utmpx 和 wtmpx 文件，相反，utmp 和 wtmp 文件包含了所有所需的信息。但为了与其他 UNIX 实现保持兼容，Linux 提供了传统的 utmp 和从 System V 演化而来的 utmpx API 来访问这些文件的内容。在 Linux 上，这两组 API 返回的信息是完全一样的。（这两组 API 之间的差别之一是 utmp API 中的一些函数是可重入的，而 utmpx 中的函数是不可重入的。）由于 SUSv3 规定了 utmpx API，因此从与其他 UNIX 实现保持可移植的角度出发，本章将介绍 utmpx 接口。

SUSv3 规范并没有覆盖到 utmpx API 的方方面面（如并没有规定 utmp 和 wtmp 文件的存放位置）。不同实现上的登录记账文件中包含的内容稍微存在一些差异，并且各种实现都提供了额外的登录记账函数，而 SUSv3 并没有对这些函数予以定义。

[Frisch, 2002]中的第 17 章对不同 UNIX 实现中 wtmp 和 utmp 文件在存放位置和使用方面的差异进行了总结。此外还介绍了 ac(1)命令的用法，这条命令可以用来对 wtmp 文件中的登录信息进行总结。

40.3 utmpx 结构

utmp 和 wtmp 文件包含 utmpx 记录。utmpx 结构式在<utmpx.h>中定义的，如程序清单 40-1 所示。

SUSv3 规范中的 utmpx 结构不包含 ut_host、ut_exit、ut_session 以及 ut_addr_v6 字段。在其他大多数实现中都存在 ut_host 和 ut_exit 字段；一些实现上还定义了 ut_session 字段；ut_addr_v6 是 Linux 特有的字段。SUSv3 规定了 ut_line 和 ut_user 字段，但并没有规定它们的长度。

在 utmpx 结构中 ut_addr_v6 字段的数据类型是 int32_t，它是一个 32 位的整数。

程序清单 40-1：utmpx 结构的定义

```
#define _GNU_SOURCE                /* Without _GNU_SOURCE the two field
struct exit_status {                  names below are prepended by "__" */
    short e_termination;           /* Process termination status (signal) */
    short e_exit;                  /* Process exit status */
};

#define __UT_LINESIZE    32
```

```
#define __UT_NAMESIZE     32
#define __UT_HOSTSIZE    256

struct utmpx {
    short ut_type;                      /* Type of record */
    pid_t ut_pid;                       /* PID of login process */
    char  ut_line[__UT_LINESIZE];       /* Terminal device name */
    char  ut_id[4];                     /* Suffix from terminal name, or
                                           ID field from inittab(5) */
    char  ut_user[__UT_NAMESIZE];       /* Username */
    char  ut_host[__UT_HOSTSIZE];       /* Hostname for remote login, or kernel
                                           version for run-level messages */
    struct exit_status ut_exit;         /* Exit status of process marked
                                           as DEAD_PROCESS (not filled
                                           in by init(8) on Linux) */
    long  ut_session;                   /* Session ID */
    struct timeval ut_tv;               /* Time when entry was made */
    int32_t ut_addr_v6[4];              /* IP address of remote host (IPv4
                                           address uses just ut_addr_v6[0],
                                           with other elements set to 0) */
    char __unused[20];                  /* Reserved for future use */
};
```

utmpx 结构中的所有字符串字段都以 null 结尾，除非值完全填满了相应的数组。

对于登录进程来讲，存储在 ut_line 和 ut_id 字段中的信息是从终端设备的名称中得出的。ut_line 字段包含了终端设备的完整的文件名。ut_id 字段包含了文件名的后缀——跟在 tty、pts 或 pty 后面的字符串（后两个分别表示 System-V 和 BSD 风格的伪终端）。因此，对于/dev/tty2 终端来讲，ut_line 的值为 tty2，ut_id 的值为 2。

在窗口环境中，一些终端模拟器使用 ut_session 字段来为终端窗口记录会话 ID（有关会话 ID 的介绍请参考 34.3 节）。

ut_type 字段是一个整数，它定义了写入文件的记录类型，其取值为下面一组常量中的一个（括号中给出了相应的数值）。

EMPTY (0)

这个记录不包含有效的记账信息。

RUN_LVL (1)

这个记录表明在系统启动或关闭时系统运行级别发生了变化。（有关运行级别的信息可以在 init(8)手册中找到。）要在<utmpx.h>中取得这个常量的定义就必须要定义_GNU_SOURCE 特性测试宏。

BOOT_TIME (2)

这个记录包含 ut_tv 字段中的系统启动时间。写入 RUN_LVL 和 BOOT_TIME 字段的进程通常是 init。这些记录会同时被写入 utmp 和 wtmp 文件。

NEW_TIME (3)

这个记录包含系统时钟变更之后的新时间，记录在 ut_tv 字段中。

OLD_TIME (4)

这个记录包含系统时钟变更之前的旧时间，记录在 ut_tv 字段中。当系统时钟发生变更时，

更多资源请访问稀酷客(www.ckook.com)

NTP daemon（或类似的进程）会将类型为 OLD_TIME 和 NEW_TIME 的记录写入到 utmp 和 wtmp 文件中。

INIT_PROCESS (5)

记录由 init 进程孵化的进程，如 getty 进程，细节信息请参考 inittab(5)手册。

LOGIN_PROCESS (6)

记录用户登录会话组长进程，如 login(1)进程。

USER_PROCESS (7)

记录用户进程，通常是登录会话，用户名会出现在 ut_user 字段中。登录会话可能是由 login(1)启动，或者也可能是由像 ftp 和 ssh 之类的提供远程登录工具的应用程序启动。

DEAD_PROCESS (8)

这个记录标识出已经退出的进程。

这里之所以给出了这些常量的数值是因为很多应用程序都要求这些常量的数值顺序与上面列出的顺序一致。如在 agetty 程序的源代码中可以发现下面这样的检查。

```
utp->ut_type >= INIT_PROCESS && utp->ut_type <= DEAD_PROCESS
```

类型为 INIT_PROCESS 通常对应于 getty(8)调用（或类似的程序，如 agetty(8)和 mingetty(8)）。在系统启动的时候，init 进程会为每个命令行和虚拟控制台创建一个子进程，每个子进程会执行 getty 程序。getty 程序会打开终端，提示用户输入用户名，然后执行 login(1)。当成功验证用户以及执行了其他一些动作之后，login 会创建一个子进程来执行用户登录 shell。这种登录会话的完整声明周期由写入 wtmp 文件的四个记录来表示，其顺序如下所示。

- 一个 INIT_PROCESS 记录，由 init 写入。
- 一个 LOGIN_PROCES 记录，由 getty 写入。
- 一个 USER_PROCESS 记录，由 login 写入。
- 一个 DEAD_PROCESS 记录，当 init 进程检测到 login 子进程死亡之后（发生在用户登出时）写入。

更多有关在用户登录期间 getty 和 login 的操作的细节可以在[Stevens & Rago, 2005]的第 9 章中找到。

一些版本的 init 会在更新 wtmp 文件之前孵化出 getty 进程，这样 init 和 getty 会在更新 wtmp 文件时形成竞争，从而导致 INIT_PROCESS 和 LOGIN_PROCESS 记录的写入顺序与正文中描述的顺序相反。

40.4 从 utmp 和 wtmp 文件中检索信息

本节介绍的函数能从包含 utmpx 格式记录的文件中获取读取信息。在默认情况下，这些函数使用标准的 utmp 文件，但使用 utmpxname()函数（稍后介绍）能够改变读取的文件。

这些函数都使用了当前位置（current location）的概念，它们会从文件中的当前位置来读取记录，每个函数都会更新这个位置。

setutxent()函数会将 utmp 文件的当前位置设置到文件的起始位置。

更多资源请访问稀酷客(www.ckook.com)

```
#include <utmpx.h>

void setutxent(void);
```

通常，在使用任意 getutx*()函数（稍后介绍）之前应该调用 setutxent()，这样就能避免因程序中已经调用到的第三方函数在之前用过这些函数而产生的混淆。根据所执行的任务的不同，在程序后面合适的地方可能需要调用 setutxent()。

当 utmp 文件没有被打开时，setutxent()函数和 getutx*()函数会打开这个文件。当用完这个文件之后可以使用 endutxent()函数来关闭这个文件。

```
#include <utmpx.h>

void endutxent(void);
```

getutxent()、getutxid()和 getutxline()函数从 utmp 文件中读取一个记录并返回一个指向 utmpx 结构（静态分配）的指针。

```
#include <utmpx.h>

struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *ut);
struct utmpx *getutxline(const struct utmpx *ut);

                    All return a pointer to a statically allocated utmpx structure,
                        or NULL if no matching record or EOF was encountered
```

getutxent()函数顺序读取 utmp 文件中的下一个记录。getutxid()和 getutxline()函数会从当前文件位置开始搜索与 ut 参数指向的 utmpx 结构中指定的标准匹配的一个记录。

getutxid()函数根据 ut 参数中 ut_type 和 ut_id 字段的值在 utmp 文件中搜索一个记录。

- 如果 ut_type 字段是 RUN_LVL、BOOT_TIME、NEW_TIME 或 OLD_TIME，那么 getutxid()会找出下一个 ut_type 字段与指定的值匹配的记录。（这种类型的记录与用户登录不相关。）这样就能够搜索与修改系统时间和运行级别相关的记录了。
- 如果 ut_type 字段的取值是剩余的有效值中的一个（INIT_PROCESS、LOGIN_PROCESS、USER_PROCESS 或 DEAD_PROCESS），那么 getutxent()会找出下一个 ut_type 字段与这些值中的任意一个匹配并且 ut_id 字段与 ut 参数中指定的值匹配的记录。这样就能够扫描文件来找出对应于某个特定终端的记录了。

getutxline()函数会向前搜索 ut_type 字段为 LOGIN_PROCESS 或 USER_PROCESS 并且 ut_line 字段与 ut 参数指定的值匹配的记录。这对于找出与用户登录相关的记录是非常有用的。

当搜索失败时（即达到文件尾时还没有找到匹配的记录），getutxid()和 getutxline()都返回 NULL。

在一些 UNIX 实现上，getutxline()和 getutxid()将用于返回 utmpx 结构的静态区域看成是某种高速缓冲存储（cache）。如果它们确定上一个 getutx*()调用放置在高速缓冲存储中的记录与 ut 指定的标准匹配，那么就不会执行文件读取操作，而是简单地再次返回同样的记录（SUSv3 允许这个行为）。因此为避免当在循环中调用 getutxline()和 getutxid()时重复返回同一个记录，必须要使用下面的代码清除这个静态数据结构。

更多资源请访问稀酷客(www.ckook.com)

```
struct utmpx *res = NULL;

/* Other code omitted */

if (res != NULL)            /* If 'res' was set via a previous call */
    memset(res, 0, sizeof(struct utmpx));
res = getutxline(&ut);
```

glibc 实现不会进行这样的缓存，但从可移植性的角度出发，在编写程序时永远不要使用这种技术。

由于 getutx*()函数返回的是一个指向静态分配的结构的指针，因此它们是不可重入的。GNU C 库提供了传统的 utmp 函数的可重入版本（getutent_r()、getutid_r()以及 getutline_r()），但并没有为 utmpx 函数提供可重入版本。（SUSv3 并没有规定可重入版本。）

在默认情况下，所有 getutx*()函数都使用标准的 utmp 文件。如果需要使用另一个文件，如 wtmp 文件，那么必须要首先调用 utmpxname()并指定目标路径名。

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

int utmpxname(const char *file);
```

Returns 0 on success, or −1 on error

utmpxname()函数仅仅将传入的路径名复制一份，它不会打开文件，但会关闭之前由其他调用打开的所有文件。这表示就算指定了一个无效的路径名，utmpxname()也不会返回错误。相反，当后面调用某个 getutx*()函数发现无法打开文件时会返回一个错误（即 NULL，errno 被设为 ENOENT）。

虽然 SUSv3 并没有对此进行规定，但大多数 UNIX 实现提供了 utmpxname()或类似的 utmpname()函数。

示例程序

程序清单 40-2 中的程序使用了本节中介绍的一些函数来输出一个 utmpx 格式文件的内容。下面的 shell 会话日志给出了使用这个程序输出/var/run/utmp（当没有调用 utmpxname()时这些函数会默认使用该文件）的内容时得到的结果。

```
$ ./dump_utmpx
user     type        PID line   id  host      date/time
LOGIN    LOGIN_PR   1761 tty1   1            Sat Oct 23 09:29:37 2010
LOGIN    LOGIN_PR   1762 tty2   2            Sat Oct 23 09:29:37 2010
lynley   USER_PR   10482 tty3   3            Sat Oct 23 10:19:43 2010
david    USER_PR    9664 tty4   4            Sat Oct 23 10:07:50 2010
liz      USER_PR    1985 tty5   5            Sat Oct 23 10:50:12 2010
mtk      USER_PR   10111 pts/0  /0           Sat Oct 23 09:30:57 2010
```

限于篇幅，这里将程序的很多输出都省去了。上面 tty1 到 tty5 是表示虚拟控制台上的登录（/dev/tty[1-6]）。输出中的最后一行表示伪终端上的 xterm 会话。

从下面输出/var/log/wtmp 文件时所产生的结果可以看出当一个用户登录和登出时会向 wtmp 文件写入两个记录。（程序的其他不相关的所有输出都被省去了。）在顺序搜索 wtmp 文件（使用 getutxline()）时可以使用 ut_line 来匹配这些记录。

更多资源请访问稀酷客(www.ckook.com)

```
$ ./dump_utmpx /var/log/wtmp
user     type        PID line   id  host      date/time
lynley   USER_PR   10482 tty3   3            Sat Oct 23 10:19:43 2010
         DEAD_PR   10482 tty3   3   2.4.20-4G Sat Oct 23 10:32:54 2010
```

程序清单 40-2：显示一个 utmpx 格式文件的内容

loginacct/dump_utmpx.c

```
#define _GNU_SOURCE
#include <time.h>
#include <utmpx.h>
#include <paths.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct utmpx *ut;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [utmp-pathname]\n", argv[0]);

    if (argc > 1)                /* Use alternate file if supplied */
        if (utmpxname(argv[1]) == -1)
            errExit("utmpxname");

    setutxent();

    printf("user     type        PID line   id  host      date/time\n");

    while ((ut = getutxent()) != NULL) {        /* Sequential scan to EOF */
        printf("%-8s ", ut->ut_user);
        printf("%-9.9s ",
                (ut->ut_type == EMPTY) ?         "EMPTY" :
                (ut->ut_type == RUN_LVL) ?       "RUN_LVL" :
                (ut->ut_type == BOOT_TIME) ?     "BOOT_TIME" :
                (ut->ut_type == NEW_TIME) ?      "NEW_TIME" :
                (ut->ut_type == OLD_TIME) ?      "OLD_TIME" :
                (ut->ut_type == INIT_PROCESS) ?  "INIT_PR" :
                (ut->ut_type == LOGIN_PROCESS) ? "LOGIN_PR" :
                (ut->ut_type == USER_PROCESS) ?  "USER_PR" :
                (ut->ut_type == DEAD_PROCESS) ?  "DEAD_PR" : "???");
        printf("%5ld %-6.6s %-3.5s %-9.9s ", (long) ut->ut_pid,
                ut->ut_line, ut->ut_id, ut->ut_host);
        printf("%s", ctime((time_t *) &(ut->ut_tv.tv_sec)));
    }

    endutxent();
    exit(EXIT_SUCCESS);
}
```

loginacct/dump_utmpx.c

40.5 获取登录名称：getlogin()

getlogin()函数返回登录到调用进程的控制终端的用户名，它会使用在 utmp 文件中维护的信息。

更多资源请访问稀酷客(www.ckook.com)

```
#include <unistd.h>

char *getlogin(void);
```

Returns pointer to username string, or NULL on error

getlogin()函数会调用 ttyname()（参见 62.10 节）来找出与调用进程的标准输入相关联的终端名，接着它将搜索 utmp 文件以找出 ut_line 值与终端名匹配的记录。如果找到了匹配的记录，那么 getlogin()会返回记录中的 ut_user 字符串。

如果没有找到匹配的记录或者发生了错误，那么 getlogin()会返回 NULL 并设置 errno 来标示错误。getlogin()可能会失败的一个原因是进程没有一个与其标准输入相关联的终端（ENOTTY），这可能是因为进程本身是一个 daemon。另一个可能的原因是终端会话并没有记录在 utmp 文件中，如一些软件终端模拟器不会在 utmp 文件中创建条目。

即使当一个用户 ID 在/etc/passwd 文件中拥有多个登录名时（不常见），getlogin()还是能够返回登录进这个终端的实际用户名，因为它依赖的是 utmp 文件。相反，getpwuid(getuid()) 总是会返回/etc/passwd 中第一个匹配的记录，不管登录名是什么。

SUSv3 规定了 getlogin()的一个可重入版本 getlogin_r()，glibc 提供了这个函数。

LOGNAME 环境变量也可以用来找出用户的登录名。但用户可以改变这个变量的值，这表示无法使用这个变量来安全地识别出一个用户。

40.6 为登录会话更新 utmp 和 wtmp 文件

在编写一个创建登录会话的应用程序（如像 login 或 sshd 那样）时应该要按照下面的步骤更新 utmp 和 wtmp 文件。

- 在登录的时候应该向 utmp 文件写入一条记录表明这个用户登录进系统了。应用程序必须要检查在 utmp 文件中是否存在这个终端的记录。如果已经存在了一个记录，那么它将重写这个记录，否则就在文件后面附加一个新记录。通常调用 pututxline()（稍后介绍）就足以确保正确执行这些步骤了（具体示例可参见程序清单 40-3）。输出的 utmpx 记录至少需要填充 ut_type、ut_user、ut_tv、ut_pid、ut_id 以及 ut_line 字段。ut_type 字段应该被设置成 USER_PROCESS。ut_id 字段应该包含用户登录的设备名（即终端或伪终端）的后缀，ut_line 字段应该包含登录设备的名称中去除了开头的/dev/的字符串。（运行程序清单 40-2 中的程序时产生的输出会显示这两个字段的内容。）一个包含完全一样的信息的记录会被附加到 wtmp 文件中。

utmp 文件中的记录以终端名（ut_line 和 ut_id 字段）作为唯一键。

- 在登出的时候应该删除之前写入 utmp 文件的记录，这是通过创建一个记录并将 ut_type 设置为 DEAD_PROCESS、同时将 ut_id 和 ut_line 设置为登录时写入的记录中相应字段的值并将 ut_user 字段的值置零来完成的。这个记录会覆盖之前的记录，同时这个记录的一个副本会被附加到 wtmp 文件中。

如果在登出时没有成功清理 utmp 中的相关记录，可能因为程序崩溃，那么在下一次重启的时候，init 会自动清理这些记录并将记录的 ut_type 设置为 DEAD_PROCESS 以及将记

更多资源请访问稀酷客(www.ckook.com)

录中其他字段置零。

通常 utmp 和 wtmp 文件是受保护的，只有特权用户可以更新这些文件。getlogin()的精确程度依赖于 utmp 文件的完整性。正因为这个原因以及其他一些原因，在 utmp 和 wtmp 文件的权限设置中应该永远都不允许非特权用户写这两个文件。

哪些程序会产生一个登录会话呢？正如读者所想的那样，通过 login、telnet 以及 ssh 登录会记录在登录记账文件中。大多数 ftp 实现也会创建登录记账记录。但系统上每个打开的终端窗口或调用 su 时会创建登录记账记录吗？这个问题的答案因 UNIX 实现的不同而不同。

在一些终端模拟程序（如 xterm）中，可以使用命令行选项以及其他一些机制来确定程序是否更新登录记账文件。

pututxline()函数会将 ut 指向的 utmpx 结构写入到/var/run/utmp 文件中（或者如果之前调用了 utmpxname()的话将是另一个文件）。

```
#include <utmpx.h>

struct utmpx *pututxline(const struct utmpx *ut);
```

Returns pointer to copy of successfully updated record on success, or NULL on error

在写入记录之前，pututxline()首先会使用 getutxid()向前搜索一个可被重写的记录。如果找到了这样的记录，那么会重写该记录，否则就会在文件尾附加一个新记录。在很多情况下，应用程序在调用 pututxline()之前会调用其中一个 getutx*()函数，因为这个函数会将当前文件位置设定到正确的记录——即与 getutxid()系列函数中 ut 指向的 utmpx 结构中的标准匹配的记录。如果 pututxline()能够确定已经重置过了当前文件位置，那么就不会调用 getutxid()。

如果 pututxline()在内部调用了 getutxid()，那么这个调用不会改变 getutx*()函数用来返回 utmpx 结构的静态区域。SUSv3 要求实现遵循这种行为。

在更新 wtmp 文件时仅仅是简单地打开文件并在文件尾附加一个记录。由于这是一个标准操作，因此 glibc 将其封装进了 updwtmpx()函数。

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

void updwtmpx(char *wtmpx_file, struct utmpx *ut);
```

updwtmpx()函数将 ut 指向的 utmpx 记录附加到 wtmpx_file 指定的文件尾。

SUSv3 没有规定 updwtmpx()，这个函数只出现了一些 UNIX 实现中，而其他实现则提供了相关函数——login(3)、logout(3)以及 logwtmp(3)——这些函数位于 glibc 中并且手册也对这些函数进行了描述。如果不存在这样的函数，那么就需要自己编写实现相同功能的函数了。（这些函数的实现并不复杂。）

示例程序

程序清单 40-3 使用了这一节中介绍的函数来更新 utmp 和 wtmp 文件。这个程序先执行记录由命令行指定的用户的登录操作所需的对 utmp 和 wtmp 文件的更新，然后睡眠的几秒

更多资源请访问稀酷客(www.ckook.com)

钟之后再登出用户。通常，此类操作会与用户的登录会话的创建和终止相关联。这个程序使用了 ttyname()来获取与文件描述符相关联的终端设备的名称，ttyname()将在第 62.10 节中予以介绍。

下面的 shell 会话日志演示了程序清单 40-3 中的程序的操作。假设程序已经拥有了更新登录记账文件的权限，然后使用这个程序来为用户 mtk 创建一个记录。

```
$ su
Password:
# ./utmpx_login mtk
Creating login entries in utmp and wtmp
        using pid 1471, line pts/7, id /7
Type Control-Z to suspend program
[1]+  Stopped                 ./utmpx_login mtk
```

在 utmpx_login 程序睡眠的过程中输入 Control-Z 以挂起该程序并将其放到后台。接着使用程序清单 40-2 中的程序来查看 utmp 文件中的内容。

```
# ./dump_utmpx /var/run/utmp
user     type        PID line   id  host      date/time
cecilia  USER_PR     249 tty1   1             Fri Feb  1 21:39:07 2008
mtk      USER_PR    1471 pts/7  /7            Fri Feb  1 22:08:06 2008
# who
cecilia  tty1     Feb  1 21:39
mtk      pts/7    Feb  1 22:08
```

上面使用了 who(1)命令来表明 who 的输出源自 utmp 文件。

接着使用程序来查看 wtmp 文件中的内容。

```
# ./dump_utmpx /var/log/wtmp
user     type        PID line   id  host      date/time
cecilia  USER_PR     249 tty1   1             Fri Feb  1 21:39:07 2008
mtk      USER_PR    1471 pts/7  /7            Fri Feb  1 22:08:06 2008
# last mtk
mtk      pts/7                      Fri Feb  1 22:08   still logged in
```

上面使用了 last(1)命令来表明 last 的输出源自 wtmp 文件。（限于篇幅，这里给出的 shell 会话日志中 dump_utmpx 和 last 命令输出已经删除了与本节讨论主题无关的内容。）

接着使用 fg 命令将 utmpx_login 程序恢复到前台。程序随后就会将登出记录写入 utmp 和 wtmp 文件。

```
# fg
./utmpx_login mtk
Creating logout entries in utmp and wtmp
```

接着再次查看 utmp 文件中的内容，从中可以看出 utmp 中的记录被重写了。

```
# ./dump_utmpx /var/run/utmp
user     type        PID line   id  host      date/time
cecilia  USER_PR     249 tty1   1             Fri Feb  1 21:39:07 2008
         DEAD_PR    1471 pts/7  /7            Fri Feb  1 22:09:09 2008
# who
cecilia  tty1     Feb  1 21:39
```

输出中的最后一行表明 who 忽略了 DEAD_PROCESS 记录。

在查看 wtmp 文件之后可以看出 wtmp 记录已经被附加进去了。

更多资源请访问稀酷客(www.ckook.com)

```
# ./dump_utmpx /var/log/wtmp
user     type        PID line   id  host      date/time
cecilia  USER_PR     249 tty1   1             Fri Feb  1 21:39:07 2008
mtk      USER_PR    1471 pts/7  /7            Fri Feb  1 22:08:06 2008
         DEAD_PR    1471 pts/7  /7            Fri Feb  1 22:09:09 2008
# last mtk
mtk      pts/7                      Fri Feb  1 22:08 - 22:09  (00:01)
```

上面输出中的最后一行表明 last 匹配了 wtmp 文件中的登录和登出记录，从而能看出整个登录会话的开始时间和结束时间。

程序清单 40-3：更新 utmp 和 wtmp 文件

loginacct/utmpx_login.c

```
#define _GNU_SOURCE
#include <time.h>
#include <utmpx.h>
#include <paths.h>              /* Definitions of _PATH_UTMP and _PATH_WTMP */
#include "tlpi_hdr.h"
int
main(int argc, char *argv[])
{
    struct utmpx ut;
    char *devName;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s username [sleep-time]\n", argv[0]);

    /* Initialize login record for utmp and wtmp files */

    memset(&ut, 0, sizeof(struct utmpx));
    ut.ut_type = USER_PROCESS;          /* This is a user login */
    strncpy(ut.ut_user, argv[1], sizeof(ut.ut_user));
    if (time((time_t *) &ut.ut_tv.tv_sec) == -1)
        errExit("time");                /* Stamp with current time */
    ut.ut_pid = getpid();

    /* Set ut_line and ut_id based on the terminal associated with
       'stdin'. This code assumes terminals named "/dev/[pt]t[sy]*".
       The "/dev/" dirname is 5 characters; the "[pt]t[sy]" filename
       prefix is 3 characters (making 8 characters in all). */

    devName = ttyname(STDIN_FILENO);
    if (devName == NULL)
        errExit("ttyname");
    if (strlen(devName) <= 8)           /* Should never happen */
        fatal("Terminal name is too short: %s", devName);

    strncpy(ut.ut_line, devName + 5, sizeof(ut.ut_line));
    strncpy(ut.ut_id, devName + 8, sizeof(ut.ut_id));

    printf("Creating login entries in utmp and wtmp\n");
    printf("        using pid %ld, line %.*s, id %.*s\n",
            (long) ut.ut_pid, (int) sizeof(ut.ut_line), ut.ut_line,
            (int) sizeof(ut.ut_id), ut.ut_id);

    setutxent();                        /* Rewind to start of utmp file */
    if (pututxline(&ut) == NULL)        /* Write login record to utmp */
```

```
    errExit("pututxline");
updwtmpx(_PATH_WTMP, &ut);          /* Append login record to wtmp */

/* Sleep a while, so we can examine utmp and wtmp files */

sleep((argc > 2) ? getInt(argv[2], GN_NONNEG, "sleep-time") : 15);

/* Now do a "logout"; use values from previously initialized 'ut',
   except for changes below */

ut.ut_type = DEAD_PROCESS;          /* Required for logout record */
time((time_t *) &ut.ut_tv.tv_sec);  /* Stamp with logout time */
memset(&ut.ut_user, 0, sizeof(ut.ut_user));
                                    /* Logout record has null username */
    printf("Creating logout entries in utmp and wtmp\n");
    setutxent();                    /* Rewind to start of utmp file */
    if (pututxline(&ut) == NULL)    /* Overwrite previous utmp record */
        errExit("pututxline");
    updwtmpx(_PATH_WTMP, &ut);      /* Append logout record to wtmp */

    endutxent();
    exit(EXIT_SUCCESS);
}
```

loginacct/utmpx_login.c

40.7 lastlog 文件

lastlog 文件记录着每个用户最近一次登录到系统的时间。（它与 wtmp 文件不同，wtmp 文件记录着所有用户的登录和登出行为。）login 程序通过 lastlog 文件能够通知用户（在新登录会话开始的时候）他们上次登录的时间。提供登录服务的应用程序除了要更新 utmp 和 wtmp 文件之外还应该更新 lastlog 文件。

与 utmp 和 wtmp 文件一样，不同系统实现中 lastlog 文件的存放位置和格式可能会存在差异。（一些 UNIX 实现并没有提供这个文件。）在 Linux 上，这个文件位于/var/log/lastlog，<paths.h>文件中定义的常量_PATH_LASTLOG 指向了这个位置。与 utmp 和 wtmp 文件一样，lastlog 文件通常也是受保护的，这样所有用户都能读取这个文件但只有特权进程才能够更新这个文件。

lastlog 文件中的记录的格式如下所示（在<lastlog.h>中定义）。

```
#define UT_NAMESIZE          32
#define UT_HOSTSIZE         256

struct lastlog {
    time_t ll_time;                 /* Time of last login */
    char   ll_line[UT_NAMESIZE];    /* Terminal for remote login */
    char   ll_host[UT_HOSTSIZE];    /* Hostname for remote login */
};
```

注意这些记录中并没有包含用户名或用户 ID。lastlog 文件中的记录是用用户 ID 作为索引的，因此要找出用户 ID 为 1000 的 lastlog 记录就需要到文件的相应位置处（1000 * sizeof(struct lastlog)）查找。程序清单 40-4 对此进行了演示，通过这个程序读者能够查看在命令行中列出的用户的 lastlog 记录，其功能与 lastlog(1)命令的功能类似。下面是运行这个程序时产生的输出。

```
$ ./view_lastlog annie paulh
annie    tty2                     Mon Jan 17 11:00:12 2011
paulh    pts/11                   Sat Aug 14 09:22:14 2010
```

更新 lastlog 文件时会打开文件，寻找到正确的位置，然后执行一个写入操作。

由于 lastlog 文件是以用户 ID 为索引的，因此无法区分拥有同样的用户 ID 的不同用户名的登录行为。（在 8.1 节中层指出过多个登录名拥有同样的用户 ID 是可能的，虽然这种情况并不常见。）

程序清单 40-4：显示 lastlog 文件中的信息

loginacct/view_lastlog.c

```
#include <time.h>
#include <lastlog.h>
#include <paths.h>                      /* Definition of _PATH_LASTLOG */
#include <fcntl.h>
#include "ugid_functions.h"             /* Declaration of userIdFromName() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct lastlog llog;
    int fd, j;
    uid_t uid;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [username...]\n", argv[0]);

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

    for (j = 1; j < argc; j++) {
        uid = userIdFromName(argv[j]);
        if (uid == -1) {
            printf("No such user: %s\n", argv[j]);
            continue;
        }

        if (lseek(fd, uid * sizeof(struct lastlog), SEEK_SET) == -1)
            errExit("lseek");

        if (read(fd, &llog, sizeof(struct lastlog)) <= 0) {
            printf("read failed for %s\n", argv[j]);    /* EOF or error */
            continue;
        }

        printf("%-8.8s %-6.6s %-20.20s %s", argv[j], llog.ll_line,
                llog.ll_host, ctime((time_t *) &llog.ll_time));
    }

    close(fd);
    exit(EXIT_SUCCESS);
}
```

loginacct/view_lastlog.c

40.8 总结

登录记账记录着当前登录的用户以及过去登录过系统的用户。这类信息维护在三个文件中：utmp 文件维护了所有当前登录进系统的用户记录；wtmp 文件维护了所有登录和登出行为的审计信息；lastlog 文件记录着每个用户最近一次登录系统的时间。很多命令，如 who 和 last，都使用了这些文件中的信息。

C 库提供了读取和更新登录记账文件中的信息的函数。提供登录服务的应用程序应该使用这些函数来更新登录记账文件，这样依赖于这些信息的命令才能够表现出正确的行为。

更多信息

除了 utmp(5)手册之外，找到更多有关登录记账函数的信息的最有用的地方是各种使用这些函数的应用程序的源代码。如可以阅读 mingetty（或 agetty）、login、init、telnet、ssh 以及 ftp 的源代码。

40.9 习题

40-1. 实现 getlogin()。在 40.5 节中曾提到过当进程运行在一些软件终端模拟器下时 getlogin()可能无法正确工作，在那种情况下就在虚拟控制台中进行测试。

40-2. 修改程序清单 40-3 中的程序（utmpx_login.c）使它除了更新 utmp 和 wtmp 文件之外还更新 lastlog 文件。

40-3. 阅读 login(3)、logout(3)以及 logwtmp(3)的手册。实现这些函数。

40-4. 实现一个简单的 who(1)。

更多资源请访问稀酷客(www.ckook.com)

第41章

共享库基础

共享库是一种将库函数打包成一个单元使之能够在运行时被多个进程共享的技术。这种技术能够节省磁盘空间和 RAM。本章将介绍共享库的基础知识，下一章将介绍共享库的几个高级特性。

41.1 目标库

构建程序的一种方式是简单地将每一个源文件编译成目标文件，然后将这些目标文件链接在一起组成一个可执行程序，如下所示。

```
$ cc -g -c prog.c mod1.c mod2.c mod3.c
$ cc -g -o prog_nolib prog.o mod1.o mod2.o mod3.o
```

> 链接实际上是由一个单独的链接器程序 ld 来完成的。当使用 cc（或 gcc）命令链接一个程序时，编译器会在幕后调用 ld。在 Linux 上应该总是通过 gcc 间接地调用链接器，因为 gcc 能够确保使用正确的选项来调用 ld 并将程序与正确的库文件链接起来。

在很多情况下，源代码文件也可以被多个程序共享。因此要降低工作量的第一步就是将这些源代码文件只编译一次，然后在需要的时候将它们链接进不同的可执行文件中。虽然这项技术能够节省编译时间，但其缺点是在链接的时候仍然需要为所有目标文件命名。此外，大量的目标文件会散落在系统上的各个目录中，从而造成目录中内容的混乱。

为解决这个问题，可以将一组目标文件组织成一个被称为对象库的单元。对象库分为两种：静态的和共享的。共享库是一种更加现代化的对象库，它比静态库更具优势，41.3 节将会对此予以介绍。

题外话：在编译程序时包含调试器信息

在上面的 cc 命令中使用了-g 选项以在编译过的程序中包含调试信息。一般来讲，创建允许调试的程序和库是一种比较好的做法。（在早期，有时候会忽略调试信息，这样产生的可执行文件会占用更少的磁盘和 RAM，但现在磁盘和 RAM 已经非常便宜了。）

此外，在一些架构上，如 x86-32，不应该指定–fomit–frame–pointer 选项，因为这会使得

更多资源请访问稀酷客(www.ckook.com)

无法调试。（在一些架构上，如 x86-64，这个选项是默认启用的，因为它不会防止调试。）出于同样的原因，可执行文件和库不应该使用 strip(1)删除调试信息。

41.2 静态库

在开始讨论共享库之前首先对静态库作一个简短的介绍，这样读者就能够弄清楚共享库与静态库之间的差别以及共享库所具备的优势了。

静态库也被称为归档文件，它是 UNIX 系统提供的第一种库。静态库能带来下列好处。

- 可以将一组经常被用到的目标文件组织进单个库文件，这样就可以使用它来构建多个可执行程序并且在构建各个应用程序的时候无需重新编译原来的源代码文件。
- 链接命令变得更加简单了。在链接命令行中只需要指定静态库的名称即可，而无需一个个地列出目标文件了。链接器知道如何搜索静态库并将可执行程序需要的对象抽取出来。

创建和维护静态库

从结果上来看，静态库实际上就是一个保存所有被添加到其中的目标文件的副本的文件。这个归档文件还记录着每个目标文件的各种特性，包括文件权限、数字用户和组 ID 以及最后修改时间。根据惯例，静态库的名称的形式为 libname.a。

使用 ar(1)命令能够创建和维护静态库，其通用形式如下所示。

```
$ ar options archive object-file...
```

options 参数由一系列的字母构成，其中一个是操作代码，其他是能够影响操作的执行的修饰符。下面是一些常用的操作代码。

- r（替换）：将一个目标文件插入到归档文件中并取代同名的目标文件。这个创建和更新归档文件的标准方法，使用下面的命令可以构建一个归档文件。

```
$ cc -g -c mod1.c mod2.c mod3.c
$ ar r libdemo.a mod1.o mod2.o mod3.o
$ rm mod1.o mod2.o mod3.o
```

从上面可以看出，在构建完库之后可以根据需要删除原始的目标文件，因为已经不再需要它们了。

- t（目录表）：显示归档中的目录表。在默认情况下只会列出归档文件中目标文件的名称。添加 v（verbose）修饰符之后可以看到记录在归档文件中的各个目标文件的其他所有特性，如下面的例子所示。

```
$ ar tv libdemo.a
rw-r--r-- 1000/100 1001016 Nov 15 12:26 2009 mod1.o
rw-r--r-- 1000/100 406668 Nov 15 12:21 2009 mod2.o
rw-r--r-- 1000/100  46672 Nov 15 12:21 2009 mod3.o
```

从左至右每个目标文件的特性为被添加到归档文件中时的权限、用户 ID 和组 ID、大小以及上次修改的日志和时间。

- d（删除）：从归档文件中删除一个模块，如下面的例子所示。

```
$ ar d libdemo.a mod3.o
```

使用静态库

将程序与静态库链接起来存在两种方式。第一种是在链接命令中指定静态库的名称，如下所示。

更多资源请访问稀酷客(www.ckook.com)

```
$ cc -g -c prog.c
$ cc -g -o prog prog.o libdemo.a
```

或者将静态库放在链接器搜索的其中一个标准目录中（如/usr/lib），然后使用-l 选项指定库名（即库的文件名去除了 lib 前缀和.a 后缀）。

```
$ cc -g -o prog prog.o -ldemo
```

如果库不位于链接器搜索的目录中，那么可以只用-L 选项指定链接器应该搜索这个额外的目录。

```
$ cc -g -o prog prog.o -Lmylibdir -ldemo
```

虽然一个静态库可以包含很多目标模块，但链接器只会包含那些程序需要的模块。

在链接完程序之后可以按照通常的方式运行这个程序。

```
$ ./prog
Called mod1-x1
Called mod2-x2
```

41.3 共享库概述

将程序与静态库链接起来时（或没有使用静态库），得到的可执行文件会包含所有被链接进程序的目标文件的副本。这样当几个不同的可执行程序使用了同样的目标模块时，每个可执行程序会拥有自己的目标模块的副本。这种代码的冗余存在几个缺点。

- 存储同一个目标模块的多个副本会浪费磁盘空间，并且所浪费的空间是比较大的。
- 如果几个使用了同一模块的程序在同一时刻运行，那么每个程序会独立地在虚拟内存中保存一份目标模块的副本，从而提高系统中虚拟内存的整体使用量。
- 如果需要修改一个静态库中的一个目标模块（可能是因为安全性或需要修正 bug），那么所有使用那个模块的可执行文件都必须要重新进行链接以合并这个变更。这个缺点还会导致系统管理员需要弄清楚哪些应用程序链接了这个库。

共享库就是设计用来解决这些缺点的。共享库的关键思想是目标模块的单个副本由所有需要这些模块的程序共享。目标模块不会被复制到链接过的可执行文件中，相反，当第一个需要共享库中的模块的程序启动时，库的单个副本就会在运行时被加载进内存。当后面使用同一共享库的其他程序启动时，它们会使用已经被加载进内存的库的副本。使用共享库意味着可执行程序需要的磁盘空间和虚拟内存（在运行的时候）更少了。

> 虽然共享库的代码是由多个进程共享的，但其中的变量却不是的。每个使用库的进程会拥有自己的在库中定义的全局和静态变量的副本。

共享库还具备下列优势。

- 由于整个程序的大小变得更小了，因此在一些情况下，程序可以完全被加载进内存中，从而能够更快地启动程序。这一点只有在大型共享库正在被其他程序使用的情况下才成立。第一个加载共享库的程序实际上在启动时会花费更长的时间，因为必须要先找到共享库并将其加载到内存中。
- 由于目标模块没有被复制进可执行文件中，而是在共享库中集中维护的，因此在修改目标模块时（需遵循 41.8 节中介绍的限制）无需重新链接程序就能够看到变更，甚至在运行着的程序正在使用共享库的现有版本的时候也能够进行这样的变更。

这项新增功能的主要开销如下所述。

- 在概念上以及创建共享库和构建使用共享库的程序的实践上，共享库比静态库更复杂。

更多资源请访问稀酷客(www.ckook.com)

- 共享库在编译时必须要使用位置独立的代码（在 41.4.2 节中予以介绍），这在大多数架构上都会带来性能开销，因为它需要使用额外的一个寄存器（[Hubicka, 2003]）。
- 在运行时必须要执行符号重定位。在符号重定位期间，需要将对共享库中每个符号（变量或函数）的引用修改成符号在虚拟内存中的实际运行时位置。由于存在这个重定位的过程，与静态链接程序相比，一个使用共享库的程序或多或少需要花费一些时间来执行这个过程。

共享库的另一种用法是作为 Java NativeInterface (JNI)中的一个构建块，它允许 Java 代码通过调用共享库中的 C 函数直接访问底层操作系统的特性，更多信息可参考[Liang, 1999]和[Rochkind, 2004]。

41.4 创建和使用共享库——首回合

为了理解共享库的操作方式，下面开始介绍构建和使用一个共享库所需完成的最少步骤，在介绍的过程中会忽略平时使用的共享库文件命名规范。遵循第 41.6 节中介绍的惯例允许程序自动加载它们所需的共享库的最新版本，同时也允许一个库的多个相互不兼容的版本（所谓的主版本）和谐地共存。

在本章中，我们只关心 Executable and Linking Format（ELF）共享库，因为现代版本的 Linux 以及很多其他 UNIX 实现的可执行文件和共享库都采用了 ELF 格式。

ELF 取代了较早以前的 a.out 和 COFF 格式。

41.4.1 创建一个共享库

为构建之前创建的静态库的共享版本，需要执行下面的步骤。

```
$ gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
$ gcc -g -shared -o libfoo.so mod1.o mod2.o mod3.o
```

第一个命令创建了三个将要被放到库中的目标模块。（下一节将对 cc –fPIC 选项进行解释。）cc –shared 命令创建了一个包含这三个目标模块的共享库。

根据惯例，共享库的前缀为 lib，后缀为.so（表示 shared object）。

在上面的例子中使用了 gcc 命令，而并没有使用与之等价的 cc 命令，这是为了突出用来创建共享库的命令行选项是依赖于编译器的，在另一个 UNIX 实现上使用一个不同的 C 编译器可能会需要使用不同的选项。

注意可以将编译源代码文件和创建共享库放在一个命令中执行。

```
$ gcc -g -fPIC -Wall mod1.c mod2.c mod3.c -shared -o libfoo.so
```

这里为了清楚区分编译和构建库两个步骤，所以在本章给出的例子中使用了两个独立的命令。

与静态库不同，可以向之前构建的共享库中添加单个目标模块，也可以从中删除单个目标模块。与普通的可执行文件一样，共享库中的目标文件不再维护不同的身份。

41.4.2 位置独立的代码

cc-fPIC 选项指定编译器应该生成位置独立的代码，这会改变编译器生成执行特定操作的代码的方式，包括访问全局、静态和外部变量，访问字符串常量，以及获取函数的地址。这些变更使得代码可以在运行时被放置在任意一个虚拟地址处。这一点对于共享库来讲是必需的，因为在链接的时候是无法知道共享库代码位于内存的何处的。（一个共享库在运行时所处的内存位置依赖于

更多资源请访问稀酷客(www.ckook.com)

很多因素，如加载这个库的程序已经占用的内存数量和这个程序已经加载的其他共享库。）

在 Linux/x86-32 上，可以使用不加–fPIC 选项编译的模块来创建共享库。但这样做的话会丢失共享库的一些优点，因为包含依赖于位置的内存引用的程序文本页面不会在进程间共享。在一些架构上是无法在不加–fPIC 选项的情况下构建共享库的。

为了确定一个既有目标文件在编译时是否使用了–fPIC 选项，可以使用下面两个命令中的一个来检查目标文件符号表中是否存在名称_GLOBAL_OFFSET_TABLE_。

```
$ nm mod1.o | grep _GLOBAL_OFFSET_TABLE_
$ readelf -s mod1.o | grep _GLOBAL_OFFSET_TABLE_
```

相应地，如果下面两个相互等价的命令中的任意一个产生了任何输出，那么指定的共享库中至少存在一个目标模块在编译时没有指定–fPIC 选项。

```
$ objdump --all-headers libfoo.so | grep TEXTREL
$ readelf -d libfoo.so | grep TEXTREL
```

字符串 TEXTREL 表示存在一个目标模块，其文本段中包含需要运行时重定位的引用。

在 41.5 节中将会介绍更多有关 nm、readelf 以及 objdump 命令的信息。

41.4.3 使用一个共享库

为了使用一个共享库就需要做两件事情，而使用静态库的程序则无需完成这两件事情。

- 由于可执行文件不再包含它所需的目标文件的副本，因此它必须要通过某种机制找出在运行时所需的共享库。这是通过在链接阶段将共享库的名称嵌入可执行文件中来完成的。（在 ELF 中，库依赖性是记录在可执行文件的 DT_NEEDED 标签中的。）一个程序所依赖的所有共享库列表被称为程序的动态依赖列表。
- 在运行时必须要存在某种机制来解析嵌入的库名——即找出与在可执行文件中指定的名称对应的共享库文件——接着如果库不在内存中的话就将库加载进内存。

将程序与共享库链接起来时自动会将库的名字嵌入可执行文件中。

```
$ gcc -g -Wall -o prog prog.c libfoo.so
```

如果现在运行这个程序，那么就会收到下面的错误消息。

```
$ ./prog
./prog: error in loading shared libraries: libfoo.so: cannot
open shared object file: No such file or directory
```

解决这个问题就需要做第二件事情：动态链接，即在运行时解析内嵌的库名。这个任务是由动态链接器（也称为动态链接加载器或运行时链接器）来完成的。动态链接器本身也是一个共享库，其名称为/lib/ld-linux.so.2，所有使用共享库的 ELF 可执行文件都会用到这个共享库。

路径名/lib/ld-linux.so.2 通常是一个指向动态链接器可执行文件的符号链接。这个文件的名称为 ld-version.so，其中 version 表示安装在系统上的 glibc 的版本——如 ld-2.11.so。在一些架构上，动态链接器的路径名是不同的。如在 IA-64 上，动态链接器符号链接的名称为/lib/ld-linux-ia64.so.2。

动态链接器会检查程序所需的共享库清单并使用一组预先定义好的规则来在文件系统上找出相关的库文件。其中一些规则指定了一组存放共享库的标准目录。如很多共享库位于/lib 和/usr/lib 中。之所以出现上面的错误消息是因为程序所需的库位于当前工作目录中，而不位于动态链接器搜索的标准目录清单中。

一些架构（如 zSeries、PowerPC64 以及 x86-64）同时支持执行 32 位和 64 位的程序。在此类系统上，32 位的库位于*/lib 子目录中，64 位的库位于*/lib64 子目录中。

更多资源请访问稀酷客(www.ckook.com)

LD_LIBRARY_PATH 环境变量

通知动态链接器一个共享库位于一个非标准目录中的一种方法是将该目录添加到 LD_LIBRARY_PATH 环境变量中以分号分隔的目录列表中。（也可以使用分号来分隔，在使用分号时必须将列表放在引号中以防止 shell 将分号解释了其他用途。）如果定义了 LD_LIBRARY_PATH，那么动态链接器在查找标准库目录之前会先查找该环境变量列出的目录中的共享库。（稍后会介绍一个生产应用程序永远都不应该依赖于 LD_LIBRARY_PATH，但此刻通过这个变量可以方便地开始使用共享库了。）因此可以使用下面的命令来运行程序。

```
$ LD_LIBRARY_PATH=. ./prog
Called mod1-x1
Called mod2-x2
```

上面的命令中使用的 shell（bash、Korn 以及 Bourne）语法在执行 prog 的进程中创建了一个环境变量定义。这个定义告诉动态链接器在.，即当前工作目录中搜索共享库。

> 在 LD_LIBRARY_PATH 列表中的空目录（如 dirx::diry 中间的空目录）等价于.，即当前工作目录（但注意将 LD_LIBRARY_PATH 的值设置为空字符串并不能达到同样效果）。需要避免这种用法（SUSv3 同样不建议在 PATH 环境变量中使用这种方式）。

静态链接和动态链接比较

通常，术语链接用来表示使用链接器 ld 将一个或多个编译过的目标文件组合成一个可执行文件。有时候会使用术语静态链接从动态链接中将在运行时加载可执行文件所需的共享库这一步骤给区分出来。（静态链接有时候也被称为链接编辑，像 ld 这样的静态链接器有时候被称为链接编辑器。）每个程序——包括那些使用共享库的程序——都会经历一个静态链接的阶段。在运行时，使用共享库的程序会经历额外的动态链接阶段。

41.4.4 共享库 soname

到目前为止介绍的所有例子中，嵌入到可执行文件以及动态链接器在运行时搜索的名称是共享库文件的实际名称，这被称为库的真实名称（real name）。但可以——实际上经常这样做——使用别名来创建共享库，这种别名称为 soname（ELF 中的 DT_SONAME 标签）。

如果共享库拥有一个 soname，那么在静态链接阶段会将 soname 嵌入到可执行文件中，而不会使用真实名称，同时后面的动态链接器在运行时也会使用这个 soname 来搜索库。引入 soname 的目的是为了提供一层间接，使得可执行程序能够在运行时使用与链接时使用的库不同的（但兼容的）共享库。

在 41.6 节中将会介绍共享库的真实名称和 soname 的命名规则。下面通过一个简化的例子来说明这些原则。

使用 soname 的第一步是在创建共享库时指定 soname。

```
$ gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
$ gcc -g -shared -Wl,-soname,libbar.so -o libfoo.so mod1.o mod2.o mod3.o
```

–Wl、–soname 以及 libbar.so 选项是传给链接器的指令以将共享库 libfoo.so 的 soname 设置为 libbar.so。

如果要确定一个既有共享库的 soname，那么可以使用下面两个命令中的任意一个。

```
$ objdump -p libfoo.so | grep SONAME
  SONAME      libbar.so
$ readelf -d libfoo.so | grep SONAME
 0x0000000e (SONAME)      Library soname: [libbar.so]
```


更多资源请访问稀酷客(www.ckook.com)

在使用 soname 创建了一个共享库之后就可以照常创建可执行文件了。

```
$ gcc -g -Wall -o prog prog.c libfoo.so
```

但这次链接器检查到库 libfoo.so 包含了 soname libbar.so，于是将这个 soname 嵌入到了可执行文件中。

现在当运行这个程序时就会看到下面的输出。

```
$ LD_LIBRARY_PATH=. ./prog
prog: error in loading shared libraries: libbar.so: cannot open
shared object file: No such file or directory
```

这里的问题是动态链接器无法找到名为 libbar.so 共享库。当使用 soname 时还需要做一件事情：必须要创建一个符号链接将 soname 指向库的真实名称，并且必须要将这个符号链接放在动态链接器搜索的其中一个目录中。因此可以像下面这样运行这个程序。

```
$ ln -s libfoo.so libbar.so          Create soname symbolic link in current directory
$ LD_LIBRARY_PATH=. ./prog
Called mod1-x1
Called mod2-x2
```

图 41-1 给出了在使用一个内嵌的 soname，将程序与共享库链接起来，以及创建运行程序所需的 soname 符号链接时所涉及到的编译和链接事项。

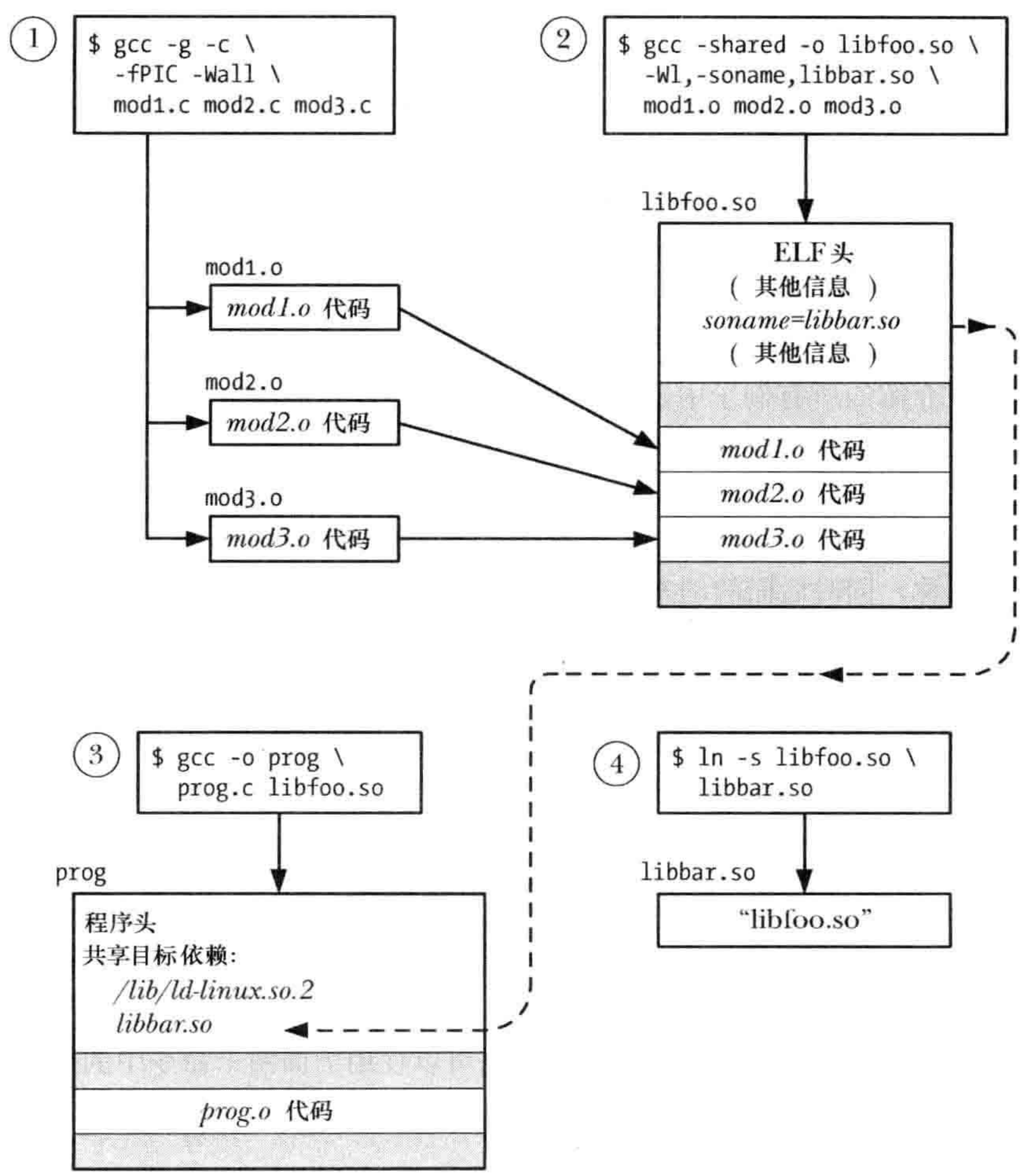


图 41-1：创建一个共享库并将一个程序与该共享库链接起来

图 41-2 给出了当图 41-1 中创建的程序被加载进内存以备执行时发生的事情。

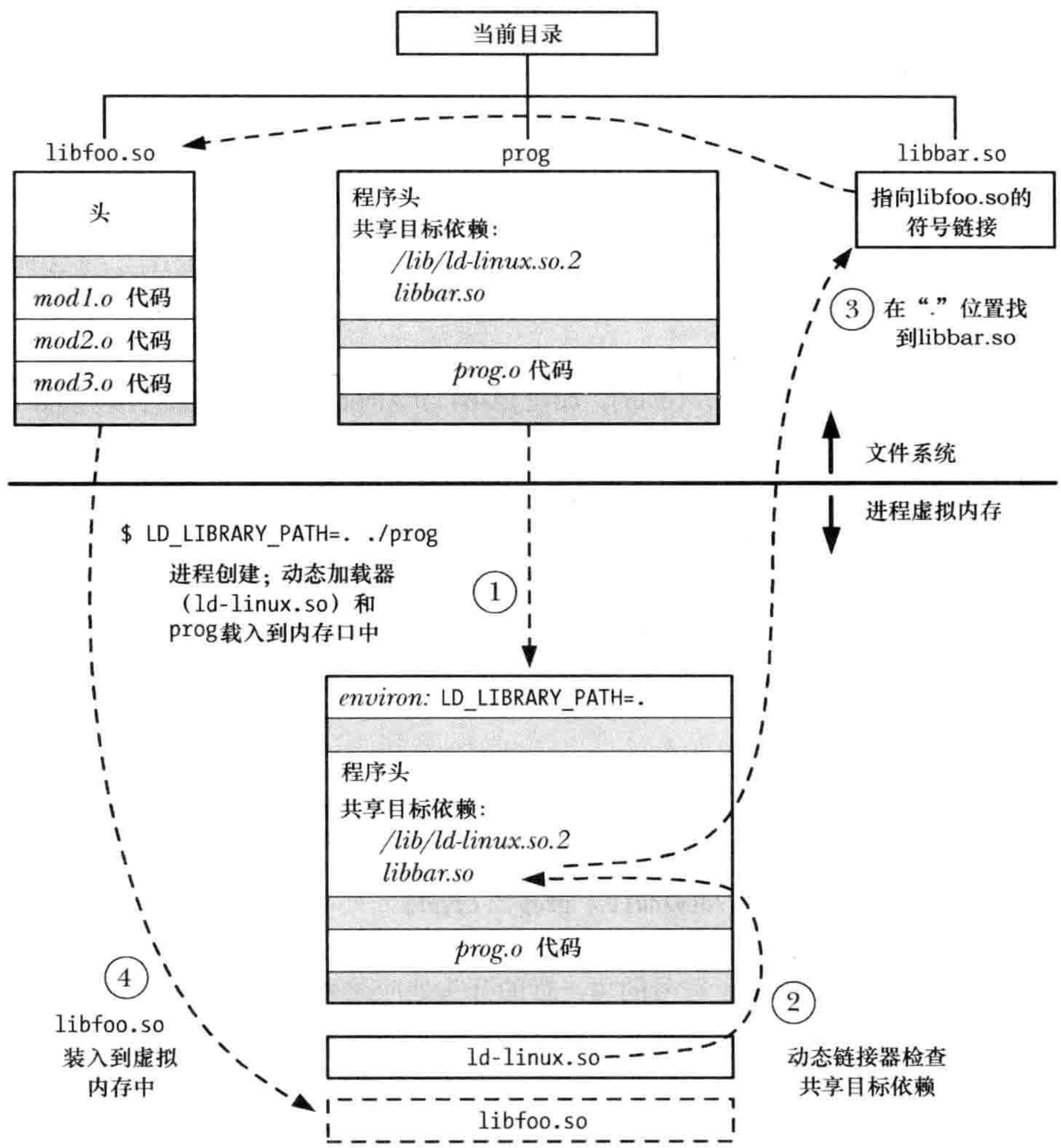


图 41-2：加载共享库的程序的执行

> 要找出一个进程当前使用的共享库则可以列出相应的 Linux 特有的/proc/PID/maps 文件中的内容（参见 48.5 节）。

41.5 使用共享库的有用工具

本节将简要介绍对分析共享库、可执行文件以及编译过的目标文件（.o）有用的一组工具。

ldd 命令

ldd(1)（列出动态依赖）命令显示了一个程序运行所需的共享库，如下所示。

更多资源请访问稀酷客(www.ckook.com)

```
$ ldd prog
        libdemo.so.1 => /usr/lib/libdemo.so.1 (0x40019000)
        libc.so.6 => /lib/tls/libc.so.6 (0x4017b000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

ldd 命令会解析出每个库引用（使用的搜索方式与动态链接器一样）并以下面的形式显示结果。

library-name => resolves-to-path

对于大多数 ELF 可执行文件来讲，ldd 至少会列出与 ld-linux.so.2、动态链接器以及标准 C 库 libc.so.6 相关的条目。

在一些架构上，C 库的名称是不同的。如在 IA-64 和 Alpha 上，这个库的名称是 libc.so.6.1。

objdump 和 readelf 命令

objdump 命令能够用来获取各类信息——包括反汇编的二进制机器码——从一个可执行文件、编译过的目标以及共享库中。它还能够用来显示这些文件中各个 ELF 节的头部信息，当这样使用 objdump 时它就类似于 readelf，readelf 能显示类似的信息，但显示格式不同。本章结尾处将会列出更多有关 objdump 和 readelf 的信息源。

nm 命令

nm 命令会列出目标库或可执行程序中定义的一组符号。这个命令的一种用途是找出哪些库定义了一个符号。如要找出哪个库定义了 crypt()函数则可以像下面这样做。

```
$ nm -A /usr/lib/lib*.so 2> /dev/null | grep ' crypt$'
/usr/lib/libcrypt.so:00007080 W crypt
```

nm 的–A 选项指定了在显示符号的每一行的开头处应该列出库的名称。这样做是有必要的，因为在默认情况下，nm 只列出库名一次，然后在后面会列出库中包含的所有符号，这对于像上面那样进行某种过滤的例子来讲是没有用处的。此外，这里还丢弃了标准错误输出以便隐藏与 nm 命令无法识别文件格式有关的错误消息。从上面的输出中可以看出，crypt()被定义在了 libcrypt 库中。

41.6 共享库版本和命名规则

下面考虑在共享库的版本化过程中需要做的事情。一般来讲，一个共享库相互连续的两个版本是相互兼容的，这意味着每个模块中的函数对外呈现出来的调用接口是一致的，并且函数的语义是等价的（即它们能取得同样的结果）。这种版本号不同但相互兼容的版本被称为共享库的次要版本。但有时候需要创建创建一个库的新主版本——即与上一个版本不兼容的版本。（在 41.8 节中将会更加明确地看到哪些方面会引起不兼容性。）同时，必须要确保使用老版本的库的程序仍然能够运行。

为了满足这些版本化的要求，共享库的真实名称和 soname 必须要使用一种标准的命名规范。

真实名称、soname 以及链接器名称

共享库的每个不兼容版本是通过一个唯一的主要版本标识符来区分的，这个主要版本标

更多资源请访问稀酷客(www.ckook.com)

识符是共享库的真实名称的一部分。根据惯例，主要版本标识符由一个数字构成，这个数字随着库的每个不兼容版本的发布而顺序递增。除了主要版本标识符之外，真实名称还包含一个次要版本标识符，它用来区分库的主要版本中兼容的次要版本。真实名称的格式规范为 libname.so.major-id.minor-id。

与主要版本标识符一样，次要版本标识符可以是任意字符串。但根据惯例，它要么是一个数字，要么是两个由点分隔的数字，其中第一个数字标识出了次要版本，第二个数字表示该次要版本中的补丁号或修订号。下面是一些共享库的真实名称。

```
libdemo.so.1.0.1
libdemo.so.1.0.2          Minor version, compatible with version 1.0.1
libdemo.so.2.0.0          New major version, incompatible with version 1.*
libreadline.so.5.0
```

共享库的 soname 包括相应的真实名称中的主要版本标识符，但不包含次要版本标识符。因此 soname 的形式为 libname.so.major-id。

通常，会将 soname 创建为包含真实名称的目录中的一个相对符号链接。下面是一些 soname 的例子以及它们可能通过符号链接指向的真实名称。

```
libdemo.so.1        -> libdemo.so.1.0.2
libdemo.so.2        -> libdemo.so.2.0.0
libreadline.so.5    -> libreadline.so.5.0
```

对于共享库的某个特定的主要版本来讲，可能存在几个库文件，这些库文件是通过不同的次要版本标识符来区分的。通常，每个库的主要版本的 soname 会指向在主要版本中最新的次要版本（如上面的 libdemo.so 例子所示）。这种配置使得在共享库的运行时操作期间版本化语义能够正确工作。由于静态链接阶段会将 soname 的副本（独立于次要版本）嵌入到可执行文件中并且 soname 符号链接后面可能会被修改指向一个更新的（次要）版本的共享库，因此可以确保可执行文件在运行时能够加载库的最新的次要版本。此外，由于一个库的不同的主要版本的 soname 不同，因此它们能够和平地共存并且被需要它们的程序访问。

除了真实名称和 soname 之外，通常还会为每个共享库定义第三个名称：链接器名称，将可执行文件与共享库链接起来时会用到这个名称。链接器名称是一个只包含库名同时不包含主要或次要版本标识符的符号链接，因此其形式为 libname.so。有了链接器名称之后就可以构建能够自动使用共享库的正确版本（即最新版本）的独立于版本的链接命令了。

一般来讲，链接器名称与它所引用的文件位于同一个目录中，它既可以链接到真实名称，也可以连接到库的最新主要版本的 soname。通常，最好使用指向 soname 的链接，因此对 soname 所做的变更会自动反应到链接器名称上。（在 41.7 节中会看到 ldconfig 程序将保持 soname 最新的任务自动化了，因此如果使用了刚才介绍的规范的话就是隐式地维护链接器名称。）

如果需要将一个程序与共享库的一个较老的主要版本链接起来，就不能使用链接器名称。相反，在链接命令中需要通过制定具体的真实名称或 soname 来标示出所需要的版本（主要版本）。

下面是一些链接器名称的例子。

```
libdemo.so          -> libdemo.so.2
libreadline.so      -> libreadline.so.5
```

表 41-1 对共享库的真实名称、soname 以及链接器名称进行了总结，图 41-3 描绘了这些名称之间的关系。

更多资源请访问稀酷客(www.ckook.com)

表 41-1：共享库名称总结

名　　称	格　　式	描　　述
真实名称	libname.so.maj.min	保存库代码的文件；每个库的 major-plus-minor 版本都存在一个真实名称
soname	libname.so.maj	库的每个主要版本都存在一个 soname；在链接时被嵌入到可执行文件中；在运行时用来找出指向相应的（最新的）真实名称的同名符号链接所引用的库
链接器名称	libname.so	指向真实名称或最新的（更常见的做法）soname 的符号链接；只存在一个实例；允许构建版本独立的链接命令

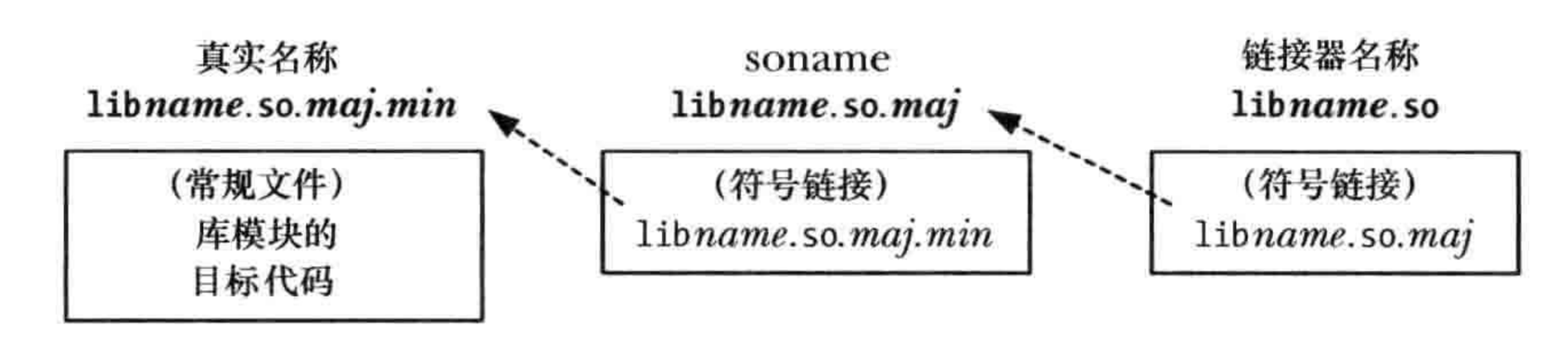


图 41-3：共享库名称的命名规范

使用标准规范创建一个共享库

根据上面介绍的相关知识，下面开始介绍如何遵循标准规范来构建一个演示库。首先需要创建目标文件。

```
$ gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
```

接着创建共享库，其真实名称为 libdemo.so.1.0.1，soname 为 libdemo.so.1。

```
$ gcc -g -shared -Wl,-soname,libdemo.so.1 -o libdemo.so.1.0.1 \
        mod1.o mod2.o mod3.o
```

接着为 soname 和链接器名称创建恰当的符号链接。

```
$ ln -s libdemo.so.1.0.1 libdemo.so.1
$ ln -s libdemo.so.1 libdemo.so
```

接着可以使用 ls 来验证配置（使用 awk 来选择感兴趣的字段）。

```
$ ls -l libdemo.so* | awk '{print $1, $9, $10, $11}'
lrwxrwxrwx libdemo.so -> libdemo.so.1
lrwxrwxrwx libdemo.so.1 -> libdemo.so.1.0.1
-rwxr-xr-x libdemo.so.1.0.1
```

接着可以使用链接器名称来构建可执行文件（注意链接命令不会用到版本号），并照常运行这个程序。

```
$ gcc -g -Wall -o prog prog.c -L. -ldemo
$ LD_LIBRARY_PATH=. ./prog
Called mod1-x1
Called mod2-x2
```

41.7　安装共享库

在本章到目前为止介绍的例子中都是将共享库创建在用户私有的目录中，然后使用 LD_LIBRARY_PATH 环境变量来确保动态链接器会搜到该目录。特权用户和非特权用户都可

更多资源请访问稀酷客(www.ckook.com)

以使用这种技术，但在生产应用程序中不应该采用这种技术。一般来讲，共享库及其关联的符号链接会被安装在其中一个标准库目录中，标准库目录包括：

- /usr/lib，它是大多数标准库安装的目录。
- /lib，应该将系统启动时用到的库安装在这个目录中（因为在系统启动时可能还没有挂载/usr/lib）。
- /usr/local/lib，应该将非标准或实验性的库安装在这个目录中（对于/usr/lib 是一个由多个系统共享的网络挂载但需要只在本机安装一个库的情况则可以将库放在这个目录中）。
- 其中一个在/etc/ld.so.conf（稍后介绍）中列出的目录。

在大多数情况下，将文件复制到这些目录中需要具备超级用户的权限。

安装完之后就必须要创建 soname 和链接器名称的符号链接了，通常它们是作为相对符号链接与库文件位于同一个目录中。因此要将本章的演示库安装在/usr/lib（只允许 root 进行更新）中则可以使用下面的命令。

