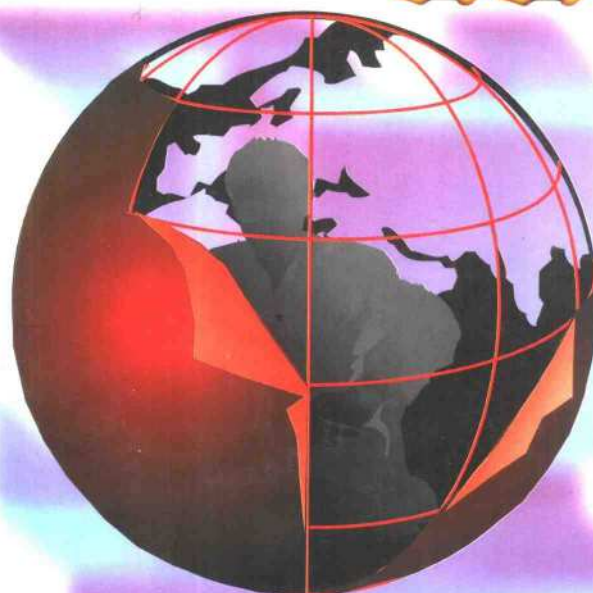


Linux 风暴系列

Linux C 高级程序员指南

毛曙福 编著



国防工业出版社



风暴系列

Linux C 高级程序员指南

毛曙福 编著

国防工业出版社

·北京·

图书在版编目(CIP)数据

Linux C 高级程序员指南/毛曙福编著. —北京:国防工业出版社,2001.2
(Linux 风暴系列)
ISBN 7-118-02439-2

I .L... II .毛... III .C 语言-程序设计 IV .TP312

中国版本图书馆 CIP 数据核字(2000)第 59242 号

国防工业出版社出版发行

(北京市海淀区紫竹院南路 23 号)

(邮政编码 100044)

三河市腾飞胶印厂印刷

新华书店经售

*

开本 787×1092 1/16 印张 26¼ 599 千字

2001 年 2 月第 1 版 2001 年 2 月北京第 1 次印刷

印数:1—3000 册 定价:35.00 元

(本书如有印装错误,我社负责调换)

序 言

Linux 是计算机发展历史上的独特现象。虽然它滥觞于一位普通大学生的灵感与才思,却已成为当今最为流行的免费操作系统。对很多人来说,Linux 是一个谜,免费的东西怎么会变得如此有价值?事实上 Linux 的确稳定而富有竞争力,许多大学与研究机构都使用 Linux 完成他们的日常计算任务,同时,Linux 也逐渐成为各公司服务器的首选操作系统,许多公司将它用于邮件服务器或是 WWW 服务器,用 DNS、路由和防火墙等。相信在不久的将来,人们在家用 PC 上也会广泛使用 Linux。

近年来 Linux 在中国也有了很大的发展,特别是随着 Internet 的普及,Linux 的发展更是如火如荼,国内很多城市都成立了 Linux 俱乐部,Linux 发烧友队伍日益壮大,从而掀起了一场 Linux 风暴。

正是在 Linux 蓬勃发展之际,我们组织编写了本套丛书,旨在为一部分读者解开 Linux 成功之谜,更为 Linux 在中国的普及和发展贡献一份力量。

国防工业出版社计算机编辑室

前 言

Linux 起源于一个学生的简单需求。Linus Torvalds, Linux 的作者与主要维护者, 在他上大学时所买得起的惟一软件是 Minix。Minix 是一个类似 Unix, 被广泛用来辅助教学的简单操作系统。Linus 对 Minix 不是很满意, 于是决定自己编写软件。他以学生时代熟悉的 Unix 作为原型, 在一台 Intel 386 PC 上开始了他的工作。他的进展很快, 受工作成绩的鼓舞, 他将这项成果通过互连网与其他同学共享, 主要用于学术领域。有人看到了这个软件并开始分发。每当出现新问题时, 有人会立刻找到解决办法并加入其中, 很快, Linux 成为了一个操作系统。值得注意的是 Linux 并没有包括 Unix 源码。它是按照公开的 POSIX 标准重新编写的。Linux 大量使用了由麻省剑桥自由软件基金会的 GNU 软件, 同时 Linux 自身也是用它们构造而成。

近年来 Linux 在中国也有了很大的发展, 特别是随着 Internet 的普及, Linux 的发展更是迅猛如潮, 国内的很多城市都成立了 Linux 俱乐部。同时, Linux 操作系统也逐渐成为各公司服务器的首选操作系统, 他们用 Linux 来作为邮件服务器或是 WWW 的服务器, Linux 还被用于各种网络应用, 如 DNS、路由和防火墙。

但是目前还有很多因素制约着 Linux 进入寻常百姓家, 首要的是硬件的驱动程序不全, 很多新的硬件不支持, 或者说厂商没有开发出 Linux 下的驱动程序; 另一个因素应用程序太少, 比在 Windows 下的应用程序少很多。所以, 要大力开发各种驱动程序和应用程序, 以适应 Linux 的蓬勃发展, 为此, 我们编写了本书, 旨在为广大 Linux 程序员排忧解难。

本书分为四大部分: 系统管理、Linux 系统编程、Linux 网络编程和 GTK 图形界面编程。第一部分(第 1~2 章)重点介绍了对用户的管理、对文件系统的管理和网络服务器的配置等内容。第二部分(第 3~10 章)重点讨论了 Linux 文件系统、Linux 文件系统调用、Linux 设备文件、Linux 进程管理、信号处理、基本进程间通信、临界区和高级进程间通信和 Linux 线程。第三部分(第 11~19 章)重点讨论了 TCP/IP 网络、套接字编程、客户机服务器编程、复杂服务器设计、远程过程调用(RPC)和 RPC 认证。第四部分(第 20~26 章)重点讨论了 GTK 图形界面编程。本书内容翔实, 不管是 Linux 普通用户, 还是 Linux 开发人员, 在阅读完本书后, 都会有很大的收获。

本书主要由毛曙福编写, 同时参与编写的人员还有: 薛小香、林章庆、刘元高、郑桂水、黄建森、康永宏、李春鹤、冯曙红、袁军、林振宁、周成福、陈培、邓冰、黄强、徐晓春、陈良程、郑国鸿、黄重阳、黄林、岑进华、林世永、郑吉林等, 在此表示感谢。

由于本书完成时间匆忙, 加上作者水平有限, 书中难免会有错误和不妥之处, 恳请读者批评指正, 作者的 E-Mail 地址是: maosf@263.net。

目 录

第 1 章 Linux 系统管理	1
1.1 登录与注销	1
1.1.1 登录 Linux	1
1.1.2 更改口令	1
1.1.3 了解 shell	2
1.1.4 了解 shell 环境	3
1.1.5 配置 shell 环境	4
1.1.6 注销	4
1.2 信息查询命令	5
1.2.1 date 命令	5
1.2.2 df 命令	6
1.2.3 du 命令	6
1.2.4 file 命令	7
1.2.5 hostname 命令	7
1.2.6 id 命令	8
1.2.7 ps 命令	8
1.2.8 quota 命令	9
1.2.9 stty 命令	9
1.2.10 time 命令	10
1.2.11 tty 命令	10
1.2.12 w 命令	10
1.2.13 whereis 命令	11
1.2.14 who 命令	11
1.3 Linux 装载程序 LILO	12
1.3.1 配置 LILO	12
1.3.2 使用 LILO	12
1.4 管理用户	12
1.4.1 添加用户	13
1.4.2 使用 adduser 命令	13
1.4.3 设置用户口令	13
1.4.4 删除用户	14

1.5 管理用户组.....	14
1.5.1 添加用户组.....	14
1.5.2 删除用户组.....	15
1.6 管理文件系统.....	15
1.6.1 安装文件系统.....	15
1.6.2 卸下文件系统.....	16
1.6.3 维护文件系统.....	17
1.7 配置 TCP/IP 网络.....	17
1.7.1 了解 TCP/IP 配置文件.....	17
1.7.2 主机配置文件.....	18
1.7.3 初始化以太网接口.....	18
1.7.4 用 ifconfig 检查网络接口.....	19
1.7.5 配置软件回送接口.....	20
1.7.6 配置网络接口.....	20
1.8 小结.....	20
第 2 章 网络服务器的配置	21
2.1 配置 FTP 服务器	21
2.1.1 FTP 服务器简介	21
2.1.2 安装 FTP 服务器	21
2.1.3 查看 FTP 服务器的设置	22
2.1.4 配置访问控制文件 ftpaccess	23
2.1.5 配置用户控制文件 ftpusers	28
2.1.6 配置主机控制文件 ftphosts	29
2.1.7 测试服务器是否正常工作.....	29
2.1.8 wu-ftp 提供的几个程序	30
2.2 配置 WWW 服务器	30
2.2.1 WWW 服务器简介	30
2.2.2 Apache 服务器介绍	31
2.2.3 安装 Apache 服务器	31
2.2.4 配置服务器控制文件 httpd.conf	31
2.2.5 配置访问控制文件 access.conf	37
2.2.6 配置资源控制文件 srm.conf	38
2.2.7 运行服务器程序.....	42
2.2.8 测试服务器运行情况.....	43
2.2.9 浏览个人主页.....	43
2.3 小结.....	44
第 3 章 Linux 文件系统	45

3.1 目录	45
3.1.1 目录结构	45
3.1.2 getcwd 函数	46
3.1.3 读取目录	47
3.2 文件	49
3.2.1 文件的存储	49
3.2.2 文件的存储权限	50
3.2.3 stat 和 fstat 函数	51
3.2.4 得到用户的信息	55
3.3 文件系统信息	58
3.3.1 ustat 函数	58
3.3.2 statfs 和 fstatfs 函数	60
3.4 小结	62
第4章 Linux 文件系统调用	63
4.1 文件描述符	63
4.2 open 和 close 函数	63
4.3 read 和 write 函数	64
4.4 lseek 函数	67
4.5 link 和 unlink 函数	71
4.6 access 函数	72
4.7 chmod, chown 和 chdir 函数	74
4.8 mkdir 和 rmdir 函数	75
4.9 mknod 函数	75
4.10 dup 和 dup2 函数	76
4.11 小结	77
第5章 Linux 设备文件	78
5.1 设备文件简介	78
5.2 设备文件的创建	78
5.3 终端设备文件	78
5.3.1 终端设备文件的读写	78
5.3.2 终端设备文件的控制	80
5.4 软盘设备文件	87
5.4.1 软盘设备文件的读写	87
5.4.2 软盘的外挂和 sync 函数	90
5.5 小结	93
第6章 Linux 进程管理	94
6.1 Linux 中的进程	94

6.1.1 task-struct 结构	94
6.1.2 进程状态	95
6.1.3 进程标识	96
6.1.4 进程调度	96
6.1.5 Linux 进程调度算法	97
6.2 Linux 进程系统调用	97
6.2.1 fork 与 vfork 函数	97
6.2.2 exec 函数	101
6.2.3 exit 与 _exit 函数	104
6.2.4 wait 与 waitpid 函数	104
6.3 小结	105
第 7 章 信号处理	106
7.1 信号简介	106
7.2 信号类别	106
7.3 关于信号的系统调用	107
7.3.1 kill 命令及 kill 函数	107
7.3.2 有关信号集合的调用	108
7.3.3 signal 与 sigaction 函数	110
7.3.4 信号处理的另外一些调用	112
7.3.5 pause 与 sigsuspend 函数	113
7.3.6 siglongjmp 与 sigsetjmp 函数	118
7.4 小结	120
第 8 章 基本进程间通信	121
8.1 管道通信	121
8.1.1 普通管道与 pipe 函数	121
8.1.2 命名管道与 mknod 函数	124
8.2 消息	129
8.2.1 msgget 函数	129
8.2.2 msgctl 函数	129
8.2.3 msgsnd 和 msgrcv 函数	130
8.3 小结	134
第 9 章 临界区与高级进程间通信	135
9.1 竞争现象与临界区	135
9.2 信号量	136
9.2.1 简介	136
9.2.2 信号量集	137
9.2.3 semget 函数	137

9.2.4	semctl 函数	138
9.2.5	semop 函数	139
9.3	共享内存	142
9.3.1	shmget 函数	142
9.3.2	shmat 函数	142
9.3.3	shmdt 函数	143
9.3.4	shmctl 函数	143
9.3.5	生产者/消费者问题	144
9.4	小结	151
第 10 章	Linux 线程	152
10.1	线程简介	152
10.1.1	传统进程的局限性	152
10.1.2	线程的动机	152
10.1.3	多线程和多处理器	153
10.1.4	线程概念	153
10.1.5	用户线程与内核线程	153
10.2	线程管理	155
10.2.1	pthread_create 函数	155
10.2.2	pthread_self 函数	156
10.2.3	pthread_exit 函数	156
10.2.4	pthread_join 函数	156
10.2.5	线程的例子	156
10.3	线程属性	159
10.3.1	线程属性对象的初始化和销毁	160
10.3.2	线程堆栈的大小	161
10.3.3	线程堆栈的地址	161
10.3.4	线程的拆卸状态	161
10.3.5	线程的作用域	162
10.3.6	线程的继承性	162
10.3.7	线程的调度策略	162
10.3.8	线程的调度参数	163
10.3.9	得到线程的属性	163
10.4	小结	166
第 11 章	TCP/IP 简介	167
11.1	网络简介	167
11.2	TCP/IP 及相关协议	167
11.2.1	IP 协议	167

11.2.2	ICMP 协议	169
11.2.3	ARP 协议	171
11.2.4	TCP 协议	172
11.2.5	UDP 协议	175
11.2.6	DNS 协议	176
11.3	小结	179
第 12 章	各种转换	180
12.1	网络字节序转换函数	180
12.2	IP 地址的转换	180
12.2.1	inet_aton 与 inet_addr 函数	180
12.2.2	inet_pton 与 inet_ntop 函数	181
12.3	名字地址的转换	182
12.3.1	gethostbyname 函数与 gethostbyname2 函数	182
12.3.2	gethostbyaddr 函数	183
12.3.3	uname 和 gethostname 函数	183
12.3.4	得到主机的信息	184
12.4	服务名的转换	186
12.4.1	getservbyname 函数	187
12.4.2	getservbyport 函数	187
12.5	高级地址转换	188
12.5.1	getaddrinfo 函数	188
12.5.2	getnameinfo 函数	190
12.6	小结	190
第 13 章	套接字编程	191
13.1	套接字简介	191
13.2	套接字编程调用	191
13.2.1	socket 函数	192
13.2.2	connect 函数	193
13.2.3	bind 函数	193
13.2.4	listen 函数	194
13.2.5	accept 函数	194
13.2.6	read 函数	195
13.2.7	recv 函数	195
13.2.8	recvfrom 函数	196
13.2.9	write 函数	196
13.2.10	send 函数	196
13.2.11	sendto 函数	196

13.2.12 close 函数	197
13.2.13 getsockname 和 getpeername 函数	197
13.3 gettime 程序	198
13.4 ourhead.h 文件	200
13.5 小结	202
第 14 章 客户机服务器编程	203
14.1 TCP 套接字编程	203
14.1.1 简介	203
14.1.2 客户机程序的简化	204
14.1.3 服务器程序的并发	205
14.1.4 ECHO 客户机程序的 TCP 版本	207
14.1.5 ECHO 服务器程序的 TCP 版本	208
14.2 UDP 套接字编程	211
14.2.1 简介	211
14.2.2 客户机程序的简化	211
14.2.3 ECHO 客户机程序的 UDP 版本 1	213
14.2.4 ECHO 客户机程序的 UDP 版本 2	215
14.2.5 ECHO 服务器程序的 UDP 版本	218
14.3 小结	219
第 15 章 复杂服务器设计	220
15.1 多协议服务器	220
15.1.1 多协议服务器简介	220
15.1.2 select 函数	220
15.1.3 ECHO 服务器的 TCP/UCP 合并版	221
15.1.4 ECHO 客户机的 TCP/UCP 合并版	225
15.2 多服务服务器	227
15.2.1 多服务服务器简介	227
15.2.2 ECHO, DAYTIME 多服务服务器程序	227
15.2.3 ECHO, DAYTIME 多服务客户机程序	231
15.2.4 多协议多服务服务器程序	233
15.2.5 多协议多服务客户机程序	239
15.3 小结	241
第 16 章 远程过程调用	242
16.1 简介	242
16.2 外部数据表示(XDR)	246
16.2.1 XDR 工作原理	246
16.2.2 初始化 XDR 流	246

16.2.3 释放 XDR 流	246
16.2.4 整数的 XDR 表示	247
16.2.5 无符号整数的 XDR 表示	247
16.2.6 枚举型的 XDR 表示	248
16.2.7 布尔量的 XDR 表示	248
16.2.8 浮点数的 XDR 表示	248
16.2.9 双精度浮点数的 XDR 表示	249
16.2.10 字符的 XDR 表示	249
16.2.11 字符串的 XDR 表示	250
16.2.12 定长数组的 XDR 表示	250
16.2.13 变长数组的 XDR 表示	251
16.2.14 XDR 的例子	252
16.3 小结	255
第 17 章 RPC 编程	256
17.1 RPC 编程简介	256
17.1.1 RPC 程序号、版本号和过程号	256
17.1.2 网络选择	256
17.1.3 rpcbind 设施	258
17.2 RPC 调用	259
17.2.1 clnt_create 函数	259
17.2.2 clnt_call 函数	260
17.2.3 svcudp_create 函数	260
17.2.4 svctcp_create 函数	260
17.2.5 svc_register 函数	261
17.2.6 svc_run 函数	261
17.2.7 svc_sendreply 函数	261
17.3 远程计算器	262
17.3.1 头文件 xdr_math.h	262
17.3.2 客户机程序	263
17.3.3 服务器程序	265
17.3.4 程序的编译和运行	267
17.4 小结	268
第 18 章 用 rpcgen 生成分布式程序	269
18.1 rpcgen 简介	269
18.2 rpcgen 的输入和输出	269
18.3 rpcgen 编程步骤	270
18.3.1 建立 .x 文件	270

18.3.2	运行 <code>rpcgen</code>	270
18.3.3	<code>rpcgen</code> 生成的 <code>math.h</code> 文件	271
18.3.4	<code>rpcgen</code> 生成的 <code>math_xdr.c</code> 文件	272
18.3.5	<code>rpcgen</code> 生成的 <code>math_clnt.c</code> 文件	273
18.3.6	<code>rpcgen</code> 生成的 <code>math_svc.c</code> 文件	274
18.3.7	<code>rpcgen</code> 生成的 <code>math_server.c</code> 文件	276
18.3.8	<code>rpcgen</code> 生成的 <code>math_client.c</code> 文件	277
18.3.9	<code>rpcgen</code> 生成的 <code>makefile.math</code> 文件	280
18.4	小结	281
第 19 章	RPC 认证	282
19.1	简介	282
19.2	取得客户机的认证风格	282
19.3	<code>AUTH_NONE</code> 认证	286
19.4	<code>AUTH_SYS</code> 认证	287
19.4.1	设置 <code>AUTH_SYS</code> 认证	287
19.4.2	<code>AUTH_SYS</code> 认证的例子	287
19.5	<code>AUTH_DES</code> 认证	293
19.5.1	设置 <code>AUTH_DES</code> 认证	293
19.5.2	<code>AUTH_DES</code> 认证的例子	295
19.6	小结	298
第 20 章	GTK 图形界面编程	299
20.1	GTK 简介	299
20.2	第一个 GUI 应用程序	299
20.2.1	初始化	300
20.2.2	建立窗口	300
20.2.3	设置标题	301
20.2.4	设置窗口大小	301
20.2.5	设置边框	301
20.2.6	第一个 GTK 程序	302
20.2.7	程序的编译和执行	302
20.2.8	增加对信号的处理	303
20.3	增加一个按钮	305
20.4	小结	307
第 21 章	按钮与封装	308
21.1	按钮	308
21.1.1	建立按钮	308
21.1.2	按钮信号处理	308

21.1.3 一个按钮的例子.....	309
21.2 双态按钮.....	310
21.3 复选按钮.....	313
21.4 单选按钮.....	313
21.5 封装构件.....	315
21.5.1 封装简介.....	315
21.5.2 用盒子封装构件.....	316
21.5.3 盒子封装的例子.....	316
21.5.4 用表格封装构件.....	318
21.5.5 表格封装的例子.....	320
21.6 图像按钮.....	322
21.7 小结.....	324
第 22 章 GTK 常用构件	325
22.1 标签.....	325
22.1.1 创建标签.....	325
22.1.2 设置标签正文.....	325
22.1.3 得到标签正文.....	325
22.2 编辑框.....	326
22.2.1 创建编辑框.....	326
22.2.2 得到编辑框正文.....	326
22.2.3 改变编辑框正文.....	326
22.2.4 设置编辑框属性.....	326
22.2.5 编辑框的信号.....	327
22.2.6 编辑框的例子.....	327
22.3 列表框.....	330
22.3.1 创建列表框.....	330
22.3.2 操作列表框.....	330
22.3.3 列表框的信号.....	331
22.4 复合框.....	331
22.4.1 创建和操作复合框.....	332
22.4.2 复合框的信号.....	332
22.5 进度条.....	332
22.5.1 建立进度条.....	333
22.5.2 用定时器测试进度条.....	333
22.6 框架.....	336
22.6.1 创建框架.....	336
22.6.2 框架的例子.....	337

22.7 小结	338
第 23 章 状态条、工具条和菜单	340
23.1 状态条	340
23.1.1 创建状态条	340
23.1.2 向状态条添加表项	340
23.1.3 从状态条移走表项	340
23.1.4 状态条的例子	341
23.2 工具条	342
23.2.1 创建工具条	343
23.2.2 向工具条增加按钮	343
23.2.3 向工具条上添加像元图	344
23.2.4 工具条的例子	345
23.3 菜单	346
23.3.1 手工建立菜单	347
23.3.2 用套件建立菜单	351
23.3.3 检查菜单项	355
23.3.4 选择菜单项	357
23.4 小结	359
第 24 章 对话框	361
24.1 初步建立对话框	361
24.2 两种类型的对话框	362
24.2.1 无模式对话框	362
24.2.2 模式对话框	366
24.3 文件选择对话框	369
24.3.1 创建文件选择对话框	369
24.3.2 文件选择对话框的例子	369
24.3.3 结构 GtkFileSelection 的定义	372
24.4 小结	372
第 25 章 GTK 式样	373
25.1 简介	373
25.2 颜色	373
25.2.1 从系统分配颜色	373
25.2.2 创建颜色选择对话框	374
25.2.3 颜色选择对话框的例子	375
25.3 字体	379
25.3.1 创建字体选择对话框	380
25.3.2 字体选择对话框的例子	380

25.4 小结.....	385
第 26 章 高级 GTK 构件	386
26.1 树构件.....	386
26.1.1 树构件的创建.....	386
26.1.2 树构件的信号.....	387
26.1.3 树构件的例子.....	387
26.2 列表构件.....	391
26.2.1 创建列表构件.....	391
26.2.2 向列表构件增加数据.....	392
26.2.3 修改列表构件中的数据.....	392
26.2.4 删除行.....	392
26.2.5 提高插入和删除速度.....	392
26.2.6 行与列参数.....	393
26.2.7 标题栏.....	393
26.2.8 插入像元图.....	394
26.2.9 列表构件的例子.....	394
26.3 笔记本构件.....	397
26.3.1 创建笔记本构件.....	397
26.3.2 操作页.....	397
26.3.3 增加和删除页.....	398
26.3.4 笔记本构件的例子.....	399
26.4 小结.....	402
参考文献	403

第 1 章 Linux 系统管理

1.1 登录与注销

1.1.1 登录 Linux

用户在有了自己的账号以后,便可登录到系统中。登录 Linux 系统有下面两种情况:

(1) 通过 Linux 机器的一个终端登录系统,则只要在该终端键入用户名和口令即可注意,用户键入的口令是不会显示出来的。

(2) 利用 Windows 登录远程 Linux 服务器,则要在 Windows 的【开始】按钮中选择【运行】,然后在运行框中键入 telnet hostname,其中 hostname 为用户要登录的主机的名字或 IP 地址。然后屏幕上就会出现与下面类似的提示:

```
TurboLinux release 6.0 (Kunlun)
Kernel 2.2.13 on i686 (linux)
TTY: tty0
```

Login:

在上面的提示中,第一行为系统说明,在这里为 TurboLinux 6.0;第二行为系统内核说明,这里为 2.2.13;第三行为终端说明,这里为 tty0。这时,用户只要键入用户名和口令即可,但要注意在 Linux 系统中大小写是区别对待的。

如果用户名和口令正确,系统就会出现提示符,表明用户登录成功,如下面所示:

```
Last login: Mon Apr 13 22:42:31 from lq
You have new mail.
[msf@linux]$
```

第一行为上次登录说明,这里表示 4 月 13 日(星期一)22 时 42 分 31 秒从主机 lq 登录;如果你有信,则系统会告诉你,如第二行所示;第三行方括号内部为“用户名@主机名”的形式,这里的用户名为 msf,主机名为 Linux,\$ 是缺省的 B shell 提示符。当然,用户也可以根据自己的喜好,定义自己的提示符。

1.1.2 更改口令

登录完成以后,为了安全起见,用户常常需要更改自己的口令。可以用 passwd 命令更改口令,下面给出了一个更改口令的例子:



```
[msf@linux]$ passwd
Changing password for msf
(current) UNIX password:
New UNIX password:
Retype new UNIX password:
Passwd: all authentication tokens updated successfully
```

用户的口令一般是 6~8 个字母、数字及某些特殊字符(如下划线“_”、“.”等)组成,设置口令时,应选择不易被人发现或破译的组合。例如不宜采用姓名或某个常用词作口令,以免账号被人盗用。口令一定要记牢,万一忘记了,可以请系统管理员帮助,他可以帮助用户去掉口令,或为用户设置新的口令。一般说来,Linux 系统管理员无法查出用户的口令,这一点与其他一些多用户系统不同。

1.1.3 了解 shell

当你登录后,Linux 就将你置于你的起始目录中,并启动一个 shell 进程来与你交互。shell 实际上为操作系统与用户的交互接口,操作系统通过 shell 接受用户的命令。在 DOS 系统中的 shell 为 command.com,在 Windows 系统中为 explorer.exe。Linux 系统的 shell 与上面两种系统有很大的区别,在上面两种系统中,shell 是固定的;而在 Linux 系统中,用户的 shell 是可以选择的,我们可以选择我们爱好的 shell,甚至我们还可以自己写一个 shell。文件/etc/shells 列出了系统中所有的 shell,下面列出了我们系统中的 shell:

```
[msf@linux]$ cat /etc/shells
/bin/sh
/bin/csh
```

在 Linux 系统中,常见的 shell 有两种:B shell 和 C shell。B shell(sh)是最初的 Unix shell,它是由 Steve Bourne 编写的,这个 shell 的可执行文件为/bin/sh;C shell(csh)是由 Bill Joy 开发的,C shell 的可执行文件为/bin/csh。

默认的 Linux shell 为 B shell,要知道你现在所用的是哪种 shell,可以使用命令:echo \$SHELL,它的运行结果如下面所示:

```
[msf@linux]$ echo $SHELL
/bin/sh
```

其实,我们可以利用一种比较简单的方法确定所使用的 shell。如果你的提示符为美元符号(\$),则你使用的为 B shell;如果你的提示符为百分号(%)或#,则你使用的为 C shell,其中%为一般用户的提示符,#为超级用户的提示符。当然,这种方法不总是正确的,因为用户可以改变自己的提示符。

如果我们要改变我们的 shell,则可以使用命令 chsh,下面给出了一个更改 shell 的例子:

```
[msf@linux]$ echo $SHELL
/bin/sh
```



```
[msf@linux]$ chsh
changing shell for msf.
Password:
New shell [/bin/sh]:/bin/csh
shell changed.
[msf@linux]% echo $SHELL
/bin/csh
```

可以看出,当 shell 由 B shell 改为 C shell 时,提示符也由美元符号(\$)变为百分号(%)

1.1.4 了解 shell 环境

登录过程的一部分为建立用户环境,所有 Linux 进程各自都有独立的并且不同于程序本身的环境。Linux 环境,也叫 shell 环境,是由很多变量以及这些变量的值所组成。这些变量和变量值允许一个正在运行的程序(如 shell)来决定这个环境看上去是怎样的。

环境包括你所使用的 shell,你的起始目录以及你所使用的终端类型等一系列信息。这些变量中有一些变量在登录过程中被定义,并且要么不能改变,要么不应该改变;而另外一些变量是能够被改变的。

在环境中,以“VARIABLE = value”的形式来设置变量。其中 VARIABLE 的含义可以被设置为你喜欢的任何内容。但是,对许多标准 Linux 程序而言,许多变量有预先定义的含义。例如,TERM 变量被定义为你的终端名字,DEC 公司花了几年的时间制作了一个名为 vt-100 的流行终端。这种终端的特性已被许多厂家复制,并且经常用于被个人计算机的软件仿真。这种终端的名字为 vt-100,在环境中被表示为 TERM = vt100。

环境中许多预定义了许多变量,表 1.1 列出了 B shell 的通用环境变量。

表 1.1 B shell 的通用环境变量

变 量	描 述
HOME = /home/login	HOME 用于设置你的起始目录,起始目录为你开始工作的位置。如果你的 ID 为 msf,则 HOME 被定义为/home/msf
LOGNAME = login	LOGNAME 和你的登录 ID 一样是被自动设置的
PATH = path	Path 为 shell 查找命令的目录列表。例如,你可以设置路径如下:PATH = /bin:/usr/bin
PS1 = prompt	PS1 为 shell 提示符。你可以设置 PS1 = “Enter Command”,那么 Enter Command > 将会成为你的命令提示符
PWD = directory	PWD 是被自动设置的。它定义你现在的在文件系统中的位置
SHELL = shell	SHELL 定义你的 shell 程序的位置
TERM = termtype	你登录终端的名字

除了表中列出的通用环境变量以外,系统还有很多其他的环境变量。如果你使用的是 C shell,那么可用 printenv 命令列出这些变量;如果你使用 B shell,则可以使用 env 命令。下面给出了在 B shell 中,env 命令的运行情况:

```
[msf@linux]$ env
```



```
PWD = /home/msf
HOSTNAME = linux.tsinghua.edu.cn
PS1 = [ \u@ \h ] \ $
USER = msf
MAIL = /usr/spool/mail/msf
LOGNAME = msf
SHELL = /bin/sh
TERM = vt100
HOME = /home/msf
PATH = ./:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
```

在上面所显示的环境变量中,只有 PS1 比较难懂,现在我们就介绍一下。PS1 的表达式中, \u 代表用户名, \h 代表登录的主机名, \ \$ 为系统默认提示符,这里就是美元符号 (\$)。例如,如果以用户名 msf 登录主机 linux,则系统的提示符为[msf@linux] \$。

1.1.5 配置 shell 环境

在 DOS 启动时,DOS 系统执行 autoexec.bat 批处理文件来设置 PATH、PROMPT 等属性;在 Linux 系统中,根据所启动的 shell 的不同,Linux 执行不同的文件来设置相应的属性。由于 Linux 是多用户的系统,每个用户都可以建立与其他用户不同的运行环境,因此这些文件放在每一用户自己的注册目录中,在 B shell 进程启动时,系统执行用户注册目录中的 .profile 文件,以设置 shell 变量的值;如果用户注册启动 C shell 进程,系统会执行用户注册目录中的 .cshrc 和 .login 文件,用户可将期望的有关设置写在这两个文件中。

由于 Linux 的默认 shell 为 B shell,因此,我们在下面给出一个 B shell 的配置文件。

```
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
alias ..='cd ..'
alias ll='ls -l'
alias la='ls -aF'
PATH=$PATH:./
export PATH
```

对于默认的 PATH 环境变量,一般不包含当前目录。这对于我们的编程工作非常不方便。因此,我们常常得在 .profile 中添加当前目录(./)到 PATH 环境变量中去。在上面的例子中,我们还利用 alias 指令为一些 shell 命令起了别名,如将 rm 设置为 rm -i 的别名可以在用户删除文件时要求确认,这样可以防止用户的误删除操作。

1.1.6 注销

在用户完成所做的工作后,要离开 Linux 系统时需要进行注销。注销工作是必要的,因



例:

```
[root@linux msf] # du /bin
4991 /bin
[root@linux msf] # du /dev
1   /dev/inet
4   /dev/ida
1   /dev/rd
49  /dev
[root@linux msf] # du -s /home
349705 /home
```

第一行命令显示当前目录中所有子目录所占的盘块数,最后显示当前命令所占的盘块总数(缺省不显示普通文件所占的盘块数)。

第二行命令统计/home目录所占的总盘块数,不显示其子目录的统计结果

1.2.4 file 命令

命令 file 判断文件类型。

格式:file [filename]

例:

```
[root@linux msf] # file /usr/bin/passwd
/usr/bin/passwd;setuid ELF 32-bit LSB executable,
Intel 80386,version 1,dynamically linked (uses shared libs),
not stripped
[root@linux msf] # file /etc/passwd
/etc/passwd:ASCII text
```

第一行命令显示/usr/bin/passwd的文件类型,在这里为32位可执行程序。

第二行命令显示/etc/passwd的文件类型,在这里为ASCII文本文件。

1.2.5 hostname 命令

命令 hostname 设置或显示主机名。

格式:hostname

例:

```
[root@linux msf] # hostname
linux.tsinghua.edu.cn
[root@linux msf] # hostname msf
[root@linux msf] # hostname
msf
```



第一行命令显示主机名,在这里为 linux.tsinghua.edu.cn。

第二行命令更改主机名为 msf。

第三行命令再次显示主机名,可以看出,主机名已经被改成 msf。

1.2.6 id 命令

命令 id 显示用户标识。

格式: id [-a] [user]

选项:

-a 报告用户名、用户 id 及用户所属的所有的组

例:

```
[root@linux msf] # id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),
3(sys),4(adm),6(disks),100(users)
```

该命令显示当前用户为 root,其 uid 是 0,该用户同时属于 7 个组,组名分别为 root(组 ID 为 0)、bin(组 ID 为 1)、daemon(组 ID 为 2)、sys(组 ID 为 3)、adm(组 ID 为 4)、disks(组 ID 为 6)和 users(组 ID 为 100)。在 Linux 中,一个用户是可以同时属于多个组的。

命令 id + user 显示指定用户的 uid、所在组的组名及组的 gid。

1.2.7 ps 命令

命令 ps 显示当前进程状态。

格式: ps [options] [namelist]

选项:

-c 显示当前运行的每一进程的信息。

-f 产生一个完整清单。

-l 产生一个长清单。

例:

```
[root@linux msf] # ps
PID  TTY    TIME    CMD
650  tty0   00:00:00  login
664  tty0   00:00:00  bash
1109 tty0   00:00:00  ps
```

命令 ps 显示与控制终端有关的进程(即用户本人的进程)的信息。内容包括进程 ID (PID)、控制进程的终端(TTY)及命令名(COMMAND)其中,TTY 的值可能为:

co,表示主终端。

a,表示接在串口的 a 口上的终端。

ttyn,表示通过窗口或网络系统使用的虚拟终端。

命令 ps - aux 用来显示系统中所有进程的完整清单。



1.2.8 quota 命令

命令 `quota` 显示用户的磁盘空间限额及使用情况(BSD 命令)。

格式: `quota [-v] [username]`

选项:

-v 显示本用户的磁盘空间限额及使用情况。

例:

```
[root@linux msf]# quota msf
Disk quotas for user msf (uid 501): none
[root@linux msf]# quota -v msf
Disk quotas for user msf (uid 501):
Filesystem    blocks quota limit
Linux:(pid442) 0    0    0
```

第一行命令显示用户 `msf` 的磁盘空间限制情况,在这里为没有限制。

第二行命令显示用户 `msf` 的磁盘空间限额及使用情况。显示的信息包括:用户在施加限额的文件系统中的磁盘块限额、已使用的盘块数、允许建立的 `i` 节点数限额。

1.2.9 stty 命令

命令 `stty` 用来设置或改变终端的任选项。

格式: `stty [options]`

选项:

-a 报告所有任选项的设置情况。

-g 报告任选项的当前设置。

例:

```
[root@linux msf]# stty
speed 9600 baud; line = 0;
- brkint - imaxbel
[root@linux msf]# stty -a
speed 9600 baud; row 0 ; columns 0; line = 0;
intr = ^c; quit = ^\ ; erase = ^?; kill = ^U; eof = ^D; eol = <undef> ;
eo12 = <undef> ; start = ^Q; stop = ^S; susp = ^Z; rpmt = ^R;
werase = ^W; lnext = ^V; flush = ^o; min = 1; time = 0;
- parenb - parodd cs8 - hupcl - cstopb cread - clocal - crtscts
- ignbrk - brkint - ignpar - parmrk - inpck - istrip - inlcr - igncr icml
ixon - ixoff - iuclc - ixany - imaxbel opost - olcuc - ocml onlcr - onocr
- onlret - ofill - ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok - echonl - noflsh - xcase
```




```
7:33pm  up 28 min, 1user, load average:0.99,0.95,0.84
USER  TTY  FROM  LOGIN@  IDLE   JCPU   PCPU   WHAT
msf  tty0  lq    7:14pm  0.00s  0.39s  0.11s  w
[root@linux msf]# w -s
7:33pm  up 28 min, 1user, load average:0.99,0.96,0.84
USER    TTY  FROM  IDLE   WHAT
msf     tty0  lq    0.00s  w -s
```

命令 `w` 显示当前正在使用系统的用户的用户名、终端号、注册地点、注册时间及当前执行的命令。命令 `w -s` 按短格式显示。

1.2.13 whereis 命令

命令 `whereis` 确定一个命令的 2 进制执行文件、源码及联机帮助文档所在的位置。

格式: `whereis command`

例:

```
[root@linux msf]# whereis sh
sh:/bin/sh /usr/man/man1/sh.1
```

这个例子告诉我们 `sh` 的可执行文件在 `/bin/sh`, 联机帮助文档在 `/usr/man/man1/sh.1`。

1.2.14 who 命令

命令 `who` 列出正在使用系统的用户。

格式 `who`

```
whoami
who am i
```

例:

```
[root@linux msf]# who
cbk tty1 Apr 14 19:36
msf tty0 Apr 14 19:14(lq)
[root@linux msf]# who am i
msf! msf tty0 Apr 14 19:14(lq)
[root@linux msf]# whoami
root
```

命令 `who` 显示当前正在使用系统的所有用户的用户名(name)、用户所使用的终端(line)、用户注册时间(time)。如果用户从远程注册,还要显示远程用户的主机名。

命令 `who am i` 显示目前正在使用本终端(或窗口)的用户在注册时使用的用户名。而命令 `whoami` 显示目前正在使用本终端(或窗口)的用户名。



1.3 Linux 装载程序 LILO

LILO 是一个引导管理程序,可以在引导时使用主引导记录在一组不同的操作系统中做选取。

1.3.1 配置 LILO

LILO 读取配置文件/etc/lilo.conf,并用它找出系统上装了什么操作系统及它们的引导信息放在何处。/etc/lilo.conf 文件以一些说明 LILO 如何操作的一般信息开始,然后包含了数小节,列出了 LILO 可引导的每个操作系统的引导信息。每个操作系统占一个小节。下面是 LILO 配置文件中的两个小节:

```
boot = /dev/hda
map = /boot/map
install = /boot/boot.b
prompt
timeout = 50
default = linux
other = /dev/hda1
    label = win
    table = /dev/hda
image = /boot/vmlinuz
    label = linux
    root = /dev/hda3
    read-only
```

1.3.2 使用 LILO

在安装 LILO 时,通常要设置默认的超时值和默认引导的操作系统。这使你在引导时有一定的时间来选取另一个操作系统。如果 LILO 在超时计数结束时,你仍没有选取操作系统,则引导默认设置的操作系统。

使用 LILO 引导计算机时,有一个 LILO: 提示。这时你可以按 < Ctrl >、< Alt > 或 < Shift >, 让 LILO 立即引导默认操作系统;也可以键入一个操作系统的名字让 LILO 引导指定的操作系统;还可以按 < Tab > 键让 LILO 显示一个可用的不同操作系统的列表,你甚至可以等着让 LILO 引导默认操作系统。

1.4 管 理 用 户

管理 Linux 系统的一个重要方面是添加和删除用户账号,即能够使别的用户登录到系



统上、设置他们的权限、为用户创建和指定起始目录、把用户分到组中及在必要时删除用户。每个用户都应有一个唯一的登录名及一个口令。

1.4.1 添加用户

添加用户时,只要在口令文件/etc/passwd中为这个用户添加一个条目,这种条目的格式是:

name:password:UID:GID:information:directory:shell

即:登录名;加密口令;用户ID号;组ID号;用户信息;登录目录;登录shell。表1.2给出了这些字段的详尽解释。

表 1.2 添加用户的条目中字段的详尽解释

字 段	描 述
name	用于登录的名称
password	证明用户所需的口令
UID	操作系统用来辨别用户的唯一号码
GID	用来辨别用户的主要的组的唯一号码或名字
information	对用户的描述
directory	用户的起始目录(该用户登录后所处的位置)
shell	用户登录时所使用的 shell

1.4.2 使用 adduser 命令

使用 adduser 命令并提供你想要添加的用户的名字,就可以添加用户了。下面给出了 adduser 命令的用法:

```
root@linux msf] # /usr/sbin/adduser
usage: adduser [-u uid [-o]] [-g group] [-G group,...]
              [-d home] [-s shell] [-c comment] [-m [-k template]]
              [-f inactive] [-e expire mm/dd/yy] [-p passwd] [-n] [-r] name
adduser -D [-g group] [-b base] [-s shell] [-f inactive]
          [-e expire mm/dd/yy]
```

关于 adduser 的各个选项的含义,我们就不详细介绍了。/etc/skel 目录应该包括你想让每个用户都拥有的模板文件,通常有“个人”配置文件,如用于 shell 配置的 .profile、.cshrc 和 .login 文件,用于设置电子邮件的 .mailrc 文件等。

1.4.3 设置用户口令

使用 passwd 命令设置用户口令时应当为每一位加入到系统中的用户设置口令,用户可以在以后登录时改变自己的口令。passwd 的使用方法如下所示:



```
root@linux msf] # passwd oracle
New UNIX passwd:
Retype new UNIX password:
Passwd: all authentication tokens updated successfully
```

在本例中,我们为用户 oracle 设置了口令。选取口令时最好慎重一些,口令尽量包括大写和小写字母、标点符号和数字,至少有六位数字长。

1.4.4 删除用户

删除用户一般有三种不同的方式,它们的程度各不相同。

(1) 删除用户的登录能力。编辑口令文件(/etc/passwd)并把一个 * 号放在这个用户的条目的第二个字段中,如下所示:

```
msf: *:471:32:TurboLinux User:/home/msf:/bin/sh
```

但这个用户的目录文件和组信息保持完好无损。

(2) 从口令文件中删除用户,但在系统上保留该用户的文件。这样其他用户仍可使用被删除掉的用户文件,还可让一个新用户代替那个老用户的职责。你只要使用一个编辑程序或使用 `userdel login_name` 命令,然后再使用 `chown` 命令和 `mv` 命令来改变被删除用户的文件的所有权和存放位置。

(3) 删除这个用户及这个用户所拥有的所有文件。你只要从口令文件删除这个用户的条目,并从系统删除该用户的文件。这样就完全彻底地删除掉了该用户。可以使用如下 `find` 命令来进行:

```
find home - directory -exec rm -rf \;
```

然后用 `rmdir home - directory` 命令来删除这个用户的起始目录,并从口令文件中删除用户的记录项。

1.5 管理用户组

用户是组的成员,不同的组可以被赋予不同的能力或权限。这样能够使不同的文件集被不同的用户合理地利用。

单个用户的信息包括在口令文件中,组信息则保存在 `/etc/group` 文件中。下面是这个文件中的一个记录项的例子:

```
user: :42:msf,cbk,oracle
```

这个组的组名是 `user`,组 ID 号是 42,组成员有 `msf`、`cbk` 和 `oracle`。

1.5.1 添加用户组

可以通过直接编辑 `/etc/group` 文件,在其中输入新用户组的信息来创建一个新的用户



组。

/etc/group 文件中的每一个组拥有一个与之相关的、唯一的组 ID 号。注意,如果两个组分配了同一个 ID 号,那么这两个组实际上便是同一个组。

1.5.2 删除用户组

同样通过编辑/etc/group 文件并删除其中你想删除的用户组的条目来删除一个组。同时应当把具有这个组 ID 的所有文件重新分配给不同的组。利用 find 命令很容易做到这一点,如下所示:

```
find / -gid group -id home -directory -exec chgrp newgroup {} \;
```

1.6 管理文件系统

文件系统的管理在 Linux 系统中是很重要的,因为用户的所有数据和程序都处在文件系统中。所以用户应当学会管理文件系统以便能创建、管理和维护系统。

1.6.1 安装文件系统

Linux 可以使用两种方法安装文件系统:动态地和静态地。

Linux 使用 mount 命令动态地安装文件系统 mount 命令的基本语法是:

```
mount device mountpoint
```

device 是要安装的实际设备; mountpoint 是文件系统树中安装这个文件系统的安装点。表 1.3 列出了 mount 命令行的一些参数。

表 1.3 mount 命令的命令行参数

参 数	描 述
-f	除了不进行实际的安装系统调用外,它完成每个操作。这个参数“假”安装文件系统
-v	为 mount 的工作提供附加信息,长格式模式
-w	安装有读和写权限的文件系统
-r	安装只有读权限的文件系统
-n	安装时不把条目写入/etc/mstab 文件中
-t type	指定要安装的文件系统的类型
-a	使 mount 试图安装/etc/fstab 中的所有文件系统
-o list_of_options	当带有有用逗号分开的选项列表时,将使 mount 把指定的选项用于正被安装的文件系统上

下面给出了 mount 命令的两个典型用法:第一个为安装光驱,文件系统类型为 iso9660,设备文件名为/dev/cdrom,安装的目录为/cdrom;第二个为安装软盘,文件类型为 msdos,设备文件为/dev/fd0,安装目录为/floppy。



```
root@linux msf: # mount -t iso9660 /dev/cdrom /cdrom
mount: block device /dev/cdrom is write-protected, mounting read-only
root@linux msf: # mount -t msdos -t conv=auto /dev/fd0 /floppy
```

一般情况下可指定在 Linux 引导时要安装的文件系统清单和关闭时要卸下的文件系统清单,此谓静态安装。这些文件系统被列在文件系统表/etc/fstab 中。表 1.4 列出了/etc/fstab 文件中的不同字段。

表 1.4 /etc/fstab 文件中的字段

字 段	描 述
File system specifier	指定要安装的块特殊设备或远程文件系统
Mount point	指定文件系统的安装点。对特殊文件系统使用 none,none 激活交换文件,但即使交换文件在文件树中看不到
Type	给出指定文件系统的文件系统类型
Mount Option	用逗号分隔的文件系统安装选项列表,必须包含文件系统的安装类型
Dump Frequency	指定应该多长时间使用 dump 命令备份文件系统一次,没有这个字段的话,则认为文件系统不需要备份
Pass Number	指定在引导系统时,fsck 命令按什么顺序检查文件系统。没有指定数值的话,则引导时不会检查一致性

下面是一个 fstab 文件的例子:

```
/dev/hda3    /          ext2        defaults
/dev/hda1    /win98     vfat        defaults
/dev/hda5    /winnt     vfat        defaults
/dev/hda6    /oracle    ext2        defaults
/dev/hda4    swap       swap        defaults
/dev/fd0     /mnt/floppy ext2        noauto
/dev/cdrom   /mnt/cdrom iso9660     noauto,ro
none        /proc      proc        defaults
```

这个文件中的每一行由四列组成,第一列为设备文件名,第二列为安装目录,第三列为文件系统类型,第四行显示是否在启动时自动安装,如为 defaults,则启动时自动安装,否则不自动安装。

1.6.2 卸下文件系统

使用 umount 命令可以卸下文件系统,umount 命令有三种基本形式:

```
umount device | mountpoint
umount -a
umount -t fstype
```

device 是要卸下的实际设备的名字,mountpoint 是安装点目录名(只需指定这两者之一



即可)。-a 和 -t fstype 是 umount 命令仅有的两个命令行参数：-a 卸下所有的文件系统，-t fstype 只卸下指定类型的文件系统。

1.6.3 维护文件系统

通常我们使用 fsck 命令来检查文件系统，fsck 命令的用法是：

```
fsck [-A] [-V] [-t fs-type] [-a] [-l] [-r] [-s] filesys
```

表 1.5 描述了 fsck 命令的选项。

表 1.5 fsck 的命令选项

参 数	描 述
-A	遍历/etc/fstab 文件，并试图一次检查所有的文件系统。如果使用了 -A，就不能再使用 filesys 参数
-V	长格式模式。打印有关 fsck 正在干什么的附加信息
-t fs-type	指定要检查的文件系统的类型
filesys	指定要检查哪个文件系统。这个参数可以是一个块特殊设备名或一个安装点
-a	在不询问任何问题的情况下，自动地修复在文件系统中发现的任何问题
-l	列出文件系统中的所有文件名
-r	在修复文件系统前请求确认
-s	在检查文件系统前列出管理块的信息

在检查一个文件系统前最好卸下它，因为这样保证了在检查这个文件系统上的文件时，没有任何文件在使用。但你在检查根文件系统时，不能直接卸下根文件系统，因为 Linux 要运行必须访问它。这时你只要从一张带有根文件系统的维护软盘进行引导，然后通过指定根文件系统的特殊设备名，从软盘上运行真正的根文件系统上的 fsck 命令。如果你的文件系统被 fsck 做了修改，最好立即重新引导系统。因为这使 Linux 能重新读取文件系统的重要信息，防止文件系统受到进一步的损坏。

如果/etc/fstab 文件的 pass number 字段的值设为大于 0，则 Linux 在引导时会自动检查文件系统。

1.7 配置 TCP/IP 网络

在对 Linux 机器的管理中，TCP/IP 的配置是你将要面对的较常见的任务之一。

1.7.1 了解 TCP/IP 配置文件

Linux 中的 TCP/IP 网络由/etc 目录中的一组配置文件控制，这些文件告诉 Linux 它的 IP 地址、主机名和域名是什么，并且还控制网络接口。表 1.6 列出了每个文件及其作用，后面将详细讨论这些文件。



表 1.6 Linux TCP/IP 网络配置文件

文 件	作 用
/etc/hosts	把主机名映射成地址
/etc/networks	把域名映射成网络地址
/etc/rc.d/rc3.d/S10network	在引导时配置和激活以太网接口

