Linux 编程与实例

前 言

当前 windows 操作系统在桌面系统中占据绝对的主流。Linux 操作系统凭着免费、开放源码、性能稳定、可靠的安全性等优势与 windows 展开了激烈的竞争，众多 Linux 自由爱好者和厂家的支持使 Linux 应用越来越广泛。可以说在 windows 下安装使用的大部分软件，Linux 下都有相应免费的开源软件可以使用。国内越来越多的企业基于 Linux 系统做软件开发，众多国内高校和科研院所对 Linux 展开了众多的研究与应用，Linux 在中国的应用越来越广泛，掌握 Linux 编程将使你在 Linux 系统应用程序开发中得心应手。

本书的主要内容

本书主要介绍了 Linux 编程，通过大量的实例使读者掌握 Linux shell 、进程、线程、进程间通信、信号等编程的基本知识。

第一章介绍了 Linux 的基础知识、体系结构、主要特性、目录结构等。
第二章介绍了 Shell 基本命令、Shell 应用、Shell 脚本等。
第三章介绍了 Linux 编辑器和编译工具，介绍了常用的 vi 编辑器、GCC 编译器、GDB 调试器 GNU make 工具，介绍了 Makefile 的编写等，并使用工具自动生成 makefile。
第四章介绍了 Linux 文件和目录，标准 IO 和文件 IO 使读者掌握基于流和文件描述符的两种创建打开文件、写入文件、读取文件、关闭文件等文件的基本操作。目录操作使读者能够创建、删除目录、读取目录文件内容等。
第五章介绍了 Linux 进程和进程控制，介绍了进程的属性、进程管理、进程调度、进程标志、创建进程、执行进程等。
第六章介绍了 Linux 线程及线程控制，介绍了线程的基本知识，线程标识、线程属性、创建线程、终止线程、清理线程等。
第七章介绍了 Linux 进程间通信，管道（有名管道和无名管道）、System-V 消息队列 、System V 信号量、System-V 共享存储、Posix 有名信号量、Posix 条件变量、Posix 共享变量等。
第八章介绍了 Linux 信号，信号的安装、信号的发送、信号屏蔽字等。
第九章介绍了 Linux Lib 库，介绍了 Linux 静态库和动态库的制作。
第十章介绍了 Linux 网络编程，Socket 概述、基本数据结构、函数接口，套接字和数据库的 Server/Client 实现的实例。

本书提供了大量的实例使读者能够边学边用，更快更好地掌握 Linux 编程的基础知识。本书中的大量源代码也将向读者提供。

本书可作为高等院校计算机类、电子类、控制类等专业高年级本科生、研究生学习 Linux 编程的教材，也可作为广大 Linux 培训班的教材和参考书，还可以供广大希望转入 Linux

和嵌入式系统领域的科研和工程技术人员参考使用。

由于时间仓促、编者水平有限，书中的不足之处在所难免，敬请读者批评指正，出错之处请读者发送到 zhouxuhong@gmail.com，欢迎读者交流。

编者：周绪宏
2009 年 1 月

目 录

实 例

第一章 LINUX 基础知识

一.1 Linux 概述

提到 Linux 就不得不先说到 Unix，Unix 是一个强大的多用户、多任务操作系统，支持多种处理器架构，最早由 Ken Thompson、Dennis Ritchie 和 Douglas McIlroy 于 1969 年在 AT&T 的贝尔实验室开发。经过长期的发展和完善，已发展成为一种主流的操作系统技术和基于这种技术的产品大家族。Unix 具有技术成熟、可靠性高、网络和数据库功能强和开放性等特色，可满足各行各业的实际需要，特别是满足企业重要业务的需要，Unix 已经发展成为主要的工作站平台和重要的企业操作平台。Unix 是一种广泛使用的商业操作系统的名称。

由于 Unix 已经成为一种商业操作系统，为了研究与学习操作系统，1983 年 9 月 27 日 Richard Stallman 公开发起了 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。Richard Stallman 最早是在 net.unix-wizards 新闻组上公布该消息，并附带一份《GNU 宣言》解释为何发起该计划，其中一个理由就是要“重现当年软件界合作互助的团结精神”。

GNU 是“GNU's Not Unix”的递归缩写。由于 GNU 要实现 Unix 系统的接口标准，GNU 计划可以分别开发不同的操作系统部件。GNU 计划采用了部分当时已经可自由使用的软件，GNU 计划也开发了大批其他的自由软件。

为保证 GNU 软件可以自由地“使用、复制、修改和发布”，所有 GNU 软件都在一份在禁止其他人添加任何限制的情况下授权所有权利给任何人的协议条款，GNU 通用公共许可证 GPL（GNU General Public License）。

1985 年 Richard Stallman 又创立了自由软件基金会 FSF（Free Software Foundation）来为 GNU 计划提供技术、法律以及财政支持。GNU 计划大部分是由个人自愿无偿贡献，但自由软件基金会有时还是会聘请程序员帮助编写。当 GNU 计划开始逐渐获得成功时，一些商业公司开始介入开发和技术支持。

1990 年 GNU 计划已经开发出的软件包括功能强大的文字编辑器 Emacs、C 语言编译器 GCC、以及大部分 Unix 系统的程序库和工具，唯一依然没有完成的重要组件就是操作系统的内核。

1991 年芬兰赫尔辛基大学学生 Linus Torvalds 编写了与 Unix 兼容的 Linux 操作系统内核并在 GPL 条款下发布。Linux 之后在网上广泛流传，许多程序员参与了开发与修改。1992 年 Linux 与其他 GNU 软件结合，完全自由的操作系统正式诞生。该操作系统往往被称为“GNU/Linux”或简称 Linux。

典型的 Linux 发行版包含 Linux 内核、应用程序和各类工具集。Linux 发行版中出现的各

种系统级和用户级的工具都来自自由软件基金会（Free Software Foundation）的 GNU 项目。Linux 内核和 GNU 工具集都在 GNU 通用公共许可证 GPL（GNU General Public License）下发行。Linux 属于自由软件，用户不需要支付任何费用就可以获得它及其源代码，并可以对它进行修改，但必须遵循 GNU GPL 规则。

一.2 Linux 主要特性

（1）开放性

开放性是指系统遵循标准规范，特别是遵循开放系统互连（OSI）国际标准。凡遵循国际标准所开发的硬件和软件，都能彼此兼容，可方便地实现互连。Linux 系统的各种软件符合标准规范。

（2）多用户

多用户是指系统资源可以被不同用户各自拥有和使用，即每个用户对自己的资源（例如：文件、设备）有特定的权限，互不影响。Linux 系统支持多用户。

（3）多任务

多任务是指计算机同时执行多个程序，而且各个程序的运行相互独立。Linux 系统调度每一个进程平等地访问处理器。由于 CPU 的处理速度非常快，启动的应用程序看起来好像在并行运行。Linux 系统支持多任务。

（4）良好的用户界面

Linux 向用户提供了两种界面：字符界面和图形界面。Linux 的传统用户界面是基于文本的字符界面，即 Shell。Shell 有很强的程序设计能力，用户可方便地用它编制程序，从而为用户扩充系统功能提供了更高级的手段。Linux 还为用户提供了图形用户界面。它利用鼠标、菜单、窗口、滚动条等设施，给用户呈现一个直观、易操作、交互性强的友好的图形化界面。

（5）丰富的网络功能

Linux 在通信和网络功能方面优于其他操作系统。Linux 为用户提供了完善的、强大的网络功能。支持 Internet 是其网络功能之一，Linux 免费提供了大量支持 Internet 的软件；文件传输是其网络功能之二，用户能通过一些 Linux 命令完成内部信息或文件的传输；远程访问是其网络功能之三，Linux 不仅允许进行文件和程序的传输，还为系统管理员和技术人员提供了访问其他系统的窗口。通过这种远程访问的功能，一位技术人员能够有效地为多个系统服务。

（6）可靠的系统安全

Linux 采取了许多安全技术措施，包括对读、写进行权限控制；带保护的子系统；核心授权等，这为网络多用户环境中的用户提供了必要的安全保障。

（7）良好的可移植性

可移植性是指将操作系统从一个平台移植到另一个平台使它仍然能按其自身的方式运行。Linux 是一种可移植的操作系统，能够在从微型计算机到大型计算机的任何环境中和任

何平台上运行。

一.3 Linux 体系结构

严格意义上可将操作系统定义为一种软件，它控制计算机硬件资源，提供硬件的运行环境。此种软件称为内核，相对较小，位于环境的中心，下图显示了 Linux 系统的体系结构。

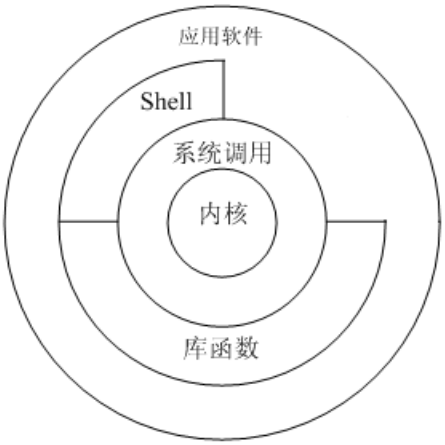

图 **1**：**Linux** 操作系统的体系结构

内核的接口称为系统调用（System Call），库函数建立在系统调用接口之上，应用软件既可使用库函数，也可使用系统调用。Shell 是一种特殊的应用程序，就是一个命令行解释器，它为用户提供了一个向 Linux 内核发送请求以便运行程序的系统级程序，用户在提示符输入的命令都由 Shell 先解释后再传给 Linux 内核。

应用软件如何通过 Shell、系统调用、库函数执行程序是本书讲述的重点。

一.4 Linux 内核

Linux Kernel（内核、核心）是一个庞大而复杂的 Linux 操作系统的核心，采用了子系统和分层的概念很好地进行了组织。

Linux 内核主要由五个子系统组成：进程调度、内存管理、虚拟文件系统、网络接口和进程间通信。

（1）进程调度（sched）

进程调度控制进程对 CPU 的访问。当需要选择下一个进程运行时，由调度程序选择最值得运行的进程。可运行进程实际上是仅等待 CPU 资源的进程，如果某个进程在等待其它资源，则该进程是不可运行进程。Linux 使用了比较简单的基于优先级的进程调度算法选择新的进程。

（2）内存管理（mm）

内存管理允许多个进程安全的共享主内存区域。Linux 内存管理支持虚拟内存，即在计算机中运行的程序，其代码、数据、堆栈的总量可以超过实际内存的大小，操作系统只把当前使用的程序块保留在内存中，其余的程序块则保留在磁盘中，必要时操作系统负责在磁盘和内存间交换程序块。虚拟内存可以是系统中实际物理空间的许多倍，每个进程运行在其独立的虚拟地址空间中。这些虚拟空间相互之间都完全隔离开来，所以进程间不会互相影响。

（3）虚拟文件系统（VFS）

虚拟文件系统为所有的设备提供了统一的接口，隐藏了各种硬件的具体细节。VFS 提供了多达数十种不同的文件系统。虚拟文件系统可以分为逻辑文件系统和设备驱动程序。逻辑文件系统指 Linux 所支持的文件系统，如 ext3、fat 等；设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。

（4）网络接口（Net）

网络接口提供了对各种网络标准的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序。网络协议负责实现每一种可能的网络传输协议；网络设备驱动程序负责与硬件设备通讯，每一种可能的硬件设备都有相应的设备驱动程序。

（5）进程间通信（IPC）

进程间通信支持进程间各种通信机制。Linux 系统的进程间通信基本上是从 Unix 平台的进程间通信继承而来的。最初 Unix IPC 包括：管道、FIFO、信号；System V IPC 包括：System V 消息队列、System V 信号量、System V 共享内存区；Posix IPC 包括：Posix 消息队列、Posix 信号量、Posix 共享内存区。

由于 Unix 版本的多样性，电子电气工程协会（IEEE）开发了一个独立的 Unix 标准，这个新的标准被称为计算机环境的可移植性操作系统界面（Posix）。现有大部分 Unix 和流行版本都是遵循 Posix 标准的，而 Linux 从一开始就遵循 Posix 标准。

Linux 内核组成

Linux 内核是由基本内核本身加上任意数量的内核模块而构成的。在很多情况下，基本内核与大量内核模块是同时编译的，并一起安装或发布，这些都基于 Linus Torvalds 所创建的代码。基本内核总是在系统引导时被加载，而且在运行期间一直驻留；内核模块在初始时有可能被加载，也可能不被加载，而且也可以在运行期间加载或卸载内核模块。

Linux 内核模块系统允许在基本内核编译之后再编译或者单独编译额外模块，额外模块可能是向运行中的 Linux 系统添加硬件设备时创建的，有时也可能是第三方所发布的，一旦内核模块被加载，它就成为运行中内核的一部分，直到被卸载。

Linux 内核命名约定

Linux 内核遵循一种命名编号约定，能够让您知道正在使用内核的重要信息。所使用的约定指出主编号、次编号、修订，在某些情况下，还包括提供商/定制字符串。这种约定适用于多种类型的文件，包括内核源代码档案文件、补丁等。

Linux 内核命名约定：

（1）Linux-主编号.次编号.修订.tar.gz

（2）Linux-主编号.次编号.修订-定制字符串.tar.gz

Linux 内核命名除了用基本的以圆点隔开的序列以外，Linux 内核还遵循了一个约定来区分稳定的分支与实验用的分支。稳定分支使用偶数编号；实验分支使用奇数编号；修订只是顺序编号，反映 bug 的修复以及向后兼容性的改进。定制字符串通常用来描述提供商或者具体特性。

例子：

（1）Linux-2.6.18.tar.gz：表示 Linux 的 2.6 内核源码文件。

（2）Linux-2.4.37-foo.tar.gz：表示来自“Foo Industries”的稳定的 2.4 内核源代码文件。

Linux 内核文件

Linux 基本内核有两个版本：zImage 和 bzImage，前者大小限制在大约 500KB 左右，后者用于更大的内核（最大约 2.5MB）。通常 Linux 发行商都使用 bzImage 内核格式以支持更多的特性。zImage 中的 z 表示使用 gzip 压缩；bzImage 中的 b 表示 big，使用的还是 gzip 压缩。在这两种情况下，由于 Linux 内核文件都是存放在 /boot 目录，所以基本内核通常都被重新命名为 vmlinuz，例如 /boot/vmlinuz-2.6.10-5-386。

在 /boot 目录中，还有一些其他与基本内核相关的文件。System.map 是描述内核符号地址的表。initrd.img 有时候会被基本内核用于在挂载全部文件系统之前在 ramdisk 中创建一个简单的文件系统。

Linux 内核模块

Linux 内核模块中包含额外的内核代码，可以在加载基本内核之后再加载。模块通常提供下列功能之一：

（1）设备驱动程序：支持特定类型的硬件；

（2）文件系统驱动程序：提供读取和写入特定文件系统的能力；

（3）系统调用：大部分在基本内核中都得到了支持，内核模块可以添加或修改系统服务；

（4）网络驱动程序：实现具体的网络协议；

（5）可执行程序加载：解析和加载另外的可执行文件格式。

一.5 Linux 目录结构

目录	内容
/	根目录，Linux 把所有的文件都放在一个目录树里面，/ 是唯一的根目录
/boot	存放操作系统启动时所要用到的程序，比如内核映像等
/bin	二进制（binary）的缩写，存放 Linux 系统的常用命令
/sbin	存放系统管理常用的系统管理程序
/usr/bin	存放系统用户使用的应用程序
/usr/sbin	存放超级用户使用的较高级的管理程序和系统守护程序

/dev	存放所有的设备文件，比如/dev/sda1 代码磁盘第一个分区
/etc	存放系统的各种配置文件，系统启动时需要读取参数进行相应的配置
/lib	存放系统动态链接共享库
/root	超级用户 root 的工作目录
/home	Linux 系统中新加用户的默认工作目录
/mnt	是光驱、软驱、硬盘、移动硬盘等的挂载点
/usr/src	存放 Linux 内核源码
/etc/X11	存放 X 系统所需要的配置文件
/etc/rc.d	存放 Linux 启动和关闭程序时用到的脚本文件
/etc/init.d	存放 Linux 系统服务程序的启动脚本
/proc	存放系统内核与执行程序所需的信息，在内存中产生不占用硬盘空间
/var	存放服务程序的日志信息
/srv	存放一些服务程序启动之后需要提取的数据
/tmp	存放不同程序执行时产生的临时文件

本章小结

本章简单概述了 Linux 的基本知识、主要特性、系统结构、Linux 内核和 Linux 的目录结构。

习题

1.1 Linux 有哪些发行版本?
1.2 Linux 操作系统有哪些优势?
1.3 Linux 内核版本是如何定义的?
1.4 Linux 安装时如何分区？swap 分区的大小?

第二章 LINUX SHELL

安装完 Linux 启动登录后就进入 Linux 图形化界面，这个界面是 Linux 图形化界面 X 窗口系统的一部分。X 窗口系统是 Linux 的一个软件，比较耗费系统资源。若希望更好地享受 Linux 所带来的高效和高稳定性，用户可以选择使用 Linux 的字符界面。用户也可以设置 Linux 启动界面为字符界面。

Linux 图形界面下启动 Shell 有两种方式：一种是 应用程序→附件→终端；另一种是按下 Ctrl+Alt+F1 ~ F6，按下 Ctrl+Alt+F7 恢复图形界面。

用户在字符界面下工作时不是直接和 Linux 内核打交道，而是由命令行解释器接收命令、分析再传给相关的程序。Shell 是一个命令行解释器，它为用户提供了一个向 Linux 内核发送请求以便运行程序的系统级程序，用户可以用 Shell 来启动、挂起、停止程序。用户在提示符输入的命令都由 Shell 解释后传给 Linux 内核。Shell 又是一种程序设计语言，作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

二.1 Shell 命令

二.1.1 基本命令

命令	含义	格式
ls	列出当前目录内容	ls [-al] [文件/目录]
pwd	显示当前工作目录	pwd
cd	切换目录	cd 新路径
mkdir	创建目录	mkdir directory
rmdir	删除目录	rmdir directory 目录必须为空
rm	删除文件或目录	rm [-dfirv][文件或目录]
cp	复制文件或目录	cp file1 file2
touch	创建一个文件	touch file1
cat	显示文件	cat file1
ln	创建链接文件	ln [-s] source_path target_path
date	显示日期时间	date
clock	显示当前时钟	clock
cal	显示本月日历	cal
tree	以树状图列出目录的内容	tree
history	显示历史记录信息	history
df	显示磁盘相关信息	df [选项] （k 或 h）
du	显示目录或文件的大小	du –sh directory

find	查找文件	find [路径] [选项] 文件名
chmod	改变文件或目录属性	chmod[who][选项][权限] 文件
chown	改变文件或目录的用户和组属性	chown -R -h owner file
clear	清除屏幕显示	clear

例 **1**：**Shell** 基本命令使用的例子

```
[root@localhost hongdy]# mkdir test                    创建 test 目录
[root@localhost hongdy]# cd test                       进入 test 目录
[root@localhost test]# touch hello                     用 touch 新建 hello 文件，或者用 vi 创建
[root@localhost test]# ls                              查看当前目录下文件
hello
[root@localhost test]# cp hello world                  拷贝 hello 文件为 world 文件
[root@localhost test]# mv hello welcome                将 hello 文件改名为 welcome 文件
[root@localhost test]# ls                              查看当前文件
welcome  world
[root@localhost test]# rm world                        删除 world 文件
[root@localhost test]# cp welcome /tmp/                将 welcome 拷贝到/tmp 目录下
[root@localhost test]# ls –al /tmp/welcome             查看/tmp/welcome 目录文件
-rw-r--r-- 1 root root 6 Jan 20 21:23 /tmp/welcome
[root@localhost test]# cd ..                           返回到上级目录
[root@localhost hongdy]# cp -a test test2              拷贝 test 目录为 test2 目录
[root@localhost hongdy]# ls test2/                     查看 test2 目录
welcome
[root@localhost hongdy]# rm -fr test                   删除 test 目录
[root@localhost hongdy]# du -sh test2                  查看 test2 目录大小
4.0K    test2/
[root@localhost hongdy]# df                            查看磁盘大小
[root@localhost hongdy]# cd test2                      进入 test2 目录
[root@localhost test2]# chmod 755 welcome              改变 welcome 文件 755 权限
[root@localhost test2]# ls -al welcome
-rwxr-xr-x 1 root root 6 Jan 20 21:22 welcome
 [root@localhost test2]# chmod a+w welcome             所有用户增加可写权限
[root@localhost test2]# ls -al welcome
-rwxrwxrwx 1 root root 6 Jan 20 21:22 welcome
[root@localhost test2]# chmod o-x welcome              其他用户去除可执行权限
[root@localhost test2]# ls -al welcome
-rwxrwxrw- 1 root root 6 Jan 20 21:22 welcome
[root@localhost test2]# chown hongdy:hongdy welcome    改变文件用户和组属性
[root@localhost test2]# ls –al welcome
-rwxr-xrw-  1 hongdy hongdy    6 Jan 20 21:22 welcome
[root@localhost ~]# find / -name welcome               查找根路径下所有名为 welcome 的文件
[root@localhost ~]# tree                               以树状图列出目录的内容
```

[root@localhost ~]# date	显示当前时间
Thu Feb 21 23:29:43 CST 2008	
[root@localhost ~]# history	显示历史记录信息
[root@localhost ~]# cd	进入登入用户的工作目录
[root@localhost ~]# cd ~	进入登入用户的工作目录

二.1.2 系统管理命令

命令	含义	格式
su	切换用户	su [用户名] 默认切换到 root 用户
useradd	添加用户帐号	useradd [选项] 用户名
usermod	设置用户帐号属性	usermod [选项] 属性值
userdel	删除用户帐号	userdel [选项] 用户名
groupadd	添加组帐号	groupadd [选项] 组帐号
groupmod	设置组帐号属性	groupmod [选项] 组帐号
groupdel	删除组帐号	groupdel [选项] 组帐号
passwd	设置帐号密码	passwd 帐号
id	显示用户 id	id [用户名]
groups	显示用户所在的组	groups [用户名]
who	显示登录到系统的用户信息	who
hostname	显示主机名	hostname
hostid	显示登录 ID	hostid
finger	显示所有登录用户	finger
uname	显示系统信息	uname –a
ps	显示当前系统中用户运行的进程	ps [选项]
top	动态显示系统中运行的程序	top
kill	杀掉指定 PID 的进程	kill [选项] PID
free	显示内存状态	free
fdisk	查看磁盘分区情况及分区管理	fdisk –l
dmesg	显示开机信息	dmesg
mount	挂载磁盘分区	mount –t vfat /dev/sda5 /mnt/c
umount	卸载磁盘分区	umount /dev/hdc
rpm	软件安装命令	rpm –ivh xxx.rpm
yum	软件安装命令	yum install/remove/info package
reboot	重新启动机器	reboot
shutdown	关闭或重启 Linux	shutdown [选项] [时间]

例 **2**：系统管理命令的例子

```
新加 embedded 用户并设置密码
# useradd embedded
# passwd embedded
```

Changing password for user embedded.
New UNIX password:
BAD PASSWORD: it does not contain enough DIFFERENT characters
Retype new UNIX password:
passwd: all authentication tokens updated successfully.

进程管理

$ ps	查看当前运行的进程
PID TTY TIME CMD	
2750 pts/4 00:00:00 bash	
28642 pts/4 00:00:01 hello	
28643 pts/4 00:00:00 ps	
$ kill -9 28642	杀死进程号为 28642 的进程
$ ps	进程 28642 已被 kill
PID TTY TIME CMD	
2750 pts/4 00:00:00 bash	
28647 pts/4 00:00:00 ps	
[1]+ Killed ./hello	
[hongdy@localhost ~]$ free	显示内存状态

磁盘挂载

# mount –t vfat /dev/hda5 /mnt/d	将 win 的 D 盘挂载到/mnt/d 目录
# mount -t iso9660 /dev/sr0 /mnt/cdrom	挂载光驱
# mount /dev/sda1 /mnt/usb	挂载移动硬盘
# umount /dev/hdc	卸载光驱

rpm 常用命令

# rpm -ivh xxx.rpm	安装一个软件包
# rpm -Uvh xxx.rpm	升级一个软件包
# rpm -e xxx	卸载一个软件包
# rpm -q < rpm package name>	查询一个软件包
# rpm -ql <rpm package name>	查询软件包安装目录文件
# rpm -qf /usr/bin/ssh	查询文件属于哪个软件包，必须是绝对路径
# rpm2cpio abc.rpm|cpio -idv	解压 rpm 包

yum 常用命令

# yum install <package_name>	用 YUM 安装软件包
# yum remove <package_name>	用 YUM 删除软件包
# yum search <keyword>	使用 YUM 查找软件包
# yum list	列出所有可安装的软件包
# yum list updates	列出所有可更新的软件包
# yum list <package_name>	列出所指定的软件包
# yum info <package_name>	使用 YUM 获取软件包信息

# yum info installed	列出所有可更新的软件包信息
# yum clean packages	除缓存目录(/var/cache/yum)下的软件包
# yum clean headers	清除缓存目录(/var/cache/yum)下的 headers
# yum clean all	清除缓存目录所有软件包

二.1.3 文件压缩相关命令

命令	含义	格式
tar	创建或解压档案.tar 文件	tar [命令] 文件
gzip/gunzip	压缩或解压缩.gz 文件	gzip/gunzip [命令] 文件
bzip2/bunzip2	压缩或解压缩.bz2 文件	bzip2/bunzip2 [命令] 文件
zip/unzip	压缩或解压缩.zip 文件	zip/unzip [命令] 文件
rar	压缩或解压缩.rar 文件	rar [命令] 文件
cpio	压缩或解压缩.cpio 文件	cpio [命令] 文件

例 **3**：文件压缩解压缩的例子

[hongdy@localhost ~]$ ls	假设当前目录下有 linux 目录
[hongdy@localhost ~]$ tar cf linux.tar linux	创建 linux.tar 文件
[hongdy@localhost ~]$ gzip linux.tar	创建 linux.tar.gz 文件
[hongdy@localhost ~]$ tar cfz linux.tar.gz linux	直接创建 linux.tar.gz 文件
[hongdy@localhost ~]$ gzip -d linux.tar.gz	解压 linux.tar.gz 文件
[hongdy@localhost ~]$ gunzip linux.tar.gz	解压 linux.tar.gz 文件
[hongdy@localhost ~]$ tar zxvf linux.tar.gz	解压 linux.tar.gz 文件
[hongdy@localhost ~]$ tar jxvf linux.tar.gz2/bz2	解压 linux.tar.bz2 或 linux.tar.gz2 文件
[hongdy@localhost ~]$ zip –r linux.zip linux	创建 linux.zip 文件
[hongdy@localhost ~]$ unzip linux.zip	解压 linux.zip 到当前目录
[hongdy@localhost ~]$ bzip2 linux.tar	创建 linux.tar.bz2 文件
[hongdy@localhost ~]$ bzip2 –d linux.tar.bz2	解压 linux.tar.bz2 文件，生成 linux.tar 文件
[hongdy@localhost ~]$ bunzip2 linux.tar.bz2	解压 linux.tar.bz2 文件，生成 linux.tar 文件
[hongdy@localhost ~]$ rar a linux.rar linux	创建 linux.rar 文件
[hongdy@localhost ~]$ rar e linux.rar	解压 linux.rar 到当前目录

下面是对压缩命令的详细解释。

二.1.3.1 tar

tar 命令可以用来为文件和目录创建档案。用户可以为某一特定文件创建档案，也可以在档案中改变文件，或者向档案中加入新的文件。

tar 命令用法：tar [命令] 文件
命令：c 创建归档文件；f 使用档案文件或设备；x 从档案文件中释放文件；r 把文件追加到档案文件末尾；v 列出档案文件信息；z 用 gzip 来压缩或解压缩文件；j 用 bz2 解压缩文件。

例子：

# tar cf linux.tar linux	创建 linux.tar 文件，linux 是目录
# tar cfz linux.tar.gz linux	创建 linux.tar.gz 文件，先生成 linux.tar，再用 gzip 压缩
# tar rf linux.tar file	添加 file 文件到 linux.tar 文件末尾
# tar xvf linux.tar	解压 linux.tar 文件，生成 linux 目录
# tar zxvf linux.tar.gz	解压 linux.tar.gz 文件，生成 linux 目录
# tar zxvf linux.tar.gz –d /tmp	解压 linux.tar.gz 文件，生成 linux 目录
# tar jxvf linux.tar.bz2	解压 linux.tar.bz2 文件，生成 linux 目录

二.1.3.2 gzip/gunzip

gzip 命令可以用来将文件压缩成常用的.gz 格式；gunzip 命令则用来解压.gz 文件，gunzip 实际上是 gzip 的硬连接。

gzip 命令用法：gzip [-命令] 文件
命令：-c 将输出写到标准输出并保留原有文件；-d 将压缩文件解压；-l 显示压缩文件信息；-r 递归进入子目录；-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。

gunzip 命令用法：gunzip files.gz　　等于 gzip –d files.gz

例子：

# gzip file.tar	创建 file.tar.gz 压缩文件，生成 file.tar.gz 文件
# gzip -9 file.tar.gz	创建 file.tar.gz 压缩文件，压缩比为 9
# gzip –l file.tar.gz	查看 file.tar.gz 压缩文件信息
# gzip –d file.tar.gz	解压 file.tar.gz 压缩文件，生成 file.tar 文件
# gunzip file.tar.gz	解压 file.tar.gz 压缩文件，生成 file.tar 文件

二.1.3.3 bzip2/bunzip2

bzip2 命令用来将文件压缩成常用的.bz2 格式；bunzip2 命令则用来解压.bz2 文件，bunzip2 实际上是 bzip2 的符号连接。

bzip2 命令用法：bzip [命令] 文件
命令：-d 解压缩；-f 输出文件与现有文件同名时强制覆盖；-v 显示详细信息；
bunzip2 命令用法：bunzip2 files.bz2　等于 bzip2 –d files.bz2

例子：

# bzip2 linux.tar	创建 linux.tar.bz2 文件
# bzip2 -vf linux.tar	创建 linux.tar.bz2 文件，显示详细信息
# bzip2 –d linux.tar.bz2	解压 linux.tar.bz2 文件，生成 linux.tar 文件
# bzip2 –df linux.tar.bz2	解压 linux.tar.bz2 文件，生成 linux.tar 文件
# bunzip2 linux.tar.bz2	解压 linux.tar.bz2 文件，生成 linux.tar 文件

二.1.3.4 zip/unzip

zip 命令可以用来将文件压缩成常用的.zip 格式；unzip 命令则用来解压缩 zip 文件。

zip 命令用法：zip [-命令] file.zip filelist　　命令：-r 递归处理
unzip 命令用法：unzip [-命令] file.zip　　命令：-l 显示列表；-d 指定解压目录。

例子：

# zip –r file.zip files	创建 file.zip 压缩文件 files 是目录或文件列表
# unzip –l file.zip	查看 file.zip 压缩文件内的文件列表
# unzip file.zip	解压 file.zip 到当前目录
# unzip file.zip –d /tmp	解压 file.zip 到/tmp 目录

二.1.3.5 rar

rar 命令用来创建或解压缩.rar 压缩文件。
rar 软件不是开源软件，从 rar 官方网站（http://www.rarlab.com/download.htm）下载 RAR 3.80 for Linux 或更高试用版本。

# tar zxvf rarlinux-3.8.0.tar.gz	解压 rar 文件
# cd rar	
# cp rar /bin	拷贝到/bin 或其它目录

rar 命令用法：rar <命令> <压缩文件> <文件列表...> <解压路径\>
命令：a 添加文件到压缩文件；e 解压文件到当前目录；x 解压文件到绝对路径；v 列出压缩文件列表。

例子：

# rar a file.rar file1 file2 file3 file4	创建 file.rar 压缩文件 file1,file2,file3,file4 是文件列表
# rar a file.rar file5	添加文件 file5 到 file.rar 压缩文件
# rar a dir.rar dir	创建 dir.rar 压缩文件 dir 是目录
# rar v file.rar	查看压缩文件列表
# rar e file.rar	解压 file.rar 文件到当前目录
# rar e file.rar /tmp	解压 file.rar 文件到/tmp 目录
# rar x file.rar	解压 file.rar 文件到当前目录
# rar x file.rar /tmp	解压 file.rar 文件到/tmp 目录

二.1.3.6 cpio

cpio 命令可以从 cpio 或 tar 格式的归档包中存入和读取文件（一行一行），归档包是一种包含其他文件和有关信息的文件，有关信息包括：文件名、属主、时间、访问权限等。

cpio 有三种操作模式：

（1）copy-out 模式，cpio 把文件复制到归档包中，从标准输入获得文件名列表，把归档包写到标准输出。生成文件名列表的典型方法是使用 find 命令，在 find 后最好用上-depth 选项减少因进入没有访问权限的目录而引起的麻烦；

（2）copy-in 模式，cpio 从归档包里读取文件或列出归档包里的内容；

（3）copy-pass 模式，cpio 把文件从一个目录树复制到另一个目录树，结合了 copy-in 和 copy-out 的操作，但不使用归档包。

cpio 命令格式：

cpio {-o|--create} [-0acvABLV] < name-list [> archive]	创建 cpio 归档文件
cpio {-i|--extract} [-bcdfmnrtsuvBSV] [< archive]	解压 cpio 归档文件
cpio {-p|--pass-through} [-0adlmuvLV] < name-list	拷贝 cpio 归档文件

Option：-o 创建归档文件；-v 显示操作过程；-i 加压归档文件；-t 检查归档文件内容；-d 自动建立目录；-u 强制覆盖已存在的内容；-m 保留时间属性。

例子：

$ man cpio	查看帮助信息
$ find /home/hongdy/linux -print | cpio -ov > linux.cpio	创建归档文件
$ cpio -ivt < linux.cpio	检查归档文件内容
$ cpio -ivdum < linux.cpio	解压归档文件

使用 cpio 工具创建的文件名一般以.cpio 为后缀名，当然也不一定非要以.cpio 为后缀名。

二.1.4 网络相关命令

命令	含义	格式
ping	查看网络上主机是否在工作	ping [选项] 主机名/ip 地址
ifconfig	查看和配置网络接口地址参数	ifconfig [选项] [网络接口]
route	查看和配置路由信息	route
arp	地址解析	arp
netstat	显示网络链接、路由表等信息	netstat [选项]
nslookup	查询 ip 地址和其对应的域名	nslookup [ip / 域名]
finger	查询用户信息	finger [选项] [用户] [用户@主机]
traceroute	显示数据包到主机间路径	traceroute
ssh	利用 ssh 协议登录对方主机	ssh [选项] [ip 地址]
telnet	利用 telnet 协议登录到对方主机	telnet [选项] [ip 地址]

hostname	查看本机机器名	hostname
ftp	利用 ftp 协议上传或下载文件	ftp [ip 地址]
scp	linux 下远程拷贝文件	scp user@host:/path/fromfile file scp file user@host:/path/tofile

例 **4**：网络相关命令的例子

```
网络配置
[hongdy@localhost ~]$ ifconfig                    查看当前网络接口配置
[hongdy@localhost ~]$ ifconfig eth0 192.168.1.21 更改当前 ip 地址
[hongdy@localhost ~]$ ifconfig eth0 192.168.1.21 netmask 255.255.255.0
更改当前 ip 地址和子网掩码
[hongdy@localhost ~]$ ifconfig eth0 down          关闭网络
[hongdy@localhost ~]$ ifconfig eth0 up            启动网络

[hongdy@localhost ~]$route                        显示 ip 路由表全部内容
[hongdy@localhost ~]$ route add default gw 192.168.0.1
增加一个默认网关地址的默认路由
[hongdy@localhost ~]$ route del 192.168.0.0/16 删除一个路由

ftp 工具使用
[hongdy@localhost ~]$ ftp 192.168.1.2
Connected to 192.168.1.2.
220 Serv-U FTP Server v6.1 for WinSock ready...
500 'AUTH': command not understood.
500 'AUTH': command not understood.
KERBEROS_V4 rejected as an authentication type
Name (192.168.1.2:hongdy): hongdy                 输入用户名
331 User name okay, need password.
Password:                                         输入密码
230 User logged in, proceed.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd ftp                                       切换目录
ftp> get test                                     下载 test 文件
ftp> put test2                                    上传 test2 文件
ftp> quit                                         退出

[hongdy@localhost ~]$ lftp 192.168.1.2            或输入 lftp hongdy:hongdy@192.168.1.2
lftp 192.168.1.2:~> user hongdy hongdy            输入用户名和密码
lftp hongdy@192.168.1.2:~> cd xx                  切换目录
lftp hongdy@192.168.1.2:~> get test               下载 test 文件
lftp hongdy@192.168.1.2:~> put test2              上传 test2 文件
```

lftp hongdy@192.168.1.2:~> mirror dir1	下载 dir1 目录
lftp hongdy@192.168.1.2:~> mirror –R dir2	上载 dir2 目录
lftp hongdy@192.168.1.2:~>quit	退出

scp 拷贝文件

[hongdy@localhost ~]$ scp test hongdy@192.168.1.2:/home/hongdy/ 将本地 test 文件拷贝到远程主机的 home/hongdy 目录下