```
$ su
Password:
# mv libdemo.so.1.0.1 /usr/lib
# cd /usr/lib
# ln -s libdemo.so.1.0.1 libdemo.so.1
# ln -s libdemo.so.1 libdemo.so
```

shell 会话中的最后两行创建了 soname 和链接器名称的符号链接。

ldconfig

ldconfig(8)解决了共享库的两个潜在问题。

- 共享库可以位于各种目录中，如果动态链接器需要通过搜索所有这些目录来找出一个库并加载这个库，那么整个过程将非常慢。
- 当安装了新版本的库或者删除了旧版本的库，那么 soname 符号链接就不是最新的。

ldconfig 程序通过执行两个任务来解决这些问题。

1. 它搜索一组标准的目录并创建或更新一个缓存文件/etc/ld.so.cache 使之包含在所有这些目录中的主要库版本（每个库的主要版本的最新的次要版本）列表。动态链接器在运行时解析库名称时会轮流使用这个缓存文件。为了构建这个缓存，ldconfig 会搜索在/etc/ld.so.conf 中指定的目录，然后搜索/lib 和 /usr/lib。/etc/ld.so.conf 文件由一个目录路径名（应该是绝对路径名）列表构成，其中路径名之间用换行、空格、制表符、逗号或冒号分隔。在一些发行版中，/usr/local/lib 目录也位于这个列表中。（如果不在这个列表中，那么就需要手工将其添加到列表中。）

命令 ldconfig –p 会显示/etc/ld.so.cache 的当前内容。

---

2. 它检查每个库的各个主要版本的最新次要版本（即具有最大的次要版本号的版本）以找出嵌入的 soname，然后在同一目录中为每个 soname 创建（或更新）相对符号链接。

为了能够正确执行这些动作，ldconfig 要求库的名称要根据前面介绍的规范来命名（即库的真实名称包含主要和次要标识符，它们随着库的版本的更新而恰当的增长）。

在默认情况下，ldconfig 会执行上面两个动作，但可以使用命令行选项来指定它执行其中一个动作：-N 选项会防止缓存的重建，-X 选项会阻止 soname 符号链接的创建。此外，-v (verbose) 选项会使得 ldconfig 输出描述其所执行的动作的信息。

更多资源请访问稀酷客(www.ckook.com)

每当安装了一个新的库，更新或删除了一个既有库，以及/etc/ld.so.conf 中的目录列表被修改之后，都应该运行 ldconfig。

下面是一个使用 ldconfig 的例子。假设需要安装一个库的两个不同的主要版本，那么需要做下面的事情。

```
$ su
Password:
# mv libdemo.so.1.0.1 libdemo.so.2.0.0 /usr/lib
# ldconfig -v | grep libdemo
        libdemo.so.1 -> libdemo.so.1.0.1 (changed)
        libdemo.so.2 -> libdemo.so.2.0.0 (changed)
```

上面对 ldconfig 的输出进行了过滤，这样读者就只会看到与名为 libdemo 的库相关的信息了。

接着列出在/usr/lib 目录中名为 libdemo 的文件来验证 soname 符号链接的设置。

```
# cd /usr/lib
# ls -l libdemo* | awk '{print $1, $$9, $10, $11}'
lrwxrwxrwx libdemo.so.1 -> libdemo.so.1.0.1
-rwxr-xr-x libdemo.so.1.0.1
lrwxrwxrwx libdemo.so.2 -> libdemo.so.2.0.0
-rwxr-xr-x libdemo.so.2.0.0
```

还需要为链接器名称创建符号链接，如下面的命令所示。

```
# ln -s libdemo.so.2 libdemo.so
```

如果安装了库的一个新的 2.x 次要版本，那么由于链接器名称指向了最新的 soname，因此 ldconfig 还能取得保持链接器名称最新的效果，如下面的例子所示。

```
# mv libdemo.so.2.0.1 /usr/lib
# ldconfig -v | grep libdemo
        libdemo.so.1 -> libdemo.so.1.0.1
        libdemo.so.2 -> libdemo.so.2.0.1 (changed)
```

如果创建和使用的是一个私有库（即没有安装在上述标准目录中的库），那么可以通过使用-n 选项让 ldconfig 创建 soname 符号链接。这个选项指定了 ldconfig 只处理在命令行中列出的目录中的库，而无需更新缓存文件。下面的例子使用了 ldconfig 来处理当前工作目录中的库。

```
$ gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
$ gcc -g -shared -Wl,-soname,libdemo.so.1 -o libdemo.so.1.0.1 \
          mod1.o mod2.o mod3.o
$ /sbin/ldconfig -nv .
.:
        libdemo.so.1 -> libdemo.so.1.0.1
$ ls -l libdemo.so* | awk '{print $1, $9, $10, $11}'
lrwxrwxrwx libdemo.so.1 -> libdemo.so.1.0.1
-rwxr-xr-x libdemo.so.1.0.1
```

在上面的例子中，当运行 ldconfig 时指定了完全路径名，因为使用的是一个非特权账号，其 PATH 环境变量不包含/sbin 目录。

41.8 兼容与不兼容库比较

随着时间的流逝，可能需要修改共享库的代码。这种修改会导致产生一个新版本的库，这个新版本可以与之前的版本兼容，也可能与之前的版本不兼容。如果是兼容的话则意味着

更多资源请访问稀酷客(www.ckook.com)

只需要修改库的真实名称的次要版本标识符即可，如果是不兼容的话则意味着必须要定义一个库的新主要版本。

当满足下列条件时表示修改过的库与既有库版本兼容。

- 库中所有公共方法和变量的语义保持不变。换句话说，每个函数的参数列表不变并且对全局变量和返回参数产生的影响不变，同时返回同样的结果值。因此提升性能或修复 Bug（导致更加行为更加符合规定）的变更可以认为是兼容的变更。
- 没有删除库的公共 API 中的函数和变量，但向公共 API 中添加新函数和变量不会影响兼容性。
- 在每个函数中分配的结构以及每个函数返回的结构保持不变。类似的，由库导出的公共结构保持不变。这个规则的一个例外情况是在特定情况下，可能会向既有结构的结尾处添加新的字段，但当调用程序在分配这个结构类型的数组时会产生问题。有时候，库的设计人员会通过将导出结构的大小定义为比库的首个发行版所需的大小大来解决这个问题，即增加一些填充字段以备将来之需。

如果所有这些条件都得到了满足，那么在更新新库名时就只需要调整既有名称中的次要版本号了，否则就需要创建库的一个新主要版本。

41.9 升级共享库

共享库的优点之一是当一个运行着的程序正在使用共享库的一个既有版本时也能够安装库的新主要版本或次要版本。在安装的过程中需要做的事情包括创建新的库版本、将其安装在恰当的目录中以及根据需要更新 soname 和链接器名称符号链接（或通常让 ldconfig 来完成这部分工作）。如要创建共享库/usr/lib/libdemo.1.0.1 的一个新次要版本，那么需要完成：

```
$ su
Password:
# gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
# gcc -g -shared -Wl,-soname,libdemo.so.1 -o libdemo.so.1.0.2 \
        mod1.o mod2.o mod3.o
# mv libdemo.so.1.0.2 /usr/lib
# ldconfig -v | grep libdemo
        libdemo.so.1 -> libdemo.so.1.0.2 (changed)
```

假设已经正确地配置了链接器名称（即指向库的 soname），那么就无需修改链接器名称了。

已经运行着的程序会继续使用共享库的上一个次要版本，只有当它们终止或重启之后才会使用共享库的新次要版本。

如果后面需要创建共享库的一个新主要版本（2.0.0），那么就需要完成：

```
# gcc -g -c -fPIC -Wall mod1.c mod2.c mod3.c
# gcc -g -shared -Wl,-soname,libdemo.so.2 -o libdemo.so.2.0.0 \
        mod1.o mod2.o mod3.o
# mv libdemo.so.2.0.0 /usr/lib
# ldconfig -v | grep libdemo
        libdemo.so.1 -> libdemo.so.1.0.2
        libdemo.so.2 -> libdemo.so.2.0.0 (changed)
# cd /usr/lib
# ln -sf libdemo.so.2 libdemo.so
```

从上面的输出可以看出，ldconfig 自动为新主要版本创建了一个 soname 符号链接，但从最后一条命令可以看出，必须要手工更新链接器名称的符号链接。

更多资源请访问稀酷客(www.ckook.com)

41.10 在目标文件中指定库搜索目录

到目前为止本章已经介绍了两种通知动态链接器共享库的位置的方式：使用 LD_LIBRARY_PATH 环境变量和将共享库安装到其中一个标准库目录中（/lib、/usr/lib 或在 /etc/ld.so.conf 中列出的其中一个目录）。

还存在第三种方式：在静态编辑阶段可以在可执行文件中插入一个在运行时搜索共享库的目录列表。这种方式对于库位于一个固定的但不属于动态链接器搜索的标准位置的位置中时是非常有用的。要实现这种方式需要在创建可执行文件时使用-rpath 链接器选项。

```
$ gcc -g -Wall -Wl,-rpath,/home/mtk/pdir -o prog prog.c libdemo.so
```

上面的命令将字符串/home/mtk/pdir 复制到了可执行文件 prog 的运行时库路径（rpath）列表中，因此当运行这个程序时，动态链接器在解析共享库引用时还会搜索这个目录。

如果有必要的话，可以多次指定–rpath 选项；所有这些列出的目录会被连接成一个放到可执行文件中的有序 rpath 列表。或者，在一个 rpath 选项中可以指定多个由分号分割开来的目录列表。在运行时，动态链接器会按照在–rpath 选项中指定的目录顺序来搜索目录。

> -rpath 选项的一个替代方案是 LD_RUN_PATH 环境变量。可以将一个由分号分隔开来的目录的字符串赋给该变量，当构建可执行文件时可以将这个变量作为 rpath 列表来使用。只有当构建可执行文件时不指定-rpath 选项时才会使用 LD_RUN_PATH 变量。

在构建共享库时使用–rpath 链接器选项

在构建共享库时–rpath 选项也是有用的。假设有一个依赖于另一个共享库 libx2.so 的共享库 libx1.so，如图 41-4 所示。另外再假设这些库分别位于非标准目录 d1 和 d2 中。下面介绍构建这些库以及使用它们的程序所需完成的步骤。

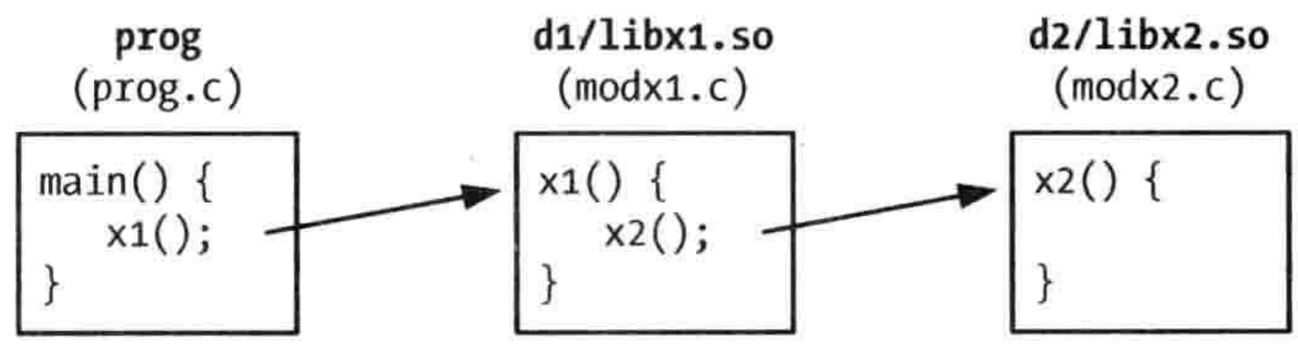


图 41-4：依赖于另一个共享库的共享库

首先在 pdir/d2 目录中构建 libx2.so。（为了使这个例子简单一点，这里省略了库的版本号和 soname。）

```
$ cd /home/mtk/pdir/d2
$ gcc -g -c -fPIC -Wall modx2.c
$ gcc -g -shared -o libx2.so modx2.o
```

接着在 pdir/d1 目录中构建 libx1.so。由于 libx1.so 依赖于 libx2.so，并且 libx2.so 位于一个非标准目录中，因此在指定 libx2.so 的运行时位置时需要使用–rpath 链接器选项。这个选项的取值与库的链接时位置（由-L 选项指定）可以不同，尽管在这个例子中这两个位置是相同的。

```
$ cd /home/mtk/pdir/d1
$ gcc -g -c -Wall -fPIC modx1.c
$ gcc -g -shared -o libx1.so modx1.o -Wl,-rpath,/home/mtk/pdir/d2 \
            -L/home/mtk/pdir/d2 -lx2
```

最后在 pdir 目录中构建主程序。由于主程序使用了 libx1.so 并且这个库位于一个非标准目录中，因此还需要使用–rpath 链接器选项。

```
$ cd /home/mtk/pdir
$ gcc -g -Wall -o prog prog.c -Wl,-rpath,/home/mtk/pdir/d1 \
          -L/home/mtk/pdir/d1 -lx1
```

注意在链接主程序时无需指定 libx2.so。由于链接器能够分析 libx1.so 中的 rpath 列表，因此它能够找到 libx2.so，同时在静态链接阶段解析出所有的符号。

使用下面的命令能够检查 prog 和 libx1.so 以便查看它们的 rpath 列表的内容。

```
$ objdump -p prog | grep PATH
  RPATH        /home/mtk/pdir/d1          libx1.so will be sought here at run time
$ objdump -p d1/libx1.so | grep PATH
  RPATH        /home/mtk/pdir/d2          libx2.so will be sought here at run time
```

还可以通过查找 readelf--dynamic（或等价的 readelf–d）命令的输出来查看 rpath 列表。

使用 ldd 命令能够列出 prog 的完整的动态依赖列表。

```
$ ldd prog
        libx1.so => /home/mtk/pdir/d1/libx1.so (0x40017000)
        libc.so.6 => /lib/tls/libc.so.6 (0x40024000)
        libx2.so => /home/mtk/pdir/d2/libx2.so (0x4014c000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

ELF DT_RPATH 和 DT_RUNPATH 条目

在第一版 ELF 规范中，只有一种 rpath 列表能够被嵌入到可执行文件或共享库中，它对应于 ELF 文件中的 DT_RPATH 标签。后续的 ELF 规范舍弃了 DT_RPATH，同时引入了一种新标签 DT_RUNPATH 来表示 rpath 列表。这两种 rpath 列表之间的差别在于当动态链接器在运行时搜索共享库时它们相对于 LD_LIBRARY_PATH 环境变量的优先级：DT_RPATH 的优先级更高，而 DT_RUNPATH 的优先级则更低（参见 41.11 节）。

在默认情况下，链接器会将 rpath 列表创建为 DT_RPATH 标签。为了让链接器将 rpath 列表创建为 DT_RUNPATH 条目必须要额外使用--enable–new–dtags(启用新动态标签)链接器选项。如果使用这个选项重建程序并且使用 objdump 查看获得的可执行文件，那么将会看到下面这样的输出。

```
$ gcc -g -Wall -o prog prog.c -Wl,--enable-new-dtags \
        -Wl,-rpath,/home/mtk/pdir/d1 -L/home/mtk/pdir/d1 -lx1
$ objdump -p prog | grep PATH
  RPATH        /home/mtk/pdir/d1
  RUNPATH      /home/mtk/pdir/d1
```

从上面可以看出，可执行文件包含了 DT_RPATH 和 DT_RUNPATH 标签。链接器采用这种方式复写了 rpath 列表是为了让不理解 DT_RUNPATH 标签的老式动态链接器能够正常工作。（glibc 2.2 增加了对 DT_RUNPATH 的支持）。理解 DT_RUNPATH 标签的链接器会忽略 DT_RPATH 标签（参见 41.11 节）。

在 rpath 中使用$ORIGIN

假设需要发布一个应用程序，这个应用程序使用了自身的共享库，但同时不希望强制要求用户将这些库安装在其中一个标准目录中，相反，需要允许用户将应用程序解压到任意异目录中，然后能够立即运行这个应用程序。这里存在的问题是应用程序无法确定存放共享库

更多资源请访问稀酷客(www.ckook.com)

的位置，除非要求用户设置 LD_LIBRARY_PATH 或者要求用户运行某种能够标识出所需的目录的安装脚本，但这两种方法都不是令人满意的方法。

为解决这个问题，在构建链接器的时候增加了对 rpath 规范中特殊字符串$ORIGIN（或等价的${ORIGIN}）的支持。动态链接器将这个字符串解释成“包含应用程序的目录”。这意味着可以使用下面的命令来构建应用程序。

```
$ gcc -Wl,-rpath,'$ORIGIN'/lib ...
```

上面的命令假设在运行时应用程序的共享库位于包含应用程序的可执行文件的目录的子目录 lib 中。这样就能向用户提供一个简单的包含应用程序及相关的库的安装包，同时允许用户将这个包安装在任意位置并运行这个应用程序了（即所谓的“turn-key 应用程序”）。

41.11 在运行时找出共享库

在解析库依赖时，动态链接器首先会检查各个依赖字符串以确定它是否包含斜线（/），因为在链接可执行文件时如果指定了一个显式的库路径名的话就会发生这种情况。如果找到了一个斜线，那么依赖字符串就会被解释成一个路径名（绝对路径名或相对路径名），并且会使用该路径名加载库。否则动态链接器会使用下面的规则来搜索共享库。

1. 如果可执行文件的 DT_RPATH 运行时库路径列表（rpath）中包含目录并且不包含 DT_RUNPATH 列表，那么就搜索这些目录（按照链接程序时指定的目录顺序）。
2. 如果定义了 LD_LIBRARY_PATH 环境变量，那么就会轮流搜索该变量值中以冒号分隔的各个目录。如果可执行文件是一个 set-user-ID 或 set-group-ID 程序，那么就会忽略 LD_LIBRARY_PATH 变量。这项安全措施是为了防止用户欺骗动态链接器让其加载一个与可执行文件所需的库的名称一样的私有库。
3. 如果可执行文件 DT_RUNPATH 运行时库路径列表中包含目录，那么就会搜索这些目录（按照链接程序时指定的目录顺序）。
4. 检查/etc/ld.so.cache 文件以确认它是否包含了与库相关的条目。
5. 搜索/lib 和/usr/lib 目录（按照这个顺序）。

41.12 运行时符号解析

假设在多个地方定义了一个全局符号（即函数或变量），如在一个可执行文件和一个共享库中或在多个共享库中。那么如何解析指向这个符号的引用呢？

假设现在有一个主程序和一个共享库，它们两个都定义了一个全局函数 xyz()，并且共享库中的另一个函数调用了 xyz()，如图 41-5 所示。

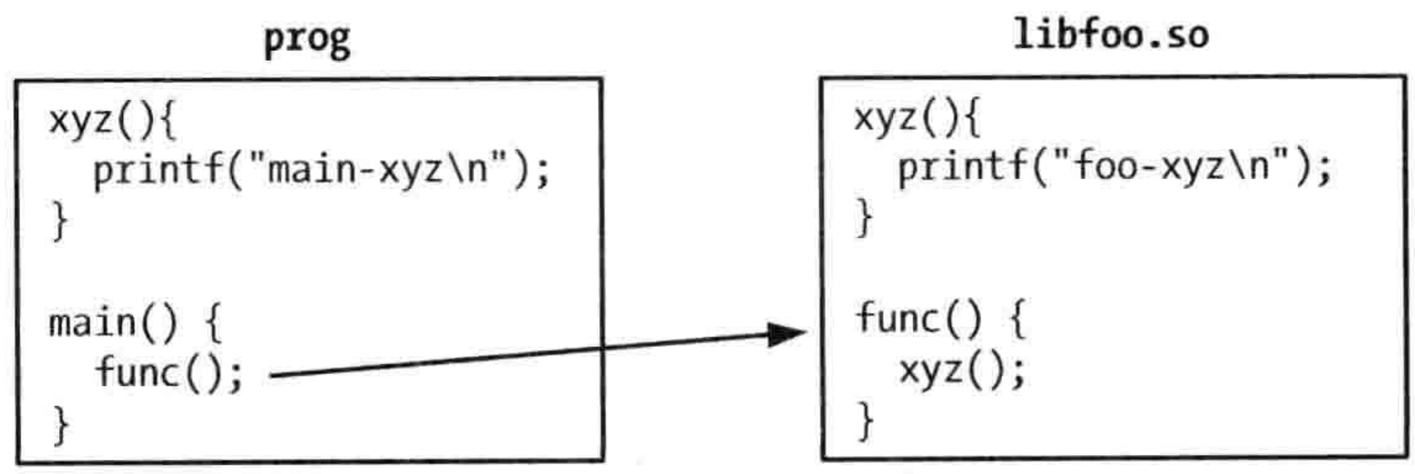


图 41-5：解析全局符号引用

在构建共享库和可执行程序并运行这个程序之后能够看到下面的输出。

```
$ gcc -g -c -fPIC -Wall -c foo.c
$ gcc -g -shared -o libfoo.so foo.o
$ gcc -g -o prog prog.c libfoo.so
$ LD_LIBRARY_PATH=. ./prog
main-xyz
```

从上面输出的最后一行可以看出，主程序中的 xyz()定义覆盖（优先）了共享库中的定义。

尽管这种处理方式在一开始看起来有些令人惊讶，但这样做是有历史原因的。第一个共享库实现在设计时的目标是使符号解析的默认语义与那些和同一库等价的静态库进行链接的应用程序中的符号解析的语义完成一致。这意味着下面的语义是正确的。

- 主程序中全局符号的定义覆盖库中相应的定义。
- 如果一个全局符号在多个库中进行了定义，那么对该符号的引用会被绑定到在扫描库时找到的第一个定义，其中扫描顺序是按照这些库在静态链接命令行中列出时从左至右的顺序。

虽然这些语义使得从静态库到共享库的转变变得相对简单了，但这种做法会导致一些问题。其中最大的问题是这些语义在使用共享库实现一个自包含的子系统时会与共享库模型产生矛盾。在默认情况下，共享库无法确保一个指向其自身的某个全局符号的引用会真正被绑定到该符号在库中的定义上，从而导致当该共享库被集成到一个更大的系统中时共享库的属性可能会发生改变。这会导致应用程序出现令人意料之外的行为，同时也使得分治调试的执行变得更加困难（即尝试使用更少或不同的共享库来重现问题）。

在上面的例子中，如果想要确保在共享库中对 xyz()的调用确实调用了库中定义的相应函数，那么在构建共享库的时候就需要使用–Bsymbolic 链接器选项。

```
$ gcc -g -c -fPIC -Wall -c foo.c
$ gcc -g -shared -Wl,-Bsymbolic -o libfoo.so foo.o
$ gcc -g -o prog prog.c libfoo.so
$ LD_LIBRARY_PATH=. ./prog
foo-xyz
```

–Bsymbolic 链接器选项指定了共享库中对全局符号的引用应该优先被绑定到库中的相应定义上（如果存在的话）。（注意不管是否使用了这个选项，在主程序中调用 xyz()总是会调用主程序中定义的 xyz()。）

41.13 使用静态库取代共享库

虽然在大多数情况下都应该使用共享库，但在某些场景中静态库则更加适合。特别地，静态链接的应用程序包含了它在运行时所需的全局代码这一事实是非常有利的。如当用户不希望或者无法在运行程序的系统上安装共享库或者程序在另一个无法使用共享库的环境中运行时（如可能是一个 chroot 监狱（jail）），静态链接就派上用场了。此外，即使是一个兼容的共享库升级也可能会在无意中引入一个 Bug，从而导致应用程序无法正常工作。通过静态链接应用程序就能确保系统上共享库的变动不会影响到它并且它已经拥有了运行所需的全局代码（付出的代价就是程序更大了，从而会需要更多的磁盘空间和内存）。

在默认情况下，当链接器能够选择名称一样的共享库和静态库时（如在链接时使用

更多资源请访问稀酷客(www.ckook.com)

–Lsomedir –ldemo 并且 libdemo.so 和 libdemo.a 都存在）会优先使用共享库。要强制使用库的静态版本则可以完成下列之一。

- 在 gcc 命令行中指定静态库的路径名（包括.a 扩展）。
- 在 gcc 命令行中指定-static 选项。
- 使用–Wl,–Bstatic 和–Wl,–Bdynamic gcc 选项来显式地指定链接器选择共享库还是静态库。在 gcc 命令行中可以使用-l 选项来混合这些选项。链接器会按照选项被指定时的顺序来处理这些选项。

41.14 总结

目标库是一组编译过的目标模块的聚合，它可以用来与程序进行链接。与其他 UNIX 实现一样，Linux 提供了两种目标库：一种是静态库，在早期的 UNIX 系统中只存在这种库，还有一种是更加现代的共享库。

由于与静态库相比，共享库存在很多优势，因此在当代 UNIX 系统上共享库用得最多。共享库的优势主要源自这样一个事实，即当一个程序与库进行链接时，程序所需的目标模块的副本不会被包含进结果可执行文件中。相反，（静态）链接器将会在可执行文件中添加与程序在运行时所需的共享库相关的信息。当文件被执行时，动态链接器会使用这些信息来加载所需的共享库。在运行时，所有使用同一共享库的程序共享该库在内存中的单个副本。由于共享库不会被复制到可执行文件中，并且在运行时所有程序都使用共享库在内存中的单个副本，因此共享库能够降低系统所需的磁盘空间和内存。

共享库 soname 为在运行时接续共享库引用提供了一层间接。如果一个共享库拥有一个 soname，那么在由静态链接器产生的可执行文件中将会记录这个 soname，而不是库的真实名称。根据共享库命名规范，其真实名称的形式为 libname.so.major-id.minor-id，其 soname 的形式为 libname.so.major-id。这种规范使得程序能够自动使用共享库的最新次要版本（无需重新链接程序），同时也允许创建库的新的不兼容的主要版本。

为了在运行时能够找到共享库，动态链接器遵循了一组标准的搜索规则，其中包括搜索一组大多数共享库安装的目录（如/lib 和/usr/lib）。

更多信息

在 ar(1)、gcc(1)、ld(1)、ldconfig(8)、ld.so(8)、dlopen(3)和 objdump(1)手册以及 ld 和 readelf 的 info 文档中可以找到与静态库和共享库相关的各种信息。[Drepper, 2004 (b)]介绍了很多在 Linux 上编写共享库的细节信息。在 David Wheeler 撰写的“Program Library HOWTO”中可以找出更多有用的信息，该书的在线版本位于 PDP 网站 http://www.tldp.org/上。GNU 共享库模型与 Solaris 上的实现存在很多相似之处，因此阅读一下 Sun 的“Linker and Libraries Guide”（在 http://docs.sun.com/上可以找到）以获取更多信息和例子是有必要的。[Levine, 2000]对静态和动态链接器的操作进行了介绍。

在在线站点 http://www.gnu.org/software/libtool 和[Vaughan et al., 2000]中可以找到有关 GNU Libtool 的信息，它是一个用来向程序员隐藏构建共享库时碰到的特定于实现的细节的工具。

Tools Interface Standards 委员会撰写的“Executable and Linking Format”文档提供了 ELF 的细节，在 http://refspecs.freestandards.org/elf/elf.pdf 上能够找到这篇文档。[Lu, 1995]也提供了

更多资源请访问稀酷客(www.ckook.com)

很多与 ELF 有关的有用细节。

41.15 习题

41-1. 在使用和不使用–static 选项的情况下编译一个程序来看看动态地与 C 库进行链接的程序与静态地与 C 库进行链接的程序在大小方面的差别。

更多资源请访问稀酷客(www.ckook.com)

第42章

共享库高级特性

上一章介绍了共享库的基础知识，本章将介绍共享库的几个高级特性，如下所示：

- 动态地加载共享库；
- 控制共享库定义的符号的可见性；
- 使用链接器脚本创建版本化的符号；
- 使用初始化和终止函数在加载和卸载库时自动地执行代码；
- 共享库预加载；
- 使用 LD_DEBUG 来监控动态链接器的操作。

42.1 动态加载库

当一个可执行文件开始运行之后，动态链接器会加载程序的动态依赖列表中的所有共享库，但有些时候延迟加载库是比较有用的，如只在需要的时候再加载一个插件。动态链接器的这项功能是通过一组 API 来实现的。这组 API 通常被称为 dlopen API，它源自 Solaris，现在其中大部分内容都在 SUSv3 中进行了规定。

dlopen API 使得程序能够在运行时打开一个共享库，根据名字在库中搜索一个函数，然后调用这个函数。在运行时采用这种方式加载的共享库通常被称为动态加载的库，它的创建方式与其他共享库的创建方式完全一样。

核心 dlopen API 由下列函数（所有这些函数都在 SUSv3 进行了规定）构成。

- dlopen()函数打开一个共享库，返回一个供后续调用使用的句柄。
- dlsym()函数在库中搜索一个符号（一个包含函数或变量的字符串）并返回其地址。
- dlclose()函数关闭之前由 dlopen()打开的库。
- dlerror()函数返回一个错误消息字符串，在调用上述函数中的某个函数发生错误时可以使用这个函数来获取错误消息。

glibc 实现还包含了一组相关的函数，其中一些将会在后面予以介绍。

要在 Linux 上使用 dlopen API 构建程序必须要指定–ldl 选项以便与 libdl 库链接起来。

更多资源请访问稀酷客(www.ckook.com)

42.1.1 打开共享库：dlopen()

dlopen()函数将名为 libfilename 的共享库加载进调用进程的虚拟地址空间并增加该库的打开引用计数。

```
#include <dlfcn.h>

void *dlopen(const char *libfilename, int flags);
                                Returns library handle on success, or NULL on error
```

如果 libfilename 包含了一个斜线（/），那么 dlopen()会将其解释成一个绝对或相对路径名，否则动态链接器会使用第 41.11 节中介绍的规则来搜索共享库。

dlopen()在成功时会返回一个句柄，在后续对 dlopen API 中的函数的调用可以使用该句柄来引用这个库。如果发生了错误（如无法找到库），那么 dlopen()会返回 NULL。

如果 libfilename 指定的共享库依赖于其他共享库，那么 dlopen()会自动加载那些库。如果有必要的话，这一过程会递归进行。这种被加载进来的库被称为这个库的依赖树。

在同一个库文件中可以多次调用 dlopen()，但将库加载进内存的操作只会发生一次（第一次调用），所有的调用都返回同样的句柄值。但 dlopen API 会为每个库句柄维护一个引用计数，每次调用 dlopen()时都会增加引用计数，每次调用 dlclose()都会减小引用计数，只有当计数为 0 时 dlclose()才会从内存中删除这个库。

flags 参数是一个位掩码，它的取值是 RTLD_LAZY 和 RTLD_NOW 中的一个，这两个值的含义分别如下。

RTLD_LAZY

只有当代码被执行的时候才解析库中未定义的函数符号。如果需要某个特定符号的代码没有被执行到，那么永远都不会解析该符号。延迟解析只适用于函数引用，对变量的引用会被立即解析。指定 RTLD_LAZY 标记能够提供与在加载可执行文件的动态依赖列表中的共享库时动态链接器的常规操作对应的行为。

RTLD_NOW

在 dlopen()结束之前立即加载库中所有的未定义符号，不管是否需要用到这些符号，这种做法的结果是打开库变得更慢了，但能够立即检测到任何潜在的未定义函数符号错误，而不是在后面某个时刻才检测到这种错误。在调试应用程序时这种做法是比较有用的，因为它能够确保应用程序在碰到未解析的符号时立即发生错误，而不是在执行了很长一段时间之后才发生错误。

通过将环境变量 LD_BIND_NOW 设置为一个非空字符串能够强制动态链接器在加载可执行文件的动态依赖列表中的共享库时立即解析所有符号（即类似于 RTLD_NOW）。这个环境变量在 glibc 2.1.1 以及后续的版本中是有效的。设置 LD_BIND_NOW 会覆盖 dlopen() RTLD_LAZY 标记的效果。

---

flags 也可以取其他的值，SUSv3 规定了下列几种标记。

RTLD_GLOBAL

这个库及其依赖树中的符号在解析由这个进程加载的其他库中的引用和通过 dlsym()查找

更多资源请访问稀酷客(www.ckook.com)

时可用。

RTLD_LOCAL

与 RTLD_GLOBAL 相反，如果不指定任何常量，那么就取这个默认值。它规定在解析后续加载的库中的引用时这个库及其依赖树中的符号不可用。

在不指定 RTLD_GLOBAL 或 RTLD_LOCAL 时，SUSv3 并没有规定一个默认值。大多数 UNIX 实现与 Linux 一样，将 RTLD_LOCAL 作为默认值，但一些实现将 RTLD_GLOBAL 作为默认值。

Linux 还支持几个并没有在 SUSv3 中进行规定的标记，如下所示。

RTLD_NODELETE（自 glibc 2.2 起）

在 dlclose()调用中不要卸载库，即使其引用计数已经变成 0 了。这意味着在后面重新通过 dlopen()加载库时不会重新初始化库中的静态变量。（对于由动态链接器自动加载的库来讲，在创建库时通过指定 gcc –Wl,–znodelete 选项能够取得类似的效果。）

RTLD_NOLOAD（自 glibc 2.2 起）

不加载库。这个标记有两个目的。第一，可以使用这个标记来检查某个特定的库是否已经被加载到了进程的地址空间中。如果已经加载了，那么 dlopen()会返回库的句柄，如果没有加载，那么 dlopen()会返回 NULL。第二，可以使用这个标记来“提升”已加载的库的标记。如在对之前使用 RTLD_LOCAL 打开的库调用 dlopen()时可以在 flags 参数中指定 RTLD_NOLOAD | RTLD_GLOBAL。

RTLD_DEEPBIND（自 glibc 2.3.4）

在解析这个库中的符号引用时先搜索库中的定义，然后再搜索已加载的库中的定义。这个标记使得一个库能够实现自包含，即优先使用自己的符号定义，而不是在已加载的其他库中定义的同名全局符号。（这与在 41.12 节中介绍的–Bsymbolic 链接器选项具有类似的效果。）

RTLD_NODELETE 和 RTLD_NOLOAD 标记在 Solaris dlopen API 中也进行了实现，但提供这个两个标记的 UNIX 实现很少。RTLD_DEEPBIND 标记是 Linux 特有的。

当将 libfilename 指定为 NULL 时 dlopen()会返回主程序的句柄。（SUSv3 将这种句柄称为“全局符号对象”的句柄。）在后续对 dlsym()的调用中使用这个句柄会导致首先在主程序中搜索符号，然后在程序启动时加载的共享库中进行搜索，最后在所有使用了 RTLD_GLOBAL 标记的动态加载的库中进行搜索。

42.1.2 错误诊断：dlerror()

如果在 dlopen()调用或 dlopen API 的其他函数调用中得到了一个错误，那么可以使用 dlerror()来获取一个指向表明错误原因的字符串的指针。

```
#include <dlfcn.h>

const char *dlerror(void);
```

Returns pointer to error-diagnostic string, or NULL if no error has occurred since previous call to *dlerror()*

如果从上一次调用 dlerror()到现在没有发生错误，那么 dlerror()函数返回 NULL，读者在下一节中就会看到这种处理方式带来的好处了。

更多资源请访问稀酷客(www.ckook.com)

42.1.3 获取符号的地址：dlsym()

dlsym()函数在 handle 指向的库以及该库的依赖树中的库中搜索名为 symbol 的符号（函数或变量）。

```
#include <dlfcn.h>

void *dlsym(void *handle, char *symbol);
```

Returns address of *symbol*, or NULL if *symbol* is not found

如果找到了 symbol，那么 dlsym()会返回其地址，否则就返回 NULL。handle 参数通常是上一个 dlopen()调用返回的库句柄，或者它也可以是下面介绍的其中一个所谓的伪句柄。

dlvsym(handle, symbol, version)与 dlsym()类似，但它能够用来在符号版本化的库中搜索版本与在字符串 version 中指定的版本匹配的符号定义。（第 42.3.2 节将会介绍符号版本化。）要从<dlfcn.h>中获取这个函数的声明必须要定义_GNU_SOURCE 特性测试宏。

dlsym()返回的符号值可能会是 NULL，这一点与“找不到符号”的返回是无法区分的。为了弄清楚具体是哪种情况就必须要先调用 dlerror()（确保之前的错误字符串已经被清除了），如果在调用 dlsym()之后 dlerror()返回了一个非 NULL 值，那么就可以得出发生错误的结论了。

如果 symbol 是一个变量的名称，那么可以将 dlsym()的返回值赋给一个合适的指针类型，并通过反引用该指针来得到变量的值。

```
int *ip;

ip = (int *) dlsym(symbol, "myvar");
if (ip != NULL)
    printf("Value is %d\n", *ip);
```

如果 symbol 是一个函数的名称，那么可以使用 dlsym()返回的指针来调用该函数。可以将 dlsym()返回的值存储到一个类型合适的指针中，如下所示。

```
int (*funcp)(int);              /* Pointer to a function taking an integer
                                   argument and returning an integer */
```

但不能简单地将 dlsym()的结果赋给此类指针，如下面的例子所示。

```
funcp = dlsym(handle, symbol);
```

其原因是 C99 标准禁止函数指针和 void *之间的赋值操作。这个问题的解决方案是使用下面这样的（稍微有些笨拙）类型转换。

```
*(void **) (&funcp) = dlsym(handle, symbol);
```

通过 dlsym()得到了指向函数的指针之后就能够通过常规的 C 语法反引用函数指针来调用这个函数了。

```
res = (*funcp)(somearg);
```

读者在将 dlsym()的返回值进行赋值时可能会使用下面这段看起来与上述代码等价的代码来取代上面的*(void **)语法。

```
(void *) funcp = dlsym(handle, symbol);
```

但 gcc –pedantic 在碰到上面这段代码时会发出“ANSI C forbids the use of cast expressions as lvalues.”的警告信息。而使用*(void **)语言就不会出现这个警告信息，因为是在向赋值语

更多资源请访问稀酷客(www.ckook.com)

句中的左值指向的地址赋值。

在很多 UNIX 实现中可以使用下面这样的类型转换类消除 C 编译器的警告。

```
funcp = (int (*) (int)) dlsym(handle, symbol);
```

但 SUSv3 Technical Corrigendum Number 1 中 dlsym()的规范指出 C99 标准仍然要求编译器对此类转换生成警告信息并列举了上面的*(void **)语法。

> SUSv3 TC1 指出由于需要用到*(void **)语法，因此标准的后续版本可能会定义一个与 dlsym()类似的 API 来处理数据和函数指针。但 SUSv4 在这一点上没有发生任何变化。

在 dlsym()中使用库伪句柄

dlsym()函数中的 handle 参数除了能够取由 dlopen()调用返回的句柄值之外，还能够取下列伪句柄值。

RTLD_DEFAULT

从主程序中开始查找 symbol，接着按序在所有已加载的共享库中查找，包括那些通过使用了 RTLD_GLOBAL 标记的 dlopen()调用动态加载的库，这个标记对应于动态链接器所采用的默认搜索模型。

RTLD_NEXT

在调用 dlsym()之后加载的共享库中搜索 symbol，这个标记适用于需要创建与在其他地方定义的函数同名的包装函数的情况。如，在主程序中可能会定义一个 malloc()（它可能完成内存分配的簿记工作），而这个函数在调用实际的 malloc()之前首先会通过调用 func = dlsym(RTLD_NEXT, “malloc”)来获取其地址。

SUSv3 并没有要求实现上述列出的伪句柄（甚至没有保留这两个值以供后续之用），并且所有 UNIX 实现也没有定义上述伪句柄。为了从<dlfcn.h>中获取这些常量的定义必须要定义 _GNU_SOURCE 特性测试宏。

示例程序

程序清单 42-1 演示了 dlopen API 的使用。这个程序接收两个命令行参数：需加载的共享库名称和需执行的库中函数的名称。下面的例子演示了这个程序的使用。

```
$ ./dynload ./libdemo.so.1 x1
Called mod1-x1
$ LD_LIBRARY_PATH=. ./dynload libdemo.so.1 x1
Called mod1-x1
```

在上述命令的第一个命令中，dlopen()注意到库路径包含了一个斜线，因此将其解释成一个相对路径名（表示一个位于当前工作目录中的库）。在第二个命令中指定了库搜索路径 LD_LIBRARY_PATH，动态链接器会根据正常的规则来解释这个搜索路径（同样表示在当前工作目录中查找库）。

程序清单 42-1：使用 dlopen API

shlibs/dynload.c

```
#include <dlfcn.h>
#include "tlpi_hdr.h"

int
```


更多资源请访问稀酷客(www.ckook.com)

```
main(int argc, char *argv[])
{
    void *libHandle;            /* Handle for shared library */
    void (*funcp)(void);        /* Pointer to function with no arguments */
    const char *err;

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

    /* Load the shared library and get a handle for later use */

    libHandle = dlopen(argv[1], RTLD_LAZY);
    if (libHandle == NULL)
        fatal("dlopen: %s", dlerror());

    /* Search library for symbol named in argv[2] */

    (void) dlerror();                           /* Clear dlerror() */
    *(void **) (&funcp) = dlsym(libHandle, argv[2]);
    err = dlerror();
    if (err != NULL)
        fatal("dlsym: %s", err);

    /* If the address returned by dlsym() is non-NULL, try calling it
       as a function that takes no arguments */

    if (funcp == NULL)
        printf("%s is NULL\n", argv[2]);
    else
        (*funcp)();

    dlclose(libHandle);                         /* Close the library */

    exit(EXIT_SUCCESS);
}
```

shlibs/dynload.c

42.1.4 关闭共享库：dlclose()

dlclose()函数关闭一个库。

```
#include <dlfcn.h>

int dlclose(void *handle);
```

Returns 0 on success, or −1 on error

dlclose()函数会减小 handle 所引用的库的打开引用的系统计数。如果这个引用计数变成了 0 并且其他库已经不需要用到该库中的符号了，那么就会卸载这个库。系统也会在这个库的依赖树中的库执行（递归地）同样的过程。当进程终止时会隐式地对所有库执行 dlclose()。

从 glibc 2.2.3 开始，共享库中的函数可以使用 atexit()（或 on_exit()）来设置一个在库被卸载时自动调用的函数。

42.1.5 获取与加载的符号相关的信息：dladdr()

dladdr()返回一个包含地址 addr（通常通过前面的 dlsym()调用获得）的相关信息的结构。

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

int dladdr(const void *addr, Dl_info *info);
```

Returns nonzero value if *addr* was found in a shared library, otherwise 0

info 参数是一个指向由调用者分配的结构的指针，其结构形式如下。

```
typedef struct {
    const char *dli_fname;          /* Pathname of shared library
                                       containing 'addr' */
    void       *dli_fbase;          /* Base address at which shared
                                       library is loaded */
    const char *dli_sname;          /* Name of nearest run-time symbol
                                       with an address <= 'addr' */
    void       *dli_saddr;          /* Actual value of the symbol
                                       returned in 'dli_sname' */
} Dl_info;
```

Dl_info 结构中的前两个字段指定了包含地址 addr 的共享库的路径名和运行时基地址。最后两个字段返回地址相关的信息。假设 addr 指向共享库中一个符号的确切地址，那么 dli_saddr 返回的值与传入的 addr 值一样。

SUSv3 并没有规定 dladdr()，所有 UNIX 实现也都没有提供这个函数。

42.1.6 在主程序中访问符号

假设使用 dlopen()动态加载了一个共享库，然后使用 dlsym()获取了共享库中 x()函数的地址，接着调用 x()。如果在 x()中调用了函数 y()，那么通常会在程序加载的其中一个共享库中搜索 y()。

有些时候需要让 x()调用主程序中的 y()实现（类似于回调机制）。为了达到这个目的就必须要使主程序中的符号（全局作用域）对动态链接器可用，即在链接程序时使用--export-dynamic 链接器选项。

```
$ gcc -Wl,--export-dynamic main.c      (plus further options and arguments)
```

或者可以编写下面这个等价的命令。

```
$ gcc -export-dynamic main.c
```

使用这些选项中的一个就能够允许动态加载的库访问主程序中的全局符号。

gcc -rdynamic 选项和 gcc -Wl、-E 选项的含义，以及-Wl、--export-dynamic 是一样的。

42.2 控制符号的可见性

设计良好的共享库应该只公开那些构成其声明的应用程序二进制接口（ABI）的符号（函数和变量），其原因如下。

- 如果共享库的设计人员不小心导出了未详细说明的接口，那么使用这个库的应用程序的作者可能会选择使用这些接口。这样在将来升级共享库时可能会带来兼容性问

更多资源请访问稀酷客(www.ckook.com)

题。库的开发人员认为可以修改或删除那些不属于文档中记录的 ABI 中的接口，而库的用户则希望继续使用名称与他们当前正在使用的接口名称一样的接口（同时语义保持不变）。

- 在运行时符号解析阶段，由共享库导出的所有符号可能会优先于其他共享库提供的相关定义（参见 41.12 节）。
- 导出非必需的符号会增加在运行时需加载的动态符号表的大小。

当库的设计人员确保只导出那些库的声明的 ABI 所需的符号就能使上述问题发生的可能性降到最低或避免上述问题的发生。下列技术可以用来控制符号的导出。

- 在 C 程序中可以使用 static 关键词使得一个符号私有于一个源代码模块，从而使得它无法被其他目标文件绑定。

> 除了使一个符号私有于源代码模块之外，static 关键词还能达到一个相反的效果。如果一个符号被标记为 static，那么在同一源文件中对该符号的所有引用会被绑定到该符号的定义上，其结果是这些引用在运行时不会被关联到其他共享库中的相应定义上（以 41.12 节中描述的方式）。static 关键词的这种效果类似于 41.12 节中介绍的链接器选项，但差别在于 static 关键词只影响单个源文件中的单个符号。

- GNU C 编译器 gcc 提供了一个特有的特性声明，它执行与 static 关键词类似的任务。

```
void
__attribute__ ((visibility("hidden")))
func(void) {
    /* Code */
}
```

static 关键词将一个符号的可见性限制在单个源代码文件中，而 hidden 特性使得一个符号对构成共享库的所有源代码文件都可见，但对库之外的文件不可见。

> 与 static 关键词一样，hidden 特性也能达到一个相反的效果，即防止在运行时发生符号插入。

- 版本脚本（参见 42.3 节）可以用来精确控制符号的可见性以及选择将一个引用绑定到符号的哪个版本。
- 当动态加载一个共享库时（参见 42.1.1 节），dlopen()接收的 RTLD_GLOBAL 标记可以用来指定这个库中定义的符号应该用于后续加载的库中的绑定操作，--export-dynamic 链接器选项（参见 42.1.6 节）可以用来使主程序的全局符号对动态加载的库可用。

更多有关符号可见性方面的细节信息可以参见[Drepper, 2004 (b)]。

42.3 链接器版本脚本

版本脚本是一个包含链接器 ld 执行的指令的文本文件。要使用版本脚本必须要指定 --version-script 链接器选项。

```
$ gcc -Wl,--version-script,myscriptfile.map ...
```

版本脚本的后缀通常（但不统一）是.map。

下面几节将介绍版本脚本的几个用途。

更多资源请访问稀酷客(www.ckook.com)

42.3.1 使用版本脚本控制符号的可见性

版本脚本的一个用途是控制那些可能会在无意中变成全局可见（即对与该库进行链接的应用程序可见）的符号的可见性。举一个简单的例子，假设需要从三个源文件 vis_comm.c、vis_f1.c 以及 vis_f2.c 中构建一个共享库，这三个源文件分别定义了函数 vis_comm()、vis_f1()以及 vis_f2()。vis_comm()函数由 vis_f1() 和 vis_f2()调用，但不想被与该库进行链接的应用程序直接使用。再假设使用常规的方式来构建共享库。

```
$ gcc -g -c -fPIC -Wall vis_comm.c vis_f1.c vis_f2.c
$ gcc -g -shared -o vis.so vis_comm.o vis_f1.o vis_f2.o
```

如果使用下面的 readelf 命令来列出该库导出动态符号，那么就会看到下面的输出。

```
$ readelf --syms --use-dynamic vis.so | grep vis_
   30  12: 00000790    59    FUNC GLOBAL DEFAULT  10 vis_f1
   25  13: 000007d0    73    FUNC GLOBAL DEFAULT  10 vis_f2
   27  16: 00000770    20    FUNC GLOBAL DEFAULT  10 vis_comm
```

这个共享库导出了三个符号：vis_comm()、vis_f1()以及 vis_f2()，但这里需要确保这个库只导出 vis_f1()和 vis_f2()符号。这种效果可以通过下面的版本脚本来实现。

```
$ cat vis.map
VER_1 {
    global:
        vis_f1;
        vis_f2;
    local:
        *;
};
```

标识符 VER_1 是一种版本标签。在 42.3.2 节对符号版本化的讨论中将会看到一个版本脚本可以包含多个版本节点，每个版本节点以括号（{}）组织起来并且在括号前面设置一个唯一的版本标签。如果使用版本脚本只是为了控制符号的可见性，那么版本标签是多余的，但老版本的 ld 仍然需要用到这个标签。ld 的现代版本允许省略版本标签，如果省略了版本标签的话就认为版本节点拥有一个匿名版本标签并且在这个脚本中不能存在其他版本节点。

在版本节点中，关键词 global 标记出了以分号分隔的对库之外的程序可见的符号列表的起始位置，关键词 local 标记出了以分号分隔的对库之外的程序隐藏的符号列表的起始位置。上面的星号(*)说明在符号规范中可以使用掩码模式，所使用的掩码字符与 shell 文件名匹配中使用的掩码字符是一样的——如*和?。（更多细节请参考 glob(7)手册。）在本例中，local 规范中的星号表示除了在 global 段中显式声明的符号之外的所有符号都对外隐藏。如果不这样声明，那么 vis_comm()仍然是可见的，因为在默认情况下 C 全局符号对共享库之外的程序是可见的。

接着可以像下面这样使用版本脚本来构建共享库。

```
$ gcc -g -c -fPIC -Wall vis_comm.c vis_f1.c vis_f2.c
$ gcc -g -shared -o vis.so vis_comm.o vis_f1.o vis_f2.o \
      -Wl,--version-script,vis.map
```

再次使用 readelf 可以看出 vis_comm()不再对外可见了。

```
$ readelf --syms --use-dynamic vis.so | grep vis_
   25   0: 00000730    73    FUNC GLOBAL DEFAULT  11 vis_f2
   29  16: 000006f0    59    FUNC GLOBAL DEFAULT  11 vis_f1
```


更多资源请访问稀酷客(www.ckook.com)

42.3.2 符号版本化

符号版本化允许一个共享库提供同一个函数的多个版本。每个程序会使用它与共享库进行（静态）链接时函数的当前版本。这种处理方式的结果是可以对共享库进行不兼容的改动而无需提升库的主要版本号。从极端的角度来讲，符号版本化可以取代传统的共享库主要和次要版本化模型。glibc 从 2.1 开始使用了这种符号版本化技术，因此 glibc 2.0 以及之前的所有版本都是通过单个主要库版本（libc.so.6）来支持的。

下面通过一个简单的例子来展示符号版本化的用途。首先使用一个版本脚本来创建共享库的第一个版本。

```
$ cat sv_lib_v1.c
#include <stdio.h>

void xyz(void) { printf("v1 xyz\n"); }
$ cat sv_v1.map
VER_1 {
        global: xyz;
        local:  *;      # Hide all other symbols
};
$ gcc -g -c -fPIC -Wall sv_lib_v1.c
$ gcc -g -shared -o libsv.so sv_lib_v1.o -Wl,--version-script,sv_v1.map
```

在版本脚本中，#开启了一段注释。

（为了使例子尽量简单点，这里没有使用显式的库 soname 和库主要版本号。）

在这个阶段，版本脚本 sv_v1.map 只用来控制共享库的符号的可见性，即只导出 xyz()，同时隐藏其他所有符号（在这个简短的例子中没有其他符号了）。接着创建一个程序 p1 来使用这个库。

```
$ cat sv_prog.c
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    void xyz(void);

    xyz();

    exit(EXIT_SUCCESS);
}
$ gcc -g -o p1 sv_prog.c libsv.so
```

运行这个程序之后就能看到预期的结果。

```
$ LD_LIBRARY_PATH=. ./p1
v1 xyz
```

现在假设需要修改库中 xyz()的定义，但同时仍然需要确保程序 p1 继续使用老版本的函数。为完成这个任务，必须要在库中定义两个版本的 xyz()。

```
$ cat sv_lib_v2.c
#include <stdio.h>

__asm__(".symver xyz_old,xyz@VER_1");
__asm__(".symver xyz_new,xyz@@VER_2");
```


更多资源请访问稀酷客(www.ckook.com)

```
void xyz_old(void) { printf("v1 xyz\n"); }

void xyz_new(void) { printf("v2 xyz\n"); }

void pqr(void) { printf("v2 pqr\n"); }
```

这里两个版本的 xyz()是通过函数 xyz_old()和 xyz_new()来实现的。xyz_old()函数对应于原来的 xyz()定义，pl 程序应该继续使用这个函数。xyz_new()函数提供了与库的新版本进行链接的程序所使用的 xyz()的定义。

修改过的版本脚本（稍后给出）中的两个.symver 汇编器指令将这两个函数绑定到了两个不同的版本标签上，下面将使用这个脚本来创建共享库的新版本。第一个指令指示与版本标签 VER_1 进行链接的应用程序（即程序 pl）所使用的 xyz()的实现是 xyz_old()，与版本标签 VER_2 进行链接的应用程序所使用的 xyz()的实现是 xyz_new()。

第二个.symver 指令使用@@（不是@）来指示当应用程序与这个共享库进行静态链接时应该使用的 xyz()的默认定义。一个符号的.symver 指令中应该只有一个指令使用@@标记。

下面是与修改过之后的库对应的版本脚本。

```
$ cat sv_v2.map
VER_1 {
        global: xyz;
        local:  *;      # Hide all other symbols
};

VER_2 {
        global: pqr;
} VER_1;
```

这个版本脚本提供了一个新版本标签 VER_2，它依赖于标签 VER_1。这种依赖关系是通过下面这行进行标记的。

```
} VER_1;
```

版本标记依赖表明了相邻两个库版本之间的关系。从语义上来讲，Linux 上的版本标签依赖的唯一效果是版本节点可以从它所依赖的版本节点中继承 global 和 local 规范。

依赖可以串联起来，这样就可以定义另一个依赖于 VER_2 的版本节点 VER_3 并以此类推地定义其他版本节点。

版本标签名本身是没有任何意义的，它们相互之间的关系是通过制定的版本依赖来确定的，因此这里选择名称 VER_1 和 VER_2 仅仅为了暗示它们之间的关系。为了便于维护，建议在版本标签名中包含包名和一个版本号。如 glibc 会使用名为 GLIBC_2.0 和 GLIBC_2.1 之类的版本标签名。

VER_2 版本标签还指定了将库中的 pqr()函数导出并绑定到 VER_2 版本标签。如果没有通过这种方式来声明 pqr()，那么 VER_2 版本标签从 VER_1 版本标签继承而来的 local 规范将会使 pqr()对外不可见。还需注意的是如果省略了 local 规范，那么库中的 xyz_old()和 xyz_new()符号也会被导出（这通常不是期望发生的事情）。

现在按照以往方式构建库的新版本。

```
$ gcc -g -c -fPIC -Wall sv_lib_v2.c
$ gcc -g -shared -o libsv.so sv_lib_v2.o -Wl,--version-script,sv_v2.map
```

现在创建一个新程序 p2，它使用了 xyz()的新定义，同时程序 p1 使用了旧版的 xyz()。

更多资源请访问稀酷客(www.ckook.com)

```
$ gcc -g -o p2 sv_prog.c libsv.so
$ LD_LIBRARY_PATH=. ./p2
v2 xyz                                          Uses xyz@VER_2
$ LD_LIBRARY_PATH=. ./p1
v1 xyz                                          Uses xyz@VER_1
```

可执行文件的版本标签依赖是在静态链接时进行记录的。使用 objdump –t 可以打印出每个可执行文件的符号表，从而能够显示出两个程序中不同的版本标签依赖。

```
$ objdump -t p1 | grep xyz
08048380       F *UND*  0000002e              xyz@@VER_1
$ objdump -t p2 | grep xyz
080483a0       F *UND*  0000002e              xyz@@VER_2
```

还可以使用 readelf –s 获取类似的信息。

更多有关符号版本化的信息可以通过使用命令 info ld scripts version 以及访问 http://people.redhat.com/drepper/symbol-versioning 来获得。

42.4 初始化和终止函数

可以定义一个或多个在共享库被加载和卸载时自动执行的函数，这样在使用共享库时就能够完成一些初始化和终止工作了。不管库是自动被加载还是使用 dlopen 接口（参见 42.1 节）显式加载的，初始化函数和终止函数都会被执行。

初始化和终止函数是使用 gcc 的 constructor 和 destructor 特性来定义的。在库被加载时需要执行的所有函数都应该定义成下面的形式。

```
void __attribute__ ((constructor)) some_name_load(void)
{
    /* Initialization code */
}
```

类似地，卸载函数的形式如下。

```
void __attribute__ ((destructor)) some_name_unload(void)
{
    /* Finalization code */
}
```

读者可以根据需要使用其他名字替换函数名 some_name_load()和 some_name_unload()。

使用 gcc 的 constructor 和 destructor 特性还能创建主程序的初始化函数和终止函数。

_init()和_fini()函数

用来完成共享库的初始化和终止工作的一项较早的技术是在库中创建两个函数_init()和_fini()。当库首次被进程加载时会执行 void _init(void)中的代码，当库被卸载时会执行 void _fini(void)函数中的代码。

如果创建了_init()和_fini()函数，那么在构建共享库时必须要指定 gcc –nostartfiles 选项以防止链接器加入这些函数的默认实现。（如果需要的话可以使用–Wl,–init 和–Wl,–fini 链接器选项来指定函数的名称。）

有了 gcc 的 constructor 和 destructor 特性之后已经不建议使用_init()和_fini()函数了，因为 gcc 的 constructor 和 destructor 特性允许定义多个初始化和终止函数。

更多资源请访问稀酷客(www.ckook.com)

42.5 预加载共享库

出于测试的目的，有些时候可以有选择地覆盖一些正常情况下会被动态链接器按照 41.11 节中介绍的规则找出的函数（以及其他符号）。要完成这个任务可以定义一个环境变量 LD_PRELOAD，其值由在加载其他共享库之前需加载的共享库名称构成，其中共享库名称之间用空格或冒号分隔。由于首先会加载这些共享库，因此可执行文件自动会使用这些库中定义的函数，从而覆盖那些动态链接器在其他情况下会搜索的同名函数。如假设有一个程序调用了函数 x1()和 x2()，并且这两个函数在 libdemo 库中进行了定义。这样当运行这个程序时会看到下面这样的输出。

```
$ ./prog
Called mod1-x1 DEMO
Called mod2-x2 DEMO
```

（在本例中假设共享库位于其中一个标准目录中，因此无需使用 LD_LIBRARY_PATH 环境变量。）

接着需要覆盖函数 x1()，这可以通过创建另一个包含了不同的 x1()定义的共享库 libalt.so 来完成。在运行这个程序时预加载这个库会得到下面的输出。

```
$ LD_PRELOAD=libalt.so ./prog
Called mod1-x1 ALT
Called mod2-x2 DEMO
```

从上面的输出可以看出程序调用了 libalt.so 中定义的 x1()，但 libalt.so 并没有定义 x2()，因此对 x2()的调用仍然会调用 libdemo.so 中定义的 x2()函数。

LD_PRELOAD 环境变量控制着进程级别的预加载行为。或者可以使用/etc/ld.so.preload 文件来在系统层面完成同样的任务，该文件列出了以空格分隔的库列表。（LD_PRELOAD 指定的库将在加载/etc/ld.so.preload 指定的库之前加载。）

出于安全原因，set-user-ID 和 set-group-ID 程序忽略了 LD_PRELOAD。

42.6 监控动态链接器：LD_DEBUG

有些时候需要监控动态链接器的操作以弄清楚它在搜索哪些库，这可以通过 LD_DEBUG 环境变量来完成。通过将这个变量设置为一个（或多个）标准关键词可以从动态链接器中得到各种跟踪信息。

如果将 help 赋给 LD_DEBUG，那么动态链接器会输出有关 LD_DEBUG 的帮助信息，而指定的命令不会被执行。

```
$ LD_DEBUG=help date
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  all         all previous options combined
  statistics display relocation statistics
```


更多资源请访问稀酷客(www.ckook.com)

```
unused      determine unused DSOs
help        display this help message and exit
```

要将调试信息输出到一个文件中而不是标准输出中，则可以使用 LD_DEBUG_OUTPUT 环境变量指定一个文件名。

当请求与跟踪库搜索相关的信息时会产生很多输出，下面的例子对输出进行了删减。

