第一篇　Linux 系统介绍

第1章　Linux简介

本章介绍Linux的起源、优缺点、硬件要求以及获得Linux的方式等内容。

1.1　Linux 的起源

应该说，Linux 起源于Internet，虽然最初的Linux 核心程序是由一名芬兰赫尔辛基的大学生 Linus Torvalds编写的。1990年，他还在读大学本科计算机专业的时候，因为不满学校的服务器一次只能接待16个人连网，于是“一气之下，我干脆自己掏钱买了一台 PC”，Linus回忆说。

Linus在自己的Intel 386 PC上，利用Tanenbaum教授自行设计的微型UNIX操作系统Minix为开发平台，开发了属于他自己的第一个程序。“这个程序包括两个进程，都是向屏幕上写字母，然后用一个定时器来切换这两个进程。”他回忆说，“一个进程写A，另一个进程写B，所以我就在屏幕上看到了AAAA、BBBB如此循环重复输出结果。”

到第二年，他完成了如今令他誉满全球的操作系统Linux 的最初内核，第三年，Linus把这一软件奉献给自由软件基金会（Free Software Foundation，简称 FSF）的GNU 计划，并公布了全部源代码，使得任何人都可以从网上下载、分析、修改、添加新功能，甚至出售赢利。现在，通过Internet，遍及世界各地的计算机高手把一个随时都有可能被遗弃的萌芽，扶植成了一个计算机领域中任何人都无法忽视的力量。不少专业人员认为 Linux 最安全、最稳定，对硬件系统最不敏感。Linux目前装机总数已超过600万台，分布于全世界。在当今金钱至上的商业社会，一个自由软件得到如此多的人的关心，不能不说是一个奇迹。

1.2　自由软件基金会的GNU计划

当前流行的软件按其提供方式可以划分为三种模式：商业软件 (Commercial software)、共享软件（Shareware）和自由软件（Freeware或Free software）。

商业软件由开发者出售拷贝并提供技术服务，用户只有使用权，但不得进行非法拷贝、扩散、修改或添加新功能；共享软件由开发者提供软件试用程序拷贝授权，用户在试用该程序拷贝一段时间之后，必须向开发者交纳使用费用，开发者则提供相应的升级和技术服务；而自由软件则由开发者提供软件全部源代码，任何用户都有权使用、拷贝、扩散、修改该软件，同时用户也有义务将自己修改过的程序源代码公开。

1984年，曾和Bill Gates 同为哈佛大学学生的 Richard Stallman组织开发了一个完全基于自由软件的软件体系计划——GNU，并拟定了一份普遍公共许可（General Public License，简称GPL）。Linux从产生到发展一直遵循的是“自由软件”的思想。GNU计划的宗旨是：消除对于计算机程序拷贝、分发、理解和修改的限制。也就是说，每一个人都可以在前人工作的基础上加以利用、修改或添加新内容，但必须公开源代码，允许其他人在此基础上继续工作。正因为

如此，Linux才发展得如此迅速和健康。1994年3月14日，Linus 发布Linux的第一个“产品”版Linux1.0的时候，是按完全自由发布版权进行发布的。它要求所有的源代码必须公开，而且任何人均不得从Linux交易中获利。

然而，半年以后，他开始意识到这种纯粹的自由软件的方式对于 Linux的发布和发展来说实际上是一种障碍，因为它限制了Linux以磁盘拷贝或者CD－ROM等媒体形式进行发布的可能，也限制了一些商业公司参与Linux的进一步开发并提供技术支持的良好愿望。于是 Linus决定转向GPL版权，这一版权除了规定有自由软件的各项许可权之外，还允许用户出售自己的程序拷贝，并从中赢利。

这一版权上的转变后来证明对于 Linux的进一步发展确实至关重要。从此以后，便有多家技术力量雄厚又善于市场运作的商业软件公司加入了原先完全由业余爱好者和网络黑客所参与的这场自由软件运动，开发出了多种 Linux的发布版本，增加了更易于用户使用的图形界面和众多的软件开发工具，极大地拓展了 Linux的全球用户基础。并有多家著名的商业软件开发公司开发了基于Linux 的商业软件，如ORACLE、INFORMIX 等。Linus本人也认为：“使Linux转向GPL是我一生中所做过的最漂亮的一件事”

1.3 Linux 的发音

世界各地的人对 Linux 的发音不尽相同，你可以在下面的网址找到 Linux 的发音：ftp://ftp.linux.org/pub/kernel/SillySounds/english.au。

1.4 Linux的特点

- **全面的多任务和真正的32位操作系统**。Linux和其他UNIX系统一样是真正的多任务系统，它允许多个用户同时在一个系统上运行多道程序。 Linux还是真正的 32位操作系统，它工作在Intel 80386 和后来的Intel 处理器的保护模式下。
- X Window **系统**。X Window 是UNIX 平台上的事实工业标准。XFree86 则是Linux平台上的X Window 系统。X Window 系统是功能强大的图形界面，支持多种应用程序。
- **支持**TCP/IP**协议**。在Linux 系统中，通过Ethernet 可以连接到Internet 或当地的局域网。使用SLIP (Serial Line Internet Protocol) 或 PPP (Point to Point Protocol)，通过电话线和调制解调器也可连到Internet上。
- **虚拟内存和共享库**。Linux 可以利用你的硬盘的一部分作为虚拟内存，从而扩展你的可用内存数量。Linux 不使用分段，也没有虚拟内存的限制。 Linux 同时利用共享库技术，允许那些使用标准子过程的程序在运行时共享子过程，从而节约了大量的系统空间。
- Linux **内核中的代码均为自由代码**。Linux 上的大部分程序是自由软件。这些软件是在自由软件基金会的GNU 计划下开发的。尽管如此，来自世界各地的黑客、程序员，甚至商业公司也加入了Linux 软件开发的行列。
- Linux **支持商业版**UNIX **的全部功能**。事实上，Linux 系统上的一些功能是UNIX 系统所不具备的。
- GNU **软件的支持**。Linux支持大部分GNU 计划下的自由软件，包括GNU C 和GCC 编译器、gawk、groff 和其他软件。
- Linux **符合**IEEE POSIX.1**标准**。Linux 特别注重可移植性，这样也支持UNIX 的其他一些标准。
- **Linux 比其他UNIX系统更为便宜**。如果通过Internet 下载Linux，则不用花一分钱。如果

购买Linux 发布，也很便宜。

- Linux**支持多种硬件平台**。从低端的Intel386直到高端的超级并行计算机系统，都可以运行Linux系统。
- Linux **系统网络功能强大**。不仅仅因为Linux系统内核中紧密地集成了网络功能和有大量的网络应用程序，更因为Linux系统在超强网络需求下表现出的令人惊奇的健壮性。

1.5 基本硬件要求

- Intel 80386 **或以上**CPU (**当然越快越好**)。Linux 可以充分利用Windows 淘汰掉的386或486 机器，且它们的运行效率会令你大吃一惊。数据协处理器不是必需的，当然 486 以上的机器不存在这个问题（如果你真的没有数据协处理器， Linux 将处理浮点运算）。
- ISA、 EISA **或**PCI **的总线结构**。Linux 现在不支持微通道（MCA）总线结构。
- **至少**4MB**内存**。如果想运行X Window，则至少需要8MB内存。
- **至少**150MB**的硬盘**。全部安装则需要至少250MB的硬盘。
- Hercules、 CGA、 EGA、 VGA或Super VGA 的显示卡和显示器。Linux支持大部分的显示器和显示卡，但X Window 不支持部分显示设置。
- **真正三键的鼠标**。Linux 会用到鼠标的中间键。但有些Microsoft 鼠标的中间键仅仅作为装饰用。
- **软盘或光驱**。虽然Linux 有软盘版，但光盘版无疑既方便又快捷。

1.6 如何获得Linux

现在人们可以买到各种不同的Linux 发布，所谓Linux 发布也就是各公司把 Linux源代码编译在一起，再加上自己特殊的软件和图形界面。有些发布可以从网上下载，有些可以通过光盘或软盘的方式购买。

1.6.1 从网上下载Linux

可以从网上下载 Linux 的地址有：

- ftp://sunsite.unc.edu/pub/Linux: 各种 Linux 文件和其他资源。
- ftp://ftp.linux.org/pub/: 一个全面的 Linux 站点，包括 Linux 核心、网络工具、文档计划和大部分 Linux 发布。
- ftp://ftp.caldera.com/pub/: Caldera 公司关于 Linux 发布的主页。
- ftp://ftp.debian.org/: Debian 公司关于 Linux 发布的主页。
- ftp://ftp.kernel.org/: 最新 Linux 核心的主页。
- ftp://ftp.cc.gatech.edu/pub/linux/: sunsite.unc.edu 的完全镜像。
- ftp://tsx.mit.edu/pub/linux/: 各种 Linux 文件和其他资源。
- ftp://ftp.pht.com/pub/linux/: 各种 Linux 文件。
- ftp://ftp.redhat.com/pub/: RedHat 公司的主页。

1.6.2 从光盘获得Linux

可以通过光盘形式购买的 Linux 发布有：

1. Caldera OpenLinux(见图1-1)

发布商：Caldera

简介：Caldera 公司的 OpenLinux 是多用户、多任务的操作系统，使你在个人计算机上感受UNIX系统的强大功能和可靠性。 OpenLinux 中还包括一些实用工具、图形界面、安装指南、第三方的应用软件等。OpenLinux 是各种公司优化其现存系统、保护培训投资的理想选择。

2. Debian GNU/Linux(见图1-2)

发布商：Debian

图 1-1

图 1-2

简介：Debian 公司的GNU/Linux 是基于操作系统的Linux 的自由发布。它由一群自愿者进行维护和升级。它的先进的管理工具包使得安装和维护都异常的简单。发布前全面的测试保证了系统的高度可靠性。一个公开的bug跟踪系统随时监控用户的反馈。

图 1-3

3. Linux Mandrake(见图1-3)

发布商：Mandrake

简介：Linux Mandrake 是基于Linux 的32 位多任务操作系统。它可以运行在所有Intel 以及与其兼容的结构中(486、Pentium、Pentium Pro、Pentium MMX、Pentium II 和其他兼容的CPU)。Linux Mandrake 在Linux 系统中加入了一个功能十分强大的图形桌面： KDE。它来自于著名的Apache 页面服务器， GNU Manipulation Image Program Gimp 1.0， Netscape Communicator (4.05) 和其他一些十分优秀的软件。

图 1-4

4. LinuxPPC(见图1-4)

发布商：PowerPC Linux Project

简介： Linux 的 PowerPC 版发布。

5. Linux Pro(见图1-5)

发布商：WorkGroup Solutions

简介：Linux Pro Plus 包括了Linux Pro 的6 张光盘和1套Linux 百科全书——1个1600多页的参考手册。

图 1-5

6．LinuxWare(见图1-6)

发布商：Trans-Ameritech

简介：这是一个十分灵活、易于安装的、类似于 UNIX 的操作系统，主要面向那些对UNIX 系统感兴趣的学生和家庭PC 使用者。可以在Windows、 Windows 95 或 DOS 系统下的CD-ROM 驱动器中安装。

7. MkLinux(见图1-7)

发布商：Apple Computer / The Open Group Research Group

图 1-6

图 1-7

简介：Power Macintosh 平台的Linux 发布。

8. RedHat Linux(见图1-8)

发布商：RedHat Software

简介：RedHat Linux 同时支持Intel、 Alpha 和SPARC 平台。这也是 RedHat 公司最引以自豪的地方。

9. Slackware Linux(见图1-9)

发布商：Walnut Creek

图 1-8

图 1-9

图 1-10

简介：Slackware Linux 支持大多数 Intel PC 。先进的2.0.30 核心提高了高端系统的性能。它支持对称多处理（最多可达16个处理器）、PCI，并为486、 Pentium和Pentium Pro 进行了特别的编码优化。

10. Stampede Linux(见图1-10)

发布商：Stampede

简介：专为超级用户设计。

11. S.u.S.E Linux(见图1-11)

发布商：S.u.S.E Linux

图 1-11

简介：S.u.S.E Linux 共有5张光盘，其中包括Linux 操作系统和超过800 个预设软件包以及400页的参考手册。其中的YaST实用工具允许用户自己安装、设置和进一步地配置系统。 S.u.S.E 支持X Servers 的高端图形卡。

12. TurboLinux(见图1-12)

发布商：Pacific HiTech

简介：TurboLinux 包括一系列的应用程序、一个 GUI (XFree86 3.3) 的桌面、文档和技术支持。

13. Yggdrasil Linux(见图1-13)

发布商：Yggdrasil Computing, Inc.

图 1-12

图 1-13

简介：这个带有即插即用功能的Plug & Play Linux 共有2张光盘。第1张光盘是系统程序，第2张光盘是源代码。只要插入启动软盘和第一张光盘，打开计算机，系统就会自动进行必要的硬件设置，完成系统安装。

1.7 涉及Linux 的Web 网址和新闻讨论组

Linux 是通过Internet 发展壮大的。所以如果有什么问题，尽管到 Internet 上去寻找答案。下面是常用的涉及 Linux 的 Web 网址：

http:// www.ssc.com/linux	Linux 资源
http:// www.caldera.com	Caldera 公司的网址
http:// www.redhat.com	RedHat 公司的网址
http:// sunsite.unc.edu/mdw	Linux 文档计划的网址
http:// www.ssc.com/lg	Linux 杂志
http:// www.linux.org	Linux 的官方网址
http:// www.li.org	Linux 国际机构网址
http:// www.uk.linux.org	Linux 欧洲网址
http:// www.blackdown.org	linux Java 的网址

下面是常见的Linux 新闻讨论组：

comp.os.linux.announce	Linux 的发展情况
comp.os.linux.development.apps	Linux 的应用程序
comp.os.linux.development.system	Linux 的操作系统内核
comp.os.linux.hardware	Linux 硬件方面的问题
comp.os.linux.admin	Linux 系统管理方面的问题
comp.os.linux.misc	Linux 的一些特别的问题和回答
comp.os.linux.setup	Linux 安装和启动
comp.os.linux.answers	关于Linux命令的问题和解答

comp.os.linux.help	Linux 的帮助
comp.os.linux.networking	关于Linux网络的问题和解答

1.8 Linux 的不足之处

- 缺乏文档，晦涩难懂，缺少统一性。
- 没有及时的技术支持。
- 安装和升级不方便。

第2章 外壳及常用命令

2.1 登录和退出

Linux 启动后，给出 login 命令，等待用户登录。

Login: <输入用户名>

Password: <输入密码>

如果是正确的用户名和密码，那么你就会进入Linux 的外壳， 外壳给出命令提示符，等待你输入命令（不要随意以root身份登录，以避免对系统造成意外的破坏）。

使用 logout命令退出外壳。

2.2 Linux 系统的外壳

外壳是一种命令解释器，它提供了用户和操作系统之间的交互接口。外壳是面向命令行的，而 X Window 则是图形界面。你在命令行输入命令，外壳进行解释，然后送往操作系统执行。外壳可以执行 Linux 的系统内部命令，也可以执行应用程序。你还可以利用外壳编程，执行复杂的命令程序。

Linux 提供几种外壳程序以供选择。常用的有 Bourne 外壳(bsh)、C外壳(csh)和Korn 外壳(ksh)。各个外壳都能提供基本的功能，又有其各自的特点。

Bourne 外壳是由Steven Bourne 编写的，是UNIX 的缺省外壳。Bourne 外壳的外壳编程能力很强。但它不能处理命令的用户交互特征。bash 是Bourne 外壳的增强版。

C外壳是由加利福尼亚大学伯克利分校的Bill Joy编写的。它能提供Bourne 外壳所不能处理的用户交互特征，如命令补全、命令别名、历史命令替换等。很多人认为， C 外壳的编程能力不如Bourne 外壳，但它的语法和C语言类似，所以C程序员将发现C 外壳很顺手。tcsh 是C外壳的增强版本和C外壳完全兼容。

Korn外壳是由Dave Korn 编写的。Korn 外壳融合了C 外壳和Bourne 外壳的优点，并和Bourne 外壳完全兼容。 Korn 外壳的效率很高，其命令交互界面和编程交互界面都很不错。Public Domain Korn 外壳(pdksh)是Korn 外壳的增强版。

bash 是大多数Linux系统的缺省外壳。它克服了Bourne 外壳的缺点，又和Bourne 外壳完全兼容。Bash有以下的特点：

- **补全命令行**。 当你在bash 命令提示符下输入命令或程序名时，你不必输全命令或程序名，按Tab 键，bash将自动补全命令或程序名。
- **通配符**。 在bash下可以使用通配符 * 和？。*可以替代多个字符，而？则替代一个字符。
- **历史命令**。 bash 能自动跟踪你每次输入的命令，并把输入的命令保存在历史列表缓冲区。缓冲区的大小由HISTSIZE 变量控制。当你每次登录后， home 目录下的 .bash_history 文件将初始化你的历史列表缓冲区。你也能通过history 和fc 命令执行、编辑历史命令。
- **别名**。 在bash下，可用alias 和unalias 命令给命令或可执行程序起别名和清除别名。这样你可以用自己习惯的方式输入命令。

- **输入/输出重定向**。 输入重定向用于改变命令的输入，输出重定向用于改变命令的输出。输出重定向更为常用，它经常用于将命令的结果输入到文件中，而不是屏幕上。输入重定向的命令是<，输出重定向的命令是>。
- **管道**。 管道用于将一系列的命令连接起来。也就是把前面的命令的输出作为后面的命令的输入。管道的命令是|。
- **提示符**。 bash 有两级提示符。第一级提示符就是你登录外壳时见到的，缺省为 $。你可以通过重新给ps1变量赋值来改变第一级提示符。当bash需要进一步提示以便补全命令时，会显示第二级提示符。第二级提示符缺省为 >，你可以通过重新给 ps2变量赋值来改变第二级提示符。一些特殊意义的字符也可以加入提示符赋值中。
- **作业控制**。 作业控制是指在一个作业执行过程中，控制执行的状态。你可以挂起一个正在执行的进程，并在以后恢复该进程的执行。按下 Ctrl+Z 挂起正在执行的进程，用 bg 命令使进程恢复在后台执行，用 fg命令使进程恢复在前台执行。

2.3 外壳的常用命令

下面简单介绍外壳下的常用命令。

2.3.1 更改帐号密码

语法： passwd
Old password: <输入旧密码>
New password: <输入新密码(最好为6～8字，英文字母与数字混合)>
Retype new password: <再输入一次密码>

2.3.2 联机帮助

语法： man 命令
例如:
man ls

2.3.3 远程登录

语法：rlogin 主机名 [-1 用户名]
例如：
rlogin doc 远程登录到工作站 doc 中。
rlogin doc -l user 使用 user 帐号登录到工作站 doc 中。
语法：telnet 主机名 或 telnet IP地址
例如：
telnet doc
telnet 140.109.20.251

2.3.4 文件或目录处理

列出文件或目录下的文件名。
语法： ls [-atFlgR] [name]

name ：文件名或目录名。
例如：

ls	列出目前目录下的文件名。
ls -a	列出包括以 . 开始的隐藏文件的所有文件名。
ls -t	依照文件最后修改时间的顺序列出文件名。
ls -F	列出当前目录下的文件名及其类型。以/ 结尾表示为目录名，以* 结尾表示为可执行文件，以@ 结尾表示为符号连接。
ls -l	列出目录下所有文件的权限、所有者、文件大小、修改时间及名称。
ls -lg	同上，并显示出文件的所有者工作组名。
ls -R	显示出目录下以及其所有子目录的文件名。

2.3.5 改变工作目录

语法：cd [name]
name：目录名、路径或目录缩写。
例如:

cd	改变目录位置至用户登录时的工作目录。
cd dir1	改变目录位置至dir1目录下。
cd ~user	改变目录位置至用户的工作目录。
cd ..	改变目录位置至当前目录的父目录。
cd ../user	改变目录位置至相对路径 user 的目录下。
cd /../..	改变目录位置至绝对路径的目录位置下。
cd ~	改变目录位置至用户登录时的工作目录。

2.3.6 复制文件

语法: cp [-r] 源地址 目的地址
例如：

cp file1 file2	将文件 file1 复制成 file2。
cp file1 dir1	将文件 file1 复制到目录 dir1 下，文件名仍为 file1。
cp /tmp/file1 .	将目录 /tmp 下的文件 file1 复制到当前目录下，文件名仍为 file1。
cp /tmp/file1 file2	将目录 /tmp 下的文件 file1 复制到当前目录下，文件名为file2。
cp -r dir1 dir2	复制整个目录。

2.3.7 移动或更改文件、目录名称

语法： mv 源地址 目的地址
例如：

mv file1 file2	将文件 file1更名为 file2。
mv file1 dir1	将文件 file1移到目录 dir1 下，文件名仍为 file1。
mv dir1 dir2	将目录 dir1 更改为目录 dir2。

2.3.8 建立新目录

语法： mkdir 目录名

例如：
mkdir dir1 建立一新目录 dir1。

2.3.9 删除目录

语法： rmdir 目录名 或 rm 目录名
例如：
rmdir dir1 删除目录 dir1，但 dir1 下必须没有文件存在，否则无法删除。
rm -r dir1 删除目录 dir1及其子目录下所有文件。

2.3.10 删除文件

语法： rm 文件名
例如：
rm file1 删除文件名为 file1 的文件。
rm file? 删除文件名中有五个字符且前四个字符为 file 的所有文件。
rm f* 删除文件名中以 f 为字首的所有文件。

2.3.11 列出当前所在的目录位置

语法： pwd

2.3.12 查看文件内容

语法： cat 文件名
例如：
cat file1 以连续显示方式，查看文件名 file1 的内容。

2.3.13 分页查看文件内容

语法： more 文件名 或 cat 文件名 | more
例如：
more file1 以分页方式查看文件名 file1 的内容。
cat file1 | more 以分页方式查看文件名 file1 的内容。

2.3.14 查看目录所占磁盘容量

语法: du [-s] 目录
例如:
du dir1 显示目录 dir1 的总容量及其子目录的容量(以KB 为单位)。
du -s dir1 显示目录 dir1 的总容量。

2.3.15 文件传输

1. 拷贝文件或目录至远程工作站
语法： rcp [-r] 源地址 主机名:目的地址
源地址文件名、目录名或路径。

主机名工作站名。目的地址路径名称。

例如：

rcp file1 doc:/home/user　　将文件file1拷贝到工作站 doc 路径 /home/user 下。

rcp -r dir1 doc:/home/user　　将目录 dir1拷贝到工作站 doc 路径/home/user 下。

2. 自远程工作站，拷贝文件或目录

语法： rcp [-r] 主机名:源地址 目的地址

主机名工作站名。

源地址路径名。

目的地址、文件名、目录名或路径 。

例如：

rcp doc:/home/user/file1 file2　　将工作站 doc路径/home/user 下的目录 dir1，拷贝到当前工作站的目录下，目录名仍为 dir1。

rcp -r doc:/home/user/dir1 .　　将工作站doc 路径/home/user 下的目录 dir1，拷贝到当前工作站的目录下，目录名仍为 dir1。

3. 本地工作站与远程工作站之间的文件传输

必须拥有远程工作站的帐号及密码，才可进行传输工作。

语法： ftp 主机名 或 ftp ip地址

例如：

ftp doc　　与远程工作站 doc 之间进行文件传输。

Name (doc:user-name): <输入帐号>

Password (doc:user-password): <输入密码>

ftp> help	列出 ftp 文件传输时可使用的命令。
ftp> !ls	列出本地工作站当前目录下的所有文件名。
ftp> !pwd	列出本地工作站当前所在的目录位置。
ftp> ls	列出远程工作站当前目录下的所有文件名。
ftp> dir	列出远程工作站当前目录下的所有文件名。
ftp> dir . |more	分页列出远程工作站当前目录下的所有文件名。
ftp> pwd	列出远程工作站当前所在的目录位置。
ftp> cd dir1	更改远程工作站的工作目录位置至 dir1 之下。
ftp> get file1	将远程工作站的文件 file1拷贝到本地工作站中。
ftp> put file2	将本地工作站的文件 file2拷贝到远程工作站中。
ftp> mget *.c	将远程工作站中扩展文件名为 c 的所有文件拷贝到本地工作站中。
ftp> mput *.txt	将本地工作站中扩展文件名为 txt 的所有文件拷贝到远程工作站中。
ftp> prompt	切换交互式指令(使用 mput/mget 时不是每个文件皆询问yes/no)。
ftp> quit	结束 ftp 工作。
ftp> bye	结束 ftp 工作。

注意　从PC与工作站间的文件传输也可透过在 PC端的 FTP指令进行文件传输，指令用法与上述指令大致相同。

2.3.16 文件权限的设定

1. 改变文件或目录的读、写、执行权限

语法：chmod [-R] mode name

name:文件名或目录名。

mode: 3个8位数字或rwx的组合。r-read(读)，w-write(写)，x-execute(执行)，u-user(当前用户)，g-group（组），o-other（其他用户）。

例如:

chmod 755 dir1　对于目录dir1，设定成任何使用者皆有读取及执行的权利，但只有所有者可做修改。

chmod 700 file1　对于文件file1，设定只有所有者可以读、写和执行的权利。

chmod u+x file2　对于文件file2，增加当前用户可以执行的权利。

chmod g+x file3　对于文件file3，增加工作组使用者可执行的权利。

chmod o-r file4　对于文件file4，删除其他使用者可读取的权利。

2．改变文件或目录的所有权

语法：chown [-R] 用户名 name

name：文件名或目录名。

例如 ：

chown user file1　将文件 file1 改为用户user 所有。

chown -R user dir1　将目录 dir1及其子目录下面的所有文件改为用户 user 所有。

2.3.17 检查自己所属的工作组名称

语法：groups

2.3.18 改变文件或目录工作组所有权

语法：chgrp [-R] 工作组名 name

name：文件名或目录名

例如：

chgrp vlsi file1　将文件 file1 的工作组所有权改为 vlsi 工作组所有。

chgrp -R image dir1　将目录dir1及其子目录下面的所有文件，改为 image 工作组所有。

2.3.19 改变文件或目录的最后修改时间

语法：touch name

name：文件名或目录名。

2.3.20 文件的链接

同一文件，可拥有一个以上的名称，也就是把一个文件进行链接。

语法：ln 老文件名 新文件名

例如 ：

ln file1 file2　将文件 file2链接至文件 file1。

语法：ln -s 老文件名 新文件名

例如 ：

ln -s file3 file4　将文件 file4 链接至文件file3。

2.3.21 文件中字符串的查寻

语法：grep string file
例如 ：
grep abc file1　　寻找文件file1中包含字符串 abc 所在行的文本内容。

2.3.22 查寻文件或命令的路径

语法：whereis command　　显示命令的路径。
语法：which command　　显示命令的路径，及使用者所定义的别名。
语法：whatis command　　显示命令功能的摘要。
语法：find search-path -name filename -print　　搜寻指定路径下某文件的路径 。
例如 ：
find / -name file1 -print 自根目录下寻找文件 file1 的路径。

2.3.23 比较文件或目录的内容

语法：diff [-r] name1 name2
name1 name2：可同时为文件名或目录名。
例如：
diff file1 file2　　比较文件file1 与 file2 内各行的不同之处。
diff -r dir1 dir2　　比较目录 dir1 与 dir2 内各文件的不同之处。

2.3.24 文件打印输出

用户可用 .login 文件中的 setenv PRINTER来设定打印机名。
例如 ：
setenv PRINTER sp 设定自 sp 打印机打印资料。

2.3.25 一般文件的打印

语法：lpr [-P打印机名] 文件名
例如：
lpr file1 或 lpr -Psp file1　　自 sp打印机打印文件 file1。
语法：enscript [-P打印机名] 文件名
例如：
enscript file3 或 enscript -Psp file3　　自 sp打印机打印文件 file3。

2.3.26 troff 文件的打印

语法：ptroff [-P打印机名] [-man][-ms] 文件名
例如：
ptroff -Psp -man /usr/man/man1/lpr1　　以 troff 格式，自 sp 打印机打印 lpr1 命令的使用说明。

2.3.27 打印机控制命令

1．检查打印机状态、打印作业顺序号和用户名

语法：lpq [-P打印机名]
例如：
lpq 或 lpq -Psp 检查 sp 打印机的状态。
2. 删除打印机内的打印作业 (用户仅可删除自己的打印作业)
语法：lprm [-P打印机名] 用户名 或 作业编号
例如：
lprm user或 lprm -Psp user 删除 sp打印机中用户user 的打印作业，此时用户名必须为 user。
lprm -Psp 456 删除 sp 打印机上编号为 456 的打印作业。

2.3.28 进程控制

1．查看系统中的进程
语法：ps [-aux]
例如:
ps或ps -x 查看系统中，属于自己的进程。
ps -au 查看系统中，所有用户的进程。
ps -aux 查看系统中，包含系统内部的及所有用户的进程。
2. 结束或终止进程
-9应该是强行终止
语法：kill [-9] PID
PID：利用 ps 命令所查出的进程号。
例如:
kill 456 或kill -9 456 终止进程号为 456 的进程。
3. 在后台执行进程的方式
语法：命令 &
例如:
cc file1.c & 将编译 file1.c 文件的工作置于后台执行。
语法：按下 Control+Z键，暂停正在执行的进程。键入 bg命令，将暂停的进程置于后台继续执行。
例如:
cc file2.c
^Z
Stopped
bg
4. 查看正在后台中执行的进程
语法：jobs
5. 结束或终止后台中的进程
语法：kill %n
n：利用jobs命令查看出的后台作业号
例如：
kill % 终止在后台中的第一个进程。
kill %2 终止在后台中的第二个进程。

2.3.29 外壳变量

1. 查看外壳变量的设定值

语法：set　　查看所有外壳变量的设定值。

语法：echo $变量名　　显示指定的外壳变量的设定值。

2. 设定外壳变量

语法：set var = value

例如:

set term=vt100　　设定外壳变量 term为 VT100 型终端。

3. 删除外壳变量

语法：unset var

例如:

unset PRINTER　　删除外壳变量 PRINTER 的设定值。

2.3.30 环境变量

1. 查看环境变量的设定值

语法：setenv　　查看所有环境变量的设定值。

语法：echo $NAME　　显示指定的环境变量NAME的设定值。

例如 :

echo $PRINTER　　显示环境变量 PRINTER 的设定值。

2. 设定环境变量

语法：setenv NAME word

例如:

setenv PRINTER sp　　设定环境变量 PRINTER 为 sp。

3. 删除环境变量

语法：unsetenv NAME

例如:

unsetenv PRINTER　　删除环境变量PRINTER的设定值。

2.3.31 别名

1. 查看所定义的命令的别名

语法： alias　　查看自己目前定义的所有命令，及所对应的别名。

语法： alias name　　查看指定的name 命令的别名。

例如:

alias dir 查看别名 dir 所定义的命令。

ls -atl

2. 定义命令的别名

语法： alias name ‘ command line ’

例如:

alias dir ‘ ls -l ’　　将命令 ls - l 定义别名为 dir。

3. 删除所定义的别名

语法： unalias name
例如：
unalias dir 删除别名 dir 的定义。
unalias * 删除所有别名的设定。

2.3.32 历史命令

1. 设定命令记录表的长度
语法： set history = n
例如:
set history = 40 设定命令记录表的长度为 40 (可记录执行过的前面 40 个命令)。
2. 查看命令记录表的内容
语法： history
3. 使用命令记录表
语法： !! 重复执行前一个命令。
语法： !n
n：命令记录表的命令编号。
语法： !string 重复前面执行过的以 string 为起始字符串的命令。
例如:!cat 重复前面执行过的以 cat 为起始字符串的命令。
4. 显示前一个命令的内容
语法： !!：p
5. 更改前一个命令的内容并执行
语法： ^oldstring ^newstring 将前一个命令中 oldstring 的部份改成 newstring并执行。
例如：
find . -name file1.c -print
^file1.c^core
find . -name core -print

2.3.33 文件的压缩

1. 压缩文件
语法：compress 文件名 压缩文件
语法：compressdir 目录名 压缩目录
2. 解压缩文件
语法：uncompress 文件名 解压缩文件
语法：uncompressdir 目录名 解压缩目录

2.3.34 管道命令的使用

语法：命令1 | 命令2 将命令1的执行结果送到命令2，做为命令2的输入。
例如：

ls -Rl | more 以分页方式列出当前目录及其子目录下所有文件的名称。
cat file1 | more 以分页方式列出文件 file1 的内容。

2.3.35 输入/输出控制

1. 标准输入的控制
语法：命令 < 文件 将文件做为命令的输入。
例如：
mail -s “ mail test ” wesongzhou@hotmail.com < file1 将文件file1 当做信件的内容，主题名称为 mail test，送给收信人。
2. 标准输出的控制
语法：命令 > 文件 将命令的执行结果送至指定的文件中。
例如:
ls -l > list 将执行 “ ls -l ” 命令的结果写入文件list 中。
语法：命令>! 文件 将命令的执行结果送至指定的文件中，若文件已经存在，则覆盖。
例如：
ls -lg >! list 将执行 “ ls - lg ” 命令的结果覆盖写入文件 list 中。
语法：命令 >& 文件 将命令执行时屏幕上所产生的任何信息写入指定的文件中。
例如：
cc file1.c >& error 将编译 file1.c 文件时所产生的任何信息写入文件 error 中。
语法：命令>> 文件 将命令执行的结果附加到指定的文件中。
例如:
ls - lag >> list 将执行 “ ls - lag ” 命令的结果附加到文件 list 中。
语法：命令 >>& 文件 将命令执行时屏幕上所产生的任何信息附加到指定的文件中。
例如:
cc file2.c >>& error 将编译 file2.c 文件时屏幕所产生的任何信息附加到文件 error 中。

2.3.36 查看系统中的用户

语法： who 或 finger
语法： w
语法： finger 用户名 或 finger 用户名@域名

2.3.37 改变用户名

语法： su 用户名
例如:
su user 进入用户user 的帐号。
passwrod: <输入用户user 的密码>

2.3.38 查看用户名

语法： who am i 查看登录时的用户名。

语法： whoami 查看当前的用户名。若已执行过su命令，则显示出此用户的用户名。

2.3.39 查看当前系统上所有工作站的用户

语法: rusers
按Ctrl+C> 结束

2.3.40 与某工作站上的用户交谈

语法: talk 用户名@主机名或talk 用户名@IP地址
例如:
1) 可先利用 rusers 指令查看网络上的用户；
2) 假设自己的帐号是 u84987 ，在工作站 indian 上使用，现在想要与 doc 上的u84123 交谈。
talk u84123@doc
此时屏幕上将会出现等待画面
在对方(u84123)屏幕上将会出现下列信息：
Message from Talk_Daemon@Local_host_name at xx:xx
talk: connection requested by u84987@indian
talk: respond with: talk u84987@indian
此时对方(u84123) 必须执行 talk u84987@indian 即可互相交谈。
最后可按Ctrl+C结束。

2.3.41 检查远程系统是否正常

语法：ping 主机名或ping IP地址
例如:
ping doc

2.3.42 电子邮件的使用简介

1. 将文件当做电子邮件的内容送出
语法：mail -s “ 主题 ” 用户名@地址 < 文件
例如：
mail -s “ program ” user < file.c 将 file.c 当做mail的内容，送至 user，主题为 program。
2. 传送电子邮件给本系统用户
语法：mail 用户名
3. 传送电子邮件至外地用户
语法： mail 用户名@接受地址
例如：
mail weisongzhou@hotmail.com
Subject : mail test
:

:
键入信文内容
:
:
按下 Ctrl+D 键或 . 键结束正文。
连按两次Ctrl+C键则中断工作，不送此信件。
Cc(Carbon copy) : 复制一份正文，给其他的收信人。
3. 检查所传送的电子邮件是否送出，或滞留在邮件服务器中
语法：/usr/lib/sendmail -bp
若屏幕显示为“ Mail queue is empty ”的信息，表示 mail 已送出。
若为其他错误信息，表示电子邮件因故尚未送出。

第3章 Linux系统的网络功能

本章介绍Linux系统的网络功能，如支持的网络协议、文件和打印共享、 Internet/Intranet功能、应用程序的远程运行、网络互连功能等。

3.1 Linux支持的网络协议

Linux支持多种不同的网络协议。

3.1.1 TCP/IP

TCP/IP从一开始就集成到了 Linux系统之中，并且其实现完全是从新编写的。现在，TCP/IP 已成为Linux系统中最健壮、速度最快和最可靠的部分，也是 Linux系统之所以成功的一个关键因素。

3.1.2 TCP/IP 版本 6

IPv6，也称为IPng (IP Next Generation)，是IPv4 协议的升级，并解决了其中的很多问题，例如：IPv4缺少足够的可用IP地址，没有处理实时网络请求的机制，缺少网络层的安全机制等。IPv6即将成为Linux 2.2.0 核心的一部分。

3.1.3 IPX/SPX

IPX/SPX (Internet Packet Exchange/Sequenced Packet Exchange) 是 Novell公司基于XNS (Xerox Network Systems) 的网络协议集。IPX/SPX 在八十年代早期成为Novell 公司的 NetWare 的一部分。Linux 系统中有IPX/SPX 的完整实现。Linux系统可以设置为：

- IPX 路由器。
- IPX 网桥。
- NCP 客户机 和/或 NCP 服务器。
- Novell 打印客户机， Novell 打印服务器。

并且可以：

- 具有 PPP/IPX 功能， Linux 系统可以作为 PPP 服务器/客户机。
- IPX 通过IP互连， 允许两个IPX 网络通过 IP 链路互连。

3.1.4 AppleTalk 协议集

Appletalk 是 Apple 公司的网络互连协议。它提供对等的网络互连模型（peer-to-peer），并提供文件共享、打印共享等基本网络功能。每个计算机都可以设置为客户机和服务器，但同时每台计算机都要安装必要的硬件和软件。

Linux 可以提供整套Appletalk 网络功能。Netatalk 是AppleTalk协议的核心层实现，它最初是为BSD UNIX系统编写的。

3.1.5 广域网

很多厂商提供 T-1、T-3、X.25 和帧中继的Linux产品。详情请参阅：http://www. secreta-gent.com/networking/wan.html。

3.1.6 ISDN

Linux 内核中集成了ISDN 功能。Isdn4linux 可以控制ISDN的PC卡并能模拟调制解调器。其应用从终端程序通过HDLC连接一直到通过PPP 连接Internet 。

3.1.7 PPP、SLIP及PLIP

Linux 内核中也集成了对 PPP (Point to Point Protocol)和SLIP (Serial Line IP)以及PLIP (Parallel Line IP)的支持。个人计算机用户连接ISP (Internet Service Provider)的最常用方式就是PPP。PLIP 允许实现两台计算机通过并行口的简单连接，速率可达到 10～20kBps。

3.1.8 业余无线电

Linux 内核中还集成了对业余无线电（Amateur Radio）协议的支持。特别令人感兴趣的是对AX.25协议的支持。 AX.25协议提供了有连接和无连接两种操作方式。 AX.25即可本身用来实现点到点的连接，也可用来传送其他协议，例如 TCP/IP 和NetRom。此协议结构上和X.25第二层十分接近，只是做了扩展以便更适合于业余无线电环境。

3.1.9 ATM

Linux 对ATM 的支持还处于实验阶段。现有一个测试版本支持 ATM连接、通过ATM的IP连接以及局域网仿真等。详情请参阅：http://lrcwww.epfl.ch/linux-atm/。

3.2 Linux系统下的文件共享和打印共享

很多计算机连接到局域网的主要目的就是共享文件和打印机。 Linux系统作为文件和打印服务器将会提供一个很好的解决方案。

3.2.1 Machintosh 环境

正如前面所说， Linux 支持Appletalk 协议。Linux系统的 netatalk 允许Macintosh客户机将Linux系统视为网络上的一台Macintosh 计算机，这样就可以共享Linux服务器上的文件系统和打印机。

3.2.2 Windows 环境

Samba由一系列的应用程序组成，它允许Linux系统既可以作为服务器，又可以作为客户机集成到 Microsoft 网络环境中。作为服务器， Samba允许Windows 95， Windows for Workgroups， DOS 和Windows NT 客户机共享Linux文件系统和打印服务。它可以完全替代Windows NT作为文件和打印服务器，包括自动为客户机下载打印机驱动程序。作为客户机，Samba允许Linux 工作站在本地安装共享的windows 文件。

3.2.3 Novell 环境

Linux 可以作为NCP客户机或服务器，允许在Novell 网络上为Novell 和UNIX 客户机提供文件和打印服务。

3.2.4 UNIX 环境

在UNIX 环境下共享文件的最好方法是通过 NFS（Network File Sharing）。NFS最初是由Sun 公司开发的，是一种在两台计算机间如同本地一样共享文件的方法。客户机可以安装 NFS服务器上共享的文件系统。对客户机来说，被安装的文件系统就象本地的文件系统一样。可以在启动时安装根文件系统，这样，无盘工作站可以远程启动并存取服务器上的文件。

3.3 Linux系统中的Internet/Intranet功能

Linux是十分优秀的Intranet/Internet服务器平台。Intranet是指在公司内部应用Internet技术发布和共享信息。Linux提供的 Internet和Intranet服务包括邮件、新闻、WWW服务器和其他一些服务。

3.3.1 邮件

1. 邮件服务器

Sendmail 是UNIX 平台上mail 服务器程序的工业标准。它的功能十分强大，易于扩展。如果硬件配置得当， Sendmail 可以轻松处理成千上万个网络请求。其他的邮件服务器程序，如smail 和qmail可以作为sendmail的替代。

2. 远程邮件存取

在公司机构或ISP中，用户可能是在本地远程存取邮件。 Linux系统提供了几种选择方案用于处理这种情况，包括POP (Post Office Protocol) 和IMAP (Internet Message Access Protocol)服务器。 POP 一般用来从服务器向客户机传送信息，而 IMAP 允许用户处理服务器中的信息，远程建立和删除服务器的文件夹，同时存取共享的邮件文件夹等。

3. 邮件用户代理

无论是在图形方式下还是在文本方式下， Linux系统都有很多MUA（Mail User Agent）。广泛使用的MUA有：pine、elm、mutt和Netscape。

4. 邮件列表管理程序

在UNIX系统中有很多MLM(Mail List Management)，Linux 系统中也有很多此类软件。

在下面的ftp中有关于各种MLM的比较：ftp://ftp.uu.net/usenet/news.answers/mail/list-admin/。

5. 读取邮件

一个和邮件有关的功能就是Fetchmail，它是一个免费的，功能全面，健壮性很好，并且文档组织很好的远程邮件读取和发送工具。它主要用于 TCP/IP 的需求既用链接（例如SLIP 或者PPP链接）。它支持各种Internet上正在使用的远程邮件协议，甚至支持Pv6 和IPSEC。

Fetchmail 从远程邮件服务器中读取邮件，并通过 SMTP传送，所以一般的邮件用户代理（Mail User Agent），象mutt，elm或BSD Mail都可以读取邮件。

Fetchmail 可以用来作为整个DNS域的POP/IMAP-to-SMTP网关，它从ISP 的一个单个信箱中搜集邮件，并根据信头地址使用SMTP发送。

因此，一个规模较小的公司可以使用一个单个信箱集中管理邮件。 Fetchmail 程序搜集所有的发出邮件，发送到Internet 上，并同时收取寄入的邮件。

3.3.2 Web 服务器

大多数 Linux 发布包括 Apache (http://www.apache.org)。 Apache可以说是Internet上的头号服务器。超过半数的Internet站点正在运行Apache或Apache的变形。 Apache的优点包括其模块化设计，超常的稳定性和速度。只要硬件配置得当， Apache能够负担极大的网络流量。Yahoo,Altavista,GeoCities,Hotmail都使用Apache 服务器的定制版本。

3.3.3 Web 浏览器

Linux 平台有很多浏览器可供选择。网景公司的导航者（ Netscape Navigator）一开始就集成在Linux 的系统中，而Mozilla (http://www.mozilla.org)也将推出Linux 版本。另一个十分流行的基于文本的Web 浏览器是lynx。在没有图形的环境下， lynx十分方便和快捷。

3.3.4 FTP 服务器和客户机

FTP（File Transfer Protocol）服务器允许用户连接并下载文件。 Linux 系统中包括很多FTP服务器和客户机端软件，其中既有基于文本的，也有基于图形界面的。

3.3.5 新闻服务

Usenet (也称做新闻) 是一个大的公告牌系统，其中涉及到各式各样的主题并按层次结构组织。 Internet 上的计算机通过NNTP协议交换文章。 Linux系统包括多种新闻服务的实现方法，分别适用于网络流量很大的站点和仅包括几个新闻组的站点。

3.3.6 域名系统

DNS（Domain Name System）服务器的任务是把名字翻译为IP 地址。一个DNS 当然无法知道所有的IP 地址，但它可以向其他服务器询问自己不知道的地址。 DNS 或者返回已知的IP地址，或者告诉用户其要求的名字没有找到。大多数 UNIX 的域名服务是由叫做named的程序完成的，这也是Internet 软件系统的一部分。

3.3.7 DHCP和bootp

DHCP和bootp是允许客户机从服务器中获取网络信息的协议。现在有很多公司开始使用这些协议，因为这些协议为管理网络带来极大的便利，尤其是比较大的网络和有很多移动用户的网络。

3.3.8 NIS

NIS (Network Information Service) 提供了一个由数据库和处理程序组成的网络查询服务。它的目的是为整个网络上的计算机提供信息服务。例如， NIS允许一个人能够登录到运行 NIS的网络上的任何计算机，而管理员无须在每台计算机上增加口令，而只须在主数据库增加口令即可。

3.4 Linux系统下应用程序的远程执行

UNIX系统的一个令人惊奇的特征就是对应用程序远程和分布执行的支持。

3.4.1 Telnet

Telnet 允许用户远程登录使用计算机，就像本地登录使用一样。 Telnet 是UNIX系统中最为强大的工具之一，它允许真正的远程管理。对用户来说 Telnet也十分有用，因为用户可以通过Internet远程存取他们的文件。如果结合 X serve，那么对用户来说，在本地登录和在地球的任何地方登录没有任何区别。大多数Linux 系统发布中都包括Telnet 。

3.4.2 远程命令

在UNIX系统中，尤其是在Linux系统中， 远程命令允许用户在外壳提示符下交互使用其他的远程计算机。例如：rlogin（允许用户登录远程计算机），rcp（允许在计算机之间传送文件），rsh(允许用户甚至不用登录远程计算机，就能在远程计算机上执行命令)。

3.4.3 X Window

X Window系统是80年代后期MIT 发布的，后来迅速成为UNIX系统图形工作站上的事实工业标准。其软件可免费获得，支持多种硬件平台。任何 X Window系统都包含两部分：X 服务器和一个或多个X 客户机。了解服务器和客户机之间的区别是十分必要的。客户机直接控制输出，并接收来自键盘和鼠标的输入。而服务器则不直接存取屏幕，他们只和客户机之间互相传送信息。真正执行程序或命令的是服务器。客户机和服务器之间相互通讯，使服务器为客户机打开一个或多个窗口处理输入和输出。

简而言之，X Window系统允许用户登录到远程计算机中，执行程序，并在自己的计算机上显示输出结果。因为实际上程序是在服务器端执行，所以客户机端并不需要有强大的处理能力。Linux 系统的X Window为xfree86。大多数Linux 发布中都包括xfree86。

3.5 Linux系统的网络互连功能

Linux 系统的网络互连功能十分强大。一个 Linux系统可以按要求设置为路由器，网桥等。下面就简单介绍一下这些功能。

3.5.1 路由器

Linux 内核集成了对路由功能的支持。一台安装了 Linux系统的计算机可以被设置为 IP 或IPX 路由器。最新的Linux内核中包括了对主要用作路由器的计算机的特殊支持：

• 多路广播(Multicasting):此功能允许Linux 计算机为有多个目的地址的IP数据包充当路由器。这在MBONE(一个宽带的Internet 网络，允许传送声音和图象)中十分重要。

• IP 策略路由(IP Policy Routing): 一般情况下，路由器仅仅通过数据包的最终目的地址来决定路由，而IP 策略路由也将考虑到源地址和数据包经过的网络设备。

3.5.2 网桥

Linux 内核中包括对以太网网桥的支持，这意味着不同的以太网段可以连接成为一个整个的以太网。多个网桥也可以连接在一起组成更大的以太网。因为 Linux 网桥是标准设备，所以Linux 网桥可以和其他第三方厂商的网桥协同工作。

3.5.3 IP伪装

IP伪装(IP Masquerade)是Linux系统独特的网络功能。如果一台 Linux主机连接到Internet ，

并且选中其IP伪装功能，则连接到主机上的计算机，无论是通过局域网还是通过调制解调器，无论有没有正式指定的IP地址，都将会连接到 Internet 上。这样既减少了费用，（因为多台计算机可以通过一个调制解调器连接到Internet ），又提供了安全保障（因为外部用户无法存取没有固定IP地址的计算机）。

3.5.4 IP统计

IP统计（IP Accounting）允许Linux 内核随时监视IP 网络流量，并产生统计信息。

3.5.5 IP 别名

与windows相似，但是怎么怎么做呢？

Linux 内核中的此功能允许为一个同样的低层网络设备设置多个网络地址。例如，在一个以太网卡上设置两个IP 地址。

3.5.6 流量限制器

流量限制器（Traffic Shaper）是一种网络虚拟设备，它用来限制能通过其他网络设备的数据流速。这在脚本（例如ISP的脚本）中非常有用，因为在这种情况下十分需要控制和限制每个客户机的带宽。另一个可能的选择（只对 web 服务器而言）可以是某种Apache 模块，它用来限制连接的客户机数目和使用的带宽。

3.5.7 防火墙

防火墙是一种把私人网络和公共网络隔离开来的装置（公共网络一般指 Internet）。防火墙被用来控制数据包的流向，而这些控制是基于每个数据包的源地址，目的地址，端口和数据包类型等信息。

Linux 内核支持多种不同的防火墙工具。其他的防火墙包括 TIS 和SOCKS。这些防火墙工具大多非常完善，并且如果结合其他工具，可以允许阻塞和重定向各种网络流量和协议。

3.5.8 端口下传

现在，越来越多的网站都在使用 cgi程序或Java 应用程序进行交互式的数据存取。因为这些数据存取可能产生安全问题，所以数据库不能直接连接到 Internet上。

端口下传（Port Forwarding）提供了一个近乎完美的解决方案。在防火墙上，到达一个特定端口的IP 数据包可以被重写，并被下传到提供真正服务的内部服务器。内部服务器的回应数据包也将被重写，使其看起来就像来自防火墙。

3.5.9 负载平衡

当很多客户机同时向服务器发出请求时，数据库 /web的存取就会产生负载平衡的要求。这就要求具备多个相同的服务器，并能将存取要求重定向到负载较小的服务器。通过网络地址翻译技术（NAT）可以获得此效果。网络管理员可以把一个提供 Web服务的服务器替换为具有同样IP 地址的逻辑服务器中的一个。进入服务器的连接根据负载平衡算法被定向到一个特别的服务器。虚拟服务器重写进入和送出的数据包，所以对客户机来说存取服务器的过程是透明的，就像只存在一个服务器一样。

3.5.10 EQL

Linux 系统内核中集成有EQL功能。如果两个计算机之间存在两个使用SLIP 或 PPP的串行连接，使用EQL驱动器可以使两条串行的连接就像一条倍速的串行连接一样。

3.5.11 代理服务器

在网络中，代理服务器作为一些客户机的代表。 HTTP 代理服务器从其他的计算机（计算机A）接收访问web 页的请求。代理服务器把结果返回到计算机 A。代理服务器安装有 cache，所以其他的计算机访问同样的网页时，代理服务器就会从 cache 中返回网页。这样就有效利用了带宽资源，并缩短了回应时间。还有一点就是，客户机不直接存取外面网络资源，保证了内部网络的安全。一个适当配置的代理服务器可以和防火墙媲美。

Linux系统中有一些代理服务器。一个常用的解决方案就是 Apache 代理模块。更加完全和可靠的代理服务器是SQUID。

3.5.12 按需拨号

按需拨号（Diald on demand）的目的是使用户看起来和远程的站点建立了一个固定的连接。一般情况下，有一个监视数据包流向的计算机。当一个特定的数据包（特定是指由一系列的规则、特性和权限决定的）到达时，它就和远程端建立一个连接。当此通道在空闲一段时间后，它就会断开连接。

3.5.13 管道、移动IP和虚拟个人网络

Linux内核允许封装协议。它可以通过IP和IPX管道（Tunnelling），将两个IPX 通过一个IP链接连接在一起。它也可以进行IP-IP 管道，这对移动IP 支持，多路广播支持和业余无线电都十分重要。

移动IP 允许IP数据包传送到Internet上的移动节点。每个移动节点一般由其家庭住址标识，而不管当前连接的Internet移动节点在哪。当移动节点离开自己家的时候，通常都由一个副地址，用来提供其当前连接的 Internet地点的信息。协议为正在注册中的副地址提供一个家庭代理。家庭代理将目的地是移动节点的数据包通过管道发往副地址。当到达管道尽头时，每个数据包都被发送到移动节点。

点到点管道协议（PPTP）可以使用户作为虚拟个人网络（ VPN）使用Internet 。PPTP和Windows NT服务器的远程存取服务集成在一起。使用 PPTP，用户可以拨号到当地的ISP，或者直接连接到Internet，并且可以和在自己的桌面一样存取网络。 PPTP是一个封闭的协议，并且安全性最近有所提高。

3.6 Linux系统中的网络管理

3.6.1 Linux系统下的网络管理应用程序

在网络管理和远程管理方面有很多优秀的工具，例如 Linuxconf 和Webmin。详情请参阅：http://www.webmin.com/webmin/及http://solucor.solucorp.qc.ca/linuxconf/。

其他的工具包括网络流量分析工具，网络安全工具，网络监控工具，网络设置工具等。详情可以查看：http://www.sunsite.unc.edu/pub/Linux/system/network/。

3.6.2 SNMP

简单网络管理协议（SNMP）是关于Internet网络管理服务的协议。它允许远程监控和设置路由器、网桥、网络适配卡和网络开关等。Linux系统上有很多基于SNMP的网络监控程序。

3.7 企业级Linux网络

在一些情况下，保证网络能够随时随刻正常工作是十分必要的。下面就简单介绍一下这方面的相关技术。

3.7.1 高可用性

冗余是用来防止由于单个点不能工作而导致整个系统瘫痪。一个配置了网络适配卡和 SCSI 磁盘的服务器有两个可能不能正常工作的单点。所以最终目标应该是使用户不受意外事故的影响而能继续工作。高可用性软件用来自动监控和侦测错误，采取适当的步骤来恢复正常操作并能及时地通知系统管理员。

3.7.2 RAID

廉价磁盘冗余阵列（Redundant Array of Inexpensive Disks）是指把信息分布存储在几个盘中，以便加快磁盘读写速度和磁盘故障恢复。共有超过 6种以上的RAID设置方法。Linux 系统的解决方案有：软件RAID、外部DASD以及RAID 磁盘控制卡。

1）软件RAID：纯软件的RAID 是在Linux 内核的磁盘代码中（块设备）实现各种级别的RAID。

2）外部DASD ：DASD （Direct Access Storage Device），即直接存取存储装置，是一个单独的自身带有电源的设备。它包括一个架子来托住硬盘。对 Linux 系统来说，DASD 就像另一个SCSI 装置一样。在很多方面来讲，DASD 是最可靠的RAID 解决方案。

3）RAID磁盘控制卡：RAID磁盘控制卡是一块插入在ISA/EISA/PCI 总线上的适配卡。就像一般的磁盘控制卡一样，RAID磁盘控制卡用一条电缆和磁盘连接。和一般的磁盘控制卡不同的是，RAID磁盘控制卡自己实现RAID ，执行各个级别RAID 所须的必要操作。

3.7.3 冗余网络

IPAT (IP Address Takeover)称为IP 地址接管。当一块网络适配卡出现故障时，它的 IP地址应该被同节点或另一个节点的网络适配卡接管。MAC 地址接管：当IP 地址接管完成后，所有网络上的节点都应升级他们的ARP缓存。

第4章 Linux系统管理简介

本章介绍Linux 系统的系统管理，如root帐号、启动和关闭系统和挂接系统等。

4.1 root 帐号

如果使用root 身份登录，你将没有任何的限制。你可以存取任何的文件，控制任何的进程。但你也必须小心从事，任何的错误都有可能导致一场灾难，有时可能破坏整个的操作系统。所以，你必须只有在十分必要的时候才使用 root 帐号（例如重建内核、安装新软件或设置一个新的文件系统等）。一般情况下，不要用root 帐号执行日常的工作。

4.2 启动和关闭系统

由于可以用多种方式安装Linux 系统，所以启动和关闭Linux 系统的方法也有很多种。

4.2.1 从软盘启动

你可以使用启动软盘启动Linux 内核。启动软盘建有root 分区（软盘root 分区可以和硬盘root 分区同时存在）。如果没有root 分区，Linux 系统将无法识别装有操作系统其他部分的硬盘。

在大多情况下，启动软盘是用来紧急恢复Linux 系统的。你可以用启动软盘装入Linux，然后再挂接出现问题的硬盘以便检查故障情况。但如果你没有使用 LILO 来选择可以启动的分区，或者没有设置Linux 的启动顺序为缺省，你就得使用启动软盘启动Linux 系统。

你可以使用rdev 命令设置root 分区：

```
rdev kernelname device
```

kernelname 是内核镜像的名字，device 是Linux 系统的root 分区的名字。例如，你可以设置软盘启动，内核为vmlinuz：

```
rdev vmlinuz /dev/fd0
```

4.2.2 使用LILO 启动

硬盘的启动部分中包括LILO程序，它允许你从硬盘启动Linux 系统。LILO 也可以和OS/2 、DOS、Windows 等其他操作系统一起使用。如果你设置了使用 LILO自动启动Linux，那么在启动后按下Ctrl、Alt或 Shift键就可以进入其他的操作系统。这时会显示一个启动提示符，让你指定其他的操作系统。

如果在启动Linux之前， LILO设置了一段等待时间，那么按Ctrl+Alt+Shift键将直接进入启动过程，而不必等到时间结束。最后，如果 LILO 没有设置为自动启动Linux， 你必须按回车键开始启动Linux ，或者输入你想启动的其他操作系统的名称。

一些Linux 发布在/etc/lilo 目录下有一个配置文件（一些版本的 Linux 的配置文件是/etc/lilo.conf），你可以按自己需要的启动信息编辑此文件。其他版本的Linux 的LILO 配置可以在安装过程中进行。如果是后一种情况，你可以使用 setup 功能来改变设置。

4.2.3 关闭Linux系统

你不应该简单地关闭电源以退出 Linux系统，因为这样可能对文件系统造成不可修复的破坏。Linux系统可以同时打开多个文件和多个进程，所以只有正确地关闭这些文件和进程后，你才可以关闭电源。

有几种方法可以关闭Linux系统。最正规的方法是使用 shutdown 命令。其格式是：

```
shutdown [minutes] [warning]
```

minutes 是关闭系统前需要等待的时间， warning 是想要向当前登录的每个用户显示的信息。

如果你希望立即关闭系统，你可以使用 halt 命令或者按Ctrl+Alt+Delete键。

4.3 挂接文件系统

只有挂接到Linux 的主文件系统后，文件系统才可以使用。即使是硬盘也得在挂接后才能使用。挂接文件系统的命令是mount 。

在启动过程中， mount 命令用来挂接/etc/fstab文件中的所有文件系统。在/etc/fstab文件中，每一个在启动过程中将要挂接的文件都有一个入口，用来描述其设备名，挂接目录（挂接点），文件系统的类型和其他信息。

你可以使用mount 命令从硬盘、光盘、软盘或者任何其他能提供 Linux支持的文件系统的设备添加一个新的文件系统。其格式如下：

```
mount filesystem mountpoint
```

filesystem 是设备名， mountpoint 是文件系统在Linux 中的挂接点。例如，你想把一个SCSI CD-ROM挂接到文件系统/usr/cdrom 下面，则使用如下的命令：

```
mount /dev/scd0 /usr/cdrom
```

/usr/cdrom 目录必须在挂接之前存在，否则 mount 命令将产生错误。当文件系统正确地挂接好了以后，只需进入/usr/cdrom 目录就可以存取CD-ROM 上的文件，就像CD-ROM 是你的文件系统的一部分。

如果/etc/fstab 文件中没有任何的文件系统入口，你只好用下面的命令挂接文件系统：

```
mount -t fstype filesystem mountpoint
```

fstype 是文件系统的类型。

4.3.1 挂接软盘

挂接软盘和挂接CD-ROM 的命令十分相似。可以使用如下的命令将软盘挂接到 /mnt目录下：

```
mount /dev/fd0 /mnt
```

如果想要挂接的文件系统不是 Linux系统的缺省类型，则你必须指明文件系统的类型。例如，想要挂接使用ext2 文件系统的软盘，则可以使用以下的命令：

```
mount -t ext2 /dev/fd0 /mnt
```

4.3.2 创建新的文件系统

如果想要在软盘上创建一个文件系统（这样才可以挂接软盘），你可以使用 mke2fs 或者mkdev fs命令，例如：

```
mke2fs /dev/fd0 1440
```

将一个文件创建成指定大小的文件系统，对于存储设备，相当于格式

用来在1.44MB 3.5英寸的软盘上创建文件系统。

4.3.3 卸载文件系统

如果想要从Linux 文件系统中卸载挂接的文件系统，则可以使用 umount 命令。例如，卸载/dev/fd0下面的软盘，则可以用如下的命令：

```
umount /dev/fd0
```

这样，软盘就可以从挂接点上卸载下来。

4.4 检查文件系统

有时文件可能被损坏或者文件系统的节点表没能和磁盘的内容保持一致。这时你可能需要检查文件系统。fsck 和e2fsck 都可以完成此任务（e2fsck 是为Linux系统的ext2fs 文件系统专门设计的）。

如果想用e2fsck 检查文件系统，可以使用如下的命令：

```
e2fsck -av /dev/hda1
```

此命令可以检查并修复 /dev/hda1 分区的错误。修复完成后，你应该重新启动计算机以便系统能够更新所做的修复。

4.5 使用文件作为交换区

在安装Linux系统时，你可能已经特意地设置了一个分区作为交换区。但你可以使用一个文件替代原先设置的交换分区，这样可以释放原先交换分区的空间。

一般情况下，使用文件作为交换分区可能时系统性能有所下降，虽然下降的幅度很小。但当你需要增加交换区时，此方法将会十分有用。使用如下的命令创建文件交换区：

```
dd if=/dev/zero of=/swap bs=1024 count=16416
```

此命令创建了一个大小为16MB（16416 blocks）的文件交换区（文件名叫做swap）。

接下来使用下面的命令物理地创建文件交换区：

```
mkswap /swap 16416
```

交换区的块数应该和上面相同。最后，使用如下的命令打开交换区：

```
swapon /swap
```

如果你想移去文件交换区而使用磁盘交换分区，则使用如下的命令：

```
swapoff /swap
```

然后再用rm 命令删除文件。

用作交换区的文件不能大于 16MB ，但你可以在一个系统中使用多达 8个的文件交换区和磁盘交换分区。

使用gzip 和compress压缩文件

compress 压缩的文件后缀名为.Z。你可以使用下面的命令压缩一个文件：

```
compress filename
```

也可以使用通配符同时压缩几个文件。缺省情况下，一个文件压缩后，未压缩的源文件将会被删除。

如果要解压缩一个压缩文件，可以使用如下命令：

```
uncompress filename
```

同样可以使用通配符*.Z 解压缩所有的压缩文件。记住，在指定文件名时要包括 .Z 后缀。

gzip 是一种新的压缩工具，它的压缩算法不同于 compress。使用gzip时，应指定文件名和

压缩类型：

```
gzip -9 filename
```

-9 选项用来告诉gzip 使用最高压缩因素，是最常用的选项。gzip 压缩文件的扩展名是.gz，并且压缩完毕后，源文件也将被删除。解压缩文件时使用 gunzip 或者 gzip -d filename。

使用 tar功能

tar 功能(tape archiver)在很多年以前就集成在UNIX 系统中了。但tar 功能的用户界面不友好，尤其当你不熟悉tar 功能的用法时。

tar 功能的目的是建立一个单一的文档文件，就像 DOS环境下的ZIP 功能一样。使用tar 可以将多个文件组合成为一个单一的大文件，这样就更加易于管理和备份。 tar 的用法如下：

```
tar [options] [file]
```

options 中包括很多选项，并且有时含混不清。文件名中可以使用通配符。下面是一个使用tar 的例子：

```
tar cvf archive1.tar /usr/tparker
```

此命令将/usr/tparker 目录下的所有文件组合成一个archive1.tar文件。c 选项用来指定建立一个文档，v 选项指示tar 在执行时显示提示信息，f 选项告诉tar 使用文件名archive1.tar 作为输出文件。

tar命令不会自动在文件后面加上扩展名.tar 。但你应当加上.tar 以便识别文档文件。

使用c 选项使tar 命令创建一个新的文档（如果文档已经存在，它将被删除）。v 选项用来把新文件添加到已存在的文档的后面。若文件不存在，则创建一个文件。 x 选项用来从文档中释放文件。例如：

```
tar xvf archive1.tar
```

没有必要指定文件名和路径名，因为文件名和路径名和以前一样。应当记住的是，路径名也存储在tar 文件中了。所以，如果把/usr/tparker 下的所以文件作成了一个文档，那么如果进入/usr/tparker 目录中使用tar xvf archive1.tar命令，则文件将被释放到/usr/tparker/usr/tparker目录下。

tar 命令在把文件组成文档后并不删除源文件。当文件释放后也不删除文档文件。

可以使用tar 命令把文件复制到磁带和软盘中。这时f 选项后面应该指定设备名。例如，把/usr/tparker 下的所有文件复制到软盘中，使用如下的命令：

```
tar cvf /dev/fd0 /usr/tparker
```

如果软盘没有足够的容量，则tar 命令将出现错误。但可以使用k 选项指定磁盘容量。例如，使用1.44MB 的软盘时，使用如下的命令：

```
tar cvfk /dev/fd0 1440 /usr/tparker
```

如果在复制完整个目录之前软盘已满，tar 则提示你使用另一张软盘。

也可以使用tar 命令存档压缩过的文件。同样也可以压缩存档文件。你会得到类似如下的文件：

```
filename.tar.gz
```

若想打开该文件，可以使用管道命令，也可以先使用 gunzip，再使用tar命令：

```
gunzip filename.tar.gz | tar xvf -
```

4.6 系统和文件的备份

可以使用tar 命令备份。例如，可以使用如下的命令把整个文件系统备份到软盘中：

```
tar -cvfbk /dev/fd0 1440 4 /
```

4.7 设置系统

4.7.1 设置系统名

系统名包含在/etc/HOSTNAME文件中。在网络环境中，系统名用来表示自己的计算机。

你可以/etc/HOSTNAME文件来改变系统名（只有在系统重新启动后，改名才会生效），也可以使用hostname 命令：

```
hostname hellfire
```

此命令把机器名设置为hellfire。

4.7.2 使用维护磁盘

每个系统都应有一个维护磁盘，以便用来检查根系统文件，恢复磁盘故障和解决一些简单的问题。应急（emergency ）磁盘，也叫做boot/root 磁盘，是在系统的设置改变时创建的，可以用来完成以上的任务。

在使用应急磁盘启动系统后，你可以使用 mount 命令挂接磁盘分区。

4.7.3 重新设置root 帐号口令

如果忘记了root 帐号口令，你可以使用boot/root 磁盘重新启动系统。然后挂接root 分区，编辑/etc/passwd文件，删除root 帐号下的口令，最后在从硬盘启动系统。系统启动后，你就可以设置新的口令了。

4.7.4 设置登录信息

如果系统上的用户多于一个，你可以使用/etc/motd 文件显示系统信息。此文件的内容在每个用户登录时都将显示。

第二篇　Linux高级语言及管理编程

第5章　外 壳 编 程

在DOS 中，你可能会从事一些例行的重复性命令，此时你会将这些重复性的命令写成批处理命令，只要执行这个批处理命令就等于执行这些命令。在 Linux系统中也有类似的批处理命令，它的功能比起DOS的批处理命令更为强大，相对也较为复杂，已经和一般的高级语言不相上下。这些批处理命令在Linux中叫做外壳脚本（外壳Script ）。

外壳脚本是以文本方式储存的，而非二进制文件。所以外壳脚本必须在 Linux系统的外壳下解释执行。不同外壳的脚本大多会有一些差异，所以我们不能将写给 A 外壳的脚本用B 外壳执行。而在Linux系统中大家最常使用是Bourne 外壳以及C 外壳，所以本章结合这两个外壳的相同点和不同点来介绍外壳编程。

5.1　创建和运行外壳程序

5.1.1　创建外壳程序

你可以用任何的编辑器编辑外壳程序，只需将要执行的外壳或 Linux命令写入外壳程序即可。例如，假设你的系统在启动时挂接有一台 CD-ROM 驱动器，而你想更换驱动器中的CD ，并读取其中的内容。一种方法是：首先把想要读取的 CD 放入 CD-ROM 驱动器中，然后使用unmount 命令卸载CD-ROM 驱动器，最后再使用mount 命令挂接CD-ROM 驱动器。命令如下：

```
umount /dev/cdrom
mount -t iso9660 /dev/cdrom /cdrom
```

你可以创建一个包含这两个命令的文件名为 remount 的外壳程序，而不必在每次更换 CD 都重复执行这两个命令。

5.1.2　运行外壳程序

如何运行我们已经写好的外壳脚本呢？可以有四种方法，下面分别介绍这几种方法：

1. 可以把外壳脚本的权限设置为可执行，这样就可以在外壳提示符下直接执行。我们可以使用下列命令更改外壳脚本的权限：

chmod u+x filename	只有自己可以执行，其他人不能执行。
chmod ug+x filename	只有自己以及同一工作组的人可以执行，其他人不能执行。
chmod +x filename	所有人都可以执行。

而我们如何指定使用哪一个外壳来解释执行外壳脚本呢？几种基本的指定方式如下所述：

1) 如果外壳脚本的第一个非空白字符不是“ # ”，则它会使用Bourne 外壳。

2) 如果外壳脚本的第一个非空白字符是“ # ”，但不以“ #! ”开头时，则它会使用C 外壳。

3) 如果外壳脚本以“ #! ”开头，则“ #! ”后面所跟的字符串就是所使用的外壳的绝对路径

名。Bourne 外壳的路径名称为/bin/sh ，而C 外壳则为/bin/csh。

例如：

1) 如使用Bourne 外壳，可用以下方式：

```
echo enter filename
```

或者

```
#!/bin/sh
```

2) 如使用C 外壳，可用以下方式：

```
# C 外壳Script
```

或者

```
#!/bin/csh
```

3) 如使用/etc/perl 作为外壳，可用以下方式：

```
#! /etc/perl
```

2. 第二种方法是执行外壳脚本想要执行的外壳，其后跟随外壳脚本的文件名作为命令行参数。例如，使用tcsh 执行上面的外壳脚本：

```
tcsh remount
```

此命令启动一个新的外壳，并令其执行remount 文件。

3. 第三种方法是在pdksh 和bash下使用. 命令，或在tcsh下使用source 命令。例如，在pdksh和bash下执行上面的外壳脚本：

```
.remount
```

或在tcsh下执行上面的外壳脚本：

```
source remount
```

4. 第四种方法是使用命令替换。

这是一个相当有用的方法。如果想要使某个命令的输出成为另一个命令的参数时，就可以使用这个方法。我们将命令列于两个 ` 号之间，而外壳会以这个命令执行后的输出结果代替这个命令以及两个 ` 符号。例如：

```
str='Current directory is `pwd`
echo $str
```

结果如下：

```
Current directory is /users/cc/mgtsai
```

在上面的例子中，pwd 这个命令输出/users/cc/mgtsai，而后整个字符串代替原来的‘ pwd ’设定str 变量，所以str 变量的内容则会包括pwd 命令的输出。

5.2 使用外壳变量

就像其他的任何高级语言一样，在外壳脚本中使用变量也是十分重要的。

5.2.1 给变量赋值

在pdksh 和bash中，给变量赋值的方法是一样的，既在变量名后跟着等号和变量值。例如，想要把5赋给变量count，则使用如下的命令：

```
count=5      （注意，在等号的两边不能有空格）
```

在tcsh中，可以使用如下的命令：

```
set count = 5
```

因为外壳语言是一种不需要类型检查的解释语言，所以在使用变量之前无须事先定义，这

和C 或 Pascal语言不一样。这也说明你可以使用同一个变量来存储字符串或整数。给字符串赋值的方法和给整数赋值的方法一样。例如：

```
name=Garry              （在pdksh 和bash中）
set name = Garry        （在tcsh中）
```

5.2.2 读取变量的值

可以使用$读取变量的值。例如，用如下的命令将 count 变量的内容输出到屏幕上：

```
echo $count
```

5.2.3 位置变量和其他系统变量

位置变量用来存储外壳程序后面所跟的参数。第一个参数存储在变量 1中，第二个参数存储在变量2中，依次类推。这些变量为系统保留变量，所以你不能为这些变量赋值。同样，你可以使用$来读取这些变量的值。例如，你可以编写一个外壳程序 reverse，执行过程中它有两个变量。输出时，将两个变量的位置颠倒。

```
#program reverse，  prints the command line parameters out in reverse order
echo "$2" "$1"
```

在外壳下执行此外壳程序：

```
reverse hello there
```

其输出如下：

```
there hello
```

除了位置变量以外，还有其他的一些系统变量，下面分别加以说明：

• 有些变量在启动外壳时就已经存在于系统中，你可以使用这些系统变量，并且可以赋予新值：

$HOME	用户自己的目录。
$PATH	执行命令时所搜寻的目录。
$TZ	时区。
$MAILCHECK	每隔多少秒检查是否有新的邮件。
$PS1	在外壳命令行的提示符。
$PS2	当命令尚未打完时，外壳要求再输入时的提示符。
$MANPATHman	指令的搜寻路径。

• 有些变量在执行外壳程序时系统就设置好了，并且你不能加以修改：

$#	存储外壳程序中命令行参数的个数。
$?	存储上一个执行命令的返回值。
$0	存储外壳程序的程序名。
$*	存储外壳程序的所有参数。
“ $@ ”	存储所有命令行输入的参数，分别表示为(“ $1 ” “ $2 ” ...)。
$$	存储外壳程序的PID。
$!	存储上一个后台执行命令的PID。

5.2.4 引号的作用

在外壳编程中，各种不同的引号之间的区别是十分重要的。单引号（‘ ’）、双引号（“”）和

反斜杠（\）都用作转义。

• 这三者之中，双引号的功能最弱。当你把字符串用双引号括起来时，外壳将忽略字符串中的空格，但其他的字符都将继续起作用。双引号在将多于一个单词的字符串赋给一个变量时尤其有用。例如，把字符串hello there赋给变量greeting时，应当使用下面的命令：

```
greeting="hello there"              （在 bash 和pdksh环境下）
set greeting = "hello there"        （在 tcsh环境下）
```

这两个命令将hello there作为一个单词存储在greeting 变量中。如果没有双引号， bash 和pdksh将产生语法错，而tcsh则将hello 赋给变量greeting 。

• 单引号的功能则最强。当你把字符串用单引号括起来时，外壳将忽视所有单引号中的特殊字符。例如，如果你想把登录时的用户名也包括在 greeting变量中，应该使用下面的命令：

```
greeting="hello there $LOGNAME"          （在 bash 和pdksh环境下）
set greeting="hello there $LOGNAME"      （在 tcsh环境下）
```

这将会把hello there root存储在变量greeting 中，如果你是以root 身份登录的话。但如果你在上面使用单引号，则单引号将会忽略 $符号的真正作用，而把字符串hello there $LOGNAME存储在greeting 变量中。

• 使用反斜杠是第三种使特殊字符发生转义的方法。反斜杠的功能和单引号一样，只是反斜杠每次只能使一个字符发生转义，而不是使整个字符串发生转义。请看下面的例子：

```
greeting=hello\ there       (在bash和pdksh环境下)
set greeting=hello\ there   (在tcsh环境下)
```

在命令中，反斜杠使外壳忽略空格，从而将hello there作为一个单词赋予变量greeting。

当你想要将一个特殊的字符包含在一个字符串中时，反斜杠就会特别地有用。例如，你想把一盒磁盘的价格$5.00赋予变量disk_price，则使用如下的命令：

```
disk_price=\$5.00       (在 bash 和pdksh环境下)
set disk_price = \$5.00  (在tcsh环境下)
```

如果没有反斜杠，外壳就会试图寻找变量5，并把变量5的值赋给disk_price。

5.3 数值运算命令

如果需要处理数值运算，我们可以使用 expr命令，下面列出可以使用的数值运算符及其用法：

```
expr expression
```

说明：

expression是由字符串以及运算符所组成的，每个字符串或是运算符之间必须用空格隔开。下面列出了运算符的种类及功能，运算符的优先顺序以先后次序排列，可以利用小括号来改变运算的优先次序。其运算结果输出到标准输出设备上。

: 字符串比较。比较的方式是以两字符串的第一个字母开始，以第二个字符串的最后一个字母结束。如果相同，则输出第二个字串的字母个数，如果不同则返回 0 。

* 乘法

/ 除法

% 取余数

+ 加法

- 减法

< 小于

<= 小于等于
= 等于
!= 不等于
>= 大于等于
> 大于
& AND运算
| OR运算

注意 当expression中含有*、(、)等符号时，必须在其前面加上\，以免被外壳解释成其他意义。

例如：

```
expr 2 \* \( 3 + 4 \)
```

输出结果为14。

test命令

在bash 和pdksh环境中，test命令用来测试条件表达式。其用法如下：

```
test expression
```

或者

```
[ expression ]
```

test命令可以和多种系统运算符一起使用。这些运算符可以分为四类：整数运算符、字符串运算符、文件运算符和逻辑运算符。

1) 整数运算符

int1 -eq int2	如果int1 和int2相等，则返回真。
int1 -ge int2	如果int1 大于等于int2，则返回真。
int1 -gt int2	如果int1 大于int2，则返回真。
int1 -le int2	如果int1小于等于int2，则返回真。
int1 -lt int2	如果int1小于int2，则返回真。
int1 -ne int2	如果int1 不等于int2，则返回真。

2) 字符串运算符

str1 = str2	如果str1 和str2相同，则返回真。
str1 != str2	如果str1 和str2不相同，则返回真。
str	如果str 不为空，则返回真。
-n str	如果str 的长度大于零，则返回真。
-z str	如果str 的长度等于零，则返回真。

3) 文件运算符

-d filename	如果filename 为目录，则返回真。
-f filename	如果filename 为普通的文件，则返回真。
-r filename	如果filename 可读，则返回真。
-s filename	如果filename 的长度大于零，则返回真。
-w filename	如果filename 可写，则返回真。
-x filename	如果filename 可执行，则返回真。

4) 逻辑运算符

! expr	如果expr 为假，则返回真。

expr1 -a expr2　　如果expr1 和expr2同时为真，则返回真。
expr1 -o expr2　　如果expr1 或 expr2有一个为真，则返回真。

tcsh中没有test命令，但它同样支持表达式。 tcsh支持的表达式形式基本上和 C语言一样。这些表达式大多数用在if 和while命令中。

tcsh表达式的运算符也分为整数运算符、字符串运算符、文件运算符和逻辑运算符四种。

1) 整数运算符

int1 <= int2　　如果int1小于等于int2，则返回真。
int1 >= int2　　如果int1 大于等于int2，则返回真。
int1 < int2　　如果int1小于等于int2，则返回真。
int1 > int2　　如果int1 大于int2，则返回真。

2) 字符串运算符

str1 == str2　　如果str1 和str2相同，则返回真。
str1 != str2　　如果str1 和str2不相同，则返回真。

3) 文件运算符

-r file　　如果file可读，则返回真。
-w file　　如果file可写，则返回真。
-x file　　如果file可执行，则返回真。
-e file　　如果file存在，则返回真。
-o file　　如果当前用户拥有file ，则返回真。
-z file　　如果file 长度为零，则返回真。
-f file　　如果file 为普通文件，则返回真。
-d file　　如果file 为目录，则返回真。

4) 逻辑运算符

exp1 || exp2　　如果exp1 为真或exp2 为真，则返回真。
exp1 && exp2　　如果exp1 和exp2同时为真，则返回真。
! exp　　如果exp 为假，则返回真。

5.4 条件表达式

bash、pdksh和tcsh 都有两种条件表达方法，即if 表达式和case 表达式。

5.4.1 if 表达式

bash、pdksh和tcsh 都支持嵌套的if...then...else 表达式。bash 和pdksh 的if 表达式如下：

```
if [ expression ]
then
commands
    elif [ expression2 ]
then
commands
else
commands
fi
```

elif 和else 在if 表达式中均为可选部分。elif是else if的缩写。只有在if表达式和任何在它之

前的elif表达式都为假时，才执行elif。fi关键字表示if表达式的结束。

在tcsh中，if表达式有两种形式。第一种形式为：

```
if (expression1) then
commands
else if (expression2) then
commands
else
commands
endif
```

tcsh的第二种形式是第一种形式的简写。它只执行一个命令，如果表达式为真，则执行，如果表达式为假，则不做任何事。其用法如下：

```
if (expression) command
```

下面是一个bash 或 pdksh 环境下if 表达式的例子。它用来查看在当前目录下是否存在一个叫.profile的文件：

```
if [ -f .profile ]
then
echo "There is a .profile file in the current directory."
else
echo "Could not find the .profile file."
fi
```

在tcsh环境下为：

```
#
if ( { -f .profile } ) then
echo "There is a .profile file in the current directory."
else
echo "Could not find the .profile file."
endif
```

5.4.2 case 表达式

case表达式允许你从几种情况中选择一种情况执行。外壳中的 case表达式的功能要比Pascal或C语言的case 或switch语句的功能稍强。这是因为在外壳中，你可以使用 case表达式比较带有通配符的字符串，而在Pascal 和C语言中你只能比较枚举类型和整数类型的值。

bash 和pdksh的case表达式如下：

```
case string1 in
str1)
commands;;
str2)
commands;;
*)
commands;;
esac
```

在此，将string1 和str1、str2比较。如果str1 和str2中的任何一个和strings1相符合，则它下面的命令一直到两个分号(;;)将被执行。如果str1 和str2中没有和strings1相符合的，则星号（*）下面的语句被执行。星号是缺省的case条件，因为它和任何字符串都匹配。

tcsh的选择语句称为开关语句。它和C语言的开关语句十分类似。

```
switch (string1)
case str1:
```

```
statements
breaksw
case str2:
statements
breaksw
default:
statements
breaksw
endsw
```

在此，string1和每一个case关键字后面的字符串相比较。如果任何一个字符串和 string1相匹配，则其后面的语句直到 breaksw将被执行。如果没有任何一个字符串和 string1匹配，则执行default后面直到breaksw的语句。

下面是bash 或 pdksh环境下case表达式的一个例子。

它检查命令行的第一个参数是否为-i 或e。如果是-i，则计算由第二个参数指定的文件中以i开头的行数。如果是 -e，则计算由第二个参数指定的文件中以 e开头的行数。如果第一个参数既不是 -i也不是 -e，则在屏幕上显示一条的错误信息。

```
case $1 in
-i)
count='grep ^i $2 | wc -l '
echo "The number of lines in $2 that start with an i is $count"
;;
-e)
count='grep ^e $2 | wc -l'
echo "The number of lines in $2 that start with an e is $count"
;;
* )
echo "That option is not recognized"
;;
esac
```

此例在tcsh 环境下为：

```
# remember that the first line must start with a # when using tcsh
switch ( $1 )
case -i | i:
set count = 'grep ^i $2 | wc -l'
echo "The number of lines in $2 that begin with i is $count"
breaksw
case -e | e:
set count = 'grep ^e $2 | wc -l'
echo "The number of lines in $2 that begin with e is $count"
breaksw
default:
echo "That option is not recognized"
breaksw
endsw
```

5.5 循环语句

外壳中提供了几种循环语句，最为常用的是for表达式。

5.5.1 for 语句

bash 和pdksh中有两种使用for 语句的表达式。

第一种形式是：

```
for var1 in list
do
commands
done
```

在此形式时，对在list 中的每一项，for语句都执行一次。List可以是包括几个单词的、由空格分隔开的变量，也可以是直接输入的几个值。每执行一次循环，var1都被赋予list中的当前值，直到最后一个为止。

第二种形式是：

```
for var1
do
statements
done
```

使用这种形式时，对变量var1中的每一项，for语句都执行一次。此时，外壳程序假定变量var1中包含外壳程序在命令行的所有位置参数。

一般情况下，此种方式也可以写成：

```
for var1 in "$@"
do
statements
done
```

在tcsh中，for循环语句叫做foreach。其形式如下：

```
foreach name (list)
commands
end
```

下面是一个在bash 或 pdksh环境下的例子。

此程序可以以任何数目的文本文件作为命令行参数。它读取每一个文件，把其中的内容转换成大写字母，然后将结果存储在以.caps作为扩展名的同样名字的文件中。

```
for file
do
tr a-z A-Z < $file >$file.caps
done
```

在tcsh环境下，此例子可以写成：

```
#
foreach file ($*)
tr a-z A-Z < $file >$file.caps
end
```

5.5.2 while 语句

while 语句是另一种循环语句。当一个给定的条件为真时，则一直循环执行下面的语句直到条件为假。在bash 和pdksh环境下，使用while 语句的表达式为：

```
while expression
do
statements
```

```
done
```

而在tcsh中，while 语句为：

```
while (expression)
statements
end
```

下面是在bash 和pdksh中while语句的一个例子。程序列出所带的所有参数，以及他们的位置号。

```
count=1
while [ -n "$*" ]
do
echo "This is parameter number $count $1"
shift
count='expr $count + 1'
done
```

其中shift命令用来将命令行参数左移一个。

在tcsh中，此例子为：

```
#
set count = 1
while ( "$*" != "" )
echo "This is parameter number $count $1"
shift
set count = 'expr $count + 1'
end
```

5.5.3 until 语句

until 语句的作用和while语句基本一样，只是当给定的条件为假时，执行 until 语句。until 语句在bash 和pdksh中的写法为：

```
until expression
do
commands
done
```

让我们用until语句重写上面的例子：

```
count=1
until [ -z "$*" ]
do
echo "This is parameter number $count $1"
shift
count='expr $count + 1'
done
```

在应用中，until语句不是很常用，因为until语句可以用while语句重写。

5.6 shift 命令

bash、 pdksh和tcsh都支持shift 命令。shift 命令用来将存储在位置参数中的当前值左移一个位置。例如当前的位置参数是：

```
$1 = -r $2 = file1 $3 = file2
```

执行shift 命令：

```
shift
```

位置参数将会变为：

```
$1 = file1 $2 = file2
```

你也可以指定shift 命令每次移动的位置个数。下面的例子将位置参数移动两个位置：

```
shift 2
```

下面是一个应用shift 命令的例子。此程序有两个命令行参数，一个代表输入文件，另一个代表输出文件。程序读取输入文件，将其中的内容转换成大写，并将结果存储在输出文件中。

```
while [ "$1" ]
do
if [ "$1" = "-i" ] then
infile="$2"
shift 2
elif [ "$1" = "-o" ]
then
outfile="$2"
shift 2
else
echo "Program $0 does not recognize option $1"
fi
done
tr a-z A-Z <$infile >$outfile
```

5.7 select 语句

select 语句只存在于pdksh中，在bash 或 tcsh中没有相似的表达式。select 语句自动生成一个简单的文字菜单。其用法如下：

```
select menuitem [in list_of_items]
do
commands
done
```

其中方括号中是select语句的可选部分。

当select语句执行时， pdksh为在list_of_items中的每一项创建一个标有数字的菜单项。list_of_items可以是包含几个条目的变量，就像choice1 choice2，或者是直接在命令中输入的选择项，例如：

```
select menuitem in choice1 choice2 choice3
```

如果没有list_of_items，select语句则使用命令行的位置参数，就像for表达式一样。

一旦你选择了菜单项中的一个， select语句就选中的菜单项的数字值存储在变量 menuitem 中。然后你可以利用do中的语句来执行选中的菜单项要执行的命令。

下面是select语句的一个例子。

```
select menuitem in pick1 pick2 pick3
do
echo "Are you sure you want to pick $menuitem"
read res
if [ $res = "y" -o $res = "Y" ]
then
break
fi
done
```

5.8 repeat 语句

repeat 语句只存在于tcsh中，在pdksh 和bash中没有相似的语句。repeat 语句用来使一个单一的语句执行指定的次数。repeat 语句如下：

```
repeat count command
```

下面给出repeat 语句的一个例子。它读取命令行后的一串数字，并根据数字在屏幕上分行输出句号。

```
#
foreach num ($*)
repeat $num echo -n "."
echo ""
end
```

任何repeat 语句都可以用while 或 for语句重写。repeat语句只是更加方便而已。

5.9 子函数

外壳语言可以定义自己的函数，就像在C 或其他语言中一样。使用函数的最大好处就是程序更为清晰可读。下面是如何在bash 和pdksh中创建一个函数：

```
fname () {
shellcommands
}
```

在pdksh中也可以使用如下的形式：

```
function fname {
shellcommands
}
```

使用函数时，只须输入以下的命令：

```
fname [parm1 parm2 parm3 ...]
```

tcsh 外壳中不支持函数。

你可以传递任何数目的参数给一个函数。函数将会把这些参数视为位置参数。请看下面的例子。

此例子包括四个函数：upper ()、lower ()、print ()和usage_error ()，他们的任务分别是：将文件转换成大写字母、将文件转换成小写字母、打印文件内容和显示出错信息。 upper ()、lower ()、print ()都可以有任意数目的参数。如果将此例子命名为 convert，你可以在外壳提示符下这样使用该程序：convert -u file1 file2 file3。

```
upper () {
shift
for i
do
tr a-z A-Z <$1 >$1.out
rm $1
mv $1.out $1
shift
done; }
lower () {
shift
for i
do
```

```
tr A-Z a-z <$1 >$1.out
rm $1
mv $1.out $1
shift
done; }
print () {
shift
for i
do
lpr $1
shift
done; }
usage_error () {
echo "$1 syntax is $1 <option> <input files>"
echo ""
echo "where option is one of the following"
echo "p — to print frame files"
echo "u — to save as uppercase"
echo "l — to save as lowercase"; }
case $1
in
p | -p) print $@;;
u | -u) upper $@;;
l | -l) lower $@;;
*) usage_error $0;;
esac
```

虽然外壳语言功能强大而且简单易学，但你会发现在有些情况下，外壳语言无法解决你的问题。这时，你可以选择Linux系统中的其他语言，例如C 和C++、gawk、以及Perl等。

第6章 gawk语言编程

awk是一种程序语言，对文档资料的处理具有很强的功能。 awk 名称是由它三个最初设计者的姓氏的第一个字母而命名的：Alfred V. Aho、 Peter J. Weinberger、 Brian W. Kernighan。awk最初在1977年完成。1985年发表了一个新版本的awk，它的功能比旧版本增强了不少。awk能够用很短的程序对文档里的资料做修改、比较、提取、打印等处理。如果使用 C 或 Pascal等语言编写程序完成上述的任务会十分不方便而且很花费时间，所写的程序也会很大。

awk不仅仅是一个编程语言，它还是 Linux系统管理员和程序员的一个不可缺少的工具。awk语言本身十分好学，易于掌握，并且特别的灵活。

gawk 是GNU计划下所做的 awk，gawk 最初在1986年完成，之后不断地被改进、更新。gawk 包含 awk 的所有功能。

6.1 gawk的主要功能

gawk 的主要功能是针对文件的每一行 (line)，也就是每一条记录，搜寻指定的格式。当某一行符合指定的格式时，gawk 就会在此行执行被指定的动作。gawk 依此方式自动处理输入文件的每一行直到输入文件档案结束。

gawk经常用在如下的几个方面：

- 根据要求选择文件的某几行，几列或部分字段以供显示输出。
- 分析文档中的某一个字出现的频率、位置等。
- 根据某一个文档的信息准备格式化输出。
- 以一个功能十分强大的方式过滤输出文档。
- 根据文档中的数值进行计算。

6.2 如何执行gawk程序

基本上有两种方法可以执行gawk程序。

如果gawk 程序很短，则可以将gawk 直接写在命令行，如下所示：

```
gawk 'program' input-file1 input-file2 ...
```

其中 program 包括一些 pattern 和action。

如果gawk 程序较长，较为方便的做法是将 gawk 程序存在一个文件中，

gawk 的格式如下所示：

```
gawk -f program-file input-file1 input-file2 ...
```

gawk 程序的文件不止一个时，执行gawk 的格式如下所示：

```
gawk -f program-file1 -f program-file2 ... input-file1 input-file2 ...
```

6.3 文件、记录和字段

一般情况下，gawk可以处理文件中的数值数据，但也可以处理字符串信息。如果数据没有存储在文件中，可以通过管道命令和其他的重定向方法给 gawk提供输入。当然， gawk只能处理文本文件（ASCII码文件）。

电话号码本就是一个gawk可以处理的文件的简单例子。电话号码本由很多条目组成，每一个条目都有同样的格式：姓、名、地址、电话号码。每一个条目都是按字母顺序排列。

在gawk中，每一个这样的条目叫做一个记录。它是一个完整的数据的集合。例如，电话号码本中的Smith John这个条目，包括他的地址和电话号码，就是一条记录。

记录中的每一项叫做一个字段。在gawk中，字段是最基本的单位。多个记录的集合组成了一个文件。

大多数情况下，字段之间由一个特殊的字符分开，像空格、 TAB、分号等。这些字符叫做字段分隔符。请看下面这个/etc/passwd文件：

```
tparker;t36s62hsh;501;101;Tim Parker;/home/tparker;/bin/bash
etreijs;2ys639dj3h;502;101;Ed Treijs;/home/etreijs;/bin/tcsh
ychow;1h27sj;503;101;Yvonne Chow;/home/ychow;/bin/bash
```

你可以看出/etc/passwd文件使用分号作为字段分隔符。 /etc/passwd文件中的每一行都包括七个字段：用户名；口令；用户 ID；工作组ID；注释；home目录；启始的外壳。如果你想要查找第六个字段，只需数过五个分号即可。

但考虑到以下电话号码本的例子，你就会发现一些问题：

```
Smith John 13 Wilson St. 555-1283
Smith John 2736 Artside Dr Apt 123 555-2736
Smith John 125 Westmount Cr 555-1726
```

虽然我们能够分辨出每个记录包括四个字段，但gawk却无能为力。电话号码本使用空格作为分隔符，所以gawk认为Smith是第一个字段， John 是第二个字段，13是第三个字段，依次类推。就 gawk而言，如果用空格作为字段分隔符的话，则第一个记录有六个字段，而第二个记录有八个字段。

所以，我们必须找出一个更好的字段分隔符。例如，像下面一样使用斜杠作为字段分隔符：

```
Smith/John/13 Wilson St./555-1283
Smith/John/2736 Artside Dr/Apt/123/555-2736
Smith/John/125 Westmount Cr/555-1726
```

如果你没有指定其他的字符作为字段分隔符，那么 gawk将缺省地使用空格或TAB作为字段分隔符。

6.4 模式和动作

在gawk语言中每一个命令都由两部分组成：一个模式（ pattern）和一个相应的动作(action)。只要模式符合， gawk就会执行相应的动作。其中模式部分用两个斜杠括起来，而动作部分用一对花括号括起来。例如：

```
/pattern1/{action1}
/pattern2/{action2}
/pattern3/{action3}
```

所有的gawk程序都是由这样的一对对的模式和动作组成的。其中模式或动作都能够被省略，但是两个不能同时被省略。如果模式被省略，则对于作为输入的文件里面的每一行，动作都会被执行。如果动作被省略，则缺省的动作被执行，既显示出所有符合模式的输入行而不做任何的改动。

下面是一个简单的例子，因为 gawk 程序很短，所以将 gawk 程序直接写在外壳命令行：

```
gawk '/tparker/' /etc/passwd
```

此程序在上面提到的/etc/passwd文件中寻找符合tparker模式的记录并显示（此例中没有动作，所以缺省的动作被执行）。

让我们再看一个例子：

```
gawk '/UNIX/{print $2}' file2.data
```

此命令将逐行查找file2.data文件中包含UNIX的记录，并打印这些记录的第二个字段。

你也可以在一个命令中使用多个模式和动作对，例如：

```
gawk '/scandal/{print $1} /rumor/{print $2}' gossip_file
```

此命令搜索文件 gossip_file中包括scandal的记录，并打印第一个字段。然后再从头搜索gossip_file中包括rumor的记录，并打印第二个字段。

6.5 比较运算和数值运算

gawk有很多比较运算符，下面列出重要的几个：

==	相等
!=	不相等
>	大于
<	小于
>=	大于等于
<=	小于等于

例如：

```
gawk '$4 > 100' testfile
```

将会显示文件testfile 中那些第四个字段大于100的记录。

下表列出了gawk中基本的数值运算符。

运 算 符	说 明	示 例
+	加法运算	2+6
-	减法运算	6-3
*	乘法运算	2*5
/	除法运算	8/4
^	乘方运算	3^2 (=9)
%	求余数	9%4 (=1)

例如：

```
{print $3/2}
```

显示第三个字段被2除的结果。

在gawk中，运算符的优先权和一般的数学运算的优先权一样。例如：

```
{print $1+$2*$3}
```

显示第二个字段和第三个字段相乘，然后和第一个字段相加的结果。

你也可以用括号改变优先次序。例如：

```
{print ($1+$2)*$3}
```

显示第一个字段和第二个字段相加，然后和第三个字段相乘的结果。

6.6 内部函数

gawk中有各种的内部函数，现在介绍如下：

6.6.1 随机数和数学函数

sqrt(x)	求x 的平方根
sin(x)	求x 的正弦函数
cos(x)	求x 的余弦函数
atan2(x, y)	求x/y的余切函数
log(x)	求x 的自然对数
exp(x)	求x 的e 次方
int(x)	求x 的整数部分
rand()	求0 和1之间的随机数
srand(x)	将 x 设置为 rand()的种子数

6.6.2 字符串的内部函数

- index(in, find) 在字符串 in 中寻找字符串 find 第一次出现的地方，返回值是字符串 find 出现在字符串 in 里面的位置。如果在字符串 in 里面找不到字符串 find，则返回值为 0。

例如：

```
print index("peanut", "an")
```

显示结果3。

- length(string) 求出 string 有几个字符。

例如：

```
length("abcde")
```

显示结果5。

- match(string, regexp) 在字符串 string 中寻找符合 regexp 的最长、最靠左边的子字符串。返回值是 regexp 在 string 的开始位置，即 index值。match 函数将会设置系统变量 RSTART 等于index的值，系统变量RLENGTH 等于符合的字符个数。如果不符合，则会设置 RSTART 为0、RLENGTH 为 -1。
- sprintf(format, expression1, ...) 和printf 类似，但是 sprintf 并不显示，而是返回字符串。

例如：

```
sprintf("pi = %.2f (approx.)", 22/7)
```

返回的字符串为pi = 3.14 (approx.)

- sub(regexp, replacement, target) 在字符串 target 中寻找符合 regexp 的最长、最靠左的地方，以字串 replacement 代替最左边的 regexp。

例如：

```
str = "water, water, everywhere"
sub(/at/, "ith", str)
```

结果字符串str会变成

```
wither, water, everywhere
```

- gsub(regexp, replacement, target) 与前面的sub类似。在字符串 target 中寻找符合 regexp的所有地方，以字符串 replacement 代替所有的regexp。

例如：

```
str="water, water, everywhere"
```

gsub(/at/, "ith", str)

结果字符串str会变成

wither, wither, everywhere

• substr(string, start, length) 返回字符串 string 的子字符串，这个子字符串的长度为 length，从第 start 个位置开始。

例如：

substr("washington", 5, 3)

返回值为ing

如果没有length ，则返回的子字符串是从第 start 个位置开始至结束。

例如：

substr("washington", 5)

返回值为ington。

• tolower(string) 将字符串string的大写字母改为小写字母。

例如：

tolower("MiXeD cAsE 123")

返回值为mixed case 123。

• toupper(string) 将字符串string的小写字母改为大写字母。

例如：

toupper("MiXeD cAsE 123")

返回值为MIXED CASE 123。

6.6.3 输入输出的内部函数

• close(filename) 将输入或输出的文件 filename 关闭。

• system(command) 此函数允许用户执行操作系统的指令，执行完毕后将回到 gawk程序。

例如：

BEGIN {system("ls")}

6.7 字符串和数字

字符串就是一连串的字符，它可以被gawk逐字地翻译。字符串用双引号括起来。数字不能用双引号括起来，并且gawk将它当作一个数值。例如：

gawk '$1 != "Tim" {print}' testfile

此命令将显示第一个字段和 Tim不相同的所有记录。如果命令中 Tim两边不用双引号，gawk将不能正确执行。

再如：

gawk '$1 == "50" {print}' testfile

此命令将显示所有第一个字段和 50这个字符串相同的记录。 gawk不管第一字段中的数值的大小，而只是逐字地比较。这时，字符串50和数值50并不相等。

6.8 格式化输出

我们可以让动作显示一些比较复杂的结果。例如：

gawk '$1 != "Tim" {print $1, $5, $6, $2}' testfile

将显示testfile文件中所有第一个字段和 Tim不相同的记录的第一、第五、第六和第二个字段。

进一步，你可以在print动作中加入字符串，例如：

```
gawk '$1 != "Tim" {print "The entry for ", $1, "is not Tim. ", $2}' testfile
```

print动作的每一部分用逗号隔开。

借用C语言的格式化输出指令，可以让gawk的输出形式更为多样。这时，应该用printf而不是print。例如：

```
{printf "%5s likes this language\n", $2}
```

printf中的%5s 部分告诉gawk 如何格式化输出字符串，也就是输出 5个字符长。它的值由printf 的最后部分指出，在此是第二个字段。 \n是回车换行符。如果第二个字段中存储的是人名，则输出结果大致如下：

```
Tim likes this language
Geoff likes this language
Mike likes this language
Joe likes this language
```

gawk 语言支持的其他格式控制符号如下：

- c 如果是字符串，则显示第一个字符；如果是整数，则将数字以ASCII 字符的形式显示。

例如：

printf “ %c ”, 65

结果将显示字母A。

- d 显示十进制的整数。
- i 显示十进制的整数。
- e 将浮点数以科学记数法的形式显示。

例如：

print “ $4.3e ”, 1950

结果将显示1.950e+03。

- f 将数字以浮点的形式显示。
- g 将数字以科学记数法的形式或浮点的形式显示。数字的绝对值如果大于等于 0.0001则以浮点的形式显示，否则以科学记数法的形式显示。
- o 显示无符号的八进制整数。
- s 显示一个字符串。
- x 显示无符号的十六进制整数。10至15以a至f表示。
- X 显示无符号的十六进制整数。10至15以A至F表示。
- % 它并不是真正的格式控制字符，%%将显示%。

当你使用这些格式控制字符时，你可以在控制字符前给出数字，以表示你将用的几位或几个字符。例如，6d表示一个整数有6位。再请看下面的例子：

```
{printf "%5s works for %5s and earns %2d an hour", $1, $2, $3}
```

将会产生类似如下的输出：

```
Joe works for Mike and earns 12 an hour
```

当处理数据时，你可以指定数据的精确位数

```
{printf "%5s earns $%.2f an hour", $3, $6}
```

其输出将类似于：

```
Joe earns $12.17 an hour
```

你也可以使用一些换码控制符格式化整行的输出。之所以叫做换码控制符，是因为 gawk对这些符号有特殊的解释。下面列出常用的换码控制符：

\a 警告或响铃字符。
\b 后退一格 。
\f 换页。
\n 换行。
\r 回车。
\t Tab。
\v 垂直的 tab。

6.9 改变字段分隔符

在gawk中，缺省的字段分隔符一般是空格符或 TAB。但你可以在命令行使用-F选项改变字符分隔符，只需在-F后面跟着你想用的分隔符即可。

```
gawk -F";"'/tparker/{print}' /etc/passwd
```

在此例中，你将字符分隔符设置成分号。注意： -F必须是大写的，而且必须在第一个引号之前。

6.10 元字符

gawk语言在格式匹配时有其特殊的规则。例如， cat能够和记录中任何位置有这三个字符的字段匹配。但有时你需要一些更为特殊的匹配。如果你想让 cat只和concatenate匹配，则需要在格式两端加上空格：

```
/ cat / {print}
```

再例如，你希望既和cat又和CAT匹配，则可以使用或（|）：

```
/ cat | CAT / {print}
```

在gawk中，有几个字符有特殊意义。下面列出可以用在 gawk格式中的这些字符：

• ^ 表示字段的开始。

例如：

```
$3 ~ /^b/
```

如果第三个字段以字符b开始，则匹配。

• $ 表示字段的结束。

例如：

```
$3 ~ /b$/
```

如果第三个字段以字符b结束，则匹配。

• . 表示和任何单字符m匹配。

例如：

```
$3 ~ /i.m/
```

如果第三个字段有字符i，则匹配。

• | 表示 “ 或 ”。

例如：

```
/cat|CAT/
```

和cat 或 CAT字符匹配。

• * 表示字符的零到多次重复。

例如：

/UNI*X/

和UNX、UNIX、UNIIX、UNIIIX等匹配。

• + 表示字符的一次到多次重复。

例如：

/UNI+X/

和UNIX、UNIIX等匹配。

• \{a，b\} 表示字符a次到b次之间的重复。

例如：

/UNI\{1，3\}X

和UNIX、UNIIX和UNIIIX匹配。

• ? 表示字符零次和一次的重复。

例如：

/UNI?X/

和UNX 和UNIX匹配。

• [] 表示字符的范围。

例如：

/I[BDG]M/

和IBM、IDM和IGM匹配

• [^] 表示不在[]中的字符。

例如：

/I[^DE]M/

和所有的以I开始、M结束的包括三个字符的字符串匹配，除了IDM和IEM之外。

6.11 调用gawk程序

当需要很多对模式和动作时，你可以编写一个gawk程序（也叫做gawk脚本）。在gawk程序中，你可以省略模式和动作两边的引号，因为在 gawk程序中，模式和动作从哪开始和从哪结束时是很显然的。

你可以使用如下命令调用gawk程序：

```
gawk -f script filename
```

此命令使gawk对文件filename执行名为script的gawk程序。

如果你不希望使用缺省的字段分隔符，你可以在 f选项后面跟着F选项指定新的字段分隔符（当然你也可以在gawk程序中指定），例如，使用分号作为字段分隔符：

```
gawk -f script -F";" filename
```

如果希望gawk 程序处理多个文件，则把各个文件名罗列其后：

```
gawk -f script filename1 filename2 filename3 ...
```

缺省情况下， gawk的输出将送往屏幕。但你可以使用 Linux的重定向命令使gawk的输出送往一个文件：

```
gawk -f script filename > save_file
```

6.12 BEGIN和END

有两个特殊的模式在gawk中非常有用。 BEGIN模式用来指明gawk开始处理一个文件之前

执行一些动作。BEGIN经常用来初始化数值，设置参数等。 END模式用来在文件处理完成后执行一些指令，一般用作总结或注释。

BEGIN 和END中所有要执行的指令都应该用花括号括起来。BEGIN 和END必须使用大写。请看下面的例子：

```
BEGIN { print "Starting the process the file" }
$1 == "UNIX" {print}
$2 > 10 {printf "This line has a value of %d", $2}
END { print "Finished processing the file. Bye!"}
```

此程序中，先显示一条信息： Starting the process the file，然后将所有第一个字段等于UNIX的整条记录显示出来，然后再显示第二个字段大于 10 的记录，最后显示信息：Finished processing the file. Bye!。

6.13 变量

在gawk中，可以用等号(=)给一个变量赋值：

```
var1 = 10
```

在gawk中，你不必事先声明变量类型。

请看下面的例子：

```
$1 == "Plastic" { count = count + 1 }
```

如果第一个字段是Plastic，则count的值加1。在此之前，我们应当给count赋予过初值，一般是在BEGIN部分。

下面是比较完整的例子：

```
BEGIN { count = 0 }
$5 == "UNIX" { count = count + 1 }
END { printf "%d occurrences of UNIX were found", count }
```

变量可以和字段和数值一起使用，所以，下面的表达式均为合法：

```
count = count + $6
count = $5 - 8
count = $5 + var1
```

变量也可以是格式的一部分，例如：

```
$2 > max_value {print "Max value exceeded by ", $2 - max_value}
$4 - var1 < min_value {print "Illegal value of ", $4}
```

6.14 内置变量

gawk语言中有几个十分有用的内置变量，现在列于下面：

NR	已经读取过的记录数。
FNR	从当前文件中读出的记录数 。
FILENAME	输入文件的名字。
FS	字段分隔符 （缺省为空格）。
RS	记录分隔符 （缺省为换行）。
OFMT	数字的输出格式（缺省为%g）。
OFS	输出字段分隔符 。
ORS	输出记录分隔符 。
NF	当前记录中的字段数 。

如果你只处理一个文件，则 NR 和FNR 的值是一样的。但如果是多个文件， NR是对所有的文件来说的，而FNR 则只是针对当前文件而言。

例如：

```
NR <= 5 {print "Not enough fields in the record"}
```

检查记录数是否小于5，如果小于5，则显示出错信息。

FS十分有用，因为FS控制输入文件的字段分隔符。例如，在 BEGIN格式中，使用如下的命令：

```
FS=":"
```

6.15 控制结构

6.15.1 if 表达式

if 表达式的语法如下：

```
if (expression){
commands
}
else{
commands
}
```

例如：

```
# a simple if loop
 (if ($1 == 0){
print "This cell has a value of zero"
}
else {
printf "The value is %d\n", $1
})
```

再看下一个例子：

```
# a nicely formatted if loop
 (if ($1 > $2){
print "The first column is larger"
}
else {
print "The second column is larger"
})
```

6.15.2 while 循环

while 循环的语法如下：

```
while (expression){
commands
}
```

例如：

```
# interest calculation computes compound interest
# inputs from a file are the amount, interest_rateand years
{var = 1
while (var <= $3) {
printf("%f\n", $1*(1+$2)^var)
var++
```

```
}
}
```

6.15.3 for 循环

for 循环的语法如下：

```
for (initialization; expression; increment) {
command
}
```

例如：

```
# interest calculation computes compound interest
# inputs from a file are the amount，interest_rateand years
{for (var=1; var <= $3; var++) {
printf("%f\n"，$1*(1+$2)^var)
}
}
```

6.15.4 next 和exit

next 指令用来告诉gawk 处理文件中的下一个记录， 而不管现在正在做什么。语法如下：

```
{ command1
command2
command3
next
command4
}
```

程序只要执行到next指令，就跳到下一个记录从头执行命令。因此，本例中， command4 指令永远不会被执行。

程序遇到exit指令后，就转到程序的末尾去执行END，如果有END的话。

6.16 数组

gawk语言支持数组结构。数组不必事先初始化。声明一个数组的方法如下：

```
arrayname[num]=value
```

请看下面的例子：

```
# reverse lines in a file
{line[NR] = $0 } # remember each line
END {var=NR # output lines in reverse order
while (var > 0){
print line[var]
var--
}
}
```

此段程序读取一个文件的每一行，并用相反的顺序显示出来。我们使用 NR作为数组的下标来存储文件的每一条记录，然后在从最后一条记录开始，将文件逐条地显示出来。

6.17 用户自定义函数

复杂的 gawk 程序常常可以使用自己定义的函数来简化。调用用户自定义函数与调用内部函数的方法一样。函数的定义可以放在 gawk 程序的任何地方。

用户自定义函数的格式如下：

```
function name (parameter-list) {
  body-of-function
}
```

name 是所定义的函数的名称。一个正确的函数名称可包括一序列的字母、数字、下标线(underscores)，但是不可用数字做开头。parameter-list 是函数的全部参数的列表，各个参数之间以逗点隔开。body-of-function 包含 gawk 的表达式，它是函数定义里最重要的部分，它决定函数实际要做的事情。

下面这个例子，会将每个记录的第一个字段的值的平方与第二个字段的值的平方加起来。

```
{print "sum =", SquareSum($1, $2)}
function SquareSum(x, y) {
sum=x*x+y*y
return sum
}
```

到此，我们已经知道了gawk的基本用法。gawk语言十分易学好用，例如，你可以用 gawk 编写一段小程序来计算一个目录中所有文件的个数和容量。如果用其他的语言，如 C语言，则会十分的麻烦，相反，gawk只需要几行就可以完成此工作。

6.18 几个实例

最后，再举几个gawk的例子：

```
gawk '{if (NF > max) max = NF}
  END {print max}'
```

此程序会显示所有输入行之中字段的最大个数。

```
gawk 'length($0) > 80'
```

此程序会显示出超过 80 个字符的每一行。此处只有模式被列出，动作是采用缺省值显示整个记录。

```
gawk 'NF > 0'
```

显示拥有至少一个字段的所有行。这是一个简单的方法，将一个文件里的所有空白行删除。

```
gawk 'BEGIN {for (i = 1; i <= 7; i++)
print int(101 * rand())}'
```

此程序会显示出范围是 0 到 100 之间的 7 个随机数。

```
ls -l files | gawk '{x += $4}; END {print "total bytes: " x}'
```

此程序会显示出所有指定的文件的总字节数。

```
expand file | gawk '{if (x < length()) x = length()}
END {print "maximum line length is " x}'
```

此程序会将指定文件里最长一行的长度显示出来。expand 会将 tab 改成 space，所以是用实际的右边界来做长度的比较。

```
gawk 'BEGIN {FS = ":"}
  {print $1 | "sort"}' /etc/passwd
```

此程序会将所有用户的登录名称，依照字母的顺序显示出来。

```
gawk '{nlines++}
  END {print nlines}'
```

此程序会将一个文件的总行数显示出来。

```
gawk 'END {print NR}'
```

此程序也会将一个文件的总行数显示出来，但是计算行数的工作由 gawk来做。

```
gawk '{print NR, $0}'
```

此程序显示出文件的内容时，会在每行的最前面显示出行号，它的函数与‘ cat -n ’类似。

第7章　Perl语言编程

本章介绍有关Perl语言编程方面的内容。

7.1　什么是Perl

Perl(Practical Extraction and Report Language)是一种解释性的语言，专门为搜索纯文本文件而做了优化。它也可以十分方便地完成很多系统管理任务。它集成了 C、sed、awk和sh语言的优点，可以运行于Linux、UNIX、MVS、VMS、MS-DOS、Macintosh、OS/2、Amiga以及其他的一些操作系统上。特别是近年来，随着 Internet 的普及，Perl也越来越多地用于World Wide Web上CGI 等的编程，逐渐成为系统、数据库和用户之间的桥梁。

7.2　Perl的现状

大概有两种程序员喜欢用Perl：系统程序员可以用Perl结合系统命令一起处理数据和过程，并且可以使用Perl的格式匹配函数进行系统信息的搜寻和总结；还有一些开发 UNIX Web服务器CGI程序的程序员发现Perl比C语言易学易用，而且更加容易处理数据库和数据搜索。

Perl的创建人Larry Wall在1994年10月发表了Perl的第5版本。Perl 5增加了面向对象的能力，提供了更多的数据结构，与系统和数据库之间的新的标准接口以及其他的一些功能。

7.3　初试Perl

一个有用的Perl程序可以很短。例如我们希望更换大量文件中的一些相同内容，可以使用下面的一条命令：

```
perl -e 's/gopher/World Wide Web/gi' -p -i.bak *.html
```

下面是一个基本的perl程序：

```
#!/usr/local/bin/perl
#
# Program to do the obvious
#
print 'Hello world.';# Print a message
```

每个perl程序都以#!/usr/local/bin/perl开始，这样系统的外壳知道应该使用perl运行该程序。用#表示此后为注释语句。Perl的表达式必须以分号结尾，就如同C语言一样。此语句为显示语句，只是简单地显示出Hello world.字符串。

7.4　Perl变量

Perl中有三种变量：标量，数组（列表）和相关数组。

7.4.1　标量

Perl中最基本的变量类型是标量。标量既可以是数字，也可以是字符串，而且两者是可以互换的。具体是数字还是字符串，可以有上下文决定。标量变量的语法为 $variable_name。例

如：

```
$priority = 9;
```

把9赋予标量变量$priority，你也可以将字符串赋予该变量：

```
$priority = 'high';
```

注意 在Perl中，变量名的大小写是敏感的，所以$a 和$A是不同的变量。

以下的数值或字符串都可以赋给标量：

```
123  12.4  5E-10  0xff (hex)  0377 (octal)
'What you $see is (almost) what \n you get'   'Don\'t Walk'
"How are you?"  "Substitute values of $x and \n in \" quotes."
`date`  `uptime -u`  `du -sk $filespec | sort -n`
$x$list_of_things[5] $lookup{'key'}
```

从上面可以看出，Perl中有三种类型的引用。双引号（""）括起来的字符串中的任何标量和特殊意义的字符都将被Perl 解释。如果不想让Perl 解释字符串中的任何标量和特殊意义的字符，应该将字符串用单括号括起来。这时，Perl 不解释其中的任何字符，除了\\ 和\'。最后，可以用（`）将命令括起来，这样，其中的命令可以正常运行，并能得到命令的返回值。请看下面的例子：

```
1 #!/usr/bin/perl
2 $folks="100";
3 print "\$folks = $folks \n";
4 print '\$folks = $folks \n';
5 print "\n\n BEEP! \a \LSOME BLANK \ELINES HERE \n\n";
6 $date = `date +%D`;
7 print "Today is [$date] \n";
8 chop $date;
9 print "Date after chopping off carriage return: [".$date."]\n";
```

注意 实际程序中不应该包括行号。

其输出结果如下：

```
$folks = 100
$folks = $folks \n

BEEP! some blank LINES HERE

Today is [03/29/96]
Date after chopping off carriage return: [03/29/96]
```

第3行显示$folks的值。$之前必须使用换码符\，以便Perl显示字符串$folks而不是$folks的值100。

第4行使用的是单引号，结果Perl不解释其中的任何内容，只是原封不动地将字符串显示出来。

第6行使用的是（`），则date +%D命令的执行结果存储在标量$date中。

上例中使用了一些有特殊意义的字符，下面列出这些字符的含义：

\n	换行。
\r	回车。
\t	制表符。
\a	蜂鸣声。

\b	Backspace。
\L \E	将 \L 和\E之间的字符转换成小写。
\l	将其后的字符转换成小写 。
\U \E	将 \U 和\E 之间的字符转换成大写 。
\u	将其后的字符转换成大写 。
\cC	插入控制字符 C。
\x##	十六进制数 ## 。
\0ooo	八进制数ooo。
\\	反斜杠。
\	按原样输出下一个字符，例如： \$ 输出 $。

Perl中的数字是以浮点形式存储的。下面列出有关的数字运算符：

$a = 1 + 2;#	1加2，结果存储在$a中。
$a = 3 - 4;#	3减4，结果存储在 $a中。
$a = 5 * 6;#	5乘6，结果存储在 $a中。
$a = 7 / 8;#	7 除以8，结果存储在$a中。
$a = 9 ** 10;#	9的10次方，结果存储在$a中。
$a = 5 % 2;#	取5除2的余数，结果存储在$a中。
++$a;# $	a 加1，然后赋予$a。
$a++;#	先将$a返回，然后$a加1。
--$a;# $	a 减1，然后赋予$a。
$a--;#	先将$a返回，然后$a减1。

Perl 支持的逻辑运算符：

$r = $x || $y	$r = $x 或 $y。
$r = $x && $y	$r = $x 与 $y。
$r = ! $x	$r = 非 $x。

对于字符标量， Perl支持下面的运算符：

$a = $b . $c;#	将 $b 和$c连接，然后赋予$a。
$a = $b x $c;#	$b 重复$c 次，然后赋予$a。

下面是Perl 中的赋值方法：

$a = $b;#	将 $b 赋予 $a。
$a += $b;#	$b 加 $a，然后赋予$a。
$a -= $b;#	$a减 $b，然后赋予$a。
$a .= $b;#	把$b 连接到$a的后面，然后赋予$a。

你也可以使用下面的比较运算符：

$x == $y	如果$x 和$y相等，则返回真。
$x != $y	如果$x 和$y不相等，则返回真。
$x < $y	如果$x 比$y小，则返回真。
$x <= $y	如果$x小于等于$y，则返回真。
$x > $y	如果$x 比$y大，则返回真。
$x >= $y	如果$x 大于等于$y，则返回真。
$x eq $y	如果字符串$x 和字符串$y相同，则返回真。

$x ne $y	如果字符串$x 和字符串$y不相同，则返回真。
$x lt $y	如果字符串$x 比字符串$y小，则返回真。
$x le $y	如果字符串$x小于等于字符串$y，则返回真。
$x gt $y	如果字符串$x 比字符串$y大，则返回真。
$x ge $y	如果字符串$x 大于等于字符串$y，则返回真。
$x cmp $y	如果$x 大于 $y，则返回1，如果$x 等于 $y，则返回0，如果$x 小于$y，则返回-1。
$w ? $x : $y	如果$w为真，则返回 $x ；如果$w为假，则返回$y。
($x..$y)	返回从$x 到 $y之间的值。

7.4.2 数组

数组也叫做列表，是由一系列的标量组成的。数组变量以 @开头。请看以下的赋值语句：

```
@food  = ("apples",  "pears",  "eels");
@music = ("whistle",  "flute");
```

数组的下标从0开始，你可以使用方括号引用数组的下标：

```
$food[2]
```

返回eels。 注意@ 已经变成了 $ ， 因为eels 是一个标量。

在Perl中，数组有多种赋值方法，例如：

```
@moremusic = ("organ",  @music,  "harp");
@moremusic = ("organ",  "whistle",  "flute",  "harp");
```

还有一种方法可以将新的元素增加到数组中：

```
push(@food,  "eggs");
```

把eggs 增加到数组@food的末端。要往数组中增加多个元素，可以使用下面的语句：

```
push(@food,  "eggs",  "lard");
push(@food,  ("eggs",  "lard"));
push(@food,  @morefood);
```

push 返回数组的新长度。

pop用来将数组的最后一个元素删除，并返回该元素。例如：

```
@food  = ("apples",  "pears",  "eels");
$grub = pop(@food);#此时 $grub = "eels"
```

请看下面的例子：

```
1 #!/usr/bin/perl
2 #
3 # An example to show how arrays work in Perl
4 #
5 @amounts = (10,  24,  39);
6 @parts = ('computer',  'rat',  "kbd");
7
8 $a = 1; $b = 2; $c = '3';
9 @count = ($a,  $b,  $c);
10
11 @empty = ();
12
13 @spare = @parts;
14
15 print '@amounts = ';
```

```
16 print "@amounts \n";
17
18 print '@parts = ';
19 print "@parts \n";
20
21 print '@count = ';
22 print "@count \n";
23
24 print '@empty = ';
25 print "@empty \n";
26
27 print '@spare = ';
28 print "@spare \n";
29
30
31 #
32 # Accessing individual items in an array
33 #
34 print '$amounts[0] = ';
35 print "$amounts[0] \n";
36 print '$amounts[1] = ';
37 print "$amounts[1] \n";
38 print '$amounts[2] = ';
39 print "$amounts[2] \n";
40 print '$amounts[3] = ';
41 print "$amounts[3] \n";
42
43 print "Items in \@amounts = $#amounts \n";
44 $size = @amounts; print "Size of Amount = $size\n";
45 print "Item 0 in \@amounts = $amounts[$[]\n";
```

以下是显示结果：

```
@amounts = 10 24 39
@parts = computer rat kbd
@count = 1 2 3
@empty =
@spare = computer rat kbd
$amounts[0] = 10
$amounts[1] = 24
$amounts[2] = 39
$amounts[3] =
Items in @amounts = 2
Size of Amount = 3
Item 0 in @amounts = 10
```

第5行，三个整数值赋给了数组 @amounts。第6行，三个字符串赋给了数组@parts。第8行，字符串和数字分别赋给了三个变量，然后将三个变量赋给了数组 @count。11行创建了一个空数组。13行将数组 @spare赋给了数组@parts。

15到28行输出了显示的前5行。34到41行分别存取数组@amounts的每个元素。

注意 $amount[3]不存在，所以显示一个空项。

43行中使用$#array方式显示一个数组的最后一个下标，所以数组 @amounts的大小是($#amounts + 1)。44行中将一个数组赋给了一个标量，则将数组的大小赋给了标量。 45行使用了一个Perl中的特殊变量$[，用来表示一个数组的起始位置（缺省为0）。

7.4.3 相关数组

一般的数组允许我们通过数字下标存取其中的元素。例如数组 @food的第一个元素是$food[0]，第二个元素是$food[1]，以此类推。但Perl允许创建相关数组，这样我们可以通过字符串存取数组。其实，一个相关数组中每个下标索引对应两个条目，第一个条目叫做关键字，第二个条目叫做数值。这样，你就可以通过关键字来读取数值。

相关数组名以百分号(%)开头，通过花括号({})引用条目。例如：

```
%ages = ("Michael Caine", 39,
"Dirty Den", 34,
"Angie", 27,
"Willy", "21 in dog years",
"The Queen Mother", 108);
```

这样我们可以通过下面的方法读取数组的值：

```
$ages{"Michael Caine"};# Returns 39
$ages{"Dirty Den"};# Returns 34
$ages{"Angie"};# Returns 27
$ages{"Willy"};# Returns "21 in dog years"
$ages{"The Queen Mother"};# Returns 108
```

7.5 文件句柄和文件操作

我们可以通过下面的程序了解一下文件句柄的基本用法。此程序的执行结果和 UNIX系统的cat命令一样：

```
#!/usr/local/bin/perl
#
# Program to open the password file,  read it in,
# print it,  and close it again.
$file = '/etc/passwd';# Name the file
open(INFO,  $file);# Open the file
@lines = <INFO>;# Read it into an array
close(INFO);# Close the file
print @lines;# Print the array
```

open函数打开一个文件以供读取，其中第一个参数是文件句柄（ filehandle）。文件句柄用来供Perl在以后指向该文件。第二个参数指向该文件的文件名。 close函数关闭该文件。

open函数还可以用来打开文件以供写入和追加，只须分别在文件名之前加上 >和>>：

```
open(INFO,  $file);# Open for input
open(INFO,  ">$file");# Open for output
open(INFO,  ">>$file");# Open for appending
open(INFO,  "<$file");# Also open for input
```

另外，如果你希望输出内容到一个已经打开供写入的文件中，你可以使用带有额外参数的print语句。例如：

```
print INFO "This line goes to the file.\n";
```

最后，可以使用如下的语句打开标准输入（通常为键盘）和标准输出（通常为显示器）：

```
open(INFO, '-');# Open standard input
open(INFO, '>-');# Open standard output
```

一个Perl 程序在它一启动时就已经有了三个文件句柄：STDIN (标准输入设备)， STDOUT (标准输出设备) 和STDERR (标准错误信息输出设备)。

如果想要从一个已经打开的文件句柄中读取信息，可以使用 < > 运算符。

使用read 和write 函数可以读写一个二进制的文件。其用法如下：

```
read(HANDLE, $buffer, $length[, $offset]);
```

此命令可以把文件句柄是HANDLE的文件从文件开始位移$offset处，共$length字节，读到$buffer中。其中$offset是可选项，如果省略$offset，则read()从当前位置的前$length个字节读取到当前位置。可以使用下面的命令查看是否到了文件末尾：

```
eof(HANDLE);
```

