

Linux 系统编程教程

编 制：信盈达研发部

版 本：V1.1

编制日期：2012 年 10 月 22 日

修改日期：2013 年 06 月 12 日

版权声明：该培训教程版权归深圳信盈达科技有限公司所有，未经公司授权禁止引用、发布、转载等，否则将追究其法律责任。

目录

第 1 章	Linux 基础知识	6
1.1	Linux 历史	6
1.2	Linux 目录结构	6
1.3	Linux 常用命令	7
1.3.1	su 用户切换	7
1.3.2	useradd 添加用户	7
1.3.3	passwd 修改密码	7
1.3.4	进程相关（系统管理）命令	8
1.3.5	磁盘相关命令（fdisk）	8
1.3.6	pwd 显示工作目录	8
1.3.7	cd 改变工作目录	9
1.3.8	ls 查看文件/目录	9
1.3.9	mkdir 创建目录	10
1.3.10	cp 文件/目录拷贝	10
1.3.11	mv 文件/目录改名、移动	11
1.3.12	rm 文件/目录删除	11
1.3.13	cat 查看文件内容	11
1.3.14	more 分页显示文件内容	12
1.3.15	less 分页显示文件内容	12
1.3.16	管道与命令替换	13
1.3.17	umask 文件/目录创建掩码	13
1.3.18	chgrp 改变文件/目录所属的组	13
1.3.19	chown 更改某个文件/目录的所有者和属组	14
1.3.20	chmod 修改文件权限	14
1.3.21	find 文件查找	15
1.3.22	locate 快速文件查找	17
1.3.23	ln 建立符号链接	17
1.3.24	whereis 和 which 查找命令所在目录	17
1.3.25	grep 搜索文件内容	18
1.3.26	tar 压缩解压	19
1.3.27	gzip/gunzip 和 bzip2/bunzip2 文件压缩/解压缩	20
1.3.28	unzip winzip 文件解压缩	21
1.3.29	ifconfig 及网络相关命令	21
1.3.30	linux 下编程环境	22
1.4	vi 编辑器	22
1.5	Linux 常用技巧	24
1.5.1	使用“Tab”键	24
1.5.2	Ret Hat Linux 启动到字符界面(不启动 XWindow)	24
1.5.3	挂接 USB 闪存	24
1.5.4	不用 samba 或 ftp，利用 VM 自身的实现本机与虚拟机的文件共享	25
1.5.5	其他常用命令	26
1.5.6	补充：vi 换行自动缩进对齐以及 vim 中设置多窗口	27
1.5.7	Shell 编程	29
第 2 章	LINUX 下编译与调试	38
2.1	gcc/g++编译器	38
2.1.1	gcc/g++编译过程	38
2.1.2	静态库和动态库	40
2.1.3	gcc 警告和优化选项	43

2.2	make 工程管理器 and Makefile.....	45
2.2.1	Makefile 中特殊处理与伪目标	46
2.2.2	变量、规则与函数.....	47
2.3	gdb 调试器.....	50
2.3.1	gdb 常用命令.....	50
2.3.2	gdb 应用举例.....	52
第 3 章	Linux 文件目录操作	54
3.1	基于文件指针的文件操作(缓冲).....	54
3.1.1	文件的创建, 打开与关闭.....	54
3.1.2	读写文件.....	55
3.1.3	文件定位.....	57
3.1.4	标准输入/输出流.....	57
3.1.5	目录操作.....	58
3.2	基于文件描述符的文件操作(非缓冲).....	61
3.2.1	文件描述符.....	61
3.2.2	打开、创建和关闭文件.....	61
3.2.3	读写文件.....	62
3.2.4	改变文件大小.....	63
3.2.5	文件定位.....	64
3.2.6	原子操作.....	64
3.2.7	进一步理解文件描述符.....	64
3.2.8	文件描述符的复制.....	65
3.2.9	文件的锁定.....	67
3.2.10	获取文件信息.....	71
3.2.11	access 函数	73
3.2.12	标准输入输出文件描述符.....	73
3.2.13	时间和日期相关函数.....	74
3.2.14	处理的模型(补充).....	75
3.2.15	串口编程(选修).....	78
第 4 章	Linux 多进程	86
4.1	Linux 进程概述.....	86
4.1.1	进程标识.....	86
4.1.2	进程的用户 ID 与组 ID(进程的运行身份).....	86
4.1.3	进程的状态.....	88
4.1.4	Linux 下的进程结构及管理	88
4.2	Linux 进程的创建与控制	89
4.2.1	fork 函数	89
4.2.2	进程的终止.....	91
4.2.3	wait 和 waitpid 函数.....	93
4.2.4	exec 函数族	95
4.2.5	system 函数.....	97
4.2.6	popen 函数.....	98
4.3	守护进程.....	99
第 5 章	Linux 信号处理	102
5.1	信号概念.....	102
5.2	signal 信号处理机制	103
5.3	sigaction 信号处理机制	105
5.3.1	信号处理情况分析.....	105
5.3.2	sigaction 信号处理注册	106

5.3.3	sigprocmask 信号阻塞.....	109
5.4	用程序发送信号.....	111
5.4.1	kill 和 raise 信号发送函数.....	111
5.4.2	sigqueue 信号发送函数.....	112
5.5	计时器与信号.....	114
5.5.1	睡眠函数.....	114
5.5.2	时钟处理.....	115
第 6 章	进程间通信.....	117
6.1	(Interprocess Communication,IPC)简介.....	117
6.2	标准流管道.....	118
6.3	无名管道(PIPE).....	118
6.4	命名管道(FIFO).....	120
6.4.1	创建、删除 FIFO 文件.....	120
6.4.2	打开、关闭 FIFO 文件.....	121
6.4.3	读写 FIFO.....	122
6.5	内存映射.....	122
6.6	System V 共享内存机制: shmget shmat shmdt shmctl.....	126
6.7	消息队列.....	129
6.8	信号量.....	132
6.8.1	System V IPC 机制: 信号量。.....	133
6.8.2	选修: Posix 有名信号量.....	137
第 7 章	Linux 多线程.....	141
7.1	Linux 多线程概述.....	141
7.1.1	多线程概述.....	141
7.1.2	线程分类.....	141
7.1.3	线程创建的 Linux 实现.....	141
7.2	线程的创建和退出.....	142
7.3	线程的等待退出.....	143
7.3.1	等待线程退出.....	143
7.3.2	线程的取消.....	145
7.3.3	线程终止清理函数.....	146
7.4	线程的互斥.....	148
7.5	线程的同步.....	153
7.5.1	条件变量.....	153
7.5.2	信号灯.....	158
7.6	生产者消费者问题(选修).....	160
7.7	线程的属性(选修).....	163
第 8 章	LINUX 网络编程.....	167
8.1	LINUX 网络编程介绍.....	167
8.1.1	TCP/IP 协议概述.....	167
8.1.2	OSI 参考模型及 TCP/IP 参考模型.....	167
8.1.3	TCP 协议.....	169
8.1.4	UDP 协议.....	170
8.1.5	协议的选择.....	171
8.2	网络相关概念.....	171
8.2.1	socket 概念.....	172
8.2.2	socket 类型.....	172
8.2.3	socket 信息数据结构.....	173
8.2.4	数据存储优先顺序的转换.....	173

8.2.5	地址格式转化.....	174
8.2.6	名字地址转化.....	175
8.3	socket 编程	177
8.3.1	使用 TCP 协议的流程图	177
8.3.2	使用 UDP 协议的流程图.....	191
8.3.3	设置套接口的选项 setsockopt 的用法	196
第 9 章	系统编程的一个小程序.....	197
附录 A:	C 语言文件操作详解	208
附录 B:	内存映射和普通文件访问的区别	215
附录 C:	ping 命令解析	221
附录 D:	大端小端.....	230

第1章 Linux 基础知识

1.1 Linux 历史

操作系统始于 20 世纪 50 年代。1991 年，一个计算机爱好者在 MINIX（一个功能很有限的类似 UNIX 的操作系统）的基础上开发出一款操作系统，叫 LINUX。最初的 LINUX 作用并不大，但是很多人开始对 LINUX 感兴趣，共同开发它，使 LINUX 变成了一个很有用的操作系统，但此时它的功能仍然不够好。

GNU 计划，又称革奴计划，它的目标是创建一套完全自由的操作系统。可以让任何人自由地使用、复制、修改和发布 GNU 软件。GNU 软件都遵守 GPL 协议（GNU General Public License）。或称“反版权”协议。在该计划的促进下，开发出了具有极大改善进步的新的 Linux 操作系统，Linux 开始流行起来。GNU 对 Linux 的贡献非常大。

LINUX 本身只是操作系统的内核，后来才添加进许多应用功能。内核非常重要，是其它程序能够运行的基础。用户或者系统管理员运行的所有程序实际上都运行在内核之上。

为什么要用 Linux 操作系统而不用 Windows？Linux 操作系统相比 Windows 具有的优点：开源，完全免费，这是其它的操作系统做不到的。支持多种平台，Linux 可运行在多种硬件的处理器平台上。最突出的是它可以很好地运行在嵌入式产品上，如掌上电脑、机顶盒、游戏机等。Linux 系统很安全，针对 Linux 的病毒很少。

Linux 也有缺点：可扩展性不强。硬件集成/支持的缺乏。缺乏洞察力等。

名词解释：GPL：GNU 通用开源许可证（GNU General Public License）或称通用开源协议。

1.2 Linux 目录结构

/bin 该目录中存放 Linux 的常用命令，在有的版本中是一些和根目录下相同的目录。

/boot 该目录下存放的都是系统启动时要用到的程序，当用 lilo 引导 Linux 时，会用到这里的一些信息。

/dev 该目录包含了 Linux 系统中使用的所有外部设备，它实际上是访问这些外部设备的端口，你可以访问这些外部设备，与访问一个文件或一个目录没有区别。例如在系统中键入“cd /dev/cdrom”，就可以看到光驱中的文件；键入“cd /dev/mouse”即可看鼠标的相关文件。

/etc 该目录存放了系统管理时要用到的各种配置文件和子目录，例如网络配置文件、文件系统、X 系统配置文件、设备配置信息、设置用户信息等。

/sbin 该目录用来存放系统管理员的系统管理程序。

/home 如果建立一个名为“xx”的用户，那么在/home 目录下就有一个对应的“/home/xx”路径，用来存放该用户的主目录。

/lib 该目录用来存放系统动态连接共享库，几乎所有的应用程序都会用到该目录下的共享库。

/lost+found 该目录在大多数情况下都是空的。但当突然停电、或者非正常关机后，有些文件就临时存放在这里。

/mnt 该目录在一般情况下也是空的，你可以临时将别的文件系统挂在该目录下。

/proc 可以在该目录下获取系统信息，这些信息是在内存中由系统自己产生的

/root 如果你是以超级用户的身份登录的，这个就是超级用户的主目录

/tmp 用来存放不同程序执行时产生的临时文件。

/usr 用户的很多应用程序和文件都存放在该目录下。

/usr/X11R6: X-Window 目录；

/usr/src: Linux 源代码；

/usr/include: 系统头文件；

/usr/lib: 存放常用动态链接共享库、静态档案库；

1.3 Linux 常用命令

1.3.1 su 用户切换

linux 下的两种帐号:

1. root--超级用户帐号（系统管理员），使用这个帐号可以在系统中做任何事情。
2. 普通用户--这个帐号供普通用户使用，可以进行有限的操作。

用户正确地输入用户名和口令后，就能合法地进入系统。屏幕显示：[root@localhost /root]#

这时可以对系统做各种操作。注意超级用户的提示符是“#”，其他用户的提示符是“\$”。利用 `whoami` 可以知道当前登录的用户账号是什么身份。

用 `exit` 或 `logout` 结束当前的 Linux 会话。也可按下<CTRL>+d 来结束此 Linux 会话

功能：切换用户。

语法：su [用户名] （[]表示可选）

说明：su 命令是最基本的命令之一，常用于不同用户间切换。例如，如果登录为 `user1`，要切换为 `user2`，只要用如下命令：`$su user2`。然后系统提示输入 `user2` 口令，输入正确的口令之后就可以切换到 `user2`。完成之后就可以用 `exit` 命令返回到 `user1`。

su 命令的常见用法是变成根用户或超级用户。如果普通用户发出不带用户名的 su 命令，则系统提示输入根口令，输入之后则可切换为根用户。

如果登录为根用户，则可以用 su 命令切换为系统上任何用户而不需要口令。

1.3.2 useradd 添加用户

功能：添加用户。

语法：useradd 用户名

说明：useradd 可用来建立用户帐号。帐号建好之后，再用 `passwd` 设定帐号的密码。而可用 `userdel` 删除帐号。使用 `useradd` 指令所建立的帐号，实际上是保存在 `/etc/passwd` 文本文件中。建立一个用户则在 `/home` 目录下建立一个主目录。

该命令必须有系统管理员权限才能执行

常用选项：`-d` 指定目录，`-m` 指定权限，`-g` 组号（把用户加入到具体的组）

利用 `id` 可以查看用户信息组号，`id lry` 可以查看 `lry` 用户的组信息。

1.3.3 passwd 修改密码

功能：添加密码和修改密码。

语法：passwd [用户名]

说明：出于系统安全考虑，Linux 系统中的每一个用户除了有其用户名外，还有其对应的用户密码。因此使用 `useradd` 命令增加时，还需使用 `passwd` 命令为每一位新增加的用户设置密码；用户以后还可以随时用 `passwd` 命令改变自己的密码。

该命令的一般格式为：`passwd [用户名]` 其中用户名为需要修改密码的用户名。只有超级用户可以使用“passwd 用户名”修改其他用户的密码，普通用户只能用不带参数的 `passwd` 命令修改自己的密码。

利用 `passwd` 命令修改自身的密码的使用方法如下：

输入：`passwd<Enter>`;

在（current） LINUX passwd:下输入当前的密码

在 new password:提示下输入新的密码（在屏幕上看不到这个密码）：

系统提示再次输入这个新密码。

输入正确后，这个新密码被加密并放入 `/etc/shadow` 文件。选取一个不易被破译的密码是很重要的。选取密码应遵守如下规则：

密码应该至少有六位（最好是八位）字符；

密码应该是大小写字母、标点符号和数字混杂的；
超级用户修改其他用户（xxq）的密码的过程如下，
passwd xxq
New LINUX password:
Retype new LINUX password:
passwd: all authentication tokens updated successfully

1.3.4 进程相关（系统管理）命令

ps 命令：类似任务管理器，ps 为我们提供了进程的一次性的查看，它所提供的查看结果并不动态连续的；如果想对进程实时监控，应该用 top 工具；常用形式：ps -aux 和 ps -ef

执行 ps -aux 之后：

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.1	0.1	1372	472	?	S	21:32	0:04	init
root	2	0.0	0.0	0	0	?	SW	21:32	0:00	[keventd]

...

USER 进程的属主；

PID 进程的 ID；（是唯一的数值，用来区分进程）

PPID 父进程；

%CPU 进程占用的 CPU 百分比；

%MEM 占用内存的百分比；

NI 进程的 NICE 值，数值大，表示较少占用 CPU 时间；

VSZ 进程虚拟大小；

RSS 驻留中页的数量；

TTY 终端 ID

WCHAN 正在等待的进程资源；

stat 进程状态；（运行 R、休眠 S、僵尸 Z、停止或被追踪 T、死掉的进程 X、优先级较低的进程 N、优先级高的进程<、进入内存交换 W、非中断休眠（常规 IO）D）

START 启动进程的时间；

TIME 进程消耗 CPU 的时间；

COMMAND 命令的名称和参数；

kill 命令：通常与 ps 命令一起使用，常用的形式：kill -9 进程 ID（表示向指定的进程 ID 发送 SIGKILL 的信号。其中-9 表示强制终止，可以省略。它是信号代码，可以利用 kill -l 列出所有的信号。）另外，pkill 进程名字(可以直接杀死指定进程名的进程)

top 命令：和 ps 相比，top 是动态监视系统任务的工具，top 输出的结果是连续的，比如#top

jobs 命令：观察后台进程。

1.3.5 磁盘相关命令（fdisk）

fdisk 可以查看硬盘分区情况，并可用于对硬盘分区进行管理，给硬盘分区等。常用功能是查看 U 盘状况。比如 fdisk -l。

1.3.6 pwd 显示工作目录

功能：在 Linux 层次目录结构中，用户可以在被授权的任意目录下利用 mkdir 命令创建新目录，也可以利用 cd 命令从一个目录转换到另一个目录。然而，没有提示符来告知用户目前处于哪一个目录中。要想知道当前所处的目录，可以使用 pwd 命令，该命令显示整个路径名。

语法：pwd

说明：此命令显示出当前工作目录的绝对路径。

举例：pwd

根目录以开头的“/”表示。如果 pwd 后面什么都没有，则显示当前所在位置。如果屏幕信息很多，用 clear 可以清除。

1.3.7 cd 改变工作目录

功能：改变工作目录。工作目录：当前左边的信息显示 的目录

语法：cd [dirname]

说明：该命令将当前工作目录切换至 dirname 所指定的目录。若没有指定 dirname，则回到用户的主目录~。为了改变到指定目录，用户必须拥有对指定目录的执行和读权限。

该命令也可以使用通配符。

例如：假设用户当前目录是：/home/lry，现需要更换到/home/lry/Linux 目录中。

\$ cd Linux 此时，用户可以执行 pwd 命令来显示工作目录。（相对路径的形式）。

\$ pwd /home/lry/Linux （绝对路径的形式）

cd .. 到父目录，即上一级目录，相当于“向上”

cd - 到上一次目录，相当于“后退”

cd / 到根目录

cd ~或者只写 cd 到用户主目录下~

1.3.8 ls 查看文件/目录

功能：ls 是英文单词 list 的简写，其功能为列出目录的内容。这是用户最常用的一个命令之一，因为用户需要不时地查看某个目录的内容。该命令类似于 DOS 下的 dir 命令。

语法：ls [选项] [目录或是文件]

说明：对于每个目录，该命令将列出其中的所有子目录与文件。对于每个文件，ls 将输出其文件名以及所要求的其他信息。默认情况下，输出条目按字母顺序排序。当未给出目录名或是文件名时，就显示当前目录的信息。注意，linux 文件系统不是根据后缀来执行文件的，而是根据此文件有没有执行权限。

常用参数：

- a 显示指定目录下所有子目录与文件，包括隐藏文件。
- A 显示指定目录下所有子目录与文件，包括隐藏文件。但不列出“.”和“..”。
- i 在输出的第一列显示文件的 i 节点号。
- l 以长格式来显示文件的详细信息。这个选项最常用。
- d 列出目录本身
- p 在目录后面加一个“/”。
- R 递归式地显示指定目录的各个子目录中的文件。

以-l 长格式显示文件的详细时，每行列出的信息依次是：

文件类型与权限 链接数 文件所有者 文件属组 文件大小 最近修改的时间 名字

对于符号链接文件，显示的文件名之后有“—>”和引用文件路径名。

对于设备文件，其“文件大小”字段显示主、次设备号，而不是文件大小。

目录中的总块数显示在长格式列表的开头，其中包含间接块。

用 ls -l 命令显示的信息中，开头是由 10 个字符构成的字符串，其中第一个字符表示文件类型，它可以是下述类型之一：

- 普通文件
- d 目录
- l 符号链接
- b 块设备文件
- c 字符设备文件
- p 命名管道（FIFO）

● s socket 文件

后面的 9 个字符表示文件的访问权限，分为 3 组，每组 3 位。

第一组表示文件所有者的权限，第二组表示同组用户的权限，第三组表示其他用户的权限。每一组的三个字符分别表示对文件的读、写和执行权限。

各权限如下所示：

- r 读 (4)
- w 写 (2)
- x 执行 (1) 对于目录，表示进入权限。
- 没有设置权限。

另外：ls 输出内容是有颜色的，比如：目录是蓝色，压缩文件是红色的显示，如果没有颜色，可以加上参数 `--color=never` 表示输出没有彩色，而 `--color=auto` 表示自动，`--color=always` 表示始终有颜色。

通配符在 ls 命令中的应用：

- * 代表 0 个或多个字符
- [] 表示内部包括任何字符
- ? 表示任何单个字符

如果需要更加详细的参数描述，可以通过如下三种方式获得 ls 的帮助：

```
ls --help
man ls
info ls
```

1.3.9 mkdir 创建目录

功能：创建一个目录

语法：mkdir [选项] dirname

说明：该命令创建由 dirname 命名的目录。要求创建目录的用户在当前目录中（dirname 的父目录中）具有写权限，并且 dirname 不能是当前目录中已有的目录或 文件名称。

参数：

- m 对新建目录设置存取权限。也可以用 chmod 命令设置。
- p 可以是一个路径名称。此时若路径中的某些目录尚不存在，加上此选项后，系统将自动建立好那些尚不存在的目录，即一次可以建立多个目录。

1.3.10 cp 文件/目录拷贝

功能：将给出的文件或目录拷贝到另一文件或目录中。

语法：cp [选项] 源文件或目录 目标文件或目录

说明：该命令把指定的源文件复制到目标文件或把多个源文件复制到目标目录中。

参数：

- a 该选项通常在拷贝目录时使用。它保留链接、文件属性，并递归地拷贝目录。
- d 拷贝时保留链接。
- f 删除已经存在的目标文件而不提示。
- i 和 f 选项相反，在覆盖目标文件之前将给出提示要求用户确认。回答 y 时目标文件将被覆盖，是交互式拷贝。
- r 若给出的源文件是一目录文件，此时 cp 将递归复制该目录下所有的子目录和文件。此时目标文件必须为一个目录名。

要说明的是，为防止用户在不经意的情况下用 cp 命令破坏另一个文件，如用户指定的目标文件名是一个已存在的文件名，用 cp 命令拷贝文件后，这个文件就会被新拷贝的源文件覆盖，因此，建议用户在使用 cp 命令拷贝文件时，最好使用 i 选项。

```
$ cp -i a.txt /home/b.txt
```

该命令将文件 a.txt 拷贝到/home 这个目录下，并改名为 b.txt。若不希望重新命名，可以使用下面的命令：

```
$ cp a.txt /home
```

```
$ cp -r /home/lry /root
```

 将/home/lry 目录中的所有文件及其子目录拷贝到目录/root 中。

1.3.11 mv 文件/目录改名、移动

功能：为文件或目录改名或将文件由一个目录移入另一个目录中。该命令如同 DOS 下 ren 和 move 的组合。

语法：mv [选项] 源文件或目录 目标文件或目录

说明：视 mv 命令中第二个参数类型的不同（是目标文件还是目标目录），mv 命令将文件重命名或将其移至一个新的目录中。当第二个参数类型是文件时，mv 命令完成文件重命名，此时，源文件只能有一个（也可以是源目录名），它将所给的源文件或目录重命名为给定的目标文件名。当第二个参数是已存在的目录名称时，源文件或目录参数可以有多个，mv 命令将各参数指定的源文件均移至目标目录中。在跨文件系统移动文件时，mv 先拷贝，再将原有文件删除，而链至该文件的链接也将丢失。

参数：

-i 交互方式操作。如果 mv 操作将导致对已存在的目标文件的覆盖，此时系统询问是否重写，要求用户回答 y 或 n，这样可以避免误覆盖文件。

-f 禁止交互操作。在 mv 操作要覆盖某已有的目标文件时不给任何指示，指定此选项后，i 选项将不再起作用。

如果所给目标文件（不是目录）已存在，此时该文件的内容将被新文件覆盖。为防止用户在不经意的情况下用 mv 命令破坏另一个文件，建议用户在使用 mv 命令移动文件时，最好使用 i 选项。

需要注意的是，mv 与 cp 的结果不同。mv 好象文件“搬家”，文件个数并未增加，而 cp 对文件进行复制，文件个数增加了。

例 1：将/lry 中的所有文件移到当前目录（用“.”表示）中：

```
$ mv /home/lry/* .
```

例 2：将文件 a.txt 重命名为 b.txt

```
$ mv a.txt b.txt
```

1.3.12 rm 文件/目录删除

功能：在 linux 中创建文件很容易，系统中随时会有文件变得过时且毫无用处。用户可以用 rm 命令将其删除。该命令的功能为删除一个目录中的一个或多个文件或目录，它也可以将某个目录及其下的所有文件及子目录均删除。对于链接文件，只是删除了链接，原有文件均保持不变。

语法：rm [选项] 文件...

说明：如果没有使用-r 选项，则 rm 不会删除目录。

参数：

-f 忽略不存在的文件，从不给出提示。

-r 指示 rm 将参数中列出的全部目录和子目录均递归地删除。

-i 进行交互式删除。

使用 rm 命令要格外小心。因为一旦一个文件被删除，它是不能被恢复的。例如，用户在输入 cp，mv 或其他命令时，不小心误输入了 rm 命令，当用户按了回车键并认识到自己的错误时，已经太晚了，文件已经没有了。为了防止此种情况的发生，可以使用 rm 命令中的 i 选项来确认要删除的每个文件。如果用户输入 y，文件将被删除。如果输入任何其他东西，文件将被保留。

1.3.13 cat 查看文件内容

功能：查看文件内容

语法：cat [选项] [文件]

参数：

-b 对非空输出行编号

-E 在每行结束处显示\$

-n 对输出的所有行编号

-s 不输出多行空行

标准的输入输出与重定向:

文件描述符是一个整数,它代表一个打开的文件,标准的三个描述符号:

标准输入: 一般指键盘,描述符为: 0

标准输出: 一般指屏幕输出,描述符为: 1

错误输出: 也是屏幕,描述符为: 2

重定向符号:

<重定向输入 >重定向输出 >>添加输出 2>错误重定向 &>错误和信息重定向

cat 常常与重定向一起使用。其中>表示创建,>>表示追加,<<表示以什么结束

如果 cat 的命令行中没有参数,它就会从标准输入中读取数据,并将其送到标准输出。

linux 中创建空文件的四种方式:

方式 1: [root@localhost ~]# echo > a.txt 活直接> a.txt

方式 2: [root@localhost ~]# touch b.txt

方式 3: [root@localhost ~]# cat > c.txt 然后按 ctrl+c 组合键退出; 或 Ctrl+d

方式 4: [root@localhost ~]# vi d.txt 进入之后: wq 退出。

1.3.14 more 分页显示文件内容

功能: 显示文件的内容, 然后根据窗口大小进行分页显示。

语法: more [参数选项] 文件

说明: 非常大的文本则要用 more 来查看, more 具有分页显示功能

参数如下:

- +num 从第 num 行开始显示;
- num 定义屏幕大小, 为 num 行;
- +pattern 从 pattern 前两行开始显示;
- c 从顶部清屏然后显示;
- d 提示 Press space to continue, 'q' to quit. (按空格键继续, 按 q 键退出), 禁用响铃功能;
- s 把连续的多个空行显示为一行
- u 把文件内容中的下划线去掉

举例:

[root@localhost ~]# more -dc /etc/profile 注: 显示提示, 并从终端或控制台顶部显示;

[root@localhost ~]# more +4 /etc/profile 注: 从 profile 的第 4 行开始显示;

[root@localhost ~]# more -4 /etc/profile 注: 每屏显示 4 行;

[root@localhost ~]# more +/MAIL /etc/profile 注: 从 profile 中的第一个 MAIL 单词的前两行开始显示;

进入 more 之后, 需要的 more 的动作指令如下:

查看一个内容较大的文件时, 要用到 more 的动作指令, f (或空格键) 是向下显示一屏, Enter 键可以向下滚动显示 1 行。=输出当前行的行号; 进入 more 环境后, 退出 more 的动作指令是 q。

一般 more 都和管道结合起来使用。命令通过管道和 more 结合的运用例子

我们列一个目录下的文件, 由于内容太多, 可以用 more 来分页显示。这要和管道 | 结合起来。

管道: 将一个程序或命令的输出作为另一个程序或命令的输入。

比如: [root@localhost ~]# ls -l /etc | more

1.3.15 less 分页显示文件内容

功能: less 工具也是对文件或其它输出进行分页显示的工具, 是 linux 正统查看文件内容的工具, 功能极其强大;

语法: less [参数] 文件

常用参数:

- c 从顶部（从上到下）刷新屏幕，并显示文件内容。而不是通过底部滚动完成刷新；
- f 强制打开文件，二进制文件显示时，不提示警告；
- i 搜索时忽略大小写；除非搜索串中包含大写字母；
- I 搜索时忽略大小写，除非搜索串中包含小写字母；
- m 显示读取文件的百分比；
- M 显示读取文件的百分比、行号及总行数；
- N 在每行前输出行号；
- p pattern 搜索 pattern；比如在/etc/profile 搜索单词 MAIL，就用 less -p MAIL /etc/profile
- s 把连续多个空白行作为一个空白行显示；
- Q 在终端下不响铃；

less 的动作命令：

- 回车键 向下移动一行；
- y 向上移动一行；
- f 或空格键 向下滚动一屏；
- b 向上滚动一屏；
- d 向下滚动半屏；
- h less 的帮助；
- u 向上滚动半屏；
- w 可以指定显示哪行开始显示，是从指定数字的下一行显示；比如指定的是 6，那就从第 7 行显示；
- g 跳到第一行；
- G 跳到最后一行；
- p n% 跳到 n%，比如 20%，也就是说比整个文件内容的 20% 处开始显示；
- /pattern 搜索 pattern，比如 /MAIL 表示在文件中搜索 MAIL 单词；
- q 退出 less

1.3.16 管道与命令替换

管道：是重定向的一种，就像一个导管一样，将一个程序或命令的输出作为另一个程序或命令的输入。eg: #ls -l /etc | wc -w

命令替换：和重定向有点相似，但区别在于命令替换是将一个命令的输出作为另一个命令的参数。常用的格式为：command1 `command2` 或 command1 \$(command2)

举例：

首先列出当前的所有信息，并重定向到 a.txt 文件中：

```
#ls | cat > a.txt 或 ls > a.txt
```

然后，通过命令替换，列出 a.txt 文件中所有的文件信息

```
#ls -l `cat a.txt` 或者用 ls -l $(cat a.txt)
```

1.3.17 umask 文件/目录创建掩码

umask 指文件（0666）或目录（0777）创建时在全部权限中要去掉的一些权限，普通用户缺省时 umask 的值为 002，超级用户为 022

002 表示创建目录时所有者的权限不去掉，所属组权限不去掉，其他组权限写属性去掉

创建一文件以后，普通用户缺省的权限为 664 超级用户： 644

创建一目录以后，普通用户缺省的权限为 775 超级用户： 755

可以通过#umask 查看默认的缺省的掩码值。通过#umask 001 修改掩码值。

1.3.18 chgrp 改变文件/目录所属的组

功能：改变文件或目录所属的组。

语法: `chgrp [选项] group filename`

说明: 该命令改变指定文件所属的用户组。其中 `group` 可以是用户组 ID（可通过 `id lry` 查看），也可以是 `/etc/group` 文件中用户组的组名。文件名是以空格分开的要改变属组的文件列表，支持通配符。如果用户不是该文件的所有者或超级用户，则不能改变该文件的组。

参数:

- R 递归式地改变指定目录及其下的所有子目录和文件的属组。

例 1: `$ chgrp -R lry /opt/local/book`

改变 `/opt/local/book/` 及其子目录下的所有文件的属组为 `lry`。

1.3.19 chown 更改某个文件/目录的所有者和属组

功能: 更改某个文件或目录的所有者和属组。这个命令也很常用。例如 `root` 用户把自己的一个文件拷贝给用户 `lry`，为了让用户 `lry` 能够存取这个文件，`root` 用户应该把这个文件的所有者设为 `lry`，否则，用户 `lry` 无法存取这个文件。

语法: `chown [选项] 用户或组 文件`

说明: `chown` 将指定文件的拥有者改为指定的用户或组。用户可以是用户名或用户 ID。组可以是组名或组 ID。文件是以空格分开的要改变权限的文件列表，支持通配符。

参数:

- R 递归式地改变指定目录及其下的所有子目录和文件的拥有者。

- v 显示 `chown` 命令所做的工作。

例 1: 把文件 `a.txt` 的所有者改为 `lry`。

`$ chown lry a.txt`

例 2: 把目录 `/his` 及其下的所有文件和子目录的所有者改成 `lry`，属组改成 `users`。

`$ chown -R lry.users /his` 或 `$ chown -R lry:users /his`

1.3.20 chmod 修改文件权限

`chmod 数字 文件(夹)`: 修改文件(夹)的权限。最常用最简便的用法是 `chmod 777 文件(夹)`。此命令经常用到。如果出现权限不允许，可执行文件不能执行，文件夹不能访问等情况，则需要执行 `chmod 777 文件(夹)` 来解决。

功能: `chmod` 命令是非常重要的，用于改变文件或目录的访问权限。用户用它控制文件或目录的访问权限。

语法: 该命令有两种用法。一种是包含字母和操作符表达式的文字设定法；另一种是包含数字的数字设定法。

说明: 我们利用 `ls -l` 长格式列出文件或目录的基本信息如下：

文件类型与权限 链接数 文件所有者 文件属组 文件大小 最近修改的时间 名字

对于权限，有第一组表示文件所有者的权限，第二组表示同组用户的权限，第三组表示其他用户的权限。每一组的三个字符分别表示对文件的读、写和执行权限。可以通过 `chmod` 来修改权限。

1. 文字设定法

`chmod [who][+|=][mode] 文件名` //修改指定文件名中 `who` 的权限增加/去除/赋值为 `mode`

参数:

操作对象 `who` 可是下述字母中的任一个或者它们的组合：

`u` 表示“用户（user）”，即文件或目录的所有者。

`g` 表示“同组（group）用户”，即与文件所有者有相同组 ID 的所有用户。

`o` 表示“其他（others）用户”。

`a` 表示“所有（all）用户”。它是系统默认值。即 `chmod +x 1.c` 表示所有人都有可执行的权限。

操作符号可以是：

- + 添加某个权限。

- 取消某个权限。

- = 赋予给定权限并取消其他所有权限（如果有的话）。

设置 `mode` 所表示的权限可用下述字母的任意组合：

r 可读
w 可写
x 可执行

当是组合的时候, 前面的 who 要指明。

文件名: 以空格分开的要改变权限的文件列表, 支持通配符。

在一个命令行中可给出多个权限方式, 其间用逗号隔开。例如: `chmod g+r, o+r` 示例: 使同组和其他用户对文件示例: 有读权限。

2. 数字设定法

我们必须首先了解用数字表示的属性的含义: 0 表示没有权限, 1 表示可执行权限, 2 表示可写权限, 4 表示可读权限, 然后将其相加。所以数字属性的格式应为 3 个从 0 到 7 的八进制数, 其顺序是 (u) (g) (o)。

例如, 如果想让某个文件的所有者有"读/写"二种权限, 需要把 4 (可读) + 2 (可写) = 6 (读/写)。

数字设定法的一般形式为: `chmod [mode] 文件名`

例子:

(1) 文字设定法:

例 1: `$ chmod a+x sort`

即设定文件 `sort` 的属性为:

文件所有者 (u) 增加执行权限
与文件所有者同组用户 (g) 增加执行权限
其他用户 (o) 增加执行权限

例 2: `$ chmod ug+w, o-x text`

即设定文件 `text` 的属性为:

文件所有者 (u) 增加写权限
与文件所有者同组用户 (g) 增加写权限
其他用户 (o) 删除执行权限

例 3: `$ chmod a - x a.txt`

`$ chmod - x a.txt`

`$ chmod ugo - x a.txt`

以上这三个命令都是将文件 `a.txt` 的执行权限删除, 它设定的对象为所有使用者。

(2) 数字设定法:

例 1: `$ chmod 644 a.txt`

`$ ls -l`

即设定文件 `a.txt` 的属性为:

`-rwxr-x---` 1lry users 30128 Dec 15 10:52 a.txt
文件所有者 (u) lry 拥有读、写权限
与文件所有者同组人用户 (g) 拥有读权限
其他人 (o) 拥有读权限

例 2: `$ chmod 750 a.txt`

`$ ls -l`

`-rwxr-x---` 1lry users 30128 Dec 15 10:52 a.txt

即设定 `a.txt` 这个文件的属性为:

文件主本人 (u) lry 可读/可写/可执行权
与文件主同组人 (g) 可读/可执行权
其他人 (o) 没有任何权限

1.3.21 find 文件查找

功能: 在目录结构中搜索文件, 并执行指定的操作。

语法: `find 起始目录 寻找条件 操作`

说明: `find` 命令从指定的起始目录开始, 递归地搜索其各个子目录, 查找满足寻找条件的文件并对之采取相

关的操作。

该命令提供的寻找条件可以是一个用逻辑运算符 not、and、or 组成的复合条件。逻辑运算符 and、or、not 的含义为：

(1) and: 逻辑与，在命令中用“-a”表示，是系统缺省的选项，表示只有当所给的条件都满足时，寻找条件才算满足。例如：

```
$ find /Linux -name a.txt -type d -user root
```

该命令寻找三个给定条件都满足的所有文件。

(2) or: 逻辑或，在命令中用“-o”表示。该运算符表示只要所给的条件中有一个满足时，寻找条件就算满足。

例如：

```
$ find /Linux -name a.txt -o -name b.txt
```

该命令在路径/Linux 下查询文件名为 a.txt 或是匹配 b.txt 的所有文件。可以使用通配符，此时需要用“”括起来。

(3) not: 逻辑非，在命令中用“!”表示。该运算符表示查找不满足所给条件的文件。

例如：

```
$ find /Linux ! -name a.txt
```

该命令在路径/Linux 下查询文件名不是 a.txt 的所有文件。

1. 以名称和文件属性查找。

- name '字串' 查找文件名匹配所给字串的所有文件，字串内可用通配符*、?、[]。
- gid n 查找属于 ID 号为 n 的用户组的所有文件。
- uid n 查找属于 ID 号为 n 的用户的所有文件。
- group '字串' 查找属于用户组名为所给字串的所有的文件。
- user '字串' 查找属于用户名为所给字串的所有的文件。
- empty 查找大小为 0 的目录或文件。
- perm 权限 查找具有指定权限的文件和目录，权限的表示可以如 711, 644。
- size n[bckw] 查找指定文件大小的文件，n 后面的字符表示单位，缺省为 b，代表 512 字节的块。
- type x 查找类型为 x 的文件，x 为下列字符之一：
 - b 块设备文件
 - c 字符设备文件
 - d 目录文件
 - p 命名管道 (FIFO)
 - f 普通文件
 - l 符号链接文件 (symbolic links)
 - s socket 文件

2. 以时间为条件查找

- amin n 查找 n 分钟以前被访问过的所有文件。（+表示 n 分钟之前，-或者不写表示 n 分钟之内）
- cmin n 查找 n 分钟以前文件状态被修改过的所有文件。
- mmin n 查找 n 分钟以前文件内容被修改过的所有文件。
- atime n 查找 n 天以前被访问过的所有文件。
- ctime n 查找 n 天以前文件状态被修改过的所有文件。
- mtime n 查找 n 天以前文件内容被修改过的所有文件。

3. 可执行的操作

-exec 命令名称 {} :对符合条件的文件执行所给的 Linux 命令，而不询问用户是否需要执行该命令。
{ }表示命令的参数即为所找到的文件；命令的末尾必须以“\;”结束。

-ok 命令名称 { } :对符合条件的文件执行所给的 Linux 命令，与 exec 不同的是，它会询问用户是否需要执行该命令。

- ls 详细列出所找到的所有文件。
- fprintf 文件名 将找到的文件名写入指定文件。
- print 在标准输出设备上显示查找出的文件名。

-printf 格式 格式的写法请参考有关 C 语言的书。

例 1: 查找当前目录中所有以 a 开头的文件, 并显示这些文件的内容。

```
$ find . -name 'a*' -exec more {} \;
```

例 2: 删除当前目录下所有一周之内没有被访问过的 a.txt 或 *.c 文件。

```
$ find . \( -name a.txt -o -name *.c \)
```

```
> -atime +7 -exec rm {} \;
```

说明如下: 命令中的“.”表示当前目录, 此时 find 将从当前目录开始, 逐个在其子目录中查找满足后面指定条件的文件。“- name a.txt”是指要查找名为 a.txt 的文件; “-name *.c”是指要查找所有名字以 .c 结尾的文件。这两个 -name 之间的 -o 表示逻辑或 (or), 即查找名字为 a.txt 或名字以 c 结尾的文件, 并且需要用 () 括起来, 对于 () 而言, 需要采用 \ (和 \) 的形式并且要跟后面的 -name 有空格隔开。find 在当前目录及其子目录下找到这样的文件之后, 再进行判断, 看其最后访问时间是否在 7 天以前 (条件 -atime +7), 若是, 则对该文件执行命令 rm (-exec rm -rf { } \;)。其中 { } 代表当前查到的符合条件的文件名, \; 则是语法所要求的。上述命令中第一行的最后一个 \ 是续行符。当命令太长而在一行写不下时, 可输入一个 \, 之后系统将显示一个 >, 指示用户继续输入命令。如果一行写完, 则是: find ./ \(-name a.txt -o -name *.c \) -atime +7 -exec rm -rf { } \;

另外, 对于 basename 和 dirname 这两个命令:

```
basename /home/lry      输出: lry
```

```
dirname /home/lry       输出: /home
```

1.3.22 locate 快速文件查找

功能: locate 命令用于查找文件, 它比 find 命令的搜索速度快, 它需要一个数据库, 这个数据库由每天的例行工作 (crontab) 程序来建立。当我们建立好这个数据库后, 就可以方便地来搜寻所需文件了。locate 的应用, 首先要通过 updatedb 建立索引数据库, 然后才能应用; 如果您新安装了软件或者存放了新的文件, 也要先运行 updatedb 命令, 以生成最新索引库。

语法: locate 相关字

例如: 查找相关字 issue

```
[root@localhost ~]#locate issue
```

```
/etc/issue
```

```
/etc/issue.net
```

```
/usr/man/man5/issue.net.5
```

要记得在搜索之前先运行 updatedb 更新索引数据库, 保证数据准确

```
[root@localhost ~]#updatedb
```

1.3.23 ln 建立符号链接

ln 连接文件或目录, 分为软链接和硬链接。

软连接语法: ln -s 源文件 目标文件 (删除源文件之后, 链接变成无效的了), 相当于快捷方式。

硬链接语法: ln 源文件 目标文件 (删除源文件之后, 目标没有影响)

#ln -s a.txt p 创建软链接, 然后 ll 查看就可以看到 p 带有一个箭头指向 a.txt。

1.3.24 whereis 和 which 查找命令所在目录

whereis:

功能: 查找系统文件的源、二进制文件和手册帮助信息等各部分

语法: whereis 命令

比如, 我们不知道 fdisk 工具放在哪里, 就可以用 whereis fdisk 来查找;

```
[root@localhost ~]# whereis fdisk
```

```
fdisk: /sbin/fdisk /usr/share/man/man8/fdisk.8.gz
```

which:

功能: 在环境变量中设置好的路径中寻找命令（或可执行程序），路径的环境变量时 PATH，可以通过 echo \$PATH 查看系统设定的环境变量。

语法: which 命令

比如，我们不知道 fdisk 工具放在哪里，就可以用 which fdisk 来查找：

```
[root@localhost ~]# which fdisk
/sbin/fdisk
```

1.3.25 grep 搜索文件内容

功能: grep 过滤器查找指定字符模式的文件，并显示含有此模式的所有行。被寻找的模式称为正则表达式。其中，过滤器是一个程序，它接受来自标准输入文件的输入，处理（过滤）它，然后再发送它的输出到标准输出文件。正则表达式本身就是一个小型的编程语言，即 perl 语言，常与 linux 结合使用。

常用的一些正则表达式如表 1.1:

表 1.1

^	以什么开头	Eg:ls -l grep ^d 显示目录
\$	以什么结尾	Eg:ls -l grep t\$ 显示所有以 t 结尾的文件
.	任意单个字符	Eg:ls -l grep ^d...r.x..x 显示所有的，以同组的读、可执行权限和其他人的可执行权限的目录

语法: grep [选项] [查找模式] [文件名 1, 文件名 2,]

参数:

- E 每个模式作为一个扩展的正则表达式对待。
- F 每个模式作为一组固定字符串对待（以新行分隔），而不作为正则表达式。
- c 只显示匹配行的数量。
- i 比较时不区分大小写。
- l 显示首次匹配串所在的文件名并用换行符将其隔开。当在某文件中多次出现匹配串时，不重复显示此文件名。
- n 在输出前加上匹配串所在行的行号（文件首行行号为 1）。
- v 只显示不包含匹配串的行。
- x 只显示整行严格匹配的行。

对该组命令的使用还需注意以下方面：

在命令后键入搜索的模式，再键入要搜索的文件。其中，文件名列表也可以使用特殊字符，如“*”等，用来生成文件名列表。如果想在搜索的模式中包含有空格的字符串，可以用单引号把要搜索的模式括起来，用来表明搜索的模式是由包含空格的字符串组成。在下面的例子中，grep 命令在文件示例中搜索模式“text file”。

\$ grep 'text file'

特殊字符在搜索一组指定的文件时非常有用。例如，如果想搜索所有的 C 程序源文件中特定的模式，您可以用“*.c”来指定文件名列表。假设用户的 C 程序中包含一些不必要的转向语句（goto 语句），想要找到这些语句，可以用如下的命令来搜索并显示所有包含 goto 语句的代码行：

\$ grep goto *.c

案例：已在 Diaz 电信注册的不同客户的材料存储在 Customers 文件中。该文件的示例数据如下：

"000001","Angela","Smith","16223 Radiance Court","Kansas City","Kansas"

"000002","Barbara","Johnson","227 Beach Ave.,"Alexandria","Virginia"

"000003","Betty","Williams","1 Tread Road","Dublin","Georgia"

写一个寻找住在 Kansas 州的人的个数的命令

grep -c Kansas <customers.txt

写一个命令，显示客户 Linda 的完整的材料及出现该材料的行号

```
grep -n Linda <customers.txt
```

1.3.26 tar 压缩解压

功能：tar 是一个压缩解压工具。利用 tar，用户可以为某一特定文件创建档案（备份文件），也可以在档案中改变文件，或者向档案中加入新的文件。tar 最初被用来在磁带上创建档案，现在，用户可以在任何设备上创建档案，如软盘。利用 tar 命令，可以把一大堆的文件和目录全部打包成一个文件，这对于备份文件或将几个文件组合成为一个文件以便于网络传输是非常有用的。Linux 上的 tar 是 GNU 版本的。语法：tar [主选项+辅选项] 目标文档 源文件或者目录

使用该命令时，主选项是必须要有的，它告诉 tar 要做什么事情，辅选项是辅助使用的，可以选用。

参数：

c 创建新的档案文件。如果用户想备份一个目录或是一些文件，就要选择这个选项。

r 把要存档的文件追加到档案文件的末尾。例如用户已经作好备份文件，又发现还有一个目录或是一些文件忘记备份了，这时可以使用该选项，将忘记的目录或文件追加到备份文件中。

t 列出档案文件的内容，查看已经备份了哪些文件。

u 更新文件。就是说，用新增的文件取代原备份文件，如果在备份文件中找不到要更新的文件，则把它追加到备份文件的最后。

x 从档案文件中释放文件。

注意：c/x/t 仅能存在一个！不可同时存在！

辅助选项：

b 该选项是为磁带机设定的。其后跟一数字，用来说明区块的大小，系统预设值为 20（20*512 bytes）。

f 使用档案文件或设备，这个选项通常是必选的。请留意，在 f 之后要立即接档名喔！不要再加参数！

k 保存已经存在的文件。例如我们把某个文件还原，在还原的过程中，遇到相同的文件，不会进行覆盖。

m 在还原文件时，把所有文件的修改时间设定为现在。

M 创建多卷的档案文件，以便在几个磁盘中存放。

v 详细报告 tar 处理的文件信息。如无此选项，tar 不报告文件信息。

w 每一步都要求确认。

z 用 gzip 来压缩/解压缩文件，后缀名为.gz，加上该选项后可以将档案文件进行压缩，但还原时也一定要使用该选项进行解压缩。

j 用 bzip2 来压缩/解压缩文件，后缀名为.bz2，加上该选项后可以将档案文件进行压缩，但还原时也一定要使用该选项进行解压缩。

例子：

例 1：把/home 目录下包括它的子目录全部做备份文件，备份文件名为 aa.tar。

```
$ tar cvf aa.tar /home
```

例 2：把/home 目录下包括它的子目录全部做备份文件，并进行压缩，备份文件名为 aa.tar.gz。

```
$ tar czvf aa.tar.gz /home //或者 tar jcvf aa.tar.bz2 /home
```

例 3：把 aa.tar.gz 这个备份文件还原并解压缩。

```
$ tar xzvf aa.tar.gz //或者 tar jxvf aa.tar.bz2
```

例 4：查看 aa.tar 备份文件的内容，并以分屏方式显示在显示器上。

```
$ tar tvf aa.tar | more
```

要将文件备份到一个特定的设备，只需把设备名作为备份文件名。

例 5：用户在/dev/fd0 设备的光盘中创建一个备份文件，并将/home 目录中所有的文件都拷贝到备份文件中。

```
$ tar cf /dev/fd0 /home
```

要恢复设备磁盘中的文件，可使用 xf 选项：

```
$ tar xf /dev/fd0
```

如果用户备份的文件大小超过设备可用的存贮空间，如光盘，您可以创建一个多卷的 tar 备份文件。M 选项指示 tar 命令提示您使用一个新的存贮设备，当使用 M 选项向一个光驱进行存档时，tar 命令在一张光盘已满的

时候会提醒您再放入一张新的光盘。这样您就可以把 tar 档案存入几张磁盘中。

```
$ tar cMf /dev/fd0 /home
```

要恢复几张盘中的档案，只要将第一张放入光驱，然后输入有 x 和 M 选项的 tar 命令。在必要时您会被提醒放入另外一张光盘。

```
$ tar xMf /dev/fd0
```

如果要把压缩文件解压到其它位置，需要用 -C 指定具体路径。如：

```
$ tar xzvf aa.tar.gz -C /home 把 aa.tar.gz 解压到/home 目录下。
```

1.3.27 gzip/gunzip 和 bzip2/bunzip2 文件压缩/解压缩

功能：减少文件大小有两个明显的好处，一是可以减少存储空间，二是通过网络传输文件时，可以减少传输的时间。gzip/gunzip 和 bzip2/bunzip2 是在 Linux 系统中经常使用的一个对文件进行压缩和解压缩的命令，既方便又好用。以 gzip 为例：

语法：gzip [选项] 压缩（解压缩）的文件名

参数：

-d 将压缩文件解压。

-l 对每个压缩文件，显示下列字段：

压缩文件的大小

未压缩文件的大小

压缩比

未压缩文件的名字

-r 递归式地查找指定目录并压缩其中的所有文件或者是解压缩。

-t 测试，检查压缩文件是否完整。

-v 对每一个压缩和解压的文件，显示文件名和压缩比。

-num 用指定的数字 num 调整压缩的速度，-1 或 --fast 表示最快压缩方法（低压缩比），-9 或 --best 表示最慢压缩方法（高压缩比）。系统缺省值为 6。

假设一个目录/home 下有文件 mm.txt、sort.txt、xx.com。

例 1：把/home 目录下的每个文件压缩成.gz 文件。

```
$ cd /home
```

```
$ gzip *
```

```
$ ls
```

```
a.txt.gz b.txt.gz c.txt.gz
```

例 2：把例 1 中每个压缩的文件解压，并列出详细的信息。

```
$ gzip -dv * //等价于 gunzip *
```

```
a.txt.gz 43.1%-----replaced with a.txt
```

```
b.txt.gz 43.1%-----replaced with b.txt
```

```
c.txt.gz 43.1%-----replaced with c.txt
```

```
$ ls
```

```
a.txt b.txt c.txt
```

例 3：详细显示例 1 中每个压缩的文件的信息，并不解压。

```
$ ls
```

```
a.txt.gz b.txt.gz c.txt.gz
```

```
$ gzip -l *
```

```
compressed uncompr. ratio uncompressed_name
```

```
277 445 43.1% a.txt
```

```
278 445 43.1% b.txt
```

```
277 445 43.1% c.txt
```

例 4：压缩一个 tar 备份文件，如 aa.tar，此时压缩文件的扩展名为.tar.gz

```
$ gzipaa.tar
```



```
$ ls
```

```
aa.tar.gz
```

几种常见的后缀名的解压和压缩方式总结:

1 .tar 后缀的

解包: `tar xvf FileName.tar`

打包: `tar cvf FileName.tar DirName`

(注: `tar` 是打包, 不是压缩!)

2 .gz 后缀的

解压: `gunzip FileName.gz`

解压: `gzip -d FileName.gz`

压缩: `gzip FileName`

3 .tar.gz 和 .tgz 后缀的

解压: `tar zxvf FileName.tar.gz`

压缩: `tar zcvf FileName.tar.gz DirName`

4 .bz2 后缀的

解压: `bzip2 -d FileName.bz2`

解压: `bunzip2 FileName.bz2`

压缩: `bzip2 -z FileName` // -z 可以去掉

5 .tar.bz2 后缀的

解压: `tar jxvf FileName.tar.bz2`

压缩: `tar jcvf FileName.tar.bz2 DirName`

1.3.28 unzip winzip 文件解压缩

功能: 用 Windows 下的压缩软件 winzip 压缩的文件如何在 Linux 系统下展开呢? 可以用 `unzip` 命令, 该命令用于解扩展名为 .zip 的压缩文件。

语法: `unzip [选项] 压缩文件名.zip`

参数:

- x 文件列表 解压缩文件, 但不包括指定的 file 文件。
- v 查看压缩文件目录, 但不解压。
- t 测试文件有无损坏, 但不解压。
- d 目录 把压缩文件解到指定目录下。
- z 只显示压缩文件的注解。
- n 不覆盖已经存在的文件。
- o 覆盖已存在的文件且不要求用户确认。
- j 不重建文档的目录结构, 把所有文件解压到同一目录下。

例子:

例 1: 将压缩文件 a.zip 在当前目录下解压缩。

```
$ unzip a.zip
```

例 2: 将压缩文件 a.zip 在指定目录/tmp 下解压缩, 如果已有相同的文件存在, 要求 `unzip` 命令不覆盖原先的文件。

```
$ unzip -n a.zip -d /home
```

例 3: 查看压缩文件目录, 但不解压。

```
$ unzip -v a.zip
```

1.3.29 ifconfig 及网络相关命令

网络命令 `ifconfig`: 用于查看和配置网络接口的地址和参数, 包括 IP 地址、子网掩码、广播地址等。详细用

法举例如下：

ifconfig eth0（查看指定的 eth0 的 ip 情况）。

修改 eth0 的 ip，可以通过命令：`#ifconfig eth0 192.168.0.102/24`（或 `ifconfig eth0 192.168.0.164 netmask 255.255.255.0`）。

禁用 `#ifconfig eth0 down`，启动 `#ifconfig eth0 up`。

`#setup` 用管理工具永久设置 ip，设置好之后，此时这个配置信息没有马上保存，需运行 `service network restart` 重新启动网络服务，如果全都看到 ok 的话说明修改成功了。

可以使用 `route -n` 检查一下默认路由看看网关是否为自己的设置。

另外，可以用 `#iptables -F` 命令关闭防火墙。`#netstat -a` 查看网络状态。ping 测试网络通断。比如 `#ping 192.168.0.101`。`#ping 127.0.0.1` 测试网卡好坏

下面讲解如何利用 PUTTY 工具远程连接 linux

配置 ssh 协议 `setup` 命令—System services—启动 SSHD 协议

配置网卡 `ifconfig` 命令：`ifconfig eth0 192.168.0.101 ...`

再用 `ifconfig` 命令查看网卡地址

`service network restart` 重启启动网络服务

`service sshd start` 启动 sshd 服务

`iptables -F` 去除所有防火墙，或 `setup—FireWall configuration—`选择 no firewall

ping 虚拟机地址

ping 通后，运行 `putty.exe` 远程连接

1.3.30 linux 下编程环境

编辑器：Vi（Vim）或 gedit 或 Emacs。编译链接器：gcc。调试器：gdb。项目管理器：make。

1.4 vi 编辑器

vi(visual)是 Linux/UNIX 系列 OS 中通用的全屏编辑器。

vi 分为两种状态，即命令状态和编辑状态，在命令状态下，所键入的字符系统均作命令来处理，如:q 代表退出，而编辑状态则是用来编辑文本资料的。当你进入 vi 时，会先进入命令状态。在命令状态下，按”i”(插入)或”a”(添加)可以进入编辑状态，在编辑状态，按 ESC 键进入命令状态。

插入文本：

- | | |
|---|-------------------------|
| a | 从光标后面开始添加文本 |
| A | 从光标所在行的末尾开始添加文本 |
| i | 从光标前面开始插入文本 |
| I | 从光标所在行的开始处插入文本 |
| o | 在目前光标所在的下一行处插入新的一行 |
| O | 在目前光标所在处的上一行插入新的一行 |
| s | 删除光标所在字符，并进入编辑模式 |
| S | 删除光标所在的行，并进入编辑模式 |
| r | 输入字符，取代光标所在的那一个字符 |
| R | 一直取代光标所在的字符，直到按下 ESC 为止 |

删除与修改：

- | | |
|---------|----------------------|
| x | 删除光标处的字符 |
| dd | 删除光标所在的整行 |
| 3dd | 删除光标所在行以及下面的两行 |
| D 或 d\$ | 删除光标到行尾的文本，常用语删除注释语句 |
| d^或 d0 | 删除光标到行首的文本 |

光标的移动:

可以通过键盘上的四个方向键, 以及 home, end, page up, page down 键移动光标, 在此不必多说。但在远程登录模式下, vi 不支持 PageUp、PageDown、Home、End 等功能键!

- h 或 向左方向键(←) → 光标向左移动一个字符
- j 或 向下方向键(↓) → 光标向下移动一个字符
- k 或 向上方向键(↑) → 光标向上移动一个字符
- l 或 向右方向键(→) → 光标向右移动一个字符
- w 光标往后移一个字
- b 光标往前移一个字
- ^ 光标移动到行首
- \$ 光标移动到行尾
- Ctrl+f 向下翻一页 forward
- Ctrl+b 向上翻一页 back
- Ctrl+d 向下翻半页 down
- Ctrl+u 向上翻半页 up
- gg 光标定位到文档头
- G 光标定位到文档尾
- H 光标定位到当前页首
- L 光标定位到当前页的最后一行的行首
- [n]+ 光标向后移动 n 行,[n]表示一个整数, 比如 10+
- [n]- 光标向前移动 n 行,[n]表示一个整数, 比如 10+
- [n]G 光标定位到第 n 行行首, [n]表示一个整数, 比如 10+

查找与替换:

/[str] 查找字符串 str, [str]表示要查找的字符串, 回车后会加亮显示所有找到的字符串, 接着命令 n 移动到下一个找到的字符串, 命令 N 移动到上一个找到的字符串

示例: /hello

部分替换 (只能替换光标之所在的行)

:s/[src]/[dst] /i 忽略大小写 /g 全部匹配 eg :s/hello/world/ig 替换一行

示例: :3,6 s/[src]/[dst]/ig (3-6 行中找) eg :3,6 s/hello/world

全部替换

示例:

:%s/[src]/[dst]/g 将文档中所有 src 的字符串替换为 dst 字符串

:%s/^ //g 将文档每一行的行首的空格去掉

块操作:

- v 可视化块选择状态, 选中块之后, 可以对块进行删除(d),复制(y),剪切(x)
- yy 复制光标所在的整行
- [n]yy 从光标开始往下复制 n 行,[n]表示一个整数
- p 将复制后的文本粘贴到光标处
- u 撤销上次操作
- ctrl + r 恢复上次操作

结束编辑:

- :q 在未修改文档的情况下退出
- :q! 放弃文档的修改, 强行退出
- :w 保存
- :wq 保存并退出

其他:

:help命令 查看该命令的帮助提示

:%!xxd 十六进制模式

:%!xxd -r 返回文本模式

如果在编辑过程中不小心按了 Ctrl+s, vi 会处于僵死状态, 按 Ctrl+q 可以恢复。

执行 vi +3 main.c //表示定位到 main.c 的第 3 行

执行 vi +/printf main.c //表示定位到第一个 printf 处

在命令模式下输入:new 2.c //表示再打开一个 vi, 是横向的 用 vnew 2.c 表示纵向

也可以通过:split vsplit sp vsp

进行切换用 Ctrl+w 然后再按 w 即可切换

在命令模式中输入 gg=G 可以自动对齐

sed: 管道查找替换程序

cat a.txt | sed 's/aa/bb/' > b.txt //将 a.txt 中的 aa 替换成 bb 并重定向输出到 b.txt 中。

1.5 Linux 常用技巧

1.5.1 使用“Tab”键

大家知道在 Linux 字符界面中输入命令时, 有时需要输入很多字符, 如果经常这样逐个地输入字符, 比较麻烦。假设键入的字符足以确定该目录下一个惟一的文件时, 我们只需按键盘上的“Tab”键就可以自动补齐该文件名的剩下部分, 例如要把目录/root 下的文件“VMwareTools-9.2.0-799703.tar.gz”解包时, 当我们在命令行中键入到“tar xvfz /ccc/V”时, 如果该文件是该目录下惟一以“V”打头的文件的话就可以直接按下“Tab”键, 这时命令会被自动补齐为: tar xvfz /ccc/ VMwareTools-9.2.0-799703.tar.gz, 从而提高了输入效率。按两次“Tab”键, 会显示所有匹配的。

1.5.2 Ret Hat Linux 启动到字符界面(不启动 XWindow)

将/etc/inittab 中 id:5:initdefault: 一行中的 5 改为 3, 注销之后, 重新启动即可进入字符界面。进入字符界面之后, 输入用户名和密码进入系统后, 我们可以输入 startx 指令进入图形用户界面, 按 Ctrl+Alt+BackSpace 切换回字符界面。

1.5.3 挂接 USB 闪存

如果想让 linux 识别 U 盘, 需要把鼠标先定位在虚拟机的 linux 里面, 然后插入 U 盘, 优盘才会被 linux 识别, 再用 fdisk -l 来查看 U 盘的情况。插入 U 盘后, 输入 fdisk -l 显示信息如下:

Disk /dev/sda: 42.9 GB, 42949672960 bytes

255 heads, 63 sectors/track, 5221 cylinders

Units = cylinders of 16065 * 512 = 8225280 bytes

Sector size (logical/physical): 512 bytes / 512 bytes

I/O size (minimum/optimal): 512 bytes / 512 bytes

Disk identifier: 0x0002d380

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1	*	1	64	512000	83	Linux
Partition 1 does not end on cylinder boundary.						
/dev/sda2		64	5222	41430016	8e	Linux LVM

Disk /dev/sdb: 8011 MB, 8011120640 bytes

246 heads, 40 sectors/track, 1590 cylinders

Units = cylinders of 9840 * 512 = 5038080 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x399e399d

Device	Boot	Start	End	Blocks	Id	System
/dev/sdb1	*	38	1591	7641088	c	W95 FAT32 (LBA)

其中 Disk /dev/sdb: 8011 MB, 8011120640 bytes 及后面的就是关于 U 盘的信息，U 盘的设备名为 Device Boot 下显示的/dev/sdb1,所以挂载 U 盘时，设备名要写上这个，表示挂载 U 盘，操作如下：

如果是 fat 格式的 U 盘，挂载命令：`mount -t vfat /dev/sdb1 /mnt/udisk`
如果是 ntfs 格式的 U 盘，挂载命令：`mount -t ntfs-3g /dev/sdb1 /mnt/udisk`
如果没有/mnt/udisk 文件夹，可以创建一个 `mkdir /mnt/udisk` 即可~
卸载 U 盘设备：`umount /dev/sdb1`

但是注意，ntfs 格式要挂载的话，如果系统不支持，需要下载一个 ntfs-3g-2011.4.12-5.el5.i386.rpm 包，安装一下。

1.5.4 不用 samba 或 ftp，利用 VM 自身的实现本机与虚拟机的文件共享

要用 vm 软件实现共享，前提是要已经安装好了 VmwareTools 才能实现共享。首先在 VMware 中设置共享文件夹：（点击 VM—setting—options—shared Folders—在 Folder Sharing 中选择 Always Enabled—Add—输入名字—浏览共享的地址），如图 1.1。

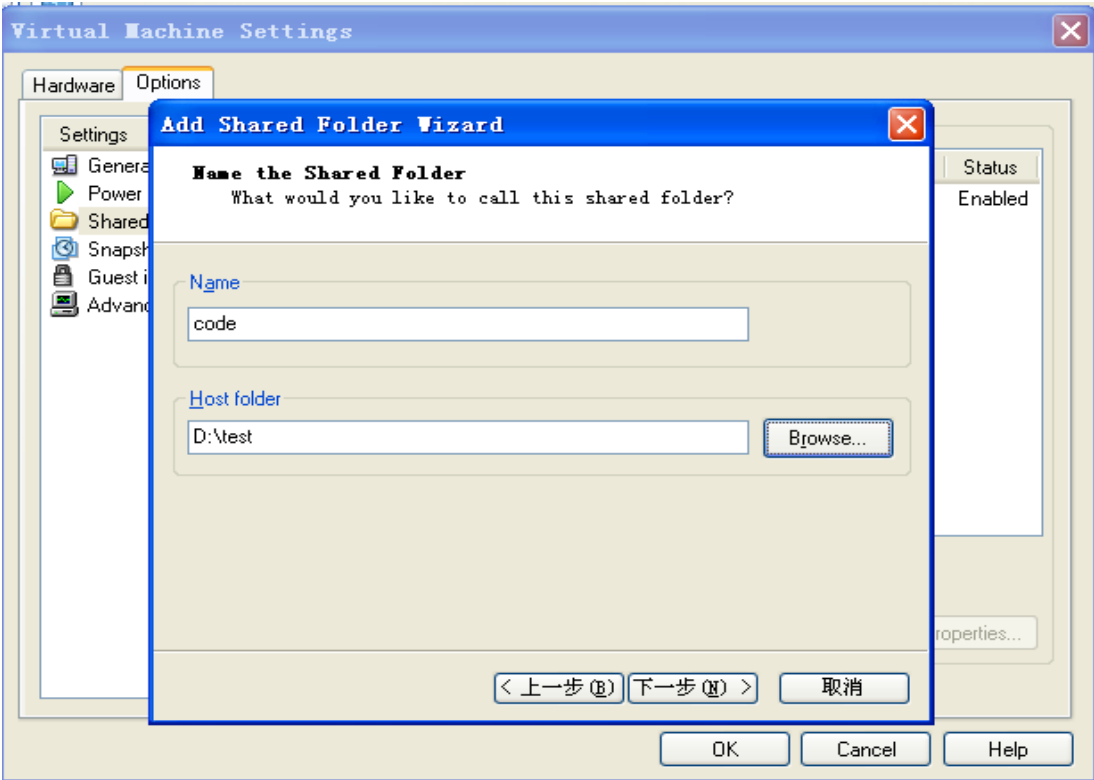


图 1.1

安装服务包 （点击 VM—Install Vmware tools），如图 1.2.