```
$ LD_DEBUG=libs date
     10687:     find library=librt.so.1 [0]; searching
     10687:      search cache=/etc/ld.so.cache
     10687:       trying file=/lib/librt.so.1
     10687:     find library=libc.so.6 [0]; searching
     10687:      search cache=/etc/ld.so.cache
     10687:       trying file=/lib/libc.so.6
     10687:     find library=libpthread.so.0 [0]; searching
     10687:      search cache=/etc/ld.so.cache
     10687:       trying file=/lib/libpthread.so.0
     10687:     calling init: /lib/libpthread.so.0
     10687:     calling init: /lib/libc.so.6
     10687:     calling init: /lib/librt.so.1
     10687:     initialize program: date
     10687:     transferring control: date
Tue Dec 28 17:26:56 CEST 2010
     10687:     calling fini: date [0]
     10687:     calling fini: /lib/librt.so.1 [0]
     10687:     calling fini: /lib/libpthread.so.0 [0]
     10687:     calling fini: /lib/libc.so.6 [0]
```

每一行开头处的 10687 是指所跟踪的进程的进程 ID，当监控多个进程（如父进程和子进程）时会用到这个值。

在默认情况下，LD_DEBUG 的输出会被写到标准错误上，但可以将一个路径名赋给环境变量 LD_DEBUG_OUTPUT 来将输出重定向到其他地方。

如果需要的话可以给 LD_DEBUG 赋多个选项，各个选项之间用逗号分隔（不能出现空格）。symbols 选项（跟踪动态链接器的符号解析）的输出特别多。

LD_DEBUG 对于由动态链接器隐式加载的库和使用 dlopen()动态加载的库都有效。

出于安全的原因，在 set-user-ID 和 set-setgroup-ID 程序中将会忽略 LD_DEBUG（自 glibc 2.2.5 起）。

42.7 总结

动态链接器提供了 dlopen API，它允许程序在运行时显式地加载其他共享库，这样程序就能够实现插件功能了。

共享库设计的一个重要方面是控制符号的可见性，这样库就能够只导出那些与该库进行链接的程序需要用到的符号了。本章介绍了几项用来控制符号可见性的技术。在这些技术中，版本脚本对符号可见性控制的粒度最细。

本章还介绍了如何使用版本脚本来实现一个共享库导出同一符号的多个定义以供与该库进行链接的不同应用程序使用的模型。（各个应用程序使用它与库进行链接链接时符号的当前定义。）这项技术为传统的在共享库真实名称中使用主要和次要版本号来继续版本化管理的方式提供了一个替代方案。

更多资源请访问稀酷客(www.ckook.com)

在共享库中定义初始化和终止函数允许在加载和卸载库时自动执行一段代码。

使用 LD_PRELOAD 环境变量能够预加载共享库。使用这种机制就能够有选择地覆盖那些动态链接器在正常情况下会在其他共享库中找到的函数和符号。

可以将各种值赋给 LD_DEBUG 环境变量以监控动态链接器的操作。

更多信息

更多信息请参考在 41.14 节中列出的信息源。

42.8 习题

42-1. 编写一个程序来验证当使用 dlclose()关闭一个库时如果其中的符号还在被其他库使用的话将不会卸载这个库。

42-2. 在程序清单 42-1 中的程序（dynload.c）中添加一个 dladdr()调用以获取与 dlsym()返回的地址有关的信息。打印出返回的 Dl_info 结构中各个字段的值并验证这些值是否与预期的值一样。

更多资源请访问稀酷客(www.ckook.com)

第43章

进程间通信简介

本章将对进程和线程之间用来相互通信和同步操作的工具进行一个简要的介绍，下面的章节将会深入介绍这些工具的细节信息。

43.1 IPC 工具分类

图 43-1 总结了 UNIX 系统上各种通信和同步工具，并根据功能将它们分成了三类。

- 通信：这些工具关注进程之间的数据交换。
- 同步：这些进程关注进程和线程操作之间的同步。
- 信号：尽管信号的主要作用并不在此，但在特定场景下仍然可以将它作为一种同步技术。更罕见的是信号还可以作为一种通信技术：信号编号本身是一种形式的信息，并且可以在实时信号上绑定数据（一个整数或指针）。第 20 章到第 22 章对信号进行了介绍。

尽管其中一些工具关注的是同步，但通用术语进程间通信（IPC）通常指代所有这些工具。

从图 43-1 中可以看出，通常几个工具会提供类似的 IPC 功能，之所以会这样是出于下列原因。

- 不同的工具在不同的 UNIX 实现上各自进行演化，随后被移植到了其他 UNIX 系统上。如 FIFO 首先是在 System V 上实现的，而（流）socket 是首先是在 BSD 上实现的。
- 新工具被开发出来用于弥补之前类似的工具存在的不足。如 POSIX IPC 工具（消息队列、信号量以及共享内存）是对较早的 System V IPC 工具的改进。

图 43-1 中被分成一组的工具在一些场景中会提供完全不同的功能。如流 socket 可以用来在网络上通信，而 FIFO 则只能用来在同一机器上的进程间进行通信。

更多资源请访问稀酷客(www.ckook.com)

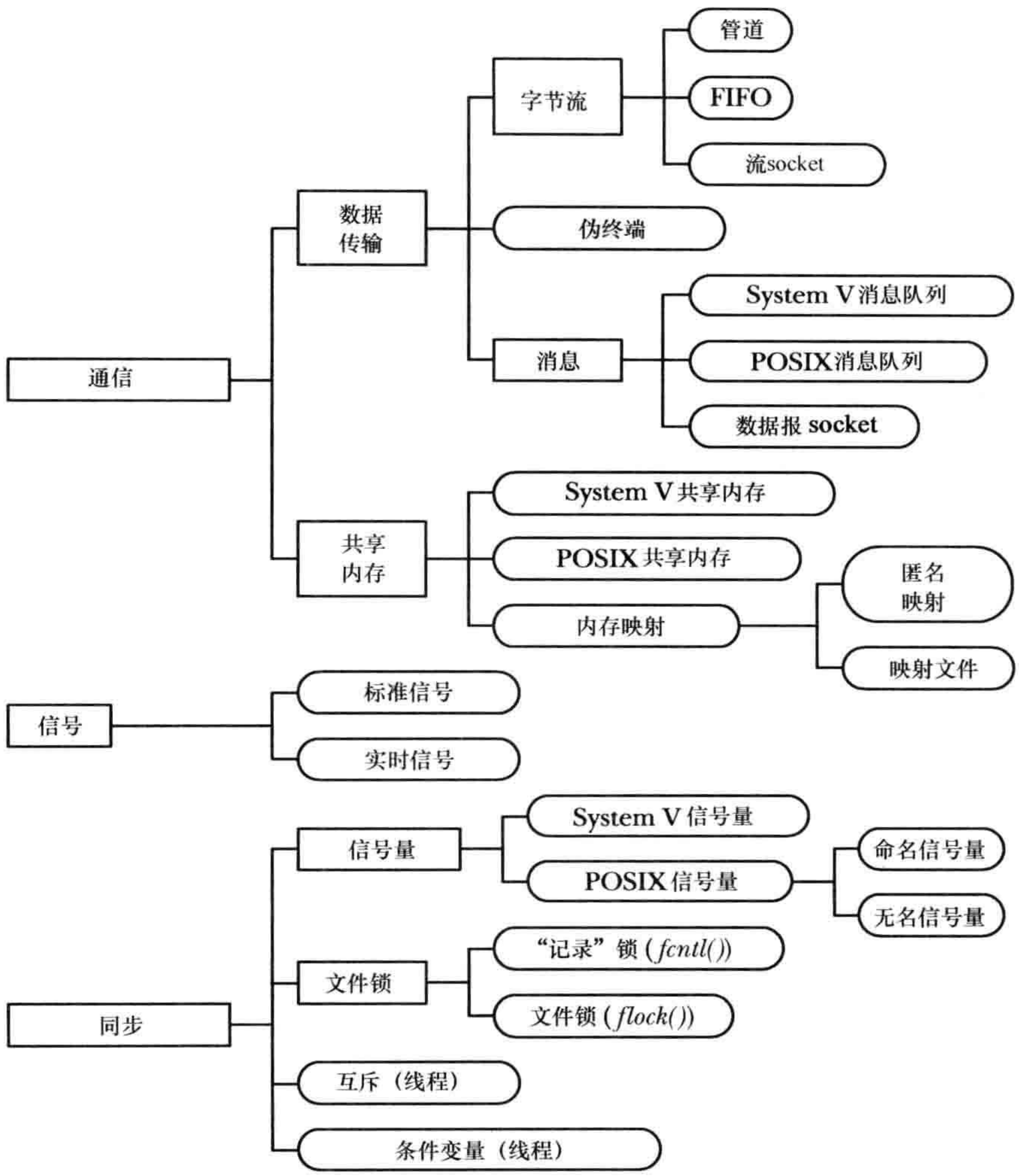


图 43-1：UNIX IPC 工具分类

43.2 通信工具

图 43-1 中列出的各种通信工具允许进程间相互交换数据。（这些工具还可以用来在同一个进程中不同线程之间交换数据，但很少需要这样做，因为线程之间可以通过共享全局变量来交换信息。）

可以将通信工具分成两类。

- 数据传输工具：区分这些工具的关键因素是写入和读取的概念。为了进行通信，一个进程将数据写入到 IPC 工具中，另一个进程从中读取数据。这些工具要求在用户内存和内核内存之间进行两次数据传输：一次传输是在写入的时候从用户内存到内核内存，另一次传输是在读取的时候从内核内存到用户内存。（图 43-2 展示了管道

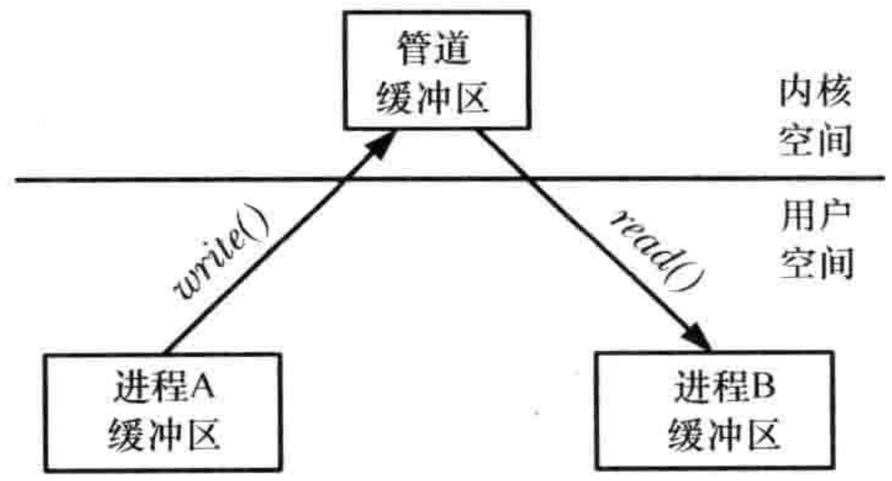


图 43-2：使用管道在两个进程间交换数据

在这一场景中的用法。）

- 共享内存：共享内存允许进程通过将数据放到由进程间共享的一块内存中以完成信息的交换。（内核通过将每个进程中的页表条目指向同一个 RAM 分页来实现这一功能，如图 49-2 所示。）一个进程可以通过将数据放到共享内存块中使得其他进程读取这些数据。由于通信无需系统调用以及用户内存和内核内存之间的数据传输，因此共享内存的速度非常快。

数据传输

可以进一步将数据传输工具分成下列类别。

- 字节流：通过管道、FIFO 以及数据报 socket 交换的数据是一个无分隔符的字节流。每个读取操作可能会从 IPC 工具中读取任意数量的字节，不管写者写入的块的大小是什么。这个模型参考了传统的 UNIX“文件是一个字节序列”模型。
- 消息：通过 System V 消息队列、POSIX 消息队列以及数据报 socket 交换的数据是以分隔符分隔的消息。每个读取操作读取由写者写入的一整条消息，无法只读取部分消息，而把剩余部分留在 IPC 工具中，也无法在一个读取操作中读取多条消息。
- 伪终端：伪终端是一种在特殊情况下使用的通信工具，在 64 章将会介绍有关伪终端的详细信息。

数据传输工具和共享内存之间的差别包括以下几个方面。

- 尽管一个数据传输工具可能会有多个读取者，但读取操作是具有破坏性的。读取操作会消耗数据，其他进程将无法获取所消耗的数据。

在 socket 中可以使用 MSG_PEEK 标记来执行非破坏性读取（参见 61.3 节）。UDP（Internet domain datagram）socket 允许将一条消息广播或组播到多个接收者处（参见 61.12 节）。

- 读取者和写者进程之间的同步是原子的。如果一个读取者试图从一个当前不包含数据的数据传输工具中读取数据，那么在默认情况下读取操作会被阻塞直至一些进程向该工具写入了数据。

共享内存

大多数现代 UNIX 系统提供了三种形式的共享内存：System V 共享内存、POSIX 共享内存以及内存映射。在后面介绍这些工具的章节中将会描述它们之间的差别（特别是在 54.5 节中）。

下面是使用共享内存时的注意点。

- 尽管共享内存的通信速度更快，但速度上的优势是用来弥补需要对在共享内存上发生的操作进行同步的不足的。如当一个进程正在更新共享内存中的一个数据结构时，另一个进程就不应该试图读取这个数据结构。在共享内存中，信号量通常用来作为同步方法。
- 放入共享内存中的数据对所有共享这块内存的进程可见。（这与上面数据传输工具中介绍的破坏性读取语义不同。）

43.3 同步工具

通过图 43-1 中的同步工具可以协调进程的操作。通过同步可以防止进程执行诸如同时更新一块共享内存或同时更新文件的同一个数据块之类的操作。如果没有同步，那么这种同时

更多资源请访问稀酷客(www.ckook.com)

更新的操作可能会导致应用程序产生错误的结果。

UNIX 系统提供了下列同步工具。

- 信号量：一个信号量是一个由内核维护的整数，其值永远不会小于 0。一个进程可以增加或减小一个信号量的值。如果一个进程试图将信号量的值减小到小于 0，那么内核会阻塞该操作直至信号量的值增长到允许执行该操作的程度。（或者进程可以要求执行一个非阻塞操作，那么就不会发生阻塞，内核会让该操作立即返回并返回一个标示无法立即执行该操作的错误。）信号量的含义是由应用程序来确定的。一个进程减小一个信号量（如从 1 到 0）是为了预约对某些共享资源的独占访问，在完成了资源的使用之后可以增加信号量来释放共享资源以供其他进程使用。最常用的信号量是二元信号量——一个值只能是 0 或 1 的信号量，但处理一类共享资源拥有多个实例的应用程序需要使用最大值等于共享资源数量的信号量。Linux 既提供了 System V 信号量，又提供了 POSIX 信号量，它们的功能是类似的。
- 文件锁：文件锁是设计用来协调操作同一文件的多个进程的动作的一种同步方法。它也可以用来协调对其他共享资源的访问。文件锁分为两类：读（共享）锁和写（互斥）锁。任意进程都可以持有同一文件（或一个文件的某段区域）的读锁，但当一个进程持有了一个文件（或文件区域）的写锁之后，其他进程将无法获取该文件（或文件区域）上的读锁和写锁。Linux 通过 flock()和 fcntl()系统调用来提供文件加锁工具。flock()系统调用提供了一种简单的加锁机制，允许进程将一个共享或互斥锁加到整个文件上。由于功能有限，现在已经很少使用 flock()这个加锁工具了。fcntl()系统调用提供了记录加锁，允许进程在同一文件的不同区域上加上多个读锁和写锁。
- 互斥体和条件变量：这些同步工具通常用于 POSIX 线程，第 30 章对此进行了介绍。

一些 UNIX 实现，包括安装了能提供 NPTL 线程实现的 glibc 的 Linux 系统，允许在进程间共享互斥体和条件变量。SUSv3 允许但并不要求实现支持进程间共享的互斥体和条件变量。所有 UNIX 系统都没有提供这个功能，因此很少使用它们来进行进程同步。

在执行进程间同步时通常需要根据功能需求来选择工具。当协调对文件的访问时文件记录加锁通常是最佳的选择，而对于协调对其他共享资源的访问来讲，信号量通常是更佳的选择。

通信工具也可以用来进行同步。如在 44.3 节中使用了一个管道来同步父进程与子进程的动作。一般来讲，所有数据传输工具都可以用来同步，只是同步操作是通过在工具中交换消息来完成的。

自内核 2.6.22 起，Linux 通过 eventfd()系统调用额外提供了一种非标准的同步机制。这个系统调用创建了一个 eventfd 对象，该对象拥有一个相关的由内核维护的 8 字节无符号整数，它返回一个指向该对象的文件描述符。向这个文件描述符中写入一个整数将会把该整数加到对象值上。当对象值为 0 时对该文件描述符的 read()操作将会被阻塞。如果对象的值非零，那么 read()会返回该值并将对象值重置为 0。此外，可以使用 poll()、select()以及 epoll 来测试对象值是否为非零，如果是非零的话就表示文件描述符可读。使用 eventfd 对象进行同步的应用程序必须要首先使用 eventfd()创建该对象，然后调用 fork()创建继承指向该对象的文件描述符的相关进程。更多细节信息可参考 eventfd(2)手册。

更多资源请访问稀酷客(www.ckook.com)

43.4 IPC 工具比较

在需要使用 IPC 时会发现有很多选择，读者在一开始可能会对这些选择感到迷惑。在后面介绍各个 IPC 工具的章节中将会把每个工具与其他类似的工具进行比较。下面介绍在确定选择何种 IPC 工具时通常需要考虑的事项。

IPC 对象标识和打开对象的句柄

要访问一个 IPC 对象，进程必须要通过某种方式来标识出该对象，一旦将对象“打开”之后，进程必须要使用某种句柄来引用该打开着的对象。表 43-1 对各种类型的 IPC 工具的属性进行了总结。

表 43-1：各种 IPC 工具的标识符和句柄

工 具 类 型	用于识别对象的名称	用于在程序中引用对象的句柄
管道	没有名称	文件描述符
FIFO	路径名	文件描述符
UNIX domain socket	路径名	文件描述符
Internet domain socket	IP 地址+端口号	文件描述符
System V 消息队列	System V IPC 键	System V IPC 标识符
System V 信号量	System V IPC 键	System V IPC 标识符
System V 共享内存	System V IPC 键	System V IPC 标识符
POSIX 消息队列	POSIX IPC 路径名	mqd_t (消息队列描述符)
POSIX 命名信号量	POSIX IPC 路径名	sem_t * (信号量指针)
POSIX 无名信号量	没有名称	sem_t * (信号量指针)
POSIX 共享内存	POSIX IPC 路径名	文件描述符
匿名映射	没有名称	无
内存映射文件	路径名	文件描述符
flock()文件锁	路径名	文件描述符
fcntl()文件锁	路径名	文件描述符

功能

各种 IPC 工具在功能上是存在差异的，因此在确定使用何种工具时需要考虑这些差异。下面首先对数据传输工具盒共享内存之间的差异进行总结。

- 数据传输工具提供了读取和写入操作，传输的数据只供一个读者进程消耗。内核会自动处理读者和写者之间的流控以及同步（这样当读者试图从当前为空的工具中读取数据时将会阻塞）。在很多应用程序设计中，这个模型都表现得很好。
- 其他应用程序设计则更适合采用共享内存的方式。一个进程通过共享内存能够使数据对共享同一内存区域的所有进程可见。通信“操作”是比较简单的——进程可以像访问自己的虚拟地址空间中的内存那样访问共享内存中的数据。另一个方面，同步处理（可能还会有流控）会增加共享内存设计的复杂性。在需要维护共享状态（如共享数

更多资源请访问稀酷客(www.ckook.com)

据结构）的应用程序中，这个模型表现得很好。

关于各种数据传输工具，下面几点是值得注意的。

- 一些数据传输工具以字节流的形式传输数据（管道、FIFO 以及流 socket），另一些则是面向消息的（消息队列和数据报 socket）。到底选择何种方法则需要依赖于应用程序。（应用程序也可以在一个字节流工具上应用面向消息的模型，这可以通过使用分隔字符、固定长度的消息，或对整条消息长度进行编码的消息头来实现，具体可参考 44.8 节）。
- 与其他数据传输工具相比，System V 和 POSIX 消息队列特有的一个特性是它们能够给消息赋一个数值类型或优先级，这样递送消息的顺序就可以与发送消息的顺序不同了。
- 管道、FIFO 以及 socket 是使用文件描述符来实现的。这些 IPC 工具都支持第 63 章中介绍的一组 I/O 模型：I/O 多路复用（select()和 poll()系统调用）、信号驱动的 I/O、以及 Linux 特有的 epoll API。这些技术的主要优势在于它们允许应用程序同时监控多个文件描述符以判断是否可以在某些文件描述符上执行 I/O 操作。与之相比，System V 消息队列没有使用文件描述符，因此并不支持这些技术。

在 Linux 上，POSIX 消息队列也是使用文件描述符来实现的，因此也支持上面介绍的各种 I/O 技术。但 SUSv3 并没有规定这种行为，因此在大多数实现上并不支持这些技术。

- POSIX 消息队列提供了一个通知工具，当一条消息进入了一个之前为空的队列中时可以使用它来向进程发送信号或实例化一个新线程。
- UNIX domain socket 提供了一个特性允许在进程间传递文件描述符。这样一个进程就能够打开一个文件并使之对另一个本来无法访问该文件的进程可用，在 61.13.3 节中将会对此特性进行简要介绍。
- UDP（Internet domain datagram）socket 允许一个发送者向多个接收者广播或组播一条消息，在 61.12 节中将会对此特性进行简要介绍。

关于进程同步工具，下面几点是值得注意的。

- 使用 fcntl()加上的记录锁由加锁的进程拥有。内核使用这种所有权属性来检测死锁（两个或多个进程持有的锁会阻塞对方后续的加锁请求的场景）。如果发生了死锁，那么内核会拒绝其中一个进程的加锁请求，因此会在 fcntl()调用中返回一个错误标示出死锁的发生。System V 和 POSIX 信号量并没有所有权属性，因此内核不会为信号量进行死锁检测。
- 当使用 fcntl()获得记录锁的进程终止之后会自动释放该记录锁。System V 信号量提供了一个类似的特性，即“撤销”特性，但这个特性仅在部分场景中可靠（参见 47.8 节）。POSIX 信号量并没有提供类似的特性。

网络通信

在图 43-1 中给出所有 IPC 方法中，只有 socket 允许进程通过网络来通信。socket 一般用于两个域中：一个是 UNIX domain，它允许位于同一系统上的进程进行通信；另一个是 Internet domain，它允许位于通过 TCP/IP 网络进行连接的不同主机上的进程进行通信。通常，将一个使用 UNIX domain socket 进行通信的程序转换成一个使用 Internet domain socket 进行通信的程序只需要做出微小的改动，这样只需要对使用 UNIX domain socket 的应用程序做较小的改动

更多资源请访问稀酷客(www.ckook.com)

就可以将它应用于网络场景。

可移植性

现代 UNIX 实现支持图 43-1 中的大部分 IPC 工具，但 POSIX IPC 工具（消息队列、信号量以及共享内存）的普及程度远远不如 System V IPC，特别是在较早的 UNIX 系统上。（只有版本为 2.6.x 的 Linux 内核系列才提供了一个 POSIX 消息队列的实现以及对 POSIX 信号量的完全支持。）因此，从可移植性的角度来看，System V IPC 要优于 POSIX IPC。

System V IPC 设计问题

System V IPC 工具被设计成独立于传统的 UNIX I/O 模型，其结果是其中一些特性使得它的编程接口的用法更加复杂。相应的 POSIX IPC 工具被设计用来解决这些问题，特别是下面几点需要注意。

- System V IPC 工具是无连接的，它们没有提供引用一个打开的 IPC 对象的句柄（类似于文件描述符）的概念。在后面的章节中有时候会将“打开”一个 System V IPC 对象，但这仅仅是描述进程获取一个引用该对象的句柄的简便方式。内核不会记录进程已经“打开”了该对象（与其他 IPC 对象不同）。这意味着内核无法维护当前使用该对象的进程的引用计数，其结果是应用程序需要使用额外的代码来知道何时可以安全地删除一个对象。
- System V IPC 工具的编程接口与传统的 UNIX I/O 模型是不一致的（它们使用整数键值和 IPC 标识符，而不是路径名和文件描述符），并且这个编程接口也过于复杂了。这一点在 System V 信号量上表现得特别明显（参见 47.11 节和 53.5 节）。

相反，内核会为 POSIX IPC 对象记录打开的引用数，这样就简化了何时删除对象的决策。此外，POSIX IPC 提供的接口更加简单并且与传统的 UNIX 模型也更加一致。

可访问性

表 43-2 中的第二列总结了各种 IPC 工具的一个重要特性：权限模型控制着哪些进程能够访问对象。下面介绍各种模型的细节信息。

- 对于一些 IPC 工具（如 FIFO 和 socket），对象名位于文件系统中，可访问性是根据相关的文件权限掩码（指定了所有者、组和其他用户的权限）来确定的（参见 15.4 节）。虽然 System V IPC 对象并不位于文件系统中，但每个对象拥有一个相关的权限掩码，其语义与文件的权限掩码类似。
- 一些 IPC 工具（管道、匿名内存映射）被标记成只允许相关进程访问。这里“相关”指通过 fork()关联的。为了使两个进程能够访问同一个对象，其中一个必须要创建该对象，然后调用 fork()。而 fork()调用的结果就是子进程会继承引用该对象的一个句柄，这样两个进程就能够共享对象了。
- POSIX 的未命名信号量的可访问性是通过包含该信号量的共享内存区域的可访问性来确定的。
- 为了给一个文件加锁，进程必须要拥有一个引用该文件的文件描述符（即在实践中它必须要拥有打开文件的权限）。
- 对 Internet domain socket 的访问（即连接或发送数据报）没有限制。如果有需要的话，必须要在应用程序中实现访问控制。

更多资源请访问稀酷客(www.ckook.com)

表 43-2：各种 IPC 工具的可访问性和持久性

工具类型	可访问性	持久性
管道	仅允许相关进程	进程
FIFO	权限掩码	进程
UNIX domain socket	权限掩码	进程
Internet domain socket	任意进程	进程
System V 消息队列	权限掩码	内核
System V 信号量	权限掩码	内核
System V 共享内存	权限掩码	内核
POSIX 消息队列	权限掩码	内核
POSIX 命名信号量	权限掩码	内核
POSIX 无名信号量	相应内存的权限	依情况而定
POSIX 共享内存	权限掩码	内核
匿名映射	仅允许相关进程	进程
内存映射文件	权限掩码	文件系统
flock()文件锁	文件的 open()操作	进程
fcntl()文件锁	文件的 open()操作	进程

持久性

术语持久性是指一个 IPC 工具的生命周期。（参见表 43-2 中的第三列。）持久性有三种。

- 进程持久性：只要存在一个进程持有进程持久的 IPC 对象，那么该对象的生命周期就不会终止。如果所有进程都关闭了对象，那么与该对象的所有内核资源都会被释放，所有未读取的数据会被销毁。管道、FIFO 以及 socket 是进程持久的 IPC 工具。

FIFO 的数据持久性与其名称的持久性是不同的。FIFO 在文件系统中拥有一个名称，当所有引用 FIFO 的文件描述符都被关闭之后该名称也是持久的。

- 内核持久性：只有当显式地删除内核持久的 IPC 对象或系统关闭时，该对象才会销毁。这种对象的生命周期与是否有进程打开该对象无关。这意味着一个进程可以创建一个对象，向其中写入数据，然后关闭该对象（或终止）。在后面某个时刻，另一个进程可以打开该对象，然后从中读取数据。具备内核持久性的工具包括 System V IPC 和 POSIX IPC。在后面章节中用来描述这些工具的示例程序中将会使用这个属性：对于每种工具都实现一个单独的程序，在程序中创建一个对象，然后删除该对象，并执行通信或同步操作。
- 文件系统持久性：具备文件系统持久性的 IPC 对象会在系统重启的时候保持其中的信息，这种对象一直存在直至被显式地删除。唯一一种具备文件系统持久性的 IPC 对象是基于内存映射文件的共享内存。

性能

在一些场景中，不同 IPC 工具的性能可能存在显著的差异。但在后面的章节中一般不会

更多资源请访问稀酷客(www.ckook.com)

对它们的性能进行比较，其原因如下。

- 在应用程序的整体性能中，IPC 工具的性能的影响因素可能不是很大，并且确定选择何种 IPC 工具可能并不仅仅需要考虑其性能因素。
- 各种 IPC 工具在不同 UNIX 实现或 Linux 的不同内核中的性能可能是不同的。
- 最重要的是，IPC 工具的性能可能会受到使用方式和环境的影响。相关的因素包括每个 IPC 操作交换的数据单元的大小、IPC 工具中未读数据量可能很大、每个数据单元的交换是否需要进行进程上下文切换、以及系统上的其他负载。

如果 IPC 性能是至关紧要的，并且不存在应用程序在与目标系统匹配的环境中运行的性能基准，那么最好编写一个抽象软件层来向应用程序隐藏 IPC 工具的细节，然后在抽象层下使用不同的 IPC 工具来测试性能。

43.5 总结

本章概述了进程（以及线程）可用来相互通信和同步动作的各种工具。

Linux 提供的通信工具包括管道、FIFO、socket、消息队列以及共享内存。Linux 提供的同步工具包括信号量和文件锁。

在很多情况下在执行一个给定的任务时存在多种技术可用于通信和同步。本章以多种方式对不同的技术进行了比较，其目标是突出可能对技术选择产生影响的一些差异。

在后面的章节中将会深入介绍各种通信和同步工具。

43.6 习题

43-1. 编写一个程序来测量管道的带宽。在命令行参数中，程序应该接收需发送的数据块数目以及每个数据块的大小。在创建一个管道之后，程序将分成两个进程：一个子进程以尽可能快的速度向管道写入数据块，父进程读取数据块。在所有数据都被读取之后，父进程应该打印出所消耗的时间和带宽（每秒传输的字节数）。为不同的数据块大小测量带宽。

43-2. 使用 System V 消息队列、POSIX 消息队列、UNIX domain 流 socket 以及 UNIX domain 数据报 socket 来重做上面的练习。使用这些程序来比较各种 IPC 工具在 Linux 上的相对性能。读者如果能够使用其他 UNIX 实现，那么在那些系统上执行同样的比较。

更多资源请访问稀酷客(www.ckook.com)

第44章

管道和 FIFO

本章介绍管道和 FIFO。管道是 UNIX 系统上最古老的 IPC 方法，它在 20 世纪 70 年代早期 UNIX 的第三个版本上就出现了。管道为一个常见需求提供了一个优雅的解决方案：给定两个运行不同程序（命令）的进程，在 shell 中如何让一个进程的输出作为另一个进程的输入呢？管道可以用来在相关进程之间传递数据（读者阅读完后面的几页之后就能够理解“相关”的含义了）。FIFO 是管道概念的一个变体，它们之间的一个重要差别在于 FIFO 可以用于任意进程间的通信。

44.1 概述

每个 shell 用户都对在命令中使用管道比较熟悉，如下面这个统计一个目录中文件的数目的命令所示。

```
$ ls | wc -l
```

为执行上面的命令，shell 创建了两个进程来分别执行 ls 和 wc。（这是通过使用 fork()和 exec()来完成的，第 24 章和第 27 章分别对这两个函数进行了介绍。）图 44-1 展示了这两个进程是如何使用管道的。

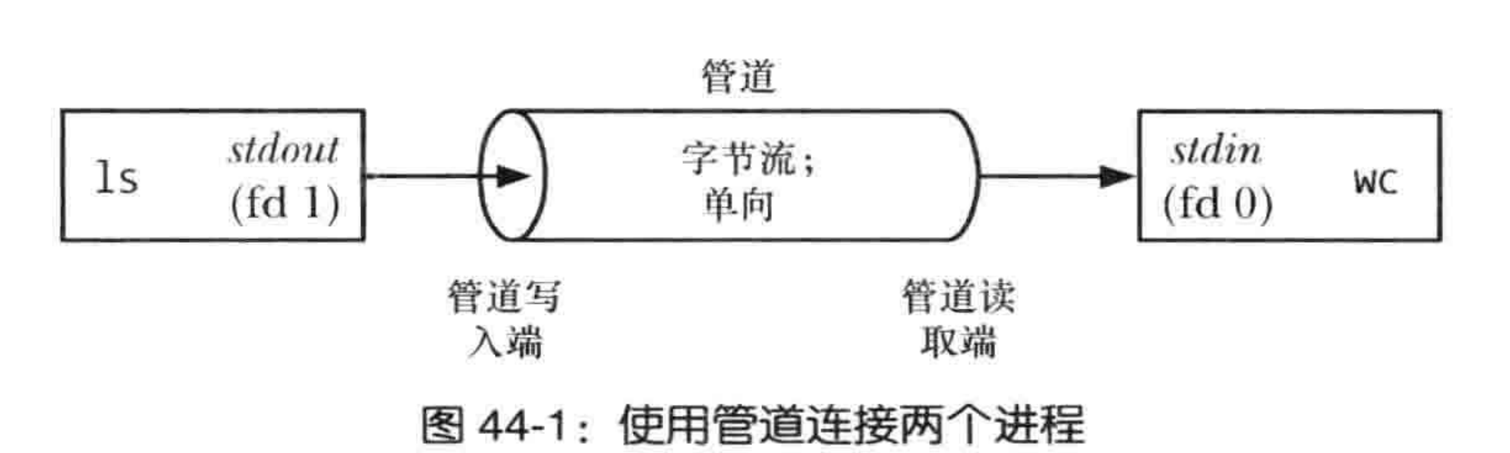


图 44-1：使用管道连接两个进程

除了说明管道的用法之外，图 44-1 的另外一个目的是阐明管道这个名称的由来。可以将管道看成是一组铅管，它允许数据从一个进程流向另一个进程。

在图 44-1 中有一点值得注意的是两个进程都连接到了管道上，这样写入进程（ls）就将其标准输出（文件描述符为 1）连接到了管道的写入端，读取进程（wc）就将其标准输入（文

更多资源请访问稀酷客(www.ckook.com)

件描述符为 0）连接到管道的读取端。实际上，这两个进程并不知道管道的存在，它们只是从标准文件描述符中读取数据和写入数据。shell 必须要完成相关的工作，在 44.4 节中将会介绍 shell 是如何完成这些工作的。

下面几个段落将会介绍管道的几个重要特征。

一个管道是一个字节流

当讲到管道是一个字节流时意味着在使用管道时是不存在消息或消息边界的概念的。从管道中读取数据的进程可以读取任意大小的数据块，而不管写入进程写入管道的数据块的大小是什么。此外，通过管道传递的数据是顺序的——从管道中读取出来的字节的顺序与它们被写入管道的顺序是完全一样的。在管道中无法使用 lseek()来随机地访问数据。

如果需要在管道中实现离散消息的概念，那么就必须要在应用程序中完成这些工作。虽然这是可行的（参见 44.8 节），但如果碰到这种需求的话最好使用其他 IPC 机制，如消息队列和数据报 socket，本书在后面几个章节中就会介绍它们。

从管道中读取数据

试图从一个当前为空的管道中读取数据将会被阻塞直到至少有一个字节被写入到管道中为止。如果管道的写入端被关闭了，那么从管道中读取数据的进程在读完管道中剩余的所有数据之后将会看到文件结束（即 read()返回 0）。

管道是单向的

在管道中数据的传递方向是单向的。管道的一段用于写入，另一端则用于读取。

在其他一些 UNIX 实现上——特别是那些从 System V Release 4 演化而来的系统——管道是双向的（所谓的流管道）。双向管道并没有在任何 UNIX 标准中进行规定，因此即使在提供了双向管道的实现上最好也避免依赖这种语义。作为替代方案，可以使用 UNIX domain 流 socket 对（通过使用 57.5 节中介绍的 socketpair()系统调用来创建），它提供了一种标准的双向通信机制，并且其语义与流管道是等价的。

可以确保写入不超过 PIPE_BUF 字节的操作是原子的

如果多个进程写入同一个管道，那么如果它们在一个时刻写入的数据量不超过 PIPE_BUF 字节，那么就可以确保写入的数据不会发生相互混合的情况。

SUSv3 要求 PIPE_BUF 至少为_POSIX_PIPE_BUF（512）。一个实现应该定义 PIPE_BUF（在<limits.h>中）并/或允许调用 fpathconf(fd,_PC_PIPE_BUF)来返回原子写入操作的实际上限。不同 UNIX 实现上的 PIPE_BUF 不同，如在 FreeBSD 6.0 其值为 512 字节，在 Tru64 5.1 上其值为 4096 字节，在 Solaris 8 上其值为 5120 字节。在 Linux 上，PIPE_BUF 的值为 4096。

当写入管道的数据块的大小超过了 PIPE_BUF 字节，那么内核可能会将数据分割成几个较小的片段来传输，在读者从管道中消耗数据时再附加上后续的数据。（write()调用会阻塞直到所有数据被写入到管道为止。）当只有一个进程向管道写入数据时（通常的情况），PIPE_BUF 的取值就没有关系了。但如果有多个写入进程，那么大数据块的写入可能会被分解成任意大小的段（可能会小于 PIPE_BUF 字节），并且可能会出现与其他进程写入的数据交叉的现象。

只有在数据被传输到管道的时候 PIPE_BUF 限制才会起作用。当写入的数据达到 PIPE_BUF 字节时，write()会在必要的时候阻塞直到管道中的可用空间足以原子地完成操作。

更多资源请访问稀酷客(www.ckook.com)

如果写入的数据大于 PIPE_BUF 字节，那么 write()会尽可能地多传输数据以充满整个管道，然后阻塞直到一些读取进程从管道中移除了数据。如果此类阻塞的 write()被一个信号处理器中断了，那么这个调用会被解除阻塞并返回成功传输到管道中的字节数，这个字节数会少于请求写入的字节数（所谓的部分写入）。

在 Linux 2.2 上，向管道写入任意数量的数据都是原子的，除非写入操作被一个信号处理器中断了。在 Linux 2.4 以及后续的版本上，写入数据量大于 PIPE_BUF 字节的所有操作都可能会与其他进程的写入操作发生交叉。（在版本号为 2.2 和 2.4 的内核中，实现管道的内核代码存在很大的差异。）

管道的容量是有限的

管道其实是一个在内核内存中维护的缓冲器，这个缓冲器的存储能力是有限的。一旦管道被填满之后，后续向该管道的写入操作就会被阻塞直到读者从管道中移除了一些数据为止。

SUSv3 并没有规定管道的存储能力。在早于 2.6.11 的 Linux 内核中，管道的存储能力与系统页面的大小是一致的（如在 x86-32 上是 4096 字节），而从 Linux 2.6.11 起，管道的存储能力是 65,536 字节。其他 UNIX 实现上的管道的存储能力可能是不同的。

一般来讲，一个应用程序无需知道管道的实际存储能力。如果需要防止写者进程阻塞，那么从管道中读取数据的进程应该被设计成以尽可能快的速度从管道中读取数据。

从理论上来讲，没有任何理由可以支持存储能力较小的管道无法正常工作这个结论，哪怕管道的存储能力只有一个字节。使用较大的缓冲器的原因是效率：每当写者充满管道时，内核必须要执行一个上下文切换以允许读者被调度来消耗管道中的一些数据。使用较大的缓冲器意味着需执行的上下文切换次数更少。

从 Linux 2.6.35 开始就可以修改一个管道的存储能力了。Linux 特有的 fcntl(fd, F_SETPIPE_SZ, size)调用会将 fd 引用的管道的存储能力修改为至少 size 字节。非特权进程可以将管道的存储能力修改为范围在系统的页面大小到/proc/sys/fs/pipe-max-size 中规定的值之内的任何一个值。pipe-max-size 的默认值是 1048576 字节。特权（CAP_SYS_RESOURCE）进程可以覆盖这个限制。在为管道分配空间时，内核可能会将 size 提升为对实现来讲更加便捷的某个值。fcntl(fd, F_GETPIPE_SZ)调用返回为管道分配的实际大小。

44.2 创建和使用管道

pipe()系统调用创建一个新管道。

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Returns 0 on success, or −1 on error

成功的 pipe()调用会在数组 filedes 中返回两个打开的文件描述符：一个表示管道的读取端（filedes[0]），另一个表示管道的写入端（filedes[1]）。

更多资源请访问稀酷客(www.ckook.com)

与所有文件描述符一样，可以使用 read()和 write()系统调用来在管道上执行 I/O。一旦向管道的写入端写入数据之后立即就能从管道的读取端读取数据。管道上的 read()调用会读取的数据量为所请求的字节数与管道中当前存在的字节数两者之间较小的那个（但当管道为空时阻塞）。

也可以在管道上使用 stdio 函数（printf()、scanf()等），只需要首先使用 fdopen()获取一个与 filedes 中的某个描述符对应的文件流即可（参见 13.7 节）。但在这样做的时候需要清楚在 44.6 节中介绍的 stdio 缓冲问题。

---

ioctl(fd, FIONREAD, &cnt)调用返回文件描述符 fd 所引用的管道或 FIFO 中未读取的字节数。其他一些实现也提供了这个特性，但 SUSv3 并没有对此进行规定。

图 44-2 给出了使用 pipe()创建完管道之后的情况，其中调用进程通过文件描述符引用了管道的两端。

在单个进程中管道的用途不多（在 63.5.2 节中将会介绍一种用途）。一般来讲都是使用管道让两个进程进行通信。为了让两个进程通过管道进行连接，在调用完 pipe()之后可以调用 fork()。在 fork()期间，子进程会继承父进程的文件描述符的副本（参见 24.2.1 节），这样就会出现图 44-3 中左边那样的情形。

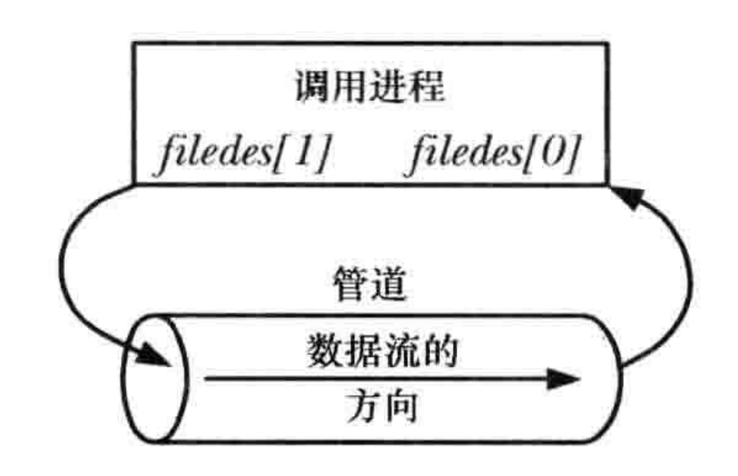
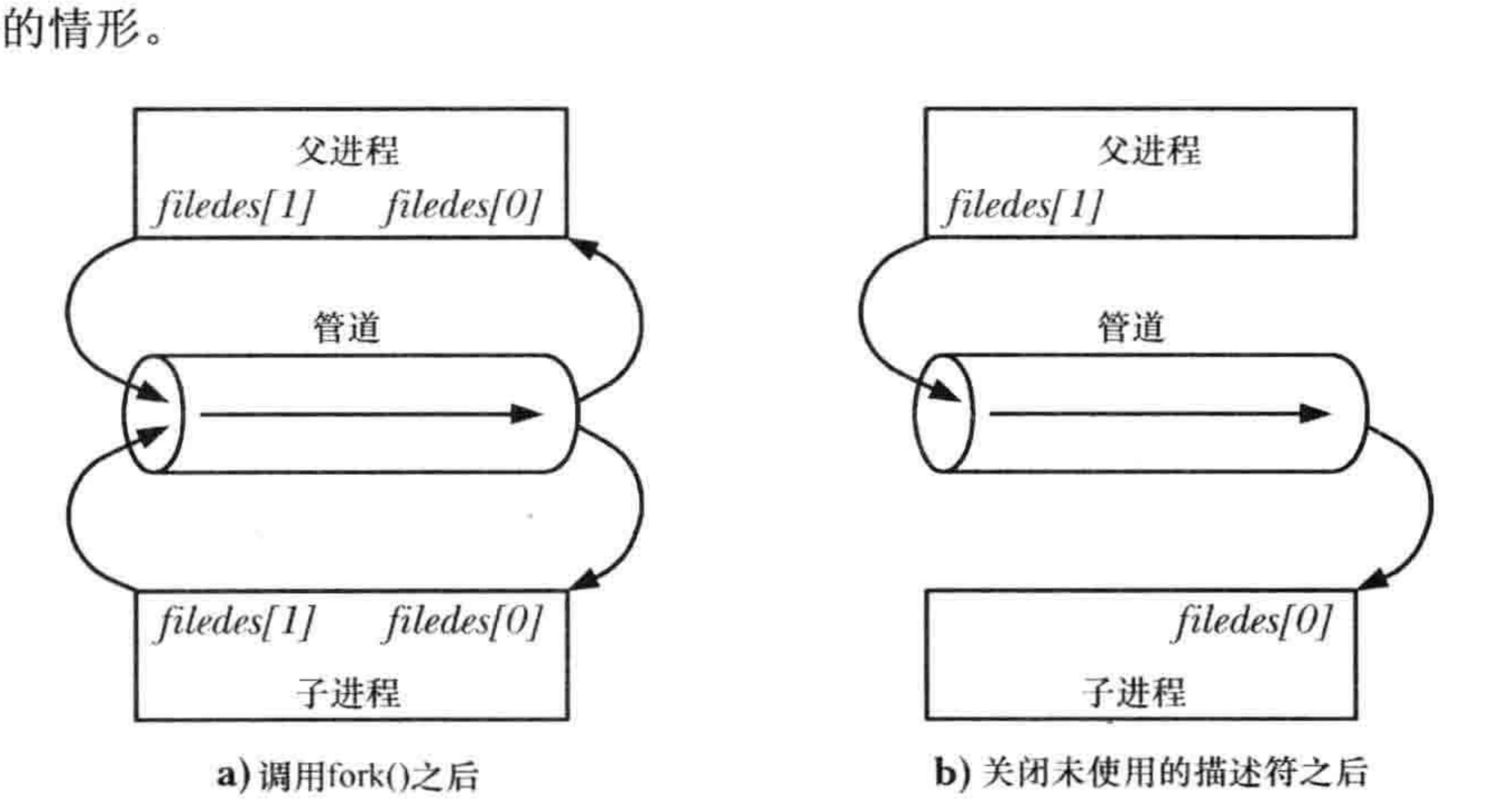


图 44-2：在创建完管道之后处理文件描述符



a) 调用fork()之后　　b) 关闭未使用的描述符之后

图 44-3：设置管道来将数据从父进程传输到子进程

虽然父进程和子进程都可以从管道中读取和写入数据，但这种做法并不常见。因此，在 fork()调用之后，其中一个进程应该立即关闭管道的写入端的描述符，另一个则应该关闭读取端的描述符。如，如果父进程需要向子进程传输数据，那么它就会关闭管道的读取端的描述符 filedes[0]，而子进程就会关闭管道的写入端的描述符 filedes[1]，这样就出现了图 44-3 中右边那样的情形。程序清单 44-1 给出了创建这个管道的代码。

程序清单 44-1：使用管道将数据从父进程传输到子进程所需的步骤

```
int filedes[2];

if (pipe(filedes) == -1)                    /* Create the pipe */
    errExit("pipe");
```


更多资源请访问稀酷客(www.ckook.com)

```
switch (fork()) {                           /* Create a child process */
case -1:
    errExit("fork");

case 0:  /* Child */
    if (close(filedes[1]) == -1)            /* Close unused write end */
        errExit("close");

    /* Child now reads from pipe */
    break;

default: /* Parent */
    if (close(filedes[0]) == -1)            /* Close unused read end */
        errExit("close");

    /* Parent now writes to pipe */
    break;
}
```

让父进程和子进程都能够从同一个管道中读取和写入数据这种做法并不常见的一个原因是如果两个进程同时试图从管道中读取数据，那么就无法确定哪个进程会首先读取成功——两个进程竞争数据了。要防止这种竞争情况的出现就需要使用某种同步机制。但如果需要双向通信则可以使用一种更加简单的方法：创建两个管道，在两个进程之间发送数据的两个方向上各使用一个。（如果使用这种技术，那么就需要考虑死锁的问题了，因为如果两个进程都试图从空管道中读取数据或尝试向已满的管道中写入数据就可能会发生死锁。）

虽然可以有多个进程向单个管道中写入数据，但通常只存在一个写者。（在 44.3 节中将会给出一个使用多个写者向一个管道写入数据的例子。）相反，在有些情况下让 FIFO 拥有多个写者是比较有用的，在 44.8 节中将会给出一个这样的例子。

从 2.6.27 内核开始，Linux 支持一个全新的非标准系统调用 pipe2()。这个系统调用执行的任务与 pipe()一样，但支持额外的参数 flags，这个参数可以用来修改系统调用的行为。这个系统调用支持两个标记，一个是 O_CLOEXEC，它会导致内核为两个新的文件描述符启用 close-on-exec 标记（FD_CLOEXEC）。这个标记之所以有用的原因与在 4.3.1 节中介绍的 open() O_CLOEXEC 标记有用的原因一样。另一个是 O_NONBLOCK 标记，它会导致内核将底层的打开的文件描述符标记为非阻塞，这样后续的 I/O 操作会是非阻塞的。这样就能够在不调用 fcntl()的情况下达到同样的效果了。

管道允许相关进程间的通信

目前为止本章已经介绍了如何使用管道来让父进程和子进程之间进行通信，其实管道可以用于任意两个（或更多）相关进程之间的通信，只要在创建子进程的系列 fork()调用之前通过一个共同的祖先进程创建管道即可。（这就是本章开头部分所讲的“相关进程”的含义。）如管道可用于一个进程和其孙子进程之间的通信。第一个进程创建管道，然后创建子进程，接着子进程再创建第一个进程的孙子进程。管道通常用于两个兄弟进程之间的通信——它们的父进程创建了管道，然后创建两个子进程。这就是在构建管道线时 shell 所做的工作。

更多资源请访问稀酷客(www.ckook.com)

管道只能用于相关进程之间的通信这个说法存在一种例外情况。通过 UNIX domain socket（在 61.13.3 节中将会简要介绍的一项技术）传递一个文件描述符使得将管道的一个文件描述符传递给一个非相关进程成为可能。

关闭未使用管道文件描述符

关闭未使用管道文件描述符不仅仅是为了确保进程不会耗尽其文件描述符的限制——这对于正确使用管道是非常重要的。下面介绍为何必须要关闭管道的读取端和写入端的未使用文件描述符。

从管道中读取数据的进程会关闭其持有的管道的写入描述符，这样当其他进程完成输出并关闭其写入描述符之后，读者就能够看到文件结束（在读完管道中的数据之后）。

如果读取进程没有关闭管道的写入端，那么在其他进程关闭了写入描述符之后，读者也不会看到文件结束，即使它读完了管道中的所有数据。相反，read()将会阻塞以等待数据，这是因为内核知道至少还存在一个管道的写入描述符打开着，即读取进程自己打开了这个描述符。从理论上来讲，这个进程仍然可以向管道写入数据，即使它已经被读取操作阻塞了。如 read()可能 hiu 被一个向管道写入数据的信号处理器中断。（这是现实世界中的一种场景，读者在 63.5.2 节中将会看到。）

写入进程关闭其持有的管道的读取描述符是出于不同的原因。当一个进程试图向一个管道中写入数据但没有任何进程拥有该管道的打开着的读取描述符时，内核会向写入进程发送一个 SIGPIPE 信号。在默认情况下，这个信号会杀死一个进程。但进程可以捕获或忽略该信号，这样就会导致管道上的 write()操作因 EPIPE 错误（已损坏的管道）而失败。收到 SIGPIPE 信号或得到 EPIPE 错误对于标示出管道的状态是有用的，这就是为何需要关闭管道的未使用读取描述符的原因。

注意：对被 SIGPIPE 处理器中断的 write()的处理是特殊的。通常，当 write()（或其他“慢”系统调用）被一个信号处理器中断时，这个调用会根据是否使用 sigaction() SA_RESTART 标记安装了处理器而自动重启或因 EINTR 错误而失败（参见 21.5 节）。对 SIGPIPE 的处理不同是因为自动重启 write()或简单标示出 write()被一个处理器中断了是毫无意义的（意味着需要手工重启 write()）。不管是何种处理方式，后续的 write()都不会成功，因为管道仍然处于被损坏的状态。

如果写入进程没有关闭管道的读取端，那么即使在其他进程已经关闭了管道的读取端之后写入进程仍然能够向管道写入数据，最后写入进程会将数据充满整个管道，后续的写入请求会被永远阻塞。

关闭未使用文件描述符的最后一个原因是只有当所有进程中所有引用一个管道的文件描述符被关闭之后才会销毁该管道以及释放该管道占用的资源以供其他进程复用。此时，管道中所有未读取的数据都会丢失。

示例程序

程序清单 44-2 中的程序演示了如何将管道用于父进程和子进程之间的通信。这个例子演示了前面提及的管道的字节流特性——父进程在一个操作中写入数据，子进程一小块一小块地从管道中读取数据。

更多资源请访问稀酷客(www.ckook.com)

主程序调用 pipe()创建管道①，然后调用 fork()创建一个子进程②。在 fork()调用之后，父进程关闭了其持有的管道的读取端的文件描述符⑧并将通过程序的命令行参数传递进来的字符串写到管道的写入端⑨。父进程接着关闭管道的读取端⑩并调用 wait()等待子进程终止⑪。在关闭了所持有的管道的写入端的文件描述符③之后，子进程进入了一个循环，在这个循环中从管道读取④数据块并将它们写到⑥标准输出中。当子进程碰到管道的文件结束时⑤就退出循环⑦，并写入一个结尾换行字符以及关闭所持有的管道的读取端的描述符，最后终止。

下面是运行程序清单 44-2 中的程序时可能看到的输出。

```
$ ./simple_pipe 'It was a bright cold day in April, '\
'and the clocks were striking thirteen.'
It was a bright cold day in April, and the clocks were striking thirteen.
```

程序清单 44-2：在父进程和子进程间使用管道通信

pipes/simple_pipe.c

```
#include <sys/wait.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 10

int
main(int argc, char *argv[])
{
    int pfd[2];                             /* Pipe file descriptors */
    char buf[BUF_SIZE];
    ssize_t numRead;

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

①   if (pipe(pfd) == -1)                   /* Create the pipe */
        errExit("pipe");

②   switch (fork()) {
    case -1:
        errExit("fork");

    case 0:             /* Child  - reads from pipe */
③       if (close(pfd[1]) == -1)            /* Write end is unused */
            errExit("close - child");

        for (;;) {              /* Read data from pipe, echo on stdout */
④           numRead = read(pfd[0], buf, BUF_SIZE);
            if (numRead == -1)
                errExit("read");
⑤           if (numRead == 0)
                break;                      /* End-of-file */
⑥           if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("child - partial/failed write");
        }

⑦       write(STDOUT_FILENO, "\n", 1);
        if (close(pfd[0]) == -1)
            errExit("close");
        _exit(EXIT_SUCCESS);

    default:            /* Parent - writes to pipe */
```

```
⑧         if (close(pfd[0]) == -1)            /* Read end is unused */
              errExit("close - parent");
⑨         if (write(pfd[1], argv[1], strlen(argv[1])) != strlen(argv[1]))
              fatal("parent - partial/failed write");

⑩         if (close(pfd[1]) == -1)            /* Child will see EOF */
              errExit("close");
⑪         wait(NULL);                         /* Wait for child to finish */
          exit(EXIT_SUCCESS);
      }
  }
```

pipes/simple_pipe.c

44.3 将管道作为一种进程同步的方法

在 24.5 节中介绍了如何使用信号来同步父进程和子进程的动作以防止出现竞争条件。也可以使用管道来取得类似的结果，如程序清单 44-3 中给出的骨架程序所示。这个程序创建了多个子进程（每个命令行参数对应一个子进程），每个子进程都完成某个动作，在本例中则是睡眠一段时间。父进程等待直到所有子进程完成了自己的动作为止。

为了执行同步，父进程在创建子进程②之前构建了一个管道①。每个子进程会继承管道的写入端的文件描述符并在完成动作之后关闭这些描述符③。当所有子进程都关闭了管道的写入端的文件描述符之后，父进程在管道上的 read()⑤就会结束并返回文件结束（0）。这时，父进程就能够做其他工作了。（注意在父进程中关闭管道的未使用写入端④对于这项技术的正常运转是至关重要的，否则父进程在试图从管道中读取数据时会被永远阻塞。）

下面是使用程序清单 44-3 中的程序创建三个分别睡眠 4、2 和 6 秒的子进程时所看到的输出。

```
$ ./pipe_sync 4 2 6
08:22:16  Parent started
08:22:18  Child 2 (PID=2445) closing pipe
08:22:20  Child 1 (PID=2444) closing pipe
08:22:22  Child 3 (PID=2446) closing pipe
08:22:22  Parent ready to go
```

程序清单 44-3：使用管道同步多个进程

pipes/pipe_sync.c

```
#include "curr_time.h"                      /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];                             /* Process synchronization pipe */
    int j, dummy;
       if (argc < 2 || strcmp(argv[1], "--help") == 0)
           usageErr("%s sleep-time...\n", argv[0]);

       setbuf(stdout, NULL);                /* Make stdout unbuffered, since we
                                               terminate child with _exit() */
       printf("%s  Parent started\n", currTime("%T"));
```

```
①       if (pipe(pfd) == -1)
            errExit("pipe");

        for (j = 1; j < argc; j++) {
②           switch (fork()) {
            case -1:
                errExit("fork %d", j);

            case 0: /* Child */
                if (close(pfd[0]) == -1)        /* Read end is unused */
                    errExit("close");

                /* Child does some work, and lets parent know it's done */

                sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
                                                /* Simulate processing */
                printf("%s  Child %d (PID=%ld) closing pipe\n",
                        currTime("%T"), j, (long) getpid());
③               if (close(pfd[1]) == -1)
                    errExit("close");

                /* Child now carries on to do other things... */

                _exit(EXIT_SUCCESS);

            default: /* Parent loops to create next child */
                break;
            }
        }

        /* Parent comes here; close write end of pipe so we can see EOF */

④       if (close(pfd[1]) == -1)                /* Write end is unused */
            errExit("close");

        /* Parent may do other work, then synchronizes with children */

⑤       if (read(pfd[0], &dummy, 1) != 0)
            fatal("parent didn't get EOF");
        printf("%s  Parent ready to go\n", currTime("%T"));

        /* Parent can now carry on to do other things... */

        exit(EXIT_SUCCESS);
    }
```

pipes/pipe_sync.c

与前面使用信号来同步相比，使用管道同步具备一个优势：它可以同来协调一个进程的动作使之与多个其他（相关）进程匹配。而多个（标准）信号无法排队的事实使得信号不适用于这种情形。（相反，信号的优势是它可以被一个进程广播到进程组中的所有成员处。）

其他同步结构也是可行的（如使用多个管道）。此外，还可以对这项技术进行扩展，即不关闭管道，每个子进程向管道写入一条包含其进程 ID 和一些状态信息的消息。或者每个子进程可以向管道写入一个字节。父进程可以计数和分析这些消息。这种方法考虑到了子进程意外终止而不是显式地关闭管道的情形。

更多资源请访问稀酷客(www.ckook.com)

44.4 使用管道连接过滤器

当管道被创建之后，为管道的两端分配的文件描述符是可用描述符中数值最小的两个。由于在通常情况下，进程已经使用了描述符 0、1 和 2，因此会为管道分配一些数值更大的描述符。那么如何形成图 44-1 中给出的情形呢，使用管道连接两个过滤器（即从 stdin 读取和写入到 stdout 的程序）使得一个程序的标准输出被定向到管道中，而另一个程序的标准输入则从管道中读取？特别是如何在不修改过滤器本身的代码的情况下完成这项工作呢？

这个问题的答案是使用在 5.5 节中介绍的技术，即复制文件描述符。一般来讲会使用下面的系列调用来获得预期的结果。

```
int pfd[2];

pipe(pfd);          /* Allocates (say) file descriptors 3 and 4 for pipe */

/* Other steps here, e.g., fork() */

close(STDOUT_FILENO);           /* Free file descriptor 1 */
dup(pfd[1]);                    /* Duplication uses lowest free file
                                   descriptor, i.e., fd 1 */
```

上面这些调用的最终结果是进程的标准输出被绑定到了管道的写入端。而对应的一组调用可以用来将进程的标准输入绑定到管道的读取端上。

注意，上面这些调用假设已经为进程打开了文件描述符 0、1 和 2。（shell 通常能够确保为它执行的每个程序都打开了这三个描述符。）如果在执行上面的调用之前文件描述符 0 已经被关闭了，那么就会错误地将进程的标准输入绑定到管道的写入端上。为避免这种情况的发生，可以使用 dup2()调用来取代对 close()和 dup()的调用，因为通过这个函数可以显式地指定被绑定到管道一端的描述符。

```
dup2(pfd[1], STDOUT_FILENO);    /* Close descriptor 1, and reopen bound
                                   to write end of pipe */
```

在复制完 pfd[1]之后就拥有两个引用管道的写入端的文件描述符了：描述符 1 和 pfd[1]。由于未使用的管道文件描述符应该被关闭，因此在 dup2()调用之后需要关闭多余的描述符。

```
close(pfd[1]);
```

前面给出的代码依赖于标准输出在之前已经被打开这个事实。假设在 pipe()调用之前，标准输入和标准输出都被关闭了。那么在这种情况下，pipe()就会给管道分配这两个描述符，即 pfd[0]的值可能为 0，pfd[1]的值可能为 1。其结果是前面的 dup2()和 close()调用将下面的代码等价。

```
dup2(1, 1);         /* Does nothing */
close(1);           /* Closes sole descriptor for write end of pipe */
```

因此按照防御性编程实践的要求最好将这些调用放在一个 if 语句中，如下所示。

```
if (pfd[1] != STDOUT_FILENO) {
    dup2(pfd[1], STDOUT_FILENO);
    close(pfd[1]);
}
```

示例程序

程序清单 44-4 使用本节介绍的技术实现了图 44-1 中给出的结构。在构建完一个管道之后，

更多资源请访问稀酷客(www.ckook.com)

这个程序创建了两个子进程。第一个子进程将其标准输出绑定到管道的写入端，然后执行 ls。第二个子进程将其标准输入绑定到管道的写入端，然后执行 wc。

程序清单 44-4：使用管道连接 ls 和 wc

pipes/pipe_ls_wc.c

```
#include <sys/wait.h>
#include "tlpi_hdr.h"
int
main(int argc, char *argv[])
{
    int pfd[2];                                 /* Pipe file descriptors */

    if (pipe(pfd) == -1)                        /* Create pipe */
        errExit("pipe");

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:             /* First child: exec 'ls' to write to pipe */
        if (close(pfd[0]) == -1)                /* Read end is unused */
            errExit("close 1");
        /* Duplicate stdout on write end of pipe; close duplicated descriptor */

        if (pfd[1] != STDOUT_FILENO) {          /* Defensive check */
            if (dup2(pfd[1], STDOUT_FILENO) == -1)
                errExit("dup2 1");
            if (close(pfd[1]) == -1)
                errExit("close 2");
        }

        execlp("ls", "ls", (char *) NULL);      /* Writes to pipe */
        errExit("execlp ls");

    default:            /* Parent falls through to create next child */
        break;
    }

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:             /* Second child: exec 'wc' to read from pipe */
        if (close(pfd[1]) == -1)                /* Write end is unused */
            errExit("close 3");

        /* Duplicate stdin on read end of pipe; close duplicated descriptor */

        if (pfd[0] != STDIN_FILENO) {           /* Defensive check */
            if (dup2(pfd[0], STDIN_FILENO) == -1)
                errExit("dup2 2");
            if (close(pfd[0]) == -1)
                errExit("close 4");
        }

        execlp("wc", "wc", "-l", (char *) NULL);    /* Reads from pipe */
        errExit("execlp wc");
```


更多资源请访问稀酷客(www.ckook.com)

```
    default:            /* Parent falls through */
        break;
    }

    /* Parent closes unused file descriptors for pipe, and waits for children */

    if (close(pfd[0]) == -1)
        errExit("close 5");
    if (close(pfd[1]) == -1)
        errExit("close 6");
    if (wait(NULL) == -1)
        errExit("wait 1");
    if (wait(NULL) == -1)
        errExit("wait 2");

    exit(EXIT_SUCCESS);
}
```

pipes/pipe_ls_wc.c

当执行程序清单 44-4 中的程序时会看到下面的输出。

```
$ ./pipe_ls_wc
     24
$ ls | wc -l                    Verify the results using shell commands
     24
```

44.5 通过管道与 shell 命令进行通信：popen()

管道的一个常见用途是执行 shell 命令并读取其输出或向其发送一些输入。popen()和 pclose()函数简化了这个任务。

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
                                    Returns file stream, or NULL on error
int pclose(FILE *stream);
                    Returns termination status of child process, or -1 on error
```

popen()函数创建了一个管道，然后创建了一个子进程来执行 shell，而 shell 又创建了一个子进程来执行 command 字符串。mode 参数是一个字符串，它确定调用进程是从管道中读取数据（mode 是 r）还是将数据写入到管道中（mode 是 w）。（由于管道是单向的，因此无法在执行的 command 中进行双向通信。）mode 的取值确定了所执行的命令的标准输出是连接到管道的写入端还是将其标准输入连接到管道的读取端，如图 44-4 所示。

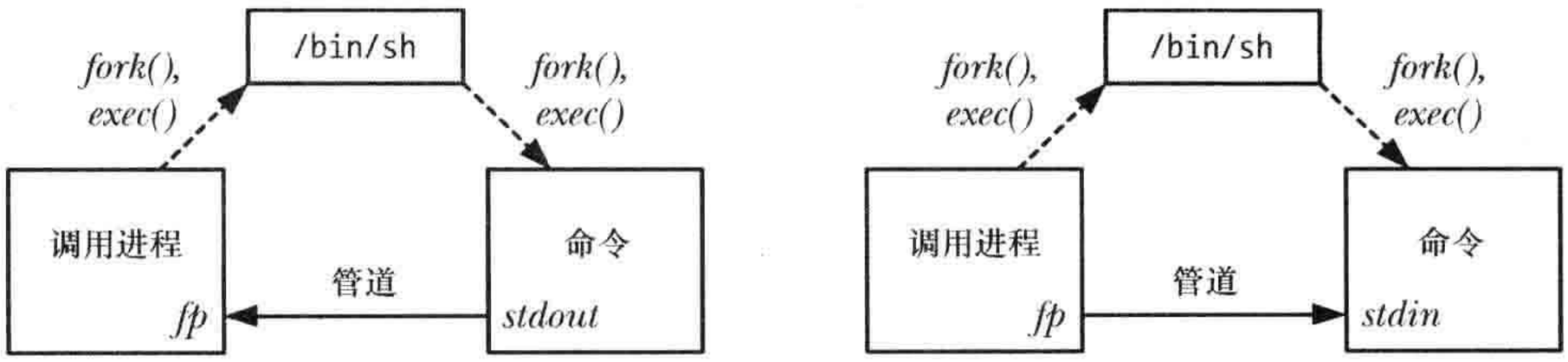


图 44-4：进程关系是 popen()中管道的使用概述

popen()在成功时会返回可供 stdio 库函数使用的文件流指针。当发生错误时（如 mode 不是 r 或 w，创建管道失败，或通过 fork()创建子进程失败），popen()会返回 NULL 并设置 errno 以标示出发生错误的原因。

在 popen()调用之后，调用进程使用管道来读取 command 的输出或使用管道向其发送输入。与使用 pipe()创建的管道一样，当从管道中读取数据时，调用进程在 command 关闭管道的写入端之后会看到文件结束；当向管道写入数据时，如果 command 已经关闭了管道的读取端，那么调用进程会收到 SIGPIPE 信号并得到 EPIPE 错误。

一旦 I/O 结束之后可以使用 pclose()函数关闭管道并等待子进程中的 shell 终止。（不应该使用 fclose()函数，因为它不会等待子进程。）pclose()在成功时会返回子进程中 shell 的终止状态（参见 26.1.3 节）（即 shell 所执行的最后一条命令的终止状态，除非 shell 是被信号杀死的）。与 system()（参见 27.6 节）一样，如果无法执行 shell，那么 pclose()会返回一个值就像是子进程中的 shell 通过调用_exit(127)来终止一样。如果发生了其他错误，那么 pclose()返回−1。其中可能发生的一个错误是无法取得终止状态，本章稍后就会介绍可能会发生这种情况的原因。

当执行等待以获取子进程中 shell 的状态时，SUSv3 要求 pclose()与 system()一样，即在内部的 waitpid()调用被一个信号处理器中断之后自动重启该调用。

一般来讲，在 27.6 节中描述的有关 system()的规范同样适用于 popen()。使用 popen()更加方便一些，它会构建管道、执行描述符复制、关闭未使用的描述符并帮助开发人员处理 fork()和 exec()的所有细节。此外，shell 处理针对的是命令。这种便捷性所牺牲的是效率，因为至少需要创建两个额外的进程：一种用于 shell，一个或多个用于 shell 执行的命令。与 system()一样，在特权进程中永远都不应该使用 popen()。

虽然 system()和 popen()以及 pclose()之间存在很多相似之处，但也存在显著的差异。这些差异源自这样一个事实，即使用 system()时 shell 命令的执行是被封装在单个函数调用中的，而使用 popen()时，调用进程是与 shell 命令并行运行的，然后会调用 pclose()。具体的差异包括以下两个方面。

- 由于调用进程与被执行的命令是并行运行的，因此 SUSv3 要求 popen()不忽略 SIGINT 和 SIGQUIT 信号。如果这些信号是从键盘产生的，那么它们会被发送到调用进程和被执行的命令中。之所以这样是因为两个进程位于同一个进程组中，而由终端产生的信号是会像 34.5 节中描述的那样被发送到（前台）进程组中的所有成员的。
- 由于调用进程在执行 popen()和执行 pclose()之间可能会创建其他子进程，因此 SUSv3 要求 popen()不能阻塞 SIGCHLD 信号。这意味着如果调用进程在 pclose()调用之前执行了一个等待操作，那么它就能够取得由 popen()创建的子进程的状态。这样当后面调用 popen()时，它就会返回−1，同时将 errno 设置为 ECHILD，表示 pclose()无法取得子进程的状态。

示例程序

程序清单 44-5 演示了 popen()和 pclose()的用法。这个程序重复读取一个文件名通配符模式②，然后使用 popen()获取将这个模式传入 ls 命令之后的结果⑤。（在较早的 UNIX 实现上会使用类似的技术执行文件名生成任务，这种技术也被称为通配 globbing，它在引入 glob()库函数之前就已经存在了。）

更多资源请访问稀酷客(www.ckook.com)

程序清单 44-5：使用 popen()通配文件名模式

pipes/popen_glob.c

```
#include <ctype.h>
#include <limits.h>
#include "print_wait_status.h"          /* For printWaitStatus() */
#include "tlpi_hdr.h"

① #define POPEN_FMT "/bin/ls -d %s 2> /dev/null"
#define PAT_SIZE 50
#define PCMD_BUF_SIZE (sizeof(POPEN_FMT) + PAT_SIZE)

int
main(int argc, char *argv[])
{
    char pat[PAT_SIZE];                 /* Pattern for globbing */
    char popenCmd[PCMD_BUF_SIZE];
    FILE *fp;                           /* File stream returned by popen() */
    Boolean badPattern;                 /* Invalid characters in 'pat'? */
    int len, status, fileCnt, j;
    char pathname[PATH_MAX];

    for (;;) {                  /* Read pattern, display results of globbing */
        printf("pattern: ");
        fflush(stdout);
②       if (fgets(pat, PAT_SIZE, stdin) == NULL)
            break;                      /* EOF */
        len = strlen(pat);
        if (len <= 1)                   /* Empty line */
            continue;

        if (pat[len - 1] == '\n')       /* Strip trailing newline */
            pat[len - 1] = '\0';

        /* Ensure that the pattern contains only valid characters,
           i.e., letters, digits, underscore, dot, and the shell
           globbing characters. (Our definition of valid is more
           restrictive than the shell, which permits other characters
           to be included in a filename if they are quoted.) */

③       for (j = 0, badPattern = FALSE; j < len && !badPattern; j++)
            if (!isalnum((unsigned char) pat[j]) &&
                    strchr("_*?[^-].", pat[j]) == NULL)
                badPattern = TRUE;

        if (badPattern) {
            printf("Bad pattern character: %c\n", pat[j - 1]);
            continue;
        }

        /* Build and execute command to glob 'pat' */

④       snprintf(popenCmd, PCMD_BUF_SIZE, POPEN_FMT, pat);
        popenCmd[PCMD_BUF_SIZE - 1] = '\0';     /* Ensure string is
                                                   null-terminated */
⑤       fp = popen(popenCmd, "r");
        if (fp == NULL) {
            printf("popen() failed\n");
            continue;
```

```
        }

        /* Read resulting list of pathnames until EOF */

        fileCnt = 0;
        while (fgets(pathname, PATH_MAX, fp) != NULL) {
            printf("%s", pathname);
            fileCnt++;
        }

        /* Close pipe, fetch and display termination status */

        status = pclose(fp);
        printf("    %d matching file%s\n", fileCnt, (fileCnt != 1) ? "s" : "");
        printf("    pclose() status == %#x\n", (unsigned int) status);
        if (status != -1)
            printWaitStatus("\t", status);
    }

    exit(EXIT_SUCCESS);
}
```

pipes/popen_glob.c

下面的 shell 会话演示了程序清单 44-5 中给出的程序的用法。在本例中首先提供了一个匹配两个文件名的模式，然后又给出了一个与任何文件名都不匹配的模式。

```
$ ./popen_glob
pattern: popen_glob*                    Matches two filenames
popen_glob
popen_glob.c
    2 matching files
    pclose() status = 0
        child exited, status=0
pattern: x*                             Matches no filename
    0 matching files
    pclose() status = 0x100             ls(1) exits with status 1
        child exited, status=1
pattern: ^D$                            Type Control-D to terminate
```

这里需要对程序清单 44-5 中通配命令的构建①④稍微解释一下。真正执行模式匹配的是 shell。ls 命令仅仅用来列出匹配的文件名，每一个行列出一个。读者可以尝试使用 echo 命令，但当模式与所有文件名都不匹配时这种做法会出现非预期的结果，然后 shell 就会保持模式不变，而 echo 会简单地打印出模式。相反，如果传递给 ls 的文件名不存在，那么它就会在 stderr（通过将 stderr 重定向到/dev/null 来丢弃写入这个描述符中的数据）上打印出一条错误消息，而不会在 stdout 上打印出任何消息，并且最后的退出状态为 1。

还需要注意程序清单 44-5 中程序所做的输入检测③。之所以这样做是为了防止非法输入引起 popen()执行一个预期之外的 shell 命令。假设忽略了这些检测，并且用户输入了下面的输入。

```
pattern: ; rm *
```

程序会将下面的命令传递给 popen()，其结果是损失惨重。

```
/bin/ls -d ; rm * 2> /dev/null
```

在使用 popen()（或 system()）执行根据用户输入构建的 shell 命令的程序中永远都需要做输入检测。（应用程序可以选择另一种方法，即将那些无需检测的字符放在引号中，这样 shell

更多资源请访问稀酷客(www.ckook.com)

就不会对那些字符进行特殊处理了。）

44.6 管道和 stdio 缓冲

由于 popen()调用返回的文件流指针没有引用一个终端，因此 stdio 库会对这种文件流应用块缓冲（参见 13.2 节）。这意味着当将 mode 的值设置为 w 来调用 popen()时，在默认情况下只有当 stdio 缓冲器被充满或使用 pclose()关闭了管道之后输出才会被发送到管道另一端的子进程。在很多情况下，这种处理方式是不存在问题的。但如果需要确保子进程能够立即从管道中接收数据，那么就需要定期调用 fflush()或使用 setbuf(fp, NULL)调用禁用 stdio 缓冲。当使用 pipe()系统调用创建管道，然后使用 fdopen()获取一个与管道的写入端对应的 stdio 流时也可以使用这项技术。

如果调用 popen()的进程正在从管道中读取数据（即 mode 是 r），那么事情就不是那么简单了。在这样情况下如果子进程正在使用 stdio 库，那么——除非它显式地调用了 fflush()或 setbuf()——其输出只有在子进程填满 stdio 缓冲器或调用了 fclose()之后才会对调用进程可用。（如果正在从使用 pipe()创建的管道中读取数据并且向另一端写入数据的进程正在使用 stdio 库，那么同样的规则也是适用的。）如果这是一个问题，那么能采取的措施就比较有限的，除非能够修改在子进程中运行的程序的源代码使之包含对 setbuf()或 fflush()调用。

如果无法修改源代码，那么可以使用伪终端来替换管道。一个伪终端是一个 IPC 通道，对进程来讲它就像是一个终端。其结果是 stdio 库会逐行输出缓冲器中的数据。第 64 章将会介绍伪终端。

44.7 FIFO

从语义上来讲，FIFO 与管道类似，它们两者之间最大的差别在于 FIFO 在文件系统中拥有一个名称，并且其打开方式与打开一个普通文件是一样的。这样就能够将 FIFO 用于非相关进程之间的通信（如客户端和服务器）。

一旦打开了 FIFO，就能在它上面使用与操作管道和其他文件的系统调用一样的 I/O 系统调用了（如 read()、write()和 close()）。与管道一样，FIFO 也有一个写入端和读取端，并且从管道中读取数据的顺序与写入的顺序是一样的。FIFO 的名称也由此而来：先入先出。FIFO 有时候也被称为命名管道。

与管道一样，当所有引用 FIFO 的描述符都被关闭之后，所有未被读取的数据会被丢弃。

使用 mkfifo 命令可以在 shell 中创建一个 FIFO。

```
$ mkfifo [ -m mode ] pathname
```

pathname 是创建的 FIFO 的名称，–m 选项用来指定权限 mode，其工作方式与 chmod 命令一样。

当在 FIFO（或管道）上调用 fstat()和 stat()函数时它们会在 stat 结构的 st_mode 字段中返回一个类型为 S_IFIFO 的文件（参见 15.1 节）。当使用 ls –l 列出文件时，FIFO 文件在第一列的类型为 p，ls –F 会在 FIFO 路径名后面附加上一个管道符（|）。

mkfifo()函数创建一个名为 pathname 的全新的 FIFO。

更多资源请访问稀酷客(www.ckook.com)

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

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

Returns 0 on success, or −1 on error

mode 参数指定了新 FIFO 的权限。这些权限是通过将表 15-4 中的常量取 OR 来指定的。与往常一样，这些权限会按照进程的 umask 值（参见 15.4.6 节）来取掩码。

以前创建 FIFO 使用的是 mknod(pathname,S_IFIFO, 0)系统调用。POSIX.1-1990 规定了 mkfifo()，它更加简单，并且消除了 mknod()具备的通用性，这种通用性允许创建各种类型的文件，包括设备文件。（SUSv3 规定了 mknod()，但并没有详细规定，它只定义了这个函数的用途是创建 FIFO。）大多数 UNIX 实现提供了 mkfifo()，它是构建于 mknod()之上的一个库函数。

一旦 FIFO 被创建，任何进程都能够打开它，只要它能够通过常规的文件权限检测（参见 15.4.3 节）。

打开一个 FIFO 具备一些不寻常的语义。一般来讲，使用 FIFO 时唯一明智的做法是在两端分别设置一个读取进程和一个写入进程。这样在默认情况下，打开一个 FIFO 以便读取数据（open() O_RDONLY 标记）将会阻塞直到另一个进程打开 FIFO 以写入数据（open() O_WRONLY 标记）为止。相应地，打开一个 FIFO 以写入数据将会阻塞直到另一个进程打开 FIFO 以读取数据为止。换句话说，打开一个 FIFO 会同步读取进程和写入进程。如果一个 FIFO 的另一端已经打开（可能是因为一对进程已经打开了 FIFO 的两端），那么 open()调用会立即成功。

在大多数 UNIX 实现（包括 Linux）上，当打开一个 FIFO 时可以通过指定 O_RDWR 标记来绕过打开 FIFO 时的阻塞行为。这样，open()就会立即返回，但无法使用返回的文件描述符在 FIFO 上读取和写入数据。这种做法破坏了 FIFO 的 I/O 模型，SUSv3 明确指出以 O_RDWR 标记打开一个 FIFO 的结果是未知的，因此出于可移植性的原因，开发人员不应该使用这项技术。对于那些需要避免在打开 FIFO 时发生阻塞的需求，open()的 O_NONBLOCK 标记提供了一种标准化的方法来完成这个任务（参见 44.9 节）。

