第一章　Linux 基础

在本书第一篇中，假定你对 Linux 或任何 UNIX 类操作系统完全没有经验或只有很少经验。从一个新用户开始，也就是说，从一个刚从 Linux 系统管理员那里取得帐号，登录名和口令的用户开始，逐步掌握 Linux 基础知识。

这是一种理想的学习情况。但是，我知道许多读者在一开始就不仅仅是一个新用户，他还必须同时担负起系统管理员的责任。处于这种情况下，最好交替阅读第一篇和第二篇前几章的内容。这样你就能在学习基础知识的同时，完成最少必需的安装和系统管理任务。事实上，这些任务并不像听起来那么困难。因为现在可供使用的许多 Linux 发布(distribution)中几乎所有的配置参数都有默认值。使它们用起来非常方便。但是，你仍然不能掉以轻心。对于一个完整的 Linux 操作系统的某些部分的安装和配置仍然需要十分小心地对待。

1.1　登录

作为开始，你的 Linux 将发出下列登录提示符，邀请你登录到系统中来：

```
login:
```

在这里敲入用户登录名，并按 enter 或 return 键。如果你的帐号设置了口令，系统将发出下列提示符请你输入口令：

```
password:
```

在输入口令后，再按 enter 或 return 键。在屏幕上不显示作为口令输入的字符。如果正确地送入了登录名和口令，系统将送回提示符请你输入命令。如果在登录过程中出现了错误，系统将发出登录错误信息：

```
Login incorrect
```

并发出新的登录提示符让你重试。注意，如果你是新用户又是系统管理员，在一般情况下，应该尽量避免使用 root 登录帐号。除非要完成系统管理任务，在非用不可时你才动用它。理由是：root 是一个特权登录帐号，它拥有很大的权力。它能越过 Linux 正常的安全和完整性检查。即使一个有经验的管理员，偶尔犯下耗费时间和代价高昂的错误也是常有的事。

要注意的另一点是 Linux(大多数 UNIX 类系统也一样)区分大、小写。小写字母和对应的大写字母被看作是不同的字符。在多数情况下，Linux 要求用小写字母输入字和命令，大写字母不起作用。

作为普通用户帐号，当你成功地完成登录后，系统将执行一个称为 shell 的程序。任何程

序，当它正在系统内执行时，被称为进程(process)。正是你的 shell 进程给你提供了命令行提示符。作为默认值，对非特权用户用美元符号 $ 作提示符，对系统特权用户（root）用 # 号作提示符。shell 还承担从键盘接受输入的命令，并安排它们执行的任务。

由于历史原因，实际上有许多不同的 shell 程序可供使用。它们大致提供相同的功能。在本书中将集中使用自由软件基金会（GNU）提供的称为 bash 的 shell 程序。它也是大多数 Linux 发布默认的 shell 程序。

一旦出现了 shell 提示符，就可以送入命令名称和命令所需要的任何参数。shell 将执行这些命令。如果一条命令花费了很长的时间来运行，或者在屏幕上产生了大量的输出，你可能希望中断它。在它正常结束之前，停止它的执行。对多数命令讲，你可以从键盘上按 ctrl-c（即在按下 ctrl 键后，同时按 c 键），用它来送出中断信号。

当你准备结束你的登录对话过程(session)时，可以在提示符后面送入 logout 命令，退出登录。还有其它退出登录的方法，最常用的是在提示符后面送入文件结束符(EOF)。这可以在键盘上按 ctrl-d 来实现。这里的问题是：shell 可以被设置成忽略 EOF。一些 Linux 发布就是这样默认设定的。我们在后面讨论 shell 的细节时，你会知道如何定制 shell 的响应方式。

1.2 文件系统的层次结构

当你完成登录后，下一任务是熟悉基本文件系统的结构。Linux 文件系统包含 3 类文件：

普通文件：这些文件只是字节的集合。系统没有在文件中加入特定的结构。它们用作文本文件(包括源程序文件)，程序使用的数据文件，以及程序本身的可执行二进制文件。

目录文件：目录是一种结构。它容许将一组文件放在一起。从概念上讲，目录好比是一个容器，可以用来存放其他文件和目录。事实上，目录本身只包含其他文件的名称和一些类似如何从磁盘上找到这些文件的简单信息。由于一个目录可以包含子目录名称，文件系统形成一个层次结构。

特殊文件：许多不同的文件类型属于这一范围。当你深入学习时将看到这一切。简单讲，特殊文件与进程之间的通信以及进程和连接到机器的各种各样的外部设备之间的通信有关。

所有这些类型的文件均放在一个大的树形层次结构中(见图 1.1)。树的顶部是一个单独的目录，称为根(root)目录(请勿与 root 登录名相混淆)。并且用斜杠符号 / 表示根目录。在根目录下，有一些用于不同目的的标准子目录和文件。这些高层的目录和文件结构只是从传统的意义讲是标准的，但并非一定要那样去做。

图 1.1 表示一个根目录(/)包含了 dev，bin 和 home 子目录。接下去 home 目录又包含了 mike 和 lynne 目录等等。在目录的最底层包含普通文件和特殊文件(如 a 和 b)。有了这些文件和目录之间的关系后，你可以说 home 是 mike 的双亲目录，而 a 是 proj1 目录的一个子文件。

两个或多个文件或目录具有相同名称的情况常常会出现。只要它们包含在不同的双亲目录中，就不会发生混淆。很明显，需要有无二义性地引用特定文件的办法。这可以用指定路径名的办法来实现。路径名是一串用斜杠字符 / 分隔的(目录或文件的)名称。如果这串字符用斜杠字符开头，则称为绝对路径名。它表示从根目录开始。例如：proj1 目录的绝对路径名是：

```
/home/lynne/proj1
```

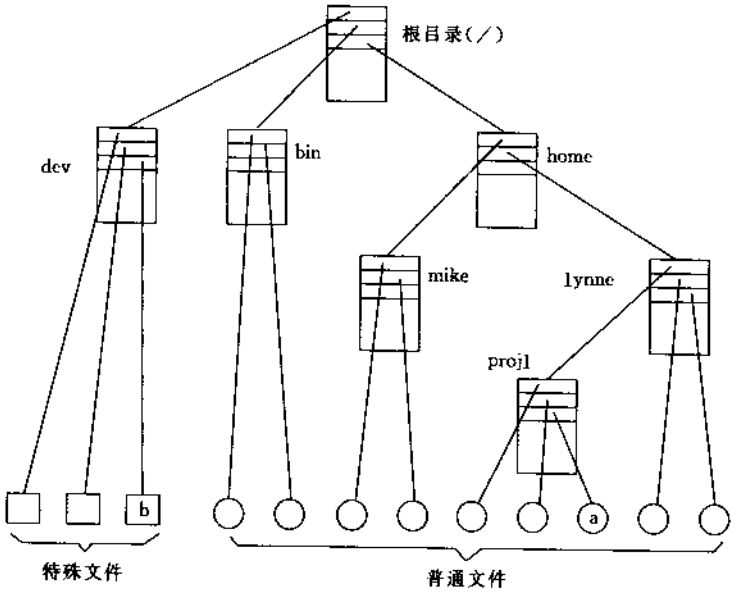

图 1.1 主要的文件类型形成树形的层次结构

而特殊文件 b 的绝对路径名则是：

```
/dev/b
```

正像你看到的，绝对路径名从根目录开始，并由此向下延伸。也可以用其他目录名作为开始来指定文件的路径名。用这种方法给出的路径名称为相对路径名。例如，从 lynne 目录(或严格地称为 /home/lynne 目录)到普通文件 a 的相对路径名是：

```
proj1/a
```

它表示在当前目录 /home/lynne 下，通过包含 a 文件的 pooj1 目录取得 a 文件的路径。

当你改变目录时，新目录成为当前的工作目录。当前工作目录的概念相当重要，每当在 Linux 系统上执行一个程序时，作为结果的进程将当前工作目录设为它的内部状态的一部分。正是这个目录，进程将它用作访向任何文件的相对路径名的起点。除非进程为自己另设特殊的当前工作目录(多数情况下不这么做)，一般情况下取用户启动程序时所处的目录作为默认的当前工作目录。

如何指定从 /home/mike 目录到 a 文件的相对路径名？问题是 /home/mike 目录和文件 a 处在目录树的不同分支上。为解决这一问题，需要知道另外两个名称，这两个名称都自动地包含在每个 Linux 目录中。它们分别称为 .(发音为 dot) 和 ..(发音为 dot-dot)。. 是当前目录的别名。而 .. 则是当前目录的双亲目录的别名。在指定相对路径名时，.. 允许你回到目录树的上一层。有了 .. 的标记法，上面问题的答案就成为从 /home/mike 目录回到上一层目录 /home，然后从右面的分支向下到达 a 文件。得到的相对路径名如下：

```
../lynne/proj1/a
```

顺便说一下,虽然根目录没有上层目录,但仍然保留名称 .. ,它被用来指向根目录本身。根目录作特殊处理,它就是它自身的双亲目录。这也表示在指定相对路径名时,使用太多的..不会造成问题。

1.3 考察文件系统的层次结构

在登录时,有一个特殊的目录和你的登录名联系在一起。它被称为你的起始(home)目录。起始目录实际是最初的当前工作目录,这是属于你的整个目录树的起点。你在系统工作时所建立的所有文件和目录,一般都放在你的起始目录下面。

在整个系统的目录层次结构中,找出起始目录的实际位置的最简单办法是在登录后直接使用 pwd 命令。pwd 命令告诉你:当前工作目录,也就是现在所处的目录是什么。在刚刚登录时,它将是起始目录。在起始目录下使用 pwd 命令将得到类似下面的输出:

```
$ pwd
/home/you
```

其中 /home/you 应该用你自己的起始目录的绝对路径名代替。

为了将当前工作目录从起始目录转换到目录层次结构中的其他位置,可以用 cd(改变目录)命令。所以,要将目录改到根目录就应该用下面的命令:

```
$ cd /
```

在 cd 命令中用所要的目标目录的路径名作为参数。

为了得到包含在当前目录下的文件和目录名称的清单,可以使用 ls 命令。对一个典型的 Linux 系统讲,在根目录下使用 ls 命令将得到下面的输出:

```
$ ls
bin     dev     home          mnt     sbin     var
boot    dos     lib           proc    tmp      vmlinuz
cdrom   etc     lost + found  root    usr
```

Linux 目录层次结构中最重要的分支表示在图 1.2 中。应该用 pwd,cd 和 ls 命令去考察你自己机器上的目录层次结构。

如果想从目录树中任何一点回到起始目录,可以用 cd 命令。并用相应的路径名作为参数。但是,也可以只送入 cd 命令而不给参数。这是使直接回到起始目录的一条捷径。在图1.2所示的 Linux 目录树中,各主要分支包含的文件的一般功能见表 1.1。

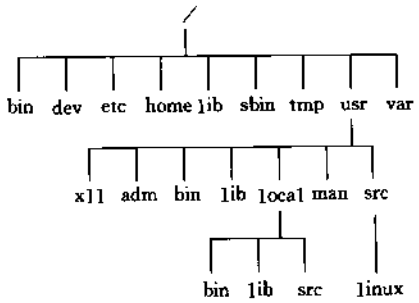

图 1.2 主要的 Linux 目录树结构

表 1.1 主要目录清单

/bin	二进制可执行命令
/dev	设备特殊文件
/etc	系统管理和配置文件
/home	用户起始目录的基点
/lib	标准程序设计库
/sbin	系统管理命令
/tmp	公用的临时文件存贮点
/usr/X11	X-windows 系统文件
/usr/adm	系统管理。数据文件
/usr/bin	其他的可执行命令
/usr/lib	库和软件包的配置文件
/usr/local/bin	本地增加的命令
/usr/local/lib	本地增加的库
/usr/local/src	本地命令的源文件
/usr/man	系统联机手册页
/usr/src/linux	Linux 内核源程序文件
/var	某些大文件的溢出区

1.4 口令文件

在 Linux 系统中,口令(password)文件是系统的主要文件之一。它将前面看到许多内容结合在一起。口令文件的内容包含:所有用户的登录名的清单;为所有用户指定的起始目录的具体位置;在登录时使用的 shell 程序的名称。口令文件还用来保存用户口令;给每个用户提供系统识别号;当一组用户需要为联合项目协同工作时,将这些用户编成组。

口令文件本身是一个允许每个用户阅读的普通文本文件。它保存在下列文件中:

```
/etc/passwd
```

可以很简单地用下面的命令行将该文件的内容列出来:

```
$ cat /etc/passwd
```

cat 命令在这里只用来列出文件的内容。它的全部用法将在下一章 2.3 节中说明。

口令文件中的每一行是一个用户登录名的所有有关信息的记录。每一条记录用冒号:分隔成 7 个字段(field),具体格式如下:

```
name:password:uid:gid:comment:home:shell
```

自左至右,7 个字段的用途如下:

name 此字段包含用户登录名。这是用户登录时必须正确地敲入的名称。

password 这是用户的口令。如果不认识你的口令,不必感到惊讶!这是正常的,因为口

令加了密。当你在登录过程中敲入口令时,系统用同样的方式对它加密。并与这一字段的内容进行比较,以此来确定是否让你访问系统。如果这一字段是空的,表示对该用户无需口令。

uid 这是系统用来分配用户识别号的字段。一旦用户登录后,系统将用 uid 而不是用登录名来查找用户。

gid 有时候,一批用户需要在一个组内共同完成同一个项目。在这种情况下,允许他们共同访问一组特定的目录和文件是很有用的。这可以在这个字段内给小组的全体成员分配同一个组识别号(gid)来实现。

comment 这是注释字段。常用来保存用户的真实姓名和个人细节。

home 这一字段用来保存用户的起始目录的绝对路径名。当用户登录时,系统从这一字段取得用户起始目录路径名。

shell 如果这一用户登录成功,要执行的命令的绝对路径名就放在这一字段。这可以是任何命令。但是对普通用户帐号讲,这将是 shell 的路径名。如果此字段没有给出路径名,它的默认值是 /bin/sh。

因为口令的加密算法十分安全,口令文件(/etc/passwd)是公开可读的。然而,如果选用一个简单的字做口令,譬如说,从词典中找一个字。那么,只要对词典中的字一一加密,并和口令字段比较是否匹配就是一件简单的事。克服这一问题有两种办法,一种办法对加密口令加以隐藏。Linux 有一些软件包专门用来干这件事。这些软件包将加密的口令隐藏在单独的文件中(常用 /etc/shadow)。这一文件不允许公众进行访问。有了额外的文件就允许给用户记录增加更多的字段,如用来确定隔多长时间用户就必须更换口令等。另一种办法更简单,但也同样有效,这就是选择好的口令。一个好的口令至少有 6 个字符长,并包括大小写字母,数字和/或标点符号的混合使用。如果这样做难以记忆的话,就用一个简单字,但将其中若干字符替换成看上去相似的其它字符。例如,shelter 和 5he!tER。这种技术能给出更安全的口令。但是高级的口令解密程序仍然能发现它。

为了达到最大限度的安全,建议你经常改变口令。在 Linux 系统中变换口令是方便的。你只要送入下列命令就行了:

```
$ passwd
```

如果帐号设置了口令,系统就提示输入旧口令。然后两次提示输入新口令。由于系统不在屏幕上显示送入的口令,输入两次新口令是必要的。如果两次输入的内容不一致,系统将不改变你的口令。这样做确保你正确地输入了新口令。

练习

1. 如何找到起始目录的绝对路径名。在自己的系统上,起始目录的绝对路径名是什么?
2. 在系统上如何发现哪些用户没有设置口令?
3. 将当前工作目录从 /usr/bin 转到 usr/local/bin 需要用什么命令?先用绝对路径名,再用相对路径名。
4. 考察系统的目录层次结构,并找出 ls 和 pwd 命令的绝对路径名。

答案

1. 这里有几种可能。首先,可以用 cat 命令查看口令文件的内容:

```
$ cat /etc/passwd
```

然后从口令行的第6个字段中提取起始目录的绝对路径名。另一种办法使用下面的命令序列：

```
$ cd
$ pwd
```

cd命令将使你从目录层次结构中的任意位置回到起始目录。然后，pwd命令将显示当前的工作目录。在我的系统上这样做时，得到我的起始目录为：

```
/home/pc
```

2. 所有的口令都以加密的方式存在 /etc/passwd 文件中。要找出哪些用户没有设置口令，只要逐行检查第2字段。第2字段为空白表示这些用户没有设置口令。
3. 使用绝对路径名将当前工作目录从 /usr/bin 转到 /usr/local/bin 可以用下面的命令：

```
$ cd /usr/local/bin
```

使用相对路径时，可用下列命令取得同样效果：

```
$ cd ../local/bin
```

4. 根据表1.1，大多数可执行的用户命令存在/bin和/usr/bin两个目录下。用 ls 命令搜索这两个目录就能得到这两个命令的路径名：

```
/bin/ls
/bin/pwd
```

在大多数系统中，实际还提供了另一条命令。它使你能找出任何能执行的命令的绝对路径名。它的名称是 which。用这条命令可以取得同样的结果：

```
$ which ls pwd
/bin/ls
/bin/pwd
```

第二章　Linux 命令

讨论了登录和文件系统结构以后，现在我们来讨论 Linux 命令的使用。Linux 有数以百计的命令，本书篇幅的限止不能一一加以详细讨论。这里只选择少量常用的命令进行重点讨论。然而，你将看到 Linux 拥有一套组织良好的联机文档，而且它的内容尚在不断扩充。正确掌握联机文档的使用，有助于全面掌握 Linux 命令的使用。

一般说来，UNIX(因而也包括 Linux)命令在执行时非常安静。这就是说，它们不产任何不必要的输出。所以，命令成功地执行完毕的一般标志是得到送入下一条命令的提示符。

2.1　目录的层次结构

如果执行下面的命令：

```
$ cd /
$ ls
```

就将当前工作目录改为根目录，然后从 ls 命令得到类似下面的输出结果：

```
bin     dev     home          mnt     sbin     var
boot    dos     lib           proc    tmp      vmlinuz
cdrom   etc     lost + found  root    usr
```

前一章中说过每个目录都包含 . 和 .. 的名称，为什么它们不在 ls 命令输出的清单中出现？事实上，这两个名称始终在每个目录中。如果在清单中列出就得到一个或二个小点。为了保证一般情况下它们不在清单中出现，ls 命令有一条默认的内部规则：任何以句点开始的名称不在清单中列出。这条规则不仅适用于 . 和 .. ，也适用于任何以句点开头的其他名称。

这提供了一种方便的机制，使一些文件包含在目录中，但在查看目录内容时不列在清单中。但不要错误地认为这提供了建立秘密文件的机制。事实上，用 ls 命令列出目录中包含的所有文件是很方便的，包括那些以句点开头的文件或目录名称。

在许多场合下，使用一条命令时，希望它完成默认功能以外操作，或者抑止它的某些常规功能。这时可以在命令中加入命令行开关(command line switch)来打开或关闭需要的功能。为了一致起见，开关值前面要加上连字符(-)。表示这是开关而不是普通的命令参数(如文件名等)。

对 ls 命令讲，要它列出目录中所有文件时应该加 -a 开关，相应的结果如下：

```
$ cd /
$ ls -a
.       cdrom     etc          mnt     tmp
..      dev       home         proc    usr
```

```
bin     dos         lib             root     var
boot    etc         lost+found      sbin     vmlinuz
```

ls 命令在任何命令行开关后面还可以给出目录的名称。在这种情况下，列出的将是指定目录的内容而不是当前目录的内容：

```
$ ls -a /tmp
.         .X11-unix  elm.rc.OLD      ls        tagfile
..        NEAT       elv_61.1        modes
.XO-lock  clocks     hosts.OLD       nets.OLD
```

请注意：在机器上执行这条命令时，会得到不同的结果。因为 /tmp 目录是供任何用户保存临时文件的公用目录。

当开始在 Linux 系统上工作时，譬如说编写程序或运行一些软件包。将开始建立你自己的文件。一般讲，应该将所有文件都放在起始目录下。但是，如果简单地将所有文件放在起始目录下，很快就会变得难以从中找出所要东西。只要看一下 /usr/bin 目录就知道这是怎么回事了。为了解决这一问题，需要对文件空间进行一番设计。可以在起始目录下建立小型的目录层次结构，并运用一些简单的自律规则保证在存储新文件时，将它们放在正确的位置。

为了建立新目录，可以用 mkdir 命令，并用新目录名称作为命令参数：

```
$ cd
$ mkdir bin text
```

第一个 cd 命令不带参数，使你回到起始目录。而 mkdir 命令则在起始目录下建立两个新目录。分别称为 bin 和 text。可以用单个 mkdir 命建立任意数量的新目录。这表示 mkdir 命令可以接受可变数目的参数值。作为一般规则，许多命令都可接受可变数目的名称参数清单，并对整个清单进行正确的操作。

建立目录后，下面的任务就要复制一些文件到目录中来。Linux 复制文件的命令称为 cp：

```
$ cp  /etc/passwd  text/mypass
```

假定送入这一命令时，仍在起始目录下。它将系统的口令文件（/etc/passwd）的副本用名称 mypass 存在 text 目录下。在这种形式下，cp 命令取两个普通文件的路径名作为参数，并将前一个文件的内容复制到后一个文件中，cp 命令还可以有另一形式，最后一个参数不是普通文件名而是一个目录名。这时，可以用可变数目的文件名清单来代替第一个参数，清单中所有文件都将复制到给定的目录中：

```
$ cp /etc/passwd /etc/motd text
$ ls text
motd    mypass  passwd
```

注意:上例中用了 cp 命令的第二种形式。在这种情况下,没有机会改变复制文件的名称。

顺便说一下,文件 /etc/motd 包含系统每天的日常信息(message of the day)。默认地在用户登录时系统将这一文件的内容列出让用户看。这表示如果系统管理员要向用户传递信息时,最简单的做法就是将这些信息放在这个文件中。

文件和目录既能被建立,也能被删除。为了删除目录,可以用 rmdir 命令:

```
$ rmdir bin
$ rmdir text
rmdir: text: Directory not empty
```

为安全起见,rmdir 只在目录为空时(. 和 .. 目录项除外)才起作用。上例中 bin 目录已被删除,而 text 目录则没有被删除。要删除 text 目录,先应删去其中的 3 个普通文件。普通文件可以用 rm 命令加以删除,如:

```
$ rm  text/mypass
```

2.2 文件系统

一般说来,一个大的硬盘空间要分成便于使用的,容量适度的几个磁盘分区(disk partition)。一个较小的磁盘可以作为单个分区对待。不管用什么方式,当一个系统大到有几个分区用作磁盘存储时(不管是在一个硬盘上还是在几个硬盘上),有一个问题要解决。系统如何处理这些分区使用户都能访问它们?一种可能的办法是每个分区都有单独的根目录,然后使用从特定分区的根目录开始的路径名来指定文件名。但是,到现在为止你所看到的只是一个单一的目录层次结构,而不是每个分区一个目录层次结构。

事实上,Linux 系统中每个分区都是一个文件系统,有它自己的顶层目录和下面的目录层次结构。然后将这些单独的文件系统形成一个系统的总的目录层次结构。办法是:将一个文件系统的顶层目录装配到另一个文件系统的子目录上,使它们形成一个无缝的整体(见图 2.1)。

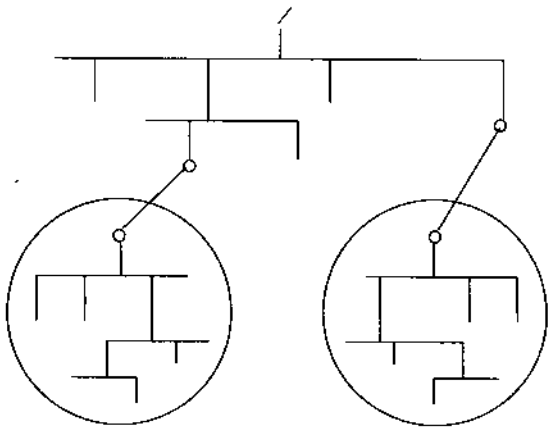

图 2.1 三个文件系统联合形成一个层次结构

存在磁盘分区中的文件，不管它是什么类型的文件，都给它分配一个号码，称为它的索引结点号(inode number)。它实际是存在盘上的一个数组的入口的索引号。数组的每个元素是一个索引结点(inode)，它保存了一个文件的管理信息（如文件是在什么时候建立的，文件属于谁，文件的数据块存在磁盘分区中什么位置等）。正是这个文件的索引结点号和文件的名称同时保存在目录中。所以，从本质上讲，目录只是将文件名称和它的索引结点号结合在一起的一张表，目录中每一对文件名称和索引结点号称为一个连接(link)。

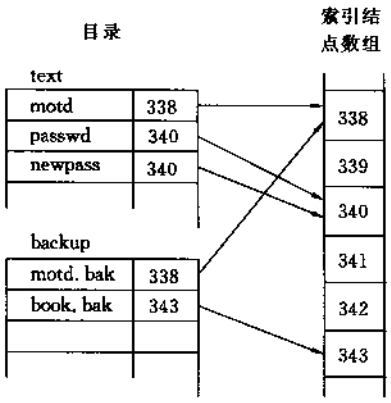

图 2.2　单个文件可以有多个名称

有了这些概念后，就没有理由认为同一个索引结点号不能出现在一个以上的连接中。就是说单个文件允许有多个有效路径名。这一概念表示在图 2.2 中。在许多场合下，这一点可能很有用。一种可能性是允许你建立连接到重要文件的目录，以提供“不删除”功能。防止偶尔删除其中的任何文件。能做到这一点的原因是索引结点有一个以上的连接。只删除了你的连接，索引结点本身和其它的连接仍然不受影响。只当对只有一个连接的文件发出删除命令时，索引结点本身，文件的数据块以及目录的连接才会被释放。

可以使用 ln 命令对一个已经存在的文件建立一个新的连接。命令中用已经存在的文件的路径名后面跟以新连接的路径名作为参数：

```
$ ln text/passwd text/newpass
$ mkdir backup
$ ln text/motd backup/motd.bak
```

为了看到索引结点号以检查所要连接是否已经建立，你可以用带 -i 开关的 ls 命令来实现：

```
$ ls -i backup text
backup:
    338 motd.bak
text:
    338 motd        340 newpass        340 passwd
```

上例中如果改变 text/motd 文件的内容，那么 backup/motd.bak 文件的内容也随之改变。因为它们就是同一个文件。问题在于每个文件系统(也就是每个分区)有它自己的索引结点数组。所以，只在一个文件系统中索引结点号才是唯一的。这表示不能用 ln 命令来建立不同文件系统之间的连接。譬如说，起始目录下的 text/motd 文件和不同文件系统中的另一个路径名之间的连接。这是因为在另一个文件系统中，索引结点号 338(这是起始目录所在的文件系统中 text/motd 文件的索引结点号)可能是另一个完全不同的文件。

如果需要在不同文件系统的路径名之间建立连接，就不能使用上面的共享索引结点技术。在这种情况下，要用 ln -s 来建立两者之间的符号连接(symbolic link)。符号连接是 Linux 特殊

文件类型的一种。事实上，它只是一个文本文件，其中包含它提供连接的另一个文件的路径名。另一个文件是实际包含所有数据的文件。所有读写文件内容的命令，当它们被用于符号连接时，将沿着连接前进去访问实际文件。很明显，这样做要付出少量的时间代价，但实际增加的时间很少，不会构成实际差别。

当需要在两个文件之间建立连接时，如果要保证所做的能移植到其他系统上，就应该使用符号连接。因为它不仅可以用于单一的文件系统中，也可用于文件系统之间，而索引结点连接只能用于单一的文件系统中：

```
$ ln -s text/motd backup/motd2.bak
$ ls -i backup text
backup:
    338 motd.bak    328 motd2.bak
text:
    338 motd        340 newpass     340 passwd
```

另一个用来操纵索引结点连接的命令是 mv。这个命令用在同一文件系统中，将目录连接从一处移到另一处。效果上它完成 ln 命令同样的功能，但它移去文件旧的连接：

```
$ mv text/newpass backup/passwd.bak
$ ls -i backup text
backup:
    338 motd.bak    328 motd2.bak    340 passwd.bak
text
    338 motd        340 passwd
```

mv 命令除了在目录之间移到文件外，还有另外的用途。如果新旧文件在同一目录中，它完成 rename(重新命名)命令相同的功能。Linux 系统中设有专门的 rename 命令，只能用 mv 命令来代替：

```
$ mv backup/motd2.bak backup/motd.sym.link
$ ls -i backup
    338 motd.bak      328 motd.sym.link  340 passwd.bak
```

和 cp 命令一样，也能将 mv 命令的第二个参数指定为目录，而将第一参数用文件名称清单代替。所有这些文件将移到指定的目录下，保持它们的名称不变。

2.3 处理文件

你经常需要检查文本文件的内容。你已经看到用 cat 命令来这样做。cat 是 concatenate 一字的缩写，它表示连接在一起。可以指定文件名称的清单作为 cat 命令参数。它将这些文件一个接一个地不间断地在屏幕上列出它们的内容：

```
$ cat /etc/passwd /etc/motd
```

问题在于如果这些文件的内容超过一满屏,屏幕更换的速度太快,很难阅读,以致只能看到最后一屏。你要的是一个类似 cat 的命令,但能在内容填满一屏以后暂时停下来。这个常用的命令称为 more:

```
$ more /etc/passwd /etc/motd
```

当 more 命令在每页后面暂停时,可以敲入下列字符中的一个,告诉它下一步做什么:

Space	列出下一页;
Enter	列出下一页;
q	退出;
:n	列出下一个文件。

另一个类似 more,但增加了许多功能的命令是 less。

如果有一个文件,要周期地将新的正文加到文件的最后。在这种情况下,可能只希望看到新增加的内容,而不是每次从头到尾看一遍。特别是,当文件变得越来越长时更是如此。只看文件最后的内容的命令称为 tail,因为它只显示文件的最后部分。tail 的默认值是显示文件最后的 10 行,但这个数值可以用 -n 开关加以改变:

```
$ tail -n 4 /etc/passwd
user:off:100:50::/home/user:/bin/sh
ftp:*:404:1::/home/ftp:/bin/false
pc:fjKppCZxEvouc:500:500::/usr1/pc/bin/bash
carey:Ytla4ffkG2r02:501:500:/usr1/carey:/bin/bash
```

实际上,将上面命令中的 -n 4 缩写为 -4,得到的效果也相同。

另一项有用的功能是对文件的行、字和字符进行计数的能力。这可以用 wc(字计数)命令来完成:

```
$ wc  /etc/passwd
      21   42   775   /etc/passwd
```

上面的三个值 21,42 和 775 分别表示 /etc/passwd 文件的行数,字数和字符数。这里 21 行表示在这台机器上有 21 个登录的帐号。你可能会认为 passwd 文件中不止 42 个字。这和在字典中查字不一样,因为 wc 使用更简单的字定义。它只将用空格,制表符和换行等字符分隔的字符串作为一个字。

同时打印三个数字是 wc 的默认操作。可以使用 -l, -w 和 -c 开关中的一个来分别得到行数、字数和字符数:

```
$ wc -l  /etc/passwd
      21  /etc/passwd
$  -w    /etc/passwd
      42  /etc/passwd
$  -c    /etc/passwd
     775  /etc/passwd
```

使用 wc 命令时也可以不给出文件名。在这种情况下，wc 命令将对从键盘直接送入的正文的行、字和字符数进行计数。

从键盘上输入正文遇到的问题在于当结束正文的输入要看计数结果时，要给 wc 发信号。这可以在新行开始处敲文件结束符(EOF)来实现。在键盘上的 ctrl-d 相当于 EOF。当按 ctrl-d 时，wc 命令将像读文件时遇到 EOF 一样，知道到达了文件的末尾。

2.4 联机帮助

一般讲，每个命令都有比我们看到的更多的命令行开关。如果想知道这些开关的格式和使用的细节以及其他系统功能，可以使用联机手册。手册中有大量的可用信息。Linux 的联机手册分成若干节，每节的用途如下：

section 1	这一节详细说明响应 shell 提示符从键盘输入的命令的细节(如 cat 和 ls)。
section 2	当写程序时(也许用 C 语言)，常常要调用操作系统内核提供的功能，这些功能是通过系统调用(system calls)来实现的。这一节详细说明这些系统调用。
section 3	除了操作系统提供的系统调用外，还有编译程序提供的收集在库中的大量程序语言函数调用，手册的第 3 节说明库函数调用。
section 4	在 Linux 系统中，所有的输入和输出都看作是来自文件或送到文件中去。这表示在写程序从键盘读入字符时，或者将字符写到打印机或屏幕上时，实际打开的是代表这些设备驱动程序的文件，然后通过它们去读，写字符。这些代表机器硬设备的文件是特殊文件。按惯例它们存在 /dev 目录中。通过这些文件驱动硬设备的有关细节在这节中加以说明。
section 5	这一节说明不同的系统数据和管理配置文件的结构和格式(如口令文件)。
section 6	这一节说明系统提供的游戏。
section 7	这一节说明其他各节不说明的杂项。
section 8	这是另一个说明从键盘输入的命令的地方。这些命令只供系统管理用，即只能由系统特权用户 root 使用。
section 9	这一节说明 Linux 内核本身的内部工作细节。

用来访问所有这些手册页的命令称为 man，它的使用方式如下：

```
$ man  ls
```

这将给出 ls 命令的手册页的清单。由于大部分手册页的内容超过一屏的长度，man 命令使用 more 命令（或 less 命令）将手册页分页送给屏幕。

每一手册页分成几个部分，典型地包括下列各部分：

NAME　在这个标题下，给出手册页的条目(例如，ls) 以及它的功能的联机说明(例如，列出目录的内容)。

SYNOPSIS　这里说明如何使用这一条目；所有可供使用的开关选项的清单以及任何必要的和可选的参数清单(例如：ls [- adgilCFR] [mane…])。任何包含在方括弧中的内容是选项，后面跟以 (…) 是可重复项。所以，本例说 ls 命令可以选择 adgilCFR 开关的任意组合或全部，它后面可以跟一个或多个文件名称。这和你已经看到的 ls 命令的用法是一致的。并增加了更多还没有看到过的开关。实际上 ls 命令可以使用的开关比上面列出的还要多。

DESCRIPTION　这部分比 NAME 部分更详细说明命令的功能。

OPTIONS　每个开关的功能在这里进行详细说明。

RETURN VALUE　在说明函数调用的手册页中，在这个标题下说明返回值的类型，也可能包括出现错误时返回的特定值。

FILES　在这一标题下列出有关的配置和数据文件。

SEE ALSO　这部分列出可能要查看的其他有关的手册页条目。

BUGS　这里列出任何已知的缺点，限制甚至已知的隐错。

一般情况下，当你查找特定的手册条目时，从手册的第一节开始顺序进行搜索，并显示第一次出现的条目。有时，特定的条目不止在一节中出现。例如，passwd，它不仅在第一节中作为改变口令的命令出现，也在第 5 节中在讨论 /etc 中 passwd 的文件格式时出现，如果你只送入下面的命令：

```
$ man  passwd
```

你只得到第 1 节中的条目。为了在特定的节中找到手册条目，可以 man 命令中给出可选的参数。下面的命令将列出手册第 5 节中的 passwd 条目：

```
$ man  5  passwd
```

也可以用 -a 开关使 man 命令列出手册各节中任何有关的条目。

```
$ man  -a  passwd
```

2.5 安全

Linux 是一个多用户操作系统，它允许多个用户同时登录和工作。你已经看到系统强制实施的安全措施，要求用户登录和提供口令。这就允许系统(通过用户的 uid)分别来确定每个用户在它的登录对话过程中做了些什么。这种识别也有助于将用户在登录对话过程中建立的文件标记为用户所有。当你是任何文件的所有者时，系统就赋予你对它们实施控制的权利。你可以确定其他用户对这些文件的访问权限。

对文件和目录讲，每个文件和目录都有一组权限标志和它们结合在一起。可以用 ls 命令加 -l 开关来查阅这些标志的设置情况：

```
$ cd
$ ls -l text
total 2
-rw-r--r--  2 pc       book       22 Apr 20 20:37 motd
-rw-r--r--  2 pc       book      796 Apr 20 20:37 passwd
```

这一开关告诉 ls 命令产生指定目录的长清单。这将包括许多其他信息,输出中的第一条信息表示目录中所有文件占用的磁盘的数据块总数(这里是两块,典型地每块 1k 字节)。输出的其余信息每个文件各占一行。每行的第一个字符说明文件的类型。本例中的连字符(-)用来表示这些是普通文件。这个位置也可以出现其他字符,最常用的是字母 d,表示这一行说明一个目录。每行中第 1 组字符的其余部分是文件的权限标志(很快将回头来说明这一点)。权限标志后面的数字说明文件的索引结点有几个目录连接。下一字段给出文件所有者的登录名,接下去是共享该文件访问的用户组组名(有时也用 gid),再后面是文件容量的字节数。接着是文件最后修改的日期和时间,最后则是文件名称本身。

顺便提一下,如果你要找出目录的细节而不是目录中文件的细节,你可以在 ls 命令中加 -d 开关来实现:

```
$ ls -ld text
drwxr-x--x  2 pc       book       1024 Apr 20 20:35 text
```

普通文件的存取权限标志按排的顺序是读文件的内容,写新数据到文件中去和能作为命令执行这一文件。目录的顺序是读包含在目录中的文件名称,写信息到目录中去(增加或删除索引结点的连接)和搜索目录(即能用该目录名称作为路径名去访问它所包含的文件或子目录)。对每个文件(和目录)都有 4 类不同的用户。每类用户各有一组读,写和执行(搜索)文件的访问权限,这 4 类用户是:

root　这是系统特权用户类,他们都有访问 root 登录帐号的权限

owner　这是实际拥有文件的用户。

group　这是共享文件的组访问权的用户类的用户组名称(有时用 gid)。

world　这是不属于上面 3 类的所有其他用户。

作为 root 用户,他们自动拥有所有文件和目录的全面的读,写和搜索权限,所以没有必要明确指定他们的权限。其他三类用户则可以在单个文件或目录的基础上分别授权或撤消权限。因此,对这三类用户,在整个目录层次结构中的每个文件和目录都有一组 9 位的权限位和它们的索引结点结合在一起,分别给文件拥有者,用户组和其他用户指定对文件和目录的读,写和执行(搜索)权。用一些例子就能说得更清楚:

```
-rw-r--r--  2  pc     book     796 Apr 20 20:37 passwd
```