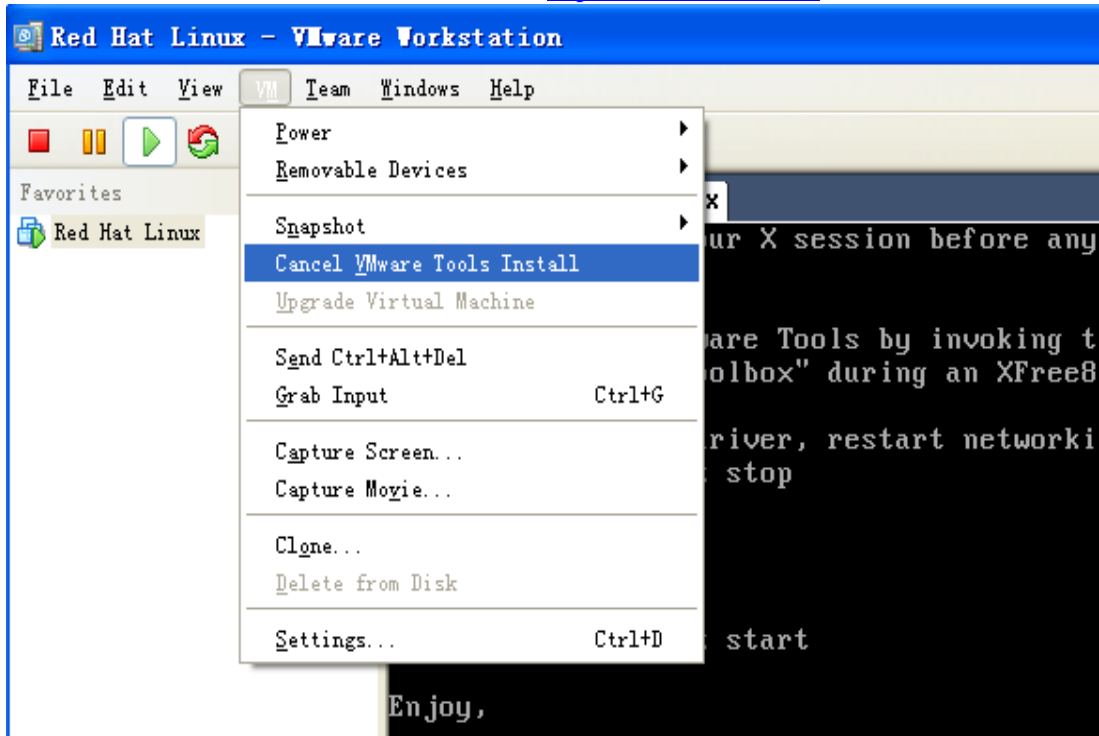


图 1.2

进到目录 `cd /mnt/hgfs` 中，然后就可以看到 Windows 共享的文件夹了

1.5.5 其他常用命令

1. `clear` 清除屏幕

eg: `#clear`

2. `id` 看用户信息组号

eg: `#id`

3. `file` 查看文件类型

eg: `#file /etc/profile, #file /etc`

4. `which` 找命令

eg: `#which ls, #which cat` (查找命令的存放地方)

5. 查看帮助信息: `man` 命令 `info` 命令 命令 `--help`

下面对 `man` 命令作一下详细说明:

Linux 的 `man` 帮助手册很强大，该手册分成很多 section，使用 `man` 时可以指定不同的 section 来浏览，各个 section 意义如下:

- 1 是普通的命令
- 2 是系统调用,如 `open,write` 之类的(通过这个，至少可以很方便的查到调用这个函数，需要加什么头文件)
- 3 是库函数,如 `printf,fread`
- 4 是特殊文件,也就是 `/dev` 下的各种设备文件
- 5 是指文件的格式,比如 `passwd`，就会说明这个文件中各个字段的含义
- 6 是给游戏留的,由各个游戏自己定义
- 7 是附件还有一些变量,比如向 `environ` 这种全局变量在这里就有说明
- 8 是系统管理用的命令,这些命令只能由 `root` 使用,如 `ifconfig`

想要指定 section 就直接在 `man` 的后面加上数字,比如 :

`man 1 ls`

`man 2 read`

`man 3 printf`

对于像 open,kill 这种既有命令,又有系统调用的来说,man open 则显示的是 open(1),也就是从最前面的 section 开始,如果想查看 open 系统调用的话,就得 man 2 open

6. logout 与 exit 退出登陆

7. more 分页显示内容很长的文件 less 正統的文件内容查看工具,分页显示,功能比 more 更强大 选项:
-num 规定屏幕大小 +num 从哪行开始显示

8. head -n 6, tail -n 6 显示尾部 6 行

9. 通配符 *: 任意 0 至多个 ? : 任意 1 个 [abc]:出现在中间任意 1 个

[a-z][a-z0-9] [!abc...]: 和上面的相反,表示除[]内的字符外的任意一个字符。

10. :set nu 打开行号 :set nonu 取消行号

11. :set hls 打开高亮 :set nohls 取消高亮

12. indent hello.c 格式化 c 代码 gg=G

13. 在命令模式中输入 gg=G 表示自动对齐 time ./main 可以查看运行的时间

14. Ctrl - 表示缩小 Ctrl Shift + 表示变大, 可以将终端的字体变大变小

15. Linux 中,复制是 Ctrl+Insert,粘贴是 Shift+Insert,也可以选中再用鼠标右键操作,在 vi 中复制粘贴的时候最好在编辑模式下进行,不然总是会有开头几个字节没有复制或粘贴过来。

16. env 命令可以查看环境变量。

1.5.6 补充: vi 换行自动缩进对齐以及 vim 中设置多窗口

vi 自动缩进对齐:

只要在 /etc/vimrc 中加上这两句就行了

```
set autoindent
```

```
set smartindent
```

详细说明:

在终端下使用 vim 进行编辑时,默认情况下,编辑的界面上是没有显示行号、语法高亮度显示、智能缩进等功能的。为了更好的在 vim 下进行工作,需要手动设置一个配置文件: .vimrc。

在启动 vim 时,当前用户根目录下的.vimrc 文件会被自动读取,该文件可以包含一些设置甚至脚本,所以,一般情况下把.vimrc 文件创建在当前用户的根目录下比较方便,即创建的命令为:

```
$vi ~/.vimrc. 设置完后$:x 或者 $wq 进行保存退出即可。
```

下面给出一个例子,其中列出了经常用到的设置,详细的设置信息请参照参考资料:

双引号开始的行为注释行,下同

去掉讨厌的有关 vi 一致性模式,避免以前版本的一些 bug 和局限:

```
set nocompatible
```

显示行号:

```
set number
```

检测文件的类型:

```
filetype on
```

记录历史的行数:

```
set history=1000
```

语法高亮度显示:

```
syntax on
```

下面两行在进行编写代码时,在格式对起上很有用:

第一行, vim 使用自动对起,也就是把当前行的对起格式应用到下一行

第二行,依据上面的对起格式,智能的选择对起方式,对于类似 C 语言编写很有用

```
set autoindent
```

```
set smartindent
```

第一行设置 tab 键为 4 个空格,第二行设置当行之间交错时使用 4 个空格:

```
set tabstop=4
```

```
set shiftwidth=4
```

设置匹配模式，类似当输入一个左括号时会匹配相应的那个右括号：

```
set showmatch
```

去除 vim 的 GUI 版本中的 toolbar：

```
set guioptions=T
```

当 vim 进行编辑时，如果命令错误，会发出一个响声，该设置去掉响声：

```
set vb t_vb=
```

在编辑过程中，在右下角显示光标位置的状态行：

```
set ruler
```

默认情况下，寻找匹配是高亮度显示的，该设置关闭高亮显示：

```
set nohls
```

查询时非常方便，如要查找 book 单词，当输入到 /b 时，会自动找到第一：

一个 b 开头的单词，当输入到 /bo 时，会自动找到第一个 bo 开头的单词，依次类推，进行查找时，使用此设置会快速找到答案，当你找要匹配的单词时，别忘记回车：

```
set incsearch
```

修改一个文件后，自动进行备份，备份的文件名为原文件名加 “~” 后缀：

```
if has( "vms" ) //注意双引号要用半角的引号 " "：
```

```
set nobackup
```

```
else
```

```
set backup
```

```
endif
```

如果去除注释后，一个完整的 .vimrc 配置信息如下所示：

```
set nocompatible
```

```
set number
```

```
filetype on
```

```
set history=1000
```

```
syntax on
```

```
set autoindent
```

```
set smartindent
```

```
set tabstop=4
```

```
set shiftwidth=4
```

```
set showmatch
```

```
set guioptions-=T
```

```
set vb t_vb=
```

```
set ruler
```

```
set nohls
```

```
set incsearch
```

```
if has("vms")
```

```
set nobackup
```

```
else
```

```
set backup
```

```
endif
```

vi 中分割窗口

要想在 vim 中设置多窗口，只需在 vi 的命令模式下输入 :vsplit，vim 以垂直方向分割一个新窗口，两个窗口可以实现不同的任务，打开或编辑不同的文件。要在多个窗口切换，连续按两次 ctrl+w，即可切换到下一个窗口。

1.5.7 Shell 编程

如果我们有一系列经常使用的 Linux 命令，我们可以把它们存储在一个文件中。Shell 可以读取这个文件并执行其中的命令。这样的文件被称为脚本文件

执行 shell 脚本

要创建一个 shell 脚本，我们要使用任何编辑器比如 vi 在文本文件中编写它，保存的文件最好是.sh 后缀的。

如：vi aa.sh

```
chmod +x aa.sh
```

```
bash aa.sh 或 ./aa.sh 或 sh aa.sh
```

shell 脚本的编写语法如下：

1.程序往往以下的行开始#!/bin/bash

2.注释 #

3.shell 变量

shell 变量没有数据类型，都是字符串，即使数值也是字符串。

创建变量：变量名称=值。如果值有空格则必须用""或者''引用起来。

示例： a="hello" (=号两边不能有空格)

引用变量：echo \$a 或 echo \${a} 或 echo "\${a}" 注意""'的区别（单引号：消除所有字符的特殊意义；双引号：消除除\$、""、''三种以外其它字符的特殊意义）。

```
1>: #echo $a          →hello 等同于#echo ${a} #echo "${a}"
```

```
2>: #echo "hello b$a"  →hello b,因为此时把 aa 作为一个整体变量，而且没有定义，所以输出前面的字符串
```

```
3>: #echo "hello b${a}" →hello bhelloa
```

```
4>: #echo "${a}"        →helloa
```

```
5>: #echo "${a}"        →${a}a,因为''会消除特殊字符的意义
```

```
6>: #echo "${a}"        →\${a}a.
```

删除变量：unset 变量名 eg: unset a。

还可以设置变量为只读变量 readonly a=3。

也可以允许用户从键盘输入，实现程序交互：read a。

echo \$? 用于显示上一条命令的执行结果（0 表示成功，1 表示失败），或者函数返回值。

转义符 #a=What's your \topic\ (→#a="What's your \"topic\"") #echo \$a.

命令替换 date 显示日期 echo `date` (小飘号) 或 echo \$(date) 显示当前系统时间，即用系统变量时，用 echo \$(命令)的形式等价于 echo `命令` eg:echo `pwd` →echo \$(pwd).

表达式计算: echo \$(expr 4 + 5) 或 echo `expr 4 + 5` 或 echo \$((4 + 5)) 或 echo \$[4 + 5] 或 expr 4 + 5。

举例：写 1.sh 要求读入 1 个目录名，在当前目录下创建该目录，并复制 etc 下的 conf 文件到该目录，统计 etc 下所有目录的数目到 etcdire.txt 中。

```
#!/bin/bash
```

```
#this is my first shell project
```

```
read dir
```

```
mkdir ${dir}
```

```
cp -rf /etc/*conf ${dir}
```

```
ls -l /etc/* | grep ^d | wc -l > etcdire.txt
```

4.标准变量或环境变量：系统预定义的变量，一般在/etc/profile 中进行定义

HOME 用户主目录 PATH 文件搜索路径

PWD 用户当前工作目录 PS1、PS2 提示符

还有 UNAME HOSTNAME LOGNAME 等

如：#echo \$PWD

用 echo \$PATH 显示，用 env 看环境所有变量，用 env | grep "name"查找。

用"export"进行设定或更改为全局变量，用unset 变量名 →取消全局变量的定义。

例: 定义本地变量 `name="Red Hat Linux"` `export name` 把 `name` 变为全局变量。

`sh` 进入子 shell `echo ${name}` 全局变量可以作用于子进程, 而本地变量不可以。

或直接输出 `export name="Red Hat Linux"`

`bash` 退出子 shell, 进入父 shell。

设置环境变量: 比如把 `/etc/apache/bin` 目录添加到 `PATH` 中:

1) `#PATH=$PATH:/etc/apache/bin`

2) `vi /etc/profile` 在里面添加 `PATH=$PATH:/etc/apache/bin`

3) `vi ~/.bash_profile` 在里面修改 `PATH` 行, 把 `/etc/apache/bin` 加进去, 此种方法针对当前用户有效。

5. 特殊变量

`$1, $2... $n` 传入的参数, `$0` 表示 shell 程序名称 → 每一项相当于 `main` 函数中 `argv[i]`。

`#` 传递到脚本的参数列表, 或表示参数个数 → 等价于 `main` 函数中的 `argc-1`。

`$@` 传入脚本的全部参数 → `argv[1] ---- argv[n-1]`。

`$*` 显示脚本全部参数

`$?` 前个命令执行情况 0 成功 1 失败

`$$` 脚本运行的当前进程号

`$!` 运行脚本最后一个命令

举例:

```
vi 1.sh
#!/bin/bash
echo $1
echo $2
echo $3
echo $#
echo $@
echo $*
echo $$
exit 3
./1.sh 1 2 hello "hello world"
echo $?
```

6. 变量赋值有五种格式

`${variable name}` 显示实际值

`${variable name:+value}` 如果设置了 `variable name`, 就把 `value` 值显示, 未设置显示空。

`${variable name:-value}` 如果设置了 `variable name`, 就显示 `name` 值, 未设置就显示 `value` 值;

`${variable name:?value}` 未设置提示用户错误信息 `value`;

`${variable name:=value}` 如未设置显示 `value` 值。

举例:

```
#a=1
#echo ${a}      #echo ${b}
#echo ${a:+2}    #echo ${b:+2}
#echo ${a:-2}    #echo ${b:-2}
#echo ${a:?2}     #echo ${b:?2}
#echo ${a:=2}     #echo ${b:=2}
```

7. 运算符与表达式

算术运算符(+、-、*、/、%)

逻辑运算符(&&、||、>、==、<、!=)

赋值运算符(=、+=、-=、*=、/=、%=、&=、^=、|=、<<=、>>=)

计算表达式有四种: 1、`$(())` 2、`$[]` 3、`let var=` 4、`expr 4 + 5`

`echo $[$v1 < $v2]` 计算逻辑表达式(用 1 表示 true, 用 0 表示 false)

echo [(\$v1<\$v2)&&(\$v1>\$v2)] 计算逻辑表达式

v3=2

let v3*=((\$v1+\$v2))

echo \$v3 或 echo \${v3}

举例：写 2.sh 要求输入 2 个数 计算 2 个数的和

```
#!/bin/bash
```

```
#this is my second shell project
```

```
echo "please input the first number:"
```

```
read a
```

```
echo "please input the second number:"
```

```
read b
```

```
c=$((a + b))
```

```
echo "The result of $a + $b is $c"
```

8.Test 命令的用法

VAR=2 test \$VAR -gt 1 echo \$?

也可以简写为[\$VAR -gt 3] echo \$? (0 表示真, 1 表示假)

1) 判断表达式 and or

test 表达式 1 -a 表达式 2 两个表达式都为真

test 表达式 1 -o 表达式 2 两个表达式有一个为真

测试是否是闰年: test \$(((\$iYear % 400)) -eq 0 -o \$(((\$iYear % 4)) -eq 0 -a \$(((\$iYear % 100)) -ne 0

2) 判断字符串

test -n 字符串 字符串的长度非零

-z 字符串长度为零 ==字符串相等 != 字符串不等

a="abc" test \$a == "abc" echo \$? (0) test \$a == "afd" echo \$? (1)

3) 判断整数

test 整数 1 -eq 整数 2 整数相等

-ge 大于等于 -gt 大于 -le 小于等于 -lt 小于 -ne 不等于

4) 判断文件

test File1 -ef File2 两个文件具有同样的设备号和 i 结点号

test File1 -nt File2 文件 1 比文件 2 新

test File1 -ot File2 文件 1 比文件 2 旧

test -b File 文件存在并且是块设备文件

test -c File 文件存在并且是字符设备文件

test -d File 文件存在并且是目录

test -e File 文件存在

test -f File 文件存在并且是普通文件

test -g File 文件存在并且是设置了组 ID

test -G File 文件存在并且属于有效组 ID

test -h File 文件存在并且是一个符号链接 (同-L)

test -k File 文件存在并且设置了 sticky 位

test -L File 文件存在并且是一个符号链接 (同-h)

test -o File 文件存在并且属于有效用户 ID

test -p File 文件存在并且是一个命名管道

test -r File 文件存在并且可读

test -s File 文件存在并且是一个套接字

test -t File 文件描述符是在一个终端打开的

test -u File 文件存在并且设置了它的 set-user-id 位

test -w File 文件存在并且可写

test -x File 文件存在并且可执行

举例：

```
a=2
test $a -ge 3
echo $?
```

9. 数组

定义 1: a=(1 2 3 4 5)下标从 0 开始 各个数据之间用空格隔开

定义 2: a[0]=1;a[1]=2;a[2]=3

定义 3: a=([1]=1 [2]=2)

引用 \${a[1]}

\${#a[@]} 数组长度 → \${#a[*]}

\${a[@]:1:2} 截取下标 1 开始的 2 个

例子

```
a=(2 5 7 10)
echo ${a[2]} #输出下标为 2 的数据
echo ${#a[*]} #输出数组的长度
echo ${a[@]:2} #截取下标从 2 到最后
echo ${a[@]:1:2} #截取从下标 1 后面 2 个
```

```
#!/bin/bash
a=(3 10 6 5 9 2 8 1 4 7)
x=0
while [ $x -lt ${#a[*]} ]
do
    echo ${a[$x]}
    x=$((x + 1))
done
```

```
#!/bin/bash
a=(3 10 6 5 9 2 8 1 4 7)
i=0
while (( i<10 )) #类似 C 语言的写法
do
    echo ${a[i]}
    i=$((i+1))
done
```

10. if 语句

if [condition] then action fi 只有当 condition 为真时, 该语句才执行操作, 否则不执行操作, 并继续执行 “fi” 之后的任何行。

if [condition] then action elif [condition2] then action2 . . . elif [condition3] then else actionx fi

在使用时, 将 “if” 和 “then” 放在不同行, 如同行放置, 则 if 语句必须要; 结束

举例: 用参数传 1 个文件名, 该文件如果是文件并且可读可写就显示该文件, 如果是目录就进入该目录, 并判断 ls.sh 存在否, 如果不存在就建立 1 个 ls.sh 的文件并运行该文件。

该文件的内容是 ls -li /etc > etc.list

```
#!/bin/bash
if [ -f $1 -a -r $1 -a -w $1 ] //判断是普通文件并可读可写
then
    cat $1 #显示文件内容
```

```
elif [ -d $1 ]    #否则如果是目录
then
    cd $1        #进入目录
    if [ -e ls.sh ] #如果 ls.sh 该文件存在
    then
        chmod +x ls.sh    #赋予可执行的权限
        ./ls.sh          #执行
    else
        touch ls.sh    #如果不存在则创建 ls.sh
        echo "#!/bin/bash" >> ls.sh    #将程序写入 ls.sh 中保存
        echo "ls -li /etc > etc.list" >> ls.sh    #将要执行的命令写入 ls.sh 中保存
        chmod +x ls.sh    #赋予可执行的权限
        ./ls.sh
    fi
fi
```

11.case 语句

case 常用的语法形式如下:

```
case $1 in
    "1")
        echo you inputed "1"
        ;;
    "2")
        echo you inputed "2"
        ;;
    *)
        echo you inputed other number
        ;;
esac
```

例子 1: 智能解压文件

```
#!/bin/sh
ftype=`file "$1"`    #file 用于判断$1 的文件类型,并保存到 ftype 中
case "$ftype" in
    "$1: Zip archive"*)
        unzip "$1"
        ;;
    "$1: gzip compressed"*)
        gunzip "$1"
        ;;
    "$1: bzip2 compressed"*)
        bunzip2 "$1"
        ;;
    *)
        echo "File $1 can not be uncompressed with smartzip"
        ;;
esac
```

例子 2

```
echo "Is it morning? Please answer yes or no."
read YES_OR_NO
case "$YES_OR_NO" in
```

```

yes|y|Yes|YES)
    echo "Good Morning!";
[nN]*) /* 表示 n 或 N 开头的任意字段 */
    echo "Good Afternoon!";
*)
    echo "Sorry, $YES_OR_NO not recognized. Enter yes or no."
    exit 1;;
esac

```

例子 3: 编写一个加减乘除取模计算器

```

echo "please input the first number:"
read a
echo "please input the second number:"
read b
echo "please input your operator:"
read c
case $c in
    "+")
        echo "the result of $a + $b is $((($a + $b)))"
        ;;
    "-")
        echo "the result of $a - $b is $((($a - $b)))"
        ;;
    "*")
        echo "the result of $a * $b is $((($a * $b)))"
        ;;
    "/" )
        echo "the result of $a / $b is $((($a / $b)))"
        ;;
    *)
        echo "no true operator!"
        ;;
esac

```

12. for 循环

例子 1:

```

for x in one two three four
do
    echo number $x
done

```

例子 2:

```

for x in /etc/???????? /var/lo* /home/* ${PATH} //列举
do
    echo $x
done

```

例子 3: /etc/r*中的文件和目录

```

for myfile in /etc/r*
do
    if [ -d "$myfile" ]
    then
        echo "$myfile(dir)"
    fi
done

```

```
    else
        echo "$myfile"
    fi
done
```

例子 4:

```
for x in /var/log/*
do
    echo `basename $x` is a file living in /var/log
done
```

basename 和 dirname 的用法:

basename /var/log → log 去当前的文件或目录名
dirname /var/log → /var 取上一级的路径名

例子 5: //冒泡排序

```
#!/bin/bash
a=(3 10 6 5 9 2 8 1 4 7)
for (( i=1; i<10; i++ ))
do
    for (( j=0; j<10-i; j++ ))
    do
        if [ ${a[j]} -gt ${a[j+1]} ]
        then
            temp=${a[j]}
            a[j]=${a[j+1]} //或者 a[j]=${a[`${j+1}`]}
            a[j+1]=$temp
        fi
    done
done

for (( i=0; i<10; i++ ))
do
    echo ${a[i]}
done
```

13. While 语句

```
myvar=0
while [ $myvar -ne 10 ]
do
    echo $myvar
    myvar=$((myvar+1))
done
```

举例:

```
#!/bin/bash
#this is my first shell project
loopcount=0
result=0
while [ $loopcount -lt 100 ]
do
    loopcount=$((loopcount + 1))
```

```
result=$((loopcount + $result))
done
echo "The result of \'1+2+3+...+100\' is $result"
```

14.until 语句

```
myvar=0
until [ $myvar -eq 10 ]
do
    echo $myvar
    myvar=$((myvar+1))
done
```

15.Shell 函数

函数名(){ 命令 1 ... }

```
function 函数名() { ... }
```

#declare a function named hello

```
function hello()
```

```
{
```

```
    echo "Hello,$1 today is `date`"
```

```
    return 11
```

```
}
```

```
echo "now going to the function hello"
```

```
hello "I LOVE CHINA"
```

```
echo $?
```

```
echo "back from the function"
```

用函数求两个数的和（类似于 C 语言的 `int add(int a, int b){ return a+b}`）

```
#!/bin/bash
```

```
function add()
```

```
{
```

```
    return $((1+$2))
```

```
}
```

```
a=10
```

```
b=20
```

```
add a b
```

```
echo $?
```

综合例子：

制作菜单：

```
#!/bin/bash
```

```
while true    #表示死循环
```

```
do
```

```
echo "====="
```

```
echo "*****student system*****"
```

```
echo "    1.create operator    "
```

```
echo "    2.insert operator    "
```

```
echo "    3.delete operator    "
```



```
echo "      4.append operator      "
echo "      5.update operator      "
echo "      6.find   operator      "
echo "      7.print   operator      "
echo "      0.exit   operator      "
echo "*****"
echo "===== "
echo "please input your operator:"
read op
clear    //清屏
case $op in
    "1")
        echo "create student succeed!"
        #clear
        ;;
    "2")
        echo "insert student succeed!"
        #clear
        ;;
    "3")
        echo "delete student succeed!"
        #clear
        ;;
    "4")
        echo "append student succeed!"
        #clear
        ;;
    "5")
        echo "update student succeed!"
        #clear
        ;;
    "6")
        echo "find student succeed!"
        #clear
        ;;
    "7")
        echo "print student succeed!"
        #clear
        ;;
    "0")
        exit -1
        #clear
        ;;
    *)
        echo "input error ,please input again!"
        #clear
        ;;
esac
done
```

第2章 LINUX 下编译与调试

2.1 gcc/g++编译器

对于.c 格式的 C 文件, 可以采用 gcc 或 g++编译

对于 .cc、.cpp 格式的 C++文件, 应该采用 g++进行编译

常用的选项:

- c 表示编译源文件
- o 表示输出目标文件
- g 表示在目标文件中产生调试信息, 用于 gdb 调试
- D<宏定义> 编译时将宏定义传入进去
- Wall 打开所有类型的警告。

2.1.1 gcc/g++编译过程

1. 预编译→编译→汇编→链接

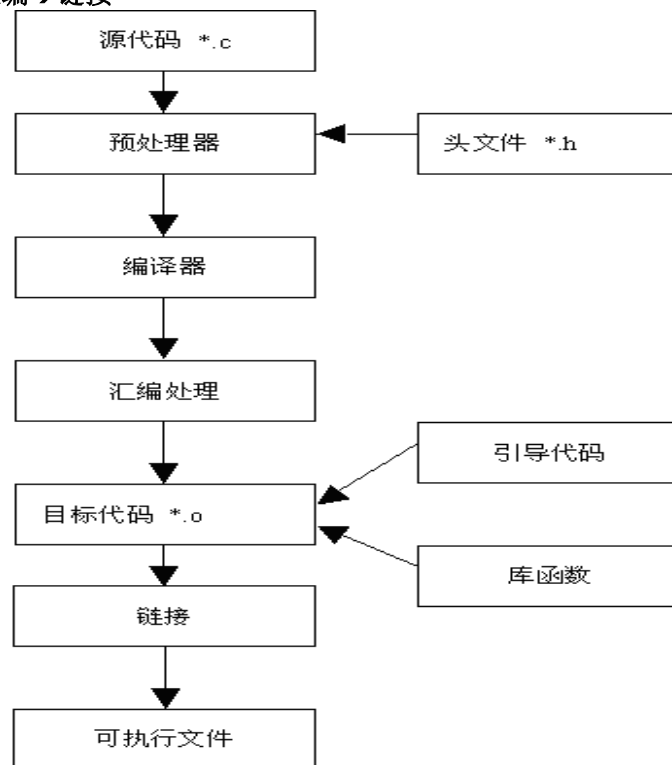


图 2.1

当我们进行编译的时候,要使用一系列的工具,我们称之为工具链.其中包括:预处理器,编译,汇编器 as,连接器.一个编译过程可以用图 2.1 来表示, 包括下面几个阶段:

- (1)预处理: 预处理器将对源文件中的宏进行展开。
- (2)编译: gcc 将 c 文件编译成汇编文件。
- (3)汇编: as 将汇编文件编译成机器码。
- (4)链接: 将目标文件和外部符号进行连接, 得到一个可执行二进制文件。

下面以一个很简单的 test.c 来探讨这个过程。

```
#include <stdio.h>
#define NUMBER (1 + 2)
int main()
{
```

```

int x = NUMBER;

return 0;

}
    
```

(1)预处理：gcc -E test.c -o test.i 我们用 cat 查看 test.i 的内容如下：int main() int x=(1+2); return 0;我们可以看到，文件中宏定义 NUMBER 出现的位置被(1+2)替换掉了，其它的内容保持不变。

(2)编译：gcc -S test.i -o test.s 通过 cat test.s 查看 test.s 的内容为汇编代码。

(3)汇编：as test.s -o test.o 利用 as 将汇编文件编译成机器码。得到输出文件为 test.o。 test.o 中为目标机器上的二进制文件。用 nm 查看文件中的符号：nm test.o 输出如下：00000000 T main。有的编译器上会显示：00000000 b .bss 00000000 d .data 00000000 t .text U __main U __alloca 00000000 T _main 既然已经是二进制目标文件了，能不能执行呢？试一下 ./test.o,提示 cannotexecute binary file.原来__main 前面的 U 表示这个符号的地址还没有定下来，T 表示这个符号属于代码。

(4)链接：gcc -o test test.o,把所有的.o 文件链接起来生成可执行程序。

2. gcc 所支持后缀名：如下图 2.2

后 缀 名	所对应的语言	后 缀 名	所对应的语言
.c	C 原始程序	.s/.S	汇编语言原始程序
.C/.cc/.cxx	C++原始程序	.h	预处理文件（头文件）
.m	Objective-C 原始程序	.o	目标文件
.i	已经过预处理的 C 原始程序	.a/.so	编译后的库文件
.ii	已经过预处理的 C++原始程序		

图 2.2

3. gcc 常用选项：如下图 2.3

选项	含义
-c	只编译不链接，生成目标文件“.o”
-S	只编译不汇编，生成汇编代码
-E	只进行预编译，不做其他处理
-g	在可执行程序中包含标准调试信息
-o file	指定将 file 文件作为输出文件
-v	打印出编译器内部编译各过程的命令行信息和编译器的版本
-I dir	在头文件的搜索路径列表中添加 dir 目录

图 2.3

预处理阶段：对包含的头文件（#include）和宏定义（#define、#ifdef 等）进行处理

gcc -E hello.c -o hello.i // -o 表示输出为指定文件类型 -E 将源文件（*.c）转换为（*.i）

编译阶段：检查代码规范性、语法错误等，在检查无误后把代码翻译成汇编语言

gcc -S hello.i -o hello.s // -S 将已预处理的 C 原始程序（*.i）转换为（*.s）

链接阶段：将.s 的文件以及库文件整合起来链接为可执行程序

gcc -o hello.exe hello.s //最后将汇编语言原始程序(*.s)和一些库函数整合成 (*.exe)

示例 1：

```

#include <stdio.h>

#define MAX 100
    
```

```
#define max(a, b) ((a) > (b) ? (a) : (b)) //宏定义，执行-E 之后被替换
main()
```

```
{
    printf("MAX=%d\n", MAX);
    printf("max(3,4)=%d\n", max(3, 4));
}
```

//法一：

```
gcc -E project1.c -o project1.i      //预编译，生成已预编译过的 C 原始程序*.i
gcc -S project1.i -o project1.s      //编译，生成汇编语言原始程序*.s
gcc -o project1.exe project1.s      //链接，生成可执行程序
```

//法二：

```
gcc -c project1.c -o project1.o      //编译
gcc -o project1.exe project1.o      //链接
```

//法三：

```
gcc -o project1.exe project1.c      //编译并链接
```

示例 2:

```
#include <stdio.h>
main()
{
    #ifdef lry //表示如果定义了 lry，即命令行参数传了 lry，就执行下面的输出
        printf("lry is defined!\n");
    #else
        printf("lry is not defined!\n");
    #endif
    printf("main exit\n");
}
```

```
gcc -E project2.c -o project2.i -D lry //条件编译，用-D 传递，如果没有传 lry 则执行#else
gcc -S project2.i -o project2.s
gcc -o project2.exe project2.s
或： gcc -o project2 project2.c -D lry
```

2.1.2 静态库和动态库

gcc 库选项如下图 2.4

选 项	含 义
-static	进行静态编译，即链接静态库，禁止使用动态库
-shared	1. 可以生成动态库文件 2. 进行动态编译，尽可能地链接动态库，只有没有动态库时才会链接同名的静态库（默认选项，即可省略）
-L dir	在库文件的搜索路径列表中添加 dir 目录
-lname	链接称为 libname.a（静态库）或者 libname.so（动态库）的库文件。若两个库都存在，则根据编译方式（-static 还是-shared）而进行链接
-fPIC（或-fpic）	生成使用相对地址的位置无关的目标代码（Position Independent Code）。然后通常使用 gcc 的-static 选项从该 PIC 目标文件生成动态库文件。

图 2.4

函数库分为静态库和动态库。

静态库是目标文件.a 的归档文件（格式为 libname.a）。如果在编译某个程序时链接静态库，则链接器将会搜索静态库并直接拷贝到该程序的可执行二进制文件到当前文件中；

动态库（格式为 libname.so[.主版本号.次版本号.发行号]）。在程序编译时并不会被链接到目标代码中，而是在程序运行时才被载入。

创建静态库

```
$ gcc -c add.c //编译 add.c 源文件生成 add.o 目标文件
```

```
$ ar crsv libadd.a add.o //对目标文件*.o 进行归档，生成 lib*.a,此处 lib 要写
```

```
$ gcc -o main main.c -L./ -ladd -L/ //不要忘记-L 后面的那个。（即在库文件的搜索路径中添加当前路径 -ladd 表示链接库文件 libadd.a/.so -L/表示包含在当前目录中的头文件）
```

```
$/main
```

创建动态库

```
$ gcc -fPIC -Wall -c add.c
```

```
$ gcc -shared -o libadd.so add.o
```

```
$ gcc -o main main.c -L./ -ladd
```

在运行 main 前，需要注册动态库的路径。方法有 3 种：修改/etc/ld.so.conf 或修改 LD_LIBRARY_PATH 环境变量或将库文件拷贝到/lib 或者/usr/lib 下（系统默认搜索库路径）。

```
$ cp libadd.so /lib //通常采用的方法，→ cp lib*.so /lib
```

```
$ ./main
```

如果不拷贝，生成.so 之后还有两种方法：

```
gcc -o main main.c -L. -Wl,-rpath,${PWD} -ladd
```

在 main.c 中修改如下：

```
#include <stdio.h>
#include <dlfcn.h>
#include "add.h"
#define LIB_NAME "./libadd.so"
#define FLAGS RTLD_NOW //表示现在就替换
int main(int argc, char * argv[])
{
    void * handle = NULL;
    void (*func)(int, int); //typedef void (*func)(int, int);
    //open
    handle = dlopen(LIB_NAME, FLAGS);
    if (NULL == handle)
    {
        printf("open err!\n");
        return -1;
    }
    //find "my_printf"
    func = dlsym(handle, "add"); //add 表示具体的函数名字
    //run
    func(3, 4);
    //close
    dlclose(handle);
    return 0;
}
```

最后执行：gcc -o main main.c -ldl

创建动态链接库之后，以后就可以使用该动态链接库了

例如在 test.c 里面调用了原来库中的函数，则执行 gcc -o test test.c -lfunc 就可以了。

静态库与动态库的比较:

动态库只在执行时才被链接使用，不是直接编译为可执行文件，并且一个动态库可以被多个程序使用故可称为共享库。

静态库将会整合到程序中，在程序执行时不用加载静态库。因此，静态库会使你的程序臃肿并且难以升级，但比较容易部署。而动态库会使你的程序轻便易于升级但难以部署。

示例：写一个求两个数 +, --, *, / 的函数 func.c(func.h)，在 main.c 中调用执行相应的算术操作，但是不直接针对 main.c 编译链接，而是在 function.sh 中对 func 函数创建静态库和动态库，并分别自动执行 main 函数。（注：当然可以直接针对 main 函数）

1. 编写 function.sh

```
#!/bin/bash
echo "===== "
echo "*****"
echo "    1.create static lib      " //静态库创建
echo "    2.create shared lib     " //动态库创建
echo "*****"
echo "===== "
echo "please input your operator:"
read op
case $op in
    "1") //以静态库的方式
        gcc -c ${1}.c // ${1}接收第一个传进来的参数 func,并编译它
        ar rcsv lib${1}.a ${1}.o //将其打包为静态库
        gcc -o ${2} ${2}.c -L. -l${1} // ${2}接收第二个传进来的 main
        ./${2} //运行 main 程序输出结果
        ;;
    "2") //以动态库的方式
        gcc -fpic -c ${1}.c
        gcc -shared -o lib${1}.so ${1}.o
        gcc -o ${2} ${2}.c -L. -l${1}
        sudo cp lib${1}.so /lib/ //切换到 root 用户下
        ./${2}
        ;;
    *)
        exit 3
        ;;
esac
```

2. 编写 func.h

```
#ifndef __FUNC_H
#define __FUNC_H
extern int add(int, int);
extern int sub(int, int);
extern int mul(int, int);
extern int div(int, int);
#endif
```

3. 编写 func.c

```
#include "func.h"
int add(int a, int b)
{
```



```
    return a + b;
}
int sub(int a, int b)
{
    return a - b;
}
int mul(int a, int b)
{
    return a * b;
}
int div(int a, int b)
{
    return a / b;
}
```

4. 编写 main.c

```
#include <stdio.h>
#include "func.h"
main()
{
    printf("add(3,4)=%d\n", add(3, 4));
    printf("sub(4,1)=%d\n", sub(4, 1));
    printf("mul(3,2)=%d\n", mul(3, 2));
    printf("div(6,2)=%d\n", div(6, 2));
}
```

sh function.sh func main //将 func,main 作为参数传递进去

当然也可以不用写 function.sh，可以直接针对 main 操作，分别采用静态库和动态库的方式将函数 func.c 打包

1. 将 func.o 打包为静态库函数 libfunc.a，并执行程序：

```
# gcc -c func.c //1. 将 func.c 编译为 func.o
# ar rcs libfunc.a func.o //2. 用 ar rcs 将 func.o 打包为静态库 libfunc.a(前面的 lib 要写)
# gcc -o main.exe main.c -L. -lfunc //3. 链接库函数和执行 main.c 生成可执行程序 main.exe
# ./main.exe //4. 执行 ./main.exe
```

2. 将 func.o 打包为动态库函数 libfunc.so，并执行程序：

```
# gcc -fpic -c func.c //1. 用动态库的方式将 func.c 编译为 func.o
# gcc -shared -o libfunc.so func.o //2. 用 gcc -shared 将 func.o 打包为动态库 libfunc.so
# gcc -o main.exe main.c -L. -lfunc //3. 链接库函数和执行 main.c 生成可执行程序 main.exe
# sudo cp libfunc.so /lib //4. 非超级用户要用 sudo 将动态库 libfunc.so 拷贝到 /lib 目录下
# ./main.exe //5. 执行 ./main.exe
```

2.1.3 gcc 警告和优化选项

gcc --- 警告选项，如图 2.5

选 项	含 义
-ansi	支持符合 ANSI 标准的 C 程序
-pedantic	允许发出 ANSI C 标准所列的全部警告信息
-pedantic-error	允许发出 ANSI C 标准所列的全部错误信息
-w	关闭所有告警
-Wall	允许发出 gcc 提供的所有有用的报警信息
-Werror	把所有的告警信息转化为错误信息，并在告警发生时终止编译过程

图 2.5

对于如下程序：

```
#include <stdio.h>
void main()
{
    long long temp = 1;
    printf("This is a bad code!\n");
    return 0;
}
```

- -ansi：生成标准语法（ANSI C 标准）所要求的警告信息（并不列出所有警告）

```
$ gcc -ansi warning.c -o warning
```

warning.c: 在函数“main”中：

warning.c:7 警告：在无返回值的函数中，“return”带返回值

warning.c:4 警告：“main”的返回类型不是“int”

可以看出，该选项并没有发现“long long”这个无效数据类型的错误

- -pedantic：列出 ANSI C 标准的全部警告信息。

```
$ gcc -pedantic warning.c -o warning
```

warning.c: 在函数“main”中：

warning.c:5 警告：ISO C89 不支持“long long”

warning.c:7 警告：在无返回值的函数中，“return”带返回值

warning.c:4 警告：“main”的返回类型不是“int”

- -Wall：列出所有的警告信息（常用）

```
$ gcc -Wall warning.c -o warning
```

warning.c:4 警告：“main”的返回类型不是“int”

warning.c: 在函数“main”中：

warning.c:7 警告：在无返回值的函数中，“return”带返回值

warning.c:5 警告：未使用的变量“tmp”

```
$ gcc -Werror warning.c -o warning
```

通常用的是-Wall 显示所有有用的报警信息。

6. gcc --- 优化选项

gcc 对代码进行优化通过选项“-O_n”来控制优化级别（_n是整数）。不同的优化级别对应不同的优化处理工作。如使用优化选项“-O1”主要进行线程跳转和延迟退栈两种优化。使用优化选项“-O2”除了完成所有“-O1”级别的优化之外，还要进行一些额外的调整工作，如处理其指令调度等。选项“-O3”则还包括循环展开或其他一些与处理器特性相关的优化工作。虽然优化选项可以加速代码的运行速度，但对于调试而言将是一个很大的挑战。因为代码在经过优化之后，原先在源程序中声明和使用的变量很可能不再使用，控制流也可能会突然跳转到意外的地方，循环语句也有可能因为循环展开而变得到处都是，所有这些对调试来讲都是不好的。所以在调试的时候最好不要使用任何的优化选项，只有当程序在最终发行的时候才考虑对其进行优化。

通常用的是-O2

-D <宏定义> 编译时将宏定义传入进去

示例: # gcc -o hello -Wall -O2 hello.c

例: 有两个文件 main.cpp,func.cpp

其中 main.cpp 内容为:

```
#include <stdio.h>
int MyFunc();
int main()
{
#ifdef _DEBUG
    printf("Debug MyFunc is:%d\n", MyFunc());
#else
    printf("NDEBUG MyFunc is:%d\n", MyFunc());
#endif
}
func.cpp 内容为:
int MyFunc()
{
    return 123;
}
```

编译和连接:

1、 g++ -c func.cpp //C++文件类型的, 如.cpp 的一定要用 g++

将编译 func.cpp,并且生成同名的但扩展名为.o 的二进制目标文件 func.o

g++ -c main.cpp

将编译 main.cpp,并且生成同名的但扩展名为.o 的二进制目标文件 main.o

2、 g++ -c func.cpp -o func.o

g++ -c main.cpp -o main.o

编译 main.cpp,并输出目标文件 main.o

3、 链接

g++ main.o func.o //默认情况下生成的是 a.out 可执行文件

g++ -o a.out main.o func.o

g++ -o a.out *.o

都将连接目标文件 main.o 和 func.o 最后形成可执行文件 a.out

对于第一种, 如果没有显式指定可执行文件名, g++默认为 a.out

4、 也可以将编译和链接的过程合为一块处理:

g++ *.cpp

g++ func.cpp main.cpp (不加任何参数表示编译并链接生成可执行文件 a.out)

g++ -o a.out func.cpp main.cpp

都将先编译指定的源文件, 如果成功的话, 再链接成可执行文件 a.out

5、 如果希望在编译时传入宏定义, 可使用-D 参数,例如

g++ *.cpp -D _DEBUG

2.2 make 工程管理器和 Makefile

可以试想一下, 有一个上百个文件的代码构成的项目, 如果其中只有一个活少数几个文件进行了修改, 如果再从头到尾将每一个文件都重新编译是个比较繁琐的过程。为此, 引入了 Make 工程管理器的概念, 工程管理器指管理较多的文件, 它是自动管理器能根据文件时间自动发现更新过的文件而减少编译的工作量, 同时通过读入

Makefile 文件来执行大量的编译工作

makefile 格式

target: dependency_files //目标项:依赖项

< TAB >command //必须以 tab 开头, command 编译命令

注意点: 在写 command 命令行的时候, 必须要在前面按 TAB 键

例如, 有 Makefile 文件, 内容如下:

```
main.exe:main.o func.o
    g++ -o main.exe main.o func.o
main.o:main.cpp
    g++ -c main.cpp
func.o:func.cpp
    g++ -c func.cpp
```

使用 make 编译

对于该 Makefile 文件,程序 make 处理过程如下:

- make 程序首先读到第 1 行的目标文件 main.exe 和它的两个依赖文件 main.o 和 func.o;然后比较文件 main.exe 和 main.o/func.o 的产生时间, 如果 main.exe 比 main.o/func.o 旧的话, 则执行第 2 条命令, 以产生目标文件 main.exe。
- 在执行第 2 行的命令前,它首先会查看 makefile 中的其他定义, 看有没有以第 1 行 main.o 和 func.o 为目标文件的依赖文件, 如果有的话, 继续按照(1)、(2)的方式匹配下去。
- 根据(2)的匹配过程, make 程序发现第 3 行有目标文件 main.o 依赖于 main.cpp, 则比较主 main.o 与它的依赖文件 main.cpp 的文件新旧,如果 main.o 比 main.cpp 旧,则执行第 4 行的命令以产生目标文件 main.o. 在执行第 4 条命令时,main.cpp 在文件 makefile 不再有依赖文件的定义,make 程序不再继续往下匹配,而是执行第 4 条命令,产生目标文件 main.o
- 目标文件 func.o 按照上面的同样方式判断产生。
- 执行(3)、(4)产生完 main.o 和 func.o 以后, 则第 2 行的命令可以顺利地执行了, 最终产生了第 1 行的目标文件 main.exe。

2.2.1 Makefile 中特殊处理与伪目标

.PHONY 是 makefile 文件的关键字, 表示它后面列表中的目标均为伪目标

.PHONY:b

b:

echo 'b' #通常用@echo "hello"

伪目标通常用在清理文件、强制重新编译等情况下。

示例 1: main.c 函数, func.c 函数为前面计算+,-,*,/运算的程序

#vi Makefile #系统默认的文件名为 Makefile

main.exe:main.o func.o #表示要想生成 main.exe 文件, 要依赖于 main.o 和 func.o 文件

gcc -o main.exe main.o func.o#如果 main.o,func.o 已经存在了, 就链接成 main.exe

main.o:main.c #表示 main.o 文件依赖于 main.c 文件

gcc -c main.c #编译 main.c, 默认生成 main.o。可写为: gcc -c main.c -o main.o

func.o:func.c #表示 func.o 文件依赖于 func.c 文件

gcc -c func.c #如果 func.c 存在, 则编译 func.c ,生成 func.o

.PHONY:rebuild clean #表示后面的是伪目标, 通常用在清理文件、强制重新编译等情况下

rebuild:clean main.exe #先执行清理, 在执行 main.exe

clean:

rm -rf main.o func.o main.exe #最后删除.o 和.exe 的文件

按 ESC 键之后, :wq 保存退出, 注意: 和 shell 编程相同, 在 Makefile 中, 也用“#”号表示注释

```
#make          //直接 make,即从默认文件名（Makefile）的第一行开始执行
#make clean    //表示执行 clean: 开始的命令段
#make func.o   //表示执行 func.o: 开始的命令段
#make rebuild  //则先执行清除，再重新编译连接
```

如果不用系统默认的文件名 Makefile，而是用户随便起的一个名字，如：

```
#vi Makefile11
则 make 后面必须要加上 -f Makefile11 ，如：
#make -f Makefile11 clean    //表示执行 clean: 开始的命令段
#make -f Makefile11 main.exe //表示执行 main.exe: 开始的命令段
```

2.2.2 变量、规则与函数

随着软件项目的变大、变复杂，源文件也越来越多，如果采用前面的方式写 makefile 文件，将会使 makefile 也变得复杂而难于维护。通过 make 支持的变量定义、规则和内置函数，可以写出通用性较强的 makefile 文件，使得同一个 makefile 文件能够适应不同的项目。

变量：用来代替一个文本字符串

定义变量的 2 种方法：

变量名=变量值 递归变量展开(几个变量共享一个值) //不常用
变量名:=变量值 简单变量展开(类似于 C++的赋值) //通常采用这种形式

使用变量的一般方法：\$(变量名)=??? 赋值

???=\$(变量名) 引用

例：将以前的那个可以写为：

```
OBJS:=main.o func.o//相当于 main.o func.o
EXE:=main.exe
$(EXE):$(OBJS)
    g++ -o $(EXE) $(OBJS)
main.o:main.cpp
    g++ -c main.cpp -o main.o
func.o:func.cpp
    g++ -c func.cpp -o func.o
clean:
    rm -rf $(EXE) $(OBJS)
```

变量分为：用户自定义变量，预定义变量，自动变量，环境变量。

自动变量：指在使用的时候，自动用特定的值替换。

常用的如表 2.1

表 2.1

变量	说明
\$@	当前规则的目标文件
\$<	当前规则的第一个依赖文件
\$^	当前规则的所有依赖文件，以逗号分隔
\$?	规则中日期新于目标文件的所有相关文件列表，逗号分隔
\$(@D)	<u>目标文件的目录名部分</u>
\$(@F)	<u>目标文件的文件名部分</u>

示例：用自动变量：

```
OBJS:= main.o func.o/$(OBJS)相当于 main.o func.o (原样替换)
```

```

EXE:= main.exe
CFLAGS:= -Wall -O2 -fpic    #显示所有警告信息，优化级别为 2
LIBFUNCSO:= libfunc.so      #动态库
LIBFUNCA:= libfunc.a        #静态库
$(EXE):$(OBJS) $(LIBFUNCSO) $(LIBFUNCA)
    gcc -o $@ $< -L. -lfunc
main.o: main.c
    gcc -c $(CFLAGS) $< -o $@
func.o: func.c
    gcc -c $(CFLAGS) $< -o $@
libfunc.a: func.o
    ar rcsv $@ $<
libfunc.so: func.o
    gcc -shared -o $@ $<
    cp -f $@ /lib
.PHNOY:rebuild clean
rebuild:clean $(EXE)
clean:
    rm -rf $(EXE) $(OBJS) $(LIBFUNCSO) $(LIBFUNCA)

```

预定义变量：内部事先定义好的变量，但是它的值是固定的，并且有些的值是为空的。

AR：库文件打包程序默认为 ar

AS：汇编程序，默认为 as

CC：c 编译器默认为 cc

CPP：c 预编译器，默认为\$(CC) -E

CXX：c++编译器，默认为 g++

RM：删除，默认为 rm -f

ARFLAGS：库选项，无默认

ASFLAGS：汇编选项，无默认

CFLAGS：c 编译器选项，无默认

CPPFLAGS：c 预编译器选项，无默认

CXXFLAGS：c++编译器选项

根据内部变量，可以将 makefile 改写为：

```

OBJS:=main.o func.o
CC:=g++
main.exe:$(OBJS)
    $(CC) -o $@ $^
main.o:main.cpp
    $(CC) -o $@ -c $^
func.o:func.cpp
    $(CC) -o $@ -c $^

```

规则分为：普通规则，隐含规则，模式规则

隐含规则：/*.o 文件自动依赖*.c 或*.cc 文件，所以可以省略 main.o:main.cpp 等

```
OBJS := main.o fun.o
```

```
CFLAGS := -Wall -O2 -g
```

```
main.exe: $(OBJS)
```

深圳信盈达科技有限公司 专业提供单片机、嵌入式、ARM、LINUX、Android、PCB、FPGA 等技术培训、技术方案。


```
gcc $^ -o $@
```

模式规则：通过匹配模式找字符串， % 匹配 1 或多个任意字符串

% .o: %.cpp 任何目标文件的依赖文件是与目标文件同名的并且扩展名为.cpp 的文件

```
OBJS := main.o fun.o
```

```
CFLAGS := -Wall -O2 -g
```

```
main.exe : $(OBJS)
```

```
gcc $^ -o $@
```

```
%.o: %.cpp      #模式通配
```

```
gcc -o $@ -c $^
```

另外还可以指定将*.o、*.exe、*.a、*.so 等编译到指定的目录中：

```
DIR:=./Debug/
```

```
EXE:=main.exe
```

```
OBJS:=main.o
```

```
LIBFUNCSO:=libfunc.so
```

```
CFLAGS:= -fpic
```

```
$(DIR)$(EXE):$(DIR)$(OBJS) $(DIR)$(LIBFUNCSO)
```

```
gcc -o $@ $< -L./ -lfunc
```

```
$(DIR)$(LIBFUNCSO):$(DIR)func.o
```

```
gcc -shared -o $@ $^
```

```
$(DIR)main.o:main.c
```

```
gcc -o $@ -c $^
```

```
$(DIR)func.o:func.c
```

```
gcc $(CFLAGS) -c $^ -o $@
```

```
.PHONY:rebuild clean
```

```
rebuild:clean $(DIR)$(EXE)
```

```
clean:
```

```
rm -rf $(DIR)*.o $(DIR)*.exe $(DIR)*.so
```

注意：当 OBJS 里面有多项的时候，此时\$(DIR)\$(OBJS)只能影响到 OBJS 中第一个，后面的全部无效，因此需要全部列出来。

函数：

1. wildcard 搜索当前目录下的文件名，展开成一系列所有符合由其参数描述的文件名，文件间以空格间隔。

SOURCES = \$(wildcard *.cpp)把当前目录下所有'.cpp'文件存入变量 SOURCES 里。

2. 字符串替换函数：\$(patsubst 要查找的子串,替换后的目标子串,源字符串)。将源字符串(以空格分隔)中的所有要查找的子串替换成目标子串。如 OBJS = \$(patsubst %.cpp,%.o,\$(SOURCES))

把 SOURCES 中'.cpp' 替换为'.o'，然后把替换后的字符串存入变量 OBJS。

3. \$(addprefix 前缀,源字符串)函数把第二个参数列表的每一项前缀上第一个参数值。

下面是一个较为通用的 makefile:

```
DIR := ./debug
```

```
EXE := $(DIR)/Main.exe
```

```
CC := g++
```

```
LIBS :=
```

```
SRCS := $(wildcard *.cpp) $(wildcard *.c) $(wildcard *.cc)
```

```
OCPP := $(patsubst %.cpp,$(DIR)/%.o,$(wildcard *.cpp))
```

```
OC := $(patsubst %.c,$(DIR)/%.co,$(wildcard *.c))
```

深圳信盈达科技有限公司 专业提供单片机、嵌入式、ARM、LINUX、Android、PCB、FPGA 等技术培训、技术方案。

```
OCC := $(patsubst %.cc, $(DIR)/%.cco, $(wildcard *.cc))
OBJS := $(OC) $(OCC) $(OCPP)
RM := rm -rf
CXXFLAGS := -Wall -g
start : mkdebug $(EXE)
mkdebug :
    @if [ ! -d $(DIR) ]; then mkdir $(DIR); fi;
$(EXE) : $(OBJS)
    $(CC) -o $@ $(OBJS) $(addprefix -l,$(LIBS))
$(DIR)/%.o : %.cpp
    $(CC) -c $(CXXFLAGS) $< -o $@
$(DIR)/%.co : %.c
    $(CC) -c $(CXXFLAGS) $< -o $@
$(DIR)/%.cco : %.cc
    $(CC) -c $(CXXFLAGS) $< -o $@
.PHONY : clean rebuild
clean :
    @$(RM) $(DIR)/*.exe $(DIR)/*.o $(DIR)/*.co $(DIR)/*.cco
rebuild: clean start
```

make 的命令行选项如图 2.6

命令格式	含 义
-C dir	读入指定目录下的 makefile
-f file	读入当前目录下的 file 文件作为 makefile
-i	忽略所有的命令执行错误
-I dir	指定被包含的 makefile 所在目录
-n	只打印要执行的命令，但不执行这些命令
-p	显示 make 变量数据库和隐含规则
-s	在执行命令时不显示命令
-w	如果 make 在执行过程中改变目录，则打印当前目录名

图 2. 6

2. 3 gdb 调试器

2.3.1 gdb 常用命令

Linux 包含了一个叫 gdb 的调试程序。gdb 可以用来调试 C 和 C++ 程序。
在程序编译时用 -g 选项可打开调试选项。
常见的调试程序的步骤如下：

```
gcc -o filename.o -Wall filename.c -g //进入调试用 gcc -o fn.o fn.c -g
gdb filename.o//进入调试
```

```
l //显示代码 (list)
b 4 //在第四行设置断点 相当于 Windows 的 F9 (break)
r //运行 相当于 Windows 的 F5 (run)
```

n //下一步不进入函数 相当于 Windows 的 F10 (next)
s //表示单步进入函数, 相当于 Windows 的 F11 (step)
p I //打印变量 I 相当于 Windows 的 Watch 窗口 (print)
c //运行到最后 (continue)
q //退出 相当于 Windows 的 Shift+F5 (quit)

gdb 的常用命令如图 2.7, 图 2.8, 图 2.9, 图 2.10

命令格式	含 义
set args 运行时的参数	指定运行时参数, 如 set args 2
show args	查看设置好的运行参数
path dir	设定程序的运行路径
show paths	查看程序的运行路径
set environment var [=value]	设置环境变量
show environment [var]	查看环境变量
cd dir	进入到 dir 目录, 相当于 shell 中的 cd 命令
pwd	显示当前工作目录
shell command	运行 shell 的 command 命令

图 2.7

info b	查看所设断点
break [文件名:]行号或函数名 <条件表达式>	设置断点
tbreak [文件名:]行号或函数名 <条件表达式>	设置临时断点, 到达后被自动删除
delete [断点号]	删除指定断点, 其断点号为 “info b” 中的第一栏。若缺省断点号则删除所有断点
disable [断点号]	停止指定断点, 使用 “info b” 仍能查看此断点。同 delete 一样, 省断点号则停止所有断点
enable [断点号]	激活指定断点, 即激活被 disable 停止的断点
condition [断点号] <条件表达式>	修改对应断点的条件
ignore [断点号] <num>	在程序执行中, 忽略对应断点 num 次
Step	单步恢复程序运行, 且进入函数调用
Next	单步恢复程序运行, 但不进入函数调用
Finish	运行程序, 直到当前函数完成返回
c	继续执行函数, 直到函数结束或遇到新的断点