在打开一个 FIFO 时避免使用 O_RDWR 标记还有另外一个原因。当采用那种方式调用 open()之后，调用进程在从返回的文件描述符中读取数据时永远都不会看到文件结束，因为永远都至少存在一个文件描述符被打开着以等待数据被写入 FIFO，即进程从中读取数据的那个描述符。

使用 FIFO 和 tee(1)创建双重管道线

shell 管道线的其中一个特征是它们是线性的，管道线中的每个进程都读取前一个进程产生的数据并将数据发送到其后一个进程中。使用 FIFO 就能够在管道线中创建子进程，这样除了将一个进程的输出发送给管道线中的后面一个进程之外，还可以复制进程的输出并将数据发送到另一个进程中。要完成这个任务需要使用 tee 命令，它将其从标准输入中读取到的数据复制两份并输出：一份写入到标准输出，另一份写入到通过命令行参数指定的文件中。

更多资源请访问稀酷客(www.ckook.com)

将传给tee命名的file参数设置为一个FIFO可以让两个进程同时读取tee产生的两份数据。下面的 shell 会话演示了这种用法，它创建了一个名为 myfifo 的 FIFO，然后在后台启动一个 wc 命令，该命令会打开 FIFO 以读取数据（这个操作会阻塞直到有进程打开 FIFO 写入数据为止），接着执行一条管道线将 ls 的输出发送给 tee，tee 会将输出传递给管道线中的下一个命令 sort，同时还会将输出发送给名为 myfifo 的 FIFO。（sort 的–k5n 选项会导致 ls 的输出按照第五个以空格分隔的字段的数值升序排序。）

```
$ mkfifo myfifo
$ wc -l < myfifo &
$ ls -l | tee myfifo | sort -k5n
(Resulting output not shown)
```

从图表上来看，上面的命令创建了图 44-5 中给出的情形。

tee 程序之所以这样命名是因为其外形。可以将 tee 看成是功能与管道类似的一个实体，但它存在一个额外的分支发送一份输出的副本。从图表上来看，其形状像是一个大写字母 T（参见图 44-5）。除了上面描述的用途之外，tee 对于管道线调试和保存复杂管道线中某些中间节点的输出结果也是非常有用的。

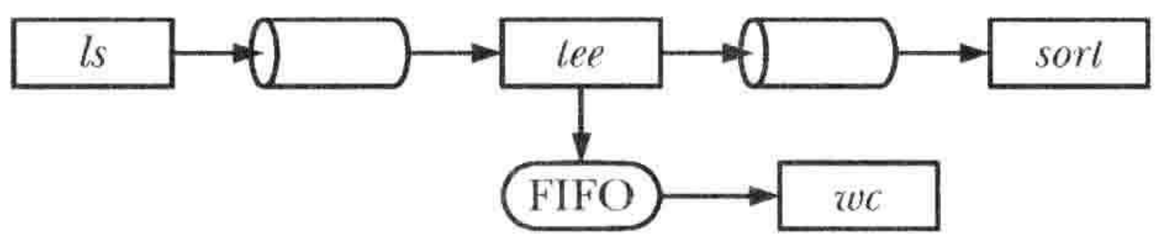


图 44-5：使用 FIFO 和 tee(1)创建双重管道线

44.8 使用管道实现一个客户端/服务器应用程序

本节将介绍一个简单的使用 FIFO 进行 IPC 的客户端/服务器应用程序。服务器提供的（简单）服务是向每个发送请求的客户端赋一个唯一的顺序数字。在对这个应用程序进行讨论的过程中将会介绍与服务器设计有关的一些概念和技术。

应用程序概述

在这个示例应用程序中，所有客户端使用一个服务器 FIFO 来向服务器发送请求。头文件（程序清单 44-6）定义了众所周知的名称（/tmp/seqnum_sv），服务器的 FIFO 将使用这个名称。这个名称是固定的，因此所有客户端知道如何联系到服务器。（在这个示例应用程序中将会在 /tmp 目录中创建 FIFO，这样在大多数系统上都能够在不修改程序的情况下方便地运行这个程序。但正如在 38.7 节中指出的那样，在一个像/tmp 这样的公共可写的目录中创建文件可能会导致各种安全隐患，因此现实世界中的应用程序不应该使用这种目录。）

在客户端-服务器应用程序中将会不断地碰到一个概念，即服务器用来使服务对客户端可见的众所周知的地址或名称。对于客户端如何知道在何处联系服务器这个问题来讲，使用众所周知的地址是一种解决方案。另一种可能的解决方案是提供某种名称服务器，服务器可以将它们的服务的名称注册到名称服务器上。然后每个客户端联系名称服务器以获取服务的位置。这个解决方案允许灵活地配置服务器的位置，而付出的代价则是需要进行额外的编程。当然，客户端和服务器需要知道到何处联系名称服务器，它位于一个众所周知的地址。

更多资源请访问稀酷客(www.ckook.com)

无法使用单个 FIFO 向所有客户端发送响应，因为多个客户端在从 FIFO 中读取数据时会相互竞争，这样就可能会出现各个客户端读取到了其他客户端的响应消息，而不是自己的响应消息。因此每个客户端需要创建一个唯一的 FIFO，服务器使用这个 FIFO 来向该客户端递送响应，并且服务器需要知道如何找出各个客户端的 FIFO。解决这个问题的一种方式是让客户端生成自己的 FIFO 路径名，然后将路径名作为请求消息的一部分传递给服务器。或者客户端和服务器可以约定一个构建客户端 FIFO 路径名的规则，然后客户端可以将构建自己的路径名所需的相关信息作为请求的一部分发送给服务器。本例中将会使用后面一种解决方案。每个客户端的 FIFO 是从一个由包含客户端的进程 ID 的路径名构成的模板（CLIENT_FIFO_TEMPLATE）中构建而来的。在生成过程中包含进程 ID 可以很容易地产生一个对各个客户端唯一的名称。

图 44-6 展示了这个应用程序如何使用 FIFO 来完成客户端和服务器进程之间的通信。

头文件（程序清单 44-6）定义了客户端发送给服务器的请求消息的格式和服务器发送给客户端的响应消息的格式。

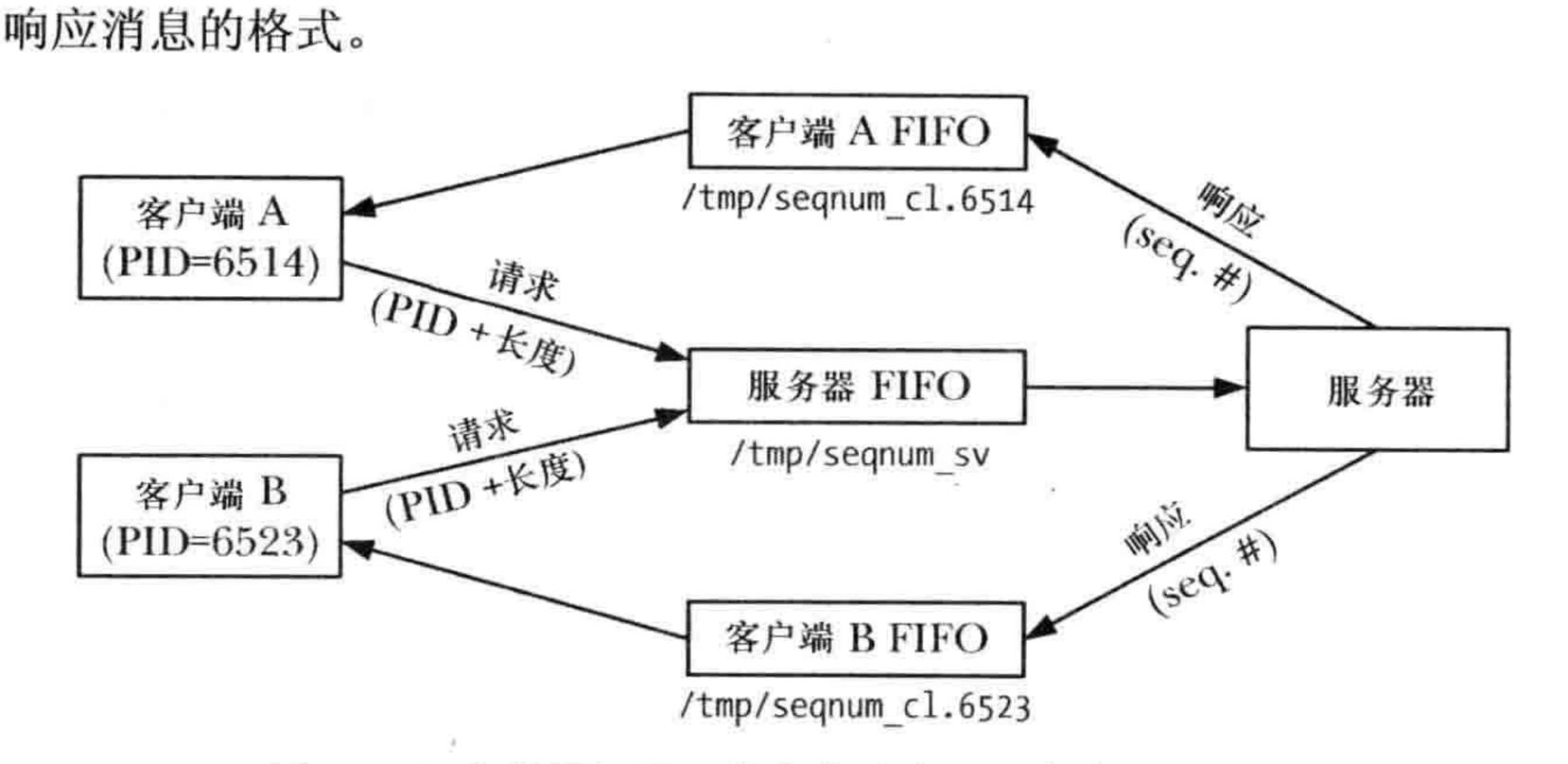


图 44-6：在单服务器、多客户端应用程序中使用 FIFO

记住管道和 FIFO 中的数据是字节流，消息之间是没有边界的。这意味着当多条消息被递送到一个进程中时，如本例中的服务器，发送者和接收者必须要约定某种规则来分隔消息。这可以使用多种方法。

- 每条消息使用诸如换行符之类的分隔字符结束。（使用这项技术的一个例子是程序清单 59-1 中的 readLine()函数。）这样就必须要保证分隔字符不会出现在消息中或者当它出现在消息中时必须要采用某种规则进行转义。例如，如果使用换行符作为分隔符，那么字符\加上换行可以用来表示消息中一个真实的换行符，而\\则可以用来表示一个真实的\。这种方法的一个不足之处是读取消息的进程在从 FIFO 中扫描数据时必须要逐个字节地分析直到找到分隔符为止。
- 在每条消息中包含一个大小固定的头，头中包含一个表示消息长度的字段，该字段指定了消息中剩余部分的长度。这样读取进程就需要首先从 FIFO 中读取头，然后使用头中的长度字段来确定需读取的消息中剩余部分的字节数。这种方法能够高效地读取任意大小的消息，但一旦不合规则（如错误的 length 字段）的消息被写入到管道中之后问题就出来了。
- 使用固定长度的消息并让服务器总是读取这个大小固定的消息。这种方法的优势在于简单性。但它对消息的大小设置了一个上限，意味着会浪费一些通道容量（因为需要对较短的消息进行填充以满足固定长度）。此外，如果其中一个客户端意外地或故意发送了一条长度不对的消息，那么所有后续的消息都会出现步调不一致的情况，并且

更多资源请访问稀酷客(www.ckook.com)

在这种情况下服务器是难以恢复的。

图 44-7 展示了这三种技术。注意不管使用这三种技术中的哪种，每条消息的总长度必须要小于 PIPE_BUF 字节以防止内核对消息进行拆分，从而造成与其他写者发送的消息错乱的情况的发生。

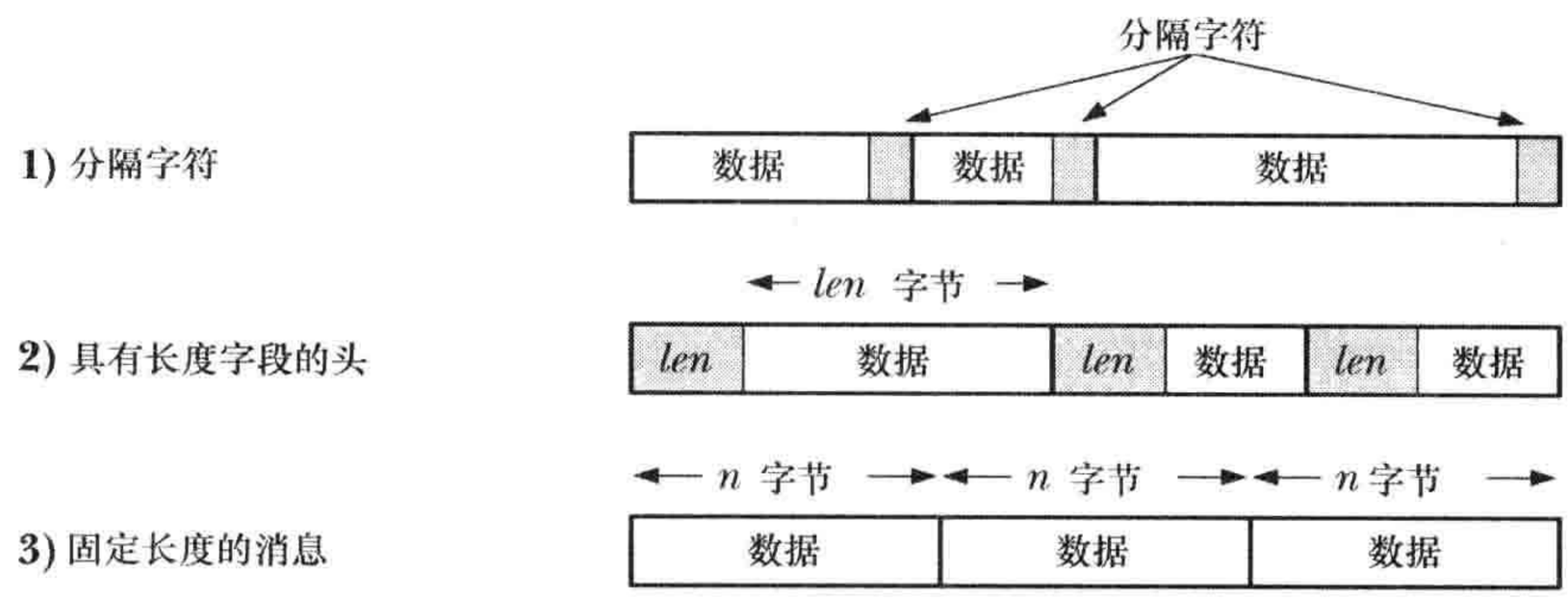


图 44-7：分隔字节流中的消息

> 在正文描述的三种技术中，所有客户端发送的所有消息都会被放在一个通道（FIFO）中。另一种方法是为每条消息使用一个连接。发送者打开通信通道，发送消息，然后关闭通道。读取进程在碰到文件结束时就知道达到消息结尾了。如果多个写者都打开了一个 FIFO，那么这种方法就不可行了，因为读取在其中一个写者关闭 FIFO 之后不会看到文件结束。但当使用流 socket 时这种方法就变得可行了，因为服务器进程会为每个进入的客户端连接创建一个唯一的通信通道。

在本章的示例应用程序中将使用上面介绍的第三种技术，即每个客户端向服务器发送的消息的长度是固定的。程序清单 44-6 中的 request 结构定义了消息。每个发送给服务器的请求都包含了客户端的进程 ID，这样服务器就能够构建客户端用来接收响应的 FIFO 的名称了。请求中还包含了一个 seqLen 字段，它指定了应该为这个客户端分配的序号的数量。服务器向客户端发送的响应消息由一个字段 seqNum 构成，它是为这个客户端分配的一组序号的起始值。

程序清单 44-6：fifo_seqnum_server.c 和 fifo_seqnum_client.c 的头文件

pipes/fifo_seqnum.h

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

#define SERVER_FIFO "/tmp/seqnum_sv"
                                /* Well-known name for server's FIFO */
#define CLIENT_FIFO_TEMPLATE "/tmp/seqnum_cl.%ld"
                                /* Template for building client FIFO name */
#define CLIENT_FIFO_NAME_LEN (sizeof(CLIENT_FIFO_TEMPLATE) + 20)
                                /* Space required for client FIFO pathname
                                   (+20 as a generous allowance for the PID) */

struct request {                /* Request (client --> server) */
    pid_t pid;                  /* PID of client */
    int seqLen;                 /* Length of desired sequence */
};
```

```
struct response {                /* Response (server --> client) */
    int seqNum;                  /* Start of sequence */
};
```

pipes/fifo_seqnum.h

服务器程序

程序清单 44-7 是服务器的代码。这个服务器按序完成了下面的工作。

- 创建服务器的众所周知的 FIFO①并打开 FIFO 以便读取②。服务器必须要在客户端之前运行，这样服务器 FIFO 在客户端试图打开它之前就已经存在了。服务器的 open()调用将会阻塞直到第一个客户端打开了服务器的 FIFO 的另一端以写入数据为止。
- 再次打开服务器的 FIFO③，这次是为了写入数据。这个调用永远不会被阻塞，因为之前已经因需读取而打开 FIFO 了。第二个打开操作是为了确保服务器在所有客户端关闭了 FIFO 的写入端之后不会看到文件结束。
- 忽略 SIGPIPE 信号④，这样如果服务器试图向一个没有读者的客户端 FIFO 写入数据时不会收到 SIGPIPE 信号（默认会杀死进程），而是会从 write()系统调用中收到一个 EPIPE 错误。
- 进入一个循环从每个进入的客户端请求中读取数据并响应⑤。要发送响应，服务器需要构建客户端 FIFO 的名称⑥，然后打开这个 FIFO⑦。
- 如果服务器在打开客户端 FIFO 时发生了错误，那么就丢弃那个客户端的请求⑧。

这是一种迭代式服务器，这种服务器会在读取和处理完当前客户端之后才会去处理下一个客户端。当每个客户端请求的处理和响应都能够快速完成时采用这种迭代式服务器设计是合理的，因为不会对其他客户端请求的处理产生延迟。另一种设计方法是并发式服务器，在这种设计中主服务器进程使用单独的子进程（或线程）来处理各个客户端的请求。第 60 章将会深入介绍服务器设计。

程序清单 44-7：使用 FIFO 的迭代式服务器

pipes/fifo_seqnum_server.c

```
#include <signal.h>
#include "fifo_seqnum.h"

int
main(int argc, char *argv[])
{
    int serverFd, dummyFd, clientFd;
    char clientFifo[CLIENT_FIFO_NAME_LEN];
    struct request req;
    struct response resp;
    int seqNum = 0;                     /* This is our "service" */
       /* Create well-known FIFO, and open it for reading */

    umask(0);                           /* So we get the permissions we want */
①   if (mkfifo(SERVER_FIFO, S_IRUSR | S_IWUSR | S_IWGRP) == -1
            && errno != EEXIST)
        errExit("mkfifo %s", SERVER_FIFO);
②   serverFd = open(SERVER_FIFO, O_RDONLY);
    if (serverFd == -1)
        errExit("open %s", SERVER_FIFO);

    /* Open an extra write descriptor, so that we never see EOF */

③   dummyFd = open(SERVER_FIFO, O_WRONLY);
```


更多资源请访问稀酷客(www.ckook.com)

```
    if (dummyFd == -1)
        errExit("open %s", SERVER_FIFO);

④   if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
        errExit("signal");

⑤   for (;;) {                          /* Read requests and send responses */
        if (read(serverFd, &req, sizeof(struct request))
                != sizeof(struct request)) {
            fprintf(stderr, "Error reading request; discarding\n");
            continue;                   /* Either partial read or error */
        }

        /* Open client FIFO (previously created by client) */

⑥       snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
                (long) req.pid);
⑦       clientFd = open(clientFifo, O_WRONLY);
        if (clientFd == -1) {           /* Open failed, give up on client */
            errMsg("open %s", clientFifo);
⑧           continue;
        }

        /* Send response and close FIFO */

        resp.seqNum = seqNum;
        if (write(clientFd, &resp, sizeof(struct response))
                != sizeof(struct response))
            fprintf(stderr, "Error writing to FIFO %s\n", clientFifo);
        if (close(clientFd) == -1)
            errMsg("close");

        seqNum += req.seqLen;           /* Update our sequence number */
    }
}
```

pipes/fifo_seqnum_server.c

客户端程序

程序清单 44-8 是客户端的代码。客户端按序完成了下面的工作。

- 创建一个 FIFO 以从服务器接收响应②。这项工作是在发送请求之前完成的，这样才能确保 FIFO 在服务器试图打开它并向其发送响应消息之前就已经存在了。
- 构建一条发给服务器的消息，消息中包含了客户端的进程 ID 和一个指定了客户端希望服务器赋给它的序号长度的数字（从可选的命令行参数中获取）④。（如果没有提供命令行参数，那么默认的序号长度是 1。）
- 打开服务器 FIFO⑤并将消息发送给服务器⑥。
- 打开客户端 FIFO⑦，然后读取和打印服务器的响应⑧。

另一个需要注意的地方是通过 atexit()③建立的退出处理器①，它确保了当进程退出之后客户端的 FIFO 会被删除。或者可以在客户端 FIFO 的 open()调用之后立即调用 unlink()。在那个时刻这种做法是能够正常工作的，因为它们都执行了阻塞的 open()调用，服务器和客户端各自持有了 FIFO 的打开着的文件描述符，而从文件系统中删除 FIFO 名称不会对这些描述符以及它们所引用的打开着的文件描述符产生影响。

下面是运行这个客户端和服务器程序时看到的输出。

更多资源请访问稀酷客(www.ckook.com)

```
$ ./fifo_seqnum_server &
[1] 5066
$ ./fifo_seqnum_client 3                    Request a sequence of three numbers
0                                           Assigned sequence begins at 0
$ ./fifo_seqnum_client 2                    Request a sequence of two numbers
3                                           Assigned sequence begins at 3
$ ./fifo_seqnum_client                      Request a single number
5
```

程序清单 44-8：序号服务器的客户端

pipes/fifo_seqnum_client.c

```
    #include "fifo_seqnum.h"

    static char clientFifo[CLIENT_FIFO_NAME_LEN];

    static void             /* Invoked on exit to delete client FIFO */
① removeFifo(void)
    {
        unlink(clientFifo);
    }

    int
    main(int argc, char *argv[])
    {
        int serverFd, clientFd;
        struct request req;
        struct response resp;
        if (argc > 1 && strcmp(argv[1], "--help") == 0)
            usageErr("%s [seq-len...]\n", argv[0]);

        /* Create our FIFO (before sending request, to avoid a race) */

        umask(0);                   /* So we get the permissions we want */
②       snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
                (long) getpid());
        if (mkfifo(clientFifo, S_IRUSR | S_IWUSR | S_IWGRP) == -1
                    && errno != EEXIST)
            errExit("mkfifo %s", clientFifo);

③       if (atexit(removeFifo) != 0)
            errExit("atexit");

        /* Construct request message, open server FIFO, and send request */

④       req.pid = getpid();
        req.seqLen = (argc > 1) ? getInt(argv[1], GN_GT_0, "seq-len") : 1;

⑤       serverFd = open(SERVER_FIFO, O_WRONLY);
        if (serverFd == -1)
            errExit("open %s", SERVER_FIFO);

⑥       if (write(serverFd, &req, sizeof(struct request)) !=
                sizeof(struct request))
            fatal("Can't write to server");

        /* Open our FIFO, read and display response */

⑦       clientFd = open(clientFifo, O_RDONLY);
        if (clientFd == -1)
```

```
        errExit("open %s", clientFifo);

⑧   if (read(clientFd, &resp, sizeof(struct response))
            != sizeof(struct response))
        fatal("Can't read response from server");

    printf("%d\n", resp.seqNum);
    exit(EXIT_SUCCESS);
}
```

pipes/fifo_seqnum_client.c

44.9 非阻塞 I/O

前面曾经提过当一个进程打开一个 FIFO 的一端时，如果 FIFO 的另一端还没有被打开，那么该进程会被阻塞。但有些时候阻塞并不是期望的行为，而这可以通过在调用 open()时指定 O_NONBLOCK 标记来实现。

```
fd = open("fifopath", O_RDONLY | O_NONBLOCK);
if (fd == -1)
    errExit("open");
```

如果 FIFO 的另一端已经被打开，那么 O_NONBLOCK 对 open()调用不会产生任何影响——它会像往常一样立即成功地打开 FIFO。只有当 FIFO 的另一端还没有被打开的时候 O_NONBLOCK 标记才会起作用，而具体产生的影响则依赖于打开 FIFO 是用于读取还是用于写入的。

- 如果打开 FIFO 是为了读取，并且 FIFO 的写入端当前已经被打开，那么 open()调用会立即成功（就像 FIFO 的另一端已经被打开一样）。
- 如果打开 FIFO 是为了写入，并且还没有打开 FIFO 的另一端来读取数据，那么 open()调用会失败，并将 errno 设置为 ENXIO。

为读取而打开 FIFO 和为写入而打开 FIFO 时 O_NONBLOCK 标记所起的作用不同是有原因的。当 FIFO 的另一个端没有写者时打开一个 FIFO 以便读取数据是没有问题的，因为任何试图从 FIFO 读取数据的操作都不会返回任何数据。但当试图向没有读者的 FIFO 中写入数据时将会导致 SIGPIPE 信号的产生以及 write()返回 EPIPE 错误。

表 44-1 对打开 FIFO 的语义进行了总结，包括上面介绍的 O_NONBLOCK 标记的作用。

表 44-1：在 FIFO 上调用 open()的语义

open()类型		open()的结果	
打开的目的	额外标记	FIFO 另一端的打开操作	FIFO 另一端的关闭操作
读取	无（阻塞）	立即成功	阻塞
	O_NONBLOCK	立即成功	立即成功
写入	无（阻塞）	立即成功	阻塞
	O_NONBLOCK	立即成功	失败（ENXIO）

在打开一个 FIFO 时使用 O_NONBLOCK 标记存在两个目的。

- 它允许单个进程打开一个 FIFO 的两端。这个进程首先会在打开 FIFO 时指定 O_NONBLOCK 标记以便读取数据，接着打开 FIFO 以便写入数据。

更多资源请访问稀酷客(www.ckook.com)

- 它防止打开两个 FIFO 的进程之间产生死锁。

当两个或多个进程中每个进程都因等待对方完成某个动作而阻塞时会产生死锁。图 44-8 给出了两个进程发生死锁的情形。各个进程都因等待打开一个 FIFO 以便读取数据而阻塞。如果各个进策划那个都可以执行其第二个步骤（打开另一个 FIFO 以便写入数据）的话就不会发生阻塞。这个特定的死锁问题是通过颠倒进程 Y 中的步骤 1 和步骤 2 并保持进程 X 中两个步骤的顺序不变来解决，反之亦然。但在一些应用程序中进行这样的调整可能并不容易。相反，可以通过在为读取而打开 FIFO 时让其中一个进程或两个进程都指定 O_NONBLOCK 标记来解决这个问题。

进程 X	进程 Y
1. 打开 FIFO A 准备读取块	1. 打开 FIFO B 准备读取块
2. 打开 FIFO B 准备写入	2. 打开 FIFO A 准备写入

图 44-8：打开两个 FIFO 的进程之间的死锁

非阻塞 read()和 write()

O_NONBLOCK 标记不仅会影响 open()的语义，而且还会影响——因为在打开的文件描述中这个标记仍然被设置着——后续的 read()和 write()调用的语义。下一节将会对这些影响进行描述。

有些时候需要修改一个已经打开的 FIFO（或另一种类型的文件）的 O_NONBLOCK 标记的状态，具体存在这个需求的场景包括以下几种。

- 使用 O_NONBLOCK 打开了一个 FIFO 但需要让后续的 read()和 write()调用在阻塞模式下运作。
- 需要启用从 pipe()返回的一个文件描述符的非阻塞模式。更一般地，可能需要更改从除 open()调用之外的其他调用中——如每个由 shell 运行的新程序中自动被打开的三个标准描述符的其中一个或 socket()返回的文件描述符——取得的任意文件描述符的非阻塞状态。
- 出于一些应用程序的特殊需求，需要切换一个文件描述符的 O_NONBLOCK 设置的开启和关闭状态。

当碰到上面的需求时可以使用 fcntl()启用或禁用打开着的文件的 O_NONBLOCK 状态标记。通过下面的代码（忽略的错误检查）可以启用这个标记。

```
int flags;

flags = fcntl(fd, F_GETFL);          /* Fetch open files status flags */
flags |= O_NONBLOCK;                 /* Enable O_NONBLOCK bit */
fcntl(fd, F_SETFL, flags);           /* Update open files status flags */
```

通过下面的代码可以禁用这个标记。

```
flags = fcntl(fd, F_GETFL);
flags &= ~O_NONBLOCK;                /* Disable O_NONBLOCK bit */
fcntl(fd, F_SETFL, flags);
```

44.10 管道和 FIFO 中 read()和 write()的语义

表 44-2 对管道和 FIFO 上的 read()操作进行了总结，包括 O_NONBLOC 标记的作用。

更多资源请访问稀酷客(www.ckook.com)

表 44-2：从一个包含 p 字节的管道或 FIFO 中读取 n 字节的语义

是否启用 O_NONBLOCK	管道或 FIFO 中可用的数据字节(p)			
	p = 0，写入端打开	p = 0，写入端关闭	p < n	p >= n
否	阻塞	返回 0（EOF）	读取 p 字节	读取 n 字节
是	失败（EAGAIN）	返回 0（EOF）	读取 p 字节	读取 n 字节

只有当没有数据并且写入端没有被打开时阻塞和非阻塞读取之间才存在差别。在这种情况下，普通的 read()会被阻塞，而非阻塞 read()会失败并返回 EAGAIN 错误。

当 O_NONBLOCK 标记与 PIPE_BUF 限制共同起作用时 O_NONBLOCK 标记对象管道或 FIFO 写入数据的影响会变得复杂。表 44-3 对 write()的行为进行了总结。

表 44-3：向一个管道或 FIFO 写入 n 字节的语义

是否启用 O_NONBLOCK	读取端打开		读取端关闭
	n <= PIPE_BUF	n > PIPE_BUF	
否	原子地写入 n 字节；可能阻塞，直到足够的数据被读取以便继续执行 write()	写入 n 字节；可能阻塞，直到足够的数据被读取以便结束 write()；数据可能会与其他进程写入的数据发生交叉	SIGPIPE + EPIPE
是	如果空间足以立即写入 n 字节，那么 write()会原子地成功；否则就失败（EAGAIN）	如果空间足以写入一些字节，那么写入的字节数在 1 到 n 之间（可能会与其他进程写入的数据发生交叉）；否则 write()会失败（EAGAIN）	

当数据无法立即被传输时 O_NONBLOCK 标记会导致在一个管道或 FIFO 上的 write()失败（错误是 EAGAIN）。这意味着当写入了 PIPE_BUF 字节之后，如果在管道或 FIFO 中没有足够的空间了，那么 write()会失败，因为内核无法立即完成这个操作并且无法执行部分写入，否则就会破坏不超过 PIPE_BUF 字节的写入操作的原子性的要求。

当一次写入的数据量超过 PIPE_BUF 字节时，该写入操作无需是原子的。因此，write()会尽可能多地传输字节（部分写）以充满管道或 FIFO。在这种情况下，从 write()返回的值是实际传输的字节数，并且调用者随后必须要进行重试以写入剩余的字节。但如果管道或 FIFO 已经满了，从而导致哪怕连一个字节都无法传输了，那么 write()会失败并返回 EAGAIN 错误。

44.11 总结

管道是 UNIX 系统上出现的第一种 IPC 方法，shell 以及其他应用程序经常会使用管道。管道是一个单项、容量有限的字节流，它可以用于相关进程之间的通信。尽管写入管道的数据块的大小可以是任意的，但只有那些写入的数据量不超过 PIPE_BUF 字节的写入操作才被确保是原子的。除了是一种 IPC 方法之外，管道还可以用于进程同步。

在使用管道时必须要小心地关闭未使用的描述符以确保读取进程能够检测到文件结束和写入进程能够收到 SIGPIPE 信号或 EPIPE 错误。（通常，最简单的做法是让向管道写入数据的应用程序忽略 SIGPIPE 并通过 EPIPE 错误检测管道是否“坏了”。）

popen()和 pclose()函数允许一个程序向一个标准 shell 命令传输数据或从中读取数据，而

更多资源请访问稀酷客(www.ckook.com)

无需处理创建管道、执行 shell 以及关闭未使用的文件描述符的细节。

FIFO 除了 mkfifo()创建和在文件系统中存在一个名称以及可以被拥有合适的权限的任意进程打开之外，其运作方式与管道完全一样。在默认情况下，为读取数据而打开一个 FIFO 会被阻塞直到另一个进程为写入数据而打开了该 FIFO，反之亦然。

本章讨论了几个相关的主题。首先介绍了如何复制文件描述符使得一个过滤器的标准输入或输出可以被绑定到一个管道上。在介绍使用 FIFO 构建一个客户端-服务器的例子中介绍了几个与客户端-服务器设计相关的主题，包括为服务器使用一个众所周知的地址以及迭代式服务器设计和并发服务器设计之间的对比。在开发示例 FIFO 应用程序时提到尽管通过管道传输的数据是一个字节流，但有时候将数据打包成消息对于通信来讲也是有用的，并且介绍了几种将数据打包成消息的方法。

最后介绍了在打开一个 FIFO 并执行 I/O 时 O_NONBLOCK 标记（非阻塞 I/O）的影响。O_NONBLOCK 标记对于在打开 FIFO 时不希望阻塞来讲是有用的，同时对读取操作在没有数据可用时不阻塞或在写入操作在管道或 FIFO 没有足够的空间时不阻塞也是有用的。

更多信息

[Bach, 1986]和[Bovet & Cesati, 2005]讨论了管道的实现。有关管道和 FIFO 的有用细节还可以在[Vahalia, 1996]中找到。

44.12 习题

44-1. 编写一个程序使之使用两个管道来启用父进程和子进程之间的双向通信。父进程应该循环从标准输入中读取一个文本块并使用其中一个管道将文本发送给子进程，子进程将文本转换成大写并通过另一个管道将其传回给父进程。父进程读取从子进程过来的数据并在继续下一个循环之前将其反馈到标准输出上。

44-2. 实现 popen()和 pclose()。尽管这些函数因无需完成在 system()实现（参见 27.7 节）中的信号处理而得到了简化，但需要小心地将管道两端正确绑定到各个进程的文件流上并确保关闭所有引用管道两端的未使用的描述符。由于通过多个 popen()调用创建的子进程可能会同时运行，因此需要需要维护一个将 popen()分配的文件流与相应的子进程 ID 关联起来的数据结构。（如果使用数组，那么可以将 fileno()函数的返回值作为数组的下标，这个函数能够取得与一个文件流对应的文件描述符。）从这个结构中取得正确的进程 ID 使得 pclose()能够选择需等待的子进程。这个结构还满足了 SUSv3 的要求，即在新的子进程中必须要关闭所有通过之前的 popen()调用仍然打开着的文件流。

44-3. 程序清单 44-7 中的服务器（fifo_seqnum_server.c）每次在启动时都会从序号 0 开始赋序号值。修改程序使它使用一个在每次赋序号时都会更新的备份文件。（在 4.3.1 节中介绍的 open() O_SYNC 标记可能会有用。）在启动时，程序应该检查这个文件是否存在，如果存在的话就使用其中包含的值来初始化序号。如果在启动时没有找到备份文件，那么程序应该创建一个新文件并从 0 开始赋序号。（另一种方法是使用在 49 章中介绍的内存映射文件。）

44-4. 在程序清单 44-7 中的服务器（fifo_seqnum_server.c）中添加代码使它在收到 SIGINT

更多资源请访问稀酷客(www.ckook.com)

或 SIGTERM 信号时删除服务器 FIFO 并终止。

44-5. 程序清单 44-7 中的服务器（fifo_seqnum_server.c）在 FIFO 上执行第二次带 O_WRONLY 标记的打开操作使之在从 FIFO 的读取描述符（serverFd）中读取数据时永远不会看到文件结束。除了这种做法之外，还可以尝试另一种方法：当服务器在读取描述符中看到文件结束时关闭这个描述符，然后再次打开 FIFO 以便读取数据。（这个打开操作将会阻塞直到下一个客户端因写入而打开 FIFO 为止。）这种方法错在哪里了？

44-6. 程序清单 44-7 中的服务器（fifo_seqnum_server.c）假设客户端进程的行为是正常的。如果一个行为异常的客户端创建了一个客户端 FIFO 和向服务器发送了一个请求，但并没有打开其 FIFO，那么服务器在打开客户端的 FIFO 时将会被阻塞，从而造成其他客户端的请求被无限延迟。（如果是恶意的，那么就可以认定为 DoS 攻击。）设计一个模型来解决这个问题，对服务器（可能还要加上程序清单 44-8 中的客户端）进行相应的扩展。

44-7. 编写程序验证 FIFO 上非阻塞打开和非阻塞 I/O 的操作（参见 44.9 节）。

更多资源请访问稀酷客(www.ckook.com)

第45章

System V IPC 介绍

System V IPC 包括三种不同的进程间通信机制。

- 消息队列用来在进程之间传递消息。消息队列与管道有点像，但存在两个重大差别。第一是消息队列是存在边界的，这样读者和写者之间以消息进行通信，而不是通过无分隔符的字节流进行通信的。第二是每条消息包括一个整型的 type 字段，并且可以通过类型类选择消息而无需以消息被写入的顺序来读取消息。
- 信号量允许多个进程同步它们的动作。一个信号量是一个由内核维护的整数值，它对所有具备相应权限的进程可见。一个进程通过对信号量的值进行相应的修改来通知其他进程它正在执行某个动作。
- 共享内存使得多个进程能够共享内存（即同被映射到多个进程的虚拟内存的页帧）的同一块区域（称为一个段）。由于访问用户空间内存的操作是非常快的，因此共享内存是其中一种速度最快的 IPC 方法：一个进程一旦更新了共享内存，那么这个变更会立即对共享同一个内存段的其他进程可见。

这三种 IPC 机制在功能上存在着很大的差异，但把它们放在一起讨论是有原因的。其中一个原因是它们是一同被开发出来的，它们在 20 世纪 70 年代后期首次出现在了 Columbus UNIX 系统中。这是 Bell 内部实现的一种 UNIX，用于运行电话公司记录保存和管理过程中用到的数据库和事物处理系统。在 1983 年左右，这些 IPC 机制出现在了主流的 System V UNIX 系统上——System V IPC 的名称由此而来。

将 System V IPC 机制放在一起讨论的一个更加重要的原因是它们的编程接口都具备一些特征，因此很多同样的概念都适用于所有这些机制。

SUSv3 因需遵从 XSI 而要求实现 System V IPC，因此有时候这种机制也被称为 XSI IPC。

本章概述了 System V IPC 机制并详细介绍了所有这三种机制共同具备的特性。后面几个章节将分别对这三种机制进行介绍。

System V IPC 是一个通过 CONFIG_SYSVIPC 选项进行配置的内核选项。

更多资源请访问稀酷客(www.ckook.com)

45.1 概述

表 45-1 对使用 System V IPC 对象需用到的头文件和系统调用进行了总结。

一些实现要求在包含表 45-1 中的头文件之前先包含<sys/types.h>。一些较早的 UNIX 实现可能还要求包含<sys/ipc.h>。（没有哪个 UNIX 规范要求这些头文件。）

表 45-1：System V IPC 对象编程接口总结

接　口	消息队列	信号量	共享内存
头文件	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
关联数据结构	msqid_ds	semid_ds	shmid_ds
创建/打开对象	msgget()	semget()	shmget() + shmat()
关闭对象	（无）	（无）	shmdt()
控制操作	msgctl()	semctl()	shmctl()
执行 IPC	msgsnd()——写入消息 msgrcv()——接收消息	semop()——测试/调整信号量	访问共享区域中的内存

在大多数部署 Linux 的硬件架构上，系统调用 ipc(2)是所有 System V IPC 操作到内核的入口，表 45-1 中列出的所有调用实际上都被实现为位于这个系统调用之上的库函数。（这个约定的两个例外情况是 Alpha 和 IA-64，在这两个架构上，表中列出的调用被实现成了各个系统调用。）这个不太常见的方法是 System V IPC 在一开始被实现成可载入的内核模块的杰作。尽管在大多数 Linux 架构上它们实际上是库函数，但在本章中会将表 45-1 中列出的函数称为系统调用。只有 C 库的实现人员才需要使用 ipc(2)ipc(2)，在任何其他应用程序中使用这个调用将会使应用程序变得不可移植。

创建和打开一个 System V IPC 对象

每种 System V IPC 机制都有一个相关的 get 系统调用（msgget()、semget()或 shmget()），它与文件上的 open()系统调用类似。给定一个整数 key（类似于文件名），get 调用完成下列某个操作。

- 使用给定的 key 创建一个新 IPC 对象并返回一个唯一的标识符来标识该对象。
- 返回一个拥有给定的 key 的既有 IPC 对象的标识符。

本章将第二种做法（宽松地）称为打开一个既有 IPC 对象。在这种情况下，get 调用所做的事情是将一个数字（key）转换称为另一个数字（标识符）。

在 System V IPC 的上下文中的对象与面向对象程序设计中的对象毫无关系。这个术语仅仅用来将 System V IPC 机制与文件区分开来。尽管文件和 System V IPC 对象之间存在几点类似之处，但与标准的 UNIX 文件 I/O 模型相比，IPC 对象的用法在几个重要方面都存在差异，这也是 System V IPC 机制之所以复杂的一个原因。

IPC 标识符与文件描述符类似，在后续所有引用该 IPC 对象的系统调用中都需要用到它。但这两者之间存在一个重要的语义上的差别。文件描述符是一个进程特性，而 IPC 标识符则

更多资源请访问稀酷客(www.ckook.com)

是对象本身的一个属性并且对系统全局可见。所有访问同一对象的进程使用同样的标识符。这意味着如果知道一个 IPC 对象已经存在，那么可以跳过 get 调用，只要能够通过某种机制来获知对象的标识符即可。例如，创建对象的进程可能会将标识符写入一个可供其他进程读取的文件。

下面的例子展示了如何创建一个 System V 消息队列。

```
id = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR);
if (id == -1)
    errExit("msgget");
```

与所有的 get 调用一样，key 是第一个参数，标识符是函数的返回结果。传递给 get 调用的最后一个参数（flags）使用与文件一样的掩码常量（表 15-4）指定了新对象上的权限。在上面的例子中只给对象的所有者赋予了在队列中读取和写入消息的权限。

进程的 umask（参见 15.4.6 节）对新创建的 IPC 对象上的权限是不适用的。

一些 UNIX 实现为 IPC 权限定义了下面的位掩码常量：MSG_R、MSG_W、SEM_R、SEM_A、SHM_R 以及 SHM_W。它们对应于各个 IPC 机制的所有者（用户）的读取和写入权限。要获取对应的组和其他用户的权限位掩码则可以将这些常量右移 3 位和 6 位。SUSv3 并没有规定这些常量，它使用了与文件一样的位掩码，并且没有在 glibc 头中对这些常量进行定义。

所有需访问同一个 IPC 对象的进程在执行 get 调用时会指定同样的 key 以获取该对象的同一个标识符。在 45.2 节中将会介绍如何为应用程序选择一个 key。

如果没有与给定的 key 对应的 IPC 对象存在并且在 flags 参数中指定了 IPC_CREAT（与 open()的 O_CREAT 标记类似），那么 get 调用会创建一个新的 IPC 对象。如果不存在相应的 IPC 对象并且没有指定 IPC_CREAT（并且没有像 45.2 节中描述的那样将 key 指定为 IPC_PRIVATE），那么 get 调用会失败并返回 ENOENT 错误。

一个进程可以通过指定 IPC_EXCL 标记（类似于 open()的 O_EXCL 标记）来确保它是创建 IPC 对象的进程。如果指定了 IPC_EXCL 并且与给定 key 对应的 IPC 对象已经存在，那么 get 调用会失败并返 EEXIST 错误。

IPC 对象删除和对象持久

各种 System V IPC 机制的 ctl 系统调用（msgctl()、semctl()、shmctl()）在对象上执行一组控制操作，其中很多操作是特定于某种 IPC 机制的，但有一些是适用于所有的 IPC 机制的，其中一个就是 IPC_RMID 控制操作，它可以用来删除一个对象。如使用下面的调用可以删除一个共享内存对象。

```
if (shmctl(id, IPC_RMID, NULL) == -1)
    errExit("shmctl");
```

对于消息队列和信号量来讲，IPC 对象的删除是立即生效的，对象中包含的所有信息都会被销毁，不管是否有其他进程仍然在使用该对象。（这也是 System IPC 对象的操作与文件的操作不相似的其中一个地方。在 18.3 节中曾经讲过如果删除了指向文件的最后一个链接，那么实际上只有当所有引用该文件的打开着的文件描述符都被关闭之后才会删除该文件。）

共享内存对象的删除的操作是不同的。在 shmctl(id,IPC_RMID, NULL)调用之后，只有当所有使用该内存段的进程与该内存段分离之后（使用 shmdt()）才会删除该共享内存段。（这一

更多资源请访问稀酷客(www.ckook.com)

点与文件删除更加接近。）

System V IPC 对象具备内核持久性。一旦被创建之后，一个对象就一直存在直到它被显式地删除或系统被关闭。System V IPC 对象的这个属性是非常有用的。因为一个进程可以创建一个对象、修改其状态、然后退出并使得在后面某个时刻启动的进程可以访问这个对象。但这种属性也是存在缺点的，其原因如下。

- 系统对每种类型的 IPC 对象的数量是有限制的。如果没有删除不用的对象，那么应用程序最终可能会因达到这个限制而发生错误。
- 在删除一个消息队列或信号量对象时，多进程应用程序可能难以确定哪个进程是最后一个需要访问对象的进程，从而导致难以确定何时可以安全地删除对象。这里的问题是这些对象是无连接的——内核不会记录哪个进程打开了对象。（共享内存段不存在这个缺点，因为它们的删除操作的语义不同。）

45.2 IPC Key

System V IPC key 是一个整数值，其数据类型为 key_t。IPC get 调用将一个 key 转换成相应的整数 IPC 标识符。这些调用能够确保如果创建的是一个新 IPC 对象，那么对象能够得到一个唯一的标识符，如果指定了一个既有对象的 key，那么总是会取得该对象的（同样的）标识符。（在内部，内核会像 45.5 节中描述的那样为各种 IPC 机制维护着一个数据结构将 key 映射成标识符。）

那么如何产生唯一的 key 呢——一种确保不会偶然地取得其他应用程序所使用的一个既有 IPC 对象的标识符？这个问题存在三种解决方案。

- 随机地选取一个整数值作为 key 值，这些整数值通常会被放在一个头文件中，所有使用 IPC 对象的程序都需要包含这个头文件。这个方法的难点在于可能会无意中选取了一个已被另一个应用程序使用的值。
- 在创建 IPC 对象的 get 调用中将 IPC_PRIVATE 常量作为 key 的值，这样就会导致每个调用都会创建一个全新的 IPC 对象，从而确保每个对象都拥有一个唯一的 key。
- 使用 ftok()函数生成一个（接近唯一）key。

IPC_PRIVATE 和 ftok()是通常采用的技术。

使用 IPC_PRIVATE 产生一个唯一的 key

在创建一个新 IPC 对象时必须要像下面这样将 key 指定为 IPC_PRIVATE。

```
id = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR);
```

在上面的代码中无需指定 IPC_CREAT 和 IPC_EXCL 标记。

这项技术对于父进程在执行 fork()之前创建 IPC 对象从而导致子进程继承 IPC 对象标识符的多进程应用程序是特别有用的。在客户端-服务器应用程序中（即那些包含非相关进程的应用程序）也可以使用这项技术，但客户端必须要通过某种机制获取由服务器创建的 IPC 对象的标识符（反之亦然）。如在创建完一个 IPC 对象之后，服务器可以将这个标识符写入一个将会被客户端读取的文件中。

使用 ftok()产生一个唯一的 key

ftok()（file to key）函数返回一个适合在后续对某个 System V IPC get 系统调用进行调用

更多资源请访问稀酷客(www.ckook.com)

时使用的 key 值。

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

key_t ftok(char *pathname, int proj);
```

Returns integer key on success, or −1 on error

key 值是使用实现定义的算法根据提供的 pathname 和 proj 值生成的。SUSv3 要求如下。

- 算法只使用 proj 的最低的 8 个有效位。
- 应用程序必须要确保 pathname 引用一个可以应用 stat()的既有文件（否则 ftok()会返回−1）。
- 如果将引用同一个文件（即 i-node）不同的路径名（链接）传递给了 ftok()并且指定了同样的 proj 值，那么函数必须要返回同样的 key 值。

换句话说，ftok()使用 i-node 号来生成 key 值，而并没有使用文件名来生成 key 值。（由于 ftok()算法依赖于 i-node 号，因此在应用程序的生命周期中不应该将文件删除和重新创建，因为重新创建文件时很有可能会分配到一个不同的 i-node 号。）proj 的目的仅仅是允许从同一个文件中生成多个 key，这对于需创建同种类型的多个 IPC 对象的应用程序来讲是有用的。以前，proj 参数的类型为 char，并且在调用 ftok()通常传入的也是 char 值。

SUSv3 并没有规定当 proj 的值为 0 时 ftok()的行为。在 AIX 5.1 上，当 proj 为 0 时 ftok()返回−1。在 Linux 上，这个值没有特殊的含义，但可移植的应用程序应该避免将 proj 值设置为 0，因为还有 255 个值可用呢。

通常，传递给 ftok()的 pathname 会引用构成应用程序或由应用程序创建的文件或目录之一，协同运行的进程会将同样的 pathname 传递给 ftok()。

在 Linux 上，ftok()返回的 key 是一个 32 位的值，它通过取 proj 参数的最低 8 个有效位、包含该文件所属的文件系统的设备的设备号（即次要设备号）的最低 8 个有效位以及 pathname 所引用的文件的 i-node 号的最低 16 个有效位组合而成。（后两项信息通过在 pathname 上调用 stat()获得。）

glibc ftok()的算法与其他 UNIX 实现所采用的算法类似，它们都存在一个类似的限制：两个不同的文件可能会产生同样的 key 值（可能性非常小）。之所以会发生这种情况是因为不同文件系统上的两个文件的 i-node 号的最低有效位可能会相同，并且两个不同的磁盘设备（位于具备多个磁盘控制器的系统上）可能会拥有同样的次要设备号。但在实践中，不同的应用程序产生同样的 key 值的可能性非常非常小以至于使用 ftok()产生 key 已经是一项可靠的技术了。

ftok()的典型用法如下所示。

```
key_t key;
int id;

key = ftok("/mydir/myfile", 'x');
if (key == -1)
    errExit("ftok");
id = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR);
if (id == -1)
    errExit("msgget");
```


更多资源请访问稀酷客(www.ckook.com)

45.3 关联数据结构和对象权限

内核为 System V IPC 对象的每个实例都维护着一个关联数据结构。这个数据结构的形式因 IPC 机制（消息队列、信号量、或共享内存）的不同而不同，它是在各个 IPC 机制（参见表 45-1）对应的头文件中进行定义的。在后续的章节中将会详细介绍各种机制的关联数据结构的细节信息。

一个 IPC 对象的关联数据结构会在通过相应的 get 系统调用创建对象时进行初始化。对象一旦被创建之后，程序就可以通过指定 IPC_STAT 操作类型使用合适的 ctl 系统调用来获取这个数据结构的一个副本。使用 IPC_SET 操作能够修改这个数据结构中的部分数据。

除了各种 IPC 对象特有的数据之外，所有三种 IPC 机制的关联数据结构都包含一个子结构 ipc_perm，它保存了用于确定对象之上的权限的信息。

```
struct ipc_perm {
    key_t          __key;          /* Key, as supplied to 'get' call */
    uid_t          uid;            /* Owner's user ID */
    gid_t          gid;            /* Owner's group ID */
    uid_t          cuid;           /* Creator's user ID */
    gid_t          cgid;           /* Creator's group ID */
    unsigned short mode;           /* Permissions */
    unsigned short __seq;          /* Sequence number */
};
```

SUSv3 要求 ipc_perm 结构中除__key 和__seq 字段之外的所有其他字段都要具备。大多数 UNIX 实现都提供了相应的字段。

uid 和 gid 字段指定了 IPC 对象的所有权。cuid 和 cgid 字段保存着创建该对象的进程的用户 ID 和组 ID。一开始，相应的用户和创建者 ID 字段的值是一样的，它们都源自调用进程的有效 ID。创建者 ID 是不可变的，而所有者 ID 则可以通过 IPC_SET 操作进行修改。下面的代码演示了如何修改共享内存段的 uid 字段（关联数据结构的类型是 shmid_ds）。

```
struct shmid_ds shmds;

if (shmctl(id, IPC_STAT, &shmds) == -1)     /* Fetch from kernel */
    errExit("shmctl");
shmds.shm_perm.uid = newuid;                /* Change owner UID */
if (shmctl(id, IPC_SET, &shmds) == -1)      /* Update kernel copy */
    errExit("shmctl");
```

ipc_perm 子结构的 mode 字段保存着 IPC 对象的权限掩码。这些权限是使用在创建该对象的 get 系统调用中指定的 flags 参数的低 9 位初始化的，但后面使用 IPC_SET 操作则可以修改这个字段的值。

与文件一样，权限被分成了三类——owner（也称为 user）、group 以及 other——并且可以为各个类别指定不同的权限。但 IPC 对象的权限模型与文件权限模型存在一些显著差别。

- 对于 IPC 对象来讲只有读和写权限有意义。（对于信号量来讲，写权限通常被称为修改（alter）权限。）执行权限是没有意义的，在执行大多数访问检测时通常会忽略这个权限。
- 权限检测会根据进程的有效用户 ID、有效组 ID 以及辅助组 ID 来进行。（这与 Linux 上文件系统权限检测不同，它使用的是进程的文件系统 ID，具体可参考 9.5 节。）

更多资源请访问稀酷客(www.ckook.com)

IPC 对象上的进程权限分配的准确规则如下。

1．如果进程是特权进程（CAP_IPC_OWNER），那么所有权限都会被赋予 IPC 对象。

2. 如果进程的有效用户 ID 与 IPC 对象的所有者或创建者 ID 匹配，那么会将对象的 owner（user）的权限赋予进程。

3．如果进程的有效用户 ID 或任意一个辅助组 ID 与 IPC 对象的所有者组 ID 或创建者组 ID 匹配，那么会将对象的 group 的权限赋予进程。

4．否则会将对象的 other 的权限赋予进程。

在内核代码中，只有当一个进程没有通过其他测试被赋予所需的权限时才会去测试该进程是否是一个特权进程。之所以这样做是为了避免不必要地设置 ASU 进程记录标记，该标记用于指示进程是否使用超级用户权限（参见 28.1 节）。

注意 IPC_PRIVATE key 值的使用和 IPC_EXCL 标记的存在不会影响进程对 IPC 对象的访问，这种访问权限只由对象的所有者和权限来确定。

如何解释一个对象的读和写权限以及是否需要这些权限依赖于对象的类型以及所执行的操作。

当需获取一个既有 IPC 对象的标识符而执行一个 get 调用时会进行初次权限检测以确定在 flags 参数中指定的权限与既有对象上的权限是否匹配。如果不匹配，那么 get 调用会失败并返回 EACCES 错误。（除非特别指出，在下面列出的所有权限被拒绝的例子中都会返回这个错误码。）为说明问题，考虑同一组中的两个不同用户，其中一个用户使用了下面的调用创建了一个消息队列。

```
msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR | S_IRGRP);
                        /* rw-r----- */
```

当第二个用户尝试使用下面的调用获取这个消息队列的标识符时会失败，因为用户没有在消息队列上的写权限。

```
msgget(key, S_IRUSR | S_IWUSR);
```

第二个用户可以通过将 msgget()调用的第二个参数指定为 0 来绕过这种检测，这样就只有当程序试图执行一个需要 IPC 对象上的写权限的操作（如使用 msgsnd()写入一条消息）时才会发生错误。

---

get 调用代表了忽略执行权限的一种情况。尽管这种权限对于 IPC 对象来讲没有意义，但如果在一个既有对象上的 get 调用中要求执行权限，那么就会检测进程是否具备这个权限。

其他常见操作所需的权限如下所述。

- 从对象中获取信息（如从消息队列中读取一条消息，获取一个信号量的值，或因读取而附上一个共享内存段）需要读权限。
- 更新对象中的信息（如向消息队列写入一条消息，修改一个信号量的值，或因写入而附上一个共享内存段）需要写权限。
- 获取一个 IPC 对象的关联数据结构的副本（IPC_STAT ctl 操作）需要读权限。
- 删除一个 IPC 对象（IPC_RMID ctl 操作）或修改其关联数据结构（IPC_SET ctl 操作）不需要读或写权限，相反，调用进程必须是特权进程（CAP_SYS_ADMIN）或有效用户 ID 与对象的所有者用户 ID 或创建者用户 ID 匹配（否则返回错误 EPERM）。

更多资源请访问稀酷客(www.ckook.com)

可以设置一个 IPC 对象的权限使得所有者或创建者不能再使用 IPC_STAT 获取包含对象权限信息的关联数据结构（这意味着使用 45.6 节中介绍的 ipcs(1)命令无法打印出这个对象），但仍然可以使用 IPC_SET 修改它们。

其他各种机制特有的操作需要读或写权限或 CAP_IPC_OWNER 能力。在后面章节中讨论各种操作时将会介绍它们所需的权限。

45.4 IPC 标识符和客户端/服务器应用程序

在客户端/服务器应用程序中，服务器通常会创建 System V IPC 对象，而客户端则仅仅需要访问它们。换句话说，服务器在执行 get 调用时需要指定 IPC_CREAT 标记，而客户端在 get 调用中则会省略这个标记。

假设一个客户端参与了服务器的一个延伸会话，其中每个进程会执行多个 IPC 操作（如交换多条消息、一组信号量操作、或多次更新共享内存）。如果服务器进程崩溃或故意停止然后重启会发生什么情况呢？这时，盲目地重用由前一个服务器进程创建的 IPC 对象是毫无意义的，因为新服务器进程不清楚与 IPC 对象的当前状态相关的历史信息。（如消息队列中可能存在客户端因响应老的服务器进程之前发送的一条消息而发出的第二个请求。）

在这种情况下，服务器唯一可做的事情可能就是丢弃所有既有的客户端、删除由上一个服务器进程创建的 IPC 对象、创建 IPC 对象的新实例。新启动的服务器首先会通过在 get 调用中同时指定 IPC_CREAT 和 IPC_EXCL 标记创建一个 IPC 对象来处理服务器的上一个实例非正常终止的情况。如果 get 调用因具备指定 key 的对象已存在而失败，那么服务器就认为老的服务器进程之前创建了该对象，因此它会使用 IPC_RMID ctl 操作删除这个对象，然后再次执行一个 get 调用来创建对象。（这组步骤可能会与其他诸如确保另一个服务器进程当前不在运行之类的步骤组合起来使用，具体可参见 55.6 节。）程序清单 45-1 给出了一个消息队列可能需要执行的步骤。

程序清单 45-1：清理服务器中的 IPC 对象

svipc/svmsg_demo_server.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include "tlpi_hdr.h"

#define KEY_FILE "/some-path/some-file"
                                /* Should be an existing file or one
                                   that this program creates */

int
main(int argc, char *argv[])
{
    int msqid;
    key_t key;
    const int MQ_PERMS = S_IRUSR | S_IWUSR | S_IWGRP;   /* rw--w---- */

    /* Optional code here to check if another server process is
```

```
    already running */

    /* Generate the key for the message queue */

    key = ftok(KEY_FILE, 1);
    if (key == -1)
        errExit("ftok");
    /* While msgget() fails, try creating the queue exclusively */

    while ((msqid = msgget(key, IPC_CREAT | IPC_EXCL | MQ_PERMS)) == -1) {
        if (errno == EEXIST) {          /* MQ with the same key already
                                           exists - remove it and try again */
            msqid = msgget(key, 0);
            if (msqid == -1)
                errExit("msgget() failed to retrieve old queue ID");
            if (msgctl(msqid, IPC_RMID, NULL) == -1)
                errExit("msgget() failed to delete old queue");
            printf("Removed old message queue (id=%d)\n", msqid);

        } else {                        /* Some other error --> give up */
            errExit("msgget() failed");
        }
    }

    /* Upon loop exit, we've successfully created the message queue,
       and we can then carry on to do other work... */

    exit(EXIT_SUCCESS);
}
```

svipc/svmsg_demo_server.c

尽管重新启动的服务器会重新创建 IPC 对象，但如果在创建新 IPC 对象时将同样的 key 传递给 get 调用，那么总是会生成同样的标识符。读者可以从客户端的角度来考虑一下这个问题的解决方案。如果服务器重新创建的 IPC 对象使用了同样的标识符，那么客户端就无法知道服务器已经重启并且 IPC 对象已经不包含预期的历史信息了。

为解决这个问题，内核采用了一个算法（下一节描述），通常能够确保在创建新 IPC 对象时，对象会得到一个不同的标识符，即使传入的 key 是一样的。其结果是所有与老的服务器进程连接的客户端在使用旧的标识符时会从相关的 IPC 系统调用中收到一个错误。

程序清单 45-1 中的解决方案并没有完全解决在使用 System V 共享内存时识别出服务器重启的问题，因为共享内存对象只有在所有进程都与其虚拟地址空间分离之后才会被删除。但共享内存对象通常与 System V 信号量组合使用，而它们则会在 IPC_RMID 操作中立即被删除。这意味着客户端在试图访问被删除的信号量对象时能够知道服务器重启这件事情。

45.5 System V IPC get 调用使用的算法

图 45-1 给出了内核内部使用的一些表示 System V IPC 对象（本例中是信号量，但细节方面与其他 IPC 机制类似）相关信息的结构，包括用于计算 IPC key 的字段。对于每种 IPC 机制（共享内存、消息队列、或信号量），内核都会维护一个关联的 ipc_ids 结构，它记录着该 IPC 机制的所有实例的各种全局信息，包括一个大小会动态变化的指针数组 entries，数组中的每个元素

更多资源请访问稀酷客(www.ckook.com)

指向一个对象实例的关联数据结构（在信号量中是 semid_ds 结构）。entries 数组的当前大小记录在 size 字段中，max_id 字段记录着当前使用中的元素的最大下标。

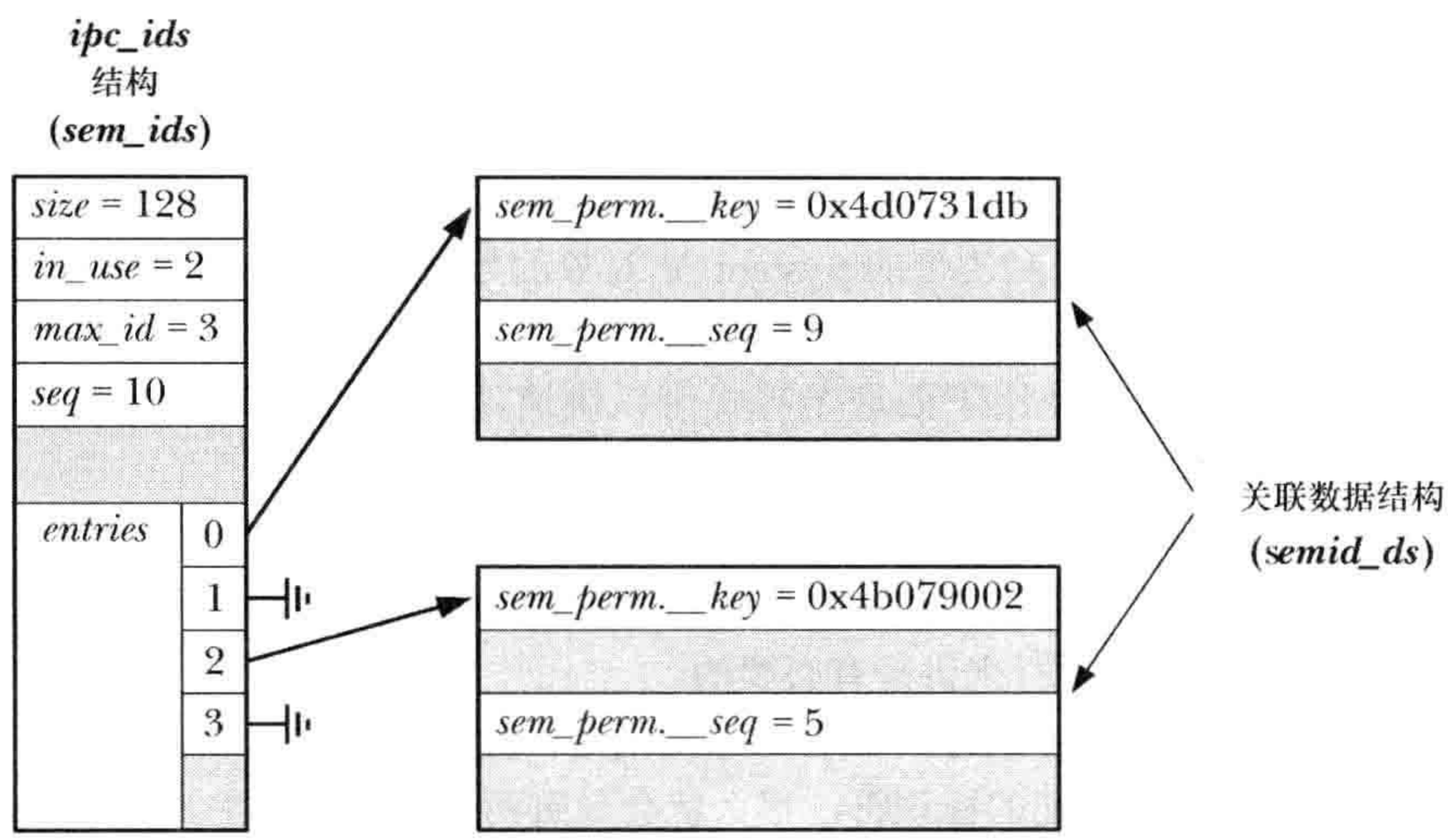


图 45-1：用于表示 System V IPC（信号量）对象的内核数据结构

在执行一个 IPC get 调用时，Linux 所采用的算法近似如下（其他系统使用了类似的算法）。

1. 在关联数据结构列表（entries 数组中的元素指向的结构）中搜索 key 字段与 get 调用中指定的参数匹配的结构。
   (a) 如果没有找到匹配的结构并且没有指定 IPC_CREAT，那么返回 ENOENT 错误。
   (b) 如果找到了一个匹配的结构，但同时指定了 IPC_CREAT 和 IPC_EXCL，那么返回 EEXIST 错误。
   (c) 否则在找到一个匹配的结构的情况下跳过下面的步骤。
2. 如果没有找到匹配的结构并且指定了 IPC_CREAT，那么会分配一个新的与所采用的机制对应的关联数据结构（在图 45-1 中是 semid_ds）并对其进行初始化。在这个操作中还会更新 ipc_ids 结构中的各个字段，并且可能还会重新设定 entries 数组的大小。指向新结构的指针会被放在 entries 中第一个未被占用的位置处。在这个初始化的过程中包含两个子步骤。
   (a) 传递给 get 调用的 key 值被复制到新分配的结构的 xxx_perm.__key 字段中。
   (b) ipc_ids 结构中 seq 字段的当前值被复制到关联数据结构的 xxx_perm.__seq 字段中，将 seq 字段的值加 1。
3. 使用下面的公式计算 IPC 对象的标识符。

```
identifier = index + xxx_perm.__seq * SEQ_MULTIPLIER
```

在用于计算 IPC 标识符的公式中，index 表示对象实例在 entries 数组中的下标，SEQ_MULTIPLIER 是一个值为 32768 的常数（内核源文件 include/linux/ipc.h 中的 IPCMNI）。如在图 45-1 中，key 值为 0x4b079002 的信号量生成的标识符为(2 + 5 * 32768) = 163842。

对于 get 调用所采用的算法需要注意下列几点。

- 即使使用同样的 key 创建了一个新 IPC 对象也几乎可以肯定对象被分配到的标识符是不同的，因为标识符的计算是根据保存在关联数据结构中的 seq 字段的值来进行的，而在同种类型的对象的创建过程中都会递增这个值。

更多资源请访问稀酷客(www.ckook.com)

内核所采用的算法在 seq 的值达到(INT_MAX / IPCMNI)——即 2147483647 / 32768 = 65535——时会将 seq 的值重置为 0。因此如果在系统运行期间已经创建了 65535 个对象，那么新 IPC 对象可能会与之前的对象拥有同样的标识符，从而导致新对象会重用之前的对象在 entries 数组中的位置（即在系统运行期间必须要释放之前的对象）。但发生这种情况的可能性非常小。

- 算法为 entries 数组的每个下标都生成一组不同的标识符值。
- 由于常量 IPCMNI 为每种类型的 System V 对象的数量设定了一个上限，因此算法确保所有既有 IPC 对象都拥有一个唯一的标识符。
- 给定一个标识符值，使用下面这个等式可以快速计算出它在 entries 数组中对应的下标。

```
index = identifier % SEQ_MULTIPLIER
```

能够快速地执行这种计算对于那些接收 IPC 对象标识符的 IPC 系统调用（即表 45-1 中除 get 调用的其他调用）的高效执行来讲是有必要的。

顺便提一下，当一个进程在执行一个 IPC 系统调用（如 msgctl()、semop()、或 shmat()）时传入了一个与既有对象不匹配的标识符，那么就会导致两个错误的发生。如果 entries 中相应下标处是空的，那么将会导致 EINVAL 错误的发生。如果下标指向了一个关联数据结构，但存储在该结构中的序号导致不会产生同样的标识符值，那么就假设这个数组下标指向的旧对象已经被删除了，该下标会被重用。通过错误 EIDRM 可以诊断出这种情况的发生。

45.6 ipcs 和 ipcrm 命令

ipcs 和 ipcrm 命令是 System V IPC 领域中类似于 ls 和 rm 文件命令的命令。使用 ipcs 能够获取系统上 IPC 对象的信息。在默认情况下，ipcs 会显示出所有对象，如下面的例子所示。

```
$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x6d0731db 262147     mtk        600        8192        2