从前面的例子取得的这一行表示 passwd 文件的拥有者是用户 pc,属于用户组 book。本行的第一个字符 (-) 表示这是一个普通文件。后面 3 个字符表示文件拥有者的读、写和执行 (rwx) 权限。本例中字符组 (rw-) 表示对用户 pc 授以读和写的权限,但没有赋予执行文件的

权限。接下来3个字符（r--）是赋予文件的用户组的权限。这里用户组只有读文件的权限而没有写和执行的权限。字符组中最后3个字符（r--）是赋予其他用户的权限，他们同样只有读文件而没有写和执行文件的权限。

```
drwxr-x--x  2  pc  book  1024 Apr 20 20:35 text
```

第二个例子表示text目录的权限标志。行中第一个字符(d)表示这是一个目录。后面3个字符（rwx）表示文件拥有者具有读，写和搜索权。文件的组权限（r-x）允许组内的用户读和搜索目录，但不允许组的成员增加和删除目录中的文件。其他用户的权限位设置（--x）只允许他们有搜索目录的权限，但没有读和写的权限。这阻止了其他用户用ls命令来查目录的内容（没有读权限），但仍允许他们访问目录中他们已知的文件。下面的例子表明其他用户类的一个成员可以看目录中一个可读文件的内容，但不拥有读目录本身的权限：

```
$ cd /home/pc
$ ls -l text
ls: text: Permission denied
$ cat text/motd
motd contents in here.
```

对文件拥有者还赋予其他的特权，不管权限位当前的设置如何，他有权将它们改为其他的值。这可以用chmod命令来完成，chmod命令的一般格式是：

```
chmod  mode  files...
```

其中mode指定新的权限位值，而files则是要改变权限位设置的文件的清单。mode参数可以用两种不同方式指定，可以用符号表示也可以用八进制数的位图表示。八进制数位图可能最简单和易于理解，因它只包含写出3个八进制数。每个八进制数是一个3位的位图，分别用来指定3类用户的rwx权限。

假定要用新值rwxrwxr-x来改变前面例子中text目录的权限值，使得目录拥有者和组内用户对text目录有读、写和搜索权，而其他用户则有读和搜索目录的权限，没有写的权限。第一件事是将所需的权限位字符串(rwxrwxr-x)转换为2进制位图。这可以用将授权位置1而将未授权位置0的办法来得到。本例中将得到111111101 9位二进制字符串，接下来按3位分组就能直接转换为八进制数。这样做得111 111 101对应的八进制数为775。改变text目录的权限值的命令如下：

```
$ chmod 775 text
$ ls -l
drwxr-x--x  2 pc  book  1024 Apr 20 20:35 backup
drwxrwxr-x  2 pc  book  1024 Apr 20 20:35 text
```

注意目录text的权限位设置已按要求作了更改，而backup目录则保持不变。

2.6 进程

程序的可执行的二进制映像保存在磁盘的一个文件中(譬如说,在/bin 或 /usr/bin 目录中)。当运行一个程序时,它的可执行机器代码以及一些初始化数据将从文件复制到内存。在这里,它和 Linux 为它提供的使它能运行的环境结合在一起。存在磁盘上代码和数据称为一个程序。程序一旦装入内存,而且和它的运行环境结合在一起,它就成为一个进程。事实上,存在磁盘上的程序是静态的,而执行中的程序(进程)则是动态的。

系统中的每一个进程用一个称为进程识别号(process identity number,PID)的唯一的整数值来加以识别。

在 Linux 系统中,只有现存的进程一分为二时才能建立新的时程。新进程称为现存进程的子进程,而现存的进程则变为双亲进程。一个单独的双亲进程可以产生许多子进程,而这些子进程又可以产生它们自己的子进程。这表示所有在 Linux 机器上运行的进程从子进程,双亲进程到祖进程等形成一个树形的层次结构。这个进程的树形结构的根是一个称为 init 的进程。它是系统中所有其他进程的共同的祖先。它的 PID 值是 1,是系统启动时运行的第一个实际进程。

可以用一些命令来给出进程的重要统计数据。我们要看的第一个命令是 ps,只送入 ps 命令本身,它就能列出为你运行的一些进程。它的典型的输出如下:

```
$ ps
  PID TTY STAT   TIME COMMAND
  325 v01 S      0:00 -bash
  359 v01 R      0:00 ps
```

这表示为该用户运行了两个进程,第一个进程的 PID 是 325,它是用户登录时的 bash shell 进程。第二个进程的 PID 为 359,正是产生输出的 ps 命令。bash 前面的连字符表示这是系统为用户运行的 shell 进程而不是用户从键盘输入的命令。ps 进程是 shell 响应你的命令而建立的进程。输出结果中的其他 3 列:TTY,STAT 和 TIME 分别表示进程控制的终端,它的系统状态和它的 CPU 使用时间。

加上适当的命令行开关,可以用 ps 命令获得更多的信息。这些信息对新用户暂时还用不上。有几点值得提一下。用 -j 开关将在 PPID 列中显示进程的双亲进程的 PID:

```
$ ps -j
PPID  PID  PGID   SID  TTY  TPGID  STAT  UID  TIME  COMMAND
   1  325  325    325  v01    393  S     500  0:00  -bash
 325  393  393    325  v01    393  R     500  0:00  ps -j
```

注意:用户的 shell 进程的双亲进程是 init 进程,因为它的 PPID 的值为 1。同时注意:shell 是 ps 进程的双亲进程。而 ps 的 PID 和前面的例子中的不一样。这表示即使运行同一个程序,它也是不同的进程。

ps 命令的输出可能变得很长,特别是让它输出系统中所有进程的清单,而不只是自己运

行的进程的清单。这可以加两个命令行开关来实现：-x 开关将列出没有终端控制的进程，而 -a 开关将列出系统中所有其他用户运行的进程以及自己的进程。下面是稍加简化的输出结果：

```
$ ps -ax
  PID  TTY      STAT   TIME  COMMAND
    1  ?        S      0:00  init
    6  ?        S      0:00  bdflush (daemon)
    7  ?        S      0:00  update (bdflush)
   25  ?        S      0:00  /usr/sbin/crond -110
   41  ?        S      0:00  /usr/sbin/syslogd
   43  ?        S      0:00  /usr/sbin/klogd
   45  ?        S      0:00  /usr/sbin/inetd
   47  ?        S      0:00  /usr/sbin/lpd
   61  v03      S      0:00  /sbin/agetty 38400 tty3
   62  v04      S      0:00  /sbin/agetty 38400 tty4
   63  v05      S      0:00  /sbin/agetty 38400 tty5
   64  v06      S      0:00  /sbin/agetty 38400 tty6
  325  v01      S      0:00  -bash
  460  v02      S      0:00  -bash
  547  v02      R      0:00  ps -ax
```

Linux 是一个真正的多用户多任务操作系统。这表示它可以同时接受多个用户登录，每个用户又可以各自运行多个进程。可以有几种不同方式登录到 Linux 系统中来，最常用的方式是通过和机器连接的键盘和屏幕。即使在这种情况下，也可有多个用户登录或者一个用户进行多次登录，这是因为 Linux 支持多个虚终端，任何时候都有一个虚终端和实际的键盘和屏幕连接。事实上，键盘和屏幕可以在一组虚终端之间切换，允许建立多个登录对话过程。只能从主要的系统键盘和屏幕上使用虚终端。虚终端的选择可以通过按下 Alt 键再按一个功能键来实现，通常使用 F1 到 F6。这就可以方便地访问 6 个不同的登录对话过程。当系统第一次启动时，用的是默认的 Alt-F1 终端。虚终端是非常有用。开发软件时，可以用一个终端进行编辑，用一个终端进行编译，再用另一终端查阅信息。

在上面的 ps 命令输出的清单中，注意 TTY 列的号码，它们表示进程是和每个虚终端结合在一起的(V01 到 V06)。PID 为 325 的 shell，是用户在 1 号虚终端上登录的 shell 进程。然而输出清单中的 ps 命令是由 2 号虚终端上的用户执行的。其他 4 个虚终都在运行下列程序的副本：

```
/sbin/agetty
```

正是这个程序给你提供登录提示符并等待送入登录名。从中可以看出这一程序的 4 个副本如何作为独立的进程在运行(它们有不同的 PID)。

一般情况下，当用户退出登录时，在 TTY 列中由用户登录时使用的同一个终端所控制的所有进程也随之结束。然而在 TTY 列中带有问号标记的进程没有和终端结合在一起。这表

示系统中即使没有用户登录,也能继续运行。

如果要很快找到登录时使用的终端,可以使用 TTY 命令:

```
$ tty
/dev/tty1
```

从中可以看出 1 号虚终端的全名是/dev/tty1。类似地,2 号虚终端是/dev/tty2,等等。按惯例,/dev 目录是系统用来保存特殊文件的地方。虚终端就是使用特殊文件的一个例子。

有时,需要强制结束一个进程。只要这是你控制的进程,通常因为执行了它,就可以用 kill 命令来结束它的运行。要做的只是将要结束的进程的 PID 作为 kill 命令的参数。kill 命令将向这一进程发出结束运行的信号。可能出现进程忽视 kill 命令发出的信号的情况。加 -9 开关它就能发出不能忽视的信号。下面的命令序列用来说明这些观点:

```
$ tty
/dev/tty2
$ ps
  PID  TTY  STAT  TIME  COMMAND
  325  v01   S    0:00  -bash
  460  v02   S    0:00  -bash
  557  v02   R    0:00  ps
$ kill 325
$ ps
  PID  TTY  STAT  TIME  COMMAND
  325  v01   S    0:00  -bash
  460  v02   S    0:00  -bash
  559  v02   R    0:00  ps
$ kill -9 325
$ ps
  PID  TTY  STAT  TIME  COMMAND
  460  v02   S    0:00  -bash
  561  v02   R    0:00  ps
```

首先,tty 命令的输出结果表示用户当前已切换到 2 号虚终端,接着 ps 命令给出 1 号虚终端上的 shell 的 PID 是 325。对这一 PID 发出 kill 命令,第二个 ps 命令显示,shell 可以忽视结束信号。然而第二个带 -9 开关的 kill 命令结束了这一 shell 的运行。这一点可以从最后一个 ps 命的输出结果中看到。

2.7 空间

除进程外,监示一些系统资源的使用情况是有用的。最常用的是磁盘空间和内存空间。为了按文件系统来检查可用的和已用去的磁盘空间,可以用 df 命令:

```
$ df
Filesystem    1024-blocks    Used  Available  Capacity  Mounted on
/dev/hda3       199270      182354     6625      96%     /
/dev/hda1       610608      209675   369394      36%     /usr
/dev/hda4       199271      147953    41027      78%     /home/pc
```

你的系统的分区的数目和容量和例子中给出的不同,但原理是一样的。df 命令的输出清单的第 1 列是代表这一磁盘分区的特殊文件的路径名;第 2 列给出分区包含的 1024 字节的数据块的数目;第 3,4 列分别表示用去的和可用的数据块数目。会感到奇怪的是,第 3,4 列块数之和不等于第 2 列中的块数。这是因为缺省地每个分区都留了少量空间供系统管理员使用。即使遇到普通用户空间已满的情况,管理员仍能登录和留有解决问题所需的工作空间。清单中 Capacity 列表示普通用户空间使用的百分比,即使这一数字达到 100%,分区仍然留有系统管理员使用的空间。最后,Mounted on 列表示在总的目录树结构(图 2.1) 中装配每一分区的根目录所使用的目录。

要找出特定分区用掉了多少空间,还留有多少空间可供使用时,df 命令是有用的。然而,它不能找出起始目录用了多少空间,它的子目录和文件用了多少空间,要找出这一类磁盘利用信息,可以用 du 命令:

```
$ cd /etc
$ du
24      ./rc.d
1       ./fs
4       ./skel/.term
8       ./skel
1       ./lilo
1       ./default
5       ./msgs
389     .
```

输出清单中的第一列是以 K 字节计的磁盘空间容量,第二列列出目录中使用这些空间的所有文件名称。

注意不带开关的 du 命令将从当前目录开始沿着目录结构向下工作直到列出所有目录的容量为止。这可能是一个很长的清单,有时只需要一个总数。这时可在 du 命令中加 -s 开关来取得总数:

```
$ du -s /etc
389     /etc
```

从上例中可以看出,作为搜索起点的目录名称可以作为 du 命令的参数使用。事实上,du 命令可以用目录名称清单作为它的参数。在它的输出结果中分别列出这些目录的容量。

现在我们来看内存空间的使用情况。机器上装了固定数量的随机访问存储器(RAM)。这

一内存用来存储运行的操作系统内核，所有在机器上正在执行的进程，以及这些进程操作的所有数据。事实上，Linux 能安排比内存能容纳的更多的进程同时运行。因为它在磁盘上留了内存溢出区，传统上称为交换空间(swap space)。一般说来，在开始时内存都没有被使用(即完全是空的)。为了加速磁盘的存取速度，Linux 作这样的安排，一旦从磁盘文件中读入一块数据，它就将它放在内存的缓冲区中，而且尽可能长时间保留在内存中。这样，当再次用到这块数据时，它可以从内存缓冲区中取数据而不必重读磁盘。从内存中读取磁盘数据要比直接从磁盘读数据快几个数量级。在写数据到磁盘的过程中也用同样的缓冲技术。

内存的另一种用途是将程序装入内存作为进程来执行。如果系统中进程的数目不断增加，那么过一段时间以后，留下的空闲内存就不足以运行任何新的程序。这时候 Linux 开始从磁盘缓冲区中取回一部分内存供运行更多的进程用。用这种方式系统中进程数目继续增加，用于磁盘缓冲区的内存空间不断减少，直到达到最少必要的磁盘缓冲区空间为止。到现在为止，已经没有空闲的内存空间，也不可能再从缓冲区腾出空间。所以，Linux 要采取新的策略。它的做法是在内存中找出已经有一段时间不用的内存块，将它写到交换空间中去。使得交换出去的内存块变为空闲，可被再次使用。当系统中进程的负载减轻时，被交换出去的内存块可以取回内存。最后，在进程结束时重新加到空闲内存清单中。

为了最大限度发挥系统内存的作用，Linux 还使用另一项技术，共享可执行的代码。如果两个不同的进程执行的是同一个程序或同一程序库，就没有理由将相同的程序在内存的不同位置装入两次。完全可以共享程序在内存中的单一副本。Linux 正是这样做的。

当操作系统针对固定的内存资源平衡各种不同的需求时，内存的数据块在空闲内存清单，磁盘高速缓存，进程内存和交换空间之间来回调度。整个过程是动态进行的。所有有关这些内存区的信息可以用 free 命令来显示：

```
$ free
             total       used       free     shared    buffers
Mem:         19208      13444       5764       7448       7784
-/+ buffers            5660      13548
Swap:        32992          0      32992
```

所有上面的数字都是以 1K 字节为单位。看输出的最上面一行，在 total 列下面的数字是启动系统时 Linux 拥有的可用的空闲内存总数。如果你知道机器上实际安装的内有总数，就会发现列出的数据比实际少 1MB 左右。这是因为这部分内存已移去用于系统内核，没有在这里显示。后面两列分别表示已经使的和尚未使用的内存块数，这两项相加等于总数，再后面是所有进程共享的内存块数。最后则是现在用作磁盘缓冲区的内存块数。输出的第二行中包括两个数字，这是上一行对应的数字减，加缓冲区数据块数得到的数字。这是只从进程的角度看内存的使用情况，因为缓冲区空间可以重新作为进程空间使用。输出的最后一行表示磁盘上交换空间的总块数，已使用的块数和未使用的块数。

监视内存使用方式如何随时间变化对管理具有指导意义。这可以用 top 命令来完成。它在屏幕上提供大量的系统统计信息。这些信息每隔几秒就改变一次。top 命令实际是 ps 命和 free 命令的结合，还增加了一些新内容。进程的显示按使用 CPU 时间长短排序。使用时间最长的排在前面，简化的 top 命令输出结果如下：

```
$ top
1:44pm up 8 days,2:21, 4 users, load average: 0.11, 0.15, 0.14
33 processes: 32 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  2.1% user,  0.0% nice,  7.4% system, 90.5% idle
Mem: 15040K av, 14340K used, 700K free, 6312K shrd, 6620K buff
Swap: 9208K av, 548K used, 8660K free

PID USER    PRI  NI SIZE  RES SHRD STAT %CPU %MEM   TIME COMMAND
471 pc       20   0  100  308  332 R     9.0  2.0   2:24 top
273 root      1   0   85  280  352 S     0.5  1.8   0:17 in.tftpd
  1 root      1   0   48  196  280 S     0.0  1.3 26:37 init
295 cew       1   0  536  656  508 S     0.0  4.3   0:05 -bash
849 mot       1   0  538  652  504 S     0.0  4.3   0:03 -bash
```

输出的第一行显示当前的时间;机器启动以来经历的时间长度;当前有多少用户登录。最后三个数字表示最近的系统平均负荷。三个系统负荷值分别表示前 1 分钟的平均负荷,前 5 分钟的和前 15 分钟的平均负荷。平均负荷相当于全速运行所有进程时,机器需要的处理机数目。值越小表示系统负荷越轻。很明显,如果所有进程仅共享一个 CPU,平均负荷大于 1 就表示这些进程没有充分得到所需要的 CPU 时间,因此它们都慢了下来。

输出的第二行说明系统内总的有多少个进程,它们分别处于什么状态。

输出的第三行给出 CPU 的利用信息。表示当前有 90%以上的时间处于空闲状态。用于运行用户和系统程序的时间少于 10%。这些数字进一步证实了前面给出的平均负荷。

下面两行包含的信息和 free 命令输出结果相同。最后是 PS 命令的输出,但是按使用 CPU 时间的百分比进行了排序。

练习

1. 完成简单的实验,找出一个符号连接是否可以指向另一个符号连接,然后指向一文件。如果可以这样做,那么,在符号连接链中符号连接的数目是否有限制?
2. 符号连接和硬连接相比有什么缺点?
3. 你能否找出一对目录连接,两者都指向同一个文件?
4. 执行下面的命令序列:

```
$ cd
$ mkdir  temp
$ cp  /etc/passwd  temp
$ chmod  400  temp
```

   如果你现在执行 ls temp 命令,你认为会得到什么结果?
5. 接着上一个问题。如果你执行 ls -l temp 命令,你认为会得到什么结果?试一下,是否得到预期的结果。如果没有,为什么?

答案

1. 可以简单地完成实验如下:

```
$ cp /etc/passwd ./p0
$ ln -s p0 p1
$ ln -s p1 p2
$ cat p2
root:1tq3uhhA2dnSM:0:0:root:/root:/bin/bash
...
```

第一条命令在当前目录下建立 passwd 文件的工作副本 p0,第二条命令对工作副本建立符号连接,将符号连接称为 p1。第三条命令对 P1 建立第二个符号连接,称为 p2。最后用 cat 命令来检验是否能用第 2 个符号连接(p2)来访问链的末端的目标文件。实际上,这是可以的。

如果你继续在链中增加新的符号连接,看上去是可行的,但当你在链中超过 6 个连接时,你就不能用来访问目标文件:

```
$ ls -l
total 1
-rw-r--r--  1 pc    500     73 1 Jan 13 16:19 p0
lrwxrwxrwx  1 pc    500      6 Jan 13 18:07 p1 -> p0
lrwxrwxrwx  1 pc    500      2 Jan 13 18:07 p2 -> p1
lrwxrwxrwx  1 pc    500      2 Jan 13 18:08 p3 -> p2
lrwxrwxrwx  1 pc    500      2 Jan 13 18:08 p4 -> p3
lrwxrwxrwx  1 pc    500      2 Jan 13 18:08 p5 -> p4
lrwxrwxrwx  1 pc    500      2 Jan 13 18:08 p6 -> p5
lrwxrwxrwx  1 pc    500      2 Jan 13 18:09 p7 -> p6
$ cat p7
cat: p7: Too many symbolic links encountered
$ cat p6
root:1tq3uhhA2dnSM:0:0:root:/root:/bin/bash
...
```

所以你要成功地进行符号连接,链的长度不能超过 6 个连接。

2. 为了存取文件的内容,需要取得它的索引结点号。记住符号连接的唯一实际缺点是每增加一个符号连接,在寻找目标文件的索引结点号的过程中就增加一次间接访问,因此减慢了寻找过程。硬连接不存在这个问题,因为每个硬连接直接和索引结点号结合在一起。符号链接还有一个要注意的缺点,当文件不存时连接却依然存在。试验一下文件不存在时,使用符号连接会得到什么结果。
3. 解决这个问题的最明显的办法是使用加 -l 开关的 ls 命令来找出有两个或两个以上连接的文件(在 ls 命令输出结果的第 2 列中)。然后用 -i 开关来显示有关文件的索引结点号,看是否能找到配对的目录连接。

此外,目录也是文件。而且每个目录至少有两个对它的索引结点的连接,即使它是空的。例如:/bin 目录在根目录中有一个连接(称为 bin)。但不要忘记在 /bin 目录内还有另一个连接(称为..)。在 ls 命令中加 -d 开关就能验证这一点:

```
$ cd /
$ ls -id bin
  43478 bin
```

```
$ cd /bin
$ ls -id
  43478
```

在机器上做试验时，/bin 目录的索引结点号的数值可能和上例中的不同，但重要的是两个索引结点号是相同的。

4. 由于建立了 temp 目录，作为目录的拥有者有读目录的权限，所以运行的 ls 命令能够读目录中的文件名称清单，也能像预期一样列出目录的内容。做这件事不需要增加其他权限。

```
$ ls temp
passwd
```

5. 现在执行带 -1 开关的 ls 命令，得到的输出结果如下：

```
$ ls -1 temp
ls: temp/passwd: Permission denied
total 0
```

这是说 ls 命令在 temp 目录中发现了 passwd 文件(从上例可知它能做到这一点)，但是，它以后要完成的操作被拒绝了。实际情况在于加了 -1 开关以后，ls 命令不能从目录中得到它要显示的信息。它要去访问文件本身来取得其他信息。然而你没有对目录的搜索权，ls 命令就不能将目录用作文件的路径名的一个组成成分。结果是当 ls 命令试图找出它所要的其他信息时，即使它知道文件就在那里，它也从目录中得到了文件名称，但它被拒绝访问文件。为了 ls 命令能按预期方式执行，你需要改变 temp 目录的访问权限如下：

```
$ chmod 500 temp
$ ls -1 temp
total 1
-rw-r--r--   1 pc     500        731 Jan 13 16:19 passwd
```

第三章　正文编辑

如果希望在 Linux 机器上做更多的事情，不仅仅停留在运用一些预先安装的软件包，还应该学习如何运用系统所提供的正文编辑程序。Linux 系统中有许多正文编辑程序可供选择。如果已经掌握其中的一种，你也许会继续使用下去。如果还没有掌握任何一种编辑程序，或者愿意将来转到 UNIX 类环境中工作，应该利用机会掌握标准的 UNIX 屏幕编辑程序 vi。虽然真正的 vi 编辑程序是专用产品，不可以直接用在 Linux 上。但是有许多（至少 5 到 6 种）vi 兼容的编辑程序，可以自由地用在许多系统上，包括 Linux 系统在内。只要掌握标准的基本命令组，用任何一个编辑程序都一样容易。从现在起，我们将用 vi 来代表任何 vi 兼容的编辑程序（包括 elvis，vim，stevie，nvi 等）。

vi 是屏幕编辑程序。这表示屏幕的内容是被编辑的文件的一个窗口。vi 对文件所做的修改是在文件的副本上进行的，并不直接对源文件本身进行修改。如果在编辑过程中发生了错误，可以将修改的结果全部放弃，重新回到原始的文件。只当认为一切进行顺利，并发出保存修改结果的命令时，才用修改了的文本取代原始文件。

为了进入 vi 编辑程序，可以用下面的命令：

```
$ vi  filename
```

其中 filename 是要编辑的文件的路径名。如果文件不存在，它将为你建立一个新文件。vi 编辑程序有三种操作方式，分别称为编辑方式，插入方式和命令方式（见图 3.1），图中表明，当运行 vi 时，首先进入编辑方式。为了实验的目的，建议从 /etc 目录中取一个系统文本文件的副本（如 /etc/passwd 或 /etc/printcap），并用这个副本实践你的编辑命令。

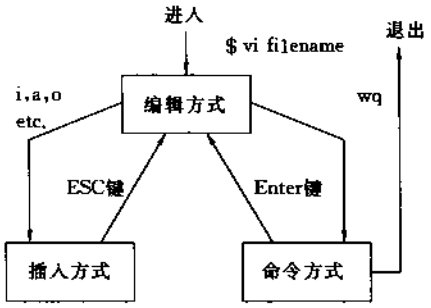

图 3.1　vi 的三种主要的操作方式

3.1　编辑方式

vi 编辑方式的主要用途是在被编辑的文件中移动光标的位置。一旦光标移到所要的位置，就可以进行剪切和粘贴正文块，删除正文和插入新的正文。然后再移动光标到另一个位置等等。vi 不要求键盘有特殊的键（如光标控制键）。在编辑方式下，许多字母键和它们的组合具有特定的功能。例如，当完成所有的编辑工作后，需要保存编辑结果，退出编辑程序和回到 shell 提示符，可以发出 ZZ 命令。它表示连按两次 Z 键。注意：编辑程序区分大小写字母，所以，你必须确实按了 ZZ 而不是 zz。

3.1.1　光标定位

如果键盘上有上，下，左，右箭头键，就由这些键来完成光标的移动。如果键盘不支持这些

键,可以用下面的键来完成同样的功能:

k 上移;
j 下移;
h 左移;
l 右移。

vi 设计成可以在几乎所有的终端上运行。一些系统的终端类型,特别是旧终端,就是没有箭头键和其他移动光标的特殊键,只是一个标准的打字机键盘。

这 4 个键将光标位置每次移动一行或一个字符。vi 还提供大范围移动光标的命令:

ctrl-f 在文件中前移一页 (相当于 page down);
ctrl-b 在文件中后移一页 (相当于 page up)。

这些按页移动的命令在读整个文件的过程中提供一定程度的连续性。

在屏幕上找到需要的一页时,可以用下面的命令快速移动光标:

H 将光标移到屏幕上的起始行(或最上行);
M 将光标移到屏幕中间;
L 将光标移到屏幕最后一行。

同样要注意字母的大小写。H 和 L 命令前面还可以加数字,表示移进屏幕的行数。如 2H 将光标移到屏幕的第 2 行,3L 将光标移到倒数第 3 行。

当将光标移到所要的行时,行中的正确位置可以用下面的命令来实现:

w 右移光标到下一个字的开头;
e 右移光标到一个字的末尾;
b 左移光标到前一个字的开头;
0 (零)左移光标到本行的开始;
^ 移动光标到行中第一个非空字符;
$ 右移光标到本行末尾。

当 w,e 或 b 命令到达行的末尾(或开头),将自动进到下一行(或前一行)

3.1.2 搜索字符串

如果需要在文件中找已知的特定的字或短语出现的位置,可以用 vi 直接进行搜索,不必用手工的办法在文件中移动光标去找。为了实现搜索,先送入斜杠字符 (/) 后面跟以要搜索的字符串,再按 Enter 或 Return 键。编辑程序将向前朝文件尾方向搜索特定的字符串。如果找到了这一字符串,编辑程序停止搜索并将光标移到字符串的开头。如果这不是要找的位置,可以用 n 命令继续搜索来找出字符串下次出现的位置。还可以用问号 ? 来取代斜杠字符 / 向后朝文件头方向进行搜索。不管向那一个方向进行搜索,当到达文件末尾或开头时,它将绕

到文件另一端,从那里继续进行搜索。总的讲:

/string　向前搜索给定的字符串 string;
? string　向后搜索给定的字符串 string;
n　　向前或向后搜索,找出字符串下次出现的位置。

3.1.3 替换和删除

当光标定位于文件中所要的位置时,可以有多种选择。在编辑方式下,可以用其他字符替换光标所指的字符,或者从当前的光标位置删除一个或多个字符。主要选项有:

rc　用 c 替换当前光标指示的字符;
x　删除当前光标位置的字符;
dw　删除光标右面的字;
db　删除光标前面的字;
dd　删除光标所在的行,并去掉空隙。

在上面的任何命令前面加上数字,它们的功能扩充如下:

*n*rc 从光标位置开始用 c 替换 n 个字符;
*n*x　从光标位置开始删除 n 个字符;
*n*dw 在光标右面删除 n 个字;
*n*db 在光标前面删除 n 个字;
*n*dd 删除 n 行,并去掉空隙。

其他常用的删除命令(前面不能加数字)是:

d$　从当前光标起删除字符直到行的结束;
d0　从当前光标起删除字符直到行的开始;
J　删除本行的回车字符(CR),并和下一行合并。

3.1.4 剪切和粘贴

想从正文中删除字符,字或行时,它们并未被抛弃。而是将它们复制到一个内存缓冲区中。这可能很有用,因为有一对命令允许将该缓冲区中的内容粘贴到正文中,或者粘贴在同一个位置(如果错误地删除了),或者粘贴到新的位置,完成标准的剪切和粘贴功能。这个命令是:

p(小写)　将缓冲区的内容粘贴到当前光标的后面;
P(大写)　将缓冲区的内容粘贴到当前光标的前面。

如果剪切缓冲区的内容是字符或字,粘贴操作发生在同一行的光标前或后,如果剪切缓冲区包含整行正文,粘贴操作发生在光标所在行的上一行或下一行。

vi 中有若干命令对(类似 p 和 P)。其中一对大小写字母提供类似的功能。在这种情况下,小写命令在光标后面进行操作,大写命令在光标前面进行操作。

有时需要复制一段正文到一个新的位置,而不是用剪切和粘贴移动块。要做的只是将这段正文复制到剪切缓冲区中而不删除它。这时可以用:

yy 将当前行复制到剪切缓冲区;
nyy 将 n 行复制到剪切缓冲区。

3.1.5 撤销和重复

顺便讨论编辑方式下的最后两条命令。当出错和需要撤销某条命令造成的不良后果时,可用第一条命令。当执行某条命令后,将光标移到下一个位置希望重复这条命令时,可以用第 2 条命令:

u 撤销前一条命令的结果;
. 重复最后一条修改正文的命令。

3.2 插入方式

在编辑方式下用了许多普通字母作为命令,当实际需要将新正文插到文件中去时,必须将编辑程序从编辑方式切换到插入方式。使它将这些字符作为普通字符对待,以便将它们加到正文中去。插入正文的工作结束后,又要能够回到编辑方式。这就是插入方式的用途。你将看到由于不同的目的有许多方法进入插入方式。不管用什么方法进入插入方式,退出插入方式的办法只有一个,用 ESC 键。有些键盘没有 ESC 键,在这种情况下,可以用 ctrl-[来完成同样的功能。

一旦光标正确定位后,最常用的进入插入方式的命令是:

i 在光标左面插入正文;
a 在光标右面插入正文。

看起来有些奇怪,为什么要两个命令。但是如果少了其中的一个,在一些情况下插入会有困难。经常需要在一新行的开头插入正文,这种情况下,将光标称到上一行的末尾,然后用 a 命令后面跟以 Enter 或 Return 键就增加一个空行,可以在空行的开头插入正文。更简单的做法是将光定位在所要位置的上一行中任何位置,然后用 o 命令在光标所在行的下面增加一新行,并在新行开头进入插入方式准备接受正文。O 命令完成同样的功能,但在当前行上面增加一空行。

o 在光标所在行下面增加新行;
O 在光标所在行上面增加新行。

还可以使用特殊形式的插入和追加命令将光标移到当前行的开头或末尾,并进入插入方

式：

I　在光标行的开头插入；
A　在光标行的末尾插入；

3.2.1　正文替换

除了单纯进入插入方式的命令外，还有一些命令允许进行正文替换。这等同于在进入插入方式接受替换正文之前先删去一段正文。用于替换的主要命令有：

s　用新正文替换光标所指的字符；
cw　用新正文替换光标右边的字；
cb　用新正文替换光标前面的字；
cd　用新正文替换当前光标行。

在这组命令的前面加上数字，它们的功能进一步扩充：

*n*s　用新正文替换光标后面的 n 个字符；
*n*cw替换光标后面的 n 个字；
*n*cb替换光标前面的 n 个字；
*n*cd用新正文替换下面的 n 行。

除前面的命令外，还有两个替换命令可以使用：

c$　用新正文替换从光标开始到本行末尾的所有字符；
c0　用新正文替换从光标开始到本行开头的所有字符。

3.3　命令方式

到目前为止用到的命令都是简单的。至少从输入命令的击键次数看是这样。还有许多结构更为复杂的命令，而且还可以设置许多配置选项。这些都是命令方式范围内的事。命令方式下的所有命令有一个共同的特征，它们都用冒号(：)开头。一旦送入了冒号，光标就跳到屏幕最后一行，在那里显示冒号。现在送入的到 Enter 或 Return 键为止的所有字符都显示在屏幕的最后一行。你可进行检验，必要时可以改正输入。

3.3.1　退出命令

前面看到的退出编辑程序的唯一方法是在编辑方式下使用 ZZ 命令。这不仅退出编辑程序，同时还将对正文所作的任何修改都保存到原始文件中去，覆盖了它原先的内容。有时候只需要退出编辑程序，不要求保存编辑的内容。这可能由于对正文未作修改，或者虽然作了修改但现在想放弃所作的修改。在这些情况下，可以用下面的命令退出编辑程序：

:q　在未作修改的情况下退出；
:q!　将任何修改作废，退出编辑程序。

尽可能养成用:q 命令的习惯是好的做法,这样可以减少偶尔送入 :q! 命令的机会,后者可能导致一些重要材料的丢失!