1.7.2 主机配置文件

配置文件/etc/hosts 是将主机名映射为 IP 地址的最原始的方法。为了便于说明,我们来看下面这个例子:

```
127.0.0.1      localhost localhost.localdomain
192.168.0.41   linux.tsinghua.edu.cn  linux
192.168.0.40   lq.tsinghua.edu.cn    lq
```

该文件的格式由每行开始的第一列上的 IP 地址、这个地址的正规主机和零个或多个别名组成,空行和后面带 # 字符的文本被认为是注释并被忽略。

IP 地址 127.0.0.1 称为本地回送地址,并为此而保留,通常把名字 localhost 分配给它。

1.7.3 初始化以太网接口

ifconfig 程序使 Linux 内核知道软件回送和以太网卡等网络接口,以便 Linux 调用,ifconfig 程序也被用于监控和改变网络接口的状态。下面是一个简单的例子:

```
ifconfig interface address
```

该命令激活指定的网络接口并给它分配一个 IP 地址。这称之为激活一个接口。ifconfig 的用法是:

```
ifconfig interface [atype] [options] | address
```

表 1.7 列出了 ifconfig 的命令行参数(选项),通常这些参数并不需全部使用。ifconfig 可以仅由接口名、网络掩码和分配的 IP 地址来设置所需的一切;当你有一个复杂的网络时,你只需显式地设置大多数参数。

表 1.7 ifconfig 命令行参数

参 数	描 述
Interface	网络接口名,通常是后跟一个标识好的设备驱动程序名。这个参数是必需的,如 eth0 等
Aftype	地址集,他们被用于解码和显示所有协议的地址。目前已支持 inet(TCP/IP)、ddp(Appletalk)、ipx(Novell)以及 AX.25 和 netrom 地址集。inet 集是默认值
Up	激活指定的接口
Down	使指定的接口不活动
[-arp]	打开或关闭在指定接口上使用的 ARP 协议、负号用于关闭该选项



(续)

参 数	描 述
[-]trailers	打开或关闭以太网帧上的跟踪器。目前还未在网络系统中实现
[-]allmulti	打开或关闭接口的无区别模式,打开这个模式让接口把网络上的所有信息流都送给内核,而不仅仅是把发给你的机器的信息发送给内核
Metric N	把接口度量值设置为整数值。度量值表示在这个路径上发送一个分组的“成本”
Mtu N	将接口在一次传输中可以处理的最大字节数设置为整数值 N。目前内核中的网络代码不处理分段,因此一定要把最大传输单元值设置得足够大
Dstaddr addr	设置点对点联络的另一段的 IP 地址。它已被 pointopoint 关键字所替代
Netmask addr	为接口设置网络掩码
[-]broadcast {addr}	当使用了一个地址时,设置这个接口的广播地址。如果没有给出地址,就打开这个指定接口的 IFF_BROADCAST 选项,前导负号表示关闭这个选项
[-]_pointopoint [addr]	打开指定接口的点对点模式,它告诉内核该接口是对另一台机器的直接链接,当包含了一个地址时,这个地址被分配给列表另一端的机器。如果没有给出地址,就打开这个接口的 IFF_POINTOPOINT 选项,前导负号表示关闭这个选项
Hw	为指定接口设置硬件地址。硬件类型名和与此硬件地址对等的 ASCII 字符必须跟在这个关键字后面。目前支持以太网(ether)、AMPR、AX.25(ax25)和 PPP(ppp)
address	分配给指定接口的主机名或 IP 地址。这里使用的主机名被解析成它们的对等 IP 地址,这个参数是必需的

1.7.4 用 ifconfig 检查网络接口

不带参数运行可使它输出内核知道的所有网络接口的状态。下面是 ifconfig 的运行结果。

```
root@linux msf] # /sbin/ifconfig
eth0 Link encap:Ethernet Hwaddr 00:60:97:0D:A0:3B
    inet addr:192.168.0.41 Bcast:192.168.0.255 Mask:255.255.255.0
    UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
    RX packets:10960 errors:11 dropped:0 overruns:0 frame:20
    TX packets:2424 errors:0 dropped:0 overruns:0 carrier:2
    Collisions:27 txqueuelen:100
    Interrupt:5 Base address:0xee80
Lo Link encap:Local Loopback
    inet addr:127.0.0.1 Mask:255.0.0.0
    UP LOOPBACK RUNNING MTU:3924 Metric:1
    RX packets:9326 errors:0 dropped:0 overruns:0 frame:0
    TX packets:9326 errors:0 dropped:0 overruns:0 frame:0
    Collisions:0 txqueuelen:0
```

在这个例子中,输出了两个接口的信息。第一个接口为 eth0,是一个以太网接口,硬件地址为 00:60:97:0D:A0:3B,以太网的硬件地址是一个 48 位的数。该以太网接口的 IP 地址为 192.168.0.41,广播地址为 192.168.0.255,网络掩码为 255.255.255.0,这个接口是



激活的(U_p),它的 Mtu 为 1500 字节。第二个接口为 Lo,即软件回送接口,可以看到分配的 IP 地址为 127.0.0.1,网络掩码为 255.0.0.0,这个接口也是激活的(U_p),它的 Mtu 为 3924 字节, Metric 为 1。在这两个接口信息的最后两行给出了有关接受的 RX 和传送的分组数,以及分组出错数、丢掉的分组数和超限数的统计数字。

1.7.5 配置软件回送接口

所有在内核中安装了网络层的 Linux 机器都有一个软件回送接口。这个接口用于测试网络应用程序,并在机器没有连接到真正的网络上时,为本地的 TCP/IP 服务提供一个网络。回送系统的网络接口名为 Lo,键入下面命令以运行 ifconfig:

```
ifconfig lo 127.0.0.1
```

此命令激活回送接口并给它分配地址 127.0.0.1,这是传统上用于回送的地址。

1.7.6 配置网络接口

配置一个以太网接口稍复杂一点,特别是在使用子网的情况下。对 ifconfig 的基本调用就像下面对 linux.tsinghua.edu.cn 所使用的那样:

```
ifconfig eth0 linux
```

这使 ifconfig 激活以太网接口 0 及在/etc/hosts 文件中查找 Linux 的 IP 地址,并将该地址分配给这个接口此时测试以太网接口。下面给出了结果:

```
root@linux msf] # /sbin/ifconfig eth0
eth0  Link encap:Ethernet Hwaddr 00:60:97:0D:A0:3B
       inet addr:192.168.0.41 Bcast:192.168.0.255 Mask:255.255.255.0
       UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
       RX packets:10960 errors:11 dropped:0 overruns:0 frame:20
       TX packets:2424 errors:0 dropped:0 overruns:0 carrier:2
       Collisions:27 txqueuelen:100
       Interrupt:5 Base address:0xee00
```

注意广播地址和网络掩码是由 ifconfig 根据/etc/hosts 中的 IP 地址自动设置的。如果你在使用子网,只要显式地指定广播地址和网络掩码。如下所示:

```
ifconfig eth0 linux broadcast 192.168.0.255 netmask 255.255.255.0
```

1.8 小 结

在本章中,我们主要讲解了 Linux 系统管理的一些知识,包括 Linux 的登录注销,常用的一些 shell 命令,用户管理以及 TCP/IP 网络的配置。在下一章中,我们将着重介绍 Linux 网络服务器的配置。

第 2 章 网络服务器的配置

Linux 系统今天能够被如此普遍地重视,很大程度上是由于 Linux 系统强大的网络功能。拥有 Linux,我们可以把我们的 PC 机改造成功能强大的网络服务器。在这一章中,我们将重点讲解 Linux 网络服务器的安装与配置。

2.1 配置 FTP 服务器

2.1.1 FTP 服务器简介

网络应用中,FTP(File Transfer Protocol,文件传输协议)有着非常重要的地位。Internet 上的文件传输大都通过 FTP,无论是 Internet 上的用户还是要将程序发送给 Internet 上用户的厂商或个人,都可以通过 FTP 来实现文件传输的功能。Internet 上的资源非常丰富,各种各样的软件几乎都可以找到,而这些软件大都放在 FTP 服务器上。

2.1.2 安装 FTP 服务器

FTP 服务器通常根据服务的对象可以分成两种:一种是类似 Unix 系统的基本 FTP 服务器,它只允许系统上的合法用户使用,这种 FTP 服务器在 Linux 安装好之后就已经装上了,不需要另外安装;另一种是匿名的 FTP 服务器(Anonymous FTP Server),是可以让任何人登录上去获取文件的 FTP 服务器。我们通常所说的 FTP 服务器指的就是后者。在这一小节中,我们重点介绍如何安装和配置匿名 FTP 服务器。

我们一般使用的匿名 FTP 服务器是 `wu-ftp`,除支持许多 Unix 系统外,`wu-ftp` 还有很强大的功能,所以非常受使用者的欢迎,它提供以下的功能:

- 让用户在下载文件的同时对文件做自动的压缩或解压缩操作;
- 可以对不同网络的机器分配不同的存取限制和存取时间;
- 可以记录文件上载和下载的时间;
- 可以显示传输时的相关信息,以便让用户知道目前的传输状态;
- 可以设定连接的数量限制,以提高工作效率。

这些功能都适于吞吐量较大的 FTP 服务器的管理要求。

下面我们就向读者介绍如何安装 `wu-ftp` 服务器,假设我们使用的服务器软件为 `wu-ftp-2.6.0.tar.gz` 这个版本。如果读者的机器上已经安装了 `wu-ftp` 服务器,则可以跳过这部分,直接看下一小节的如何配置 FTP 服务器。

首先,我们得把这个软件解压缩,如下所示:



```
tar zxvf wu-ftp-2.6.0.tar.gz
```

文件解压缩后,我们就会得到 `wu-ftp-2.6.0/` 这个目录,我们进入这个目录后,首先要查看 `README` 和 `INSTALL` 这两个文件,以便对安装有一个大致的了解。

接着,我们要依次执行 `configure`、`make` 和 `build install` 这三个命令。如果没有错误,则会生成 `bin` 目录,该目录中包含了服务器的几个可执行文件。文件名及其用途如表 2.1 所示。

表 2.1 ftp 实用程序文件名及其用途

文 件 名	用 途
<code>ftpd</code>	FTP 服务器程序
<code>ckconfig</code>	检查 FTP 的设置
<code>ftpstart</code>	重新启动 FTP 服务器
<code>ftpwho</code>	查看 FTP 服务器的用户
<code>ftpcount</code>	查看 FTP 服务器的上线用户数量
<code>ftpsht</code>	关闭 FTP 服务器程序

2.1.3 查看 FTP 服务器的设置

查看服务器的设置是配置服务器的第一步,这样我们可以得到服务器配置文件的位置。我们执行 `ckconfig` 命令,如下所示:

```
[root@linux bin] # ckconfig
Checking _PATH_FTPUSERS : /etc/ftpusers
I can't find it . . . look in doc/examples for an example.
Checking _PATH_FTPSERVERS : /etc/ftpservers
I can't find it . . . look in doc/examples for an example.
Checking _PATH_FTPACCESS : /etc/ftpaccess
I can't find it . . . look in doc/examples for an example.
Checking _PATH_PIDNAMES : /var/run/ftp.pids - %s
Ok
Checking _PATH_FTPCVT : /etc/ftpconversions
I can't find it . . . look in doc/examples for an example.
Checking _PATH_XFERLOG : /var/log/xferlog
Ok
Checking _PATH_PRIVATE : /etc/ftpgroups
I can't find it . . . look in doc/examples for an example.
You only need this if you want SITE GROUP and SITE GPASS
Functionality. If you do, you will need to edit the example.
Checking _PATH_FTPHOSTS : /etc/ftphosts
```



```

shutdown /etc/shutmsg
delete      no      guest,anonymous      # delete permission?
overwrite   no      guest,anonymous      # overwrite permission?
rename      no      guest,anonymous      # rename permission?
chmod       no      anonymous          # chmod permission?
umask       no      anonymous          # umask permission?
upload /var/ftp *      no      nobody      nogroup 0000 nodirs
upload /var/ftp /bin    no
upload /var/ftp /etc    no
upload /var/ftp /incoming yes    root      daemon 0600 dirs
alias inc: /incoming
cdpeth /incoming
cdpeth /pub
cdpeth /
path-filter anonymous /etc/pathmsg '[ -A-Za-z0-9_\. ] * $^\. ^-
peth-filter guest /etc/pethmsg '[ -A-Za-z0-9_\. ] * $^\. ^-
guestgroup ftponly
email user@hostname

```

下面逐一介绍该文件中涉及的命令:

- **loginfails** 次数

功能: 设定当有用户登录到 FTP 服务器时, 允许用户有几次密码输入错误的机会, 如果超过此数目就会断线。

示例: loginfails 2

含义: 密码输入错两次就断线。

- **class** 类名 类别(real、guest、anonymous) IP 地址

功能: 这个指令的功能是设定 FTP 服务器上用户的类别。在 FTP 服务器上的用户基本上可以分为以下三类:

(1) real 在该 FTP 服务器上属于有合法账号的用户。

(2) guest 另外定义某些使用群组的用户。

(3) anonymous 权限最低的匿名用户。

示例 1: class remote real,guest,anonymous *

含义: 定义一个名为 remote 的类, 里面有三种类型的用户, “*”代表网上的所有机器, 也就是说, 任何人都可以连到你这台 FTP 服务器上。

示例 2: class local real,guest,anonymous *.tsing.edu.cn 166.111.

含义: 将 tsinghua.edu.cn 网段中的机器定义为类名 local, 它们连到 FTP 服务器时有特别的权限。

- **limit** 类别 人数 时间 文件名



- **tar** 选项(yes/no) 类别
功能: 设定哪一个类别的用户可以使用 tar 功能。
示例: `tar yes local remote`
- **private** 选项(yes/no)
功能: 设定是否支持群组对文件的取用。
示例: `private yes`
- **passwd - check** 选项(none/trivial/rfc822) 选项(enforce/warn)
功能: 设定当用户使用 anonymous 账号登录时的密码使用方式, 下面是各选项的说明:
 - (1) None 表示不做密码验证, 任何密码都可以登录。
 - (2) Trivial 表示只要输入的密码中含有字符“@”就可以登录。
 - (3) Rfc822 表示密码一定要符合 Rfc822 中所规定的 E-mail 地址格式才能登录, 如 `nsf@263.net`。
 - (4) Enforce 表示输入的密码不符合指定格式时就无法登录。
 - (5) Warn 表示密码输入不正确时只会出现警告信息, 但仍然可以登录。示例: `passwd - check rfc822 warn`
含义: 匿名用户的密码必须符合 E-mail 地址格式时才可以进入, 密码输入不正确时只会出现警告信息。
- **log commands** 类别(real/guest/anonymous)
功能: 设定哪些用户登录时, 所使用的操作会被记录在文件 `/usr/adm/xferlog` 中。
示例: `log commands real`
含义: 已经在系统中登记的用户(real)登入系统时, 登入时的操作会记录在文件 `/usr/adm/xferlog` 中。
- **log transfers** 类别(real/guest/anonymous) 选项(inbound/outbound)
功能: 设定所指定的用户群组在上载(inbound)或是下载(outbound)时的有关信息会记录在文件 `/etc/xferlog` 中。
示例: `log transfers anonymous, real inbound, outbound`
含义: 当 anonymous 或 real 这两类用户登录后, 他们上载和下载的操作将会被记录在文件 `/etc/xferlog` 中。
- **shutdown** 文件名
功能: FTP 服务器关闭的时间可以设定在后面所指定的文件中, 设定的时间一到, 使无法登录 FTP 服务器了, 要恢复的话只有将这个文件删掉。文件的格式可以由指令 `/etc/ftpsht` 建立。
示例: `shutdown /etc/shutmsg`
- **delete** 选项(yes/no) 类别(real/anonymous/guest)
功能: 设定是否允许指定用户使用 delete 这个指令。
示例: `delet no guest, anonymous`
含义: 登录的用户为 guest 或 anonymous 时, 不能进行文件的删除操作。



- **overwrite** 选项(yes/no) 类别(real/anonymous/guest)
功能: 设定是否允许指定用户使用 **overwrite** 指令。
示例: `overwrite no guest,anonymous`
含义: 登录的用户为 **guest** 或 **anonymous** 时, 不能执行指令 **overwrite**。
- **readme** 选项(yes/no) 类别(real/anonymous/guest)
功能: 设定是否允许指定用户使用 **readme** 指令。
示例: `readme no guest,anonymous`
含义: 登录的用户为 **guest** 或 **anonymous** 时, 不能执行指令 **readme**。
- **chmod** 选项(yes/no) 类别(real/anonymous/guest)
功能: 设定是否允许指定用户使用 **chmod** 指令。
示例: `chmod no anonymous`
含义: 登录的用户为 **anonymous** 时, 不能执行指令 **chmod**。
- **umask** 选项(yes/no) 类别(real/anonymous/guest)
功能: 设定是否允许指定用户使用 **umask** 指令。
示例: `umask no anonymous`
含义: 登录的用户为 **anonymous** 时, 不能执行指令 **umask**。
- **upload** 根目录 上载目录 选项(yes/no) 用户群组权限 选项(dirs/nodirs)
功能: 这个指令设定哪一个目录是可以上载文件的。
示例 1: `upload /home/ftp /incoming yes root daemon 0600 dirs`
含义: 表示 `/home/ftp` 是指定根目录, 在这个目录中只有 `incoming` 这个子目录可以上载文件, 而上载文件的只有 `root` 用户而且群组是 `daemon`, 文件的操作权限为 `0600`, `dirs` 是指可以在这个目录中建立子目录。
示例 2: `upload /home/ftp /bin no`
示例 3: `upload /home/ftp /etc no`
含义: 表示 `/home/ftp/bin` 和 `/home/ftp/etc` 这两个目录是不能上载文件的。
- **alias** 目录别名 目录
功能: 给指定目录设置一个别名, 供换目录时, 只要用别名就可以了。这个功能当实际目录名很长时尤其有用。
示例: `alias inc:/incoming`
含义: 为目录 `/incoming` 设置别名 `inc`, 要进入 `/incoming` 目录只要执行下面的命令就行了: `cd inc`。
- **cdpath** 目录
功能: 这个功能与 DOS 中的路径设定有些类似, 但只用于 `cd` 命令时的路径搜索。例如当执行 `cd etc` 时, 首先在当前目录中搜索, 看是否有目录 `etc`, 如果没有, 则看是否有 `etc` 的别名设置, 如果也没有, 就根据 `cdpath` 指令所指定的顺序来搜索, 看是否有 `etc` 这个目录。
示例: `cdpath /incoming`



- cdpath /pub
- cdpath /
- 含义:搜索顺序为/incoming /pub 和 /。
- path - filter 类别(real/anonymous/guest) 目录
- 功能:设定上载的文件名限制。
- 示例: path - filter anonymous /etc/pathmsg '[-A-Za-z0-9_\.] * \$^\. ^-
- path - filter guest /etc/pathmsg '[-A-Za-z0-9_\.] * \$^\. ^-
- 含义:设定限制 anonymous 和 guest 用户上传的文件名只能包含 A~Z、a~z、0~9 和_ '但名字以“.”和“-”开头的文件也不能上载到 FTP 服务器。
- guestgroup 功能
- 功能:设定 guest 组的功能。
- 示例: guestgroup ftponly
- email guest 的 E-mail 地址
- 功能:若将某些 E-mail 地址设置在这个地方,则这些用户登入 FTP 服务器时,其身份为 guest,权限比 real 低一点,比 anonymous 高,但在此之前要将 guestgroup 指令删掉。
- 示例: email msf@263.net
- deny IP 地址/域名 说明文件
- 功能:设定限制哪一个 IP 地址或域名的用户登入 FTP 服务器。
- 示例: deny 166.112.1. * *.com.cn /etc/deny.msg
- 含义:使 IP 地址为 166.112.1. * 开头的或域名以 .com.cn 结尾的机器不能登入到服务器上,并将/etc/deny.msg 中的内容发送给该用户,以说明不能登录的原因。

2.1.5 配置用户控制文件 ftpusers

配置文件/etc/ftpusers 是用来限制用户使用 FTP 传送文件的,这样做的原因是为了系统能够更加安全。下面给出了系统默认的 ftpusers 文件。

```
root
bin
boot
daemon
digital
field
gateway
guest
nobody
operator
ris
```




```
socs  
sys  
uucp
```

2.1.6 配置主机控制文件 ftphosts

配置文件/etc/ftphosts 用来限制主机对 FTP 服务器的登录,这样做的原因也是为了系统能够更加安全。下面给出了系统默认的 ftphosts 文件。其中以 allow 打头的是可以登录的主机域,以 deny 打头的是禁止登录的主机域。

```
# Example host access file  
#  
# Everything after a '#' is treated as comment,  
# empty lines are ignored  
  
allow    bartm    somehost.domain  
deny     fred     otherhost.domain 131.211.32.*
```

2.1.7 测试服务器是否正常工作

当上面介绍的各种设置工作都已经完成时,我们就可以用下面的方法测试服务器是否正常工作:

```
[msf@linux msf] $ftp localhost  
Connected to localhost.  
220 linux.tsinghua.edu.cn FTP server (Version wu-2.6.0(1))  
2000) ready.  
Name (localhost:msf): ftp  
331 Guest login ok, send your complete e-mail address as password.  
Password:  
230 Guest login ok, access restriction apply.  
Remote system is UNIX.  
Using binary mode to transfer files.  
ftp> ls  
200 PORT command successful.  
150 Operatong ASCII mode data connection for /bin/ls.  
226 Transfer Complete.  
ftp> bye  
221 - You have transferred 0 bytes in 0 files.  
221 - Totel traffic for using the FTP service on linux.tsinghua.edu.cn.  
221 Goodbye.
```



```
LoadModule asis_module /usr/libexec/mod_asis.so
LoadModule imap_module /usr/libexec/mod_imap.so
LoadModule action_module /usr/libexec/mod_actions.so
LoadModule speling_module /usr/libexec/mod_speling.so
LoadModule userdir_module /usr/libexec/mod_userdir.so
LoadModule proxy_module /usr/libexec/libproxy.so
LoadModule alias_module /usr/libexec/mod_alias.so
LoadModule rewrite_module /usr/libexec/mod_rewrite.so
LoadModule access_module /usr/libexec/mod_access.so
LoadModule auth_module /usr/libexec/mod_auth.so
LoadModule anon_auth_module /usr/libexec/mod_auth_anon.so
LoadModule dbm_auth_module /usr/libexec/mod_auth_dbm.so
LoadModule db_auth_module /usr/libexec/mod_auth_db.so
LoadModule digest_module /usr/libexec/mod_digest.so
LoadModule cern_meta_module /usr/libexec/mod_cern_meta.so
LoadModule expires_module /usr/libexec/mod_expires.so
LoadModule headers_module /usr/libexec/mod_headers.so
LoadModule usertrack_module /usr/libexec/mod_usertrack.so
LoadModule example_module /usr/libexec/mod_example.so
LoadModule unique_id_module /usr/libexec/mod_unique_id.so
LoadModule setenvif_module /usr/libexec/mod_setenvif.so
ClearModuleList
AddModule mod_mmap_static.c
AddModule mod_env.c
AddModule mod_log_config.c
AddModule mod_log_agent.c
AddModule mod_log_referer.c
AddModule mod_mime_magic.c
AddModule mod_mime.c
AddModule mod_negotiation.c
AddModule mod_status.c
AddModule mod_info.c
AddModule mod_include.c
AddModule mod_autoindex.c
AddModule mod_dir.c
AddModule mod_oci.c
AddModule mod_asis.c
AddModule mod_imap.c
```



```
AddModule mod_actions.c
AddModule mod_speling.c
AddModule mod_userdir.c
AddModule mod_proxy.c
AddModule mod_alias.c
AddModule mod_rewrite.c
AddModule mod_access.c
AddModule mod_auth.c
AddModule mod_auth_anon.c
AddModule mod_auth_dbm.c
AddModule mod_auth_db.c
AddModule mod_digest.c
AddModule mod_cern_meta.c
AddModule mod_expires.c
AddModule mod_headers.c
AddModule mod_usertrack.c
AddModule mod_example.c
AddModule mod_unique_id.c
AddModule mod_so.c
AddModule mod_setenvif.c
ServerType standalone
Port 80
HostnameLookups off
User nobody
Group nobody
ServerAdmin root@localhost
ServerRoot "/home/httpd/html"
ErrorLog /var/log/httpd/error_log
LogLevel warn
LogFormat "%h %l %u %t \"%r\" \"%s %b\" \"%{Referer}i\" \"%{User-Agent}i\""
combined
LogFormat "%h %l %u %t \"%r\" \"%s %b\" common"
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{User-Agent}i" agent
CustomLog /var/log/httpd/access_log common
CustomLog /var/log/httpd/referer_log referer
CustomLog /var/log/httpd/agent_log agent
PidFile /var/run/httpd.pid
```



```
ExtendedStatus On
ServerSignature on
UseCanonicalName on
Timeout 300
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 15
MinSpareServers 8
MaxSpareServers 20
StartServers 10
MaxClients 300
MaxRequestsPerChild 100
ProxyRequests On
ProxyVia On
CacheRoot /home/httpd/proxy
CacheSize 5
CacheGcInterval 4
CacheMaxExpire 24
CacheLastModifiedFactor 0.1
CacheDefaultExpire 1
```

下面我们就逐一讲解配置文件 `httpd.conf` 中涉及到的各种指令：

- **ServerType** 选项(`standalone/inted`)
功能: 设定服务器的启动方式, 是开机后立即启动(`standalone`), 还是需要时临时启动(`inted`), 一般使用 `standalone` 方式。
示例: `ServerType standalone`
含义: 开机后立即启动 `apache` 服务器。
- **ServerRoot** 路径
功能: 指明服务器所在的路径。
示例: `ServerRoot /usr/local/etc/apache`
含义: 说明服务器所在的路径为 `/usr/local/etc/apache`。
- **PidFile** PID 文件
功能: 指明服务器进程 ID 的文件。
示例: `PidFile /usr/local/etc/apache/logs/httpd.pid`
含义: 将服务器进程的 ID 记录在文件 `/usr/local/etc/apache/logs/httpd.conf` 中。
- **Port** 端口号
功能: 为该服务器的服务指定端口号, 默认的 `WWW` 服务器的端口号为 80。如果设定了别的端口号, 在连接时就要指出这个端口号。



示例:Port 100

含义:设定端口号为 100,则连接的 URL 就要写成 `http://host.domain:100/`。

- User 用户名/#uid

功能:设定 httpd 的真正执行者,这对确保系统的安全很重要,这里的用户名必须是/etc/passwd 中已定义的使用者。

示例:User nobody

- Group 组名/#gid

功能:含义与 User 类似,只是这里设定的是群组。

示例:Group nobody

- ServerAdmin 管理员 E-mail 地址

功能:设定管理员的 E-mail 地址,这个地址会出现在系统连接出错的时候,以便访问者与管理员联系。

示例:ServerAdmin msf@mail.tsinghua.edu.cn

- BindAddress 选项(* /IP/FQDN)

功能:设定要从哪个地址接受服务,可以使用 IP 或完整的主机名(FQDN),建议使用“*”来表示全部接受。

示例:BindAddress *

- ErrorLog 记录文件名

功能:设定错误信息的记录文件,如果文件名不是以“/”开头的,那就不会连接到 ServerRoot 的路径下。

示例 1:ErrorLog error/error.log

含义:将错误信息记录到错误记录文件/usr/local/etc/apache/error/error.log,这里的 ServerRoot 为/usr/local/etc/apache。

示例 2:ErrorLog /usr/local/error/error.log

含义:将错误信息记录到/usr/local/error/error.log。

- TransferLog 记录文件名

功能:设定传输的记录文件,设定方式同上。

示例:TransferLog logs/access-log

- ScoreBoardFile 被使用过的文件名

功能:记录被使用过的文件,设定方式同上。

示例:ScoreBoardFile logs/apache status

- Server Name 主机名称

功能:设定此服务器名称,但一定要记住在 DNS 上定义这个服务器名。

示例:Server Name tsinghua.edu.cn

- CacheNegotiateDocs

功能:Apache 服务器在发送数据时都会加上 Pragma:no-cache 这个文件头,也就是不让代理服务器将数据留在缓存中。



示例:CacheNegotiateDocs

- Timeout 秒数

功能:设定客户端终止的等待时间,只要客户端超过此设定的秒数而没有完成指令,就会终止服务,如果网络速度较慢,建议设定较长的秒数。

示例:Timeout 1600

- KeepAlive 选项(On/Off)

功能:是否需要开启连续请求的功能,On 是开启,Off 是拒绝。

示例:KeepAliveOn

- MaxKeepAliveRequests 次数

功能:最大的连续请求的次数,如果连续请求大于这个数,则服务器会自动将其清除。

示例:MaxKeepAliveRequests 100

- KeepAliveTimeout 秒数

功能:设定客户端连续请求的等待时间,如果客户端连续请求的时间超过此数,则不再执行此请求。

示例:KeepAliveTimeout15

- MinSpareServer 数目

功能:设定等待服务的服务器最小闲置个数,当系统的闲置个数小于此设定时,系统会自动开启更多的服务器,如果实际应用中的服务器活动很频繁,则应该将该数目设得大些。

示例:MinSpareServer5

- MaxSpareServer 数目

功能:设定等待服务的服务器之最大闲置个数,当系统中的闲置个数大于此设定值时,多余的将被删掉。

示例:MaxSpareServer10

- StartServer 数目

功能:刚刚启动服务器时,要开启多少服务器进行服务。

示例:StartServers 5

- MaxClients 数目

功能:设定客户端可以连接的最大上限,当客户端达到此数时,就不再接受新的客户端连接了,主要目的是要维护系统的稳定性,控制系统的负载。

示例:MaxClients 150

- MaxRequestsPerChild 次数

功能:设定每个子程序可以接受的最大服务次数,当达到此数目时,子程序会自动结束以免出现漏失。除了 Solaris 等少数系统,大部分系统都不会出现此现象。

示例:MaxRequestsPerChild 30

- ProxyRequests 选项(On/Off)



功能:设定是否要将代理的功能打开,大部分是不需要的,将其省略就可以了。

示例:ProxyRequests On

- < VirtualHost host.some-domain.com > ... < /VirtualHost >

功能:设定虚拟主机,必须是支持多地址或有一块以上网卡的主机才适用此设定。

示例: < VirtualHost dns.tsinghua.edu.cn >

```
ServerAdmin msf@mail.tsinghua.edu.cn
DocumentRoot /home/www/
ServerName tsinghua.edu.cn
ErrorLog logs/mouse-error-log
TransferLog logs/mouse-access-log
< /VirtualHost >
```

含义:假设原来这台主机的名称为 www.tsinghua.edu.cn,而在这台主机上使用了多地址方式产生了另一个 IP 地址并注册为 dns.tsinghua.edu.cn。当客户端以 dns.tsinghua.edu.cn 来和这台主机进行连接时,服务器会用以上的信息来作为回应。

2.2.5 配置访问控制文件 access.conf

配置文件/etc/httpd/conf/access.conf 主要是用来设置系统的存取方式和环境的,系统默认的 access.conf 如下所示:

```
<Directory />
Options FollowSymLinks
AllowOverride None
</Directory>
<Directory "/home/httpd/html">
Options Indexes FollowSymLinks
AllowOverride None
order allow,deny
allow from all
</Directory>
<Directory "/home/httpd/cgi-bin">
AllowOverride None
Options ExecCGI
</Directory>
<Location /cgi-bin/phf*>
deny from all
ErrorDocument 403 http://phf.apache.org/phf_abuse_log.cgi
</Location>
```



下面我们就逐一讲解配置文件 `access.conf` 中涉及到的几条指令。

- `<Directory 目录> ... </Directory>`

功能: 用来设定所指定目录可以使用的控制权限。

示例: `<Directory /usr/local/etc/httpd/htdocs>`

`</Directory>`

含义: 设定 `/usr/local/etc/httpd/htdocs` 这个目录的使用权限。

- Options 选项

功能: 控制所设定的目录中可以使用的功能。

可用的选项有:

All	准许以下所有的功能 (Multiviews 除外)
Multiviews	准许内容协商的 Multiviews
Indexes	若不含索引文件, 则准许该目录以下的文件供选择
IncludesNOEXEC	准许 SSI, 但不可使用 <code>#exec</code> 和 <code>#include</code> 的功能
Includes	准许 SSI, 也就是 Server-side Includes
FollowSymLinks	准许符号链接到其他目录
ExecCGI	准许该目录下使用 CGI
YmLinksfOwner Math	准许符号链接的目录, 即使该目录的属主与原目录的属主不同也准许

示例: `Options Indexes FollowSymLinks`

2.2.6 配置资源控制文件 `srm.conf`

`/etc/httpd/conf/srm.conf` 主要完成对资源的配置, 默认的 `srm.conf` 如下所示:

```
DocumentRoot "/home/httpd/html"
UserDir myhtml
DirectoryIndex index.html index.shtml index.cgi
FancyIndexing on
AddIconByEncoding (CMP,/icons/compressed.gif) x-compress x-gzip
AddIconByType (TXT,/icons/text.gif) text/*
AddIconByType (IMG,/icons/image2.gif) image/*
AddIconByType (SND,/icons/sound2.gif) audio/*
AddIconByType (VID,/icons/movie.gif) video/*
AddIcon /icons/binary.gif .bin .exe
AddIcon /icons/birtex.gif .hqx
AddIcon /icons/tar.gif .tar
AddIcon /icons/world2.gif .wrl .wrl.gz .vml .vrm .iv
AddIcon /icons/compressed.gif .Z .z .tgz .gz .zip
AddIcon /icons/a.gif .ps .ai .eps
```




功能:设定谁能从这个服务器取得控制,选项为 deny 和 allow。

示例:Order allow,deny

allow from all

含义:所有的人都可以取得控制。

- AddIconByEncoding 图标文件的位置 MIME - encoding 方式

功能:当 FancyIndex 列出文件时,根据不同的 MIME - encoding 方式以不同的图标来表示。

示例:AddIconByEncoding(CMP,./icons/compressed.gif)x-compress x-gzip

- AddIconByType 图标位置 MIME - type 类型

功能:类似 AddIconByEncoding,但这是以 MIE.M - type 方式表示。

示例:AddIconByType(TXT./icons/text.gif)text/*

- AddIcon 图标的位置文件

功能:以文件的后缀来定义相关的图标。

示例:AddIcon /icons/binary.gif.bin.exe

- DefaultIcon 图标位置区

功能:设定当文件没有定义相关的图标表示时,以哪种图标表示。

示例:DefaultIcon /icons/unknown.gif

- ReadName 文件名

功能:在 FancyIndex 时设置文件的说明。

示例:ReadName README

- HeaderName 文件名

功能:与 ReadName 相似,但在 FancyIndex 开始时。显示顺序与前者相同。

示例:HeaderName HEADER

- IndexIgnore 文件

功能:设定在 FancyIndex 时不列出的文件。

示例:IndexIgnore README.htaccess

- AccessFileName 文件名

功能:这是设置控制权限的文件,如果客户端要读取的目录中有此设定文件,则服务器会要求确认用户的身份,后面将有详细的说明。

示例:AccessFileName .htaccess

- DefaultType MIE.M 文件类型

功能:当有服务器不认识的文件类型时,在传输给客户端时应该告诉客户端数据的文件类别。

示例:DefaultType text/plain。

含义:将不认识的文件类型定义为 text/plain。

- AddEncoding MIME - enc 文件类型

功能:设定文件的压缩类型,主浏览器在传回时可以直接解压缩并显示。



示例: `AddEncoding x-compress Z`

- `AddLanguage MIME-lang` 类型

功能: 设定文件的语言类型, 大部分是针对非英文显示方式的设置。

示例: `AddLanguage en-en`

- `LanguagePriority MIME-lang`

功能: 设定语言的类型优先顺序。

示例: `LanguagePriority en fr de`

- `Redirect` 文件名 url

功能: 设定当客户访问不存在的文件时通知客户转到另外一个 URL。

示例: `Redirect welcome.html http://host.domain/~msf/index.html`

含义: 当用户要访问 `welcome.html` 这个文件时, 服务器会传回一个含 `host.domain/~msf/index.html` 的文件头, 让客户端转向取得的新文件。

- `Alias` 虚拟目录, 实际目录。

功能: 设定非 `DocumentRoot` 下的目录所对应的目录位置。

示例: `Alias /www/home/www`

含义: 当客户端连接到 `www` 时, 服务器会将文件对应到 `/home/www/`, 而不是原来 `DocumentRoot` 的 `www` 之下。

- `ScriptAlias` 虚拟目录, 实际目录

功能: 类似 `Alias`, 但这里设定的是执行文件的目录区, 大部分是指 CGI 目录。

示例: `ScriptAlias /cgi-bin /usr/local/etc/apache/cgi-bin`

- `AddType MIME-type` 类型

功能: 设定不存在于文件 `mime.type` 中的其他 MIME 类型。

示例: `AddType type/tasttype type`

- `AddHandler MIME-type` 类型

功能: 跟 `AddType` 类似, 但这里设定的是执行文件, 大部分是指 CGI 文件。

示例: `AddHandler cgi-script.cgi`

- `ErrorDocument` 错误编号, 文件/文字说明

功能: 设定当服务器在回应错误信息给客户端时, 不采用默认的信息, 而是采用自定义的方式。如果要用文字说明, 则要以“`”`”为开始, 但不必以“`”`”结束。

示例: `ErrorDocument 404/missing.html`

`ErrorDocument 405`“信息有错误……”

2.2.7 运行服务器程序

设置完成之后, 我们可以键入 `httpd` 命令启动 Apache 服务器, 然后就可以用如下的命令看到 `httpd` 进程在运行:

```
ps -aux
```



2.2.8 测试服务器运行情况

启动服务器后,我们就可以测试它的工作情况了,在浏览器的位置栏键入 `http://linux/`,其中 `linux` 为安装 Apache 的机器,运行情况如图 2.1 所示。

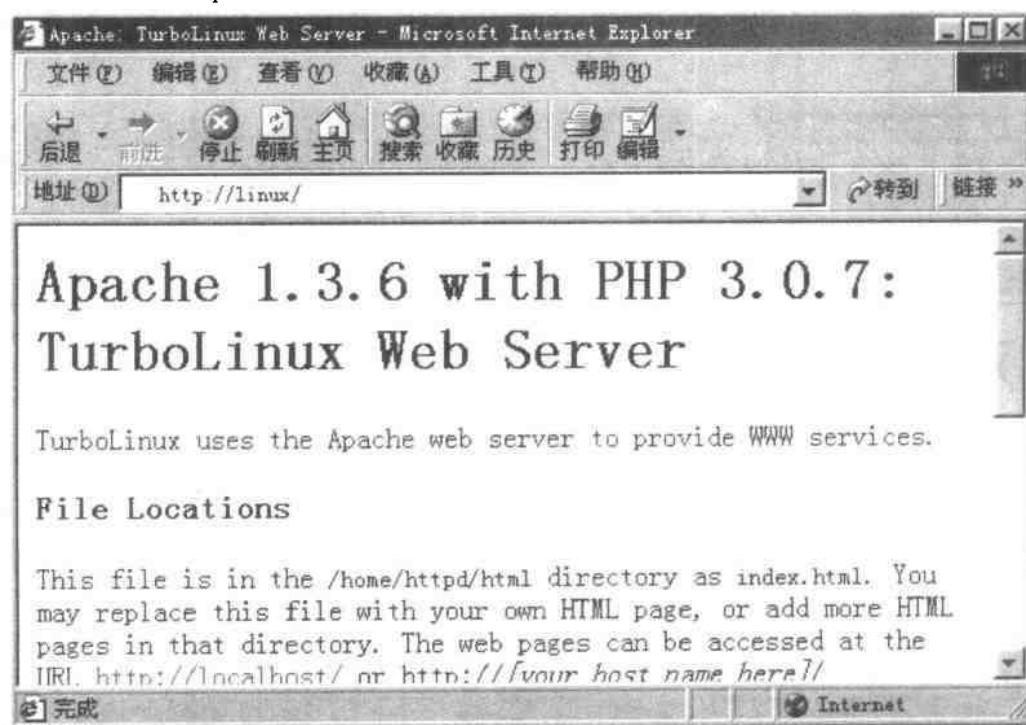


图 2.1 测试服务器运行情况

由图 2.1 可以知道,Apache 服务器已经成功地启动了

2.2.9 浏览个人主页

接下来我们要进行个人主页的测试,首先要先确认 `srm.conf` 这个文件中有相关的设定,如下面所示:

```
userdir myhtml
```

这里设成了目录 `myhtml`,即表示用户的个人主页目录为 `myhtml`。如果系统中有一个叫 `msf` 的用户,他的根目录为 `/home/msf`,并且写了一个个人主页,内容如下:

```
<HTML>
<HEAD>
<TITLE>我的主页</TITLE>
</HEAD>
<BODY>
<H1>欢迎光临个人主页</H1>
```

第 3 章 Linux 文件系统

3.1 目 录

3.1.1 目录结构

数据和程序被组织成文件的形式存放在磁盘上,为了有效地管理这些文件,引进了目录与文件的概念,目录与文件的定义如下:

目录:目录是一种数据结构,它记录所有附属于它的目录和文件的位置。最高的目录称为根目录。

文件:文件也是一种数据结构,它直接记载所要保存的数据。在 Linux 系统中,文件可以分为两类:文本文件与二进制文件。

在 Linux 系统中,目录可以含有任意数量的子目录和文件,而子目录同样还可以再有子目录,如此构成一个层次化的结构。图 3.1 就给出了一个简单的 Linux 目录结构,在 Linux 系统中,根目录用“/”来表示。

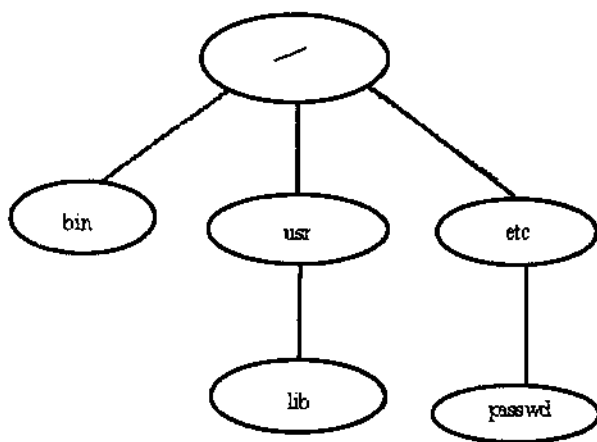


图 3.1 Linux 的目录结构

在图 3.1 中,bin,usr 和 etc 都是根目录(/)下的第一层子目录,而 usr 下的 include 则为再下一层的子目录,passwd 则为 etc 目录下的文件。

Linux 系统是用完整的路径来识别文件的,如 passwd 文件的完整路径为/etc/passwd。并非在所有的情况下程序都要给出完整的路径名。在每一个时间点上,每个进程都有一个



称为当前工作目录的相关目录,它可用于路径的确定。如果路径名不以 / 开头,则认为以当前工作目录的路径开头。单点(.)可用来指明当前目录,而双点(..)则可用来指明当前目录的上一级目录。对根目录则单点和双点都指向其自身。与用户登录相关联的当前工作目录为这个用户的主目录。

3.1.2 getcwd 函数

函数 `getcwd` 用于得到当前工作目录的路径名。

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

在函数 `getcwd` 中,第 1 个参数为缓冲地址 `buf`;第 2 个参数为给出的最大路径名长度 `size`,如果当前工作目录的路径名长度长于给定的长度,则函数返回 `NULL` 并置 `errno` 为 `ERANGE`。函数调用成功时,返回指向路径名的指针;否则返回 `NULL`。

下面的程序输出当前的工作目录的路径名。

```
/* filename: ex3_1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#ifndef MAX_PATH
#define MAX_PATH 255
#endif
int main()
{
    char name[MAX_PATH+1];
    if (getcwd(name, MAX_PATH) == NULL)
    {
        printf("error: getcwd \n");
        exit(1);
    }

    printf("Current working directory: %s \n", name);
    exit(0);
}
```

下面给出了本程序的运行结果:

```
[root@linux /root] # cc ex3_1.c
[root@linux /root] # a.out
Current working directory: /root
```



3.1.3 读取目录

为了在程序中操作某一目录中的文件,首先得读取该目录。读取目录要用到 `opendir`、`readdir`、`rewinddir` 和 `closedir` 这些函数。

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

DIR * opendir(const char * dirname);
struct dirent * readdir (DIR * dirp);
void rewinddir (DIR * dirp);
int closedir (DIR * dirp);
```

在上面函数中涉及的结构 `dirent` 的定义如下:

```
struct dirent
{
    ino64_t d_ino;
    off64_t d_off;
    unsigned short int d_reclen;
    unsigned char d_type;
    char d_name[256]; /* We must not include limits.h! */
};
```

在 `dirent` 结构中,最重要的字段是 `d_name` 和 `d_type`。`d_name` 为文件或子目录的名称。`d_type` 说明类型,常见的类型如表 3.1 所示。

表 3.1 常见的文件类型

字段 <code>d_type</code> 的值	表示的含义
<code>DT_FIFO</code>	FIFO,即命名管道
<code>DT_CHR</code>	字符设备文件
<code>DT_BLK</code>	块设备文件
<code>DT_DIR</code>	子目录
<code>DT_REG</code>	一般文件
<code>DT_LNK</code>	链接文件
<code>DT SOCK</code>	套接字文件
<code>DT_UNKNOWN</code>	未知类型的文件

但是遗憾的是,在实现过程中, Linux 系统并没有考虑到字段 `d_type`,而是简单地将 `d_type` 置为 0,即 `DT_UNKNOWN`。如要得到类型,那还得调用函数 `stat`,关于 `stat`,在下面的章节中会给予介绍。

函数 `opendir` 的参数为要打开的目录名 `dirname`,返回一个指向 `DIR` 的指针,这个指针



在后面的函数中都将用到

函数 `readdir` 的参数为调用 `opendir` 生成的指针,函数返回一个指向 `dirent` 结构的指针,如果返回 `NULL`,则表示该目录下的所有文件和子目录都已经遍历过了。

函数 `rewinddir` 用来重新开始遍历目录,参数也为调用 `opendir` 返回的指针。

函数 `closedir` 在目录遍历结束后调用。

下面的例子用来列出某一目录下的所有文件和子目录,该程序的参数为目录名。

```
/* filename: ex3_2.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
void usage(char * name)
{
    printf("Usage: %s dirname \n", name);
    exit(1);
}
int main(int argc, char * * argv)
{
    DIR * dirp;
    struct dirent * direntp;
    if (argc != 2)
        usage(argv[0]);
    if ((dirp = opendir(argv[1])) == NULL)
    {
        printf("Could not open directory: %s \n", argv[1]);
        exit(1);
    }
    while((direntp = readdir(dirp)) != NULL)
    {
        printf("%s \n", direntp->d_name);
    }
    closedir(dirp);
    exit(0);
}
```

程序的运行结果如下面所示:

```
[root@linux /root] # cc ex3_2.c
[root@linux /root] # a.out rootdir
```

```

.
..
filec
dir0
filea
fileb

```

3.2 文 件

3.2.1 文件的存储

在 Linux 中,一个文件对应于一个描述,被存储在一个称为索引节点(简称 i 节点)的结构中。索引节点包含了文件的很多信息,如文件长度、文件位置、文件所有者、创建时间、上次存取时间、上次修改时间、许可权限等。目录也表示成文件的形式并具有一个关联的索引节点,设备被表示成特殊文件,FIFO 文件用于进程间的通信。

图 3.2 给出了文件的索引节点结构。除了有关文件的描述信息外,索引节点还含有一些指向文件开头一些数据块的指针。如果文件很大,间接指针将含有一个指向指针块的指针,而指针块的指针指向附加的数据块。如果文件还要大,二次间接指针将含有一个指向间接指针块的指针,而间接指针指向直接指针块,直接指针指向间接数据块后的数据块。如果文件更大,三次间接指针将含有一个指向二次间接指针块的指针。

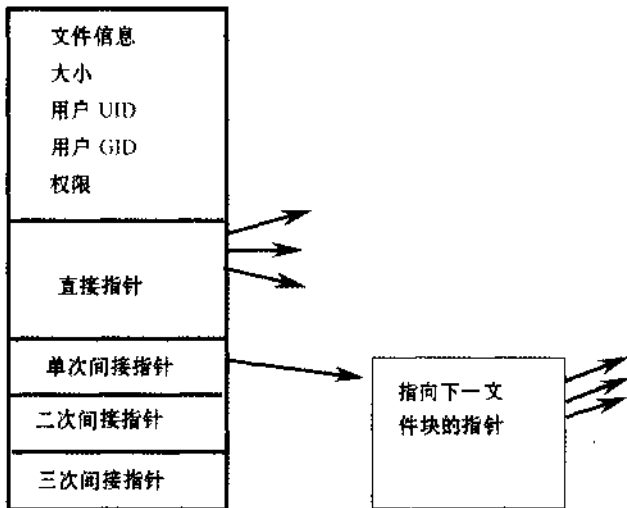


图 3.2 文件的索引节点结构

在 Linux 中,典型块的大小为 8KB,如果一个索引节点为 128B,指针为 4B,而状态信息占了 68B。这样直接指针的个数为 $(128 - 68 - 12)/4 = 12$,直接指针的寻址能力为 $(12 \times 8192) = 98304B$ 。单次间接指针指向一大小为 8KB 的块,一块中最多可保存 $8192/4 = 2048$



个指针,所以单次间接指针的寻址能力为 $(2048 \times 8192) = 16777216\text{B}$ 约 16MB。同样二次间接指针的寻址能力 $2048 \times 2048 \times 8192\text{B}$ 约 32GB,三次间接指针的寻址能力为 $2048 \times 2048 \times 2048 \times 8192\text{B}$ 约 64TB。所以在 Linux 中,一个文件大小的极限约为 64T。

3.2.2 文件的存储权限

在 Linux 系统中,每个文件都有一个 16 位的字段表示该文件的属性、用户、用户组标识符以及存储权限,其中 0 到 8 位表示文件的存储权限,9、10、11 三位则用来控制文件被装入内存后是否被置换出,用户标识符设定状态及用户组标识符设定状态,而 12 到 15 位则在文件创建时被设定好,它的值可以被读取却不能被修改。所有表示文件状态的位元及表示的意义如表 3.2 所示。

表 3.2 文件的存储权限

位 元	符号常量	表 示 的 意 义
15	S_IFREG	一般文件
14	S_IFDIR	目录
13	S_IFCHR/S_IFBLK	字符或块设备文件
12	S_IFIFO	命名管道
11	S_ISUID	用户标识符是否被设定
10	S_ISGID	用户组标识符是否被设定
9	S_ISVTX	是否被置换
0~8	rwX-rwX-rwX	文件存取权限

由表 3.2 可以看出,12~15 位是用来判别文件类型的,它们在文件创建时就被设置,而且不能更改。这 4 位中仅有一个会被置为 1,这是因为任何一个文件只能具有其中一种属性。

下面讨论存储权限的 0~8 位,如图 3.3 所示。

8	7	6	5	4	3	2	1	0
r	w	x	r	w	x	r	w	x

图 3.3 文件存储权限的 0~8 位

各位表示的意义如表 3.3 所示。

表 3.3 文件存储权限的 0~8 位的含义

位	表 示 的 含 义	位	表 示 的 含 义
8	所有者有权读取	7	所有者有权写入
6	所有者有权执行文件	5	同组的用户有权读取
4	同组的用户有权写入	3	同组的用户有权执行文件
2	其他用户有权读取	1	其他用户有权写入
0	其他用户有权执行文件		



`r, w, x` 对于不同类型的文件有不同的含义,具体如表 3.4 所示。

表 3.4 `r, w, x` 的具体含义

	一般文件	目 录	设备文件
<code>r</code>	可读取和拷贝文件内容	可显示目录内容	可由此设备读入数据
<code>w</code>	可修改和删除该文件	可将数据写入该目录	可将信息由此设备输出
<code>x</code>	可执行该文件	可搜索该目录	无意义

表示存储权限的 0~8 位经常用三位的八进制数表示,例如八进制数 755 表示任何用户都可以读取和执行该文件,但只有文件的所有者才可以修改该文件。

3.2.3 stat 和 fstat 函数

函数 `stat` 和 `fstat` 可以得到文件的信息。

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

int stat (const char * file, struct stat * buf);
int fstat (int fd, struct stat * buf);
```