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x6107c0b8 0          cecilia    660        6
0x6107c0b6 32769      britta     660        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes  messages
0x71075958 229376     cecilia    620        12          2
```

在 Linux 上，ipcs(1)只显示出拥有读权限的 IPC 对象的信息，而不管是否拥有这些对象。在一些 UNIX 实现上，ipcs 的行为与它在 Linux 上的行为一样，但在其他实现上，ipcs 会显示出所有对象，不管当前用户是否拥有这些对象上的读权限。

在默认情况下，ipcs 会显示出每个对象的 key、标识符、所有者以及权限（用一个八进制数字表示），后面跟着对象所特有的信息。

- 对于共享内存，ipcs 会显示出共享内存区域的大小、当前将共享内存区域附加到自己的虚拟地址空间的进程数以及状态标记。状态标记标识出了区域是否被锁进了 RAM 以防止交换（参见 48.7 节）以及在所有进程都与该区域分离之后是否已经将其标记为

更多资源请访问稀酷客(www.ckook.com)

待销毁了。

- 对于信号量，ipcs 会显示出信号集的大小。
- 对于消息队列，ipcs 会显示出队列中数据占据的字节总数以及消息数量。

ipcs(1)手册对各种能够显示 IPC 对象的其他信息的选项进行了说明。

ipcrm 命令删除一个 IPC 对象。这个命令的常规形式为下面两种形式中的一种。

```
$ ipcrm -X key
$ ipcrm -x id
```

在上面给出的命令中既可以将一个 IPC 对象的 key 指定为参数 key，也可以将一个 IPC 对象的标识符指定为参数 id 并且使用小写的 x 替换其大写形式或使用小写的 q（用于消息队列）或 s（用于信号量）或 m（用于共享内存）。因此使用下面的命令可以删除标识符为 65538 的信号量集。

```
$ ipcrm -s 65538
```

45.7 获取所有 IPC 对象列表

Linux 提供了两种获取系统上所有 IPC 对象列表的非标准方法。

- /proc/sysvipc 目录中的文件会列出所有 IPC 对象。
- 使用 Linux 特有的 ctl 调用。

本节将会介绍/proc/sysvipc 目录中的文件，在 46.6 节将会对 ctl 调用进行介绍，并提供一个示例程序来列出系统上所有 System V 消息队列。

其他一些 UNIX 实现提供了它们自己的用于获取所有 IPC 标识符列表的非标准方法，如 Solaris 为此提供了 msgids()、semids()以及 shmids()系统调用。

/proc/sysvipc 目录中三个只读文件提供的信息与通过 ipcs 获取的信息是一样的。

- /proc/sysvipc/msg 列出所有消息队列及其特性。
- /proc/sysvipc/sem 列出所有信号量集及其特性。
- /proc/sysvipc/shm 列出所有共享内存段及其特性。

与 ipcs 命令不同，这些文件总是会显示出相应种类的所有对象，不管是否在这些对象上拥有读权限。

下面给出了一个示例/proc/sysvipc/sem 文件的内容（为符合版面的要求，这里删除了一些空格）。

```
$ cat /proc/sysvipc/sem
key      semid perms   nsems   uid   gid   cuid  cgid     otime        ctime
  0   16646144   600       4  1000   100   1000   100         0   1010166460
```

这三个/proc/sysvipc 文件为程序和脚本提供了一种遍历给定种类的所有既有 IPC 对象的方法（不可移植）。

获取给定种类的所有 IPC 对象的最佳可移植的做法是解析 ipcs(1)的输出。

45.8 IPC 限制

由于 System V IPC 对象会消耗系统资源，因此内核对各种 IPC 对象进行了各式各样的限制以防止资源被耗尽。SUSv3 没有对用于限制 System V IPC 对象的方法进行规定，但大多数 UNIX

更多资源请访问稀酷客(www.ckook.com)

实现（包括 Linux）都采用了类似的框架来对对象进行各种各样的限制。在下面介绍各种 System V IPC 机制的章节中将会对相关的限制进行讨论并指出与其他 UNIX 实现之间的差别。

尽管在不同 UNIX 实现之间对各种 IPC 对象所能施加的限制类型通常是类似的，但查看和修改这些限制的方法则是不同的。下面章节中介绍的方法是 Linux 特有的（它们一般都需要使用/proc/sys/kernel 目录中的文件），而在其他实现上则需要使用不同的方法。

在 Linux 上，ipcs –l 命令可以用来列出各种 IPC 机制上的限制。程序可以使用 Linux 特有的 IPC_INFO ctl 操作来获取同样的信息。

45.9 总结

System V IPC 是首先在 System V 中被广泛使用的三种 IPC 机制的名称并且之后被移植到了大多数 UNIX 实现中以及被加入了加入了各种标准中。这三种 IPC 机制允许进程之间交换消息的消息队列，允许进程同步对共享资源的访问的信号量，以及允许两个或更多进程共享内存的同一个页的共享内存。

这三种 IPC 机制在 API 和语义上存在很多相似之处。对于每种 IPC 机制来讲，get 系统调用会创建或打开一个对象。给定一个整数 key，get 调用返回一个整数标识符用来在后续的系统调用中引用对象。每种 IPC 机制还拥有一个对应的 ctl 调用用来删除一个对象以及获取和修改对象的关联数据结构中的各种特性（如所有权和权限）。

用来为新 IPC 对象生成标识符的算法被设计成将（立即）复用同样的标识符的可能性降到最小，即使相应的对象已经被删除了，甚至是使用同样的 key 来创建新对象也一样。这样客户端-服务器应用程序就能够正常工作了——重新启动的服务器进程能够检测到并删除上一个服务器进程创建的 IPC 对象，并且这个动作会令上一个服务器进程的客户端所保存的标识符失效。

ipcs 命令列出了当前位于系统上的所有 System V IPC 对象。ipcrm 命令用来删除 System IPC 对象。

在 Linux 上，/proc/sysvipc 目录中的文件可以用来获取系统上所有 System V IPC 对象的信息。

每种 IPC 机制都有一组相关的限制，它们通过阻止创建任意数量的 IPC 对象来避免系统资源的耗尽。/proc/sys/kernel 目录中的各个文件可以用来查看和修改这些限制。

更多信息

在[Maxwell, 1999]和[Bovet & Cesati, 2005]中能够找到 System V IPC 在 Linux 上的实现的信息。[Goodheart & Cox, 1994]介绍了 System V Release 4 中 System V IPC 的实现。

45.10 习题

45-1. 编写一个程序来验证 ftok()所采用的算法是否如 45.2 节中描述的那样使用了文件的 i-node 号、次要设备号以及 proj 值。（通过几个例子打印出所有这些值以及 ftok()的返回值的十六进制形式即可。）

45-2. 实现 ftok()。

45-3. 验证（通过实验）45.5 节中有关用于生成 System V IPC 标识符的算法的声明。

更多资源请访问稀酷客(www.ckook.com)

第46章

System V 消息队列

本章介绍 System V 消息队列。消息队列允许进程以消息的形式交换数据。尽管消息队列在某些方面与管道和 FIFO 类似，但它们之间仍然存在显著的差别。

- 用来引用消息队列的句柄是一个由 msgget()调用返回的标识符。这些标识符与 UNIX 系统上大多数其他形式的 I/O 所使用的文件描述符是不同的。
- 通过消息队列进行的通信是面向消息的，即读者接收到由写者写入的整条消息。读取一条消息的一部分而让剩余部分遗留在队列中或一次读取多条消息都是不可能的。这一点与管道不通，管道提供的是一个无法进行区分的字节流（即使用管道时读者一次可以读取任意数量的字节数，不管写者写入的数据块的大小是什么）。
- 除了包含数据之外，每条消息还有一个用整数表示的类型。从消息队列中读取消息既可以按照先入先出的顺序，也可以根据类型来读取消息。

本章最后（46.9 节）将会对 System V 消息队列所存在的限制进行总结。由于存在这些限制，因此新应用程序应该尽可能避免使用 System V 消息队列，而应该使用其他形式的 IPC 机制，如 FIFO、POSIX 消息队列以及 socket。但在消息队列一开始被设计出来的时候，这些候选机制不是还没有被发明出来就是还没有在 UNIX 实现中被广泛采用，其结果是存在各类使用消息队列的既有应用程序，这也是在这里对消息队列进行介绍的主要原因之一。

46.1 创建或打开一个消息队列

msgget()系统调用创建一个新消息队列或取得一个既有队列的标识符。

```
#include <sys/types.h>        /* For portability */
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

Returns message queue identifier on success, or −1 on error

key 参数是使用 45.2 节中描述的方法之一生成的一个键（即通常是值 IPC_PRIVATE 或 ftok()返回的一个键）。msgflg 参数是一个指定施加于新消息队列之上的权限或检查一个既有队

更多资源请访问稀酷客(www.ckook.com)

列的权限的位掩码。此外，在 msgflg 参数中还可以将下列标记中的零个或多个标记取 OR（|）以控制 msgget()的操作。

IPC_CREAT

如果没有与指定的 key 对应的消息队列，那么就创建一个新队列。

IPC_EXCL

如果同时还指定了 IPC_CREAT 并且与指定的 key 对应的队列已经存在，那么调用就会失败并返回 EEXIST 错误。

在 45.1 节中对这些标记进行了详细描述。

msgget()系统调用首先会在所有既有消息队列中搜索与指定的键对应的队列。如果找到了一个匹配的队列，那么就会返回该对象的标识符（除非在 msgflg 中同时指定了 IPC_CREAT 和 IPC_EXCL，那样的话就返回一个错误）。如果没有找到匹配的队列并且在 msgflg 中指定了 IPC_CREAT，那么就会创建一个新队列并返回该队列的标识符。

程序清单 46-1 为 msgget()系统调用提供了一个命令行界面。这个程序允许使用命令行选项和参数来指定传递给 msgget()调用的 key 和 msgflg 参数的所有组合。usageError()函数给出了这个程序所接受的命令格式的细节信息。在成功创建队列之后，这个程序会打印出队列标示符。46.2.2 节将会演示这个程序的用法。

程序清单 46-1：使用 msgget()

svmsg/svmsg_create.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include "tlpi_hdr.h"
static void             /* Print usage info, then exit */
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s", msg);
    fprintf(stderr, "Usage: %s [-cx] {-f pathname | -k key | -p} "
                        "[octal-perms]\n", progName);
    fprintf(stderr, "    -c           Use IPC_CREAT flag\n");
    fprintf(stderr, "    -x           Use IPC_EXCL flag\n");
    fprintf(stderr, "    -f pathname  Generate key using ftok()\n");
    fprintf(stderr, "    -k key       Use 'key' as key\n");
    fprintf(stderr, "    -p           Use IPC_PRIVATE key\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int numKeyFlags;            /* Counts -f, -k, and -p options */
    int flags, msqid, opt;
    unsigned int perms;
    long lkey;
    key_t key;

    /* Parse command-line options and arguments */
```


更多资源请访问稀酷客(www.ckook.com)

```
    numKeyFlags = 0;
    flags = 0;

    while ((opt = getopt(argc, argv, "cf:k:px")) != -1) {
        switch (opt) {
        case 'c':
            flags |= IPC_CREAT;
            break;

        case 'f':                   /* -f pathname */
            key = ftok(optarg, 1);
            if (key == -1)
                errExit("ftok");
            numKeyFlags++;
            break;

        case 'k':                   /* -k key (octal, decimal or hexadecimal) */
            if (sscanf(optarg, "%li", &lkey) != 1)
                cmdLineErr("-k option requires a numeric argument\n");
            key = lkey;
            numKeyFlags++;
            break;

        case 'p':
            key = IPC_PRIVATE;
            numKeyFlags++;
            break;
        case 'x':
            flags |= IPC_EXCL;
            break;

        default:
            usageError(argv[0], "Bad option\n");
        }
    }

    if (numKeyFlags != 1)
        usageError(argv[0], "Exactly one of the options -f, -k, "
                            "or -p must be supplied\n");

    perms = (optind == argc) ? (S_IRUSR | S_IWUSR) :
                getInt(argv[optind], GN_BASE_8, "octal-perms");

    msqid = msgget(key, flags | perms);
    if (msqid == -1)
        errExit("msgget");

    printf("%d\n", msqid);
    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_create.c

46.2 交换消息

msgsnd()和 msgrcv()系统调用执行消息队列上的 I/O。这两个系统调用接收的第一个参数是消息队列标识符（msqid）。第二个参数 msgp 是一个由程序员定义的结构的指针，该结构用

更多资源请访问稀酷客(www.ckook.com)

于存放被发送或接收的消息。这个结构的常规形式如下。

```
struct mymsg {
    long mtype;                 /* Message type */
    char mtext[];               /* Message body */
}
```

这个定义仅仅简要地说明了消息的第一个部分包含了消息类型，它用一个类型为 long 的整数来表示，而消息的剩余部分则是由程序员定义的一个结构，其长度和内容可以是任意的，而无需是一个字符数组。因此 mgsp 参数的类型为 void *，这样就允许传入任意结构的指针了。

mtext 字段长度可以为零，当对于接收进程来讲所需传递的信息仅通过消息类型就能表示或只需要知道一条消息本身是否存在时，这种做法有时候就变得非常有用了。

46.2.1 发送消息

msgsnd()系统调用向消息队列写入一条消息。

```
#include <sys/types.h>          /* For portability */
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Returns 0 on success, or −1 on error

使用 msgsnd()发送消息必须要将消息结构中的 mtype 字段的值设为一个大于 0 的值（在下一节讨论 msgrcv()时会介绍这个值的用法）并将所需传递的信息复制到程序员定义的 mtext 字段中。msgsz 参数指定了 mtext 字段中包含的字节数。

在使用 msgsnd()发送消息时并不存在 write()所具备的部分写的概念。这也是成功的 msgsnd()只需要返回 0 而不是所发送的字节数的原因。

最后一个参数 msgflg 是一组标记的位掩码，用于控制 msgsnd()的操作，目前只定义了一个这样的标记。

IPC_NOWAIT

执行一个非阻塞的发送操作。通常，当消息队列满时，msgsnd()会阻塞直到队列中有足够的空间来存放这条消息。但如果指定了这个标记，那么 msgsnd()就会立即返回 EAGAIN 错误。

当 msgsnd()调用因队列满而发生阻塞时可能会被信号处理器中断。当发生这种情况时，msgsnd()总是会返回 EINTR 错误。（在 21.5 节中曾指出过 msgsnd()系统调用永远不会自动重启，不管在建立信号处理器时是否设置了 SA_RESTART 标记。）

向消息队列写入消息要求具备在该队列上的写权限。

程序清单 46-2 为 msgsnd()系统调用提供了一个命令行界面。usageError()函数显示了这个程序接受的命令行格式。注意这个程序没有使用 msgget()系统调用。（在 45.1 节中讲过一个进程无需使用 get 调用来访问一个 IPC 对象。）相反，这里通过将消息队列标识符设定为命令行参数的值来指定消息队列。在 46.2.2 节中将会演示这个程序的用法。

程序清单 46-2：使用 msgsnd()发送一条消息

svmsg/svmsg_send.c

```
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"
```


更多资源请访问稀酷客(www.ckook.com)

```
#define MAX_MTEXT 1024

struct mbuf {
    long mtype;                         /* Message type */
    char mtext[MAX_MTEXT];              /* Message body */
};
static void             /* Print (optional) message, then usage description */
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s", msg);
    fprintf(stderr, "Usage: %s [-n] msqid msg-type [msg-text]\n", progName);
    fprintf(stderr, "    -n       Use IPC_NOWAIT flag\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int msqid, flags, msgLen;
    struct mbuf msg;                    /* Message buffer for msgsnd() */
    int opt;                            /* Option character from getopt() */

    /* Parse command-line options and arguments */

    flags = 0;
    while ((opt = getopt(argc, argv, "n")) != -1) {
        if (opt == 'n')
            flags |= IPC_NOWAIT;
        else
            usageError(argv[0], NULL);
    }

    if (argc < optind + 2 || argc > optind + 3)
        usageError(argv[0], "Wrong number of arguments\n");

    msqid = getInt(argv[optind], 0, "msqid");
    msg.mtype = getInt(argv[optind + 1], 0, "msg-type");

    if (argc > optind + 2) {            /* 'msg-text' was supplied */
        msgLen = strlen(argv[optind + 2]) + 1;
        if (msgLen > MAX_MTEXT)
            cmdLineErr("msg-text too long (max: %d characters)\n", MAX_MTEXT);

        memcpy(msg.mtext, argv[optind + 2], msgLen);

    } else {                            /* No 'msg-text' ==> zero-length msg */
        msgLen = 0;
    }

    /* Send message */

    if (msgsnd(msqid, &msg, msgLen, flags) == -1)
        errExit("msgsnd");

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_send.c

46.2.2 接收消息

msgrcv()系统调用从消息队列中读取（以及删除）一条消息并将其内容复制进 msgp 指向的缓冲区中。

```
#include <sys/types.h>        /* For portability */
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t maxmsgsz, long msgtyp, int msgflg);
```

Returns number of bytes copied into *mtext* field, or −1 on error

msgp 缓冲区中 mtext 字段的最大可用空间是通过 maxmsgsz 参数来指定的。如果队列中待删除的消息体的大小超过了 maxmsgsz 字节，那么就不会从队列中删除消息，并且 msgrcv()会返回错误 E2BIG。（这是默认行为，可以使用 MSG_NOERROR 标记来改变这种行为，稍后就会对此进行介绍。）

读取消息的顺序无需与消息被发送的一致。可以根据 mtype 字段的值来选择消息，而这个选择过程是由 msgtyp 参数来控制的，具体如下所述。

- 如果 msgtyp 等于 0，那么会删除队列中的第一条消息并将其返回给调用进程。
- 如果 msgtyp 大于 0，那么会将队列中第一条 mtype 等于 msgtyp 的消息删除并将其返回给调用进程。通过指定不同的 msgtyp 值，多个进程能够从同一个消息队列中读取消息而不会出现竞争读取同一条消息的情况。比较有用的一项技术是让各个进程选取与自己的进程 ID 匹配的消息。
- 如果 msgtyp 小于 0，那么就会将等待消息当成优先队列来处理。队列中 mtype 最小并且其值小于或等于 msgtyp 的绝对值的第一条消息会被删除并返回给调用进程。

下面通过一个例子将讲解 msgtyp 小于 0 时的情况。假设一个消息队列包含了图 46-1 中显示的一组消息，接着执行一系列的 msgrcv()调用，其形式如下。

```
msgrcv(id, &msg, maxmsgsz, -300, 0);
```

这些 msgrcv()调用会按照 2（类型为 100）、5（类型为 100）、3（类型为 200）、1（类型为 300）的顺序读取消息。后续的调用会阻塞，因为剩余的消息的类型（400）超过了 300。

msgflg 参数是一个位掩码，它的值通过将下列标记中的零个或多个取 OR 来确定。

IPC_NOWAIT

执行一个非阻塞接收。通常如果队列中没有匹配 msgtyp 的消息，那么 msgrcv()会阻塞直到队列中存在匹配的消息为止。指定 IPC_NOWAIT 标记会导致 msgrcv()立即返回 ENOMSG 错误。（返回 EAGAIN 错误会使一致性更强一点，因为非阻塞的 msgsnd()和 FIFO 中的非阻塞读取也是返回这个错误，但之所以返回 ENOMSG 错误是存在历史原因的，SUSv3 也要求返回 ENOMSG。）

MSG_EXCEPT

只有当 msgtyp 大于 0 时这个标记才会起作用，它会强制对常规操作进行补足，即将队列中第一条 mtype 不等于 msgtyp 的消息删除并将其返回给调用者。这个标记是 Linux 特有的，只有当定义了_GNU_SOURCE 之后才会在<sys/msg.h>中提供这个标记。在图 46-1 中给出的消息队列上执行一系列形式为 msgrcv(id, &msg, maxmsgsz, 100, MSG_EXCEPT)的调用将会按照 1、3、4 顺序读取消息，之后发生阻塞。

更多资源请访问稀酷客(www.ckook.com)

MSG_NOERROR

在默认情况下，当消息的 mtext 字段的大小超过了可用空间时（由 maxmsgsz 参数定义），msgrcv()调用会失败。如果指定了 MSG_NOERROR 标记，那么 msgrcv()将会从队列中删除消息并将其 mtext 字段的大小截短为 maxmsgsz 字节，然后将消息返回给调用者。被截去的数据将会丢失。

msgrcv()成功完成之后会返回接收到的消息的 mtext 字段的大小，发生错误时则返回−1。

与 msgsnd()一样，如果被阻塞的 msgrcv()调用被一个信号处理器中断了，那么调用会失败并返回 EINTR 错误，不管在建立信号处理器时是否设置了 SA_RESTART 标记。

从消息队列中读取消息需要具备在队列上的读权限。

队列位置	消息类型 (*mtype*)	消息正文 (*mtext*)
1	300	...
2	100	...
3	200	...
4	400	...
5	100	...

图 46-1：包含不同类型的消息的示例消息队列

示例程序

程序清单 46-3 为 msgrcv()系统调用提供了一个命令行界面。usageError()函数显示了这个程序接受的命令行格式。与程序清单 46-2 中演示 msgsnd()的用法的程序一样，这个程序也没有使用 msgget()系统调用，相反它需要在命令行参数中传入一个消息队列标识符。

下面的 shell 会话演示了程序清单 46-1、程序清单 46-2、程序清单 46-3 中的程序的用法。这里首先使用 IPC_PRIVATE 键创建了一个消息队列，然后向队列中写入了三条不同类型的消息。

```
$ ./svmsg_create -p
32769                                          ID of message queue
$ ./svmsg_send 32769 20 "I hear and I forget."
$ ./svmsg_send 32769 10 "I see and I remember."
$ ./svmsg_send 32769 30 "I do and I understand."
```

接着使用程序清单 46-3 中的程序从队列中读取类型小于或等于 20 的消息。

```
$ ./svmsg_receive -t -20 32769
Received: type=10; length=22; body=I see and I remember.
$ ./svmsg_receive -t -20 32769
Received: type=20; length=21; body=I hear and I forget.
$ ./svmsg_receive -t -20 32769
```

上面最后一条命令会阻塞，因为队列中已经没有类型小于或等于 20 的消息了。因此需要输入 Control-C 来终止这个命令，然后执行一个从队列中读取任意类型的消息的命令。

```
Type Control-C to terminate program
$ ./svmsg_receive 32769
Received: type=30; length=23; body=I do and I understand.
```

程序清单 46-3：使用 msgrcv()读取一条消息

svmsg/svmsg_receive.c

```
#define _GNU_SOURCE     /* Get definition of MSG_EXCEPT */
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

#define MAX_MTEXT 1024
```

```
struct mbuf {
    long mtype;                 /* Message type */
    char mtext[MAX_MTEXT];      /* Message body */
};

static void
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s", msg);
    fprintf(stderr, "Usage: %s [options] msqid [max-bytes]\n", progName);
    fprintf(stderr, "Permitted options are:\n");
    fprintf(stderr, "    -e       Use MSG_NOERROR flag\n");
    fprintf(stderr, "    -t type  Select message of given type\n");
    fprintf(stderr, "    -n       Use IPC_NOWAIT flag\n");
#ifdef MSG_EXCEPT
    fprintf(stderr, "    -x       Use MSG_EXCEPT flag\n");
#endif
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int msqid, flags, type;
    ssize_t msgLen;
    size_t maxBytes;
    struct mbuf msg;            /* Message buffer for msgrcv() */
    int opt;                    /* Option character from getopt() */

    /* Parse command-line options and arguments */

    flags = 0;
    type = 0;
    while ((opt = getopt(argc, argv, "ent:x")) != -1) {
        switch (opt) {
        case 'e':       flags |= MSG_NOERROR;   break;
        case 'n':       flags |= IPC_NOWAIT;    break;
        case 't':       type = atoi(optarg);    break;
#ifdef MSG_EXCEPT
        case 'x':       flags |= MSG_EXCEPT;    break;
#endif
        default:        usageError(argv[0], NULL);
        }
    }

    if (argc < optind + 1 || argc > optind + 2)
        usageError(argv[0], "Wrong number of arguments\n");

    msqid = getInt(argv[optind], 0, "msqid");
    maxBytes = (argc > optind + 1) ?
                getInt(argv[optind + 1], 0, "max-bytes") : MAX_MTEXT;

    /* Get message and display on stdout */

    msgLen = msgrcv(msqid, &msg, maxBytes, type, flags);
    if (msgLen == -1)
        errExit("msgrcv");
```

```
    printf("Received: type=%ld; length=%ld", msg.mtype, (long) msgLen);
    if (msgLen > 0)
        printf("; body=%s", msg.mtext);
    printf("\n");

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_receive.c

46.3 消息队列控制操作

msgctl()系统调用在标识符为 msqid 的消息队列上执行控制操作。

```
#include <sys/types.h>        /* For portability */
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Returns 0 on success, or −1 on error

cmd 参数指定了在队列上执行的操作，其取值是下列值中的一个。

IPC_RMID

立即删除消息队列对象及其关联的 msqid_ds 数据结构。队列中所有剩余的消息都会丢失，所有被阻塞的读者和写者进程会立即醒来，msgsnd()和 msgrcv()会失败并返回错误 EIDRM。这个操作会忽略传递给 msgctl()的第三个参数。

IPC_STAT

将与这个消息队列关联的 msqid_ds 数据结构的副本放到 buf 指向的缓冲区中。在 46.4 节将会介绍 msqid_ds 结构。

IPC_SET

使用 buf 指向的缓冲区提供的值更新与这个消息队列关联的 msqid_ds 数据结构中被选中的字段。

45.3 节介绍了更多有关这些操作的细节，包括调用进程所需的特权和权限。46.6 节将会介绍 cmd 可取的其他一些值。

程序清单 46-4 演示了如何使用 msgctl()来删除一个消息队列。

程序清单 46-4：删除 System V 消息队列

svmsg/svmsg_rm.c

```
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int j;
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [msqid...]\n", argv[0]);

    for (j = 1; j < argc; j++)
        if (msgctl(getInt(argv[j], 0, "msqid"), IPC_RMID, NULL) == -1)
```


更多资源请访问稀酷客(www.ckook.com)

```
            errExit("msgctl %s", argv[j]);

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_rm.c

46.4 消息队列关联数据结构

每个消息队列都有一个关联的 msqid_ds 数据结构，其形式如下。

```
struct msqid_ds {
    struct ipc_perm msg_perm;       /* Ownership and permissions */
    time_t          msg_stime;      /* Time of last msgsnd() */
    time_t          msg_rtime;      /* Time of last msgrcv() */
    time_t          msg_ctime;      /* Time of last change */
    unsigned long   __msg_cbytes;   /* Number of bytes in queue */
    msgqnum_t       msg_qnum;       /* Number of messages in queue */
    msglen_t        msg_qbytes;     /* Maximum bytes in queue */
    pid_t           msg_lspid;      /* PID of last msgsnd() */
    pid_t           msg_lrpid;      /* PID of last msgrcv() */
};
```

名称 msqid_ds 中的缩写 msg 会令程序员感到糊涂。只有这一个消息队列接口使用这种拼写方式。

msgqnum_t 和 msglen_t 数据类型——用于定义 msg_qnum 和 msg_qbytes 字段的类型——在 SUSv3 中被规定为无符号整型。

各种消息队列系统调用会隐式地更新 msqid_ds 结构中的字段，使用 msgctl() IPC_SET 操作则可以显式地更新其中一些字段。细节信息如下。

msg_perm

在创建消息队列之后会按照 45.3 节中描述的那样初始化这个子结构中的字段。uid、gid 以及 mode 子字段可以通过 IPC_SET 来更新。

msg_stime

在队列被创建之后这个字段会被设置为 0；后续每次成功的 msgsnd()调用都会将这个字段设置为当前时间。这个字段和 msqid_ds 结构中其他时间戳字段的类型都是 time_t；它们存储自新纪元到现在的秒数。

msg_rtime

在消息队列被创建之后这个字段会被设置为 0，然后每次成功的 msgrcv()调用都会将这个字段设置为当前时间。

msg_ctime

当消息队列被创建或成功执行了 IPC_SET 操作之后会将这个字段设置为当前时间。

__msg_cbytes

当消息队列被创建之后会将这个字段设置为 0，后续每次成功的 msgsnd()和 msgrcv()调用都会对这个字段进行调整以反映出队列中所有消息的 mtext 字段包含的字节数总和。

msg_qnum

当消息队列被创建之后会将这个字段设置为 0，后续每次成功的 msgsnd()调用会递增这个字

更多资源请访问稀酷客(www.ckook.com)

段的值并且每次成功的 msgrcv()调用会递减这个字段的值以便反映出队列中的消息总数。

msg_qbytes

这个字段的值为消息队列中所有消息的 mtext 字段的字节总数定义了一个上限。在队列被创建之后会将这个字段的值初始化为 MSGMNB。特权（CAP_SYS_RESOURCE）进程可以使用 IPC_SET 操作将 msg_qbytes 的值调整为 0 字节到 INT_MAX（32 位平台上是 2147483647）字节之间的任意一个值。特权用户可以修改 Linux 特有的/proc/sys/kernel/ msgmnb 文件中包含的值以修改所有后续创建的消息队列的初始msg_qbytes设置以及非特权进程后续对msg_qbytes修改时所能设置的上限。46.5 节将会介绍更多有关消息队列限制方面的内容。

msg_lspid

当队列被创建之后会将这个字段设置为 0，后续每次成功的 msgsnd()调用会将其设置为调用进程的进程 ID。

msg_lrpid

当消息队列被创建之后会将这个字段设置为 0，后续每次成功的 msgrcv()调用会将其设置为调用进程的进程 ID。

SUSv3 对上面除__msg_cbytes 字段之外的所有其他字段都进行了规定。而大多数 UNIX 实现都提供了一个与__msg_cbytes 字段等价的字段。

程序清单 46-5 演示了如何使用 IPC_STAT 和 IPC_SET 操作来修改一个消息队列的 msg_qbytes 设置。

程序清单 46-5：修改一个 System V 消息队列的 msg_qbytes 设置

svmsg/svmsg_chqbytes.c

```
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct msqid_ds ds;
    int msqid;
    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s msqid max-bytes\n", argv[0]);

    /* Retrieve copy of associated data structure from kernel */

    msqid = getInt(argv[1], 0, "msqid");
    if (msgctl(msqid, IPC_STAT, &ds) == -1)
        errExit("msgctl");

    ds.msg_qbytes = getInt(argv[2], 0, "max-bytes");

    /* Update associated data structure in kernel */

    if (msgctl(msqid, IPC_SET, &ds) == -1)
        errExit("msgctl");

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_chqbytes.c

46.5 消息队列的限制

大多数 UNIX 实现会对 System V 消息队列的操作施加各种各样的限制。下面会对 Linux 系统上的限制进行介绍并指出其与其他 UNIX 实现之间的差别。

Linux 会对队列操作施加下列限制。括号中列出了限制所影响到的系统调用以及当达到限制时所产生的错误。

MSGMNI

这是系统级别的一个限制，它规定了系统中所能创建的消息队列标识符（换句话说是消息队列）的数量。（msgget()，ENOSPC）

MSGMAX

这是系统级别的一个限制，它规定了单条消息中最多可写入的字节数（mtext）。（msgsnd()，EINVAL）

MSGMNB

一个消息队列中一次最多保存的字节数（mtext）。这个限制是一个系统级别的参数，它用来初始化与消息队列相关联的 msqid_ds 数据结构的 msg_qbytes 字段。根据 46.4 节中的描述可以修改各个队列的 msg_qbytes 值。如果达到一个队列的 msg_qbytes 限制，那么 msgsnd()会阻塞或在 IPC_NOWAIT 被设置时返回 EAGAIN 错误。

一些 UNIX 实现还定义了下列限制。

MSGTQL

这是系统级别的一个限制，它规定了系统中所有消息队列所能存放的消息总数。

MSGPOOL

这是系统级别的一个限制，它规定了用来存放系统中所有消息队列中的数据的缓冲池的大小。

尽管 Linux 没有规定上述限制，但它会根据队列的 msg_qbytes 限制来限制单个队列中的消息总数。只有当向队列写入长度为零的消息时才会涉及到这个限制，其效果是对向队列可写入的长度为零的消息的数量的限制与对向队列可写入的长度为 1 字节的消息的数量的限制是一样的。这样就能够防止向队列无限制地写入长度为零的消息。尽管这些消息不包含数据，但每个长度为零的消息都会消耗一小块内存以便系统进行簿记工作。

在系统启动的时候会将消息队列限制设置为默认值。不同版本的内核上的默认值是不同的。（一些发行厂商发行的内核中的默认设置与 vanilla 内核中的默认设置是不同的。）在 Linux 上可以通过/proc 文件系统中的文件来查看和修改这些限制。表 46-1 显示了与各个限制对应的/proc 文件。下面是一个 x86-32 系统上 Linux 2.6.31 内核中的默认限制。

```
$ cd /proc/sys/kernel
$ cat msgmni
748
$ cat msgmax
8192
$ cat msgmnb
16384
```


更多资源请访问稀酷客(www.ckook.com)

表 46-1：System V 消息队列限制

限　　制	上限值（x86-32）	/proc/sys/kernel 中的对应文件
MSGMNI	32768（IPCMNI）	msgmni
MSGMAX	依赖于可用内存	msgmax
MSGMNB	2147483647（INT_MAX）	msgmnb

表 46-1 中的上限值那一列显示了在 x86-32 架构上每个限制所能达到的最大值。注意尽管可以将 MSGMNB 限制的值设置为 INT_MAX，但在消息队列中载入这么多的数据之前可能会达到其他一些限制（如缺少内存）。

Linux 特有的 msgctl() IPC_INFO 操作能够获取一个类型为 msginfo 的结构，其中包含了各种消息队列限制的值。

```
struct msginfo buf;

msgctl(0, IPC_INFO, (struct msqid_ds *) &buf);
```

有关 IPC_INFO 和 msginfo 结构的细节信息可以在 msgctl(2)手册中找到。

46.6　显示系统中所有消息队列

在 45.7 节中曾讲过一种获取系统中所有 IPC 对象列表的方法：通过/proc 文件系统中的一组文件。下面介绍获取相同信息的第二种方法：通过 Linux 特有的一组 IPC ctl（msgctl()、semctl() 以及 shmctl()）操作。（ipcs 程序使用了这些操作。）这些操作如下。

- MSG_INFO、SEM_INFO 以及 SHM_INFO：MSG_INFO 操作完成两件事情。第一件事情是它将返回一个结构来详细描述系统上所有消息队列的资源消耗情况。第二件事情是作为 ctl 调用的函数结果，它将返回指向表示消息队列对象的数据结构的 entries 数组中最大项的下标（参见图 45-1）。SEM_INFO 和 SHM_INFO 操作分别对信号量集合共享内存段执行了类似的任务。要从相应的 System V IPC 头文件中获取这三个常量的定义就必须要定义_GNU_SOURCE 特性测试宏。

本书随带的源代码中 svmsg/svmsg_info.c 文件中给出了一个使用 MSG_INFO 获取 msginfo 结构的例子，该结构包含了与所有消息队列对象所消耗的资源相关的信息。

- MSG_STAT、SEM_STAT 以及 SHM_STAT：与 IPC_STAT 操作一样，这些操作获取一个 IPC 对象的关联数据结构，但它们之间存在两方面的不同。第一，与 ctl 调用的第一个参数为 IPC 标识符不同，这些操作的第一个参数是 entries 数组中的一个下标。第二，如果操作执行成功了，那么作为函数结果，ctl 调用会返回与该下标对应的 IPC 对象的标识符。要从相应的 System V IPC 头文件中获取这三个常量的定义就必须要定义_GNU_SOURCE 特性测试宏。

按照下面的步骤可以列出系统上所有消息队列。

1. 使用 MSG_INFO 操作找到消息队列的 entries 数组的最大下标（maxind）。
2. 执行一个循环，对 0 到 maxind（包含）之间的每一个值都执行一个 MSG_STAT 操作。在循环过程中忽略因 entries 数组中的元素为空而发生的错误（EINVAL）以及在数组中元素所引用的对象上不具备相应的权限而发生的错误（EACCES）。

更多资源请访问稀酷客(www.ckook.com)

程序清单 46-6 按照上面的步骤实现了对消息队列的处理。下面的 shell 会话日志演示了这个程序的用法。

```
$ ./svmsg_ls
maxind: 4

index     ID       key       messages
   2    98306  0x00000000       0
   4   163844  0x000004d2       2
$ ipcs -q                                  Check above against output of ipcs

------ Message Queues --------
key        msqid      owner     perms     used-bytes   messages
0x00000000 98306      mtk       600       0            0
0x000004d2 163844     mtk       600       12           2
```

程序清单 46-6：显示系统上所有 System V 消息队列

svmsg/svmsg_ls.c

```
#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int maxind, ind, msqid;
    struct msqid_ds ds;
    struct msginfo msginfo;

    /* Obtain size of kernel 'entries' array */

    maxind = msgctl(0, MSG_INFO, (struct msqid_ds *) &msginfo);
    if (maxind == -1)
        errExit("msgctl-MSG_INFO");

    printf("maxind: %d\n\n", maxind);
    printf("index     id       key       messages\n");

    /* Retrieve and display information from each element of 'entries' array */

    for (ind = 0; ind <= maxind; ind++) {
        msqid = msgctl(ind, MSG_STAT, &ds);
        if (msqid == -1) {
            if (errno != EINVAL && errno != EACCES)
                errMsg("msgctl-MSG_STAT");              /* Unexpected error */
            continue;                                   /* Ignore this item */
        }

        printf("%4d %8d  0x%08lx %7ld\n", ind, msqid,
                (unsigned long) ds.msg_perm.__key, (long) ds.msg_qnum);
    }

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_ls.c

46.7 使用消息队列实现客户端-服务器应用程序

在客户端-服务器应用程序设计中使用 System V 消息队列的方式有很多种，本节将介绍其中两种。

- 在服务器和客户端之间使用单个消息队列进行双向消息交换。
- 服务器和各个客户端使用单独的消息队列，服务器上的队列用来接收进入的客户端请求，相应的响应则通过各个客户端队列来发送给客户端。

至于选择何种方法依赖于应用程序的需求，稍后介绍可能会影响到决策的其中一些因素。

服务器和客户端使用一个消息队列

当服务器与客户端之间交换的消息大小较小时使用一个消息队列是合适的，但需要注意以下几点。

- 由于多个进程可能会同时读取消息，因此必须要使用消息类型（mtype）字段来让各个进程只选择那些发给自己的消息。完成这个任务的一种方法是将客户端的进程 ID 作为服务器发送给客户端的消息的消息类型。客户端可以将其进程 ID 作为消息的一部分发送给服务器。此外，发送给服务器的消息也必须要能够使用唯一的消息类型来加以区分，而这可以使用数字 1 来完成，因为 1 是永远运行着的 init 进程的进程 ID，客户端进程的进程 ID 永远都不可能为这个值。（另一种方法是将服务器的进程 ID 作为消息类型，但客户端要获取这个信息就比较困难了。）图 46-2 给出了这种计数模型。

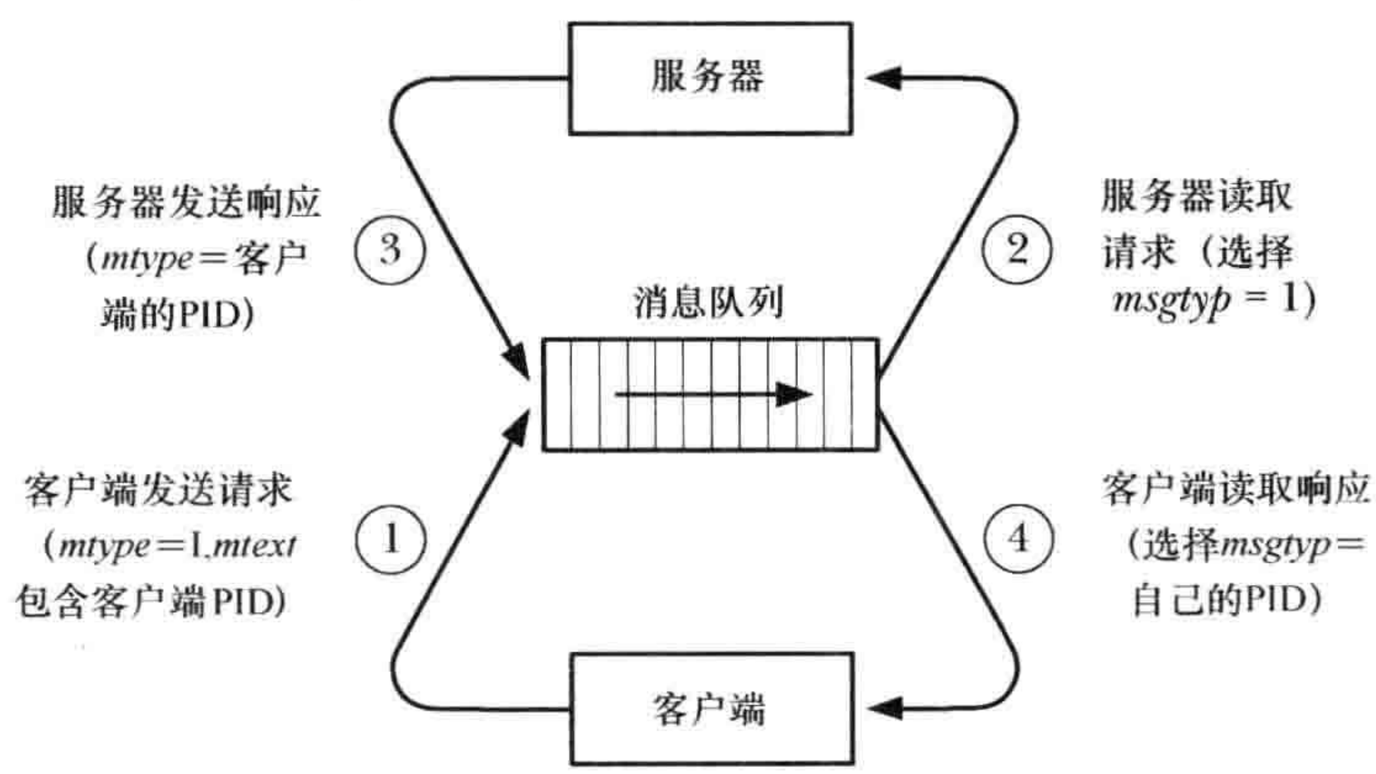


图 46-2：在客户端-服务器 IPC 中使用单个消息队列

- 消息队列的容量是有限的，而这可能会导致一系列问题的发生。其中一个问题就是多个并行的客户端可能会填满消息队列，从而导致死锁的发生，即所有新客户端都无法提交请求，服务器在写入任何响应时会发生阻塞。另一个问题是行为不良或恶意的客户端可能不会读取服务器的响应，从而导致队列中充满了未被读取的消息，进而阻止了客户端和服务器之间的通信。（使用两个队列——一个用于存放客户端发送给服务器的消息，另一个用于存放服务器发送给客户端的消息——将会解决第一个问题，但无法解决第二个问题。）

更多资源请访问稀酷客(www.ckook.com)

一个客户端使用一个消息队列

当需要交换的消息的大小较大或当使用单个消息队列可能会导致发生前面列出的问题时最好为每个客户端都使用一个消息队列（服务器也需要一个队列）。使用这种方法需要注意以下几点。

- 每个客户端必须要创建自己的消息队列（通常使用 IPC_PRIVATE 键）并通知服务器队列的标识符，这通常通过将标识符作为客户端发送给服务器的消息的一部分来完成。
- 系统对消息队列的数量是有限制的（MSGMNI），这个限制的默认值在一些系统上是非常低的。如果同时运行的客户端数量庞大，那么可能就需要提高这个限制的值。
- 服务器应该允许出现客户端的消息队列不再存在的情况（可能是由于客户端不小心删除了队列）。

下一节将会对为每个客户端使用一个队列这种方法进行深入介绍。

46.8 使用消息队列实现文件服务器应用程序

本节将介绍一个为每个客户端使用一个消息队列的客户端/服务器应用程序。这个应用程序是一个简单的文件服务器。客户端向服务器的消息队列发送一个请求消息请求指定名称的文件的内容。服务器将文件的内容作为一系列的消息返回到客户端私有的消息队列中。图 46-3 概览了该应用程序。

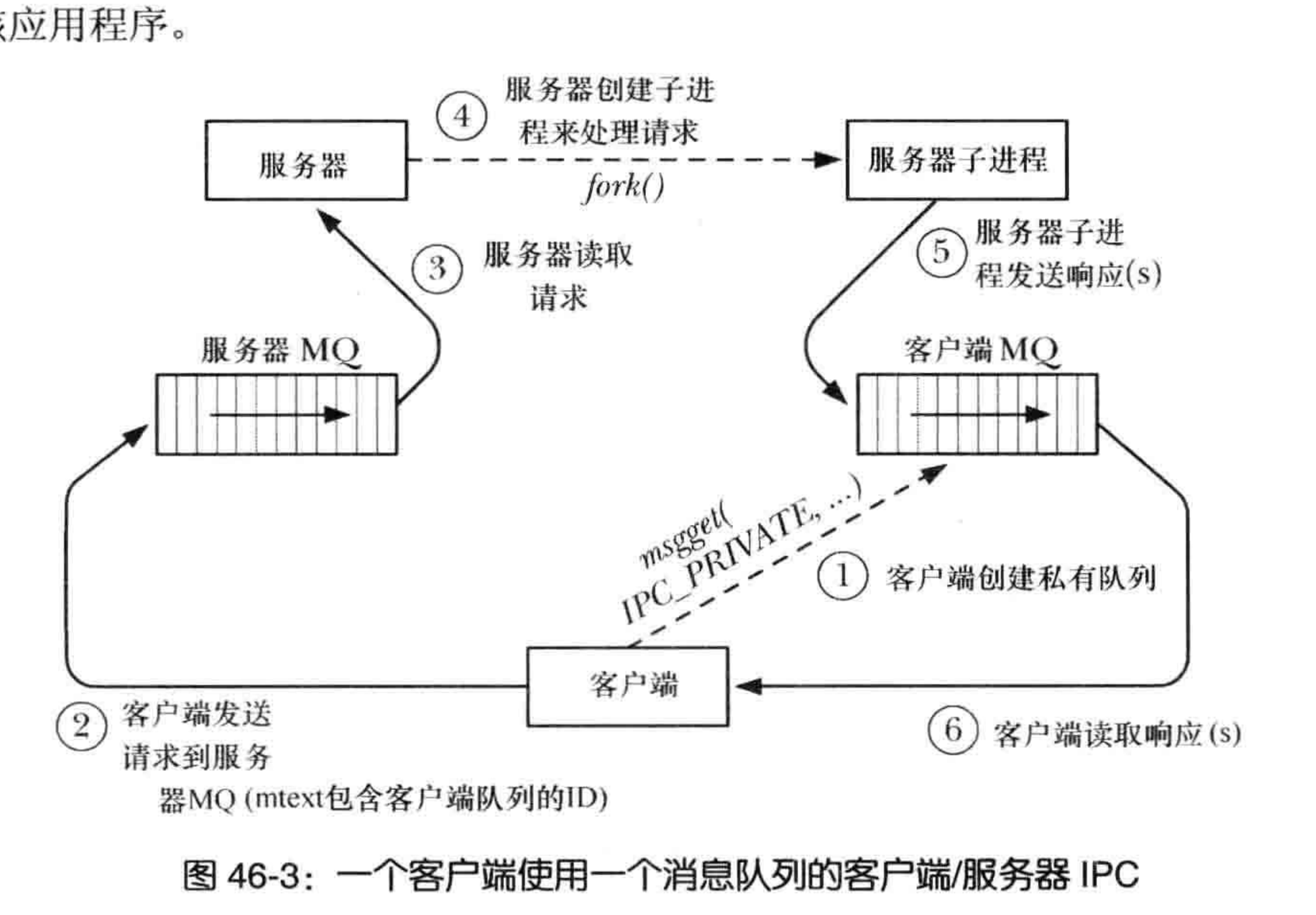


图 46-3：一个客户端使用一个消息队列的客户端/服务器 IPC

由于服务器对客户端不做任何鉴权，因此所有使用客户端的用户都能够获取服务器所能访问的所有文件。更复杂一点的服务器会在返回请求的文件之前对客户端完成某种鉴权操作。

公共头文件

程序清单 46-7 给出了服务器和客户端都需要包含的头文件。这个头文件为服务器的消息队列定义了一个众所周知的键（SERVER_KEY），并且定义了客户端和服务器之间传递的消息的格式。

更多资源请访问稀酷客(www.ckook.com)

requestMsg 结构定义了客户端发送给服务器的请求格式。在这个结构中，mtext 部分由两个字段构成：客户端消息队列的标识符和客户端请求的文件的路径名。常量 REQ_MSG_SIZE 等于这两个字段大小的总和，它在使用这个结构的 msgsnd()调用中是作为 msgsz 参数使用的。

responseMsg 结构定义了服务器返回给客户端的响应消息的格式。响应消息中的 mtype 字段提供了与消息内容有关的信息，其取值由 RESP_MT_*常量规定。

程序清单 46-7：svmsg_file_server.c 和 svmsg_file_client.c 的公共头文件

svmsg/svmsg_file.h

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <stddef.h>                     /* For definition of offsetof() */
#include <limits.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/wait.h>
#include "tlpi_hdr.h"

#define SERVER_KEY 0x1aaaaaa1           /* Key for server's message queue */

struct requestMsg {                     /* Requests (client to server) */
    long mtype;                         /* Unused */
    int  clientId;                      /* ID of client's message queue */
    char pathname[PATH_MAX];            /* File to be returned */
};

/* REQ_MSG_SIZE computes size of 'mtext' part of 'requestMsg' structure.
   We use offsetof() to handle the possibility that there are padding
   bytes between the 'clientId' and 'pathname' fields. */

#define REQ_MSG_SIZE (offsetof(struct requestMsg, pathname) - \
                      offsetof(struct requestMsg, clientId) + PATH_MAX)

#define RESP_MSG_SIZE 8192

struct responseMsg {                    /* Responses (server to client) */
    long mtype;                         /* One of RESP_MT_* values below */
    char data[RESP_MSG_SIZE];           /* File content / response message */
};

/* Types for response messages sent from server to client */

#define RESP_MT_FAILURE 1               /* File couldn't be opened */
#define RESP_MT_DATA    2               /* Message contains file data */
#define RESP_MT_END     3               /* File data complete */
```

svmsg/svmsg_file.h

服务器程序

程序清单 46-8 给出了这个应用程序的服务器程序。有关服务器需要注意以下几点。

- 服务器被设计成并发地处理请求。并发服务器设计最好像程序清单 44-7 中所做的那样采用迭代式设计，因为需要避免出现因一个客户端请求一个大文件而导致所有其他客户端请求等待的情况。

更多资源请访问稀酷客(www.ckook.com)

- 每个客户端请求通过创建一个子进程返回请求的文件来完成⑧。同时，主服务器进程等待后续的客户端请求。有关服务器子进程需要注意以下几点。
  - 由于通过 fork()创建的子进程会继承父进程栈的一个副本，因此它能够获取主服务器进程读取的请求消息的一个副本。
  - 服务器子进程在处理完相关的客户端请求之后会终止⑨。
- 为避免创建僵死进程（参见 26.2 节），服务器为 SIGCHLD 建立了一个处理器⑥并在处理器中调用了 waitpid()①。
- 父服务器进程中的 msgrcv()调用可能会阻塞，其结果是可能会被 SIGCHLD 处理器中断。为处理这种情况，需要使用一个循环来完成 EINTR 错误发生之后的重启操作。
- 服务器子进程执行 serveRequest()函数②，该函数向客户端返回三种消息。mtype 为 RESP_MT_FAILURE 时表示服务器无法打开请求的文件③；RESP_MT_DATA 用来表示包含文件数据的一系列消息④；RESP_MT_END（data 字段的长度为零）用来表示文件数据传输的结束⑤。

在练习 46-4 中将会考虑几种改进和扩展服务器程序的方法。

程序清单 46-8：一个使用 System V 消息队列的文件服务器

svmsg/svmsg_file_server.c

```
#include "svmsg_file.h"

static void             /* SIGCHLD handler */
grimReaper(int sig)
{
    int savedErrno;

    savedErrno = errno;                 /* waitpid() might change 'errno' */
①   while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
    errno = savedErrno;
}
static void             /* Executed in child process: serve a single client */
② serveRequest(const struct requestMsg *req)
{
    int fd;
    ssize_t numRead;
    struct responseMsg resp;

    fd = open(req->pathname, O_RDONLY);
    if (fd == -1) {                     /* Open failed: send error text */
③       resp.mtype = RESP_MT_FAILURE;
        snprintf(resp.data, sizeof(resp.data), "%s", "Couldn't open");
        msgsnd(req->clientId, &resp, strlen(resp.data) + 1, 0);
        exit(EXIT_FAILURE);             /* and terminate */
    }

    /* Transmit file contents in messages with type RESP_MT_DATA. We don't
       diagnose read() and msgsnd() errors since we can't notify client. */

④   resp.mtype = RESP_MT_DATA;
    while ((numRead = read(fd, resp.data, RESP_MSG_SIZE)) > 0)
        if (msgsnd(req->clientId, &resp, numRead, 0) == -1)
```

```
            break;

        /* Send a message of type RESP_MT_END to signify end-of-file */

⑤       resp.mtype = RESP_MT_END;
        msgsnd(req->clientId, &resp, 0, 0);          /* Zero-length mtext */
    }

    int
    main(int argc, char *argv[])
    {
        struct requestMsg req;
        pid_t pid;
        ssize_t msgLen;
        int serverId;
        struct sigaction sa;

        /* Create server message queue */

        serverId = msgget(SERVER_KEY, IPC_CREAT | IPC_EXCL |
                                S_IRUSR | S_IWUSR | S_IWGRP);
        if (serverId == -1)
            errExit("msgget");

        /* Establish SIGCHLD handler to reap terminated children */

        sigemptyset(&sa.sa_mask);
        sa.sa_flags = SA_RESTART;
        sa.sa_handler = grimReaper;
⑥       if (sigaction(SIGCHLD, &sa, NULL) == -1)
            errExit("sigaction");
        /* Read requests, handle each in a separate child process */

        for (;;) {
            msgLen = msgrcv(serverId, &req, REQ_MSG_SIZE, 0, 0);
            if (msgLen == -1) {
⑦               if (errno == EINTR)          /* Interrupted by SIGCHLD handler? */
                    continue;                /* ... then restart msgrcv() */
                errMsg("msgrcv");            /* Some other error */
                break;                       /* ... so terminate loop */
            }

⑧           pid = fork();                    /* Create child process */
            if (pid == -1) {
                errMsg("fork");
                break;
            }

            if (pid == 0) {                  /* Child handles request */
                serveRequest(&req);
⑨               _exit(EXIT_SUCCESS);
            }
            /* Parent loops to receive next client request */
        }

        /* If msgrcv() or fork() fails, remove server MQ and exit */

        if (msgctl(serverId, IPC_RMID, NULL) == -1)
```

```
        errExit("msgctl");
    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_file_server.c

客户端程序

程序清单 46-9 给出了这个应用程序的客户端。有关客户端程序需注意以下几点。

- 客户端使用 IPC_PRIVATE 键创建一个消息队列②并使用 atexit()③建立了一个退出处理器①以确保在客户端退出时删除队列。
- 客户端将其队列标识符以及所请求的文件的路径名打包在请求中传递给服务器④。
- 客户端对服务器发送的第一个响应消息即为失败通知（mtype 等于 RESP_MT_FAILURE），这种情况的处理方式是打印服务器返回的错误消息并退出⑤。
- 如果成功打开了文件，那么客户端会循环⑥接收包含文件内容的一系列消息（mtype 等于 RESP_MT_DATA）。整个循环过程在收到文件结束消息（mtype 等于 RESP_MT_END）之后结束。

这个简单的客户端并没有对由服务器故障而引起的各种情况进行处理。在练习 46-5 中将会考虑一些改进方案。

程序清单 46-9：使用 System V 消息队列的文件服务器的客户端

svmsg/svmsg_file_client.c

```
#include "svmsg_file.h"

static int clientId;

static void
removeQueue(void)
{
    if (msgctl(clientId, IPC_RMID, NULL) == -1)
①       errExit("msgctl");
}

int
main(int argc, char *argv[])
{
    struct requestMsg req;
    struct responseMsg resp;
    int serverId, numMsgs;
    ssize_t msgLen, totBytes;

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

    if (strlen(argv[1]) > sizeof(req.pathname) - 1)
        cmdLineErr("pathname too long (max: %ld bytes)\n",
                (long) sizeof(req.pathname) - 1);

    /* Get server's queue identifier; create queue for response */

    serverId = msgget(SERVER_KEY, S_IWUSR);
    if (serverId == -1)
        errExit("msgget - server message queue");
```

```
②       clientId = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR | S_IWGRP);
        if (clientId == -1)
            errExit("msgget - client message queue");

③       if (atexit(removeQueue) != 0)
            errExit("atexit");

        /* Send message asking for file named in argv[1] */

        req.mtype = 1;                          /* Any type will do */
        req.clientId = clientId;
        strncpy(req.pathname, argv[1], sizeof(req.pathname) - 1);
        req.pathname[sizeof(req.pathname) - 1] = '\0';
                                                /* Ensure string is terminated */

④       if (msgsnd(serverId, &req, REQ_MSG_SIZE, 0) == -1)
            errExit("msgsnd");
        /* Get first response, which may be failure notification */

        msgLen = msgrcv(clientId, &resp, RESP_MSG_SIZE, 0, 0);
        if (msgLen == -1)
            errExit("msgrcv");

⑤       if (resp.mtype == RESP_MT_FAILURE) {
            printf("%s\n", resp.data);      /* Display msg from server */
            if (msgctl(clientId, IPC_RMID, NULL) == -1)
                errExit("msgctl");
            exit(EXIT_FAILURE);
        }

        /* File was opened successfully by server; process messages
           (including the one already received) containing file data */

        totBytes = msgLen;                      /* Count first message */
⑥       for (numMsgs = 1; resp.mtype == RESP_MT_DATA; numMsgs++) {
            msgLen = msgrcv(clientId, &resp, RESP_MSG_SIZE, 0, 0);
            if (msgLen == -1)
                errExit("msgrcv");
            totBytes += msgLen;
        }

        printf("Received %ld bytes (%d messages)\n", (long) totBytes, numMsgs);

        exit(EXIT_SUCCESS);
    }
```

svmsg/svmsg_file_client.c

下面的 shell 会话演示了程序清单 46-8 和程序清单 46-9 中的程序的使用。

```
$ ./svmsg_file_server &                 Run server in background
[1] 9149
$ wc -c /etc/services                   Show size of file that client will request
764360 /etc/services
$ ./svmsg_file_client /etc/services
Received 764360 bytes (95 messages)     Bytes received matches size above
$ kill %1                               Terminate server
[1]+  Terminated          ./svmsg_file_server
```

46.9 System V 消息队列的缺点

UNIX 系统为同一系统上不同进程之间的数据传输提供了多种机制，既包括无分隔符的字节流形式（管道、FIFO 以及 UNIX domain 流 socket），也包括有分隔符的消息形式（System V 消息队列、POSIX 消息队列以及 UNIX domain 数据报 socket）。

System V 消息队列的一个与众不同的特性是它能够为每个消息加上一个数字类型。应用程序可以使用这个完成两件事情：读取进程可以根据类型来选择消息或者它们可以采用一种优先队列策略以便优先读取高优先级的消息（即那些消息类型值更低的消息）。

但 System V 消息队列也存在几个缺点。

- 消息队列是通过标识符引用的，而不是像大多数其他 UNIX I/O 机制那样使用文件描述符。这意味着在第 63 章介绍的各种基于文件描述符的 I/O 技术（如 select()、poll() 以及 epoll）将无法应用于消息队列上。此外，在程序中编写同时处理消息队列的输入和基于文件描述符的 I/O 机制的代码要比编写只处理文件描述符的代码更加复杂。（在练习 63-3 中将考虑一种组合两种 I/O 模型的方法。）
- 使用键而不是文件名来标识消息队列会增加额外的程序设计复杂性，同时还需要使用 ipcs 和 ipcrm 来替换 ls 和 rm。ftok()函数通常能产生一个唯一的键，但却无法保证。使用 IPC_PRIVATE 键能确保产生唯一的队列标识符，但需要使这个标识符对需要用到它的其他进程可见。
- 消息队列是无连接的，内核不会像对待管道、FIFO 以及 socket 那样维护引用队列的进程数。因此就难以回答下列问题。
  - 一个应用程序何时能够安全地删除一个消息队列？（不管是否有进程在后面某个时刻需要从队列中读取数据而过早地删除队列会导致数据丢失。）
  - 应用程序如何确保不再使用的队列会被删除呢？
- 消息队列的总数、消息的大小以及单个队列的容量都是有限制的。这些限制都是可配置的，但如果一个应用程序超出了这些默认限制的范围，那么在安装应用程序的时候就需要完成一些额外的工作了。

总体上来讲，最好避免使用 System V 消息队列。当碰到需要使用根据类型选择消息的工具的情况时应该考虑使用其他替代方案。POSIX 消息队列（第 52 章）就是这样一种替代方案。更深层次一点的替代方案是使用基于多文件描述符的通信通道，它们在提供与根据类型选择消息类似的功能的同时还允许使用在 63 章介绍的另一种 I/O 模型。例如，如果需要传输“普通”和“优先”消息，那么可以为两种消息类型使用一组 FIFO 或 UNIX domain socket，然后使用 select()或 poll()监控两个通道上的文件描述符。

46.10 总结

System V 消息队列允许进程通过交换由一个数字类型和一个包含任意数据的消息体构成的消息的形式来进行通信。消息队列的区别于其他机制的特性是消息是有边界的，并且接收者能够根据类型来选择消息，而无需按照先入先出的顺序来读取消息。

之所以得出其他 IPC 机制通常要优于 System V 消息队列的结论是因为几个因素，其中最主要的一个是引用消息队列不会用到文件描述符。这意味着在消息队列上无法使用另一种 I/O

更多资源请访问稀酷客(www.ckook.com)

模型，特别是同时监控消息队列和文件描述符以查看是否可进行 I/O 将变得复杂。此外，消息队列无连接（即不进行引用计数）这个事实使得应用程序难以知道何时能够安全地删除一个队列。

46.11 习题

46-1. 试验程序清单 46-1（svmsg_create.c）、程序清单 46-2（svmsg_send.c）以及程序清单 46-3（svmsg_receive.c）中的程序以验证对 msgget()、msgsnd()以及 msgrcv()系统调用的理解。

46-2. 改造 44.8 节中的序号客户端-服务器应用程序使之使用 System V 消息队列。使用单个消息队列来传输客户端到服务器以及服务器到客户端之间的消息。使用 46.8 节中介绍的消息类型规范。

46-3. 在 46.8 节中的客户端-服务器应用程序中，客户端为何在消息体（在 clientId 字段中）中传递其消息队列的标识符，而不是在消息类型（mtype）中传递？

46-4. 对 46.8 节中的客户端-服务器应用程序做出下列变更。

（a）替换服务器中硬编码的消息队列键使之使用 IPC_PRIVATE 生成一个唯一的标识符，然后将这个标识符写入一个众所周知的文件中。客户端必须要从这个文件中读取标识符。服务器在终止时需要删除这个文件。

（b）在服务器程序的 serveRequest()函数中并没有对系统调用错误进行诊断。添加使用 syslog()（参见 37.5 节）记录错误的代码。

（c）在服务器中添加代码使之在启动时成为一个 daemon（参见 37.2 节）。

（d）在服务器中为 SIGTERM 和 SIGINT 添加一个处理器来执行一个干净的退出。处理器需要删除消息队列以及（如果这个练习的前面一部分已经实现的话）用来存放服务器的消息队列标识符的文件。在处理器中加入分离处理器，然后再次触发同样一个调用该处理器的信号的代码（26.1.4 节介绍了其中的原理以及完成这个任务的步骤）。

（e）服务器子进程并没有对客户端可能过早终止的情况进行处理，这样服务器子进程就会填充客户端的消息队列，然后无限阻塞下去。修改服务器使之处理这种情况，即像 23.3 节中描述的那样在调用 msgsnd()时设置一个超时。如果服务器子进程确信客户端已经消失了，那么它就应该删除客户端的消息队列，然后退出（可能要在使用 syslog()记录一条消息之后）。

46-5. 程序清单 46-9 中给出的客户端（svmsg_file_client.c）没有对服务器发生故障的各种情况进行处理。特别是如果服务器消息队列被填满了（可能由于服务器终止而队列被其他客户端填满了），那么 msgsnd()调用会无限阻塞下去。类似地，如果服务器没有成功地将响应发送给客户端，那么 msgrcv()调用会无限阻塞下去。在客户端中添加代码使之在这些调用上设置超时（参见 23.3 节）。只要其中一个调用超时了，那么程序就需要将错误报告给用户并终止。

46-6. 使用 System V 消息队列编写一个简单的聊天应用程序（与 talk(1)类似，但没有 curses 界面）。为每个客户端使用一个消息队列。

更多资源请访问稀酷客(www.ckook.com)

第47章

System V 信号量

本章将介绍 System V 信号量。与上一章中介绍的 IPC 机制不同，System V 信号量不是用来在进程间传输数据的。相反，它们用来同步进程的动作。信号量的一个常见用途是同步对一块共享内存的访问以防止出现一个进程在访问共享内存的同时另一个进程更新这块内存的情况。

一个信号量是一个由内核维护的整数，其值被限制为大于或等于 0。在一个信号量上可以执行各种操作（即系统调用），包括：

- 将信号量设置成一个绝对值；
- 在信号量当前值的基础上加上一个数量；
- 在信号量当前值的基础上减去一个数量；
- 等待信号量的值等于 0。

上面操作中的后两个可能会导致调用进程阻塞。当减小一个信号量的值时，内核会将所有试图将信号量值降低到 0 之下的操作阻塞。类似的，如果信号量的当前值不为 0，那么等待信号量的值等于 0 的调用进程将会发生阻塞。不管是何种情况，调用进程会一直保持阻塞直到其他一些进程将信号量的值修改为一个允许这些操作继续向前的值，在那个时刻内核会唤醒被阻塞的进程。图 47-1 显示了使用一个信号量来同步两个交替将信号量的值在 0 和 1 之间切换的进程的动作。

在控制进程的动作方面，信号量本身并没有任何意义，它的意义仅由使用信号量的进程赋予其的关联关系来确定。一般来讲，进程之间会达成协议将一个信号量与一种共享资源关联起来，如一块共享内存区域。信号量还有其他用途，如在 fork()之后同步父进程和子进程。（在 24.5 节中介绍了如何使用信号量来完成同样的任务。）

更多资源请访问稀酷客(www.ckook.com)

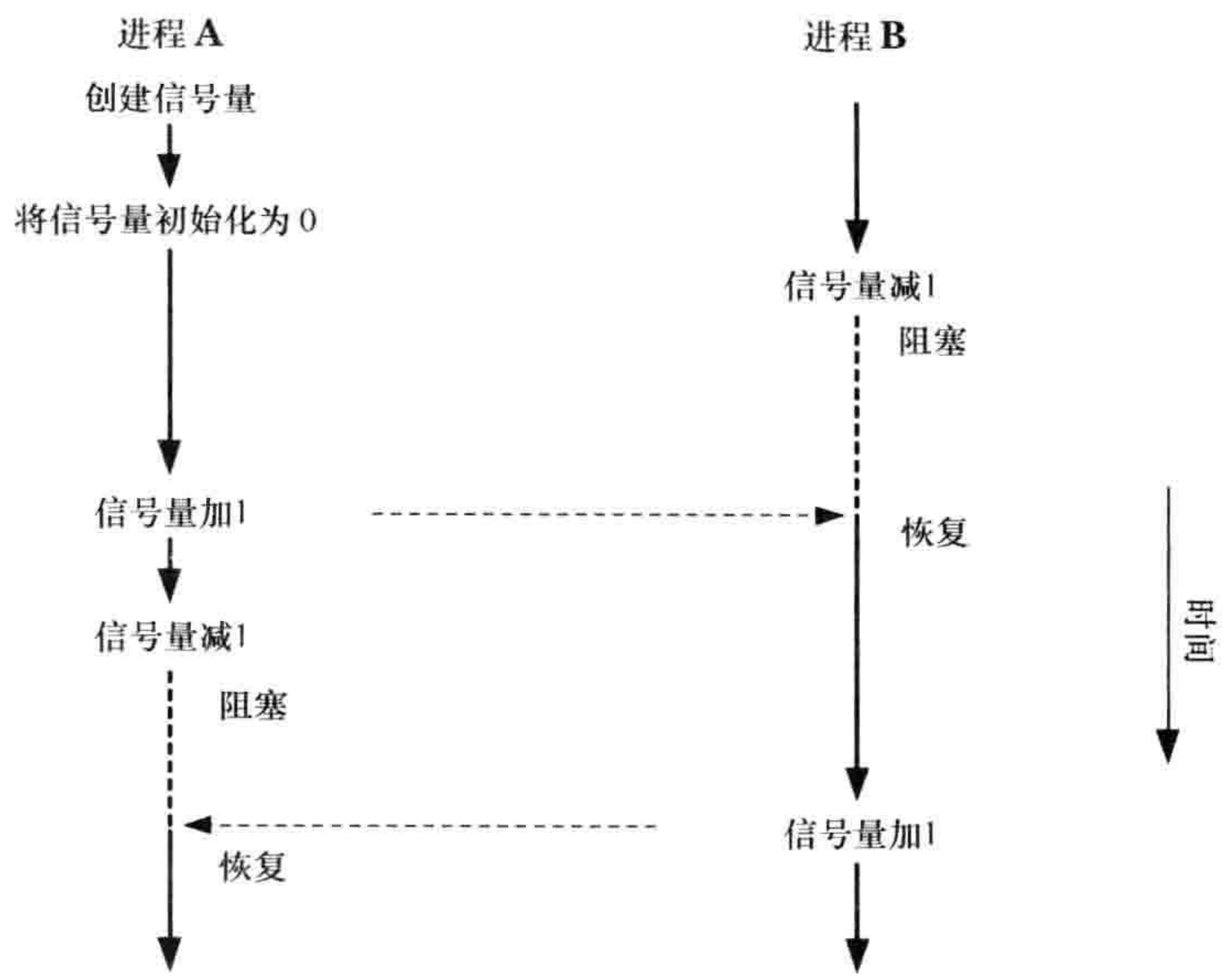


图 47-1　使用信号量同步两个进程

47.1　概述

使用 System V 信号量的常规步骤如下。

- 使用 semget()创建或打开一个信号量集。
- 使用 semctl() SETVAL 或 SETALL 操作初始化集合中的信号量。（只有一个进程需要完成这个任务。）
- 使用 semop()操作信号量值。使用信号量的进程通常会使用这些操作来表示一种共享资源的获取和释放。
- 当所有进程都不再需要使用信号量集之后使用 semctl() IPC_RMID 操作删除这个集合。（只有一个进程需要完成这个任务。）

大多数操作系统都为应用程序提供了一些信号量原语。但 System V 信号量表现出了不同寻常的复杂性，因为它们的分配是以备称为信号量集的组为单位进行的。在使用 semget()系统调用创建集合的时候需要指定集合中的信号量数量。虽然在同一时刻通常只操作一个信号量，但通过 semop()系统调用可以原子地在同一个集合中的多个信号量之上执行一组操作。

由于 System V 信号量的创建和初始化是在不同的步骤之后完成的，因此当两个进程同时都试图执行这两个步骤时就会出现竞争条件。要描述清楚这种竞争条件以及如何避免出现这种情况需要先对 semctl()进行介绍，然后再对 semop()进行介绍，这意味着在掌握完全理解信号量所需的所有细节信息之前还需要对很多材料进行学习。

与此同时，程序清单 47-1 给出了一个简单的例子，它演示了各种信号量系统调用的用法。这个程序可以在两种模式下运行。

- 当在命令行参数中传入一个整数时程序会创建一个只包含一个信号量的新信号量集并将信号量值初始化为通过命令行参数传入的值。程序会打印出这个新信号量集的标识符。
- 当在命令行参数中传入两个整数时程序会将它们看成是（按照顺序）一个既有信号量集的标识符和一个将被加到集合中第一个信号量（序号为 0）上的值。程序会在该信

更多资源请访问稀酷客(www.ckook.com)

号量上执行指定的操作。为了能够监控信号量操作，程序在操作之前和之后都会打印出消息。每条消息都以进程 ID 打头，这样就可以对这个程序的多个实例所产生的输出进行区分了。

下面的 shell 会话日志演示了程序清单 47-1 中的程序的用法。下面首先创建一个信号量并将其初始化为 0。