如果返回一个非零值，则说明已经到达文件的末尾。

打开文件时可能出错，所以可以使用 die()显示出错信息。下面打开一个叫做“ test.data ”的文件：

```
open(TESTFILE, "test.data") || die "\n $0 Cannot open $! \n";
```

7.6 循环结构

7.6.1 foreach循环

在Perl 中，可以使用foreach循环来读取数组或其他类似列表结构中的每一行。请看下面的例子：

```
foreach $morsel (@food)# Visit each item in turn
# and call it $morsel
{
print "$morsel\n";# Print the item
print "Yum yum\n";# That was nice
}
```

每次要执行的命令用花括号括出。第一次执行时 $morsel被赋予数组@food的第一个元素的值，第二次执行时$morsel被赋予数组@food的第二个元素的值，以此类推直到数组的最后一个元素。

7.6.2 判断运算

在Perl中任何非零的数字和非空的字符串都被认为是真。零、全为零的字符串和空字符串都为假。

下面是一些判断运算符：

$a == $b　　如果$a 和$b相等，则返回真。

$a != $b　　如果$a和$b不相等，则返回真。

$a eq $b　　如果字符串$a和字符串$b相同，则返回真

$a ne $b　　如果字符串$a和字符串$b不相同，则返回真。

你可以使用逻辑运算符：

($a && $b)　$a与$b。

($a || $b)$a　或$b。

!($a)　　非$a。

7.6.3 for循环

Perl 中的 for 结构和C语言中的for 结构基本一样：

```
for (initialise; test; inc){
 first_action;
 second_action;
 etc
}
```

下面是一个for 循环的例子，用来显示从0到9的数字：

```
for ($i = 0; $i < 10; ++$i)# Start with $i = 1
        # Do it while $i < 10
# Increment $i before repeating
{
print "$i\n";
}
```

7.6.4 while 和until循环

下面是一个while 和until循环的例子。它从键盘读取输入直到得到正确的口令为止。

```
#!/usr/local/bin/perl
print "Password? ";# Ask for input
$a = <STDIN>;# Get input
chop $a;# Remove the newline at end
while ($a ne "fred")# While input is wrong...
{
print "sorry. Again? ";# Ask again
$a = <STDIN>;# Get input again
chop $a;# Chop off newline again
}
```

当输入和口令不相等时，执行while 循环。

你也可以在执行体的末尾处使用while 和until ， 这时需要用do语句：

```
#!/usr/local/bin/perl
do
{
"Password? ";# Ask for input
$a = <STDIN>;# Get input
chop $a;# Chop off newline
}
while ($a ne "fred")# Redo while wrong input
```

7.7 条件结构

Perl 也允许 if/then/else 表达式。请看下面的例子：

```
if ($a) {
print "The string is not empty\n";
}
else {
print "The string is empty\n";
}
```

注意 在Perl中，空字符被认为是假。

If结构中也可以使用嵌套结构：

```
if (!$a)# The ! is the not operator
{
print "The string is empty\n";
}
elsif (length($a) == 1)# If above fails，try this
{
print "The string has one character\n";
}
elsif (length($a) == 2)# If that fails，try this
{
print "The string has two characters\n";
}
else# Now，everything has failed
{
print "The string has lots of characters\n";
}
```

7.8 字符匹配

Perl 字符匹配功能十分强大。字符匹配功能的核心是规则表达式（ RE），也就是字符匹配过程中涉及到的格式。=~运算符用来进行格式匹配和替换。例如：

如果：

```
$s = 'One if by land and two if by sea';
```

则：

```
if ($s =~ /if by la/) {print "YES"}
else {print"NO"}
```

将会显示 YES，因为if by la在字符串$s中。再例如：

```
if ($s =~ /one/) {print "YES"}
else {print "NO"}
```

将显示 NO，因为RE是对大小写敏感的。如果使用 i选项，则将忽略大小写，则下面会显示出YES：

```
if ($s =~ /one/i) {print "YES"}
else {print "NO"}
```

下面列出了RE中许多具有特殊意义的字符：

. 任何字符除了换行符（\n）

^ 一行和一个字符串的开始

$ 一行和一个字符串的结束

* 其前一个字符重复零次或多次

+ 其前一个字符重复一次或多次

? 其前一个字符重复零次或一次

例如：

```
if ($x =~ /l.mp/) {print "YES"}
```

对于$x = “ lamp ”、“ lump ”、“ slumped ”将显示 YES，但对于$x = “ lmp ”或“ less amperes ”将不会显示YES。

再看下面的例子：

/fr.*nd/ 匹配 frnd、friend、front and back。
/fr.+nd/ 匹配 frond、friend、front and back。但不匹配frnd。
/10*1/ 匹配 11、101、1001、100000001。
/b(an)*a/ 匹配 ba、 bana、banana、bananananana。
/flo?at/ 匹配 flat和float，但不匹配flooat。

方括号用来匹配其中的任何字符。方括号中的 -符号用来表明在什么之间， ^符号表明非的意思。

[0123456789] 匹配任何单个的数字。
[0-9] 匹配任何单个的数字。
[a-z]+ 匹配任何由小写字母组成的单词。
[^0-9] 匹配任何非数字的字符。

反斜杠还是用于转义。如果你想匹配+、?、.、*、^、$、(、)、[、]、{、}、|、\ 和/这些字符，则其前面必须用反斜杠（\）。例如：

/10.2/ 匹配 10Q2、1052和10.2。
/10\.2/ 匹配 10.2，但不和10Q2或 1052匹配。
/*+/ 匹配一个或多个星号。
/A:\\DIR/ 匹配 A:\DIR。
/\/usr\/bin/ 匹配 /usr/bin。

下面还有一些特殊意义的字符：

\n 换行。
\t 制表符。
\w 任何字母和数字和[a-zA-Z0-9_]一样。
\W 任何非字母和数字和[^a-zA-Z0-9_]一样。
\d 任何数字和[0-9]一样。
\D 任何非数字和[^0-9]一样。
\s 任何空白字符：空格、 tab、换行等。
\S 任何非空白字符。
\b 单词边界，只对 [] 以外有效。
\B 非单词边界。

7.9 替换和翻译

7.9.1 替换

Perl可以使用s函数利用字符匹配的结果进行字符替换。 s 函数和vi编辑器的作用基本一样。这时还是使用字符匹配运算符=~，例如：

将字符串$sentence中所出现的london 用London 替换，可以使用如下的命令：

```
$sentence =~ s/london/London/
```

命令的返回值是所做的替换数目。

但此命令只能替换第一个出现的 london。如果希望将所有在字符串中出现的 london都用London替换，则应使用/g选项：

```
s/london/London/g
```

此命令的对象是$_变量，也就是当前的缺省变量。

如果希望还能替换类似lOndon、lonDON、LoNDoN的字符串，可以使用：

```
s/[Ll][Oo][Nn][Dd][Oo][Nn]/London/g
```

但更为简单的方法是使用i选项，也就是忽略大小写：

```
s/london/London/gi
```

7.9.2 翻译

tr函数允许逐字地翻译。下面的命令使得字符串$sentence中的a、b、c分别由e、f、d代替：

```
$sentence =~ tr/abc/efd/
```

结果返回所做的替换数目。

大多数RE中的特殊字符在tr函数中并不存在。例如下面的命令用来计算字符串 $sentence 中星号（*）的数目，并将结果存储在$count:

```
$count = ($sentence =~ tr/*/*/);
```

7.10 子过程

7.10.1 子过程的定义

Perl语言也可以定义自己的子过程。子过程的定义如下：

```
sub mysubroutine{
print "Not a very interesting routine\n";
print "This does the same thing every time\n";
}
```

下面的几种方法都可以调用这个子过程：

```
&mysubroutine;# Call the subroutine
&mysubroutine($_);# Call it with a parameter
&mysubroutine(1+2, $_);# Call it with two parameters
```

7.10.2 参数

调用一个子过程时，所有的参数都传送到了数组 @_中。下面子过程的例子显示出所有的参数：

```
sub printargs{
print "@_\n";
}
&printargs("perly", "king");# Example prints "perly king"
&printargs("frog", "and", "toad"); # Prints "frog and toad"
```

7.10.3 返回值

下面的例子返回两个输入参数的最大值：

```
sub maximum{
if ($_[0] > $_[1]){
$_[0];
}
else{
$_[1];
```

```
}
}
$biggest = &maximum(37, 24);# Now $biggest is 37
```

7.11 Per程序的完整例子

最后请看一个Perl语言的完整的例子。

此程序从一个记录学生信息的文件 stufile 和一个记录学生成绩的文件 scorefile 中生成一个学生成绩报告单。

输入文件stufile 由学生ID、姓名和年级三个字段组成，其间由分号隔开：

```
123456；Washington, George；SR
246802；Lincoln, Abraham "Abe"；SO
357913；Jefferson, Thomas；JR
212121；Roosevelt, Theodore "Teddy"；SO
```

文件scorefile 由学生ID、科目号、分数三个字段组成，由空格隔开：

```
123456 1 98
212121 1 86
246802 1 89
357913 1 90
123456 2 96
212121 2 88
357913 2 92
123456 3 97
212121 3 96
246802 3 95
357913 3 94
```

程序应该输出如下的结果：

```
Stu-ID Name...1 2 3 Totals:
357913 Jefferson, Thomas90 92 94 276
246802 Lincoln, Abraham "Abe"89 95 184
212121 Roosevelt, Theodore "Teddy"86 88 96 270
123456 Washington, George98 96 97 291
 Totals: 363 276 382
```

源程序如下：

```
#!/usr/local/bin/perl
# Gradebook - demonstrates I/O, associative
# arrays, sorting, and report formatting.
# This accommodates any number of exams and students
# and missing data. Input files are:
$stufile='stufile';
$scorefile='scorefile';
# If file opens successfully, this evaluates as "true", and Perl
# does not evaluate rest of the "or" "||"
open (NAMES, "<$stufile") || die "Can't open $stufile $!";
open (SCORES, "<$scorefile") || die "Can't open $scorefile $!";
# Build an associative array of student info
# keyed by student number
while (<NAMES>) {
($stuid, $name, $year) = split(':', $_);
```

```
$name{$stuid}=$name;
if (length($name)>$maxnamelength) {
$maxnamelength=length($name);
}
}
close NAMES;
# Build a table from the test scores:
while (<SCORES>) {
($stuid,  $examno,  $score) = split;
$score{$stuid,  $examno} = $score;
if ($examno > $maxexamno) {
$maxexamno = $examno;
}
}
close SCORES;
# Print the report from accumulated data!
printf "%6s %-${maxnamelength}s ",
'Stu-ID',  'Name...';
foreach $examno (1..$maxexamno) {
printf "%4d",  $examno;
}
printf "%10s\n\n",  'Totals:';
# Subroutine "byname" is used to sort the %name array.
# The "sort" function gives variables $a and $b to
# subroutines it calls.
# "x cmp y" function returns -1 if x<y,   0 if x=y,
# +1 if x>y.  See the Perl documentation for details.
sub byname { $name{$a} cmp $name{$b} }
# Order student IDs so the names appear alphabetically:
foreach $stuid ( sort byname keys(%name) ) {
# Print scores for a student,   and a total:
printf "%6d %-${maxnamelength}s ",
$stuid,  $name{$stuid};
$total = 0;
foreach $examno (1..$maxexamno) {
printf "%4s",  $score{$stuid,  $examno};
$total += $score{$stuid,  $examno};
$examtot{$examno} += $score{$stuid,  $examno};
}
printf "%10d\n",  $total;
}
printf "\n%6s %${maxnamelength}s ",  "Totals: ";
foreach $examno (1..$maxexamno) {
printf "%4d",  $examtot{$examno};
}
print "\n";
exit(0);
```

第三篇　Linux系统内核分析

第8章　Linux内核简介

本章介绍有关Linux的内核内容，如系统初始化、运行以及内核提供的各种调用等。

8.1　系统初始化

当PC启动时，Intel系列的CPU首先进入的是实模式，并开始执行位于地址0xFFFF0处的代码，也就是ROM-BIOS起始位置的代码。BIOS先进行一系列的系统自检，然后初始化位于地址0的中断向量表。最后BIOS将启动盘的第一个扇区装入到0x7C00，并开始执行此处的代码。这就是对内核初始化过程的一个最简单的描述。

最初，Linux核心的最开始部分是用8086汇编语言编写的。当开始运行时，核心将自己装入到绝对地址0x90000，再将其后的2k字节装入到地址0x90200处，最后将核心的其余部分装入到0x10000。当系统装入时，会显示Loading...信息。装入完成后，控制转向另一个实模式下的汇编语言代码boot/Setup.S。

Setup部分首先设置一些系统的硬件设备，然后将核心从0x10000处移至0x1000处。这时系统转入保护模式，开始执行位于0x1000处的代码。

接下来是内核的解压缩。0x1000处的代码来自于文件zBoot/head.S，它用来初始化寄存器和调用decompress_kernel()程序。decompress_kernel()程序由zBoot/inflate.c, zBoot/unzip.c 和zBoot/misc.c组成。解压缩后的数据被装入到了0x100000处，这也是Linux不能在内存小于2M的环境下运行的主要原因。

解压后的代码在0x1010000处开始执行，紧接着所有的32位的设置都将完成：IDT、GDT和 LDT将被装入，处理器初始化完毕，设置好内存页面，最终调用start_kernel过程。这大概是整个内核中最为复杂的部分。

start_kernel()程序用于初始化系统内核的各个部分，包括：

- 设置内存边界，调用paging_init()初始化内存页面。
- 初始化陷阱，中断通道和调度。
- 对命令行进行语法分析。
- 初始化设备驱动程序和磁盘缓冲区。
- 校对延迟循环。

最后，系统核心转向move_to_user_mode()，以便创建初始化进程（init）。此后，进程0开始进入无限循环。

8.2　系统运行

初始化进程开始执行/etc/init、/bin/init 或 /sbin/init中的一个之后，系统内核就不再对程序进行直接控制了。之后系统内核的作用主要是给进程提供系统调用，以及提供异步中断事件的

处理。多任务机制已经建立起来，并开始处理多个用户的登录和 fork()创建的进程。

8.3 内核提供的各种系统调用

8.3.1 进程的基本概念和系统的基本数据结构

从系统内核的角度看来，一个进程仅仅是进程控制表（process table）中的一项。

进程控制表中的每一项都是一个 task_struct结构，而 task_struct结构本身是在include/linux/sched.h中定义的。在task_struct结构中存储各种低级和高级的信息，包括从一些硬件设备的寄存器拷贝到进程的工作目录的链接点。

由于内核更新，这里说的头文件都不准确了

进程控制表既是一个数组，又是一个双向链表，同时又是一个树。其物理实现是一个包括多个指针的静态数组。此数组的长度保存在 include/linux/tasks.h 定义的常量NR_TASKS中，其缺省值为128，数组中的结构则保存在系统预留的内存页中。链表是由 next_task 和 prev_task两个指针实现的，而树的实现则比较复杂。

系统启动后，内核通常作为某一个进程的代表。一个指向 task_struct的全局指针变量current用来记录正在运行的进程。变量 current只能由kernel/sched.c中的进程调度改变。当系统需要查看所有的进程时，则调用 for_each_task，这将比系统搜索数组的速度要快得多。

某一个进程只能运行在用户方式（user mode）或内核方式（kernel mode）下。用户程序运行在用户方式下，而系统调用运行在内核方式下。在这两种方式下所用的堆栈不一样：用户方式下用的是一般的堆栈，而内核方式下用的是固定大小的堆栈（一般为一个内存页的大小）。

8.3.2 创建和撤消进程

Linux系统使用系统调用 fork()来创建一个进程，使用 exit()来结束进程。 fork()和exit()的源程序保存在kernel/fork.c and kernel/exit.c中。fork()的主要任务是初始化要创建进程的数据结构，其主要的步骤有：

1）申请一个空闲的页面来保存task_struct。
2）查找一个空的进程槽（find_empty_process()）。
3）为kernel_stack_page申请另一个空闲的内存页作为堆栈。
4）将父进程的LDT表拷贝给子进程。
5）复制父进程的内存映射信息。
6）管理文件描述符和链接点。

撤消一个进程可能稍微复杂些，因为撤消子进程必须通知父进程。另外，使用 kill()也可以结束一个进程。 sys_kill()、sys_wait()和sys_exit()都保存在文件exit.c中。

8.3.3 执行程序

使用fork()创建一个进程后，程序的两个拷贝都在运行。通常一个拷贝使用 exec()调用另一个拷贝。系统调用exec()负责定位可执行文件的二进制代码，并负责装入和运行。

Linux系统中的exec()通过使用linux_binfmt结构支持多种二进制格式。每种二进制格式都代表可执行代码和链接库。 linux_binfmt结构种包含两个指针，一个指向装入可执行代码的函数，另一个指向装入链接库的函数。

Unix系统提供给程序员 6种调用 exec()的方法。其中的 5种是作为库函数实现，而

sys_execve()是由系统内核实现的。它执行一个十分简单的任务：装入可执行文件的文件头，并试图执行它。如果文件的头两个字节是#!,那么它就调用在文件第一行中所指定的解释器，否则，它将逐个尝试注册的二进制格式。

8.4 存取文件系统

众所周知，文件系统是Unix系统最基本的资源。最初的Unix系统一般都只支持一种单一类型的文件系统，在这种情况下，文件系统的结构深入到整个系统内核中。而现在的系统大多都在系统内核和文件系统之间提供一个标准的接口，这样不同文件结构之间的数据可以十分方便地交换。Linux也在系统内核和文件系统之间提供了一种叫做VFS（virtual file system）的标准接口。

这样，文件系统的代码就分成了两部分：上层用于处理系统内核的各种表格和数据结构；而下层用来实现文件系统本身的函数，并通过VFS来调用。这些函数主要包括：

- 管理缓冲区(buffer.c)。
- 响应系统调用fcntl() 和 ioctl()(fcntl.c and ioctl.c)。
- 将管道和文件输入/输出映射到索引节点和缓冲区(fifo.c, pipe.c)。
- 锁定和不锁定文件和记录(locks.c)。
- 映射名字到索引节点(namei.c, open.c)。
- 实现select()函数(select.c)。
- 提供各种信息(stat.c)。
- 挂接和卸载文件系统(super.c)。
- 调用可执行代码和转存核心(exec.c)。
- 装入各种二进制格式(bin_fmt*.c)。

VFS接口则由一系列相对高级的操作组成，这些操作由和文件系统无关的代码调用，并且由不同的文件系统执行。其中最主要的结构有inode_operations 和 file_operations。

file_system_type是系统内核中指向真正文件系统的结构。每挂接一次文件系统，都将使用file_system_type组成的数组。file_system_type组成的数组嵌入到了fs/filesystems.c中。相关文件系统的read_super函数负责填充super_block结构。

第9章 系 统 进 程

本章介绍进程的基本概念、进程结构、进程调度以及进程使用的文件等方面的内容。

9.1 什么是进程

进程是运行于自己的虚拟地址空间的一个程序。可以说，任何在 Linix 系统下运行的都是进程。

Linux系统中包括下面几种类型的进程：

- 交互进程：该进程是由shell控制和运行的。它即可以在前台运行，也可以在后台运行。
- 批处理进程：该进程不属于某个终端，它被提交到一个队列中以便顺序执行。
- 守护进程：该进程只有在需要时才被唤起在后台运行。它一般在 Linux启动时开始执行。

进程是动态的，在处理器执行机器代码时进程一直在变化。进程不但包括程序的指令和数据，而且包括程序计数器和CPU的所有寄存器以及存储临时数据的进程堆栈。所以，正在执行的进程包括处理器当前的一切活动。 Linux是一个多进程的操作系统。每个进程都有自己的权限和任务。某一进程的失败一般不会导致其他进程的失败。进程之间可以通过由内核控制的机制相互通讯。

在进程的整个运行期间，它将会用到各种系统资源，会用到 CPU运行它的指令，需要物理内存保存它的数据。它可能打开和使用各种文件，直接或间接地使用系统中的各种物理设备。Linux系统内核必须了解进程本身的情况和进程所用到的各种资源，以便在多个进程之间合理地分配系统资源。

系统中最为宝贵的资源是 CPU，因为一般情况下一个系统只有一个 CPU。Linux是一个多进程的操作系统，所以，其他的进程必须等到正在运行的进程空闲 CPU后才能运行。当正在运行的进程等待其他的系统资源时， Linux内核将取得CPU的控制权，并将CPU分配给其他正在等待的进程。内核中的调度算法决定将CPU分配给那一个进程。

9.2 进程的结构

Linux系统中的每一个进程都包括一个叫做 task_struct的数据结构，而所有指向这些数据结构的指针组成系统中的一个进程向量数组。

缺省情况下，系统的进程向量数组大小是512，这表示系统中同时最多容纳的进程为512个。每当一个新的进程创建时，一个新的 task_struct结构将分配给该进程，并同时增加到进程向量数组中。系统还有一个当前进程指针，用来指向正在运行的进程。

进程的task_struct结构分为以下几个字段：

1) 状态

进程在运行时总是在不停地改变它的状态。在Linux系统中，有以下几个状态：

- 运行态：此时进程或者正在运行，或者准备运行。
- 等待态：此时进程在等待一个事件的发生或某种系统资源。Linux系统分为两种等待进程：可中断的和不可中断的。可中断的等待进程可以被某一信号中断，而不可中断的等待进

程将一直等待硬件状态的改变。

- 停止态：此时进程已经被中止。
- 死亡态：这是一个停止的进程，但还在进程向量数组中占有一个task_struct结构。

2) 调度信息

调度算法需要此信息来决定系统中的那一个进程需要执行。

3) 标识符

系统中的每一个进程都有一个进程标识符。进程标识符并不是指向进程向量的索引。每个进程同时还包括用户标识符和工作组标识符。

4）内部进程通讯

Linux系统支持信号、管道、信号量等内部进程通讯机制。

5）链接

在Linux系统中，每个进程都和其他的进程有所联系。除了初始化进程，其他的进程都有父进程。一个新的进程一般都是由其他的进程复制而来的。task_struct结构中包括指向父进程，兄弟进程和子进程的指针。

6）时间和计时器

内核需要记录进程的创建时间和进程运行所占用的 CPU的时间。Linux系统支持进程特殊间隔计时器。进程可以使用系统调用设置计时器，并当计时器失效时给进程一个信号。计时器可以是一次性的或周期性的。

7）文件系统

进程在运行时可以打开和关闭文件。task_struct结构中包括指向每个打开文件的文件描述符的指针，并且包括两个指向VFS索引节点的指针。VFS的索引节点用来在文件系统内唯一地描述一个文件或目录，并且提供文件系统操作的统一的接口。第一个索引节点是进程的根目录，第二个节点是当前的工作目录。两个VFS索引节点都有一个计数字段用来表明指向节点的进程数。

8）虚拟内存

大多数的进程都需要虚拟内存。 Linux系统必须了解如何将虚拟内存映射到系统的物理内存。

9）处理器的内容

一个进程可以说是系统当前状态的总和。每当一个进程正在运行时，它都要使用处理器的寄存器及堆栈等资源。当一个进程挂起时，所有有关处理器的内容都要保存到进程的task_struct中。当进程恢复运行时，所有保存的内容再装入到处理器中。

在一个进程的task_struct中，有4对进程和组标识符：

- uid, gid：正在运行的进程用户标识符和组标识符。
- 有效uid和 gid：有些程序可能将正在运行的进程的uid和gid改为自己所有。这些程序一般被称为setuid程序，它们十分有用，因为这是一种限制服务存取权限的方法。有效 uid和gid是那些setuid程序的uid 和gid，并且它们保持不变。每当内核检查权限时，都要检查有效uid和gid。
- 文件系统uid 和gid：文件系统uid和gid一般和有效uid 和gid相同，它们用于检查文件系统的存取权限。
- 保留 uid 和gid：它们用来和POSIX标准兼容。在程序通过系统调用改变进程的 uid和gid时，它们可以保存真正的uid 和gid 。

9.3 进程调度

所有进程都是既运行于用户方式下，又运行于系统方式下。这就需要有一个安全机制便于在两种方式下切换。用户方式的权限要比系统方式的权限小得多。进程每次调用一个系统调用时，都要从用户方式切换到系统方式，并继续执行。在 Linux系统中，进程没有绝对的优先权，也就是说一个进程不能停止另一个正在运行的进程以便运行它自己。每个进程根据自己是否需要等待某些系统资源以决定是否放弃所占用的 CPU。例如，一个进程需要等待从一个文件中读取字符，该等待在系统方式下的系统调用中发生，此时，该进程将被挂起，系统选择另一个进程继续运行。

虽然进程需要经常调用系统调用，但如果一个进程要直到系统调用发生时才放弃 CPU,那么它也会占用过多的CPU时间。因此， Linux系统使用一种时间片的调度算法。每个进程只能运行200ms，然后放弃CPU给其他可以运行的进程，该进程将在以后恢复运行。

可以运行的进程是指只需要 CPU后就可以运行的进程。 Linux系统使用一个基于调度算法的简单权限来决定运行哪个进程。每个进程的task_struct结构中包括下面这些调度信息：

1）策略。

这是进程将会使用的调度策略。 Linux系统共有两种类型的进程：一般进程和实时进程。实时进程的权限要比其他进程的权限高。如果有一个实时进程等待运行，那么一般情况下都将首先运行。实时进程也有两种策略：轮流策略和先进先出策略。在轮流策略中，每个进程轮流运行，而在先进先出策略中，进程按照运行队列中的顺序执行，并且运行队列的顺序永远不变。

2）优先权。

这是调度算法给予进程的优先权，也即当进程被允许运行时能够运行的时间的长短。你可以通过系统调用改变此优先权。

3）实时优先权。

这是给予实时进程之间的一个相对的优先权。你也可以通过系统调用改变此实时进程的优先权。

4）计数器。

这是进程允许运行的时间。当进程第一次运行时，它将被设置为进程的优先权，并且在每个时钟周期后减一。

调度算法可以在多种情况下发生。它可以在将当前进程放入等待队列时发生，也可以在一个系统调用后发生。当调度算法发生时，它将执行以下几个步骤：

1）处理内核中的工作。

2）处理当前进程。

在其他进程运行之前，当前进程必须处理好。

- 如果当前进程使用的是轮流策略，则当前进程被放到运行队列的最后。
- 如果当前进程是可以被中断的，并且在最后一次调度之后接收到过中断信号，则进程的状态被设置为可运行。
- 如果当前进程的运行时间用完，则进程的状态设置为可运行。
- 如果当前进程为可运行，则进程状态保持为可运行。
- 那些既不是可运行也不是可中断的进程将被从运行队列中移走。

3）选择进程。

调度算法查找运行队列以找出最需要运行的进程。如果队列中有实时进程，那么实时进程将优先运行。一个普通进程的优先权是其计数器的值，而实时进程的优先权是其计数器的值加上1000。当前进程由于已经消耗了一些时间片，所以和其他的具有相同优先权的进程相比将处于不利的位置。如果有几个进程具有相同的优先权，则最靠近队列前端的进程将被执行。

4）进程交换。

如果最需要执行的进程不是当前进程，那么当前进程就会被挂起，同时一个新的进程将被执行。在结束当前进程时，进程所涉及的一切机器状态，包括程序计数器以及 CPU寄存器将保存到进程的task_struct中，而即将运行的进程的task_struct中的状态将装入到机器中。如果当前进程和即将运行的进程使用了虚拟内存的话，系统的内存页面表页会被更新。

9.4 进程使用的文件

系统中的每一个进程都包括两个描述文件系统特定信息的数据结构，如图 9-1所示。一个是fs_struct，它包括指向进程的VFS索引节点和指向进程的umask的指针。umask是创建新文件时的缺省模式，可以通过系统调用来改变。另一个数据结构是 files_struct，它包括进程正在使用的所有文件的信息。程序从标准输入设备中读入信息，并将输出信息写入标准输出设备中。这些设备在程序看来都是文件，每一个文件都包括自己的文件描述符。 files_struct结构中包括多达256个的文件数据结构，每个文件数据结构都描述一个进程正在使用的文件。文件数据结构中的f_mode用来描述创建文件的方式，例如只读，可读写或只写。 f_pos保存文件中下一个读写操作将要发生的位置。 f_inode指向VFS的索引节点。 f_op是一个指向包括一系列文件操作程序地址的向量的指针。

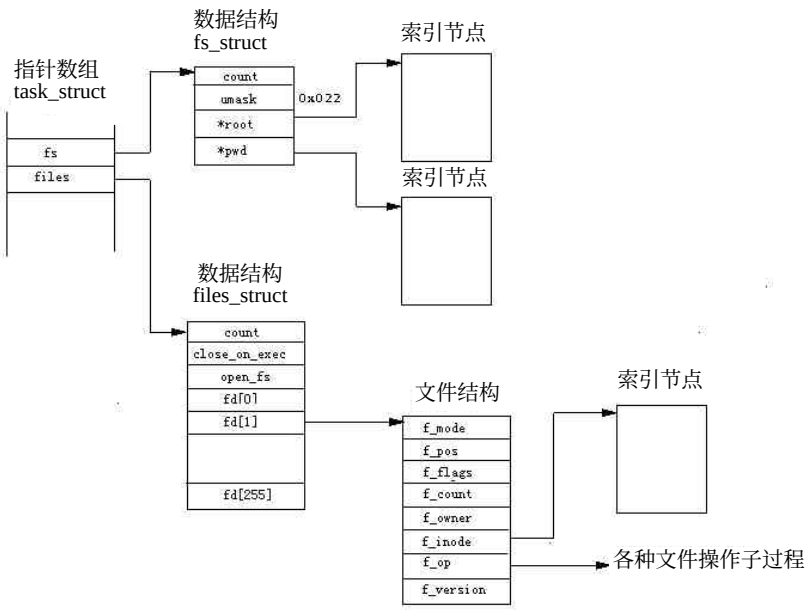

图9-1 文件数据结构示意图

每当进程打开一个文件时， files_struct的一个空的文件指针就用来指向新文件结构。在Linux系统中，一个进程在打开时就已经存在三个文件描述符，这三个文件描述符分别是标准输入、标准输出和标准错误文件，它们在进程的文件数据结构向量中分别是 0、1、2。

9.5 进程使用的虚拟内存

进程的虚拟内存中包括可执行代码和数据。首先，程序的镜像被装入，它包括可执行代码和数据。然后，进程分配处理过程中需要使用的虚拟内存。这些新分配的虚拟内存应该被链接到进程已经存在的虚拟内存。最后， Linux系统使用共享库，这些共享库中的代码和数据也应该被链接到进程的虚拟内存地址空间中。

因为在一个固定的时间中，进程只用到一部分代码和数据，所以把进程的全部代码和数据都装入到物理内存将是十分浪费的。 Linux使用一种需求装入的技术，也就是说，只有当进程试图存取一个虚拟内存页时，此虚拟内存页才被装入到物理内存中。所以， Linux内核修改进程的内存页面表，标记出不在物理内存中的虚拟内存页。当进程试图存取虚拟内存页时，系统硬件将产生一个页面错误，并将系统的控制权交给 Linux内核以处理此错误。因此，对于进程的地址空间中的每一个虚拟内存页， Linux内核都将知道它的来源以及如何将它调入到物理内存，以便修正页面错误。

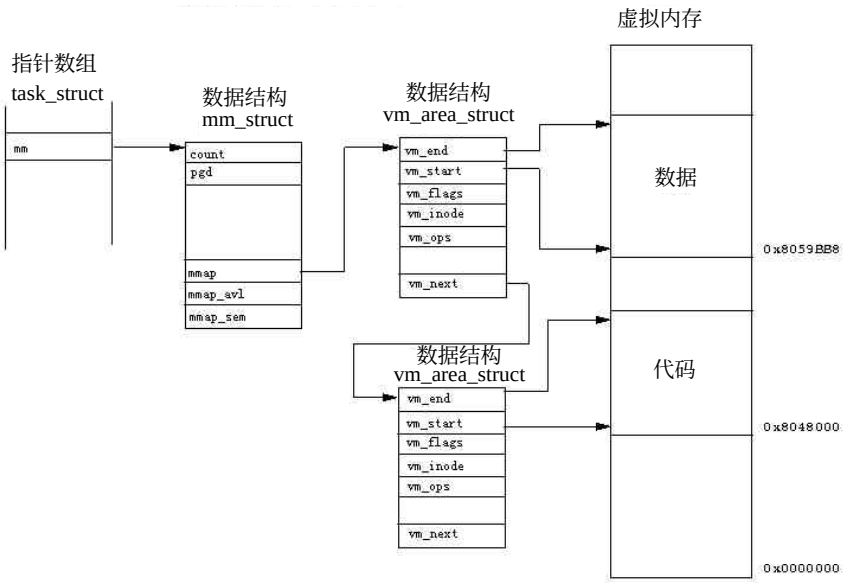

图9-2 进程使用的虚拟内存示意图

每一个进程的虚拟内存的内容都由进程的 task_struct中的mm指针指向的mm_struct结构来描述，如图9-2所示。进程的mm_struct结构包括已装入的可执行镜像的信息和一个指向进程页面表的指针。进程页面表由一系列的 vm_area_struct结构组成，每一个vm_area_struct结构代表进程内一片虚拟内存区域。 vm_area_struct结构中还包括vm_ops指针，用来指向一系列的虚拟内存处理程序。

当Linux内核为进程创建新的虚拟内存区或当 Linux内核处理虚拟内存页面错误时，内核都将经常存取进程的vm_area_struct结构。所以快速查找正确的 vm_area_struct就成为系统性能的关键。为了加快系统的查找时间， Linux同时将vm_area_struct组成了AVL (Adelson-Velskii and Landis)树。这种树的结构使得每一个 vm_area_struct节点都有各一个左右指针指向节点的左右节点。这样，当内核查找正确的 vm_area_struct时，可以从根节点开始从左右两个方向同时查找，大大提高了查找的效率。当然，插入一个vm_area_struct时，将会需要额外的处理时间。

9.6 创建进程

当系统刚刚启动时，系统运行于内核方式，也就是只有一个初始化进程在运行和其他进程一样，初始化进程也是由一系列机器状态组成的。当其他进程创建并运行时，初始化进程的状态被保存到初始化进程的task_struct结构中。

初始化进程的进程标识符是 1，它是系统的第一个真正的进程。它首先做一些系统的初始化设置（例如打开系统控制台和挂接根目录文件系统），然后执行系统初始化程序。初始化程序是/etc/init、/bin/init 或 /sbin/init中的一个。初始化程序使用/etc/inittab作为脚本文件来创建新的进程。这些新的进程同样可以创建其他新的进程。系统中所有的进程都是初始化进程的子进程。

新进程是通过复制老进程或当前进程而创建的。新进程的创建使用系统调用 fork()或者clone(),并且是在内核内部的内核方式下完成的。系统调用结束时，如果调度算法选择的话，新进程就可以准备运行。系统从物理内存中分配给新进程一个 task_struct数据结构和进程堆栈。新的task_struct结构加入到进程向量中。进程还得到一个和系统中的其他进程不同的唯一的标识符。

在复制进程时，Linux允许两个进程共享系统资源，包括进程要用到的文件、信号处理程序以及虚拟内存等。当资源共享时，共享资源的计数字器将会增加。这样，除非所有使用该资源的进程都停止运行，否则该资源不会被删除。

复制一个进程的虚拟内存是十分麻烦的。首先需要建立一系列的 vm_area_struct数据结构和它们的所有者 mm_struct，还需复制进程的页面表。此时并没有真正复制进程的虚拟内存，因为此时复制虚拟内存是十分麻烦的：一部分虚拟内存在物理内存中，一部分虚拟内存在进程可执行镜像中，还可能一部分内存在交换文件中。因此， Linux使用一种叫做“写入时复制”的技术，也就是只有当父进程或子进程试图写入虚拟内存时，子进程的虚拟内存才将会被复制。任何虚拟内存只要不执行写入，即使可以写入，也是可以被进程共享的。例如，可执行代码通常都是共享的。使用“写入时复制”技术时，可以写入的虚拟内存区的页面表入口标记为 read only，而虚拟内存的vm_area_struct数据结构标记为copy on write。当一个进程试图写入这样的虚拟内存页时，将会发生一个页面错误。这时， Linux将会复制虚拟内存，并修改两个进程的页面表和虚拟内存数据结构。

9.7 进程的时间和计时器

内核中保存有进程的创建时间和进程运行时消耗的 CPU时间。除了计时器， Linux还支持一些间隔时钟，用来在一定的时间间隔后产生信号中断。系统中的间隔时钟共有三种：

9.7.1 实时时钟

该时钟按实时方式运行，失效时产生一个SIGALRM信号。

9.7.2 虚拟时钟

该时钟只在进程运行时才运行，失效时产生一个SIGVTALRM信号。

9.7.3 形象时钟

该时钟在进程运行和在系统代表进程运行时都运行，失效时产生一个 SIGPROF 信号。

9.8 程序的执行

在Linux中，程序和命令是由命令解释器 shell解释执行的。当键入一个命令时， shell搜索保存在PATH环境变量中的进程及路径里的目录，找出和命令名相同的可执行镜像，如果发现则装入并执行。系统的 shell首先使用 fork()系统调用复制自己，然后子进程用找到的可执行的二进制文件的内容替换正在执行的 shell二进制文件。一般情况下， shell要等待命令运行结束或子进程的退出。你也可以按 CTRL+Z键以产生一个 SIGSTOP信号来停止子进程，以重新恢复 shell的运行。之后，可以使用bg命令将进程推入后台运行。

可执行文件可以有各种形式，甚至可能只是一个脚本文件。可执行的目标文件包括可执行代码和数据，以便操作系统装入和执行。 Linux系统最为常用的目标文件格式是 EFL，但它还可以处理大多数格式的可执行目标文件。

9.8.1 ELF文件

ELF（Executable and Linkable Format）文件格式是各种Unix系统中最为常用的格式。虽然EFL文件和其他格式的文件 (例如ECOFF 和a.out)相比系统开销稍大，但EFL文件更为灵活。EFL可执行文件包括可执行代码和数据。可执行镜像中的表格描述了程序应该装入到进程虚拟内存的位置。可执行镜像还指定了镜像在内存中的分布和镜像中第一条执行代码的地址。

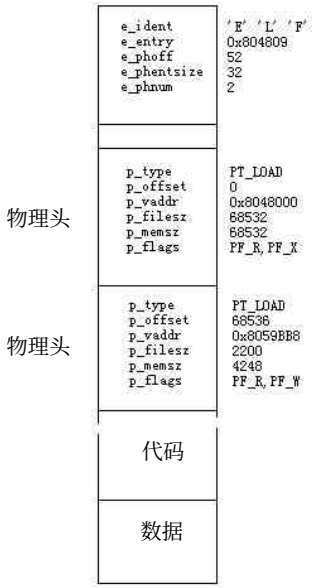

图9-3 ELF可执行文件镜像示意图

图9-3是一个静态链接的EFL可执行镜像的布局。

这是一个显示hello world的简单C语言程序。文件头表示这是一个 EFL镜像，带有两个物理文件头 (e_phnum的值为 2)，物理文件头从镜像开头的第 52个字节开始(e_phoff的值为52)。第一个物理文件头描述了镜像中的可执行代码。可执行代码从虚拟内存的0x8048000地址开始，一共有 65532字节。这是因为此镜像是一个静态链接镜像，它包括库函数 printf()的所有代码。镜像的入口点，也就是程序的第一条指令，是在虚拟内存地址 0x8048090 (e_entry的值为0x8048090)。可执行代码紧跟在第二个物理文件头的后面。第二个物理文件头用来描述程序的数据。程序的数据部分被装入到虚拟内存的地址 0x8059BB8。这些数据是可读写的（p_flags的值为PF_R、PF_w)。

9.8.2 脚本文件

脚本文件需要一个解释器以便运行。 Linux系统中包括很多的解释器，例如 perl 和不同的 shell程序。Linux脚本文件的第一行是运行此脚本文件的解释器的名称，

Linux系统在运行脚本文件时，先试图打开文件第一行中的可执行文件。如果可执行文件可以打开，则它获得一个指向VFS索引节点的指针，并解释执行脚本文件。

第10章 内 存 管 理

本章介绍有关内存管理方面的内容，如虚拟内存的抽象模型和共享、按需调入页面、页面交换等。

10.1 内存管理的作用

内存管理系统是操作系统中最为重要的部分，因为系统的物理内存总是少于系统所需要的内存数量。虚拟内存就是为了克服这个矛盾而采用的策略。系统的虚拟内存通过在各个进程之间共享内存而使系统看起来有多于实际内存的内存容量。

虚拟内存可以提供以下的功能：

• 广阔的地址空间。

系统的虚拟内存可以比系统的实际内存大很多倍。

• 进程的保护。

系统中的每一个进程都有自己的虚拟地址空间。这些虚拟地址空间是完全分开的，这样一个进程的运行不会影响其他进程。并且，硬件上的虚拟内存机制是被保护的，内存不能被写入，这样可以防止迷失的应用程序覆盖代码的数据。

• 内存映射。

内存映射用来把文件映射到进程的地址空间。在内存映射中，文件的内容直接连接到进程的虚拟地址空间。

• 公平的物理内存分配。

内存管理系统允许系统中每一个运行的进程都可以公平地得到系统的物理内存。

• 共享虚拟内存。

虽然虚拟内存允许进程拥有自己单独的虚拟地址空间，但有时可能会希望进程共享内存。

10.2 虚拟内存的抽象模型

在讨论Linux系统虚拟内存的实现方法之前，让我们先看看虚拟内存的抽象模型。

当处理器执行一个程序时，它从内存中读取指令并解码执行。当执行这条指令时，处理器将还会需要在内存的某一个位置读取或存储数据。在一个虚拟内存系统中，所有程序涉及到的内存地址均为虚拟内存地址而不是机器的物理地址。处理器根据操作系统保存的一些信息将虚拟内存地址转换为物理地址。

为了让这种转换更为容易进行，虚拟内存和物理内存都分为大小固定的块，叫做页面。每一个页面有一个唯一的页面号，叫做PFN(page frame number)

在这种分页方式下，一个虚拟内存地址由两部分组成：一部分是位移地址，另一部分是PFN。每当处理器遇到一个虚拟内存地址时，它都将会分离出位移地址和 PFN地址。然后再将PFN地址翻译成物理地址，以便正确地读取其中的位移地址。处理器利用页面表来完成上述的工作。

图10-1是两个进程，进程 X和进程Y的虚拟内存示意图。两个进程分别有自己的页面表。

这些页面表用来将进程的虚拟内存页映射到物理内存页中。可以看出进程 X的虚拟内存页0映射到了物理内存页1，进程Y的虚拟内存页1映射到了物理内存4。页面表的每个入口一般都包括以下的内容：

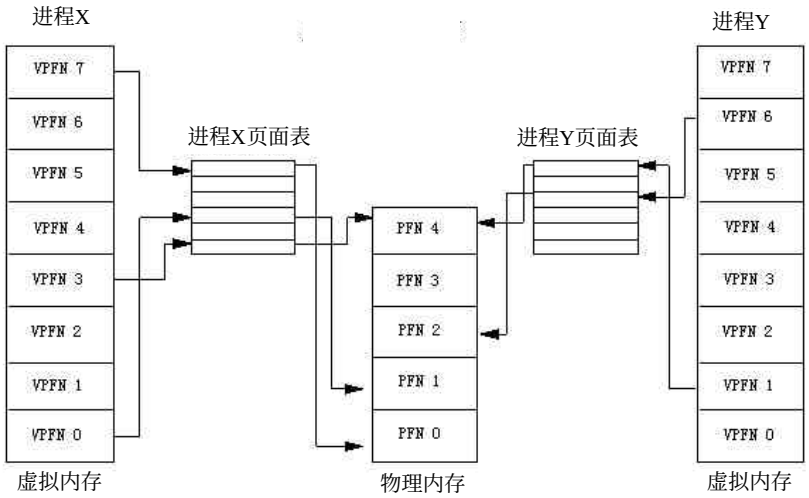

图10-1 虚拟内存示意图

• 有效标志。

此标志用于表明页面表入口是否可以使用。

• 物理页面号。

页面表入口描述的物理页面号

• 存取控制信息。

用来描述页面如何使用，例如，是否可写，是否包括可执行代码等。

处理器读取页面表时，使用虚拟内存页号作为页面表的位移，例如，虚拟内存页 5是页面表的第6个元素。

在将虚拟内存地址转换成物理内存地址时，处理器首先将虚拟内存地址分解为 PFN和位移值。例如，在上图中，一个页面的大小是0x2000 字节 (十进制的8192)，那么进程Y的一个虚拟内存地址0x2194将被分解成虚拟内存页号PFN为1和位移0x194。

然后处理器使用PFN作为进程页面表的位移值来查找页面表的入口。如果该入口是有效入口，处理器则从中取出物理内存的页面号。如果入口是无效入口，处理器则产生一个页面错误给操作系统，并将控制权交给操作系统。

假定此处是一个有效入口，则处理器取出物理页面号，并乘以物理页面的大小以便得到此物理页面在内存中的地址，最后加上位移值。

再看上面的例子：进程Y的PFN为1，映射到物理内存页号为4，则此页从0×8000 (4×0×2000)开始，再加上位移0×194，得到最终的物理地址为0×8194。

10.3 按需装入页面

由于物理内存要比虚拟内存小很多，所以操作系统一定要十分有效地利用系统的物理内存。一种节约物理内存的方法是只将执行程序时正在使用到的虚拟内存页面装入到系统的物理页面

中。当一个进程试图存取一个不在物理内存中的虚拟内存页面时，处理器将会产生一个页面错误给操作系统。如果发生页面错误的虚拟内存地址为无效的地址，说明处理器正在存取一个它不应该存取的地址。这时，有可能是应用程序出现了某一方面的错误，例如写入一个内存中的随机地址。在这种情况下，操作系统将会中止进程的运行，以防止系统中的其他进程受到破坏。

如果发生页面错误的虚拟内存地址为有效的地址，但此页面当前并不在物理内存中，则操作系统必须从硬盘中将正确的页面读入到系统内存。相对来说，读取硬盘要花费很长的时间，所以处理器必须等待直到页面读取完毕。如果此时有另外的进程等待运行，则操作系统将选择一个进程运行。从硬盘中读取的页面将被写入到一个空的物理内存页中，然后在进程的页面表中加入一个虚拟内存页面号入口。此时进程就可以重新运行了。

Linux系统使用需求时装入技术将可执行代码装入到进程的虚拟内存中。每当一个命令执行时，包括此命令的文件将被打开并映射到进程的虚拟内存中。此过程是通过修改描述进程内存映射的数据结构来实现的，通常被叫做内存映射。但此时只有文件镜像的第一部分被装入到了系统的物理内存中，而镜像的其他部分还保留在硬盘中。当此镜像执行时，处理器将产生页面错误，Linux使用进程的内存映射表决定应该把镜像的哪一部分装入到内存中执行。

10.4 交换

当一个进程需要把一个虚拟内存页面装入到物理内存而又没有空闲的物理内存时，操作系统必须将一个现在不用的页面从物理内存中扔掉以便为将要装入的虚拟内存页腾出空间。

如果将要扔掉的物理内存页一直没有被改写过，则操作系统将不保存此内存页，而只是简单地将它扔掉。如果再需要此内存页时，再从文件镜像中装入。

但是，如果此页面已经被修改过，操作系统就需要把页面的内容保存起来。这些页面称为“脏页面”（dirty page）。当它们从内存中移走时，将会被保存到一个特殊的交换文件中。

Linux系统使用一种叫做“最近最少使用”的技术（LRU）来决定把哪一个页面从物理内存中移出。系统中的每一个页面都有一个年龄，当一个页面被存取时，它的年龄将发生变化。页面被存取的越频繁，页面的年龄就越年轻；页面被存取的越少，它的年龄就越大。年龄大的页面将首先被交换出去。

10.5 共享虚拟内存

由于使用了虚拟内存，则几个进程之间的内存共享变得很容易。每个内存的存取都要通过页面表，而且每个内存都有自己的单独的页面表。如果希望两个进程共享一个物理内存页，只需将它们页面表入口中的物理内存号设置为相同的物理页面号即可。

10.6 存取控制

页面表中还包括存取控制信息，这样，在处理器使用页面表把进程的虚拟内存地址转换为物理内存地址时，可以方便地使用存取控制信息来检查进程是否存取了它不该存取的信息。

使用存取控制信息是完全必要的。例如，一些内存中包括可执行代码，而这些可执行代码通常为只读，操作系统将不会允许向一段只读代码中写入数据。同样，包括数据的内存通常为可读写的，而试图执行此内存中的代码将产生错误。大多数的处理器有内核和用户两种可执行方式。用户不能执行内核的代码，而内核的数据用户也无法存取。

10.7 高速缓存

为了获得最大的系统效用，操作系统一般使用高速缓存来提高系统性能。 Linux系统使用了几种涉及到高速缓存的内存管理方法。

10.7.1 缓冲区高速缓存

缓冲区高速缓存中保存着块设备驱动程序所用到的数据缓冲区。

这些缓冲区的大小固定，一般包括从块设备中读入的和将要写入到块设备中的信息块。快设备一次只能处理大小固定的数据块。硬盘就是块设备中的一种。

缓冲区高速缓存使用设备标识符和块号作为索引来快速地查找数据块。块设备只通过缓冲区高速缓存进行存取。如果所需要的数据存在于缓冲区高速缓存中，那么就不需要从物理块设备中读取，这样存取的速度就会加快。

10.7.2 页面高速缓存

页面高速缓存用来加速磁盘中文件镜像和数据的存取，我们将在后面的 10.12节中对它进行详细讨论。

10.7.3 交换高速缓存

交换文件中只保存那些被修改过的页面。

只要在页面被写入到交换文件中后没有被修改过，那么此页面下一次从内存中交换出来时就不用再写入到交换文件中了，因为交换文件中已经有了该页面。这样，该页面就可以简单地扔掉，节省了大量的系统操作。

10.7.4 硬件高速缓存

一个常用的硬件高速缓存是在处理器中，它一般保存着页面表的入口。

10.8 系统页面表

Linux系统共有三级的页面表。上一级页面表的PFN指向下一级页面表的入口。如图10-2所示，一个虚拟内存地址分成了几个字段，每个字段提供一个相应的页面表位移值。要把一个虚

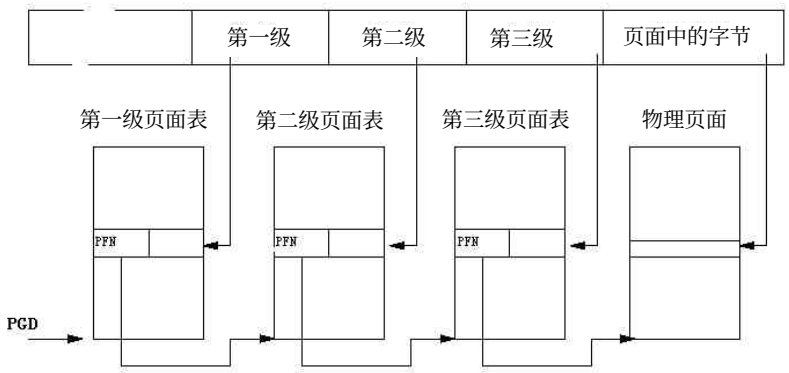

图10-2 虚拟内存地址示意图

拟内存地址翻译成一个物理内存地址，处理器必须读取每一个字段的内容，把其转化成包含页面表的物理页的位移值，再取出其中指向下一个页面表的 PFN。这个过程重复三次，直到找到包含物理页面号的虚拟内存地址。

Linux运行的所有平台都提供地址翻译的宏函数以便内核为某一个进程转化页面表。这样，内核就无需知道页面表入口的格式以及它们是如何安排的。

10.9 页面的分配和释放

系统在运行时会经常地需要物理内存页。例如，当一个文件镜像从磁盘调入到内存时，操作系统需要为它分配物理内存页。当程序执行完毕时，操作系统需要释放内存页。物理页的另一个用途是存储内核所需要的数据结构，例如页面表。页面的分配和撤消机制以及所涉及的数据结构对内存管理来说是至关重要的。

系统中所有的物理内存页都包括在 mem_map数据结构中，而mem_map是由mem_map_t结构组成的链表。mem_map_t在系统启动时初始化。每个mem_map_t结构都描述了系统中的一个物理页。其重要的字段有：

• count

此字段用于记录使用此页面的用户数。当几个进程共享此物理内存页时， count的值将会大于1。

• age

此字段描述了页面的年龄，通过此字段，操作系统可以决定是否将此页面扔掉或交换出去。

• map_nr

此字段为页面的物理页面号。

页面分配程序使用free_area向量查找和释放页面。对于页面分配程序来说，页面本身的大

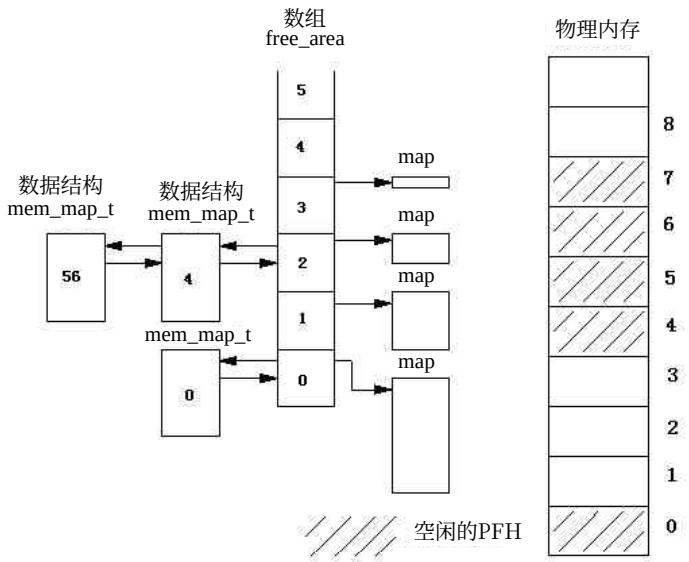

图10-3 空闲内存示意图

小和处理器使用的物理页面机制是无关的。

free_area中的每个元素都包括页面块的信息。第一个元素描述了单个页面的页面块，第二个元素描述了两个页面的页面块，第三个元素描述了四个页面的页面块，以次类推以 2的次方数增加。向量中的list元素用来作为指向mem_map数据结构中page结构的队列的头指针，指向空闲的页面。指针 map指向同样大小页面组的一个位图。如果第 N个页面块是空闲的，那么该位图的第N位置1。

图10-3显示了free_area结构。元素0有一个空闲页块（页号0），元素2有两个空闲页块，一个从页号4开始，另一个从页号56开始。

10.9.1 页面的分配

Linux系统使用Buddy算法来分配和释放页面块。如果系统对于请求的分配有足够的空闲页面(nr_free_pages > min_free_pages)，页面分配程序将会查找free_area以便找到一个和请求的页面块大小相同的页面块。它根据 free_area中list元素指向的空闲页面队列进行查找。如果没有同样大小的空闲页面块，则继续查找下一个空闲页面块（其大小为上一个页面块的 2倍）。如果有空闲的页面块，则把页面块分割成所请求的大小，返回到调用者。剩下的空闲页面块则插入在空闲页面块队列中。

10.9.2 页面的释放

以上页面块的分配策略会造成将一个个大的内存块分割成小块的结果。而内存页面释放程序却总是试图将一个个的比较小的页面块合并为大的页面块。

每当一个页面块释放时，页面释放程序就会检查其周围的页面块是否空闲。如果存在空闲的页面块，则空闲的页面块就会和释放的页面块合并在一起组成更大的页面块。

10.10 内存映射

当执行一个文件镜像时，可执行镜像的内容必须装入到进程的虚拟地址空间。可执行镜像

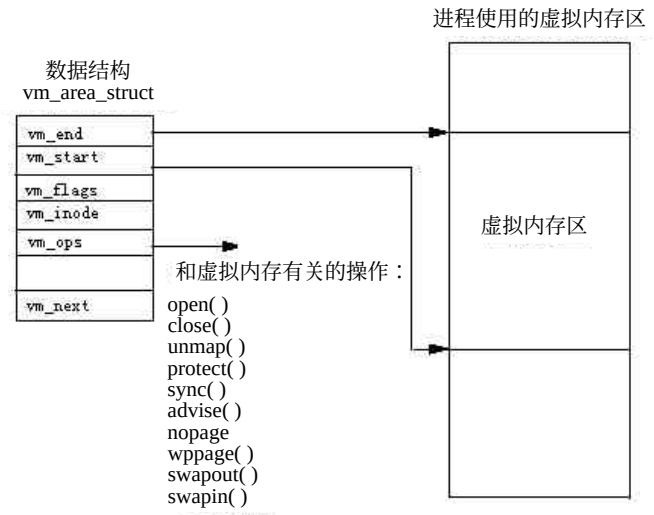

图10-4 虚拟内存数据结构示意图

链接的共享库也是一样要装入到虚拟内存空间。可执行文件并不是真正地装入到物理内存空间，它只是简单地链接到进程的虚拟内存。然后，随着应用程序运行时的需要，可执行镜像才逐渐地装入到物理内存。这种将一个文件的镜像和一个进程的虚拟内存地址空间连接起来的方法叫做内存映射。

数据结构mm_struct代表每个进程的虚拟内存空间。它包含了正在执行的镜像的信息和一些指向vm_area_struct结构的指针，如图10-4所示。每个vm_area_struct结构都描述了进程的虚拟内存的起始和结束位置，进程的存取权限以及和内存有关的一系列的操作。这些操作是Linux系统在处理虚拟内存时将要用到的。

当一个可执行镜像映射到一个进程的虚拟内存地址时，操作系统将创建一系列的数据结构vm_area_struct，每一个vm_area_struct代表可执行镜像的一部分。Linux系统支持多种标准虚拟内存操作，创建vm_area_struct时，相应的虚拟内存操作就会和vm_area_struct链接起来。

10.11 请求调页

一旦一个可执行镜像映射到了一个进程的虚拟内存中，它就可以开始执行了。因为开始时只有镜像开头的一小部分装入到了系统的物理内存中，所以不久进程就会存取一些不在物理内存中的虚拟内存页，这时处理器会通知 Linux发生了页面错误。页面错误将会描述页面错误发生时的虚拟内存地址和存取内存操作的类型。

Linux首先查找代表发生页面错误的虚拟内存区的 vm_area_struct。如果没有代表此出错虚拟地址的vm_area_struct，就说明进程存取了一个非法的虚拟内存地址。 Linux系统将向进程发出SIGSEGV信号。

Linux下一步检查页面错误的类型是否和此虚拟内存区所允许的操作类型相符。如果不符，Liniux系统也将报告内存错误。

如果Linux认为此页面错误是合法的，它将处理此页面错误。

Linux还必须区分页面是在交换文件中还是作为文件镜像的一部分存在于磁盘中。它靠检查出错页面的页面表来区分：

如果页面表的入口是无效的，但非空，说明页面在交换文件中。

最后，Linux调入所需的页面并更新进程的页面表。

10.12 页面高速缓存

Linux系统中页面高速缓存的作用是加快对磁盘中的文件的存取。对于已经作好了磁盘映射的文件，Linux每次读取一页，并将读取的页面存储到页面高速缓存中。

图10-5显示页面高速缓存由 page_hash_table组成， page_hash_table是一个包含指向mem_map_t结构指针的数组。

每当从一个内存映射文件中读取一个页面时，页面都要从页面高速缓存中读取。如果页面在高速缓存中，则将一个指向mem_map_t的指针返回给页面错误处理程序。否则，页面必须从磁盘上读入到内存中。

如果可能，Linux系统将会提前读取文件中的下一个页面，这样，如果文件是顺序执行的，那么下一个页面就已经在内存中了。

随着文件的读入和执行，页面高速缓存也将变得越来越大。不用的页面将被移出高速缓存。

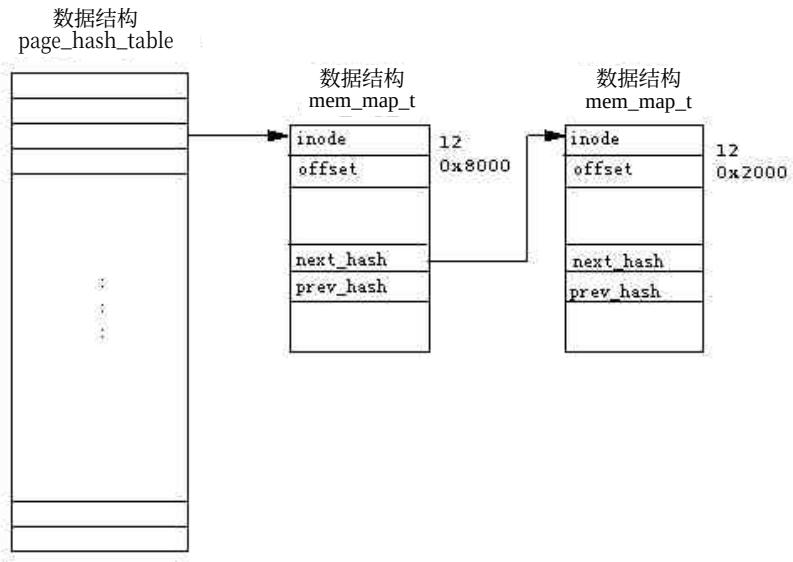

图10-5 页面缓存示意图

10.13 内核交换守护进程

当物理内存变少时，Linux内存管理必须释放物理内存页。此任务由内核中的交换守护进程(kswapd)完成。

交换守护进程是一个内核线程。内核线程是无需使用虚拟内存的进程，它们在物理内存中运行于内核方式下。交换守护进程还有一个重要的任务是保证系统中有足够的空闲内存。

交换守护进程是由内核的初始化进程启动的，并一直等候直到内核交换计时器周期性截止。

每当计时器截止时，交换守护进程都要检查系统中的空闲的页面数是否太少。它使用两个变量free_pages_high 和 free_pages_low来决定是否需要释放一些内存。只要空闲的内存数大于free_pages_high，交换守护进程就不做任何事，它将再一次进入睡眠状态直到计时器再一次截止。

检查系统中空闲页的目的是使交换守护进程可以计算出需要往交换文件中写入的页面数，此数目保存在nr_async_pages中。每次有页面排队准备写入到交换文件中时，nr_async_pages的值就会增加，而当写入结束后，nr_async_pages的值就将减少。free_pages_low 和 free_pages_high是在系统启动时设定的，并和系统的物理页面数有关。如果系统中的空闲页面数低于free_pages_high，或者甚至低于free_pages_low，则内核的交换守护进程将用以下三种办法减少正在使用的页面数：

- 减少缓冲区和页面高速缓存的大小。
- 把System V的共享内存页交换出系统内存。
- 交换或扔掉内存页。

如果系统中的空闲页面数低于free_pages_low，则内核交换守护进程将试图一次空出6个内存页，否则，内核交换守护进程将空出 3个内存页。上述的三种办法将轮流使用，直到空闲出足够的内存页为止。内核交换守护进程将会记录最后一次释放内存时使用的方法，下一次它再运行时，将会首先使用此方法。

在释放出足够的内存页之后，内核交换守护进程将进入睡眠状态，直到计时器截止。如果上一次的交换是由于内存空闲的页面数低于free_pages_low，则内核交换守护进程只睡眠一半的时间。

第11章　进程间通信

本章介绍有关进程间通信方面的内容。

系统中的进程和系统内核之间，以及各个进程之间需要相互通信，以便协调它们的运行。Linux系统支持多种内部进程通信机制（IPC），其中最为常用的是信号、管道以及Unix系统支持的System V IPC机制。

11.1　信号机制

信号(signal)机制是Unix系统中最为古老的进程之间的通信机制。它用于在一个或多个进程之间传递异步信号。信号可以由各种异步事件产生，例如键盘中断等。 Shell 也可以使用信号将作业控制命令传递给它的子进程。

Linux系统中定义了一系列的信号，这些信号可以由内核产生，也可以由系统中的其他进程产生，只要这些进程有足够的权限。你可以使用 kill命令（kill -l）在你的机器上列出所有的信号，一般如下所示：

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGIOT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	17) SIGCHLD
18) SIGCONT	19) SIGSTOP	20) SIGTSTP	21) SIGTTIN
22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO
30) SIGPWR			

进程可以屏蔽掉大多数的信号，除了SIGSTOP和SIGKILL。SIGSTOP用于暂停一个正在运行的进程，而SIGKILL使得正在运行的进程退出运行。进程可以选择系统的缺省方式处理信号，也可以选择自己的方式处理产生的信号。信号之间不存在相对的优先权，系统也无法处理同时产生的多个同种的信号，也就是说，进程不能分辨它收到的是1个或者是42个SIGCONT信号。

Linux使用进程中task_struct结构中的信息来实现信号。系统支持的信号数量受到处理器中字的大小的限制，例如， 32位的处理器可以支持 32个信号。当前等待处理的信号保存在task_struct中的signal字段中。每个进程的task_struct 中包括一个指向数组sigaction的指针，此数组是关于每一个进程如何处理可产生的信号的信息，其中有处理信号的子过程的地址，或者包括某种标志位，可以通知Linux系统进程希望忽略此信号或者进程希望 Linux系统本身处理此信号。进程可以通过系统调用修改系统缺省的信号处理程序。

除了内核和超级用户，并不是每个进程都可以向其他的进程发送信号。一般的进程只能向具有相同uid 和gid的进程发送信号，或向相同进程组中的其他进程发送信号。可以通过设置进程中task_struct结构中的signal字段的适当位来产生信号。如果进程没有屏蔽信号并且处于等待运行可中断状态，那么进程将被唤醒并进入到运行状态。这样调度程序将会在下一次调度时考虑运行此进程。

信号在产生时并不马上送给进程，信号必须等待直到进程再一次运行。每当一个进程从系统调用中退出时，系统都将检查进程的信号和屏蔽字段。如果有任何不屏蔽的信号，则可以立

写入进程使用标准的写入函数将数据写入到管道中。这些函数把文件描述符传递到进程的文件数据结构中，每一个文件数据结构都代表一个打开的文件，或者在此是一个打开的管道。Linux的系统调用使用描述此管道文件数据结构中的指针指向写入子进程，在 VFS索引节点存储有关此写入子进程的信息。

如果有足够的空间可以一次将所有的字节写入到管道中，并且读取子进程没有锁定管道，那么Linux系统将会使写入子进程锁定管道，并将从进程的地址空间中将写入的字节复制到共享数据页面中。如果管道没有足够的字节或者读取进程锁定了管道，那么当前进程在管道索引节点的等待队列中睡眠，这时调度进程可以调度运行其他等待运行的进程。因为睡眠的进程是可以中断的，所以它可以接收信号，当有足够的写入空间或者管道解锁时，读取进程可以唤醒睡眠的写入进程。

从管道中读取数据的过程和写入数据的过程十分类似。

系统允许进程使用无屏蔽的读取，也就是说，当没有数据可供读取时，或者管道锁定时，读取进程将会返回一个错误信息。这意味着，进程可以不必等待而继续运行。当写入和读取进程都完成对管道的操作时，系统将会放弃管道的索引节点和共享的内存页面。

Linux系统也支持命名管道，也叫做 FIFO。FIFO遵循先入先出的原则和一般的管道不同，FIFO不是临时的，而是文件系统的一部分。你可以用 mkfifo命令创建一个FIFO。只要有适当的权限，进程可以自由地使用 FIFO。FIFO的打开方式和一般管道的打开方式不一样。 Linux必须处理以下的情况，例如读取进程试图在写入进程打开 FIFO之前打开FIFO，或者在写入进程写入任何数据之前读取进程试图读取。除此之外， FIFO和一般管道一样使用相同的数据结构和操作。

11.2 System V IPC 机制

Linux系统支持三种类型的进程间通信机制，它们是信息队列、信号量和共享内存。这三种通信机制首先出现在Unix TM System V (1983)中,它们使用相同的授权方法。进程只有通过使用系统调用传递给系统内核一个唯一的参考标识符来存取这些资源。进程可以使用系统调用来设置System V IPC目标的存取权限。每个通信机制都使用目标的参考标识符作为资源列表的索引。这些标识符并不是真正的索引，而必须通过一定的转化才能生成实际的索引。

系统中所有代表System V IPC的目标数据结构中都包括一个ipc_perm结构，此结构中包含进程的用户和工作组标识符的拥有者和创建者，以及此目标的存取方式和 IPC目标关键字。其中的关键字作为一种定位System V IPC目标的参考标识符的方法。系统支持两类关键字：公共的和私人的。如果是公共关键字，那么系统中的任何进程，只要遵循权限检查，都可以利用它找到System V IPC目标的参考标识符。

11.2.1 信息队列

信息队列即一个或多个进程写入信息，同时一个或多个进程也可以读取此信息。 Linux系统中有一个信息队列的链表 msgque，此链表中每一个元素都指向一个叫做 msqid_ds的数据结构。msqid_ds结构保存着信息队列的有关信息。每当一个信息队列创建时，系统将从内存中分配一个新的数据结构msqid_ds，并将其插入到链表中。

每一个msqid_ds结构中都包含一个ipc_perm结构以及指向此队列中的信息的指针。另外，msqid_ds中还包括有关队列修改时间的一些信息，例如最后一次修改的时间等。它还包含两个等待队列：一个是写入进程的等待队列，另一个是读取进程的等待队列。

系统将把第二个进程挂起，直到第一个进程完成对数据文件的操作为止。第一个进程完成对数据文件的操作时，将会对信号量的值加 1，使得信号量的值从0又恢复到了1。现在，被挂起的进程将会被唤醒，它将再一次试图把信号量的值加1，而这一次它将会允许运行。

Linux系统使用数据结构semid_ds表示 System V IPC中的信号量目标，如图11-3所示。系统中的每一个信号量目标都代表一个信号量数组。系统中一个叫做 semary的指针数组保存着系统中所有的semid_ds的地址。每一个信号量数组都包括 sem_nsems，而每个sem_nsems又都描述了由semid_ds中的指针sem_base指向的sem结构。系统中所有允许使用信号量数组的进程可以通过系统调用来执行信号量的操作。系统调用可以指定很多的操作，每一个操作都由三个输入来描述：信号量索引、操作值和一系列的标志。信号量索引是信号量数组中的索引值，操作值是希望和当前信号量的值相加的数值。首先 Liunx将会检测所有的操作是否会成功。如果操作值和当前信号量值相加后大于 0，或者如果操作值和信号量当前值都为 0的话，操作就会成功。如果任何一个操作失败的话，并且当操作标志没有要求系统调用是无屏蔽时， Linux将会挂起此操作。如果进程被挂起， Linux就将把要执行的信号量操作的状态保存起来，并将当前的进程插入到等待队列中。系统需要在堆栈中建立一个数据结构 sem_queue，并把进程的状态保存到sem_queue中。新建立的数据结构 sem_queue将会被插入到等待队列的末尾（使用 sem_pending 和 sem_pending_last指针）。

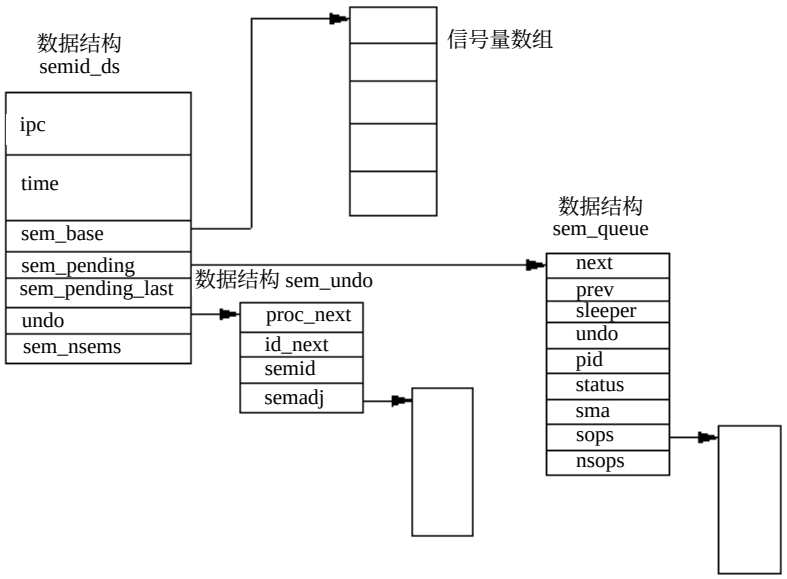

图11-3 信号量结构示意图

如果所有的信号量操作都取得成功，那么 Linux将会继续进行信号量数组中适当元素的操作。此时， Linux系统也必须检查任何等待的挂起进程是否可以进行它们的操作。如果有任何的进程可以进行信号量操作，那么它将从等待队列中移走数据结构 sem_queue，并对信号量数组中适当的元素进行信号量操作。

信号量操作可能会出现僵局。当一个进程修改了信号量以后，它开始运行信号量以后的操作，但在此期间，此进程由于崩溃或者由于被撤消而不能正常地离开操作区域，这时将会发生僵局。Linux系统通过维护信号量数组的调整链表来避免这个问题。其方法是，当信号量改变后，信号量将被调整到操作以前的值。系统把此调整保存在数据结构 sem_undo中，而数据结

到由shmid_ds中的指针指向的vm_area_struct数组中。vm_area_struct结构使用vm_next_shared和 vm_prev_shared指针将vm_area_struct链接起来。

当进程第一次存取共享虚拟内存的其中一个页面时，就会产生页面错误。当 Linux系统恢复页面错误时，系统将会查找描述页面错误的 vm_area_struct结构。vm_area_struct结构中包含指向这种共享虚拟内存错误的处理程序的指针。共享内存错误处理程序将会进一步在内存页面表链表中查找此shmid_ds结构的入口，以便了解共享虚拟内存的页面是否存在。如果此共享内存的页面不存在，系统将为其分配一个物理页，并同时为其创建一个新的页面表入口。此入口既会插入到进程的页面表中，也会保存到 shmid_ds中。这就意味着当下一个试图存取此页面的进程发生页面错误时，共享内存处理程序将会使用这个新创建的物理页。所以，第一个存取共享内存页面的进程使得系统创建共享内存页面，而以后其他进程存取此共享内存页面将导致此页面添加到进程的虚拟地址空间中。

当进程不希望再使用共享内存时，它将从共享内存上脱离。只要还有其他的进程在使用此共享内存页面，则进程的脱离只会影响到该进程本身。进程的 vm_area_struct结构将从 shmid_ds 结构中移走，进程的页面表也将被更新，其中共享的虚拟内存将变成无效的区域。当共享内存上的最后一个进程从共享内存中脱离时，共享内存页面所占用的物理内存将会被释放，共享内存的shmid_ds 结构也将会被释放掉。

如果共享内存没有被锁定到物理内存中，那么处理的情况将会比较复杂。在这种情况下，共享内存页面有可能被交换到系统的交换文件中。有关内存交换的过程请参考第 10章“ 内存管理 ”。

第12章 PCI

PCI是外围部件互连（Peripheral Component Interconnect）的简称，它是一种描述如何将一个系统的外围部件用一种结构化和易于控制的方式连接在一起的标准。 PCI标准具体描述了系统部件的电气特性和它们应该具有的接口。下面我们看看 Linux系统内核是如何初始化系统的PCI总线和PCI设备的。

12.1 PCI 系统

图12-1是一个PCI总线系统的逻辑示意图。 PCI总线和PCI-PCI桥将系统的各部件连接在一起，其中CPU和Video设备连接到PCI总线0，也就是主PCI总线。PCI-PCI桥一个特殊的PCI设备，它将主PCI总线和从PCI总线，也就是PCI总线1，连接在一起。PCI总线1也称为PCI-PCI桥的下游，而PCI总线0称为PCI-PCI桥的上游。连接到从PCI总线上的是系统的SCSI设备和以太网设备。在物理上， PCI-PCI桥、从PCI总线和连接到从PCI总线上的这两个设备将有可能做在一块PCI卡上。PCI-ISA桥用于连接比较老的ISA设备。

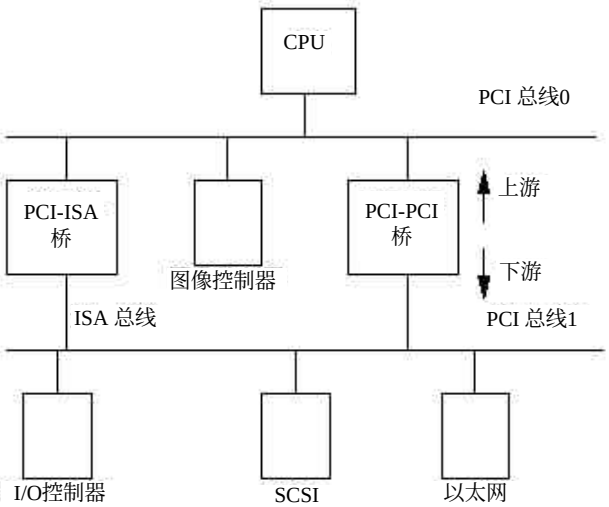

图12-1 PCI系统示意图

12.2 PCI地址空间

CPU和各种PCI设备都需要存取系统内存。设备驱动程序需要使用系统内存控制 PCI设备，并且可以通过系统内存在PCI设备之间传递信息。一般情况下，共享内存中包括设备的控制及状态寄存器。驱动程序可以利用这些寄存器来控制和读取设备的状态。例如， PCI SCSI设备驱动程序可以读取设备的状态寄存器，以便了解PCI SCSI设备是否已经准备好了向PCI SCSI磁盘

写入信息；或者，驱动程序可以在设备打开以后，向控制寄存器中写入命令启动设备。

CPU的系统内存可以用来作为这样的共享内存。但如果使用 CPU的系统内存，那么每当PCI 设备存取内存时，CPU都要停止以等待PCI设备完成存取操作。并且，一般一次只能有一个系统部件存取内存。这样，系统的性能将大大地降低。如果让系统的外围设备无限制地存取系统的主内存也十分的危险，因为一个失去控制的设备有可能使得系统瘫痪。

所以，系统的外围设备一般自己带有内存。CPU可以存取这些外围设备的内存，但外围设备只能通过DMA(Direct Memory Access)方式存取系统的内存。ISA设备有两种的地址空间：ISA I/O (Input/Output) 和 ISA 内存。PCI设备则有三种地址空间：PCI I/O, PCI 内存 和 PCI 设置。CPU可以通过设备驱动程序存取PCI I/O和 PCI 内存地址空间，并且可以通过Linux系统内核中的PCI初始化程序存取 PCI设置地址空间。

12.3 PCI设置头

系统中每一个PCI设备，包括PCI-PCI桥，都包含一个存储在PCI地址空间中有关设置的数据结构，即PCI设置头。系统通过PCI设置头识别和控制设备。设置头在PCI地址空间的具体位置依据PCI的拓扑结构的不同而不同。例如，一个 PCI视频卡插入到主板的某一个 PCI插槽中，那么它的设置头将在地址空间中有一个固定的位置，但如果将 PCI视频卡插入到主板的另一个PCI插槽中，它将又有另外的一个位置。但无论 PCI设备的设置头在哪，系统都能找到它，并且可以通过设置头中的状态及设置寄存器来设置它。

一般情况下，主板上的每一个 PCI插槽都对应一个相对固定的PCI设置头的地址。例如，主板上的第一个插槽的设置头从地址空间位移 0开始，第二个插槽的设置头从位移256开始（所有的设置头都是256字节），以此类推。根据系统硬件的不同，在一个给定的PCI总线中，PCI设置程序可以试着检查所有可能的PCI设置头，同时只需简单地读取设置头中的某一个字段（通常是厂家标识符字段），就可以知道系统中有无PCI设备。例如，对于一个没有 PCI设备的PCI插槽，如果试图读取其设置头的厂家标识符字段，将会返回一个0xFFFFFFFF错误。

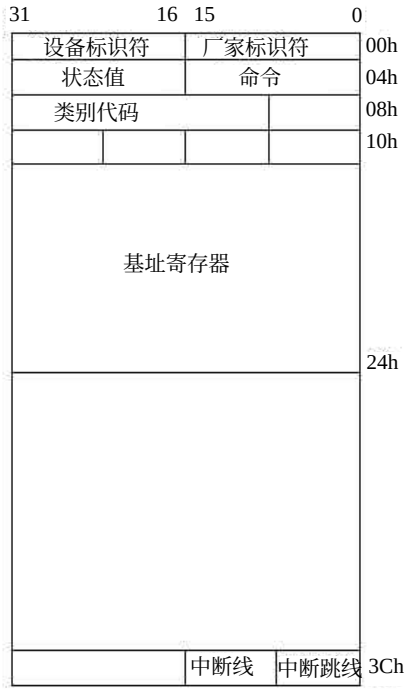

图12-2 PCI设置头示意图

图12-2是一个包括256个字节的设置头，其中的字段有：

- 厂家标识符：代表PCI设备厂家的一个唯一的数值。例如，Intel公司的厂家标识符是0x8086。
- 设备标识符：代表设备自身的一个唯一的数值。例如，Digital公司的21141型快速以太网卡的设备标识符是0x0009。
- 状态：此字段是设备的状态，其中的每一位的含义都有标准的规定。
- 命令：通过向此字段中写入命令，系统可以控制设备。
- 类别代码 :此字段代表设备的类型。例如，SCSI的类别代码是0x0100。
- 基址寄存器：这些寄存器用来决定和分配设备可以使用的 PCI I/O 和 PCI内存地址空间的

类型，数量和位置。

- 中断跳线：代表PCI卡上的4根中断跳线。通过这4根跳线，可以在PCI卡和PCI总线之间传递中断。4根跳线的标准标志为A、B、C、D。中断跳线字段描述PCI设备使用了哪一个中断跳线。一般情况下，对于某一个设备来说，它是由硬件决定的。中断处理程序可以通过此字段管理设备的中断。
- 中断线：此字段用来在PCI初始化程序、设备驱动程序以及Linux系统的中断处理程序之间传递中断句柄。此字段的数值对于设备来说没有任何意义。但它可以使得中断处理程序在Linux操作系统内正确地把PCI设备的一个中断送到正确的设备驱动程序的中断处理过程中。

12.4 PCI I/O 和 PCI 内存地址

PCI设备和它们运行在Linux系统内核上的设备驱动程序之间进行通信的地址空间有两种：PCI I/O 和 PCI 内存。例如，DEC公司的21141快速以太网设备把它的内部寄存器映射到 PCI I/O地址空间。Linux系统中的设备驱动程序通过读写这些寄存器来控制 PCI设备。而视频驱动程序一般使用较大的PCI内存地址空间来保存视频信息。

在PCI系统建立之前，或者在使用 PCI设置头的命令字段打开设备对地址空间的存取之前，任何程序都不能存取这些地址空间。应该注意的是，只有 PCI设置程序才可以读写包含PCI设置头的PCI设置地址，而Linux系统的设备驱动程序只能读写 PCI I/O 和 PCI 内存地址。

12.5 PCI-ISA桥

PCI-ISA桥通过将对PCI I/O 和 PCI 内存地址空间的访问转换成对ISA I/O 和 ISA内存地址空间的访问来兼容对以前的ISA设备的支持。系统保留低端的PCI I/O 和 PCI 内存地址空间提供给ISA外围设备使用，同时使用一个 PCI-ISA桥来把对此内存区域的访问转换成对 ISA的访问。

12.6 PCI-PCI 桥

PCI-PCI 桥是一种特殊的PCI设备，它把系统中的多个PCI总线集成为一个。单个PCI总线支持的设备数目有一定的限制。使用PCI-PCI 桥可以使系统支持更多的PCI设备。

PCI-PCI 桥只传送其下游需要的读写内存地址。例如，在PCI总线逻辑示意图中，PCI-PCI桥只传递从PCI总线0到PCI总线1中SCSI设备和快速以太网设备拥有的读写地址，所有的其他PCI I/O 和 PCI 内存地址都将被忽略。这样就防止了系统中不必要的地址传播。为了达到此目的，PCI-PCI桥必须设置一个它要传递的PCI I/O 和 PCI内存地址的起点和大小限制。一旦系统中的PCI-PCI桥设置完毕，那么Linux系统的设备驱动程序只通过设置的窗口来访问 PCI I/O 和PCI内存地址，PCI-PCI桥本身对驱动程序来说则是透明的。这对 PCI设备驱动程序的编写者来说十分重要。这也使得Linux系统对PCI-PCI桥的设置更为复杂。

因为PCI初始化程序需要寻找那些不在主 PCI总线上的设备，所以PCI-PCI桥必须可以判断是否需要从主PCI总线向从PCI总线传递设置环路。一个环路就是 PCI总线上的地址。 PCI中规定了两种PCI设置地址的格式：类型0和类型1，如图12-3所示：

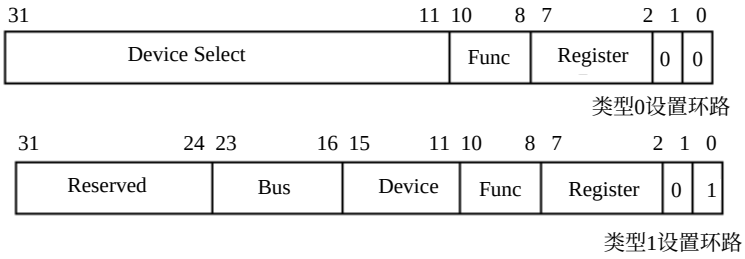

图12-3 PCI地址格式

在类型0设置环路中不包括总线值，所以它可以被PCI总线上的所有设备视为设置地址。类型0设置环路的31位到11位被视为设备选择字段。你可以使字段的每一位都代表一个不同的设备，那么第11位将会是插槽0中的PCI设备，第12位是插槽1中的PCI设备，以此类推。也可以把设备的插槽号直接地写入字段中。系统中到底使用哪一种机制取决于系统的 PCI内存控制器。

类型1设置环路包括一个PCI总线值，所以这种总线环路只有对PCI-PCI桥才有效。PCI-PCI桥根据自己的设置决定是否将类型1的设置环路传送到下游。每一个PCI-PCI桥都有一个主总线接口号和一个从总线接口号。主总线是靠近 CPU的总线，从总线是远离 CPU的总线。每一个PCI-PCI桥还有一个下属总线数目，也就是PCI-PCI桥下游的最多可以有的总线数。当一个PCI-PCI桥检测到一个类型1的设置环路时，它将按以下的方式之一来处理：

- 如果设置环路中的总线号不在 PCI-PCI桥的从总线号和从总线的下属总线号之间， PCI-PCI桥将忽略此设置环路。
- 如果设置环路中的总线号和PCI-PCI桥的从总线号相符， PCI-PCI桥将会把类型1的设置环路转化成为类型0的设置环路。
- 如果设置环路中的总线号大于 PCI-PCI桥的从总线号，但小于等于从总线的下属总线号，PCI-PCI桥将会把此设置环路不加任何改变地传送到从总线的接口中。

12.7 PCI初始化

Linux系统中PCI设备的初始化程序分为以下三个逻辑部分：

- PCI 设备驱动程序

这是一个伪设备驱动程序，它从总线 0开始搜索整个PCI系统，并且定位系统的所有 PCI设备和总线桥。它创建一个描述系统拓扑结构的数据结构链表。同时，它也将可以找到的所有的总线桥编号。

- PCI BIOS

此软件层提供在bib-pci-bios-specification中规定的各种函数。

- PCI Fixup

不同系统的Fixup程序用来处理不同系统中PCI初始化后的遗留问题。

12.7.1 Linux系统内核有关PCI的数据结构

当Linux系统内核初始化PCI系统时，它将创建一些代表系统中实际 PCI拓扑结构的数据结构。图12-4所示的数据结构关系即反映了本章开始时的PCI系统的拓扑结构。

每个数据结构pci_dev都描述一个PCI设备，包括PCI-PCI桥。每个pci_bus结构则描述一个PCI总线。这是一个树型的PCI总线结构，每一个PCI总线下面带有多个PCI设备。因为除了主

PCI总线以外的其他PCI总线只能由PCI-PCI桥连接，所以每个描述从PCI总线的pci_bus结构中都包括一个指向PCI-PCI桥的指针，而此PCI-PCI桥是此从PCI总线的父亲PCI总线的孩子。

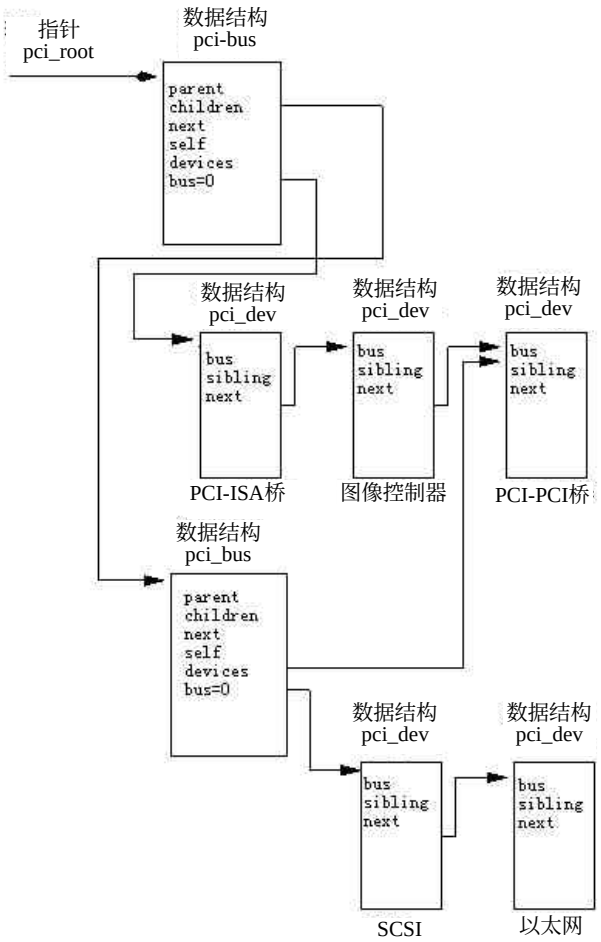

图12-4 PCI结构示意图

系统中所有的PCI设备的pci_dev结构组成了一个队列，pci_devices指针指向此队列。Linux系统内核使用此队列快速搜索系统的PCI设备。

12.7.2 PCI 设备驱动程序

这里的PCI设备驱动程序不是真正的设备驱动程序，而只是操作系统在系统初始化时调用的一个函数。PCI初始化程序必须搜索系统中所有的PCI总线，以便定位系统中全部的PCI设备，包括PCI - PCI桥。

PCI初始化程序通过PCI BIOS程序确定当前正在搜索的PCI总线中的每一个插槽是否被占用。PCI初始化程序从PCI总线0开始扫描，它首先试着读取每一个PCI插槽中可能存在的PCI设备的厂家标识符和设备标识符。如果对某一个插槽读取成功，那么说明此PCI插槽已经被占用，PCI设备驱动程序将创建一个描述此设备的数据结构pci_dev，并且将它插入到已经存在的所有PCI设备的链表。（pci_devices指针指向此链表）

PCI初始化程序如果发现设备是一个 PCI-PCI桥，那么将创建一个pci_bus结构，然后将它插入到pci_bus结构树中。PCI初始化程序通过类别代码 0x060400来确定此设备是否为PCI-PCI桥。如果是PCI-PCI桥，系统内核则设置PCI-PCI桥下游的PCI总线。如果在其中再发现PCI-PCI桥,则继续设置PCI-PCI桥下游的PCI总线。

只有知道了以下内容，PCI-PCI桥才能传递PCI I/O, PCI 内存或 PCI设置地址：

- 主总线号
- 从总线号
- 下属总线号：PCI-PCI桥的下游的所有总线的最大的总线号。
- PCI I/O 和 PCI内存窗口：PCI-PCI桥的下游地址的PCI I/O 和 PCI内存地址空间的窗口起点和大小。

问题是当你希望设置某一个PCI-PCI桥时，你并不知道该PCI-PCI桥的下属的总线号。也不知道PCI-PCI桥的下游是否还有 PCI-PCI桥，如果有的话，它们的总线号是多少。 Linux系统内核使用深度搜索算法搜索每一个总线号。每当找到一个 PCI-PCI桥和指定它的从总线号时，同时赋给该PCI-PCI桥一个临时的下属号 —— 0xFF，然后搜索和指定该 PCI-PCI桥下游所有 PCI-PCI桥的总线号。下面举例说明如何设置总线号：

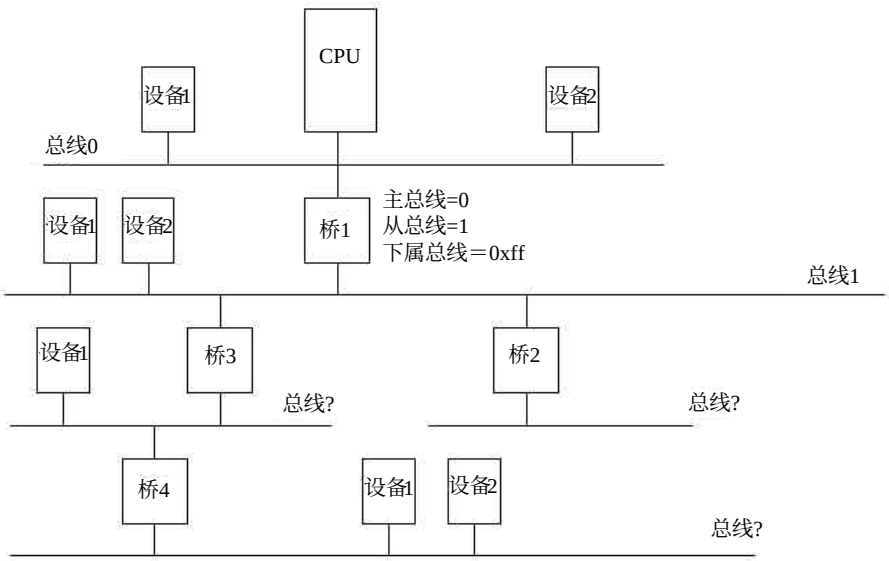

图12-5 PCI-PCI 设置示意图一

第一步：

如图12-5所示，初始化程序首先找到的是PCI - PCI桥1。它下游的总线将被设置为1，同时指定下属总线号为0xFF。这意味着所有类型1设置环路中凡是PCI总线号大于1的都将通过PCI-PCI桥1进入 PCI总线1。如果设置环路中的总线号是1，那么类型1设置环路将会被转换成类型0设置环路，但其他总线号的设置环路的类型则保持不变。这也正是 Linux系统的PCI初始化程序通过和扫描PCI总线1时所需要做的。

第二步：

Linux系统的PCI初始化程序使用的是深度搜索算法，所以它继续搜索 PCI总线1。如图12 - 6所示，初始化程序找到了 PCI-PCI桥2。因为PCI-PCI桥2以下再没有PCI - PCI桥了，所以PCI-

PCI桥2的下属总线号设置为2，同指定给PCI-PCI桥2的从总线号相同。

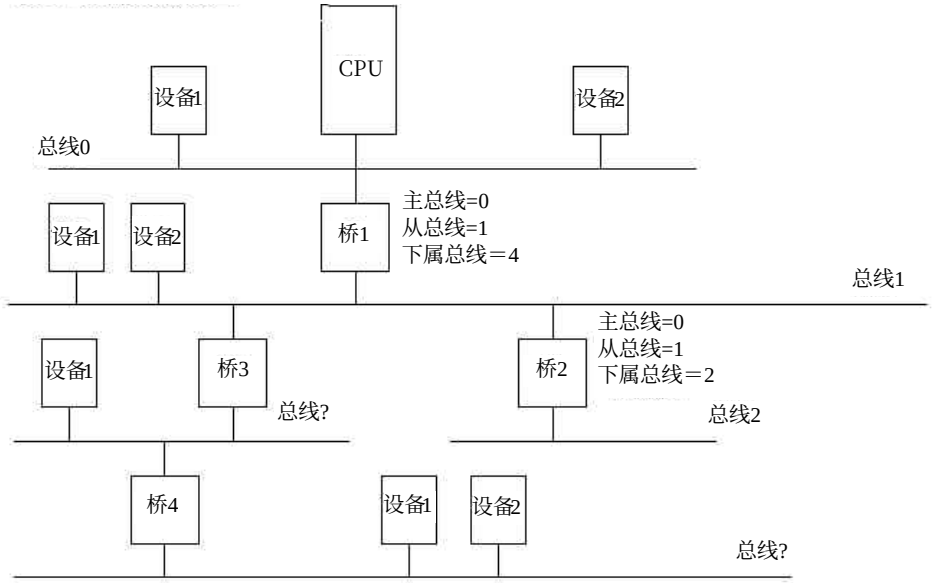

图12-6 PCI-PCI设置示意图二

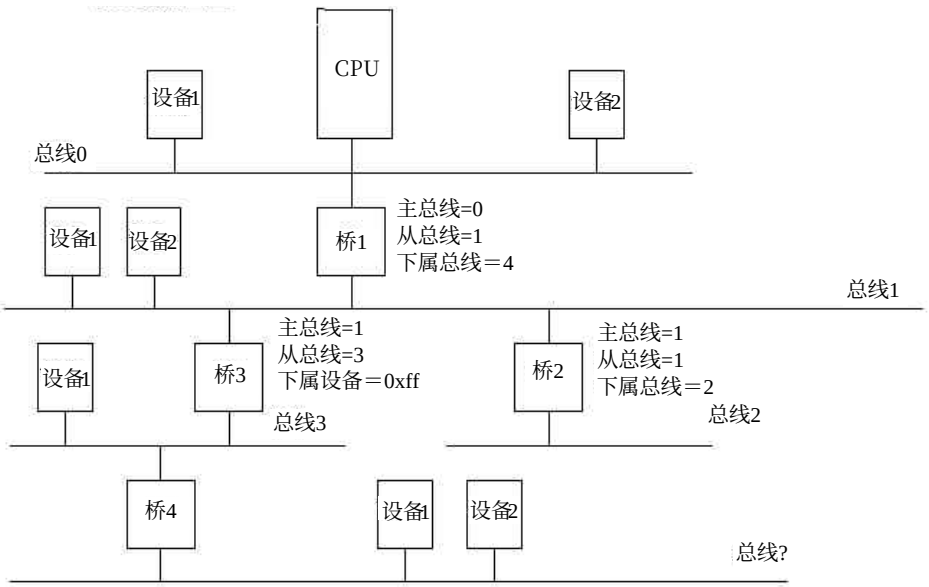

图12-7 PCI-PCI设置示意图三

第三步

PCI初始化程序接着返回重新去搜索PCI总线1，这样找到了另一个PCI-PCI桥3，如图12-7所示。初始化程序将它的主总线号指定为 1，从总线号指定为 3，同时指定下属的总线号为0xFF。现在，类型1设置环路中带有总线号1、2、3的都将可以正确地传送到相应的PCI总线。

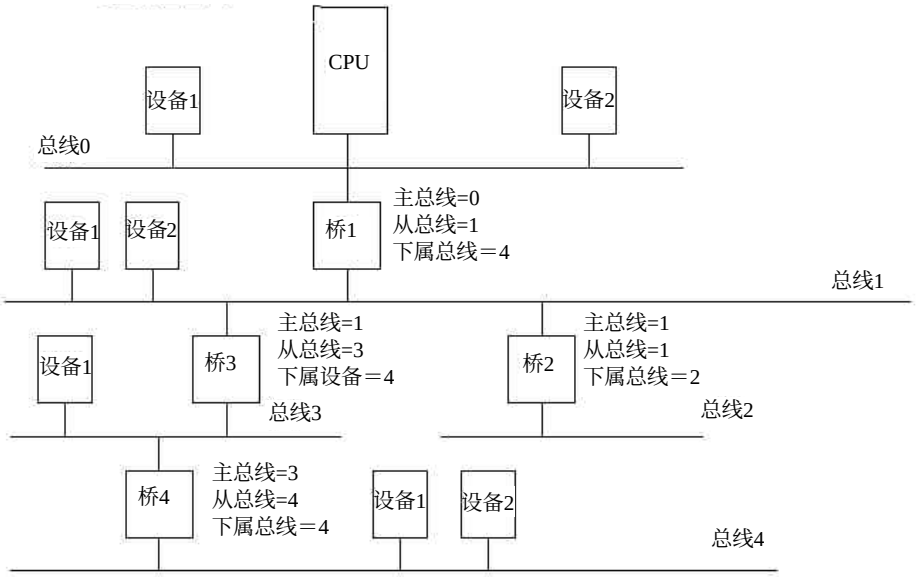

图12-8 PCI-PCI设置示意图四

第四步：

Linux系统的PCI初始化程序开始搜索PCI总线3，以及PCI-PCI桥3的下游。如图12-8所示，PCI总线3上有一个PCI-PCI桥4，所以初始化程序指定它的主总线号为3，从总线号为4。因为它是此分支上的最后一个总线桥，所以初始化程序指定它的从属总线号为 4。这样此系统中的所有PCI总线就都指定了总线号。

12.7.3 PCI BIOS 函数

PCI BIOS函数是一系列标准的跨平台子程序。它允许CPU控制对所有PCI地址空间的存取。只有Linux内核代码和设备驱动程序可以使用PCI BIOS函数。

12.7.4 PCI Fixup

对于基于Intel的系统来说，系统的BIOS已经在启动时比较全面地设置了 PCI系统。这样，Linux 系统就只需将这些设置映射到系统中了。

第13章　中断和中断处理

本章介绍Linux系统内核处理中断的方式。

13.1　中断

Linux系统中有很多不同的硬件设备。你可以同步使用这些设备，也就是说你可以发出一个请求，然后等待一直到设备完成操作以后再进行其他的工作。但这种方法的效率却非常的低，因为操作系统要花费很多的等待时间。一个更为有效的方法是发出请求以后，操作系统继续其他的工作，等设备完成操作以后，给操作系统发送一个中断，操作系统再继续处理和此设备有关的操作。

在将多个设备的中断信号送往CPU的中断插脚之前，系统经常使用中断控制器来综合多个设备的中断。这样即可以节约CPU的中断插脚，也可以提高系统设计的灵活性。中断控制器用来控制系统的中断，它包括屏蔽和状态寄存器。设置屏蔽寄存器的各个位可以允许或屏蔽某一个中断，状态寄存器则用来返回系统中正在使用的中断。

大多数处理器处理中断的过程都相同。当一个设备发出中段请求时， CPU停止正在执行的指令，转而跳到包括中断处理代码或者包括指向中断处理代码的转移指令所在的内存区域。这些代码一般在CPU的中断方式下运行。在此方式下，将不会再有中断发生。但有些 CPU的中断有自己的优先权，这样，更高优先权的中断则可以发生。这意味着第一级的中断处理程序必须拥有自己的堆栈，以便在处理更高级别的中断前保存CPU的执行状态。

当中断处理完毕以后， CPU将恢复到以前的状态，继续执行中断处理前正在执行的指令。中断处理程序十分简单有效，这样，操作系统就不会花太长的时间屏蔽其他的中断。

13.2　可编程中断控制器

系统设计者可以随意选择中断结构，但IBM PC使用Intel 82C59A-2 CMOS可编程控制器。

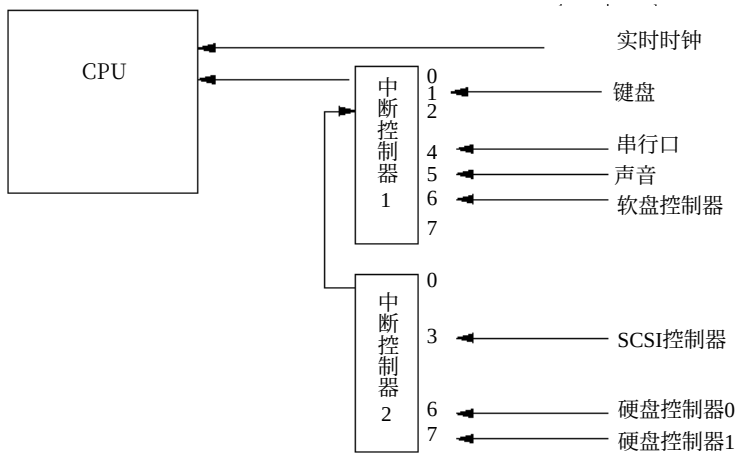

图13-1　中断控制示意图

此控制器的寄存器在ISA系统的内存空间中占有固定位置。

图13-1显示有两个8位的中断控制器级联在一起，每一个都有自己的屏蔽寄存器和中断状态寄存器。屏蔽寄存器的地址是0x21 和 0xA1，而状态寄存器的地址是0x20 和 0xA0。往屏蔽寄存器中的某一位写入1，则允许相应的中断；如果写入0，则会禁止相应的中断。所以，向屏蔽寄存器中的第三位写入1，将允许中断3；写入0，则将禁止中断3。但不幸的是，屏蔽寄存器是只写的，你无法读取该寄存器。这样，Linux系统自己必须保留一份写入到寄存器中的内容。

当发生中断信号时，中断处理程序读取两个中断状态寄存器（ ISR）的值。它将位于0x20的ISR放入一个16位的中断寄存器的低8位，同时将位于0xA0的ISR放入高8位。PIC1中的第二位用于中断控制器的级连，所以任何来自PIC2的中断都将导致PIC1中的第二位置1。

13.3 初始化中断处理的数据结构

系统内核有关中断处理的数据结构是在设备驱动程序要求对系统中断控制时创建起来的。在此过程中，设备驱动程序使用了一系列用于请求中断、允许中断以及禁止中断的 Linux系统内核服务。每一个设备驱动程序都将调用这些子过程来登记它们的中断处理过程的地址。

在PC机中，有些中断是固定的，所以当初始化时，驱动程序只需简单地请求中断即可。系统中的软盘驱动程序就是这样，它请求中断 6。但有时一个设备驱动程序不知道设备将使用哪一个中断。在 PCI结构中这不会成为一个问题，因为 PCI的设备驱动程序总是知道它们的中断号。但对于ISA结构而言，一个设备驱动程序找到自己使用的中断号却并不容易。 Linux系统通过允许设备驱动程序探测自己的中断来解决这个问题。

首先，设备驱动程序使得设备产生一个中断。然后，允许系统中所有没有指定的中断，这意味着设备挂起的中断将会通过中断控制器传送。 Linux系统读取中断状态寄存器然后将它的值返回到设备驱动程序。一个非 0的结果意味着在探测期间发生了一个或者多个的中断。设备驱动程序现在可以关闭探测，这时所有还未被指定的中断将继续被禁止。

一个ISA 设备驱动程序知道了它的中断号以后，就可以请求对中断的控制了。

PCI 结构的系统中断比ISA结构的系统中断要灵活得多。ISA设备使用中断插脚经常使用跳线设置，所以在设备驱动程序中是固定的。但 PCI设备是在系统启动过程中PCI初始化时由PCI BIOS 或 PCI 子系统分配的。每一个PCI设备都有可能使用A、B、C 或者 D这4个中断插脚中的一个。缺省情况下设备使用插脚A。

每个PCI插槽的PCI中断A、B、C 和 D是通过路由选择连接到中断控制器上的。所以PCI插槽4的插脚A可能连接到中断控制器的插脚 6，PCI插槽4的插脚B可能连接到中断控制器的插脚7,以此类推。

PCI中断具体如何进行路由一般依照系统的不同而不同，但系统中一定存在 PCI中断路由拓扑结构的设置代码。在Intel PC机中，系统的BIOS代码负责中断的路由设置。对于没有BIOS的系统，Linux系统内核负责设置。

PCI的设置代码将中断控制器的插脚号写入到每个设备的PCI 设置头中。PCI的设置代码根据所知道的PCI中断路由拓扑结构、 PCI设备使用的插槽，以及正在使用的PCI中断的插脚号来决定中断号，也就是IRQ号。

系统中可以有很多的PCI中断源，例如当系统使用了PCI-PCI桥时。这时，中断源的数目可能超过了系统可编程中断控制器上插脚的数目。在这种情况下，某些 PCI设备之间就不得不共享一个中断，也就是说，中断控制器上的某一个插脚可以接收来自几个设备的中断。 Linux系统通过让第一个中断源请求者宣布它使用的中断是否可以被共享来实现中断在几个设备之间共

享的。中断共享使得irq_action数组中的同一个入口指向几个设备的irqaction结构。当一个共享的中断有中断发生时，Linux系统将会调用和此中断有关的所有中断处理程序。所有可以共享中断的设备驱动程序的中断处理程序都可能在任何时候被调用，即使在自身没有中断需要处理时。

13.4 中断处理

Linux系统处理中断所需要的数据结构参见图13-2。

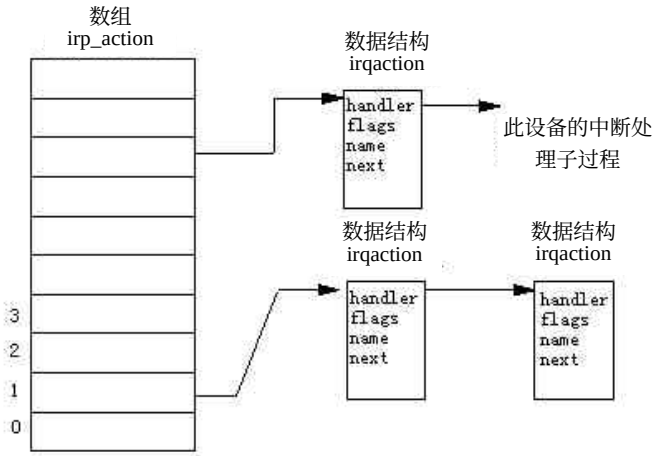

图13-2 中断处理数据结构示意图

Linux系统中断处理程序的一个主要任务就是将中断定位到中断处理程序适当部分的代码上，此代码必须了解中断的拓扑结构。例如，如果软盘控制器的中断在中断控制器的插脚为 6，那么系统的中断处理程序必须能够识别此中断是来自软盘控制器，并且可以将中断定位到软盘驱动程序的中断处理代码中。Linux系统使用一个指针数组指向包括系统中断处理程序地址的数据结构。这些处理程序属于设备驱动程序的一部分，并且每个设备驱动程序自己负责在初始化时请求中断。图13-2显示irq_action是指向irqaction结构的指针的数组。每一个irqaction结构都包含有关此中断处理程序的信息，包括中断处理程序的地址。由于各个系统之间的中断数目和具体如何处理中断并不完全相同，所以Linux系统的中断处理代码是随系统而异的。

当中断发生时，Linux系统首先通过读取系统中可编程中断控制器的中断状态寄存器来确定中断的来源。然后将此来源转换成irq_action数组中的位移。例如，来自软盘的中断控制器插脚6的中断将会转换成irq_action数组中第七个指针。如果发生的中断不存在中断处理程序，那么Linux系统内核将记录一个错误。否则它将调用此中断来源的所有 irqaction结构所指向的中断处理程序。

当Linux系统内核调用设备驱动程序的中断处理代码时，它必须能够迅速找出中断的原因和处理办法。设备驱动程序可以通过读取被中断的设备的状态寄存器的值来确定中断的原因。一旦找出中断的原因，设备驱动程序可能需要做更多的工作。如果是这样，Linux系统内核可以将工作推迟到以后再完成。这样就避免了CPU在中断方式下花费太多的时间。

第14章　设备驱动程序

操作系统的一个功能就是将用户和系统的硬件特性隔离。例如，虚拟文件系统（ VFS）使得系统上挂接的文件系统具有相同的接口，使用户不必关心底层的硬件设备。本章介绍 Linux 系统内核是如何管理系统中的硬件设备的。

14.1　硬件设备的管理

计算机系统中的物理设备都有自己的硬件控制器。例如键盘、鼠标和串行口是由 SuperIO 芯片控制的；IDE硬盘是由IDE控制器控制的，等等。每一个硬件控制器都有自己的控制及状态寄存器（CSR）,而且随设备的不同而不同。CSR用来启动和停止设备、初始化设备和诊断设备错误。设备驱动程序一般集成在操作系统内核，这样不同的应用程序就可以共享这些代码。Linux系统内核中的设备驱动程序在本质上是一些特殊的，常驻内存的低级硬件处理程序的共享库。正是Linux系统的设备驱动程序处理系统中的硬件设备的具体细节。

设备驱动程序的一个基本特点就是对设备的抽象处理。系统中的所有硬件设备看起来都和一般的文件一样，它们可以使用处理文件的标准系统调用来打开、关闭和读写。系统中的每一个设备都由一个设备文件来代表，例如，主 IDE硬盘的设备文件是 /dev/had。对于块设备和字符设备来说，这些设备文件可以使用 mknod命令创建。新建的设备文件使用主设备号和从设备号来描述此设备。网络设备的设备文件是当系统查找到网络设备并初始化网络控制器之后才建立的。一个设备驱动程序控制的所有设备有一个相同的主设备号，通过不同的从设备号来区分设备和它们的控制器。例如，主 IDE硬盘的每一个分区都有一个不同的从设备号，这样主 IDE 硬盘的第二个分区的设备文件是 /dev/hda2。Linux系统使用主设备号和系统中的一些表来将系统调用中使用的设备文件映射到设备驱动程序中。

Linux 系统支持三种类型的硬件设备：字符设备、块设备和网络设备。字符设备是直接读取的，不必使用缓冲区。例如，系统的串行口 /dev/cua0 和 /dev/cua1。块设备每次只能读取一定大小的块的倍数，通常一块是 512或者1024字节。块设备通过缓冲区读写，并且可以随机地读写。块设备可以通过它们的设备文件存取，但通常是通过文件系统存取。只有块设备支持挂接的文件系统。网络设备是通过BSD 套接字界面存取的。

Linux系统支持多种设备，这些设备的驱动程序之间有一些共同的特点：

- 内核代码：设备驱动程序是系统内核的一部分，所以如果驱动程序出现错误的话，将可能严重地破坏整个系统。
- 内核接口：设备驱动程序必须为系统内核或者它们的子系统提供一个标准的接口。例如，一个终端驱动程序必须为 Linux内核提供一个文件I/O接口；一个SCSI设备驱动程序应该为SCSI子系统提供一个SCSI设备接口，同时SCSI子系统也应为系统内核提供文件 I/O和缓冲区。
- 内核机制和服务：设备驱动程序利用一些标准的内核服务，例如内存分配等。
- 可装入：大多数的Linux设备驱动程序都可以在需要时装入内核，在不需要时卸载。
- 可设置：Linux系统设备驱动程序可以集成为系统内核的一部分，至于哪一部分需要集成到内核中，可以在系统编译时设置。

- 动态性：当系统启动并且各个设备驱动程序初始化以后，驱动程序将维护其控制的设备。如果设备驱动程序控制的设备并不存在，也并不妨碍系统的运行。在这种情况下，设备的驱动程序只是多占用了一点系统的内存。

14.2 轮询和中断

每当设备发出一个命令，例如“移动读写头到软盘的第 42 扇区”，设备驱动程序可以选择如何确定命令是否完成。设备驱动程序即可以选择轮询设备，也可以选择使用中断。

轮询设备是指每隔一定的时间驱动程序就读取一次设备的状态寄存器，直到状态寄存器的值有所改变为止。但当设备驱动程序是系统内核的一部分时，轮询将会使系统的效率变得极低，因为在设备的请求完成之前，系统将无法进行任何其他的工作。使用轮询的设备驱动程序通过系统计时器来使系统内核在稍后的时间调用设备驱动程序的一个子过程。 Linux系统中的软盘就是这样工作的。比此方法更好的方法是使用中断。

中断驱动的设备驱动程序是指每当硬件设备需要服务时，它就会产生一个硬件中断。例如，每当一个以太网设备从网络中接收一个以太网数据包时会产生一个中断。 Linux系统内核能够将中断从产生中断的设备传递到相应的设备驱动程序中，这可以通过设备驱动程序在系统内核中登记中断的使用来实现。设备驱动程序将中断处理过程的地址和设备希望使用的中断号登记在内核中。通过/proc/interrupts可以知道设备驱动程序使用的是哪一个中断，以及每种类型的中断有几个：

```
0:     727432    计时器
1:      20534    键盘
2:          0    级联（cascade）
3:      79691 +  串口
4:      28258 +  串口
5:          1    声霸卡
11:     20868 +  aic7xxx
13:         1    数学错误
14:       247 +  ide0
15:       170 +  ide1
```

这种中断源的请求是在驱动程序初始化时完成的。系统中的一些中断是固定的，例如，软盘控制器使用中断6。其他有些中断，例如来自PCI设备的中断是在系统启动时动态分配的。在这种情况下，设备驱动程序必须在要求中断的所有权之前首先知道设备的中断号。对于 PCI中断，Linux系统支持标准的PCI BIOS反馈来确定设备在系统中的信息，包括设备的中断号。

14.3 直接内存存取

当交换的数据量很小时，中断驱动的设备驱动程序可以通过使用中断来和设备之间交换数据。但如果是高速设备，例如硬盘或者以太网设备，那么中断方式就会占用过多的系统内存。

解决此问题的方法是使用直接内存存取，也就是 DMA。一个DMA控制器允许在设备没有处理器的干预下和系统内存直接交换数据。一个 PC机的 ISA DMA控制器共有8个DMA通道，其中7个可以用于设备驱动程序。每一个DMA通道都包括一个16位的地址寄存器和一个16位的记数寄存器。在开始传输数据之前，设备驱动程序需要设置 DMA通道的地址和记数寄存器，以及数据传输的方向 —— 是读数据还是写数据。在这之后，设备就可以随时开始 DMA传输了。

当传输结束后，设备将会中断系统的CPU，但在传输过程中，CPU可以做其他工作。

设备驱动程序在使用DMA时应该小心。首先， DMA控制器不知道虚拟内存的存在，它只能存取系统的物理内存。这样 DMA使用的系统内存必须是连续的物理内存块。这意味着你不能直接使用DMA传输数据到一个进程的虚拟内存地址空间。但你可以将一个进程的物理页面锁定到内存中，这样在DMA操作期间进程的物理页面就不会被交换出系统的物理内存。第二，DMA控制器无法存取整个的物理内存。 DMA通道的地址寄存器存储的是DMA地址的后16位，而前8位则来自页面寄存器。这表明DMA请求只能使用系统最开始的16M地址。

DMA通道是稀少资源，它们只有7个，并且设备驱动器之间不能共享DMA通道。就像使用中断一样，设备驱程序也必须了解它使用的是哪一个 DMA通道。系统中一些设备有固定的DMA通道，例如，软盘使用的是 DMA通道2。有时一个设备的DMA通道可以通过跳线设置。更为灵活的设备可以通过它们的 CSR了解使用的DMA通道，这样，设备驱动程序可以随便地选择一个空闲的DMA通道。

Linux系统使用数据结构dma_chan来了解系统中DMA通道的使用情况。dma_chan结构只有两个字段，一个是指向描述 DMA通道使用者的字符串的指针，另一个用来标识DMA通道是否已被分配。你在系统中使用cat/proc/dma命令时，显示的正是dma_chan结构的数组。

14.4 内存

设备驱动程序在使用内存时也应十分小心。因为设备驱动程序是系统内核的一部分，所以它也无法使用虚拟内存。每当一个设备驱动程序运行时，当前进程都有可能改变。设备驱动程序不能依靠一个特殊的进程来运行，即使设备驱动程序代表这个进程。象系统内核的其他部分一样，设备驱动程序使用数据结构来保持和它所控制的设备的联系。这些数据结构可以作为设备驱动程序的一部分来静态地分配，但这样可能会使得系统的内核变大。大多数设备驱动程序使用内核中没有分页的内存来存储它们的数据。

系统的设备驱动程序使用Linux系统提供的内核内存分配和撤消程序。内核中的内存是以 2的乘方为单位分配的，例如， 128或者512字节。

在请求内核内存时， Linux系统可能需要做很多的额外工作。例如，如果系统中的空闲的物理内存太少，被占用的物理内存可能被放弃或者交换到系统的交换文件中。一般情况下，Linux系统将会挂起设备驱动程序的请求，把进程送入到一个等待队列中直到有足够的内存以后再继续运行。但并不是系统中所有的设备驱动程序都希望以这种方式运行。所以当系统不能立即为设备驱动程序分配足够的内存时，内核的内存分配程序将会返回一个错误。如果设备驱动程序希望使用DMA和被分配的内存之间交换数据，那么它可以将内存指定为可以直接存取的。这样，只有Linux系统内核，而不是设备驱动程序需要知道DMA是如何构成的。

14.5 设备驱动程序和内核之间的接口

Linux系统和设备驱动程序之间使用标准的交互接口。无论是字符设备、块设备还是网络设备的设备驱动程序，当系统内核请求它们的服务时，都使用同样的接口。这样， Linux系统内核可以用同样的方法来使用完全不同的各种设备。

Linux系统是一个动态的操作系统，每当Linux系统内核启动时，它都有可能检测到不同的物理设备，这样就可能需要不同的驱动程序。 Linux允许你在构建系统内核时使用设置脚本将设备驱动程序包括在系统内核中。这些驱动程序在系统启动时初始化，它们可能找不到所控制的设备。其他的设备驱动程序可以在需要时作为内核模块装入到系统内核中。为了适应设备驱

动程序的这种动态的特性，设备驱动程序在其初始化时就在系统内核中进行登记。 Linux系统使用设备驱动程序的登记表作为内核和驱动程序接口的一部分。这些表中包括指向处理程序的指针和其他信息。

14.5.1 字符设备

存取系统中的字符设备和存取系统文件一样。应用程序使用标准的系统调用来打开、读写和关闭字符设备，就像使用一个文件一样，即使是 PPP使用的调制解调器也是一样。当字符设备初始化时，设备驱动程序通过在由数据结构 device_struct组成的chrdevs数组中添加一个入口来向系统内核注册。设备的主设备号用来作为此 chrdevs的索引。一个设备的主设备号是固定的。

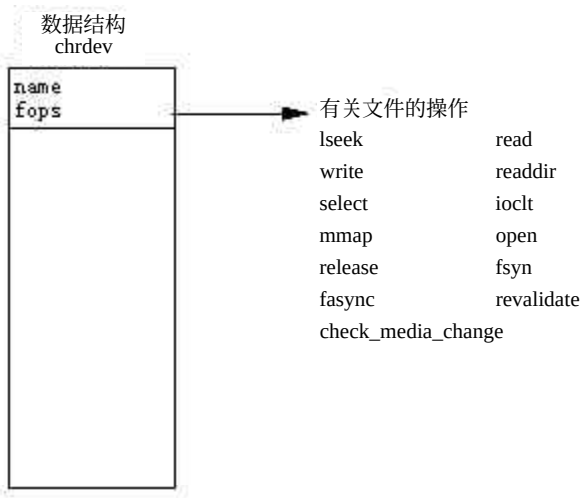

图14-1 字符设备驱动程序示意图

数组chrdevs中的每一个入口都是一个device_struct结构。如图14-1所示，device_struct结构包括两个指针元素：一个指向登记的设备驱动程序名，另一个指向一个包括各种文件操作过程的地址的数组。此数组中包括的地址指向设备驱动程序中处理文件的操作，例如，打开、读写和关闭子过程。字符设备的/proc/devices中内容就来自chrdevs数组。

当代表一个字符设备的字符设备文件打开以后，系统必须知道如何正确地调用相应字符设备的文件操作过程。和一般的文件或者目录一样，每一个字符设备文件都由 VFS索引节点来表示，VFS索引节点包括设备的主标识符和从标识符。 VFS索引节点是在文件系统检测到设备文件名时，由文件系统创建的。

每一个VFS索引节点都和一系列的文件操作相连，并且这些文件操作随索引节点代表的文件的不同而不同。每当一个VFS索引节点所代表的字符设备文件创建时，它的有关文件的操作就设置为缺省的字符设备操作。缺省的文件操作只包含一个打开文件的操作。当应用程序打开一个字符设备文件时，通用的文件操作使用设备的主标识符作为 chrdevs数组的索引，依此可以找到有关此设备的各种文件操作。它还将建立起描述此字符设备文件的文件数据结构，使得其中的文件操作指针指向此设备驱动程序中的有关文件的操作。这样，应用程序中的文件操作将会映射到字符设备的文件操作调用中。

14.5.2 块设备

对于块设备的存取和对于文件的存取一样。它的机制和字符设备使用的机制相同。 Linux系统中有一个blkdevs数组，它描述了一系列在系统中注册的块设备。数组 blkdevs也使用设备的主设备号作为索引。它的每一个入口均由数据结构 device_struct组成。和字符设备不一样的是，块设备有几种类型，例如 SCSI设备和IDE设备。每一类的块设备都在 Linux系统内核中注册，并向内核提供自己的文件操作。例如，一个 SCSI 设备驱动程序为SCSI子系统提供接口，反过来SCSI子系统使用这些接口为系统内核提供有关此设备的文件操作。

每一个块设备驱动程序必须既向缓冲区提供接口，也提供一般的文件操作接口。每一个块设备都在blk_dev数组中有一个blk_dev_struct结构的入口。数据结构blk_dev_struct 包括一个请求过程的地址和一个指向请求数据结构链表的指针，每一个请求数据结构都代表一个来自缓冲区的请求。

每当缓冲区希望和一个在系统中注册的块设备交换数据，它都会在 blk_dev_struct中添加一个请求数据结构。如图14-2所示，每一个请求都有一个指针指向一个或者多个 buffer_head数据结构，每一个buffer_head结构都是一个读写数据块的请求。每一个请求结构都在一个静态链表all_requests中。如果请求添加到了一个空的请求链表中，则调用设备驱动程序的请求函数来开始处理请求队列。否则，设备驱动程序只是简单地处理请求队列中的每一个请求。

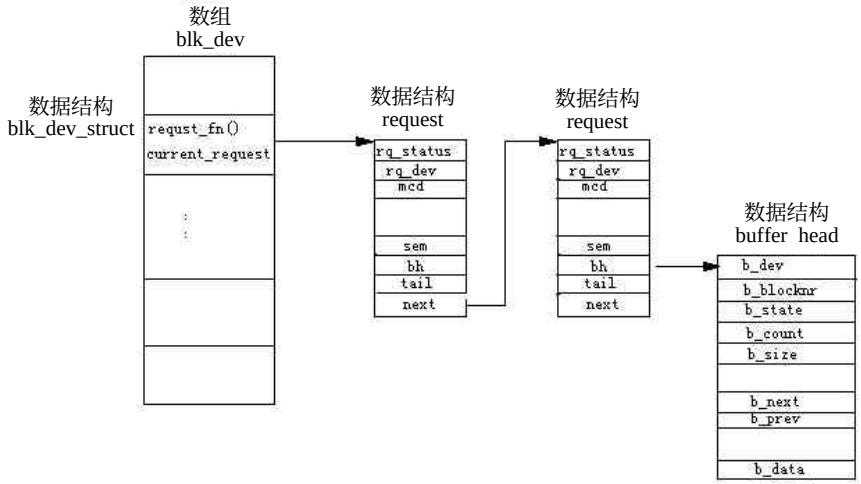

图14-2 块设备驱动程序数据结构示意图

一旦设备驱动程序完成了一个请求，它将把 buffer_head 结构从request结构中移走，并把buffer_head结构标记为已更新，同时将它解锁。这样就可以唤醒等待锁定操作完成的进程。