如果调用成功,函数 `stat` 和 `fstat` 都返回 0;如果失败,则返回 -1。函数 `stat` 的第 1 个参数为文件的路径 `file`,第 2 个参数为存放文件信息的地址 `buf`;函数 `fstat` 的第 1 个参数为文件的描述字 `fd`,第 2 个参数也是存放文件信息的地址 `buf`。文件信息存放在结构 `stat` 中,下面是 `stat` 结构的定义:

```
struct stat
{
    dev_t st_dev;        /* Device. */
    ino_t st_ino;        /* File serial number. */
    mode_t st_mode;      /* File mode. */
    nlink_t st_nlink;    /* Link count. */
    uid_t st_uid;        /* User ID of the file's owner. */
    gid_t st_gid;        /* Group ID of the file's group. */
    dev_t st_rdev;       /* Device number, if device. */
    off_t st_size;       /* Size of file, in bytes. */
    time_t st_atime;     /* Time of last access. */
    time_t st_mtime;     /* Time of last modification. */
    time_t st_ctime;     /* Time of last status change. */
};
```

在 `stat` 结构中,`st_dev` 表示磁盘设备的标识符;`st_ino` 表示文件的 i 节点值;`st_mode` 表示文件的存储权限;`st_nlink` 表示文件的链接数;`st_uid` 表示文件所有者的用户标识符;



st_gid 为文件所有者的用户组标识符;st_rdev 表示该文件所指向设备的标识符,这只有当该文件为设备文件时才有意义;st_size 为文件的长度,设备文件的长度为 0;st_atime 为最近一次访问文件的时间;st_mtime 为最近一次修改文件的时间;st_ctime 为最后一次更改文件属性的时间。

下面给出的例子用来显示文件的信息,该程序的参数为文件名。在程序中出现的常量 S_IFMT 在 sys/stat.h 中被定义为八进制数 170000,语句 st.st_mode & S_IFMT 可以得到文件的类型。

```
/* filename: ex3_3.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <time.h>
void usage(char * name)
{
    printf("Usage: %s filename", name);
    exit(1);
}
int main(int argc, char * * argv)
{
    struct stat st;
    int isdevice = 0;
    if (argc != 2)
        usage(argv[0]);
    if (stat(argv[1], &st) == -1)
    {
        printf("error: stat \n");
        exit(1);
    }
    if ((st.st_mode & S_IFMT) == S_IFDIR)
        printf("type: Directory \n");
    else if ((st.st_mode & S_IFMT) == S_IFBLK)
    {
        printf("type: Block special file \n");
        isdevice = 1;
    }
    else if ((st.st_mode & S_IFMT) == S_IFCHR)
    {

```



```

        printf("type:Character special file \n");
        isdevice = 1;
    }
    else if ((st.st_mode & S_IFMT) == S_IFREG)
        printf("type:Ordinary file \n");
    else if ((st.st_mode & S_IFMT) == S_IFIFO)
        printf("type:FIFO(named pipe) \n");
    if (isdevice)
    {
        printf("Device number: %d, %d \n",
               (st.st_rdev > > 8) & 0xff, st.st_rdev & 0xff);
    }
    printf("Resides
device: %d, %d \n", (st.st_dev > > 8) & 0xff, st.st_dev & 0xff);
    printf("l- node: %d, links: %d, size: %ld \n", st.st_ino, st.st_nlink, st.st_size);
    printf("Owner ID: %d, group ID %d \n", st.st_uid, st.st_gid);
    printf("Permission: %o \n", st.st_mode & 0177);
    printf("Last access: %15s", ctime(&st.st_atime));
    printf("Last modification: %15s", ctime(&st.st_mtime));
    printf("Last inode change: %15s", ctime(&st.st_ctime));
    exit(0);
}

```

程序运行情况如下面所示:

```

[root@linux /chapter03] # cc ex3_3.c
[root@linux /chapter03] # a.out /dev/fd0
type:Block special file
Device bunber:2,0
Resides on device:3,0
l- node:66548, links:1, size:0
Owner ID:0, group ID 19
Permission:600
Last access:Wed May 6 04:23:26 2000
Last modification:Wed May 6 04:32:26 2000
Last inode change:Wed Apr 29:23:59:24 2000

```

下面给出一个遍历目录的程序,参数为目录名。

```

/* filename:ex3_4.c */
#include <stdio.h>

```



```
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/stat.h>
#include <string.h>

void usage(char * name)
{
    printf("Usage: %s dimame \n", name);
    exit(1);
}

int dirwalk(char * dimame)
{
    char filename[256];
    DIR * dirp;
    struct dirent * direntp;
    struct stat st;

    if ((dirp = opendir(dimame)) == NULL)
    {
        printf("Coult not open directory: %s \n", dimame);
        exit(1);
    }

    while((direntp = readdir(dirp)) != NULL)
    {
        strcpy(filename, dimame);
        if (filename[strlen(filename) - 1] != '/')
            strcat(filename, "/");

        if ((strcmp(direntp -> d_name, ".") != 0)
            && (strcmp(direntp -> d_name, "..") != 0))
        {
            strcat(filename, direntp -> d_name);

            if (stat(filename, &st) == -1)
            {
                continue;
            }
        }
    }
}
```



```

        printf("error: stat \ n");
        exit(1);
    }
    if ((st.st_mode & S_IFMT) == S_IFDIR)
    {
        dirwalk(filename);
    }
    else
        printf("%s \ n", filename);
}

}

closedir(dirp);
}

int main(int argc, char * * argv)
{

    if (argc != 2)
        usage(argv[0]);

    dirwalk(argv[1]);

    exit(0);
}

```

程序的运行情况如下面所示：

```

[root@linux /chapter03] # cc ex3_4.c
[root@linux /chapter03] # a.out rootdir
rootdir/filec
rootdir/dir0/filea
rootdir/dir0/fileb
rootdir/filea
rootdir/fileb

```

3.2.4 得到用户的信息

在上一节中讲解了得到文件信息的系统调用,在这一节中,将介绍得到用户和用户组信息的系统调用。

函数 `getpwnuid` 和 `getpwnam` 用于得到用户的信息。



在 `group` 结构中, `gr_name` 为用户组的名字; `gr_passwd` 为加密后的密码; `gr_gid` 为用户组 ID; `gr_mem` 为用户列表

下面的例子用于得到用户的信息并输出, 程序参数为用户名字或用户 ID。

```
/* filename: ex3_5.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>

void usage(char * name)
{
    printf("Usage: %s username/userID \n", name);
    exit(1);
}

int main(int argc, char * * argv)
{
    struct passwd * pwd;
    struct group * grp;
    char * * namelist;
    int i;

    if (argc != 2)
        usage(argv[0]);

    if ((pwd = getpwnam(argv[1])) == NULL)
    {
        if ((pwd = getpwuid(atoi(argv[1]))) == NULL)
        {
            printf("error: username or user ID \n");
            exit(1);
        }
    }

    printf("User name : %s \n", pwd->pw_name);
    printf("User passwd : %s \n", pwd->pw_passwd);
    printf("User ID : %d \n", pwd->pw_uid);
    printf("User Gid : %d \n", pwd->pw_gid);
}
```



```
printf("User gecos : %s \n", pwd->pw_gecos);
printf("User Dir : %s \n", pwd->pw_dir);
printf("User shell : %s \n", pwd->pw_shell);
if ((grp = getgrgid(pwd->pw_gid)) == NULL)
{
    printf("error: getgrgid \n");
    exit(1);
}

printf("Group name : %s \n", grp->gr_name);
printf("Group passwd: %s \n", grp->gr_passwd);
printf("Group Member: \n");
namelist = grp->gr_mem;
for (i = 0; namelist[i] != NULL; i++)
{
    printf("\t%s \n", namelist[i]);
}

exit(0);
}
```

程序的运行情况如下面所示:

```
[root@linux /chapter03] # cc ex3_5.c
[root@linux /chapter03] # a.out msf
User name      :msf
User passwd    :x
User ID        :501
User GID       :500
User gecos     :msf
User Dir       :/home/msf
User shell     :/bin/sh
Group name     :msf
Group passwd   :*
Group Member   :
```

3.3 文件系统信息

3.3.1 ustat 函数

函数 ustat 的定义如下所示:



```
if (ustat(atoi(argv[1]), &ubuf) == -1)
{
    printf("Could ustat file system on device %s \n", argv[1]);
    exit(1);
}

printf("The total number of free space is %d \n", ubuf.f_tfree);
printf("The total number of i_nodes is %d \n", ubuf.f_tinode);
printf("The fsname is %6s, the pack name is %6s \n",
        ubuf.f_fname, ubuf.f_fpack);
exit(0);
}
```

程序的运行情况如下所示：

```
_root@linux /chapter03] # cc ex3_6.c
[root@linux /chapter03] # a.out 0003
The total number of free space is :0
The total number of i_nodes is :0
the fsname is:
the pack name is:
```

3.3.2 statfs 和 fstatfs 函数

系统调用 statfs 和 fstatfs 如下所示：

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

int statfs (const char * file, struct statfs * buf);
int fstatfs (int fd, struct statfs * buf);
```

这两个函数完成相同的功能，都是将文件系统的信息存入 statfs 结构中。不同的是：statfs 的第 1 个参数为文件系统所在的设备文件的路径 file，而 fstatfs 的第 1 个参数为与文件系统所在的设备文件相对应的文件描述符 fd。这两个函数的第 2 个参数为一个指向 statfs 结构的指针，即存放信息的地址。这两个函数调用成功时返回 0，失败时返回 -1。

下面是结构 statfs 的定义：

```
struct statfs
{
    int f_type;
    int f_bsize;
```



```
fsblkcnt_t f_blocks;  
fsblkcnt_t f_bfree;  
fsblkcnt_t f_bavail;  
fsfilcnt_t f_files;  
fsfilcnt_t f_ffree;  
fsid_t f_fsid;  
int f_namelen;  
int f_spare[6];  
};
```

在结构 `statfs` 中, `f_type` 为文件系统类型, `f_bsize` 为块的大小, `f_blocks` 为块的总数, `f_bfree` 为空闲块的数目, `f_bavail` 为可用块的数目, `f_files` 为 i 节点总数, `f_ffree` 为空闲的 i 节点数目, `f_fsid` 为文件系统标识符, `f_namelen` 为文件系统名称, `f_spare` 为卷名。

下面的程序显示了系统调用 `statfs` 的用法, 程序的参数为设备文件名。

```
/* filename: ex3_7.c */  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/statfs.h>  
  
void usage(char * name)  
{  
    printf("Usage: %s dev _file \n", name);  
    exit(1);  
}  
  
int main(int argc, char * * argv)  
{  
    struct statfs buf;  
  
    if (argc != 2)  
        usage(argv[0]);  
  
    if (statfs(argv[1], &buf) == -1)  
    {  
        printf("Could statfs file system %s \n");  
        exit(1);  
    }  
  
    printf("The type of the file system is: %d \n", buf.f_type);  
}
```

第 4 章 Linux 文件系统调用

4.1 文件描述符

在 C 程序中,文件是用文件指针或文件描述符来表示的。在 ANSI C 的标准 I/O 库(如 `fopen`、`fread` 等)使用文件指针,而在 Linux 和 Unix 的 I/O 库(如 `open`、`read` 等)都使用文件描述符。文件指针和文件描述符为执行设备无关的输入和输出提供逻辑名或句柄。标准输入、标准输出和标准错误输出的文件指针句柄分别为 `stdin`、`stdout` 和 `stderr`,它们在文件 `stdio.h` 中定义;标准输入、标准输出和标准错误输出的文件描述符句柄分别为 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO`,它们在文件 `unistd.h` 中定义。

4.2 `open` 和 `close` 函数

系统调用 `open` 用于打开或创建一个文件,在文件 `fcntl.h` 中定义;`close` 用于关闭一个文件,在文件 `unistd.h` 中定义。

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

int open (const char * file, int oflag, ...);
int close (int fd);
```

函数 `open` 的第 1 个参数 `file` 为要打开或要创建的文件名;第 2 个参数为选项,常见的选项有如下几种:

- `O_RDONLY` 以只读的方式打开。
- `O_WRONLY` 以只写的方式打开。
- `O_RDWR` 以读、写的方式打开。
- `O_APPEND` 将写的内容加到文件的末尾。
- `O_CREAT` 若文件不存在则创建它。使用此选项时,需同时指定第三个参数 `mode`,用其来说明文件的存取权限。
- `O_EXCL` 如果同时指定了 `O_CREAT`,而且文件已经存在,则出错。这个选项可以用来测试一个文件是否存在。
- `O_TRUNC` 如果文件存在,而且为只读或只写成功打开,则将文件长度截短为 0。



```

{
    char sourcefile[256];
    char destfile[256];
    char buf[1024];
    DIR *dirp;
    struct dirent *direntp;
    struct stat st;
    int sourcefd, destfd;
    int num;

    if (mkdir(destdir, 0x1ff) == -1)
    {
        printf("could not create dir: %s \n", destdir);
        exit(1);
    }

    if ((dirp = opendir(sourcedir)) == NULL)
    {
        printf("Could not open directory: %s \n", sourcedir);
        exit(1);
    }

    while((direntp = readdir(dirp)) != NULL)
    {
        strcpy(sourcefile, sourcedir);
        if (sourcefile[strlen(sourcefile) - 1] != '/')
            strcat(sourcefile, "/");

        strcpy(destfile, destdir);
        if (destfile[strlen(destfile) - 1] != '/')
            strcat(destfile, "/");

        if ((strcmp(direntp -> d_name, ".") != 0)
            && (strcmp(direntp -> d_name, "..") != 0))
        {
            strcat(sourcefile, direntp -> d_name);
            strcat(destfile, direntp -> d_name);
        }
    }
}

```



```
    if (stat(sourcefile, &st) == -1)
    {
        printf("error: stat \n");
        exit(1);
    }

    if ((st.st_mode & S_IFMT) == S_IFDIR)
    {
        dircopy(sourcefile, destfile);
    }
    else
    {
        if ((sourcefd = open(sourcefile, O_RDONLY, 0)) == -1)
        {
            printf("Could not open source file: %s \n", sourcefile);
            exit(1);
        }

        if ((destfd = open(destfile, O_CREAT | O_WRONLY, 0)) == -1)
        {
            printf("Could not create dest file: %s \n", destfile);
            exit(1);
        }

        while ((num = read(sourcefd, buf, 1024)) > 0)
        {
            if (write(destfd, buf, num) != num)
            {
                printf("Could not write dest file: %s \n", destfile);
                exit(1);
            }
        }

        close(sourcefd);
        close(destfd);
    }
}

closedir(dirp);
```



```
int main(int argc, char * * argv)
{

    if (argc != 3)
        usage(argv[0]);

    dircopy(argv[1], argv[2]);

    exit(0);
}
```

4.4 lseek 函数

每个文件都有一个当前位置,它指向当前读取或写入的位置。系统调用 `lseek` 可以用来改变这个位置。当文件刚被打开时,当前位置即为文件的开头;随着读写操作的进行,当前位置也随之改变。

```
#include <fcntl.h>

long lseek (int fd, long offset, int origin);
```

函数 `lseek` 的第 1 个参数 `fd` 为文件描述符。第 2 个参数 `offset` 为偏移量,如为正,表示向后的偏移量;如为负,表示向前的偏移量。第 3 个参数 `origin` 为当前位置的基点,如为 0,则表示基点为文件的开头;如为 1,则表示基点为文件的当前位置;如为 2,则表示基点为文件的尾端。

函数 `lseek` 返回文件的开头到当前读写位置的字节总数;如果返回 -1,则表示调用失败。

对于基点位置,系统在 `unistd.h` 中定义了三个常量, `SEEK_SET`(值为 0), `SEEK_CUR`(值为 1)和 `SEEK_END`(值为 2)。

在 Linux 系统中,有一条叫 `grep` 的 shell 命令,用于在给定文件中搜索给定的字符串。现在给出一个类似于 `grep` 的程序,该程序在给定的文件或目录中搜索给定的字符串,如果某个文件中存在该字符串,则打印该文件名。

```
/* filename: ex4_2.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
```



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

int find = 0;
int inputright = 0;

void usage(char * name)

    printf("Usage: %s charline dname/filename \ n", name);
    exit(1);
:

void filefind(char * charline, char * filename)
{
    int fd;
    char buf[256];
    int count = 0;
    int length;
    long offset = 0;

    length = strlen(charline);
    if ((fd = open(filename, O_RDONLY, 0)) == -1)
        return;
    if (! inputright)
        inputright = 1;

    for (;;)
    {
        if (lseek(fd, offset, 0) == -1)
        {
            printf("error: lseek \ n");
            exit(1);
        }

        if (read(fd, buf, length) < length)
            break;
        buf[length] = ' \ 0';
```



```

        if (strcmp(buf, charline) == 0)
        {
            if (! find)
                find = 1;
            count ++;
        }
        offset ++;
    }
    if (count)
        printf("There are %d \"%s\" in file %s\n", count, charline, filename);

    close(fd);
}

void dirfind(char * charline, char * dirname)
{
    char filename[256];
    DIR * dirp;
    struct dirent * direntp;
    struct stat st;

    if ((dirp = opendir(dirname)) == NULL)
    {
        return ;
    }

    inputright = 1;

    while((direntp = readdir(dirp)) != NULL)
    {
        strcpy(filename, dirname);
        if (filename[strlen(filename) - 1] != '/')
            strcat(filename, "/");

        if ((strcmp(direntp -> d_name, ".") != 0) &&
            (strcmp(direntp -> d_name, "..") != 0))
        {
            strcat(filename, direntp -> d_name);

```




```
        if (stat(filename, &st) == -1)
        {
            printf("error: stat \n");
            exit(1);
        }
        if ((st.st_mode & S_IFMT) == S_IFDIR)
        {
            dirfind(charline, filename);
        }
        else
        {
            filefind(charline, filename);
        }
    }

    closedir(dirp);
}

int main(int argc, char * * argv)
{
    if (argc != 3)
        usage(argv[0]);

    dirfind(argv[1], argv[2]);

    if (! inputright)
        filefind(argv[1], argv[2]);

    if (! inputright)
    {
        printf("Input error \n");
        exit(1);
    }

    if (! find)
        printf("Sorry: Could not find \n");
}
```



```
exit(0);
```

```
|
```

程序的运行结果如下所示,举的例子为在当前目录下搜索字符串“filename”。

```
[root@linux chapter04] # cc ex4_2.c
[root@linux chapter04] # a.out filename ./
There are 1 "filename" in file ./a.out
There are 14 "filename" in file ./ex4_2.c
There are 2 "filename" in file ./ex4_3.c
```

4.5 link 和 unlink 函数

系统调用 link 和 unlink 是用来创建和删除链接的。

在 Linux 系统中,链接就是文件名和索引节点之间的关联。Linux 系统存在两种链接:硬链接和符号链接。目录项为硬链接,因为它将文件名和索引节点直接链接起来;符号链接将文件作为一个指向另一个文件的指针来使用。

目录项对应一个简单的硬链接,而索引节点可能具有若干个这样的链接。每个索引节点都含有该索引节点的硬链接个数,即含有这个索引节点的目录项的个数。操作系统产生一个新的目录项并赋予每个新创建的文件一个索引节点。用户可以通过命令 ln 或系统调用 link 来创建对一个文件的附加硬链接。附加硬链接仅分配目录项而不占有其他磁盘空间,这个新的硬链接会使该索引节点的链接计数项增加 1。硬链接仅有一个目录项。

当用户通过命令 rm 或系统调用 unlink 删除文件时,操作系统将删除相应的目录项并使该索引节点的链接计数减 1。如果这个索引节点的链接计数没有变为 0,操作系统将不释放这个索引节点和相应的数据块。

图 4.1 显示了目录/rootdir 中含有称为 filea 的文件的一个目录项。这个文件使用的索引节点为 11111。此索引节点值有一个链接,且第一个数据块为 22222,所有的文件内容都在这个数据块中。

如果执行命令:ln /rootdir/filea /rootdir/fileb,则在目录项中增加一项,索引节点的链接计数变为 2,但没有增加数据块。如图 4.2 所示。

现在讲解 link 与 unlink 的用法。

```
#include <unistd.h>

int link (const char * from, const char * to);
int unlink (const char * name);
```

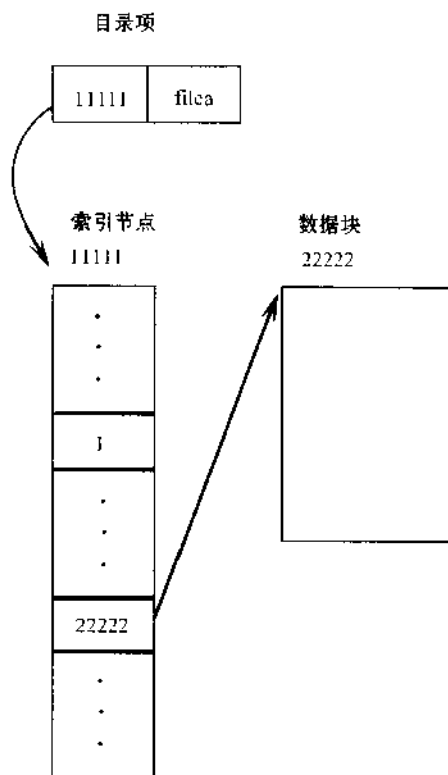


图 4.1 文件的存储

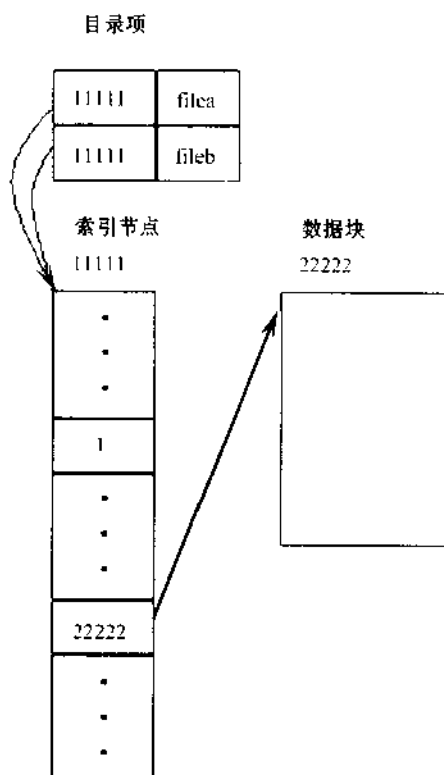


图 4.2 为文件 filea 增加一个链接 fileb

函数 `link` 创建一个链接,名称由函数的第 2 个参数 `to` 给定,该链接指向函数的第 1 个参数 `from`。

函数 `unlink` 删除一个文件。事实上,该函数使索引节点的链接计数减 1,如链接计数变为 0,系统才真正删除数据块。

函数 `link` 与 `unlink` 在调用成功时都返回 0,失败时返回 -1。

4.6 access 函数

函数 `access` 用来检验给定文件的属性,即文件是可读、可写、可执行或是否存在。

```
#include <unistd.h>
int access(const char * name, int type);
```



```
        else
        {
            printf("& Write");
        }

        |
        |
        if (access(argv[1],R_OK) == 0)

        {
            if (! flag)
            {
                flag = 1;
                printf("Read");
            }
            else
            {
                printf("& Read");
            }
        }

        |
        |
        if (! flag)
        {
            printf("Exists");
        }

        |
        |
        printf("\n");
        |
        |
        exit(0);
    }
}
```

下面给出了该程序的运行情况：

```
[msf@linux chapter04] $cc ex4_3.c
[msf@linux chapter04] $a.out /etc/passwd
mode:Execute & Read
[msf@linux chapter04] $a.out /bin/sh
mode:Execute & Read
```

4.7 chmod, chown 和 chdir 函数

函数 chmod, chown 和 chdir 都非常简单。函数 chmod 在 sys/stat.h 中定义, 函数 chown 和 chdir 在unistd.h 中定义。



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

int chmod (monst char * file, mode_t mode);
int chown (const char * file, uid_t owner, gid_t group);
int chdir (const char * path);
```

函数 `chmod` 用来改变文件的存储权限,第 1 个参数 `file` 为文件名,第 2 个参数 `mode` 为存储权限。函数 `chown` 用来改变文件的用户,第 1 个参数 `file` 为文件名,第 2 个参数 `owner` 为用户的 UID,第 3 个参数 `group` 为用户的 GID。函数 `chdir` 用来改变当前工作目录,参数 `path` 为新的工作目录名称。

这三个参数在调用成功时都返回 0,失败时返回 -1。

4.8 mkdir 和 rmdir 函数

函数 `mkdir` 用来创建新的目录,`rmdir` 用来删除空目录。函数 `mkdir` 在文件 `sys/stat.h` 中定义,函数 `rmdir` 在文件 `unistd.h` 中定义。

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

int mkdir (const char * path, mode_t mode);
int rmdir (const char * path);
```

函数 `mkdir` 的两个参数分别为要创建的目录名和存储权限,函数 `rmdir` 的参数为要删除的目录名。这两个函数在调用成功时返回 0,失败时返回 -1。

4.9 mknod 函数

Linux 系统中的所有文件,包括普通文件、目录、命名管道和设备文件都可以调用 `mknod` 建立。

```
#include <sys/stat.h>
int mknod (const char * path, mode_t mode, dev_t dev);
```

函数 `mknod` 的第 1 个参数 `path` 为要建立文件或目录的路径名。第 2 个参数为类型,它是两部分的组合,第一部分可以为 `S_IFREG`、`S_IFDIR`、`S_IFIFO`、`S_IFCHR` 或 `S_IFBLK`,分别代表普通文件、目录、命名管道、字符设备和块设备,第二部分为存储权限。函数 `mknod` 的最后一个参数只有在创建设备文件时才用到,高字节为主设备号,低字节为从设备



号。函数调用成功时返回 0,失败时返回 -1。

对于一般用户来说,只能调用 `mknod` 创建一般文件或命名管道,因为用 `mknod` 创建目录和设备文件需要有超级用户的权限。

例如:可以这样创建目录 `/dir`:

```
mknod(/dir,S_IFDIR|0755,0);
```

4.10 dup 和 dup2 函数

函数 `dup` 和 `dup2` 可以用来复制一个现存的文件描述符。通常,这可以用来达到输入输出重定向的目的。

```
#include <unistd.h>

int dup (int fd);
int dup2 (int fd, int fd2);
```

函数 `dup` 是将输入输出到最近关闭的文件描述符的数据重定向到文件描述符 `fd` 上。函数 `dup2` 是将输入输出到文件描述符 `fd2` 的数据重定向到文件描述符 `fd` 上。这两个函数调用成功时返回 0,失败时返回 -1。

函数 `dup2` 可以很容易地由 `dup` 实现。如下所示:

```
int dup2 (int fd, int fd2)
{
    close (fd2);
    return dup (fd);
}
```

下面的简单程序说明了 `dup` 的用法:

```
/* filename:ex4_4.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

void usage(char * name)
{
    printf("Usage: %s newfile \n", name);
    exit(1);
}
```



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

{
    int fd;

    if (argc != 2)
        usage(argv[0]);

    if ((fd = open(argv[1], O_WRONLY | O_CREAT, 0644)) == -1)

        printf("error: open \n");
        exit(1);

    printf("This line go to screen \n");

    close(STDOUT_FILENO);
    if (dup(fd) == -1)

        printf("error: dup \n");
        exit(1);

    printf("This line go to %s \n", argv[1]);
    close(fd);
    exit(0);
}
```

程序的运行结果为：

```
msf@linux chapter04] $cc ex4_4.c
[msf@linux chapter04] $a.out filea
This line go to screen
msf@linux chapter04] $cat filea
This line go to filea
```

4.11 小 结

本章主要讲解了 Linux 文件系统常用的系统调用。通过了解文件系统的系统调用,可以了解 Linux 文件系统的工作原理,这对于 Linux 系统开发人员是很重要的。

第5章 Linux 设备文件

5.1 设备文件简介

在 Linux 系统中,设备也被看作文件,称为设备文件。因此,对设备的读写操作也就是对文件的读写操作,这样就减轻了程序员的工作量。

设备可以分为两类:字符设备和块设备。

字符设备以字符中的方式传递信息,不考虑任何块信息。它不用寻址,并且没有任何查找操作。

块设备将信息存储在定长的块中,每一个块都有地址。块设备的基本特点是能够读写每一块而与所有别的块无关。

终端、网络接口与鼠标等通常为字符设备,而软盘、硬盘等为块设备。

每个设备文件都有一对主从设备号。主设备号表示设备类型,而从设备号表示具体的设备。如 `/dev/ttyh01` 和 `/dev/ttyh02` 是同一终端控制器上的两个接口。

5.2 设备文件的创建

设备文件的创建只能通过调用函数 `mknod` 来实现。关于 `mknod` 的用法,在前面的章节已经讨论过,注意,只有超级用户操能够创建设备文件。假如要创建一个字符设备文件 `/dev/cdev1`,其主设备号位 21,从设备号为 35,则可以如下调用 `mknod`:

```
mknod("/dev/cdev1",S_IFCHR|0644,(21<<8)|35);
```

5.3 终端设备文件

5.3.1 终端设备文件的读写

对终端设备文件的操作主要是 `read` 和 `write`,可以先调用 `open` 打开终端设备文件,然后调用 `read` 和 `write` 进行读写终端。但是一般不这样做,因为这样的程序依赖于具体的终端设备。在实际应用中,都是利用 `STDIN_FILENO`,`STDOUT_FILENO` 和 `STDERR_FILENO` 三个文件描述符来进行终端文件的读写的。

函数 `read` 在读终端时,与普通文件有所不同。对普通文件的读操作总是返回读入的实际字节数,除非已经读到文件尾,否则它与要读的字符数相同。而读终端时,所读入的字符



数则和要求读入的字符数,当前输入行的起读位置及当前行的长度有关。一次最多读入的字符数为起始位置到行尾间的字符数。如读的是用户正在终端键入行中的字符,则通常只有等待输入完整行后,才能进行读操作。

函数 `write` 对终端写的信息直接显示在终端屏幕上,显示的位置为当前的光标位置。

下面的程序就是调用 `read` 从终端读取一行,然后调用 `write` 写回终端。

```
/* filename:ex5_1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int echoline(char *s,int maxsize)
{
    int count=0;
    char c;

    while(1)
    {
        switch(read(STDIN_FILENO, &c,1))
        {
            case -1:
                printf("error:read \n");
                exit(1);
            case 0:
                return 0;
            default:
                if (count>=maxsize)
                {
                    printf("Too long \n");
                    exit(1);
                }
                if (c == '\n')
                {
                    s[count]='\0';
                    write(STDOUT_FILENO,s,count);
                    return 1;
                }
                s[count]=c;
                count++;
        }
    }
}
```



```
int main()  
{  
    char linebuf[256];  
  
    printf("Input a line: \n");  
    if (! echoline(linebuf,256))  
    {  
        printf("error:echoline \n");  
        exit(1);  
    }  
    exit(0);  
}
```

下面给出了程序的运行情况。

```
[msf@linux chapter05] $cc ex5_1.c  
[msf@linux chapter05] # a.out  
Input a line:  
12345abcde  
12345abcde
```

5.3.2 终端设备文件的控制

函数 `ioctl` 可以对终端进行状态的设置和控制。控制终端的 `ioctl` 调用有两种形式：普通的调用形式和扩展的调用形式。普通形式如下所示：

```
#include <sys/ioctl.h>  
#include <termio.h>  
  
int ioctl (int fd, int request, struct termio * tbuf);
```

扩展形式如下所示：

```
#include <sys/ioctl.h>  
int ioctl(int fd, int request, int arg);
```

扩展调用形式的 `ioctl` 的第 2 个参数 `request` 为控制命令，它和第 3 个参数 `arg` 配合使用，由 `request` 参数解释 `arg` 参数的含义。参数 `request` 的取值如下所示：



- TCSBRK:等待输出队列的信息显示完。如 arg 为 0,则输出队列为空时发送一个中断信号。
- TCXONC:控制中断驱动程序的启动和停止。当 arg 为 0 时,输出暂时被悬挂起来;当 arg 为 1 时,重新启动被悬挂的输出。
- TCFLSH:如 arg 为 0,则刷新输入队列,如 arg 为 1,则刷新输出队列;如 arg 为 2,则同时刷新输入和输出队列。

普通调用形式的 ioctl 的第 2 个参数 request 为控制命令或重置的终端状态。参数 request 的取值如表 5.1 所示。

表 5.1 参数 request 的取值及其含义

取 值	含 义
TCGETA	获取 fd 指定终端的信息并存入 tbuf 中
TCSETA	把 tbuf 中的内容设置为终端的状态信息
TCSETAW	完成与 TCSETA 同样的功能,但必须等待到输出队列为空时,才设置新的终端状态。当重置终端状态会影响当前的输出内容时,应使用此命令
TCSETAF	完成与 TCSETA 相同的功能,但必须等待到当前输出队列为空,刷新输入队列后,才设置新的终端状态。当开始一个新的终端交互方式时,一般使用此命令

普通调用形式的 ioctl 的第 3 个参数为指向 termio 结构的指针。结构 termio 的定义如下所示:

```
struct termio
{
    unsigned short int c_iflag;    /* input mode flags */
    unsigned short int c_oflag;    /* output mode flags */
    unsigned short int c_cflag;    /* control mode flags */
    unsigned short int c_lflag;    /* local mode flags */
    unsigned char c_line;          /* line discipline */
    unsigned char c_cc[NCC];      /* control characters */
};
```

下面讨论结构 termio 的各个字段的功能。

(1) c_iflag 定义了输入控制方式,可以为表 5.2 中列出的值或它们的组合。

表 5.2 c_iflag 的取值及含义

值	含 义
IGNBRK	忽略所有的 break 键
BRKINT	对 break 键产生一个 INTR 信号
IGNPAR	忽略奇偶错字符



(续)

值	含 义
PARMRK	标识奇偶偶
INPCK	激活输入奇偶校验
ISTRIP	除去字符最高位
INCR	在输入中将换行符改为回车符
IGNCR	忽略回车符
ICRNL	在输入中将回车符改为换行符
ULCLC	在输入中把大写变为小写
IXON	允许启动/停止输出控制
IXANY	任何字符都能重新启动输出
IXOFF	允许启动/停止输入控制

(2) `c_oflag` 定义了输出控制方式,可以为表 5.3 中列出的值或它们的组合

表 5.3 `c_oflag` 的取值及含义

值	含 义
OPOST	处理后输出
OLCUC	在输出时将小写改为大写
ONLCR	在输出时将换行符映射为回车符和换行符
OCRNL	在输出时将回车符映射为换行符
ONOCR	不输出回车符
ONLRET	以换行符取代回车符
OFILL	以填充字符延迟终端的输出,常用于高速终端
OFDEL	以 DEL 作为填充字符
CR0, CR1, CR2, CR3	选择回车延时,CR0 为不延迟,CR1,CR2,CR3 递增
NL0, NL1	选择换行符输出延时,NL0 为不延迟,NL1 延迟 0.1s
TAB0, TAB1, TAB2, TAB3	选择制表符输出延时,TAB0 为不延迟,TAB1,TAB2,TAB3 递增
BS0, BS1	选择退格延时,BS0 为不延迟
FF0, FF1	选择换页延时,FF0 为不延迟
VT0, VT1	选择垂直制表符延时,VT0 为不延迟

(3) `c_cflag` 用于指定终端的硬件设置,如波特率,传送字节长度等 `c_cflag` 可以为表 5.4 中列出的值或它们的组合

表 5.4 `c_cflag` 的取值及含义

值	含 义
B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600	选择波特率,B0 为中断,B50 为 50 波特,B110 为 110 波特,B9600 为 9600 波特等

(续)

值	含 义
CS5,CS6,CS7,CS8	选择传送字节长度,CS5 为 5 位,CS6 为 6 位等
CSTOPB	如设置,则使用两个停止位;否则用一个停止位
CREAD	启动接收器
PARENB	启动奇偶校验
PARODD	如设置,则为奇检验,否则为偶检验
HUPCL	最后关闭时挂起
CLOCAL	本地机或拨号
LOBLK	封锁作业层输出

(4) `c_lflag` 定义一组标志来控制各种不同终端显示数据的功能,这些功能与控制的方式由 `c_line` 的值来决定。`c_lflag` 可以为表 5.5 中列出的值或它们的组合。

表 5.5 `c_lflag` 的取值及含义

值	含 义
ISIG	如为 on,则 INTR,QUIT 和 SUSP 字符会产生信号
ICANON	如为 on,则默认为标准输入模式
XCASE	标准输入队列允许大小写混合模式
ECHO	输入的字符会响应在终端上
ECHOE	将删除字符的操作响应在终端上
ECHONL	响应换行字符
NOFLSH	在中断后不清除输入输出队列的数据
ECHOK	删除字符后会送换行符

(5) `c_line` 定义了对应到 `c_lflag` 中用作控制各种不同终端显示数据功能的标志,这些控制方式即为所谓的线上控制。线上控制为介于硬件驱动程序与 tty 驱动程序间的一组例程。`c_lflag` 定义的功能就是通过这批例程来执行的。

(6) `c_cc` 为定义了一组控制字符的字符数组,实现终端输入是特定键的控制功能。表 5.6 列出的一组符号常数定义控制符在 `c_cc` 数组中的存放位置。

表 5.6 符号常数

符号常数	值	符号常数	值
VINTR	0	VEOL	5
VQUIT	1	VEOL2	6
VERASE	2	VMIN	4
VKILL	3	VTIME	5
VEOF	4	VSWTCH	7



擦除控制字符(ERASE)存放在 `c_cc[2]` 中,而文件结束控制字符存放在 `c_cc[4]` 中由 `c_cc` 定义了一些缺省控制字符如表 5.7 所示。

表 5.7 缺省控制字符

字符	值	键	含 义
CESC	'\'	ESC 键	转义字符
CINTR	0177	DEL 键	按下此键产生信号 SIGINT
CQUIT	034	ESC 键	按下此键产生信号 SIGQUIT
CERASE	'#'	# 键	按下此键擦除本行先前键入的字符
CKILL	'@'	@ 键	按下此键删除本行所有内容
CEOF	04	Ctrl-D	按下此键产生一个文件结束符
CSTOP	021	Ctrl-S	按下此键挂起输出
CSTART	023	Ctrl-Q	按下此键恢复挂起的输出
CSWCH	032	Ctrl-Z	终端切换控制键

下面的程序将字符“*”定义为 Del 键,当用户按下“*”键时,产生 SIGINT 信号。这个程序非常简单,只是说明了 `ioctl` 的一般用法。

```
/* filename: ex5_2.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <termio.h>
#include <unistd.h>
#include <signal.h>

void sig_handle(int signo)

{
    printf("Received the signal: SIGINT \n");
    printf("You must have printed \"*\" \n");
    exit(1);
}

int main()
{
    struct termio tbuf;

    if (ioctl(0, TOGETA, &tbuf) == -1)
    {
        printf("error: ioctl \n");
        exit(1);
    }
}
```



```

tbuf.c_cc[VINTR] = ' * ';
if (ioctl(0, TCSETA, &tbuf) == -1)
{
    printf("error: ioctl \n");
    exit(1);
}
signal(SIGINT, sig_handle);
printf("Print \" * \" : \n");
for (;;)

    exit(0);
}

```

程序运行情况如下所示:

```

[msf@linux chapter05] $ cc ex5_2.c
[msf@linux chapter05] $ a.out
Print " * ":
Received the signal: SIGINT
You must have printed " * "

```

读者都还记得,当登录输入密码时,密码并不回显,下面的程序显示了如何实现这一功能。

```

/* filename: ex5_3.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <termio.h>
#include <fcntl.h>

int getline(char *s, int maxsize)
{
    int count = 0;
    char c;

    while(1)
    {
        switch(read(STDIN_FILENO, &c, 1))
        {

```



```
        case -1:
            printf("error:read \n");
            exit(1);
        case 0:
            return 0;
        default:
            if (count >= maxsize)
            {
                printf("Too long \n");
                exit(1);
            }

            if (c == '\n')
            {
                s[count] = '\0';
                return 1;
            }

            s[count] = c;
            count++;
        }
    }

int main()
{
    struct termio tbuf;
    int savedflag;
    char passwd[256];

    if (ioctl(0, TCGETA, &tbuf) == -1)

        printf("error: ioctl \n");
        exit(1);

    savedflag = tbuf.c_lflag;
    tbuf.c_lflag &= ~(ECHO|ECHOE|ECHOK|ECHONL);
    if (ioctl(0, TCSETA, &tbuf) == -1)

        printf("error: ioctl \n");
```




```
/* filename: ex5_4.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/errno.h>
#include <unistd.h>
#include <fcntl.h>
main()
{
    int nread, nwrite;
    int rfd, wfd;
    char buf[40 * 1024];

    printf("Input the src disk and press ENTER:");
    getchar();
    if ((rfd = open("/dev/fd0", O_RDONLY, 0)) == -1 ||
        (wfd = open("/tmp/floppy.tmp", O_WRONLY | O_CREAT, 0)) == -1)
    {
        switch(errno)
        {
            case EBUSY:
                printf("the device /dev/fd0 is used by others \n");
                exit(1);
            case EIO:
                printf("the disk does not exist or door is not closed \n");
                exit(1);
            case ENXIO:
                printf("disk format does not support \n");
                exit(1);
            default:
                printf("error: open \n");
                exit(1);
        }
    }

    while((nread = read(rfd, buf, 40 * 1024)) >= 0)
    {
        if (nread == 0)
            break;
    }
}
```



```
        break;
    if ((nwrite = write(wfd, buf, nread)) == -1)
    {
        if (errno == ENXIO)
        {
            printf("dest disk out of space \n");
            exit(1);
        }
        else
        {
            printf("error; write \n");
            exit(1);
        }
    }

    close(rfd);
    close(wfd);
    printf("Disk copy OK! \n");
    unlink("/tmp/floppy.tmp");
    exit(0);
}
```

程序的运行情况如下所示:

```
[msf@linux chapter05] $ cc ex5_4.c
[msf@linux chapter05] $ a.out
Input the src disk and press ENTER;
Input the dest disk and press ENTER;
Disk copy OK!
```

5.4.2 软盘的外挂和 sync 函数

在调用 open、read 和 write 读写软盘时,很难对软盘上的文件进行读写。如果要对软盘上的文件进行读写,必须把软盘外挂进系统,然后操作软盘就像操作某一目录一样。

外挂软盘的函数为 mount,卸载软盘的函数为 umount

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

int mount(const char * special_file, const char * dir,
          char * filesystem, int mode, char * data);

int umount(const char * dir);
```



函数 `mount` 的第 1 个参数 `special_file` 为设备文件名,软盘多为 `/dev/fd0`。第 2 个参数 `dir` 为软盘外挂的目录,这个目录必须为空。第 3 个参数为文件系统名称,如“`msdos`”第 4 个参数为读写模式,1 代表只读,其余代表可读可写。最后一个参数为数据,一般设为 `NULL`。函数 `umount` 的参数为挂软盘的目录名。这两个函数在调用成功时返回 0,失败时返回 -1。

注意 只有超级用户才能够外挂软盘。

Linux 系统开辟了大量的缓冲区用于文件读写操作,这样就会导致主存映像和外部存储介质数据的不一致,为了解决这个问题,Linux 系统提供了系统调用 `sync`,用于强制将主存中应写到磁盘的数据立即写到相应的介质上,以保证内外数据的一致性。

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

```
int sync(void);
```

下面的程序用来说明 `mount`、`umount` 和 `sync` 的用法,程序的参数为软盘外挂的目录,注意此目录必须为空。

```
/* filename: ex5_5.c */
```

```
#include <stdio.h>
```

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

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

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

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

```
#include <fcntl.h>
```

```
char spe_file[20] = "/dev/fd0";
```

```
char cur_dir[256];
```

```
void usage(char * name)
```

```
{
```

```
    printf("Usage: %s dirname \n", name);
```

```
    exit(1);
```

```
}
```

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

```
{
```

```
    int fd;
```

```
    char c;
```

```
    int result;
```



```
        exit(1);
    }

    while ((result = read(fd, &c, 1)) > 0)
    {
        if (write(STDOUT_FILENO, &c, 1) == -1)
        {
            printf("error: write \n");
            exit(1);
        }
    }

    printf("\n");
    if (result < 0)
    {
        printf("error: read \n");
        exit(1);
    }

    close(fd);
    unlink("tmpfile.tmp");
    chdir(cur_dir);
    umount(argv[1]);
    exit(0);
}
```

程序的运行情况如下所示:

```
[root@linux chapter05] # cc ex5_5.c
[root@linux chapter05] # a.out /floppy
Input some characters, end with ENTER: I Love Linux!
This is in the file: I Love Linux!
```

5.5 小 结

本章讨论了 Linux 系统的一类特殊文件——设备文件,讨论了创建设备文件的函数 `mknod` 和控制设备文件的函数 `ioctl`。同时还重点讲解了两种设备文件——终端和软盘的读写和控制。



这意味着系统中的最大进程数目受 task 数组大小的限制,缺省值一般为 512。创建新进程时,Linux 将从系统内存中分配一个 task_struct 结构并将其加入 task 数组。当前运行进程的结构用 current 指针来指示。

下面给出了 task_struct 结构的定义:

```
struct task_struct {
    /* these are hardcoded - don't touch */
    volatile long    state;    /* -1 unrunnable, 0 runnable, > 0 stopped */
    long             counter;
    long             priority;
    unsigned         long signal;
    unsigned         long blocked; /* bitmap of masked signals */
    unsigned         long flags; /* per process flags, defined below */
    int              errno;
    int              pid;
    int              pgrp;
    int              tty_old_pgrp;
    int              session;
    /* boolean value for session group leader */
    int              leader;
    int              groups[NGROUPS];
    /*
    unsigned short    uid, euid, suid, fsuid;
    unsigned short    gid, egid, sgid, fsgid;
    unsigned long     timeout, policy, rt_priority;
    ...
    ...
    ...
};
```

6.1.2 进程状态

由 task_struct 结构的定义可知,task_struct 非常复杂,事实上,没有必要弄清楚每一个字段的含义,只要了解几个与本书内容有关的字段的含义。首先看 state 字段,顾名思义,这是表示进程的运行状态。Linux 中进程的运行状态如表 6.1 所示。

表 6.1 进程运行状态

state	进 程 的 状 态 描 述
running	表示进程处于运行或者准备运行状态



(续)

state	进 程 的 状 态 描 述
waiting	表示进程在等待一个事件或者资源。Linux 将等待进程分成两类:可中断与不可中断。可中断等待进程可以被信号中断;不可中断等待进程直接在硬件条件下等待,并且任何情况下都不可中断。
stopped	表示进程被停止,通常是通过接收一个信号来终止进程。正在被测试的进程可能处于停止状态。
zombie	表示进程由于某些原因被终止,但是在 task 数组中仍然保留 task_struct 结构。它像一个已经死亡的进程。

6.1.3 进程标识

Linux 系统使用用户和组标志符来检查对系统中文件和可执行映像的访问权限。Linux 系统中所有的文件都有所有者和允许的权限,这些权限描述了系统使用者对文件或者目录的使用权。基本的权限是读、写和可执行,这些权限被分配给三类用户:文件的所有者、属于相同组的进程以及系统中所有进程。每类用户具有不同的权限,例如一个文件允许其拥有者读写,但是同组的只能读而其他进程不允许访问。

Linux 使用组将文件和目录的访问特权授予一组用户,而不是单个用户或者系统中所有进程。如可以为某个软件项目中的所有用户创建一个组,并将其权限设置成只有他们才允许读写项目中的源代码。一个进程可以同时属于多个组(最多为 32 个),这些组都被放在进程的 task_struct 中的 group 数组中。只要某组进程可以存取某个文件,则由此组派生出的进程对这个文件有相应的组访问权限。

task_struct 结构中有四对进程和组标志符,如表 6.2 所示。

表 6.2 进程和组标识符

进程和组标志符	含 义
uid 和 gid	表示运行进程的用户标志符和组标志符
euid 和 egid	表示有效的用户标识符和组标识符,有些程序可以在执行过程中将执行进程的 uid 和 gid 改成其程序自身的 uid 和 gid。当进程试图访问特权数据或代码时,核心将检查进程的 egid 和 euid。
fsuid 和 fsgid	与 euid 和 egid 相似,一般用来检验进程的文件系统访问权限。
suid 和 sgid	这两个标识符被那些通过系统调用改变进程 uid 和 gid 的程序使用。当进程的原始 uid 和 gid 变化时,它们被用来保存真正的 uid 和 gid。

6.1.4 进程调度

在 Linux 系统中,进程有两种运行模式:用户模式和系统模式。用户模式的权限比系统模式下的小得多。对于一般的进程,都是部分时间运行于用户模式,部分时间运行于系统模式。进程通过系统调用在这两种模式之间切换。当系统调用发生时,进程将由用户模式切换到系统模式继续执行;当系统调用返回时,进程将由系统模式切换回用户模式。

在 Linux 系统中,进程不能被抢占。只要能够运行它们就不会被停止。当进程必须等待某个系统事件时,它才决定释放出 CPU。进程常因为执行系统调用而需要等待。由于处于等待状态的进程还可能占用 CPU 时间,所以 Linux 采用了预加载调度策略。在此策略



中,每个进程只允许运行很短的时间(200ms),当这个时间用完之后,系统将选择另一个进程来运行,原来的进程必须等待一段时间以继续运行。这段时间称为时间片。

可运行进程是一个只等待 CPU 资源的进程。Linux 使用基于优先级的简单调度算法来选择下一个运行进程。当选定新进程后,系统必须将当前进程的状态、处理器中的寄存器以及上下文状态保存到 `task_struct` 结构中。同时它将重新设置新进程的状态并将系统控制权交给此进程。为了将 CPU 时间合理地分配给系统中每个可执行进程,调度管理器必须将这些时间信息也保存在 `task_struct` 中。

在 `task_struct` 结构中保存的调度信息如表 6.3 所示。

表 6.3 进程调度信息

字段名	含 义
<code>policy</code>	该字段表示了进程的调度策略。系统中有两类进程:普通与实时进程。实时进程的优先级要高于普通进程。实时进程也有两种策略:时间片轮转和先进先出
<code>priority</code>	该字段表示了进程的优先级,同时它也就是进程允许运行的时间
<code>rt_priority</code>	该字段表示了实时进程的相对优先级
<code>counter</code>	该字段表示了进程允许运行的时间。进程首次运行时为进程优先级的数值,它随时间变化递减

6.1.5 Linux 进程调度算法

系统在运行队列中选择一个最迫切需要运行的进程。如果运行队列中存在实时进程,则它们比普通进程具有更多的优先级权值,因为普通进程的权值是它的 `counter` 值,而实时进程则是 `counter` 值加上 1000。这表明如果系统中存在可运行的实时进程,它们将总是在普通进程之前运行。如果系统中存在和当前进程相同优先级的其他进程,这时当前运行进程已经用掉了一些时间片,所以它将处在不利形势(原因是它的 `counter` 值已经变小);而原来优先级与它相同的进程的 `counter` 值显然比它大,这样位于运行队列中最前面的进程将开始执行而当前进程被放回到运行队列中。在存在多个相同优先级进程的平衡系统中,每个进程被依次执行。

6.2 Linux 进程系统调用

6.2.1 fork 与 vfork 函数

在 Linux 系统中,有两个函数可以产生新的进程:fork 与 vfork。这两个函数都没有参数,返回 0 到子进程,返回子进程号到父进程,如返回 -1,表示出错。