图 2.8

命令格式	含 义
list <行号> <函数名>	查看指定位置代码
file [文件名]	加载指定文件
forward-search 正则表达式	源代码前向搜索
reverse-search 正则表达式	源代码后向搜索
dir dir	停止路径名
show directories	显示定义的源文件搜索路径
info line	显示加载到 gdb 内存中的代码

图 2.9

<code>print</code> 表达式 变量↵	查看程序运行时对应表达式和变量的值↵
<code>x <n/f/u>↵</code>	查看内存变量内容。其中 <code>n</code> 为整数表示显示内存的长度， <code>f</code> 表示显示的格式， <code>u</code> 表示从当前地址往后请求显示的字节数↵
<code>display</code> 表达式↵	设定在单步运行或其他情况中，自动显示的对应表达式的内容↵
<code>backtrace</code> ↵	查看当前栈的情况，即可以查到调用哪些函数尚未返回。↵

图 2.10

按 Tab 键补齐命令，用光标键上下翻动历史命令，用 `help up` 看帮助

2.3.2 gdb 应用举例

下面列出了将被调试的程序它显示一个简单的问候，再用反序将它列出

main.cpp:

```
void MyPrint(const char *pszSrc);
void MyPrint2(const char *pszSrc);
int main ()
{
    char szSrc[] = "hello there";
    MyPrint(szSrc);
    MyPrint2(szSrc);
}
```

func.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
void MyPrint(const char *pszSrc){
    printf("The string is %s\n", pszSrc);
}
```

```
void MyPrint2(const char *pszSrc)
{
    char *pszRev;
    int i,iLen;
    iLen=strlen(pszSrc);
    pszRev=(char *)malloc(iLen+1);
    for(i=0;i<iLen;i++)
        pszRev[i]=pszSrc[iLen-i]; //经过调试，发现此处应为: [iLen-i-1]
    pszRev[iLen]='\0';
    printf("The revert string is:%s\n",pszRev);
    free(pszRev);
}
```

用下面的命令编译它(注意加上-g 的编译选项):

```
g++ -g *.cpp //或者为 g++ -o main.o main.cpp func.cpp -g
gdb main.o //如果不用 gdb main.o 则可以利用 gdb 进入调试之后再输入(gdb) file main.o
(gdb) l[ist] func.cpp:1 //列出源代码: 技巧: 在 gdb 提示符下按回车键将重复上一个命令
(gdb) break 17 //b
(gdb) run //r
(gdb) watch pszRev[iLen-i] //w
```

```
(gdb) next //n
(gdb) info b //查看所有断点信息
(gdb) continue //继续全速运行
(gdb) p i
(gdb) p iLen
```

下面介绍一下以后可能用到的 GDB 多线程调试的基本方法。

info threads 显示当前可调试的所有线程，每个线程会有一个 GDB 为其分配的 ID，后面操作线程的时候会用到这个 ID。前面有*的是当前调试的线程。

thread ID 切换当前调试的线程为指定 ID 的线程。

break thread_test.c:123 thread all 在所有线程中相应的行上设置断点

thread apply ID1 ID2 command 让一个或者多个线程执行 GDB 命令 command。

thread apply all command 让所有被调试线程执行 GDB 命令 command。

set scheduler-locking off|on|step 估计是实际使用过多线程调试的人都可以发现，在使用 step 或者 continue 命令调试当前被调试线程的时候，其他线程也是同时执行的，怎么只让被调试程序执行呢？通过这个命令就可以实现这个需求。

off 不锁定任何线程，也就是所有线程都执行，这是默认值。

on 只有当前被调试程序会执行。

step 在单步的时候，除了 next 过一个函数的情况(熟悉情况的人可能知道，这其实是一个设置断点然后 continue 的行为)以外，只有当前线程会执行。

在介绍完基本的多线程调试命令后，大概介绍一下 GDB 多线程调试的实现思路。

小知识：

1. 在 linux 中利用 system("clear");实现类似于 windows 里面的清屏函数 system("cls");

2. LINUX 中可以通过下面的方式可以实现 system("pause");功能：

```
printf("Press any key to continue...");
```

```
getchar();
```

```
getchar(); //要用两个 getchar()函数
```

3. linux 中如何刷新输入缓冲区，利用 getchar()函数即可。输出缓冲区可以利用 fflush(stdout);

第3章 Linux 文件目录操作

3.1 基于文件指针的文件操作(缓冲)

对文件进行的操作有打开文件，关闭文件，读写文件。其中打开文件是第一步，可以说是为其它操作做准备的。

文件指针：每打开一个文件，就返回一个指针(FILE*类型)，称为文件指针。这个指针指向了这个文件相关的所有信息，即我们就可以用这个指针代表这个文件，通过这个指针可以对这个打开的文件进行各种操作。

缓冲区：输入输出的数据并不是一下子直接到电脑内存和显示器中，输入的数据先暂时存放在键盘缓冲区中，然后程序从该缓冲区中读取数据。输出的数据先暂时存放在输出缓冲区中，然后再把该数据输出到屏幕中。本节介绍的输入输出相关函数都是要用到缓冲区的。

linux 中一切皆文件。对目录和设备的操作都是文件操作。文件分为普通文件，管道文件，目录文件，链接文件和设备文件。

普通文件：也称磁盘文件，并且能够进行随机的数据存储(能够自由 seek 定位到某一个位置)；

管道文件：是一个从一端发送数据，另一端接收数据的数据通道；

目录文件：它包含了保存在目录中文件列表的简单文件。

设备文件：Linux 下各种硬件设备都是文件，该类型的文件提供了大多数物理设备的接口。它又分为两种类型：字符型设备和块设备。字符型设备一次只能读出和写入一个字节的数据，包括调制解调器、终端、打印机、声卡以及鼠标；块设备必须以一定大小的块来读出或者写入数据，块设备包括 CD-ROM、RAM 驱动器和磁盘驱动器等，一般而言，字符设备用于传输数据，块设备用于存储数据。

套接字文件：在 Linux 中,套接字也可以当作文件来进行处理。

3.1.1 文件的创建，打开与关闭

原型为：

```
#include <stdio.h> //头文件包含
FILE *fopen(const char *path,const char *mode); //文件名 模式
FILE *fdopen(int filds,const char *mode);
FILE *freopen(const char *path,const char *mode, FILE *stream);
int fclose(FILE *stream);
```

fopen 以 mode 的方式打开或创建文件，如果成功，将返回一个文件指针，失败则返回 NULL.

fopen 创建的文件的访问权限将以 0666 与当前的 umask 结合来确定。

mode 的可选模式列如表 3.1 所示。

表 3.1

模式	读	写	位置	截断原内容	创建
rb	Y	N	文件头	N	N
r+b	Y	Y	文件头	N	N
wb	N	Y	文件头	Y	Y
w+b	Y	Y	文件头	Y	Y
ab	N	Y	文件尾	N	Y
a+b	Y	Y	文件尾	N	Y

fdopen 函数用来打开某些不能直接用 fopen 方式打开的文件，比如管道和网络套接字文件。

freopen 打开一个特定的文件（由 path 指定），并将打开后的文件与指定的流（由 fp 指定）关联起来。一般用来将一个指定的文件打开为一个指定的流：标准输入、标准输出、标准出错。

在 Linux 系统中,mode 里面的'b'(二进制)可以去掉，但是为了保持与其他系统的兼容性，建议不要去掉。ab 和 a+b 为追加模式，在此两种模式下，无论文件读写点定位到何处，在写数据时都将是文件末尾添加，所以比较适合于多进程写同一个文件的情况下保证数据的完整性。

3.1.2 读写文件

数据块读写:

基于文件指针的读写函数较多, 可分为数据块读写、格式化读写、单个字符读写、字符串读写。

先来看数据块读写:

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

fread 从文件流 stream 中读取 nmemb 个元素, 写到 ptr 指向的内存中, 每个元素的大小为 size 个字节。

fwrite 从 ptr 指向的内存中读取 nmemb 个元素, 写到文件流 stream 中, 每个元素 size 个字节。

所有的文件读写函数都从文件的当前读写点开始读写, 读写完以后, 当前读写点自动往后移动 size*nmemb 个字节。

示例:

```
#include <stdio.h>

int main()
{
    char buf[50] = {'h','e','l','l','o'};
    FILE *p;//定义一个 FILE 结构体类型的指针 p
    p = fopen("a.txt", "r+b");//p 这个指针此时就和文件 a.txt 关联起来了。
    fwrite(buf, 1, 5, p);//把 buf 里面的内容写到 p 指向的文件中。
    char buf1[50] = {0};
    fread(buf1, 1, 5, p);
    printf("buf1:%s\n", buf1);
    fclose(p);//关闭 p 代表的文件 a.txt
    return 0;
}
```

格式化读写:

函数原型为:

```
#include <stdio.h>

int printf(const char *format, ...); //相当于 fprintf(stdout,format,...);
int scanf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...); // eg: sprintf(buf,"the string is:%s",str);
int sscanf(char *str, const char *format, ...);
```

以 f 开头的将格式化后的字符串写入到文件流 stream 中。

以 s 开头的将格式化后的字符串写入到字符串 str 中。

注意:

读写 : 从内存的角度而言的。

输入输出: 从人的角度而言。

读: 也就是输入, 表示数据从其它的地方传输到内存

写: 也就是输出, 表示数据从内存传输到其它的地方

这里的“其它的地方”指: 硬盘(文件) 键盘 显示器, 以及各种外设等。

示例:

```
#include <stdio.h>

int main()
{
    char buf[50] = {0};
```

```
FILE * fp = fopen("a.txt", "r+");
fscanf(fp, "%s", buf);
printf("buf = %s\n", buf);
char buf1[50] = "world";
fprintf(fp, "abc: %s\n", buf1);
return 0;
}
```

单个字符读写:

使用下列函数可以一次读写一个字符。

原型为

```
#include <stdio.h>
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
int getc(FILE *stream);           等同于 fgetc(FILE* stream)
int putc(int c, FILE *stream);    等同于 fputc(int c, FILE* stream)
int getchar(void);               等同于 fgetc(stdin);
int putchar(int c);              等同于 fputc(int c, stdout);
```

fgetc 从文件中读取一个字符，成功后返回这个字符，读完了返回 EOF。失败了返回 -1

getchar 和 putchar 从标准输入输出流中读写数据，其他函数从文件流 stream 中读写数据。

示例:

```
#include <stdio.h>
int main()
{
    char ch;
    FILE *fp = fopen("a.txt", "r+");
    ch = fgetc(fp);
    printf("ch = %c\n", ch);
    fputc('B', fp);
    fclose(fp);
    return 0;
}
```

字符串读写:

```
char *fgets(char *s, int size, FILE *stream);
int fputs(const char *s, FILE *stream);
int puts(const char *s);          等同于 fputs(const char *s, int size, stdout);
char *gets(char *s);             等同于 fgets(const char *s, int size, stdin);
```

fgets 和 fputs 从文件流 stream 中读写一行数据;

puts 和 gets 从标准输入输出流中读写一行数据。

fgets 可以指定目标缓冲区的大小，所以相对于 gets 安全，但是 fgets 调用时，如果文件中当前行的字符个数大于 size，则下一次 fgets 调用时，将继续读取该行剩下的字符，fgets 读取一行字符时，保留行尾的换行符。

fputs 不会在行尾自动添加换行符，但是 puts 会在标准输出流中自动添加一换行符。

示例:

```
#include <stdio.h>
int main()
{
    char buf[50] = {0};
    FILE *fp = fopen("a.txt", "r+");
```

```
fgets(buf, 5, fp);
printf("buf = %s\n", buf);
fputs("1234567", fp);
fclose(fp);
return 0;
}
```

3.1.3 文件定位

文件定位指读取或设置文件当前读写点，所有的通过文件指针读写数据的函数，都是从文件的当前读写点读写数据的。

常用的函数有：

```
#include <stdio.h>
int feof(FILE * stream);    //通常的用法为 while(!feof(fp))
int fseek(FILE *stream, long offset, int whence); //设置当前读写点到偏移 whence 长度为 offset 处
long ftell(FILE *stream);    //用来获得文件流当前的读写位置
void rewind(FILE *stream);    //把文件流的读写位置移至文件开头    fseek(fp, 0, SEEK_SET);
```

feof 判断是否到达文件末尾的下一个（注意到达文件末尾之后还会做一次）

fseek 设置当前读写点到偏移 whence 长度为 offset 处，whence 可以是：

```
SEEK_SET (文件开头    0)
SEEK_CUR (文件当前位置    1)
SEEK_END (文件末尾    2)
```

ftell 获取当前的读写点

rewind 将文件当前读写点移动到文件头

fseek 使用示例：

```
#include <stdio.h>
int main()
{
    char buf[50] = {'h','e','l','l','o'};
    FILE *p;
    p = fopen("a.txt", "r+b");
    fwrite(buf, 1, 5, p);
    char buf1[50] = {0};
    fseek(p, 0, SEEK_SET);
    fread(buf1, 1, 5, p);
    printf("buf1:%s\n", buf1);
    fclose(p);
    return 0;
}
```

3.1.4 标准输入/输出流

系统为每个进程预先打开了三个特殊的文件，对应的三个文件指针分别为：stdin(标准输入)、stdout(标准输出)、stderr(标准出错)。定义在头文件<stdio.h>中。

在进程一开始运行，就自动打开了三个对应设备的文件，它们是标准输入、输出、错误流，分别用全局文件指针 stdin、stdout、stderr 表示，它们都是 FILE *类型。stdin 具有可读属性，缺省情况下是指从键盘的读取输入，stdout 和 stderr 具有可写属性，缺省情况下是指向屏幕输出数据。

示例：

```
#include <stdio.h>
int main()
{
    char szBuf[32];
    printf("Input string:");
    fread(szBuf, 1, 5, stdin);
    fwrite(szBuf, 1, 5, stdout);
    fwrite(szBuf, 1, 5, stderr);
    return 0;
}
```

3.1.5 目录操作

改变目录或文件的访问权限

```
#include <sys/stat.h>
int chmod(const char* path, mode_t mode);    //mode 形如: 0777
path 参数指定的文件被修改为具有 mode 参数给出的访问权限。
```

获取、改变当前目录:

原型为:

```
#include <unistd.h>    //头文件
char *getcwd(char *buf, size_t size); //获取当前目录, 相当于 pwd 命令
int chdir(const char *path);          //修改当前目录, 即切换目录, 相当于 cd 命令
```

其中 `getcwd()` 函数: 将当前的工作目录绝对路径复制到参数 `buf` 所指的内存空间, 参数 `size` 为 `buf` 的空间大小. 在调用此函数时, `buf` 所指的内存空间要足够大, 若工作目录绝对路径的字符串长度超过参数 `size` 大小, 则回值 `NULL`, `errno` 的值则为 `ERANGE`. 倘若参数 `buf` 为 `NULL`, `getcwd()` 会依参数 `size` 的大小自动配置内存(使用 `malloc()`), 如果参数 `size` 也为 0, 则 `getcwd()` 会依工作目录绝对路径的字符串长度来决定所配置的内存大小, 进程可以在使用完此字符串后自动利用 `free()` 来释放此空间. 所以常用的形式: `getcwd(NULL, 0)`;

`chdir()` 函数: 用来将当前的工作目录改变成以参数 `path` 所指的目录。

示例:

```
#include<unistd.h>
main()
{
    chdir("/tmp");
    printf("current working directory: %s\n",getcwd(NULL,0));
}
```

创建和删除目录:

原型为:

```
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);    //创建目录,mode 是目录权限,没用
int rmdir(const char *pathname);                //删除目录
```

获取目录信息:

原型为:

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

```
DIR *opendir(const char *name);           //打开一个目录
struct dirent *readdir(DIR *dir); //读取目录的一项信息，并返回该项信息的结构体指针
void rewinddir(DIR *dir);                //重新定位到目录文件的头部
void seekdir(DIR *dir, off_t offset); //用来设置目录流目前的读取位置
off_t telldir(DIR *dir);                 //返回目录流当前的读取位置
int closedir(DIR *dir);                  //关闭目录文件
```

读取目录信息的步骤为：

- 用 `opendir` 函数打开目录；
- 使用 `readdir` 函数迭代读取目录的内容，如果已经读取到目录末尾，又想重新开始读，则可以使用 `rewinddir` 函数将文件指针重新定位到目录文件的起始位置；
- 用 `closedir` 函数关闭目录

`opendir()` 用来打开参数 `name` 指定的目录，并返回 `DIR*` 形态的目录流，和文件操作函数 `open()` 类似，接下来对目录的读取和搜索都要使用此返回值。函数失败则返回 `NULL`；

`readdir()` 函数用来读取目录的信息，并返回一个结构体指针，该指针保存了目录的相关信息。有错误发生或者读取到目录文件尾则返回 `NULL`；`dirent` 结构体如下：

```
struct dirent
{
    ino_t d_ino;           /* inode number (此目录进入点的 inode) */
    off_t d_off;           /* offset to the next dirent (目录开头到进入点的位移) */
    unsigned short d_reclen; /* length of this record (目录名的长度) */
    char d_name[256];      /* filename (文件名) */
};
```

`seekdir()` 函数用来设置目录流目前的读取位置，再调用 `readdir()` 函数时，便可以从此新位置开始读取。参数 `offset` 代表距离目录文件开头的偏移量。

`telldir()` 函数用来返回目录流当前的读取位置。

示例：

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

int main(int argc, char * argv[])
{
    struct dirent * pDirInfo;
    DIR * pDir;
    if (argc < 2)
        pDir = opendir(".");
    else
        pDir = opendir(argv[1]);
    if (NULL == pDir)
    {
        perror("open dir fail!");
        return -1;
    }
    while ((pDirInfo = readdir(pDir)) != NULL)
        printf("%s\n", pDirInfo->d_name);
    closedir(pDir);
    return 0;
}
```

示例：以树形结构的形式输出指定目录下面的所有文件

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys / stat.h>
#include <stdlib.h>
void printdir(char * dir, int depth)
{
    DIR * dp = opendir(dir);
    if (NULL == dp)
    {
        fprintf(stderr, "cannot open directory: %s\n", dir);
        return;
    }
    chdir(dir);
    struct dirent * entry;
    struct stat statbuf;
    while ((entry = readdir(dp)) != NULL)
    {
        stat(entry->d_name, &statbuf);
        if (S_ISDIR(statbuf.st_mode))
        {
            if (strcmp(".", entry->d_name) == 0 || strcmp("..", entry->d_name) == 0)
                continue;
            printf("%*s%s\n", depth, "", entry->d_name);
            printdir(entry->d_name, depth + 4);
        }
        else
            printf("%*s%s\n", depth, "", entry->d_name);
        //printf("%*s",4,"**"); 该函数表示输出"__*",前面输出 3 个空格。
        //如果是 printf("%*s",4,"**");则表示输出"__**", 前面输出 2 个空格。
    }
    chdir("..");
    closedir(dp);
}

int main(int argc, char * argv[])
{
    char * topdir, pwd[2] = ".";
    if (argc < 2)
        topdir = pwd;
    else
        topdir = argv[1];
    printf("Directory scan of %s\n", topdir);
    printdir(topdir, 0);
    printf("done.\n");
    exit(0);
}
```


3.2 基于文件描述符的文件操作(非缓冲)

3.2.1 文件描述符

一个程序(进程)可以在运行的过程中同时打开多个文件，每个程序运行起来后，系统中就有一个记录表专门记录这个程序打开的各个文件。每打开一个文件，记录表就会用一个新的结构体变量来保存这个文件的相关信息。如果打开多个文件，记录表中就会有多个这样的结构体变量分别保存多个文件的相关信息，它们构成了一个结构体数组，而数组的每一个元素的下标就是所谓的文件描述符。

所以文件描述符是一个较小的非负整数（0—1023），它代表记录表的一项，即上面说的数组中的某个元素的下标。通过文件描述符和一组基于文件描述符的文件操作函数，就可以实现对文件的打开、关闭、读写、删除等操作。常用基于文读、写、创建、文件描述符的函数有 open（打开）、creat（创建）、close（关闭）、read（读取）、write（写入）、ftruncate（改变文件大小）、lseek（定位）、fsync（同步）、fstat（获取文件状态）、fchmod（权限）、flock（加锁）、fcntl（控制文件属性）、dup（复制）、dup2、select 和 ioctl。基于文件描述符的文件操作并非 ANSI C（标准 C）的函数，而是 Linux 的系统调用，即 Linux 下的 API 函数。

如果不清楚某个函数的具体实现形式，可以通过下面的方式查询
man 函数名，即可查看该函数的帮助信息。
如果要复制里面的内容，按 Ctrl+Insert 键，再粘贴的话用：Shift+Insert 键。

3.2.2 打开、创建和关闭文件

```
open 和 creat 都能打开和创建文件，原型为
#include <sys/types.h>    //头文件
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);      //文件名 打开方式
int open(const char *pathname, int flags, mode_t mode); //文件名 打开方式 权限
int creat(const char *pathname, mode_t mode); //文件名 权限 //现在已经不常用了
creat 函数等价于→open(pathname,O_CREAT|O_TRUNC|O_WRONLY,mode);
int close(int fd); //fd 表示文件描述符
```

open 和 creat 函数出错时返回-1，成功则返回一个整数，这个整数就是和这个打开的文件相对应的文件描述符。通过这个返回的文件描述符我们可以对这个文件进行读写等各种操作。

相关参数如下：
flags 和 mode 都是一组掩码的合成值，flags 表示打开或创建的方式，mode 表示文件的访问权限。

对于 open 函数来说，第三个参数 mode 仅当创建新文件时（即使用了 O_CREAT 时）才使用，即只有在建立新文件时才会生效，用于指定文件的访问权限位。如果打开一个已有的文件，但是又使用了第三个参数指定了权限，那第三个参数将不会生效，等于没有。

flags 的可选项如表 3.2 所示

表 3.2

掩码	含义
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_CREAT	如果文件不存在，则创建文件
O_EXCL	仅与 O_CREAT 连用，如果文件已存在，则强制 open 失败
O_TRUNC	如果文件存在，将文件的长度截至 0
O_APPEND	以追加的方式打开文件，每次调用 write 时，文件指针自动先移到文件尾，用于多进程写同一个文件的情况。

O_NONBLOCK	非阻塞方式打开，无论有无数据读取或等待，都会立即返回进程之中。
O_NODELAY	非阻塞方式打开
O_SYNC	同步打开文件，只有在数据被真正写入物理设备设备后才返回

mode 的可选项有：

S_IRWLRW 00700 权限，代表该文件所有者具有可读、可写及可执行的权限。

S_IRUSR 或 S_IREAD, 00400 权限，代表该文件所有者具有可读取的权限。

S_IWUSR 或 S_IWRITE, 00200 权限，代表该文件所有者具有可写入的权限。

S_IXUSR 或 S_IEXEC, 00100 权限，代表该文件所有者具有可执行的权限。

S_IRWXG 00070 权限，代表该文件用户组具有可读、可写及可执行的权限。

S_IRGRP 00040 权限，代表该文件用户组具有可读的权限。

S_IWGRP 00020 权限，代表该文件用户组具有可写入的权限。

S_IXGRP 00010 权限，代表该文件用户组具有可执行的权限。

S_IRWXO 00007 权限，代表其他用户具有可读、可写及可执行的权限。

S_IROTH 00004 权限，代表其他用户具有可读的权限

S_IWOTH 00002 权限，代表其他用户具有可写入的权限。

S_IXOTH 00001 权限，代表其他用户具有可执行的权限。

但是通常采用直接赋数值的形式,如:

```
int fd = open("1.txt", O_WRONLY | O_CREAT, 0755); //表示给 755 的权限
if (-1 == fd)
{
    perror("open failed!\n");
    exit(-1);
}
```

注意：Linux 中基于文件描述符的 open 函数，对于一个不存在的文件，不能通过 O_WRONLY 的方式打开，必须加上 O_CREAT 选项。

close 用于文件的关闭：

int close(int fd); //fd 表示文件描述词,是先前由 open 或 creat 创建文件时的返回值。

文件使用完毕后，应该调用 close 关闭它。

3.2.3 读写文件

读写文件的函数原型为：

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count); //文件描述词 缓冲区 长度
ssize_t write(int fd, const void *buf, size_t count);
```

对于 read 和 write 函数，出错返回-1，读取完了之后，返回 0，其他情况返回读写的个数。

示例 1：打开 a.txt 文件，读出 6 个字节到 buf 中，以字符串的形式输出到屏幕，再向 a.txt 写入"hello"，最后关闭 a.txt 文件。

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
```

```
int fd;
char buf[32] = {0};
fd = open("a.txt", O_RDWR);
read(fd, buf, 6);
printf("%s\n", buf);
strcpy(buf, "hello");
write(fd, buf, 6);
close(fd);
return 0;
}
```

示例 2: 将 aa.txt 中的内容复制到 bb.txt 中, 其中 bb.txt 起初不存在。

```
#include <stdio.h>
#include <stdlib.h> //包含 exit
#include <sys / types.h>
#include <sys / stat.h>
#include <fcntl.h>
#include <errno.h> //用 perror 输出错误
#include <unistd.h>
#define FILENAME1 "./aa.txt" //用宏定义文件的路径, 可以实现一改都改
#define FILENAME2 "./bb.txt"
main()
{
    char buf[512] = {0};
    int fo1 = open(FILENAME1, O_RDONLY); //fo1, fo2 都是文件描述词
    int fo2 = creat(FILENAME2, 0755); //创建文件
    /*int fo2 = open(FILENAME2, O_WRONLY | O_CREAT);*/
    if ((-1 == fo1) || (-1 == fo2))
    {
        perror("open failed!\n");
        /*用于输出错误信息.类似于: fputs("open failed\n", stderr)*/;
        exit(-1);
    }
    int fr = 0;
    while ((fr = read(fo1, buf, sizeof(buf))) > 0)
        /*如果 read 读取成功, 返回的是长度, 否则, 返回-1*/
    {
        write(fo2, buf, fr);
    }
    close(fo1);
    close(fo2);
}
```

本示例: 稍加改进, 比如加上命令行参数, 通过命令行参数打开相应的文件即可复制任何文件。

3.2.4 改变文件大小

函数原型:

```
#include <unistd.h>
int ftruncate(int fd, off_t length);
```

函数 ftruncate 会将参数 fd 指定的文件大小改为参数 length 指定的大小。参数 fd 为已打开的文件描述词, 而

且必须是以写入模式打开的文件。如果原来的文件大小比参数 `length` 大，则超过的部分会被删去。

返回值 执行成功则返回 0，失败返回-1。

实例：

```
int main()
{
    int fd = open("a.txt", O_WRONLY);
    ftruncate(fd, 1000);
    close(fd);
    return 0;
}
```

3.2.5 文件定位

函数 `lseek` 将文件指针设定到相对于 `whence`，偏移值为 `offset` 的位置

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence); //fd 文件描述词
```

`whence` 可以是下面三个常量的一个

`SEEK_SET` 从文件头开始计算

`SEEK_CUR` 从当前指针开始计算

`SEEK_END` 从文件尾开始计算

利用该函数可以实现文件空洞（对一个新建的空文件，可以定位到偏移文件开头 1024 个字节的地方，在写入一个字符，则相当于给该文件分配了 1025 个字节的空间，形成文件空洞）通常用于多进程间通信的时候的共享内存。

```
int main()
{
    int fd = open("c.txt", O_WRONLY | O_CREAT);
    lseek(fd, 1024, SEEK_SET);
    write(fd, "a", 1);
    close(fd);
    return 0;
}
```

3.2.6 原子操作

如果对文件的读写操作很重要，不能被打断，可以用原子操作函数。

对文件的读写原子操作有 `pread`、`pwrite`。

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset); //文件描述符,缓冲区长度,相对于文件头的偏移量
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

3.2.7 进一步理解文件描述符

如果一个程序打开一个文件，那么 `open` 函数就会返回一个文件描述符。前面已经说了，文件描述符是一个较小的非负整数（0-1023）。既然如此，那么这个文件描述符的值是多少呢？它的值是由系统分配的。而实际上，当一个程序打开第一个文件时，分配的文件描述符的值是 3，第二个是 4……

就这样依次往后排。那么既然知道了它的值，我们可不可以在打开两个文件后，在其后的 `read`、`write` 等函数的相应位置上，直接填上 3 或 4 呢来进行操作呢？请运行下面的程序。

示例：

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

```
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
    int fda, fdb;
    char buf[32] = {0};
    fda = open("a.txt", O_RDWR);
    fdb = open("b.txt", O_RDWR);
    printf("fda = %d fdb = %d\n", fda, fdb);
    read(3, buf, 6);
    printf("%s\n", buf);
    read(4, buf, 6);
    printf("%s\n", buf);
    close(3);
    close(4);
    return 0;
}
```

运行发现，这样做是可以的。因为我们事先已经猜出了这两个文件描述符的值，即 3 和 4，所以我们在后面用到它们的地方直接写上 3 和 4，也是可以的。这说明文件描述符就是一个普通的非负整数，不必把它想得太过复杂和神秘。

3.2.8 文件描述符的复制

函数 dup 和 dup2 可以实现文件描述符的复制，此时文件描述符是没有打开的最小文件描述符。原型为：

```
#include <unistd.h> //头文件包含
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

文件描述符的复制是指用另外一个文件描述符指向同一个打开的文件，它完全不同于直接给文件描述符变量赋值，例如：

描述符变量的直接赋值：

```
char buf[32];
int fd1=open("./a.txt",O_RDONLY);
int fd2=fd1; //类似于 C 语言的指针赋值，当释放掉一个得时候，另一个已经不能操作了
close(fd1); //导致文件立即关闭
printf("read:%d\n",read(fd2),buf,sizeof(buf)-1)); //读取失败
close(fd2); //无意义
```

在此情况下，两个文件描述符变量的值相同，指向同一个打开的文件，但是内核的文件打开引用计数还是为 1，所以 close(fd)或者 close(fd2)都会导致文件立即关闭掉。

描述符的复制：

```
char buf[32];
int fd1=open("./a.txt",O_RDONLY);
int fd2=dup(fd1); //内核的文件打开引用计算+1，变成 2 了
close(fd1); //当前还不会导致文件被关闭，此时通过 fd2 照样可以访问文件
printf("read:%d\n",read(fd2),buf,sizeof(buf)-1);
close(fd2); //内核的引用计数变为 0，文件正式关闭
```

此时 fd2 如果修改了文件内容，则文件内容将会改变。即 fd1 和 fd2 中有一个修改文件之后对应的另一个也

变了。

示例 1:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    char buf[32] = {0};
    int fd1 = open("./a.txt", O_RDONLY);
    int fd2 = dup(fd1);
    //int fd2 = dup2(fd1, 100); //换成这个则把文件描述符人为指定为 100
    close(fd1);
    printf("fd1 = %d, fd2 = %d\n", fd1, fd2);
    printf("read:%d\n", read(fd2, buf, sizeof(buf)-1));
    puts(buf);
    close(fd2);           //无意义
}
```

解析：假设 a.txt 中的内容为：abcdef。上面的例子会发现第一次输出的结果是 abcdef。关闭 close(fda) 的时候，文件实际上还没有真正的关闭，此时文件指针已经向后移动了。执行 write 命令将 hellojava 又追加到 a.txt 的后面，最后关闭 fd2 的时候，输出：abcdef hellojava。

dup2(int fdold, int fdnew) 也是进行描述符的复制，只不过采用此种复制，新的描述符由用户用参数 fdnew 显示指定，而不是象 dup 一样由内核帮你选定（内核选定的是从最小找的）。对于 dup2，如果 fdnew 已经指向一个已经打开的文件，内核会首先关闭掉 fdnew 所指向的原来的文件。此时再针对于 fdnew 文件描述符操作的文件，则采用的是 fdold 的文件描述符。如果成功 dup2 的返回值与 fdnew 相同，否则为 -1。把上例中的 dup 的那一行换成 dup2 那一行就可以看到效果了。

思考下面程序的结果：

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    char buf[32] = {0};
    int fda = open("./a.txt", O_RDONLY);
    int fdb = open("./b.txt", O_RDONLY);
    int fdbb = dup(fdb);
    int fda2 = dup2(fda, fdb); //可以设定为：int fda2 = dup2(fda, 5); 即自己设为 5
    printf("fda:%d fdb:%d fdbb:%d fda2:%d", fda, fdb, fdbb, fda2);
    read(fdb, buf, sizeof(buf)-1); //此时 fdb 已经不再定位 b.txt 而是 a.txt
    printf("result:%s\n", buf);
    close(fda);
    close(fdb);
}
```



```
close(fdbb);
close(fda2);
}
```

3.2.9 文件的锁定

在多线程对同一个文件进行读写访问时，为了保证数据的完整性，有时需要对文件进行锁定。可以通过 `fcntl` 对文件进行锁定和解锁。

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);    //文件描述词 欲操作的命令
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock * lock);
int ioctl (int d, int request, ...);
```

//`ioctl` 函数用来控制硬件，如光驱、硬盘、摄像头、lcd 等硬件，后面会详细介绍。

返回值 成功则返回 0，若有错误则返回-1

参数 `struct flock` 为结构体，表示文件锁信息，如下所示：

```
struct flock
{
    short int l_type;    /*锁定的状态*/           用 F_RDLCK 和 F_WRLCK 和 F_UNLCK
    short int l_whence; /*决定 l_start 位置*/       用 SEEK_SET, SEEK_CUR, SEEK_END
    off_t l_start;      /*锁定区域的开头位置*/
    off_t l_len;        /*锁定区域的大小*/         用 struct stat 结构体 中的 st_size 可以获得
    pid_t l_pid;        /*锁定动作的进程*/         用 getpid()函数获得
};
```

//通常将 `l_start` 和 `l_len` 都设为 0，`l_whence` 为 `SEEK_SET` 表示整个文件。

参数 `cmd`，常用的如下：

1. 加锁解锁：

置为 `F_SETLK`：设置文件锁定的状态。此时 `flock` 结构体的 `l_type` 值必须是 `F_RDLCK`、`F_WRLCK` 或 `F_UNLCK`。如果无法建立锁定，则返回-1

对于 `F_RDLCK`：共享锁（或读锁），许多不同的进程可以拥有文件同一（或重叠）区域上的共享锁。只要任一进程拥有共享锁，那么就不能再有进程可以获得该区域上的独占锁。为了获得一把共享锁，文件必须为“读”或“读/写”方式打开。

对于 `F_WRLCK`：独占锁（或写锁），只有一个进程可以在文件的任一特定区域拥有独占锁。一旦一个进程拥有了这样的锁，其他任何进程都无法在该区域上获得任何类型的锁。为了获得一把独占锁，文件必须为“写”或“读/写”方式打开。

对于 `F_UNLCK`：解锁，用来清除锁。

置为 `F_GETLK`：获取文件锁定的状态。此时同样需要对第三个参数指向的结构体填充，即设置为相应的锁，`fcntl` 会取得第一个能够阻止事先设置的锁生成的锁。取得的该锁的信息将覆盖传到 `fcntl()` 的 `flock` 结构的信息。如果没有发现能够阻止本次锁(`flock`)生成的锁,这个结构体 `l_type` 将为 `F_UNLCK`，并不是上锁文件没有被上锁，而是说别人对这个文件设置的锁不会阻碍我这个锁的获得，所以对我来说相当于是解锁的状态。

给文件加锁步骤如下：

```
int main()
{
    int fd = open("1.dat", O_RDONLY); //1. open 打开文件
    //计算文件长度，因为设置 flock 结构体时要设置长度 l_len
    struct stat st;
    fstat(fd, &st);
    int len = st.st_size;
```



```
struct flock lock; //2. 设置 flock 结构体的参数
lock.l_type = F_RDLCK; // 设置枷锁类型为读锁
lock.l_whence = SEEK_SET; // 从头往后
lock.l_start = 4; // 开始位置
lock.l_len = len; // 大小
lock.l_pid = getpid(); // 进程
fcntl(fd, F_SETLK, &lock); //3. 用 fcntl(fd,F_SETLK,&lock);加锁 设置文件锁的状态
sleep(10);
close(fd); //4. 关闭文件
return 0;
}
```

文件锁有两种类型，读取锁(共享锁)和写入锁(互斥锁)。

对于已经加读取锁的文件，再加写入锁将会失败，但是允许其它进程继续加读取锁；

对于已经加写入锁的文件，再加读取锁和写入锁都将会失败。

注意：文件锁只会对其它试图加锁的进程有效，对直接访问文件的进程无效，也就是说自己对一个文件加了锁，自己是检测不到锁的状态的。当然自己还是可以读写文件。那么是不是可以说我加了锁别的程序就不能对这个文件读写了呢？不是的，其它的程序照样可以读写这个加锁的文件。文件锁并不能阻止对文件有读写权限的任何其他进程对文件进行的读写操作。那文件锁到底有什么用呢？加上相应的锁只是告诉别人我此时在访问文件，起一个标识的作用，至于应用，一个应用程序的多个进程在都需要修改一个数据文件时，这种锁就显的重要了。多个进程修改的文件区不同时就可以并行执行，有交差时就可以互斥执行。另外，在对加了锁的文件进行读写操作时，应采用底层的 read 和 write 函数，因为高层的 fread 和 fwrite 函数会有缓冲区。这样就不能实现同步更新了。

示例如下：将其保存为 1.c，然后 cp 1.c 2.c，将 2.c 中的 F_RDLCK 修改为 F_WRLCK，然后同时运行就可以看到效果了。

示例 1:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
int main()
{
    int fd = open("b.txt", O_RDWR);

    struct flock lock, locka;
    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = 0;
    lock.l_pid = getpid();
    if(fcntl(fd, F_SETLK, &lock) == 0)
    {
        if(lock.l_type == F_RDLCK)
        {
            printf("上了读锁: %d\n", lock.l_pid);
        }
    }
}
```

```
        else if(lock.l_type == F_WRLCK)
        {
            printf("上了写锁: %d\n", lock.l_pid);
        }
    }
    locka = lock;
    locka.l_type = F_RDLCK;

    fcntl(fd, F_GETLK, &locka);
    if(locka.l_type == F_RDLCK)
    {
        printf("检测到读锁: %d\n", locka.l_pid);
    }
    if(locka.l_type == F_WRLCK)
    {
        printf("检测到写锁: %d\n", locka.l_pid);
    }
    sleep(8);
    close(fd);
    return 0;
}
```

示例 2:

```
/*fcntl.c*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char ** argv)
{
    /*判断 main 函数的参数个数*/
    if(argc != 2)
    {
        perror("main err");
        return -1;
    }

    int fd, fdl;
    fd = creat(argv[1], 0777);
    if(fd == -1)
    {
        perror("creat err");
        return -1;
    }
    printf("fd = %d\n", fd); //输出文件描述符 fd 的值

    /*将 hello 写入文件中*/
```

```
    if(write(fd, "hello", 6) == -1)
    {
        perror("write err");
        return -1;
    }

    /*调用 fcntl 函数，并输出返回值*/
    fdl = fcntl(fd, F_DUPFD, 10);
    if(fdl == -1)
    {
        perror("fcntl err");
        return -1;
    }
    printf("fdl = %d\n", fdl);

    /*向 fdl 代表的打开的文件中写入 test*/
    if(write(fdl, "test\n", 5) == -1)
    {
        perror("write err");
        return -1;
    }

    close(fdl); //关闭文件
    return 0;
}
```

改变文件状态（open 设置的状态）

置为 F_GETFL：取得文件描述词状态旗标。

置为 F_SETFL：设置文件描述词状态旗标，参数 arg 为新旗标，但只允许 O_APPEND、O_NONBLOCK 和 O_SYNC 位的改变，其他位的改变将不受影响。

示例 3：把标准输入改为非阻塞输入（即不会让程序中断等待输入，而是直接跳过）

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
main()
{
    char buf[512]={0};
    int flags = fcntl(STDIN_FILENO,F_GETFL);//1. 获取标准输入的状态旗标存于 flags 中
    printf("%d\n",flags);
    flags |= O_NONBLOCK; //2. 重新设置状态旗标。此处表示非阻塞输入
    fcntl(STDIN_FILENO,F_SETFL,flags); //3. 将新的状态旗标设置进去
    read(STDIN_FILENO,buf,512);
    puts(buf);
}
```

3.2.10 获取文件信息

可以通过 fstat 和 stat 函数获取文件信息，调用完毕后，文件信息被填充到结构体 struct stat 变量中，函数原型为：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);    //文件名  stat 结构体指针
int fstat(int fd, struct stat *buf);                //文件描述词  stat 结构体指针
int lstat(const char *file_name, struct stat *buf);
```

lstat 函数类似于 stat，但是当命名的文件是一个符号链接时，lstat 得到该符号链接文件的有关信息，而不是该符号链接指向的文件的信息。

结构体 stat 的定义为：

```
struct stat {
    dev_t      st_dev;      /*如果是设备，返回设备表述符，否则为 0*/
    ino_t      st_ino;      /* i 节点号 */
    mode_t     st_mode;     /* 文件类型 */
    nlink_t    st_nlink;    /* 链接数 */
    uid_t      st_uid;     /* 属主 ID */
    gid_t      st_gid;     /* 组 ID */
    dev_t      st_rdev;     /* 设备类型*/
    off_t      st_size;     /* 文件大小，字节表示 */
    blksize_t  st_blksize; /* 块大小*/
    blkcnt_t   st_blocks;  /* 块数 */
    time_t     st_atime;    /* 最后访问时间*/
    time_t     st_mtime;    /* 最后修改时间*/
    time_t     st_ctime;    /* 创建时间 */
};
```

对于结构体的成员 st_mode，有一组宏可以进行文件类型的判断，如表 3.3 所示

表 3.3

宏	描述
S_ISLNK(mode)	判断是否是符号链接
S_ISREG(mode)	判断是否是普通文件
S_ISDIR(mode)	判断是否是目录
S_ISCHR(mode)	判断是否是字符型设备
S_ISBLK(mode)	判断是否是块设备
S_ISFIFO(mod e)	判断是否是命名管道
S_ISSOCK(mo de)	判断是否是套接字

通常用于判断：if(S_ISDIR(st.st_mode)){ }

示例：获得文件的大小

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

```
mian()
{
    struct stat buf;
    stat ("/etc/passwd",&buf);
    printf( "/etc/passwd file size = %d \n" ,buf.st_size); //st_size 可以得到文件大小
}
```

如果用 fstat 函数实现，如下：

```
int fd = open ( "/etc/passwd" ,O_RDONLY); //先获得文件描述词
fstat(fd, &buf);
```

实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>

int main()
{
    int fd = open("./a.txt", O_RDONLY);
    if(fd == -1)
    {
        perror("open error");
        exit(-1);
    }

    struct stat buf;
    int iRet = fstat(fd, &buf);
    if(iRet == -1)
    {
        perror("fstat error");
        exit(-1);
    }

    if(S_ISREG(buf.st_mode))
    {
        printf("regular file!\n");
    }
    if(S_ISDIR(buf.st_mode))
    {
        printf("directory!\n");
    }
    if(S_ISLNK(buf.st_mode))
    {
        printf("link file!\n");
    }
}
```

```
printf("the size of file is : %d\n", buf.st_size);
close(fd);
return 0;
}
```

3.2.11 access 函数

Linux 下不同的用户对一个文件的可操作权限不同，access 函数可检测当前用户（运行这个程序的用户）对某文件是否有某权限。其是按实际用户 ID 和实际组 ID 进行权限许可测试的。

```
#include <unistd.h>

int access (const char *pathname, int mode) ;//要检测的文件名 要检测的权限
mode 可以是以下宏： R_OK 读 W_OK 写 X_OK 执行 F_OK 测试文件是否存在
和其它函数一样，有相应权限返回 0，没有就返回-1。
```

示例：

```
#include <unistd.h>
#include <stdio.h>
void quanxian(char *filename);
int main(int argc, char *argv[])
{
    quanxian(argv[1]);
    return 0;
}
void quanxian(char *filename)
{
    if(!access(filename, F_OK))
    {
        if(!access(filename, R_OK))
            printf("r");
        else
            printf("-");
        if(!access(filename, W_OK))
            printf("w");
        else
            printf("-");
        if(!access(filename, X_OK))
            printf("x");
        else
            printf("-");
    }
    else
        printf("file not exist!\n");
}
```

3.2.12 标准输入输出文件描述符

与标准的输入输出流对应，在更底层的实现是用标准输入、标准输出、标准错误文件描述符表示的。它们分别用 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO 三个宏表示，值分别是 0、1、2 三个整型数字。

标准输入文件描述符	➔ STDIN_FILENO	➔ 0
标准输出文件描述符	➔ STDOUT_FILENO	➔ 1

标准错误输出文件描述符 → STDERR_FILENO → 2

为什么前面提到的打开一个文件返回的描述符从 3 开始分配？原因就在这里。

示例：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
int main()
{
    char buf[50] = {0};
    read(0, buf, 10);
    write(1, buf, 10);
    write(2, buf, 10);
    return 0;
}
```

自己运行下面的程序，看看有什么结果，并思考为什么会这样？运行前确保文件 a.txt 和 b.txt 都存在并且文件里有一些内容。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int main()
{
    close(0);
    close(1);
    open("a.txt", O_RDWR);
    open("b.txt", O_RDWR);
    char buf[20];
    scanf("%s", buf);
    printf("hello world %s\n", buf);
    return 0;
}
```

3.2.13 时间和日期相关函数

日历时间：用“从一个标准时间点到此时的时间经过的秒数”来表示的时间。这个标准时间点对不同的编译器会有所不同。在 Linux 系统中，这个标准时间点是 1970 年 1 月 1 日 00: 00: 00。用 time_t 这种数据类型来表示从那一刻到现在所经过的秒数。

格林威治时间：即国际标准时间。

本地时间：本地时区的时间，全球分为 24 个时区，我国是东 8 区的时间。

```
#include <time.h>
time_t time(time_t *t);
struct tm *gmtime(const time_t *timep);
struct tm *localtime(const time_t *timep);
```

time_t: 保存日历时间的数据类型 tm: 保存时间的结构体

struct tm 结构如下：

```
struct tm {
    int tm_sec;
```



```
int tm_min;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};
```

示例：获取系统时间并打印

```
#include <stdio.h>
#include <time.h>
int main()
{
    /*获取日历时间， 从 1970 -1 -1 0 点到现在的秒数。*/
    time_t n = time(NULL);
    printf("n = %d\n", n);
    /*将日历时间转换为格林威治时间（世界标准时间）*/
    struct tm *p = gmtime(&n);
    printf("%4d-%d-%d %d:%d:%d\n", 1900+p->tm_year, 1+p->tm_mon, p->tm_mday, p->tm_hour,
p->tm_min, p->tm_sec);
    /*将日历时间转换为本地时间*/
    p = localtime(&n);
    printf("%4d-%d-%d %d:%d:%d\n", 1900+p->tm_year, 1+p->tm_mon, p->tm_mday, p->tm_hour,
p->tm_min, p->tm_sec);
    return 0;
}
```

运行这个程序，将分别显示日历时间，国际标准时间和本地时间。由于年是从 1900 年开始计算的，即现在的年份到 1900 年经过的年份差，所以要加上 1900，而月份的下标是从 0 开始计算的，所以加上 1。我国是东 8 区，所以本地时间的小时比国际标准时间多 8 小时。

3.2.14 处理的模型(补充)

阻塞 I/O 模型：在这种模型下，若所调用的 I/O 函数没有完成相关的功能，则会使进程挂起，直到相关数据到达才会返回。如常见对管道设备、终端设备和网络设备进行读写时经常会出现这种情况。

非阻塞模型：在这种模型下，当请求的 I/O 操作不能完成时，则不让进程睡眠，而且立即返回。非阻塞 I/O 使用户可以调用不会阻塞的 I/O 操作，如 `open()`、`write()`和 `read()`。如果该操作不能完成，则会立即返回出错（例如：打不开文件）或者返回 0（例如：在缓冲区中没有数据可以读取或者没空间可以写入数据）。

I/O 多路转接模型：在这种模型下，如果请求的 I/O 操作阻塞，且它不是真正阻塞 I/O，而是让其中的一个函数等待，在这期间，I/O 还能进行其他操作。如本节要介绍的 `select()`函数，就是属于这种模型。

信号驱动 I/O 模型：在这种模型下，通过安装一个信号处理程序，系统可以自动捕获特定信号的到来，从而启动 I/O。这是由内核通知用户何时可以启动一个 I/O 操作决定的。

异步 I/O 模型：在这种模型下，当一个描述符已准备好，可以启动 I/O 时，进程会通知内核。现在，并不是所有的系统都支持这种模型。

Select 函数：

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

int select(int maxfd, fd_set *readset, fd_set *writeset, fd_set *exceptionset, const struct timeval * timeout);
```

返回:就绪描述字的正数目，0——超时，-1——出错

参数解释:

maxfd: 最大的文件描述符 (其值应该为最大的文件描述符字 + 1)

readset: 内核读操作的描述符字集合

writeset: 内核写操作的描述符字集合

exceptionset: 内核异常操作的描述符字集合

timeout: 等待描述符就绪需要多少时间。NULL 代表永远等下去, 一个固定值代表等待固定时间, 0 代表根本不等待, 检查描述符之后立即返回。

其中 readset、writeset、exceptionset 都是 fd_set 集合。该集合的相关操作如下:

```
void FD_ZERO(fd_set *fdset); /* 将所有 fd 清零 */
```

```
void FD_SET(int fd, fd_set *fdset); /* 增加一个 fd */
```

```
void FD_CLR(int fd, fd_set *fdset); /* 删除一个 fd */
```

```
int FD_ISSET(int fd, fd_set *fdset); /* 判断一个 fd 是否有设置 */
```

一般来说, 在使用 select 函数之前, 首先要使用 FD_ZERO 和 FD_SET 来初始化文件描述符集, 在使用 select 函数时, 可循环使用 FD_ISSET 测试描述符集, 在执行完对相关文件描述符之后, 使用 FD_CLR 来清除描述符集。

另外, select 函数中的 timeout 是一个 struct timeval 类型的指针, 该结构体如下:

```
struct timeval
{
    long tv_sec; /* second */ //秒
    long tv_usec; /* microsecond */ //微秒
};
```

示例: 多路转接模型 select

```
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#define FILENAME1 "a.txt"
#define FILENAME2 "dir.c"
int main()
{
    char buf[10] = {0};

    int fd1 = open(FILENAME1, O_RDWR);
    int fd2 = open(FILENAME2, O_RDWR);
    int fd3 = open(FILENAME1, O_RDWR);
    int fd4 = open(FILENAME2, O_RDWR);

    if( (-1 == fd1) || (-1 == fd2) || (-1 == fd3) || (-1 == fd4) )
    {
        perror("open");
        exit(-1);
    }

    fd_set fdrd, fdwr; //绑定读写集合
```

```
FD_ZERO(&fdrd);           //清除以前读的绑定
FD_ZERO(&fdwr);           //清除以前写的绑定

FD_SET(fd1,&fdrd);         //将 fd1 与读绑定
FD_SET(fd2,&fdrd);
FD_SET(fd3,&fdwr);         //将 fd3 与写绑定
FD_SET(fd4,&fdwr);

int max1 = fd1 > fd2 ? fd1 : fd2;      //获取读绑定中的文件描述词最大值
int max2 = fd3 > fd4 ? fd3 : fd4;      //获取写绑定中的文件描述词最大值
int max = max1 > max2 ? max1 : max2;    //获得读写文件描述词最大值

struct timeval tv;           //用于记录时间，表示过这么长时间不响应就退出
tv.tv_sec = 2;              //秒
tv.tv_usec = 0;            //微妙

while(1)
{
    if( select(max+1, &fdrd, &fdwr, NULL, &tv) == -1 ) //从 1—max+1 查找
    {
        perror("select");
        break;
    }
    if( FD_ISSET(fd1,&fdrd) ) //如果 fd1 设置的是读绑定
    {
        read(fd1,buf,sizeof(buf)-1);
        puts(buf);
        sleep(1);
    }

    if( FD_ISSET(fd2,&fdrd) )
    {
        read(fd2,buf,sizeof(buf)-1);
        puts(buf);
        sleep(1);
    }

    if( FD_ISSET(fd3,&fdwr) ) //如果 fd3 设置的是写绑定
    {
        write(fd3,buf,sizeof(buf));
        sleep(2);
    }

    if( FD_ISSET(fd4,&fdwr) )
    {
        write(fd4,buf,sizeof(buf));
        sleep(2);
    }
}
```

```
    close(fd1);
    close(fd2);
    close(fd3);
    close(fd4);
}
```

3.2.15 串口编程(选修)

who 看有多少个终端 设备文件的文件/dev 下 echo "hello" >/dev/pts/0

示例：端口编程

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
#include <string.h>
int main()
{
    int fd = open("/dev/pts/2",O_RDWR);
    while(1)
    {
        char buf1[512] = {0};
        char buf2[512] = {0};
        if(read(STDIN_FILENO, buf1, sizeof(buf1)-1) > 0)
        {
            if(strstr(buf1,"EOF"))    break;
            sprintf(buf2, "\njerry say:\n    %s",buf1);
            write(fd, buf2, sizeof(buf2));
        }
    }
    close(fd);
}
```

串口概述：用户常见的数据通信的基本方式分为并行通信和串行通信。

并行通信：利用多条数据传输线将一个资料的各位同时传送。特点是传输速度快，适用于短距离通信，但要求传输速度较高的应用场合。

串行通信：利用一条传输线将资料一位位地顺序传送。特点是通信线路简单，利用简单的线缆就可实现通信，降低成本，适用于长远距离通信，但传输速度慢的应用场合。

串行口是计算机一种常用的接口，具有连接线少，通讯简单，得到广泛的使用。

常用的串口是 RS-232-C 接口，它是 25 针 DB25 连接器

常见的一些串口参数的配置如图 3.1

序号	信号名称	符号	流向	功能
2	发送数据	TXD	DTE→DCE	DTE发送串行数据
3	接收数据	RXD	DTE←DCE	DTE 接收串行数据
4	请求发送	RTS	DTE→DCE	DTE 请求 DCE 将线路切换到发送方式
5	允许发送	CTS	DTE←DCE	DCE 告诉 DTE 线路已接通可以发送数据
6	数据设备准备好	DSR	DTE←DCE	DCE 准备好
7	信号地			信号公共地
8	载波检测	DCD	DTE←DCE	表示 DCE 接收到远程载波
20	数据终端准备好	DTR	DTE→DCE	DTE 准备好
22	振铃指示	RI	DTE←DCE	表示 DCE 与线路接通，出现振铃

图 3.1

Linux 中串口文件是位于/dev 下，串口 1 为 /dev/ttyS0，串口 2 为 /dev/ttyS1。

tty 看当前自己的设备文件名称

stty -a 看终端所有属性

如何配置终端：用命令

#stty erase x, (将退格键设置为 x)

#stty eof ^E (将文件结束设置为 ctrl+E)

#stty -echo 取消回显 #stty echo 加上回显

可以利用 minicom -s 选 port 设置 按 A:/dev/ttyS0 按 E:先按 E 再按 Q

串口的操作步骤如下：

在串口编程之前，需要执行下面的操作，首先你的机子上要安装 Serial Port(串口),例如，在 VM 下的 settings... 下在 “Hardware” 下可以执行 Add 添加，然后再列表中选 “Serial Port”，在右边选择 “Use output file”，然后浏览保存到 windows 下的一个文件中。

就可以直接很简单的进行串口编程了：

```
int fd = open("/dev/ttyS0", O_RDWR | O_SYNC);
char buf[8] = {"hello"};
write(fd, buf, 8);
close(fd);
```

也可以直接利用 echo “asjdf” > /dev/ttyS0 往里面写内容。

1.打开串口是通过使用标准的文件打开函数操作：

```
int fd = open( "/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY); /*以读写方式打开串口*/
if ( -1 == fd){/* 不能打开*/ perror(" 提示错误！ "); exit(-1);}
```

O_NOCTTY 标志通知 linux 系统，这个程序不会成为对应这个端口的控制终端（也就是不能是当前终端）。如果没有指定这个标志，那么任何一个输入都将会影响用户的进程。

O_NDELAY 标志通知 linux 系统，这个程序不关心 DCD 信号线所处的状态（端口的另一端是否激活或者停止）。如果用户指定了这个标志，则进程将会一直处于睡眠状态，直到 DCD 信号线被激活。

接下来可恢复串口的状态为阻塞状态，用于等待串口数据的读入。可用 fcntl 函数实现：

```
fcntl(fd, F_SETFL, 0);
```

再接下来可以测试打开文件描述符是否引用一个终端设备，以进一步确认串口是否正确打开：

```
isatty(STDIN_FILENO);
```

2.设置串口：波特率，校验位和停止位，设置信息在 struct termios 结构体中：

```
struct termios {
    unsigned short c_iflag;    /* 输入模式 */
```

```

    unsigned short c_oflag;    /* 输出模式 */
    unsigned short c_cflag;    /* 控制模式 */
    unsigned short c_lflag;    /* 本地模式 */
    unsigned char c_cc[NCC];   /* 控制字符 */
};
    
```

在这个结构体中，最重要的是 `c_cflag`，通过对它的赋值，用户可以设置波特率、字符大小、数据位、停止位、奇偶校验位和硬件流控等。其中设置波特率为相应的波特率前加上‘B’。还有 `c_iflag` 和 `c_cc` 也是比较常用的标志。如表 3.4 所示：

表 3.4： `f_cflag` 支持的常量名称

CBAUD	波特率的位掩码
B0	0 波特率（放弃 DTR）
B1800	1800 波特率
B2400	2400 波特率
B4800	4800 波特率
B9600	9600 波特率
B19200	19200 波特率
B38400	38400 波特率
B57600	57600 波特率
B115200	115200 波特率
EXTA	外部时钟率
EXTB	外部时钟率
CSIZE	数据位的位掩码
CS5	5 个数据位（发送和接收时使用 5 比特）
CS6	6 个数据位（发送和接收时使用 6 比特）
CS7	7 个数据位（发送和接收时使用 7 比特）
CS8	8 个数据位（发送和接收时使用 8 比特）
CSTOPB	2 个停止位（不设则是 1 个停止位）
CREAD	接收使能
PARENB	校验位使能
PARODD	使用奇校验而不适用偶校验
HUPCL	最后关闭时挂线（放弃 DTR）
CLOCAL	本地连接（不改变端口所有者）
LOBLK	块作业控制输出
CNET_CTSRTS	硬件流控制使能

在这里，对于 `c_cflag` 成员不能直接对其初始化，而要将其通过“与”、“或”操作使用其中的某些选项。

输入模式 `c_iflag` 成员控制端口接收端的字符输入处理如表 3.5 所示。

表 3.5： `c_iflag` 支持的常量名称

INPCK	奇偶校验使能
IGNPAR	忽略奇偶校验错误的字符
PARMRK	对奇偶校验错误做出标记
ISTRIP	去掉第 8 位，将所有接收到的字符裁剪为 7 比特
IXON	启动出口硬件流控

IXOFF	启动入口硬件流控
IXANY	允许字符重新启动流控
IGNBRK	忽略中断情况
BRKINT	当发生中断时发送 SIGINT 信号
INLCR	将 NL 映射到 CR（将新行符—回车符）
IGNCR	忽略 CR（忽略回车符）
ICRNL	将 CR 映射到 NL（将回车符—新行符）
IUCLC	将高位情况映射到低位情况
IMAXBEL	当输入太长时恢复 ECHO

c_cc 包含了超时参数和控制字符的定义，如表 3.6 所示

表 3.6：c_cc 支持的常量名称

VINTR	中断控制，对应键为 CTRL+C
VQUIT	退出操作，对应键为 CTRL+Z
VERASE	删除操作，对应键为 Backspace（BS）
VKILL	删除行，对应键为 CTRL+U
VEOF	位于文件结尾,对应键为CTRL+D
VEOL	位于行尾，对应键为 Carriage return（CR）
VEOL2	位于第二行尾，对应键 Line feed（LF）
VMIN	指定了最少读取的字符数
VTIME	指定了读取每个字符的等待时间

下面就详细讲解设置串口属性的基本流程：

1. 保存原先串口配置

首先，为了安全起见和以后调试程序方便，可以先保存原先串口的配置，在这里可以使用函数 tcgetattr(fd, &oldtio)。该函数得到与 fd 指向对象的相关参数，并将它们保存于 oldtio 引用的 termios 结构体中，该函数可以测试配置是否正确，该串口是否可用等。若调用成功，函数返回值为 0，若调用失败，函数返回值为-1.其使用如下所示：

```

struct termios newtio, oldtio;
if (tcgetattr(fd, &oldtio) == -1) { perror("tcgetattr error"); exit(-1); }
bzero(&newtio, sizeof(struct termios)); //→memset(&newtio, 0, sizeof(struct termios));

```

2. 激活选项有 CLOCAL 和 CREAD

CLOCAL 和 CREAD 用于本地连接和接受使能，因此，首先要通过位掩码的方式激活这两项：

```

newtio.c_cflag |= CLOCAL | CREAD;

```

3. 设置波特率

设置波特率有专门的函数，用户不能直接通过位掩码方式来操作。设置波特率的主要函数有：cfsetispeed 和 cfsetospeed。这两个函数的使用很简单，如下所示：

```

cfsetispeed(&newtio, B115200);
cfsetospeed(&newtio, B115200);

```

一般情况下，用户需将输入输出的波特率设为一样。函数成功返回 0，失败返回-1

4. 设置字符大小

与设置波特率不同，设置字符大小并没有现成可用的函数，需要用位掩码。一般首先去除数据位中的位掩码，再重新按要求设置。如下所示：

```

options.c_cflag &= ~CSIZE;
options.c_cflag |= CS8;

```


5. 设置奇偶校验位

设置奇偶校验位需要用到两个 `termios` 中的成员：`c_cflag` 和 `c_iflag`。首先要激活 `c_cflag` 中的检验位使能标志 `PARENB` 和是否进行偶校验，同时还要激活 `c_iflag` 中的奇偶校验使能。

如使能奇校验时，代码如下所示：

```
newtio.c_cflag |= PARENB;
newtio.c_cflag |= PARODD;
newtio.c_iflag |= (INPCK | ISTRIP);
```

如使能偶校验时，代码如下所示：

```
newtio.c_cflag |= PARENB;
newtio.c_cflag &= ~PARODD;
newtio.c_iflag |= (INPCK | ISTRIP);
```

无奇偶校验位时，代码如下所示：

```
newtio.c_cflag &= ~PARENB;
```

6. 设置停止位

设置停止位是通过激活 `c_cflag` 中的 `CSTOPB` 而实现的。若停止位为 1，则清除 `CSTOPB`，若停止位为 2，则激活 `CSTOPB`。下面是停止位是 1 时的代码：

```
newtio.c_cflag &= ~CSTOPB;
```