```
$ ./svsem_demo 0
Semaphore ID = 98307                    ID of new semaphore set
```

然后执行一个后台命令将信号量值减去 2。

```
$ ./svsem_demo 98307 -2 &
23338: about to semop at  10:19:42
[1] 23338
```

这个命令会阻塞，因为无法将信号量的值减到小于 0。现在执行一个命令将信号量值加上 3。

```
$ ./svsem_demo 98307 +3
23339: about to semop at  10:19:55
23339: semop completed at 10:19:55
23338: semop completed at 10:19:55
[1]+  Done                ./svsem_demo 98307 -2
```

这个信号量增加操作会立即成功，并且会导致后台命令中的信号量缩减操作能够向前执行，因为在执行该操作之后不会导致信号量值小于 0。

程序清单 47-1：创建和操作 System V 信号量

svsem/svsem_demo.c

```
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include "curr_time.h"                  /* Declaration of currTime() */
#include "semun.h"                      /* Definition of semun union */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int semid;

    if (argc < 2 || argc > 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s init-value\n"
                 "   or: %s semid operation\n", argv[0], argv[0]);

    if (argc == 2) {            /* Create and initialize semaphore */
        union semun arg;

        semid = semget(IPC_PRIVATE, 1, S_IRUSR | S_IWUSR);
        if (semid == -1)
            errExit("semid");

        arg.val = getInt(argv[1], 0, "init-value");
        if (semctl(semid, /* semnum= */ 0, SETVAL, arg) == -1)
            errExit("semctl");

        printf("Semaphore ID = %d\n", semid);

    } else {                    /* Perform an operation on first semaphore */
```


更多资源请访问稀酷客(www.ckook.com)

```
    struct sembuf sop;              /* Structure defining operation */

    semid = getInt(argv[1], 0, "semid");

    sop.sem_num = 0;                /* Specifies first semaphore in set */
    sop.sem_op = getInt(argv[2], 0, "operation");
                                    /* Add, subtract, or wait for 0 */
    sop.sem_flg = 0;                /* No special options for operation */

    printf("%ld: about to semop at  %s\n", (long) getpid(), currTime("%T"));
    if (semop(semid, &sop, 1) == -1)
        errExit("semop");

    printf("%ld: semop completed at %s\n", (long) getpid(), currTime("%T"));
    }

    exit(EXIT_SUCCESS);
}
```

svsem/svsem_demo.c

47.2 创建或打开一个信号量集

semget()系统调用创建一个新信号量集或获取一个既有集合的标识符。

```
#include <sys/types.h>        /* For portability */
#include <sys/sem.h>
```

```
int semget(key_t key, int nsems, int semflg);
```

Returns semaphore set identifier on success, or −1 on error

key 参数是使用 45.2 节中描述的其中一种方法生成的键（通常使用值 IPC_PRIVATE 或由 ftok()返回的键）。

如果使用 semget()创建一个新信号量集，那么 nsems 会指定集合中信号量的数量，并且其值必须大于 0。如果使用 semget()来获取一个既有集的标识符，那么 nsems 必须要小于或等于集合的大小（否则会发生 EINVAL 错误）。无法修改一个既有集中的信号量数量。

semflg 参数是一个位掩码，它指定了施加于新信号量集之上的权限或需检查的一个既有集合的权限。指定权限的方式与为文件指定权限的方式是一样的（表 15-4）。此外，在 semflg 中可以通过对下列标记中的零个或多个取 OR 来控制 semget()的操作。

IPC_CREAT

如果不存在与指定的 key 相关联的信号量集，那么就创建一个新集合。

IPC_EXCL

如果同时指定了 IPC_CREAT 并且与指定的 key 关联的信号量集已经存在，那么返回 EEXIST 错误。

45.1 节对这些标记进行了更加深入的介绍。

semget()在成功时会返回新信号量集或既有信号量集的标识符。后续引用单个信号量的系统调用必须要同时指定信号量集标识符和信号量在集合中的序号。一个集合中的信号量从 0 开始计数。

更多资源请访问稀酷客(www.ckook.com)

47.3 信号量控制操作

semctl()系统调用在一个信号量集或集合中的单个信号量上执行各种控制操作。

```
#include <sys/types.h>        /* For portability */
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ... /* union semun arg */);
```

Returns nonnegative integer on success (see text); returns −1 on error

semid 参数是操作所施加的信号量集的标识符。对于那些在单个信号量上执行的操作，semnum 参数标识出了集合中的具体信号量。对于其他操作则会忽略这个参数，并且可以将其设置为 0。cmd 参数指定了需执行的操作。

一些特定的操作需要向 semctl()传入第四个参数，在本节余下的部分中将这个参数命名为 arg。这个参数是一个 union，程序清单 47-2 给出了其定义。在程序中必须要显式地定义这个 union。程序清单 47-2 中的示例程序通过包含这个头文件来完成这个任务。

虽然将 semun union 的定义放在标准头文件中是比较明智的做法，但 SUSv3 要求程序员显式地定义这个 union。然而，一些 UNIX 实现在<sys/sem.h>中提供了这个定义。glibc 较早以前的版本（2.0 以下，包括 2.0）也提供了这个定义。为了与 SUSv3 保持一致，glibc 最近的版本并没有提供这个定义，并且通过将<sys/sem.h>中的_SEM_SEMUN_UNDEFINED 宏的值定义为 1 来表明这个事实（即使用 glibc 编译的应用程序可以通过测试这个宏来确定程序自己是否需要定义 semun union）。

程序清单 47-2：semun union 的定义

svsem/semun.h

```
#ifndef SEMUN_H
#define SEMUN_H                 /* Prevent accidental double inclusion */

#include <sys/types.h>          /* For portability */
#include <sys/sem.h>

union semun {                   /* Used in calls to semctl() */
    int                 val;
    struct semid_ds *   buf;
    unsigned short *    array;
#if defined(__linux__)
    struct seminfo *    __buf;
#endif
};

#endif
```

svsem/semun.h

SUSv2 和 SUSv3 规定 semctl()的最后一个参数是可选的。但一些（主要是较早之前的）UNIX 实现（以及 glibc 的早期版本）将 semctl()的原型定义如下。

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

这意味着第四个参数是必需的，即使在那些不需要用到这个参数的情况下也是如此（如

更多资源请访问稀酷客(www.ckook.com)

下面描述的 IPC_RMID 和 GETVAL 操作）。为使程序能够完全可移植，在那些无需最后一个参数的 semctl()调用中需要传入一个哑参数。

在本节余下的部分中将介绍通过 cmd 参数可指定的各种控制操作。

常规控制操作

下面的操作与可应用于其他类型的 System V IPC 对象上的操作是一样的。所有这些操作都会忽略 semnum 参数。45.3 节提供了有关这些操作的更多细节，包括调用进程所需的特权和权限。

IPC_RMID

立即删除信号量集及其关联的 semid_ds 数据结构。所有因在 semop()调用中等待这个集合中的信号量而阻塞的进程都会立即被唤醒，semop()会报告错误 EIDRM。这个操作无需 arg 参数。

IPC_STAT

在 arg.buf 指向的缓冲器中放置一份与这个信号量集相关联的 semid_ds 数据结构的副本。47.4 节将对 semid_ds 结构进行介绍。

IPC_SET

使用 arg.buf 指向的缓冲器中的值来更新与这个信号量集相关联的 semid_ds 数据结构中选中的字段。

获取和初始化信号量值

下面的操作可以获取或初始化一个集合中的单个或所有信号量的值。获取一个信号量的值需具备在信号量上的读权限，而初始化该值则需要修改（写）权限。

GETVAL

semctl()返回由 semid 指定的信号量集中第 semnum 个信号量的值。这个操作无需 arg 参数。

SETVAL

将由 semid 指定的信号量集中第 semnum 个信号量的值初始化为 arg.val。

GETALL

获取由 semid 指向的信号量集中所有信号量的值并将它们放在 arg.array 指向的数组中。程序员必须要确保该数组具备足够的空间。（通过由 IPC_STAT 操作返回的 semid_ds 数据结构中的 sem_nsems 字段可以获取集合中的信号量数量。）这个操作将忽略 semnum 参数。程序清单 47-3 给出了一个使用 GETALL 操作的例子。

SETALL

使用 arg.array 指向的数组中的值初始化 semid 指向的集合中的所有信号量。这个操作将忽略 semnum 参数。程序清单 47-4 演示了 SETALL 操作的用法。

如果存在一个进程正在等待在由 SETVAL 或 SETALL 操作所修改的信号量上执行一个操作并且对信号量所做的变更将允许该操作继续向前执行，那么内核就会唤醒该进程。

使用或 SETALL 修改一个信号量的值会在所有进程中清除该信号量的撤销条目。在 47.8 节中将会对信号量撤销条目予以介绍。

注意 GETVAL 和 GETALL 返回的信息在调用进程使用它们时可能已经过期了。所有依赖

更多资源请访问稀酷客(www.ckook.com)

由这些操作返回的信息保持不变这个条件的程序都可能会遇到检查时（time-of-check）和使用时（time-of-use）的竞争条件（参见 38.6）。

获取单个信号量的信息

下面的操作返回（通过函数结果值）semid 引用的集合中第 semnum 个信号量的信息。所有这些操作都需要在信号量集合中具备读权限，并且无需 arg 参数。

GETPID

返回上一个在该信号量上执行 semop()的进程的进程 ID；这个值被称为 sempid 值。如果还没有进程在该信号量上执行过 semop()，那么就返回 0。

GETNCNT

返回当前等待该信号量的值增长的进程数；这个值被称为 semncnt 值。

GETZCNT

返回当前等待该信号量的值变成 0 的进程数；这个值被称为 semzcnt 值。

与上面介绍的 GETVAL 和 GETALL 操作一样，GETPID、GETNCNT 以及 GETZCNT 操作返回的信息在调用进程使用它们时可能已经过期了。

程序清单 47-3 演示了这三个操作的用法。

47.4 信号量关联数据结构

每个信号量集都有一个关联的 semid_ds 数据结构，其形式如下。

```
struct semid_ds {
    struct ipc_perm sem_perm;       /* Ownership and permissions */
    time_t          sem_otime;      /* Time of last semop() */
    time_t          sem_ctime;      /* Time of last change */
    unsigned long   sem_nsems;      /* Number of semaphores in set */
};
```

> SUSv3 要求实现定义上面的 semid_ds 结构中给出的所有字段。其他一些 UNIX 实现包含了额外的非标准字段。在 Linux 2.4 以及之后的版本上，sem_nsems 字段的类型为 unsigned long。SUSv3 将这个字段的类型规定为 unsigned short，并且在 Linux 2.2 以及大多数其他 UNIX 实现上也是这么定义的。

各种信号量系统调用会隐式地更新 semid_ds 结构中的字段，使用 semctl() IPC_SET 操作能够显式地更新 sem_perm 字段中的特定子字段，其细节信息如下。

sem_perm

在创建信号量集时按照 45.3 中所描述的那样初始化这个子结构中的字段。通过 IPC_SET 能够更新 uid、gid 以及 mode 子字段。

sem_otime

在创建信号量集时会将这个字段设置为 0，然后在每次成功的 semop()调用或当信号量值因 SEM_UNDO 操作而发生变更时将这个字段设置为当前时间（参见 47.8 节）。这个字段和 sem_ctime 的类型为 time_t，它们存储自新纪元到现在的秒数。

更多资源请访问稀酷客(www.ckook.com)

sem_ctime

在创建信号量时以及每个成功的 IPC_SET、SETALL 和 SETVAL 操作执行完毕之后将这个字段设置为当前时间。（在一些 UNIX 实现上，SETALL 和 SETVAL 操作不会修改 sem_ctime。）

sem_nsems

在创建集合时将这个字段的值初始化为集合中信号量的数量。

本节后面将介绍两个使用 semid_ds 数据结构和一些在 47.3 节中描述的 semctl()操作的例子。在 47.6 节中将演示这两个程序的用法。

监控一个信号量集

程序清单 47-3 使用了各种 semctl()操作来显示标识符为命令行参数值的既有信号量集的信息。这个程序首先显示了 semid_ds 数据结构中的时间字段，然后显示了集合中各个信号量的当前值及其 sempid、semncnt 和 semzcnt 值。

程序清单 47-3：一个信号量监控程序

svsem/svsem_mon.c

```
#include <sys/types.h>
#include <sys/sem.h>
#include <time.h>
#include "semun.h"                      /* Definition of semun union */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct semid_ds ds;
    union semun arg, dummy;             /* Fourth argument for semctl() */
    int semid, j;
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s semid\n", argv[0]);

    semid = getInt(argv[1], 0, "semid");

    arg.buf = &ds;
    if (semctl(semid, 0, IPC_STAT, arg) == -1)
        errExit("semctl");

    printf("Semaphore changed: %s", ctime(&ds.sem_ctime));
    printf("Last semop():      %s", ctime(&ds.sem_otime));

    /* Display per-semaphore information */

    arg.array = calloc(ds.sem_nsems, sizeof(arg.array[0]));
    if (arg.array == NULL)
        errExit("calloc");
    if (semctl(semid, 0, GETALL, arg) == -1)
        errExit("semctl-GETALL");

    printf("Sem #  Value  SEMPID  SEMNCNT  SEMZCNT\n");

    for (j = 0; j < ds.sem_nsems; j++)
        printf("%3d   %5d   %5d  %5d    %5d\n", j, arg.array[j],
                semctl(semid, j, GETPID, dummy),
```

```
                semctl(semid, j, GETNCNT, dummy),
                semctl(semid, j, GETZCNT, dummy));

    exit(EXIT_SUCCESS);
}
```

svsem/svsem_mon.c

初始化一个集合中的所有信号量

程序清单 47-4 为初始化一个既有集合中的所有信号量提供了一个命令行界面。第一个命令行参数是待初始化的信号量集的标识符。剩下的命令行参数指定了每个信号量所初始化的值（参数的数量必须要与集合中信号量的数量一致）。

程序清单 47-4：使用 SETALL 操作初始化一个 System V 信号量集

svsem/svsem_setall.c

```
#include <sys/types.h>
#include <sys/sem.h>
#include "semun.h"                      /* Definition of semun union */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct semid_ds ds;
    union semun arg;                    /* Fourth argument for semctl() */
    int j, semid;
    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s semid val...\n", argv[0]);

    semid = getInt(argv[1], 0, "semid");

    /* Obtain size of semaphore set */

    arg.buf = &ds;
    if (semctl(semid, 0, IPC_STAT, arg) == -1)
        errExit("semctl");

    if (ds.sem_nsems != argc - 2)
        cmdLineErr("Set contains %ld semaphores, but %d values were supplied\n",
                (long) ds.sem_nsems, argc - 2);

    /* Set up array of values; perform semaphore initialization */

    arg.array = calloc(ds.sem_nsems, sizeof(arg.array[0]));
    if (arg.array == NULL)
        errExit("calloc");

    for (j = 2; j < argc; j++)
        arg.array[j - 2] = getInt(argv[j], 0, "val");

    if (semctl(semid, 0, SETALL, arg) == -1)
        errExit("semctl-SETALL");
    printf("Semaphore values changed (PID=%ld)\n", (long) getpid());

    exit(EXIT_SUCCESS);
}
```

svsem/svsem_setall.c

47.5 信号量初始化

根据 SUSv3 的要求，实现无需对由 semget()创建的集合中的信号量值进行初始化。相反，程序员必须要使用 semctl()系统调用显式地初始化信号量。（在 Linux 上，semget()返回的信号量实际上会被初始化为 0，但为取得移植性就不能依赖于此。）前面曾经提及过，信号量的创建和初始化必须要通过单独的系统调用而不是单个原子步骤来完成的事实可能会导致在初始化一个信号量时出现竞争条件。本节将详细介绍竞争的本质并考虑一种基于[Stevens, 1999]提出的思想来避免出现这种情况的方法。

假设一个应用程序由多个地位平等的进程构成，这些进程使用一个信号量来协调相互之间的动作。由于无法保证哪个进程会首先使用信号量（这就是地位平等的含义），因此每个进程都必须要做好在信号量不存在时创建和初始化信号量的准备。基于此，可以考虑使用程序清单 47-5 中给出的代码。

程序清单 47-5：错误地初始化了一个 System V 信号量

from **svsem/svsem_bad_init.c**

```
/* Create a set containing 1 semaphore */

semid = semget(key, 1, IPC_CREAT | IPC_EXCL | perms);

if (semid != -1) {                  /* Successfully created the semaphore */
    union semun arg;

    /* XXXX */

    arg.val = 0;                    /* Initialize semaphore */
    if (semctl(semid, 0, SETVAL, arg) == -1)
        errExit("semctl");

} else {                            /* We didn't create the semaphore */
    if (errno != EEXIST) {          /* Unexpected error from semget() */
        errExit("semget");

    semid = semget(key, 1, perms);  /* Retrieve ID of existing set */
    if (semid == -1)
       errExit("semget");
}

/* Now perform some operation on the semaphore */

sops[0].sem_op = 1;                 /* Add 1... */
sops[0].sem_num = 0;                /* to semaphore 0 */
sops[0].sem_flg = 0;
if (semop(semid, sops, 1) == -1)
    errExit("semop");
```

from **svsem/svsem_bad_init.c**

程序清单 47-5 中的代码存在的问题是如果两个进程同时执行，如果第一个进程的时间片在代码中标记为 XXXX 处期满，那么就可能会出现图 47-2 中给出的顺序。这个顺序之所以存在问题有两个原因。首先，进程 B 在一个未初始化的信号量（即其值是一个任意值）上执行了一个 semop()。其次，进程 A 中的 semctl()调用覆盖了进程 B 所做出的变更。

更多资源请访问稀酷客(www.ckook.com)

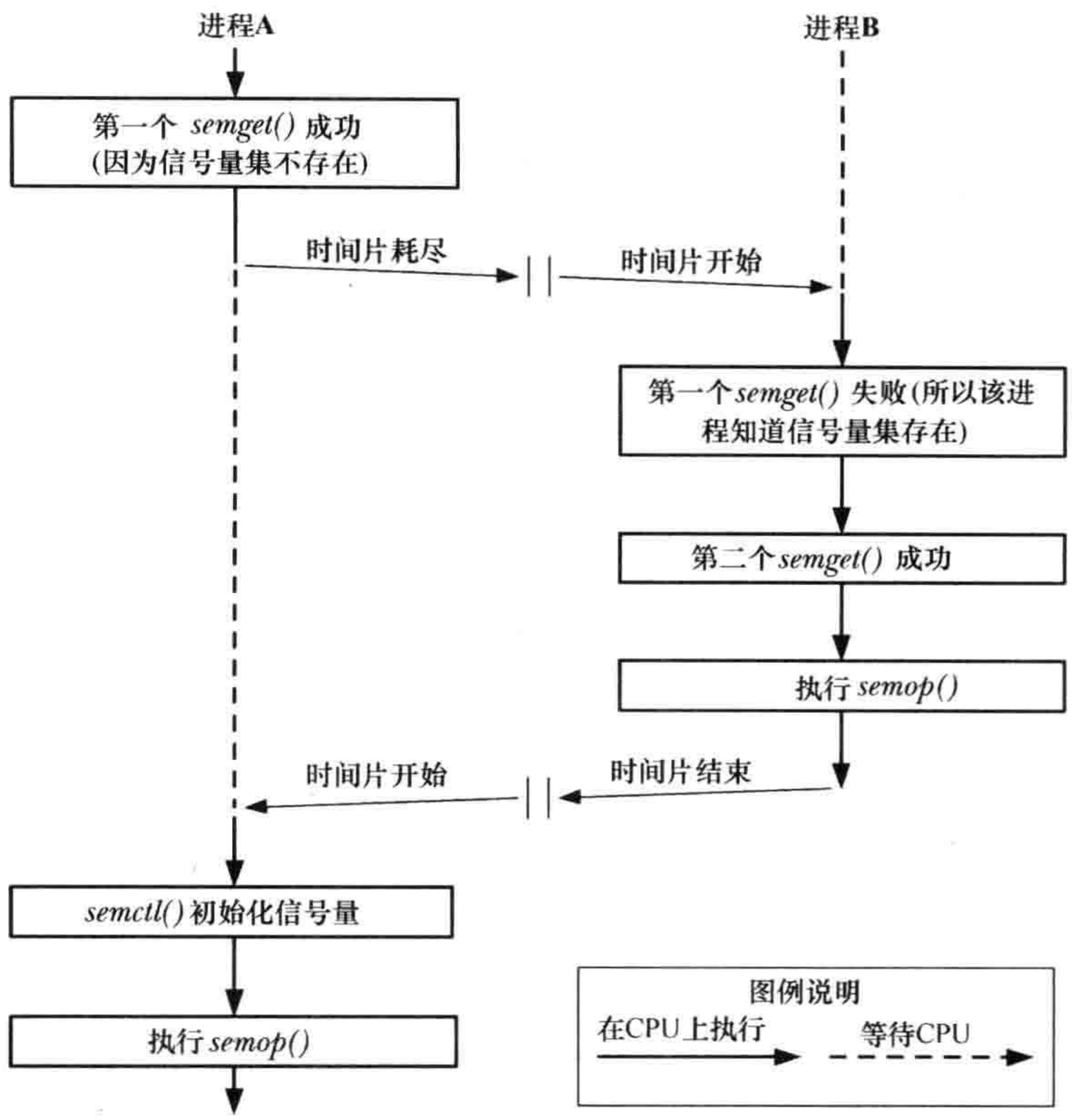


图 47-2：两个进程竞争初始化同一个信号量

这个问题的解决方案依赖于一个现已成为标准的特性，即与这个信号量集相关联的 semid_ds 数据结构中的 sem_otime 字段的初始化。在一个信号量集首次被创建时，sem_otime 字段会被初始化为 0，并且只有后续的 semop()调用才会修改这个字段的值。因此可以利用这个特性来消除上面描述的竞争条件，即只需要插入额外的代码来强制第二个进程（即没有创建信号量的那个进程）等待直到第一个进程既初始化了信号量又执行了一个更新 sem_otime 字段但不修改信号量的值的 semop()调用为止。程序清单 47-6 给出了修改之后的代码。

> 遗憾的是，正文中描述的初始化问题的解决方案无法在所有 UNIX 实现上正常工作。在一些现代 BSD 衍生版中，semop()不会更新 sem_otime 字段。

程序清单 47-6：初始化一个 System V 信号量

from **svsem/svsem_good_init.c**

```
semid = semget(key, 1, IPC_CREAT | IPC_EXCL | perms);

if (semid != -1) {                      /* Successfully created the semaphore */
    union semun arg;
    struct sembuf sop;

    arg.val = 0;                        /* So initialize it to 0 */
    if (semctl(semid, 0, SETVAL, arg) == -1)
        errExit("semctl");

    /* Perform a "no-op" semaphore operation - changes sem_otime
       so other processes can see we've initialized the set. */
```

```
        sop.sem_num = 0;                /* Operate on semaphore 0 */
        sop.sem_op = 0;                 /* Wait for value to equal 0 */
        sop.sem_flg = 0;
        if (semop(semid, &sop, 1) == -1)
            errExit("semop");

    } else {                            /* We didn't create the semaphore set */
        const int MAX_TRIES = 10;
        int j;
        union semun arg;
        struct semid_ds ds;

        if (errno != EEXIST) {          /* Unexpected error from semget() */
            errExit("semget");

        semid = semget(key, 1, perms);  /* Retrieve ID of existing set */
        if (semid == -1)
            errExit("semget");

        /* Wait until another process has called semop() */

        arg.buf = &ds;
        for (j = 0; j < MAX_TRIES; j++) {
            if (semctl(semid, 0, IPC_STAT, arg) == -1)
                errExit("semctl");
            if (ds.sem_otime != 0)      /* semop() performed? */
                break;                  /* Yes, quit loop */
            sleep(1);                   /* If not, wait and retry */
        }

        if (ds.sem_otime == 0)          /* Loop ran to completion! */
            fatal("Existing semaphore not initialized");
    }

    /* Now perform some operation on the semaphore */
```

from **svsem/svsem_good_init.c**

使用程序清单 47-6 中给出的技术的各种变体能够确保一个集合中的多个信号量正确地被初始化以及一个信号量被初始化为一个非零值。

并不是所有应用程序都需要使用这个及其负责的解决方案来解决竞争问题。如果能够确保一个进程在其他进程使用信号量之前创建和初始化信号量就无需使用这个解决方案。如父进程在创建与其共享信号量的子进程之前先创建和初始化信号量。在这种情况中，让第一个进程在调用完 semget()之后执行一个 semctl() SETVALSETALL 操作就足够了。

47.6 信号量操作

semop()系统调用在 semid 标识的信号量集中的信号量上执行一个或多个操作。

```
#include <sys/types.h>          /* For portability */
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned int nsops);
```

Returns 0 on success, or −1 on error

更多资源请访问稀酷客(www.ckook.com)

sops 参数是一个指向数组的指针，数组中包含了需要执行的操作，nsops 参数给出了数组的大小（数组至少需包含一个元素）。操作将会按照在数组中的顺序以原子的方式被执行。sops 数组中的元素是形式如下的结构。

```
struct sembuf {
    unsigned short sem_num;     /* Semaphore number */
    short          sem_op;      /* Operation to be performed */
    short          sem_flg;     /* Operation flags (IPC_NOWAIT and SEM_UNDO) */
};
```

sem_num 字段标识出了在集合中的哪个信号量上执行操作。sem_op 字段指定了需执行的操作。

- 如果 sem_op 大于 0，那么就将 sem_op 的值加到信号量值上，其结果是其他等待减小信号量值的进程可能会被唤醒并执行它们的操作。调用进程必须要具备在信号量上的修改（写）权限。
- 如果 sem_op 等于 0，那么就对信号量值进行检查以确定它当前是否等于 0。如果等于 0，那么操作将立即结束，否则 semop()就会阻塞直到信号量值变成 0 为止。调用进程必须要具备在信号量上的读权限。
- 如果 sem_op 小于 0，那么就将信号量值减去 sem_op。如果信号量的当前值大于或等于 sem_op 的绝对值，那么操作会立即结束。否则 semop()会阻塞直到信号量值增长到在执行操作之后不会导致出现负值的情况为止。调用进程必须要具备在信号量上的修改权限。

从语义上来讲，增加信号量值对应于使一种资源变得可用以便其他进程可以使用它，而减小信号量值则对应于预留（互斥地）进程需使用的资源。在减小一个信号量值时，如果信号量的值太低——即其他一些进程已经预留了这个资源——那么操作就会被阻塞。

当 semop()调用阻塞时，进程会保持阻塞直到发生下列某种情况为止。

- 另一个进程修改了信号量值使得待执行的操作能够继续向前。
- 一个信号中断了 semop()调用。发生这种情况时会返回 EINTR 错误。（在 21.5 节中指出过 semop()在被一个信号处理器中断之后是不会自动重启的。）
- 另一个进程删除了 semid 引用的信号量。发生这种情况时 semop()会返回 EIDRM 错误。

在特定信号量上执行一个操作时可以通过在相应的 sem_flg 字段中指定 IPC_NOWAIT 标记来防止 semop()阻塞。此时，如果 semop()本来要发生阻塞的话就会返回 EAGAIN 错误。

尽管通常一次只会操作一个信号量，但也可以通过一个 semop()调用在一个集合中的多个信号量上执行操作。这里需要指出的关键一点是这组操作的执行是原子的，即 semop()要么立即执行所有操作，要么就阻塞直到能够同时执行所有操作。

尽管作者所知晓的系统上的 semop()都按照数组中顺序来执行操作，但一些系统仍然通过文档显式地规定了这种行为，并且一些应用程序也依赖于这种行为。SUSv4 在文中显式地规定了这种行为。

程序清单 47-7 演示了如何使用 semop()在一个集合中的三个信号量上执行操作。根据信号量的当前值不同，在信号量 0 和 2 之上的操作可能无法立即往前执行。如果无法立即执行在信号量 0 上的操作，那么所有请求的操作都不会被执行，semop()会被阻塞。另一方面，如果可以立即执行在信号量 0 上的操作，但无法立即执行在信号量 2 上的操作，那么——由于指定了 IPC_NOWAIT 标记——所有请求的操作都不会被执行，并且 semop()会立即返回 EAGAIN 错误。

更多资源请访问棒酷客(www.ckook.com)

semtimedop()系统调用与 semop()执行的任务一样，但它多了一个 timeout 参数，通过这个参数可以指定调用所阻塞的时间上限。

```
#define _GNU_SOURCE
#include <sys/types.h>        /* For portability */
#include <sys/sem.h>

int semtimedop(int semid, struct sembuf *sops, unsigned int nsops,
               struct timespec *timeout);
```

Returns 0 on success, or −1 on error

timeout 参数是一个指向 timespec 结构（参见 23.4.2 节）的指针，通过这个结构能够将一个时间间隔表示为秒数和纳秒数。如果在信号量操作完成之前所等待的时间已经超过了规定的时间间隔，那么 semtimedop()会返回 EAGAIN 错误。如果将 timeout 指定为 NULL，那么 semtimedop()就与 semop()完全一样了。

与使用 setitimer()和 semop()相比，semtimedop()系统调用提供了一种更加高效的方式来为信号量操作设定一个超时时间。对于那些需要经常执行此类操作的应用程序（特别是一些数据库系统）来讲，这种方式所带来的性能上的提升是非常显著的。但 SUSv3 并没有规定 semtimedop()，并且只有其他一些 UNIX 实现提供了这个函数。

semtimedop()系统调用作为一个新特性出现在了 Linux 2.6 上，后来又被移回了 Linux 2.4，从内核 2.4.22 开始就存在这个函数了。

程序清单 47-7：使用 semop()在多个 System V 信号量上执行操作

```
struct sembuf sops[3];

sops[0].sem_num = 0;                /* Subtract 1 from semaphore 0 */
sops[0].sem_op = -1;
sops[0].sem_flg = 0;

sops[1].sem_num = 1;                /* Add 2 to semaphore 1 */
sops[1].sem_op = 2;
sops[1].sem_flg = 0;

sops[2].sem_num = 2;                /* Wait for semaphore 2 to equal 0 */
sops[2].sem_op = 0;
sops[2].sem_flg = IPC_NOWAIT;       /* But don't block if operation
                                       can't be performed immediately */
if (semop(semid, sops, 3) == -1) {
    if (errno == EAGAIN)            /* Semaphore 2 would have blocked */
        printf("Operation would have blocked\n");
    else
        errExit("semop");           /* Some other error */
}
```

示例程序

程序清单 47-8 为 semop()系统调用提供了一个命令行界面。这个程序接收的第一个参数是操作所施加的信号量集的标识符。

更多资源请访问稀酷客(www.ckook.com)

剩余的命令行参数指定了在单个 semop()调用中需要执行的一组信号量操作。单个命令行参数中的操作使用逗号分隔。每个操作的形式为下面中的一个。

- semnum+value：将 value 加到第 semnum 个信号量上。
- semnum-value：从第 semnum 个信号量上减去 value。
- semnum=0：测试第 semnum 信号量以确定它是否等于 0。

在每个操作的最后可以可选地包含一个 n、一个 u 或同时包含两者。字母 n 表示在这个操作的 sem_flg 值中包含 IPC_NOWAIT。字母 u 表示在 sem_flg 中包含 SEM_UNDO。（在 47.8 节中将会对 SEM_UNDO 标记进行介绍。）

下面的命令行在标识符为 0 的信号量集上执行了两个 semop()调用。

```
$ ./svsem_op 0 0=0 0-1,1-2n
```

第一个命令行参数规定 semop()调用等待直到第一个信号量值等于 0 为止。第二个参数规定 semop()调用从信号量 0 中减去 1 以及从信号量 1 中减去 2。信号量 0 上的操作的 sem_flg 为 0，信号量 1 上的操作的 sem_flg 是 IPC_NOWAIT。

程序清单 47-8：使用 semop()执行 System V 信号量操作

svsem/svsem_op.c

```
#include <sys/types.h>
#include <sys/sem.h>
#include <ctype.h>
#include "curr_time.h"          /* Declaration of currTime() */
#include "tlpi_hdr.h"

#define MAX_SEMOPS 1000         /* Maximum operations that we permit for
                                   a single semop() */

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s semid op[,op...] ...\n\n", progName);
    fprintf(stderr, "'op' is either: <sem#>{+|-}<value>[n][u]\n");
    fprintf(stderr, "            or: <sem#>=0[n]\n");
    fprintf(stderr, "       \"n\" means include IPC_NOWAIT in 'op'\n");
    fprintf(stderr, "       \"u\" means include SEM_UNDO in 'op'\n\n");
    fprintf(stderr, "The operations in each argument are "
                    "performed in a single semop() call\n\n");
    fprintf(stderr, "e.g.: %s 12345 0+1,1-2un\n", progName);
    fprintf(stderr, "      %s 12345 0=0n 1+1,2-1u 1=0\n", progName);
    exit(EXIT_FAILURE);
}

/* Parse comma-delimited operations in 'arg', returning them in the
   array 'sops'. Return number of operations as function result. */

static int
parseOps(char *arg, struct sembuf sops[])
{
    char *comma, *sign, *remaining, *flags;
    int numOps;                 /* Number of operations in 'arg' */

    for (numOps = 0, remaining = arg; ; numOps++) {
        if (numOps >= MAX_SEMOPS)
            cmdLineErr("Too many operations (maximum=%d): \"%s\"\n",
```

```
                MAX_SEMOPS, arg);

    if (*remaining == '\0')
        fatal("Trailing comma or empty argument: \"%s\"", arg);
    if (!isdigit((unsigned char) *remaining))
        cmdLineErr("Expected initial digit: \"%s\"\n", arg);

    sops[numOps].sem_num = strtol(remaining, &sign, 10);

    if (*sign == '\0' || strchr("+-=", *sign) == NULL)
        cmdLineErr("Expected '+', '-', or '=' in \"%s\"\n", arg);
    if (!isdigit((unsigned char) *(sign + 1)))
        cmdLineErr("Expected digit after '%c' in \"%s\"\n", *sign, arg);

    sops[numOps].sem_op = strtol(sign + 1, &flags, 10);
    if (*sign == '-')                       /* Reverse sign of operation */
        sops[numOps].sem_op = - sops[numOps].sem_op;
    else if (*sign == '=')                  /* Should be '=0' */
        if (sops[numOps].sem_op != 0)
            cmdLineErr("Expected \"=0\" in \"%s\"\n", arg);

    sops[numOps].sem_flg = 0;
    for (;; flags++) {
        if (*flags == 'n')
            sops[numOps].sem_flg |= IPC_NOWAIT;
        else if (*flags == 'u')
            sops[numOps].sem_flg |= SEM_UNDO;
        else
            break;
    }

    if (*flags != ',' && *flags != '\0')
        cmdLineErr("Bad trailing character (%c) in \"%s\"\n", *flags, arg);

    comma = strchr(remaining, ',');
    if (comma == NULL)
        break;                              /* No comma --> no more ops */
    else
        remaining = comma + 1;
}

return numOps + 1;
}

int
main(int argc, char *argv[])
{
    struct sembuf sops[MAX_SEMOPS];
    int ind, nsops;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageError(argv[0]);

    for (ind = 2; argv[ind] != NULL; ind++) {
        nsops = parseOps(argv[ind], sops);

        printf("%5ld, %s: about to semop()  [%s]\n", (long) getpid(),
                currTime("%T"), argv[ind]);

        if (semop(getInt(argv[1], 0, "semid"), sops, nsops) == -1)
```

```
            errExit("semop (PID=%ld)", (long) getpid());

        printf("%5ld, %s: semop() completed [%s]\n", (long) getpid(),
                currTime("%T"), argv[ind]);
    }

    exit(EXIT_SUCCESS);
}
```

svsem/svsem_op.c

使用程序清单 47-8 中的程序以及本章中给出的其他程序可以研究 System V 信号量的操作，如下面的 shell 会话所示。下面首先使用一个进程创建了一个包含两个信号量的信号量集并将这两个信号量的值分别初始化为 1 和 0。

```
$ ./svsem_create -p 2
32769                                          ID of semaphore set
$ ./svsem_setall 32769 1 0
Semaphore values changed (PID=3658)
```

本章并没有给出 svsem/svsem_create.c 程序的代码，读者可以在本章随带的源代码中找到这个程序的代码。这个程序在信号量上执行的功能与程序清单 46-1 中的程序在消息队列上执行的功能一样，即它创建了一个信号量集。唯一值得注意的差别是 svsem_create.c 额外接收了一个参数，该参数规定了所创建的信号量集的大小。

接下来在后台启动三个程序清单 47-8 中给出的程序实例来在信号量集上执行 semop()操作。程序会在执行每个信号量操作之前和之后都打印出消息。这些消息包括时间（这样就能够看到每个操作何时开始和何时结束）和进程 ID（这样就能够跟踪程序的多个实例的操作）。第一个命令要求将两个信号量值都减去 1。

```
$ ./svsem_op 32769 0-1,1-1 &                   Operation 1
 3659, 16:02:05: about to semop()  [0-1,1-1]
[1] 3659
```

从上面的输出中可以看出这个程序打印出了一条消息，指出就要执行 semop()操作了，但并没有打印出更多的消息，这是因为 semop()调用被阻塞了。这个调用之所以被阻塞是因为信号量 1 的值为 0。

接着执行一个命令要求将信号量 1 的值减去 1。

```
$ ./svsem_op 32769 1-1 &                       Operation 2
 3660, 16:02:22: about to semop()  [1-1]
[2] 3660
```

这个命令也会阻塞。接着执行一个命令等待信号量 0 的值等于 0。

```
$ ./svsem_op 32769 0=0 &                       Operation 3
 3661, 16:02:27: about to semop()  [0=0]
[3] 3661
```

这个命令再次阻塞了，这是因为信号量 0 的值当前为 1。

现在使用程序清单 47-3 中的程序来检查信号量集。

```
$ ./svsem_mon 32769
Semaphore changed: Sun Jul 25 16:01:53 2010
Last semop():      Thu Jan  1 01:00:00 1970
Sem #  Value  SEMPID  SEMNCNT  SEMZCNT
  0       1       0       1        1
  1       0       0       2        0
```

在一个信号量集被创建之后，其关联 semid_ds 数据结构的 sem_otime 字段会被初始化为 0。日历时间值 0 对应于新纪元的起点（参见 10.1 节），并且 ctime()会将这个值显示为 1 AM, 1 January 1970，这是因为本地时区为 Central Europe，它比 UTC 早一个小时。

更仔细地检查一下输出可以发现信号量 0 的 semncnt 值为 1，这是因为操作 1 正在等待减小信号量值，而 semzcnt 值为 1，这是因为操作 3 正在等待这个信号量的值等于 0。信号量 1 的 semncnt 值为 2，它反映出了操作 1 和操作 2 正在等于减小这个信号量值的事实。

接着在信号量集上尝试执行一个非阻塞操作。这个操作等于信号量 0 的值等于 0。由于无法立即执行这个操作，因此 semop()会返回 EAGAIN 错误。

```
$ ./svsem_op 32769 0=0n                                    Operation 4
 3673, 16:03:13: about to semop()  [0=0n]
ERROR [EAGAIN/EWOULDBLOCK Resource temporarily unavailable] semop (PID=3673)
```

现在向信号量 1 加上 1。这会导致之前两个被阻塞的操作（1 和 3）能够继续往前执行。

```
$ ./svsem_op 32769 1+1                                     Operation 5
 3674, 16:03:29: about to semop()  [1+1]
 3659, 16:03:29: semop() completed [0-1,1-1]               Operation 1 completes
 3661, 16:03:29: semop() completed [0=0]                   Operation 3 completes
 3674, 16:03:29: semop() completed [1+1]                   Operation 5 completes
[1]   Done                ./svsem_op 32769 0-1,1-1
[3]+  Done                ./svsem_op 32769 0=0
```

当使用监控程序来观察信号量集的状态时可以发现其关联 semid_ds 数据结构的 sem_otime 字段已经被更新了，并且两个信号量的 sempid 值也被更新了。此外，还可以看出信号量 1 的 semncnt 值为 1，这是因为操作 2 仍然被阻塞着，它等待减小这个信号量的值。

```
$ ./svsem_mon 32769
Semaphore changed: Sun Jul 25 16:01:53 2010
Last semop():      Sun Jul 25 16:03:29 2010
Sem #  Value  SEMPID  SEMNCNT  SEMZCNT
  0       0    3661      0        0
  1       0    3659      1        0
```

从上面的输出中可以看出 sem_otime 字段的值已经被更新过了。此外，还可以看出最近操作信号量 0 的进程的进程 ID 为 3661（操作 3），最近操作信号量 1 的进程的进程 ID 为 3659（操作 1）。

最后删除信号量集。这将会导致仍被阻塞的操作 2 返回 EIDRM 错误。

```
$ ./svsem_rm 32769
ERROR [EIDRM Identifier removed] semop (PID=3660)
```

本章并没有给出 svsem/svsem_rm.c 程序的源代码，读者可以在本章附带的源代码中找到这个程序的代码。这个程序删除通过命令行参数指定的信号量集。

47.7 多个阻塞信号量操作的处理

如果多个因减小一个信号量值而发生阻塞的进程对该信号量减去的值是一样的，那么当条件允许时到底哪个进程会首先被允许执行操作是不确定的（即哪个进程能够执行操作依赖于各个内核自己的进程调度算法）。

另一方面，如果多个因减小一个信号量值而发生阻塞的进程对该信号量减去的值是不同的，那么会按照先满足条件先服务的顺序来进行。假设一个信号量的当值为 0，进程 A 请求

更多资源请访问稀酷客(www.ckook.com)

将信号量值减去 2，然后进程 B 请求将信号量值减去 1。如果第三个进程将信号量值加上了 1，那么进程 B 首先会被解除阻塞并执行它的操作，即使进程 A 首先请求在该信号量上执行操作也一样。在一个糟糕的应用程序设计中，这种场景可能会导致饿死情况的发生，即一个进程因信号量的状态无法满足所请求的操作继续往前执行的条件而永远保持阻塞。回到本节的例子，考虑多个进程交替地调整信号量值使其值永远不会出现大于 1 的情况，这就会导致进程 A 永远保持阻塞。

当一个进程因试图在多个信号量上执行操作而发生阻塞时也可能会出现饿死的情况。考虑下面的这些在一组信号量上执行的操作，两个信号量的初始值都为 0。

1. 进程 A 请求将信号量 0 和 1 的值减去 1（阻塞）。
2. 进程 B 请求将信号量 0 的值减去 1（阻塞）。
3. 进程 C 将信号量 0 的值加上 1。

此刻，进程 B 解除阻塞并完成了它的请求，即使它发出请求的时间要晚于进程 A。同样，也可以设计出一个让进程 A 饿死的同时让其他进程调整和阻塞于单个信号量值的场景。

47.8 信号量撤销值

假设一个进程在调整完一个信号量值（如减小信号量值使之等于 0）之后终止了，不管是有意终止还是意外终止。在默认情况下，信号量值将不会发生变化。这样就可能会给其他使用这个信号量的进程带来问题，因为它们可能因等待这个信号量而被阻塞着——即等待已经被终止的进程撤销对信号量所做的变更。

为避免这种问题的发生，在通过 semop()修改一个信号量值时可以使用 SEM_UNDO 标记。当指定这个标记时，内核会记录信号量操作的效果，然后在进程终止时撤销这个操作。不管进程是正常终止还是非正常终止，撤销操作都会发生。

内核无需为所有使用 SEM_UNDO 的操作都保存一份记录。只需要记录一个进程在一个信号量上使用 SEM_UNDO 操作所作出的调整总和即可，它是一个被称为 semadj（信号量调整）的整数。当进程终止之后，所有需要做的就是从信号量的当前值上减去这个总和。

> 自 Linux 2.6 起，当指定了 CLONE_SYSVSEM 标记之后使用 clone()创建的进程（线程）会共享 semadj 值。之所以这样做是为了与 POSIX 线程的实现保持一致。NPTL 线程实现在 pthread_create()的实现中使用了 CLONE_SYSVSEM。

当使用 semctl() SETVAL 或 SETALL 操作设置一个信号量值时，所有使用这个信号量的进程中相应的 semadj 会被清空（即设置为 0）。这样做是合理的，因为直接设置一个信号量的值会破坏与 semadj 中维护的历史记录相关联的值。

通过 fork()创建的子进程不会继承其父进程的 semadj 值，因为对于子进程来讲撤销其父进程的信号量操作毫无意义。另一方面，semadj 值会在 exec()中得到保留。这样就能在使用 SEM_UNDO 调整一个信号量值之后通过 exec()执行一个不操作该信号量的程序，同时在进程终止时原子地调整该信号量。（这项技术可以允许另一个进程发现这个进程何时终止。）

SEM_UNDO 的效果举例

下面的 shell 会话日志显示了在两个信号量上执行操作的效果：一个操作使用了 SEM_UNDO 标记，另一个没有使用。下面首先创建一个包含两个信号量的集合。

更多资源请访问稀酷客(www.ckook.com)

```
$ ./svsem_create -p 2
131073
```

接着执行一个命令在两个信号量上都加上 1，然后终止。信号量 0 上的操作指定了 SEM_UNDO 标记。

```
$ ./svsem_op 131073 0+1u 1+1
 2248, 06:41:56: about to semop()
 2248, 06:41:56: semop() completed
```

现在使用程序清单 47-3 中的程序检查信号量的状态。

```
$ ./svsem_mon 131073
Semaphore changed: Sun Jul 25 06:41:34 2010
Last semop():      Sun Jul 25 06:41:56 2010
Sem #  Value  SEMPID  SEMNCNT  SEMZCNT
  0      0    2248      0        0
  1      1    2248      0        0
```

从上面输出的最后两行中的信号量值可以看出信号量 0 上的操作被撤销了，但信号量 1 上的操作没有被撤销。

SEM_UNDO 的限制

最后需要指出的是，SEM_UNDO 其实并没有其一开始看起来那样有用，原因有两个。一个原因是由于修改一个信号量通常对应于请求或释放一些共享资源，因此仅仅使用 SEM_UNDO 可能不足以允许一个多进程应用程序在一个进程异常终止时恢复。除非进程终止会原子地将共享资源的状态返回到一个一致的状态（在很多情况下是不可能的），否则撤销一个信号量操作可能不足以允许应用程序恢复。

第二个影响 SEM_UNDO 的实用性的因素是在一些情况下，当进程终止时无法对信号量进行调整。考虑下面应用于一个初始值为 0 的信号量上的操作。

1. 进程 A 将信号量值增加 2，并为该操作指定了 SEM_UNDO 标记。
2. 进程 B 将信号量值减去 1，因此信号量的值将变成 1。
3. 进程 A 终止。

此时就无法完全撤销进程 A 在第一步中的操作中所产生的效果，因为信号量的值太小了。解决这个问题的潜在方法有三种。

- 强制进程阻塞直到能够完成信号量调整。
- 尽可能地减小信号量的值（即减到 0）并退出。
- 退出，不执行任何信号量调整操作。

第一个解决方案是不可行的，因为它可能会导致一个即将终止的进程永远阻塞。Linux 采用了第二种解决方案。其他一些 UNIX 实现采纳了第三种解决方案。SUSv3 并没有规定一个实现应该采用哪种解决方案。

试图将一个信号量值提升到其许可的最大值 32767（第 47.10 节描述的 SEMVMX 限制）的撤销操作也会导致异常行为的发生。在这种情况下，内核总是会执行这个调整，从而（非法地）导致信号量的值大于 SEMVMX。

47.9 实现一个二元信号量协议

System V 信号量的 API 是比较复杂的，之所以会这样既因为对信号量值的调整量可以是

更多资源请访问稀酷客(www.ckook.com)

任意的，又因为信号量的分配和操作是以几何为单位的。但也正因为这些特性，System V 信号量提供的功能要多于常规应用程序所需的功能，因此以 System V 信号量为基础实现一个更加简单的协议（APIs）则是非常有用的。

一种常见的协议是二元信号量。一个二元信号量有两个值：可用（空闲）或预留（使用中）。二元信号量有两个操作。

- 预留：试图预留这个信号量以便互斥地使用。如果信号量已经被另一个进程预留了，那么将会阻塞直到信号量被释放为止。
- 释放：释放一个当前被预留的信号量，这样另一个进程就可以预留这个信号量了。

在学校教授的计算机科学中，这两个操作通常被称为 P 和 V，即这两个操作在荷兰语中的首字母。这种命名方式后来由荷兰计算机科学家 Edsger Dijkstra 所确定，他完成了很多有关信号量方面的早期理论工作。术语 down（减小信号量）和 up（增大信号量）也会被用到。POSIX 将这两个操作称为 wait 和 post。

有时候还会定义第三个操作。

- 有条件地预留：非阻塞地尝试预留这个信号量以便互斥地使用。如果信号量已经被预留了，那么立即返回一个状态标示出这个信号量不可用。

在实现二元信号量时必须要选择如何表示可用和预留状态以及如何实现上面的操作。读者稍微思考一下就会发现表示这些状态的最佳方式是使用值 1 表示空闲和值 0 表示预留，同时预留和释放操作分别为将信号量的值减 1 和加 1。

程序清单 47-9 和程序清单 47-10 给出了使用 System V 信号量实现二元信号量的一个实现。程序清单 47-9 中的头文件除了给出了实现中的函数的原型之外还声明了实现将会用到的两个全局布尔变量。bsUseSemUndo 变量控制实现是否在 semop()调用中使用 SEM_UNDO 标记。bsRetryOnEintr 变量控制实现是否在 semop()调用被信号中断之后自动重启该调用。

程序清单 47-9：binary_sems.c 的头文件

svsem/binary_sems.h

```
#ifndef BINARY_SEMS_H          /* Prevent accidental double inclusion */
#define BINARY_SEMS_H

#include "tlpi_hdr.h"

/* Variables controlling operation of functions below */

extern Boolean bsUseSemUndo;          /* Use SEM_UNDO during semop()? */
extern Boolean bsRetryOnEintr;        /* Retry if semop() interrupted by
                                         signal handler? */

int initSemAvailable(int semId, int semNum);

int initSemInUse(int semId, int semNum);

int reserveSem(int semId, int semNum);

int releaseSem(int semId, int semNum);

#endif
```

svsem/binary_sems.h

更多资源请访问稀酷客(www.ckook.com)

程序清单 47-10 给出了二元信号量函数的实现。这些实现中的每个函数都接收两个参数，它们分别标识出了信号量集和信号量在该集合中的序号。（这些函数既没有处理信号量集的创建和删除，也没有处理 47.5 节中描述的竞争条件。）在 48.4 节中给出的示例程序将会使用这些函数。

程序清单 47-10：使用 System V 信号量实现二元信号量

svsem/binary_sems.c

```
#include <sys/types.h>
#include <sys/sem.h>
#include "semun.h"                      /* Definition of semun union */
#include "binary_sems.h"

Boolean bsUseSemUndo = FALSE;
Boolean bsRetryOnEintr = TRUE;

int                     /* Initialize semaphore to 1 (i.e., "available") */
initSemAvailable(int semId, int semNum)
{
    union semun arg;

    arg.val = 1;
    return semctl(semId, semNum, SETVAL, arg);
}

int                     /* Initialize semaphore to 0 (i.e., "in use") */
initSemInUse(int semId, int semNum)
{
    union semun arg;

    arg.val = 0;
    return semctl(semId, semNum, SETVAL, arg);
}

/* Reserve semaphore (blocking), return 0 on success, or -1 with 'errno'
   set to EINTR if operation was interrupted by a signal handler */

int                     /* Reserve semaphore - decrement it by 1 */
reserveSem(int semId, int semNum)
{
    struct sembuf sops;

    sops.sem_num = semNum;
    sops.sem_op = -1;
    sops.sem_flg = bsUseSemUndo ? SEM_UNDO : 0;

    while (semop(semId, &sops, 1) == -1)
        if (errno != EINTR || !bsRetryOnEintr)
            return -1;

    return 0;
}
int                     /* Release semaphore - increment it by 1 */
releaseSem(int semId, int semNum)
{
    struct sembuf sops;
```

```
    sops.sem_num = semNum;
    sops.sem_op = 1;
    sops.sem_flg = bsUseSemUndo ? SEM_UNDO : 0;

    return semop(semId, &sops, 1);
}
```

svsem/binary_sems.c

47.10 信号量限制

大多数 UNIX 实现都对 System V 信号量的操作进行了各种各样的限制。下面列出了 Linux 上信号量的限制。括号中给出了当限制达到时会受影响的系统调用及其所返回的错误。

SEMAEM

在 semadj 总和中能够记录的最大值。SEMAEM 的值与 SEMVMX（稍后介绍）的值是一样的。（semop(), ERANGE）

SEMMNI

这是系统级别的一个限制，它限制了所能创建的信号量标识符的数量（换句话说是信号量集）。（semget(), ENOSPC）

SEMMSL

一个信号量集中能分配的信号量的最大数量。（semget(), EINVAL）

SEMMNS

这是系统级别的一个限制，它限制了所有信号量集中的信号量数量。系统上信号量的数量还受 SEMMNI 和 SEMMSL 的限制。实际上，SEMMNS 的默认值是这两个限制的默认值的乘积。（semget(), ENOSPC）

SEMOPM

每个 semop()调用能够执行的操作的最大数量。（semop(), E2BIG）

SEMVMX

一个信号量能取的最大值。（semop(), ERANGE）

大多数 UNIX 实现都定义了上面列出的限制。一些 UNIX 实现（不包括 Linux）在信号量撤销操作方面（参见 47.8 节）还定义了下面的限制。

SEMMNU

这是系统级别的一个限制，它限制了信号量撤销结构的总数量。撤销结构是分配用来存储 semadj 值的。

SEMUME

每个信号量撤销结构中撤销条目的最大数量。

在系统启动时，信号量限制会被设置成默认值。不同的内核版本中的默认值可能会不同。（一些内核厂商设置的默认值与 vanilla 内核设置的默认值可能会不同。）其中一些限制可以通过修改存储在 Linux 特有的/proc/sys/kernel/sem 文件中的值来改变。这个文件包含了四个用空

更多资源请访问稀酷客(www.ckook.com)

格分隔的数字，它们按序定义了 SEMMSL、SEMMNS、SEMOPM 以及 SEMMNI 限制。（SEMVMX 和 SEMAEM 限制是无法修改的，它们的值都被定义成 32767。）下面是 x86-32 系统上 Linux 2.6.31 定义的默认限制。

```
$ cd /proc/sys/kernel
$ cat sem
250     32000   32      128
```

Linux/proc 文件系统在三种 System V IPC 机制上所使用的格式是不一致的。对于消息队列和共享内存，每个可配置的额限制是通过单个文件来控制的。对于信号量则是由一个文件来保存所有可配置的限制。之所以这样是因为在这些 API 的开发过程中发生了一个历史性意外事件，并且由于兼容性的原因，这种现状已经很难改变了。

表 47-1 给出了 x86-32 架构上每个限制所能取的最大值。有关这张表格需要注意下列辅助信息。

表 47-1：System V 信号量限制

限　　制	最大值（x86-32）
SEMMNI	32768 (IPCMNI)
SEMMSL	65536
SEMMNS	2147483647 (INT_MAX)
SEMOPM	参见正文

- 可以将 SEMMSL 的值设置为一个大于 65536 的值，并且所创建的信号量集中最多可包含该数量的信号量。但无法使用 semop()调整集合中第 65536 个元素之后的元素。

由于在当前实现中存在一些限制，因此在实践中建议将一个信号量集容量的上限值设置为 8000 左右。

- SEMMNS 实际最大值是由系统上可用的 RAM 来控制的。
- SEMOPM 限制的最大值是由内核所使用的内存分配原语来确定的，建议的最大值是 1000。在实际使用中，在单个 semop()调用中执行过多的操作没有太大的用处。

Linux 特有的 semctl() IPC_INFO 操作返回一个类型为 seminfo 的结构，它包含了各种信号量限制的值。

```
union semun arg;
struct seminfo buf;

arg.__buf = &buf;
semctl(0, 0, IPC_INFO, arg);
```

相关的 Linux 特有的 SEM_INFO 操作会返回包含与信号量对象实际消耗的资源相关的信息的 seminfo 结构。本书随带的源代码中 svsem/svsem_info.c 文件给出了一个使用 SEM_INFO 的例子。

有关 IPC_INFO、SEM_INFO 以及 seminfo 结构的细节信息可以在 semctl(2)手册中找到。

47.11　System V 信号量的缺点

System V 信号量存在的很多缺点与消息队列（参见 46.9 节）的缺点是一样的，包括以下几点。

更多资源请访问稀酷客(www.ckook.com)

- 信号量是通过标识符而不是大多数 UNIX I/O 和 IPC 所采用的文件描述符来引用的。这使得执行诸如同时等待一个信号量和文件描述符的输入之类的操作就会变得比较困难。（通过创建一个子进程或线程来操作这个信号量并使用第 63 章中介绍的其中一种方法将消息写入一个被监控的管道以及其他文件描述符就可以解决这个难题。）
- 使用键而不是文件名来标识信号量增加了额外的编程复杂度。
- 创建和初始化信号量需要使用单独的系统调用意味着在一些情况下必须要做一些额外的编程工作来防止在初始化一个信号量时出现竞争条件。
- 内核不会维护引用一个信号量集的进程数量。这就给确定何时删除一个信号量集增加了难度并且难以确保一个不再使用的信号量集会被删除。
- System V 提供的编程接口过于复杂。在通常情况下，一个程序只会操作一个信号量。同时操作集合中多个信号量的能力有时侯是多余的。
- 信号量的操作存在诸多限制。这些限制是可配置的，但如果一个应用程序超出了默认限制的范围，那么在安装应用程序时就需要完成额外的工作了。

不管怎样，与消息队列所面临的情况不同，替代 System V 信号量的方案不多，其结果是在很多情况下都必须要用到它们。信号量的一个替代方案是记录锁，在第 55 章中将会对此予以介绍。此外，从内核 2.6 以及之后的版本开始，Linux 支持使用 POSIX 信号量来进行进程同步。第 53 章将会介绍 POSIX 信号量。

47.12 总结

System V 信号量允许进程同步它们的动作。这在当一个进程必须要获取对某些共享资源（如一块共享内存区域）的互斥性访问时是比较有用的。

信号量的创建和操作是以集合为单位的，一个集合包含一个或多个信号量。集合中的每个信号量都是一个整数，其值永远大于或等于 0。semop()系统调用允许调用者在一个信号量上加上一个整数、从一个信号量中减去一个整数、或等待一个信号量等于 0。后两个操作可能会导致调用者阻塞。

信号量实现无需对一个新信号量集中的成员进行初始化，因此应用程序就必须要在创建完之后对它们进行初始化。当一些地位平等的进程中任意一个进程试图创建和初始化信号量时就需要特别小心以防止因这两个步骤是通过单独的系统调用来完成的而可能出现的竞争条件。

如果多个进程对该信号量减去的值是一样的，那么当条件允许时到底哪个进程会首先被允许执行操作是不确定的。但如果多个进程对信号量减去的值是不同的，那么会按照先满足条件先服务的顺序来进行并且需要小心避免出现一个进程因信号量永远无法达到允许进程操作继续往前执行的值而饿死的情况。

SEM_UNDO 标记允许一个进程的信号量操作在进程终止时自动撤销。这对于防止出现进程意外终止而引起的信号量处于一个会导致其他进程因等待已终止的进程修改信号量值而永远阻塞情况来讲是比较有用的。

System V 信号量的分配和操作是以集合为单位的，并且对其增加和减小的数量可以是任意的。它们提供的功能要多于大多数应用程序所需的功能。对信号量常见的要求是单个二元信号量，它的取值只能是 0 和 1。本章介绍了如何以 System V 信号量为基础实现一个二元信号量。

更多资源请访问稀酷客(www.ckook.com)

更多信息

[Bovet & Cesati, 2005]和[Maxwell, 1999]提供了一些有关 Linux 上信号量实现的背景信息。[Dijkstra, 1968]是早期有关信号量理论的一片经典论文。

47.13 习题

47-1. 试验程序清单 47-8 中的程序（(svsem_op.c)来确认对 semop()系统调用的理解。

47-2. 修改程序清单 24-6 中的程序（fork_sig_sync.c）使之使用信号量来替代信号完成父进程和子进程之间的同步。

47-3. 试验程序清单 47-8 中的程序（svsem_op.c）和本章中提供的其他有关信号量的程序来检查当一个既有进程对一个信号量执行了一个 SEM_UNDO 调整时 sempid 值会发生什么情况。

47-4. 在程序清单 47-10 给出的代码（binary_sems.c）中增加一个 reserveSemNB()函数来使用 IPC_NOWAIT 标记实现有条件的预留操作。

47-5. 在 VMS 操作系统上，Digital 提供了一种类似于二元信号量的同步方法，它被称为事件标记（event flag）。一个事件标记可以取两个值 clear 和 set，并且在其之上可以执行下面 4 种操作：setEventFlag 来设置标记；clearEventFlag 来清除标记；waitForEventFlag 阻塞直到标记被设置；getFlagState 获取标记的当前状态。使用 System V 信号量为事件标记设计一种实现。这个实现要求上面每个函数都接收两个参数：一个是信号量标识符，一个是信号量序号。（在考虑 waitForEventFlag 操作时将会发现为 clear 和 set 状态取值不是一件容易的事情。）

47-6. 使用命名管道实现一个二元信号量协议。提供函数来预留、释放以及有条件地预留信号量。

47-7. 编写一个与程序清单 46-6 中给出的程序（svmsg_ls.c）类似的程序使之使用 semctl() SEM_INFO 和 SEM_STAT 操作来获取和显示系统上所有信号量集列表。

更多资源请访问稀酷客(www.ckook.com)

第48章

System V 共享内存

本章将介绍 System V 共享内存。共享内存允许两个或多个进程共享物理内存的同一块区域（通常被称为段）。由于一个共享内存段会成为一个进程用户空间内存的一部分，因此这种 IPC 机制无需内核介入。所有需要做的就是让一个进程将数据复制进共享内存中，并且这部分数据会对其他所有共享同一个段的进程可用。与管道或消息队列要求发送进程将数据从用户空间的缓冲区复制进内核内存和接收进程将数据从内核内存复制进用户空间的缓冲区的做法相比，这种 IPC 技术的速度更快。（每个进程也存在通过系统调用来执行复制操作的开销。）

另一方面，共享内存这种 IPC 机制不由内核控制意味着通常需要通过某些同步方法使得进程不会出现同时访问共享内存的情况（如两个进程同时执行更新操作或者一个进程在从共享内存中获取数据的同时另一个进程正在更新这些数据）。System V 信号量天生就是用来完成这种同步的一种方法。当然，还可以使用其他方法，如 POSIX 信号量（第 53 章）和文件锁（第 55 章）。

在 mmap()术语中，一块内存区域会被映射到一个地址，而在 System V 术语中，一个共享内存段是被附加到一个地址上的。这些术语是等价的，它们在术语上之所以存在差异是因为这两组 API 的起源不同。

48.1 概述

为使用一个共享内存段通常需要执行下面的步骤。

- 调用 shmget()创建一个新共享内存段或取得一个既有共享内存段的标识符（即由其他进程创建的共享内存段）。这个调用将返回后续调用中需要用到的共享内存标识符。
- 使用 shmat()来附上共享内存段，即使该段成为调用进程的虚拟内存的一部分。
- 此刻在程序中可以像对待其他可用内存那样对待这个共享内存段。为引用这块共享内存，程序需要使用由 shmat()调用返回的 addr 值，它是一个指向进程的虚拟地址空间中该共享内存段的起点的指针。

更多资源请访问稀酷客(www.ckook.com)

- 调用 shmdt()来分离共享内存段。在这个调用之后，进程就无法再引用这块共享内存了。这一步是可选的，并且在进程终止时会自动完成这一步。
- 调用 shmctl()来删除共享内存段。只有当当前所有附加内存段的进程都与之分离之后内存段才会被销毁。只有一个进程需要执行这一步。

48.2 创建或打开一个共享内存段

shmget()系统调用创建一个新共享内存段或获取一个既有段的标识符。新创建的内存段中的内容会被初始化为 0。

```
#include <sys/types.h>        /* For portability */
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

Returns shared memory segment identifier on success, or -1 on error

key 参数是使用在 45.2 节中介绍的其中一种方法（即通常是 IPC_PRIVATE 值或由 ftok()返回的键）生成的键。

当使用 shmget()创建一个新共享内存段时，size 则是一个正整数，它表示需分配的段的字节数。内核是以系统分页大小的整数倍来分配共享内存的，因此实际上 size 会被提升到最近的系统分页大小的整数倍。如果使用 shmget()来获取一个既有段的标识符，那么 size 对段不会产生任何效果，但它必须要小于或等于段的大小。

shmflg 参数执行的任务与其在其他 IPC get 调用中执行的任务一样，即指定施加于新共享内存段上的权限或需检查的既有内存段的权限（表 15-4）。此外，在 shmflg 中还可以对下列标记中的零个或多个取 OR 来控制 shmget()的操作。

IPC_CREAT

如果不存在与指定的 key 对应的段，那么就创建一个新段。

IPC_EXCL

如果同时指定了 IPC_CREAT 并且与指定的 key 对应的段已经存在，那么返回 EEXIST 错误。

45.1 节对上述标记进行了详细的介绍。此外，Linux 还允许使用下列非标准标记。

SHM_HUGETLB（自 Linux 2.6 起）

特权（CAP_IPC_LOCK）进程能够使用这个标记创建一个使用巨页（huge page）的共享内存段。巨页是很多现代硬件架构提供的一项特性用来管理使用超大分页尺寸的内存。（如 x86-32 允许使用 4MB 的分页大小来替代 4KB 的分页大小。）在那些拥有大量内存的系统上并且应用程序需要使用大量内存块时，使用巨页可以降低硬件内存管理单元的超前转换缓冲器（translation look-aside buffer，TLB）中的条目数量。这之所以会带来益处是因为 TLB 中的条目通常是一种稀缺资源。更多信息可参考内核源文件 Documentation/vm/ hugetlbpage.txt。

SHM_NORESERVE（自 Linux 2.6.15 起）

这个标记在 shmget()中所起的作用与 MAP_NORESERVE 标记在 mmap()中所起的作用一样，具体可参见 49.9 节。

shmget()在成功时返回新或既有共享内存段的标识符。

更多资源请访问稀酷客(www.ckook.com)

48.3 使用共享内存

shmat()系统调用将 shmid 标识的共享内存段附加到调用进程的虚拟地址空间中。

```
#include <sys/types.h>        /* For portability */
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Returns address at which shared memory is attached on success, or *(void *)* −1 on error

shmaddr 参数和 shmflg 位掩码参数中 SHM_RND 位的设置控制着段是如何被附加上去的。

- 如果 shmaddr 是 NULL，那么段会被附加到内核所选择的一个合适的地址处。这是附加一个段的优选方法。
- 如果 shmaddr 不为 NULL 并且没有设置 SHM_RND，那么段会被附加到由 shmaddr 指定的地址处，它必须是系统分页大小的一个倍数（否则会发生 EINVAL 错误）。
- 如果 shmaddr 不为 NULL 并且设置了 SHM_RND，那么段会被映射到的地址为在 shmaddr 中提供的地址被舍入到最近的常量 SHMLBA（shared memory low boundary address）的倍数。这个常量等于系统分页大小的某个倍数。将一个段附加到值为 SHMLBA 的倍数的地址处在一些架构上是有必要的，因为这样才能够提升 CPU 的快速缓冲性能和防止出现同一个段的不同附加操作在 CPU 快速缓冲中存在不一致的视图的情况。

在 x86 架构上，SHMLBA 的值与系统分页大小是一样的，这意味着此类缓冲不一致性不可能在那些架构上出现。

为 shmaddr 指定一个非 NULL 值（即上面列出的第二种和第三种情况）不是一种推荐的做法，其原因如下。

- 它降低了一个应用程序的可移植性。在一个 UNIX 实现上有效的地址在另一个实现上可能是无效的。
- 试图将一个共享内存段附加到一个正在使用中的特定地址处的操作会失败。例如，当一个应用程序（可能在一个库函数中）已经在该地址处附加了另一个段或创建一个内存映射时就会发生这种情况。

shmat()的函数结果是返回附加共享内存段的地址。开发人员可以像对待普通的 C 指针那样对待这个值，段与进程的虚拟内存的其他部分看起来毫无差异。通常会将 shmat()的返回值赋给一个指向某个由程序员定义的结构的指针以便在该段上设定该结构（参见程序清单 48-2 中给出的例子）。

要附加一个共享内存段以供只读访问，那么就需要在 shmflg 中指定 SHM_RDONLY 标记。试图更新只读段中的内容会导致段错误（SIGSEGV 信号）的发生。如果没有指定 SHM_RDONLY，那么就既可以读取内存又可以修改内存。

一个进程要附加一个共享内存段就需要在该段上具备读和写权限，除非指定了 SHM_RDONLY 标记，那样的话就只需要具备读权限即可。

更多资源请访问稀酷客(www.ckook.com)

在一个进程中可以多次附加同一个共享内存段，即使一个附加操作是只读的而另一个是读写的也没有关系。每个附加点上内存中的内容都是一样的，因为进程虚拟内存页表中的不同条目引用的是同样的内存物理页面。

最后一个可以在 shmflg 中指定的值是 SHM_REMAP。在指定了这个标记之后 shmaddr 的值必须为非 NULL。这个标记要求 shmat()调用替换起点在 shmaddr 处长度为共享内存段的长度的任何既有共享内存段或内存映射。一般来讲，如果试图将一个共享内存段附加到一个已经在用的地址范围时将会导致 EINVAL 错误的发生。SHM_REMAP 是一个非标准的 Linux 扩展。

表 48-1 对 shmat()的 shmflg 参数中能取 OR 的常量进行了总结。

表 48-1：shmat()的 shmflg 位掩码值

值	描　述
SHM_RDONLY	附加只读段
SHM_REMAP	替换位于 shmaddr 处的任意既有映射
SHM_RND	将 shmaddr 四舍五入为 SHMLBA 字节的倍数

当一个进程不再需要访问一个共享内存段时就可以调用 shmdt()来讲该段分离出其虚拟地址空间了。shmaddr 参数标识出了待分离的段，它应该是由之前的 shmat()调用返回的一个值。