3.3.2 文件的使用

下面讨论的一组带冒号的命令允许从文件中读取正文和写正文到文件中去:

:w	将编辑的内容写到原始文件中去;
:wq	写文件和退出编辑程序(相当于 ZZ);
:w *file*	将编辑的内容写到名称为 *file* 的文件中去;
:r *file*	将 *file* 文件的内容读入,放在当前光标行的后面;
:e *file*	编辑新文件 *file* 代替旧的内容;
:f *file*	将当前正文的名称改为 *file*;
:f	打印当前正文的名称和状态。

:w 命令将编辑的正文写回到进入 vi 时指定的文件中去。这一命令不退出编辑程序,在写文件以后继续留在编辑方式下。应该经常用这一命令来保存中间结果。

:wq 命令相当于 :w 命令和 :q 命令的结合。

有时,要修改一个文件的内容,将修改的结果写到新文件中去,保留原有的文件内容不变。这时可以用 :w file 命令。这条命令还有另外一种形式,可以在它的前面加上用逗号分开的行的范围参数。这时它将相应的行写到指定的文件 file 中。

:a,*bw* *file* 将 a 行到 b 行的内容写到 *file* 中。

:r file 命令用来读入指定文件 file 的内容,并将它插在当前光标行的后面。这条命令有助于将原先分段准备的材料结合在一起。

当完成了当前的正文的编辑工作后,如果想接着编辑另一个文件,可以使用 :e file 命令。它的优点是:不用退出编辑程序再重新进入。在编辑文件的过程中,可以改变正在编辑的文件的名称。这可用 :f file 命令来实现。如果在这条命令中不给出文件的名称,vi 将列出当前的正文的名称以及一些状态信息。如文件的行数,光标所在行的行号等。

3.3.3 行号

正在进行编辑的正文的所有各行都有一个行号与它们结合在一起。可以用下面的命令将光标移到指定的行:

:*n* 将光标移到第 *n* 行。

其中 n 是你要的行号。

可以要求 vi 在显示正文时加上行号。这一点将在 3.3.8 节中说明。

必要时,冒号命令前可以规定命令操作的行号范围。有几种给出行号的办法。可以指定数值,它用作绝对行号;可以用句点给出当前光标所在行的行号;可以用美元符号给出最后一

行的行号;还可以用简单数值表达式给出行号,如 .+5 表示当前行前面的第 5 行。下面是一些具体例子:

:345	将光标移到第 345 行;
:345w *myfile*	将第 345 行写到 myfile 文件中;
:3,8w *myfile*	将第 3 行到第 8 行写到 myfile 文件中;
:1,.w *myfile*	将第 1 行到当前行写到 myfile 文件中;
:.,$w *myfile*	将当前行到最后一行写到 myfile 文件中;
:.,.+4w *mgfile*	从当前行开始将 5 行的内容写到 myfile 文件中;
:1,$w *myfile*	写整个文件(相当于:w myfile)。

3.3.4 字符串搜索

除了用数字指定行号外,还可以给出要搜索的正文字符串来指定要的行号。如果希望从当前行向前搜索正文,将要搜索的正文放在两个斜杠字符 (/) 之间。向后搜索正文则将要搜索的正文放在两个问号 (?) 之间。

:/str/	将光标前移到下一个包含 str 字符串的行;
:?str?	将光标后移到最近的包含 str 字符串的行;
:/str/w *myfile*	将第一个包含 str 字符串的行写到 myfile 文件中;
:/str1/,/str2/w *myfile*	将从包含 str1 的行到包含 str2 的行的正文写到 myfile 文件中。

3.3.5 规则表达式

当给 vi 指定搜索字符串时,可以包含具有特殊含义的字符。包含这些特殊字符的搜索字符串称为规则表达式(regular expressions)。

例如,要搜索一行正文,这行正文的开头包含 struct 字。下面的命令做不到这一点:

```
:/struct/
```

因为它只找出在行中任意位置包含 struct 的第一行,并不一定在行的开始包含 struct。解决问题的办法是在搜索字符串前面加上特殊字符 ^ :

```
:/^struct/
```

^ 字符比较每行开头的字符串。所以上面的命令表示:找出以字符串 struct 开头的行。

也可以用类似办法在搜索字符串后面加上表示行的末尾的特殊字符 $ 来找出位于行末尾的字:

```
:/struct $/
```

下表给出大多数特殊字符和它们的含义:

^	放在字符串前面,匹配行首的字;
$	放在字符串后面,匹配行尾的字;
\ <	匹配一个字的字头;
\ >	匹配一个字的字尾;
.	匹配任何单个正文字符;
[*str*]	匹配 *str* 中的任何单个字符;
[^*str*]	匹配任何不在 *str* 中的单个字符;
[a - b]	匹配 a 到 b 之间的任一字符;
*	匹配前一个字符的 0 次或多次出现;
\	不管后面的字符的特殊含义。

3.3.6 正文替换

:s 命令用另一串字符替换行中的一串字符。这条命令有几种变形。可以用新字符串取代行中特定字符串的首次出现,或取代一组行中或整个文件中的所有出现:

:s/*str1*/*str2*/	用 *str2* 替换行中首次出现的 *str1*;
:s/*str1*/*str2*/g	用 *str2* 替换行中每一个 *str1*;
:., $ s/*str1*/*str2*/g	用 *str2* 替换当前行到文件末尾的所有的 *str1*;
:1, $ s/*str1*/*str2*/g	用 *str2* 替换整个文件中的 *str1*;
:g/*str1*/s//*str2*/g	另一种用 *str2* 替换文件中每一个 *str1* 的方法。

在这些替换命令中,g 放在命令的末尾表示在当前光标行中对搜索字符串的每次出现重复使用本命令。不加 g,命令只对行中搜索字符串的首次出现进行替换操作。g 放在命令的开头规定命令对文件中所有包含搜索字符串的行进行替换操作。

3.3.7 删除正文

在编辑方式下可以删除正文,但仍然有删除行的冒号命令:

:d	删除当前光标行;
:3d	在正文中删除 3 行;
:., $ d	从当前光标行到文件末尾全部删除;
:/*str1*/,/*str2*/d	删除从 str1 到 str2 的所有行。

3.3.8 编辑程序的选项

在 vi 中,有许多控制不同编辑功能使用情况的内部变量。可以用:set 命令来设置这些变量的值。

:set *option*	设置 option 变量的不同值;

其中 option 指定变量和设置的值。一些有用的选项是:

autoindent	自动缩进。如果设置这一项,当产生新行时,光标自动移到与前一行第一个非空字符对齐(对 O 命令讲,与下一行第一个非空字符对齐)。例如,正文中包含计算机程序时,这使得对齐变得容易。如果在按 ESC 键退出插入方式之前或者在按 Enter 或 Return 进入另一新行之前,在新行中没有输入任何正文,那么加在那一行中缩进字符将被去掉。使得空行不留缩进字符。加了自动缩进功能后,可能从某一行开始又不用这一功能。这时,可以用 noautoindent 选项来关闭它。
ignorecase	设置这一选项时,将不区分规则表达式中的任何大小写字母。可以用 noignorecase 选项来关闭本选项。
number	行号。当设置此选项时,vi 在屏幕显示的每一行前面加上它的当前行号。这些行号只出现在显示中,并不加到文件中去。number 选项可以用 nonumber 选项来关闭。
ruler	标尺。在屏幕的底行显示当前光标行的行号及光标在行中的位置。这有助于在编辑过程中跟踪正文中的位置。可以用 noruler 来关闭此选项。
tabstop	本选项允许设置按一次 Tab 键跳过的空格数。它的格式是 :set tabstop=n。其中 n 是以空格数计算的宽度。此选项的默认值是 8。没有特殊命令来关闭这一选项。只能将它重新设为默认值。

3.3.9 shell 切换

当处于编辑的对话过程中时,可能需要执行一些 Linux 命令。如果需要保存当前的结果,退出编辑程序,再执行所需的 Linux 命令,然后再回头继续编辑过程,就显得十分累赘。如果能在编辑的环境中运行 Linux 命令就要省事得多。在 vi 中,可以用下面的命令来做到这一点:

:! *command*　　执行 *command* 命令后回到编辑程序。

这称为 shell 切换。它允许执行任何可以在标准的 Shell 提示符下执行的命令。当这条命令执行完毕,控制返回给编辑程序。又可以继续编辑对话过程。

练习

确定完成下列操作的 vi 冒号命令:

1. 找出文件中当前光标行的行号。
2. 删除当前光标行及后面的 3 行。
3. 从当前光标位置向后搜索字符串 8.7 的最近的一次出现。
4. 删除文件的最后 10 行。
5. 从当前光标行向前搜索找出文件中下一个包含 3 位或 3 位以上的数字的行。
6. 在当前行中删除从 aa 到 zz 的所有字符。
7. 在整个文件用 and 代替所有的 & 字符。

答案

1. f

   此命令除给出当前行的行号外,还给出文件的名称和它当前的长度。

2. .,.+3d

   . 表示当前光标行。所以,.,.+3 给出从当前光标行到光标行前面的 3 行的范围。后面的 d 表示删除这个范围。

3. ? 8 \ .7

问号表示从当前行向后搜索。点前的 \ 是必要的,因为点是一个通配符。

4. $ -9, $ d

$ 字符表示文件的末尾

5. /[0 - 9][0 - 9][0 - 9]

向前搜索找出三个连续的在 0 ~ 9 范围的字符。

6. s/aa. * zz//

搜索从 aa 开始,后面跟以任何数目的单个字符(. *),并以 zz 结束的字符串。用空串代替搜索的字符串,即删除前面的字符串。

7. g/&/s//and/g

命令的中间部分表示用 and 取代 &。前面的 g 表示文件中的每一行,后面的 g 表示行中的每次出现。

第四章　Bash

Linux 和 UNIX 类操作系统一样，为用户提供了不同的 shell 程序。所有这些 shell 程序大致都有相同的目标：在用户和系统之间提供有用的接口；同时提供简单的编程接口，允许将不同的命令结合在一起，像执行普通的 Linux 系统命令一样执行组合命令。

本章将集中讨论 bash（Bourne Again SHell）shell 的交互接口。

Bash 由自由软件基金会作为 Bourne shell 的兼容程序发布。它还容纳了其他 shell 程序的许多好的特征，它是功能全面的 shell 程序。许多 Linux 发布都提供 Bash shell。

4.1　路径名扩展

在前面的章节中，所有 Linux 命令中的路径名都是用全称表示的。路径名可以用不同方式缩写，在将路径名传送给命令之前由 shell 加以扩展。包含缩写字符的路径名称为规则表达式（regular expressions）。例如：

```
$ cp  text/*   backup
```

其中星号（*）用来表示任何文件名（以点开头的文件名称除外，因为大多数命令对它们作特殊处理。见 2.1 节）。因此，这条命令说：将 text 目录下的所有文件按原有的名称全部复制到 backap 目录中。cp 命令实际不需要知道 * 字符的任何情况，因为 shell 在 cp 命令执行前已完成路径名的扩展。这表示当 cp 命令读取传送给它的命令行参数时，看到的将是得下面那样的命令：

```
$ cp  text/motd  text/passwd  backup
```

因为 shell 已查了 text 目录，找到了目录中的所有文件名，并将它们在命令行中一一列出再传送给 cp 命令。cp 命令能同时复制多个文件到一个目录中去。因此，它执行 shell 扩展的命令行不存在任何问题。

另一个用作文件名称，并由 shell 扩展的特殊字符是问号（?）。这个字符用来代表任何单个字符。下面的命令和输出结果说明这一点：

```
$ ls /dev/tty?
/dev/tty0  /dev/tty2  /dev/tty4  /dev/tty6  /dev/tty8
/dev/tty1  /dev/tty3  /dev/tty5  /dev/tty7  /dev/tty9
```

这里列出的 tty0 到 tty9 10 个文件是 /dev 目录下与 tty? 模式匹配的所有文件（即 tty 后面跟以任何单个字符）。

除了允许匹配任何单个字符外，还可以给出字符的显式清单，可以匹配清单中的任一字符。显式清单用方括号 [ ] 括起来。具体的例子如下：

```
$ ls /dev/tty?[23456]
/dev/ttyS2  /dev/ttyp2  /dev/ttyq2  /dev/ttyr2  /dev/ttys2
/dev/ttyS3  /dev/ttyp3  /dev/ttyq3  /dev/ttyr3  /dev/ttys3
/dev/ttyS4  /dev/ttyp4  /dev/ttyq4  /dev/ttyr4  /dev/ttys4
/dev/ttyS5  /dev/ttyp5  /dev/ttyq5  /dev/ttyr5  /dev/ttys5
/dev/ttyS6  /dev/ttyp6  /dev/ttyq6  /dev/ttyr6  /dev/ttys6
```

这里列出的文件符合在 /dev/tty 后面跟以任何单个字符，再跟以 2 到 6 的数字中的任何一个数字的模式。

匹配的范围还可用一对字符中间加连字符（-）来表示，如下面命令也得到同样的结果：

```
$ ls  /dev/tty?[2-6]
```

最后一种路径名扩展方式允许指定整个字的清单。在这种情况下，对整个字进行匹配而不是对单个字符进行匹配。清单中字与字之间用逗号分开，并用花括号({ })括起来，下面就是一个例子：

```
$ mkdir /usr/tmp/{bin,doc,lib,src}
$ ls /usr/tmp
bin doc lib src
```

所有这些路径名的扩展都由 shell 自身完成，因而 shell 调用的命令没有必要去处理这些特殊字符。这同样表示：如果写自己的命令，譬如说用 C 语言写，这些命令同样能用 shell 提供的服务，不必由这些命令自己去完成路径名的扩展。

有时用一条命令时（或写命令时），送给命令的参数中包含部分或全部特殊字符。这时，要用到引用（guoting）技术。它告诉 shell 不管这些被引用字符的特殊含义，将它们作为普通字符一样对待。

bash shell 提供三种引用方法：转义字符（ \ ），单引号（ ' ）和双引号（ " ）。

放在任何 shell 特殊字符前面的反斜杠字符（ \ ）告诉 shell 不管它的特殊含义。如果使用这种方法，每个需要抑止特殊含义的特殊字符前面都要放一个转义字符：

```
$ cd
$ mv text/motd text/m\ * \?
$ ls text
m*?       passwd
```

如果将字符串放在一对单引号之间，则单引号之间的所有字符的特殊含义都将被抑止：

```
$ cat 'text/m*?'
motd contents in here.
```

如果将字符串放在一对双引号之间,所有在本章内见到的路径名扩展字符的特殊含义也将被抑止:

```
$ mv "text/m*?" text/motd
$ ls text
motd    passwd
```

注意:还有一些尚未见到的 bash 使用的特殊字符,即使将它们放在双引号中仍然保留它的特殊功能。在第 6 章中,当我们讨论 shell 的程序设计功能时会看到这些字符。

4.2 输入/输出重新定向

Linux 命令的基本设计思想是提供许多有效地完成特殊任务的命令,这些命令又可以很容易地结合在一起完成复杂的功能。

默认的情况是 UNIX (因此,也包括 Linux) 命令从键盘接受输入,并将命令的输出送给屏幕显示。它们的错误信息也送到屏幕上显示。有时候,从文件接受输入或将结果送到文件中去是很有用的。在这种情况下,在编写命令时,增加文件名作为附加的或可选的参数是能做到的,一些命令也是这样做的。但是,如果每条命令都要增加这些并不经常使用的功能,将会不必要地增加命令的执行文件的长度。

为了避免这样做,Linux 和 shell 一起提供了重新定向的功能。在必要时,允许将程序的标准输入和输出进行重新定向。考虑下面的命令:

```
$ ls  -l  /usr/tmp  >dir
```

这一命令的前半部分产生 /usr/tmp 目录的长清单。在正常情况下,在屏幕上显示清单。在命令后面跟以大于号 (>)和文件名时,清单将送到指定的文件中,而不在屏幕上显示清单。使用这种重新定向功能时,如果指定的文件不存在,将建立这一文件。如果指定的文件存在,它的原有的内容将被覆盖。可以方便地验证实际发生的情况:

```
$ cat dir
total 4
drwxr-xr-x  2 pc      book        1024 May 22 23:31 bin
drwxr-xr-x  2 pc      book        1024 May 22 23:31 doc
drwxr-xr-x  2 pc      book        1024 May 22 23:31 lib
drwxr-xr-x  2 pc      book        1024 May 22 23:31 src
```

ls 命令并不知道 > 符号的特殊功能。事实上,ls 命令甚至不知道 > 符号的存在。因为 shell 将任何命令行参数送给 ls 命令以前已将 > 符号和后面的文件名称移去。ls 命令执行时

像往常一样将它的输出送到标准输出设备。由 shell 在 ls 命令执行之前将它的输出从屏幕重新定向到指定的文件。其他重新定向功能也是类似的。

有时候,希望使用输出重新定向功能,将另一条命令输出的结果追加到已有的文件的后面。这时候,可以使用追加重新定向操作符(>>)。它将两个大于符号放在一起:

```
$ ls /usr/tmp >>dir
$ cat dir
total 4
drwxr-xr-x  2 pc      book        1024 May 22 23:31 bin
drwxr-xr-x  2 pc      book        1024 May 22 23:31 doc
drwxr-xr-x  2 pc      book        1024 May 22 23:31 lib
drwxr-xr-x  2 pc      book        1024 May 22 23:31 src
bin  doc  lib  src
```

顺便提一下,如果在 cat 命令中不给出文件名,它将从标准输入设备(键盘)接受输入,一直到遇到文件结束符为止。即在键盘上在一行的开始按 Ctrl-d 将结束从键盘的输入。知道这些以后,就可以用下面的简单命令利用重新定向功能从键盘上直接将正文送到文件中去:

```
$ cat >text.file
any text entered here goes into text.file up to
Ctrl-d
$
```

和程序的标准输出重新定向一样,程序的错误输出也可以重新定向。程序的标准输出和错误输出通常作为两件不同的事情对待,可以对它们分别进行重新定向。例如,要在屏幕上看到程序的正常输出结果,但又要将程序的任何错误信息送到一个文件中去以备以后检查,可以用下面的命令做到这一点:

```
$ ls  /usr/tmp  2>err.file
bin doc lib src
```

使用符号 2>(或追加符号 2>>)表示对错误输出设备重新定向。

为了将标准输出和错误输出同时送到同一文件中,可以使用另一个输出重新定向操作符(&>):

```
$ ls  /usr/tmp  &>output.file
```

如同程序的输出可以写到文件中去一样,它的标准输入也可以读自文件,而不是从键盘输入。这可以用小于符号(<)作重新定向操作符:

```
$ wc  </etc/passwd
```

```
21   42   775
```

由于由 shell 处理小于符号后面的部分，wc 命令看不到文件名 /etc/passwd，它像不包括文件名一样，认为输入来自键盘。

另一种输入重新定向称为 here 文档。它告诉 shell 当前命令的标准输入来自命令行。here 文档的重新定向操作符使用(< <)。它将一对分隔符 (delimiter) 之间的正文重新定向输入给命令：

```
$ wc < <delim
> this text forms the content
> of the here document, which
> continues until the end of
> text delimiter
> delim
       4      17      98
```

当将正文送入 here 文档时，不必在每行前面加大于符号 (>)，它们由 shell 作为提示符提供。shell 用这个提示符告诉你当前的命令还未结束。它在执行命令之前将等待更多的输入。

在 < < 操作符后面，任何字符都可以用作正文开始前的分隔符，本例中使用 delim 作为分隔符。here 文档的正文一直继续到遇到另一个分隔符为止。第二个分隔符应出现在新行的开头。这时 here 文档的正文，不包括开始和结束的分隔符，将重新定向送给命令 (本例中的 wc)，作为它的标准输入。

利用重新定向将命令组合在一起，实现系统单个命令未曾提供的新功能是完全可能的。例如，为了统计目录中的文件数目，可以使用下面的命令序列：

```
$ ls /usr/bin >/tmp/dir
$ wc -w </tmp/dir
   459
$ rm /tmp/dir
```

开始使用 ls 命令列出 /usr/bin 目录的内容，将结果重新定向送给 /tmp/dir 文件。第二条命令对 ls 输出文件的字数进行统计，并显示总数为 459。这个目录中的文件真不少！和这个例子一样，应在结束工作时，将在/tmp 目录下建立的临时文件删去。这样作不致于因为忘记而白白浪费磁盘空间。

4.3 管道

将一个程序的标准输出写到一个文件中去，再将这个文件的内容作为另一条命令的标准输入，等效于通过临时文件将两个命令结合在一起。这种情况很普遍，需要 Linux 提供一种功能。它不需要或不必使用临时文件就能将两条命令结合在一起。这种功能称为管道 (Pipe)。

管道使用竖杠字符 (|) 作为重新定向操作符。如果用管道重写上面的命令就得到：

```
$ ls  /usr/bin | wc  -w
    459
```

Linux 是多任务操作系统。执行上面的命令行或者管道行（Pipeline）时，将同时运行 ls 和 wc 两条命令。所以，从 ls 命令的输出将由 wc 命令读入。

管道行不限于将两条命令结合在一起。任何数目的命令都可以用管道操作符将相邻的一对命令结合在一起。这种安排使得第一条命令的输出成为第二条命令的输入，而第二条命令的输出又成为第三条命令的输入等等。

4.4 后台作业

到现在为止的所有例子中，当送入命令并按 Enter 或 Return 键后，系统似乎转去执行你的命令。在命令执行完毕后，shell 送回提示符请你送入下一条命令。图 4.1 说明为什么是这样。

从图中可以看到，一旦 shell 启动新进程使命令运行时，它自己就转入休眠状态，等待命令执行完毕。在命令结束时，将送回一个唤醒 shell 的信号。shell 将送出一个提示符等待你输入下一条命令。

Linux 是多任务系统，没有理由认为 shell 不能和命令同时运行。在命令开始执行它的任务时，shell 立即送回提示符（见图 4.2）。这是非常有用的。例如，当命令要用很长时间去完成它的任务时，你就可以去做其他事情。

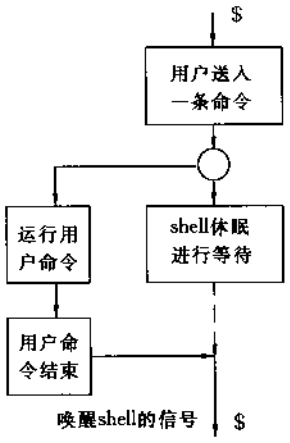

图 4.1 shall 等待命令结束

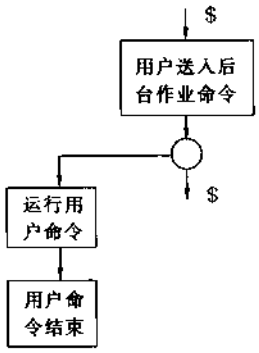

图 4.2 执行后台作业

shell 等待命令执行完毕，还是立即送回提示符，这完全由你来作出决定。如果你不要 shell 等待而是让它立即送回提示符，你要做的只是在命令行后面加上 & 字符。这称为将命令放到后台（background）去执行：

```
$ ls -lR / >/tmp/ls.lR &
[1] 341
$
```

其中 ls 命令的输出将写到 /tmp/ls.lR 文件中。在命令行后面加 & 字符表示命令放在后台执行,shell 不等待命令执行完毕。ls 命令中的 -R 开关将产生递归的目录清单输出。从起点开始(这里是根目录)向下列出所有目录的清单。这里将产生整个目录层次结构的分类的长清单,这是一项需要很长时间才能完成的任务。命令行后面是 shell 显示的数字,方括号中的数字是这一个命令行的后台作业号。这将在 4.5 节中说明。第 2 个数字是执行中的命令的进程识别号 (PID)。如果牵涉到管道行,将是行中最后一个命令的 PID。

一般说,后台执行的命令必须不从键盘接受输入。因为 shell 给出提示符后,所有键盘输入都送给 shell。默认的条件是如果后台作业试图从键盘接受输入,它的执行将被停止,它的所有活动都被挂起。

4.5 作业控制

每次用 bash 执行一条一命令时,它都给命令分配一个作业号(job number)。当多个命令在管道行中给合在一起时,将它们作为一个作业对待,给它们分配一个作业号。shell 用作业号对作业进行控制。

作业控制允许将进程挂起并在以后恢算进程的执行。shell 保存当前执行的作业清单。它可以用 jobs 命令进行显示。为了挂起当前的前台作业,只要在键盘上按 ctrl-z。这样做时,该作业将停止运行,并像下面的例子一样送回 shell 提符:

```
$ cat >text.file
Ctrl-z
[1]+  Stopped                 cat >text.file
$ jobs
[1]+  Stopped                 cat >text.file
```

在上面的命令序列中,在键盘上按 ctrl-z 后,它将挂起当前执行的 cat 命令。使用 jods 命令显示 shell 的作业清单,其中包括具体的作业,它的作业号以及作业当前所处的状态。

当恢复进程执行时,有两种选择:可以用 fg 命令让它回到前台执行。在这种情况下,它将从挂起的位置继续向下执行。也可以用 bg 命令将这一作业放到后台去执行。这种情况下,它像一开始就在命令行后面加 & 字符一样进行执行。

注意:上例中 cat 命令从键盘接受输入。这表明如果将这一作业放到后台去执行,它将立即从键盘进行非法的读入操作,因而再一次被系统自动挂起:

```
$ bg
[1]+ cat >text.file &
$ jobs
[1]+  Stopped (tty input)    cat >text.file
```

按照约定,fg 和 bg 命令对最近停止的作业进行操作。当你用 jobs 命令输出 shell 的作业清单时,这一默认作业的作业号后跟有 + 号。如果不是恢复默认作业,而是希望恢复作业表中

其他作业的运行,可以在命令中用%号后面跟作业号的办法给以明确规定。

```
$ fg  %1
cat  >text.file
```

fg命令将cat作业恢复到前台运行。所以,后面的键盘输入(到ctrl-d为止)将直接送到text.file文件中。

有时,不是恢复挂起的作业的运行,而是结束作业。这可以用kill命令来完成:

```
$ cat >text.file
Ctrl-z
[1]+  Stopped                 cat >text.file
$ jobs
[1]+  Stopped                 cat >text.file
$ kill %1
[1]+  Terminated              cat >text.file
```

在还有作业被挂起的情况下,如果试图退出系统(发出logout命令),将得到出错警告信息,logout命令失效:

```
$ cat >text.file
Ctrl-z
$ logout
There are stopped jobs.
```

在这一点上,可以处理作业,或者再次退出系统,作业将自动结束,成功地完成退出系统。

4.6 历史表

在Linux系统上工作时,经常会重复执行同一组命令。例如,开发程序和查错时,将重复进行标准的循环。编辑源程序,编译源程序,运行可执行文件来测试它的性能,然后再从头开始。为了避免重复输入同样的命令,bash在送入命令时将它们保存起来。当需要时可以重复使用这些命令。bash用历史表(history list)保存这些命令。历史表一般能保存500行命令。

这样规模的历史表足以容纳几天工作中送入的所有命令。为了能使用这样规模的历史表,在你退出登录时,bash自动将当前的历史表保存到一个文件中。默认的文件名称是.bash_history,它存于起始目录下。注意:文件名称开头的句点。它表示除非在ls命令中加上-a开关,否则它将不在ls命令输出的清单中出现。再次登录时,并开始下一轮对话过程,shell自动将历史文件的内容加载到历史表中,相当于让你从上次对话过程的结束处继续新的对话过程。可以用history命令看到保存的命令行清单。由于历史表长达500行,需要将history命令的输出通过管道送给more或tail命令:

```
$ history | tail -5
  511  cat >text.file
  512  cd ..
  513  ls -al
  514  cd book
  515  history | tail -5
```

历史表中的每一行称为一个事件(event),行号称为事件号。一旦知道要重复的行的事件号,就很容易执行它。假定要重复执行上例中的历史命令(事件号515),只要送入历史替换操作符(!)和事件号就行了:

```
$ ! 515
history | tail -5
  512  cd ..
  513  ls -al
  514  cd book
  515  history | tail -5
  516  history | tail -5
```

如果你要重复最后一条命令,只要送入特殊符号 !! 就行:

```
$ !!
history | tail -5
  513  ls -al
  514  cd book
  515  history | tail -5
  516  history | tail -5
  517  history | tail -5
```

除了用事件号以外,也可以让 shell 对历史表进行正文搜索找出特定的命令。这一点可以用历史替换操作符加要搜索的字符串来实现。shell 将向后搜索历史表找出以指定字符串开头的第一行命令并执行它:

```
$ ! ls
ls -al
-rw-r--r--   1 pc     book     3392 May 29 07:55 .bash_history
-rw-------   1 pc     book       25 May 26 1994 .profile
drwxr-xr-x   3 pc     book     1024 May 28 23:30 book
```

对照前面的例子,可以看出这里执行的是513号事件。如果指定的字符串用一对问号括起来,Shell 将对整个命令行进行搜索,不限于在命令行的开头进行搜索。如果后一个问号的后面没有其他内容,则可以省略它:

```
$ !?456
ls /dev/tty?[23456]
dev/ttyS2  /dev/ttyp2  /dev/ttyq2  /dev/ttyr2  /dev/ttys2
dev/ttyS3  /dev/ttyp3  /dev/ttyq3  /dev/ttyr3  /dev/ttys3
dev/ttyS4  /dev/ttyp4  /dev/ttyq4  /dev/ttyr4  /dev/ttys4
dev/ttyS5  /dev/ttyp5  /dev/ttyq5  /dev/ttyr5  /dev/ttys5
dev/ttyS6  /dev/ttyp6  /dev/ttyq6  /dev/ttyr6  /dev/ttys6
```

用这种方法你能重复执行一条命令,即使它的命令名称最近用于其他不同的用途。

有时,你愿意重复执行前面的命令,但要作一些小的修改。这时可以在命令行后面指定一项替换:s/old/new。其中被选命令行中的 old 字符串的第一次出现将被 new 字符串取代,如:

```
$ !? 456?:s/234/4/
ls /dev/tty?[456]
dev/ttyS4  /dev/ttyp4  /dev/ttyq4  /dev/ttyr4  /dev/ttys4
dev/ttyS5  /dev/ttyp5  /dev/ttyq5  /dev/ttyr5  /dev/ttys5
dev/ttyS6  /dev/ttyp6  /dev/ttyq6  /dev/ttyr6  /dev/ttys6
```

如果历史替换达到这样的复杂程度,也许重新输入所需的命令行会更方便。

4.7 命令行编辑

为了简化寻找、修改和重新执行历史表中的命令的任务,bash 同样允许你对命令进行简单的编辑。如果键盘支持光标移动键(箭头键)。每按一次上箭头键,shell 将在历史表中后移一行,并显示上一命令行。按下箭头键则向前移一行。当找到需要的命令时,只要按 Enter 或 Return 键就重复执行这条命令。并将这行命令加到历史表的最后。如果需要在执行命令前编辑这一条命令,可以用左、右箭头键将光标移到所要的位置,然后用 backspace 键删去光标前面的字符,接着敲入取代的字符。当完成命令行的修改后,按 Enter 键将执行这一条命令。

4.8 命令补全

bash 还有另一项有用的功能,它可以为你补全命令行。在送入命令的任何时刻,可以按 Tab 键。当这样做时,shell 将试图补全此时已输入的部分命令。例如,用 passwd 命令改变口令时,可以像下面那样去做(其中符号 < Tab > 表示按 Tab 键):

```
$ pass < Tab >
```

当送入这组字符时,bash 知道送入的是命令的名称。它将进行搜索,找出以 Tab 键前面的字符串开头的命令名称。shell 能找到的以 pass 开头的唯一命令是 passwd。所以,它将在输入的字符串后面补充其他字符以补全你的命令名称。

如果你输入的字符串不足以使 bash 唯一地确定它应该使用的命令，它将发出警告声。再按一次 Tab 键将使 bash 显示可能补全的命令名称清单：

```
$ pas<Tab> <Tab>
passwd   paste
```

这样，就能在命令中增加足够的字符，使得下次按 Tab 键时 bash 能克服多义性。

除了用这种方式补全命令名称外，当用文件名作为命令参数时，bash 也能补全文件名称：

```
$ tail -2 /etc/p <Tab>  <Tab>
passwd  printcap  profile
$ tail -2 /etc/pa <Tab>
pc:fjKppCZxEvouc:500:500::/usr1/pc:/bin/bash
carey:Yt1a4ffkG2r02:501:500::/usr1/carey:/bin/bash
```

这里我们打算显示 passwd 文件中的最后两行。第一次使用 /etc/p<Tab> 时，shell 不能找出文件名。第二次按 Tab 键时，显示了/etc 目录中的三种选择：passwd，printcap 和 profile，因为它们都以 p 字母开头。从这一清单可知，再增加一个字符（本例中为 a）就可以克服 shell 的多义性。正确地得到了所需执行的命令行。

4.9 shell 函数

用 shell 函数来组合和存储一组供以后执行的命令是一种方便的途径。shell 函数定义的形式如下：

```
name( )  { list; }
```

其中 name 是用来执行函数的命令名称。而 list 则是命令或管道行清单，它们之间用分号分开。list 用来规定函数执行的任务。

例如，若送入下列函数定义：

```
$ ll( )  { ls -l; }
```

就能用 ll 作为命令来完成 ls -l 命令同样的功能：

```
$ ll
drwxr-xr-x  3 pc     book     1024 May 28 23:30 book
```

有了 ll 的定义，可能会认为可以用 ll -a 来执行 ls -al 命令。遗憾的是情况并不是这么简单。这是做得到的，但 ll 的定义必须按以下的方式加以改变：

```
$ ll( ) { ls -l $*; }
```

增加 $* 字符表示在 ll 命令后面送入的任何参数或命令开关应插到 ls -l 命令中取代 $* 字符。利用 ll 新定义就能得到预期的结果：

```
$ 11 -a
-rw-r--r--  1 pc    book    3392 May 29 07:55 .bash_history
-rw-------  1 pc    book      25 May 26 1994 .profile
drwxr-xr-x  3 pc    book    1024 May 28 23:30 book
```

在第 6 章讨论 shell 程序设计时，将对 $* 和有关题目作进一步说明。

shell 的函数体可以由若干条命令或管道行组成，命令或管道行之间用分号分开。下面是一个例子（用低效率方式编程，只是为了示范的目的），用来显示给定目录中的文件数。这一函数称为 lswc，因为它组合了两条命令的功能：

```
$ lswc( ) { ls $* >/tmp/dir; wc -w </tmp/dir; rm /tmp/dir; }
```

定义中用了 $* 字符，允许 lswc 命令接受一个参数。这一参数必须是 lswc 命令操作的目录的路径名：

```
$ lswc /usr/bin
   459
```

练习

本组练习的目的之一是使你习惯于从系统手册页中去查找所要的资料。如果不能立即想到问题的答案，不必为此担心。只要查找相应的手册页，就能找到所要的其他信息。

编写和检验完成下列各项功能的命令管道行：

1. 显示当前系统中的登录名的数目。
2. 显示当前系统中有多少个进程。

说明下列管道行分别完成什么功能：

3. df -a | wc -l
4. who | wc -l

答案

1. 这个问题有几种解决方法。它们都对/etc/passwd 文件的行数进行计数。因为系统中每个登录名在文件中各占一行。

   - $ cat /etc/passwd | wc -l

   这里用 cat 命令列出文件内容，然后将结果通过管道送给 wc 命令，由它进行行数统计。

   - $ wc -l </etc/passwd

这里直接使用 wc 命令，对它的输入进行重新定向，取得同样的效果。

- `$ wc -l /etc/passws`

最后，在 wc 命令中直接用文件名作为参数。

2. 可以用 ps 命令列出系统中的进程，每个进程各占一行。对它输出的行数进行计数，应该得到进程数的计数。但有个问题，ps 命令的输出中包括一个标题行，需要加以抑止。查手册页可知，做这件事应该使用 -h 开关。所以，解决的方法是：

```
$ ps -axh | wc -l
```

3. df -a 命令列出系统中所有的文件系统。这些文件系统装配在一起构成系统总的目录层次结构。每个装配的文件系统在命令的输出中各占一行。对输出结果的行数进行计数，应得出装配的文件系统总数。问题同样出在 df 输出中的标题行。但和 ps 命令不一样，它不能被抑止。这表示从管道行得到的计数比实际数多一个。如果这个问题是重要的，也有办法解决。参阅 6.7.2 节。
4. 这里用到了一个新命令，查阅手册页可知：who 命令用来列出所有登录的用户，每个用户各占一行。对行数进行计数，就能得到当前登录的用户数。

第五章　工具和实用程序

Linux 系统还有更多的命令可供普通用户使用。本章的重点将放在一些更复杂的命令上。这些命令不需要借助程序设计技术，即可用单条命令或命令的组合来做许多有用的事情。

本章说明的许多命令从标准输入设备接受输入，并将它们的输出送到标准输出设备。这使它们成为构造更复杂的管道行的理想的候选命令。其中，前一条命令的输出通过管道作为下一条命令的输入。这一类命令称为过滤程序（filter）。这些命令读它们的标准输入，对它完成某些转换，然后将结果送到标准输出。

5.1　搜索和排序

搜索文件找出包含特定字符串的行，或在清单（list）中搜索特定的项（item），这些都是很常用的操作。另一项常用的操作是按某种顺序对行（line）进排序。Linux 提供完成这两项操作的命令。

5.1.1　grep

讨论的第一个属于过滤程序的命令称为 grep。它是通用规则表达式分析程序（General Regular Expression Parser）的缩写。它的含义是：grep 命令可以在它的输入中搜索指定的字符串模式（Pattern）。输入中所有包含指定字符串模式的行组成 grep 命令的输出。例如，要找出 carey 用户是否在系统中登录，只要在 passwd 文件中搜索这一用户名称的字符串：

```
$ grep carey /etc/passwd
carey:Yt1a4ffkG2r02:501:500::/usr1/carey:/bin/bash
```

grep 产生的输出表明，在系统中有 carey 用户。注意：grep 命令并不仅对 passwd 文件的每一行的用户登录名字段进行搜索，它对整个文件进行搜索。这说明虽然 grep 找到了你要找的字符串，但并不一定符合你预期的要求，如：

```
$ grep public /etc/passwd
uucp:*:10:14:uucp:/var/spool/uucppublic:
```

本例中，虽然没有 public 用户在系统中登录，grep 还是产生了一行输出，为解决这个问题，grep 使用了 vi 使用的类似办法，使用规则表达式。

grep 命令用于规则表达式中的基本特殊字符集和 vi 使用的相同。如果在 grep 命令中加 -E 开关，还可以使用一组扩充的特殊字符集。下表给出 grep 使用的大多数特殊字符和它们的具体含义，在中间列中，字母 B 和 E 分别表示基本的和扩充的：

^	B	在每行的开始进行匹配；
$	B	在每行的末尾进行匹配；
\<	B	在字的开始进行匹配；
\>	B	在字的末尾进行匹配；
.	B	对任何单个字符进行匹配；
[*str*]	B	对 *str* 中的任何单个字符进行匹配；
[^*str*]	B	对任何不在 *str* 中的单个字符进行匹配；
[*a* - *b*]	B	对 *a* 到 *b* 之间的任何字符进行匹配；
\	B	抑止后面的一个字符的特殊含义；
*	B	对前一项（*item*）进行 0 次或多次重复匹配；
+	E	对前一项进行 1 次或多次重复匹配；
?	E	对前一项进行 0 次或 1 次重复匹配；
{*j*}	E	对前一项进行 *j* 次重复匹配；
{*j*,}	E	对前一项进行 *j* 次或更多次重复匹配；
{,*k*}	E	对前一项最多进行 *k* 次重复匹配；
{*j*,*k*}	E	对前一项进行 *j* 到 *k* 次重复匹配；
s|*t*	E	匹配 *s* 项或 *t* 项中的一项；
(*exp*)	E	将 *exp* 作为单项处理。

不要忘记，在 vi 和 grep 中使用规则表达式的主要差异。在 vi 编辑程序中，输入的任何字符只被编辑程序本身看到和使用。而 grep 命令看到的任何参数必须先经过 shell。由于在规则表达式中使用的一些特殊字符对 shell 同样具有特殊含义，要告诉 shell：不管这些字符。这样，grep 才有机会见到它们。用引用方法实现这一点。shell 的引用方法见 4.1 节。

回到前面的例子，搜索 passwd 文件找登录名。现在，形成表示 public 不是有效登录名的 grep 命令行将是一件容易的事：

```
$ grep '^public' /etc/passwd
```

前面的 ^ 字符强制 grep 命令只在每行的开头找 public 。整个搜索模式（pattern）用单引号括起来，使 shell 不理会它们。shell 只将单引号去掉，将搜索模式送给 grep 命令。grep 命令不产生输出的事实表示：passwd 文件中没有以 public 开头的行。

作为一个稍复杂的例子，显示 passwd 文件中那些只有登录名而没有设置口令的行。如果行中第 2 字段（第 1,2 个冒号之间）是空的，就表示这些行没有设置口令：

```
$ grep -E '^[^:]+::' /etc/passwd
sync::3:2::/:/bin/sync
mtos::9876:9876:student login:/mtos/home:/bin/bash
load::9876:9876:mount floppy:/:/mtos/bin/load
unload::9876:9876:umount floppy/:/mtos/bin/unload
```

从输出结果可以看出，本例中有 4 个用户没有设置口令。他们的登录名是：sync mtos load 及 unload。

为了充分理解这一点,需要对本例中使用的搜索模式('^[^:]+::')作些解释。开始和结束的单引号告诉 shell:不管这一字符串的内容,将它不加改变地传给 grep。搜索模式用来找3件事:在行的开头,跟以登录名,跟以一对冒号。后者表示空的口令字段。行的开头用第一个字符来找到,空的口令字段用字符串后面的一对冒号来进行匹配,其余部分 [^:]+ 用来匹配任何登录名。要理解这一点,你要考虑登录名是由到第一个冒号为止的可变数目的字符所组成。这正是搜索模式所表示的:一个或多个非冒号字符。grep 命令中的 -E 开关是必要的。这样,它可以使用扩充特殊字符集中的 + 号重复操作符。

如果没有对 grep 命令指定任何搜索的文件,grep 命令将从标准输入设备接受输入。在这种状态下,grep 可以通过管道从前一条命令接收输入。下面的例子是一管道行,用来列出根目录下普通用户具有写权限的任何子目录:

```
$ ls -l / | grep '^d.....w'
drwxrwxrwt  10 root     root         1024 May 30 14:30 tmp
```