[hongdy@localhost ~]$ scp hongdy@192.168.1.2:/home/hongdy/test2 . 将远程主机上的 test2 文件拷贝到本地当前目录下

查看修改机器名

# hostname	查看机器名，内容/proc/sys/kernel/hostname
# cat /proc/sys/kernel/hostname	查看机器名
# hostname hongdy.org	更改机器名
# vi /etc/sysconfig/network	修改网络配置相关机器名，修改下面一行
HOSTNAME=hongdy.org	
# vi /etc/hosts	修改机器名文件
127.0.0.1 localhost.localdomain localhost	
192.168.1.21 hongdy.org hongdy	

修改后系统有两个机器名：localhost 和 hongdy，使用 ping 命令可查看是否通

二.1.5 环境变量相关命令

命令	含义	格式
export	设置或显示环境变量	export [-fnp][变量]=[变量值]
echo	显示环境变量	echo $变量
set	设置 shell，显示环境变量	set [+-abCdefhHklmnpPtuvx]
unset	删除变量或函数	unset 变量
env	显示所有环境变量	env

例 **5**：环境变量设置的例子

[root@localhost ~]# export ABC=abc	export 导出 ABC 环境变量
[root@localhost ~]# echo $ABC	echo 查看导出的 ABC 环境变量
abc	
[root@localhost ~]# set | grep abc	set 查看导出的 ABC 环境变量
ABC=abc	
[root@localhost ~]# env | grep abc	env 查看导出的 ABC 环境变量
ABC=abc	
[root@localhost ~]# unset ABC	删除导出的 ABC 环境变量
[root@localhost ~]# export -n ABC	删除导出的 ABC 环境变量

[root@localhost ~]# set | grep abc	看不到导出的 ABC 环境变量

二.2 Shell 应用

为了防止未授权用户访问你的文件，可以在文件和目录上设置权限位。文件的属主可以设定谁具有读、写、执行该文件的权限。按照所针对的用户，文件的权限可分为三类：（1）文件属主，创建该文件的用户；（2）同组用户，拥有该文件的用户组中的任何用户；（3）其他用户，即不属于拥有该文件的用户组的某一用户。

当创建一个文件的时候，系统保存了该文件的全部信息，包括：文件的位置、文件类型文件长度、文件的权限位、文件的修改时间等。

Linux 将目录、设备等都当作文件来处理。Linux 常见的文件类型有：普通文件、目录、块设备文件、字符设备文件、符号链接文件、套接字文件、命令管道文件等。

在 linux Shell 中输入 ls –al 可以查看文件权限位：

```
$ ls –al /etc/passwd
total 4232
-rwxr-xr-x 1 root root 3576 Oct 13 04:44 /etc/passwd
```

文件的权限位用 10 个字符表示：第 1 位表示文件类型：d 目录； b 块设备；c 字符设备；l 符号链接；s 套接字；p 管道；第 234 位表示文件属主具有的文件权限：r 读；w 写；x 执行；第 567 位表示同组用户具有的文件权限；第 8910 位表示其他用户具有的文件权限。

目录的读权限位意味着可以列出其中的内容。写权限位意味着可以在该目录中创建文件，如果不希望其他用户在你的目录中创建文件，可以取消相应的写权限位。执行权限位则意味着搜索和访问该目录。如果把同组用户或其他用户针对某一目录的权限设置为- - x，那么他们将无法列出该目录中的文件。

二.2.1 chmod 改变权限位

改变权限位有两种模式：符号模式和绝对模式。

（1）符号模式：chmod [who]operator [permission] filename

who	u 文件属主；g 同组用户；o 其他用户；a 所有用户
operator	+增加权限；-取消权限；=设定权
permission	r 读权限；w 写权限；x 执行权限

例子：

$ chmod u+x file	给文件属主加上可执行权限
$ chmod a+w file	给所有用户加上可写权限
$ chmod o-x file	取消其它用户的可执行权限

（2）绝对模式：chmod [mode] file；
mode: 读权限(4)，写权限(2)，执行权限(1) 用数字表示文件的权限位

例子：

$ chmod 0764 file；	文件权限 rwx rw r
$ chmod 777 file;	给所有用户加上可读、可写、可执行权限

二.2.2 chown 改变文件属主

当用户创建一个文件时，用户就是该文件的属主。一旦你拥有某个文件，就可以改变它的所有权，把它的所有权交给另外一个 /etc/passwd 文件中存在的合法用户。

chown 命令的一般形式为： chown -R -h owner file

-R 选项意味着对所有子目录下的文件也都进行同样的操作。-h 选项意味着在改变符号链接文件的属主时不影响该链接所指向的目标文件。

例子：

```
$ chown –R embedded:embedded /home/embedded/shell
```

将/home/embedded/shell 及其目录下所有的文件的文件属主改成 embedded 用户；组属性改成 embedded 组。

二.2.3 chgrp 改变文件组属性

chgrp 命令的一般形式为：chgrp -R -h owner file

例子：

```
$ chgrp -R -h root /home/embedded/shell
```

将/home/embedded/shell 及其目录下所有的文件的文件属主改成 embedded 用户；组属性改成 embedded 组。

二.2.4 umask 创建文件目录缺省模式

登陆到 Linux 系统时，umask 命令确定了你创建文件的缺省模式，和 chmod 命令正好相反。一般在/etc/profile 中设置。

打印符号形式：umask -s
常用缺省模式值：umask 002/022
002 对应的文件和目录创建缺省权限分别为 664 和 775。
022 对应的文件和目录创建缺省权限分别为 644 和 755。

二.2.5 find 查找文件

find 命令用来查找文件

Find 命令的一般形式为：find path -options [-print -exec -ok]

Find 命令的参数

参数	含义
path	指定查找的路径
-options	-name 指定查找的内容是文件; -type 指定查找的文件的类型：d 目录、l 链接文件、b 块设备文件、c 字符设备文件、p 管道、f 是普通文件; -mtime 文件的修改时间; -ctime 文件的创建时间; -atime 最后打开文件时间; -user 查找指定用户帐号的文件 -group 查找指定组帐号的文件
-print -exec -ok	-print 把查找到的内容输出到指定的地方; -exec 执行领出的 command 命令，不给用户显示提示或操作信息; -ok 执行领出的 command 命令，提示询问操作时给用户显示信息，直到用户做了选择时才继续执行。

例 **6**：使用 **find** 查找文件的例子

（1）找出/etc 目录下是 passw 开头的所有文件
$ find /etc -name "passw*"
（2）找出/var/log 目录下所有的前 5 天的.log 文件
$find /var/log -name "*.log" -mtime +5 "+5"是指 5 天前，"-5"则是 5 天内
（3）找出/home 目录下是"hongdy"这个用户的文件
$ find /home -user "hongdy"
（4）找出/home 目录下是"hongdy"这个用户的所有的普通档的文件
$ find /home -user "hongdy" -type f
（5）找出/var/log 目录下的所有的.log 文件并查看它的详细信息
$ find /var/log -name "*.log" -type f -exec ls -l {} \;
其中的-exec 或者-ok 的用法都要在它所执行的 command 后面接" {} \;"

在使用命令行时，有很多时间都用来查找你所需要的文件。Shell 提供了一套完整的字符串模式匹配规则，可以按照所要求的模式来匹配文件。

符号	含义
*	使用*可以匹配文件名中的任何字符串
?	使用？可以匹配文件名中的任何单个字符
[...]和[!...]	使用[. . .]可以用来匹配方括号[ ]中的任何字符，还可以使用一个横杠-来连接两个字母或数字，以此来表示一个范围。

二.2.6 grep 搜索

全局正则表达式版本（grep）允许对文本文件进行模式查找，可以和正则表达式一起使用。

grep 命令的一般形式为：grep [选项] 基本正则表达式 [文件]

在 grep 命令中输入字符串参数时，最好将其用双引号括起来。有两个原因：一是以防被误解为 shell 命令；二是可以用来查找多个单词组成的字符串。

grep 常用选项有：

选项	含义
-R or r	递归进入子目录查询
-c	只输出匹配行的计数
-i	不区分大小写
-H	查询多文件时显示文件名
-h	查询多文件时不显示文件名
-l	查询多文件时只输出包含匹配字符的文件名
-n	显示匹配行及行号
-s	不显示不存在或无匹配文本的错误信息
-v	显示不包含匹配文本的所有行

缺省情况下，grep 是大小写敏感的，如要查询大小写不敏感字符串，必须使用-i 开关。

例子：

$ grep –RHn “passwd” * 在当前目录下搜索包含 passwd 字符串的文件，并显示文件名和行号

二.3 Shell 脚本

因为 shell 程序是解释执行的，所以不需要编译装配成目标程序。
按照 shell 编程的惯例，以 bash 为例，程序的第一行一般为“#!/bin/sh”，其中#表示该行是注释，叹号“!”告诉 shell 运行叹号之后的命令并用文件的其余部分作为输入，也就是运行/bin/bash 并让/bin/bash 去执行 shell 程序的内容。
执行 shell 程序的方法有三种：
（1）sh shell 程序文件名
（2）将 hello 这个档案的权限设定为可执行 chmod +x hello; ./hello
（3）使用 bash 内建指令"source"或"."

例 **7**：**Shell** 应用的例子

$ vi hellowrold.sh　　编辑 Shell 脚本 #!/bin/sh

```
echo “helloworld”

执行这个 shell 脚本有三种方法
（1）sh 程序名
$ sh helloworld.sh
helloworld

（3）./程序名
$ chmod +x hellowrold.sh
$ ./hellowrold.sh

（3）source 程序名
$ source hellworld.sh
$ . helloworld.sh
```

本章小结

本章首先介绍了 Shell 命令，包括常用的基本命令、系统管理命令、文件压缩命令、网络管理命令等。然后介绍了 shell 的一些基本应用，比如改变文件的权限位、文件属主、组属性，查找文件、搜索等。最后介绍了 shell 脚本的简单实现。

习题

2.1 如何设置 Linux 直接启动文本方式模式?
2.2 如何设置一个目录不允许访问?
2.3 如何设置防火墙? 开启 ftp 服务?
2.4 写一个 shell 脚本，实现复制文件的功能。

第三章 **LINUX** 编辑器及编译工具

三**.1 Linux vi** 编辑器

Vi 全屏幕编辑器：Vi（Visual）是以视觉为导向的全屏幕编辑器，是 Linux 常用的文本编辑器。在 Linux Shell 中输入 vi 即可打开 vi 编辑器。

三**.1.1 vi** 模式

Vi 编辑器共分为三种模式（mode)：

（1）Command：任何输入作为编辑命令不会出现在屏幕上，任何输入都引起立即反应。

（2）Insert：任何输入的数据会显示在屏幕上。在 Command 模式下输入 i 可进入 Insert 模式；Insert 方式下按 Esc 可进入 Command 模式。

（3）Escape：以 : 或 / 为前导的指令，出现在屏幕的最下一行，任何输入都被当作特别指令。在 Command 模式下输入 : 或 / 可进入 Escape 模式；Escape 模式下按 Esc 可进入 Command 模式。

三**.1.2 vi** 使用

进入 **vi**（在 Linux Shell 中输入以下指令）

vi filename	进入 vi 并读入指定名称的文件，如果文件不存在则创建新文件；
vi +filename	进入 vi 并且由文件的最后一行开始。

vi 打开文件后立即进入 Command 模式。

存储及退出 **vi**

Escape 模式下	
:w	存入文件
:w!	如果文件具有只读属性，强制存入文件
:w filename	存入指定文件，类似于另存为文件
:wq	存入文件并退出 vi
:q	不作任何修改并退出 vi
:q!	不作任何修改并强制退出 vi
:!command	暂时退出 vi 并执行 shell 指令，执行完毕后再回到 vi

光标移动

Command 模式下	
0	移到一行的开始 或者 按 Home 按钮
$	移到一行的最后 或者 按 End 按钮
[	移到文件开始位置

]	移到文件结束位置
Ctrl + u	屏幕上卷半个菜单 或者 按 Page Down 按钮
Ctrl + d	屏幕下卷半个菜单 或者 按 Page Up 按钮

三.1.3 vi 指令

删除指令

Command 模式下	
x	删除当前光标下的字符;
d0	删除本行的开始到光标位置的字符;
d$	删除光标位置起始到行尾的字符;
dw	删除光标之后的单词剩余部分;
dd	删除光标位置所在的整行（常用）
:start,endd	删除文件的第 start 到 end 行

修改指令

Command 模式下	
r	修改光标文件的字符，然后输入你想写入的字符
R	从光标位置开始修改，结束时按 ESC 键

寻找指令

Escape 模式下	
/text	从光标位置往下找字串 text
n	继续找下一个字串（在输入上面的寻找指令之后使用）

替换指令

Escape 模式下	
:%s/<old>/<new>/g	查找所有的 old 字符串并用 new 字符串代替

复制及移动

Command 模式下

（1）使用 v 进入可视模式，选定文本块，移动光标键选定内容;

（2）复制选定块到缓冲区，用 y;

（3）剪切选定块到缓冲区，用 d;

（4）粘贴缓冲区中的内容，用 p。

其他命令

Command 模式下	
.	重复前一指令
u	取消前一指令
Escape 模式下	

:set number	显示文件的行号
:set nonumber鐗	解除行号显示
:f 或 ctrl + g	告诉用户有关现行编辑文件的数据。
:set ts=4	设置 tab 间距

三.1.4 vi 配置文件

每次 vi 运行时都会读取 vimrc 配置文件，对于所有用户都适用的是/etcvimrc 文件，对于单个用户适用的是用户工作目录下~/.vimrc 文件。

$ vi ~/.vimrc	
set ts=4	设置 Tab 间距为 4 个空格
set nu	默认显示行号

这样 vi 运行时默认 Tab 间距是 4 个空格，默认显示行号等。当然，更多的默认配置也可以写入 vi 配置文件。

三.2 GCC 编译器

GNU Tools 是 Linux 环境下主要的开发工具集，作为 Linux 程序开发者，一般需要了解和掌握如下几类工具：

（1）编译开发工具（即能够把一个源程序编译生成为一个可执行程序的软件）GCC；
（2）调试工具（即能够对执行程序进行源码或汇编级调试的软件）GDB；
（3）软件工程工具（用于协助多人开发或大型软件项目的管理的软件）make；
（4）文本差异处理工具（用于比较两个文件之间的差异的工具软件）diff、patch；
（5）二进制工具（即能够对二进制文件进行处理的工具软件）binutils。

GCC 是代表 GNU C Compiler，但是现在它代表 GNU Compiler Collection。GCC 是用于 C、C++、Java 等编程语言的一个编译器集。GCC 主要包括的工具：（1）cpp--GNU 预处理器；（2）gcc--符合 ISO 标准的 C 编译器；（3）g++--符合 ISO 标准的 C++编译器。

三.2.1 GCC 编译过程

编译器 GCC 在编译时首先调用 cpp 进行预处理，在预处理过程中对源代码文件中的文件包含(include)、预编译语句（宏定义 define 等）进行分析；接着调用 gcc 进行编译，这个阶段根据输入文件生成以.o 为后缀的目标文件；最后编译器调用 ld 链接器将引导代码、库代码和目标代码链接成可执行文件。

编译器 GCC 的编译过程见下图所示：

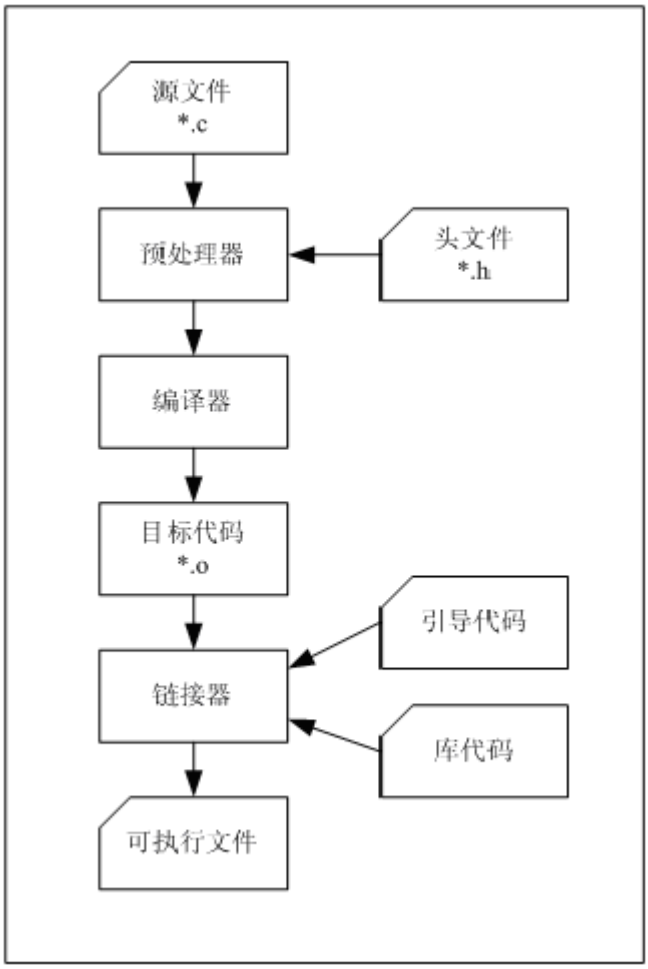

图 **2**：**GCC** 编译过程

三.2.2 GCC 编译选项

（**1**）输入选项

编译器必须知道要处理的是哪种类型的输入数据。例如 C 源程序文件的处理就与 C++文件不同。对于编译器来说，有一个基本的隐式选项：源程序文件的扩展名。该选项将决定调用哪一个 GCC 编译器。例如：file.c 调用的是 C 编译器，而 foo.C 或 foo.cxx 则会调用 C++编译器。在 GCC 手册中列出了可接收源代码文件的扩展名的完整列表。

文件扩展名	文件类型
.c	C 语言代码
.C, .cc, .cpp	C++语言代码
.i	预处理后的 C 语言代码
.s, .S	汇编语言代码
.o	目标代码
.a	静态链接库
.so	动态链接库

（**2**）输出选项

编译器还必须知道用户期望获得哪种类型的输出。GCC 驱动器可以为整个工具链的其他部分产生一些指令，从而生成最终的可执行程序，或者在生成一些中间文件时就停止。您可以使用源程序的文件扩展名或命令行选项来控制编译器的输出。

GCC 常用编译选项

选项	含义
-c	编译但不链接，将输入源文件编译成目标文件，输出.o 文件
-S	汇编但不编译，将 C/C++文件生成汇编文件，输出.s 文件
-o	将输出内容存于指定的输出文件
-E	预处理但不编译，对.c 文件进行预处理
-g	编译文件中生成调试信息
-ansi	支持所有 ANSI 程序
-w	关闭所有警告
-Wall	打开所有警告
-static	静态链接，程序可不倚赖任何库就能运行
-shared	动态链接，生成支持动态共享库的可执行文件
-fPIC	产生与位置无关的代码
-x	从指定的步骤开始编译
-O	O0 不优化；O1 一级优化；O2 二级优化；O3 三级优化
-Dmacro	定义宏 macro -D macro=defn，定义宏 macro 为 denf
-I dir	设置头文件搜索路径
-L dir	设置库文件搜索路径
-Wimplicit	如果有隐含申明，显示警告信息
-Wno-implict	不显示对隐含信息的警告
-v	打印出编译过程中执行的命令

三.2.3 GCC 编译使用

使用 gcc 编译器可以将一个源文件编译成可执行文件，先看一个具体的例子：

例 **8**：使用编译器 **GCC** 的例子

```
/* hello.c */

#include <stdio.h>
int main(int argc, char *argv[])
{
   printf(“hello, world!\n”);
   return(0);
}
```

编译、运行：

```
$ gcc hello.c –o hello
$ ./hello
hello, world!
```

第一行命令 gcc 对源程序 hello.c 进行编译和链接，参数-o 指定创建名为 hello 的可执行文件；
第二行命令是执行这个程序，可执行文件前加上./ 表示执行当前目录下的 hello 程序；
第三行是运行结果。

GCC 编译器首先运行预处理程序 cpp 展开 hello.c 中的宏并在其中插入#include 文件所包含的内容；然后把预处理后的源代码编译成目标代码；最后链接程序创建一个名为 hello 的二进制文件。

在编译过程中也可以通过手工来逐步执行编译过程：

```
$ gcc –E hello.c –o hello.cpp
$ gcc –x cpp-output –c hello.cpp –o hello.o
$ gcc hello.o –o hello
$ ./hello
```

第一行运行预处理程序，选项-E 告诉 GCC 编译器在预处理后停止编译；编译后生成 hello.cpp 文件；
第二行将 hello.cpp 编译成目标代码，-x 选项告诉 GCC 编译器从指定的步骤开始编译，本例从预处理程序处理后的 cpp-output 开始，-c 选项告诉 GCC 编译器编译但不链接；
第三行将目标代码 hello.o 链接生成 hello 二进制代码。

编译器的选项 –o FILE 告诉 GCC 编译器把输出定向到 FILE 文件。如果不指定-o 选项，对于源文件 file.c，其生成的可执行程序名为 a.out，目标文件是 file.o，汇编文件是 file.s。

$ gcc hello.c	生成 a.out 可执行文件
$ gcc hello.c -c	生成 hello.o 的目标文件
$ gcc hello.c -S	生成 hello.s 的汇编文件

其他 GCC 编译器的编译选项将在下面的例子中具体介绍。

三.3 GDB 调试器

三.3.1 GDB 概述

GDB 调试器主要用来调试 C/C++程序。要调试 C/C++程序，在使用编译器 GCC 编译程序时必须加上-g 选项，表示把调试信息加到可执行文件中。

启动 GDB 的方法有以下几种：
（1）gdb <program>　　用 GDB 调试 program 可执行文件；

（2）gdb <program> core　　用 GDB 同时调试 program 可执行文件和 core 文件，core 是程序非法执行后 core dump 后产生的文件；
（3）gdb <program> <PID>　　如果调试的程序是一个服务程序，那么可以指定这个服务程序运行时的进程 PID，GDB 会自动 attach 上去并调试，不需要先停止原来的程序。

三.3.2 GDB 常用命令

命令	含义
file	载入命令，先运行 gdb，然后载入要调试的文件 file gdbtest
break 或 b	设置断点 break function 或 break linenum 或 break filename:linenum;
run 或 r	运行载入的可执行程序
next 或 n	执行下一行源代码
step 或 s	单步执行并进入函数
continue 或 c	继续执行正在调试的程序
quit 或 q	退出 gdb 调试器
info	查看程序信息 source; stack; local; br; prog; var; function
list 或 l	显示源代码段 function; linenum; list; list-; inelname:function;
finish	跳出执行函数
print 或 p	打印数据，print exp; /F 选择输入打印格式（x d o t a c f）
display	每次程序停止后显示表达式的值 display exp
set	修改变量值，set variable=value
x	检查内存
clear	删除设置的的断点
bt	显示所有的调用堆栈，可以用来显示函数的调用顺序
kill	终止正被调试的程序
help	显示指定命令的帮助信息　help name
shell <command>	运行 shell 命令
directory <dirname>或 dir <dirname >	设置或清除源文件搜索路径信息
show directories	显示源文件搜索路径
set args arg1 arg2 ...	设置命令行参数

三.3.3 GDB 调试

调试程序中，暂停程序运行是必须的，GDB 可以方便地暂停程序的运行。可以设置程序的在哪行停住、在什么条件下停住、在收到什么信号时停往，以便于查看运行时的变量以及运行时的流程。当进程被 GDB 停住时，可以使用 info program 来查看程序的是否在运行、进程号、被暂停原因等。

在 GDB 调试中的暂停方式：断点（Break Point）、观察点（Watch Point）、捕捉点

（Catch Point）、信号（Signals）、线程停止（Thread Stops）。如果要恢复程序运行，可以使用 c 或是 continue 命令。

（1）设置断点（Break Point）

用 break 命令来设置断点，有下面几种设置断点的方法：

break <function>	进入指定函数时停住，C++中可使用 class::function 来指定函数名
break <linenum>	在指定的行号停住
break filename:linenum	在源文件 filename 的 linenum 行处停住
break filename:function	在源文件 filename 的 function 函数的入口处停住
break *address	在程序运行的内存地址处停住
break <xx> if <condition><xx>可以是上述的参数，condition 表示条件，在条件成立时停住。比如在循环体中，可以设置 break if i=100，表示当 i 为 100 时停住程序	

查看断点可用 info 命令：info b

（2）设置观察点（Watch Point）

观察点一般用来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化马上停住程序。有下面几种设置观察点的方法：

watch <expr>	为表达式（变量）expr 设置一个观察点，一量值有变化马上停住
rwatch <expr>	当表达式（变量）expr 被读时停住程序
awatch <expr>	当表达式（变量）的值被读或被写时停住程序

查看所有观察点可用 info 命令：info watchpoints

（3）设置捕捉点（Catch Point）

可以设置捕捉点来捕捉程序运行时的一些事件。如：载入共享库（动态链接库）或 C++中的异常。设置捕捉点的格式为：

catch <event>	当 event 事件发生时停住程序
tcatch <event>	只设置一次捕捉点，当程序停住以后，捕捉点被自动删除

（4）维护停止点

上面说了如何设置程序的停止点，GDB 中的停止点也就是上述的三类。在 GDB 中如果觉得已定义好的停止点没有用了，可以使用 delete、clear、disable、enable 这些命令来维护。比删除停止点更好的一种方法是 disable 停止点，disable 了的停止点，GDB 不会删除，当需要时 enable 即可。

clear	清除所有的已定义的停止点
clear <function>	
clear <filename:function>	清除所有设置在函数上的停止点
clear <linenum>	
clear <filename:linenum>	清除所有设置在指定行上的停止点
disable <linenum>	禁止设置在指定行上的停止点

enable <linenum>	使能设置在指定行上的停止点

GDB 有能力在调试程序的时候处理任何一种信号，可以告诉 GDB 需要处理哪一种信号。可以要求 GDB 收到你所指定的信号时，马上停住正在运行的程序进行调试。可以用 GDB 的 handle 命令来完成这一功能。

handle <signal> <keywords...>

在 GDB 中定义一个信号处理。信号<signal>可以以 SIG 开头或不以 SIG 开头，可以用定义一个要处理信号的范围，也可以使用关键字 all 来标明要处理所有的信号。一旦被调试的程序接收到信号，运行程序马上会被 GDB 停住以供调试。其<keywords>可以是以下几种关键字的一个或多个。

关键字	含义
nostop	当被调试的程序收到信号时 GDB 不停住程序的运行
sstop	当被调试的程序收到信号时 GDB 会停住程序的运行
print	当被调试的程序收到信号时 GDB 会显示出一条信息
noprint	当被调试的程序收到信号时 GDB 不会告诉你收到信号的信息
ignore	当被调试的程序收到信号时 GDB 不让被调试程序处理这个信号
noignore	当被调试的程序收到信号时 GDB 不处理信号

handle SIGPIPE nostop 当程序收到 SIGPIPE 信号时不停止程序运行

查看有哪些信号在被 GDB 检测到：info signals 和 info handle

三.3.4 GDB 调试实例

下面根据具体的实例介绍 GDB 调试器的使用。

例 **9**：使用 **GDB** 调试程序的例子

```
/* gdb_test.c */

#include <stdio.h>

int my_sum(int n)
{
   int i, sum = 0;
   for(i=0; i<n; i++)
   {
      sum += i;
   }
   return sum;
}
```

```
int main(int argc, char *argv[])
{
   int i;
   long result = 0;

   for(i=1; i<=100; i++)
   {
      result += i;
   }

   printf("result[1-100] = %d \n", result );

   printf("result[1-200] = %d \n", my_sum(200) );

   return(0);
}
```

编译、调试:

```
$ gcc gdb_test.c -g -o gdb_test                    编译具有调试功能的可执行程序 加上-g 选项
$ gdb gdb_test                                     使用 GDB 调试程序
GNU gdb Red Hat Linux (6.5-8.fc6rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

(gdb) l                                            使用 list 命令查看源代码，list 1 从第一行开
始
1      #include <stdio.h>
2
3      int my_sum(int n)
4      {
5         int i, sum = 0;
6         for(i=0; i<n; i++)
7         {
8            sum += i;
9         }
10         return sum;
11     }
```

```
12
13      int main(int argc, char *argv[])
14      {
15          int i;
16          long result = 0;
(gdb)                                                   回车，继续执行上一次执行的命令
17
18          for(i=1; i<=100; i++)
19          {
20              result += i;
21          }
22
23          printf("result[1-100] = %d \n", result );
24
25          printf("result[1-200] = %d \n", my_sum(200) );
(gdb) b 16                                              根据行号 设置断点
Breakpoint 1 at 0x8048392: file gdb_test.c, line 16.
(gdb) b my_sum                                          根据函数名 设置断点
Breakpoint 2 at 0x804835a: file gdb_test.c, line 5.
(gdb) info b                                            查看断点信息
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048392 in main at gdb_test.c:16
2   breakpoint     keep y   0x0804835a in my_sum at gdb_test.c:5
(gdb) r                                                 运行程序 run
Starting program: /home/hongdy/linux/trainning/chap02/gdb_test

Breakpoint 1, main () at gdb_test.c:16                  运行到断点处终止
16          long result = 0;
(gdb) n                                                 单步执行 next
18          for(i=1; i<=100; i++)
(gdb) n
20              result += i;
(gdb) p i                                               使用 print 命令打印 i
$1 = 1
(gdb) p result                                          使用 print 命令打印 result
$2 = 0
(gdb) n
18          for(i=1; i<=100; i++)
(gdb) p result
$3 = 1
(gdb) display result                                    使用 display 命令显示 result
1: result = 1
(gdb) bt                                                查看函数堆栈
```

```
#0  my_sum (n=200) at gdb_test.c:5
#1  0x080483d1 in main () at gdb_test.c:25
(gdb) n
6           for(i=0; i<n; i++)
(gdb) n
8               sum += i;
(gdb) finish                                        跳出 my_sum 函数
Run till exit from #0  my_sum (n=200) at gdb_test.c:6
0x080483d1 in main () at gdb_test.c:25
25          printf("result[1-200] = %d \n", my_sum(200) );
1: result = 5050                                    运行结果
Value returned is $9 = 19900
(gdb) c                                             继续运行直至结束
Continuing.
result[1-200] = 19900                               运行结果

Program exited normally.
(gdb)quit                                           退出 GDB 调试
```

使用 list 命令查看代码时，可以使用 list 1；表示查看第一行代码。

如果需要向程序传递参数时使用 set args 设置，例如程序运行时 ./a.out abc 123，使用 GDB 调试时输入参数 set args abc 123。

使用 GDB 调试代码时非常有用的，特别是当你执行代码发生错误时，你不知道哪一行发生了错误，使用 GDB 调试，设置断点然后单步调试，很容易就可以找到出错的地方。

三.4 GNU Make

GNU Make（简称 make）是一种代码维护工具，根据工程项目中中各个模块的更形情况，自动维护和生成目标代码。Make 的主要任务是读入一个文本文件（默认 Makefile 或 makefile），并根据这个文本文件定义的规则和步骤，完成整个项目的维护和代码生成等工作。

首先我们来看一个不使用 makefile 的例子。

例 **10**：不使用 **Makefile** 的例子

编辑 hello.h、hello.c、world.h、world.c、main.c 五个源文件。

```
/* hello.h */

```

```
void hello();
```

```
/* hello.c */

#include “hello.h”
#include <stdio.h>
void hello()
{
   printf(“hello, “);
}
```

```
/* world.h */

void world();
```

```
/* world.c */

#include “world.h”
#include <stdio.h>
void world()
{
   printf(“world!\n”);
}
```

```
/* main.c */

#include “hello.h”
#include “world.h”
#include <stdio.h>

int main(int argc, char *argv[])
{
   hello();
   world();

   return(0);
}
```

编译、运行：

```
$ gcc hello.c world.c main.c –o helloworld
$ ./helloworld
hello, world!
```

如果修改了其中的某个文件，每次编译时都要把所有的文件编译并链接成可执行文件。如果此工程的文件比较多，编译时就比较费时费力。

看一下使用 makefile 的例子。

例 **11**：使用 **Makefile** 的例子 **1**

还是用上例中的五个源文件，将这五个文件 hello.h hello.c world.h world.c main.c 拷贝到 make_1 目录下。

```
$ vi makefile                                    以下为文件内容
all:helloworld
helloworld:hello.o world.o main.o
    gcc hello.o world.o main.o -o helloworld
hello.o:hello.c
    gcc -c hello.c -o hello.o
world.o:world.c
    gcc -c world.c -o world.o
main.o:main.c
    gcc -c main.c -o main.o

.PHONY : clean
clean:
    rm -fr *.o helloworld
```

编译、运行：

```
$ make
$ ./helloworld
hello, world!
```

从 makefile 可以看出，helloworld 是 hello.o world.o main.o 三个目标文件链接而成。这三个目标文件分别右其.c 文件编译而来，gcc 使用 –c 选项只编译不链接。

从使用 makefile 的例子中可以看出，任何时刻修改了项目文件，编译时输入 make 即可完成所有的操作，比较方便。

三.4.1 Makefile 概述

makefile 基本上就是目标（target），关联（dependencies）和动作命令（command）三者所组成的一系列规则。make 就会根据 Makefile 的规则来决定如何编译（compile）和连接（link）程序。Makefile 是一个文本形式的数据库文件，其中包含一些规则告诉 make 编译那些文件，怎样编译以及在什么条件下编译。

每条规则包含的内容：一个目标体（target），make 最终需要创建的二进制文件或目标文件；一个或多个依赖体（dependency）的列表，依赖体是编译目标体需要的文件；为了从指定的依赖体创建出目标体所需执行的命令（command）的列表。

Makefile 有下列通用形式：

```
target : dependency [dependency […] ]
    command
    command
    […….]
```

注意：每一个命令的第一个字符必须是 Tab 制表符（4 个空格无效）

target 是需要创建的二进制文件或目标文件；dependency 是创建 target 时需要输入的一个或多个文件的列表；command 是创建 target 需要的步骤。

GNU make 的一般用法：make [选项] [变量定义] [目标]

选项	说明
-f filename	使用指定的文件作为 makefile
-k	执行命令出错时放弃当前目标，继续维护其他目标
-r	忽略内部规则
-s	执行但不显示命令，常用来检查 makefile 正确性
-S	执行命令出错就退出
-t	修改每个目标文件的创建日期

三.4.2 Makefile 变量

Make 允许在 makefile 中创建和使用变量，makefile 中的变量就像 Linux 中的环境变量，可以在 makefile 的任何地方被引用。变量可以用来保存文件名列表；保存可执行文件名；保存参数等。

定义变量：VARNAME = some_text

使用变量：$(VARNAME)

Make 使用两种变量：递归展开变量和简单展开变量。

（1）递归展开变量在引用时逐层展开，在展开式中如包含对其他变量的引用，这些变量也被展开。例如：TOPDIR=/home/embedded/project　SRCDIR=$(TOPDIR)/src

（2）简单展开变量在定义处展开并且只展开一次，从而消除变量的嵌套调用。例如：CC :=gcc -o CC += -o2　(CC=gcc -o -o2)

Makefile 中定义了一些自动变量和预定义变量。

Makefile 自动变量

变量	说明
$@	规则的目标所对应的文件名

$<	规则中的第一个相关文件名
$^	规则中所有相关文件的列表，以空格分割
$?	规则中日期新于目标的所有相关文件的列表，以空格分割
$(@D)	目标文件的目录部分
$(@F)	目标文件的文件名部分