7. 设置最少字符和等待时间

在对接收字符和等待时间没有特殊要求的情况下，可以将其设置为 0。如下所示：

```
newtio.c_cc[VTIME] = 0;
newtio.c_cc[VMIN] = 0;
```

8. 处理要写入的引用对象

由于串口在重新设置之后，在此之前要写入的引用对象要重新处理，这时就可调用函数 `tcflush(fd, queue_selector)` 来处理要写入引用的对象。对于尚未传输的数据，或者收到的但是尚未读取的数据，其处理方式取决于 `queue_selector` 的值。

这里，`queue_selector` 可能的取值有以下几种：

TCIFLUSH: 刷新收到的数据但是不读

TCOFLUSH: 刷新写入的数据但是不传送

TCIOFLUSH: 同时刷新收到的数据但是不读，并刷新写入的数据但是不传送

示例：`tcflush(fd, TCIFLUSH);`

9. 激活配置

在完成全部串口配置之后，要激活刚才的配置并使配置生效，这里用到的函数是 `tcsetattr`，这个函数的原型是 `tcsetattr(fd, OPTION, &newtio);` 成功返回 0，失败返回-1。

这里的 `newtio` 就是 `termios` 类型的变量，`OPTION` 可能的取值有以下 3 种：

TCSANOW: 改变的配置立即生效

TCSADRAIN: 改变的配置在所有写入 `fd` 的输出都结束后生效

TCSAFLUSH: 改变的配置在所有写入 `fd` 引用对象的输出都被结束后生效，所有已接受但未读入的输入都在改变发生前丢弃。

3 读写串口

读写串口就是把串口当文件去读写

发送数据

```
char buffer[1024];
int nwrite = write(fd, buffer, 1024)
```

读取串口数据

```
char buff[1024];
while((nread = read(fd, buff, 1024))>0)
{
    buff[nread+1] = '\0';
```

```
printf( "\n%s", buff);  
}
```

4. 关闭串口就是关闭文件。

```
close(fd);
```

示例：串口编程

```
#include <termios.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <errno.h>  
int main()  
{  
    //open  
    int fd = open("/dev/ttyS0", O_WRONLY);  
    if(fd == -1)  
    {  
        perror("open error");  
        exit(-1);  
    }  
  
    //setattr  
    //1. save old attr  
    struct termios newtms, oldtms;  
    bzero(&newtms, sizeof(struct termios));  
    if( tcgetattr(fd, &oldtms) == -1)  
    {  
        perror("tcgetattr error");  
        exit(-1);  
    }  
  
    //2. set CREAD CLOCAL  
    newtms.c_cflag |= CREAD;  
    newtms.c_cflag |= CLOCAL;  
  
    //3. set baudspeed  
    if(cfsetispeed(&newtms, B115200) == -1)  
    {  
        perror("cfsetispeed error");  
        exit(-1);  
    }  
    if(cfsetospeed(&newtms, B115200) == -1)  
    {  
        perror("cfsetospeed error");  
        exit(-1);  
    }  
}
```

```
}

//4. set char size
newtms.c_cflag &= ~CSIZE;
newtms.c_cflag |= CS8;

//5. set ji'ou check
newtms.c_cflag &= ~PARENB;

//6. set stop bit
newtms.c_cflag &= ~CSTOPB;

//7. set time and min size
newtms.c_cc[VTIME] = 0;
newtms.c_cc[VMIN] = 0;

//8. tcflush
if(tcflush(fd, TCIOFLUSH) == -1)
{
    perror("tcflush error");
    exit(-1);
}

//9. set attr
if(tcsetattr(fd, TCSANOW, &newtms) == -1)
{
    perror("tcsetattr error");
    exit(-1);
}

//read/write
char buf[512] = {'\0'};
int ir = 0;
if((ir = read(STDIN_FILENO, buf, 512)) > 0)
{
    if(write(fd, buf, ir) == -1)
    {
        perror("write error");
        exit(-1);
    }
}

//close
close(fd);
return 0;
}
```

补充:

1. 程序参数:

深圳信盈达科技有限公司 专业提供单片机、嵌入式、ARM、LINUX、Android、PCB、FPGA 等技术培训、技术方案。

```
int main(int argc,char ** argv)
{
    int i = 0;
    for( ; i< argc; i++)
        printf("%s",argv[i]);
}
```

参数 `argc` 表示命令行传入的参数个数，并且一次保存到 `argv` 数组中。例如程序叫 `main.c`，最后执行的时候，如果直接是 `./main` 则 `argc=1`，此时 `argv[0]` 就是 `./main`；如果是 `./main aaa bbb`，则 `argc=3`，`argv[0]:./main`，`argv[1]:aaa` `argv[2]:bbb`。

2. 日志

许多应用程序需要记录它们的活动，系统程序经常需要向控制台或日志文件写消息。这些消息可能指示错误、警告或者与系统状态有关的一般信息。通常是在 `/var/log` 目录下的 `messages` 中包含了系统信息。通过 `syslog` 可以向系统的日志发送日志信息。

函数原型如下：

```
#include <syslog.h>
void syslog(int priority, const char* message, arguments...);
```

对于 `priority` 有如下几个常见的：

<code>LOG_EMERG</code>	紧急情况
<code>LOG_ALERT</code>	高优先级故障（如：数据库崩溃）
<code>LOG_CRIT</code>	严重错误（如：硬件错误）
<code>LOG_ERR</code>	错误
<code>LOG_WARNING</code>	警告
<code>LOG_NOTICE</code>	需要注意的特殊情况
<code>LOG_INFO</code>	一般信息
<code>LOG_DEBUG</code>	调试信息(写不到 <code>messages</code> 里面)

```
#include <syslog.h>
main()
{
    //openlog("log",LOG_PID|LOG_CONS|LOG_NOWAIT,LOG_USER);
    syslog(LOG_ALERT,"this is alert\n");
    syslog(LOG_INFO,"this is info\n");
    syslog(LOG_DEBUG,"this is debug%d\t %s",10,"aaaa");
    syslog(LOG_ERR,"err");
    syslog(LOG_CRIT,"crit");
    //closelog();
}
```

利用 `tail -10 /var/log/messages` 可以查看。

还可以通过函数 `openlog` 函数来改变日志信息的表达方式。`openlog` 的原型如下：

```
#include <syslog.h>
void openlog(const char* ident, int logopt, int facility);
void closelog(void);
```

它可以设置一个字符串 `ident`，该字符串会添加在日志信息的前面。你可以通过它来指明是哪个应用程序创建了这条信息。`facility` 值为 `LOG_USER`。`logopt` 参数对后续 `syslog` 调用的行为进行配置。如下：

<code>LOG_PID</code>	在日志信息中包含进程标识符，这是系统分配给每个进程的一个唯一值
<code>LOG_CONS</code>	如果信息不能被记录到日志文件中，就把它发送到控制台

第4章 Linux 多进程

4.1 Linux 进程概述

进程是一个程序一次执行的过程，是操作系统动态执行的基本单元。进程的概念主要有两点：第一，进程是一个实体。每个进程都有自己的虚拟地址空间，包括文本区、数据区、和堆栈区。文本区域存储处理器执行的代码；数据区存储变量和动态分配的内存；堆栈区存储着活动进程调用的指令和本地变量。第二，进程是一个“执行中的程序”，它和程序有本质区别。程序是静态的，它是一些保存在磁盘上的指令的有序集合；而进程是一个动态的概念，它是一个运行着的程序，包含了进程的动态创建、调度和消亡的过程，是 Linux 的基本调度单位。只有当处理器赋予程序生命时，它才能成为一个活动的实体，称之为进程。

内核的调度器负责在所有的进程间分配 CPU 执行时间，称为时间片(time slice)，它轮流在每个进程分得的时间片用完后从进程那里抢回控制权。

4.1.1 进程标识

OS 会为每个进程分配一个唯一的整型 ID，做为进程的**标识号(pid)**。进程 0 是调度进程，常被成为交换进程，它不执行任何程序，是内核的一部分，因此也被成为系统进程。进程除了自身的 ID 外，还有**父进程 ID(ppid)**。也就是说每个进程都必须有它的父进程，操作系统不会无缘无故产生一个新进程。所有进程的祖先进程是同一个进程，它叫做 **init 进程**，ID 为 1，init 进程是内核自举后的第一个启动的进程。init 进程负责引导系统、启动守护（后台）进程并且运行必要的程序。它不是系统进程，但它以系统的超级用户特权运行。

4.1.2 进程的用户 ID 与组 ID(进程的运行身份)

进程在运行过程中，必须具有类似于用户的身份，以便进行进程的权限控制，缺省情况下，哪个登录用户运行程序，该程序进程就具有该用户的身份。例如，假设当前登录用户为 gotter，他运行了 ls 程序，则 ls 在运行过程中就具有 gotter 的身份，该 ls 进程的用户 ID 和组 ID 分别为 gotter 和 gotter 所属的组。这类型的 ID 叫做进程的真实用户 ID 和真实组 ID。真实用户 ID 和真实组 ID 可以通过函数 getuid()和 getgid()获得。

与真实 ID 对应，进程还具有有效用户 ID 和有效组 ID 的属性，内核对进程的访问权限检查时，它检查的是进程的有效用户 ID 和有效组 ID，而不是真实用户 ID 和真实组 ID。缺省情况下，用户的（有效用户 ID 和有效组 ID）与（真实用户 ID 和真实组 ID）是相同的。有效用户 id 和有效组 id 通过函数 geteuid()和 getegid()获得。

示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    printf("uid:%d gid:%d euid:%d egid:%d\n",getuid(),getgid(),geteuid(),getegid());
    return 0;
}
```

shell>id

uid=500(ghaha) gid=500(ghaha) groups=500(ghaha)

编译生成可执行文件 a.out，程序文件的属性可能为：

```
-rwxrwxr-x    1 ghaha    ghaha        12132 Oct  7 09:26 a.out
```

执行结果可能为：

```
shell>./a.out
```

```
uid:500 gid:500 euid:500 egid:500
```

现在将 a.out 的所有者可执行属性改为 s

```
shell>chmod u+s a.out
```

```
shell>ll
```

```
-rwsrwxr-x    1 ghaha    ghaha        12132 Oct  7 09:26 a.out
```

此时改另外一个用户 gotter 登录并运行程序 a.out

```
shell>id
```

```
uid=502(gotter) gid=502(gotter) groups=502(gotter)
```

```
shell>./a.out
```

```
uid:502 gid:502 euid:500 egid:502
```

可以看到,进程的有效用户身份变为了 ghaha,而不是 gotter 了,这是因为文件 a.out 的访问权限的所有者可执行为设置了 s 的属性,设置了该属性以后,用户运行 a.out 时,a.out 进程的有效用户身份将不再是运行 a.out 的用户,而是 a.out 文件的所有者。

s 权限最常见的例子是

/usr/bin/passwd 程序,它的权限位为

```
shell>ll /usr/bin/passwd
```

```
-r-s--x--x    1 root      root          16336 Feb 13  2003 /usr/bin/passwd
```

我们知道,用户的用户名和密码是保存在/etc/passwd (后来专门将密码保存在/etc/shadow,它是根据/etc/passwd 文件来生成/etc/shadow 的,它把所有口令从/etc/passwd 中移到了/etc/shadow 中。这里用到的是影子口令,它将口令文件分成两部分:/etc/passwd 和/etc/shadow,此时/etc/shadow 就是影子口令文件,它保存的是加密的口令,而/etc/passwd 中的密码全部变成 x) 下的。通过 ls -l 查看/etc/passwd 这个文件,你会发现,这个文件普通用户都没有可写的权限,那我们执行 passwd 的时候确实能够修改密码,那么这是怎么回事呢?也就是说,任何一个用户运行该程序时,该程序的有效身份都将是 root (用普通身份去执行这个操作的时候,它会暂时得到文件拥有者 root 的权限),而这样 passwd 程序才有权读取/etc/passwd 文件的信息。

模拟 passwd 实现步骤:

1. 用 touch 创建一个 a.txt 输入内容 “hello”
2. 写一个程序 1.c 如下:

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

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

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

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

```
#include <errno.h>
```

```
int main()
```

```
{
```

```
    printf("uid : %d    gid : %d\n", getuid(), getgid());
```

```
    printf("euid : %d    egid : %d\n", geteuid(), getegid());
```

```
    FILE* fp = fopen("a.txt", "a");//注意这里是以追加形式打开,说明 a.txt 要具有可写权限
```

```
    if(fp == NULL)
```

```
    {
```

```
        perror("fopen error");
```

```
        exit(-1);
```

```
    }
```

```
    fputs("world", fp);
```

```
    fclose(fp);
```

```
    return 0;
```

```
}
```

3. 编译 gcc -o 1 1.c 生成可执行程序 1

4. 用 ll 查看 a.txt 发现权限是 -rw-r--r-- 的权限,说明普通用户没有可写权限,如果直接切换到普通用户执行 ./1 会报错。

5. 用 root 身份将 1 的权限修改为 -rwsr-xr-x (操作命令: chmod u+s 1),此时再切换到普通用户 wangxiao,

深圳信盈达科技有限公司 专业提供单片机、嵌入式、ARM、LINUX、Android、PCB、FPGA 等技术培训、技术方案。

执行./1 发现可以执行成功，因为此时普通用户借助的是 root 的身份。

4.1.3 进程的状态

进程是程序的执行过程，根据它的生命周期可以划分成 3 种状态，如图 4.1 所示。

- 执行态：该进程正在运行，即进程正在占用 CPU。
- 就绪态：进程已经具备执行的一切条件，正在等待分配 CPU 的处理时间片。
- 等待态：进程不能使用 CPU，若等待事件发生（等待的资源分配到）则可将其唤醒。

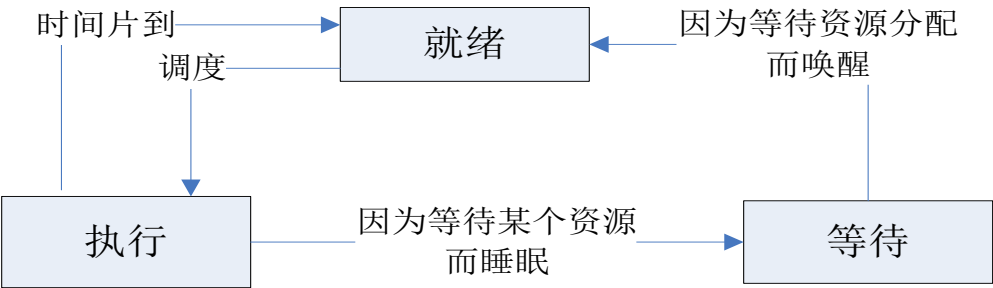


图 4.1

4.1.4Linux 下的进程结构及管理

Linux 系统是一个多进程的系统，它的进程之间具有并行性、互不干扰等特点。也就是说，进程之间是分离的任务，拥有各自的权利和责任。其中，每个进程都运行在各自独立的虚拟地址空间，因此，即使一个进程发生了异常，它也不会影响到系统的其他进程。进程中各段如图 4.2 和 4.3 所示。



图 4.2

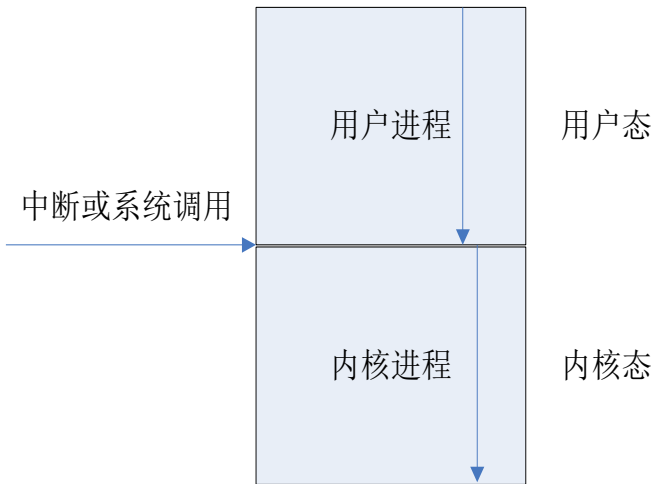


图 4.3

Linux 中的进程包含以下几个部分：

- “数据段”放全局变量、常数以及动态数据分配的数据空间。数据段分成普通数据段（包括可读可写/只读数据段，存放静态初始化的全局变量或常量）、BSS 数据段（存放未初始化的全局变量）以及堆（存放动态分配的数据）。
- “正文段”存放的是 CPU 执行的机器指令部分。
- “堆栈段”存放的是子程序的返回地址、子程序的参数以及程序的局部变量等。

一些进程相关信息：

进程 process：是 os 的最小单元 大小为 4g，其中 1g 给 os 3g 给进程 {代码区 数据区 堆 栈}。

ps：查看活动进程。

ps -aux：查看各个进程状态，包括运行就绪等待等状态。

ps -aux| grep 'aa': 查找指定（aa）进程。

ps -ef：查看所有的进程的 pid，ppid 等信息。

ps -aux 看%cpu(cpu 使用量) %mem（内存使用量）

stat 状态 {S 就绪 T 中断 R 运行 Z 僵尸}

via.c & (&表示后台运行)，

jobs 查看后台任务，fg 1 把后台任务带到前台 ctrl+z 把进程带入后台

kill -9 进程号➔杀掉某个进程 top 显示前 20 条进程，动态的改变

pgrep 'vi'查找进程

nice 改变优先级

crontab 计划任务，定时操作等

4.2 Linux 进程的创建与控制

4.2.1 fork 函数

原型：

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

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

在 linux 中 fork 函数是非常重要的函数，它从已存在进程中创建一个新进程。新进程为子进程，而原进程为父进程。它和其他函数的区别在于：它执行一次返回两个值。其中父进程的返回值是子进程的进程号，而子进程的返回值为 0。若出错则返回-1。因此可以通过返回值来判断是父进程还是子进程。

fork 函数创建子进程的过程为：使用 fork 函数得到的子进程是父进程的一个复制品，它从父进程继承了进程的地址空间，包括进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设定、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端，而子进程所独有的只有它的进程号、资源使用和计时器等。通过这种复制方式创建出子进程后，原有进程和子进程都从函数 fork 返回，各自继续往下运行。也就是说，fork 不仅仅复制了进程的资源，更复制了原有进程的运行状态，所以复制出的新的进程（子进程）虽然什么都和父进程一样，但它从 fork 函数后面开始运行。但是原进程的 fork 返回值与子进程的 fork 返回值不同，在原进程中，fork 返回子进程的 pid，而在子进程中，fork 返回 0，如果 fork 返回负值，表示创建子进程失败。

（vfork 函数）其作用和返回值与 fork 相同，但有一些区别。二者都创建一个子进程，但是它并不是将父进程的地址空间完全复制到子进程中，因为子进程会立即调用 exec（或 exit），所以也就不会存放该地址空间。而且 vfork 保证子进程先比父进程先运行，在它调用 exec 或 exit 之后父进程才可能被调度运行。

示例 1:

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    fork();
    //fork();
    //fork();
    printf("Hello World\n");
    return 0;
}
```

自己运行上面的程序看看结果是什么，并思考为什么会这样。如果有多个 fork() 呢？并输出其返回值。

示例 2:

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    int m = 100;
    printf("aaaa\n");
    int n = fork();
    if(n > 0)
    {
        m++;
        printf("bbbb n = %d m = %d\n", n, m);
    }
    else
    {
        sleep(2);
        printf("bbbb n = %d m = %d\n", n, m);
    }
    return 0;
}
```

示例 3:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
```

```
printf("父进程的 id:%d\n",getpid());
pid_t n = fork();
if(n == 0)
{
    printf("子进程的 id:%d ppid:%d\n",getpid(),getppid());
}
else
{
    printf("创建子进程成功,子进程 id:%d\n\n");
}
return 0;
}
```

此时相当于有两份 main 函数代码的拷贝，其中一份做的操作是 if(iRet == 0)的情况；另外一份做的操作是 else(父)的情况。所以可以输出 2 句话。

fork 以后的子进程自动继承了父进程的打开的文件，继承以后，父进程关闭打开的文件不会对子进程造成影响。

示例 4：用 fork 继承父进程打开的文件

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{
    char szBuf[32] = {"0"};
    int iFile = open("./a.txt", O_RDONLY);
    if(fork() > 0)//父进程
    {
        close(iFile);
        return 0;
    }
    //子进程
    sleep(3); //wait for parent process closing fd
    read(iFile, szBuf, sizeof(szBuf)-1)
    printf("string:%s\n",szBuf);
    close(iFile);
    return 0;
}
```

4.2.2 进程的终止

进程的终止有 5 种方式：

- main 函数的自然返回；
- 调用 exit 函数
- 调用 _exit 函数
- 接收到某个信号。如 ctrl+c SIGINT ctrl+\ SIGQUIT
- 调用 abort 函数，它产生 SIGABRT 信号，所以是上一种方式的特例。

前 3 种方式为正常的终止，后 2 种为非正常终止。但是无论哪种方式，进程终止时都将执行相同的关闭打开的文件，释放占用的内存等资源。只是后两种终止会导致程序有些代码不会正常的执行比如对象的析构、atexit

函数的执行等。

`exit` 和 `_exit` 函数都是用来终止进程的。当程序执行到 `exit` 和 `_exit` 时，进程会无条件的停止剩下的所有操作，清除包括 PCB 在内的各种数据结构，并终止本程序的运行。但是它们是有区别的，`exit` 和 `_exit` 的区别如图 4.4 所示：

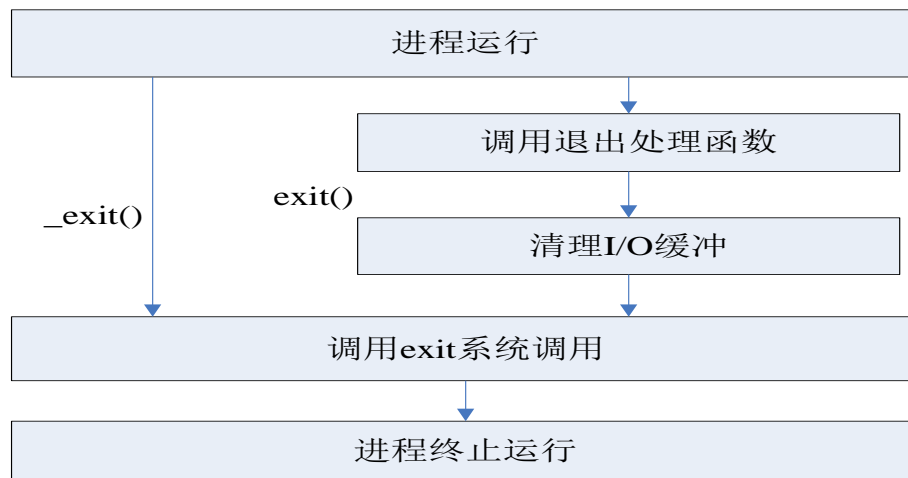


图 4.4

`exit` 函数和 `_exit` 函数的最大区别在于 `exit` 函数在退出之前会检查文件的打开情况，把文件缓冲区中的内容写回文件，就是图中的“清理 I/O 缓冲”。

由于 linux 的标准函数库中，有一种被称作“缓冲 I/O”操作，其特征就是对应每一个打开的文件，在内存中都有一片缓冲区。每次读文件时，会连续读出若干条记录，这样在下次读文件时就可以直接从内存的缓冲区中读取；同样，每次写文件的时候，也仅仅是写入内存中的缓冲区，等满足一定的条件（如达到一定数量或遇到特定字符等），再将缓冲区中的内容一次性写入文件。这种技术大大增加了文件读写的速度，但也为编程带来了麻烦。比如有一些数据，认为已经写入文件，实际上因为没有满足特定的条件，它们还只是保存在缓冲区内，这时用 `_exit` 函数直接将进程关闭，缓冲区中的数据就会丢失。因此，如想保证数据的完整性，建议使用 `exit` 函数。

`exit` 和 `_exit` 函数的原型：

```
#include <stdlib.h> //exit 的头文件
#include <unistd.h> // _exit 的头文件
void exit(int status);
void _exit(int status);
```

`status` 是一个整型的参数，可以利用这个参数传递进程结束时的状态。一般来说，0 表示正常结束；其他的数值表示出现了错误，进程非正常结束。

示例 1： `exit` 的举例如下：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("hello\n");
    printf("world");
    exit(0);
}
```

可以发现，调用 `exit` 函数，缓冲区中的记录也能正常输出。

示例 2： `_exit` 的举例如下：

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("hello\n");
    printf("world");
}
```

```
    _exit(0);    正常退出
}
```

可以发现，最后的输出结果没有 world，说明_exit 函数无法输出缓冲区中的记录。

4.2.3 wait 和 waitpid 函数

用 fork 函数启动一个子进程时，子进程就有了它自己的生命并将独立运行。

如果父进程先于子进程退出，则子进程成为**孤儿进程**，此时将自动被 PID 为 1 的进程（即 init）接管。孤儿进程退出后，它的清理工作有祖先进程 init 自动处理。但在 init 进程清理子进程之前，它一直消耗系统的资源，所以要尽量避免。

示例 1：写一个孤儿进程：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
main()
{
    pid_t pid = fork();
    if( pid == 0)
    {
        printf("子进程...\n");
        while(1);
    }
    else
    {
        printf("父进程 8 秒后退出...\n");
        sleep(8);
        printf("父进程退出\n");
        exit(10);
    }
}
```

通过 ps -ef 就可以看到此时子进程一直在运行，并且父进程是 1 号进程。

如果子进程先退出，系统不会自动清理掉子进程的环境，而必须由父进程调用 wait 或 waitpid 函数来完成清理工作，如果父进程不做清理工作，则已经退出的子进程将成为僵尸进程(defunct),在系统中如果存在的僵尸（zombie）进程过多，将会影响系统的性能，所以必须对僵尸进程进行处理。

函数原型：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

wait 和 waitpid 都将暂停父进程，等待一个子进程退出，并进行清理工作；

wait 函数随机地等待一个子进程退出，并返回该子进程的 pid；

waitpid 等待指定 pid 的子进程退出；如果为-1 表示等待所有子进程，同样返回该子进程的 pid。

status 参数是传出参数，它会将子进程的退出码保存到 status 指向的变量里

比如 waitpid(pid, &n, 0)；其中 n 是之前定义的整型变量。

通常用下面的两个宏来获取状态信息：

WIFEXITED(n) 如果子进程正常结束，它就取一个非 0 值。

WEXITSTATUS(n) 如果 WIFEXITED 非零，它返回子进程的退出码，即子进程的 exit()函数的参数的值，或者 return 返回的值。

options 用于改变 waitpid 的行为，其中最常用的是 WNOHANG，它表示无论子进程是否退出都将立即返回，不会将调用者的执行挂起。

示例 1： 写一个僵尸进程：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
main()
{
    pid_t pid = fork();
    if( pid == 0 )
    {
        exit(10);
    }
    else
    {
        sleep(10);
        //while(1);
    }
}
```

通过用 ps - aux 快速查看发现 Z 的僵尸进程。

示例 2： 避免僵尸进程：(wait()函数)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void)
{
    pid_t ret, rew;

    ret = fork();
    if(ret == 0)
    {
        printf("子进程的 pid = %d\n", getpid());
        sleep(2);    //sleep(260);
        exit(10);
    }
    else
    {
        /*等待子进程结束*/
        int n;
        rew = waitpid(ret, &n, 0);
        printf("父进程中 rew = %d ret = %d\n", rew, ret);
        printf("n = %d\n", n);
        sleep(10);
    }
    return 0;
}
```

```
}
```

通过运行测试发现，如果子进程中是 `exit(m)` 或者 `return m`，则父进程中 `n` 的值最终为 $(m\%256)*256$ 。

4.2.4 exec 函数族

exec* 由一组函数组成

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

exec 函数族的作用是运行第一个参数指定的可执行程序。但其工作过程与 `fork` 完全不同，`fork` 是在复制一份原进程，而 `exec` 函数执行第一个参数指定的可执行程序之后，这个新程序运行起来后也是一个进程，而这个进程会覆盖原有进程空间，即原有进程的所有内容都被新运行起来的进程全部覆盖了，所以 `exec` 函数后面的所有代码都不再执行，但它之前的代码当然是可以被执行的。

`path` 是包括执行文件名的全路径名 `file` 既可是全路径名也可是可执行文件名

`arg` 是可执行文件的全部命令行参数，可以用多个，注意最后一个参数必须为 `NULL`。

`argv` 是一个字符串的数组 `char *argv[]={“full path”, “param1”, “param2”, ..., NULL};`

`envp` 指定新进程的环境变量 `char *envp[]={“name1=val1”, “name2=val2”, ..., NULL};`

exec 函数族的参数传递有两种方式：一种是逐个列举的方式，而另一种则是将所有参数整体构造指针数组传递。在这里是以函数名的第 5 位字母来区分的，字母为“l”（list）的表示逐个列举的方式，其语法为 `char *arg`；字母为“v”（vector）的表示将所有命令行参数整体构造指针数组传递，其语法为 `char *const argv[]`。

以字母 `p` 结尾的函数通过搜索系统 `PATH` 这个环境变量来查找新程序的可执行文件的路径。如果可执行程序不在 `PATH` 定义的路径中，我们就需要把包括目录在内的使用绝对路径的文件名作为参数传递给函数。

对有参数 `envp` 的函数调用，其以字母 `e` 结尾，函数通过传递 `envp` 传递字符串数组作为新程序的环境变量。新进程中的全局变量 `environ` 指针指向的环境变量数组将会被 `envp` 中的内容替代。

注意：对于有参数 `envp` 的函数，它会使用程序员自定义的环境变量，如果自定义的环境变量中包含了将要执行的可执行程序的路径，那么第一个参数中是不是我们就可以不用写全路径了呢？不是的，必须写全路径。因为我们自定义的环境变量不是用来寻找这个可执行程序的，而是在这个可执行程序运行起来之后给新进程用的。

可以用 `env` 命令查看环境变量。

execl 示例：

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    printf("aaaa\n");
    execl("/bin/ls", "ls", "-l", NULL);
    printf("bbbb\n");
    printf("bbbb\n");
    return 0;
}
```

execlp 示例：（在上面的基础上去掉了 `ls` 的路径 `/bin/`）

```
#include <unistd.h>
#include <stdio.h>
int main()
{
```



```
    printf("aaaa\n");
    execlp("ls", "ls", "-l", NULL);
    printf("bbbb\n");
    printf("bbbb\n");
    return 0;
}
```

execv 示例：(把"ls","-l",NULL 这些命令行参数通过指针数组 str 传给 exec 函数)

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    char *str[] = {"ls", "-l", NULL};
    printf("aaaa\n");
    execv("/bin/ls", str);
    printf("bbbb\n");
    printf("bbbb\n");
    return 0;
}
```

execvp 示例：(在上面的基础上去掉了 ls 的路径/bin/)

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    char *str[] = {"ls", "-l", NULL};
    printf("aaaa\n");
    execvp("ls", str);
    printf("bbbb\n");
    printf("bbbb\n");
    return 0;
}
```

execle 示例：

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    char *arrEnv[] = {"PATH=/bin:/usr/bin", "TERM=console", NULL};
    printf("aaaa\n");
    execle("/bin/ls", "ls", "-l", NULL, arrEnv);
    printf("bbbb\n");
    printf("bbbb\n");
    return 0;
}
```

execve 示例：

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    char *str[] = {"ls", "-l", NULL};
    char *arrEnv[] = {"PATH=/bin:/usr/bin", "TERM=console", NULL};
```

```
printf("aaaa\n");
execve("/bin/ls", str, arrEnv);
printf("bbbb\n");
printf("bbbb\n");
return 0;
}
```

其它示例：get_arg.c 测试 environ，用来输出本进程默认的环境变量。

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
extern char **environ; //声明该变量
int main(int argc,char *argv[])
{
    int i=0;
    for(i<argc;i++)
        printf("arg%d:%s\n",i,argv[i]);
    char **ppEnv=environ;
    while(ppEnv && *ppEnv)
        printf("%s\n",*ppEnv++);
    return 0;
}
```

exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何 Linux 下可执行的脚本文件，如果不是可以执行的文件，那么就解释成为一个 shell 文件，sh **执行！

上面 6 条函数看起来似乎很复杂，但实际上无论是作用还是用法都非常相似，只有很微小的差别。

参数 argc 指出了运行该程序时命令行参数的个数，数组 argv 存放了所有的命令行参数，数组 envp 存放了所有的环境变量。环境变量指的是一组值，从用户登录后就一直存在，很多应用程序需要依靠它来确定系统的一些细节，我们最常见的环境变量是 PATH，它指出了应到哪里去搜索应用程序，如 /bin；HOME 也是比较常见的环境变量，它指出了我们在系统中的个人目录。环境变量一般以字符串"XXX=xxx"的形式存在，XXX 表示变量名，xxx 表示变量的值。

值得一提的是，argv 数组和 envp 数组存放的都是指向字符串的指针，这两个数组都以一个 NULL 元素表示数组的结尾。

前 3 个函数都是以 execl 开头的，后 3 个都是以 execv 开头的，它们的区别在于，execv 开头的函数是以"char *argv[]"这样的形式传递命令行参数，而 execl 开头的函数采用了我们更容易习惯的方式，把参数一个一个列出来，然后以一个 NULL 表示结束。这里的 NULL 的作用和 argv 数组里的 NULL 作用是一样的。

这里建议使用 (char *)0 代替 NULL。

在全部 6 个函数中，只有 execl 和 execve 使用了 char *envp[]传递环境变量，其它的 4 个函数都没有这个参数，这并不意味着它们不传递环境变量，这 4 个函数将把默认的环境变量不做任何修改地传给被执行的应用程序。而 execl 和 execve 会用指定的环境变量去替代默认的那些。

还有 2 个以 p 结尾的函数 execlp 和 execvp，乍看起来，它们和 execl 与 execv 的差别很小，事实也确是如此，除 execlp 和 execvp 之外的 4 个函数都要求，它们的第 1 个参数 path 必须是一个完整的路径，如"/bin/ls"；而 execlp 和 execvp 的第 1 个参数 file 可以简单到仅仅是一个文件名，如 "ls"，这两个函数可以自动到环境变量 PATH 制定的目录里去寻找。

4.2.5 system 函数

原型：

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

```
int system(const char *string);
```

system 函数通过调用 shell 程序/bin/sh -c 来执行 string 所指定的命令，该函数在内部是通过调用 fork、execve("/bin/sh",...)、waitpid 函数来实现的。通过 system 创建子进程后，原进程和子进程各自运行，相互间关联较少。如果 system 调用成功，将返回 0。

示例：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    system("ls -l"); //system("clear");表示清屏
    return 0;
}
```

4.2.6 popen 函数

popen 函数类似于 system 函数，与 system 的不同之处在于它使用管道工作。原型为：

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

command 为可执行文件的全路径和执行参数；

type 可选参数为“r”或“w”，如果为“w”，则 popen 返回的文件流做为新进程的标准输入流，即 stdin，如果为“r”，则 popen 返回的文件流做为新进程的标准输出流。

如果 type 是“r”，（即 command 命令执行的输出结果作为当前进程的输入结果）。被调用程序的输出就可以被调用程序使用，调用程序利用 popen 函数返回的 FILE*文件流指针，就可以通过常用的 stdio 库函数（如 fread）来读取被调用程序的输出；如果 type 是“w”，（即当前进程的输出结果作为 command 命令的输入结果）。调用程序就可以用 fwrite 向被调用程序发送数据，而被调用程序可以在自己的标准输入上读取这些数据。

pclose 等待新进程的结束，而不是杀新进程。

示例：

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

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("ps -ax", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            buffer[chars_read - 1] = '\0';
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
    }
```

```
    exit(EXIT_SUCCESS);
}
exit(EXIT_FAILURE);
}
```

4.3 守护进程

Daemon 运行在后台也称作“后台服务进程”。它是没有控制终端与之相连的进程。它独立与控制终端、通常周期的执行某种任务。那么为什么守护进程要脱离终端后台运行呢？守护进程脱离终端是为了避免进程在执行过程中的信息在任何终端上显示并且进程也不会被任何终端所产生的任何终端信息所打断。那么为什么要引入守护进程呢？由于在 linux 中，每一个系统与用户进行交流的界面称为终端，每一个从此终端开始运行的进程都会依赖这个终端，这个终端就称为这些进程的控制终端。当控制终端被关闭时，相应的进程都会自动关闭。但是守护进程却能突破这种限制，它被执行开始运转，直到整个系统关闭时才退出。几乎所有的服务器程序如 Apache 和 wu-FTP，都用 daemon 进程的形式实现。很多 Linux 下常见的命令如 inetd 和 ftpd，末尾的字母 d 通常就是指 daemon。

守护进程的特性：

1> 守护进程最重要的特性是后台运行。

2> 其次，守护进程必须与其运行前的环境隔离开来。这些环境包括未关闭的文件描述符、控制终端、会话和进程组、工作目录已经文件创建掩码等。这些环境通常是守护进程从父进程那里继承下来的。

3> 守护进程的启动方式

daemon 进程的编程规则

- 创建子进程，父进程退出：

调用 fork 产生一个子进程，同时父进程退出。我们所有后续工作都在子进程中完成。这样做我们可以交出控制台的控制权，并为子进程作为进程组长作准备；由于父进程已经先于子进程退出，会造成子进程没有父进程，变成一个孤儿进程（orphan）。每当系统发现一个孤儿进程，就会自动由 1 号进程收养它，这样，原先的子进程就会变成 1 号进程的子进程。代码如下：

```
pid = fork();
if(pid>0)
    exit(0);
```

- 在子进程中创建新会话：

使用系统函数 `setsid()`。由于创建守护进程的第一步调用了 fork 函数来创建子进程，再将父进程退出。由于在调用 fork 函数的时候，子进程全盘拷贝了父进程的会话期、进程组、控制终端等，虽然父进程退出了，但会话期、进程组、控制终端并没有改变，因此，还不是真正意义上的独立开来。而调用 setsid 函数会创建一个新的会话并自任该会话的组长，调用 setsid 函数有下面 3 个作用：让进程摆脱原会话的控制，让进程摆脱原进程组的控制，让进程摆脱原控制终端的控制；

进程组：是一个或多个进程的集合。进程组有进程组 ID 来唯一标识。除了进程号(PID)之外，进程组 ID(GID)也是一个进程的必备属性。每个进程都有一个组长进程，其组长进程的进程号等于进程组 ID。且该进程组 ID 不会因为组长进程的退出而受影响。

会话周期：会话期是一个或多个进程组的集合。通常，一个会话开始于用户登录，终止于用户退出，在此期间该用户运行的所有进程都属于这个会话期。

控制终端：由于在 linux 中，每一个系统与用户进行交流的界面称为终端，每一个从此终端开始运行的进程都会依赖这个控制终端。

- 改变当前目录为根目录：
- 使用 fork 函数创建的子进程继承了父进程的当前工作目录。由于在进程运行中，当前目录所在的文件是不能卸载的，这对以后的使用会造成很多的不便。利用 `chdir("/")` 把当前工作目录切换到根目录。
- 重设文件权限掩码：
- `umask(0)`；将文件权限掩码设为 0，Deamon 创建文件不会有太大麻烦；
- 关闭所有不需要的文件描述符：

- 新进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不会被守护进程读写，而它们一直消耗系统资源。另外守护进程已经与所属的终端失去联系，那么从终端输入的字符不可能到达守护进程，守护进程中常规方法（如 `printf`）输出的字符也不可能在终端上显示。所以通常关闭从 0 到 `MAXFILE` 的所有文件描述符。

```
for(i=0;i<MAXFILE;i++)
```

```
    close(i);
```

（注：有时还要处理 `SIGCHLD` 信号 `signal(SIGCHLD, SIG_IGN)`；防止僵尸进程（zombie））

下面就可以添加任何你要 `daemon` 做的事情

示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
void Daemon()
{
    const int MAXFD=64;
    int i=0;
    if(fork()!=0)//父进程退出
        exit(0);
    setsid();          //成为新进程组组长和新会话领导，脱离控制终端
    chdir("/"); //设置工作目录为根目录
    umask(0);          //重设文件访问权限掩码
    for(i<MAXFD;i++) //尽可能关闭所有从父进程继承来的文件
        close(i);
}
int main()
{
    Daemon(); //成为守护进程
    while(1){
        sleep(1);
    }
    return 0;
}
```

示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <syslog.h>
#include <sys/stat.h>
#include <time.h>
main()
{
    int i = 0;
    if(fork() > 0)
        exit(0);
    setsid();
    chdir("/");
```

```
umask(0);
for(; i < 64; i++)
{
    close(i);
}
i = 0;
while(i < 10)
{
    printf("%d\n",i);
    time_t ttime;
    time(&ttime);
    struct tm *pTm = gmtime(&ttime);
    syslog(LOG_INFO,"%d   %04d:%02d:%02d", i, (1900 + pTm->tm_year), (1 + pTm->tm_mon),
(pTm->tm_mday));
    i++;
    sleep(2);
}
}
```

通过查看 vi /var/log/messages

第5章 Linux 信号处理

5.1 信号概念

信号是进程在运行过程中,由自身产生或由进程外部发过来的消息(事件)。信号是硬件中断的软件模拟(软中断)。信号是 **unix** 进程通信最古老的方法。信号可以直接进行用户空间进程和内核进程之间的交互,内核进程利用它来通知用户空间进程发生了哪些系统事件。它可以在任何时候发给某一进程,而无需知道该进程的状态。如果该进程当前并未处于执行态,则该信号就由内核保存起来,直到该进程恢复执行再传递给它为止;如果一个信号被进程设置为阻塞,则该信号的传递被延迟,直到其阻塞被取消时才被传递给进程一个完整的信号生命周期可以分为 3 个重要阶段,这 3 个阶段由 4 个重要事件来刻画的:信号产生、信号在进程中注册、信号在进程中注销、执行信号处理函数

信号的生命周期如图 5.1 所示:

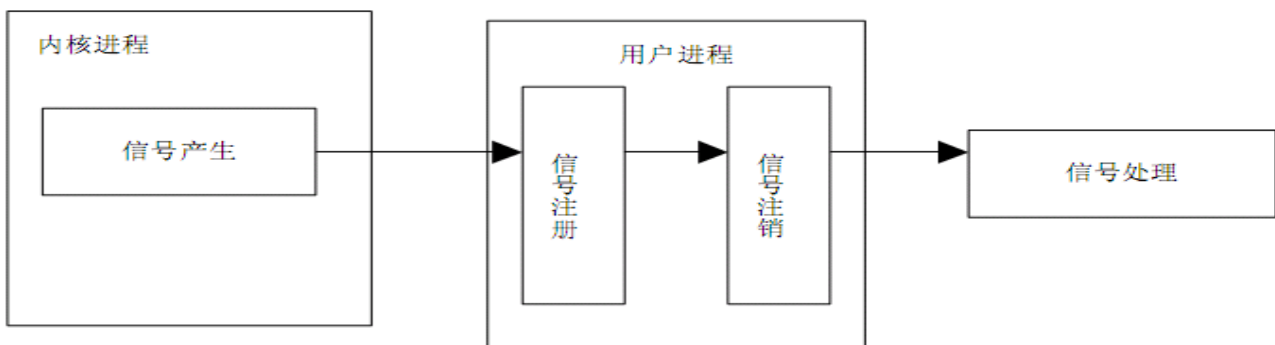


图 5.1

每个信号用一个整型常量宏表示,以 **SIG** 开头,比如 **SIGCHLD**、**SIGINT** 等,它们在系统头文件<signal.h>中定义,也可以通过在 **shell** 下键入 **kill -l** 查看信号列表,或者键入 **man 7 signal** 查看更详细的说明。

信号的生成来自内核,让内核生成信号的请求来自 3 个地方:

- 用户: 用户能够通过输入 **CTRL+c**、**Ctrl+**,或者是终端驱动程序分配给信号控制字符的其他任何键来请求内核产生信号;
- 内核: 当进程执行出错时,内核会给进程发送一个信号,例如非法段存取(内存访问违规)、浮点数溢出等;
- 进程: 一个进程可以通过系统调用 **kill** 给另一个进程发送信号,一个进程可以通过信号和另外一个进程进行通信。

进程接收到信号以后,可以有如下 3 种选择进行处理:

- 接收默认处理: 接收默认处理的进程通常会导致进程本身消亡。例如连接到终端的进程,用户按下 **CTRL+c**,将导致内核向进程发送一个 **SIGINT** 的信号,进程如果不对该信号做特殊的处理,系统将采用默认的方式处理该信号,即终止进程的执行;
- 忽略信号: 进程可以通过代码,显示地忽略某个信号的处理;但是某些信号是不能被忽略的;
- 捕捉信号并处理: 进程可以事先注册信号处理函数,当接收到某个信号时,由信号处理函数自动捕捉并且处理该信号。

有两个信号既不能被忽略也不能被捕捉,它们是 **SIGKILL** 和 **SIGSTOP**。即进程接收到这两个信号后,只能接受系统的默认处理,即终止进程。

常见信号: 用 **kill -l** 查看信号

5.2 signal 信号处理机制

可以用函数 `signal` 注册一个信号处理函数。原型为：

```
#include <signal.h>
typedef void (*sighandler_t)(int); //函数指针 void (*)(int a)
sighandler_t signal(int signum, sighandler_t handler);
```

`signal` 的第 1 个参数 `signum` 表示要捕捉处理的信号，第 2 个参数是个函数指针，表示要对该信号进行捕捉处理的函数，该参数也可以是 `SIG_DFL`(表示交由系统缺省处理，相当于白注册了)或 `SIG_IGN`(表示忽略掉该信号而不做任何处理)。`signal` 如果调用成功，返回该信号的处理函数的地址，否则返回 `SIG_ERR`。

`sighandler_t` 类型的形参（函数指针）指向一个信号处理函数，由 `signal` 函数注册，注册以后，在整个进程运行过程中均有效，并且对不同的信号可以注册同一个信号处理函数。该函数只有一个整型参数，表示信号值。

示例：

1、捕捉终端 `CTRL+c` 产生的 `SIGINT` 信号：

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>
void fun(int n)
{
    int i = 0;
    while(i < 5)
    {
        printf("sig_fun i = %d\n", i++);
        sleep(1);
    }
}

int main()
{
    signal(SIGINT,fun);//也可以写成 signal(2,fun)
    while(1)
    {
        printf("main\n");
        sleep(1);
    }
    return 0;
}
```

该程序运行起来以后，通过按 `CTRL+c` 将不再终止程序的运行（或者另开一个终端，然后发送消息：“`kill -s 2 进程号`”或“`kill -2 进程号`”，可以实现 `Ctrl + c` 同样的效果。因为 `CTRL+c` 产生的 `SIGINT` 信号已经由进程中注册的 `SignHandler` 函数捕捉了。所以当用户按下 `Ctrl+C` 的时候，主函数会暂停运行，执行该信号处理函数，当其执行完后，主函数再继续允许。该程序可以通过 `Ctrl+\` 终止，因为组合键 `Ctrl+\` 能够产生 `SIGQUIT` 信号，而该信号的捕捉函数尚未在程序中注册。

2、忽略掉终端 `CTRL+c` 产生的 `SIGINT` 信号：

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
```

```
#include <signal.h>

void fun(int n)
{
    int i = 0;
    while(i < 5)
    {
        printf("sig_fun i = %d\n", i++);
        sleep(1);
    }
}

int main()
{
    signal(SIGINT,SIG_IGN);//也可以写成 signal(2,fun)
    while(1)
    {
        printf("main\n");
        sleep(1);
    }
    return 0;
}
```

该程序运行起来以后，将 CTRL+C 产生的 SIGINT 信号忽略掉了，所以 CTRL+C 将不再能使该进程终止，要终止该进程，可以向进程发送 SIGQUIT 信号，即组合键 CTRL+\. 或者另外开一个端口，然后执行 ps -aux 查看进程，发现该进程号之后用 kill -9 进程号 杀掉该进程。

3、接受信号的默认处理,接受默认处理就相当于没有写信号处理程序:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>

void fun(int n)
{
    int i = 0;
    while(i < 5)
    {
        printf("sig_fun i = %d\n", i++);
        sleep(1);
    }
}

int main()
{
    signal(SIGINT,SIG_IGN);//也可以写成 signal(2,fun)
    while(1)
    {
        printf("main\n");
        sleep(1);
    }
}
```

```
    return 0;
}
```

5.3 sigaction 信号处理机制

5.3.1 信号处理情况分析

在 signal 处理机制下，还有许多特殊情况需要考虑：

- 注册一个信号处理函数，并且处理完毕一个信号之后，是否需要重新注册，才能够捕捉下一个信号；（不需要）
- 如果信号处理函数正在处理信号，并且还没有处理完毕时，又发生了一个同类型的信号，这时该怎么处理；（挨着执行）
- 如果信号处理函数正在处理信号，并且还没有处理完毕时，又发生了一个不同类型的信号，这时该怎么处理；（跳转去执行另一个信号，之后再执行剩下的没有处理完的信号）
- 如果程序阻塞在一个系统调用(如 read(...))时，发生了一个信号，这时是让系统调用返回错误再接着进入信号处理函数，还是先跳转到信号处理函数，等信号处理完毕后，系统调用再返回。

示例：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
int g_iSeq = 0;

void SignHandler(int iSignNo)
{
    int iSeq = g_iSeq++;
    printf("%d Enter SignHandler, signo:%d\n", iSeq, iSignNo);
    sleep(3);
    printf("%d Leave SignHandler, signo:%d\n", iSeq, iSignNo);
}

int main()
{
    char szBuf[8];
    int iRet;
    signal(SIGINT, SignHandler);      //不同的信号调用同一个处理函数
    signal(SIGQUIT, SignHandler);
    do{
        iRet = read(STDIN_FILENO, szBuf, sizeof(szBuf)-1);
        if(iRet < 0){
            perror("read fail.");
            break;
        }
        szBuf[iRet] = 0;
        printf("Get: %s", szBuf);
    }while(strcmp(szBuf, "quit\n") != 0);
    return 0;
}
```

}

程序运行时，针对于如下几种输入情况(要输入得快)，看输出结果：

- [CTRL+c] [CTRL+c] (一个一个挨着执行)
- [CTRL+c] [CTRL+] (先执行 c 的进入，被\打断，转而执行\，最后执行 c 的退出)
- hello [CTRL+] [Enter] (先执行中断，没有任何输出)
- [CTRL+] hello [Enter] (先执行中断，输出内容)
- hel [CTRL+] lo[Enter] (先执行中断，只输出 lo)

5.3.2 sigaction 信号处理注册

函数原型：

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

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

sigaction 也用于注册一个信号处理函数

参数 signum 为需要捕捉的信号

参数 act 是一个结构体，里面包含信号处理函数地址、处理方式等信息

参数 oldact 是一个传出参数，sigaction 函数调用成功后，oldact 里面包含以前对 signum 的处理方式的信息，通常为 NULL

如果函数调用成功，将返回 0，否则返回-1

结构体 struct sigaction(注意名称与函数 sigaction 相同)的原型为：

```
struct sigaction {
    void (*sa_handler)(int);    //老类型的信号处理函数指针
    void (*sa_sigaction)(int, siginfo_t *, void *); //新类型的信号处理函数指针
    sigset_t sa_mask;          //将要被阻塞的信号集合
    int sa_flags;               //信号处理方式掩码 (→ SA_SIGINFO)
    void (*sa_restorer)(void); //保留，不要使用
};
```

该结构体的各字段含义及使用方式：

1、字段 sa_handler 是一个函数指针，用于指向原型为 void handler(int)的信号处理函数地址，即老类型的信号处理函数（如果用这个再将 sa_flags = 0,就等同于 signal()函数）

2、字段 sa_sigaction 也是一个函数指针，用于指向原型为：

```
void handler(int iSignNum, siginfo_t *pSignInfo, void *pReserved);
```

的信号处理函数，即新类型的信号处理函数

该函数的三个参数含义为：

iSignNum: 传入的信号

pSignInfo: 与该信号相关的一些信息，它是个结构体

pReserved: 保留，现没用，通常为 NULL

3、字段 sa_handler 和 sa_sigaction 只应该有一个生效，如果想采用老的信号处理机制，就应该让 sa_handler 指向正确的信号处理函数，并且让字段 sa_flags 为 0；否则应该让 sa_sigaction 指向正确的信号处理函数，并且让字段 sa_flags 包含 SA_SIGINFO 选项

4、字段 sa_mask 是一个包含信号集合的结构体，该结构体内的信号表示在进行信号处理时，将要被阻塞的信号。针对 sigset_t 结构体，有一组专门的函数对它进行处理，它们是：

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

```
int sigemptyset(sigset_t * set); //清空信号集合 set
```

```
int sigfillset(sigset_t * set); //将所有信号填充进 set 中
```

```
int sigaddset(sigset_t * set, int signum); //往 set 中添加信号 signum
```

```
int sigdelset(sigset_t * set, int signum); //从 set 中移除信号 signum
```

```
int sigismember(const sigset_t * set, int signum); //判断 signum 是不是包含在 set 中
```

int sigpending(sigset_t * set); //将被阻塞的信号集合由参数 set 指针返回 例如，如果打算在处理信号 SIGINT 时，只阻塞对 SIGQUIT 信号的处理，可以用如下方法：

```
struct sigaction act;
act.sa_flags = SA_SIGINFO;
act.sa_sigaction = newHandler;
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask, SIGQUIT);
sigaction(SIGINT, &act, NULL);
```

6、 字段 sa_flags 是一组掩码的合成值，指示信号处理时所应该采取的一些行为，各掩码的含义如表 5.1 所示：

表 5.1

掩码	描述
SA_RESETHAND	处理完毕要捕捉的信号后，将自动撤消信号处理函数的注册，即必须再重新注册信号处理函数，才能继续处理接下来产生的信号。该选项不符合一般的信号处理流程，现已经被废弃。
SA_NODEFER	在处理信号时，如果又发生了其它的信号，则立即进入其它信号的处理，等其它信号处理完毕后，再继续处理当前的信号，即递规地处理。如果 sa_flags 包含了该掩码，则结构体 sigaction 的 sa_mask 将无效！（不常用）
SA_RESTART	如果在发生信号时，程序正阻塞在某个系统调用，例如调用 read() 函数，则在处理完毕信号后，接着从阻塞的系统返回。如果不指定该参数，中断处理完毕之后，read 函数读取失败。
SA_SIGINFO	指示结构体的信号处理函数指针是哪个有效，如果 sa_flags 包含该掩码，则 sa_sigaction 指针有效，否则是 sa_handler 指针有效。（常用）

示例 1： 用 sigaction 实现和 signal（只能传递一个参数）一样的功能。

```
#include<signal.h>
#include<stdio.h>
void handle(int signo)
{
    printf("signo: %d\n",signo);
}
main()
{
    struct sigaction st;
    st.sa_handler = handle;
    st.sa_flags = 0;
    sigaction(SIGINT,&st,NULL);
    while(1)
    {
        sleep(1);
    }
}
```

示例 2： 用 sigaction 实现调用新的信号处理函数

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

```
#include <signal.h>
int g_iSeq = 0;
void SignHandlerNew(int iSignNo, siginfo_t *pInfo, void *pReserved)
{
    int iSeq = g_iSeq++;
    printf("%d Enter SignHandlerNew, signo: %d\n", iSeq, iSignNo);
    sleep(3);
    printf("%d Leave SignHandlerNew, signo: %d\n", iSeq, iSignNo);
}
int main()
{
    struct sigaction act;
    act.sa_sigaction = SignHandlerNew;
    act.sa_flags = SA_SIGINFO;
    sigaction(SIGINT, &act, NULL);
    sigaction(SIGQUIT, &act, NULL);
    while(1)
    {
        sleep(1);
    }
    return 0;
}
```

练习与验证:

针对于先前的 5 种输入情况, 给下面代码再添加一些代码, 使之能够进行如下各种形式的响应:

- 1、[CTRL+c] [CTRL+c]时, 第 1 个信号处理阻塞同类型第 2 个信号处理;
- 2、[CTRL+c] [CTRL+c]时, 第 1 个信号处理时, 允许递规地同类型第 2 个信号处理;
- 3、[CTRL+c] [CTRL+]时, 第 1 个信号阻塞不同类型的第 2 个信号处理;
- 4、read 不要因为信号处理而返回失败结果。

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

int g_iSeq = 0;

void SignHandlerNew(int iSignNo, siginfo_t *pInfo, void *pReserved)
{
    int iSeq = g_iSeq++;
    printf("%d Enter SignHandlerNew, signo: %d\n", iSeq, iSignNo);
    sleep(3);
    printf("%d Leave SignHandlerNew, signo: %d\n", iSeq, iSignNo);
}

int main()
{
    char szBuf[8] = {0};
    int iRet = 0;
    struct sigaction act;
```

```
act.sa_sigaction = SignHandlerNew;
act.sa_flags = SA_SIGINFO | SA_RESTART;
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask, SIGQUIT);
sigaction(SIGINT,&act,NULL);
do{
    iRet = read(STDIN_FILENO,szBuf,sizeof(szBuf)-1);
    if(iRet < 0){
        perror("read fail");
        break;
    }
    szBuf[iRet] = 0;
    printf("Get: %s",szBuf);
}while(strcmp(szBuf,"quit\n") != 0);
return 0;
}
```

5.3.3 sigprocmask 信号阻塞

函数 sigaction 中设置的被阻塞信号集合只是针对于要处理的信号，例如

```
struct sigaction act;
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask,SIGQUIT);
sigaction(SIGINT,&act,NULL);
```

表示只有在处理信号 SIGINT 时，才阻塞信号 SIGQUIT；

函数 sigprocmask 是全程阻塞，在 sigprocmask 中设置了阻塞集合后，被阻塞的信号将不能再被信号处理函数捕捉，直到重新设置阻塞信号集合。

原型为：

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

参数 how 的值为如下 3 者之一：

- a: SIG_BLOCK,将参数 2 的信号集合添加到进程原有的阻塞信号集合中
- b: SIG_UNBLOCK,从进程原有的阻塞信号集合移除参数 2 中包含的信号
- c: SIG_SETMASK, 重新设置进程的阻塞信号集为参数 2 的信号集

参数 set 为阻塞信号集

参数 oldset 是传出参数，存放进程原有的信号集，通常为 NULL

示例：添加全程阻塞

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

```
int g_iSeq=0;
```

```
void SignHandlerNew(int iSignNo,siginfo_t *pInfo,void *pReserved)
{
    int iSeq=g_iSeq++;
    printf("%d Enter SignHandlerNew,signo:%d\n",iSeq,iSignNo);
```



```
    sleep(3);
    printf("%d Leave SignHandlerNew,signo:%d\n",iSeq,iSignNo);
}

int main()
{
    char szBuf[8];
    int iRet;

    //屏蔽掉 SIGINT 信号，SigHandlerNew 将不能再捕捉 SIGINT
    sigset_t sigSet;
    sigemptyset(&sigSet);
    sigaddset(&sigSet,SIGINT);
    sigaddset(&sigSet,SIGQUIT);
    sigprocmask(SIG_BLOCK,&sigSet,NULL);//将 SIGINT、SIGQUIT 屏蔽

    struct sigaction act;
    act.sa_sigaction=SignHandlerNew;
    act.sa_flags=SA_SIGINFO;
    sigemptyset(&act.sa_mask);
    sigaction(SIGINT,&act,NULL);
    sigaction(SIGQUIT,&act,NULL);
    do{
        iRet=read(STDIN_FILENO,szBuf,sizeof(szBuf)-1);
        if(iRet<0){
            perror("read fail.");
            break;
        }
        szBuf[iRet]=0;
        printf("Get: %s",szBuf);
    }while(strcmp(szBuf,"quit\n")!=0);
    return 0;
}
```

示例：取消指定信号的全程阻塞。

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
int g_iSeq=0;
void SignHandlerNew(int iSignNo,signinfo_t *pInfo,void *pReserved)
{
    int iSeq=g_iSeq++;
    printf("%d Enter SignHandlerNew,signo:%d\n",iSeq,iSignNo);
    sleep(3);
    printf("%d Leave SignHandlerNew,signo:%d\n",iSeq,iSignNo);
}
```

```
int main()
{
```

//屏蔽掉 SIGINT 和 SIGQUIT 信号, SigHandlerNew 将不能再捕捉 SIGINT 和 SIGQUIT

```
sigset_t sigSet;
sigemptyset(&sigSet);
sigaddset(&sigSet, SIGINT);
sigaddset(&sigSet, SIGQUIT);
sigprocmask(SIG_BLOCK, &sigSet, NULL); //将 SIGINT、SIGQUIT 屏蔽
```

```
struct sigaction act;
act.sa_sigaction=SigHandlerNew;
act.sa_flags=SA_SIGINFO;
sigemptyset(&act.sa_mask);
sigaction(SIGINT, &act, NULL);
sigaction(SIGQUIT, &act, NULL);
int iCount = 0;
while(1)
{
    if(iCount > 3)
    {
        sigset_t sigSet2;
        sigemptyset(&sigSet2);
        sigaddset(&sigSet2, SIGINT);
        sigprocmask(SIG_UNBLOCK, &sigSet2, NULL);
    }
    iCount ++;
    sleep(2);
}
return 0;
}
```

5.4 用程序发送信号

5.4.1 kill 和 raise 信号发送函数

原型为:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

```
#include <signal.h>
int raise(int sig)
```

kill 函数向指定进程发送指定信号。raise 函数则允许进程向自身发送信号。

参数 pid 为将要接受信号的进程的 pid, 可以通过 getpid()函数获得

参数 sig 为要发送的信号

如果成功, 返回 0, 否则为-1

示例: 输入结束后, 将通过发送信号 SIGQUIT 把自己杀掉:

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

```
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>
void fun(int n)
{
    printf("aaaa n = %d\n",n);
}

int main()
{
    signal(3,fun);
    while(1)
    {
        printf("hello\n");
        raise(3);
        //kill(getpid(), 3);//getpid()是获取本进程的 pid，等价于 raise(3);
        sleep(2);
    }
    return 0;
}
```

5.4.2 sigqueue 信号发送函数

sigqueue 也可以发送信号，并且能传递附加的信息。

原型为：

```
#include <signal.h>
int sigqueue(pid_t pid, int sig, const union sigval value);
```