像所预期那样,这一管道行的输出结果表示:普通用户对 /tmp 目录具有写权限。它用 ls -l / 命令产生根目录下的所有文件和子目录的长清单。ls 命令的输出通过管道送给 grep 命令,grep 命令寻找以 d 开头 (^d) 的那些行。后面是一组权限标志,目录拥有者的权限标志可以取任何值(...),用户组的权限标志可以取任何值(...),其他用户的读权限标志可以取任何值(.)。最后其他用户的写权限标志设为 w,表示其他用户有写目录的权限。

grep 命令的有用的开关如下:

-E	用扩充规则表达式进行模式匹配;
-i	不区分大小写;
-n	在每一输出行前显示文件内的行号;
-q	与其他命令一起使用时,抑止输出显示;
-s	抑止文件的出错信息;
-*num*	在每 -匹配行前后各显示 *num* 行。

5.1.2 find 命令

find 命令的主要作用是对树形目录层次结构进行彻底检查。从给定的起点开始通过目录的所有分支一直检查到结点。可以在命令中指定一组操作,对 find 命令发现的每个文件和子目录进行操作。find 命令最常用来产生给定目录下的所有文件和子目录清单:

```
$ cd
$ find . -print
./backup
./backup/motd.bak
./backup/passwd.bak
./text
./text/passwd
./text/motd
```

本例中先用 cd 命令将当前工作目录改为起始目录。find 命令的两个参数是点和 -print，它们告诉 find 命令从当前目录开始搜索，并显示它发现的所有文件和子目录名称。

find 命令输出的文件名清单写到标准输出设备，因而也可以通过管道供其他命令使用。下面的例子用来说明这一点：

```
$ find . -print | grep passwd
./backup/passwd.bak
./text/passwd
```

事实上，你将很快看到，这一应用可以直接用 find 命令完成，不必借助于 grep 命令。

find 命令的一般格式如下：

```
find  pathname  -expressions
```

其中 pathname 可以是任何目录路径名清单，对清单中每个目录进行递归搜索来产生文件名输出。而 expressions 则是任何表达式的清单，用来定义条件和对每个被发现的文件进行的操作。如果没有指定 pathname，默认值是当前目录。如果没有指定 -expressions，，默认值是 -print。例如：

```
$ find
./backup
./backup/motd.bak
./backup/passwd.bak
./text
./text/passwd
./text/motd
```

find 命令使用三种表达式：选项，条件和操作。每个表达式都送回一个值。这个值是真是假取决于使用的表达式和对它的评价方式。当 find 命令具有一个以上的表达式时，表达式之间用逻辑操作符来控制它们的执行。

给出表达式 e1 和 e2：

e1 -a *e2*	仅当 *e1* 为真时，对 *e2* 求值；
e1 *e2*	和 *e1* -a *e2* 相同；
e1 -o *e2*	仅当 *e1* 为假时，对 *e2* 求值；
e1 , *e2*	对两个表达式都求值，先 *e1*，后 *e2*。

复合表达式的值是最后实际求值的表达式的返回值。由两个以上表达式组成的复合表达式的求值按自左至右的顺序进行，除非括号（）改变了求值顺序。所以，

```
e1   -o   e2   -a   e3
```

将和下面一样进行求值：

```
(e1   -o   e2)   -a   e3
```

除非加括号改变了顺序，如：

```
e1   -o   (e2   -a   e3)
```

还可以使用逻辑非操作符：

```
!    e1
```

当 e1 为假时，结果为真；反过来也一样。

下表详细说明可以和 find 命令一起使用的一些表达式。每个表达式的类型和它返回的值：

-mount	选项表达式，用来防止 find 命令的搜索范围超出当前文件系统的边界。返回值常为真。
-group *grp*	条件表达式，检查当前的文件是否具有与 *grp* 相同的 GID 或组名。如果两者一致，返回真值，否则返回假值。
-name *pattern*	条件表达式，检查文件名是否和模式 *pattern* 相同。*pattern* 可以用规则表达式给出。必要时使用引号。当文件名与 *pattern* 一致时，返回真值，否则为假。
-type *t*	条件表达式，检查当前文件的类型是否是 *t*。对目录讲，*t* 值可以是 *d*。对普通文件讲，*t* 值可以是 *f*。对连接讲，*t* 值可以是 l 等等。如果当前的文件的类型是 *t*，返回真值，否则为假。
-user *usr*	条件表达式，检查当前的文件的所有者或 UID 是否是 *usr*。如果两者一致，返回真值，否则为假。
-exec *cmd*;	操作表达式，用来执行 *cmd* 命令。如果要将当前的文件名传送给命令，应该加{}标记，分号用来表示 *cmd* 的结束，并和后面可能出现的表达式分开。如果成功地执行了 *cmd* 命令，返回真值，否则为假。
-print	操作表达式，将当前的文件名送到标准输出设备显示，返回值常为真。

注意：find 命令不允许超越系统对文件和目录的权限设置。因此，如果让 find 命令从一个目录开始搜索，而这个目录下有不能访问的子目录，find 命令每遇到一个这样的目录，将向标准输出设备传送一条错误信息。由于默认的标准出错信息输出设备是屏幕，如果使用了 -print操作，可能实际输出许多出错信息，让你看都看不过来：

```
$ find / -name passwd -print
find: /var/spool/cron: Permission denied
find: /var/spool/atjobs: Permission denied
find: /var/spool/atspool: Permission denied
/usr/bin/passwd
```

```
find: /usr/doc/ttysnoop: Permission denied
/etc/passwd
find: /root: Permission denied
```

最简单的做法是抑止这些不必要的输出,将它们重新定向,送到一个文件中去。但这样做以后又要删除一个不必要的文件。系统专门提供了一个文件,可以将不需要的东西送到那里去。这个文件的路径名是:

```
/dev/null
```

这是特殊安排的文件,任何时候你将不需要的东西写到它那里去,相当于将它们抛弃。它还作了这样的安排,当试图从它那里读取字符时,经常得到的是文件结束的指示,而不会读到任何其他东西。

利用 null 文件来抑止 find 命令的出错信息输出,上例就可以写成:

```
$  find / -name passwd -print 2>/dev/null
/usr/bin/passwd
/etc/passwd
```

注意:本例中用了两个 find 命令的表达式,它们之间没有操作符(默认的操作符是 -a)。对这一默认的操作符讲,find 命令每产生一个文件名,先对第 1 表达式求值,仅当第 1 表达式的结果为真时,才用到第 2 表达式。本例中第 1 表达式是 -name passwd。它从 find 命令提供的路径名后面找文件名 passwd。如果两者相同,它返回真值,否则返回假值。这一返回值决定了是否用到第 2 表达式。如果第 1 表达式返回值为真,第 2 表达式(-print)就将 find 发现的当前路径名送到标准输出设备去显示。总的结果是:将 find 命令发现的后面带有 passwd 文件名的所有路径名显示出来。

可以用这种方式单独使用 find 命找出路径名清单,不必像前面的例子那样通过管道使用 grep 命令。但是两者是有差别的,-name 表达式只在路径名的末尾找完整的字,而 grep 命令搜索子字符串(substring):

```
$ cd
$ find . -name motd -print 2>/dev/null
./text/motd
$ find . -print 2>/dev/null | grep motd
./text/motd
./backup/motd.bak
```

作为最后一个稍为复杂的例子,让我们看如何产生一个目录子树(directory subtree)下的所有包含特定字符串的普通文件的路径名清单。一种可能的解决办法是用 find 命令产生路径名,然后用 grep 命令检查这些文件中是否包含特定字符串,并显示所有包含特定字符串的文件的路径名。唯一的问题是 grep 命令同时在标准输出设备上输出它发现的行和错误信息。

克服这一问题的一种办法是将标准输出和出错误信息都重新定向送到 /dev/null。但 grep 命令提供了更简单的办法,可以用命令开关抑止这些输出:

```
$ find /etc -type  f -exec grep -q -s mycroft {} \; -print
/etc/HOSTNAME
/etc/hosts
/etc/lilo.conf
```

这里用 find 命令产生 /etc 目录下所有路径名的清单,然后对第 1 表达式 -type f 求值。如果当前路径名属于普通文件,送回真值。然后对所有普通文件执行第 2 表达式 (-exec grep -q -s mycroft {} \ ;),在每个普通文件中搜索 mycroft 字符串。不要忘记,在执行 grep 命令时,{} 已用当前路径名代替,-q 和 -s 开关使 grep 命令不显示标准输出和出错信息。所以,这一表达式只用来产生真值或假值,用来决定是否执行第 3 表达式。第 2 表达式中分号前面的反斜杠用来引用分号,使 shell 不加改变地将它传送给 find 命令,如果第 2 表达式返回真值,将执行 -print 表达式,它的作用是显示当前的路径名。

5.1.3 sort

sort (排序) 命令是一个过滤程序。它将输入的正文中的行按命令中指定的顺序进行排序,并将排序结果送给标准输出设备。sort 命令非常灵活,它将输入行看作是字段的集合,并使用一个或几个字段作为排序键 (sort key),对输入的行进行排序操作。还可以选择 sort 使用的字段分隔符,对文件进行不同类型的排序。

在下面的例子中,我们将使用虚构的口令文件 pw.test:

```
root:awmku76tr43d6:0:0::/root/:/bin/sh
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
```

默认的 sort 字段分隔符为空白字符 (如空格,Tab 等)。pw.test 的行中没有空白字符,因而作为一个字段对待。如果 sort 对这一文件进行排序,它将每行看成一个字,并按字母顺序对它进行排序:

```
$ sort pw.test
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
root:awmku76tr43d6:0:0::/root/:/bin/sh
```

sort 命令可以从命令行中指定的任何文件中接受输入。如果没有指定文件,则从标准输

入设备接受输入。

sort 命令可以使用许多命令行开关,下表是一些最有用的开关:

-b	不管排序键前面的空格字符;
-f	不区分大小写;
-n	将排序键作为数字而不作为正文;
-r	按从高到低的顺序而不是从低到高的顺序进行排序;
-o *file*	将输出送到 *file* 而不是送到标准输出设备;
-t *s*	用 s 代替空白字符作为字段分隔符;
-k *s1*,*s2*	用 *s1* 字段到 (*s2* - *1*) 字段作为排序键。

排序键的说明要比上面给出的更复杂些,它允许给出重复的 -k 开关。所以,可以指定字段的任意组合。sort 命令按命令行中给出的顺序使用字段,字段的顺序不必和文件中出现的顺序一致。每一字段说明符 (s1 和 s2) 可以用 f.c 的形式给出,其中 f 是字段号,c 则是字段中的字符位置。这两个值都从 1 算起。通过下面的例子就能看得更清楚:

-k3	排序键从字段 3 开始到行的结束;
-k3,6	排序键由行中第 3,4,5 字段组成;
-k4,5 -k1,3	排序键是第 4 字段和第 1,2 字段;
-k3.3,4	排序键是第 3 字段,但去掉字段中前 2 个字符;
-k3.2,3.6	排序键是第 3 字段,从第 2 字符开始到第 5 字符共 4 个字符。

回到 pw.test 文件,如果想对这个文件进行复杂的处理,首先应该将它的字段分隔符从默认的空白字符改为冒号。这样,sort 就能访问文件中的各个字段。下面用第 3 字段中的 UID 对文件进行排序:

```
$ sort -t: -k3,4 pw.test
root:awmku76tr43d6:0:0::/root/:/bin/sh
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
pc:bdhd74hs8jh3h:500:50::/usr1/pc:/bin/bash
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
```

作为默认的要求,如果两行的排序键的值相同,将整行作为一个单字进行排序。上例中第 3,4 行就出现这种情况,它们的排序键的值都是 1500。

可以选择你自己的解决办法。当第 1 排序键不能解决问题时,指定第 2 排序键,或更多的排序键。例如,上面的例子中第 3 字段不能确定行的顺序时,接着用第 7 字段进行排序,得到的结果如下:

```
$ sort -t: -k3,4 -k7 pw.test
root:awmku76tr43d6:0:0::/root/:/bin/sh
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
```

```
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
pc:hdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
```

上面两个例子看上去都有点问题,似乎没有正确按 UID 的值进行排序。用的关键字段出现的顺序是 0,1500,1500,500,501。这是因为 sort 将 UID 的内容作为字而不是作为数来对待。因为 0 在 1 前面,1 在 5 前面,所以 sort 做的是对的。但并不一定要这样做。解决的办法是使用 -n 开关,将按数字的顺序对字段进行排序:

```
$ sort -t: -n -k3,4 -k7 pw.test
root:awmku76tr43d6:0:0::/root/:/bin/sh
pc:bdhd74hs9jh3h:500:50::/usr1/pc:bin/bash
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
```

看上去解决了问题。

使用 sort 命令的最后一个例子说明如何用字段的一部分作为排序键,这个例中用口令字段的第 1,2 字符作为排序键,并按相反的顺序显示结果:

```
$ sort -t: -r -k2.1,2.3 pw.test
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
root:awmku76tr43d6:0:0::/root/:bin/sh
```

所有用到 sort 命令的例子中都将结果送到屏幕显示,因为这是标准输出设备。也可以将输出重新定向到管道和文件。有一点要注意,不能将 sort 的输出重新定向到输入文件。因为 shell 在做重新定向的准备工作时会删去你的输入文件。所以不要像下面那样去做:

```
$ sort somefile > somefile
```

如果你一定要将输出送回同一个文件,应在 sort 命令中加上 -o 开关;

```
$ sort somefile -o somefile
```

5.2 修改文件

一些 Linux 命令允许修改文件中的或管道传送过程中的正文内容,本节说明这类命令和它们的功能。

5.2.1 sed 命令

如果要编辑一个文本文件，可用 vi 编辑程序在键盘上用交互方式进行编辑。如果要编辑的正文只存在于两个进程之间的管道之中，情况就不一样了。这就是用到 sed 命令的地方。sed 是一专门设计的编辑程序。它从标准输入设备接受正文的行，并将它的输出送到标准输出设备上去。在它们通过时，对它们完成编操作。为了使用 sed 命令，在送入命令时，需要在命令行中指定要完成的编辑操作。由于 sed 命令只在正文通过它时才能看到正文，它能做的工作是有限度的。例如，让它去完成需要在正文中前后移动的编辑操作是不可能的。

sed 命令的标准操作重复地从输入中读入一行，不加改变地送给输出，一直进行到文件结束为止。如果在命令中指定了编辑操作，只要它是合适的，将在每行读入时进行相应的编辑操作。然后将操作的结果代替原始的输入送给输出。sed 命令还对通过它的正文行数进行计数。因此，它可以对特定的行或一段正文进行操作。

通常 sed 命令从标准输入设备接受输入，如果在命令行中指定一个或多个文件名，它也可以从文件接受输入。

sed 命令的格式为：

```
sed  actions  files
```

其中 actions 是编辑操作的清单。如果只有一项操作，可以直接放在命令中。如果需要 sed 命完成两项以上的操作，可以有两种选择：在命令行中加入这些操作，每项操作前加上 -e 开关；或者将这些操作写到一个文件中去，用 -f 文件开关引导 sed 去访问这一文件。

有时，需要在正文中指定进行编辑操作的行或行的范围。这可以在每项操作前面给出零个，1 个或 2 个行地址来实现。行地址可以用行号或规则表达式表示。在规则表达式中，又可指定在行中搜索的字符串模式。sed 命令使用 grep 命令的基本规则表达式。参阅 5.1.1 节。

没有行地址的编辑操作适用于输入的正文中的每一行。如果给出一个行地址，则此项操作只适用于特定的行。如果给出两个行地址，则此项操作适用于从第 1 地址到第 2 地址之间的所有各行。

让我们看一些应用 sed 命令的例子。仍使用前面的数据文件 pw.test，第 1 个例子说明删除操作：

```
$ cat pw.test | sed '4,$ d'
root:awmku76tr43d6:0:0::/root/:/bin/sh
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
```

cat 命令用来列出 pw.test 文件的内容，将它的输出通过管道送给 sed 命令。由 sed 命令完成单项编辑操作（'4, $ d'）。加引号是必要的，以免和 shell 发生冲突。如果你对当前的情况有否必要使用引号有怀疑，从安全出发还是加上引号。本项操作的含义是：从第 4 行起到文件的最后一行（$）全部加以删除。

使 sed 命令在处理完第 3 行后退出，能达到同样的效果：

```
$ sed 3q pw.test
root:awmku76tr43d6:0:0::/root/:/bin/sh
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
```

其中 pw.test 文件作为 sed 命令的参数给出,不像上例那样通过管道送给它。

有时,需要抑止将所有经过编辑的行送到标准输出设备的默认操作。可以用 -n 开关来实现这一点。用了 -n 开关后,应明确规定那些行应在标准设备上输出。这可以用 p 编辑操作来完成。下面是从文件中送出两行进行显示的例子:

```
$ sed -n 2,3p pw.test
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
```

下面的例子稍复杂一些。它使用了字符转换操作 (y)。字符转换的基本含义是:指定两个等长的字符串。在选择的行地址范围内,将行中出现在第 1 字符串中的每个字符,用第 2 字符串中的对应字符替换。在下例中,选择所有包含 bash 的行。在这些行中将所有的冒号字符转换为下划字符 (_),将所有的零用百分号(%)替换:

```
$ sed /bash/y/:0/_%/ pw.test
root:awmku76tr43d6:0:0::/root/:/bin/sh
pc_bdhd74hs9jh3h_5%%_5%__/usr1/pc_/bin/bash
carey_esJ9ohd8HH89i_5%1_5%__/usr1/carey_/bin/bash
mot_dhjd83kjdJS6D_15%%_6%__/usr1/mot_/bin/bash
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
```

如果在 Y 操作符前面插入惊叹号 (!),则此项操作适用于行地址未被选中的所有各行,而不是选中的各行:

```
$ sed '/bash/! y/:0/_%/' pw.test
root_awmku76tr43d6_%_%__/root/_/bin/sh
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
grex_cj8AjoWE8h8fs_15%%_6%__/usr1/mot_/bin/sh
```

另一个主要的编辑操作 (s) 使你能用另一字符串取代用规则表达式指定的字符串。

```
s/expr/new/flags
```

其中 expr 是 sed 用来搜索的规则表达式，new 是用来取代规则表达式的正文，而 flags 则可以从下面清单中进行选择：

num	一般情况下，只有 *expr* 的首次出现才被取代。如果指定了 *num*(0－9)，则相应的出现次数都应被取代。
g	如果选用 g 标志，expr 的每次出现都被 *new* 取代。
P	如选用 P 标志，将在标准输出设备上显示所有发生取代操作的当前行。
w *file*	此标志将导致发生取代操作的当前行被追加到指定文件 file 的最后。如果文件不存在，则建立新文件。

下面是另一个使用替换操作的例子。修改 pw.test 文件，阻止从 pc 到 mot 的所有用户进行登录。具体做法是将它们的加密口令移去，用字符串 off 取代。并将所有修改过的行追加到 banned 文件中：

```
$ sed '/^p/,/^m/s/:.............:/: off :/w banned' pw.test
root:awmku76tr43d6:0:0::/root/:/bin/sh
pc: off :500:50::/usr1/pc:/bin/bash
carey: off :501:50::/usr1/carey:/bin/bash
mot: off :1500:60::/usr1/mot:/bin/bash
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
```

在本例中，有两点需要解释。^字符表示行的开头。所以，/^p/ ，/^m/ 表示从 p 开头的行起到 m 开头的行为止所有各行。另一点要注意的是加密口令总长 13 个字符。所以，规则表达式 :.............: 将与被选行中两头包含冒号的 13 个字符串的首次出现相匹配(即口令字段) 。

检查 banned 文件的内容，它列出了被拒绝登录的用户名清单：

```
$ cat banned
pc: off :500:50::/usr1/pc:/bin/bash
carey: off :501:50::/usr1/carey:/bin/bash
mot: off :1500:60::/usr1/mot:/bin/bash
```

下面的例子引入了一些新概念。本例假定：本系统有了 shell 程序 (/bin/sh 和/bin/bash) 的新版本。新版本放在 /usr/local/bin 目录下。需要修改口令文件的 shell 路径名，使用户去试用新的 shell 程序：

```
$ sed 's? /bin/.*sh$? /usr/local&?' pw.test
root:awmku76tr43d6:0:0::/root/:/usr1/pc:/usr/local/bin/bash
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/usr/local/bin/bash
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/usr/local/bin/bash
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/usr/local/bin/sh
```

要注意的第一点是：搜索的规则表达式中包含 / 字符。这一字符常用作替换操作的分隔

符。sed 允许选用另一对限界符,它可以是你喜欢使用的任何字符(本例中使一对问号)。另一点是规划表达式后面的 $ 字符,它表示在行的结束处进行比较。因此,表达式 /bin/.*sh $ 将与行末尾的/bin/sh 或/bin/bash 匹配。最后,替换字符串后面的 & 字符本身应该用与规则表达式匹配的正文所代替。这正好将正确的 shell 名称放在/usr/local 字符串后面。

最后的例子用到多项编辑操作。它表明在这种情况下每次进行一项操作,并严格按自左自右的顺序执行:

```
$ sed -e s/:/_:/ -e /carey_/s//cew_/ pw.test
root_:awmku76tr43d6:0:0::/root/:/bin/sh
pc_:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
cew_:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
mot_:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
grex_:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
```

在本例中,要求在每个用户的登录名后面加下划字符,并将登录名 carey_ 改为 cew_。用两项操作来完成,第 1 项操作在每个登录名后面加下划字符,第 2 项操作更改 carey 的名称。要注意的是:在第 2 项编辑操作中,替换命令后面没有指定规则表达式。在这种情况下,将用行地址中的规则表达式的值或前一替换命令中的表达式来代替这一表达式。在本例中即是 carey_,它是第 2 项编辑操作的行地址。

5.2.2 字符转换

可以用 tr 命令完成 sed 命令的编辑操作 y 的类似字符转换功能,但稍有不同。在一些典型的应用中,如将大写改为小写或删除标点符号等,它是一个很快和很简单的程序。这是一个真正的过滤程序。它从标准输入中读正文,将转换结果送到标准输出。tr 命令的基本形式是:

```
tr  str1  str2
```

其中,str1 和 str2 是字符串。在 str1 中出现的字符将被转换为 str2 中的对应的字符。下面是一个例子:

```
tr ABCDEFGHIJKLMNOPQRSTUVWXYZ   abcdefghijklmnopqrstuvwxyz
```

它将所有的大写字母转换为对应的小写字母。可以用更简单的方法来取得同样的效果。连续的字符串可以用第一个字符和最后一个字符中间加连字符来表示:

```
tr A-Z  a-z
```

这些例子表明:两个参数中的字符数必须相同。可能出现两种不同的情况:如果 str1 比 str2 短,将 str2 后面的字符截去 。如果 str1 比 str2 长,将 str2 用它最后一个字符补充使它和 str1 等长。所以:

```
tr  ABC  abcde   等同于 tr  ABC  adc
tr  ABCDE  abc   等同于 tr  ABCDE  abccc
```

前一种情况的实用价值不大,后一种情况很有用。特别是,当它和 -c 命令开关一起使用时。-c 开关的作用是:用所有不在 str1 的字符来代替 str1 中的字符(即 str1 的补)。这使你能用下面的命令将所有非字母和非数字的字符转换为空格字符:

```
tr  -c  A-Za-z0-9 ' '
```

这条命令的副作用是它留下许多连在一起的空格,需要将它们压缩为单个空格字符。tr 命令的 -s 开关用来完成此项任务。-s 开关的功能是将 str2 中的任何字符的多次连续出现压缩为单个字符,如:

```
tr  -cs  A-Za-z0-9  ' '
```

将这条命令用于 pw.test 文件时,将得到下面的输出:

```
$ tr -cs A-Za-z0-9 ' ' <pw.test
root awmku76tr43d6 0 0 root bin sh pc bdhd74hs9jh3h 500
50 usr1 pc bin bash carey esJ9ohd8HH89i 501 50 usr1 car
ey bin bash mot dhjd83kjdJS6D 1500 60 usr1 mot bin bash
grex cj8AjoWE8h8fs 1500 60 usr1 mot bin sh
```

如果你只要 tr 命令将某些字符的多次连续出现压缩为单个字符,可以用下面的命令:

```
tr  -s  str1
```

其中只指定 str1,它将对 str1 中的字符进行压缩操作。

一些特殊字符(如回车和 ESC)难以直接放在转换字符串中,为此提供了特殊的标记,这些标记用反斜杠字符后面跟所需的特殊字符的代码组成:

```
\b     退格字符 (ctrl-h);
\n     新行字符 (ctrl-j);
\r     回车字符 (ctrl-m);
\t     Tab 字符 (ctrl-i);
\\     反斜杠字符;
\000   任何字符都可以用 3 位 8 进制数表示,如:
\033   ESC 字符的 ASCII 代码 (ctrl-[)。
```

这些特殊字符和 C 语言中使用的相同。

tr 命令中使用的最后一个命令开关是 -d

```
tr  -d  str1
```

本开关的用途是删除字符。这一选项只指定 str1,它用来提供需要删除的字符清单。

5.3 简单的数据处理

本节说明一组简单命令。它们允许你以特殊和有用的方式来处理文本和数据文件的内容以及其他命令的输出。

5.3.1 cut 和 paste 命令

sed 和 tr 一类命令在处理文件的行(或记录)时非常有用。但它们在处理文件的的列(或记录的特定字段)时并非那么有效。cut 命令提供了从它的输入行中提取特定列的简单方法:

```
$ cut  -d:  -f1  pw.test
root
pc
carey
mot
grex
```

此命令用冒号作为字段分隔符(-d:)。在 pw.test 文件中提取第一字段(-f1),即文件中的登录名字段。注意:这些命令中使用不同的命令开关来完成同一件事,cut 命令用 -d 来引入字段分隔符,而 sort 则用 -t 来完成同样的任务。

下列说明如何提取登录名和起始目录,并将它分别放在两个不同的文件中(/tmp/p1 和 /tmp/p6),然后用 paste 命令将两个文件中的信息合成一个表:

```
$ cut -d: -f1 pw.test >/tmp/p1
$ cut -d: -f6 pw.test >/tmp/p6
$ paste /tmp/p1 /tmp/p6
root    /root/
pc      /usr1/pc
carrey  /usr1/carey
mot     /usr1/mot
grex    /usr1/mot
$ rm /tmp/p1 /tmp/p6
```

当用 paste 命令将字段接在一起时,它将在字段之间自动插入分隔符。默认的分隔符是 Tab(ctrl-i)。你可以用 -d 开关来指定你自己的字段间的分隔字符。

在前面的 cut 命令中使用特定字符作字段分隔符。但是字段也可以放在记录中固定的字

节位置（fixed-byte position）上，而不必使用分隔符。可以用 -b 开关来指定字节位置。对 cut 命令的 -d 和 -f 开关而言，字段或字节位置可以用一个数，一组数或用逗号分开的数的范围来表示。两种情况下都从 1 开始计数。

下面的问题稍复杂些。从 date 命令输出的日期和时间中提取分钟数。不带参数的 date 命令将用固定格式显示日期和时间：

```
$ date
Sun  Jun  4  17:27:23  BST  1995
```

这一字符串可以看成是用空格分开的字段所组成。这些字段是：

```
sun       每周 7 天的名称；
Jun       月份名；
4         日期；
17:27:23  当天的时间（24 小时时钟，时:分:秒；）；
BST       时区（英国夏令时）；
1995      年份。
```

用 cut 命令从这一字符串中提取字段的唯一问题是月份中的日期字段。如果日期小于 10，它的前面加了空格来保持字段的正确长度。如果告诉 cut 命令字段用空格分开，多出的空格将被看作一个额外的字段。解决的办法是用 tr 命令将多个空格压缩为一个空格：

```
$ date | tr -s ' '
Sun Jun 4 17:27:23 BST 1995
```

现在可以用空格作为分隔符用 cut 命令提取当天的时间字段（字段 4）：

```
$ date | tr -s ' ' | cut -d' ' -f4
17:27:23
```

从这一管道输出的字符串本身又包含以冒号为分隔符的 3 个字段。我们需要提取中间的字段的值。cut 命令能很好地完成这一任务：

```
$ date | tr -s ' ' | cut -d' ' -f4 | cut -d: -f2
27
```

解决这一问题的更简单办法仍然使用 cut 命令。注意 date 命令输出的所有字段长度是固定的，可以算出我们感兴趣的字段的字节位置的范围，并使用 -b 开关来切出这一字段（本例中为第 15 和 16 字节）：

```
$ date | cut -b15-16
```

5.3.2 比较文件内容

一些命令可以用来比较两个文件内容的差别。最简单的命令只告诉你这两个文件是否是同一个文件。这个命令称为 cmp。它的做法是:对作为参数给出的两个文件的内容按字节进行比较,直到发现差别或者到文件结束为止。这个命令的一般形式是:

```
cmp  file1  file2
```

如果 file1 和 file2 两个参数中间的一个用连字符(-)给出,这个文件的正文从标准输入设备取得。如果没有使用重新定向,那就从建盘输入。cmp 命令的输出是两个文件不同的第 1 个字节和行号。

如果两个文件的行已经经过排序,就可以用 comm 命令来进行比较。comm 命令的一般格式是:

```
comm  options  file1  file2
```

如果没有指定选项(options),comm 命令的输出分三列显示。出现在 file1 中而没有出现在 file2 中的行显示在第 1 列。出现在 file2 中而没有出现在 file1 中的行显示在第 2 列,同时出现在两个文件中的行显示在第 3 列。

作为例子,让我们先对 pw.test 进行排序,将排序结果送给 pw1.test 文件:

```
$ sort pw.test -o pw1.test
$ cat pw1.test
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
root:awmku76tr43d6:0:0::/root/:/bin/sh
```

然后用 sed 命令对 pw.test 文件中的一些行进行修改,对结果进行排序,将排序结果送到 pw2.test 文件中去:

```
$ sed /pc/,/carey/s/bash/sh/ < pw.test | sort > pw2.test
$ cat pw2.test
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/sh
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/sh
root:awmku76tr43d6:0:0::/root/:/bin/sh
```

最后,用 comm 命令对这两个文件进行比较,并观察比较结果:

```
$ comm pw1.test pw2.test
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
        carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/sh
                grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
                mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
        pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/sh
                root:awmku76tr43d6:0:0::/root/:bin/sh
```

可以指定的 comm 命令的主要选项有：-1，-2 和 -3，分别用来抑止相应的列的输出。例如，只看那些出现在 pw1.test 文件中而没有出现在 pw2.test 文件中的行，需要抑止第 2，3 列输出，结果如下：

```
$ comm -2 -3 pw1.test pw2.test
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
```

如果只看同时出现在两个文件中的行，需要抑止第 1，2 列输出：

```
$ comm -1 -2 pw1.test pw2.test
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
root:awmku76tr43d6:0:0::/root/:/bin/sh
```

另一个用来比较文件内容的主要命令是 diff。和 cmp 及 comm 相比，这一命令完成更复杂的检查。它对一对文件系统地进行工作，不要求对文件事先进行排序，并显示两个文件中所有不同的行：

```
$ diff pw1.test pw2.test
1c1
< carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
---
> carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/sh
4c4
< pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
---
> pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/sh
```

输出中带小于符号的行出现在第 1 个文件中，而没有出现在第 2 个文件中。类似地，带大于符号的行出现在第 2 文件中而没有出现在第 1 文件中。

5.3.3 uniq

文件经过处理后在它的输出文件中可能出现重复的行。例如，使用 cut 命令后再用 sort 命

令就可能出现重复行。可以用 uniq 命令将这些重复行从输出文件中移去，只留下每条记录的唯一的样本：

```
$ cat pw.test | cut -d: -f7 | sort | uniq
/bin/bash
/bin/sh
```

这一管道行将 pw.test 文件的内容送给 cat 命令，由 cut 提取每个用户的 shell 程序名称即第 7 字段。然后通过管道送给 sort 命令。sort 命令将重复的行排在一起，通过管道送给 uniq。最后由 uniq 将它发现的重复行移去，产生最后的输出结果。

5.4 其他工具

还有一些可供使用的其他工具值得在这里提一下。其中之一是 pr 命令。它将文件分成适当大小的页送给打印机，并在每页中插入了标题。标题的内容包括日期，时间，文件名和页号：

```
$ pr pw.test

Jun 1 09:20 1995 pw.test page 1

root:awmku76tr43d6:0:0::/root/:/bin/sh
pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
```

可以用 -n 开关使 pr 命令在正文中加上行号：

```
$ pr -n pw.test

Jun 1 09:20 1995 pw.test page 1

    1  root:awmku76tr43d6:0:0::/root/:/bin/sh
    2  pc:bdhd74hs9jh3h:500:50::/usr1/pc:/bin/bash
    3  carey:esJ9ohd8HH89i:501:50::/usr1/carey:/bin/bash
    4  mot:dhjd83kjdJS6D:1500:60::/usr1/mot:/bin/bash
    5  grex:cj8AjoWE8h8fs:1500:60::/usr1/mot:/bin/sh
```

要讨论的下一条命令是 od。它允许你将文件看作一组字节来进行检查。它可以检查文本文件或数据文件的内容。每一行输出包括在文件内的当前字节位移值和一组 16 字节值。输出中的所有数据字节和位移值可以用不同方式显示。包括 ASCII 字符，八进制或十六进制数：

```
$ od /tmp/text/motd
0000000 064514 072556 020170 027061 027062 027061 024040 047520
0000020 044523 024530 005056
0000026
```

od 的默认显示方式是八进制数,这也是它的名称的由来 (Octal Dump)。但这不是最有用的显示方式。用 ASCII 和十六进制数组合的显示方式能提供更有价值的信息输出:

```
$ od -Ax -tcx1 /tmp/text/motd
000000  m  o  t  d     c  o  n  t  e  n  t  s     i  n
       6d 6f 74 64 20 63 6f 6e 74 65 6e 74 73 20 69 6e
000010     h  e  r  e  .  \n
       20 68 65 72 65 2e 0a
000017
```

od 命令的 -A 开关用来指定位移值的显示方式,包括:

x 十六进制;
o 八进制(默认值);
d 十进制;
n 不输出位移值。

od 命令的 -t 开关用来指定数据字节的显示方式,主要的选项有:

xn 十六进制;
on 八进制 (默认值是 o2);
dn 带符号位的十进制;
un 不带符号位的十进制;
c ASCII 字符或反斜杠序列。

其中 n 表示每个显示值包含几个字节。od 命令的其他选项请参阅手册页。

本节中最后一条命令称为 file。这一命令的用途是提供文件的类型信息。file 命令通过一组测试来确定文件的类型。当它认为已经确定时,停止测试并显示有关信息。由于它使用的一些测试的性质所决定,file 命令可能被愚弄。但这种情况很少,不成为主要问题:

```
$ file *
book.dvi:    TeX DVI file data
book.log:    Tex transcript text
book.ps:     PostScript document
book.tex:    English text
fdr:         ascii text
inc:         directory
```

5.5 DOS 文件

一组称为 mtools 的可移植工具允许不同的操作系统（包括 Linux）从标准的 DOS 磁盘上读、写文件和目录。它们对 DOS 和 Linux 环境之间交换文件非常有用。它们也是不具备共同的文件系统格式的系统之间交换文件的有力手段。

mtools 的主要命令如下：

mcd *dname*	改变 DOS 盘上的目录；
mcopy path *dname*	复制文件到 DOS 盘；
mcopy *dname path*	从 DOS 盘复制文件；
mdel *dname*	删除 DOS 文件；
mdir *dname*	显示 DOS 目录；
mformat *drive*	对 DOS 软盘进行低级格式化；
mlabel *drive*	改变 DOS 盘的卷标号；
mmd *dname*	建 DOS 盘目录；
mrd *dname*	删除 DOS 盘上的目录；
mren *dname dname*	重新命名 DOS 盘上的文件；
mtype *dname*	显示 DOS 文本文件的内容。

这些命令和对应的不加 m 的 DOS 命令非常相似。在表中，path 是任何标准的 Linux 文件路径名，drive 是 DOS 磁盘驱动器的字母后面跟以冒号，而 dname 则是 DOS 磁盘文件名。DOS 文件名由磁盘驱动器字母，冒号和到文件或目录的路径名组成。在这些路径名中使用的目录分隔符可以用 Linux 的斜杠字符（/），也可以用 DOS 的反斜杠字符（\）。如果使用反斜杠字符，需要加引号。为了看 DOS 盘最上层的目录的内容：

```
$ mdir a:
 Volume in drive A has no label
 Directory for A:/

BOOK            <DIR>      4-13-95  10:58a
FDR                 70     4-13-95  11:00a
FDW                 83     4-13-95  11:00a
DOT             <DIR>      6-05-95   6:01p
LTI      DOC       5732    5-02-95   9:02a
       8 File(s)     828928 bytes free
```

或者看同一盘的子目录内容：

```
$ mdir a:/dot
 Volume in drive A has no label
 Directory for A:/DOT

.               <DIR>      6-05-95   6:01p
..              <DIR>      6-05-95   6:01p
D10      TGZ       65317    6-05-95   6:02p
       3 File(s)     828928 bytes free
```

当在 DOS 盘上发现有关文件，将它从 DOS 盘复制到当前目录下就很方便，还可以验证复制是否实际完成：

```
$ mcopy a:/dot/d10.tgz.
Copying D10.TGZ
$ ls -l
-rw-r--r--   1 pc       book       65317 Jun  6 23:13 d10.tgz
```

练习

用单条命令或管道行完成下列各项功能：

1. 假定你作为普通用户访问系统，显示文件系统中所有的称为 core 的文件清单。
2. 假定你作为普通用户登录，显示系统中的所有进程的 ID 清单。
3. 假如你作为 root 登录，前面问题的答案有什么变化？
4. 在/dev 目录下显示名称的 hda 开头后面跟以数字的所有文件清单，以数字的升序显示清单(在我们的系统中，有 hda1 到 hda12 共 12 个文件)。
5. 在 /bin/cat 文件中找出所有以空白 (null) 字符结束的可打印字符串，并用十六进制显示每一字符串的开始字节的位移值。字符串长度不少于 3 个字节，需要仔细查阅手册页来找出问题的答案。