Makefile 预定义变量

变量	说明
AR	归档维护程序 默认值=ar
AS	汇编程序 默认值=as
CC	C 编译程序 默认值=cc
CPP	C 预处理程序 默认值=cpp
RM	文件删除程序 默认值=rm -f
CFLAGS	传递给 C 编译器的标志，没有默认值
CPPFLAGS	传递给 C 预处理程序的标志，没有默认值
LDFLAGS	传递给链接程序(ld)的标志，没有默认值

如果需要，用户可以在 makefile 中重新定义这些与定义变量。

三.4.3 Makefile 规则

（**1**）隐式规则

除了在 Makefile 中显式指定的规则外，make 还有一整套隐式规则，或称为预定义规则。对每一个 somefile.o，make 首先找到与之相对应的 somefile.c，并用 gcc -c somefile.c -o somefile.o 生成对应的 somefile.o 文件

```
######  Inplict rules
.SUFFIXES: .cpp .cxx .cc .C .c
.cpp.o:
	$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.cxx.o:
	$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.cc.o:
	$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.C.o:
	$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.c.o:
	$(CC) -c $(CFLAGS) $(INCPATH) -o $@ $<
```

（**2**）模式规则

通过定义用户自己的隐式规则，模式规则提供了扩展 make 的隐式规则的一种方法。模式规则类似于普通规则，但目标必须含有特定符号 %, 例如： %.o : %.c 所有.o 的目标文件

都是从.c 源文件编译而来。

```
%.o : %.c
    $(CXX) –c $(CPPFLAGS) $< -o $@
```

注释：在 Makefile 中插入注释时必须在注释前加上符号#，make 读到#后忽略该符号及这行余下的字母。

函数：Makefile 中的函数调用必须指定函数名以及参数等，函数调用格式：
$(function arguments) 或者 ${function arguments}
Makefile 中的 Shell 函数很有用，它是 make 与外部环境的通信工具。

例 **12**：使用 **Makefile** 的例子 **2**

还是用上例中的五个源文件，在目录下创建 makefile 文件

```
$ vi makefile                                   以下为文件内容
SOURCES = hello.c world.c main.c
OBJECTS = hello.o world.o main.o
TARGET = helloworld

all: $(TARGET)
$(TARGET):$(OBJECTS)
    $(CC) $(OBJECTS) -o $(TARGET)

.c.o:
    $(CC) -c -o $@ $<

.PHONY  : clean
clean:
    rm -fr *.o helloworld
```

本例中使用了 Makefile 的预定义变量 $CC；使用了变量保存文件列表；使用了编译的隐式规则.c.o。

例 **13**：使用 **Makefile** 的例子 **3**

```
$ vi makefile                                   以下为文件内容
CC      =   gcc
CXX     =   g++
CFLAGS      =   -pipe -Wall -W -O2 -DNO_DEBUG
CXXFLAGS  =   -pipe -fno-rtti -Wall -W -O2 -DNO_DEBUG

INCPATH     =   -I.
LINK    =   gcc
```

```
LFLAGS        =
LIBS      =   -L.

SOURCES =   hello.c world.c main.c
OBJECTS =   hello.o world.o main.o
TARGET  =   helloworld

######  Build rules
all: $(TARGET)
$(TARGET):  $(OBJECTS)
      $(LINK) $(LFLAGS) -o $(TARGET) $(OBJECTS) $(LIBS)

######  Inplict rules
.SUFFIXES: .cpp .cxx .cc .C .c
.cpp.o:
      $(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.cxx.o:
      $(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.cc.o:
      $(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.C.o:
      $(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.c.o:
      $(CC) -c $(CFLAGS) $(INCPATH) -o $@ $<

.PHONY  :clean
clean:
      rm -f $(OBJECTS) $(TARGET)
```

本例中重新定义了预定义变量，显式地显示了隐式规则。

例 **14**：使用 **Makefile** 的例子 **4**

假设当前目录为 make_4, 有 hello 目录，内有 hello.c 和 Makefile 文件；welcome 目录内有 welcome.c 和 Makefile 文件；make_4 目录下有 Rules.mak 和 Makefile 两个文件。

hello 目录下 hello.c 文件

```
/* hello.c */

#include <stdio.h>

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

```
    printf("hello!\n");
    return(0);
}
```

hello 目录下 Makefile 文件

```
TOPDIR  =  ../
include $(TOPDIR)/Rules.mak

EXEC    =  hello
OBJS    =  hello.o

all:$(EXEC)
$(EXEC):$(OBJS)
        $(LINK) $(LFLAGS) -o $@ $< $(LIBS)

install:
        $(INSTALL) $(EXEC) $(INSTALL_DIR)

clean:
        rm -f *.o $(EXEC)
```

hello 目录下的 Makefile 包含上一目录的规则文件，生成 hello 可执行文件，并可安装到$(INSTALL_DIR) 目录下。

welcome 目录下 welcome.c 文件

```
/* welcome.c */

#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("welcome!\n");
    return(0);
}
```

Welcome 目录下 Makefile 文件

```
TOPDIR  =  ../
include $(TOPDIR)/Rules.mak

EXEC   =  welcome
OBJS   =  welcome.o

all:$(EXEC)
```

```
$(EXEC):$(OBJS)
	$(LINK) $(LFLAGS) -o $@ $< $(LIBS)

install:
	$(INSTALL) $(EXEC) $(INSTALL_DIR)

clean:
	rm -f *.o $(EXEC)
```

welcome 目录下的 Makefile 包含上一目录的规则文件，生成 welcome 可执行文件，并可安装到$(INSTALL_DIR)目录下。

make_4 目录下的 Rules.mak 规则文件

```
### Rules.mak

CXX          =  g++
CFLAGS       =  -pipe -Wall -W -O2 -g
CXXFLAGS     =  -pipe -fno-exceptions -fno-rtti -Wall -W -O2 -g
INCPATH      =  -I.
LINK         =  g++
LFLAGS       =
LIBS         =  -L.

INSTALL      =    install -m 755
INSTALL_DIR=    ../bin

######  Inplict rules
.SUFFIXES: .cpp .cxx .cc .C .c
.cpp.o:
	$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.cxx.o:
	$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.cc.o:
	$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.C.o:
	$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
.c.o:
	$(CC) -c $(CFLAGS) $(INCPATH) -o $@ $<
```

定义了规则文件，安装路径是当前目录下的的 bin 目录。

Make_4 目录下的 Makefile 文件

```
include $(PWD)/Rules.mak
```

```
DIRS    =   hello   welcome

all:
   for i in $(DIRS);   do make -C $$i || exit $?;  done

install:
   for i in $(DIRS);   do [ ! -d $$i ] || make -C $$i install; done

clean:
   for i in $(DIRS);   do [ ! -d $$i ] || make -C $$i clean;   done
```

make_4 目录下的 makefile 逐步进入 hello 和 welcome 目录并调用执行其内的 makefile，可执行文件可安装到定义的安装目录下。

编译、执行：

```
$ cd make_4
$ mkdir bin
$ make
$ make install
$ ls
bin  hello  Makefile  Rules.mak  welcome
$ ls bin
hello  welcome
```

先进入工作目录并建立 bin 目录作为安装目录，make 后在 hello 和 welcome 目录内分别生成可执行文件，make install 将生成的可执行文件安装到 bin 目录下，此时可执行 bin 目录下的两个可执行文件。

三.4.4 Makefile 自动生成

可以使用 automake 自动生成 Makefile 文件，如下图所示：

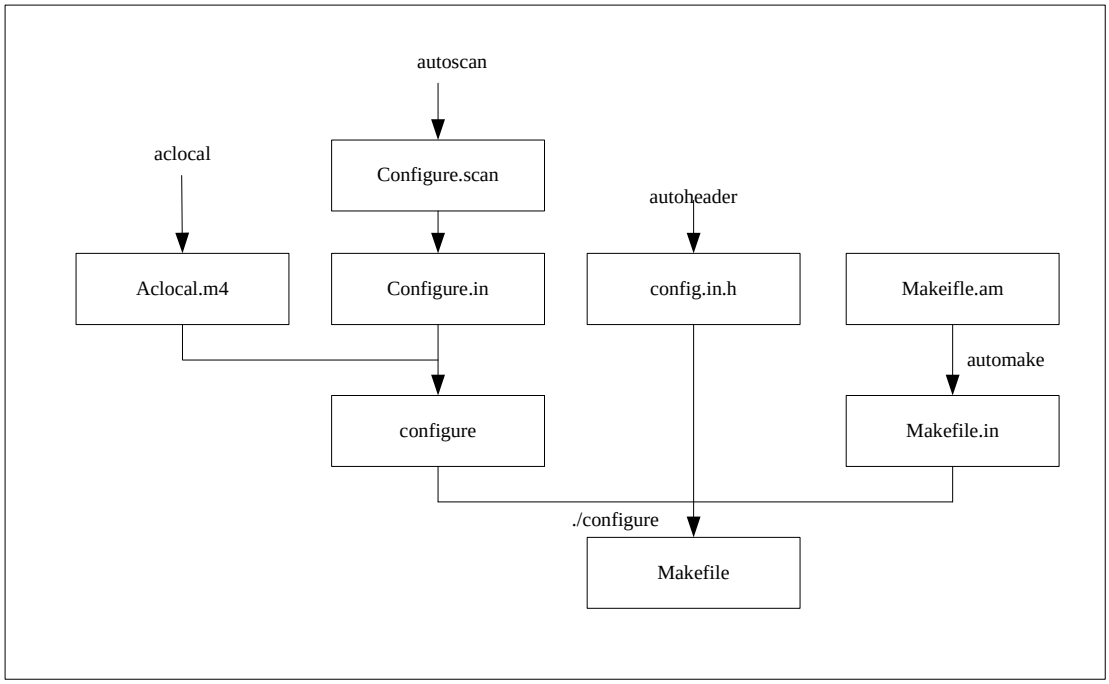

图 **3**：**automake** 生成 **Makefile** 流程图

Automake 工具可以自动生成 Makefile 文件。程序源代码所存放的目录结构最好符合 GNU 的标准惯例，接下来就用一个 hello.c 来作为例子。

例 **15**：使用工具自动生成 **Makefile** 的例子

在用户的工作目录建立一个 auto_make 目录，并创建一个 hello.c 文件

```
$ mkdir auto_make
$ cd auto_make
$ vi hello.c                                    以下为文件内容
#include <stdio.h>
int main(int argc, char *argv[])
{
   printf(“hello, world!\n”);
   return(0);
}
```

（1）autoscan

检查当前目录及其子目录下的文件，执行 autoscan 后会产生一个 configure.scan 的文件，这个文件是 configure.in 的原型。

```
$ autoscan。
$ ls
autoscan.log  configure.scan  hello.c
```

（2）configure.in

编辑 configure.scan 文件并改名为 configure.in 文件。

```
$ mv configure.scan configure.in
$ vi configure.in                                    以下为文件内容
AC_PREREQ(2.59)

AC_INIT(hello, 1.0)
AM_INIT_AUTOMAKE(hello, 1.0)
AC_CONFIG_SRCDIR([hello.c])
AC_CONFIG_HEADER([config.h])

AC_PROG_CC

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

（3）aclocal

configure.in 是 autoconf 的脚本配置文件，运行 aclocal 生成 aclocal.m4 文件，该文件主要处理本地的宏定义。

```
$ aclocal
$ ls
aclocal.m4  autom4te.cache  autoscan.log  configure.in  hello.c
```

（4）auconf

运行 autoconf 生成 configure 可执行文件。

```
$ autoconf
$ ls
aclocal.m4  autom4te.cache  autoscan.log  configure  configure.in  hello.c
```

（5）autoheader

运行 autoheader 生成 config.h.in 文件。

```
$ autoheader
$ ls
aclocal.m4      autoscan.log  configure     hello.c
autom4te.cache  config.h.in   configure.in
```

（6）Makefile.am

创建 Makefile.am 文件

```
$ vi Makefile.am                                     以下为文件内容
AUTOMAKE_OPTIONS = foreign
bin_PROGRAMS = hello
hello_PROGRAMS = hello.c
```

（7）automake

执行 automake --add-missing，Automake 会根据 Makefile.am 文件产生一些文件，包含最重要的 Makefile.in

```
$ automake --add-missing
```

（8）configure

运行 confiure，把 Makefile.in 变成最终的 Makefile

```
$ ./configure
```

（9）make

Make 可生成可执行文件

```
$ make
$./hello
hello,world!
```

（10）make install

把可执行文件安装到/usr/bin 目录下，需要有 root 权限。安装好后在任何目录下可直接运行 hello 可执行文件。

```
$ make install
$ hello
hello，world！
```

（11）make clean

删除可执行文件

```
$ make clean
```

本章小结

本章开始介绍了 Linux 下常用的 vi 编辑器，然后介绍了 Linux 下的开发工具，包括 GCC 编译器，GDB 调试器。介绍了 GCC 的编译过程，用实例叙述了 GDB 调试代码的步骤。最后介绍了 make 工具管理项目工程，Makefile 变量、规则、编写等，还介绍了如何使用工具自动生成 Makefile。

习题

3.1 vi 编辑器有几种模式？它们之间如何切换？
3.2 vi 编辑器如何使用复制、剪贴和拷贝功能？如何使用查找、替换功能？
3.3 简述从源代码到可执行文件的编译过程？
3.4 gdb 调试器设置断点有哪些方法？
3.5 如何在 makefile 中使用函数？

第四章 **LINUX** 文件和目录

四**.1** 标准 **IO**

标准 IO 对文件的操作是围绕流进行的，当用标准 IO 库打开或创建一个文件时，就已经使用了一个流与文件相关联。标准 IO 流：标准输入 stdin；标准输出 stdout；标准出错 stderr。定义在 stdio.h 文件里。

本节讲述如何用标准 IO 库打开文件、写入或读取文件、关闭文件等。

标准 IO 函数

四**.1.1** 打开文件

创建或打开文件使用 fopen 函数

```
#include <stdio.h>
FILE *fopen(const char *restrict pathname, const char *restrict type);
返回：成功返回文件指针；错误返回 NULL
```

参数：（1）pathname 创建或打开文件名；（2）type 文件打开类型：r、w、a、r+、w+、a+。

四**.1.2** 关闭文件

fclose 函数

```
#include <stdio.h>
int fclose(FILE *fp);
返回值：成功返回 0；错误返回 EOF
```

四**.1.3** 写入文件

格式化输出函数

```
#include <stdio.h>
int fprintf(FILE *restrict fp, const char *restrict format, ...);
返回：成功返回输出字符数；出错为负值
```

写入一个字符

```
#include <stdio.h>
int fputc(int c, FILE *fp);
返回值：成功返回 C；出错返回 EOF
```

写入字符串

```
#include <stdio.h>
int fputs(const char *restrict str, FILE *restrict fp);
返回值：成功返回非负值；出错返回 EOF
```

二进制 **IO** 输出

```
#include <stdio.h>
size_t fwrite(const void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp)
返回值：写入文件的对象数
```

可以写入二进制数组、结构体

四**.1.4** 读取文件

格式化输入函数

```
#include <stdio.h>
int fscanf(FILE *restrict fp, const char *restrict format, ...);
返回值：指定的输入项目数，出错或在任一变换前文件结束则为 EOF
```

读入字符串

```
#include <stdio.h>
char *fgets(char *restrict buf, int n，FILE *restrict fp);
返回值：成功返回 buf，已处文件尾端则为 NULL
```

二进制 **IO** 输入

```
#include <stdio.h>
size_t fread(void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp);
返回值：读取文件的对象数
```

可以输入二进制数组、结构体

例 **16**：流文件写入读取的例子

```
/* rw_file1.c */

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

```
int main(int argc, char *argv[])
{
   FILE *readfp;
   FILE *writefp;
   char name[20];
   int  age;
   char buf[50];

   if( (writefp=fopen("testfile", "w")) == NULL){          以写入方式创建 testfile 文件
      perror("Error: create file\n");
      exit(1);
   }

   fprintf(writefp, "Name: %s  Age: %d\n", "xuhong", 28); 格式化输出
   fputc('I', writefp);                                        写入一个字符
   fputs(" Love you", writefp);                                写入字符串

   fclose(writefp);                                        关闭 testfile 文件

   if( (readfp=fopen("testfile", "r")) == NULL){           以只读方式打开 testfile 文件
      perror("Error: open file\n");
      exit(1);
   }

   fscanf(readfp, "Name: %s Age: %d\n", name, &age);           格式化读取
   printf("Name: %s Age:%d\n", name, age);
   fgets(buf, 50, readfp);                                 读入一行字符串
   printf("%s\n", buf);

   fclose(readfp);

   return(0);
}
```

编译、运行：

```
$ gcc rw_file1.c -o rw_file1
$ ./rw_file1
Name: hongdy Age:28
I Love you
```

例 **17**：二进制文件的读写的例子

```
/* rw_file2.c */
```

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

struct person
{
   char name[20];
   int  age;
   char address[50];
};

int main(int argc, char *argv[])
{
   FILE *readfp;
   FILE *writefp;
   struct person p1, p2;

   strcpy(p1.name, "hongdy");
   p1.age = 28;
   strcpy(p1.address, "shanghai, china");

   if( (writefp=fopen("testfile", "wb")) == NULL){
      perror("Error: create file\n");
      exit(1);
   }
   if( (fwrite(&p1, sizeof(struct person), 1, writefp)) != 1){
      perror("Error: write file\n");
      exit(1);
   }
   fclose(writefp);

   if( (readfp=fopen("testfile", "rb")) == NULL){
      perror("Error: open file\n");
      exit(1);
   }
   if( (fread(&p2, sizeof(struct person), 1, readfp)) != 1){
      perror("Error: read file\n");
      exit(1);
   }
      printf("Name: %s\nAge:%d\nAddress:%s\n", p2.name, p2.age, p2.address);
      fclose(readfp);
```

```
    return(0);
}
```

编译、运行：

```
$ gcc rw_file2.c -o rw_file2
$ ./rw_file2
Name: hongdy
Age:28
Address:shanghai, china
```

四.1.5 刷新文件

fflush 函数，强制冲洗一个流

```
#include<stdio.h>
int fflush(FILE *fp);
返回值：成功返回 0；出错返回 EOF
```

四.1.6 定位文件

定位流函数

```
#include <stdio.h>
int fseek(FILE *fp,long offset,int whence);
whence：SEEK_SET 0；SEEK_CUR 1；SEEK_END 2
返回值：成功返回 0；出错返回非 0
```

四.1.7 文件状态

每个流在 FILE 对象中维持了两个标志：出错标志和文件结束标志。

出错函数

```
#include <stdio.h>
int ferror(FILE *fp);                    返回值：出错返回非 0，
int feof(FILE *fp);                      返回值：为真则为非 0（真），否则为 0
（假）
void clearerr(FILE *fp);                 清除出错标志和文件结束标志
```

例 **18**：文件定位流和文件状态的例子

```
/* fseek_file.c */

#include <stdio.h>
#include <stdlib.h>

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

```
{
   FILE *readfp;
   FILE *writefp;
   char buf[50];

   if( (writefp=fopen("testfile", "w")) == NULL){
      perror("Error: create file\n");
      exit(1);
   }
   fprintf(writefp, "Name: %s  Age: %d\n", "hongdy", 28);
   fputc('I', writefp);
   fputs(" Love you", writefp);
   fputs("\nchina, shanghai", writefp);
   fclose(writefp);;

   if( (readfp=fopen("testfile", "r")) == NULL){
      perror("Error: open file\n");
      exit(1);
   }

   fseek(readfp, -20, SEEK_END);

   while(! feof(readfp))
   {
      fgets(buf, 50, readfp);
      printf("%s\n", buf);
   }

   fclose(readfp);

   return(0);
}
```

编译、运行：

```
$ gcc fseek_file.c -o fseek_file
$ ./fseek_file
 you

china, shanghai
```

程序先创建一个文件并写入一些数据，然后打开这个文件并定位到最后 20 个字符处，最后每次读取一行并输出直到文件结束。

四.2 文件 IO

文件 IO 是针对文件描述符的，当打开或创建一个文件时即返回文件描述符，IO 操作就是针对该文件描述符进行的。在 POSIX 应用程序中，幻数 0、1、2 应被换成符号常数 STDIN_FILENO、STDOUT_FILENO、STDERR_FILENO，这些常数都定义在头文中<unistd.h>。标准 I/O 流通过预定义文件指针 stdin、stdout 和 stderr 加以引用，这三个文件指针定义在头文件<stdio.h>中。

文件 IO 函数

四.2.1 打开文件

open 函数

```
#include <fcntl.h>
int open(const char *pathname, int oflag,.../*, mode_t mode */);
返回值：成功返回文件描述符；出错返回-1
```

参数：

（1）pathname 新建或打开文件名；

（2）oflag 打开方式：O_RDONLY、O_WRONLY、O_RDWR、O_APPEND、O_CREAT、O_TRUNC、O_NONBLOCK；

（3）mode 文件权限位：r 读权限(4)；w 写权限(2)；x 执行权限(1)，644。

四.2.2 创建文件

creat 函数

```
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
返回值：成功返回为只写打开的文件描述符，若出错返回-1
```

此函数等效于：open(pathname, O_WRONLY｜O_CREAT｜O_TRUNC, mode);

四.2.3 关闭文件

close 函数

```
#include <unistd.h>
int close (int filedes);
返回值：成功返回 0，出错为-1
```

四.2.4 定位文件

lseek 函数

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence);
返回值：成功返回新的文件偏移量；出错为-1
```

参数：

（1）filedes 文件描述符；　（2）offset 文件的偏移量；

（3）whence 文件的偏移方式：SEEK_SET 为 0、EEK_CUR 为 1、EEK_END 为 2。

四**.2.5** 写入文件

write 函数

```
#include <unistd.h>
ssize_t write(int filedes, const void *buff, size_t nbytes);
返回值：成功返回已写的字节数；错为-1
```

四**.2.6** 读取文件

read 函数

```
#include <unistd.h>
ssize_t read(int filedes, void *buff, size_t nbytes);
返回值：读到的字节数，若已到文件尾为 0，若出错为-1
```

四**.2.7** 链接文件

link 函数

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
返回值：成功返回 0；错误返回-1
```

四**.2.8** 删除文件

unlink 函数

```
#include <unistd.h>
int unlink(const char *pathname);
返回值：成功返回 0；错误返回-1
```

remove 函数

```
#include <stdio.h>
int remove(const char *pathname);
返回值：成功返回 0；错误返回-1
```

四**.2.9** 重命名文件

rename 函数

```
#include <stdio.h>
int rename(const char *oldpath, const char *newpath);
返回值：成功返回 0；错误返回-1
```

例 **19**：文件读写的例子

```
/* rw_file3.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

typedef struct person
{
   char name[20];
   int  age;
   char address[50];
}person_t;

int main(int argc, char *argv[])
{
   int wfd, rfd;
   person_t p1, p2;

   if(argc < 2){
      fprintf(stderr, "Usage: command <path>\n");
      exit(1);
   }

   // Create file and write file
   if( (wfd=open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0644)) == -1){
      fprintf(stderr, "Error: create file\n");
      exit(1);
   }
   strcpy(p1.name, "hongdy");
   p1.age = 28;
   strcpy(p1.address, "shanghai, china");
   if( (write(wfd, &p1, sizeof(person_t))) == -1){
      fprintf(stderr, "Error: write file\n");
```

```
        exit(1);
    }
    close(wfd);

    // open file and read file
    if( (rfd=open(argv[1], O_RDONLY, 0644)) == -1){
        fprintf(stderr, "Error: open file\n");
        exit(1);
    }
    if( (read(rfd, &p2, sizeof(person_t))) == -1){
        fprintf(stderr, "Error: read file\n");
        exit(1);
    }
    printf("Name:%s\nAge:%d\nAddress:%s\n", p2.name, p2.age, p2.address);
    close(rfd);

    return(0);
}
```

编译、运行：

```
$ gcc rw_file3.c -o rw_file3
$ ./rw_file3 testfile
Name:hongdy
Age:28
Address:shanghai, china
```

四.3 目录操作

Linux 中目录也看作文件，称为目录文件可能更恰当些，它只包含了目录下保存的文件名列表的简单文件。处理目录需要使用特殊的编程接口。

四.3.1 创建目录

mkdir 函数

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
返回值：成功返回 0；失败返回-1
```

四.3.2 删除目录

rmdir 函数

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
int rmdir(const char *pathname);
返回值：成功返回 0；失败返回-1
```

四.3.3 当前目录

getcwd 函数

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

把当前工作目录的绝对路径复制到 buf 中，如果 buf 不够大则返回 NULL，并把 errno 设为 ERANGE。

四.3.4 改变目录

chdir 函数

```
#include <unistd.h>
int chdir(const char *path);
```

把当前目录改为 path 所包含的新目录

例 **20**：创建目录改变目录的例子

```
/* mkdir_dir.c */

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

int main(int argc, char *argv[])
{
   char pathname[100];
   char dirname[150];
   DIR *dir1;
   struct dirent *dirent1;
```

```
    int fd;

    if( (getcwd(pathname, 100)) == NULL){
        perror("Error: pathname");
        exit(1);
    }else{
        printf("Current directory: %s\n", pathname);
    }

    sprintf(dirname, "%s/test", pathname);
    if(mkdir(dirname, 0777) == -1){
        perror("Error: mkdir");
        exit(1);
    }

    if(chdir(dirname) == -1){
        perror("Error: chdir");
        exit(1);
    }

    if(fd=open("hello", O_RDWR | O_CREAT | O_TRUNC, 0644) == -1){
        perror("Error: creat file");
        exit(1);
    }

    close(fd);

    return(0);
}
```

编译、运行：

```
$ gcc mkdir_dir.c -o mkdir_dir
$ ./mkdir_dir
Current directory: /home/hongdy/linux/chap04
$ ls test
hello
```

代码首先得到当前的工作目录，然后在当前目录下创建 test 目录，改变工作目录在此目录下创建 hello 文件，最后关闭目录。
代码运行后查看当前目录可知在当前目录下创建了 test 目录，test 目录下有一个 hello 文件。

四**.3.5** 读取目录

列出目录内容意味着读出目录文件的内容，基本处理操作：

（1）使用 opendir 函数打开目录文件；

（2）使用 readdir 函数读出目录文件的内容，rewinddir 函数把文件指针重定位到目录文件的起始位置；

（3）使用 closedir 函数关闭目录文件。

```
#include <dirent.h>
#include <sys/types.h>
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dir);
int rewinddir(DIR *dir);
int closedir(DIR *dir);
```

```
struct dirent
{
   long d_ino;                          /* inode number */
   off_t d_off;                         /* offset to this dirent */
   unsigned short d_reclen;                  /* length of this d_name */
   char d_name [NAME_MAX+1];                 /* filename (null-terminated) */
}
```

例 **21**：读取目录文件的例子

```
/* read_dir.c */

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>

int main(int argc, char *argv[])
{
   char pathname[100];
   DIR *dir1;
   struct dirent *dirent1;

   if( (getcwd(pathname, 100)) == NULL){
      perror("Error: pathname");
      exit(1);
   }else{
      printf("Current directory: %s\n", pathname);
   }
```

```
    if((dir1=opendir(pathname)) == NULL){
        perror("Error: opendir");
        exit(1);
    }
    while(dirent1=readdir(dir1))
    {
        printf("%s\n", dirent1->d_name);
    }

    closedir(dir1);

    return(0);
}
```

编译、运行：

```
$ gcc read_dir.c -o read_dir
./read_dir
……
```

代码首先得到当前的工作目录，然后打开该目录并列出目录文件的内容，最后关闭目录。

删除文件目录的例子

```
/* del_dir.c */

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

int main(int argc, char *argv[])
{
    char dirname[150];
    char newdirname[150];

    if(getcwd(dirname, 150) == NULL){
        perror("Error: pathname");
        exit(1);
    }
```

```
    sprintf(newdirname, "%s/test", dirname);
    if(chdir(newdirname) == -1){
        perror("Error: chdir");
        exit(1);
    }

    if(remove("hello") == -1){
        perror("Error: delete file\n");
        exit(1);
    }

    if(chdir("../") == -1){
        perror("Error: chdir\n");
        exit(1);
    }

    if(rmdir(newdirname) == -1){
        perror("Error: delete dir\n");
        exit(1);
    }

    return(0);
}
```

编译、运行：

```
$ gcc del_dir.c -o del_dir
./del_dir
……
```

代码首先获取当前的工作目录，然后进入当前目录下的 test 目录，删除 test 目录下的 hello 文件，然后返回到上级目录，最后删除 test 目录。

四.4 出错处理

当 Linux 函数出错时常常返回一个负值，而且整数型变量 errno 通常被设置为含有附加信息的一个值。文件<errno.h>中定义了符号 errno 以及可赋予它的各种常量，这些常量都以字符 E 开头。

C 标准定义了两个函数帮助打印出错信息。

四.4.1 strerror 函数

```
#include <string.h>
char *strerror(int errnum);
返回值：指向消息字符串的指针
```

此函数将 errnum（通常是 errno）映射为一个出错信息字符串并且返回此字符串的指针。

四.4.2 perror 函数

```
#include <stdio.h>
void perror(const char *msg);
```

perror 函数基于 errno 的当前值，在标准出错上生产一条出错信息，然后返回。
首先输出一个有 msg 指向的字符串，然后是一个冒号、一个空格，接着是对应于 errno 值的出错信息，最后是一个换行符。

例 **22**：出错处理的例子

```
/* print_error.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

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

   if(argc < 2){
      fprintf(stderr, "Usage: command <path>\n");
      exit(1);
   }

   if( (fd=open(argv[1], O_RDONLY, 0644)) == -1){
      fprintf(stderr, "1.fprintf  -- Error: create file\n");
      printf("2.strerror -- Error: create file %s\n", strerror(errno));
      perror("3.perror   -- Error: create file");
      exit(1);
   }

   close(fd);
```

```
    return(0);
}
```

编译、运行：

```
$ gcc print_error.c -o print_error
$ ./print_error 123
1.fprintf  -- Error: create file
2.strerror -- Error: create file No such file or directory
3.perror   -- Error: create file: No such file or directory
```

本例子中以只读打开一个并不存在的文件，打印出错信息有三种方法，读者可以比较这三种方法的差别之处。

本章小结

本章首先介绍了基于流的文件打开、写入、读取、定位和关闭文件。其次介绍了基于文件描述符的文件读写方式。然后介绍了对目录的操作，创建目录、删除目录、读取目录文件内容等。最后介绍了代码执行的出错处理。

习题

4.1 分别使用基于流和文件描述符的方式创建文件，写入、读取 struct（成员自定义）数据类型的数据，关闭文件。比较两种方式的差异和运行结果。

4.2 写代码实现在/tmp 目录下创建一个目录，在该目录下创建一个文件并写入数据，将该文件拷贝到当前目录下，删除/tmp 目录下创建的临时目录和包含的文件。

第五章 LINUX 进程及进程控制

五.1 Linux 进程概述

Linux 是一个多用户多任务的操作系统。多用户是指多个用户可以在同一时间使用计算机系统；多任务是指 Linux 可以同时执行几个任务，它可以在还未执行完一个任务时又执行另一项任务。进程是 Linux 操作系统中一个重要的概念，Linux 操作系统可以同时启动多个进程。

根据操作系统的定义：进程是系统资源管理的最小单位。一个进程是一个程序的一次执行的过程，程序是静态的，是保存在磁盘上的可执行的代码和数据集合；进程是动态的是 Linux 系统的基本调度单位。

Linux 操作系统中的每个进程在创建时都会被分配一个数据结构，称为进程控制块 PCB（Process Control Block）。进程控制块中包含了很多重要的信息，供系统调度和进程本身执行使用，其中最重要的是进程 ID（Process ID），进程 ID 也被称作进程标识符，是一个非负的整数，在 Linux 操作系统中唯一地标志了一个进程，Linux 中的每个进程都有不同的进程 PID。一个或多个进程可以合起来构成一个进程组，一个或多个进程组可以合起来构成一个会话。

Linux 系统中的一个进程可以创建一个新进程，即创建子进程。创建子进程的进程称为父进程，所以 Linux 中的每个进程都有父进程（Parent Process ID）。所有的进程追溯到祖先最终会落到进程 PID 为 1 的进程身上，这个进程叫做 init 进程，是内核自举后第一个启动的进程。init 进程作用是扮演终结父进程的角色，因为 init 进程永远不会被终止。

Linux 操作系统包括三种不同类型的进程，每种进程都有自己的特点和属性。
（1）交互进程：由 Shell 启动的进程，交互进程既可以在前台运行，也可以在后台运；
（2）批处理进程：这种进程和终端没有联系，是一个进程序列；
（3）监控进程：也称守护进程，Linux 系统启动时启动的进程，并在后台运行。

五.2 Linux 进程管理

Linux 操作系统下管理进程最好方法就是使用命令行下的系统命令。

五.2.1 启动进程

启动一个进程有两个主要途径：手工启动和调度启动，手工启动进程是在 Shell 终端里由用户输入程序名直接启动一个进程；调度启动进程是用户事先进行设置，根据用户要求自行启动。

启动前台进程是手工启动一个进程的最常用的方式。例如用户在 Shell 里输入一个命令 xeyes，这就已经启动了一个前台进程。在前台进程的运行期间，用户不能输入任何命令，必须等待前台启动的进程运行结束或者终止前台运行的进程，用户才能输入命令。

启动后台进程的一种方法是在 Shell 终端里输入：程序名 &。后面的&符号表示让此进程在后台运行。在后台启动的进程运行期间，用户可以在终端正常输入命令。后台进程运行结束就自动退出。

五.2.2 查看进程

使用 ps 命令可以查看系统中的进程；使用 pstree 可以查看系统中的进程树；使用 top 命令可以动态地显示系统中的进程。

ps 命令的一般用法：ps [options]
ps 命令选项:
（1）a 显示终端上的所有进程，包括其他用户的进程;
（2）u 按用户名和启动时间的顺序来显示进程;
（3）r 显示正在运行中的进程;
（4）x 显示无控制终端的进程;
（5）-e 显示所有进程;
（6）-l 用长格式显示进程;
（7）-f 用树形格式显示进程。

例 **23**：使用 **ps** 命令查看进程的例子