```
#include <unistd.h>

pid_t fork(void);
pid_t vfork(void);
```



对于 fork,则子进程得到父进程的一个拷贝,由于工作量很大,因此导致速度较慢。对于 vfork,假设子进程在调用 vfork 后立即调用 exec,因此子进程并不拷贝父进程的页面,只是初始化自用的数据结构与准备足够的页面表,所以执行速度大为加快,但这样做有潜在的危险,因为在 vfork 之后父子进程实际上是共享同一块存储区(除非子进程调用了 exec 或 exit),因此子进程能改动父进程的数据和堆栈的信息,所以 vfork 的使用必须非常小心。另外父进程将一直处于阻塞状态直到子进程调用 exit 退出或调用 exec 执行新的程序。

下面的程序给出了 fork 的用法:

```
/* filename:ex6_1.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HMASK 0x0000ff00
#define LMASK 0x000000ff
#define HBYTE(x) ((x & HMASK) > > 8)
#define LByte(x) (x & LMASK)

int main(int argc, char * * argv)
{
    int pid, status;

    printf("Before fork() system call \n");
    printf("The process ID is %d \n", getpid());
    if ((pid = fork()) == -1)
    {
        printf("error: fork \n");
        exit(1);
    }
    else if (pid == 0)
    {
        printf("After fork() system call \n");
        printf("This is child process, ID is %d \n", getpid());
        exit(1);
    }
    else
    {
        wait(&status);
        printf("This is parent process, ID is %d, child process ID is %d \n",
```




```

        getpid(),pid);
    if (LBYTE(status) == 0)
        printf("The exit code of child process is %d \n",HBYTE(status));
    }
    exit(0);
}

```

在该程序中,wait 为等待子进程结束的系统调用,关于它的用法,在下面的章节会讲解。当程序执行到 fork 语句时,程序就产生了两个分枝,子进程与父进程。fork 返回为 0 的为子进程,返回正整数的为父进程。为了让子进程先执行,在父进程一开始时就调用 wait 等待子进程的结束,如果没有 wait 调用,则子进程与父进程的执行顺序是随机的。

下面给出了该程序的运行结果:

```

[msf@linux chapter06] $ cc ex6_1.c
[msf@linux chapter06] $ a.out
Before fork() system call
The process ID is 1446
After fork() system call
This is child process, ID is 1447
This is parent process, ID is 1446, child process ID is 1447
The exit code of child process is 1

```

为了掌握 fork 与 vfork 的区别,再举一个例子。这个程序有两种执行方式,调用 fork 或调用 vfork,通过给定的参数来设置。当选择 fork 时,由于子进程是父进程的拷贝,所以子进程中参数的改变不影响父进程;而当选择 vfork 时,由于子进程与父进程共享数据,所以子进程中参数的改变会影响到父进程。

```

/* filename: ex6_2.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void usage(char * name)
{
    printf("%s fork/vfork \n", name);
    exit(1);
}

int main(int argc, char * * argv)
{
    int a;

```



```
int mode;

if (argc != 2)
    usage(argv[0]);

if ((strcmp(argv[1], "fork") != 0) && (strcmp(argv[1], "vfork") != 0))
    usage(argv[0]);
if (strcmp(argv[1], "fork") == 0)
    mode = 0;
else
    mode = 1;
a = 0;
printf("In father process, a = %d \n", a);
if (mode == 0)
{
    printf("You choosed FORK \n");
    if (fork() == 0)
    {
        a = 1;
        printf("In child process, change a to %d \n", a);
        exit(0);
    }
    else
    {
        wait(NULL);
        printf("In father process, a = %d \n", a);
        exit(0);
    }
}
else
{
    printf("You choosed VFORK \n");
    if (vfork() == 0)
    {
        a = 1;
        printf("In child process, change a to %d \n", a);
        exit(0);
    }
}
```



```
        else
        {
            printf("In father process,a= %d \n",a);
            exit(0);
        }
    }
}
```

程序的运行结果如下所示:

```
[msf@linux chapter06] # cc ex6_2.c
[msf@linux chapter06] # a.out fork
In father process,a=0
You choosed FORK
In child process,change a to 1
In father process,a=0
[msf@linux chapter06] # a.out vfork
In father process,a=0
You choosed VFORK
In child process,change a to 1
In father process,a=1
```

6.2.2 exec 函数

为了能够在一个程序中执行另一个程序,系统提供了 exec 函数,由于 exec 是采用覆盖旧进程的内存的方式,因此原来程序的堆栈、数据段和程序段都会被修改,只是用户区域维持不变。所以一旦新程序被执行,原来的程序将一去不复返。但有一点值得注意,就是进程标识符保持不变。

exec 系统调用共有 6 个,如下所示:

```
#include <unistd.h>

int execl(const char *path, char *argv[]);
int execlp(const char *path, const char *arg,...);
int execl(const char *path, const char *arg,...);
int execvp(const char *file, char *argv[]);
int execlp(const char *file, const char *arg,...);
int execve(const char *path, char *argv[], char *envp[]);
```

在这些函数中,path 为被执行程序的完整路径名;file 为被执行程序的文件名;arg 为一组传给新程序的参数,以 NULL 结束;argv[]及 envp[]与 main 函数中的 argv[]和 envp[]相同。



下面给出了一个调用 `execvp` 的例子,这是一个简单的 shell,它能处理一些简单的命令

在该程序中,有一个非常有用的函数 `makeargv`,它能够根据一个字符串创建一个参数数组。该函数的第一个参数为指向该字符串的指针;第二个参数为分隔符,一般为空格;第三个参数为指向参数数组的指针,为 `char ***` 类型。函数返回参数数组中参数的个数,如返回 `-1`,表示出错。

```
/* filename:ex6_3.c */
#include <string.h>
#include <stdlib.h>

int makeargv(char * s,char * delimits,char *** argvp)
{
    char * t;
    char * snew;
    int numtokens;
    int i;

    snew = s + strspn(s,delimits);
    if ((t = calloc(strlen(snew) + 1, sizeof(char))) == NULL)
    {
        * argvp = NULL;
        numtokens = -1;
    }
    else
    {
        strcpy(t,snew);
        if (strtok(t,delimits) == NULL)
        {
            numtokens = 0;
        }
        else
        {
            for (numtokens = 1; strtok(NULL,delimits) != NULL; numtokens++);
        }
        if (( * argvp = calloc(numtokens + 1, sizeof(char *))) == NULL)
        {
            free(t);
            numtokens = -1;
        }
    }
}
```



该程序的执行情况如下所示:

6.2.3 exit 与 _exit 函数

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

void exit (int status);
void _exit (int status);
```

6.2.4 wait 与 waitpid 函数

在 `waitpid` 中,第 1 个参数 `pid` 为等待的子进程的标识符,如为 `-1`,则表示等待第 1 个终止的子进程;第 2 个参数与 `wait` 中的参数相同;第 3 个参数为选项,如设置了 `WNOHANG`,则当子进程还没有终止时, `waitpid` 不像 `wait` 那样阻塞,而是返回 0;如设置了 `WUNTRACED`,则返回停止的子进程的状态。



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

```
pid_t wait (int *stat_loc);
```

```
pid_t waitpid (pid_t pid, int *stat_loc, int options);
```

调用 `wait` 或 `waitpid` 后,得到了包含子进程退出值的整数。该整数与传给 `exit` 的进程退出值之间的关系如图 6.2 所示。

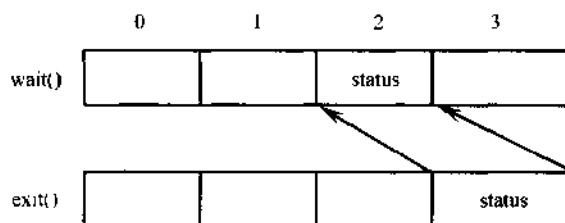


图 6.2 `wait` 与 `exit` 的关系

6.3 小 结

在这一章中,讲解了操作系统中的最重要的一个概念——进程。介绍了 Linux 系统中进程的核心数据结构 `task_struct`,进程的调度算法以及 Linux 系统中关于进程的系统调用。重点介绍了 `fork`、`vfork` 以及 `exec` 等系统调用的用法并给出了相应的例子。

第7章 信号处理

7.1 信号简介

信号是进程间通信的方法之一,它用以指示某些事件的发生。信号可以由系统核心程序发出,也可由某些进程发送,但是大部分的时候都是由核心程序发出的。

系统核心程序只有在下面几种情况下才会向进程发送信号:

- 程序有异常的行为发生,例如企图除以零。
- 系统测出一个可能将出现的电源故障。
- 该进程的子进程执行终止。
- 用户由终端对目标进程输入中断(Break),删除>Delete),暂停(Ctrl-S),继续(Ctrl-Q)或停止(Ctrl-\)键。

进程在接收到信号后,处理的方式有五种:一种为忽略这个信号;一种为执行处理该信号的函数;还有的是暂停进程的执行;也有的是重新启动刚才被暂停的进程;最常见的是采用系统默认的行动。大部分信号的默认操作都是终止进程的执行。有些信号除了会终止进程的执行,还会留下一个叫 core 的文件。这个文件存有进程当时在主存中的内容,通常用于以后查错。

7.2 信号类别

在 Linux 中,常见信号的代码和表示的意义如表 7.1 所示。

表 7.1 常见的信号

信号代码	信号名称	信号表示的意义
1	SIGHUP	挂断控制终端,当一个终端被切断时,核心程序就将此信号传给该终端所控制的一切进程
2	SIGINT	控制终端键盘的中断键被按下,该中断键一般为 Ctrl-C
3	SIGQUIT	控制终端的停止键被按下,该停止键一般为 Ctrl-\
4	SIGILL	不正确的硬件指令,应用程序通常会捕获该信号以响应程序执行的错误
5	SIGTRAP	程序跟踪中断点,这是给调试程序专用的信号
7	SIGBUS	总线故障
8	SIGFPE	浮点运算异常,例如除以零或者溢位
9	SIGKILL	删除一个或一组进程,该信号不能忽略
10	SIGUSR1	应用程序自己定义的信号



(续)

信号代码	信号名称	信号表示的意义
11	SIGSEGV	存储器存取错误(通常是指进程引用了其正常地址范围之外的地址)
12	SIGUSR2	应用程序自己定义的信号
13	SIGPIPE	一个进程不停地向管道写入数据却没有进程将数据读出,也就是写进程异常终止
14	SIGALRM	时钟信号,用于测量进程的真实运行时间,注意不是 CPU 时间
15	SIGTERM	软件结束
17	SIGCHLD	子进程终止或暂停时,系统发送此信号给父进程,该信号的默认处理方式忽略
18	SIGCONT	在信号被暂停后重新启动,该信号不能忽略,它通常与 SIGTSTP 配合使用。假设此信号送往的进程未被暂停,则此信号将被忽略。该信号的默认处理方式忽略
19	SIGSTOP	将停止信号发送给在后台执行的程序
20	SIGTSTP	将停止信号发送给联机会话的进程,该信号通常由 Ctrl-Z 产生
21	SIGTTIN	后台执行的进程要从控制终端读取数据
22	SIGTTOU	后台执行的进程要向控制终端写入数据
23	SIGURG	此信号用于通知应用程序有立刻要处理的情况,该信号的默认处理方式忽略
24	SIGXCPU	进程超出了所设定给它的最大 CPU 使用时间
25	SIGXFSZ	进程超出了所设定给它的最大文件极限
26	SIGVTALRM	用以测量进程的虚拟时间,即实际被执行进程的时间
27	SIGPROF	用以测量进程的概括时间,即被核心程序执行的实际时间
28	SIGWINCH	窗口的大小被改变,该信号的默认处理方式忽略
29	SIGIO	指示进程可以对某个文件描述符进行读写
30	SIGPWR	电源故障,该信号的默认处理方式忽略

7.3 关于信号的系统调用

7.3.1 kill 命令及 kill 函数

kill 既是 shell 命令,又是系统调用。利用 kill 命令可以从 shell 产生信号,然后以可将该信号发送给指定的应用程序。kill 命令的使用如下所示:

```
kill -s signal pid ...
kill -l [exit_status]
kill [-signal] pid ...
```

利用下面的命令可以将信号 SIGUSR1 发送给进程号为 500 的进程:

```
kill -USR1 500
```

同样,利用下面的命令可以列出系统的所有信号:

```
kill -l
```




系统输出如下所示：

```
[msf@linux chapter07] $ kill -l
1)SIGHUP      2)SIGINT      3)SIGQUIT     4)SIGILL
5)SIGTRAP     6)SIGABRT    7)SIGBUS      8)SIGFPE
9)SIGKILL     10)SIGUSR1   11)SIGSEGV    12)SIGUSR2
13)SIGPIPE    14)SIGALRM   15)SIGTERM    17)SIGCHLD
18)SIGCONT    19)SIGSTOP   20)SIGSTP     21)SIGTTIN
22)SIGTTOU    23)SIGURG    24)SIGXCPU    25)SIGXFSZ
26)SIGVTALRM  27)SIGPROF   28)SIGWINCH   29)SIGIO
30)SIGPWR
```

系统调用 `kill` 完成同样的功能，也是将一个信号发送给某一进程。该函数返回 0 表示调用成功；返回 -1 表示调用失败，并且设定 `errno`（错误号）。函数的第 1 个参数为进程号，第 2 个参数为信号代码。

```
#include <signal.h>
int kill (pid_t pid,int signo);
```

7.3.2 有关信号集合的调用

进程能够利用阻塞信号来暂时阻止信号的释放。阻塞信号在其被释放前不会影响进程的行为，信号屏蔽进程显示出当前为阻塞状态的信号集合。信号屏蔽的结构类型为 `sigset_t`。

阻塞一个信号与忽略一个信号是不同的。当进程阻塞一个信号时，此信号将被挂起直到进程取消阻塞，进程使用 `sigprocmask` 函数改变信号屏蔽来阻塞信号；而当进程忽略一个信号时，进程将释放并抛弃该信号。

下面的信号集合函数指定使用 `sigset_t` 类型的信号集合的信号组的某些特殊操作，如阻塞和取消阻塞。

```
#include <signal.h>

int sigemptyset (sigset_t * set);
int sigfillset (sigset_t * set);
int sigaddset (sigset_t * set, int signo);
int sigdelset (sigset_t * set, int signo);
int sigismember (const sigset_t * set, int signo);
```

在上面的函数中，`sigemptyset` 将信号集合初始化为空，`sigfillset` 将信号集合初始化为包含所有的信号，在使用信号集合时，都要调用其中之一来初始化。

函数 `sigaddset` 将某一信号增加到信号集合中，`sigdelset` 将某一信号从信号集合中删除。



mask 恢复原来的阻塞信号集合。

```
#include <signal.h>
#include <stdio.h>

sigset_t newset, oldset;
int i;
sigfillset(&newset);
printf("Before doing sth \n");
sigprocmask(SIG_SETMASK, &newset, &oldset);
printf("Doing sth \n");
for (i = 0; i < 65535; i++);
sigprocmask(SIG_SETMASK, &oldset, NULL);
```

7.3.3 signal 与 sigaction 函数

上面介绍了很多信号和信号集合的系统调用,却没有介绍信号处理的系统调用。在这一小节中,将介绍两个函数 signal 和 sigaction,这两个函数都是用来处理信号的。

```
#include <signal.h>

void (* signal (int signo, void (func *) (int))(int));
int sigaction (int signo, const struct sigaction
               * act, struct sigaction * oact);
```

函数 signal 无返回值。该函数有两个参数,第 1 个参数 signo 为要处理的信号,第 2 个参数 func 为处理该信号的函数。

函数 sigaction 有三个参数,第 1 个参数 signo 为要处理的信号,后两个参数都为指向 sigaction 结构的指针类型。如果调用成功, sigaction 则返回 0;如果调用失败,则返回 -1。结构 sigaction 的定义如下所示:

```
struct sigaction
{
    void (* sa_handler());
    sigset_t sa_mask;
    int sa_flags;
};
```

结构 sigaction 的 sa_handler 为信号处理函数名, SIG_DFL 表示默认的信号处理, SIG_IGN 表示忽略信号; sa_mask 为阻塞的信号集合; sa_flags 为选项。

函数 sigaction 的第 2 个参数为设置的处理某信号的 sigaction,第 3 个参数为返回的原来的处理该信号的 sigaction。



下面举一个处理信号 SIGUSR1 的程序,它有两个版本,分别用 signal 和 sigaction 实现。
版本 1:

```
/* filename: ex7_1v1.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>

void sig_handle(int signo)
{
    printf("I received the signal: SIGUSR1 \n");
    exit(1);
}

int main()
{
    int i;
    signal(SIGUSR1, sig_handle);
    for (;;)
    {
        for (i=0; i<65535; i++);
    }
}
```

版本 2:

```
/* filename: ex7_1v2.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>

struct sigaction oldsigaction;

void sig_handle(int signo)
{
    printf("I received the signal: SIGUSR1 \n");
    if (sigaction(SIGUSR1, &oldsigaction, NULL) == -1)
    {
    }
}
```



```
        printf("error: sigaction \n");
        exit(1);
    }
    exit(0);
}

int main()
{
    int i;
    struct sigaction newsigaction;

    newsigaction.sa_handler = sig_handle;
    sigemptyset(&newsigaction.sa_mask);
    newsigaction.sa_flags = 0;
    if (sigaction(SIGUSR1, &newsigaction, &oldsigaction) == -1)
    {
        printf("error: sigaction \n");
        exit(1);
    }

    for (;;)
    {
        for (i = 0; i < 65535; i++);
    }
}
```

这两种实现基本上是相同的,首先为信号 SIGUSR1 设置信号处理函数,然后进入一个无限循环。在程序接收到信号 SIGUSR1 后,打印消息后退出应用程序。所不同的是:版本 2 在设置信号处理函数时,保存了原来的信号处理函数,在程序退出前又恢复到原来的信号处理函数;而版本 1 没有这样做。

在运行程序时,为了向进程发送信号 SIGUSR1,利用了前面讲的 kill 命令。这两个程序的运行情况完全相同,如下面所示:

```
[msf@linux chapter07] $cc ex7_1v1.c
[msf@linux chapter07] $a.out
I received the signal: SIGUSR1
```

7.3.4 信号处理的另外一些调用

这里介绍另外一些信号处理的系统调用,有 sigset, sighold, sigignore 和 sigelse 四个。



```
int sig_received = 0;
struct sigaction oldsigaction;

void sig_handle(int signo)
{
    printf("received signal: SIGINT \n");
    if (sig_received == 1)
    {
        if (sigaction(SIGUSR1, &oldsigaction, NULL) == -1)
        {
            printf("error: sigaction \n");
            exit(1);
        }
        printf("program terminated \n");
        exit(0);
    }
    sig_received = 1;
    return;
}

int main()
{
    int i;
    struct sigaction newsigaction;

    newsigaction.sa_handler = sig_handle;
    sigemptyset(&newsigaction.sa_mask);
    newsigaction.sa_flags = 0;
    if (sigaction(SIGINT, &newsigaction, &oldsigaction) == -1)
    {
        printf("error: sigaction \n");
        exit(1);
    }

    printf("press CTRL - C to continue \n");

    while (sig_received == 0)
        pause();
}
```



```
printf("Enter the cycle \ n");
for (;;)
{
    for (i = 0; i < 65535; i++);
}

exit(0);

}
```

程序 2:

```
/* filename: ex7_2v2.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
int sig_received = 0;
struct sigaction oldsigaction;

void sig_handle(int signo)
{
    printf("received signal: SIGINT \ n");
    if (sig_received == 1)
    {
        if (sigaction(SIGUSR1, &oldsigaction, NULL) == -1)
        {
            printf("error: sigaction \ n");
            exit(1);
        }
        printf("program terminated \ n");
        exit(0);
    }
    sig_received = 1;
    return ;
}

int main()
{
    int i;
```



```

sigset_t sigset;
sigset_t sigoldmask;
struct sigaction newsigaction;

newsigaction.sa_handler = sig_handle;
sigemptyset(&newsigaction.sa_mask);
newsigaction.sa_flags = 0;
if (sigaction(SIGINT, &newsigaction, &oldsigset) == -1)
{
    printf("error: sigsetion \n");
    exit(1);
}

sigprocmask(SIG_SETMASK, NULL, &sigoldmask);
sigprocmask(SIG_SETMASK, NULL, &sigset);
sigedddset(&sigset, SIGINT);
sigprocmask(SIG_BLOCK, &sigset, NULL);
sigdelset(&sigset, SIGINT);

printf("press CTRL - C to continue \n");

while (sig_received == 0)
    sigsuspend(&sigset);

sigprocmask(SIG_SETMASK, &sigoldmask, NULL);

printf("Enter the cycle \n");
for (;;)
{
    for (i = 0; i < 65535; i++);
}

exit(0);
}

```

这两个程序的运行情况完全相同,下面给出了程序的运行情况:

```

[msf@linux chapter07] $cc ex7_2v2.c
[msf@linux chapter07] $a.out

```



```
press CTRL - C to continue
```

```
^C
```

```
received signal:SIGINT
```

```
Enter the cycle
```

```
^C
```

```
received signal:SIGINT
```

7.3.6 siglongjmp 与 sigsetjmp 函数

有时候,希望程序在接收到某一信号后将重新从开始位置或程序中的某一特定位置执行。系统调用 sigsetjmp 和 siglongjmp 可以实现这一功能。这种功能经常用于处理程序中某处可能发生的但不是致命的错误。

```
# include < setjmp.h >

int sigsetjmp(sigjmp _ buf env, int savemask);
void siglongjmp (sigjmp _ buf env, int val);
```

函数 sigsetjmp 类似于标号,而 siglongjmp 类似于 goto 语句

在程序需要返回的地点调用函数 sigsetjmp,这时 sigsetjmp 将在程序中充当一个标志。函数的第 1 个参数为 sigjmp _ buf 类型,用来记录跳转返回所需的信息;如果第 2 个参数非 0,则信号屏蔽的当前状态会被记录到第 1 个参数中。当程序直接调用函数 sigsetjmp 时,函数返回 0。为了从信号处理函数中跳到 sigsetjmp 处,必须使 siglongjmp 的第 1 个参数为 sigsetjmp 的第 1 个参数,这时,程序将跳到 sigsetjmp 处继续执行,并且好像 sigsetjmp 的返回值就是 siglongjmp 的第 2 个参数。

下面的程序依次输出 0,1,2,...,当用户按下 Ctrl - C 时,程序又重新从 0 开始输出。为了保证 sigsetjmp 在 siglongjmp 之前执行,增加一个变量 jmp _ ok,初始化为 0,当 sigsetjmp 执行完后,将 jmp _ ok 设置为 1。在信号处理函数中,首先监测 jmp _ ok 的值,如为 0,则返回;否则,则调用 siglongjmp 跳到 sigsetjmp 处,由于 siglongjmp 的第二个参数为 2,所以这时 sigsetjmp 返回 2,然后重新开始打印输出。注意,本程序无法退出,只能通过 shell 命令 kill 将其杀掉。

```
/* filename:ex7_3.c */
# include < stdlib.h >
# include < stdio.h >
# include < signal.h >
# include < setjmp.h >
static int jmp _ ok=0;
static sigjmp _ buf jmpbuf;
void sig _ handle(int signo)
```




```
|
|
|   if (jmp_ok == 0)
|       return ;
|
|   siglongjmp(jmpbuf,2);
|
|
|
|
|
|
|
|
|   int main()
|   {
|
|       int count;
|
|       signal(SIGINT,sig_handle);
|
|       if (sigsetjmp(jmpbuf,1) == 2)
|           printf("You pressed CTRL - C and begin \n");
|       jmp_ok = 1;
|
|       count=0;
|       for (;;)
|       {
|           printf("count = %d \n",count);
|           count ++ ;
|           sleep(1);
|       }
|   }
|
```

本程序的运行如下面所示,在打印 count = 3 时,按下了 Ctrl - C,然后接着重新从 0 开始打印。

```
[msf@linux chapter07] $cc ex7_3.c
[msf@linux chapter07] $a.out
count=0
count=1
count=2
^C
You pressed CTRL - C and begin
count=0
count=1
```

第 8 章 基本进程间通信

进程在核心的协调下进行相互间的通信。Linux 支持大量进程间通信(IPC)机制,包括管道、消息以及共享内存等。本章重点讲解管道和消息这两种进程间通信方式。

8.1 管道通信

在 Linux 中,用于进程间通信的管道有两种:普通管道和命名管道。它们的区别如表 8.1 所示。

表 8.1 普通管道与命名管道的区别

特 点	普通管道	命名管道
使用限制	必须由父子进程使用	没有限制
文件名称	无文件名	有文件名,属性为 P
数据读写	先进先出,read)和 write	先进先出,open,read 和 write
创建方式	调用 pipe 创建	调用 mknod 创建
删除方式	使用 rm 命令或 unlink 调用	无需删除,用完会自动删除

8.1.1 普通管道与 pipe 函数

系统调用 pipe 用于创建普通管道。

```
# include <unistd.h>
int pipe (int fd[2]);
```

函数 pipe 的参数为数组名 fd,当该函数调用成功时,文件描述符 fd[0]为数据的读出端,fd[1]为数据的写入端,同时返回 0;当调用失败时,返回 -1。

一个管道只能够实现父子进程之间的单向通信,如要实现双向通信,则通常需要创建两个管道,如图 8.1 所示。

由图 8.1 可知,通信方式为:父进程将要处理的数据写入管道 1,子进程从管道 1 读出数据,进行处理,然后将结果写入管道 2,父进程最后从管道 2 读取结果。

下面的例子就采用这种工作方式,父进程读取一字符串,交给子进程处理。子进程读取字符串,将里面的字符反向后再交给父进程,父进程最后打印反向后的字符串。在程序中,用字符'\n' 作为结束符,可以不与输入字符混淆。

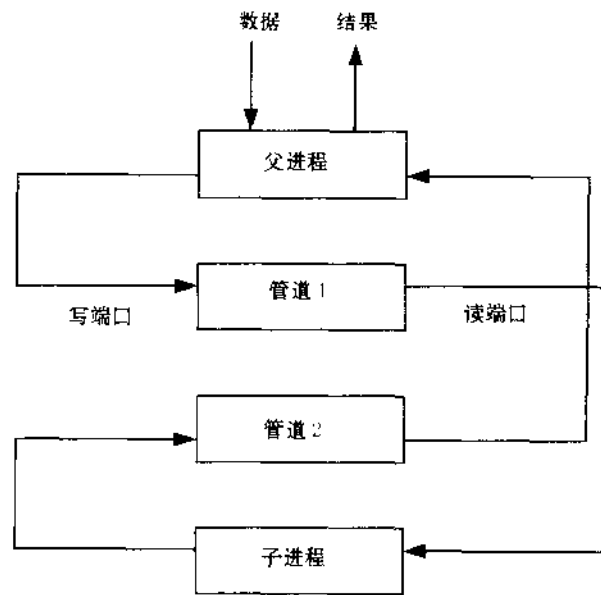


图 8.1 利用管道进行双向通信

```
/* filename: ex8_1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    char buf[256];
    int fd1[2], fd2[2];
    int count;
    char c;
    int result;
    int left, right;

    if ((pipe(fd1) == -1) || (pipe(fd2) == -1))
    {
        printf("error: pipe \n");
        exit(1);
    }

    printf("Input a line of char:");
    scanf("%s", buf);
```



```

if ((result = fork()) == -1)
{
    printf("error; fork \n");
    exit(1);
}
else if (result == 0)
{
    count = 0;
    while (read(fd1[0], &c, 1) > 0)
    {
        if (c == '\n')
            break;
        buf[count] = c;
        count++;
    }
    buf[count] = '\0';

    close(fd1[0]);
    for (left = 0, right = count - 1; left < right; left++, right--)
    {
        c = buf[left];
        buf[left] = buf[right];
        buf[right] = c;
    }
    c = '\n';
    if ((write(fd2[1], buf, count) < 0) || (write(fd2[1], &c, 1) < 0))
    {
        printf("error; write \n");
        exit(1);
    }
    close(fd2[1]);
    exit(0);
}
else
{
    c = '\n';
    count = strlen(buf);

```



```
if ((write(fd1[1], buf, count) < 0) || (write(fd1[1], &c, 1) < 0))
{
    printf("error: write \n");
    exit(1);
}
close(fd1[1]);
wait(NULL);
count = 0;
while (read(fd2[0], &c, 1) > 0)
{
    if (c == '\n')
        break;
    buf[count] = c;
    count++;
}
buf[count] = '\0';
printf("The reversed line: %s \n", buf);
close(fd2[0]);
exit(0);
}
```

程序的运行如下面所示:

```
[msf@linux chapter08] $ cc ex8_1.c
[msf@linux chapter08] $ a.out
Input a line of char: 12345abcde
The reversed line is: edcba54321
```

8.1.2 命名管道与 mknod 函数

系统调用 `mknod` 可以用来创建命名管道,关于 `mknod`,将在以后的章节给予详细讲解。在这里,只要知道第 1 个参数为管道名,第 2 个参数建议为 `S_IFIFO|0644`,第 3 个参数为 0。因此,创建命名管道的 `mknod` 如下所示:

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

int mknod(char *fifoname, S_IFIFO|0644, 0);
```

函数 `mknod` 在成功调用时返回 0,失败时返回 -1。

下面给出了一个使用命名管道进行进程间通信的例子。



这是一个客户机/服务器例子,完成的功能与上一个例子相同。服务器程序打开一个公开的管道用于接收客户机的请求,客户机如果要请求服务,则向公开的管道写请求,在请求中,包含了请求的信息和另一个管道名,服务器收到请求后,则将应答发往客户机指定的管道,最后客户机得到应答。关于客户机的请求,需要有固定的格式,在 request.h 中定义。

```
/* filename: request.h */
#ifndef REQUEST_H
#define REQUEST_H

struct request
{
    char fifo[10];
    int buflen;
    char buf[256];
};

#endif
```

下面分别给出了服务器程序 server.c 和客户机程序 client.c。在创建管道前,就调用 unlink 删除该管道,为的是防止程序在以前运行时非正常退出而没有删除管道,如果不存在该管道,unlink 返回 -1,可以不管它。

```
/* filename: ex8_2server.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include "request.h"

void sig_handle(int signo)
{
    unlink("tmp/svcfifo");
    printf("server quit \n");
    exit(0);
}

int main()
{
    int rfd, wfd;
```



```
struct request request;
int left, right;
char c;
int result;

unlink("/tmp/svcfifo");
if (mknod("/tmp/svcfifo", S_IFIFO|0644, 0) == -1)
{
    printf("error: mknod \n");
    exit(1);
}
signal(SIGINT, sig_handle);

if ((rfd = open("/tmp/svcfifo", O_RDONLY, 0)) == -1)
{
    printf("error: open /tmp/svcfifo \n");
    exit(1);
}
for (;;)
{
    if ((result = read(rfd, &request, sizeof(request))) < 0)
    {
        printf("error: read /tmp/svcfifo \n");
        exit(1);
    }
    if (result == 0)
        continue;
    left = 0;
    right = request.buflen - 1;
    request.buf[request.buflen] = '\0';
    while (left < right)
    {
        c = request.buf[left];
        request.buf[left] = request.buf[right];
        request.buf[right] = c;
        left++;
        right--;
    }
}
```



```

        if ((wfd = open(request.fifo, O_WRONLY, 0)) == -1)
        {
            printf("error: open %s \n", request.fifo);
            exit(1);
        }
        if (write(wfd, request.buf, request buflen) == -1)
        {
            printf("error: write %s", request.fifo);
            exit(1);
        }
        close(wfd);
    }
    close(rfd);
    exit(0);
}

```

/* filename: ex8_2client.c */

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include "request.h"

```

```

int main()
{
    struct request request;
    int fd;
    char buf[256];
    int count;

    printf("Input a line:");
    scanf("%s", request.buf);
    sprintf(request.fifo, "/tmp/fifo%d", getpid());
    request.buflen = strlen(request.buf);
    unlink(request.fifo);
    if (mknod(request.fifo, S_IFIFO | 0644, 0) == -1)
    {

```




```
Input a line of char:12345abcde
The reversed line is:edcba54321
Input a line of char:0987654321
The reversed line is:1234567890
```

8.2 消 息

消息队列能够将格式化数据送往任意的进程,它与命名管道相似,但由于核心程序处理了大部分的控制工作,所以它用起来比命名管道简单。对于每一个消息队列,都有它的标识符。

8.2.1 msgget 函数

系统调用 `msgget` 用于创建一个新的消息队列或者取得一个现存的消息队列的标识符。

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

int msgget (key_t key, int msgflg);
```

函数 `msgget` 有两个参数,第1个为消息队列的名字,如为 `IPC_PRIVATE(0)`,则表示此消息队列为该进程所专用,否则为一个以上的进程所共用。第2个参数为存取权限,与一般文件类似,如果是 `IPC_CREATE` 与存取权限的“或”,则新的消息队列将被创建,标识符为参数 `key`。

函数成功调用时返回创建的消息队列的标识符,失败时返回 `-1`。

8.2.2 msgctl 函数

系统调用 `msgctl` 用于取得、修改消息队列的属性或者删除消息队列。

```
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (int msqid, int cmd, struct msqid_ds * huf);
```

函数 `msgctl` 的第1个参数为消息队列的标识符。第2个参数为操作名称,可以为 `IPC_STAT`、`IPC_SET` 或 `IPC_RMID`。`IPC_STAT` 用于得到消息队列的属性,然后存入第3个参数所指的 `msqid_ds` 结构中;`IPC_SET` 用于设置消息队列的属性;`IPC_RMID` 用于删除创建的消息队列。第3个参数为指向 `msqid_ds` 的指针,结构 `msqid_ds` 的定义如下:

```
struct msqid_ds
{
    struct ipc_perm msg_perm; /* structure of permission */
    struct msg * msg_first; /* pointer to first message on queue */
    ...
}
```



```

struct msg *msg_last; /* pointer to last message on queue */
time_t msg_stime; /* time of last msgsnd command */
time_t msg_rtime; /* time of last msgrcv command */
time_t msg_ctime; /* time of last change */
struct wait_queue *wwait; /* ??? */
struct wait_queue *rwait; /* ??? */
unsigned short int msg_bytes; /* current number of bytes on queue */
unsigned short int msg_qnum; /* number of message current on queue */
unsigned short int msg_qbytes; /* max number of byte allow on queue */
ipc_pid_t msg_lspid; /* pid of last msgsnd() */
ipc_pid_t msg_lrpid; /* pid of last msgrcv() */
;

```

在 `msqid_ds` 结构中, `msg_perm` 用于描述消息队列的权限, 结构 `ipc_perm` 定义如下:

```

struct ipc_perm
{
    key_t key; /* Key. */
    unsigned short int uid; /* Owner's user ID. */
    unsigned short int gid; /* Owner's group ID. */
    unsigned short int cuid; /* Creator's user ID. */
    unsigned short int cgid; /* Creator's group ID. */
    unsigned short int mode; /* Read/write permission. */
    unsigned short int seq; /* Sequence number. */
};

```

关于结构 `msqid_ds` 的其他字段和结构 `ipc_perm` 的各字段的含义, 就用不着详细解释了, 因为注释中已经写得很清楚了。

函数 `msgctl` 调用成功时返回 0, 失败时返回 -1。

8.2.3 msgsnd 和 msgrcv 函数

系统调用 `msgsnd` 和 `msgrcv` 分别用于发送和接收消息。

```

#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (int msqid, void *msgp, size_t msgsz, int msgflg);
int msgrcv (int msqid, void *msgp, size_t msgsz,
            long msgtyp, int msgflg);

```



这两个函数的参数很相似,参数 `msqid` 为消息队列的标识符。参数 `msgp` 为指向消息缓冲区的指针。参数 `msgsz` 为消息的大小。参数 `msgtyp` 表示消息队列中读出消息的类型,如为 0 则读出消息队列中的所有消息。参数 `msgflag` 用来指明核心程序在队列中没有数据的状态下的行动,如 `msgflag` 与 `IPC_NOWAIT` 合用(按位与),则产生如下效果:在队列满的情况下调用 `msgsnd`,`msgsnd` 会返回 -1,而在队列空的情况下调用 `msgrcv`,`msgrcv` 会返回 -1,并设置 `errno` 为 `ENOMSG`;如 `msgflag` 为 0 时,则在上面两种情况下采取等候的处理模式。

这两个函数在调用成功时返回 0,失败时返回 -1。

关于消息缓冲区的结构,一般如下所示,但其中的 `LENGTH` 为一常量,由程序员自己决定。

```
struct msgbuf
{
    long mtype;
    char mtext[LENGTH];
};
```

注意 在定义自己的消息缓冲区的结构时最好不要采用名称 `msgbuf`,因为结构 `msgbuf` 已有系统定义成如下形式:

```
struct msgbuf
{
    long int mtype;    /* type of received/sent message */
    char mtext[1];    /* text of the message */
};
```

下面给出了一个利用消息进行通信的例子,这也是一个客户机/服务器程序,完成与上一个例子相同的功能。

在这个例子中,在文件 `message.h` 中定义了消息缓冲区的格式,如下所示:

```
/* filename: message.h */
#ifndef MESSAGE_H
#define MESSAGE_H

struct mymsgbuf
{
    long mtype;
    char mtext[256];
};

#endif
```



服务器和客户机程序分别如下所示。与命名管道相似,为了防止由于程序的异常终止而导致消息队列没有被删除,在创建消息队列之前检查该消息队列是否存在,如果存在,则调用 `msgctl` 删除它。

```
/* filename: ex8_3server.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include "message.h"

int msqid = -1;

void sig_handle(int signo)
{
    if (msqid != -1)
        msgctl(msqid, IPC_RMID, NULL);
    exit(0);
}

int main()
{
    struct mymsgbuf msgbuf;
    int left, right;
    char c;

    if ((msqid = msgget(999, 0666)) != -1)
    {
        msgctl(msqid, IPC_RMID, NULL);
    }
    if ((msqid = msgget(999, IPC_CREAT | 0666)) == -1)
    {
        printf("error: getmsg \n");
        exit(1);
    }
    signal(SIGINT, sig_handle);
    for (;;)
    {
```

第9章 临界区与高级进程间通信

9.1 竞争现象与临界区

进程具有独立性和异步性等并行特征,而在计算机系统中,由于资源有限,所以导致不同的进程之间经常要竞争资源。

关于竞争现象,最显著的例子就是打印程序。通常打印程序由两个进程组成:一个进程把要打印文件的文件名送入专门的假脱机目录;另一个进程定期检查是否有文件要打印,如有,它就打印该文件,并从假脱机目录中删除该文件。

设想我们的假脱机目录有大量的槽,编号为 0,1,2,...,还假想有两个共同的变量:out 指向要打印的下一个文件,in 指向目录中的下一个空槽。在某一瞬间,0 至 3 号槽是空的(文件已打印),4 到 6 号槽是满的(文件在排队等待打印)。几乎同时,进程 A 和进程 B 决定让一个文件为打印而排队,情况如图 9.1 所示。

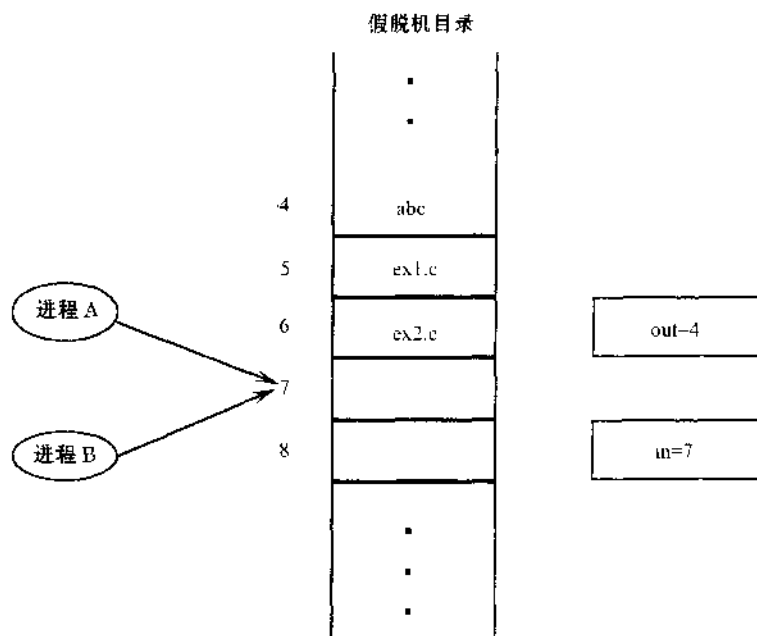


图 9.1 打印队列

这时候,可能发生如下情况:进程 A 读变量 in,把 7 存入 next_free 的局部变量中。恰在此时,时钟中断发生,CPU 认为进程 A 运行超时,于是切换到进程 B。进程 B 也读变量 in,还是得



到 7,故把自己的文件名存入 7 号槽,把 in 变量修改为 8,然后再去做别的事情。

进程 A 后来又运行,从它原来停下的地方开始。它查看 next_free 变量,发现为 7,就把文件名写入 7 号槽,然后就把变量 in 设置为 next_free + 1 即 8。至此,假脱机目录内部是一致的,打印程序也不会觉察到有任何差错发生,这样进程 B 要打印文件就得不到打印,有两个或两个以上的进程在读写某些共享数据时,最终结果取决于谁走运,运行时间巧,这种现象就叫做竞争现象。

竞争现象的问题可以用下面的方式来描述。有一部分时间,进程忙于内部计算和其他不致形成竞争现象的事情;而另一部分时间,进程访问共享存储和文件,或者做别的能够导致竞争的危险事情。程序访问共享存储的那一部分叫做临界区,如果我们能够保证不同时有两个进程进入它们的临界区,我们就可避免竞争现象的发生。

9.2 信号量

9.2.1 简介

信号量的概念最先是由荷兰科学家 E. W. Dijkstra 提出的。信号量是一整数,当它的值大于等于零时代表可供并发进程使用的资源实体数,当它的值小于零时表示正在等待使用临界区的进程数。显然,用于互斥的信号量的初始值应该大于零。

信号量的值只能通过 P、V 原语操作而改变。

P 原语操作的主要动作为:信号量的值减 1,如果减 1 后值仍然大于等于零,则进程继续执行;如果减 1 后值小于 0,该进程就被阻塞到与该信号相对应的队列中。

V 原语操作的主要动作为:信号量的值加 1,如果加 1 后值大于零,进程继续执行;如果加 1 后值小于等于零,则从该信号的等待队列中唤醒一等待进程,然后再返回原进程继续执行。

用信号量实现两并发进程 Pa、Pb 互斥的描述如下:

Pa:	Pb:
...	...
...	...
...	...
P(sem)	P(sem)
<S>	<S>
V(sem)	V(sem)
...	...
...	...
...	...

在上面的描述中,我们在进程 Pa、Pb 进入临界区之前需要将信号量 sem 的值初始化为 1,其中符号 <S> 表示进程进入临界区。



9.2.2 信号量集

信号量与消息类似,也是进程间通信的一种方法。我们在这里讲的信号量,实际上是一个包含信号量元素数组的信号量集。信号量元素与 E.W.Dijkstra 提出的整数信号量相对应。在一个单系统调用中,进程可在完整的信号量集上操作。

信号量集的内部表示和各自信号量元素不是直接可以访问的,但每个信号量元素必须包括下列各项:

- 一个表示信号量元素值的非负整数。
- 最后操作信号量元素的进程 ID。
- 等待信号量元素值加 1 的进程数。
- 等待信号量元素值等于 0 的进程数。

信号量操作允许一个进程阻塞直到信号量元素值为 0 或者直到它变为正数。每个元素具有两个相关联的队列:一个是等待信号量元素值加 1 的进程队列,另一个是等待信号量元素值等于 0 的进程队列。

9.2.3 semget 函数

系统调用 `semget` 用来创建一个信号量集并将每个元素初始化为 0。

```
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key_t key, int nsems, int semflg);
```

函数 `semget` 有 3 个参数。第 1 个参数为标识信号量集的关键字,程序指定关键字的方法有三种:使用 `IPC_PRIVATE` 让系统产生一个关键字、挑选一个随机数,或者是使用 `flok` 从文件路径名中产生一个关键字。第 2 个参数为信号量集中的元素个数。第 3 个参数为信号量存取权标志与建立标志,该参数的低 9 位是信号量的存取权标志;建立标志有 2 个, `IPC_CREAT` 和 `IPC_EXCL`。 `IPC_CREAT` 表示如果信号量集不存在,则创建它; `IPC_EXCL` 表示只有在信号量集不存在的情况下,新的信号量集才会被创建,否则 `semget` 返回 -1,并设定 `errno`。如果这两个标志联合使用,则功能如同 `O_EXCL` 标志于 `open`。

函数 `semget` 调用成功时返回一个用于以后 `semctl` 等操作的整数句柄,失败时返回 -1。在指定信号量集的关键字时,我们可以使用 `flok` 根据文件路径名产生一个关键字。

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

函数 `flok` 的第 1 个参数为文件路径名,该文件必须存在且进程对该文件有访问权。第 2 个参数为程序员指定的一个整数 ID。该函数成功调用时返回一个关键字,如调用失败,则返回 -1。



9.2.4 semctl 函数

系统调用 `semctl` 可以对信号量进行很多控制。

```
#include <sys/ipc.h>
#include <sem.h>

int semctl (int semid, int semnum, int cmd, /* union semum arg */...);
```

函数 `semctl` 的第 1 个参数为信号量集的句柄。第 2 个参数指定信号量集中的元素。第 3 个参数为执行的控制命令, 常见的命令如表 9.1 所示。

表 9.1 常见的命令

命令名称	表示的含义
GETVAL	获得一个指定信号量的值
GETPID	获得操纵此元素的最进程的 ID
GETNCNT	获得等待元素增 1 的进程数
GETZCNT	获得等待元素变为 0 的进程数
SETVAL	设置一个指定信号量元素的值为 <code>arg.val</code>
IPC_RMID	删除一个信号量
IPC_SET	设置信号量的许可权

函数 `semctl` 在出现错误时返回 -1 并设置 `errno`。当调用成功时, 其返回值决定于参数 `cmd`, 当 `cmd` 为 `GETVAL`、`GETPID`、`GETNCNT` 或 `GETZCNT` 时, 函数返回相应的值, 当 `cmd` 为别的值时, 返回 0。

`union semum` 定义可直接包含在程序中, 因为系统的头文件中并没有定义它。它的定义如下:

```
union semum
{
    int val;
    struct semid_ds *buf;
    ushort *array;
};
```

`union semum` 中提到的 `semid_ds` 结构定义如下:

```
struct semid_ds
{
    struct ipc_perm sem_perm; /* operation permission struct */
    time_t sem_otime; /* last semop() time */
    time_t sem_ctime; /* last time changed by semctl() */
};
```




```

struct sem * sembase;      /* ptr to first semaphore in array */
struct sem_queue * sem_pending; /* pending operations */
struct sem_queue * sem_pending_last; /* last pending operation */
struct sem_undo * undo;    /* undo requests on this array */
unsigned short int sem_nsems; /* number of semaphores in set */
};

```

结构 `semid_ds` 与前一章的 `msgqid_ds` 类似,这里就不详细介绍了。

9.2.5 semop 函数

系统调用 `semop` 可以对信号量增 1、减 1 或测试其是否为 0。

```

#include <sys/ipc.h>
#include <sys/sem.h>

int semop (int semid, struct sembuf * sops, unsigned int nsops);

```

函数 `semop` 的第 1 个参数为 `semget` 返回的句柄,第 2 个参数指向元素操作数组,第 3 个参数指定在数组中元素操作的个数。

函数成功调用时返回 0,失败时返回 -1。如果是被信号中断,则返回 -1,同时设置 `errno` 为 `EINTR`。

结构 `sembuf` 的定义如下所示:

```

struct sembuf
{
    short int sem_num; /* semaphore number */
    short int sem_op; /* semaphore operation */
    short int sem_flg; /* operation flag */
};

```

在结构 `sembuf` 中,`sem_num` 表示信号量元素的个数,`sem_op` 表示在信号量元素上执行的特别操作,`sem_flg` 表示操作选项的标志。如果 `sem_op` 为正数,`semop` 将该值加到相应信号量元素中,并唤醒所有等待元素增 1 的进程。如果 `sem_op` 为 0 而信号量的值不为 0,`semop` 将阻塞调用进程并增加那个元素的等零进程个数。如果 `sem_op` 为负数,`semop` 将该值加到相应的信号量元素中(只要结果不为负数),如结果为负数,`semop` 将阻塞进程等待信号量元素值增加;如值为 0,`semop` 将唤醒等零进程。

上面的描述假设 `sem_flg` 为 0。如果 `sem_flg & IPC_NOWAIT` 为真,调用不会阻塞,而是返回 -1 并设置 `errno` 为 `EAGAIN`。如果 `sem_flg & SEM_UNDO` 为真,函数也将为进程修改信号量的调整值。这个调整值允许进程在退出时恢复它在信号量上的作用。

下面的程序给出了关于信号量集的系统调用 `semget`,`semctl` 和 `semop` 的基本用法。