答案

1. 一种可能的解决办法用 find 命令找出根目录下所有文件的清单，然后搜索任何以 core 命名的文件并显示它们的路径名

   ```
   $ find  /  -name core  -priut  2>/dev/null
   ```

   不要忘记：作为普通用户你对许多目录没有访问权。这将导致 find 命令产生大量出错信息。需要将它们重新定向到 /dev/null，抑止它们在屏幕上显示。

2. 单独的 ps 命令只显示与你的终端联系在一起的所有进程清单。加上 -x 开关将使 ps 同时显示为你运行的，但不和你的终端联系在一起的任何进程。用 -h 开关抑止 ps 的标题行的输出。这以后，ps 输出的每一行的第一个字段包含进程 ID，总长 5 个字符。这些可以用下面的办法提取的显示：

   ```
   $ ps  -xh  |  cut  -b1-5
   ```

3. 作为 root 用户登录，ps -xh 命令将列出系统中所有的进程，不仅仅是属于 root 的进程。这使得只列出属于 root 的进程 ID 需要作进一步思考，但仍然能做到。一种办法是：

   ```
   $ ps  -uxh  |  grep  "^root"  |  cut  -b10-14
   ```

   其中 -u 开关使 ps 在每行开始列出进程所属的用户名。将结果经过管道送给 grep 提取以 root 开始的所有行。从这些行中提取和显示进程 ID 和上例相同。但注意 PID 的位置和上例不同。通过对 ps 命令输出的检查就能确定字段的确切位置。

4. 用合适的 ls 命令取得正确的文件名称清单，通过管道送给 sort 程序，并用 -n 开关指定用数字排序，另一个参数用来确定这些数字在行中的具体位置：

```
$ ls  /dev/hda[0-9]*  |  sort  -n  +0.8
```

注意:当将输出送给管道时,ls 命令自动产生每个文件占一行的输出。而不像送给屏幕显示时几个文件占一行。可以用两个命令来检验这一点:

```
$ ls
$ ls  |  cat
```

一般认为两个命令会产生同样的输出,实际却不是。

5. 查阅 od 命令的手册页。会发现它有一个 -s 开关,它可以用来找出所有以 null 结束的字符串。-Ax 开关使文件内的字节位移值按要求用十六进制数显示。最后通过管道用 cut 命令提取字节位移字段。

```
$ od  -s  -Ax  /bin/cat  |  cut  -b1-6
```

第六章　Shell Script

我们已经看到 bash shell 有一组可以从键盘使用的强有力的交互功能：输入/输出重新定向，作业控制，历史命令表以及命令行编辑等。这只是事情的一个方面。shell 还提供完整的程序语言解释程序以及读取和执行你自己的 shell 程序的手段。这些 shell 程序称为 Shell Script。

6.1　命令文件

任何标准的 Linux 命令或管道行，或相关的一组命令或管道行，都可以用 cat 命令或 vi 类编辑程序送到文本文件中去；并用 chmod 命令将这一文本文件变为可执行文件；然后像执行其他系统命令一样，送入文件名称来执行它。下面一组命令将有助于使这些概念具体化：

```
$ cat >dirsize
ls  /usr/bin | wc -w
Ctrl-d
$ chmod 700 dirsize
$ ls -l dirsize
-rwx------  1 pc       book         20 Jun 10 22:04 dirsize
$ dirsize
   459
```

其中，cat 命令将 ls /usr/bin | wc -w 管道行送入名为 dirsize 的文本文件中。然后用 chmod 700 dirsize 命令将这个文本文件标记为可执行文件。下一条 ls 命令用来检查是否按要求设置了文件的权限标志。最后用文件名 dirsize 作为命令名，由 shell 执行写入文件的管道行，就像直接在命令行中送入命令一样。输出的 459 是 shell script 中指定的目录的文件计数。

当你写经常使用的 shell script 时，应该将它们放在专门设置的目录下，并从那里执行它们。最好在起始目录下建立 bin 目录，将这些文件都放在那里。问题是如何在不需要送入路径名的情况下从当前工作目录执行这些命令。这个问题将在后面说明。

6.2　变量

像多数程序设计语言一样，shell 提供说明和使用变量的功能。对 shell 讲，变量只是内存中赋予名称的一些存储单元。它们保存一串字符作为它们的值。事实上，当看到 shell 提示符时，shell 已经定义了许多变量。并在 shell 开始执行时设置了这些变量的值。其中一些变量的名称和用途如下：

PS1　它保存 shell 命令行提示字符串。到目前为止的所有例子中，这一变量的值是'$ '。

PS2　它保存 shell 的第 2 提示字符串。当 shell 发现你的命令不全，还需要有更多输入时使用这

—提示符。在 4.2 节中讨论 here 文档时,用到了第 2 提示符。当时使用的变量值是'>'。

PWD　它保存当前工作目录的绝对路径名。由 cd 命令设置这一变量的值。

UID　它保存当前用户的用户识别号。以字符串形式保存变量的值,即使用户的 UID 是一个数也一样。

PATH　从键盘送入命令时,shell 需要知道到那里去找可执行文件。PATH 变量用来保存用冒号分隔的目录路径名清单。shell 将按 PATH 变量中给出的顺序搜索这些目录。找到的第 1 个与命令名称一致的可执行文件便将被执行。

HOME　当登录到系统中时,此变量将保存起始目录的完整的路径名。

用 echo(回送)命令查阅 shell 变量的内容是方便的。此命令在标准输出设备上显示送入的任何命令行参数:

```
$ echo three extra parameters
three extra parameters
$ echo UID
UID
```

很明显,用 echo 回送 shell 变量的名称(上例中的 UID)只显示变量的名称,而不显示它的内容。必须用其他方式来显示它的内容。

当希望从 shell 变量提取它的内容时,必须在变量名称前加上美元符($)。美元符是特殊字符,它的含义由 Sehll 解释。这也表示如果要将美元符作为参数传给另一条命令需要加引号。只有反斜杠字符和单引号对美元符起作用。参阅 4.1 节 shell 引用方法。UID 变量的值可以显示如下:

```
$ echo the value of UID is $UID
the value of UID is 500
```

echo 命令既能显示正文,也能显示 shell 变量的值。因此,它在 shell script 中非常有用。

可用下面的句法给 shell 变量赋新值:

```
variable=value
```

下面的命令改变了 PS1 变量的值,因此也改变了 shell 的主要提示符:

```
$ PS1='hello:'
hello: echo prompt is now $PS1
prompt is now hello:
```

注意:在实际改变变量值时,变量名称前不加美元符。只在提取变量值时才用美元符。同时注意:给变量赋值时,等号两边不能留空格。如果在这些位置留了空格,shell 将不认为这是变量赋值表达式,因而可能产生出错信息。

除标准的 shell 变量外,可以自由建立自己的变量并给它们赋值。如果试图对未赋值的变

量取值，shell 只送回一个空串的作为它的值，但不产生出错信息：

```
$ echo $sn flowers are pretty
flowers are pretty
$ sn = sun
$ echo $sn flowers are pretty
sun flowers are pretty
```

如果希望将上例中的输出变为 sunflowers are pretty，需要消除 su 变量的值和 flower 之间的空格，正像你从下面看到的一样，简单的办法行不通：

```
$ echo $snflowers are pretty
are pretty
```

简单去掉空格的做法使 shell 误认为 snflowers 是一个新变量，一个未赋值的新变量。正确的做法是将变量名称放在一对花括号中，将它和 flowers 隔开，但不必留空格。

```
$ echo ${sn}flowers are pretty
sunflowers are pretty
```

一般说来，可以对变量赋新值来改变变量已有的值：

```
$ sn = 'all kinds of '
$ echo ${sn}flowers are pretty
all kinds of flowers are pretty
```

在上面的例子中，赋予 su 变量的字符串中有空格，在两边加引号是必要的。一般情况下，shell 用空格作为分隔符将一行输入分成许多单独的字。本例中，需要将包含空格在内的一串字符赋值给 shell 变量，就要在两边加引号。

有时要说明一个变量，对它设为特定值，并希望以后不再改变它的值。要做到这一点只要在变量赋值后，将它设为只读（readonly）即可：

```
$ readonly sn
$ sn = sun
bash: sn: read-only variable
$ echo ${sn}flowers are pretty
all kinds of flowers are pretty
```

你已经看到设为只读的变量的另一个例子是 UID。它是标准的 shell 变量。当用户登录时，系统在口令文件的用户 UID 字段中设置它的值。这是一个不可以随意改变的值，将它设为只读是必要的。

在任何时候，建立的变量只是当前 shell 的局部变量。局部变量不被 shell 运行的其他命令

或 script 所利用。但在许多情况下,将这些值供 shell 执行的其他命令使用是必要的。这可以用 export 命令来实现。上例中,su 是局部变量。它可以用下面的方式使它可供其他命令使用:

```
$ export su
```

也可以在给变量赋值的同时使用 export 命令:

```
$ export su = sun
```

6.3 shell script 参数

前面写的 dirsize shell script 有一个主要问题,它只能给出已在 script 中指定的 /usr/bin 目录的文件计数。如果能在使用 dirsize 时指定需要的目录路径名那就更有用了。理想的情况应能像下面那样给出命令:

```
$ dirsize  /etc
```

使 dirsize 能对指定的任何目录进行文件计数。

这是容易做到的,因为 shell 将送入的命令行自动分成单字,然后将这组字作为值赋予一组特殊的变量。可以在 script 内部读取这些变量的值。这些变量分别用 $0, $1, $2 符号表示。$0 的值是命令行中第一个字 (即命令的名称)。$1 的值是命令行的第一个参数, $2 的值是第 2 个参数等。从 $1 起称为位置参数(positional parameters)。例如,按下列方式执行名为 testl 的 shell script:

```
$ testl paraml param2
```

那么在 shell script 内部这些特殊变量的值为:

```
$0 = testl
$1 = paraml
$2 = param2
```

有了这些概念后,按照要求改写 dirsize 就容易了:

```
$ cat >dirsize
ls $1 | wc -w
Ctrl-d
$ chmod 700 dirsize
$ dirsize /etc
    69
```

当这一 shell script 运行时,将用参数 /etc 取代 ls 命令中的 $1。由 shell 完成这一替换。ls 命令并不知道发生了替换。

为了能在目录层次结构中的任何位置都能使用这条命令,必须将包含这条命令的目录的路径名列在 PATH 变量的清单中。下面的例子在起始目录下建立 bin 目录,将 dirsize 移到该目录下,并将 bin 目录的路径名加到 PATH 变量的清单的前面。使 shell 在送入命令时自动搜索这一目录。本例中还使用了 HOME 变量的值,因为它包含了起始目录的路径名:

```
$ echo $PATH
/bin:/usr/bin:.
$ mkdir $HOME/bin
$ mv dirsize $HOME/bin
$ PATH=$HOME/bin:$PATH
$ echo $PATH
/usr1/pc/bin:/bin:/usr/bin:.
```

PATH 变量中的目录清单用冒号分开。PATH 变量的原始值表示它将在 3 个目录中搜索你送入的任何命令:先在 /bin 目录下,然后在 /usr/bin 目录下,最后在点即当前目录下进行搜索。下面的命令将 /usr1/pc/bin 目录加到 PATH 变量的值的前面。这样,当送入命令时,首先搜索这一目录。

当引用起始目录时,可用另外一个符号(代字符~)来代替 $HOME,使用这个符号,$HOME/bin 和~/bin 一样。还可以在代字符后面跟另一个用户的登录名,这时,此符号扩展为特定用户的起始目录的路径名,例如:

```
$ ls ~carey/bin
```

将列出 carey 用户的起始目录下的 bin 目录的内容。这里假定我有读这一目录的权限。

6.4 登录 script

通过修改 PS1 变量的值来改变 shell 提示符可能很有意思。在 PATH 变量中增加与个人有关的内容可能很有用。但是,如果必须在每次登录后去完成这些操作,不用过多久就会感到厌烦。

解决的办法是:建立个人的登录 script。在每次登录时,shell 将自动读取和执行这一 script。个人登录 script 也就是你登录时执行的 script。如果写了个人的登录 script,应该将它放在起始目录下,当登录时,bash 将自动搜索起始目录找 3 个特殊文件中的一个。这些文件是:

```
~/.bash_profile
~/.bash_login
~/.profile
```

bash 按上面的顺序找这些文件名称,并且只执行最先找到的一个。

当退出登录时,bash 也能执行个人的退出登录 script。需要用下列文件名称将它存在你的起始目录下。

```
~/.bash_logout
```

bash 除了执行个人 script 外,还允许系统管理员建立全局登录 script。它的名称是:

```
/etc/profile
```

在每个用户登录时, bash 执行这一 script。它允许系统管理员从系统整体出发设置用户的局部环境。如果全局 script 存在,它将在个人 script 之前执行。这样个人 script 将能进一步使局环境适合特殊要求。例如,在全局 script 中经常包括 ignoreeof,禁止用文件结束符 (ctrl-d) 作为退出登录的方法使用。如果它包含了 ignoreeof,可以在个人 script 中将它移去,这样就能用 ctrl-d 退出登录。

下例表明如何建立改变 shell 提示符和将 bin 目录加到 PATH 变量的清单中去的个人 script:

```
$ cat > ~/.profile
PS1 = 'What Now! '
PATH = ~/bin: $ PATH
Ctrl-d
```

6.5 更多变量的赋值方法

前面的例子表明:可以用一串字符对特定的变量赋值。还有两种情况需要对变量赋值。首先,可能需要 shell script 以交互方式从键盘取值,然后将值存在变量中。其次,可能希望执行一条命令,并将命令的标准输出送给变量而不是送到屏幕上显示。正如你预期的,这两种赋值方式都是可行的。

6.5.1 交互输入

shell script 可以用 read 命令从标准输入设备读入若干行字符,并将它赋值给一个或多个变量。read 命令的最简单形式是只有命令名称,不指定变量。这种情况下,从标准输入设备读入的字符串赋值给 REPLY 变量。如果用下面的正文写成一个名称为 readtest1 的 shell script:

```
echo -n 'Enter some text: '
read
echo The text was: $ REPLY
```

然后运行它,你将得到下面的结果:

```
$ readtest1
Enter some text: THIS IS THE TEXT I ENTERED!!
The text was: THIS IS THE TEXT I ENTERED!!
```

通常用到 echo 命令时,它在行的结束处自动增加新行字符。所以 ,以后的输入和回送都从下一行开始。本例中用 -n 开关抑制新行字符的输出。允许在 shell script 内部给出提示的字符串,接着在同一行的后面输入正文。这里是响应 read 命令输入的正文和回送的结果。

如果在 read 命令名称后面指定变量名,输入行将直接送给变量而不送给 REPLY。

当给 read 命令指定多个变量时就更有意思了,考虑用下面的正文写成的 readtest2 shell script:

```
echo -n 'Enter some text: '
read one two restofline
echo The first word was: $one
echo The second word was: $two
echo The rest of the line was: $restofline
```

当执行这一 script 时,它和上例一样读入一行,然而,这一次它将输入行分成单独的字。分别赋予 read 命令中的 3 个变量。将第 1 个字赋予第 1 个变量,将第 2 个字赋予第 2 变量等。直到用完字或变量或两者都用完为止。

如果同时用完变量和字,正好每个变量都赋了值:

```
$ readtest2
Enter some text: 1 2 3
The first word was: 1
The second word was: 2
The rest of the line was: 3
```

如果先用完字,后面的变量就设有从输入中得到它的值,因而设为空值(null):

```
$ readtest2
Enter some text: 1 2
The first word was: 1
The second word was: 2
The rest of the line was:
```

如果字数多于变量数,每个变量都得到相应的值。不是将剩余的字抛弃而是将它附加在最后一个变量后面:

```
$ readtest2
Enter some text: 1 2 3 4
The first word was: 1
```

```
The second word was: 2
The rest of the line was: 3 4
```

read 命令的字定义范围很宽。默认的字分隔符是空格,Tab 和新行字符。这些分隔符存在称为 IFS 的 shell 变量中。如果需要,可以将这些分隔符改为其他字符。

6.5.2 命令替换

命令替换允许将管道行的输出赋值给变量。下面的例子有助于说明这一点:

```
$ date
Wed Jun 14 22:50:52 BST 1995
$ datestore = 'date'
$ echo $datestore
Wed  Jun  14 22:51:11 BST 1995
```

本例中第一条 date 命令用来显示它的输出格式,第二条命令放在引号中,使得 date 命令的标准输出直接赋值给变量 datestore。

另一种标记方式,将命令放在一对括号内,再在前面加上美元符,也能起到同样的效果:

```
$ datestore = $(date)
$ echo $datestore
Wed Jun 14 22:53:21 BST 1995
```

另一条常用作命令替换的命令称为 basename。这条命令用完整的文件路径名作为参数,它将路径名去掉只送回基本文件名,例如:

```
$ basefile = 'basename /usr/bin/man'
$ echo $basefile
man
```

6.6 自动化状态变量

执行任何 Linux 命令时,可能出现两种状态:它可能无错误地顺利结束运行,也可能遇到某些情况使它不能完成任务。在 shell script 中执行命令时,知道究竟发生了那一种情况是很有用的。如果出现错误就可以采取补救措施。所有的 Linux 命令都作了这样的安排。在它们运行结束时返回一个值,这个值称为它的退出状态(exit status)。退出状态为 0 表示命令的执行是成功的,任何非 0 值表示命令执行不成功。对管道行讲,返回的退出状态只是管道行中最后一条命令(最右边的命令)的退出状态。

任何时候当你运行一条命令或管道行时,shell 都用 $? 变量保存返回的状态值:

```
$ cd /zz
bash: /zz: No such file or directory
$ echo $?
1
$ echo $?
0
```

本例中试图用 cd 命令改变目录失败了,因为不存在 /zz 目录。接着显示 $? 的值表明 cd 的退出状态不是 0。注意第 2 次显示 $? 得到不同的结果。这是因为每条命令或管道行执行后这一变量的值都要作相应的修改。第 2 次显示结果表示第一条 echo 命令执行是成功的。

shell script 本身可以用 exit 命令返回它的退出状态。你应该养成习惯,使 script 返回明确和合适的退出状态值:

```
$ cat > tryexit
exit 55
Ctrl-d
$ chmod 700 tryexit
$ tryexit
$ echo $?
55
```

$? 只是 bash 使用的许多特殊状态变量中的一个。这些状态变量都可以在 shell script 内部使用。主要的状态变量有:

$? 用来保存 shell 最后执行的命令的退出状态。

$$ 用来保存当前 shell 的 PID。在 shell script 内部用它的值来形成唯一的临时文件名(例如 /tmp/tmp$$)。

$# 它保存传送给当前 script 的命令参数的数目。它对应于当前使用的变量 $1, $2 等的数目。

$* 它保存传送给当前 script 的所有命令参数($1, $2 等)清单。

只允许读取这些特殊状态变量的值,不允许对它们赋值。只有 shell 才能自动更新它们的值。

6.7 流程控制

像其他高级程序设计语言一样,shell 提供用来控制程序执行流程的命令,包括一些选择和循环结构。可以用这些命令建立非常复杂的程序。

你将看到这些结构的不寻常的地方在于:它们用命令和字符串来指定条件值。在传统的语言中,用的是布尔表达式。

6.7.1 if 命令

当 shell 执行命令或管道行时,返回退出状态值。可以用一种结构来测试返回值并决定下

一步做什么。if 命令完成这一功能。它的一般格式如下：

```
if condition_command
then
    true_commands
else
    false_commands
fi
```

其中先执行条件命令(condition_command),它的退出状态用来决定执行条件为真时执行的命令 (true_commands) 还是条件为假时执行的命令 (false_commands)。由于退出状态值为 0 表示成功地执行了命令,因此 0 作条件的真值。任何非 0 值作为条件的假值。

else 子句和 false_comamand 是选项。如果不用这些选项,if 命令将成为：

```
if condifion_command
then
  true_commands
fi
```

下面是使用 if 命令的 script 的例子,我称它为 user1。它用一个字作为参数,检查口令文件,以证实这一参数是否是系统中有效的用户登录名：

```
# usage is: user1 login_name
if grep "^$1:" /etc/passwd >/dev/null 2>/dev/null
then
    echo $1 is a valid login name
else
    echo $1 is not a valid login name
fi
exit 0
```

上例中的条件测试用 grep 命令完成。因为要的只是它的退出状态,所以,将它的标准输出和错误输出统统抛弃。如果 grep 命令成功地找到这个字,返回值为 0 (真),否则返回值为非 0 (假)。相应的信息将送到标准输出设备：

```
$ user1 bash
bash is not a valid login name
$ user1 mot
mot is a valid login name
```

user1 第一次运行很不错,因为 bash 一字确实出现在口令文件中,只是没有出现在行的开头和后面跟以冒号 (按表达式 '$1:' 要求)。但是这并不是说 user1 不能被愚弄,下面的例子

说明这一点：

```
$ user1 carey:esJ9ohd8HH89i
carey:esj9ohd8HH89i is a valid login name
```

user1 程序表明，在 shell script 内部可以用 # 号加入注释。在 # 号后面，注释可以一直延伸到行的末尾。

6.7.2 test 命令

如果有一条标准命令能做你要做的事，那就执行这条命令，然后测试退出状态看它的执行是否成功。如果不成功，需要采取补救措施。这些都是容易做到的。但仍然留下许多条件需要进行测试，这需要用另一个命令来完成这些功能。这条命令称为 test，它既不接受任何输入也不产生任何输出。它的用途是测试一系列条件然后返回相应的退出状态。test 命令的格式是：

```
test  expression
```

如果指定的表达式（expression）为真，返回 0 退出状态；如果是假，返回 1 退出状态。

有一点值得在这里提一下，当写简单的 Script 或程序来检查你的想法时，很容易将文件的名称称为 test。当执行程序时，实际执行的可能是 Linux 的 test 命令，而不是自己的 test 文件（这取决于 PATH 变量中给出的目录顺序）。执行不带参数的 test 命令不做什么事情，只送回下一个 shell 提示符。给你的印象是 test 程序没有工作。这一讨论同样适合于其他 Linux 命令。但是，test 命令却是最容易搞错的一个。

test 命令可以用来测试几种不同类型的表达式，最常用的表达式类型有：检验文件的特征，对不同的字符串进行比较以及完成数值比较。

在第一种情况下，给出一个命令行开关，后面跟单个参数，即文件名称。tset 命令根据开关指定的特征对文件进行检验，并根据检验结果给出退出状态。最常用的命令行开关有：

```
-e   f   如果 f 文件存在，返回真值；
-f   f   如果 f 文件是普通文件，返回真值；
-d   f   如果 f 文件是目录，返回真值；
-r   f   如果你能读 f 文件，返回真值；
-w   f   如果你能写 f 文件，返回真值；
-x   f   如果你能执行 f 文件，返回真值。
```

字符串类型表达式可以用一个或两个字符串参数，它既可以是文字字符串也可以是 shell 变量的内容。最常用的串表达式有：

```
-z   str      如 str 的长度为 0，返回真值；
-n   str      如 str 的长度不为 0，返回真值；
str1 = str2   如 str1 和 str2 相同，返回真值；
str1 != str2  如 str1 和 str2 不相同，返回真值。
```

数值表达式具有将字符串或变量的内容作为数值来处理的能力,并完成标准的数值比较。数值表达式见下表。

num1 -eq *num2*	如果 *num1* 等于 *num2* 为真;
num1 -ne *num2*	如果 *num1* 不等于 *num2* 为真;
num1 -lt *num2*	如果 *num1* 小于 *num2* 为真;
num1 -gf *num2*	如果 *mum1* 大于 *num2* 为真;
num1 -le *num2*	如果 *num1* 小于或等于 *num2* 为真;
num1 -ge *num2*	如果 *num1* 大于或等于 *num2* 为真。

test 命令还可以使用逻辑操作符 AND,OR 或 NOT 组合表达式:

exp1 -a *exp2*	如果表达式 *exp1* 和 *exp2* 都为真时取真值;
exp1 -o *exp2*	如果表达式 *exp1* 或 *exp2* 之一为真时取真值;
! *exp*	如果 *exp* 为真取假值,反过来也一样。

像 test 命令能将字符串作为数值比较一样,bash shell 也能完成简单的算术运算,并用下面的方式建立算术表达式:

```
$[expression]
```

下例表明这些概念的具体运用:

```
$ num1 = 2
$ num1 = $[ $num1 * 3 + 1]
$   echo $num1
7
```

下面是称为 later 的简单的 shell script,它用 test 命令进行简单比较并利用 shell 算术表达式进行计算。当给出从现在算起的小时数,later 将计算和显示几小时以后的时间:

```
if test $ #  -ne 1              # Check for 'hours' parameter
then
    echo "usage: later <hours>"
    exit 1
fi
now = 'date'                    # Get and split current time
hour = 'echo $now | cut -b 11 - 12'
minsec = 'echo $now | cut -b 13 - 18'
hour = $[( $hour + $1) % 24]    # Add 'hours' to current hour
if test $hour -ge 12a           # Convert to 12 hour clock
```

```
then                                #       and print later time
    echo $[ $hour - 12] $minsec pm
else
    echo $hour $minsec am
fi
exit 0
```

开始,它检查命令行中是否指定了一个参数,它用检查 $ # 的值来做到这一点。如果命令行参数数目不等于 1,输出命令的使用格式,并结束运行。

下一步用 date 命令取得当前的日期和时间,从中提取 24 小时格式的时间,并将小时和分秒分别存入 hour 和 minsec 两个变量中。

接着它将命令中给出的小时数和当前的小时数相加。如果相加的时数超过 24,用模 24 进行处理,其中%号是模运算符。

最后,将 24 小时格式转换为 12 小时格式显示,并在后面相应加上 am 或 pm。

执行 later 将得到下面的结果:

```
$ later 5
8:32:42 pm
$ later 12
3:32:49 am
```

6.7.3 while 命令

while 命令允许你在 shell scrept 中插入条件循环,命令的一般格式为:

```
while condition
do
   commands
done
```

其中 condition 可以是任何命令或管道行,它的退出状态将用来决定下一步操作。如果退出状态为真(0),就执行 do 和 done 之间的各条命令。接着转移到循环顶部,重新检查条件。如果退出状态为假(非 0),则跳过这些命令,执行关键字 done 后面的命令。

一个简单的例子有助于说明这一概念。下面的 shell script 称为 filecheck,将反复测试指定的文件是否继续存在。只要文件被删除,循环也就结束,退出 script:

```
while test -e $1
do
    :
done
exit 0
```

do 和 done 之间的冒号用来指定不做任何事情的空命令。在上面的循环内不进行任何操作，如果没有冒号，bash 将产生出错信息，因为它不允许有空的循环体。

一般情况下，会把这样的 script 放到后台去执行。否则，在文件被删除以前将得不到 shell 提示符。可能是长时间的等待！

将这一 script 放到后台去执行，将消耗大量的处理机时间，因为它不停地反复测试指定的文件是否存在。对这一具体应用讲，也许隔几秒钟检查一次更合适，没有必要尽快得到结果。这样，冒号命令就可以用 sleep 命令代替，在指定的时间（秒数）内将 script 挂起。在此期间，它将不给处理机增加负担，使其他进程能充分利用处理机时间。

```
while test -e $1
do
    sleep 2
done
echo file $1 does not exist ...
exit 0
```

改写的 filecheck 还增加了 echo 命令。当指定的文件不存在时，你将得到一条信息。下面一组命令建立文件，在后台运行 filecheck 来检查这一文件。然后删除文件，使文件不再存在的信息的屏幕上显示：

```
$ cat > lock
Ctrl-d
$ filecheck lock &
[1] 2364
$ rm lock
file lock does not exist...
```

上例中，cat 命令只用来建立空文件，可以用其他办法做到这一点。利用 touch 命令是最简单的办法。它的主要用途是将文件的时间标记（timestamp）改为当前时间，作为附带的效果是：如果文件不存在，它将先建立空文件，然后更改它的时间标记：

```
$ ls -l lock
ls: lock: No such file or directory
$ touch lock
$ ls -l lock
-rw r   r--  1 pc    book      0 Jul  3 21:52 lock
```

6.7.4 until 命令

until 命令是另一种循环结构。它和 while 命非常相似，并具有下面的一般结构：

```
until condition
do
    commands
done
```

它和 while 命令的差别在于:while 命令在条件为真时,继续执行循环。而 until 则相反,在条件为假时,继续执行循环。

假如,要写一个 script,在它结束之前等待特定文件的出现,用 until 命令就很合适:

```
until test -e $1
do
    sleep 2
done
echo file $1 now exists...
exit 0
```

有时需要建立不定时的循环。需要一条命令经常返回 0 退出状态(建立不定时的 while 循环)或非 0 退出状态(建立不定时的 until 循环)。这两条命令分别称为 true 和 false。例如:

```
until false
do
    read firstword restofline
    if test $firstword = end
    then
        exit 0
    else
        echo $firstword $restofline
    fi
done
```

这一 shell script 读入一行正文并回送到显示设备,一直进行到在一行的开始读到 end 为止。

6.7.5 for 循环

if,until 和 while 命令都用退出状态值来控制它们的操作。有些情况下需要用字符串的值,而不是退出状态值来控制循环和选择。bash 同样满足这方面的需要。

设想一下,要写一个 shell script,用来找出一组登录名是否出现在口令文件中。其实,在 6.7.1节中看到的 script 已能满足部分要求。这个称为 user1 的 script 只接受一个用户名(在 $1 中),并使用 grep 命令搜索口令文件来确定他是否在系统中。这里要做的只是对一组用户名称($1,$2 等)重复使用 user1 就可以了。

可以用 for 命令来做到这一点。命令的一般格式是:

```
for variable in wordlist
do
    commands
done
```

基本概念是:从 wondlist 清单中依次取指定变量 variable 的每个值,然后用取得的值执行循环体内的命令,例如:

```
for i in 1 2 3 4 5
do
  echo value of i is $i
done
```

它的执行结果如下:

```
value of i is 1
value of i is 2
value of i is 3
value of i is 4
value of i is 5
```

现在回到原来的问题。注意特殊变量 $ * 给出传送给 shell script 的所有位置参数的清单。所以上面问题解答就变得容易了。一种可能的解决办法(称它为 user2)是:

```
for i in $ *
do
    if grep "^$i:" /etc/passwd >/dev/null 2>/dev/null
    then
        echo $i is a valid login name
    else
        echo $i is not a valid login name
    fi
done
exit 0
```

运行这一 script 得到下面的结果:

```
$ user2 carey jill rnc bill
carey is a valid login name
jill is not a valid login name
rnc is a valid login name
bill is not a valid login name
```

'for variable in $ * '是经常用到的结构,它可以被缩写为' for variable' 。由命令自动补充 in $ * 的部分。

6.7.6 case选择

根据字符串或变量的值,从许多选项中选出一项,这是 case 命令要完成的任务。命令的一般形式为:

```
case string in
expression 1)
    commands_1
    ;;
expression_2)
    commands_2
    ;;
.
.
.
*)
    default_commands
    ;;
esac
```

在执行 case 命令时,shell 将计算字符串 string 的值,然后将结果依次和表达式 expression_1 expression_2 等进行比较,直到找到一个匹配的表达式为止。如果找到了匹配项,将执行它下面的有关命令直到遇到一对分号(; ;)为止。

case 表达式也可以用简单的规则表达式给出,允许使用 shell 用于扩展文件名的特殊字符(即:*,? 及[ ]),用 * 号作为 case 命令的最后表达式的含义是:它可以和任何字符串匹配。如果前面的比较全部失败,这就是默认的入口。

下面是使用 case 命令的 script 的例子。称它为 append。它从键盘或输入文件读入正文并将它附加到指定的输出文件的后面:

```
case $ # in
1)
    cat > > $1
    ;;
2)
    cat > > $1 < $2
    ;;
*)
    echo "usage: append out_file [in file]"
    ;;
esac
```

```
exit 0
```

注意，$ # 变量用来保存命令行中位置参数的数目，不包括命令名称。对 append script 讲，取决于输入正文的来源，它的值只能是 1 或 2。

在使用方法说明中，方括号内的项是选项。在 Linux 命令文档中，经常用方括号表示选项。

6.8 信号

你已经知道：可以从键盘按 ctrl-c 来中断大多数正在运行的程序的执行。

实际情况是控制键盘的 Linux 内核软件向运行中的进程发送了信息。这一信息称为信号（signal）。在 2.6 节中第一次讲到 kill 命令时已经用到了信号的概念。实际上，约有 30 种不同的信号可以发送给 Linux 进程。信号是由特定的硬件和软件条件引起的。除了正常的产生和发送信号的方式外，kill 命令也可以发送信息给进程。

每一种信号用一个小整数表示，并将它作为 kill 命令的开关。kill 命令的格式是：

```
kill [ -sig] pid
```

其中 sig 是表示具体信号的整数，pid 是接受信号的进程识别号。方括号表示 sig 是选项。如果不给 sig 的值。kill 就发送默认的结束信号（terminate）。

一些常用的信号和对应的数值如下：

信号	数值	说明
hangup	1	当退出登录时，用来结束你的进程的信号；
interrupt	2	从键盘产生的中断信号（ctrl-c）；
quit	3	从键盘产生的退出信息（ctrl-\）；
kill	9	不能忽视的强制结束进程信号；
alarm	14	在 alarm（）系统调用结束时产生的警告信号；
terminate	15	kill 命令的默认结束信号。

当进程接到一个信号时，它可以用三种方式作出反应：

1. 忽视这一信号；
2. 接受信号的默认操作；
3. 执行一段程序来处理这一信号。

大多数信号的默认操作是结束接受信号的进程的运行（这就是为什么从键盘按 ctrl-c 能结束进程运行的原因）。在改变默认操作时，进程需要执行一段预先准备的程序。但是，对 kill 信号讲，这一默认操作不能改变。接受 kill 信号的进程将结束它的运行。

在 shell script 内部捕捉信号是可能的。当收到信号后执行一组命令。这可以用 trap 命令来完成，trap 命令有三种基本形式，分别对应三种不同的信号响应方式：

```
trap "commands" signal_list
```

第一种形式的 trap 命令在 script 接到与 signal list 清单中数值相同的信号时，执行双引号中的命令。

为了恢复信号的默认操作，使用第二种形式的 trap 命令：

```
trap signal_list
```

其中不指定任何命令。

第三种形式允许忽视信号：

```
trap "" signal_list
```

其中只指定空命令串。

使用 trap 命令的典型例子是：在程序非正常结束时清除临时文件。否则，这些文件将白白占用你的文件空间。下面的程序段建立名称唯一的临时文件/tmp/tmp $ $ 供 script 内部使用。如果发生非正常结束 script 的信号（接到中断信号或退出信号），则在结束 script 运行前删除临时文件：

```
trap "rm -f /tmp/tmp$$; exit 0" 2 3
touch /tmp/tmp$$
#
# Rest of shell script here
#
trap 2 3
```

这一 script 在一开始就准备捕捉信号 2 和 3（从键盘发出的 ctrl-c 和 ctrl-\）。如果发生其中一种信号（通常是用户试备中断或停止 script 的运行）。在 trap 命令执行后，信号 2 或 3 将导致下列命令的执行：

```
rm -f /tmp/fmp$$;exit 0
```

rm 命令将删除临时文件，而 exit 命令将结束 script 的运行。注意，如果用分号隔开，同一行中可以加入多条命令。

在正常情况下，这一 script 将用 touch 命令建立临时文件，接着执行 script 的其余命令（rest of shell script）。最后一行用 trap 命令恢复信号 2 和 3 的默认操作，导致临时文件的删除。

练习

写出完成下列各项操作的 shell script：

1. 取两个目录名称作参数，将第一个目录中的所有文件复制到第二个目录中去。在复制过程中，将任何文件中出现的所有字符串 SP 变换为 SU。
2. 检查 PATH 变量是否包含 /usr/local/bin 目录作为搜索目录，并显示适当的信息。
3. 从标准输入设备读入正文行，并将它复制到标准输出。用类似 cat -n 命令的方式（但不用 cat 命

令）在每一输出行前面加一行号。

4. 接受2～15范围内的一个数作为参数。在标准输出设备上用 + 号，- 号和竖杠（|）画出以给定的参数值为边长的正方形：

```
$ drawsquare 4
+ - - +
|     |
|     |
+ - - +
```

如果命令行中参数数目不对或数值超出范围，显示相应的错误信息。

5. 取两个命令行参数，第一个是目录名，第二个是以字节计的文件容量。这一命令应列出给定目录中具有读权限，而且，容量小于给定容量的所有普通文件名。应检查命令行中只有两个参数，而且第一个参数是目录名。

答案

1. 一种解决办法是依次取每个源路径名，用 basename 从路径名提取文件名，然后用 sed 命令用同一文件名将源文件复制到指定的目录下，同时完成可需的转换：

```
for i in $1/*
do
    x='basename $i'
    sed s/SP/SU/ $i > $2/$x
done
exit 0
```

2. 用 grep 命令检查 PATH 变量中是否包括/usr/local/bin 目录，并显示相应的信息是容易的：

```
if echo $PATH | grep "/usr/local/bin" &>/dev/null
then
    echo PATH contains /usr/local/bin
else
    echo PATH does not contain /usr/local/bin
fi
exit 0
```

注意 grep 命令的标准输出和错误输出都已抛弃，因为这里只有 grep 命令的退出状态是重要的。

3. 解决这一问题的最简单办法是：利用 shell 对数值表达式的计算能力，在 script 中加行计数变量。将它的值放在每行输出的前面，回送到标准输出。然后将行计数值加 1，供下一行使用：

```
linecount=1
while read line
do
    echo $linecount $line
```

```
    linecount = $[ $linecount + 1]
done
exit 0
```