```
$ ps                                          查看正在运行的进程
PID TTY  TIME CMD
2195 pts/0 00:00:00 bash
$ ps u                                        按用户名显示进程
USER  PID %CPU  %MEM  VSZ  RSS  TTY  STAT START  TIME  COMMAND
hongdy 2195  0.0   0.2   5728  1456  pts/0  Ss   04:45   0:00   -bash
$ ps l                                        用长格式显示进程
F UID PID PPID PRI NI VSZ  RSS WCHAN STAT TTY  TIME  COMMAND
0 500 2195 2194 15  0  5728 1456  wait   Ss   pts/0   0:00  -bash
$ ps aux                                      显示所用进程
$ ps aux | more                               结合 | 管道和 more 分页查看
$ ps -elf | grep httpd                        结合 grep 查看具有父子关系的进程
```

从打印的输出结果来看，使用 ps 命令查看进程的显示属性如下表所示:

ps 命令的显示属性

字符	含义

USER	进程的属主
PID	进程的 ID
PPID	父进程
%CPU	进程占用的 CPU 百分比
%MEM	占用内存的百分比
PRI	进程优先级
NI	进程的 NICE 值
VSZ	进程虚拟大小
RSS	驻留中页的数量
TTY	终端 ID
STAT	进程状态
WCHAN	正在等待的进程资源
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

ps 命令标识进程的 5 种状态码：（1）D 不可中断的；（2）R 正在运行的；（3）S 休眠状态的；（4）T 停止或被追踪的；（5）僵尸进程的；另外+号表示位于后台的进程组。

命令 pstree 用来打印输出 PID 为 1 的 init 进程开始的所有的进程树。
pstree 命令的一般用法：pstree

命令 top 用来显示系统当前的进程和其他状况；即可以通过用户按键来不断刷新当前状态，如果在前台执行该命令，它将独占前台直到用户终止该程序为止。
top 命令的一般用法：top –d delay –p pid
-d 选项指定每两次屏幕信息刷新之间的时间间隔。-p 选项指定进程 PID 来仅仅监控这个进程的状态。

例 **24**：使用 **top** 命令动态显示进程状态的例子

$ top	动态显示所有进程状态
$ top -d 1	每隔一秒刷新一次进程状态
$ top -p 2604	动态显示 PID 为 2604 的进程状态

五**.2.3** 终止进程

当需要中断一个前台进程的时候，通常是使用 Ctrl+C 组合键；但是对于一个后台进程就不是一个组合键所能解决的了，必须使用 kill 命令。后台运行的进程也许占用 CPU 资源过多，也许该进程已经变成僵尸进程，kill 命令可以终止后台运行的进程。

kill 命令的使用是和 ps 命令结合在一起使用的；先用 ps 查看进程的 ID，再使用 kill 向此进程发送终止信号。

kill 命令的一般用法：kill [信号代码] 进程 PID

信号代码可以省略，常用的信号代码是 SIGKILL，即数字 9，表示强制终止进程。

例 **25**：使用 **kill** 命令终止进程的例子

$ xeyes &	后台运行 xeyes 进程
$ ps	查看 xeyes 进程的 PID
PID TTY TIME CMD	
2415 pts/0 00:00:00 bash	
$ kill 2415	杀死 xeyes 进程，信号代码可以省略
$ kill -9 2415	使用-9 信号代码，强制终止 xeyes 进程

五**.2.4** 调度进程

Linux 系统中的进程之间是资源竞争关系。这个竞争优劣是通过一个数值来实现的，也就是谦让度。高谦让度表示进程优化级别最低。负值或 0 表示对其它进程不谦让，也就是拥有优先占用系统资源的权利。谦让度的值范围是（-20，19）。

可以在创建进程时使用 nice 命令为进程指定谦让度的值，此时进程优先级的值是父进程优先级的值与 nice 指定谦让度的值的和。所以在用 nice 设置程序的优先级时指定的数值是一个增量，并不是优先级的绝对值;

nice 的一般用法：nice -n 谦让度的增量值 程序

例 **26**：使用 **nice** 和 **renice** 命令改变进程优先级的例子

$ nice -n 5 xeyes &	后台运行 xeyes 进程
$ ps -l	查看 xeyes 进程和父进程属性
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD	
0 S 45179 31625 30638 0 75 0 - 1252 wait pts/3 00:00:00 bash	
0 S 45179 32447 31625 0 80 5 - 1480 - pts/3 00:00:00 xeyes	

程序 xeyes 的父进程 bash 的优先级（PRI）是 75，xeyes 的谦让值（NI）是 5，所以 xeyes 的优先级（PRI）是 75+5=80。

使用 renice 命令可以通过进程 ID 来改变进程的谦让值，进而达到改变进程的优先级。
renice 的一般用法：renice 谦让度 进程 PID

# xeyes &	后台运行 xeyes 进程
# ps -l	查看 xeyes 进程和父进程属性
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD	
4 S 0 32496 32491 0 75 0 - 1188 wait pts/3 00:00:00 bash	
0 S 0 32596 32496 0 76 0 - 1480 - pts/3 00:00:00 xeyes	
# renice -5 32596	改变进程的谦让值
# ps -l	查看 xeyes 进程和父进程属性

```
F S UID PID  PPID  C PRI  NI ADDR SZ  WCHAN TTY  TIME   CMD
4 S 0   32496 32491  0  75  0 -   1188  wait   pts/3  00:00:00  bash
0 S 0   32596 32496  0  71  -5 -   1480  -      pts/3  00:00:00  xeyes
```

使用 renice 命令前，xeyes 程序的优先级是 76，谦让值是 0；使用 renice 命令后，xeyes 程序的谦让值变成-5，优先级变成 76-5=71。

五.3 Linux 进程函数接口

五.3.1 进程标志

每个进程都有自己的进程 ID；每个进程有自己的父进程；调用进程有实际用户 ID；有效用户 ID；调用进程的实际组 ID；有效用户组 ID。

通过下列的函数可以分别得到进程的 ID；父进程的 ID；调用进程的实际用户 ID；有效用户 ID；调用进程的实际组 ID；有效用户组 ID。

```
#include <unistd.h>
pid_t getpid(void);                    返回值：调用进程的进程 ID
pid_t getppid(void);                   返回值：调用进程的父进程 ID
uid_t getuid(void);                    返回值：调用进程的实际用户 ID
uid_t geteuid(void);                   返回值：调用进程的有效用户 ID
gid_t getgid(void);                    返回值：调用进程的实际组 ID
gid_t getegid(void);                   返回值：调用进程的有效组 ID
```

返回类型 pid_t 定义为：typedef int pid_t

五.3.2 创建进程

使用 fork 函数创建一个新进程

```
#include <unistd.h>
pid_t fork(void);
返回值：子进程中为 0，父进程中为子进程 ID，出错为-1
```

fork 函数被调用一次，但返回两次，区别在于子进程的返回值是 0；父进程的返回值是新建子进程的进程 ID。

子进程是父进程的拷贝，子进程获得父进程的数据空间、堆和栈的副本。注意，这是子进程所拥有的副本，父子进程并不共享这些存储空间部分。父进程和子进程的区别： Fork 返回值不同；进程 PID 不同；两个进程具有不同的父进程 ID 等。

使用 fork 函数创建进程失败的主要原因是：（1）系统中已经有了太多的进程；（2）该实际用户 ID 的进程总数超过了系统限制。

例 **27**：创建进程并得到进程标志的例子

```
/* fork_process.c */

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

int glob = 10;

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

   pid_t pid;

   if((pid=fork())<0){
      fprintf(stderr, "Error: fork\n");
      exit(1);
   }else if(pid==0){
      printf("Child Process:\n");
      printf("\tParent PID = %u\n", getppid());
      printf("\tChild  PID = %u\n", getpid());
      glob++;
      printf("\tglob = %d\n", glob);
   }else{
      printf("Parent Process:\n");
      printf("\tSelf PID  = %d\n", getpid());
      printf("\tParent PID = %u\n", getppid());
      printf("\tglob=%d\n", glob);
   }

   return(0);
}
```

编译、运行：

```
$ gcc fork_process.c -o fork_process
$ ./fork_process
Child Process:
      Parent PID = 27281
      Child  PID = 27282
      glob = 11                                  子进程的 glob 值改变了
Parent Process:
      Self PID  = 27281
```

```
    Parent PID = 2185
    glob=10                                父进程中的 glob 未变
```

从例子中可以看出，在子进程里可以得到子进程和它的父进程的 PID，在父进程里可以得到父进程和父父进程的 PID。子进程对变量所做的改变并没有影响到父进程中该变量的值。

另外，父进程和子进程，谁先执行顺序是不一定的，这取决于操作系统的内核调度多使用的调度算法。

从使用 fork 创建进程时的返回值 pid 来看：

pid < 0，创建进程失败；

pid = 0，为子进程，else if(pid==0){ …} 为子进程要执行的代码；

pid > 0，为父进程。else{ …} 为父进程要执行的代码；

读者经常感到疑惑的是：C 语言的控制逻辑 if elif else …条件中只会执行一个条件的代码，这里为什么 else if(pid==0){ …}和 else{ …} 这两段代码都执行了呢?

答：实际上此时有两个进程在同时执行这段代码，每个进程在 if elif else …的条件里，也同样只执行一个条件的代码；子进程的 pid 等于 0，所以会执行 else if(pid==0){ …}这段代码；父进程的 pid 为子进程的 PID，所以会执行 else{ …}这段代码。父子两个进程都向标准输出打印信息，从打印的结果来看，好像程序同时执行了 else if(pid==0){ …} 和 else{ …} 这两段代码。这是一种假象，实际的打印信息是由两个不同的进程输出的。

假如把创建进程的代码改成

```
/* fork_process2.c */

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

int main(int argc, char *argv[])
{
   pid_t pid;

   if((pid=fork())<0){
      fprintf(stderr, "Error: fork\n");
      exit(1);
   }else if(pid==0){
      sleep(10);
      printf("Child Process:\n");
      printf("\tParent PID = %u\n", getppid());
      printf("\tChild  PID = %u\n", getpid());
   }else{
      sleep(10);
```

```
        printf("Parent Process:\n");
        printf("\tSelf PID  = %d\n", getpid());
        printf("\tParent PID = %u\n", getppid());
    }

    return(0);
}
```

编译、运行：

```
gcc fork_process2.c -o fork_process2
$ ./fork_process2 &
$ ps
27105 pts/0    00:00:00 fork_process2
27106 pts/0    00:00:00 fork_process2
…
```

代码是父子两个进程先休眠 10 秒钟然后再打印输出信息。让程序在后台运行，然后使用 ps 命令查看，此时有两个 fork_process2 在运行，一个是父进程，另一个是子进程。

vfork 函数

```
#include <unistd.h>
pid_t vfork(void);
返回值：子进程中为 0，父进程中为子进程 ID，出错为-1
```

vfork 用于创建一个新进程，而该新进程的目的是 exec 一个新程序，但是它并不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），也就不会访问该地址空间。在子进程调用 exec 或 exit 之前，它在父进程的空间中运行。

Vfork 和 fork 的另一个区别是 vfork 保证子进程先运行，在它调用 exec 或 exit 之后父进程才可能被调度运行。

例 **28**：使用 **vfork** 创建进程的例子

```
/* vfork_process.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int glob = 10;

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

    pid_t pid;
```

```
    if((pid=vfork())<0){
        fprintf(stderr, "Error: fork\n");
        exit(1);
    }else if(pid==0){
        printf("Child Process:\n");
        printf("\tParent PID = %u\n", getppid());
        printf("\tChild  PID = %u\n", getpid());
        glob++;
        printf("\tglob = %d\n", glob);
        exit(0);                          调用 exit 终止子进程，或者调用 exec 一个新程序
    }else{
        printf("Parent Process:\n");
        printf("\tSelf PID  = %d\n", getpid());
        printf("\tParent PID = %u\n", getppid());
        glob++;
        printf("\tglob = %d\n", glob);
    }

    return(0);
}
```

编译、运行：

```
$ gcc vfork_process.c -o vfork_process
$ ./vfork_process
Child Process:
    Parent PID = 27353
    Child  PID = 27354
    glob = 11
Parent Process:
    Self PID  = 27353
    Parent PID = 2185
    glob = 12
```

从打印输出结果来看，子进程先执行，然后父进程才执行。父进程必须等待子进程执行完后才开始调度执行。如果将例子中的 vfork 变成 fork，子进程就有可能先执行。子进程对变量 glob 的操作影响了父进程对变量的值，因为此时子进程在父进程的地址空间执行。

五**.3.3** 等待进程

当子进程正常或异常终止时，内核向其父进程发送 SIGCHILD 信号，父进程可以忽略信号或提供该信号发生时被调用执行的函数。等待子进程可以使用 wait 和 waitpid 函数。

```
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *statloc, int options);
两个函数返回值：成功返回子进程 ID；出错返回-1
```

Wait 函数使其调用者阻塞直到有一个子进程终止。
Waitpid 函数可使调用者不阻塞，如果 pid = -1 等待任一子进程；pid > 0 等待与 pid 值相等的子进程；pid = 0 等待其组 ID 等于调用进程组 ID 的任一子进程。

进程一旦调用了 wait，就立即阻塞自己，由 wait 自动分析是否当前进程的某个子进程已经退出，如果它找到了这样一个已经变成僵尸的子进程，wait 就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait 就会一直阻塞在这里，直到有一个出现为止。

例 **29**：使用 **wait** 函数等待进程的例子

```
/* wait_process.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

static int glob = 10;

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

   if((pid=fork())<0){
      fprintf(stderr, "Error: fork\n");
      exit(1);
   }else if(pid==0){       子进程
      glob++;
      printf("child PID = %u\n", getpid());
      printf("\tglob = %d\n", glob);
   }else{                  父进程
      if(pid == wait(&status) ){
         printf("\tchild process exit, return value = %d\n", status);
      }
      glob++;
      printf("Parent PID = %u\n", getpid());
      printf("\tglob = %d\n", glob);
```

```
    }

    return(0);
}
```

编译、运行：

```
gcc wait_process.c -o wait_process
$ ./wait_process
child PID = 3678
    glob = 11
    child process exit, return value = 0
Parent PID = 3677
    glob = 11
```

例 **30**：使用 **waitpid** 函数等待进程的例子

```
/* waitpid_process.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

static int glob = 10;

int main(int argc, char *argv[])
{
    pid_t pid;

    if((pid=fork())<0){
        fprintf(stderr, "Error: fork\n");
        exit(1);
    }else if(pid==0){      子进程
        glob++;
        printf("child PID = %u\n", getpid());
        printf("\tglob = %d\n", glob);
    }else{              父进程
        if(pid == waitpid(pid, NULL, WNOHANG ){
            printf("\tchild process exit\n");
        }
        glob++;
        printf("Parent PID = %u\n", getpid());
        printf("\tglob = %d\n", glob);
```

```
    }

    return(0);
}
```

编译、运行：

```
$ gcc waitpid_process.c -o waitpid_process
$ ./waitpid_process
child PID = 3703
        glob = 11
        child process exit
Parent PID = 3702
        glob = 11
```

此时 waitpid 等待 pid 等于 pid 的进程，即子进程。

五.3.4 exec 函数

当进程调用一种 exec 函数时，该进程执行的程序完全替换为新程序，而新程序从其 main 函数开始执行。因为调用 exec 并不创建新进程，所以前后进程的 PID 并未改变。exec 只是用一个新程序替换了当前进程的正文、数据、堆和栈段。

有 6 中不同的 exec 函数可供使用，它们常被称为 exec 函数。

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);
int execv(const char *pathname, char *const argv [] );
int execle(const char *pathname, const char *arg0, .../* (char *)0, char *const envp []  */);
int execve(const char *pathname, char *const argv [], char *const envp [] );
int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);
int execvp(const char *filename, char *const argv [] );
六个函数返回值：成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。
```

前四个函数去路径名作为参数；后两个函数去文件名作为参数，当指定 filename 作为参数时，如果 filename 中包含/，则将其视为路径名，否则按 PATH 环境变量所指定的各目录中搜寻可执行文件。函数名中的 l 表示 list；v 表示 vector。函数 execl、execlp 和 execle 要求将新程序的每个命令行参数都说明为一个单独的参数，这些参数表以空指针结尾。函数 ececv、execvp 和 execve 则应先构造一个指向各参数的指针数组，然后将该数组地址作为这三个函数的参数。

这 6 个 exec 函数的参数比较难记，函数名中的字母会给我们一些帮助。字母 p 表示函数取 filename 为参数，并且用 PATH 环境变量寻找可执行文件；字母 l 表示函数读取一个参数表；字母 v 表示函数读取一个 argv[]矢量；字母 e 表示函数读取 envp[]数组。

例 **31**：使用 **exec** 相关函数创建进程的例子

（1）execl

```
/* execl.c */

#include <unistd.h>

int main(int argc, char *argv[])
{
   execl("/bin/ls", "ls", "-al", "/etc/passwd", (char *)0);
   return(0);
}
```

编译、运行：

```
$ gcc execl.c -o execl
$ ./execl
-rw-r--r-- 1 root root 1989 Jun 19 22:39 /etc/passwd
```

函数读取参数列表，参数列表的最后一个参数是 0 空指针。

（2）execlp

```
/* execlp.c */

#include <unistd.h>

int main(int argc, char *argv[])
{
   execlp("ls", "ls", "-al", "/etc/passwd", (char *)0);
   return(0);
}
```

编译、运行：

```
$ gcc execlp.c -o execlp
$ ./execlp
-rw-r--r-- 1 root root 1989 Jun 19 22:39 /etc/passwd
```

函数取文件名 ls 为参数，并且根据 PATH 环境变量指定的目录搜索可执行文件 ls，后面跟的是参数列表。

（3）execv

```
/* execv.c */

#include <unistd.h>
int main(int argc, char *argv[])
{
```

```
    char *argv[]={"ls", "-al", "/etc/passwd", (char *)0};
    execv("/bin/ls", argv);
    return(0);
}
```

编译、运行：

```
$ gcc execv.c -o execv
$ ./execv
-rw-r--r-- 1 root root 1989 Jun 19 22:39 /etc/passwd
```

函数读取一个 argv[]矢量作为参数。先将参数列表保存在 argv[]数组里。

一般我们用 fork 创建新进程，exec 执行新程序，exit 函数和两个 wait 函数处理终止和等待终止。

例 **32**：使用 **fork-exec-wait** 创建进程的例子

```
/* fork_exec_wait.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

static int glob = 10;

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

    static int glob=10;

    if((pid=fork())<0){
        fprintf(stderr, "Error: fork\n");
        exit(1);
    }else if(pid==0){                          子进程
        printf("Child Process --> exec \n");
        execl("/bin/ls", "ls", "-al", "/etc/passwd", (char *)0);

    }else{                                     父进程
        if(pid == wait(&status) ){
            printf("\nParend Process --> waite child process exit\n");
```

```
        printf("\tchild process exit, return value = %d\n", status);
    }
  }
  return(0);
}
```

编译、运行：

```
$ gcc fork_exec_wait.c -o fork_exec_wait
$ ./fork_exec_wait
Child Process --> exec
-rw-r--r-- 1 root root 2033 03-30 04:19 /etc/passwd

Parend Process --> waite child process exit
      child process exit, return value = 0
```

程序先用 fork 新建一个子进程，子进程里调用 exec 一个新进程；父进程使用 wait 等待子进程结束。

五.3.5 system 函数

System 函数

```
#include <stdlib.h>
int system( const char* string );
```

system 在其实现中调用了 fork、exec 和 waitpid。system 函数把参数传递给/bin/sh -c 来执行 string 所指定的命令，string 中可以包含选项和参数，接着整个命令行 (/bin/sh -c string) 又传递给系统调用 execve，如果没有找到/bin/sh，system 返回 127；如果出现其他错误则返回－1；如果执行成功则返回 string 的代码。如果 string 为 NULL，system 返回一个非 0 值，否则返回 0。

例 **33**：使用 **system** 函数创建进程的例子

```
/* system1.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int  ret;
  char cmdstring[30];
```

```
    printf("please input command: ");
    gets(cmdstring);

    ret = system(cmdstring);

    if(ret==127){
        fprintf(stderr, "Error: /bin/sh is not available\n");
        exit(127);
    }else if(ret == -1){
        fprintf(stderr, "Error: system\n");
        exit(-1);
    }else if(ret != 0){
        fprintf(stderr, "Error: return value = %d\n", ret);
        exit(ret);
    }else{
        printf("\ncommand executed successfully\n");
    }

    return(0);
}
```

编译、运行：

```
$ gcc system1.c -o system1
$ ./system1
please input command: ls -al
total 56
drwxr-xr-x  2 hongdy hongdy 4096 Oct 30 19:28 .
drwxr-xr-x 19 hongdy hongdy 4096 Oct 30 17:14 ..
-rwxr-xr-x  1 hongdy hongdy 5457 Oct 30 19:27 a.out
-rwxr-xr-x  1 hongdy hongdy 5567 Oct 30 19:28 system1
-rw-r--r--  1 hongdy hongdy  528 Oct 30 19:28 system1.c
-rw-r--r--  1 hongdy hongdy  647 Oct 30 19:06 waitpid_process.c

command executed successfully
$ ./system1
please input command: la
sh: la: command not found
Error: return value = 32512
```

第一次运行执行 ls –al 命令，成功执行；第二次运行，输入一个不存在的命令 la 返回错误。

本章小结

本章首先介绍了进程的基础知识，包括进程属性、进程管理和进程调度等；接着介绍了进程标识、进程创建等，进程创建时有带参数和不带参数两种方式；最后介绍了 exec 函数和 system 函数。

习题

5.1 简单描述进程的几种状态以及这几种状态间的转换?

5.2 写代码实现创建一个带参数的子进程，根据传递的参数，子进程再用 system 执行一个子进程。

第六章 **LINUX** 线程及线程控制

六**.1** 线程概述

传统上并发多任务的实现采用的是运行多个进程。由于各个进程拥有自己独立的运行环境，进程间的耦合关系差，并发粒度过于粗糙，并发实现不太容易。针对传统 Unix 进程的概念在支持中微粒度并行程序设计方面的不足，提出了线程概念。如果把进程所占资源与进程中的运行代码相分离，那么在一个地址空间中便可运行多个指令流，由此产生线程概念。一般说来，线程是程序中的一个单一的顺序控制流。线程是操作系统分配 CPU 时间的基本单位，一个进程中可以有多个线程。

下图中示意了把一个任务按 2 个并发进程和 2 个并发线程分解后的情况。

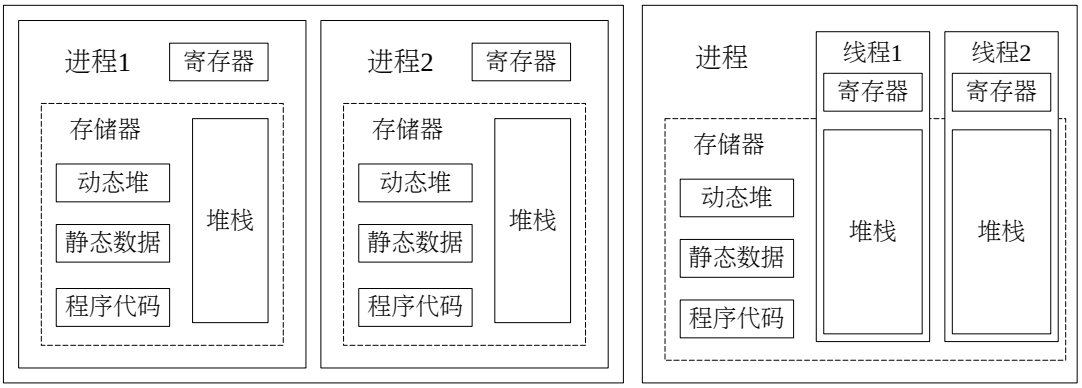

图 **4**：进程和线程比较图

比较图中进程与进程之间、线程与线程之间的关系可以看出：

（1）进程间的关系比较疏远，各个进程是在自己独立的地址空间执行，不但寄存器和堆栈是独有的，动态数据堆、静态数据区和程序代码也相互独立。

（2）线程间的关系则要紧密的多，各个线程为保持自己的控制流而拥有独立的寄存器和堆栈，但由于两线程从属于同一进程，它们共享同一地址空间，所以动态堆、静态数据区及程序代码为各线程共享。进程作为独立的实体，为线程提供运行的资源并构成静态环境

多线程程序设计是在单个程序中包含并发执行的多个线程。当多线程程序执行时，该程序对应的进程中就有多个控制流在同时运行，即具有并发执行的多个线程。在一个进程中包含并发执行的多个控制流，而不是把多个控制流一一分散在多个进程中，这是多线程程序设计与多进程程序设计的不同之处。

线程的优点

线程与进程之间的差别，决定了多线程技术有许多优点：

（1）快速的关联切换，由于进程拥有独立的虚地址空间，调度进程时，系统必须交换地址空间，因而进程切换时间较长。在同一程序内的多个线程共享同一地址空间，因而能使线

程快速切换。

（2）系统开销小，系统对多个进程的创建、调度等有比较大的系统开销。每个进程都要创建自己的进程地址空间，包括寄存器、堆栈、静态数据和程序代码等。而所有的线程共享地址空间，因而消耗系统资源少。

（3）程序执行速度快，程序对应的进程中有多个控制流在同时运行，因而执行速度快。

（4）通信容易实现，为了实现协作，进程或线程之间需要进行数据交换。进程的各个线程共享同一地址空间，所有的全局数据都可以访问，因而不需要什么特殊手段就能自动实现数据共享。而进程之间的通信则比较复杂。

（5）线程个数比进程个数多，许多操作系统限制用户进程总数，如不少 UNIX 版本的典型值为 40～100，这对许多并发应用来说远远不够。在多线程系统中，虽存在线程总数限额，但个数多得多。

线程的缺点

线程也有不足之处，编写多线程程序需要更全面更深入的思考。在一个多线程程序里，因时间分配上的细微偏差或者因共享了不该共享的变量而造成不良影响的可能性是很大的调试一个多线程程序也比调试一个单线程程序困难得多。

线程的结构

线程包含了表示进程内执行环境必需的信息，其中包括进程中标识线程的线程 ID，一组寄存器值、栈、调度优先级和策略、信号屏蔽子，errno 变量以及线程私有数据。进程的所有信息对该进程的所有线程都是共享的，包括可执行的程序文本，程序的全局内存和堆内存、栈以及文件描述符。

六.2 Linux 线程函数接口

六.2.1 线程标识

就像每个进程有一个进程 ID 一样，每个线程也有一个线程 ID，进程 ID 在整个系统中是唯一的，但线程不同，线程 ID 只在它所属的进程环境中有效。线程 ID 用 pthread_t 数据类型来表示，实现的时候可以用一个结构来代表 pthread_t 数据类型，所以可以移植的操作系统不能把它作为整数处理。因此必须使用函数来对来对两个线程 ID 进行比较。

获得线程 ID

```
#include <pthread.h>
pthread_t pthread_self(void);
返回值：调用线程的线程 ID
```

比较两个线程 ID，不能直接用等于符号，必须用 pthread_equal 函数。

```
#include <pthread.h>
int pthread_equal(pthread_t tid1,pthread_t tid2);
返回值：若相等返回非 0 值，否则返回 0
```

六.2.2 创建线程

创建一个线程使用 pthread_create 函数

```
#include <pthread.h>
int pthread_create(pthread_t *tidp, const pthread_attr_t *attr,  void *(*start_rtn)(void),
            void *restrict arg);
返回值：若成功则返回 0；则返回错误编号
```

当 pthread_creat 成功返回时，tidp 参数指向的内存单元被设置为新创建线程的线程 ID；attr 参数用于定制各种不同的线程属性，如果把它设置为 NULL 则创建默认的线程属性；新创建的线程从参数 start_rtn 函数的地址开始运行；该函数只有一个无类型指针参数 arg，如果需要向 start_rtn 函数传递的参数不止一个，可以根据需要把这些参数放到一个结构中，然后把这个结构的地址作为 arg 参数传入。

例 **34**：创建线程的例子

```
/* create_pthread.c */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thr_fn(void *arg)
{
   printf ("new  thread: pid = %u  tid = %u \n", getpid(), pthread_self());
}

int main(int argc, char *argv[])
{
   int err;
   pthread_t tid;
   if( (err=pthread_create(&tid, NULL, thr_fn, NULL)) != 0){
      fprintf(stderr, "can't create thread: %s\n", strerror(err));
      exit(1);
   }

   printf("main thread: pid = %u  tid = %u \n", getpid(), pthread_self());

   sleep(1);
   return(0);
}
```

编译、运行：

```
$ gcc create_pthread.c -lpthread -o create_pthread
$ ./create_pthread
main thread: pid = 25809  tid = 3085817536
new thread : pid = 25809  tid = 3085814672
```

例 **35**：创建带参数线程的例子

```
/* create_pthread2.c */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thr_fn(void *arg)
{
   printf("new  thread: pid = %u  tid = %u \n", getpid(), pthread_self());
   printf("parameter = %s\n", (char *)arg);
}

int main(int argc, char *argv[])
{
   int err;
   char *str="hello, world!";
   pthread_t tid;

   if( (err=pthread_create(&tid, NULL, thr_fn, (void *)str)) != 0){
      fprintf(stderr, "can't create thread: %s\n", strerror(err));
      exit(1);
   }

   printf("main thread: pid = %u  tid = %u \n", getpid(), pthread_self());

   sleep(1);
   return(0);
}
```

编译、运行：

```
$ gcc create_thread2.c -lpthread -o create-pthread2
$ ./create-pthread2
main thread: pid = 22837  tid = 3086796480
new thread : pid = 22837  tid = 3086793616
parameter = hello, world!
```

传递的参数可以是整型，字符串等。如果要传递的参数很多，可以把这些参数放入一个结构体传递过去。

```
/* create_pthread3.c */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

typedef struct person
{
   char name[20];
   int age;
   char address[50];
}person_t;

void *thr_fn(void *arg)
{
   person_t *p;
   p = (person_t *)arg;
   printf("new  thread: pid = %u  tid = %u \n", getpid(), pthread_self());
   printf("%s,%d,%s\n", p->name, p->age, p->address);
}

int main(int argc, char *argv[])
{
   int err;
   pthread_t tid;
   person_t *p1 = malloc(sizeof(person_t));

   strcpy(p1->name, "hongdy");
   p1->age = 28;
   strcpy(p1->address, "shanghai, china");

   if( (err=pthread_create(&tid, NULL, thr_fn, (void *)p1)) != 0){
      fprintf(stderr, "can't create thread: %s\n", strerror(err));
      exit(1);
   }

   printf("main thread: pid = %u  tid = %u \n", getpid(), pthread_self());
```

```
    sleep(1);
    return(0);
}
```

编译、运行：

```
$ gcc create_pthread3.c -lpthread -o create_pthread3
$ ./create_pthread3
main thread: pid = 2460  tid = 3086837440
new thread : pid = 2460  tid = 3086834576
hongdy,28,shanghai, china
```

六.2.3 线程属性

线程属性结构为 pthread_attr_t，在头文件/usr/include/bits/pthreadtypes.h 中定义。结构中的元素分别对应着新线程的运行属性，主要包括以下几项：

（1）detachstate，表示线程的分离状态，有两个状态值：PTHREAD_CREATE_DETACH（分离状态）和 PTHREAD_CREATE_JOINABLE（非分离）状态。缺省为非分离状态。

（2）schedpolicy，表示线程的调度策略，主要包括 SCHED_OTHER（正常、非实时）、SCHED_FIFO（实时、先入先出）和 SCHED_RR（实时、轮转）三种，缺省为 SCHED_OTHER。

（3）schedparam，表示线程的调度参数（优先级），仅当调度策略为 SCHED_RR 或 SCHED_FIFO 时才有效，缺省为 0。

（4）inheritsched，表示继承父进程调度，有两个值：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，后者表示继承调用者线程的值。缺省为 PTHREAD_EXPLICIT_SCHED。

（5）scope，表示线程的绑定状态，有两个状态值：PTHREAD_SCOPE_SYSTEM（绑定）和 PTHREAD_SCOPE_PROCESS（非绑定），缺省为非绑定状态。

线程属性对象主要包括是否分离、调度策略、优先级、是否绑定、堆栈大小等。默认的属性为非分离、与父进程相同调度策略、相同优先级、非绑定、缺省 1M 的堆栈等。

为了设置这些属性，Posix 定义了一系列属性设置函数，包括 pthread_attr_init()、pthread_attr_destroy() 函数、pthread_attr_get() 函数，pthread_attr_set() 函数。

初始化线程属性

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
返回值：成功返回 0，出错返回错误值
```

销毁线程属性

```
#include <pthread.h>
int pthread_attr_destroy(pthread_attr_t *attr);
返回值：成功返回 0，出错返回错误值
```

设置线程属性绑定状态

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
返回值：成功返回 0，出错返回错误值
```

参数：（1）attr 指向属性结构的指针；（2）contentionscope 绑定类型：有两个取值：PTHREAD_SCOPE_SYSTEM（0 绑定的）和 PTHREAD_SCOPE_PROCESS（1 非绑定的）。

获取线程属性绑定状态

```
#include <pthread.h>
int pthread_attr_getscope(const pthread_attr_t *restrict attr, int *restrict contentionscope);
返回值：成功返回 0，出错返回错误值
```

参数：（1）attr 指向属性结构的指针；（2）contentionscope 制定线程绑定类型的指针。

线程的分离状态决定一个线程以什么样的方式来终止自己。线程的默认属性为非分离状态这种情况下原有的线程等待创建的线程结束。只有当 pthread_join() 函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。而分离线程不是这样子的，它没有被其他的线程所等待，自己运行结束了，线程也就终止了，马上释放系统资源。程序员应该根据自己的需要选择适当的分离状态。

设置线程分离状态

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
返回值：成功返回 0，出错返回错误值
```

参数：（1）attr 线程属性；（2）detachstate 线程状态属性：PTHREAD_CREATE_JOINABLE（0 非分离线程）和 PTHREAD_CREATE_DETACHED（分离线程）。

获取线程分离状态

```
#include <pthread.h>
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
返回值：成功返回 0，出错返回错误值
```

参数：（1）attr 指向线程属性的指针；（2）detachstate 指向线程分离状态的指针。

设置线程属性调度策略

```
#include <pthread.h>
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
返回值：成功返回 0，出错返回错误值
```

参数：（1）attr 指向线程属性的指针；（2）policy 调度策略，有 SCHED_OTHER / 0 、SCHED_FIFO / 1 和 SCHED_FIFO / 2。

获取线程属性调度策略

```
#include <pthread.h>
int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr, int *policy);
返回值：成功返回 0，出错返回错误值
```

参数：（1）attr 指向线程属性的指针；（2）policy 指向调度策略的指针。

设置线程属性调度优先级

```
#include <pthread.h>
int pthread_attr_setschedparam(pthread_attr_t * attr,const struct sched_param *param);
返回值：成功返回 0，出错返回错误值
```

参数：（1）attr 指向线程属性的指针；（2）param 指向线程调度参数的指针，sched_param 结构体如下所示。

```
struct sched_param
{
   int sched_priority;
};
```

获取线程属性调度优先级

```
#include <pthread.h>
int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param);
返回值：成功返回 0，出错返回错误值
```

参数：（1）attr 指向线程属性的指针；（2）param 指向线程调度参数的指针。

例 **36**：设置获取线程属性的例子

```
/* pthread_attr.c */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thr_fn(void *arg)
{
   printf ("new  thread: pid = %u  tid = %u \n", getpid(), pthread_self());
}

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

```
{
   int err;
   int policy;
   int scope;
   int detachstate;
   pthread_t tid;
   pthread_attr_t attr;
   struct sched_param param;

   pthread_attr_init(&attr);

   if( (err=pthread_create(&tid, &attr, thr_fn, NULL)) != 0){
      fprintf(stderr, "can't create thread: %s\n", strerror(err));
      exit(1);
   }

   pthread_attr_getscope(&attr, &scope);
   printf("pthread scope  state = %d (0 PTHREAD_SCOPE_SYSTEM /
         1 PTHREAD_SCOPE_PROCESS )\n", scope);

   pthread_attr_getdetachstate(&attr, &detachstate);
   printf("pthread detach state = %d (0 PTHREAD_CREATE_JOINABLE/
         1 PTHREAD_CREATE_DETACHED)\n", detachstate);

   pthread_attr_getschedpolicy(&attr, &policy);
   printf("pthread sched policy = %d (0 SCHED_OTHER /
         1 SCHED_FIFO / 2 SCHED_FIFO)\n", policy);

   pthread_attr_getschedparam(&attr, &param);
   printf("pthread old priority = %d\n", param.sched_priority);

   param.sched_priority = 100;
   pthread_attr_setschedparam(&attr, &param);
   pthread_attr_getschedparam(&attr, &param);
   printf("pthread new priority = %d\n", param.sched_priority);

   pthread_attr_destroy(&attr);

   sleep(1);
   return(0);
}
```

编译、运行:

```
$ gcc pthread_attr.c -lpthread -o pthread_attr
$ ./pthread_attr
pthread scope  state = 0 (0 PTHREAD_SCOPE_SYSTEM /1 PTHREAD_SCOPE_PROCESS )
pthread detach state =0 (0 PTHREAD_CREATE_JOINABLE /
            1PTHREAD_CREATE_DETACHED)
pthread sched policy = 0 (0 SCHED_OTHER / 1 SCHED_FIFO / 2 SCHED_FIFO)
pthread old priority = 0
pthread new priority = 100
new  thread: pid = 20222  tid = 3086510992
```

六.2.4 终止线程

一般来说 Posix 线程终止有两种情况：正常终止和非正常终止。线程主动调用 pthread_exit() 或者从线程函数中 return 都将使线程正常退出，这是可预见的退出方式；非正常终止是线程在其他线程的干预下，或者由于自身运行出错（比如访问非法地址）而退出，这种退出方式是不可预见的。

如果进程中任何一个线程中调用 exit，_Exit,或者是_exit，那么整个进程就会终止，与此类似，如果信号的默认的动作是终止进程，那么，把该信号发送到线程会终止进程。

线程的正常退出的方式:

（1）线程只是从启动例程中返回，返回值是线程中的退出码;

（2）线程可以被另一个进程进行终止;

（3）线程自己调用 pthread_exit 函数;

pthread_exit 函数

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);
```

pthread_join 函数

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **rval_ptr);
返回值：若成功返回 0，否则返回错误编号
```