```
#include <sys/types.h>        /* For portability */
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```

Returns 0 on success, or −1 on error

分离一个共享内存段与删除它是不同的。删除是通过 48.7 节中介绍的 shmctl() IPC_ RMID 操作来完成的。

通过 fork()创建的子进程会继承其父进程附加的共享内存段。因此，共享内存为父进程和子进程之间的通信提供了一种简单的 IPC 方法。

在一个 exec()中，所有附加的共享内存段都会被分离。在进程终止之后共享内存段也会自动被分离。

48.4 示例：通过共享内存传输数据

下面介绍一个使用 System V 共享内存和信号量的示例程序。这个应用程序由两个程序构成：写者和读者。写者从标准输入中读取数据块并将数据复制（“写”）到一个共享内存段中。读者将共享内存段中的数据块复制（“读”）到标准输出中。实际上，程序在某种程度上将共享内存当成了管道来处理。

两个程序使用了二元信号量协议（在 47.9 节中定义的 initSemAvailable()、initSemInUse()、reserveSem()以及 releaseSem()函数）中的一对 System V 信号量来确保：

- 一次只有一个进程访问共享内存段；
- 进程交替地访问段（即写者写入一些数据，然后读者读取这些数据，然后写者再次写入数据，以此类推）。

图 48-1 概述了这两个信号量的使用。注意写者对两个信号量进行了初始化，这样它就成

更多资源请访问稀酷客(www.ckook.com)

为两个程序中第一个能够访问共享内存段的程序了，即写者的信号量初始时是可用的，而读者的信号量初始时是正在被使用中的。

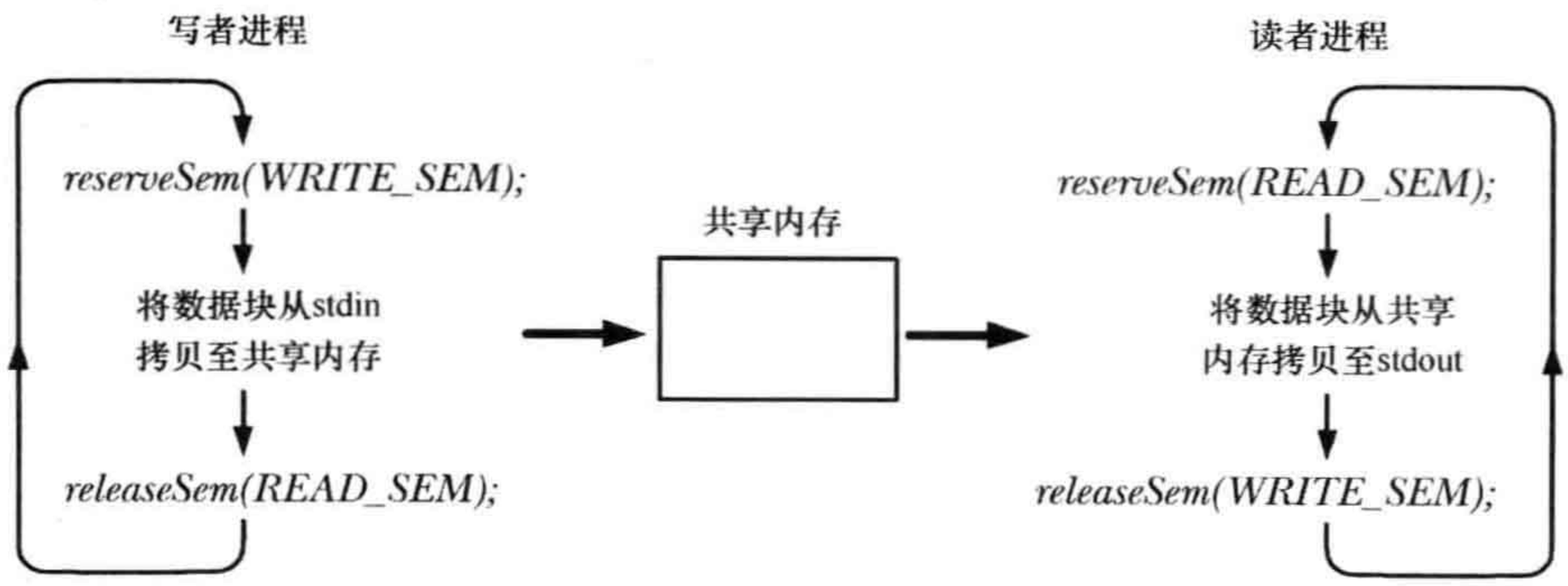


图 48-1：使用信号量确保对共享内存的互斥、交替的访问

这个应用程序的源代码由三个文件构成。第一个文件是由读者程序和写者程序共享的头文件，如程序清单 48-1 所示。这个头文件定义了 shmseg 结构，程序使用了这个结构来声明指向共享内存段的指针，这样就能给共享内存段中的字节规定一种结构。

程序清单 48-1：svshm_xfr_writer.c 和 svshm_xfr_reader.c 的头文件

svshm/svshm_xfr.h

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "binary_sems.h"        /* Declares our binary semaphore functions */
#include "tlpi_hdr.h"

#define SHM_KEY 0x1234          /* Key for shared memory segment */
#define SEM_KEY 0x5678          /* Key for semaphore set */

#define OBJ_PERMS (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)
                                /* Permissions for our IPC objects */

#define WRITE_SEM 0             /* Writer has access to shared memory */
#define READ_SEM 1              /* Reader has access to shared memory */

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

struct shmseg {                 /* Defines structure of shared memory segment */
    int cnt;                    /* Number of bytes used in 'buf' */
    char buf[BUF_SIZE];         /* Data being transferred */
};
```

svshm/svshm_xfr.h

程序清单 48-2 是写者程序。这个程序按序完成下列任务。

- 创建一个包含两个信号量的集合，写者和读者程序会使用这两个信号量来确保它们交替地访问共享内存段①。信号量被初始化为使写者首先访问共享内存段。由于是由写者来创建信号量集的，因此必须在启动读者之前启动写者。
- 创建共享内存段并将其附加到写者的虚拟地址空间中系统所选择的一个地址处②。

更多资源请访问稀酷客(www.ckook.com)

- 进入一个循环将数据从标准输入传输到共享内存段③。每个循环迭代需要按序完成下面的任务：
  -预留（减小）写者的信号量④。
  -从标准输入中读取数据并将数据复制到共享内存段⑤。
  -释放（增加）读者的信号量⑥。
- 当标准输入中没有可用的数据时循环终止⑦。在最后一次循环中，写者通过传递一个长度为 0 的数据块（shmp->cnt 为 0）来通知读者没有更多的数据了。
- 在退出循环时，写者再次预留其信号量，这样它就能知道读者已经完成了对共享内存的最后一次访问了⑧。写者随后删除了共享内存段和信号量集⑨。

程序清单 48-3 是读者程序。它将共享内存段中的数据块传输到标准输出中。读者按序完成了下面的任务。

- 获取写者程序创建的信号量集合共享内存段的 ID①。
- 附加共享内存段供只读访问②。
- 进入一个循环从共享内存段中传输数据③。在每个循环迭代中需要按序完成下面的任务。
  -预留（减小）读者的信号量④。
  -检查 shmp->cnt 是否为 0，如果为 0 就退出循环⑤。
  -将共享内存段中的数据块写入标准输出中⑥。
  -释放（增加）写者的信号量⑦。
- 在退出循环之后分离共享内存段⑧并释放写者的信号量⑨，这样写者程序就能够删除 IPC 对象了。

程序清单 48-2：将 stdin 中的数据块传输到一个 System V 共享内存段中

svshm/svshm_xfr_writer.c

```
#include "semun.h"              /* Definition of semun union */
#include "svshm_xfr.h"

int
main(int argc, char *argv[])
{
    int semid, shmid, bytes, xfrs;
    struct shmseg *shmp;
    union semun dummy;

①   semid = semget(SEM_KEY, 2, IPC_CREAT | OBJ_PERMS);
    if (semid == -1)
        errExit("semget");
    if (initSemAvailable(semid, WRITE_SEM) == -1)
        errExit("initSemAvailable");
    if (initSemInUse(semid, READ_SEM) == -1)
        errExit("initSemInUse");

②   shmid = shmget(SHM_KEY, sizeof(struct shmseg), IPC_CREAT | OBJ_PERMS);
    if (shmid == -1)
        errExit("shmget");

    shmp = shmat(shmid, NULL, 0);
    if (shmp == (void *) -1)
        errExit("shmat");

    /* Transfer blocks of data from stdin to shared memory */
```

```
③     for (xfrs = 0, bytes = 0; ; xfrs++, bytes += shmp->cnt) {
④         if (reserveSem(semid, WRITE_SEM) == -1)         /* Wait for our turn */
              errExit("reserveSem");

⑤         shmp->cnt = read(STDIN_FILENO, shmp->buf, BUF_SIZE);
          if (shmp->cnt == -1)
              errExit("read");

⑥         if (releaseSem(semid, READ_SEM) == -1)          /* Give reader a turn */
              errExit("releaseSem");

          /* Have we reached EOF? We test this after giving the reader
             a turn so that it can see the 0 value in shmp->cnt. */

⑦         if (shmp->cnt == 0)
              break;
      }

      /* Wait until reader has let us have one more turn. We then know
         reader has finished, and so we can delete the IPC objects. */

⑧     if (reserveSem(semid, WRITE_SEM) == -1)
          errExit("reserveSem");

⑨     if (semctl(semid, 0, IPC_RMID, dummy) == -1)
          errExit("semctl");
      if (shmdt(shmp) == -1)
          errExit("shmdt");
      if (shmctl(shmid, IPC_RMID, 0) == -1)
          errExit("shmctl");

      fprintf(stderr, "Sent %d bytes (%d xfrs)\n", bytes, xfrs);
      exit(EXIT_SUCCESS);
  }
```

svshm/svshm_xfr_writer.c

程序清单 48-3：将一个 System V 共享内存段中的数据块传输到 stdout 中

svshm/svshm_xfr_reader.c

```
#include "svshm_xfr.h"

int
main(int argc, char *argv[])
{
    int semid, shmid, xfrs, bytes;
    struct shmseg *shmp;

    /* Get IDs for semaphore set and shared memory created by writer */

①   semid = semget(SEM_KEY, 0, 0);
    if (semid == -1)
        errExit("semget");

    shmid  = shmget(SHM_KEY, 0, 0);
    if (shmid == -1)
        errExit("shmget");

②   shmp = shmat(shmid, NULL, SHM_RDONLY);
```


更多资源请访问稀酷客(www.ckook.com)

```
    if (shmp == (void *) -1)
        errExit("shmat");

    /* Transfer blocks of data from shared memory to stdout */

③   for (xfrs = 0, bytes = 0; ; xfrs++) {
④       if (reserveSem(semid, READ_SEM) == -1)          /* Wait for our turn */
            errExit("reserveSem");

⑤       if (shmp->cnt == 0)                     /* Writer encountered EOF */
            break;
        bytes += shmp->cnt;

⑥       if (write(STDOUT_FILENO, shmp->buf, shmp->cnt) != shmp->cnt)
            fatal("partial/failed write");

⑦       if (releaseSem(semid, WRITE_SEM) == -1)         /* Give writer a turn */
            errExit("releaseSem");
    }

⑧   if (shmdt(shmp) == -1)
        errExit("shmdt");

    /* Give writer one more turn, so it can clean up */

⑨   if (releaseSem(semid, WRITE_SEM) == -1)
        errExit("releaseSem");

    fprintf(stderr, "Received %d bytes (%d xfrs)\n", bytes, xfrs);
    exit(EXIT_SUCCESS);
}
```

svshm/svshm_xfr_reader.c

下面的 shell 会话演示了如何使用程序清单 48-2 和程序清单 46-9 中的程序。这里在调用读者时将文件/etc/services 作为输入，然后调用了读者并将其输出定向到另一个文件中。

```
$ wc -c /etc/services                                   Display size of test file
764360 /etc/services
$ ./svshm_xfr_writer < /etc/services &
[1] 9403
$ ./svshm_xfr_reader > out.txt
Received 764360 bytes (747 xfrs)                        Message from reader
Sent 764360 bytes (747 xfrs)                            Message from writer
[1]+  Done                ./svshm_xfr_writer < /etc/services
$ diff /etc/services out.txt
$
```

diff 命令不产生任何输出，这说明读者产生的输出文件中的内容与写者使用的输入文件中的内容是一样的。

48.5 共享内存在虚拟内存中的位置

在 6.3 节中介绍了一个进程的各个部分在虚拟内存中的布局。现在在介绍附加 System V 共享内存段的时候重温一下这个主题是比较有帮助的。如果遵循所推荐的方法，即允许内核选择在何处附加共享内存段，那么（在 x86-32 架构上）内存布局就会像图 48-2 中所示的那样，段被附加在向上增长的堆和向下增长的栈之间未被分配的空间中。为给堆和栈的增长腾出空

间，附加共享内存段的虚拟地址从 0x40000000 开始。内存映射（第 49 章）和共享库（第 41 和 42 章）也是被放置在这个区域中的。（共享内存映射和内存段默认被放置的位置可能会有些不同，这依赖于内核版本和进程的 RLIMIT_STACK 资源限制的设置。）

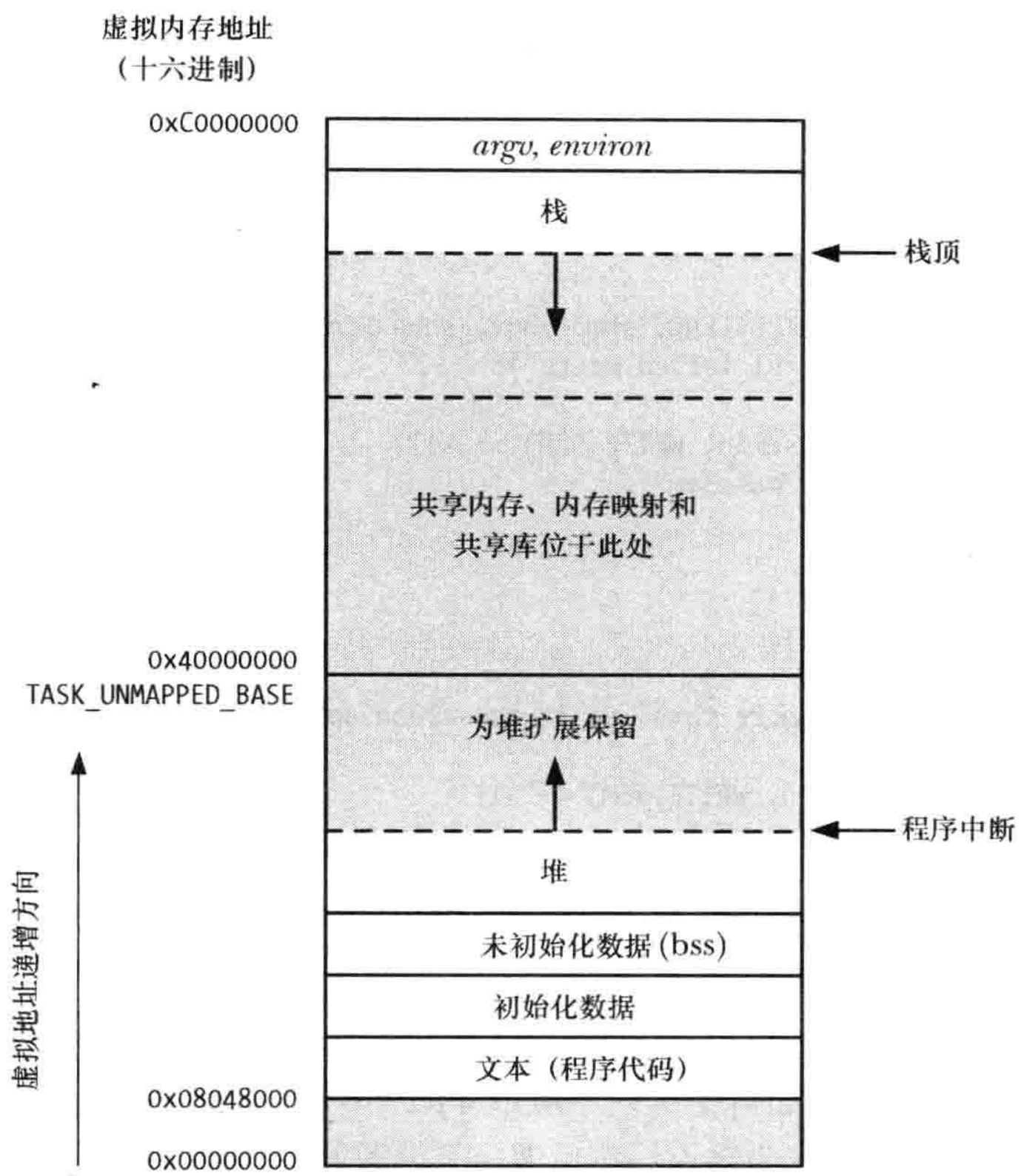


图 48-2：共享内存、内存映射、以及共享库的位置（x86-32）

地址 0x40000000 被定义成了内核常量 TASK_UNMAPPED_BASE。通过将这个常量定义成一个不同的值并且重建内核可以修改这个地址的值。

如果在调用 shmat()（或 mmap()）时采用了不推荐的方法，即显式地指定一个地址，那么一个共享内存段（或内存映射）可以被放置在低于 TASK_UNMAPPED_BASE 的地址处。

通过 Linux 特有的/proc/PID/maps 文件能够看到一个程序映射的共享内存段和共享库的位置，如下面的 shell 会话所示。

从内核 2.6.14 开始，Linux 还提供了/proc/PID/smaps 文件，它给出了有关一个进程中各个映射的内存消耗方面的更多信息。更多细节可参考 proc(5)手册。

在下面的 shell 会话中使用了三个在本章中没有给出的程序，读者可以在本书随带的源代码的 svshm 子目录中找到这三个程序。这些程序执行了下面的任务。

- svshm_create.c 程序创建了一个共享内存段。这个程序与在介绍消息队列（程序清单 46-1）和信号量时给出的相应程序接收同样的命令行选项，但它包含了一个额外的用来规定

段大小的参数。

- svshm_attach.c 程序附加通过其命令行参数指定的共享内存段。每个参数都由一对用分号隔开的数字构成，两个数字分别是共享内存标识符和附加地址。将附加地址指定为 0 表示系统应该选择地址。程序会显示出实际附加内存段的地址。此外，为提供更多的有用信息，程序还显示出了 SHMLBA 常量的值和运行这个程序的进程的进程 ID。
- svshm_rm.c 程序删除通过其命令行参数指定的共享内存段。

首先在 shell 中创建两个共享内存段（大小分别为 100kB 和 3200kB）。

```
$ ./svshm_create -p 102400
9633796
$ ./svshm_create -p 3276800
9666565
$ ./svshm_create -p 102400
1015817
$ ./svshm_create -p 3276800
1048586
```

然后启动一个将这两个段附加到由内核选择的地址处的程序。

```
$ ./svshm_attach 9633796:0 9666565:0
SHMLBA = 4096 (0x1000), PID = 9903
1: 9633796:0 ==> 0xb7f0d000
2: 9666565:0 ==> 0xb7bed000
Sleeping 5 seconds
```

从上面的输出中可以看出附加这两个段的地址。在程序完成睡眠之前挂起这个程序，然后检查相应的/proc/PID/maps 文件中的内容。

```
Type Control-Z to suspend program
[1]+  Stopped                 ./svshm_attach 9633796:0 9666565:0
$ cat /proc/9903/maps
```

程序清单 48-4 给出了 cat 命令产生的输出。

程序清单 48-4：示例/proc/PID/maps 的内容

```
  $ cat /proc/9903/maps
① 08048000-0804a000 r-xp 00000000 08:05 5526989  /home/mtk/svshm_attach
  0804a000-0804b000 r--p 00001000 08:05 5526989  /home/mtk/svshm_attach
  0804b000-0804c000 rw-p 00002000 08:05 5526989  /home/mtk/svshm_attach
② b7bed000-b7f0d000 rw-s 00000000 00:09 9666565  /SYSV00000000 (deleted)
  b7f0d000-b7f26000 rw-s 00000000 00:09 9633796  /SYSV00000000 (deleted)
  b7f26000-b7f27000 rw-p b7f26000 00:00 0
③ b7f27000-b8064000 r-xp 00000000 08:06 122031   /lib/libc-2.8.so
  b8064000-b8066000 r--p 0013d000 08:06 122031   /lib/libc-2.8.so
  b8066000-b8067000 rw-p 0013f000 08:06 122031   /lib/libc-2.8.so
  b8067000-b806b000 rw-p b8067000 00:00 0
  b8082000-b8083000 rw-p b8082000 00:00 0
④ b8083000-b809e000 r-xp 00000000 08:06 122125   /lib/ld-2.8.so
  b809e000-b809f000 r--p 0001a000 08:06 122125   /lib/ld-2.8.so
  b809f000-b80a0000 rw-p 0001b000 08:06 122125   /lib/ld-2.8.so
⑤ bfd8a000-bfda0000 rw-p bffea000 00:00 0        [stack]
⑥ ffffe000-fffff000 r-xp 00000000 00:00 0        [vdso]
```

从程序清单 48-4 中给出的/proc/PID/maps 文件的输出可以看出：

- 有三行是与主程序 shm_attach 相关的。它们对应于程序的文本和数据段①，其中第二行是一个保存程序所使用的字符串常量的只读分页。

更多资源请访问稀酷客(www.ckook.com)

- 有两行是与被附加的 System V 共享内存段相关的②。
- 有几行与两个共享库的段对应。其中一行是标准 C 库（libc-version.so）③，其他的则是在 41.4.3 节中介绍的动态链接器（ld-version.so）④。
- 一行被标记为[stack]，它对应于进程栈⑤。
- 一行包含的标签[vdso]⑥。它是用来表示 linux-gate 虚拟动态共享对象（DSO）的一个条目。这个条目只出现在了 2.6.12 以及之后的内核中。有关这个条目的更多信息可参考 http://www.trilithium.com/johan/2005/08/linux-gate/。

下面是/proc/PID/maps 文件中每行所包含的列，其顺序为从左至右。

1. 一对用连字符隔开的数字，它们分别表示内存段被映射到的虚拟地址范围（以十六进制表示）和段结尾之后第一个字节的地址。
2. 内存段的保护位和标记位。前三个字母表示段的保护位：读（r）、写（w）以及执行（x）。使用连字符（-）来替换其中任意字母表示禁用相应的保护位。最后一个字母表示内存段的映射标记，其取值要么是私有（p），要么是共享（s）。有关这些标记的详细解释可参见第 49.2 节中对 MAP_PRIVATE 和 MAP_SHARED 标记的描述。（System V 共享内存段总是被标记为共享。）
3. 段在对应的映射文件中的十六进制偏移量（以字节计数）。这个列以及随后的两列的含义在第 49 章中介绍 mmap()系统调用时会变得更加清晰。对于 System V 共享内存段来讲，偏移量总是为 0。
4. 相应的映射文件所位于的设备的设备号（主要和次要 ID）。
5. 映射文件的 i-node 号或 System V 共享内存段的标识符。
6. 与这个内存段相关联的文件名或其他标识标签。对于 System V 共享内存段来讲，这一列由字符串 SYSV 后面接上这个的段的 shmget()键（以十六进制表示）构成。在本例中，SYSV 后面跟着零，这是因为在创建段时使用了 IPC_PRIVATE 键（其值为 0）。System V 共享内存段的 SYSV 字段后面的字符串（deleted）是共享内存段实现的产物。这种段会被创建成不可见的 tmpfs 文件系统（14.10 节）中的映射文件，然后再被解除链接。共享匿名内存映射也是采用同样的方式实现的。（在第 49 章中将会介绍映射文件和共享匿名内存映射。）

48.6 在共享内存中存储指针

每个进程都可能会用到不同的共享库和内存映射，并且可能会附加不同的共享内存段集。因此如果遵循推荐的做法，让内核来选择将共享内存段附加到何处，那么一个段在各个进程中可能会被附加到不同的地址上。正因为这个原因，在共享内存段中存储指向段中其他地址的引用时应该使用（相对）偏移量，而不是（绝对）指针。

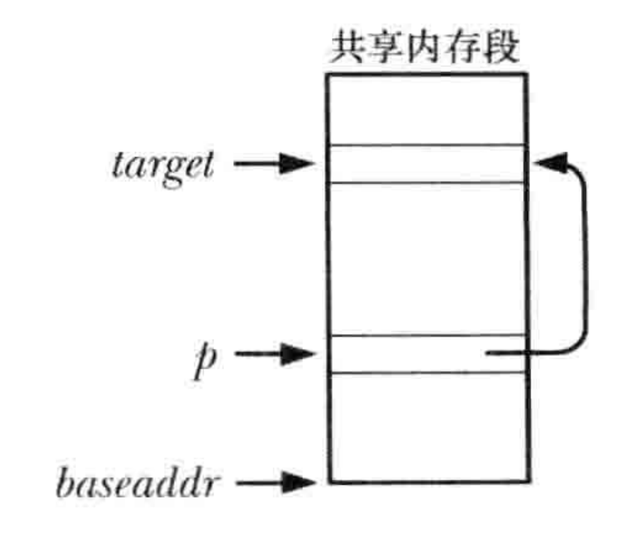


图 48-3：在共享内存段中使用指针

例如，假设一个共享内存段的起始地址为 baseaddr（即 baseaddr 的值为 shmat()的返回值）。再假设需要在 p 指向的位置处存储一个指针，该指针指向的位置与 target 指向的位置相同，如图 48-3 所示。如果在段中构建一个链表或二叉树，那么这种操作就是非常典型的一种操作。在 C 中设置*p 的传统做法如下所示。

```
*p = target;                    /* Place pointer in *p (WRONG!) */
```

上面这段代码存在的问题是当共享内存段被附加到另一个进程中时 target 指向的位置可能会位于一个不同的虚拟地址处，这意味着在那个进程中那个策划中存储在*p 中的值是是无意义的。正确的做法是在*p 中存储一个偏移量，如下所示。

```
*p = (target - baseaddr);        /* Place offset in *p */
```

在解引用这种指针时需要颠倒上面的步骤。

```
target = baseaddr + *p;          /* Interpret offset */
```

这里假设在各个进程中 baseaddr 指向共享内存段的起始位置（即各个进程中 shmat()的返回值）。给定这种假设，那么就能正确地对偏移量值进行解释，不管共享内存段被附加在进程的虚拟地址空间中的何处。

或者如果是将一组固定大小的结构链接起来的话就可以将共享内存段（或部分共享内存段）强制转换成一个数组，然后使用下标数作为“指针”来在一个结构中引用另一个结构。

48.7 共享内存控制操作

shmctl()系统调用在 shmid 标识的共享内存段上执行一组控制操作。

```
#include <sys/types.h>        /* For portability */
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Returns 0 on success, or −1 on error

cmd 参数规定了待执行的控制操作。buf 参数是 IPC_STAT 和 IPC_SET 操作（稍后介绍）会用到的，并且在执行其他操作时需要将这个参数的值指定为 NULL。

在本节余下的部分中将介绍通过 cmd 可指定的各种操作。

常规控制操作

下列操作与其他 System V IPC 对象上的操作是一样的。有关这些操作的细节信息，包括调用进程所需的特权和权限，可参考 45.3 节中的描述。

IPC_RMID

标记这个共享内存段及其关联 shmid_ds 数据结构以便删除。如果当前没有进程附加该段，那么就会执行删除操作，否则就在所有进程都已经与该段分离（即当 shmid_ds 数据结构中 shm_nattch 字段的值为 0 时）之后再执行删除操作。在一些应用程序中可以通过在所有进程将共享内存段附加到其虚拟地址空间之后立即使用 shmat()将共享内存段标记为删除来确保在应用程序退出时干净地清除共享内存段。这种做法与在打开一个文件之后立即断开到该文件的链接的做法是类似的。

在 Linux 上，如果已经使用 IPC_RMID 将一个共享段标记为删除，但因为还存在一些进程仍然附加了该段而没有删除该段，那么其他进程还能够附加该段。但这种行为是不可移植的：大多数 UNIX 实现会阻止进程将被标记为删除的段附加到自己的地址空间中。（SUSv3 并没有对这种情况的处理方式进行规定。）一些 Linux 应用程序已经依赖了这种行为，这也是 Linux 为何不改变这种行为以与其他 UNIX 实现匹配的原因。

更多资源请访问棒酷客(www.ckook.com)

IPC_STAT

将与这个共享内存段关联的 shmid_ds 数据结构的一个副本防止到 buf 指向的缓冲区中。（在 48.8 节中将介绍这个数据结构。）

IPC_SET

使用 buf 指向的缓冲区中的值来更新与这个共享内存段相关联的 shmid_ds 数据结构中被选中的字段。

加锁和解锁共享内存

一个共享内存段可以被锁进 RAM 中，这样它就永远不会被交换出去了。这种做法能够带来性能上的提升，因为一旦段中的所有分页都驻留在内存中，就能够确保一个应用程序在访问分页时永远不会因发生分页故障而被延迟。通过 shmctl()可以完成两种锁操作。

- SHM_LOCK 操作将一个共享内存段锁进内存。
- SHM_UNLOCK 操作为共享内存段解锁以允许它被交换出去。

SUSv3 并没有规定这些操作，并且所有 UNIX 实现也都没有提供这些操作。

在版本号小于 2.6.10 的 Linux 上只有特权（CAP_IPC_LOCK）进程才能够将一个共享内存段锁进内存。自 Linux 2.6.10 开始，非特权进程能够在一个共享内存段上执行加锁和解锁操作，其前提是进程的有效用户 ID 与段的所有者或创建者的用户 ID 匹配并且（在执行 SHM_LOCK 操作的情况下）进程具备足够高的 RLIMIT_MEMLOCK 资源限制，细节信息可参考 50.2 节。

锁住一个共享内存段无法确保在 shmctl()调用结束时段的所有分页都驻留在内存中。非驻留分页会在附加该共享内存段的进程引用这些分页时因分页故障而一个一个地被锁进内存。一旦分页因分页故障而被锁进了内存，那么分页就会一直驻留在内存中直到被解锁为止，即使所有进程都与该段分离之后也不会发生改变。（换句话说，SHM_LOCK 操作为共享内存段设置了一个属性，而不是为调用进程设置了一个属性。）

> 因分页故障而加载进内存表示当进程引用了一个非驻留页面时会发生一个分页故障。这时如果分页在交换区域中，那么它将会被重新加载进内存。如果分页是首次被引用，那么在交换文件中就不存在对应的分页。因此内核会在物理内存中分配一个新分页并调整进程的页表以及共享内存段的簿记数据结构。

作为给内存加锁的一种替代方法，可以使用 mlock()，它的语义与内存加锁稍微有些不同，50.2 节对此进行了介绍。

48.8 共享内存关联数据结构

每个共享内存段都有一个关联的 shmid_ds 数据结构，其形式如下。

```
struct shmid_ds {
    struct ipc_perm shm_perm;   /* Ownership and permissions */
    size_t   shm_segsz;         /* Size of segment in bytes */
    time_t   shm_atime;         /* Time of last shmat() */
    time_t   shm_dtime;         /* Time of last shmdt() */
    time_t   shm_ctime;         /* Time of last change */
    pid_t    shm_cpid;          /* PID of creator */
    pid_t    shm_lpid;          /* PID of last shmat() / shmdt() */
```


更多资源请访问稀酷客(www.ckook.com)

```
    shmatt_t shm_nattch;        /* Number of currently attached processes */
};
```

SUSv3 要求实现提供上面给出的所有字段。其他一些 UNIX 实现在 shmid_ds 结构中包含了额外的非标准字段。

各种共享内存系统调用会隐式地更新 shmid_ds 结构中的字段，使用 shmctl() IPC_SET 操作可以显式地更新 shm_perm 字段中的特定子字段。细节信息如下。

shm_perm

在创建共享内存段之后会像 45.3 节中描述的那样对这个子结构中的字段进行初始化。uid、gid 以及（低 9 位）mode 子字段是通过 IPC_SET 来更新的。除了常规的权限位之外，shm_perm.mode 字段还有两个只读位掩码标记。其中第一个是 SHM_DEST（销毁），它表示当所有进程的地址空间都与该段分离之后是否将该段标记为删除（通过 shmctl() IPC_RMID 操作）。另一个标记是 SHM_LOCKED，它表示是否将段锁进物理内存中（通过 shmctl() SHM_LOCK 操作）。这两个标记都没有在 SUSv3 中被标准化，并且只有一些 UNIX 实现提供了与这两个标记等价的标记，同时有些实现上的名称也是不同的。

shm_segsz

在创建共享内存段时这个字段会被设置成段所需要的字节数（即 shmget()调用中 size 参数的值）。在 48.2 节中提到过共享内存是以分页为单位来分配的，因此段所需的实际大小可能会大于这个值。

shm_atime

在创建共享内存段时会将这个字段设置为 0，当一个进程附加该段时（shmat()）会将这个字段设置为当前时间。这个字段以及 shmid_ds 结构中的另一个时间戳字段的类型为 time_t，它们存储着自新纪元到现在的秒数。

shm_dtime

在创建共享内存段时会将这个字段设置为 0，当一个进程与该段分离（shmdt()）之后会将这个字段设置为当前时间。

shm_ctime

当段被创建时以及每个成功的 IPC_SET 操作都会将这个字段设置为当前时间。

shm_cpid

这个字段会被设置成使用 shmget()创建这个段的进程的进程 ID。

shm_lpid

在创建共享内存段时会将这个字段设置为 0，后续每个成功的 shmat()或 shmdt()调用会将这个字段设置成调用进程的进程 ID。

shm_nattch

这个字段统计当前附加该段的进程数。在创建段时会将这个字段初始化为 0，然后每次成功的 shmat()调用会递增这个字段的值，每次成功的 shmdt()调用会递减这个字段的值。用来定义这个字段的 shmatt_t 数据类型是一个无符号整型，SUSv3 要求这个类型的大小最少为 unsigned short。（在 Linux 上这个类型被定义成了 unsigned long。）

更多资源请访问稀酷客(www.ckook.com)

48.9 共享内存的限制

大多数 UNIX 实现会对 System V 共享内存施加各种各样的限制。下面是一份 Linux 共享内存的限制列表。括号中列出了当限制达到时受影响的系统调用及其返回的错误。

SHMMNI

这是一个系统级别的限制，它限制了所能创建的共享内存标识符（换句话说是共享内存段）的数量。（shmget(), ENOSPC）

SHMMIN

这个一个共享内存段的最小大小（字节数）。这个限制的值被定义成了 1（无法修改这个值），但实际的限制是系统分页大小（shmget(), EINVAL）。

SHMMAX

这个是一个共享内存段的最大大小（字节数）。SHMMAX 的实际上限依赖于可用的 RAM 和交换空间。（shmget(), EINVAL）

SHMALL

这是一个系统级别的限制，它限制了共享内存中的分页总数。其他大多数 UNIX 实现并没有提供这个限制。SHMALL 的实际上限依赖于可用的 RAM 和交换空间。（shmget(), ENOSPC）

其他一些 UNIX 实现还施加了下列限制（Linux 并没有实现这些限制）。

SHMSEG

这个是进程级别的限制，它限制了所能附加的共享内存段数量。

在系统启动时共享内存限制会被设置成默认值。（这些默认值在不同的内核版本中可能存在差异，一些发行厂商发行的内核中的默认设置与 vanilla 内核中的默认设置是不同的。）在 Linux 上，可以通过/proc 文件系统中的文件来查看其中一些限制。表 48-2 列出了与各个限制对应的/proc 文件。下面是 Linux 2.6.31 在 x86-32 系统上的默认限制。

```
$ cd /proc/sys/kernel
$ cat shmmni
4096
$ cat shmmax
33554432
$ cat shmall
2097152
```

Linux 特有的 shmctl() IPC_INFO 操作返回一个类型为 shminfo 的结构，它包含了各个共享内存限制的值。

```
struct shminfo buf;

shmctl(0, IPC_INFO, (struct shmid_ds *) &buf);
```

相关的 Linux 特有的 SHM_INFO 操作返回一个类型为 shm_info 的结构，它包含了共享内存对象所消耗的实际资源相关的信息。本书随带的源代码的 svshm/svshm_info.c 文件中提供了一个使用 SHM_INFO 的例子。

更多资源请访问稀酷客(www.ckook.com)

有关 IPC_INFO、SHM_INFO 以及 shminfo 和 shm_info 结构的细节可以在 shmctl(2)手册中找到。

表 48-2：System V 共享内存限制

限　　制	最大值（x86-32）	/proc/sys/kernel 中对应的文件
SHMMNI	32768 (IPCMNI)	shmmni
SHMMAX	依赖于可用内存	shmmax
SHMALL	依赖于可用内存	shmall

48.10 总结

共享内存允许两个或多个进程共享内存的同一个分页。通过共享内存交换数据无需内核干涉。一旦一个进程将数据复制进一个共享内存段中之后，数据将会立即对其他进程可见。共享内存是一种快速的 IPC 机制，尽管这种速度上的提升通常会因必须要使用某种同步技术而被抵消掉一部分，如使用一个 System V 信号量来同步对共享内存的访问。

在附加一个共享内存段时推荐的做法是允许内核选择将段附加在进程的虚拟地址空间的何处。这意味着段在不同进程中虚拟地址可能是不同的。正因为这个原因，所有对段中地址的引用都应该表示成为相对偏移量，而不是一个绝对指针。

更多信息

[Bovet & Cesati, 2005]介绍了 Linux 内存管理模式和一些共享内存实现方面的细节。

48.11 习题

48-1. 使用事件标记来替换程序清单 48-2（svshm_xfr_writer.c）和程序清单 48-3（svshm_xfr_reader.c）中的二元信号量。

48-2. 解释为何程序清单 48-3 在 for 循环被修改成如下形式时会错误地报告了传输字节数。

```
for (xfrs = 0, bytes = 0; shmp->cnt != 0; xfrs++, bytes += shmp->cnt) {
    reserveSem(semid, READ_SEM);            /* Wait for our turn */

    if (write(STDOUT_FILENO, shmp->buf, shmp->cnt) != shmp->cnt)
        fatal("write");

    releaseSem(semid, WRITE_SEM);           /* Give writer a turn */
}
```

48-3. 尝试为程序清单 48-2（svshm_xfr_writer.c）和程序清单 48-3（svshm_xfr_reader.c）中的程序中用来交换数据的缓冲区指定不同大小（由常量 BUF_SIZE 定义）并编译这两个程序。记录在各种缓冲区大小下 svshm_xfr_reader.c 的执行时间。

48-4. 编写一个程序显示与共享内存段关联的 shmid_ds 数据结构（48,8 节）中的内容。段的标识符应该通过命令行参数来指定。（参见程序清单 47-3 中的程序，它在 System V 信号量上执行了一个类似的任务。）

更多资源请访问稀酷客(www.ckook.com)

48-5. 编写一个目录服务使之使用一个共享内存段来发布名称-值对。程序需要提供一个 API 来允许调用者创建新名称、修改一个既有名称、删除一个既有名称以及获取与一个名称相关联的值。使用信号量来确保一个执行共享内存段更新操作的进程能够互斥地访问段。

48-6. 编写一个程序（类似于程序清单 46-6 中的程序）使之使用 shmctl() SHM_INFO 和 SHM_STAT 操作来获取和显示系统中所有共享内存段列表。

更多资源请访问稀酷客(www.ckook.com)

第49章

内存映射

本章将介绍如何使用 mmap()系统调用来创建内存映射。内存映射可用于 IPC 以及其他很多方面。下面在深入介绍 mmap()之前首先概述一些基础概念。

49.1 概述

mmap()系统调用在调用进程的虚拟地址空间中创建一个新内存映射。映射分为两种。

- 文件映射：文件映射将一个文件的一部分直接映射到调用进程的虚拟内存中。一旦一个文件被映射之后就可以通过在相应的内存区域中操作字节来访问文件内容了。映射的分页会在需要的时候从文件中（自动）加载。这种映射也被称为基于文件的映射或内存映射文件。
- 匿名映射：一个匿名映射没有对应的文件。相反，这种映射的分页会被初始化为 0。

另一种看待匿名映射的角度（并且也接近于事实）是把它看成是一个内容总是被初始化为 0 的虚拟文件的映射。

一个进程的映射中的内存可以与其他进程中的映射共享（即各个进程的页表条目指向 RAM 中相同分页）。这种行为会在两种情况下发生。

- 当两个进程映射了一个文件的同一个区域时它们会共享物理内存的相同分页。
- 通过 fork()创建的子进程会继承其父进程的映射的副本，并且这些映射所引用的物理内存分页与父进程中相应映射所引用的分页相同。

当两个或更多个进程共享相同分页时，每个进程都有可能会看到其他进程对分页内容做出的变更，当然这要取决于映射是私有的还是共享的。

- 私有映射（MAP_PRIVATE）：在映射内容上发生的变更对其他进程不可见，对于文件映射来讲，变更将不会在底层文件上进行。尽管一个私有映射的分页在上面介绍的情况中初始时是共享的，但对映射内容所做出的变更对各个进程来讲则是私有的。内核使用了写时复制（copy-on-write）技术完成了这个任务（参见 24.2.3 节）。这意味着每

更多资源请访问稀酷客(www.ckook.com)

当一个进程试图修改一个分页的内容时，内核首先会为该进程创建一个新分页并将需修改的分页中的内容复制到新分页中（以及调整进程的页表）。正因为这个原因，MAP_PRIVATE 映射有时候会被称为私有、写时复制映射。

- 共享映射（MAP_SHARED）：在映射内容上发生的变更对所有共享同一个映射的其他进程都可见，对于文件映射来讲，变更将会发生在底层的文件上。

上面介绍的两个映射特性（文件与匿名以及私有和共享）可以以四种不同的方式加以组合，表 49-1 对此进行了总结。

表 49-1：各种内存映射的用途

变更的可见性	映射类型	
	文　件	匿　名
私有	根据文件内容初始化内存	内存分配
共享	内存映射 I/O；进程间共享内存（IPC）	进程间共享内存（IPC）

这四种不同的内存映射的创建和使用方式如下所述。

- 私有文件映射：映射的内容被初始化为一个文件区域中的内容。多个映射同一个文件的进程初始时会共享同样的内存物理分页，但系统使用写时复制技术使得一个进程对映射所做出的变更对其他进程不可见。这种映射的主要用途是使用一个文件的内容来初始化一块内存区域。一些常见的例子包括根据二进制可执行文件或共享库文件的相应部分来初始化一个进程的文本和数据段。
- 私有匿名映射：每次调用 mmap()创建一个私有匿名映射时都会产生一个新映射，该映射与同一（或不同）进程创建的其他匿名映射是不同的（即不会共享物理分页）。尽管子进程会继承其父进程的映射，但写时复制语义确保在 fork()之后父进程和子进程不会看到其他进程对映射所做出的变更。私有匿名映射的主要用途是为一个进程分配新（用零填充）内存（如在分配大块内存时 malloc()会为此而使用 mmap()）。
- 共享文件映射：所有映射一个文件的同一区域的进程会共享同样的内存物理分页，这些分页的内容将被初始化为该文件区域。对映射内容的修改将直接在文件中进行。这种映射主要用于两个用途。第一，它允许内存映射 I/O，这表示一个文件会被加载到进程的虚拟内存中的一个区域中并且对该块内容的变更会自动被写入到这个文件中。因此，内存映射 I/O 为使用 read()和 write()来执行文件 I/O 这种做法提供了一种替代方案。这种映射的第二种用途是允许无关进程共享一块内容以便以一种类似于 System V 共享内存段（第 48 章）的方式来执行（快速）IPC。
- 共享匿名映射：与私有匿名映射一样，每次调用 mmap()创建一个共享匿名映射时都会产生一个新的、与任何其他映射不共享分页的截然不同的映射。这里的差别在于映射的分页不会被写时复制。这意味着当一个子进程在 fork()之后继承映射时，父进程和子进程共享同样的 RAM 分页，并且一个进程对映射内容所做出的变更会对其他进程可见。共享匿名映射允许以一种类似于 System V 共享内存段的方式来进行 IPC，但只有相关进程之间才能这么做。

在本章余下的部分中将分别介绍各种映射的细节信息。

一个进程在执行 exec()时映射会丢失，但通过 fork()创建的子进程会继承映射，映射类型（MAP_PRIVATE 或 MAP_SHARED）也会被继承。

更多资源请访问稀酷客(www.ckook.com)

通过 Linux 特有的/proc/PID/maps 文件能够查看在 48.5 节中介绍过的与一个进程的映射有关的所有信息。

mmap()的另一个用途是与 POSIX 共享内存对象一起使用，它允许无关进程在不创建关联磁盘文件（共享文件映射需要这样的文件）的情况下共享一块内存区域。第 54 章将会介绍 POSIX 共享内存对象。

49.2 创建一个映射：mmap()

mmap()系统调用在调用进程的虚拟地址空间中创建一个新映射。

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

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Returns starting address of mapping on success, or MAP_FAILED on error

addr 参数指定了映射被放置的虚拟地址。如果将 addr 指定为 NULL，那么内核会为映射选择一个合适的地址。这是创建映射的首选做法。或者在 addr 中指定一个非 NULL 值时，内核会在选择将映射放置在何处时将这个参数值作为一个提示信息来处理。在实践中，内核至少会将指定的地址舍入到最近的一个分页边界处。不管采用何种方式，内核会选择一个不与任何既有映射冲突的地址。（如果在 flags 包含了 MAP_FIXED，那么 addr 必须是分页对齐的。在 49.10 节中将会对这个标记进行介绍。）

成功时 mmap()会返回新映射的起始地址。发生错误时 mmap()会返回 MAP_FAILED。

在 Linux（以及大多数其他 UNIX 实现）上，MAP_FAILED 常量等同于((void *) –1)。但 SUSv3 规定了这个常量值，因为 C 标准无法确保能够将((void *) –1)与成功的 mmap()调用的返回值区分开来。

length 参数指定了映射的字节数。尽管 length 无需是一个系统分页大小（sysconf(_SC_PAGESIZE)返回值）的倍数，但内核会以分页大小为单位来创建映射，因此实际上 length 会被向上提升为分页大小的下一个倍数。

prot 参数是一个位掩码，它指定了施加于映射之上的保护信息，其取值要么是 PROT_NONE，要么是表 49-2 中列出的其他三个标记的组合（取 OR）。

表 49-2：内存保护值

值	描　述
PROT_NONE	区域无法访问
PROT_READ	区域内容可读取
PROT_WRITE	区域内容可修改
PROT_EXEC	区域内容可执行

flags 参数是一个控制映射操作各个方面的选项的位掩码。这个掩码必须只包含下列值中一个。

更多资源请访问稀酷客(www.ckook.com)

MAP_PRIVATE

创建一个私有映射。区域中内容上所发生的变更对使用同一映射的其他进程是不可见的，对于文件映射来讲，所发生的变更将不会反应在底层文件上。

MAP_SHARED

创建一个共享映射。区域中内容上所发生的变更对使用 MAP_SHARED 特性映射同一区域的进程是可见的，对于文件映射来讲，所发生的变更将直接反应在底层文件上。对文件的更新将无法确保立即生效，具体可参加 49.5 节中对 msync()系统调用的介绍。

除了 MAP_PRIVATE 和 MAP_SHARED 之外，在 flags 中还可以有选择地对其他标记取 OR。在 49.6 和 49.10 节中将会对这些标记进行介绍。

剩余的参数 fd 和 offset 是用于文件映射的（匿名映射将忽略它们）。fd 参数是一个标识被映射的文件的文件描述符。offset 参数指定了映射在文件中的起点，它必须是系统分页大小的倍数。要映射整个文件就需要将 offset 指定为 0 并且将 length 指定为文件大小。在 49.5 节中将会介绍更多有关文件映射的内容。

有关内存保护的更多细节

前面提过 mmap() prot 参数指定了新内存映射上的保护信息。这个参数可以取 PROT_NONE 或者 PROT_READ、PROT_WRITE、以及 PROT_EXEC 中一个或多个标记的掩码。如果一个进程在访问一个内存区域时违反了该区域上的保护位，那么内核会向该进程发送一个 SIGSEGV 信号。

尽管 SUSv3 规定 SIGSEGV 应该被用来通知内存保护违背，但在一些实现上使用的则是 SIGBUS。

标记为 PROT_NONE 的分页内存的一个用途是作为一个进程分配的内存区域的起始位置或结束位置的守护分页。如果进程意外地访问了其中一个被标记为 PROT_NONE 的分页，那么内核会通过生成一个 SIGSEGV 信号来通知该进程这样一个事实。

内存保护信息驻留在进程私有的虚拟内存表中。因此，不同的进程可能会使用不同的保护位来映射同一个内存区域。

使用 mprotect()系统调用（50.1 节）能够修改内存保护位。

在一些 UNIX 实现上，实际施加于一个映射分页上的保护位于在 prot 中指定的信息可能不完全一致。特别地，底层硬件在保护粒度上的限制（如老式的 x86-32 架构）意味着在很多 UNIX 实现上 PROT_READ 会隐含 PROT_EXEC，反之亦然，并且在一些实现上指定 PROT_WRITE 会隐含 PROT_READ。但应用程序不应该依赖于这种行为；prot 指定的信息应该总是与所需的内存保护信息一致。

现代 x86-32 架构为将页表标记为 NX（no execute）提供了硬件支持，并且自内核 2.6.8 起，Linux 利用这个特性来合适地分隔 Linux/x86-32 上的 PROT_READ 和 PROT_EXEC 权限。

标准中规定的对 offset 和 addr 的对齐约束

SUSv3 规定 mmap()的 offset 参数必须要与分页对齐，而 addr 参数在指定了 MAP_FIXED 的情

更多资源请访问稀酷客(www.ckook.com)

况下也必须要与分页对齐。Linux 遵循了这些要求，但后面又发现 SUSv3 的要求与之前的标准提出的要求是不同的，之前的标准对这些参数的要求要低一些。SUSv3 中的措辞会（不必要地）导致一些之前符合标准的实现变得不符合标准了。SUSv4 则放宽了这方面的要求：

- 一个实现可能会要求 offset 为系统分页大小的倍数。
- 如果指定了 MAP_FIXED，那么一个实现可能会要求 addr 是分页对齐的。
- 如果指定了 MAP_FIXED 并且 addr 为非零值，那么 addr 和 offset 除以系统分页大小所得的余数应该相等。

mprotect()、msync()以及 munmap()中的 addr 参数也存在类似的情况。SUSv3 规定这个参数必须是分页对齐的。SUSv4 表示一个实现可以要求这个参数是分页对齐的。

示例程序

程序清单 49-1 演示了如何使用 mmap()来创建一个私有文件映射。这个程序是一个简单版本的 cat(1)，它将映射通过命令行参数指定的（整个）文件，然后将映射中的内容写入到标准输出中。

程序清单 49-1：使用 mmap()创建一个私有文件映射

mmap/mmcat.c

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

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

    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");
    /* Obtain the size of the file and use it to specify the size of
       the mapping and the size of the buffer to be written */

    if (fstat(fd, &sb) == -1)
        errExit("fstat");

    addr = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (write(STDOUT_FILENO, addr, sb.st_size) != sb.st_size)
        fatal("partial/failed write");
    exit(EXIT_SUCCESS);
}
```

mmap/mmcat.c

49.3 解除映射区域：munmap()

munmap()系统调用执行与 mmap()相反的操作，即从调用进程的虚拟地址空间中删除一个映射。

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

int munmap(void *addr, size_t length);
```

Returns 0 on success, or −1 on error

addr 参数是待解除映射的地址范围的起始地址，它必须与一个分页边界对齐。（SUSv3 规定 addr 必须是分页对齐的。SUSv4 表示一个实现可以要求这个参数是分页对齐的。）

length 参数是一个非负整数，它指定了待解除映射区域的大小（字节数）。范围为系统分页大小的下一个倍数的地址空间将会被解除映射。

一般来讲通常会解除整个映射。因此可以将 addr 指定为上一个 mmap()调用返回的地址，并且 length 的值与 mmap()调用中使用的 length 的值一样。下面是一个例子。

```
addr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
if (addr == MAP_FAILED)
    errExit("mmap");

/* Code for working with mapped region */

if (munmap(addr, length) == -1)
    errExit("munmap");
```

或者也可以解除一个映射中的部分映射，这样原来的映射要么会收缩，要么会被分成两个，这取决于在何处开始解除映射。还可以指定一个跨越多个映射的地址范围，这样的话所有在范围内的映射都会被解除。

如果在由 addr 和 length 指定的地址范围中不存在映射，那么 munmap()将不起任何作用并返回 0（表示成功）。

在解除映射期间，内核会删除进程持有的在指定地址范围内的所有内存锁。（内存锁是通过 mlock()或 mlockall()来建立的，50.2 节将会对此予以介绍。）

当一个进程终止或执行了一个 exec()之后进程中所有的映射会自动被解除。

为确保一个共享文件映射的内容会被写入到底层文件中，在使用 munmap()解除一个映射之前需要调用 msync()（参见 49.5 节）。

49.4 文件映射

要创建一个文件映射需要执行下面的步骤。

1. 获取文件的一个描述符，通常通过调用 open()来完成。
2. 将文件描述符作为 fd 参数传入 mmap()调用。

执行上述步骤之后 mmap()会将打开的文件的内容映射到调用进程的地址空间中。一旦 mmap()被调用之后就能够关闭文件描述符了，而不会对映射产生任何影响。但在一些情况

更多资源请访问稀酷客(www.ckook.com)

下，将这个文件描述符保持在打开状态可能是有用的——如参见程序清单 49-1 以及参见第 54 章。

除了普通的磁盘文件，使用 mmap()还能够映射各种真实和虚拟设备的内容，如硬盘、光盘以及/dev/mem。

在打开描述符 fd 引用的文件时必须要具备与 prot 和 flags 参数值匹配的权限。特别地，文件必须总是被打开以允许读取，并且如果在 flags 中指定了 PROT_WRITE 和 MAP_SHARED，那么文件必须总是被打开以允许读取和写入。

offset 参数指定了从文件区域中的哪个字节开始映射，它必须是系统分页大小的倍数。将 offset 指定为 0 会导致从文件的起始位置开始映射。length 参数指定了映射的字节数。offset 和 length 参数一起确定了文件的哪个区域会被映射进内存，如图 49-1 所示。

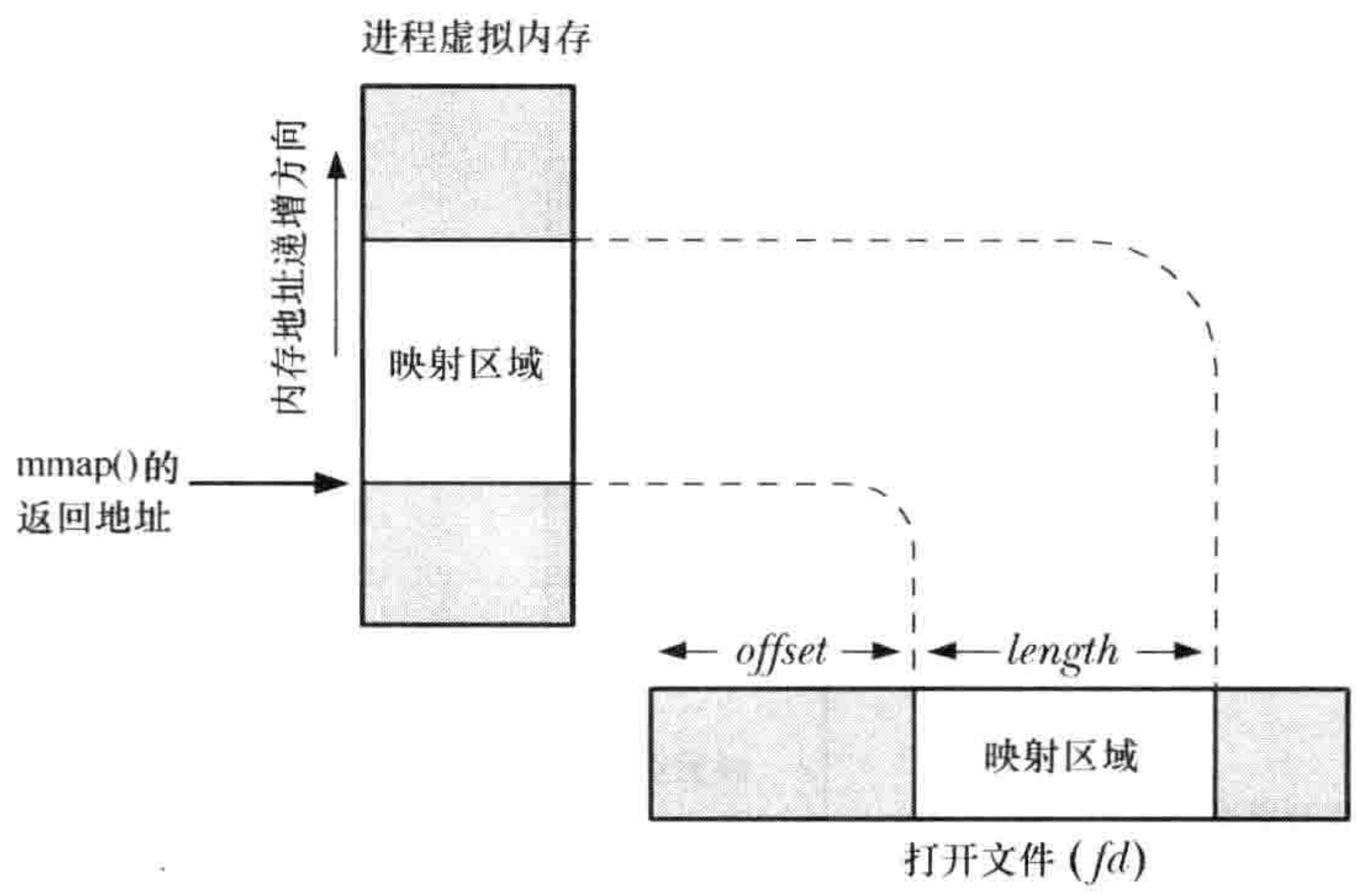


图 49-1：内存映射文件概览

在 Linux 上，一个文件映射的分页会在首次被访问时被映射进内存。这意味着如果在 mmap()调用之后修改了文件区域，但映射的对应部分（即分页）还没有被访问过，那么如果相应分页还没有被加载进内存的话，变更对这个进程可能是可见的。这个行为是依赖于实现的，可移植的应用程序应该避免依赖某个特定内核在这种场景中的行为。

49.4.1 私有文件映射

私有文件映射最常见的两个用途如下所述。

- 允许多个执行同一个程序或使用同一个共享库的进程共享同样的（只读的）文本段，它是从底层可执行文件或库文件的相应部分映射而来的。

尽管可执行文件的文本段通常是被保护成只允许读取和执行访问（PROT_READ | PROT_EXEC），但在被映射时仍然使用了 MAP_PRIVATE 而不是 MAP_SHARED，这是因为调试器或自修改的程序能够修改程序文本（在修改了内存上的保护信息之后），而这样的变更是不应该发生在底层文件上或影响到其他进程的。

更多资源请访问稀酷客(www.ckook.com)

- 映射一个可执行文件或共享库的初始化数据段。这种映射会被处理成私有使得对映射数据段内容的变更不会发生在底层文件上。

mmap()的这两种用法通常对程序是不可见的，因为这些映射是由程序加载器和动态链接器创建的。读者可以在 48.5 节中给出的/proc/PID/maps 的输出中发现这两种映射。

私有文件映射的另一个不太常见的用途是简化程序的文件输入逻辑。这与使用共享文件映射来完成内存映射 I/O（下一节将予以介绍）类似，但它只允许文件输入。

49.4.2 共享文件映射

当多个进程创建了同一个文件区域的共享映射时，它们会共享同样的内存物理分页。此外，对映射内容的变更将会反应到文件上。实际上，这个文件被当成了该块内存区域的分页存储，如图 49-2 所示。（这幅图是简化过的，它并没有指出映射分页在物理内存中通常是不连续的这样一个事实。）

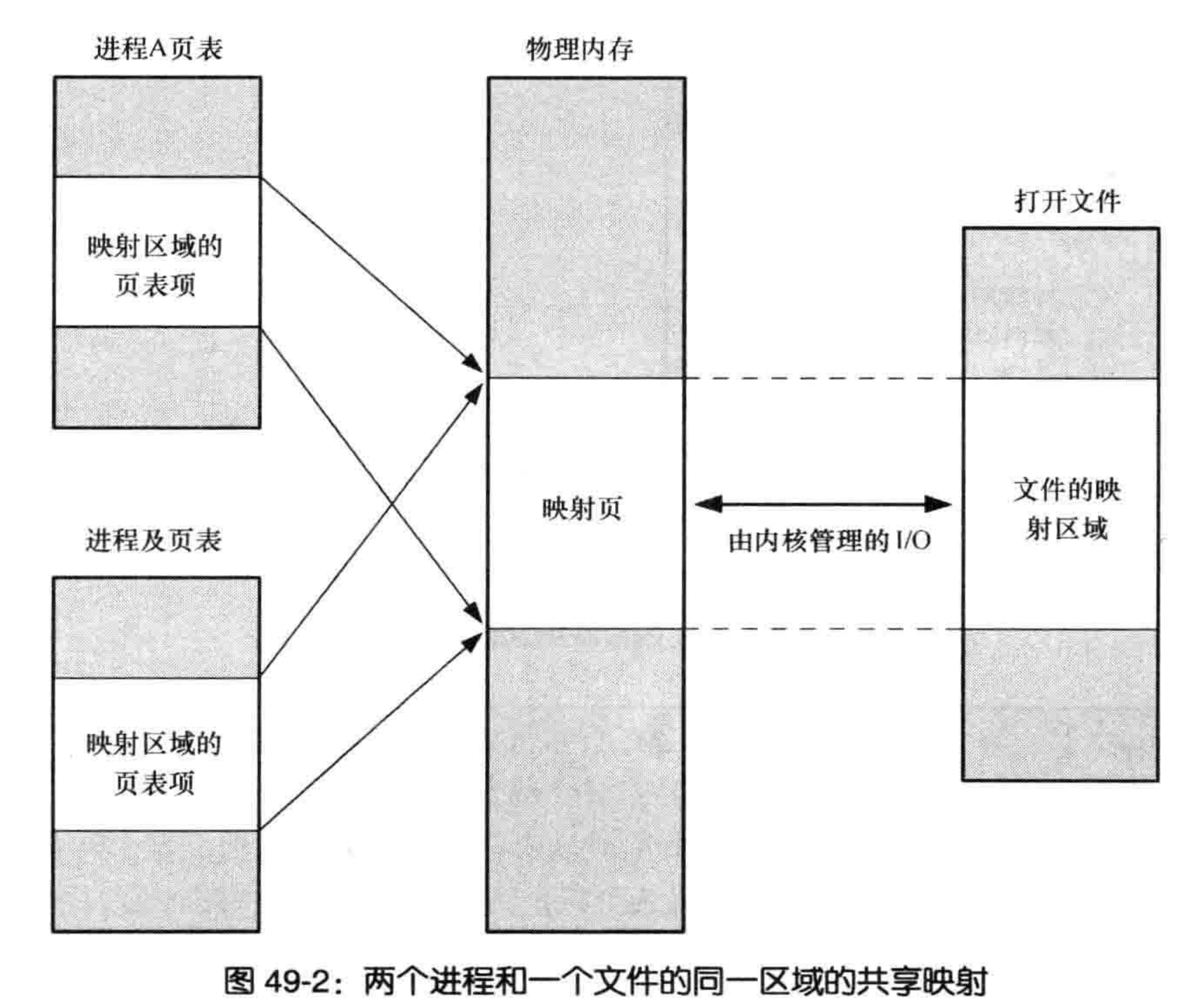


图 49-2：两个进程和一个文件的同一区域的共享映射

共享文件映射存在两个用途：内存映射 I/O 和 IPC。下面将分别介绍这两种用途。

内存映射 I/O

由于共享文件映射中的内容是从文件初始化而来的，并且对映射内容所做出的变更都会自动反应到文件上，因此可以简单地通过访问内存中的字节来执行文件 I/O，而依靠内核来确保对内存的变更会被传递到映射文件中。（一般来讲，一个程序会定义一个结构化数据类型来与磁盘文件中的内容对应起来，然后使用该数据类型来转换映射的内容。）这项技术被称为内存映射 I/O，它是使用 read()和 write()来访问文件内容这种方法的替代方案。

内存映射 I/O 具备两个潜在的优势。

更多资源请访问稀酷客(www.ckook.com)

- 使用内存访问来取代 read()和 write()系统调用能够简化一些应用程序的逻辑。
- 在一些情况下，它能够比使用传统的 I/O 系统调用执行文件 I/O 这种做法提供更好的性能。

内存映射 I/O 之所以能够带来性能优势的原因如下。

- 正常的 read()或 write()需要两次传输：一次是在文件和内核高速缓冲区之间，另一次是在高速缓冲区和用户空间缓冲区之间。使用 mmap()就无需第二次传输了。对于输入来讲，一旦内核将相应的文件块映射进内存之后用户进程就能够使用这些数据了。对于输出来讲，用户进程仅仅需要修改内存中的内容，然后可以依靠内核内存管理器来自动更新底层的文件。
- 除了节省了内核空间和用户空间之间的一次传输之外，mmap()还能够通过减少所需使用的内存来提升性能。当使用 read()或 write()时，数据将被保存在两个缓冲区中：一个位于用户空间，另一个位于内核空间。当使用 mmap()时，内核空间和用户空间会共享同一个缓冲区。此外，如果多个进程正在在同一个文件上执行 I/O，那么它们通过使用 mmap()就能够共享同一个内核缓冲区，从而又能够节省内存的消耗。

内存映射 I/O 所带来的性能优势在在大型文件中执行重复随机访问时最有可能体现出来。如果顺序地访问一个文件，并假设执行 I/O 时使用的缓冲区大小足够大以至于能够避免执行大量的 I/O 系统调用，那么与 read()和 write()相比，mmap()带来的性能上的提升就非常有限或者说根本就没有带来性能上的提升。性能提升的幅度之所以非常有限的原因是不管使用何种技术，整个文件的内容在磁盘和内存之间只传输一次，效率的提高主要得益于减少了用户空间和内核空间之间的一次数据传输，并且与磁盘 I/O 所需的时间相比，内存使用量的降低通常是可以忽略的。

> 内存映射 I/O 也有一些缺点。对于小数据量 I/O 来讲，内存映射 I/O 的开销（即映射、分页故障、解除映射以及更新硬件内存管理单元的超前转换缓冲器）实际上要比简单的 read()或 write()大。此外，有些时候内核难以高效地处理可写入映射的回写（在这种情况下，使用 msync()或 sync_file_range()有助于提高效率）。

使用共享文件映射的 IPC

由于所有使用同样文件区域的共享映射的进程共享同样的内存物理分页，因此共享文件映射的第二个用途是作为一种（快速的）IPC 方法。这种共享内存区域与 System V 共享内存对象（第 48 章）之间的区别在于区域中内容上的变更会反应到底层的映射文件上。这种特性对那些需要共享内存内容在应用程序或系统重启时能够持久化的应用程序来讲是非常有用的。

示例程序

程序清单 49-2 提供了一个简单的例子来演示如何使用 mmap()创建一个共享文件映射。这个程序首先映射一个名称通过第一个命令行参数指定的文件，然后打印出映射区域起始位置的字符串值。最后，如果提供了第二个命令行参数，那么该字符串会被复制进共享内存区域中。

下面的 shell 会话日志演示了如何使用这个程序。下面首先创建了一个大小为 1024 字节的文件并在其中填满零。

```
$ dd if=/dev/zero of=s.txt bs=1 count=1024
1024+0 records in
1024+0 records out
```


更多资源请访问稀酷客(www.ckook.com)

然后使用程序映射这个文件并将一个字符串复制进映射区域中。

```
$ ./t_mmap s.txt hello
Current string=
Copied "hello" to shared memory
```

程序在打印当前字符串时不会显示任何内容，因为映射文件的初始值是以 null 字节打头的（即长度为零的字符串）。

接着再次使用程序映射这个文件并复制一个新字符串到映射区域中。

```
$ ./t_mmap s.txt goodbye
Current string=hello
Copied "goodbye" to shared memory
```

最后通过输出文件的内容来对其中的内容进行验证，每行显示了 8 个字符。

```
$ od -c -w8 s.txt
0000000   g   o   o   d   b   y   e nul
0000010 nul nul nul nul nul nul nul nul
*
0002000
```

这个简单的程序没有使用任何机制来同步多个进程对映射文件的访问。但现实世界中的应用程序通常需要同步对共享映射的访问。这可以通过使用各种技术来完成，包括信号量（第 47 和 53 章）和文件加锁（第 55 章）。