14.6 硬盘

硬盘驱动器提供了一个永久性存储数据的方法。硬盘一般使用它的柱面号、磁头号和扇区号来描述。例如，启动时Linux系统可能这样描述一个IDE硬盘：

hdb: Conner Peripherals 540MB - CFS540A, 516MB w/64kB Cache, CHS=1050/16/63

这说明此硬盘一共有1050个柱面，16个磁头以及每个磁道上有63个扇区。

硬盘还可以进一步地分区。很多的 Linux系统的硬盘都包括三个分区：一个是 DOS文件系统分区，一个是EXT2文件系统分区，第三个是交换分区。硬盘使用硬盘分区表描述分区，分

区表的每一个入口都包括以磁头、扇区和柱面表示的分区起始位置。

以下是包括两个主分区的硬盘：

Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders

Units = cylinders of 2048 * 512 bytes

Device Boot	Begin	Start	End	Blocks	Id	System
/dev/sda1	1	1	478	489456	83	Linux native
/dev/sda2	479	479	510	32768	82	Linux swap

Expert command (m for help): p

Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders

Nr	AF	Hd	Sec	Cyl	Hd	Sec	Cyl	Start	Size	ID
1	00	1	1	0	63	32	477	32	978912	83
2	00	0	1	478	63	32	509	978944	65536	82
3	00	0	0	0	0	0	0	0	0 00	
4	00	0	0	0	0	0	0	0	00	

这表明硬盘的第一个分区开始于柱面0，磁头1和扇区1，一直到柱面477，磁头63和扇区32。第二个分区从柱面478开始一直到最里面的柱面为止。

系统初始化时，Linux系统将确定硬盘的数目和类型。另外，Linux系统也检测硬盘是如何分区的。这些信息都保存在gendisk_head指针指向的由数据结构gendisk组成的链表中。每一个硬盘子系统，例如IDE硬盘系统，初始化时将会产生代表此硬盘的数据结构 gendisk，并且同时会注册有关此硬盘的文件操作并把它添加到数据结构 blk_dev中。每一个gendisk结构中都包括一个唯一的主设备号，并且和设备文件中的主设备号匹配。例如，SCSI硬盘子系统创建一个gendisk结构，其入口是sd，主设备号是8。图14-3显示了两个gendisk结构，一个是SCSI硬盘系统的gendisk结构，另一个是IDE硬盘的gendisk结构。

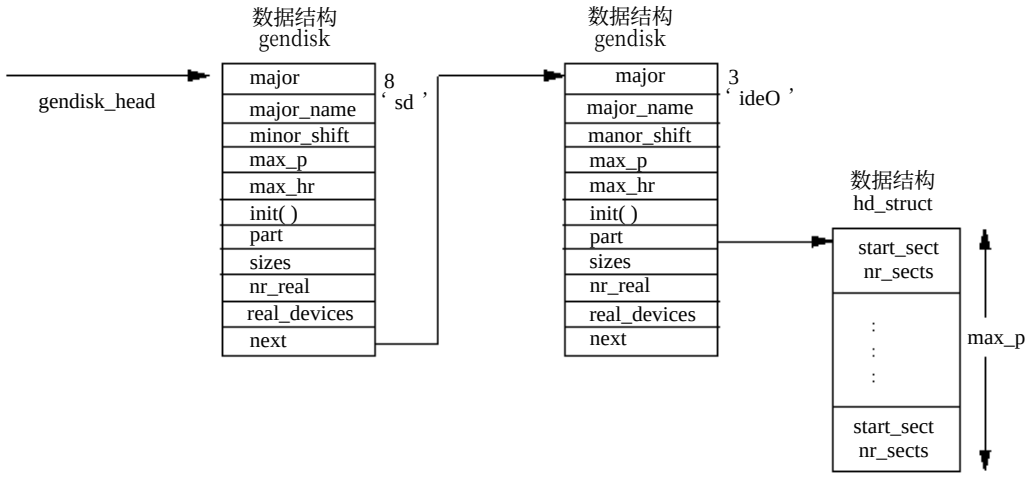

图14-3 硬盘链表示意图

虽然是硬盘子系统在初始化时创建的 gendisk结构，但只有当Linux系统进行分区检测时才会用到gendisk结构。每一个硬盘子系统都有一个自己的数据结构，允许它将设备的主设备号和从设备号映射到物理硬盘的各个分区。每当从硬盘中读取一个数据块时，无论是通过缓冲区

还是通过有关的文件操作，系统内核都将使用设备文件（例如 /dev/sda2）中的主设备号找到相应的设备，然后由设备驱动程序或者设备子系统将从设备号映射到真正的物理设备中。

14.6.1 IDE 硬盘

Linux系统中最为常用的硬盘就是IDE（Integrated Disk Electronic）硬盘。一个IDE硬盘控制器最多可以支持两个硬盘，一个是主硬盘，另一个是从硬盘。而系统中的第一个硬盘控制器叫做主硬盘控制器，第二个硬盘控制器叫做从硬盘控制器。 Linux根据系统检测到硬盘的顺序来命名硬盘。这样，主硬盘控制器上的主硬盘是 /dev/hda，而主硬盘控制器上的从硬盘是/dev/hdb。/dev/hdc则是从硬盘控制器上的主硬盘。 IDE硬盘子系统将硬盘控制器而不是硬盘登记到Linux系统中。系统中主硬盘控制器的主标识符号是 3，从硬盘控制器的主标识符是 22。这意味着如果一个系统中有两个IDE硬盘控制器，那么在blk_dev 和 blkdevs数组中的入口分别是3和22。 在硬盘的设备文件中，硬盘 /dev/hda 和 /dev/hdb都代表主硬盘控制器，它们的主标识符号都是3。任何对这两个设备文件的有关文件和缓冲区进行的操作都将转向到同一个 IDE控制器，因为系统内核使用主标识符号作为索引。当有请求产生时，由 IDE控制器使用设备标识符中的从设备号来辨别是哪一个硬盘的请求。例如，主硬盘控制器的从硬盘 /dev/hdb的设备标识符为（3，64），而此硬盘的第一个分区/dev/hdb1的设备标识符为（3，65）。

14.6.2 初始化IDE 硬盘子系统

IDE硬盘的发展已经有很长的时间了。在这期间， IDE硬盘的接口有了很大的变化，这就使得IDE硬盘的初始化变得比较复杂。

Linux系统最多支持4个IDE硬盘控制器。每一个硬盘控制器都对应数组 ide_hwifs中的数据结构ide_hwif_t，ide_hwif_t结构中包括两个数据结构 ide_drive_t，每一个代表一个主硬盘驱动器或一个从硬盘驱动器。在硬盘系统的初始化过程中， Linux系统首先查看系统CMOS中有关硬盘的信息。Linux系统使用CMOS中的硬盘信息来创建ide_hwif_t结构，以反映系统中的硬盘控制器和所连接的硬盘的信息。在硬盘操作过程中， IDE硬盘驱动程序把命令写入到系统I/O内存空间中的IDE命令寄存器中。缺省的主硬盘控制器的控制和状态寄存器的 I/O地址是0x1F0 - 0x1F7。IDE硬盘驱动程序记录每一个硬盘控制器的 Linux系统缓冲区和VFS索引节点，并把这些信息添加到blkdevs数组中的blk_dev结构内。IDE硬盘也需要适当的中断号。一般情况下，主硬盘控制器的中断号是14，从硬盘控制器的中断号是15，但这些数值可以使用内核中的命令行参数覆盖。IDE硬盘驱动程序在系统启动过程中，也会在 gendisk链表中添加gendisk入口。此链表在稍后会用来检测所有的硬盘分区表。

14.6.3 SCSI 硬盘

SCSI总线是一个十分有效的对等结构的数据总线，一条 SCSI总线可以支持多达8个设备。总线上的每一个设备都有一个唯一的标识符。总线上任何两个设备之间可以进行同步或者异步数据传输，传输的数据宽度是32位，每秒的传输速率可达40M。从源设备到目的设备的一次单独的传输交易可以分为 8个不同的阶段。你可以通过总线上的 5个信号来分辨SCSI总线当前所处的阶段。这8个阶段分别是：

- 总线空闲

没有设备在控制总线，而且当前总线上没有交易发生。

- 设备判优

如果一个SCSI设备希望获得SCSI总线的控制权，那么它将会把它的 SCSI标识符发送到地址插脚上。具有最大的SCSI标识符的设备获得总线的控制权。

• 选择设备

当一个设备获得总线的控制权以后，它将会把目地设备的 SCSI标识符发送到地址插脚中，以便通知目地设备它将发送一个命令。

• 重新选择设备

SCSI设备可以在一次请求的处理过程中中断此请求。这时目的设备可以重新选择源设备。不是所有的SCSI设备都支持这个阶段。

• 命令

可以从源设备向目的设备传送6、10 或者 12 个字节的命令。

• 数据输入和输出

在此阶段，数据在源设备和目的设备之间传送。

• 状态

所有的命令完成之后，它允许目的设备发送一个状态字节以通知源设备传送的成功或者失败。

• 信息的输入、输出

在此阶段传送一些额外的信息。

Linux系统中的SCSI硬盘子系统由两个基本元素构成，每一个都由一个数据结构来表示：

• 主机（host）

一个SCSI主机就是一个SCSI控制器，例如NCR810 PCI SCSI控制器就是一个SCSI主机。如果Linux系统中包括多于一个的同类型的SCSI控制器，则每一个都是单独的SCSI主机。这意味着一个SCSI设备驱动程序可以控制多于一个的控制器。 SCSI主机一般是SCSI命令的源设备。

• 设备（Device）

最常见的SCSI设备是SCSI硬盘，但SCSI也支持其他类型的设备：磁带， CD-ROM以及其他的SCSI设备。SCSI设备一般就是SCSI命令的目的。但不同的设备必须区别对待，例如，对于像SCSI磁带和CD-ROM等设备， Linux系统必须检测其中的读写介质是否已经被移动。不同的SCSI磁盘类型有不同的主设备号,这样Linux 系统就可以直接找到相应的SCSI类型的设备。

14.6.4 初始化 SCSI 磁盘子系统

因为SCSI总线和SCSI设备的动态特性，初始化一个 SCSI子系统相对来说特别地麻烦。Linux系统在系统启动时对SCSI子系统进行初始化。系统首先查找SCSI 控制器（也就是SCSI主机），然后逐个检测SCSI总线以便搜寻所有的SCSI设备。最后系统初始化这些设备，使得系统内核的其他部分可以通过一般的文件形式和缓冲区设备操作来使用 SCSI设备。SCSI设备的初始化可以分为4个阶段：

• 首先， Linux系统查找在Linux系统构建内核时集成到内核中的 SCSI控制器。如图14-4所示，每一个SCSI控制器都有一个包括在builtin_scsi_hosts数组中的Scsi_Host_Template结构的入口。 Scsi_Host_Template结构包括一些指向SCSI过程的指针，这些过程用来实现SCSI主机的动作，例如检测SCSI主机都带有哪些SCSI设备。SCSI子系统在设置时要调用这些过程，并且这些过程是此种SCSI主机的驱动程序的一部分。每一个带有 SCSI设备的SCSI主机的Scsi_Host_Template数据结构组成了一个scsi_hosts的链表，链表中的每一个Scsi_Host数据结构都代表一个SCSI主机类型。例如，一个系统中包括两个 NCR810 PCI

SCSI控制器，则每一个控制器都有一个Scsi_Host结构，每一个Scsi_Host结构都指向代表设备驱动程序的Scsi_Host_Template。

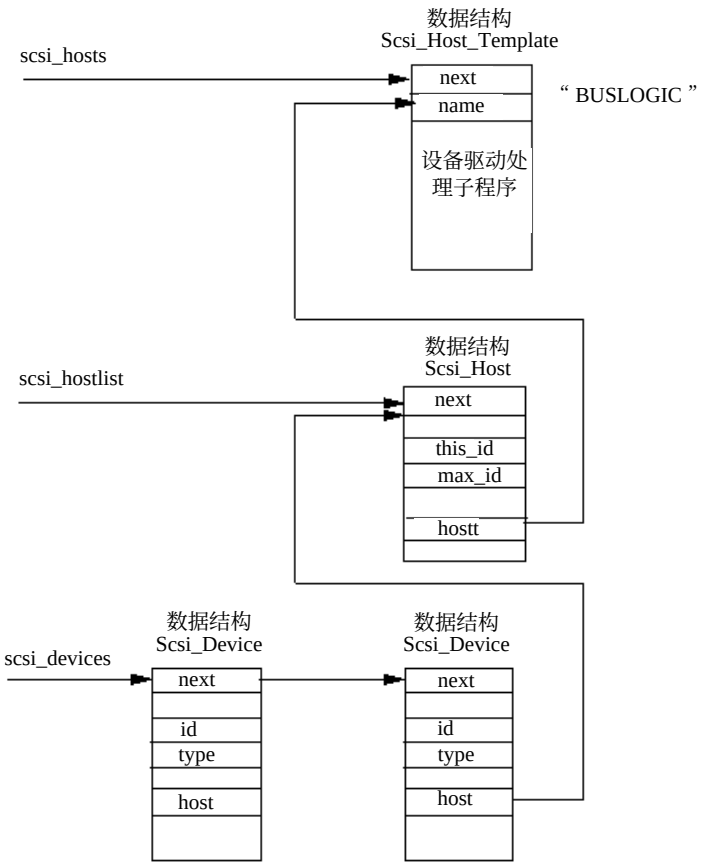

图14-4 SCSI 设备示意图

- 系统在检测到SCSI主机以后，SCSI子系统必须进一步确认每一个SCSI主机总线上所带的SCSI设备。SCSI设备的标号从0到7，而且每个设备的设备号或者标识符在设备所在的总线上都是唯一的。SCSI初始化程序使用TEST_UNIT_READY命令来检测SCSI总线上的SCSI设备。当某一个设备有回应后，初始化程序再使用 ENQUIRY命令读取设备的标识符，标识符中包含厂商名称、设备型号以及版本号。 Scsi_Cmnd数据结构代表所有的SCSI命令，而这些命令通过调用在Scsi_Host_Template结构中的设备驱动程序子过程来传递给此SCSI主机的设备驱动程序。系统中的每一个SCSI设备都有一个Scsi_Device数据结构，每一个Scsi_Device数据结构都指向Scsi_Host。所有的Scsi_Device结构组成了一个scsi_devices 链表。
- SCSI设备共有4种：磁盘，磁带， CD-ROM如通用设备。每一种类型的SCSI设备都使用不同的主设备号在系统中单独注册。每一种SCSI设备，例如SCSI 磁盘，都有自己的设备表，它使用设备表和系统中相应的设备驱动程序以及 CSI主机进行通信。每一种SCSI 类型的Scsi_Device_Template结构都包括此种SCSI设备类型的信息和执行各种任务的子程序的地址。
- SCSI子系统初始化的最后阶段是调用每一个注册的 Scsi_Device_Template结构的结束

函数。

14.6.5 传递块设备请求

Linux 系统初始化SCSI子系统之后，就可以使用 SCSI设备了。每一个可以使用的 SCSI设备都在系统内核中注册，所以 Linux系统可以直接将块设备请求传递给相应的 SCSI设备。在Linux系统中可以通过blk_dev数据结构进行缓冲区请求，或者通过 blkdevs结构进行文件操作。例如，对于一个包含多个EXT2文件系统分区的SCSI磁盘驱动器，系统内核缓冲区请求是如何传递到相应的挂接一个EXT2文件系统分区的磁盘上的？每当产生一个从 SCSI磁盘分区中读写数据块的请求时，都会创建一个新的请求结构，并将其添加到 blk_dev数组的current_request链表中。如果请求链表正在处理，那么缓冲区则不需要做任何事情，否则它将通知 SCSI处理请求队列。

系统中的每一个SCSI硬盘都由一个Scsi_Disk数据结构来代表。Scsi_Disk数据结构保存在rscsi_disks数组中，并且使用SCSI硬盘分区的从设备号作为索引。例如， /dev/sdb1的主设备号是8，从设备号是17，那么它的索引是17。每一个Scsi_Disk数据结构都包括一个指向代表此设备的Scsi_Device结构的指针，Scsi_Device结构又指向拥有此设备的SCSI主机的Scsi_Host结构。来自缓冲区的请求数据结构将会被翻译成描述SCSI命令的Scsi_Cmd结构。

14.7 网络设备

对Linux系统而言，网络设备是一个收发数据包的整体。它经常是一个物理设备，例如以太网卡。一些网络设备可以是软件的，例如用来给你自己发送数据的回馈设备。每一个网络设备都用一个数据结构来代表它。 Linux系统启动时，随着网络的初始化，网络设备驱动程序将在Linux系统中登记它所控制的设备。设备数据结构中包括有关设备的信息和允许系统支持的各种网络协议所使用的设备服务的各种函数的地址。这些函数主要用于传输数据。设备使用标准的网络服务机制来把接受到的数据传输到相应的协议层。所有传输的网络数据包都使用一个sk_buff数据结构。此sk_buff数据结构十分灵活，它可以允许网络协议头方便地添加和移走。

下面介绍device结构中所包括的有关网络设备的信息。

14.7.1 网络设备文件名

网络设备文件是在系统的网络设备被系统检测到和初始化的同时创建的。它们的名字代表了设备的类型，同种类型的多个设备则从 0开始编号。例如，以太网卡的设备文件是 /dev/eth0、/dev/eth1、/dev/eth2、以此类推。一些常见的网络设备是：

/dev/ethN	Ethernet 设备
/dev/slN	SLIP 设备
/dev/pppN	PPP 设备
/dev/lo	Loopback 设备

14.7.2 总线信息

总线信息是设备驱动程序在控制设备时所使用的信息。 irq号是设备使用的中断号，基地址是设备的控制和状态寄存器在I/O内存中的地址， DMA通道是网络设备使用的DMA通道号。所有这些信息都是在系统启动时设备初始化的过程中设置的。

14.7.3 网络接口标记

网络接口标记用来描述网络设备的特点和功能：

IFF_UP	网络接口已经打开并且正在运行 。
IFF_BROADCAST	设备中的广播地址有效 。
IFF_DEBUG	打开设备调试。
IFF_LOOPBACK	此设备是一个回馈设备。
IFF_POINTTOPOINT	这是一个点到点的连接 (SLIP 和 PPP)。
IFF_NOTRAILERS	不存在网络尾部。
IFF_RUNNING	资源分配。
IFF_NOARP	不支持ARP 协议。
IFF_PROMISC	设备在混合状态，它将接收各个地址的数据包 。
IFF_ALLMULTI	接收所有的IP贞。
IFF_MULTICAST	可以接收IP贞。

14.7.4 协议信息

每个设备中都有网络协议层如何使用设备的信息：

• mtu

指出网络可以传输的数据包的最大值，但不包括数据包上添加的链路层的数据头。

• Family

指出设备可以支持的协议族。

• Type

硬件接口类型描述了网络设备可以连接的介质类型，包括 Ethernet、X.25、Token Ring、Slip、PPP 和 Apple Localtalk。

• Addresses

和网络设备有关的一系列地址，包括IP地址。

• 数据包队列

这是sk_buff的队列，里面是等待传输的数据包。

• 支持的函数

每一个设备都提供一个标准的子函数集，这样设备的链路层协议可以调用这些子函数。这些函数包括设置子函数、帧传输子函数以及一些搜集统计信息的子函数。可以使用 ifconfig命令查看这些统计信息。

14.7.5 初始化网络设备

网络设备驱动程序象其他的Linux设备驱动程序一样，可以嵌入到Linux系统中内核中。每一个网络设备都由一个网络设备数据结构代表，这些数据结构组成一个由 dev_base指针指向的网络设备链表。如果网络各协议层需要设备执行某些操作时，它们将调用设备提供的各种设备服务子程序，这些子程序的地址保存在设备的数据结构中。但设备初始化时，设备数据结构中只包括初始化程序和检测子程序的地址。

网络设备驱动程序需要解决两个问题。第一个问题是，不是所有构建到 Linux内核中的网络设备驱动程序都有可以控制的设备。第二个问题是，系统中的以太网设备一般调用 /dev/eth0、

/dev/eth1和类似的文件，而不管该文件中的设备驱动程序到底是什么。关于第一个问题十分容易解决。每当调用一个网络设备的初始化程序时，它都将返回一个状态值用来指示它是否找到它所控制的一个设备。如果驱动程序无法找到任何设备，那么它在设备链表中的入口将会被移走。如果设备驱动程序找到一个它控制的设备，那么它将有关该设备的信息和网络设备驱动程序支持的子函数的地址添加到设备的数据结构中。

第二个问题的解决可能稍微地复杂一些。在网络设备链表中一共有 8个标准的入口，从eth0一直到 eth7。它们初始化的过程是一样的，也就是初始化程序轮流检测每一个以太网设备驱动程序，直到发现某一个驱动程序下带有设备为止。当检测到设备以后，驱动程序将会建立此设备的数据结构。同时，驱动程序也开始初始化它控制的物理设备和有关的一些信息，例如，使用的中断号，DMA通道等等。如果驱动程序检测到它控制的设备多于一个，那么它将控制所有这些设备的数据结构。一旦所有 8个标准的/dev/ethN分配完毕以后，就不再检测其他的以太网设备了。

第15章 文 件 系 统

本章介绍有关Linux文件系统的有关内容。

15.1 Linux文件系统概述

Linux系统的一个重要特征就是支持多种不同的文件系统。这样，Linux系统就十分的灵活，并且可以十分容易地和其他操作系统共存。目前， Linux系统支持大约15个文件系统：EXT、EXT2、 XIA、MINIX、UMSDOS、MSDOS、VFAT、 PROC、SMB、NCP、ISO9660、SYSV、HPFS、AFFS 和 UFS。并且，毫无疑问，Linux系统支持的文件系统还会增加。

在Linux系统中，每一个单独的文件系统都是代表整个系统的树状结构的一部分。当挂接一个新的文件系统时， Linux把它添加到这个树状的文件系统中。所有系统中的文件系统，不管是什么类型，都挂接到一个目录下，并隐藏掉目录中原有的内容。这个目录叫做挂接目录或者挂接点。当文件系统卸载掉时，目录中的原有内容将再一次的显示出来。

初始化时磁盘将被划分成几个逻辑分区。每一个逻辑分区可以使用一种文件系统，例如EXT2文件系统。文件系统把存储在物理驱动器中的文件组织成一个树状的目录结构。可以存储文件的设备称为块设备。 Linux文件系统把这些块设备当作简单的线形块的集合，而不管物理磁盘的结构如何，而将读写某一个设备块的请求转换成特定的磁道、扇区和柱面是通过设备的驱动程序实现的。因此，不同的设备控制器控制的不同设备中的不同文件系统在 Linux中都可以同样地使用。文件系统甚至可以不在当地的系统中，也就是说，文件系统可以通过网络远程连接到本地磁盘上。请看下面的例子，这是一个Linux在SCSI 磁盘上的根文件系统：

```
A       E       boot    etc     lib     opt         tmp     usr
C       F       cdrom   fd      proc    root        var     sbin
D       bin     dev     home    mnt     lost+found
```

在这里，用户和使用文件的程序都不必知道 /C实际上是挂接的VFAT文件系统，而VFAT文件系统本身却存储在系统中的一个 IDE硬盘上。同样， /E是系统中第二个IDE控制器控制的主硬盘系统。也可以使用调制解调器将远程的文件系统挂接到 /mnt/remote目录下。

一个文件系统中不仅包括含有数据的文件，而且还存储着文件系统的结构。文件系统中的信息必须是安全和保密的。

Linux系统中的第一个文件系统是 Minux，但它的文件名只能有14个字符，最大的文件长度是64M字节。所以， 1992年4月引进了第一个专门为Linux设计的文件系统 —— ext(extended file system)。但ext的功能还是有限。最后在1993年又推出了一个新的文件系统 —— ext2。

当Linux引进ext文件系统时有了一个重大的改进：真正的文件系统从操作系统和系统服务中分离出来，在它们之间使用了一个接口层 —— 虚拟文件系统VFS（Virtual File system）。

VFS允许Linux支持多种不同的文件系统，每个文件系统都要提供给 VFS一个相同的接口。这样所有的文件系统对系统内核和系统中的程序来说看起来都是相同的。 Linux系统中的VFS层使得你可以同时在系统中透明地挂接很多不同的文件系统。

VFS能高速度及高效率地存取系统中的文件，同时它还得确保文件和数据的正确性。这两个目标有时可能相互矛盾。 VFS在每个文件系统挂接和使用时把文件系统的有关信息暂时保存

在内存中，所以当内存中的信息改变时，例如创建、写入和删除目录和文件时，系统要保证正确地升级文件系统中相关的内容。如果你了解在系统内核中运行的文件系统的数据结构，那么你也就了解了文件系统读取的数据块的情况。系统缓存中最重要的就是缓冲区缓存，它被集成到每个单独的文件系统存取它们的块设备所使用的方法中。每当系统存取数据块时，数据块都被放入缓冲区缓存中，并且依照它们的状态保存到各种各样的队列中。缓冲区缓存中不仅保存了数据缓冲区，而且还可以帮助管理和块设备驱动程序之间的异步接口。

15.2 ext2文件系统

ext2的物理结构图参见图15-1。

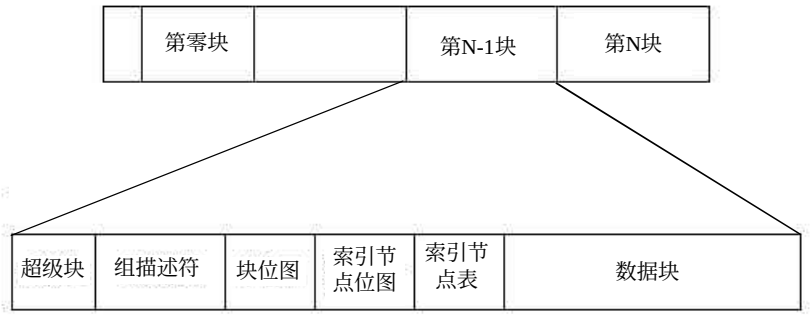

图15-1 文件使用ext2示意图

ext2文件系统是Linux系统中最为成功的文件系统，各种 Linux的系统发布都将ext2文件系统作为操作系统的基础。

ext2文件系统中的数据是以数据块的方式存储在文件中的。这些数据块具有同样的大小，并且其大小可以在ext2创建时设定。每一个文件的长度都要补足到块的整数倍。例如，如果一个块的大小是1024字节，那么一个1025字节大小的文件则占用两个数据块。所以，平均来说一个文件将浪费半个数据块的空间。但这样可以减轻系统中 CPU的负担。文件系统中不是所有的数据块都存储数据，一些数据块用来存储一些描述文件系统结构的信息。 ext2通过使用索引节点（inode）数据结构来描述系统中的每一个文件。索引节点描述了文件中的数据占用了哪一个数据块以及文件的存取权限、文件的修改时间和文件类型等信息。 ext2文件系统中的每一个文件都只有一个索引节点，而每一个索引节点都有一个唯一的标识符。文件系统中的所有的索引节点都保存在索引节点表中。 ext2中的目录只是一些简单的特殊文件，这些文件中包含指向目录入口的索引节点的指针。

对于一个文件系统来说，某一个块设备只是一系列可以读写的数据块。文件系统无须关心数据块在设备中的具体位置，这是设备驱动程序的工作。每当文件系统需要从块设备中读取数据时，它就要求设备驱动程序读取整数数目的数据块。 ext2文件系统将它所占用的设备的逻辑分区分成了数据块组。每一个数据块组都包含一些有关整个文件系统的信息以及真正的文件和目录的数据块。

15.2.1 ext2的索引节点

ext2的索引节点图参见图15-2。

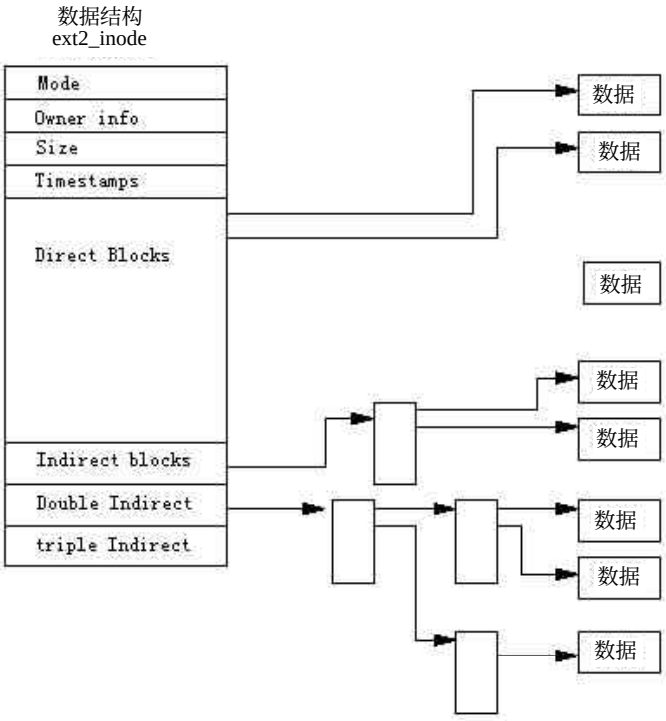

图15-2 EXT2 系统索引节点示意图

在ext2文件系统中索引节点是一切的基础，文件系统中的每一个文件和目录都使用一个唯一的索引节点。每一个数据块组中的索引节点都保存在索引节点表中。数据块组中还有一个索引节点位图，它用来记录系统中已分配和未分配的索引节点。下面是 ext2的索引节点的一些主要的字段：

1. mode

这里保存两个信息：一个是此索引节点描述的是什么，另一个是用户拥有的权限。例如，对于ext2，一个索引节点可以描述文件、目录、符号连接、块设备、字符设备以及 FIFO结构。

2. Owner Information

这是文件或目录所有者的用户和组标识符。这使得文件系统可以正确地授权某种存取操作。

3. Size

文件的字节大小。

4. Timestamps

索引节点建立的时间和索引节点最后修改的时间。

5. Datablocks

指向存储此索引节点描述文件的数据块的指针。前十二个指针是指向存储数据的物理数据块的指针，而后三个指针则包括不同级别的间接指针。例如，两级指针指向一个指向其他指针块的指针块。这意味着小于或者等于 12个数据块的文件的存取速度要高于多于12个数据块的文件。

你应该注意到ext2的索引节点可以描述一些特殊的设备文件。这些设备文件不是真正的文

件，但系统中的程序可以使用这些设备文件来存取它们相关的设备。所有的这些设备文件都在/dev目录下面。例如，挂接程序可以把它希望挂接的设备文件作为它的一个参数。

15.2.2 ext2超级块

超级块（Superblock）存储着描述文件系统的大小和形状的基本信息。文件系统的管理员可以使用其中的信息来使用和维护文件系统。一般情况下，当文件系统挂接时，系统只读取数据块组0中的超级块，但每一个数据块组中都包含一个超级块的副本，以防系统崩溃时使用。超级块包括如下的主要信息：

1. Magic Number（幻数）

使挂接程序确认这是ext2文件系统的超级块。目前其值为0xEF53。

2. Revision Level（修订级别）

这是文件系统的主版本号和从版本号。挂接程序可以根据此信息决定此文件系统是否支持一些特定文件系统的函数。

3. Mount Count（挂接数）和 Maximum Mount Count（最大挂接数）

系统用来决定文件系统是否应该全面地检查。文件系统每挂接一次， mount count的值就会加1。当mount count的值和maximum mount count的值相等时，系统将显示maximal mount count reached, running e2fsck is recommended信息，提示用户进行文件系统的检查。

4. Block Group Number（块组号）

包含此超级块的数据块组号。

5. Block Size（块大小）

文件系统中数据块的大小，例如1024字节。

6. Blocks per Group（每组块数）

数据块组中的数据块数目和Block Size一样，它在文件系统创建以后就是固定的了。

7. Free Blocks（空闲块）

文件系统中空闲的数据块的数目。

8. Free Inodes（空闲索引节点）

文件系统中空闲的索引节点的数目。

9. First Inode（第一个索引节点）

文件系统中的第一个索引节点号。在一个ext2根文件系统中，第一个索引节点是/目录的入口。

15.2.3 ext2 数据块组描述符

每一个数据块组都有一个描述它的数据结构和超级块一样，在每一个数据块组中都要复制一份数据块组描述符。

数据块组描述符包含以下的信息：

1. Blocks Bitmap（块位图）

数据块组中数据块分配位图所占的数据块数。在数据块分配和数据块撤消时使用。

2. Inode Bitmap（索引节点位图）

数据块组中索引节点分配位图所占的数据块数。在索引节点的分配和撤消中使用。

3. Inode Table（索引节点表）

数据块组中索引节点表所占的数据块数。系统中的每一个索引节点都由一个数据结构来描述。

4. Free blocks count（空闲块数）、Free Inodes count（空闲索引节点数）, Used directory count（已用目录数）

这是空闲的数据块数，空闲的索引节点数和已用的目录数。

数据块组描述符组成一个数据块组描述符表。每一个数据块组在其超级块之后都包含一个数据块组描述符表的副本。 EXT2文件系统事实上只是使用数据块组 0中的数据块描述符表。

15.2.4 ext2 中的目录

在ext2文件系统中，目录是一些特殊的文件，它们用来创建和保存系统中文件的存取路径。图15-3是一个目录入口在内存中的结构图。

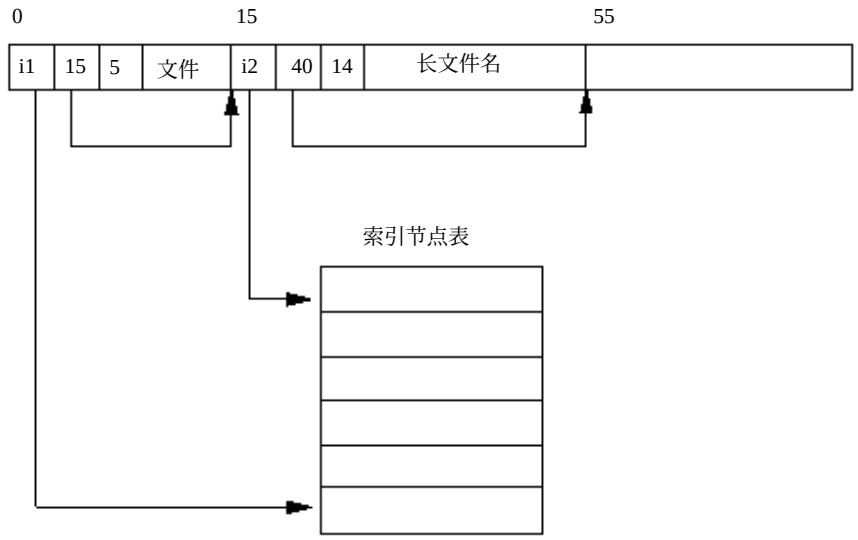

图15-3 EXT2文件系统目录结构示意图

一个目录文件包括很多的目录入口，一个目录入口包括以下的内容：

1. inode（索引节点）

目录入口的索引节点。这是保存在数据块组中的索引节点表数组的索引值。在上图中，file的目录入口的索引节点号的引用是i1。

2. name length（名称长度）

目录入口的字节长度。

3. name（名称）

目录入口的名字。

每一个目录的头两个入口都是标准的 . 和 .., 分别代表本目录和父目录。

15.2.5 在ext2 文件系统中查找文件

Linux系统的文件名格式和 Unix系统的文件名格式一样，其中的目录名用斜杠（ /）分隔。

例如，文件名/home/zws/.cshrc，其中/home 和 /zws是目录名，.cshrc则是文件名。Linux系统中的文件名可以由任何可打印的字符组成，也可以是任何的长度。系统通过分析目录中的文件，来查找文件对应的索引节点。

系统需要的第一个索引节点是文件系统根目录的索引节点，它的值保存在文件系统的超级块中。要读取一个ext2文件系统的索引节点，我们必须在相应的数据块组中的索引节点表中查找。例如，如果一个根目录的索引节点值是 42，那么我们需要读取在数据块组 0中的索引节点表中的第42个索引节点。根目录索引节点是一个 ext2的目录节点，也就是说，根目录索引节点的模式（mode）是目录，而它的数据块中包含的是ext2目录的入口。

home目录只是众多目录入口中的一个，我们可以从中查找出描述 /home/zws目录的索引节点值。接下来我们读取这个目录（首先读取它的索引节点，然后读取此索引节点描述的数据块中的zws目录），从中查找描述/home/zws目录的索引节点值。最后，在描述/home/zws目录的索引节点指向的目录入口中找到 .cshrc文件的索引节点值，通过它的索引节点值找到存储在文件中的数据的数据块。

15.2.6 改变ext2 文件系统中文件的大小

文件系统的一个常见的问题就是文件碎片问题。存储文件数据的数据块可能散布在整个文件系统中，这样顺序存取一个文件的数据块可能变得效率越来越低。 ext2文件系统通过为一个文件的新数据块分配靠近当前数据块的物理块或同一个数据块组中的数据块来解决文件碎片问题。只有当这样的分配策略不能实现时， ext2才为新的文件数据块分配其他数据块组中的数据块。

当进程往文件中写入数据时， ext2文件系统都要检查数据是否已经超出了文件最后分配的数据块的范围。如果写入的数据超出了该范围，那么文件系统必须为此文件分配一个新的数据块。在这之前，进程不能继续运行。 ext2文件系统的数据块分配过程所做的第一件事就是锁定此文件系统的 ext2超级块。分配和撤消数据块需要改变超级块中字段的内容，并且Linux文件系统不允许多个进程同时修改超级块。如果另外的进程在此时也需要分配新的数据块，那么它必须等待直到正在运行的进程处理完对超级块的修改并释放对超级块的控制才能恢复运行。对超级块的操作本着先来先服务的策略。在锁定了超级块以后，进程将检查文件系统中有无足够的空闲数据块。如果没有，那么进程的分配请求将失败，进程将放弃对文件系统超级块的控制权。如果系统中有足够的空闲数据块，则进程将为文件分配所需要的数据块。

如果ext2文件系统内建有预分配的数据块，那么我们可以使用预分配数据块。预分配数据块实际上并不存在，它们保留在已分配的数据块位图中。申请分配新数据块的文件的 VFS索引节点有两个特别的字段：prealloc_block 和 prealloc_count，分别是第一个预分配数据块的块号和可以使用的预分配块数。如果没有预分配数据块或者系统中没有预分配数据块功能， ext2文件系统必须分配一个新的数据块。它首先查看文件中最后一个数据块的后一个数据块是否空闲。逻辑上，这个数据块是最为有效的数据块，因为这样可以加速对文件顺序的存取。如果此数据块没有空闲，那么文件系统将加大搜索的范围，在 64个数据块的范围内查找理想的数据块。

如果甚至连这样的空闲数据块找不到，进程则开始查看其他数据块组直到找到一个空闲的数据块为止。数据块分配程序查找某一个数据块组中的一簇，也就是 8个空闲的数据块。如果无法找到8个连续的空闲数据块，数据块分配程序将查找较少的连续的空闲数据块。

无论在那一个数据块组中查找到空闲的数据块，数据块分配程序都将更新数据块组中的数据块位图，并且在缓冲区缓存中分配一个数据缓冲区。这个分配的数据缓冲区由设备标识符和已分配的数据块的块号来标识。最后将超级块标记为 dirty来表明超级块已经改动，同时文件系统将超级块解锁。此时，如果有任何进程等待使用超级块，那么系统将允许等待队列中的第一个进程控制超级块继续运行。进程的数据被写入到新分配的数据块中。如果新分配的数据块无法容纳全部的数据，则整个分配过程将重复进行。

15.3 VFS

图15-4显示了Linux系统内核中的VFS和实际文件系统之间的关系。VFS必须管理同时挂接在系统上的不同的文件系统。它通过使用描述整个 VFS的数据结构和描述实际挂接的文件系统的数据结构来管理这些不同的文件系统。

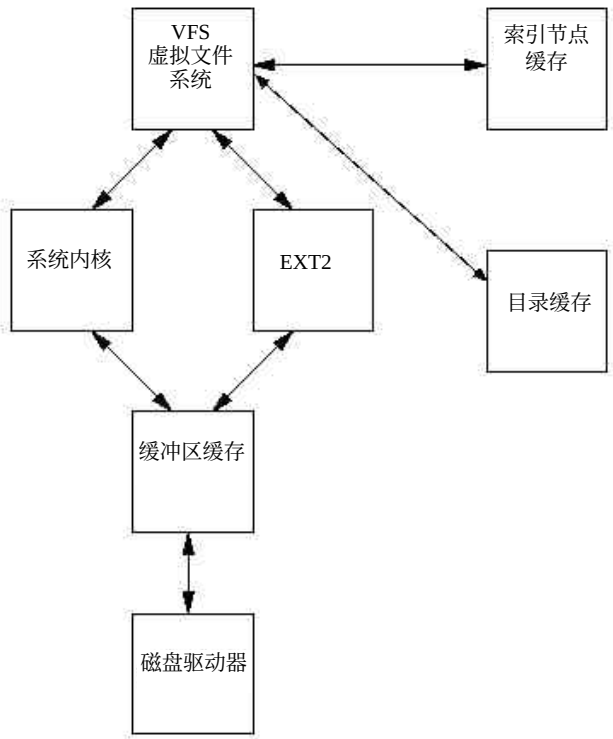

图15-4 VFS系统结构示意图

令人易于混淆的是，VFS和ext2文件系统一样使用超级块和索引节点来描述系统中的文件。和ext2中的索引节点一样，VFS的索引节点用来描述系统中的文件和目录。

当一个文件系统初始化时，它将在 VFS中登记。这些过程在系统启动操作系统初始化时完成。实际的文件系统或者内建到系统内核中，或者作为可装入模块在需要时装入。当一个基于块设备的文件系统挂接时，当然也包括根文件系统， VFS首先要读取它的超级块。每一个文件系统的超级块读取程序都必须先清楚文件系统的结构，然后把有关的信息添加到 VFS的超级块数据结构中。VFS中保存了系统中挂接的文件系统的链表以及这些文件系统对应的 VFS超级块。每一个VFS超级块都包含一些信息和指向一些执行特别功能的子程序的指针。例如，代表挂接的ext2文件系统的VFS超级块包含了一个指向ext2索引节点读取程序的指针。这个ext2索引节点

读取程序，和其他文件系统索引节点的读取程序一样，把信息添加到VFS索引节点中的字段中。每一个 VFS超级块都包含了一个指向文件系统中第一个VFS索引节点的指针。对于根文件系统来说，第一个VFS索引节点是代表/目录的索引节点。这种对应关系对于ext2文件系统来说十分有效，但对于其他的文件系统则效果一般。

当系统中的进程存取目录和文件时，需要调用系统中的子程序来遍历搜索系统中的 VFS索引节点。

例如，键入ls或者cat命令将导致VFS搜索整个文件系统中的VFS索引节点。因为系统中的每一个文件和目录都由一个索引节点来表示，那么有些索引节点将经常会被重复地搜索。这些索引节点将保存在索引节点缓存中，这样将增快以后的存取速度。如果要找的索引节点不在索引节点缓存中，那么进程将调用一个特殊的系统程序来读取相应的索引节点。读取了索引节点以后，此索引节点将放在索引节点缓存中。最少用到的索引节点将被交换出索引节点缓存。

所有的Linux系统中的文件系统都使用一个相同的缓冲区缓存来保存相应的块设备中的数据缓冲区，这样可以加速所有的文件系统对其物理设备的存取。

这个缓冲区缓存和 Linux系统中的各个不同的文件系统无关，并且缓冲区缓存集成到了Linux内核用来分配和读写数据缓冲区的机制中。把 Linux中的各个文件系统和其相应的物理设备分开是有很大好处的。所有的块设备都在 Linux系统内核中注册，并且提供一个相同的，以数据块为基础的，一般情况下是异步的接口。当实际文件系统从相应的物理设备中读取数据时，将导致文件系统控制的物理设备读取它的物理块。当文件系统读取数据块时，它们把数据块保存在所有文件系统和系统内核共同使用的公共的缓冲区缓存中。缓冲区缓存中的缓冲区以它们的数据块号和读取设备的标识符名来区分。所以，如果需要一些相同的数据，那么这些数据将从系统的缓冲区缓存而不是磁盘中读出，这就加快了读取的速度。 VFS中也保存了一个目录查找缓存，一些经常使用的目录的索引节点将会很快地找到。目录缓存中并不保存目录的索引节点本身，索引节点保存在索引节点缓存中，目录缓存只是简单地保存目录的名字和目录的索引节点号的对应关系。

15.3.1 VFS 超级块

每一个挂接的文件系统都有一个VFS超级块， VFS超级块包括以下主要信息：

1. 设备

这是存储文件系统的物理块设备的设备标识符。例如，系统中第一个 IDE磁盘/dev/hda1的标识符为0x301。

2. 索引节点指针

挂接的（mounted）索引节点指针指向文件系统的第一个索引节点。覆盖的（ covered）索引节点指针指向文件系统挂接的目录的索引节点。根目录文件系统的 VFS超级块中没有覆盖的索引节点指针。

3. 数据块大小

文件系统数据块的字节数，例如， 1024字节。

4. 超级块操作

指向文件系统的一系列超级块子过程的指针。 VFS可以使用这些子过程读写索引节点和超级块。

5. 文件系统类型

指向挂接的文件系统的file_system_type数据结构的指针。

6. 文件系统的特殊的信息

指向文件系统所需要的信息的指针。

15.3.2 VFS 索引节点

像EXT2文件系统一样，VFS 中每一个文件和目录都有一个且仅有一个VFS索引节点。

每一个VFS索引节点中的信息都是文件系统的一些特殊的子过程根据相应的文件系统的信息产生的。VFS索引节点只在系统需要时才保存在系统内核的内存及 VFS索引节点缓存中。它包括以下主要内容：

1. 设备

这是索引节点代表的文件或目录所在的设备的设备标识符。

2. 索引节点号

索引节点号在文件系统中是唯一的。索引节点号加上设备号在 VFS中是唯一的。

3. 模式

用来描述VFS索引节点代表的是文件、目录或其他内容，以及对它的存取权限。

4. 用户的标识符

表示用户的标识符。

5. 时间

创建、修改和写入的时间。

6. 数据块大小

文件数据块的大小，例如，1024字节。

7. 索引节点操作

一个指向一系列子程序地址的指针。这些子程序和相应的文件系统有关，它们执行有关此索引节点的各种操作，例如，截断此索引节点代表的文件。

8. 计数器

正在使用VFS索引节点的系统进程。

9. 锁定

用于锁定VFS索引节点，例如，当文件系统读取索引节点的时候。

10. 已修改（dirty）

指示索引节点是否已经被修改了，如果已修改，则相应的文件系统也需要修改。

11. 文件系统的一些特殊的信息

指向文件系统所需要的信息的指针。

15.3.3 登记文件系统

构建Linux系统内核时，需要选择文件系统。当系统内核建好以后，文件系统的起始程序将包括调用系统中所有文件系统的初始化程序的调用。

Linux文件系统也可以作为系统的模块在需要时装入，或者使用 insmod命令手工地装入。每当一个文件系统装入时，它都要在系统内核中登记。同样，当文件系统卸载时，也要从系统内核中取消登记。每一个文件系统的初始化程序都在 VFS中登记，并且创建一个file_system_type数据结构，其中包括文件系统的名字和指向VFS超级块读取程序地址的指针。

图15-5显示所有file_system_type 结构组成的一个由file_systems 指针指向的链表。每一个

file_system_type 结构都包括以下的信息：

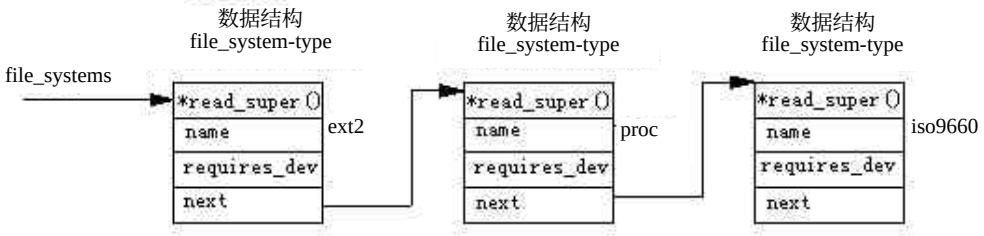

图15-5 文件系统结构示意图

1. 超级块读取程序

当一个文件系统挂接时，VFS使用此程序读取文件系统的超级块。

2. 文件系统名

文件系统的名称，例如ext2。

3. 需要的设备

指出文件系统是否需要设备的支持。并不是所有的文件系统都需要设备的支持，例如，/proc就不需要块设备的支持。

你可以通过查看/proc/filesystems了解系统中已注册的文件系统，例如：

```
ext2
nodev          proc
iso9660
```

15.3.4 挂接文件系统

当一个超级用户试图挂接一个文件系统时，Linux系统内核必须首先检查系统调用需要使用的参数的有效性。虽然挂接时要做一些基本的检查，但它并不知道此系统内核中已经支持的文件系统，也不知道文件系统的挂接点是否存在。请看下面这个命令：

```
$ mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom
```

这个mount命令将向内核传递三个参数：文件名、包含文件系统的物理设备和文件系统的挂接点。

VFS所要做的第一件事就是找到要挂接的文件系统。

VFS首先通过查找file_systems指针指向的链表中的每一个file_system_type数据结构来搜索已知的文件系统。

如果VFS找到了一个匹配的名字，那么说明系统内核支持此种文件系统。这样可以得到读取文件系统的超级块的程序的地址。如果没有找到相应的名字，但内核可以装入文件系统模块，那么系统内核将要求内核守护进程装入相应的文件系统模块。

接下来如果mount命令中的物理设备没有挂接到系统中，那么 VFS必须查找作为新文件系统挂接点的目录的VFS索引节点。一旦找到索引节点，VFS将检查此目录下有无其他挂接的文件系统。同一个目录下不能挂接多个文件系统。

VFS挂接程序必须分配一个VFS超级块，并且传递给它一些有关文件系统挂接的信息。系统中所有的VFS超级块都在由super_block数据结构组成的super_blocks数组中。超级块读取程序使用它从物理设备中读取的信息来填充 VFS超级块的有关字段。对于ext2文件系统，这个映射和翻译信息的过程十分的容易，它只是简单地把 ext2的超级块转换成VFS的超级块。对于其

他的文件系统，例如msdos文件系统，这个过程将较为复杂。无论是那一种的文件系统，创建VFS超级块都意味着文件系统必须从存储该文件系统的块设备中读取有关的信息。如果该块设备无法读取，或者该块设备不包括此文件系统， mount 命令都将失败。

一个挂接的文件系统参见图15-6。

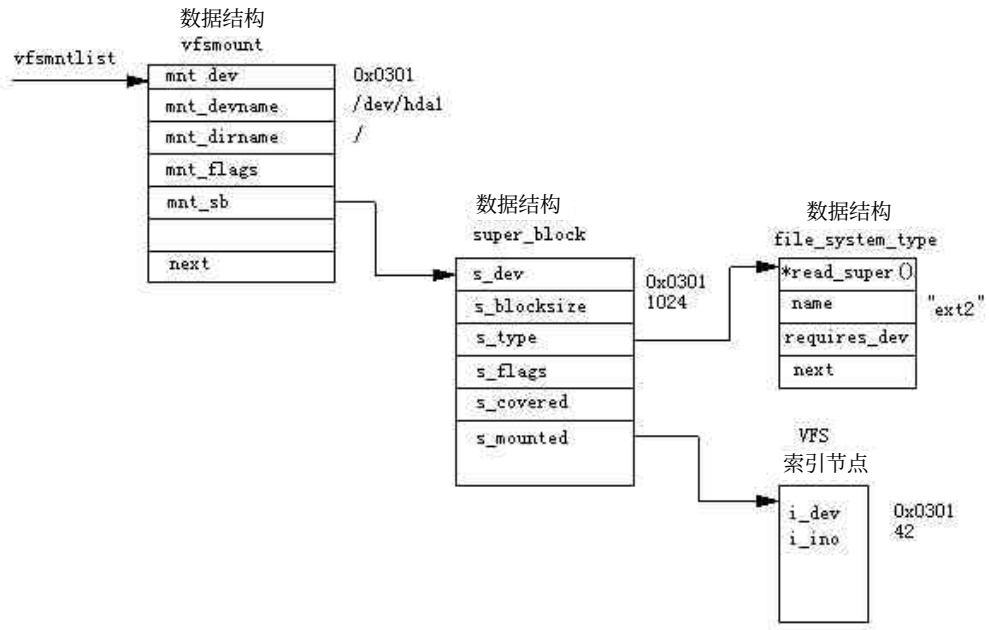

图15-6 虚拟文件系统挂接结构示意图

每一个挂接的文件系统都使用一个vfsmount数据结构，这些vfsmount结构组成了一个由指针vfsmntlist指向的链表。另一个指针 vfsmnttail指向链表的最后一个入口，并且还有一个mru_vfsmnt指针指向链表中最近被使用的文件系统。每一个 vfsmount结构都包括存储文件系统的块设备的设备号，文件系统挂接的目录和一个指向文件系统的 VFS超级块的指针。而VFS超级块中则包括指向此文件系统的file_system_type数据结构的指针和指向此文件系统根索引节点的指针。当文件系统装入以后，此根索引节点就一直保存在 VFS索引节点缓存中。

15.3.5 在VFS中查找文件

如果希望在VFS中查找一个文件的VFS索引节点， VFS必须一个个解释文件路径名中的中间目录名，查找中间目录名的索引节点。每一个目录名的查找都要调用文件系统的查找子程序，这个子程序的地址保存在该目录的父目录的 VFS索引节点中。每当文件系统查找一个索引节点时，它都要在目录缓存中查找该目录。如果目录缓存中没有该目录，它将从文件系统中或索引节点缓存中调入此目录的索引节点。

15.3.6 撤消文件系统

撤消文件系统基本上与挂接文件系统的过程相反。

如果文件系统的某一个文件正在被使用，那么该文件系统就不能被撤消。例如，如果系统中的一个进程正在使用/mnt/cdrom目录或者它的一个子目录，那么你就不能撤消此目录。如果

有进程正在使用将要被撤消的文件系统，那么此文件系统的某一个 VFS索引节点可能正在VFS索引节点缓存中。检查程序可以通过查找索引节点链表来确定有无该文件系统的索引节点。如果此挂接文件系统的VFS超级块已经被改写，那么VFS超级块就必须写回到磁盘的文件系统中。写回完成以后， VFS超级块占用的内存就可以释放。最后这个文件系统的 vfsmount结构从vfsmntlist链表中断开。

15.3.7 VFS 索引节点缓存

当查找挂接的文件系统时，文件系统中的索引节点将被不断的读取。 VFS提供的索引节点缓存可以加快对系统中所有挂接的文件系统的存取。

VFS索引节点缓存使用散列表实现，表的入口是指向具有相同 hash值的VFS索引节点链表的指针。索引节点的hash值是通过索引节点号和存储文件系统的物理设备号计算出来的。每当VFS需要存取一个索引节点时，它将首先查找索引节点缓存。为了在索引节点缓存中找到一个特定的索引节点，系统首先需计算索引节点的 hash值，然后使用hash值作为索引节点的散列表的索引。这样就可以找到一个指向具有相同 hash值的索引节点链表的指针。系统接着逐个读取链表中的每一个索引节点直到找到一个索引节点号和设备标识符都相同的索引节点为止。

如果系统可以在索引节点缓存中找到索引节点，那么此索引节点的计数器将加 1，表明又有另一个进程在使用该索引节点。否则，系统必须找到一个空闲的 VFS索引节点，这样系统才可以从内存中读取索引节点。 VFS有几种方法可以获得一个空闲的索引节点。如果系统可以分配更多的索引节点，那么系统将分配新的索引节点。系统把新分配的内存页面分隔成一些新的、空闲的索引节点，然后把这些新的索引节点放到索引节点链表中。系统中的所有的 VFS索引节点都放在一个由指针 first_inode指向的链表中，当然也存放在索引节点散列表中。如果系统已经不能分配新的索引节点了，那么它必须查找一个已用的索引节点以便重新分配。可以重新分配的索引节点是那些用户计数器为 0的索引节点，这说明系统中没有任何进程正在使用这些索引节点。一些非常重要的索引节点，例如文件系统的根目录索引节点，它们的索引节点计数器总是大于0，这样它们永远不能被重新分配。一旦找到一个可以重新分配的索引节点，它将被清除。如果此索引节点被修改过，则必须写回到文件系统中。如果它被锁定，则系统需要等到此索引节点解锁以后再继续进行。

不论用什么方法找到一个新的索引节点，系统都必须调用一个特殊的子程序来把实际文件系统的信息添加到此索引节点中。当向新的索引节点中写入信息时，此索引节点计数器的值为1，并且系统将锁定此索引节点直到索引节点中的信息有效为止。

为了获得可以使用的VFS索引节点，文件系统可能需要读取很多其他的索引节点。例如，当你读取一个目录时，只有最后一个目录的索引节点才是你所需要的，但你还必须要读取此目录路径名中其他中间目录的索引节点。当 VFS索引节点缓存用完以后，最少用到的索引节点将会被扔掉，而较为常用的索引节点将继续保存在 VFS索引节点缓存中。

15.3.8 VFS目录缓存

为了加速对常用目录的存取， VFS还提供一个目录缓存。

当实际文件系统读取一个目录的时候，目录的详细信息将添加到目录缓存中。下一次查找同一个目录时，系统就可以在目录缓存中找到此目录的有关信息。只有短于 15个字符的目录才能保存在目录缓存中，但正是这些短目录才是最经常用到的目录。

目录缓存包括一个散列表，表中的每一个入口都是指向具有相同 hash值的目录缓存链表的

指针。散列表使用文件系统所在设备的设备号和目录名来计算位移。

为了保证缓存的有效和及时的更新，VFS使用一个最新用到（LRU）目录缓存入口的链表。当一个目录入口第一次被放入到目录缓存时，也就是说当第一次查找此目录时，此目录将添加到第一层LRU链表的末尾。如果此缓存已满，它将替换LRU链表中最前面的一个目录入口。当此目录再一次被存取的时候，此目录将提升到第二层LRU链表的末尾。它也可能替换掉第二层LRU链表中最前面的目录入口。这种替换第一层和第二层LRU链表最前面的目录入口的策略可以把最近最不常用到的目录入口替换掉。第二层LRU链表的目录入口的级别要比第一层LRU链表的入口高。这也是我们希望得到的效果。

15.4 缓冲区缓存

当系统中的进程使用挂接的文件系统时，挂接的文件系统将会产生很多和块设备之间的读写请求。所有的数据块读写请求都通过标准的内核调用以buffer_head数据结构的形式传递给设备驱动程序。数据结构buffer_head中包含了设备驱动程序所需要的所有信息，其中设备标识符唯一地表示了所使用的设备，数据块号表明了驱动程序需要读取设备中的那一块。所有的块设备都可以认为是同样大小的数据块的一个线性的集合。为了加速对物理块设备的存取， Linux系统中使用了一个数据块缓冲区缓存。系统中所有的数据块缓冲区都存储在这个缓冲区缓存的某一个地方，即使是一些新的，没有用到的数据块缓冲区。所有的物理块设备都可以共享这个缓冲区缓存。在某一时刻，缓存中可以保存着属于不同块设备的，不同状态的多个缓冲区。如果缓冲区缓存中的数据是有效的，那么就可以节省系统存取物理块设备的时间。任何一个从块设备中读取的数据块缓冲区或者往块设备中写入的数据块缓冲区都要通过缓冲区缓存。

缓存中的数据块缓冲区是以拥有此数据块的设备的设备标识符和缓冲区的块号来唯一标识的。缓冲区缓存由两部分组成。第一部分是空闲的数据块缓冲区的链表。每一个有效大小的缓冲区都有一个链表。当系统中的空闲缓冲区新建或者缓冲区被扔掉以后就在这里排队。当前系统支持的缓冲区的大小是512、1024、2048、4096 和 8192 字节。第二部分是缓存本身。这是一个由指针数组构成的散列表，其中的指针指向具有相同 hash值的缓冲区。散列表的索引由设备的标识符和数据块的块号产生。图 15-7显示了散列表和几个散列表的入口。数据块缓冲区要么在一个空闲数据块缓冲区链表中，要么在缓冲区缓存中。当处于缓冲区缓存中时，它们也会在LRU链表中。每一种缓冲区类型都会有一个LRU链表。系统使用这些链表对某一种类型的缓冲区进行某种操作，例如，把保存了新数据的缓冲区写入到磁盘中。数据库缓冲区的类型反映了缓冲区的状态， Linux系统现在支持以下几种类型：

1. clean（干净的）

未被使用的，新的缓冲区

2. locked（锁定的）

已经被锁定的缓冲区，等待数据的写入。

3. dirty（已用的）

已经改变的缓冲区。也就是缓冲区中包括有效的新数据。这些数据将要被写入到磁盘中，但目前还未写入。

4. shared（共享的）

可以共享的缓冲区。

5. unshared（非共享的）

这些缓冲区以前可以被共享，但现在不能被共享。

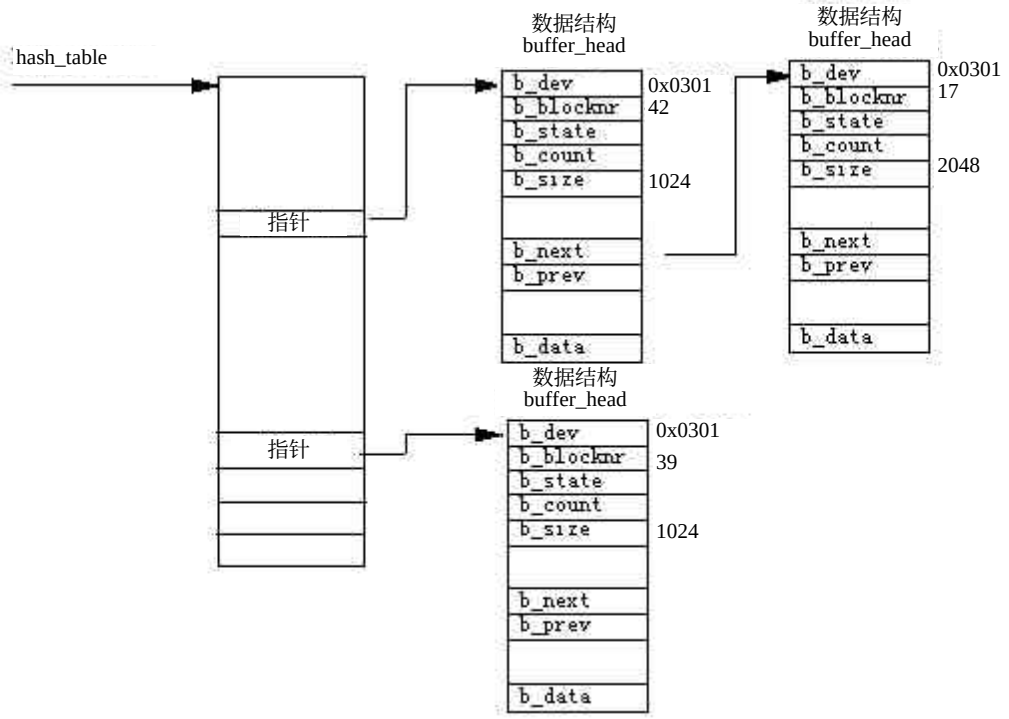

图15-7 缓冲区缓存示意图

当一个文件系统想要从物理块设备中读取一个数据块时，它都首先试图从缓冲区缓存中查找此数据块。如果缓冲区缓存中没有此数据块，那么文件系统将会从相应大小的空闲数据块缓冲区链表中分配一个空闲的数据块缓冲区，然后把它放入到缓冲区缓存中。如果文件系统希望查找的数据块缓冲区已经在缓冲区缓存中了，那么也许此数据块缓存包含的不是最新的数据。如果是这样，或者这是一个新分配的数据块缓冲区，那么文件系统则必须请求设备驱动程序从磁盘中读取相应的数据块。

像所有的其他缓存一样，系统必须经常地维护缓冲区缓存以使得它可以十分有效地运行，并且可以在各个使用缓冲区缓存的块设备之间公平地分配缓存入口。 Linux系统使用bdflush内核守护进程来完成管理缓存的功能。

bdflush 内核守护进程

内核守护进程bdflush的任务是当缓冲区缓存中被改动的缓冲区数量太多时，提供一个管理功能。它在系统启动时作为一个系统的线程运行，此线程的名字叫做 kflushd。你可以使用ps命令查看系统中的此线程。在大部分时间中，此守护进程都处于睡眠状态，等待系统中被改动的数据块缓冲区的数量增大到一定的值。每当缓冲区缓存中的数据块缓冲区分配和放弃时，文件系统都要检查缓存中被改动的数据块缓冲区的数量。当此数量达到一定的百分比数时， bdflush守护进程将被唤醒。缺省的百分比是60%，但如果系统中急需数据块缓冲区，那么守护进程可以随时被唤醒。你可以使用update命令查看和改变这个百分比：

```
# update -d

bdflush version 1.4
0:    60 Max fraction of LRU list to examine for dirty blocks
```

```
1:  500 Max number of dirty blocks to write each time bdflush activated
2:   64 Num of clean buffers to be loaded onto free list by refill_freelist
3:  256 Dirty block threshold for activating bdflush in refill_freelist
4:   15 Percentage of cache to scan for free clusters
5: 3000 Time for data buffers to age before flushing
6:  500 Time for non-data (dir, bitmap, etc) buffers to age before flushing
7: 1884 Time buffer cache load average constant
8:    2 LAV ratio (used to determine threshold for buffer fratricide).
```

所有被改动的数据块缓冲区都连接到BUF_DIRTY LRU链表中。

15.5 /proc 文件系统

/proc文件系统是最能显示Linux的VFS作用的文件系统。它实际上并不存在，并且 /proc目录和它下面的子目录以及其中的文件也不存在。但和其他实际存在的文件系统一样， /proc文件系统也在VFS中注册。当VFS调用/proc文件系统中打开的目录或者文件的索引节点时， /proc文件系统才利用系统内核中的有关信息创建这些文件和目录。例如， /proc/devices文件是从内核中描述系统设备的数据结构中产生的。/proc文件系统提供给用户一个可以了解内核内部工作过程的可读窗口。

第16章 网 络 系 统

网络和Linux系统几乎可以说是同义词，因为Linux系统是WWW的产物。下面我们讨论一下Linux系统是如何支持TCP/IP协议的。

TCP/IP协议最初用来支持计算机和ARPANET网络之间通信。现在广泛使用的World Wide Web就是从ARPANET中发展来的，并且World Wide Web使用的也是TCP/IP协议。在UNIX系统中，首先带有网络功能的版本是4.3 BSD。Linux系统的网络功能就是以UNIX 4.3 BSD为模型发展起来的，它支持BSD套接口和全部的TCP/IP功能。这样UNIX系统中的软件就可以十分方便地移植到Linux 系统中了。

16.1 TCP/IP 网络简介

现在简单地介绍TCP/IP网络的主要原理。

在一个IP(Internet Protocol)网络中，每一台计算机都有一个32位的IP地址。每台计算机的IP地址都是唯一的。WWW是一个范围十分大，并且不断增长的IP网络，所以网络上的每台计算机都必须有一个唯一的IP地址。IP地址是用 . 分隔开的4个十进制数，例如16.42.0.9。实际上IP地址可以分为两部分：一部分是网络地址，另一部分是主机地址，例如，在 16.42.0.9中，16.42是网络地址，0.9则为主机地址。而主机地址又可以分为子网地址和主机地址。计算机的IP地址很不容易记忆，如果使用一个名字就可以方便得多。如果使用名字，则必须有某一种机制将名字转化为IP地址。这些名字可以静态地保存在 /etc/hosts文件中，或者Linux系统请求域名服务器（DNS服务器）来转换名字。如果使用DNS服务器的话，本地的主机则必须知道一个或者多个DNS服务器的IP地址，这些信息保存在/etc/resolv.conf文件中。

当你和其他计算机相连时，系统要使用IP地址和其他计算机交换数据。数据保存在IP数据包中。每一个IP数据包都有一个IP数据头，其中包括源地址和目的地址，一个数据校验和以及其他一些有关的信息。IP数据包的大小随传输介质的不同而不同，例如，以太网的数据包要大于PPP的数据包。目的地址的主机在接收数据包后，必须再将数据装配起来，然后传送给接收的应用程序。

连接在同一个IP子网上的主机之间可以直接传送IP数据包，而在不同子网之间的主机却要使用网关。网关用来在不同的子网之间传送数据包。

IP协议是一个传输层的协议，其他的协议可以利用 IP协议来传输数据。TCP(Transmission Control Protocol)协议是一个可靠的点到点之间的协议，它使用IP协议来传送和接收自己的数据包，如图16-1所示。TCP协议是基于连接的协议。需要通信的两个应用程序之间将建立起一条虚拟的连接线路，即使其中要经过很多子网、网关和路由器。 TCP协议保证在两个应用程序之间可靠地传送和接收数据，并且可以保证没有丢失的或者重复的数据包。当TCP协议使用IP协议传送它自己的数据包时，IP数据包中的数据就是TCP数据包本身。相互通信的主机中的IP协议层负责传送和接收IP数据包。每一个IP数据头中都包括一个字节的协议标识符。当TCP协议请求IP协议层传送一个IP数据包时，IP数据头中的协议标识符指明其中的数据包是一个TCP数据包。接收端的IP层则可以使用此协议标识符来决定将接收到的数据包传送到那一层, 在这里是TCP协议层。当应用程序使用TCP/IP通信时，它们不仅要指明目标计算机的IP地址，也要指明应用程序使用

的端口地址。端口地址可以唯一地表示一个应用程序，标准的网络应用程序使用标准的端口地址，例如，web服务器使用端口80。你可以在/etc/services中查看已经登记的端口地址。

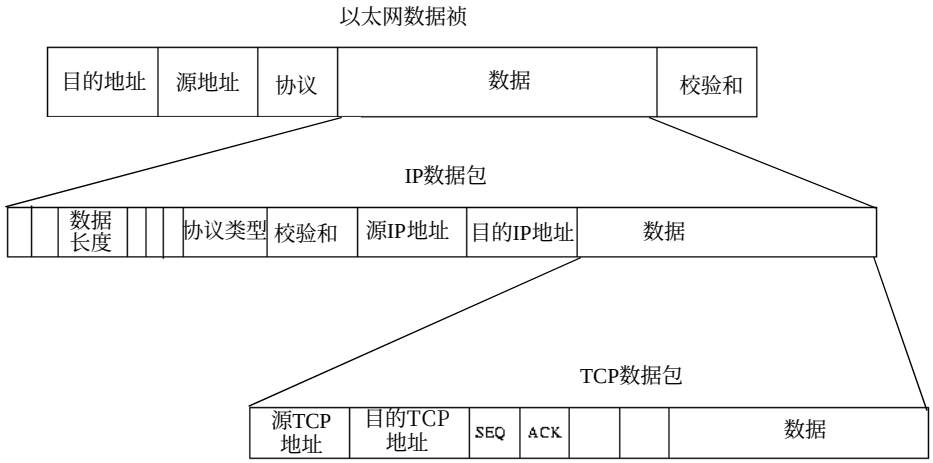

图16-1 网络协议示意图

IP 协议层也可以使用不同的物理介质来传送IP数据包到其他的IP地址主机。这些介质可以自己添加协议头。例如以太网协议层、 PPP协议层或者SLIP协议层。以太网可以同时连接很多个主机，每一个主机上都有一个以太网的地址。这个地址是唯一的，并且保存在以太网卡中。所以在以太网上传输IP数据包时，必须将IP数据包中的IP地址转换成主机的以太网卡中的物理地址。Linux系统使用地址解决协议（ARP）来把IP地址翻译成主机以太网卡中的物理地址。希望把IP地址翻译成硬件地址的主机使用广播地址向网络中的所有节点发送一个包括 IP地址的ARP请求数据包。拥有此 IP地址的目的计算机接收到请求以后，返回一个包括其物理地址的ARP应答。ARP协议不仅仅限于以太网，它还可以用于其他的物理介质，例如 FDDI等。那些不能使用ARP的网络设备可以标记出来，这样 Linux系统就不会试图使用 ARP。系统中也有一个反向的翻译协议，叫做RARP，用来将主机的物理地址翻译成 IP地址。网关可以使用此协议来代表远程网络中的IP地址回应ARP请求。

16.2 TCP/IP网络的分层

图16-2显示了Linux系统网络实现的分层结构。 BSD 套接口是最早的网络通信的实现，它由一个只处理BSD 套接口的管理软件支持。其下面是 INET套接口层，它管理TCP协议和UDP协议的通信末端。UDP(User Datagram Protocol)是无连接的协议，而TCP则是一个可靠的端到端协议。当网络中传送一个 UDP数据包时，Linux系统不知道也不关心这些 UDP数据包是否安全地到达目的节点。TCP数据包是编号的，同时TCP传输的两端都要确认数据包的正确性。 IP协议层是用来实现网间协议的，其中的代码要为上一层数据准备 IP数据头，并且要决定如何把接收到的IP数据包传送到TCP协议层或者UDP协议层。在IP协议层的下方是支持整个 Linux 网络系统的网络设备，例如 PPP和以太网。网络设备并不完全等同于物理设备，因为一些网络设备，例如回馈设备是完全由软件实现的。和其他那些使用 mknod命令创建的Linux系统的标准设备不同，网络设备只有在软件检测到和初始化这些设备时才在系统中出现。当你构建系统内核时，即使系统中有相应的以太网设备驱动程序，你也只能看到 /dev/eth0。ARP协议在IP协议

层和支持ARP翻译地址的协议之间。

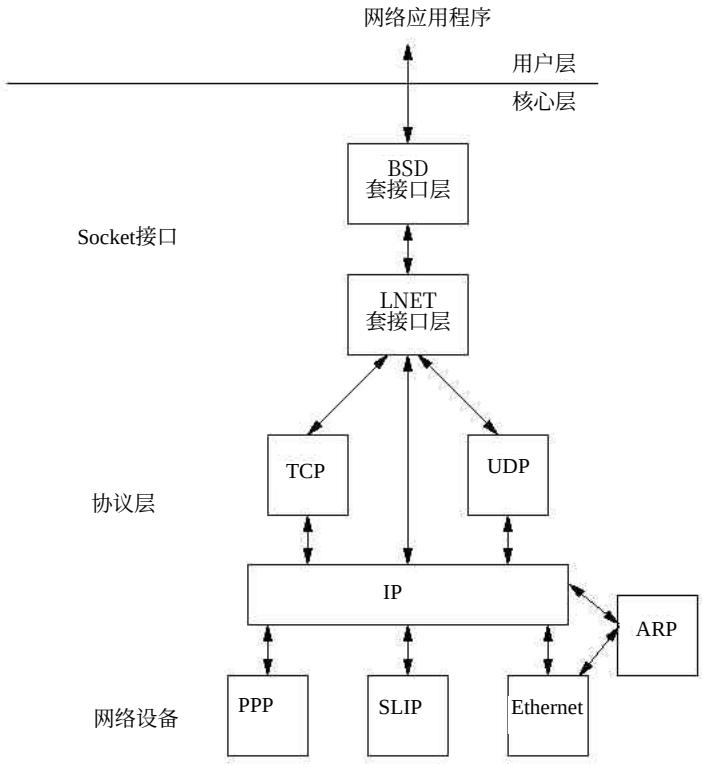

图16-2 TCP/ IP结构分层示意图

16.3 BSD套接口

BSD是UNIX系统中通用的网络接口，它不仅支持各种不同的网络类型，而且也是一种内部进程之间的通信机制。两个通信进程都用一个套接口来描述通信链路的两端。套接口可以认为是一种特殊的管道，但和管道不同的是，套接口对于可以容纳的数据的大小没有限制。Linux支持多种类型的套接口，也叫做套接口寻址族，这是因为每种类型的套接口都有自己的寻址方法。Linux支持以下的套接口类型：

UNIX	UNIX域套接口
INET	Internet地址族TCP/IP协议支持通信。
AX25	Amateur radio X25
IPX	Novell IPX
APPLETALK	Appletalk DDP
X25	X25

这些类型的套接口代表各种不同的连接服务。

Linux的BSD 套接口支持下面的几种套接口类型：

1. 流式（stream）

这些套接口提供了可靠的双向顺序数据流连接。它们可以保证数据传输中的完整性、正确

性和单一性。INET寻址族中的TCP协议支持这种类型的套接口。

2. 数据报（Datagram）

这种类型的套接口也可以像流式套接口一样提供双向的数据传输，但它们不能保证传输的数据一定能够到达目的节点。即使数据能够到达，也无法保证数据以正确的顺序到达以及数据的单一性、正确性。UDP协议支持这种类型的套接口。

3. 原始（Raw）

这种类型的套接口允许进程直接存取下层的协议。

4. 可靠递送消息（Reliable Delivered Messages）

这种套接口和数据报套接口一样，只能保证数据的到达。

5. 顺序数据包（Sequenced Packets）

这种套接口和流式套接口相同，除了数据包的大小是固定的。

6. 数据包（Packet）

这不是标准的BSD 套接口类型，而是Linux 中的一种扩展。它允许进程直接存取设备层的数据包。

利用套接口进行通信的进程使用的是客户机 /服务器模式。服务器用来提供服务，而客户机可以使用服务器提供的服务，就像一个提供 web页服务的Web服务器和一个读取并浏览 web页的浏览器。服务器首先创建一个套接口，然后给它指定一个名字。名字的形式取决于套接口的地址族，事实上也就是服务器的当地地址。系统使用数据结构 sockaddr来指定套接口的名字和地址。一个INET 套接口可以包括一个IP端口地址。你可以在/etc/services中查看已经注册的端口号，例如，一个web页面服务器的端口号是80。在服务器指定套接口的地址以后，它将监听和此地址有关的连接请求。请求的发起者，也就是客户机，将会创建一个套接口，然后再创建连接请求，并指定服务器的目的地址。对于一个 INET 套接口来说，服务器的地址就是它的IP地址和端口号。这些连接请求必须通过各种协议层，然后等待服务器的监听套接口。一旦服务器接收到了连接请求，它将接受或者拒绝这个请求。如果服务器接受了连接请求，它将创建一个新的套接口。一旦服务器使用一个套接口来监听连接请求，它就不能使用同样的套接口来支持连接。当连接建立起来以后，连接的两端都可以发送和接收数据。最后，当不再需要此连接时，可以关闭此连接。

使用BSD套接口的确切含义在于套接口所使用的地址族。设置一个 TCP/IP连接就和设置一个业余无线电X.25连接有很大的不同。和VFS一样，Linux从BSD 套接口协议层中抽象出了套接口界面，此界面负责和各种不同的应用程序之间进行通信。内核初始化时，内核中的各个不同的地址族将会在BSD 套接口界面中登记。稍后当应用程序创建和使用 BSD 套接口时，就将会在BSD 套接口和它支持的地址族之间建立一个连接。此连接是通过交叉关联的数据结构和地址族表建立的。例如，当一个应用程序创建一个新的套接口时，将产生一个可以被 BSD 套接口使用的与特定的地址族有关的套接口创建子过程。

设置系统内核时，一系列的地址族和协议将会保存在协议向量中。每一个协议都由它的名字代表，例如，INET和其初始化进程的地址。当系统启动并初始化套接口界面时，将会调用每一个协议的初始化进程。对于套接口地址族来说，这意味着它们注册的一系列有关协议操作。这是一系列的子程序，每一个都执行一个和特定的地址族有关的操作。已经注册的和协议相关的操作保存在pops向量中，而此向量由一系列指向数据结构proto_ops的指针组成。

数据结构proto_ops包括地址族的类型以及指向与特定地址族有关的套接口操作程序的指针。Pops向量用地址族标识符作为索引。

16.4 INET套接口层

INET 套接口层包括支持TCP/IP协议的Internet地址族。正如上面提到的，这些协议是分层的，每一个协议都使用另一个协议的服务。 Linux系统中的TCP/IP代码和数据结构也反映了这种分层的思想。它和BSD 套接口层的接口是通过一系列与 Internet地址族有关的套接口操作来实现的，而这些套接口操作是在网络初始化的过程中由 INET 套接口层在BSD 套接口层中注册的。这些操作和其他地址族的操作一样保存在 pops向量中。 BSD 套接口层通过 INET的proto_ops数据结构来调用与INET 层有关的套接口子程序来实现有关INET层的服务。例如，当BSD 套接口创建一个发送给INET地址族的请求时将会使用INET的套接口创建功能。BSD 套接口层将会把套接口数据结构传递给每一个操作中的 INET层。INET 套接口层在它自己的数据结构sock中而不是在BSD 套接口的数据结构中插入有关TCP/IP的信息，但sock数据结构是和BSD套接口的数据结构有关的，通过图16 - 3可以看出这种相关关系。它使用 BSD 套接口中的数据指针来连接sock数据结构和BSD 套接口数据结构，这意味着以后的 INET 套接口调用可以十分方便地得到sock数据结构。数据结构sock中的协议操作指针也会在创建时设置好，并且此指针是和所需要的协议有关的。如果需要的是 TCP协议，那么数据结构sock中的协议操作指针将会

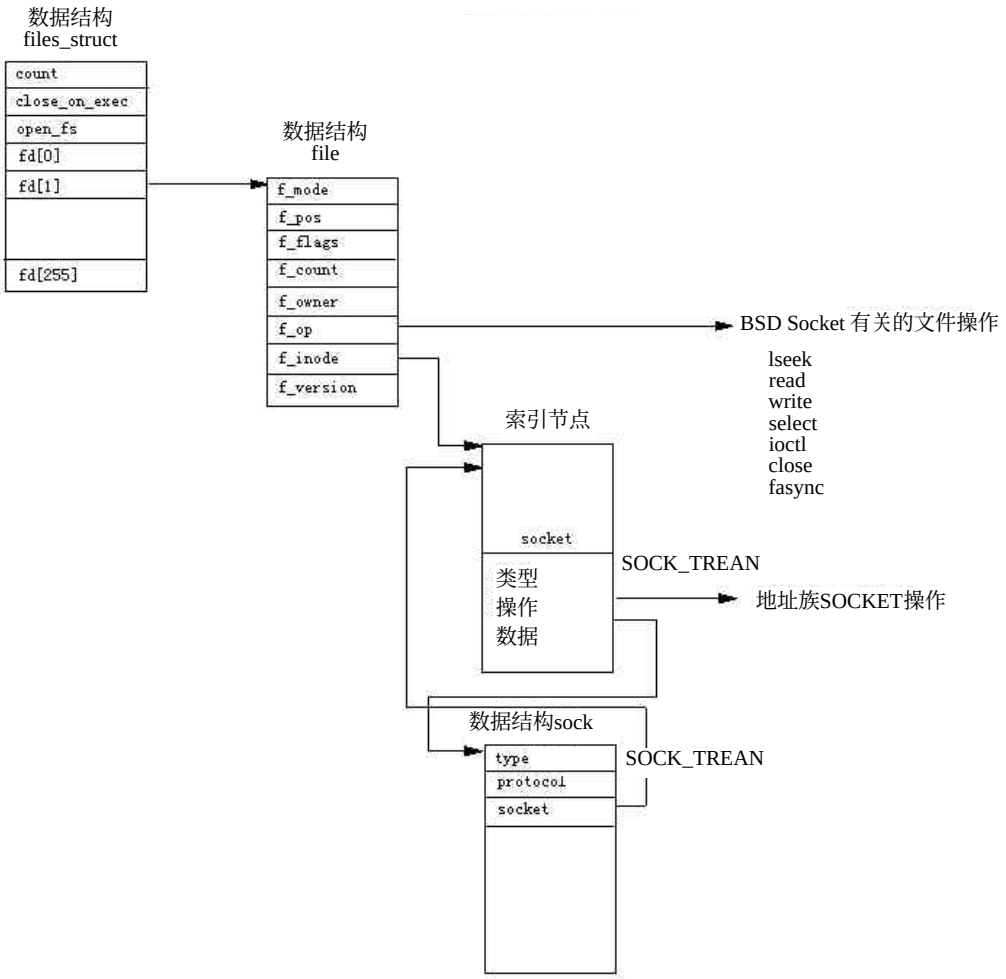

图16-3 Socket 数据结构示意图

指向一系列的TCP协议操作。

16.4.1 创建BSD 套接口

创建一个新套接口的系统调用将传递有关地址族、套接口类型和协议的标识符。

首先，系统调用使用需要的地址族来搜索pops向量以便找到一个匹配的地址族。也许某一个地址族是作为系统内核模块实现的，这时kerneld守护进程将会调入此模块以便继续运行。实际上，套接口数据结构物理上是VFS索引节点数据结构的一部分，所以分配一个套接口就等于分配一个VFS索引节点，这样就可以和使用普通文件一样来操作套接口。因为所有的文件都有一个VFS索引节点，故为了和文件操作之间的相互兼容，BSD 套接口也有一个代表它的VFS索引节点。

新创建的BSD 套接口数据结构包括一个指向特定地址族的套接口子过程的指针，它被设置为指向数据结构proto_ops,该数据结构从pops向量中获取。BSD 套接口数据结构的类型为SOCK_STREAM、SOCK_DGRAM或其他的套接口类型。与特定的地址族相关的创建进程通过保存在数据结构proto_ops中的地址来调用。

系统将会从当前进程的fd向量中分配一个空闲的文件描述符，同时也将初始化它所指向的文件的数据结构。这包括将文件操作指针设置到 BSD 套接口支持的文件操作中。任何以后的操作都将使用套接口，同时套接口将会通过调用相应的地址族操作子程序把这些操作传递到所支持的地址族中。

16.4.2 给INET BSD 套接口指定地址

为了能够监听到网络中的连接请求，每一个服务器都必须创建一个 INET BSD 套接口，并且指定一个地址。指定地址的操作一般都在 INET 套接口层中处理，但可能需要TCP层和UDP层的支持。已经指定了地址的套接口不能用于其他任何的通信，这意味着套接口的状态必须是TCP_CLOSE。传递给指定地址操作的sockaddr包括指定的IP地址和一个可选的端口号。一般情况下，指定的IP地址将会和指定给网络设备的IP地址相同，并且此网络设备支持INET地址族。你可以使用ifconfig命令查看系统中当前活动的网络接口。IP地址也可以是全为1或者全为0的IP广播地址。如果机器是作为一个透明的代理服务器或者防火墙，那么它就可以指定为任意的 IP地址，但只有具有超级用户优先权的进程可以指定它的 IP地址。指定的IP地址保存在数据结构sock中的recv_addr 和 saddr字段中，其中一个用于散列表的查找，另一个用于发送 IP地址。端口号是可选的，并且如果没有指定端口号的话，系统将要求网络选择一个空闲的端口号。按照惯例，没有超级用户权限的进程不能使用小于1024的端口号。如果网络确实分配了一个端口号，那么通常是大于1024的。

当系统中的网络设备接收到数据包时，这些数据包必须传送到正确的INET 和 BSD 套接口，以便进行下一步处理。正因为这样，UDP层和 TCP层提供一个用于查找接收的IP信息的地址的散列表，并且将数据包发送到正确的 socket/sock对中。TCP是以连接为导向的协议，所以在处理TCP数据包时要比处理UDP数据包涉及更多的信息。

UDP层中也有一个保存已分配的UDP端口的散列表 —— udp_散列表，表中包含指向sock数据结构的指针，并按照端口号作为索引。因为 UDP的散列表要比可以使用的端口数目少很多，所以散列表中的一些入口指向由多个数据结构sock组成的链表。

TCP层的情况较为复杂，因为它提供了几个散列表。尽管如此，TCP层在指定地址操作时也并不将数据结构sock添加到它的散列表中，而只是仅仅检查要求的端口号是否已经被占用。

数据结构sock反在监听操作时才被添加到TCP的散列表中。

16.4.3 在INET BSD套接口上创建连接

系统创建了一个套接口以后，如果此套接口没有用于监听进入的连接请求，那么它就可以用于向外发送连接请求。对于无连接的协议，例如 UDP，这种套接口操作并无太大的作用。但对于已连接为导向的协议，例如 TCP，套接口操作将在两个应用程序之间建立一个虚拟的链路。

INET BSD 套接口只有处于正确的状态时才能用于建立向外的连接，也就是说，此套接口没有用于连接，也没有用于监听进入的连接。这意味着 BSD 套接口数据结构必须处于SS_UNCONNECTED状态。UDP协议不需要在两个应用程序之间建立虚拟连接，在此协议中信息是以数据报的形式传送的。但它同样支持连接的 BSD 套接口操作。UDP INET BSD 套接口上的连接操作仅仅只是设置远程应用程序的地址，即它的 IP地址和IP端口地址。另外，它还建立一个路由表入口缓冲区，这样发送到此 BSD 套接口的UDP数据包则无须再次检查路由数据库。INET sock数据结构中的ip_route_cache指针指向缓冲区的路由信息。如果没有指定地址信息，此缓冲区路由和IP地址信息将会自动用于使用此BSD 套接口发送的信息，同时UDP的状态变为TCP_ESTABLISHED。

对于一个在TCP BSD 套接口上的连接操作，TCP必须创建一个包含连接信息的TCP信息块，并把它发送到指定的IP目的地址。TCP信息块包含关于连接的信息，包括一个唯一的起始信息序列号、信息的最大长度以及传送和接收的窗口大小等等。在 TCP协议中，所有的信息都是按顺序排好的，其中的起始序列号用于识别第一个信息块。 Linux系统会选择一个合理的随机值来避免恶意的协议的攻击。从TCP连接的一端传送的信息要经过连接的另一端的确认，这样才能保证信息传送的正确性，没有经过确认的信息将重发。发送和接收窗口的大小等于没有发送确认信息之前的信息个数。信息的最大长度是由用于初始化端的网络设备决定的。如果接收端的网络设备支持更小的信息长度，那么它们之间的连接将会使用两者之间较小的数值。创建向外的TCP连接请求的应用程序必须等待目标程序接收或者拒绝连接请求的回应。因为 TCP sock处于等待信息进入的状态，它将会添加到 tcp_listening_散列表中以便进入的TCP信息可以直接发送到此sock 数据结构中。TCP协议也将启动一个计时器，如果目标程序没有回应连接请求的话，向外的连接请求将会自动失效。

16.4.4 监听INET BSD 套接口

给套接口指定了地址以后，它将监听此指定地址的进入的连接请求。网络应用程序可以使用某一个套接口而不必首先指定一个地址，在这种情况下， INET 套接口层将会自动为套接口指定一个空闲的端口号。监听功能使得套接口进入到 TCP_LISTEN状态，同时做一些与特定的网络有关的允许进入连接的工作。

对于UDP 套接口来说，只须改变它的状态就已经足够了。但 TCP需要把套接口的sock 数据结构添加到tcp_bound_hash 和 tcp_listening_散列表中。

对于一个处于活动状态的正在监听的套接口来说，每当接收一个进入的 TCP连接请求，TCP都将建立一个新的 sock数据结构来代表此连接请求。它还将复制进入的包含连接信息的sk_buff，并将其插入到receive_queue队列中。在复制的sk_buff中包含一个指向新创建的sock数据结构的指针。

16.4.5 接收连接请求

UDP并不支持连接的概念，INET 套接口连接请求只适用于TCP协议。接收操作将会被传送到所支持的协议层，在这里是INET接收任何到来的连接请求。如果INET协议层下面的协议不支持连接，例如UDP，那么INET协议层将会放弃接收操作。否则，接收操作将会传送到真正的协议中，在这里为TCP协议。接收操作可以是阻塞的和非阻塞的。在非阻塞的情况下，如果没有到来的连接请求，接收操作将会失败，同时新创建的套接口数据结构也会被放弃。在阻塞的情况下，执行接收操作的网络应用程序将会添加到等待队列中，然后挂起直到接收到 TCP连接请求。一旦接收到连接请求以后，包含请求的sk_buff将会被扔掉，同时sock数据结构将返回到INET 套接口层。新的套接口文件描述符（fd）返回到网络应用程序中，这样应用程序就可以在套接口操作中使用此文件描述符。

16.5 IP 层

16.5.1 套接口缓冲区

在网络环境中，每一层的协议都可以使用其下面一层的协议。但这也同时产生一个问题，那就是每一个协议都需要在传送时数据上添加协议头和协议尾，而在接收处理数据的时候移走协议头和协议尾。这样就使得在协议之间传递数据缓冲区变得十分的困难，因为每一个协议层都需要了解其特殊的协议头和协议尾的位置。一个解决办法就是在每一个协议层中复制缓冲区，但这样做的效率是非常低的。Linux使用套接口缓冲区或者sk_buffs来在协议层中间和网络设备中间传递数据。数据结构sk_buffs包括指针和长度字段，这样就允许每一个协议层通过标准的功能或者方法来处理应用程序的数据。

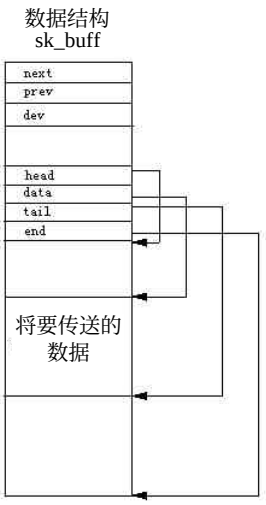

图16-4 SOCKET 缓冲区数据结构示意图

图16-4显示了数据结构sk_buff。每一个sk_buff都有一个数据块，并包括 4个数据指针，用来处理和管理套接口缓冲区的数据。

1. head（首部）

指向内存中数据区起始位置的指针。当sk_buff和它的数据块分配完毕以后，此指针就变为固定的值。

2. data（数据）

指向协议中数据当前起始位置的指针。它依据当前拥有sk_buff的协议层的不同而不同。

3. tail（尾部）

指向协议中数据当前末尾的指针。它同样依据当前拥有sk_buff的协议层的不同而不同。

4. end（结束）

指向内存中数据区的结束位置的指针。当sk_buff分配好了以后，此指针就为固定的值。

数据结构sk_buff中还包含两个长度字段，len 和 truesize，分别用于描述当前协议包的长度和整个数据缓冲区的长度。sk_buff的处理程序提供了标准机制，用于在应用程序的数据中添加

和移走协议头和协议尾。这些机制可以安全地处理数据结构 sk_buff中的数据、协议尾和长度字段：

1. push

此操作将data指针往数据区的起始位置移动，同时增加 len字段的值。往数据区的起始位置添加数据或协议头的时候可以使用它。

2. pull

此操作将 data 指针向数据区尾部的方向移动，同时减少 len字段的值。当把接收到的数据或协议头从数据的起始位置移走的时候可以使用它

3. put

此操作将tail指针向数据区的末端移动，同时增加 len字段的值。当往要传送的数据的末端添加数据或协议信息时可以使用它。

4. trim

此操作将tail指针向数据区的起始位置移动，同时减少 len字段的值。当从接收的数据包中移走数据或协议尾时可以使用它。

16.5.2 接收IP数据包

Linux系统内核初始化网络设备时，将产生一系列的 device数据结构，它们组成了dev_base链表。数据结构device 描述了其设备的状态并提供一系列可以调用的程序，在网络协议层需要网络设备工作时调用。这些程序大多涉及传送数据和网络设备的地址。当一个网络设备从网络中接收数据包时，它必须将接收到的数据转换成数据结构 sk_buff。这些sk_buff将会在接收时被网络设备添加到backlog队列中。如果backlog队列变得太大，那么系统将会扔掉一些接收到的sk_buff'。

当Linux网络层初始化时，每一个协议都将通过在 ptype_all链表或ptype_base 散列表中添加数据结构packet_type的方法登记。数据结构packet_type包含协议类型、一个指向网络设备的指针、一个指向接收数据处理过程的指针以及一个指向下一个 packet_type数据结构的指针。

16.5.3 发送IP数据包

数据产生时，将创建一个 sk_buff数据结构，它包含数据和协议层添加的各种协议头。sk_buff需要传送给网络设备以后才能发送。首先由协议，例如IP协议，决定要使用的网络设备，这取决于数据包的最佳路由。对于使用调制解调器，通过 PPP协议连接的单个网络来说，路由的选择相对简单。但对于以太网连接的多个计算机的网络，路由的选择则相对复杂。

对于每一个传送的IP数据包，IP协议使用路由表解决路由问题。在路由表中可以查找到的IP目的地址将会返回一个描述可以使用的路由 rtable数据结构，它包括一个源IP地址、device数据结构的地址以及一个预先创建的硬件头。这个硬件头与网络设备有关，并包含源和目的物理地址以及其他一些与特定介质有关的信息。如果网络设备是以太网设备，源和目的物理地址将会是以太网设备的物理地址。硬件头中也许包含需要使用 ARP协议解释的物理地址。硬件头将会保存在缓冲区中，以加快以后的查找速度。

16.5.4 数据碎片

每一个网络设备都有一个最大的数据包大小，它不能接收和发送大于此大小的数据包。 IP

协议遵守此规定，它将大的数据块分成较小的数据块，以便网络设备处理。IP协议头包括一个碎片字段、一个标志和一个碎片位移。

当一个IP数据包等待传送时，IP要找到发送数据的网络设备。IP通过路由表来决定使用哪一个设备发送IP数据包。每一个设备都包括一个描述设备可以传送的最大数据单位的字段，也就是mtu字段。如果设备的mtu字段比等待传送的IP数据包的大小的话，那么IP数据包就必须分割成mtu大小的碎片。每一个碎片都有一个sk_buff来代表，sk_buff的IP头可以表明这是一个IP数据包碎片，以及此碎片在IP数据包中的位移。如果在分割IP数据包的过程中，IP无法分配sk_buff,那么这个传送将会失败。

接收IP数据包碎片要比发送更为困难，因为接收端可以以任意的顺序接收到数据包碎片。而只有在接收了所有的数据包碎片以后才能将数据包碎片重新组装起来。每次接收一个IP数据包的时候，接收端都要查看此数据包是否为IP数据包碎片。第一次接收数据包碎片时，IP协议将创建一个新的ipq数据结构，并将此结构插入到ipqueue链表中。当接收到更多的IP数据包碎片时，接收端找到相应的ipq数据结构，同时创建一个新的数据结构ipfrag来描述此数据包碎片。每一个ipq数据结构都使用源和目的IP地址，上层协议的标识符以及此IP数据包的标识符用来唯一地描述接收的IP碎片。当接收到所有的碎片以后，它们组合成一个sk_buff数据结构，并将其传送到下一层的协议进一步处理。每一个ipq数据结构中都包含一个计时器，每当接收到一个有效的IP碎片以后，计时器都将重新工作。如果计时器失效，那么说明此IP数据包接收失败，上一层的协议将会负责重新传送此数据包。

16.6 地址解析协议

地址解析协议（ARP）负责将IP地址翻译成网络物理地址，例如以太网地址。IP协议在将数据发送到相应的网络设备前需要进行此种翻译。

ARP将会检查设备是否需要一个硬件头以及如果需要的话，此数据包的硬件头是否需要重新创建。Linux将硬件头保存在缓冲区中，以此避免频繁地重建硬件头。如果硬件头确实需要重建，ARP将调用和网络设备有关的硬件头重建子过程。所有的以太网设备都使用同样的硬件头重建子过程，而此子过程反过来使用ARP服务将目的IP地址翻译成物理地址。

ARP协议本身十分简单，它由两种类型的信息组成：ARP请求和ARP应答。ARP请求中包括需要翻译的IP地址，而ARP应答中包含的是已经翻译过的IP地址，也就是硬件地址。ARP请求将传送到网络中的所有主机中，所以对于一个以太网来说，连接在以太网上的所以机器都将接收到ARP请求。但只有包含此IP地址的计算机才作出应答，返回一个包含自己的物理地址的ARP应答。

在Linux中，ARP协议主要是一个由数据结构arp_table构成的表，表中的每一个数据结构都描述了IP地址到物理地址的翻译。arp_table入口在IP地址需要翻译时创建，并且在一段时间不用后将会从表中移走。数据结构arp_table中包含以下的字段：

last used	ARP入口最后被使用的时间。
last updated	ARP入口最新更新的时间。
flags	描述入口的状态。
IP address	入口描述的IP 地址。
hardware address	翻译以后的硬件地址 。
hardware header	指向硬件头的指针。
timer	ARP请求没有得到回应的失效的计时器。

retries	ARP请求重试的次数。
sk_buff queue	等待翻译的sk_buff 的入口。

当一个IP地址需要翻译时，通常在ARP表中并没有相应的arp_table入口，所以ARP将发送一个ARP请求信息，同时在ARP表中创建一个新的arp_table入口，并将需要地址翻译的sk_buff数据结构插入到新的入口的sk_buff queue等待队列中。在ARP发送出ARP请求以后，计时器就开始工作。如果在计时器失效以后还没有回应，那么 ARP将会重复在retries字段中设置的次数的请求。如果还是没有回应， ARP 将会把此arp_table入口从ARP表中移走。这时，等待翻译的队列中所有的sk_buff数据结构都将得到通知，同时 ARP告诉上一层的协议地址翻译请求失败。如果ARP请求得到了应答，那么arp_table入口就可以标记为complete，所有等待的sk_buff'都将从等待队列中移走，继续传送。翻译后的硬件地址将会保存在每一个 sk_buff的硬件头内。

经过一段时间，网络的拓扑结构可能会发生变化， IP地址也可能重新指定给其他的硬件地址。例如，拨号服务就是在连接时才指定 IP地址。为了保证ARP表保存最新的入口信息， ARP定时检查ARP表中的arp_table，以便移走已经失效的入口。有些入口是永久性的入口，它们不能被移走。

第17章　系统内核机制

本章介绍Linux系统内核提供的机制，以便系统内核中的各个部分可以有效地协作。

17.1　Bottom Half处理

有时在系统内核中，你也许不想处理任何任务。例如在一个中断处理过程中，当发生一个中断时，处理器停止正在处理的工作，同时操作系统将中断发送到相应的设备驱动程序。但设备驱动程序不能花费太多的时间来处理中断，因为在这期间系统将不能做任何的工作，所以一些工作可以以后再进行处理。bottom half处理程序正是用来实现此功能的。图17-1显示了内核中有关bottom half处理的数据结构。

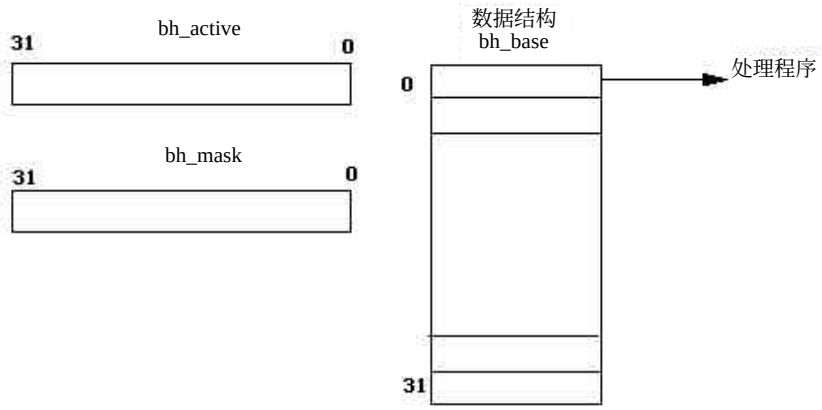

图17-1　Bottom Half 数据结构示意图

系统内核中可以有多达32个不同的bottom half处理程序。bh_base中保存着指向每一个bottom half处理程序的指针。bh_active和bh_mask根据安装和活动状态进行设置。如果bh_active的第N位置位，那么bh_base中的第N个元素将包含bottom half处理程序的地址。如果bh_active的第N位置位，那么第N个bottom half处理程序就可以在调度程序认为合适的时候调用它。一般情况下，bottom half处理程序都有一个和它们关联的任务列表。

一些内核的bottom half处理程序和设备有关，但以下几个却较为通用：

1.timer

当系统中的周期计时器中断时，此处理程序将标记为活动状态。它用来驱动内核中的计时器队列机制。

2. console

用于处理有关控制台的信息。

3. tqueue

用于处理有关tty 信息。

4. net

用于处理一般的网络问题。

5. immediate

这是一个通用的处理程序，可以被多个设备驱动同时使用。

当设备驱动程序或内核的其他部分需要把一些工作等待以后完成时，它将会将工作添加到相应的系统队列中，例如timer 队列，然后通知内核需要做bottom half处理。它通过将bh_active中适当的位置 1而通知内核。如果驱动程序将一些工作放入了 immediate队列中，它将会把bh_active的第8位置1，这样immediate bottom half处理程序就可以运行和处理它了。在每一次系统调用结束之前，系统内核都将会查看 bh_active。如果有任何的一位被置位，内核将会调用相应的bottom half处理程序。检查的顺序是先检查位0，然后是位1，以此类推直到位31。

17.2 任务队列

任务队列是系统内核将任务推迟到以后再做的方法。 Linux系统有一个机制可以把任务放入到队列中等待以后处理。

任务队列通常和bottom half处理程序一同使用，例如当timer bottom halfe处理程序运行时，系统将处理计时器任务队列。一个任务队列就是一个简单的数据结构，如图 17-2所示。它是一个由tq_struct数据结构组成的链表，每一个数据结构都包含处理程序的地址和指向相关数据的指针。

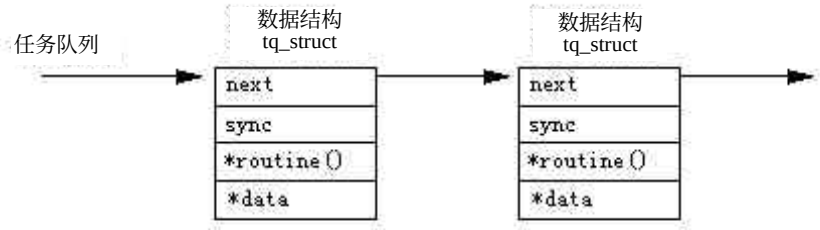

图17-2 任务队列示意图

当系统处理任务队列中的任务时，系统内核将调用处理程序，同时内核传递给处理程序一个指向数据的指针。

系统内核中的任何东西，例如设备驱动程序，都可以创建和使用任务队列，但系统内核只创建和管理三个任务队列：

1. timer

此队列中的任务将会在下一个系统时钟时执行。每次系统时钟开始的时候，系统内核都要检查队列中是否含有任何的入口，如果有的话，内核就将 timer 队列 bottom half处理程序设置为活动的。当调度进程下一次运行时， timer 队列 bottom half处理程序和其他的bottom half处理程序将会运行。

2. immediate

此队列bottom half处理程序的优先权要比timer 队列 bottom half处理程序的优先权低，所以将会在稍后运行。

3. scheduler

此队列由调度程序直接处理。它用来支持系统中其他的任务队列。

当系统内核处理任务队列时，队列中的第一个元素的指针将会从队列中移走，并代以空指针。事实上，这种移走队列中元素的操作是自动的，并且是不能中断的。然后系统内核将轮流调用队列中每一个元素的处理程序。

17.3 计时器

操作系统需要能够调度可能在将来发生的事件，所以系统中需要存在一种机制，使得事件在较为精确的时间调度运行。任何一个希望支持操作系统的处理器都应该有一个可编程的内部时钟，可以周期性地中断处理器。这就像一个节拍器一样可以协调系统中的各个任务。

Linux系统中有两种系统时钟，图17-3显示了这两种机制。

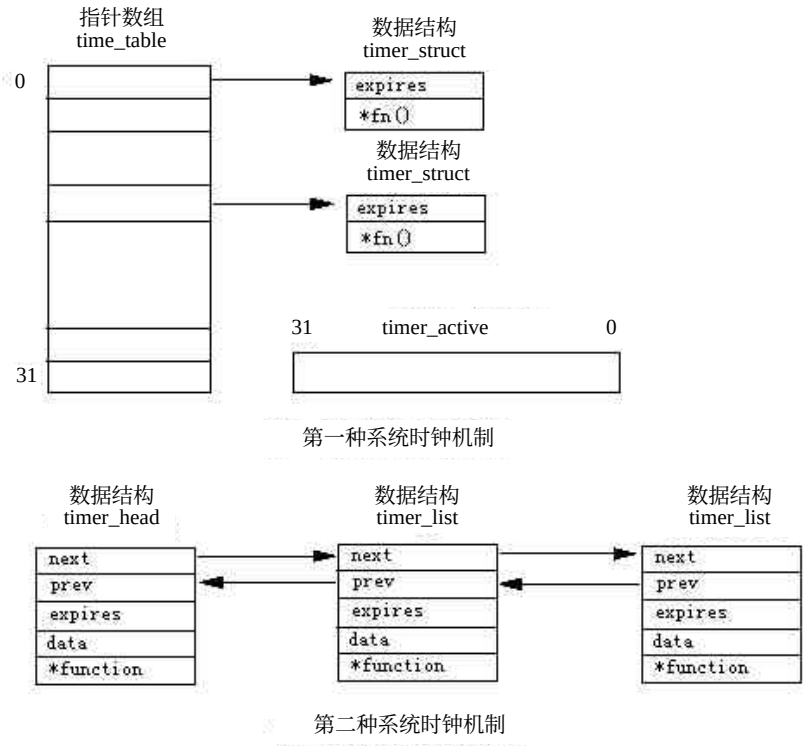

图17-3 系统时钟结构示意图

第一种机制是一种老的系统时钟机制，它是一个包括 32个指针的静态数组，每一个指针都指向一个timer_struct数据结构，并且还有一个timer_active来标识活动的系统时钟。

第二种系统时钟机制是一个timer_list数据结构的链表，它以失效的时限按升序排列。

17.4 等待队列

系统中经常发生进程需要等待系统资源的情况。例如，进程可能需要文件系统中描述目录的VFS 索引节点，但该索引节点并不在系统的缓冲区缓存中。在这种情况下，进程必须等到系统内核将索引节点从物理设备中读取到缓冲区缓存中才能继续运行。

Linux系统内核使用了一个简单的数据结构，它包括一个指向进程 task_struct的指针和一个指向队列中下一个元素的指针。

当进程添加到等待队列的末端时，它们或者是可以中断的，或者是不可以中断的。

当系统内核处理等待队列时，等待队列中的每一个进程都被设置成 RUNNING状态。当等待队列中的进程可以被调度运行时，系统内核所做的第一件事就是把进程从等待队列中移走。

等待队列可以用于同步系统资源的存取，Linux系统内核也常常使用它来实现信号量。

17.5 信号量

信号量用于保护系统中关键的代码或者数据结构。应当注意的是，每一次对关键数据，例如VFS索引节点的存取，都是由系统内核代表某一个进程来完成的。允许一个进程修改其他进程可能正在使用的数据是十分危险的。Linux系统中使用信号量技术使得某一时刻只有一个进程可以存取关键区域的代码和数据。其他希望存取此资源的进程只有等待直到此资源空闲为止。等待的进程将会被挂起，但系统中的其他进程可以正常地运行。

Linux系统中信号量的数据结构包括以下的信息：

1. count

记录希望使用此资源的进程个数。正值意味着此资源可用，负值或零值意味着进程正在等待此资源。其初始值为1，说明此时有一个且仅有一个进程可以使用此资源。当进程使用此资源时，它们将会减少此字段的值。当它们释放此资源时，将会增加此字段的值。

2. waking

等待此资源的进程个数，同时也表示当此资源空闲时，可以唤醒以便被执行的进程个数。

3. wait queue

当进程等待此资源时，它们将会被放入此等待队列。

4. lock

用于存取waking 字段时所用的锁。

假设信号量的初始值为1，第一个进程将会发现此计数器为正值并将其减1，也就是说此时计数器的值为0。现在此进程拥有了由信号量保护的关键代码或者数据。当进程放弃此资源时，它将会把信号量加1。最为理想的状态就是没有其他任何的进程竞争此资源。

当一个进程正在使用此资源时，如果其他进程也试图使用此资源，这时进程将把信号量减1。信号量现在变为了一个负值（-1），所以其他进程无法使用此资源，它只有等待使用资源的进程释放资源以后才能使用。系统内核使等待的进程进入到睡眠的状态，直到使用资源的进程退出时才唤醒它。等待的进程将自己插入到信号量的等待队列中。

第四篇　Linux系统高级编程

第18章　Linux内核模块编程

本章从一个简单的程序Hello World开始介绍Linux内核模块的编程。

18.1　一个简单程序Hello World

一个内核模块至少包括两个函数：一个是初始化函数 init_module，在模块被插入到内核时使用；另一个是清除函数 cleanup_module，在模块被从内核中移走时使用。一般情况下，函数 init_module或者在内核中注册一个处理程序，或者使用自己的代码替换一个内核函数。函数 cleanup_module的作用是清除init_module所作的任何事情，这样模块可以安全地卸载。

下面来看一个简单的程序。

```
/* hello.c */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* Initialize the module */
int init_module()
{
  printk("Hello, world - this is the kernel speaking\n");
  /* If we return a non zero value, it means that init_module failed and
   * the kernel module can't be loaded */
  return 0;
}
/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
 # Makefile for a basic kernel module
CC=gcc
MODCFLAGS := -O6 -Wall -DCONFIG_KERNELD -DMODULE -D__KERNEL__ -DLinux

hello.o:        hello.c /usr/include/linux/version.h
                $(CC) $(MODCFLAGS) -c hello.c
                echo insmod hello.o to turn it on
                echo rmmod hello to turn if off
                echo
                echo X and kernel programming do not mix.
                echo Do the insmod and rmmod from outside X.
```

现在，你可以登录到超级用户下，然后执行 insmod hello 和rmmod hello。你可以在/proc/modules中查看新的内核模块。

18.2 设备文件

内核模块包括两种方法和一个进程通信。一种方法是使用设备文件，例如目录 /dev下的文件，另一种方法是使用文件系统。因为编写内核模块的一个主要原因就是为了驱动某种硬件设备，所以我们从设备文件开始。

设备文件的主要目的是允许进程和内核中的设备驱动程序通信，从而和物理设备通信。以下实现的方法。

每一个负责某硬件设备的设备驱动程序都有一个主设备号，设备驱动程序表和对应的主设备号可以在/proc/devices中查看到。每一个设备驱动程序控制的物理设备有一个从设备号。目录/dev下为每一个设备都对应了一个特殊的文件，叫做设备文件，而不管此设备是否安装到了系统中。

例如，如果你执行ls -l /dev/hd[ab]*，你将会看到机器中所有的 IDE硬盘的分区。请注意，这些硬盘分区的主设备号都是3，但从设备号却各不相同。

系统安装时，所有这些设备文件都是使用 mknod命令创建的。没有什么特别的原因将设备文件放在/dev目录下，这只是一种习惯。

设备可以分为两种类型：字符设备和块设备。通过使用 ls -1命令可以分辨设备是字符设备还是块设备。如果此命令输出的第一个字符是 b，那么它是一个块设备；如果第一个字符是 c，那么它是一个字符设备。

内核模块分为两个部分：用于注册设备的模块部分和设备驱动程序部分。函数 init_module通过调用module_register_chrdev来把设备驱动程序添加到内核的字符设备驱动表中，同时返回驱动程序可以使用的主设备号。函数cleanup_module用于清除注册的设备。

设备驱动程序由4个device_<action>函数组成。当某一个进程需要拥有这个主设备号的设备文件作某些工作时将会调用这些函数。内核通过数据结构 Fops可以了解调用它们的方法。该数据结构是在设备注册时给定的，它包含指向这 4个函数的指针。

下面是chardev.c的源程序，该程序用来创建一个字符设备文件。

```
/* chardev.c
 * Create a character device (read only)
 */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* For character devices */
#include <linux/fs.h>       /* The character device definitions are here */
#include <linux/wrapper.h>  /* A wrapper which does next to nothing at
                             * present, but may help for compatibility
                             * with future versions of Linux */
#define SUCCESS 0
/* Device Declarations ********************************************** */
/* The name for our device, as it will appear in /proc/devices */
#define DEVICE_NAME "char_dev"
/* The maximum length of the message from the device */
```

```
#define BUF_LEN 80
/* Is the device open right now? Used to prevent concurent access into
 * the same device */
static int Device_Open = 0;
/* The message the device will give when asked */
static char Message[BUF_LEN];
/* How far did the process reading the message get? Useful if the
 * message is larger than the size of the buffer we get to fill in
 * device_read. */
static char *Message_Ptr;
/* This function is called whenever a process attempts to open the device
 * file */
static int device_open(struct inode *inode, struct file *file)
{
  static int counter = 0;
#ifdef DEBUG
  printk ("device_open(%p,%p)\n", inode, file);
#endif
  /* We don't want to talk to two processes at the same time */
  if (Device_Open)
    return -EBUSY;
  /* If this was a process, we would have had to be more careful here.
   *
   * In the case of processes, the danger would be that one process
   * might have check Device_Open and then be replaced by the schedualer
   * by another process which runs this function. Then, when the first process
   * was back on the CPU, it would assume the device is still not open.
   * However, Linux guarantees that a process won't be replaced while it is
   * running in kernel context.
   *
   * In the case of SMP, one CPU might increment Device_Open while another
   * CPU is here, right after the check. However, in version 2.0 of the
   * kernel this is not a problem because there's a lock to guarantee
   * only one CPU will be kernel module at the same time. This is bad in
   * terms of performance, so it will probably be changed in the future,
   * but in a safe way.
   */
  Device_Open++;
  /* Initialize the message. */
  sprintf(Message,
    "If I told you once, I told you %d times - Hello, world\n",
    counter++);
  /* The only reason we're allowed to do this sprintf, is because the
   * maximum length of the message (assuming 32 bit integers - up to 10 digits
   * with the minus sign) is less than BUF_LEN, which is 80. BE CAREFUL NOT TO
   * OVERFLOW BUFFERS, ESPECIALLY IN THE KERNEL!!!
   */
  Message_Ptr = Message;
  /* Make sure that the module isn't removed while the file is open by
   * incrementing the usage count (the number of opened references to the
   * module, if it's not zero rmmod will fail)
```

```
   */
  MOD_INC_USE_COUNT;
  return SUCCESS;
}
/* This function is called when a process closes the device file. It
 * is not allowed to fail */
static void device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
  printk ("device_release(%p,%p)\n", inode, file);
#endif
  /* We're now ready for our next caller */
  Device_Open --;
  /* Decrement the usage count, otherwise once you opened the file you'll
   * never get rid of the module.
   */
  MOD_DEC_USE_COUNT;
}
/* This function is called whenever a process which already opened the
 * device file attempts to read from it. */
static int device_read(struct inode *inode,
                       struct file *file,
                       char *buffer,   /* The buffer to fill with the data */
                       int length)     /* The length of the buffer
                                  * (mustn't write beyond that!) */
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;
#ifdef DEBUG
  printk("device_read(%p,%p,%p,%d)\n",
    inode, file, buffer, length);
#endif
  /* If we're at the end of the message, return 0 (which signifies end
   * of file) */
  if (*Message_Ptr == 0)
    return 0;
  /* Actually put the data into the buffer */
  while (length && *Message_Ptr)  {
    /* Because the buffer is in the user data segment, not the kernel
     * data segment, assignment wouldn't work. Instead, we have to use
     * put_user which copies data from the kernel data segment to the user
     * data segment. */
    put_user(*(Message_Ptr++), buffer++);
    length --;
    bytes_read ++;
  }
#ifdef DEBUG
   printk ("Read %d bytes, %d left\n",
     bytes_read, length);
#endif
   /* Read functions are supposed to return the number of bytes actually
```

```
    * inserted into the buffer */
  return bytes_read;
}
/* This function is called when somebody tries to write into our device
 * file - currently unsupported */
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
{
#ifdef DEBUG
  printk ("device_write(%p,%p,%s,%d)",
    inode, file, buffer, length);
#endif
  return -EINVAL;
}
/* Module Declarations ********************************************** */
/* The major device number for the device. This is static because it
 * has to be accessible both for registration and for release. */
static int Major;
/* This structure will hold the functions to be called when
 * a process does something to the device we created. Since a pointer to
 * this structure is kept in the devices table, it can't be local to
 * init_module. NULL is for unimplemented functions. */
struct file_operations Fops = {
  NULL,   /* seek */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* select */
  NULL,   /* ioctl */
  NULL,   /* mmap */
  device_open,
  device_release  /* a.k.a. close */
};
/* Initialize the module - Register the character device */
int init_module()
{
  /* Register the character device (atleast try) */
  Major = module_register_chrdev(0,
                                 DEVICE_NAME,
                                 &Fops);
  /* Negative values signify an error */
  if (Major < 0) {
    printk ("Sorry, registering the character device failed with %d\n",
      Major);
    return Major;
  }
  printk ("Registeration is a success. The major device number is %d.\n",
    Major);
  printk ("If you want to talk to the device driver, you'll have to\n");
```

```
  printk ("create a device file. We suggest you use:\n");
  printk ("mknod <name> c %d 0\n", Major);
  return 0;
}
/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
  int ret;
  /* Unregister the device */
  ret = module_unregister_chrdev(Major, DEVICE_NAME);
  /* If there's an error, report it */
  if (ret < 0)
    printk("Error in module_unregister_chrdev: %d\n", ret);
}
```

18.3 /proc文件系统

在Linux系统中，系统内核和内核模块还有另一种向进程发送信息的方法 —— /proc文件系统。最初设计/proc文件系统的目的是为了方便地存取进程中的信息，而现在内核中所有希望发送信息的，例如保存模块表的/proc/modules和保存内存使用统计的/proc/meminfo都在使用/proc文件系统。

使用/proc文件系统的方法和使用设备驱动程序的方法十分相似。首先，创建一个包括/proc文件系统所需要的信息的数据结构，该数据结构包括指向处理程序的指针。然后，使用init_module在内核中注册此数据结构，要清除这些注册则使用 cleanup_module。

使用proc_register_dynamic的原因是我们并不希望事先决定文件的索引节点，而是允许内核来决定以避免冲突。

下面是程序procfs.c的源代码：

```
/* procfs.c -  create a "file" in /proc
 */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* Necessary because we use the proc fs */
#include <linux/proc_fs.h>
/* Put data into the proc fs file.
  Arguments
  =========
  1. The buffer where the data is to be inserted, if you decide to use
    it.
  2. A pointer to a pointer to characters. This is useful if you don't
    want to use the buffer allocated by the kernel.
  3. The current position in the file.
  4. The size of the buffer in the first argument.
  5. Zero (for future use?).
  Usage and Return Value
  ======================
  If you use your own buffer, like I do, put its location in the
  second argument and return the number of bytes used in the buffer.
```

```
   A return value of zero means you have no further information at this
   time (end of file). A negative return value is an error condition.
   For More Information
   ====================
   The way I discovered what to do with this function wasn't by reading
   documentation, but by reading the code which used it. I just looked to
   see what is using the get_info field of proc_dir_entry's (I used a
   combination of find and grep, if you're interested), and I saw that
   it is used in <kernel source directory>/fs/proc/array.c.
   If something is unknown about the kernel, this is usually the way
   to go. In Linux we have the great advantage of having the kernel source
   code for free - use it.
 */
int procfile_read(char *buffer, char **buffer_location, off_t offset,
                  int buffer_length, int zero)
{
  int len;  /* The number of bytes actually used */
  /* This is static so it will still be in memory when we leave this
   * function */
  static char my_buffer[80];
  static int count = 1;
  /* We give all of our information in one go, so if the user asks us
   * if we have more information the answer should always be no.
   *
   * This is important because the standard read function from the library
   * would continue to issue the read system call until the kernel replies
   * that it has no more information, or until its buffer is filled.
   */
  if (offset > 0)
    return 0;
  /* Fill the buffer and get its length */
  len = sprintf(my_buffer, "For the %d%s time, go away!\n", count,
    (count % 100 > 10 && count % 100 < 14) ? "th" :
      (count % 10 == 1) ? "st" :
        (count % 10 == 2) ? "nd" :
          (count % 10 == 3) ? "rd" : "th" );
  count++;
  /* Tell the function which called us where the buffer is */
  *buffer_location = my_buffer;
  /* Return the length */
  return len;
}
struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register_dynamic */
    4, /* Length of the file name */
    "test", /* The file name */
    S_IFREG | S_IRUGO, /* File mode - this is a regular file which can
                        * be read by its owner, its group, and everybody
                        * else */
```

```
    1,  /* Number of links (directories where the file is referenced) */
    0, 0,  /* The uid and gid for the file - we give it to root */
    80, /* The size of the file reported by ls. */
    NULL, /* functions which can be done on the inode (linking, removing,
          * etc.) - we don't support any. */
    procfile_read, /* The read function for this file, the function called
                    * when somebody tries to read something from it. */
    NULL /* We could have here a function to fill the file's inode, to
          * enable us to play with permissions, ownership, etc. */
  };

/* Initialize the module - register the proc file */
int init_module()
{
  /* Success if proc_register_dynamic is a success, failure otherwise */
  return proc_register_dynamic(&proc_root, &Our_Proc_File);

  /* proc_root is the root directory for the proc fs (/proc). This is
   * where we want our file to be located.
   */
}
/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

18.4 使用/proc输入

现在我们已经有了两种方法从内核模块中产生输出。一种方法是注册一个设备驱动程序，并且使用mknod创建一个设备文件，另一种方法是创建一个/proc文件。这就允许内核模块告诉我们它可以了解的任何事情。但目前为止我们却无法回应内核模块。向内核模块中发送信息的第一种方法是写入到/proc文件中。

因为/proc文件系统的主要作用是允许内核向进程报告它的状态，所以没有专门为输入设置的机制。数据结构proc_dir_entry中并不包括指向输入函数的指针，就像它包括指向输出函数的指针一样。为了能够写入到一个/proc文件中，需要使用标准的文件系统机制。

在Linux系统中有一个标准的文件系统注册机制。因为每一个文件系统都有处理索引节点和文件操作的函数，所以有一个特殊的数据结构来保存指向这些函数的指针，也就是数据结构inode_operations。在/proc中，每当注册一个新文件时，都允许指定使用哪一个数据结构inode_operations来存取它，这就是使用的机制，一个数据结构inode_operations中包括指向数据结构file_operations的指针，同时数据结构 file_operations中包括指向 module_input 和module_output函数的指针。

应该注意的是，在内核中读和写的角色是互换的。读用于输出，而写用于输入，这是因为所说的读写是以用户的角度来说的。如果进程需要从内核中读取信息，内核就需要输出信息。如果进程需要向内核中写入信息，那么内核将接收此信息作为输入。

下面我们看看module_permission函数。每当一个进程试图使用/proc文件时都会调用此函数。它可以决定是否允许对文件的存取。

使用put_user 和 get_user的原因是Linux系统中的内存是分段的。这意味着指针不能指向内存中一个唯一的地址，只能指向内存段中唯一的地址。所以需要知道使用的内存段。系统中有一个内核使用的内存段，而每一个进程还有一个单独的内存段。

进程只能够存取自己的内存段，所以运行进程时没有必要担心内存段。当你编写一个内存模块时，一般情况下你希望存取内核内存段，这是由系统自动处理的。尽管如此，当需要将一个内存缓冲区中的内容在当前进程和内核中传递时，内核函数接收到一个指针，指向进程内存段中的内存缓冲区。宏put_user 和 get_user macros允许你存取该内存。

下面是procfs.c的源代码：