当一个线程通过调用 pthread_exit 退出或者简单地从启动历程中返回时，进程中的其他线程可以通过调用 pthread_join 函数获得线程的退出状态。调用 pthread_join 线程将一直阻塞，直到指定的线程调用 pthread_exit。如果线程只是从它的启动历程返回，rval_ptr 将包含返回码。线程退出不仅仅可以返回线程的整型数值，还可以返回一个复杂的数据结构。

例 **37**：进程终止和终止状态的例子

```
/* exit_pthread.c */

#include <stdio.h>
```

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

void *th_ret;

void *thr_fn1(void *arg)
{
   char *str="thread 1 exit";
   th_ret = (void *)str;
   printf("thread 1 processing\n");
   pthread_exit(th_ret);
}

void *thr_fn2(void *arg)
{
   char *str2="thread2 exit";
   th_ret = (void *)str2;
   printf("thread 2 processing\n");
   pthread_exit(th_ret);
}

int main(int argc, char *argv[])
{
   pthread_t tid1,tid2;
   void *ret;

   pthread_create(&tid1, NULL, thr_fn1, NULL);
   pthread_create(&tid2, NULL, thr_fn2, NULL);

   pthread_join(tid1, &ret);
   printf("thread 1 return: %s\n", (char *)ret);

   pthread_join(tid2, &ret);
   printf("thread 2 return: %s\n", (char *)ret);

   sleep(1);
   return(0);
}
```

编译、运行:

```
$ gcc exit_pthread.c -lpthread -o exit-pthread
$ ./exit-pthread
```

```
thread 1 processing
thread 1 return: thread 1 exit
thread 2 processing
thread 2 return: thread 2 exit
```

六.2.5 线程互斥锁

线程互斥锁类型结构为 pthread_mutex_t，互斥锁属性结构为 pthread_mutexattr_t，它们都是在文件/usr/include/bits/pthreadtypes.h 中定义。

互斥锁属性在创建的时候指定，Linux 线程实现中仅有一个锁类型属性，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。有四个类型的互斥锁：
（1）PTHREAD_MUTEX_TIMED_NP，这是缺省值，也就是普通锁。当一个线程加锁以后，其余请求锁的线程将形成一个等待队列，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；
（2）PTHREAD_MUTEX_RECURSIVE_NP，嵌套锁，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；
（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；
（4）PTHREAD_MUTEX_ADAPTIVE_NP，适应锁，动作最简单的锁类型，仅等待解锁后重新竞争。

初始化互斥锁
使用互斥锁之前要先进行初始化。初始化互斥锁有两种方式：静态方式和动态方式

静态方式

```
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Posix 定义了一个宏 PTHREAD_MUTEX_INITIALIZER 来静态初始化互斥锁，
pthread_mutex_t 是一个结构，而 PTHREAD_MUTEX_INITIALIZER 则是一个结构常量。

动态方式

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
返回值：成功返回 0；错误返回其他值
```

参数：（1）mutex 指定的互斥量；（2）mutexattr 互斥量属性，如果为 NULL 则使用默认属性。

销毁互斥锁

```
#include <pthread.h>
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
返回值：成功返回 0；错误返回其他值
```

销毁一个互斥锁意味着释放它所占用的资源，且要求锁当前处于开放状态。Linux 系统中互斥锁并不占用任何资源，pthread_mutex_destroy() 函数除了检查锁状态以外（锁定状态则返回 EBUSY）没有其他动作。

加锁解锁操作

锁操作主要包括加锁 pthread_mutex_lock()、解锁 pthread_mutex_unlock() 和测试加锁 pthread_mutex_trylock()，无论哪种类型的锁都不可能被两个不同的线程同时得到。对于普通锁和适应锁类型，解锁者可以是同一进程内的任何线程；而检错锁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。

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
int pthread_mutex_trylock(pthread_mutex_t *mutex)
返回值：成功返回 0，错误返回其他值
```

pthread_mutex_trylock() 语义与 pthread_mutex_lock() 类似，不同的是在锁已经被占据时返回 EBUSY 而不是挂起等待。

注意：如果线程在加锁后解锁前被取消，互斥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中解锁。

例 **38**：线程互斥锁的例子

```
/* pthread_mutex.c */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex;

void *send_pthread(void *arg)
{
   int i;
   for(i=0; i<10; i++)
   {
      pthread_mutex_lock (&mutex);
      printf("send_pthread:   send   data %d...\n", i);
```

```
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}

void *receive_pthread(void *arg)
{
    int i;
    for(i=0; i<10; i++)
    {
        pthread_mutex_lock(&mutex);
        printf("receive_pthread: receive data %d...\n", i);
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}

int main(int argc, char *argv[])
{
    int ret;
    pthread_t tid1, tid2;

    printf("create mutex\n");
    pthread_mutex_init(&mutex, NULL);

    // create pthread1
    if( (ret=pthread_create(&tid1, NULL, send_pthread, NULL)) != 0) {
        fprintf(stderr, "Error: create pthread 1 %s\n", strerror(ret));
        exit(1);
    }

    // create pthread2
    if( (ret=pthread_create(&tid2, NULL, receive_pthread, NULL)) != 0){
        fprintf(stderr, "Error: create pthread 2 %s\n", strerror(ret));
        exit(1);
    }

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("destroy mutex\n");
    pthread_mutex_destroy(&mutex);
    return(0);
```

```
}
```

程序创建一个互斥锁，然后创建两个线程，每个线程分别获得互斥锁、打印信息、休眠一秒钟，最后销毁互斥锁。

编译、运行：

```
$ gcc pthread_mutex.c -lpthread -o pthread_mutex
$ ./pthread_mutex
create mutex
send_pthread:    send   data 0...
receive_pthread:  receive data 0...
send_pthread:    send   data 1...
receive_pthread:  receive data 1...
send_pthread:    send   data 2...
receive_pthread:  receive data 2...
send_pthread:    send   data 3...
receive_pthread:  receive data 3...
send_pthread:    send   data 4...
receive_pthread:  receive data 4...
send_pthread:    send   data 5...
receive_pthread:  receive data 5...
send_pthread:    send   data 6...
receive_pthread:  receive data 6...
send_pthread:    send   data 7...
receive_pthread:  receive data 7...
send_pthread:    send   data 8...
receive_pthread:  receive data 8...
send_pthread:    send   data 9...
receive_pthread:  receive data 9...
destroy mutex
```

一次初始化

有时需要对一些 Posix 变量只进行一次初始化，如果程序中多次初始化某个变量就会出现错误。在传统设计中一次性初始化经常通过使用布尔变量来控制，该变量被静态初始化为 0，而任何依赖于初始化的代码都能测试该变量。如果变量值为 0，则可以进行初始化，然后将变量变为 1；如果变量值为 1，则跳过初始化。但在多线程程序设计中，一次性初始化就变得比较复杂，如果多个线程并发地执行初始化代码，两个线程可能发现控制变量都为 0，并且都实行初始话，而该过程本该仅仅执行一次。使用 pthread_once 函数就比较方便。

pthread_once 函数

```
#include <pthread.h>
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
返回值：成功返回 0，错误返回出错号
```

参数：（1）once_control 控制变量；（2）init_routine 初始化服务例程。

例 **39**：一次初始化的例子

```
/* pthread_once.c */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;
pthread_mutex_t mutex;

void once_init_routine(void)
{
   if (pthread_mutex_init(&mutex, NULL) != 0){
      fprintf(stderr, "Error: once_init_routine -- init mutex");
      exit(1);
   } else {
      printf("once_init_routine only execute one time\n");
   }
}

void *thread1(void *arg)
{
   if (pthread_once(&once_control, once_init_routine) != 0){
      fprintf(stderr, "Error: Thread1 pthread_once");
      exit(1);
   }

   if (pthread_mutex_lock(&mutex) != 0){
      fprintf(stderr, "Error: Thread1 thread mutex lock");
      exit(1);
   }

   printf("Thread1: do critical things\n");

   if (pthread_mutex_unlock(&mutex) != 0){
      fprintf(stderr, "Error: Thread1 mutex unlcok");
      exit(1);
```

```
    }
}

void *thread2(void *arg)
{
    if (pthread_once(&once_control, once_init_routine) != 0){
        fprintf(stderr, "Error: Thread2 pthread_once");
        exit(1);
    }

    if (pthread_mutex_lock(&mutex) != 0){
        fprintf(stderr, "Error: Thread2 thread mutex lock");
        exit(1);
    }

    printf("Thread2: do critical things\n");

    if (pthread_mutex_unlock(&mutex) != 0){
        fprintf(stderr, "Error: Thread1 mutex unlcok");
        exit(1);
    }
}

int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;

    if( pthread_create(&tid1, NULL, thread1, NULL) != 0){
        fprintf(stderr, "Error: create thread1");
        exit(1);
    }
    if( pthread_create(&tid2, NULL, thread2, NULL) != 0){
        fprintf(stderr, "Error: create thread2");
        exit(1);
    }

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return(0);
}
```

编译、运行:

```
$ gcc pthread_once.c -lpthread -o pthread_once
$ ./pthread_once
once_init_routine only execute one time
Thread1: do critical things
Thread2: do critical things
```

本章小结

本章首先介绍了线程的基础知识，线程的优点、缺点等；接着介绍了线程标识、线程创建等，线程创建时有带参数和不带参数两种方式。

习题

6.1 简单描述线程和进程的区别，线程有哪些有点和缺点?
6.2 写代码实现一个不断读取用户输入的数据，根据数据指令创建线程处理并将处理结果写入指定的文件。

第七章 **LINUX** 进程间通信

Linux 进程间通讯（IPC）支持进程间各种通信机制。Linux 系统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 的贝尔实验室和 BSD（加州大学伯克利分校）在进程间通信方面的侧重点有所不同。前者对 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”，通信进程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，Linux 则把两者继承了下来。

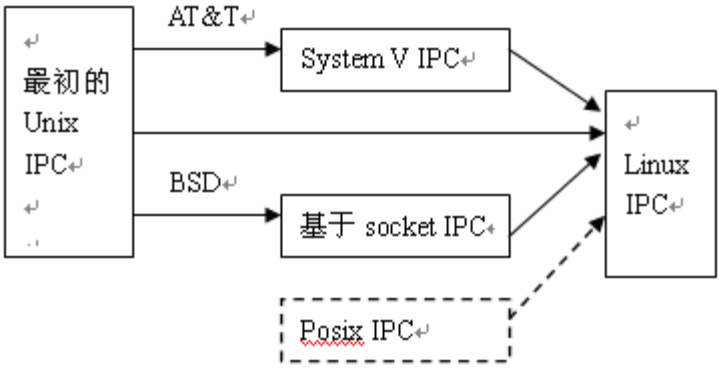

图 **5**：**Linux** 所继承的进程间通信

其中，最初 Unix IPC 包括：管道、FIFO、信号；System V IPC 包括：System V 消息队列、System V 信号量、System V 共享内存区；Posix IPC 包括：Posix 消息队列、Posix 信号量 、Posix 共享内存区。由于 Unix 版本的多样性，电子电气工程协会（IEEE）开发了一个独立的 Unix 标准，这个新的 ANSI Unix 标准被称为计算机环境的可移植性操作系统界面 POSIX（Portable Operating System Interface of Unix）。现有大部分 Unix 和流行版本都是遵循 POSIX 标准的，而 Linux 从一开始就遵循 POSIX 标准。

七**.1** 管道

管道是 Linux 支持的最初 Unix IPC 形式之一，具有以下特点：
（1）管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；
（2）只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）。

七**.1.1 PIPE** 无名管道

创建无名管道

```
#include <unistd.h>
int pipe(int filedes[2]);
返回值：成功返回 0；错则返回-1
```

Linux 用函数 pipe 来创建管道，经由参数 filedes 返回两个文件描述符：filedes[0]为读而打开，filedes[1]为写而打开。filedes[1]的输出是 filedes[0]的输入。

例 **40**：使用 **Pipe** 无名管道的例子

```
/* pipe1.c */

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

int main(int argc, char *argv[])
{
   int  fd[2];
   char write_buf[100]="hello, world";
   char read_buf[100];
   pid_t pid;

   if(pipe(fd) < 0){
      perror("pipe");
      exit(1);
   }

   if((pid = fork()) < 0){
      perror("fork error");
      exit(1);
   }else if(pid > 0) {        父进程
      close(fd[0]);
      printf("\nparent write > %s\n", write_buf);
      write(fd[1], write_buf, 100);
   }else {                  子进程
      close(fd[1]);
      read(fd[0], read_buf, 100);
      printf("\nchild  read > %s\n", read_buf);
   }
   return(0);
}
```

编译、运行：

```
$ gcc pipe1.c -o pipe1
$ ./pipe1
parent write > hello, world
```

```
child  read > hello, world
```

程序先创建一个管道，然后创建一个子进程。父进程向管道里写入数据，子进程从管道读取数据，父进程和子进程分别关闭不使用的管道描述符，这要做的好处是提高了安全性，因为如果父子进程同时往管道里写将发生错误。

七.1.2 FIFO 有名管道

管道应用的一个重大限制是它没有名字，只能用于具有亲缘关系的进程间通信，在有名管道（named pipe 或 FIFO）提出后，该限制得到了克服。FIFO 不同于管道之处在于它提供一个路径名与之关联，以 FIFO 文件的形式存在于文件系统中。这样即使与 FIFO 的创建进程没有亲缘关系的进程，只要可以访问该路径，就能够彼此通过 FIFO 相互通信。FIFO 严格遵循先进先出（First In First Out）原则，对管道及 FIFO 的读取总是从其开始处返回数据，对它们的写入则把数据添加到末尾。

FIFO 的打开规则：
（1）如果当前进程是为读取而打开 FIFO 时，如果已经有其它进程为写入而打开了该 FIFO，则当前进程的打开 FIFO 操作将成功返回。否则：如果该进程为读取而打开该 FIFO 并设置了阻塞标志，则当前进程将阻塞只到有其它进程为写入而打开了该 FIFO；如果该进程为读取而打开该 FIFO 而且没有设置阻塞标志，则当前进程的打开操作也将成功返回。
（2）如果当前进程是为写入而打开 FIFO 时，如果已经有其它进程为读取而打开该 FIFO，则当前进程的打开操作将成功返回。否则：如果该进程为写入而打开该 FIFO 并设置了阻塞标志，则当前进程将阻塞只到有其它进程为读取而打开了该 FIFO；如果该进程为写入而打开该 FIFO 而且没有设置阻塞标志，则当前进程的打开操作也将成功返回。

创建有名管道
Mkfifo 函数

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
返回值：成功返回 0；错误返回-1
```

该函数的第一个参数是一个普通的路径名，也就是创建后 FIFO 的名字。第二个参数与打开普通文件的 open()函数中的 mode 参数相同。如果 mkfifo 的第一个参数是一个已经存在的路径名时，会返回 EEXIST 错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开 FIFO 的函数就可以了。一般文件的 I/O 函数都可以用于 FIFO，如 close、read、write 等等。

例 **41：使用 Fifo 有名管道**的例子

```
/* read_fifo.c */

#include <stdio.h>
```

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

int main(int argc, char *argv[])
{
   char buf[100];
   int fd;
   int readnum;

   if((mkfifo("myfifo", O_CREAT | O_RDWR | O_NONBLOCK | 0644) < 0) &&
      (errno != EEXIST)) {
      fprintf(stderr, "Error:create fifoserver\n");
      exit(1);
   }

   if ((fd = open("myfifo", O_RDONLY | O_NONBLOCK )) < 0) {
      fprintf(stderr, "Error: oepn fifo");
      exit(1);
   }

   printf("Preparing for read from fifo...\n");
   while (1)
   {
      bzero(buf, sizeof(buf));
      if ((readnum = read(fd, buf, sizeof(buf))) > 0) {
         buf[readnum] = '\0';
         printf("read from fifo > %s\n", buf);
         break;
      }
      sleep(1);
    }
    return 0;
}
```

```
/* write_fifo.c */

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

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

int main(int argc, char **argv)
{
   int  fd;
   char buf[100];

   if ((fd = open("myfifo", O_WRONLY | O_NONBLOCK)) < 0) {
      perror("open");
      exit(1);
   }

   printf("Please write data > ");
   fgets(buf, 100, stdin);
   write(fd, buf, 100);

   return 0;
}
```

编译、运行：

打开两个 shell 窗口分别运行这两个进程，一个运行 read_fifo；另一个运行 write_fifo。

```
$ gcc read_fifo.c -o read_fifo
$ ./read_fifo
Preparing for read from fifo...
read from fifo > hongdy
```

```
$ gcc write_fifo.c -o write_fifo
$ ./write_fifo
Please write data > hongdy
```

先运行 read_fifo, 然后再运行 write_fifo，write_fifo 进程输入数据，read_fifo 进程读取数据。

（1）如果 read_fifo.c 在打开 FIFO 时去除 O_NONBLOCK，也就是设置了阻塞标志；write_fifo.c 有 O_NONBLOCK 标志。根据 FFO 的打开规则 read_fifo 进程先运行时将被阻塞，直到 write_fifo 进程为写入 FIFO。

（2）如果 write_fifo.c 在打开 FIFO 时去除 O_NONBLOCK，也就是设置了阻塞标志；read_fifo.c 有 O_NONBLOCK 标志。根据 FFO 的打开规则 write_fifo 进程先运行时将被阻塞，直到 write_fifo 进程为写入 FIFO。

七.2 System V 消息队列

七.2.1 消息队列概述

消息队列是一个消息的链表，有足够写权限的进程可以往队列中放置消息，有足够读权限的进程可从队列中取走消息。消息队列是随内核持续的。一个进程可以先往某个队列写入一些消息后终止，让另外一个进程在以后某个时刻读出这些消息。

消息队列的分类：System V 消息队列和 Posix 消息队列。System V 消息队列目前被大量使用。考虑到程序的可移植性，新开发的应用程序应尽量使用 Posix 消息队列。

与消息队列相关的结构

每个消息队列都有一个队列头，用结构 struct msg_queue 来描述。队列头中包含了该消息队列的大量信息，包括消息队列键值、用户 ID、组 ID、消息队列中消息数目等等，甚至记录了最近对消息队列读写进程的 ID。

```
struct msg_queue
{
   struct kern_ipc_perm q_perm;
   time_t q_stime;                         /* last msgsnd time */
   time_t q_rtime;                         /* last msgrcv time */
   time_t q_ctime;                         /* last change time */
   unsigned long q_cbytes;                      /* current number of bytes on queue */
   unsigned long q_qnum;                        /* number of messages in queue */
   unsigned long q_qbytes;                      /* max number of bytes on queue */
   pid_t q_lspid;                               /* pid of last msgsnd */
   pid_t q_lrpid;                               /* pid of last receive */
   struct list_head q_messages;
   struct list_head q_receivers;
   struct list_head q_senders;
};
```

结构 msqid_ds 用来设置或返回消息队列的信息，存在于用户空间。

七.2.2 创建打开消息队列

消息队列的内核持续性要求每个消息队列都在系统范围内对应唯一的键值，所以，要获得一个消息队列的描述字，只需提供该消息队列的键值即可。msgget 用于创建或打开一个已经存在的队列。

msgget 函数

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

```
#include <sys/msg.h>
#inlcude <sys/ipc.h>
int msgget(key_t key, int msgflag);
返回：成功则为消息队列描述字若出错则为-1。
```

参数：（1）key 是消息队列的键值，键值必须是唯一的，通常通过 ftok 函数获得；
（2）msgflg 是创建消息队列的标识，常用值为：IPC_CREAT、IPC_EXCL、IPC_NOWAIT。

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
返回值：成功返回创建的键值；失败返回-1
```

参数：（1）pathname 文件名；（2）proj_id 工程标识。
ftok 函数将一个文件名和工程标识转化为一个 System V IPC 的一个键值。

七.2.3 控制消息队列

获得、修改消息队列属性，删除消息队列
msgctl 函数

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
返回值：成功返回 0；出错返回-1
```

参数：（1）msgid 消息队列键值；（2）cmd 控制方式：IPC_STAT、IPC_SET 、IPC_RMID。

Cmd	功能
IPC_STAT	获取消息队列信息，返回的信息存贮在 buf 指向的 msqid_da 结构中
IPC_SET	设置消息队列的属性，要设置的属性存储在 buf 指向的 msqid_ds 结构中
IPC_RMID	删除 msqid_ds 标识的消息队列

七.2.4 发送消息队列

msgsnd 函数

```
#include <sys/types.h>
#include <sys/msg.h>
#inlcude <sys/ipc.h>
int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg);
返回值：成功则为 0，若出错则为-1。
```

参数：
（1）msqid 消息队列键值；　　（2）msgp 指向 msgbuf 的指针；

（3）msgsz 发送消息大小，sizeof(struct msgbug)；
（4）msgflg 等待标志 IPC_NOWAIT，当消息队列无足够空间容纳发送的消息时是否等待。

向 msgid 代表的消息队列发送一个消息，即将发送的消息存储在 msgp 指向的 msgbuf 结构中，用户可以自定义 msgbuf 结构。

```
typedef struct my_msgbuf
{
   long mtype；
   int  mshort；
   char mchar[100]；
}my_message_t;
```

七**.2.5** 接收消息队列

msgrcv 函数

```
#include <sys/types.h>
#include <sys/msg.h>
#inlcude <sys/ipc.h>
int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);
返回值：成功则为消息数据部分的长度；出错则为-1
```

参数：
（1）msqid 为消息队列描述字；
（2）msgp 消息返回后存储的指针；
（3）msgsz 消息内容的长度；
（4）msgtyp 请求读取的消息类型：msgtyp==0 返回队列中的第一个消息；msgtyp>0 返回队列中消息类型为 msgtyp 的第一个消息；msgtyp<0 返回队列中消息类型值小于或等于 msgtyp 绝对值的消息。
（5）msgflg 读消息标志，可以为以下几个常值的或：IPC_NOWAIT 如果没有满足条件的消息，调用立即返回，此时，errno=ENOMSG；IPC_EXCEPT 与 msgtyp>0 配合使用，返回队列中第一个类型不为 msgtyp 的消息；IPC_NOERROR 如果队列中满足条件的消息内容大于所请求的 msgsz 字节，则把该消息截断，截断部分将丢失。

例 **42**：使用消息队列的例子

```
/* system-v-message_1.c */

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

```
#include <string.h>

void msg_stat(int,struct msqid_ds);

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

   if(argc<2){
      perror("usage: <pathname>");
      exit(1);
   }

   int msgid;
   struct msqid_ds msg;

   struct msgbuf
   {
      int  mtypes;
      char mtext[30];
   }msg_buf;



   // create message queue
   if( (msgid=msgget(ftok(argv[1], 0), 0644 | IPC_CREAT)) == -1){
      perror("msgget");
      exit(1);
   }else{
      msg_stat(msgid, msg);
   }

   // send message
   msg_buf.mtypes=10;
   strcpy(msg_buf.mtext, "my first message queue");
   if((msgsnd(msgid, &msg_buf, sizeof(struct msgbuf), IPC_NOWAIT)) == -1){
      perror("msgsend");
      exit(1);
   }else{
      printf("\nsend message: %s\n", msg_buf.mtext);
      msg_stat(msgid,msg);
   }

   // receive message
   if( (msgrcv(msgid, &msg_buf, sizeof(struct msgbuf), 10, IPC_NOWAIT |
```

```
        MSG_NOERROR)) == -1){
        perror("msgrcv");
        exit(1);
    }else{
        printf("\nreceived message: %s\n", msg_buf.mtext);
}

    // change msg attribute
    msg.msg_perm.uid=8;
    msg.msg_perm.gid=8;
    msg.msg_qbytes=16388;
    msgctl(msgid,IPC_SET,&msg);
    msg_stat(msgid,msg);

    // delete msg
    if((msgctl(msgid, IPC_RMID, NULL)) == -1){
        perror("msgctl");
        exit(1);
    }
    return(0);
}

void msg_stat(int msgid, struct msqid_ds msg)
{
    msgctl(msgid, IPC_STAT, &msg);

    printf("\ncurrent number of bytes on queue is %d\n",msg.msg_cbytes);
    printf("number of messages in queue is %d\n",msg.msg_qnum);
    printf("max number of bytes on queue is %d\n",msg.msg_qbytes);
    printf("pid of last msgsnd is %d\n",msg.msg_lspid);
    printf("pid of last msgrcv is %d\n",msg.msg_lrpid);
    printf("last msgsnd time is %s", ctime(&(msg.msg_stime)));
    printf("last msgrcv time is %s", ctime(&(msg.msg_rtime)));
    printf("last change time is %s", ctime(&(msg.msg_ctime)));
    printf("msg uid is %d\n",msg.msg_perm.uid);
    printf("msg gid is %d\n",msg.msg_perm.gid);
}
```

编译、运行：

```
# gcc system-v-message_1.c –o system-v-message_1
./system-v-message_1 123
```

例 **43**：使用消息队列实现进程间通信的例子

```
/* system-v-message_server.c */

#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MSGKEY 75

typedef struct MSG_T
{
   long mtype;
   pid_t pid;
   char mtext[100];
}my_msg_t;

int msgid;

void server( )
{
   my_msg_t msg;

   // create message
   if( (msgid=msgget(MSGKEY, 0644 | IPC_CREAT)) == -1){
      perror("msgget");
      exit(1);
   }

   do
   {
      msgrcv(msgid, &msg, sizeof(my_msg_t), 0, 0);
      printf("server receive >name: %d\ttype:%d\tdate:%s\n", msg.pid, msg.mtype, msg.mtext);
   }while(msg.mtype!=5);

   // delete message queue
   if( (msgctl(msgid, IPC_RMID, 0)) == -1){
      perror("msgctl");
      exit(1);
   }
}
```

```
int main(int argc, char *argv[])
{
   server();
   return(0);
}
```

system-v_server.c 创建消息队列，然后接受消息队列，一旦某个消息队列 mtype 等于 5 退出等待，然后删除消息队列。

```
/* system-v-message_client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>

#define MSGKEY 75

typedef struct MSG_T
{
   long mtype;
   pid_t pid;
   char mtext[100];
}my_msg_t;

int msgid;


void client()
{
   int i;
   my_msg_t msg;

   // open message queue
   if( (msgid=msgget(MSGKEY, 0)) == -1){
      perror("msgget");
      exit(1);
   }

   for(i=1; i<6; i++)
```

```
    {
        msg.mtype=i;
        printf("client send > ");
        fgets(msg.mtext, 100, stdin);
        msgsnd(msgid, &msg, sizeof(my_msg_t),0);
        sleep(1);
    }
}

int main(int argc, char *argv[])
{
    client();
    return(0);
}
```

system-v_client.c 打开消息队列，然后发送 5 个消息队列。

编译、运行：

```
# gcc system-v-message_client.c -o system-v-message_client
# gcc system-v-message_server.c -o system-v-message_server
# ./system-v-message_server &
# ./system-v-message_client
client send > hongdy
server receive >name: 9078472   type:1  date:hongdy

client send > hongdy
server receive >name: 9078472   type:2  date:hongdy

client send > shanghai jiaotong university
server receive >name: 9078472   type:3  date:shanghai jiaotong university

client send > china
server receive >name: 9078472   type:4  date:china

client send > shanghai
server receive >name: 9078472   type:5  date:shanghai
```

七.3 System V 信号量

七.3.1 信号量概述

信号量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。

信号量分类：

（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，但常用于线程。

System V 信号量

System V 每一个信号量函数都能对成组的通用信号量进行操作，这在一个进程需要锁定多个资源的时候是很容易办到的。

信号量有一个结构 semid_ds 用于设置信号量的信息。

```
struct semid_ds
{
   struct ipc_perm  sem_perm;                设置权限和所有者
   struct sem sem_base;                      描述一个信号量集中的每个信号量结构
   unsigned short sem_nsems;                 信号量集中信号量的数量
   time_t sem_otime;
   time_t sem_ctime;

};
```

sem_perm 结构的 uid 和 cuid 成员被置为调用进程的有效用户 ID；gid 和 cgid 成员被置为调用进程的有效组 ID；sem_otime 被置为 0；sem_ctime 则被置为当前时间；sem_nsems 被置为 nsems 参数的值。

sem 结构是内核用于维护某个给定信号量的一组值的内部数据结构。一个信号量集的每个成员由下面的结构描述：

```
struct sem
{
   unsigned short_t semval;                  信号量的值
   short sempid;                             对信号量最后调用 semop 函数的进程 ID
   unsigned short_t semncnt;                 等待其值增长的进程数
   unsigned short_t semzcnt;                 等待其值变为 0 的进程数
};
```

内核除维护一个信号量集中每个信号量的实际值之外，内核还给该集合中每个信号量维护另外三个信息：对其值执行最后一次操作的进程的进程 ID、等待其值增长的进程数计数以及等待其值变为 0 的进程数计数。

七**.3.2** 创建打开信号量

semget 函数用于获得一个信号量 ID，作为信号量的外部标识。

```
#include <sys/types.h>
#include <sys/ipc.h>
#inlcude <sys/sem.h>
int semget(key_t key, int nsems, int flag);
返回值：成功信号量 id，若出错返回-1
```

参数：

（1）key 是键值，可以用 ftok 函数来获得；

（2）nsems 参数是需要使用的信号量个数。如果是创建新集合，则必须制定 nsems。如果引用一个现存的集合，则将 nsems 指定为 0。一旦创建完毕一个信号量集，我们就不能改变其中的信号量数；

（3）oflag 参数是一组标志，其作用与 open 函数的各种标志很相似。

例 **44**：使用信号量的例子 **1**

```
/* system-v-semaphore_1.c*/

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

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

   if(argc!=2) {
      perror("Usage:<pathname>");
      exit(1);
   }

   int semid;

   // create sem
   if((semid=semget(ftok(argv[1],0), 1, 0644|IPC_CREAT))==-1){
      perror("semget");
      exit(1);
   }

   // delete sem
   if((value=semctl(semid,0,IPC_RMID)==-1)){
```

```
        perror("semctl");
        exit(1);
    }

    return(0);
}
```

编译、运行：

```
# gcc system-v-semaphore_1 –o system-v-semaphore_1
# ./system-v-semaphore_1 test
```

七.3.3 设置信号量属性

semct 函数，实现对信号量的各种控制操作

获取或设置信号量属性

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid，int semnum，int cmd，union semun arg)；
返回值：调用失败返回-1，成功返回与 cmd 相关
```

参数：

（1）semid 指定信号量集；

（2）semnum 指定对哪个信号量操作，只对几个特殊的 cmd 操作有意义；

（3）cmd 指定具体的操作类型；

（4）arg 用于设置或返回信号量信息。

cmd	含义
IPC_STAT	获取信号量信息，信息由 arg.buf 返回；
IPC_SET	设置信号量信息，待设置信息保存在 arg.buf 中；
GETALL	返回所有信号量的值，结果保存在 arg.array 中，参数 sennum 被忽略；
GETNCNT	返回等待 semnum 所代表信号量的值增加的进程数；
GETPID	返回最后一个对 semnum 所代表信号量执行 semop 操作的进程 ID；
GETVAL	返回 semnum 所代表信号量的值；
GETZCNT	返回等待 semnum 所代表信号量的值变成 0 的进程数；
SETALL	通过 arg.array 更新所有信号量的值；
SETVAL	设置 semnum 所代表信号量的值为 arg.val；

七.3.4 改变信号量状态

semop 函数

```
#include <sys/types.h>
#include <sys/ipc.h>
#inlcude <sys/sem.h>
int semop(int semid, struct sembuf *sops,  size_t nops);
返回值：成功则为 0，若出错则为-1
```

参数：

（1）semid 是信号量的标识码；

（2）sops 是个指向一个结构数组的指针；

（3）nsops 为 sops 指向数组的大小。

结构数组中的元素至少应该包含以下几个成员：

```
struct sembuf
{
   unsigned short sem_num;                信号量编号
   short sem_op;                              信号量操作
   short sem_flg;                             信号量标志
};
```

（1）sem_num 是信号量的编号，0 对应第一个信号量。

（2）sem_op 成员是信号量一次 PV 操作时加减的数值。sem_op 的值大于 0，等于 0 以及小于 0 确定了对 sem_num 指定的信号量进行的三种操作。sem_op>0 对应相应进程要释放 sem_op 数目的共享资源；sem_op=0 可以用于对共享资源是否已用完的测试；sem_op<0 相当于进程要申请-sem_op 个共享资源。

（3）sem_flg 可取 IPC_NOWAIT 以及 SEM_UNDO 两个标志。如果设置了 SEM_UNDO 标志，那么在进程结束时，相应的操作将被取消，这是比较重要的一个标志位。如果设置了该标志位，那么在进程没有释放共享资源就退出时，内核将代为释放。如果为一个信号量设置了该标志，内核都要分配一个 sem_undo 结构来记录它，为的是确保以后资源能够安全释放。事实上，如果进程退出了，那么它所占用就释放了，但信号量值却没有改变，此时信号量值反映的已经不是资源 占有的实际情况，在这种情况下，问题的解决就靠内核来完成。这有点像僵尸进程，进程虽然退出了，资源也都释放了，但内核进程表中仍然有它的记录，此时就需要父进程调用 waitpid 来解决问题了。

例 **45**：使用信号量的例子 **2**

```
/* system-v-semaphore_semops.c*/

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

```
#include <unistd.h>

int main(int argc, char *argv[])
{
   if(argc!=2) {
      perror("Usage:<pathname>");
      exit(1);
   }

   int semid;
   int value;
   struct sembuf sembuf1;

   // create sem
   if((semid=semget(ftok("test", 0), 1, IPC_CREAT | 0644 |) == -1){
      perror("semget");
      exit(1);
   }else{
      printf("create sem : semid=%d\n",semid);
   }

   // set value of semaphore
   if((semctl(semid, 0, SETVAL, 1)) == -1){
      perror("semctl");
      exit(1);
   }

   // get value of semaphore
   if((value=semctl(semid, 0, GETVAL, 0)) == -1){
      perror("semctl");
      exit(1);
   }else{
      printf("value 1 = %d\n",value);
}
   // semop - apply for semaphore
   sembuf1.sem_num = 0;
   sembuf1.sem_op = -1;
   sembuf1.sem_flg = IPC_NOWAIT;
   if((semop(semid, &sembuf1, 1)) == -1){
      perror("semop");
      exit(1);
   }
```

```
    if((value=semctl(semid, 0, GETVAL)) == -1){
        perror("semctl");
        exit(1);
    }else{
        printf("value 2 = %d\n", value);
    }

    // semop - release semaphore
    sembuf1.sem_num = 0;
    sembuf1.sem_op = 1;
    sembuf1.sem_flg = IPC_NOWAIT;
    if((semop(semid, &sembuf1, 1)) == -1){
        perror("semop");
        exit(1);
    }

    if((value=semctl(semid, 0, GETVAL, 0)) == -1){
        perror("semctl");
        exit(1);
    }else{
        printf("value 3 = %d\n",value);
    }

    // delete semaphore
    if((semctl(semid, 0, IPC_RMID, 0)) == -1){
        perror("semctl");
        exit(1);
    }

    return(0);
}
```

编译、运行:

```
# gcc system-v-semaphore_semops –o system-v-semaphore_semops
# ./ system-v-semaphore_semops test
```

七.4 System V 共享存储

七.4.1 共享存储概述

共享内存区是最快的可用 IPC 形式，它允许多个不相关的进程去访问同一部分逻辑内存。如果需要在两个运行中的进程之间传输数据，共享内存将是一种效率极高的解决方案。

一旦这样的内存区映射到共享它的进程的地址空间，这些进程之间数据的传输就不再涉及内核。这样就可以减少系统调用时间，提高程序效率。

共享内存是由 IPC 为一个进程创建的一个特殊的地址范围，它将出现在进程的地址空间中。其他进程可以把同一段共享内存段“连接到”它们自己的地址空间里去。所有进程都可以访问共享内存中的地址。如果一个进程向这段共享内存写了数据，所做的改动会立刻被有访问同一段共享内存的其他进程看到。

共享内存本身没有提供任何同步功能。也就是说，在第一个进程结束对共享内存的写操作之前，并没有什么自动功能能够预防第二个进程开始对它进行读操作。共享内存的访问同步问题必须由程序员负责。可选的同步方式有互斥锁、条件变量、读写锁、纪录锁、信号量。

共享内存区结构

和其他 XSI IPC 一样，内核为每个共享存储段设置一个 shmid_ds 结构。