在 49.5 节中将会对程序清单 49-2 中用到的 msync()系统调用进行解释。

程序清单 49-2：使用 mmap()创建一个共享文件映射

mmap/t_mmap.c

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

#define MEM_SIZE 10

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

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

    fd = open(argv[1], O_RDWR);
    if (fd == -1)
        errExit("open");
    addr = mmap(NULL, MEM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1)                /* No longer need 'fd' */
        errExit("close");

    printf("Current string=%.*s\n", MEM_SIZE, addr);
                /* Secure practice: output at most MEM_SIZE bytes */

    if (argc > 2) {                     /* Update contents of region */
```

```
        if (strlen(argv[2]) >= MEM_SIZE)
            cmdLineErr("'new-value' too large\n");

        memset(addr, 0, MEM_SIZE);      /* Zero out region */
        strncpy(addr, argv[2], MEM_SIZE - 1);
        if (msync(addr, MEM_SIZE, MS_SYNC) == -1)
            errExit("msync");

        printf("Copied \"%s\" to shared memory\n", argv[2]);
    }

    exit(EXIT_SUCCESS);
}
```

mmap/t_mmap.c

49.4.3 边界情况

在很多情况下，一个映射的大小是系统分页大小的整数倍，并且映射会完全落入映射文件的范围之内。但这种要求不是必需的，下面来看一下当这些条件不满足时会发生什么事情。

图 49-3 描绘了映射完全落入映射文件的范围之内但区域的大小并不是系统分页大小的一个整数倍的情况（在这个讨论中假设分页大小为 4096 字节）。

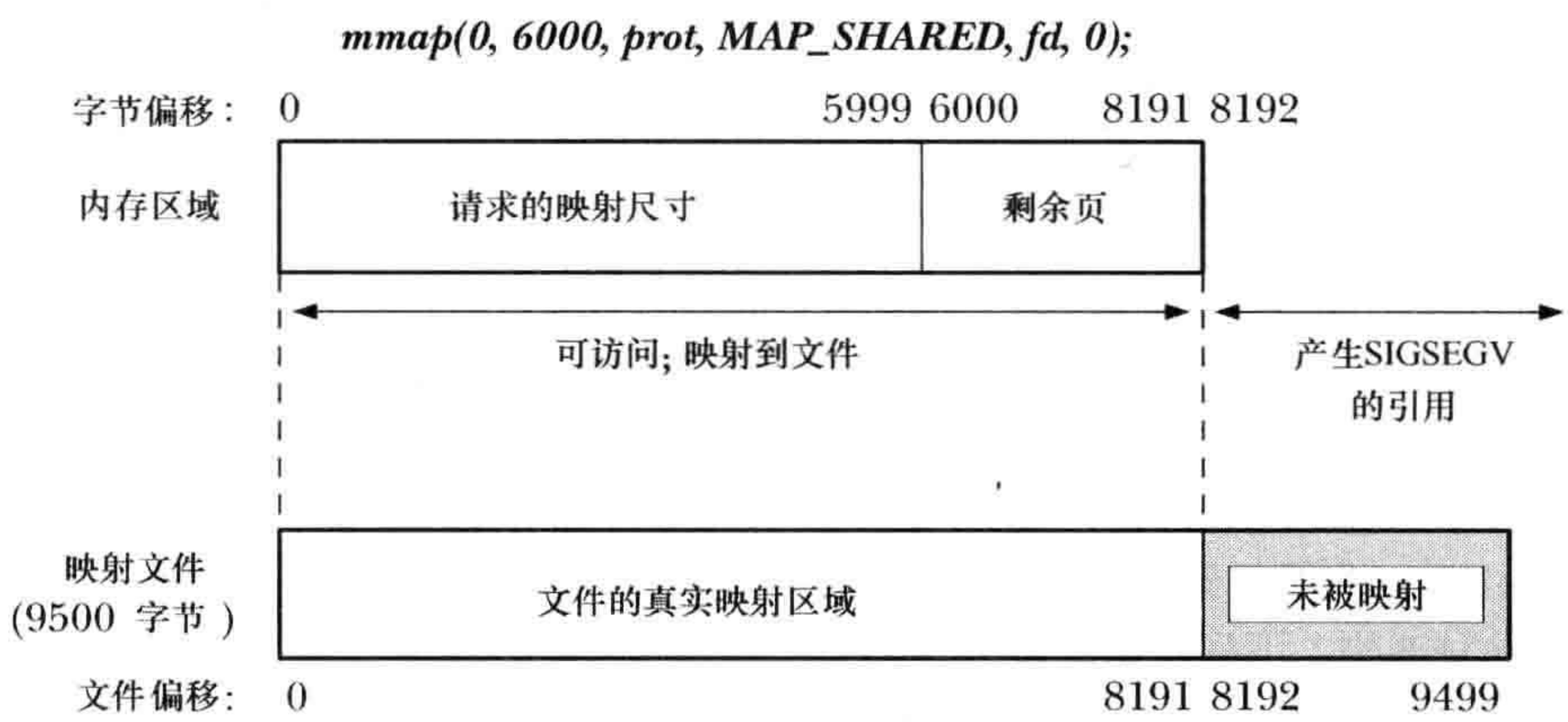


图 49-3：length 不是系统分页大小的整数倍的内存映射

由于映射的大小不是系统分页大小的整数倍，因此它会被向上舍入到系统分页大小的下一个整数倍。由于文件的大小要大于这个被向上舍入的大小，因此文件中对应字节会像图 49-3 中那样被映射。

试图访问映射结尾之外的字节将会导致 SIGSEGV 信号的产生（假设在该位置处不存在其他映射）。这个信号的默认动作是终止进程并打印出一个 core dump。

当映射扩充过了底层文件的结尾处时（参见图 49-4）情况就变得更加复杂了。与之前一样，由于映射的大小不是系统分页大小的整数倍，因此它会被向上舍入。但在这种情况下，虽然在向上舍入区域（即图中 2200 字节和 4095 字节）中的字节是可访问的，但它们不会被映射到底层文件上（由于在文件中不存在对应的字节），并且它们会被初始化为 0（SUSv3 对此进行了规定）。当然，这些字节也不会与映射同一个文件的其他进程共享，即使它们指定了足够大的 length 参数。对这些字节做出的变更不会被写入到文件中。

更多资源请访问稀酷客(www.ckook.com)

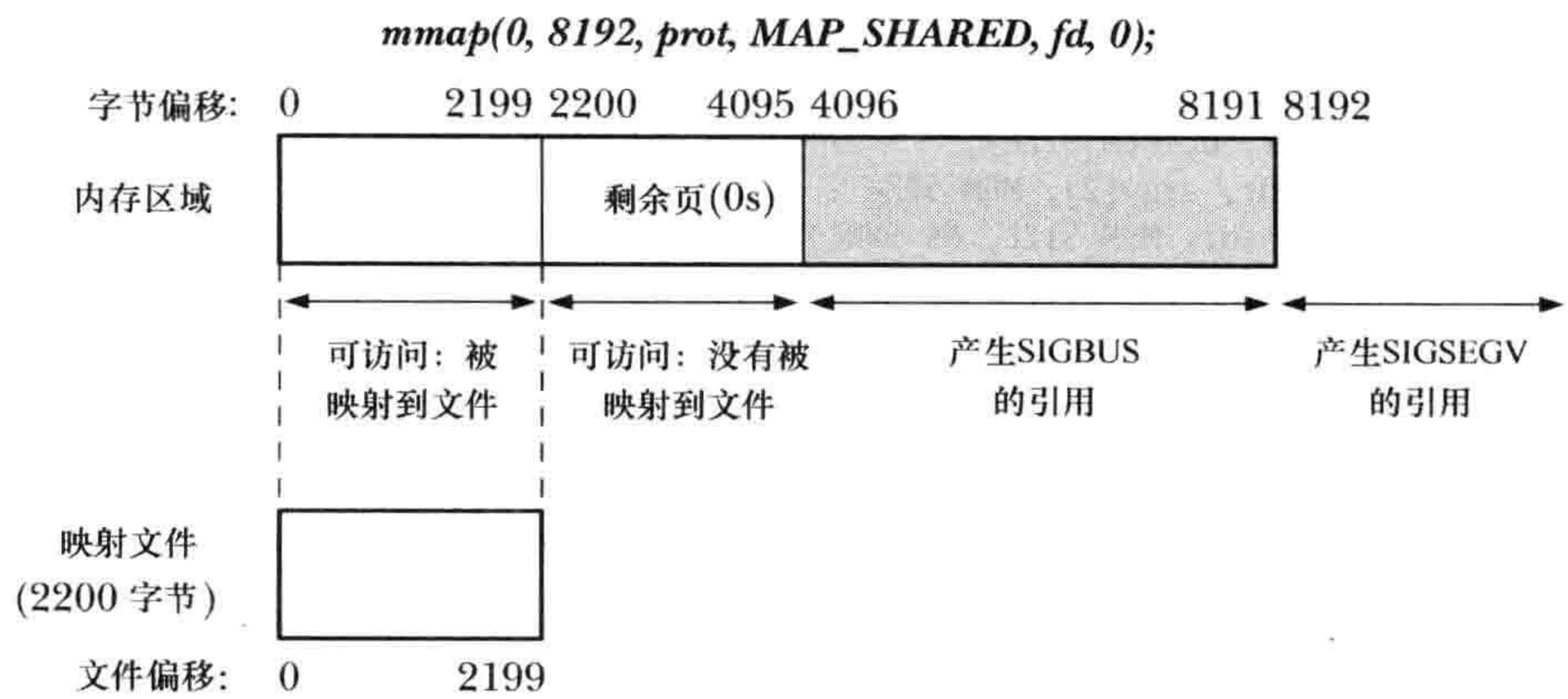


图 49-4：内存映射扩充过了映射文件的结尾

如果映射中包含了超出向上舍入区域中（即图 49-4 中 4096 以及之后的字节）的分页，那么试图访问这些分页中的地址将会导致 SIGBUS 信号量的产生，即警告进程文件中没有区域与这些地址对应。与之前一样，试图访问超过映射结尾处的地址将会导致 SIGSEGV 信号的产生。

从上面的描述中可以看出，创建一个大小超过底层文件大小的映射可能是无意义的。但通过扩展文件的大小（如使用 ftruncate()或 write()），可以使得这种映射中之前不可访问的部分变得可用。

49.4.4 内存保护和文件访问模式交互

到目前为止还没有详细解释的一点是通过 mmap() prot 参数指定的内存保护与映射文件被打开的模式之间的交互。从一般原则来讲，PROT_READ 和 PROT_EXEC 保护要求被映射的文件使用 O_RDONLY 或 O_RDWR 打开，而 PROT_WRITE 保护要求被映射的文件使用 O_WRONLY 或 O_RDWR 打开。

然而，由于一些硬件架构提供的内存保护粒度有限，因此情况会变得复杂起来（参见 49.2 节）。对于这种架构，下列结论是适用的。

- 所有内存保护组合与使用 O_RDWR 标记打开文件是兼容的。
- 没有内存保护组合——哪怕仅仅是 PROT_WRITE——与使用 O_WRONLY 标记打开的文件是兼容的（导致 EACCES 错误的发生）。这与一些硬件架构不允许对一个分页的只写访问这样一个事实是一致的。在 49.2 节中指出过在那些架构上 PROT_WRITE 隐含了 PROT_READ，这意味着如果分页可写入，那么它也能被读取。而读取操作与 O_WRONLY 是不兼容的，该操作是不能暴露文件的初始内容的。
- 使用 O_RDONLY 标记打开一个文件的结果依赖于在调用 mmap()时是否指定了 MAP_PRIVATE 或 MAP_SHARED。对于一个 MAP_PRIVATE 映射来讲，在 mmap()中可以指定任意的内存保护组合——因为对MAP_PRIVATE分页内容做出的变更不会被写入到文件中，因此无法写入文件不会成为问题。对于一个 MAP_SHARED 映射来讲，唯一与 O_RDONLY 兼容的内存保护是 PROT_REA 和(PROT_READ | PROT_EXEC)。这是符合逻辑的，因为一个 PROT_WRITE, MAP_SHARED 映射允许更新被映射的文件。

更多资源请访问稀酷客(www.ckook.com)

49.5 同步映射区域：msync()

内核会自动将发生在 MAP_SHARED 映射内容上的变更写入到底层文件中，但在默认情况下，内核不保证这种同步操作会在何时发生。（SUSv3 要求一个实现提供这种保证。）

msync()系统调用让应用程序能够显式地控制何时完成共享映射与映射文件之间的同步。同步一个映射与底层文件在多种情况下都是非常有用的。如，为确保数据完整性，一个数据库应用程序可能会调用 msync()强制将数据写入到磁盘上。调用 msync()还允许一个应用程序确保在可写入映射上发生的更新会对在该文件上执行 read()的其他进程可见。

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

int msync(void *addr, size_t length, int flags);
```

Returns 0 on success, or −1 on error

传给 msync()的 addr 和 length 参数指定了需同步的内存区域的起始地址和大小。在 addr 中指定的地址必须是分页对齐的，length 会被向上舍入到系统分页大小的下一个整数倍。（SUSv3 规定 addr 必须要分页对齐。SUSv4 表示一个实现可以要求这个参数是分页对齐的。）

flags 参数的可取值为下列值中的一个。

MS_SYNC

执行一个同步的文件写入。这个调用会阻塞直到内存区域中所有被修改过的分页被写入到底盘为止。

MS_ASYNC

执行一个异步的文件写入。内存区域中被修改过的分页会在后面某个时刻被写入磁盘并立即对在相应文件区域中执行 read()的其他进程可见。

另一种区分这两个值的方式可以表述为在 MS_SYNC 操作之后，内存区域会与磁盘同步，而在 MS_ASYNC 操作之后，内存区域仅仅是与内核高速缓冲区同步。

如果在 MS_ASYNC 操作之后不采取进一步的动作，那么内存区域中被修改过的分页最终会作为由 pdflush 内核线程（在 Linux 2.4 以及之前的版本上是 kupdated）执行的自动缓冲区刷新的一部分被写入到磁盘。在 Linux 上存在两种更快的发动输出的（非标准）方法。在 msync()调用之后可以在映射对应的文件描述符上调用一个 fsync()（或 fdatasync()）。这个调用会阻塞直到快速缓冲区与磁盘同步为止。或者可以使用 posix_fadvise() POSIX_FADV_DONTNEED 操作启动一个异步的分页写入。（Linux 特有的这两个操作并没有在 SUSv3 中予以规定。）

在 flags 参数中还可以加上下面这个值。

MS_INVALIDATE

使映射数据的缓存副本失效。当内存区域中所有被修改过的分页被同步到文件中之后，内存区域中所有与底层文件不一致的分页会被标记为无效。当下次引用这些分页时会从文件的相应位置处复制相应的分页内容，其结果是其他进程对文件做出的所有更新将会在内存区

更多资源请访问稀酷客(www.ckook.com)

域中可见。

与很多其他现代 UNIX 实现一样，Linux 提供了一个所谓的同一虚拟内存系统。这表示内存映射和高速缓冲区块会尽可能地共享同样的物理内存分页。因此通过映射获取的文件视图与通过 I/O 系统调用（read()、write()等）获得的文件视图总是一致的，而 msync()的唯一用途就是强制将一个映射区域中的内容写入到磁盘。

不管怎样，SUSv3 并没有要求实现统一虚拟内存系统，并且并不是所有的 UNIX 实现都提供了同一虚拟内存系统。在这类系统上需要调用 msync()来使得一个映射上发生的变更对其他 read()该文件的进程可见，并且在执行逆操作时需要使用 MS_INVALIDATE 标记来使得其他进程对文件所做出的写入对映射区域可见。使用 mmap()和 I/O 系统调用操作同一个文件的多进程应用程序如果希望可被移植到不具备统一虚拟内存系统的系统之上的话就需要恰当使用 msync()。

49.6 其他 mmap()标记

除了 MAP_PRIVATE 和 MAP_SHARED 之外，Linux 允许在 mmap() flags 参数中包含其他一些值（取 OR）。表 49-3 对这些值进行了总结。除了 MAP_PRIVATE 和 MAP_SHARED 之外，在 SUSv3 中仅规定了 MAP_FIXED 标记。

表 49-3：mmap() flags 参数的位掩码值

值	描　述	SUSv3
MAP_ANONYMOUS	创建一个匿名映射	
MAP_FIXED	原样解释 addr 参数（49.10 节）	●
MAP_LOCKED	将映射分页锁进内存（自 Linux 2.6 起）	
MAP_HUGETLB	创建一个使用巨页的映射（自 Linux 2.6.32 起）	
MAP_NORESERVE	控制交换空间的预留（49.9 节）	
MAP_NORESERVE	对映射数据的修改是私有的	●
MAP_POPULATE	填充一个映射的分页（自 Linux 2.6 起）	
MAP_SHARED	发生在映射数据上的变更对其他进程可见并会被反映到底层文件上（与 MAP_PRIVATE 相反）	●
MAP_UNINITIALIZED	不清除匿名映射（自 Linux 2.6.33 起）	

下面提供了与表 49-3 中列出的 flags 值有关的更多细节信息（不包含 MAP_ PRIVATE 和 MAP_SHARED，因为之前已经介绍过这两个标记了）。

MAP_ANONYMOUS

创建一个匿名映射，即没有底层文件对应的映射。在 49.7 节中将会对这个标记进行深入介绍。

MAP_FIXED

在 49.10 节中将会对这个标记进行介绍。

MAP_HUGETLB（自 Linux 2.6.32 起）

这个标记在 mmap()所起的作用与 SHM_HUGETLB 标记在 System V 共享内存段中所起的作用一样。参见 48.2 节。

更多资源请访问稀酷客(www.ckook.com)

MAP_LOCKED（自 Linux 2.6 起）

按照 mlock()的方式预加载映射分页并将映射分页锁进内存。在 50.2 节中将会对使用这个标记所需的特权以及管理其操作的限制进行介绍。

MAP_NORESERVE

这个标记用来控制是否提前为映射的交换空间执行预留操作。细节信息请参见 49.9 节。

MAP_POPULATE（自 Linux 2.6 起）

填充一个映射的分页。对于文件映射来讲，这将会在文件上执行一个超前读取。这意味着后续对映射内容的访问不会因分页故障而发生阻塞（假设此时不会因内存压力而导致分页被交换出去）。

MAP_UNINITIALIZED（自 Linux 2.6.33 起）

指定这个标记会防止一个匿名映射被清零。它能够带来性能上的提升，但同时也带来了安全风险，因为已分配的分页中可能会包含上一个进程留下来的敏感信息。因此这个标记一般只供嵌入式系统使用，因为在这种系统中性能是一个至关重要的因素，并且整个系统都处于嵌入式应用程序的控制之下。这个标记只有在使用 CONFIG_MMAP_ALLOW_ UNINITIALIZED 选项配置内核时才会生效。

49.7 匿名映射

匿名映射是没有对应文件的一种映射。本节将介绍如何创建匿名映射以及私有和共享匿名映射的用途。

MAP_ANONYMOUS 和/dev/zero

在 Linux 上，使用 mmap()创建匿名映射存在两种不同但等价的方法。

- 在 flags 中指定 MAP_ANONYMOUS 并将 fd 指定为−1。（在 Linux 上，当指定了 MAP_ANONYMOUS 之后会忽略 fd 的值。但一些 UNIX 实现要求在使用 MAP_ANONYMOUS 时将 fd 指定为−1，因此可移植的应用程序应该确保它们这样做了。）

> 要从<sys/mman.h>中获得 MAP_ANONYMOUS 的定义必须要定义_BSD_SOURCE 特性测试宏或_SVID_SOURCE 特性测试宏。Linux 提供了常量 MAP_ANON 作为 MAP_ANONYMOUS 的一个同义词，其目的是为了与其他一些采用这种命名方式的 UNIX 实现保持兼容。

- 打开/dev/zero 设备文件并将得到的文件描述符传递给 mmap()。

> /dev/zero 是一个虚拟设备，当从中读取数据时它总是会返回 0，而写入到这个设备中的数据总会被丢弃。/dev/zero 的一个常见用途是使用 0 来组装一个文件（如使用 dd(1)命令）。

不管是使用 MAP_ANONYMOUS 还是使用/dev/zero 技术，得到的映射中的字节会被初始化为 0。在两种技术中，offset 参数都会被忽略（因为没有底层文件，所以也无从指定偏移量）。稍后将会介绍使用这两种技术的例子。

更多资源请访问稀酷客(www.ckook.com)

MAP_ANONYMOUS 和/dev/zero 技术并没有在 SUSv3 进行规定，尽管大多数 UNIX 实现都支持其中一种或两种。之所以存在两种不同的技术实现同样的语义的原因是其中一种（MAP_ANONYMOUS）源自 BSD，而另一种（/dev/zero）则源自 System V。

MAP_PRIVATE 匿名映射

MAP_PRIVATE 匿名映射用来分配进程私有的内存块并将其中的内容初始化为 0。下面的代码使用/dev/zero 技术创建了一个 MAP_PRIVATE 匿名映射。

```
fd = open("/dev/zero", O_RDWR);
if (fd == -1)
    errExit("open");
addr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
if (addr == MAP_FAILED)
    errExit("mmap");
```

glibc 中的 malloc()实现使用 MAP_PRIVATE 匿名映射来分配大小大于 MMAP_THRESHOLD 字节的内存块。这样在后面将这些内存块传递给 free()之后就能高效地释放这些块（通过 munmap()）。（它还降低了重复分配和释放大内存块而导致内存分片的可能性。）MMAP_THRESHOLD 在默认情况下是 128 kB，但可以通过 mallopt()库函数来调整这个参数。

MAP_SHARED 匿名映射

MAP_SHARED 匿名映射允许相关进程（如父进程和子进程）共享一块内存区域而无需一个对应的映射文件。

MAP_SHARED 匿名映射只在 Linux 2.4 以及之后的版本上可用。

下面的代码使用 MAP_ANONYMOUS 技术创建了一个 MAP_SHARED 匿名映射。

```
addr = mmap(NULL, length, PROT_READ | PROT_WRITE,
            MAP_SHARED | MAP_ANONYMOUS, -1, 0);
if (addr == MAP_FAILED)
    errExit("mmap");
```

如果在上面的代码之后加上一个对 fork()的调用，那么由于通过 fork()创建的子进程会继承映射，两个进程就会共享内存区域。

示例程序

程序清单 49-3 演示了如何使用 MAP_ANONYMOUS 或/dev/zero 技术来在父进程和子进程之间共享一个映射区域。至于到底该选择何种技术则由在编译程序时是否定义了 USE_MAP_ANON 来确定。父进程在调用 fork()之前将共享区域中的一个整数初始化为 1。然后子进程递增这个共享整数并退出，而父进程则等待子进程退出，然后打印出该整数的值。运行这个程序之后能看到下面这样的输出。

```
$ ./anon_mmap
Child started, value = 1
In parent, value = 2
```


更多资源请访问稀酷客(www.ckook.com)

程序清单 49-3：在父进程和子进程之间共享一个匿名映射

mmap/anon_mmap.c

```
#ifdef USE_MAP_ANON
#define _BSD_SOURCE             /* Get MAP_ANONYMOUS definition */
#endif
#include <sys/wait.h>
#include <sys/mman.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int *addr;                  /* Pointer to shared memory region */

#ifdef USE_MAP_ANON             /* Use MAP_ANONYMOUS */
    addr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

#else                           /* Map /dev/zero */
    int fd;

    fd = open("/dev/zero", O_RDWR);
    if (fd == -1)
        errExit("open");

    addr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1)        /* No longer needed */
        errExit("close");
#endif

    *addr = 1;                  /* Initialize integer in mapped region */

    switch (fork()) {           /* Parent and child share mapping */
    case -1:
        errExit("fork");

    case 0:                     /* Child: increment shared integer and exit */
        printf("Child started, value = %d\n", *addr);
        (*addr)++;

        if (munmap(addr, sizeof(int)) == -1)
            errExit("munmap");
        exit(EXIT_SUCCESS);

    default:                    /* Parent: wait for child to terminate */
        if (wait(NULL) == -1)
            errExit("wait");
        printf("In parent, value = %d\n", *addr);
        if (munmap(addr, sizeof(int)) == -1)
            errExit("munmap");
        exit(EXIT_SUCCESS);
```


更多资源请访问稀酷客(www.ckook.com)

```
    }
}
```

mmap/anon_mmap.c

49.8 重新映射一个映射区域：mremap()

在大多数 UNIX 实现上一旦映射被创建，其位置和大小就无法改变了。但 Linux 提供了（不可移植的）mremap()系统调用来执行此类变更。

```
#define _GNU_SOURCE
#include <sys/mman.h>

void *mremap(void *old_address, size_t old_size, size_t new_size, int flags, ...);
```

Returns starting address of remapped region on success, or MAP_FAILED on error

old_address 和 old_size 参数指定了需扩展或收缩的既有映射的位置和大小。在 old_address 中指定的地址必须是分页对齐的，并且通常是一个由之前的 mmap()调用返回的值。映射预期的新大小会通过 new_size 参数指定。在 old_size 和 new_size 中指定的值都会被向上舍入到系统分页大小的下一个整数倍。

在执行重映射的过程中内核可能会为映射在进程的虚拟地址空间中重新指定一个位置，而是否允许这种行为则是由 flags 参数来控制的。它是一个位掩码，其值要么是 0，要么包含下列几个值。

MREMAP_MAYMOVE

如果指定了这个标记，那么根据空间要求的指令，内核可能会为映射在进程的虚拟地址空间中重新指定一个位置。如果没有指定这个标记，并且在当前位置处没有足够的空间来扩展这个映射，那么就返回 ENOMEM 错误。

MREMAP_FIXED（自 Linux 2.4 起）

这个标记只能与 MREMAP_MAYMOVE 一起使用。它在 mremap()中所起的作用与 MAP_FIXED 在 mmap()（49.10 节）中所起的作用类似。如果指定了这个标记，那么 mremap()会接收一个额外的参数 void *new_address，该参数指定了一个分页对齐的地址，并且映射将会被迁移至该地址处。所有之前在由 new_address 和 new_size 确定的地址范围之内的映射将会被解除映射。

mremap()在成功时会返回映射的起始地址。由于（如果指定了 MREMAP_MAYMOVE 标记）这个地址可能与之前的起始地址不同，从而导致指向这个区域中的指针可能会变得无效，因此使用 mremap()的应用程序在引用映射区域中的地址时应该只使用偏移量（不是绝对指针）（参见 48.6 节）。

> 在 Linux 上，realloc()函数使用 mremap()来高效地为 malloc()之前使用 mmap() MAP_ANONYMOUS 分配的大内存块重新指定位置。（在 49.7 节中介绍 glibc malloc()实现的时候曾提及过这个特性。）使用 mremap()来完成这种任务使得在重新分配空间的过程中避免复制字节成为可能。

更多资源请访问稀酷客(www.ckook.com)

49.9 MAP_NORESERVE 和过度利用交换空间

一些应用程序会创建大（通常是私有匿名的）映射，但只使用映射区域中的一小部分。如特定的科学应用程序会分配非常大的数组，但只使用其中一些散落在数组各处的元素（所谓的稀疏数组）。

如果内核总是为此类映射分配（或预留）足够的交换空间，那么很多交换空间可能会被浪费。相反，内核可以只在需要用到映射分页的时候（即当应用程序访问分页时）为它们预留交换空间。这种方法被称为懒交换预留（lazy swap reservation），它的一个优点是应用程序总共使用的虚拟内存量能够超过 RAM 加上交换空间的总量。

换个角度来看，懒交换预留允许交换空间被过度利用。这种方式能够很好地工作，只要所有进程都不试图访问整个映射。但如果所有应用程序都试图访问整个映射，那么 RAM 和交换空间就被耗尽。在这种情况下，内核会通过杀死系统中的一个或多个进程来降低内存压力。在理想情况下，内核会尝试选择引起内存问题的进程（参见下面对 OOM 杀手的讨论），但这是无法保证的。正因为这个原因，有时候可能会选择防止懒交换预留，转而强制系统在映射被创建时分配所有所需的交换空间。

内核如何处理交换空间的预留是由调用 mmap()时是否使用了 MAP_NORESERVE 标记以及影响系统层面的交换空间过度利用操作的/proc 接口来控制的。表 49-4 对这些因素进行了总结。

表 49-4：在 mmap()中处理交换空间预留

overcommit_memory 值	是否在 mmap()调用指定了 MAP_NORESERVE	
	否	是
0	拒绝明显的过度利用	允许过度利用
1	允许过度利用	允许过度利用
2（自 Linux 2.6 起）	严格的过度利用	

Linux 特有的/proc/sys/vm/overcommit_memory 文件包含了一个整数值，它控制着内核对交换空间过度利用的处理。在 2.6 之前的 Linux 上这个文件中的整数只能取两个值：0 表示拒绝明显的过度利用（遵从 MAP_NORESERVE 标记的使用），大于 0 表示在所有情况下都允许过度利用。

拒绝过度利用意味着大小不超过当前可用空闲内存的映射是被允许的。既有的分配可能会被过度利用（因为它们可能不会使用映射的所有分页）。

从 Linux 2.6 起，1 的含义与之前的内核中正数的含义一样，但 2（或更大）则会导致使用采用严格的过度利用。在这种情况下，内核会在所有 mmap()分配上执行严格的记账并将系统中此类分配的总量控制在小于或等于：

```
[swap size] + [RAM size] * overcommit_ratio / 100
```

overcommit_ratio 的值是一个整数——用百分比表示——它位于 Linux 特有的/proc/sys/vm/overcommit_ratio 文件中。这个文件中包含的默认值是 50，表示内核最多可分配的空间为系统 RAM 总量的 50%，只要所有进程不同时试图全部用完给它们分配的内存，那么这种空间的分

更多资源请访问稀酷客(www.ckook.com)

配就不会有问题。

注意过度利用监控只适用于下面这些映射。

- 私有可写映射（包括文件和匿名映射），这种映射的交换“开销”等于所有使用该映射的进程为该映射所分配的空间总和。
- 共享匿名映射，这种映射的交换“开销”等于映射的大小（因为所有进程共享该映射）。

为只读私有映射预留交换空间是没有必要的，因为映射中的内容是不可变更的，从而无需使用交换空间。共享文件映射也不需要使用交换空间，因为映射文件本身担当了映射的交换空间。

当一个子进程在 fork()调用中继承了一个映射时，它将会继承该映射的 MAP_NORESERVE 设置。MAP_NORESERVE 标记并没有在 SUSv3 中予以规定，它只在其他一些 UNIX 实现上得到了支持。

本节讨论了 mmap()调用在增长一个进程的地址空间时是如何因系统在 RAM 和交换空间上的限制而可能发生失败的。mmap()调用还可能因为碰到了进程级别的 RLIMIT_AS 资源限制（在 36.3 节中予以了介绍）而发生失败，该限制给调用进程的地址空间大小规定了一个上限。

OOM 杀手

上面提及过当使用懒交换预留时，如果应用程序试图使用整个映射的话就会导致内存被耗尽。在这种情况下，内核会通过杀死进程来缓解内存消耗情况。

内核中用来在内存被耗尽时选择杀死哪个进程的代码通常被称为 out-of-memory（OOM）杀手。OOM 杀手会尝试选择杀死能够缓解内存消耗情况的最佳进程，这里的“最佳”是由一组因素来确定的。如一个进程消耗的内存越多，它就越可能成为 OOM 杀手的候选目标。其他能提高一个进程被选中的可能性的因素包括进程是否创建了很多子进程以及进程是否拥有一个较低的 nice 值（即大于 0 的值）。内核一般不会杀死下列进程。

- 特权进程，因为它们可能正在执行重要的任务。
- 正在访问裸设备的进程，因为杀死它们可能会导致设备处理一个不可用的状态。
- 已经运行了很长时间或已经消耗了大量 CPU 的进程，因为杀死它们可能会导致丢失很多“工作”。

为杀死被选中的进程，OOM 杀手会向其发送一个 SIGKILL 信号。

从 2.6.11 内核开始，Linux 特有的/proc/PID/oom_score 文件给出了在需要调用 OOM 杀手时内核赋给每个进程的权重。在这个文件中，进程的权重越大，那么在必要的时候被 OOM 杀手选中的可能性就越大。同样也是从 2.6.11 内核开始，Linux 特有的/proc/PID/oom_adj 文件能够用来影响一个进程的 oom_score 值。这个文件可以被设置成范围在−16 到+15 之间的任意一个值，其中负数会减小 oom_score 值，而正数则会增大 oom_score 值。特殊值−17 会完全将进程从 OOM 杀手的候选目标中删除。有关这一方面的更多细节请参考 proc(5)手册。

49.10 MAP_FIXED 标记

在 mmap() flags 参数中指定 MAP_FIXED 标记会强制内核原样地解释 addr 中的地址，而不是只将其作为一种提示信息。如果指定了 MAP_FIXED，那么 addr 就必须是分页对齐的。

更多资源请访问稀酷客(www.ckook.com)

一般来讲，一个可移植的应用程序不应该使用 MAP_FIXED，并且需要将 addr 指定为 NULL，这样就允许系统选择将映射放置在哪个地址处了。之所以需要这样做的原因与 48.3 节中解释在使用 shmat()附加一个 System V 共享内存段时通常倾向于将 shmaddr 指定为 NULL 的原因是一样的。

然而，还是存在一种可移植应用程序需要使用 MAP_FIXED 的情况。如果在调用 mmap()时指定了 MAP_FIXED，并且内存区域的起始位置为 addr，覆盖的 length 字节与之前的映射的分页重叠了，那么重叠的分页会被新映射替代。使用这个特性可以可移植地将一个文件（或多个文件）的多个部分映射进一块连续的内存区域，如下所述。

1. 使用 mmap()创建一个匿名映射（参见 49.7 节）。在 mmap()调用中将 addr 指定为 NULL 并且不指定 MAP_FIXED 标记。这样就允许内核为映射选择一个地址了。
2. 使用一系列指定了 MAP_FIXED 标记的 mmap()调用来将文件区域映射（即重叠）进在上一步中创建的映射的不同部分中。

尽管可以忽略第一个步骤而直接使用一系列 mmap() MAP_FIXED 操作来在应用程序选中的地址范围内创建一组连续的映射，但这种做法的可移植性与上面这种两步式做法相比就要差一些了。上面提及过，一个可移植的应用程序应该避免在固定的地址处创建新映射。上面的第一步避免了移植性问题的出现，因为这一步让内核选择了一个连续的地址范围，然后在该地址范围中创建新映射。

从 Linux 2.6 开始，使用 remap_file_pages()系统调用（下一节介绍）也能够取得同样的效果，但使用 MAP_FIXED 的可移植性更强，因为 remap_file_pages()是 Linux 特有的。

49.11 非线性映射：remap_file_pages()

使用 mmap()创建的文件映射是连续的：映射文件的分页与内存区域的分页存在一个顺序的、一对一的对应关系。对于大多数应用程序来讲，线性映射已经够用了。然而一些应用程序需要创建大量的非线性映射——文件分页的顺序与它们在连续内存中出现的顺序不同的映射。图 49-5 给出了一种非线性映射。

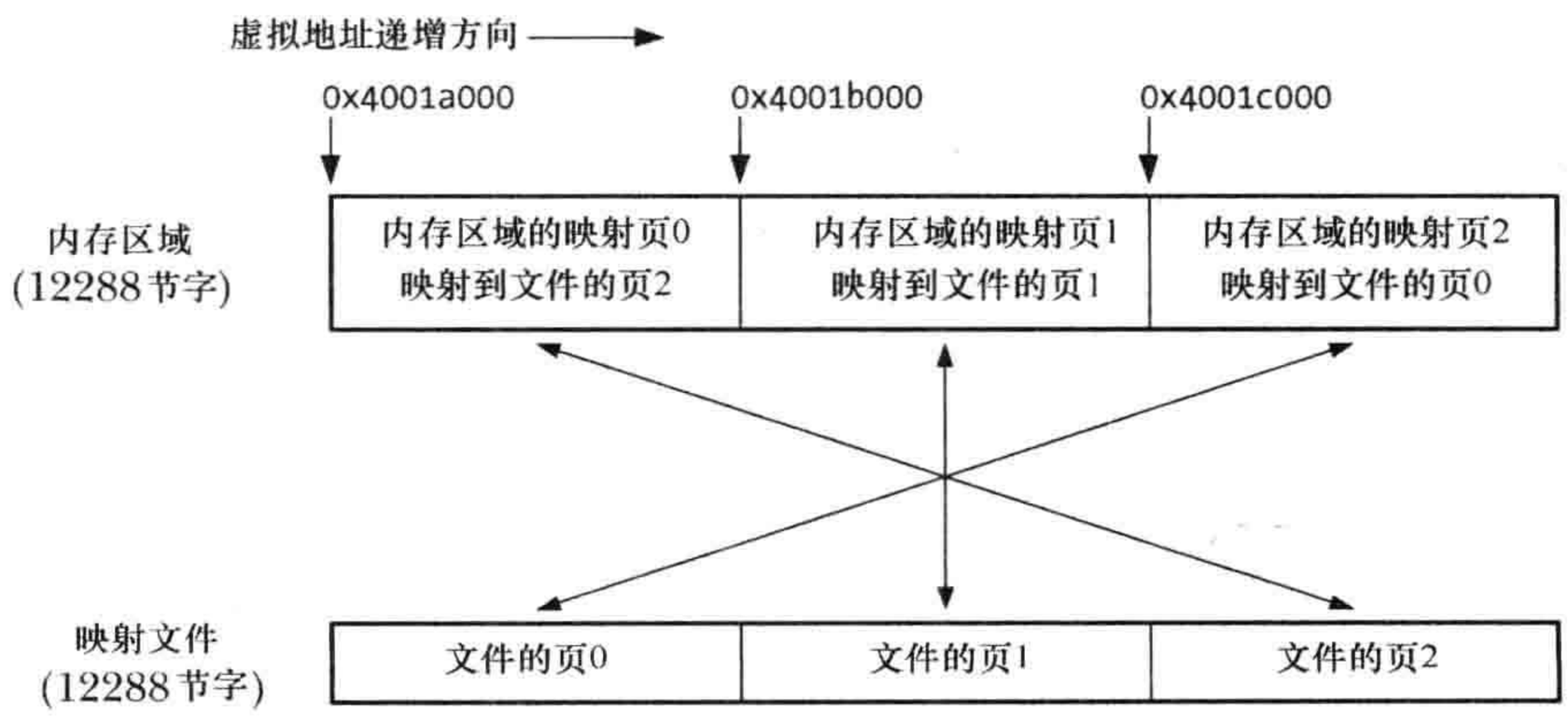


图 49-5：一个非线性文件映射

在上一节中介绍了一种创建非线性映射的方法：使用多个带 MAP_FIXED 标记的 mmap()调用。然而这种方法的伸缩性不够好，其问题在于其中每个 mmap()调用都会创建一个独立的

更多资源请访问稀酷客(www.ckook.com)

内核虚拟内存区域（VMA）数据结构。每个 VMA 的配置需要花费时间并且会消耗一些不可交换的内核内存。此外，大量的 VMA 会降低虚拟内存管理器的性能。特别地，当存在数以万计的 VMA 时处理每个分页故障所花费的时间会大幅度提高。（这对于一些在一个数据库文件中维护多个不同视图的大型数据库管理系统来讲是一个问题。）

/proc/PID/maps 文件（参见 48.5 节）中一行表示一个 VMA。

从内核 2.6 开始，Linux 提供了 remap_file_pages()系统调用来在无需创建多个 VMA 的情况下创建非线性映射，具体如下。

1. 使用 mmap()创建一个映射。
2. 使用一个或多个 remap_file_pages()调用来调整内存分页和文件分页之间的对应关系。（remap_file_pages()所做的工作是操作进程的页表。）

```
#define _GNU_SOURCE
#include <sys/mman.h>

int remap_file_pages(void *addr, size_t size, int prot, size_t pgoff, int flags);
```

Returns 0 on success, or −1 on error

pgoff 和 size 参数标识了一个在内存中的位置待改变的文件区域，pgoff 参数指定了文件区域的起始位置，其单位是系统分页代销（sysconf(_SC_PAGESIZE)的返回值）。size 参数指定了文件区域的长度，其单位为字节。addr 参数起两个作用。

- 它标识了分页需调整的既有映射。换句话说，addr 必须是一个位于之前通过 mmap()映射的区域中的地址。
- 它指定了通过 pgoff 和 size 标识出的文件分页所处的内存地址。

addr 和 size 都应该是系统分页大小的整数倍。如果不是，那么它们会被向下舍入到最近的分页大小的整数倍。

假设使用了下面的 mmap()调用来映射通过描述符 fd 引用的打开着的文件的三个分页，并且该调用将返回地址 0x4001a000 赋给了 addr。

```
ps = sysconf(_SC_PAGESIZE);                /* Obtain system page size */
addr = mmap(0, 3 * ps, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

下面的调用将会创建一个非线性映射，如图 49-5 所示。

```
remap_file_pages(addr, ps, 0, 2, 0);
                            /* Maps page 0 of file into page 2 of region */
remap_file_pages(addr + 2 * ps, ps, 0, 0, 0);
                            /* Maps page 2 of file into page 0 of region */
```

到现在为止还没有对 remap_file_pages()中的其他两个参数进行介绍。

- prot 参数会被忽略，其值必须是 0。在将来可能能够使用这个参数来修改受 remap_file_pages()影响的内存区域的保护信息。在当前实现中，保护信息保持与整个 VMA 上的保护信息一致。

虚拟机和垃圾收集器是其他一些使用多个 VMA 的应用程序，其中一些应用程序需要能够写保护单个分页。因此人们预期 remap_file_pages()将会还允许修改一个 VMA 中单个分页上的权限，但到目前为止这种特性还没有被实现。

- flags 参数当前未被使用。

更多资源请访问稀酷客(www.ckook.com)

在当前的实现上，remap_file_pages()仅适用于共享（MAP_SHARED）映射。

remap_file_pages()系统调用是 Linux 特有的，SUSv3 并没有对这个函数进行规定，并且其他 UNIX 实现也没有提供这个函数。

49.12 总结

mmap()系统调用在调用进程的虚拟地址空间中创建一个新内存映射。munmap()系统调用执行逆操作，即从进程的地址空间中删除一个映射。

映射可以分为两种：基于文件的映射和匿名映射。文件映射将一个文件区域中的内容映射到进程的虚拟地址空间中。匿名映射（通过使用 MAP_ANONYMOUS 标记或映射/dev/zero 来创建）并没有对应的文件区域，该映射中的字节会被初始化为 0。

映射既可以是私有的（MAP_PRIVATE），也可以是共享的（MAP_SHARED）。这种差别确定了在共享内存上发生的变更的可见性，对于文件映射来讲，这种差别还确定了内核是否会将映射内容上发生的变更传递到底层文件上。当一个进程使用 MAP_PRIVATE 映射了一个文件之后，在映射内容上发生的变更对其他进程是不可见的，并且也不会反应到映射文件上。MAP_SHARED 文件映射的做法则相反——在映射上发生的变更对其他进程可见并且会反应到映射文件上。

尽管内核会自动将发生在一个 MAP_SHARED 映射内容上的变更反应到底层文件上，但它不保证何时会完成这个操作。应用程序可以使用 msync()系统调用来显式地控制一个映射的内容何时与映射文件进行同步。

内存映射有很多用途，包括：

- 分配进程私有的内存（私有匿名映射）；
- 对一个进程的文本段和初始化数据段中的内容进行初始化（私有文件映射）；
- 在通过 fork()关联起来的进程之间共享内存（共享匿名映射）；
- 执行内存映射 I/O，还可以将其与无关进程之间的内存共享结合起来（共享文件映射）。

在访问一个映射的内容时可能会遇到两个信号。如果在访问映射时违反了映射之上的保护规则（或访问一个当前未被映射的地址），那么就会产生一个 SIGSEGV 信号。对于基于文件的映射来讲，如果访问的映射部分在文件中没有相关区域与之对应（即映射大于底层文件），那么就会产生一个 SIGBUS 信号。

交换空间过度利用允许系统给进程分配比实际可用的 RAM 与交换空间之和更多的内存。过度利用之所以可能是因为所有进程都不会全部用完为其分配的内存。使用 MAP_NORESERVE 标记可以控制每个 mmap()调用的过度利用情况，而使用/proc 文件则可以控制整个系统的过度利用情况。

mremap()系统调用允许调整一个既有映射的大小。remap_file_pages()系统调用允许创建非线性文件映射。

更多信息

Linux 上有关 mmap()的实现的信息可以在[Bovet &Cesati, 2005]中找到。其他 UNIX 系统上有关 mmap()的实现的信息可以在[McKusick et al., 1996] (BSD)、[Goodheart & Cox, 1994](System V Release 4)以及[Vahalia, 1996] (System V Release 4)中找到。

更多资源请访问稀酷客(www.ckook.com)

49.13 习题

49-1. 使用 mmap()和 memcpy()调用（不是 read()或 write()）编写一个类似于 cp(1)的程序来将一个源文件复制到目标文件。（使用 fstat()获取输入文件的大小，然后可以使用这个大小来设置所需的内存映射的大小，使用 ftruncate()设置输出文件的大小。）

49-2. 重写程序清单 48-2（svshm_xfr_writer.c）和程序清单 48-3（svshm_xfr_reader.c）使它们使用共享内存映射来取代 System V 共享内存。

49-3. 编写程序验证在 49.4.3 节中描述的情况下会产生 SIGBUS 和 SIGSEGV 信号。

49-4. 使用 49.10 节中介绍的 MAP_FIXED 技术编写一个程序来创建一个与图 49-5 中给出的映射类似的非线性映射。

更多资源请访问稀酷客(www.ckook.com)

第50章

虚拟内存操作

本章介绍在进程的虚拟地址空间上执行操作的各个系统调用。

- mprotect()系统调用修改一块虚拟内存区域上的保护信息。
- mlock()和 mlockall()系统调用将一块虚拟内存区域锁进物理内存，从而防止它被交换出去。
- mincore()系统调用让一个进程能够确定一块虚拟内存区域中的分页是否驻留在物理内存中。
- madvise()系统调用让一个进程能够将其对虚拟内存区域的使用模式报告给内核。

其中一些系统调用只有与共享内存区域结合起来之后才能够发挥特别的作用（第 48 章、第 49 章以及第 54 章），但它们可以被应用于一个进程的虚拟内存中的任何区域。

本章介绍的技术实际上与 IPC 一点关系也没有，之所以将本章的内容放在本书的这个部分是因为有时候将它们与共享内存结合起来使用。

50.1 改变内存保护：mprotect()

mprotect()系统调用修改起始位置为 addr 长度为 length 字节的虚拟内存区域中分页上的保护。

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

int mprotect(void *addr, size_t length, int prot);
```

Returns 0 on success, or −1 on error

addr 的取值必须是系统分页大小（sysconf(_SC_PAGESIZE)的返回值）的整数倍。（SUSv3 规定 addr 必须是分页对齐的。SUSv4 表示一个实现可以要求这个参数是分页对齐的。）由于保护是设置在整个分页上的，因此实际上 length 会被向上舍入到系统分页大小的下一个整数倍。

prot 参数是一个位掩码，它指定了这块内存区域上的新保护，其取值是 PROT_NONE 或 PROT_READ、PROT_WRITE、以及 PROT_EXEC 这三个值中的一个或多个取 OR。所有这些

更多资源请访问稀酷客(www.ckook.com)

值的含义与它们在 mmap()中的含义是一样的（表 49-2）。

如果一个进程在访问一块内存区域时违背了内存保护，那么内核就会向该进程发送一个 SIGSEGV 信号。

mprotect()的一个用途是修改原先通过 mmap()调用设置的映射内存区域上的保护，如程序清单 50-1 所示。这个程序创建了一个最初拒绝所有访问（PROT_NONE）的匿名映射，然后将该区域上的保护修改为读加写。在做出变更之前和之后，程序使用 system()函数执行了一个 shell 命令来打印出与该映射区域对应的/proc/PID/maps 文件中的内容，这样就能够看到内存保护上发生的变更了。（其实通过直接解析/proc/self/maps 就能获取映射信息，这里之所以使用 system()调用是因为这种做法所需的编码量更少。）运行这个程序之后可以看到下面的输出。

```
$ ./t_mprotect
Before mprotect()
b7cde000-b7dde000 ---s 00000000 00:04 18258    /dev/zero (deleted)
After mprotect()
b7cde000-b7dde000 rw-s 00000000 00:04 18258    /dev/zero (deleted)
```

从上面输出的最后一行可以看出 mprotect()已经将内存区域上的权限修改为 PROT_READ | PROT_WRITE。（至于在 shell 输出中为何在/dev/zero 后面出现了(deleted)字符串的原因请参考 48.5 节。）

程序清单 50-1：使用 mprotect()修改内存保护

vmem/t_mprotect.c

```
#define _BSD_SOURCE         /* Get MAP_ANONYMOUS definition from <sys/mman.h> */
#include <sys/mman.h>
#include "tlpi_hdr.h"

#define LEN (1024 * 1024)

#define SHELL_FMT "cat /proc/%ld/maps | grep zero"
#define CMD_SIZE (sizeof(SHELL_FMT) + 20)
                            /* Allow extra space for integer string */
int
main(int argc, char *argv[])
{
    char cmd[CMD_SIZE];
    char *addr;

    /* Create an anonymous mapping with all access denied */

    addr = mmap(NULL, LEN, PROT_NONE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    /* Display line from /proc/self/maps corresponding to mapping */

    printf("Before mprotect()\n");
    snprintf(cmd, CMD_SIZE, SHELL_FMT, (long) getpid());
    system(cmd);

    /* Change protection on memory to allow read and write access */

    if (mprotect(addr, LEN, PROT_READ | PROT_WRITE) == -1)
        errExit("mprotect");
```


更多资源请访问稀酷客(www.ckook.com)

```
    printf("After mprotect()\n");
    system(cmd);                /* Review protection via /proc/self/maps */

    exit(EXIT_SUCCESS);
}
```

vmem/t_mprotect.c

50.2 内存锁：mlock()和 mlockatt()

在一些应用程序中将一个进程的虚拟内存的部分或全部锁进内存以确保它们总是位于物理内存中是非常有用的。之所以需要这样做的一个原因是它可以提高性能。对被锁住的分页的访问可以确保永远不会因为分页故障而发生延迟。这对于那些需要确保快速响应时间的应用程序来讲是很有用的。

给内存加锁的另一个原因是安全。如果一个包含敏感数据的虚拟内存分页永远不会被交换出去，那么该分页的副本就不会被写入到磁盘。如果该分页被写入到了磁盘，那么从理论上来讲就可以在后面某个时刻直接从磁盘中读取该分页。（攻击者可能会故意通过运行一个消耗大量内存的程序来构造这种场景，从而强制其他进程占据的内存被交换到磁盘上。）由于内核不保证会清除交换空间中保存的数据，因此即使在进程终止之后也可能从交换空间中读取信息。（一般来讲，只有特权进程才能够从交换设备上读取数据。）

> 膝上型计算机以及一些桌面系统上的挂起模式将系统的 RAM 副本保存到磁盘上，不管是否存在内存锁。

本节将介绍用于给一个进程的虚拟内存的部分或全部进行加锁和解锁的系统调用。下面在开始介绍这些系统调用之前首先看一下管理内存加锁的资源限制。

RLIMIT_MEMLOCK 资源限制

在 36.3 节中对 RLIMIT_MEMLOCK 限制进行了简要的介绍，它为一个进程能够锁进内存的字节数设定了一个上限。下面开始对这个限制进行详细介绍。

在 2.6.9 之前的 Linux 内核中，只有特权进程（CAP_IPC_LOCK）才能给内存加锁，RLIMIT_MEMLOCK 软资源限制为一个特权进程能够锁住的字节数设定一个上限。

从 Linux 2.6.9 开始，内存加锁模型发生了变化，即允许非特权进程给一小段内存进行加锁。这对于那些需要将一小部分敏感信息锁进内存以确保这些信息永远不会被写入到磁盘上的交换空间的应用程序来讲是非常有用的，如 gpg 是通过密码短语来完成这件事情的。这种模型上的变更会导致：

- 特权进程能够锁住的内存数量是没有限制的（即 RLIMIT_MEMLOCK 会被忽略）；
- 非特权进程能够锁住的内存数量上限由软限制 RLIMIT_MEMLOCK 定义。

软和硬 RLIMIT_MEMLOCK 限制的默认值都是 8 个分页（即在 x86-32 上是 32768 字节）。RLIMIT_MEMLOCK 限制影响：

- mlock()和 mlockall()；
- mmap() MAP_LOCKED 标记，该标记用来在映射被创建时将内存映射锁进内存，具体可参见 49.6 节中的描述；
- shmctl() SHM_LOCK 操作，该操作用来给 System V 共享内存段加锁，具体可参见 48.7

更多资源请访问稀酷客(www.ckook.com)

节中的描述。

由于虚拟内存的管理单位是分页，因此内存加锁会应用于整个分页。在执行限制检查时，RLIMIT_MEMLOCK 限制会被向下舍入到最近的系统分页大小的整数倍。

尽管这个资源限制只有一个（软）值，但实际上它定义了两个单独的限制：

- 对于 mlock()、mlockall()以及 mmap() MAP_LOCKED 操作来讲，RLIMIT_ MEMLOCK 定义了一个进程级别的限制，它限制了一个进程的虚拟地址空间中能够被锁进内存的字节数。
- 对于 shmctl() SHM_LOCK 操作来讲，RLIMIT_MEMLOCK 定义了一个用户级别的限制，它限制了这个进程的真实用户 ID 在共享内存段中能够锁住的字节数。当一个进程执行了一个 shmctl() SHM_LOCK 操作时，内核会检查被调用进程的真实用户 ID 锁住的 System V 共享内存的总字节数。如果待加锁的段的大小不会导致总量违背进程的 RLIMIT_MEMLOCK 限制，那么操作就会成功。

RLIMIT_MEMLOCK 在 System V 共享内存上之所以存在不同语义是因为共享内存段即使在没有被附加到任何一个进程之上时也是能够继续存在的。（共享内存段只有在显式的 shmctl() IPC_RMID 操作之后并且所有进程都在它们的地址空间中与之分离之后才会被删除。）

给内存区域加锁和解锁

一个进程可以使用 mlock()和 munlock()来给一块内存区域加锁和解锁。

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

int mlock(void *addr, size_t length);
int munlock(void *addr, size_t length);
```

Both return 0 on success, or −1 on error

mlock()系统调用会锁住调用进程的虚拟地址空间中起始地址为 addr 长度为 length 字节的区域中的所有分页。与传入其他一些与内存相关的系统调用中的相应参数相比，这里的 addr 无需是分页对齐的：内核会从 addr 下面的下一个分页边界开始锁住分页。然而，SUSv3 允许一个实现要求 addr 为系统分页大小的整数倍，可移植的应用程序在调用 mlock()和 munlock()应该确保这一点。

由于加锁操作的单位是分页，因此被锁住的区域的结束位置为大于 length 加 addr 的下一个分页边界。例如，在一个分页大小为 4096 字节的系统上，mlock(2000, 4000)调用会将 0 到 8191 之间的字节锁住。

通过查看 Linux 特有的/proc/PID/status 文件中的 VmLck 条目能够找出一个进程当前已经锁住的内存数量。

在 mlock()调用成功之后就能确保指定区域中的分页会被锁住并驻留在物理内存中。当没有足够的物理内存来锁住所有所请求的分页或请求违背 RLIMIT_MEMLOCK 软资源限制时 mlock()系统调用就会失败。

程序清单 50-2 给出了一个使用 mlock()的例子。

munlock()系统调用执行的操作与 mlock()相反，即删除之前由调用进程创建的内存锁。addr 和 length 参数被解释的方式与它们在 munlock()中被解释的方式是相同的。给一组分页解锁并不能确保它们就不会驻留在内存中了：只有在其他进程请求内存的时候才会从 RAM 中删除分页。

除了显式地使用 munlock()之外，内存锁在下列情况下会被自动删除。

更多资源请访问稀酷客(www.ckook.com)

- 在进程终止时。
- 当被锁住的分页通过 munmap()被解除映射时。
- 当被锁住的分页被使用 mmap() MAP_FIXED 标记的映射覆盖时。

内存加锁语义的细节信息

在下面的几个段落中将会介绍内存锁语义的一些细节。

内存锁不会被通过 fork()创建的子进程继承，也不会在 exec()执行期间被保留。

当多个进程共享一组分页时（如 MAP_SHARED 映射），只要还存在一个进程持有着这些分页上的内存锁，那么这些分页就会保持被锁进内存的状态。

内存锁不在单个进程上叠加。如果一个进程重复地在一个特定虚拟地址区域上调用 mlock()，那么只会建立一个锁，并且只需要通过一个 munlock()调用就能够删除这个锁。另一方面，如果使用 mmap()将同一组分页（即同样的文件）映射到单个进程中的几个不同的位置，然后分别给所有这些映射加锁，那么这些分页会保持被锁进 RAM 的状态直到所有的映射都被解锁为止。

内存锁的加锁单位为分页以及无法叠加的事实意味着独立地将 mlock()和 munlock()调用应用于同一个虚拟分页上的不同数据结构在逻辑上是不正确的。如假设在同一个虚拟内存分页中存在两个数据结构，指针 p1 和 p2 分别指向了这两个结构，接着执行下面的调用。

```
mlock(*p1, len1);
mlock(*p2, len2);                /* Actually has no effect */
munlock(*p1, len1);
```

上面的所有调用都会成功，但最后整个分页都会被解锁，即 p2 指向的数据结构将不会被锁进内存。

注意 shmctl() SHM_LOCK 操作（48.7 节）的语义与 mlock()和 mlockall()的语义是不同的，具体如下。

- 在 SHM_LOCK 操作之后，分页只有在因后续引用而发生故障时才会被锁进内存。与之相反的是，mlock()和 mlockall()调用在返回之前会将所有分页锁进内存。
- SHM_LOCK 操作会设置共享内存段的一个属性，而不是进程的属性。（正因为这个原因，/proc/PID/status VmLck 字段的值中并没有包含使用 SHM_LOCK 锁住的所有附加 System V 共享内存段的大小。）这意味着分页一旦因故障被锁进了内存，那么即使所有进程都与这个共享内存段分离了，分页还是会保持驻留在内存中的状态。与之相反的是，使用 mlock()（或 mlockall()）锁进内存的区域只有在还存在进程持有该区域上的锁时才会保持被锁进内存的状态。

给一个进程占据的所有内存加锁和解锁

一个进程可以使用 mlockall()和 munlockall()给它占据的所有内存加锁和解锁。

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

int mlockall(int flags);
int munlockall(void);
```

Both return 0 on success, or −1 on error

mlockall()系统调用根据 flags 位掩码的取值将一个进程的虚拟地址空间中当前所有映射的

更多资源请访问稀酷客(www.ckook.com)

分页或将来所有映射的分页或两者锁进内存，其中 flags 参数的取值为下面这些常量中的一个或多个取 OR。

MCL_CURRENT

将调用进程的虚拟地址空间中当前所有映射的分页锁进内存，包括当前为程序文本段、数据段、内存映射以及栈分配的所有分页。当指定了 MCL_CURRENT 标记的调用成功之后就能够确保调用进程的所有这些分页都驻留在了内存中。这个标记不会对后续在进程的虚拟地址空间中分配的分页产生影响；要控制这些分页则必须要使用 MCL_FUTURE。

MCL_FUTURE

将后续映射进调用进程的虚拟地址空间的所有分页锁进内存。例如，此类分页可能是通过 mmap()或 shmat()映射的一个共享内存区域的一部分，或向上增长的堆或向下增长的栈的一部分。指定 MCL_FUTURE 标记的结果是后续的内存分配操作（如 mmap()、sbrk()或 malloc()）可能会失败，或者栈增长可能会产生 SIGSEGV 信号，当然前提是系统已经没有 RAM 分配给进程或者已经达到了 RLIMIT_MEMLOCK 软资源限制。

通过 mlock()创建的内存锁上有关约束、生命周期以及继承性方面的规则同样也适用于通过 mlockall()创建的内存锁。

munlockall()系统调用将调用进程的所有分页解锁并撤销之前的 mlockall(MCL_ FUTURE)调用所产生的结果。与 munlock()一样，这个调用无法保证会从 RAM 中删除被解锁的分页。

在 Linux 2.6.9 之前，调用 munlockall()需要特权（CAP_IPC_LOCK）（不一致性，munlock()无需特权）。从 Linux 2.6.9 开始已经不再需要特权了。

50.3 确定内存驻留性：mincore()

mincore()系统调用是内存加锁系统调用的补充，它报告在一个虚拟地址范围中哪些分页当前驻留在 RAM 中，因此在访问这些分页时也不会导致分页故障。

SUSv3 并没有规定 mincore()，很多 UNIX 实现都提供了这个函数，但不是所有的 UNIX 实现都提供了这个函数。在 Linux 上从内核 2.4 开始提供了 mincore()。

```
#define _BSD_SOURCE          /* Or: #define _SVID_SOURCE */
#include <sys/mman.h>

int mincore(void *addr, size_t length, unsigned char *vec);
```

Returns 0 on success, or −1 on error

mincore()系统调用返回起始地址为 addr 长度为 length 字节的虚拟地址范围中分页的内存驻留信息。addr 中的地址必须是分页对齐的，并且由于返回的信息是有关整个分页的，因此 length 实际上会被向上舍入到系统分页大小的下一个整数倍。

内存驻留相关的信息会通过 vec 返回，它是一个数组，其大小为(length + PAGE_SIZE – 1) / PAGE_SIZE 字节。（在 Linux 上，vec 的类型是 unsigned char *；在其他一些 UNIX 实现上，vec 的类型为 char *。）每个字节的最低有效位在相应分页驻留在内存中时会被设置，而其他位的设置在一些 UNIX 实现上是未定义的，因此可移植的应用程序应该只测试最低有效位。

mincore()返回的信息在执行调用的时刻与检查 vec 中的元素的时刻期间可能会发生变化。唯一能够确保保持驻留在内存中的分页是那些通过 mlock()或 mlockall()锁住的分页。

更多资源请访问棒酷客(www.ckook.com)

在 Linux 2.6.21 之前，各种各样的实现问题导致 mincore()无法正确地报告 MAP_PRIVATE 映射和非线性映射（通过使用 remap_file_pages()创建）的内存驻留信息。

程序清单 50-2 演示了如何使用 mlock()和 mincore()。这个程序首先分配并使用 mmap()映射了一块内存区域，然后以固定的时间间隔使用 mlock()将整个区域或一组分页锁进内存。（传给这个程序的所有命令行参数的单位是分页，程序会将这些参数转换成字节，因为 mmap()、mlock()以及 mincore()使用的是字节。）在调用 mlock()之前和之后，程序使用 mincore()来获取这个区域中分页的内存驻留信息并图形化地将这些信息展现了出来。

程序清单 50-2：使用 mlock()和 mincore()

vmem/memlock.c

```
#define _BSD_SOURCE     /* Get mincore() declaration and MAP_ANONYMOUS
                           definition from <sys/mman.h> */
#include <sys/mman.h>
#include "tlpi_hdr.h"

/* Display residency of pages in range [addr .. (addr + length - 1)] */

static void
displayMincore(char *addr, size_t length)
{
    unsigned char *vec;
    long pageSize, numPages, j;

    pageSize = sysconf(_SC_PAGESIZE);

    numPages = (length + pageSize - 1) / pageSize;
    vec = malloc(numPages);
    if (vec == NULL)
        errExit("malloc");
    if (mincore(addr, length, vec) == -1)
        errExit("mincore");

    for (j = 0; j < numPages; j++) {
        if (j % 64 == 0)
            printf("%s%10p: ", (j == 0) ? "" : "\n", addr + (j * pageSize));
        printf("%c", (vec[j] & 1) ? '*' : '.');
    }
    printf("\n");

    free(vec);
}

int
main(int argc, char *argv[])
{
    char *addr;
    size_t len, lockLen;
    long pageSize, stepSize, j;

    if (argc != 4 || strcmp(argv[1], "--help") == 0)
        usageErr("%s num-pages lock-page-step lock-page-len\n", argv[0]);

    pageSize = sysconf(_SC_PAGESIZE);
```

```
    if (pageSize == -1)
        errExit("sysconf(_SC_PAGESIZE)");

    len =      getInt(argv[1], GN_GT_0, "num-pages") * pageSize;
    stepSize = getInt(argv[2], GN_GT_0, "lock-page-step") * pageSize;
    lockLen =  getInt(argv[3], GN_GT_0, "lock-page-len") * pageSize;

    addr = mmap(NULL, len, PROT_READ, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    printf("Allocated %ld (%#lx) bytes starting at %p\n",
            (long) len, (unsigned long) len, addr);

    printf("Before mlock:\n");
    displayMincore(addr, len);

    /* Lock pages specified by command line arguments into memory */

    for (j = 0; j + lockLen <= len; j += stepSize)
        if (mlock(addr + j, lockLen) == -1)
            errExit("mlock");

    printf("After mlock:\n");
    displayMincore(addr, len);

    exit(EXIT_SUCCESS);
}
```

vmem/memlock.c

下面的 shell 会话给出了运行程序清单 50-2 中的程序时输出。在这个例子中分配了 32 个分页，每组为 8 个分页，并给三个连续分页加锁。

```
$ su                                          Assume privilege
Password:
# ./memlock 32 8 3
Allocated 131072 (0x20000) bytes starting at 0x4014a000
Before mlock:
0x4014a000: ................................
After mlock:
0x4014a000: ***.....***.....***.....***.....
```

在程序输出中，点表示分页不在内存中，星号表示分页驻留在内存中。从最后一行输出中可以看出，每组 8 个分页中有 3 个分页是驻留在内存中的。

在这个例子中假设了超级用户特权，这样程序就能够使用 mlock()。从 Linux 2.6.9 开始就无需这种特权了，只要待加锁的内存量不超过 RLIMIT_MEMLOCK 软资源限制即可。

50.4 建议后续的内存使用模式：madvise()

madvise()系统调用通过通知内核调用进程对起始地址为 addr 长度为 length 字节的范围之内分页的可能的使用情况来提升应用程序的性能。内核可能会使用这种信息来提升在分页之下的文件映射上执行的 I/O 的效率。（有关文件映射的讨论可参考 49.4 节。）在 Linux 上从内核 2.4 开始提供了 madvise()。

更多资源请访问稀酷客(www.ckook.com)

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

int madvise(void *addr, size_t length, int advice);
```

Returns 0 on success, or -1 on error

addr 中的值必须是分页对齐的，length 实际上会被向上舍入到系统分页大小的下一个整数倍。advice 参数的取值为下列之一。

MADV_NORMAL

这是默认行为。分页是以簇的形式（较小的一个系统分页大小的整数倍）传输的。这个值会导致一些预先读和事后读。

MADV_RANDOM

这个区域中的分页会被随机访问，这样预先读将不会带来任何好处，因此内核在每次读取时所取出的数据量应该尽可能少。

MADV_SEQUENTIAL

在这个范围中的分页只会被访问一次，并且是顺序访问，因此内核可以激进地预先读，并且分页在被访问之后就可以将其释放了。

MADV_WILLNEED

预先读取这个区域中的分页以备将来的访问之需。MADV_WILLNEED 操作的效果与 Linux 特有的 readahead()系统调用和 posix_fadvise() POSIX_FADV_WILLNEED 操作的效果类似。

MADV_DONTNEED

调用进程不再要求这个区域中的分页驻留在内存中。这个标记的精确效果在不同 UNIX 实现上是不同的。下面首先对其在 Linux 上的行为予以介绍。对于 MAP_PRIVATE 区域来讲，映射分页会显式地被丢弃，这意味着所有发生在分页上的变更会丢失。虚拟内存地址范围仍然可访问，但对各个分页的下一个访问将会导致一个分页故障和分页的重新初始化，这种初始化要么使用其映射的文件内容，要么在匿名映射的情况下就使用零来初始化。这个标记可以作为一种显式初始化一个 MAP_PRIVATE 区域的内容的方法。对于 MAP_SHARED 区域来讲，内核在一些情况下可能会丢弃修改过的分页，这取决于运行系统的架构（在 x86 上不会发生这种行为）。其他一些 UNIX 实现的行为方式与 Linux 一样，但在一些 UNIX 实现上，MADV_DONTNEED 仅仅是通知内核指定的分页在必要的时候可以被交换出去。可移植的应用程序不应该依赖于 MADV_DONTNEED 在 Linux 上的破坏性语义。

Linux 2.6.16 增加了三个新的非标准 advice 值：MADV_DONTFORK、MADV_ DOFORK 以及 MADV_REMOVE。Linux 2.6.32 和 2.6.33 又增加了四个非标准的 advice 值：MADV_HWPOISON、MADV_SOFT_OFFLINE、MADV_MERGEABLE 以及 MADV_UNMERGEABLE。这些值是在特殊情况下使用的，具体可参考 madvise(2)手册。

大多数 UNIX 实现都提供了一个 madvise()，它们通常至少支持上面描述的 advice 常量。然而 SUSv3 使用了一个不同的名称来标准化了这个 API，即 posix_madvise()，并且在相应的 advice 常量上加上了一个前缀字符串 POSIX_。因此，这些常量变成了 POSIX_MADV_NORMAL、POSIX_MADV_RANDOM、POSIX_MADV_SEQUENTIAL、POSIX_MADV_WILLNEED 以

更多资源请访问稀酷客(www.ckook.com)

及 POSIX_MADV_DONTNEED。在 glibc 中（2.2 以及之后的版本）是通过调用 madvise()来实现这个候选接口的，但所有 UNIX 实现都没有提供这个接口。

SUSv3 表示 posix_madvise()不应该影响一个程序的语义。然而在版本 2.7 之前的 glibc 中，POSIX_MADV_DONTNEED 操作是通过使用 madvise() MADV_DONTNEED 来实现的，而正如之前描述的那样，这个操作会影响一个程序的语义。自 glibc 2.7 开始，posix_madvise()包装器将 POSIX_MADV_DONTNEED 实现为不做任何事情，这样它就不会影响一个程序的语义了。

50.5 小结

本章对可在一个进程的虚拟内存上执行的各种操作进行了介绍。

- mprotect()系统调用修改一块虚拟内存区域上的保护。
- mlock()和 mlockall()系统调用将一个进程的虚拟地址空间中的部分或全部分别锁进物理内存。
- mincore()系统调用报告一块虚拟内存区域中哪些分页当前驻留在物理内存中。
- madvise()系统调用和 posix_madvise()函数允许一个进程将其预期的内存使用模式报告给内核。

50.6 习题

50-1. 编写一个程序使其为 RLIMIT_MEMLOCK 资源限制设置一个值之后将数量超过这个限制的内存锁进内存来验证 RLIMIT_MEMLOCK 资源限制的作用。

50-2. 写一个程序来验证 madvise() MADV_DONTNEED 操作在一个可写 MAP_PRIVATE 映射上的操作。

更多资源请访问稀酷客(www.ckook.com)