```
/* procfs.c -  create a "file" in /proc, which allows both input and
 * output. */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* Necessary because we use proc fs */
#include <linux/proc_fs.h>
/* The module's file functions ********************************** */
/* Here we keep the last message received, to prove that we can process
 * our input */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];
/* Since we use the file operations struct, we can't use the special proc
 * output provisions - we have to use a standard read function, which is
 * this function */
static int module_output(struct inode *inode, /* The inode read */
                         struct file *file,   /* The file read */
                         char *buf, /* The buffer to put data to (in the
                                       * user segment) */
                         int len)  /* The length of the buffer */
{
  static int finished = 0;
  int i;
  char message[MESSAGE_LENGTH+30];
  /* We return 0 to indicate end of file, that we have no more information.
   * Otherwise, processes will continue to read from us in an endless loop. */
  if (finished) {
    finished = 0;
    return 0;
  }
  /* We use put_user to copy the string from the kernel's memory segment
   * to the memory segment of the process that called us. get_user, BTW, is
   * used for the reverse. */
  sprintf(message, "Last input:%s", Message);
  for(i=0; i<len && message[i]; i++)
    put_user(message[i], buf+i);
  /* Notice, we assume here that the size of the message is below len, or
   * it will be received cut. In a real life situation, if the size of the
   * message is less than len then we'd return len and on the second call
   * start filling the buffer with the len+1'th byte of the message. */
```

```
  finished = 1;
  return i;  /* Return the number of bytes "read" */
}
/* This function receives input from the user when the user writes to
 * the /proc file. */
static int module_input(struct inode *inode, /* The file's inode */
                        struct file *file,   /* The file itself */
                        const char *buf,     /* The buffer with the input */
                        int length)          /* The buffer's length */
{
  int i;
  /* Put the input into Message, where module_output will later be
   * able to use it */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
    Message[i] = get_user(buf+i);
  Message[i] = '\0';  /* we want a standard, zero terminated string */
  /* We need to return the number of input characters used */
  return i;
}
/* This function decides whether to allow an operation (return zero) or
 * not allow it (return a non-zero which indicates why it is not allowed).
 *
 * The operation can be one of the following values:
 * 0 - Execute (run the "file" - meaningless in our case)
 * 2 - Write (input to the kernel module)
 * 4 - Read (output from the kernel module)
 *
 * This is the real function that checks file permissions. The permissions
 * returned by ls -l are for referece only, and can be overridden here.
 */
static int module_permission(struct inode *inode, int op)
{
  /* We allow everybody to read from our module, but only root (uid 0)
   * may write to it */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;
  /* If it's anything else, access is denied */
  return -EACCES;
}
/* The file is opened - we don't really care about that, but it does mean
 * we need to increment the module's reference count. */
int module_open(struct inode *inode, struct file *file)
{
  MOD_INC_USE_COUNT;
  return 0;
}
/* The file is closed - again, interesting only because of the reference
 * count. */
void module_close(struct inode *inode, struct file *file)
```

```
{
  MOD_DEC_USE_COUNT;
}
/* Structures to register as the /proc file, with pointers to all the
 * relevant functions. ********************************************** */
/* File operations for our proc file. This is where we place pointers
 * to all the functions called when somebody tries to do something to
 * our file. NULL means we don't want to deal with something. */
static struct file_operations File_Ops_4_Our_Proc_File =
 {
   NULL,  /* lseek */
   module_output,  /* "read" from the file */
   module_input,   /* "write" to the file */
   NULL,  /* readdir */
   NULL,  /* select */
   NULL,  /* ioctl */
   NULL,  /* mmap */
   module_open,    /* Somebody opened the file */
   module_close    /* Somebody closed the file */
   /* etc. etc. etc. (they are all given in /usr/include/linux/fs.h).
    * Since we don't put anything here, the system will keep the default
    * data, which in UNIX is zeros (NULLs when taken as pointers). */
 };
/* Inode operations for our proc file. We need it so we'll have some
 * place to specify the file operations structure we want to use, and
 * the function we use for permissions. It's also possible to specify
 * functions to be called for anything else which could be done to an
 * inode (although we don't bother, we just put NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
 {
   &File_Ops_4_Our_Proc_File,
   NULL, /* create */
   NULL, /* lookup */
   NULL, /* link */
   NULL, /* unlink */
   NULL, /* symlink */
   NULL, /* mkdir */
   NULL, /* rmdir */
   NULL, /* mknod */
   NULL, /* rename */
   NULL, /* readlink */
   NULL, /* follow_link */
   NULL, /* readpage */
   NULL, /* writepage */
   NULL, /* bmap */
   NULL, /* truncate */
   module_permission /* check for permissions */
 };
/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
 {
```

```
    0, /* Inode number - ignore, it will be filled by
        * proc_register_dynamic */
    7, /* Length of the file name */
    "rw_test", /* The file name */
    S_IFREG | S_IRUGO | S_IWUSR, /* File mode - this is a regular file which
                    * can be read by its owner, its group, and everybody
                    * else. Also, its owner can write to it.
                    *
                    * Actually, this field is just for reference, it's
                    * module_permission that does the actual check. It
                    * could use this field, but in our implementation it
                    * doesn't, for simplicity. */
    1,  /* Number of links (directories where the file is referenced) */
    0, 0,  /* The uid and gid for the file - we give it to root */
    80, /* The size of the file reported by ls. */
    &Inode_Ops_4_Our_Proc_File, /* A pointer to the inode structure for
                            * the file, if we need it. In our case we
                            * do, because we need a write function. */
    NULL  /* The read function for the file. Irrelevant, because we put it
           * in the inode structure above */
  };
/* Module initialization and cleanup ********************************** */
/* Initialize the module - register the proc file */
int init_module()
{
  /* Success if proc_register_dynamic is a success, failure otherwise */
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
}
/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

18.5 与设备文件通信

设备文件是用来代表物理设备的。大部分的物理设备既可以用作输出，也可以用作输入，所以系统内核中必须有某种机制使得设备驱动程序接收进程发送给设备的信息。这可以通过打开一个设备文件来实现，该设备文件用于输出，同时进程向设备文件写入，就像写入到一个普通文件中一样。在下面的例子中，这是由device_write函数实现的。

但这还不够。想一下你有一个串行口连接到一个调制解调器中，最自然的事情就是使用设备文件向调制解调器中写入信息，然后从调制解调器中读取信息。但问题是如果你希望和串行口自身通信时该怎么办？

在UNIX系统中解决这个问题的办法是使用一个特殊的函数，叫做 ioctl。每一个设备都可以有它自己的ioctl命令，可以用来从进程向内核发送信息或者从内核向进程返回信息。函数ioctl包括三个参数：一个相应的设备文件的文件描述符、 ioctl数目以及长整型的参数，它可以用来传递任何信息。

参数ioctl数目包括主设备号、ioctl的类型、命令和参数的类型。它通常由宏调用在头文件中创建。在下面的例子中，这个头文件是chardev.h，使用它的程序是ioctl.c。

下面是chardev.c 的源代码：

```
/* chardev.c
 *
 * Create an input/output character device
 */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* For character devices */
#include <linux/fs.h>       /* The character device definitions are here */
#include <linux/wrapper.h>  /* A wrapper which does next to nothing at
                             * at present, but may help for compatibility
                             * with future versions of Linux */
/* Our own IOCTL numbers */
#include "chardev.h"
#define SUCCESS 0
/* Device Declarations ********************************************** */
/* The name for our device, as it will appear in /proc/devices */
#define DEVICE_NAME "char_dev"
/* The maximum length of the message for the device */
#define BUF_LEN 80
/* Is the device open right now? Used to prevent concurent access into
 * the same device */
static int Device_Open = 0;
/* The message the device will give when asked */
static char Message[BUF_LEN];
/* How far did the process reading the message get? Useful if the
 * message is larger than the size of the buffer we get to fill in
 * device_read. */
static char *Message_Ptr;
/* This function is called whenever a process attempts to open the device
 * file */
static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
  printk ("device_open(%p,%p)\n", inode, file);
#endif
  /* We don't want to talk to two processes at the same time */
  if (Device_Open)
    return -EBUSY;
  /* If this was a process, we would have had to be more careful here,
   * because one process might have checked Device_Open right before the
   * other one tried to increment it. However, we're in the kernel, so
   * we're protected against context switches.
   */
  Device_Open++;
  /* Initialize the message */
```

```
  Message_Ptr = Message;
  MOD_INC_USE_COUNT;
  return SUCCESS;
}
/* This function is called when a process closes the device file. It
 * is not allowed to fail */
static void device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
  printk ("device_release(%p,%p)\n", inode, file);
#endif
  /* We're now ready for our next caller */
  Device_Open --;
  MOD_DEC_USE_COUNT;
}
/* This function is called whenever a process which already opened the
 * device file attempts to read from it. */
static int device_read(struct inode *inode,
                       struct file *file,
                       char *buffer,   /* The buffer to fill with the data */
                       int length)     /* The length of the buffer
                                  * (mustn't write beyond that!) */
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;
#ifdef DEBUG
  printk("device_read(%p,%p,%p,%d)\n",
    inode, file, buffer, length);
#endif
  /* If we're at the end of the message, return 0 (which signifies end
   * of file) */
  if (*Message_Ptr == 0)
    return 0;
  /* Actually put the data into the buffer */
  while (length && *Message_Ptr)  {
    /* Because the buffer is in the user data segment, not the kernel
     * data segment, assignment wouldn't work. Instead, we have to use
     * put_user which copies data from the kernel data segment to the user
     * data segment. */
    put_user(*(Message_Ptr++), buffer++);
    length --;
    bytes_read ++;
  }
#ifdef DEBUG
   printk ("Read %d bytes, %d left\n",
     bytes_read, length);
#endif
   /* Read functions are supposed to return the number of bytes actually
    * inserted into the buffer */
  return bytes_read;
}
```

```
/* This function is called when somebody tries to write into our device
 * file. */
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
{
  int i;
  return 1;
#ifdef DEBUG
  printk ("device_write(%p,%p,%s,%d)",
    inode, file, buffer, length);
#endif
  for(i=0; i<length && i<BUF_LEN; i++)
    Message[i] = get_user(buffer+i);
  Message_Ptr = Message;
  /* Again, return the number of input characters used */
  return i;
}
/* This function is called whenever a process tries to do an ioctl on
 * our device file. We get two extra parameters (additional to the
 * inode and file structures, which all device functions get): the number
 * of the ioctl called and the parameter given to the ioctl function.
 *
 * If the ioctl is write or read/write (meaning output is returned to
 * the calling process), the ioctl call returns the output of this
 * function.
 */
int device_ioctl(struct inode *inode,
                 struct file *file,
                 unsigned int ioctl_num,    /* The number of the ioctl */
                 unsigned long ioctl_param) /* The parameter to it */
{
  int i;
  char *temp;
  /* Switch according to the ioctl called */
  switch (ioctl_num) {
    case IOCTL_SET_MSG:
      /* Receive a pointer to a message (in user space) and set that to
       * be the device's message. */
      /* Get the parameter given to ioctl by the process */
      temp = (char *) ioctl_param;
      /* Find the length of the message */
      for (i=0; get_user(temp) && i<BUF_LEN; i++, temp++)
        ;
      /* Don't reinvent the wheel - call device_write */
      device_write(inode, file, (char *) ioctl_param, i);
      break;
    case IOCTL_GET_MSG:
      /* Give the current message to the calling process - the parameter
       * we got is a pointer, fill it. */
```

```
      i = device_read(inode, file, (char *) ioctl_param, 99);
      /* Warning - we assume here the buffer length is 100. If it's less
       * than that we might overflow the buffer, causing the process to
       * core dump.
       *
       * The reason we only allow up to 99 characters is that the NULL
       * which terminates the string also needs room. */
      /* Put a zero at the end of the buffer, so it will be properly
       * terminated */
      put_user('\0', (char *) ioctl_param+i);
      break;
    case IOCTL_GET_NTH_BYTE:
      /* This ioctl is both input (ioctl_param) and output (the return
       * value of this function) */
      return Message[ioctl_param];
      break;
  }
  return SUCCESS;
}
/* Module Declarations ********************************************** */
/* This structure will hold the functions to be called when
 * a process does something to the device we created. Since a pointer to
 * this structure is kept in the devices table, it can't be local to
 * init_module. NULL is for unimplemented functions. */
struct file_operations Fops = {
  NULL,   /* seek */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* select */
  device_ioctl,   /* ioctl */
  NULL,   /* mmap */
  device_open,
  device_release  /* a.k.a. close */
};
/* Initialize the module - Register the character device */
int init_module()
{
  int ret_val;
  /* Register the character device (atleast try) */
  ret_val = module_register_chrdev(MAJOR_NUM,
                                   DEVICE_NAME,
                                   &Fops);
  /* Negative values signify an error */
  if (ret_val < 0) {
    printk ("Sorry, registering the character device failed with %d\n",
      ret_val);
    return ret_val;
  }
  printk ("Registeration is a success. The major device number is %d.\n",
```

```
    MAJOR_NUM);
  printk ("If you want to talk to the device driver, you'll have to\n");
  printk ("create a device file. We suggest you use:\n");
  printk ("mknod %s c %d 0\n", DEVICE_FILE_NAME, MAJOR_NUM);
  printk ("The device file name is important, because the ioctl program\n");
  printk ("assumes that's the file you'll use.\n");
  return 0;
}
/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
  int ret;
  /* Unregister the device */
  ret = module_unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
  /* If there's an error, report it */
  if (ret < 0)
    printk("Error in module_unregister_chrdev: %d\n", ret);
}
```

下面是chardev.h的源代码：

```
/* chardev.h - the header file with the ioctl definitions.
 *
 * The declarations here have to be in a header file, because they need
 * to be known both to the kernel module (in chardev.c) and the process
 * calling ioctl (ioctl.c)
 */
#ifndef CHARDEV_H
#define CHARDEV_H
#include <linux/ioctl.h>
/* The major device number. We can't rely on dynamic registration any
 * more, because ioctls need to know it. */
#define MAJOR_NUM 100
/* Set the message of the device driver */
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
/* _IOR means that we're creating an ioctl command number for passing
 * information from a user process to the kernel module.
 *
 * The first arguments, MAJOR_NUM, is the major device number we're using.
 *
 * The second argument is the number of the command (there could be
 * several with different meanings).
 *
 * The third argument is the type we want to get from the process to the
 * kernel.
 */
/* Get the message of the device driver */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
 /* This IOCTL is used for output, to get the message of the device driver.
  * However, we still need the buffer to place the message in to be input,
  * as it is allocated by the process.
  */
```

```
/* Get the n'th byte of the message */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
 /* The IOCTL is used for both input and output. It receives from the
  * user a number, n, and returns Message[n]. */
/* The name of the device file */
#define DEVICE_FILE_NAME "char_dev"
#endif
```

下面是ioctl.c的源代码：

```
/* ioctl.c - the process to use ioctl's to control the kernel module
 *
 * Until now we could have used cat for input and output. But now we need
 * to do ioctl's, which require writing our own process.
 */
#include "chardev.h"    /* device specifics, such as ioctl numbers and
                         * the major device file. */

#include <fcntl.h>      /* open */
#include <unistd.h>     /* exit */
#include <sys/ioctl.h>  /* ioctl */
/* Functions for the ioctl calls */
ioctl_set_msg(int file_desc, char *message)
{
  int ret_val;
  ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);
  if (ret_val < 0) {
    printf ("ioctl_set_msg failed:%d\n", ret_val);
    exit(-1);
  }
}
ioctl_get_msg(int file_desc)
{
  int ret_val;
  char message[100];
  /* Warning - this is dangerous because we don't tell the kernel how
   * far it's allowed to write, so it might overflow the buffer. In a
   * real production program, we would have used two ioctls - one to tell
   * the kernel the buffer length and another to give it the buffer to fill
   */
  ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);
  if (ret_val < 0) {
    printf ("ioctl_get_msg failed:%d\n", ret_val);
    exit(-1);
  }
  printf("get_msg message:%s\n", message);
}
ioctl_get_nth_byte(int file_desc)
{
  int i;
```

```
  char c;
  printf("get_nth_byte message:");
  i = 0;
  while (c != 0) {
    c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);
    if (c < 0) {
      printf("ioctl_get_nth_byte failed at the %d'th byte:\n", i);
      exit(-1);
    }
    putchar(c);
  }
  putchar('\n');
}
/* Main - Call the ioctl functions */
main()
{
  int file_desc, ret_val;
  char *msg = "Message passed by ioctl\n";
  file_desc = open(DEVICE_FILE_NAME, 0);
  if (file_desc < 0) {
    printf ("Can't open device file: %s\n", DEVICE_FILE_NAME);
    exit(-1);
  }
  ioctl_get_nth_byte(file_desc);
  ioctl_get_msg(file_desc);
  ioctl_set_msg(file_desc, msg);
  close(file_desc);
}
```

18.6 启动参数

在前面的例子中，不得不硬性规定某些参数，例如/proc文件系统的文件名，或者设备的主设备号。这样应用起来十分的不方便，我们的目标是编写一个灵活易用的程序。

我们使用命令行参数为程序或者内核模块传递启动参数。在使用内核模块时，不使用参数argc 和 argv。可以在内核模块中定义全局变量，然后使用 insmod命令来写入这些变量。

在此内核模块中，我们定义了两个变量：str1和str2。所需要做的是编译内核模块，然后运行insmod str1=xxx str2=yyy即可。当调用init_module时，str1将指向字符串xxx，str2将指向字符串yyy。

下面是param.c的源程序：

```
/* param.c
 *
 * Receive command line parameters at module installation
 */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
```

```
#include <stdio.h>  /* I need NULL */
char *str1, *str2;
/* Initialize the module - show the parameters */
int init_module()
{
  if (str1 == NULL || str2 == NULL)
    printk("Next time, do insmod param str1=<something> str2=<something>\n");
  else
    printk("Strings:%s and %s\n", str1, str2);
  return 0;
}
/* Cleanup */
void cleanup_module()
{
}
```

18.7 系统调用

到现在为止，我们所做的只是使用已经定义好的内核机制来注册 /proc文件系统和设备处理程序，这对编写设备驱动程序是很有用的。但如果希望对系统作某些改动的话，这是不够的。

在系统中真正被所有进程都使用的内核通信方式是系统调用。例如当进程请求内核服务时，就使用的是系统调用。

一般情况下，进程是不能够存取系统内核的。它不能存取内核使用的内存段，也不能调用内核函数，CPU的硬件结构保证了这一点。只有系统调用是一个例外。进程使用寄存器中适当的值跳转到内核中事先定义好的代码中执行，（当然，这些代码是只读的）。在Intel结构的计算机中，这是由中断0x80实现的。

进程可以跳转到的内核中的位置叫做 system_call。在此位置的过程检查系统调用号，它将告诉内核进程请求的服务是什么。然后，它再查找系统调用表 sys_call_table，找到希望调用的内核函数的地址，并调用此函数，最后返回。

所以，如果希望改变一个系统调用的函数，需要做的是编写一个自己的函数，然后改变sys_call_table中的指针指向该函数，最后再使用 cleanup_module将系统调用表恢复到原来的状态。

下面的代码是这样一个例子。我们希望监视一个用户，每当该用户打开一个文件时就打印一条信息。我们用自己的程序 our_sys_open来替代打开文件的系统调用。此程序检查当前进程的uid，如果它与所监视用户的uid相等，则调用printk来显示将要打开的文件的名称。然后，它再调用原来的打开文件的系统调用，真正地打开文件。

在这个例子中，函数init_module用来替换sys_call_table表中的相应的位置，同时保存其原先的指针。函数 cleanup_module则用来将此变量恢复到中断表中。

下面是syscall.c的源代码：

```
/* syscall.c
 *
 * System call "stealing" sample
 */
/* The necessary header files */
/* Standard in kernel modules */
```

```
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
#include <sys/syscall.h>  /* The list of system calls */
#include <linux/sched.h>  /* For the current (process) structure, we need
                           * this to know who the current user is. */
/* The system call table (a table of functions). We just define this as
 * external, and the kernel will fill it up for us when we are insmod'ed
 */
extern void *sys_call_table[];
int uid;  /* UID we want to spy on - will be filled from the command line */
/* A pointer to the original system call. The reason we keep this, rather
 * than call the original function (sys_open), is because somebody else might
 * have replaced the system call before us. Note that this is not 100% safe,
 * because if another module replaced sys_open before us, then when we're
 * inserted we'll call the function in that module - and it might be removed
 * before we are. */
asmlinkage int (*original_call)(const char *, int, int);
/* The function we'll replace sys_open (the function called when you call
 * the open system call) with. To find the exact prototype, with the number
 * and type of arguments, we find the original function first (it's at
 * fs/open.c.
 * In theory, this means that we're tied to the current version of the
 * kernel. In practice, the system calls almost never change (it would
 * wreck havoc and require programs to be recompiled, since the system
 * calls are the interface between the kernel and the rest of the world).
 */
asmlinkage int our_sys_open(const char *filename, int flags, int mode)
{
  int i = 0;
  char ch;
  /* Check if this is the user we're spying on */
  if (uid == current->uid) {  /* current->uid is the uid of the user who
                                 ran the process which called the system
                                 call we got */
    /* Report the file, if relevant */
    printk("Opened file: ");
    do {
      ch = get_user(filename+i);
      i++;
      printk("%c", ch);
    } while (ch != 0);
    printk("\n");
  }
  /* Call the original sys_open - otherwise, we lose the ability to open
   * files */
  return original_call(filename, flags, mode);
}
/* Initialize the module - replace the system call */
int init_module()
```

```
{
  /* Warning - too late for it now, but maybe for next time... */
  printk("I'm dangerous. I hope you did a sync before you insmod'ed me.\n");
  printk("My counterpart, cleapup_module(), is even more dangerous. If\n");
  printk("you value your file system, it will be \"sync; rmmod\" \n");
  printk("when you remove it.\n");
  /* Keep a pointer to the original function in original_call, and
   * then replace the system call in the system call table with
   * our_sys_open */
  original_call = sys_call_table[__NR_open];
  sys_call_table[__NR_open] = our_sys_open;
  /* To get the address of the function for system call foo, go to
   * sys_call_table[__NR_foo]. */
  return 0;
}
/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
  /* Return the system call back to normal */
  if (sys_call_table[__NR_open] != our_sys_open) {
    printk("Somebody else also played with the open system call\n");
    printk("The system may be left in an unstable state.\n");
  }
  sys_call_table[__NR_open] = original_call;
}
```

18.8 阻塞进程

当进程请求内核模块服务时，如果此时内核模块正忙，那么可以将进程放入睡眠状态直到模块空闲。

下面就是这样一个例子。文件/proc/sleep每次只可以被一个进程打开。如果文件已经打开，内核模块将调用 module_interruptible_sleep_on函数，此函数将进程的状态改为TASK_INTERRUPTIBLE，这意味着进程在被唤醒之前一直处于 WaitQ队列中。然后，此函数调用调度算法切换到另一个进程。

当进程处理完文件以后，它将关闭文件，然后调用 module_close函数。此函数唤醒队列中所有的等待进程。也可以使用一个信号，例如用 Ctrl+C (SIGINT)来唤醒一个进程。

下面是 sleep.c的源代码：

```
/* sleep.c - create a /proc file, and if several processes try to open it
 * at the same time, put all but one to sleep */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* Necessary because we use proc fs */
#include <linux/proc_fs.h>
/* For putting processes to sleep and waking them up */
#include <linux/sched.h>
```

```
#include <linux/wrapper.h>
/* The module's file functions ********************************** */
/* Here we keep the last message received, to prove that we can process
 * our input */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];
/* Since we use the file operations struct, we can't use the special proc
 * output provisions - we have to use a standard read function, which is
 * this function */
static int module_output(struct inode *inode, /* The inode read */
                         struct file *file,   /* The file read */
                         char *buf, /* The buffer to put data to (in the
                                       * user segment) */
                         int len)  /* The length of the buffer */
{
  static int finished = 0;
  int i;
  char message[MESSAGE_LENGTH+30];
  /* Return 0 to signify end of file - that we have nothing more to say
   * at this point. */
  if (finished) {
    finished = 0;
    return 0;
  }
  /* If you don't understand this by now, you're hopeless as a kernel
   * programmer. */
  sprintf(message, "Last input:%s\n", Message);
  for(i=0; i<len && message[i]; i++)
    put_user(message[i], buf+i);
  finished = 1;
  return i;  /* Return the number of bytes "read" */
}
/* This function receives input from the user when the user writes to
 * the /proc file. */
static int module_input(struct inode *inode, /* The file's inode */
                        struct file *file,   /* The file itself */
                        const char *buf,     /* The buffer with the input */
                        int length)          /* The buffer's length */
{
  int i;
  /* Put the input into Message, where module_output will later be
   * able to use it */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
    Message[i] = get_user(buf+i);
  Message[i] = '\0';  /* we want a standard, zero terminated string */
  /* We need to return the number of input characters used */
  return i;
}
```

```
/* 1 if the file is currently open by somebody */
int Already_Open = 0;
/* Queue of processes who want our file */
static struct wait_queue *WaitQ = NULL;
/* Called when the /proc file is opened */
static int module_open(struct inode *inode,
                       struct file *file)
{
  /* If the file's flags include O_NONBLOCK, it means the process doesn't
   * want to wait for the file. In this case, if the file is already open,
   * we should fail with -EAGAIN, meaning "you'll have to try again",
   * instead of blocking a process which would rather stay awake. */
  if ((file->f_flags & O_NONBLOCK) && Already_Open)
    return -EAGAIN;
  /* This is the correct place for MOD_INC_USE_COUNT because if a process is
   * in the loop, which is within the kernel module, the kernel module must
   * not be removed. */
  MOD_INC_USE_COUNT;
  /* If the file is already open, wait until it isn't */
  while (Already_Open)
  {
    /* This function puts the current process, including any system calls,
     * such as us, to sleep. Execution will be resumed right after the
     * function call, either because somebody called wake_up(&WaitQ) (only
     * module_close does that, when the file is closed) or when a signal,
     * such as Ctrl-C, is sent to the process */
    module_interruptible_sleep_on(&WaitQ);
    /* If we woke up because we got a signal we're not blocking, return
     * -EINTR (fail the system call). This allows processes to be killed
     * or stopped. */
    if (current->signal & ~current->blocked) {
      /* It's important to put MOD_DEC_USE_COUNT here, because for processes
       * where the open is interrupted there will never be a corresponding
       * close. If we don't decrement the usage count here, we will be left
       * with a positive usage count which we'll have no way to bring down to
       * zero, giving us an immortal module, which can only be killed by
       * rebooting the machine. */
      MOD_DEC_USE_COUNT;
      return -EINTR;
    }
  }
  /* If we got here, Already_Open must be zero */
  /* Open the file */
  Already_Open = 1;
  return 0;  /* Allow the access */
}
/* Called when the /proc file is closed */
static void module_close(struct inode *inode,
                         struct file *file)
```

```
{
  /* Set Already_Open to zero, so one of the processes in the WaitQ will
   * be able to set Already_Open back to one and to open the file. All the
   * other processes will be called when Already_Open is back to one, so
   * they'll go back to sleep. */
  Already_Open = 0;
  /* Wake up all the processes in WaitQ, so if anybody is waiting for the
   * file, they can have it. */
  module_wake_up(&WaitQ);
  /* One less process interested in us */
  MOD_DEC_USE_COUNT;
}
/* This function decides whether to allow an operation (return zero) or
 * not allow it (return a non-zero which indicates why it is not allowed).
 *
 * The operation can be one of the following values:
 * 0 - Execute (run the "file" - meaningless in our case)
 * 2 - Write (input to the kernel module)
 * 4 - Read (output from the kernel module)
 *
 * This is the real function that checks file permissions. The permissions
 * returned by ls -l are for referece only, and can be overridden here.
 */
static int module_permission(struct inode *inode, int op)
{
  /* We allow everybody to read from our module, but only root (uid 0)
   * may write to it */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;
  /* If it's anything else, access is denied */
  return -EACCES;
}
/* Structures to register as the /proc file, with pointers to all the
 * relevant functions. ********************************************** */
/* File operations for our proc file. This is where we place pointers
 * to all the functions called when somebody tries to do something to
 * our file. NULL means we don't want to deal with something. */
static struct file_operations File_Ops_4_Our_Proc_File =
  {
    NULL,  /* lseek */
    module_output,  /* "read" from the file */
    module_input,   /* "write" to the file */
    NULL,  /* readdir */
    NULL,  /* select */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    module_open,      /* called when the /proc file is opened */
    module_close      /* called when it's classed */
  };
/* Inode operations for our proc file. We need it so we'll have some
 * place to specify the file operations structure we want to use, and
```

```
 * the function we use for permissions. It's also possible to specify
 * functions to be called for anything else which could be done to an
 * inode (although we don't bother, we just put NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
 {
   &File_Ops_4_Our_Proc_File,
   NULL, /* create */
   NULL, /* lookup */
   NULL, /* link */
   NULL, /* unlink */
   NULL, /* symlink */
   NULL, /* mkdir */
   NULL, /* rmdir */
   NULL, /* mknod */
   NULL, /* rename */
   NULL, /* readlink */
   NULL, /* follow_link */
   NULL, /* readpage */
   NULL, /* writepage */
   NULL, /* bmap */
   NULL, /* truncate */
   module_permission /* check for permissions */
 };
/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
 {
   0, /* Inode number - ignore, it will be filled by
       * proc_register_dynamic */
   5, /* Length of the file name */
   "sleep", /* The file name */
   S_IFREG | S_IRUGO | S_IWUSR, /* File mode - this is a regular file which
                 * can be read by its owner, its group, and everybody
                 * else. Also, its owner can write to it.
                 *
                 * Actually, this field is just for reference, it's
                 * module_permission that does the actual check. It
                 * could use this field, but in our implementation it
                 * doesn't, for simplicity. */
   1,  /* Number of links (directories where the file is referenced) */
   0, 0,  /* The uid and gid for the file - we give it to root */
   80, /* The size of the file reported by ls. */
   &Inode_Ops_4_Our_Proc_File, /* A pointer to the inode structure for
                 * the file, if we need it. In our case we
                 * do, because we need a write function. */
   NULL  /* The read function for the file. Irrelevant, because we put it
       * in the inode structure above */
 };
/* Module initialization and cleanup ********************************** */
/* Initialize the module - register the proc file */
int init_module()
{
```

```
  /* Success if proc_register_dynamic is a success, failure otherwise */
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
  /* proc_root is the root directory for the proc fs (/proc). This is
   * where we want our file to be located.
   */
}
/* Cleanup - unregister our file from /proc. This could get dangerous if
 * there are still processes waiting in WaitQ, because they are inside our
 * open function, which will get unloaded. I'll explain how to avoid removal
 * of a kernel module in such a case in chapter 9. */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

18.9 替换printk

可以使用一个指向当前任务的指针来获得它的tty的数据结构，在此数据结构中可以找到一个指向字符串写函数的指针。

下面是printk.c的源程序：

```
/* printk.c - send textual output to the tty you're running on, regardless
 * of whether it's passed through X11, telnet, etc. */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* Necessary here */
#include <linux/sched.h>    /* For current */
#include <linux/tty.h>      /* For the tty declarations */
/* Print the string to the appropriate tty, the one the current task uses */
void print_string(char *str)
{
  struct tty_struct *my_tty;
  /* The tty for the current task */
  my_tty = current->tty;
  /* If my_tty is NULL, it means that the current task has no TTY you can
   * print to (this is possible, for example, if it's a daemon). In this
   * case, there's nothing we can do. */
  if (my_tty != NULL) {
    /* my_tty->driver is a struct which holds the TTY's functions, one of
     * which (write) is used to write strings to the tty. It can be used to
     * take a string either from the user's memory segment or the kernel's
     * memory segment.
     *
     * The function's first parameter is the tty to write to, because the
     * same function would normally be used for all tty's of a certain type.
     * The second parameter controls whether the function receives a string
     * from kernel memory (false, 0) or from user memory (true, non zero).
     * The third parameter is a pointer to a string, and the fourth parameter
     * is the length of the string.
```

```
     */
    (*(my_tty->driver).write)(my_tty,   /* The tty itself */
                              0, /* We don't take the string from user space */
                              str, /* String */
                              strlen(str));  /* Length */
    /* TTYs were originally hardware devices, which (usually) adhered strictly
     * to the ASCII standard. According to ASCII, to move to a new line
     * you need two characters, a carriage return and a line feed. In UNIX,
     * on the other hand, the ASCII line feed is used for both purposes -
     * so we can't just use \n, because it wouldn't have a carriage return and
     * the next line will start at the column right after the line feed.
     * BTW, this is the reason why the text file  is different between
     * UNIX and Windows. In CP/M and its derivatives, such as MS-DOS and
     * Windows, the ASCII standard was strictly adhered to, and therefore a
     * new line requires both a line feed and a carriage return.
     */
    (*(my_tty->driver).write)(my_tty,
                              0,
                              "\015\012",
                              2);
  }
}
/* Module initialization and cleanup ********************************** */
/* Initialize the module - register the proc file */
int init_module()
{
  print_string("Module Inserted");
  return 0;
}
/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  print_string("Module Removed");
}
```

18.10 调度任务

我们经常需要在一个特定的时间执行某一个任务。如果任务是由一个进程执行的，可以把它放入crontab中。如果任务是由一个内核模块执行的，那么有两种实现方法。一种是把一个进程放入crontab中，等待需要时由系统调用来唤醒。但这样做的效率很低。

另一种方法是创建一个函数，此函数可以在计时器中断时调用。再创建一个任务，把它放在数据结构tq_struct中，此数据结构中包括一个指向这个函数的指针。然后，使用 queue_task 把任务放入到任务队列 tq_timer中，此队列中保存着下一次计时器中断时可以执行的任务。因为我们希望此函数可以持续地执行，所以需要把它不断地放回到队列 tq_timer中。

下面是sched.c的源程序：

```
/* sched.c - scheduale a function to be called on every timer interrupt. */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
```

```
#include <linux/module.h>   /* Specifically, a module */
/* Necessary because we use the proc fs */
#include <linux/proc_fs.h>
/* We scheduale tasks here */
#include <linux/tqueue.h>
/* We also need the ability to put ourselves to sleep and wake up later */
#include <linux/sched.h>
/* The number of times the timer interrupt has been called so far */
static int TimerIntrpt = 0;
/* This is used by cleanup, to prevent the module from being unloaded while
 * intrpt_routine is still in the task queue */
static struct wait_queue *WaitQ = NULL;
static void intrpt_routine(void *);
/* The task queue structure for this task, from tqueue.h */
static struct tq_struct Task = {
  NULL,   /* Next item in list - queue_task will do this for us */
  0,      /* A flag meaning we haven't been inserted into a task queue yet */
  intrpt_routine, /* The function to run */
  NULL    /* The void* parameter for that function */
};
/* This function will be called on every timer interrupt. Notice the void*
 * pointer - task functions can be used for more than one purpose, each
 * time getting a different parameter. */
static void intrpt_routine(void *irrelevant)
{
  /* Increment the counter */
  TimerIntrpt++;
  /* If cleanup wants us to die */
  if (WaitQ != NULL)
    wake_up(&WaitQ);   /* Now cleanup_module can return */
  else
    queue_task(&Task, &tq_timer);  /* Put ourselves back in the task queue */
}
/* Put data into the proc fs file. */
int procfile_read(char *buffer, char **buffer_location, off_t offset,
                  int buffer_length, int zero)
{
  int len;  /* The number of bytes actually used */
  /* This is static so it will still be in memory when we leave this
   * function */
  static char my_buffer[80];
  static int count = 1;
  /* We give all of our information in one go, so if the anybody asks us
   * if we have more information the answer should always be no.
   */
  if (offset > 0)
    return 0;
  /* Fill the buffer and get its length */
  len = sprintf(my_buffer, "Timer was called %d times so far\n", TimerIntrpt);
  count++;
  /* Tell the function which called us where the buffer is */
```

```
  *buffer_location = my_buffer;
  /* Return the length */
  return len;
}
struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register_dynamic */
    5, /* Length of the file name */
    "sched", /* The file name */
    S_IFREG | S_IRUGO, /* File mode - this is a regular file which can
                        * be read by its owner, its group, and everybody
                        * else */
    1,  /* Number of links (directories where the file is referenced) */
    0, 0,  /* The uid and gid for the file - we give it to root */
    80, /* The size of the file reported by ls. */
    NULL, /* functions which can be done on the inode (linking, removing,
           * etc.) - we don't support any. */
    procfile_read, /* The read function for this file, the function called
                    * when somebody tries to read something from it. */
    NULL /* We could have here a function to fill the file's inode, to
          * enable us to play with permissions, ownership, etc. */
  };
/* Initialize the module - register the proc file */
int init_module()
{
  /* Put the task in the tq_timer task queue, so it will be executed at
   * next timer interrupt */
  queue_task(&Task, &tq_timer);
  /* Success if proc_register_dynamic is a success, failure otherwise */
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
}
/* Cleanup */
void cleanup_module()
{
  /* Unregister our /proc file */
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
  /* Sleep until intrpt_routine is called one last time. This is necessary,
   * because otherwise we'll deallocate the memory holding intrpt_routine and
   * Task while tq_timer still references them. Notice that here we don't
   * allow signals to interrupt us.
   *
   * Notice that since WaitQ is now not NULL, this automatically tells
   * the interrupt routine it's time to die. */
 sleep_on(&WaitQ);
}
```

第19章　有关进程通信的编程

本章介绍有关Linux系统进程间通信的内容。

19.1　进程间通信简介

Linux 系统中的IPC (InterProcess Communication)函数提供了系统中多个进程之间相互通信的方法。对于Linux系统中的C语言程序员来说，系统中包括如下几种方式的IPC：

- 半双工UNIX 管道。
- FIFO(命名管道)。
- SYSV 风格的消息队列。
- SYSV 风格的信号量设置。
- SYSV 风格的共享内存段。
- 网络套接口 (Berkeley 风格)。
- 全双工管道 (STREAMS 管道)。

以上的这些函数，如果使用得当的话，提供了在 UNIX系统（当然也包括Linux系统）上开发客户机/服务器模式应用程序的有效工具。

19.2　半双工UNIX管道

19.2.1　基本概念

简单地说，管道就是一种连接一个进程的标准输出到另一个进程的标准输入的方法。管道是最古老的IPC工具，从UNIX系统一开始就存在。它提供了一种进程之间单向的通信方法。管道在系统中的应用很广泛，即使在shell环境中也要经常使用管道技术。

当进程创建一个管道时，系统内核设置了两个管道可以使用的文件描述符。一个用于向管道中输入信息(write)，另一个用于从管道中获取信息 (read)。在这一点上，管道没有多少的实际用途，因为创建管道的进程只能使用管道和进程自己通信。

如果进程通过管道fd0来发送数据，它可以通过fd1来读取此数据。当管道最初和进程连接时，在管道中传送数据是通过内核进行的。在 Linux系统环境下，管道最初在系统内核中都是使用索引节点表示的。当然，此索引节点保存在内核之中，而并不存在于任何的物理文件系统中。这就为我们提供了一种特别方便地使用I/O的方法，这一点我们以后将会看到。

这时，管道基本上是没有任何用的，因为进程没有必要和自己进行通信。通常的做法是进程派生出一个子进程。因为子进程将会继承父进程中所有打开的文件描述符，那么我们就可以在父进程和子进程之间通信了。

一个简单的管道创建完了。下面介绍如何使用管道。如果要直接存取管道，可以使用和低级的文件I/O同样的系统调用，因为在系统内核中管道实际上是由一个有效的索引节点表示的。

如果希望向管道中发送数据，可以使用系统调用 write()，反之，如果希望从管道中读取数据，可以使用系统调用read()。虽然大部分低级的文件I/O系统调用可以使用文件描述符，但一

些系统调用，例如lseek()不能通过文件描述符使用管道。

19.2.2 使用C语言创建管道

使用C语言创建管道要比在 shell下使用管道复杂一些。如果要使用 C语言创建一个简单的管道，可以使用系统调用 pipe()。它接受一个参数，也就是一个包括两个整数的数组。如果系统调用成功，此数组将包括管道使用的两个文件描述符。创建一个管道之后，一般情况下进程将产生一个新的进程。

系统调用：pipe()；

原型：int pipe(int fd[2])；

返回值：如果系统调用成功，返回0

如果系统调用失败返回-1：errno = EMFILE (没有空闲的文件描述符)
EMFILE (系统文件表已满)
EFAULT (fd数组无效)

注意 fd[0] 用于读取管道，fd[1] 用于写入管道。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
 main()
{
        int     fd[2];
        pipe(fd);
        .
        .
    }
```

一旦创建了管道，我们就可以创建一个新的子进程：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main()
{
        int     fd[2];
        pid_t   childpid;
        pipe(fd);
        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .
    }
```

如果父进程希望从子进程中读取数据，那么它应该关闭 fd1，同时子进程关闭 fd0。反之，如果父进程希望向子进程中发送数据，那么它应该关闭 fd0，同时子进程关闭fd1。因为文件描述符是在父进程和子进程之间共享，所以我们要及时地关闭不需要的管道的那一端。单从技术

的角度来说，如果管道的一端没有正确地关闭的话，你将无法得到一个 EOF。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main()
{
        int     fd[2];
        pid_t   childpid;
pipe(fd);
if((childpid = fork()) == -1)
{
        perror("fork");
        exit(1);
}
if(childpid == 0)
{
        /* Child process closes up input side of pipe */
        close(fd[0]);
}
 else
{
        /* Parent process closes up output side of pipe */
        close(fd[1]);
        }
        .
        .
}
```

正如前面提到的，一但创建了管道之后，管道所使用的文件描述符就和正常文件的文件描述符一样了。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{
    int     fd[2], nbytes;
    pid_t   childpid;
    char    string[] = "Hello, world!\n";
    char    readbuffer[80];
    pipe(fd);
    if((childpid = fork()) == -1)
    {
        perror("fork");
        exit(1);
    }
    if(childpid == 0)
    {
        /* Child process closes up input side of pipe */
        close(fd[0]);
        /* Send "string" through the output side of pipe */
        write(fd[1], string, strlen(string));
        exit(0);
```

```
    }
    else
    {
        /* Parent process closes up output side of pipe */
        close(fd[1]);
        /* Read in a string from the pipe */
        nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
        printf("Received string: %s", readbuffer);
    }
    return(0);
}
```

一般情况下，子进程中的文件描述符将会复制到标准的输入和输出中。这样子进程可以使用exec()执行另一个程序，此程序继承了标准的数据流。

系统调用：dup();

原型： int dup(int oldfd);

返回：如果系统调用成功，返回新的文件描述符

如果系统调用失败，返回-1: errno = EBADF (oldfd 不是有效的文件描述符)

EBADF (newfd超出范围)

EMFILE (进程的文件描述符太多)

注意 旧文件描述符oldfd没有关闭。

虽然旧文件描述符和新创建的文件描述符可以交换使用，但一般情况下需要首先关闭一个。系统调用dup()使用的是号码最小的空闲的文件描述符。

再看下面的程序：

```
.
.
childpid = fork();
if(childpid == 0)
{
    /* Close up standard input of the child */
    close(0);
    /* Duplicate the input side of pipe to stdin */
    dup(fd[0]);
    execlp("sort", "sort", NULL);
    .
}
```

因为文件描述符0（stdin）被关闭，所以dup()把管道的输入描述符复制到它的标准输入中。这样我们可以调用execlp()，使用sort程序覆盖子进程的正文段。因为新创建的程序从它的父进程中继承了标准输入/输出流，所以它实际上继承了管道的输入端作为它的标准输入端。现在，最初的父进程送往管道的任何数据都将会直接送往sort函数。

系统调用： dup2();

原型：int dup2(int oldfd, int newfd);

返回值： 如果调用成功，返回新的文件描述符

如果调用失败，返回-1 ： errno = EBADF (oldfd 不是有效的文件描述符)

EBADF (newfd 超出范围)

EMFILE (进程的文件描述符太多)

注意 dup2()将关闭旧文件描述符。

使用此系统调用，可以将close操作和文件描述符复制操作集成到一个系统调用中。另外，此系统调用保证了操作的自动进行，也就是说操作不能被其他的信号中断。这个操作将会在返回系统内核之前完成。如果使用前一个系统调用dup()，程序员不得不在此之前执行一个close()操作。

请看下面的程序：

```
.
.
childpid = fork();
if(childpid == 0)
{
    /* Close stdin, duplicate the input side of pipe to stdin */
    dup2(0, fd[0]);
    execlp("sort", "sort", NULL);
    .
    .
}
```

19.2.3 创建管道的简单方法

如果你认为上面创建和使用管道的方法过于繁琐的话，你也可以使用下面的简单的方法：

库函数：popen();

原型： FILE *popen (char *command, char *type);

返回值：如果成功，返回一个新的文件流。

如果无法创建进程或者管道，返回NULL。

此标准的库函数通过在系统内部调用pipe()来创建一个半双工的管道，然后它创建一个子进程，启动shell，最后在shell上执行command参数中的命令。管道中数据流的方向是由第二个参数type控制的。此参数可以是r或者 w，分别代表读或写。但不能同时为读和写。在Linux系统下，管道将会以参数type中第一个字符代表的方式打开。所以，如果你在参数type中写入rw，管道将会以读的方式打开。

虽然此库函数的用法很简单，但也有一些不利的地方。例如它失去了使用系统调用 pipe()时可以有的对系统的控制。尽管这样，因为可以直接地使用shell命令，所以shell中的一些通配符和其他的一些扩展符号都可以在command参数中使用。

使用popen()创建的管道必须使用pclose()关闭。其实， popen/pclose和标准文件输入/输出流中的fopen() / fclose()十分相似。

库函数： pclose();

原型： int pclose(FILE *stream);

返回值： 返回系统调用wait4()的状态 。

如果stream无效，或者系统调用wait4()失败，则返回-1。

注意 此库函数等待管道进程运行结束，然后关闭文件流。

库函数pclose()在使用popen()创建的进程上执行wait4()函数。当它返回时，它将破坏管道和文件系统。

在下面的例子中，用sort命令打开了一个管道，然后对一个字符数组排序：

```
#include <stdio.h>
#define MAXSTRS 5
int main(void)
{
        int  cntr;
        FILE *pipe_fp;
        char *strings[MAXSTRS] = { "echo", "bravo", "alpha",
                                "charlie", "delta"};
        /* Create one way pipe line with call to popen() */
        if (( pipe_fp = popen("sort", "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }
        /* Processing loop */
        for(cntr=0; cntr<MAXSTRS; cntr++) {
                fputs(strings[cntr], pipe_fp);
                fputc('\n', pipe_fp);
        }
        /* Close the pipe */
        pclose(pipe_fp);
        return(0);
}
```

因为popen()使用shell执行命令，所以所有的shell扩展符和通配符都可以使用。此外，它还可以和popen()一起使用重定向和输出管道函数。再看下面的例子：

```
popen("ls ~scottb", "r");
popen("sort > /tmp/foo", "w");
popen("sort | uniq | more", "w");
```

下面的程序是另一个使用popen()的例子，它打开两个管道（一个用于ls命令，另一个用于sort命令）：

```
#include <stdio.h>
int main(void)
{
        FILE *pipein_fp, *pipeout_fp;
        char readbuf[80];
        /* Create one way pipe line with call to popen() */
        if (( pipein_fp = popen("ls", "r")) == NULL)
        {
                perror("popen");
                exit(1);
        }
        /* Create one way pipe line with call to popen() */
        if (( pipeout_fp = popen("sort", "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }
        /* Processing loop */
        while(fgets(readbuf, 80, pipein_fp))
```

```
            fputs(readbuf, pipeout_fp);
        /* Close the pipes */
        pclose(pipein_fp);
        pclose(pipeout_fp);
        return(0);
}
```

最后，我们再看一个使用popen()的例子。此程序用于创建一个命令和文件之间的管道：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        FILE *pipe_fp, *infile;
        char readbuf[80];
        if( argc != 3) {
                fprintf(stderr, "USAGE:  popen3 [command] [filename]\n");
                exit(1);
        }
        /* Open up input file */
        if (( infile = fopen(argv[2], "rt")) == NULL)
        {
                perror("fopen");
                exit(1);
        }
        /* Create one way pipe line with call to popen() */
        if (( pipe_fp = popen(argv[1], "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }
        /* Processing loop */
        do {
                fgets(readbuf, 80, infile);
                if(feof(infile)) break;
                fputs(readbuf, pipe_fp);
        } while(!feof(infile));
        fclose(infile);
        pclose(pipe_fp);
        return(0);
}
```

下面是使用此程序的例子：

```
popen3 sort popen3.c
popen3 cat popen3.c
popen3 more popen3.c
popen3 cat popen3.c | grep main
```

19.2.4　使用管道的自动操作

所谓管道的自动操作就是指此操作不能被任何事件中断，整个操作一次发生。按照 POSIX 标准，在/usr/include/posix1_lim.h中定义了管道自动操作的最大缓冲区大小：

```
#define _POSIX_PIPE_BUF        512
```

每一次从自动管道中读取的数据和写入管道的数据可以达到 512个字节。超出这个缓冲区大小的数据将可能被分割，也就是说不能自动地操作。在 Linux系统下，此自动操作大小的限

制是在linux/limits.h中定义的：

```
#define PIPE_BUF        4096
```

当涉及到多个进程时，这种自动操作就显得特别重要。

19.2.5 使用半双工管道时的注意事项

可以通过打开两个管道来创建一个双向的管道。但需要在子进程中正确地设置文件描述符。

必须在系统调用fork()中调用pipe()，否则子进程将不会继承文件描述符。

当使用半双工管道时，任何关联的进程都必须共享一个相关的祖先进程。因为管道存在于系统内核之中，所以任何不在创建管道的进程的祖先进程之中的进程都将无法寻址它。而在命名管道中却不是这样。

19.3 命名管道

19.3.1 基本概念

命名管道和一般的管道基本相同，但也有一些显著的不同：

- 命名管道是在文件系统中作为一个特殊的设备文件而存在的。
- 不同祖先的进程之间可以通过管道共享数据。
- 当共享管道的进程执行完所有的I/O操作以后，命名管道将继续保存在文件系统中以便以后使用。

19.3.2 创建FIFO

可以有几种方法创建一个命名管道。头两种方法可以使用 shell。

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

上面的两个命名执行同样的操作，但其中有一点不同。命令 mkfifo提供一个在创建之后直接改变FIFO文件存取权限的途径，而命令mknod需要调用命令chmod。

一个物理文件系统可以通过p指示器十分容易地分辨出一个FIFO文件。

```
$ ls -l MYFIFO
prw-r--r--   1 root     root           0 Dec 14 22:15 MYFIFO|
```

请注意在文件名后面的管道符号“|”。

我们可以使用系统调用mknod()来创建一个FIFO管道：

库函数： mknod();

原型：int mknod(char *pathname, mode_t mode, dev_t dev);

返回值：如果成功,返回0

如果失败，返回 -1：errno = EFAULT (无效路径名)
EACCES (无存取权限)
ENAMETOOLONG (路径名太长)
ENOENT (无效路径名)
ENOTDIR (无效路径名)

下面看一个使用C语言创建FIFO管道的例子：

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

在这个例子中，文件/tmp/MYFIFO是要创建的FIFO文件。它的存取权限是0666。存取权限也可以使用umask 修改：

```
final_umask = requested_permissions & ~original_umask
```

一个常用的使用系统调用umask()的方法就是临时地清除umask的值：

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

另外，mknod()中的第三个参数只有在创建一个设备文件时才能用到。它包括设备文件的主设备号和从设备号。

19.3.3 FIFO操作

FIFO上的I/O操作和正常管道上的I/O操作基本一样，只有一个主要的不同。系统调用 open 用来在物理上打开一个管道。在半双工的管道中，这是不必要的。因为管道在系统内核中，而不是在一个物理的文件系统中。在我们的例子中，我们将像使用一个文件流一样使用管道，也就是使用fopen()打开管道，使用fclose()关闭它。

请看下面的简单的服务程序进程：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <linux/stat.h>
#define FIFO_FILE      "MYFIFO"
int main(void)
{
      FILE *fp;
      char readbuf[80];
      /* Create the FIFO if it does not exist */
      umask(0);
      mknod(FIFO_FILE, S_IFIFO|0666, 0);
      while(1)
      {
            fp = fopen(FIFO_FILE, "r");
            fgets(readbuf, 80, fp);
            printf("Received string: %s\n", readbuf);
            fclose(fp);
      }
      return(0);
}
```

因为FIFO管道缺省时有阻塞的函数，所以你可以在后台运行此程序：

```
$ fifoserver&
```

再来看一下下面的简单的客户端程序：

```
#include <stdio.h>
#include <stdlib.h>
#define FIFO_FILE      "MYFIFO"
int main(int argc, char *argv[])
{
      FILE *fp;
      if ( argc != 2 ) {
```

```
            printf("USAGE: fifoclient [string]\n");
            exit(1);
        }
        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
            perror("fopen");
            exit(1);
        }
        fputs(argv[1], fp);
        fclose(fp);
        return(0);
}
```

19.3.4 FIFO的阻塞

一般情况下，FIFO管道上将会有阻塞的情况发生。也就是说，如果一个 FIFO管道打开供读取的话，它将一直阻塞，直到其他的进程打开管道写入信息。这种过程反过来也一样。如果你不需要阻塞函数的话，你可以在系统调用 open()中设置O_NONBLOCK标志，这样可以取消缺省的阻塞函数。

19.3.5 SIGPIPE信号

最后一点是，一个管道必须既有读取进程，也要有写入进程。如果一个进程试图写入到一个没有读取进程的管道中，那么系统内核将会产生 SIGPIPE信号。当两个以上的进程同时使用管道时，这一点尤其重要。

19.4 System V IPC

19.4.1 基本概念

在UNIX的System V版本，AT&T引进了三种新形式的IPC功能（消息队列、信号量、以及共享内存）。但BSD版本的UNIX使用套接口作为主要的IPC形式。Linux系统支持这两个版本。我们先看一下System V版本的IPC功能。

1. IPC 标识符

每一个IPC目标都有一个唯一的IPC标识符。这里所指的IPC目标是指一个单独的消息队列、一个信号量集或者一个共享的内存段。系统内核使用此标识符在系统内核中指明 IPC目标。例如，如果希望存取一个共享的内存段，你唯一需要的就是给该内存段的标识符指定值。

我们所说的标识符的唯一性和所谈到的 IPC 目标的类型有关。我们假设一个标识符是12345。虽然两个消息队列不能使用这个相同的IPC标识符，但一个消息队列和一个共享内存段可以同时拥有此相同的标识符。

2. IPC 关键字

想要获得唯一的标识符，则必须使用一个 IPC关键字。客户端进程和服务器端进程必须双方都同意此关键字。这是建立一个客户机/服务器框架的第一步。

在System V IPC机制中，建立两端联系的路由方法是和IPC关键字直接相关的。

通过在应用程序中设置关键字值，每一次使用的关键字都可以是相同的。一般情况下，可以使用ftok()函数为客户端和服务器端产生关键字值。

库函数： ftok();
原型： key_t ftok (char *pathname, char proj);
返回值： 如果成功，则返回一个新的IPC 关键字值。
如果失败，则返回-1 ，可以使用系统调用stat() 得到错误的状态。

从ftok()中返回的值是结合索引节点值，第一个参数中的文件的从设备号和第2个参数的标识符的一个字符所产生的。这并不能保证它的唯一性，但应用程序可以检测冲突，并重新产生新的关键字。

```
key_t   mykey;
mykey = ftok("/tmp/myapp", 'a');
```

在上面的例子中，目录/tmp/myapp和a一起构成了关键字。另一个例子是用当前目录：

```
key_t   mykey;
mykey = ftok(".", 'a');
```

IPC关键字，无论是如何产生的，都用于在其后的IPC系统调用中创建或者存取IPC目标。

3. ipcs 命令

命令ipcs用于读取System V IPC目标的状态。

ipcs -q: 只显示消息队列。
ipcs -s: 只显示信号量。
ipcs -m: 只显示共享内存。
ipcs --help: 其他的参数。

缺省情况下，此命令将同时显示三种的IPC目标。请看下面的ipcs命令输出的例子：

```
------ Shared Memory Segments --------
shmid    owner    perms    bytes    nattch    status

------ Semaphore Arrays --------
semid    owner    perms    nsems    status

------ Message Queues --------
msqid    owner    perms    used-bytes  messages
0        root     660      5           1
```

这里我们可以看到一个单一的消息队列，它的标识符是0。它的所有者是root，存取权限是660，也就是-rw-rw--。消息队列中有一个消息，消息的大小是5个字节。

命令ipcs是一个十分有用的工具，它提供了一种观察系统内核中存储IPC目标的机制的方法。

4. ipcrm 命令

命令ipcrm用于将IPC目标从系统内核中移走。虽然也可以通过系统调用在程序中移走 IPC目标，但在一些情况下，特别是在开发环境中，经常需要手工移走 IPC目标。它的用法十分的简单：

```
ipcrm <msg | sem | shm>  <IPC ID>
```

只需指出要移走的是消息队列(msg)、信号量集 (sem)或者共享内存段(shm)既可。你可以通过ipcs命令得到IPC ID。

19.4.2 消息队列基本概念

消息队列是系统内核地址空间中的一个内部的链表。消息可以按照顺序发送到队列中，也可以以几种不同的方式从队列中读取。每一个消息队列用一个唯一的 IPC标识符表示。

1. 内部和用户数据结构

了解在系统内核中的数据结构是了解System V IPC机制如何工作的最好的方法。对一些数据结构的存取，对于一些最基本的操作是十分必要的，而一些其他的数据结构则存在于一些较为低级的层次中。

2. 消息缓冲区

首先我们看一下数据结构msgbuf。此数据结构可以说是消息数据的模板。虽然此数据结构需要用户自己定义，但了解系统中有这样一个数据结构是十分重要的。在 inux/msg.h中，此数据结构是这样定义的：

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
   long mtype;       /* type of message */
   char mtext[1];    /* message text */
};
```

在数据结构msgbuf中共有两个元素：

mtype 指消息的类型，它由一个整数来代表，并且，它只能是整数。

mtext 是消息数据本身。

这种给一个给定的消息指定类型的能力，使得你能够在一个单个的队列中重复使用消息。例如，你可以指定客户端进程一个多功能的数值，它可以作为服务器端发送过来的消息的类型。服务器端本身可以使用其他的数值，而客户端可以使用此数值向服务器端发送消息。在另一种情况中，一个应用程序可以将错误消息标志为类型 1，需要的消息标志为类型2，以此类推。这样组合的可能性是无穷无尽的。

另一个问题是，不要误解消息数据元素(mtext)的名字。这个字段不但可以存储字符，还可以存储任何其他的数据类型。此字段可以说是完全任意的，因为程序员自己可以重新定义此数据结构。请看下面重新定义的例子：

```
struct my_msgbuf {
      long    mtype;        /* Message type */
      long    request_id;   /* Request identifier */
      struct  client info;  /* Client information structure */
};
```

这里的消息类型字段和前面的一样，但数据结构的其余部分则由其他的两个字段所代替，而其中的一个还是另外一个结构。这就体现了消息队列的灵活之处。内核本身并不对消息结构中的数据做任何翻译。你可以在其中发送任何信息，但确实存在一个内部给定的消息大小的限制。在Linux系统中，这是在linux/msg.h中定义的：

```
#define MSGMAX  4056   /* <= 4056 */   /* max size of message (bytes) */
```

消息的最大的长度是4056个字节，其中包括mtype，它占用4个字节的长度。

3. 内核中msg 数据结构

系统内核将每一个消息保存在数据结构 msg组成的队列中。数据结构 msg在linux/msg.h中是如下定义的：

```
/* one msg structure for each message */
struct msg {
   struct msg *msg_next;   /* next message on queue */
   long  msg_type;
   char *msg_spot;         /* message text address */
   short msg_ts;           /* message text size */
```

```
};
```

各个字段的意义如下：

msg_next 是指向队列中下一个消息的指针。

msg_type 是消息类型，和用户自己在msgbuf结构中指定的消息类型相同。

msg_spot 指向消息体开始的指针。

msg_ts 是消息体的长度。

4. 内核 msqid_ds 数据结构

三种IPC目标都有自己的由系统内核维护的内部数据结构。对于消息队列来说，这个数据结构是msqid_ds。系统内核为系统中创建的每一个消息队列创建、存储和维护一个此数据结构的实例。这是在linux/msg.h中定义的：

```
/* one msqid structure for each queue on the system */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;  /* first message on queue */
    struct msg *msg_last;   /* last message in queue */
    time_t msg_stime;       /* last msgsnd time */
    time_t msg_rtime;       /* last msgrcv time */
    time_t msg_ctime;       /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes;      /* max number of bytes on queue */
    ushort msg_lspid;       /* pid of last msgsnd */
    ushort msg_lrpid;       /* last receive pid */
};
```

虽然你可能很少需要了解此数据结构中的大部分元素，但一个简要的了解还是很有必要的：

msg_perm 是数据结构ipc_perm的一个实例，而数据结构ipc_perm是在linux/ipc.h中定义的。它保存的是消息队列的存取权限的信息，和其他的例如队列的创建者等信息。

msg_first 指向队列中的第一个消息。

msg_last 指向队列中的最后一个消息。

msg_stime 发送到队列中的最后一条消息的时间。

msg_rtime 从队列中读取的最后一条消息的时间。

msg_ctime 是队列最后一次改动的时间。

wwait 和rwait 指向内核中等待队列的指针。当一个在消息队列中的操作认为进程需要进入到睡眠状态时使用。

msg_cbytes 是队列中所有消息的总长度。

msg_qnum 是当前在队列中的消息的数目。

msg_qbytes 是队列中的最大的字节数。

msg_lspid 为发送最后一条消息的进程的PID。

msg_lrpid 为最后一个读取队列中的消息的进程的PID。

5. 内核ipc_perm 数据结构

系统内核将IPC目标的权限的信息存储在数据结构 ipc_perm中。例如，在上面讨论的数据

结构中，元素msg_perm就是此类型的数据结构，它是在linux/ipc.h中定义的：

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* slot usage sequence number */
};
```

19.4.3 系统调用msgget()

如果希望创建一个新的消息队列，或者希望存取一个已经存在的消息队列，你可以使用系统调用msgget()。

系统调用： msgget();

原型： int msgget (key_t key, int msgflg);

返回值：如果成功，返回消息队列标识符

如果失败，则返回-1：errno =EACCESS (权限不允许)

EEXIST (队列已经存在，无法创建)

EIDRM (队列标志为删除)

ENOENT (队列不存在)

ENOMEM (创建队列时内存不够)

ENOSPC (超出最大队列限制)

系统调用msgget()中的第一个参数是关键字值（通常是由 ftok()返回的）。然后此关键字值将会和其他已经存在于系统内核中的关键字值比较。这时，打开和存取操作是和参数 msgflg中的内容相关的。

IPC_CREAT 如果内核中没有此队列，则创建它。

IPC_EXCL 当和IPC_CREAT一起使用时，如果队列已经存在，则失败。

如果单独使用IPC_CREAT，则msgget()要么返回一个新创建的消息队列的标识符，要么返回具有相同关键字值的队列的标识符。如果 IPC_EXCL 和 IPC_CREAT一起使用，则msgget()要么创建一个新的消息队列，要么如果队列已经存在则返回一个失败值 -1。IPC_EXCL单独使用是没有用处的。

下面看一个打开和创建一个消息队列的例子：

```
int open_queue( key_t keyval )
{
    int     qid;
    if((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
    {
        return(-1);
    }
    return(qid);
}
```

19.4.4 系统调用msgsnd()

一旦我们得到了队列标识符，我们就可以在队列上执行我们希望的操作了。如果想要往队列中发送一条消息，你可以使用系统调用 msgsnd（）：

系统调用： msgsnd();

原型： int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg);

返回值：如果成功， 0。

如果失败， -1 ：errno = EAGAIN (队列已满，并且使用了IPC_NOWAIT)
EACCES (没有写的权限)
EFAULT (msgp 地址无效)
EIDRM （消息队列已经删除)
EINTR (当等待写操作时，收到一个信号)
EINVAL (无效的消息队列标识符，非正数的消息类型，或者无效的消息长度)
ENOMEM (没有足够的内存复制消息缓冲区)

系统调用msgsnd()的第一个参数是消息队列标识符，它是由系统调用 msgget返回的。第二个参数是msgp，是指向消息缓冲区的指针。参数 msgsz中包含的是消息的字节大小，但不包括消息类型的长度（4个字节）。

参数msgflg可以设置为0（此时为忽略此参数），或者使用 IPC_NOWAIT。

如果消息队列已满，那么此消息则不会写入到消息队列中，控制将返回到调用进程中。如果没有指明，调用进程将会挂起，直到消息可以写入到队列中。

下面是一个发送消息的程序：

```
int send_message( int qid, struct mymsgbuf *qbuf )
{
        int     result, length;
        /* The length is essentially the size of the structure minus sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);
        if((result = msgsnd( qid, qbuf, length, 0)) == -1)
        {
                return(-1);
        }
        return(result);
}
```

这个小程序试图将存储在缓冲区qbuf中的消息发送到消息队列qid中。下面的程序是结合了上面两个程序的一个完整程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>
main()
{
        int    qid;
        key_t  msgkey;
        struct mymsgbuf {
                long    mtype;          /* Message type */
```

```
        int     request;        /* Work request number */
        double  salary;         /* Employee's salary */
    } msg;
    /* Generate our IPC key value */
    msgkey = ftok(".", 'm');
    /* Open/create the queue */
    if(( qid = open_queue( msgkey)) == -1) {
        perror("open_queue");
        exit(1);
    }
    /* Load up the message with arbitrary test data */
    msg.mtype   = 1;        /* Message type must be a positive number! */
    msg.request = 1;        /* Data element #1 */
    msg.salary  = 1000.00;  /* Data element #2 (my yearly salary!) */
    /* Bombs away! */
    if((send_message( qid, &msg )) == -1) {
        perror("send_message");
        exit(1);
    }
}
```

在创建和打开消息队列以后，我们将测试数据装入到消息缓冲区中。最后调用 send_messag 把消息发送到消息队列中。

现在在消息队列中有了一条消息，我们可以使用 ipcs命令来查看队列的状态。下面讨论如何从队列中获取消息。可以使用系统调用 msgrcv()：

系统调用： msgrcv();

原型： int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg);

返回值：如果成功，则返回复制到消息缓冲区的字节数。

如果失败，则返回-1： errno = E2BIG (消息的长度大于msgsz,没有MSG_NOERROR)

EACCES (没有读的权限)

EFAULT (msgp 指向的地址是无效的)

EIDRM (队列已经被删除)

EINTR (被信号中断)

EINVAL (msgqid 无效, 或者msgsz 小于0)

ENOMSG (使用IPC_NOWAIT, 同时队列中的消息无法满足要求)

很明显，第一个参数用来指定将要读取消息的队列。第二个参数代表要存储消息的消息缓冲区的地址。第三个参数是消息缓冲区的长度，不包括 mtype的长度，它可以按照如下的方法计算：

```
msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

第四个参数是要从消息队列中读取的消息的类型。如果此参数的值为 0，那么队列中最长时间的一条消息将返回，而不论其类型是什么。

如果调用中使用了 IPC_NOWAIT作为标志，那么当没有数据可以使用时，调用将把 ENOMSG返回到调用进程中。否则，调用进程将会挂起，直到队列中的一条消息满足 msgrcv() 的参数要求。如果当客户端等待一条消息的时候队列为空，将会返回 EIDRM。如果进程在等

待消息的过程中捕捉到一个信号，则返回EINTR。

下面就是一个从队列中读取消息的程序：

```
int read_message( int qid, long type, struct mymsgbuf *qbuf )
{
        int     result, length;
        /* The length is essentially the size of the structure minus sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);
        if((result = msgrcv( qid, qbuf, length, type,  0)) == -1)
        {
                return(-1);
        }
        return(result);
}
```

在成功地读取了一条消息以后，队列中的这条消息的入口将被删除。

参数msgflg中的MSG_NOERROR位提供一种额外的用途。如果消息的实际长度大于msgsz，同时使用了MSG_NOERROR，那么消息将会被截断，只有与msgsz长度相等的消息返回。一般情况下，系统调用msgrcv()会返回-1，而这条消息将会继续保存在队列中。我们可以利用这个特点编制一个程序，利用这个程序可以查看消息队列的情况，看看符合我们条件的消息是否已经到来：

```
int peek_message( int qid, long type )
{
        int     result, length;
        if((result = msgrcv( qid, NULL, 0, type,  IPC_NOWAIT)) == -1)
        {
                if(errno == E2BIG)
                        return(TRUE);
        }
        return(FALSE);
}
```

在上面的程序中，我们忽略了缓冲区的地址和长度。这样，系统调用将会失败。尽管如此，我们可以检查返回的E2BIG值，它说明符合条件的消息确实存在。

19.4.5 系统调用msgctl()

下面我们继续讨论如何使用一个给定的消息队列的内部数据结构。我们可以使用系统调用msgctl()来控制对消息队列的操作。

系统调用： msgctl();

调用原型： int msgctl (int msgqid, int cmd, struct msqid_ds *buf);

返回值： 0，如果成功。

-1，如果失败：errno = EACCES (没有读的权限同时cmd 是IPC_STAT)

EFAULT (buf 指向的地址无效)

EIDRM (在读取中队列被删除)

EINVAL (msgqid无效, 或者msgsz 小于0)

EPERM (IPC_SET或者IPC_RMID 命令被使用，但调用程序没有写的权限)

下面我们看一下可以使用的几个命令：

IPC_STAT

读取消息队列的数据结构msqid_ds，并将其存储在buf指定的地址中。

IPC_SET

设置消息队列的数据结构msqid_ds中的ipc_perm元素的值。这个值取自buf参数。

IPC_RMID

从系统内核中移走消息队列。

我们在前面讨论过了消息队列的数据结构 (msqid_ds)。系统内核中为系统中的每一个消息队列保存一个此数据结构的实例。通过使用 IPC_STAT命令，我们可以得到一个此数据结构的副本。下面的程序就是实现此函数的过程：

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
    if( msgctl( qid, IPC_STAT, qbuf) == -1)
    {
        return(-1);
    }
    return(0);
}
```

如果不能复制内部缓冲区，调用进程将返回 -1。如果调用成功，则返回 0。缓冲区中应该包括消息队列中的数据结构。

消息队列中的数据结构中唯一可以改动的元素就是 ipc_perm。它包括队列的存取权限和关于队列创建者和拥有者的信息。你可以改变用户的 id、用户的组id以及消息队列的存取权限。下面是一个修改队列存取模式的程序：

```
int change_queue_mode( int qid, char *mode )
{
    struct msqid_ds tmpbuf;
    /* Retrieve a current copy of the internal data structure */
    get_queue_ds( qid, &tmpbuf);
    /* Change the permissions using an old trick */
    sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);
    /* Update the internal data structure */
    if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
    {
        return(-1);
    }
    return(0);
}
```

我们通过调用get_queue_ds来读取队列的内部数据结构。然后，我们调用 sscanf()修改数据结构msg_perm中的mode 成员的值。但直到调用msgctl（）时，权限的改变才真正完成。在这里msgctl（）使用的是IPC_SET命令。

最后，我们使用系统调用msgctl()中的IPC_RMID命令删除消息队列：

```
int remove_queue( int qid )
{
    if( msgctl( qid, IPC_RMID, 0) == -1)
    {
        return(-1);
```

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

19.4.6　一个msgtool的实例

程序msgtool使用命令行参数来决定它要执行的操作。它的函数包括创建、发送、读取、改变权限以及删除消息队列。它的用法如下：

1. 发送消息

```
msgtool s (type) "text"
```

2. 读取消息

```
msgtool r (type)
```

3. 改变权限

```
msgtool m (mode)
```

4. 删除队列

```
msgtool d
```

下面是使用的例子：

```
msgtool  s   1 test
msgtool  s   5 test
msgtool  s   1 "This is a test"
msgtool  r   1
msgtool  d
msgtool  m   660
```

下面是msgtool的源代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_SEND_SIZE 80
struct mymsgbuf {
        long mtype;
        char mtext[MAX_SEND_SIZE];
};
void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);
int main(int argc, char *argv[])
{
        key_t key;
        int   msgqueue_id;
        struct mymsgbuf qbuf;
        if(argc == 1)
                usage();
        /* Create unique key via call to ftok() */
```

```
        key = ftok(".", 'm');
        /* Open the queue - create if necessary */
        if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
                perror("msgget");
                exit(1);
        }
        switch(tolower(argv[1][0]))
        {
                case 's': send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
                                atol(argv[2]), argv[3]);
                        break;
                case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
                        break;
                case 'd': remove_queue(msgqueue_id);
                        break;
                case 'm': change_queue_mode(msgqueue_id, argv[2]);
                        break;
                 default: usage();
        }

        return(0);
}
void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
        /* Send a message to the queue */
        printf("Sending a message ...\n");
        qbuf->mtype = type;
        strcpy(qbuf->mtext, text);
        if((msgsnd(qid, (struct msgbuf *)qbuf,
                strlen(qbuf->mtext)+1, 0)) ==-1)
        {
                perror("msgsnd");
                exit(1);
        }
}
void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
        /* Read a message from the queue */
        printf("Reading a message ...\n");
        qbuf->mtype = type;
        msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
        printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
}
void remove_queue(int qid)
{
        /* Remove the queue */
        msgctl(qid, IPC_RMID, 0);
}
void change_queue_mode(int qid, char *mode)
```

```
{
        struct msqid_ds myqueue_ds;
        /* Get current info */
        msgctl(qid, IPC_STAT, &myqueue_ds);
        /* Convert and load the mode */
        sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);
        /* Update the mode */
        msgctl(qid, IPC_SET, &myqueue_ds);
}
void usage(void)
{
        fprintf(stderr, "msgtool - A utility for tinkering with msg queues\n");
        fprintf(stderr, "\nUSAGE: msgtool (s)end <type> <messagetext>\n");
        fprintf(stderr, "              (r)ecv <type>\n");
        fprintf(stderr, "              (d)elete\n");
        fprintf(stderr, "              (m)ode <octal mode>\n");
        exit(1);
}
```

19.5　使用信号量编程

19.5.1　基本概念

信号量是一个可以用来控制多个进程存取共享资源的计数器。它经常作为一种锁定机制来防止当一个进程正在存取共享资源时，另一个进程也存取同一资源。下面先简要地介绍一下信号量中涉及到的数据结构。

1. 内核中的数据结构semid_ds

和消息队列一样，系统内核为内核地址空间中的每一个信号量集都保存了一个内部的数据结构。数据结构的原型是semid_ds。它是在linux/sem.h中做如下定义的：

```
/* One semid data structure for each set of semaphores in the system. */
struct semid_ds {
        struct ipc_perm sem_perm;       /* permissions .. see ipc.h */
        time_t          sem_otime;      /* last semop time */
        time_t          sem_ctime;      /* last change time */
        struct sem      *sem_base;      /* ptr to first semaphore in array */
        struct wait_queue *eventn;
        struct wait_queue *eventz;
        struct sem_undo  *undo;         /* undo requests on this array */
        ushort          sem_nsems;      /* no. of semaphores in array */
};
```

sem_perm　是在linux/ipc.h定义的数据结构ipc_perm的一个实例。它保存有信号量集的存取权限的信息，以及信号量集创建者的有关信息。

sem_otime　最后一次semop() 操作的时间。

sem_ctime　最后一次改动此数据结构的时间。

sem_base　指向数组中第一个信号量的指针。

sem_undo　数组中没有完成的请求的个数。

sem_nsems 信号量集（数组）中的信号量的个数。

2. 内核中的数据结构sem

在数据结构semid_ds中包含一个指向信号量数组的指针。此数组中的每一个元素都是一个数据结构sem。它也是在linux/sem.h中定义的：

```
/* One semaphore structure for each semaphore in the system. */
struct sem {
        short   sempid;       /* pid of last operation */
        ushort  semval;       /* current value */
        ushort  semncnt;      /* num procs awaiting increase in semval */
        ushort  semzcnt;      /* num procs awaiting semval = 0 */
};
```

sem_pid 最后一个操作的PID（进程ID）。

sem_semval 信号量的当前值。

sem_semncnt 等待资源的进程数目。

sem_semzcnt 等待资源完全空闲的进程数目。

19.5.2 系统调用semget()

我们可以使用系统调用 semget()创建一个新的信号量集，或者存取一个已经存在的信号量集：

系统调用：semget();

原型： int semget (key_t key, int nsems, int semflg);

返回值：如果成功，则返回信号量集的IPC 标识符。

如果失败，则返回-1： errno = EACCESS (没有权限)

EEXIST (信号量集已经存在，无法创建)

EIDRM (信号量集已经删除)

ENOENT (信号量集不存在，同时没有使用IPC_CREAT)

ENOMEM (没有足够的内存创建新的信号量集)

ENOSPC (超出限制)

系统调用semget()的第一个参数是关键字值（一般是由系统调用 ftok()返回的）。系统内核将此值和系统中存在的其他的信号量集的关键字值进行比较。打开和存取操作与参数 semflg中的内容相关。

IPC_CREAT 如果信号量集在系统内核中不存在，则创建信号量集。

IPC_EXCL 当和IPC_CREAT 一同使用时，如果信号量集已经存在，则调用失败。

如果单独使用 IPC_CREAT ，则semget()要么返回新创建的信号量集的标识符，要么返回系统中已经存在的同样的关键字值的信号量的标识符。如果 IPC_EXCL和IPC_CREAT一同使用，则要么返回新创建的信号量集的标识符，要么返回 -1。IPC_EXCL 单独使用没有意义。

参数nsems指出了一个新的信号量集中应该创建的信号量的个数。信号量集中最多的信号量的个数是在linux/sem.h中定义的：

```
#define SEMMSL  32      /* <=512 max num of semaphores per id */
```

下面是一个打开和创建信号量集的程序：

```
int open_semaphore_set( key_t keyval, int numsems )
{
```

```
    int     sid;

    if ( ! numsems )
        return(-1);

    if((sid = semget( mykey, numsems, IPC_CREAT | 0660 )) == -1)
    {
        return(-1);
    }

    return(sid);
}
```

19.5.3 系统调用semop()

系统调用：semop();
调用原型： int semop (int semid, struct sembuf *sops, unsigned nsops);
返回值： 0，如果成功。
-1，如果失败： errno = E2BIG (nsops 大于最大的ops 数目)
EACCESS (权限不够)
EAGAIN (使用了IPC_NOWAIT，但操作不能继续进行)
EFAULT (sops 指向的地址无效)
EIDRM (信号量集已经删除)
EINTR (当睡眠时接收到其他信号)
EINVAL (信号量集不存在, 或者semid 无效)
ENOMEM (使用了SEM_UNDO ,但无足够的内存创建所需的数据结构)
ERANGE (信号量值超出范围)

第一个参数是关键字值。第二个参数是指向将要操作的数组的指针。第三个参数是数组中的操作的个数。参数sops指向由sembuf组成的数组。此数组是在linux/sem.h中定义的：

```
/* semop system call takes an array of these */
struct sembuf {
    ushort  sem_num;      /* semaphore index in array */
    short   sem_op;       /* semaphore operation */
    short   sem_flg;      /* operation flags */
};
```

sem_num 将要处理的信号量的个数。
sem_op 要执行的操作。
sem_flg 操作标志。

如果sem_op是负数，那么信号量将减去它的值。这和信号量控制的资源有关。如果没有使用IPC_NOWAIT，那么调用进程将进入睡眠状态，直到信号量控制的资源可以使用为止。

如果sem_op是正数，则信号量加上它的值。这也就是进程释放信号量控制的资源。

最后，如果sem_op是0，那么调用进程将调用sleep()，直到信号量的值为0。这在一个进程等待完全空闲的资源时使用。

19.5.4 系统调用semctl()

系统调用：semctl();
原型： int semctl (int semid, int semnum, int cmd, union semun arg);
返回值：如果成功，则为一个正数。
如果失败，则为-1：errno = EACCESS (权限不够)
EFAULT (arg 指向的地址无效)
EIDRM (信号量集已经删除)
EINVAL (信号量集不存在，或者semid 无效)
EPERM (EUID 没有cmd 的权利)
ERANGE (信号量值超出范围)

系统调用semctl 用来执行在信号量集上的控制操作。这和在消息队列中的系统调用 msgctl 是十分相似的。但这两个系统调用的参数略有不同。因为信号量一般是作为一个信号量集使用的，而不是一个单独的信号量。所以在信号量集的操作中，不但要知道 IPC关键字值，也要知道信号量集中的具体的信号量。

这两个系统调用都使用了参数 cmd，它用来指出要操作的具体命令。两个系统调用中的最后一个参数也不一样。在系统调用 msgctl中，最后一个参数是指向内核中使用的数据结构的指针。我们使用此数据结构来取得有关消息队列的一些信息，以及设置或者改变队列的存取权限和使用者。但在信号量中支持额外的可选的命令，这样就要求有一个更为复杂的数据结构。

系统调用semctl()的第一个参数是关键字值。第二个参数是信号量数目。

参数cmd中可以使用的命令如下：

IPC_STAT 读取一个信号量集的数据结构semid_ds，并将其存储在semun中的buf参数中。

IPC_SET 设置信号量集的数据结构semid_ds中的元素ipc_perm，其值取自semun中的buf参数。

IPC_RMID 将信号量集从内存中删除。

GETALL 用于读取信号量集中的所有信号量的值。

GETNCNT 返回正在等待资源的进程数目。

GETPID 返回最后一个执行semop操作的进程的PID。

GETVAL 返回信号量集中的一个单个的信号量的值。

GETZCNT 返回这在等待完全空闲的资源的进程数目。

SETALL 设置信号量集中的所有的信号量的值。

SETVAL 设置信号量集中的一个单独的信号量的值。

参数arg代表一个semun的实例。semun是在linux/sem.h中定义的：

```
/* arg for semctl system calls. */
union semun {
    int val;              /* value for SETVAL */
    struct semid_ds *buf;   /* buffer for IPC_STAT & IPC_SET */
    ushort *array;        /* array for GETALL & SETALL */
    struct seminfo *__buf;  /* buffer for IPC_INFO */
    void *__pad;
};
```

val 当执行SETVAL命令时使用。

buf 在IPC_STAT/IPC_SET命令中使用。代表了内核中使用的信号量的数据结构。

array 在使用GETALL/SETALL命令时使用的指针。

下面的程序返回信号量的值。当使用GETVAL命令时，调用中的最后一个参数被忽略：

```
int get_sem_val( int sid, int semnum )
{
    return( semctl(sid, semnum, GETVAL, 0));
}
```

下面是一个实际应用的例子：

```
    #define MAX_PRINTERS 5

    printer_usage()
    {
        int x;

        for(x=0; x<MAX_PRINTERS; x++)
            printf("Printer %d: %d\n\r", x, get_sem_val( sid, x ));
    }
```

下面的程序可以用来初始化一个新的信号量值：

```
void init_semaphore( int sid, int semnum, int initval)
{
    union semun semopts;

    semopts.val = initval;
    semctl( sid, semnum, SETVAL, semopts);
}
```

注意 系统调用semctl中的最后一个参数是一个联合类型的副本，而不是一个指向联合类型的指针。

19.5.5 使用信号量集的实例：semtool

你可以使用semtool程序在shell中创建、使用、控制和删除一个信号量集。

1) 创建一个信号量集

```
semtool c (number of semaphores in set)
```

2) 锁定一个信号量

```
semtool l (semaphore number to lock)
```

3) 解锁一个信号量

```
semtool u (semaphore number to unlock)
```

4) 改变信号量的权限

```
semtool m (mode)
```

5) 删除一个信号量

```
semtool d
```

下面是一些使用的实例：

```
semtool  c 5
semtool  l
semtool  u
semtool  m 660
semtool  d
```

最后是此程序的完整的源代码：

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEM_RESOURCE_MAX        1       /* Initial value of all semaphores */
void opensem(int *sid, key_t key);
void createsem(int *sid, key_t key, int members);
void locksem(int sid, int member);
void unlocksem(int sid, int member);
void removesem(int sid);
unsigned short get_member_count(int sid);
int getval(int sid, int member);
void dispval(int sid, int member);
void changemode(int sid, char *mode);
void usage(void);
int main(int argc, char *argv[])
{
        key_t key;
        int   semset_id;
        if(argc == 1)
                usage();
        /* Create unique key via call to ftok() */
        key = ftok(".", 's');
        switch(tolower(argv[1][0]))
        {
                case 'c': if(argc != 3)
                                usage();
                          createsem(&semset_id, key,  atoi(argv[2]));
                          break;
                case 'l': if(argc != 3)
                                usage();
                          opensem(&semset_id, key);
                          locksem(semset_id, atoi(argv[2]));
                          break;
                case 'u': if(argc != 3)
                                usage();
                          opensem(&semset_id, key);
                          unlocksem(semset_id, atoi(argv[2]));
                          break;
                case 'd': opensem(&semset_id, key);
                          removesem(semset_id);
                          break;
                case 'm': opensem(&semset_id, key);
                          changemode(semset_id, argv[2]);
                          break;
                 default: usage();

        }
```

```
        return(0);
}

void opensem(int *sid, key_t key)
{
        /* Open the semaphore set - do not create! */

        if((*sid = semget(key, 0, 0666)) == -1)
        {
                printf("Semaphore set does not exist!\n");
                exit(1);
        }
}
void createsem(int *sid, key_t key, int members)
{
        int cntr;
        union semun semopts;
        if(members > SEMMSL) {
                printf("Sorry, max number of semaphores in a set is %d\n",
                        SEMMSL);
                exit(1);
        }
        printf("Attempting to create new semaphore set with %d members\n",
                        members);
        if((*sid = semget(key, members, IPC_CREAT|IPC_EXCL|0666))
                        == -1)
        {
                fprintf(stderr, "Semaphore set already exists!\n");
                exit(1);
        }
        semopts.val = SEM_RESOURCE_MAX;
        /* Initialize all members (could be done with SETALL) */
        for(cntr=0; cntr<members; cntr++)
                semctl(*sid, cntr, SETVAL, semopts);
}
void locksem(int sid, int member)
{
        struct sembuf sem_lock={ 0, -1, IPC_NOWAIT};
        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "semaphore member %d out of range\n", member);
                return;
        }
        /* Attempt to lock the semaphore set */
        if(!getval(sid, member))
        {
                fprintf(stderr, "Semaphore resources exhausted (no lock)!\n");
                exit(1);
        }
        sem_lock.sem_num = member;
        if((semop(sid, &sem_lock, 1)) == -1)
```

```
        {
                fprintf(stderr, "Lock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources decremented by one (locked)\n");
        dispval(sid, member);
}
void unlocksem(int sid, int member)
{
        struct sembuf sem_unlock={ member, 1, IPC_NOWAIT};
        int semval;
        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "semaphore member %d out of range\n", member);
                return;
        }
        /* Is the semaphore set locked? */
        semval = getval(sid, member);
        if(semval == SEM_RESOURCE_MAX) {
                fprintf(stderr, "Semaphore not locked!\n");
                exit(1);
        }
        sem_unlock.sem_num = member;
        /* Attempt to lock the semaphore set */
        if((semop(sid, &sem_unlock, 1)) == -1)
        {
                fprintf(stderr, "Unlock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources incremented by one (unlocked)\n");
        dispval(sid, member);
}
void removesem(int sid)
{
        semctl(sid, 0, IPC_RMID, 0);
        printf("Semaphore removed\n");
}
unsigned short get_member_count(int sid)
{
        union semun semopts;
        struct semid_ds mysemds;
        semopts.buf = &mysemds;
        /* Return number of members in the semaphore set */
        return(semopts.buf->sem_nsems);
}
int getval(int sid, int member)
{
        int semval;
        semval = semctl(sid, member, GETVAL, 0);
        return(semval);
```

```
}
void changemode(int sid, char *mode)
{
        int rc;
        union semun semopts;
        struct semid_ds mysemds;
        /* Get current values for internal data structure */
        semopts.buf = &mysemds;
        rc = semctl(sid, 0, IPC_STAT, semopts);
        if (rc == -1) {
                perror("semctl");
                exit(1);
        }
        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);
        /* Change the permissions on the semaphore */
        sscanf(mode, "%ho", &semopts.buf->sem_perm.mode);
        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);
        printf("Updated...\n");
}
void dispval(int sid, int member)
{
        int semval;
        semval = semctl(sid, member, GETVAL, 0);
        printf("semval for member %d is %d\n", member, semval);
}
void usage(void)
{
        fprintf(stderr, "semtool - A utility for tinkering with semaphores\n");
        fprintf(stderr, "\nUSAGE:  semtool4 (c)reate <semcount>\n");
        fprintf(stderr, "               (l)ock <sem #>\n");
        fprintf(stderr, "               (u)nlock <sem #>\n");
        fprintf(stderr, "               (d)elete\n");
        fprintf(stderr, "               (m)ode <mode>\n");
        exit(1);
}
```

19.6　共享内存

19.6.1　基本概念

共享内存就是由几个进程共享一段内存区域。这可以说是最快的 IPC形式，因为它无须任何的中间操作（例如，管道、消息队列等）。它只是把内存段直接映射到调用进程的地址空间中。这样的内存段可以是由一个进程创建的，然后其他的进程可以读写此内存段。

19.6.2　系统内部用户数据结构shmid_ds

每个系统的共享内存段在系统内核中也保持着一个内部的数据结构 shmid_ds。此数据结构是在linux/shm.h中定义的，如下所示：

```
        /* One shmid data structure for each shared memory segment in the system. */
```

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

                                    /* the following are private */

    unsigned short   shm_npages;    /* size of segment (pages) */
    unsigned long   *shm_pages;     /* array of ptrs to frames -> SHMMAX */
    struct vm_area_struct *attaches;    /* descriptors for attaches */
};
```

shm_perm　是数据结构ipc_perm的一个实例。这里保存的是内存段的存取权限，和其他的有关内存段创建者的信息。

shm_segsz　内存段的字节大小。

shm_atime　最后一个进程存取内存段的时间。

shm_dtime　最后一个进程离开内存段的时间。

shm_ctime　内存段最后改动的时间。

shm_cpid　内存段创建进程的PID。

shm_lpid　最后一个使用内存段的进程的PID。

shm_nattch　当前使用内存段的进程总数。

19.6.3 系统调用shmget()

系统调用：shmget();

原型：int shmget (key_t key, int size, int shmflg);

返回值：如果成功，返回共享内存段标识符。

如果失败，则返回-1：errno = EINVAL (无效的内存段大小)
EEXIST (内存段已经存在，无法创建)
EIDRM (内存段已经被删除)
ENOENT (内存段不存在)
EACCES (权限不够)
ENOMEM (没有足够的内存来创建内存段)

系统调用shmget() 中的第一个参数是关键字值（它是用系统调用 ftok()返回的）。其他的操作都要依据shmflg中的命令进行。

IPC_CREAT　如果系统内核中没有共享的内存段，则创建一个共享的内存段。

IPC_EXCL　当和IPC_CREAT一同使用时，如果共享内存段已经存在，则调用失败。

当IPC_CREAT单独使用时，系统调用shmget()要么返回一个新创建的共享内存段的标识符，要么返回一个已经存在的共享内存段的关键字值。如果 IPC_EXCL和IPC_CREAT一同使用，则要么系统调用新创建一个共享的内存段，要么返回一个错误值 -1。IPC_EXCL单独使用没有意义。

下面是一个定位和创建共享内存段的程序：

```
int open_segment( key_t keyval, int segsize )
{
    int    shmid;

    if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
    {
        return(-1);
    }

    return(shmid);
}
```

一旦一个进程拥有了一个给定的内存段的有效 IPC标识符，它的下一步就是将共享的内存段映射到自己的地址空间中。

19.6.4　系统调用shmat()

系统调用： shmat();
原型：int shmat (int shmid, char *shmaddr, int shmflg);
返回值：如果成功，则返回共享内存段连接到进程中的地址。
　　如果失败，则返回-1：errno =　EINVAL (无效的IPC ID 值或者无效的地址)
　　ENOMEM (没有足够的内存)
　　EACCES (存取权限不够)

如果参数addr的值为0，那么系统内核则试图找出一个没有映射的内存区域。我们推荐使用这种方法。你可以指定一个地址，但这通常是为了加快对硬件设备的存取，或者解决和其他程序的冲突。

下面的程序中的调用参数是一个内存段的IPC标识符，返回内存段连接的地址：

```
char *attach_segment( int shmid )
{
    return(shmat(shmid, 0, 0));
}
```

一旦内存段正确地连接到进程以后，进程中就有了一个指向该内存段的指针。这样，以后就可以使用指针来读取此内存段了。但一定要注意不能丢失该指针的初值。

19.6.5　系统调用shmctl()

系统调用：shmctl();
原型：int shmctl (int shmqid, int cmd, struct shmid_ds *buf);
返回值： 0，如果成功。
　　-1，如果失败：errno =　EACCES (没有读的权限，同时命令是IPC_STAT)
　　EFAULT (buf指向的地址无效，同时命令是 IPC_SET和IPC_STAT)
　　EIDRM (内存段被移走)
　　EINVAL (shmqid 无效)
　　EPERM (使用IPC_SET 或者 IPC_RMID 命令，但调用进程没有写的权限)

IPC_STAT

读取一个内存段的数据结构shmid_ds，并将它存储在buf参数指向的地址中。

IPC_SET

设置内存段的数据结构shmid_ds中的元素ipc_perm的值。从参数buf中得到要设置的值。

IPC_RMID

标志内存段为移走。

命令IPC_RMID并不真正从系统内核中移走共享的内存段，而是把内存段标记为可移除。

进程调用系统调用shmdt（）脱离一个共享的内存段。

19.6.6 系统调用shmdt()

系统调用：shmdt();

调用原型：int shmdt (char *shmaddr);

返回值：如果失败，则返回 -1：errno = EINVAL (无效的连接地址)

当一个进程不在需要共享的内存段时，它将会把内存段从其地址空间中脱离。但这不等于将共享内存段从系统内核中移走。当进程脱离成功后，数据结构 shmid_ds中元素shm_nattch将减1。当此数值减为0以后，系统内核将物理上把内存段从系统内核中移走。

19.6.7 使用共享内存的实例：shmtool

最后一个System V IPC目标的例子是shmtool，它同样也可以在shell下创建、读取、写入和删除共享的内存段。

1) 将字符串写入到内存段中

```
shmtool w "text"
```

2) 从内存段中读取字符串

```
shmtool r
```

3) 改变内存段的权限

```
shmtool m (mode)
```

4) 删除内存段

```
shmtool d
```

下面是应用的实例：

```
shmtool  w   test
shmtool  w   "This is a test"
shmtool  r
shmtool  d
shmtool  m   660
```

最后在这里给出程序的源代码如下：

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

#define SEGSIZE 100
main(int argc, char *argv[])
{       key_t key;
        int   shmid, cntr;
char  *segptr;
```

```
    if(argc == 1)
        usage();        /* Create unique key via call to ftok() */
    key = ftok(".", 'S');
    /* Open the shared memory segment - create if necessary */
    if((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
    {
        printf("Shared memory segment exists - opening as client\n");
        /* Segment probably already exists - try as a client */
        if((shmid = shmget(key, SEGSIZE, 0)) == -1)
         {
            perror("shmget");
            exit(1);
        }
    }
    else
    {
        printf("Creating new shared memory segment\n");
    }
    /* Attach (map) the shared memory segment into the current process */
    if((segptr = shmat(shmid, 0, 0)) == -1)
    {
        perror("shmat");
        exit(1);
    }
    switch(tolower(argv[1][0]))
    {
        case 'w': writeshm(shmid, segptr, argv[2]);
                  break;
        case 'r': readshm(shmid, segptr);
                  break;
        case 'd': removeshm(shmid);
                  break;
        case 'm': changemode(shmid, argv[2]);
                  break;
         default: usage();
    }
}
writeshm(int shmid, char *segptr, char *text)
{
    strcpy(segptr, text);
    printf("Done...\n");
}
readshm(int shmid, char *segptr)
{
    printf("segptr: %s\n", segptr);
}
removeshm(int shmid)
{
    shmctl(shmid, IPC_RMID, 0);
    printf("Shared memory segment marked for deletion\n");
}
```

```
changemode(int shmid, char *mode)
 {      struct shmid_ds myshmds;
       /* Get current values for internal data structure */
       shmctl(shmid, IPC_STAT, &myshmds);       /* Display old permissions */
       printf("Old permissions were: %o\n", myshmds.shm_perm.mode);
       /* Convert and load the mode */
       sscanf(mode, "%o", &myshmds.shm_perm.mode);       /* Update the mode */
       shmctl(shmid, IPC_SET, &myshmds);
       printf("New permissions are : %o\n", myshmds.shm_perm.mode);
}
usage()
{
       fprintf(stderr, "shmtool - A utility for tinkering with shared memory\n");
       fprintf(stderr, "\nUSAGE:  shmtool (w)rite <text>\n");
       fprintf(stderr, "               (r)ead\n");
       fprintf(stderr, "               (d)elete\n");
       fprintf(stderr, "               (m)ode change <octal mode>\n");
       exit(1);
}
```

第20章　高级线程编程

本章从线程的基本概念入手，介绍Linux的高级编程内容。

20.1　线程的概念和用途

线程通常叫做轻型的进程。虽然这个叫法有些简单化，但这有利于了解线程的概念。线程和UNIX系统中的进程十分接近，要了解这两者之间的区别，我们应该看一下 UNIX系统中的进程和Mach的任务和线程之间的关系。在UNIX系统中，一个进程包括一个可执行的程序和一系列的资源，例如文件描述符表和地址空间。在 Mach中，一个任务仅包括一系列的资源；线程处理所有的可执行代码。一个 Mach的任务可以有任意数目的线程和它相关，同时每个线程必须和某个任务相关。和某一个给定的任务相关的所有线程都共享任务的资源。这样，一个线程就是一个程序计数器、一个堆栈和一系列的寄存器。所有需要使用的数据结构都属于任务。一个UNIX系统中的进程在Mach中对应于一个任务和一个单独的线程。

因为线程和进程比起来很小，所以相对来说，线程花费更少的 CPU资源。进程往往需要它们自己的资源，但线程之间可以共享资源，所以线程更加节省内存。 Mach的线程使得程序员可以编写并发运行的程序，而这些程序既可以运行在单处理器的机器上，也可以运行在多处理器的机器中。另外，在单处理器环境中，当应用程序执行容易引起阻塞和延迟的操作时，线程可以提高效率。

20.2　一个简单的例子

用子函数pthread_create创建一个新的线程。它有四个参数：一个用来保存线程的线程变量、一个线程属性、当线程执行时要调用的函数和一个此函数的参数。例如：

```
pthread_t           a_thread;
pthread_attr_t      a_thread_attribute;
void                thread_function(void *argument);
char                *some_argument;

pthread_create( &a_thread, a_thread_attribute, (void *)&thread_function,
        (void *) &some_argument);
```

线程属性只指明了需要使用的最小的堆栈大小。在以后的程序中，线程的属性可以指定其他的值，但现在大部分的程序可以使用缺省值。不像 UNIX系统中使用 fork系统调用创建的进程，它们和它们的父进程使用同一个执行点，线程使用在 pthread_create中的参数指明要开始执行的函数。

现在我们可以编制第一个程序了。我们编制一个多线程的应用程序，在标准输出中打印“ Hello World ”。首先我们需要两个线程变量，一个新线程开始执行时可以调用的函数。我们还需要指明每一个线程应该打印的信息。一个做法是把要打印的字符串分开，给每一个线程一个字符串作为开始的参数。请看下面的代码：

```
void print_message_function( void *ptr );
```

```
main()
{
  pthread_t thread1, thread2;
  char *message1 = "Hello";
  char *message2 = "World";

  pthread_create( &thread1, pthread_attr_default,
            (void*)&print_message_function, (void*) message1);
  pthread_create(&thread2, pthread_attr_default,
            (void*)&print_message_function, (void*) message2);

  exit(0);
}

void print_message_function( void *ptr )
{
  char *message;
  message = (char *) ptr;
  printf("%s ", message);
}
```

程序通过调用pthread_create创建第一个线程，并将“ Hello ”作为它的启动参数。第二个线程的参数是“ World ”。当第一个线程开始执行时，它使用参数“ Hello ”执行函数print_message_function。它在标准输出中打印“ Hello ”，然后结束对函数的调用。线程当离开它的初始化函数时就将终止，所以第一个线程在打印完“ Hello ”后终止。当第二个线程执行时，它打印“ World ”然后终止。但这个程序有两个主要的缺陷。

首先也是最重要的是线程是同时执行的。这样就无法保证第一个线程先执行打印语句。所以你很可能在屏幕上看到“ World Hello ”，而不是“ Hello World ”。请注意对exit的调用是父线程在主程序中使用的。这样，如果父线程在两个子线程调用打印语句之前调用 exit，那么将不会有任何的打印输出。这是因为 exit函数将会退出进程，同时释放任务，所以结束了所有的线程。任何线程(不论是父线程或者子线程)调用exit 都会终止所有其他线程。如果希望线程分别终止，可以使用pthread_exit函数。

我们可以使用一个办法弥补此缺陷。我们可以在父线程中插入一个延迟程序，给子线程足够的时间完成打印的调用。同样，在调用第二个之前也插入一个延迟程序保证第一个线程在第二个线程执行之前完成任务。

```
void print_message_function( void *ptr );

main()
{
  pthread_t thread1, thread2;
  char *message1 = "Hello ";
  char *message2 = "World";

  pthread_create( &thread1, pthread_attr_default,
            (void *) &print_message_function, (void *) message1);
  sleep(10);
  pthread_create(&thread2, pthread_attr_default,
            (void *) &print_message_function, (void *) message2);
```

```
    sleep(10);
    exit(0);
}

void print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s", message);
    pthread_exit(0);
}
```

这样是否达到了我们的要求了呢？不尽如此，因为依靠时间的延迟执行同步是不可靠的。这里遇到的情形和一个分布程序和共享资源的情形一样。共享的资源是标准的输出设备，分布计算的程序是三个线程。

其实这里还有另外一个错误。函数 sleep和函数exit一样和进程有关。当线程调用 sleep时，整个的进程都处于睡眠状态，也就是说，所有的三个线程都进入睡眠状态。这样我们实际上没有解决任何的问题。希望使一个线程睡眠的函数是 pthread_delay_np。例如让一个线程睡眠2秒钟，用如下程序：

```
    struct timespec delay;
    delay.tv_sec = 2;
    delay.tv_nsec = 0;
    pthread_delay_np( &delay );
```

20.3 线程同步

POSIX提供两种线程同步的方法，mutex和条件变量。mutex是一种简单的加锁的方法来控制对共享资源的存取。我们可以创建一个读 /写程序，它们共用一个共享缓冲区，使用 mutex来控制对缓冲区的存取。

```
void reader_function(void);
void writer_function(void);

 char buffer;
int buffer_has_item = 0;
pthread_mutex_t mutex;
struct timespec delay;

main()
{
    pthread_t reader;

    delay.tv_sec = 2;
    delay.tv_nsec = 0;

    pthread_mutex_init(&mutex, pthread_mutexattr_default);
    pthread_create( &reader, pthread_attr_default, (void*)&reader_function,
                NULL);
    writer_function();
}
```

```
void writer_function(void)
{
  while(1)
  {
      pthread_mutex_lock( &mutex );
      if ( buffer_has_item == 0 )
      {
          buffer = make_new_item();
          buffer_has_item = 1;
      }
      pthread_mutex_unlock( &mutex );
      pthread_delay_np( &delay );
  }
}

void reader_function(void)
{
  while(1)
  {
      pthread_mutex_lock( &mutex );
      if ( buffer_has_item == 1)
      {
          consume_item( buffer );
          buffer_has_item = 0;
      }
      pthread_mutex_unlock( &mutex );
      pthread_delay_np( &delay );
  }
}
```

在上面的程序中，我们假定缓冲区只能保存一条信息，这样缓冲区只有两个状态，有一条信息或者没有信息。使用延迟是为了避免一个线程永远占有 mutex。

但mutex的缺点在于它只有两个状态，锁定和非锁定。 POSIX的条件变量通过允许线程阻塞和等待另一个线程的信号方法，从而弥补了 mutex的不足。当接受到一个信号时，阻塞线程将会被唤起，并试图获得相关的mutex的锁。

20.4 使用信号量协调程序

我们可以使用信号量重新看一下上面的读 /写程序。涉及信号量的操作是 semaphore_up、semaphore_down、semaphore_init、semaphore_destroy和 semaphore_decrement。所有这些操作都只有一个参数，一个指向信号量目标的指针。

```
void reader_function(void);
void writer_function(void);

char buffer;
Semaphore writers_turn;
Semaphore readers_turn;

main()
```

```
{
    pthread_t reader;

    semaphore_init( &readers_turn );
    semaphore_init( &writers_turn );

    /* writer must go first */
    semaphore_down( &readers_turn );

    pthread_create( &reader, pthread_attr_default,
                (void *)&reader_function, NULL);
    writer_function();
}

void writer_function(void)
{
    while(1)
    {
        semaphore_down( &writers_turn );
        buffer = make_new_item();
        semaphore_up( &readers_turn );
    }
}

void reader_function(void)
{
    while(1)
    {
        semaphore_down( &readers_turn );
        consume_item( buffer );
        semaphore_up( &writers_turn );
    }
}
```

这个例子也没有完全地利用一般信号量的所有函数。我们可以使用信号量重新编写“Hello world”的程序：

```
void print_message_function( void *ptr );

Semaphore child_counter;
Semaphore worlds_turn;

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Hello";
    char *message2 = "World";

    semaphore_init( &child_counter );
    semaphore_init( &worlds_turn );

    semaphore_down( &worlds_turn ); /* world goes second */
```

```
    semaphore_decrement( &child_counter ); /* value now 0 */
    semaphore_decrement( &child_counter ); /* value now -1 */
    /*
     * child_counter now must be up-ed 2 times for a thread blocked on it
     * to be released
     *
     */

    pthread_create( &thread1, pthread_attr_default,
            (void *) &print_message_function, (void *) message1);

    semaphore_down( &worlds_turn );

    pthread_create(&thread2, pthread_attr_default,
            (void *) &print_message_function, (void *) message2);

    semaphore_down( &child_counter );

    /* not really necessary to destroy since we are exiting anyway */
    semaphore_destroy ( &child_counter );
    semaphore_destroy ( &worlds_turn );
    exit(0);
}

void print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s ", message);
    fflush(stdout);
    semaphore_up( &worlds_turn );
    semaphore_up( &child_counter );
    pthread_exit(0);
}
```

信号量child_counter用来强迫父线程阻塞，直到两个子线程执行完 printf语句和其后的semaphore_up(&child_counter)语句才继续执行。

20.5 信号量的实现

20.5.1 Semaphore.h

```
#ifndef SEMAPHORES
#define SEMAPHORES

#include
#include


typedef struct Semaphore
```

```
{
    int         v;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
}
Semaphore;


int         semaphore_down (Semaphore * s);
int         semaphore_decrement (Semaphore * s);
int         semaphore_up (Semaphore * s);
void        semaphore_destroy (Semaphore * s);
void        semaphore_init (Semaphore * s);
int         semaphore_value (Semaphore * s);
int         tw_pthread_cond_signal (pthread_cond_t * c);
int         tw_pthread_cond_wait (pthread_cond_t * c, pthread_mutex_t * m);
int         tw_pthread_mutex_unlock (pthread_mutex_t * m);
int         tw_pthread_mutex_lock (pthread_mutex_t * m);
void        do_error (char *msg);

#endif
```

20.5.2 Semaphore.c

```
#include "semaphore.h"

/*
 * function must be called prior to semaphore use.
 *
 */
void
semaphore_init (Semaphore * s)
{
    s->v = 1;
    if (pthread_mutex_init (&(s->mutex), pthread_mutexattr_default) == -1)
     do_error ("Error setting up semaphore mutex");

    if (pthread_cond_init (&(s->cond), pthread_condattr_default) == -1)
     do_error ("Error setting up semaphore condition signal");
}

/*
 * function should be called when there is no longer a need for
 * the semaphore.
 *
 */
void
semaphore_destroy (Semaphore * s)
{
    if (pthread_mutex_destroy (&(s->mutex)) == -1)
     do_error ("Error destroying semaphore mutex");
```

```
    if (pthread_cond_destroy (&(s->cond)) == -1)
     do_error ("Error destroying semaphore condition signal");
}

/*
 * function increments the semaphore and signals any threads that
 * are blocked waiting a change in the semaphore.
 *
 */
int
semaphore_up (Semaphore * s)
{
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));

    (s->v)++;
    value_after_op = s->v;

    tw_pthread_mutex_unlock (&(s->mutex));
    tw_pthread_cond_signal (&(s->cond));

    return (value_after_op);
}

/*
 * function decrements the semaphore and blocks if the semaphore is
 * <= 0 until another thread signals a change.
 *
 */
int
semaphore_down (Semaphore * s)
{
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));
    while (s->v <= 0)
    {
     tw_pthread_cond_wait (&(s->cond), &(s->mutex));
    }

    (s->v)--;
    value_after_op = s->v;

    tw_pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
}

/*
```

```
 * function does NOT block but simply decrements the semaphore.
 * should not be used instead of down -- only for programs where
 * multiple threads must up on a semaphore before another thread
 * can go down, i.e., allows programmer to set the semaphore to
 * a negative value prior to using it for synchronization.
 *
 */
int
semaphore_decrement (Semaphore * s)
{
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));
    s->v--;
    value_after_op = s->v;
    tw_pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
}

/*
 * function returns the value of the semaphore at the time the
 * critical section is accessed.  obviously the value is not guarenteed
 * after the function unlocks the critical section.  provided only
 * for casual debugging, a better approach is for the programmar to
 * protect one semaphore with another and then check its value.
 * an alternative is to simply record the value returned by semaphore_up
 * or semaphore_down.
 *
 */
int
semaphore_value (Semaphore * s)
{
    /* not for sync */
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));
    value_after_op = s->v;
    tw_pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
}

/* ------------------------------------------------------------------- */
/* The following functions replace standard library functions in that  */
/* they exit on any error returned from the system calls.  Saves us    */
/* from having to check each and every call above.                     */
/* ------------------------------------------------------------------- */

int
tw_pthread_mutex_unlock (pthread_mutex_t * m)
```

```
{
    int         return_value;
    if ((return_value = pthread_mutex_unlock (m)) == -1)
     do_error ("pthread_mutex_unlock");
    return (return_value);
}
int
tw_pthread_mutex_lock (pthread_mutex_t * m)
{
    int         return_value;
    if ((return_value = pthread_mutex_lock (m)) == -1)
     do_error ("pthread_mutex_lock");
    return (return_value);
}
int
tw_pthread_cond_wait (pthread_cond_t * c, pthread_mutex_t * m)
{
    int         return_value;
    if ((return_value = pthread_cond_wait (c, m)) == -1)
     do_error ("pthread_cond_wait");
    return (return_value);
}
int
tw_pthread_cond_signal (pthread_cond_t * c)
{
    int         return_value;
    if ((return_value = pthread_cond_signal (c)) == -1)
     do_error ("pthread_cond_signal");
    return (return_value);
}
/*
 * function just prints an error message and exits
 *
 */
void
do_error (char *msg)
{
    perror (msg);
    exit (1);
}
```

第21章 Linux系统网络编程

本章介绍Linux系统网络编程的内容，如套接口的概念与使用、网络编程的结构等。

21.1 什么是套接口

简单地说，套接口就是一种使用UNIX系统中的文件描述符和系统进程通信的一种方法。

因为在UNIX系统中，所有的I/O操作都是通过读写文件描述符而产生的。文件描述符就是一个和打开的文件相关连的整数。但文件可以是一个网络连接、一个 FIFO、一个管道、一个终端、一个真正存储在磁盘上的文件或者 UNIX系统中的任何其他的东西。所以，如果你希望通过Internet和其他的程序进行通信，你只有通过文件描述符。

使用系统调用socket()，你可以得到socket()描述符。然后你可以使用send() 和 recv()调用而与其他的程序通信。你也可以使用一般的文件操作来调用 read() 和 write()而与其他的程序进行通信，但send() 和 recv()调用可以提供一种更好的数据通信的控制手段。下面我们讨论 Internet套接口的使用方法。

21.2 两种类型的Internet套接口

有两种最常用的 Internet 套接口，“ 数据流套接口 ”和“ 数据报套接口 ”，以后我们用“ SOCK_STREAM ” 和 “ SOCK_DGRAM ”分别代表上面两种套接口。数据报套接口有时也叫做“ 无连接的套接口 ”。

数据流套接口是可靠的双向连接的通信数据流。如果你在套接口中以“ 1, 2 ”的顺序放入两个数据，它们在另一端也会以“ 1, 2 ”的顺序到达。它们也可以被认为是无错误的传输。

经常使用的telnet应用程序就是使用数据流套接口的一个例子。使用 HTTP的WWW浏览器也使用数据流套接口来读取网页。事实上，如果你使用 telnet 登录到一个WWW站点的80端口，然后键入“ GET 网页名 ”，你将可以得到这个HTML页。数据流套接口使用TCP得到这种高质量的数据传输。数据报套接口使用UDP，所以数据报的顺序是没有保障的。数据报是按一种应答的方式进行数据传输的。

21.3 网络协议分层

由于网络中的协议是分层的，所以上层的协议是依赖于下一层所提供的服务的。也就是说，你可以在不同的物理网络中使用同样的套接口程序，因为下层的协议对你来说是透明的。UNIX系统中的网络协议是这样分层的：

- 应用层 (telnet、ftp等)。
- 主机到主机传输层 (TCP、UDP)。
- Internet层 (IP和路由)。
- 网络访问层(网络、数据链路和物理层)。

21.4 数据结构

下面我们要讨论使用套接口编写程序可能要用到的数据结构。

首先是套接口描述符。一个套接口描述符只是一个整型的数值： int。

第一个数据结构是struct sockaddr，这个数据结构中保存着套接口的地址信息。

```
struct sockaddr {
    unsigned short    sa_family;    /* address family, AF_xxx       */
    char              sa_data[14];  /* 14 bytes of protocol address */
};
```

sa_family 中可以是其他的很多值，但在这里我们把它赋值为“ AF_INET ”。sa_data包括一个目的地址和一个端口地址。

你也可以使用另一个数据结构sockaddr_in，如下所示：

```
struct sockaddr_in {
    short int          sin_family;  /* Address family         */
    unsigned short int sin_port;    /* Port number            */
    struct in_addr     sin_addr;    /* Internet address       */
    unsigned char      sin_zero[8]; /* Same size as struct sockaddr */
};
```

这个数据结构使得使用其中的各个元素更为方便。要注意的是 sin_zero应该使用bzero() 或者 memset()而设置为全0。另外，一个指向sockaddr_in数据结构的指针可以投射到一个指向数据结构sockaddr的指针，反之亦然。

21.5 IP地址和如何使用IP地址

有一系列的程序可以使你处理IP地址。

首先，你可以使用inet_addr()程序把诸如“ 132.241.5.10 ”形式的IP地址转化为无符号的整型数。

```
ina.sin_addr.s_addr = inet_addr("132.241.5.10");
```

如果出错， inet_addr()程序将返回-1。

也可以调用inet_ntoa()把地址转换成数字和句点的形式：

```
printf("%s",inet_ntoa(ina.sin_addr));
```

这将会打印出IP地址。它返回的是一个指向字符串的指针。

21.5.1 socket()

我们使用系统调用socket()来获得文件描述符：

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

int socket(int domain, int type, int protocol);
```

第一个参数domain设置为“ AF_INET ”。第二个参数是套接口的类型：SOCK_STREAM 或 SOCK_DGRAM。第三个参数设置为0。

系统调用socket()只返回一个套接口描述符，如果出错，则返回 -1。

21.5.2 bind()

一旦你有了一个套接口以后，下一步就是把套接口绑定到本地计算机的某一个端口上。但如果你只想使用connect()则无此必要。

下面是系统调用bind()的使用方法：

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

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

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

第一个参数sockfd 是由socket()调用返回的套接口文件描述符。第二个参数my_addr 是指向数据结构sockaddr的指针。数据结构 sockaddr中包括了关于你的地址、端口和 IP地址的信息。第三个参数addrlen可以设置成sizeof(struct sockaddr)。

下面是一个例子：

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

#define MYPORT 3490

main()
{
    int sockfd;
    struct sockaddr_in my_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */

    my_addr.sin_family = AF_INET;     /* host byte order */
    my_addr.sin_port = htons(MYPORT); /* short, network byte order */
    my_addr.sin_addr.s_addr = inet_addr("132.241.5.10");
    bzero(&(my_addr.sin_zero), 8);    /* zero the rest of the struct */

    /* don't forget your error checking for bind(): */
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    .
    .
    .
```

如果出错，bind() 也返回-1。

如果你使用connect()系统调用，那么你不必知道你使用的端口号。当你调用 connect()时，它检查套接口是否已经绑定，如果没有，它将会分配一个空闲的端口。

21.5.3 connect()

系统调用connect()的用法如下：

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

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

第一个参数还是套接口文件描述符，它是由系统调用 socket()返回的。第二个参数是serv_addr是指向数据结构sockaddr的指针，其中包括目的端口和IP地址。第三个参数可以使用sizeof(struct sockaddr)而获得。下面是一个例子：

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

#define DEST_IP   "132.241.5.10"
```

```
#define DEST_PORT 23

main()
{
        int sockfd;
        struct sockaddr_in dest_addr;   /* will hold the destination addr */

        sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */

        dest_addr.sin_family = AF_INET;         /* host byte order */
        dest_addr.sin_port = htons(DEST_PORT); /* short, network byte order */
        dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
        bzero(&(dest_addr.sin_zero), 8);        /* zero the rest of the struct */

        /* don't forget to error check the connect()! */
        connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
        .
        .
        .
```

同样，如果出错， connect()将会返回-1。

21.5.4 listen()

如果你希望不连接到远程的主机，也就是说你希望等待一个进入的连接请求，然后再处理它们。这样，你通过首先调用listen()，然后再调用accept()来实现。

系统调用listen()的形式如下：

```
int listen(int sockfd, int backlog);
```

第一个参数是系统调用 socket()返回的套接口文件描述符。第二个参数是进入队列中允许的连接的个数。进入的连接请求在使用系统调用 accept()应答之前要在进入队列中等待。这个值是队列中最多可以拥有的请求的个数。大多数系统的缺省设置为 20。你可以设置为5或者10。

当出错时，listen()将会返回-1值。

当然，在使用系统调用 listen()之前，我们需要调用 bind()绑定到需要的端口，否则系统内核将会让我们监听一个随机的端口。所以，如果你希望监听一个端口，下面是应该使用的系统调用的顺序：

```
socket();
bind();
listen();
/* accept() goes here */
```

21.5.5 accept()

系统调用 accept()比较起来有点复杂。在远程的主机可能试图使用 connect()连接你使用 listen()正在监听的端口。但此连接将会在队列中等待，直到使用 accept()处理它。调用accept()之后，将会返回一个全新的套接口文件描述符来处理这个单个的连接。这样，对于同一个连接来说，你就有了两个文件描述符。原先的一个文件描述符正在监听你指定的端口，新的文件描述符可以用来调用send()和recv()。

调用的例子如下：

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

int accept(int sockfd, void *addr, int *addrlen);
```

第一个参数是正在监听端口的套接口文件描述符。第二个参数 addr是指向本地的数据结构sockaddr_in的指针。调用connect()中的信息将存储在这里。通过它你可以了解哪个主机在哪个端口呼叫你。第三个参数同样可以使用sizeof(struct sockaddr_in)来获得。

如果出错，accept()也将返回-1。下面是一个简单的例子：

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

#define MYPORT 3490    /* the port users will be connecting to */

#define BACKLOG 10     /* how many pending connections queue will hold */

main()
{
        int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
        struct sockaddr_in my_addr;    /* my address information */
        struct sockaddr_in their_addr; /* connector's address information */
        int sin_size;

        sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */

        my_addr.sin_family = AF_INET;              /* host byte order */
        my_addr.sin_port = htons(MYPORT);          /* short, network byte order */
        my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
        bzero(&(my_addr.sin_zero), 8);             /* zero the rest of the struct */

        /* don't forget your error checking for these calls: */
        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

        listen(sockfd, BACKLOG);

 sin_size = sizeof(struct sockaddr_in);
 new_fd = accept(sockfd, &their_addr, &sin_size);
.
.
.
```

下面，我们将可以使用新创建的套接口文件描述符 new_fd来调用send()和recv()。

21.5.6 send() 和 recv()

系统调用send()的用法如下：

```
int send(int sockfd, const void *msg, int len, int flags);
```

第一个参数是你希望给发送数据的套接口文件描述符。它可以是你通过 socket()系统调用返回的，也可以是通过accept()系统调用得到的。第二个参数是指向你希望发送的数据的指针。第三个参数是数据的字节长度。第四个参数标志设置为 0。

下面是一个简单的例子：

```
char *msg = "Beej was here!";
int len, bytes_sent;
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.
```

系统调用send()返回实际发送的字节数，这可能比你实际想要发送的字节数少。如果返回的字节数比要发送的字节数少，你在以后必须发送剩下的数据。当 send()出错时，将返回 -1。系统调用recv()的使用方法和send()类似：

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

第一个参数是要读取的套接口文件描述符。第二个参数是保存读入信息的地址。第三个参数是缓冲区的最大长度。第四个参数设置为0。

系统调用recv()返回实际读取到缓冲区的字节数，如果出错则返回 -1。

这样使用上面的系统调用，你可以通过数据流套接口来发送和接受信息。

21.5.7 sendto() 和 recvfrom()

因为数据报套接口并不连接到远程的主机上，所以在发送数据包之前，我们必须首先给出目的地址，请看：

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
const struct sockaddr *to, int tolen);
```

除了两个参数以外，其他的参数和系统调用 send()时相同。参数to是指向包含目的IP地址和端口号的数据结构sockaddr的指针。参数tolen可以设置为sizeof(struct sockaddr)。

系统调用sendto()返回实际发送的字节数，如果出错则返回 -1。

系统调用recvfrom()的使用方法也和recv()的十分近似：

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags
        struct sockaddr *from, int *fromlen);
```

参数from是指向本地计算机中包含源 IP地址和端口号的数据结构 sockaddr的指针。参数fromlen设置为sizeof(struct sockaddr)。

系统调用recvfrom()返回接收到的字节数，如果出错则返回 -1。

21.5.8 close() 和 shutdown()

你可以使用close()调用关闭连接的套接口文件描述符：

```
close(sockfd);
```

这样就不能再对此套接口做任何的读写操作了。

使用系统调用shutdown()，可有更多的控制权。它允许你在某一个方向切断通信，或者切断双方的通信：

```
int shutdown(int sockfd, int how);
```

第一个参数是你希望切断通信的套接口文件描述符。第二个参数 how值如下：

0 —— Further receives are disallowed

1 —— Further sends are disallowed

2 —— Further sends and receives are disallowed (like close())

shutdown() 如果成功则返回0，如果失败则返回-1。

21.5.9 getpeername()

这个系统的调用十分简单。它将告诉你是谁在连接的另一端：

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

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

第一个参数是连接的数据流套接口文件描述符。第二个参数是指向包含另一端的信息的数据结构sockaddr的指针。第三个参数可以设置为sizeof(struct sockaddr)。

如果出错，系统调用将返回-1。

一旦你获得了它们的地址，你可以使用inet_ntoa() 或者 gethostbyaddr()来得到更多的信息。

21.5.10 gethostname()

系统调用gethostname()比系统调用getpeername()还简单。它返回程序正在运行的计算机的名字。系统调用gethostbyname()可以使用这个名字来决定你的机器的IP地址。

下面是一个例子：

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

如果成功，gethostname将返回0。如果失败，它将返回-1。

21.6 DNS

DNS 代表“ Domain Name Service ”，即域名服务器。它可以把域名翻译成相应的 IP地址。你可以使用此IP地址调用bind()、connect()、sendto()或者用于其他的地方。

系统调用gethostbyname()可以完成这个函数：

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

它返回一个指向数据结构hostent的指针，数据结构hostent如下：

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

h_name —— 主机的正式名称。

h_aliases —— 主机的别名。

h_addrtype —— 将要返回的地址的类型，一般是 AF_INET。

h_length —— 地址的字节长度。

h_addr_list —— 主机的网络地址。

h_addr ——h_addr_list中的第一个地址。

系统调用gethostbyname()返回一个指向填充好的数据结构 hostent的指针。当发生错误时，则返回一个NULL指针。下面是一个实际例子：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{
    struct hostent *h;

    if (argc != 2) {  /* error check the command line */
        fprintf(stderr,"usage: getip address\n");
        exit(1);
    }

    if ((h=gethostbyname(argv[1])) == NULL) {  /* get the host info */
        herror("gethostbyname");
        exit(1);
    }

    printf("Host name  : %s\n", h->h_name);
    printf("IP Address : %s\n",inet_ntoa(*((struct in_addr *)h->h_addr)));

    return 0;
}
```

在使用gethostbyname()时，你不能使用perror()来打印错误信息。你应该使用的是系统调用herror()。

21.7 客户机/服务器模式

在网络上大部分的通信都是在客户机/服务器模式下进行的。例如telnet。当你使用telnet连接到远程主机的端口 23时，主机上的一个叫做 telnetd的程序就开始运行。它处理所有进入的telnet连接，为你设置登录提示符等。

应当注意的是客户机/服务器模式可以使用SOCK_STREAM、SOCK_DGRAM或者任何其他的方式。例如telnet/telnetd、ftp/ftpd和 bootp/bootpd。每当你使用ftp时，远程计算机都在运行一个ftpd为你服务。

一般情况下，一台机器上只有一个服务器程序，它通过使用 fork()来处理多个客户端程序的请求。最基本的处理方法是：服务器等待连接，使用 accept()接受连接，调用fork()生成一个子进程处理连接。

21.8 简单的数据流服务器程序

此服务器程序所作的事情就是通过一个数据流连接发送字符串“ Hello, World!\n ”。你可以在一个窗口上运行此程序，然后在另一个窗口使用 telnet：

```
$ telnet remotehostname 3490
```

其中，remotehostname是你运行的机器名。下面是此程序的代码：

```
#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 MYPORT 3490    /* the port users will be connecting to */

#define BACKLOG 10     /* how many pending connections queue will hold */

main()
{
    int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;    /* my address information */
    struct sockaddr_in their_addr; /* connector's address information */
    int sin_size;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;         /* host byte order */
    my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
    my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
    bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) \
                                                  == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

while(1) {  /* main accept() loop */
        sin_size = sizeof(struct sockaddr_in);
        if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, \
                                        &sin_size)) == -1) {
            perror("accept");
            continue;
        }
```

```
            printf("server: got connection from %s\n", \
                                    inet_ntoa(their_addr.sin_addr));
            if (!fork()) { /* this is the child process */
                if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                    perror("send");
                close(new_fd);
                exit(0);
            }
            close(new_fd);  /* parent doesn't need this */

            while(waitpid(-1,NULL,WNOHANG) > 0); /* clean up child processes */
        }
    }
```

你也可以使用下面的客户机程序从服务器上得到字符串。

21.9 简单的数据流客户机程序

客户机所做的是连接到你在命令行中指定的主机的 3490端口。它读取服务器发送的字符串。

下面是客户机程序的代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 3490    /* the port client will be connecting to */

#define MAXDATASIZE 100 /* max number of bytes we can get at once */

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in their_addr; /* connector's address information */

    if (argc != 2) {
        fprintf(stderr,"usage: client hostname\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  /* get the host info */
        herror("gethostbyname");
        exit(1);
    }
```