```
struct shmid_ds
{
   struct ipc_perm shm_perm;                    /*operation perms*/
   int shm_segez;                               /*size of segment*/
   time_t shm_atime;                            /*last attach time*/
   time_t shm_dtime;                            /*last detach time*/
   time_t shm_ctime;                            /*last change time*/
   unsigned short shm_lpid;                  /*pid of creator*/
   unsigned short shm_cpid;                  /*pid of last operator*/
   short shm_nattch;                            /*no.of current attaches*/
};
```

其中 ipc_perm 是我们在 XSI IPC 里介绍的权限结构。

```
struct ipc_perm
{
   key_t key;
   ushort uid;                               /*owner euid and egid*/
   ushort gid;
   ushort cuid;                              /*creator euid and egid*/
   ushort cgid;
   ushort mode;                                 /*lower 9 bits of shmflg*/
    ushort seq;                              /*sequence number*/
};
```

七.4.2 创建打开共享存储

shmget 函数获得一个共享存储标识符

```
#inlcude <sys/shm.h>
#include <sys/ipc.h>
int shmget(key_t key,int size,int shmflg);
```

返回值：成功则返回共享内存 id，若出错则为-1

参数：

（1）key 为共享存储的键值，通过 ftok 获得；

（2）size 共享存储段的长度；如果正在创建一个新段，则必须指定其 size。如果正在引用一个现存的段，则将 size 指定为 0。当创建一新段时，段内的内容初始化为 0。

（3）shmflg 由九个权限标志构成，它们的用法和创建文件时使用的 mode 模式标志是一样的。当需要创建新的共享内存时需要与 IPC_CREAT 标志按位或。设置 IPC_CREAT 标志并传递已存在的共享内存段不会产生错误。如果想创建一个读一无二的共享内存区可以与 IPC_CREAT|IPC_EXCL 按位或，这样如果系统已存在这个共享内存，shmget 函数就会报错。

例 **46**：创建共享内存区的例子

```
/* system-v-shm_shmget.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/ipc.h>

int main(int argc, char **argv)
{
   int id;
   void *buf;

   if(argc!=2){
      perror("usage:shmget <pathname>\n");
      exit(1);
   }

   if( (id=shmget(ftok(argv[1],0),100, 0644 | IPC_CREAT)) == -1){
      perror("shmget");
      exit(1);
   }

   if(buf=shmat(id, NULL, 0)){
      strcpy((char *)buf, "shanghai");
      printf("shm_id=%d buf=%s\n",id, buf);
   }else{
      perror("shmat");
      exit(1);
   }
```

```
    return(0);
}
```

编译、运行：

```
# gcc system-v-shm_shmget.c -o system-v-shm_shmget
# ./system-v-shm_shmget shm1
shm_id=1835013 buf=shanghai
```

七.4.3 映射共享内存地址空间

在共享内存段刚被创建的时候，任何进程还都不能访问它。为了建立这个共享内存段的访问渠道，必须由我们来把它连接到某个进程的地址空间。这项工作是由 shmat 函数完成的。

shmat 函数将共享内存段连接到他的地址空间

```
#include <sys/ipc.h>
#inlcude <sys/shm.h>
void *shmat(int shm_id,void *shm_addr,int shmflg);
返回值：成功则为指向共享存储的指针，若出错则为-1
```

参数：

（1）shm_id 是 shmget 返回的共享内存标识码；

（2）shm_addr 是把共享内存连接到当前进程去的时候准备放置它的那个地址。这通常是一个空指针，表示把选择共享内存出现处的地址这项工作交给系统去完成。

（3）shmflg 是一组按位或的标志。它的两个可能值是 SHM_RND 和 SHM_RDONLY。

shmat 的返回值是该段所连接的实际地址，如果出错则返回-1。如果 shmat 成功执行，那么内核将该共享存储段 shmid_ds 结构的 shm_nattch 计算器加 1。

缺省情况下，只要调用进程具有某个共享内存区的读写权限，它附接该内存区后就能够同时读写该内存区。只有 flag 参数指定 SHM_RDONLY 值时，它以只读方式访问。

七.4.4 断开映射共享存储地址空间

shmdt 函数

```
#inlcude <sys/shm.h>
int shmdt(void *shmaddr);
返回值：成功则为 0，若出错则为-1
```

当一个进程终止时，它的所有当前附接着的共享内存区都自动断掉。注意本函数调用并不是从系统中删除其标识符以及其数据结构。该标识符仍然存在，直至某个进程调用 shmctl 特地删除它。

例 **47**：写共享存储的例子

```
/* system-v-shm_shmwrite.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char **argv)
{
   int id;
   void *buf;

   if(argc!=2){
      perror("usage:shmget <pathname>\n");
      exit(1);
   }

   if((id=shmget(ftok(argv[1],0), 0, 0)) == -1){
      perror("shmget");
      exit(1);
   }

   if(buf=shmat(id, NULL, 0)){
      strcpy((char *)buf, "beijing");
   }else{
      perror("shmat");
      exit(1);
   }

   if(shmdt(buf)== -1){
      perror("shmdt");
      exit(1);
   }

   return(0);
}
```

编译、运行

```
# gcc system-v-shm_shmwrite.c –o system-v-shm_shmwrite
# ./ system-v-shm_shmwrite shm1
```

例 **48**：读共享存储的例子

```
/* system-v-shm_shmread.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char **argv)
{
   int id;
   void *buf;

   if(argc!=2){
      perror("usage:shmget <pathname>\n");
      exit(1);
   }

   if( (id=shmget(ftok(argv[1],0),0,0)) == -1){
      perror("shmget");
      exit(1);
   }

   if(buf=shmat(id, NULL, 0)){
      printf("buf=%s\n",(char *)buf);
   }else{
      perror("shmat");
      exit(1);
   }

   if(shmdt(buf) == -1){
      perror("shmdt");
      exit(1);
   }
   return(0);
}
```

编译、运行：

```
# gcc system-v-shm_shmread.c -o system-v-shm_shmread
# ./ system-v-shm_shmread shm1
buf=beijing
```

七.4.5 控制共享存储

shmctl 函数对共享存储执行多种操作

```
#include <sys/ipc.h>
#inlcude <sys/shm.h>
int shmctl(int shm_id,int cmd,struct shmid_ds *buf);
返回值：成功则为 0，若出错则为-1
```

参数：

（1）shm_d 为共享存储的 ID，用于内部标识共享存储；

（2）cmd 参数指定下列命令中的一种，使其在 shmid 指定的段上执行；

（3）buf 指向一个保存着共享内存的模式状态和访问权限的数据结构。

Cmd	含义
IPC_STAT	取此段的 shmid_ds 结构，并将它存放在由 buf 指向的结果中
IPC_SET	按 buf 指向结构中的值设置与此段相关结构中的下列三个字段：shm_perm.uid，shm_perm.gid 和 shm_perm.mode.此命令只对有效用户 ID 等于 shm_perm.cuid 或 shm_perm.uid 的进程和具有超级用户特权的用户有效
IPC_RMID	从系统中删除该共享存储段
SHM_LOCK	将共享存储段锁定在内存中。此命令只能用超级用户执行
SHM_UNLOCK	解锁共享存储段。此命令只能用超级用户执行

例 **49**：控制共享存储的例子

```
/* system-v-shm_shmctl.c  */

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char *argv[])
{
   int shmid;

   if((shmid=shmget(ftok(argv[1], 0), 0, 0)) == -1){
      perror("shmget");
      exit(1);
   }
```

```
    if((shmctl(shmid,IPC_RMID,NULL)) == -1){
        perror("shmctl");
        exit(1);
    }else{
        printf("delete shared memory: shm_id=%d\n", shmid);
    }

    return(0);
}
```

编译、运行：

```
# gcc system-v-shm_shmctl.c -o system-v-shm_shmctl
# ./system-v-shm_shmctl shm1
```

七.5 Posix 有名信号量

七.5.1 创建打开有名信号量

sem_open 函数

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, /*mode_t mode,unsigned int value*/);
返回值：成功时返回指向信号量的指针，出错时为 SEM_FAILED
```

参数：（1）name 信号量名称；（2）oflag 打开标志，oflag 参数可以是 0、O_CREAT 或 O_CREAT | O_EXCL 如果指定了 O_CREAT，那么第三个和第四个参数是需要的；其中 mode 参数指定权限位，value 参数指定信号量的初始值。该初始不能超过 SEM_VALUE_MAX，这个常值必须低于为 32767。信号量的初始值通常为 1，计数信号量的初始值则往往大于 1。

有名信号量既可用于线程间的同步，又可以用于进程间的同步。

例 **50**：创建 **Posix** 有名信号量的例子

```
/* posix-semaphore_semopen.c */

#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

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

```
{
   if(argc!=2){
      perror("Usage: command <path name>");
      exit(1);
   }

   sem_t *sem;
   if((sem=sem_open(argv[1], O_CREAT, 0644, 1))==SEM_FAILED){
      perror("sem_open");
      exit(1);
   }

   return(0);
}
```

编译、运行：

```
# gcc posix-semaphore_semopen.c -o posix-semaphore_semopen -lrt
# ./posix-semaphore_semopen test
```

七.5.2 关闭有名信号量

sem_close 函数

```
#include <semaphore.h>
int sem_close(sem_t *sem);
返回值：成功则返回 0，否则返回-1。
```

一个进程终止时，内核还对其上仍然打开着的所有有名信号量自动执行这样的信号量关闭操作。但应注意的是关闭一个信号量并没有将它从系统中删除。

七.5.3 删除有名信号量

sem_unlink 函数
从系统中删除信号量

```
#include <semaphore.h>
int sem_unlink(count char *name);
返回值：成功则返回 0，否则返回-1
```

有名信号量使用 sem_unlink 从系统中删除。每个信号量有一个引用计数器记录当前的打开次数，sem_unlink 必须等待这个数为 0 时才能把 name 所指的信号量从文件系统中删除。也就是要等待最后一个 sem_close 发生。

七.5.4 测试信号量

sem_getvalue 函数

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem,int *valp)
返回值：成功则返回 0，否则返回-1
```

sem_getvalue 在由 valp 指向的正数中返回所指定信号量的当前值。如果该信号量当前已上锁，那么返回值或为 0，或为某个负数，其绝对值就是等待该信号量解锁的线程数。

例 **51**：使用 **Posix** 有名信号量的例子

```
/* posix-semaphore_1.c */

#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
   if(argc<2){
      perror("Usage: command <path name>");
      exit(1);
   }

   sem_t *sem;
   int value;

   // create posix semaphore
   if((sem=sem_open(argv[1], O_CREAT, 0644, 1))==SEM_FAILED){
      perror("sem_open");
      exit(1);
   }

   // test posix semaphore
   if((sem_getvalue(sem, &value))==-1){
      perror("sem_getvalue");
      exit(1);
   }else{
      printf("the value of semaphore = %d\n", value);
   }

   // close posix semaphore
```

```
    if((sem_close(sem)) == -1){
        perror("sem_close");
        exit(1);
    }

    // delete posix semaphore
    if((sem_unlink(argv[1])) == -1){
        perror("sem_unlink");
        exit(1);
    }

    return(0);
}
```

编译、运行：

```
# gcc posix-semaphore_1.c -o posix-semaphore_1 -lrt
# ./posix-semaphore_1 sem1
the value of semaphore = 1
```

七.5.5 申请信号量

sem_wait sem_trywait 函数，等待共享资源 申请信号量

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
返回值：成功则返回 0，否则返回-1。
```

可以用 sem_wait 来申请共享资源，sem_wait 函数可以测试所指定信号量的值，如果该值大于 0，那就将它减 1 并立即返回。我们就可以使用申请来的共享资源了。如果该值等于 0，调用线程就被进入睡眠状态，直到该值变为大于 0，这时再将它减 1，函数随后返回 。sem_wait 操作必须是原子的。sem_wait 和 sem_trywait 的差别是：当所指定信号量的值已经是 0 时，后者并不将调用线程投入睡眠。相反，它返回一个 EAGAIN 错误。

例 **52**：申请 **Posix** 有名信号量的例子

```
/* posix-semaphre_semwait.c*/

#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
```

```
int main(int argc, char **argv)
{
   sem_t *sem;
   int value;

   // open posix semaphore
   if((sem=sem_open(argv[1], 0)) == SEM_FAILED){
      perror("sem_open");
      exit(1);
   }

   // test current value of semaphore
   if((sem_getvalue(sem, &value))==-1){
      perror("sem_getvalue");
      exit(1);
   }else{
      printf("current value of semaphore = %d\n", value);
   }

   if((sem_wait(sem)) == -1){
      perror("sem_post");
      exit(1);
   }

   // test value of semaphore after sem_wait
   if((sem_getvalue(sem, &value))==-1){
      perror("sem_getvalue");
      exit(1);
   }else{
      printf("after sem_wait, the value of semaphore = %d\n", value);
   }

   // close posix semaphore
   if((sem_close(sem))==-1){
      perror("sem_close");
      exit(1);
   }
   return(0);
}
```

编译、运行:

```
# gcc posix-semaphore_semwait.c -o posix-semaphore_semwait -lrt
# ./posix-semaphore_semopen test
```

```
# ./posix-semaphore_semwait test
current value of semaphore = 1
after sem_wait, the value of semaphore = 0
```

七.5.6 释放信号量

sem_post 函数

```
#include <semaphore.h>
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem,int *valp);
返回值：成功则返回 0，否则返回-1
```

当一个线程使用完某个信号量时，它应该调用 sem_post 来告诉系统申请的资源已经用完。本函数和 sem_wait 函数的功能正好相反，它把所指定的信号量的值加 1。

例 **53**：释放 **Posix** 有名信号量的例子

```
/* posix-semaphre_sempost.c*/

#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
   sem_t *sem;
   int value;

   // open posix semaphore
   if((sem=sem_open(argv[1], 0)) == SEM_FAILED){
      perror("sem_open");
      exit(1);
   }

   // test current value of semaphore
   if((sem_getvalue(sem, &value))==-1){
      perror("sem_getvalue");
      exit(1);
   }else{
      printf("current value of semaphore = %d\n", value);
   }
```

```
    if((sem_post(sem)) == -1){
        perror("sem_post");
        exit(1);
    }

    // test value of semaphore after sem_post
    if((sem_getvalue(sem, &value))==-1){
        perror("sem_getvalue");
        exit(1);
    }else{
        printf("after sem_post, the value of semaphore = %d\n", value);
}

    // close posix semaphore
    if((sem_close(sem))==-1){
        perror("sem_close");
        exit(1);
    }
    return(0);
}
```

编译、运行：

```
# gcc posix-semaphore_sempost.c -o posix-semaphore_sempost -lrt
# ./posix-semaphore_semopen test
# ./posix-semaphore_sempost test
current value of semaphore = 1
after sem_post, the value of semaphore = 2
```

关于 posix 有名信号量使用的几点注意

（1）Posix 有名信号量的值是随内核持续的。也就是说，一个进程创建了一个信号量，这个进程结束后，这个信号量还存在，并且信号量的值也不会改变。

（2）当持有某个信号量锁的进程没有释放它就终止时，内核并不给该信号量解锁。

例 **54**：等待 **Posix** 有名信号量的例子

```
/* posix-semaphre_process.c*/

#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
```

```
int main(int argc, char **argv)
{
   sem_t *sem;

   // open posix semaphore
   if((sem=sem_open(argv[1], 0)) == SEM_FAILED){
      perror("sem_open");
      exit(1);
   }

   printf("wait for semaphore\n");
   if((sem_wait(sem)) == -1){
      perror("sem_post");
      exit(1);
   }

   printf("processing ....\n");

   printf("release semaphore\n");
   if((sem_post(sem)) == -1){
      perror("sem_post");
      exit(1);
   }

   // close posix semaphore
   if((sem_close(sem))==-1){
      perror("sem_close");
      exit(1);
   }

   return(0);
}
```

编译、运行:

```
#  gcc posix-semaphore_process.c -o posix-semaphore_process -lrt
# ./posix-semaphore_semwait test
current value of semaphore = 1
after sem_wait, the value of semaphore = 0
///run ./posix-semaphore_semwait test repeatly until semaphore become zero
# ./posix-semaphore_process test
wait for semaphore
```

```
///execute “./posix-semaphore_sempost test” in another shell, then will see the following

processing ....
release semaphore
```

七.6 Posix 条件变量

七.6.1 条件变量概述

Posix 条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥锁同时使用。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）。

条件变量的检测是在互斥锁保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。

条件变量类型为 pthread_condattr_t，条件变量属性类型为 pthread_condattr_t，它们都是在文件/usr/include/bits/pthreadtypes.h 中定义。

七.6.2 初始化条件变量

使用条件变量之前要先进行初始化。初始化条件变量有两种方式：静态方式和动态方式。

静态方式是用 Posix 定义的宏 PTHREAD_COND_INITIALIZER 来初始化条件变量。

```
#include <pthread.h>
pthread_cond_t my_condition = PTHREAD_COND_INITIALIZER;
```

动态方式是用 pthread_cond_init()函数来初始化条件变量。

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
返回值：成功返回 0，出错返回错误号
```

参数：（1）cond 指向条件变量的指针；（2）attr 指向条件变量属性的指针。

初始化条件变量属性

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);
返回值：成功返回 0，若失败返回错误号
```

销毁条件变量属性

```
#include <pthread.h>
int pthread_condattr_destroy(pthread_condattr_t *attr);
返回值：成功返回 0，若失败返回错误号
```

七.6.3 销毁条件变量

条件变量不再使用时使用 pthread_cond_destroy() 函数来销毁条件变量。

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
返回值：成功返回 0，出错返回错误编号。
```

参数：cond 指向一个条件变量的指针

pthread_cond_destroy() 函数可用来销毁指定的条件变量，同时将释放所给它分配的资源。

七.6.4 等待条件变量

pthread_cond_wait/pthread_cond_timedwait 函数

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,pthread_mutex_t mytex,const struct timespec *abstime);
返回值：成功返回 0，出错返回错误编号。
```

参数：（1）cond 指向一个条件变量的指针；（2）mutex 则是对相关的互斥锁的指针。

函数 pthread_cond_timedwait() 和函数 pthread_cond_wait() 区别在于如果达到或是超过所引用的参数*abstime，它将结束并返回错误 ETIME。

七.6.5 通知条件变量

pthread_cond_signal/pthread_cond_broadcast 函数

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
返回值：成功返回 0，出错返回错误号
```

参数：cond 指向条件变量的指针

当调用 pthread_cond_signal 时一个在相同条件变量上阻塞的线程将被解锁。如果同时有多个线程阻塞，则由调度策略确定接收通知的线程。如果调用 pthread_cond_broadcast,则将通知

阻塞在这个条件变量上的所有线程。一旦被唤醒，线程仍然会要求互斥锁。如果当前没有线程等待通知，则上面两种调用实际上成为一个空操作。如果参数*cond 指向非法地址，则返回值 EINVAL。

例 **55**：使用 **Posix** 条件变量的例子

```
/* posix_condition.c */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int i=1;
pthread_mutex_t mutex;
pthread_cond_t cond;

void *thread1(void *);
void *thread2(void *);

int main(int argc,char *argv[])
{
   int ret;
   pthread_t tid1;
   pthread_t tid2;

   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init(&cond, NULL);

   if( (ret=pthread_create(&tid1, NULL, thread2, NULL)) != 0){
      fprintf(stderr, "Error: pthread_create %s\n", strerror(ret));
      exit(1);
   }

   if( (ret=pthread_create(&tid2, NULL, thread1, NULL)) != 0){
      fprintf(stderr, "Error: pthread_create %s\n", strerror(ret));
      exit(1);
   }

   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);

   pthread_mutex_destroy(&mutex);
   pthread_cond_destroy(&cond);
```

```
    return(0);
}

void *thread1(void *junk)
{
    for(i=1;i<=9;i++)
    {
        pthread_mutex_lock(&mutex);
        if(i%3 == 0){
            pthread_cond_signal(&cond);
            printf("thread1: i=%d, signal condtion\n", i);
        } else {
            printf("thread1: i=%d\n", i);
        }

        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}

void *thread2(void *junk)
{
    while(i<9)
    {
        pthread_mutex_lock(&mutex);
        if(i%3 != 0){
            pthread_cond_wait(&cond, &mutex);
            printf("thread2: i=%d, waite condition\n", i);
        }
        printf("thread2: i=%d\n", i);
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}
```

编译、运行：

```
$ gcc posix_condtion.c -lpthread
$ ./a.out
thread1: i=1
thread1: i=2
thread1: i=3, signal condtion
thread2: i=3, waite condition
```

```
thread2: i=3
thread1: i=4
thread1: i=5
thread1: i=6, signal condtion
thread2: i=6, waite condition
thread2: i=6
thread1: i=7
thread1: i=8
thread1: i=9, signal condtion
thread2: i=9, waite condition
thread2: i=9
```

条件变量与互斥锁、信号量的区别：
（1）互斥锁必须总是由给它上锁的线程解锁，信号量的挂出即不必由执行过它的等待操作的同一进程执行。一个线程可以等待某个给定信号量，而另一个线程可以挂出该信号量；
（2）互斥锁要么锁住，要么被解开（二值状态，类型二值信号量）；
（3）由于信号量有一个与之关联的状态（它的计数值），信号量挂出操作总是被记住。然而当向一个条件变量发送信号时，如果没有线程等待在该条件变量上，那么该信号将丢失；
（4）互斥锁是为了上锁而优化的，条件变量是为了等待而优化的，信号量即可用于上锁，也可用于等待，因而可能导致更多的开销和更高的复杂性。

七.7 Posix 共享存储

七.7.1 共享存储概述

Posix 共享内存区是最快的可用 IPC 形式，它允许多个不相关的进程去访问同一部分逻辑内存。如果需要在两个运行中的进程之间传输数据，共享内存将是一种效率极高的解决方案。一旦这样的内存区映射到共享它的进程的地址空间，这些进程间数据的传输就不再涉及内核。这样就可以减少系统调用时间，提高程序效率。

共享内存是由 IPC 为一个进程创建的一个特殊的地址范围，它将出现在进程的地址空间中。其他进程可以把同一段共享内存段“连接到”它们自己的地址空间里去。所有进程都可以访问共享内存中的地址。如果一个进程向这段共享内存写了数据，所做的改动会立刻被有访问同一段共享内存的其他进程看到。

要注意的是共享内存本身没有提供任何同步功能。也就是说，在第一个进程结束对共享内存的写操作之前，并没有什么自动功能能够预防第二个进程开始对它进行读操作。共享内存的访问同步问题必须由程序员负责。可选的同步方式有互斥锁、条件变量、读写锁、纪录锁、信号量。

七.7.2 映射共享存取区

mmap 函数把一个文件或一个 Posix 共享内存区对象映射到调用进程的地址空间。使用该函数有三个目的：

（1）使用普通文件以提供内存映射 I/O；

（2）使用特殊文件以提供匿名内存映射；

（3）使用 shm_open 以提供无亲缘关系进程间的 Posix 共享内存区。

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flag, int filedes, off_t off);
返回值：成功则返回映射区的起始地址，若出错则返回 MAP_FAILED
```

参数：

（1）addr 用于指定映射存储区的起始地址。通常将其设置为 NULL，表示由系统选择该映射区的起始地址；

（2）len 是映射的字节数；

（3）prot 参数说明对映射存储区的保护要求，可将 prot 参数指定为 PROT_NONE, 或者是 PROT_READ(映射区可读), PROT_WRITE（映射区可写），PROT_EXEC（映射区可执行）任意组合的按位或。对指定映射存储区的保护要求不能超过文件 open 模式访问权限；

（4）flag 影响映射区的多种属性：MAP_SHARED、MAP_FIXED、MAP_PRIVATE；

（4）filedes 指被映射文件的描述符。在映射该文件到一个地址空间之前，先要打开该文件；

（5）off 是要映射字节在文件中的起始偏移量。通常将其设置为 0。

七.7.3 解除共享存储映射

Munmap 函数

```
#include <sys/mman.h>
int munmap(caddr_t addr,size_t len);
返回值：成功则返回 0，若出错则返回-1
```

参数：（1）addr 是由 mmap 返回的地址；（2）len 是映射区的大小。

再次访问这些地址导致向调用进程产生一个 SIGSEGV 信号。如果被映射区是使用 MAP_PRIVATE 标志映射的，那么调用进程对它所作的变动都被丢弃掉。

内核的虚存算法保持内存映射文件（一般在硬盘上）与内存映射区（在内存中）的同步（前提它是 MAP_SHARED 内存区）。这就是说，如果我们修改了内存映射到某个文件的内存区中某个位置的内容，那么内核将在稍后某个时刻相应地更新文件。然而有时候我们希望确信硬盘上的文件内容与内存映射区中的文件内容一致，于是调用 msync 来执行这种同步。

七.7.4 同步共享存储

Msync 函数

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

```
int msync(void *addr, size_t len, int flags);
返回值：成功则返回 0，若出错则返回-1
```

参数：

（1）addr 同步共享内存的地址；（2）len 共享内存大小；

（3）flags 参数为 MS_ASYNC(执行异步写)，MS_SYNC（执行同步写），MS_INVALIDATE（使高速缓存的数据实效）。其中 MS_ASYNC 和 MS_SYNC 这两个常值中必须指定一个，但不能都指定。它们的差别是，一旦写操作已由内核排入队列，MS_ASYNC 即返回，而 MS_SYNC 则要等到写操作完成后才返回。如果还指定了 MS_INVALIDATE，那么与其最终拷贝不一致的文件数据的所有内存中拷贝都失效。后续的引用将从文件取得数据。

七.7.5 复制映射存储区

Memcpy 函数

```
#include <string.h>
void *memcpy(void *dest,const void *src,size_t n);
返回：dest 的首地址
```

memcpy 拷贝 n 个字节从 src 到 dst。下面就是利用 mmap 函数影射 I/O 实现的 cp 命令。

例 **56**：使用 **Posix** 共享存储的例子

```
/*mycp.c*/

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

int main(int argc,char *argv[])
{
   if(argc!=3) {
      perror("Usage: command <source path> <destion path>");
      exit(1);
   }

   int fdin;
   int fdout;
   void *src;
```

```
    void *dst;
    struct stat statbuf;

    if((fdin=open(argv[1], O_RDONLY))<0){
        perror(argv[1]);
        exit(1);
    }

    if((fdout=open(argv[2], 0644|O_RDWR|O_CREAT|O_TRUNC))<0){
        perror(argv[2]);
        exit(1);
    }

    if((fstat(fdin,&statbuf))<0){
        perror("fstat");
        exit(1);
    }

    if((lseek(fdout, statbuf.st_size-1, SEEK_SET))==-1){
        perror("lseek");
        exit(1);
    }

    write(fdout, "hello", 6);

    if((src=mmap(0, statbuf.st_size, PROT_READ, MAP_SHARED, fdin, 0))==MAP_FAILED){
        perror("src mmap");
        exit(1);
    }

        if((dst=mmap(0, statbuf.st_size, PROT_READ|PROT_WRITE, MAP_SHARED, fdout,
0))==MAP_FAILED){
        perror("dst mmap");
        exit(1);
}

    munmap(src,statbuf.st_size);
    munmap(dst,statbuf.st_size);

    close(fdin);
    close(fdout);
}
```

编译、运行：

```
# gcc –o mycp mycp.c
# vi file1   //write 123456
# ./mycp file1 file2
# cat file2
123456
```

posix 共享内存区涉及两个步骤：

（1）指定一个名字参数调用 shm_open,以创建一个新的共享内存区对象或打开一个以存在的共享内存区对象。

（2）调用 mmap 把这个共享内存区映射到调用进程的地址空间。传递给 shm_open 的名字参数随后由希望共享该内存区的任何其他进程使用。

七.7.6 创建打开一个共享存储区

shm_open 函数

```
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
返回值：功返回共享存储区标志，出错返回-1
```

参数：

（1）name 共享存储名称；

（2）oflag 打开标志，必须含有 O_RDONLY 和 O_RDWR 标志，以及 O_CREAT、O_EXCL 或 O_TRUNC；

（3）mode 参数指定权限位，它指定 O_CREAT 标志的前提下使用。

shm_open 的返回值是一个整数描述字，它随后用作 mmap 的第五个参数。

例 **57**：创建 **Posix** 共享存储的例子

```
/* posix-shm_open.c */

#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

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

```
    struct stat buf;
    char *ptr;

    if(argc!=2) {
        perror("Usage: command <path name>");
        exit(1);
    }

    if((shm_id=shm_open(argv[1], O_RDWR | O_CREAT, 0644)) == -1){
        perror("shm_open");
        exit(1);
    }else{
        printf("shared memory ID = %d\n", shm_id);
    }

    return(0);
}
```

编译、运行：

```
# gcc posix-shm_open.c -o posix-shm_open -lrt
#./posix-shm_open test
shared memory ID = 3
```

七.7.7 删除共享存储区

shm_unlink 函数

```
#include <sys/mman.h>
int shm_unlink(const char *name);
返回值：成功返回 0，出错返回-1
```

例 **58**：删除 **Posix** 共享存储的例子

```
/* posix-shm_unlink.c */

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

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

```
{
   if(argc<2){
      perror("Usage: command <path name>");
      exit(1);
   }

   // delete posix shared memory
   if( (shm_unlink(argv[1])) == -1){
      perror("shm_unlink");
      exit(1);
   }

   return(0);
}
```

编译、运行：

```
# gcc posix-shm_unlink.c -o posix-shm_unlink -lrt
# ./posix-shm_unlink test
```

七.7.8 调整文件或共享存储区大小

ftruncate 函数

```
#include <unistd.h>
int ftruncate(int fd,off_t length);
返回值：成功返回 0，出错返回-1
```

七.7.9 获取文件或共享存储信息

fstat 函数

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *file_name,struct stat *buf);
返回值：成功返回 0，出错返回-1
```

对于普通文件 stat 结构可以获得 12 个以上的成员信息，然而当 fd 指代一个共享内存区对象时，只有四个成员含有信息。

```
struct stat
{
   mode_t st_mode;
   uid_t st_uid;
   gid_t st_gid;
   off_t st_size;
```

```
};
```

例 **59**：获取 **Posix** 共享存储信息的例子

```
/* posix-shm_fstat.c */

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

int main(int argc, char **argv)
{
   if(argc<2){
      perror("Usage: command <path name>");
      exit(1);
   }

   int shmid;
   struct stat buf;

   // create posix shared memory
   if( (shmid=shm_open(argv[1], O_RDWR | O_CREAT, 0644)) == -1){
      perror("shm_open");
      exit(1);
   }else{
      printf("shared memory ID = %d\n", shmid);
   }

   // adjust sharme memory size
   if( (ftruncate(shmid, 100)) != 0){
      perror("ftruncate");
      exit(1);
   }

   // get shared memory stat
   if( (fstat(shmid, &buf)) == -1){
      perror("fstat");
      exit(1);
   } else{
```

```
        printf("mode : %d\n", buf.st_mode);
        printf("uid_t: %d\n", buf.st_uid);
        printf("gid_t: %d\n", buf.st_gid);
        printf("size : %d\n", buf.st_size);
    }

    // delete posix shared memory
    if( (shm_unlink(argv[1])) == -1){
        perror("shm_unlink");
        exit(1);
    }

    return(0);
}
```

编译、运行：

```
# gcc posix-shm_fstat.c -o posix-shm_fstat -lrt
# ./posix-shm_fstat test
shared memory ID = 3
mode : 33188
uid_t: 45179
gid_t: 888
size : 100
```

例 **60**：写 **Posix** 共享存储的例子

```
/* posix-shm_write.c */

#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int shmid;
    struct stat buf;
    void *ptr;

    if(argc!=2) {
```

```
        perror("Usage: command <path name>");
        exit(1);
    }

    if((shmid=shm_open(argv[1], O_RDWR, 0)) == -1){
        perror("shm_open");
        exit(1);
    }

    if((ftruncate(shmid, 100)) == -1){
        perror("ftruncate");
        exit(1);
    }

    if((fstat(shmid, &buf)) == -1){
        perror("fstat");
        exit(1);
    }

    if((ptr=mmap(NULL, buf.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, shmid, 0))
== MAP_FAILED){
        perror("mmap");
        exit(1);
    }else{
        strcpy((char *)ptr, "hello linux");
        printf("%s\n", (char *)ptr);
    }
    return(0);
}
```

例 **61**：读 **Posix** 共享存储的例子

```
/* posix-shm_read.c */

#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

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

```
{
   int shmid;
   struct stat buf;
   void *ptr;

   if(argc!=2) {
      perror("Usage: command <path name>");
      exit(1);
   }

   if((shmid=shm_open(argv[1], O_RDWR, 0)) == -1){
      perror("shm_open");
      exit(1);
   }

   if((ftruncate(shmid,100))==-1){
      perror("ftruncate");
      exit(1);
   }

   if((fstat(shmid, &buf)) == -1){
      perror("fstat");
      exit(1);
   }

   if((ptr=mmap(NULL, buf.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, shmid, 0))
== MAP_FAILED){
      perror("mmap");
      exit(1);
   }else{
      printf("%s\n", (char *)ptr);
   }

   return(0);
}
```

编译、运行:

```
# gcc posix-shm_write.c -o posix-shm_write -lrt
# gcc posix-shm_read.c -o posix-shm_read -lrt
#./posix-shm_open test
shared memory ID = 3
#./posix-shm_write test
hello linux
```

```
#./posix-shm_read test
hello linux
#./posix-shm_unlink test
```

例 **62**：利用 **Posix** 共享存储实现进程间通信的例子

服务器进程读出共享内存区内容，然后清空。客户进程向共享内存区写入数据。直到用户输入“q”程序结束。程序用 posix 信号量实现互斥。

```
/* posix-shm_server.c */

#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int shm_id;
   sem_t *sem;
   char *ptr;

   if(argc!=2) {
      perror("Usage: command <path name>");
      exit(1);
   }

   // create shared memory
   if((shm_id=shm_open(argv[1], O_RDWR | O_CREAT, 0644)) == -1){
      perror("shm_open");
      exit(1);
   }

   // set shared memory size
   if((ftruncate(shm_id, 100)) == -1){
      perror("ftruncate");
      exit(1);
   }
```

```
    // create semaphore
    if(!(sem=sem_open(argv[1], O_CREAT, 0644, 1))){
        perror("sem_open");
        exit(1);
    }

     if((ptr=mmap(NULL, 100, PROT_READ | PROT_WRITE, MAP_SHARED, shm_id, 0)) ==
MAP_FAILED){
        perror("mmap");
        exit(1);
    }

    strcpy(ptr, "\0");

    while(1)
    {
        if((strcmp(ptr, "\0"))==0){
            continue;
        }else {
            sem_wait(sem);                  // apply for semaphore
            if((strcmp(ptr, "q\n"))==0)
                break;
            printf("server > %s\n", ptr);           // printf shared memory content
            strcpy(ptr, "\0");              // clear shared memory
            sem_post(sem);                  // release semaphore
         }
    }

    // delete semaphore
    sem_unlink(argv[1]);

    // delete shared memory
    shm_unlink(argv[1]);

    return(0);
}
```

```
/* posix-shm_client.c */

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

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

int main(int argc, char *argv[])
{
   int shm_id;
   char *ptr;
   sem_t *sem;

   if(argc!=2) {
      perror("Usage: command <path name>");
      exit(1);
   }

   // open shared memory
   if((shm_id=shm_open(argv[1], O_RDWR, 0))==-1){
      perror("shm_open");
      exit(1);
   }

   // open semaphore
   if(!(sem=sem_open(argv[1], 0))){
      perror("sem_open");
      exit(1);
   }

   // map shared memory
    if( (ptr=mmap(NULL, 100, PROT_READ | PROT_WRITE, MAP_SHARED, shm_id, 0)) ==
MAP_FAILED){
      perror("mmap");
      exit(1);
   }

   while(1)
   {
      sem_wait(sem);
      printf("client > ");
      fgets(ptr, 100, stdin);
      sem_post(sem);
      if((strcmp(ptr, "q\n"))==0)
         break;
      sleep(1);
```

```
    }
  return(0);
}
```

编译、运行：

```
# gcc posix-shm_server.c -o posix-shm_server –lrt
# gcc posix-shm_client.c -o posix-shm_client -lrt
# ./posix-shm_server test &
server > hongdy
server > shanghai
server > china