```
/* filename: ex9_1.c */
```



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/ipc.h>

main()
{
    int semid, pid, i, j;
    static struct sembuf lock = {0, -1, SEM_UNDO};
    static struct sembuf unlock = {0, 1, SEM_UNDO|IPC_NOWAIT};

    if ((semid = semget(998, 1, IPC_CREAT|IPC_EXCL|0666)) == -1)
    {
        printf("error; semget \n");
        exit(1);
    }

    if (semctl(semid, 0, SETVAL, 1) == -1)
    {
        printf("error; semctl \n");
        exit(1);
    }

    setbuf(stdout, (char *)NULL);
    for (i=0; i<3; i++)
    {
        if (fork() == 0)
        {
            if ((semid = semget(998, 1, 0)) == -1)
            {
                printf("error; semget \n");
                exit(1);
            }

            for (j=0; j<3; j++)
            {
                sleep(i);
                if (semop(semid, &lock, 1) == -1)
                {

```



```

        printf("error:semop \ n");
        exit(1);
    }

    printf("process %d get into critical section, ".getpid());
    sleep(1);
    printf("process %d left critical section \ n",getpid());
    if (semop(semid, &unlock, 1) == -1)
    {
        printf("error:semop \ n");
        exit(1);
    }

    exit(0);

}

for (i=0;i<3;i++)
    wait(NULL);
if (semctl(semid,0,IPC_RMID,0) == -1)
{
    printf("error:semctl \ n");
    exit(1);
}

exit(0);
}

```

程序的运行情况如下面所示:

```

[msf@linux chapter09]$ cc ex9_1.c
[msf@linux chapter09]$ a.out
process 817 get into critical section,
    process 817 left critical section
process 818 get into critical section,
    process 818 left critical section
process 817 get into critical section,
    process 817 left critical section
process 818 get into critical section,
    process 818 left critical section
process 819 get into critical section,

```



```
process 819 left critical section
process 817 get into critical section.
process 817 left critical section
process 818 get into critical section,
process 818 left critical section
process 819 get into critical section,
process 819 left critical section
process 819 get into critical section,
process 819 left critical section
```

9.3 共享内存

9.3.1 shmget 函数

对于大量的数据,共享内存是一种很好的方法,它兼有简单和快速的特点。其实共享内存非常简单,就是让多个用户,多个进程使用同一块内存。

系统调用 shmget 用于创建共享内存。

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key_t key, int size, int shmflg);
```

函数 shmget 的第 1 个参数 key 为一个整数,可以调用 flock 得到,也可以程序员自己设置或设置为 IPC_PRIVATE。第 2 个参数 size 为共享内存块的大小,以字节为单位。第 3 个参数 shmflg 用来建立共享内存与设置其存取权限,它的第 9 位为存取权限,如果标志设置了 IPC_CREAT,则 shmget 会创建共享内存;如设置了 IPC_EXCL,则表示只有在指定的共享内存不存在时,才会创建新的共享内存。

函数 shmget 成功调用时返回共享内存的标识符,失败时返回 -1。

9.3.2 shmat 函数

在创建共享内存后,我们还是无法对其进行存取,这时就需要调用 shmat 得到共享内存的地址。得到共享内存地址后,我们就可以像操作一般内存那样操作共享内存。

```
#include <sys/ipc.h>
#include <sys/shm.h>

void * shmat (int shmid, const void * shmaddr, int shmflg);
```



函数 `shmat` 的第 1 个参数 `shmid` 为共享内存的标识符,也就是 `shmget` 的返回值。第 2 个参数 `shmaddr` 与第 3 个参数 `shmflg` 结合起来用来指向共享内存被触及的地址。`shmflg` 的作用是决定在定址时是否进位。如果设置了 `SHM_RND`,则地址会进位到可以用来分页的最低位。如果 `shmaddr` 为 `NULL`,则表示该位置由系统指定,一般情况下,我们都是将 `shmaddr` 设为 `NULL`。第 3 个参数还可以用来控制共享内存的读写权,如果设置了 `SHM_RDONLY`,则共享内存仅可以读。

9.3.3 shmdt 函数

与 `shmat` 的作用相反,系统调用 `shmdt` 用来脱离已经触及的共享内存。

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt (const void * shmaddr);
```

函数 `shmdt` 的唯一参数就是 `shmat` 返回的共享内存地址。函数 `shmdt` 在调用成功时返回 0,失败时返回 -1。

9.3.4 shmctl 函数

系统调用 `shmctl` 用来对共享内存进行控制。

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (int shmid, int cmd, struct shmid_ds * buf);
```

函数 `shmctl` 的第 1 个参数 `shmid` 为共享内存的标识符。第 2 个参数 `cmd` 的取值及作用如表 9.2 所示。

表 9.2 参数 `cmd` 的取值

cmd 的取值	作用描述
<code>IPC_STAT</code>	将现在共享内存的状态存入 <code>buf</code> 所指的结构中
<code>IPC_SET</code>	设定共享内存的用户标识符,组标识符与存取权限
<code>IPC_RMID</code>	删除共享内存

函数 `shmctl` 的第 3 个参数为指向结构 `shmid_ds` 的指针,结构 `shmid_ds` 的定义如下所示:

```
struct shmid_ds
{
    struct ipc_perm shm_perm; /* operation permission struct */
    int 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 by shmctl() */
ipc_pid_t shm_cpid; /* pid of creator */
ipc_pid_t shm_lpid; /* pid of last shmop */
unsigned short int shm_nattch; /* number of current attaches */
unsigned short int shm_npages; /* size of segment (pages) */
unsigned long int * shm_pages; /* array of ptrs to frames -> SHMMAX */
struct vm_area_struct * attaches; /* descriptors for attaches */
};

```

在结构 `shmid_ds` 中, `shm_perm` 为存取权限, `shm_segsz` 为共享内存大小, 其他字段的含义就不一一介绍了。

9.3.5 生产者/消费者问题

下面我们将提出一个操作系统中有名的生产者/消费者问题, 并给出了成功的解决方法。

生产者/消费者问题其实是将并发进程问题一般化而得到的一个抽象模型。在计算机系统中, 每个进程都申请使用和释放各种不同类型的资源, 这些资源包括像外设、内存和缓冲区等硬件资源, 也包括临界区、数据等软件资源。我们把系统中使用某一类资源的进程称为该资源的消费者, 而把释放同类资源的进程称为生产者。例如打印问题中的申请打印的进程为消费者, 而打印程序本身即为生产者。

我们把一个长度为 n 的有界缓冲区与一群生产者进程 P_1, P_2, \dots, P_m 和一群消费者进程 C_1, C_2, \dots, C_k 联系起来, 如图 9.2 所示。

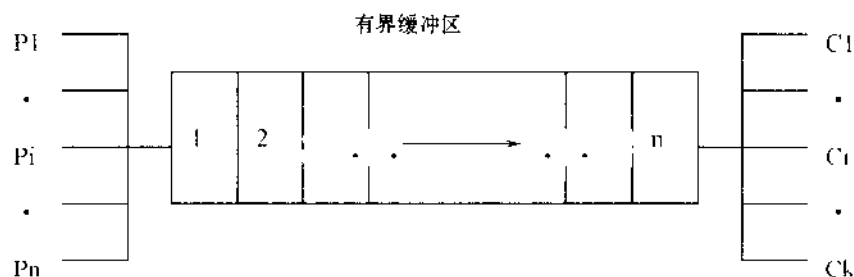


图 9.2 有界缓冲区

生产者在发送数据时, 有界缓冲区中必须要有一个单元是空的; 而消费者在接收数据时, 有界缓冲区中必须有一个单元是满的。

在我们下面给出的模拟程序中, 我们由父进程 `fork` 的四个子进程, 其中两个充当生产者, 两个充当消费者。两个生产者执行相同的程序, 只是调用了函数 `producer`, 两个消费者也类似地调用了 `consumer` 函数。关于有界缓冲区, 我们使用的是循环队列。



```
/* filename:ex9_2.c */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define N 10
#define M 2
#define K 2

struct PQueue
{
    int buf[N+1];
    int front;
    int rear;
};

struct PQueue queue, * queue_ptr;

static struct sembuf lockempty = {0, -1, SEM_UNDO};
static struct sembuf lockfull = {1, -1, SEM_UNDO};
static struct sembuf lockmutex = {2, -1, SEM_UNDO};
static struct sembuf unlockempty = {0, 1, SEM_UNDO|IPC_NOWAIT};
static struct sembuf unlockfull = {1, 1, SEM_UNDO|IPC_NOWAIT};
static struct sembuf unlockmutex = {2, 1, SEM_UNDO|IPC_NOWAIT};

int shmids;
int semids;

void queue_init(struct PQueue * ptr)
{
    ptr->front = 0;
    ptr->rear = 0;
    return ;
}
```



```
void enqueue(struct PQueue * ptr, int id)
{
    ptr->buf[ptr->rear] = id;
    ptr->rear = (ptr->rear + 1) % (N + 1);
    return ;
}

int dequeue(struct PQueue * ptr)
{
    int e;

    e = ptr->buf[ptr->front];
    ptr->front = (ptr->front + 1) % (N + 1);
    return e;
}

struct PQueue * get_shm_ptr()
{
    struct PQueue * ptr;
    char * mem_ptr;

    if ((mem_ptr = shmat(shmid, (char *)0, 0)) == (char *)-1)
    {
        printf("error: shmat \n");
        exit(1);
    }

    ptr = (struct PQueue *)mem_ptr;
    return ptr;
}

void producer(struct PQueue * ptr)
{
    static int count = 0;
    while(1)
    {
        sleep(6);
        semop(semid, &lockempty, 1);
        printf("Empty: %d. Full: %d \n", semctl(semid, 0, GETVAL, 0),
```




```

        semctl(semid, 1, GETVAL, 0));
    printf("Producer %d : LockEmpty ,", getpid());

    semop(semid, &lockmutex, 1);
    printf("Lock mutex ,");

    enqueue(ptr, getpid());
    printf("produce No. %d, ", count + +);

    semop(semid, &unlockmutex, 1);
    printf("UnLock mutex ,");
    semop(semid, &unlockfull, 1);
    printf("UnLock full \n");
    fflush(stdout);
}

void consumer(struct PQueue * ptr)
{
    int id;
    while(1)
    {
        sleep(6);
        semop(semid, &lockfull, 1);
        printf("Empty: %d, Full: %d \n", semctl(semid, 0, GETVAL, 0),
            semctl(semid, 1, GETVAL, 0));
        printf("Consumer %d: Lockfull ,", getpid());

        semop(semid, &lockmutex, 1);
        printf("Lock mutex ,");

        id = dequeue(ptr);
        printf("recive from %d, ", id);

        semop(semid, &unlockmutex, 1);
        printf("Unlock mutex ,");

        semop(semid, &unlockempty, 1);
    }
}

```



```
    if (pid == 0)
    {
        queue_ptr = get_shm_ptr();
        consumer(queue_ptr);
        shmdt((char *)queue_ptr);
        exit(0);
    }
}

for(i=0; i<(M+K); i++)
{
    wait(NULL);
}

shmdt(shm_ptr);
shmctl(shmid, IPC_RMID, 0);
semctl(semid, 0, IPC_RMID, 0);
exit(0);
}
```

程序的运行情况如下面所示:

```
[msf@linux chapter09] $ cc ex9_2.c
[msf@linux chapter09] $ a.out
Empty:9,Full:0
Producer 838 :Lock empty,Lock mutex,
                produce No.0,Unlock mutex,Unlock full
Empty:8,Full:1
Producer 839 :Lock empty,Lock mutex,
                produce No.0,Unlock mutex,Unlock full
Empty:8,Full:1
Consumer 840 :Lock full,Lock mutex,
                receive from 838,Unlock mutex,Unlock empty
Empty:9,Full:0
Consumer 841 :Lock full,Lock mutex,
                receive from 839,Unlock mutex,Unlock empty
Empty:9,Full:0
Producer 838 :Lock empty,Lock mutex,
                produce No.1,Unlock mutex,Unlock full
Empty:8,Full:1
Producer 839 :Lock empty,Lock mutex,
```



```
produce No.1,Unlock mutex,Unlock full
Empty:8,Full:1
Consumer 840 :Lock full,Lock mutex,
receive from 838,Unlock mutex,Unlock empty
```

由于本程序比较长,有 200 多行。因此,我们有必要对程序中的函数作一下介绍:

程序一开始的 3 个函数 `queue_init`、`enqueue` 和 `dequeue` 是用来操纵循环队列的,`queue_init` 用来初始化循环队列,`enqueue` 用来向队列中增加一项,`dequeue` 用来从队列中取走一项。关于队列的具体算法,我们这里就不作介绍了,读者可以参考数据结构的教材。需要提一下的是,我们在增加元素时并没有考虑队列是否为空,取走元素时同样也没有考虑队列是否已经满了,因为信号量 `empty` 和 `full` 已经控制了缓冲区,使它不可能发生越界情况。

函数 `get_shm_ptr` 返回一个指向队列的指针,该指针指向共享内存区。这样,以后操作共享内存区的队列就非常简单。

函数 `producer` 和 `consumer` 是本程序的关键,函数 `producer` 由生产者进程调用,函数 `consumer` 由消费者进程调用。在函数 `procedure` 中,我们首先对信号量 `empty` 和 `mutex` 依次进行 P 操作,然后对信号量 `mutex` 和 `full` 依次进行 V 操作。两个 V 操作的顺序,是可以调换的;而两个 P 操作的顺序,是不可以随便调换的,否则可能引起死锁。函数 `consumer` 与 `producer` 类似,首先对信号量 `full` 和 `mutex` 依次进行 P 操作,然后对信号量 `mutex` 和 `empty` 依次进行 V 操作。同样,两个 V 操作的顺序,是可以调换的;而两个 P 操作的顺序,是不可以随便调换的。如果读者想要了解死锁的原因,可以参考操作系统的教材,我们这里就不介绍了。

函数 `sig_handle` 用来处理 `SIGINT` 信号,由于程序按照正常情况执行无法终止,所以我们只有靠敲 `Ctrl-C` 键来终止程序。

在主函数 `main` 中,我们完成了程序的初始化工作。我们首先创建了三个信号量:`empty`、`full` 和 `mutex`,将它们的值分别设置为 `N`、`0`、`1`。接着调用 `queue_init` 完成循环队列的初始化。然后我们创建了共享内存,并且将循环队列拷入共享内存中,这样我们的循环队列才能够被多个进程访问。最后我们创建了 `M` 个生产者,`K` 个消费者,到此为止,程序的初始化工作已经全部完成,主函数就等待子进程的终止。

9.4 小 结

本章介绍了竞争现象和临界区的概念,重点讨论了进程间通信的另外两种形式:信号量和共享内存。在本章的最后,我们还介绍了生产者/消费者问题,并且给出了一个模拟程序。

第 10 章 Linux 线程

10.1 线程简介

10.1.1 传统进程的局限性

传统上的进程模型有两个严重的局限性。首先,许多应用程序都想并发执行那些彼此间独立的任务,但是它们必须要共享一个公共的地址空间和其他的资源,这类应用的例子包括服务器上的数据库管理服务器,事务处理监测程序,以及中间层和高层的网络协议。这些进程本质上是并行的,所以需要支持并行的编程模型。传统的 Unix 系统强行地把这些应用中独立的任务串行化或者是设计一些糟糕的、而且效率很低的机制来管理这些操作。

其次,传统的进程不能很好地利用多处理器体系结构。因为一个进程的某个时刻只能使用一个处理器。一个应用必须创建许多个进程,并把它们分配到所有那些可用的处理器上去执行。这些进程应该找到一种方法来共享内存和其他资源,并且实现互相间的同步。

10.1.2 线程的动机

许多程序必须执行一些独立的不需要串化的任务。比如,一个数据库服务器应该能监听和处理大量的客户请求。因为这些请求不需要按照一个特定的顺序来得到服务,所以它们可以被当作独立的执行体来看待,原则上它们可以并行运行的。如果系统提供了子任务可以并发执行的机制,那么这些应用程序可以执行得更好。

在传统的 Unix 系统中,这些程序使用多个进程。许多关键的服务器应用程序有一个监听进程在不停地运行,等待客户请求到来。当一个请求到达时,这个监听进程创建(fork)一个新的进程为这个请求服务。因为对请求进行服务经常包括一些 I/O 操作,它可能阻塞进程。这种方法甚至在单处理器的系统中也能获得一些并发性。

在一个应用程序中使用多个进程有着一些明显的缺点。创建这些进程增加了一些基本的开销,由于 fork 是一个花销很大的系统调用。由于每个进程都有它自己的地址空间,它必须使用进程间通信的手段如消息传递或者共享内存。要把这些进程分配到不同的机器或处理器上去运行,及在进程之间传递信息、等待进程的完成、收集结果等都需要额外的开销。

这些都说明了进程抽象概念不充分之处,因此迫切需要一种能很好地适合于并行计算机的方法。现在能够建立一个独立的计算单元的概念模型,这些计算单元是一个应用程序全部处理工作的一个部分。这些单元之间的交互相对是很少的,因此需要很少的同步。一个应用程序可能含一个或多个这种的单元。线程这种抽象概念就代表了一个单独计算单元。传统的 Unix 进程是单线程的,这意味着所有的计算都被串行化在同一个单元之中了。

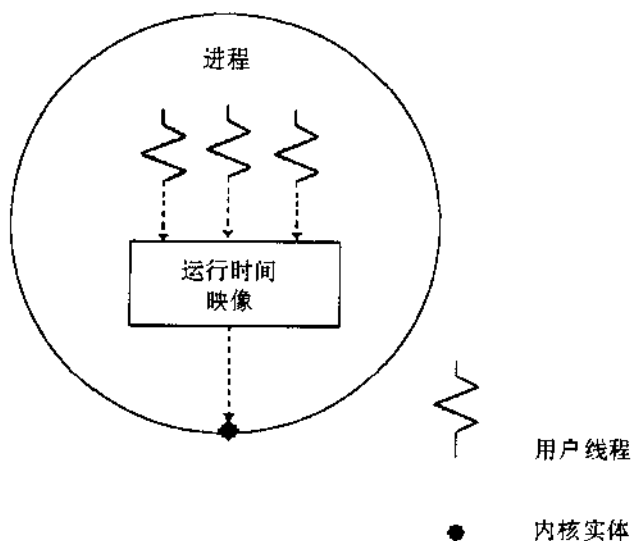


图 10.1 用户线程在进程之外是不可见的

用户级线程具有特别低的开销,但它们具有一些缺点。这个模型依赖于那些系统可以重获运行时间的线程的拥有。而 CPU 受限的线程几乎不执行系统调用或库调用,以防止线程运行时间系统重获控制来安排别的线程。因此,程序员只得避开这种封锁的情况,通过显式地强迫受限的 CPU 线程在合适的点上产生控制。用户级线程更加严重的问题是线程仅分享分配给封装进程的处理机资源。这个约束限制了并行的数量,因为线程每次只能在一个处理机上运行。最初使用线程的目的是利用多处理机工作站提供的并行性,但就用户级线程自身来说,它是不可接受的方法。

在内核级线程中,内核将每个线程看作一个调度的实体。如图 10.2 所示,线程在一个广阔的基础上竞争处理机资源。内核级线程的调度如同调度进程自身一样耗时,但内核内

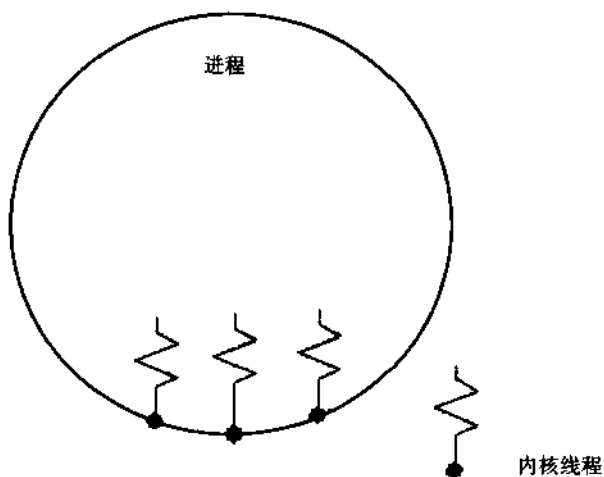


图 10.2 内核线程像进程一样被调度



线程可利用多级处理机。内核级线程提供的同步和共享数据的耗时不比全部进程长,但与用户级线程相比,却要长得多。

混合线程模型具有用户线程和内核级线程的优点,它提供两级控制。用户依照用户线程来写程序,然后指定与进程相连的内核可调度的实体的数量。当进程正在运行以获得并行性时,用户级线程被映射到内核可调度的实体上。在映射上,用户具有的控制级别依赖于具体的实现。

10.2 线程管理

典型的线程包含一个运行时间系统,它可以按透明的方式来管理线程。通常线程包括对线程创建和删除,以及对互斥和条件变量的调用。POSIX 标准线程库具有这些调用。这些包还提供线程的动态创建和删除,因此直到运行时间之前,线程的个数不必知道。

线程具有一个 ID、一个堆栈、一个执行优先权,以及执行的开始地址。POSIX 线程通过 `pthread_t` 类型的 ID 来引用。`pthread_t` 其实就是无符号长整数,在文件 `/usr/include/bits/pthreadtypes.h` 中有如下的定义:

```
typedef unsigned long int pthread_t;
```

线程的内部数据结构也包含调度和使用信息。进程的线程共享进程的完整地址空间,它们能够修改全局变量,访问打开的文件描述符,或用别的方式相互作用。

10.2.1 pthread_create 函数

如果线程可在进程的执行期间的任意时刻被创建,并且线程的数量事先没有必要指定,这样的线程称为动态线程。在 POSIX 中,线程是用 `pthread_create` 动态地创建的。`pthread_create` 能创建线程,并将它放入就绪队列。创建线程需要调用 `pthread_create`,该函数的定义如下所示:

```
#include <pthread.h>
int pthread_create (pthread_t * thread,

    const pthread_attr_t * attr,
    void * (* start_routine) (void *), void * arg);
```

该函数的第 1 个参数 `thread` 将指向创建的线程的 ID。线程的属性是由 `attr` 指向的属性对象来包容的。如果 `attr` 为 `NULL`,新线程具有缺省的属性。第 3 个参数 `start_routine` 是线程开始执行时调用的函数名字。`start_routine` 占用一个参数 `arg`,它是一个指向 `void` 的指针。`start_routine` 返回一个指向 `void` 的指针,此指针被看作是 `pthread_join` 的退出状态。

函数 `pthread_create` 在成功调用时返回 0,失败时返回 -1。



10.2.2 pthread_self 函数

在创建线程之后,可以调用 pthread_self 得到线程的 ID,该函数的定义如下所示:

```
#include <pthread.h>

pthread_t pthread_self(void);
```

10.2.3 pthread_exit 函数

与进程退出时调用 exit 类似,线程退出时也需要调用 pthread_exit, pthread_exit 的定义如下所示:

```
#include <pthread.h>

void pthread_exit(void *retval);
```

函数 pthread_exit 终止调用的线程。参数 retval 的值对 pthread_join 函数的成功有实际意义。然而, pthread_exit 的 retval 必须指定,在线程退出时它才退出的数据,因此它不能够作为正在退出的线程的自动局部数据被分配。

函数 pthread_exit 在成功调用时返回 0,失败时返回 -1。

10.2.4 pthread_join 函数

在成功地创建线程之后,可以调用 pthread_join 将新创建的线程加入到原进程中去, pthread_join 的定义如下所示:

```
#include <pthread.h>

int pthread_join(pthread_t th,
void ** thread_return),
```

如果调用了 pthread_join,那么进程会等待线程调用 pthread_exit 之后才退出,这个函数的作用类似于 wait 系统调用。 pthread_join 的第 1 个参数 th 为所创建线程的 ID,第 2 个参数指向线程的退出码。

函数 pthread_join 在成功调用时返回 0,失败时返回 -1。

10.2.5 线程的例子

下面给出了一个运用上面介绍的线程基本管理函数的例子,这是关于多线程的最简单的例子,但它说明了实现多线程的一般步骤。

```
/* filename: ex10_1.c */
```



```
# include <stdio.h>
# include <errno.h>
# include <pthread.h>
# include <unistd.h>
# define THREAD_NUM 2

void * my_thread(void * arg)
{
    pthread_t id;
    int * nump;
    int num;
    int retval;

    nump = (int *) (arg);
    num = * nump;
    printf("thread %d: startad \n", num);
    id = pthread_self();
    printf("thread %d: ID is %ld \n", num, id);
    retval = 100 + num;
    printf("thread %d: exit with %d \n", num, retval);
    pthread_exit( &retval);
}

int main()
{
    int count;
    pthread_t thread[ THREAD_NUM+1 ];
    int * retval[ THREAD_NUM+1 ];
    int arg[ THREAD_NUM+1 ];

    count = 1;
    while(count <= THREAD_NUM)
    {
        printf("Create thread %d \n", count);
        arg[ count ] = count;
        if (pthread_create( &thread[ count ], NULL, my_thread,
                           (void *) ( &arg[ count ] )) != 0)
        {
            fprintf(stderr, "Count not create thread %d: %s \n",

```




```
count, strerror(errno));  
:  
count++;  
:  
for (count = 1; count <= THREAD_NUM; count++)  
:  
    if (pthread_join(thread_count, (void *)(&retval[count])) != 0)  
    :  
        fprintf(stderr, "No thread %d to join: %s\n", count, strerror(errno));  
  
    else  
    :  
        printf("Thread %d returned %d\n", count, *retval[count]);  
    :  
:  
{  
exit(0);  
|
```

这个例子的运行结果如下面所示:

```
[msf@linux chapter10] $cc ex10_1.c /usr/lib/libpthread.so  
[msf@linux chapter10] $a.out  
Create thread 1  
Create thread 2  
thread 1: started  
thread 1: ID is 1026  
thread 1: exit with 101  
thread 1 returned 101  
thread 2: started  
thread 2: ID is 2051  
thread 2: exit with 102  
thread 2 return 102
```

从例子的编译命令可以知道,有关线程的程序在编译时都要用到`/usr/lib/libpthread.so`这个线程库文件。

另外还有一点要注意,在 `main` 中利用了数组 `arg` 向线程传递参数。读者或许会认为这没有必要,可以直接将 `count` 作为参数传给新的线程,那为什么不这样做呢?从源程序我们看出,在调用 `pthread_create` 时,如果把 `count` 作为参数传给线程函数 `my_thread`,从那以后,原进程有两个线程,一个调用 `my_thread` 函数,另外一个将执行 `count++` 语句。如果前一个线程在后一个线程执行 `count++` 语句后才执行 `num = *num_p` 语句,则 `num` 的值



在该结构中, detachstate 表示线程的拆卸状态, schedpolicy 表示线程的调度策略, schedparam 表示线程的调度参数, inheritsched 表示线程的继承性, scope 表示线程的作用域, stackaddr 表示线程堆栈的位置, stacksize 表示线程堆栈的大小。

表 10.1 给出了线程属性的可设置的特性以及与这些特性相关的函数。

表 10.1 线程的属性函数

函数名称	函数作用
pthread_attr_init	初始化线程属性
pthread_attr_destroy	销毁线程属性对象
pthread_attr_setstacksize	设置线程堆栈的大小
pthread_attr_getstacksize	得到线程堆栈的大小
pthread_attr_setstackaddr	设置线程堆栈的位置
pthread_attr_getstackaddr	得到线程堆栈的位置
pthread_attr_setdetachstate	设置线程的拆卸状态
pthread_attr_getdetachstate	得到线程的拆卸状态
pthread_attr_setscope	设置线程的作用域
pthread_attr_getscope	得到线程的作用域
pthread_attr_setinheritsched	设置线程的继承性
pthread_attr_getinheritsched	得到线程的继承性
pthread_attr_setschedpolicy	设置线程的调度策略
pthread_attr_getschedpolicy	得到线程的调度策略
pthread_attr_setschedparam	设置线程的调度参数
pthread_attr_getschedparam	得到线程的调度参数

10.3.1 线程属性对象的初始化和销毁

在使用一个线程属性对象之前,必须对其进行初始化, pthread_attr_init 就是完成对线程属性对象初始化的;在使用完一个线程属性对象之后,必须对其进行销毁, pthread_attr_destroy 就是完成对线程属性对象销毁的。这两个函数的定义如下所示:

```
#include <pthread.h>

int pthread_attr_init (pthread_attr_t * attr);
int pthread_attr_destroy (pthread_attr_t * attr);
```

函数 pthread_attr_init 和 pthread_attr_destroy 都只有一个参数,此参数为一指向属性对象的指针。



这两个函数在成功调用时返回 0, 失败时返回 -1。

10.3.2 线程堆栈的大小

函数 `pthread_attr_setstacksize` 和 `pthread_attr_getstacksize` 分别用来设置和得到线程堆栈的大小, 这两个函数的定义如下所示:

```
#include <pthread.h>

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr,
                              size_t *stacksize);
```

这两个函数具有两个参数, 第 1 个是指向属性对象的指针, 第 2 个是堆栈大小或指向堆栈大小的指针。

这两个函数在成功调用时返回 0, 失败时返回 -1。

10.3.3 线程堆栈的地址

函数 `pthread_attr_setstackaddr` 和 `pthread_attr_getstackaddr` 分别用来设置和得到线程堆栈的位置, 这两个函数的定义如下所示:

```
#include <pthread.h>

int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(const pthread_attr_t *attr,
                              void **stackaddr);
```

这两个函数具有两个参数, 第 1 个是指向属性对象的指针, 第 2 个是堆栈地址或指向堆栈地址的指针。

这两个函数在成功调用时返回 0, 失败时返回 -1。

10.3.4 线程的拆卸状态

函数 `pthread_attr_setdetachstate` 和 `pthread_attr_getdetachstate` 分别用来设置和得到线程的拆卸状态, 这两个函数的定义如下所示:

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

这两个函数具有两个参数, 第 1 个是指向属性对象的指针, 第 2 个是拆卸状态或指向拆卸状态的指针。拆卸状态可能的值是 `PTHREAD_CREATE_JOINABLE` 或是 `PTHREAD_CREATE_DETACHED`, 缺省值是前者。



这两个函数在成功调用时返回 0,失败时返回 -1。

10.3.5 线程的作用域

函数 `pthread_attr_setscope` 和 `pthread_attr_getscope` 分别用来设置和得到线程的作用域,这两个函数的定义如下所示:

```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

这两个函数具有两个参数,第 1 个是指向属性对象的指针,第 2 个是作用域或指向作用域的指针。作用域控制线程是否在进程内或在系统级上竞争资源,可能的值是 `PTHREAD_SCOPE_PROCESS` 和 `PTHREAD_SCOPE_SYSTEM`。

这两个函数在成功调用时返回 0,失败时返回 -1。

10.3.6 线程的继承性

函数 `pthread_attr_setinheritsched` 和 `pthread_attr_getinheritsched` 分别用来设置和得到线程的继承性,这两个函数的定义如下所示:

```
#include <pthread.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
                                int *inherit);
```

这两个函数具有两个参数,第 1 个是指向属性对象的指针,第 2 个是继承性或指向继承性的指针。继承性决定调度的参数是从创建的线程中继承的还是从显示指定中继承。继承性可能的值是 `PTHREAD_INHERIT_SCHED` 和 `PTHREAD_EXPLICIT_SCHED`。

这两个函数在成功调用时返回 0,失败时返回 -1。

10.3.7 线程的调度策略

函数 `pthread_attr_setschedpolicy` 和 `pthread_attr_getschedpolicy` 分别用来设置和得到线程的调度策略,这两个函数的定义如下所示:

```
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                                int *policy);
```

这两个函数具有两个参数,第 1 个是指向属性对象的指针,第 2 个是调度策略或指向调度策略的指针。调度策略可能的值是先进先出(`SCHED_FIFO`)、轮转法(`SCHED_RR`),或是其它未定义(`SCHED_OTHER`)。



这两个函数在成功调用时返回 0,失败时返回 -1。

10.3.8 线程的调度参数

函数 `pthread_attr_setschedparam` 和 `pthread_attr_getschedparam` 分别用来设置和得到线程的调度参数,这两个函数的定义如下所示:

```
#include <pthread.h>

int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                               struct sched_param *_param);
```

这两个函数有两个参数,第 1 个参数是指向属性对象的指针,第 2 个参数是 `sched_param` 结构或指向该结构的指针。结构 `sched_param` 在文件 `/usr/include/bits/sched.h` 中定义,如下所示:

```
struct sched_param
{
    int sched_priority;
};
```

结构 `sched_param` 的子成员 `sched_priority` 控制一个优先权值,大的优先权值对应高的优先权。

这两个函数在成功调用时返回 0,失败时返回 -1。

10.3.9 得到线程的属性

下面给出了一个例子,在这个例子中,调用 `pthread_create` 创建一个线程,然后在创建的线程中,得到线程的各个属性,并且将它们打印出来。

```
/* filename: ex10_2.c */
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <unistd.h>

void *my_thread(void *arg)
{
    int retval=0;
    pthread_attr_t attr;
    struct sched_param param;
```



```
size_t stacksize;
int detachstate;
int scope;
int inherit;
int policy;

if (pthread_attr_init(&attr) == 0)
{
    if (pthread_attr_getstacksize(&attr, &stacksize) == 0)
    {
        printf("StackSize: %d \n", stacksize);
    }
    if (pthread_attr_getdetachstate(&attr, &detachstate) == 0)
    {
        if (detachstate == PTHREAD_CREATE_JOINABLE)
            printf("DetachState: PTHREAD_CREATE_JOINABLE \n");
        if (detachstate == PTHREAD_CREATE_DETACHED)
            printf("DetachState: PTHREAD_CREATE_DETACHED \n");
    }
    if (pthread_attr_getscope(&attr, &scope) == 0)
    {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("Scope : PTHREAD_SCOPE_PROCESS \n");
        if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("Scope : PTHREAD_SCOPE_SYSTEM \n");
    }
    if (pthread_attr_getinheritsched(&attr, &inherit) == 0)
    {
        if (inherit == PTHREAD_INHERIT_SCHED)
            printf("InheritSched: PTHREAD_INHERIT_SCHED \n");
        if (inherit == PTHREAD_EXPLICIT_SCHED)
            printf("InheritSched: PTHREAD_EXPLICIT_SCHED \n");
    }
    if (pthread_attr_getschedpolicy(&attr, &policy) == 0)
    {
        if (policy == SCHED_FIFO)
            printf("SchedPolicy: SCHED_FIFO \n");
        if (policy == SCHED_RR)
            printf("SchedPolicy: SCHED_RR \n");
    }
}
```



```

        printf("SchedPolicy:SCHED _RR \n");
    else
        printf("SchedPolicy:SCHED _OTHER \n");
    :
    if (pthread_attr_getschedparam( &attr, &param) == 0)
    {
        printf("SchedPriority: %d \n", param.sched_priority);
    }
    pthread_attr_destroy( &attr);
}
pthread_exit( &retval);
}

int main()
{
    int count;
    pthread_t thread;
    int * retval;

    if (pthread_create( &thread, NULL, my_thread, (void *) NULL) != 0)
    {
        fprintf(stderr, "Count not create thread: %s \n", strerror(errno));
    }
    if (pthread_join(thread, (void **) (&retval)) != 0)
    {
        fprintf(stderr, "No thread to join: %s \n", strerror(errno));
    }
    else
    {
        printf("Thread returned %d \n", *retval);
    }
    exit(0);
}

```

程序的运行结果如下面所示:

```

[msf@linux chapter10] $cc ex10_2.c /usr/lib/libpthread.so
[msf@linux chapter10] $a.out
StackSize: 0

```



```
DetachState:PTHREAD_CREATE_JOINABLE
Scope:PTHREAD_SCOPE_SYSTEM
InheritSched:PTHREAD_EXPLICIT_SCHED
SchedPolicy:SCHED_OTHER
SchedPriority:0
Thread returned :0
```

10.4 小 结

本章讨论了线程的一些概念,包括线程与进程之间的区别,线程模型的优点,线程的基本管理函数和线程的属性等。

第 11 章 TCP/IP 简介

11.1 网络简介

计算机网络是一种地理上分散的、具有独立功能的多台计算机通过通信设备和线路连接起来的,在配有相应的网络软件的情况下实现资源共享的系统。一台主控机和多台从属机的系统不能称为网络。同样地,一台带有大量终端的大型机也不能称为网络。处于网络中的计算机应具有独立性,如果一台计算机可以强制地启动、停止或控制另一台计算机,这些计算机就不具备独立性。

网络和 Linux 是密切相关的。从某种意义上说 Linux 是一个针对 Internet 和 WWW 的产品。它的开发者和用户用 Web 来交换信息、程序代码,而 Linux 自身常常被用来支持各种组织机构的网络需求。

11.2 TCP/IP 及相关协议

TCP/IP 协议最初是为支持 ARPANET(一个美国政府资助的研究性网络)上计算机通信而设计的。ARPANET 提出了一些网络概念如包交换和协议分层(一个协议使用另一个协议提供的服务)。ARPANET 于 1988 年隐退,但是它的继承者(NSF1 NET 和 Internet)却变得更大了。现在我们所熟知的万维网 World Wide Web 就是从 ARPANET 演变过来的,它自身支持 TCP/IP 协议。Unix TM 被广泛应用于 ARPANET,它的第一个网络版本是 4.3 BSD。Linux 的网络实现是以 4.3 BSD 为模型的,它支持 BSD sockets(及一些扩展)和所有的 TCP/IP 网络。选这个编程接口是因为它很流行,并且有助于将应用程序从 Linux 平台移植到其它 Unix TM 平台。

11.2.1 IP 协议

IP 协议是网际层的主要协议。它的主要功能有:无连接数据报传送、数据报路由选择和差错控制。将报文传送到目的主机后,不管传送正确与否都不进行检验,不回送确认,也不保证分组的正确顺序,这些功能都留给 TCP 完成。

一个数据报由一个头部和一个正文部分构成。头部的组成如表 11.1 所示。

版本域指示每个 IP 协议数据报的 IP 协议版本,占 4 比特。目前的流行版本为 4,即 IPv4。

报头长度域占 4 比特,以 32 比特长度为一个单位。最小值是 5 个单位。这个 4 位字段的最大值是 15 个单位,它限制了头部的最大长度是 60 字节,因此可选字段最多为 40 字节



表 11.1 IP 头部的组成

版本	报头长	服务类型			总 长	
标 志				D	M	分片位移
生命期		协 议		头校验和		
源 地 址						
目 的 地 址						
选项(0 或更多字节)						

对于某些选项,如记录分组所经过的路由,40 字节往往不够,在这种情况下,这些选项也变得毫无用处。

服务类型占 8 比特,指示如何处理数据报。该域的前三位指明优先关系,共有八级,“7”代表最高优先级,“0”代表最低优先级。接下来的 4,5,6 位分别指示低延时,高吞吐和高可靠性。取值为 1 时,数据报所期望的服务类型有效。

总长域占 16 位,只是整个 IP 数据报的长度,包括报头和数据,最大值为 65535,即最大 IP 数据报的长度为 65535 字节。

标识域占 16 位,用于控制分片重组,同一个数据报不管分成多少片都具有相同的标识号。

紧跟的两位无用。然后为 2 个 1 位字段。D 代表 DF,数据报不能分片。M 代表 MF。M = 1 代表该分片后还有分片;M = 0 代表该分片是末分片。

分片位移占 13 位,表示分片在当前数据报中的位置。

生命期占 8 位,用来确定数据报被允许在网络中传输最多可用多少秒。其作用为防止网络中出现环路而长期存在,以秒为单位,最多位 255 秒。当该域为 0 时,将丢弃此数据报。

协议域占 8 位,用来指示传输协议的类型,如“6”代表 TCP 协议,“7”代表 UDP 协议。

头检验和域占 16 位,用来确保数据报头的无差错传输。

源地址和目的地址指明了源主机和目的主机的 IP 地址。

选项域主要用于网络的控制。表 11.2 为已定义的选项。

TCP/IP 用 IP 地址来标识源地址和目的地址。Internet 网上的任何一台主机都有一个 IP 地址,IP 地址必须保证全球唯一。IP 地址为 32 位,常用带点的十进制数书写。在这种格式下,每字节用十进制数表示,取值从 0 到 255。例如,十六进制地址 C0290614 被记为 192.41.6.20。

表 11.2 IP 数据报的选项

选 项	描 述
安全性	指明数据报的机密程度
严格的源路由选择	给出完整的路由
松散的源路由选择	给出一个不能漏掉的路由列表
记录路由	使每个路由器都附上它的 IP 地址
时间标记	使每一个路由器都附上它的 IP 地址和时间标记

值 0 和 FFFFFFFF 有特殊的意义,0 表示本网络或本主机,FFFFFFF 表示广播地址,



下面将介绍 ICMP 报文的几种常见类型,如果读者希望知道全部的 ICMP 类型,请查看其他相关文献或帮助。

1) 回应请求与回应应答

这种报文主要用于源站检测目的站是否可通。在 Linux 系统中有个程序叫 ping,专门用于检测目的站可否到达。发送方发送回应请求报文,目的站接到回应请求报文后,发送回应应答报文。

报文格式如表 11.5 所示。

表 11.5 回应请求与回应应答的报文格式

类型 = 8/0	代码 = 0	校验和
标识符		序列号
数 据		

类型为 8 代表回应请求,类型为 0 代表回应应答。

2) 目的站不可达

这种报文主要用于向源站返回目的站不可达的信息。路由器在丢弃相应的 IP 数据报之前,就会发送目的站不可达的报文给源站。

报文格式如表 11.6 所示。

表 11.6 报文格式

类型 = 3	代码	校验和
没有用		
IP 数据报头 + 8 字节的原始数据报报文		

类型为 3 代表目的站不可达。关于代码域,有六种情况,表 11.7 列出了这六种情况。

表 11.7 报文代码

代 码	含 义
0	目的网络不可达
1	目的主机不可达
2	目的协议不可达
3	目的端口不可达
4	需要分片,但 DF 位已经设置
5	源站路由失败

IP 数据报头 + 8 字节的原始数据报报文的用途为:主机要用这些数据来判定到底将该报文发送给哪个进程,高层协议利用段口号,一般来说,段口号在数据报的开始 8 字节中。

3) 超时报文

当网关在处理数据报时,它必须丢弃生存期为 0 的数据报。网关就用超时报文来通知



表 11.11 ARP 和 RARP 的报文格式

0	16	31
硬件类型		协议类型
MAC 地址长度	IP 地址长度	操作
源 MAC 地址(N 字节,用不完填充 0)		
源 IP 地址(M 字节,用不完填充 0)		
目的 MAC 地址(N 字节,用不完填充 0)		
目的 IP 地址(M 字节,用不完填充 0)		

当服务请求者给出欲进行地址解析的 IP 地址后,ARP 查找 IP 地址与 MAC 地址对应表,若找到的话,则将查到的 MAC 地址返回给服务请求者。否则的话,广播报文“四处寻找”目标地址。网络中的主机截获报文,如其中有一台主机的 IP 地址出现在该报文的目的 IP 地址域中,则将其 MAC 地址填入报文中发回给寻求者,然后寻求者将此对应关系加到其 MAC 地址和 IP 地址的对应表中,并将 MAC 地址返回给服务请求者。

11.2.4 TCP 协议

1) TCP 数据段头

表 11.12 给出了 TCP 数据段的布局格式。每个数据段均以固定格式的 20 字节的头开始。固定的头后面可能是头的一些可选项。在可选项后面最多有 $65535 - 20 - 20 = 65495$ 字节数据,第一个 20 指 IP 头,第二个 20 指 TCP 头。数据段不带任何数据也是合法的,一般用于确认报文和控制报文。

表 11.12 TCP 数据段头

32 比特									
源端口					目的端口				
顺序号									
确认号									
TCP 头长	未用字段	U	A	P	R	S	F	窗口大小	
		R	C	S	S	Y	I		
		G	K	H	T	N	N		
校验和					紧急指针				
可选项(0 或更多的 32 位字)									
数据(可选项)									

源端口和目的端口字段标识本地和远端的连接点。端口号加上主机的 IP 地址构成一个 48 位的唯一的 TSAP。用源端和目的端机器的套接字序号一起来标识一个连接。

顺序号和确认号字段执行它的通用功用,二者均为 32 位。TCP 头长表明在 TCP 头中包含多少个 32 位字,因为可选项字段是变长的,所以 TCP 头也是变长的。这一字段实际是指明数据在数据段中的开始位置,它是以 32 位字为单位来测量的,但它给出的是头部包括多少个 32 位字。



接下来的 6 位未用字段保留了 10 多年原封未动,这表明 TCP 协议的考虑是十分慎重的。

如果用到了应急指针,那么 URG 位置 1 应急指针指从当前顺序号到紧急数据位置的偏移量。该设置用于代替中断报文,它允许发送方向接受方发送信号,同时又避免 TCP 本身陷入探究中断原因中去。

ACK 位置 1 表明确认号是合法的。如果 ACK 为 0,那么数据段不包含确认信息,确认号字段被省略。

PSH 位表示是带有 PUSH 标志的数据。接受方因此请求数据段一到便可送往应用程序而不必等到缓冲区装满时才传送。

RST 位用于复位由于上机崩溃或其他原因而出现错误的连接,还用于拒绝非法的数据段或拒绝连接请求。

SYN 位用于建立连接。在连接请求中,SYN = 1,ACK = 0,表示捎带确认字段无效。连接响应数据段应带有确认,因此 SYN = 1,ACK = 1。实际上,SYN 位用来代表 CONNECTION REQUEST 和 CONNECTION ACCEPTED,用 ACK 位来区分这两种可能。

FIN 位用于释放连接。表明发送方已经没有数据发送了,但当断开连接后进程还可以继续接收数据。

TCP 中的流量控制是通过使用可变大小的滑动窗口来处理的。窗口大小字段表示在确认了字节之后还可以发送多个字节。窗口大小字段值为 0 是合法的,表示它已经收到了包括确认号减 1 在内的所有数据段。

校验和也是为了确保可靠性而设置的。它校验 TCP 头部、数据和表 11.13 所示的概念上的伪 TCP 头之和。执行此操作之前,TCP 的校验和字段被设置为 0,并且当数据长度是奇数时数据字段附加填充一个 0 字节。校验的算法是将所有 16 位字以补码形式相加,然后再对相加和取补。

表 11.13 包括在 TCP 校验和中的伪头

32 比特		
源地址		
目的地址		
00000000	协议 = 6	TCP 数据段长

伪 TCP 头包含:源机器和目的机器的 32 位 IP 地址,TCP 的协议编号(6),TCP 数据段(包括 TCP 头)的字节数。在校验和计算中包括了伪 TCP 头,这有助于检测传送的分组是否正确,但却违反了协议的分层规则,因为其中的 IP 地址是属于 IP 层的而非 TCP 层。

选项域用于提供一种增加额外设置的方法,而这种设置在常规的 TCP 头中并不包括。

2) TCP 连接管理

在 TCP 中为了建立连接,服务器或客户方中的一方可通过执行 LISTEN 和 ACCEPT 原语被动地等待一个到达的连接请求。另一方执行 CONNECT 原语,同时指明它想连接到的 IP 地址和端口号,设置它能够接受的 TCP 数据段的最大值,以及一些可选的用户数据(如口令)。

CONNECT 原语发送一个 $\text{SYN}=1, \text{ACK}=0$ 的数据段到目的端,并等待对方响应。

该数据段到达目的端后,那里的 TCP 实体将查看是否有进程在侦听目的端口字段指定的端口。如果没有,它将发送一个 $\text{RST}=1$ 的应答,拒绝建立该连接。如果某个进程正在对该端口进行侦听,便将到达的数据段交给该进程。该进程可以拒绝或接受建立连接的请求,接受的话,便发回一个确认数据段。TCP 数据段的发送顺序如图 11.1(a)所示,SYN 数据段使用了 1 字节的顺序空间,所以可以明确地得到确认。

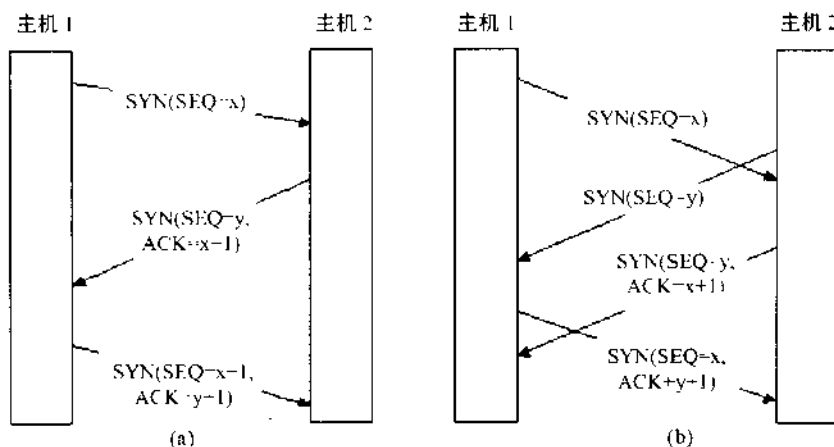


图 11.1 TCP 连接管理示意图

(a) 一般情况下连接的建立过程; (b) 呼叫碰撞的情形。

两个主机若同时在相同的两个套接字之间建立连接,它的最终结果是只有一个连接建立起来(事件发生顺序如图 11.1(b)所示)。因为连接是由其端点所标识的。如果首先建立的连接由 (x, y) 标识,第二个连接也是如此,那么只生成一个表记录,即 (x, y) 。

建立连接和释放连接所需要的步骤可用具有 11 种状态的有限状态机表示,如表 11.14 所示。

表 11.14 有限状态机的状态

状 态	描 述
CLOSED	没有连接是活动的或正在进行
LISTEN	服务器在等待进入呼叫
SYN RCVD	一个连接请求已到达,等待确认
SYN SENT	应用已开始,打开一个连接
ESTABLISHED	正常数据传输状态
FIN WAIT 1	应用说它已完成
FIN WAIT 2	另一边已同意释放
TIME WAIT	等待所有分组死掉
CLOSING	两边同时尝试关闭
CLOSE WAIT	另一边将要释放连接
LAST ACK	等待所有分组死掉