参数 pid 为接收信号的进程；

参数 sig 为要发送的信号；

参数 value 为一整型与指针类型的共用体：

```
union sigval {
    int sival_int; //可以传递整型
    void * sival_ptr; //其他类型
};
```

由 sigqueue 函数发送的信号的第 3 个参数 sival_ptr 的值，可以被进程的信号处理函数的第 2 个参数 info->si_ptr 接收到、整型的 sival_int 可以被信号处理函数的第二个参数 info->si_int 接收。

示例 1：进程给自己发信号,并且带上附加信息：

```
#include <signal.h>
#include <sys / types.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

void SignHandlerNew(int signum, siginfo_t * info, void * myact)
{
    char * pszInfo = (char *) (info->si_ptr);
    printf("Get:%d info:%s\n", signum, pszInfo);
}
```

```
}

int main(int argc, char ** argv)
{
    if (argc < 2){
        printf("usage: SIGNUM\n");
        return -1;
    }
    int sig = atoi(argv[1]); //通过命令行传入信号值 (eg: 不能写成 SIGINT, 要写成 2)
    struct sigaction act; //1. 定义结构体 sigaction
    sigemptyset(&act.sa_mask); //2. 将 sa_mask 置空
    act.sa_sigaction = SignHandlerNew; //3. 信号处理函数
    act.sa_flags = SA_SIGINFO; //4. 令 sa_flags = SA_SIGINFO;
    sigaction(sig, &act, NULL); //5. 调用 sigaction 函数
    union sigval mysigval; char data[] = "other info";
    mysigval.sival_ptr = data;
    while (1){
        printf("wait for the signal\n"); sigqueue(getpid(), sig, mysigval);
        sleep(2);
    }
}
```

下面为示例 2：一个进程向另外一个进程发送信号。注意：发送进程不要将自己进程空间的地址发送给接收进程，因为接收进程接收到地址也访问不到发送进程的地址空间的。要用 `ps -aux` 查看接收端的进程号，然后发送端向该进程号发送数据。（注意：经过验证，发现在不同进程之间利用 `sigqueue` 传递数据时，只能传递整型值，传递字符串失败。）

示例 2： 信号接收程序：receive.c

```
#include <signal.h>
#include <sys / types.h>
#include <unistd.h>
#include <stdio.h>

void new_op(int, siginfo_t *, void *);

int main(int argc, char ** argv)
{
    pid_t pid = getpid();
    int sig = atoi(argv[1]); //可以直接用 SIGINT
    struct sigaction act;
    sigemptyset(&act.sa_mask);
    act.sa_sigaction = new_op;
    act.sa_flags = SA_SIGINFO;
    if (sigaction(sig, &act, NULL) < 0)
    {
        printf("install sigal error\n");
    }
    while (1)
    {
        sleep(2);
        printf("wait for the signal\n");
    }
}
```

```
}  
void new_op(int signum, siginfo_t * info, void * myact)  
{  
    printf("the int value is %d \n", info->si_int);  
}
```

示例 2: 信号发送程序: send.c

```
#include <signal.h>  
#include <sys / time.h>  
#include <unistd.h>  
#include <sys / types.h>  
#include <stdio.h>  
main(int argc, char ** argv)  
{  
    union sigval mysigval;  
    int signum = atoi(argv[1]); //表示信号值  
    pid_t pid = (pid_t)atoi(argv[2]); //接收端的进程号  
    mysigval.sival_int = 10; //不代表具体含义, 只用于说明问题  
    if (sigqueue(pid, signum, mysigval) == -1)  
        printf("send error\n");  
    sleep(2);  
}
```

→ 首先执行 ./receive 2 然后用 ps -aux 查看其进程号 假设为 11250
再执行 ./send 2 11250 即可在接收端收到数据

5.5 计时器与信号

5.5.1 睡眠函数

Linux 下有两个睡眠函数, 原型为:

```
#include <unistd.h>  
unsigned int sleep(unsigned int seconds);  
void usleep(unsigned long usec);
```

函数 sleep 让进程睡眠 seconds 秒, 函数 usleep 让进程睡眠 usec 微秒。

sleep 睡眠函数内部是用信号机制进行处理的, 用到的函数有:

```
#include <unistd.h>  
unsigned int alarm(unsigned int seconds);
```

//告知自身进程, 要进程在 seconds 秒后自动产生一个 SIGALRM 的信号

```
int pause(void); //将自身进程挂起, 直到有信号发生时才从 pause 返回
```

示例: 模拟睡眠 3 秒:

```
#include <unistd.h>  
#include <stdio.h>  
#include <signal.h>  
void fun(int n) //空函数, 用于对 SIGALRM 信号进行捕捉处理。  
{  
}  
void my_sleep(int n)  
{  
    signal(SIGALRM, fun);
```

```
alarm(n);
pause();
}
int main()
{
    printf("aaaa\n");
    my_sleep(3);
    printf("bbbb\n");
    return 0;
}
```

注意：因为 sleep 在内部是用 alarm 实现的，所以在程序中最好不要 sleep 与 alarm 混用，以免造成混乱。

5.5.2 时钟处理

Linux 为每个进程维护 3 个计时器，分别是真实计时器、虚拟计时器和实用计时器。

- 真实计时器计算的是程序运行的实际时间；
- 虚拟计时器计算的是程序运行在用户态时所消耗的时间(可认为是实际时间减掉(系统调用和程序睡眠所消耗)的时间)；
- 实用计时器计算的是程序处于用户态和处于内核态所消耗的时间之和。

例如：有一程序运行，在用户态运行了 5 秒，在内核态运行了 6 秒，还睡眠了 7 秒，则真实计算器计算的结果是 18 秒，虚拟计时器计算的是 5 秒，实用计时器计算的是 11 秒。

用指定的初始间隔和重复间隔时间为进程设定好一个计时器后，该计时器就会定时地向进程发送时钟信号。

3 个计时器发送的时钟信号分别为：SIGALRM、SIGVTALRM 和 SIGPROF。

用到的函数与数据结构：

```
#include <sys/time.h>
int getitimer(int which, struct itimerval *value);    //获取计时器的设置
```

参数 which 指定哪个计时器，可选项为 ITIMER_REAL(真实计时器)、ITIMER_VIRTUAL(虚拟计时器、ITIMER_PROF(实用计时器))

参数 value 为一结构体的传出参数，用于传出该计时器的初始间隔时间和重复间隔时间

返回值：如果成功，返回 0，否则-1

```
int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue); //设置计时器
```

参数 which 指定哪个计时器，可选项为 ITIMER_REAL(真实计时器)、ITIMER_VIRTUAL(虚拟计时器、ITIMER_PROF(实用计时器))

参数 value 为一结构体的传入参数，指定该计时器的初始间隔时间和重复间隔时间

参数 ovalue 为一结构体传出参数，用于传出以前的计时器时间设置。

返回值：如果成功，返回 0，否则-1

```
struct itimerval {
    struct timeval it_interval; /* next value */    //重复间隔
    struct timeval it_value; /* current value */    //初始间隔
};
struct timeval {
    long tv_sec; /* seconds */    //时间的秒数部分
    long tv_usec; /* microseconds */    //时间的微秒部分
};
```

示例：启用真实计时器的进行时钟处理（获得当前系统时间，并一秒更新一次）

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

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

void sigHandler(int iSigNum)
{
    time_t tt;
    time(&tt);
    struct tm *pTm = gmtime(&tt);
    printf("%04d-%02d-%02d  %02d:%02d:%02d\n", (1900+pTm->tm_year), (1+pTm->tm_mon),
pTm->tm_mday, (8+pTm->tm_hour), pTm->tm_min, pTm->tm_sec);
}

void InitTime(int tv_sec, int tv_usec)
{
    signal(SIGALRM, sigHandler);

    struct itimerval tm;
    tm.it_value.tv_sec = tv_sec;
    tm.it_value.tv_usec = tv_usec;

    tm.it_interval.tv_sec = tv_sec;
    tm.it_interval.tv_usec = tv_usec;

    if(setitimer(ITIMER_REAL, &tm, NULL) == -1)
    {
        perror("setitimer error");
        exit(-1);
    }
}

int main()
{
    InitTime(1, 0);
    while(1);
    return 0;
}
```


第6章 进程间通信

6.1 (Interprocess Communication, IPC)简介

Linux 下的进程通信手段基本上是从 UNIX 平台上的进程通信手段继承而来的。而对 UNIX 发展做出重大贡献的两大主力 AT&T 的贝尔实验室及 BSD（加州大学伯克利分校的伯克利软件发布中心）在进程间的通信方面的侧重点有所不同。前者是对 UNIX 早期的进程间通信手段进行了系统的改进和扩充，形成了“system V IPC”，其通信进程主要局限在单个计算机内；后者则跳过了该限制，形成了基于套接口（socket）的进程间通信机制。而 Linux 则把两者的优势都继承了下来，如图 6.1 所示。

UNIX 进程间通信（IPC）方式包括管道、FIFO 以及信号。

System V 进程间通信（IPC）包括 System V 消息队列、System V 信号量以及 System V 共享内存区。

Posix 进程间通信（IPC）包括 Posix 消息队列、Posix 信号量以及 Posix 共享内存区

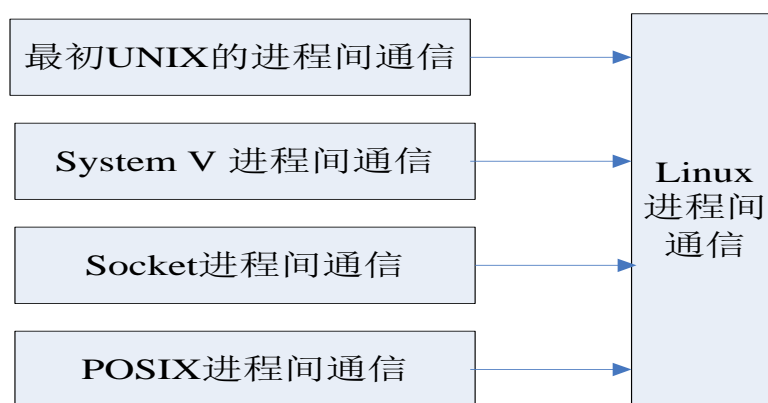


图 6.1

linux 进程之间的通讯主要有下面几种：

- 1 管道 pipe 和命名管道：管道有亲缘关系进程间的通信，命名管道还允许无亲缘关系进程间通信
- 2 信号 signal：在软件层模拟中断机制，通知进程某事发生
- 3 消息队列：消息的链表包括 posix 消息队列和 SystemV 消息队列
- 4 共享内存：多个进程访问一块内存主要以同步
- 5 信号量：进程间同步
- 6 套接字 socket：不同机器间进程通信

下面是对它们的详解：

（1）管道（Pipe）及有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，有名管道，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。

（2）信号（Signal）：信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一样的。

（3）消息队列（Message Queue）：消息队列是消息的链接表，包括 Posix 消息队列和 SystemV 消息队列。它克服了前两种通信方式中信息量有限的缺点，具有写权限的进程可以按照一定的规则向消息队列中添加新消息；对消息队列有读权限的进程则可以从消息队列中读取消息。

（4）共享内存（Shared memory）：可以说是最有用的进程间通信方式，是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种通信方式需要依靠某种同步机制，如互斥锁和信号量等。

（5）信号量（Semaphore）：主要作为进程之间以及同一进程的不同线程之间的同步和互斥手段。

（6）套接字（Socket）：这是一种更为一般的进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。

6.2 标准流管道

像文件操作有标准 io 流一样，管道也支持文件流模式。用来创建连接到另一进程的管道 `popen` 和 `pclose`。
函数原型：

```
#include <stdio.h>
FILE* popen(const char* command, const char* open_mode);
int pclose(FILE* fp);
```

函数 `popen()`：允许一个程序将另一个程序作为新进程来启动，并可以传递数据给它或者通过它接收数据。
`command` 字符串是要运行的程序名。`open_mode` 必须是“r”或“w”。如果 `open_mode` 是“r”，被调用程序的输出就可以被调用程序使用，调用程序利用 `popen` 函数返回的 `FILE*` 文件流指针，就可以通过常用的 `stdio` 库函数（如 `fread`）来读取被调用程序的输出；如果 `open_mode` 是“w”，调用程序就可以用 `fwrite` 向被调用程序发送数据，而被调用程序可以在自己的标准输入上读取这些数据。

函数 `pclose()`：用 `popen` 启动的进程结束时，我们可以用 `pclose` 函数关闭与之关联的文件流。

示例 1：从标准管道流中读取 打印/etc/profile 的内容

```
#include <stdio.h>
int main()
{
    FILE* fp = popen("cat /etc/profile", "r");
    char buf[512] = {0};
    while(fgets(buf, sizeof(buf), fp))
    {
        puts(buf);
    }
    pclose(fp);
    return 0;
}
```

示例 2：写到标准管道流 统计 buf 单词数

```
#include <stdio.h>
int main()
{
    char buf[] = {"aaa bbb ccc ddd eee fff ggg hhh"};
    FILE *fp = popen("wc -w", "w");
    fwrite(buf, sizeof(buf), 1, fp);
    pclose(fp);
    return 0;
}
```

6.3 无名管道(PIPE)

管道是 linux 进程间通信的一种方式，如命令 `ps -ef | grep ntp`

无名管道的特点：

- 1 只能在亲缘关系进程间通信（父子或兄弟）
- 2 半双工（固定的读端和固定的写端）
- 3 他是特殊的文件，可以用 `read`、`write` 等，只能在内存中

管道函数原型：

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

管道好比一条水管，有两个端口，一端进水，另一端出水。管道也有两个端口，分别是读端和写端，进水可看成数据从写端被写入，出水可看数据从读端被读出。我们分别用 `read`、`write` 函数来对管道的读端和写端进行读写，所以必须要知道读写两端分别对应的文件描述符。这两个文件描述符我们通常保存在一个有两个整型元素的数组中，如 `int fds[2]`；然后调用函数 `pipe(fds)`，这个函数会创建一个管道，并且数组 `fds` 中的两个元素会成为管道读端和写端对应的两个文件描述符。即 `fds[0]` 和读端相对应，`fds[1]` 和写端相对应。**`fds[0]` 有可读属性，`fds[1]` 有可写的属性。**

函数 `pipe` 用于创建一个无名管道，如果成功，`fds[0]` 存放可读的文件描述符，`fds[1]` 存放可写文件描述符，并且函数返回 0，否则返回 -1。

通过调用 `pipe` 函数获取这对打开的文件描述符后，一个进程就可以从 `fds[0]` 中读数据，而另一个进程就可以往 `fds[1]` 中写数据。当然两进程间必须有继承关系，才能继承这对打开的文件描述符。其模型如图 6.2 所示。

管道不象真正的物理文件，不是持久的，即两进程终止后，管道也自动消失了。

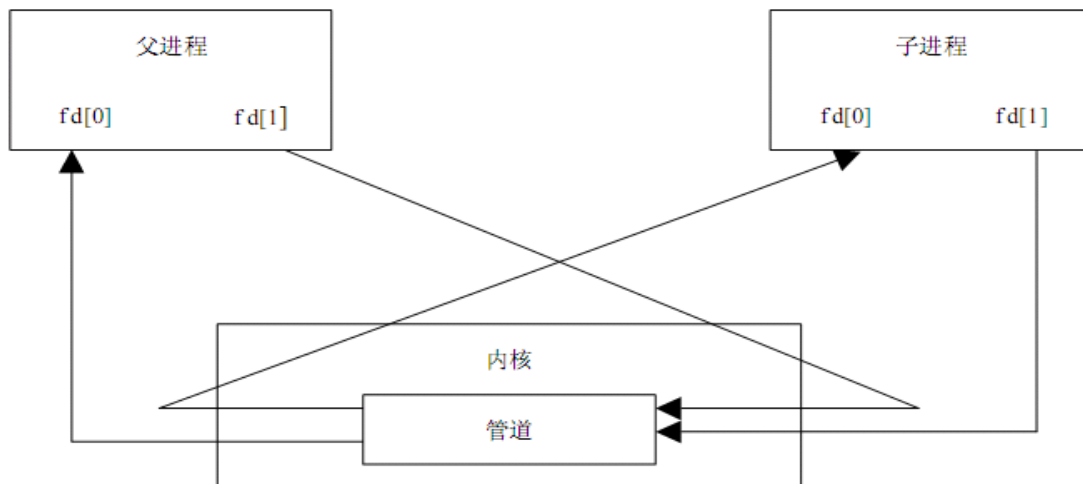


图 6.2

示例：创建父子进程，创建无名管道，父写子读

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
int main()
{
    int fds[2] = {0};
    pipe(fds);
    char buf[32];
    if(fork() == 0)
    {
        sleep(2);
        read(fds[0], buf, sizeof(buf)); //从读端读数据
        puts(buf);
        close(fds[0]);
        close(fds[1]);
    }
    else
    {
        write(fds[1], "hello", 6); //向写端写数据
        waitpid(-1, NULL, 0); //等子退出
        close(fds[0]);
        close(fds[1]);
    }
}
```

```
    }  
    return 0;  
}
```

管道两端的关闭是有先后顺序的，如果先关闭写端则从另一端读数据时，`read` 函数将返回 0，表示管道已经关闭；但是如果先关闭读端，则从另一端写数据时，将会使写数据的进程接收到 `SIGPIPE` 信号，如果写进程不对该信号进行处理，将导致写进程终止，如果写进程处理了该信号，则写数据的 `write` 函数返回一个负值，表示管道已经关闭。

示例：

```
#include <stdio.h>  
#include <string.h>  
#include <unistd.h>  
void fun(int n)  
{  
    printf("receive a signal:SIGPIPE\n");  
}  
int main()  
{  
    int fds[2];  
    pipe(fds);  
    signal(SIGPIPE, fun);  
    char buf[10];  
    if(fork() == 0)  
    {  
        sleep(2); //确保父关闭读  
        read(fds[0], buf, sizeof(buf));  
        puts(buf);  
        close(fds[0]); //子关闭读  
        close(fds[1]); //子关闭写  
    }  
    else  
    {  
        close(fds[0]); //父关闭读  
        write(fds[1], "hello", 6);  
        wait(NULL);  
        write(fds[1], "world", 6); //管道读端已全关闭，父还在写，将产生 SIGPIPE 信号  
        close(fds[1]); //父关闭写  
    }  
    return 0;  
}
```

6.4 命名管道(FIFO)

无名管道只能在亲缘关系的进程间通信大大限制了管道的使用，有名管道突破了这个限制，通过指定路径名的形式实现不相关进程间的通信

6.4.1 创建、删除 FIFO 文件

创建 FIFO 文件与创建普通文件很类似，只是创建后的文件用于 FIFO。

1. 用函数创建和删除 FIFO 文件

创建 FIFO 文件的函数原型：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

参数 `pathname` 为要创建的 FIFO 文件的全路径名；

参数 `mode` 为文件访问权限，比如 0666

如果创建成功，则返回 0，否则-1。

删除 FIFO 文件的函数原型为：

```
#include <unistd.h>
int unlink(const char *pathname);
```

示例：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])//演示通过命令行传递参数
{
    if(argc != 2)
    {
        puts("Usage: MkFifo.exe {filename}");
        return -1;
    }
    if(mkfifo(argv[1], 0666) == -1)
    {
        perror("mkfifo fail");
        return -2;
    }
    //unlink(argv[1]);//加上这句会将创建的 FIFO 文件删除。
    return 0;
}
```

2. 用命令创建和删除 FIFO 文件

用命令 `mkfifo` 创建 不能重复创建

用命令 `unlink` 删除

创建完毕之后，就可以访问 FIFO 文件了：

一个终端：`cat < myfifo`

另一个终端：`echo "hello"> myfifo`

6.4.2 打开、关闭 FIFO 文件

对 FIFO 类型的文件的打开/关闭跟普通文件一样，都是使用 `open` 和 `close` 函数。如果打开时使用 `O_WRONLY` 选项，则打开 FIFO 的写入端，如果使用 `O_RDONLY` 选项，则打开 FIFO 的读取端，写入端和读取端都可以被几个进程同时打开。

该管道可以通过路径名来指出，并且在文件系统中是可见的。在建立了管道之后，两个进程就可以把它当作普通文件一样进行读写操作，使用非常方便。不过值得注意的是，FIFO 是严格地遵循先进先出规则的，对管道及 FIFO 的读总是从开始处返回数据，对它们的写则把数据添加到末尾，它们不支持如 `lseek()` 等文件定位操作，必须按照从前到后的顺序依次读写。

如果以读取方式打开 FIFO，并且还没有其它进程以写入方式打开 FIFO，`open` 函数将被阻塞；同样，如果以写入方式打开 FIFO，并且还没其它进程以读取方式 FIFO，`open` 函数也将被阻塞。但是，如果 `open` 函数中包含

O_NONBLOCK 选项，则上述两种情况下调用 open 都不被阻塞。

与 PIPE 相同，关闭 FIFO 时，如果先关读取端，将导致继续往 FIFO 中写数据的进程接收 SIG_PIPE 的信号。

6.4.3 读写 FIFO

可以采用与普通文件相同的读写方式读写 FIFO。

示例：先执行 #mkfifo f.fifo 命令

然后 vi write.c 如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int main()
{
    int fd = open("f.fifo", O_WRONLY);    //1. 打开（判断是否成功打开略）
    write(fd, "hello", 6);                //2. 写
    close(fd);                            //3. 关闭
    return 0;
}
```

然后 vi read.c 如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int main()
{
    char buf[128];
    int fd = open("f.fifo", O_RDONLY);
    read(fd, buf, sizeof(buf));
    puts(buf);
    close(fd);
    return 0;
}
```

然后 gcc -o write write.c

gcc -o read read.c

./write //发现阻塞，要等待执行 ./read

./read

在屏幕上输出 hello

6.5 内存映射

内存映射函数：mmap munmap msync

函数原型：

```
#include <unistd.h>
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
int msync(const void *start, size_t length, int flags);    //把对内存区域所做的更改同步到文件
int munmap(void *start, size_t length);
```

mmap(): 用来将某个文件内容映射到内存中, 对该内存区域的存取即是直接对该文件内容的读写。通过这样可以加快文件访问速度。返回值: 成功, 返回映射的内存的起始地址。

第一个参数 **start** 指向欲对应的内存起始地址, 通常设为 **NULL**, 代表让系统自动选定地址, 对应成功后该地址会返回。

第二个参数 **length** 代表将文件中多大的部分对应到内存。

第三个参数 **prot** 代表映射区域的保护方式有下列组合:

PROT_EXEC 映射区域可被执行

PROT_READ 映射区域可被读取

PROT_WRITE 映射区域可被写入

PROT_NONE 映射区域不能存取

第四个参数 **flags** 会影响映射区域的各种特性:

MAP_FIXED 如果参数 **start** 所指的地址无法成功建立映射时, 则放弃映射, 不对地址做修正。通常不鼓励用此旗标。

MAP_SHARED 对映射区域的写入数据会复制回文件内, 而且允许其他映射该文件的进程共享。

MAP_PRIVATE 对映射区域的写入操作会产生一个映射文件的复制, 即私人的“写入时复制”(copy on write) 对此区域作的任何修改都不会写回原来的文件内容。

MAP_ANONYMOUS 建立匿名映射。此时会忽略参数 **fd**, 不涉及文件, 而且映射区域无法和其他进程共享。

MAP_DENYWRITE 只允许对映射区域的写入操作, 而不能对 **fd** 指向的文件进行读写, 对该文件直接写入的操作将会被拒绝。

MAP_LOCKED 将映射区域锁定住, 这表示该区域不会被置换 (swap)。

在调用 **mmap()** 时必须指定 **MAP_SHARED** 或 **MAP_PRIVATE**。

第五个参数 **fd** 为 **open()** 返回的文件描述词, 代表欲映射到内存的文件。

第六个参数 **offset** 为文件映射的偏移量, 通常设置为 0, 代表从文件最前方开始对应, **offset** 必须是分页大小的整数倍。

munmap(): 用来取消参数 **start** 所指的映射内存起始地址, 参数 **length** 则是欲取消的内存大小。当进程结束, 映射内存会自动解除, 但关闭对应的文件描述词时不会解除映射。返回值: 如果解除映射成功则返回 0, 否则返回 -1

示例 1:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
main()
{
    int i;
    int fd = open("mm.dat", O_RDWR | O_CREAT);
    char buf[10] = {"abcdefghi"};
    write(fd, buf, sizeof(buf));
    char *p = (char *)mmap(NULL, 10, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    p[2] = 'C';
    p[3] = 'D';
    for(i = 0; i <= 9; i++)
        printf("p[%d] = %c\n", i, p[i]);
    munmap(p, 10);
    close(fd);
}
```

示例 2:

深圳信盈达科技有限公司 专业提供单片机、嵌入式、ARM、LINUX、Android、PCB、FPGA 等技术培训、技术方案。

写端：（映射的文件不能为空，可以利用文件空洞技术）

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
main()
{
    int fd = open("mm.dat",O_RDWR|O_CREAT); //创建一个新文件并以读写方式打开
    char buf[20] = {0};
    write(fd,buf,sizeof(buf));//文件不能为空,写入 20 个 0，保证文件不为空
    char*p=(char*)mmap(NULL,20*sizeof(char),PROT_READ|PROT_WRITE,MAP_PRIVATE,fd,0);
    memset(p, 0, 20); //最好先初始化为 0
    memcpy(p, "hello world", 12);//写 hello 到内存
    sleep(5);
    munmap(p, 20);
    close(fd);
}
```

读端：

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
main()
{
    int fd = open("mm.dat",O_RDWR); //需要打开同一个文件
    char*p=(char*)mmap(NULL,20*sizeof(char),PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
    puts(p);    输出
    munmap(p, 20);
    close(fd);
}
```

此时你会发现文件 mm.dat 的内容为 hello wrold。因为 MAP_SHARED 会将修改的内容写回文件。如果你将上面的 MAP_SHARED 换成 MAP_PRIVATE，则没有任何输出效果。因为 MAP_PRIVATE 是私人的“写入时复制”，不会写回文件，只是自己本身的进程有效，即使你用有亲缘关系的也不行，这是内存映射 MAP_PRIVATE 与 SYSTEM V 的共享内存的 IPC_PRIVATE 的区别：

示例 3：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/mman.h>
#include <string.h>
```

```
int main()
{
```

```
int fd = open("a.txt", O_RDWR | O_CREAT);
if(fd == -1)
{
    perror("open error");
    exit(-1);
}
if(fork() > 0)
{
    char buf[64] = {"\0"};
    write(fd, buf, sizeof(buf));
    lseek(fd, 0, 0);
    char* pStr = (char*)mmap(NULL, 64, PROT_WRITE | PROT_READ, MAP_PRIVATE, fd, 0);
    memset(pStr, 0, 64);
    memcpy(pStr, "world", 6);
    sleep(10);
    munmap(pStr, 64);
    close(fd);
}
else
{
    sleep(2); //保证父有时间写
    char* pStr = (char*)mmap(NULL, 64, PROT_WRITE | PROT_READ, MAP_PRIVATE, fd, 0);
    puts(pStr);    可以再进行输出
    munmap(pStr, 64);
    close(fd);
}
return 0;
}
```

此时还是没有数据输出，还是要将 MAP_PRIVATE 更换成 MAP_SHARED 即可。

示例 4：与结构体混用

```
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>

typedef struct {
    int integer;
    char string[24];
} RECORD;

#define NRECORDS 100

int main()
{
    RECORD record, *mapped;
    int i, f;
    FILE *fp;
```

```
fp = fopen("records.dat","w+");
//写入 100 个学生
for(i=0; i<NRECORDS; i++) {
    record.integer = i;
    sprintf(record.string,"RECORD-%d",i);
    fwrite(&record,sizeof(record),1,fp);
}
fclose(fp);

/* 采用普通的方式修改 */
fp = fopen("records.dat","r+");
fseek(fp,43*sizeof(record),SEEK_SET);//跳过前 43 个
fread(&record,sizeof(record),1,fp);//读出第 44 个

record.integer = 143;//修改第 44 个
sprintf(record.string,"RECORD-%d",record.integer);

fseek(fp,43*sizeof(record),SEEK_SET);//定位到 43
fwrite(&record,sizeof(record),1,fp);//写入第 44 个
fclose(fp);

/* 采用内存映射的形式修改 */
fd= open("records.dat",O_RDWR);
mapped = (RECORD *)mmap(NULL, NRECORDS*sizeof(record), PROT_READ|PROT_WRITE,
MAP_SHARED, fd, 0);
//内存从 0 开始,长度是 100*学生的大小,可读可写,可以进程间共享,fd 是文件描述符,0 是偏移量
mapped[43].integer = 243;//直接修改 44
sprintf(mapped[43].string,"RECORD-%d",mapped[43].integer);

msync((void *)mapped, NRECORDS*sizeof(record), MS_ASYNC);//写入文件
munmap((void *)mapped, NRECORDS*sizeof(record));//取消内存映射
close(fd);

exit(0);
}
```

6.6 System V 共享内存机制: shmget shmat shmdt shmctl

共享内存也是进程间(进程间不需要有继承关系)通信的一种常用手段。一般 OS 通过内存映射与页交换技术,使进程的内存空间映射到不同的物理内存,这样能保证每个进程运行的独立性,不至于受其它进程的影响。但可以通过共享内存的方式,使不同进程的虚拟内存映射到同一块物理内存,一个进程往这块物理内存中更新的数据,另外的进程可以立即看到这块物理内存中修改的内容。

内存映射和共享内存的区别:

内存映射: 跟普通文件的读写相比, 加快对文件/设备的访问速度。

共享内存: 多进程间进行通信。

原理及实现: system V IPC 机制下的共享内存本质是一段特殊的内存区域, 进程间需要共享的数据被放在该共享内存区域中, 所有需要访问该共享区域的进程都要把该共享区域映射到本进程的地址空间中去。这样一个使

用共享内存的进程可以将信息写入该空间，而另一个使用共享内存的进程又可以通过简单的内存读操作获取刚才写入的信息，使得两个不同进程之间进行了一次信息交换，从而实现进程间的通信。共享内存允许一个或多个进程通过同时出现在它们的虚拟地址空间的内存进行通信，而这块虚拟内存的页面被每个共享进程的页表条目所引用，同时并不需要在所有进程的虚拟内存都有相同的地址。进程对象对于共享内存的访问通过 key（键）来控制，同时通过 key 进行访问权限的检查。

函数定义如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
key_t ftok(const char *pathname, int proj_id);
int shmget(key_t key, int size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

函数 ftok 用于创建一个关键字，可以用该关键字关联一个共享内存段。

参数 **pathname** 为一个全路径文件名，并且该文件必须可访问。

参数 **proj_id** 通常传入一非 0 字符。

通过 **pathname** 和 **proj_id** 组合可以创建唯一的 key（对任何进程都是唯一且相同的）。

如果调用成功，返回一关键字，否则返回-1。

函数 shmget 用于创建或打开一共享内存段，该内存段由函数的第一个参数标识。函数成功则返回一个该共享内存段的唯一标识号（唯一的标识了这个共享内存段），对任何进程都是唯一且相同的，后面会用到。

参数 **key** 是一个与共享内存段相关联的关键字，如果事先已经存在一个与指定关键字关联的共享内存段，则直接返回该内存段的标识。**key** 的值既可以用 **ftok** 函数产生，也可以是 **IPC_PRIVATE**（用于创建一个只属于创建进程的共享内存，主要用于父子通信），表示总是创建新的共享内存段。

参数 **size** 指定共享内存段的大小，以字节为单位。

参数 **shmflg** 是一掩码合成值，可以是访问权限值与(**IPC_CREAT** 或 **IPC_EXCL**)的合成。**IPC_CREAT** 表示如果不存在该内存段，则创建它。**IPC_EXCL** 表示如果该内存段存在，则函数返回失败结果(-1)。如果调用成功，返回内存段标识，否则返回-1。

函数 shmat 将共享内存段映射到进程空间的某一地址。

参数 **shmid** 是共享内存段的标识 通常应该是 **shmget** 的成功返回值。

参数 **shmaddr** 指定的是共享内存连接到当前进程中的地址位置。通常是 **NULL**，表示让系统来选择共享内存出现的地址。

参数 **shmflg** 是一组位标识，通常为 0 即可。

如果调用成功，返回映射后的进程空间的首地址，否则返回(void*)-1。

函数 shmdt 用于将共享内存段与进程空间分离。

参数 **shmaddr** 通常为 **shmat** 的成功返回值。

函数成功返回 0，失败时返回-1.注意，将共享内存分离并没删除它，只是使得该共享内存对当前进程不再可用。

函数 shmctl 是共享内存的控制函数，可以用来删除共享内存段。

参数 **shmid** 是共享内存段标识 通常应该是 **shmget** 的成功返回值

参数 **cmd** 是对共享内存段的操作方式，可选为 **IPC_STAT**,**IPC_SET**,**IPC_RMID**。通常为 **IPC_RMID**，表示删除共享内存段。

参数 **buf** 是表示共享内存段的信息结构体数据，通常为 **NULL**。

例如 **shmctl(kshareMem,IPC_RMID,NULL)**表示删除调共享内存段 **kHareMem**

示例 1：有亲缘关系

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
```

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define PERM S_IRUSR | S_IWUSR //表示用户可读可写 即 0600

int main(int argc, char **argv)
{
    /*在两个有亲属关系进程间通信,KEY 采用 IPC_PRIVATE 由系统自选*/
    int shmid = shmget(IPC_PRIVATE, 1024, PERM);
    if(fork() > 0)
    {
        char *p = (char*)shmat(shmid, NULL, 0); //将共享内存映射至本进程空间某一地址
        memset(p, 0, 1024); //初始化为 0
        strncpy(p, "share memory", 1024); //存入（写入）内容
        printf("parent %d Write buffer: %s\n", getpid(), p);
        wait(NULL); //防止僵尸进程
        sleep(2);
        shmctl(shmid, IPC_RMID, 0); /*删除共享内存,用 ipcs -m 看共享内存*/
        exit(0);
    }
    else
    {
        sleep(5); //让父有足够的时间写
        char *p = (char*)shmat(shmid, NULL, 0); //将共享内存映射至本进程空间某一地址
        printf("Client pid=%d,shmid=%d Read buffer: %s\n",getpid(),shmid,p);
        exit(0);
    }
}
```

示例 2： 非亲进程间通信的实现步骤如下：

写内存端：

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
int main()
{
    key_t key = ftok("b.dat",1); //1. 写入端先用 ftok 函数获得 key
    int shmid = shmget(key,4096,IPC_CREAT); //2. 写入端用 shmget 函数创建一共享内存段
    printf("key = %d shmid = %d\n", key, shmid);

    char *p = (char *)shmat(shmid, NULL, 0); //3. 获得共享内存段的首地址
    memset(p, 0, 4096);
    memcpy(p, "hello world", 4096); //4. 往共享内存段中写入内容
    shmdt(p); //5. 关闭共享内存段
    return 0;
}
```

```
}
```

读内存端：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
int main()
{
    key_t key = ftok("b.dat",1);
    int shmid = shmget(key,4096,IPC_CREAT);
    printf("key = %d shmid = %d\n", key, shmid);
    char *p = (char *)shmat(shmid, NULL, 0);
    printf("receive the data:%s\n",p);          //4. 读取共享内存段中的内容
    shmctl(shmid, IPC_RMID, 0);                //5. 删除共享内存段
    return 0;
}
```

6.7 消息队列

消息队列与 FIFO 很相似，都是一个队列结构，都可以有多个进程往队列里面写信息，多个进程从队列中读取信息。但 FIFO 需要读、写的两端事先都打开，才能够开始信息传递工作。而消息队列可以事先往队列中写信息，需要时再打开读取信息。

System V IPC 机制消息队列。

函数原型：

```
#include <sys / types.h>
#include <sys / ipc.h>
#include <sys / msg.h>
int msgget(key_t key, int msgflg);
int msgsnd(int msqid, struct msgbuf * msgp, size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, struct msgbuf * msgp, size_t msgsz, long msgtyp, int msgflg);
int msgctl(int msqid, int cmd, struct msqid_ds * buf);
```

函数 msgget 创建和访问一个消息队列。函数成功的时候，返回一个消息队列的唯一标识符 id（跟进程 ID 是一个类型），失败的时候，会返回-1；

参数 key 是唯一标识一个消息队列的关键字，如果为 IPC_PRIVATE (值为 0，用创建一个只有创建者进程才可以访问的消息队列)，表示创建一个只由调用进程使用的消息队列，非 0 值的 key (可以通过 ftok 函数获得) 表示创建一个可以被多个进程共享的消息队列；

参数 msgflg 指明队列的访问权限和创建标志，创建标志的可选值为 IPC_CREAT 和 IPC_EXCL 如果单独指定 IPC_CREAT, msgget 要么返回新创建的消息队列 id, 要么返回具有相同 key 值的消息队列 id；如果 IPC_EXCL 和 IPC_CREAT 同时指明，则要么创建新的消息队列，要么当队列存在时，调用失败并返回-1。

函数 msgsnd 和 msgrcv 用来将消息添加到消息队列中和从一个消息队列中获取信息。

参数 msqid 指明消息队列的 ID；通常是 msgget 函数成功的返回值。

参数 msgbuf 是消息结构体，它的长度必须小于系统规定的上限，必须以一个长整型成员变量开始，接收函数将用这个成员变量来确定消息的类型。必须重写这个结构体，其中第一个参数不能改，其他自定义。如下：

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

};

字段 `mtype` 是用户自己指定的消息类型（通常是 0—5 中的任意一个数值），该结构体第 2 个成员仅仅是一种说明性的结构，实际上用户可以使用任何类型的数据，就是消息内容；

参数 `msgsz` 是消息体的大小，每个消息体最大不要超过 4K；

参数 `msgflg` 可以为 0（通常为 0）或 `IPC_NOWAIT`，如果设置 `IPC_NOWAIT`，则 `msgsnd` 和 `msgrcv` 都不会阻塞，此时如果队列满并调用 `msgsnd` 或队列空时调用 `msgrcv` 将返回错误；也就是说如果没有设置为 `IPC_NOWAIT`，则当队列满并调用 `msgsnd` 或队列空时调用 `msgrcv` 将会阻塞，直到队列条件满足为止。

参数 `msgtyp` 有 3 种选项：

- `msgtyp == 0` 接收队列中的第 1 个消息（通常为 0）
- `msgtyp > 0` 接收对列中的第 1 个类型等于 `msgtyp` 的消息
- `msgtyp < 0` 接收其类型小于或等于 `msgtyp` 绝对值的第 1 个最低类型消息

函数 `msgctl` 是消息队列的控制函数，常用来删除消息队列。

参数 `msgqid` 是由 `msgget` 返回的消息队列标识符。

参数 `cmd` 通常为 `IPC_RMID` 表示删除消息队列。

参数 `buf` 通常为 `NULL` 即可。

另外，可用 `ipcs -q` 命令查看系统的消息队列，`ipcs -m` 查看系统的共享内存，`ipcs -s` 查看系统的信号量集。

示例：有亲缘关系的消息队列（`IPC_PRIVATE`）：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>

struct MSG{
    long mtype;
    char buf[64];
};

int main()
{
    int msgid = msgget(IPC_PRIVATE, 0666|IPC_CREAT);
    printf("msgid = %d\n", msgid);
    struct MSG msg;
    memset(&msg, 0, sizeof(struct MSG));
    if(fork() > 0)
    {
        msg.mtype = 1;
        strcpy(msg.buf, "aaaa");
        msgsnd(msgid, &msg, sizeof(struct MSG), 0);
        wait(NULL);
        msgctl(msgid, IPC_RMID, NULL);
    }
    else
    {
        sleep(2);      //让父进程有时间往消息队列里面写
        msgrcv(msgid, &msg, sizeof(struct MSG), 1, 0);
        puts(msg.buf);
    }
}
```



```
    return 0;
}
```

示例：没有亲缘关系

消息发送

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#define BUFFER 255
struct msgtype {    //重新定义该结构体
    long mtype;      //第一个参数不变
    char buf[BUFFER+1];
};

int main()
{
    int msgid = msgget((key_t)1235,0666 | IPC_CREAT); //获取消息队列
    struct msgtype msg;
    memset(&msg,0,sizeof(struct msgtype));
    msg.mtype = 1; //给结构体的成员赋值
    strncpy(msg.buf,"hello",BUFFER);
    msgsnd(msgid,&msg,sizeof(struct msgtype),0); //发送信号

    memset(&msg,0,sizeof(struct msgtype)); //清空结构体
    msgrcv(msgid,&msg,sizeof(struct msgtype),2,0); //接收信号
    printf("客户端收到消息: %s\n", msg.buf);
    return 0;
}
```

消息接收

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <sys/msg.h>
#define BUFFER 255

struct msgtype {
    long mtype;
    char buf[BUFFER+1];
};
```

```
int main()
{
    int msgid = msgget((key_t)1235, 0666 | IPC_CREAT); //获得消息队列
    struct msgtype msg;
    memset(&msg,0,sizeof(struct msgtype));
    while(1)
    {
        msgrcv(msgid,&msg,sizeof(struct msgtype),1,0);    //接收信号
        printf("服务器收到消息: %s\n", msg.buf);

        msg.mtype = 2;
        strncpy(msg.buf,"world",BUFFER);
        mgsnd(msgid,&msg,sizeof(struct msgtype),0);    //4. 发送信号
    }
    return 0;
}
```

两个程序分别编译，依次运行，不分先后顺序，就可以看到效果了。

6.8 信号量

内容提要：

信号量集：由若干个信号量组成的集合。就像整型数组是由多个整数组成的一样。

信号量：是信号量集中的一个元素。就好比整形数组中的一个元素。

每个信号量都有它的值：非负整数。就好比数组中的每个元素都有它的值。

同时每个信号量也有它在这个信号量集中的编号，就好比数组中的每个元素都有下标一样。

数组下标从 0 开始，信号量的编号也从 0 开始……

总之一句话，信号量集和数组很像……

信号量（也叫信号灯）是一种用于提供不同进程间或一个给定进程的不同线程间同步手段的原语。信号量是进程/线程同步的一种方式，有时候我们需要保护一段代码，使它每次只能被一个执行进程/线程运行，这种工作就需要一个二进制开关；有时候需要限制一段代码可以被多少个进程/线程执行，这就需要用到关于计数信号量。信号量开关是二进制信号量的一种逻辑扩展，两者实际调用的函数都是一样。

信号量分为以下三种。

1、System V 信号量，在内核中维护，可用于进程或线程间的同步，常用于进程的同步。

2、Posix 有名信号量，一种来源于 POSIX 技术规范的实时扩展方案（POSIX Realtime Extension），可用于进程或线程间的同步，常用于线程。

3、Posix 基于内存的信号量，存放在共享内存区中，可用于进程或线程间的同步。

为了获得共享资源,进程需要执行下列操作：

（1）测试控制该资源的信号量。

（2）若信号量的值为正，则进程可以使用该资源。然后将信号量值减 1，表示它使用了一个资源单位。此进程使用完共享资源后对应的信号量应该加 1。以便其他进程使用。

（3）若对信号量进行减一时，信号量的值为 0，则进程进入阻塞休息状态，直至信号量值大于 0。进程被唤醒，返回第（1）步。

为了正确地实现信号量，信号量值的测试及减 1 操作应当是原子操作（原子操作是不可分割的，在执行完毕不会被任何其它任务或事件中断）。为此信号量通常是在内核中实现的。

6.8.1 System V IPC 机制：信号量。

函数原型：

```
#include <sys/sem.h>
#include <sys/ipc.h>
#include <sys/types.h>
int semget(key_t key,int nsems,int flag);
int semop(int semid,struct sembuf *sops,size_t nops);
int semctl(int semid, int semnum, int cmd, union semun arg);
```

函数 semget 创建一个信号量集或访问一个已存在的信号量集。返回值：成功时，返回一个称为信号量集标识符的整数，semop 和 semctl 会使用它；出错时，返回-1。

参数 key 是唯一标识一个信号量的关键字，如果为 IPC_PRIVATE(值为 0，创建一个只有创建者进程才可以访问的信号量)，表示创建一个只由调用进程使用的信号量，非 0 值的 key(可以通过 ftok 函数获得)表示创建一个可以被多个进程共享的信号量；

参数 nsems 指定需要使用的信号量数目。如果是创建新集合，则必须指定 nsems。如果引用一个现存的集合，则将 nsems 指定为 0。

参数 flag 是一组标志，其作用与 open 函数的各种标志很相似。它低端的九个位是该信号量的权限，其作用相当于文件的访问权限。此外，它们还可以与键值 IPC_CREAT 按位或操作，以创建一个新的信号量。即使在设置了 IPC_CREAT 标志后给出的是一个现有的信号量的关键字，也并不是一个错误。我们也可以通过 IPC_CREAT 和 IPC_EXCL 标志的联合使用确保自己将创建出一个新的独一无二的信号量来，如果该信号量已经存在，就会返回一个错误。

函数 semop 用于改变信号量对象中各个信号量的状态。返回值：成功时，返回 0；失败时，返回-1。

参数 semid 是由 semget 返回的信号量标识符。

参数 sops 是指向一个结构体数组的指针。可以指向单个或者多个结构体变量。每个数组元素至少包含以下几个成员：

```
struct sembuf{
    short sem_num; //要操作的信号量在信号量集中的编号，第一个信号量的编号是 0。
    short sem_op; //sem_op 成员的值是信号量在一次操作中需要改变的数值。通常只会用到两个值，一个是-1，也就是 p（减一）操作，它等待信号量变为可用；一个是+1，也就是 v（加一）操作，它发送信号通知信号量现在可用。
```

```
    short sem_flg; //通常设为：SEM_UNDO，程序结束，信号量为 semop 调用前的值。
};
```

这段话很关键：它可以一次对一个信号量进行操作，此时数组长度为 1，也可以一次操作多个信号量。如果一次操作多个信号量，每个信号量按照这个数组各个元素指定的编号和值进行改变。

参数 nops 为 sops 指向的 sembuf 结构数组 的长度。

函数 semctl 用来直接控制信号量信息。函数返回值：成功时，返回 0；失败时，返回-1。

参数 semid 是由 semget 返回的信号量标识符。

参数 semnum 为要进行操作的集合中信号量的编号，当要操作到成组的信号量时，从 0 开始。一般取值为 0，表示这是第一个也是唯一的一个信号量。

参数 cmd 为执行的操作。通常为：IPC_RMID（立即删除信号集，唤醒所有被阻塞的进程）、GETVAL（根据 semun 指定的编号返回相应信号的值，此时该函数返回值就是你要获得的信号量的值，不再是 0 或-1）、SETVAL（根据 semun 指定的编号设定相应信号的值）、GETALL（获取所有信号量的值，此时第二个参数为 0，并且会将所有信号的值存入 semun.array 所指向的数组的各个元素中，此时需要用到第四个参数 union semun arg）、SETALL（将 semun.array 指向的数组的所有元素的值设定到信号量集中，此时第二个参数为 0，此时需要用到第四个参数 union semun arg）等。

第四个参数 arg 是一个 union semun 类型（具体的需要由程序员自己定义），它至少包含以下几个成员：

```
union semun{
    int val; /* Value for SETVAL */
    struct semid_ds *buf; /* Buffer for IPC_STAT, IPC_SET */
```

```
    unsigned short *array; /* Array for GETALL, SETALL */
};
```

示例 1: 一个关于信号量的直观易懂的程序。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <string.h>
union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short* array;
};

int main()
{
    int i;
    int key = ftok("b.dat", 1);
    int id = semget(key, 4, IPC_CREAT);
    unsigned short sz0[4], sz1[4] = {10, 20, 30, 40};
    union semun arg;
    arg.array = sz1;
    semctl(id, 0, SETALL, arg);
    struct sembuf aaa[2] = {{0, 10, 0}, {2, -5, 0}};
    semop(id, aaa, 2);
    arg.array = sz0;
    semctl(id, 0, GETALL, arg);
    for(i = 0; i < 4; i++)
        printf("%d:%d\n", i, sz0[i]);
    semctl(id, IPC_RMID, 0);
    return 0;
}
```

示例 2: 有亲缘关系的信号量进程间通信(生产者消费者问题)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <string.h>

union semun          //必须重写这个共用体
{
    int val;
    struct semid_ds *buf;
```

```
unsigned short* array;
};
int main()
{
    int semid = semget(IPC_PRIVATE, 1, 0666|IPC_CREAT); //创建信号量集
    if(fork() == 0)
    {
        struct sembuf sem = {0, 1, 0};           //定义信号量结构体
        while(1)
        {
            semop(semid, &sem, 1); //执行指定的 V 操作,表示生产产品
            printf("productor total number:%d\n", semctl(semid, 0, GETVAL)); //获得第一个信号的值
            sleep(1);
        }
    }
    else
    {
        sleep(2);           //先让子进程有时间生产
        struct sembuf sem = {0, -1, 0};           //定义信号量结构体
        while(1)
        {
            semop(semid, &sem, 1); //执行指定的 P 操作消费产品
            printf("costomer total number:%d\n", semctl(semid, 0, GETVAL)); //获得第一个信号的值
            sleep(2);
        }
    }
    return 0;
}
```

示例 3：没有亲缘关系的生产者消费者问题

生产者源码：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>
void init(); //initialization semaphore
void del(); //delete semaphore
int sem_id;
int main(int argc, char *argv[])
{
    struct sembuf sops[2] = {{0, 1, 0}, {1, -1, 0}};
    init(); //初始化操作
    printf("this is a producior\n");
    while(1)
    {
        printf("\n\nbefore produce\n");
```

```

        printf("productor number is %d\n", semctl(sem_id, 0, GETVAL));
        printf("space number is %d\n", semctl(sem_id, 1, GETVAL));
        semop(sem_id, sops+1, 1);
        printf("now producing .....n");
        semop(sem_id, sops, 1);

        printf("space number is %d\n", semctl(sem_id, 1, GETVAL));
        printf("productor number is %d\n", semctl(sem_id, 0, GETVAL));
        sleep(2);
    }
    del();
    return 0;
}
void init()
{
    int ret;
    union semun
    {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
    }arg;
    sem_id = semget((key_t)1234, 2, IPC_CREAT|0644);/*get semaphore include two sem*/

    unsigned short sem_array[2] = {0, 10};
    arg.array = sem_array;
    semctl(sem_id, 0, SETALL, arg); //将所有 semun.array 的值设定到信号集中，第二个参数为 0
    printf("productor init is %d\n", semctl(sem_id, 0, GETVAL));
    printf("space init is %d\n", semctl(sem_id, 1, GETVAL));
}
void del()
{
    semctl(sem_id, IPC_RMID, 0);    //删除信号集
}

```

消费者源码:

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

```

```

void init();
int sem_id;

```

```

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

```

```

struct sembuf sops[2] = {{0, -1, 0}, {1, 1, 0}};
init();
printf("this is a customer\n");
while(1)
{
    printf("\n\nbefore consume\n");
    printf("customer number is %d\n", semctl(sem_id, 0, GETVAL));
    printf("space number is %d\n", semctl(sem_id, 1, GETVAL));
    semop(sem_id, sops, 1);
    printf("now consume ..... \n");
    semop(sem_id, sops+1, 1);
    //printf("now consume ..... \n")前后的两句也可以合为一句 semop(sem_id, sops, 2);
    printf("space number is %d\n", semctl(sem_id, 1, GETVAL));
    printf("customer number is %d\n", semctl(sem_id, 0, GETVAL));
    sleep(1);
}
}
void init()
{
    sem_id = semget((key_t)1234, 2, IPC_CREAT|0644);/*get semaphore include two sem*/
}

```

在生产者源码里，首先用函数 `semctl()` 初始化信号量集合 `sem_id`，它包含两个信号，分别表示生产的数量和空仓库的数量，那么在消费者的进程中用相同的 `key` 值就会得到该信号量集合；实现两个进程之间的通信。

在主函数里，设定对两个信号量的 `PV` 操作，然后在各自的进程中对两个信号进行操作。

(1) 如果只运行生产者进程，则生产 10 个之后，该进程就会因为在得不到空仓库资源而阻塞，这个时候运行消费者进程，阻塞就会被解除；

(2) 如果先运行生产者进程，生产几个产品之后，关闭该进程，则运行消费者进程，当消费完生产的产品后，该进程就会因为在得不到产品资源而阻塞，这个时候运行生产者进程，阻塞就会被解除；

(3) 如果同时运行两个进程，由于消费比生产快，因此消费者每次都要等待生产者生产产品之后才能消费；在每次运行程序之前，一定要先运行生产者进程先初始化信号量。

6.8.2 选修：Posix 有名信号量

可用于进程和线程间通信。通常用于多线程的同步和互斥

函数原型：

```

#include <semaphore.h>
sem_t * sem_open(const char *pathname, int flags, mode_t mode, int init);
int sem_close(sem_t *psem);
sem_wait( sem_t *psem ); // P 操作（减一）如果信号量已减为 0， 则会阻塞，直到条件满足为止。
sem_post( sem_t *psem ); // V 操作（加一）

```

`P` 操作和 `V` 操作都是原子性的，如果操作的条件不满足，则会阻塞。

注意：在编译的过程中必须要加上 `-lpthread` 或 `-lrt` 选项。

示例 1：互斥模型

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <string.h>

```



```
#define SEM_NAME  "my_sem"
#define OPEN_FLAG O_RDWR|O_CREAT
#define OPEN_MODE 0777
#define INIT_VALUE 1

#define CHILD_STR "123456789"
#define PARENT_STR "abcdefgh"

void print_str(char* p)
{
    int i = 0 ;
    for( i = 0 ; i < strlen(p) ; i++ )
    {
        fputc( p[i] , stderr );
        usleep(1000*100);
    }
    fputc('\n' , stderr );
}

int main(int argc, char* argv[])
{
    pid_t  fork_id = 0;
    sem_t  *psem = NULL ;
    psem = sem_open( SEM_NAME, OPEN_FLAG, OPEN_MODE, INIT_VALUE );
    if( SEM_FAILED == psem )
    {
        perror("sem_open failed");
        return -1;
    }

    printf("psem is %X\n" , psem );

    fork_id = fork();
    if( 0 == fork_id )
    {
        sem_wait( psem );      // P 操作（减一），信号量的值减为 0
        print_str( CHILD_STR );
        sem_post( psem );      // V 操作（加一），信号量的值加为 1
    }
    else if( fork_id > 0 )
    {
        sem_wait( psem );      // P 操作（减一），信号量的值减为 0
        print_str( PARENT_STR );
        sem_post( psem );      // V 操作（加一），信号量的值加为 1
    }
    else
    {
        return -1;
    }
}
```

```
}
if( -1 ==sem_close( psem ))
{
    perror("sem_close failed");
    return  -1;
}
return  0;
}
```

示例 2：同步模型

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <string.h>
#define SEM_NAME1  "my_sem1"
#define INIT_VALUE1 1
#define SEM_NAME2  "my_sem2"
#define INIT_VALUE2 0
#define OPEN_FLAG O_RDWR|O_CREAT
#define OPEN_MODE 0777
#define SEM_FAILED -1
#define CHILD_STR "123456789"
#define PARENT_STR "abcdefgh"
void print_str(char* p)
{
    int i = 0 ;
    for( i = 0 ; i < strlen(p) ; i++ )
    {
        fputc( p[i] , stderr );
        usleep(1000*100);
    }
    fputc('\n' , stderr );
}

int main(int argc , char* argv[])
{
    pid_t fork_id = 0 ;
    sem_t *psem1 = NULL ;
    sem_t *psem2 = NULL ;
    psem1 = sem_open( SEM_NAME1 , OPEN_FLAG , OPEN_MODE , INIT_VALUE1 );
    if( SEM_FAILED == psem1 )
    {
        perror("sem_open failed");
        return  -1;
    }
    psem2 = sem_open( SEM_NAME2 , OPEN_FLAG , OPEN_MODE , INIT_VALUE2 );
    if( SEM_FAILED == psem2 )
```

```
{
    perror("sem_open failed");
    return -1;
}
printf("psem1 is %X\n", psem1 );
printf("psem2 is %X\n", psem2 );
fork_id = fork();
if( 0 == fork_id )
{
    while(1)
    {
        sem_wait( psem1 );    // P 操作（减一），信号量的值减为 0
        print_str( CHILD_STR );
        sem_post( psem2 );    // V 操作（加一），信号量的值加为 1
    }
}
else if( fork_id > 0 )
{
    while(1)
    {
        sem_wait( psem2 );    // P 操作（减一），信号量的值减为 0
        print_str( PARENT_STR );
        sem_post( psem1 );    // V 操作（加一），信号量的值加为 1
    }
}
else
{
    return -1;
}
if( -1 == sem_close( psem1 ))
{
    perror("sem_close failed");
    return -1;
}
if( -1 == sem_close( psem2 ))
{
    perror("sem_close failed");
    return -1;
}

return 0;
}
```

第7章 Linux 多线程

7.1 Linux 多线程概述

7.1.1 多线程概述

进程是系统中程序执行和资源分配的基本单位。每个进程有自己的数据段、代码段和堆栈段。这就造成进程在进行切换等操作时都需要有比较负责的上下文切换等动作。为了进一步减少处理器的空转时间支持多处理器和减少上下文切换开销，也就出现了线程。

线程通常叫做轻量级进程。线程是在共享内存空间中并发执行的多道执行路径，是一个更加接近于执行体的概念，拥有独立的执行序列，是进程的基本调度单元，每个进程至少都有一个 `main` 线程。它与同进程中的其他线程共享进程空间 {堆 代码 数据 文件描述符 信号等}，只拥有自己的栈空间，大大减少了上下文切换的开销。进程和线程的关系如下图 7.1 所示。

线程和进程在使用上各有优缺点：线程执行开销小，占用的 CPU 资源少，线程之间的切换快，但不利于资源的管理和保护；而进程正相反。从可移植性来讲，多进程的可移植性要好些。

同进程一样，线程也将相关的变量值放在线程控制表内。一个进程可以有多个线程，也就是有多个线程控制表及堆栈寄存器，但却共享一个用户地址空间。要注意的是，由于线程共享了进程的资源 and 地址空间，因此，任何线程对系统资源的操作都会给其他线程带来影响。

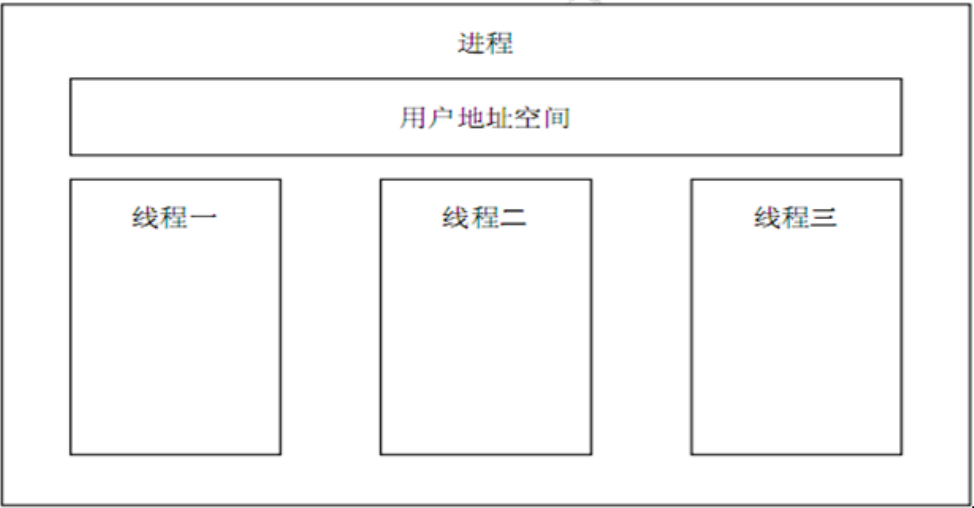


图 7.1

7.1.2 线程分类

按调度者分为用户级线程和核心级线程

用户级线程：主要解决上下文切换问题，调度算法和调度过程全部由用户决定，在运行时不需要特定的内核支持。缺点是无法发挥多处理器的优势

核心级线程：允许不同进程中的线程按照同一相对优先调度方法调度，发挥多处理器的并发优势

现在大多数系统都采用用户级线程和核心级线程并存的方法。一个用户级线程可以对应一个或多个核心级线程，也就是“一对一”或“一对多”模型。

7.1.3 线程创建的 Linux 实现

Linux 的线程是通过用户级的函数库实现的，一般采用 `pthread` 线程库实现线程的访问和控制。它用第 3 方 `posix` 标准的 `pthread`，具有良好的可移植性。 编译的时候要在后面加上 `-lpthread`

创建 退出 等待

多进程	fork()	exit()	wait()
多线程	pthread_create	pthread_exit()	pthread_join()

7.2 线程的创建和退出

创建线程实际上就是确定调用该线程函数的入口点，线程的创建采用函数 `pthread_create`。在线程创建以后，就开始运行相关的线程函数，在该函数运行完之后，线程就退出，这也是线程退出的一种方式。另一种线程退出的方式是使用函数 `pthread_exit()` 函数，这是线程主动退出行为。这里要注意的是，在使用线程函数时，不能随意使用 `exit` 退出函数进行出错处理，由于 `exit` 的作用是使调用进程终止，往往一个进程包括了多个线程，所以在线程中通常使用 `pthread_exit` 函数来代替进程中的退出函数 `exit`。

由于一个进程中的多个线程是共享数据段的，因此通常在线程退出之后，退出线程所占用的资源并不会随着线程的终止而得到释放。正如进程之间可以通过 `wait()` 函数系统调用来同步终止并释放资源一样，线程之间也有类似的机制，那就是 `pthread_join` 函数。`pthread_join` 函数可以用于将当前线程挂起，等待某个线程的结束。这个函数是一个线程阻塞函数，调用这函数的线程将一直等待直到被等待的线程结束为止，当函数返回时，被等待线程的资源被回收。

函数原型：

```
#include <pthread.h>
int pthread_create(pthread_t* thread, pthread_attr_t * attr, void *(*start_routine)(void *), void * arg);
void pthread_exit(void *retval);
```

通常的形式为：

```
pthread_t pthread;
pthread_create(&pthread, NULL, pthreadfunc, NULL); 或 pthread_create(&pthread, NULL, pthreadfunc, (void*)3);
pthread_exit(NULL); 或 pthread_exit((void*)3); //3 作为返回值被后面的 pthread_join 函数捕获。
```