另一个 shell 中
# ./posix-shm_client test
client > hongdy
client > shanghai
client > china
client > q
```

本章小结

本章介绍了 Linux 进程间通信，管道（有名管道和无名管道）、System-V 消息队列、System V 信号量、System-V 共享存储、Posix 有名信号量、Posix 条件变量、Posix 共享变量等。

习题

7.1 比较本章讲述的七种进程间通信的差异?
7.2 分别用这七种进程间通信方式实现文件传输功能?

第八章 **LINUX** 信号

八**.1** 信号概述

八**.1.1** 信号来源

信号是在软件层次上对中断机制的一种模拟。原理上一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，而且进程也不可能知道信号什么时候到达。信号是进程间通信机制中唯一的异步通信机制，可以看作是异步通知，通知接收信号的进程有哪些事情发生了。

信号事件的发生有两个来源：（1）硬件来源，按下键盘或其它硬件故障；（2）软件来源，使用发送信号的系统函数，软件来源还包括一些非法运算等操作。

八**.1.2** 信号分类

可以从两个不同的分类角度对信号进行分类：
（1）可靠性方面：可靠信号与不可靠信号；
（2）时间方面：实时信号与非实时信号。

在 Linux 系统的命令行终端下输入 kill -l 命令可以得到如下信号值：

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	16) SIGSTKFLT
17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU
25) SIGXFSZ	26) SIGVTALRM	27) SIGPROF	28) SIGWINCH
29) SIGIO	30) SIGPWR	31) SIGSYS	34) SIGRTMIN
35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3	38) SIGRTMIN+4
39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12
47) SIGRTMIN+13	48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

不可靠信号与可靠信号

Linux 系统中的信号机制基本上是从 Unix 系统中继承过来的。早期的 Unix 信号机制比较简单和原始，在后来的实践中暴露出一些问题，因此把那些建立在早期机制上的信号叫

做“不可靠信号”，信号值小于 SIGRTMIN 的信号都是不可靠信号。

不可靠信号的主要问题是进程每次处理完信号后，将对信号的响应设置为默认动作。在某些情况下将会导致对信号的错误处理；因此用户如果不希望这样的操作，那么就要在信号的处理函数结束处重新安装该信号。Linux 系统支持不可靠信号，但是对不可靠信号机制做了改进，在调用完信号处理函数后，不必再重新安装该信号。

随着时间的发展，实践证明有必要对信号的原始机制加以改进和扩充。后来出现的各种 Unix 版本分别在这方面对信号进行了研究，力图实现“可靠信号“。由于原来定义的信号已有许多应用，不好再做改动，最终只好又新增加了一些信号，并在一开始就把它们定义为可靠信号，同时信号的发送和安装函数也出现了新的版本，Posix 4 对可靠信号机制做了标准化。可靠信号克服了信号可能丢失的问题。Linux 在支持新版本的信号安装函数 sigation 以及信号发送函数 sigqueue，同时也仍然支持早期的 signal 信号安装函数等。

注意：可靠信号与不可靠信号只与信号值有关，与信号的安装及发送无关。信号值小于 SIGRTMIN 的信号为不可靠信号，信号值在 SIGRTMIN 和 SIGRTMAX 之间的为可靠信号。

非实时信号与实时信号

早期的 Unix 系统只定义了 32 种信号，现在的 Linux 系统支持 64 种信号，编号 0-63（SIGRTMIN=31，SIGRTMAX=63），将来可能进一步增加，这需要得到内核的支持。前 32 种信号已经有了预定义值，每个信号有了确定的用途及含义，并且每种信号都有各自的缺省动作。如按键盘的 Ctrl+c 时，会产生 SIGINT 信号，对该信号的默认反应就是进程终止。后 32 个信号表示实时信号，等同于前面阐述的可靠信号。这保证了发送的多个实时信号都被接收。

注意：非实时信号和实时信号只与信号值有关，与信号的安装及发送无关。信号值小于 SIGRTMIN 的信号为非实时信号，信号值在 SIGRTMIN 和 SIGRTMAX 之间的为实时信号。

八.1.3 信号响应

进程可以通过三种方式来响应一个信号：

（1）忽略信号，对信号不做任何处理。其中有两个信号不能忽略：SIGKILL 及 SIGSTOP；

（2）捕捉信号，定义信号的处理函数，当信号发生时，执行相应的处理函数；

（3）执行缺省操作，Linux 系统对每种信号都规定了默认动作。

注意，进程对实时信号的缺省反应是进程终止。Linux 究竟采用上述三种方式的哪一个来响应信号，取决于传递给相应 API 函数的参数。

八.2 信号安装

信号的安装（设置信号的关联动作），如果进程要处理某一信号，那么就要在进程中安装该信号。安装信号主要用来确定信号值与进程对该信号值的动作之间的映射关系，即进程将要处理哪个信号；该信号被传递给进程时，将执行何种操作。

Linux 主要有两个函数实现信号的安装：signal 函数和 sigaction 函数。早期的 Unix 系统用 signal 来安装信号，Linux 系统对其作了改进。Signal 函数是库函数，它只有两个参数，不支持信号传递信息，主要是用于前 32 种非实时信号的安装；而 sigaction 函数是新版本的信号安装函数，它有三个参数，支持信号参数传递，主要用来与 sigqueue 系统调用配合使用，当然 sigaction 函数同样支持非实时信号的安装。sigaction 函数优于 signal 函数主要体现在支持信号带有参数。

八.2.1 signal 信号安装

signal 信号安装函数

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
返回值：成功则为以前的信号处理配置，若出错则为 SIG_ERR
```

参数：（1）Signum 是要安装的信号值；（2）handler 是对该信号的处理动作。

handler 值可以是：（1）常数 SIG_IGN：表示忽略此信号；（2）常数 SIG_DFL：表示接到此信号后的动作是系统默认动作；（3）函数地址：表示捕捉此信号后执行用户设置的信号处理函数。

例 **63**：忽略信号的例子

```
/* ignore_signal.c */

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

int main(int argc, char *argv[])
{
   printf("ignore signal\n");
   signal(SIGINT, SIG_IGN);                安装 SIGINT 信号，响应动作是忽略此信号

   sleep(10);
   return(0);
```

```
}
```

编译、运行：

```
$ gcc ignore_signal.c -o ignore-signal
$ ./ignore-signal
ignore signal
```

程序执行会在屏幕上先打印一个”ignore signal”，然后睡眠 10 秒钟。在此期间用户按 ctrl+c 没有任何反应，因为 signal 信号安装函数已将 SIGINT 信号（按 ctrl+c 会产生）的响应动作设置为忽略信号。

例 **64**：捕足信号的例子

```
/* catch_signal.c */

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

void catch(int sig)
{
   printf("catch signal\n");
}

int main(int argc, char *argv[])
{
   printf("wait for signal:\n");
   signal(SIGINT,catch);                    安装 SIGINT 信号，响应动作是捕捉此信号

   sleep(10);
   return(0);
}
```

编译、运行：

```
$ gcc catch_signal.c -o catch_signal
$ ./catch_signal
wait for signal:
catch signal
```

程序运行打印”wait for signal”，然后休眠 10 秒钟等待信号，在此期间如果用户按下 ctrl+c 时，进程被中断，catch 函数被执行。中断处理函数处理完转回中断点执行下面的指令。

信号安装好后，程序运行时如果没有信号发生，程序也可以正常结束，因为信号是异步的

假如在程序中要等待某个信号，程序才能继续进行时，可以使用 pause 函数来等待信号。

pause 等待信号函数

```
#include <unistd.h>
int pause(void);
返回值：返回-1，同时 errno 值设置为 EINTR
```

pause 函数使调用进程挂起直至捕捉到一个信号，pause 才返回。在这种情况下，pause 返回-1，errno 设置为 EINTR

例 **65**：使用 **pause** 函数等待信号的例子

```
/* pause_signal.c */

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

static void sig_usr(int signo)
{
   if(signo==SIGUSR1){
      printf("received SIGUSR1\n");
   }else if(signo==SIGUSR2){
      printf("received SIGUSR2\n");
   }else{
      printf("received signal %d\n", signo);
   }
}

int main(int argc, char *argv[])
{
   signal(SIGUSR1, sig_usr);
   signal(SIGUSR2, sig_usr);

   pause();
   return(0);
}
```

编译、运行：

```
$ gcc pause_signal.c -o pause-signal
$ ./pause-signal &
[1] 22206
$ kill -USR1 22206
received SIGUSR1
$
```

```
$ kill -USR2 22206
received SIGUSR2
$ kill -9 22206
 [1]+  Killed                  ./pause-signal
```

让程序在后台运行，当用户 kill -USR1 22206 时产生了 SIGUSR1 信号，signal 定义了处理此信号要调用 sig_usr 函数，所以就在屏幕上打印出 received SIGUSR1。

八.2.2 sigaction 信号安装

sigaction 信号安装函数

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
返回值：成功返回 0，若出错返回-1
```

参数：

（1）signo 是安装的信号值；可以为除 SIGKILL 及 SIGSTOP 外的任何一个信号；

（2）act 指向结构体 sigaction 的指针，在结构 sigaction 的实例中，指定了对安装信号的处理，可以为空，进程会以缺省方式对信号处理；

（3）oact 指向的对象用来保存原来对相应信号的处理，可指定 oldact 为 NULL。如果把第二第三个参数都设为 NULL，那么该函数可用于检查信号的有效性。

sigaction 结构的原形为：

```
struct sigaction
{
   void (*sa_handler)(int signo) ;
   void (*sa_sigaction)(int signu, siginfo_t *info, void *act) ;
   sigset_t sa_mask;
   int sa_flags;
   void (*sa_restore)(void);
}
```

（1）sa_handler 设置信号的处理函数，指定的信号处理函数只有一个参数信号值，不能传递除信号值之外的任何信息；

（2）sa_sigaction 设置信号的处理函数，指定的信号处理函数带有三个参数，第一个参数为信号值，第二个参数是指向 siginfo_t 结构的指针，结构中包含信号携带的数据值，第三个参数没有使用（Posix 没有规范使用该参数的标准）；

（3）sa_mask 用来指定在信号处理程序执行过程中哪些信号应当被阻塞。缺省情况下当前安装的信号被阻塞；

（4）sa_flags 用来设置信号操作的各个情况，一般设置为 0；

（5）sa_restorer 已经过时，Posix 不支持它，不应再被使用。

Siginfo_t 结构的原型为：

```
typedef struct
{
   int si_signo;
   int si_errno;
   int si_code;
   union sigval si_value;
} siginfo_t;
```

该结构体的第四个成员为一个联合数据结构：

```
union sigval
{
   int sival_int;
   void *sival_ptr;
}
```

采用联合数据结构，说明 siginfo_t 结构中的 si_value 要么是整数值，要么是指针，这样就可以用来传递信号的参数。当使用 sigqueue 函数发送信号时，该数据结构中的数据就将拷贝到信号处理函数的第二个参数中。这样在发送信号同时就可以让信号传递一些附加信息。信号可以传递信息对程序开发是非常有意义的。

信号参数的传递如下图所示：

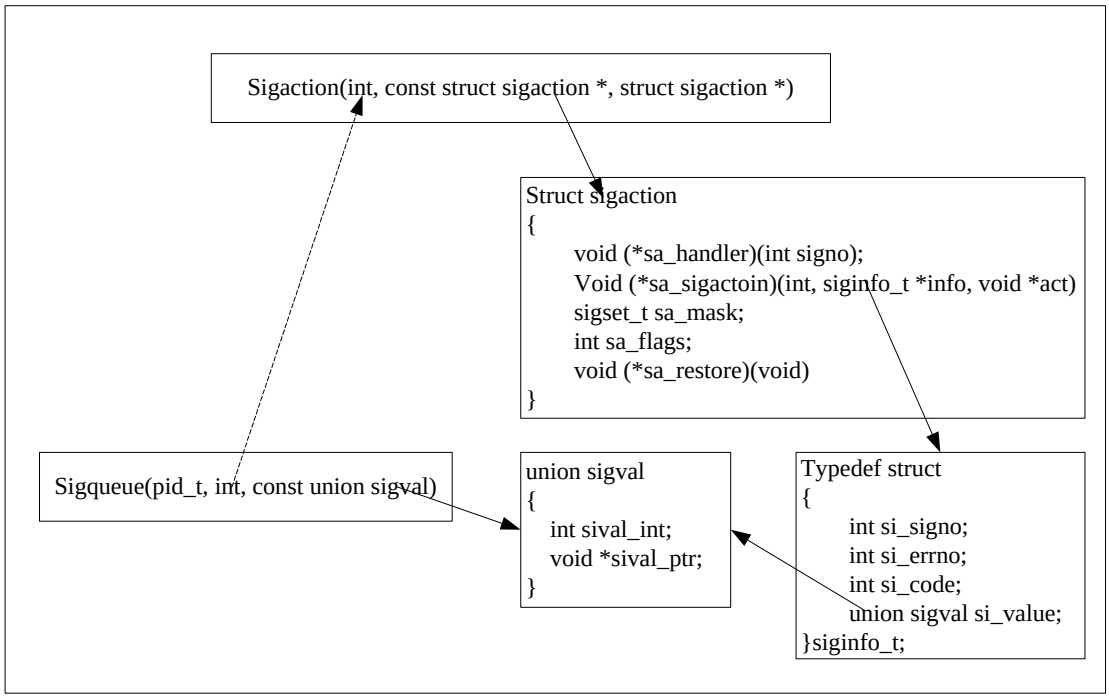

图 **6**：信号参数传递图

信号发送函数 sigqueue 中的第三个参数是 union sigval 联合体；信号安装函数 sigaction 中的第二个参数 struct sigaction 通过其结构体中的成员的成员也是一个 union sigval 联合体。所以通过 sigqueue 信号发送函数时 union sigval 联合体中的数据就被拷贝到 sigaction 信号安装函

数中第二个参数的相关成员中。

例 **66**：使用 **sigaction** 函数安装信号的例子

```
/* sigaction.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
#include <unistd.h>

void sig_act(int signo)
{
   printf("Catch the signal of 'ctrl+c'\nPlease enter 'ctrl+z' to exit\n");
}

int main(int argc, char *argv[])
{
   struct sigaction action;
   action.sa_handler = sig_act;
   action.sa_flags = 0;

   if(sigaction(SIGINT, &action, NULL) <0){
      fprintf(stderr, "Error: sigacton");
      exit(1);
   }

   while(1) { ; }
   return(0);
}
```

编译、运行：

```
$ gcc sigaction.c -o sigaction
$ ./sigaction
Catch the signal of 'ctrl+c'
Please enter 'ctrl+z' to exit

catch the signal of ‘ctrl+c’
please enter ‘ctrl+z’ to exit
```

运行程序后，当用户按 ctrl+c（会产生 SIGINT 信号）后屏幕上会打印信息。

八.3 信号发送

发送信号的主要函数有：kill 函数、raise 函数、alarm 函数、abort 函数和 sigqueue 函数。

八.3.1 kill 函数

kill 函数将信号发送给进程或进程组。

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid,int signo);
返回值：成功返回 0；错误返回-1
```

参数 PID 的值	信号的接收进程
pid>0	进程 ID 为 pid 的进程
pid=0	同一个进程组的进程
pid<0&&pid!=-1	进程组 ID 为-pid 的所有进程
Pid == -1	除发送进程自身外，pid>1 的进程

Sinno 是信号值，当为 0 时（即空信号），实际不发送任何信号，但照常进行错误检查，因此，可用于检查目标进程是否存在。Kill()最常用于 pid>0 时的信号发送，调用成功返回 0；否则返回-1。

例 **67**：使用 **kill** 函数发送信号的例子

```
/* kill_sig.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/signal.h>

void kill_act(int signo)
{
   printf("Catch kill signal\n");
}

int main(int argc, char *argv[])
{
   struct sigaction action;
   action.sa_handler = kill_act;
   action.sa_flags = 0;
```

```
    sigaction(SIGUSR1, &action, NULL);
    printf("wait for kill signal....\n");

    sleep(3);
    kill(getpid(), SIGUSR1);
    return(0);
}
```

编译、运行：

```
$ gcc kill_sig.c -o kill_sig
$ ./kill_sig
wait for kill signal....
Catch kill signal
```

程序先安装 SIGUSR1 信号，然后通过 kill 函数向父进程发送 SIGUSR1 信号，收到信号后执行设定的信号处理函数。

八.3.2 raise 函数

raise 函数

```
#include <signal.h>
int raise(int signo);
返回值：成功返回 0；否则返回 -1
```

调用 raise(signo) 等价于调用 kill(getpid(), signo)函数。

例 **68**：使用 **raise** 函数发送信号的例子

```
/* raise_sig.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/signal.h>

void raise_act(int signo)
{
    printf("Catch raise signal %d\n", signo);
}

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

```
    action.sa_handler = raise_act;
    action.sa_flags = 0;

    sigaction(SIGUSR1, &action, NULL);
    printf("wait for raise signal...\n");
    sleep(3);
    raise(SIGUSR1);
    return(0);
}
```

编译、运行：

```
$ gcc raise_sig.c -o raise_sig
$ ./raise_sig
wait for raise signal...
Catch raise signal 10
```

程序先安装 SIGUSR1 信号，然后通过 raise 函数发送 SIGUSR1 信号，收到信号后执行设定的信号处理函数。

八.3.3 alarm 函数

使用 alarm 函数可以设置一个计时器，当计时器超时时发出 SIGALRM 信号。信号由内核产生，每个进程只能有一个闹钟时间。

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
返回值：如果进程中已经设置了闹钟时间则返回上一个闹钟时间的剩余时间；否则返回 0
```

进程调用 alarm 后，任何以前的 alarm 函数调用都将无效。如果参数 seconds 为零，那么进程内将不再包含任何闹钟时间。如果不忽略或不捕捉此信号，则其默认动作是终止该进程。

例 **69**：使用 **alarm** 函数发送信号的例子

```
/* alarm_sig.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/signal.h>

void alarm_act(int signo)
{
    printf("Catch alarm signal %d\n", signo);
```

```
}

int main(int argc, char *argv[])
{
   struct sigaction action;
   action.sa_handler = alarm_act;
   action.sa_flags = 0;

   if(sigaction(SIGALRM, &action, NULL)<0){
      fprintf(stderr, "Error: sigaction\n");
      exit(1);
   }

   alarm(3);
   pause();
   return(0);
}
```

编译、运行：

```
$ gcc alarm_sig.c -o alarm_sig
$ ./alarm_sig
Catch alarm signal 10
```

程序设置了一个 3 秒的计时器，计时器到时后发出 SIGALRM 信号，执行 alarm_act 函数。Pause 函数使处理程序挂起直到捕足到一个信号。

八.3.4 abort 函数

abort 函数

```
#include <stdlib.h>
void abort(void);
```

向调用该函数的进程发送 SIGABORT 信号，默认情况进程会异常退出，当然用户可定义自己的信号处理函数。该函数无返回值。

例 **70**：使用 **abort** 函数发送信号的例子

```
/* abort_sig.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/signal.h>
```

```
void abort_act(int signo)
{
   printf("Catch abort signal %d\n", signo);
}

int main(int argc, char *argv[])
{
   struct sigaction action;
   action.sa_handler = abort_act;
   action.sa_flags = 0;

   if(sigaction(SIGABRT, &action, NULL)<0){
      fprintf(stderr, "Error: sigaction\n");
      exit(1);
   }

   sleep(3);
   abort();
   return(0);
}
```

编译、运行：

```
$ gcc abort_sig.c -o abort_sig
$ ./abort_sig
Catch abort signal 10
Aborted
```

程序运行后，休眠 3 秒后发出 abort 函数，系统收到 SIGABRT 信号后执行 abort_act 函数。

八.3.5 sigqueue 函数

sigqueue 函数是比较新的发送信号函数，主要是针对实时信号提出的，支持信号带有参数，与 sigaction 信号安装函数配合使用。

```
#include <sys/types.h>
#include <signal.h>
int sigqueue(pid_t pid, int sig, const union sigval val);
返回值：成功返回 0；错误返回-1
```

参数：

（1）pid 指定接收信号的进程 ID；　（2）sig 要发送的信号值；

（3）val 是一个联合数据结构 union sigval，指定了信号传递的参数。union sigval 联合体在 sigaction 信号安装函数时已有介绍。

sigqueue 函数比 kill 函数传递了更多的附加信息，但 sigqueue 只能向一个进程发送信号，而不能发送信号给一个进程组。如果 signo=0，将会执行错误检查，但实际上不发送任何信号，0 值信号可用于检查 pid 的有效性以及当前进程是否有权限向目标进程发送信号。

sigqueue 信号发送函数时 union sigval 联合体中的数据就被拷贝到 sigaction 信号安装函数中第二个参数的相关成员中。由于 sigqueue 系统调用支持发送带参数信号，所以比 kill()系统调用的功能要灵活和强大得多。

例 **71**：使用 **sigqueue** 函数发送信号的例子

```
/* sigqueue_sig.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/signal.h>

void sig_act(int signo)
{
   printf("Catch signal %d\n", signo);
}

int main(int argc, char *argv[])
{
   sigval_t sigval;

   struct sigaction action;
   action.sa_handler = sig_act;
   action.sa_flags = 0;

   sigaction(SIGUSR1, &action, NULL);
   printf("wait for signal...\n");

   sleep(3);

   sigqueue(getpid(), SIGUSR1, sigval);

   return(0);
}
```

编译、运行：

```
$ gcc sigqueue_sig.c -o sigqueue_sig
```

```
$ ./sigqueue_sig
wait for signal...
Catch signal 10
```

例 **72**：使用 **sigqueue** 函数发送信号并传递整型参数的例子

```
/* sigqueue_int.c */

#inlcude <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>

void new_op(int signum, siginfo_t *info, void *act)
{
   printf("Received Signal\n--> Value = %d\n", signum);
   printf("--> Paremeter = %d\n", info->si_int);
}

int main(int argc,char**argv)
{
   if(argc<2){
      fprintf(stderr, “Usage: command signum\n”);
      exit(1);
   }

   int signum;
   struct sigaction act;
   union sigval mysigval;

   signum = atoi(argv[1]);
   mysigval.sival_int = 10;

   sigemptyset(&act.sa_mask);
   act.sa_sigaction=new_op;
   act.sa_flags=SA_SIGINFO;

   // Install signal
   sigaction(signum, &act, NULL);

   printf(“Wait for signal…\n);
   sleep(3);
```

```
    // Send signal
    sigqueue(getpid(), signum, mysigval);

    return(0);
}
```

编译、运行：

```
$ gcc sigqueue_int.c -o sigqueue_int
$ ./sigqueue_int 35
Wait for signal ...
Received Signal
--> Value = 35
--> Paremeter = 10
```

程序运行时先安装一个信号，信号值从运行时的参数获得，这里使用 35 信号值，接着休眠 3 秒钟等待信号，然后向当前进程发送信号并传递参数，进程收到信号后打印收到的信号值和参数。这里传递的参数是整型，也可以传递指针指向的数据。

注意：在信号处理 new_op 函数中的信息 info->si_int 跟前面定义的结构体变量并不一致，这是 si_int 实际上是一个指向 union sigval 成员 sival_int 的宏，具体请查看头文件/usr/include/bits/siginfo.h 中的具体定义。

例 **73**：使用 **sigqueue** 函数发送信号并传递指针参数的例子

```
/* sigqueue_ptr.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>

struct person
{
    char name[10];
    int age;
};

void new_op(int signum, siginfo_t *info, void *act)
{
    struct person *p1 = (struct person *)info->si_ptr;
```

```
    printf("Received signal\n");
    printf("-->value = %d\n", signum);
    printf("-->name  = %s\n", p1->name);
    printf("-->age   = %d\n", p1->age);
}

int main(int argc, char *argv[])
{
    if(argc<2){
        fprintf(stderr, "Usage : command signum\n");
        exit(1);
    }

    int signum;
    struct sigaction act;
    union sigval mysigval;

    signum = atoi(argv[1]);

    struct person p1;
    strcpy(p1.name, "hongdy");
    p1.age = 28;
    mysigval.sival_ptr = (void *)&p1;

    sigemptyset(&act.sa_mask);
    act.sa_sigaction = new_op;
    act.sa_flags = SA_SIGINFO;

    install signal
    sigaction(signum, &act, NULL);

    printf("wait signal...\n");
    sleep(3);

    send signal
    sigqueue(getpid(), signum, mysigval);
    return(0);
}
```

编译、运行:

```
$ gcc sigqueue_ptr.c -o sigqueue_ptr
$ ./sigqueue_ptr 35
wait signal...
```

```
Received signal
-->value = 35
-->name = hongdy
-->age = 28
```

程序运行时先安装一个信号，信号值从运行时的参数获得，这里使用 35 信号值，接着休眠 3 秒钟等待信号，然后向当前进程发送信号并传递参数，进程收到信号后打印收到的信号值和结构体参数。这里传递信号的参数是字符串。

八.4 信号屏蔽字

有时我们希望进程正确的执行而不受到信号的影响，信号操作最常用的方法是信号屏蔽字。

八.4.1 处理信号集

Posix.1 定义了数据类型 sigset_t 的信号集。
信号集函数

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set,int signum);
int sigdelset(sigset_t *set,int signum);
int sigismember(const sigset_t *set,int signum);
返回值：成功返回 0；错误返回-1
```

函数 sigemptyset 初始化由 set 指向的信号集，清除其中所有信号；
函数 sigfillset 初始化由 set 指向的信号集，使其包含所有信号；
函数 sigaddset 将一个信号添加到现有集中；
函数 sigdelset 则从信号集中删除一个信号；
函数 sigismember 查询信号是否在信号集合之中。

例 **74**：使用信号集的例子

```
/* sigset_member.c */

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

int main(int argc, char *argv[])
{
   sigset_t *set;
   set=(sigset_t*)malloc(sizeof(sigset_t));
```

```
    sigemptyset(set);
    sigaddset(set,SIGUSR1);
    sigaddset(set,SIGINT);

    if((sigismember(set,SIGUSR1))==1)
        printf("sigset have SIGUSR1\n");

    if((sigismember(set,SIGUSR2))==1)
        printf("sigset have SIGUSR2\n");

    if((sigismember(set,SIGINT))==1)
        printf("sigset have SIGINIT\n");

    free(set);
    return(0);
}
```

编译、运行：

```
$ gcc sigset_member.c -o sigset_member
$ ./sigset_member
sigset have SIGUSR1
sigset have SIGINIT
```

程序先初始化信号集，清除其中所有信号，然后把 SIGUSR1 和 SIGINT 添加到信号集中，然后测试 SIGUSR1，SIGUSR2，SIGINT 信号是否在信号集中。因为 SIGUSR2 不在信号集中，所以程序并不打印 SIGUSR2。

八**.4.2** 设置信号屏蔽字

函数 sigprocmask 可以设置或更改信号屏蔽字。

```
#include <signal.h>
int sigprocmask(int how, const sigsett_t *set, sigset_t *oset);
返回值：成功返回 0；出错返回-1
```

参数：

（1）how 对信号屏蔽字的处理方法 SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK；

（2）set 信号集，如果是空指针则不改变信号屏蔽字；（3）oset 保存当前进程的阻塞信号集。

使用之前要先设置好信号集合 set。这个函数的作用是将指定的信号集合 set 加入到进程的信号阻塞集合之中去，如果提供了 oldset 那么当前的进程信号阻塞集合将会保存在 oldset 里面。

参数 how 决定函数的操作方式：

（1）SIG_BLOCK：增加一个信号集合到当前进程的阻塞集合之中；

（2）SIG_UNBLOCK：从当前的阻塞集合之中删除一个信号集合；

（3）SIG_SETMASK：将当前的信号集合设置为信号阻塞集合。

例 **75**：使用 **sigprocmask** 函数屏蔽信号的例子

```
/* sigset_procmask.c */

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

void sig_act(int signo)
{
   printf("catch signal %d\n", signo);
}

int main(int argc, char *argv[])
{
   sigset_t set;

   struct sigaction action;
   action.sa_handler = sig_act;
   action.sa_mask = set;
   action.sa_flags = 0;

   sigaction(SIGINT, &action, NULL);

   sigemptyset(&set);
   sigaddset(&set,SIGINT);
   sigprocmask(SIG_SETMASK, &set, NULL);
   printf("SIGINI is blocked\n");
   sleep(10);

   sigprocmask(SIG_UNBLOCK, &set, NULL);
   printf("SIGINT is unblocked\n");
   sleep(10);
   return(0);
}
```

编译、运行：

```
$ gcc sigset_procmask.c -o sigset_procmask
```

```
$ ./sigset_procmask
SIGINI is blocked
SIGINT is unblocked
catch sigint signal
```

程序先定义信号集 set，然后把信号 SIGINT 添加到 set 信号集中，最后把 set 设置为信号阻塞集合。当我们运行程序时，在前 10 秒内，按住 ctrl+c 没有任何响应；10 秒后解除阻赛，按下 ctrl+c 可以接收到 SIGINT 信号并处理。

八**.4.3** 返回阻塞信号集

sigpending 函数

```
#include <signal.h>
int sigpending(sigset_t *set);
返回值：成功返回 0，若出错返回-1。
```

阻塞信号并不是丢弃信号，它们被保存在一个进程的信号阻塞队列里，sigpending 可以获得当前已递送到进程，却被阻塞的所有信号，在 set 指向的信号集中返回这些信号。

例 **76**：使用 **sigpending** 函数阻塞信号的例子

```
/* sigset_pending.c */

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

int main(int argc, char *argv[])
{
   sigset_t newmask, oldmask, pendmask;

   sigemptyset(&newmask);
   sigaddset(&newmask,SIGINT);

   printf("block SIGINIT signal\n");
   sigprocmask(SIG_BLOCK, &newmask, &oldmask);

   sleep(5);

   sigpending(&pendmask);
   if(sigismember(&pendmask,SIGINT)){
      printf("sigset have SIGINT pending\n");;
   }
```

```
    return(0);
}
```

编译、运行：

```
$ gcc sigset_pending.c -o sigset_pending
$ ./sigset_pending
block SIGINIT signal
sigset have SIGINT pending
```

本章小结

本章介绍了 Linux 信号的基础知识，信号的种类：可靠信号与不可靠信号，实时信号与非实时信号；信号的安装、信号的发送、信号屏蔽字等。

习题

8.1 写代码实现打印用户输入的信号。
8.2 写代码用信号实现每五分钟取当前的日期和时间，并将日期时间的具体信息写入文件。

第九章 LINUX LIB 库

九.1 Linux Lib 概述

所谓的库就是可以被多个软件项目使用或重用的代码集。库是软件开发所追求的目标，把经常使用的编程例程集中在一起。系统 C 语言库就是一个例子，它包含了数百个经常使用的例程，比如输出函数 printf 和输入函数 scanf 函数等。如果每次编程都重写这些函数耗时耗力。

库有两个优点：（1）能够实现代码重用；（2）提供了数百个经过测试和调试的工具代码，方便编程人员使用。

九.1.1 库命名和编号约定

库的命名约定

（1）所有的库名都以 lib 开头，开发工具依赖这个约定，GCC 会在以-l 选项所指向的文件名前自动插入 lib；

（2）静态库是以.a（代表存档文件 archive）结尾的库；

（3）动态库是以.so（代表共享目标文件 shared object）结尾的库；

（4）调试库是以名字以_g 结尾的库，编入了特殊的符号和功能，能够增加对采用这个库的应用程序进行调试的能力。

库的编号约定

一般格式为：library_name.major_num.minor_num.patch_num。

例子：libqte.so.2.3.7，library_name 是 libqte.so，major_num 是 2，minor_num 是 3，patch_num 是 7。

通常约定当库的变化不能和以前的版本兼容时就要增加主版本号（major_num）；当库有了新变化并能兼容以前的版本时只修改次版本号（minor_num）；为修订库汇中的错误而进行改动时修改补丁级别号（patch_num）。

九.1.2 常用库

下表列出了常用库，声明他们公共接口的头文件以及对他们提供功能的简化要描述。

库名	头文件	描述
libc		实现标准 C 库，不需要头文件
libm.so	<math.h>	标准 C 数学库
libcurses.so	<curses.h>	光标字符模式的屏幕操作库
libdb.so	<db.h>	创建和操作数据库的库
libdl.so	<dlfcn.h>	让程序在运行时加载和使用库代码而无须在编译时链接库

libglib.so	<glib.h>	Glib 库，提供了大多数程序需要的基本工具函数
libjpeg.so	<jpeglib.h>	定义到 jpeg 的接口，赋予应用程序适合用 jpeg 图形文件的能力
libgthread.so	<pthread.h>	实现 posix 线程库，标准的 Linux 多线程库
libz.so	<zlib.h>	通用压缩例程库

九.2 库操作工具

九.2.1 nm 命令

命令 nm 列出编入目标文件或二进制文件的所有符号，可以用来查看程序调用了哪些函数，也可以用来查看一个给定的库或目标文件是否提供了所需要的函数。

nm 命令语法：nm [options] file

nm 命令选项

选项	描述
-c	将符号名转换为用户级的名字
-s	当用于存档（.a）文件时，输出把符号名映射为定义该符号的模块或成员名的索引
-u	只显示未定义的符号
-I	使用调试信息输出定义每个符号的行号

九.2.2 ar 命令

ar 命令用来操作高度结构化的存档（archive）文件，该命令最常用来创建静态库—包含一个或多个目标文件。Ar 也能创建和维护符号名的交叉索引表。

ar 命令的语法：ar {dmpqrtx} [option] archive files

Ar 命令的选项

选项	描述
-c	如果存档不存在，则从多个文件创建存档文件
-s	创建后升级从符号到定义他们的成员之间的交叉索引映射表
-r	向存档文件插入文件，替换已有的任何同名成员，新成员添加到末尾
-q	把文件添加到末尾而不检查是否进行替换

对于用 ar 命名创建的存档文件，可以通过使用索引来加快访问它的速度。工具 ranlib 能够精确地完成此项工作，它把存档文件索引保存在存档文件本身里。
Ranlib 命令的语法：ranlib [-v | -V] file 等价于 ar –s file

九.2.3 ldd 命令

ldd 命令列出程序正常运行所需要的共享库。

ldd 命令的语法：ldd [option] file

ldd 输出 file 所要求的共享库的名字

例 **77**：使用 **ldd** 命令的例子

```
/* hello.c */

#include <stdio.h>
int main(int argc, char *argv[])
{
   printf(“hello\n”);
   return(0);
}
```

编译：

```
$ gcc hello.c –o hello
$ ldd hello
linux-gate.so.1 =>  (0x003e0000)
libc.so.6 => /lib/libc.so.6 (0x41dec000)
/lib/ld-linux.so.2 (0x4141d000)
```

ldd hello 列出了运行 hello 程序所需要的库。

九.2.4 ldconfg 命令

ldconfig 决定位于目录/usr/lib 和/lib 下共享库所需要的运行的链接，这些链接在命令行上的 libs 指定并保存在/etc/ld.so.conf 中。ldconfig 命令和动态链接/装载工具 ld.so 协同工作，一起来创建和维护对最新版本共享库的链接。

ldconfig 命名的语法：ldconfig [option] [libs]

ldconfig 命名选项

选项	描述
-p	仅打印文件/etc/ld.so.cache 内容，此文件是 ld.so 知道的共享库的当前列表
-v	更新/etc/ld.so.cache 内容，列出库版本号、扫描目录、创建和更新的链接

九.2.5 环境变量和配置文件

动态链接/装载工具 ld.so 使用两个环境变量：
（1）第一个是$LD_LIBRARY_PATH，由冒号分割的目录清单，在这些目录下可搜索运行时的共享库，如果程序运行时需要用到没有保存在标准位置的库，可把需要用的库的路径添加到 LD_LIBRARY_PATH，告诉 ld.so 在哪儿可以找到你的库。
（2）第二个变量是$LD_PRELOAD，由空格分隔的、附加的、用户指定的共享库，它需要在其他所有库加载前加载。

ld.so 还用到两个配置文件：除了标准目录/usr/lib 和 /lib 外，/etc/ld.so.conf 文件列出了链接器/加载器搜索共享库时要看的目录， /etc/ld.so.proload 文件是环境变量$LD_PRELOAD 的基于磁盘的版本，它包含了一个在执行程序之前要加载的由空格分隔的共享库列表。

九.3 静态库的制作

静态库是包含了目标文件的文件，这些目标文件被称为模块或成员，是可以充要的预编译代码。为了使用库代码，必须在源代码文件中包含适当的头文件并链接到库。

例 **78**：编写并使用静态库的例子

（1）编辑文件

头文件 mylib.h