每个连接均开始于 CLOSED 状态。当一方执行了主动的连接原语 (CONNECT) 或被动的连接原语 (LISTEN) 时, 它便会脱离 CLOSED 状态。如果另一方同时执行了相对应的连接原语, 连接便建立了, 此时状态变为 ESTABLISHED。当连接被释放后, 状态又回到了 CLOSED。

有限状态机如图 11.2 所示。服务器首先发起一个“被动打开”的操作, 这导致服务器的有限状态机进入“听”(LISTEN) 状态。此时服务器等待有一个客户机与它联络。当有一个客户机发起“主动打开”操作时, 它导致该机器上的 TCP 软件发送一个 SYN 报文端给服务器, 并进入“SYN 已发出”(SYN-SENT) 状态。当在“听”状态中等待的服务器接受到 SYN 报文段后, 以一个 SYN 再加一个 ACK 报文段作为回答, 并创建一个新的 TCP, 将新的 TCP 置于“SYN 收到”(SYN-RCVD) 状态。当 SYN 加上 ACK 的报文段到达客户机后, 客户机的 TCP 以一个 ACK 作为回答, 并从“SYN 已发出”状态变到“已经建立”(ESTABLISHED) 状态。最后, 当客户机的 ACK 报文段到达新创建的 TCB 后, 该 TCB 也进入“已经建立”状态, 这时就可以进行数据传输了。

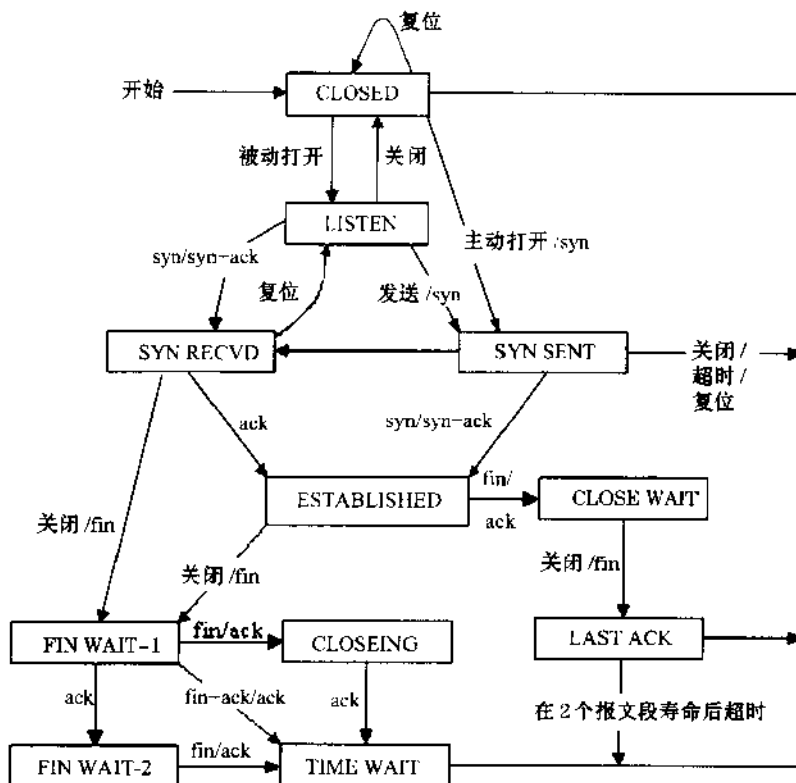


图 11.2 TCP 用于控制进程动作的有限状态机

11.2.5 UDP 协议

UDP (User Data Protocol) 协议是一种无连接的传输协议, 它提供的服务不可靠、无流量控制且不排序。与 TCP 相比, 它要简单得多, 它在与 IP 或其他协议连接时, 只充当数据报



的发送者和接收者。UDP 一般用于传输数据较少的交互式业务,如很多有一个请求和一个响应的客户/服务器应用程序采用 UDP,这样可以避免建立和释放连接的麻烦。一个 UDP 数据段包括一个 8 字节的头和数据部分。格式如表 11.15 所示。

表 11.15 UDP 数据段头

1	16	31
源端口		目的端口
UDP 长度		UDP 校验和

两个端口的作用与 TCP 中的相同,长度域指定 UDP 报头和数据的字节数。UDP 校验和字段包括伪 UDP 头、UDP 头和 UDP 数据,它是根据 UDP 数据报内容生成的,接收方重新计算校验和并与之比较,若相同则认为发送正确。

11.2.6 DNS 协议

随着计算机网络的不断发展,最初的只用一个文件表示网上所有的主机地址及其对应的二进制 IP 地址的方法不再行得通。因为这样的话,在文件下载时所用时间过长、主机名也可能发生冲突,网络往往不能正常工作。域名系统 DNS(Domain Name System)就是在这种情况下产生的。

1) DNS 解析机制

DNS 主要用来把主机名和电子邮件地址映射为 IP 地址,它的核心是分级的、基于域的命名机制以及为了实行这个命名机制的分布式数据库系统。为了把一个名字映射为一个 IP 地址,应用程序调用一种名叫解析器的库过程。解析器将 UDP 分组传送到本地 DNS 服务器上,然后本地 DNS 服务器查找名字并将 IP 地址返回给解析器,解析器再把它返回给调用者。这样程序就可以和目的方建立 TCP 连接,或者向它发送 UDP 分组。图 11.3 是域名解析过程示意图。

2) 名字空间

DNS 的工作方式类似于信件,每个域由从它往上到根的路径命名。成员由点分隔,如 ee.tsinghua.edu.cn。域名对大小写不敏感,成员名最多可有 63 个字符,路径全名不能超过 255 个字符。域的分布就像一棵树,它有最高层,下面为子域,子域下面还可划分。所有这些域如图 11.4 所示。一个树叶域可以代表一个部门包含许多的主机,也可以只包含一台主机。

每个域或子域都有其固有的域名,域名 ee.tsinghua.edu.cn 中 cn 为国家级域名,表示中国。edu 为子域名,表示教育部门。tsinghua 表示清华大学,ee 表示电子系,是第四级域名。

顶层域目前分为三类:国家、国际和一般的域。每个域都控制分配它下面的域。如日本的 ac.jp 域和 co.jp 域分别对应 edu 和 com。而荷兰却把所有的组织直接放在 nl 下。要创建一个新的域必须征得它所属的域的同意。例如,一个新的大学在中国成立,不妨取名为 Newje,它就必须请求 edu 域的管理者将它命名为 newje.edu。这样就可以避免名字冲突。

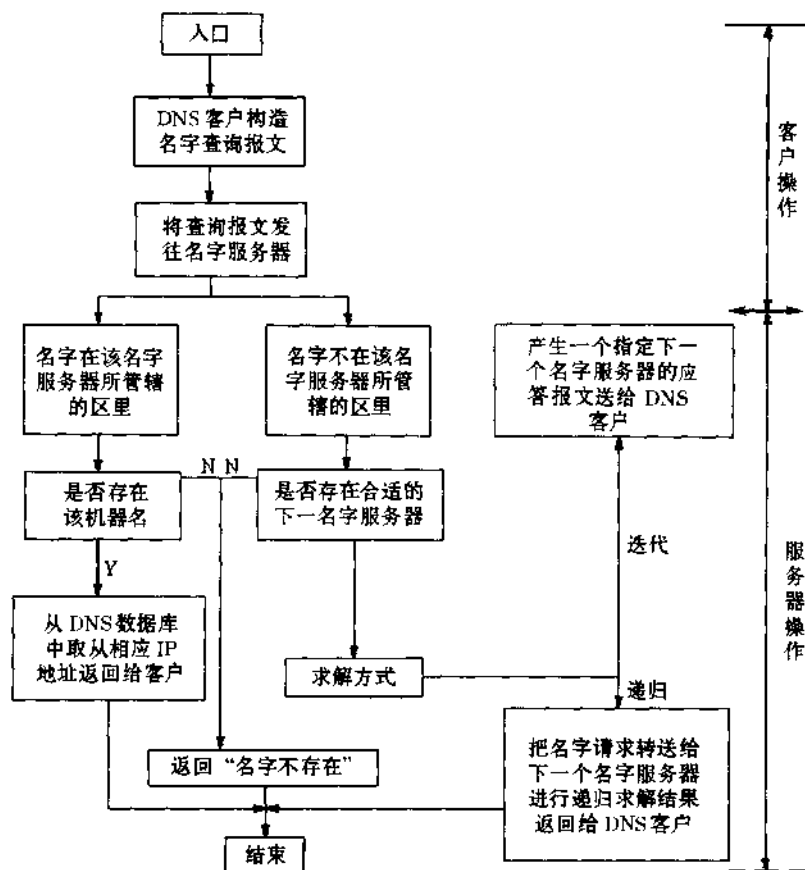


图 11.3 域名解析

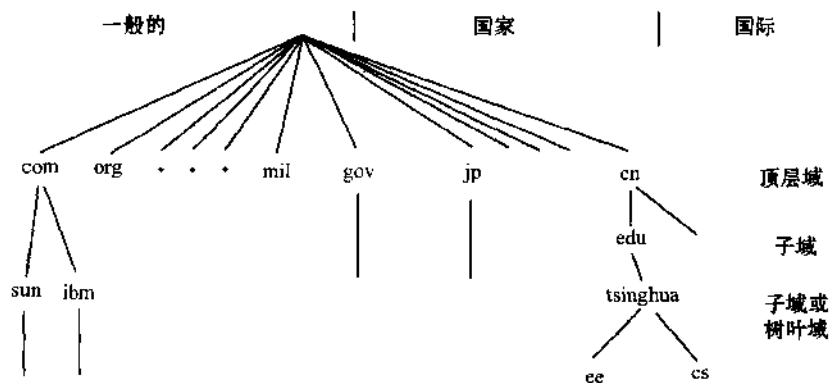


图 11.4 域的分析

一旦一个新的域被创建和登记,它就可以创建子域而无须得上级域的同意。

3) 资源记录

每个域都有其相关的资源记录,当解析器交给 DNS 一个域名后,它将获得与这个域名



必须进行划分,划分成包含若干命名树或全部命名树的一些区。每个区中通常配置一个名字服务器。

由区域管理员负责对区域边界进行划分。这个决定很大程度上取决于需要多少名字服务器以及将其放在何处。

现在我们来举一个域名解析过程的实例。用户甲的电子邮箱名为 jia@mail.pku.edu.cn,用户乙的电子邮箱名为 yi@tsinghua.edu.cn。当甲发送电子邮件给乙后,为了获得乙电子邮箱的 IP 地址,就开始了名字解析过程。如图 11.5 所示。

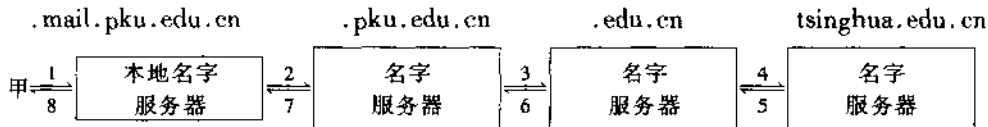


图 11.5 域名解析过程示意图

第一步,在发送者本地名字服务器 .pku.edu.cn 中无法查询到接收者的域名。于是发送一个询问至高一层的名字服务器 .edu.cn,这时 .edu.cn 就能查询到接收者域名的 IP 地址及其他相应的反馈信息。IP 地址与其他相应的反馈信息由步骤 5、6、7、8 返回给发送者。

除了上述递归查询方式外,还有迭代查询方式。当本地名字服务器不能获得查询答案时,就返回下一个名字服务器的名字给客户,以此类推,直至找到具有接收者名字的服务器。

11.3 小 结

本章讨论了 TCP/IP 及相关协议,重点讲解了 IP、TCP、UDP 等协议,了解这些协议是进入以后网络编程的基础,从下一章开始,我们将讨论网络编程。

第 12 章 各种转换

12.1 网络字节序转换函数

TCP/IP 对协议首部中所使用的二进制整数规定了一种标准的表示方式。这种表示方式为网络字节序,它在表示整数时,让最高位在前。尽管协议对应用程序掩藏了协议首部中使用的大部分值,但还是必须注意这个标准,因为在很多程序中要求参数按网络字节序存储。例如 `sockaddr_in` 结构中的协议端口域就使用网络字节序。

系统提供了一些在网络字节序和主机字节序之间进行转换的函数。程序应该调用这些转换函数。这些函数分为短和长两类,用以表示 16 位和 32 位的整数。函数 `htons` 将一个短整数从主机字节序转换为网络字节序,函数 `htonl` 将一个长整数从主机字节序转换为网络字节序,函数 `ntohs` 将一个短整数从网络字节序转换为主机字节序,函数 `ntohl` 将一个长整数从网络字节序转换为主机字节序。

```
#include <netinet/in.h>

short  htons(short  x);
short  ntohs (short  x);
long   htonl (long   x);
long   ntohl (long   x);
```

12.2 IP 地址的转换

12.2.1 `inet_aton` 与 `inet_addr` 函数

在一般情况下,IP 地址都是用带点的十进制数来表示,但是在计算机内部,IP 地址都存储为 32 位二进制数。因此,需要一些函数将带点的十进制数转换为 32 位二进制数,或者将 32 位二进制数转换为带点的十进制数。在这一小节中,讨论这些转换。

```
#include <arpa/inet.h>

u_int32_t  inet_addr (const char *cp);
int        inet_aton (constchar *cp, struct in_addr *inp);
char*      inet_ntoa (struct in_addr in);
```



函数 `inet_addr` 的参数 `cp` 为带点的十进制字符串,返回网络字节序的 32 位二进制数。如果出错,则返回 `INADDR_NONE`,`INADDR_NONE` 一般被定义为 `FFFFFFFF` 这就意味着带点的十进制字符串 `255.255.255.255` 不能由此函数处理,因为它的二进制数被用来指示函数失败。

函数 `inet_aton` 的第 1 个参数 `cp` 为带点的十进制字符串,第 2 个参数为一个 `in_addr` 结构的地址。正确时,函数返回 1,同时将转换结果写入 `in_addr` 结构。如返回 0,则表示发生错误。

函数 `inet_ntoa` 完成的转换与前两个函数相反,该函数的唯一参数 `in` 为一个 `in_addr` 结构,返回带点的十进制字符串。

在上面函数中提到的结构 `in_addr` 在文件 `<netinet/in.h>` 中被定义为如下形式:

```
struct in_addr
{
    in_addr_t s_addr;
};
```

在结构 `in_addr` 的定义中,`s_addr` 为 `in_addr_t` 类型,`in_addr_t` 就是 32 位整数。

12.2.2 inet_pton 与 inet_ntop 函数

上一小节讨论的三个函数虽然能够完成带点的十进制数与 32 位二进制数的转换,但不建议读者使用,我们希望读者能够使用这一小节讨论的两个转换函数,因为前三个函数不支持 IPv6,而这小节讨论的 `inet_pton` 和 `inet_ntop` 同时支持 IPv4 和 IPv6。

```
#include <arpa/inet.h>

int    inet_pton (int af, const char * cp, void * buf);
const char* inet_ntop (int af, const void * cp,
                      char * buf, size_t len);
```

函数 `inet_pton` 将带点的十进制数转换为二进制数。该函数成功时返回 1,如返回 0 代表输入的带点的十进制数有误,如返回 -1 代表出错。函数的第 1 个参数 `af` 为 `AF_INET` (代表 IPv4)或 `AF_INET6` (代表 IPv6)。第 2 个参数 `cp` 为带点的十进制数。第 3 个参数 `buf` 为存放二进制数的地址。

函数 `inet_ntop` 将二进制数转换为带点的十进制数。该函数返回指向带点的十进制数的指针,如返回 `NULL`,则代表出错。函数的第 1 个参数 `af` 为 `AF_INET` (代表 IPv4)或 `AF_INET6` (代表 IPv6)。第 2 个参数 `cp` 为指向二进制数的指针。第 3 个参数 `buf` 为存放带点的十进制字符串的地址。最后一个参数 `len` 是长度,可以是 `INET_ADDRSTRLEN` (IPv4)或 `INET6_ADDRSTRLEN` (IPv6)。`INET_ADDRSTRLEN` 和 `INET6_ADDRSTRLEN` 为常数,在 `<netinet/in.h>` 被分别定义为 16 和 46。



12.3 名字地址的转换

12.3.1 gethostbyname 函数与 gethostbyname2 函数

在实际生活中,经常见到的地址都形如 `www.tsinghua.edu.cn`,而很少见到形如 `166.111.5.4` 的地址。一般认为 `www.tsinghua.edu.cn` 为一台计算机的名字,而 `166.111.5.4` 为这台计算机的地址。因此,还得提供另一种转换,将一台计算机的名字转换为它的地址,或者将它的地址转换为它的名字。

在这一小节中,将讨论这些转换。函数 `gethostbyname` 提供由名字到地址的转换。

```
#include <netdb.h>

struct hostent *gethostbyname(const char *hostname);
```

函数返回一个指向 `hostent` 结构的指针,如返回 `NULL`,则表示出错。函数的唯一参数 `hostname` 可以为主机的名字,也可以是像 `166.111.1.1` 那样的带点十进制数。

该函数完成的功能,实际上为域名的解析。该函数的执行顺序为:首先查看文件 `/etc/hosts`,看是否有这一名字的表项,如果有,解析完毕;否则,它将发送 DNS 查询给域名服务器,如查到结果,则函数成功返回,如没有结果,函数返回 `NULL`。

现在看一下函数返回的 `hostent` 结构的定义。`hostent` 结构记录了一台主机的很多信息。`hostent` 的结构如下所示:

```
struct hostent
{
    char *h_name;
    char * *h_aliases;
    int h_addrtype;
    int h_length;
    char * *h_addr_list;
};

#define h_addr h_addr_list[0]
```

在 `hostent` 结构中,字段 `h_name` 为主机的规范名字;`h_aliases` 指向主机的别名字符串数组;`h_addrtype` 为地址类型,`AF_INET` 或 `AF_INET6`;`h_length` 为地址长度,对 IPv4 为 4,对 IPv6 为 16;`h_addr_list` 指向主机的地址字符串数组。

为了支持 IPv6,后来用定义了一个函数 `gethostbyname2`,该函数的返回值与 `gethostbyname` 相同。函数的第 1 个参数为主机的名字,第 2 个参数为 `AF_INET` 或 `AF_INET6`。函数 `gethostbyname2` 的定义如下:

```
#include <netdb.h>

struct hostent *gethostbyname2(char *name, int family);
```



度。主机名长度的最大值在文件 `<sys/param.h>` 中被定义为常数 `MAXHOSTNAMELEN`。

12.3.4 得到主机的信息

在本小节中,给出一个利用前面介绍的函数得到主机信息的程序。在该程序中,函数 `usage` 显示了该程序的用法。函数 `printinfo` 用于显示主机的一些信息,如主机的正式名字、别名以及地址。该函数的参数为 `hostent` 结构的指针。在主函数 `main` 中,如 `argc = 1`,则显示本机的信息;如 `argc = 2`,则用函数 `gethostbyname` 得到 `hostent` 结构的指针,再调用函数 `printinfo` 显示信息;否则调用依次 `inet_ntop`, `gethostbyaddr` 得到 `hostent` 结构的指针,再调用函数 `printinfo` 显示信息。下面是源程序:

```
/* filename : ex12_1.c */
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/utsname.h>
#include <netinet/in.h>
#include <netdb.h>
void usage(char *str)
{
    printf("usage: %s _name/ipaddress] \n", str);
    return ;
}
void printinfo(struct hostent *phost)
{
    char *ptr, * *pptr;
    char str[INET6_ADDRSTRLEN];
    printf("official name: %s \n", phost->h_name);
    pptr = phost->h_aliases;
    for (; *pptr != NULL; pptr++)
    {
        printf(" \t aliases: %s \n", *pptr);
    }
    pptr = phost->h_addr_list;
    for (; *pptr != NULL; pptr++)
    {
        printf(" \t address: %s \n", inet_ntop(phost->h_addrtype,
            *pptr, str, sizeof(str)));
    }
    return ;
}
```



```

int main(int argc, char * * argv)
{
    struct hostent host, * phost;
    struct utsname name;
    char str[INET6_ADDRSTRLEN];
    char * ptr;
    char * * pptr;
    struct in_addr addr;
    if (argc != 1 & &argc != 2)
    {
        usage(argv[0]);
        exit(1);
    }
    if (argc == 1)
    {
        if (uname(&name) == -1)
        {
            printf("Error:uname \n");
            exit(1);
        }
        if ((phost = gethostbyname(name.nodename)) == NULL)
        {
            printf("Error:gethostbyname \n");
            exit(1);
        }
        printinfo(phost);
        exit(0);
    }

    if ((phost = gethostbyname(argv[1])) != NULL)
    {
        printinfo(phost);
        exit(0);
    }
    else if (inet_pton(AF_INET, argv[1], (void *) &addr) == 1)
    {

```



```
        if ((phost = gethostbyaddr((char *) &addr, 4, AF_INET)) != NULL)
        {
            printinfo(phost);
            exit(0);
        }
        else
        {
            printf("Error: gethostbyname \n");
            exit(1);
        }
    }
    else
    {
        printf("unknown host %s \n", argv[1]);
        exit(1);
    }
}
```

下面给出了该程序的运行情况：

```
[msf@linux chapter12] $ cc ex12_1.c
[msf@linux chapter12] $ a.out
official name: linux.tsinghua.edu.cn
alias: linux
address: 192.168.0.41
[msf@linux chapter12] $ a.out lq
official name: lq.tsinghua.edu.cn
alias: lq
address: 192.168.0.40
[msf@linux chapter12] $ a.out linux
official name: linux.tsinghua.edu.cn
alias: linux
address: 192.168.0.41
```

12.4 服务名的转换

一台服务器向外提供很多服务,每一个服务都监听一个端口号,如 WWW 服务就监听 80 端口。但有些时候只知道服务名,而不知道端口号,应该如何访问这些服务呢? 答案在于只要提供服务名与端口号之间的转换,就能解决这一问题。下面就讨论服务名的转换。



12.4.1 getservbyname 函数

函数 `getservbyname` 提供了由服务名得到端口号的方法。下面是该函数的定义：

```
#include <netdb.h>

struct servent * getservbyname (const char * servname, const char
                                * proto name);
```

函数 `getservbyname` 返回一个指向 `servent` 结构的指针,如返回 `NULL`,则调用失败。函数的第 1 个参数 `servname` 为服务名,第 2 个参数 `proto name` 为协议名。其中,参数 `servname` 必须被指定,而协议名可以为 `NULL`,当协议名为 `NULL` 时,返回值依赖于具体的实现。但对于一般支持多协议的服务经常使用同一个端口号,因此还不碍事。

对于返回的 `servent` 结构,一般来说,最感兴趣是端口号。由于返回的端口号是网络字节的,我们在填充 `sockaddr_in` 结构时就不能再次调用函数 `htons`。

该函数的执行为:查看文件 `/etc/service`,看是否有这一名字的表项,如果有,则函数成功返回,如没有结果,函数返回 `NULL`。

此函数返回 `servent` 结构的指针,`servent` 结构的定义如下:

```
struct servent
{
    char * s_name;
    char * * s_aliases;
    int s_port;
    char * s_proto;
};
```

下面是函数 `getservbyname` 的典型用法:

```
struct servent * sptr;

sptr = getservbyname("ftp","tcp");
sptr = getservbyname("ftp","udp");
sptr = getservbyname("ftp",NULL);
sptr = getservbyname("domain","udp");
```

12.4.2 getservbyport 函数

函数 `getservbyport` 提供了由端口号得到服务名的方法。下面是该函数的定义:

```
#include <netdb.h>

struct servent * getservbyport (int port, const char * proto name);
```



函数 `getservbyport` 同样返回一个指向 `servent` 结构的指针,如返回 `NULL`,则调用失败。函数的第 1 个参数 `port` 为端口号,第 2 个参数 `protoname` 为协议名。其中,参数 `port` 必须被指定,而且必须为网络字节序;而协议名可以为 `NULL`。

函数 `getservbyport` 的典型用法如下所示:

```
struct servent * sptr;
sptr = getservbyport(htons(53), "udp"); /* DNS using UDP */
sptr = getservbyport(htons(21), "tcp"); /* FTP using TCP */
sptr = getservbyport(htons(21), NULL); /* FTP using TCP */
sptr = getservbyport(htons(21), "udp"); /* fail, because no such service */
```

最后一个调用会失败,是因为系统的 21 端口号上没有 UDP 服务。

12.5 高级地址转换

在前面几节中讨论的地址转换函数,如 `gethostbyname`, `gethostbyaddr` 都依赖于具体的协议。在本节中,将讨论两个新函数, `getaddrinfo` 和 `getnameinfo`,它们都是不依赖于协议的,无论为 IPv4 还是 IPv6,都能准确地工作。

12.5.1 `getaddrinfo` 函数

函数 `getaddrinfo` 隐藏了所有的与协议有关的部分,这样,在 IPv4 的环境下编的应用程序就能够轻易地移植到 IPv6 环境中。函数 `getaddrinfo` 的定义如下:

```
#include <netdb.h>

int getaddrinfo (const char * hostname, const char * service,
                 const struct addrinfo * hints, struct addrinfo ** result);
```

函数 `getaddrinfo` 如成功,则返回 0;如返回非 0,则表示出错。函数返回由 `result` 指向的 `addrinfo` 结构的链表。第 1 个参数 `hostname` 为主机名,可以为名字或 IP 地址。第 2 个参数 `service` 为服务名,也可以为名字或端口号。第 3 个参数可以为 `NULL` 或一个 `addrinfo` 结构的指针,如为 `NULL`,则返回所有的 `addrinfo` 结构;如 `hints` 的某些域被指定,则指返回满足条件的 `addrinfo` 结构。

结构 `addrinfo` 的定义如下:

```
struct addrinfo
{
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    size_t ai_addrlen;
```



```
# include < netdb.h>
void freeaddrinfo (struct addrinfo * ai);
```

12.5.2 getnameinfo 函数

函数 `getnameinfo` 与 `getaddrinfo` 互补,它以一个套接字地址和该地址的长度为参数,返回一个描述主机的字符串和一个描述服务的字符串。函数 `getnameinfo` 的定义如下:

```
# include < netdb.h>
int getnameinfo (const struct sockaddr * sockaddr, socklen_t addrlen,
                 char * host, size_t hostlen, char * serv, size_t servlen,
                 int flags);
```

函数 `getnameinfo` 调用成功,则返回 0,如返回 -1,则表示出错。函数的 `sockaddr` 结构与结构的长度通常由函数调用 `accept`、`recvfrom` 或 `getpeername` 返回。

调用者要为主机名字符串和服务名字符串分配空间,参数 `hostlen` 和 `servlen` 分别指出了这两个空间的长度。最大的主机名长度在文件 `< netdb.h>` 中被定义为 `NI_MAXHOST` (1025),最大的服务名长度也被定义为 `NI_MAXSERV` (32)。函数的最后一个参数 `flags` 用于控制返回的格式。它的取值如表 12.2 所示。

表 12.2 flag 的取值

常 量	意 义
<code>NI_DGRAM</code>	数据报服务
<code>NI_NAMEREQD</code>	如名字不能解析,则返回错误
<code>NI_NOFQDN</code>	只返回主机名
<code>NI_NUMERICHOST</code>	返回主机名的点串形式
<code>NI_NUMERICSERV</code>	返回服务名的点串形式

12.6 小 结

本章讨论了用于网络编程的各种转换,包括网络字节序的转换、IP 地址的转换、名字地址的转换、服务名的转换和高级地址转换。这些各种各样的转换,虽然简单,但却是在编程过程中必不可少的,因此,读者必须掌握这些转换。

第 13 章 套接字编程

13.1 套接字简介

为了提供进程间通信的一般方法和允许使用复杂的协议,实现不同主机的进程间的通信,BSD UNIX 提出了一种称为套接字(socket)的机制。套接字的出现使开发人员得以方便地访问 TCP/IP 协议。

套接字是一种双向的通信端口,一对互连的套接字提供通信接口,使两端可以传输数据。套接字通常采用客户机/服务器模型:服务器在作为双向通信通路一端的套接字上监听,而客户进程在通信通路的另一端(可能在另一台机器上)的套接字上与之通信。

套接字的通信机制提供了一系列的函数供开发人员调用,在下面的章节中,将详细讨论这些函数。

13.2 套接字编程调用

在介绍套接字的各种调用之前,首先讨论一些结构,这些结构是套接字的各种调用的基础。

最一般的结构为 sockaddr 结构,它包含了一个占两字节的地址族标识符,还有一个占 14 字节的数组来存储地址:

```
struct sockaddr
{
    u_char sa_len;
    u_short sa_family;
    char sa_data[14];
};
```

但是,并不是所有的地址族都定义了适合这种 sockaddr 结构的端点。这样,应用程序就不能在变量声明中使用 sockaddr 结构了。

因为 sockaddr 结构适用于 AF_INET 族的地址,这在实际中常常引起混淆。于是,甚至把变量声明为 sockaddr 类型时,TCP/IP 软件也能正确工作。然而,为使程序可移植和可维护,TCP/IP 代码不能在声明中使用 sockaddr 结构。这种结构只能用于覆盖,而且代码只能引用该结构中的 sa_family 字段。

使用套接字的每个协议都精确地定义了它的端口地址,套接口也提供了相应的结构声



明。每个 TCP/IP 端点地址由下列字段构成：一个 2 字节字段来识别地址类型，一个 2 字节的端口字段，一个 4 字节的 IP 地址字段，一个未用的 8 字节字段。结构 `sockaddr_in` 指明了这种格式：

```
struct sockaddr_in
{
    u_char sin_len;
    u_short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};

struct in_addr
{
    in_addr_t s_addr;
};
```

在 `sockaddr_in` 结构中，`sin_len` 为长度字段。然而，一般情况下，不必设置它，而是简单地将它置 0。`sin_family` 为协议类型，可以为 IPv4 (`AF_INET`) 或 IPv6 (`AF_INET6`)。`sin_port` 为网络字节序的端口号，常常是一项服务对应于一个端口号，例如 `www` 服务对应于 80 端口，`ftp` 服务对应于 21 端口等。`sin_addr` 为 32 位长整数，通常，要通过转换函数将主机的 IP 地址或名字转换为 32 位长整数。下面是一个简单的例子：

```
struct sockaddr_in servaddr;

bzero(&servaddr);
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(13);
inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
```

由上面的例子可以知道，在使用 `sockaddr_in` 结构前，必须首先调用 `bzero` 将结构的各个字段置 0，然后依次对 `sockaddr_in` 结构中的 `sin_family`、`sin_port` 和 `sin_addr` 字段赋值，并且还要注意，`sin_port` 字段是网络字节序的，因此赋值前必须调用 `htons` 进行网络字节序的转换。

13.2.1 socket 函数

应用程序调用 `socket` 来创建一个新的套接字，这个套接字可用于网络通信。下面是函数 `socket` 的定义：

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

int socket(int family, int type, int protocol);
```



该函数返回所创建的套接字的描述符,如为 -1,则代表创建套接字失败。该函数的第 1 个参数 `family` 指明协议,表 13.1 列出了常见的协议类型。函数的第 2 个参数 `type` 指明服务的类型,常见的服务类型如表 13.2 所示。

表 13.1 常见协议类型

协议类型	含 义
AF_INET	IPv4 协议
AF_INET6	IPv6 协议
AF_LOCAL	UNIX 域协议
AF_ROUTE	路由套接字

表 13.2 常见服务类型

服务类型	含 义
SOCK_STREAM	字节流套接字
SOCK_DGRAM	数据报套接字
SOCK_RAW	原始套接字

但是,并非所有的协议类型和服务类型的组合都是有效的,表 13.3 给出了这些组合和对应的真正协议。标 Y 表示组合有效,N 表示无效。

表 13.3 协议类型与服务类型的组合

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE
SOCK_STREAM	TCP	TCP	Y	N
SOCK_DGRAM	UDP	UDP	Y	N
SOCK_RAW	IPv4	IPv6	N	Y

第 3 个参数 `protocol` 指明使用的协议号,如为 0 则表示由前两个参数决定的默认协议号。一般来说都是将第 3 个参数置 0。

13.2.2 connect 函数

在创建了一个套接字后,客户机程序调用 `connect` 以便同远程服务器建立一个主动的连接。下面是函数 `connect` 的定义:

```
#include <sys/socket.h>
int connect (int sockfd, const struct sockaddr * servaddr,
             socklen_t addrlen);
```

该函数调用成功时返回 0,如返回 -1,则建立连接失败。在该函数中,第 1 参数 `sockfd` 为函数 `socket` 返回的套接字的描述符。第 2 个参数 `servaddr` 为指向 `sockaddr` 结构的指针,第 3 个参数 `addrlen` 为 `sockaddr` 结构的长度。客户端调用 `connect` 后将阻塞,直到服务器同意建立连接或拒绝建立连接。

客户机程序在调用 `connect` 与服务器建立连接前,可以调用 `bind` 绑定一个端口地址,也可以不调用 `bind` 函数,建议不要调用 `bind`。

13.2.3 bind 函数

在创建套接字后,它还没有任何关于端口地址的概念。应用程序要调用 `bind` 以便为每



```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr * cliaddr, socklen_t * addrlen);
```

该函数调用成功就返回新建的套接字的描述符,如返回 - 1,便是发生了错误。函数的第 1 个参数 sockfd 为套接字的描述符。第 2 个参数为返回的指向 sockaddr 结构的指针,在该结构中,包含了客户机的 IP 地址和协议端口号。第 3 个参数为返回的指向 sockaddr 结构的长度的指针。

在该函数中,参数 addrlen 为值 - 结果参数,在调用 accept 前首先要将其初始化为套接字地址结构的大小。accept 的典型用法如下所示:

```
struct sockaddr_in cliaddr;
socklen_t len;
char buf[256];

len = sizeof(cliaddr);
accept(sockfd, (struct sockaddr*) &cliaddr, &len);
printf("connecting from %s, port %d\n",
       inet_ntop(AF_INET, &cliaddr.sin_addr, buf, sizeof(buf)),
       ntohs(cliaddr.sin_port));
```

在上面的代码中,最后一行用于输出客户机的 IP 地址和端口号。

13.2.6 read 函数

当客户机与服务器建立连接后,双方就可以调用函数 read 从套接字获取输入。下面是函数 read 的定义:

```
#include <unistd.h>
int read (int sockfd, char * buff, int buflen);
```

函数返回获取的输入字节数,如返回 - 1,便是发生了错误。函数的第 1 个参数为用于传输数据的套接字的描述符。第 2 个参数为存放输入字符的数组的指针。第 3 个参数指明获取输入的最大字节数,一般为 buff 数组的大小。

13.2.7 recv 函数

函数 recv 与 read 相似,也是从套接字获取输入。下面是函数 recv 的定义:

```
#include <sys/socket.h>
int recv (int sockfd, char * buff, int buflen, int flags);
```

函数返回获取的输入字节数,如返回 - 1,便是发生了错误。函数的第 1 个参数为用于传输数据的套接字的描述符。第 2 个参数为存放输入字符的数组的指针。第 3 个参数指明



获取输入的最大字节数,一般为 buff 数组的大小。第 4 个参数为控制比特,表明是否接受带外数据和是否预览报文。

13.2.8 recvfrom 函数

与上面的 read,recv 不同,函数 recvfrom 用于非连接状态下的数据传输。recvfrom 从套接字获取下一个报文,并记录发送者的地址。下面是 recvfrom 的定义:

```
#include <sys/socket.h>
int recvfrom (int sockfd, char * buff, int buflen, int flags,
              struct sockaddr * from, int * fromlen)
```

函数返回获取的输入字节数,如返回 -1,便是发生了错误。函数的第 1 个参数为由函数 socket 创建的套接字的描述符。第 2 个参数为存放输入字符的数组的指针。第 3 个参数指明获取输入的最大字节数,一般为 buff 数组的大小。第 4 个参数为控制比特,表明是否接受带外数据和是否预览报文。第 5 个参数为指向存放发送方地址的结构体的指针。第 6 个参数为指向发送方地址大小的指针。

13.2.9 write 函数

当客户机与服务器建立连接后,双方就可以调用函数 write 将数据输出到套接字。下面是函数 write 的定义:

```
#include <unistd.h>
int write (int sockfd, char * buf, int buflen);
```

函数返回输出的字节数,如返回 -1,便是发生了错误。函数的第 1 个参数为用于传输数据的套接字的描述符。第 2 个参数为存放输出字符的数组的指针。第 3 个参数为 buff 数组中的字节数。

13.2.10 send 函数

函数 send 与 write 相似,也是将数据输出到套接字。下面是函数 send 的定义:

```
#include <sys/socket.h>
int send (int sockfd, char * buff, int buflen, int flags);
```

函数返回输出的字节数,如返回 -1,便是发生了错误。函数的第 1 个参数为用于传输数据的套接字的描述符。第 2 个参数为存放输出字符的数组的指针。第 3 个参数为 buff 数组中的字节数。第 4 个参数为控制比特,表明是否接受带外数据和是否预览报文。

13.2.11 sendto 函数

与上面的 write,send 不同,函数 sendto 用于非连接状态下的数据传输。sendto 从结构中获取目的地址,并发送报文。下面是函数 sendto 的定义:



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

int sendto (int sockfd, char * buff, int buflen, int flags,
            struct sockaddr * to, int * tolen);
```

函数返回输出的字节数,如返回 -1,便是发生了错误。函数的第 1 个参数为由函数 socket 创建的套接字的描述符。第 2 个参数为存放输出字符的数组的指针。第 3 个参数为 buff 数组中的字节数。第 4 个参数为控制比特,表明是否接受带外数据和是否预览报文。第 5 个参数为指向存放接收方地址的结构体的指针。第 6 个参数为指向接收方地址大小的指针。

13.2.12 close 函数

当应用程序使用完一个套接字后调用函数 close,该函数终止通信,并删除套接字。下面是函数 close 的定义:

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

int close (int sockfd);
```

函数成功返回 0,如返回 -1,便是发生了错误。唯一的参数为要关闭的套接字的描述符。

13.2.13 getsockname 和 getpeername 函数

函数 getsockname 用于得到与套接字关联的本地协议地址,函数 getpeername 用于得到与套接字关联的远程协议地址。这两个函数的定义如下所示:

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

int getsockname (int sockfd, struct sockaddr * localaddr,
                 socklen_t * addrlen);

int getpeername (int sockfd, struct sockaddr * peeraddr,
                 socklen_t * addrlen);
```

在这两个函数中,参数 sockfd 为套接字描述符,参数 localaddr 和 peeraddr 为协议地址,参数 addrlen 为值 - 结果参数,在调用这两个函数前必须对其进行初始化。

这两个函数有以下几个用途:

(1) 在不调用 bind 的客户机程序中,当 connect 成功返回后,可以调用 getsockname 得到内核分配给套接字的本地 IP 地址和端口号。

(2) 在服务器程序调用 accept 成功返回时,可以调用 getpeername 得到客户机的 IP 地址和端口号。



```

else if (inet_pton(AF_INET,argv[1], &servaddr.sin_addr) <= 0)
{
    printf("host error \n");
    exit(1);
}
if (connect(sockfd,(struct sockaddr *) &servaddr,sizeof(servaddr)) < 0)
{
    printf("Error:connect \n");
    exit(1);
}
while((n= read(sockfd,buf,1024)) > 0)
{
    buf[n] = 0;
    printf("From server,now is :");
    if (fputs(buf,stdout) == EOF)
    {
        printf("Error:fputs \n");
        exit(1);
    }
}
if (n < 0)
{
    printf("Error:read \n");
    exit(1);
}
ticks= time(NULL);
snprintf(buf,sizeof(buf),"% .24s \r \n",ctime( & ticks));
printf("At host,now is :");
if (fputs(buf,stdout) == EOF)
{
    printf("Error:fputs \n");
    exit(1);
}
close(sockfd);
exit(0);
}

```

下面给出了程序运行的结果：



```
[msf@linux chapter13] $cc ex13_1.c
[msf@linux chapter13] $a.out linux
From Server,now is :Tue May 3 19:04:29 2000
At host,now is      :Tue May 3 19:04:29 2000
```

13.4 ourhead.h 文件

ourhead.h 文件封装了网络编程的许多细节,包括数十个头文件,函数 readline 以及一些常量的定义等。

下面给出了 ourhead.h 的源文件:

```
/* filename:ourhead.h */
#ifndef _OURHEAD_H
#define _OURHEAD_H

#include <error.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <fcntl.h>
#include <netdb.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/uio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/un.h>
#include <sys/select.h>
#include <poll.h>
#include <string.h>
#include <sys/ioctl.h>
```



```

#include <pthread.h>
#define LISTENQ 1024
#ifndef MAXLINE
#define MAXLINE 4096
#endif
#define BUFFERSIZE 8192
#define min(a,b) ((a)<(b)? (a):(b))
#define max(a,b) ((a)>(b)? (a):(b))
ssize_t readline(int fd,void * vptr,ssize_t maxlen)
{
    ssize_t rc;
    char c,* ptr;

    ptr=vptr;
    for (n=1;n<maxlen;n++)
    {
        again;
        if((rc=read(fd,&c,1))!=1)
        {
            *ptr++=c;
            if (c=='\n')
                break;
        }
        else if (rc==0)
        {
            if (n==1)
                return 0; /* EOF ,no data read */
            else
                break; /* EOF,some data read */
        }
        else
        {
            if (errno==EINTR)
                goto again;
            return (-1);
        }
    }
    *ptr=0;
}

```



14.1.2 客户机程序的简化

为了使编程工作能够得到简化,需要考虑如何构建客户机程序的问题。为了与某个服务器建立连接,客户机程序必须选择一个协议(在这里为 TCP)、得到服务器的 IP 地址、得到所期望的服务所对应的端口号、分配一个套接字并与服务器建立连接。所有上面的一切,都是非常繁琐的,譬如机器名与 IP 地址之间的转换,服务名与端口号之间的转换等,而且有些地方还特别容易出错,如网络字节序问题。

如果能够将建立连接的过程抽象为一个函数,譬如 `tcp_connect`,那么以后的编程工作必然能够得到很大的简化。

函数 `tcp_connect` 完成了机器名与 IP 地址之间的转换,服务名与端口号之间的转换。因此,该函数的参数可以设计成这样:一个参数为服务器的机器名或带点的 IP 地址,另一个参数为服务名或端口号。函数调用成功时返回建立连接的套接字描述符。如果服务名或端口号错误,则返回 -4;服务器名字或 IP 错误,则返回 -3;创建套接字错误,则返回 -2;连接服务器错误,返回 -1。

函数 `tcp_connect` 在文件 `tcp_connect.h` 中定义。下面是 `tcp_connect.h` 文件:

```
#ifndef TCP_CONNECT_H
#define TCP_CONNECT_H
#include "ourhead.h"

int tcp_connect(const char * host, const char * service)
{
    struct hostent * phe;
    struct servent * pse;
    struct sockaddr_in sin;
    int s;
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    if (pse = getservbyname(service, NULL))
        sin.sin_port = pse->s_port;
    else if ((sin.sin_port = htons((u_short)atoi(service))) == 0)
        return -4;
    if (phe = gethostbyname(host))
        memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
    else if ((sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE)
        return -3;
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0)
        return -2;
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
```

服务器的下一步是调用 `fork`, 创建一个新的进程。图 14.4 给出了 `fork` 返回后的状态。注意, 套接字描述符 `listenfd` 和 `connfd` 是父子进程之间共享的。因此, 父进程要关闭 `connfd`, 子进程要关闭 `listenfd`。这时就到了期待的状态: 子进程处理与客户的连接, 父进程继续调用 `accept`, 等待下一个连接请求的到来。

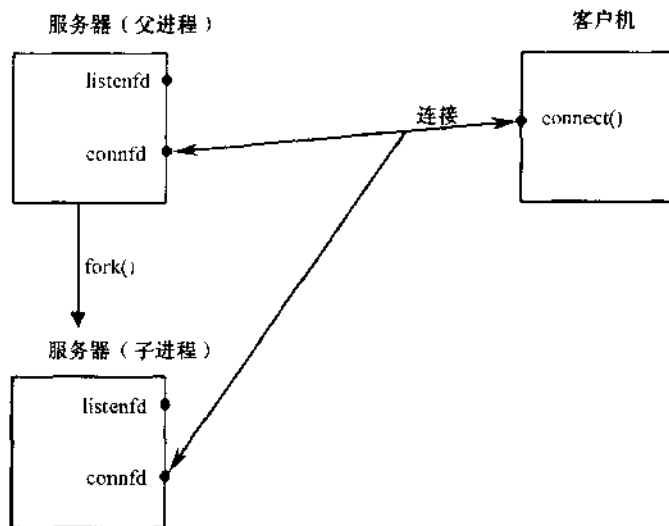


图 14.4 `fork` 返回后

下面给出实现并发服务器的代码。

```
int listenfd, connfd;
listenfd = socket(...);
bind(listenfd, ...);
listen(listenfd, ...);
for (;;)
{
    connfd = accept(listenfd, ...);
    if (fork() == 0)
    {
        close(listenfd);
        ...
        ...
        close(connfd);
        exit(0);
    }
    else
```



```
close(connfd);
```

在调用 `fork` 创建子进程后,子进程立刻调用 `close(listenfd)` 关闭监听套接字;同样,父进程也是立刻调用 `close(connfd)` 关闭连接套接字。为什么要这样做呢?在前面的章节中,已经讲过系统调用 `fork` 的执行情况,子进程拷贝父进程所有的上下文,当然也包括文件标识符, Linux 系统对文件标识符的关闭决定于它的引用计数,如引用计数不为 0,则系统就不关闭此文件。在本程序中,父进程与子进程都有 `connfd` 与 `listenfd` 这两个套接字标识符,而对于子进程, `listenfd` 毫无用处;同样,对于父进程, `connfd` 也是毫无用处。如果不调用两个 `close`,则当子进程终止时,系统也不关闭 `connfd`,因为它的引用计数不为 0(因为父进程中的 `connfd` 没有关闭),这不是所希望的,这就是调用两个 `close` 的原因。

14.1.4 ECHO 客户机程序的 TCP 版本

TCP/IP 的 TCP 和 UDP 都提供了 ECHO 服务。ECHO 客户机从标准输入读取一行字符,然后将它通过网络传到服务器,服务器把它接收的数据又返回给客户机。

图 14.5 给出了 ECHO 客户机和服务器的工作方式。

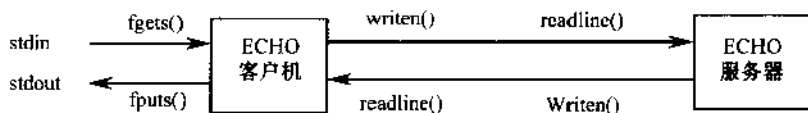


图 14.5 ECHO 客户机和服务器的工作方式

标准的 ECHO 服务采用的端口号为 7,例中采用的端口号为 9999。由于绑定 1024 以下的端口号需要超级用户的权限,因此,选定端口号为 9999,当然,读者也可使用别的端口号。

在 `main` 函数中,首先调用函数 `tcp_connect`,与服务器建立连接;接着从标准输入读一行字符,调用函数 `write` 将这行字符写到服务器端,然后调用函数 `read` 从服务器端读出这行字符,写到标准输出;最后关闭套接字,退出应用程序。

下面给出了 TCP ECHO 的客户端程序:

```
/* filename: ex14_echocli.c */
#include "ourhead.h"
#include "tcp_connect.h"
void usage(char * name)
{
    printf("Usage: %s [hostname or IP address] \n", name);
    return ;
}

int main(int argc, char * * argv)
{

```



```
char * host = "127.0.0.1";
char buf[ MAXLINE + 1 ];
int sockfd, count;
if ( argc != 1 && argc != 2 )
{
    usage(argv[0]);
    exit(1);
}
if ( argc == 2 )
{
    host = argv[1];
}

sockfd = tcp_connect(host, "9999");
while( fgets(buf, MAXLINE, stdin) )
{
    buf[ MAXLINE ] = '\0';
    count = strlen(buf);
    write(sockfd, buf, count);
    if ( read(sockfd, buf, count) < 0 )
    {
        printf("sock read error : %s \n", strerror(errno));
        exit(1);
    }

    fputs(buf, stdout);
}

close(sockfd);
exit(0);
}
```

14.1.5 ECHO 服务器程序的 TCP 版本

由于服务器要处理多个客户的请求,因此,服务器程序要比客户端程序复杂得多。服务器端有以下几个比较微妙的地方需要考虑:

(1) 处理 SIGCHLD 信号的函数中,可以调用函数 `wait` 或 `waitpid`。但是,如果调用 `wait`,则仍然不能避免子进程变为僵尸。譬如,在一个客户机程序中与服务器建立多个连接,然后调用 `exit` 退出进程,同时连接被迫关闭。这时,服务器端就有多个子进程几乎同时退出,也就有多个 SIGCHLD 信号被发送给父进程。而 Linux 系统没有提供信号缓冲,所以多个信号就不会都被处理,而有几个信号就会被丢失。因此,就不能避免子进程中有变成僵尸的。所以采用函数 `waitpid`,并将该函数的第一个参数设置为 -1,代表等待第一个退出的



子进程;第三个参数设置为 WNOHANG,代表即使还有未终止的子进程,该函数也不阻塞。

(2) 进程调用函数 `accept` 失败时,不是直接退出进程,而是看 `errno` 的值是不是 `EINTR`,如果是,则进程不退出。这其实是所有阻塞函数都要考虑的问题,细心的读者或许在上一章的文件 `ourhead.h` 的 `readline` 函数中就看出这种情况。它的原因是这样的:在进程阻塞于 `accept` 时,它可能被别的事情中断,在现在的例子中,最有可能中断进程的事情是子进程退出,发送 `SIGCHLD` 给父进程,当父进程收到此信号时,马上中断 `accept`,转而去执行信号处理函数。这时,对于阻塞的函数 `accept`,内核就使它返回错误 `EINTR`,代表该函数被系统调用中断。此时就应该忽略这个错误。

服务器程序的 `main` 函数中,首先创建套接字,绑定端口号,接着调用 `listen`,`accept` 等待客户机与之建立连接。当有客户机建立连接时,`accept` 便返回新的套接字描述符,然后在子进程中利用这个套接字进行数据传输。对于父进程,仍然调用 `accept` 监听新的客户机与之建立连接。