函数 `pthread_create` 用来创建线程。返回值：成功，则返回 0；失败，则返回-1。各参数描述如下：

- 参数 `thread` 是传出参数，保存新线程的标识；
- 参数 `attr` 是一个结构体指针，结构中的元素分别指定新线程的运行属性，`attr` 可以用 `pthread_attr_init` 等函数设置各成员的值，但通常传入为 `NULL` 即可；
- 参数 `start_routine` 是一个函数指针，指向新线程的入口点函数，线程入口点函数带有一个 `void *` 的参数由 `pthread_create` 的第 4 个参数传入；
- 参数 `arg` 用于传递给第 3 个参数指向的入口点函数的参数，可以为 `NULL`，表示不传递。

函数 `pthread_exit` 表示线程的退出。其参数可以被其它线程用 `pthread_join` 函数捕获。

示例：

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <sys/stat.h>
void * fun(void *p) //参数的 p 值为 123
{
    int i;
    for(i = 1; i<10; i++)
    {
        printf("world,p = %d \n", p);
        sleep(1);
    }
    pthread_exit(200);
}
int main()
```

```
{
    pthread_t id;
    int a, i;
    pthread_create(&id, NULL, fun, 123);
    for(i = 1; i<10; i++)
    {
        printf("hello \n");
        sleep(1);
    }
    pthread_join(id, &a);
    return 0;
}
```

编译时需要带上线程库选项: gcc -o a a.c -lpthread

另外, 编译的时候会产生一些警告信息, 是因为某些参数的类型不匹配, 但这个不影响生成可执行程序, 可以不管它, 直接运行即可, 下同。

7.3 线程的等待退出

7.3.1 等待线程退出

线程从入口点函数自然返回, 或者主动调用 pthread_exit() 函数, 都可以让线程正常终止

线程从入口点函数自然返回时, 函数返回值可以被其它线程用 pthread_join 函数获取

pthread_join 原型为:

```
#include <pthread.h>
```

```
int pthread_join(pthread_t pthread, void **thread_return);
```

该函数是一个阻塞函数, 一直等到参数 pthread 指定的线程返回; 与多进程中的 wait 或 waitpid 类似。

thread_return 是一个传出参数, 接收线程函数的返回值。如果线程通过调用 pthread_exit() 终止, 则 pthread_exit() 中的参数相当于自然返回值, 照样可以被其它线程用 pthread_join 获取到。

示例 1: 返回值的例子

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

```
#include <pthread.h>
```

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

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

```
void * fun(void * p) //参数的 p 值为 123
```

```
{
    int i;
    for(i = 1; i<10; i++)
    {
        if(i == 5)
            pthread_exit(200);
        //return 200;
        printf("world, p = %d \n", p);
        sleep(1);
    }
}

int main()
{
```

```
pthread_t id;
int a, i;
pthread_create(&id, NULL, fun, 123);
for(i = 1; i < 10; i++)
{
    printf("hello \n");
    sleep(1);
}
pthread_join(id, &a);
printf("a = %d\n", a);
return 0;
}
```

运行可以发现，不论是 `pthread_exit` 还是 `return` 都会导致子进程退出，并且返回值 200 都会保存到变量 `a` 中。即 `pthread_join` 函数的第二个参数传的是哪个变量的地址，那么就会把子线程退出的返回值保存到哪个变量里。而不要被它的第二个参数的形式 `void **thread_return` 以及 `pthread(void *retval)` 的形式所迷惑。这个例子编译时会产生一些警告信息，是因为某些参数类型不匹配，如果要去掉警告信息，可以在相关参数前面加上强制类型转换即可。如下面的示例。但无论是加不加强制类型转换，此时都不影响其值的传递和程序运行结果。故也不要被程序中的强制类型转换所迷惑，它不会对运行结果有任何影响，学习的时候读者可以直接无视他们。前面的例子故意不加强制转换产生警告，是为了突出这些函数的根本特征。

示例 2: (加上强制类型转换后的示例程序)

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <sys/stat.h>
void * fun(void *p) //参数的 p 值为 123
{
    int i;
    for(i = 1; i < 10; i++)
    {
        if(i == 5)
            pthread_exit((void *)200);
        //return (void *)200;
        printf("world,p = %d \n", (int)p);
        sleep(1);
    }
}

int main()
{
    pthread_t id;
    int a, i;
    pthread_create(&id, NULL, fun, (void *)123);
    for(i = 1; i < 10; i++)
    {
        printf("hello \n");
        sleep(1);
    }
    pthread_join(id, (void **)&a);
    printf("a = %d\n", a);
}
```



```
    return 0;
}
```

该函数还有一个非常重要的作用，由于一个进程中的多个线程共享数据段，因此通常在一个线程退出后，退出线程所占用的资源并不会随线程结束而释放。如果 `th` 线程类型并不是自动清理资源类型的，则 `th` 线程退出后，线程本身的资源必须通过其它线程调用 `pthread_join` 来清除，这相当于多进程程序中的 `waitpid`。

7.3.2 线程的取消

线程也可以被其它线程杀掉，在 Linux 中的说法是一个线程被另一个线程取消(cancel)。

线程取消的方法是一个线程向目标线程发 `cancel` 信号，但是如何处理 `cancel` 信号则由目标线程自己决定，目标线程或者忽略、或者立即终止、或者继续运行至 `cancellation-point`(取消点)后终止。

根据 POSIX 标准，`pthread_join()`、`pthread_testcancel()`、`pthread_cond_wait()`、`pthread_cond_timedwait()`、`sem_wait()`、`sigwait()`等函数以及 `read()`、`write()`等会引起阻塞的系统调用都是 `Cancellation-point`，而其他 `pthread` 函数都不会引起 `Cancellation` 动作。但是 `pthread_cancel` 的手册页声称，由于 Linux 线程库与 C 库结合得不好，因而目前 C 库函数都不是 `Cancellation-point`；但 `CANCEL` 信号会使线程从阻塞的系统调用中退出，并置 `EINTR` 错误码，因此可以在需要作为 `Cancellation-point` 的系统调用前后调用 `pthread_testcancel()`，从而达到 POSIX 标准所要求的目标，即如下代码段：

```
pthread_testcancel();
retcode = read(fd, buffer, length);
pthread_testcancel();
```

但是从 RedHat9.0 的实际测试来看，至少有些 C 库函数的阻塞函数是取消点，如 `read()`、`getchar()`等，而 `sleep()` 函数不管线程是否设置了 `pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL)`，都起到取消点作用。总之，线程的取消一方面是一个线程强行杀另外一个线程，从程序设计角度看并不是一种好的风格，另一方面目前 Linux 本身对这方面的支持并不完善，所以在实际应用中应该谨慎使用！！

```
int pthread_cancel(pthread_t thread); //尽量不要用，linux 支持并不完善。
```

示例：子线程释放空间

```
#include <stdio.h>
#include <pthread.h>
#include <malloc.h>
void* threadfunc(void *args)
{
    char *p = (char*)malloc(10);        //自己分配了内存
    int i = 0;
    for(; i < 10; i++)
    {
        printf("hello,my name is lirongye!\n");
        sleep(1);
    }
    free(p);                            //如果父线程中没有调用 pthread_cancel，此处可以执行
    printf("p is freed\n");
    pthread_exit((void*)3);
}
int main()
{
    pthread_t pthread;
    pthread_create(&pthread, NULL, threadfunc, NULL);
    int i = 1;
    for(; i < 5; i++) //父线程的运行次数比子线程的要少，当父线程结束的时候，如果没有 pthread_join
```

函数等待子线程执行的话，父线程会退出，而主线程的退出会导致进程的退出，故子线程也会退出。

```
{
    printf("hello,nice to meet you!\n");
    sleep(1);
    //if(i % 3 == 0)
        //pthread_cancel(pthreadid); //表示当 i%3==0 的时候就取消子线程，该函数将导致子线程直接退出，不会执行上面紫色的 free 部分的代码，即释放空间失败。要想释放指针类型的变量 p，此时必须要用 pthread_cleanup_push 和 pthread_cleanup_pop 函数释放空间，见后面的例子
}
int retvalue = 0;
pthread_join(pthreadid,(void**)&retvalue); //等待子线程释放空间，并获取子线程的返回值
printf("return value is :%d\n",retvalue);
return 0;
}
```

7.3.3 线程终止清理函数

不论是可预见的线程终止还是异常终止，都会存在资源释放的问题，在不考虑因运行出错而退出的前提下，如何保证线程终止时能顺利的释放掉自己所占用的资源，特别是锁资源，就是一个必须考虑解决的问题。

最经常出现的情形是资源独占锁的使用：线程为了访问临界共享资源而为其加上锁，但在访问过程中该线程被外界取消，或者发生了中断，则该临界资源将永远处于锁定状态得不到释放。外界取消操作是不可预见的，因此的确需要一个机制来简化用于资源释放的编程。

在 POSIX 线程 API 中提供了一个 pthread_cleanup_push()/pthread_cleanup_pop()函数对用于自动释放资源--从 pthread_cleanup_push()的调用点到 pthread_cleanup_pop()之间的程序段中的终止动作都将执行 pthread_cleanup_push()所指定的清理函数。API 定义如下：

```
void pthread_cleanup_push(void (*routine) (void *), void *arg)
```

```
void pthread_cleanup_pop(int execute)
```

pthread_cleanup_push()/pthread_cleanup_pop()采用先入后出的栈结构管理

void routine(void *arg)函数在调用 pthread_cleanup_push()时压入清理函数栈，多次对 pthread_cleanup_push()的调用将在清理函数栈中形成一个函数链，在执行该函数链时按照压栈的相反顺序弹出。execute 参数表示程序自然执行到 pthread_cleanup_pop()时是否在弹出清理函数的同时执行该函数，为 0 表示不执行，非 0 为执行；这个参数并不影响异常终止时清理函数的执行。

pthread_cleanup_push()/pthread_cleanup_pop()是以宏方式实现的，这是 pthread.h 中的宏定义：

```
#define pthread_cleanup_push(routine,arg) \
{ struct _pthread_cleanup_buffer _buffer; \
  _pthread_cleanup_push (&_buffer, (routine), (arg)); \
#define pthread_cleanup_pop(execute) \
  _pthread_cleanup_pop (&_buffer, (execute)); }
```

可见，pthread_cleanup_push()带有一个"{"，而 pthread_cleanup_pop()带有一个"}"，因此这两个函数必须成对出现，且必须位于程序的同一级别的代码段中才能通过编译。

pthread_cleanup_pop 的参数 execute 如果为非 0 值，则按栈的顺序注销掉一个原来注册的清理函数的时候，会执行该函数；当 pthread_cleanup_pop()函数的参数为 0 时，仅仅在线程调用 pthread_exit 函数或者其它线程对本线程调用 pthread_cancel 函数时，才在弹出“清理函数”的同时执行该“清理函数”。

示例 1：

```
#include <stdio.h>
#include <pthread.h>
void fun(void *p)
{
    printf("fun: %d\n",(int)p);
}
```

```
}
void *thread(void *p)
{
    pthread_cleanup_push(fun,(void *)1);
    pthread_cleanup_push(fun,(void *)2);
    sleep(2);
    //pthread_exit((void *)100);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(1);
}
int main()
{
    pthread_t id;
    pthread_create(&id, NULL, thread, (void *)2);
    pthread_join(id,NULL);
    return 0;
}
```

运行结果为：fun:1。

如果把//pthread_exit((void *)100)的注释取消掉再编译运行，结果为：

fun: 2

fun: 1

如果将里面的两次 pthread_cleanup_pop(1);改为 pthread_cleanup_pop(0);推测一下结果是怎样？

→没有任何输出（此时 fun 函数得不到执行）

如果修改为 0 之后，再在 sleep(2)之后添加 pthread_exit(NULL);则此时的结果又是如何：

→跟 pthread_cleanup_pop(1);实现的结果一样了。

示例 2：用 pthread_cleanup_push 和 pthread_cleanup_pop 来释放子线程分配的内存空间

```
#include <stdio.h>
#include <pthread.h>
#include <malloc.h>
void freemem(void * args)
{
    free(args);
    printf("clean up the memory!\n");
}
void* threadfunc(void *args)
{
    char *p = (char*)malloc(10);          //自己分配了内存
    pthread_cleanup_push(freemem,p);
    int i = 0;
    for(; i < 10; i++)
    {
        printf("hello,my name is lry!\n");
        sleep(1);
    }
    pthread_exit((void*)3);
    pthread_cleanup_pop(0);
}
int main()
{

```

```
pthread_t pthread;  
pthread_create(&pthread, NULL, threadfunc, NULL);  
int i = 1;  
for(i < 5; i++) //父线程的运行次数比子线程的要少，当父线程结束的时候，如果没有 pthread_join  
函数等待子线程执行的话，子线程也会退出，即子线程也只执行了 4 次。  
{  
    printf("hello,nice to meet you!\n");  
    sleep(1);  
    if(i % 3 == 0)  
        pthread_cancel(pthread); //表示当 i%3==0 的时候就取消子线程，该函数将导致直接退出，不  
会执行上面紫色的 free 部分的代码，即释放空间失败。要想释放指针类型的变量 p，必须要用  
pthread_cleanup_push 和 pthread_cleanup_pop 函数释放空间  
}  
int retvalue = 0;  
pthread_join(pthread,(void*)&retvalue); //等待子线程释放空间，并获取子线程的返回值  
printf("return value is :%d\n",retvalue);  
return 0;  
}
```

7.4 线程的互斥

在 Posix Thread 中定义了一套专门用于线程互斥的 mutex 函数。mutex 是一种简单的加锁的方法来控制对共享资源的存取，这个互斥锁只有两种状态（上锁和解锁），可以把互斥锁看作某种意义上的全局变量。为什么需要加锁，就是因为多个线程共用进程的资源，要访问的是公共区间时（全局变量），当一个线程访问的时候，需要加上锁以防止另外的线程对它进行访问，以实现资源的独占。在一个时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程才能够对共享资源进行操作。若其他线程希望上锁一个已经上锁了的互斥锁，则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。

互斥锁有三种类型：快速锁，嵌套锁（递归锁），检错锁。

互斥锁有两种创建（初始化）方法：静态方式（使用宏）和动态方式（用函数创建）。

互斥锁有五个函数：创建（初始化）锁，加锁，解锁，测试加锁，销毁锁。

互斥锁在创建（初始化）的时候必须指定类型。如果用函数创建锁，则由函数的第二个参数指定锁的类型。

1. 创建和销毁锁

有两种方法创建互斥锁，静态方式和动态方式。

● 静态方式：

POSIX 定义了一个宏 PTHREAD_MUTEX_INITIALIZER 来静态初始化互斥锁，方法如下：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

在 Linux Threads 实现中，pthread_mutex_t 是一个结构，而 PTHREAD_MUTEX_INITIALIZER 则是一个宏常量。

● 动态方式：

动态方式是采用 pthread_mutex_init() 函数来初始化互斥锁，API 定义如下：

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
```

其中 mutexattr 用于指定互斥锁属性（见下），如果为 NULL 则使用缺省属性。通常为 NULL。

pthread_mutex_destroy() 用于注销一个互斥锁，API 定义如下：

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

销毁一个互斥锁即意味着释放它所占用的资源，且要求锁当前处于开放状态。由于在 Linux 中，互斥锁并不占用任何资源，因此 Linux Threads 中的 pthread_mutex_destroy() 除了检查锁状态以外（锁定状态则返回 EBUSY）

没有其他动作。

2. 互斥锁属性

互斥锁属性结构体的定义为：

```
typedef struct
{
    int __mutexkind;
} pthread_mutexattr_t;
```

互斥锁的属性在创建锁的时候指定，在 LinuxThreads 实现中仅有一个锁类型属性__mutexkind，不同的锁类型在试图对一个已经被锁定的互斥锁加锁时表现不同也就是是否阻塞等待。有三个值可供选择：

1、PTHREAD_MUTEX_TIMED_NP，这是缺省值（直接写 NULL 就是表示这个缺省值），也就是普通锁(或快速锁)。当一个线程加锁以后，其余请求锁的线程将形成一个阻塞等待队列，并在解锁后按优先级获得锁。这种锁策略保证了资源分配的公平性

示例：初始化一个快速锁。

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
```

2、PTHREAD_MUTEX_RECURSIVE_NP，嵌套锁（递归锁），允许同一个线程对同一个锁成功获得多次，并通过多次 unlock 解锁。如果是不同线程请求，则在加锁线程解锁时重新竞争。

示例：初始化一个嵌套锁。

```
pthread_mutex_t lock;
pthread_mutexattr_t mutexattr;
mutexattr.__mutexkind = PTHREAD_MUTEX_RECURSIVE_NP;
pthread_mutex_init(&lock, &mutexattr);
```

3、PTHREAD_MUTEX_ERRORCHECK_NP，检错锁，如果同一个线程请求同一个锁，则返回 EDEADLK，否则与 PTHREAD_MUTEX_TIMED_NP 类型动作相同。这样就保证当不允许多次加锁时不会出现最简单情况下的死锁。

示例：初始化一个检错锁。

```
pthread_mutex_t lock;
pthread_mutexattr_t mutexattr;
mutexattr.__mutexkind = PTHREAD_MUTEX_ERRORCHECK_NP;
pthread_mutex_init(&lock, &mutexattr);
```

与上面的锁类型相对应，有 3 种类型的静态锁：

```
//快速静态锁 pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
//递归锁 pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
//检错锁 pthread_mutex_t errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

3. 锁操作

锁操作主要包括

```
加锁      int pthread_mutex_lock(pthread_mutex_t *mutex)
解锁      int pthread_mutex_unlock(pthread_mutex_t *mutex)
测试加锁  int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

- **pthread_mutex_lock：**加锁，不论哪种类型的锁，都不可能两个不同的线程同时得到，而必须等待解锁。对于普通锁，解锁者可以是同进程内任何线程；而检错锁和嵌套锁则必须由加锁者解锁才有效，否则返回 EPERM；在同一进程中的线程，如果加锁后没有解锁，则任何其他线程都无法再获得锁。
- **pthread_mutex_unlock：**根据不同的锁类型，可实现不同的行为：
 - 对于快速锁，pthread_mutex_unlock 解除锁定；
 - 对于递归锁，pthread_mutex_unlock 使锁上的引用计数减 1；
 - 对于检错锁，如果锁是本线程锁定的，则解除锁定，否则什么也不做。
- **pthread_mutex_trylock：**语义与 pthread_mutex_lock()类似，如果互斥锁未被上锁则对其上锁，但不同的是在锁已经被占据时返回 EBUSY 而不是挂起等待。

示例：比较 pthread_mutex_trylock()与 pthread_mutex_lock()

```
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t lock;
void* pthfunc(void *args)
{
    pthread_mutex_lock(&lock);          //先加一次锁
    pthread_mutex_lock(&lock);          //再用 lock 加锁，会挂起阻塞
    //pthread_mutex_trylock(&lock);      //用 trylock 加锁，则不会挂起阻塞
    printf("hello\n");
    sleep(1);
    pthread_exit(NULL);
}
main()
{
    pthread_t pthid = 0;
    pthread_mutex_init(&lock,NULL);
    pthread_create(&pthid,NULL,pthfunc,NULL);
    pthread_join(pthid,NULL);
    pthread_mutex_destroy(&lock);
}
```

4. 加锁注意事项

如果线程在加锁后解锁前被取消，锁将永远保持锁定状态，因此如果在关键区段内有取消点存在，则必须在退出回调函数 pthread_cleanup_push/pthread_cleanup_pop 中解锁。同时不应该在信号处理函数中使用互斥锁，否则容易造成死锁。

请看下面的例子：

```
#include <stdio.h>
#include <pthread.h>
#include <signal.h>
pthread_mutex_t lock;
void fun(int n)
{
    pthread_mutex_lock(&lock);          //此处加锁会成死锁，导致信号处理函数无法返回，进而导致主
    线程也不能再执行了
    int i = 0;
    while(i < 5)
    {
        printf("fun i = %d\n", i);
        i++;
        sleep(1);
        pthread_mutex_unlock(&lock);    //解锁
    }
}
void* pthfunc(void *args)
{
    int n;
    signal(2, fun);
    pthread_mutex_lock(&lock);          //先加一次锁
    while(1)
```



```
{
    printf("子线程正在运行\n");
    sleep(1);
}
pthread_mutex_unlock(&lock);    //解锁
pthread_exit(NULL);
}
void main()
{
    pthread_t pthid = 0;
    pthread_mutex_init(&lock,NULL);
    pthread_create(&pthid,NULL,pthfunc,NULL);
    while(1)
    {
        printf("主线程正在运行\n");
        sleep(1);
    }
    pthread_join(pthid,NULL);
    pthread_mutex_destroy(&lock);
}
```

5. 互斥锁实例

示例：火车站售票（此处不加锁，则会出现卖出负数票的情况）

```
#include <stdio.h>
#include <pthread.h>
int ticketcount = 11; //火车票，公共资源（全局）
void* salewinds1(void* args)
{
    while(ticketcount > 0)
    {
        printf("windows1 start sale ticket!the ticket is:%d\n",ticketcount);
        sleep(2);
        ticketcount --;
        printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
    }
}
void* salewinds2(void* args)
{
    while(ticketcount > 0)
    {
        printf("windows2 start sale ticket!the ticket is:%d\n",ticketcount);
        sleep(2);
        ticketcount --;
        printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
    }
}
int main()
{
    pthread_t pthid1 = 0;
    pthread_t pthid2 = 0;
```



```
pthread_create(&pthid1,NULL,salewinds1,NULL);
pthread_create(&pthid2,NULL,salewinds2,NULL);
pthread_join(pthid1,NULL);
pthread_join(pthid2,NULL);
return 0;
}
```

示例：加锁之后的火车票

```
#include <stdio.h>
#include <pthread.h>
int ticketcount = 11;
pthread_mutex_t lock;
void* salewinds1(void* args)
{
    while(1)
    {
        pthread_mutex_lock(&lock); //因为要访问全局的共享变量，所以就要加锁
        if(ticketcount > 0) //如果有票
        {
            printf("windows1 start sale ticket!the ticket is:%d\n",ticketcount);
            sleep(2);
            ticketcount --;
            printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
        }
        else
        {
            pthread_mutex_unlock(&lock);
            pthread_exit(NULL);
        }
        pthread_mutex_unlock(&lock);
        sleep(1); //要放到锁的外面，让另一个有时间锁
    }
}
void* salewinds2(void* args)
{
    while(1)
    {
        pthread_mutex_lock(&lock);
        if(ticketcount>0)
        {
            printf("windows2 start sale ticket!the ticket is:%d\n",ticketcount);
            sleep(2);
            ticketcount --;
            printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
        }
        else
        {
            pthread_mutex_unlock(&lock);
            pthread_exit(NULL);
        }
    }
}
```

```
        pthread_mutex_unlock(&lock);
        sleep(1);
    }

}

int main()
{
    pthread_t pthread1 = 0;
    pthread_t pthread2 = 0;
    pthread_mutex_init(&lock, NULL);    //初始化锁
    pthread_create(&pthread1, NULL, salewinds1, NULL);
    pthread_create(&pthread2, NULL, salewinds2, NULL);
    pthread_join(pthread1, NULL);
    pthread_join(pthread2, NULL);
    pthread_mutex_destroy(&lock);    //销毁锁
    return 0;
}
```

总结：线程互斥 mutex：加锁步骤如下：

1. 定义一个全局的 pthread_mutex_t lock；或者用

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; //则 main 函数中不用 init

2. 在 main 中调用 pthread_mutex_init 函数进行初始化，初始化一个互斥锁之后，其处于被解锁的状态，可以被上锁

3. 在子线程函数中调用 pthread_mutex_lock 加锁
4. 在子线程函数中调用 pthread_mutex_unlock 解锁
5. 最后在 main 中调用 pthread_mutex_destroy 函数进行销毁

7.5 线程的同步

7.5.1 条件变量 信号量同步

条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待条件变量的条件成立而挂起；另一个线程使条件成立（给出条件成立信号）。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。

1、创建和注销

条件变量和互斥锁一样，都有静态、动态两种创建方式：

静态方式使 PTHREAD_COND_INITIALIZER 常量，如下：

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

动态方式调用 pthread_cond_init()函数，API 定义如下：

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
```

尽管 POSIX 标准中为条件变量定义了属性，但在 Linux Threads 中没有实现，因此 cond_attr 值通常为 NULL，且被忽略。

注销一个条件变量需要调用 pthread_cond_destroy()，只有在没有线程在该条件变量上等待的时候才能注销这个条件变量，否则返回 EBUSY。因为 Linux 实现的条件变量没有分配什么资源，所以注销动作只包括检查是否有等待线程。API 定义如下： int pthread_cond_destroy(pthread_cond_t *cond);

2、等待和激发

等待条件有两种方式：无条件等待 pthread_cond_wait()和计时等待 pthread_cond_timedwait():

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
```

线程解开 mutex 指向的锁并被条件变量 cond 阻塞。其中计时等待方式表示经历 abstime 段时间后，即使条件变量不满足，阻塞也被解除。无论哪种等待方式，都必须和一个互斥锁配合，以防止多个线程同时请求 pthread_cond_wait()（或 pthread_cond_timedwait()，下同）的竞争条件（Race Condition）。mutex 互斥锁必须是普通锁（PTHREAD_MUTEX_TIMED_NP），且在调用 pthread_cond_wait()前必须由本线程加锁

（pthread_mutex_lock()），而在更新条件等待队列以前，mutex 保持锁定状态，并在线程挂起进入等待前解锁。在条件满足从而离开 pthread_cond_wait()之前，mutex 将被重新加锁，以与进入 pthread_cond_wait()前的加锁动作对应。（也就是说在做 pthread_cond_wait 之前，往往要用 pthread_mutex_lock 进行加锁，而调用 pthread_cond_wait 函数会将锁解开，然后将线程挂起阻塞。直到条件被 pthread_cond_signal 激发，该函数内部又会将锁状态恢复为锁定状态，最后再用 pthread_mutex_unlock 进行解锁）。

激发条件有两种形式，pthread_cond_signal()激活一个等待该条件的线程，存在多个等待线程时按入队顺序激活其中一个；而 pthread_cond_broadcast()则激活所有等待线程

3、其他

pthread_cond_wait()和 pthread_cond_timedwait()都被实现为取消点，也就是说如果 pthread_cond_wait()被取消，则退出阻塞，然后将锁状态恢复，然后当前线程才会终止。即互斥锁又恢复锁定状态，然而当前线程已经被取消掉，那么这个互斥锁就不会被解开了，此时锁得不到释放，就会造成死锁，因而需要定义退出回调函数来为其解锁。

示例 1：

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

pthread_mutex_t mutex;//定义了一个互斥锁变量
pthread_cond_t cond;//定义了一个条件变量

void * child1(void *arg)
{
    printf("线程 1 开始运行！ \n");
    printf("线程 1 上锁了: %d\n", pthread_mutex_lock(&mutex));
    printf("线程 1 开始等待条件被激活\n");
    pthread_cond_wait(&cond,&mutex); //等待父线程发送信号
    printf("条件满足后，停止阻塞！ \n");
    pthread_mutex_unlock(&mutex);
    return 0;
}

int main(void)
{
    pthread_t tid1;
    printf("测试条件变量！ \n");
    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&cond,NULL);
    pthread_create(&tid1,NULL,child1,NULL);
    sleep(5);           //6
    pthread_cond_signal(&cond);//让条件满足，即激活条件
    sleep(3);
    pthread_cond_destroy(&cond);//注销条件变量
    pthread_mutex_destroy(&mutex);//注销互斥锁变量
    return 0;
}
```

```
}
```

以下示例集中演示了互斥锁和条件变量的结合使用，以及取消对于条件等待动作的影响。在例子中，有两个线程被启动，并等待同一个条件变量，如果不使用退出回调函数（见范例中的注释部分），则 tid2 将在 pthread_mutex_lock()处永久等待。如果使用回调函数，则 tid2 的条件等待及主线程的条件激发都能正常工作。

示例 2:

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

pthread_mutex_t mutex;
pthread_cond_t cond;

void ThreadClean(void *arg)
{
    pthread_mutex_unlock(&mutex);
}

void * child1(void *arg)
{
    //pthread_cleanup_push(ThreadClean,NULL); //1
    while(1){
        printf("thread 1 get running \n");
        printf("thread 1 pthread_mutex_lock returns %d\n", pthread_mutex_lock(&mutex));
        pthread_cond_wait(&cond,&mutex); //等待父线程发送信号
        printf("thread 1 condition applied\n");
        pthread_mutex_unlock(&mutex);
        sleep(5);
    }
    //pthread_cleanup_pop(0); //2
    return 0;
}

void *child2(void *arg)
{
    while(1){
        sleep(3); //3
        printf("thread 2 get running.\n");
        printf("thread 2 pthread_mutex_lock returns %d\n", pthread_mutex_lock(&mutex));
        pthread_cond_wait(&cond,&mutex);
        printf("thread 2 condition applied\n");
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}

int main(void)
{
    pthread_t tid1,tid2;
    printf("hello, condition variable test\n");
```

```

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond,NULL);
pthread_create(&tid1,NULL,child1,NULL);
pthread_create(&tid2,NULL,child2,NULL);
while(1){ //父线程
    sleep(2);           //4
    pthread_cancel(tid1); //5
    sleep(2);           //6
    pthread_cond_signal(&cond);
}
sleep(10);
return 0;
}

```

不做注释 1, 2 则导致 child1 中的 unlock 得不到执行, 锁一直没有关闭, 而 child2 中的锁不能执行 lock, 则会一直在 pthread_mutex_lock() 处永久等待。如果不做注释 5 的 pthread_cancel() 动作, 即使没有那些 sleep() 延时操作, child1 和 child2 都能正常工作。注释 3 和注释 4 的延迟使得 child1 有时间完成取消动作, 从而使 child2 能在 child1 退出之后进入请求锁操作。如果没有注释 1 和注释 2 的回调函数定义, 系统将挂起在 child2 请求锁的地方, 因为 child1 没有释放锁; 而如果同时也不做注释 3 和注释 4 的延时, child2 能在 child1 完成取消动作以前得到控制, 从而顺利执行申请锁的操作, 但却可能挂起在 pthread_cond_wait() 中, 因为其中也有申请 mutex 的操作。child1 函数给出的是标准的条件变量的使用方式: 回调函数保护, 等待条件前锁定, pthread_cond_wait() 返回后解锁。

条件变量机制和互斥锁一样, 不能用于信号处理中, 在信号处理函数中调用 pthread_cond_signal() 或者 pthread_cond_broadcast() 很可能引起死锁。

示例: 火车售票, 利用条件变量, 当火车票卖完的时候, 再重新设置票数为 10

```

#include <pthread.h>
#include <stdio.h>
int ticketcount = 10;
pthread_mutex_t lock; //互斥锁
pthread_cond_t cond; //条件变量
void * salewinds1(void * args)
{
    while (1)
    {
        pthread_mutex_lock(&lock); //因为要访问全局的共享变量 ticketcount, 所以就要加锁
        if (ticketcount > 0) //如果有票
        {
            printf("windows1 start sale ticket!the ticket is:%d\n", ticketcount);
            ticketcount--; //则卖出一张票
            if (ticketcount == 0)
                pthread_cond_signal(&cond); //通知没有票了
            printf("sale ticket finish!,the last ticket is:%d\n", ticketcount);
        }
        else //如果没有票了, 就解锁退出
        {
            pthread_mutex_unlock(&lock);
            break;
        }
        pthread_mutex_unlock(&lock);
        sleep(1); //要放到锁的外面
    }
}

```

```
}
void * salewinds2(void * args)
{
    while (1)
    {
        pthread_mutex_lock(&lock);
        if (ticketcount > 0)
        {
            printf("windows2 start sale ticket!the ticket is:%d\n", ticketcount);
            ticketcount--;
            if (ticketcount == 0)
                pthread_cond_signal(&cond); //发送信号
            printf("sale ticket finish!,the last ticket is:%d\n", ticketcount);
        }
        else
        {
            pthread_mutex_unlock(&lock);
            break;
        }
        pthread_mutex_unlock(&lock);
        sleep(1);
    }
}
void * setticket(void * args) //重新设置票数
{
    pthread_mutex_lock(&lock); //因为要访问全局变量 ticketcount，所以要加锁
    if (ticketcount > 0)
        pthread_cond_wait(&cond, &lock); //如果有票就解锁并阻塞，直到没有票就执行下面的
    ticketcount = 10; //重新设置票数为 10
    pthread_mutex_unlock(&lock); //解锁
    sleep(1);
    pthread_exit(NULL);
}
main()
{
    pthread_t pthid1, pthid2, pthid3;
    pthread_mutex_init(&lock, NULL); //初始化锁
    pthread_cond_init(&cond, NULL); //初始化条件变量
    pthread_create(&pthid1, NULL, salewinds1, NULL); //创建线程
    pthread_create(&pthid2, NULL, salewinds2, NULL);
    pthread_create(&pthid3, NULL, setticket, NULL);
    pthread_join(pthid1, NULL); //等待子线程执行完毕
    pthread_join(pthid2, NULL);
    pthread_join(pthid3, NULL);
    pthread_mutex_destroy(&lock); //销毁锁
    pthread_cond_destroy(&cond); //销毁条件变量
}
```

7.5.2 信号灯

信号灯与互斥锁和条件变量的主要不同在于"灯"的概念，灯亮则意味着资源可用（即可以执行加锁操作），灯灭则意味着不可用（即可以执行解锁操作）。如果说后两种同步方式侧重于"等待"操作，即资源不可用的话，信号灯机制则侧重于点灯，即告知资源可用；没有等待线程的解锁或激发条件都是没有意义的，而没有等待灯亮的线程的点灯操作则有效，且能保持灯亮状态。当然，这样的操作原语也意味着更多的开销。

信号灯的应用除了灯亮/灯灭这种二元灯以外，也可以采用大于 1 的灯数，以表示资源数大于 1，这时可以称之为多元灯。

1. 创建和注销

POSIX 信号灯标准定义了有名信号灯和无名信号灯两种，但 LinuxThreads 的实现仅有无名灯，同时有名灯除了总是可用于多进程之间以外，在使用上与无名灯并没有很大的区别，因此下面仅就无名灯进行讨论。

```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem, int pshared, unsigned int value); //通常 pshared 为 0.表示线程间
```

这是创建信号灯的 API，其中 value 为信号灯的初值，pshared 表示是否为多进程共享而不仅仅是用于一个进程之间的多线程共享。LinuxThreads 没有实现多进程共享信号灯，因此所有非 0 值的 pshared 输入都将使 sem_init() 返回-1，且置 errno 为 ENOSYS。初始化好的信号灯由 sem 变量表征，用于以下点灯、灭灯操作。

```
int sem_destroy(sem_t * sem);
```

被注销的信号灯 sem 要求已没有线程在等待该信号灯，否则返回-1，且置 errno 为 EBUSY。除此之外，LinuxThreads 的信号灯注销函数不做其他动作。

2. 点灯和灭灯

```
int sem_post(sem_t * sem); //相当于解锁
```

点灯操作将信号灯值原子地加 1，表示增加一个可访问的资源。只有信号灯值大于 0，才能访问公共资源。主要用来增加信号量的值。当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程不在阻塞。

```
int sem_wait(sem_t * sem); //相当于加锁
```

```
int sem_trywait(sem_t * sem);
```

sem_wait()为灭灯操作（等待灯亮操作），主要被用来阻塞当前线程直到信号量 sem 的值大于 0，解除阻塞后将 sem 的值减一，表明公共资源经使用后减少。等待灯亮（信号灯值大于 0），然后将信号灯原子地减 1，并返回。sem_trywait() 为 sem_wait()的非阻塞版，如果信号灯计数大于 0，则原子地减 1 并返回 0，否则立即返回-1，errno 置为 EAGAIN。

3. 获取灯值

```
int sem_getvalue(sem_t * sem, int * sval);
```

读取 sem 中的灯计数，存于*sval 中，并返回 0。

4. 其他

sem_wait()被实现为取消点，而且在支持原子"比较且交换"指令的体系结构上，sem_post()是唯一能用于异步信号处理函数的 POSIX 异步信号安全的 API。

示例：sem_post 表示点灯（资源可用，V 操作），sem_wait 表示灭灯（资源不可用，P 操作）

```
#include<pthread.h>
```

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

```
#include <semaphore.h> //头文件包含
```

```
int ticketcount = 10;
```

```
sem_t lock;
```

```
void *chk1(void *args)
```

```
{
```

```
    while(1)
```

```
    {
```

```
        sem_wait(&lock); //因为要访问全局的共享变量 ticketcount，所以就要加锁
```

```
        if(ticketcount > 0) //如果有票
```

```
        {
```

```
            printf("windows1 start sale ticket!the ticket is:%d\n",ticketcount);
```



```
        ticketcount --; //则卖出一张票
        sleep(3);
        printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
    }
    else    //如果没有票了，就解锁退出
    {
        sem_post(&lock);
        break;
    }
    sem_post(&lock);
    sleep(1);    //要放到锁的外面
}
pthread_exit(NULL);
}
void *chk2(void *args)
{
    while(1)
    {
        sem_wait(&lock); //因为要访问全局的共享变量 ticketcount，所以就要加锁
        if(ticketcount > 0) //如果有票
        {
            printf("windows2 start sale ticket!the ticket is:%d\n",ticketcount);
            ticketcount --; //则卖出一张票
            sleep(3);
            printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
        }
        else    //如果没有票了，就解锁退出
        {
            sem_post(&lock);
            break;
        }
        sem_post(&lock);
        sleep(1);    //要放到锁的外面
    }
    pthread_exit(NULL);
}
main()
{
    pthread_t pthid1,pthid2;
    sem_init(&lock,0,1); //信号灯值初始为 1，表示资源可用
    pthread_create(&pthid1,NULL,chk1,NULL);
    pthread_create(&pthid2,NULL,chk2,NULL);
    pthread_join(pthid1,NULL);
    pthread_join(pthid2,NULL);
    sem_destroy(&lock);
}
```

7.6 生产者消费者问题（选修）

示例 1：链表实现如下

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <malloc.h>

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

struct node {
    int n_number;
    struct node *n_next;
} *head = NULL;

static void cleanup_handler(void *arg)
{
    printf("Cleanup handler of second thread\n");
    free(arg);
    pthread_mutex_unlock(&mtx);
}

static void *thread_func(void *arg)//消费者
{
    struct node *p = NULL;
    pthread_cleanup_push(cleanup_handler, p);
    while (1)
    {
        pthread_mutex_lock(&mtx);
        while (head == NULL){ pthread_cond_wait(&cond,&mtx);}
        p = head;
        head = head->n_next;
        printf("Got %d from front of queue\n", p->n_number);
        free(p);
        pthread_mutex_unlock(&mtx);
    }
    pthread_exit(NULL);
    pthread_cleanup_pop(0);    //必须放在最后一行
}

int main(void)
{
    pthread_t tid;
    int i;
    struct node *p;
    pthread_create(&tid, NULL, thread_func, NULL);
    for (i = 0; i < 10; i++)//生产者
    {
        p = (struct node*)malloc(sizeof(struct node));
        p->n_number = i;
```

```
pthread_mutex_lock(&mtx); //因为 head 是共享的，访问共享数据必须要加锁
p->n_next = head;
head = p;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mtx);
sleep(1);
}
printf("thread 1 wanna end the line. So cancel thread 2.\n");
pthread_cancel(tid);
pthread_join(tid, NULL);
printf("All done-----exiting\n");
return 0;
}
```

示例 2： 队列实现如下

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#define BUFFER_SIZE 16 //表示一次最多可以不间断的生产 16 个产品
struct prodcons {
    int buffer[BUFFER_SIZE]; /* 数据 */
    pthread_mutex_t lock; /* 加锁 */
    int readpos, writepos; /* 读 pos 写位置 */
    pthread_cond_t notempty; /* 不空，可以读 */
    pthread_cond_t notfull; /* 不满，可以写 */
};
/* 初始化 */
void init(struct prodcons * b)
{
    pthread_mutex_init(&b->lock, NULL); //初始化锁
    pthread_cond_init(&b->notempty, NULL); //初始化条件变量
    pthread_cond_init(&b->notfull, NULL); //初始化条件变量
    b->readpos = 0; //初始化读取位置从 0 开始
    b->writepos = 0; //初始化写入位置从 0 开始
}
/* 销毁操作 */
void destroy(struct prodcons * b)
{
    pthread_mutex_destroy(&b->lock);
    pthread_cond_destroy(&b->notempty);
    pthread_cond_destroy(&b->notfull);
}
void put(struct prodcons * b, int data) //生产者
{
    pthread_mutex_lock(&b->lock);
    while ((b->writepos + 1) % BUFFER_SIZE == b->readpos) { //判断是不是满了
        printf("wait for not full\n");
        pthread_cond_wait(&b->notfull, &b->lock); //此时为满，不能生产，等待不满的信号
    }
}
```

```
//下面表示还没有满，可以进行生产
b->buffer[b->writepos] = data;
b->writepos++;    //写入点向后移一位
if (b->writepos >= BUFFER_SIZE) b->writepos = 0; //如果到达最后，就再转到开头
pthread_cond_signal(&b->notempty); //此时有东西可以消费，发送非空的信号
pthread_mutex_unlock(&b->lock);
}
int get(struct prodcons * b) //消费者
{
    pthread_mutex_lock(&b->lock);
    while (b->writepos == b->readpos) { //判断是不是空
        printf("wait for not empty\n");
        pthread_cond_wait(&b->notempty, &b->lock); //此时为空，不能消费，等待非空信号
    }
    //下面表示还不为空，可以进行消费
    int data = b->buffer[b->readpos];
    b->readpos++;    //读取点向后移一位
    if (b->readpos >= BUFFER_SIZE) b->readpos = 0; //如果到达最后，就再转到开头
    pthread_cond_signal(&b->notfull);    //此时可以进行生产，发送不满的信号
    pthread_mutex_unlock(&b->lock);
    return data;
}
/*-----*/
#define OVER (-1)    //定义结束标志
struct prodcons buffer; //定义全局变量
/*-----*/
void * producer(void * data)
{
    int n = 0;
    for (; n < 50; n++) {
        printf(" put-->%d\n", n);
        put(&buffer, n);
    }
    put(&buffer, OVER);
    printf("producer stopped!\n");
    pthread_exit(NULL);
}
/*-----*/
void * consumer(void * data)
{
    while (1) {
        int d = get(&buffer);
        if (d == OVER ) break;
        printf(" %d-->get\n", d);
    }
    printf("consumer stopped!\n");
    pthread_exit(NULL);
}
/*-----*/
```

```
int main(void)
{
    pthread_t th_a, th_b;
    init(&buffer);
    pthread_create(&th_a, NULL, producer, 0);
    pthread_create(&th_b, NULL, consumer, 0);
    pthread_join(th_a, NULL);
    pthread_join(th_b, NULL);
    destroy(&buffer);
    return 0;
}
```

7.7 线程的属性（选修）

`pthread_create` 的第二个参数 `attr` 是一个结构体指针，结构中的元素分别指定新线程的运行属性，各成员属性为：

`__detachstate` 表示新线程是否与进程中其他线程脱离同步，如果置位则新线程不能用 `pthread_join()` 来同步，且在退出时自行释放所占用的资源。缺省为 `PTHREAD_CREATE_JOINABLE` 状态。这个属性也可以在线程创建并运行以后用 `pthread_detach()` 来设置，而一旦设置为 `PTHREAD_CREATE_DETACH` 状态（不论是创建时设置还是运行时设置）则不能再恢复到 `PTHREAD_CREATE_JOINABLE` 状态。

`__schedpolicy`，表示新线程的调度策略，主要包括 `SCHED_OTHER`（正常、非实时）、`SCHED_RR`（实时、轮转法）和 `SCHED_FIFO`（实时、先入先出）三种，缺省为 `SCHED_OTHER`，后两种调度策略仅对超级用户有效。运行时可以用过 `pthread_setschedparam()` 来改变。

`__schedparam`，一个 `sched_param` 结构，目前仅有一个 `sched_priority` 整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即 `SCHED_RR` 或 `SCHED_FIFO`）时才有效，并可以在运行时通过 `pthread_setschedparam()` 函数来改变，缺省为 0。

`__inheritsched`，有两种值可供选择：`PTHREAD_EXPLICIT_SCHED` 和 `PTHREAD_INHERIT_SCHED`，前者表示新线程使用显式指定调度策略和调度参数（即 `attr` 中的值），而后者表示继承调用者线程的值。缺省为 `PTHREAD_EXPLICIT_SCHED`。

`__scope`，表示线程间竞争 CPU 的范围，也就是说线程优先级的有效范围。POSIX 的标准中定义了两个值：`PTHREAD_SCOPE_SYSTEM` 和 `PTHREAD_SCOPE_PROCESS`，前者表示与系统中所有线程一起竞争 CPU 时间，后者表示仅与同进程中的线程竞争 CPU。目前 Linux 仅实现了 `PTHREAD_SCOPE_SYSTEM` 一值。

属性设置是由一些函数来完成的，通常调用 `pthread_attr_init` 函数进行初始化。设置绑定属性的函数为 `pthread_attr_setscope`，设置分离属性的函数是 `pthread_attr_setdetachstate`，设置线程优先级的相关函数 `pthread_attr_getschedparam`（获取线程优先级）和 `pthread_attr_setschedparam`（设置线程优先级）。再设置完成属性后，调用 `pthread_create` 函数创建线程。

- 线程属性初始化：

```
int pthread_attr_init(pthread_attr_t *attr);
```

`attr`：传出参数，表示线程属性，后面的线程属性设置函数都会用到。

返回值：成功 0，错误 -1。

- 设置绑定属性：

```
pthread_attr_setscope(pthread_attr_t *attr, int scope);
```

`attr`：线程属性

`scope`：`PTHREAD_SCOPE_SYSTEM`(绑定) `PTHREAD_SCOPE_PROCESS`(非绑定)

返回值：成功 0，错误 -1。

- 设置分离属性：

```
pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

attr: 线程属性

detachstate : PTHREAD_CREAT_DETACHED(分离) PTHREAD_CREAT_JOINABLE(非分离)

返回值: 成功 0, 错误-1。

- 获取线程优先级:

```
int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *param);
```

attr: 线程属性

param: 线程优先级

返回值: 成功 0, 错误-1。

- 设置线程优先级:

```
int pthread_attr_setschedparam(pthread_attr_t *attr, struct sched_param *param);
```

attr: 线程属性

param: 线程优先级

返回值: 成功 0, 错误-1。

示例:

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

```
void * thread_function(void * arg);
```

```
char message[] = "Hello World";
```

```
int thread_finished = 0;
```

```
int main()
```

```
{
    int res = 0;
    pthread_t a_thread;
    void * thread_result;
    pthread_attr_t thread_attr; //定义属性
    struct sched_param scheduling_value;
    res = pthread_attr_init(&thread_attr); //属性初始化
    if (res != 0)
    {
        perror("Attribute creation failed");
        exit(EXIT_FAILURE); // EXIT_FAILURE -1
    }
}
```

```
//设置调度策略
```

```
res = pthread_attr_setschedpolicy(&thread_attr, SCHED_OTHER);
if (res != 0)
{
    perror("Setting schedpolicy failed");
    exit(EXIT_FAILURE);
}
```

```
//设置脱离状态
```

```
res = pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
```

```
//创建线程
res = pthread_create(&a_thread, &thread_attr, thread_function, (void *)message);
if (res != 0) {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);
}

//获取最大优先级别
int max_priority = sched_get_priority_max(SCHED_OTHER);
//获取最小优先级
int min_priority = sched_get_priority_min(SCHED_OTHER);

//重新设置优先级别
scheduling_value.sched_priority = min_priority + 5;
//设置优先级别
res = pthread_attr_setschedparam(&thread_attr, &scheduling_value);

pthread_attr_destroy(&thread_attr);

while (!thread_finished)
{
    printf("Waiting for thread to say it's finished...\n");
    sleep(1);
}
printf("Other thread finished, bye!\n");
exit(EXIT_SUCCESS);
}

void * thread_function(void * arg)
{
    printf("thread_function is running. Argument was %s\n", (char *)arg);
    sleep(4);
    printf("Second thread setting finished flag, and exiting now\n");
    thread_finished = 1;
    pthread_exit(NULL);
}
```


第8章 LINUX 网络编程

8.1 LINUX 网络编程介绍

8.1.1 TCP/IP 协议概述

协议 protocol: 通信双方必须遵循的规矩 由 iso 规定 rpc 文档

osi 参考模型: （应-表-会-传-网-数-物）

➔ 应用层 表示层 会话层 传输层 网络层 数据链路层 物理层

tcp/ip 模型 4 层:

应用层{http 超文本传输协议 ftp 文件传输协议 telnet 远程登录 ssh 安全外壳协议 stmp 简单邮件发送 pop3 收邮件}

传输层{tcp 传输控制协议,udp 用户数据包协议}

网络层{ip 网际互联协议 icmp 网络控制消息协议 igmp 网络组管理协议}

网络接口层{arp 地址转换协议,rarp 方向地址转换协议, mpls 多协议标签交换}

TCP 协议: 传输控制协议 面向连接的协议 能保证传输安全可靠 速度慢（有 3 次握手）

UDP 协议: 用户数据包协议 非面向连接 速度快 不可靠

通常是 ip 地址后面跟上端口号: ip 用来定位主机 port 区别应用（进程）

http 的端口号 80 ssh-->22 telnet-->23 ftp-->21 用户自己定义的通常要大于 1024

8.1.2 OSI 参考模型及 TCP/IP 参考模型

其模型如图 8.1 和图 8.2 所示:

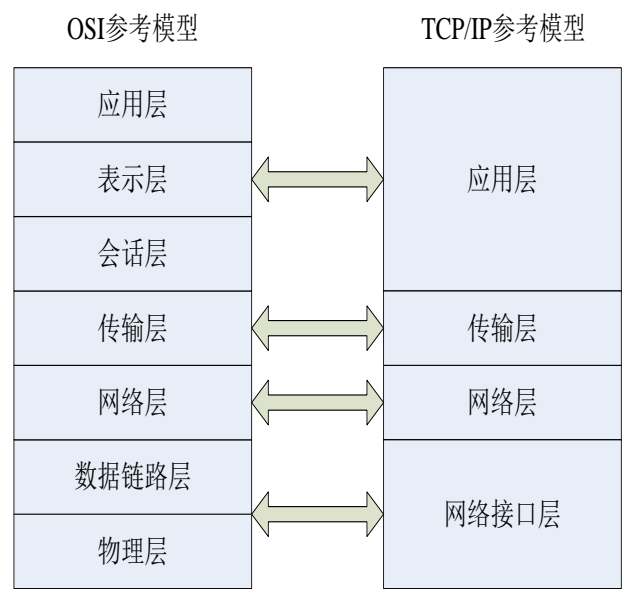


图 8.1

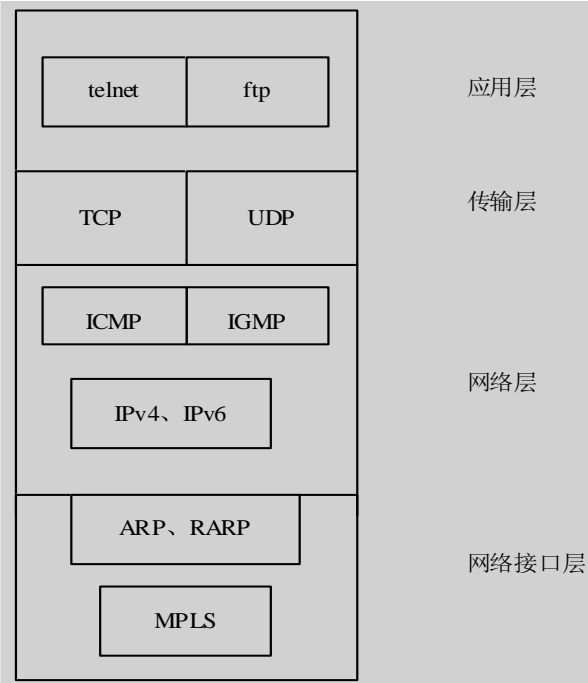


图 8.2

TCP/IP 协议族的每一层的作用:

- 网络接口层: 负责将二进制流转换为数据帧, 并进行数据帧的发送和接收。要注意的是数据帧是独立的网络信息传输单元。
- 网络层: 负责将数据帧封装成 IP 数据报, 并运行必要的路由算法。
- 传输层: 负责端对端之间的通信会话连接和建立。传输协议的选择根据数据传输方式而定。

- 应用层：负责应用程序的网络访问，这里通过端口号来识别各个不同的进程。
- TCP/IP 协议族的每一层协议的相关注解如下，各层关系如图 8.3 和图 8.4 所示。
- ARP：（地址转换协议）用于获得同一物理网络中的硬件主机地址。
 - MPLS：（多协议标签交换）很有发展前景的下一代网络协议。
 - IP：（网际互联协议）负责在主机和网络之间寻址和路由数据包。
 - ICMP：（网络控制消息协议）用于发送报告有关数据包的传送错误的协议。
 - IGMP：（网络组管理协议）被 IP 主机用来向本地多路广播路由器报告主机组成员的协议。
 - TCP：（传输控制协议）为应用程序提供可靠的通信连接。适合于一次传输大批数据的情况。并适用于要求得到相应的应用程序。
 - UDP：（用户数据包协议）提供了无连接通信，且不对传送包进行可靠的保证。适合于一次传输少量数据。

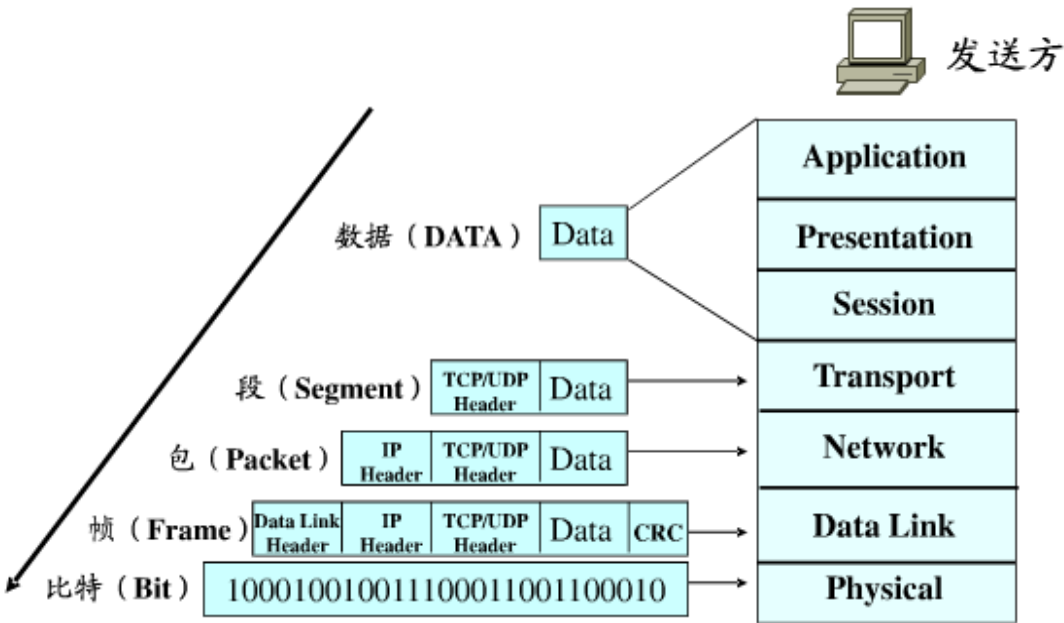


图 8.3

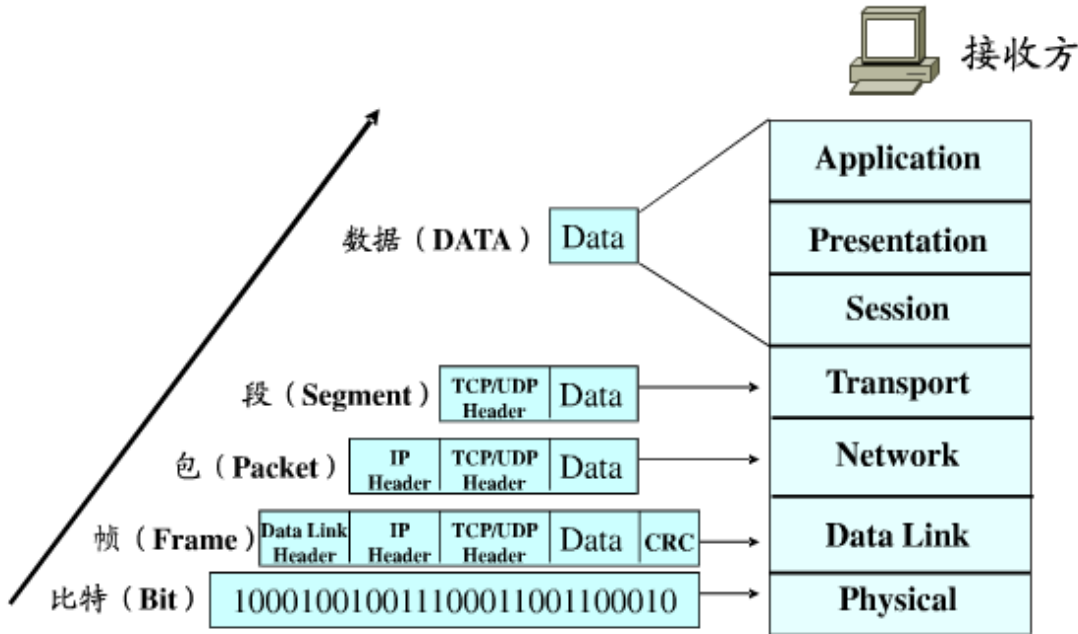


图 8.4

8.1.3 TCP 协议

(1) 概述

TCP 是 TCP/IP 体系中面向连接的运输层协议，它提供全双工和可靠交付的服务。它采用许多机制来确保端到端结点之间的可靠数据传输，如采用序列号、确认重传、滑动窗口等。

首先，TCP 要为所发送的每一个报文段加上序列号，保证每一个报文段能被接收方接收，并只被正确的接收一次。

其次，TCP 采用具有重传功能的积极确认技术作为可靠数据流传输服务的基础。这里“确认”是指接收端在正确收到报文段之后向发送端回送一个确认（ACK）信息。发送方将每个已发送的报文段备份在自己的缓冲区里，而且在收到相应的确认之前是不会丢弃所保存的报文段的。“积极”是指发送方在每一个报文段发送完毕的同时启动一个定时器，加入定时器的定时期满而关于报文段的确认信息还没有达到，则发送方认为该报文段已经丢失并主动重发。为了避免由于网络延时引起迟到的确认和重复的确认，TCP 规定在确认信息中捎带一个报文段的序号，使接收方能正确的将报文段与确认联系起来。

最后，采用可变长的滑动窗口协议进行流量控制，以防止由于发送端与接收端之间的不匹配而引起的数据丢失。这里所采用的滑动窗口协议与数据链路层的滑动窗口协议在工作原理上完全相同，唯一的区别在于滑动窗口协议用于传输层是为了在端对端节点之间实现流量控制，而用于数据链路层是为了在相邻节点之间实现流量控制。TCP 采用可变长的滑动窗口，使得发送端与接收端可根据自己的 CPU 和数据缓存资源对数据发送和接收能力来进行动态调整，从而灵活性更强，也更合理。

三次握手协议

在利用 TCP 实现源主机和目的主机通信时，目的主机必须同意，否则 TCP 连接无法建立。为了确保 TCP 连接的成功建立，TCP 采用了一种称为三次握手的方式，三次握手方式使得“序号/确认号”系统能够正常工作，从而使它们的序号达成同步。如果三次握手成功，则连接建立成功，可以开始传送数据信息。

其三次握手分别为：

1) 源主机 A 的 TCP 向主机 B 发送连接请求报文段，其首部中的 SYN（同步）标志位位置为 1，表示想跟目标主机 B 建立连接，进行通信，并发送一个同步序列号 X（例：SEQ=100）进行同步，表明在后面传送数据时的第一个数据字节的序号为 X+1（即 101）。

2) 目标主机 B 的 TCP 收到连接请求报文段后，如同意，则发回确认。再确认报中应将 ACK 位和 SYN 位置为 1。确认号为 X+1，同时也为自己选择一个序号 Y。

3) 源主机 A 的 TCP 收到目标主机 B 的确认后要想目标主机 B 给出确认。其 ACK 置为 1，确认号为 Y+1，而自己的序号为 X+1。TCP 的标准规定，SYN 置 1 的报文段要消耗掉一个序号。

运行客户进程的源主机 A 的 TCP 通知上层应用进程，连接已经建立。当源主机 A 向目标主机 B 发送第一个数据报文段时，其序号仍为 X+1，因为前一个确认报文段并不消耗序号。

当运行服务进程的目标主机 B 的 TCP 收到源主机 A 的确认后，也通知其上层应用进程，连接已经建立。至此建立了一个全双工的连接，如图 8.5 所示。

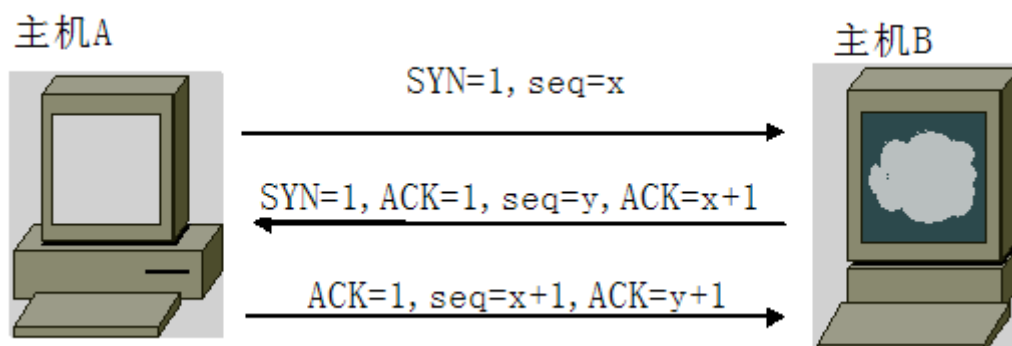


图 8.5

三次握手：为应用程序提供可靠的通信连接。适合于一次传输大批数据的情况。并适用于要求得到响应的应用程序。

(2) TCP 数据报头如图 8.6 所示



图 8.6 TCP 头信息

- 源端口、目的端口：16 位长。标识出远端和本地的端口号。
- 序号：32 位长。标识发送的数据报的顺序。
- 确认号：32 位长。希望收到的下一个数据报的序列号。
- TCP 头长：4 位长。表明 TCP 头中包含多少个 32 位字。
- 6 位未用。
- ACK：ACK 位置 1 表明确认号是合法的。如果 ACK 为 0，那么数据报不包含确认信息，确认字段被省略。
- PSH：表示是带有 PUSH 标志的数据。接收方因此请求数据报一到便可送往应用程序而不必等到缓冲区装满时才发送。
- RST：用于复位由于主机崩溃或其他原因而出现的错误的连接。还可以用于拒绝非法的数据报或拒绝连接请求。
- SYN：用于建立连接。
- FIN：用于释放连接。
- 窗口大小：16 位长。窗口大小字段表示在确认了字节之后还可以发送多少个字节。
- 校验和：16 位长。是为了确保高可靠性而设置的。它校验头部、数据和伪 TCP 头部之和。
- 可选项：0 个或多个 32 位字。包括最大 TCP 载荷，窗口比例、选择重复数据报等选项。

8.1.4 UDP 协议

(1) 概述

UDP 即用户数据报协议，它是一种无连接协议，因此不需要像 TCP 那样通过三次握手来建立一个连接。同时，一个 UDP 应用可同时作为应用的客户或服务方。由于 UDP 协议并不需要建立一个明确的连接，因此建立 UDP 应用要比建立 TCP 应用简单得多。

它比 TCP 协议更为高效，也能更好地解决实时性的问题。如今，包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都使用 UDP 协议。

(2) Udp 数据包头格式如图 8.7 所示。

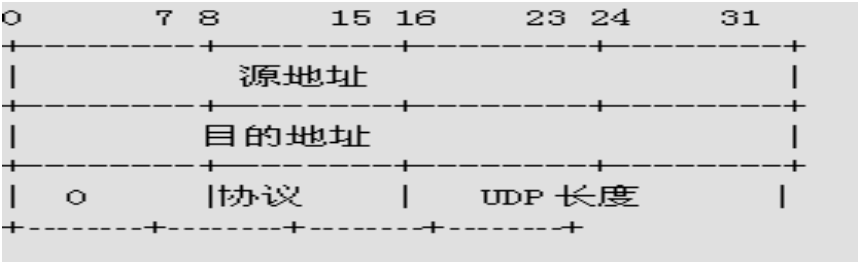


图 8.7

8.1.5 协议的选择

(1) 对数据可靠性的要求

对数据要求高可靠性的应用需选择 TCP 协议，如验证、密码字段的传送都是不允许出错的，而对数据的可靠性要求不那么高的应用可选择 UDP 传送。

(2) 应用的实时性

TCP 协议在传送过程中要使用三次握手、重传确认等手段来保证数据传输的可靠性。使用 TCP 协议会有较大的时延，因此不适合对实时性要求较高的应用，如 VOIP、视频监控等。相反，UDP 协议则在这些应用中能发挥很好的作用。

(3) 网络的可靠性

由于 TCP 协议的提出主要是解决网络的可靠性问题，它通过各种机制来减少错误发生的概率。因此，在网络状况不是很好的情况下需选用 TCP 协议（如在广域网等情况），但是若在网络状况很好的情况下（如局域网等）就不需要再采用 TCP 协议，而建议选择 UDP 协议来减少网络负荷。

8.2 网络相关概念

(1) 套接口的概念：

套接口，也叫“套接字”。是操作系统内核中的一个数据结构，它是网络中的节点进行相互通信的门户。它是网络进程的 ID。网络通信，归根到底还是进程间的通信（不同计算机上的进程间通信）。在网络中，每一个节点（计算机或路由）都有一个网络地址，也就是 IP 地址。两个进程通信时，首先要确定各自所在的网络节点的网络地址。但是，网络地址只能确定进程所在的计算机，而一台计算机上很可能同时运行着多个进程，所以仅凭网络地址还不能确定到底是和网络中的哪一个进程进行通信，因此套接口中还需要包括其他的信息，也就是端口号（PORT）。在一台计算机中，一个端口号一次只能分配给一个进程，也就是说，在一台计算机中，端口号和进程之间是一一对应关系。所以，使用端口号和网络地址的组合可以唯一的确定整个网络中的一个网络进程。

例如，如网络中某一台计算机的 IP 为 10.92.20.160，操作系统分配给计算机中某一应用程序进程的端口号为 1500，则此时 10.92.20.160 1500 就构成了一个套接口。

(2) 端口号的概念：

在网络技术中，端口大致有两种意思：一是物理意义上的端口，如集线器、交换机、路由器等用于连接其他网络设备的接口。二是指 TCP/IP 协议中的端口，端口号的范围从 0~65535，一类是由互联网指派名字和号码公司 ICANN 负责分配给一些常用的应用程序固定使用的“周知的端口”，其值一般为 0~1023。例如 http 的端口号是 80，ftp 为 21，ssh 为 22，telnet 为 23 等。还有一类是用户自己定义的，通常是大于 1024 的整型值。

(3) ip 地址的表示：

通常用户在表达 IP 地址时采用的是点分十进制表示的数值（或者是为冒号分开的十进制 Ipv6 地址），而在通常使用的 socket 编程中使用的则是二进制值，这就需要将这两个数值进行转换。

ipv4 地址：32bit, 4 字节，通常采用点分十进制记法。

例如对于：10000000 00001011 00000011 00011111

点分十进制表示为：128.11.3.31

ip 地址的分类如图 8.8 所示：

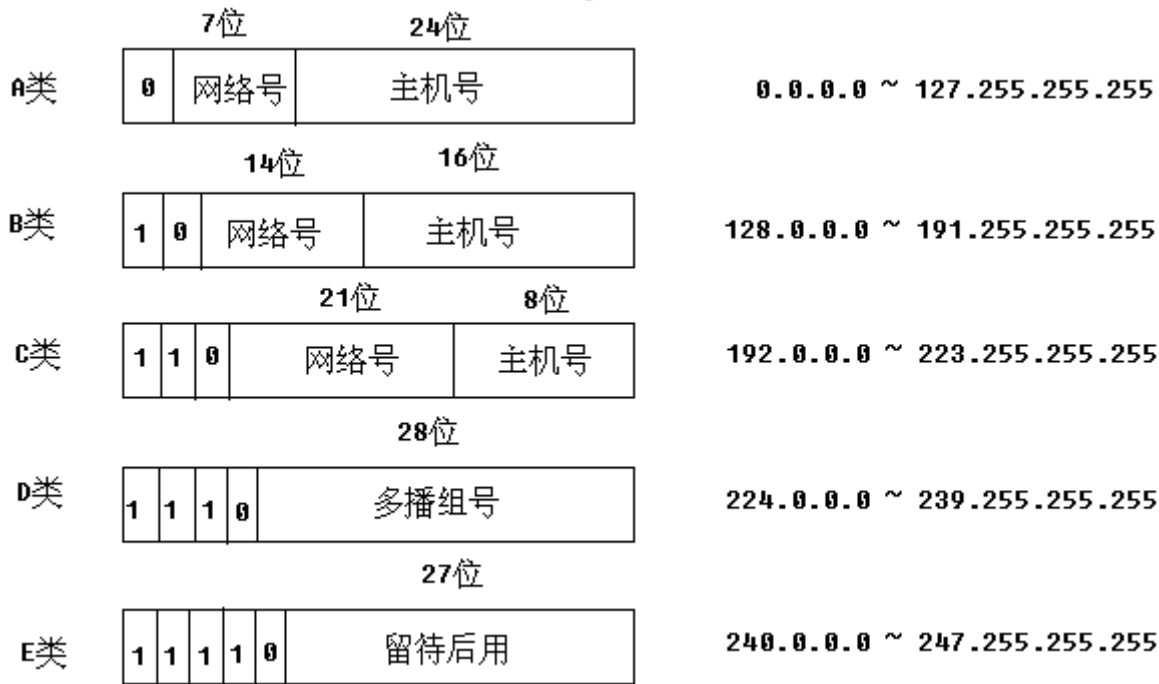


图 8.8

特殊的 ip 地址如图 8.9 所示：

网络部分	主机部分	地址类型	用途
Any	全“0”	网络地址	代表一个网段
Any	全“1”	广播地址	特定网段的所有节点
127	any	环回地址	环回测试
全“0”		所有网络	港湾路由器用于指定默认路由
全“1”		广播地址	本网段所有节点

图 8.9

8.2.1 socket 概念

Linux 中的网络编程是通过 socket 接口来进行的。socket 是一种特殊的 I/O 接口，它也是一种文件描述符。它是一种常用的进程之间通信机制，通过它不仅能实现本地机器上的进程之间的通信，而且通过网络能够在不同机器上的进程之间进行通信。

每一个 socket 都用一个半相关描述{协议、本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述{协议、本地地址、本地端口、远程地址、远程端口}来表示。socket 也有一个类似于打开文件的函数调用，该函数返回一个整型的 socket 描述符，随后的连接建立、数据传输等操作都是通过 socket 来实现的；

8.2.2 socket 类型

- (1) 流式 socket (SOCK_STREAM) →用于 TCP 通信

流式套接字提供可靠的、面向连接的通信流；它使用 TCP 协议，从而保证了数据传输的正确性和顺序性。

(2) 数据报 socket (SOCK_DGRAM) → 用于 UDP 通信

数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证是可靠、无差错的。它使用数据报协议 UDP。

(3) 原始 socket (SOCK_RAW) → 用于新的网络协议实现的测试等

原始套接字允许对底层协议如 IP 或 ICMP 进行直接访问，它功能强大但使用较为不便，主要用于一些协议的开发。

8.2.3 socket 信息数据结构