```
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    their_addr.sin_family = AF_INET;      /* host byte order */
    their_addr.sin_port = htons(PORT);    /* short, network byte order */
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(their_addr.sin_zero), 8);     /* zero the rest of the struct */

    if (connect(sockfd, (struct sockaddr *)&their_addr, \
                                sizeof(struct sockaddr)) == -1) {
        perror("connect");
        exit(1);
    }

    if ((numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1) {
        perror("recv");
        exit(1);
    }

    buf[numbytes] = '\0';

    printf("Received: %s",buf);

    close(sockfd);

    return 0;
}
```

如果你在运行服务器程序之前运行客户机程序，则将会得到一个“ Connection refused ”的信息。

21.10 数据报套接口

程序listener在机器中等待端口4950到来的数据包。程序talker向指定的机器的4950端口发送数据包。

下面是listener.c的代码：

```
#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 MYPORT 4950    /* the port users will be sending to */

#define MAXBUFLEN 100
```

```
main()
{
        int sockfd;
        struct sockaddr_in my_addr;    /* my address information */
        struct sockaddr_in their_addr; /* connector's address information */
        int addr_len, numbytes;
        char buf[MAXBUFLEN];

        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }

        my_addr.sin_family = AF_INET;         /* host byte order */
        my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
        my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
        bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

        if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) \
                                                    == -1) {
            perror("bind");
            exit(1);
        }

        addr_len = sizeof(struct sockaddr);
        if ((numbytes=recvfrom(sockfd, buf, MAXBUFLEN, 0, \
                    (struct sockaddr *)&their_addr, &addr_len)) == -1) {
            perror("recvfrom");
            exit(1);
        }

        printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));
        printf("packet is %d bytes long\n",numbytes);
        buf[numbytes] = '\0';
        printf("packet contains \"%s\"\n",buf);

        close(sockfd);
}
```

下面是talker.c的代码：

```
#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 MYPORT 4950    /* the port users will be sending to */
```

```
int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; /* connector's address information */
    struct hostent *he;
    int numbytes;

    if (argc != 3) {
        fprintf(stderr,"usage: talker hostname message\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  /* get the host info */
        herror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    their_addr.sin_family = AF_INET;      /* host byte order */
    their_addr.sin_port = htons(MYPORT);  /* short, network byte order */
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(their_addr.sin_zero), 8);     /* zero the rest of the struct */

    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0, \
         (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1) {
        perror("sendto");
        exit(1);
    }

    printf("sent %d bytes to %s\n",numbytes,inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}
```

你可以在一台机器上运行 listener程序，在另一台机器上运行 talker程序，然后观察它们之间的通信。

21.11 阻塞

当使用上面的 listener程序时，此程序在等待直到一个数据包到来。这是因为它调用了 recvform()，如果没有数据，recvform()就一直阻塞，直到有数据到来。

很多函数都有阻塞。系统调用 accept()阻塞，所有的类似 recv*()的函数也可以阻塞。它们之所以可以阻塞是因为系统内核允许它们阻塞。当你第一次创建一个套接口文件描述符时，系

统内核将它设置为可以阻塞。如果你不希望套接口阻塞，你可以使用系统调用 fcntl()：

```
#include <unistd.h>
#include <fcntl.h>
.
.
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
```

如果你设置为不阻塞，那么就得频繁地询问套接口以便检查有无信息到来。如果你试图读取一个没有阻塞的套接口，同时它又没有数据，那么你将得到 -1。

询问套接口以检查有无信息得到来可能会占用太多的 CPU时间。另一个可以使用的方法是 select()。

select()用于同步 I/O多路复用。这个系统调用十分有用。考虑一下下面的情况：你是一个服务器，你希望监听进入的连接，同时还一直从已有的连接中读取信息。

也许你认为可以使用一个 accept()调用和几个recv()调用。但如果调用 accept()阻塞了怎么办？如果在这时你希望调用recv()接受数据呢？

系统调用select()使得你可以同时监视几个套接口。它可以告诉你哪一个套接口已经准备好了以供读取，哪一个套接口已经可以写入。

下面是select()的用法：

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

int select(int numfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

此函数监视几个文件描述符，特别是 readfds、writefds和 exceptfds。如果你希望检查是否可以从标准输入中和一些其他的套接口文件描述符 sockfd中读取数据，只需把文件描述符 0和sockfd添加到readfds中。参数numfds应该设置为最高的文件描述符的值加1。

当select()返回时，readfds将会被修改以便反映你选择的那一个文件描述符已经准备好了以供读取。你可以使用FD_ISSET()测试。

FD_ZERO(fd_set *set) —— 清除文件描述符集。

FD_SET(int fd, fd_set *set) —— 把fd 添加到文件描述符集中 。

FD_CLR(int fd, fd_set *set) —— 把fd 从文件描述符中移走 。

FD_ISSET(int fd, fd_set *set) —— 检测fd 是否在文件描述符集中。

数据结构timeval包含下面的字段：

```
struct timeval {
    int tv_sec;     /* seconds */
    int tv_usec;    /* microseconds */
};
```

把tv_sec设置成需要等待的时间秒数， tv_usec设置成需要等待的微秒数。一秒中包括1 000 000μs 。下面的程序等待2.5s：

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

```
#define STDIN 0  /* file descriptor for standard input */

main()
{
        struct timeval tv;
        fd_set readfds;

        tv.tv_sec = 2;
        tv.tv_usec = 500000;

        FD_ZERO(&readfds);
        FD_SET(STDIN, &readfds);

        /* don't care about writefds and exceptfds: */
        select(STDIN+1, &readfds, NULL, NULL, &tv);

        if (FD_ISSET(STDIN, &readfds))
            printf("A key was pressed!\n");
        else
            printf("Timed out.\n");
}
```

第22章 Linux I/O端口编程

本章介绍有关Linux I/O端口编程的内容，如在C语言下使用I/O端口、硬件中断与IMA存取等方面的内容。

22.1 如何在 C 语言下使用I/O端口

22.1.1 一般的方法

用来存取 I/O 端口的子过程都放在文件 /usr/include/asm/io.h 里(或放在内核源代码程序的linux/include/asm-i386/io.h 文件里)。这些子过程是以嵌入宏的方式写成的，所以使用时只要以#include<asm/io.h> 的方式引用就够了，不需要附加任何函数库。

因为 gcc以及 egcs的限制，你在编译任何使用到这些子过程的源代码时必须打开最优化选项 (gcc -O1或较高层次的)，或者在做 #include <asm/io.h> 这个动作前使用 #define extern 将 extern 定义成空白。

为了除错的目的，你编译时可以使用 gcc -g -O (至少现在的 gcc 版本是这样)，但是最优化之后有时可能会让调试器的行为变得有点奇怪。如果这个状况对你而言是个困扰，你可以将所有使用到 I/O端口的子过程集中放在一个文件里，并只在编译该文件时打开最优化选项。

在你存取任何 I/O 端口之前，你必须让程序有如此做的权限。要完成这个目的，你可以在程序一开始的地方 (但是，要在任何 I/O 端口存取动作之前) 调用 ioperm()这个函数 (该函数在文件 unistd.h中，并且被定义在 内核中)。 使用语法是 ioperm(from，num，turn_on)，其中 from 是第一个允许存取的 I/O 端口地址，num是接着连续存取 I/O 端口地址的数目。 例如，ioperm(0x300，5，1)的意思就是说允许存取端口 0x300 到 0x304 (一共五个端口地址)。

而最后一个参数是一个布尔代数值，用来指定是否给予程序存取 I/O 端口的权限 (true (1)) 或者除去存取的权限 (false (0))。 你可以多次调用函数 ioperm()以便使用多个不连续的端口地址。

你的程序必须拥有 root 权限才能调用函数 ioperm()；所以你如果不是以 root身份执行该程序，就得将该程序设置成 root。 当你调用过函数 ioperm() 打开I/O 端口的存取权限后你便可以拿掉 root 的权限。在你的程序结束之后并不特别要求你以 ioperm(...，0) 这个方式拿掉 I/O 端口的存取权限；因为当你的程序执行完毕之后，这个动作会自动完成。

调用函数 setuid() 将目前执行程序的有效用户识别码 (ID) 设定成非 root的用户，并不影响其先前以 ioperm() 的方式所取得的 I/O 端口存取权限，但是调用函数fork() 的方式却会有所影响 (虽然父进程保有存取权限，但是子进程却无法取得存取权限)。

函数 ioperm() 只能让你取得端口地址 0x000 到 0x3ff 的存取权限；至于较高地址的端口，你得使用函数 iopl() (该函数让你一次可以存取所有的端口地址)。将权限等级参数值设为 3 (例如，iopl(3))，以便你的程序能够存取所有的I/O 端口(因此要小心，如果存取到错误的端口地址将对你的计算机造成各种不可预期的损害。同样地，调用函数 iopl() 你得拥有 root 的权限。

接着，我们来实际地存取 I/O 端口。要从某个端口地址输入一个字节(8位)的信息，你得调用函数 inb(port) ，该函数会传回所取得的一个字节的信息。要输出一个字节的信息，你得调

用函数 outb(value, port) (请记住参数的次序)。要从某两个端口地址 x 和 x+1 (两个字节组成一个字，故使用组合语言指令inw)输入一个字 (16 个 bit) 的信息，你得调用函数 inw(x); 要输出一个字的信息到两个端口地址，你得调用函数 outw(value, x)。如果你不确定使用哪个端口指令(字节或字)，你大概须要 inb() 与 outb() 这两个端口指令，因为大多数的设备都是采用字节大小的端口存取方式来设计的。注意所有的端口存取指令都至少需要大约 1μs 的时间。

如果你使用的是 inb_p()、outb_p()、inw_p()以及 outw_p() 等宏指令，在你对端口地址存取动作之后只需很短的(大约为1μs) 延迟时间就可以完成;你也可以让延迟时间变成大约 4μs ，方法是在使用 #include <asm/io.h> 之前使用#define REALLY_SLOW_IO。 这些宏指令通常 (除非你使用的是 #define SLOW_IO_BY_JUMPING，这个方法可能不太准确) 会利用输出信息到端口地址 0x80以便达到延迟时间的目的，所以你得先以函数 ioperm() 取得端口地址 0x80 的使用权限(输出信息到端口地址 0x80 不应该会对系统的其他部分造成影响)。 至于其他通用的延迟时间的方法，请参考下面的内容。

22.1.2 另一个替代方法: /dev/port

另一个存取 I/O 端口的方法是以函数 open() 打开文件 /dev/port(一个字符设备，主设备编号为 1，次设备编号为 4)， 以便执行读与(/或)写的动作(注意标准输出入 函数 f*() 有内部的缓冲 ，所以要避免使用)。接着使用 lseek() 函数以便在该字符设备文件中找到某个字节信息的正确位置(文件位置 0 = 端口地址 0x00，文件位置 1 = 端口地址 0x01，以此类推)，然后你可以使用read() 或 write() 函数对某个端口地址做读或写一个字节的动作。

这个方法就是在你的程序里使用 read/write 函数来存取 /dev/port字符设备文件。 这个方法的执行速度或许比前面所讲的一般方法还慢，但是不需要编译器的最优化函数，也不需要使用函数 ioperm() 。 如果你允许非 root 用户或群组存取/dev/port 字符设备，操作时就不需拥有 root 权限。但是，对于系统安全而言，这样做非常糟糕，因为它可能伤害到你的系统，或许，会有人因此而取得 root 的权限，利用 /dev/port 字符设备文件直接在硬盘、网络卡等设备上进行存取操作。

22.2 硬件中断 与 DMA 存取

你的程序如果在用户模式下执行，不可以直接使用硬件中断 (IRQ) 或DMA。你必须编写一个内核驱动程序。也就是说，你在用户模式中所写的程序无法控制硬件中断的产生。

22.3 高精确的时间

22.3.1 延迟时间

在用户模式中执行的进程不能精确地控制时间，因为 Linux 是个多用户的操作环境，在执行中的进程随时会因为各种原因被暂停大约 10ms到数秒 (在系统负荷非常高的时候)。 然而，对于大多数使用 I/O 端口的应用程序而言，这个延迟时间实际上算不了什么。要缩短延迟时间，你得使用函数 nice 将你在执行中的进程设定成高优先权(请参考 nice(2)使用说明文件)，或使用即时调度法 (real-time scheduling) (请看下面介绍)。

如果你想获得比在一般用户模式中执行的进程还要精确的时间，有一些方法可以让你在用户模式中做到“ 即时调度 ”的支持。Linux 2.x 版本的内核中有软件方式的即时调度支持。

1. 睡眠: sleep() 与 usleep()

现在，让我们开始进行较简单的时间函数调用。 想要延迟数秒的时间，最佳的方法大概是使用函数 sleep() 。 想要延迟至少数十毫秒的时间 (10 ms 似乎已是最短的延迟时间了)，函数 usleep() 应该可以使用。这些函数把 CPU 的使用权让给其他想要执行的进程，所以没有浪费掉CPU的时间。

如果让出 CPU 的使用权因而使得时间延迟了大约 50ms(这取决于处理器与机器的速度，以及系统的负荷)，那就浪费掉 CPU 太多的时间，因为Linux 的调度器 (scheduler) (单就 x86 结构而言) 在将控制权发还给你的进程之前通常至少要花费 10～30ms的时间。因此，短时间的延迟，使用函数usleep(3) 所得到的延迟结果通常会大于你在参数所指定的值，大约至少有 10 ms。

2. nanosleep()

在 Linux 2.0.x 一系列的内核发行版本中，有一个新的系统调用， nanosleep() (请参考 nanosleep(2) 的说明文件)，它让你能够休息或延迟一个短的时间 (数微秒或更多)。

3. 使用 I/O 端口延迟时间

另一个延迟数微秒的方法是使用 I/O 端口。 就是从端口地址 0x80 输入或输出任何字节的信息 (请参考前面) 等待的时间应该几乎只要 1μs 这要看你的处理器的类型与速度。如果要延迟数微秒的时间，你可以将这个动作多做几次。 在任何标准的机器上输出信息到该端口地址，应该不会有不良的后果 (而且有些内核的设备驱动程序也在使用它)。{in|out}[bw]_p() 等函数就是使用这个方法来产生时间延迟的。

实际上，一个使用到端口地址范围为 0～0x3ff 的 I/O 端口指令，几乎只要 1μs 的时间，所以如果你要如此做，例如，直接使用并行端口，只要加上几个 inb() 函数从该端口地址范围读入字节的信息即可。

4. 使用组合语言来延迟时间

如果你知道执进程序所在机器的处理器类型与时钟速度，你可以执行某些组合语言指令以获得较短的延迟时间 (但是记住，你在执行中的进程随时会被暂停， 所以，有时延迟的时间会比实际长)。如下面列表所示，内部处理器的速度决定了所要使用的时钟周期数；例如，一个 50 MHz 的处理器 (486DX-50或 486DX2-50)，一个时钟周期要花费 1/50 000 000 s (=20ns)。

指令	i386 时钟周期数	i486 时钟周期数
nop	3	1
xchg %ax，%ax	3	3
or %ax，%ax	2	1
mov %ax，%ax	2	1
add %ax，0	2	1

上面的列表中，指令 nop 与 xchg 应该不会有不良的后果。 指令最后可能会改变标志寄存器的内容，但是，这没关系，因为 gcc 会处理。 指令 nop 是个好的选择。

想要在你的程序中使用这些指令，你得使用 asm('instruction')。指令的语法就如同上面列表的用法; 如果你想要在单一的 asm() 叙述中使用多个指令，可以使用分号将它们隔开。 例如，asm('nop ; nop ; nop ; nop') 会执行4个 nop 指令，在 i486 或 Pentium 处理器中会延迟4个时钟周期 (i386 会延迟 12 个时钟周期)。

gcc 会将 asm() 翻译成单行组合语言程序码，所以不会有调用函数的负荷。在 Intel x86 结构中不可能有比1个时钟周期还短的时间延迟。

5. 在 Pentium 处理器上使用函数 rdtsc

对于 Pentium 处理器而言，你可以使用下面的 C 语言程序计算自从上次重新开机到现在经

过了多少个时钟周期:

```
extern __inline__ unsigned long long int rdtsc()
{
   unsigned long long int x;
   __asm__ volatile (".字节 0x0f, 0x31" : "=A" (x));
   return x;
}
```

你可以查询并参考此值以便延迟你想要的时钟周期数。

22.3.2 时间的量测

想要时间精确到 1s, 使用函数 time() 或许是最简单的方法。想要时间更精确, 函数 gettimeofday() 大约可以精确到微秒 (但是如前所述会受到 CPU 调度的影响)。至于Pentium 处理器, 使用上面的程序片断就可以精确到一个时钟周期。

如果要执行中的进程在一段时间到了之后能够被通知 (get a signal), 可以使用函数 setitimer() 或 alarm() 。

22.4 使用其他程序语言

上面的说明集中在 C 程序语言。它应该可以直接应用在 C++ 及 Objective C语言之上。至于组合语言部分, 虽然你必须先在 C 语言中调用函数 ioperm() 或 iopl(), 但是, 随后可以直接使用 I/O 端口读写指令。

至于其他程序语言, 除非你可以在该程序语言中插入单行组合语言或 C语言的程序码, 或者使用上面所说的系统调用, 否则, 倒不如编写一个内含有存取 I/O端口或延迟时间所必须使用的函数的C原始程序, 编译之后再与你的程序连接。要不然就使用前面所说的 /dev/port 字符设备文件。

22.5 一些有用的 I/O 端口

如果你要按照其原始的设计目的来使用这些或其他常用的 I/O 端口(例如, 控制一般的打印机或数据机), 你应该使用现成的设备驱动程序 (它通常含在内核中), 而不会如本文所说的去编写 I/O 端口程序。本节主要是提供给那些想要将液晶显示器 (LCD)、步进电动机或其他商业电子产品连接到PC标准 I/O 端口的人。

22.5.1 并行端口

并行端口的基本端口地址 (以下称为BASE)对于 /dev/lp0 是 0x3bc , 对于 /dev/lp1是 0x378 , 对于 /dev/lp2 是 0x278 。

除了下面即将描述的标准只输出模式, 大多数的并行端口都有扩充的双向模式。因为在用户模式中的程序无法使用 IRQ 或 DMA, 想要使用 ECP/EPP模式, 或许得编写一个内核的设备驱动程序。

端口地址 BASE+0 (信息端口) 用来控制信息端口的信号电平 (D0 到 D7 分别代表着 bit 0 到7, 电平状态: 0 = 低电平 (0 V), 1 = 高电平 (5 V))。一个写入信息到该端口的动作, 会将信息信号电平拴在端口的端脚上。一个将该端口的信息读出的动作会将上一次以标准只输出模式或扩充的写入模式所拴住的信息信号电平读回, 或者以扩充读出模式从另外一个设备将端脚上的信息信号电平读回。

端口地址 BASE+1 (状态端口) 是个只读入的端口，会将下面的输入信号电平读回：

bits 0 和 1 保留不用。

bit 2 IRQ 的状态。

bit 3 ERROR (1=高电平)。

bit 4 SLCT (1=高电平)。

bit 5 PE (1=高电平)。

bit 6 ACK (1=高电平)。

bit 7 -BUSY (0=高电平)。

端口地址 BASE+2 (控制端口) 是个只写入的端口(一个将该端口的信息读出的动作仅会将上一次写入的信息信号电平读回)，用来控制下面的状态信号：

bit 0 -STROBE (0=高电平)。

bit 1 AUTO_FD_XT (1=高电平)。

bit 2 -INIT (0=高电平)。

bit 3 SLCT_IN (1=高电平)。

bit 4 当被设定为 1 时允许并行端口产生 IRQ 信号 (发生在 ACK端脚的电平由低变高的瞬间)。

bit 5 用来控制扩充模式时端口的输出入方向 (0 = 写，1 = 读)，这是个只写的端口 (一个将该端口的信息读出的动作对此 bit 一点用处也没有)。

bits 6和7 保留不用。

端口的端脚排列方式 (该端口是一个 25 只脚 D 字形外壳的母连接器，i=输入，o=输出)如下：

1io —— STROBE，2io —— D0，3io —— D1，4io —— D2，5io —— D3，6io —— D4，7io —— D5，8io —— D6，9io D7，10i —— ACK，11i —— BUSY，12i —— PE，13i —— SLCT，14o —— AUTO_FD_XT，15i —— ERROR，16o INIT，17o —— SLCT_IN，18 —— 25 Ground

22.5.2 游戏端口

游戏端口的端口地址范围为 0x200-0x207。

端口的端脚排列方式 (该端口是一个 15 只脚 D 字形外壳的母连接器)如下：

1、8、9、15: +5 V (电源)。

4、5、12: 接地。

2、7、10、14: BA1、BA2、BB1和 BB2 等数位输入。

3、6、11、13: AX、AY、BX和 BY 等“类比”输入。

+5 V 的端脚似乎通常会直接连接到主机板的电源线上，所以它应该提供相当的电力，这还要看所使用主机板、电源以及游戏端口的类型。

数位输入用于操纵口的按钮可以让你连接两个操纵口的四个按钮 (操纵口A和操纵口B，各有两个按钮) 到游戏端口也就是数位输入的四个端脚。 它们应该是一般 TTL电压电平的输入，你可以直接从状态端口 (参考下面说明) 读出它们的电平状态。一个实际的操纵口在按钮被按下时会传回低电平 (0 V) 状态，否则传回高电平 (5V经由1kΩ的电阻连接到电源端脚) 状态。

所谓的类比输入实际是量测到的阻抗值。 游戏端口有四个单晶体多谐振荡器(一个 558 晶片) 连接到四个类比输入端脚。每个类比输入端脚与多谐振荡器的输出之间连接着一个 2.2k Ω 的电阻，而且多谐振荡器的输出与地之间连接着一个 0.01 μF 的时间电容。一个实际的操纵口的每个坐标 (X 和 Y) 上会有一个可变电阻，连接在 +5 V与每个相对的类比输入端脚之间 (端脚

AX 或 AY 是给操纵口A用的，而端脚 BX 或 BY是给操纵口B用的)。

操作的时候，多谐振荡器将其输出设定为高电平 (5 V)，并且等到时间电容上的电压达到 3.3 V 之后将相对的输出设定为低电平。 因此操纵口中多谐振荡器输出的高电平时间周期与可变电阻的电阻值成正比 (也就是，操纵口在相对坐标的位置)，如下所示:

R = (t - 24.2) / 0.011

其中，R是可变电阻的阻值 (Ω)，而t是高电平时间周期的长度 (s)。

因此，要读出类比输入端脚的数值，首先得启动多谐振荡器 (以端口写入的方式，请看下面)，然后查询四个坐标的信号状态(以持续的端口读出方式)，一直到信号状态由高电平变成低电平，计算其高电平时间周期的长度。这个持续查询的动作花费相当多的 CPU 时间，而且在一个非即时的多用户环境，所得的结果不是非常准确的。因为，你无法以固定的时间来查询信号的状态(除非你使用内核层次的驱动程序而且你得在查询的时候抑制掉中断的产生，但是这样做会浪费更多的 CPU 时间)。如果你知道信号的状态会花费一段不短的时间 (数十毫秒)而成为低电平，可以在查询之前调用函数 usleep() 将 CPU的时间让给其他想要执行的进程。

游戏端口中唯一需要你来存取的端口地址是 0x201 (其他的端口地址不是动作一样就是没用)。任何对这个端口地址所做的写入动作 (不论你写入什么) 都会启动多谐振荡器。对这个端口地址做读出动作会取回输入信号的状态:

bit 0: AX (1=高电平，多谐振荡器的输出状态)
bit 1: AY (1=高电平，多谐振荡器的输出状态)
bit 2: BX (1=高电平，多谐振荡器的输出状态)
bit 3: BY (1=高电平，多谐振荡器的输出状态)
bit 4: BA1 (数位输入，1=高电平)
bit 5: BA2 (数位输入，1=高电平)
bit 6: BB1 (数位输入，1=高电平)
bit 7: BB2 (数位输入，1=高电平)

22.5.3 串行端口

如果你的设备支持RS-232之类的接口，你应该可以使用串行端口。 Linux所提供的串行端口驱动程序应该能够应用在任何地方 (你应该不需要直接编写串行端口程序，或是内核的驱动程序)。它具有相当的通用性，所以如果使用非标准的速率以及其他等等，应该不是问题。

第五篇　Linux系统安全分析

第23章　系统管理员安全

本章从系统管理员的角度讨论安全问题。系统管理员是管理系统的人，其工作包括：启动系统、停止系统运行、安装新软件、增加新用户、删除老用户以及完成保持系统发展和运行的日常事务工作。

23.1　安全管理

安全管理主要分为四个方面：

• 防止未授权存取：这是计算机安全最重要的问题。用户意识、良好的口令管理（由系统管理员和用户双方配合）、登录活动记录和报告、用户和网络活动的周期检查，这些都是防止未授权存取的关键。

• 防止泄密：这也是计算机安全的一个重要问题。防止已授权或未授权的用户存取他人的重要信息。文件系统查帐、su登录和报告、用户意识、加密都是防止泄密的关键。

• 防止用户拒绝系统的管理：这一方面的安全应由操作系统来完成。一个系统不应被一个有意试图使用过多资源的用户损害。不幸的是，Linux不能很好地限制用户对资源的使用，一个用户能够使用文件系统的整个磁盘空间，而Linux基本不能阻止用户这样做。系统管理员最好用PS命令，记帐程序df和du周期地检查系统。查出过多占用CUP的进程和大量占用磁盘的文件。

• 防止丢失信息：这一安全方面与一个好系统管理员的实际工作（例如：周期地备份文件系统，系统崩溃后运行fsck检查，修复文件系统，当有新用户时，检测该用户是否可能使系统崩溃的软件）和保持一个可靠的操作系统有关(即用户不能经常性地使系统崩溃)。本书主要涉及前两个问题。

23.2　超级用户

一些系统管理命令只能由超级用户运行。超级用户拥有其他用户所没有的特权，超级用户不管文件存取许可方式如何，都可以读写任何文件，运行任何程序。系统管理员通常使用命令： /bin/su 或以 root 进入系统从而成为超级用户。在后面文章中以#表示由超级用户运行的命令，用$表示其他用户运行的命令。

23.3　文件系统安全

23.3.1　Linux文件系统概述

Linux文件系统是Linux系统的心脏部分，提供了层次结构的目录和文件。文件系统将磁盘空间划分为每1024个字节一组，称为块(也有用512字节 为一块的，如：SCO XENIX)。编号从0到整个磁盘的最大块数。

全部块可划分为四个部分，块 0称为引导块，文件系统不用该块 ;块1称为专用块，专用块含有许多信息，其中有磁盘大小和全部块的其他两部分的大小。从块 2开始是i节点表，i节点表中含有i节点，表的块数是可变的，后面将做讨论。 i节点表之后是空闲存储块（数据存储块），可用于存放文件内容。文件的逻辑结构和物理结构是十分不同的，逻辑结构是用户敲入 cat命令后所看到的文件，用户可得到表示文件内容的字符流。物理结构是文件实际上如何存放在磁盘上的存储格式。用户认为自己的文件是边疆的字符流，但实际上文件可能并不是以边疆的方式存放在磁盘上的，长于一块的文件通常将分散地存放在盘上。然而当用户存取文件时，Linux文件系统将以正确的顺序取出各块，给用户提供文件的逻辑结构。

当然，在Linux系统的某处一定会有一个表，告诉文件系统如何将物理结构转换为逻辑结构。这就涉及到i节点了。i节点是一个64字节长的表，含有有关一个文件的信息，其中有文件大小、文件所有者、文件存取许可方式，以及文件为普通文件、目录文件还是特别文件等。在i节点中最重要的一项是磁盘地址表。

该表中有13个块号。前10个块号是文件前10块的存放地址。这10个块号能给出一个至多10块长的文件的逻辑结构，文件将以块号在磁盘地址表中出现的顺序依次取得相应的块。

当文件长于10块时又怎样呢?磁盘地址表中的第11项给出一个块号，这个块号指出的块中含有256个块号，至此，这种方法满足了至多长于 266块的文件（272 384字节）。如果文件大于266块，磁盘地址表的第12项给出一个块号，这个块号指出的块中含有 256个块号，这256个块号的每一个块号又指出一块，块中含 256个块号，这些块号才用于取文件的内容。磁盘地址中和第13项索引寻址方式与第12项类似，只是多一级间接索引。

这样，在Linux系统中，文件的最大长度是16 842 762块，即17 246 988 288字节，有幸是Linux系统对文件的最大长度（一般为 1到2M字节）加了更实际的限制，使用户不会无意中建立一个用完整个磁盘区所有块的文件。

文件系统将文件名转换为 i节点的方法实际上相当简单。一个目录实际上是一个含有目录表的文件：对于目录中的每个文件，在目录表中有一个入口项，入口项中含有文件名和与文件相应的i节点号。当用户敲入cat xxx时，文件系统就在当前目录表中查找名为 xxx的入口项，得到与文件xxx相应的i节点号，然后开始取含有文件xxx的内容的块。

23.3.2 设备文件

Linux系统与本系统上的各种设备之间的通信，通过特别文件来实现， 就程序而言，磁盘是文件，调制解调器是文件，甚至内存也是文件。所有连接到系统上的设备都在 /dev目录中有一个文件与其对应。当在这些文件上执行I/O操作时，由Linux系统将I/O操作转换成实际设备的动作。例如，文件 /dev/mem是系统的内存，如果使用 cat命令显示这个文件，实际上是在终端显示系统的内存。为了安全起见，这个文件对普通用户是不可读的。因为在任一给定时间，内存区可能含有用户登录口令或运行程序的口令，某部分文件的编辑缓冲区，缓冲区可能含有用ed -x命令解密后的文本，以及用户不愿让其他人存取的种种信息。

在/dev中的文件通常称为设备文件，用ls /dev命令可以看看系统中的一些设备：

acuo	呼叫自动拨号器。
console	系统控制台。
dsknn	块方式操作磁盘分区。
kmem	核心内存。
mem	内存。

lp	打印机。
mto	块方式操作磁带。
rdsknn	流方式操作的磁盘分区。
rmto	流方式操作的磁带。
swap	交换区。
syscon	系统终端。
ttynn	终端口。
x25	网络端口。
等等	

23.3.3 /etc/mknod命令

用于建立设备文件。只有系统管理员能使用这个命令建立设备文件。其参数是文件名，字母c或b分别代表字符特别文件或块特别文件、主设备号、次设备号。块特别文件是像磁带、磁盘这样一些以块为单位存取数据的设备。字符特别文件是如像终端、打印机、调制解调器或者其他任何与系统通信时，一次传输一个字符的设备，包括模仿对磁盘进行字符方式存取的磁盘驱动器。主设备号指定了系统子程序（设备驱动程序），当在设备上执行I/O时，系统将调用这个驱动程序。调用设备驱动程序时，次设备号将传递给该驱动程序（次设备规定具体的磁盘驱动器，带驱动器，信号线编号，或磁盘分区）。每种类型的设备一般都有自己的设备驱动程序。

文件系统将主设备号和次设备号存放在 i节点中的磁盘地址表内，所以没有磁盘空间分配给设备文件（除i节点本身占用的磁盘区外）。当程序试图在设备文件上执行 I/O操作时，系统识别出该文件是一个特别文件，并调用由主设备号指定的设备驱动程序，次设备号作为调用设备驱动程序的参数。

23.3.4 安全考虑

将设备处理成文件，使得 Linux程序独立于设备，即程序不必一定要了解正使用的设备的任何特性，存取设备也不需要记录长度、块大小、传输速度、网络协议等这样一些信息，所有烦人的细节由设备驱动程序去关心考虑，要存取设备，程序只需打开设备文件，然后作为普通的Linux文件来使用。

从安全的观点来看这样处理很好，因为任何设备上进行的I/O操作只经过了少量的渠道（即设备文件），用户不能直接地存取设备。所以如果正确地设置了磁盘分区的存取许可，用户就只能通过Linux文件系统存取磁盘。文件系统有内部安全机制（文件许可）。不幸的是，如果磁盘分区设备得不正确，任何用户都能够写一个程序，读磁盘分区中的每个文件，作法很简单：读一个i节点，然后以磁盘地址表中块号出现的顺序，依次读这些块号指出的存有文件内容的块。故除了系统管理员以外，决不要使盘分区对任何人可写。因为所有者，文件存取许可方式这样一些信息存放于i节点中，任何人只要具有已安装分区的写许可，就能设置任何文件的 SUID许可，而不管文件的所有者是谁，也不必用chmod（）命令，还可绕过系统建立的安全检查。

以上所述对内存文件 mem、kmem和对换文件swap也是一样的。这些文件含有用户信息，一个耐心的程序可以将用户信息提取出来。

要避免磁盘分区（以及其他设备）可读可写，应当在建立设备文件前先用 umask命令设置文件建立屏蔽值。

一般情况下，Linux系统上的终端口对任何人都是可写的，从而使用户可以用 write命令发

送信息。虽然write命令易引起安全方面的问题，但大多数用户觉得用 write得到其他用户的信息很方便，所以系统将终端设备的存取许可设置成对所有用户可写。

/dev目录应当是755存取许可方式，且属系统管理员所有。

不允许除系统管理员外的任何用户读或写盘分区的原则有一例外，即一些程序（通常是数据库系统）要求对磁盘分区直接存取，解决这个问题的经验的盘分区应当由这种程序专用（不安装文件系统），而且应当告知使用这种程序的用户，文件安全保护将由程序自己而不是 Linux 文件系统完成。

23.3.5 find命令

find命令用于搜索目录树，并对目录树上的所有文件执行某种操作，参数是目录名表（指出从哪些起点开始搜索），还可给出一个或多个选项，规定对每个文件执行什么操作。

find / -print 将列出当前工作目录下的目录树的每一个文件。

find / -user bob -print 将列出在系统中可找到的属于bob用户的所有文件。

find /usr/bob -perm 666 -print 将列出/usr/bob目录树下所有存取许可为666的文件。若将666改为-666则将列出所有具有包含了666在内的存取许可方式的文件（如777）。

find /usr/bob -type b -print 将列出/usr/bob目录树下所有块特别文件（c为字符特别文件）。

find / -user root -perm -4000 -exec ls -l {} \; 是一个较复杂的命令，-exec COMMAND \;允许对所找到的每个文件运行指定的命令 COMMAND。若COMMAND中含有{}，则{}将由find所找到的文件名替换。COMMAND必须以\;结束。

以上举例介绍了find的用法，各选项可组合使用以达到更强的功能。

23.3.6 secure程序

系统管理员应当做一个程序以定期检查系统中的各个系统文件，包括检查设备文件和SUID和SGID程序，尤其要注意检查SUID和SGID程序，检查/etc/passwd和/etc/group文件，寻找久未登录的帐户和校验各重要文件是否被修改。（源程序清单将在今后发表。）

23.3.7 ncheck命令

用于检查文件系统，只用一个磁盘分区名作为参数，将列出 i节点号及相应的文件名。i节点相同的文件为建链文件。

注意 所列出的清单文件名与 mount命令的第一个域相同的文件名前部分将不会列出来。因为是做文件系统内部的检查，ncheck并不知道文件系统安装点以上部分的目录。也可用此命令来搜索文件系统中所有的SUID和SGID程序和设备文件，使用 -s选项来完成此项功能。

23.3.8 安装和拆卸文件系统

Linux文件系统是可安装的，这意味着每个文件系统可以连接到整个目录树的任意节点上（根目录总是被安装上的）。安装文件系统的目录称为安装点。

/etc/mount命令用于安装文件系统，用这条命令可将文件系统安装在现有目录结构的任意处。

安装文件系统时，安装点的文件和目录都是不可存取的，因此未安装文件系统时，不要将文件存入安装点目录。文件系统安装后，安装点的存取许可方式和所有者将改变为所安装的文

件根目录的许可方式和所有者。

安装文件系统时要小心：安装点的属性会改变！还要注意新建的文件，除非新文件系统是由标准文件建立的，系统标准文件会设置适当的存取许可方式，否则新文件系统的存取许可将是777!

可用-r选项将文件系统安装成只读文件系统。需要写保护的带驱动器和磁盘，应当以这种方式来安装。

不带任何参数的/etc/mount可获得系统中所安装的文件系统的有关信息。包括：文件系统被安装的安装点目录，对应/dev中的设备，只读或可读写，安装时间和日期等。从安全的观点来讲，可安装系统的危险来自用户可能请求系统管理员为其安装用户自己的文件系统。如果安装了用户的文件系统，则应在允许用户存取文件系统前，先扫描用户的文件系统，搜索SUID/SGID程序和设备文件。在除了系统管理员外任何人不能执行的目录中安装文件系统，用find命令或secure列出可疑文件，删除不属用户所有的文件的SUID/SGID许可。

用户的文件系统用完后，可用umount命令卸下文件系统。并将安装点目录的所有者改回系统管理员，存取许可改为755。

23.3.9 系统目录和文件

Linux系统中有许多文件和目录不允许用户写，如：/bin、/usr/bin、/usr/lbin、/etc/passwd、/usr/lib/crontab、/Linux、/etc/rc、/etc/inittab，可写的目录允许移动文件，这样会引起安全问题。

系统管理员应经常检查系统文件和目录的许可权限和所有者。可做一个程序根据系统提供的规则文件（在/etc/permlist文件中）所描述的文件所有者和许可权规则检查各文件。

注意 如果系统的安全管理不好，或系统是新安装的，其安全程序不够高， 可以用make方式在安全强的系统上运行上述程序，将许可规则文件拷贝到新系统来，再以设置方式在新系统上运行上述程序，就可提高本系统的安全程序。但要记住，两个系统必须运行相同的Linux系统版本。

23.4 作为root运行的程序

在Linux系统中，有些程序由系统作为 root进程运行。这些程序并不一定具有 SUID许可，因为其不少程序仅由root运行，系统管理员需要清楚这些程序做什么，以及这些程序还将运行其他什么程序。

23.4.1 启动系统

当某些Linux系统（如SCO Linux/XENIX）启动时，是以被称为单用户的方式运行的，在这种方式中普通用户不能登录，唯有的进程是init、swapper以及一些由系统管理员从控制台运行的进程。Linux系统的单用户方式启动，使系统管理员能在允许普通用户登录以前，先检查系统操作，确保系统一切正常，当系统处于单用户方式时，控制台作为超级用户，命令揭示是“#”，有些Linux系统不要确认超级用户口令就认可控制台是root，给出#提示符。这就可能成为一个安全问题。

23.4.2 init 进程

Linux系统总是以某种方式或称为某种级运行，系统有若干种运行级，这些运行级由 init进程控制。

Linux系统启动时以单用户方式运行，也叫1级或S级。对于其他用户登录进入系统，Linux有一种多用户运行方式，也叫2级。init进程控制系统运行级，它读入文件 /etc/inittab，该文件详细地规定了哪些进程在哪一级运行。当系统管理员敲入 init n（数字），系统就进入n级。init读该文件以确定终止哪些进程，启动哪些进程。

有效的运行级的数值是从0到6与s。

注意 由init建立的进程以UID为0运行（root），从/etc/inittab运行的程序也作为root运行，所以系统管理员要确保自己知道/etc/inittab中的程序做什么工作，确保这些程序以及这些程序所在的目录（直到/和/etc/inittab）除root外无人可写。

23.4.3 进入多用户

当Linux系统进入多用户方式时，将初始化一系列事件，接着开始执行 gettys，允许其他用户登录进入系统。如果再看看 /etc/inittab文件，会看到gettys定义在运行级2，至少三个外壳程序/etc/brc、/etc/bcheckrc、/etc/rc*也定义在运行级2。这些程序都在gettys启动前运行。

这些外壳程序作为root运行，也不能仅对root可写还应当检查外壳程序运行的命令，因为这些命令也将作为root运行。

23.4.4 shutdown命令

用shutdown命令关系统，shutdown 外壳程序发出警告，通知所有用户离开系统，在“给定的期限时间”到了后，就终止进程，拆卸文件系统，进入单用户方式或关机状态。一旦进入单用户方式，所有的 gettys停止运行，用户不能再登录。进入关机状态后可将系统关电。shutdown仅能由作为root登录的用户从系统控制台上运行。所以任何的 shutdown运行的命令仅能对root可写。

23.4.5 系统V的cron程序

cron在Linux系统是在多用户方式时运行的，根据规定的时间安排执行指定的命令，每隔一分钟检查一次文件 /usr/lib/crontab，寻找是否有应当运行的程序，如果找到要运行的程序，就运行该程序，否则睡眠等待一分钟。

实际的/usr/lib/crontab用于根据全天的规则时间表运行程序，也可在夜晚运行白天不愿运行怕降低其他用户速度的程序。通常由 cron运行的程序是如记帐、存文件这样的程序。 cron一般在系统进入多用户方式后由/etc/rc启动，当 shutdown运行killall命令时便终止运行。由cron运行的程序作为root，所以应当注意放什么程序在crontab中，还要确保/usr/lib/crontab和该表中列出的任何程序对任何人不可写。

如果用户需要由cron执行一个程序，系统管理员可用 su命令在crontab表中建立一个入口，使用户的程序不能获得root的权限。

23.4.6 系统V版本2之后的cron程序

在系统V版本2中，cron被修改成允许用户建立自己的crontab入口， /usr/lib/crontab文件不再存在，由目录/usr/spool/cron/crontabs中的文件代替。这些文件的格式与crontab相同，但每个文件与系统中的一个用户对应，并以某用户的名义由 cron运行。

如果想限制能建立crontab的用户，可在文件/usr/lib/cron/cron.allow文件中列出允许运行crontab命令的用户。任何未列于该文件的用户不能运行 crontab。反之，若更希望列出不允许运行crontab命令的用户，则可将他们列入/usr/lib/cron/cron.deny文件中，未列于该文件的其他用户将被允许建立crontab。

注意 若两个文件都存在，系统将使用cron.allow，忽略cron.deny。如果两个文件都不存在，则只有root可运行crontab。所以，若要允许系统中的所有用户都可运行crontab命令，应当建立一个空的cron.deny文件，如果cron.allow也存在，则删除该文件。

这个版本的cron命令的安全程度比前一个高，因为用户只能看自己的 crontab，系统管理员也不必担心其他用户的程序是否会作为root运行，由于允许每个系统登录用户有自己的crontab，也简化了对程序必须由cron运行，但不必作为root运行的系统程序的处理。

必须确保root的crontab文件仅对root可写，并且该文件所在的目录及所有的父目录也仅对root可写。

23.4.7 /etc/profile

每当用户（包括root在内）登录时，由外壳执行/etc/profile文件，应确保这个文件以及从这个文件运行的程序和命令都仅对root可写。

23.5 /etc/passwd文件

/etc/passwd文件是Linux安全的关键文件之一。该文件用于用户登录时校验用户的口令，当然应当仅对root可写。文件中每行的一般格式为：

LOGNAME：PASSWORD：UID：GID：USERINFO：HOME：SHELL

每行的头两项是登录名和加密后的口令，后面的两个数是 UID和GID，接着的一项是系统管理员想写入的有关该用户的信息，最后两项是两个路径名：一个是分配给用户的HOME目录，另一个是用户登录后将执行的外壳（若为空格则缺省为/bin/sh）。

23.5.1 口令时效

/etc/passwd文件的格式使系统管理员能要求用户定期地改变他们的口令。在口令文件中可以看到，有些加密后的口令有逗号，逗号后有几个字符和一个冒号。如： steve：xyDfcc Trt180x，My 8：0：0：admin：/：/bin/sh

restrict：pomJk109Jky41，1：0：0：admin：/：/bin/sh

pat：xmotTVoyumjls：0：0：admin：/：/bin/sh

可以看到，steve的口令逗号后有4个字符，restrict有2个，pat没有逗号。逗号后第一个字符是口令有效期的最大周数，第二个字符决定了用户再次修改口令之前，原口令应使用的最小周数（这就防止了用户改了新口令后立刻又改回成老口令）。其余字符表明口令最新修改时间。

要能读懂口令中逗号后的信息，必须首先知道如何用 passwd_esc计数，计数的方法是：.=0 /=1 0-9=2-11 A-Z=12-37 a-z=38-63

系统管理员必须将前两个字符放进/etc/passwd文件，以要求用户定期地修改口令，另外两个字符当用户修改口令时，由passwd命令填入。

注意 若想让用户修改口令，可在最后一次口令被修改时，放两个“.”，则下一次用户登录时将被要求修改自己的口令。

有两种特殊情况：

• 最大周数（第一个字符）小于最小周数（第二个字符），则不允许用户修改口令，仅超级用户可以修改用户的口令。

• 第一个字符和第二个字符都是“ . ”，这时用户下次登录时被要求修改口令，修改口令后，passwd命令将“ . ”删除，此后再不会要求用户修改口令。

23.5.2 UID和GID

/etc/passwd中UID信息很重要，系统使用 UID而不是登录名区别用户。一般来说，用户的UID应当是独一无二的，其他用户不应当有相同的 UID数值。根据惯例，从0到99的UID保留用作系统用户的UID（root、bin、uucp等）。

如果在/etc/passwd文件中有两个不同的入口项有相同的 UID，则这两个用户对相互的文件具有相同的存取权限。

23.6 /etc/group文件

/etc/group文件含有关于小组的信息， /etc/passwd中的每个GID在本文件中应当有相应的入口项，入口项中列出了小组名和小组中的用户。这样可方便地了解每个小组的用户，否则必须根据GID在/etc/passwd文件中从头至尾地寻找同组用户。

/etc/group文件对小组的许可权限的控制并不是必要的，因为系统用 UID和GID（取自/etc/passwd）决定文件存取权限，即使/etc/group文件不存在于系统中，具有相同的 GID用户也可以小组的存取许可权限共享文件。

小组就像登录用户一样可以有口令。如果/etc/group文件入口项的第二个域为非空，则将被认为是加密口令，newgrp命令将要求用户给出口令，然后将口令加密，再与该域的加密口令比较。

给小组建立口令一般不是个好方法。第一，如果小组内共享文件，而某人猜中小组口令，则该组的所有用户的文件就可能泄密：其次，管理小组口令很费事，因为对于小组没有类似的passwd命令。可用/usr/lib/makekey生成一个口令写入/etc/group。

以下情况必须建立新组：

• 可能要增加新用户，该用户不属于任何一个现有的小组。

• 有的用户可能时常需要独自为一个小组。

• 有的用户可能有一个SGID程序，需要独自为一个小组。

• 有时可能要安装运行SGID的软件系统，该软件系统需要建立一个新组。

要增加一个新组，必须编辑该文件，为新组加一个入口项。

由于用户登录时，系统从 /etc/passwd文件中取GID，而不是从/etc/group中取GID，所以group文件和口令文件应当具有一致性。对于一个用户的小组， UID和GID应当是相同的。多用户小组的GID应当不同于任何用户的UID，一般为5位数，这样在查看/etc/passwd文件时，就可根据5位数据的GID识别多用户小组，这将减少增加新组和新用户时可能产生的混淆。

23.7 增加、删除和移走用户

23.7.1 增加用户

增加用户有三个过程：

• 在/etc/passwd文件中写入新用户的入口项。
• 为新登录用户建立一个HOME目录。
• 在/etc/group中为新用户增加一个入口项。

在/etc/passwd文件中写入新的入口项时，口令部分可先设置为 NOLOGIN，以免有人作为此新用户登录。在修改文件前，应使用命令 mkdir /etc/ptmp，以免他人同时修改此文件。新用户一般独立为一个新组，GID号与UID号相同（除非他要加入目前已存在的一个新组），UID号必须和其他人不同，HOME目录一般设置在/usr 或/home目录下，建立一个以用户登录名为名称的目录做为其主目录。

23.7.2 删除用户

删除用户与增加用户的工作正好相反，首先在 /etc/passwd和/etc/group文件中删除用户的入口项，然后删除用户的HOME目录和所有文件。

rm -r /usr/loginname 删除整个目录树。

如果用户在/usr/spool/cron/crontabs中有crontab文件，也应当删除。

23.7.3 将用户移到另一个系统

这是一个复杂的问题，不只是拷贝用户的文件和用户在 /etc/passwd文件中的入口项。首先一个问题是用户的UID和GID可能已经用于另一个系统，若是出现这种情况，必须给要移动的用户分配另外的 UID和GID，如果改变了用户的 UID和GID，则必须搜索该用户的全部文件，将文件的原UID和GID改成新的UID和GID。

用find命令可以完成这一修改：

```
find.-user olduid -exec chown newuid {} \;
find. -group oldgid -exec chgrp newgid {} \;
```

也许还要为用户移走其他一些文件：/usr/mail/user和/usr/spool/cron/crontabs/user。

如果用户从一个不是本系统管理员的系统移来，则应对该用户的目录结构运行程序来检查。一个不安全系统的用户，可能有与该用户其他文件存在一起的 SUID/SGID程序，而这个SUID/SGID程序属于另一个用户。在这种情况下，如果用cpio或tar命令将用户的目录结构拷贝到本系统，SUID/SGID程序也将会拷贝到本系统而没有任何警告信息。应当在允许用户使用新系统以前先删除这种文件的SUID/SGID许可。总之，始终坚持检查所移用户的文件会更安全些。也可以用su命令进入用户的帐户，再拷贝用户文件，这样文件的所有者就是该用户，而不是root。

23.8 安全检查

像find和secure这样的程序称为检查程序，它们搜索文件系统，寻找出 SUID/SGID文件、设备文件、任何人可写的系统文件、设有口令的登录用户、具有相同 UID/GID的用户等等。

23.8.1 记帐

Linux记帐软件包可用作安全检查工具，除最后登录时间的记录外，记帐系统还能保存全天运行的所有进程的完整记录，对于一个进程所存贮的信息包括UID、命令名、进程开始执行与结束的时间、CPU时间和实际消耗的时间、该进程是否是root进程，这将有助于系统管理员了解系统中的用户在干什么。acctcom命令可以列出一天的帐目表。标明系统中有多个记帐数据文件，记帐信息保存在文件/usr/adm/pacct*中，/usr/adm/pacct是当前记录文件，/usr/adm/pacctn是以前的记帐文件（n为整型数）。若有若干个记帐文件要查看，可在acctcom命令中指定文件名：

acctcom /usr/adm/pacct? /usr/adm/pacct要检查的一个问题是：在acctcom的输出中查找一个用户过多的登录过程，若有，则说明可能有人一遍遍地尝试登录，猜测口令，企图非法进入系统。此外，还应查看root进程，除了系统管理员用su命令从终端进入root，系统启动，系统停止时间，以及由init（通常init只启动getty，login，登录外壳），cron启动的进程和具有root SUID许可的命令外，不应当有任何root进程。由记帐系统也可获得有关每个用户的CPU利用率，运行的进程数等统计数据。

23.8.2 其他检查命令

1) du 报告在层次目录结构（当前工作目录或指定目录起）中各目录占用的磁盘块数。可用于检查用户对文件系统的使用情况。

2) df 报告整个文件系统当前的空间使用情况。可用于合理调整磁盘空间的使用和管理。

3) ps 检查当前系统中正在运行的所有进程。对于用了大量CPU时间的进程、同时运行了许多进程的用户、运行了很长时间但用了很少CPU时间的用户进程应当深入检查。还可以查出运行了一个无限循环的后台进程的用户，未注销帐户就关闭终端的用户（一般发生在直接连线的终端）。

4) who 可以告诉系统管理员系统中工作的进展情况等等许多信息，检查用户的登录时间，登录终端。

5) su 每当用户试图使用su命令进入系统用户时，命令将在/usr/adm/sulog文件中写一条信息，若该文件记录了大量试图用su进入root的无效操作信息，则表明可能有人企图破译root口令。

6) login 在一些系统中，login程序记录了无效的登录企图（若本系统的login程序不做这项工作而系统中有login源程序，则应修改login）。

每天总有少量的无效登录，若无效登录的次数突然增加了两倍，则表明可能有人企图通过猜测登录名和口令，非法进入系统。

这里最重要的一点是：系统管理员越熟悉自己的用户和用户的工作习惯，就越能快速发现系统中任何不寻常的事件，而不寻常的事件意味着系统已被人窃密。

23.8.3 安全检查程序的问题

以上的检查方法没有几个能防止诱骗。如find命令，如果碰到路径名长于256个字符的文件或含有多于200个文件的目录，将放弃处理该文件或目录，用户就有可能利用建立多层目录结构或大目录隐藏SUID程序，使其逃避检查（但find命令会给出一个错误信息，系统管理员应手工检查这些目录和文件）。也可用ncheck命令搜索文件系统，但它没有find命令指定搜索哪种文件的功能。

如果定期存取.profile文件，则检查久未登录用户的方法就不奏效了。而用户用 su命令时，除非用参数，否则su不读用户的.profile文件。

有三种方法可寻找久未登录的帐户：

- Linux记帐系统在文件/usr/adm/acct/sum/login中为每个用户保留了最后一次登录日期。用这个文件的好处是，该文件由系统维护，所以可完全肯定登录日期是准确的。缺点是必须在系统上运行记帐程序以更新 loginlog文件，如果在清晨（或午夜后）运行记帐程序，一天的登录日期可能就被清除了。
- /etc/passwd文件中的口令时效域将能告诉系统管理员，用户的口令是否过期了，若过期，则意味着自过期以来，帐户再未被用过。这一方法的好处在于系统记录了久未用的帐户，检查过程简单，且不需要记帐系统所需要的磁盘资源，缺点是也许系统管理员不想在系

统上设置口令时效，而且这一方法仅在口令的最大有效期（只有几周）才是准确的。

- 系统管理员可以写一个程序，每天（和重新引导系统时）扫描 /etc/wtmp，自己保留下用户最后登录时间记录，这一方法的好处是不需要记帐程序，并且时间准确，缺点是要自己写程序。

以上任何方法都可和/usr/adm/sulog文件结合起来，查出由login或su登录帐户的最后登录时间。如果有人存心破坏系统安全，第一件要做的事就是寻找检查程序。破坏者将修改检查程序，使其不能报告任何异常事件，也可能停止系统记帐，删除记帐文件，使系统管理员不能发现破坏者干了些什么。

23.8.4 系统泄密后怎么办

如果发现有人已经破坏了系统安全，这时系统管理员首先应做的是面对肇事用户。如果该用户所做的事不是蓄意的，而且公司没有关于“破坏安全”的规章，也未造成损失，则系统管理员只需清理系统，并留心该用户一段时间。如果该用户造成了某些损失，则应当报告有关人员，并且应尽可能地将系统恢复到原来的状态。

如果肇事者是非授权用户，那就得做最坏的假设了：肇事者已设法成为 root且本系统的文件和程序已经泄密了。系统管理员应当想法查出谁是肇事者，他造成了什么损坏，还应当对整个文件做一次全面的检查，并不只是检查 SUID和SGID，设备文件。如果系统安全被一个敌对的用户破坏了，应当采用下面的步骤：

- 关闭系统，然后重新引导，不要进入多用户方式，进入单用户方式。
- 安装含有本系统原始Linux版本的带和软盘。
- 将/bin、/usr/bin、/etc、/usr/lib中的文件拷贝到一个暂存目录中。
- 将暂存目录中所有文件的校验和（用原始版本的 sum程序拷贝做校验和，不要用 /bin中的sum程序做）与系统中所有对旧的文件的校验和进行比较，如果有任何差别，要查清差别产生的原因。如果两个校验和不同，是由于安装了新版本的程序，确认是否的确安装了新版本程序。如果不能找出校验和不同的原因，用暂存目录中的命令替换系统中的原有命令。
- 在确认系统中的命令还未被篡改之前，不要用系统中原命令。用暂存目录中的外壳，并将PATH设置为仅在暂存目录中搜索命令。
- 根据暂存目录中所有系统命令的存取许可，检查系统中所有命令的存取许可。
- 检查所有系统目录的存取许可，如果用了 perms，检查permlist文件是否被篡改过。
- 如果系统Linux（/Linux）的校验和不同于原版的校验和，并且系统管理员从未修改过核心，则应当认为，一个非法者“很能干”，从暂存缓冲区重新装入系统。系统管理员可以从逐步增加的文件系统备份中恢复用户的文件，但是在检查备份中的“有趣”文件之前，不能做文件恢复。
- 改变系统中的所有口令，通知用户他们的口令已改变，应找系统管理员得到新口令。
- 当用户来要新口令时，告诉用户发生了一次安全事故，让他们查看自己的文件和目录是否潜伏着危害（如SUID文件、特洛依木马、任何人可写的目录），并报告系统管理员任何异乎寻常的情况。
- 设法查清安全破坏是如何发生的。如果没有肇事者说明，这也许是不可能弄清的。如果能发现肇事者如何进入系统，设法堵住这个安全漏洞。

第一次安装Linux系统时，可以将外壳、sum命令、所有文件的校验和存放在安全的介质上（磁带、软盘、硬盘和任何可以卸下并锁起来的介质）。这样不必再从原版系统带上重新装入文件，

可以安装备份介质，装入外壳和sum，将存在磁带上的校验和与系统中文件的校验和进行比较。系统管理员也许想自己写一个计算校验和的程序，破坏者将不能知道该程序的算法，如果将该程序及校验和保存在磁带上，这一方法的保密问题就减小到一个物理安全问题，即只需将磁带锁起来。

23.9 加限制的环境

23.9.1 加限制的外壳

这种外壳几乎与普通的外壳相同，但是该外壳能限制一个用户的能力，不允许用户有某些标准外壳所允许的行为：

- 不能改变工作目录（cd）。
- 不能改变PATH或SHELL 外壳变量。
- 不能使用含有“ / ”的命令名。
- 不能重定向输出（>和>>）。
- 不能用exec执行程序。

用户在登录时，在配置文件后系统就强加上了这些限制，如果用户在 .profile文件正被解释时按了Break键或Delete键，该用户将被注销。

这些简单的限制，使用写受限制用户的 .profile文件的系统管理员可以对用户能使用什么命令，进行完全的控制。

应当注意：系统V加限制的外壳实际上不很安全，在有敌对的用户时不要用。系统 V版本2以后的版本中加限制的外壳更安全些。但若允许受限制的用户使用某些命令（如 env、cp、ln），用户将能绕过加限制的外壳，进入非限制的外壳。

23.9.2 用chroot（）限制用户

如果的确想限制一个用户，可用 chroot（）子程序为用户建立一个完全隔离的环境。这个子程序改变了进程对根目录的概念，因此可用于将一个用户封在整个文件系统的某一层目录结构中，使用户无法用cd命令转出该层目录结构，不能存取文件系统中其他部分的任何文件。这种限制方式比加限制的外壳好得多。用户使用的命令应由系统管理员在新的 root目录中建立一个bin目录，并建立用户可用命令的链到系统的 /bin目录中相应命令文件上（若在不同的文件系统则应拷贝命令文件）。

还应建立新的passwd文件，保留系统登录帐户（为了使 ls -l正确地报告与受限制的子文件系统中的文件相关的正确登录名）和用户帐户，但系统帐户的口令改为 NOLOGIN以使受限制的用户不能取得系统登录的真实口令，使“ 破密 ”程序的任何企图成为泡影。 utmp文件是who所需要的，该文件含有系统中已登录用户的列表。

新的/etc/profile文件也不是建链文件，以便受限制的用户可以执行不同的启动命令。

/dev目录中的终端设备文件被链接到新的 /dev目录下，因为命令 who产生输出时要查看这些文件。在系统V及以后的Linux版本中，login命令有chroot（）的功能。如果口令文件中用户入口项的登录外壳域（最后一个域）是 *，login将调用chroot（）把用户的根目录设置成为口令文件中用户入口项登录目录域指定的目录。然后再调用 exec（）执行login，新的login将在新子系统文件中执行该用户的登录。

chroot（）并不是把root封锁在一个子文件系统中，所以给受限制用户用的命令时应加以考虑，具有root的SUID许可的程序可能会给予用户 root的能力。应当将这种可能减低到最小程

度，交给用户使用的命令应当取自清除了SUID陷井的系统命令。链接文件可减少磁盘占用区，但要记住，当与敌对用户打交道时，链接到chroot目录结构（尤其是命令）的系统文件是很危险的。如果建立一个像这样的限制环境，应确保对安装到新的 /bin的每条命令都做过测试，有些程序可能有系统管理员未曾想到的出乎意料的执行结果。为了使这些命令能运行，还得在加限制的子文件系统中加服务目录或文件，如： /tmp、/etc/termcap、/dev/swap、/dev/mem、/usr/lib/terminfo、/dev/kmem，用户所登录的/dev中的tty文件以及/Linux。

有些程序在子文件系统中运行时不会很好，如果将假脱机程序和网络命令拷贝到加限制的子文件系统中，并放在为两条命令专建的目录层结构下，它们可能也运行不了。

23.10 小系统安全

任何足够小的运行于办公室的Linux系统就是小系统，这类小系统也包括所有台式 Linux机器。根据安全观点，小系统的特别之处有以下几点：

- 小系统的用户比大系统的用户少，通常是很小一组用户，使系统管理员能熟悉每个人，安全问题可以直接地面对面处理。
- 由于小Linux系统管理更简单，可能只需要一个系统管理员，因而维护系统安全的责任只有一个人担负。
- 如果既是用户又是系统管理员，将不能花大量时间考虑系统安全。
- 如果自己拥有系统并且是系统管理员，就可能有权直接将违反规定的用户从系统中删除，而一般大系统的管理员没有这种权利。
- 如果自己是系统的唯一用户，则既是用户又是管理员，维护系统安全的任务就很简单了，只须确保系统中所有登录帐户的口令是好的即可。
- 如果不能将系统锁起来，就把重要的数据存放在软盘上，把软盘锁起来。
- 即使系统中有若干个用户，但如果系统的终端之间是有线连接，并且用户们保持门上锁，则系统也将是安全的，至少在本组用户内是安全的。
- 小系统通常有可移动的介质（软盘），可用mount命令将其安装到系统上，提供一种安全的方法让用户自己在系统上安装软盘，否则系统管理员要一天到晚地干这些琐碎的安装盘事务。允许用户安装软盘的通常做法是给用户一个 SUID程序，该程序与系统管理员安装用户软盘基本完成同样的操作，首先检查软盘上有无 SUID/SGID/设备文件，若发现任何奇怪的文件，则拒绝安装该软盘。
- 当小系统开电源后，系统一般在从硬盘引导以前，先试图从软盘引导。这就意味着计算机将首先试图从软盘装入程序，若软盘不在驱动器中，系统将从硬盘装入 Linux内核。软盘几乎可以含有任何程序，包括在控制台启动 root 外壳的Linux系统版本。如果破坏者有一把螺丝起子和有关系统内部的一些知识，则即便系统有被认为防止安全事故发生的特殊“ 微码 ”口令，也可能被诱骗去从软盘引导。
- 即使小系统晚上不锁，凡从不将个人的或秘密的信息存放在大系统上的人（他们不可能认识所有系统上的用户），也不会想把这样的信息存放在小系统上。
- 小系统的系统管理员在使用Linux系统方面常不如大系统管理员有经验，而安全地管理系统需要一定的使用系统的知识。

23.11 物理安全

对于运行任何操作系统的小型或大型计算机，物理安全都是一个要考虑的重要问题，物理

安全包括：锁上放置计算机的屋子、报警系统、警卫、所有安置在不能上锁的地方的通信设施（包括有线通信线、电话线、局域网、远程网、应答调制解调器）、钥匙或信用卡识别设备、给用户的口令和钥匙分配、任何前置通信设施的加密装置、文件保护、备份或恢复方案（称为安全保险方案，用作应付偶然的或蓄意的数据或计算设备被破坏的情况）、上锁的输出端、上锁的废物箱和碎纸机。物理安全中的总考虑应是：在安全方案上所付出的代价不应当多于值得保护的（硬件或软件的）价值。

下面着重讨论保护用户的各种通信线。对于任何可在不上锁的地方存取的系统，通信是特别严重的安全薄弱环节。当允许用户通过挂到地方电话公司的拨号调制解调器存取系统时，系统的安全程度就将大大地削弱，有电话和调制解调器的任何人就有可能非法进入该系统。应当避免这一情况，要确保调制解调器的电话号码不被列于电话簿上，并且最好将电话号码放在不同于本公司普通电话号码所在的交换机上。总之，不要假设没人知道自己的拨入号码 !大多数家庭计算机都能编程用一个调制解调器整天地依次调用拨号码，记录下连接上其他调制解调器的号码。如果可能，安装一个局域PBX，使得对外界的拨号产生一秒钟的拨号蜂音，并且必须输入一个与调制解调器相关联的扩展号码。

23.12 用户意识

Linux系统管理员的职责之一是保证用户安全。这其中一部分工作由用户的管理部门来完成，但是作为系统管理员，有责任发现和报告系统的安全问题，因为系统管理员负责系统的运行。

避免系统安全事故的方法是预防性的，当用户登录时，其外壳在给出提示前先执行/etc/profile文件，要确保该文件中的PATH指定最后搜索当前工作目录，这样将减少用户能运行特洛依木马的机会。将文件建立屏蔽值的设置放在该文件中也是很合适的，可将其值设置成至少将防止用户无意中建立任何人都能写的文件（ 022/026）。要小心选择此值，如果限制太严，则用户会在自己的配置文件中重新调用 umask以抵制系统管理员的意愿，如果用户大量使用小组权限共享文件，系统管理员就要设置限制小组存取权限的屏蔽值。系统管理员必须建立系统安全和用户的“ 痛苦量 ”间的平衡（痛苦量是安全限制引起的愤怒的函数）。定期地用grep命令查看用户配置文件中的umask，可了解系统安全限制是否超过了用户“ 痛苦量 ”极限。

系统管理员可每星期随机抽选一个用户，将该用户的安全检查结果（用户的登录情况简报，SUID/SGID文件列表等）发送给他的管理部门和他本人。主要有四个目的：

- 大多数用户会收到至少有一个文件检查情况的邮件，这将引起用户考虑安全问题（虽然并不意味着用户们会采取加强安全的行动）。
- 有大量可写文件的用户，将一星期得到一次邮件，直到他们取消可写文件的写许可为止。冗长的烦人的邮件信息也许足以促使这些用户采取措施，删除文件的写许可。
- 邮件将列出用户的SUID程序，引起用户注意自己有 SUID程序，使用户知道是否有不是自己建立的SUID程序。
- 送安全检查表可供用户管理自己的文件，并使用户知道对文件的管理关系到数据安全。如果系统管理员打算这样做，应事先让用户知道，以便他们了解安全检查邮件的目的。发送邮件是让用户具有安全意识，不要抱怨发送邮件。

管理意识是提高安全性的另一个重要因素。如果用户的管理部门对安全要求不强烈，系统管理员可能也忘记强化安全规则。最好让管理部门建立一套每个人都必须遵守的安全标准，如果系统管理员在此基础上再建立自己的安全规则，就强化了安全。管理有助于加强用户意识，让用户明确，信息是有价值的资产。

系统管理员应当使安全保护方法对用户尽可能地简单，提供一些提高安全的工具，如：公布锁终端的lock程序，让用户自己运行secure程序，将pwexp（检查用户口令信息的程序）放入/etc/profile中，使用户知道自己的口令时间。多教给用户一些关于系统安全的知识，确保用户知道自己的许可权限和umask命令的设置值。如果注意到用户在做蠢事，就给他们一些应当怎样做才对的提示。用户知道的关于安全的知识越多，系统管理员在保护用户利益方面做的事就越少。

23.13 系统管理员意识

23.13.1 保持系统管理员个人的登录安全

若系统管理员的登录口令泄密了，则窃密者离窃取 root只有一步之遥了，因为系统管理员经常作为root运行，窃密者非法进入到系统管理员的帐户后，将用特洛依木马替换系统管理员的某些程序，系统管理员将作为root运行这些已被替换的程序。正是因为这个原因，在Linux系统中，管理员的帐户最常受到攻击。即使su命令通常要在任何都不可读的文件中记录所有想成为root的企图，还可用记帐数据或ps命令识别运行su命令的用户。正因为如此，系统管理员作为root运行程序时应当特别小心，因为最微小的疏忽也可能“沉船”。下列一些指导规则可使系统管理员驾驶一艘“坚固的船”：

- 不要作为root或以自己的登录帐户运行其他用户的程序，首先用su命令进入用户的帐户。
- 决不要把当前工作目录排在PATH路径表的前边，那样实际是招引特洛依木马。当系统管理员用su命令进入root时，他的PATH将会改变，就让PATH保持这样，以避免特洛依木马的侵入。
- 敲入/bin/su执行su命令。若有su源码，将其改成必须用全路径名运行（即su要确认argv[0]的头一个字符是“/”才运行）。随着时间的推移，用户和管理员将养成敲/bin/su的习惯。
- 不要未注销帐户就离开终端，特别是作为root用户时更不能这样。当系统管理员作为root用户时，命令提示符是“#”，这个提示符对某些人来说可能是个红灯标志。
- 不允许root在除控制台外的任何终端登录（这是login的编译时的选项），如果没有login源码，就将登录名root改成别的名，使破坏者不能在root登录名下猜测各种可能的口令，从而非法进入root的帐户。
- 经常改变root的口令。
- 确认su命令记下的想运行su企图的记录/usr/adm/sulog，该记录文件的许可方式是600，并属root所有。这是非法者喜欢选择来替换成特洛依木马的文件。
- 不要让某人作为root运行，即使是几分钟，即使是系统管理员在一旁注视着也不行！

23.13.2 保持系统安全

- 考虑系统中一些关键的薄弱环节：
- 系统是否有调制解调器?电话号码是否公布？
- 系统是否连接到网络？还有什么系统也连接到该网络？
- 系统管理员是否使用未知来处或来处不可靠的程序？
- 系统管理员是否将重要信息放在系统中？
- 系统的用户是熟悉系统的使用还是新手？
- 用户是否很重视关心安全?
- 用户的管理部门是否重视安全？

- 保持系统文件安全的完整性。检查所有系统文件的存取许可，任何具有 SUID许可的程序都是非法者想偷换的选择对象。
- 要特别注意设备文件的存取许可。
- 要审查用户目录中具有系统ID/系统小组的SUID/SGID许可的文件。
- 在未检查用户的文件系统的SUID/SGID程序和设备文件之前，不要安装用户的文件系统。
- 将磁盘的备份存放在安全的地方。
- 设置口令时效，如果能存取Linux的源码，将加密口令和信息移到仅对root可读的文件中，并修改系统的口令处理子程序。这样可增加口令的安全。修改 passwd，使passwd能删去口令打头和末尾的数字，然后根据spell词典和/etc/passwd中用户的个人信息，检查用户的新口令，也检查用户新口令中子串等于登录名的情况。如果新口令是 spell词典中的单词，或/etc/passwd中的入口项的某项值，或是登录名的子串，passwd将不允许用户改变口令。
- 记录本系统的用户及其授权使用的系统。
- 查出久未使用的登录帐户，并取消该帐户。
- 确保没有无口令的登录帐户。
- 启动记帐系统。
- 查出不寻常的系统使用情况，如大量地占用磁盘，大量地使用 CPU时间，大量的进程，大量使用su的企图，大量无效地登录，大量的到某一系统的网络传输，奇怪的 uucp请求。
- 修改外壳，使其等待了一定时间而无任务时终止运行。
- 修改login，使其打印出用户登录的最后时间，三次无效登录后，将通信线挂起，以便系统管理员能检查出是否有人试图非法进入系统。确保 login不让root在除控制台外的任何地方登录。
- 修改su，使得只有root能以过期口令通过su进入某一帐户。
- 当安装来源不可靠的软件时，要检查源码和makefile文件，查看特殊的子程序调用或命令。
- 即使是安装来源可靠的软件，也要检查是否有 SUID（SGID）程序，确认这些许可的确是必要的。如果可能，不要让这些程序具有系统 ID（或组）的SUID（SGID）许可，而应该建立一个新用户（或给）供该软件运行。
- 如果系统在办公室中，门应上锁，将重要数据保存在软盘上或磁带上，并锁起来。
- 将secure、perms和任何其他做安全检查的外壳程序存取许可置为仅执行，更好的办法是将这些外壳程序存于可拆卸的介质上。
- 记住，只要系统有任何人都可调用的拨号线，系统就不可能真正的安全。系统管理员可以很好地防止系统受到偶然的破坏。但是那些有耐心、有计划、知道自己在干什么的破坏者，对系统直接的有预谋的攻击却常常能成功。
- 如果系统管理员认为系统已经泄密，则应当设法查出肇事者。若肇事者是本系统的用户，与用户的管理部门联系，并检查该用户的文件，查找任何可疑的文件，然后对该用户的登录小心地监督几个星期。如果肇事者不是本系统的用户，可让本公司采取合法的措施，并要求所有的用户改变口令，让用户知道出了安全事故，用户们应当检查自己的文件是否有被篡改的迹象。

如果系统管理员认为系统软件已被更改了，就应当从原版系统带（或 ;软盘）上重装入所有系统软件，保持系统安全比道歉更好。

第24章　系统程序员安全

Linux系统为程序员提供了许多子程序，这些子程序可存取各种安全属性。有些是信息子程序,返回文件属性，实际的和有效的 UID、GID等信息。有些子程序可改变文件属性。 UID、GID等有些处理口令文件和小组文件，还有些完成加密和解密。

本章主要讨论有关系统子程序，标准C库子程序的安全，如何写安全的C程序并从root的角度介绍程序设计（仅能被root调用的子程序）。

24.1　系统子程序

24.1.1　I/O子程序

• creat （）

建立一个新文件或重写一个暂存文件。需要两个参数：文件名和存取许可值（8进制方式）。如：creat（“ /usr/pat/read_write ”，0666） /* 建立存取许可方式为0666的文件 */调用此子程序的进程必须要有建立的文件的所在目录的写和执行许可，置给 creat （）的许可方式变量将被umask （）设置的文件建立屏蔽值所修改，新文件的所有者和小组由有效的 UID和GID决定。

返回值为新建文件的文件描述符。

• fstat （）

见后面的stat （）。

• open （）

在C程序内部打开文件。需要两个参数：文件路径名和打开方式（ I、O、I&O）。如果调用此子程序的进程没有对于要打开的文件的正确存取许可（包括文件路径上所有目录分量的搜索许可），将会引起执行失败。如果此子程序被调用去打开不存在的文件，除非设置了 O_CREAT标志，调用将不成功。此时，新文件的存取许可将作为第三个参数（可被用户的 umask修改）。当文件被进程打开后再改变该文件或该文件所在目录的存取许可，不影响对该文件的 I/O操作。

• read （）

从已由open （）打开并用作输入的文件中读信息。它并不关心该文件的存取许可。一旦文件作为输入打开，即可从该文件中读取信息。

• write （）

输出信息到已由open （）打开并用作输出的文件中。它同 read （）一样也不关心该文件的存取许可。

24.1.2　进程控制

1. exec （）族

包括execl （）、execv （）、execle （）、execve （）、execlp （）和execvp （），可将一可执行模快拷贝到调用进程占有的存贮空间。正被调用进程执行的程序将不复存在，新程序取代其位置。这是Linux系统中一个程序被执行的唯一方式：用将执行的程序复盖原有的程序。

安全注意事项：

- 实际的和有效的UID和GID传递给由exec（）调入的不具有SUID和SGID许可的程序。
- 如果由exec（）调入的程序有SUID和SGID许可，则有效的UID和GID将设置给该程序的所有者或小组。
- 文件建立屏蔽值将传递给新程序。
- 除设了对exec（）关闭标志的文件外，所有打开的文件都传递给新程序。用 fcntl（）子程序可设置对exec（）的关闭标志。

2. fork（）

用来建立新进程。其建立的子进程是与调用 fork（）的进程（父进程） 完全相同的拷贝（除了进程号外）。安全注意事项：

- 子进程将继承父进程的实际和有效的UID和GID。
- 子进程继承文件方式建立屏蔽值。
- 所有打开的文件传给子进程。

3. signal （）

允许进程处理可能发生的意外事件和中断。需要两个参数：信号编号和信号发生时要调用的子程序。信号编号定义在 signal.h中。信号发生时要调用的子程序可由用户编写，也可用系统给的值，如：SIG_IGN则信号将被忽略，SIG_DFL则信号将按系统的缺省方式处理。如许多与安全有关的程序禁止终端发中断信息（BREAK和DELETE），以免自己被用户终端终止运行。有些信号使Linux系统的产生进程的核心转储（进程接收到信号时所占内存的内容，有时含有重要信息），此系统子程序可用于禁止核心转储。

24.1.3 文件属性

1.access（）

检测指定文件的存取能力是否符合指定的存取类型。需要两个参数：文件名和要检测的存取类型（整数）。

存取类型定义如下：

0：检查文件是否存在。

1：检查是否可执行（搜索）。

2：检查是否可写。

3：检查是否可写和执行。

4：检查是否可读。

5：检查是否可读和执行。

6：检查是否可读可写可执行。

这些数字的意义和chmod命令中规定许可方式的数字意义相同。此子程序使用实际的 UID和GID检测文件的存取能力（一般有效的UID和GID用于检查文件存取能力）。

返回值：0:许可 -1:不许可。

2. chmod （）

将指定文件或目录的存取许可方式改成新的许可方式。

需要两个参数：文件名和新的存取许可方式。

3. chown （）

同时改变指定文件的所有者和小组的UID和GID。（与chown命令不同。）

由于此子程序同时改变文件的所有者和小组，故必须取消所操作文件的SUID和SGID许可，以防止用户建立SUID和SGID程序，然后运行chown（）去获得别人的权限。

4. stat（）

返回文件的状态（属性）。需要两个参数：文件路径名和一个结构指针，指向状态信息的存放的位置。

结构定义如下:

st_mode:	文件类型和存取许可方式。
st_ino:	I节点号。
st_dev:	文件所在设备的ID。
st_rdev:	特别文件的ID。
st_nlink:	文件链接数。
st_uid:	文件所有者的UID。
st_gid:	文件小组的GID。
st_size:	按字节计数的文件大小。
st_atime:	最后存取时间（读）。
st_mtime:	最后修改时间（写）和最后状态的改变。
st_ctime:	最后的状态修改时间。
返回值: 0:	成功1:失败。

5. umask（）

将调用进程及其子进程的文件建立屏蔽值设置为指定的存取许可。

需要一个参数: 新的文件建立屏值。

24.1.4 UID和GID的处理

1） getuid（） 返回进程的实际UID。

2） getgid（） 返回进程的实际GID。

以上两个子程序可用于确定是谁在运行进程。

3） geteuid（） 返回进程的有效UID。

4） getegid（） 返回进程的有效GID。

以上两个子程序用于确定某程序是否在运行某用户而不是其他用户的 SUID程序时很有用，可调用它们来检查确认本程序的确是以该用户的SUID许可在运行。

5） setuid（）： 用于改变有效的UID。

对于一般用户，此子程序仅对要在有效和实际的UID之间变换的SUID程序才有用（从原有效UID变换为实际UID），以保护进程不受到安全危害。实际上该进程不再使用 SUID方式运行。

6） setgid（） 用于改变有效的GID。

24.2 标准C程序库

24.2.1 标准I/O

1. fopen（）

打开一个文件供读或写，安全方面的考虑同open（）一样。

2. fread（）、getc（）、fgetc（）、gets（）、scanf（）和fscanf（）

从已由fopen（）打开供读的文件中读取信息。它们并不关心文件的存取许可。这一点同read（）。

3. fwrite（）、put（）、fputc（）、puts，fputs（）、printf（）、fprintf（）

把信息写到已由fopen（）打开供写的文件中。它们也不关心文件的存取许可。这一点同write（）。

4. getpass（）

从终端上读取至多8个字符长的口令，不回显用户输入的字符。

需要一个参数：提示信息。该子程序将提示信息显示在终端上，禁止字符回显功能，从/dev/tty读取口令，然后再恢复字符回显功能，返回刚敲入的口令的指针。

5. popen（）

将在后面的运行外壳中介绍。

24.2.2 /etc/passwd的处理

有一组子程序可对/etc/passwd文件进行方便地存取，可在入口项对文件读取、写入或更新等等。

1. getpwuid（）

从/etc/passwd文件中获取指定的UID的入口项。

2. getpwnam（）

对于指定的登录名，在/etc/passwd文件检索入口项。以上两个子程序返回一个指向 passwd结构的指针，该结构定义在/usr/include/pwd.h中，定义如下：

```
  struct passwd {
 char * pw_name;  /* 登录名 */
 char * pw_passwd;/* 加密后的口令 */
 uid_t  pw_uid;   /* UID */
 gid_t  pw_gid;   /* GID */
 char * pw_age;   /* 代理信息 */
 char * pw_comment; /* 注释 */
 char * pw_gecos;
 char * pw_dir;   /* 主目录 */
 char * pw_shell; /* 使用的外壳 */
  };
```

3. getpwent（）、setpwent（）、endpwent（）

对口令文件作后续处理。

首次调用getpwent（），打开/etc/passwd并返回指向文件中第一个入口项的指针，保持调用之间文件的打开状态。再调用 getpwent（）可顺序地返回口令文件中的各入口项。调用setpwent（）把口令文件的指针重新置为文件的开始处。使用完口令文件后调用 endpwent（）关闭口令文件。

4. putpwent（）

修改或增加/etc/passwd文件中的入口项。

此子程序将入口项写到一个指定的文件中，一般是一个临时文件，直接写口令文件是很危险的。最好在执行前做文件封锁，使两个程序不能同时写一个文件。算法如下：

• 建立一个独立的临时文件，即/etc/passnnn，nnn是PID号。

- 建立新产生的临时文件和标准临时文件 /etc/ptmp的链，若建链失败，则为有人正在使用/etc/ptmp，等待，直到/etc/ptmp可用为止或退出。
- 将/etc/passwd拷贝到/etc/ptmp，可对此文件做修改。
- 将/etc/passwd移到备份文件/etc/opasswd。
- 建立/etc/ptmp和/etc/passwd的链。
- 断开/etc/passnnn与/etc/ptmp的链。

注意 临时文件应建立在/etc目录，才能保证文件处于同一文件系统中，建链才能成功，且临时文件不会不安全。此外，若新文件已存在，即便建链的是 root用户，也将失败，从而保证了一旦临时文件成功地建链后没有人能再插进来干扰。当然，使用临时文件的程序应确保清除所有临时文件，正确地捕捉信号。

24.2.3 /etc/group的处理

有一组类似于前面的子程序处理 /etc/group的信息，使用时必须用 include语句将/usr/include/grp.h文件加入到自己的程序中。该文件定义了 group结构，将由getgrnam（）、getgrgid（）、getgrent（）返回group结构指针。

1. getgrnam（）

在/etc/group文件中搜索指定的小组名，然后返回指向小组入口项的指针。

2. getgrgid（）

类似于前一子程序，不同的是搜索指定的GID。

3. getgrent（）

返回group文件中的下一个入口项。

4. setgrent（）

将group文件的文件指针恢复到文件的起点。

5. endgrent（）

用于完成工作后，关闭group文件。

6. getuid（）

返回调用进程的实际UID。

7. getpruid（）

以getuid（）返回的实际UID为参数，确定与实际UID相应的登录名，或指定一个UID为参数。

8. getlogin（）

返回在终端上登录的用户的指针。

系统依次检查STDIN、STDOUT、STDERR是否与终端相联，与终端相联的标准输入用于确定终端名，终端名用于查找列于 /etc/utmp文件中的用户，该文件由login维护，由who程序用来确认用户。

9. cuserid（）

首先调用getlogin（），若getlogin（）返回NULL指针，再调用 getpwuid （getuid（））。

10. logname

列出登录进终端的用户名。

11. who am I

显示出运行这条命令的用户的登录名。

12. id

显示实际的UID和GID（若有效的UID和GID和实际的不同时也显示有效的UID和GID）和相应的登录名。

24.2.4 加密子程序

1977年1月，NBS宣布一个用于美国联邦政府ADP系统的网络的标准加密法:数据加密标准即DES用于非机密应用方面。DES一次处理64BITS的块，56位的加密键。

1. setkey（）和encrypt（）

提供用户对DES的存取。

这两个子程序都取64bits长的字符数组，数组中的每个元素代表一个位，为0或1。setkey（）设置将按DES处理的加密键，忽略每第8位构成一个56位的加密键。encrypt（）然后加密或解密给定的64bits长的一块，加密或解密取决于该子程序的第二个变元，0:加密 1:解密。

2. crypt（）

是Linux系统中的口令加密程序，也被/usr/lib/makekey命令调用。

crypt（）子程序与crypt命令无关，它与/usr/lib/makekey一样取8个字符长的关键词，2个salt字符。关键词送给setkey（），salt字符用于混合encrypt（）中的DES算法，最终调用encrypt（）重复25次，加密一个相同的字符串。返回加密后的字符串指针。

24.2.5 运行外壳

1. system（）

运行/bin/sh执行其参数指定的命令，当指定命令完成时返回。

2. popen（）

类似于system（），不同的是命令运行时，其标准输入或输出联到由 popen（）返回的文件指针。

二者都调用fork（）、exec（）、popen（）还调用pipe（），完成各自的工作，因而fork（）和exec（）的安全方面的考虑开始起作用。

24.3 编写安全的C程序

24.3.1 需要考虑的安全问题

一般有两方面的安全问题，在写程序时必须考虑:

• 确保自己建立的任何临时文件不含有机密数据，如果有机密数据，设置临时文件仅对自己可读/写。确保建立临时文件的目录仅对自己可写。

• 确保自己要运行的任何命令（通过system（）、popen（）、execlp（）、execvp（）运行的命令）的确是自己要运行的命令，而不是其他什么命令，尤其是自己的程序为 SUID或SGID许可时要小心。

第一方面比较简单，在程序开始前调用 umask（077）。若要使文件对其他人可读，可再调chmod（），也可用下述语名建立一个“ 不可见 ”的临时文件。

```
creat（"/tmp/xxx",0）;
file=open（"/tmp/xxx",O_RDWR）;
unlink（"/tmp/xxx"）;
```

文件/tmp/xxx建立后，打开，然后断开链，但是分配给该文件的存储器并未删除，直到最终指向该文件的文件通道被关闭时才被删除。打开该文件的进程和它的任何子进程都可存取这个临时文件，而其他进程不能存取该文件，因为它在/tmp中的目录项已被unlink（）删除。

第二方面比较复杂而微妙，由于system（）、popen（）、execlp（）、execvp（）执行时，若不给出执行命令的全路径，就能“骗”用户的程序去执行不同的命令。因为系统子程序是根据PATH变量确定哪种顺序搜索哪些目录，以寻找指定的命令，这称为SUID陷井。最安全的办法是在调用system（）前将有效UID改变成实际UID，另一种比较好的方法是以全路径名命令作为参数。execl（）、execv（）、execle（）、execve（）都要求全路径名作为参数。对付SUID陷井的另一方式是在程序中设置PATH，由于system（）和popen（）都启动外壳，故可使用外壳句法。如:

```
system（"PATH=/bin:/usr/bin cd"）;
```

这样允许用户运行系统命令而不必知道要执行的命令在哪个目录中，但这种方法不能用于execlp（）和execvp（）中，因为它们不能启动外壳执行调用序列传递的命令字符串。关于外壳解释传递给system（）和popen（）的命令行的方式，有两个其他的问题:

• 外壳使用IFS 外壳变量中的字符，将命令行分解成单词（通常这个外壳变量中是空格、tab、换行），如IFS中是/，字符串/bin/ed被解释成单词bin，接下来是单词ed，从而引起命令行的曲解。

• 再强调一次:在通过自己的程序运行另一个程序前，应将有效UID改为实际的UID，等另一个程序退出后，再将有效UID改回原来的有效UID。

24.3.2 SUID/SGID程序指导准则

1） 不要写SUID/SGID程序，大多数时候无此必要。

2） 设置SGID许可，不要设置SUID许可，应独自建立一个新的小组。

3） 不要用exec（）执行任何程序。记住exec（）也被system（）和popen（）调用。

• 若要调用exec（）（或system（）、popen（）），应事先用setgid（getgid（）） 将有效GID置加实际GID。

• 若不能用setgid（），则调用system（）或popen（）时，应设置IFS:

```
popen（"IFS=\t\n;export IFS;/bin/ls","r"）;
```

• 使用要执行的命令的全路径名。

• 若不能使用全路径名，则应在命令前先设置PATH:

```
popen（"IFS=\t\n;export IFS;PATH=/bin:/usr/bin;/bin/ls","r"）;
```

• 不要将用户规定的参数传给system（）或popen（）;若无法避免则应检查变元字符串中是否有特殊的外壳字符。

• 若用户有个大程序，调用exec（）执行许多其他程序，这种情况下不要将大程序设置为SGID许可。可以写一个（或多个）更小、更简单的SGID程序执行必须具有SGID许可的任务，然后由大程序执行这些小SGID程序。

4） 若用户必须使用SUID而不是SGID，以相同的顺序记住2）、3）项内容，并相应调整。不要设置root的SUID许可，选一个其他帐户。

5） 若用户想给予其他人执行自己的外壳程序的许可，但又不想让他们能读该程序，可将程序设置为仅执行许可，并只能通过自己的外壳程序来运行。

24.3.3 编译、安装SUID/SGID程序的方法

1）确保所有的SUID/SGID程序是对于小组和其他用户都是不可写的，存取权限的限制低于4755（2755）将带来麻烦，只能更严格。4111（2111）将使其他人无法寻找程序中的安全漏洞。

2）警惕外来的编码和make/install方法。

- 某些make/install方法不加选择地建立SUID/SGID程序。
- 检查违背上述指导原则的SUID/SGID许可的编码。
- 检查makefile文件中可能建立SUID/SGID文件的命令。

24.4 root用户程序的设计

有若干个子程序可以从有效UID为0的进程中调用。许多前面提到的子程序，当从root进程中调用时，将完成和原来不同的处理。主要是忽略了许可权限的检查。

由root用户运行的程序当然是root进程（SUID除外），因有效UID用于确定文件的存取权限，所以从具有root的程序中，调用fork（）产生的进程，也是root进程。

1. setuid（）

从root进程调用setuid（）时，其处理有所不同， setuid（）将把有效的和实际的UID都置为指定的值。这个值可以是任何整型数。而对非 root进程则仅能以实际UID或本进程原来有效的UID为变量值调用setuid（）。

2. setgid（）

在系统进程中调用setgid（）时，与setuid（）类似，将实际和有效的GID都改变成其参数指定的值。

调用以上两个子程序时，应当注意下面几点：

- 调用一次setuid（）（setgid（））将同时设置有效和实际UID（GID），独立分别设置有效或实际UID（GID）固然很好，但无法做到这点。
- setuid（）（setgid（））可将有效和实际UID（GID）设置成任何整型数，其数值不必一定与/etc/passwd（/etc/group）中用户（小组）相关联。
- 一旦程序以一个用户的UID调用了setuid（），该程序就不再做为root运行，也不可能再获root特权。

3. chown（）

当root进程运行chown（）时， chown（）将不删除文件的SUID和/或SGID许可，但当非root进程运行chown（）时， chown（）将取消文件的SUID和/或SGID许可。

4. chroot（）

改变进程对根目录的概念，调用chroot（）后，进程就不能把当前工作目录改变到新的根目录以上的任一目录，所有以/开始的路径搜索，都从新的根目录开始。

5. mknod（）

用于建立一个文件，类似于creat（），差别是mknod（）不返回所打开文件的文件描述符，并且能建立任何类型的文件（普通文件，特殊文件，目录文件）。若从非root进程调用mknod（）

将执行失败，只有建立FIFO特别文件（有名管道文件）时例外，其他任何情况下，必须从 root进程调用mknod（）。由于creat（）仅能建立普通文件，mknod（）是建立目录文件的唯一途径，因而仅有root能建立目录，这就是为什么mkdir命令具有SUID许可并属root所有。一般不从程序中调用mknod（）。通常用/etc/mknod命令建立特别设备文件，而这些文件一般不能在使用时建立和删除，mkdir命令用于建立目录。当用 mknod（）建立特别文件时，应当注意确保所建的特别文件不允许存取内存、磁盘、终端和其他设备。

6. unlink（）

用于删除文件。参数是要删除文件的路径名指针。当指定了目录时，必须从 root进程调用unlink（），这是必须从root进程调用unlink（）的唯一情况，这就是为什么 rmdir命令具有root的SGID许可的原因。

7. mount（）和umount（）

由root进程调用，分别用于安装和拆卸文件系统。这两个子程序也被 mount和umount命令调用，其参数基本和命令的参数相同。调用 mount（），需要给出一个特别文件和一个目录的指针，特别文件上的文件系统将安装在该目录下，调用时还要给出一个标识选项，指定安装的文件系统可被读/写（0）还是仅读（1）。umount（）的参数需要一个可拆卸的特别文件的指针。

第25章 Linux系统的网络安全

本章主要讨论网络和数据通信安全问题。

25.1 UUCP系统概述

UUCP系统是一组程序，可以完成文件传输，执行系统之间的命令，维护系统使用情况的统计，保护安全。UUCP是Linux系统最广泛使用的网络实用系统，这其中有两个原因：第一，UUCP是各种Linux版本都可用的唯一的标准网络系统，第二，UUCP是最便宜的网络系统。只需要一根电缆连接两个系统，就可建立UUCP。如果需要在相距数百或数千公里远的两个系统间传输数据，只需要两个具有拨号功能的调制解调器即可。

25.1.1 UUCP命令

UUCP命令之一是uucp，该命令用于两个系统间的文件传输，uucp命令格式类似于cp命令的格式，只是uucp允许用户在系统间拷贝文件，命令的一般格式如下：

```
uucp source_file destination_file
```

source_file通常是本系统的文件（但不必一定是），destination_file通常是另一系统的文件或目录。指定destination_file的格式为：

```
system!filename或system!directory
```

uucp给系统管理员提供了一个选项，可以把传入和传出本系统的 uucp文件只传到/usr/spool/uucppublic目录结构中。若告诉uucp将传输的文件存放在其他目录中，系统将会送回一个邮件：remote access to path / file denied。uucp允许以简化符号~代替/usr/spool/uucppublic/。如：

```
uucp names remote!~/john/names
```

有时也可用uucp将文件从另一个系统拷贝到本系统，只要将要传入本系统的文件指定为源文件（用system!file）即可，如：

```
uucp remotes!/usr/john/file1 file1
```

如果在远地机限制了文件传输的目录，上条命令不能拷贝到文件。拷贝文件到本系统的最安全的方法是：在两个系统上都通过uucppublic目录进行文件传输：

```
uucp remotes!~/john/file1 ~/pat/file1
```

25.1.2 uux命令

uux命令可用于在另一个系统上执行命令，这一特点称为“远程命令仞行”。uux最通常的用处是在系统之间发送邮件（mail在其内部执行uux）。典型的uux请求如下：

```
pr listing| uux - "remote1!lp -d pr1"
```

这条命令将文件listing格式编排后，再连接到系统remote1的打印机pr1上打印出来。uux的选项“-”使uux将本命令的标准输入设备建立为远程命令的标准输入设备。当若干个系统中只有一个系统连接了打印机时，常用uux打印文件。当然必须严格地限制远程命令进入，以保护系统安全。如：本系统不应允许其他系统上的用户运行下面的命令：

```
uux "yoursys!uucp yoursys!/etc/passwd （outside!~/passwd）"
```

这条命令将使本系统传送/etc/passwd文件到系统outside上，一般只有几条命令允许执行。rmail是加限制的mail程序，常常为允许通过 uux执行的命令之一。也允许 rnews（加限制的netnews伪脱机命令）在运行netnews的系统上执行，还允许lp在提供了打印设备的系统上运行。

25.1.3 uucico程序

uucp和uux命令实际上并不调用另一个系统及传送文件和执行命令，而是将用户的请求排入队列，并启动uucico程序。uucico完成实际的通信工作。它调用其他的系统，登录，传送数据（可以是文件或请求远程命令执行）。如果电话线忙，或其他系统已关机，传输请求仍针保留在队列中，uucico后续的职能操作（通常是cron完成）将发送这些传输请求。

uucico完成数据的发送和接收。在本系统的/etc/passwd文件中，有其他系统的uucico登录进入本系统的入口项，该入口项中指定的缺省外壳是 uucico。因此，其他系统调用本系统时，直接与uucico对话。

25.1.4 uuxqt程序

当另一系统的uucico调用本系统请求远程命令执行时，本系统的 uucico将该请求排入队列，并在退出之前，启动uuxqt程序执行远程命令请求。

下面举例说明数据是如何传输的。假设本系统的一个用户发送邮件给另一远程系统remote1的某人，mail会执行uux，在remote1系统上远程地运行 remail程序，要传送的邮件为remail命令的输入。uux将传输请求排入队列，然后启动uucico招待实际的远程调用和数据传输。如果remote1响应请求，uucico登录到remote1，然后传送两个文件：邮件和将在 remote1上由uuxqt执行的uux命令文件。uux命令文件中含有运行remail请求。如果remote1在被调时已关机，uucico则将无法登和传送文件，但是 cron会周期地（1小时）启动uucico。uucico查找是否有还未传送出的数据，若发现 uux指定的传输目标系统是 remote1，就尝试再调用 remote1，直到调通remote1为止，或者过了一定天数，仍未调通 remote1，未送出的邮件将作为“不可投递”的邮件退回给发送该邮件的用户。

25.2 UUCP的安全问题

UUCP系统未设置限制，允许任何本系统外的用户执行任何命令和拷贝进 /出uucp用户可读/写的任何文件。在具体的uucp应用环境中应了解这点，根据需要设置保护。

在UUCP中，有两个程序处理安全问题。第一个是 uucico程序，该程序在其他系统调用本系统时启动。这个程序是本系统uucp安全的关键，完成本系统文件传输的传进和传出。第二个程序是uuxqt，该程序为所有的远程命令执行服务。

25.2.1 USERFILE文件

uucico用文件/usr/lib/uucp/USERFILE确定远程系统发送或接收什么文件，其格式为：

```
login，sys[c] path_name [path_name…]
```

其中login是本系统的登录名，sys是远程系统名，c是可选的call_back标志， path_name是目录名。

uucico作为登录外壳启动时，将得到远程系统名和所在系统的登录名，并在 USERFILE文件中找到匹配login和sys的行。如果该行含有call_back标志c， uucico将不传送文件，连接断开，

调用远程系统（即，任何系统可以告诉本系统它的名是 xyz，于是本系统挂起，调用实际的 xyz 执行文件传输），若无c，uucico将执行远程系统请求的文件传送，被传送的文件名被假定为以path_name开头的。

用户需要了解以下几点：

- 如果远程系统使用的登录名未列于USERFILE的登录域中，uucico将拒绝允许其他系统做任何事，并挂起。
- 如果系统名未列于 sys域中，uucico将使用USERFILE中有匹配的登录名和空系统名的第一行，如：nuucp，/usr/spool/uucppublic应用到作为nuucp登录的所有系统。cbuucp，c将迫使作为cbuucp登录的所有系统自己执行文件传输的请求。若调用系统名不匹配 sys系统中的任何一个，并且无空入口项，uucico也将拒绝做任何事。
- 若两个机器都设置了 call_back标志，传送文件的请求决不会被执行，两个系统会一直互相调用，直到两个系统中的一个取消call_back时，才能进行文件传送。
- 如果一个用户的登录名列于USERFILE文件的login域中，则当调用本系统的uucico为该用户传送文件时，uucico只传送至path_name指定的目录中的文件。空登录名用于所有未明确列于USERFILE文件中的用户进行登录。所以 pat，/usr/pat 只允许pat传送/usr/pat目录结构中的文件。/usr/spool/uucppublic /tmp其他用户仅允许传送目录 /usr/spool/uucppublic和/tmp中的文件。不要允许uucico将文件拷进/出到除了/usr/spool/uucppublic目录以外的其他任何目录，否则可能会有人用下面的命令拷贝走本系统的重要信息：

```
uucp yoursys!/etc/passwd to-creep
```

25.2.2 L.cmds文件

uuxqt利用/usr/lib/uucp/L.cmds文件确定要执行的远程执行请求命令。该文件的格式是每行一条命令。如果只需uuxqt处理电子邮件，该文件中就只须一行命令：

```
rmail
```

系统管理员可允许登录用户执行netnews（rnews）的命令或远程打印命令（lp），但决不允许用户执行拷贝文件到标准输出的命令，如 cat命令或网络命令uucp，否则这些人只需在他们自己的系统上敲入：

```
uux "yoursys!uucp yoursys!/etc/passwd （outside!~/passwd）"
```

然后就可等待本系统发送出命令文件。

25.2.3 uucp登录

UUCP系统需要两个登录帐户，一个是其他系统登录的帐户，另一个是系统管理使用的帐户。例如，数据传输登录帐户是 nuucp，管理登录帐户是uucp，则在/etc/passwd文件中应当有两行。

UID和GID的5号通常留给uucp，由于uucico具有管理登录的SUID许可，因此nuucp帐户的UID和GID应当用其他值。

25.2.4 uucp使用的文件和目录

/usr/lib/uucp用于存放不能由用户直接运行的各种uucp，如uuxqt和uucico。该目录还含有若干个确定uucp如何操作的文件，如 L.cmds和USERFILE。这些文件只能对 uucp管理帐户可写（系统管理员一定不愿让用户更改远程可执行命令表）。根据安全的观点，该目录中另一个系统

管理员必须清楚的文件是L.sys。该文件中含有uucico能调用的每个系统的入口项。入口项数据包括uucico所调用系统的电话号码、登录名、未加密的口令。不用说， L.sys应当属于uucp管理帐户所有，且应当具有400或600存取许可。

uucp用/usr/spool/uucp目录存放工作文件。文件名以 C.开头的文件是送到其他系统的命令文件，含有在其他系统上拷入（ /出）数据和执行命令的请求。文件名以 D.开头的文件用作C.文件的数据文件。文件名以 X.开头的文件是来自其他系统的远程执行请求，由 uuxqt解释。文件名以 TM.开始的文件是从其他系统传送数据到本系统过程中 uucp所使用的暂存文件。XQTDIR是uuxqt用于执行 X.文件的目录。 LOGFILE可有助于管理 uucp的安全，它含有执行uucp请求成功与否的信息。系统管理员可时常查看该文件，了解有哪些系统正登录本系统执行uucp请求，是什么请求，特别要检查这些请求是否试图做不允许的操作。

25.3 HONEYDANBER UUCP

有两个主要的UUCP版本，第一个是与 Linux系统V一起颁布的，在本节将称为老 UUCP。另一个版本称为HONEYDANBER UUCP，由AT&T颁布。HONEYDANBER UUCP与老UUCP相比，有若干的优点：

1） 支持更多的拨号和网络。

- 智能自动拨号调制解调器以及标准 AT&T技术的801自动拨号器。
- 网络，如DATAKIT VCS、UNET/ETHERNET、3COM/ETHERNET、SYTEK、TCP（BSD Linux系统）。
- 连接到LAN的拨号器。
- X.25永久性虚拟环网（用X.25协议）。

2）重新组织了/usr/spool/uucp目录，在该目录下，对每个远程系统有一个目录。

3）加强了安全。

- USERFILE和L.cmds文件组合成一个文件Permissions。
- 可以在一级系统上指定远程可执行命令。
- 可分别控制文件传入和文件传出。
- 缺省的安全设置很严格。

25.3.1 HONEYDANBER UUCP与老UUCP的差别

HONEYDANBER UUCP中的/usr/lib/uucp/Systems文件是原来UUCP中的/usr/lib/uucp/L.sys。HONEYDANBER UUCP中/usr/spool/uucp/.log下的一个目录代替了老UUCP的文件/usr/spool/uucp/logFILE。/usr/spool/uucp/.log中的目录uucico， uucp， uux， uuxqt含有相应命令的记录文件，各目录对应最近处于活跃状态的远程系统都有一个记录文件（记录文件在这些目录中通常保存一个星期）。

如果一个调用本系统的远程系统未列于Systems文件中， uucico将不允许该远程系统执行任何操作，而是启动外壳程序 /usr/lib/uucp/remote.unknown， 由UUCP提供的该外壳程序的缺省版本将在/usr/spool/uucp/中。Admin/Foreign文件中记下远程系统的登录时间，如日期及系统名。只要使remote.unknown不可执行，就能禁止这一操作，以达到与老 UUCP兼容。

C.， D.， X.， TM.等文件存放在/usr/spool/uucp下的不同目录中，目录名就是文件对应的远程系统名。

在HONEYDANBER UUCP中USERFILE与L.cmds文件合并在一起，这个新文件

/usr/lib/uucp/Permissions提供了更灵活的授予外系统存取许可的控制方法。文件中的规则表定义了可以发出请示的各种系统。规则与选项的格式如下：

```
rule=list option=yes|no option=list...
```

其中rule是登录名或机器名，list是用以分隔各项的规则表（表中各项随rule或option而变），option是下边将讨论的各选项之一，或为一个选项表，或只取yes/no决定允许/不允许一项操作。

25.3.2 登录名规则

LOGNAME规则用于控制作为登录外壳启动的uucico。

```
LOGNAME=nuucp
```

指定对所有登录到nuucp帐户下的系统加缺省限制的方法如下：

- 远程系统只能发送文件到/usr/spool/uucppublic目录中。
- 远程系统不能请求接收任何文件。
- 当uucico调用远程系统时，才发送已排入队列要发送到该远程系统的文件。这是 uucico准确地识别远程系统的唯一方法（任何系统都可调用本系统并冒充是 xyz系统）。
- 由uuxqtux远程系统的名义可执行的命令是缺省规定的命令，这些缺省命令在编译时定义（通常只有rmail和rnews命令）。
- 可用冒号分隔开若干个其他系统的uucico的登录帐户，如下所示：

```
LOGNAME=nuucp：xuucp：yuucp
```

任何设有LOGNAME规则的系统，若要登录请求UUCP传送，都会被回绝（系统将给出错信息“ get lost ”，并挂起）。

一个LOGNAME规则就足够启动HONEYDANBER UUCP系统。事实上，当该系统运行时，将在Permissions文件中放一个无选项的LOGNAME规则，该规则应用于在/etc/passwd文件入口项外壳域中有/usr/lib/uucp/uucico的所有登录帐户。可使用若干选择忽略缺省限制，这些选项可组合，允许或限制各种操作。例如可用 WRITE选项指定一个或多个送入文件的目录，而不必送入/usr/spool/uucppublic目录，例如：

```
LOGNAME=nuucp WRITE=/
```

这一规则允许文件送入本系统的任何目录。 2-4项的限制依然保持。注意，远程UUCP请求可重写任何有写许可的文件，可指定多个写入文件的目录。用冒号分隔开，如下所示：

```
LOGNAME=nnuucp WRITE=/usr：/floppy
```

该规则允许远程系统将文件写到/usr和/floppy目录中。用REQUEST=yes选项可允许远程系统的用户从本系统拷贝文件，如：

```
LOGNAME=nuucp REQUEST=yes
```

能被拷贝的文件只能是存放在 /usr/spool/uucppublic目录中的文件， 1、3、4项的限制仍然有效。若要允许远程系统可从其他目录拷贝文件，用 READ选择：

```
LOGNAME=nuucp REQUEST=yes READ=/usr
```

该规则允许远程系统拷贝/usr目录中任何其他人可读的文件。也可像 WRITE选项一样指定目录表。用SENDFILES=yes选项可允许uucico在远程系统调用本系统时发送已排队的文件，如下所示：

```
LOGNAME=nuucp SENDFILES=yes
```

1、2、4项的限制依然有效。用 CALLBACK=yes选项迫使任何登录到指定帐户的系统 call back。

注意 CALLBACK=yes不能与其他选项组合作用。如果其他选项与这条选项列在一起，

其他选项将被忽略。

NOREAD和NOWRITE选项可分别与READ和WRITE选项一起使用。指定NOREAD选项下的目录表，可建立对READ选项的例外处理（即指出READ目录中不能由远程系统请求的目录），例如：

```
LOGNAME=nuucp，REQUEST=yes READ=/ NOREAD=/etc
```

该规则允许远程系统请求系统中任何其他人可读的文件，但不包括 /etc中的文件，NOWRITE，WRITE的联合用法与上类似。

一般来说，不要将缺省限制改得太多。若本系统被另一系统调去存贮电话费用或系统管理员没有办法拨出，可以用 SENDFILE选项。若要对某些机器取消限制，则应当建立一个仅用于那些机器的uucico登录帐户。例如：

```
LOGNAME=nuucp SENDFILES=yes
LOGNAME=trusted SENDFILES=yes REQUEST=yes READ=/ WRITE=/
```

上面的规则允许在trusted帐户下登录的系统在本系统中具有另一种文件存取许可， nuucp帐户的口令应送给所有要与本系统uucp建立连接的系统管理员，trusted帐户的口令则只能送给信任系统的管理员。如系统有信任和非信任的 uucp帐户，最好用PUBDIR选项为这两种帐户建立不同的公共帐户， PUBDIR允许系统管理员改变 uucico对公共目录的概念（缺省为/usr/spool/uucppublic）。例如：

```
LOGNAME=nuucp SENDFILES=yes REQUEST=yes \
PUBDIR=/usr/spool/uucppublic/nuucp
LOGNAME=trusted SENDFILES=yes REQUEST=yes READ=/ WRITE=/ \
PUBDIR=/usr/spool/uucppublic/trusted
```

上面的选项使要送到公共目录中的文件，对于不同登录 nuucp和trusted分别放入不同的目录中。这将防止登录到 nuucp的非信任系统在信任系统的公共目录中拷进和拷出文件（注意：上面的选项允许nuucp请求文件传送）。行尾倒斜杠指明下一行是该行的续行。用 MYNAME选项可以给登录进某一帐户的系统赋予一个系统名：

```
LOGNAME=Xuucp MYNAME=IOnker
```

25.3.3 MACHINE规则

MACHINE规则用于忽略缺省限制，在MACHINE规则中指定一个系统名表，就可使uucico调用这些系统时改变缺省限制。 READ、WRITE、REQUEST、NOREAD、NOWRITE、PUBDIR选项的功能与LOGNAME相同。忽略CALLBACK、SENDFILES选项，MYNAME选项所定义的必须与LOGNAME规则联用，指定将赋给调用系统的名，该名仅当调用所定义的系统时才用。MACHINE规则的格式如下：

```
MACHINE=zuul：gozur：enigma WRITE=/ READ=/
```

这条规则使远程系统zuul、gozar、enigma能够发送/请求本系统上任何其他人可读/写的文件。一般不要让远程系统在除/usr/spool/uucppublic目录外的其他目录下读写文件，因此，对于信任的系统也要少用 MACHINE规则。系统名OTHER用于为指定用户外的所有其他用户建立MACHINE规则。COMMANDS选项用于改变uuxqt通过远程请求执行的缺省命令表。

```
MACHINE=zuul COMMANDS=rmail：rnews：lp
```

上面的选项允许系统zuul请求远程执行命令rmail、rnews、lp。uucico不用这个选项。uuxqt用该选项确定以什么系统的名义执行什么命令。 COMMANDS选项所指定的命令将用缺省设置的路径PATH。PATH在编辑uuxqt时被建立通常设置为/bin：/usr/bin。在COMMANDS选项中给出全路径名可以忽略缺省PATH。

```
MACHINE=zuul COMMANDS=umail：/usr/local/bin/rnews：lp
```

同样地，对HONEYDANBER UUCP也应当像老UUCP一样不允许远程系统运行uucp或cat这样的命令。任何能读写文件的远程执行命令都可能威胁局域安全。虽然局域系统对远程系统名进行一定程序的校核，但是任何远程系统在调用局域系统时都可自称是“ xyz ”，而局域系统却完全相信是真的。因此局域系统的系统可能认为只允许了 zuul运行lp命令。但实际上任何自称是zuul的系统也允许运行lp命令。

有两种方法可以证实系统的身份。一种方法是拒绝用 CALLBACK=yes与调用系统对话。只要电话和网络线未被破密或改变，局域系统就能肯定地确认远程系统的身份。另一种方法是在LOGNAME规则中用VALIDATE选项。

若必须允许某些系统运行“ 危险 ”的命令，可联用 COMMANDS和VALIDATE选项，VALIDATE选项用于LOGNAME规则中指定某系统必须登录到LOGNAME规定的登录帐户下：

```
LOGNAME=trusted VALIDATE=zuul
MACHINE=COMMANDS=rmail：rnews：lp
```

当一个远程系统自称是zuul登录时，uucico将查Permissions文件，找到LOGNAME=trusted规则中的VALIDATE=zuul，若该远程系统使用了登录帐户trusted， uucico将认为该系统的确是zuul继续往下执行，否则 uucico将认为该系统是假冒者，拒绝执行其请求。只要唯有 zuul有trusted帐户的登录口令，其他系统就不能假冒它。仅当登录口令是保密的，没有公布给其他非信任的系统管理员或不安全的系统， VALIDATE选项才能奏效。如果信任系统的登录口令泄漏了，则任何系统都可伪装为信任系统。

在COMMANDS选项中给出ALL时，将允许通过远程请求执行任何命令。因此，不要使用ALL!规定ALL实际上就是把自己的帐户给了远程系统上的每一个用户。

25.3.4 组合MACHINE和LOGNAME规则

将MACHINE和LOGNAME规则组合在一行中，不管远程系统调用局域系统还是局域系统调用远程系统，可以确保一组系统的统一安全。

```
LOGNAME=trusted MACHINE=zuul：gozur VALIDATE=zuul：gozur \
REQUEST=yes SENDFILES=yes \
READ=/ WRITE=/ PUBDIR=/usr/spool/trusted \
COMMANDS=rmail：rnews：lp：daps
```

25.3.5 uucheck命令

一旦建立了Permissions文件，可用uucheck -v命令了解uucp如何解释该文件。其输出的前几行是确认 HONEYDANBER UUCP使用的所有文件、目录、命令都存在，然后是对Permissions文件的检查。

25.3.6 网关

邮件转送可用于建立一个网关机器。网关是一个只转送邮件给其他系统的系统。有了网关，使有许多Linux系统的部门或公司对其所有用户只设一个电子邮件地址。所有发来的邮件都通过网关转送到相应的机器。

网关也可用于加强安全：可将调制解调器连接到网关上，由网关转送邮件的所有系统通过局域网或有线通信线与网关通信。所有这些局域系统的电话号码， uucp登录帐户，口令不能对该组局域系统外的系统公布。如果有必要，可使网关是唯一连接了调制解调器的系统。建立一

个最简单的网关是很容易的：对每个登录进系统，想得到转送邮件的用户，只需在文件/usr/mail/login中放入一行：

```
Forward to system !login
```

要发送给帐户login的邮件进入网关后，将转送给登录在系统 system的帐户login下的用户。两个登录名可以不同。

网关建立了一个安全管理的关卡，网关的口令必须是不可猜测的。网关应尽可能只转送邮件而不做别的事，至少不要将重要数据存放在该机上。网关上还应做日常例行安全检查，并且要对uucp的登录进行仔细地检查。

网关也为坏家伙提供了一个入口：如果有人非法进入了网关，他将通过 uucp使用的通信线存取其他的局域系统和存取含有关于其他局域系统 uucp信息的Systems文件。若这人企图非法进入其他系统，这些信息将对他有很大的用处。

使用网关的经验如下：

- 若要建立网关，应确保其尽可能地无懈可击。
- 可在网关和局域系统间建立 uucp连接，使得局域系统定期的与网关通信获取邮件，而网关完全不用调用局域系统。这样做至少能防止一个坏家伙通过网关非法进入局域系统。
- 利用局域系统的Permissions文件对网关的行为加以限制，使其裸露程度达到最小，即只转发邮件。这样可使窃密者不能利用网关获取其他系统的文件。

25.3.7 登录文件检查

HONEYDANBER UUCP自动地将登录信息邮给uucp。login文件应当定期地读这个文件。系统管理员应当检查那些不成功的大量请求，特别是其他系统对本系统的文件请求。还要检查不允许做的远程命令执行请求。登录信息都保存在文件中，如果要查看，可用 grep命令查看。/usr/spool/uucp/.Log/uucico/system文件中含有uucico登录，/usr/spool/uucp/.Log/uuxqt/system文件含有uuxqt登录。下面一行命令将打印出uuxqt执行的所有命令（rmail除外）：

```
grep -v rmail /usr/spool/uucp/.Log/uuxqt/*
```

下面一行命令将打印所有对本系统文件的远程请求：

```
grep -v REMOTE /usr/spool/uucp/.Log/uucico/* | grep "<"
```

总之，HONEYDANBER UUCP比老UUCP提供了更强的安全特性，特别是提高了远程命令执行的安全性。

25.4 其他网络

25.4.1 远程作业登录

远程作业登录（remote job entry，RJE）系统提供了一组程序及相应的硬件，允许Linux系统与IBM主机上的作业输入子系统（job entry subsystems，JES）通信。可通过两条命令的send和usend存取RJE。send命令是RJE的通用的作业提供程序，它将提供文件给 JES，就好像这些作业文件是从卡片阅读机读入的穿孔卡片一样。 usend命令用于在使用了RJE系统的Linux系统间传送文件，它将建立一个作业（虚拟的一叠穿孔卡片），并以send命令的送文件的同样方式将该作业提供给JES。该作业卡片叠中的控制卡告诉 JES数据传送到何处（这里，数据是正被传送的文件）。文件传送的目的地是Linux系统，但JES认为是一个“行式打印机”。RJE系统通常以每秒9600位的速率与JES通信。典型的usend命令句法如下：

```
usend -d system -u login file（s）
```

system是挂到IBM JES上的另一个Linux系统名，login是另一个系统上的接收用户的登录名，file是用户希望传送的文件。

下面是几个关于RJE的安全问题：

- 缺省时，RJE将把文件传送到接收用户的HOME目录中的rje目录。该目录必须对其他人可写、可执行，这意味着存入rje目录的文件易受到检查、移动、修改。然而如果该目录的许可方式是733，其他用户就不能用ls列目录内容寻找感兴趣的文件。被建立的文件对所有者、小组或其他人都是可读的，所以通过 RJE网络传送的安全文件在系统上都是可读的。为什么这些问题不同于UUCP和/usr/uucppublic目录？
- UUCP定期地清除/usr/spool/uucppublic目录的内容，几天前或几星期前的老文件将被删除，通常用户将把自己的文件移出 uucppublic目录，以免文件被删除，而存在用户 rje目录中的文件不会被清除，所以有些用户从来不把自己的文件移到其他目录。
- 用户清楚地知道uucppublic目录是一个公共目录，存入重要信息之前，首先注意将其加密。但是用户却总是容易忘记自己rje目录实际上也是公共目录，经常忘记将重要文件加密。
- usend命令在其他人可写的目录中建立文件，并重写其他人可写的文件。
- RJE服务子程序只执行一些功能而不执行文件传送。 RJE系统像UUCP一样也执行远程命令，运行RJE的大多数系统用远程命令执行转送电子邮件。因为 RJE的传输率通常比UUCP更高。遗憾的是RJE没有像UUCP那样的能力限制能执行的命令和能存取的文件。一个好的经验是把连接到同一个JES的一组系统看作同一系统。

25.4.2 NSC网络系统

网络系统公司（network systems corporation，NSC）宽信道网络是一个高速局域网络。NSC可将数千个最远相距5000英尺的系统挂在一起，传输速率可高达 50MBIT/S，NSC也可通过通信（如微波或人造卫星）线连接不同系统。

Linux用户可通过nusend命令存取NSC宽信道，nusend命令的句法与usend命令相同，除用-c选项传送其他人不可存取的文件外，大多数情况下，nusend的用法与usend是一样的，换言之，如果无-c选项，文件就是可读的，而且文件路径名中列出所有目录对其他人也都是可搜索的，前边讨论过的关于RJE的安全问题的考虑也适合于NSC网络。

可查看NSC记录文件，了解NSC是否正在执行任何不应执行的命令。记录文件保存在目录/usr/nsc/log中。下面的命令将打印出所有由NSC在本系统上执行的命令（rmail除外）：

```
grep execute /usr/nsc/log/LOGFILE|grep -v rmail
```

25.5 通信安全

有两种方法可以提供安全的通信，第一种是保证传输介质的物理安全，任何人都不可能在传输介质上接上自己的窃密线，第二种方法是加密重要数据。下面分别介绍这两种方法。

25.5.1 物理安全

如果所有的系统都锁在屋里，并且所有连接系统的网络和接到系统上的终端都在上锁的同一屋内，则通信与系统一样安全（假定没有调制解调器）。但是系统的通信线在上锁的室外时，就会发生问题了。

尽管从网络通信线提取信息所需要的技术，比从终端通信线获取数据的技术高几个数量级，

但同样会存在安全问题。用一种简单的（但很昂贵）高技术 —— 加压电缆，可以获得通信的物理安全。这一技术是若干年前，为美国国家电话系统开发的。通信电缆密封在塑料中，埋置于地下，并在线的两端加压。线上连接了带有报警器的监示器，用来测量压力。如果压力下降，则意味电缆可能破了，维修人员将寻找与修复出问题的电缆。电缆加压技术提供了安全的通信线。电缆不用埋置于地下，可架于整座楼中，每寸电缆都将暴露在外。如果任何人企图割电缆，监示器会自动报警器，通知安全保卫人员电缆已被破坏。如果任何人成功地在电缆上接了自己的通信线，安全人员定期地检查电缆的总长度，应可以发现电缆拼接处。加压电缆是屏蔽在波纹铝钢包皮中的，因此几乎没有电磁发射，如果要用电磁感应窃密，势必需用大型可见的设备。

采用这样的电缆，终端就不必锁在办公室，而只需将安全电缆的端头锁在办公室的一个盒子里。另一个增加外部终端物理安全的方法是，在每天下午 5点使用计算机的时间结束时，即当所有用户回家时，断开终端的连接。这样某人若想非法进入系统，将不得不试图在白天人们来来回回的时间里获取终端的存取权，或不得不在下午 5点后试图潜入计算机房（如果5点后计算机房有操作人员或有安全人员，潜入计算机房的企图就不可能得逞）。

光纤通信线曾被认为是不可搭线窃听的，其断破处立即可被检测到，拼接处的传输速度会令人难以忍耐地缓慢。光纤没有电磁辐射，所以也不能用电磁感应窃密。不幸的是光纤的最大长度有限制，长于这一长度的光纤系统必须定期地放大（复制）信号。这就需要将信号转换成电脉冲，然后再恢复成光脉冲，继续通过另一条线传送。完成这一操作的设备（复制器）是光纤通信系统的安全薄弱环节，因为信号可能在这一环节被搭线窃听。有两个办法可解决这一问题，距离大于最大长度限制的系统间，不要用光纤线通信（目前，网络覆盖范围半径约 100公里），或加强复制器的安全（用加压电缆、警报系统、警卫）。

25.5.2 加密

加密也可提高终端和网络通信的物理安全，有三种方法加密传输数据：

- 链接加密法：在网络节点间加密，在节点间传输加密，传送到节点后解密，不同节点对间用不同的密码。
- 节点加密法：与链接加密类似，不同的只是当数据在节点间传送时，不用明码格式传送，而是用特殊的加密硬件进行解密和重加密，这种专用硬件通常旋转在安全保险箱中。
- 首尾加密法：对进入网络的数据加密，然后待数据从网络传送出后再进行解密。网络本身并不会知道正在传送的数据是加密数据。这一方法的优点是，网络上的每个用户（通常是每个机器的一个用户）可有不同的加密关键词，并且网络本身不需增添任何专门的加密设备。缺点是每个系统必须有一个加密设备和相应的软件（管理加密关键词）。或者每个系统必须自己完成加密工作（当数据传输率是按兆位 /秒的单位计算时，加密任务的计算量是很大的）。

终端数据加密是一种特殊情况，此时链接加密法和首尾加密法是一样的方法，终端和计算机都是既为节点又为终止端点。

通信数据加密常常不同于文件加密，加密所用的方法不应降低数据的传送速度。丢失或被歪曲了的数据不应当引起丢失更多的数据位，即解密进程应当能修复坏数据，而不能由于坏数据对整个文件或登录进行不正确地解密。对于登录会话，必须一次加密一个字节，特别是在Linux系统的情况下，系统要将字返回给用户，更应一次加密一个字节。在网络中，每一链可能需要不同的加密关键字，这就提出了对加密关键词的管理、分配和替换问题。

DES传送数据的一般形式是以代入法密码格式按块传送数据，不能达到上述的许多要求。DES采用另一加密方法，一次加密一位或一个字节，形成密码流。密码流具有自同步的特点，被传送的密码文本中发生的错误和数据丢失，将只影响最终的明码文本的一小段（ 64位），这称为密码反馈。在这种方法中， DES被用作虚拟随机数发生器，产生出一系列用于对明码文本的随机数。明码文本的每n位与一个DES n位的加密输出数进行异或， n的取值为1～64，DES加密处理的输入是根据前边传送的密码文本形成的64位的数据。

当n为1时，加密方法是自同步方式：错一位或丢失 1位后，64位的密码文本将不能被正确地解密，因为不正确的加密值将移入DES输入的末端。但是一旦接收到正确的64位密码，由于DES的加密和解密的输入是同步的，故解密将继续正确地进行。

DES的初始输入值称为种子，是一个同时由传输器和接收器认可的随机数。通常种子由一方选择，在加密前给另一方。而加密关键词不能以明码格式通过网络传送，当加密系统加电时在两边都写入加密关键词，并且在许多阶段期间加密关键词都保持不变，用户可以选择由主关键词加密的阶段关键词，发送到数据传送的另一端，当该阶段结束后，阶段关键词就不再使用了。主关键词对用户是不可见的，由系统管理员定期改变，选择哪一种关键词管理方法，常由所用的硬件来确定。如果加密硬件都有相应的设备，则用种子还是用主关键词、阶段关键词是无关紧要的。

25.5.3 用户身份鉴别

口令只是识别一个用户的一种方法，实际上有许多方法可以用来识别用户，下面介绍其中的几种。

- 回叫（call back）调制解调器方法：是维护系统有效用户表及其相应电话号码的设备。当用户拨号调用系统时，回叫调制解调器获得用户的登录帐户，先挂起，再回头调用用户的终端。这种方法的优点是，限制只有电话号码存于调制解调器中的人才是系统的用户，从而使非法侵入者不能从其家里调用系统并登录，这一方法的缺点是限制了用户登录的灵活性，并仍需要使用口令，因为调制解调器不能仅从用户发出调用的地方，唯一地标识用户。
- 标记识别方法：标记是口令的物理实现，许多标记识别系统使用某种形式的卡（如背面有磁条的信用卡），这种卡含有一个编码后的随机数。卡由连接到终端的阅卡机读入，不用再敲入口令。为了增加安全性，有的系统要求读入卡和敲入口令。有些卡的编码方法使得编码难于复制。标记识别的优点是，标识可以是随机的，并且必须长于口令。不足之处是每个用户必须携带一个卡（卡也可与公司的徽记组合使用）。并且每个终端上必须连接一个阅读机。
- 一次性口令方法：即“ 询问应答系统 ”。这种系统允许用户每次登录时使用不同的口令。这种系统使用一种称做口令发生器的设备，设备是便携式的（大约为一个袖珍计算器的大小），并有一个加密程序和唯一的内部加密关键词。系统在用户登录时给用户提供一个随机数，用户将这个随机数送入口令发生器，口令发生器用用户的关键词对随机数加密，然后用户再将口令发生器输出的加密口令（回答）送入系统，系统将用户输入的口令，与它用相同的加密程序、关键词和随机数产生的口令进行比较，如果二者相同，允许用户存取系统。这种方法的优点是，用户可每次敲入不同的口令，因此不需要口令保密，只有口令发生器需要安全保护。为了增加安全性， Linux系统甚至不需联机保存关键词，实际的关键词可保存在有线连接于系统的一个特殊加密计算机中。在用户登录期间，加

密计算机将为用户产生随机数和加密口令。这样一种系统的优点是，口令实际不由用户输入，系统中也不保存关键词，即使是加密格式的关键词也可保存于系统中。其不足之处类似于标记识别方法，每个用户必须携带口令发生器，如果要脱机保存关键词，还需要有一个特殊硬件。

- 个人特征方法：有些识别系统检测如指印、签名、声音、零售图案这样的物理特征。大多数这样的系统是实验性的、昂贵的，并且不是百分之百地可靠。任何一个把数据送到远程系统去核实的系统都有被搭线窃听的危险，非法入侵者只须记录下送去系统校核的信息，以后再重显示这些信息，就能窃密。注意，这同样也是标记识别系统的一个问题。

25.6 SUN OS系统的网络安全

美国SUN Microsystem公司的SUN OS操作系统是建立在贝尔实验室的Linux System V和加州大学伯克得分校的Linux 4.3基础上的Linux操作系统。SUN OS 4.0版提供了专门的鉴别系统，该系统极大地提高了网络的安全性。它也可用来确保其他 Linux系统或非Linux系统的安全。它使用DES密码机构和公共关键字密码机构来鉴别在网络中的用户和机器。 DES表示数据编码标准，而公共数据编码机构是包含两种密钥的密码系统，一种是公用的，另一种是专用的。公用的密钥是公开的而专用密钥是不公开的。专用的密钥用来对数据进行编码和解码。

SUN OS系统与其他公共关键字编码系统的不同之处为， SUN OS的公用和专用密钥都被用来生成一个通用密钥，该密钥又用来产生DES密钥。

25.6.1 确保NFS的安全

在网络文件系统NFS上建立安全系统，首先文件系统必须开放并保证装配的安全。

- 编辑/etc/exports文件，并将-Secure任选项加在要使用DES编码机构的文件系统上。在屏幕上显示服务器怎样开放安全的/home目录，如：