下面给出了 ECHO 的服务器程序:

```
/* filename: ex14 _echoserv.c */
#include "ourhead.h"
void con _handle(int conrfd)
{
    ssize_t n;
    char buf[MAXLINE + 1];
    for (;;)
    {
        if ((n = readline(conrfd, buf, MAXLINE)) == 0)
            return ;
        if (write(conrfd, buf, n) < 0)
        {
            printf("write error \n");
            exit(1);
        }
    }
    return ;
}

void sig _handle(int signo)
{
    pid_t pid;
    int stat;
    while((pid = waitpid(-1, &stat, WNOHANG)) > 0)
        printf("child %d terminated \n", pid);
    return ;
}
```



```
int main(int argc, char * * argv)
{
    int listenfd, connfd;
    pid_t childpid;
    socklen_t clien;
    struct sockaddr_in cliaddr, servaddr;
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("socket error \n");
        exit(1);
    }
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(9999);
    if (bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0)
    {
        printf("bind error \n");
        exit(1);
    }
    if (listen(listenfd, LISTENQ) < 0)
    {
        printf("listen error \n");
        exit(1);
    }
    signal(SIGCHLD, sig_handle);
    for (;;)
    {
        clien = sizeof(cliaddr);
        if ((connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clien)) < 0)
        {
            if (errno == EINTR)
                continue;
            else
            {
                printf("accept error \n");
                exit(1);
            }
        }
    }
}
```



```
    }  
    if ((childpid = fork()) == 0)  
    {  
        close(listenfd);  
        con_handle(connfd);  
        exit(0);  
    }  
    close(connfd);  
}  
close(listenfd);  
exit(0);  
}
```

下面给出了程序的执行结果：

```
[msf@linux chapter14] $cc -o server ex14_echoserv.c  
[msf@linux chapter14] $server &  
[3] 1510  
[msf@linux chapter14] $cc -o client ex14_echocli.c  
[msf@linux chapter14] $client linux  
Hello  
Hello  
Linux  
Linux  
^D  
child 1521 terminated
```

14.2 UDP 套接字编程

14.2.1 简介

与 TCP 程序类似,UDP 程序也有一定的固定格式,一般来说,服务器进程首先启动,等待客户请求的到来。当客户进程启动后,它直接向服务器请求服务,服务器给出应答。与 TCP 程序的区别是:UDP 程序无需建立连接。

具体说来,服务器首先调用函数 `socket` 创建一个套接字,接着调用函数 `bind` 指定端口号,然后调用 `recvfrom` 接收客户的数据,调用 `sendto` 发送数据给客户。客户机首先调用函数 `socket` 创建一个套接字,然后调用 `sendto` 发送数据给服务器,调用 `recvfrom` 接收服务器的数据。等待双方数据交换完毕,就各自调用函数 `close` 关闭套接字,通知释放建立的连接。如图 14.6 所示。

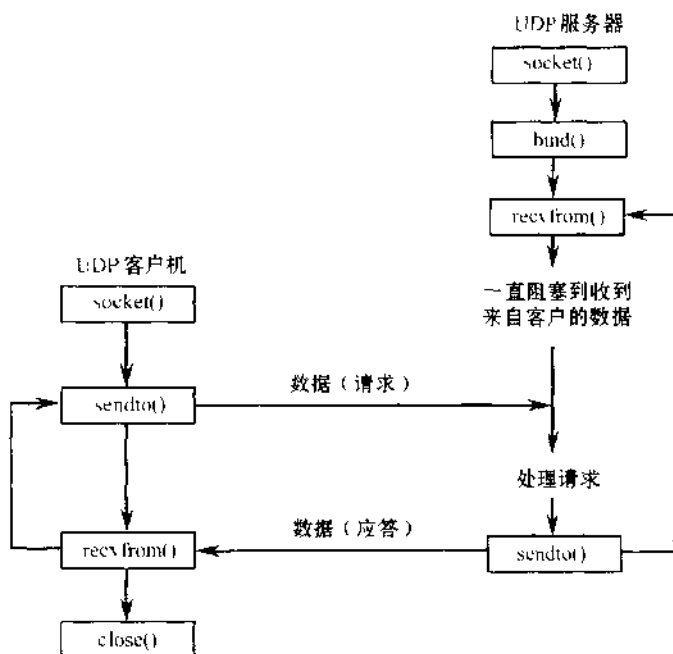


图 14.6 UDP 套接字的函数调用

14.2.2 客户机程序的简化

与 TCP 程序的 `tcp_connect` 类似,也可以编一个名叫 `udp_client` 的函数,处理 UDP 客户机程序中的名字转换、服务名转换等一系列问题,从而使编程工作得以简化。

函数 `udp_client` 的第一个参数 `hostname` 为服务器 IP 地址或机器名,参数 `service` 为服务名或端口号。最后两个参数是由函数 `udp_client` 返回的服务器的套接字地址和地址长度,这两个参数用于以后的 `sendto` 函数。函数的返回值与 `tcp_connect` 相同。

函数 `udp_client` 在文件 `udp_client.h` 中定义,下面是 `udp_client.h` 文件:

```
#ifndef UDP_CLIENT_H
#define UDP_CLIENT_H
#include "ourhead.h"

int udp_client(const char * host, const char * service,
               struct sockaddr * sa, socklen_t * salen)
;

struct hostent * phe;
struct servent * pse;
struct sockaddr_in sin;
int s;
memset( &sin, 0, sizeof(sin) );
sin.sin_family = AF_INET;
```



```

    if (pse = getservbyname(service, NULL))
        sin.sin_port = pse->s_port;
    else if ((sin.sin_port = htons((u_short)atoi(service))) == 0)
        return -4;
    if (phe = gethostbyname(host))
    {
        memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
        memcpy(sa, &sin, sizeof(sin));
        *salen = sizeof(sin);
    }
    else if ((sin.sin_addr.s_addr = inet_addr(host)) != INADDR_NONE)
    {
        memcpy(sa, &sin.sin_addr, sizeof(sin.sin_addr));
        *salen = sizeof(sin);
    }
    else
        return -3;
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s < 0)
        return -2;
    return s;
}
#endif

```

下面是 `udp_client` 的一种典型用法：

```

struct sockaddr_in servaddr;
socklen_t salen;
int sockfd;
sockfd = udp_client("127.0.0.1", "13", (struct sockaddr *)&servaddr, &salen);
sendto(sockfd, ..., (struct sockaddr *)&servaddr, salen);

```

14.2.3 ECHO 客户机程序的 UDP 版本 1

ECHO 的 UDP 版本与 TCP 版本类似，ECHO 客户机从标准输入读取一行字符，然后将它通过网络传到服务器，服务器把它接收的数据又返回给客户机。标准的 ECHO 服务采用的端口号为 7，在例子中采用的端口号为 9999。

在 `main` 函数中，首先调用函数 `socket` 创建套接字，接着从标准输入读一行字符，调用函数 `sendto` 将这行字符写到服务器端，然后调用函数 `recvfrom` 从服务器端读出这行字符，写到标准输出；最后关闭套接字，退出应用程序。



```

        printf("can not send data: %s \n",strerror(errno));
        exit(1);
    }

    count = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
    if (count < 0)
    {
        printf("can not recv data: %s \n",strerror(errno));
        exit(1);
    }

    recvline[count] = 0;
    fputs(recvline, stdout);
}

close(sockfd);
exit(0);
}

```

下面给出了程序的运行结果：

```

[msf@linux chapter14] $cc -o server ex14 _ echoserv1.c
[msf@linux chapter14] $server &
[3] 1544
[msf@linux chapter14] $cc -o client ex14 _ echocli1.c
[msf@linux chapter14] $client linux
Hello
Hello
Linux
Linux
^D
[msf@linux chapter14] $

```

14.2.4 ECHO 客户机程序的 UDP 版本 2

可以看出 ECHO 客户机程序还是比较麻烦,要进行很多的指针转换,而且要调用 `sendto`、`recvfrom` 这些有很多参数的函数。能不能再简化一点呢? 其实是能够的,只要调用 `connect` 函数,然后就可以调用 `send`、`recv` 来发送和接收数据了。这样一简化,UDP 程序好像与 TCP 程序一样了,唯一不同的是 TCP 程序中调用 `tcp_connect`,而在 UDP 程序中调用 `udp_connect`。函数 `udp_connect` 是这一节将要介绍的函数。

函数 `udp_connect` 在文件 `udp_connect.h` 中定义,下面是 `udp_connect.h` 文件:

```

#ifndef UDP_CONNECT_H
#define UDP_CONNECT_H

```



```
#include "ourhead.h"

int udp_connect(const char * host, const char * service)

{
    struct hostent * phe;
    struct servent * pse;
    struct sockaddr_in sin;
    int s;
    memset( &sin, 0, sizeof( sin) );
    sin.sin_family = AF_INET;
    if ( pse = getservbyname( service, NULL ) )
        sin.sin_port = pse->s_port;
    else if ( (sin.sin_port = htons( (u_short)atoi( service) ) ) == 0 )
        return -4;
    if ( phe = gethostbyname( host ) )
        memcpy( &sin.sin_addr, phe->h_addr, phe->h_length );
    else if ( (sin.sin_addr.s_addr = inet_addr( host ) ) == INADDR_NONE )
        return -3;
    s = socket( AF_INET, SOCK_DGRAM, 0 );
    if ( s < 0 )
        return -2;
    if ( connect( s, ( struct sockaddr * ) &sin, sizeof( sin ) ) < 0 )
        return -1;
    return s;
}

#endif
```

从函数 `udp_connect` 的实现可以看出,它与 `tcp_connect` 长得太像了,几乎不能将它与 `tcp_connect` 区别开来。它们俩的唯一区别是:调用函数 `socket` 时的第三个参数不同,一个为 `SOCK_DGRAM`,表示使用 UDP 协议;另一个为 `SOCK_STREAM`,表示使用 TCP 协议。函数 `udp_connect` 虽然调用了 `connect`,但却是无连接的,因为 UDP 协议是无连接的。为了使读者充分了解 UDP 与 TCP 的区别,所以,先讨论 `udp_client`,然后才讨论 `udp_connect`。

同样,利用 `udp_connect` 实现的 ECHO 客户机程序的 UDP 版本也与 TCP 版本差不多,唯一的区别是 UDP 版本调用 `udp_connect`,而 TCP 版本调用的是 `tcp_connect`。下面是使用 `udp_connect` 的 ECHO 客户机程序。

```
/* filename: ex14_echocli2.c */
#include "ourhead.h"
#include "udp_connect.h"
```



```
[msf@linux chapter14] $cc -o server ex14 _ echoserv1.c
[msf@linux chapter14] $server &
[3] 1546
[msf@linux chapter14] $cc -o client ex14 _ echocli2.c
[msf@linux chapter14] $client linux
Hello
Hello
Linux
Linux
^D
[msf@linux chapter14] $
```

14.2.5 ECHO 服务器程序的 UDP 版本

UDP 版本的服务器程序相对 TCP 版本要简单多了。服务器程序的 main 函数中, 首先创建套接字, 绑定端口号, 接着调用 `recvfrom` 等待数据的到来。当有客户机向它发送数据时, `recvfrom` 便返回, 同时得到客户机的地址, 然后再利用这个地址向客户机发送数据。由于不用建立连接, 因此也用不着创建子进程。

下面给出了 ECHO 服务器程序:

```
/* filename : ex14 _ echoserv1.c */
#include "ourhead.h"
int main(int argc, char * * argv)
{
    int sockfd;
    struct sockaddr _ in servaddr, cliaddr;
    int re;
    char buf[ MAXLINE + 1 ];
    int count;
    socklen _ t clien;
    sockfd = socket( AF _ INET, SOCK _ DGRAM, 0 );
    if ( sockfd < 0 )
    {
        printf( "can not create socket: %s \ n", strerror( errno ) );
        exit( 1 );
    }

    bzero( & servaddr, sizeof( servaddr ) );
    servaddr. sin _ family = AF _ INET;
    servaddr. sin _ port = htons( 9999 );
```




```

servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
re = bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
if (re < 0)
{
    printf("can not bind address: %s \n", strerror(errno));
    exit(1);
}
for (;;)
{
    cliilen = sizeof(cliaddr);
    count = recvfrom(sockfd, buf, MAXLINE, 0, &cliaddr, &cliilen);
    if (count < 0)
    {
        printf("can not recv data: %s \n", strerror(errno));
        exit(1);
    }
    re = sendto(sockfd, buf, count, 0, &cliaddr, cliilen);
    if (re < 0)
    {
        printf("can not send data: %s \n", strerror(errno));
        exit(1);
    }
}
close(sockfd);
exit(0);
}

```

14.3 小 结

本章围绕客户机/服务器的编程模式介绍了 TCP 和 UDP 编程的一般步骤,并且分别给出了 ECHO 服务器的 TCP 和 UDP 版本。在下一章中,还要继续讨论复杂服务器的设计问题。

第 15 章 复杂服务器设计

15.1 多协议服务器

15.1.1 多协议服务器简介

在大多数情况下,一个给定的服务器处理针对一个特定服务的请求,这些请求是通过特定的传输协议发来的。例如:一个提供 ECHO 服务的计算机系统往往要运行两个服务器,一个服务器处理来自 UDP 的请求,一个服务器处理来自 TCP 的请求。

为每个协议使用一个单独服务器的优点是便于控制:一个系统管理员可以通过控制系统所运行的服务器来很容易地控制计算机所提供的协议。每个协议使用一个服务器的主要缺点是重复。由于很多服务器可以通过 UDP 也可以通过 TCP 来访问,因此每种服务都需要两个服务器。此外,由于 UDP 和 TCP 服务器都是用相同的算法来计算响应,它们都要包含执行计算所需要的代码。如果两个程序都含有执行某个给定服务的代码,软件管理和排错就变得冗长乏味了。当改正程序中的差错或者为适应新发布的系统软件而需要改变服务器时,程序员必须保证两个服务器程序保持一致。此外,为保证 TCP 和 UDP 服务器在任何时间都能够准确地提供相同的服务,系统管理员必须小心谨慎地协调它们的执行。为每个协议单独运行服务器的另一个缺点来自对资源的利用:多个服务器进程不必要地消耗了进程的许多项目以及其他系统资源。只要回忆一下 TCP/IP 标准所定义的那几十种服务,问题的严重性就很清楚了。

多协议服务器由一个进程构成,这个进程既可以在 TCP 也可以在 UDP 之上使用异步 I/O 通信。在多协议服务器中,服务器最初就打开两个套接字,一个使用无连接的传输(UDP),另一个使用面向连接的传输(TCP)。接着,服务器使用异步 I/O 等待其中一个套接字就绪。如果 TCP 套接字就绪,服务器就调用函数 `accept` 与客户机建立连接。如果 UDP 套接字就绪,服务器就调用 `recvfrom` 读取数据,并且记下客户机的 IP 地址及端口号,然后再调用 `sendto` 发送响应给客户机。

15.1.2 select 函数

为了能够实现异步 I/O,一般都使用函数 `select`,该函数的定义如下:

```
#include <sys/select.h>
#include <sys/time.h>
int select(int maxfdl, fd_set * read_set, fd_set * write_set,
           fd_set * except_set, const struct timeval * timeout);
```



该函数的第 1 个参数 `mxfdl` 为集合中文件描述符的数目,也就是最大的文件描述符值加 1;第 2 个参数为输入的文件描述符集合的地址;第 3 个参数为输出的文件描述符集合的地址;第 4 个参数为异常处理的文件描述符集合的地址;第 5 个参数为最大等待的时间值。如果不要等待输入的文件描述符,则可将第 2 个参数设置为 `NULL`,另外两个参数也可设置为 `NULL`;如果该函数一直等到有准备好的文件描述符才返回,可将第 5 个参数设置为 `NULL`。该函数返回已经准备好了的文件描述符个数,如返回 0,表示超时,返回 -1,表示出错。

该函数中,出现了 `fd_set` 类型,`fd_set` 为整数数组,对 `fd_set` 的操作,主要有以下几种:

```
void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
```

其中,`FD_ZERO` 将 `fd_set` 的各位置零;`FD_SET` 将某一位位置为 1;`FD_CLR` 将某一位清零;`FD_ISSET` 用于看某一位是否被置 1,如果是,返回 1,否则返回 0。下面给出了一个 `fd_set` 的典型用法:

```
fd_set fdset;
FD_ZERO(&fdset);
FD_SET(1, &fdset);
FD_SET(3, &fdset);
FD_CLR(1, &fdset);
```

另外,函数 `select` 中还用到了结构 `timeval`,该结构的定义如下所示:

```
struct timeval
{
    long tv_sec;
    long tv_usec;
};
```

结构 `timeval` 表示一时间值,其中 `tv_sec` 为秒数,`tv_usec` 为微秒数。

函数 `select` 的典型用法是:先调用 `FD_CLR`,将 `fd_set` 的各位清零,接着对于需要等待的文件标志符,调用 `FD_SET` 将对应的位设置为 1,然后调用 `select` 一直阻塞,直到有文件标志符已经准备好或者超时,这时候,系统将已经准备好的位设置为 1,没有准备好的位设置为 0,就可以调用 `FD_ISSET` 来检查哪位已经准备好。

15.1.3 ECHO 服务器的 TCP/UCP 合并版

为了实现多协议,需要在服务器的 `main` 函数中创建两个套接字,分别处理 TCP 和 UDP 的请求;接着,用函数 `select` 实现异步 I/O,等待准备好了的套接字,如果 TCP 套接字



```

char buf[MAXLINE + 1];
struct sockaddr_in cliaddr, servaddr;
/* tcp socket */
if ((listenfd1 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    printf("can not create socket: %s \n", strerror(errno));
    exit(1);
}

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(9999);
if (bind(listenfd1, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
{
    printf("can not bind address: %s \n", strerror(errno));
    exit(1);
}

if (listen(listenfd1, LISTENQ) < 0)
{
    printf("can not listen on 9999 port: %s \n", strerror(errno));
    exit(1);
}

/* udp socket */
if ((listenfd2 = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
{
    printf("can not create socket: %s \n", strerror(errno));
    exit(1);
}

if (bind(listenfd2, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
{
    printf("can not bind address: %s \n", strerror(errno));
    exit(1);
}

signal(SIGCHLD, sig_handle);
FD_ZERO(&rset);
for (;;)
{
    FD_SET(listenfd1, &rset);

```



```

    }
    printf("can not recv data: %s \n", strerror(errno));
    exit(1);
}
if (sendto(listenfd2, buf, count, 0, &cliaddr, clien) < 0)
{
    printf("can not send data: %s \n", strerror(errno));
    exit(1);
}
}
}
close(listenfd2);
exit(0);
}

```

15.1.4 ECHO 客户机的 TCP/UDP 合并版

客户机程序基本上是将两个客户机程序合并起来,提供了一个选项, -u 代表访问 UDP 服务,否则,就访问 TCP 服务。源程序如下所示:

```

/* filename: ex15_echocli1.c */
#include "ourhead.h"
#include "tcp_connect.h"
#include "udp_connect.h"
void usage(char * name)
{
    printf("Usage: %s [-t/-u] [hostname or IP address] \n", name);
    return ;
}
int main(int argc, char * * argv)
{
    char * host = "127.0.0.1";
    char buf[MAXLINE + 1];
    int sockfd, count;
    char c;
    if(argc >= 2)
        host = argv[argc - 1];
    if ((c = getopt(argc, argv, "u:t:")) != -1)
    {
        switch(c)

```



```
        case 'u':
            sockfd = udp_connect(host, "9999");
            break;
        case 't':
        default:
            sockfd = tcp_connect(host, "9999");
            break;
    }

    while(fgets(buf, MAXLINE, stdin))
    {
        buf[MAXLINE] = '\0';
        count = strlen(buf);
        write(sockfd, buf, count);
        if (read(sockfd, buf, count) < 0)
        {
            printf("sock read error : %s \n", strerror(errno));
            exit(1);
        }
        fputs(buf, stdout);
    }

    close(sockfd);
    exit(0);
}
```

下面给出了程序运行情况:

```
[msf@linux chapter15] $cc -o server ex15__echoserv1.c
[msf@linux chapter15] $server &
^C_ 1609
[msf@linux chapter15] $cc -o client ex15__echocli1.c
[msf@linux chapter15] $client -u linux
Hello
Hello
^D
[msf@linux chapter15] $client -t linux
Linux
Linux
```



```
^D
child 1643 terminated
[msf@linux chapter15]$
```

由运行情况来看,当为 TCP 服务器时,程序输出 child XXX terminated,而 UDP 服务器没有这一行的输出。其原因是:TCP 服务器为多进程,而 UDP 服务器为单进程。

15.2 多服务服务器

15.2.1 多服务服务器简介

在前面讨论了这样的问题:一个使用多协议的服务器如何有助于节约系统资源,并使程序易于维护。把多个服务合并到一个多服务服务器中的动机,同设计多协议服务器的动机是相同的,它们具有同样的优点。

多服务服务器既可以使用无连接的也可以使用面向连接的传输协议。

15.2.2 ECHO, DAYTIME 多服务服务器程序

为了举一个多服务服务器的例子,再引进一个 DAYTIME 服务,服务器程序得到系统的时间,发送给客户机。下面例子的服务器方提供 ECHO 和 DAYTIME 两种服务,ECHO 服务用端口号 9999, DAYTIME 服务用端口号 9998, ECHO 服务用客户机根据选项访问这两项服务。在本小节中,提供的都为面向连接的服务,在下一小节中,将把两种协议与两种服务结合起来。

下面给出了服务器方的程序:

```
/* filename: ex15_serv2.c */
#include "ourhead.h"
void con_handle(int sockfd, int semo)
{
    ssize_t n;
    char buf[MAXLINE + 1];
    time_t ticks;
    if (semo == 1)
    {
        for (;;)
        {
            if ((n = readline(sockfd, buf, MAXLINE)) == 0)
                return;
            if (write(sockfd, buf, n) < 0)
            {
                printf("write error \n");
            }
        }
    }
}
```



```
        exit(1);
    }

    else
    {
        ticks = time(NULL);
        snprintf(buf, MAXLINE, "%.24s \r \n", ctime(&ticks));
        write(sockfd, buf, strlen(buf));
    }

    return ;
}

void sig_handle(int signo)
{
    pid_t pid;
    int stat;
    while((pid = waitpid(-1, &stat, WNOHANG)) > 0)
        printf("child %d terminated \n", pid);
    return ;
}

int main(int argc, char * * argv)
{
    int listenfd1, listenfd2, connfd1, connfd2;
    pid_t childpid;
    socklen_t clien1, clien2;
    int maxfd;
    fd_set rset;
    int re;
    struct sockaddr_in cliaddr1, cliaddr2, servaddr1, servaddr2;
    /* ECHO */
    if ((listenfd1 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("can not create socket; %s \n", strerror(errno));
        exit(1);
    }

    bzero(&servaddr1, sizeof(servaddr1));
    servaddr1.sin_family = AF_INET;
    servaddr1.sin_addr.s_addr = htonl(INADDR_ANY);
```




```

servaddr1.sin_port = htons(9999);
if (bind(listenfd1, (struct sockaddr *) &servaddr1, sizeof(servaddr1)) < 0)
{
    printf("can not bind address: %s \n", strerror(errno));
    exit(1);
}

if (listen(listenfd1, LISTENQ) < 0)
{
    printf("can not listen on 9999 port: %s \n", strerror(errno));
    exit(1);
}

/* DAYTIME */
if ((listenfd2 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    printf("can not create socket: %s \n", strerror(errno));
    exit(1);
}

bzero(&servaddr2, sizeof(servaddr2));
servaddr2.sin_family = AF_INET;
servaddr2.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr2.sin_port = htons(9998);
if (bind(listenfd2, (struct sockaddr *) &servaddr2, sizeof(servaddr2)) < 0)
{
    printf("can not bind address: %s \n", strerror(errno));
    exit(1);
}

if (listen(listenfd2, LISTENQ) < 0)
{
    printf("can not listen on 9999 port: %s \n", strerror(errno));
    exit(1);
}

signal(SIGCHLD, sig_handle);
FD_ZERO(&rset);
for (;;)
{
    FD_SET(listenfd1, &rset);
    FD_SET(listenfd2, &rset);
    maxfd = max(listenfd1, listenfd2) + 1;

```



```
if(select(maxfd, &rset,NULL,NULL) < 0)
{
    if (errno == EINTR)
        continue;
    else
    {
        printf("select error : %s \n",strerror(errno));
        exit(1);
    }
}

if (FD_ISSET(listenfd1, &rset))
{
    cliilen1 = sizeof(cliaddr1);
    if ((connfd1 = accept(listenfd1, (struct sockaddr *) &cliaddr1, &cliilen1)) < 0)
    {
        if (errno == EINTR)
            continue;
        else
        {
            printf("accept error \n");
            exit(1);
        }
    }

    if ((childpid = fork()) == 0)
    {
        close(listenfd1);
        con _ handle(connfd1, 1);
        exit(0);
    }

    close(connfd1);
}

if(FD_ISSET(listenfd2, &rset))
{
    cliilen2 = sizeof(cliaddr2);
    if ((connfd2 = accept(listenfd2, (struct sockaddr *) &cliaddr2, &cliilen2)) < 0)
    {
        if (errno == EINTR)
            continue;
```



```

        else
        {
            printf("accept error \n");
            exit(1);
        }
    }
    if ((childpid = fork()) == 0)
    {
        close(listenfd2);
        con _ handle(connfd2, 2);
        exit(0);
    }
    close(connfd2);
}

close(listenfd1);
close(listenfd2);
exit(0);
}

```

15.2.3 ECHO, DAYTIME 多服务客户机程序

多服务的客户机程序与多协议的客户机程序类似,通过选项来分别访问这两个服务, - 1 代表访问 ECHO 服务, - 2 代表访问 DAYTIME 服务,默认为 ECHO 服务。下面给出了客户机程序:

```

/* filename: ex15 _ cli2.c */
#include "ourhead.h"
#include "tcp _ connect.h"
void usage(char * name)
{
    printf("Usage: %s [ - 1 / - 2 ] [hostname or IP address] \n", name);
    return ;
}

int main(int argc, char * * argv)
{
    char * host = "127.0.0.1";
    char buf[ MAXLINE + 1 ];
    int sockfd, count;
}

```



```
char c;
int is_echo = 1;
if (argc != 1 && argc != 2 && argc != 3)
{
    usage(argv[0]);
    exit(1);
}
if (argc >= 2)
    host = argv[argc - 1];
if ((c = getopt(argc, argv, "1:2")) != -1)
{
    switch(c)
    {
        case '2':
            printf("time");
            sockfd = tcp_connect(host, "9998");
            is_echo = 0;
            break;
        case '1':
        default:
            sockfd = tcp_connect(host, "9999");
            break;
    }
}
if (is_echo)
{
    while (fgets(buf, MAXLINE, stdin))
    {
        buf[MAXLINE] = '\0';
        count = strlen(buf);
        write(sockfd, buf, count);
        if (read(sockfd, buf, count) < 0)
        {
            printf("sock read error : %s \n", strerror(errno));
            exit(1);
        }
        fputs(buf, stdout);
    }
}
```



```

        close(sockfd);
        exit(0);
    }
    else
    {
        if ((count = read(sockfd, buf, MAXLINE)) < 0)
        {
            printf("sock read error : %s \n", strerror(errno));
            exit(1);
        }
        buf[count] = 0;
        fputs(buf, stdout);
        close(sockfd);
        exit(0);
    }
}

```

下面为程序的运行结果：

```

[msf@linux chapter15] $ cc -o server ex15 _echoserv2.c
[msf@linux chapter15] $ server &
[3] 1661
[msf@linux chapter15] $ cc -o client ex15 _echocli2.c
[msf@linux chapter15] $ client -1 linux
Hello
Hello
^D
child 1678 terminated
[msf@linux chapter15] $ client -2 linux
time: Tue May 23 19:27:26 2000
child 1648 terminated
[msf@linux chapter15] $

```

15.2.4 多协议多服务服务器程序

有了多协议服务器与多服务服务器，很自然地就想把这两者结合起来，这样就产生了多协议多服务服务器。在这一小节中，就讨论多协议多服务服务器程序，服务器提供的服务仍然是 ECHO 和 DAYTIME 两种，协议也仍然为 TCP 与 UDP 两种。在本例子中，用一个结构来描述服务，用一个数组来定义各种服务，数组中的每一项对应于采用某种协议的某种服务，因为这样的程序更加容易扩展，可以很容易地将两种服务扩充为几十种服务。



下面给出了服务器方的程序：

```
/* filename: ex15 serv3.c */
#include "ourhead.h"
#define SERV_NUM 4
#define PRO_TCP 0
#define PRO_UDP 1
struct ourservice
{
    char * ser_name;
    int ser_id;
    int ser_pro;
    int ser_port;
};

struct ourservice service[SERV_NUM] =
{
    {"echo", 0, PRO_TCP, 9999},
    {"echo", 1, PRO_UDP, 9999},
    {"daytime", 2, PRO_TCP, 9998},
    {"daytime", 3, PRO_UDP, 9998}
};

void init(int sockfd[])
{
    int count;
    static struct sockaddr_in servaddr[SERV_NUM];
    for (count = 0; count < SERV_NUM; count++)
    {
        if(service[count].ser_pro == PRO_TCP)
        {
            if ((sockfd[count] = socket(AF_INET, SOCK_STREAM, 0)) < 0)
            {
                printf("can not create socket: %s \n", strerror(errno));
                exit(1);
            }
        }
        else
        {
            if ((sockfd[count] = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
            {
                printf("can not create socket: %s \n", strerror(errno));
            }
        }
    }
}
```



```

        exit(1);
    }
}

bzero(&servaddr[count], sizeof(servaddr[count]));
servaddr[count].sin_family = AF_INET;
servaddr[count].sin_addr.s_addr = htonl(INADDR_ANY);
servaddr[count].sin_port = htons(service[count].ser_port);
if (bind(sockfd[count], (struct sockaddr *)&servaddr[count],
        sizeof(servaddr[count])) < 0)
{
    printf("can not bind address: %s \n", strerror(errno));
    exit(1);
}

if(service[count].ser_pro == PRO_TCP)
{
    if (listen(sockfd[count], LISTENQ) < 0)
    {
        printf("can not listen on 9999 port: %s \n", strerror(errno));
        exit(1);
    }
}

}

void con_handle(int sockfd, int semo)
{
    ssize_t n;
    char buf[MAXLINE + 1];
    time_t ticks;
    int clien;
    struct sockaddr cliaddr;
    switch(semo)
    {
    case 0:
        for (;;)
        {
            if ((n = readline(sockfd, buf, MAXLINE)) == 0)
                return;
            if (write(sockfd, buf, n) < 0)

```



232



```

    for(i=0;i<SERV_NUM;i++)
        close(listenfd[i]);
    exit(0);
}

```

15.2.5 多协议多服务客户机程序

关于客户机程序,只要将前面的程序综合起来即可,下面就是客户机程序的代码。

```

/* filename: ex15_cli3.c */
#include "ourhead.h"
#include "tcp_connect.h"
#include "udp_connect.h"
void usage(char * name)
{
    printf("Usage: %s [-0/-1/-2/-3] [hostname or IP address] \n", name);
    return ;
}

int main(int argc, char * * argv)
{
    char * host="127.0.0.1";
    char buf[MAXLINE+1];
    int sockfd, count;
    char c;
    int is_echo=0;
    if (argc!=1 && argc!=2 && argc!=3)
    {
        usage(argv[0]);
        exit(1);
    }

    if(argc>=2)
        host=argv[argc-1];
    if ((c=getopt(argc,argv,"0:1:2:3"))!=1)
    {
        switch(c)
        {
            case '0':
                sockfd=tcp_connect(host,"9999");
                is_echo=1;
                break;

```



```
        case '1':
            sockfd = udp_connect(host, "9999");
            is_echo = 1;
            break;
        case '2':
            sockfd = tcp_connect(host, "9998");
            break;
        case '3':
            sockfd = udp_connect(host, "9998");
            write(sockfd, buf, 1);
            break;
    }

    if (is_echo)
    {
        while(fgets(buf, MAXLINE, stdin))
        {
            buf[MAXLINE] = '\0';
            count = strlen(buf);
            write(sockfd, buf, count);
            if (read(sockfd, buf, count) < 0)
            {
                printf("sock read error : %s \n", strerror(errno));
                exit(1);
            }
            fputs(buf, stdout);
        }
        close(sockfd);
        exit(0);
    }
    else
    {
        if ((count = read(sockfd, buf, MAXLINE)) < 0)
        {
            printf("sock read error : %s \n", strerror(errno));
            exit(1);
        }

        buf[count] = 0;
    }
}
```



```
        fputs(buf, stdout);  
        close(sockfd);  
        exit(0);  
    }  
}
```

下面给出了程序的运行结果：

```
[msf@linux chapter15] $cc -o server ex15__echoserv3.c  
[msf@linux chapter15] $server &  
^3] 1734  
[msf@linux chapter15] $cc -o client ex15__echocli3.c  
[msf@linux chapter15] $client -0 linux  
Hello  
Hello  
^D  
child 1743 terminated  
[msf@linux chapter15] $client -1 linux  
Linux  
Linux  
^D  
[msf@linux chapter15] $client -2 linux  
time: Tue May 23 19:27:26 2000  
child 1746 terminated  
[msf@linux chapter15] $client -3 linux  
time: Tue May 23 19:27:30 2000  
[msf@linux chapter15] $
```

15.3 小 结

为了节约系统资源,常常使用多协议多服务服务器来代替单一的服务器。在本章中详细讨论了多协议多服务服务器的设计和实现,并给出了多协议多服务服务器的例子。

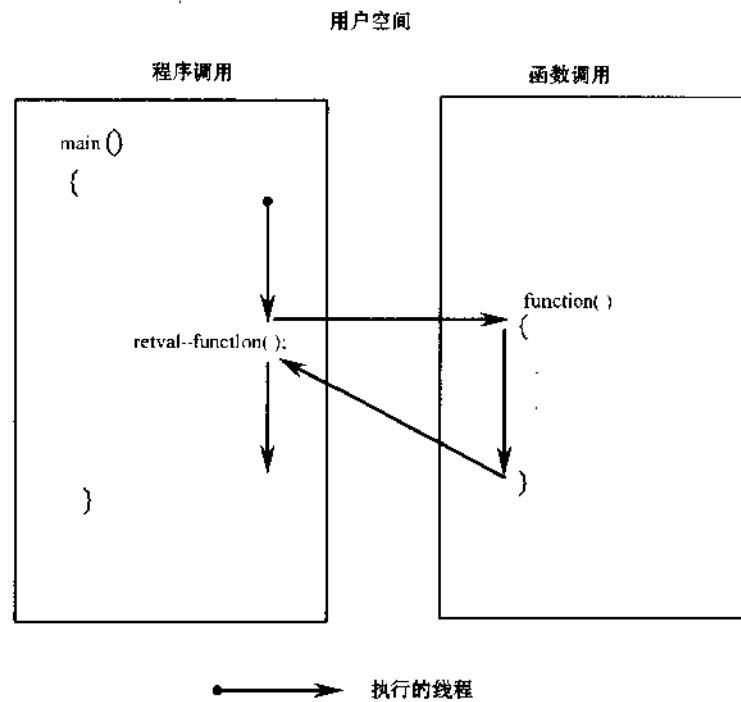


图 16.1 调用普通函数的执行过程

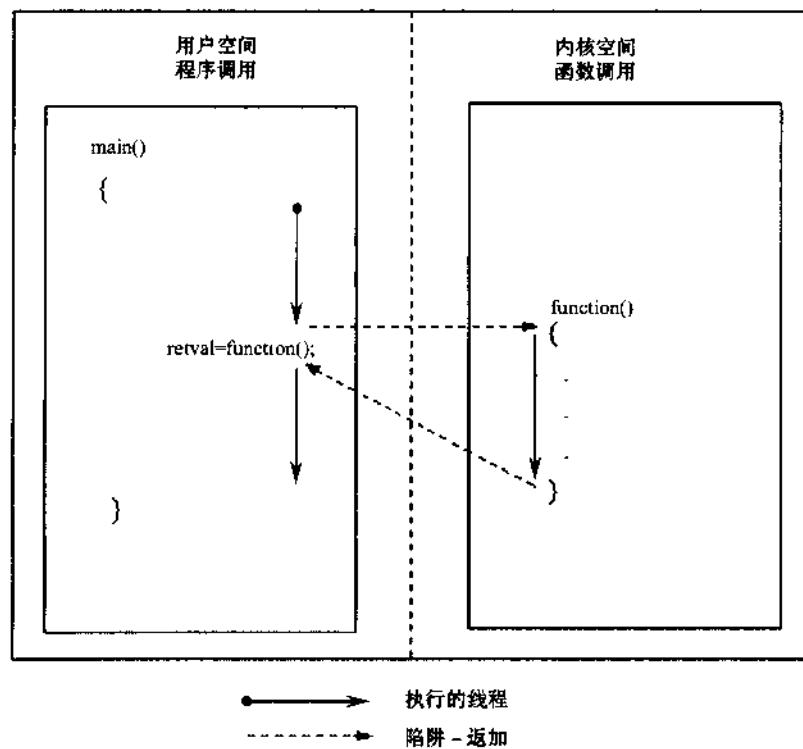


图 16.2 系统调用的执行过程



系统调用是进入内核入口的陷阱,它引起调用执行线程的阻塞。在内核中带堆栈执行的单独线程将执行系统调用。当陷阱返回时,原执行线程将取消阻塞,继续执行下一条语句。

现在重新回到远程过程调用的问题上来,首先看下面程序的执行过程:

```
main()

{
    struct inargs inargs;
    struct outargs outargs;
    int retval;
    ...
    ...
    ...
    retval = remote_fun( &inargs, &outargs );
    ...
    ...
    ...
}
```

这一段程序与前一段没有太大的区别,只是在程序中调用了远程机上的函数 `remote_fun`。图 16.3 给出了该程序的执行线程。像系统调用一样,这个远程调用产生一个新的线程(此时处于远程服务器的地址空间中),调用者被阻塞直到新线程完毕为止。

实际调用要比开始出现的情况复杂得多:客户机程序与被称为客户程序存根(client stub)的附加代码一起编译,以形成单个进程。客户程序存根负责转换参数并将它们装配成报文以适合网络传播。转换为消息的过程称为“整理参数”。程序实际上调用的是客户程序存根中的函数。客户程序存根被用来对参数打包以便以后使用。客户程序存根通过将参数转换为与机器无关的格式来完成整理,这样,不同结构的机器可以共享。标准的独立于机器的数据格式称为外部数据表示(XDR)。客户程序存根可以作为内核中负责传送消息到远程服务器的网络函数的陷阱。然后,客户程序存根就等待回答消息的到来。

同样,被远程调用函数也与被称为服务器程序存根(server stub)的附加代码一起编译。当客户的请求通过网络到达时,远程主机的内核将这个请求传送到等待的服务器程序存根中。服务器程序存根反向整理参数并像一个普通函数调用那样调用请求的服务。

当服务函数返回时,服务器程序存根将返回值整理为适当的网络报文,并且调用服务器内核的系统调用,请求在网络上传送给客户主机的应答。内核将这个报文传送给正在等待的客户程序存根,该客户程序存根整理这些报文并将它们像传送普通返回值一样传送给客户机。

对调用者来说,RPC 机制是透明的。客户程序所见的都是一个对客户程序存根的普通函数的调用,而底层的网络通信隐藏在视线之外。在服务器端,服务器函数像普通函数那样被调用,这是因为服务器程序存根是服务器进程的一部分。网络上传输请求与返回值的底层机制称为传输协议。

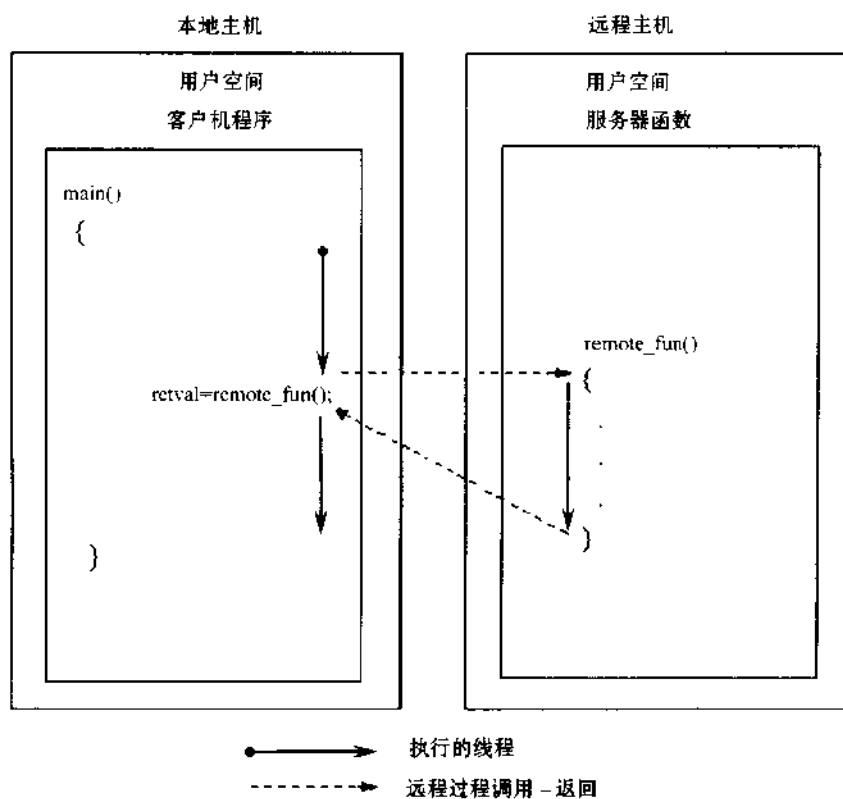


图 16.3 远程过程调用的执行过程

图 16.4 给出了远程过程调用的基本步骤。

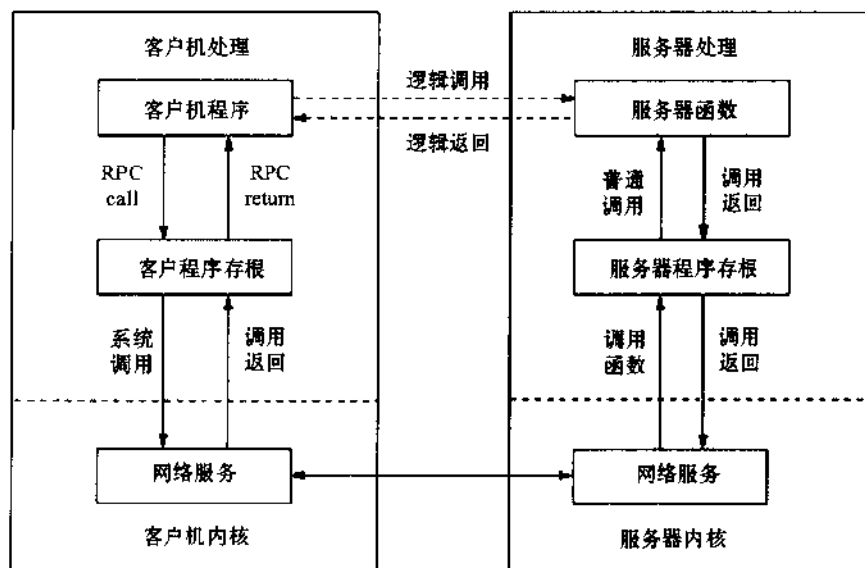


图 16.4 远程过程调用

16.2 外部数据表示(XDR)

XDR 是数据描述与编码的标准。XDR 协议对于在不同体系的计算机之间进行数据传输非常有用。XDR 属于 ISO 表示层。

XDR 用语言描述数据结构并且仅用于描述数据。

16.2.1 XDR 工作原理

在对 XDR 有了基本了解后,让我们看一下 XDR 的基本工作原理。XDR 的工作原理如图 16.5 所示。

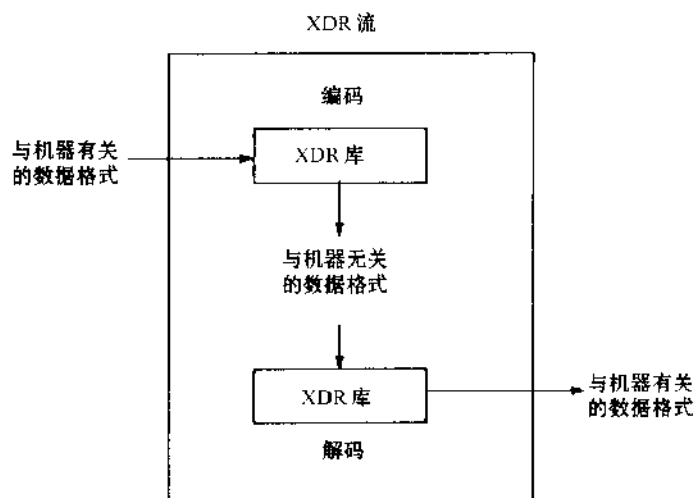


图 16.5 XDR 的工作原理

16.2.2 初始化 XDR 流

函数 `xdrstdio_create` 用于初始化一个 XDR 流,下面是该函数的定义:

```
#include<rpc/xdr.h>
#include<rpc/types.h>
void xdrstdio_create(XDR* handle, FILE * fp, enum xdr_op op);
```

函数 `xdrstdio_create` 中,参数 `fp` 为指向 `FILE` 结构的指针。参数 `op` 为枚举类型,取值为 `XDR_ENCODE` 或 `XDR_DECODE`。`XDR_ENCODE` 操作将数据编码后输出到流, `XDR_DECODE` 操作将把流上的数据解码。参数 `handle` 为指向 XDR 结构的指针,在以后的操作中,都要用到此指针。

16.2.3 释放 XDR 流

函数 `xdr_destroy` 用于释放已经分配的 XDR 流,参数 `handle` 为指向 XDR 句柄的指针



```
#include<rpc/xdr.h>
void xdr_destroy(XDR * handle)
```

16.2.4 整数的 XDR 表示

一个 XDR 符号整数是一个 32 位的数,编码范围为 $[-2147483648, 2147483647]$ 。整数以 2 的补码方式表示,最高有效位(MSB)和最低有效位字节(LSB)分别为 0 和 3。

整数用以下方式声明:

```
int identifier;
```

整数的编码为图 16.6 所示。

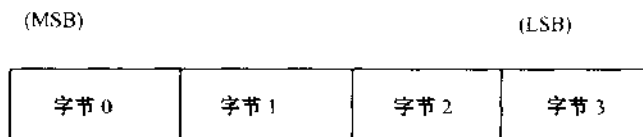


图 16.6 整数的编码

转换的函数为:

```
bool_t xdr_int(XDR * handle, int * p);
```

其中 handle 为调用 xdrstdio_create 生成的,参数 p 为指向要转换的整数的指针。如转换成功,则返回 TRUE,否则返回 FALSE。

16.2.5 无符号整数的 XDR 表示

一个 XDR 无符号整数是一个 32 位的数,编码范围为 $[0, 4294967295]$ 。该整数以无符号二进制数表示,最高有效位(MSB)和最低有效位字节(LSB)分别为 0 和 3。

无符号整数用以下方式声明:

```
unsigned int identifier;
```

无符号整数的编码为图 16.7 所示:

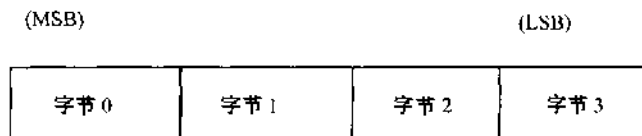


图 16.7 无符号整数的编码

转换的函数为:

```
bool_t xdr_u_int(XDR * handle, uint_t * p);
```

其中 handle 为调用 xdrstdio_create 生成的,参数 p 为指向要转换的无符号整数的指



针。如转换成功,则返回 TRUE,否则返回 FALSE。

16.2.6 枚举型的 XDR 表示

枚举与符号整数的表示方法相同,主要用于描述整数的子集

枚举数据用以下方式声明:

```
enum name = constant, ...; identifier;
```

举例

```
enum {BOY = 1, GIRL = 2} sex;
```

编码与整数相同。

转换的函数为:

```
bool_t xdr_enum(XDR *handle, enum_t *p);
```

其中 handle 为调用 xdrstdio_create 生成的,参数 p 为指向要转换的枚举类型的指针。如转换成功,则返回 TRUE,否则返回 FALSE。

16.2.7 布尔量的 XDR 表示

布尔量为取值为 0 或 1 的整数

枚举数据用以下方式声明:

```
bool identifier;
```

等价于

```
enum {FALSE = 0, TRUE = 1} identifier;
```

编码与整数相同。

转换的函数为:

```
bool_t xdr_enum(XDR *handle, bool_t *p);
```

其中 handle 为调用 xdrstdio_create 生成的,参数 p 为指向要转换的布尔类型的指针。如转换成功,则返回 TRUE,否则返回 FALSE。

16.2.8 浮点数的 XDR 表示

浮点数编码采用 IEEE 标准中规格化的单精度浮点数的标准。用下面三个字段来描述单精度浮点数:

S: 数字的符号。0 和 1 分别表示正和负,占一位。

E: 数字的阶码。该字段占 8 位。阶码偏 127。

F: 数字尾数的小数部分,基为 2。该字段占 23 位。

浮点数用以下方式声明:

```
float identifier;
```



编码如图 16.8 所示。

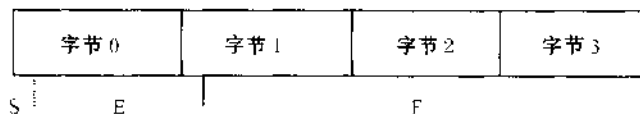


图 16.8 浮点数的编码

转换的函数为：

```
bool_t xdr_float(XDR *handle, float *p);
```

其中 `handle` 为调用 `xdrstdio_create` 生成的, 参数 `p` 为指向要转换的浮点数的指针。如转换成功, 则返回 `TRUE`, 否则返回 `FALSE`。

16.2.9 双精度浮点数的 XDR 表示

双精度浮点数编码采用 IEEE 标准中规格化的双精度浮点数的标准。用下面三个字段来描述单精度浮点数：

- S: 数字的符号。0 和 1 分别表示正和负, 占 1 位。
 - E: 数字的阶码。该字段 11 位。阶码偏 1023。
 - F: 数字尾数的小数部分, 基为 2。该字段占 52 位。
- 浮点数用以下方式声明：

```
double identifier;
```

编码如图 16.9 所示。

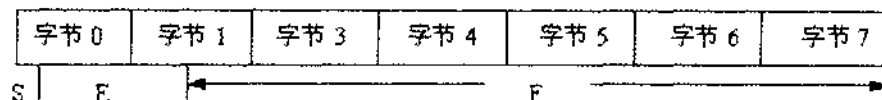


图 16.9 双精度浮点数的编码

转换的函数为

```
bool_t xdr_double(XDR *handle, double *p);
```

其中 `handle` 为调用 `xdrstdio_create` 生成的, 参数 `p` 为指向要转换的双浮点数的指针。如转换成功, 则返回 `TRUE`, 否则返回 `FALSE`。

16.2.10 字符的 XDR 表示

字符用以下方式声明：

```
char identifier;
```

字符的编码为图 16.10 所示。

用四个字节表示, 前三个字节为 0, 第四个字节即为该字符。



(MSB)

(LSB)