4. 开始,先检查命令行参数的数目是否有错,然后程序从左至右,自上而下沿着输出正方形的字符位置用简单的逻辑将4种字符（包括正方形内部的空格）,放在合适的位置上进行显示:

```
if test $# -ne "1"
then
    echo 'usage: drawsquare <n>'
    exit 1
fi
if test $1 -lt "2" -o $1 -gt "15"
then
    echo 'usage: drawsquare <n>'
    echo '       (where 2<=n<=15)'
    exit 1
fi
xcount=$1
ycount=$1
while test $ycount -gt "0"
do
    while test $xcount -gt "0"
    do
        if test $xcount -eq "1" -o $xcount -eq $1
        then
            if test $ycount -eq "1" -o $ycount -eq $1
            then
                echo -n "+"
            else
                echo -n "|"
            fi
        else
            if test $ycount -eq "1" -o $ycount -eq $1
            then
                echo -n "-"
            else
                echo -n " "
            fi
        fi
        xcount=$[ $xcount-1]
    done
    xcount=$1
    ycount=$[ $ycount-1]
```

```
    echo
done
exit 0
```

5. 对指定目录中每个文件进行检查，看它是否可读和是否是一个普通文件。如果检查结果为真，进一步检查文件容量是否小于第二个参数规定的容量。如果小于规定容量，在标准输出设备上显示文件名称和它的实际容量：

```
if test $ # -ne 2
then
    echo 'usage: listfiles <dirpath> <size>'
    exit 1
fi
if ! text -d $1
then
    echo 'usage: listfiles <dirpath> <size>'
    exit 1
fi
for i in $1/*
do
    if test -r $i -a -f $i
    then
        size='wc -c < $i'
        if test $size -lt $2
        then
            echo 'basename $i' has size $size bytes
        fi
    fi
done
```

第二篇

管　　理

第七章　准备和运行

为了取得最佳的效果，应该以普通用户的身份仔细阅读资料。在有了大量实践以后，再逐步进入 Linux 系统管理。当真正开始你的管理旅程时，最好在周围已经有人在这条路上走在前头，可以为你的开始阶段指引方向。或者，至少有一组计算机爱好者和你共同走完这段旅程。

你们中间许多人会遇到这种理想的情况。但是，仍然会有许多人缺乏必要的经验和帮助。如果你只有有限的经验，而且只是从头开始摸索前进时，这将是艰难而又容易出错的过程。开始阶段进展并不顺利；或者，当你准备解决某个具体问题，反复阅读资料后仍然找不出丝毫头绪。处在这种情况下，最重要的是不要气馁，不要灰心和失望，好好休息一下，从头开始。

如果有使用 DOS 的经验，又在个人计算机上运行 Linux。必须充分意识到：虽然它在小机器上运行，不管从表面去看它的外观如何，它毕竟是大机器上的操作系统。一个真正的多用户多任务操作系统。它比 DOS 复杂得多，功能也强大得多。如果想充分发挥 Linux 的功能，或者想避免由于疏忽而造成意想不到的困难，就必须通过学习逐步养成使用大机器操作系统的习惯和心理素质。

7.1　硬件

为了在 IBM PC 或其兼容机上运行 Linux，机器必须具有最小的硬件配置。但是，对硬件的要求并不高。唯一的实际要求是：至少是一台 386，内存容量不低于 4Mb。

事实上，Linux 可以使用任何 386 或 386 以上的机器。包括由先进的 CPU 生产厂家生产的各种型号。这些厂家包括 AMD，Cyrix 和 Intel 等。如果 CPU 包含浮点处理器，Linux 将用它进行浮点运算。如果 CPU 不包含浮点处理器，内核配置将包含浮点处理器模拟软件，用它来代替浮点处理器进行浮点运算。

Linux 内核可以用来驱动一些母板。这些母板可以使用任何下列标准总线：包括 ISA，EISA，VESA 局部总线（VLB）和 PCI。对单用户机器讲，母板至少应装 4Mb 内存。对配置 X-Window 的单用户机至少应装 8Mb 内存。对联网和提供多用户登录服务的机器讲，至少应装 16Mb 以上的内存。除了插在母板上的内存外，Linux 还将硬盘的一部分作为扩充内存。这种在硬盘上的虚内存，传统上称为交换空间（Swap Space）。

Linux 支持大量外部设备接口卡。其中包括：光盘，磁盘，网络，图形，调制解调器，鼠标和音响设备接口卡。它们的数量还在不断增长之中。如果在这里列出 Linux 支持的接口卡清单，你看到它们时可能已经过时了。Linux 提供了称为 Linux HOWTO 的文档。这些文档提供了 Linux 系统配置的硬件和软件的各个不同方面的专门信息。和上面内容直接有关的是 hardware HOWTO 文档。它的最近的版本提供 Linux 支持的所有硬件的最新清单。虽然系统支持的大多数接口卡的驱动程序和内核源程序一起提供，这一文档也告诉你如何取得特殊接口卡的驱动程序的有关信息。

从总的看，Linux 支持的接口卡清单几乎是无所不包的。在磁盘驱动器的栏目下，Linux 支持所有流行的磁盘控制器。包括：MFM，RLL，IDE，大多数 ESDI 和许多 SCSI 等。任何特定安装

所需的磁盘容量，在很大程度上，与使用的具体的 Linux 发布以及安装时选择的软件包有关。总的讲，最小的安装大约需要 20Mb 的磁盘空间。如果想增加大的应用软件，更完整的开发工具，系统文档和网络功能，大约需要 50 到 70Mb 的硬盘空间。如果要使用 X-Window，TeX 和 LaTeX 以及更多的程序语言和工具，硬盘空间的要求将进下一步增加。大致需要 150Mb，甚至更大。除此之外，还需要用于虚内存的硬盘空间，以及分配给系统其他用户作起始目录和工作存储的硬盘空间。

大多数个人计算机都用 SVGA 显示卡和监视器。如果只需要在屏幕上显示正文，Linux 支持任何 SVGA 以及普通的 VGA，EGA，不同的 CGA 和单色卡。如果需要通过 X-Window 和 SVGA 库显示图形，选择会受到一定的限止。但仍然包括：Tseng，Western Digital，Trident，ATI，Cirrus Logic 及其他公司生产的标准 SVGA 芯片组。不支持另一些 SVGA 卡的原因是：它们的生产厂家不提供如何驱动这些接口卡的细节。就是说，无法写这些卡的驱动程序。如果希望得到 Linux 支持的图形接口卡的最新的清单，可参阅另一个 HOWTO 文档，XFree 86 HOWTO。

设置 Linux 机器使它能在以太网上运行是直接了当的。因为内核提供大多数价格不高的网卡的驱动程序。包括：3com，Novell，Western Digital，Hewlett Packard 公司的产品及其兼容产品。如果需要得到 Linux 支持的网卡的最新清单，可以参阅 Ethernet HOWTO 文档。

对光盘存储器讲，Linux 支持标准的 ISO-9660 文件系统。许多光盘驱动器使用 IDE 或 SCSI 接口。只要有一块支持 IDE 或 SCSI 的接口卡，光盘正常工作是不成问题的。此外，Linux 还支持若干使用专用接口卡的光盘驱动器。其中包括：Mitsumi，Panasonic，Sony 和 Philips 驱动器。最新的清单可查阅 CD-ROM HOWTO 文档。

最后，Linux 还支持整系列的并行打印机，串行调制解调器以及串行和总线鼠标。

7.2 各种 Linux 发布

发布 Linux 不是某个个人或某个组织的责任。Linux 是自由提供的软件，任何人都能收集它的组成部分。然后，遵照自由软件基金会发表的 GNU 通用许可证条款自由发布这些软件。通用公共许可证的条款见附录。作为主要的 Linux 发布，维护和发布任务是困难和耗费时间的。只有具有高度献身精神的和热情的 Linux 支持者才能承担起这样的责任。即使这样，仍然有许多 Linux 发布可供选择。

较小的发布可以放在 4－5 张软盘上，较大的发布需要用上百张软盘。其余的可能介于两者之间。很明显，较小的发布只包含少量的软件。而较大的发布几乎无所不包。包含多数大型软件包：如 X-windows，emacs，ghostscript，TeX 和 LaTeX。

选用那一种发布纯粹出于个人的爱好，部分也受到手头可用的发布的限制。如果你有 Internet 访问权，情况就好多了。Linux 首先是在 Internet 上作为系统发布的，现在它仍然是取得 Linux 发布和问题答案的最方便的途径。USENET 新闻是最好的交互信息的来源，它类似于全球的电子公告牌系统，专为 Linux 有关专题提供若干新闻组。

此外，通过邮购取得 Linux 发布的软盘或光盘已成为流行的来源。许多杂志和刊物现在在登载这些产品的广告。许多计算机商店或市场也在销售 Linux 发布的软盘和光盘。

有关不同的 Linux 发布的最新的信息来源可参阅 Distribution HOWTO 文档。

一个特别流行的和完整的，也是到处都有的 Linux 发布称为 Slackware 发布。这一发布以软盘组的形式提供。每个盘组包含许多软件包。下面是当前可供使用的一些盘组：

A　不管你选择什么盘组的组合，这是 slackware 系统必须安装的基本盘组。

AP　包含许多不需要 X-Windows 即能运行的应用程序和软件包。其中包括许多手册页，拼写检查程序，其他的编辑程序，ghostscript 以及光盘驱动器的唱片播放程序。

D　程序开发用软件包。包括：C/C++编译程序（分别称为 gcc 和 g++）词法和语法分析程序（BSD 的 byacc 以及 GNU 的 bison 和 flex）；源程序查错程序（GNU 的 gdb）。C 语言库包括：SVGAlib 图形库以及 ncurses 终端控制库。其他程序设计语言的编译程序包括：lisp(clisp)；fortran(f2c)；passcal(p2c)及 perl。还有许多程序员的手册页。

E　包含 GNU 的字处理程序 emacs。包括不需要 X-Window 支持的和需要 X-Window 支持的两种版本。

F　包含所有的 HOWTO 文档以及常问的问题（FAQ）文档。

K　这一盘组包含 Linux 内核源程序（在早期的 Linux 发布中，它包含在 D 组中）。

I　包含许多 GNU 程序开发软件包的用户和系统手册。

IV　包含 Interviews X-Windows 库，其中包括应用样本 doc 和 idraw。

N　包含连网软件包（TCP/IP，ppp，UUCP）；邮件软件包（mailx，deliver，elm，pine 以及 sendmail）和网络新闻软件包（cnews，tin，trn 以及 nn）。

OOP　面向对象的程序设计。包含 Smalltalk 以及 smalltalk 到 X-Windows 的接口 STIX 的 GNU 版本。

Q　包含许多为不同硬件配置预先建立的内核。坦率讲，只要有足够的磁盘空间，建立自己的内核会更适合于自己的具体硬件配置。

T　包含 TEX 和 LaTeX。这是包含专业排字规则的文档排版系统。本书就是用 LaTeX 排版的。

TCL　用于快速开发 X-Windows 应用的 Tcl script 设计语言和 Tk 工具。

X　包含 XFree86。这是使用 fvwm 窗口管理程序的基本 X-Windows 系统。

XAP　包含其他的 X-Windows 应用软件包：绘图软件包（xpaint）；影像处理软件包 xv（注意：xv 是当前尚未注册的共享软件而不是自由软件）；电子表 xspread；一些 X-Windows 的游戏；postscript 预检程序和打印程序（ghostview）以及调制解调器通信程序（seyon）。

XD　开发自己的 X 服务程序所需的软件包。

XV　Xview 库；Open Look 窗口管理程序以及 X-Windows 唱盘播放程序（workman）。

Y　不需要 X-Windows 的游戏程序。包括 asteroids，tetris 以及 doom 等。

从清单可以看出：如果选择安装 A，AP，D，F，K，N，X，XAP 和 Y 软盘组，将得到一个功能强大的通用系统。

除了这些软件包外，还需要弄到安装这些软件包的软件。对 Slackware 发布讲，这些软件放在另外两张软盘上。它们和其他盘组一起提供。应经常阅读 Linux 发布中的 README 文件，因为这些文件中经常包含一般文档的最新的更新或修改信息。

7.3　安装 Linux

在安装和配置 Linux 系统的过程中，记录你做的每一件事是个好主意。特别是当你作出选择时，记录你实际的选择。在整个过程中要作出许多选择。其中一些你会知道它的影响，容易排除错误的选择。还有许多选择却是非常任意的。当重复实践时，不容易搞清当初实际做了什么选择，除非把它们记下来。

一旦有了 Linux 发布，下一步就要做安装计划。如果希望将 Linux 和另外一个操作系统

(如 DOS)同时放在一台机器上,计划就显得更加重要。要决定的最大问题是为每个操作系统留出多少磁盘空间。如果决定错了,唯一的实际解决办法就得将所有软件从机器上移走,这样才能重新移动两个系统之间的边界。按新分配的容量把所有软件重新安装一边。否则要另加一个磁盘驱动器,把装不下的软件放到新盘上去。

7.3.1 磁盘分区

假定要将 DOS 和 Linux 同时放在一台机器上,最好从一台带空盘的机器开始。如果是一台过去全部用来运行 DOS 的机器,那就需要对磁盘上需要的所有内容作备份。这样才能对磁盘重新进行分区,给 Linux 软件留出足够的空间。

在 DOS 系统下,磁盘可以划分为分区。每个驱动器最多可以建立四个主要分区。在单个硬盘的系统中进行简单的 Linux 安装,通常这就够了。例如,用单个 320Mb 磁盘,可以将磁盘空间分成如图 7.1 那样的三个分区:

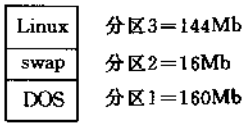

图 7.1 对 320Mb 硬盘进行分区

其中第一分区用作 DOS C:驱动器,第二分区用作 Linux 交换分区,第三分区用来保存 Linux 的根文件系统。Linux 系统的每个单独的文件系统都应该有它自己的分区。

有时,需要在一个硬盘上使用 4 个以上的分区,特别是大容量的磁盘更需要这样做。在这种情况下,4 个主要分区中的一个可用作扩充分区。它像一个容器一样用来保存一组逻辑分区(DOS 系统下最多允许有 23 个逻辑分区)。

除了要考虑扩充分区和逻辑分区外,大容量磁盘驱动器还有另外一个问题:由于设计的失误(设计者未曾想到会有超过设计极限的硬盘驱动器可供实际使用),一般情况下,不能从柱面数超过 1023 的磁盘分区中引导 DOS。这同样表示:Linux 的根文件系统必须全部放在柱面数少于 1024 的分区中。数据和文件的分区可以超出这一界限。但是不能从这些分区引导 Linux。对 IDE 盘来说,柱面数 1024 达到 504Mb。除非用新的母板,它的 BIOS 支持大容量磁盘驱动器,才能不受这一限止的影响。

假设你有一个 720Mb 的 IDE 硬盘,要对它进行如下的划分:150Mb 用作 DOS C:驱动器,200Mb 用作 DOS D:(可能用作压缩文件的分区),16Mb 用作 Linux 交换分区,留下的 354Mb 用作 Linux 文件系统空间。图 7.2 表示一种可能的分区方案。方案中已经考虑到所有的限止。

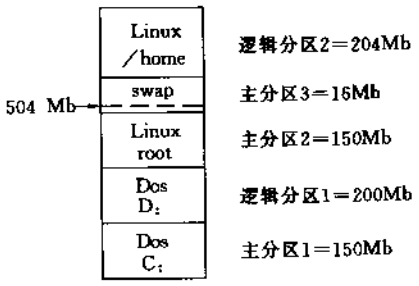

图 7.2 对 720Mb 硬盘进行分区

注意:Linux 空间已经分成两个部分,保存根文件系统的分区仍然留在 504Mb 的边界内。这样其他文件系统或分区可以装配到根文件系统下。本例表明,Linux 空间已分为系统分区和用户分区。所有 Linux 的安装工作将在根分区内进行。开始时,其他分区为空白。以后将它作为 /home 目录装配在根目录下,用作用户的起始目录和工作空间。可以采用许多其他方案,需要进行试验以找到最适合于你的特定环境的方案。

对磁盘进行分区的程序称为 fdisk。DOS 和 Linux 下各有一个 fdisk。一般只能分别用来建立各自的分区。现在,第一项任务就要从主盘(master disk)来安装 DOS。利用 DOS 的 fdisk 建

立 DOS 分区,留下其余的磁盘空间不加分配。

大多数 Linux 发布提供 1 张或 2 张软盘,作为引导 Linux 系统用。slackware 发布提供两张盘,分别称为 boot(引导)和 root(根)盘。它们都有不同的版本,你需要选择合适的版本。选择的依据是硬盘的类型(是 SCSI 盘还是其他盘)以及 Linux 源文件使用的媒体(光盘,软盘,硬盘或网络)。

应阅读 Linux 发布的发行注释(可能在 README 文件中,也可能在薄薄的分页印刷品中)帮你找出合理的选择。它也告诉你如何根据文件说明来建立 boot 和 root 盘。

在本节的其余部分中,假定有一个 320Mb 的 IDE 盘,并已按照 7.1 图进行分区。DOS 分区已经建立,而且安装了 DOS 软件。进一步假定愿意安装 slackware 发布的 A,AP,D,K,N,F,X,XAP 和 Y 软件包,使用的安装媒体是软盘(其他安装媒体的安装过程本质上是相同的)。

一旦有了你需要的 boot 和 root 盘,应插入 boot 盘重新启动机器。大多数 Linux 发布(包括 slackware)提供的 boot 盘允许在提示下送入硬件配置的细节。只在 boot 盘上的内核不能检测到特定的硬设备时才需这样做。这种情况并不经常发生。

本例的配置不存在问题。只要在出现提示时,按 Enter 键使引导过程继续进行即可。经过短时间后,将得到去掉 boot 盘,换上 root 盘的提示。用 root 盘更换 boot 盘后,继续按 Enter 键来完成初始的引导过程。

在这一阶段,你将得到一个完全在内存中运行的,用虚拟盘(ramdisk)作为根文件系统的最小的 Linux 系统。如果一切进行顺利,将得到 login 提示符。现在,你作为 root 用户登录。这一账号现在还不设置口令,所以能直登录到系统中去。按惯例,系统管理员或超级用户的提示符用 # 号表示。本书的例子中也这样使用。这有助于避免与普通用户的提示符相混淆。

为了建立 Linux 分区,要在 bash 提示符后面送入 fdisk。它的输出如下:

```
# fdisk
using  /dev/hda  as default  device!

command  (m for help):
```

设备特殊文件名 /dev/hda 代表第一硬盘驱动器的整体(/dev/sda 用来表示 SCSI 盘)。在 fdisk 命令中不给参数时,这是 fdisk 进行分区的默认驱动器。如果有第 2 个硬盘驱动器需要进行分区,可以在命令中指定第 2 驱动器的设备特殊文件的路径名作为参数。第 2 驱动器的路径名一般用 /dev/hdb 或 /dev/sdb 表示:

```
# fdisk  /dev/hdb
```

在 Linux 系统中,每个磁盘驱动器和分区都在 /dev 目录下给出相应的名称,这些名称如下:

```
hda   第 1 硬盘驱动器的整体;
sda   第 1 SCSI 硬盘驱动器的整体;
hdb   第 2 硬盘驱动器的整体;
sdb   第 2 SCSI 硬盘驱动器的整体;
fd0   软盘驱动器 0(DOS A:驱动器);
```

fd1　软盘驱动器1(DOS B:驱动器);

任何硬盘驱动器上的分区都可以用驱动器的整体名称后面加相应的数字来进行访问。下面的清单表示如何对 hda 驱动器的分区命名:

hda1　第1硬盘驱动器第1主分区;
hda2　第1硬盘驱动器第2主分区;
hda3　第1硬盘驱动器第3主分区;
hda4　第1硬盘驱动器第4主分区;
hda5　第1硬盘驱动器第1逻辑分区;
hda6　第1硬盘驱动器第2逻辑分区;
hda7　第1硬盘驱动器第3逻辑分区;

Linux fdisk 命令和 DOS 的对应命令看上去不同,但两者完成类似的功能。在 fdisk 运行时,可以用它的 m 命令得到它的功能清单:

```
# fdisk
Using /dev/hda as default device!

Command (m for help): m
Command action
  a   toggle a bootable flag
  c   toggle the dos compatibility flag
  d   delete a partition
  l   list known partition types
  m   print this menu
  n   add a new partition
  p   print the partition table
  q   quit without saving changes
  t   change a partition's system id
  u   change display/entry units
  v   verify the partition table
  w   write table to disk and exit
  x   extra functionality (experts only)
```

现在,用 p 命令显示已有的分区表,检查 DOS 分区是否已正确建立起来:

```
Command (m for help): p

Disk /dev/hda: 16 heads, 40 sectors, 1024 cylinders
Units = cylinders of 640 * 512 bytes

   Device Boot Begin Start  End  Blocks  Id  System
```

```
/dev/hda1   *     1     1      511   163812+   6   DOS 16-bit >=32M
```

从上面的结果可知它已正确建立。所以,现在可以建立 Linux 分区。从 /dev/hda2 上建立交换分区开始。可以用.n 命令增加一个新分区。然后,fdisk 将询问建立主要分区还是扩充分区。在本例中要建立主要分区。所以在提示时应该回答 p。接着它问你使用的分区号,应回答 2。fdisk 的下一个问题是新分区从那一个柱面开始,应给出 DOS 分区后面的第一空闲柱面号。本例中是 512。最后的问题是:分区在那里结束。这个问题可以用两种方式回答。可以用柱面号,也可以用分区的容量来回答。指定一个容量为 16Mb 的交换分区,可以回答 +16M。整个过程如下:

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
P
Partition number (1-4): 2
First cylinder (512-1023): 512
Last cylinder or +size or +sizeM or +sizeK (512-1023): +16M
```

顺便提一下,你可以有多个交换分区。如果硬盘不止一个,这种方案可能是最方便的。

建立了交换分区后,可以用类似的过程在 /dev/hda3 上建立 Linux 根分区如下:

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
P
Partition number (1-4): 3
First cylinder (564-1023): 564
Last cylinder or +size or +sizeM or +sizeK (412-1023): 1023
```

最后,用 p 命令来证实所有的一切是否按计划完成:

```
Command (m for help): p

Disk /dev/hda: 16 heads, 40 sectors, 1024 cylinders
Units = cylinders of 640 * 512 bytes

   Device Boot Begin Start   End  Blocks  Id  System
/dev/hda1   *      1     1   511  163812+  6  DOS 16-bit >=32M
/dev/hda2        512   512   563   16640  83  Linux native
/dev/hda3        564   564  1023  147200  83  Linux native
```

看上去很满意,但还有一件事要做。分区 2 的的系统类型仍然标为 Linux native。它应该

标为 Linux swap。可以用 t 命令将 Id 从 83 改为 82。也可以用 t 命令列出这些代码：

```
Command (m for help): t
Partition number (1-4): 2
Hex code (type L to list codes): L

0 Empty           8 AIX           75 PC/IX        b7 BSDI fs
1 DOS 12-bit FAT  9 AIX bootable  80 Old MINIX    b8 BSDI swap
2 XENIX root      a OS/2 Boot Man 81 Linux/MINIX  c7 Syrinx
3 XENIX usr       40 Venix 80286  82 Linux swap   db CP/M
4 DOS 16-bit <32M 51 Novell?      83 Linux native e1 DOS access
5 Extended        52 Microport    93 Amoeba       e3 DOS R/O
6 DOS 16-bit >=32 63 GNU HURD     94 Amoeba BBT   f2 DOS sec
7 OS/2 HPFS       64 Novell       a5 BSD/386      ff BBT
Hex code (type L to list codes): 82
```

现在分区表已按要求设置，最后用 p 命令再检查一次：

```
Command (m for help): p

Disk /dev/hda: 16 heads, 40 sectors, 1024 cylinders
Units = cylinders of 640 * 512 bytes

   Device Boot Begin Start   End  Blocks  Id  System
/dev/hda1   *      1     1   511  163812+  6  DOS 16-bit >=32M
/dev/hda2        512   512   563   16640  82  Linux swap
/dev/hda3        564   564  1023  147200  83  Linux native
```

为了以后使用方便，建议你记下 Linux 分区的块数(16640 和 147200)。使用 w 命令将新的分区表写回磁盘，然后退出 fdisk。运行 fdisk 时，如果在匆忙之中出了错误，可以用 q 命令退出 fdisk，不对磁盘的原始分区表作任何修改。

为了确定对分区表的修改产生了实际效果，应该用 boot 和 root 盘重新启动系统。一旦系统重新启动，作为 root 用户重新登录到系统中，就可以继续进行下面的工作。

7.3.2 建立文件系统

现在已经建立了 Linux 分区，下一任务是在这些分区内建立文件系统结构。这相当于 DOS 的格式化。先从交换分区开始。使用两条命令：用 mkswap 命令对交换分区进行格式化，再用 swapon 命令告诉 Linux 将分区投入使用：

```
# mkswap  -c  /dev/hda2  16640
# swapon  /dev/hda2
```

mkswap 命令的 -c 开关强制对分区内的坏块进行检查。命令行后面的数字是交换分区的块数，也就是你建立分区时记下的数字。swapon 命令只用一个参数，也就是使用的分区名称(这里用的是 /dev/hda2)。

在 Linux native 分区中建立文件系统之前，必须先选择使用的文件系统类型。虽然，Linux 支持许多不同的文件系统结构，其中最好的一个称为第 2 扩充文件系统（Second Extended Filesystem）。这也是我们要用的文件系统。可以使用 mke2fs 命令在 Linux native 分区中建立第二扩充文件系统：

```
# mke2fs  -c  /dev/hda3  147200
```

参数用来指定使用的分区和以块数计算的分区容量。同样，-c 开关强制进行坏块检查。如果有多个 Linux native 分区，需要对每个分区使用 mke2fs 命令。

7.3.3 安装软件包

对磁盘分区作好准备以后，现在可以将软件装入分区中去。每种 Linux 发布都有不同的操作方法。最重要的是将各个 Linux 分区装配在根文件系统下（现在它还在内存的虚盘中）。然后，可以对不同的软件包从软盘（或其他安装媒体）解压缩，并将它们复制到相应的位置。

一些发布要求用手工方式完成这些操作。操作过程虽然繁琐，但还不是特别困难的任务。另一些发布（包括 Slackware）提供了安装 script，自动完成这一操作过程。Linux 发布提供的安装 script 称为 setup。

```
# setup
```

运行这一命令将提供一个菜单，引导你完成其余的安装步骤。菜单的选项有：

help 显示 setup 的帮助文件。

keymap 如果使用的不是美式键盘，允许改变键盘的映射方式。这是使用非美式键盘的用户的第一项选择。完成选择后，setup 直接转移到 addswap 选项，不再回到主菜单。

quick 这里有 quick（快的）和 verbose（冗长的）两种选择。作为新用户 最好用 verbose 选项。这也是默认的选择。

make tags 当你取得 slackware 发布的安装经验后，这一选项允许你建立自己的标记（tag）文件，从每个盘组中预选要安装的软件包。

addswap 允许指定一个 Linux 交换分区作为最终的系统使用的交换空间。前面说过，可以建立一个以上的交换分区，为什么不让系统自由使用所有交换分区？这些信息将在后面放到一个配置文件中。当作出选择后，setup 将问你是否要对这一分区执行 mkswap 和 swapon 命令。如果在运行 setup 前已经运行了这两条命令，就不必再让 setup 命令去做这件事。如果你用的是美式键盘，这将是开始安装的第一个选项。完成选择后，setup 进入下一项，不回到主菜单。

taget 这里指定那些磁盘分区将包含在最终的目录层次结构中。setup 显示可能使用的分区清单，让你指定用作根文件系统的分区。在我们的例子中只有一种选择：/dev/hda3。如果你尚未用 mke2fs 命令在分区中建立文件系统，现在就可以这样做。本例已在前面做过了。如果还有其他分区，它将问你是否使用这些分区，希望将这些文件系统装配在那个目录下。例如，如果我们使用图 7.2 的分区设置，就在这里指定 /home 分区。

source 这里指定安装的源媒体。一般的选择有：光盘、软盘、硬盘或网络。如果选择软盘，它将进一步让你指定驱动器和软盘的容量。

disk sets	这里选择要安装的软盘组。必须安装 A 组软盘,其他就由你选择。如果你选择 A,AP,D,K,N,F,X,XAP 和 Y 组,你将得到一个很完整的通用系统。
install	这一选项对选择的软件包解压缩,并将它复制到 Linux 文件系统中去。通常从上一选项直接进入本选项。它将问你如何提示。作为新用户应选用 normal。每一盘组都包含软件包的清单。一些软件包标记为 REQUIRED(必要的),将自动进行安装。另一些标记为 RECOMMENDED(推荐的)或 OPTIONAL(可选的)。对这些软件包它将问你是否愿意安装它们。必须确实在安装过程中仔细阅读了屏幕上的所有信息。需要作出选择,而不是将它们统统装进去。
configure	在软件包安装后,此选项允许你选择从硬盘还是从软盘引导 Linux 系统。它也允许你完成一部分机器配置工作。如鼠标,调制解调器和网络的配置。作为开始最好跳过其中一些选择。以后再用手工来完成配置,使你有机会熟悉配置过程的细节。然而必须在此时利用机会建立引导软盘和配置 lilo(有关 lilo 的信息请参阅 7.3.6 节)
pkgtool	此选项供以后作软件包的维护用。它允许你增加和删除单个软件包。
exit	离开 setup,回到 shell 提示符。

7.3.4 引导 Linux

不管选择和安装了那一种 Linux 发布,它必须提供一些手段,当你第一次打开机器时用来引导 Linux。它可以提供一张软盘,从软盘进行引导。它也可以从硬盘利用内核装入程序(Linux kernal loader)来引导。Slackware 发布提供两种手段。可以在 setup 命令的 configure 选项中建立 boot 盘和配置 lilo 引导装入程序。

为了说明这是简单的,我们将用手工来建立 boot 盘。这也可以作为 setup 建立的 boot 盘的备份。事实上,只要将 Linux 内核映像从硬盘复制到软盘,它就自动成为可引导盘。

不同的 Linux 发布将内核映像文件存放在不同的位置。最常用的是:

```
/vmlinuz
/zImage
/etc/zImage
```

slackware 发布用 /vmlinuz 作为默认的位置。确定内核映像的名称和位置后,你仍然需要为它在软盘上运行进行正确配置。可以用 rdev 命令来做到这一点。rdev 命令允许直接在内核文件中修改内核配置的细节。首先要弄清楚内核文件中根分区的名称是否已经正确设置。根分区在初始装配时设置为只读状态,所以这是可以检验的。这些操作可以用下面两条命令来实现:

```
# rdev  /vmlinuz  /dev/hd3
# rdev  -R  /vmlinuz  1
```

很明显,如果说内核映像名称和根分区名称和上面例子中的不同,应该用自己的名称代替。

现在需要一张空盘来进行复制(一张 DOS 格式化盘就行)。如果需要格式化软盘,可以用 fdformat 命令。例如,可用下面的命令在 A:驱动器(fd0)格式化一张 1.44Mb 的软盘:

```
# fdformat  /dev/fd0H1440
```

注意:在设备特殊文件名称中 fd0 后面跟了 H1440。这将强制 fdformat 将软盘作为 1440Kb (即 1.44Mb)的高密度盘处理。还有代表其他标准容量的软盘的其他的特殊文件。可以用下面的命令来显示这些特殊文件:

```
# ls /dev/fd0 *
/dev/fd0        /dev/fd0H1440  /dev/fd0H720  /dev/fd0h360
/dev/fd0D360    /dev/fd0H2880  /dev/fd0d360  /dev/fd0h720
/dev/fd0D720    /dev/fd0H360   /dev/fd0h1200
```

有了空盘和内核映像名称后,用下面的命令复制就行了:

```
# cp  /vmlinuz  /dev/fd0
```

现在,可以像 setup 建立的盘一样用新盘来引导机器了。

7.3.5 重新引导的过程

除非在 setup 过程中安装了 lilo,否则,现在只能从软盘来引导 Linux 系统。有了两张软盘,现在是重新引导机器检查一切是否正常的好机会。不要忘记,此时此刻仍然用虚盘上的根文件系统在运行系统。

Linux 是大机器的操作系统。任何情况下,不能用关掉机器的办法来停止 Linux 的运行。要使用正确的方法停止系统的运行。正常情况下,Linux 尽可能将它所做的许多工作留在内存中而不是写在磁盘上。理由是明显的,访问内存中的数据要比访问磁盘中的数据快几个数量级。所以,当要停止 Linux 系统运行时,要事先给它警告。使它有时间将常驻内存的数据写到磁盘上去。

使用 shutdown 命令停止 Linux 运行是最安全的办法,它的一般格式为:

```
shutdown flags time [message]
```

这一命令只能由 root 用户使用。如果在命令中给出信息(message),执行命令将传送给所有用户。如果说没有在命令中指定信息,它将给出默认的警告信息。当关闭的时间到达时,将给所有进程送出 SIGTERM 信号,使它们在关闭系统前有机会正确地结束运行。

命令中的 flags 表示命令开关:-h 开关表示执行命令后停机,-r 表示重新启动机器。time 参数用来指定关闭系统的时间。它有两种格式:用 hh:mm 指定具体时间;用 +mm 表示从当前时间算起的分钟数;now 可以用来表示 +0 分钟。

在 shutdown 命令的运行期间,它保证磁盘上的数据的和内存中的数据内容一致,不丢失数据。下列命令将立即停止 Linux 运行,然后停机:

```
# shutdown  -h  now
```

配置 Linux 内核，用 Ctrl-Alt-Del 重新引导机器是可能的。很明显，任何人在任何时候都可以完成这一操作，不管他是否是 root 用户。在家用或个人使用的 Linux 机器上，这可能不会出现安全问题。如果它是一台主机，它的主键盘和屏幕允许未经许可的人接触的话，就要在这台机器上禁止这一功能。你将在前 7.4 节中知道如何去做。

由于虚盘根文件系统的空间的限制，许多 Linux 发布在 setup 盘上并不提供 shutdown 程序。因此在刚安装系统时，可能需要用 Ctrl-Alt-Del 来做第一次关闭系统的操作。

7.3.6 lilo

从软盘引导 Linux 应该工作得不错。但是，如果能直接从硬盘启动机器就会更加方便。特别是在机器上操作系统不止一个时，可以选择其中一个进行启动。正确设置 lilo 的配置文件就能做到这一点。

如果使用 lilo，在引导过程一开始就在屏幕上出现：

```
LILO
```

并短暂停顿一下。如果不作出任何反应，lilo 将引导默认的操作系统。如果在停顿时，你按了 Shift，Ctrl 或 Alt 三个键中的一个，系统将给出下列提示符：

```
boot:
```

在提示符后面，送入要引导的操作系统或内核的名称。如果在提示符后面按 Tab 键，将得到可供选择的操作系统名称清单。

为了使 lilo 提供这些选择，应设置 lilo 的配置文件：

```
/etc/lilo.conf
```

将要引导的操作系统或内核的有关信息写到文件中去。其中包括：它们在磁盘上的具体位置，在 lilo 的 boot：提示符后面使用的名称等。然后送入下面命令来完成安装：

```
# lilo
```

我们用例子说明 lilo 配置文件的具体内容。下面的例子可以和 320Mb 的硬盘的例子一起使用。引导的默认操作系统是 DOS，在 boot：提示符后面可以选择 DOS 或 Linux：

```
# LILO configuration file
# Global section
boot = /dev/hda
delay = 50

# DOS bootable partition section
other = /dev/hda1
```

```
  label = dos
  table = /dev/hda

# Linux bootable partition section
image = /vmlinuz
  root = /dev/hda3
  label = linux
  read-only
```

配置文件内部分成段:从 global 段开始,以后的各段说明在引导时可选择的操作系统或内核。每个系统或内核各占一段,global 段后面的第一段是默认的系统或内核的细节。

本例中,global 段包括两行。第一行告诉 lilo 从那个硬盘驱动器引导机器。第二行指出在 lilo 引导默认的系统前的停顿时间。用十分之一秒为单位给出。

由于 DOS 是默认的操作系统,紧接着就是 DOS 段。其中三行分别表示:包含 DOS 的分区名称,在 boot:提示符后面送入的标号以及分区表的位置。

最后一段包含引导 Linux 的信息:可引导的内核映像文件的完整路径名,包含根文件系统的分区名称,在 boot:提示符后面应输入的标号。最后根文件系统必须以只读方式装配,使得 Linux 文件系统检查程序能进行完整性检查。

如果愿意将 Linux 作为引导的默认操作系统,只要将 DOS 段和 Linux 段交换位置,将 Linux 段放在前面即可。

建立自己的配置文件时,应将自己的硬盘分区,系统标号和内核映像名称填到配置文件中去。

必须记住:每次修改配置文件时和每次修改内核映像文件时(不管是否修改配置了文件)都应该运行 lilo 命令。即使用旧的内核文件名称来命名新的内核文件,将它放在旧文件同样的位置,仍然要运行 lilo 命令来安装新内核。

7.4 建立内核

为了在尽可能多的机器上安装 Linux,大多数发布提供的内核配置成支持广泛的硬件系列。所以在安装和运行 Linux 以后,最好是建立适合于你的机器的实际配备的内核。一般讲,这样做能减少内核占用的内存空间,可将更多的内存用作工作空间。如果机器的内存本来就不大(例如,4Mb),这样做能带来明显的效果。

假如安装了内核的源程序和 GNU 的 C/C++编译程序,建立新内核就很方便。

第一项任务是检验 Linux 源程序是否已在 /usr/include 目录中建立了两个符号连接:

```
# cd /usr/include
# ls -l asm linux
lrwxrwxrwx  1 root  root   26 Jul 4 17:27 asm -> /usr/src/linux
/include/asm
lrwxrwxrwx  1 root  root   28 Jul 4 17:27 linux -> /usr/src/lin
ux/include/linux
```

如果符号连接已经存在,就不必进行第二步。如果这些连接不存在,需要建立。也许需要更换一对同名的目录。做法如下:

```
# cd /usr/include
# rm -fr asm linux
# ln -s /usr/src/linux/include/asm asm
# ln -s /usr/src/linux/include/linux linux
```

7.4.1 配置

下一步选择新内核中要包括那些功能和硬件驱动器。可以用下列命令来做这件事:

```
# cd  /usr/src/linux
# make config
```

现在将得到一系列提问和回答的对话过程。问在新内核中保留那些内容,去掉那些内容。在每个问题后面将看到[Y]或[N]。如果只按 Enter 键,就接受默认的回答。在完成配置选择后,系统将记下回答。下次再运行 make config 时,这些又成为默认的回答。