```
home -Secure，access=engineering
```

其中engineering是网络中唯一能存取/home文件系统的用户组。

- 对于每台客户机，编辑/etc/fastab文件时， -Secure将作为一个装配任选项出现在每个需要确保安全的文件系统中。
- SUN OS中包括有/etc/publickey数据库，该库对每个用户均包含有三个域：用户的网络名、公用密钥和编码后的密钥。当正常安装时， X唯一的用户是nobody，这个用户可以无需管理员的干预即可建立自己的专用密钥（使用 chkey（1））。为了进一步确保安全，管理员可为每个使用newkey（8）的用户建立一个公用密钥。
- 确认keyserv（8c）进程由/etc/rc.local启动，并且仍在运行。该进程执行对公用密码的编码，并将编码后的专用密钥存入/etc/keystore中。
- 此时，所有的用户（除超级用户）都必须使用 yppasswd来代替passwd，以使得登录的口令与用户的密钥一致。其结果是在网络中每台客户机的 /etc/passwd文件中不能有每个用户的用户名，因而应使用有缺省值的/etc/passwd文件。
- 当安装、移动或升级某台机器时，要将/etc/keystore和/etc/.rootkey两个文件保留。

注意 当你使用login、rlogin或telnet命令到远程机器时，你会被要求输入口令。一旦你输入正确的口令，也就泄漏了你的帐号。因为此时你的密钥存放在 /etc/keystore中，当然这是指用户对远程机器的安全不信任时。如果用户觉得远程机器在安全保密方面不

可靠，那就不要登录到远程机器去，而可使用NFS来装配你所查找的文件。

25.6.2 NFS安全性方面的缺陷

SUN的远程过程调用（RPC）机制已被证明可以用来建立有效的网络服务，最有名的服务是NFS，它实现了不同机器，不同操作系统之间透明的文件共享。但 NFS并非毫无缺陷。通常NFS鉴别一个写文件的请求时是鉴别发出这个请求的机器，而不鉴别用户。因而，在基于 NFS的文件系统中，运行su命令而成为某个文件的拥有者并不是一件困难的事情。同样， rlogin命令使用的是与NFS同样的鉴别机制，在安全性方面也存在与NFS一样的弱点。

对网络安全问题，一个通常的办法是针对每一个具体的应用来进行解决。而更好的办法是在RPC层设置鉴别机构，使对所有的基于RPC的应用都使用标准的鉴别机构（比如NFS和Yellow pages）。于是在SUN OS系统中就可以对用户的机器都进行鉴别。这样做的优点是使计算机网络系统更像过去的分时系统。在每台机器上的用户都可登录到任何一台机器。就像分时系统中任何一个终端上的用户都可登录到主机系统一样，用户的登录口令就是网络的安全保证。用户不需要有任何有关鉴别系统的基础。SUN系统的目标是让网络系统成为既安全又方便的分时系统。

使用SUN系统要注意以下几点：

- 任何人只要拥有root存取权并具备较好的网络程序设计知识，就可以向网络中加入二进制数据或从网络中获得数据。
- 在采用以太网结构的局域网的工作中，不可能发生信息包被篡改（即被传送的信息包在到达目的站之前，被捕获并将其修改后按原路径发出）。但在网关上发生包被篡改则是有可能的。因此，应确保网络中所有网关都是可靠的。
- 对网络系统最危险的攻击是同向网络中加入数据有关的事件，例如通过生成一个合法的信息包来冒充某个用户;或记录下用户会话的内容，并在晚一些时候再回答它们。这些都会严重地影响数据的完整性。
- 至于偷看信息这类侵袭（仅仅是偷看网络中传送的内容而不冒充任何人）将可能造成失密，但并不十分危险，因为数据的完整性没有被破坏，而且用户可通过对需要保密的数据进行编码来保护数据的安全。

总之，在任何意义上要完全明白网络传送的各种问题并不是很容易的，需不断实践分析。

25.6.3 远程过程调用鉴别

远程过程调用（RPC）是网络安全的核心，要明白这一点就必须清楚在 RPC中鉴别机制是怎样工作的。RPC的鉴别机制是端口开放式的，即各种鉴别系统都可插入其中并与之共存。当前SUN OS有两个鉴别系统：Linux和DES，前者是老的，功能也弱；后者是在本节要介绍的新系统。对于RPC鉴别机制有两个词是很重要的：证书和核对器（credential和verify）。这好比身份证一样，证书记录一个人的姓名、地址、出生日期等，而核对器就是身份证的照片，通过这张照片就能对持有者进行核对。在RPC机制中也是这样：客户进程在RPC请求时要发出证书和核对器信息。而服务器收到后只返回核对器信息，因为客户已知证书内容。

25.6.4 Linux鉴别机制

SUN早期的各种网络服务都建立在Linux鉴别机制之上，证书部分包含站名、用户号、组号和同组存取序列，而核对器是空白的。这个系统存在两个问题：首先，最突出的问题是核对

器为空，这就使得伪造一份证书是非常容易的。如果网络中所有的系统管理员都是可以信赖的，那不会有什么问题。但是在许多网络（特别是在大学）中，这样是不安全的。而 NFS对通过查寻发出mount请求的工作站的Internet地址作为hostname域的核对器来弥补 Linux鉴别系统的不足，并且使它只按受来自特权 Internet口的请求。但这样来确保系统安全仍然是不够的，因为NFS仍然无法识别用户号ID。

另一个问题是 Linux鉴别系统只适用于 Linux系统，但需要在一个网络中所有的站都使用Linux系统是不现实的。因为 NFS可运行于MS-DOS和VMS系统的机器上，但在这些操作系统中Linux鉴别系统是不能运行的，例如：MS-DOS系统甚至就没有用户号的概念。

由此可知，鉴别系统应具有独立于操作系统的证书并使用核对器，例如 DES鉴别系统就是这样。

25.6.5 DES鉴别系统

DES鉴别系统的安全性建立在发送者对当前时间的编码能力上，它使接收者能解码并对照自己的时钟来进行检验，时钟标记也使用 DES编码。这样的机制要工作必须具备以下两个条件：

- 发送者和接收者双方必须对什么是当前时间进行约定。
- 发送者和接收者必须使用同样的编码关键字。

如果网络有时间同步机制，那么客户机服务器之间的时间同步将自己执行。如果没有这样的机制，时间标记将按服务器的时间来计算。为计算时间，客户机在开始 RPC调用之前必须向服务器询问时间，然后计算自己和服务器之间的时间差，当计算时间标记时，这个差值将校正客户方面的时钟。一旦客户机和服务器时钟不同步，服务器就开始拒绝客户机的请求，并且DES鉴别系统将使它们的时间同步。

客户和服务器是怎样来获得相同的编码关键字的呢？当客户希望与服务器交谈时，它生成一个随机关键字来对时间标记进行编码;这个关键字称为会话关键字CK，客户对CK按公用关键字模式进行编码，并在第一次会话时发送给服务器。这个 CK是唯一使用公用关键字编码的关键字。这时只有这一客户与服务器两者才知道它们的 DES关键字，这个关键字称为共有关键字。

第一次请求时，客户的证书包括三项：名字、用共有关键字编码的会话关键字和用会话关键字编码的时窗，时窗告诉服务器：以后即将给你发送许多证书 ;也许会有人用伪造的时间标记冒充新的会话向你发送证书。当你收到时间标志时，请查看你的当前时间是否在时间标记和时间标记加时窗之间，如果不对请拒绝。

为创建安全的NFS文件，时窗缺省值为30分钟。在发出首次请求时，客户的核对器中包含被编码的时间标记和特定时窗（WIN+1）的编码核对器。这样做的原因是：如果某人想写一个程序并且在证书和核对器的编码域中填充一些任意的二进制值，服务器将 CK解码成DES关键字，并且用它来对时窗和时间标记解码，最后产生随机值。在经过上千次的努力后，这些随机的时窗/时间标记对才有可能通过鉴别系统，因此，时窗核对器将增加了猜测出正确的证书的难度，提高了安全性。

在对客户进行鉴别后，服务器将在证书表中存放四项值：客户名A、会话关键字CK、时窗、时间标记。在服务器中保留前三项的目的是以备将来使用。保留时间标记的目的是为防止再次执行，服务器只接收比以前的时间标记晚的时间标记。服务器将向客户返回的核对器包括一个序号ID和负的时间标记（该标记是被CK编码后的）。客户机知道，只有服务器能返送回这样的

核对器，因为只有服务器知道时间标记。

第一次会话过程是很复杂的，以后就容易多了，客户每次向服务器发送它的 ID和编码后的时间标记，而服务器则返送回编码后的时间标记。

25.6.6 公共关键字的编码

SUN OS使用Diffie-Hellman法进行公共关键字的编码，该算法随机产生一个秘密关键字（SK），简称密钥。可用一个公式来计算出公共关键字（ PK），公共关键字存放在公共目录中，而密钥存放在专用的目录中。由PK和SK生成普通关键字K，由于计算K必须知道两个密钥中的一个，所以除了服务器和客户外没有任何人能计算 K。计算将与另一个已知常数M求模。尽管某人的密钥可能会被人采用对公共关键字求对数的方法来得到，但是由于 M的值很大，要计算出M来几乎是不可能的。为了确保安全， K必须用较多位的二进制数作为DES密钥，最多可从K中取56位来形成DES密钥。

PK和SK都是以在文件publickey、byname中的网络名的顺序存放， SK用登录号时的口令编码后存放。当你登录到一个站时， Login程序先取你的编码关键字后再用你的口令对其进行编码，并将解码后的密钥送给确保安全的本地密钥服务器，以备以后进行 RPC处理时使用。

注意 一般的应用是不需要知道公共关键字和密钥的。

除改变登录口令外， yppasswd程序还将随机地产生新的公共关键字和密钥关键字对。密钥服务器是一个驻留于本机的RPC服务器，它执行以下三种公共关键字操作：

- setsecretkey（secretkey）：告诉密钥服务器将密钥 SK存贮起来，以备将来使用（通常是被login程序采用）。
- encrytsessionkey（severname，des_key）：使在第一次RPC处理中将会话关键字传送给服务器，密钥服务器查找severname中的公共关键字，并将它和 setsecretkey设置的client的密钥组合，以生成用于对des_key编码的密钥。
- decrytsessionkey（clientname，des_key）：服务器又请求密钥服务器通过调用本操作来对会话密钥解码。

注意 隐含在这些调用中的使用者名必须鉴别，密钥服务器中可能使用 DES鉴别系统（因为会产生死馈）。密钥服务器通过按uid存贮的密钥来解决这个问题，它只允许对本机的root所属进程的请求。然后client进程又执行setuid 进程，该进程属于root，执行对client的请求，并将真正的client的uid告诉密钥服务器。

以上三种操作都是系统调用，内核将与密钥服务器直接通信，而不是通过执行 setuid程序来通信。

25.6.7 网络实体的命名

原有的Linux鉴别系统对网络实体的命名存在一些问题，对 Linux鉴别系统最基本的网络实体uid，已经陈述了这个系统的一个问题（太 Linux系统化了），而且这个系统还有两个问题：一个是当许多域联系起来时的 uid冲突;另一个是超级用户不是以每个域为基础赋值，而是以每台机器为基础赋值。在缺省情况下， NFS以一种严密的方式解决这一问题：它不允许根通过网络以uid0存取。

DES鉴别系统通过建立在新名字（网络名）基础上的命名机制纠正这些问题。简单地说，

网络名是一串可打印字符，从根本上说，我们所要鉴别的正是这些网络名。公共关键字和密钥按网络名存贮而不是按用户名存贮。yellow page map netid.byname 将网络名映射为本机器中的用户名uid和同组存取序列，而非SUN环境会将网络名映射为其他序列。

我们采用全局唯一的网络名来解决网络命名问题，这比选择全局唯一的用户号要容易得多。在SUN环境中，对每个YP域，用户名是唯一的。如将操作系统名在 YP域中的用户号和 ARPA域名组合在一起就构成了网络名。在为一个域命名时将 ARPA域名加在本地域名之后是一个好习惯。

像对用户赋以网络名一样，对机器也赋以网络名，这样就可解决多个超级用户的问题。机器的网络名的形式与用户的网络名的形式相似，正确的机器鉴别系统对网络中的无盘工作站是非常重要的，它必须保证无盘工作站能通过网络存取本机的 home目录。

非SUN环境中，网络名的产生也许与前述有较大区别，但这并不妨碍它们通过 SUN的网络安全系统合法地存取信息，为鉴别一个来自另一个域的用户，只需在两个 YP数据库建立实体。一个实体是有关密钥和公开密钥的，另一个是有关 uid和同组存取序列的。完成这项工作后，在远程域中的用户就可利用本域的网络服务。

25.6.8 DES鉴别系统的应用

一个应用是广义的YP更新服务，这个服务允许用户更新YP数据库中的专用域。

另一个应用也是最重要的应用是：更安全的网络文件系统 NFS。使用Linux鉴别系统的NFS存在以下三个问题：

- 证书的检验仅仅在装配时进行，这时客户机从服务器获得一条信息，这条信息是以后请求的关键：文件 handle。如果有人不通过服务器就能通过猜想或偷听网络传输内容而获得文件handle，那么他也就能破坏Linux鉴别系统。因为在NFS文件装配完毕后，当发生文件请求时，不再进行证书的检验。
- 假如一个文件系统已从一个为多个客户机服务的服务器中装配到一台客户机中，当一个具有超级用户特权的用户使用 su命令非法存取别人的文件时，文件系统不能提供任何保护。
- NFS的第三个问题是，由于它不能鉴别远程客户机的超级用户，它不得不采用一种严厉的方法：拒绝所有的超组用户存取。

新的鉴别系统解决了所有这些问题。如果某人想获得非法存取权，他不得不猜出正确的被编码后的时间标记并放在证书中，而这几乎是不可能的，这样他就不能猜出文件 handle。由于新的系统可鉴别机器，上述第二、第三问题也解决了。但是在这点上，根文件系统不能使用安全的文件，而非文件系统的根用户由IP地址识别。

实际上，与每个文件系统相联系的安全级别可由系统管理员改变。文件 /etc/exports包含有文件和可装配它们的机器名，在缺省的情况下文件系统向 Linux鉴别系统开放。但管理员在任意行后加上-Secure就可改变为向DES鉴别系统开放。与DES鉴别系统相应的是一个参数：服务器能接收的最大窗口的大小。

25.6.9 遗留的安全问题

尽管使用su不能破坏DES鉴别系统，但仍有几种方法可做到这点。为了通过鉴别，你的密钥必须存放在工作站中，这通常在登录时发生，login程序用你的口令对你的密钥解码，并存放起来以备使用，由于别人不能对你的密钥解码，因而任何人用 su命令冒充你都不可能。编辑

/etc/passwd文件也不可能对他有什么帮助，因为他必须修改存放在YP中的被编码后的密钥。如果你用你的口令登录到别人的工作站中，你的密钥就会存放在该工作站中，他们就能用 su命令冒充你，由于你不可能将你的口令泄露给你不信任的机器，因而这是不可能发生的。但在其他机器上的人可以修改login程序将所有口令存放在他能看到的文件中。

由于使用su命令不能破坏DES鉴别系统，也许最容易的方式就是猜出口令，因此选择安全的口令对用户是至关重要的。另一个最方便的方法就是试图重新执行。因此服务器应放置在安全的地方。

还有其他打破DES的方法，但都非常困难，需要花费巨型计算机几个月的时间来计算。

另一个DES不曾考虑的安全问题，就是网络偷听，即使有了 DES，也不能阻止任何人偷听网络传输的内容。大多数情况下这不是一个大的问题，因为网络中传送的大多数内容虽不是不可读的，但要搞清网络中传送的二进制的含义却不是一件轻松的工作。对登录来说，由于你希望别人不能通过网络获得你的口令，故你传送的是编码后的口令，正如前面所提到的一样，鉴定系统是信息交换的关键，网络传输内容被偷听的问题可以在每个具体应用中获得解决。

25.6.10 性能

众所周知公共关键字系统的速度是很慢的，但在 SUN系统中，公共关键字编码很少发生，它仅仅发生在每个服务的第一次事务处理时，即使如此，还有缓冲区加速编码的进行。当客户机第一次与服务器接触时，客户机和服务器都必须计算出普通密钥，计算普通密钥的时间主要是计算幂关于M的模，在SUN3系统中使用192位模，这需花1秒钟计算普通密钥，也就是说总共需要2秒。因为客户机和服务器都必须计算普通密钥。因此，在客户机与服务器第一次接触时，必须等待这个时间，而且关键字服务器将保存计算的结果，以后就用不着每次都计算幂了。

DES系统最重要的网络服务就是快速安全的NFS、DES鉴别系统，相对于Linux鉴别系统多花的时间就是编码的时间。时间标记和 DES块都是64位，在一次RPC中平均要进行四次编码操作：客户机对请求时间标记编码，服务器对它进行解码，服务器对时间标记编码，客户机对它解码。在SUN3系列中对一个块进行编码的硬件执行需 1毫秒，软件执行需1.2毫秒。这样进行一次RPC调用，若由硬件执行需多花2毫秒，若由软件执行需多花5毫秒。进行一次NFS请求大约需20秒，这样由DES鉴别会使NFS请求的性能降低10%（假如有编码硬件），25%（假如没有编码硬件）。这就是DES对网络性能的冲击，事实上并不是所有的文件操作都需通过网络，因而DES对系统性能的影响要低得多。另外是否采用 DES鉴别系统是任选的，因此，在需要高速的环境下可以不采用DES鉴别系统。

25.6.11 启动和setuid程序引起的问题

考虑这样的情况：计算机因发生某种事件后重新启动。这时机内保存的所有密钥都被清除，如果采用的是DES鉴别系统，那么所有的进程都不能再利用网络服务。这时起关键作用的是根进程。如果根的密钥保存在机内同时没有人输入口令，对该密钥进行编码，那么根进程就将能够利用网络服务。对以上问题的解决就是将根的密钥存放在关键字服务器可读的某个文件中。这样的方式对有盘工作站来说是很好的，但对无盘工作站来说，即存在一个致命的问题：它的密钥必须通过网络存取。这样在无盘工作站启动时，如有人窃听网络传送内容，他就能发现编码后的密钥，尽管完成，但这一工作并不容易。

众所周知有一种启动方式叫单用户启动，启动后根的登录外壳出现在主终端上，这儿出现

的问题是，如果安装了C2安全系统，从单用户启动仍需口令；当没有安装C2安全系统时，只要/etc/ttytab文件中的console项标记为secure，机器的启动就不需口令。

另一个问题是无盘工作站启动不安全，因为有人可以冒充启动服务器，启动一个不正当的内核记录远程无盘工作站的密钥，因为仅仅在内核和关键字服务器运行之后， SUN系统才能对这一问题提供保护。在此以前没有任何方式可以鉴别回答是否来自正确的启动服务器。但我们不考虑这种情况，因为一个不知道源码的人，要想写这样的内核几乎是不可能的。另外，犯罪者也极易留下证据，只要你对网络中的启动服务器进行检测，就能发现谁是服务器。

并不是所有的setuid程序都会按我们希望的那样运行，比如一个由用户 dave拥有的setuid程序，只要在机器启动后， dave没有进行登录，那么程序 setuid就不能存取安全的网络服务（即采用DES鉴别系统的网络服务），好在绝大多数setuid程序都为root所拥有，而且，根的密钥在系统忘却后总是存放在系统中，因而程序 setuid在采用了DES系统之后，仍能像原来那样运行。

25.6.12 小结

SUN的目标是要让网络系统像分时系统一样安全，这个目标已经达到。在分时系统中，用户被口令鉴别，在DES鉴别系统中，网络中的用户也由口令鉴别。在分时系统中，用户信任系统管理员，他的职业道德不允许他改变用户的口令以冒充该用户。在 SUN系统中有用户，信息网络管理员，他不会改变用户在公共密钥数据库中的实体。 SUN的系统从某种意义上说比分时系统更安全，因为在SUN的系统中旋转“窃听”装置来“窃听”网络中传送的口令和编码用的密钥是无用的（因为这些口令和密钥都已被编码）。而大多数分时系统对来自终端的数据并不进行编码，用户必须相信，没有人在终端与主机的传送线上安装“窃听”装置。

DES鉴别系统也许不是最终完善的鉴别系统，在将来，很可能有更好的算法和硬件来证明DES鉴别系统无用并放弃它，但至少可以说 DES为将来的发展指出了一个方向。从理论上讲，协议规定会话密钥甚至公共密钥的编码要采用Diff3-Hellman方法。为了使DES鉴别系统更有力，我们要做的仅仅是使会话密钥的编码更有力，从理论上说这样会形成另一个协议，但是 RPC的优点在于它可以采用任何鉴别系统而本身不会受到影响。

至少在目前我们可以说DES鉴别系统满足了我们对网络服务的安全要求，在一个不友好的网络系统中建立起了一个足够安全的系统，而所付出的代价也不高。用户不需使用磁卡或记住上百位的数字，用户像往常一样使用口令让系统鉴别自己，只是系统的性能略有降低。但是，如果用户认为不能使性能降低，并且他的网络系统非常友好的话，他可以不采用DES鉴别系统。

第26章　Linux系统的用户安全性

本章从用户角度讨论Linux系统安全，阐述口令、文件保护、目录保护、与用户程序有关的某些特殊特性和使用crypt命令加密，并给出一些重要的安全忠告，以帮助用户保护自己的帐户安全。

26.1　口令安全

Linux系统中的/etc/passwd文件含有全部系统需要知道的关于每个用户的信息（加密后的口令也可能存于/etc/shadow文件中）。/etc/passwd中包含用户的登录名、经过加密的口令、用户号、用户组号、用户注释、用户主目录和用户所用的外壳程序。其中用户号（ UID）和用户组号（GID）用于Linux系统唯一地标识用户和同组用户及用户的访问权限。 /etc/passwd中存放的加密的口令用于与用户登录时输入的口令经计算后相比较，符合则允许登录，否则拒绝用户登录。用户可用passwd命令修改自己的口令，不能直接修改/etc/passwd中的口令部分。

一个好的口令应当至少有6个字符长，不要取用个人信息（如生日、名字、反向拼写的登录名、房间中可见的东西），普通的英语单词也不好（因为可用字典攻击法），口令中最好有一些非字母（如数字、标点符号、控制字符等），还要好记一些，不能写在纸上或计算机中的文件中，选择口令的一个好方法是将两个不相关的词用一个数字或控制字符相连，并截断为 8个字符。当然，如果你能记住8位乱码自然更好。

不应使用同一个口令在不同机器中使用，特别是在不同级别的用户上使用同一口令，会引起全盘崩溃。用户应定期改变口令，至少 6个月要改变一次，系统管理员可以强制用户定期做口令修改。为防止他人窃取口令，在输入口令时应确保无人在身边。

26.2　文件许可权

文件属性决定了文件的被访问权限，即谁能存取或执行该文件。用 ls -l可以列出详细的文件信息，如：

-rwxrwxrwx 1 pat cs440 70 Jul 28 21：12 zombin

包括了文件许可、文件联结数、文件所有者名、文件相关组名、文件长度、上次存取日期和文件名。

其中文件许可分为四部分：

-：表示文件类型。

第一个rwx：表示文件属主的访问权限。

第二个rwx：表示文件同组用户的访问权限。

第三个rwx：表示其他用户的访问权限。

若某种许可被限制则相应的字母换为-。

在许可权限的执行许可位置上，可能是其他字母， s、S、t、T。s和S可出现在所有者和同组用户许可模式位置上，与特殊的许可有关，后面将要讨论， t和T可出现在其他用户的许可模式位置上，与“ 粘贴位 ”有关而与安全无关。小写字母（ x，s，t）表示执行许可为允许，负号或大写字母（-，S或T）表示执行许可为不允许。改变许可方式可使用 chmod命令，并以新

许可方式和该文件名为参数。新许可方式以3位8进制数给出，r为4，w为2，x为1。如rwxr-xr--为754。chmod也有其他方式的参数可直接对某组参数修改，在此不再多说，详见 Linux系统的联机帮助。

文件许可权可用于防止偶然性地重写或删除一个重要文件（即使是属主自己）！

改变文件的属主和组名可用chown和chgrp，但修改后原属主和组员就无法修改回来了。

26.3 目录许可

在Linux系统中，目录也是一个文件，用 ls -l列出时，目录文件的属性前面带一个 d，目录许可也类似于文件许可，用ls列目录要有读许可，在目录中增删文件要有写许可，进入目录或将该目录作路径分量时要有执行许可，故要使用任一个文件，必须有该文件及找到该文件的路径上所有目录分量的相应许可。仅当要打开一个文件时，文件的许可才开始起作用，而 rm、mv只要有目录的搜索和写许可，不需文件的许可，这一点应注意。

26.4 umask命令

umask设置用户文件和目录的文件创建缺省屏蔽值，若将此命令放入 .profile文件，就可控制该用户后续所建文件的存取许可。 umask命令与chmod命令的作用正好相反，它告诉系统在创建文件时不给予什么存取许可。

26.5 设置用户ID和同组用户ID许可

用户ID许可（SUID）设置和同组用户ID许可（SGID）可给予可执行的目标文件（只有可执行文件才有意义）当一个进程执行时就被赋于 4个编号，以标识该进程隶属于谁，分别为实际和有效的UID，实际和有效的GID。有效的UID和GID一般和实际的UID和GID相同，有效的UID和GID用于系统确定该进程对于文件的存取许可。而设置可执行文件的 SUID许可将改变上述情况，当设置了SUID时，进程的有效UID为该可执行文件的所有者的有效UID，而不是执行该程序的用户的有效UID，因此，由该程序创建的都有与该程序所有者相同的存取许可。这样，程序的所有者将可通过程序的控制在有限的范围内向用户发表不允许被公众访问的信息。同样，SGID是设置有效GID。

用chmod u+s 文件名和chmod u-s文件名来设置和取消SUID设置。用chmod g+s 文件名和chmod g-s文件名来设置和取消 SGID设置。当文件设置了 SUID和SGID后，chown和chgrp命令将全部取消这些许可。

26.6 cp mv ln和cpio命令

cp拷贝文件时，若目的文件不存在，则将同时拷贝源文件的存取许可，包括 SUID和SGID许可。新拷贝的文件属拷贝的用户所有，故拷贝他人的文件时应小心，不要被其他用户的SUID程序破坏自己的文件安全。mv移文件时，新移的文件存取许可与原文件相同， mv仅改变文件名。只要用户有目录的写和搜索许可，就可移走该目录中某人的 SUID程序且不改变其存取许可。若目录许可设置不正确，则用户的 SUID程序可被移到一个他不能修改和删除的目录中，将出现安全漏洞。

ln为现有文件建立一个链，即建立一个引用同一文件的新名字。如目的文件已经存在，则该文件被删除而代之以新的链，或存在的目的文件不允许用户写它，则请求用户确认是否删除

该文件，只允许在同一文件系统内建链。若要删除一个 SUID文件，就要确认文件的链接数，只有一个链才能确保该文件被删除。若 SUID文件已有多个链，一种方法是改变其存取许可方式，将同时修改所有链的存取许可；另一种方法以 chmod 000 文件名，不仅取消了文件的SUID和SGID许可，而且也取消了文件的全部链。要想找到谁与自己的 SUID程序建立了链，不要立刻删除该程序，系统管理员可用ncheck命令找到该程序的其他链。

cpio命令用于将目录结构拷贝到一个普通文件中，然后可再用 cpio命令将该普通文件转成目录结构。用-i选项时，cpio从标准输入设备读文件和目录名表，并将其内容按档案格式拷贝到标准输出设备，使用-o选项时，cpio从标准输入设备读取事先已建好的档案，重建目录结构。cpio命令常用以下命令做一完整的目录系统档案：

```
find fromdir -print|cpio -o > archive
```

根据档案文件重建一个目录结构命令为：

```
cpio -id < archive
```

cpio的安全约定如下：

1) 档案文件存放每个文件的信息，包括文件所有者、小组用户、最后修改时间、最后存取时间、文件存取许可方式。

- 根据档案建立的文件保持存放于档案中的存取许可方式。
- 从档案中提取的每个文件的所有者和小组用户设置给运行 cpio -i命令的用户，而不是设置给档案中指出的所有者和小组用户。
- 当运行cpio -i命令的用户是root时，被建立的文件的所有者和小组用户是档案文件所指出的。
- 档案中的SUID/SGID文件被重建时，保持SUID和SGID许可，如果重建文件的用户不是root，SUID/SGID许可是档案文件指出的用户/小组的许可。

2) 现存文件与cpio档案中的文件同名时，若现存文件比档案中的文件更新，这些文件将不被重写。

3) 如果用修改选项U，则同名的现存的文件将被重写。可能会发生一件很奇怪的事：如被重写的文件原与另一个文件建了链，文件被重写后链并不断开，换言之，该文件的链将保持，因此，该文件的所有链实际指向从档案中提取出来的文件，运行 cpio无条件地重写现存文件以及改变链的指向。

4) cpio档案中可包含的全路径名或父目录名给出的文件。

26.7 su和newgrp命令

26.7.1 su命令

可不必注销帐户而将另一用户又登录进入系统，作为另一用户工作。它将启动一新的外壳并将有效和实际的UID和GID设置给另一用户。因此必须严格将root口令保密。

26.7.2 newgrp命令

与su相似，用于修改当前所处的组名。

26.8 文件加密

crypt命令可提供给用户以加密文件，使用一个关键词将标准输入的信息编码为不可读的杂

乱字符串，送到标准输出设备。再次使用此命令，用同一关键词作用于加密后的文件，可恢复文件内容。

一般来说，在文件加密后，应删除原始文件，只留下加密后的版本，且不能忘记加密关键词。

在vi中一般都有加密功能，用vi -x命令可编辑加密后的文件。加密关键词的选取规则与口令的选取规则相同。

由于crypt程序可能被做成特洛伊木马，故不宜用口令做为关键词。最好在加密前用 pack或compress命令对文件进行压缩后再加密。

26.9 其他安全问题

26.9.1 用户的.profile文件

由于用户的HOME目录下的.profile文件在用户登录时就被执行。若该文件对其他人是可写的 则系统的任何用户都能修改此文件，使其按自己的要求工作。这样可能使得其他用户具有该用户相同的权限。

26.9.2 ls -a

此命令用于列出当前目录中的全部文件，包括文件名以“ .”开头的文件，查看所有文件的存取许可方式和文件所有者，任何不属于自己的但存在于自己的目录中的文件都应怀疑和追究。

26.9.3 . exrc文件

为编辑程序的初始化文件，使用编辑文件后，首先查找 $HOME/.exrc文件和./.exrc文件，若该文件是在$HOME目录中找到，则可像.profile一样控制它的存取方式，若在一个自己不能控制的目录中运行编辑程序，则可能运行其他人的.exrc文件，或许该.exrc文件存在那里正是为了损害他人的文件安全。为了保证所编辑文件的安全，最好不要在不属于自己或其他人可写的目录中运行任何编辑程序。

26.9.4 暂存文件和目录

在Linux系统中暂存目录为/tmp和/usr/tmp，对于程序员和许多系统命令都使用它们，如果用这些目录存放暂存文件，别的用户可能会破坏这些文件。使用暂存文件最好将文件屏蔽值定义为007，但最保险的方法是建立自己的暂存文件和目录： $HOME/tmp，不要将重要文件存放于公共的暂存目录。

26.9.5 UUCP和其他网络

UUCP命令用于将文件从一个Linux系统传送到另一个Linux系统，通过UUCP传送的文件通常存于/usr/spool/uucppublic/login目录，login是用户的登录名，该目录存取许可为777，通过网络传输并存放于此目录的文件属于UUCP所有，文件存取许可为666和777，用户应当将通过UUCP传送的文件加密，并尽快移到自己的目录中。其他网络将文件传送到用户 HOME目录下的rjc目录中。该目录应对其他人是可写可搜索的，但不必是可读的，因而用户的 rjc目录的存

取许可方式应为733，允许程序在其中建立文件。同样，传送的文件也应加密并尽快移到自己的目录中。

26.9.6 特洛伊木马

在Linux系统安全中，用特洛伊木马来代表一种程序，这种程序在完成某种具有明显意图的功能时，还破坏用户的安全。如果 PATH设置为先搜索系统目录，则受特洛伊木马的攻击会大大减少。如模拟的crypt程序。

26.9.7 诱骗

类似于特洛伊木马，模拟一些东西使用户泄漏一些信息，不同的是，它由某人执行，等待无警觉的用户来上当。如模拟的login。

26.9.8 计算机病毒

计算机病毒是通过把其他程序变成病毒从而传染系统的，可以迅速地扩散，特别是系统管理员的粗心大意，作为 root运行一个被感染的程序时。实验表明，一个病毒可在一个小时内（平均少于30分钟）取得root权限。

26.9.9 要离开自己已登录的终端

除非能对终端上锁，否则一定要注销帐户。

26.9.10 智能终端

由于智能终端有send和enter换码序列，告诉终端把当前行送给系统，就像是用户敲入的一样。这是一种危险的能力，其他人可用write命令发送信息给本用户终端，如果信息中含有以下的换码序列：

- 移光标到新行（换行）。
- 在屏幕上显示“ rm -r * ”。
- 将该行送给系统。

其结果大家可以想象。

禁止其他用户发送信息的方法是使用mesg命令，mesg n不允许其他用户发信息，mesg y允许其他用户发信息。即使如此仍是有换码序列的问题存在，任何一个用户用 mail命令发送同样一组换码序列，不同的要用 !rm -r *替换rm -r *。mail将以!开头的行解释为一条外壳命令，启动外壳，由外壳解释该行的其他部分，这被称为外壳换码。为避免 mail命令发送换码序列到自己的终端，可建立一个过滤程序，在读mail文件之前先运行过滤程序，对mail文件进行处理：

```
myname="$LOGNAME";
tr -d[\001-\007][-\013-\037]</usr/mail/$myname>>$HOME/mailbox;
> /usr/mail/$myname;
mail -f $HOME/mailbox
```

其中tr将标准输入的字符转换手写到标准输出中。这只是一个简单的思路，从原则上来说，此程序应为一个C程序，以避免破坏正发送到的文件，可用锁文件方式实现。

26.9.11 断开与系统的连接

用户应在看到系统确认登录注销后再离开，以免在用户未注销时由他人潜入。

26.9.12 cu命令

该命令使用户能从一个Linux系统登录到另一个Linux系统，此时，在远地系统中注销用户后还必须输入“ ~ ”后回车，以断开cu和远地系统的连接。

cu还有两个安全问题：

- 如果本机安全性弱于远地机，不提倡用 cu去登录远地机，以免由于本地机的不安全而影响较安全的远地机。
- 由于cu的老版本处理“ ~ ”的方法不完善，从安全性强的系统调用安全性弱的系统时，会使弱系统的用户使用强系统用户的 cu传送强系统的/etc/passwd文件，除非确定正在使用的cu是正确版本，否则不要调用弱系统。

26.10 保持帐户安全的要点

1）保持口令的安全。

- 不要将口令写下来。
- 不要将口令存于终端功能键或调制解调器的字符串存储器中。
- 不要选取显而易见的信息作口令。
- 不要让别人知道。
- 不要交替使用两个口令。
- 不要在不同系统上使用同一个口令。
- 不要让人看见自己在输入口令。

2）不要让自己的文件或目录可被他人写。

- 如果不信任本组用户，umask设置为022。
- 确保自己的.profile除自己外对他人都不可读写。
- 暂存目录最好不用于存放重要文件。
- 确保HOME目录对任何人不可写。
- uucp传输的文件应加密，并尽快私人化。

3) 若不想要其他用户读自己的文件或目录，就要使自己的文件和目录不允许任何人读。

- umask设置为006/007。
- 若不允许同组用户存取自己的文件和目录，umask设置为077。
- 暂存文件按当前umask设置，存放重要数据到暂存文件的程序，就被写成能确保暂存文件对其他用户不可读。
- 确保HOME目录对每个用户不可读。

4) 不要写SUID/SGID程序。

5) 小心地拷贝和移文件。

- cp拷贝文件时，记住目的文件的许可方式将和文件相同，包括 SUID/SGID许可在内，如目的文件已存在，则目的文件的存取许可和所有者均不变。
- mv移文件时，记住目的文件的许可方式将和文件相同，包括 SUID/SGID许可在内，若在同一个文件系统内移文件，目的文件的所有者和小组都不变，否则，目的文件的所有者和小组将设置成本用户的有效UID和GID。
- 小心使用cpio命令，它能复盖不在本用户当前目录结构中的文件，可用 t选项首先列出要被拷贝的文件。

6) 删除一个SUID/SGID程序时，先检查该程序的链接数，如有多个链，则将存取许可方式改为000，然后再删除该程序，或先写空该程序再删除，也可将该程序的 i结点号给系统管理员去查找其他链。

7) 用crypt加密不愿让任何用户（包括超级用户）看的文件。

• 不要将关键词做为命令变量。

• 用ed -x或vi -x编辑加密文件。

8) 除了信任的用户外，不要运行其他用户的程序。

9) 在自己的PATH中，将系统目录放在前面。

10）不要离开自己登录的终端。

11) 若有智能终端，当心来自其他用户，包括 write命令、mail命令和其他用户文件的信息中有换码序列。

12) 用Ctrl+D或Exit退出后，在断开与系统的联接前等待看到 login：提示。

13) 注意cu版本。

• 不要用cu调用安全性更强的系统。

• 除非确信cu不会被诱骗去发送文件，否则不要用 cu调用安全性更弱的系统。

第六篇　X Window系统的内部结构和使用

X Window 系统是在UNIX类的操作系统中应用最为广泛的基于窗口的用户图形界面。它使用方便，界面直观，并且和具体的计算机的硬件无关。同时它支持分布式的网络操作。所以，基于X Window 的应用程序一直在UNIX类的操作系统中占有主导的地位。 Linux出现以后， X Window 系统也有了在Linux系统上的实现。这就是XFree86系统。XFree86 比标准的MIT版的X Window 支持更多的硬件，这样，它的应用就更加的广泛。关于 XFree 86的安装和启动，请参考附录B。

麻省理工学院发布的 X Window系统以及其源代码叫做 X11系统。 Linux系统上使用的XFree86.3.1.2就是基于X11R6版本的。其他UNIX系统中的X Window应用程序基本上可以在Linux 的XFree 86系统上直接使用。而在Linux系统中编写基于XFree 86 的应用程序的需求也越来越多。所以在本篇，我们讨论一下Linux系统中X Window窗口系统的内部结构和具体的设置和使用。有了这些知识，你就可以尝试着在 XFree 86系统上编写一些使用的程序。等到积累了一定的经验以后，就可以编写一些大型的基于 X Window系统的程序了。下面，我们先从 X Window系统的最基本的概念讲起。

第27章　X Window系统的基本知识

27.1　X Window系统介绍

X Window 系统是一套在各种图形显示器上均可使用的窗口系统。它是由麻省理工学院(MIT)开发出来的。 X Windows系统(以下简称为X) 可在许多系统上执行。由于它和生产厂商无关、具可移植性、对彩色掌握的多样性及对网络之间的操作透明性，使得 X成为一个工业的标准。由于原始程序代码可自由使用，所以它也是一个优秀的研究媒介。

程序员可以利用X开发可移植性图形用户界面。 X最重要的特征之一是它独特的与设备无关结构。任何硬件只要提供X协议，便可以执行应用程序显示一系列包含图文的窗口，而不需要重新编译和链接。这种与设备无关的特性，使得只要是根据 X标准所开发的应用程序，均可在不同的环境下(如大型电脑、工作站、个人电脑上)执行，因而奠定了X成为工业标准的地位。

X可以在一些 UNIX 系统的电脑上执行，如 Alliant、Apollo、DEC、IBM、Hewlett-Packard、Sun等，也可在 DEC的 VAX/VMS、MS-DOS及一些其他的系统上执行。其他的一些厂商如 AT&T、Adobe、Control Data、Data General、Fujitsu、Prime、Siemens、Silicon Graphics、 Sony、Texas Instruments、Wang、Xerox 均曾表示支持X。

27.1.1　X的特点

X的特点及其受大众欢迎的原因如下：

• X具有网络透明性：通过网络，应用程序在其他计算机上输出显示，就和在自己计算机

上输出一样容易。此种通信结构和网络上另一端的计算机使用的语言完全无关，也和计算机硬件无关，甚至不需使用相同的操作系统。总而言之，程序可以在另一种不同类型显示器下执行，而不需要重新编译和重新链接。

- 可支持许多不同样式的用户界面。管理窗口的功能 (例如窗口的摆放、大小及显示顺序等等)并不包含于系统中，而是由应用程序来控制，因此可轻易地更换。不同样式的界面和不同的应用程序有关，例如滚动窗口中的文字和选择窗口中的一个物体，彼此间不会互相限制。
- X不是电脑操作系统的一部分：对操作系统而言， X只是一个应用程序而已，因此， X很容易在不同的系统上安装。
- 窗口是层次式的： 应用程序可以直接利用窗口系统已有的工具便可满足大部分的需求，而不需要借助于其他的输入或控制结构。 (例如：可利用一个分支的子窗口来产生菜单。)

27.1.2 什么是窗口系统

本节讨论一般窗口系统的一些基本概念， X可以当作其中一个特例，如果你已熟悉其他的窗口系统，本节内容仅需快速浏览即可。

X是一个在图形显示屏幕上建立和管理窗口的系统，它可以在拥有图形显示器和键盘的工作站或其他型号拥有图形显示器的终端机上执行。 X把指示位置的设备叫做指针，这个设备通常为鼠标。X支持现今电脑上常见的窗口用户界面。

使用窗口系统的情形与在普通办公桌上的工作相似。你的办公桌上通常放了一些纸、邮件和手边相关的工作、一些有用的工具 (如时钟，日历，计算器等)。当进行到工作的另一个部分时，你会重新安排桌面上的纸，你可能把工具集中放在一起，也可能不时参考桌上仍然看得到的纸的内容，过了一阵子，你可能把其中的一些资料暂时摆到一边，或者通通从桌面上移走。

上述是一个人的工作模式，如果电脑能提供这样的功能是很理想的。不幸的是，老式的终端机或显示器使得你一次只能在屏幕上做一件工作，而且只能看见一小部分的文字资料 (通常为24行)，图形的工作就更别提了。现在窗口系统正克服这点，通常它提供一个较大的屏幕，允许你同时看到几件工作项目，可以显示图形，甚至有彩色。

X便是依照上述窗口的工作模式开发出来的。 在X的环境下，一个窗口是屏幕上的一块长方形区域，且平行于屏幕的边，通常，每一个窗口被一个独立的应用程序所专用，数个应用程序可以“同时”在它自己所拥有的窗口上显示输出结果， X允许窗口重叠。

但即使窗口的一部分或全部被其他窗口遮盖，应用程序仍然可以对它自己所拥有的窗口输出信息。设备程序提供在屏幕上移动窗口、改变窗口大小、把窗口放在最上一层或最下一层等功能。即便是窗口可以重叠，但在同一屏幕开了许多窗口仍然非常费时。 因此和其他的窗口系统一样， X提供图标功能。我们在屏幕上用一个图标代表一个应用窗口，当我们对应用窗口图标化后，窗口以图标代之，从而空出了较多的屏幕空间；相反的动作为解除图标化，也就是以原先的窗口替换图标。

一些实用的功能，例如时钟或日历，并非系统内部提供的，而是由许多小的应用程序所提供的。

对于输出， X提供了许多在窗口写文字和画图形到的功能选择。许多种字体可供选择，并且提供许多图形的结构和绘图的基本方法，例如提供点、线、弧线、区域的画法。颜色的选择更是丰富。这些复杂的部分对用户而言是隐藏起来的，用户可以简单地使用它们，例如，在使

用时，你可以用“ *times-bold-i* ”表示要使用加倍粗的斜体字体；当你需要使用彩色时，你只要用日常的名称，例如“ yellow(黄色) ”或“ navy blue (天蓝色) ”。

X也提供多样化的输入功能。 X可以使用不同形式的键盘，如传统的 QWERTY键盘或Dvorak Style键盘，或者是不同国家的有特殊规定的键盘。处理用户界面是输入功能很重要的一个部分，键盘和鼠标发出的指令告诉系统如何构造一个窗口和处理窗口中的内容。

由于X的窗口处理功能并非是系统内部提供的，而是建立在用户层次上的，因此容易修改或更换。所以X能提供不同形态的用户界面。换个角度来说，用户界面所必需具有的灵活性几乎完全可由X独立提供。

27.1.3 X发展的历史

X于1984年在美国麻省理工学院 (MIT)电脑科学研究室开始开发。当时 Bob Scheifler 正在开发分布式系统，同一时间 DEC公司的 Jim Gettys 正在麻省理工学院做 Athena 计划的一部分，两个计划都需要一个相同的东西 —— 一套可以在 UNIX 机器上运行的窗口系统。因此他们从斯坦福大学得到了一套叫做W的实验性窗口系统。 因为是根据W窗口系统开始开发的，所以当开发到了足以和原先系统有明显区别时，他们把这个新系统叫做 X。

工作持续地进行，新的版本不断地产生 (当软件和前一版不相容时，新的版本便产生了)。在1985年中期决定了任何人只要付版权费便可使用 X。以下为X的大事记:

- 第10版: 1985年底。从此，在 MIT以外的人和组织，才开始对X有实质的贡献。
- 第一套商业化的X产品: DEC 于 1986 年 1月推出 VAXstation-II/GPX。
- 第10版第 3次发行: 1986年 2月。从此时起， X开始流传于世，人们把它移植到许多新的系统上。
- 第10版第 4次发行: 1986年11月。
- 第一次X技术会议: 1987年 1月于 MIT。
- 在1986年间，第10版X无法满足所有的需求已非常明显， MIT 和 DEC便从事于重新设计整个协议，这就是X第11版。
- 第11版第 1次发行: 1987年 9月。
- X协会成立: MIT X协会成立，目的是为了研究开发及控制标准，目前有 30个以上的机构加入。
- 第二次X技术会议: 1988年 1月。
- 第11版第 2次发行: 1988年 3月。
- 第11版第 3次发行: 1988年10月。

27.1.4 X的产品

严格地说， X窗口系统并不是一个软件，而是一个协议。这个协议定义一个系统产品所必需具备的功能 (就如同 TCP/IP、DECnet 或IBM的 SNA，这些也都是协议，定义软件所应具备的功能)。任何系统能满足此协议及符合X协会其他的规定，便可称为X的产品。

简单地说，本附录从现在起不再区分协议和软件。当我们提到 X，意指一个完整且适当的系统产品。

27.1.5 MIT 发行的X

MIT 所发行的X可以支持许多厂家的电脑，目前的版本 (第11版第 3次发行) 支持以下系统:

• Apple A/UX
• Apollo Domain/IX
• 4.3 + tahoe
• Digital Equipment Coporation Ultrix
• Hewlett Packard HP-UX
• IBM AOS
• Sun Mircosystems SunOS

此外还有更多的商业化产品。

此系统一直处于开发中，而且越来越多的人开始使用它，由第三方厂家开发的软件也逐渐增加，从而使系统版本一分为二：

• core版 —— 软件由MIT X协会提供。
• corelib 版 —— 软件由用户或第三方厂家提供。

为了实用，core和corelib 软件保存在不同的磁带上发行。

1. MIT 版包含了什么

这个版本包含了文件说明、源代码、配置文件、实用程序和其他一些建立完整操作系统所必需的东面，(没有提供任何目标文件或二进制文件，系统必需由源代码编译建立)，在此我们只从用户观点看这个系统，所以只描述那些窗口系统程序本身或一些用户所需的工具程序，省略设置实用程序、配置工具程序、本版需知等。

core版的程序可分为以下几类：

1) X窗口系统本身的程序。
2) 使用窗口系统必备的工具和设备程序：
• 日常的窗口相关功能的工具程序 (例如将窗口内容打印至打印机)。
• 一些你常常保持在桌面的实用程序 (例如时钟，日历)。
3) 可以利用窗口环境的一般应用程序。
4) 演示程序和游戏程序。
5) 信息和状态报告程序。
6) 定制你自己的环境的工具程序。

2. 系统程序

以下程序包含了所有和基本系统相关的程序。

1) X显示服务器 —— 这个软件控制了你的工作站的键盘、鼠标和屏幕。这是 X的心脏。此程序可以建立、删除窗口，同时可以应其他客户机程序的需求做写和画的动作。

这个服务器程序在各种硬件上有不同的实现，例如：

Xapollo —— 针对Apollo显示器。
Xhp —— 针对Hp 9000/300 的Topcat显示器。
Xibm —— 针对IBM 的APA16 和Megapel 显示器。
Xmacll —— 针对Apple 的Macintosh II。
Xplx —— 针对Parallax图形控制器。
Xqdss —— 针对DEC 的GPX 显示器 (VAXstation II/GPX)。
Xqvss —— 针对DEC 的QVSS显示器。
Xsun —— 针对Sun/2，Sun/3，Sun/4 和Sun/386i工作站。

2) Xinit —— 初始化程序，启动系统和设定服务器执行方式。

3) Xdm —— X显示管理器，它可以灵活地启动系统，使系统启动成符合个别需求的环境。它可以和 Xinit两者之间选择一个使用。

4) Uwm —— X窗口管理器。此程序决定如何管理你的桌面、移动窗口、重定窗口大小等等，你可以利用菜单，结合鼠标的按钮或键盘完成窗口操作。

这中间只有服务器程序是绝对必需的。不需其他的程序，你就可以在 X系统上运行其他的应用程序 。(Xinit 等程序可由其他相同功能程序替代。)

以上程序包含了窗口系统的基本元素。但除了在窗口上移动光标外，什么事也不能做。因此实际上，你需要更多的实用程序和应用程序。

3. 窗口系统中的实用程序

以下的工具程序并不是窗口系统的一部分，但它们是你使用窗口系统或利用窗口系统做更多的事所不可缺少的，它们分为以下两个部分：

(1) 窗口系统操作时常用的工具程序

只要你用窗口系统代替一般的电脑终端机，这些程序几乎是天天需要的：

xterm - X终端机模拟器。你的系统内大多数的程序并非特别为使用窗口系统所设计的。举例来说，一些最普通的系统程序 —— 列出文件目录，编辑器，编译器等，它们在普通的终端机可以正常的执行，可是它们不知道如何在X下操作运行。Xterm的作用就是建立一个X的窗口，且允许这些普通的 “ 哑终端机 ” 程序能够在这个窗口中执行。这些普通程序会认为它们是在 “ 真的 ” 终端机上执行。当然，你也可以用xterm 去启动其他的X程序而并非一定是那些普通程序。

xhost —— 让你控制网络上那些可以存取你的显示屏幕的其他主机。

xkill —— 一个可终止不希望运行的应用程序的工具程序。

xwd —— 将你窗口内目前的图像存储到一个文件中，使得你可以在稍后重建这个窗口、打印它或做一些其他你想做的事。

xpr —— 将先前 xwd所抓取的窗口图像转换成适合硬拷贝输出的格式。

xdpr —— 结合了 xwd和 xpr，允许你在一个步骤中就可以输出窗口的内容。

xmag —— 将屏幕上选中的一部分图像加以放大。

xwud —— 将先前 xwd所抓取的窗口图像重新显示于屏幕上。

x10tox11 —— 将能在第10版X执行的程序转换成可在第11版执行。

xrefresh —— 更新显示，将某些或全部的窗口重画一遍。

(2) 实用的程序

xclock —— 一个指针式或数字式的时钟。

xclac —— 一个计算器，可模拟科学工程型的计算器。

xload —— 用累计图来显示目前机器的负载分布。

xbiff —— X版的 biff。邮件到达告知程序， xbiff 会显示一个信箱的图标，当信箱上的旗子升起时，表示有你的信到达。

4. 一般的应用程序和工具程序

这些程序不直接和窗口系统相关，但他们可以利用窗口系统环境更加有效地运行。

xedit —— 一个文字编辑器，你可以用菜单或键盘发送命令，也可以用鼠标指针指定位置或一段文字。

xman —— 一个说明书或系统文件的浏览器。

xmh —— 一个邮件管理程序。

5. 示例和游戏程序

这些程序演示了X图形和彩色的能力。在你开始熟悉使用系统时，它们是一个良好的起点。

ico —— 显示一个20面体 (或其他多面体) 在窗口内进行碰撞运动的情形。

maze —— 以随机数建立一个迷宫并找出它的解法。

muncher —— 在窗口上描绘大量动态的图案。

plaid —— 在窗口上画一些持续变化的花格子图形。

xlogo —— 在窗口上印一个X的字形。

puzzle —— 智慧盘。在一个4×4方块盘上，移动编号1～15 的小方块，以排成特定型状的游戏。

6. 显示信息和状态的程序

以下的程序为你提供有关于窗口系统的信息和状态，你可以和自己的工具程序一起来使用他们。

xfd —— 在窗口内显示一个X中的字体，且可以选择提供更多有关此字体的信息。

xlsfonts —— X字体的目录程序，告诉你一个显示器上有哪些字体可供使用。

showsnf —— 显示服务器上原有的格式有关某一种字体的细节。

xwininf —— 显示某个特定窗口的信息，如大小、位置及其他特征。

xlswins —— 列出系统内所有的窗口，并可以选择地列出每个窗口的一些细节。

xprop —— 显示窗口的属性和字体。

xdpyinfo —— 提供你的显示器及控制它的服务器的细节。

xev —— 输出和窗口相关所有X“ 事件 ”的细节，用来侦错或给有经验的人使用的工具程序。

7. 可以用来定制适合你的系统的工具程序

一开始你可能不会使用这些程序，但过了一段时间，你可能发现你必需修改一下系统，例如想使用较大的缺省字体，窗口边框换成自己喜欢的颜色等。用以下的程序，可以使你的工作环境更加适合你。

xset —— 允许依照你的喜好设定显示特性。你可以设定一个键使它有效或无效，调整警告铃的音量，指定字体从何处获取等。

xsetroot —— 你可以选择显示屏幕背景的外观，当你的鼠标指针不在任何应用窗口内时，你可以改变屏幕背景的颜色或图案以及光标形状和颜色。

xmodmap —— 显示键盘的对应关系，也就是按什么键对应什么字符。利用此程序可以修改成适合你的对应关系，通常用来设定一些特殊键 (如 META，Shift-Lock等) 和功能键，但你可以视需要设定。

bitmap —— 让你建立和编辑图形的程序。例如用来改变光标的式样、编辑图标、改变窗口的背景图案等等。

xrdb —— 让你在数据库中显示或改变你喜爱的颜色或字体等等。以后应用程序可以使用的字体和颜色。也就是说，你可以设定一些缺省的特性，所有的应用程序或只有一些特定的应用程序使用这些特性作为缺省特性。

bdftosnf —— 将一种字体从BDF格式转成你的服务器原有格式。

27.2 X的基本结构

这里描述X的基本结构，并介绍许多基础的概念，其目的在于使你在稍后使用 X系统时能有一个了解。你将会知道系统程序做些什么和如何做，这样你将更快和更有效率地使用系统。

我们也会指出系统额外的作用，以及使用系统对你的影响。

27.2.1 X的基本元素

X不像早期的窗口系统，把一堆同类软件集中在一起，而是由三个相关的部分组合起来的：

1)“服务”程序：是控制实际显示设备和输入设备的程序。

2)“客户”程序：需凭借服务程序在指定的窗口中完成特定的操作。

3)“通信通道”：客户程序和服务程序用此来彼此交互信息。

1. 服务程序

服务程序用来控制实际的显示设备和输入设备 (键盘和鼠标或其他输入设备)的软件。服务程序可以建立窗口、在窗口中画图形、图像和文字；回应客户程序的需求。它不会自己执行动作，只有在客户程序提出请求后才完成动作。

每一个显示设备只有一个唯一的服务程序。服务程序一般由系统的供应厂商提供，用户通常无法修改。对操作系统而言，服务程序只是一个普通的用户程序而已，因此很容易更换一个新的版本，甚至可编译运行由第三方厂商提供的原始程序。

2. 客户程序

客户程序是指使用系统窗口功能的一些应用程序。把 X下的应用程序称作“客户”程序，原因是它们是服务程序的“顾客”：客户程序要求服务器应它的请求完成特定的动作。

客户程序无法直接影响窗口或显示，它们只能向服务程序发送请求，让服务程序来完成它们的需求。典型的请求通常是：“在 XYZ窗口中写一行‘ Hello，world ’的字符串”，或“在CDE窗口中用这种颜色从 A点到B点画一条直线”。

当然，针对窗口操作提出需求只是客户机程序的一部分，其他的部分是那些让用户执行的程序部分。例如：编辑文字、画一个系统的工程图、执行一个计算表格的计算等等。一般来说，客户程序和窗口系统是独立的，它们对于窗口几乎不需要知道什么。通常 (特别是指大型的标准绘图套装软件，统计套装软件等) 应用程序对许多的输出设备具有输出的能力。在 X窗口上的显示只是客户程序许多输出格式中的一种。所以，客户程序中和 X相关的部分在整个程序中只占了非常小的一部分。

用户可以使用不同来源的客户程序：一些是由系统提供的 (例如时钟) ，一些来自于第三方厂商，一些是用户为了特殊应用而编写的自己的客户程序。

3. 通信通道

X的第三个元素为通信通道，客户程序借助于它给服务程序发送请求。而服务程序借助于它向客户程序回送状态及一些其他的信息。

只要客户程序和服务程序都知道如何使用通道，通道的本身的结构并不是很重要。在系统或网络上支持通信类型的请求内建于系统基本的 X窗口函数库中，所有和通信类型有关的事件都从函数库独立出来，客户程序和服务程序之间的通信只要借助于使用这函数库（在标准 X版为xlib）即可。

总之，只要客户程序利用函数库，自然可以用到所有可用的通信方法。

客户程序和服务程序之间的通信大约可以分为两类，分别对应于两种 X系统的基本操作模式：

- 服务程序和客户程序在同一部电脑执行，则它们彼此均可使用机器上任何可用的方法做进程内通信(简称IPC)。在这种模式下， X可以像许多传统的窗口系统一样有效率地操作。

- 客户程序在一部机器上执行，而显示和服务程序在另一部机器上。则客户程序和服务程序的通信必需通过网络利用彼此同意的协议方可。目前，最常见的协议为 TCP/IP和DECnet，但其他任何的可信赖的协议亦可使用。

这种通过网络使得应用程序的操作如同在本地机器一样的能力称为网络透明性，这几乎是X独一无二的特性。这种特性使得它非常适合建立在灵活的多目标混合的计算机网络上。

因为客户程序和服务程序完全独立，这样开发了一种称为 X-terminal 的新型显示器。简单的说，X-terminal 是一种除了能直接在上面执行X服务程序外，什么也没有的工作站。它有键盘、鼠标和屏幕，以及一些和网络互相通信的方法（所以在其他主机上的客户机可在它上面显示），但并没有文件系统，也不提供一般目的的程序，一般目的的程序需要在网络上执行。

27.2.2 服务程序和客户程序如何交互通信

本节描述客户程序和服务程序互相通信时，双方各传输些什么信息。简单说，一个客户程序要求服务程序执行输出，输入则借助于“事件”来通知服务程序来处理（“事件”的含义：如按下键盘的键或鼠标的按钮等等）。

1. 客户程序送达服务程序的信息

当一个客户程序要求服务程序做一个动作，例如在一个指定的屏幕上建立一个有特殊特征的窗口，或者在一个窗口中写一行字符串，这时客户程序是借助于送请求到服务程序上来完成的。一个请求是一个被封装的简单区块，区块包含一个“操作码”来指示要执行何种操作，同时包括一些参数提供更多的请求细节。例如：清除一个窗口内的一个长方形区域，客户程序会送一个16位字节的请求区块，来指定是哪一个窗口，该长方形区域的左上角坐标及区域的高和宽。

这个格式有几个重要的特征：

- 请求区块的内容和客户程序与服务程序在何种类型上的机器上执行完全无关。一个客户程序可以输出请求给任何型号显示器上的任何 X窗口服务程序。请求和语言、机器及操作系统均无关。
- 每一个请求包含了窗口的细节和其他被使用的资源。对一个客户程序送至特定服务程序的请求可以提供超过一种以上的连接方法，所以在网络结构上提供的窗口数目没有限制。
- 请求区块通常大小为20个字节左右，算是相当小的，因为请求是设定为相当高级的，例如画一条线是指定两个端点而非记录一串屏幕上的点。通常屏幕上被影响到的像素的数目往往是区块本身大小的十到一百倍，这样不会使网络的负荷太重，网络的使用效率会非常高（一般认为X的服务程序和客户程序之间的传输是位图的观念是错误的）。

2. 服务程序送达客户程序的信息

服务程序也会利用通信通道送信息回到客户程序，这些信息包括回应客户程序请求是否成功和告诉客户程序有兴趣的特殊事件，这些事件包含的信息类似“窗口 XYZ 的鼠标左按钮被按”或“窗口ABC 已被重定大小等”。

就像从客户程序来的请求一样，服务程序的回应也是一些和语言、机器、操作系统无关的简单区块。

事件是X的基本功能。所有的键盘输入，鼠标按钮输入和鼠标移动都是由事件来控制。更进一步说，客户程序完全依赖事件才能获得那些在系统中发生的，而它必需知道的信息。下面我们将从一些普通的输入和移动功能着手，实际了解 事件是如何工作的。

3. 键盘输入

当你从键盘按下一个键，服务程序将会查觉到这个动作。服务程序便送出一个 <Key Press> 的事件通知那些已经登记的对这种情况有兴趣的应用程序。这种通知有一些限制：不是通知目前被鼠标指针指到的窗口，便是通知目前被指定接受所有键盘输入的窗口。这种限制称为设定键盘焦点。

当键被释放时，另外一个<Key Release> 的事件产生了（通常几乎是立刻），一般除了那些修饰键（例如Shift或 Control），应用程序很少会对释放键这个 事件有兴趣。

送到客户程序的信息区块告诉客户程序它们是键盘 事件，只是“ 编号第几的键已被按下（或释放）”，不包含是不是ASCII 或EBCDIC字符及如何解释等内容，而把这些留给客户程序去处理，这种做法使得客户程序看起来似乎复杂，但是标准的 xlib函数库中有非常简单的子程序可供控制解释键盘事件，而且通常缺省成你所希望的键盘型号。换个角度来看，这种“ 软件 ”的键盘字符相关方式提供了很大的灵活性。在服务程序这方面，对不同型号的键盘均可以完全重新对应；在客户程序这方面，每一个单独的键都可以“ 程序化 ”，例如按一个键即可以输入一串用户特定的字符串，或者完成一个特殊的功能等。

稍后我们会再详细讨论，到目前为止，这些将不会影响你使用系统。事实上，对于 X系统如何处理你按下一个“ A ”键，并将它转换成一个ASCII 的“ A ”字符送到你的应用程序的这类事情，你不需要太关心。

4. 关于鼠标指针位置的 事件

当屏幕上的鼠标指针进入或离开客户程序所控制的窗口时，客户程序可以要求了解这些事件。这些事件(<EnterWindow>和<LeaveWindow>) 告诉客户程序是进入或离开窗口以及是哪一个窗口。

当鼠标指针进入窗口时通常用类似“ 高亮度显示 ”窗口这一类的方式表达，有些应用程序可改变窗口的边框（例如从灰到黑），有些则会改变颜色，用以强调你目前正在处理这个应用程序（窗口）。

5. 当一个窗口未被覆盖时

X和大多数其他的窗口系统有一个很大的不同点，那就是客户程序必需负责保持它的窗口最近的内容，服务程序只是维持窗口在任何时刻均在屏幕上显示，但它不负责保持窗口的内容。

当原先被其他的窗口遮住的窗口（或窗口的一部分）变成可见时，服务程序并不知道应该显示这个窗口的哪个部分。服务程序送一个 exposure事件给拥有这个窗口的客户程序，告诉它窗口的哪一个部分刚刚已变成可见，客户程序便会决定该怎么作，在大多数的情况（一般为简单的应用程序或小窗口），客户程序只是重画整个窗口，因为只画窗口未被遮盖的部分往往要多花额外计算，并不值得。在更复杂的应用程序，客户程序才会只重画窗口必需要出现的部分，这是由应用程序的编写者决定，他必需在效率（窗口更新的速度）和只重画部分窗口程序码的复杂程度之间作取舍。

依赖客户程序来重画窗口内容的方式对效率特别重视，尤其是下拉式菜单，你总不希望选下菜单之后，菜单消失一段事件之后才让下面的窗口显示出来吧，为了克服这点，有些 X的产品包含了被称为save-under（存下层）的实用程序。

你可以告诉服务程序，如果可能的话，尽量在一个窗口被遮盖前将其被遮盖的内容存下，当遮盖的窗口被移走时便可立即重现而不需要送重现 事件给客户程序。

一个类似而更常用的，被称为backing store 的方式也被开发出来，你可以告诉服务程序尽可能在一个窗口被遮盖前将其全部内容存下，同样的，这种方式可以改进客户程序重画窗口的

效率，backing store 和save-under两者的不同处是前者保存整个窗口的内容，而后者只存被遮盖的部分。

虽然有了save-under和back store这两种产品，但你不能指望此种结构，客户机程序仍然随时保持准备接受重现事件。即使服务程序真的维护了一段时间的窗口内容，也可能因为内存不足而被迫停止，转而开始重新送出重现事件。

27.2.3 X的网络概况

我们曾经提过，客户程序和服务程序只需通过网络便可在不同的机器上执行，下面几节我们将看看如何利用这种实用的方法节省了计算的资源，从而增进了网络的成长。

1. 如何实际使用X网络

当服务程序在一个连接了显示器的机器上执行，而客户程序在另一部机器上执行时，鼠标和键盘的输入由服务程序所在的机器搜集，可是客户程序却可以在别的地方使用到这些输入，这是如何办到的？我们以下面的例子解释。

假如你在使用一个由X服务程序控制显示器的工作站，如果它是独立的，很明显，客户程序也在此工作站上执行。即使连接了网络，大部分的时候你还是在你自己的工作站执行客户程序。可是因为有一些特殊的实用程序，你的机器上并没有，而你却希望在你的机器上显示程序的输出，这时你便需要网络上的机器了。利用你的操作系统提供的一些普通的网络设备程序，你便可以让客户程序在远程的机器上执行，而指定输出显示在你自己工作站的显示器上。

假设客户程序的名称为xgraph，在UNIX系统上，你所发出的命令类似下面：

```
rsh neptune xgraph -display venus:0
```

则xgraph程序在远程名为neptune 的机器上执行，且xgraph的输出会送到你自己名为venus的机器上的 0号显示器上。从现在起，我们将参照这种远程显示的模式操作，当客户程序在一部机器上执行时，服务程序在另一部机器上执行。

现在总结一下：你使用远程显示的设备程序使得客户程序在远程的机器上执行，而且告诉它将输出显示在执行X服务器程序的本地机器上。

2. X的网络设备有何用途

在一部机器上执行客户程序而把输出显示到另一部机器有何用途？这些用途和实用是极常见的，以下是一小部分的用途：

- 远程的机器速度比你的计算机快很多（可能是因为加了浮点运算器或它根本就是一部超级电脑）。
- 在你的局域网上，远程的机器是一部文件服务器，它提供了大量磁盘资源，为了降低网络的负担，你可以把一些类似搜索的大量操作，需要用到的大量磁盘动作的程序放在远程机器上执行，这样一来，只有执行结果而不是大量操作磁盘的动作会通过网络传送。
- 远程机器有特殊的结构适合特别的工作，可能是专门的数据库机器，或者是为一个单独的应用特别设计的特殊目的机器。
- 远程的机器有只能在其上执行的特殊软件。在现代的工作站，在网络上有些软件许可只有少数的机器拥有已是愈来愈多的趋势，因为软件许可只发给那些付费的工作站。在这种情况下，可以实际地在远程的机器上执行这些有许可的软件，而将显示传回你自己机器上，这相当地实用。
- 你需要同时存取好几部机器，通常系统的管理员有此需求。
- 你需要同时输出到数部显示器。

3. X网络结构产生的简易性

就像前面所提过的，所有从客户程序向服务程序发出的请求，由于它们的格式和内容是和设备无关的。而所有和设备相关的事完全集中在服务程序，对于任何显示器的硬件，只有对应于此种显示器的服务程序才需要去关心。只要提供针对一个显示器的服务程序，所有可执行 X 客户程序的其他机器立即可使用这个显示器，不需要重新编译或重新链接，甚至连显示器是什么型号都不需知道。

这种把设备的相关性独立出来给服务程序的方式，对许多工作站网络的供应商变得可行且轻松，这种灵活性在两方面特别有用：

当一部执行X客户程序的新机器加入网络，它立即可以使用任何执行 X 的显示器。

相反，当一个新的显示器加入时，它立即可被任何机器上现存的所有 X客户程序使用。

4. 在网络上使用非X的应用程序 —— 终端机模拟器

如果在远程机器上的程序并不是 X客户程序或甚至连 X是什么都不知道，你仍然可以像远程的机器一样使用它们。这就需要用到 X窗口“ 终端机模拟器 ”，一个假装它是终端机的程序。这样一来，你便可以让任何程序在这个假的终端机执行。这个终端机模拟器利用 X显示输出（和得到键盘输入），当然输出也可以送到本地或远程的显示器。

27.3 从用户界面的角度概观X

本节将观察重点放到系统控制的用户界面，例如，系统如何显示有人使用，以及包含那些结构等。

X设计的目标之一就是能支持许多不同型号的用户界面。一般其他的窗口系统提供特殊的交互方法，而X则提供一般性的结构，让系统建立者据以建造所需的交互的样式。例如，在一个X系统中可从菜单中选一个动作来构建窗口，但其他对窗口的操作则全靠鼠标来做，这种灵活性允许系统开发者完全在 X的基础上产生全新的界面，也因为界面并未内建于窗口系统，因此用户在任何时刻根据他们特别的需求可选用适当的界面。例如，对于完成一些相同的工作 —— 建立、移动、重定大小屏幕上的窗口等，初学者较老手喜欢简单的系统，而 X可分别提供最适合他们的用户界面。

用户界面分为两个部分：

- 管理界面：命令最高层的窗口如何在屏幕上配置或重配置，也就是说，如何管理桌面。
- 应用界面：决定你和应用程序间交互的“ 样式 ”，即你如何利用窗口系统的设备程序来控制应用程序及输入资料给它。

27.3.1 管理界面：窗口管理器

管理界面是系统的一部分，用以控制屏幕上最上层的窗口（换句话说：如何重新配置你的桌面），这个部分在系统中称之为窗口管理器，它的功能有改变窗口的大小或位置、将窗口在堆栈中重新安排位置或将窗口改变成图标 等等。

在X中，窗口管理器只是另一个客户程序程序而已。它以及系统界面的开发和服务程序是完全分开的，因此你可以更换它们，这类似于 UNIX系统中的外壳命令行解释器。外壳只是一个用户处理程序，如果你改变它，你也改变了系统的用户界面。

1. 手动的和自动的窗口管理器

有两类的窗口管理器：手动的和自动的。手动的窗口管理器，窗口在屏幕上的位置和大小完全由用户控制，手动的窗口管理器只是用户用来完成工作的工具，大部分的手动窗口管理器

允许应用窗口重叠。

而自动的窗口管理器尽可能由它自己来控制桌面，对于屏幕的布置尽可能让用户少插手。它在新建立一个窗口时自动决定窗口的大小和位置，和当窗口移动时如何重新安排其余的窗口，通常自动的窗口管理器将屏幕分成一块块像磁砖一样的区域，也就是说安排应用窗口彼此不会重叠，而且尽量占用最多的屏幕空间。

通常当你希望告诉手动的窗口管理器你要完成什么动作时，是借助于使用菜单或者结合了按鼠标的按钮和移动鼠标指针的方法。例如，重新摆放一个窗口的位置，你可以移动鼠标指针进入窗口，按住左边的按钮，移动鼠标指针然后在新位置释放按钮，窗口管理器是如何知道这些鼠标事件的意图的？或是换个角度，服务程序是如何知道事件是来自应用窗口还是窗口管理器？

答案是由窗口管理器告知服务程序有哪些特定的 事件（按按钮等等）需要被送达，这和哪一个窗口发生的无关。这种处理称之为抓取。窗口管理器可以指定希望抓取哪一个鼠标按钮，而这抓取发生在鼠标的按钮被按下且键盘上一些特定的键（一般称为修饰键)也被按住，当按钮被按下时，抓取开始操作，服务程序送出所有鼠标的事件（包括鼠标的移动事件）到窗口管理器直到按钮再度释放，窗口管理器把这些事件的资料解释成来自用户的指令来工作。以移动窗口为例，窗口管理器在按钮按下时被告知鼠标指针的位置，而当按钮释放时再度被告知，对鼠标指针的位移做一些简单计算便可据以移动窗口。

有一件事需要用户配合，那就是鼠标和修饰键组合而成的抓取不应该为应用程序所知道，所以必需确定窗口管理器这种抓取键的组合不会和应用程序冲突，大多数的窗口管理器可以很容易地定义这些抓取的组合键，而保留给它自己使用。

2. 窗口管理器额外提供的功能

窗口管理器除了具有重新配置窗口的基本功能外，也提供额外的功能改进界面的品质。通常，加入额外功能的目的是为了降低键盘输入的需要，而改成尽量多用鼠标。

一个常见的功能是提供自己可以配置的一般性菜单，这样你只要选取一个菜单选项便可启动窗口应用程序。这个启动的命令通常包含了指示应用窗口在何处出现，大小多少，文本用什么颜色等等。所以应用程序不需要太多的用户输入便能启动。一个常见的菜单用法为，当你在网络上工作时，你可以定义一个菜单列出所有你在网络上可用的主机，这样在菜单上只要选择主机名称便能和任一主机建立连接。

3. 窗口管理器和图标

当一个窗口转换成一个图标时，图标是如何来的？窗口又发生了哪些事？

图标的结构非常简单，它只是窗口的代表图案，当系统图标化一个应用窗口时，窗口管理器只是不映射这个窗口（也就是说，告诉服务程序不再显示这个窗口）而把图标窗口映射出来。解除图标化则把上述的处理反过来。

窗口管理器通常提供缺省的图标，但是客户程序可以提供它自己的图标并建议使用它，有些窗口管理器接受这个要求，有些则忽略此请求仍用自己的图标，只把这个请求当作给窗口管理器的暗示。

当应用程序被图标化以后，它的主窗口便不再被对应出来。如果窗口管理器因任何理由中断了，则这个窗口永远也无法再对应出来了。如果希望避免这样的事情发生，当窗口管理器图标化一个窗口时，它把这个窗口加入一个名为 save set的名单，这个名单由服务程序负责维护，这样当窗口管理器被中断时，此窗口也可重新对应出来。

4. 应用程序给窗口管理器传递配置信息

就如同要求显示一个特定的图标一样，应用程序也能给窗口管理器传递其他的暗示或配置信息，这包括：

- 应用程序和图标窗口的名称。
- 当应用程序和图标窗口建立时，它们在屏幕上位置的信息。
- 对窗口大小的限制（例如，客户程序可以宣告“我所占用的窗口大小绝不可小于宽度若干x 长度若干”）。
- 对窗口重定大小的特别要求（例如，一个显示文本的窗口，可以要求在重定大小时按特定的间隔放大或缩小，以使得窗口内的字符永远是完整的一个，不致窗口边框的那一行(列) 有半个字的情况出现）。

我们可以注意到大部分重定大小或图标化的事是由窗口管理器做的，这是因为它是一个公共的客户程序。任何客户程序均可随意重定大小，但如果所有客户程序这样做，便会造成混乱，因此要这些应用程序和平共存的原则是：不要自行重定大小，把它交给窗口管理器，也就是让用户去决定。

27.3.2 应用程序界面和工具箱

应用程序界面决定了用户和应用程序间交互的样式。例如如何用鼠标选一个选项等。 X不提供标准的应用程序界面，只提供基本的结构以便建造它们。

把那些具有通用性的应用程序界面放在一起，便形成了一个工具箱，它是基础窗口系统软件中最高最有效率的层次。较低层次的细节，将被隐藏起来，因此简化了程序，同时维持界面的一贯样式变得容易。当用户控制应用程序时好像有一套“虚拟语法”一般。需要注意很重要的一点是，工具箱在编译程序的时候才能决定，所以一个客户机的应用程序界面在编译的时候就被决定了，如果不重新编译便无法改变。

MIT 版的X大多数的应用程序均使用标准的工具箱和一套来自 MIT 的工具箱软件构成要素，这使得你可以得到一个一致性的界面。除此之外，有些结构提供了定制的应用程序操作方法和设定它们的缺省值。

27.3.3 其他系统角度

本节将讨论应用程序之间传递信息所用的属性结构，窗口的树状层次组织和 X不包含在操作系统中的优点。

1. 客户程序之间的通信 —— “属性”

客户程序和服务程序之间的通信是借助于送出请求和接收事件。但有时客户程序需要和其他的客户程序传递信息，例如，正常的应用程序需要告诉窗口管理器它的位置和大小，这就需要X的属性结构了。

“属性”是一小段数据的名称，这一小段数据存在服务程序中且关联到一个特定的窗口，任何客户程序均可向服务程序查询某一特定窗口“属性”的值。

让我们看一个客户程序如何把它所喜欢的图标名称传递给窗口管理器的例子：客户程序把图标名称存到这个窗口的WIM_ICON_NAME “属性”去。当窗口管理器执行图标化这个应用窗口时，它会去找这个应用窗口的WIM_ICON_NAME的“属性”，而后显示“属性”中的图标名称。

应用程序也可以和不是窗口管理器的其他的应用程序通信，一个常见的例子是在分属不同应用程序的窗口之间做剪贴操作，一段文本从一个应用程序中“切下”稍后再“贴”到另一个

应用窗口，在此用到了“属性”，“属性”依序编成“CUT_BUFFER0”、“CUT_BUFFER1”…等等，所有的应用窗口便可借此交换信息。

最后一个例子是称为资源的属性，它用来定义应用程序的缺省值设定。在根窗口中有一个名为RESOURSE_MANAGER的属性存放着所有设定的名单，所有的应用程序都会存取它，用来做是否要执行任何设定的依据。

2. X中窗口的层次式

本节描述窗口在系统中的组织及如何建立和窗口对应用程序的影响。

所有在X中的窗口都可视为一个树状结构层次的一部分，树的根部便是根窗口，涵盖了整个屏幕，应用窗口都是根窗口的子代，上层的窗口可以拥有它自己的子窗口。

在X的设计理念下，制造一个窗口非常容易，你可以利用窗口来控制选项，像菜单、滚动条、控制按钮等等，即使是大量的选项也无妨。这种观点更有利于程序员而不是用户，但的确对用户“定制”特定的程序时有影响，在本节以后会再度提到。

为了允许应用程序有子窗口，X提供了大量的设备程序供客户程序使用，这样不但能完成一致性，也避免了相同的需求造成了重复的工作。

子窗口的位置和大小并不受父窗口的限制，子窗口可大可小，可以大过父窗口或只占父窗口的一部分，但是它会被父窗口剪裁，也就是说，子窗口所有超出父窗口的部分将会消失不见。

在实际的应用上，你可以将上层的窗口定义成几乎占住整个屏幕，就不必担心子窗口有些部分会看不到了。

另外一种方式就是把下拉式菜单定义成为根窗口的子窗口，这样菜单便可以比应用窗口还大。

3. X不是嵌入于操作系统中的

不像其他大多数的系统，X并非嵌入于操作系统中，而只是比用户层次稍高而已。更精确地说，X不需要嵌入于系统。虽然有些制造厂商可能是为了效率的原因将服务程序和操作系统结合在一起，但不嵌入于操作系统的结构有下列的好处：

- 易于安装和改版，甚至去除。这种工作不需要重新启动系统，也不会对其他应用程序造成干扰。
- 第三方厂商很容易加强它的功能。例如你的制造厂商提供的系统不够好，你可以向别人买更好或更快的版本。
- X不会指定操作系统，因此成为一种标准，这也是第三方厂商开发软件的原动力。
- 为了开发者利益。在服务程序上开发工作时，当程序中断只会中断窗口系统，不会造成机器的损坏或操作系统核心的破坏。没有操作系统核心码的程序也较易排错。

27.4 术语和符号

27.4.1 术语

在X中，一个窗口是指屏幕上的一块长方形区域，它的边平行于屏幕的边。大多数的窗口以一种颜色作为背景色，而以另一种颜色作为前景色。例如一个典型的文字窗口，背景色为白色，前景色 (也就是文字本身) 则为黑色。窗口可以有一个边框，通常边框的颜色和背景色不同。有些窗口在窗口上方可能有一个标题栏或控制栏，在某些情况下用以显示有关这个窗口的信息，你可以对控制栏作某些固定的动作来管理窗口。系统会显示在屏幕上显示一个鼠标指

针，当你移动鼠标时，整个屏幕只有一个鼠标指针在对应移动。屏幕上许多文字窗口拥有自己专用的文字光标，这些光标通常指示你输入文字的位置。

1. 几何意义：位置和大小

X用到一些有几何意义的术语来说明一个窗口的位置和大小。 大部分的X程序接受一个含有几何意义的命令行来启动它们，这个命令行说明了这个程序的窗口有多大，以及在屏幕的哪一个位置显示。通常格式如下:

宽度×高度 + x偏移量 + y偏移量

宽度和高度的单位为像素(屏幕上的一点) 或字符，视应用的状况而定。程序的说明通常会告诉你用什么单位。上述的式子是说明建立一个大小为宽×高的窗口，窗口的位置为左边框距屏幕左边界 x偏移量个像素，上边框距屏幕上边界 y偏移量个像素。例如假设一个程序以字符为窗口大小单位，则格式

80×24 + 600 + 400

其意义为: 建立一个 80 字符宽 24 字符高的窗口，并且窗口的左边框距屏幕左边界 600个像素，上边框距屏幕上边界 400个像素。

如果需要的话，也可以只指定大小或只指定位置，程序对未指定的部分会使用缺省值 ，或给你一些提示，视实际在系统中执行的状况而定。

2. 鼠标和鼠标指针的术语

有一些输入设备会在执行 X时在显示器上指出屏幕上你有兴趣的项目或区域，通常为一个有数个按钮的鼠标 (一般为三个按钮，分别称为左按钮，中按钮，右按钮)。当你移动鼠标，系统会对应地移动屏幕上的鼠标指针。接下来，我们对鼠标上的三种操作术语定义如下 :

- 单击 : 按下鼠标的按钮随即释放，按钮被按下的时间，仅有一瞬间而已。
- 按住按钮: 将鼠标的按钮按下，且一直保持按住按钮的状态。
- 释放按钮: 将先前按住的按钮释放。

通常单击用来指定屏幕上的一个对象，按住按钮再释放按钮 (一般在这期间会移动鼠标) 往往用来移动或描绘一块区域。

拖动对象: 利用鼠标指针指定一个对象，按住按钮，保持按住状态移动鼠标指针直到某处再释放按钮。做这种操作时，系统通常有一些方式来表示对象被移动，例如在拖动一个对象的期间，系统会将对象周围加上一个细线的方框。

我们常常利用拖动方式来改变一个对象的大小，通常系统显示方框，根据你的拖动动作改变大小，此种方法叫作橡皮筋法。(因为方框好像用橡皮筋做的一样)

3. 键盘的术语

标准的终端机键: Shift、Delete、Backspace、Esc 或 Escape、Return、CapsLock。

光标控制键: 采有上下左右箭头的键，如 Up、Down、Left、Right。

特殊键: 按住Ctrl或Control键，再按其他的键 (例如 A键)，用Ctrl-A表示，有些终端机有Meta键，也同样的用Meta-A表示。

27.4.2 符号

在一些情况下，你输入的命令行或系统输出的文字，因为太长而无法在同一列而必需分为数列，如果它是shell 命令，或是一段 C语言程序码，我们在第一行的最后加上一个倒斜线(\)后，在下一行继续，例如:

```
mkfontdir/usr/lib/X11/fonts/misc\
```

```
/usr/lib/X11/fonts/15dpi\
/usr/lib/X11/fonts/100dpi
```

然而极少数的情况下，我们用符号“ (contd.) ”表示本行因排版限制的缘故在下列继续，如:

```
PID TT STAT  TIME COMMAD
1901 c0 S 0:01 x :0
1902 c0 S 0:01 xterm -geometry +1+1 (contd.)
      -n login -display unix:0 -c
1903 p1 S 0:00 -sh (csh)
```

当X设置时，需要设定一些目录树。我们把目录树的顶端定为 $TOP，在我们的系统中，$TOP对应的目录为 /usr/local/src/X11，相同地，主目录参考自 $HOME。

27.5 启动和关闭X

在此假设系统管理员已经在系统上设置好了X，事实上即使不曾用过或不熟悉X，设置X也不会很困难。因此如果有必要，需自己设置X。

在还未开始前，我们需要先知道已设置好的 X的执行程序在哪里。 MIT 版缺省的目录为 /usr/bin/X11，但很多地方是用 /usr/local/bin 或/usr/local/bin/X11。 当你知道了之后，把它加到你的搜索路径里，如果你使用 C-Shell，可以在你的.login文件 (或者可能是.cshrc文件) 设定路径，如果你使用Bourne Shell，则在 .profile文件中设定。例如，在 .login文件中使用C-Shell 的命令行设定路径:

```
set path = (. /usr/local/bin/X11 /usr/ucb /usr/bin /bin)
```

如果不设定路径，X将无法正常启动。 当你设好之后，为了确定起见，先登录再退出系统一次，以便检查路径是否设定正确 (用 echo $PATH 指令)。

27.5.1 启动X

在你的显示器启动X，键入命令：

```
xinit
```

则会发生：

1）你的整个屏幕会被设定成灰色。

2）一个巨大的“ X ”光标出现。你可以用鼠标在屏幕上移动它，但按鼠标按钮或键盘都对它无影响。

3）一个xterm 终端机模拟器的窗口出现在屏幕左上角，当光标移到这个窗口时，会改变成文本光标，xterm 准备接受你的命令。

X现在已启动，你可以把xterm 这个窗口当成一个普通的终端机来使用，执行一些普通的指令。不过它最大的价值在让你可以开始执行其他的 X程序，这一点我们将于稍后告诉你。现在先来了解一下X的启动动作做了些什么。

首先，xinit在你的显示器上启动执行X服务器程序。服务器程序建立一个它自己的根窗口，并把窗口的背景色设定成灰色，把光标设定成一个大“ X ”。

在服务程序执行的期间，服务程序一直控制着键盘及鼠标，这就是你能在屏幕上移动光标的原因。但是因为目前没有任何客户程序要求了解键盘和鼠标 “ 事件 ”，所以服务程序只是追踪鼠标和光标的移动，而所有其他的键盘或鼠标输入虽然都经过服务程序处理但均被放弃， (因为没有客户程序有兴趣)，这就是按键盘或鼠标按钮没有反应的原因。

接下来，xinit 启动执行xterm 程序。xterm 对服务程序而言是一个客户程序，xterm 要求服

务程序建立一个窗口，而且保持了解在这个窗口中的鼠标和键盘事件， xterm 设定在窗口中执行一个外壳程序，当鼠标指针移至窗口之内便准备接受输入。

键盘输入被送至外壳就如同在一部真的终端机上输入一样，从外壳（及其子程序）的输出借助于xterm 显示在窗口上。 xterm 也接受鼠标输入，使得你能设定不同的程序操作参数和进行文本的剪贴。

你可以观察到系统执行这些动作的步骤。例如当在系统启动后，在 xterm窗口内执行 ps a 命令:

```
PID TT STAT TIME COMMAND
1900 C0 S 0:00 xinit
1901 C0 S 0:01 X:0
1902 C0 S 0:01 xterm -geometry +1+1 -n login -display unix:0 -c
1903 p1 S 0:00 -sh (csh)
1904 p1 R 0:00 ps
```

以上的显示说明 xinit 在主控台显示器上运行。它初始化服务程序， X显示为零。接着 xterm 在一个虚拟的终端机上执行。 xterm 启动一个外壳程序，使得它能处理你在xterm 窗口所下的命令。最后，我们执行ps命令产生上述的列表。

我们将在以后讨论更多的xterm 细节。 从现在起，我们假设xterm 被视为一个DEC VT102 的终端机，我们把重点转移到系统启动之后，我们能做些什么。

27.5.2 执行X程序的方式

你目前有一个X服务程序控制的显示器，一个叫 xterm 的客户程序允许你输入命令。本节介绍如何执行其他的X程序。

因为X的客户程序和X服务程序完全独立，所以不需要特别的动作启动它们。你可以像执行一般的程序一样执行它们。但是这些客户程序需要确实知道它们用的是哪个显示器。实际上因为xterm 一开始设定了DISPLAY 环境变量，给定了它使用的显示器名称，而其他的客户程序用此当作缺省显示器，因此你不需多做其他的事。

1. 如何执行X的时钟，xclock

我们用X的时钟当作一个简单的例子。先确定鼠标指针停在xterm 窗口中，然后输入命令：

```
xclock
```

一个小的时钟影像出现在屏幕左上角，覆盖了第一个窗口一部分。

现在有三个问题要解决：

- 第一个问题：由于xterm 这个“终端机”已经有一个程序（xclock）在执行，所以我们无法再输入其他的命令，该怎么办？

唯一的办法就是停掉xclock，但当你按下Ctrl-c或DEL 键时，xclock便会消失。要克服这种状况，你需要非同步执行xclock，用命令:

```
xclock &
```

则目前xterm 至少能接受你输入其他的命令。

- 第二个问题：如何中止xclock？

X服务器程序本身没有提供直接的界面中止应用程序，但是有一个叫 xkill的客户程序可让你终止应用程序。在xterm 窗口内输入xkill 命令便可启动这个程序。xkill 会显示一个覆盖性的方形光标，移动这个光标到任何你想终止的应用程序的窗口中，按左按钮，应用程序的窗口会消失，且应用程序和xkill 会一起结束。你也会得到如下的信息：

xkill:killing creator of resource 0x40004d

XIO:fatal IO error 32 (Broken pipe) on X server “ unix:0.0" after 207 requests (178 known processed) with 0 events remaining。

The connection was probably broken by a server shutdown or kill-client。

如果为了某些缘故无法进到应用程序的窗口内用 xkill 中止它，你通常可以用UNIX的办法：找出进程的ID，然后终止它。例如：

```
$ps a | grep xclock
1907 p2 I 0:00 xclock
1909 p2 S 0:00 grep xclock
$kill 1907
[1] Terminated xclock
$
```

• 第三个问题：如何避免时钟和 xterm 窗口重叠？这个问题换个问法是：你如何安排应用程序窗口的位置？

你可以用前面说明过的几何意义的参数来解决。例如输入命令：

```
xclock -geometry 200x300+400+500 &
```

这个命令告诉xclock建一个宽200 高300 个像素的窗口，位于屏幕左上角右边400 个像素，下边500 个像素。

如果你拥有彩色显示器，那么不妨以 xclock进行你指定和使用彩色的实验。 xclock有数种选项做彩色识别：

-bg color 设定背景颜色

-fg color 设定前景颜色

-hd color 设定时钟指针的颜色

-hl color 设定时钟指针边线的颜色

输入指令：

```
xclock -bg turquoise -fg red -hd magenta
```

你可以看到一个彩色的钟，稍后我们会再说明颜色的正确使用名称。

xclock启动之后，便不再需要和用户交互了。后面我们将介绍另一个需要从键盘和鼠标输入的小程序。

2. xcalc —— 桌上型计算器

xcalc 是一个X的计算器，移动鼠标指针到xterm 窗口，输入命令：

```
xcalc - geometry +700+500 &
```

一个像TI-30 型计算器的窗口出现了，你可以用鼠标或键盘来操作它。

使用鼠标时，你可以移动鼠标指针到你需要的计算器按钮，按鼠标左按钮表示按下按钮。如果是用键盘，键盘上的一些键对应计算器按钮，例如依序按键盘键 1、+、2、+、3 和 =键，代表了算1、2、3 的总和。由于至少目前你可以用鼠标指针指到计算器的任一按钮，因此那些键盘和计算器比较不明显的对应关系，在此不作进一步说明。

xcalc 比 xclock 有一个优点，那就是容易中止它。在计算器AC按钮上按鼠标右按钮即可中止，大部分的X应用程序均有类似的中止设备。

27.5.3 关闭X

要关闭X窗口，只要移动鼠标指针到最初xterm 的窗口，输入：

```
logout
```

则窗口消失，服务器程序终止，X也被关闭。

详细来说，xterm 查觉到外壳程序终止时，也将终止自己。而 xinit 一查觉xterm已经结束，便终止服务程序后离开。

27.6 窗口管理器基础——uwm

系统应该需要更多、更实用及容易使用的功能，在X中，这些功能由窗口管理器提供。

27.6.1 什么是窗口管理器

X系统最基本的部分——服务程序，只提供最基本的窗口功能，如建立窗口、在窗口中写入文字或画图形、控制键盘和鼠标的输入和删除窗口等。服务程序不提供用户界面，它只提供建立界面的基本结构。

我们把用户界面分为两个部分——管理界面和应用界面，下面先讨论讨论管理界面。管理界面由窗口管理器控制，提供管理桌面的功能，例如建立应用窗口，在屏幕上移动它们，重定大小等等。

你也需要能够：

- 使一个原来被遮住的窗口重新显现。
- 实用地启动或中止应用程序。
- 刷新屏幕。
- 图标化和解除图标化。

27.6.2 启动 uwm

当X启动后，你可以在屏幕上的任何外壳窗口启动uwm，因为窗口管理器也只是一个普通程序而已。你可以在执行X的任何期间内启动uwm，但通常是在一开始的时候。

现在你可以先启动X，接着在xterm 窗口内输入下列命令：

```
uwm &
```

uwm 执行后会让终端机的喇叭发出声音表示它已初始化且准备工作，但你在屏幕上看不到有任何改变。执行一个ps a命令，你可以看到现在有一个uwm 程序如下：

```
PID TT STAT TIME COMMAND
1900 co S 0:00 xinit
1901 co S 0:01 x:0
1902 co S 0:01 xterm -geometry +1+1 -n login -display unix:0 -c
1903 p1 S 0:00 -sh (csh)
1904 p1 I 0:00 uwm
1905 p1 R 0:00 ps
```

现在我们有一个窗口管理器了，接下来我们将利用它完成一些基本的操作。

27.6.3 基本窗口操作——uwm 的菜单

uwm 有一个菜单的功能，可用来管理菜单，其存取的方法如下：

1）将鼠标指针移到灰色屏幕背景的任何地方。

2）按住鼠标的中间按钮不放，一个标头为“WindowOps”的下拉式菜单将会出现。

3）继续按住按钮，上下移动鼠标指针，被鼠标指针指到的选项会以高亮度显示或以反白表示，当你释放按钮，表示此高亮度显示的选项被选择。

如果你不想选择，那就按一下其他鼠标键，或者将鼠标指针移到菜单的边框外面，则菜单将会消失。

现在选择Refresh Screen (刷新屏幕)，并且释放按钮，则屏幕闪动一下并完全重画。

27.6.4 移动窗口

在屏幕上移动窗口的步骤如下：

1）将鼠标指针移至背景，按住鼠标中间按钮，调出 uwm 的下拉式菜单。

2）选择“ Move ”选项并释放按钮，此时光标改变成“ 手指 ”形状。

3）将“ 手指 ”移动到你打算移动的窗口中，按下任何按钮，同时按住不放，窗口上出现了九字格，且光标变成十字箭头形状。

4）继续保持按住按钮，移动光标，将九字格拖动至你想摆放窗口的新位置。

5）释放按钮，窗口会跳到新的位置，同时九字格消失。

27.6.5 重定窗口大小

你可以在一维空间或两维空间中重定窗口大小。例如：你可以只把窗口加宽，或同时将窗口变高及变窄。重定窗口大小步骤如下：

1) 调出uwm 的下拉式菜单，选择Resize选项，如同移动窗口，你的光标变成“ 手指 ”形。

2) 移动光标到欲重定大小之窗口的右下角。

3) 按住鼠标按钮，保持按住状态，有三种变化发生。

- 光标变成“ 十字箭头 ”形。
- 九字格出现，但不像前节和窗口一样大，它比较小。
- 出现一个长方框，显示目前窗口的大小。

4) 移动光标，延展或挤压九字格直到大小合乎需求。

5) 释放鼠标按钮，窗口改变大小将和九字格一致，同时九字格消失。

1. 九字格的目的

在重定大小的操作中，九字格具有让你预先看到重定窗口的大小，而当你在步骤 3按下按钮时，当时光标在九字格的位置决定了你的动作：

- 当你在九字格的四个角的格子或最中间那一格按下按钮，你可以任意水平或垂直改变窗口的大小。
- 当你在九字格四边中间那一格按下按钮，你就只能在一度空间改变大小，你只能移动窗口最接近你按下按钮的格子的那一边。

2. 大小限制

那个显示目前窗口大小的长方框，其大小的单位视情况有所不同。文字窗口，其意义为若干行乘若干行字符 (例如xterm 通常为80×24 字符大小)，图形窗口，其单位则为像素 (例如xclock缺省的大小为150×150 像素)。

有些窗口的外形或大小会有所限制，例如xcalc 有最小尺寸的限制：它不允许你把窗口缩小到连计算器上按钮都无法显示的地步。xterm 虽然可以任意重定大小，但它以字符为单位，它不会允许窗口最下一行字符只出现一半的情况发生。而xclock几乎对任意大小或外形均不受限制。

27.6.6 建立新窗口

利用窗口管理器uwm的NewWindow选项，我们可以很容易地建立一个新窗口。我们在下

面描述如何启动一个新的xterm ，uwm 如何帮助你启动其他的应用程序，以及你如何控制应用窗口的起始位置和大小。

1. 建立新的xterm 窗口

建立新的xterm 窗口步骤如下：

1) 移动光标到背景窗口，调出uwm 的下拉式菜单，选择“ New Window ”选项，在释放按钮的一瞬间，有三种变化发生：

- 光标改变成“ 左上角 ”形 。
- 一个闪动的新窗口边框出现了，光标在左上角。
- 一个类似我们前面讲过表示窗口大小的长方框出现和以前不同的是，它比以前多了窗口的名称 。

2) 移动光标使得新窗口的左上角移到你所需要的位置。

3) 按一下左按钮，一个新的窗口便产生了，显示窗口大小的长方框和闪动的边框同时消失。

你可以像使用原始xterm 窗口一样使用这个新窗口来执行普通的应用程序或X的应用程序。

2. 建立可以供任何应用程序使用的窗口

我们仍然可用以前的方法 —— 在xterm 窗口的外壳程序中输入一行命令来启动应用程序。但是现在你有窗口管理器程序在执行，所以你可以用交互的方式来控制窗口的起始位置，而不需在命令行中设定几何意义的参数。 (事实上， uwm也可控制窗口起始的大小，我们会在下面描述。)

举一个例子，假设我们要在屏幕的右上角启动 xclock：

1) 在xterm 窗口中，输入命令行：

```
xclock &
```

就如同NewWindow选项一般，你可以看到一个描述窗口大小的长方框，一个“ 左上角 ”形光标，一个和时钟同样大小的闪动边框。

2) 不要按任何钮，只要把边框拖动到任何你想要摆放的位置。

3) 按左按钮，一个时钟替换了闪动边框出现。

3. 指定新窗口的大小

前面提到当你建立新窗口时，若你按的不是左按钮，会有一些奇怪的情况发生，事实上三个按钮各有不同的意义，你可以依需要做适当的选择：

1) 左按钮：按下左按钮会使得：

- 位置 —— 将窗口左上角的位置依目前光标的位置决定。
- 大小 —— 应用程序本身原先缺省的大小。

2) 中间按钮：你不应该按中间按钮。但如果你按住不放的话，你可以借助于改变窗口的右下角来改变窗口的大小，然后释放按钮：

- 位置：窗口左上角的位置依你按下中间按钮时光标的位置决定，右下角则根据你释放按钮时决定，按住按钮的期间，窗口的边框就像橡皮筋般可延展或压缩。
- 大小：根据释放按钮时的右下角决定。

如果应用程序指定了窗口最小的尺寸限制，则橡皮筋边框被压缩到比最小窗口还小时会自动消失，确保你无法建立一个比最小窗口限制还小的窗口。

3）右按钮：按右按钮会使得：

- 位置：窗口左上角依目前光标的位置决定。

• 大小：窗口的宽度为缺省的宽度，窗口的高度由光标的位置直到屏幕的底边，如果大小低于应用程序缺省之最小窗口限制的话，则用缺省的高度来代替。当然，这也意味着会有一部分的窗口超出屏幕，所以无法看到。

4. 更多有关于几何意义的参数的设定

关于几何意义的参数的设定，过去我们都是用窗口左上角的位置相对于屏幕左上角位置的方式设定，其实，我们可以用窗口的任何一个角来决定窗口位置，先复习一下几何意义的参数设定方式：

width × height <xpos> <ypos>

宽度×高度 <x位置><y位置>

<xpos> 决定了窗口水平的坐标，可用下列方式表示：

+offset ：表示窗口的左边位于距离屏幕左边offset个像素的位置。

-offset ：表示窗口的右边位于距离屏幕右边offset个像素的位置。

<ypos> 决定了窗口垂直的坐标，同样地也可用下列方式表示：

+offset ：表示窗口的上边位于距离屏幕上边offset个像素的位置。

-offset ：表示窗口的下边位于距离屏幕下边offset个像素的位置。

以下有几个例子：

100×100+50+60：这是我们过去用的方式，窗口的左上角位于距离屏幕左边 50个像素，上边60个像素。

100×100-0-0：窗口的右下角位于屏幕的右下角。

100×100-80+160：窗口的右上角位于距离屏幕右边80个像素，屏幕上边160 个像素。

100×100+20-40：窗口的左下角位于距离屏幕左边20个像素，屏幕下边40个像素。

上述例子的正负号代表了窗口的边和屏幕的边的关系，而不是偏移量的正负号，事实上偏移量有它自己的正负号，例如：

100×100+600+-50：窗口位于屏幕的中上方，且窗口的上半部超出屏幕。

100×100--50-+20：窗口位于屏幕的右下角，且窗口的下边距屏幕 20个像素，窗口的右半部超出屏幕。

27.6.7 管理屏幕空间

现在可以启动许多的应用程序，建立许多的窗口，这些窗口很可能会互相重叠。但是你有三种方法可以用来管理你的窗口，使你可以加更方便地存取它们：

• 把窗口缩小，利用前述的Resize选项。

• 把窗口“堆栈”式地排列起来，你现在需要的窗口摆到堆栈最上层，其他的放在较下层，你可以用菜单上的 Raise Lower CircUp 和CircDown来改变堆栈次序。

• 把窗口换成非常小的窗口，称为“图标”，因此所占的屏幕空间极小，但只要需要你随时可还原它们，你可以利用菜单上的NewIconify和AutoIconify选项来办到。

1. 变动堆栈中窗口的次序

窗口在屏幕上，就如同文件在你桌面上，可以互相重叠。

为了让你容易获得你想要的窗口， uwm 允许你：

• 将一个窗口移到堆栈最上层，不管它现在在堆栈的哪个位置。

• 将一个窗口移到堆栈最下层，不管它现在在堆栈的哪个位置。

• 循环堆栈，将所有在堆栈中的窗口移动一层，将最后一层的窗口移到堆栈另一端开头，

你可以向上或向下循环。

(1) 将窗口移到堆栈最上层 —— Raise

Raise 选项将一个窗口移到堆栈最上层，所以这个窗口应该变成全部可见。你可以移动任何窗口而不管它目前在堆栈何处。把一个窗口移动到堆栈的最上层的步骤是：

1) 从菜单中选取 “ Raise ” 选项，光标变成手指状。

2) 将光标移到你想要移动的窗口上。

3) 按任意一个鼠标按钮，窗口保持在原来的位置，但那些原来被其他的窗口遮住的部分均会重现，其他的窗口则被盖在下面。

(2) 将窗口移到堆栈最下层 —— Lower。

Lower 选项可将一个窗口移到堆栈的最下层，你可以移动任何窗口而不管它目前在堆栈何处。把一个窗口移动到堆栈的最下层的步骤是：

1)从菜单中选取 “ Lower ” 选项，光标变成手指状。

2)将光标移到你想要移动的窗口上。

3)按任意一个鼠标按钮，窗口保持在原来的位置，其他原来被它遮住的窗口会显现出来，而它本身的部分则被这些窗口遮住。

(3) 循环堆栈 —— CircUp和CircDown

CircUp和CircDown选项用来旋转堆栈内的窗口，所差别的只是它的 “ 方向 ” 而已。循环向下的步骤为：

从菜单中选取CircDown选项，所有在屏幕上的窗口位置均不变，但原来在最上层的窗口被移至最下层，所有原来被它遮住的窗口现在变成遮住它。

CircUp和上述成对比，它把原来最下层的窗口移至最上层，遮住那些原来遮住它的窗口。

2. 图标化窗口

虽然你可以靠着Raise 或Lower 变动窗口的顺序。但有时窗口实在太多了，为了给你自己更多的屏幕空间，你可以将那些目前不需要的窗口图标化。图标化的意义是把应用窗口换成一个非常小的窗口后摆在一边，直到再次需要用它们为止。有些应用程序拥有它们特别的图标，但是大部分都是让窗口管理器去建一个。 uwm 的缺省图标是一个把应用程序名称摆在中间的灰色长方形。

共有两种方法可以图标化一个窗口，第一种特别适合尚未图标化的窗口，第二种适合曾经图标化的窗口。

(1) 图标化一个新窗口 —— NewIconify

1) 从菜单上选取NewIconify选项，出现 “ 手指状 ” 光标。

2) 将光标移到需要图标化的窗口。

3) 按下鼠标任意钮，保持按住状态，光标变成 “ 十字箭头 ” 形，且出现一个小九字格，这个九字格代表未来的图标。

4) 保持按住按钮，将九字格拖动至你想要的位置。

5) 释放按钮，九字格会被图标替换，原来的窗口消失。

因为NewIconify让你选择图标的位置，所以它适合新的窗口；当然对任何应用窗口均可使用，特别是你想改变图标位置的时候。

(2) 图标化一个 “ 旧 ” 的窗口 —— AutoIconify

AutoIconify 会将图标放在上一次出现的位置，如果这个窗口未曾图标化过，则放在光标所在的位置。

1) 从菜单上选取AutoIconify选项，出现“手指状”光标。

2) 将光标移到需要图标化的窗口。

3) 按任何按钮，原来的窗口消失，图标出现在上一次出现的位置，若这个窗口是第一次图标化，则图标出现在目前光标所在的位置。

(3) 移动图标

一个图标就像一个窗口，因此你可以利用Move选项，像移动窗口一样移动图标。

3. 解除图标化 —— 将图标还原成一个窗口

将图标还原成一个正常的窗口，它的步骤和图标化类似。甚至在菜单上，使用相同的选项，换句话说，AutoIconify和NewIconify这两个选项，如果是在窗口的状况下选择，会变成图标，反之如果是图标的情况下，则会变成窗口。

对于位置的处理也是同样。使用AutoIconify时，当你在图标上按按钮，原来的窗口会在原来的位置出现。如果用NewIconify选项，按住按钮则会出现和原窗口大小相同的九字格，你可以拖动九字格至你要摆放窗口的位置，释放按钮则在选定的位置上出现原来的窗口。

27.6.8 中止应用程序窗口

uwm 菜单有一个选项让你删除一个应用程序窗口，当你决定不再需要或是想要删除一个窗口时，步骤如下：

1）从菜单上选取KillWindow选项，光标变成“手指状”。

2）将光标移到你想要删除的窗口上。

3）按一下鼠标任何按钮，窗口消失，内含的应用程序随之中止执行。

当窗口消失后，你可以在原来下命令的xterm 窗口看到和前面使用xkill后类似的信息。

27.6.9 激活uwm 菜单的其他方式

截至目前为止，我们激活 uwm 菜单唯一的方法就是将光标移到屏幕的背景上且按住鼠标的中按钮，但是如果一个应用窗口占用了整个屏幕，那该怎么办？你会因为找不到屏幕背景而无法激活菜单，那么什么事都不能做了吗？

答案很简单，有另外的办法激活菜单：

1) 同时按下Meta和Shift 键，按住不放。

2) 按住鼠标的中按钮，uwm 菜单即可出现 (你可以现在或稍后放开Meta和Shift键)。

3) 像前几节的方法一样选择选项。

菜单的操作方法和以前一样，只有一点不同：如果你把光标移出菜单的边，菜单不仅是消失而已，一个题为Preferences(首选项) 的菜单出现了，你可以利用这个菜单来设定一些参数。例如键盘被按时会不会有声音，喇叭的音量等等。如果你并不需要设定，将光标移出菜单，或者按鼠标的任一按钮即可离开菜单。

27.7 使用X的网络设备

X的网络特点在于让你可以在网络上的任何机器执行应用程序，而将其输出显示在你自己机器的显示器上。这是X最重要的功能之一，而且很容易使用。

以下将描述你如何指定一个远程终端机，如何实际使用这些功能。最后，我们再描述如何在网络上从其他的机器上控制或限制存取你的显示器。

27.7.1 指定远程终端机——display选项

几乎所有的X程序都接受以一个命令行的选项来指定使用哪一个显示器 (换个说法，连接到哪一个X服务器)，这个选项的格式为：

-display displayname

让我们更进一步讨论显示器名称(displayname)的格式。

你会告诉程序它的输出是哪一个显示器 (网络上任何你可以选择的显示器)。明显地，网络上指定机器的名称一定包含在内。但不止于此，因为一些 (大型)机器可以有好几个I/O 工作站，每一个工作站又拥有自己的键盘，鼠标等等；更进一步，一个工作站还可能控制了好几部终端机。综上所述，显示器名称需要包含三个元素， hostname（主机名），display number（显示器号）和screen number（屏幕号），我们将详细解释并举例说明。

1. 主机名

主机名是在网络上与显示器直接连接的机器名称，主机名也决定了应用程序和服务程序是如何连接的。简单地说：

假使服务程序在你自己本地的机器上执行，你有两种选择：

1) 省略掉主机名，系统会选择最有效率的方式和服务程序交互。

2) 定主机名为“ unix ”，系统将用UNIX域套接字作通信。(“ Unix域 ”意指套接字用传统Unix文件名称 (例如/dev/urgent)来命名。)在命名之后需加一个冒号 (:)， 即使你省略主机名，你仍需要加冒号。

假使服务程序在远程的机器上执行，你一样有两种选择，依你网络上用的通信系统而定：

1) TCP/IP ：大多数的UNIX系统使用此种通信方式。 简单的方法是用在你局域网上已知的普通名称(例如venus或saturn)。你也可以用完全的 Internet名称(例如expo.lcs.mit.edu或它的Internet地址129.89.12.73)。在名称后，需要加一个冒号。

2) DECnet ：用你连接到的机器上的DECnet节点名，在主机名加两个冒号(::)。

2. 显示器号

显示器是一组监视器，屏幕，连接一个键盘和鼠标的逻辑屏幕的组合。换句话说，即是用户工作的地方，在一个给定的CPU 上，显示器从 0开始编号，显示器号即是指哪个编号的显示器被使用，即使显示器号为 0，也不可省略。

3. 屏幕号

对于连接到显示器上数个屏幕也被从 0开始编号，屏幕号为你使用屏幕的编号和显示器号以一个句点 (.)隔开，屏幕号 为 0时可省略，若省略时，其前面的句点一并省略。

4. 例子

以下为一些显示器格式的例子：

• 假设为本地的机器，缺省屏幕为 0，以下二者均可：

```
unix:0
:0
```

• 假设你指定你自己的机器 (通常是venus)，但你需要检验TCP/IP网络的操作和明显地指定屏幕：

```
venus:0.0
```

• TCP/IP网络上，远程的机器名为pluto ， 仅有一个显示器，指定屏幕号为 0：

```
pluto:0.1
```

• DECnet网络上，显示器号为 1，缺省屏幕号 为 0：

vomvx2::1

27.7.2 实际使用远程的显示器

我们已经知道了如何指定远程的显示器，现在来练习一下：假设你是在 venus 工作，想要在saturn上执行一个例如是xterm 的应用程序。你必须在saturn执行xterm 且指定venus 的显示器，则命令如下：（为了清楚起见，下面我们的命令行包含了命令行前外壳程序对机器名称的提示。）

```
venus% xterm -display venus:0.0（注意：不完整！）
```

以上的指令是在本地的机器启动xterm ，并非在远程的机器启动，不符合需求。

如果在你的操作系统上，并未支持远程机器的操作，你可以借助于连接到 saturn的终端机输入下面的命令：

```
saturn% xterm -display venus:0.0 （注意：不完整！）
```

则xterm 会在saturn启动，在venus 上建立窗口，窗口会向 venus 的鼠标和键盘取得输入，这的确是你想要的，现在你可以回到venus 机器开始工作。

但由于你的操作系统事实上支持远程机器的功能，所以你不需要离开你的机器便可完成上述的指定，命令如下：

```
venus% rsh saturn xterm -display venus:0.0
```

以上是利用普通的远程外壳的设备程序 —— rsh。

1. 容易发生的错误

如果你搞混了，一开始发出这样的命令：

```
venus% xterm -display saturn:0.0 (不正确)
```

会发生什么事？假如这命令被接受， xterm 在你本地的机器上执行，而在远程的机器saturn上建立窗口，你在你的屏幕上只能看到外壳读到的命令行，其他什么也没有，系统是正确的工作，但不是你想要的。

如果你很幸运，你可能因没有足够的权限或 saturn上并没有服务器程序在执行，以致无法和saturn上的服务器程序连接上， xterm 会传回一个类似下列的信息而结束：

```
X Toolkit Error: Can ’t open display。
```

现在你就知道有错了。

2. 设定缺省显示器

如果你不明确地指定显示器名称，程序会以UNIX环境变量DISPLAY 来决定使用哪一个显示器，在启动xterm 时，系统会设定这个变量的内容，所以大部分情况下，你什么都不必担心。

如果你远程登录其他的机器，在其间你执行 X的应用程序，并希望回到你自己的机器上显示，那你必需明确地设定DISPLAY 变量，类似下面：

```
venus% rlogin saturn
Last login: Mon Nov 28 20:01:02 on console
  ...(在远程机器上的登录标题)
saturn%  (远程机器上的外壳提示)
saturn%  setenv  DISPLAY  venus:0.0
saturn%  xcalc  &
```

换句话说，如果不设定 DISPLAY 变量，则在saturn上执行的每一个 X程序都必须包含 -display venus:0.0 选项。

27.7.3 控制存取显示器 —— xhost

我们前面提到过有时你无法连接到特定的显示器，通常的原因是你没有相应的权限，所以

X否认你的存取。

X用很简单的结构控制存取：你指定一份可以存取你的显示器的主机名单，在这些主机上执行的应用程序均可存取你的显示器，其他不在名单上的主机则不被允许。你可以用 xhost 程序来控制存取：

允许一或多个机器存取：

```
xhost + host1 [+host2...]
```

去掉允许一或多个机器存取：

```
xhost - host1 [-host2...]
```

所有的机器均被允许存取：

```
xhost +
```

换言之，所有的存取控制均被解除。

恢复存取控制：（通常因为曾经下了xhost + 的命令）

```
xhost -
```

再次取得对存取的控制，只有先前明确地被允许的机器可供存取。

27.8 终端机模拟器——详细介绍xterm

xterm 是终端机模拟器——它是一个可以使X应用程序窗口看起来像普通终端机一样的程序，而这些应用程序无需知道有关窗口系统的功能。我们已经使用过 xterm的一小部分，在下面将更深入地探讨它所提供的特殊额外功能。并且说明许多 X程序共用的一些应用程序界面的角度。

xterm 模拟一个“哑终端机”，但它也提供许多一般终端机没有的功能：

- 设定终端机模式与特性的弹出菜单
- 可以上下移动屏幕图像的滚动条。当文字行因屏幕滚动而消失时，可以将它拉回。
- 模拟 Tektronix 4014 终端机。
- 可选择性地记录屏幕行到一个使用记录文件中。
- 剪贴文字区块。
- 可选择文字颜色，窗口背景等。
- 可选择 VT100 与 Tek 窗口字体。
- 可设定键盘。

我们首先描述选择功能的菜单结构，接着描述如何使用选择功能。

27.8.1 选择 xterm 功能——菜单与命令行选项

xterm 有它自己的内建菜单结构，可在使用期间改变设定。有三个菜单可供利用。

- xterm X11：这里的大多数选择项目为程序控制功能，例如 ：continue program (程序继续或终止程序)。欲下拉此菜单，必需同时按住 Control键与鼠标左按钮。
- modes：设定大多数终端机的特性与选择 Tektronix 模拟功能。欲下拉此菜单，须同时按住 Control键与鼠标中间按钮 (当处于VT102 窗口时)。
- Tektronix：控制 Tektronix 窗口的外表。当处于 Tektronix 窗口时，须同时按住Control键与鼠标中间按钮即可下拉此菜单。

菜单的操作类似 uwm，借助于按鼠标按钮可下拉菜单，不释放按钮移动鼠标指针至想选的项目上；释放按钮后即选定该项。然而，有一点不同的地方是，不能被选择的菜单项目 (因

为此时选择将无意义) 是以较淡的型号显示。例如：因为尚未打开一个 Tektronix 窗口，所以 Hide VT Window 项目的颜色较淡。

许多菜单的功能也能以启动 xterm 的命令行选项来设定。(事实上有某些功能仅能以命令行选项的型号去选择)。下面几节我们将告诉你可以设定不同功能的菜单选择与命令行选项的选择方式。

27.8.2 滚动 xterm屏幕

下拉 xterm X11 菜单并选择 Scrollbar 项目。高亮度显示的部分告诉你两件事：

1) 屏幕上的行数与保存在滚动条缓冲区的行数之比率。

2) 缓冲区的哪个部分目前显示在屏幕上。

你可以利用鼠标按钮移动滚动区的高亮度显示部分，以改变显示在屏幕上的文字。为简化说明，我们假设滚动缓冲区包含 100行文字。

1. 移动滚动条到指定点

假如你想移动文本到某一指定位置，例如，想看第 50 行之后的内容：

1) 移动鼠标指针到滚动条。光标变成垂直双箭头。

2) 按鼠标中间按钮，光标变成水平箭头，且高亮度显示的顶端跳至光标处。例如，假如你想看的部分从 50 行开始，你应该将光标移到滚动区的中央。

3) 假如窗口显示你所要的部分，则可以放开按钮。

4) 保持按住按钮，移动鼠标指针，高亮度显示部分跟随着鼠标指针移动 (而窗口内的文本也随着高亮度显示区而滚动)，直到释放按钮。

2. 向前滚动文本

滚动窗口内的文本使文字行往上移出屏幕顶端 ，高亮度显示区向滚动条底部移动，窗口内并显示最近打入的文本。上卷的步骤如下 ：

1）移动鼠标指针到滚动条。和前面一样光标变成垂直双箭头。

2）按下鼠标左按钮，光标变成向上箭头。

3）放开按钮，与箭头在同一行的文字移到屏幕顶端，且高亮度显示区也随着调整。注意到移动量的多少与你放开按钮时的位置有关 (若接近顶端，你可以获得的移动量小，接近底部则当然可以获得较大的滚动量)。

3. 向后滚动文本

向后滚动窗口文本，文字行由屏幕底部移出，使你可以看见先前打入的文字行。操作程序类似向前滚动，但方向相反，此时使用鼠标右按钮，出现向下箭头 。

4. 其他滚动选项

只要你已经启动滚动功能，有两个方式的菜单的选项可供利用。

- Scroll to bottom on tty output (若有 TTY 输出将输出自动卷到底) 若你目前不在滚动区的底部，稍后某些终端机的输出到达窗口时会自动地移动到滚动区的尾端。此功能为缺省的。若此功能被关闭，你要看最新的输出必须自行滚动窗口。
- Scroll to bottom on key press (按键才卷到底) 若你不在滚动区的底部，稍后你按一个键，窗口会自动移动至滚动区的尾端。此功能不缺省，但通常你的终端机设定成当你键入时回应一个字符，这些字符为TTY 输出，且将引起窗口被卷到底部。

5. 以命令行选项控制滚动

-sb：允许使用滚动条(缺省：禁用disable)。

-sl num : 保存被滚离屏幕的若干行文本(缺省为 64)。

-sk : 启用当按键才滚到底(缺省：禁用)。

-si : 启用当终端机输出时滚到底(缺省：启用)。

27.8.3 记录与终端机的交互过程——写记录

下拉 xterm X11 菜单，并选择 logging 选项(假如你现在是第二次选择同样的菜单，在 logging 选项旁边，你会看到一个沙漏标志，表示它是启动的)。从此以后，所有终端机输出除了被送到屏幕以外，也会被送至一个文件。你可以获得一个使用过程的永久记录。缺省的状况是将输出写到 xtermlog.pid 文件中。其中 pid 为xterm 处理识别码。此文件保存在启动 xterm 时的目录 (你也可以利用下面介绍的命令行选项去改变日志文件名)。

你可以借助于 xterm X11菜单停止或再度记录，反覆的停止和开始记录，你可以作选择性的记录，记录的输出永远附加在日志文件之后，每一次都不会覆写日志文件。

1. 以命令行选项控制登录使用过程

-l : 启用登录使用过程。

-lf file : 将日志文件写入指定文件，以替换缺省文件 (指定日志文件仅设定日志文件名而不启用登录功能; 必需另外使用 -l 来启用登录)使用一个管道作为登录文件。

-lf 选择项有一个特殊功能 ：假如 file 参数以管道记号 (|) 开头，则其余部分视为登录输出的一个管路。例如，假设你的系统外壳提示是 venus%，使用下列命令去启动 xterm 并记录于 cmdlog 文件，只需键入：