图 16.10 字符的编码

转换的函数为：

```
bool_t xdr_char(XDR *handle, char *p);
```

其中 `handle` 为调用 `xdrstdio_create` 生成的, 参数 `p` 为指向要转换的字符的指针。如转换成功, 则返回 `TRUE`, 否则返回 `FALSE`。

16.2.11 字符串的 XDR 表示

本标准定义了 n 个 ASCII 字节的串, 它的计数 n 按无符号整数编码, 并放在串的头。串中的字节 b 总是在串中的字节 $b+1$ 之前, 而串中的字节 0 总跟在串长度之后。 n 字节后面填充足够的 (0 到 3) 零字节 r , 以保证总字节数为 4 的倍数。

字符串用以下方式声明：

```
string identifier < m >;
```

或

```
string identifier < >;
```

常数 m 表示串可容纳的字节数的上界。如没有声明 m , 那么它以 $2^{32}-1$ 作为最大长度。常量 m 通常可在协议中找到。如归档协议说明文件名不能超过 255 个字节。

字符串的编码如图 16.11 所示。

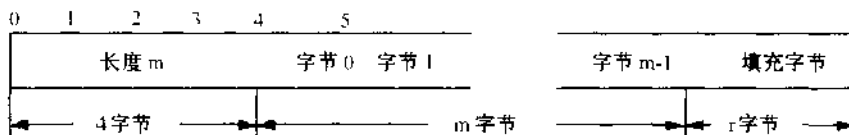


图 16.11 字符串的 XDR 表示

转换的 API 为：

```
bool_t xdr_string(XDR *handle, char **pp, uint_t maxsize);
```

其中 `handle` 为调用 `xdrstdio_create` 生成的, 参数 `pp` 为指向要转换的字符串的指针, `maxsize` 为字符串的最大长度。如转换成功, 则返回 `TRUE`, 否则返回 `FALSE`。

16.2.12 定长数组的 XDR 表示

元素编号为 0 到 $m-1$ 的定长数组按其元素的自然顺序 0 到 $m-1$ 的特点进行编码。每个元素的长度均为 4 的倍数。



定长数组用以下方式声明：

```
type - name identifier[m];
```

定长数组的编码如图 16.12 所示。

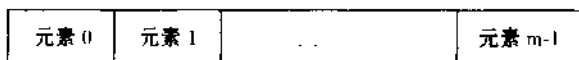


图 16.12 定长数组的编码

转换的函数为：

```
bool_t xdr_vector(XDR * handle, char * p, uint_t size,
                  uint_t elsize, xdrproc_t fn);
```

其中 handle 为调用 xdrstdio_create 生成的，参数 p 为指向要转换的定长数组的指针。参数 size 为数组中元素的个数。参数 elsize 为每一个元素所占的字节数。参数 fn 为能为单个元素编解码的函数名（如为符号整数数组，则 fn 为 xdr_int）。如转换成功，则返回 TRUE，否则返回 FALSE。

16.2.13 变长数组的 XDR 表示

变长数组将元素计数 m（无符号整数）放在数组的第一个元素之前，数组从元素 0 开始，直到 m-1 结束。

变长数组用以下方式声明：

```
type - name identifier<m>;
```

或

```
type - name identifier<>;
```

常数 m 表示串可容纳的字节数的上界。如没有声明 m，那么它以 $2^{32}-1$ 作为最大长度。

变长数组的编码如图 16.13 所示。

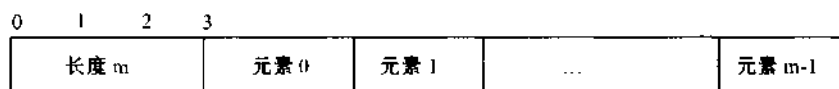


图 16.13 变长数组的编码

要转换变长数组，首先要得到变长数组的长度，函数 xdr_array 可以得到数组长度。

```
bool_t xdr_array(XDR * handle, char * * pp, uint_t * sizep,
                 uint_t maxsize, uint_t elsize, xdrproc_t fn);
```

其中 handle 为调用 xdrstdio_create 生成的，参数 pp 为指向要转换的变长数组的指针。参数 sizep 为指向数组所占字节数的指针。参数 maxsize 为数组中的最大元素个数。参数



```

int main(int argc, char * * argv)
{
    FILE * fp;
    XDR xh;
    struct utmp * up;
    int err=0;
    if (argc != 2)
        usage(argv[0]);
    if(utmpname("/var/run/utmp") != 0)
    {
        printf("utmpname error: %s \n", strerror(errno));
        exit(1);
    }

    if ((fp = fopen(argv[1], "w")) == NULL)
    {
        printf("error: cannot open file %s \n", argv[1]);
        exit(1);
    }

    xdrstdio_create(&xh, fp, XDR_ENCODE);
    while((up = getutent()) != NULL)
    {
        if (xdr_vector(&xh, up->ut_user, sizeof(up->ut_user), sizeof(char),
                      (xdrproc_t)xdr_char) == FALSE)
        {
            err = 1;
            break;
        }

        if (xdr_vector(&xh, up->ut_id, sizeof(up->ut_id), sizeof(char),
                      (xdrproc_t)xdr_char) == FALSE)
        {
            err = 2;
            break;
        }

        if (xdr_vector(&xh, up->ut_line, sizeof(up->ut_line), sizeof(char),
                      (xdrproc_t)xdr_char) == FALSE)
        {
            err = 3;
        }
    }
}

```



```
        break;
    }

    if (xdr_vector(&xh, up->ut_line, sizeof(up->ut_line), sizeof(char),
                  (xdrproc_t)xdr_char) == FALSE)
    {
        err = 3;
        break;
    }

    if (xdr_short(&xh, &up->ut_pid) == FALSE)
    {
        err = 4;
        break;
    }

    if (xdr_short(&xh, &up->ut_type) == FALSE)
    {
        err = 5;
        break;
    }

    if (xdr_short(&xh, &up->ut_exit.e_termination) == FALSE)
    {
        err = 6;
        break;
    }

    if (xdr_short(&xh, &up->ut_exit.e_exit) == FALSE)
    {
        err = 7;
        break;
    }

    if (xdr_long(&xh, &up->ut_time) == FALSE)
    {
        err = 8;
        break;
    }

    if (err)
    {
        printf("error %d: XDR converting data \n", err);
        exit(1);
    }
}
```



```
exit(0);
```

```
;
```

16.3 小 结

本章讨论了远程过程调用(RPC)的一些概念, RPC 程序与普通程序的区别。重点讲解了外部数据表示(XDR)。在下一章中,将讨论 RPC 程序的实现。

第 17 章 RPC 编程

17.1 RPC 编程简介

RPC 程序避免了网络界面的细节,为程序员提供了网络服务,而不要求他们了解基本网络的存在方式及功能

一个 RPC 程序通常包括下列主要部分:

(1) “RPC 程序的程序号,版本号和过程号”——RPC 程序用程序号,程序版本和过程号来唯一地标志通过 RPC 调用的过程。

(2) “网络选择”——可以编写在指定传输和传输类型操作的程序,也可以编写在系统或用户选择的传输上操作的程序

(3) “rpcbnd 设施”——rpcbnd 是用来连接网络服务和通过网络地址的设施

(4) “外部数据表示 XDR”——在 RPC 客户机方和服务器方之间传送的数据按 XDR 传输语法编码。

17.1.1 RPC 程序号、版本号和过程号

每个 RPC 过程由程序号、版本号和过程号唯一标志。

程序号标志一组相关的远程过程,每一个过程具有不同的过程号。每个程序还有一个版本号,因此对远程服务进行小的改动(如增加一个新的过程)时,就不必指定一个新的程序号

RPC 程序按照一定的规则指定程序号,规则如表 17.1 所示。

表 17.1 RPC 程序的程序号规则

程序号范围	描 述
0x00000000 - 0x1FFFFFFF	由 Sun 公司定义,提供特定服务
0x20000000 - 0x3FFFFFFF	由程序员自己定义,提供本地服务或用于调试
0x40000000 - 0x5FFFFFFF	用于短时间使用的程序,例如回调程序
0x60000000 - 0xFFFFFFFF	保留程序号

17.1.2 网络选择

网络选择是一种简单的方法,通过它,用户和应用程序可以根据最喜爱的和可用的传输动态地选择所用的传输。这基于两种机制,一个是/etc/netconfig 数据库,它列出了主机可用的传输并定义了它们的类型;另一个是可选的环境变量 NETPATH,该变量允许用户根据意



愿,在/etc/netconfig 中选择应用程序所能接受的可用的传输。

/etc/config 文件有若干行,每一行对应于一个可用的传输。下面给出了/etc/netconfig 文件:

```
#
# The "Network Configuration" File.
#
# Each entry is of the form:
#
#      <network_id> <semantics> <flags> <protofamily> <protoname> \
#      <device> <nametoaddr_libs>
#
# The "-" in <nametoaddr_libs> for inet family transports indicates
# redirection to the name service switch policies for "hosts" and
# "services". The "-" may be replaced by nametoaddr libraries that
# comply with the SVr4 specs, in which case the name service switch
# will not be used for netdir_getbyname, netdir_getbyaddr,
# gethostbyname, gethostbyaddr, getservbyname, and getservbyport.
#
udp      tpi_clts      v    inet      udp      /dev/udp      -
tcp      tpi_cots_ord v    inet      tcp      /dev/tcp      -
rawip    tpi_raw      -    inet      -        /dev/rawip    -
ticlts   tpi_clts      v    loopback -        /dev/ticlts   straddr.so
ticotsord tpi_cots_ord v    loopback -        /dev/ticotsord straddr.so
ticots   tpi_cots     v    loopback -        /dev/ticots   straddr.so
```

现在对文件/etc/config 作简要的介绍:

(1) 每一项含有一个标志符(第一个字段),它给出了网络的标志,通常通过该标志符识别传输。

(2) 每一项还含有一个或一组标志(第三个字段)用以表明类型。例如,v 标志表明传输是“visible 可见的”。

(3) 最后一个字段命名一个实时的可连接模块,它含有一个与传输有关的名字到地址的转换例程。

(4) 用 rpcbnd 登记服务时要求回环传输。它们是本地传输,只能由本地的客户机和服务器使用,因此比其他的传输更安全。

NETPATH 的格式很简单:是用冒号(:)分隔的一串有序的网络标志(如: ucp: tcp: starlen)。通过设置 NETPATH,用户可以规定应用程序可能尝试的各种网络的次序。如果没有 NETPATH,那么系统默认所有在/etc/netconfig 中规定的可见传输,并按它们在该文件中出现的次序进行尝试。



基于 RPC 的服务通常在运行时取得网络地址的映像,然后用 RPC 登记,而且无论是服务器还是客户机都不能假设使用这些地址。rpcbind 由系统管理员或 RPC 管理员启动,服务器和客户机都要调用 rpcbind。

服务器程序在初始化时,通过主机的 rpcbind 进程,将自己登记在主机的地址服务器登记表中。服务器程序调用 rpcbind 更新登记表,而客户机程序呼叫 rpcbind 查询这些映像。为了找到一个远程程序的地址,客户机向服务器的 rpcbind 进程发送一个 RPC 呼叫报文,如果该进程程序在服务器方,那么 rpcbind 进程返回在 RPC 回应报文中的适当地址。然后,客户机程序可向该地址进行远程调用。

rpcbind 协议提供一个函数 RPCBPROC_CALLIT,通过该函数,rpcbind 能够帮助客户机建立一个远程过程呼叫。客户机程序将传递目标过程的程序号、版本号、过程号和 RPC 呼叫消息的实参。然后 rpcbind 在地址映像中查找目标过程的地址并把一个 RPC 呼叫消息发送给目标进程,该消息中包括从客户机接收到的实参。

当目标过程返回结果时,RPCBPROC_CALLIT 将结果传递给客户机,同时将目标过程的地址返回,这样客户机可直接调用该目标过程。

2. rpcinfo 命令

rpcbind 是一个 shell 命令,用于报告 rpcbind 所知的当前 RPC 登记情况。rpcinfo 也可用于查找所有在指定主机上登记的 RPC 服务,并报告每个登记的通用地址和传输;也可用于在使用 TCP 和 UDP 传输的特定主机上呼叫特定程序的特定版本,并报告是否接收到应答。

17.2 RPC 调用

RPC 编程调用如图 17.1 所示,图中的??? 代表 UDP 或 TCP。在客户机方,首先调用函数 clnt_create 生或一个 CLIENT 结构的指针,然后调用函数 clnt_call 来调用远程过程,最后退出应用程序。在服务器方,首先调用函数 svcudp_create 或 svctcp_create 生成一个 SVCXPRT 结构的指针,然后调用函数 svc_registry 注册提供的服务,最后调用 svc_run。

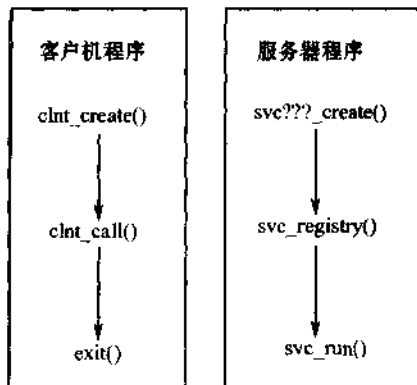


图 17.1 RPC 编程调用



17.2.1 clnt_create 函数

函数 `clnt_create` 用于生成一个指向 `CLIENT` 结构的指针,如返回 `NULL`,则表示出错。该函数的第 1 个参数为服务器名称或 IP 地址,第 2 个参数为程序号,第 3 个参数为版本号,第 4 个参数为协议类型。

```
#include <rpc/clnt.h>
CLIENT *clnt_create(const char *host, const u_long prog,
                    const u_long vers, const char *prot);
```

当该函数出错时,一般都是调用函数 `clnt_pcreateerror`,然后退出应用程序。所以 `clnt_create` 函数的典型用法是:

```
CLIENT *client;
if((client=clnt_create(host,prog,vers,prot)) == NULL)
{
    clnt_pcreateerror("error: create client\n");
    exit(1);
}
```

17.2.2 clnt_call 函数

函数 `clnt_call` 用于调用服务器的某一过程,函数返回 `RPC_SUCCESS`,则代表调用成功,否则出错。下面是函数 `clnt_call` 的定义。

```
#include <rpc/clnt.h>
enum clnt_stat clnt_call(CLIENT *rh, u_long proc, xdrproc_t xargs,
                        caddr_t argsp, xdrproc_t xres, caddr_t resp, struct timeval timeout);
```

该函数的第 1 个参数为 `CLIENT` 结构的指针,一般为调用 `clnt_create` 生成的;第 2 个参数为过程号;第 3 个参数为调用的过程的返回类型,如 `xdr_void`;第 4 个参数为传递给过程的参数地址;第 5 个参数为编解码的函数名;第 6 个参数为编解码函数的参数地址;最后一个参数为给定的超时值,如在给定的时间内服务器方的应答报文还未到达时,函数就出错返回。

17.2.3 svcudp_create 函数

函数 `svcudp_create` 用于生成一个 `SVCXPRT` 结构的指针,返回 `NULL` 则出错。函数的唯一参数为套接字类型,一般为 `RPC_ANYSOCK`。

```
#include <rpc/svc.h>
SVCXPRT *svcudp_create(int sock);
```



17.2.4 svctcp_create 函数

函数 `svctcp_create` 跟 `svculdp_create` 类似,不过前者用来生成无连接的 UDP 服务,后者用来生成面向连接的 TCP 服务。函数的第 1 个参数为套接字类型,一般为 `RPC_ANYSOCK`;第 2 个参数为发送缓冲区大小,如为 0 代表取系统默认值;第 3 个参数为接收缓冲区大小,如为 0 代表系统默认值。

```
#include<rpc/svc.h>
SVCXPRT *svctcp_create( int sock,u_int sendsize,
                        u_int recvsz);
```

17.2.5 svc_register 函数

在调用函数 `svculdp_create` 或 `svctcp_create` 生成 `SVCXPRT` 指针后,就可以调用函数 `svc_register` 向主机注册一项服务,返回 `TRUE` 代表成功,返回 `FALSE` 代表失败。下面是函数 `svc_register` 的定义。

```
#include<rpc/svc.h>
bool_t svc_register(SVCXPRT *xp, u_long prog, u_long vers,
                    dispatch_fn_t dispatch, u_long protocol);
```

该函数的第 1 个参数即为 `SVCXPRT` 指针;第 2 个参数为程序号;第 3 个参数为版本号;第 4 个参数为处理请求的函数名;最后一个参数为协议名,`IPPROTO_UDP` 代表 UDP 协议,`IPPROTO_TCP` 代表 TCP 协议。

17.2.6 svc_run 函数

当服务器程序注册了提供的服务后,就调用函数 `svc_run` 等待客户机请求的到来,该函数不返回,也没有参数。

```
#include<rpc/svc.h>
void svc_run(void);
```

17.2.7 svc_sendreply 函数

当服务器得到客户机请求的结果时,服务器程序就要调用函数 `svc_sendreply` 向客户机发送报文。函数返回 `TRUE` 则调用成功,`FALSE` 则失败。下面是函数 `svc_sendreply` 的定义。

```
#include<rpc/svc.h>
bool_t svc_sendreply (SVCXPRT *xp, xdrproc_t xdr_results,
                      caddr_t xdr_location);
```



函数的第 1 个参数为 SVCXPRT 指针;第 2 个参数为编解码函数名称;第 3 个参数为结果的地址。

17.3 远程计算器

在这一小节中,介绍一个远程计算器的程序,客户机从标准输入得到要运算的表达式,接着将表达式封装成报文发送给服务器,服务器得到结果后将结果发送给客户机,然后客户机程序打印结果。从运行结果来看,根本就不能感觉到服务器的存在。

17.3.1 头文件 xdr_math.h

利用结构 MATH 封装运算表达式,MATH 的定义如下:

```
struct MATH
{
    int op; /* 0-ADD,1-SUB,2-MUL,3-DIV */
    float arg1;
    float arg2;
    float result;
}
```

编解码用函数 xdr_math,该函数只是简单地把 MATH 结构的各个数据项进行编解码,如下所示:

```
bool_t xdr_math(XDR *xdrsp, struct MATH *resp)
{
    if (! xdr_int(xdrsp, &resp->op))
        return FALSE;
    if (! xdr_float(xdrsp, &resp->arg1))
        return FALSE;
    if (! xdr_float(xdrsp, &resp->arg2))
        return FALSE;
    if (! xdr_float(xdrsp, &resp->result))
        return FALSE;
    return TRUE;
}
```

上面介绍的结构、函数均在文件 xdr_math.h 中定义,该文件除了定义 MATH 结构,xdr_math 函数外,还定义了程序号、版本号以及过程号等常量,完整的文件如下所示:

```
/* filename: xdr_math.h */
#include <rpc/types.h>
```



```
# include "ourhead.h"
struct MATH
{
    int op; /* 0-ADD,1-SUB,2-MUL,3-DIV */
    float arg1;
    float arg2;
    float result;
};

bool_t xdr_math(XDR * xdrsp, struct MATH * resp)
{
    if (! xdr_int(xdrsp, &resp->op))
        return FALSE;
    if (! xdr_float(xdrsp, &resp->arg1))
        return FALSE;
    if (! xdr_float(xdrsp, &resp->arg2))
        return FALSE;
    if (! xdr_float(xdrsp, &resp->result))
        return FALSE;
    return TRUE;
}

#define MATH_PROG ((u_long)0x20000001)
#define MATH_VER ((u_long)1)
#define MATH_PROC ((u_long)1)
#define ADD 0
#define SUB 1
#define MUL 2
#define DIV 3
```

17.3.2 客户机程序

下面接着介绍计算器的客户机程序,该程序有一个参数为远程服务器的名称或 IP 地址。在主函数 main 中,先调用函数 clnt_create 生成一个 CLIENT 结构的指针,接着调用函数 clnt_call 来调用远程过程,然后输出结果。源程序如下所示:

```
/* filename: math_client.c */
#include "xdr_math.h"
#include "ourhead.h"
void usage(char * name)
{
```





```
printf("Usage: %s hostname/IPaddr \n".name);
exit(1);
}

int main(int argc, char * * argv)

{
    struct MATH math;
    struct timeval timeout;
    CLIENT * client;
    enum clnt_stat stat;
    char c;
    if (argc != 2)
        usage(argv[0]);
    printf("choose the operation: \n \t0 - - - ADD \n \t1 - - - SUB \n \t2 - - - MUL \n \t3
        - - - DIV \n");

    c = getchar();
    switch ( c )
    {
        case '0':
            math.op = ADD;
            break;
        case '1':
            math.op = SUB;
            break;
        case '2':
            math.op = MUL;
            break;
        case '3':
            math.op = DIV;
            break;
        default :
            printf("error :operation \n");
            exit(1);
    }

    printf("Input the first number: ");
    scanf("%f", &math.arg1);
    printf("Input the second number: ");
    scanf("%f", &math.arg2);
    client = clnt_create(argv[1], MATH_PROG, MATH_VER, "visible");
```



```

if (client == NULL)
{
    clnt_pcreateerror("error: create client \n");
    exit(1);
}

timeout.tv_sec = 30;
timeout.tv_usec = 0;
stat = clnt_call(client, MATH_PROC, xdr_math, &math, xdr_math, &math, timeout);
if (stat != RPC_SUCCESS)
{
    clnt_perror(client, "Call Failed");
    exit(1);
}

printf("The Result is %.3f \n", math.result);
exit(0);
}

```

17.3.3 服务器程序

服务器程序同时提供 TCP 和 UDP 服务, 首先依次调用函数 `svctcp_create`、`svc_registry` 和 `svcdp_create`、`svc_registry` 注册服务, 然后调用 `svc_run` 函数。源程序如下所示:

```

/* filename: math_server.c */
#include "xdr_math.h"
static void mathprog(struct svc_req * rqstp, SVCXPRT * transp)
{
    struct MATH math;
    switch(rqstp->rq_proc)
    {
        case NULLPROC:
            svc_sendreply(transp, xdr_void, NULL);
            return;
        case MATH_PROC:
            break;
        default:
            svcerr_noproc(transp);
            return;
    }

    memset((char *)&math, 0, sizeof(math));
    if (!svc_getargs(transp, xdr_math, (caddr_t) &math))

```




```
[msf@linux chapter16] $ math client localhost
choose the operation:
0 -- ADD
1 -- SUB
2 -- MUL
3 -- DIV
0
Input the first number : 1.5
Input the second number : 2.5
The Result is : 4.000
[msf@linux chapter16] $
```

17.4 小 结

本章重点讨论了 RPC 编程的各个环节,包括 RPC 程序的程序号、版本号、过程号和网络的选择。重点讲解了 RPC 程序的各个函数调用,在本章的最后,还给出了一个例子:远程计算器。

第 18 章 用 rpcgen 生成分布式程序

18.1 rpcgen 简介

很明显,自己手工实现一个 RPC 服务器程序需要很多代码,而且有很多代码是不变的。为避免不必要的编程,系统为程序员提供了一个工具,利用它可以自动生成 RPC 服务器程序的大多数代码。这个工具即为 `rpcgen`,它的输入为一个规格说明文件,它的输出为一个 C 语言的源程序。规格说明文件包含常量,全局数据类型以及远程过程的声明。`rpcgen` 产生的代码包含了实现客户机和服务器程序所需要的大部分源代码。具体地说,`rpcgen` 为客户机端和服务端生成 stub 过程,它包括参数整理,发送 RPC 报文,参数和结果的外部数据表示以及本地数据表示的转换等。`rpcgen` 的输出在与一个应用程序和程序员编写的少数文件相结合后,便产生了完整的客户机和服务器的程序。

在由 `rpcgen` 生成的源文件中,没有过程的具体实现,程序员必须要手工编辑这些文件而实现这些过程。在大多数情况下,程序员不必考虑具体细节,因此节省了程序员的工作量。

18.2 rpcgen 的输入和输出

`rpcgen` 的输入为一个规格说明文件,该文件一般以 `.x` 作为其扩展名。`rpcgen` 的输出与选项有关,选项 `-a` 告诉 `rpcgen` 生成客户机和服务器源程序,如无该选项,则要程序员自己编写客户机与服务器源程序。另外选项 `-C` 表示使用 ANSI C。假如输入文件为 `file.x`,如果使用命令 `rpcgen -C file.x` 则生成五个文件,`file_xdr.h`,`file.h`,`makefile.file`,`file_svc.c` 和 `file_clnt.c`;如果使用命令 `rpcgen -C -a file.x` 则多生成了两个文件,`file_server.c` 和 `file_client.c`。

表 18.1 给出了各个文件的用处。

表 18.1 `rpcgen` 自动生成的文件

文件名	作 用
<code>makefile.file</code>	该文件用于编译所有客户机,服务器代码
<code>file_clnt.c</code>	该文件包含 client stub,程序员一般不用修改
<code>file_svc.c</code>	该文件包含 server stub,程序员一般不用修改
<code>file.h</code>	该文件包含了从说明中产生的所有 XDR 类型
<code>file_xdr.c</code>	该文件包含了客户机和服务器 stub 所需的 XDR 过滤器,程序员一般不用修改
<code>file_server.c</code>	如果生成此文件,则该文件包含远程服务的 stub
<code>file_client.c</code>	如果生成此文件,则该文件包含了骨架客户机程序



18.3 rpcgen 编程步骤

18.3.1 建立 .x 文件

利用 rpcgen 实现 RPC 编程,必须有一定的步骤。假如要实现像上一章一样的远程计算器,要经历如下步骤:首先,必须决定服务器的程序号、版本号和各个过程的过程号,现在决定该程序的程序号为 0x20000001,版本号为 2,有一个过程,过程号为 1。接着要自己写一个规格说明文件 math.x,如下所示:

```
/* filename: math.x */
const ADD 0;
const SUB 1;
const MUL 2;
const DIV 3;
struct MATH
{
    int op; /* 0-ADD,1-SUB,2-MUL,3-DIV */
    float arg1;
    float arg2;
    float result;
};
program MATH_PROG
|
    version MATH_VER
    {
        struct MATH MATH_PROC(struct MATH) = 1;
    } = 2;
} = 0x20000001;
```

该文件中,常量的声明为 const,例如 const ADD 0 声明 ADD 为 0;结构的声明与 C 语言相同;程序的声明用 program;版本的声明用 version;过程的声明与 C 语言相似,只不过增加了过程号的声明。

18.3.2 运行 rpcgen

这里用如下的命令运行 rpcgen:

```
rpcgen -C -a math.x
```

运行 rpcgen 后,生成了七个文件: math.h、math_xdr.c、math_svc.c、math_clnt.c、makefile.math、math_client.c 和 math_server.c。在生成 RPC 客户机和服务器源程序后,



用如下的命令编译程序：

```
make -f makefile.math
```

运行 make 后,就得到了 RPC 客户机程序码 math_client 和服务程序 math_server。

图 18.1 给出了用 rpcgen 生成 RPC 程序的示意图。

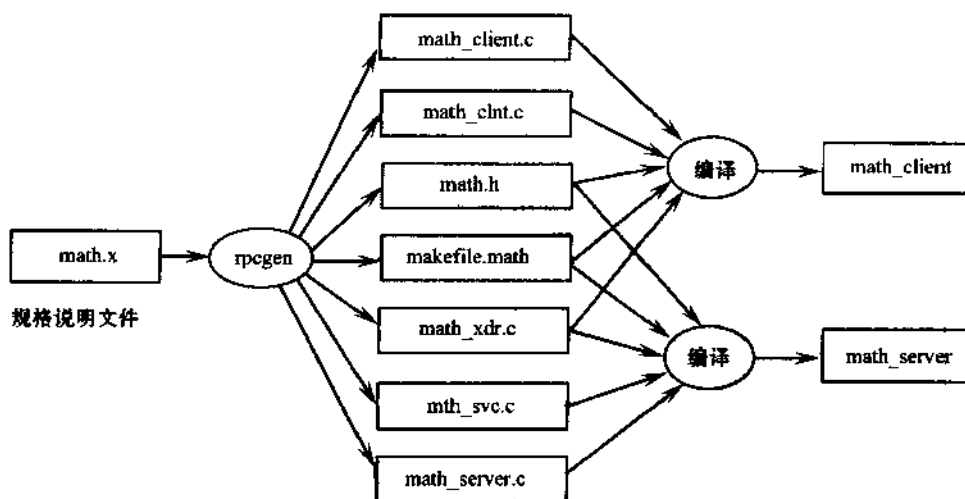


图 18.1 rpcgen 生成 RPC 程序示意图

18.3.3 rpcgen 生成的 math.h 文件

下面给出了 rpcgen 自动生成的文件 math.h,该文件中包含了所声明的所有常量和数据类型 C 合法声明,如定义了 ADD、SUB、MATH_PROG 等常量和 MATH 结构。同时,该文件还定义了远程过程:math_proc_2、math_proc_2_svc 和 math_proc_2_freeresult。函数 math_proc_2 让客户机调用,math_proc 为过程名号的小写形式,2 代表版本号为 2。函数 math_proc_2_svc 为过程 math_proc 的具体实现,这个函数一般为空,需要程序员自己编写代码。

```

/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#ifndef _MATH_H_RPCGEN
#define _MATH_H_RPCGEN
#include <rpc/rpc.h>
#ifdef __cplusplus
extern "C" {
#endif
#endif

```



```
# include "math.h"
bool_t
xdr_MATH (XDR * xdrs, MATH * objp)
{
    register int32_t * buf;
    if (! xdr_int (xdrs, &objp->op))
        return FALSE;
    if (! xdr_float (xdrs, &objp->arg1))
        return FALSE;
    if (! xdr_float (xdrs, &objp->arg2))
        return FALSE;
    if (! xdr_float (xdrs, &objp->result))
        return FALSE;
    return TRUE;
}
```

18.3.5 rpcgen 生成的 math_clnt.c 文件

下面给出了 rpcgen 自动生成的文件 math_clnt.c, 该文件包含了客户机调用的函数 math_proc_2 的具体实现, 同时还定义了超时值为 25 秒, 程序员可以改变这一默认值。

```
# include <memory.h> /* for memset */
# include "math.h"
/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };
struct MATH *
math_proc_2(struct MATH * argp, CLIENT * clnt)
{
    static struct MATH clnt_res;
    memset((char *) &clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, MATH_PROC,
        (xdrproc_t) xdr_MATH, (caddr_t) argp,
        (xdrproc_t) xdr_MATH, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```



```

    if (! svc_getarg (transp, _xdr_argument, (caddr_t) &argument))
    {
        svcerr_decode (transp);
        return;
    }

    result = (* local)((char *) &argument, rqstp);
    if (result != NULL && ! svc_sendreply(transp, _xdr_result, result))
    {
        svcerr_systemerr (transp);
    }

    if (! svc_freeargs (transp, _xdr_argument, (caddr_t) &argument))
    {
        fprintf (stderr, "unable to free arguments");
        exit (1);
    }

    return;
}

int
main (int argc, char * * argv)
{
    register SVCXPRT * transp;
    pmap_unset (MATH_PROG, MATH_VER);
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf (stderr, "cannot create udp service.");
        exit(1);
    }
    if (! svc_register(transp, MATH_PROG, MATH_VER,
                      math_prog_2, IPPROTO_UDP))
    {
        fprintf (stderr, "unable to register (MATH_PROG,
                      MATH_VER, udp).");
        exit(1);
    }

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        fprintf (stderr, "cannot create tcp service.");
        exit(1);
    }
}

```



```
    }
    if (! svc_register(transp, MATH_PROG, MATH_VER,
                       math_prog_2, IPPROTO_TCP))
    {
        fprintf(stderr, "unable to register (MATH_PROG,
                                MATH_VER, tcp).");
        exit(1);
    }
    svc_run();
    fprintf(stderr, "svc_run returned");
    exit(1);
    /* NOTREACHED */
}
```

18.3.7 rpcgen 生成的 math_server.c 文件

下面给出了 rpcgen 自动生成的文件 math_server.c, 该文件包含了空的函数 math_proc_2_svc。必须自己编写代码实现该函数。

```
#include "math.h"
struct MATH *
math_proc_2_svc(struct MATH * argp, struct svc_req * rqstp)
{
    static struct MATH result;
    /*
     * insert server code here
     */
    return &result;
}
```

添加代码后的文件 math_server.c 如下所示:

```
#include "math.h"
struct MATH *
math_proc_2_svc(struct MATH * argp, struct svc_req * rqstp)
{
    static struct MATH result;
    switch(argp->op)
    {
        case ADD:
            result.result = argp->arg1 + argp->arg2;
    }
}
```



```

        break;
    case SUB:
        result.result = argp -> arg1 - argp -> arg2;
        break;
    case MUL:
        result.result = argp -> arg1 * argp -> arg2;
        break;
    case DIV:
        result.result = argp -> arg1 / argp -> arg2;
        break;
    default:
        break;
}

return &result;
}

```

18.3.8 rpcgen 生成的 math_client.c 文件

下面给出了 rpcgen 自动生成的文件 math_client.c, 该文件是一个客户机程序。一般来说, 必须修改该文件的部分内容以达到要求。

```

#include "math.h"
void
math_prog_2(char *host)
{
    CLIENT *clnt;
    struct MATH *result_1;
    struct MATH math_proc_2_arg;
    #ifndef DEBUG
        clnt = clnt_create(host, MATH_PROG, MATH_VER, "udp");
        if (clnt == NULL) {
            clnt_pcreateerror(host);
            exit(1);
        }
    #endif /* DEBUG */
    result_1 = math_proc_2(&math_proc_2_arg, clnt);
    if (result_1 == (struct MATH *) NULL) {
        clnt_perror(clnt, "call failed");
    }
    #ifndef DEBUG

```




```
        clnt _destroy (clnt);
    #endif /* DEBUG */
}

int
main (int argc, char * argv[])
{
    char * host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    math _prog _2 (host);
    exit (0);
}
```

下面显示了修改后的文件 `math _client.c`, 该文件的主要改变是增加了输入和输出部分。输入部分负责接收运算符和两个数, 输出部分负责将运算结果打印出来。

```
# include "math.h"
void
math _prog _2(char * host)
{
    CLIENT * clnt;
    struct MATH * result _1;
    struct MATH math _proc _2 _arg;
    char c;
    printf("choose the operation: \n\n\t
        0 -- ADD \n\t1 -- SUB \n\t2 -- MUL \n\t3 -- DIV \n");
    c = getchar();
    switch ( c )
    {
        case '0':
            math _proc _2 _arg.op = ADD;
            break;
        case '1':
            math _proc _2 _arg.op = SUB;
            break;
        case '2':
```



```

        math_proc_2_arg.op = MUL;
        break;
    case '3':
        math_proc_2_arg.op = DIV;
        break;
    default :
        printf("error :operate \n");
        exit(1);
}

printf("Input the first number: ");
scanf("%f", &math_proc_2_arg.arg1);
printf("Input the second number: ");
scanf("%f", &math_proc_2_arg.arg2);
# ifndef DEBUG
clnt = clnt_create (host, MATH_PROG, MATH_VER, "udp");
if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
}
# endif /* DEBUG */
result_1 = math_proc_2 (&math_proc_2_arg, clnt);
if (result_1 == (struct MATH *) NULL) {
    clnt_perror (clnt, "call failed");
}
# ifndef DEBUG
clnt_destroy (clnt);
# endif /* DEBUG */
printf("The Result is %.3f \n", result_1->result);
}

int
main (int argc, char * argv[])
{
    char * host;
    if (argc < 2) {
        printf ("usage: %s server_host \n", argv[0]);
        exit (1);
    }
    host = argv[1];

```





```
math_prog 2 (host);
exit (0); \
```

18.3.9 rpcgen 生成的 makefile.math 文件

最后介绍的文件是 makefile.math, 这是一个 makefile 文件, 编译时用命令 `make -f makefile.math` 即可生成两个可执行程序: `math_client` 和 `math_server`, 这两个客户机和服务器程序的运行情况与 17 章的相同。

```
# This is a template Makefile generated by rpcgen
# Parameters
CLIENT = math_client
SERVER = math_server
SOURCES_CLNT.c =
SOURCES_CLNT.h =
SOURCES_SVC.c =
SOURCES_SVC.h =
SOURCES.x = math.x
TARGETS_SVC.c = math_svc.c math_server.c math_xdr.c
TARGETS_CLNT.c = math_clnt.c math_client.c math_xdr.c
TARGETS = math.h math_xdr.c math_clnt.c math_svc.c math_client.c math_server.c
OBJECTS_CLNT = $(SOURCES_CLNT.c:%.c=%.o) $(TARGETS_CLNT.c:%.c=%.o)
OBJECTS_SVC = $(SOURCES_SVC.c:%.c=%.o) $(TARGETS_SVC.c:%.c=%.o)
# Compiler flags
CFLAGS += -g
LDLIBS += -lnsl
RPGENFLAGS =
# Targets
all : $(CLIENT) $(SERVER)
$(TARGETS) : $(SOURCES.x)
    rpcgen $(RPGENFLAGS) $(SOURCES.x)
$(OBJECTS_CLNT) : $(SOURCES_CLNT.c) $(SOURCES_CLNT.h) $(TARGETS_CLNT.c)
$(OBJECTS_SVC) : $(SOURCES_SVC.c) $(SOURCES_SVC.h) $(TARGETS_SVC.c)
$(CLIENT) : $(OBJECTS_CLNT)
    $(LINK.c) -o $(CLIENT) $(OBJECTS_CLNT) $(LDLIBS)
$(SERVER) : $(OBJECTS_SVC)
    $(LINK.c) -o $(SERVER) $(OBJECTS_SVC) $(LDLIBS)
clean:
    $(RM) core $(TARGETS) $(OBJECTS_CLNT) $(OBJECTS_SVC) $(CLIENT) $(SERVER)
```



18.4 小 结

本章主要介绍了 `rpcgen` 这个工具,并且在讲解时穿插介绍了远程计算器的例子。通过这一章的学习,大家都会感觉到,`rpcgen` 的确是一个有用的工具,有了它,能够很容易地实现一个 RPC 程序的编制。

第 19 章 RPC 认证

19.1 简介

在前面讲的 RPC 服务都是没有提供认证机制的,也就是说,只要客户机请求服务,服务器就提供服务。这一机制对于某些网络服务,如网络文件系统,是不适合的,因为必须要有很高的网络安全性。

每个 RPC 调用都要服从服务器方的认证风格,同样,RPC 客户机也必须产生和发送符合认证风格的认证参数。系统默认的认证风格为 AUTH_NONE,即不提供认证。

除了 AUTH_NONE 之外,系统还支持表 19.1 中列出的几种认证风格。

表 19.1 系统支持的认证风格

认证风格	对认证风格的描述
AUTH_SYS 或 AUTH_UNIX	一种基于传统的 System V R4.0 V1.0 操作系统进程保护权限证明的认证风格
AUTH_SHORT	某些服务为提高效率使用的 AUTH_SYS 的替代形式。使用 AUTH_SYS 认证的客户机方应准备接收来自服务器方的 AUTH_SHORT 应答验证程序
AUTH_DES	基于 DES 加密技术的一种认证风格

19.2 取得客户机的认证风格

当客户机向服务器请求服务时,会传递一个 svc_req 结构,在这个结构中就包含了客户机的认证风格。下面给出了结构 svc_req 的定义:

```
struct svc_req
{
    u_long      rq_prog;          /* service program number */
    u_long      rq_vers;          /* service protocol version */
    u_long      rq_proc;          /* the desired procedure */
    struct opaque_auth rq_cred;    /* raw creds from the wire */
    caddr_t     rq_clntcred;      /* read only cooked cred */
    SVCXPRT     *rq_xprt;         /* associated transport */
};
```



在 `svc_req` 结构中, `rq_prog` 为过程号, `rq_vers` 为版本号, `rq_proc` 为过程号, `rq_cred` 为有关认证风格的信息, `rq_clntcred` 为客户机认证信息的地址。其中认证信息为一个 `opaque_auth` 结构, 该结构的定义如下所示:

```
struct opaque_auth
{
    enum_t oa_flavor;          /* flavor of auth */
    caddr_t oa_base;          /* address of more auth stuff */
    u_int oa_length;          /* not to exceed MAX_AUTH_BYTES */
};
```

在 `opaque_auth` 结构中, `oa_flavor` 为认证风格, 可以为 `AUTH_NONE`, `AUTH_SYS`, `AUTH_SHORT` 或 `AUTH_DEST` 中的一种。

为了得到认证风格, 可以利用下面的文件代替上一章中的文件 `math_server.c`, 然后重新编译即可。运行程序时, 服务器程序也要在后台运行。在此文件中, 主要的改动是在函数 `math_prog_2` 中添加了讨论 `rqstp->rq_cred.oa_flavor` 的代码, 其余的代码都没有改变。

```
#include "math.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
#ifndef SIG_PF
#define SIG_PF void (*)(int)
#endif
static void
math_prog_2(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union
    {
        struct MATH math_proc_2_arg;
    } argument;
    char *result;
    xdrproc_t _xdr_argument, _xdr_result;
    char * (*local)(char *, struct svc_req *);
    switch (rqstp->rq_cred.oa_flavor)
```



```
|
case AUTH_NONE:
    printf("The AUTH style is AUTH_NONE \n");
    break;
case AUTH_SYS:
    printf("The AUTH style is AUTH_SYS \n");
    break;
case AUTH_SHORT:
    printf("The AUTH style is AUTH_SHORT \n");
    break;
case AUTH_DEST:
    printf("The AUTH style is AUTH_DEST \n");
    break;
default:
    break;
|
switch (rqstp->rq_proc)
{
case NULLPROC:
    (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
    return;
case MATH_PROC:
    _xdr_argument = (xdrproc_t) xdr_MATH;
    _xdr_result = (xdrproc_t) xdr_MATH;
    local = (char * (*)(char *, struct svc_req *)) math_proc_2_svc;
    break;
default:
    svcerr_noproc (transp);
    return;
}
memset ((char *) &argument, 0, sizeof (argument));
if (! svc_getargs (transp, _xdr_argument, (caddr_t) &argument))
{
    svcerr_decode (transp);
    return;
}
result = (* local)((char *) &argument, rqstp);
if (result != NULL && ! svc_sendreply(transp, _xdr_result, result))
```

第 1 章 Linux 系统管理

1.1 登录与注销

1.1.1 登录 Linux

用户在有了自己的账号以后,便可登录到系统中。登录 Linux 系统有下面两种情况:

(1) 通过 Linux 机器的一个终端登录系统,则只要在该终端键入用户名和口令即可注意,用户键入的口令是不会显示出来的。

(2) 利用 Windows 登录远程 Linux 服务器,则要在 Windows 的【开始】按钮中选择【运行】,然后在运行框中键入 telnet hostname,其中 hostname 为用户要登录的主机的名字或 IP 地址。然后屏幕上就会出现与下面类似的提示:

```
TurboLinux release 6.0 (Kunlun)
Kernel 2.2.13 on i686 (linux)
TTY: tty0
```

Login:

在上面的提示中,第一行为系统说明,在这里为 TurboLinux 6.0;第二行为系统内核说明,这里为 2.2.13;第三行为终端说明,这里为 tty0。这时,用户只要键入用户名和口令即可,但要注意在 Linux 系统中大小写是区别对待的。

如果用户名和口令正确,系统就会出现提示符,表明用户登录成功,如下面所示:

```
Last login: Mon Apr 13 22:42:31 from lq
You have new mail.
[msf@linux]$
```

第一行为上次登录说明,这里表示 4 月 13 日(星期一)22 时 42 分 31 秒从主机 lq 登录;如果你有信,则系统会告诉你,如第二行所示;第三行方括号内部为“用户名@主机名”的形式,这里的用户名为 msf,主机名为 Linux,\$ 是缺省的 B shell 提示符。当然,用户也可以根据自己的喜好,定义自己的提示符。

1.1.2 更改口令

登录完成以后,为了安全起见,用户常常需要更改自己的口令。可以用 passwd 命令更改口令,下面给出了一个更改口令的例子:



```
[msf@linux]$ passwd
Changing password for msf
(current) UNIX password:
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully
```

用户的口令一般是 6~8 个字母、数字及某些特殊字符(如下划线“_”、“.”等)组成,设置口令时,应选择不易被人发现或破译的组合。例如不宜采用姓名或某个常用词作口令,以免账号被人盗用。口令一定要记牢,万一忘记了,可以请系统管理员帮助,他可以帮助用户去掉口令,或为用户设置新的口令。一般说来,Linux 系统管理员无法查出用户的口令,这一点与其他一些多用户系统不同。

1.1.3 了解 shell

当你登录后,Linux 就将你置于你的起始目录中,并启动一个 shell 进程来与你交互。shell 实际上为操作系统与用户的交互接口,操作系统通过 shell 接受用户的命令。在 DOS 系统中的 shell 为 command.com,在 Windows 系统中为 explorer.exe。Linux 系统的 shell 与上面两种系统有很大的区别,在上面两种系统中,shell 是固定的;而在 Linux 系统中,用户的 shell 是可以选择的,我们可以选择我们爱好的 shell,甚至我们还可以自己写一个 shell。文件/etc/shells 列出了系统中所有的 shell,下面列出了我们系统中的 shell:

```
[msf@linux]$ cat /etc/shells
/bin/sh
/bin/csh
```

在 Linux 系统中,常见的 shell 有两种:B shell 和 C shell。B shell(sh)是最初的 Unix shell,它是由 Steve Bourne 编写的,这个 shell 的可执行文件为/bin/sh;C shell(csh)是由 Bill Joy 开发的,C shell 的可执行文件为/bin/csh。

默认的 Linux shell 为 B shell,要知道你现在所用的是哪种 shell,可以使用命令:echo \$SHELL,它的运行结果如下面所示:

```
[msf@linux]$ echo $SHELL
/bin/sh
```

其实,我们可以利用一种比较简单的方法确定所使用的 shell。如果你的提示符为美元符号(\$),则你使用的为 B shell;如果你的提示符为百分号(%)或#,则你使用的为 C shell,其中%为一般用户的提示符,#为超级用户的提示符。当然,这种方法不总是正确的,因为用户可以改变自己的提示符。

如果我们要改变我们的 shell,则可以使用命令 chsh,下面给出了一个更改 shell 的例子:

```
[msf@linux]$ echo $SHELL
/bin/sh
```



```
[msf@linux]$ chsh
changing shell for msf.
Password:
New shell [/bin/sh]:/bin/csh
shell changed.
[msf@linux]% echo $SHELL
/bin/csh
```

可以看出,当 shell 由 B shell 改为 C shell 时,提示符也由美元符号(\$)变为百分号(%)

1.1.4 了解 shell 环境

登录过程的一部分为建立用户环境,所有 Linux 进程各自都有独立的并且不同于程序本身的环境。Linux 环境,也叫 shell 环境,是由很多变量以及这些变量的值所组成。这些变量和变量值允许一个正在运行的程序(如 shell)来决定这个环境看上去是怎样的。

环境包括你所使用的 shell,你的起始目录以及你所使用的终端类型等一系列信息。这些变量中有一些变量在登录过程中被定义,并且要么不能改变,要么不应该改变;而另外一些变量是能够被改变的。

在环境中,以“VARIABLE = value”的形式来设置变量。其中 VARIABLE 的含义可以被设置为你喜欢的任何内容。但是,对许多标准 Linux 程序而言,许多变量有预先定义的含义。例如,TERM 变量被定义为你的终端名字,DEC 公司花了几年的时间制作了一个名为 vt-100 的流行终端。这种终端的特性已被许多厂家复制,并且经常用于被个人计算机的软件仿真。这种终端的名字为 vt-100,在环境中被表示为 TERM = vt100。

环境中许多预定义了许多变量,表 1.1 列出了 B shell 的通用环境变量。

表 1.1 B shell 的通用环境变量

变 量	描 述
HOME = /home/login	HOME 用于设置你的起始目录,起始目录为你开始工作的位置。如果你的 ID 为 msf,则 HOME 被定义为/home/msf
LOGNAME = login	LOGNAME 和你的登录 ID 一样是被自动设置的
PATH = path	Path 为 shell 查找命令的目录列表。例如,你可以设置路径如下:PATH = /bin:/usr/bin
PS1 = prompt	PS1 为 shell 提示符。你可以设置 PS1 = “Enter Command”,那么 Enter Command > 将会成为你的命令提示符
PWD = directory	PWD 是被自动设置的。它定义你现在的在文件系统中的位置
SHELL = shell	SHELL 定义你的 shell 程序的位置
TERM = termtype	你登录终端的名字

除了表中列出的通用环境变量以外,系统还有很多其他的环境变量。如果你使用的是 C shell,那么可用 printenv 命令列出这些变量;如果你使用 B shell,则可以使用 env 命令。下面给出了在 B shell 中,env 命令的运行情况:

```
[msf@linux]$ env
```



```
PWD = /home/msf
HOSTNAME = linux.tsinghua.edu.cn
PS1 = [ \u@ \h] \ $
USER = msf
MAIL = /usr/spool/mail/msf
LOGNAME = msf
SHELL = /bin/sh
TERM = vt100
HOME = /home/msf
PATH = ./:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
```

在上面所显示的环境变量中,只有 PS1 比较难懂,现在我们就介绍一下。PS1 的表达式中, \u 代表用户名, \h 代表登录的主机名, \ \$ 为系统默认提示符,这里就是美元符号 (\$)。例如,如果以用户名 msf 登录主机 linux,则系统的提示符为[msf@linux] \$。

1.1.5 配置 shell 环境

在 DOS 启动时,DOS 系统执行 autoexec.bat 批处理文件来设置 PATH、PROMPT 等属性;在 Linux 系统中,根据所启动的 shell 的不同,Linux 执行不同的文件来设置相应的属性。由于 Linux 是多用户的系统,每个用户都可以建立与其他用户不同的运行环境,因此这些文件放在每一用户自己的注册目录中,在 B shell 进程启动时,系统执行用户注册目录中的 .profile 文件,以设置 shell 变量的值;如果用户注册启动 C shell 进程,系统会执行用户注册目录中的 .cshrc 和 .login 文件,用户可将期望的有关设置写在这两个文件中。

由于 Linux 的默认 shell 为 B shell,因此,我们在下面给出一个 B shell 的配置文件。

```
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
alias ..='cd ..'
alias ll='ls -l'
alias la='ls -aF'
PATH=$PATH:./
export PATH
```

对于默认的 PATH 环境变量,一般不包含当前目录。这对于我们的编程工作非常不方便。因此,我们常常得在 .profile 中添加当前目录(./)到 PATH 环境变量中去。在上面的例子中,我们还利用 alias 指令为一些 shell 命令起了别名,如将 rm 设置为 rm -i 的别名可以在用户删除文件时要求确认,这样可以防止用户的误删除操作。

1.1.6 注销

在用户完成所做的工作后,要离开 Linux 系统时需要进行注销。注销工作是必要的,因