```
#ifndef MYLIB_H
#define MYLIB_H

#ifdef __cplusplus
extern "C"{
#endif

int my_min(int a, int b);
int my_max(int a, int b);

#ifdef __cplusplus
}
#endif

#endif
```

实现文件 libmylib.c

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

int my_min(int a, int b)
```

```
{
   return( (a>b)?b:a );
}

int my_max(int a, int b)
{
   return( (a>b)?a:b );
}
```

测试文件 test_static_lib.c

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

int main(int argc, char *argv[])
{
   int a = 3;
   int b = 5;
   int c = 0;
   int d = 0;

   c = my_min(a, b);
   printf("my_min(%d, %d) = %d\n", a, b, c);

   d = my_max(a, b);
   printf("my_max(%d, %d) = %d\n", a, b, d);

   return(0);
}
```

（2）把代码编译成目标文件形式

```
$ gcc -c libmylib.c -o libmylib.o –I.
```

（3）使用工具 ar 创建归档文件

```
$ ar rcs libmylib.a libmylib.o
```

（4）编译、测试

```
$ gcc test_static_lib.c -o test_static_lib -static –I. -L. -lmylib;     编译
$ ./test_static_lib                                                     运行
```

九.4 动态库的制作

共享库和静态库相比有几个优点：
（1）共享库占用的系统资源少，共享库没有编译进二进制文件中，只是在运行时从共享库中链接加载，因此占用较少的磁盘空间；
（2）共享库比静态库快，因为他们只需要向内存里加载一次；
（3）共享库使得代码维护工作大大简化，当修正了一些错误或添加新特性时，用户只要获得升级后的库安装即可，而使用静态库链接的程序每次修订需要重新编译链接到新库。

在运行时动态链接/装载工具 ld.so 把二进制文件中的符号名链接到适当的共享库上，共享库由库名和主版本号组成，应用程序链接到 so 名字，ldconfig 工具创建一个从实际库到 so 名字的符号链接，ldconfig 创建了从库文件到 so 名字的符号链接，并把这些信息保存在缓冲文件/etc/ld.so.cache 中，在运行时，ld.so 读取该缓冲，找到所需的 so 名字，把实际的库加载到内存中，并把应用程序中的函数调用链接到加载库中合适的符号上。

例 **79**：编写并使用动态库的例子

（1）编辑文件
同静态库文件类似

头文件 mylib.h

```
#ifndef MYLIB_H
#define MYLIB_H

#ifdef __cplusplus
extern "C"{
#endif

int my_min(int a, int b);
int my_max(int a, int b);

#ifdef __cplusplus
}
#endif

#endif
```

实现文件 libmylib.c

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

int my_min(int a, int b)
```

```
{
   return( (a>b)?b:a );
}

int my_max(int a, int b)
{
   return( (a>b)?a:b );
}
```

测试文件 test_dynamic_lib.c

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

int main(int argc, char *argv[])
{
   int a = 3;
   int b = 5;
   int c = 0;
   int d = 0;

   c = my_min(a, b);
   printf("my_min(%d, %d) = %d\n", a, b, c);

   d = my_max(a, b);
   printf("my_max(%d, %d) = %d\n", a, b, d);

   return(0);
}
```

（2）编译目标文件

```
$ gcc -fPIC -g -c libmylib.c -o libmylib.o -I.
```

使用 gcc 的 -fPIC 选项，产生与位置无关的代码并能被加载到任何地址

（3）链接库

```
$ gcc -g -shared -Wl,-soname, -o libmylib.so.1 libmylib.o
```

（4）建立符号连接

```
$ ln -s libmylib.so.1 libmylib.so
```

如果不想把该库作为系统库安装到/usr/lib 和/lib 上，必须建立符号链接

（5）编译、运行
可以将编译好的动态库文件放入/usr/lib 或 /lib 目录下，也可以放入任意目录，但该目录必须导入到环境变量 LD_LIBRARY_PATH 中。
比如当前 mylib 库文件位于/home/embedded/lib 目录下。

$ gcc test_dynamic_lib.c -o test_dynamic_lib -I. -L. –lmylib	编译
$ export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH	导出环境变量
$ ./test_dynamic_lib	运行

静态库和动态库
从刚才制作的静态库和动态库的库文件，可以比较静态库和动态库的区别：

静态库

$ file test_static_lib	查看可执行文件属性
$ du –sh test_static_lib	查看可执行文件大小
$ ldd test_static_lib	查看可执行文件动态库

动态库

$ file test_dynamic_lib	查看可执行文件属性
$ du –sh test_dynamic_lib	查看可执行文件大小
$ ldd test_dynamic_lib	查看可执行文件动态库

用户可将自定义库的头文件放入/usr/include 目录下，将自定义的静态库或动态库放入/usr/lib 目录下。否则用户在使用编译器 GCC 编译自己的应用时要使用-I 指明头文件的搜索路径，使用-L 指明库的搜索路径。

本章小结

本章介绍了 Linux lib 的基本知识，介绍了静态库和动态库的制作，以及如何使用制作的静态库和动态库。

习题

9.1 封装 5 个以上的自定义的有用的函数，分别制作动态库和静态库以方便自己编程使用。

第十章 LINUX 网络编程

十.1 Socket 概述

Linux 系统中网络编程是通过 Socket 接口进行的，Socket 接口是一种特殊的 IO，也是一种文件描述符。Socket 提供了进程通信的端点，每一个 Socket 都用一个半相关描述：{协议、本地地址、本地端口}。一个完整的 Socket 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}。Socket 类似于打开文件的文件描述符，连接、数据传输等都是通过 Socket 实现的。

Socket 有三种类型：流式套接字（SOCK_STREAM）、数据报套接字（SOCK_DGRAM）及原始套接字（SOCK_RAM）。

（1）流式的套接字提供了一种可靠的、面向连接的服务，使用 TCP 协议。

（2）数据报套接字提供了一种不可靠的、无连接的服务，使用 UDP 协议。数据通过相互独立的报文进行传输，是无顺序的，并且不保证可靠，无差错。

（3）原始套接字主要用于一些协议的开发，可以进行比较底层的操作。功能强大，但没有上面介绍的两种套接字使用方便，一般的程序也涉及不到原始套接字。

Linux 基于 Socket 的网络编程主要有两种工作流程:

（1）面向流连接的 Socket 工作流程;

（2）面向无流连接的 Socket 工作流程。

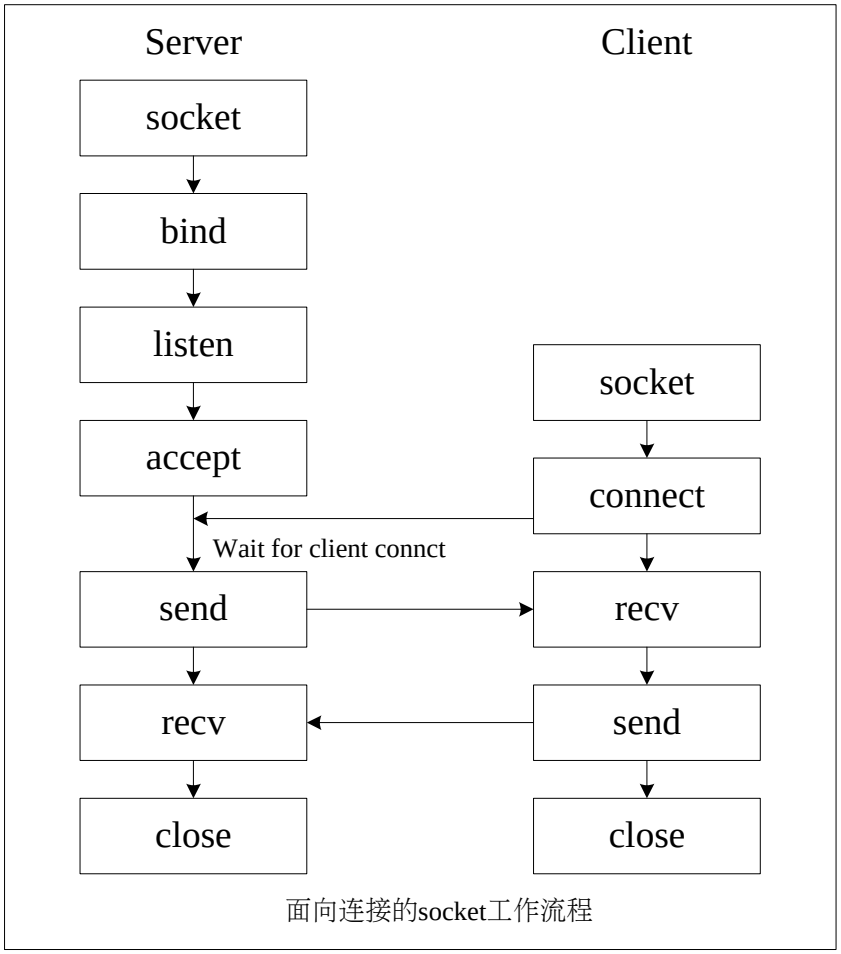

图 **7**：面向流连接的 **Socket** 工作流程

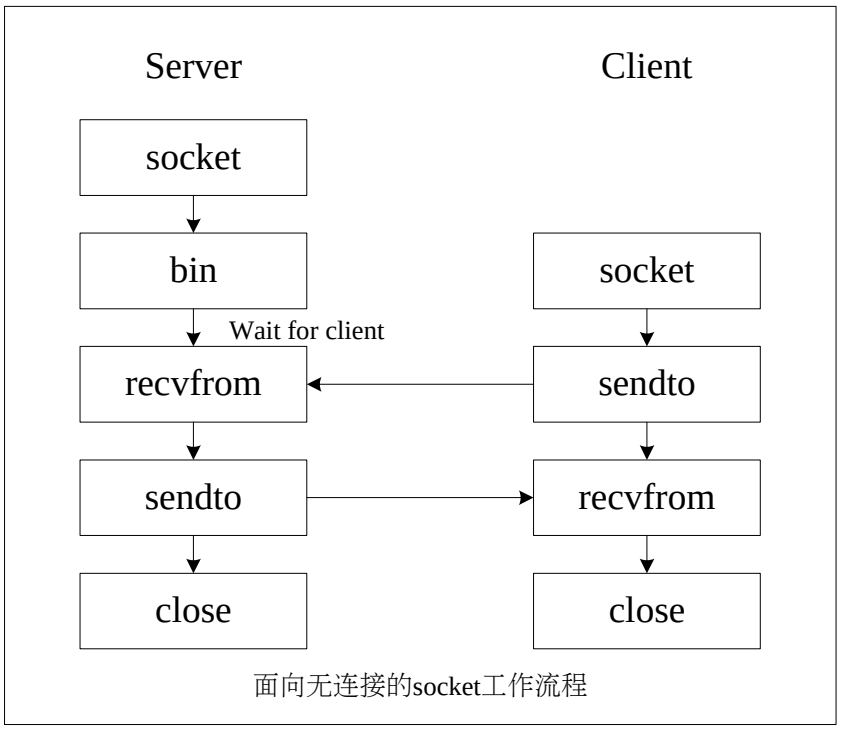

图 **8**：面向无流连接的 **Socket** 工作流程

十.2 Socket 基本数据结构

十.2.1 struct sockaddr

用来保存 socket 信息

```
struct sockaddr
{
   unsigned short sa_family;                地址族   2 字节
   char sa_data[14];                            协议地址      14 字节
};
```

（1）sa_family：AFINET、AFINET6；
（2）sa_data：协议地址，包括 socket 的 IP 地址和端口号。

十.2.2 struct in_addr

用来保存 IP 地址信息

```
struct in_addr
{
   unsigned long s_addr;                    IP 地址   4 字节
};
```

十.2.3 struct sockaddr_in

用来保存 socket 信息，in 代表 Internet。

```
struct sockaddr_in
{
   short int sin_family;                    地址族   2 字节
   unsigned short int sin_port;                 端口号   2 字节
   struct in_addr sin_addr;                     IP 地址   4 字节
   unsigned char sin_zero[8];               填充 0    8 字节保持与 sockaddr 同样大小
};
```

struct sockaddr 和 struct sockaddr_in 大小相等，都是 16 字节，是等价的，可以相互转化。

十.2.4 struct hostent

保存主机信息

```
struct hostent
{
   char  *h_name;                           主机名
   char  **h_aliases;                           主机别名
   int    h_addrtype;                       地址类型
```

```
    int    h_length;                    地址长度
    char  **h_addr_list;                指向 IPv4/6 的地址指针数组
};
```

十.3 Socket 函数接口

十.3.1 转换函数

介绍一下机器的大小端、主机字节顺序、网络字节顺序及它们之间的转换关系。

（1）大小端
小端：数据的最低有效字节 LSB（Least Significant Byte）存储在内存的最低地址；
大端：数据的最高有效字节 MSB（Most Significant Byte）存储在内存的最低地址。

也就是说 LSB 和 MSB 谁位于内存的最低地址，决定了机器的小端（little-endian）和大端（big-endian）。如果 LSB 在 MSB 前面，即 LSB 是低地址，则该机器的数据存储方式是小端；反之 MSB 是低地址，则该机器的数据存储方式是大端。

例如 32bit 的数据 0x12345678 在小端（little-endian）模式内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12

而在大端（big-endian）模式内存中的存放方式则为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

（2）主机字节顺序（Host Byte Order）
计算机数据的存储方式有两种字节优先顺序：小端（little-endian）低位字节优先和大端（big-endian）高位字节优先。不同的机器主机字节顺序是不相同，比如 Intel 处理器是小端字节顺序。

（3）网络字节顺序（Network Byte Order）
按从高到低的顺序存储，在网络上使用统一的网络字节顺序，可以避免兼容性问题。网络字节顺序是大端，也就是先传整数的高位字节，在传整数的低位字节。

Internet 上数据以高位字节优先顺序在网络上传输，所以对于在内部是以低位字节优先方式存储数据的机器，在 Internet 上传输数据时就需要进行转换。

主机字节顺序和网络字节顺序的转换

```
#include <arpa/inet.h>
```

unsigned long htonl(unsigned long);	主机字节顺序转换为网络字节顺序
unsigned short htons(unsigned short);	主机字节顺序转换为网络字节顺序
unsigned long ntohl(unsigned long);	网络字节顺序转换为主机字节顺序
unsigned short ntohs(unsigned short);	网络字节顺序转换为主机字节顺序

函数 inet_aton、inet_addr 和 inet_ntoa 在网络地址字符串（“192.168.1.10”）与 32 位网络字节二进制之前进行 IPV4 地址转换。

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
unsigned long inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
```

inet_aton 函数将网络 IP 地址字符串（192.168.1.10) 转换为网络字节顺序二进制值，并存储在 struct in_addr 结构中。函数返回非 0 表示 cp 主机有地有效，返回 0 表示主机地址无效。

inet_addr 函数将网络 IP 地址字符串（192.168.1.10)转换为网络字节顺序二进制值。 如果参数 char *cp 无效，函数返回-1(INADDR_NONE)。

inet_ntoa 函数将网络字节顺序的结构体 struct in_addr 转换为网络 IP 地址字符串（192.168.1.10)。该函数返回指向点分开的字符串地址的指针。

例 **80**：大小端主机字节网络字节转换的例子

```
/* hostNetByte.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
   int la = 0x12345678;
   char *p = (char *)&la;
   printf("little endian: %x %x %x %x\n", *p, *(p+1), *(p+2), *(p+3));   // 78 56 34 12

   int ba = htonl(la);
   p = (char *)&ba;
```

```
    printf("big endian:    %x %x %x %x\n", *p, *(p+1), *(p+2), *(p+3));  // 12 34 56 78

    char charIp[20];
    printf("please input ip(a.b.c.d): ");
    scanf("%s",charIp);                              输入 192.168.1.1
    printf("\nIp Address: %s\n", charIp);

    int a,b,c,d;
    sscanf(charIp, "%d.%d.%d.%d", &a, &b, &c, &d);
    printf("Get a b c d from Ip: %d %d %d %d\n", a, b, c, d);       // 192 168 1 1
    unsigned hostbyte = (a<<24) | (b<<16) | (c<<8) | d;
    printf("Ip hostByte: 0x%x, %u\n", hostbyte, hostbyte);         // 0xc0a80101

    unsigned long netIp = inet_addr(charIp);
    printf("Ip netByte:  0x%x, %u\n", netIp, netIp);               // 0x101a8c0

    unsigned long hostIp = ntohl(netIp);                        // 0xc0a80101
    printf("Ip hostByte: 0x%x, %u\n", hostIp, hostIp);

    struct in_addr ina;
    inet_aton(charIp, &ina);
    printf("charIp  -> in_addr : %s -> %u\n", charIp, ina.s_addr);
    printf("in_addr -> charIp  : %u -> %s\n", ina.s_addr, inet_ntoa(ina));

    return(0);
}
```

编译、运行:

```
$ gcc hostNetByte.c –o hostNetByte
$ ./hostNetByte
little endian: 78 56 34 12
big endian:    12 34 56 78
please input ip(a.b.c.d): 192.168.1.1                                  输入 IP 地址

Ip Address: 192.168.1.1
Get a b c d from Ip: 192 168 1 1
Ip hostByte: 0xc0a80101, 3232235777
Ip netByte:  0x101a8c0, 16885952
Ip hostByte: 0xc0a80101, 3232235777
charIp  -> in_addr : 192.168.1.1 -> 16885952
in_addr -> charIp  : 16885952 -> 192.168.1.1
```

本例介绍了大小端、主机字节和网络字节顺序及其相互转换。

十.3.2 socket 函数

创建套接字描述符

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain , int type , int protocol）
返回值：成功返回套接字描述符；错误返回-1
```

参数:

（1）domain 域 AF_INET（IPv4）和 AF_INET6（IPv6）；

（2）type 发送类型 SOCK_STREAM、SOCK_DGRAM 和 SOCK_RAM;

（3）protocol 一般为 0。

十.3.3 bind 函数

指定一个套接字使用的端口

```
#include <sys/types.h>
#include <sys/socket.h>
int bind (int sockfd , struct sockaddr *my_addr , int addrlen);
返回值：成功返回 0；错误返回-1
```

参数:

（1）sockfd 套接字描述符；　　（2）my_addr 本地地址;

（3）addrlen 地址长度，一般为 sizeof(struct sockaddr)。

十.3.4 connect 函数

连接到套接字指定的服务器端口

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, struct sockaddr *serv_addr, int addrlen);
返回值：成功返回 0；错误返回-1
```

参数:

（1）sockfd 套接字描述符；　　（2）serv_addr 服务器端口地址;

（3）addrlen 地址长度，一般为 sizeof(struct sockaddr)。

十.3.5 listen 函数

监听、等待其它连接

```
#include <sys/socket.h>
int listen (int sockfd, int backlog);
```

返回值：成功返回 0；错误返回-1

参数：（1）sockfd 套接字描述符；（2）backlog 连接请求队列可以容纳的最大数目。

十.3.6 accept 函数

接受连接

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *addr, int *addrlen);
返回值：成功返回新连接的描述符；错误返回-1
```

参数：

（1）sockfd 套接字描述符；　　（2）addr 客户端地址；

（3）addrlen 地址长度，一般为 sizeof(struct sockaddr_in)；

十.3.7 send 函数

用于 TCP 连接的数据发送

```
#include <sys/types.h>
#include <sys/socket.h>
int send (int sockfd, const void *msg, int len, int flags);
返回值：成功返回发送的字节数；错误返回-1
```

参数：

（1）sockfd 套接字描述符；　　（2）msg 发送数据的指针；

（3）len 发送数据的长度；　　（4）flags 发送标记，一般都设为 0。

十.3.8 recv 函数

用于 TCP 连接的数据接收

```
#include <sys/types.h>
#include <sys/socket.h>
int recv (int sockfd, void *buf, int len, unsigned int flags）;
返回值：成功返回接收的字节数；错误返回-1
```

参数：

（1）sockfd 套接字描述符；　　（2）buf 存储数据的指针；

（3）len 缓存区的最大尺寸；　　（4）flags 接收标记，一般都为 0。

十.3.9 sendto 函数

用于 UDP 连接的数据发送

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

```
#include <sys/socket.h>
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
返回值：成功返回真正发送的字节数；错误返回-1
```

参数：

（1）sockfd 套接字描述符；　（2）msg 发送数据的指针；

（3）len 发送数据的长度；　（4）flags 发送标记，一般为 0；

（5）to 目的端 IP 地址和端口；　（6）tolen 地址长度，一般为 sizeof(struct sockaddr)。

十.3.10 recvfrom 函数

用于 UDP 连接的数据接收

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom (int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);
返回值：成功返回真正接收的字节数；错误返回-1
```

参数：

（1）sockfd 套接字描述符；　（2）buf 存储数据的内存缓存区域；

（3）len 缓存区的最大长度；　（4）flags 接收标记，一般为 0；

（5）from 源端 IP 地址和端口；　（6）fromlen 地址长度，一般为 sizeof (struct sockaddr)。

十.3.11 close 函数

关闭套接字

```
int close (sockfd);
```

执行 close()之后，套接字将不会在允许进行读操作和写操作。任何有关对套接字描述符进行读和写的操作都会接收到一个错误。

十.3.12 gethostname 函数

取得本地主机的信息

```
#include <unistd.h>
int gethostname(char *hostname, size_t size);
返回值：成功返回 0；错误返回-1
```

参数：

（1）hostname 指向字符数组的指针，数据就是本地的主机的名字；

（2）size 是 hostname 指向的数组的长度。

它返回正在执行它的计算机的名字。返回的这个名字可以被 gethostbyname()函数使用，由此可以得到本地主机的 IP 地址。

十.3.13 gethostbyname 函数

由主机名得到主机信息

```
#include <netdb.h>
struct hostent *gethostbyname (const char *hostname);
返回值：成功返回指向主机信息；错误返回-1
```

以上介绍了 Linux 网络编程需要用到的函数原型，Linux 新版本的函数原型有些做了改变，比如参数 int len 变成了 size_t len，返回值由 int 变成了 ssize_t 等。各位在使用时使用 man 查询最新的函数原型。

十.4 实例

例 **81**：套接字 **Server/Client** 实现的例子

Server 端

```
/* server_net_tcp.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <libgen.h>
#include <getopt.h>
#include <unistd.h>
#include <pthread.h>
#include <syslog.h>
#include <resolv.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <unistd.h>

void *handle_request(void *arg)
```

```
{
    int sock_fd = *(int *)arg;

    char buf[1024] = {0, };
    if (recv(sock_fd, buf, 1024, 0) == -1){
        perror("recv");
        exit(1);
    } else {
        printf("Receive -- %s\n", buf);
    }

    memset(buf, 0, 1024);
    strcpy(buf, "ACK From Server");
    if (send(sock_fd, buf, strlen(buf), 0) == -1){
        perror("send");
        exit(1);
    } else {
        printf("Send -- %s\n", buf);
    }

    close(sock_fd);
    return(NULL);
}

int main(int argc, char *argv[])
{
    if (argc < 3){
        fprintf(stderr, "Usaging: ./server ip port\n");
        exit(1);
    }

    char *server_host = argv[1];
    int  server_port = atoi(argv[2]);
    //char *server_host = NULL;
    //int server_port = 5999;

    // Create Socket
    int sockfd;
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "[Error] -- gethostbyname Failed : %s", strerror(errno));
        exit(1);
    }
    int on = 1;
```

```
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

// Fill with Server Info
struct sockaddr_in server_addr;
bzero(&server_addr, sizeof(server_addr));
server_addr.sin_family = AF_INET;
if (server_host == NULL){
   server_addr.sin_addr.s_addr = INADDR_ANY;
} else {
   unsigned long inaddr = 0x0l;
   if ((inaddr = inet_addr(server_host)) == INADDR_NONE) {
      struct hostent *host;
      if ((host = gethostbyname(server_host)) == NULL){
         fprintf(stderr, "[Error] -- gethostbyname Failed : %s", strerror(errno));
         close(sockfd);
         return(-1);
      }
      server_addr.sin_addr = *((struct in_addr *)host->h_addr);
   } else {
      inet_aton(server_host, &(server_addr.sin_addr));
   }
}
server_addr.sin_port = htons(server_port);

// Bind Socket
if (bind(sockfd, (struct sockaddr *) &server_addr, sizeof(struct sockaddr)) == -1) {
   fprintf(stderr, "[Error] -- bind Failed : %s", strerror(errno));
   exit(1);
}

// Listen Socket Connection
int listen_num = 100;
if (listen(sockfd, listen_num) == -1) {
   fprintf(stderr, "[Error] -- listen Failed : %s", strerror(errno));
   exit(1);
}

// Accept Client Connection and spawn new thread to process it
struct sockaddr_in client_addr;
socklen_t len = sizeof(struct sockaddr);
int ret;
while (1)
{
```

```
        int new_fd;
        if ((new_fd = accept(sockfd, (struct sockaddr *) &client_addr, &len)) == -1) {
            fprintf(stderr, "[Error] -- accept Failed : %s", strerror(errno));
            exit(errno);
        } else {
                                    printf("\nGot Connection from %s, port %d, socket %d\n",
inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port), new_fd);
        }
        pthread_t tid;
        ret = pthread_create(&tid, NULL, handle_request, (void *)&new_fd);
        if (ret){
            fprintf(stderr, "[Error] -- pthread_create Failed : %s", strerror(errno));
            break;
        }
    }

    // Close Socket fd
    close(sockfd);

    return(0);
}
```

服务器端 IP 地址和端口通过参数传递，也可以不指定 IP 地址，默认使用本机 IP 地址。

Client 端

```
/* client_net_tcp.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <libgen.h>
#include <getopt.h>
#include <unistd.h>
#include <pthread.h>
#include <syslog.h>
#include <resolv.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

```
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
   if (argc < 3){
      fprintf(stderr, "Usaging: ./client ip port\n");
      exit(1);
   }

   char *server_host = argv[1];
   int  server_port = atoi(argv[2]);
   //char *server_host = NULL;
   //int server_port = 5999;

   // Create Socket for monitor 5999 port
   int sockfd;
   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      fprintf(stderr, "[Error] -- gethostbyname Failed : %s", strerror(errno));
      exit(1);
   }
   int on = 1;
   setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

   // Fill with Server Info
   struct sockaddr_in server_addr;
   bzero(&server_addr, sizeof(server_addr));
   server_addr.sin_family = AF_INET;
   if (server_host == NULL){
      server_addr.sin_addr.s_addr = INADDR_ANY;
   } else {
      unsigned long inaddr = 0x0l;
      if ((inaddr = inet_addr(server_host)) == INADDR_NONE) {
         struct hostent *host;
         if ((host = gethostbyname(server_host)) == NULL){
            fprintf(stderr, "[Error] -- gethostbyname Failed : %s", strerror(errno));
            close(sockfd);
            return(-1);
         }
         server_addr.sin_addr = *((struct in_addr *)host->h_addr);
      } else {
         inet_aton(server_host, &(server_addr.sin_addr));
```

```
        }
    }
    server_addr.sin_port = htons(server_port);

    // connect
    if ((connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr))) == -1){
        perror("connect");
        exit(1);
    }

    char buf[1024] = {0,};
    strcpy(buf, "REQ from client");

    // send
    if (send(sockfd, buf, strlen(buf), 0) == -1){
        perror("send");
        exit(1);
    } else {
        printf("Send -- %s\n", buf);
    }

    // receive
    memset(buf, 0, 1024);
    if (recv(sockfd, buf, 1024, 0) == -1){
        perror("recv");
        exit(1);
    } else {
        printf("Receive -- %s\n", buf);
    }

    // Close Socket fd
    close(sockfd);

    return(0);
}
```

编译、运行:

```
# gcc server_net_tcp.c -o server_net_tcp –lpthread
# ./server_net_tcp 192.168.1.21 5998
Got Connection from 172.16.5.10, port 55603, socket 4
Receive -- REQ from client
Send -- ACK From Server
```

在另一台机器上或者本机的另一个 Shell 终端

```
# gcc client_net_tcp.c -o client_net_tcp
# ./client_net_tcp 192.168.1.21 5998
Send -- REQ from client
Receive -- ACK From Server
```

在一个 Shell 终端运行 server_net_tcp 服务程序，在另一个 Shell 终端运行 client_net_tcp 客户程序，测试时也可以输入用户指定数据。

例 **82**：数据报套接字的 **Server/Client** 实现的例子

Server 端

```
/* server_net_udp.c */

#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 PORT 5000
#define MAXBUF 1024

int main(int argc, char *argv[])
{
   int sock_fd;
   int numbytes;
   int addr_len;
   char receive_buf[MAXBUF];
   char send_buf[MAXBUF] = "hello ";
   struct sockaddr_in server_addr;
   struct sockaddr_in client_addr;

   // Create socket
   if( (sock_fd=socket(AF_INET, SOCK_DGRAM, 0)) == -1){
      perror("socket");
      exit(1);
   }
```

```
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(server_addr.sin_zero), 8);

    // Bind
    if( (bind(sock_fd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr))) == -1){
        perror("bind");
        exit(1);
    }
    while(1)
    {
        recv data
        bzero(receive_buf, MAXBUF);
        addr_len = sizeof(struct sockaddr);
        if( (numbytes=recvfrom(sock_fd, receive_buf, MAXBUF, 0,(struct sockaddr *)&client_addr,
&addr_len))==-1){
            continue;
        }
        receive_buf[numbytes]='\0';
        printf("Server Got Connection From %s\n", inet_ntoa(client_addr.sin_addr));
        printf("Receive Content: %s\n", receive_buf);

        // Send date
        strcpy(send_buf+6, receive_buf);
            if ( sendto(sock_fd, send_buf, strlen(send_buf), 0,(struct sockaddr *)&client_addr,
sizeof(struct sockaddr)) == -1){
            perror("send");
            exit(1);
        }
        printf("Send Content: %s\n", send_buf);
    }

    // Close socket
    close(sock_fd);

    return(0);
}
```

Client 端

```
/* client_net_udp.c */

#include <stdio.h>
```

```
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define PORT 5000
#define MAXBUF 1024

int main(int argc, char *argv[])
{
   int sock_fd;
   int numbytes;
   char send_buf[MAXBUF];
   char receive_buf[MAXBUF];
   struct sockaddr_in server_addr;
   struct hostent *host;

   if(argc != 2){
      fprintf(stderr, "Usage: command server_ip\n");
      exit(1);
   }

   if( (host=gethostbyname(argv[1])) == NULL){
      perror("gethostbyname");
      exit(1);
   }

   // Create socket
   if( (sock_fd=socket(AF_INET, SOCK_DGRAM, 0)) == -1){
      perror("socket");
      exit(1);
   }

   server_addr.sin_family = AF_INET;
   server_addr.sin_port = htons(PORT);
   server_addr.sin_addr = *((struct in_addr *)host->h_addr);
   bzero(&(server_addr.sin_zero), 8);

   // Send data
```

```
    printf("please input send data: ");
    gets(send_buf);
     if ( (numbytes=sendto(sock_fd, send_buf, strlen(send_buf), 0,(struct sockaddr *)&server_addr,
sizeof(struct sockaddr))) == -1){
        perror("recvfrom");
        exit(1);
    }
    // Recv data
    int addr_len = sizeof(struct sockaddr);
     if ((numbytes=recvfrom(sock_fd, receive_buf, MAXBUF, 0,(struct sockaddr *)&server_addr,
&addr_len))==-1){
        perror("recvfrom");
        exit(1);
    }
    receive_buf[numbytes]='\0';
    printf("Receive Content: %s\n", receive_buf);

    // Close socket
    close(sock_fd);
    return 0;
}
```

编译、运行：

```
# gcc server_net_udp.c -o server_net_udp
# ./server_net_udp
Server Got Connection From 192.168.1.21
Receive Content: china
Send Content: hello china
Server Got Connection From 192.168.1.21
Receive Content: shanghai
Send Content: hello shanghai
```

```
在另一台机器上或者本机的另一个 Shell 终端
# gcc client_net_udp.c -o client_net_udp
# ./client_net_udp 192.168.1.21
please input send data: china
Receive Content: hello china

# ./client_net_udp 192.168.1.21
please input send data: shanghai
Receive Content: hello shanghai
```

在一个 Shell 终端运行 server_net_udp 服务程序，在另一个 Shell 终端运行 client_net_udp 客

户程序，提示用户输入发送内容，第一次输入 china，可看到服务器端发来的信息 hello china；再次运行 client_net_tcp 客户程序，输入 shanghai，再次看到服务器端发来的信息 hello shanghai。服务器端可看到客户端两次连接发送的内容 china, shanghai。

例 **83**：网页服务程序的例子

Web Server 监听 80 端口，有网页请求，打印信息，并向网页客户端发送一个简单的页面。

Server 端

```
/* web_server_tcp.c */

#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 WEB_SERVER_PORT 80
#define MAXBUF 4096

struct remoteconnection
{
   int sock_fd;
   struct sockaddr_in remote_addr;
};

void *handler_web_request(void *arg)
{
   struct remoteconnection *rc = (struct remoteconnection *)arg;
   printf("Remote Connection From IP: %s\n", inet_ntoa(rc->remote_addr.sin_addr));
   char receive_buf[MAXBUF] = {0,};
   char send_buf[MAXBUF] = "<html><head><title>Test Web Page</title></head>
            <body><h1>This is a test web page</h1></body></html>";

   if (recv(rc->sock_fd, receive_buf, MAXBUF, 0) == -1){
      perror("sock recv:");
      return;
   }

   char reqtype[10], request[100], prot[20];
```

```
    sscanf(receive_buf, "%s%s%s", reqtype, request, prot);
    printf("reqtype = %s, request = %s, prot = %s\n", reqtype, request, prot);

    if (send(rc->sock_fd, send_buf, strlen(send_buf), 0) == -1){
        perror("sock send:");
        return;
    }
    close(rc->sock_fd);
}

int main(int argc, char *argv[])
{
    int sock_fd;//, new_fd ;
    struct sockaddr_in server_addr;
    int sin_size;

    // Create socket
    if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
        perror("socket");
        exit(1);
    }

    bzero(&server_addr, sizeof(struct sockaddr_in));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(WEB_SERVER_PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    // Bind
    if( bind(sock_fd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr) ) == -1){
        perror("bind");
        exit(1);
    }

    // Listen
    if ((listen(sock_fd, 10)) == -1){
        perror("listen");
        exit(1);
    }

    while(1)
    {
        struct remoteconnection rc;
        pthread_t tid;
```

```
        sin_size = sizeof(struct sockaddr_in);
        if ((rc.sock_fd = accept(sock_fd, (struct sockaddr *)&rc.remote_addr, &sin_size)) == -1){
            perror("accept");
            continue;
        }

        pthread_create(&tid, NULL, handler_web_request, (void *)&rc);
        sleep(1);
    }
    return(0);
}
```

编译、运行：

```
# gcc web_server_tcp.c -o web_server -lpthread
# ./web_server
Remote Connection From IP: 192.168.1.100
reqtype = GET, request = /, prot = HTTP/1.1
```

web_server 服务程序在运行时，打开浏览器输入服务器 IP 地址（http://192.168.1.21，默认端口 80，可以不用输入，也可以在程序里指定其他端口），就可以看到一个简单的测试页面。服务器端可以打印客户端信息。服务器可以根据页面请求的路径，返回不同的页面。

本章小结

本章介绍了 Linux 网络编程 Socket 的基本知识、Socket 基本数据结构、Socket 应用的函数接口，并以面向连接的 TCP 连接和面向无连接的 UDP 连接的实例说明 Linux 网络编程的基本应用。

习题

10.1 写一个程序判断当前 CPU 处理器的大小端。
10.2 写一个面向连接的 TCP 服务器端，可以处理连接到该服务器端的所有连接，并根据客户端发来的出生日期请求信息，发回对应的星座信息。

参考文献

1. W. Richard Stevens, Stephen A. Rago. Advanced Programming in the UNIX® Environment: Second Edition. Addison Wesley Professional. June 17, 2005
2. Kurt Wall 著、张辉译. GNU/Linux 编程指南. 清华大学出版社.2002.06
3. 尤晋元、张亚英、戚正伟. Unix 环境高级编程(第二版).人民邮电出版社.2006.05