```
xterm -l -lf '| grep "^venus% " > cmdlog'
```

27.8.4 剪贴文本

你可以从 xterm 窗口“剪” 部分文本，即拷贝文本到一个“剪切缓冲区”，稍后可将文本“贴”回，即取回。你可以将文本贴回同一个窗口，或任何提供相同结构的窗口。你可以现在或稍后“贴” 回。 但你只有一个缓冲区，所以后来所“剪” 的资料将覆盖掉先前的资料。

1. 剪切

“剪” 一段文字的操作：

1) 移动光标至你想要 “剪” 的那一段文字的一端。

2) 按下鼠标左按钮，并保持按住。

3) 拖动光标至该段文字的另一端 ，在你移动的时候介于开始位置与光标位置间的文字会以高亮显示。

4) 放开按钮，被选到的文字维持高亮度显示，任何先前所选择的高亮度显示文本 (甚至在别的窗口中) 变为非高亮度显示。

2. 粘贴

“贴” 一段文本的操作：

1）移动光标至你想要插入一段文字的位置。

2）按鼠标中间按钮，先前被选定的文字被插入(目前的选择仍保持高亮度显示)。

当你将文本“贴” 入一个窗口，它真的就像你用键盘打入的一样，你可以使用正常的行编辑键去消除字符、单字、或整行 (当然假如“贴”了许多行，你只能编辑最后一行，就像你只能编辑最后敲入的一行一样)。

3. 剪一个字或一行

假如你想剪一个字或一行，你可以直接选择它而不需拖动过它。

- “ 剪 ” 一个字 : 鼠标指针移到一个字的任何位置，并按两次鼠标左按钮，该字即被选择。
- “ 剪 ” 一行 : 鼠标指针移到一行的任何位置，并按三次鼠标左按钮，该行即被选择。

按鼠标按钮两次与三次是所有基于鼠标的系统的共同用法，但在这个例子有特殊的功能。连续而独立的几次按与一个多次按是不同的，其差异取决于按下按钮与释放按钮的期间内有没有其他的事发生。所以下列操作算作按三次 :

按下 ... 暂停几秒 ... 释放 ... 另一个按钮

暂停 ... 释放 ... 暂停

这是很有用的，因为在按后只要你保持按下，只要更进一步借助于使用 up/down，你可以改变选择模式 (字符，字，行)。

4. 扩大选定区块或 “ 剪 ”

只要你有一个选定区块，你可以扩大它 (或缩减它)，如下 :

1) 移动鼠标指针至你想选择的新端点，它可以是在已存在的一个区域里面 (当你想缩减它时) 或外面 (当你想扩大它时)。

2) 按鼠标左按钮 : 选择区的端点调整为目前鼠标指针位置。或若以按下、拖动和放开按钮来替换按。在这个情况下，选择区跟随光标变动，且为高亮度显示。

有一个实用的技巧可以选定文字区块。首先标明你想选定的文本的一端，按鼠标左按钮，然后移至另一端按鼠标右按钮，中间的文字即被选定。 (这是扩大选定区的变相方法。开始时选择区是空的，即你已按鼠标左按钮，但没拖动通过任何文字。然后你按鼠标右按钮来扩大这个空选定区。)

5. 字或文字行边界的选择

假如你想选择一些字或文字行，你可以借助于在按按钮之前小心地定位鼠标指针来完成。但这里有一个简捷的方式 —— 再次利用多次按按钮，选择文字或文字行 :

1）将光标移到你想 “ 剪 ” 的文字的一端。

2）按下鼠标左按钮，并保持按着。

3）拖动光标至你想要 “ 剪 ” 的文字的另一端，开始点至光标间的文本为高亮度显示。

4）放开并迅速连续地再按下按钮，高亮度显示区扩展至最接近单字的边界。

5）放开并迅速连续地再按下按钮，高亮度显示区扩展至已选定行的尾端。

6）放开并迅速连续地再按下按钮，高亮度显示区回复至原来的大小，亦即选定区回到字符边界。

27.8.5 使用 Tektronix 模拟功能

xterm 可以模拟一个 Tektronix 4014 终端机和一个 VT102 终端机，使你可以用它来显示图形。当你在一个远程机器执行非-X应用程序而想在你的显示器上看图时，此功能特别有用。

xterm 为每一个 “ 终端机 ” 使用一个不同的窗口，所以你可以将所有文字显示于一个窗口，而另一个窗口显示图形。但在某个时间只有一个窗口活动着，所以所有的键盘输入或希望 “ 贴 ” 入的文本将被导引至活动窗口，甚至当鼠标指针在别的窗口时也一样。你可以使用终端机的换码序列或使用方式菜单选择你需要的窗口。你可以使用你的窗口管理器完全分开地处理两个窗口。例如，你可以图标化 VT 窗口，然而保留一个打开的 Tek窗口等。你也可以使用适当的 xterm菜单选项 (Tek Window Showing、Hide VT Window等) 去隐藏或显现一个窗口。

1. Tektronix 的特殊功能

Tektronix 菜单 (同时按下 Control键与鼠标中间按钮可得到)。它提供一些类似方式控制 xterm 窗口的功能。但它仅提供用于 Tektronix 窗口的项目。

- 改变字符的大小 ：你可以从四个不同的大小选择，范围从大字符 (缺省值) 到小字符。你可以在任何时刻改变它，甚至在一行中间。在改变之前已出现在屏幕上的字符不受影响。
- 清除屏幕 ：Tektronix 的一个特性是它的屏幕不滚动。在屏幕上有两列 (左与右) 为文本，当其中之一已写满，输出切换到另一端。然而，已显示的字符不清除，因此屏幕不久会变混乱，除非你下命令清除它。想这样做的话需于 Tektronix 菜单选择 PAGE : 屏幕会被清除，且光标被移至左上角。
- 重置“ 终端机 ”：在Tektronix 菜单选择 RESET。字符的大小与线条的型号 (可能因一个程序输出至窗口而改变) 被设回缺省值，同时清除屏幕。
- 将窗口内容复制至一个文件 ：在Tektronix 菜单选择 COPY，自从最近一次的PAGE功能后任何写到屏幕的内容都会被复制到名为 COPYyy-mm-dd-hh.mm.ss 的文件。yy-mm-dd-hh.mm.ss为当时时刻。该文件被保存到启动 xterm 时的目录下。

注意 重绘 Tek 窗口会花一些时间，当它重绘时，Tek 窗口内的光标变成一个闹钟。

27.8.6 使用不同的字体

xterm 可以让你从正常的文字与粗体文字选择不同的字体。字体选择必须有固定的宽度且彼此大小相同。你目前尚不知道如何找到可以利用的字体，但它的应用范围很广。 下面的例子我们将只用其中的两种字体，这两种字体是 core 版本提供的字体的一部分：

- 8x13 (一个字符大小为 8 像素宽，13 像素高)。
- 8x13b (一个粗体变体)。

欲指定特殊字体必须使用命令行选项 :

- -fn font : 使用 font 的正常字体，替换缺省的字体。
- -fb font : 使用 font 的粗体字体，替换缺省的字体; 缺省状态下，xterm 不区分粗体字的文本。

27.8.7 使用颜色

假如你有彩色显示器，你可以用命令行选项设定一组 (些) 窗口元素去指定颜色：

- -fg colour : 以 colour 颜色输出前景，亦即文字。
- -bg colour : 以 colour 颜色作窗口背景。
- -bd colour : 以 colour 颜色画窗口边界。
- -ms colour : 以 colour 颜色为鼠标指针颜色。
- -cr colour : 以 colour 颜色为光标颜色。

参照连接在网络上的机器对窗口设定的颜色码，你可以发现非常有用。设定鼠标与光标为显眼的颜色也是有帮助的，使你在纷杂的窗口中较容易看得到它们。

27.8.8 其他 xterm 选项

Xterm可以接受许多其他的选项。有些是设定终端机的特性，例如 :-display与 -geometry，这在前面已经讨论过。所有的这些都在xterm 联机帮助中有具体的描述，但下面是一些有用的杂项:

- -iconic : xterm 应该以图标启动的方式替换由正常方式打开窗口。 (当使用uwm 为你的窗

口管理器，图标的初始位置将决定于图标被产生当时的光标位置。我们将在“定义应用程序的缺省选项 —— Resources”里教你如何明确地指定一个图标位置。)

- -title string：使用 string 为窗口标题头，且某些窗口管理器可能将它包含在窗口标题条中。
- -C：这个窗口应该将接收的输出送到系统控制台 (例如，磁盘已满信息，设备错误等)。若你没有一个窗口具有这个选项指定，控制台信息可能直接出现在你的屏幕 (亦即不在一个固定窗口中) 并扰乱显示。若这种情况发生时，只要使用 uwm 的菜单选择 RefreshScreen 来恢复正常显示即可。
- -e prog [args]：在窗口中执行具有选择性参数的 prog 程序，替换启动一个外壳程序(此选项必须在命令行的最后，所有在它后面的部分都将视为 args 的部分)。你经常需要使用 -e 去远程登录到一个不支持X的远程系统，例如：

```
xterm -title saturn -e rlogin saturn -l root
```

27.8.9 设定终端机键盘

X 本身可让你改变键盘对照表，所以你可以针对不同的情况改变它以适合一个国家的使用习惯。但这个对照表仅决定那个“字符码”联结到那个给定的键。客户程序则可指定任意的字符串给任何键或键组 (组合键)。使用这个结构你可以设定一个 xterm。这个功能特别适用于邮件程序，或排错程序。你可以指定一般命令给功能键，或控制字符，甚至单一字符。

第28章　实用程序和工具

本节介绍一些小程序。利用这些程序可以更加有效地使用系统并使你的工作更实用。

28.1　实用程序

本节所描述的一些程序功能虽不太重要，但是却可以使你工作得更实用轻松。有些我们曾经提过，如xclock 与 xcalc。但在这里将更加系统地描述这些功能。我们将看一些可以显示目前机器负载和告诉你有邮件送达等功能的新程序。

1. 一个模拟或数字时钟 —— xclock

xclock有下列命令行选项：

- 指定窗口初始大小与位置：用 -geometry geomspec。
- 设定背景色：用 -bg colour。
- 设定前景色：用 -fg colour。
- 设定指针的颜色：用 -hd colour。
- 指针边缘高亮度显示：用 -hl colour，以高亮度显示 colour 颜色为指针的边缘。
- 使用数字时钟：-digital，告诉 xclock 使用一个 24 小时的数字时钟，以替换缺省的模拟型始终。
- 设定“时钟-滴答”频率：-update num，使时钟每 num 秒更新显示一次。处于指针状态下，若 num 小于 30 秒则以一个钻石形秒针每num 秒移动一次。(缺省值为 60 秒)
- 设定半小时钟声：-chime，每半小时整使终端机铃响一次，每一个小时整铃响两次。

2. 桌上型计算器 —— xcalc

xcalc的命令行选项如下：

(1) 指定窗口初始大小与位置命令：-geometry geomspec。

设定桌面颜色的选项为：

- 设定背景色: -bg colour。
- 设定前景色: -fg colour。

(2) 其他选项:

- 指定计算尺模式: -analog，执行程序模拟一个计算尺替换电子式计算器。这实在很少用，但一些更进一步的指令包含在下面，因为它不包含在联机帮助内。
- 指定 HP-10C 型计算器：-rpn (代表逆波兰记数法)，告诉 xcalc 模拟 HP-10C 型计算器。

可以使用鼠标按钮操作计算尺。当鼠标指针进入窗口内，鼠标指针变成手型光标，用鼠标按扭操作计算尺有如下操作：

- 定位滑尺 - 左端：移动鼠标指针进入滑尺区，定位至你想要的地方，按左按钮，滑尺左端跳至鼠标指针位置。
- 定位滑尺 - 右端：与设定左端一样，但按右按钮。
- 定位计算尺的光标：定位鼠标指针至你想要的尺体位置，并按左按钮：尺的光标跳至鼠标指针位置。
- 滑动滑动条: 定位光标至滑尺，按下中间按钮，并保持按着。拖动滑尺到你要的位置，然

后释放按钮。

• 将尺的长度变为两倍: 在尺体上按中间按钮。

• 将尺的长度变为一半: 在尺体上按右按钮。

3. 显示机器的负载平均 —— xload

xload 可以显示系统平均负载的柱状图，它会定期地更新。

4. 邮件通知程序 —— xbiff

xbiff 是一个监视邮件文件并通知何时有邮件到达的小程序。它显示一个邮箱的图案。当没有新邮件时邮箱的旗子是向下的，当有新邮件进来时， xbiff 响铃，竖起旗子，并使图案变成反相显示 。

你可以在 xbiff 窗口上按任何鼠标按钮，强迫旗子放下。

xbiff 接受一般对颜色的命令行选项 (-bg、-fg、-bd) 与对窗口特性的命令行选项(-display、-geometry、-bw)。其他选项有：

• 指定检查邮件频率 ：-update num，告诉 xbiff 每 num 秒检查邮箱一次，看是否有新的邮件到达(缺省值为 60 秒)。

• 指定一个特定邮件文件 ：-file filename，使 xbiff 检查在 filename 内的邮件，替换缺省名称的文件。缺省名称为 /usr/spool/mail/username，其中username为你的登录名称。

xbiff 特有的 -file 选项，对网络上的邮件集中处理和你的邮箱由另外的机器处理等两方面非常有用。下面的命令让你执行某个邮件机器 (比如说 mars) 上的 xbiff程序，查看你的邮箱，并将显示送回你自己的机器 venus：

```
rsh mars xbiff -file /var/spool/mail/smith -display venus:0 &
```

28.2 保存、显示和打印屏幕图像

X是一个图形系统，你可以利用它在窗口内显示文字与图形。 X窗口的用户会经常想要捕捉屏幕上的某些图像，以便稍后可以重新显示它，或送至硬拷贝设备打印。下面将描述这些功能。

1. 保存窗口的图像 —— xwd

xwd 把窗口的图像保存到一个文件中。这个文件稍后可以被其他程序处理 (例如打印)。

有许多种不同的使用 xwd的方法。你可以明确地指定一个输出文件 (使用命令行选项-out name)，或者使用 xwd 打印图像到标准输出。你也可以明确地指定想打印的窗口，或可以让 xwd 提示你一下。

让我们举最简单的一个例子：在 xterm 下使用下列命令启动程序

```
xwd > outfile
```

只要 xwd 启动，光标便会变成“ 十字线 ”状。移动鼠标指针进入你想要打印的窗口，并按任何按钮 ：xwd 响一次铃表示它已经开始记录窗口图像，且在它完成时响两次铃。然后光标恢复正常。

2. 如何指定被打印的窗口

有两种方法可以通过命令行选项告诉 xwd 要打印哪一个窗口：

• -root : 打印根窗口。例如，想得到屏幕的一个完整图片，使用命令 ：

```
xwd -root > screenpic
```

• -id win-id : 打印窗口识别码为 win-id 的窗口。(每个由X服务程序建立的窗口皆有一个唯一的窗口识别码 —— 它只是一个识别窗口的号码。)

使用这些选项的好处是你不必使用鼠标去指定你感兴趣的窗口。因为有时候在打印的同时，鼠标必须出现在某个特定的地方执行某项特定的功能，例如：呼叫一个下拉式菜单，或使一个命令按钮出现你想要的状态。举例来说：欲保存屏幕图像，必须用鼠标下拉 uwn 菜单，因而造成你没有办法同时使用鼠标去指定 xwd 的目标。下面是捕捉图像的步骤：

1) 在一个 xterm 窗口，键入命令

```
sleep 10.xwd -root > uwmfile
```

在捕捉图像之前，给你自己时间使屏幕进入到需要的状态。

2) 移动鼠标指针至背景窗口上。

3) 按下鼠标中间按钮，出现 WindoeOps 菜单，并保持按下状态。

4) 等一下，直到 xwd 响铃一次告诉你开始，完成则响铃二次。然后释放按钮。

2. 放大屏幕窗口图像 —— Xmag

xmag 可以快拍任何屏幕的图像，并以任意倍数放大它们。最简单的方法为采用交互方式指定你感兴趣的区域：

1) 启动程序 (用 xmag 命令) 。xmag 显示一个闪动的矩形外框。

2) 将矩形框放置到想要的地方。

3) 按任何按钮。xmag 快拍矩形框内的区域，并显示一个清新的窗口外框，里面正显示放大的快拍图像。

4) 仿照启动 xclock 后所采用的方法一样使用鼠标指针与按钮放置外框。例如，按鼠标左按钮将窗口放置在鼠标指针目前所在的位置。

5) xmag 在你刚放置的窗口内，重绘放大的图像。

你现在可以选择如下动作之一：

- 按 Q 或 q 或 Ctrl-C，接着指定 xmag 窗口内的选项，跳出 xmag。
- 按鼠标中间按钮或左按钮以移开目前放大的窗口， xmag 再次显示出它的闪烁矩形，所以你可以放大屏幕上另一个区域。
- 按下鼠标左按钮 ：xmag 显示出在鼠标指针下的像素坐标，即像素的号码 (它是一个该像素颜色的内部表示法)，以及像素的 RGB 值，亦即像素颜色的红，绿，蓝成分。当你移动鼠标指针，这些显示随之更新，直到你释放按钮。

程序记录快拍图像只是为了立即再显示 —— 没有任何方法可以取得它以便打印图像至一个文件。

3. xmag 的命令行选项

缺省的操作方法有一些限制，你必须以交互方式指定被放大的区域，区域的形状与大小被固定，放大率都是 5。但事实上你可以使用命令行选项改变上述所有的特性：

- 指定不同的放大率：-mag num，放大图像 num 倍 (num 须为整数)。例如： -mag 2，将产生一个为原来两倍宽与两倍高的图像。
- 指定大小与放大区域的位置：使用 -source geomspec。例如：

```
-source 300x100 + 450 + 762
```

xmag 不提示任何信息，但将直接跳到它显示放大窗口外框的步骤，假如 geomspec 只有位置部分，则大小部分缺省为 64×64。

- 仅指定放大区域的大小 ：使用 -source geomspec、geomspec 仅由大小部分组成，就像 -source 300×100

xmag 将显示一个闪烁的 300×100 的矩形外框，等你定位与按按钮。

4. 保存一部分屏幕的图像

xwd 仅对单一的完整窗口操作。假如想捕捉一个窗口的某部分或某些窗口，你必须采用两个阶段处理：

1) 使用 xmag 程序选择你要的区域，并在一个单一窗口 (亦即 xmag 本身) 显示它。

2) 用 xwd 打印 xmag 窗口至一个文件。假如你想打印与最初一样大小的图像，别忘了给xmg指定 -mag 1。

5. 显示先前打印的图像 —— xwud

xwud “反转储”一个先前曾经被打印至一个文件的图像，也就是再次将图像显示于屏幕上。缺省的作法是它从标准输入读入打印文件，所以反打印一个你想要的文件如下：

```
xwud < screenpic
```

xwud 显示闪烁的窗口外框，准备让你使用窗口管理器。当你已放置好它，图像即被显示。当然你可以利用窗口管理器移动窗口、重定大小、图标化等，就如同处理其他窗口一样。

注意 在联机帮助内提到的 -inverse 可能无法正常运作。

6. 打印先前打印的图像 —— xpr

xpr 能够把一个先前曾打印的图像转化为可送至硬拷贝机打印的格式。它支持各种打印机 —— PostScript、DEC LN03、LA100以及 IBM PP3812。由于缺省是写到标准输出，所以典型的用法应该是：

```
xpr -device ln03 < screenpic | lpr
```

xpr 接受数个选项，包括控制图像在纸上的位置，大小与配置和指定输出的打印机型号：

- 指定打印机型号：使用 -device type，这里的 type 是指 ln03、la100、ps (PostScript) 或 pp (指 PP3812)。
- 控制图像的大小：由于缺省 xpr 以最大的尺寸打印图像，你可以用 -width num 或 -height num 指定最大的宽度或高度，num 的单位为 inch (不必是整数)。
- 对图像标刻度：你可以使用 -scale num 对图像标刻度，num 为整数。作法与 xmag 的 -mag 选项类似，但大小可能扭曲，因为打印机可能有不同的分辨率，亦即每个 inch 有不同的点 (像素) 数，例如 -scale 2 表示每个屏幕像素被打印机印成 2×2 点的方形，所以假如你的屏幕为 75 dpi 且你的打印机为 300 dpi，打印图像仅为屏幕图像大小的一半 (75×2/300 = 0.5)。
- 加图像标题：使用 -header string 或 -trailer string 分别将一个文字字符串印在图像的上面或下面。

7. 利用单一操作命令打印与打印一个窗口 —— xdpr

xdpr 是一个使用 xwd 打印窗口图像，接着使用 xpr 格式化打印图像，最后使用 lpr 输出图像的一个程序。它接受所有这三种程序的命令行选项 。事实上 xdpr是一个shell脚本，它将上述三种程序包装在一起以便使用。它真正的工作为检查命令行上的各个选项且传送它们给适当的程序。

28.3 使用X的应用程序

本节将介绍一些包含在 core 版本内的实用程序，它们的主要功能和窗口系统并无密切关系，但使用它们却可提供一个相当不错的用户界面，这些应用程序如下：

- Xedit —— 一个以窗口为基础的一般目的文字编辑器。
- Xman —— 一个联机帮助或系统文件的浏览工具。

• Xmh —— 一个 mh 邮件处理程序的用户窗口。

28.3.1 文字编辑器 —— Xedit

Xedit 是一个非常简单而具有窗口界面的文字编辑器。对 Xedit 所显示的选择框按按钮，你可以完成某些操作。但通过使用键盘你可以执行的更多操作，特别是使用控制字符。在这个程序内大多数以键盘为基础的功能由一个标准软件提供。

1. 启动与结束程序

假如你想编辑一个名叫 foobar的 文件，在 xterm 窗口键入命令：

```
xedit foobar &
```

(假如你不想编辑已存在的文件，只要键入“ xedit & ”即可。)

窗口分成三部分：

• 上面是命令菜单，具有 Quit、Save 等命令。

• 中间部分是信息窗口， xedit 在里面显示错误信息与状态，你也能把它当作一个抓取区来使用，可在里面键入一段你想剪贴到其他地方的文字。

• 下面较大的部分是编辑窗口，它显示你正在编辑或建立的文本。

当你想结束程序时，在Quit命令上按左按钮，假如你做了一些改变但未保存，则 xedit会在信息窗口内输出警告信息：

```
Unsaved changes，Save them，or press Quit again.
```

2. 插入文字

首先确定鼠标指针在编辑窗口内，然后键入你要的文字 (键盘上任意的打印字符)，在你键入时，文字被直接地插入到窗口中。当键入时，你会注意到新插入的文字是不断地推送一个在它前面的一个小脱字符 (^) 光标，这个光标是插入点或简称为点。任何你键入的或贴入的文字都将在这个点的位置插入。插入点总是位于两个字符间，而不在一个字符上面 (就像一个正常终端机光标的状况)。

在接近一行的尾端插入文字时，假如键入的字太长以致无法在这行放下时， xedit 将自动移至下一行。假如剪短该字使得可以适合它原来的行，它将再跳回上一行。

这个作用与 xedit 的段落定义息息相关 ：一个段落的终结是换行字符。折行仅作用于一个段落内，且能正确地处理特殊状况，例如：在移动一个单字到这行时造成这行必须折到下一行的状况等等。

(1) 特殊插入操作

下面有一些插入新行的特殊命令，各种命令稍为有些不同：

• 插入一个新行，且插入点移至新行 ：按 Return键便可以插入一个新行，事实上就像插入任何打印字符一样。且假如你不使用 xedit 的折行，这是移至新行的正常方法。

• 插入新行，且对齐 ：按 LINEFEED键插入一新行并移动插入点到下一行，但任何你键入的打印文字将与上一行的文字对齐(对键入表格等很有用)。

• 插入一新行，但不移动插入点 ：按 Ctrl-O。可以插入一个新行，但点仍留在同一行 (当你想分开一行并加文字到被分成二行中的第一行尾端时使用)。

(2) 剪切与粘贴

就像在 xterm 中一样的，你可以选择并 “ 剪 ” 出一段文字。但是，有一个讨厌的问题，当选择字或行时，你必须迅速按二次或三次按钮 —— 你不能在按钮按下或释放时暂停。

在插入点的地方贴入文字时，你可以像以前一样的按中间按钮，或按 Meta-Y键。

3.移动插入点

移动插入点最简单的方法是使用鼠标，移动窗口文字光标到任何你想要的地方，并按鼠标左按钮。插入点的 ^ 标志即跳到新位置。

但通常使用键盘移动插入点也很简单。当你正在编辑时，它会打断你的节奏，因为必须拿起鼠标，移动它，最后再次将你的手移回键盘。 xedit 提供一次移动一个字符、单字、行或页的方式移动插入点，方法如下。

(1) 一次一字符移动插入点

这是移动点的最基本方式:

• 向前一个字符：按 Ctrl-f 或右箭头。
• 向后一个字符：按 Ctrl-b 或左箭头。
• 向上一个字符：按 Ctrl-p 或上箭头。
• 向下一个字符：按 Ctrl-n 或下箭头。

假如你接近一行的开头，按几次 Ctrl-b 后，你将回到前一行。按几次Ctrl-f 后可回复至原位。假如你是在一页的最上面 (最下面) 一行，类似的效果会发生。移至前 (或下)一行，将引起文字滚动使你移至的行能被看得见。

(2) 一次移动一个单字、一行或一个段落

这些是编辑文字时所须的自然移动功能。

• 向前一个单字：按 meta-F 或 meta-f。
• 向后一个单字：按 meta-B 或 meta-b。
• 移至行尾：按 Ctrl-E。
• 移到行的开头：按 Ctrl-A。
• 向前一个段落：按 meta-]。
• 向后一个段落：按 meta-[。

你可以看到一个规律 ：Ctrl-char对单个字符操作。而 meta-char对单字作相同的事。例如 Ctrl-F 向前一个字符， meta-F 向前一个字。

(3) 大量移动 —— 以页与文件为单位

假如你想获得较大的跳越，就像在 xterm 里面一样，你可以使用滚动条。然而，滚动文字并不会移动插入点。当你键入或清除任何内容时，文字将自动地卷回插入点，让你可以看到你作了什么。

如果你想滚动显示内容且移动插入点，方法如下：

• 往前一页：按 Ctrl-V。
• 往后一页：按 meta-V。
• 到文件的开头：按 meta->。
• 到文件的尾端：按 meta-<。

4. 删除不想要的文字

有两种删除文字的技巧，删除只将文字移除，销毁则除了移除文字外，还将文字保存至“ 剪切缓冲区 ”，以便稍后你可以取回。销毁命令仅对较大单位的文字作用 (最小单位为单字)。因为假如你删除一个字符，它就像没有销毁它一样很容易再打入。

在下面的叙述中，“ 下一个 ”意思是正好在插入点之后，“ 前一个 ”意思是正好在插入点之前。

(1) 删除字符

• 删除前一个字符：按 Delete 或 Backspace 或 Cth-H。
• 删除下一个字符：按 Ctl-D。

假如你在一行的开头按下 Delete 键，它将删除前一行的尾端的新行字符，而将两行合并成一行。

(2) 删除文字

• 删除下一个单字：按 Meta-D。
• 删除前一个单字：按 Meta-H 或 Shift-Meta-Delete Shift-Meta-Backspace。
• 从光标位置删除到行尾：按 Ctrl-K。
• 从光标位置删除到段尾：按 Meta-k。
• 删除目前选定的文字区块：按 Ctrl-W。

(3) 救回删除的文字; 拷贝与移动文字

只要你曾经销毁某些文字，你就能按 Ctrl-Y 救回它，最近被删除的文字在插入点的地方被插入。但有几点须注意：

• 只有最近被删除的文字可以被救回。你无法将一连串被删除的文字取回。
• 救回的文字是被插入到目前插入点的位置，不是该段文字原先被移除的地方。
• 若是需要的话你可以救回相同的文字许多次，即按 Ctrl-Y 并不影响缓冲区的内容。

基于“ 删除 ”动作的原理，你可以用它来移动或拷贝文字区域。

• 移动文字：先销毁它，再将点移动到你想重新放置的地方，最后再救回。
• 拷贝文字：先销毁它并救回它，使拷贝的来源不变; 再将点移动到你想拷贝的地方，最后再一次救回。

5. 撤消改变

假如你作了某些改变(删除或删除或键入或贴入文字)，但稍后发现并不是想要的，你可以撤消它。在undo 框里按左键，取消最近的改变。 undo本身也是一个改变，所以假如你再次按 undo，它将取消前一个 undo的作用; 你可以永远地像这样一直切换下去。

undo 仅对最近的改变有作用。假如你想更往回追溯，你可以利用 more 框连续地撤消更前面的改变。

6. 使用文件

前面曾提到在启动程序时你可以指定欲编辑的文件，事实上当程序执行时，你也能抓取文件。为达到这个目的你将会使用介于 load 与 undo间的文字框，称为“ 文件名框 ”。

• 要把文字保存到一个文件，在 save 框上按左按钮; xedit 会将编辑窗口内的文字存到文件名为文件名框内文件名的文件; 若没有名称， xedit会在信息窗内发出 save : no filename specified —— noting saved因此在你再次在 save 上按左按钮之前你必须将鼠标指针移到文件名框并键入文件名。当它已经保存好文件 xedit 会发出确认信息。
• 编辑一个不同文件，在文件名框键入文件名，并在 load 上按左按钮。若它无法取用文件，xedit 会发出错误信息。
• 再目前的文本处插入一个文件: 按下 meta-I，xedit 下拉一个小窗口。在上端的文字框，键入你想插入的文件名，并在 DoIt上按左按钮。该文件的内容即插入目前的插入点位置。

7. 搜索指定的文字字符串

假如你想在你编辑文本中找出一个指定字符串出现的地方：

1) 在我们称为搜索字符串框的里面，在Search >> 的右方键入字符串。
2) 在 Search >> 上按左按钮：插入点便会移至文本中下一个出现该字符串的地方。

搜索是由插入点开始，并且 xedit 缺省的搜索方向是向前，而不往后搜索。假如你要从插入点往回搜索，你可以在 << 上按左按钮。

8. 替换字符串

假如你想将出现许多次的一个字符串 (旧) 置换成为另一个字符串 (新)：

1) 先搜索出文本中第一次出现欲替换字符串的地方，如前一小节所诉述。

2) 在 all 框右方的替换字符串框内键入新字符串。

3) 在 replace 上按左按钮，旧字符串即被替换成新字符串，且插入点移到下一个出现旧字符串的地方。

4）假如你也要替换它，再次在 replace 上按。假如你不想替换这个，但想改变它后面的，在 search >> 上按按钮直到你要改变的地方，即可以再次替换它。

在你往回移动文件时你无法执行替换，换言之你无法用简单的方法替换在插入点之前的字符串。

9. 杂项功能

- 重绘文本显示：按 Ctrl-L。
- 向前滚动一行：按 Ctrl-Z。
- 往回滚动一行：按 meta-Z。
- 跳至指定的行数 ：在信息窗口内，键入你要跳往的行数，用鼠标选择你正在打的文本，并在 jump 上按左按钮， 插入点便会跳至指定行的开头。假如在屏幕上有应用程序的文字片段含有你要指定的行数数字，你可以利用它，而不必依赖 xedit 编辑信息窗口。

28.3.2 邮件/信息处理系统——xmh

xmh 是一个架在 mh 邮件/信息处理系统之上的X界面。当你启动它时， xmh会建立一个图像的窗口。程序的窗口相关画面如编辑文本与管理窗口方框，均与 xedit 及 xman十分类似，事实上是由相同的内部结构所提供的。正因为这样，且由于大多数描述是与程序的邮件功能有关而较少与X有关，我们不进一步讨论。假如你想知道如何使用程序，联机帮助中有一个简单但内容丰富的描述。

28.4 示例和游戏程序

MIT 发行的 core 版本提供了少数的示例程序和仅有的一个游戏程序。它们展现出某些窗口系统的威力，且能给人一种强烈的美好印象——特别是在彩色屏幕上。

28.4.1 找出通过随机迷宫的路径——maze

maze 在窗口中产生一个随机的迷宫，它会自动找出从入口通过迷宫到达出口的路径。它会追踪它走过的轨迹，当它从一个死巷中退出时则将轨迹消除。你可以用鼠标按钮启动、暂停、继续或停止程序，就如同在联机帮助中所描述的。 maze不提供颜色。

注意 中间按钮对暂停与重新启动的作用并不可靠。

28.4.2 担任鼠标指针的大眼睛——xeyes

xeyes 在窗口中绘出两个大眼睛，且它们永远看着鼠标指针。当鼠标指针移动时眼睛也随着调整，甚至当移动鼠标指针到两眼之间，它们会变成斗鸡眼！

你可以明确地给窗口的每个元素(瞳孔、背景、外框等)设定颜色。

注意 xeyes 会使你的系统执行速度变慢。

28.4.3 智慧盘游戏——puzzle

puzzle 是一种古老的游戏，有 15 个编号的小方块被一个 4×4 框架围住。你必须移动小方块使它们按照数字顺序排列。

你可以使用鼠标控制游戏：

• 启动游戏：在控制条的左上边框内按按钮可以重新打乱小方块。

• 移动小方块：将鼠标指针移到与空白位置相同的行或列的小方块上，按按钮以移动该小方块与它之前的所有小方块进入空的位置(所以移动后，空位置在你按按钮的地方)。

• 由 puzzle 自己去解：在控制条的右手边框上按按钮。

• 离开 (跳出)：在控制条的中间按中间按钮。

puzzle 的命令行选项

• 使用一个大小不是 4×4 的框架，使用选项 - size width x height，其中尺寸以小方块为单位。

• 改变小方块被移动的速率，使用选项 -speed num。此处 num 是每秒移动的数目 (缺省值为 5)。

28.4.4 打印一个大X标志——xlogo

xlogo 建立一个窗口并在它里面显示一个 X标志。假如你重定窗口大小，标志再次绘出，且尽可能地将窗口填满。

28.4.5 跳动的多面体——ico

ico 产生一个窗口且在它里面有一个 12 面体——一个具有 12 面的实心体。这个多面体是会动的，它在窗口内不断地碰撞跳跃。在单色显示器上你可能只有一个直线构成的图 (使用选项 -i 可以反白显示)，但是，在彩色显示屏幕上，你可以看到实心的彩色面。

试试下面的例子：

```
ico -nodeges -faces -colors red blue yellow green
```

你可以在一个根窗口内 (背景窗口)设定一个跳跃的12面体，而不必靠它自己使用 -r 选项指定特定窗口。另外，ico 可指定多面体的面数，不是只有 12面体。如果你想得到一个完整的列表可以键入下列命令：

```
ico -objhelp
```

28.4.6 动态几何图案——muncher 与 plaid

muncher 与 plaid 重复地绘出变化多端而有趣的几何图案。

28.7 显示信息和状态的程序

下面，我们将讨论一些提供窗口系统本身信息和目前状态的程序。包括检查系统上窗口各种属性的工具和一个观察X事件结构是如何工作的程序。这些程序有下列用途：

• 当你使用系统时，这些工具程序所提供的有关系统内部组织和操作的信息，可以帮助你

了解发生了什么事。

- 当你要做某种处理时，可以借助所给的信息来确认系统的组件。例如你必须知道一个窗口的window-id，才可以用xwd 来打印它。
- 可把从这些工具获得的信息，当成定制系统工具程序的输入值。

28.7.1 列出X服务程序的特征——xdpyinfo

xdpyinfo列出有关X服务程序和服务程序所控制屏幕的各种项目的信息。

28.7.2 获取有关窗口的信息

有三个程序可以提供目前在你显示器上窗口的信息。它们从不同的角度看系统：

1）打印窗口的层次——xlswins。

2）对单一窗口详细的信息——xwininfo。

3）列出窗口的属性——xprop。

1. 打印窗口的层次 ——xlswins

在X系统上的窗口被安排成树状的层次，根窗口（也叫背景窗口）在最上层，在其下才是应用窗口，每一个应用窗口可以拥有它自己的子窗口层次。

xlswins 打输出这个树状结构，从根窗口或所指定的窗口起至其下所有的树状结构，对每一个窗口，xlswins 列出窗口的window-id 并用括号括住它的名称（如果有的话），子窗口则在下面的几行依序以缩进两格的方式列出。下面对 xman应用程序列出子树输出命令，以演示xlswins如何剖析系统的结构。

```
xwd -id 0x60005d | xwud
```

注意 并非所有的窗口都可打印输出信息，如果你碰到这种无信息可供打印窗口，将获得类似下面的信息：

```
x Protocol error: BadMatch, invalid parameter attributes
Major opcode of failed request: 73 (X_GetImage)
```

如果在已经有好几个应用程序的系统上，你将可以看到uwm 和xterm 菜单相关的子树，或者与 xmh和 xedit 的命令按钮相关的子树。

2. 关于单一窗口的详细信息——xwininfo

xwininfo能够针对特定的窗口给你大量的信息，你可以借助于命令行的选项告诉 xwininfo你要哪一部分的信息。

用和xwd 相同的方法来指定你感兴趣的窗口：

- 交互式（缺省）：开始时，xwininfo给你一个十字线光标，你可以将它移到你所需的窗口再按鼠标按钮。
- 使用命令行选项：你可以用选项-root 指定根窗口，或用选项-id window-id 指定其他的窗口。

窗口信息可分为以下几类：

- 窗口的window-id ：window-id 是窗口系统识别每一个窗口的参考，就如同先前所看到的，几个程序（包含 xwininfo它本身）给一个数字代表 window-id 。例如要打印一个窗口，可以先执行xwininfo得到它的window-id，然后将此window-id 做为xwd 的id选项之参数。
- 层次的信息：可以看到这个窗口的父窗口的 id，几个子窗口的id，以及根窗口的id，在

xlswins 中可得到相同的信息，但在这里只能得到最近一层子窗口的id，而不是整个子树。

- 几何细节：窗口的大小和位置，以及它的四个角的位置。
- 和服务程序有关的重配置参数：像“ gravity ”和“ backing store ”这些状态，当窗口改变大小或从被遮盖的状态下重新显露出来时，服务程序需要用到这些参数，这些参数你自己不会用到，但可参考它了解系统如何运作。
- 事件参数：这些参数也是给服务程序而非给用户用的。
- 窗口管理器信息：我们曾经提过应用程序借助于给窗口管理器一些提示来完成通信（这些提示包括应用程序所希望窗口的大小，以及重定大小时的限制等）。xwininfo在这个部分的输出便是告诉你这些“ 提示 ”的资料。 Program supplied location 为应用程序建议它自己应该摆哪里。如果你曾给过位置，不论是在命令行或源文件，都会在 User supplied location 出现，在窗口大小方面同理可推， resize increments 解释了为什么有些窗口（例如xterm 和xfd ）不能把大小定为任意数目的像素，因为应用程序已经告诉了窗口管理器在重定大小时按多少个像素的倍数放大或缩小（ xterm 和xfd 它们的大小和所使用的字体有关），你也可以由这个参数知道目前窗口的位置，所以稍后你可以在同一位置上重建它。

3. 列出窗口的属性——xprop

“ 属性 ”是指一小段有关窗口的数据， xprop 可列出一个指定窗口的所有“ 属性 ”，也可以打印一个字体的属性。可以经由常用的方式来选择窗口（按鼠标按钮或使用 -root 或-id 选项），如果是指定字体，则用选项-font fontname。

显示出来的格式为：对每一个属性，均有一个属性名称，在其后用小括号括住的为属性的类型或格式，最后则为属性的值。大部分你所看到的属性类型为 STRING，属性的值用“ ”括起来，其他的属性类型的格式是专用的，从属性的值很容易了解它的意义，对字体显示的格式稍有不同，它没有属性类型，但属性值的意义也很明显。

注意 xprop 的输出相当的复杂，我们并不需要了解其所有的内容，视需要而定。

以下让我们看看从应用窗口、根窗口、字体所获得不同的输出：

其他的属性如下：

- WM_COMMAND：执行启动这个应用程序的命令行，被切成一个个用双引号括起来的单字。
- WM_CLIENT_MACHINE ：执行这个客户程序的机器名称（这个例子应用程序和服务程序在同一部机器上执行，所以机器名称为 venus ）。
- WM_CLASS：显示应用程序的instant name和class name，instant name是命令行中-name选项的值。
- WM_ICON_NAME：应用程序的图标所要显示出来的名称（窗口管理器必须能够支持方可）。
- WM_NAME ：很奇怪的是，这不是应用程序的名称，而是由 -title选项指定的窗口标题名称，有些窗口管理器会把标题名称显示在应用窗口的标题栏上。

注意 上述命令行中-name 和-title两个选项使用得很广，但它们并非通用的选项，应用程序在撰写时必需要使用到X Toolkit（工具箱）才能把这两个选项当成标准选项来用。

有趣的选项如下：

- RESOURCE_MANAGER：这个根窗口属性是resource结构的输入源之一，我们将在下面

详细地讨论它。

- 几个CUT_BUFFER：当你切取一段文本（做剪贴动作常用），这段被切取的文本被放在一个切取缓冲器中，这些缓冲器被当作是根窗口的属性来保存，对于切取，缓冲器是循环使用的，例如上次用5 号缓冲器，则下次用6 号，接下来7 号、0 号、1 号等等，但是粘贴则一定使用上次切取动作所用的缓冲器。

28.7.3 观察X的事件——xev

事件或多或少驱动着整个窗口系统。所有的输入，不论是鼠标或键盘，均由事件来掌握。事件也被用来驱动窗口的重新配置和显示。 xev 程序让你看到当不同的动作发生时，会产生什么事件，以及和事件有关的信息。

由于 xev提供大量的系统内部操作细节，你如果想要测试系统，这是一个很有用的程序，有两个说明联机帮助上未提到的选项可以影响到xev 的行为：

- -bs option：此选项改变xev 对服务程序是否使用backing store，使用backing store 将减少暴露窗口事件的次数（也就是减少应用程序重新更新它自己窗口的次数），正确的选项内容为always、whenmapped和notuseful 。
- -s：使用save-unders，也就是说，要求服务程序保存那些被 xev的窗口遮盖之窗口的内容。

如果将鼠标指针移入xev窗口且按下键盘上的某一个键，则一个（或多个）键盘事件会发生，事件的信息包含了keycode 和keysym，这是最容易观察该机器上某一个键是什么键码的方法。执行xev ，按一个键，xev 便会给出信息，这对定制键盘非常有用。

第29章　定制X Window系统

29.1　使用X的字体和颜色

X支持多种的字体以及几乎无限多种变化的颜色。大多数的应用程序允许你指定应用窗口中各个不同部分的颜色，而几乎所有的X程序均允许你指定你想要使用的字体。

X中的字体：

- 有固定的宽度（像哑终端机的字符）或成比例的间隙。
- 由文本字符或符号组成，或以上两者均有。
- 具有多种的磅尺寸。
- 可以修改以适应特定的屏幕分辨率（例如对于同一点尺寸的某一种字体，你可能对 75 dpi 的屏幕有一种版本，对100 dpi 的屏幕有另一种版本）。
- 有一种标准命名的传统。
- 可以以全名存取，也可以用通配符。
- 保存在特定配置的目录树中，只要服务器在执行，字体便可以加入或移出。

在系统间进行字体的交换有一套标准的格式，并且有工具程序可以将这个格式转换成你的服务器能了解的格式，工具程序也包含了列出可用字体的目录、观察某一特定字体内容等功能。

29.1.1　字体初步

本节的目的是让你尽快地能使用字体，我们将告诉你如何找出有哪些字体可用、指定你欲使用的字体名、看字体的外观、如何在X应用程序中使用字体。

1. 字体命名

有些字体名太长以致使用不便，但很幸运，它们也不常使用，并且， X支持字体名可使用通配符：

- ? 对应任何一个字符。
- * 对应从（字符）长度为零至长度若干的字符串。

这和UNIX外壳传统的通配符文件名相同，使用通配符可使你更容易指定字体名。

注意　如果你在外壳程序的命令行指定一个通配符的字体名，需要把名称加上双引号 。

2. 观察特定的字体

xfd (X font displayer 的缩写) 程序由参数得到字体的名称之后，建立一个窗口并且在窗口中显示此名称的字符字体，例如：

```
xfd -fn "*symbol*-180-*"
```

3. 以X程序使用字体

大多数的X程序使用文字，并且允许你指定使用的字体。如何使用的详细细节可能因不同的程序而异，如果有问题的话可以看联机帮助。但是几乎都是以命令行中选项 -fn fontname 或 -font fontname 来指定字体名， bitmap、xclock、xterm、xload、xmb 和 xedit都是这样操作的。

例如假设你是为了展示的缘故，以很大的字体执行 xterm ，你可以用下列命令行：

```
xterm -fn "*courier-bold-r-*-240-*"
```

注意 如果你给程序的指定对应到一种以上的字体，则服务程序会随便在其中选取一个，例如：如果你省略了上例中的 -r 的指定，则你会使用到斜体字体或反斜体字体，和原来所指定的罗马字体的机会是一样的。

29.1.2 字体命名

在X中，字体可以取成任何名称，但几乎所有的字体均依照它们的本质来命名，这样的命名方式，名字是由几个不相关的部分组合而成，而我们在使用应用程序时，光凭着字体名便可以大略了解字体的内涵。

我们以一个字体名为例，逐一解释它的组件，组件之间是用短横线 (-)分开的，而且可以包含空白，字体名对字符大小写并不会区别，样例如下：

```
-adobe-times-bold-normal--12-120-75-75-p-67-iso8859-1
```

- adobe ：字体的制造厂商。
- times ：字型家族，其他尚包含courier、helvetica 和new century schoolbook。
- bold ：粗体字，其他包含light （细）和medium（中等）。
- normal.字体。例如， r 是roman（罗马体）， i是italic(意大利体）， o是oblique(倾斜体）。
- 12：字符的高度，单位为像素。
- 120 ：字体的磅尺寸，为磅的10倍（120 意为12磅）。
- 75-75 ：字体被设计在显示设备上的水平和垂直的分辨率。
- p ：字和字之间的间隙， p 是proportional（成比例的），相对的是m （monospaced，固定宽度）。

常用的项目为字型、字体粗细、何种斜体字以及字体大小，除了指定这几项的值外，其他的项目不妨借助于通配符的方式去指定。

1. 通配符和字体名

我们曾经解释过通配符的规则：星号（*）表示对应零或多个字符，问号（?）对应任意的单一字符。

你可以随意地使用通配符。当你的设定对应一种以上的可用字体时，服务程序会随便挑一种字体来用，如果你没有设定字体，通常会获得一行信息，而服务程序将会使用缺省字体。

你可以对字体的点尺寸使用通配符，而不是像素尺寸，因为在显示器上一个给定点尺寸的字体对不同的分辨率有不同的像素尺寸，所以用通配符指定点尺寸可以造成与装备无关的效果，上述的样例你可以这样设定：

```
*-times-bold-r-*-120-*
```

也就是说以-120-替换-12-

2. 列出可用的字体 —— xlsfonts

xlsfonts可列出在服务程序中可用的字体（如果你使用命令行中 -display选项，便可列出其他服务程序中可用的字体）。缺省值是列出所有的字体，但是就如同 UNIX的ls命令一样，如果你加上限制，便只会列出合乎限制的项目，例如：

```
xlsfonts  "*-times-*-180-*"
```

列出所有18点Times的字体。

原则上， xlsfonts试图在每行打出尽量多的字体名称，但实际上，大部分的字体名称都很

长，以致一次只能印一个名称，但是要小心，当字体名含有空格时，一行有数个字体名，常常容易混淆。

注意 许多的字体名开头为一短横线(-)，所以xlsfonts会误把此种状况当成命令行的选项来解释以致发生错误，例如：

```
xlsfonts  "-adobe-*"
```

会失败，你可以用选项-fn 加以区分，或者只要在设定之前加一个星号（*）即可：

```
xlsfonts  "*-adobe-*"
xlsfonts  -fn "-adobe-*"
```

29.1.3 观察特定字体的内容——xfd

xfd 是一个字体显示的程序，它建立一个窗口，而后在窗口中将字体的元素显示在长方格中。窗口可能没有大到一次将字体中所有的字符显示出来（尤其是你可能对它重定过大小），但你仍然可以存取它们：

- 向前移动：在xfd 窗口中按鼠标右按钮，窗口的下一页将会出现。
- 向后移动：按鼠标左按钮。
- 获取字符的信息：在字符上按鼠标中按钮， xfd 会给你字符号码，如果你在程序一开始设定命令行选项 -verbose，你将获得一些更多的信息，例如字符的大小以及它在字符“ cell ”中的位置。

29.1.4 保存字体和位置

本节描述字体不同的格式，以及转换两种不同格式的工具，然后讨论服务程序是如何存取字体和你如何更改对字体的选择。最后，我们会给一个完整的样例来说明如何为系统加入一种新的字体。

1. 字体的格式——SNF（Server Natural Format）

字体在服务程序上是以SNF 方式保存，这种格式并不是一种标准，而且为服务程序所专用，所以你不能将字体移到不同类型的服务程序中。

showsnf 程序输出保存在SNF 文件中字体的信息，对字体本身执行 xprop 可获得更多类似的信息(showsnf的参数为文件名，xprop 则为字体名，字体名和文件名并不相关)。

(1) Bitmap Distribution Format（位图分布格式）——BDF

为了克服字体流传的问题，X协会对字体交换指定了一种格式，就是BDF ，BDF 以ASCII的方式表示字符的图形，并且只包含可输出的字符，所以它具有完整的可移植性。

在“ Bitmap Distribution Format ”文件中包含了对BDF 完整的描述。

(2) 从BDF 转换成SNF——bdftosnf

为了让BDF 能够有用，你必须能将BDF 字体文件转换成SNF 文件，目前X协会放弃让这个需求成为X的产品。

在MIT 版中，你可以用bdftosnf来完成转换。

(3) 由其他的格式转换

许多绘图机器拥有它们制造商自己开发的字体，通常特别适合它们的显示器。如果这些字体能在X中使用，那是再好也不过了，但是因为格式的问题，你不能使用它们。

MIT core版并不管这个问题，但是 core版有许多工具程序可将制造商特制的字体转换成BDF 格式，从BDF 你又可以用bdftosnf转换成你自己的SNF。

2. 字体保存在何处 —— 字体目录

字体保存在服务程序上某一个或多个字体目录中，字体目录由三个部分组成：

1) 一个普通的目录，为包含着字体的SNF 文件之所在。

2) 一个被X使用，将SNF 文件名对应到字体名称的数据库。

3) 一个可选择性的别名文件，可以让你用一个以上的名称参考到同一字体（不论你使用了多少个目录，你只需要一个别名文件）。

3. 维护字体目录 —— mkfontdir

mkfontdir 设定新的字体目录并且可以修改它：

1) 在文件目录中搜集了所有你要使用字体的文件，文件可以是 BDF 文件（通常文件名结尾为 .bdf）、SNF 文件(.snf)或被压缩的SNF 文件(.snf.Z)，mkfontdir会自动将非SNF 文件转换为SNF 文件（被压缩的文件是被BSD 压缩程序执行过用以节省文件空间）。

2) 如果你要使用别名，需要在字体目录中建立（或编辑）一个名为 fonts.aliase的文件。有关此文件格式的细节部分在联机帮助中有说明。简单地说，它的格式为每行以空白间隔出两个栏位，第一栏是别名的名称，第二栏则是字体的名称（可包含通配符），例如：

```
tbi12  *-times-bold-i*-120*
```

注意 你对字体定义的第一个别名将造成该字体真正的名称无法使用，以上例而言，你只能以tbi12 来存取字体，这种情形也许下一版会改进，但目前你可以在第二行将第一行反过来即可（但不可使用通配符）。

```
tbi12  *-times-bold-i*-120*
-adobe-times-bold-i-normal--12-120-75-755-p-68-iso8859-1 tbi12
```

3) 执行mkfontdir ，需把文件名当成参数输入，以你使用缺省的 X配置为例 ：

```
mkfontdir /usr/lib/x11/fonts/misc\
      /usr/lib/x11/fonts/75dpi\
      /usr/lib/x11/fonts/100dpi
```

（如果文件目录中没有包含字体数据库， mkfontdir 会忽略它。）

注意 建立字体目录并不会导致服务程序注意它，你必需重新启动服务程序或重设字体搜索路径。

4. 字体搜索路径 —— xset

你可以使用任何数目的字体目录，但如果它们有任何和缺省配置不同的地方，你需明确地告诉服务程序，这些字体目录的列表称之为字体搜索路径或字体路径，你可以设定这个一连串以逗点分隔的文件目录。

• 查看目前的字体路径：使用命令 xset q，这样会输出一大堆信息，其中有一行包含着你的字体路径如下：

```
Font Path : /usr/lib/x11/fonts/misc/,(cond.)
/usr/lib/x11/fonts/75dpi/,/usr/lib/x11/fonts/100dpi/
```

• 设定不同的字体路径：使用命令 xset fp new-path，例如，如果你有大量的本地字体且不想使用多数的标准字体：

```
xset fp /usr/local/xfonts, /usr/lib/x11/fonts/75dpi
```

注意 fp之前并无一短横线(-) ，是fp而非 -fp（-fp 的意义不同，见下述）。

• 当你想重新设定服务程序对字体路径的缺省值时，使用命令：

```
xset fp default
```

• 告诉服务程序重新读入字体的目录，使用命令：

```
xset fp rehash
```

它告诉服务程序你可能已经改变了字体目录的内容而和它必须重读字体数据库，现在新加入的字体可以开始存取了。

• 在现存的路径加入新的字体目录，使用命令：

```
xset +fp dirlist
```

• 在现存路径的左面加入一行由逗号分隔的目录列表，而

```
xset fp+ dirlist
```

则将目录列表加到路径的右面。

• 将字体目录自路径移去：下两个命令行

```
xset -fp dirlist
xset fp- dirlist
```

均可将在dirlist 中的目录自现有路径中移去。

注意 字体路径由服务程序所掌握，而被所有使用该服务程序的客户程序所应用。

字体路径的次序是重要的，我们曾经提过字体设定可以对应一个或多个字体，服务程序会自行选择，但如果对应的字体是在不同的目录中，则服务程序会选择在路径中较早出现者。

你可以利用这个原则来安排最适合你的显示器分辨率的字体。假设你的显示器分辨率为100dpi，则将100dpi字体设在75dpi 之前，例如：

```
xset fp /usr/lib/x11/fonts/100dpi/,\
/usr/lib/x11/fonts/75dpi/
```

如果你指定字体为

```
* -times-bold-r-*-120-*
```

虽然字体有75dpi 和100dpi两种版本，但你会用到100dpi的字体，这正是你所需要的。

29.1.5 例子：在你的服务程序中增加新字体

现在我们将说明如何在你的服务程序中增加一个新的字体的完整样例。为了真实起见，我们以Sun所提供的字体为例，将它转换至BDF ，然后安装它，字体开始时在以下目录：

```
/usr/lib/fonts/fixedwidthfonts/screen.r.7
```

欲将Sun 的字体转换成BDF ，我们需使用contrib 版的软件程序vtobdf（其他系统也有类似的工具）。vtobdf有两个参数，分别是输入文件文件名和欲建立的 BDF 文件文件名，我们可以事先从contrib 磁带中取得此程序，编译它，而后加入到我们可执行的目录中，我们就可以使用它了，我们将或多或少依据 X的标准来命名这个新字体，我们喜欢把输出文件的文件尾名用.bdf，但由于vtobdf会在字型名后自动产生.bdf，所以可以省略它，但以后要重定名称时，则不可省略。

```
venus% cd/tmp
venus% vtobdf /usr/lib/fonts/fixedwidthfonts/screen.r.7\
  -sun-screen--r-normal---70-75-75-m---
```

现在重新命名文件，并将其搬入字体目录：

```
venus% mv- -sun-screen--r-normal---70-75-75-m---\
/usr/lib/x11/fonts/misc/-sun-screen--r-normal---70-75-75-m---.bdf
```

最后，执行mkfontdir 和告诉服务程序重新读入字体目录以便能使用此字体：

```
venus% mkfontdir
venus% xset fp rehash
```

检查一下此字体是否真的可用：

```
venus% xlsfonts "*-sun_screen*"\
-sun-screen--r-normal---70-75-75-m---
```

注意 你的字体可能可以替换其他的缺省字体，但这些字体文件可能因有保护而无法更改，必须问一下你的系统管理员。

29.1.6 使用X的颜色

X允许用日常常用的彩色名，在本节我们描述一些其他指定颜色的方法、解释命令结构如何工作和如何设定一些自己拥有的颜色名。

1. RGB颜色设定

换一种指定颜色的方式，你可以用RGB (Red (红)、Green (绿)、Blue (蓝))三基色来指定，设定形式为

```
#<r><g><b>
```

必须合乎以下的原则：

- 设定必须以井字号(#) 开头。
- 基色需依照红、绿、蓝的次序依序设定。
- 三基色均必须指定。
- 每一个基色为十六进位，共占1～4个字节。因此ffff代表颜色的最大强度，0000代表没有该颜色，例如：

```
#0000ffff0000
```

是最亮的绿色，红色和蓝色一点都没有，同样地，

#000000000000 黑色（什么颜色都没有）
#ffff0000ffff 紫色（全部为红色加蓝色）
#ffffffffffff 白色（全部的颜色）

注意 #rgb和#rrrgggbbb代表的颜色强度是相同的，但后者较亮一些。

- 每一个基色可由1～4个字节代表，但每个基色的位数则相同（例如你不可以用#rrbbbbgg）。

你可以在设定颜色时直接使用颜色名称，例如：

```
xclock -fg #3d7585 -background pink
```

颜色设定的形式往往和你的显示器非常相关，通常没有什么可移植性。

2. X颜色数据库

为了克服#rgb颜色设定不可移植的缺点，而且使系统更易于使用，X使用一个保存颜色名及其相关的rgb 值的数据库。

除非你的系统在设置之后做了明显的改变，应该会有一个 /usr/lib/x11/rgb.txt的文字文件说明数据库的内容。这个文件的前数行类似于：

112 219 147 aquamarine（绿玉色、碧绿色）
50 204 153 medium aquamarine （中度碧绿色）
0 0 0 black （黑色）
0 0 255 blue （蓝色）
95 159 159 cadet blue（学生蓝）

每一行前三个数字表示rgb 的基色值，但这里数值是10进位的，且只为0～255 ， 255 代表

颜色最大强度，第四个部分为颜色名称，允许名称中间有空格。

你可以用程序$TOP/rgb/rgb将此文字文件转换为内部的形式，当你的X系统建立时，并不会设置它。所以，要在你的数据库中加入一个新的颜色，先用文字编辑器将颜色输入rgb.txt文件，然后用以下命令：

```
venus% cd usr/lib/x11
venus% $TOP/rgb/rgb < rgb.txt
```

事实上，rgb并不需要每次均重建内部数据库，只需加入新增（或修改）的项目即可，所以你可以用标准输入来输入颜色：

```
venus% $TOP/rgb/rgb
255  50  50  mypink
…
```

因为没有任何标准的工具程序可以查询内部数据库的内容，因此上面的做法会造成rgb.txt和内部的数据库不一致，所以还是以修改rgb.txt的方式为佳。

29.2 定义和使用图形

一个图的显现是由像素组成的，而像素又对应一个位，当位为“ 1 ”时，像素为“黑色”，而当位为“ 0 ”时，像素为“白色”。X有许多的实用程序来管理图形，你可以用不同的方法来建立、编辑和保存它们。有一些用户程序允许你直接使用它们。其他大部分的程序则以内部的形式使用它们，这些实用程序大都放在X程序库中，使得用户写程序时很容易运用。

29.2.1 系统图形程序库

图形文件的程序库是系统的一部分，缺省保存在下面这个目录中：

```
/usr/include/x11/bitmaps
```

在你的工作站上或许不同，但我们将以此目录为准，并用其中的一些文件作为本节的例子。

29.2.2 交互编辑图形——bitmap

bitmap程序是一个让你以交互式建立或编辑图形的工具，它将位映像以方格子来表示，每一个格子代表一个像素，你可以用鼠标设定或清除像素。

1. 启动bitmap

通过bitmap你可以编辑一个包含一个图形的文件，或从头开始建立一个图形并将它保存为文件。不论是哪一种情况，当你启动bitmap时，你需要给一个文件名，不论是现存的文件或是新建的文件，都要为文件命名。

当建立一个新的图形时，你可以选择性地指定大小，如果你未指定，缺省大小为16×16。举例来说，假如我们想要建立一个比较大一点的十字体数位图像，我们可以用下面的命令行：

```
bitmap big-cross 40×50 &
```

2. 使用bitmap

假如我们要编辑一个现存的文件，可以用下面的命令行启动程序：

```
bitmap /usr/include/×11/bitmaps/cntr-ptr
```

则会有一个窗口出现在屏幕上，右下角以实际大小显示出目前图形的状态，另一个则为相反的图形，其他在右边的“框”你可以用按鼠标按钮的方式来操作它们。

用三钮鼠标编辑图形最简单的方法如下：

- 设定像素：在一个像素上按鼠标左按钮，或者按住左按钮并拖动它，每一个经过的像素方格均会被设定，直到释放按钮为止。
- 清除像素：和上述相同的方法，但是要用鼠标右按钮。
- 反转像素：在一个像素上按鼠标中按钮（也就是黑的像素被清除而白的像素被设定），当你按住中按钮并拖动，所经过的像素格均会反转。

bitmap还有其他的角度：如果你观察接近箭头的上端部分，你可以在其中的一个方格中看到有一个小菱形，这代表了热点，当bitmap用来建造一个光标时会应用到：热点是光标真正动作的点。指向型的光标，热点通常在顶端，而圆形或方形的光标，热点则在中心（你可以用Set Hot Spot和Clear Hot Spot两个命令来更改热点的位置或消去它。）

当你结束了更改动作，可以按Write Output将图形保存，但不会离开bitmap程序。

离开程序，按Quit按钮，如果你编辑了图形却试图在未保存前离开程序，你将会得到提示，以确定你是否真要这样做。

3. 画图

bitmap的画图功能如下：

- 画一条线：按Line，光标会变成一个大黑点，在所欲画的线的一端按一下按钮，而后在另一端也按一下，bitmap会画出这条线。
- 画一个中空的圆：按Circle按钮，同样，光标会变成一个大黑点，在你所欲画圆的圆心按一下，而后在所欲画圆的圆周上的任一点按一下， bitmap将画出这个圆的圆周。
- 画一个填满的圆：按Filled Circle按钮 ，其余同上。

4. 在长方形区域内工作

命令Clear Area、Set Area和Invert Area 必须在长方形区域下操作，长方形区域的决定方式是你在它的左上角以按住鼠标任意按钮的方式指定，然后拖动到右下角，当你拖动时，目前被指定的区域会以高亮度显示。

你可以拷贝、移动或重叠一个区域。你以拖动的方式指定原始区域，而后在目标区域上的左上角按按钮，各种命令的动作如下：

- 拷贝：目标区域会被消除，而所有对应于原始区域为黑像素的均会被设定。
- 移动：原始区域和目标区域均被清除，目标区域对应于原始区域为黑像素的均会被设定。
- 重叠 ：在目标区域中对应于原始区域被设定的像素均会被设定，其他没有改变。

5.图形的文件格式

一个图形会如同ASCII 文字一样保存到文件中，其格式类似C 语言程序。

例如：文件 /usr/include/X11/bitmaps/cntr_ptr 的内容：

```
#define cntr_ptr_width  16
#define cntr_ptr_height 16
#define cntr_ptr_x_hot  7
#define cntr_ptr_y_hot  1
static char cntr_ptr_bits[]=
0x00, 0x00, 0x80, 0x01, 0x80, 0x01,\
0xc0, 0x03, 0xc0, 0x03, 0xe0, 0x07,
0xe0, 0x07, 0xf0, 0x0f, 0xf0, 0x0f,\
0x98, 0x19, 0x88, 0x11, 0x80, 0x01,
0x80, 0x01, 0x80, 0x01, 0x80, 0x01,\
0x00, 0x00;
```

带有_x_hot和_y_hot的变量仅在热点被指定后才会包含进来。

更多的细节包含在bitmap(1) 的联机帮助中，不过无论如何，你不需要直接以此种格式处理图形，任何你想要做的事均有工具程序来处理。

29.2.3 编辑图形的其他方法

bitmap程序对于一个小的图形工作起来算是相当实用的，但它有一些缺点：

• 它不接受较简单格式的输入文件，例如像一些由扫描现存图形所产生的文件。

• 它必须以交互方式执行，对一些程序性的编辑动作并不实用。

• 你可能希望用它产生一些图形来显示，但它无法在非X系统上执行。

要克服上述的问题，需要以字符图片的形式来建立位映像，并提供这个格式和 bitmap的格式相互转换的程序。字符图格式是非常明显的：每一行的像素用一行的字符来表示，黑的像素用一个指定的字符（缺省为# ），而白的像素用另一个字符（缺省为 -）来表示。

你能以文本编辑器或任何其他系统上任何其他合适的程序编辑这些图形，也可以由扫描仪或其他图像设备产生。

X提供了两个程序做字符图格式和图形格式间的转换：

• atobm : 转换一个字符图为标准的图形。

• bmtoa : 转换一个标准的图形为字符图。

两个程序均允许你指定任何字符来代表黑和白像素。

29.2.4 定制根窗口——xsetroot

xsetroot 让你设定你的根窗口的特征，你可以改变窗口背景的颜色和图案，以及窗口所使用的光标。

1. 设定背景的位图

你可以指定任何图形来当作屏幕的背景（只要它是 X的标准格式），在 xsetroot 的命令行上，-bitmap 选项跟随着图形的文件名。例如：

```
xsetroot -bitmap /usr/include/X11/bitmaps/mensetmanus
```

会出现一个精致的背景图 。

2. 设定背景光标

如果你不要使用缺省的大的“ X ”光标，你可以用选项 -CURSOR 加上cursorbitmap 和maskbitmap两个参数来改变它，两个参数均为图形文件名。例如：设定光标为前节所示的图形，使用命令：

```
xsetroot -cursor /usr/include/X11/bitmaps/cntr_ptr\
/usr/include/X11/bitmaps/cntr_ptrmsk
```

maskbitmap决定了cursorbitmap的哪些像素真正被显示出来，光标像素中只有对应到遮盖像素为黑的部分才会用到，而不会显示光标其他的像素。总的来说，遮盖决定了光标的外形，反之，光标图形则决定了外形的颜色。遮盖和光标的图形必须大小相同。

这种遮盖结构在两种情况下非常有用：

1) 它允许 “ 干净地 ”显示出非长方形光标，而不需显示出多余的空白。例如如果没有遮盖，cntr_ptr会显示成一个 16×16白方形中有一个箭头，当你用它指对象时，对象的一部分会被矩形外框遮盖住。

2) 适当地设定遮盖，你可以保证不论背景的颜色为何均能看得到光标。例如 cntr_ptrmsk比 cntr_ptr 的边均大一个元素，所以光标周围围绕着一圈白边。如果遮盖和光标大小相同的话，

当光标在黑色的区域将会消失不见。

你可以让遮盖和光标使用相同的图形：光标的外形会如你所期望（因为遮盖决定外形，而这外形正是你想要的），它们可以工作，但是当光标进入和它相同颜色的区域时，你就很难看到光标了。实际上，并非所有在 /usr/include/X11/bitmaps中的图形均有相对应的遮盖，如果你使用它们当作光标，你必须使用光标图形当作遮盖。

有兴趣的话，试一试把mensetmanus 当作光标和遮盖（热点是在左上角）。

3. 其他背景设定选项

你可以用命令行选项 -solid colour设定背景为单一颜色（在单色显示器上只有黑色和白色）。你可以用 -grey或 -gray设定颜色的灰度，你也可以用 -mod x y 设定格子图案，x 和 y为 1到16的整数。

4. 重定缺省的背景和光标

如果你不喜欢既有的设定，你可以用下列两者之一恢复缺省的光标和背景：

```
xsetroot -def
xsetroot
```

29.3 定义应用程序的缺省选项——Resources

大多数X程序接受命令行选项，以便让你指定前景和背景的颜色、字体、起始位置等等。这种需求是有必要的，因为如果你在程序内硬性规定使用某种字体，而在执行此程序的机器上并没有这种字体，则将使得程序无法执行，所以你不应硬性规定某些参数。

每次执行程序时不太可能在命令行中指定所有需要的选项，因为有太多种可能的组合了，所以X提供了一个叫做“资源”的一般性结构，用来传递缺省的设定给应用程序。

你在系统中几乎所有的定制动作都将运用到“资源”，事实上你为一个应用程序所选择的每一个选项的设定都要用到“资源”，从简单的项目(例如颜色或字体)，到定制键盘或管理显示器如何工作。“资源”非常实用，而且在系统中到处都用得到。

29.3.1 什么是资源

在X的文献中，“资源”有两种意义。第一种是相当低阶的，意指被服务程序管理或建立而被应用程序使用的东西。窗口、光标、字体等均属于这种资源。

另一种是通常在联机帮助中常看到的“资源”的意义：它是一种传递缺省设定、参数和其他值给应用程序的方法。在本节中，我们局限于讨论这种意义的“资源”。在解释现行系统如何工作前，先回顾一下X的早期版本是如何掌握这些功能的，因为现行的结构由此产生。

1. “缺省”的背景

在X较早的版本，对于像窗口背景颜色、窗口边界的颜色、应用程序所使用的字体这类项目你可以轻易地设定其缺省值。

缺省值的设定方式很直接，你只需指定一个窗口的属性和它的缺省值。例如：

```
.Border : red
```

意即所有的窗口均为红色的边（除非你在命令行中重新设定边的颜色），你也可以把程序的名称放在属性之前，则只有被指名的程序才会改变，所以把以下这个规范

```
xclock.Border：blue
```

和先前的规范结合在一起的意义为：缺省时所有窗口均为红色的边，只有 xclock的窗口为蓝色的边。

每次设定缺省值，程序会自动取用该值，所以你不需每次均指定你的选择，它让你依照适合你的工作习惯来使用字体，不论你要用较小的字体以获得更多的信息显示，或用较大的字体以便阅读，它让你为特定的应用程序选择颜色，你可以定义应用程序的起始位置，所以你可以自行设计一些启动屏幕的布置，因为许多缺省值（字体、颜色等）实际上精确的意义为“资源”，所以“资源”的意义逐渐扩增为“缺省值设定”或“设定缺省选项”。

2. “资源”向应用程序发送信息

随着X的开发，应用程序也随之扩增，需要有一个设施传递大量的信息到应用程序中，以定制或指定它们的行为，而不再只是有关颜色和字体的信息而已，例如你可以告诉 xbiff检查邮件的频率，或定义 xterm的功能键12为插入某一特定的字符串，或在 xedit中连续按两次鼠标中按钮代表选择目前这段文本等等。

所以逐渐地开发了“资源”及缺省设施。到目前对于向一个应用程序传递任何信息已是一个一般性的结构。你可以像文字字符串一样地指定信息，应用程序会在内部解释它：例如把这字符串当作一个鼠标按钮的名称，或一种颜色，或由应用程序所发出的一个功能和资源是如何指定的。

这个结构也逐渐地复杂起来，以便让你能正确地指定在何处应用缺省值。在以前，你只能指定所有的程序或某一个特定的程序。现在的系统你可以设定的缺省值如：“终端机窗口的菜单选项”或“在所有窗口的标签”或甚至“除了 xterm以外的所有编辑器窗口按钮框中的功能按钮”。

X Toolkit使得资源在使用上有很大的兼容性并且增进了应用的精确度。你需要先了解Toolkit是什么，才能适当地使用资源结构。

29.3.2 XToolkit

我们先前曾提过，X并不决定用户界面，它只是提供一些结构，让应用程序设计者能组合成任何形式的界面。理论上这是非常合乎需求的，它使得系统拥有一个一般性目的的工具且没有使用上的限制，但从另外的角度来看，它有很大的缺点：

- 对一个用户而言，不同的应用程序有不同的界面，不只是难学难记，且应用程序无法顺利地相互协调工作（例如无法在窗口之间做剪贴），你得到的是一组个别的、独立的程序，而不是一个一致的、合作无间的系统。
- 从程序员的立场来看，意味着基本窗口系统上的每一件事均需从头做起，菜单、滚动条、时钟、功能钮等等都必须一一生产。甚至在单一的产品中，不同的程序员做了一点稍有不同的事，便会导致许多不相容的情况。

为了克服上述的问题，工具箱的方式应运而生。在某些范围，工具箱会决定用户界面的形式。但是无论如何，它会尽量减少这种影响，并让用户界面开发者有更多选择的可能性，工具箱被分为两个部分：

- 一组基本的结构和函数用以配置用户界面的元素称为工具箱内部工具。不论是什么样的界面，任何工具均需使用到它，所以我们可以把工具箱内部工具视为“固定的”，也就是无可替代的。
- 一组提供特定的用户界面（或界面的形式）的元素，这些元素称为 Widget，而工具箱的第二部分称为Widget组，我们认为这是可以替换的，不同的界面提供不同的 Widget组，甚至它们都使用内部工具。

1. 工具箱的第一个部分 —— 内部工具

内部工具定义的实体称为Widget，并提供了所有建立、管理和毁坏Widget所需的设施。从理论上来说，Widget是一个处理特定动作的用户界面的元素，实际上一个 Widget是X窗口加上规则和功能以决定它的输入和输出的动作，也就是说，它如何对用户有所反应。

为了帮助解释Widget的观念，我们将给一个样例。但是请注意。它们并不是内部工具的一部分，而是我们将于下一小节讨论的一个特定的 Widget组的一部分，在此提出的目的只是为了实用。

- Command Widget（命令Widget）：这是一个在屏幕上含有一些文字的长方形“ 按钮 ”（也就是一个小窗口）。当鼠标指针在这个按钮之上时，它的边会呈现高亮度显示，当一个鼠标按钮在这个Widget被按时，一个被程序员指定的软件过程便会被执行。

你已经使用过命令Widget好几次了：主要在xedit 的命令菜单和在xman的主选项窗口中。

- Scrollbar Widget（滚动条Widget）：同样，你也已经在 xterm 和xedit 中滚动文本，在xman中滚动文本和目录列表中使用过了好几次。
- 内部工具提供了基本的结构，任何一个提供界面的较高级软件均需要使用到它，它提供了以下的功能：
- 建立和毁坏Widget。
- 把一组Widget当成一个单元来管理。
- 掌握位置，包括从最高级（也就是应用程序的 -geometry 选项）到最低阶（管理应用程序用到的sub-Widget的位置（例如菜单按钮的位置））。
- 掌握“ 事件 ”，例如在一个Widget中，当鼠标按钮按时呼叫适当的程序、管理窗口的公布和掌握键盘的输入。
- 管理给资源和缺省的每一个Widget。

2. 工具箱的第二部分 —— Widget集

广义来说，内部工具只提供你建造一个用户界面的骨架， Widget集则实际地提供了一个既定的界面，且不同的 Widget组提供不同形式的界面，虽然对任何范围的需求并无法预防混用Widget，但一般仍希望一个系统固定在一致性的Widget集上。

在X core 版上只有一个Widget集提供，我们描述于下：

Athena Widget Set （雅典娜Widget集）

大部分的MIT core版的应用程序使用工具箱和 Athena Widget Set（名称的来源是X由MIT的Athena计划产生出来的）。这些Widget的定义你已在许多应用程序中用过。

我们在前节提过了Command Widget和Scrollbar Widget，至于Athena Widget Set 的其他部分包含：

- Label Widget：这个你可以想象，在窗口中显示的一个字符串或图（例如在 xman主选项菜单中的Manual Browser的标题。）
- Text Widget ：它提供我们所使用的编辑功能。
- Viewport Widget ：一个具有滚动条的窗口，让你可以滚动窗口的内容， xman使用其中之一用以显示联机帮助的目录。
- Box Widget：它以一个指定大小的框管理sub-Widget的布置，且试着将 sub-Widget尽量集中在一起，例如xmh 的Reply 、Forward 等命令钮即是由Box Widget布置。
- VPaned Widget ：它管理sub-Widget，将它们保存在垂直堆栈中，且显示了在两个 sub-Widget之间的分隔线上的“ 把手 ”(grip)，把手可以选择性地让你改变一个Widget的大小，而且另一个相关的Widget大小亦伴随变化，例如我们在看到的 xman窗口的主要元素是由

VPaned Widget 管理的。

- Form Widget ：另一种管理一组sub-Widget的方法，但对位置的选择有更多的灵活性。
- List Widget ：它管理一组字符串，将它们安排在行列中，任何字符串可以通过它来选择动作：字符串会转为高亮度显示，且呼叫一个指定的函数以完成特定的动作。 xman使用一个List Widget来管理它在有关联机帮助中的列表。

我们现在来看一下如何组合Widget以获得所需要功能，我们仍然以 xman为例。xman的联机帮助的目录在它的低阶是 list Widget ，管理目录页名称的列表以及它本身的内容是用viewport Widget（让用户滚动至列表中所需的位置），将Widget聚集在一起，它们是包含在一个VPaned Widget 中，所以事实上这是一个层次式的 Widget，每一个可以完成它的专门功能，而所有的应用程序所使用的工具箱均含有这三个 Widget结构。

3. Widget：名称和类

“ 资源 ”和缺省结构是在Widget名称的基础下工作的，所以我们将介绍对名称的处理。

工具箱提供给程序员一个面向对象的编程系统。它定义对象的类，也就是指定何时对象被建立或如何操作等等的对象属性。这些对象即是 Widget，系统将确保它们和其他的Widget以及其他部分的应用软件以定义明确的方式交互。

当一个程序员建立一个特定类的Widget，它被称为该类的实例（概括地说，一个类是一个抽象的定义，而一个实例在某些地方实际上符合这些定义）。建立Widget必须有一个名称，由程序员指定，例如：程序代码的实际型号为 Creat a Widget, of the class Box Widget,and call it topBox 。在某些环境下Widget的类名也会引用到。总之，一个 Widget有一个实例名称和类名称；更简单地说，有一个名称和一个类。

29.3.3 管理资源 —— 资源管理器

用“ 资源 ”来做什么？用“ 资源 ”能传递信息给一个应用程序，告诉它以某些方式改变它的一般性动作，例如，将窗口的边以粉红色替换原来的黑色，或使用一些特别的字体。

例如，你设定一个包含许多项资源规范的数据库，每一个资源规范以一个应用程序的某些特征命名，且设定一个值给这个特征当缺省值，也就是说，一个规范 (spec) 的形式为

```
characteristic : value（特征：值）
```

当应用程序开始执行时，它会先询问数据库是否有任何特征符合自己所要的设定，或使用相关的值，例如：

```
xclock*foreground:blue
```

意为将值blue设定给特征xclock*foreground 。用以决定一个程序的需求是否符合在数据库中规范的系统部分，被称作“ 资源管理器 ”。

资源缺省值能应用到一个应用程序中的对象（通常是 Widget），就如同设计整个程序一般（例如，你可以对一个特定的子窗口在某一个命令按钮的背景色设定缺省值，而不是只能针对所有应用程序的窗口背景。）为了能达到这一点，我们需要一些严谨的命名方法，以设定对象应用缺省值。

1. 指定资源预定应用到何处

资源管理器根据特征值决定一个缺省规范是否能应用在特别的情况下，我们可将特征值分为三个部分。

1) 你用以设定缺省值的程序属性，例如：背景色、字体等。

你必须指定属性，也就是说你必须设定一个值。给定一个资源的规范而不说明它的值是无

意义的。

注意 在X的文献和联机帮助中，属性通常被称为“资源”或“资源名”，“Resource”通常也被用来当作特征值。

特征值的其他两个部分指定缺省值在何处使用。例如只在特定的程序或特定类型的对象，使用。

2) 应用到这个规范的应用程序的名称，如果你省略它，规范将应用到所有的应用程序中。

3) 一连串的限定条件：当对象符合限定条件时，才会产生指定的应用。限定通常为 Widget 的名称，你可以指定从零开始的任何数目的限定。例如：

```
xclock*foreground:blue
xedit*row1*Command*Cursor:Cntr_ptr
```

第一个例子没有任何限定，第二个例子有两个限定(row1 和Command)。

三个部分依序排列：

```
[<program name>] [<restrictions>] <attribute>
```

并以特殊的分隔符分开，我们将于稍后说明分隔符的细节，但我们先看一些特征值的例子（为了简单起见，我们在例子中只用到颜色属性）。

• 指定在任何地方中的前景色缺省值为黄色。

```
*foreground:yellow
```

我们未指定任何应用程序的名称，所以此规范可应用到所有的应用程序；我们也未指定任何限制，所以对一个应用程序在任何地方都适用。（“*”这个符号就是我们方才提及的特殊分隔符的一种）

• 指定只有在xclock应用程序中的前景色缺省值为粉红色。

```
xclock*foreground:pink
```

这个规范仅能在xclock中适用，但是只要项目中的属性叫做foreground的均适用。

• 针对一个特定应用程序的特定地方：

```
xman*topBox*foreground:blue
```

这个规范仅能在xman适用，而且只能在xman主选项菜单中名为topBox的对象适用（应该适用于xman中所有叫topBox的对象，但实际上只有一个topBox对象）。

• 在第二个例子（粉红色）中我们包含了应用程序名称，但忽略了限制条件，现在我们反过来：

```
*command*foreground:green
```

也就是说，我们指定在任何应用程序中对象名称为 command 的前景色预设值为绿色。

2. 用类名称使得规范说明一般化

前述的例子说明了我们对缺省值结构所需的大部分功能，但它们有一个限制：你必须知道应用程序员设计在每一个应用程序中的 Widget名称，这些信息有时包含在程序的联机帮助中的一部分，但通常被省略。

无论如何，资源管理器有一个尽量减低这个问题的方式：当你在特征中不论何处用到一个应用程序名称、限制或属性名称，你均可用类名称来代替它。

• 应用程序类名称：描述程序的类型，例如xterm 可以是Term Emul （终端机模拟器）的类，xedit 和emacs是Editor（编辑器）的类（但如果xterm 是xterm 的类、xedit是xedit 的类则失去意义）。

• 限定类名称 ：限定几乎是一定不变的Widget名称，所以在此地你可以用Widget类名称。

• 属性类名称 ：属性是如同Widget一般的一个类型或类的实例。

传统上，所有的类名称以一个大写的字母开头，其后则为小写字母，例如属性“foreground”是属于“Foreground”类。我们将简单地解释你如何去发现你需要用来指定项目的类名称。首先，我们将看一些更多的样例，这次用到了类或一个混合了类和实例的例子。

(1) 含有类名称的资源规范说明例子

这些例子演示出你如何在资源规范中使用类，而较前述以更一般性的方式设定缺省值，它们也解释了你如何能使用一个类来设定一个缺省值给较大范围的情况，和将类与实例结合起来以拒绝缺省值在某些特殊情况下的设定。

• 指定在任何地方前景色的缺省值为黄色

```
*Foreground:yellow
```

这个例子和先前例子的区别在于我们是对Foreground类指定缺省值。这个区别之所以重要，是因为并非所有在类Foreground的属性，它的实例名称都是叫foreground。例如，xclock的指针的颜色可由类Foreground的属性来决定，但它的实例名称不叫foreground而叫hand。

• 我们可以用这种结构来帮助我们在文件不清楚的情况下，借助于以强烈对比的组合设定缺省值来分辨对象的类：

```
xmh*Command*Foreground:khaki（土黄色）
xmh*Command*Background:maroon （栗色）
```

这样将使所有的命令Widget(command Widget)呈现醒目而美丽的颜色。

• 对所有的文本Widget(text Widget) 窗口设定一个缺省值，除了xedit 窗口以外：

```
*Text*Background:pink
xedit*Text*Background:navy
```

• 和上例原理相同：

```
*Command*Backgrond:green
xman*Command*Backgrond:white
xman*manualBrowser*Command*Background:orange
```

(2) 如何发现实例和类名称

这很困难，因为没有简单和一致的Widget名称、类、属性等等的文件，我们只能列出每一个最好的来源，并且提示你如何获得更多的信息。

应用程序实例名称：是你所执行的应用程序名称。如果此程序使用工具箱，你能在命令行以选项-name string明确地指定一个不同的应用程序名称，为何需这样做？因为它让你在单一应用程序中定义超过一组的缺省值，而你可以使用 -name 在其间切换。例如，你可以定义一个xterm 的正常缺省值，但对名为demo的应用程序定义一个很大的窗口尺寸和大尺寸的字体，你可以用

```
xterm -name demo
```

(3) 用来演示或教学的xterm

• 应用程序类名称：这没有文件说明，最简单找寻它的方法是启动应用程序并在窗口中使用xprop ，属性当中的WM_CLASS会给你应用程序实例及类的名称，例如，对 xterm 你会得到

```
WM_CLASS(STRING) = "xterm","XTerm"
```

• Restriction/Object/Widget 实例名称：程序的联机帮助会列出你最想要存取的对象名称，例如：xman列出topBox、help、manualBrowser 等等，如果联机帮助并未给你实例名称，则唯一的方法是如果可能，直接看它们的原始程序代码（这种方法通常无法令人满意）。

• Restriction/Object/Widget 类名称：这容易些，大部分的联机帮助会告诉你想知道的对象类，即使没有的话，大部分的对象也是标准集合中的 Widget，当你从系统中使用它们时，

你通常能猜出它们属于哪一个类（例如： scrollbar 的实例名称的99.9% 是类Scrollbar的Widget）。

- 属性名称和类：大多数的联机帮助会列出名称，通常也会有类， xclock的联机帮助便是非常清楚的样例。

无论如何，利用工具箱写的程序通常使用标准的 Widget，它的属性并不会在联机帮助中列出，但通常由一组全部或部分的属性组成，要找到这些属性，你必须在工具箱文件中寻找：

- “ X 工具箱内部工具 ”联机帮助中的附录E 列出所有标准的“ 资源 ”（也就是属性）的名称和类。实例名称项目看起来类似：

```
#define XtNborderWidth "borderWidth "
```

所有的名称均以XtN 开头，跟随其后的名称则以小写字母开头，而类别的名称则以 XtC 开头，类的项目看起来像

```
#define XtCBorderWidth "BorderWidth"
```

在双引号中的便是名称，也就是说， borderWidth 是实例名称， BorderWidth 是类名称。

- 查看“ X 工具箱Athena Widgets ”联机帮助的2.3 节可看到被所有Widget使用到的资源名单，包括名称、类型、缺省值和一段文字叙述。名称的定法如上所述，也就是以 XtN 开头， XtN 之后则为属性名称。
- 查看“ X 工具箱Athena Widgets ”联机帮助中对Widget的描述，每一个会列出它所使用的“ 资源 ”，和上述相同。
- 如果以上均行不通时，你可以查看 Widget的源代码、资源可用到的部分列在 XtResource 数据中。例如， Athena Scrollbar Widget的程序内包含

```
static XtResource 资源[]=
{XtNwidth, XtCWidth,..）},
XtNheight, XtCHeight,..）}。
```

3. 资源规范分隔符

你可以用星号 (*)或点号 (.)来分隔资源规范的元素，星号比较通用一些，它让你指定那些符合范围的案例的特征。我们看到

```
xclock*foreground:pink
```

用来指定xclock中任何东西均使用 foreground属性，所以在此样例中可以看出，星号具有通用字符的效果，甚至可以再一般化一点：

```
*Foreground:yellow
```

它将适合任何应用程序。而点号只是分隔组件，它表示每个组件都必须一一对应，例如：

```
xman.Manual Browser.Help.background:black
```

并不适用于命令按钮或含有 xman的窗口的不同Widget。在我们对这两种分隔符做更精确描述前，我们需要更详细地看一下资源管理器的操作。

4. 资源管理器如何运作

前面一节，我们曾说过一个应用程序会查询资源缺省规范的数据库看是否符合，现在我们描述查询如何掌握这些规范。

应用程序中的个别对象（通常是 Widget）使用资源，而对象则在应用程序上端hierarchicallyixWidget + hierarchy安排，然后可能由一个Widget管理其他的Widget配置，例如文本窗口、命令菜单等等。

对每一个对象，应用程序欲查询资源数据库时，它必须把对象的实例全名和类全名传递给资源管理器，并传递对象所用的一组属性和类名称的一组属性，例如对 SAVE按钮，应用程序

指定：

```
full instance namexedit.vpaned.row1.Save
full class name   Xedit.VPaned.Box.Command
attribute instance-names borderWidth,cursor,font,label,...
attribute class-namesBorderWidth,Cursor,Font,Label,...
```

而后，资源管理器检查每一个在数据库中的规范，看它是否和应用程序所传来的属性和对象名称相符。如果相符，在数据库中规范值的部分会传回应用程序。

在这种相符的操作中，星号和句号的区别非常重要。简单来说，我们可以想到资源管理器只是以单字为基准来对应文字字符串，句号正是每一个单字的分隔符，星号也是分隔符，但不同的是它可以通用字符的方式代表从零到任意数目的单字，对于对应唯一的限制是在数据库中规范的属性必须对应查询应用程序所传来的属性，你不可对属性用通用字符。

现在你可以看到不同的规范如何工作：

```
*foreground:yellow
```

可以应用于任何应用程序中的任何对象。因为星号对应所有的应用程序和所有的限定和对象名称。

```
*Command.Foreground:violet
```

应用于任何应用程序中任何Command 类型中Foreground类的任何属性。

```
xedit.vpaned.row1.Help.background:navy
```

是一个完整的规范但是将只影响到命名当中的对象名称的属性（本例中，尽管事实上是大写的，Help是一个实例名称，它的类是Command）。

除非你有一些非常特别的需求，最好不要用句点当分隔符，尽量以星号代替，这样可减少错误发生的可能，而且在重写应用程序时，比较不会受到层次结构改变的影响。

5. 多种资源规范对应的优先规则

我们现在有一个非常灵活性的方法来指定应用程序的资源，但正因为它太灵活，以致当一个应用程序查询资源数据库时常常有数种规范与之对应，如何解决呢？

简单地说，如果同时有超过一个规范对应，则使用最具体的一个，资源管理器有一组优先规则用来决定是否一个规范较另一个具体。

- 使用句号作为分隔符较使用星号更为具体，例如：*Command.Foreground 较*Command*Foreground 更为具体。
- 实例名称较类名称更具体，例如：*foreground 较*Foreground 更具体。
- 指定一个元素较省略它更具体，例如：xmh*command*foreground较xmh*foreground更具体。
- 元素靠近规范左边的星号较靠近右边的更具体，例如：xmh*foreground较 *command*foreground更具体。

这些规则相当直接，它们大部分可用另一种方法来说明：“如果一个规范对应到另一个规范而为其子集合者，则前者较后者更具体。”

6. 在工具箱程序中应用程序资源

通常一个应用程序使用资源管理器来定义程序层次中 Widget的属性缺省值，但有时需要有和Widget不直接相关的设定缺省值（或传值）的能力。

为了达到这一点，工具箱提供了一个叫做 Application Resource的设施，它和非工具箱缺省的外表原则上相同 —— 应用程序定义了它本身选择的属性。类名称也相同，所以事实上这些属性和一般常见的层次没有什么不同。

xman使用到一点这个设施，它让你能在帮助窗口中指定不同的文本文件，是否在主选择窗口中指定一个你要的窗口，或当程序启动时直接进入一个联机帮助等。

7. 资源和非工具箱应用程序

并非所有的程序均使用工具箱，但工具箱几乎掌握了所有对一个应用程序的资源管理，特别是应用程序的Widget结构定义了对象和子对象的层次，并能适当地查询资源管理器。但是非工具箱应用程序要如何使用资源管理器？

答案是应用程序只须明确地查询每一个它有兴趣的属性。稍早我们曾说过资源管理器对资源无限制，因此应用程序能使用任何它想要的属性名称，只要程序的文件告诉用户它们在何处，它们就如同其他的应用程序一样。

xcalc 应用程序是一个不使用工具箱的程序样例，它也利用上述方式掌握资源规范。

有几点需要注意：

• 此种类型的缺省值没有类。

• 程序以类似类名称（也就是说，第一个字母大写）来定义属性，例如 xcalc 使用 Background、Foreground、BorderWidth 等等。

• 如果大小写错误，你的规范不会工作，例如：规范

```
xcalc.foreground:green
```

会被xcalc 忽略。

• 即使这个程序定义的属性并非层次的一部分，你仍能使用星号当分隔符，例如：

```
xcalc*Foreground:orange
```

29.3.4 资源的类型 —— 如何指定值

直到现在，我们仍然只看资源规范的“ 左半边 ”，而忽略了值的部分。现在，我们来看一看“ 右半边 ”（值的部分）。

简单地说，值只是一个传递到应用程序的文本字符串和资源管理器完全相关，之后，应用程序以此值做它所要做的事。当然，在实际的操作上，应用程序必须明确地做某些事，而工具箱的确也掌握了大多数这一部分的工作，所以你可获得一致的界面。

所以当我们以一个资源值传递我们所需时，实际上我们使用少数的类型，你已看过它们的大部分，你在任何地方均可以资源规范来使用它们：

• Colours （颜色）：我们已广泛地使用过它们 —— 不需要多做解释。

• Fonts （字体）：在一般的方法中我们已描述过，在资源规范中，你也可使用通用字符或全名。例如：

```
*Font: *-courier-medium-r-*-140-*
xterm*Font: 8*13
xterm*boldFont: 8*13
demo*font: *-courier-medium-r-*-240-*
demo*boldFont: *-courier-bold-r-*-240-*
```

设定一个整体性的缺省字体，但使用一个正常的 xterm 指定一个明确的一对字体，和一对被demo应用程序使用的较大的字体（可用 xterm -name demo）。

• Numeric quantities：在不同的情形中，例如：

```
xclock*update:30
xclock*update:60
BorderWidth:10
xlogo*Width:120
```

```
xterm*saveLines:200
```

• Boolean values：指定yes或no，你可以使用yes、on、true和no、off、false，例如：

```
xterm*scrollBar:false
xman*bothShown:true
```

• Cursor names：指定在/usr/include/X11/bitmaps中包含你所要的光标的文件名，例如：

```
xterm*pointer Shape:cntr_ptr
```

• Geometry spec ：全部或部分。

```
xcalc*Geometry: 180*240-0-0
xcolock*Geometry: -0+0
```

设定一个计算器的缺省尺寸及其启动位置在右下角，时钟的启动位置在右上角。

• 键盘转换(keyboard translations) ：安排特定的字符串给一个键，或安排特殊（非输出）动作给键或按钮，这相当的复杂，在下面会全面专门讨论它。

• Pixmaps ：Pixmaps 是像图形纹理(texture) 一般的图案，像位映像或光标一样地指定它们。当你在单色屏幕上工作时非常实用，一旦为不同类的 Widget设定背景，你便能看到应用程序在何处使用到它们。例如：以下的资源规范：

```
*Pixmap: mensetmanu;
List*backgroundPixmap: scales
Box*backgroundPixmap: cntr_ptr
Command*backgroundPixmap: sipb
```

使应用程序看起来很讨厌 —— 杂乱的窗口，每一个空间都以某种图案填满，但有时这样做可能会有用，backgroundPixmap是类Pixmap的属性。

29.4 实际使用资源

前一节解释X资源的规则，结构如何工作和资源规范的格式。本节继续讨论资源，但较强调实用性，我们告诉你如何及何处设定资源缺省值，来影响系统的一部分或全部。在本节结束前，我们将完成一些例子，指出常见的错误，并告诉你如何克服它们。

在这些例子中，我们假设你的工作站叫做 venus ，并且大部分时间你是使用它。从 venus 的显示器，你可在远程的机器 saturn和mars上执行客户应用程序且和 venus 共享文件系统；neptune 则不可，我们曾在上面描述过。

当你在本节中时，记得资源结构是：传递信息给应用程序，通常这些信息是用来传递一些比较感兴趣的缺省值（例如颜色和字体），但只要应用程序取得协调你就能使用这种设施传递任何信息。所以在一般状况下，我们倾向于把“ 资源规范 ”“ 缺省值 ”“ 资源 ”这三个名词视为同一含意。

29.4.1 在何处保存资源的缺省值

在上一节我们只告诉你输入资源规范到一个数据库中，但未告诉你如何做。事实上有几个不同的地方可以保存缺省值 :这些地方通常是一个你可以用任何编辑器修改的简单的文字文件，但有一个特殊的位置需要特殊的工具来设定它，我们先很快地给你一个概念，再讨论细节部分。

首先它的结构非常复杂：包含命令行选项总共有八种设定资源方法，但有两个重点需要注意：

1) 你最好只使用其中的一种或两种设置，只要你做完启动设定，你将只须改变缺省的设

定。

2) 系统掌握许多不同模式的工作，并满足那些在许多显示器上工作或在一台显示器上工作而存取远程机器的用户。

总之，这些设置让系统尽可能富于灵活性，但无论何时，你将只须存取其中的子集合即可。

1. 设定资源的八种方法

总共有八种方法设定资源，但它们可分为下面几类：

• 应用程序专用的资源：资源的表列，限定文件只能被特定的应用程序读取。

• 服务程序专用的资源：应用设定，不管应用程序在哪一种主机上执行。

• 主机专用的设定：对应用程序在主机上执行有关的设定和显示器无关。

• 命令行选项：在执行时期做一次关闭设定。

(1) 应用程序专用的资源 —— 方法1 和方法2

工具箱程序初始时在和应用程序直接相关的两个文件中寻找资源，这些文件只能被特定的应用程序读取：

1) 应用程序类资源文件：这个文件包含了机器一般性对应用程序的类的缺省值，通常为系统管理员所设定。它的名称就是应用程序类的名称，在标准安装的系统中它是保存在目录/usr/lib/X11/app-defaults 中，例如xterm 的相关文件为：

```
/usr/lib/X11/app-defaults/XTerm
```

在core版中，有一个相关于Xmh 的此种文件，观察此文件可以看所使用之设定的类型。

2) 你自己拥有的应用程序专用的资源文件：这个文件的名称和上述相同，但它存放在不同的地方 —— 由外壳变量$XAPPLRESDIR所指定的目录，如果未定义，则放在 home目录。例如对Xmh 类的程序，它的文件放在下列二者之一：

```
$XAPPLRESDIR/Xmh
$HOME/Xmh
```

你可以使用此种文件，处理方法1 中你不喜欢的文件使其无效。

(2) 服务程序专用的资源 —— 方法3 和方法4

这是对你目前工作的服务程序（显示器）做有关的设定。键盘的设定通常是服务程序专用的（因为不同的显示器有不同的键盘）。另一个服务程序专用的特征为显示器是彩色或单色。

资源和这些有关的项目会被所有与这个终端机相关的应用程序应用到，并且不论应用程序在何主机上执行。（例如，如果你使用的显示器为单色，则不管你的应用程序在何处执行，你还是不会要它使用彩色。）

保存服务程序专用设定的方法是：

3) 服务程序的RESOURCE_MANAGER属性：使用下述的xrdb程序，你可以在服务程序的根窗口的RESOURCE_MANAGER属性中保存资源设定。它的优点如下：

a) 你不需编辑任何文件即可设定缺省值。（当你为了了解系统而实验系统时特别有用）

b) 资源被服务程序掌握，所以不论应用程序在哪一部主机上执行，均能被所有的应用程序应用。在我们的样例中，在neptune 的情况下特别有用，甚至在不和我们的显示机器venus 共享文件系统时，它仍然自动地选出为了使用此显示器所必需的资源设定。

4) 你的$HOME/.Xdefaults文件：（只有在根窗口没有RESOURCE_MANAGER属性定义的情况下使用）。如果你对xrdb尚不熟悉，你便可以此文件取代，但你必须在每一部你执行客户机应用程序的机器上均设定一个。

(3) 主机专用设定 —— 方法5 和方法6

主机专用缺省值和服务程序专用相反，不管应用程序所使用机器的终端机为何，只要应用程序在此主机上执行，均使用主机专用缺省值，你可以用它们来：

- 让应用程序在不同的机器上对不同的文件系统作计算，例如：被一个应用程序读取的数据文件可能在不同的主机上保持不同的位置。
- 区分显示在同一个屏幕上不同的主机的窗口（这些窗口可能由同一个应用程序执行），例如：你可以要所有在mars机器上执行的xterm 的窗口为红色的边框，而在saturn上执行的窗口为黄边。
- 调高一个相同的应用程序在不同的客户机机器上版本的差异，例如： xterm 在venus 是标准的MIT 版，但在neptune 机器上是由第三方厂商修改过以适应机器结构的产品，这两版的xterm 可能并不完全相容。

主机专用Resource保存在：

1) 由$XENVIRONMENT 来的文件名称：如果外壳变量$ENVIRONMENT有定义，它会被解释为一个含有资源设定的文件的完整的路径名称。

2) 你的$HOME/.Xdefaults-thishost 文件：（当$XENVIRONMENT 未被定义时使用）。注意它和我们先前的文件有所不同，它必须附加上主机名称，例如，如果你在 neptune 执行应用程序而在venus 显示（假设RESOURCE MANAGER属性未定义），则服务程序专用资源读取自：

```
Xdefaults
```

而主机专用资源则是：

```
Xdefaults-neptune
```

两者均在neptune 的主目录中。

注意 我们曾说过类似“服务程序专用资源读取自...”这可能造成误导：“如果你实际需要，你可以放置任何类型的资源设定到任何的文件或数据库。”我们真正的意思是你应该放置机器特性或不论资源到任何地方，如果你这样做，你将获得你需要的动作。

(4) 命令行选项 —— 方法7 和方法8

最后，你可以借助于命令行选项设定应用程序的值。通常当你设定缺省值时，为的是你不需要使用选项为你的程序作X相关的设定。但你实际上可以用它们来：

- 一次关闭，例如：你暂时性地在屏幕上需要一个极小的xedit。
- 为了区别在相同应用程序中各自的实例。你已看过一个这样的例子，当我们使用命令

```
xterm -name demo
```

来设定应用程序的实例名称给demo，将造成以应用程序名称为demo的资源替换xterm 的资源。

命令行选项分为下列两种：

1) 应用程序专用选项：例如xclock的-chime的xpr 或-scale。

2) 工具箱标准选项：所有用到工具箱的应用程序均接受一些标准的命令行选项，我们看过其中的大部分，包括-fg, -bg, -display,-geometry 等等。

在其中有一个选项-xrm，重要的足以用一个小节来描述。

(5) 工具箱标准选项-xrm

大多数一般的资源均能被命令行选项明确地设定，例如你可以用 -bg colour设定窗口背景颜色。但无论如何，有一些资源并没有符合的选项。为了克服这点，工具箱提供一个“捕捉遗漏”的选项-xrm（X 资源管理器的缩写）。

-xrm以一个参数当做资源规范，就如同你在缺省值文件中输入的相同，例如：你可以输

入：

```
xclock -xrm "*update:30"
```

和

```
xclock -update 30
```

是相等的。

在同一命令行你可以使用数次-xrm，但每一次只能包含一个资源规范，

例如：

```
xclock -xrm "*update:30" -xrm "*chime:on"
```

-xrm的好处在于你可以用它来设定任何资源供应用程序使用，尤其是那些和命令行选项不符合的资源。其中一些非常有用的像：

iconX, iconY：窗口图标左上角x,y 坐标的位置。

iconPixmap：被用来当作窗口图标的图形的名称，你可以用它来指定任何的图形当作应用程序图标。（图形为已有或利用bitmap程序建立。）例如：命令

```
xedit -iconic -xrm "*IconPixmap:cntr_ptr"\
      -xrm "*iconX:500"\
      -xrm "*iconY:400"
```

其意义为将xedit 设定以图标开始启动，图标的左上角坐标为 (500,400) （在大多数的显示器会在屏幕中央），使用名为cntr_ptr的图形来当作图标。

backgroundPixmap：设定用一个图形当作背景。

borderPixmap：设定以一个图形当作窗口的边，例如：

```
xclock -bw 20 -xrm "*backgroundPixmap: scales"\
xrm "*borderPixmap: cntr_ptr"
```

执行xclock，用一个宽达20个像素的边框，窗口的背景为鱼鳞图案，边框则用 cntr_ptr的图形。

所有的这些资源当然也可用类指定。（如IconX ， BorderPixmap等等。）

注意 请记住，-xrm只有在程序有用到工具箱才可应用。

2. 设定资源不同方法的总结

现在我们将如何对一个指定应用程序资源设定的八种方法进行总结：

- 应用程序专用资源：它们被两个文件掌握，且仅能被工具箱使用，其中一个文件通常由系统管理员设定，另一个由你自己设定。
- 服务程序专用的资源：不是存在根窗口的 RESOURCE_MANAGER属性中，便是在你的$HOME/.Xdefaults文件中。
- 主机专用资源：如果外壳变量$XENVIRONMENT 有定义的话，存在其所定义的文件中，否则在你的$HOME/.Xdefaults-host 文件。
- 一次关闭设定：用应用程序的本身命令行选项来设定和用工具箱标准命令行选项，包含“捕捉遗漏”-xrm。

它们以下列顺序处理：

```
if （程序使用工具箱）
    读取 /usr/lib/X11/app-defaults/class文件 (1)
    读取你的 $HOME/class文件 (2)
if （RESOURCE_MANAGER属性被定义）
    处理内含的指定 (3)
else
```

读取你的$HOME/.Xdefaults文件 (4)
if （外壳变量XENVIRONMENT被定义）
读取所定义名称的文件 (5)
else
读取你的$HOME/.Xdefaults-host 文件 (6)
if （程序使用工具箱）
处理标准的资源选项，包含-xrm (7)
处理应用程序本身的选项 (8)

29.4.2 在服务程序上保存缺省值 —— xrdb

大部分缺省值的结构均和文件有关，当应用程序启动时，不同的文件被读取且其内容被处理，这种方式的缺点为你希望所有的客户程序在一个特定的服务程序上使用同一组的缺省值，但客户程序所执行的机器上如果没有一个共同的文件系统，你该怎么办？

答案是在服务器本身保存缺省值。 X的属性设施是一个具有一般性目的的结构。（记住，一个“ 属性 ”是一小段已知格式资料的名称，被保存在服务程序），指定由服务程序根窗口的RESOURCE_MANAGER属性加载，且当应用程序启动时系统会注意此事。当窗口系统启动时，RESOURCE_MANAGER属性未定义：如果你要使用这个设施，你必须明确地设定它。

并没有一个一般性的工具来操作一个属性，所以X提供了一个特殊的程序来处理资源属性，它就是xrdb（the X Resource DataBase 实用程序）。

1. xrdb能做什么

为了实用起见，本节剩余的部分，我们只把 RESOURCE_MANAGER属性和它的内容当成“ 数据库 ”。

xrdb的功能非常简单，它让你能：

- 设定新的数据库。
- 看目前有那些资源在数据库中。
- 在现存的数据库加入一个新的资源。
- 完全去除数据库。

这些是基本操作，且很容易完成。当然也有一些更进一步的功能可以很精确地让你控制资源，但我们先来讨论基本操作。

2. 使用xrdb的基本功能

xrdb的操作类似大多数UNIX的程序：它从一个文件或标准输入读取输入数据，并且你可以用命令行选项来控制它的操作模式，它所读取的输入是我们曾经看过的一系列资源设定，不过比较特别的是它把这些设定加载数据库，让我们看一看它主要的功能：

- 设定一个新的数据库：输入下面命令两者之一：

```
xrdb filename
xrdb < filename
```

用以将一个文件中的设定加载到一个数据库中，如果只键入 xrdb，表示你将由标准输入（通常为键盘）直接输入设定，稍后我们将说明 xrdb所接受的文件格式，但现在先把输入资源设定当作和 .Xdefaults文件或-xrm参数相同的方法，例如，你可以用下列的方式定义 xclock设定：

```
venus% xrdb
xclocks*Background: pink
xclock*update: 30
```

```
xclock*backgroundPixmap: cntr_ptr
<end-of-file>
```

通常你用一个文件当作xrdb的输入，也就是说，xrdb从一个文件加载缺省值作为你的窗口系统初始化的一部分。如果你很有经验，直接输入它的设定也许容易些。

• 查看现存数据库中的内容，输入命令：

```
xrdb -query
```

则xrdb将以纯文本格式输出数据库的内容（-query可以缩写为 -q ）。

你可能记得也可以在根窗口用 xprop 来看数据库的内容，但 xprop 的输出格式不太灵巧，它给你其他一大堆你不需要的信息。

如果需要，你可以抓取xrdb的输出到一个文件，编辑它，更改设定后可再用它当作 xrdb的输入。

注意 查看数据库，你必须使用选项-query。如果忽略这个选项而只输入xrdb，将造成会清除数据库，且xrdb在等待你自标准输入键入新的设定。

• 在现存数据库加入新的设定：加入新的设定到数据库且不要破坏原有的设定，使用命令：

```
xrdb -merge filename
```

（-merge可缩写为-m，如果你省略文件名称，xrdb会自标准输入读取。）xrdb自指定的文件中读取资源设定，并加入现存的数据库中；对于数据库中已存在的资源，如果有新的设定，旧值会为新值替换，否则不会变动。

• 完全移去数据库：如同先前所述，当系统结束时数据库会自动消失，但如果你在系统仍在执行时移去数据库，使用命令：

```
xrdb -remove
```

3. xrdb的文件格式

你已知道大多数的格式细节，你可以用标准的资源规范的形式：

```
characteristic: value
```

上述的格式你已看过多次，但xrdb有两个额外的规则：

1) 注解：每一行的开头如果是惊叹号（!）会被忽略，所以你可以此当作注解。

2) xrdb缺省将它的输入行传到C 预处理器。

让我们进一步看一看预处理器的过程。

让我们看一看一个你可能碰到的典型问题。假设在一般的场景，你使用下列显示器：

venus	彩色屏幕，正常分辨率。
saturn	单色屏幕，正常分辨率。
mars	彩色屏幕，高分辨率。

以上三者共享一个共同网络文件系统。当你在一个显示器上启动 X，你需要定义缺省值来反应显示器的特征。例如：在高分辨率屏幕你可能需要较大的缺省字体，或是你不需要在单色系统上定义彩色缺省值。

如何做呢？让我们看一看，如果你能使用 .Xdefaults-host 文件：在 .Xdefaults-venus我们包含了彩色指定，而在 .Xdefaults-saturn 我们只放入单色类型的参数。行得通吗？当然，但是是有限度的：它只能掌握应用程序在和服务程序相同的机器上执行，如果应用程序在其他的机器上执行会得到它们主机上的缺省文件。所以如果你使用 venus 且在saturn启动远程的客户机，将会用到 .Xdefault-saturn而错失所有的彩色指定。

你能够只使用 .Xdefault 文件来区分机器吗？不能，因为三台主机共享相同的文件系统，

所以$HOME/.Xdefaults会被venus 获得也会被其他的机器获得。

答案是在资源处理程序的某些地方，有一个结构可以分辨出所使用服务程序的某些特征。xrdb可以用相当简单的办法做到这点，它先定义一些说明服务程序特征的 C 预处理器符号，而后将它所有的输入传递到预处理器，最后将处理过的数据加载数据库。联机帮助列出所有的xrdb定义的预处理器的符号，但在此处我们需要用到的是：

X_RESOLUTION=n:n是每公尺长屏幕有多少像素。（根据我们的服务器，我们正常分辨率的屏幕为每公尺3454个像素。）

COLOR ：只有屏幕支持彩色才被定义。

WIDTH,HEIGHT：屏幕的宽度和高度，单位为像素。

• 你可以使用所有预处理器的功能。例如，我们使用它的表示掌握能力：

```
#if X_RESOLUTION > 3600
```

• 你可以在文件中任何地方使用预处理器符号，并不只是前面有 # 号的行，例如，当

```
xload*Width: WIDTH
```

在venus 上xrdb执行到时，它将会读取成：

```
xload:Width: 1152
```

所以由缺省值可知， xload 窗口宽度将和屏幕宽度相同，高为 80个像素，且在屏幕的正上方。

注意 大多数UNIX预处理器定义了一些和它们机器结构与操作系统相关的符号，这些可能会干扰你，特别是UNIX通常定义的符号。现在xrdb定义HOST为显示器名称中主机名称的部分，所以你可能认为你可以像这样使用一个资源规范：

```
demo*title: X demo using display HOST
```

比方在venus 上，可能希望它相当于：

```
demo*title: X demo using display venus
```

事实上，在我们的机器上会得到

```
demo*title: X demo using display 1
```

原因为显示器名称是unix:0.0，所以主机名称部分为unix，但预处理器已定义了unix，所以整个解释的顺序为：

```
HOST -> unix -> 1
```

你可以用xrdb的-u选项来解除符号的定义，用以克服这点，也就是

```
xrdb -UNIX < filename
```

但即使这样，主机名称仍为unix，除非你明确地指定显示器：

```
xrdb -display venus:0 < filename
```

另一个会产生干扰的样例，如何你输入规范

```
xedit*Font: *-sun-screen-*
```

使用xrdb，现在用一个xrdb -query ，你可以看到在数据库中实际地设定：

```
xedit*Font: *-1-screen-*
```

在我们sun 的机器上，预处理器定义成另一个符号。如果你使用和你的机器相关的名称，你可能也会得到相同的效应。如果你决定不需要预处理器的功能，你可以用 xrdb的-nocpp选项停止它的功能。

4. 如何将数据库设定和xrdb输入文件连接在一起

借助于像前述在一个含有大量预处理器命令的文件执行 xrdb，你初始化了数据库，在稍后的期间，交互式的使用xrdb，你将对数据库做大量的更动，现在你需要记录这些设定，且将之

与原来的输入文件连接，以备将来之用。

如果你只使用xrdb -query ，你只能获得目前的设定：所有在输入文件中的条件指令行若和现在的服务器不符则不会被包含。例如在 saturn上执行前述的文件，则所有颜色和高分辨率的设定，均被忽略（当然以 #开头的也不例外）。为了克服这点， xrdb提供-edit 选项，例如命令：

```
xrdb -edit myresf
```

连接目前在数据库中的值到文件 myresf内存在的内容，它借助于比对资源指定特征值的部分做到这点：如果在文件中某一行和数据库中某一项特征相同，则文件中值的部分会被在数据库中的值替换。用此方法，所有的以 #开头的行和条件设定均会保留在文件中。

注意 预处理器完全不可以使用 -edit 选项，那会导致问题，我们看一下当我们使用 venus ，且以前述文件初始化数据库时，会发生什么情况，假设我们做了更动：

```
venus% xrdb -merge
XTerm*font: *-courier-medium-r-*-140-*
<end-of-file>
```

然后用命令：

```
xrdb -edit myresf
```

将设定更改的部分放回文件，我们看到两件事：

- 前处理器符号在规范中值的部分会被字面(literal)值替换，

例如：

```
xload*Width: 1152 会被
xload*Width: WIDTH 替换
```

- 在规范中只要特性符合，值均会被替换，甚至那些在条件段中目前尚未应用到的也不例外。例如，在前述文件，设定 XTerm*font的那两行（一行在高分辨率那段，一行在正常显示器那段）都会被更改，即使我们只需要改变正常显示器也不例外。

29.4.3 常见的错误和修正

你对系统不熟悉的时候，资源看起来相当的复杂。当有些状况不能正常执行，而系统无法帮助你查觉是什么错误，或你在何处犯了错误。这里列出一些常见错误，并提出如何修正它们。

- 如果你未设定一个应用程序的名称和类，确定在你的资源规范之前加一个星号，（如果你省略这个星号，将没有任何东西会对应这个规范）这个错误在你使用 -xrm时特别常见，例如：

```
xclock -xrm "update:3"  （错误）
xclock -xrm "*update:3"  （正确）
```

- 并非所有的应用程序均使用工具箱，非工具箱的程序不使用类，且它们的属性名称也可能不同。例如，规范

```
* Geometry: 300*400+500+600
```

对xclock，xlogo 有效，但对xcalc 无效，因它不使用工具箱。xcalc 使用属性名称Geometry（开头为大写的G ），因为在这种情况，工具箱类名称和 xcalc 的属性名称相同，所以单独一个规范

```
*Geometry: 300*400+500+600
```

可以对所有这类的应用程序有效。

- 你可能在规范中用了错误的属性或Widget的名称，特别是容易把类名称和实例名称搞混，

例如：以下两者均错：

```
xclock*Update: 10
xclock*interval: 10
```

其他常见的错误如：

```
xterm*Text*background:blue
```

它不能执行的原因为xterm 并未使用Text Widget ， xterm 正常的窗口和Tektronix 的窗口分别使用Widget类 VT100和Tek 。最后，当你知道一个Widget是什么类，你可能对实例名称假设错误，不是Widget 本身便是其中之一的属性。试着更换类名称来修正这个问题。

- 即使你已设定实际的Widget和属性名称或类，应用程序可能以不是你预期的方式使用它们。例如：你可能设定如下：

```
xterm*Width: 40
xterm*Height: 10
```

希望用比平常较小的窗口启动xterm ，但它不能执行。 xterm 只能在Tektronix window应用这些值，无法在正常的窗口。

- 你可能所有的设定完全正确，但仍然什么也没发生，例如：

```
xmh -xrm "*inc.Label: Include"
```

是一个正确的方式。但在标准系统的发行版，是没有任何动作发生的，原因是 xmh 有一个应用程序设定缺省值文件/usr/lib/X11/app-defaults/Xmh ，其中有一行：

```
xmh*inc.label:Incorporate New Mail
```

这个规范较我们的设定有较高优先。

- 将规范

```
*Width: 200
```

单独包含在数据库将导致大多数的工具箱程序启动失败，且有一个信息说它的“ shell Widget has zero height or width ”。如果你设定height和width 二者之一，你必须也设定另外一个。

- 如果你用编辑器建立一个资源文件，你可能省略了最后一个新行。这将导致当你试图用xrdb加载它时整个文件均被忽略。为了避免这样，当加载资源时，用一个命令行像：

```
xrdb resfile; xrdb -query
```

如果xrdb无法打输出resfile 中的内容，就是有问题了。

- 你可能忘了用 -xrm选项的参数来获得资源规范，有时有人会把资源规范放入一个文件，而以文件名称为-xrm的参数，预期它自此文件中读取资源。
- 最后，一个非常人性的错误。当你发生问题，你通常会循环动作：编辑 resource文件，保存它，加载资源到数据库，执行应用程序和看一看发生什么状况。但是常常会忽略其中 " 加载资源到数据库 ” 以致你更为困惑。

29.5 定制键盘和鼠标

电脑的键盘通常含有一些特殊功能键，在此有一些方法来制定这些特殊功能键，使它们能完成特定的功能以适合你工作的方式。例如，你可以定义一些键来输入那些你常用的命令，或只需按一个键便能够输入一段程序。

在X中，你能制定的不只是功能键，其他一般的键和鼠标的按钮也都可制定。对每一个应用程序，你均可指定特别的功能给键盘和鼠标按钮，或两者的组合 (例如在 xedit中你可以结合 Shift 键和鼠标的右按钮来让你向前移动一个单字)。所有使用X 工具箱的程序均允许用户利用

一个被称之为键盘转换的设施来执行此种定义，且此种定义借助于正规的资源结构传递给应用程序。那些不使用X工具箱的应用程序，同样地也可以用相同的设施来制定，但它们需个别的定义所以不能广泛地应用，从现在起，我们假设每当讨论有关转换的种种，均为对那些使用 X 工具箱的应用程序而言。

就如同所有的资源一样，转换是当应用程序执行时才处置。例如你可以拥有数个具备不同转换设定的 xedit，在同时一起执行。你可以让一个 xedit 适合编辑文本，另一个适合编辑程序码，而另一个适合编辑文书。

本节讨论转换 —— 包括它们的定义格式，如何将它们设定到应用程序和它们所涵盖功能的范围。我们首先以实例来介绍，逐渐地导引你看到不同的角度。而后比较正式和详细地讨论转换。最后，我们列出当你使用转换时常会碰到的错误，并给你一些如何克服这些问题的提示。

29.5.1 实际使用转换

工具箱转换结构最简单的用途便是让你制定你键盘的键。例如，当你使用 xterm 为一个执行一般外壳命令的窗口时，你可能希望定义一些特殊功能键来输入你常用的命令，且希望指定的关系如下：

当我按下F1这个键时，我希望这个字符串输入

```
F1  rm core *.tmp <newline>
```

利用工具箱达到此目的方法为：给使用转换的 Widget中的资源指定一个值。此值设定应用程序中所必需的定制，且使用工具箱的 Translation Manager (转换管理器) 所处理。此资源属于类Translation,且其实例名称几乎一定是translations。

1. 如何对应用程序指定转换

对前述xterm 的例子，我们定义 (在即将被应用程序读入的资源数据库中或一些资源文件中) 一个规范类似：

```
xterm*VT100*Translations: (contd。)
<key>F1: string("rm core *。tmp")          (注意：不完整！！)
```

其意为在任何类 VT100的 xterm Widget 中，当键 F1 被按下时，插入字符串“ rm core *.tmp ”)。

不幸的是，并没有这么简单。转换管理器会把上面的规范解释为 “ 去掉所有现存的转换，且加入..)”，所以所有正常的像“ A键是插入一个 A ”这种对应关系都会消失。为了克服这点，你必需在资源值之前插入一些称为伪指令的语法：

```
xterm*VT100*Translations: #override(contd.)
<key>F1: string("rm core *.tmp")
```

通常你会希望保持大部分现存的对应关系，而只是把你明确指定的值覆盖上去。所以你一般都是在你的转换表中，指定 #override。

现在这个规范可以开始工作了。通过启动一个 xterm，且把此规范 (在两个单引号 (’)中间的部分) 当成选项 -xrm 的参数来测试它：

```
xterm -xrm 'xterm*VT100*Translations: ...'
```

按下特殊功能键 F1,你将看到指定的字符串成功的插入，但并未包含新行字符，你可以用一点语法的技巧来克服它，像：

```
 xterm*VT100*Translations: #override(contd.)
  <key>F1: string("rm core *.tmp")string(0xd)
```

这解释了以下两点：

1）string()的作用和它的参数相关。你可以直接输入文本（例如 string(lpq)），但如果文本包含空白或非字母字符，则必需在文本前后加上双引号。如果参数是以“ 0x ”开头，则将其后解释为十六进位，并插入相对的ASCII 字符。(例如，0xd是RETURN)

2）在你指定此功能时可结合一个以上的作用。在上例，我们用到 string()作用两次，如果我们知道其他的作用，我们也一样可以对应起来。

你可以根据需求在一个表中定义许多的转换。假设，我们在前面的转换中增加对应关系：

当按下F2这个键时， 我希望这个字符串被输入

```
F2  lpq-Plpa3 <newline>
```

对此的转换为：

```
<Key>F2: string("lpq-Plpa3")string(0xd)
```

所以可以将本行加入前面的表中。但是转换管理器的格式规则告诉我们必需将两个转换以“ \n ”分开且独立成为一行：

```
xterm*VT100*Translations: #override(contd.)
<key>F1: string("rm core *.tmp")sting(0xd) \n(cond.)
<Key>F2: string("lpq-Plpa3")string(0xd)
```

以上的形式将造成管理上的困难。你可以借助于包含“ 隐藏的新行字符 ”来使它容易理解： (新行字符以倒斜线 “ \ ”处理)

```
xterm*VT100*Translations: #override\n\
<key>F1: string("rm core *.tmp")sting(0xd)\n\
<Key>F2: string("lpq-Plpa3")string(0xd)
```

你可以放置任意多个你所需的“ 隐藏的新行字符 ” ,且几乎在任何地方均可，它们只是被忽略而已。(只要和转换管理器相关，甚至你每隔一个单字便使用一个也没关系。但千万不要在一个规范的资源特征部分使用它们，资源管理器无法解释它们，也没有相同的效果。) 如果你感觉有些混淆，不用担心。简单地说，资源结构需要的是要在一行中的一个资源规范的“ 值 ”的部分，而转换管理器以分开的行来分开 (也就是以\n终结)，而用户刚好以每一个实际分开的行代表一个意义以增加可读性，所以规则很简单：

在除了最后一行的每一个转换行均加上一个 “ \n\ ”。

2. 转换可对应许多型号的作用

上述的 xterm例子，演示了如何能够当你按下一个键时，插入任意的字符串。但转换结构的功能比这更多，它可以将任何Widget所提供的作用对应到按键，让我们详细一点地看一下这些作用。

前述的例子，我们在 xterm的 VT100 Widget完成了键F1和F2在 string()上的对应。我们将仍以 xterm为例，说明更多的作用。

查阅xterm 的联机帮助，在标题KEY TRANSLATIONS 和 KEY/BUTTON BINDINGS你将发现列有数个作用。我们将定义一个转换对应键 F3到insert-selection()作用之上，所以我们可以用键盘来替换鼠标，将先前“ 剪 ”下的文本“ 贴 ”出。联机帮助告诉我们此作用需要一个参数，从列出的缺省对应，我们可以看出缺省的 “ 剪贴 ”结构为使用 CUT_BUFFER0, 所以我们将CUT_BUFFER0当作参数。我们的资源规范是：

```
xterm*VT100*Translations: #override\n\
<key>F3: insert-selection(CUT_BUFFER0)
```

到目前为止，这只是一点小小的使用。然而，假定说你花了许多时间在文本文件上工作，你用 tbl格式化，你用 nroff在屏幕上预览它们，用troff 排版，且将输出送到你的一个用过滤器为tr2printer的打印机上。设定转换为：

```
xterm*VT100*Translations: #override\n\
<key>F3: string("ed") insert-selection(CUT_BUFFER0)\
string(0xd)\n\
<key>F4: string("tbl") insert-selection(CUT_BUFFER0)\
string("| nroff -man") string(0xd)\n\
<key>F5: string("tbl") insert-selection(CUT_BUFFER0)\
string("| troff -man -t | tr2printer") string(0dx)
```

xterm 会确定这些转换是以xrdb自数据库加载或是在一个资源文件中，并加以处理。现在当你启动xterm，用鼠标“剪”取你所需的工作的文件名称。接下来，便可按 F3键编辑它，按F4键预览它和按F5键在硬拷贝上排版它。

(1) 更多的Widget作用样例 —— xbiff

查阅xbiff 的联机帮助：在ACTIONS 的标题下，你将看到Mailbox Widget所支持作用的名单。它惟一缺省的转换为当你按下任何按钮时降下邮件邮件的标记(作用unset())。我们将设定转换让你以键盘来运用这些作用，将这些作用对应到“？”和“UP”“DOWN”两个方向键如下：

?	check()有新的邮件吗?
UP	set() 升起邮件的标记
DOWN	unset()降下邮件的标记

以下是相关的转换表：

```
xbiff*Mailbox*Translations: #override\n\
<Key>?: check()\n\
<Key>Down: unset()\n\
<Key>Up: set()
```

以此测试之：用xrdb从你的资源数据库加载这些设定，然后启动xbiff ，将鼠标指针移到窗口内。重复地按下Up和Down光标控制键以升起和降下邮件标记。

(2) 找出提供哪些作用

你对Widget作用将和Widget名称遇到相同的问题：如何找出某个 Widget到底提供哪些作用以及它们能做些什么？同样地，没有一个完美的解答。但有一个合理的方法来处理：

1) 查看应用程序的联机帮助。大多数的应用程序有它们自己专门的作用文件。例如： xbiff有一节叫做ACTION，而xterm 有两节关于转换和作用的文件 —— KEY TRANSLATIONS 和 KEY/BUTTON BINDING。

2) 最初的联机帮助可能给你提示，或甚至直接告诉你它用到何种 Widget的类，所以你可以查看它的Widget集文件中的特定的Widget。(在 core 版中惟一的Widget集为Athena，所以你在此不易出错)。即使联机帮助未告诉你 Widget的类，当你对系统熟悉之后，你将对一个 Widget是否为标准类型较具有概念，如果还是不行 ...

3) 查看程序的源代码，看看用到什么Widget的类，以及Widget提供了哪些作用。

3. 转换对应作用到一序列事件，不只是单一键

我们已经看到转换使得你设定插入，转换结构也能让你对应这些作用：它可以是单一的键，或是一序列的键，或者是事实上一序列任何的X事件。

让我们继续以xbiff 为例，看看如何转换一序列的键盘字符。例如我们定义字符字符串的转换如下：

```
look   check()
raise   set()
lower   unset()
```

以下为相关的转换表：

```
xbiff*Mailbox*Translation: #override\n\
<Key>l,<Key>o,<Key>o,<Key>k: check()\n\
<Key>r,<Key>a,<Key>i,<Key>s,<Key>e: set()
<Key>l,<Key>o,<Key>w,<Key>e,<Key>r: unset()
```

以此测试之 —— 加载设定和启动xbiff ，将鼠标指针移到窗口内。现在你可借助于输入完整的字符串来升起和降下标记。例如键入五个字符 r、a、i、s、e以升起标记。对xbiff 的两个表有几点值得说明：

- 键的名称可以用不同的方式指定。正常的输出字符直接指定 (如<Key>w), 其他的字符则拼出全名(如<Key>Down)。
- 对字符字符串，你必需一一指定，并以逗点分开(如<Key>l,<Key>o, <Key>o, <Key>k) 。
- 转换可允许相同开头的键，例如 look和lower均拥有相同的开头lo，对转换管理器不会形成问题。

(1) 找出键的名称

找出转换所需的键的名称，最简单的方法为执行 xev ，将鼠标指针移到窗口内，按下你所需的键，则键的名称会出现在括号内字符串 keysym和一个十六进位数之后。例如在xev 的窗口内按下光标控制键DOWN，在其中你会看到：

- (keysym 0xff54, Down)。
- 也就是说，键的名称为Down。
- 你可以在转换中使用任何类型的事件。

到目前为止，我们所写的转换都是对应作用到一个按下的键盘字符。但我们曾说过，转换结构可对应作用到任何事件，而不只于按下键盘而已。可能的事件类型非常的多，在此我们只提及一小部分：

<Key>	按下一个键
<KeyDown>	按下一个键 (只是另一个名称)
<KeyUp>	释放一个键
<BtnDown>	按下一个鼠标按钮
<BtnUp>	释放一个鼠标按钮
<Enter>	鼠标指针进入窗口内
<Leave>	鼠标指针移出窗口外

我们已经使用过按下一个键的事件，让我们绑定xbiff 作用到鼠标按钮以替换之：

```
xbiff*Mailbox*Translations: #override\n\
<BtnDown>Button1: unset()\n\
<BtnDown>Button2: check()\n\
<BtnDown>Button3: set()
```

你可以看到语法和前面相似：你先给定一般性的事件类型 (例如 <Key> 或<BtnDown>), 其后跟着你所需事件的事件细节部分，例如s和Button3 (Button 1、2、3分别对应到左、中、右按钮)。

(2) 对一序列的事件的转换

就如同我们定义了一序列按下键事件的转换 (set、unset和check)，我们当然也可以定义一序列的鼠标事件。事实上你转换的一序列的事件可以任意组合在一起，你可以在一个转换的左边随意混合事件的类型。所以你可以定义如下的转换表：

```
xbiff*Mailbox*Translations: #override\n\
<BtnDown>Button1, <Key>?, <BtnDown>Button3: check()\n\
```