大多数问题有明确合理的回答。有几点值得在这里提一下:

```
Kernel math emulation  (CONFIG_MATH_EMULATION)  [n]
```

如果系统没有装浮点协处理器,回答 Yes。使浮点协处理器模拟程序包括在内核中。如果装了协处理器,回答 No。使内核减小一些。装有协处理器时回答 Yes 也不成问题,内核会检测到协处理器并使它投入工作。

```
Networking support  (CONFIG_NET)  [y]
```

即使没有装网卡,也应包括网络支持。因为许多软件包都用到 socket,它是主要的网络进程之间的通信方式。内核可以在单台机器的进程之间使用局部回环(local loopback),使这些进程能像通过实际网络一样使用 socket 来进行对话。

```
Limit memory to low 16MB  (CONFIG_MAX_16M)  [y]
```

除非机装有 16Mb 以上的内存,否则应回答 yes。

```
TCP/IP networking (CONFIG_INET) [y]
IP forwarding/gatewaying (CONFIG_IP_FORWARD) [y]
IP multicasting (CONFIG_IP_MULTICAST) [y]
IP firewalling (CONFIG_IP_FIREWALL) [y]
IP accounting (CONFIG_IP_ACCT) [n]
```

应包括 TCP/IP 连网软件在内。后面的 IP 选项只在机器完成重要的网络应用时才选用，如机器用作网络间的安全网关(secure gateway)。

```
Standard (minix) fs support (CONFIG_MINIX_FS) [n]
Extended fs support (CONFIG_EXT_FS) [n]
Second extended fs support (CONFIG_EXT2_FS) [y]
xiafs filesystem support (CONFIG_XIA_FS) [n]
msdos fs support (CONFIG_MSDOS_FS) [y]
umsdos: Unix like fs on top of MSDOS fs (CONFIG_UMSDOS_FS) [n]
/proc filesystem support (CONFIG_PROC_FS) [y]
NFS filesystem support (CONFIG_NFS_FS) [y]
ISO9660 cdrom filesystem support (CONFIG_ISO9660_FS) [y]
OS/2 HPFS filesystem support (read only) (CONFIG_HPFS_FS) [n]
System V and Coherent filesystem support (CONFIG_SYSV_FS) [n]
```

Linux 有能力支持多种类型的文件系统。实际上，只需要其中的几种。如果在建立 Linux native 分区时用了第二扩充文件系统，支持这一文件系统是必要的。装配 DOS 硬盘分区和软盘都很有用。你也肯定需要包含 /proc 文件系统，因为许多常用的 Linux 命令(如 ps)都要用到它。NFS 文件系统允许将机器的磁盘分区通过网络装配到另一台机器的目录层次结构中去。如果装有光盘驱动器，ISO9660 文件系统也是必要的。

运行 make config 命令以后，应编辑 /usr/src/linux/Makefile，以保证符号 ROOT_DEV 的定义是正确的。如果用作根文件系统的磁盘分区和当前运行系统中用的是同一分区，这一符号应设为：

```
ROOT_DEV = CURRENT
```

如果不是，应该使用根文件系统的分区名称。例如：

```
ROOT_DEV = /dev/hda3
```

7.4.2 编译

可能需要经过几次反复才能得到需要的配置。一旦配置完毕，应进行准备和编译。

应该先用下面两条命令进行准备：

```
# make dep
# make clear
```

第一条命令将配置所有的 makefile，使它们符合特定安装和内核配置选择。第二条命令将保证在源树(source tree)中不留下旧的二进制文件，使它们不影响选择的内核配置的编译。

在成功地完成前两项准备工作后，大约要用 1 到 2 分钟，即可开始编译：

```
# make zImage
```

送入这条命令后，可以休息一下。一台带 16Mb 内存的 486DX /66 大约需要 20 分钟。

编译完毕后，系统应产生一个可用于安装的，压缩的内核映像文件：

```
/usr/src/linux/arch/i386/boot/zImage
```

7.4.3 安装

为了测试新内核，应该用 7.3.4 节所讲的办法做一张可引导软盘来进行检验。

如果从新软盘引导机器取得成功，可以考虑设置新内核，使它能从硬盘引导。最安全的做法是在 lilo.conf 文件设置另外一个内核。然后运行 lilo。

对 lilo.conf 文件可以进行如下修改：

```
# LILO configuration file
# Global section
boot = /dev/hda
delay = 50

# DOS bootable partition section
other = /dev/hda1
  label = dos
  table = /dev/hda

# New Linux bootable partition section
image = /vmlinuz
  root = /dev/hda3
  label = linux
  read-only

# Old Linux bootable partition section
image = /.vmlinuz
  root = /dev/hda3
  label = linux.old
  read-only
```

其中有两段用于可引导的内核。前一段用于刚建立的新内核。后一段用于原有的内核。如果新内核有问题，总能回到旧内核。

为了使用新的 lilo.conf 配置文件，应执行下面的一组命令：

```
# mv /vmlinuz /.vmlinuz
# cp /usr/src/linux/arch/i386/boot/zImage /vmlinuz
# lilo
```

第一条命令用新名称保存当前的内核。第二条命令将新内核复制到 lilo.conf 文件中指定的位置,最后一条命令完成 lilo 安装。

现在,当重新引导系统时,在 boot:提示符后面有三种选择 。送入 dos,引导 DOS 操作系统;送入 linux,引导新内核;送入 linux.old,引导原有的内核。

7.4.4 loadlin

lilo 软件包只是 Linux 引导装入程序中的一个。另一个在 DOS 下运行的称为 loadlin 的软件包完成类似的功能。

从用户的观点看,两者的主要差别在于:lilo 允许用户在引导时选择操作系统。而 loadin 需要先引导 DOS 系统。只在进入 DOS 系统后,才能选择留在 DOS 内,还是用 loadlin 引导 Linux。

为了使用 loadlin,必须先将它和 Linux 内核文件复制到 DOS 分区。最方便的做法是将 DOS 分区装配在 Linux 下,然后用 Linux 的 cp 命令进行复制。

包括 Slackware 在内的一些 Linux 发布提供了这样的机会:允许在安装过程中将 DOS 分区配置成能在引导过程中自动将它装配到 Linux 目录中去。如果你的发布没有提供这种机会,或者你没有利用这种机会,可以用手工方式装配 DOS 分区。

可以使用 mount 命令将一个文件系统最上层目录装配到另一个文件系统的子目录下。它的一般格式如下:

```
mount -t type device directory
```

其中 type 表示文件系统的类型(例如,msdos,ext2,iso9660,nfs 等);device 表示要进行装配的文件系统的设备特殊文件的名称(如 /dev/fd0,/dev/hdb1 等);而 directory 表示当前目录结构中装配新文件系统的具体位置。

当用完文件系统后,可以用 umount 命令卸下文件系统。它的格式是:

```
umount directory
```

对软盘驱动器讲,在从驱动器中取出软盘之前,先将软盘驱动器卸下显得特别重要。这是因为可能有一些内存中的数据还没有写到软盘上去。而 umount 命令将把这些数据写到软盘上。

只有 root 用户才能使用 mount 和 umount 命令。

可以用下面的命令将 /dev/hda1 DOS 分区装配到 Linux 的 /mut 目录下:

```
# mount   -t  msdos  /dev/hda1  /mut
```

然后,将 loadlin.exe 和 loadlinx.exe 两个文件复制到 DOS 分区中。在 slackware 发布中,这两个文件压缩成 .zip 文件存在 /root 目录下。可以用下面的命令对文件解压缩并复制到 DOS 分区中去:

```
# cd /root
```

```
# unzip lodlin*.zip
# cp /root/LOADLIN/LOADLIN.EXE /mnt/dos
# cp /root/LOADLIN/LOADLINX.EXE /mnt/dos
```

如果你没有 Linux 的 unzip 命令，可以将 .zip 文件复制到 DOS 目录下，用 pkunzip 命令对它们进行解压缩。

现在你可以用下面的命令将压缩的 Linux 内核映像复制到 DOS 分区：

```
# cp  /usr/src/linux/arch/i386/boot/zImage  /mut/vmlinuz
```

现在一切已准备就绪，可以进行再引导了。

一旦机器被引导进入 DOS 系统，可以用下面的命令引导 Linux：

```
c:\ > loadlin  c:\vmlinuz  root=/dev/hda3  ro
```

第八章　用　　户

许多 Linux 机器作为单机使用，它们和外界没有联系。机上只有自己一个用户，看上去只是一台个人计算机。在这种情况下，考虑用户帐号和系统管理员义务也许会觉得多余。

不能用这种方式去考虑问题。不管你在家中，在一台单用户的专用个人计算机上；还是在大型机关中，在一台拥有 200 个用户的与 Internet 全面联接的服务器上；只要你有 root 帐号，就必须慎重对待。必须将自己看成守护人，而不是把自己看成主宰一切的神灵。你必须以普通用户和守护人双重身份在机器上工作。用普通用户的身份去完成所有日常工作，不需要特权用户帐号就能完成的系统管理工作。只当绝对必要时，才以守护人的身份以最短的时间去完成系统管理工作。当你拥有一台机器特别是一台同其他用户一起工作的机器的 root 帐号时，你就处于有义务和责任的位置上。对这一点怎么强调也不过份。这是不能掉以轻心的位置，也是不能由怀有不良意图的人利用的位置。

8.1　帐号的口令

刚装完 Linux 时，系统中只有一个 root 帐号。开始时，它不设置口令。第一项任务是对它设置口令。应该慎重选择 root 的口令。这是防止其他任何人以特权用户身份使用你的机器的重要手段。口令必须不容易被人猜出来，最好经常更换的口令。可以用 passwd 命令改变 root 和普通用户的口令：

```
#  passwd
```

即使你作为 root 帐号登录，也无法辩认其他用户用什么口令。当其他用户忘记口令时，你可以将他的口令全部去掉，让用户在下次登录时选择新口令。或者你为用户直接设置新口令。将新口令告诉用户，让他用新口令登录后再改变口令。后一种方法更安全，因为用户始终处于口令的保护之下。为了用 passwd 命令直接修改用户的口令，可以用用户的登录名作为参数：

```
# passwd  login_name
```

可以用 who 命令找出登录到你的机器上的其他用户：

```
# who
pc        ttyl     Jul  3 19:25
mot       ttyp0    Jul  3 20:06 (grexiz.soc.staff)
carey     ttyp1    Jul  3 20:08 (gregory.soc.staf)
rvc       ttyp2    Jul  3 20:12 (rachel.soc.staff)
```

who 命令显示的信息分成几个字段。第一字段是登录名;后面是用户使用的终端名称;接着是用户登录的日期和时间。最后,如果用户通过网络登录到你的机器上,将给出连接的机器的名称。

8.2 增加用户帐号

当你为 root 帐号设置口令后,下一任务应该给自己建立一个普通用户帐号。许多 Linux 发布提供专门的命令。这一命令称为 adduser 或 useradd。不管用什么名称,它将问你有关新帐号的一系列问题。在提供答复(或接受默认值)后,就建立了新用户帐号。

这样做简化了整个过程。作为新的管理员了解机器内部发生的过程更为重要。这样,你可以作出有根据的选择。为了了解过程的细节,我们将用手工的办法增加帐号。事实上,手工增加用户帐号并不复杂。它包括下列步骤:

1. 在 /etc/passwd 文件中建立登录登记项(entry)。
2. 建立用户起始目录。
3. 设置起始目录的拥有者。
4. 设置起始目录的访问权限。
5. 在 /etc/group 文件中设立用户组登记项。

在建立口令文件的登记项之前,应先确定帐号的一些细节:包括登录名称,UID,GID,起始目录的路径名以及用户使用的 shell。

8.2.1 口令文件

有了这些信息后,现在可以编辑 /etc/passwd 口令文件。在文件的最后增加一行,将这些细节按下面的格式填进去:

```
login_name:OFF:uid:gid:user's_real_name:home_directory:shell
```

除非你的单位有命名的准则,否则选择登录名纯属个人爱好。

由于系统用 UID 来确定用户,UID 必须是唯一的。除了这一点以外,UID 和 GID 的选择也是任意的。如果你愿意使用 GID,可以用新用户组的组号。Slackware 发布的新用户组(users)的标准 GID 是 100。如果不需要用户属于特定的用户组,可以将他的 GID 设成和 UID 一样。这些用户属于由一个用户组成的组。

用户的起始目录可以放在目录层次结构中的任何位置。为了防止到处乱放,对它们的位置应该有统一的考虑。将普通用户的起始目录放在 /home 目录下是好的选择。如果预计系统中将有很多用户,在 /home 目录下装配大的分区是合理的。这样用户帐号既能获得他们所需的空间,而又不致用尽根文件系统的空间。

最后,虽然有许多其他的 shell 可供选择,bash shell 应是最好的选择。

写到口令文件的新行具有类似下面的格式:

```
roger:OFF:1234:100:R.N.Foxcroft:/home/roger:/bin/bash
```

8.2.2 起始目录

建立新帐号的后3步可以用3条命令完成：

```
# mddir  /home/roger
# chown  roger.  /home/roger
# chmod  700  /home/roger
```

第一条命令建立起始目录（/home/roger）。chown命令将起始目录归roger用户所有。用户名称后面的句点告诉chown命令，将起始目录的用户组同时改为用户 的GID（它也可以用chgrp命令来完成）。最后的命令给目录的拥有者全面的读、写和搜索权限。而其他任何用户都没有访问权。这一点以后可以由roger用户自己根据需要加以改变。

对本例讲，建立新帐号的第5步也就是最后一步，现在不需要做什么。因为新用户使用了/etc/group文件中已有的GID(100)。另一种情况是用户的GID和UID相同，在这一步也不需要做什么。如果用户使用的GID在group文件中没有相应的条目，那么系统只在显示组名的位置上用这一GID来代替（例如用ls -l命令列出清单时）。只在下面两种情况下，才真正需要在group文件中建立或修改相应的条目：当要增加实际超过一个用户的用户组时，或者当你要在多于一个用户的用户组内增加新用户时，才需要修改group文件。

8.3 用户组

大多数Linux机器很少或实际没有使用Linux提供的用户组功能。然而，它能对一些安全问题提供简练的解决办法。

使用用户组功能使组内的用户对一组文件具有共同访问的权限，而拒绝系统中其他用户访问这些文件。可能出现这样的情况，需要特定的用户在不同的时间参加不同的组。这种情况下，除非用户自己能改变参加的组。否则，组的概念用途不大。事实上，Linux系统能够做到这一点。

为了更好理解这一点，先看一下group文件的格式。它由许多行组成，每行代表一个组。行的格式如下：

```
group_name:password:gid:user_list
```

它和口令文件的格式大致相同。每行用冒号分成4个字段。它们分别是：

group_name	组的名称。它出现在ls -l命令的输出清单中的第4列。
password	可选用的口令。如果一个组设置了口令，当你希望改变到这个组时，就要求提供口令。
gid	与用户组名结合在一起的用户组识别号GID。
user_list	选项。用逗号分开的用户名清单。可以在这里增加用户名，使用户加入这一组。

/etc/group 文件具有类似下面的内容：

```
root::0:root
bin::1:root,bin,daemon
daemon::2:root,bin,daemon
sys::3:root,bin,adm
adm::4:root,adm,daemon
users::100:
book::500:
```

当用户登录时，系统在口令文件的 GID 字段给用户指定默认的组号。以 roger 用户为例，他的帐号刚建立时，在口令文件中指定的默认 GID 是 100。也就是 roger 用户登录后，立即将他放在 users 组内。从上面的 group 文件的内容看，roger 用户不具有改变到任何其他用户组的权限。

假如，临时想让 roger 改变到 book 组，使他和 book 组一起工作。这可以用两种办法实现。最好的做法像下面那样将 roger 写到 book 组的用户名清单中去：

```
book: :500:roger
```

现在，roger 用户可以用 newgrp 命令改变到 book 组。

```
$ newgrp book
```

这样，改变组的要求就实现了。使用不带参数的 newgrp 命令将使用户回到默认的组。

另一种改变用户组的办法是在组内设置口令。不必再在 group 文件中送入用户名。一旦设置了口令，当 roger 用户试图转到 book 组时，在允许他访问新组前，要求他提供组的口令。

第二种方法看上去不错，但不如第一种方法安全。虽然 roger 现在需要提供口令，但是系统中任何知道组口令的用户也可以转到 book 组去。如果口令落入非授权用户的手中，安全就成为问题。当一个组比较大时，发生这种情况的可能性就很大。另一问题是：如何在不改变组内其他用户的口令情况下，将一个知道组口令的用户从组内去掉。

这些就足以使你明白：为什么在 group 文件中明确列出用户名清单的做法更为安全。

8.4 安全

在讨论安全和口令问题时，我们会提出另一个问题。假定你运行一个程序，所产生的进程将拥有和你一样的文件访问权限。作为普通用户，你并没有写 /etc/passwd 文件的权限。你又如何去改变文件中的口令呢？

在一般情况下，上面的说法是对的。然而，为了让你能写口令文件，必须要有另一种控制方式。当你运行程序时，所产生的进程就像其他人运行的进程一样，能够做一些你的权限所不允许做的事情。对口令文件讲，这个其他人就是 root。因为，只有 root 才对口令文件有写的权限：

```
$ ls -l /etc/passwd
-rw-r--r--  1 root   root    775 Jul 25 15:41 /etc/passwd
```

我们再看 passwd 命令的执行文件的权限就明白了：

```
$ ls -l /usr/bin/passwd
-rws--x--x  1 root   bin    3964 Mar 21 09:37 /usr/bin/passwd
```

这里有些不寻常的地方。文件拥有者的权限位设置为 rws。文件拥有者的执行权限位设为 s 表示什么含义？它表示对文件拥有执行权的任何人，这里是每个人，将用文件拥有者(root)的权限而不是他们自己的权限去运行这一程序。

同样可以将组的执行设为 s(但用得不普遍)。下面是一个例子：

```
$ ls -l /usr/sbin/lpc
-r-xr-s--x  1 root   lp    21508 Feb 14 19:03 /usr/sbin/lpc
```

系统中任何运行 lpc 命令的普通用户好像是 lp 用户组成员一样，对 lpc 访问的文件具有 lp 用户组的文件访问权限。现在，不用管 lpc 具体用来做什么。它和打印机操作有关，后面我们会讲到它。

用标准的 chmod 命令来设置这些“特殊”的权限位。数值 4 用来设置文件拥有者的执行权限位。数值 2 用来设置用户组的执行权限位。数值 6 表示两者都设置。数值 0 表示两者都不设置。如果在 ls -l 输出清单中出现小写 s，表示执行位已设置。下面的例子用来说明这些概念：

```
$ ls -l /tmp/testbits
-rwx--x--x  1 pc       book       0 Jul 26 09:14 /tmp/testbits
$ chmod 4711 /tmp/testbits
$ ls -l /tmp/testbits
-rws--x--x  1 pc       book       0 Jul 26 09:14 /tmp/testbits
$ chmod 2711 /tmp/testbits
$ ls -l /tmp/testbits
-rwx--s--x  1 pc       book       0 Jul 26 09:14 /tmp/testbits
$ chmod 711 /tmp/testbits
$ ls -l /tmp/testbits
-rwx--x--x  1 pc       book       0 Jul 26 09:14 /tmp/testbits
```

8.4.1 umask

作为附加的安全特征，有一条命令可以用来限止你运行的进程所产生的文件的访问权限。这条命令称为 umask。

umask 命令一般格式为：

```
unask [permission_mask]
```

其中，permission_mask 是 3 位八进制数，对应于你要设置的权限位。它不同于为你建立文件时设置的默认值。

如果你没有指定 permission_mask 值，umask 命令将显示当前的 umask 值。典型的情况如下：

```
$ umask
022
```

下面的一组命令说明 umask 的使用，以及在文件建立时它对文件权限产生的影响：

```
$ touch file1
$ umask 027
$ touch file2
$ umask 0
$ touch file3
$ ls -l file?
-rw-r--r--  1 pc   book        0 Aug  3 00:53 file1
-rw-r-----  1 pc   book        0 Aug  3 00:53 file2
-rw-rw-rw-  1 pc   book        0 Aug  3 00:54 file3
```

从上面你能看到，如果只是 touch 命令起作用(即 umask 的值为 0 时)，它产生的文件的权限位设置为 rw-rw-rw。你也可以看到 umask 的值为 022 和 027 对产生的文件的权限位设置的影响。

8.5 成批作业

一般情况下，当你送入一条命令时，总希望它能立即执行。然而，用户有时也需要以批作业的方式在以后某一时刻，甚至在用户退出登录后才执行这条命令。

8.5.1 cron

有一个称为 crond 的程序，大多数系统将它安排在引导过程中的某一时刻开始运行。名称后面的 d 表示这是 cron 守护程序。和许多其他守护程序一样，这一程序将在机器运行期间持续运行。它的主要功能是周期地检查一组命令文件的内容，并在适当的时间执行这些文件中的命令。

每个命令文件包括一组命令行。命令行的一般格式是：

```
minutes hours days months day  names command
```

command 前面的所有字段分别用来指定命令执行的日期和时间。例如：

```
25 8 27 Jan * birthday
```

表示每当元月 27 日上午 8 点 25 分执行称为 birthday 的用户程序。在星期字段中的 * 号表示不管当天是星期几。

一般用 * 号来匹配任何可能的值。它也可以用在日期和时间的其他字段。如：

```
25 8 * * * apponintments
```

这是说，不管几月几号星期几，每天上午 8 时 25 分运行 appointments 程序。

如果你只要在工作日运行 appointments，可以列出星期 1 到星期 5，其间用逗号分开。

```
25 8 * * mon,tue,wed,thu,fri appointments
```

如果是相继的值的清单，则可以用范围来表示：

```
25 8 * * mon-fri appointments
```

如果在同一命令行中同时指定日期和星期，则两者都有效：

```
0, 15,30,45 * 1 * mon oddtimes
```

上面的例子说，将在每月 1 日和每星期一每隔 15 分钟执行一次 oddtimes 命令。每个登录的用户都可以有自己的，由 crond 命令执行的命令文件。这些命令文件存在下列目录下：

```
/usr/spool/cron/crontabs
```

但是，普通用户没有直接访问这一目录的权限。他们必须用一条特殊命令来建立和修改他们的 crond 命令文件。这一命令称为 crontab。它有一些命令开关，使 crontab 命令可以用来删除，建立和修改个人的命令文件。主要的命令开关有：

file [- u *user*]	用 *file* 文件取代当前的命令文件；
- [- u *user*]	用标准输入取代当前的命令文件；
- l [*user*]	列出当前命令文件的内容；
- e [*user*]	用 vi 编辑当前的命令文件；
- d [*user*]	删除当前的命令文件；
- c *dir*	指定新的命令文件目录。

只有 root 用户有权使用 - c 开关来改变存放命令文件的目录。也只有 root 用户才能在命令行后面给出用户名称，将这一开关的操作用于指定的用户的命令文件。注意，方括号用来表示选项，不应和参数一起送入命令行。

8.5.2 at

使用 crontab 表目允许你以有规律的方式执行指定的命令。有时提交系统的作业只需在特定的时刻执行一次。可以用 at 命令来满足这一要求。这条命令仍然依赖 crond 进行操作，但是，将提供所需要的一次性的执行功能。

为了使 at 命令进行工作，在 root 帐号的 crontab 文件中有特殊的命令行，它的形式为：

```
0,5,10,15,20,25,30,35,40,45,50,55 * * * * /usr/lib/atrun
```

这一命令行使 /usr/lib/atrun 命令每隔 5 分钟执行一次。当执行 atrun 时，它检查 at 命令作业清单，找出任何需要在当前运行的作业，并使它投入运行。

提交 at 作业是方便的。at 命令的一般格式为：

```
at [ - f file] time
```

它将在指定的时间(time)执行 file 文件中给出的所有命令。如果命令中没有给出可选的文件开关，at 命令将从标准输入设备接受输入(默认为键盘)：

```
$ at 10:55
date >/tmp/at.job.test
Ctrl-d
Job c00cd3093.00 will be executed using /bin/sh
```

从键盘输入的 ctrl - d 是文件结束符(EOF)。注意：at 命令分配了唯一的作业访问号(本例为 c00cd3093.00)。这是 at 命令作出的响应。用户可以用 atq 命令检查未完成作业队列中的作业：

```
$ atq
Date                Owner  Queue  Job#
10:55:00 07/27/95   pc     c      c00cd3093.00
```

在上午 10 点 55 分以后检查/tmp/at.job.test 文件的内容得到下面的结果：

```
$ cat /tmp/at.job.test
Thu Jul 27 10:55:02 BST 1995
```

可以在 atrm 命令后面跟以作业访问号，将作业从队列中去掉。

可以用类似下面的灵活方式指定 at 命令的执行时间：

```
10am Aug 20
6pm tomorrow
```

```
7:21 +4 days
```

最后的例子表示4天后的7点21分。此外,还可用不同的格式给出日期,包括:

```
MM/DD/YY
DD.MM.YY
```

允许用任一种形式给出月份和日子。

8.5.3 batch

batch命令完成at命令类似功能。除了不用指定时间外,送入batch命令的方式和at命令相同。提交的作业的运行时间取决于机器的平均负荷。当机器负荷轻时就完成这一作业。能否使用batch命令取决于你是否安装了/proc文件系统,因为batch命令要从下面的文件中读取机器平均负荷:

```
/proc/loadavg
```

root用户可以根据具体情况决定是否让普通用户使用at和batch功能,基本规则如下:

- root帐号经常可以使用at和batch功能。
- 如果存在/etc/at.allow文件,则在此文件中列出登录名的用户有权使用这些功能。
- 如果不存在/etc/at.allow文件,只存在/etc/at.deny文件;则不在此文件中列出登录名的用户才能使用这些功能。
- 如果两个文件都不存在,则普通用户就无权使用at和batch命令。Slackware发布中有/etc/at.deny文件,而且是空的,因此允许每个人使用这些功能。

8.6 档案

在一些系统中,root帐号要为用户提供的另一项服务是档案和备份的建立和维护。

用来保存备份的最常用和最容易得到的媒体是软盘。如果保存的材料的数量少,任务就很简单。在软盘上建立Linux第二扩充文件系统,将它装配到根文件系统下。然后,将文件复制到软盘。假定开始时你有1.44Mb的非格式化软盘,将指定目录(本例中的src_dir)及其所有了目录和文件保存到软盘上去的过程如下:

```
# fdformat /dev/fd0H1440
# mke2fs /dev/fd0 1440
# mount -t ext2 /dev/fd0 /mnt
# cp -a src_dir /mnt
# umount /mnt
```

上面的5条命令完成以下的功能;

- 对软盘进行低级格式化。参数规定:软盘放在 fd0(即 A:驱动器中)。它必须按 1.44Mb 的高密度进行格式化处理。
- 在 /dev/fd0 上建立第 2 扩充文件系统,后面的参数给出以块计的容量。
- 将第 2 扩充文件系统从 /dev/fd0 设备装配到 /mnt 目录。这是专门用来装配的空目录。
- cp 命令中的 -a 开关告诉它进行档案复制。它表示:将 src_dir 目录下的所有子目录和文件统统复制到 /mnt 目录下,保持原有的文件名称和目录结构不变。
- 最后一条命令从目录层次结构中卸下软盘。你就可以将它们保存在安全的地方。

8.6.1 gzip

当归档的材料中有一个或几个大文件时,可能需要对它们进行压缩以节省软盘的空间。Linux 用于文件压缩的标准命令是 GNU 的 gzip 命令。它的一般格式是:

```
gzip [options] file
```

指定的 file 文件将被压缩为与原始文件同名的带后缀 .gz 的新文件。新文件将取代原始文件。

gzip 有许多选项,最有用的选项是:

-1　用最快的速度进行压缩,得到的文件要大一些;
-9　用最大压缩比进行压缩,运行的时间要长一些;
-l　列出被压缩文件的统计数字;
-h　显示帮助信息和选项。

在典型的情况下,大的文本文件能缩小 60%到 70%,这对磁盘空间将是很大的节约。

假定文件用 gzip 进行压缩,它可以用 gunzip 命令解压缩,使它恢复为原来的状态:

```
gunzip file
```

输出文件将取代输入文件,并将文件的后缀去掉。

zcat 命令可以在压缩状态下读出压缩文件的内容。除了它对压缩文件进行操作外,它和普通的 cat 命令相似。zcat 命令的输出送到标准输出设备。像下面那样读压缩的文本文件就特别有用:

```
$ zcat textfile.gz | more
```

这一命令将 textfile.gz 文件的解压缩形式通过管道送给 more 命令,用分页的方式显示文件的正文。

8.6.2 tar

当文件容量达到一定规模时,将文件备份到 Linux 文件系统中就变得不合适。需要将整

个目录和所有子目录和文件直接输出到磁带中去.。完成这项操作的命令称为 tar(即 Tape ARchiver)。

这条命令将指定目录中的所有文件有效地压缩为一个称为 tarfile 的大文件,然后写到磁带上去。

由于支持磁带驱动器的系统不多,tar 命令的用途似乎有限。但是,tar 命令能将它的输出文件写到软盘上,输出到屏幕上,甚至输出到普通文件中。除此以外,tar 命令还能完成其他功能。如果写到软盘的 tarfile 文件太大,不能装在一张软盘上,tar 命令可以将文件分到多张软盘上(称为多卷档案)。也可以告诉 tar 命令,在它将结果送到输出设备之前用 gzip 算法压缩 tarfile 文件。

tar 命令具有如下的格式:

```
tar [options] files
```

其中 files 是包含在档案中或从档案中提取的文件和目录清单。

tar 命令有许多选项(options),最常用的有:

-c	建立新的 tarfile 档案;
-x	从 tarfile 档案中提取文件;
-t	显示 tarfile 档案的内容;
-f *file*	指定 tarfile 的名称为 *file*;
-v	使 tar 命令显示它所做的工作;
-l	留在本文件系统中,不跨越装配点;
-M	建立/提取/显示多卷档案;
-N *date*	只存储指定日期 *date* 以后建立或修改的文件;
-z	用 gzip 压缩/解压缩档案。

按照惯例,tar 档案文件名称应加后缀 .tar。压缩的档案文件名称应加后缀 .tgz(Linux 发布的所有软件包可能都有这个后缀)。

为用户 rachel 的起始目录的内容建立压缩档案的命令如下:

```
# tar -cvzf  /tmp/rachel.tgz  /home/rachel
```

检验这一档案的命令如下:

```
# tar -tzf  /tmp/rachel.tgz
```

在不慎删除某些原始文件后,从档案恢复 rachel 用户的起始目录的命令如下:

```
# cd  /
# tar -xvzf  /tmp/rachel.tgz
```

在上面的例子中，使用 cd / 命令的理由是：tar 命令用绝对路径名，即相对于根目录的文件名保存档案中的所有文件。所以需要在根目录下将这些文件恢复到正确位置。如果在其他目录下对档案解压缩，tar 命令照样执行解压缩操作，但是将它们放在相对于当前目录的位置上。

最简单办法可以利用 tar　-t 选项检查档案文件相对于那一目录存放的。

如果要让普通用户在软盘上建立档案，要解决几个问题。首先，在一般情况下禁止普通用户读写软盘设备特殊文件(/dev/fd*)。其次，装卸软盘的 mount 和 unmount 命令是 root 用户专用的命令。这样做为了防止普通用户改变目录结构和可能出现的安全问题。这个问题的最安全解决办法是：用 C 语言写两个简单程序，专用来将软盘装到指定目录(如 /floppy)下或从指定目录中卸下。这些命令的实现请参阅 21 章后面的练习。

第九章　外部设备

任何操作系统(包括 Linux)的主要功能之一是提供应用程序和外部设备之间的接口。最基本的外部设备包括:

- 用于键盘输入和显示输出的控制台终端;
- 与其他系统进行简单的串行通信的调制解调器;
- 另一种输入设备鼠标;
- 用于列表,正文和图形输出的打印机。

本章将依次讨论这些设备以及 Linux 用来设置和使用这些设备的功能。

9.1　控制台终端

多数 Linux 系统只有一个用户,访问系统的主要手段是机器的键盘和屏幕,也就是控制台终端。事实上,控制台终端由两个独立的设备组成:键盘输入设备和屏幕/显示输出设备。

我们将看到这两个设备也有许多设置工作。

9.1.1　键盘输入

个人计算机的键盘似乎是简单的设备。看上去它只不过是一个上面有按键的盒子。每个键上标有数字,字母和符号。按一个键,在屏幕上就显示相应的字符。尽管看上去简单,实际它要复杂得多。

首先,许多使用个人计算机的国家在它们的字符集中需要增加少量的特殊字符,以克服不同语言之间的差异。因而存在许多格式和内容稍有差异的键盘。

从生产的角度看,生产电气性能和实际结构相同的单一键盘,再改变键上的符号,要比分别生产各种键盘便宜得多。为了做到这一点,需要安排某种形式的对照表。使得键盘上特定位置的单个键能对键上的符号产生不同的字节值。

Linux 系统中,这种对照表以软件的形式做在内核的键盘驱动程序内。等效于内核中有一张表对键盘上每一个具体的键以及这些键和 Ctrl,Shift 和 Alt 键的不同组合产生对应的字节值。

对键盘上每个具体的键,分别在 1 ~ 127 的范围给它们指定唯一的键控代码。然后用键盘映射文件对你按的键控代码或键控代码的组合产生相应的字节值。键盘映射文件默认地存在下列目录下:

```
/usr/lib/kbd/keytables
```

通常在引导系统时,用 loadkeys 命令将键盘映射文件装入内核。例如,你要装入英国键盘的映射文件,你可以用下面的命令:

```
$  /usr/bin/loadkeys  /usr/lib/kbd/keytables/uk.map
```

这条命令需要放在一个系统启动(system startup)文件中。这些文件通常放在 /etc 目录下(具体的位置随 Linux 发布而定)。这些文件的名称具有以下的形式:

```
rc.XXX
```

其中 XXX 表示文件用来配置什么。在 Slackware 发布中,loadkays 命令放在下列文件中:

```
/etc/rc.d/rc.keymap
```

键盘映射文件本身只是普通的文本文件,很容易看明白它的内容。如果需要为特殊的用途建立自己的映射文件,可以对它进行修改。只需要对这些修改建立映射文件,在标准键盘映射文件装入后,再装入修改的映射文件。

为了找出要修改的那些键的键控代码,可以使用 showkey 命令。在命令运行时,按下或释放任何键时,它的键控代码都在屏幕上显示出来。为了结束 Showkey 命令,只要在 10 秒钟内不按任何键,它将自动退出:

```
$ showkey
press any key,program terminates after 10s of last keypress
keycode  28 release
keycode  25 press
keycode  25 release
keycode  46 press
keycode  46 release
$
```

知道了要修改的键的键控代码,修改和再装入键盘映射文件就比较简单。映射文件的格式简单,手册页中对它有很好的说明:

```
$  man  5  keytables
```

可能需要多次修改键盘映射表,直到找到满意的结果为止。在这种情况下,可以用 dumpkeys 命令从当前的内核表中产生供以后装入的映射文件:

```
$  dumpkeys  >newkey.map
```

并在这一基础上对它进行修改。

9.1.2 显示输出

正像键盘一样,屏幕输出也有对照表。它的值可以根据特殊需要,或仅仅为了好玩而加以

改变。

在正常情况下,将一个字节送给屏幕,它的8位值用作显示的字符的代码。可以在显示过程中间插入一张转换表。这张表是由256个元素组成的字节数组。当你用转换表时,任何送给屏幕显示的字节不直接送去显示,而是将字节值作为转换表的索引,用它来从转换表中选取一个元素。选中的数组元素的8位值才真正送去显示。假如,表的默认设置为:

```
table[0] = 0
table[1] = 1
table[2] = 2
  -
  -
table[255]=255
```

这将和没有转换表时得到同样的效果。改变默认表中的任何值,在显示时就实现了从一个字符到另一字符的转换。

为了修改输出转换表,可以用 mapscrn 命令:

```
$  mapscrn  filename
```

其中,filename 是包含新的字符映射表的文件的名称。一旦表已经装入。可以送入一组特殊字符使它投入使用。这组特殊字符是 Esc 字符后面跟'(K'。还可以用另一组特殊字符来禁止它的使用。这组特殊字符是 Esc 字符后面跟'(B'。

在 mapscrn 命令的手册页中,对如何在转换文件中指定转换条目的格式有详细说明。实际上,它是一张从上到下排列的字符对照表。指定一个字符,后面跟以它的新的转换值。下面的例子有助于说明这一点。

假定由于某种原因,你要将任何大写字母 W,X 和 Y 转换为 * 号。指定这一转换的转换文件将包括以下内容:

```
'W'  '*'
'X'  '*'
'Y'  '*'
```

假定这个文件称为 newtab,可以用下面的命令将变换文件装入控制台显示驱动程序:

```
$  mapscrn  newtab
```

用 echo 命令传送特殊字符组给控制台驱动程序,使转换文件发挥作用:

```
$  echo  -e  '\033(K'
```

像下例所表示的那样:在传送这组特殊字符后,任何送给屏幕显示的 X,Y,和 W 字符都将

显示为 * 号。

```
$ echo -e '\033(K' VWXYZ
V* * *Z
$ echo -e '\033(B' V* * *Z
VWXYZ
```

在第 2 条 echo 命令中实际送的是 WXY,但出现了 3 个星号。这是因为转换表仍然在起作用。

显示过程中还有另外一张表,它保持在显示卡中。它的内容用来控制显示字符的外观。改变表的内容可以改变显示的字体。Lunux 下有许多可用的字体,要形成自己的字体也不难。

为了装入一种标准字体,可以用 setfont 命令:

```
$ setfont sc.fnt
```

其中 sc.fnt 是要装入的字体文件名称。如果你试用它,将发现 sc.fnt 是一种很有趣的字体。虽然不会经常去用它。保存字体文件的默认目录是:

```
/usr/lib/kbd/consolefonts
```

为了理解字符的字体文件的内容和格式,简单看一下显示正文字符的方式。在 VGA 屏幕上显示每行 80 个字符的 25 行正文时,每个字符在屏幕上占一个像元栅格。这个栅格高 16 个像元,宽 8 个像元。