```
struct sockaddr
{
    unsigned short sa_family;    /*地址族*/
    char sa_data[14];           /*14 字节的协议地址，包含该 socket 的 IP 地址和端口号。*/
};
struct sockaddr_in
{
    short int sa_family;        /*地址族*/
    unsigned short int sin_port; /*端口号*/
    struct in_addr sin_addr;     /*IP 地址*/
    unsigned char sin_zero[8];  /*填充 0 以保持与 struct sockaddr 同样大小*/
};
struct in_addr
{
    unsigned long int s_addr;    /* 32 位 IPv4 地址，网络字节序 */
};
```

头文件<netinet/in.h>

sa_family:AF_INET →IPv4 协议, AF_INET6 →IPv6 协议

8.2.4 数据存储优先顺序的转换

计算机数据存储有两种字节优先顺序：高位字节优先（称为大端模式）和低位字节优先（称为小端模式）。内存的低地址存储数据的低字节，高地址存储数据的高字节的方式叫小端模式。内存的高地址存储数据的低字节，低地址存储数据高字节的方式称为大端模式。

示例：对于内存中存放的数 0x12345678 来说

如果是采用大端模式存放的，则其真实的数是：0x12345678

如果是采用小端模式存放的，则其真实的数是：0x78563412

如果称某个系统所采用的字节序为主机字节序，则它可能是小端模式的，也可能是大端模式的。而端口号和 IP 地址都是以网络字节序存储的，不是主机字节序，网络字节序都是大端模式。要把主机字节序和网络字节序相互对应起来，需要对这两个字节存储优先顺序进行相互转化。这里用到四个函数：htons(), ntohs(), htonl() 和 ntohl()。这四个地址分别实现网络字节序和主机字节序的转化，这里的 h 代表 host, n 代表 network, s 代表 short, l 代表 long。通常 16 位的 IP 端口号用 s 代表，而 IP 地址用 l 来代表。

函数原型如图 8.10 所示：

所需头文件	#include <netinet/in.h>
函数原型	uint16_t htons(uint16_t host16bit)+ uint32_t htonl(uint32_t host32bit)+ uint16_t ntohs(uint16_t net16bit)+ uint32_t ntohs(uint32_t net32bit)
函数传入值	host16bit: 主机字节序的 16bit 数据
	host32bit: 主机字节序的 32bit 数据
	net16bit: 网络字节序的 16bit 数据
	net32bit: 网络字节序的 32bit 数据
函数返回值	成功: 返回要转换的字节序
	出错: -1

图 8.10

8.2.5 地址格式转化

通常用户在表达地址时采用的是点分十进制表示的数值（或者是为冒号分开的十进制 Ipv6 地址），而在通常使用的 socket 编程中使用的则是 32 位的网络字节序的二进制值,这就需要将这两个数值进行转换。这里在 Ipv4 中用到的函数有 inet_aton()、inet_addr()和 inet_ntoa(), 而 IPV4 和 Ipv6 兼容的函数有 inet_pton()和 inet_ntop()。

IPv4 的函数原型:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *straddr, struct in_addr *addrptr);
char *inet_ntoa(struct in_addr inaddr);
in_addr_t inet_addr(const char *straddr);
```

函数 inet_aton(): 将点分十进制数的 IP 地址转换成为网络字节序的 32 位二进制数值。返回值: 成功, 则返回 1, 不成功返回 0.

参数 straddr: 存放输入的点分十进制数 IP 地址字符串。

参数 addrptr: 传出参数, 保存网络字节序的 32 位二进制数值。

函数 inet_ntoa(): 将网络字节序的 32 位二进制数值转换为点分十进制的 IP 地址。

函数 inet_addr(): 功能与 inet_aton 相同, 但是结果传递的方式不同。inet_addr()若成功则返回 32 位二进制的网络字节序地址。

IPv4 和 IPv6 的函数原型:

```
#include <arpa/inet.h>
int inet_pton(int family, const char *src, void *dst);
const char *inet_ntop(int family, const void *src, char *dst, socklen_t len);
```

函数 inet_pton 跟 inet_aton 实现的功能类似, 只是多了 family 参数, 该参数指定为 AF_INET, 表示是 IPv4 协议, 如果是 AF_INET6, 表示 IPv6 协议。

函数 inet_ntop 跟 inet_ntoa 类似, 其中 len 表示表示转换之后的长度（字符串的长度）。

示例:

```
#include <stdio.h>
#include <sys / socket.h>
#include <netinet / in.h>
#include <arpa / inet.h>
#include <sys / socket.h>
#include <netinet / in.h>
#include <arpa / inet.h>
```

```
int main()
{
    char ip[] = "192.168.1.101";

    struct in_addr myaddr;
    /* inet_aton */
    int iRet = inet_aton(ip, &myaddr);
    printf("%x\n", myaddr.s_addr);

    /* inet_addr */
    printf("%x\n", inet_addr(ip));

    /* inet_pton */
    iRet = inet_pton(AF_INET, ip, &myaddr);
    printf("%x\n", myaddr.s_addr);

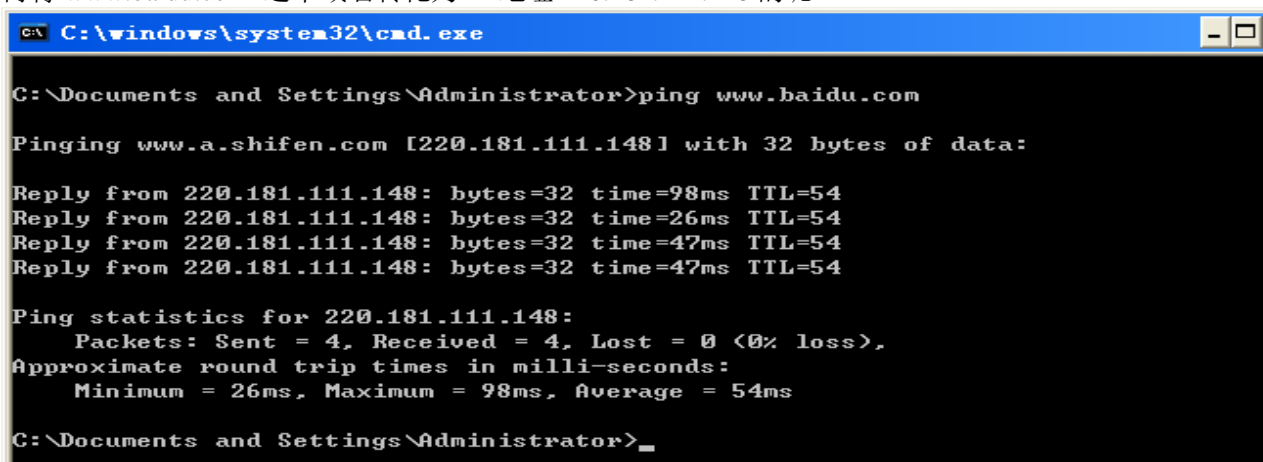
    myaddr.s_addr = 0xac100ac4;
    /* inet_ntoa */
    printf("%s\n", inet_ntoa(myaddr));

    /* inet_ntop */
    inet_ntop(AF_INET, &myaddr, ip, 16);
    puts(ip);
    return 0;
}
```

8.2.6 名字地址转化

通常，人们在使用过程中都不愿意记忆冗长的 IP 地址，尤其到 Ipv6 时，地址长度多达 128 位，那时就更加不可能一次性记忆那么长的 IP 地址了。因此，使用主机名或域名将会是很好的选择。

众所周知，百度的域名为：www.baidu.com，而这个域名其实对应了一个百度公司的 IP 地址，那么百度公司的 IP 地址是多少呢？我们可以利用 ping www.baidu.com 来得到百度公司的 ip 地址，如图 8.11 所示。那么，系统是如何将 www.baidu.com 这个域名转化为 IP 地址 220.181.111.148 的呢？



```
C:\windows\system32\cmd.exe

C:\Documents and Settings\Administrator>ping www.baidu.com

Pinging www.a.shifen.com [220.181.111.148] with 32 bytes of data:

Reply from 220.181.111.148: bytes=32 time=98ms TTL=54
Reply from 220.181.111.148: bytes=32 time=26ms TTL=54
Reply from 220.181.111.148: bytes=32 time=47ms TTL=54
Reply from 220.181.111.148: bytes=32 time=47ms TTL=54

Ping statistics for 220.181.111.148:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 26ms, Maximum = 98ms, Average = 54ms

C:\Documents and Settings\Administrator>
```

图 8.11

在 linux 中，有一些函数可以实现主机名和地址的转化，最常见的有 gethostbyname()、gethostbyaddr()等，它们都可以实现 IPv4 和 IPv6 的地址和主机名之间的转化。其中 gethostbyname()是将主机名转化为 IP 地址，gethostbyaddr()则是逆操作，是将 IP 地址转化为主机名。

函数原型：

深圳信盈达科技有限公司 专业提供单片机、嵌入式、ARM、LINUX、Android、PCB、FPGA 等技术培训、技术方案。

```
#include <netdb.h>
```

```
struct hostent* gethostbyname(const char* hostname);
```

```
struct hostent* gethostbyaddr(const char* addr, size_t len, int family);
```

结构体:

```
struct hostent
```

```
{
```

```
    char *h_name;    /*正式主机名*/
```

```
    char **h_aliases; /*主机别名*/
```

```
    int h_addrtype;    /*主机 IP 地址类型 IPv4 为 AF_INET*/
```

```
    int h_length;    /*主机 IP 地址字节长度, 对于 IPv4 是 4 字节, 即 32 位*/
```

```
    char **h_addr_list; /*主机的 IP 地址*/
```

```
}
```

函数 `gethostbyname()`: 用于将域名 (www.baidu.com) 或主机名转换为 IP 地址。参数 `hostname` 指向存放域名或主机名的字符串。

函数 `gethostbyaddr()`: 用于将 IP 地址转换为域名或主机名。参数 `addr` 是一个 IP 地址, 此时这个 ip 地址不是普通的字符串, 而是要通过函数 `inet_aton()` 转换。`len` 为 IP 地址的长度, `AF_INET` 为 4。`family` 可用 `AF_INET`: Ipv4 或 `AF_INET6`: Ipv6。

示例: 将百度的 www.baidu.com 转换为 ip 地址

```
#include <netdb.h>
```

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

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

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

```
{
```

```
    char * ptr, ** pptr;
```

```
    struct hostent * hptr;
```

```
    char str[32] = {'\0'};
```

```
    /* 取得命令后第一个参数, 即要解析的域名或主机名 */
```

```
    ptr = argv[1]; //如 www.baidu.com
```

```
    /* 调用 gethostbyname()。结果存在 hptr 结构中 */
```

```
    if ((hptr = gethostbyname(ptr)) == NULL)
```

```
    {
```

```
        printf(" gethostbyname error for host:%s\n", ptr);
```

```
        return 0;
```

```
    }
```

```
    /* 将主机的规范名打出来 */
```

```
    printf("official hostname:%s\n", hptr->h_name);
```

```
    /* 主机可能有多个别名, 将所有别名分别打出来 */
```

```
    for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
```

```
        printf(" alias:%s\n", *pptr);
```

```
    /* 根据地址类型, 将地址打出来 */
```

```
    switch (hptr->h_addrtype)
```

```
    {
```

```
        case AF_INET:
```

```
        case AF_INET6:
```

```
            pptr = hptr->h_addr_list;
```

```
            /* 将刚才得到的所有地址都打出来。其中调用了 inet_ntop()函数 */
```

```
            for (; *pptr != NULL; pptr++)
```

```
                printf(" address:%s\n", inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
```

```
printf(" first address: %s\n", inet_ntop(hptr->h_addrtype, hptr->h_addr, str, sizeof(str)));
break;
default:
printf("unknown address type\n");
break;
}
return 0;
}
```

编译运行

```
#gcc test.c
#./a.out www.baidu.com
official hostname:www.a.shifen.com
alias:www.baidu.com
address: 220.181.111.148
.....
first address: 220.181.111.148
```

8.3 socket 编程

8.3.1 使用 TCP 协议的流程图

TCP 通信的基本步骤如图 8.12 所示：
服务端： socket---bind---listen---while(1){---accept---recv---send---close---}---close
客户端： socket-----connect---send---recv-----close

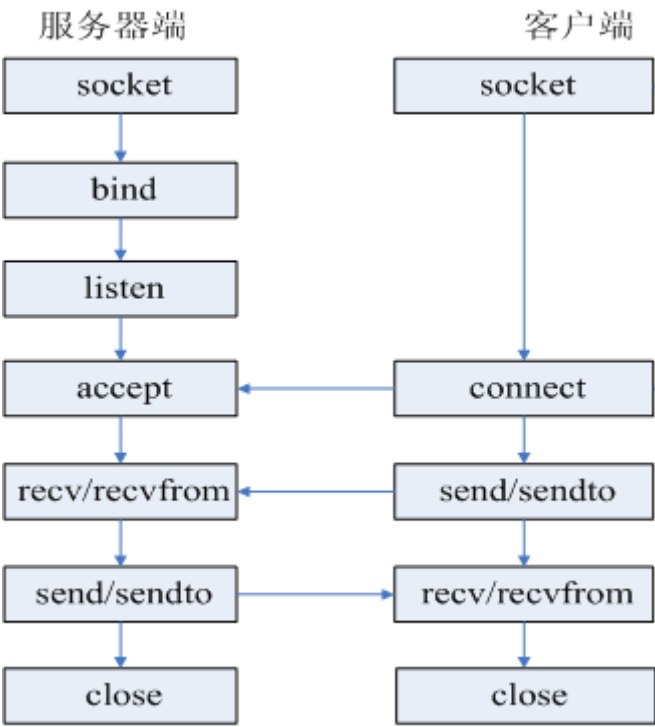


图 8.12

服务器端：

- 1. 头文件包含：

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
```

2. socket 函数：生成一个套接口描述符。

原型：int socket(int domain,int type,int protocol);

参数：domain→{ AF_INET: Ipv4 网络协议 AF_INET6: Ipv6 网络协议 }

type→ { tcp: SOCK_STREAM udp: SOCK_DGRAM }

protocol→指定 socket 所使用的传输协议编号。通常为 0。

返回值：成功则返回套接口描述符，失败返回-1。

常用实例：

```
int sfd = socket(AF_INET, SOCK_STREAM, 0);
if(sfd == -1){perror("socket");exit(-1);}
```

3. bind 函数：用来绑定一个端口号和 IP 地址，使套接口与指定的端口号和 IP 地址相关联。

原型：int bind(int sockfd,struct sockaddr * my_addr,int addrlen);

参数：sockfd→为前面 socket 的返回值。

my_addr→为结构体指针变量

对于不同的 socket domain 定义了一个通用的数据结构

```
struct sockaddr //此结构体不常用
{
    unsigned short int sa_family; //调用 socket（）时的 domain 参数，即 AF_INET 值。
    char sa_data[14]; //最多使用 14 个字符长度
};
```

此 sockaddr 结构会因使用不同的 socket domain 而有不同结构定义，

例如使用 AF_INET domain，其 sockaddr 结构定义便为

```
struct sockaddr_in //常用的结构体
{
    unsigned short int sin_family; //即为 sa_family →AF_INET
    uint16_t sin_port; //为使用的 port 编号
    struct in_addr sin_addr; //为 IP 地址
    unsigned char sin_zero[8]; //未使用
};
struct in_addr
{
    uint32_t s_addr;
};
```

addrlen→sockaddr 的结构体长度。通常是计算 sizeof(struct sockaddr);

返回值：成功则返回 0，失败返回-1

常用实例：

```
struct sockaddr_in my_addr; //定义结构体变量
memset(&my_addr, 0, sizeof(struct sockaddr)); //将结构体清空
//或 bzero(&my_addr, sizeof(struct sockaddr));
my_addr.sin_family = AF_INET; //表示采用 Ipv4 网络协议
```


my_addr.sin_port = htons(8888); //表示端口号为 8888, 通常是大于 1024 的一个值。

//htons()用来将参数指定的 16 位 hostshort 转换成网络字符顺序

my_addr.sin_addr.s_addr = inet_addr("192.168.0.101"); // inet_addr()用来将 IP 地址字符串转换成网络所使用的二进制数字, 如果为 INADDR_ANY, 这表示服务器自动填充本机 IP 地址。

```
if(bind(sfd, (struct sockaddr*)&my_str, sizeof(struct socket)) == -1)
```

```
{perror("bind");close(sfd);exit(-1);}
```

(注: 通过将 my_addr.sin_port 置为 0, 函数会自动为你选择一个未占用的端口来使用。同样, 通过将 my_addr.sin_addr.s_addr 置为 INADDR_ANY, 系统会自动填入本机 IP 地址。)

4. listen 函数: 使服务器的这个端口和 IP 处于监听状态, 等待网络中某一客户机的连接请求。如果客户端有连接请求, 端口就会接受这个连接。

原型: int listen(int sockfd, int backlog);

参数: sockfd→为前面 socket 的返回值.即 sfd

backlog→指定同时能处理的最大连接要求, 通常为 10 或者 5。最大值可设至 128

返回值: 成功则返回 0, 失败返回-1

常用实例:

```
if(listen(sfd, 10) == -1)
```

```
{perror("listen");close(sfd);exit(-1);}
```

5. accept 函数: 接受远程计算机的连接请求, 建立起与客户机之间的通信连接。服务器处于监听状态时, 如果某时刻获得客户机的连接请求, 此时并不是立即处理这个请求, 而是将这个请求放在等待队列中, 当系统空闲时再处理客户机的连接请求。当 accept 函数接受一个连接时, 会返回一个新的 socket 标识符, 以后的数据传输和读取就要通过这个新的 socket 编号来处理, 原来参数中的 socket 也可以继续使用, 继续监听其它客户机的连接请求。(也就是说, 类似于移动营业厅, 如果有客户打电话给 10086, 此时服务器就会请求连接, 处理一些事务之后, 就通知一个话务员接听客户的电话, 也就是说, 后面的所有操作, 此时已经于服务器没有关系, 而是话务员跟客户的交流。对应过来, 客户请求连接我们的服务器, 我们服务器先做了一些绑定和监听等等操作之后, 如果允许连接, 则调用 accept 函数产生一个新的套接字, 然后用这个新的套接字跟我们的客户进行收发数据。也就是说, 服务器跟一个客户端连接成功, 会有两个套接字。)

原型: int accept(int s, struct sockaddr * addr, int * addrlen);

参数: s→为前面 socket 的返回值.即 sfd

addr→为结构体指针变量, 和 bind 的结构体是同种类型的, 系统会把远程主机的信息(远程主机的地址和端口号信息)保存到这个指针所指的结构体中。

addrlen→表示结构体的长度, 为整型指针

返回值: 成功则返回新的 socket 处理代码 cfd, 失败返回-1

常用实例:

```
struct sockaddr_in clientaddr;
```

```
memset(&clientaddr, 0, sizeof(struct sockaddr));
```

```
int addrlen = sizeof(struct sockaddr);
```

```
int cfd = accept(sfd, (struct sockaddr *) & clientaddr, &addrlen);
```

```
if (cfd == -1)
```

```
{perror("accept");close(sfd);exit(-1);}
```

```
printf("%s %d success connect\n", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));
```

6. recv 函数: 用新的套接字来接收远端主机传来的数据, 并把数据存到由参数 buf 指向的内存空间

原型: int recv(int sockfd, void *buf, int len, unsigned int flags);

参数: sockfd→为前面 accept 的返回值.即 cfd, 也就是新的套接字。

buf→表示缓冲区

len→表示缓冲区的长度

flags→通常为 0

返回值：成功则返回实际接收到的字符数，可能会少于你所指定的接收长度。失败返回-1

常用实例：

```
char buf[512] = {0};
if(recv(cfd, buf, sizeof(buf), 0) == -1)
    {perror("recv");close(cfd);close(sfd);exit(-1);}
puts(buf);
```

7. send 函数：用新的套接字发送数据给指定的远端主机

原型：int send(int s,const void * msg,int len,unsigned int flags);

参数：s→为前面 accept 的返回值.即 cfd

msg→一般为常量字符串

len→表示长度

flags→通常为 0

返回值：成功则返回实际传送出去的字符数，可能会少于你所指定的发送长度。失败返回-1

常用实例：

```
if(send(cfd, "hello", 6, 0) == -1)
    {perror("send");close(cfd);close(sfd);exit(-1);}
```

8. close 函数：当使用完文件后若已不再需要则可使用 close()关闭该文件，并且 close()会让数据写回磁盘，并释放该文件所占用的资源

原型：int close(int fd);

参数：fd→为前面的 sfd,cfd

返回值：若文件顺利关闭则返回 0，发生错误时返回-1

常用实例： close(cfd); close(sfd);

客户端：

1. connect 函数：用来请求连接远程服务器，将参数 sockfd 的 socket 连至参数 serv_addr 指定的服务器 IP 和端口号上去。

原型：int connect (int sockfd,struct sockaddr * serv_addr,int addrlen);

参数：sockfd→为前面 socket 的返回值，即 sfd

serv_addr→为结构体指针变量，存储着远程服务器的 IP 与端口号信息。

addrlen→表示结构体变量的长度

返回值：成功则返回 0，失败返回-1

常用实例：

```
struct sockaddr_in seraddr;//请求连接服务器
memset(&seraddr, 0, sizeof(struct sockaddr));
seraddr.sin_family = AF_INET;
seraddr.sin_port = htons(8888); //服务器的端口号
seraddr.sin_addr.s_addr = inet_addr("192.168.0.101"); //服务器的 ip
if (connect(sfd, (struct sockaddr *) & seraddr, sizeof(struct sockaddr)) == -1)
    {perror("connect");close(sfd);exit(-1);}
```

将上面的头文件以及各个函数中的代码全部拷贝就可以形成一个完整的例子，此处省略。

示例：将一些通用的代码全部封装起来，以后要用直接调用函数即可。如下：

通用网络封装代码头文件： tcp_net_socket.h

```
#ifndef __TCP__NET__SOCKET__H
#define __TCP__NET__SOCKET__H
```

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

深圳信盈达科技有限公司 专业提供单片机、嵌入式、ARM、LINUX、Android、PCB、FPGA 等技术培训、技术方案。

```
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <signal.h>

extern int tcp_init(const char* ip,int port);
extern int tcp_accept(int sfd);
extern int tcp_connect(const char* ip,int port);
extern void signalhandler(void);

#endif
```

具体的通用函数封装如下： tcp_net_socket.c

```
#include "tcp_net_socket.h"
int tcp_init(const char* ip, int port)    //用于初始化操作
{
    int sfd = socket(AF_INET, SOCK_STREAM, 0); //首先创建一个 socket，向系统申请
    if(sfd == -1)
    {
        perror("socket");
        exit(-1);
    }
    struct sockaddr_in serveraddr;
    memset(&serveraddr, 0, sizeof(struct sockaddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(port);
    serveraddr.sin_addr.s_addr = inet_addr(ip); //或 INADDR_ANY
    if(bind(sfd, (struct sockaddr*)&serveraddr, sizeof(struct sockaddr)) == -1)
        //将新的 socket 与制定的 ip、port 绑定
    {
        perror("bind");
        close(sfd);
        exit(-1);
    }
    if(listen(sfd, 10) == -1) //监听它，并设置其允许最大的连接数为 10 个
    {
        perror("listen");
        close(sfd);
        exit(-1);
    }
    return sfd;
}
```

```
int tcp_accept(int sfd)    //用于服务端的接收
{
```

```
    struct sockaddr_in clientaddr;
    memset(&clientaddr, 0, sizeof(struct sockaddr));
    int addrlen = sizeof(struct sockaddr);
    int cfd = accept(sfd, (struct sockaddr*)&clientaddr, &addrlen);
    //sfd 接受客户端连接，并创建新的 socket 为 cfd，将请求连接的客户端的 ip、port 保存在结构体中
    if(cfd == -1)
    {
        perror("accept");
        close(cfd);
        exit(-1);
    }
    printf("%s %d success connect...\n",inet_ntoa(clientaddr.sin_addr),ntohs(clientaddr.sin_port));
    return cfd;
}
```

```
int tcp_connect(const char* ip, int port)    //用于客户端的连接
{
    int sfd = socket(AF_INET, SOCK_STREAM, 0); //向系统注册申请新的 socket
    if(sfd == -1)
    {
        perror("socket");
        exit(-1);
    }
    struct sockaddr_in serveraddr;
    memset(&serveraddr, 0, sizeof(struct sockaddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(port);
    serveraddr.sin_addr.s_addr = inet_addr(ip);
    if(connect(sfd, (struct sockaddr*)&serveraddr, sizeof(struct sockaddr)) == -1)
    //将 sfd 连接至制定的服务器网络地址 serveraddr
    {
        perror("connect");
        close(sfd);
        exit(-1);
    }
    return sfd;
}
```

```
void signalhandler(void)    //用于信号处理，让服务端在按下 Ctrl+c 或 Ctrl+\的时候不会退出
{
    sigset_t sigSet;
    sigemptyset(&sigSet);
    sigaddset(&sigSet,SIGINT);
    sigaddset(&sigSet,SIGQUIT);
    sigprocmask(SIG_BLOCK,&sigSet,NULL);
}
```

服务器端： tcp_net_server.c

```
#include "tcp_net_socket.h"
```

深圳信盈达科技有限公司 专业提供单片机、嵌入式、ARM、LINUX、Android、PCB、FPGA 等技术培训、技术方案。

```
int main(int argc, char * argv[])
{
    if (argc < 3)
    {
        printf("usage:./servertcp ip port\n");
        exit(-1);
    }
    signalhandler();
    int sfd = tcp_init(argv[1], atoi(argv[2])); //或 int sfd = tcp_init("192.168.0.164", 8888);
    while (1) //用 while 循环表示可以与多个客户端接收和发送，但仍是阻塞模式的
    {
        int cfd = tcp_accept(sfd);
        char buf[512] = {0};
        if (recv(cfd, buf, sizeof(buf), 0) == -1) //从 cfd 客户端接收数据存于 buf 中
        {
            perror("recv");
            close(cfd);
            close(sfd);
            exit(-1);
        }
        puts(buf);
        if (send(cfd, "hello world", 12, 0) == -1) //从 buf 中取向 cfd 客户端发送数据
        {
            perror("send");
            close(cfd);
            close(sfd);
            exit(-1);
        }
        close(cfd);
    }
    close(sfd);
}
```

客户端: tcp_net_client.c

```
#include "tcp_net_socket.h"
int main(int argc, char* argv[])
{
    if(argc < 3)
    {
        printf("usage:./clienttcp ip port\n");
        exit(-1);
    }
    int sfd = tcp_connect(argv[1],atoi(argv[2]));
    char buf[512] = {0};
    send(sfd, "hello", 6, 0); //向 sfd 服务端发送数据
    recv(sfd, buf, sizeof(buf), 0); //从 sfd 服务端接收数据
    puts(buf);
    close(sfd);
}

#gcc -o tcp_net_server tcp_net_server.c tcp_net_socket.c
```

```
#gcc -o tcp_net_client tcp_net_client.c tcp_net_socket.c
```

```
#./tcp_net_server 192.168.0.164 8888
```

```
#./tcp_net_client 192.168.0.164 8888
```

备注:

可以通过 `gcc -fpic -c tcp_net_socket.c -o tcp_net_socket.o`

`gcc -shared tcp_net_socket.o -o libtcp_net_socket.so`

`cp lib*.so /lib` //这样以后就可以直接使用该库了

`cp tcp_net_socket.h /usr/include/` //这样头文件包含可以用 `include <tcp_net_socket.h>` 了

以后再用到的时候就可以直接用:

`gcc -o main main.c -ltcp_net_socket` //其中 main.c 要包含头文件: `include <tcp_net_socket.h>`

`./main`

注: 上面的虽然可以实现多个客户端访问, 但是仍然是阻塞模式 (即一个客户访问的时候会阻塞不让另外的客户访问)。解决办法有:

1. 多进程 (因为开销比较大, 所以不常用)

```
int main(int argc, char* argv[])
{
    if(argc < 3)
    {
        printf("usage: ./servertcp ip port\n");
        exit(-1);
    }
    int sfd = tcp_init(argv[1], atoi(argv[2]));
    char buf[512] = {0};
    while(1)
    {
        int cfd = tcp_accept(sfd);
        if(fork() == 0)
        {
            recv(cfd, buf, sizeof(buf), 0);
            puts(buf);
            send(cfd, "hello", 6, 0);
            close(cfd);
        }
        else
        {
            close(cfd);
        }
    }
    close(sfd);
}
```

2. 采用多线程

将服务器上文件的内容全部发给客户端

/* TCP 文件服务器 演示代码 */

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

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

```
#include <errno.h>
```

```
#include <string.h>
```

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

```
#include <sys / fcntl.h>
#include <netinet / in.h>
#include <sys / socket.h>
#include <sys / wait.h>
#include <pthread.h>

#define DEFAULT_SVR_PORT 2828
#define FILE_MAX_LEN 64
char filename[FILE_MAX_LEN + 1];

static void * handle_client(void * arg)
{
    int sock = (int)arg;
    char buff[1024];
    int len;
    printf("begin send\n");
    FILE * file = fopen(filename, "r");
    if (file == NULL)
    {
        close(sock);
        exit;
    }
    //发文件名
    if (send(sock, filename, FILE_MAX_LEN, 0) == -1)
    {
        perror("send file name\n");
        goto EXIT_THREAD;
    }
    printf("begin send file %s....\n", filename);
    //发文件内容
    while (!feof(file))
    {
        len = fread(buff, 1, sizeof(buff), file);
        printf("server read %s,len %d\n", filename, len);
        if (send(sock, buff, len, 0) < 0)
        {
            perror("send file:");
            goto EXIT_THREAD;
        }
    }
EXIT_THREAD:
    if (file)
        fclose(file);
    close(sock);
}

int main(int argc, char * argv[])
{
    int sockfd, new_fd;
```

```
//第 1.定义两个 ipv4 地址
struct sockaddr_in my_addr;
struct sockaddr_in their_addr;
int sin_size, numbytes;
pthread_t cli_thread;
unsigned short port;
if (argc < 2)
{
    printf("need a filename without path\n");
    exit;
}
strncpy(filename, argv[1], FILE_MAX_LEN);
port = DEFAULT_SVR_PORT;
if (argc >= 3)
{
    port = (unsigned short)atoi(argv[2]);
}
//第一步:建立 TCP 套接字 Socket
// AF_INET --> ip 通讯
// SOCK_STREAM --> TCP
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    perror("socket");
    exit(-1);
}
//第二步:设置侦听端口
//初始化结构体, 并绑定 2828 端口
memset(&my_addr, 0, sizeof(struct sockaddr));
//memset(&my_addr,0,sizeof(my_addr));
my_addr.sin_family = AF_INET; /* ipv4 */
my_addr.sin_port = htons(port); /* 设置侦听端口是 2828 , 用 htons 转成网络序*/
my_addr.sin_addr.s_addr = INADDR_ANY; /* INADDR_ANY 来表示任意 IP 地址可能其通讯 */
//bzero(&my_addr.sin_zero,8);
//第三步:绑定套接口,把 socket 队列与端口关联起来.
if (bind(sockfd, (struct sockaddr *) & my_addr, sizeof(struct sockaddr)) == -1)
{
    perror("bind");
    goto EXIT_MAIN;
}
//第四步:开始在 2828 端口侦听,是否有客户端发来联接
if (listen(sockfd, 10) == -1)
{
    perror("listen");
    goto EXIT_MAIN;
}
printf("#@ listen port %d\n", port);
//第五步:循环与客户端通讯
while (1)
{
```



```
    sin_size = sizeof(struct sockaddr_in);
    printf("server waiting...\n");
    //如果有客户端建立连接, 将产生一个全新的套接字 new_fd, 专门用于跟这个客户端通信
    if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1)
    {
        perror("accept:");
        goto EXIT_MAIN;
    }
    printf("---client (ip=%s:port=%d) request \n", inet_ntoa(their_addr.sin_addr), ntohs(their_addr.sin_port));
    //生成一个线程来完成和客户端的会话, 父进程继续监听
    pthread_create(&cli_thread, NULL, handle_client, (void *)new_fd);
}
//第六步:关闭 socket
EXIT_MAIN:
    close(sockfd);
    return 0;
}

/* TCP 文件接收客户端 */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys / types.h>
#include <netinet / in.h>
#include <sys / socket.h>
#include <sys / wait.h>

#define FILE_MAX_LEN 64
#define DEFAULT_SVR_PORT 2828

main(int argc, char * argv[])
{
    int sockfd, numbytes;
    char buf[1024], filename[FILE_MAX_LEN + 1];
    char ip_addr[64];
    struct hostent * he;
    struct sockaddr_in their_addr;
    int i = 0, len, total;
    unsigned short port;
    FILE * file = NULL;
    if (argc < 2)
    {
        printf("need a server ip \n");
        exit;
    }
    strncpy(ip_addr, argv[1], sizeof(ip_addr));
    port = DEFAULT_SVR_PORT;
    if (argc >= 3)
```

```
{
    port = (unsigned short)atoi(argv[2]);
}
//做域名解析(DNS)
//he = gethostbyname(argv[1]);
//第一步:建立一个 TCP 套接字
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
}
//第二步:设置服务器地址和端口 2828
memset(&their_addr, 0, sizeof(their_addr));
their_addr.sin_family = AF_INET;
their_addr.sin_port = htons(port);
their_addr.sin_addr.s_addr = inet_addr(ip_addr);
//their_addr.sin_addr = *((struct in_addr *)he->h_addr);
//bzero(&(their_addr.sin_zero),8);
printf("connect server %s:%d\n", ip_addr, port);
/*第三步:用 connect 和服务端建立连接,注意,这里没有使用本地端口,将由协议栈自动分配一个端口*/
if (connect(sockfd, (struct sockaddr *)& their_addr, sizeof(struct sockaddr)) == -1){
    perror("connect");
    exit(1);
}
if (send(sockfd, "hello", 6, 0) < 0)
{
    perror("send ");
    exit(1);
}
/* 接收文件名,为编程简单,假设前 64 字节固定是文件名,不足用 0 来增充 */
total = 0;
while (total < FILE_MAX_LEN){
    /* 注意这里的接收 buffer 长度,始终是未接收文件剩下的长度,*/
    len = recv(sockfd, filename + total, (FILE_MAX_LEN - total), 0);
    if (len <= 0)
        break;
    total += len;
}
/* 接收文件名出错 */
if (total != FILE_MAX_LEN){
    perror("failure file name");
    exit(-3);
}
printf("recv file %s.....\n", filename);
file = fopen(filename, "wb");
//file = fopen("/home/hxy/abc.txt", "wb");
if (file == NULL)
{
    printf("create file %s failure", filename);
    perror("create:");
}
```

```
    exit(-3);
}
//接收文件数据
printf("recv begin\n");
total = 0;
while (1)
{
    len = recv(sockfd, buf, sizeof(buf), 0);
    if (len == -1)
        break;
    total += len;
    //写入本地文件
    fwrite(buf, 1, len, file);
}
fclose(file);
printf("recv file %s success total lenght %d\n", filename, total);
//第六步:关闭 socket
close(sockfd);
}
```

备注:

1. 读写大容量的文件时, 通过下面的方法效率很高

ssize_t readn(int fd, char * buf, int size)//读大量内容

```
{
    char * pbuf = buf;
    int total, nread;
    for (total = 0; total < size;)
    {
        nread = read(fd, pbuf, size - total);
        if (nread == 0)
            return total;
        if (nread == -1)
        {
            if (errno == EINTR)
                continue;
            else
                return -1;
        }
        total += nread;
        pbuf += nread;
    }
    return total;
}
```

ssize_t writen(int fd, char * buf, int size)//写大量内容

```
{
    char * pbuf = buf;
    int total, nwrite;
    for (total = 0; total < size;)
    {
        nwrite = write(fd, pbuf, size - total);
```

```
    if (nwrite <= 0)
    {
        if (nwrite == -1 && errno == EINTR)
            continue;
        else
            return -1;
    }
    total += nwrite;
    pbuf += nwrite;
}
return total;
}
```

2. 调用 fcntl 将 sockfd 设置为非阻塞模式。

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

.....
sockfd = socket(AF_INET,SOCK_STREAM,0);
iflags = fcntl(sockfd, F_GETFL, 0);
fcntl(sockfd,F_SETFL,O_NONBLOCK | iflags);
.....
```

3. 多路选择 select

Select 在 Socket 编程中还是比较重要的，它可以实现同时对多个文件活套接口描述符的监听，如果监听的事件(比如可读)没有发生则阻塞，如果发生则执行相应的操作。通过 select 既能实现多样化的功能，又不至于使处理器满负荷运行。只有在有了 select 函数后才可以写出像样的网络程序来。

```
#include <sys/select.h>
#include "tcp_net_socket.h"
#define MAXCLIENT 10
main()
{
    int sfd = tcp_init("192.168.0.164", 8888);
    int fd = 0;
    char buf[512] = {0};
    fd_set rdset;
    while(1)
    {
        FD_ZERO(&rdset);
        FD_SET(sfd,&rdset);
        if(select(MAXCLIENT + 1, &rdset, NULL, NULL, NULL) < 0)
            continue;
        for(fd = 0; fd < MAXCLIENT; fd++)
        {
            if(FD_ISSET(fd,&rdset))
            {
                if(fd == sfd)
                {
                    int cfd = tcp_accept(sfd);
                    FD_SET(cfd,&rdset);
                }
            }
        }
    }
}
```

```
else
{
    bzero(buf, sizeof(buf));
    recv(fd, buf, sizeof(buf), 0);
    puts(buf);
    send(fd, "java", 5, 0);
    // FD_CLR(fd, &rdset);
    close(fd);
}
}
}
close(sfd);
}
```

8.3.2 使用 UDP 协议的流程图

UDP 通信流程图如图 8.13 所示：
服务端： socket---bind---recvfrom---sendto---close
客户端： socket-----sendto---recvfrom---close

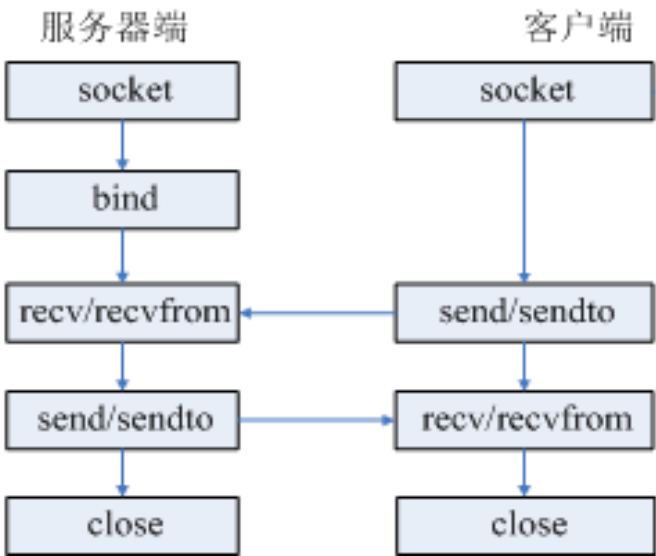


图 8.13

sendto()函数原型：
int sendto(int sockfd, const void *msg,int len,unsigned int flags,const struct sockaddr *to, int tolen);
该函数比 send()函数多了两个参数,to 表示目地机的 IP 地址和端口号信息,而 tolen 常常被赋值为 sizeof (struct sockaddr)。sendto 函数也返回实际发送的数据字节长度或在出现发送错误时返回-1。
recvfrom()函数原型：
int recvfrom(int sockfd,void *buf,int len,unsigned int flags,struct sockaddr *from,int *fromlen);
from 是一个 struct sockaddr 类型的变量,该变量保存连接机的 IP 地址及端口号。fromlen 常置为 sizeof (struct sockaddr)。当 recvfrom()返回时, fromlen 包含实际存入 from 中的数据字节数。Recvfrom()函数返回接收到的字节数或 当出现错误时返回-1, 并置相应的 errno。

示例：UDP 的基本操作

```
服务器端：
#include <sys / types.h>
#include <sys / socket.h>
```

```
#include <netinet / in.h>
#include <arpa / inet.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
main()
{
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sfd == -1)
    {
        perror("socket");
        exit(-1);
    }

    struct sockaddr_in saddr;
    bzero(&saddr, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(8888);
    saddr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sfd, (struct sockaddr *) & saddr, sizeof(struct sockaddr)) == -1)
    {
        perror("bind");
        close(sfd);
        exit(-1);
    }

    char buf[512] = {0};
    while (1)
    {
        struct sockaddr_in fromaddr;
        bzero(&fromaddr, sizeof(fromaddr));
        int fromaddrlen = sizeof(struct sockaddr);
        if (recvfrom(sfd, buf, sizeof(buf), 0, (struct sockaddr *) & fromaddr, &fromaddrlen) == -1)
        {
            perror("recvfrom");
            close(sfd);
            exit(-1);
        }

        printf("receive from %s %d,the message is:%s\n", inet_ntoa(fromaddr.sin_addr),
            ntohs(fromaddr.sin_port), buf);

        sendto(sfd, "world", 6, 0, (struct sockaddr *) & fromaddr, sizeof(struct sockaddr));
    }

    close(sfd);
}
```

客户端:

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

深圳信盈达科技有限公司 专业提供单片机、嵌入式、ARM、LINUX、Android、PCB、FPGA 等技术培训、技术方案。

```
#include <sys / socket.h>
#include <netinet / in.h>
#include <arpa / inet.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sfd == -1)
    {
        perror("socket");
        exit(-1);
    }

    struct sockaddr_in toaddr;
    bzero(&toaddr, sizeof(toaddr));
    toaddr.sin_family = AF_INET;
    toaddr.sin_port = htons(atoi(argv[2])); //此处的端口号要跟服务器一样
    toaddr.sin_addr.s_addr = inet_addr(argv[1]); //此处为服务器的 ip
    sendto(sfd, "hello", 6, 0, (struct sockaddr *) & toaddr, sizeof(struct sockaddr));

    char buf[512] = {0};
    struct sockaddr_in fromaddr;
    bzero(&fromaddr, sizeof(fromaddr));
    int fromaddrlen = sizeof(struct sockaddr);
    if (recvfrom(sfd, buf, sizeof(buf), 0, (struct sockaddr *) & fromaddr, &fromaddrlen) == -1)
    {
        perror("recvfrom");
        close(sfd);
        exit(-1);
    }
    printf("receive from %s %d,the message is:%s\n", inet_ntoa(fromaddr.sin_addr), ntohs(fromaddr.sin_port),
    buf);

    close(sfd);
}
```

示例：UDP 发送文件 先发文件大小 再发文件内容

//服务器端

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
```



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

```
main()
```

```
{
```

```
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
```

```
    if(sfd == -1)
```

```
    {
```

```
        perror("socket");
```

```
        exit(-1);
```

```
    }
```

```
    struct sockaddr_in saddr;
```

```
    bzero(&saddr, sizeof(saddr));
```

```
    saddr.sin_family = AF_INET;
```

```
    saddr.sin_port = htons(8888);
```

```
    saddr.sin_addr.s_addr = INADDR_ANY;
```

```
    if(bind(sfd, (struct sockaddr*)&saddr, sizeof(struct sockaddr)) == -1)
```

```
    {
```

```
        perror("bind");
```

```
        close(sfd);
```

```
        exit(-1);
```

```
    }
```

```
    char buf[512] = {0};
```

```
    struct sockaddr_in fromaddr;
```

```
    bzero(&fromaddr, sizeof(fromaddr));
```

```
    int fromaddrlen = sizeof(struct sockaddr);
```

```
    if(recvfrom(sfd, buf, sizeof(buf), 0, (struct sockaddr*)&fromaddr, &fromaddrlen) == -1)
```

```
    {
```

```
        perror("recvfrom");
```

```
        close(sfd);
```

```
        exit(-1);
```

```
    }
```

```
    printf("receive from %s %d,the message is:%s\n", inet_ntoa(fromaddr.sin_addr), ntohs(fromaddr.sin_port),
buf);
```

```
    FILE* fp = fopen("1.txt", "rb");
```

```
    struct stat st; //用于获取文件内容的大小
```

```
    stat("1.txt", &st);
```

```
    int filelen = st.st_size;
```

```
    sendto(sfd, (void*)&filelen, sizeof(int), 0, (struct sockaddr*)&fromaddr, sizeof(struct sockaddr));
```

```
    while(!feof(fp)) //表示没有到文件尾
```

```
    {
```

```
        int len = fread(buf, 1, sizeof(buf), fp);
```

```
        sendto(sfd, buf, len, 0, (struct sockaddr*)&fromaddr, sizeof(struct sockaddr));
```

```
    }
```

```
    close(sfd);
```

```
}
```

//客户端

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#define BUFSIZE 512
int main(int argc, char* argv[])
{
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    if(sfd == -1)
    {
        perror("socket");
        exit(-1);
    }

    struct sockaddr_in toaddr;
    bzero(&toaddr, sizeof(toaddr));
    toaddr.sin_family = AF_INET;
    toaddr.sin_port = htons(atoi(argv[2]));
    toaddr.sin_addr.s_addr = inet_addr(argv[1]);
    sendto(sfd, "hello", 6, 0, (struct sockaddr*)&toaddr, sizeof(struct sockaddr));

    char buf[BUFSIZE] = {0};
    struct sockaddr_in fromaddr;
    bzero(&fromaddr, sizeof(fromaddr));
    int fromaddrlen = sizeof(struct sockaddr);
    int filelen = 0;    //用于保存文件长度
    FILE* fp = fopen("2.txt", "w+b");
    //接收文件的长度
    recvfrom(sfd, (void*)&filelen, sizeof(int), 0, (struct sockaddr*)&fromaddr, &fromaddrlen);
    printf("the length of file is %d\n", filelen);
    printf("Create a new file!\n");
    printf("begin to receive file content!\n");
    //接收文件的内容
    while(1)
    {
        int len = recvfrom(sfd, buf, sizeof(buf), 0, (struct sockaddr*)&fromaddr, &fromaddrlen);
        if(len < BUFSIZE)
        //如果接收的长度小于 BUFSIZE, 则表示最后一次接收, 此时要用 break 退出循环
        {
            fwrite(buf, sizeof(char), len, fp);
            break;
        }
        fwrite(buf, sizeof(char), len, fp);
    }
}
```

```
    printf("receive file finished!\n");
    close(sfd);
}
```

8.3.3 设置套接口的选项 setsockopt 的用法

函数原型：

```
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

sockfd: 标识一个套接口的描述字
level: 选项定义的层次；支持 SOL_SOCKET、IPPROTO_TCP、IPPROTO_IP 和 IPPROTO_IPV6
optname: 需设置的选项
optval: 指针，指向存放选项值的缓冲区
optlen: optval 缓冲区长度

下面的操作如果要做的的话，全部都必须要放在 bind 之前，另外通常是用于 UDP 的。

1. 如果在已经处于 ESTABLISHED 状态下的 socket(一般由端口号和标志符区分)调用 closesocket (一般不会立即关闭而经历 TIME_WAIT 的过程)后想继续重用该 socket:

```
int reuse=1;
setsockopt(s,SOL_SOCKET,SO_REUSEADDR,(const char*)& reuse,sizeof(int));
```

2. 如果要已经处于连接状态的 socket 在调用 closesocket 后强制关闭，不经历 TIME_WAIT 的过程:

```
int reuse=0;
setsockopt(s,SOL_SOCKET,SO_REUSEADDR,(const char*)& reuse,sizeof(int));
```

3. 在 send(),recv()过程中有时由于网络状况等原因，发收不能预期进行,而设置收发时限:

```
int nNetTimeout=1000; // 1 秒
// 发送时限
setsockopt(socket, SOL_SOCKET,SO_SNDTIMEO, (char *)&nNetTimeout,sizeof(int));
// 接收时限
setsockopt(socket, SOL_SOCKET,SO_RCVTIMEO, (char *)&nNetTimeout,sizeof(int));
```

4. 在 send()的时候，返回的是实际发送出去的字节(同步)或发送到 socket 缓冲区的字节(异步),系统默认的状态发送和接收一次为 8688 字节(约为 8.5K); 在实际的过程中发送数据和接收数据量比较大，可以设置 socket 缓冲区，而避免了 send(),recv()不断的循环收发:

```
// 接收缓冲区
int nRecvBuf=32*1024;    // 设置为 32K
setsockopt(s,SOL_SOCKET,SO_RCVBUF,(const char*)&nRecvBuf,sizeof(int));
// 发送缓冲区
int nSendBuf=32*1024;    // 设置为 32K
setsockopt(s,SOL_SOCKET,SO_SNDBUF,(const char*)&nSendBuf,sizeof(int));
```

5. 如果在发送数据时，希望不经历由系统缓冲区到 socket 缓冲区的拷贝而影响程序的性能:

```
int nZero=0;
setsockopt(socket, SOL_SOCKET,SO_SNDBUF, (char *)&nZero,sizeof(int));
```

6. 同在上在 recv()完成上述功能(默认情况是将 socket 缓冲区的内容拷贝到系统缓冲区):

```
int nZero=0;
setsockopt(socket, SOL_SOCKET,SO_RCVBUF, (char *)&nZero,sizeof(int));
```

7. 一般在发送 UDP 数据报的时候，希望该 socket 发送的数据具有广播特性:

```
int bBroadcast = 1;
setsockopt(s,SOL_SOCKET,SO_BROADCAST,(const char*)&bBroadcast,sizeof(int));
```

实例请参考《broadcast》

第9章 系统编程的一个小程序

本章介绍了一个用 C 语言编程实现 linux 下的聊天室功能的程序。该程序可以实现群聊的功能，即当某个客户发出消息后，服务器和其他客户端都能收到此消息。还能够显示客户端的用户名及消息时间，保存聊天记录。当客户端退出聊天室后，服务器和其他在线客户端会有提示。

实现群聊的机制是：当某个客户端需要发送消息是，它将此消息发送给服务器，服务器再将此消息转发给各客户端，各客户端之间是无连接的，即相互之间不能直接通信。因此，在服务器中，有两个子线程，一个是用来监听是否有客户端登录服务器，若有，建立与其连接的套接字，并存入在线客户序列里；另一个是接收转发线程，其阻塞等待是否有客户端发消息过来，若有，取出，并转发给所有在线用户。在这里用到了 select 函数实现多路选择，即对各个客户端的 socket 进行读监听。在客户端也有两个子线程，一个用来向服务器发送消息，另一个线程用来接收服务器发来的消息。

具体细节如下所述：

用户程序命名为 client.c;服务器程序命名为 server.c

client 可以通过 socket 连接 server

在 client 端开始运行后,提示输入服务器 ip

若连接 server 的 socket 建立成功，返回提示信息

client 输入的聊天内容在所有 client 端（多个）和 server 端都会实时显示

多个 client 可同时接入 server，以进入聊天室，最多支持 100 个 client

client 端输入 quit、Quit 或 QUIT 退出连接，server 端提示 client 退出

输入收到消息时会显示时间，保存聊天记录

其他细节见输出结果

服务器程序源代码如下：

```
/****** 服务器程序 (server.c) *****/  
  
#include <stdlib.h>  
#include <stdio.h>  
#include <errno.h>  
#include <string.h>  
#include <netdb.h>  
#include <sys / types.h>  
#include <netinet / in.h>  
#include <sys / socket.h>  
#include <time.h>  
#include <unistd.h>  
#include <sys / select.h>  
#include <sys / types.h>  
#include <sys / stat.h>  
#include <fcntl.h>  
  
#define LISTENQ 100 //最大监听队列  
#define PORT 6000 //监听端口  
#define MAXFD 100 //最大的在线用户数量  
#define NAMESIZE 20 //客户端姓名占用的字节数  
#define TIMESIZE 30 //收到消息时的时间占用的字节数  
#define SENDBUFSIZE 1024 //消息的数组长度  
#define FILENAMESIZE 40 //聊天记录文件名的长度  
#define NEWCLIENTINFORMSIZE 70 //通知新用户上线的消息数组长度
```

```
int sockfd; //绑定服务器端的 ip 地址和端口的套接字
```

FILE * fp; //文件指针，用于指向保存聊天记录的文件

int maxi = 0; //maxi 表示当前 client 数组中最大的用户的 i 值

static int client[MAXFD]; //全局数组，保存所有客户端对应的套接口描述符

```
void tcp_init(int argc, char * argv[])
{
    struct sockaddr_in server_addr;
    if (argc != 1)
    {
        fprintf(stderr, "Usage:%s portnumber\n", argv[0]);
        exit(1);
    }

    /* 服务器端开始建立 socket 描述符 */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        fprintf(stderr, "Socket error:%s\n", strerror(errno));
        exit(1);
    }
    /* 服务器端填充 sockaddr 结构 */
    bzero(&server_addr, sizeof(struct sockaddr_in));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(PORT);
    int reuse = 1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const char *) & reuse, sizeof(int));
    /* 捆绑 sockfd 描述符 */
    if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) == -1)
    {
        fprintf(stderr, "Bind error:%s\n", strerror(errno));
        exit(1);
    }
    printf("服务器监听端口%d...\n", PORT);
    /* 监听 sockfd 描述符 */
    if (listen(sockfd, LISTENQ) == -1)
    {
        fprintf(stderr, "Listen error:%s\n", strerror(errno));
        exit(1);
    }
}