```
<BtnDown>Button1: <Key>u, <BtnDown>Button3: unset()\n\
<BtnDown>Button1: <Key>s, <BtnDown>Button3: set()
```

也就是说，用到check()，你必需依序先按下按钮1 (左按钮)，然后按下“？”键，最后按下按钮3 (右按钮)。这个样例并不是很好，但对于一些危险或不可取消 (irreversible) 的作用 (例如删除一个文件，或是覆写一个缓冲区的内容) ，你可以依照这种方式来使用转换。你需要使用一个非常谨慎的命令序列，才能用到此作用，这样使得用户不可能因意外而输入此命令。

(3) 使用非键盘和非鼠标事件的转换

通常你是对按下或释放鼠标按钮或键盘的键定义转换。但我们曾经说过，你可以对任何事件设定转换，例如鼠标指针移入或移出一个 Widget的窗口。让我们以xman的主选项窗口为样例来解释它。这是一个相当人为的样例，因为它没有任何用途。但无论如何，它很容易被看出在做些什么操作。

查看xman的联机帮助，在X DEFAULTS标题下，你将看到概括的xman所用到的Widget的名称和类：主选择项窗口Widget的名称叫topBox，类名为Command。这是一个好的猜测，因为在菜单操作框的方法。我们可用上面提过的技巧来确认它，使用以下的命令：

```
xman -xrm "*Command*backgroundPixmap: scales"
```

这和我们先前的样例有一个重要的不同：我们所用到的作用不是由特定的应用程序指定，而是由标准的Widget提供 (本例中为Command Widget，在“X 工具箱Athena Widget”使用联机帮助中有描述)。

在我们定义任何东西之前，先来看一看此Widget缺省的功用，以便我们能够了解有些什么事发生和有哪些Widget的作用会做。启动xman，移动鼠标指针进入 Help 框，你会看到框的外框变成高亮度显示 ，这是 highlight() 在作用。将指标移出，框的外框恢复正常，这是 unhighlight() 作用。将鼠标指针再度移入 Help 框，按下一个鼠标按钮，保持按住不放。则框内的颜色反转 (框内的文字变成缺省的背景色，而原来窗口的背景变成窗口的前景色)，这是set() 在作用。继续保持按住鼠标按钮，将鼠标指针移出窗口外，框内颜色恢复正常，这是 reset() 在作用。一个正常“按一下”(clicking on) Help框的次序为：

1) 移动鼠标指针进入框中：highlight()将外框变为高亮度显示。

2) 按下按钮：set() 反转框中的颜色。

3) 释放按钮：notify()开始作用，造成程序建立求助窗口 (help window)。在进行中时，框的颜色保持反相。当窗口建立完成之后， reset()反转框内的颜色为正常，但外框仍保持高亮度显示。

4) 将鼠标指针移出窗口：unhighlight()将外框恢复正常。

现在你了解了有哪些作用，我们将定义一些转换来改变原先进出窗口的作用：

```
*Command*translations: #override\n\
<EnterWindow>: reset()\n\
<LeaveWindow>: set()
```

用这个奇怪的转换表，当你一开始移动鼠标指针进入框中，什么事也不会发生，但当你移出鼠标指针时，颜色会反转。如果你再度移动鼠标指针进入框中，颜色会变回正常。其他的作用和前述相同。

(4) 使用修饰键来修饰事件规范

有时你指定的转换希望能同时按下一或多个修饰键，例如你要对应一个作用到和 META键同时按下的一个键，或是当Ctrl和Shift 同时按下的鼠标按钮。到目前为止我们还没有任何办法可指定这样。我们不能用事件序列完成这点，因为它是依序定义的，而我们需要的是指定同时，

例如“按下X键且ctrl键同时被按下”。

欲在转换中指定修饰键，你只需在事件名称之前加上你所需的修饰键名。例如在 xterm 中，定义meta-i为“贴”上一次“剪”的文本，使用：

```
*VT100*Translations: #override\
Meta <Key>i: insert-selection(PRIMARY, CUT_BUFFER0)
```

因为这种修饰键／事件类型的组合十分常见，转换管理器允许使用一种缩写的形式。相等于上面第二行的写法为：

```
<Meta>i: insert-selection(PRIMARY, CUT_BUFFER0)
```

我们可以对鼠标事件做同样的处理。让我们对 xedit 定义转换，使得使用鼠标可以在文本上实用地移动，我们首先的尝试如下：

```
*Text*Translation: #override\
Shift <Btn1Down>: forward-character()\n\
Shift <Btn2Down>: forward-word()\n\
Shift <Btn3Down>: next-line()\n\
Ctrl <Btn1Down>: backward-character()\n\
Ctrl <Btn2Down>: backward-word()\n\
Ctrl <Btn3Down>: previous-line()
```

如果你测试它，奇怪的现象会发生——光标好像会自行其是，而且文本段会一下子被选择，一下子又取消选择。发生这种现象的原因是 Text Widget 的缺省对应仍然会作用，它包含的转换像：

```
<Btn1Up>: extend-end(PRIMARY, CUT_BUFFER0)
```

你可能认为这不会影响你，因为当你释放按钮时你总是按着 Shift键或Ctrl键。但事实上会作用：转换管理器对于你未定义的修饰键解释为你不在乎它们的影响，所以释放 Button1 时会对应到上述的规范。为了克服这点，我们对那些可能不小心便会发生的按钮释放事件定义转换，并对应绑定到一个空作用。这些转换当被对应到时会盖掉缺省的转换。对使用 Text Widget 我们需再增加两行，才是一个完整的转换表：

```
*Text*Translation: #override\
Shift <Btn1Down>: forward-character()\n\
Shift <Btn2Down>: forward-word()\n\
Shift <Btn3Down>: next-line()\n\
Ctrl <Btn1Down>: backward-character()\n\
Ctrl <Btn2Down>: backward-word()\n\
Ctrl <Btn3Down>: previous-line()\n\
Shift <BtnUp>: do-nothing()\n\
Ctrl <BtnUp>: do-nothing()
```

这说明了下列几点：

- 我们对鼠标事件使用了缩写的语法，也就是先前的语法像 <BtnDown>Button1以<Btn1Down>替换。转换管理器容许一些缩写的语法存在。我们在前面看到的 <Meta> 也是一例。
- 我们用 do-nothing() 当作一个哑作用，就好像它是列在 Text Widget的文件中一样。事实上这个作用是不存在的，因此会导致错误的信息出现，但因为我们本来就是要用它来什么事也不做的，所以无需介意。
- 对于我们方才指定的哑作用，我们用了一个事件 <BtnUp>便代表了三个按钮。相同地，转换管理器把从缺的修饰规范的解释为“对任何”，在一个事件中缺少细节部分 (例如在规范“<BtnUp>Button1”中“Button1”的部分) 解释为“对任何所有的细节部分”。

这点在转换中有一个非常常用的形式为：

```
<Key>: ...
```

因为缺少细节部分，所以可被用于所有按下键 (key-press)事件，也就是对所有的键。事实上在Text Widget 上有一个缺省的转换为：

```
<Key>: insert-char()
```

insert-char()作用的功能为当一个键被按下时，插入相对应的 ASCII字符。

4. 复合的转换表及例子

到目前为止，我们把所有的转换均应用于整体的 Widget类。但你能对个别的Widget指定转换，就如同资源一般。在此我们将对 xman定义更多的转换。我们将对Help框Widget (对应作用到助忆符)只用到键盘事件，对 Quit框只用到窗口事件。为了达到此点，我们将对转换应用到的Widget 给予明确的名称。我们的转换表如下：

```
*Help*translations: \
<Key>h: highlight()\n\
<Key>u: unhighlight()\n\
<Key>n: notify()\n\
<Key>s: set()\n\
<Key>r: reset()\n\
<Key>LineFeed: set() notify()
Quit*translations: #override\n\
<EnterWindow>: reset()\n\
<LeaveWindaw>: set()
```

有几点特别的语法需要注意：

- 在此我们对相同类中不同的 Widget指定不同的转换，所以我们需要知道实例名称。不幸的是，这些实例名称 (Help、Quit、Manual Page)并不明显。如果它们在文件中找不到(本例即找不到)，那你只能用猜的或是去查看原始程序了。
- 对于Help，我们省略了常用的#override，因为我们对此Widget不需要考虑任何缺省的绑定。特别的是，当鼠标指针进入窗口时，我们不要此 Widget呈现高亮度显示，这样我们才能看出这个转换的效用。
- 由于省略 #override，我们将这个转换规范移至第一行。如果不这么做，而且对第一行仍以\n\ 作结束，我们将得到错误：

```
X Toolkit Warning: translation table syntax er-
ror: Missing ':' after event sequence.
X Toolkit Warning: ...found while parsing ''
```

因为\n是用来区隔转换规范或类似像 #override 指令的。而将此行和第一个规范以隐藏的新行字符区隔，就如同：

```
*Help*translations: \
<Key>h: highlight()\n\
...
```

- 对LineFeed那一行的转换，包含了复合的作用，和前面 xterm 中复合的string()作用类似。

29.5.2 转换 —— 格式和规则

转换是一个由工具箱提供的一般性结构，它让用户指定当某些特定的事件由 Widget接收到时，一个Widget应完成何种作用。工具箱中处理转换的部分被称之为转换管理器。

转换由Widget指定，它的确是一个Widget的每一个实例。一个转换的集合称之为一个转换

表，而这个表借助于标准的资源结构传递给应用程序。Widget 会有一个Translation 类的资源属性，通常的实例名称为 translation。这个转换资源期待的一个值，即一个转换表。就像所有其他的资源一般，你可以在同一个应用程序对不同的 Widget指定不同的资源，而且你能以类名称或实例名称或二者混合来指定它们。

每一个Widget定义了它所提供的作用，不论是在数量或类型上，它们都是极富变化的。

转换可被各种不同类型的事件指定，不仅只于键盘和鼠标事件而已。任何序列的事件均能被处理，就如同单一事件一般。

1. 转换表的格式

一个转换表大体上的格式如下：

[optional-directive\n] list-of-translations

每一个 list-of-translations 由一或多个转换组成，格式如下：

event-sequence : list-of-actions

当event-sequence发生时，规范中的list-of-actions 会由Widget所完成。如果在一个表中，有多于一个的转换，每一个需以“ \n ”区隔开。

2. 转换伪指令 —— #override 等等

选项伪指令告诉转换管理器，它应对任何已设定之相关 Widget在此转换集合中应如何处理。

#replace : 清除所有现存的对应，只采用在转换表中所含有的 (只使用新的)。

#override : 强制留下现有的对应，加入转换表中。如果在表中有任何项目设定，旧有的即被覆写。也就是说，旧有的被新有的替换。结合旧有的和新的，但新的比较重要。

#augment : 强制留下现有的对应，加入转换表中。如果在表中有任何项目设定在现有的设定存在，使用旧的而忽略新的(结合旧有的和新的，但旧的比较重要)。

如果未设定伪指令，缺省为 #replace 。

3. 个别的转换规范格式

每一个转换的格式为：

event-sequence : list-of-actions

让我们来看一看此规范的两个部分。

(1) 事件和事件序列的格式

一个事件序列包含一或多个事件规范, 其格式为：

[modifiers] <event-type> [repeat-count] [detail]

除了事件类型外，均为可选择。(<>中为必需)。

modifiers : 这是基本设计中比较精巧的部分，我们在下一段说明。

event-type : 指定我们有兴趣的事件的类型，例如按键 (<KeyDown>)、释放按钮(<BtnUp>)或鼠标指针离开窗口(<Leave>)等等。

detail : 指定我们有兴趣的特定类型。如果你省略细节栏 (detail field)，事件规范将对应到任何detail，这样，<Key>将对应到所有的按键事件。此格式指定到每一个事件类型。对指定事件类型的细节栏为：

- 对<Key>、<KeyUp>和<KeyDown>事件，细节如果不是键的名称 (例如<Key>s)，便是keysym (keysym是按键以开头为0x的十六进位数表示，将于下一节详细解释)。
- 对于按钮事件，细节就是按钮的名称，也就是 Button1到Button5 中的一个。例如我们先前使用过的<BtnDown>Button1。

类型/细节的缩写：常用于转换管理器的一些事件类型和细节的组合，允许你对它们使用

缩写：

缩写	相等的全名
<Btn1Down>	<BtnDown>Button1
...	
<Btn5Down>	<BtnDown>Button5
<Btn1Up>	<BtnUp>Button1
...	
<Btn5Up>	<BtnUp>Button5

repeat-count：这指定了事件需要的次数。如果指定，它们包含在括号之中。例如：

```
<Btn1Down>(2)
```

指定需对一号按钮(button-1)按两次。如果你在后面再加上加号 (+)，其意为按的数目需大于或等于指定。例如：

```
<Btn1Down>(3+)
```

意为需按三或更多次。缺省的重复次数为一次。

一个事件序列以一或多个事件规范组成，以逗点分开。当这个事件的序列在其 Widget发生时，相关的作用便会运作。

当序列发生时，转换管理器会根据一些规则决定它自己是否被满足。我们用一个例子以便仔细地观察，假设你对两个字符序列set 和unset定义了转换：

- 简单地说，如果个别的事件依序发生，转换管理器会被满足，其他的事件 (那些你未指定的事件) 如果在指定的序列中间发生，不会妨碍序列被满足。例如， set 可被sweat 和serpent 对应。
- 如果介于其间的未指定事件，启动了转换表中的另一个事件序列，转换管理器会放弃原先的序列，而尝试着去满足新的序列。例如， set 不会被sauerkraut对应，因为u 会使得转换管理器对应到unset。
- 如果在一个事件的集合中有超过一个的事件序列发生，转换管理器只会应用到一个转换：
- 如果一个序列对应到结束 (右端)，较短的那个序列只有在不包含于较长的序列才会发生。所以如果unset 被对应到，对set 转换将不会作用。
- 如果一个序列是在另一个序列的中间发生，例如，如果你定义序列 at和rate，则较长的那个永远不会被对应到。

(2) 事件修饰键

修饰键是一些键或按钮，系指当主要事件发生时，那些必需被按下才会让转换管理器满足的键或按钮。你可以对键、按钮、移动、进出窗口等事件指定修饰键。常见的修饰键为：

```
Button1  ...Button5
Ctrl  Shift  Meta
Lock
```

如果未指定任何的修饰键，转换管理器会解释为：“ 当事件发生时，不论修饰键是否被按下，均会被接受 ”。例如，<BtnDown>会被满足，不论当时Shift 或Meta键是否有被按下。

如果真的需要指定 “ 只有在没有修饰键被按下时才接受此事件 ”。则需使用伪修饰键None。例如， None <BtnDown> 会使得当按钮按下时若META键也被按下则不会满足。

对一个事件指定一些修饰键意为 “ 只要符合转换中指定的修饰键，其他的修饰键不需介意 ”。它并没有 “ 一定要完全恰好符合才可以 ” 的意思。例如， Ctrl <Key>a 在你按下 meta-Ctrl-shift-a 时仍会被满足。

如果你真的要指定 “ 只有刚好符合修饰键的才要 ”，则需要在修饰键之前加一个惊叹号 (!)。

例如，!Ctrl <Key>a 在你按下 meta-Ctrl-shift-a 时不会被满足。

对一个修饰键的集合 (可能是空集合) 作限制，意为“ 除了这些修饰键不接受 ”，需要在不接受的修饰键之前加一个 (~)号。例如， Shift~Meta <Key>t会被Ctrl-shift-t满足，不会被meta-shift-t满足。

键事件通常忽略大小写，如果你要区分，需在之前加一个冒号 (:)。例如，不论 H或 h均可符合<Key>H，但只有H 才符合 :<Key>H。

就如同对常用的事件类型 /细节配对有缩写一般，转换管理器对常用的修饰键 /事件类型配对同样地提供缩写：

缩写	相等的全名
<Ctrl>	Ctrl <KeyDown>
<Shift>	Shift <KeyDown>
<Meta>	Meta <KeyDown>
<Btn1Motion>	Button1 <Motion>
...	
<Btn5Motion>	Button5 <Motion>
<BtnMotion>	任何按钮的 <Motion>

(3) 作用的格式和作用的表列

每一个转换在一或多个作用之上对应一个序列的一或多个事件。在表列中的个别作用是以空白分开的。不可用逗点分开，那将会导致错误。

个别的作用格式如下：

```
action-name(parameters)
```

即使没有参数被指定，在作用名称 (action-name)后的括号，仍然不可省略。

例如：

```
start-selection()
```

如果在作用名称和左括号中间留有空白，你将会得到一个错误。

作用名称只包含了字母、数字、美元号 ($)、底线(_)四种字符。每一个 Widget提供它自己的作用集合 (如果有的话)，且自我包含这些作用名称的硬代码列表。

参数是一个零到多个字符字符串的列表，中间以逗点分开。参数的意义为对特定的作用作指定 (事实上大多数的作用并没有任何参数)。参数字符串可以不加引号，例如：

```
insert-selection(PRIMARY)
```

或者前后加上双引号，这种情形通常为参数字符串内包含了空白或一个逗点，例如：

```
string("plot<x,y>")
```

没有一个一般性的方法，让你在参数字符串中的任何位置包含一个双引号，虽然像这样 string(ab"cd) ’ 将双引号放在字符串中间是可以处理的。也没有一般性的方法在同一个参数字符串中同时包含字符串和双引号。因为这样，有些 Widget在解释它们自己的参数时，可以自行加入它们自己的语法规则。例如：对 xterm 的VT100 Widget的 string()作用，如果一个不带双引号且开头为“ 0x ”的字符串，此字符串被解释为代表一个 ASCII 字符的十六进位数。

现在结束我们对转换规范及格式的描述。由此，你应有能力了解在不同 X联机帮助列出的转换，且可写你自己的转换。为了帮助你，下节列出你常见的问题，以及如何克服它们。

29.5.3 转换规范中常见的问题

转换在观念上简单，但实际上很混乱。即使你常常使用，语法仍然复杂而难解。无论如何，如果你是初学者，最好的方式是你以别人的转换当作自己的转换的基础。在联机帮助中有几个

对xbiff、xdm、xterm 的转换样例，将对你有所帮助。

如果你发现转换有错误的话，有几点值得去检查：

- 转换只能应用在使用工具箱的程序上。如果你试图对非工具箱应用程序定义转换，看起来不会有任何问题，只是转换不会作用而已。

让我们来看一下为什么，以对xcalc (这是一个非工具箱程序)使用转换为例。你对一个资源名称像 *xcalc*translations定义一个转换表，且用xrdb加载至你的数据库。xrdb并不会抱怨，因为它不知道是那一个应用程序使用到资源。它只会设定数据库，稍后供资源管理器查询。现在你执行xcalc ，它对转换是一无所知，所以不会向数据库查询转换，当然也绝不会编译它们了。

- 不要省略 #override，除非你确实知道你要做什么。如果你因错误省略它，例如在xedit 中，你将发现没有任何的键可输入任何的东西 (因为缺省的转换“ <Key>:insert- char() ”被去掉了)。
- 检查你的每一行均有结束符。如果你在转换表中的一行忽略了“ \n\ ”或“ \n ”，在其后所有的转换都会被忽略。如果你在最后一行的末端加上一个倒斜线 (\) ，或是省略了文件中最后一个新行字符，整个转换表都会被忽略 (不过这是xrdb的问题，而非转换管理器的问题)。这种错误在编辑现存的转换表时特别容易发生。
- 当你定义的转换和缺省有冲突时，可能会导致奇怪的行为，特别是对鼠标按钮事件，每一次按下或Down事件，会相关到一个释放或 Up事件，当你对此部分没有明确定义时，可能会有一个缺省的对应仍然存在(键盘的按下和释放也是成对的事件)。所以：

1) 检查缺省绑定的文件。

2) 如果你只对按下/释放配对的一半指定一个转换，确定另一半并非缺省转换的一部分，如果是的话，需对它明确地指定一个转换。

3) 如果你仍然不能解决，暂时由表中移去#override，这将去掉所有的预设转换，让你了解问题是由于和缺省转换冲突所造成，还是因为你的转换表有错误。

- 转换管理器对语法不正确的问题，无法很好的告诉你原因何在。例如如果你有一个转换像：

```
<Key>F6: string("abc""def")
```

参数的语法并不正确， F6键将没有作用，但你也看不到错误信息。

- 如果你转换一序列的事件，且需要对每一个均指定修饰键，你必需明确地对每一个都指定。例如，如果你需要一个转换使用Ctrl-X Ctrl-K：

```
Ctrl <Key>X, Ctrl <Key>K:...
```

而如果使用：

```
Ctrl <Key>X, <Key>K: ...
```

你的指定为 Ctrl-X K

- 检查你所需的Widget是否有你指定的名称和类。例如对 xterm ，你可以在一个表的开头指定：

```
xterm*Text*translations:
```

这将什么事也没作， xterm 正规窗口Widget的类VT100。通常，不论xrdb或转换管理器均不会有反应，因为看起来没错。

- 转换可能指定正确，也可以工作，但它的作用和你预期的不符。例如对 xterm 的转换：

```
Meta Ctrl <Key>m: mode-menu()
```

是正确的，且会工作。但mode-menu()实际上检查鼠标左或中按钮是否有招唤它，其他方面不做任何事。

- 在一个转换中不指定修饰键，并不意味着当修饰键按下时转换会无效。它真正的意义为：

“我并不在乎有没有修饰键”。如果需要的话，使用“None”，“”或!符号。使用时要小心缺省的转换是否会妨碍到你。

- 转换是针对Widget而指定的，所有在转换中的作用必需由Widget提供。在你指定转换资源名称的地方很容易忘掉这一点。例如：

```
xman*translations: \
<EnterWindow>: reset()\n\
<LeaveWindow>: set()
```

将导致许多错误：set()和reset()作用只有被Command Widget定义，但xman有数种其他类型的Widget可接受转换，且转换管理器会抱怨这些 Widget并未提供set()和reset()。解决的办法为更完整些的指定资源名称，例如在本例为 xman*Command*translations 。

- 对任何给定的资源，当资源数据库被询问时，资源管理器会传回一个值给 Widget。这个传回的值的“特征值”(资源名称)大多与Widget的和属性的完整类/实例名称相符。所以你对所有的Text Widget指定一个一般性的转换后，又对xedit指定一个转换，希望它们并存是不可能的，只有一个转换表会传给Widget。例如：

```
*Text*Translation: #override\
(对Text一般性的转换)
...
 xedit*Text*Translation: #override\
(对xedit 的Text特定的转换)
...
```

你只能得到在xedit 中特定的转换，或是在别处得到一般性的转换。

#override 会有所混淆，它的意义为“把转换加入现存之中”。但这完全由转换管理器处理，当时候到时，转换管理器会决定传递哪个值给由资源管理器所造的Widget。对资源管理器而言，#override只是传递给Widget值的部分中的一个文字字符串而已。

因为你使用资源来指定转换，所以错误可能在两个领域均会发生。为了减少错误的范围，当你对转换颇有经验时，在你已加载转换资源之后，最好能明确地打输出你的资源数据库。例如：如果你对xprog 写入转换，且转换在文件mytrans 中，以下列命令来执行程序：

```
xrdb mytrans ; xrdb -q ; xprog ...
```

29.6 键盘和鼠标——对应和参数

在前节我们看到了工具箱所提供的转换结构，它让你对于一个应用程序的个别实例，定制你的键盘和鼠标。在本节，我们来看另一种较低层次的定制，它是由服务程序管理，称之为映射，你只需要告诉服务程序你的键盘所需的不同的配置，它就会被每一个连接到你服务器上的应用程序应用到。例如：替换通常的 QWERTY键盘，你可能希望重新安排键盘以适应那些对键盘并不熟悉的用户 (你可能把键盘按ABCDEF..）重新排过，当然键盘按钮上所印的字也需更改成相符)。你也能对一些Control、Shift等等的修饰键作指定。对鼠标按钮，一样有一个相关的映射，可将“逻辑的”按钮映射到实际动作。总而言之，你使用这些键盘和鼠标的映射的频率，将小于转换。

此外，尚有非常常用的第三种类型的定制可用：你可以设定有关你键盘和鼠标各种不同的参数。例如响铃声音的大小，按下键时是否有滴答声等等。

29.6.1 键盘和鼠标映射—— xmodmap

服务器本身处理一个层次的定制，它对于所有使用到此服务器或显示器的应用程式均发生

效用：这就是键盘映射。

每一个键，有一个单独的码映射它，称之为键码。键和键码之间的关系是绝对固定的 (粗略来说，你可以说“键码就是键”)。

连接到每一个键码 (或键)的是一个keysym的列表。一个keysym是一个代表印在键盘符号上的数字常数。在缺省的情况，大多数的键只有一个 keysym 与之映射，例如 Shift、A、B、Delete、Linefeed等等。keysym既非ASCII或EBCDIC字符，也非服务器用以维持keysym和字符的关系。你可以对每一个键有两个 keysym。在缺省映射中，有很多连接到两个 keysym的键，例如冒号(:) 和分号(;)，7 和 &等等。对一个键附属的keysym列表中，第一个keysym是未按下修饰键的状况下的键。第二个keysym是指当Shift(或Lock)已被同时按下时的键，如果在列表中只有一项，且为字母，则系统自动假设第二项为相对的大写字母。超过两项的 keysym并没有特别的意义，键盘和keysym之间的关系被称之为键盘映射。

应当尽量地使用服务程序处理一般的键和 keysyms 。它对键码没有附属意义，且它自己本身不会使用映射从键码映射至keysyms ：它只是传递信息给客户应用程序。特别的是，服务程序对ASCII 或其他的字符集合毫无概念；它只是说明“某键被按下，某修饰键也同时被按下，keysym列表中某keysym和某键相关”。它是客户机程序 (典型的使用标准的 X Library) 对keysym和修饰键附属的意义：例如，它决定如果 keysym产生时ctrl也被按下，它必需被解释为ASCII 字符 hex 0x1，也就是说Ctrl-A。特定的客户机可以决定特殊的修饰键的意义；例如在xterm 中，当你和MTEA键同时按下一个键，程序将此转换为ESC 后面跟随着被按下的字符(也就是说，如果你按下meta-A，实际上会产生两个字符 ASCII 0x1b, ASCII 0x41)。

服务程序在此领域内提供一个额外的设施。你可以定义让服务程序将键码解释成修饰键，例如“当键码为若干的键被按下时，它相同于 CONTROL修饰键被实际按下”。这种定义并不互斥：如果你定义键F7为Shift 修饰键，它并不会影响任何现存的修饰键。此种设施称之为修饰键映射。 X提供八个修饰键： Shift、Lock(caps-lock)、Control、Mod1到Mod5。习惯上，Mod1被解释为Meta。

最后，对鼠标按钮有一个类似的鼠标指针的映射。对每一个实际的按钮，你可以对它们指定一个相关的逻辑按钮数字。

实际上，如果你改变你的键盘或鼠标的映射，你相当于是说制造厂商对你的输入设备配置不当，你将把它修正为适合你所需要的。当然，如果你改变了映射，你应该把映射键上面所印的符号也随之修改；不过，通常更改的都是一些控制和修饰键，所以就不是那么需要了。换句话说，如果你改变了映射，使得键盘配置和一个特定国家标准 (例如：法国或德国) 相符，你必需更换实际键盘上的符号。

你可以想象得到，改变键盘映射是一件相当稀罕的事，你可能设定它一次之后就不再改变它。在以下几节，我们将很快的看一看如何使用程序 xmodmap，查看现有的映射和修改它们。

1. 查看现有的映射

你使用xmodmap 来列出现有的映射，就如同改变它们一样。你可以指定不同的命令行选项，来选择想要输出的不同的映射：

- 列出现有键的映射：指定 -pk选项。
- 列出现有修饰键的映射：指定 -pm选项 (或是什么选项也不选，因为这是xmodmap的缺省作用)。
- 列出现有鼠标指针 (按钮)的映射：指定 -pp选项。

例如，将所有的映射一起输出，使用命令：

```
xmodmap -pm -pk -pp
```

当xmodmap 用来改变或设定映射，它可以处理一或多个表达式的作用。你可以把这些输入在一个文件中，假设此文件名称叫myfile，可用下列命令两者之一：

```
xmodmap myfile
xmodmap - <myfile
```

第二行的短横线是必需的，如果少了它，程序将只完成缺省的作用 (列出修饰键的映射) 。除了在文件中输入规范之外，你也可以在命令行中用 -e 选项直接指定它们：

```
xmodmap -e expression
xmodmap -e expression-1  -e expression-2
```

为了得到更多有关xmodmap 作用的信息，可以指定冗赘选项，-v 或-verbose。你可以借助于使用 -n 选项不实际的改变映射而获得相同的打印输出 。(此功能和UNIX中make命令的 -n 选项相同，其意为“ 假装执行我要求你做的事，正确告诉我你将如何进行，但并不实际完成作用 ”)。这个选项对新手或不确定自己是否做的正确的情况非常有用。

每一个表达式的语法并不相同，但一般性的格式为：

```
keyword target = value(s)
```

(等号的两边均需为空白)。

2. 改变鼠标指针的映射

鼠标指针的映射是一个逻辑按钮数字的表。 (逻辑的button-1我们称为LEFT，逻辑的button-2 称为MIDDLE等等，实际的 button-1 是鼠标左边的按钮，button-2是隔壁的按钮等等，所以缺省的逻辑的按钮和实际的一致。)在表中的第一个项目是逻辑的按钮和实际的 button-1 的关系，下一个则是对实际的 button-2的关系，以此类推。例如，颠倒按钮的次序，使用命令：

```
xmodmap -e "pointer = 3 2 1"
```

结果按下鼠标右边的按钮，会被解释成LEFT。

3. 改变键映射

xmodmap 让你将一个键 (也就是说键码) 链接到一个新的keysym表，使用表达式：

```
keycode keycode = keysym-1 [keysym-2 ...]
```

安排keysym-1链接到键时没有修饰键，当Shift按下时keysym-2链接到键，如果还有下一个keysym的话，对keycode 而言是第三顺位等等。(请记住，在前两个之后的keysym，系统并未附属特别的意义，应用程序如果需要的话可以附属意义)。

让我们举一个实际的例子。一些键盘把一些非字母数字键放在不标准的地方，所以我们假设你要将 F6键重定义当没有修饰键按下时为“ 9 ”，当Shift 按下时为“ (”。要写入这个xmodmap 的表达式，你需要知道三件事： F6的键码和“ 9 ”与“ (”的keysym。我们在第１２节提到过，执行xev 便可获得这些：分别按下“ F6 ”,“ 9 ”,“ (”三个键，你便可得到它们的键码和keysym。然后将它们放入你的表达式中。例如在我们的系统中我们使用命令：

```
xmodmap -e "keycode 21 = 9 parenleft"
```

为了容易一些，你通常不需要查问键码， xmodmap 允许你使用下列格式：

```
keysym target-keysym = keysym-1 [keysym-2 ...]
```

它的意义为“ 附属在此键的keysym列表现在改由target-keysym来附属 ”。例如针对我们方才的样例，我们可以用：

```
xmodmap -e "keysym F6 = 9 parenleft"
```

如果将相同的 keysym附属到数个键， xmodmap 会搞混掉，像这种情况你应坚持使用keycode 这种符号表示法。

4. 改变修饰键映射

在服务器中修饰键映射是一个表的集合，每个修饰键有一个表。对一个修饰键的表中，包含了所有当此修饰键被按下时会有意义的键 (键码)。xmodmap允许你在一个表中增加项目，去除项目，或完全清除一个表。对此三个操作的格式为：

```
add modifier = list-of-keysyms
remove modifier = list-of-keysyms
clear modifier
```

不幸的是，语法有点儿混淆，因为替换你所需的键码，你必需指定 keysym附属到键码。

举一个例子：假如你需要在你键盘的右边有一个第二个的 Ctrl键。在我们的键盘上有一个 Alternate 键没有被用来做任何事，所以我们将修改它，命令为：

```
xmodmap -e "add Control = Alt_R"
```

为了多解释一些情况，让我们假设你没有一个多余的键，但有一个第二个的 Meta键在键盘的右手边，而我们要用它。我们首先必需去除它的Mod1映射 (你必须使用Mod1，Meta没有用)，而后将它加入 Control 映射。(如果有需要的话，我们可以拥有双重的映射，所以在 Control-Meta组合键时才会有作用，在一些编辑器中常会用到)。命令为：

```
remove Mod1 = Meta_r
add Control = Meta_r
```

将上述命令行放入一个比方说叫mymaps的文件中，执行命令xmodmap mymaps。它可以工作，但如果你用xmodmap -pm 去查看，你会发觉Control 和Meta混合在一起，所以最好改变键上的keysym为：

```
remove Mod1 = Meta_R
add Control = Meta_R
keysym Meta_R = Control_R
```

在xmodmap 的联机帮助中，有几个更多的交换修饰键的样例。

注意 当增加一个键到修饰键映射，keysym只是用来指定xmodmap 中的键。它完全是 xmodmap 本地的，且只是一个符号而已：只有当相关的键码传递到服务器，才实际上的改变映射。同样地，keysym 和 keycode 表达式对修饰键映射绝对没有影响。一个常见的错误是执行下面这个命令：

```
xmodmap -e "keysym F1 = Contrl_R"
```

这个命令期望F1键能像一个control 键般作用，但它不会。因为你相当于告诉系统“我已经把这个符号印在F1键上面”而已。你应该这样作：

```
xmodmap -e "add Control = F1"
```

如果你合并上一行的命令会使得映射表行看起来清楚些。

我们对不同映射的处理的描述到此结束。

29.6.2 键盘和鼠标参数设定——xset

最后我们来看一看对键盘、鼠标和屏幕设定不同的参数。这些参数使用 xset程序 (我们曾经用来控制服务程序的字体搜索路径) 来设定。在以下的叙述中，我们只用一组参数来演示 xset，但你可以同时指定多组不同定义的设定。

1. 控制终端机响铃

用xset你可以让铃声响或不响，设定它的音调和它持续的时间 (假设你的机器提供这些操作)：

让铃声不响	xset -b
	xset b off
让铃声能响	xset b
	xset b on
设定铃声的音量	xset b vol
(最大音量的vol%)	例：xset b 50
设定铃声的音量和音调(单位Hz)	xset b vol p
	例：xset b 50 300
设定铃声的音量，音调	xset b vol p d
和持续的时间(单位百万	例：xset b 50 300 100
分之一秒)	
控制键的滴答(click)	
让键的滴答不作用	xset -c
	xset c off
让键的滴答作用	xset c
	xset c on
设定滴答声的音量	xset c vol
(最大音量之vol%)	例：xset c 50
控制键的自动重复	
让键的自动重复不作用	xset -r
	xset r off
让键的自动重复作用	xset r
	xset r on

2. 鼠标参数

鼠标指针在屏幕上的移动和鼠标的移动是成比例的。加速值是应用在鼠标指针移动上的一个乘数，例如如果加速值是 4，当你移动鼠标时，鼠标指针将以正常 4倍的速度移动(如果鼠标指针正常时移动n 个像素，现在则会移动 4×n个像素)。

当你希望在屏幕上将鼠标指针移动一段长距离时，相当高的加速值是很实用的，但当你要作一些细部的伪指令时，它看起来就很笨拙 —— 鼠标指针看起来在来回跳动。

为了克服这点，服务程序提供了一个阈值：如果当鼠标指针一次移动超过阈值个像素，加速值也会被带进来执行。

设定鼠标的加速值到a	xset m a
	例：xset m 5
设定加速值为a，设定阈值到t	xset m a t
	例：xset m 5 10

3. 控制屏幕保护程序结构

屏幕保护程序是一种设施，它的目的是降低一个固定的图案老是在屏幕上出现的机率。它的原理是屏幕损害大都起因于让系统闲置很长时间，所以屏幕保护程序在一段特定的时间内如果没有输入动作后，不是整体性地闪动屏幕，便是显示一个不同的图案。

如果你选择的是显示一个不同的图案，根窗口的背景涵盖整个屏幕，一个大 X 的光标出现在屏幕上，且会移动。当大 X 光标在移动时，会改变大小，而且背景也会随机的变动。 (在背

景图案较小时你可能不会注意到，但若比较大时，你可以看到它在跳动。)

当屏幕保护程序结束作用后，如果要花许多时间才能重画应用窗口，你可以指定只有在重画屏幕而不需产生任何窗口暴露事件 (也就是不必要求应用程序重画它们自己的窗口) 的情况下，屏幕保护程序才会作用。这只应用于显示不同的图案的情况，整体性的闪动屏幕纯为硬件作用，不会影响到应用程序。

让屏幕保护程序能作用	xset s
让屏幕保护程序不能作用	xset s off
用屏幕闪动的方式	xset s blank
只有在无曝光事件下才作用	xset s noexpose
允许有曝光事件下仍然作用	xset s expose
用不同图案的方式	xset s noblank
当系统闲置t 秒后作用	xset s t
	例：xset s 600
每p 秒之后改变图案	xset s t p
	例：xset s 600 10

让我们将这些组合起来，假设我们希望屏幕保护程序在系统闲置 80秒后开始执行，用不同的图案的方式，会话为3 秒，不介意曝光事件是否发生：

```
xset s noblank s 80 3 s expose
```

注意 xset s 并不提供 on 这个值。

29.7 进一步介绍和定制uwm

在27.6节，你学到如何使用uwm 来完成基础的窗口配置工作需求，而能以一个舒服的方式使用窗口。现在我们继续谈窗口，集中于两个主要的范围：

1) 提供特别功能，如：

- 不使用菜单，直接使用鼠标配置窗口。
- 我们尚未描述过的一些菜单选择。
- 编辑现存图标的标题。

2) 定制uwm，包含：

- 对任何你所需的命令定义你自己的菜单。
- 将各种不同的窗口管理器功能对应到鼠标按钮和修饰键 (Shift、Control等等)。

29.7.1 uwm 的新特征

现在我们来讨论一些在先前介绍窗口管理器时，为了保持尽量地简单，而省略的标准的uwm 功能。

1. 不使用uwm 的菜单管理窗口

目前，你仍然依赖着uwm 的菜单来配置你的窗口 —— 移动它们、对它们重定大小等等。如果所有的情况都使用菜单，是相当慢的，所以uwm 提供直接完成它的命令选项。

可以使用鼠标按钮和修饰键，来指定要执行的功能和所要操作的窗口。你现在应该已非常熟悉各种不同的窗口管理器功能和它们如何工作，所以我们将很快地说明如何不使用菜单来选择这些功能。

(1) 移动一个窗口

1) 按下Meta键，保持按住。

2) 鼠标指针位置所在的窗口将被移动。

3) 用右按钮，拖动窗口到新的位置。

(2) 重定一个窗口的大小

1) 按下Meta键，保持按住。

2) 鼠标指针位置所在的窗口将被重定大小。

3) 用中按钮，拖动窗口的外框到新的大小。

(3) 将一个窗口送到堆栈的底部

1) 按下Meta键，保持按住。

2) 将欲被送到堆栈的底部的窗口，按一下左按钮。

(4) 将一个窗口升到堆栈的顶端。

1) 按下Meta键，保持按住。

2) 将欲被送到堆栈的顶端的窗口，按一下右按钮。

(5) 将最底层被遮蔽的窗口升到最上层，你有两种选择：

1) 按下Meta键，保持按住。

2) 在根窗口上，按一下右按钮。

或

1) 同时按下Meta和Shift 键，保持按住。

2) 在屏幕上的任何地方，按一下右按钮。

(6) 将最上层的窗口移到最底层：

作法同Circulate Up，但改为左按钮。

(7) 图标化一个新的窗口

1) 按下Meta键，保持按住。

2) 将鼠标指针位置移至欲被图标化的窗口。

3) 按下Left按钮，保持按住。

4） 拖动图标的外框到你所需的位置。

5) 释放按钮和Meta键。

注意它和Lower 操作程序的不同点，在此你是按下、拖动、释放鼠标按钮，而对 Lower，你只是按一下按钮。

(8) 图标化一个曾经图标化过的窗口

1) 同时按下Meta和Ctrl键，保持按住。

2) 在你欲图标化的窗口上，按一下左按钮。

如果你对先前并未图标化的窗口执行这个操作，或经由资源结构无法取得图标的位置，图标将出现在鼠标指针所在的位置。

(9) 将图标还原为它的窗口 (在窗口原来的位置)：

1) 按下Meta键，保持按住。

2) 在图标上，按一下中按钮。

如果你觉得这些对鼠标按钮功能的结合十分笨拙且不易记忆，别担心，很多人都是这样。有更好的法子，刚才那些只是缺省的设定，你可以完全由自己来配置。在本节的后半部，我们将告诉你如何做。现在我们先来看一看，在标准菜单的一些功能和它们能做些什么。

2. 更多的菜单选择

这是一些我们在27.6节中没有解释的标准的菜单选择。

(1) 让你设定键盘的焦点

也就是说，将键盘附属于一个窗口，所以不论屏幕上的鼠标指针在何处，键盘的输入总是在同一个窗口。一般键盘的输入总是指向目前鼠标指针所在的窗口。

(2) 设定焦点到一个特定的窗口

选择focus，出现手指形光标，在你所欲指定的窗口按一下按钮。

(3) 恢复正常

选择focus，在背景窗口上按一下。

(4) 停止uwm，重新启动它

重新读入配置文件（下节说明）且执行它。在你改变配置文件且希望马上执行新的设定时(否则将等到你重新启动一个新的会话) 使用此选择。

(5) 暂停屏幕上所有的显示

当你要对你的屏幕拍照时可以使用这个选择。

(6) 重新恢复显示

所有的窗口会立即更新。

(7) 中止uwm

当你要删除uwm 时使用，例如在启动一个不同的窗口管理器之前。

(8) Perferences 菜单

我们在上面提过，有两种方法激活 uwm 的WindowOps菜单，在背景窗口上按下中按钮，或在按住Meta和Shift两个键的情况下，在任何地方按一下中按钮。用第二种方法让你调出第二个菜单，只要将鼠标指针移到WindowOps菜单的外边，标头为Preferences 的窗口就会出现。

在Preference中的选择，只是一些xset程序中设定鼠标和键盘的选项而已。

注意 Lock On 和 Lock Off选择是和记录有关的，可能会导致在你的主控台窗口输出一个错误的信息。

3. 改变现存图标上的标题

如果你对同一个应用程序执行数次拷贝后会有缺点，例如有三个 xterm 的图标，你无法明确的区分它们。为了克服这点， uwm 允许你可以编辑图标中的字符串为你所需的任何字符串。(这只能在uwm 自己缺省的图标使用，例如你无法编辑在xclock的特定图标中的字符串。)

编辑在一个图标中的名称方法如下：

1) 将鼠标指针移至所要编辑的图标。

2) 键入你所希望的任何文字。

3) 你可以去掉文字，不论是先前存在或方才才输入的，方法如下：

• 去掉前一个字符：按Delete。

• 去掉整个名称：按Ctrl-U。

29.7.2 定制uwm

uwm 具有高度的可配置性。你可以将整个范围的参数和定义保存在一个配置文件中，当 uwm 启动时会将之读入。我们前节曾经提过，你可以在中途改变配置文件，用 WindowOps菜单中Restart选项，告诉uwm 重新读入它。

1. uwm 的配置文件

缺省uwm 有两个配置文件，其中之一为：

/usr/lib/X11/uwm/system.uwmrc

通常由系统管理员设定，且第一个被读入。另一个

$HOME/.uwmrc

是你自己的配置文件。两个文件均需要存在， uwm 硬性规定了缺省设定。

注意 如果你用不正确的语法设定一个配置文件，当uwm 读入时，你会得到一个错误信息，像：

```
uwm: /usr/nmm/.uwmrc: 38: syntax error
uwm: Bad .uwmrc file...aborting
```

uwm 将不会启动。当在一个新的会话启动时，这没有什么大问题。然而，如果你在中途重新设定uwm ，你可以结束但没有窗口管理器，且没有 xterm ，没有编辑窗口来编辑这个错误的文件，无法启动其他的窗口。如果此种情况发生，你必需从其他的终端机或机器关闭 X，或毁坏你的系统。

2. uwm 的命令行选项

如果你不需要系统配置文件，也不需要任何缺省的设定，你可以借助于 uwm 的命令行选项 -b 抑制它们。

如果你要使用其他的文件，就像两个缺省的配置文件一般，你可以用 -f filename 来指定它。

3. 把功能对应到键和按钮

uwm 让你定义当一个特定的鼠标按钮按下时，有某个功能会作用。例如当你在一个窗口中按一下中按钮，它将被升到堆栈的顶层。这种对应结构和工具箱转换并没有牵连，它完全由 uwm 本身来完成。

为了让这些结构更有用，你可以指定其他的条件来运用更多的功能，或许一个修饰键 (像 Meta) 需被按下，或许作用只发生在鼠标指针位于一个图标上而非应用程序窗口或背景窗口。

用.uwmrc(或其他的配置文件)所包含的对应规范来指定对应。规范的格式和上面的表格类似，就像：

```
uwm-function = modifiers : window context : mouse events
( uwm 功能  =  修饰键  : 窗口的环境:  鼠标事件  )
```

这些元素为：

- uwm 功能：uwm 的内建功能之一的名称。例如功能 f.move即是移动窗口的功能， f.lower 将窗口降低一层等等。这些功能将在下面更完整地描述。

功能名称必需跟随着一个等号(=)。

- 修饰键：在运用上述功能时，当指定的鼠标事件发生时，必需被按下的修饰键表列。正确的修饰键名称为：

  ctrl(或c)，对Control键。

  meta(或m或mod1)，对Meta键。

  shift(或s)，对Shift键。

  lock(或l)，对CapsLock键。

这些名称必需正确地列出。你可以使用 1～2个修饰键，如果你使用两个键，用一个“ |”符号来分开它们。

你可以省略整个修饰键列表 (即此功能映射于鼠标事件发生时并没有修饰键被按下), 但尾端的冒号“ : ”不可省略。

- 窗口的环境：限制只有鼠标指针在屏幕上指定位置的类型符合特定条件时，功能才会发生。正确的环境如下：

window(或w)：鼠标指针必需位于一个应用窗口中。

icon(或i)：鼠标指针必需位于一个图标中。

root(或r)：鼠标指针必需位于根窗口或背景窗口中。

你可以指定任何数目的环境，用“ | ”来区隔它们。如果你没有指定，则功能的发生与鼠标指针位置无关。

- 鼠标事件：何种鼠标事件映射到此功能。指定的事件为一个按钮名称 —— 任何的

left(或l)。

middle(或m)。

right(或r)。

跟随着一个动作：

down : 当按钮被按下时会符合。

up : 当按钮被释放时会符合。

delta : 当按钮被按下且移动超过一定数目的像素时会符合。

所有的这些你已实际使用过它们，在本节开头所描述的一些作用的对应为：

```
f.resize = meta : window : middle delta
f.iconify = meta : icon : middle up
f.raise = meta : window|icon : right down
```

uwm 的缺省对应在文件 $TOP/clients/uwm/default.uwmrc。

4. uwm 的内建功能

uwm 的联机帮助列出可应用的功能。你可以看出，功能是和 WindowOps及Preferences中的选项相关。

然而，有一个有关 pushing窗口(f.pushleft, f.pushup等等)的功能集合你从未见过。pushing的意思为，朝一个特定的方向移动一个窗口，移动的距离固定。与 f.move不同的是，后者以交互的方式指定窗口移动的方向和距离。

缺省f.pushdown对应到同时按下Control和Meta键，且按住中按钮。试它几次，你将发现你的窗口稍微移动了一点 。push 功能对微小移动窗口非常有用。

另一个功能为f.moveopaque。它也移动一个窗口，但不像 f.move，它并不会给你一个指示窗口新的位置的方格，你直接拖动整个窗口本身。这可以让整个屏幕清爽些，但比较慢，且一般窗口移动时会有抖动的现象。

5. 定义菜单

f.menu是一个非常强大的uwm 的功能：它允许定义自己的菜单。此菜单可选用到 uwm 本身的功能，或任何的外壳命令，或一个特定的动作，像是在一个剪缓冲区插入文本。

在配置文件中定义一个菜单共有两个步骤。首先定义菜单上所需的对应，其次定义菜单本身的内容。对应的部分像我们先前所用过的，但在尾端增加了一栏菜单名称。例如 WindowOps 菜单 (借助于在背景窗口中按下中按钮来呼叫)的对应是：

```
f.menu = : root : middle down : "WindowOps"
```

在此，菜单名称既是用以显示菜单出现时的名称，也链接到配置文件中的菜单内容规范。

菜单内容的格式很简单：对每一个选择项，包含了一行当选择项出现在选单的名称和当它

被选择到时所做的动作。让我们观察一个省略的WindowOps定义：

```
menu = "WindowOps" {
New Window : !"xterm &"
RefreshScreen : f.refresh
Redraw : f.redraw
Move : f.move
}
```

从这里，我们可以看到其语法为：

```
menu = "menu name" {
...
selection lines
...
}
```

菜单名称和对应所指定的相同。选择项列包含了选择项名称，分隔的冒号和负责的动作。这些动作为下列三者之一：

1）一个uwm 的功能：只用到它们的名称，在上例为move那一行。

2）一个外壳命令：命令包含在双引号中间 (用外壳的& 语法使其在背景窗口中执行) 且在前面加一个惊叹号。在上例为 xterm 那一行。(如果省略& ， uwm 将被挂起来，等待命令的完成，如果此程序为X的应用程序，它需要uwm 来安排它的窗口，这将会招致麻烦。)

3）一个文本字符串：这将插入到一个“ 剪 ”的缓冲区，而后你可以像平常一样的“ 贴 ”它。

(1) 多种的菜单链接对应到同一个键

通常对一个特定的键 /按钮的组合，只会对应到一个菜单，但可以对同一个键对应有多种菜单：如果在一个菜单中不选择任何项目且把鼠标指针移动到菜单的边上，将得到下一个菜单。你已经实际看过这种例子：在同时按下 Meta和Shift键的情况下按下中按钮，可以得到WindowOps 菜单，然后是Preferences菜单。

对应多种菜单非常容易，只要在定义每一个对应时当作其他的对应并不存在，而在定义菜单的内容时用标准的方式即可。例如uwm 的缺省设定包含了对应：

```
f.menu = meta | shift : : middle down : "WindowOps"
f.menu = meta | shift : : middle down : "Preferences"
```

注意 一个菜单只能定义一次，但可以用它来做任意多次的对应。查看缺省设定，将看到WindowOps 菜单被定义了一次但使用到两次。

(2) 指定菜单的颜色

你可以指定在一个菜单中所用的颜色。对菜单名称标题、每一个选择项、鼠标指针所在的高亮度显示选择项，你都可以指定一个前景色和背景颜色。一个有颜色的菜单的格式如下：

```
menu = "menu name" (head-fg : head-bg : hilite-bg : hilite-fg) {
...
selection-name : (item-fg : item-bg) : action
...
}
```

以下为一个混合的样例，使你的WindowOps 能拥有更多的颜色：

```
menu = "WindowOps" (yellow : blue : red : green) {
New Window : !"xterm &"
RefreshScreen : f.refresh
Redraw : (navy : magenta) : f.redraw
```

```
Move : f.move
}
```

此菜单标题为蓝底黄字，大多数的选择项为白底黑字 (缺省值)，只有Move选择项为紫红色底海蓝色字，而目前鼠标指针所在的选择项为绿底红字。

6. 控制uwm 的参数变量

到目前为止，你可以用指定鼠标和键的前后关系，来改变所指定的功能。有一个另一种类型的uwm 的定制可以改变许多内建功能操作的模式和样式，例如可以指定在 resize或move操作下，指示窗口新的位置的九字格，改变为只是一个外框而已。在联机帮助中列出所有的变量和它的意义，在此我们只介绍一些特别有用的和比较模糊的类型。

• 让缺省配置文件中的设定无效。

uwm 并没有结构抑制读取系统和用户配置文件。(-b 不会影响 $HOME/.uwmrc。)欲取消早先文件中的设定，可以含入 uwm的变量resetbinding、resetmenus和resetvariables，将会分别取消早先定义的对应、菜单和变量。(确保你将这些变量放在文件的顶端，否则它将取消在文件中所有在它之前的定义。)

• 限制窗口和图标在屏幕范围以内。

X允许你指定窗口位于屏幕的任何位置，甚至部分或全部在屏幕之外，这样有时会引起麻烦。当你建立一个窗口， uwm 并未提供任何帮助。但当你使用 f.newiconify对一个图标作解除图标化，如果变量normalw 被设定，则窗口会被完整地放在屏幕中，且尽量接近你用鼠标指针指定的位置。(如果你包含了normali 变量，同样可用于图标。)

• 控制push作用。

缺省f.pushxxx 功能将一个窗口往适当的方向推动一个像素的距离。可以指定 push=num 来推动num 个像素。也可以完全地改变操作的作法：不用通常的推动固定数目的像素的作法 (叫做pushabsolute)，可以指定pushrelative，这种情况窗口会被推动num 分之一大小的窗口。例如如果你指定：

```
push=5
pushrelative
```

则一个f.pushup将把窗口向上推动窗口本身高度五分之一的距离。

• 防止uwm 功能锁定应用程序。

一些缺省 uwm 的操作，像重新定义大小和移动会导致所有其他的客户程序被冻结，也就是说，防止它们输出到它们的窗口。可以用 nofreeze取消它。

如果你需要获得一些 uwm 所属窗口的打印，则这是必需的。它的副作用为当使用重新定义大小和移动时，外框格会大量地闪动，以致难以看到。

29.8 显示器管理器——xdm

现在我们已经讲述了有关 X的所有单个项目，例如，如何启动系统，如何设定一个窗口管理器的执行，如何执行应用程序，如何从不同的角度定制系统，如何退出系统。

本节中，我们把这些分开的部分放在一起，且描述一个完整的文件设定，用来定制涵盖所有的系统机器环境。本节将介绍最后一个 X工具：显示管理器 —— xdm，它提供一个启动X的精巧和清楚的方法。

29.8.1 需要做些什么

当我们启动X之后，我们需要安排屏幕，让一些在整个执行期间中都会使用的应用程序打

开，让一些偶然用到的应用程序以图标为开始时的表示方式。我们需要执行窗口管理器，对某些种类的功能做一些设定。详细来说，我们需要下列的程序：

- 一个xterm 的主控台，在屏幕左上角。
- uwm 在后台执行。
- 一个我们的（正常）编辑器的全屏幕xterm 窗口，以图标方式启动。
- 在右上角一个（较一般为小）的时钟。
- xbiff 在时钟之下。
- 一个计算器在右下角。
- 一个用到我们所有最小的字体的图标化的xterm ，它的高度比屏幕高。
- 在xbiff 之下，排列我们使用远程机器的频率图。

除了程序之外的项目：

- 设定背景窗口为亮灰色。
- 启动键盘输入功能。
- 从我们常用的网络主机访问到我们的服务器。
- 加载我们对所有客户程序用到的服务程序设定的资源，在 29.4节定义的 $HOME/.Xresource 文件中，根窗口RESOURCE_MANAGER属性之上。
- 启动一个屏幕保护程序。

并且需要uwm 有菜单以便：

- 容易地访问在网络上其他的主机。
- 更改一些键盘和鼠标的设定，且设定背景窗口的颜色。
- 启动那些我们偶而会用到的应用程序。
- 启动一些被选定的示例程序。

对这些我们自己的设定，在网络上其他的用户需要不同的初始设定，所以需要安排每一个用户依自己的喜好设定。理想上，用户应能自行设定而不需要系统管理员的帮助，下一节我们来看程序xdm 如何能帮助我们完成这些目的。

29.8.2 xdm

xdm 为X 显示管理器。xdm 管理一个或多个显示器，xdm 可在同一机器或远程的机器上执行。它可以做到所有xinit 能做到的功能，而且更多。它所隐含的概念为它应控制当你在 X工作时的会话，也就是从进入直到结束窗口系统的会话。 (用xinit ，有效会话为当执行xinit 开始，到注销最初的xterm窗口和关闭服务程序。)

xdm 较这更进一步，可以用它执行一个不确定的会话。当一个结束，下一个便准备开始。实际上，如果有需要，它可以不变地指定一个显示器。

xdm可完全替换xinit 。从现在起你可忘掉xinit，而且不再需要使用它，我们在最初使用xinit 的原因为它较易观察和了解系统的运作。

xdm 是一个非常灵活的程序，几乎可用它配置任何所需要的功能，在进一步深入之前，让我们观察一个样例会话的缺省行为，然后我们来看一看如何改进当一个用户进入 X系统所看到的初始界面。

1. 用xdm 的例子会话

我们将使用xdm 来设定X。机器已经启动，但尚未有窗口系统在其上执行。用下列的命令启动xdm：

```
xdm
```

xdm 开始执行，几乎立刻又看到外壳提示。然后屏幕背景更改为通常灰色形式，且看到一个大的X光标，所以知道服务程序已经启动。

接下来是一长段修止状态(大约持续15秒或更久)，而后，一个带着欢迎标题的窗口出现了，要求你输入登录名称和密码。输入你的用户名称和密码后，又过了一会儿，你可以看到一个xterm 窗口在左上角出现，以后的工作的方式和以前叙述的相同 —— 启动窗口管理器，执行应用程序等等。

需要结束时，也可用以前相同的方式结束： logout（注销）最初始的 xterm窗口。但这里xdm 和xinit 有不相同的地方，关掉服务程序的方法是，回到非 X的环境，屏幕回到最初始灰色的背景，过一会之后，又再度看到X的登录窗口。事实上，xdm 是执行一个循环的会话。

注意 就像许多的UNIX程序一样，最大的登录名称长度为8 个字符，如果超过这个长度，登录将会失败。（如果实际登录程序允许使用较长的名称，这种限制也许让人感到奇怪。）

2. 关闭xdm

有时可能需要完全地关闭X。为了做到这点，需要关闭xdm 。

在MIT 版中的服务程序，如果收到UNIX信号SIGTERM ，便会执行中止程序。xdm 利用到这点，如果送给它一个 SIGTERM ，它将中止所有它所控制的服务程序后离开。这就是中止系统的方法。

要实际中止xdm，可以在一个xterm 窗口（在你的机器上）用 ps来找出xdm 的process-id，而后用kill送给它SIGTERM。例如，在我们的机器上执行中止的动作如下：

```
venus% ps ax 1 grep xdm
1997 ? IW 0:00 xdm
1998 ? IW 0:00 xdm
2000 ? IW 0:00 xdm
2078 p0 S 0:00 grep xdm
venus% kill -TERM 1997
```

所有的应用程序将被强迫中止，服务程序也随之关闭。

注意 当相关于X的每一件事都结束后，屏幕可能只显示通常X背景的灰色形式，没有任何的shell 提示或任何事。此时，你的shell 已准备好接受你的命令，按下Return键将会看到。因为在以交互式下xdm 命令之后，外壳已将提示信号送出，所以不再重复,除非你按下Return键。

29.8.3 xdm的更多信息

我们在前所述的是xdm 的缺省模式下的操作，所以看起来并没有比 xinit 提供得更多。如果使用一个正常的工作站或显示器，一些外貌将不是很有趣。无论如何， X终端机是一个日渐增加的大众化设备，而xdm 可大量地简化管理类似的系统。X终端机通常没有它自己的文件系统，且不能支持一般目的的程序，必需在网络的某处执行包含窗口管理器和显示管理器的控制终端机软件，xdm 正是符合此需要的软件。

xdm 在下列这些场合比xinit 好：

- 它可控制数个服务程序，也就是说，其中有一些为远程的服务程序，也许是在 X终端机或相当小的工作站上。

- 它提供密码来存取系统，同样地，在 X终端机上非常有用（但在一个你已经登录的工作站会有一点困扰）。
- 它提供无限期的X的会话。你可以配置显示器总是以X方式操作，所以用户不需要担心如何启动系统。
- 它具有高度的可配置性。系统管理员可以设定根据机器特性启动和结束程序，掌握这些项目以供记帐、授权、文件系统等之用，且能让每一个个别的用户全范围性地修定他们所需的自己的环境。
- 从用户的观点，它提供一个干净而简单的方法来启动系统。

所以大体上，xdm 主要是一个系统管理工具，但它也提供让一个普通用户定制他希望的一致和一贯的系统结构。

xdm 的联机帮助包含了大量的有关如何使用系统的教学信息和伪指令，在此我们不再重复，我们将在以下说明如何正确地配置xdm 以提供在本节一开头所描述的环境。

xdm 非常灵活，且可以用许多不同的方式选择设定，我们将使用最简单的处理，并试着大致和联机帮助的描述保持一致。偶而我们在一些文件中使用不同的名称，用以强调此名称并非硬性的规定。

在我们工作的会话中，请注意我们事实上在扮演两个不同的角色：第一是系统管理员，为使用系统的人设定xdm ，第二是一般的用户，为我们自己的需求设定xdm 。

1. 系统管理员对xdm 的配置

缺省xdm 先查看文件：

/usr/lib/X11/xdm/xdm-config

如果它存在，会把它当成多设定几个其他参数的资源文件。我们将使用它，因为它可简化我们的工作。

联机帮助会列出所有能通过xdm-config文件设定的参数，但与我们紧密相关的参数有：

- 包含一个服务程序的目录的文件名称。
- 当任何错误发生时，xdm 用来记录的文件名称。
- 包含和启动系统有关的文件系统名称。
- 当服务程序启动后执行程序的名称，这个程序定义了你的“会话”——当这个程序中止时，xdm 视其为你的会话已结束，且回到它登录时的顺序，缺省时这个程序为 xterm ，就和使用xinit 一样，在退出你初始的xterm之前，会话将一直持续。

这是我们已在系统上定义的设定：

```
DisplayManager.severs:   /usr/lib/X11/xdm/our-server
DisplayManager.errorLogFile:   /usr/lib/X11/xdm/errors
DisplayManager*resource:   /usr/lib/X11/xdm/our-resource
DisplayManager*session:   /usr/lib/X11/xdm/our-session
```

我们已选择在目录/usr/lib/X11/xdm保持所有xdm 相关的文件，这只是代表名称，可以用任何你喜欢的目录。

所以可以看到我们使用 xdm-config，实际上是两个步骤，首先我们定义在 xdm-config中的一些文件名称，接着我们来设定方才命名的文件。现在我们来看一下我们在 xdm-config中定义的每一个资源。

(1) xdm 的服务程序的名单

这个被DisplayManager.servers设定的文件资源包含了一个 xdm能管理的服务程序的名单。每一行中包含了服务程序的名称（也就是显示器），服务程序的类型和类型有关的项目。

类型指出了显示器是本地的或远程的和是否为无限或单一的会话（详见 xdm 联机帮助）。我们将使用类型localTransient(单一会话在本地显示器上)。因为以此方式，如果发生任何错误，我们不致于陷入无穷循环中。稍后，当我们每件事都设定好且执行无误的话，我们会将类型改为本地而循环的会话。

对本地的显示器而言，和类型相关的信息是在此显示器上执行的服务程序的名称及其任何所需的参数。对远程的显示器，此信息可被忽略，但你仍需输入一个假的程序名称。

所以，在我们所建立的文件/usr/lib/xdm/our-服务程序包含这一行：

```
:0 localTransient   /usr/bin/X11/X  :0
```

如果我们喜欢执行循环会话，此文件便不再需要 —— 缺省设定会做到我们所需要的。所以我们在配置文件中不需定义DisplayManagers.servers resource。

(2) xdm 的错误日志文件

此文件从xdm 和xdm 的会话程序接收所有错误的信息, 且如果xdm 设定工作发生问题的话，这是第一个需要查看的地方。

当你开始设定系统，把此文件设定成任何人均可写入，否则，有问题的程序可能因没有写入权限而无法在文件中记录。

(3) 启动时的资源文件

此文件包含一个资源的名单，在 Authentication Widget 启动之前被xrdb加载。因此，能用它来为那些Widget设定资源。当然也能放置任何其他的资源规范，但通常会话程序的用户设定加载时会凌驾其上，所以通常不把其他的规范放在这里。

authentication Widget 资源的缺省设定在某些情况是很细的，但为了举例，我们只设定和bunner不同的标题，我们建立我们的文件 /usr/bin/X11/xdm/our-resource 包含一行：

```
xlogin.Login.greeting:X-Window on the Plants network
```

(4) xdm 的会话程序

可以指定任何程序为会话中所需的程序。可是当会话结束，通常选择一个程序让你能启动其他的程序。通过xdm 的缺省设定可以执行xterm ，但这种方式每当xterm 执行时你仍必需手动设置所有的设定。我们需要定义会话程序来执行所有的设定，且让会话程序保持活动的状态直到结束为止。但记住，我们希望用户如果需要能定义自己的会话程序，所以我们将使用两阶段的处理。如果是系统管理员，我们将设定一个一般性目的、基础的会话程序来调用一个用户自己的程序（如果它存在）。但其他方面将执行一个合理的缺省值。

基础的会话程序非常简单。如果用户设定有文件$HOME/.Xsession，则可以使用它，否则，将执行合理的缺省值 —— 启动uwm， 而后向一个xterm（xterm为我们指定在屏幕左上方的那一个）传递控制信息。但在这之前，我们先检查是否用户设定了文件 $HOME/.X资源，如果有的话，可以使用xrdb加载它。

2. 用户对于xdm 的配置

现在我们改变角色：我们不再是系统管理员，而是一个用户。我们可以依赖系统管理员已定义的缺省会话，但我们比较喜欢定义自己的会话，所以我们要获取那些说明我们所需的起始设定。

实例 .Xsession

我们已建立自己的$HOME/.Xsession，此程序的操作十分简单。我们假设你的会话程序是一个外壳。虽然不一定如此，但通常都是这样，除非你要写一个 xetrm 的复杂的代替品。

• 文件中的命令依序排列，所以最后一行所执行的是一个程序。它可以在整个会话中持续。

因为，当此程序结束，则会话程序结束，且每一件事也均将结束。

- 除了后台的最后一个命令，所有的命令均被执行。也就是说，在命令行最后加一个 &符号。在实例程序中，如果在uwm那一行省略“ & ”号，uwm 也会启动。但在uwm 结束时，它的下一行将不会继续执行，这样是不对的！
- 最后的命令必为exec命令，所以它继续执行且保持会话继续活动。如果你像其它命令一样在后台中执行它，它会好好地执行，但此会话程序执行至文件结束将会中断，而结束会话。如果你不用 exec，且省略“ & ”号，则它会执行，且此会话将地持续工作。你只是较你所需要的多执行了一个处理，就如同你仍有最后的程序和会话程序本身。
- 对所有的程序建立窗口时设定几何位置规范 ，否则，当它们启动后，你将以“ 手动 ”方式指定它们的位置。
- 在文件中最后一行的程序通常用来启动 xterm ，因为它定义了会话的生命期，在执行X时此窗口总是存在，所以通常设定两个特别的选项：

1) 使用-C选项使得xterm 为一个主控制台，所以系统信息会在此窗口显示。

2) 设定-ls 选项使它的外壳为登录外壳。这样使得外壳读入 .login或.profile文件，所以你的环境变量会适当地设定。

3) 此会话程序文件必需有执行许可。使用上述站点范围的会话程序，这对用户会话脚本不是绝对需要，它实际是对站点范围的程序本身。如果那不能执行，你只能获得 xdm 的缺省设定。

在设置 .Xsession 和依赖它启动窗口会话之前，最好能从一个 .xterm 窗口启动 .Xsession 以便严格测试它。

29.8.4 uwm 配置

我们需要设定四个uwm 菜单：一是连接到其他的主机，二是执行一些 X的应用程序，三是设定一些键盘和鼠标参数（有点儿像缺省的 Preference菜单），四是执行演示程序。

对主机菜单，我们现在希望只要借助于从菜单中选取主机名称便可在任何主机上启动 xterm 。我们常常需在mars上做一些系统管理，所以我们将设定选择为超级用户，我们将在左下角建立一个超级用户窗口。但对一般的 xterm，省略几何位置规范，所以当它建立时，可以明确地定位它。我们将以Meta-Shift-Left 对应此菜单。所以在$HOME/.uwmrc中包含了此行。

uwmrc 剩余的部分可以用来设定定制的对应和一般窗口配置操作的参数。注意下列几点：

- 选择一个较缺省值稍大的字体 (用menufont=fixed)，降低菜单选项中的空白空间 (用vmenupad=1)，所以菜单不会很大(menufont 可能未在联机帮助中描述)。
- 我们设定为可重设所有的菜单、对应和变量。这就清除了 uwm 的配置。
- 如果可能，我们较愿意使用鼠标的UP事件函数而非DOWN事件。这种方式能在释放按钮之前按下其他的按钮，来改变操作或中止操作。
- 我们已包含一些uwm 菜单的功能 —— 一个是删除应用程序窗口，另一个是重新启动uwm 。它们不是必要的，但当你对系统很有经验时会很有用。

附录A　GCC使用介绍

A.1　获得GCC的方法

GCC是一种C++语言的编译器，它可以免费获得，其功能十分强大。你可以在 URL:ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/ 的网站上找到正式的Linux GCC发布系统，而且是已经编译好的可执行文件。

自由软件基金会（Free Software Foundation）所发布的GCC最新源代码可以从网站 GNU archives3上取得。没有必要非得使用上述的版本，不过这个版本的确是目前最新的。 Linux GCC的维护网友让你可以很轻松的自行编译这个最新的版本。 configure命令脚本会帮你自动设置好所有该做的事情。

A.2　C程序库与头文件

该选哪一套程序库取决于你的系统是ELF格式的还是a.out格式的，以及你希望系统变成哪一种格式。

• libc-5.2.18.bin.tar.gz

ELF共享程序库，静态程序库与头文件。

• libc-5.2.18.tar.gz

libc-5.2.18.bin.tar.gz的源代码。你也需要这个文件，因为.bin.套件中含有必需的头文件。

• libc-4.7.5.bin.tar.gz

这个文件是a.out的共享程序库与静态程序库，是为了与前述的libc5套件兼容而设计的。除非你想要继续使用a.out的程序或者继续开发a.out的程序，否则，是不需要它的。

A.3　一些有用的工具 (as、ld、 ar、 strings 等)

你也可以从网站tsx-116上找到这些工具程序。目前的版本是binutils-2.6.0.2.bin.tar.gz。

需要注意的是binutils只适用于ELF格式，而且目前libc的版本也都是属于ELF格式的；当然，习惯a.out格式的人如果有个ELF格式的libc与a.out格式的libc联合起来一起使用，是再好不过的事了。不可否认， C程序库的发展正以坚定的脚步迈向 ELF格式，除非你真的有很好的理由，否则应该放弃a.out格式。

A.4　GCC的安装与GCC的设置

A.4.1　GCC的版本

在外壳提示符号下键入gcc -v，屏幕上就会显示出你目前正在使用的 GCC的版本。同时也可确定你现在所用的是ELF格式或是a.out格式。在我的系统上，执行gcc -v的结果是：

```
$ gcc -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
```

上面的信息指出了几件重要的事情：

1) i486 这是指明你现在正在用的gcc是为486微处理器写的(你的电脑可能是386或是586)。这3种微处理器的芯片所编译而成的程序代码，彼此间是可以兼容使用的。差别之处是 486的程序代码在某些地方可加上 padding的功能，所以在486上面可以运行得比较快。对 386的计算机而言，执行程序的效能并不会有什么不良的影响，只不过程序代码变得稍稍的大了一些。

2) box 可以说没有什么用处。

3) linux 其实这是指linuxelf或是linuxaout。这一项会令人引起不必要的困惑，究竟是指哪一种会根据你所用的版本而异。

linux 若版本序号是2.7.0（或者更新）就是指ELF格式；否则，就是指a.out格式。

linuxaout 意指a.out格式。当linux的定义从a.out更换到ELF时，linuxaout就成为一个目标。因此，你不会看到任何新于2.7.0.的版本gcc有linuxaout。

linuxelf 已经过时了。通常这是指2.6.3版的gcc，而且这个版本也可以用来产生 ELF的可执行文件。要注意的是， gcc2.6.3版在产生ELF程序代码时会有错误，所以如果你目前用的恰好是这个版本，建议你赶快升级。

4) 版本的序号。

所以，总结起来，我的系统用的是2.7.2版的gcc，可以产生ELF格式的程序代码。

A.4.2 Gcc的安装目录

如果在安装gcc时没有仔细看着屏幕，或者你是从一个完整的发行系统里把 gcc单独提出来安装的话，那么，也许你会想知道到底这些东西装好后是在整个文件系统的哪些地方。几个重点目录如下：

1) /usr/lib/gcc-lib/target/version/（与子目录）

大部分的编译器就是在这个地方。在这里有可执行的程序，实际在做编译的工作；另外，还有一些特定版本的程序库与头文件等也会保存在此。

2) /usr/bin/gcc

指的是编译器的驱动程序，也就是你实际在命令行上执行的程序。这个目录可供各种版本的gcc使用，只要你用不同的编译器目录（如上所述）来安装就可以了。要知道内定的版本是哪一个，在外壳提示符号下键入 gcc -v。如果想强迫执行某个版本，就键入 gcc -V version。例如：

```
# gcc -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
# gcc -V 2.6.3 -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.6.3/specs
gcc driver version 2.7.2 executing gcc version 2.6.3
```

3) /usr/target/(bin|lib|include)/

如果你装了几种目标对象，例如a.out与elf，或者某种交叉编译器，那些属于非主流目标对象的程序库 —— binutils（as、ld等等），工具与头文件等都可以在这里找到。即使你只安装了一种gcc，还是可以在这里找到这些原本就是替它们准备的东西。如果不在这个目录中，那么就应该是在/usr/(bin|lib|include)中了。

4) /lib/、/usr/lib

与其他的相似的目录都是主流系统的程序库目录。许多的应用程序都会用到/lib/cpp，因此，

你也需要它，或者从/usr/lib/gcc-lib/target/version/ 目录里拷贝，或者使用符号链接指向那里。

A.4.3 头文件

假如把你自行安装在/usr/local/include目录底下的头文件排除在外的话， Linux还有下面目录下的另外3种主要的头文件：

/usr/include/

与其子目录底下的头文件，大部分都是由 H.J.Lu开发的libc套件所提供的。说“ 大部分 ”的原因是因为你可能有其他来源的头文件放在这里；尤其是，如果你现在用的是最新的 libc发行系统的话，那东西之多是人人为之咋舌的！

在内核源代码的发行系统内，应该有 /usr/include/linux与 /usr/include/asm （里头有这些文件：<linux/*.h> 与 <asm/*.h>）的符号链接。把源代码解压缩后，可能你也会发现，需要在内核的目录底下做make config的动作。很多的文件都会依赖<linux/autoconf.h>的帮忙，可是这个文件却有可能因版本不同而不存在。若干内核版本里， asm只是它自己的一个符号链接，仅仅在make config时才建立出来而已。

所以，当你在目录/usr/src/linux底下，解开内核的程序代码时，可以参照下面的命令：

```
$ cd /usr/src/linux
$ su
# make config
```

回答接下来的问题正不正确并不重要，除非你打算继续构造内核。

```
# cd /usr/include
# ln -s ../src/linux/include/linux .
# ln -s ../src/linux/include/asm .
```

诸如<float.h>、<limits.h>、<varargs.h>、<stdarg.h> 与<stddef.h>之类的文件，会随着不同的编译器版本而不同，属于你个人的文件，可以在 /usr/lib/gcc-lib/i486-box- linux/2.7.2/include/与其他相类似（相同）的目录名称的地方找到。

A.5 建立交叉编译器

假设你已经拿到gcc的源代码，通常你只要依循INSTALL文件的指示便可完成一切的设置。make后面再接configure --target=i486-linux --host=XXX on platform XXX，就能帮你完成操作。要注意的是，你会需要Linux内核的头文件；同时也需要建立交叉汇编器与交叉链接器，你可以在 URL:ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ 中找到它们。

A.6 移植程序与编译程序

A.6.1 gcc自行定义的符号

在执行gcc时，附加-v这个参数，就能找出你所用的这版 gcc自动帮你定义了什么符号。例如，我的计算机输出结果是：

```
$ echo 'main(){printf("hello world\n");}'| gcc -E -v -
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
 /usr/lib/gcc-lib/i486-box-linux/2.7.2/cpp -lang-c -v -undef
-D__GNUC__=2 -D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux
-D__ELF__ -D__unix__ -D__i386__ -D__linux__ -D__unix -D__i386
```

```
-D__linux -Asystem(unix) -Asystem(posix) -Acpu(i386)
-Amachine(i386) -D__i486__ -
```

假若你正在编写的程序代码会用到一些 Linux独有的特性，那么把那些无法移植的程序代码，以条件式编译的前置命令封括起来。如下所示：

```
#ifdef __linux__
/* ... funky stuff ... */
#endif /* linux */
```

用__linux__就可以完成任务；看仔细一点，不是 linux。尽管linux也有定义，毕竟，这个不是POSIX的标准。

A.6.2 调用编译程序

在命令行上执行gcc时，只要在它的后面加上 -On的选项，就能让gcc编译出最优化的计算机编码。这里的n是一个可以省略的小整数，不同版本的 gcc，n的意义与其作用都不一样，不过，典型的范围是从0变化到2，再升级到3。

gcc在其内部会将这些数字转换成一系列的 -f与-m的选项。执行gcc时带上标志-v与-Q，你就能很清楚的看出每一种等级的-O对应到哪些选项。好比说，就-O2来讲，我的gcc会显示：

```
enabled: -fdefer-pop -fcse-follow-jumps -fcse-skip-blocks
-fexpensive-optimizations
    -fthread-jumps -fpeephole -fforce-mem -ffunction-cse -finline
    -fcaller-saves -fpcc-struct-return -frerun-cse-after-loop
    -fcommon -fgnu-linker -m80387 -mhard-float -mno-soft-float
    -mno-386 -m486 -mieee-fp -mfp-ret-in-387
```

要是你用的最优化编码等级高于你的编译器所能支持的等级（例如 -O6），那么它的效果就跟你用你的编译器所能提供的最高等级的效果是一样的。

A.6.3 和特定的微处理器相关

有一些-m的标志十分有用，但是却无法根据各种等级的 -O来使用。这之中最重要的是 -m386和-m486这两种。它们用来告诉gcc该把正在编译的程序代码视作专为 386或是486计算机所写的代码。不论是用哪一种-m来编译程序代码，都可以在彼此的计算机上执行。但 -m486编译出来的代码会比较大，不过拿来在386的计算机上运行也不会比较慢就是了。

目前尚无-mpentium或是-m586的标志。Linux建议我们可以用 -m486 -malign-loops=2-malign-jumps=2 -malign-functions=2来得到最佳的486程序代码。

A.6.4 Internal compiler error: cc1 got fatal signal 11

Signal 11是指 SIGSEGV，或者“ segmentation violation ”。通常这是说gcc对自己所用的指标感到困惑，而且还试图把信息写入不属于它的内存里。所以，这可能是一个 gcc的错误。然而，大体而言， gcc是一个经过严密测试且可靠度良好的软件。它也使用了大量复杂的信息结构与惊人数量的指标。假如你无法再一次重复这个错误 (当你重新开始编译时，错误的信息并没有一直出现在同一个地方)那几乎可以确定是你的硬件(CPU、内存、主机板或是高速缓存)本身有问题。千万不要因为你的电脑可以通过开机程序的测试、或者 Windows可以运行得好、或者其他什么程序可以运行得好，就回过头来说这是 gcc的一个错误；你所做的这些开机测试动作，通常没有什么实际上的价值。

A.6.5 移植能力

1. BSD的用户 (有 bsd_ioctl、守护进程和 <sgtty.h>)

你可以使用 -I/usr/include/bsd来编译程序，同时使用 -lbsd来链接程序。例如：在你的Makefile文件内，把-I/usr/include/bsd加到CFLAGS那一行；把-lbsd加到LDFLAGS那一行。如果你确实需要BSD类型的信号，也不需要再加上-D__USE_BSD_SIGNAL了。那是因为当你用了-I/usr/include/bsd，并且包括了头文件<sig?nal.h>之后，建立时就会自动加入这些信号。

2. 丢失的信号(SIGBUS、SIGEMT、SIGIOT、SIGTRAP、SIGSYS 等)

Linux与POSIX是完全相容的。不过，有些信号并不是POSIX中定义的。

在POSIX.1中省略了SIGBUS、SIGEMT、SIGIOT、SIGTRAP与SIGSYS信号，那是因为它们的行为与实际运行的方式息息相关，而且也无法进行适当的分类。确认实际运行方式后，便可以发送这些信号，可是必须以文件形式说明它们是在什么样的环境下发送出来的，以及指出任何与它们的发展相关的限制。

想要修正这个问题，最简单也是最笨的方法就是用 SIGUNUSED重新定义这些信号。正确的方法应该是以条件式的编译#ifdef的形式来处理这些问题：

```
#ifdef SIGSYS
/* ... non-posix SIGSYS code here .... */
#endif
```

3. K & R代码

gcc是一个与ANSI相容的编译器；奇怪的是，目前大多数的程序代码都不符合 ANSI所定的标准。如果你喜欢用ANSI提供的标准来编写C程序，似乎除了加上-traditional的标志之外，就没有其他什么可以多谈的了

要注意的是，尽管你用了-traditional来改变语言的特性，它的效果也仅局限于gcc所能够接受的范围。例如，-traditional会打开-fwritable-strings，使得字符串常数移至数据内存空间内(从程序代码内存空间移出来，这个地方是不能任意写入的)。这样做会让程序代码的内存空间无形中增加了。

4. 预处理符号和函数原型的冲突

最常见的问题是Linux中有许多常用的函数都定义成宏存放在头文件内。此时若有相似的函数原型出现在程序代码内，预处理程序会拒绝进行语法分析的预处理。常见的函数有 atoi()与atol()。

5. sprintf()函数

在大部分的Unix系统上， sprintf(string, fmt,...)传回的是string的指针，然而， Linux（遵循ANSI）传回的却是放入string内的字符数目。

6. select()函数 —— 程序执行时会处于忙碌/等待的状态

很久以前， select()的计时参数还只是只读的。即使到了最近，操作联机帮助中仍然有下面这段的警告：

select()应该是根据修正时间的数值（如果有的话），再传回自原始计时开始后所剩余的时间。未来的版本可能会使这项功能实现。因此，就目前而言，若以为调用 select()之后，计时指针仍然不会被修正过，是不正确的。

函数select()传回的是扣除等待尚未到达的信息所耗费的时间后，其剩余的时间数值。如果在计时结束时，都没有信息传送进来，计时参数便会设为 0；如果接着还有select()函数，以同样的计时数据结构来调用，那么select()便会立刻结束。

若要修正这项问题，只要每次调用 select()前，都把计时数值放到计时数据结构内，这样就没有问题了。把下面的程序代码，

```
struct timeval timeout;
timeout.tv_sec = 1; timeout.tv_usec = 0;
while (some_condition)
 select(n,readfds,writefds,exceptfds,&timeout);
```

改成，

```
struct timeval timeout;
while (some_condition) {
      timeout.tv_sec = 1; timeout.tv_usec = 0;
      select(n,readfds,writefds,exceptfds,&timeout);
  }
```

这个问题，在有些版本的Mosaic里是相当著名的，只要有一次的等待， Mosaic就停止在那里了。Mosaic的屏幕右上角，有个圆圆的、会旋转的地球动画。那个球转得愈快，就表示信息从网络上传送过来的速率愈慢！

7. 产生中断的系统调用

当一个程序以Ctrl-Z中止、然后再重新执行时，或者有其他可以产生Ctrl-C中断信号的情况，如子程序的终结等，系统就会显示“ interrupted system call ”或者“ write: unknown error ”，或者诸如此类的信息。

比起一些旧版的Unix ， POSIX的系统检查信号的次数是多一点。而Linux可能会执行signal处理程序。

8. 系统调用的返回值

在下列系统调用的执行期间：

select()、pause()、connect()、accept()、read()（终端上）、套接字、管道或文件（/proc中）、write()（终端上）、套接字、管道或行式打印机、 open()（FIFO上）、PTY或串行线路、 ioctl()（终端上）、fcntl()（具有命令F_SETLKW）、wait4()、syslog()、任何TCP或NFS操作。

就其他的操作系统而言，你需要的可能就是下面这些系统调用了： creat()、close()、getmsg()、putmsg()、msgrcv()、msgsnd()、recv()、send()、wait()、waitpid()、wait3()、tcdrain()、sigpause()、semop() 。

在系统调用期间，若有一信号产生，处理程序就会被调用。当处理程序将控制权转移回系统调用时，它会侦测出已经产生中断，而且返回值会立刻设置成 -1，而errno设置成EINTR。程序并没有想到会发生这种事，所以就停止了。

有两种修正的方法可以选择：

1) 对每个你自行安装的信号处理程序，都须在 sigaction的标志加上SA_RESTART。例如，把下列的程序，

```
  signal (sig_nr, my_signal_handler);
```

改成：

```
  signal (sig_nr, my_signal_handler);
  { struct sigaction sa;
      sigaction (sig_nr, (struct sigaction *)0, &sa);
#ifdef SA_RESTART
      sa.sa_flags |= SA_RESTART;
#endif
#ifdef SA_INTERRUPT
```

```
        sa.sa_flags &= ~ SA_INTERRUPT;
#endif
        sigaction (sig_nr, &sa, (struct sigaction *)0);
    }
```

要注意的是，当这部分的更改大量应用到系统调用之后，调用 read()、write()、ioctl()、select()、 pause() 与 connect()时，你仍然得自行检查EINTR。如下所示：

2) 你自己明确地检查EINTR：

这里有两个针对read()与ioctl()的例子。

原始的程序使用read()：

```
int result;
while (len > 0) {
  result = read(fd,buffer,len);
  if (result < 0) break;
  buffer += result; len -= result;
}
```

修改成，

```
int result;
while (len > 0) {
  result = read(fd,buffer,len);
  if (result < 0) { if (errno != EINTR) break; }
  else { buffer += result; len -= result; }
}
```

原始的程序使用ioctl()：

```
int result;
result = ioctl(fd,cmd,addr);
```

修改成，

```
int result;
do { result = ioctl(fd,cmd,addr); }
while ((result == -1) && (errno == EINTR));
```

注意一点，有些版本的BSD Unix，其内定的行为是重新执行系统调用。若要让系统调用中断，得使用 SV_INTER?RUPT或SA_INTERRUPT标志。

9. 可以写入的字符串

gcc认为当用户打算让某个字符串当作常数来使用时，它就只是字符串常数而已。因此，这种字符串常数会保存在程序代码的内存区段内。这块区域可以交换到磁盘的 image文件上，避免耗掉交换区占用的内存空间，而且任何尝试写入的举动都会造成分页的错误。

对旧一点的程序而言，这可能会产生一个问题。例如，调用 mktemp()，传递的参数是字符串常数。mktemp()会试图在“适当的位置”重新写入它的参数。

修正的方法不外乎如下：

- 以-fwritable-strings编译，迫使gcc将此常数置放在数据内存空间内。
- 将侵犯地址的部分重新改写，配置一个不为常数的字符串，在调用前，先以 strcpy()将拷贝进去。

10. 为什么调用execl()会失败？

那是因为调用的方式不对。 execl的第一个参数是想要执行的程序名。第二个与后续的参数会变成所调用的程序的argv数组。记住：传统上， argv[0]是只有当程序没有带着参数执行时才会有的设置值。所以，你应该这样写：

```
execl("/bin/ls","ls",NULL);
```

而不是只有，

```
execl("/bin/ls", NULL);
```

执行程序而不带任何参数，可解释成是一种邀请，目的是把此程序的动态程序库独立的特性显示出来。至少，a.out是这样的。就ELF而言，事情就不是这样了。

A.7 除错与监管

A.7.1 预防

lint对Linux而言并没有很广泛的用途，主要是因为大部分的人都能满足于 gcc所提供的警告信息。可能最有用的就是 -Wall参数了。这个参数的用途是要求 gcc将所有的警告信息显现出来。

A.7.2 除错

怎样做才能将除错信息放到一个程序里？你需要添加 -g的参数来编译与链接程序，而且不可以用-fomit-frame-pointer参数。事实上，你不需要重新编译所有的程序，只需重新编译目前你正在除错的部分即可。

就a.out格式而言，共享程序库是以-fomit-frame-pointer编译而成，这个时候，gdb就变得没有用处了。链接时给定 -g的选项，应该就隐含着静态链接的意义了；这就是为什么要加 -g的原因了。

如果链接器链接失败，告诉你找不到 libg.a，那就是在/usr/lib/的目录底下少了 libg.a文件。libg.a是一个C语言使用的一个特殊的侦错程序库。一般在 libc的套件内就会提供libg.a；不然的话（新版是这样的），你可能需要拿libc的源代码自己设置了。不过，实际上你应该不需要才对。不管是什么目的，大部分的情况下，只需将 libg.a链接到/usr/lib/libc.a，你就能得到足够的信息了。

很多的GNU软件在编译链接时，都会设置 -g的选项；这样做会造成执行文件过大的问题（通常是静态的链接）。实际上，这并不是一个很好的做法。

如果程序本身有 autoconf，产生了 configure命令脚本，通常你就可以用 ./configure CFLAGS=或是./configure CFLAGS=-O2来关掉除错信息。否则，你得检查 Makefile了。当然，假如你用的是ELF格式，程序便会以动态的方式来链接，而不管是否有 -g的设置。因此，你可以把-g参数去掉。

1. 实用的软件

据了解，大部分人都是用gdb来除错的。可以从GNU archive sites7得到原始程序，或者到tsx-118得到可执行文件。xxgdb是一个X界面的除错程序，它是基于gdb的（也就是说你得先安装好 gdb程序，才能再安装 xxgdb程序）。xxgdb的源程序代码可以在 URL:ftp://ftp.x.org/con?trib/xxgdb-1.08.tar.gz 中找到。

另外，UPS除错程序已由 Rick Sladkey移植成功。UPS可以在X窗口环境下运行，不像xxgdb那样仅是gdb的X前端界面。这个除错程序有一大堆的优点，而且如果你得花时间去除一个错误很多的程序，建议你考虑使用 xxgdb。事先编译好的 Linux版与修正版的源代码可以在 URL:ftp://sun?site.unc.edu/pub/Linux/devel/debuggers/ 找到。而最初的原始程序则放在 URL:ftp://ftp.x.org/contrib/ups-2.45.2.tar.Z 中 。

另一个用来除错的工具 strace也是相当的有用。它可以显示出由程序所产生的系统调用，而且还拥有其他众多繁复的功能。如果你手边没有源代码的话， strace可以帮你找出有那些已编译进执行文件内路经名称，另外， strace可指导学习程序是怎么在电脑中执行的。最新的版本（目前是3.0.8）可在URL:ftp://ftp.std.com/pub/jrs/ 找到。

2. 守护程序

早期典型的守护程序是执行fork()，然后终止父程序。这样的做法使得除错的时间减短了。了解这点的最简单的方法就是替fork()设一个中断点。当程序停止时，强迫fork()传回0。

```
(gdb) list
1    #include <stdio.h>
2
3    main()
4    {
5      if(fork()==0) printf("child\n");
6      else printf("parent\n");
7    }
(gdb) break fork
Breakpoint 1 at 0x80003b8
(gdb) run
Starting program: /home/dan/src/hello/./fork
Breakpoint 1 at 0x400177c4
Breakpoint 1, 0x400177c4 in fork ()
(gdb) return 0
Make selected stack frame return now? (y or n) y
#0  0x80004a8 in main ()
      at fork.c:5
5      if(fork()==0) printf("child\n");
(gdb) next
Single stepping until exit from function fork,
which has no line number information.
child
7    }
```

3. 内核文件

当Linux开机时，通常状态会设置成不要产生内核文件。要是你那么喜欢内核文件的话，可以用外壳的builtin命令使其重新生效：就C-shell相容的外壳程序（如 tcsh）而言，会是下面这样：

```
% limit core unlimited
```

而类似Bourne shell的外壳（sh, bash, zsh, pdksh）则使用下面的语法：

```
$ ulimit -c unlimited
```

如果你想要有个多样化的内核文件名，那么你可以对你的内核程序做一点小小的更改。找一找fs/binfmt_aout.c与fs/binfmt_elf.c文件中与下列相符的程序段：

```
      memcpy(corefile,"core.",5);
#if 0
      memcpy(corefile+5,current->comm,sizeof(current->comm));
#else
      corefile[4] ='\0 ' ;
#endif
```

将0换成1。

A.7.3 监管

监管是用来检核一个程序中哪些部分是最常调用，或者执行的时间最久的方法。这对程序的最佳化与找出时间是如何浪费，是相当好的方式。你必须就你所要的时间信息的目的文件加上-p来编译，而且如果要让输出的文件有意义，你也会需要 gprof（来自binutils套件的命令）。参阅gprof的联机帮助，可得知其细节。

A.8 链接

我们称执行期间所发生的事为动态加载，这一主题会在下一节中谈到。你也会在别的地方看到我把动态加载描述成动态链接。换句话说，这一节所谈的，全部是指发生在编译结束后的链接。

A.8.1 共享程序库和静态程序库

建立程序的最后一个步骤便是链接，也就是将所有分散的小程序组合起来，看看是否遗漏了些什么。显然，有一些事情是很多程序都会想做的，例如打开文件。接着所有与打开文件有关的小程序就会将保存程序库的相关文件提供给你的程序使用。在一般的 Linux系统上，这些小程序可以在/lib与/usr/lib/目录底下找到。

当你用的是静态程序库时，链接器会找出程序所需的模块，然后实际将它们拷贝到执行文件内。然而，对共享程序库而言，就不是这样了。共享程序库会在执行文件内留下一个记号，指明当程序执行时，首先必须加载这个程序库。显然，共享程序库是试图使执行文件变得更小，等同于使用更少的内存与磁盘空间。 Linux内定的行为是链接共享程序库，只要Linux能找到这些共享程序库的话，就没什么问题。不然 ,Linux就会链接静态程序库了。如果你想要共享程序库的话，检查这些程序库（*.sa对于 a.out格式, *.so 对于ELF格式）是否在它们该在的地方，而且是可以读取的。

在Linux上，静态程序库会有类似 libname.a这样的名称；而共享程序库则称为libname.so.x.y.z，此处的x.y.z是指版本序号。共享程序库通常都会有链接符号指向静态程序库（很重要的）与相关联的.sa文件。标准的程序库会包含共享与静态程序库两种格式。

你可以用ldd（List Dynamic Dependencies）来查出某支程序需要哪些共享程序库。

```
$ ldd /usr/bin/lynx
    libncurses.so.1 => /usr/lib/libncurses.so.1.9.6
    libc.so.5 => /lib/libc.so.5.2.18
```

这是说在我的系统上， WWW浏览器lynx会依赖libc.so.5 (the C library)与libncurses.so.1（终端机屏幕的控制）的存在。若某个程序缺乏独立性， ldd就会说“ statically linked ”或者“ statically linked (ELF) ”。

A.8.2 sin() 在哪个程序库里

nm 程序库名称

此命令应该会列出程序库名称所指向的所有符号。这个指令可以应用在静态与共享程序库上。假设你想知道tcgetattr()是在哪儿定义的：你可以这样做，

```
$ nm libncurses.so.1 | grep tcget
U tcgetattr
```

U指出未定义 —— 也就是说ncurses程序库用到了tegetattr()，但是并没有定义它。你也可以

这样做：

```
$ nm libc.so.5 | grep tcget
00010fe8 T __tcgetattr
00010fe8 W tcgetattr
00068718 T tcgetpgrp
```

W说明了弱态，意指符号虽已定义，但可由不同程序库中的另一定义所替代。最简单的正常定义（像是tcgetpgrp）是由T所标示的。

标题所谈的问题，最简明的答案便是 libm.(so|a)了。所有定义在 <math.h>的函数都保留在maths程序库内。因此，当你用到其中任何一个函数时，都需要以 -lm的参数链接此程序库。

A.8.3 X文件

```
ld: Output file requires shared library `libfoo.so.1`
```

ld与其相类似的命令在搜索文件的方法上，会依据版本的差异而有所不同，但是你可以合理假设的唯一内定目录便是/usr/lib了。如果你希望它所在的程序库也列入搜索的行列中，那么你就必须以-L选项告知gcc或是ld。

要是你发现一点效果也没有，就赶紧察看此文件是不是还在原来的位置。就 a.out而言，以-lfoo参数来链接，会使得ld去寻找libfoo.sa；如果没有成功，就会换成寻找 libfoo.a。就ELF而言，ld会先找libfoo.so，然后是libfoo.a。libfoo.so通常是一个链接符号，链接至libfoo.so.x。

A.8.4 建立自己的程序库

1. 控制版本

与其他任何程序一样，程序库也有修正不完的错误。它们也可能产生出一些新的特点，更改目前存在的模块的功效，或者将旧的移除掉。这对正在使用它们的程序而言，可能会是一个大问题。如果有一个程序是根据那些旧的特点来执行的话，那怎么办？

所以，我们引进了程序库版本编号的观念。我们将程序库次要与主要的更改分类，同时规定次要的更改是不允许用到这程序库的旧程序发生中断的现象。你可以从程序库的文件名分辨出它的版本。

libfoo.so.1.2的主要版本是1，次要版本是2。次要版本的编号可能真有其事，也可能什么都没有。libc在这一点上用了修正程度的观念，而订出了像 libc.so.5.2.18这样的程序库名称。次要版本的编号内若是放一些字母、底线、或者任何可以显示的 ASCII字符，也是很合理的。

ELF与a.out格式最主要的差别之一就是在设置共享程序库这件事上；我们先看 ELF，因为它比较简单一些。

2. ELF

ELF（Executable and Linking Format）最初是由USL（UNIX System Laboratories）开发的二进位格式，目前正应用于Solaris与System V Release4上。由于ELF所具有的灵活性远远超过Linux过去所用的a.out格式，因此GCC与C程序库的发展人士于1995年决定改用ELF为Linux标准的二进位格式。

ELF是由SVR4所引进的新式改良目标文件格式。 ELF比COFF多出了不少功能。 ELF可由用户自行延伸。ELF视一个文件为节区，如串行般的组合；而且此串行可为任意的长度（而不是一固定大小的阵列）。这些节区与COFF的不一样，并不需要固定在某个地方，也不需要以某种顺序排列。如果用户希望捕捉到新的数据，便可以加入新的节区到目标文件内。 ELF也有一个更强有力的除错方式，称为 DWARF（Debugging With Attribute Record Format），但目前

Linux并不完全支持。DWARF DIE（Debugging Information Entries）的链接串行会在ELF内形成.debug的节区。DWARF DIE的每一个.debug节区并非一些少量且固定大小的信息记录的集合，而是一任意长度的串行组合，拥有复杂的属性，而且程序的数据会以有范围限制的树状数据结构写出来。DIE所能捕捉到的大量信息是COFF的.debug节区无法比拟的。

ELF文件是从SVR4（Solaris 2.0 ）ELF存取程序库（ELF access library）内存取的。此程序库可为ELF提供一简便快速的界面。使用ELF存取程序库最主要的好处之一便是，你不再需要去察看一个ELF文件的qua了。UNIX文件以Elf的型式来存取；调用elf_open()之后，从此时开始，你只需调用elf_foobar()来处理文件的某一部分即可，并不需要把文件实际在磁盘上的镜像搞得一团乱。

3. ELF共享程序库

如果想让libfoo.so成为共享程序库，基本的步骤如下：

```
$ gcc -fPIC -c *.c
$ gcc -shared -Wl,-soname,libfoo.so.1 -o libfoo.so.1.0 *.o
$ ln -s libfoo.so.1.0 libfoo.so.1
$ ln -s libfoo.so.1 libfoo.so
$ LD_LIBRARY_PATH=`pwd`:$LD_LIBRARY_PATH ; export LD_LIBRARY_PATH
```

这会产生一个名为libfoo.so.1.0的共享程序库，以及给予ld适当的链接（libfoo.so）还有使得动态加载程序（dynamic loader）能找到它（libfoo.so.1）。为了进行测试，我们将目前的目录加到LD_LIBRARY_PATH里。

当你的程序库制做成功时，别忘了把它移到/usr/local/lib的目录底下，并且重新设置正确的链接路径。libfoo.so.1与libfoo.so.1.0的链接会由ldconfig不断地更新。就大部分的系统来说，ldconfig会在开机过程中执行。libfoo.so的链接必须由手动方式更新。如果你对程序库所有组成（如头文件等）的升级，总是抱持着认真的态度，那么最简单的方法就是让 libfoo.so 符号链接到 libfoo.so.1；这样一来，ldconfig便会替你同时保留最新的链接。要是你没有这么做，你自行设置的东西就会在数日后有千奇百怪的问题出现。

```
$ su
# cp libfoo.so.1.0 /usr/local/lib
# /sbin/ldconfig
# ( cd /usr/local/lib ; ln -s libfoo.so.1 libfoo.so )
```

4. 版本编号、soname与符号链接

每一个程序库都有一个soname。当链接器发现它正在搜索的程序库中有这样的一个名称，链接器便会将soname箝入链接中的二进位文件内，而不是它正在运行的实际的文件名。在程序执行期间，动态加载程序会搜索拥有soname这样的文件名的文件，而不是程序库的文件名。因此，一个名为libfoo.so的程序库，就可以有一个libbar.so的soname了。而且所有链接到libbar.so的程序，当程序开始执行时，会寻找的便是libbar.so了。

这听起来好像一点意义也没有，但是这一点，对于了解多个不同版本的同一个程序库是如何在单一系统上共存却是十分关键。Linux程序库标准的命名方式，比如说是libfoo.so.1.2，而且给这个程序库一个libfoo.so.1的soname。如果此程序库是加到标准程序库的目录底下（ e.g. /usr/lib），ldconfig会建立符号链接libfoo.so.1 -> libfoo.so.1.2，使其正确的镜像能于执行期间找到。你也需要链接libfoo.so -> libfoo.so.1，使ld能于链接期间找到正确的soname。

所以，当你修正程序库内的错误时，或是是添加了新的函数进去时（任何不会对现存的程序造成不利的影响的改变），你会重建此程序库，保留原本已有的 soname，然后更改程序库文件名。当你对程序库的更改会使得现有的程序中断，那么你只需增加soname中的编号。此例中，

称新版本为libfoo.so.2.0，而soname变成libfoo.so.2。紧接着，再将libfoo.so的链接转向新的版本；至此，一切又恢复了正常。

其实你不需要以此种方式来为程序库命名，不过这的确是个好的传统。 ELF赋予你在程序库命名上的灵活性，会使得人不清楚状况；有这样的灵活性在，也并不表示你就得去用它。

假设你发现有个惯例：程序库主要的升级会破坏相容性，而次要的升级则可能不会。那么以下面的方式来链接，所有的一切就都会相安无事了。

```
gcc -shared -Wl,-soname,libfoo.so.major -o libfoo.so.major.minor
```

5. 文件配置

a.out(DLL)的共享程序库包含两个真正的文件与一个链接符号。就 foo这个用于整份文件做为样例的程序库而言，这些文件会是libfoo.sa与libfoo.so.1.2；链接符号会是libfoo.so.1，而且会指向libfoo.so.1.2。

在编译时，ld会寻找libfoo.sa。这是程序库的stub文件。而且含有所有执行期间链接所需的exported的资料与指向函数的指针。

执行期间，动态加载程序会寻找libfoo.so.1。这仅仅是一个符号链接，而不是真正的文件。故程序库可更新成较新的且已修正错误的版本，而不会损毁任何此时正在使用此程序库的应用程序。在新版(比如说libfoo.so.1.3)已完整呈现时， ldconfig会以一极微小的操作，将链接指向新的版本，使得任何原本使用旧版的程序不会有任何得冲突。

DLL程序库通常会比它们的静态副本要大得多。它们是以洞的形式来保留空间以便日后的扩充。这种洞可以不占用任何的磁盘空间。一个简单的 cp调用，或者使用makehole程序，就可以达到这样效果。因为它们的位址是固定在同一位置上的，所以在建立程序库后，你可以把它们拿掉。不过，千万不要试着拿掉ELF的程序库。

6.“ libc-lite ”

libc-lite是简单的libc版本。可用来存放在磁盘上，也可以结束大部分低级的 UNIX任务。它没有包含curses, dbm, termcap等等的程序代码。如果你的/lib/libc.so.4是链接到一个lite的libc，那么建议你以完整的版本取代它。

A.9 动态加载

A.9.1 基本概念

Linux有共享程序库，所以有一些照惯例是在链接时期便该完成的工作，必须延迟到加载时期才能完成。

A.9.2 控制动态加载器的运作

有一组环境变量会让动态加载器有所反应。大部分的环境变量对 ldd的用途要比对一般users的还要多。而且可以很方便地设置成由 ldd配合各种参数来执行。这些变量包括 LD_BIND_NOW 。正常来讲，函数在调用之前是不会让程序寻找的。设置这个标志会使得程序库一加载，所有的寻找便会发生，同时也造成起始的时间较慢。当你想测试程序，确定所有的链接都没有问题时，这项标志就变得很有用。

LD_PRELOAD可以设置一个文件，使其具有覆盖函数定义的能力。例如，如果你要测试内存分配的策略，而且还想置换 malloc，那么你可以写好准备替换的副程序，并把它编译成mallolc。然后：

```
$ LD_PRELOAD=malloc.o; export LD_PRELOAD
$ some_test_program
```

LD_ELF_PRELOAD 与 LD_AOUT_PRELOAD很类似，但是仅适用于正确的二进位类型。如果设置了 LD_something_PRELOAD 与LD_PRELOAD，比较明确的那一个会被用到。

LD_LIBRARY_PATH是一连串以分号隔离的目录名称，用来搜索共享程序库。对 ld而言，并没有任何的影响；这项只有在执行期间才有影响。另外，对执行 setuid与set gid的程序而言，这一项是无效的。而LD_ELF_LIBRARY_PATH与LD_AOUT_LIBRARY_PATH这两种标志可根据各别的二进位型式分别导向不同的搜索路径。一般正常的运作下，不应该会用到LD_LIBRARY_PATH；把需要搜索的目录加到/etc/ld.so.conf/里；然后重新执行ldconfig。

LD_NOWARN 仅适用于 a.out。一旦设置了这一项（ LD_NOWARN=true; export LD_NOWARN），它会告诉加载器必须处理致命错误（像是次要版本不相容等）的警告。LD_WARN仅适用于ELF。设置这一项时，它会将致命信息“ Can ’ t find library ”转换成警告信息。对正常的操作而言，这并没有多大的用处，可是对 ldd就很重要了。

LD_TRACE_LOADED_OBJECTS仅适用于ELF。而且会使得程序以为它们是由ldd执行的：

```
$ LD_TRACE_LOADED_OBJECTS=true /usr/bin/lynx
        libncurses.so.1 => /usr/lib/libncurses.so.1.9.6
        libc.so.5 => /lib/libc.so.5.2.18
```

A.9.3 以动态加载编写程序

如果你很熟悉Solaris 2.x所支持的动态加载功能的话，你会发现Linux在这点上与其非常相近。这一部分在H.J.Lu的ELF程序设计文件内与dlopen(3)的操作联机帮助（可以在ld.so的套件上找到）上有广泛的讨论。下面是一个简单样例(以-ldl链接)：

```
#include <dlfcn.h>
#include <stdio.h>
main()
{
  void *libc;
  void (*printf_call)();
  if(libc=dlopen("/lib/libc.so.5",RTLD_LAZY))
  {
      printf_call=dlsym(libc,"printf");
      (*printf_call)("hello, world\n");
  }
}
```

附录B　安装X Window窗口系统

B.1　安装X Window的提示

安装X Window(以下简称X)的注意事项如下：

- 尽快将所有文件复制下来。理想的状况是在开始安装系统之前，你可以将光盘上所包含的文件打印出来。
- 首先，对已有的配置文件尽量少做更动。如果你是第一次安装 X，我们推荐你使用缺省的目录来安装那些系统的各个不同的部分，只有在有确实需要更动目录名称时，才考虑重新安装。（如果你不这样做，当有些问题发生时，你将无法知道，它真的是一个问题还是你安装时犯了一个小错误。）
- 如果你是用这次发行的版本来建立一个标准的系统，你几乎不需要做任何更动便可完成安装 —— 配置文件已正确地设定好了。
- 当开始安装时，对大多数的系统而言，不需明确指定系统的形态，它的建立是根据C语言的预处理器来定义系统名称，从而自动找到的。
- 在你熟悉core版之前不要安装contrib 软件，因为它安装费时，且易产生问题，当熟悉系统之后，你可能总会修改一些配置参数。
- 如果真的需要使用不同缺省值的配置，请阅读文件 $TOP/util/imake.includes/ README ，你可得到关于如何改动的信息。

B.2　建立此版本

本次发行的软件分为两类：一部分核心软件由 X协会提供，另一部分则由其他厂商提供。要使核心软件这一部分能在大多数的地方重新配置、编译和安装存在一定的难度，而那些其他厂商开发的部分，也就是说，未经 X协会标准编译和测试的部分则只能自行逐一建立。在util/scripts中的ximake.sh可能对Imakefiles和Makefiles 非常有用。

几乎所有在核心软件中的 makefiles 都是由名叫 imake 的实用程序所产生的。所有的Makefiles 初始版本均包含了在哪些地方不能使用 imake （无疑地，在特定的机器上需要修正）。无论如何，它可以使用目录 util/imake.includes/中的文件site.def和*.macros中的参数，来重新配置核心软件。

B.3　安装摘要

欲加载和安装此次发行的X Window System，将需要：

1）读完本次发行的备忘录。

2）建一个你读取软件的目录（通常名称类似 /usr/local/src/X或/src/R31）。核心软件将需要大约30 MB的空间，用户软件需要超过80 MB的空间。

3）既然用户软件必须自行建立，你可以等到以后再加载。

4）阅读文件util/imake.includes/README以了解如何配置特定的X系统。同样地，依照目

录服务程序/ddx/ 中README文件来计划安装服务程序。

5）如果你计划在多部机器上编译这个版本并且要有一个分类文件系统，你可能希望每一个目的机器上依照util/scripts/lndir.sh 的说明来建立符号连接树，这样就允许所有的机器使用 X 的同一组原始程序。

6）如果你是在 Macintosh机器上安装，确定你是从目录服务程序 /ddx/macII执行R3 setup.sh程序。它用来在util/cpp中的公共源代码中建立一个C的预处理器，和修正一些放错位置的系统文件。如果你是在一部SUN计算机上，确定你在文件util/imake.includes/Sun.macros的上面设定了四个OS的参数。这样作可防止在不同的 SunOS 编译器下产生的错误。如果你是在Apollo上安装，确定你是使用 9.7.1 或更后面版本的C编译器，否则，服务程序将无法正确编译。

7）确定你已遵循所有和机器特性有关的指示，并且 imake 也是依你的机器的特性而配置的。

8）当你完成了配置，你可以准备安装 core版。注意那些在util/imake.includes/中的.macros文件。它们在接近顶端有一行设定一个 make的变量名称为BOOTSTRAPCFLAGS 。如果这个变量值是空白，你可以用下列的命令开始安装：

```
% make World > & make.world &
```

如果不是空白，你必须将此定义附加到命令行上。这可用 imake 对所 有的编译器设定特定的cpp 符号。（如果你在不同的系统安装，见 util/imake/imake.c）在所有的core版提供的macro文件中，只有A/UX需要这个标志：

```
% make BOOTSTRAPCFLAGS=-DmacII World >& make.world &
```

9）当make完成后，检查日志文件是否有问题。如果没有严重的问题， A/UX的用户可以忽略有关枚举类型冲突的编译警告， Applo 的用户可以忽略最优化的警告。

10）如果一切正常且建立了正确程序，测试一些不同的程序（如server、xterm、xinit、etc）。你也许需要在根目录执行server或xterm，如果你执行上有问题，第二部的工作站或终端机将很有用。

11）对你旧版的X头文件、二进制文件、字体、库等作备份。

12）到build tree的顶端并键入：

```
% make install >& make.install
```

在大多数的系统上， xterm 程序必需把setuid安装到根目录， xload 程序必需把setguid 安装到那些属于/dev/kmem 的文件中。

13）如需要安装手册，在build tree顶端键入：

```
% make install.man
```

14）如果你要建立和安装lint libraries，在build tree的顶端输入：

```
% make install.ln
```

如果是第一次安装 X，可能也需要以下的步骤。检查一下在 server/ddx目录中不同的README文件中附加的指示。

15）加入设备驱动程序或重新配置内核。

16）建立附加的虚拟终端机。查看操作系统说明/dev/MAKEDEV并询问系统管理员相关的细节部分。

17）阅读有关新的显示管理器xdm 的手册，这个程序提供了一个自动执行 X的方法，同时为了初学者提供了很多非常好的界面符号。

18）确定所有的X11 的用户的搜索路径中均为BINDIR的目录（通常为/usr/bin/X11）。

B.4 操作系统需求

X受欢迎的原因之一是与操作系统无关。虽然我们讨论的样例均为在 UNIX系统下，但有许多非UNIX操作系统的制造厂商也提供X。本次发行的服务程序已在下列系统建立：

4.3+tahoe
Ultrix 3.0 FT2（Ultrix 2.0）
SunOS 3.4
HP-UX 6.01
Apollo Domain/IX 9.7
IBM AOS 4.3
A/UX 1.0

如果使用较早的版本，可能招致一些麻烦，特别是服务程序将无法在 IBM 4.2A第二次发行版上执行。在server/ddx/ 中不同的README文件描述了对编译器、函数库、预处理器等等的特定需求。例如前述的A/UX 1.0的用户将需要建立一个新版的C预处理器，而 Apollo的用户将需要9.7.1版的C编译器。

你应在使用X前先确定你的网络系统和相互通信实用程序作用正常。如果像 talk和rlogin这样的程序无法工作， X大概也不行。

B.5 使用符号链接

本发行版使用符号链接以避免文件重复。如果你在未提供配置文件情况下建立本发行版，你必须在util/imake.includes/*.macros文件中检查LN配置参数。如果操作系统不提供软件链接，LN需被设定为建立硬链接或拷贝原始文件。

如果需要在安装后将本发行版移动到其他的机器，可以用 tar 替代CP或rcp以保留日期和链接，通常可用下列的命令完成：

```
% (chdir/usr/local/src/X;tar cf - .) | \
rsh othermachine “ (chdir/moredisk/X; tar xpBf -)"
```

可向系统管理员咨询以获得帮助。

B.6 配置本发行版

这个发行版广泛使用 imake 实用程序，它可从特定机器的 Imakefiles中产生机器特有的Makefiles。假如你不在乎可移植性，虽然我们建议你去使用imake 与 makedepend，但是仍可以使用 makefiles。

imake的配置文件在 util/imake.includes目录中。.makefiles文件是由一个称做imake.tmpl 的模板文件、一个机器特有的 .macros 文件与一个站点特有的 site.def 文件所产生的。除极少的例外情况，配置参数应该为 cpp 符号。 它可以在每个服务器的基础上定义，或在给定站点里的所有服务器上定义。不应该修改此模板文件。

util/imale.includes/README文件指出了每一个配置参数以及它可以设定的值。缺省值的选择是考虑到了可适用于多种机器且易于维护。站点特有的配置应在 site.def文件内使用下列语法描述：

```
#ifndef BuildParameter
#define BuildParameter site-specific-value
#endif
```

B.7 编译本发行版

配置参数设定后，你应该可以在构造树的最上层用下列命令来编译 core 软件：

```
% make World > &make.world &
```

若在建立过程中，将 make.log 这个特殊文件删除，则请不要将输出重定向到make.log中。依照使用机器的不同，这个步骤将花费 2～12 小时，且在大多数的机器上皆能顺利完成。

在所有 Makefiles 与相关性都建好之后，你必须重新启动以便使编译发生作用。 请在构造树的最上层输入下列命令：

```
%make -k > &make.out &
```

如果以后决定改变配置参数，你将需要作另一个完整的编译过程。

B.8 安装发行版

假如每一个编译皆成功，可以输入下列命令，从建造树最上层安装软件：

```
# make install
```

假如你不想安装在根目录，你必须建一些可写入的目录并安装它们，然后检查在 BINDIR 目录 (通常是 /USR/BIN/X11) 内的 xterm 与 xload 的所有权与保护模式 。xterm必须把setuid安装至根目录以便它能设定虚拟终端机与更新 /etc/utmp。 xload需要把 setuid 安装至根目录，或把setgid安装至拥有文件 /dev/kmem 的目录以使它们可以取得系统的平均负载。

假如 /etc/termcap 与 /user/lib/terminfo 数据库没有 xterm 的入口,可参考在客户机/xterm 目录内的样本入口。System V 的用户将需要用 tic实用程序编译 terminfo 入口。

假如你计划使用 xinit 程序去执行X，你可能会想产生一个将指定的 X指到适当服务程序的链接(通常的名称像 /usr/bin/x11/ 目录里的 Xmachine)。

假如你想安装使用手册，请检查在 util/imake.includes/ 里的MamDirectoryRoot、ManDir 与 LibManDir 三个配置参数，并在构造树的最上层输入：

```
# make install.man
```

假如你喜欢建立与安装 lint 程序库，在构造树的最上层输入下列命令：

```
# make install.ln
```

最后，确定所有用户在它们的 PATH 环境变量都有 BINDIR (通常是/usr/bin/x11/)。

B.9 内核与特殊文件的注意事项

在某些机器上，如果出现了一个新的设备驱动程序，则必须重建内核或至少经过重新配置。假如你以前从未执行X，你可能需要确定在你的内核配置文件中的csr位置能与你的硬件相匹配。另外，你应该确定在系统启动时内核会自动配置显示器。

你可能需要为显示器、鼠标或键盘建立特殊的设备。例如：

```
# /etc/mknod/dev/bell c 12 2 # for bell on Sun
# MAKEDEV displays   # for displays on the RT/PC
```

应把显示器设备文件上的保护模式设定成只有服务程序可以打开它。假如服务程序是以/etc/init启动，这个保护可以被根用户读/写，任何其他的人不能存取。

更详细的资料请看服务程序/ddx/ 目录里的 README 文件与手册。

B.10 测试本发行版

即使你计划使用 xdm 执行X，你也应该要从另一个终端机执行它，以便检查每个安装部分

是否已安装且正常工作。

来自服务程序的错误信息将会显示在终端机上，而不是写到xdm-errors 或/usr/adm/X?msgs文件中(? 是显示器的号码)。

测试服务程序最简单的方法是先进入 /usr/bin/x11 (或任何你已安装 X程序的地方)然后执行xinit, 如下:

```
% cd/usr/bin/X11
% xinit
```

你应该可以看到一个灰色的带状图案覆盖屏幕、一个形状像“ X ”的光标追踪着鼠标指针以及一个终端机模拟窗口。如果没有这样的显示，请按下列方式检查：

1) 假如没有显示出灰色屏幕背景，检查是否有在server/ddx/ 子目录内的README 里描述的任何特殊设备文件的使用权 (通常是保存在 /dev/中)。

2) 假如背景出现，但光标仍是白色方形。请确定字体是否已安装 (特别是 usr/lib/ X11/fonts/misc/ 里的字体 cursor.snf。见配置参数DefaultFontPath)。并确定在你的每一个字体目录都有一个文件叫做font.dir。这个文件是由 mkfontdir 程序产生的。服务程序用它来找出一个目录里的字体。

3) 假如光标出现但不追踪鼠标指针，请确定相应特定设备文件 (像 /dev/mouse) 已安装 (见服务程序的 README 文件)。

4) 假如服务程序启动但不久之后变成黑色，表示启动的客户程序 (xterm 或xdm)停止运行了。请确定 xterm 安装在根目录并建立足够的虚拟终端机。假如你正执行 xinit, 且在你的home目录一个有叫做 .xinitrc 的文件,确定它是可执行的而且上一个它所启动的程序在前台执行 (亦即命令行尾端没有一个 & 符号)。否则, .xinitrc 将立刻完成, 这也是 xinit 所假设的以及你所想要的。只要你有正确工作的初始窗口, 试着从 xterm 执行其他的程序。若想用 uwm 窗口管理器定位一个新窗口, 可在闪动的方框出现时按下按钮 1 (通常在鼠标的最左边按钮) :

```
% xlock -g 200x200-0+0 &
% uwm &
% xlogo &
% xeyes &
...
```

X Window现在可以使用了, 读一下手册, 看看新字体, 你会从中得到乐趣。

B.11 建立额外的虚拟终端机

因为每个 xterm 需要一个不同的虚拟终端机，所以你应该给它们建立很多个虚拟终端机(在一个小型多用户系统你可能至少要 32 个)。每个 ptr 有两个设备，一个主控，一个从属，它们通常叫做 /dev/tty/[pqrstu][0-f]与 /dev/pty[pqrstu][0-f]。假如你没有最少的p与q行配置，你应该要求系统管理员加上它们。这通常是以 /dev里执行的 MAKEDEV script来完成:

```
# cd/dev
# ./MAKEDEV pty0
# ./MAKEDEV pty1
```

B.12 从 /etc/rc启动X

本发行版提供一个可以从系统启动文件 /etc/rc中 执行X服务器的实用程序，叫做 xdm。由于考虑到可以很容易地适应不同工作站的需求，xdm可以周到地维护服务程序的执行，提示输入用户名与密码并管理用户的对话。当前样本配置使用外壳脚本，以提供相当简单的环境。

xdm 灵活性的关键是它广泛使用资源，允许站点管理员快速地测试不同的设置。当 xdm 启动时，它读入一个配置文件 (缺省是/usr/lib/X11/xdm/xdm-config 但可以用命令行指定-config) 列出不同的数据文件、缺省参数以及启动与关机时执行的程序。因为它使用标准的工具箱资源文件格式，所以任何可以在 xdm-config 文件设定的参数也可以在命令行使用 -xrm 选项指定。

缺省结构如下：

```
DisplayManager.servers: /use/lib/X11/Xservers
DisplayManager.errorLogFile:/use/lib/X11/Xdm-errors
DisplayManager*resource:   /use/lib/X11/Xresource
DisplayManager*startup: /use/lib/X11/Xstartup
DisplayManager*session: /use/lib/X11/Xsession
DisplayManager*reset:   /use/lib/X11/Xreset
```

服务程序文件包含要启动的服务器列表。 errorLogFile 是 xdm 重定向输出的文件。resource文件包含 xdm Login 窗口缺省的资源。特别的地方是可以指定特殊键的顺序 (在xlogin*login.ranslation resource)，startup文件应该是一个程序或可执行的脚本，它可以在用户打入正确的密码之后执行。它对站点进行初始化、记录等的挂接。 session 入口是用于指定会话管理程序或可执行脚本的名称。假如用户在主目录内没有一个可执行的 .xsession 文件，有一个简单的版本可提供一个简单的 xterm 窗口与 uwm 窗口管理器。最后，在用户注销之后，将执行一个复位程序或可执行的脚本。如果希望使用缺省配置执行 xdm，在boot文件加入下面一行(通常叫做 /etc/rc 或 /etc/rc.local)：

```
/usr/bin/X11/xdm &
```

毫无疑问，大多数站点想建立它们自己的配置。我们建议你们将站点特有的 xdm-config 文件与其他的 xdm 文件放在不同的目录。假如将文件保存在 /usr/local/lib/xdm中,可用下面的命令启动xdm：

```
/usr/bin/X11/xdm -config /usr/local/lib/xdm-config &
```

许多服务程序都假设它们是尝试读取键盘的唯一程序，这样键盘设定成非阻断式 I/O。不幸的是，某些 /etc/getty 的版本 (特别是A/UX) 将立刻显示一个零长度读取的连续流，因为将它解释成 EOF 文件尾指示。最后 /etc/init 将取消登录，一直到某人打入下列命令：

```
# kill -HUP 1
```

在 A/UX 下，一个变通的方法是禁止从控制台登录，且从 /etc/inittab中执行 xdm。然而，请确定你存有一个旧的 /etc/inittab 拷贝，以备在出错状况以及必须从网络上或单用户模式恢复登录的时候使用。

另一个较少为人彻底研究的是如何设定一个帐号，它的外壳是在client/xdm/ 里的 xdmshell 程序。缺省状况下，这个程序不会安装 ，所以站点管理员需要特别查看它是否符合需求。xdmshell 实用程序确定它执行于适当的终端机型号、启动 xdm、等待它完成与重置控制台 (如果需要的话)。假如 xdm 资源文件包含一个终止 —— 显示器动作的对应，就像下列命令：

```
xlogin*login.translations: #override Ctrl<Key>R: abort-display()
```

然后在 xdm login 窗口按下指定键 (上例是 <Ctrl-R>)， 控制台即可以被恢复。

xdmshell 程序通常安装在根目录，但仅能被一些特殊的用户执行。这样以 xdmshell作为外壳的帐号的人是唯一的：

```
% grep xdm  /etc/passwd
X:aB9i7vhDVa82z:101:51:Acount for starting up X: (contd.)
/tmp:/etc/xdmshell
% grep 51 /etc/group
xdmgrp:*:51:
```

```
% ls -g /etc/xdmshell
-rws--x--- 1 root xdmgrp 20338 Nov 1 01:32 /etc/xdmshell
```

假如 xdm 资源不曾有一个键对应至 abort-display() 动作，一般用户将没有方法直接登录到控制台。

1. 支持旧系统 —— 从 /etc/init 启动X

警告 下面所提供的适合较老的系统且在未来的发行版将不再提供。

Ultrix 与 4.3bsd 使用 /etc/ttys 配置文件的一个新的扩充格式，它允许你指定一个窗口系统与初始化时被执行的程序。虽然一般人比较喜欢使用 xdm,但系统仍然提供可在 xterm里启动X与一个从 /etc/ttys 初始的 xterm 窗口。

2. 建立 ttyv 终端机

因为大多数的 /etc/init 版本需要给每个 /etc/ttys 的入口一个真实的终端行，所以你必须给每一个显示器指定一个虚拟终端机。虽然正常状态下 xterm动态地配置一个 pty，但是 -L 选项可以强迫它使用从 /etc/init 传过来的虚拟终端机。

依照协议，最高的小设备号码改名成[pt]tyv0，次高的改为 [pt]tyv1，依此类推。最高的被改为最低的，可使他们不以正常地从低至高的搜索方法 (大多数程序配置一个 pty 的方法) 取得。在仅有p与q的小系统，下面的命令可用以将两个显示器设定成v终端机。

```
# cd /dev
# mv ttyqf ttyv0; mv ptyqf ptyv0
# mv ttyqe ttyv1; mv ptyqf ptyv1
```

从系统管理员那里可以获得更多的帮助。

3. 在 /etc/ttys中增加更多的窗口入口

只要你已经改变了虚拟终端机的名称，就能为它们在 /etc/ttys 增加一个入口。再次提醒这个仅对具有以新的 4.3bsd 格式启动窗口系统的系统有效，不能用在使用 4.2bsd 小进入口格式与 /etc/ttytype 的较老系统上使用。

x服务程序手册里包含关于设定 /etc/ttys 入口的更详细叙述。通常，ttyv 置于文件的底部且类似如下：

```
ttyv0"/usr/bin/X11/xterm -L -grometry 80x24+0+0 -display :0"(contd。)
xterm off window="/usr/bin/X11/X :0"
```

注意,给X服务程序命令的服务程序号码的参数必须在前面置一个冒号。额外的命令行选项可以在 xterm 命令行或X命令行指定。然而,许多 init 的版本具有相当小的程序名称缓冲区，限制了入口的长度。也有某些版本不允许在入口里有 pound记号，这意谓着不能指定任意数目的颜色，这也是为什么要使用 xdm 的原因。

只要你曾经增加或改变入口，你必须通知 init，告诉它重读 /etc/ttys与重新启动。在根目录下打入下列命令即可以达到这个目的：

```
# kill -HUP 1
```

这将会中止命令行上存在的进程，所以应该只能由系统管理员去执行它。