/*接收对应的客户端的消息，并转发给所有在线客户端*/
void recv_and_send_to_all_client(int index)
{
    int nbytes = 0;
    char sendbuf[SENDBUFSIZE] = {0};
    int outindex = 0;
    time_t tt;
    struct tm * pTm;
```

```
nbytes = 0;
nbytes = read(client[index], sendbuf, sizeof(sendbuf));
sendbuf[0] = 'M'; //M'表示正常聊天信息
if (nbytes > 0)
{
    tt = time(NULL);
    pTm = localtime(&tt);
    sprintf(sendbuf + NAMESIZE, " %d-%d-%d %0d:%0d:%0d\n", (1900 + pTm->tm_year), (1 +
pTm->tm_mon), (pTm->tm_mday), (pTm->tm_hour), (pTm->tm_min), (pTm->tm_sec));
    outindex = 0;
    /*以下循环将消息转发给所有客户端*/
    while (outindex < maxi)
    {
        if (write(client[outindex], sendbuf, sizeof(sendbuf)) == -1)
        {
            fprintf(stderr, "Write Error:%s\n", strerror(errno));
            close(sockfd);
            fclose(fp);
            exit(1);
        }
        outindex++;
    }

    /*如果某个客户端下线，则删除全局数组 client 中对应的套接字*/
    if (strcmp(sendbuf + NAMESIZE + TIMESIZE, "QUIT") == 0 || strcmp(sendbuf + NAMESIZE +
TIMESIZE, "Quit") == 0
        || strcmp(sendbuf + NAMESIZE + TIMESIZE, "quit") == 0)
    {
        for (; index < maxi - 1; index++)
            client[index] = client[index + 1];
        maxi--;
    }
    /*服务器打印出消息内容并保存消息到文件*/
    printf("%s", sendbuf + 1);
    printf("%s", sendbuf + NAMESIZE);
    printf(" %s\n", sendbuf + NAMESIZE + TIMESIZE);
    fprintf(fp, "%s", sendbuf + 1);
    fprintf(fp, "%s", sendbuf + NAMESIZE);
    fprintf(fp, "%s\n", sendbuf + NAMESIZE + TIMESIZE);
}
}
```

void * check_client_send(void * arg) //监听转发线程入口函数

```
{
    static int index = 0; //very important
    int select_n = 0;
    int in = 0;
    fd_set allset, rset;
    FD_ZERO(&allset);
```

```
while (1)
{
    if (maxi > 0)
    {
        for (index = 0; index < maxi; index++)
            FD_SET(client[index], &allset);

        rset = allset;
        select_n = select(FD_SETSIZE, &rset, NULL, NULL, NULL);/*阻塞监听所有的客户端对应的套
接字有没有可读的，即是否有人发消息过来*/

        switch (select_n)
        {
            case 0:continue;
            case -1:
                perror("select error!");
                exit(-1);
            default:
                for (index = 0, in = 0; index < maxi && in < select_n; index++)
                    if (FD_ISSET(client[index], &rset))
                    {
                        in++;
                        recv_and_send_to_all_client(index);/*调用函数接收客户端的消息
                    }
                }
        }
    }
}
```

/*该线程等待客户端连接，并将新的套接字存入全局数组 static int client[MAXFD]中*/

```
void * wait_client_connect(void * pthidcheck)
{
    struct sockaddr_in client_addr;
    int sin_size, portnumber;
    int new_fd = 0;
    char clientname[NAMESIZE] = {0};
    char new_client_inform[NEWCLIENTINFORMSIZE] = {0};
    static int index = 0;
    time_t tt;
    struct tm * pTm = NULL;
    while (1)
    {
        /* 服务器阻塞,直到客户程序建立连接 */
        if (maxi >= MAXFD)
        {
            printf("已经达到人数上线\n");
            continue;
        }
    }
}
```



```
    }
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size)) == -1)
    {
        fprintf(stderr, "Accept error:%s\n", strerror(errno));
        exit(1);
    }
    client[maxi] = new_fd;
    if (-1 == read(client[maxi], clientname, sizeof(clientname)))
    {
        fprintf(stderr, "Read client name error:%s\n", strerror(errno));
        close(sockfd);
        exit(1);
    }
    maxi++;
    tt = time(NULL);
    pTm = localtime(&tt);
    sprintf(new_client_inform, "N%d-%d-%d %0d:%0d:%0d", (1900 + pTm->tm_year), (1 +
    pTm->tm_mon), (pTm->tm_mday), (pTm->tm_hour), (pTm->tm_min), (pTm->tm_sec)); /*'N'表示向所有在线
    用户通知新用户上线了
    sprintf(new_client_inform + TIMESIZE, " 新用户%s 进入聊天室", clientname);
    int outindex = 0;
    while (outindex < maxi) //向聊天室中所有的客户通知新用户上线
    {
        if (write(client[outindex], new_client_inform, sizeof(new_client_inform)) == -1)
        {
            fprintf(stderr, "Write Error:%s\n", strerror(errno));
            close(sockfd);
            exit(1);
        }
        outindex++;
    }

    /*创建线程收客户端的消息并转发给所有其它在线客户端，同时保存聊天记录*/
    if (maxi == 1)
    {
        if (-1 == pthread_create((pthread_t *)&pthidcheck, NULL, check_client_send, NULL))
        {
            fprintf(stderr, "Creat pthread Error:%s\n", strerror(errno));
            close(sockfd);
            exit(1);
        }
        fp = fopen("server_msgrecord.txt", "ab");
        if (fp == NULL)
        {
            fprintf(stderr, "Fopen Error:%s\n", strerror(errno));
            close(sockfd);
            exit(1);
        }
    }
```

```
    }
    else if (maxi > 1)
    {
        pthread_cancel(*(pthread_t *)pthidcheck);
        if (-1 == pthread_create((pthread_t *)pthidcheck, NULL, check_client_send, NULL))
        {
            fprintf(stderr, "Creat pthread Error:%s\n", strerror(errno));
            fclose(fp);
            close(sockfd);
            exit(1);
        }
    }

    printf("%s\n", new_client_inform + 1);
    printf("%s\n", new_client_inform + TIMESIZE);
    fprintf(fp, "%s\n", new_client_inform + 1);
    fprintf(fp, "%s\n", new_client_inform + TIMESIZE);
}
}

int main(int argc, char * argv[])
{
    pthread_t thidaccept, thidcheck = -1;
    tcp_init(argc, argv);
    printf("欢迎来到本聊天室\n");
    /*创建线程等待接受客户端连接请求*/
    if (-1 == pthread_create(&thidaccept, NULL, wait_client_connect, (void *) & thidcheck))
    {
        fprintf(stderr, "Creat pthread Error:%s\n", strerror(errno));
        close(sockfd);
        exit(1);
    }

    pthread_join(thidaccept, NULL);
    if (thidcheck != -1)
        pthread_join(thidcheck, NULL);
    close(sockfd);
    fclose(fp);
    exit(0);
}
```

客户端程序源代码如下:

/****** 客户端程序 (client.c) *****/

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
```

```
#include <sys / types.h>
#include <netinet / in.h>
#include <sys / socket.h>
#include <sys / types.h>
#include <sys / stat.h>
#include <fcntl.h>
#include <time.h>

#define PORT 6000 //监听端口
#define NAMESIZE 20 //客户端姓名占用的字节数
#define TIMESIZE 30 //收到消息时的时间占用的字节数
#define SENDBUFSIZE 1024 //消息的数组长度
#define FILENAMESIZE 40 //聊天记录文件名的长度
#define NEWCLIENTINFORMSIZE 70 //通知新用户上线的消息数组长度
```

```
FILE * fp; //文件指针，用于指向保存聊天记录的文件
static int sockfd; //绑定服务器端的 ip 地址和端口的套接字
char clientname[NAMESIZE] = {0}; //保存自己的昵称
```

```
/*连接服务器*/
```

```
void connect_server()
{
    struct sockaddr_in server_addr;
    struct hostent * host;
    char strhost[16];
    printf("请输入服务器 ip 地址\n");
    scanf("%s", strhost);
    //strcpy(strhost, "192.168.0.101");
    if ((host = gethostbyname(strhost)) == NULL)
    {
        fprintf(stderr, "Gethostname error\n");
        exit(1);
    }
    /* 客户程序开始建立 sockfd 描述符 */
    printf("正在建立套接口...\n");
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        fprintf(stderr, "Socket Error:%s\n", strerror(errno));
        exit(1);
    }
    /* 客户程序填充服务端的资料 */
    bzero(&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr = *((struct in_addr *)host->h_addr);
    printf("套接口创建成功，正在连接服务器...\n");
    /* 客户程序发起连接请求 */
    if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) == -1)
    {
```

```
        fprintf(stderr, "Connect Error:%s\n", strerror(errno));
        close(sockfd);
        exit(1);
    }
    /* 连接成功了 */
    printf("连接服务器成功\n 欢迎来到聊天室\n");
    printf("请输入你的用户昵称\n>");
    fflush(stdout);
    scanf("%s", clientname);
    write(sockfd, clientname, sizeof(clientname));
    char recvbuf[SENDBUFSIZE] = {0};
    if (-1 == read(sockfd, recvbuf, sizeof(recvbuf)))
    {
        fprintf(stderr, "Read my inform Error:%s\n", strerror(errno));
        close(sockfd);
        exit(1);
    }
    if (recvbuf[0] == 'N')
    {
        char filename[FILENAME_SIZE] = {0};
        strcpy(filename, clientname);
        strcat(filename, "_msgrecord.txt");
        fp = fopen(filename, "ab");
        if (fp == NULL)
        {
            fprintf(stderr, "Fopen Error:%s\n", strerror(errno));
            close(sockfd);
            exit(1);
        }
        printf("%s\n", recvbuf + 1);
        printf("%s\n", recvbuf + TIMESIZE);
        fprintf(fp, "%s\n", recvbuf + 1);
        fprintf(fp, "%s\n", recvbuf + TIMESIZE);
        printf("开始聊天吧 (\"QUIT\"断开连接) \n");
    }
}

/*接收服务器消息线程入口函数*/
void * recvfromserver(void * arg)
{
    time_t tt;
    struct tm * pTm;
    char recvbuf[SENDBUFSIZE];
    int nbytes = 0;

    while (1)
    {
        memset(recvbuf, 0, sizeof(recvbuf));
        if (0 < read(sockfd, recvbuf, sizeof(recvbuf)))
```

```
{
    if (recvbuf[0] == 'M') //如果消息是 'M' 表示消息收到服务器的正常聊天消息
    {
        fprintf(fp, "\n%s", recvbuf + 1);
        fprintf(fp, " %s", recvbuf + NAMESIZE);
        fprintf(fp, " %s", recvbuf + NAMESIZE + TIMESIZE);
        fflush(fp);
        if (strcmp(recvbuf + 1, clientname) == 0)
            continue;
        printf("\n%s", recvbuf + 1);
        printf(" %s", recvbuf + NAMESIZE);
        printf(" %s", recvbuf + NAMESIZE + TIMESIZE);
    }
    else if (recvbuf[0] == 'N') //如果是 'N' 表示收到服务器的新用户通知上线消息
    {
        printf("%s\n", recvbuf + 1);
        printf("%s\n", recvbuf + TIMESIZE);
        fprintf(fp, "%s\n", recvbuf + 1);
        fprintf(fp, "%s\n", recvbuf + TIMESIZE);
    }
    else
        printf("收到不能识别的信息\n");
    /*显示时间*/
    tt = time(NULL);
    pTm = localtime(&tt);
    printf("\n%s  %d-%d-%d %0d:%0d:%0d\n>", clientname, (1900 + pTm->tm_year), (1 +
pTm->tm_mon), (pTm->tm_mday), (pTm->tm_hour), (pTm->tm_min), (pTm->tm_sec));
    fflush(stdout);
}
}
```

/*向服务器发送消息线程入口函数*/

void * sendtoserver(void * arg)

```
{
    time_t tt;
    struct tm * pTm;
    char sendbuf[SENDBUFSIZE];
    fgets(sendbuf + NAMESIZE + TIMESIZE, sizeof(sendbuf) - NAMESIZE + TIMESIZE, stdin); //清除输入缓冲区中的换行符
    while (1)
    {
        memset(sendbuf, 0, sizeof(sendbuf));
        tt = time(NULL);
        pTm = localtime(&tt);
        printf("%s  %d-%d-%d %0d:%0d:%0d\n>", clientname, (1900 + pTm->tm_year), (1 + pTm->tm_mon),
(pTm->tm_mday), (pTm->tm_hour), (pTm->tm_min), (pTm->tm_sec));
        fflush(stdin);
        fflush(stdout);
    }
}
```

```
fgets(sendbuf + NAMESIZE + TIMESIZE, sizeof(sendbuf)-NAMESIZE + TIMESIZE, stdin);
sendbuf[49 + strlen(sendbuf + NAMESIZE + TIMESIZE)] = '\0';
strcpy(sendbuf + 1, clientname); //在消息前面加上自己的昵称
if ((write(sockfd, sendbuf, sizeof(sendbuf))) == -1)
{
    fprintf(stderr, "Write Error:%s\n", strerror(errno));
    close(sockfd);
    fclose(fp);
    exit(1);
}
/*如果输入 “quit” 或 “Quit”或 “QUIT” 都会退出*/
if (strcmp(sendbuf + NAMESIZE + TIMESIZE, "QUIT") == 0 || strcmp(sendbuf + NAMESIZE +
TIMESIZE, "Quit") == 0 || strcmp(sendbuf + NAMESIZE + TIMESIZE, "quit") == 0)
{
    close(sockfd);
    fclose(fp);
    printf("您已退出聊天室\n");
    exit(1);
}
}
```

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

    pthread_t thidrcv, thidsnd;
    connect_server(); //连接服务器
    /*创建发送和接收消息的线程*/
    if (-1 == pthread_create(&thidrcv, NULL, rcvfromserver, NULL))
    {
        fprintf(stderr, "Creat pthread Error:%s\n", strerror(errno));
        close(sockfd);
        exit(1);
    }

    if (-1 == pthread_create(&thidsnd, NULL, sendtoerver, NULL))
    {
        fprintf(stderr, "Creat pthread Error:%s\n", strerror(errno));
        close(sockfd);
        fclose(fp);
        exit(1);
    }
    pthread_join(thidrcv, NULL);
    pthread_join(thidsnd, NULL);
    close(sockfd);
    fclose(fp);
    exit(0);
}
```

以下是其 Makefile

```
all:server client
server:server.c
    cc -o server server.c -lpthread
client:client.c
    cc -o client client.c -lpthread
rebuild:clean all
clean:
    rm -rf client server
```

将以上源文件分别保存好后，终端下输入 `make` 编译就可以生成可执行文件 `server` 和 `client` 了，然后就可以运行它们测试效果了。

附录 A: C 语言文件操作详解

在 C 语言文件操作的过程中, 通常有以下几种方式:

1. 单个字符的操作:

fputc

函数原型: `int fputc(int c, FILE *fp);`

功能: 把一字节代码 `c` 写入 `fp` 指向的文件中

返回值: 正常, 返回 `c`; 出错, 为 `EOF(-1)`

fgetc

函数原型: `int fgetc(FILE *fp);`

功能: 从 `fp` 指向的文件中读取一字节代码

返回值: 正常, 返回读到的代码值; 读到文件尾或出错, 为 `EOF(-1)`

feof

函数原型: `int feof(FILE *fp);`

功能: 判断文件是否到末尾

返回值: 文件未结束, 返回 0; 文件结束, 返回真 (非 0)

示例:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    char ch;
    FILE *fp1 = fopen("d:\\a.dat", "wb"); /* 以写的方式打开二进制文件 */
    FILE *fp2;
    if(NULL == fp1)
    {
        printf("Can not open this file!\n");
        exit(0);
    }
    printf("please input a string:");
    ch = getchar();
    while(ch != '#') /* 输入#号结束 */
    {
        fputc(ch, fp1); /* 依次向文件中写 */
        ch = getchar();
    }
    fflush(stdin); /* 清空输入缓冲区 */
    fclose(fp1); /* 关闭文件 */

    fp2 = fopen("d:\\a.dat", "rb"); /* 以读的方式打开二进制文件 */
    if(NULL == fp2)
    {
        printf("Can not open this file!\n");
        exit(0);
    }
    while(!feof(fp2)) /* 判断是否到达文件末尾 */
    {
        ch = fgetc(fp2); /* 从文件中依次读取 */
        putchar(ch);
    }
}
```

```
    }  
    putchar('\n');  
    fflush(stdout);    /* 清空输出缓冲区 */  
    fclose(fp2);      /* 关闭文件 */  
}
```

注意:

putc(ch,fp)与 fputc(ch,fp)效果一致, 但是 putc 可以作为宏替换, 速度更快

getc(fp)与 fgetc(fp)效果一致, 但是 getc 可以作为宏替换, 速度更快

putchar(c) 则是 fputc(c,stdout)

getchar() 则是 fgetc(stdin)

2. 对字符串的操作:

函数原型:

```
char *fgets(char *s, int n, FILE *fp)
```

```
int fputs(char *s, FILE *fp)
```

功能:

fgets: 从 fp 指向的文件中读一个长度为 n 的字符串, 保存到 s 中。其中 char* s 要是已经有内存空间的, 通常是用数组定义的, 或者用 char*则要用 malloc 分配内存。

fputs: 向 fp 所指向的文件中写入字符串 s。

返回值:

fgets: 正常时返回读取字符串的首地址; 出错或文件尾, 返回 NULL

fputs: 正常时返回写入的最后一个字符; 出错为 EOF (-1)

示例:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
void main()  
{  
    FILE *fp;  
    char string[100];  
    if((fp = fopen("file.txt", "w")) == NULL)  
    {  
        printf("can't open this file!\n");  
        exit(0);  
    }  
    while(strlen(gets(string)) > 0) /* 从键盘输入字符串, 回车停止 */  
    {  
        fputs(string, fp);    /* 将字符串写入文件 */  
        fputs("\n", fp);      /* 在最后加上一个'\n' */  
    }  
    fclose(fp);  
  
    if((fp = fopen("file.txt", "r")) == NULL)  
    {  
        printf("can't open this file!\n");  
        exit(0);  
    }  
    while(fgets(string, 10, fp) != NULL) /* 判断是否到达文件末尾 */
```

```
    fputs(string, stdout); /* 等价于 puts(string); */
    fclose(fp);
}
```

3. 对二进制形式的块读写:

函数原型:

```
size_t fread(void *buffer, size_t size, size_t count, FILE *fp);
size_t fwrite(void *buffer, size_t size, size_t count, FILE *fp);
```

功能: 读/写数据块 fread 与 fwrite 一般用于二进制文件的输入/输出

fread: 从 fp 所指向的文件中读取 count 个块, 每个块的长度为 size 个字节, 存放到以 buffer 为首地址的内存中。其中 buffer 必须已经是由内存空间的。

fwrite: 从以 buffer 为首地址的内存中读取 count 个块, 每个块的长度为 size 个字节, 写入到 fp 所指向的文件中。

返回值: 成功, 返回读/写的块数; 出错或文件尾, 返回 0

说明:

buffer: 指向要输入/输出数据块的首地址的指针
size: 读/写的数据块的大小(字节数)
count: 要读/写的数据块的个数
fp: 要读/写的文件指针

示例:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct STUDENT
{
    char sNo[5];
    char sName[20];
    double score;
}STUDENT;
void main()
{
    /* 只对单个的时候 */
    // int x = 19;
    // int y = 0;
    // FILE* fp = fopen("d:\\a.dat", "wb");
    // if(NULL == fp)
    // {
    //     printf("Can not open this file!\n");
    //     exit(0);
    // }
    // fwrite(&x, sizeof(x), 1, fp);
    // fclose(fp);
    //
    // fp = fopen("d:\\a.dat", "rb");
    // if(NULL == fp)
    // {
    //     printf("Can not open this file!\n");
    //     exit(0);
    // }
    // fread(&y, sizeof(y), 1, fp);
```

```
// printf("%d\n", y);
// fclose(fp);

/* 对于块的时候 */
// int x[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
// int y[9] = {0};
// int i;
// FILE* fp = fopen("d:\\a.txt", "wb");
// if(NULL == fp)
// {
//     printf("Can not open this file!\n");
//     exit(0);
// }
// fwrite(x, sizeof(int), sizeof(x)/sizeof(int), fp);
// fclose(fp);
//
// fp = fopen("d:\\a.txt", "rb");
// if(NULL == fp)
// {
//     printf("Can not open this file!\n");
//     exit(0);
// }
// fread(y, sizeof(int), sizeof(y)/sizeof(int), fp);
// for(i = 0; i < 9; i++)
// {
//     printf("%d ", y[i]);
// }
// printf("\n");
// fclose(fp);

/* 操作一块 */
STUDENT stu[3] = {
    "0001", "赵军", 89,
    "0002", "李千", 90,
    "0003", "张芳", 100
};
STUDENT stu2;
FILE* fp = fopen("d:\\b.txt", "wb+");
if(NULL == fp)
{
    printf("Can not open this file!\n");
    exit(0);
}
fwrite(stu, sizeof(STUDENT), sizeof(stu)/sizeof(STUDENT), fp);

rewind(fp); /* 文件指针重新回到头 */

fseek(fp, sizeof(STUDENT), SEEK_SET); /* 定位, 此处跳过一条记录 */
fread(&stu2, sizeof(STUDENT), 1, fp); /* 从文件中读取指定大小的块 */
```

```
printf("%s-->%s-->%lf\n", stu2.sNo, stu2.sName, stu2.score);
fclose(fp);
}
```

4. 格式化操作文本文件:

函数原型:

```
int fprintf(FILE *fp,const char *format[,argument,...])
```

```
int fscanf(FILE *fp,const char *format[,address,...])
```

功能: 按格式对文件进行读写操作。二进制的文件不要用, 而改用 fread 和 fwrite。

fprintf: 输出列表中的各项数据按指定的格式写入到 fp 指向的文件中

scanf: 从 fp 所指向的文件中按指定的格式读取数据依次存放到输出列表中的各项。

返回值: 成功,返回读写的个数;出错或文件尾,返回 EOF (-1), eg:

```
fprintf(fp, "%d,%6.2f", i,t); //将 i 和 t 按%d,%6.2f 格式输出到 fp 文件
```

```
fscanf(fp, "%d,%f", &i,&t); //若文件中有 3,4.5 ,则将 3 送入 i, 4.5 送入 t
```

示例:

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

void main()
{
    int a = 5, b;
    double x = 3.5, y;
    char szText[20] = "HelloWorld!", szNewText[20];
    FILE* fp = fopen("d:\\c.txt", "w+");
    if(fp == NULL)
    {
        printf("Can not open this file!\n");
        exit(0);
    }
    fprintf(fp, "%d %lf %s\n", a, x, szText);

    rewind(fp);

    fscanf(fp, "%d %lf %s\n", &b, &y, szNewText);
    printf("%d---%lf---%s\n", b, y, szNewText);

    fclose(fp);
}
```

5. rewind 函数

函数原型: void rewind(FILE *fp)

功能: 重置文件位置指针到文件开头

返回值: 无

6. fseek 函数

函数原型: int fseek(FILE *fp,long offset,int whence)

功能: 文件随机定位函数, 改变文件位置指针的位置。一般用于二进制文件。

返回值: 成功, 返回 0; 失败, 返回非 0 值

示例: fseek(fp,100L,0);

```
fseek(fp,50L,1);
```

```
fseek(fp,-10L,2);
```

第 2 个参数 offset 表示相对 whence 为基点的偏移字节数，要求是长整型数据，可以是正整数（向文件尾方向移动）、0（不移动）、负整数（向文件头方向移动）。

第 3 个参数可以用如下的宏替换：

文件开始	SEEK_SET	0
------	----------	---

文件当前位置	SEEK_CUR	1
--------	----------	---

文件末尾	SEEK_END	2
------	----------	---

7. ftell 函数

函数原型： `long ftell(FILE *fp)`

功能：返回位置指针当前位置(用相对文件开头的位移量表示)

返回值：成功，返回当前位置指针位置（长整型）；失败，返回-1L，

可以利用 fseek 和 ftell 函数实现求文件的元素个数。示例如下：

```
#include <iostream>
using namespace std;

int main()
{
    FILE* fp1 = fopen("c:\\a.txt", "w");
    if(NULL == fp1)
    {
        cout << "Can not open this file!" << endl;
        exit(0);
    }

    char c = '\0';

    while((c = getchar()) != '#')
    {
        fputc(c, fp1);
    }

    fseek(fp1, 0L, 2); //定位到文件末尾
    cout << ftell(fp1) << endl; //告诉当前文件指针的位置

    fclose(fp1);
    return 0;
}
```

8. ferror 函数

函数原型： `int ferror(FILE *fp)`

功能：测试文件是否出现错误

返回值：未出错，0；出错，非 0

说明：

每次调用文件输入输出函数，均产生一个新的 ferror 函数值，所以应及时测试

fopen 打开文件时，ferror 函数初值自动置为 0

9. remove 函数

函数原型: `int remove(char* path)`

功能: 删除指定路径的文件

返回值: 成功删除, 0; 否则, -1

示例如下:

```
#include <stdio.h>

void main()
{
    if(-1 == remove("d:\\c.txt")) //删除 d 盘下的 c.txt
    {
        printf("remove failed!\n");
    }
}
```

10. clearerr 函数

函数原型: `void clearerr(FILE *fp)`

功能: 使文件错误标志置为 0

返回值: 无

说明: 出错后, 错误标志一直保留, 直到对同一文件调 `clearerr(fp)`或 `rewind` 或任何其它一个输入输出函数

示例如下:

```
#include <stdio.h>

int main(void)
{
    FILE *fp = fopen("abc.txt", "w");
    getc(fp); //此时没有内容, 读取失败
    if (ferror(fp))
    {
        printf("Error reading from abc.txt\n");
        clearerr(fp); //清除错误标志
    }
    if(!ferror(fp))
        printf("Error indicator cleared!\n");
    fclose(fp);
    return 0;
}
```

11. 系统自动打开和关闭三个标准文件:

标准输入-----键盘	<code>stdin</code>
标准输出-----显示器	<code>stdout</code>
标准出错输出-----显示器	<code>stderr</code>

附录 B：内存映射和普通文件访问的区别

在讲述文件映射的概念时，不可避免的要牵涉到虚存(SVR 4 的 VM)。实际上，文件映射是虚存的中心概念，文件映射一方面给用户提供了一组措施，好似用户将文件映射到自己地址空间的某个部分，使用简单的内存访问指令读写文件；另一方面，它也可以用于内核的基本组织模式，在这种模式种，内核将整个地址空间视为诸如文件之类的一组不同对象的映射。中的传统文件访问方式是，首先用 open 系统调用打开文件，然后使用 read, write 以及 lseek 等调用进行顺序或者随即的 I/O。这种方式是非常低效的，每一次 I/O 操作都需要一次系统调用。另外，如果若干个进程访问同一个文件，每个进程都要在自己的地址空间维护一个副本，浪费了内存空间。而如果能够通过一定的机制将页面映射到进程的地址空间中，也就是说首先通过简单的产生某些内存管理数据结构完成映射的创建。当进程访问页面时产生一个缺页中断，内核将页面读入内存并且更新页表指向该页面。而且这种方式非常便于同一副本的共享。

VM 是面向对象的方法设计的，这里的对象是指内存对象：内存对象是一个软件抽象的概念，它描述内存区与后备存储之间的映射。系统可以使用多种类型的后备存储，比如交换空间，本地或者远程文件以及帧缓存等等。VM 系统对它们统一处理，采用同一操作集操作，比如读取页面或者回写页面等。每种不同的后备存储都可以用不同的方法实现这些操作。这样，系统定义了一套统一的接口，每种后备存储给出自己的实现方法。这样，进程的地址空间就被视为一组映射到不同数据对象上的的映射组成。所有有效地址就是那些映射到数据对象上的地址。这些对象为映射它的页面提供了持久性的后备存储。映射使得用户可以直接寻址这些对象。

值得提出的是，VM 体系结构独立于 Unix 系统，所有的 Unix 系统语义，如正文，数据及堆栈区都可以建构在基本 VM 系统之上。同时，VM 体系结构也是独立于存储管理的，存储管理是由操作系统实施的，如：究竟采取什么样的对换和请求调页算法，究竟是采取分段还是分页机制进行存储管理，究竟是如何将虚拟地址转换成为物理地址等等(Linux 中是一种叫 Three Level Page Table 的机制)，这些都与内存对象的概念无关。

下面介绍 Linux 中 VM 的实现。

一个进程应该包括一个 mm_struct(memory manage struct)，该结构是进程虚拟地址空间的抽象描述，里面包括了进程虚拟空间的一些管理信息：start_code, end_code, start_data, end_data, start_brk, end_brk 等等信息。另外，也有一个指向进程虚存区表(vm_area_struct: virtual memory area)的指针，该链是按照虚拟地址的增长顺序排列的。在 Linux 进程的地址空间被分作许多区(vma)，每个区(vma)都对应虚拟地址空间上一段连续的区域，vma 是可以被共享和保护独立实体，这里的 vma 就是前面提到的内存对象。下面是 vm_area_struct 的结构，其中，前半部分是公共的，与类型无关的一些数据成员，如：指向 mm_struct 的指针，地址范围等等，后半部分则是与类型相关的成员，其中最重要的是一个指向 vm_operation_struct 向量表的指针 vm_ops，vm_pos 向量表是一组虚函数，定义了与 vma 类型无关的接口。每一个特定的子类，即每种 vma 类型都必须在向量表中实现这些操作。这里包括了：open, close, unmap, protect, sync, nopage, wppage, swapout 这些操作。

```
struct vm_area_struct {
    /*公共的，与 vma 类型无关的 */
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct * vm_next;
    pgprot_t vm_page_prot;
    unsigned long vm_flags;
    short vm_avl_height;
    struct vm_area_struct * vm_avl_left;
    struct vm_area_struct * vm_avl_right;
    struct vm_area_struct * vm_next_share;
    struct vm_area_struct ** vm_pprev_share;

    /* 与类型相关的 */
    struct vm_operations_struct * vm_ops;
    unsigned long vm_pgoff;
    struct file * vm_file;
```

```
unsigned long vm_raend;
void * vm_private_data;
};
vm_ops: open, close, no_page, swapin, swapout.....
```

介绍完 VM 的基本概念后, 我们可以讲述 mmap 和 munmap 系统调用了. mmap 调用实际上就是一个内存对象 vma 的创建过程, mmap 的调用格式是:

```
void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

其中 start 是映射地址, length 是映射长度, 如果 flags 的 MAP_FIXED 不被置位, 则该参数通常被忽略, 而查找进程地址空间中第一个长度符合的空闲区域; Fd 是映射文件的文件句柄, offset 是映射文件中的偏移地址; prot 是映射保护权限, 可以是 PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE, flags 则是指映射类型, 可以是 MAP_FIXED, MAP_PRIVATE, MAP_SHARED, 该参数必须被指定为 MAP_PRIVATE 和 MAP_SHARED 其中之一, MAP_PRIVATE 是创建一个写时拷贝映射(copy-on-write), 也就是说如果有多个进程同时映射到一个文件上, 映射建立时只是共享同样的存储页面, 但是某进程企图修改页面内容, 则复制一个副本给该进程私用, 它的任何修改对其它进程都不可见. 而 MAP_SHARED 则无论修改与否都使用同一副本, 任何进程对页面的修改对其它进程都是可见的.

mmap 系统调用的实现过程是:

- 1.先通过文件系统定位要映射的文件;
- 2.权限检查, 映射的权限不会超过文件打开的方式, 也就是说如果文件是以只读方式打开, 那么则不允许建立一个可写映射;
- 3.创建一个 vma 对象, 并对之进行初始化;
- 4.调用映射文件的 mmap 函数, 其主要工作是给 vm_ops 向量表赋值;
- 5.将该 vma 链入该进程的 vma 链表中, 如果可以和前后的 vma 合并则合并;
- 6.如果是要求 VM_LOCKED(映射区不被换出)方式映射, 则发出缺页请求, 把映射页面读入内存中.

```
munmap(void * start, size_t length):
```

该调用可以看作是 mmap 的一个逆过程. 它将进程中从 start 开始 length 长度的一段区域的映射关闭, 如果该区域不是恰好对应一个 vma, 则有可能会分割几个或几个 vma.

```
msync(void * start, size_t length, int flags):
```

把映射区域的修改回写到后备存储中. 因为 munmap 时并不保证页面回写, 如果不调用 msync, 那么有可能在 munmap 后丢失对映射区的修改. 其中 flags 可以是 MS_SYNC, MS_ASYNC, MS_INVALIDATE, MS_SYNC 要求回写完成后才返回, MS_ASYNC 发出回写请求后立即返回, MS_INVALIDATE 使用回写的内容更新该文件的其它映射. 该系统调用是通过调用映射文件的 sync 函数来完成工作的.

```
brk(void * end_data_segement):
```

将进程的数据段扩展到 end_data_segement 指定的地址, 该系统调用和 mmap 的实现方式十分相似, 同样是产生一个 vma, 然后指定其属性. 不过在此之前需要做一些合法性检查, 比如该地址是否大于 mm->end_code, end_data_segement 和 mm->brk 之间是否还存在其它 vma 等等. 通过 brk 产生的 vma 映射的文件为空, 这和匿名映射产生的 vma 相似, 关于匿名映射不做进一步介绍. 库函数 malloc 就是通过 brk 实现的.

Linux 提供了内存映射函数 mmap, 它把文件内容映射到一段内存上(准确说是虚拟内存上), 通过对这段内存的读取和修改, 实现对文件的读取和修改, 先来看一下 mmap 的函数声明:

头文件:

```
<unistd.h>
```

```
<sys/mman.h>
```

原型: void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offsize);

返回值: 成功则返回映射区起始地址, 失败则返回 MAP_FAILED(-1).

参数:

addr: 指定映射的起始地址, 通常设为 NULL, 由系统指定.

length: 将文件的多大长度映射到内存.

prot: 映射区的保护方式, 可以是:

PROT_EXEC: 映射区可被执行.

PROT_READ: 映射区可被读取.

PROT_WRITE: 映射区可被写入.

PROT_NONE: 映射区不能存取.

flags: 映射区的特性, 可以是:

MAP_SHARED: 对映射区域的写入数据会复制回文件, 且允许其他映射该文件的进程共享.

MAP_PRIVATE: 对映射区域的写入操作会产生一个映射的复制(copy-on-write), 对此区域所做的修改不会写回原文件.

此外还有其他几个 flags 不很常用, 具体查看 linux C 函数说明.

fd: 由 open 返回的文件描述符, 代表要映射的文件.

offset: 以文件开始处的偏移量, 必须是分页大小的整数倍, 通常为 0, 表示从文件头开始映射.

下面说一下内存映射的步骤:

用 open 系统调用打开文件, 并返回描述符 fd.

用 mmap 建立内存映射, 并返回映射首地址指针 start.

对映射(文件)进行各种操作, 显示(sprintf), 修改(sprintf).

用 munmap(void *start, size_t lenght)关闭内存映射.

用 close 系统调用关闭文件 fd.

注意事项:

在修改映射的文件时, 只能在原长度上修改, 不能增加文件长度, 因为内存是已经分配好的.

Linux-mmap 函数介绍

mmap 函数是 unix/linux 下的系统调用, 来看《Unix Network programming》卷二 12.2 节对 mmap 的介绍:

The mmap function maps either a file or a Posix shared memory object into the address space of a process. We use this function for three purposes:

1. with a regular file to provide memory-mapped I/O
2. with special files to provide anonymous memory mappings
3. with shm_open to provide Posix shared memory between unrelated processes

mmap 系统调用并不是完全为了用于共享内存而设计的。它本身提供了不同于一般对普通文件的访问方式, 进程可以像读写内存一样对普通文件的操作。而 Posix 或系统 V 的共享内存 IPC 则纯粹用于共享目的, 当然 mmap() 实现共享内存也是其主要应用之一。

mmap 系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后, 进程可以像访问普通内存一样对文件进行访问, 不必再调用 read(), write() 等操作。

我们的程序中大量运用了 mmap, 用到的正是 mmap 的这种“像访问普通内存一样对文件进行访问”的功能。实践证明, 当要对一个文件频繁的访问, 并且指针来回移动时, 调用 mmap 比用常规的方法快很多。

来看看 mmap 的定义:

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

参数 fd 为即将映射到进程空间的文件描述字, 一般由 open() 返回, 同时, fd 可以指定为 -1, 此时须指定 flags 参数中的 MAP_ANON, 表明进行的是匿名映射 (不涉及具体的文件名, 避免了文件的创建及打开, 很显然只能用于具有亲缘关系的进程间通信)。

len 是映射到调用进程地址空间的字节数, 它从被映射文件开头 offset 个字节开始算起。

prot 参数指定共享内存的访问权限。可取如下几个值的或: PROT_READ (可读), PROT_WRITE (可写), PROT_EXEC (可执行), PROT_NONE (不可访问)。

flags 由以下几个常值指定: MAP_SHARED, MAP_PRIVATE, MAP_FIXED。其中, MAP_SHARED, MAP_PRIVATE 必选其一, 而 MAP_FIXED 则不推荐使用。

如果指定为 MAP_SHARED, 则对映射的内存所做的修改同样影响到文件。如果是 MAP_PRIVATE, 则对映射的内存所做的修改仅对该进程可见, 对文件没有影响。

offset 参数一般设为 0, 表示从文件头开始映射。

参数 addr 指定文件应被映射到进程空间的起始地址, 一般被指定一个空指针, 此时选择起始地址的任务留给内核来完成。函数的返回值为最后文件映射到进程空间的地址, 进程可直接操作起始地址为该值的有效地址。

最后, 举个例子来结束本段内容。Fileinformation 数组是以二进制的形式写进一个叫 inforindex 的文件中。那么, 当要访问 Fileinformation 数组时, 代码类似这样:

```
struct stat st;
```

深圳信盈达科技有限公司 专业提供单片机、嵌入式、ARM、LINUX、Android、PCB、FPGA 等技术培训、技术方案。

```
char buffer = "inforindex";
Fileinformation * _fileinfoIndexptr = NULL;
if (stat(buffer, &st) < 0)
{
    fprintf(stderr, "error to stat %s\n", buffer);
    exit(-1);
}
// mmap the inforindex to _fileinfoIndexptr
int fd = open(buffer, O_RDONLY);
if (fd < 0)
{
    printf("error to open %s\n", buffer);
    exit(-1);
}

_fileinfoIndexptr = (Fileinformation *)mmap(NULL, st.st_size, PROT_READ, MAP_SHARED, fd, 0);
if (MAP_FAILED == _fileinfoIndexptr)
{
    printf("error to mmap %s\n", buffer);
    close(fd);
    exit(-1);
}
close(fd);
```

下面这个例子显示了把文件映射到内存的方法

源代码是：

```
/******关于本文档*****
*filename: mmap.c
*purpose: 说明调用 mmap 把文件映射到内存的方法
*Hope:希望越来越多的人贡献自己的力量，为科学技术发展出力
* 科技站在巨人的肩膀上进步更快！感谢有开源前辈的贡献！
*****/

#include <sys / mman.h> /* for mmap and munmap */
#include <sys / types.h> /* for open */
#include <sys / stat.h> /* for open */
#include <fcntl.h> /* for open */
#include <unistd.h> /* for lseek and write */
#include <stdio.h>

int main(int argc, char ** argv)
{
    int fd;
    char * mapped_mem, *p;
    int flength = 1024;
    void * start_addr = 0;
    fd = open(argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    flength = lseek(fd, 1, SEEK_END);
    write(fd, "\0", 1); /* 在文件最后添加一个空字符，以便下面 printf 正常工作 */
    lseek(fd, 0, SEEK_SET);
```

```
mapped_mem = mmap(start_addr, flength, PROT_READ, //允许读
    MAP_PRIVATE, //不允许其它进程访问此内存区域
    fd, 0);
```

```
/* 使用映射区域. */
```

```
printf("%s\n", mapped_mem); /* 为了保证这里工作正常, 参数传递的文件名最好是一个文本文件 */
close(fd);
munmap(mapped_mem, flength);
return 0;
```

```
}
```

编译运行此程序:

```
gcc -Wall mmap.c
```

```
./a.out text_filename
```

上面的方法因为用了 `PROT_READ`, 所以只能读取文件里的内容, 不能修改, 如果换成 `PROT_WRITE` 就可以修改文件的内容了。又由于 用了 `MAAP_PRIVATE` 所以只能此进程使用此内存区域, 如果换成 `MAP_SHARED`, 则可以被其它进程访问, 比如下面的:

```
#include <sys / mman.h> /* for mmap and munmap */
#include <sys / types.h> /* for open */
#include <sys / stat.h> /* for open */
#include <fcntl.h> /* for open */
#include <unistd.h> /* for lseek and write */
#include <stdio.h>
#include <string.h> /* for memcpy */
```

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

```
{
```

```
    int fd;
    char * mapped_mem, *p;
    int flength = 1024;
    void * start_addr = 0;
    fd = open(argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    flength = lseek(fd, 1, SEEK_END);
    write(fd, "\0", 1); /* 在文件最后添加一个空字符, 以便下面 printf 正常工作 */
```

```
    lseek(fd, 0, SEEK_SET);
    start_addr = 0x80000;
    mapped_mem = mmap(start_addr, flength, PROT_READ | PROT_WRITE, //允许写入
        MAP_SHARED, //允许其它进程访问此内存区域
        fd, 0);
```

```
/* 使用映射区域. */
```

```
printf("%s\n", mapped_mem);
/* 为了保证这里工作正常, 参数传递的文件名最好是一个文本文件 */
while ((p = strstr(mapped_mem, "Hello"))) { /* 此处来修改文件 内容 */
    memcpy(p, "Linux", 5);
    p += 5;
}
close(fd);
munmap(mapped_mem, flength);
```

```
    return 0;  
}
```


附录 C: ping 命令解析

ping 命令是用来查看网络上另一个主机系统的网络连接是否正常的工具。ping 命令的工作原理是：向网络上的另一个主机系统发送 ICMP 报文，如果指定系统得到了报文，它将把报文一模一样地传回给发送者，这有点象潜水艇声纳系统中使用的发声装置。

例如，在 Linux 终端上执行 ping localhost 命令将会看到以下结果

```
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of data.
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=255 time=112 usec
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=255 time=79 usec
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=2 ttl=255 time=78 usec
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=3 ttl=255 time=82 usec

--- localhost.localdomain ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.078/0.087/0.112/0.018 ms
```

由上面的执行结果可以看到，ping 命令执行后显示出被测试系统主机名和相应 IP 地址、返回给当前主机的 ICMP 报文序号、ttl 生存时间和往返时间 rtt（单位是毫秒，即千分之一秒）。要写一个模拟 ping 命令，这些信息有启示作用。

要真正了解 ping 命令实现原理，就要了解 ping 命令所使用到的 TCP/IP 协议。

ICMP(Internet Control Message Protocol,网际控制报文协议)是为网关和目标主机而提供的一种差错控制机制，使它们在遇到差错时能把错误报告给报文源发方。ICMP 协议是 IP 层的一个协议，但是由于差错报告在发送给报文源发方时可能也要经过若干子网，因此牵涉到路由选择等问题，所以 ICMP 报文需通过 IP 协议来发送。ICMP 数据报的数据发送前需要两级封装：首先添加 ICMP 报头形成 ICMP 报文，再添加 IP 报头形成 IP 数据报。如下表 C.1 所示

表 C. 1

IP 报头
ICMP 报头
ICMP 数据报

IP 报头格式

由于 IP 层协议是一种点对点的协议，而非端对端的协议，它提供无连接的数据报服务，没有端口的概念，因此很少使用 bind()和 connect() 函数，若有使用也只是用于设置 IP 地址。发送数据使用 sendto()函数，接收数据使用 recvfrom()函数。IP 报头格式如下表 C.2:

表 C. 2

版本号 VER	IP 报头长度 IHL	服务类型 TOS	数据报长度 TL
报文标志 ID	报文标志 F	分段偏移量 FO	
生存时间 TTL	协议号 PORT	报头校验和	
源地址			
目标地址			
任选项和填充位			

在 Linux 中，IP 报头格式数据结构(<netinet/ip.h>)定义如下：

```
struct ip
{
#ifdef __BYTE_ORDER == __LITTLE_ENDIAN
```



```
    unsigned int ip_hl:4;      /* header length */
    unsigned int ip_v:4;      /* version */
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int ip_v:4;      /* version */
    unsigned int ip_hl:4;      /* header length */
#endif
    u_int8_t ip_tos;          /* type of service */
    u_short ip_len;           /* total length */
    u_short ip_id;            /* identification */
    u_short ip_off;           /* fragment offset field */
#define IP_RF 0x8000          /* reserved fragment flag */
#define IP_DF 0x4000          /* dont fragment flag */
#define IP_MF 0x2000          /* more fragments flag */
#define IP_OFFMASK 0x1fff     /* mask for fragmenting bits */
    u_int8_t ip_ttl;          /* time to live */
    u_int8_t ip_p;            /* protocol */
    u_short ip_sum;           /* checksum */
    struct in_addr ip_src, ip_dst; /* source and dest address */
};
```

其中 ping 程序只使用以下数据：

IP 报头长度 IHL（Internet Header Length）以 4 字节为一个单位来记录 IP 报头的长度，是上述 IP 数据结构的 ip_hl 变量。

生存时间 TTL（Time To Live）以秒为单位，指出 IP 数据报能在网络上停留的最长时间，其值由发送方设定，并在经过路由的每一个节点时减一，当该值为 0 时，数据报将被丢弃，是上述 IP 数据结构的 ip_ttl 变量。

ICMP 报头格式

ICMP 报文分为两种，一是错误报告报文，二是查询报文。每个 ICMP 报头均包含类型、编码和校验和这三项内容，长度为 8 位，8 位和 16 位，其余选项则随 ICMP 的功能不同而不同。

Ping 命令只使用众多 ICMP 报文中的两种：“请求回送”(ICMP_ECHO)和“请求回应”(ICMP_ECHOREPLY)。在 Linux 中定义如下：

```
    #define ICMP_ECHO    0
    #define ICMP_ECHOREPLY  8
```

这两种 ICMP 类型报头格式如下表 C.3：

表 C. 3

类型 TYPE (8 或 0)	编码 CODE (没有使用)	校验和 CHECKSUM
标志符 Identifier		顺序号 Sequence NO

在 Linux 中 ICMP 数据结构(<netinet/ip_icmp.h>)定义如下：

```
struct icmp
{
    u_int8_t icmp_type; /* type of message, see below */
    u_int8_t icmp_code; /* type sub code */
    u_int16_t icmp_cksum; /* ones complement checksum of struct */
    union
    {
        u_char ih_pptr; /* ICMP_PARAMPROB */
```

```
struct in_addr ih_gwaddr;    /* gateway address */
struct ih_idseq             /* echo datagram */
{
    u_int16_t icd_id;
    u_int16_t icd_seq;
} ih_idseq;
u_int32_t ih_void;

/* ICMP_UNREACH_NEEDFRAG -- Path MTU Discovery (RFC1191) */
struct ih_pmtu
{
    u_int16_t ipm_void;
    u_int16_t ipm_nextmtu;
} ih_pmtu;

struct ih_rtradv
{
    u_int8_t irt_num_addrs;
    u_int8_t irt_wpa;
    u_int16_t irt_lifetime;
} ih_rtradv;
} icmp_hun;

#define icmp_pptr    icmp_hun.ih_pptr
#define icmp_gwaddr icmp_hun.ih_gwaddr
#define icmp_id      icmp_hun.ih_idseq.icd_id
#define icmp_seq      icmp_hun.ih_idseq.icd_seq
#define icmp_void    icmp_hun.ih_void
#define icmp_pmvoid icmp_hun.ih_pmtu.ipm_void
#define icmp_nextmtu icmp_hun.ih_pmtu.ipm_nextmtu
#define icmp_num_addrs icmp_hun.ih_rtradv.irt_num_addrs
#define icmp_wpa      icmp_hun.ih_rtradv.irt_wpa
#define icmp_lifetime icmp_hun.ih_rtradv.irt_lifetime

union
{
    struct
    {
        u_int32_t its_otime;
        u_int32_t its_rtime;
        u_int32_t its_ttime;
    } id_ts;
    struct
    {
        struct ip idi_ip;
        /* options and then 64 bits of data */
    } id_ip;
    struct icmp_ra_addr id_radv;
    u_int32_t id_mask;
    u_int8_t id_data[1];
}
```

```
    } icmp_dun;
#define icmp_otime    icmp_dun.id_ts.its_otime
#define icmp_rtime    icmp_dun.id_ts.its_rtime
#define icmp_ttime    icmp_dun.id_ts.its_ttime
#define icmp_ip        icmp_dun.id_ip.idi_ip
#define icmp_radv      icmp_dun.id_radv
#define icmp_mask      icmp_dun.id_mask
#define icmp_data      icmp_dun.id_data
};
```

使用宏定义令表达更简洁,其中 ICMP 报头为 8 字节,数据报长度最大为 64K 字节。

这一校验算法称为网际校验和算法,把被校验的数据 16 位进行累加,然后取反码,若数据字节长度为奇数,则数据尾部补一个字节的 0 以凑成偶数。此算法适用于 IPv4、ICMPv4、IGMPv4、ICMPv6、UDP 和 TCP 校验和,更详细的信息请参考 RFC1071,校验和字段为上述 ICMP 数据结构的 icmp_cksum 变量。

标识符用于唯一标识 ICMP 报文,为上述 ICMP 数据结构的 icmp_id 宏所指的变量。

序号 ping 命令的 icmp_seq 便从这里读出,代表 ICMP 报文的发送顺序,为上述 ICMP 数据结构的 icmp_seq 宏所指的变量。

ICMP 数据报

ping 命令中需要显示的信息,包括 icmp_seq 和 ttl 都已有实现的办法,但还缺 rtt 往返时间。为了实现这一功能,可利用 ICMP 数据报携带一个时间戳。使用以下函数生成时间戳:

```
#include
int gettimeofday(struct timeval *tp,void *tzp)
其中 timeval 结构如下:
struct timeval{
    long tv_sec;
    long tv_usec;
}
```

其中 tv_sec 为秒数, tv_usec 微秒数。在发送和接收报文时由 gettimeofday 分别生成两个 timeval 结构,两者之差即为往返时间,即 ICMP 报文发送与接收的时间差,而 timeval 结构由 ICMP 数据报携带,tzp 指针表示时区,一般都不使用,赋 NULL 值。

数据统计

系统自带的 ping 命令当它接送完所有 ICMP 报文后,会对所有发送和所有接收的 ICMP 报文进行统计,从而计算 ICMP 报文丢失的比率。为达此目的,定义两个全局变量:接收计数器和发送计数器,用于记录 ICMP 报文接受和发送数目。丢失数目=发送总数-接收总数,丢失比率=丢失数目/发送总数。

现给出模拟 Ping 程序功能的代码如下:

```
/*
*****
* 作者:
* 时间:
* 名称: myping.c
* 说明:本程序用于演示 ping 命令的实现原理
*****
#include <stdio.h>
#include <signal.h>
#include <arpa/inet.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#include <setjmp.h>
#include <errno.h>

#define PACKET_SIZE      4096
#define MAX_WAIT_TIME    5
#define MAX_NO_PACKETS   3

char sendpacket[PACKET_SIZE];
char recvpacket[PACKET_SIZE];
int sockfd, datalen=56;
int nsend=0, nreceived=0;
struct sockaddr_in dest_addr;
pid_t pid;
struct sockaddr_in from;
struct timeval tvrecv;

void statistics(int signo);
unsigned short cal_chksum(unsigned short *addr, int len);
int pack(int pack_no);
void send_packet(void);
void recv_packet(void);
int unpack(char *buf, int len);
void tv_sub(struct timeval *out, struct timeval *in);

void statistics(int signo)
{
    printf("\n-----PING statistics-----\n");
    printf("%d packets transmitted, %d received , %%%d lost\n", nsend, nreceived,
        (nsend - nreceived) / nsend * 100);
    close(sockfd);
    exit(1);
}
/*校验和算法*/
unsigned short cal_chksum(unsigned short * addr, int len)
{
    int nleft = len;
    int sum = 0;
    unsigned short * w = addr;
    unsigned short answer = 0;

    /*把 ICMP 报头二进制数据以 2 字节为单位累加起来*/
```

```
while (nleft > 1)
{
    sum += *w++;
    nleft -= 2;
}
/*若 ICMP 报头为奇数个字节，会剩下最后一字节。把最后一个字节视为一个 2 字节数据的高字节，这个 2 字节数据的低字节为 0，继续累加*/
if (nleft == 1)
{
    *(unsigned char *)&answer = *(unsigned char *)w;
    sum += answer;
}
sum = (sum >> 16) + (sum & 0xffff);
sum += (sum >> 16);
answer = ~sum;
return answer;
}
/*设置 ICMP 报头*/
int pack(int pack_no)
{
    int i, packsize;
    struct icmp * icmp;
    struct timeval * tval;

    icmp = (struct icmp *)sendpacket;
    icmp->icmp_type = ICMP_ECHO;
    icmp->icmp_code = 0;
    icmp->icmp_cksum = 0;
    icmp->icmp_seq = pack_no;
    icmp->icmp_id = pid;
    packsize = 8 + datalen;
    tval = (struct timeval *)icmp->icmp_data;
    gettimeofday(tval, NULL); /*记录发送时间*/
    icmp->icmp_cksum = cal_checksum((unsigned short *)icmp, packsize); /*校验算法*/
    return packsize;
}

/*发送三个 ICMP 报文*/
void send_packet()
{
    int packetsize;
    while (nsend < MAX_NO_PACKETS)
    {
        nsend++;
        packetsize = pack(nsend); /*设置 ICMP 报头*/
        if (sendto(sockfd, sendpacket, packetsize, 0,
            (struct sockaddr *)& dest_addr, sizeof(dest_addr)) < 0)
        {
            perror("sendto error");
            continue;
        }
        sleep(1); /*每隔一秒发送一个 ICMP 报文*/
    }
}
```

```
    }
}

/*接收所有 ICMP 报文*/
void recv_packet()
{
    int n, fromlen;
    extern int errno;

    signal(SIGALRM, statistics);
    fromlen = sizeof(from);
    while (nreceived < nsend)
    { alarm(MAX_WAIT_TIME);
    if ((n = recvfrom(sockfd, recvpacket, sizeof(recvpacket), 0,
        (struct sockaddr *) &from, &fromlen)) < 0)
    { if (errno == EINTR)continue;
    perror("recvfrom error");
    continue;
    }
    gettimeofday(&tvrecv, NULL); /*记录接收时间*/
    if (unpack(recvpacket, n) == -1)continue;
    nreceived++;
    }

}

/*剥去 ICMP 报头*/
int unpack(char * buf, int len)
{
    int i, iphdrlen;
    struct ip * ip;
    struct icmp * icmp;
    struct timeval * tvsend;

    double rtt;

    ip = (struct ip *)buf;
    iphdrlen = ip->ip_hl << 2; /*求 ip 报头长度,即 ip 报头的长度标志乘 4*/
    icmp = (struct icmp *) (buf + iphdrlen); /*越过 ip 报头,指向 ICMP 报头*/
    len -= iphdrlen; /*ICMP 报头及 ICMP 数据报的总长度*/
    if (len < 8) /*小于 ICMP 报头长度则不合理*/
    { printf("ICMP packets's length is less than 8\n");
    return -1;
    }

    /*确保所接收的是我所发的 ICMP 的回应*/
    if ((icmp->icmp_type == ICMP_ECHOREPLY) && (icmp->icmp_id == pid))
    { tvsend = (struct timeval *)icmp->icmp_data;
    tv_sub(&tvrecv, tvsend); /*接收和发送的时间差*/
    rtt = tvrecv.tv_sec * 1000 + tvrecv.tv_usec / 1000; /*以毫秒为单位计算 rtt*/
```

```
/*显示相关信息*/
printf("%d byte from %s: icmp_seq=%u ttl=%d rtt=%.3f ms\n",
       len,
       inet_ntoa(from.sin_addr),
       icmp->icmp_seq,
       ip->ip_ttl,
       rtt);
}
else return -1;
}

main(int argc, char * argv[])
{
    struct hostent * host;
    struct protoent * protocol;
    unsigned long inaddr = 0l;
    int waittime = MAX_WAIT_TIME;
    int size = 50 * 1024;

    if (argc < 2)
    { printf("usage:%s hostname/IP address\n", argv[0]);
      exit(1);
    }

    if ((protocol = getprotobyname("icmp")) == NULL)
    { perror("getprotobyname");
      exit(1);
    }
    /*生成使用 ICMP 的原始套接字,这种套接字只有 root 才能生成*/
    if ((sockfd = socket(AF_INET, SOCK_RAW, protocol->p_proto)) < 0)
    { perror("socket error");
      exit(1);
    }
    /* 回收 root 权限,设置当前用户权限*/
    setuid(getuid());
    /*扩大套接字接收缓冲区到 50K 这样做主要为了减小接收缓冲区溢出的
    的可能性,若无意中 ping 一个广播地址或多播地址,将会引来大量应答*/
    setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
    bzero(&dest_addr, sizeof(dest_addr));
    dest_addr.sin_family = AF_INET;

    /*判断是主机名还是 ip 地址*/
    if (inaddr = inet_addr(argv[1]) == INADDR_NONE)
    { if ((host = gethostbyname(argv[1])) == NULL) /*是主机名*/
      { perror("gethostbyname error");
        exit(1);
      }
      memcpy((char *) & dest_addr.sin_addr, host->h_addr, host->h_length);
    }
```



```
    }
    else /*是 ip 地址*/
        memcpy((char *) & dest_addr, (char *) & inaddr, host->h_length);
    /*获取 main 的进程 id,用于设置 ICMP 的标志符*/
    pid = getpid();
    printf("PING %s(%s): %d bytes data in ICMP packets.\n", argv[1],
        inet_ntoa(dest_addr.sin_addr), datalen);
    send_packet(); /*发送所有 ICMP 报文*/
    recv_packet(); /*接收所有 ICMP 报文*/
    statistics(SIGALRM); /*进行统计*/

    return 0;

}
/*两个 timeval 结构相减*/
void tv_sub(struct timeval * out, struct timeval * in)
{
    if ((out->tv_usec -= in->tv_usec) < 0)
    { --out->tv_sec;
        out->tv_usec += 1000000;
    }
    out->tv_sec -= in->tv_sec;
}
/*----- The End -----*/
```

特别注意：

只有 root 用户才能利用 socket()函数生成原始套接字，要让 Linux 的一般用户能执行以上程序，需进行如下的特别操作：

用 root 登陆，编译以上程序：gcc -o myping myping.c，其目的有二：一是编译，二是让 myping 属于 root 用户。

再执行 chmod u+s myping，目的是把 myping 程序设成 SUID 的属性。

退出 root，用一般用户登陆，执行 ./myping www.cn.ibm.com，有以下执行结果：

PING www.cn.ibm.com(202.95.2.148): 56 bytes data in ICMP packets.

64 byte from 202.95.2.148: icmp_seq=1 ttl=242 rtt=3029.000 ms

64 byte from 202.95.2.148: icmp_seq=2 ttl=242 rtt=2020.000 ms

64 byte from 202.95.2.148: icmp_seq=3 ttl=242 rtt=1010.000 ms

-----PING statistics-----

3 packets transmitted, 3 received , %0 lost

附录 D：大端小端

大端格式：

在这种格式中，字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中

小端格式：

与大端存储格式相反，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节

请写一个 C 函数，若处理器是 Big_endian 的，则返回 0；若是 Little_endian 的，则返回 1

```
int checkCPU()  
{  
    {  
        union w  
        {  
            int  a;  
            char b;  
        } c;  
        c.a = 1;  
        return(c.b ==1);  
    }  
}
```

嵌入式系统开发者应该对 Little-endian 和 Big-endian 模式非常了解。例如，16bit 宽的数 0x1234 在 Little-endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）如表 D.1 所示：

表 D. 1

内存地址	0x4000	0x4001
存放内容	0x34	0x12

而在 Big-endian 模式 CPU 内存中的存放方式则如表 D.2 所示：

表 D. 2

内存地址	0x4000	0x4001
存放内容	0x12	0x34

32bit 宽的数 0x12345678 在 Little-endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）如表 D.3 所示：

表 D. 3

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12

而在 Big-endian 模式 CPU 内存中的存放方式如表 D.4 所示：

表 D. 4

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

联合体 union 的存放顺序是所有成员都从低地址开始存放。

```
int big_endian (void)  
{  
    union{  
        long l;
```

```
char c[sizeof(long)];  
}u;  
  
u.l = 1;  
return (u.c[sizeof(long) - 1] == 1);  
}
```

有时候，用 C 语言写程序时需要知道是大端模式还是小端模式。所谓的大端模式，是指数据的低位保存在内存的高地址中，而数据的高位，保存在内存的低地址中；所谓的小端模式，是指数据的低位保存在内存的低地址中，而数据的高位保存在内存的高地址中。为什么会有大小端模式之分呢？这是因为在计算机系统中，我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为 8bit。但是在 C 语言中除了 8bit 的 char 之外，还有 16bit 的 short 型，32bit 的 long 型（要看具体的编译器），另外，对于位数大于 8 位的处理器，例如 16 位或者 32 位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如果将多个字节安排的问题。因此就导致了大端存储模式和小端存储模式。例如一个 16bit 的 short 型 x，在内存中的地址为 0x0010，x 的值为 0x1122，那么 0x11 为高字节，0x22 为低字节。对于大端模式，就将 0x11 放在低地址中，即 0x0010 中，0x22 放在高地址中，即 0x0011 中。小端模式，刚好相反。我们常用的 X86 结构是小端模式，而 KEIL C51 则为大端模式。很多的 ARM，DSP 都为小端模式。有些 ARM 处理器还可以由硬件来选择是大端模式还是小端模式。

下面这段代码可以用来测试一下你的编译器是大端模式还是小端模式：

```
short int x;  
char x0,x1;  
x=0x1122;  
x0=((char*)&x)[0]; //低地址单元  
x1=((char*)&x)[1]; //高地址单元  
若 x0=0x11,则是大端; 若 x0=0x22,则是小端.....
```

本教程到此结束！谢谢！

深圳信盈达科技有限公司

网址：<http://www.edu118.com> 邮箱：superc51@163.com

V1.1 版于 2013 年 06 月 12 日于深圳信盈达科技有限公司研发部完成；

V1.2 版于 2013 年 09 月 16 日修订。