每个可显示的字符在 8×16 像元的栅格内由一组亮和暗的像元组成。为了在屏幕上特定的位置显示特定的字符,要做的工作只是将要显示的字符栅格的像元值复制到屏幕上的对应的像元栅格中。

在字符栅格中,用 1 位来表示像元亮和暗。可以用 16 个字节来保存一个完整的 8×16 字符栅格的像元信息。每个字节表示像元栅格的一行。图 9.1 表示字母 A 的像元栅格。右边是 16 字节的值(用 16 进制表示)。

从字母 A 的例子可以看出,它的像元栅格的编码如下:

```
00 00 10 38 6c c6 c6 fc c6 c6 c6 c6 00 00 00 00
```

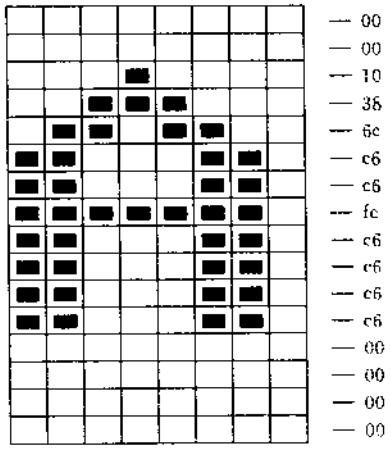

图 9.1 字母 A 的字符栅格和它的编码

你已经看到,送去显示的字符用 0~255 的数值表示(总数 256 个)。一组完整的 8×16 字符集将由 256 个不同的栅格组成。它的字体文件将包含 256 组 16 字节,将它们一个接一个地放在一起,总共将为 4096 个字节。

建立自己的字符集和字体文件的过程包括：设计每个字符的栅格，算出每个栅格的 16 字节值，将这些值存到一个文件中。然后就可以用 setfont 命令装入和使用。

除了 4096 字节长的字体文件外，还有不同长度的字体文件。这些字符集用于不同的屏幕分辨率方式或用于不同的适配器(如 EGA)。它们仍然包括 256 个字符，仍然用一个字节表示每一像元行，只是每个字符栅格占的行数要少一些。8×8 像元字符集的字体文件总长 2048 个字节。8×14 字体文件总长 3584 个字节。

setfont 命令还可以使用另一种类型的字体文件。这种文件的内容和格式和前面的一样，只在前面加了 4 个字节的标题信息。这样，总共有下列不同长度的字体文件：

```
2048 字节
2052 字节
3584 字节
3588 字节
4096 字节
4100 字节
```

setfont 从文件的容量就能确定文件的类型和格式。

9.2 调制解调器

将调制解调器接到系统中是一项技术性较强的操作。用调制解调器将计算机和电话线接起来，通过拨号与另一台计算机和调制解调器的组合进行连接，使得你的计算机可以和远程计算机进行通信。

为个人计算机购置调制解调器时，第一项选择是外置的还是内置的。这纯属于个人的选择，因为两者在操作上没有差异。我们倾向于用外置的调制解调器，仅仅是因为它不仅可以接在个人计算机上，还可以接到其他机器上。

调制解调器是串行设备，它需要通过串行口(COM1，COM2 等)进行操作。如果买的是内置调制解调器，它在内部设有串行口，只要按照手册说明对它进行配置即可。对于外置调制解调器，连接也不成问题。因为多数输入/输出卡带两个串行口，许多人用其中的一个接鼠标，还有一个空着可以用来接调制解调器。用外置调制解调器还要考虑另外一个问题。如果是高速的(19200 位/秒或更高)，应该使用串行口带缓冲器的 UART 芯片的输入/输出卡。

UART 是完成串行口功能的集成电路。个人计算机的标准设计用的是普通的 UART，它产生大量的中断请求(每个字符一次)，给 CPU 带来很重的负载。在调制解调器速率较高时，CPU 可能跟不上数据传送的速率而丢失数据。为克服这一问题，提供了带 16 个字符的缓冲器的 UART 的输入/输出卡。这种 UART 只在缓冲器满时才中断 CPU，一次将缓冲器内的字符全部取走。这使中断次数大为减少，使高速传送数据成为可能。

带缓冲器的 UART(有时称为 FIFO UART)的芯片号为 16550A。

用于 4 个串行口的设备特殊文件如下：

```
呼出设备     呼入设备     DOS 名称
/dev/cua0    /dev/ttys0   COM1
```

```
/dev/cua1   /dev/ttys1  COM2
/dev/cua2   /dev/ttys2  COM3
/dev/cua3   /dev/ttys3  COM4
```

为了便于记忆,将你的调制解调器用称为 /dev/modem 的设备进行符号连接。可以用类似下面的命令来实现这一点:

```
# ln  -s  /dev/cua0  /dev/modem
```

购买调制解调器要考虑的另一个问题是速率。尽量买快速的调制解调器,不要低于14400 位/秒。大多数现代的调制解调器用固定的高速率和计算机对话,并以运行速率与联接的远程调制解调器对话。由它本身完成两种速率转换。因而简化了计算机设置。许多现代调制解调器还完成数据压缩功能以提高有效的数据传输速率。将计算机和调制解调器的对话速率设为最高可用速率是好的做法。

这里有个小问题。Linux 下最高可用的标准串行速率(位/秒)为:

```
300,600,1200,1800,2400,4800,9600,19200 及 38400
```

而最高调制解调器速率为:

```
57600 和 115200
```

为解决这一问题,可借助于 setserial 命令。它能完成串行口的所有基本操作。现在我们需要它将串行口设为能达到的较高速率。这可以用下面的命令来完成:

```
#  setserial  /dev/cua0  spd_hi
#  setserial  /dev/cua0  spd vhi
```

参数 spd_hi 将 /dev/cua0(COM1)最高速率设为 57600 位/秒,而 spd_vhi 将它的最高速率设为 115200 位/秒。这些命令所做的只是将标准的 38400 速率换为指定的更高的速率。所以当你将串行口速率设为 38400 时,实际得到的是更高的速率 57600 或 115200。

通常在引导系统时设置串行口,而且用下列启动文件进行设置:

```
rc.serial
```

在 slackware 发布中,则运行 rc.S 文件中的最后一条命令来完成 。

setserial 命令还可以用来设置串行口地址和中断请求(IRQ)。这些以及其他的功能的细节参阅相应的手册页。

前面是将调制解调器用于呼出(outgoing call)所需要的全部设置。现在,可以运行你选择的终端模似程序,告诉它调制解调器进行连接的设备和使用的速率(不要忘记 38400),就可以拨号并与外界连接。

将调制解调器用于呼入(incoming call)的情况要复杂一些。首先,要设置一个在串行线上运行的程序,给任何呼叫人送去 login:提示符。其次,要设置调制解调器使它回答进来的振铃,并将远程机器连到你的机器上。

第一件事是容易的,用来给出 login:提示符的程序称为 getty。在 Linux 系统中,这一程序有两种形式,分别称为 agetty 和 getty_ps。它们完成类似的功能,设置串行线的特征和给出 login:提示符。

由 init 进程按下列配置文件的具体说明来运行 getty 程序:

```
/etc/inittab
```

使用 agetty 时,如果设置 /dev/ttyS0(COM1)以最大速率接受进来的登录请求,上述文件中的相应行应设为:

```
d1:45:respawn:/sbin/agetty -t60 38400 ttyS0
```

inittab 文件中的每一配置行包含用冒号分隔的 4 个字段。这些字段是:

1. 唯一的两个字符的识别符(d1 = 1 号拨号线)。
2. 这条线路的运行级别。当系统用于多用户方式下用 4 和 5 级。
3. init 对这条线路应采取的行动。关键字 respawn 表示当它的进程结束时,init 应重新运行这行的命令。
4. 这是 init 运行的命令行。这里,它在串行线 /dev/ttyS0 上用 38400 的速率运行 agetty 命令。并允许有 60 秒钟来响应 login: 提示符。

对应于 getty_ps 程序的 inittab 行应为:

```
d1:45:respawn:/sbin/getty -t60 ttyS0 F38400
```

其中 F38400 并不直接用作串行线的速度,只是作为下列配置文件中的一行的标号:

```
/etc/gettydefs
```

它用来设置串行线的特征。

在线路上有了 login:提示符后,接着要设置调制解调器使它监视电话线和振铃。这一过程很大程度上取决于使用的调制解调器类型。许多调制解调器是 Hayes 兼容的,能使用 Hayes 调制解调器的命令集进行操作。

如果调制解调器是 Hayes 兼容的,对它进行配置只少要用下面的一组命令:

```
ATS0 = 1
ATE0
ATQ1
```

所有的 Hayes 命令以 AT 开头(少量调制解调器区分大小写)以此来吸引调制解调器的注意。后面直接跟要执行的命令。上面三条命令的操作如下：

1. Hayes 兼容的调制解调器有一组内部寄存器,它们的值用来控制调解调器的不同操作。这些寄存器的名称为 S0,S1,S2 等。S0 寄存器用来指定回答呼叫前的振铃次数。S0 设为 1 表示调制解调器接到振铃后,立即断开振铃进行回答。将 S0 设为 0 时,将禁止调制解调器自动回答功能。
2. 一般情况下,当给 Hayes 兼容的调制解调器送去一条命令时,它将回送你的命令以备检查。但是,当 getty 在线路上运行等待用户登录时,调制解调器回送的任何字符将被看作是用户的登录名和口令。为防止这一点,必须抑止回送功能。E0 命令用来抑止回送,E1 用来恢复回送。
3. 第 3 条命令类似前一条命令。一般情况下,调制解调器对你要它完成的许多操作回送状态信息和结果代码。任何这样的信息同样会被 getty 误认为是登录名和口令。Q1 命令将抑止这一类信息,使调制解调器处于沉默状态。Q0 恢复正常操作。设置了调制解调器后,你应保存它的设置(如果你的调制解调器支持这一功能)。这样你就不必在每次开机和打开调制解调器时重新进行设置。

9.3 鼠标

鼠标是现代个人计算机的标准外设。Linux 能在标准正文模式和 X-window 模式下充分发挥它的作用。

鼠标有两种形式:串行的和总线型的(PS/2 型鼠标称为总线型鼠标)。差别在于它们使用的计算机接口不同。

9.3.1 串行鼠标

串行鼠标直接插在计算机的串行口上工作。用于串行鼠标的设备特殊文件是：

```
/dev/ttyS0
/dev/ttyS1
/dev/ttyS2
/dev/ttyS3
```

和调制解调器的情况一样,通常的做法是用 /dev/mouse 对串行设备进行符号连接：

```
# ln -s  /dev/ttyS1  /dev/mouse
```

9.3.2 总线鼠标

如果你买鼠标,除非有充分的理由,否则还是买串行鼠标好。如果计算机已经有 PS/2 鼠标接口,或者已经有了总线鼠标,或者你决定非买总线鼠标不可,那就用总线鼠标吧! 它的安装和配置要多花一些工夫。

总线鼠标要用自己的接口卡,鼠标直接插在卡上。有时总线鼠标接口卡插在其他卡上,如显示卡。这样,它就不需要在母板上占用一个插槽。

在机器上插上接口卡后,就要设置接口卡使它产生的中断请求不和其他设备发生冲突。

然后重新编译内核使它包含支持总线鼠标的软件。大多数总线鼠标接口卡使用 IRQ5 作为它们默认的中断号。许多其他卡也用它作为默认值。所以要小心选择。不能使用的 IRQ 有：

0 1 2 6 8 13 14 和 15

这些已用于一些标准设备，不能加以改变。留下你能选择的有：

3 4 5 7 9 10 11 和 12

其中前 4 个已用于标准串行口和并行口：

IRQ	分配
3	COM2
4	COM1
5	LPT2
7	LPT1

多数机器装有两个串行口和一个并行口，留下 LPT2 口的 IRQ 作为备用。其他的 IRQ(9 到 12)虽然作为备用，但不能用 8 位的 ISA 卡进行访问。包括多数独立的总线鼠标接口卡在内，所以你可能会遇到问题。

9.3.3 gpm

gpm 是 Linux 系统中特别有用的应用程序之一。这条命令实际作为守护进程来运行。通常在引导系统时，在一个标准的 rc.XXX 文件(常常用 rc.local)中执行。

它允许在虚终端屏幕或虚终端之间使用剪切和粘贴功能。当 gpm 运行 时，移动鼠标将导致鼠标光标在屏幕上的同步移动。将鼠标光标移到你要剪切的正文的起点，按下鼠标左键，拖动光标到正文块的终点。释放左键，将在屏幕上对要剪切的正文块作出标记。现在按三键鼠标的中键或两键鼠标的右键将使标记的正文块粘贴在正文光标所在的位置。也可以在标记正文后切换虚终端将它粘贴到其他不同的应用中。

当运行 gpm 命令时，需要用下面的命令指定安装的鼠标类型。

```
# gpm -t msc &
```

gpm 支配的鼠标类型包括：

ms microsoft 串行鼠标；
msc mouse system 串行鼠标；
bm 总线鼠标协议总线鼠标；
ps2 PS/2 总线鼠标。

9.4 打印机

多数个人计算机的打印机用并行口作为 Centronics(森特)接口。在 DOS 环境下,打印机接口称为 LPT1 和 LPT2。Linux 系统中对应的设备特殊文件是:

```
/dev/lp1
/dev/lp2
```

大多数系统只安装一台打印机,通常用 /dev/lp1。

如果你习惯于在 DOS 环境下使用打印机,你会习惯于直接和单独使用打印机。然而,Linux 是多用户和多任务操作系统。即使系统上只有一个用户,仍然需要按多个用户可能同时请求打印的情况来设置打印机。

标准的做法是采用假脱机(Spooling)打印程序。当假脱机打程序运行时,任何打印请求并不直接送给打印机,而是将它送到 Spool 目录下的文件中。然后另一个程序将打印作业每次一个从 Spool 目录下取走,送给打印机打印。

Linux 的情况稍复杂些。因为它除了可在本地打印机上打印外,还可以通过网络在远程打印机上打印。

9.4.1 打印服务

扫描 Spool 目录(每个打印机有一个目录)以及打印等待打印的文件的任务由称为 lpd 的程序来完成。lpd 实际是行式打印机的守护进程,它通常需要在引导系统时从启动文件 rc.m 或 rc.inet2 中执行。

用于提交,删除,监视和控制打印作业的标准命令是:

- lpr 这是提交打印作业的命令。它将文档复制到打印机的 Spool 目录中,供 lpd 取出和打印。
- lpq 打印作业提交后,每个作业被赋于一个唯一的作业号。lpq 命令给出特定打印机的 Spool 目录中的所有作业和作业号的清单。
- lprm 使用 lpq 命令知道了特定作业的作业号后,就可以用 lprm 命令将它从打印机的 Spool 目录中移去或删除。如果命令中用(-)代替作业号,它将删除所有属于你的打印作业。
- lpc 这是打印系统的控制程序。它形成不同的状态并提供一些控制命令。作为普通用户只能用它显示打印机,假脱机队列以及守护进程的状态。其它命令可用来启动和停止打印机,假脱机和打印守护进程。需要作为 root 用户登录才能使用。

所有这些命令都对默认的 lp 打印机进行操作。如果需要指定其他打印机,可以用 -Pname 命令行开关,其中 name 是使用的打印机的名称。或者可以用打印机名称对 shell 的 PRINTER 变量赋值。

9.4.2 printcap

现在讨论如何设置不同的打印服务配置选项。这些工作都在下列文件中完成:

```
/etc/printcap
```

这是一个文本文件。如果作为 root 用户登录,就可以用 vi 编辑程序修改它。文件的内容称为 printcap 登记项。每一登记项只是一组配置选项。访问的每个打印机各占一登记项。

下面是两个典型的 printcap 登记项。第一项用于输出到本地的称为 lp 的无声点阵打印机,第二项通过网络访问 gregory 机器上的称为 hp 的 HP 4L 激光打印机:

```
lp|starsg10:\
        :lp=/dev/lp1:\
        :sd=/usr/spool/lp:\
        :lf=/usr/spool/lp/err-log:\
        :mx#0:\
        :sh:

hp|hp41:\
        :lp=:\
        :rm=gregory:\
        :sd=/usr/spool/hp:\
        :lf=/usr/spool/hp/err-log:\
        :if=/usr/local/bin/apsfilter:\
        :mx#0:
```

第一点要注意的是每一 printcap 登记项只占单独的逻辑行。其中(\ )字符表示逻辑行还未结束。

每一登记项的第一字段在行的开始,它包括使用的打印机名称清单,名称之间用(|)字符分开。

第一字段后面是一组选项和它们的相应的值。

我们依次讨论两个登记项。先讨论 lp 登记项。在名称字段后面的第二行是打印机的设备特殊文件名称(这里是 /dev/lp1),第三行将这一打印机的 Spool 目录指定为:/usr/spool/lp。第 4 行指定记录任何错误信息的 log 文件的路径名。第 5 行的 mx 指定打印文件的最大容量(以 1kb 计的块计数),0 表示对容量不加限止。最后一行中选项(sh)没有参数表示抑止标题页的打印。

现在看 hp 登记项,第 2 行指定打印机的设备特殊文件名称。由于这台打印机接在另一台机器上,这一选项中不给参数。第 3 行指定对这台打印机的打印请求应该送到那一台计算机;这里是 gregory。第 4 行指定 Spool 目录。网络打印机也要用 Spool 目录,在文件通过网络传出以前保存文件。第 5 行指定 log 文件。第 6 行指定过滤程序(这里是 apsfilter),所有送给打印机的文档都要先经过它。过滤程序可以在正文打印之前进行必要的处理。这一过滤程序将 postscript 文件转换为能在 HP 4L 打印机上打印的格式,因为它不是 postscript 打印机。这种转换用 GNU Ghostscript 程序完成。最后一行表示对文件容量不加限止。

如果网络打印机接在你的机器上,必须确定那些远程计算机有权使用打印机。最简单的做法是在 /etc/hosts.lpd 文件中列出准予使用的主机名称清单。

上面的例子使你能设置简单的打印机配置。在 printcap 登记项中可以指定更多的选项，printcap(5)手册项中包含完整的说明。在打印机设置方面，你还需要读另一个 Linux 文档，printing-HOWTO 文档。

第十章　X-windows

X-windows 是 Linux 和所有 UNIX 类操作系统的标准图形接口。许多 Linux 发布都提供 X-windows 软件。它可以和其他软件包一起安装。X-windows 的选项特别多，刚开始安装时很容易搞错。所以，在安装和配置时要特别小心。

10.1　服务程序和客户程序

X-windows 系统本身实际是一项客户机/服务器的应用。它能在同一台机器上或者通过网络在不同的机器上运行服务程序和客户程序。

在 X-windows 系统内部服务程序用来控制输入/输出设备(如键盘，鼠标和屏幕)，而 X 客户程序则是要使用服务程序的输入/输出功能的应用程序。服务程序和客户程序之间借助 X 通信协议进行通信。

服务程序要在联接键盘，鼠标和显示器的机器上运行，而且随机器的硬件配置不同而不同。

在安装 X-windows 软件包时，要从不同的 X-服务程序中选择适合你的硬件配置的 X 服务程序。选择的依据是所用的图形显示卡。Linux 的 X-windows 支持大多数图形显示卡。

假定装的是普通 SVGA 卡，没有使用加速或增强卡。就应选择下面的服务程序进行安装：

```
XF86_SVGA
```

服务程序装在 /usr/X11R6/bin 目录下，并且应该和同一目录下的 X 命令进行连接。在多数情况下，这项任务已由安装 script 完成。要是没有，可以用下面的命令实现连接。

```
ln  -sf  /usr/X11R6/bin/XF86_SVGA  /usr/X11R6/bin/X
```

要改变使用的服务程序时，要重新运行上面的命令，并将命令中的服务程序名称用新的服务程序名称代替。

10.2　X 配置

选择服务程序后，下一项任务是对它进行配置。这需要建立配置文件：

```
/etc/XF86config
```

在做这件事以前，提醒一句。如果配置不当，可能出现下列情况：驱动显示器超出技术规范运行，因而导致显示器的永久性损坏。因此，设置 XF86config 文件时要小心从事。

配置文件本身由若干信息段组成。分别说明硬件的配置情况。它们是：

Files	指定 RGB 数据库和不同的字体文件的路径名。
Server flags	用来指定 X 服务程序选项。
keyboad	给出键盘的设置。
Pointer	给出使用的鼠标类型，鼠标使用的设备特殊文件名，串行鼠标使用的速率以及是否在两键鼠标上模拟第三键等。
moniter	指定显示器特性。如带宽，水平同步，纵向刷新频率(摘自显示器手册)。并给出适合于显示器的显示方式(mode)清单。
device	本段指定显示卡的技术细节。如果没有指定所有必要的参数值，X 服务程序能检查硬件找出未指定的项。
Screen	本段给出使用的显示卡，显示器和 X 服务程序等的综合说明。

在建立配置文件和设置它的内容时，XF86config 的手册页和下面文件的内容特别有帮助：

```
/usr/X11R6/lib/X11/XF86config.eg
```

上面的文件中包含 X 配置的例子。设置自己的 X 配置文件的最简单办法是复制这一文件，并对它作适当修改。但是，不要不加任何修改就将它用作你自己的配置文件。

10.2.1 Files 段

本段中将以类似下面的方式定义 RGB 数据库路径名(RgbPath)和字体文件路径名(FontPath)：

```
Section "Files"
    RgbPath   "/usr/X11R6/lib/X11/rgb"
    FontPate  "/usr/X11R6/lib/X11/fonts/75dpi/"
EndSection
```

对于 RgbPath 的定义，上面指定的路径名不需要改变。由于 X-windows 中有不同的字体文件，可能出现多个 FontPath 定义。要保证 /usr/X11R6/lib/X11/fonts 目录下每个字体文件在这里都有相应的 FontPath 定义。

10.2.2 Server Flags 段

本段用来指定服务程序的全局特征。一般情况下，使用默认值就行了，所以不必指定服务程序特征。即使本段是空的，它仍然需要出现在配置文件中：

```
Section "ServerFlags"
EndSection
```

10.2.3 Keyboard 段

本段用来指定键盘的有关参数。一般讲,下面的设置适合多数情况,不必加以修改:

```
Section "Keyboard"
    Protocol  "Standard"
    AutoRepeat 500 5
EndSection
```

10.2.4 Pointer 段

本段用来指定鼠标(或跟踪球等)的有关参数。先说明鼠标的类型。接着说明鼠标的设备特殊文件名。对串行鼠标,下一项说明速率(标准是 1200 位/秒)。最后说明和指示器有关的一些特征。如 Emulate3Buttons 选项说明在两键鼠标上模拟三键鼠标的中键。

典型的 Pointer 段如下:

```
Section "Pointer"
    Protocol    "Microsoft"
    Device      "/dev/mouse"
    EBEndSection
```

10.2.5 Monitor 段

本段用来指定监视器特性。将本段中的参数设置正确是至关重要的。X 服务程序将检查这些参数以保证在驱动显示器时不超出这里给定的技术规范。

本段的重要参数值是带宽(Bandwidth),水平同步(HorizSync)和纵向刷新频率(VertRefresh)。这些可以直接从显示器手册中查到。还有 modeline 项,用来指定监视器使用的每种分辨率:

```
Section "Monitor"
    Identifier    "My monitor"
    Bandwidth     65
    HorizSync     31 - 49
    VertRefresh   56 - 87
    ModeLine  "640x480"  25   640   672   768   800   480   491   493   525
    ModeLine  "800x600"  36   800   864   936  1024   600   600   602   625
    ModeLine  "1024x768" 65  1024  1088  1248  1344   768   768   777   806
EndSection
```

上例仅仅用于说明。任何情况下,都不应不加修改地照搬这些参数值。

Identifier 定义监视器段的标号,供以后从文件中访问 monitor 段用。标号可以是任意的字

符串。这里用的是 My monitor

本段中的 modeline 参数是 X 配置文件中最难确定的部分。这些项给出不同的可用分辨率与显示器技术参数之间的关系。在下面的实例文件中给出许多常用 modeline 项：

```
/usr/X11R6/lib/X11/doc/modeDB.txt
```

只要有详细的硬件资料，这些值可以算得出来。但这已超出本书的范围。

如果在上面的文件中找不到合适 modeline 项，还可以阅读下面的文档：

```
/usr/X11R6/lib/X11/doc/VideoModes.doc
```

每一 modeline 项在双引号内给出标号，供以后访问这些项用。按照惯例这些标号应便于记忆，但并不一定非这样做不可。

10.2.6 Device 段

本段说明使用的显示卡和它的特性。一开始，不必在这段中设置许多值。因为 X 服务程序能检查硬件为它自己找出大部分值。你只须提供 Identifier 的定义。它是本段的标号，供以后访问本段使用。

在使用 X 服务程序检查硬件后(这将在后面说明)，就可以根据检查结果填写本段中的其他参数，得到的结果可能像下面那样：

```
Section "Device"
    Identifier    "My video card"
    Chipset       "et4000"
    VideoRam      1024
    Clocks        25.20 28.32 32.50 36.00 40.00 44.90 31.50 37.50
    Clocks        50.30 56.70 64.90 72.00 80.00 89.80 63.00 75.10
EndSection
```

10.2.7 Screen 段

从前面各段中提取标号，将它们集中放在最后一段中：

```
Section "Screen"
    Driver        "SVGA"
    Device        "My video card"
    Monitor       "My monitor"
    Subsection    "Display"
        Depth         8
        Virtual       1024 1024
        ViewPort      0 0
```

```
        Modes          "800x600" "1024x768" "640x480"
    EndSubsection
EndSection
```

Driver 定义使用的 X 服务程序。SVGA 表示 XF86_SVGA 服务程序。Device 和 Monitor 使用前面定义的符号名称,供以后访问相应的段使用。

然后,在 Display 子段中给出显示的更多的细节。其中 Depth 指定每个像元使用的位数。Virtual 指定虚屏幕的像元数。虚屏幕的大小应能放入显示卡的显示内存中。View Point 表示实际屏幕的左上角在虚屏幕中的对应位置。如果实际屏幕小于虚屏幕,可用鼠标控制移动虚屏幕。最后 modes 给出使用的显示方式清单。当 X-windows 开始运行时,清单中第一种方式作为默认方式。

在 X-windows 运行时,其他显示方式可以用 Ctrl-Alt-Plus 和 Ctrl-Alt-Minus 两种键组合来切换。其中 Plus 和 Minus 是键盘上的 + , - 符号键。

10.2.8 检查硬件

现在该启动 X 服务程序了。先用它来检查硬件。从检查结果中提取所需的信息,用来填写 Device 段的细节。可以用下面的命令检查硬件:

```
$  X  -probeonly
```

下面的清单是它的部分输出:

```
XF86Config: /etc/XF86Config
(**) stands for supplied, (--) stands for probed/default values
(**) Mouse: type: MouseSystems, device: /dev/mouse, baudrate: 1200
(**) SVGA: Graphics device ID: "ET4000"
(**) SVGA: Monitor ID: "Tatung XGA"
(**) FontPath set to "/usr/X11R6/lib/X11/fonts/misc/,/usr/X11R6/lib/
X11/fonts/Type1/,/usr/X11R6/lib/X11/fonts/Speedo/,/usr/X11R6/lib/X11/
fonts/75dpi/"
(--) SVGA: ET4000: Initial hibit state: high
(--) SVGA: chipset:  et4000
(--) SVGA: videoram: 1024k
(--) SVGA: clocks: 25.14 28.32 32.43 35.93 39.95 44.85 31.44 37.52
(--) SVGA: clocks: 50.28 56.55 64.86 71.96 79.97 89.75 62.94 74.91
(--) SVGA: Maximum allowed dot-clock: 90.000 MHz
(**) SVGA: Mode "800x600": mode clock = 36.000, clock used = 35.930
(**) SVGA: Mode "640x480": mode clock = 25.000, clock used = 25.140
(**) SVGA: Mode "1024x768": mode clock = 65.000, clock used = 64.860
(**) SVGA: Virtual resolution set to 1024x1024
(--) SVGA: Generic SpeedUps selected (Flags=0x30)
(--) SVGA: ET4000: SpeedUps selected (Flags=0xf)
```

在输出的结果中,前面标有(* *)的行表示:这是已经在 XF86_ Config 文件中指定的值。前面标有(- -)的行表示:这是由服务程序检查硬件找到的值。从中可以找出所要的信息来填写 X 配置文件的 Device 段。如果这些值已填到配置文件中,X 服务程序在以后的运行中就不再检查硬件来找出这些值。

10.3 启动 X-windows

一旦 XF86_Config 文件已经设置完毕,就启动 X-windows。在非 Linux 系统中,用 xinit 启动 X-windows。在 Linux 系统中,用下面的命令来启动:

```
$ startx
```

实际上,它还是运行了 xinit。在 X 初始化的过程中,xinit 还将运行在你的起始目录下的 xinitrc 文件中的任何命令。如果没有个人的 .xinitrc 文件,xinit 命令将搜索下列文件:

```
/usr/X11R6/lib/X11/xinit/xinitrc
```

并执行这一文件的内容。

10.3.1 .xinitrc

.xinitrc 文件用来确定 X-windows 显示的最初的配置,并使一些应用程序开始运行,再启动 X-windows。它也用于确定最初的屏幕布局,指定最初运行的一些应用程序的窗口放在屏幕上什么位置。例如:

```
#!/bin/sh

xterm -geometry 80x43+220+0 -bg gray -fg black &
exec fvwm
```

文件中的第一行指出文件的其余部分应该用 /bin/sh shell 程序而不是用当前的 shell 程序进行解释。第 2 行使 xterm 程序得到执行。这是一个标准的终端模拟程序。它同时提供 DEC VT102 正文终端和 Tektronics 4014 图形终端的模拟。后面的 & 字符是必要的。它将这条命令放到后台的执行。使得 script 的其余命令不必等待 xterm 先结束才能运行。本例中最后一条命令启动窗口管理程序(这里用的 fvwm)。有许多窗口管理程序可供使用,每一种分别提供外观和感受稍有不同的 X-windows 环境。

在启动 shell 和窗口管理程序的命令之间可以启动其他的应用程序。一般讲,其他命令行后面也应和 xterm 行一样加上 & 符号。注意,启动窗口管理程序的那一行并没放到后台去执行。它运行 exec 命令,用窗口管理程序来取代当前的进程。

当窗口管理程序结束时,X 服务程序停止运行。X-window 对话过程也随之结束。通常用菜单选项来完成这一操作。可以直接从键盘上按 Ctrl-Alt-Backspace 来停止服务程序的运行。

10.4 fvwm

fvwm 窗口管理程序提供虚窗口系统(virfual windows system)。通过单独的页面管理程序(pager)窗口访问多个桌面。当 fvwm 运行时,它搜索起始目录下的配置文件 .fvwmrc。

如果这个文件存在,它将用来设置窗口管理程序的运行环境的许多细节。如窗口背景的颜色,激活和未激活窗口边框的颜色。在切换桌面时,那个窗口随着移动,那些窗口留在后面。它也允许指定一些重要应用程序的窗口始终不被复盖,不管窗口在桌面上如何堆积,总能看到它们。它也允许你结合鼠标键(和键盘上的某些键)建立自己的菜单和操作。当你选择窗口边框的不同部分和背景时使用它们。

下例给出 .fvwmrc 文件的一些典型的内容。所有选项的各种细节可参阅 fvwm 命令的手册页:

```
# set up the colors
StdForeColor          Black
StdBackColor          SteelBlue
HiForeColor           Black
HiBackColor           red

# set up the desk top and pager
DeskTopSize 3x2
DeskTopScale 32
Pager 0 0

# list the windows that don't get a decorative border
NoTitle xclock

# list the windows that follow when changing desktop
Sticky xclock

# list windows that "stay on top" of the window stack
StaysOnTop xclock
```

同样用 .fvwmrc 来指定沿窗口边框或在桌面背景上按一个鼠标键所对应的操作。常用的选项有:在桌面上移动窗口,改变窗口大小,改变窗口堆积顺序以及从 fvwm 退出(同时退出 X-windows)。fvwm 菜单的典型说明如下:

```
Popup "Window Ops"
    Title     "Window Ops"
    Move      "Move Winodw"
    Resize    "Resize Window"
    Raise     "Raise Window"
```

```
    Lower     "Lower Window"
    Iconify   "(De)Iconify Window"
    Stick     "(Un)Stick Window"
    Nop       ""
    Destroy   "Destroy Window"
    Delete    "Delete Window"
    Nop       ""
    Title     "Exit"
    Quit      "Exit Fvwm"
EndPopup
```

其中第一列指定菜单选项进行的操作。第二列指定屏幕上菜单显示的正文。

鼠标键和菜单的结合方式用类似以下的语句作出规定：

```
# Button   Context   Modifiers   Function
Mouse 1      R           N        PopUp "Window Ops"
```

上例说明：当光标处于桌面背景(根)窗口的任何位置时，按鼠标左键将弹出“Window Ops”菜单。

10.5 xterm

在 X-windows 系统中有几种终端模拟程序可供使用。最常用的是 xterm。为了同时提供 VT102 和 4014 两种终端模拟，xterm 提供了两个独立窗口。可以同时看到两个或其中的一个。即使同时看到两个窗口，每次也只能用其中一个进行输入和输出。

每个窗口都有一组菜单和它结合在一起。这些菜单的激活办法是将鼠标光标移到窗口边界内，然后按下 Ctrl 键和一个鼠标键。

在 VT102 窗口内有 3 个菜单可供使用。每个鼠标键对应一个。左键显示主要选项菜单，中键显示 VT 选项菜单，右键显示 VT 字体菜单。

VT 选项菜单提供控制选项，最有用的是：

- 启用和禁用窗口的滚动条。滚动条允许你将窗口超出屏幕的部分拉回来。
- 提供终端恢复功能。当你向终端送入无效的控制字符时，终端可能处于不能使用的状态。这些功能使终端恢复正常状态。
- 显示和激活 Tektronics 4014 窗口。

在 4014 窗口内，有两种菜单，主要选项菜单和 VT102 相同。Tek 选项菜单用来控制显示字符的大小以及显示和激活 VT102 窗口。

第十一章 通 信

计算机领域新的最重要的发展既不是机器的规模，也不是处理器的速度和类型，而是将不同的计算机连在一起。即使这些机器分散在世界各地，也能共享数据和资源。

从 Linux 连接其他系统的能力看，包括连接其他 Linux 系统和非 Linux 系统，它都毫不逊色。所有流行的标准通信方式都可以在 Linux 系统上使用。

作为系统管理员，为了能在你的机器和外部世界之间建立起通信链路，需要掌握许多知识。特别是，可能承担起安装和配置串行、并行链路或以太网连接的任务以及配置机器使它能在这些链路上运行的任务。

11.1 独立的机器

即使你的机器不和任何其他机器连接，仍然有许多软件需要利用 socket 软件包作为进程之间的通信方式。Linux 支持 socket 软件包，这是在两台不同机器上的进程之间通过网络相互进行通信的标准方式。虽然 socket 程序假定两个通信的进程不在同一台机器上，但并不妨碍它作为同一台机器上的两个进程之间的通信方式。

为了做到这一点，需要进行适当配置。首先要做到在安装系统时已经装了网络软件包，如果使用 slackware 发布进行安装，网络软件包含在 N 软盘组中。

其次，在建立新内核时，在运行 make config 命令的过程中，只需要包括基本的网络选项。除非你实际安装了网卡，否则你不应该选择任何网卡。

最后，对许多发布讲，只要运行 netconfig 命令和回答几个简单的问题就可以了。

对一台独立的机器讲，netconfig 只要三条信息，其中包括：

hostname	这是你的机器的名称（严格讲，它是网络接口的名称）。名称可以是你喜欢的任何单字。一些人选用宠物的名字，小说人物或角色扮演者的名字等。譬如说，wendy。
domain name	这是机器连接的网络的名称(或域名)。如果你的机器是独立的，网络名称的选择是任意的。最简单的方式也许用一个单字，如 mynet。
loopback	这是独立机器的选项。独立机器只能用本地回送(local loopback)。你对这一项回答 yes 就成。

到此为止，netconfig 程序已经有了配置本地回送所有必要的信息。它不再进行提问。

你的机器的完整网络名称由主机名和网络名组成。例如：

```
wendy.mynet
```

11.2 TCP/IP

如果使用的 Linux 发布没有 netconfig 命令。需用手工配置网络软件；或者要配置你的机器

使它能通过以太网卡在网上运行,你就要提供更多的信息。不仅仅是主机名和网络名。

hostname 像前面一样,主机名只是一个单字,用来表示你连到网上的机器。如果你将机器连到已有的网络上,可能要遵循本地的命名规则。在这种情况下,应向本地网络管理员请教。

domain name 如果你的机器连到已有的网络上,要用已有网络的名称或域名。这也需要向本地网络管理员请教。

IP address 我们利用网际协议(Internet Protocol,IP)在 TCP/ IP 网上传输数据。连到 TCP/IP 网(包括 Internet)上的每台机器都用一个唯一的数字对它进行编址。这个数字称为这台机器的 IP 地址。IP 地址是一个 32 位的数字,通常用 4 个分开的字节表示。并写成用句点分开的 4 个十进制数。例如,199.38.231.66。IP 地址的前面部分是网络地址,后面部分用来表示网络中特定的机器(主机号)。

network mask 网络表征码是一个 4 字节的数。它用来确定 IP 地址中网络地址占的位数和主机号占的位数。网络表征码前面置 1 的位用来确定网络地址的实际长度。后面置 0 的位用来确定主机号的长度。用 IP 地址的第一字节的值就能推算出它的表征码的值:

IP 地址范围	网络表征码
127 以下	255.0.0.0
128 到 191	255.255.0.0
192 以上	255.255.255.0

network address 网络地址是一个 4 字节的数。IP 地址和网络表征码按位进行逻辑乘(AND)就得到网络地址。例如,设 IP 地址为:195.123.48.12,从第一字节的值为 195 可知网络表征码应为 255.255.255.0。所以,网络地址应为 195.123.48.0。

broadcast address 广播地址也是一个 4 字节数。用网络地址和网络表征码的反码按位进行逻辑加(OR)得到广播地址。继续上面的例子,网络地址是 195.123.48.0,网络表征码的反码是 0.0.0.255,广播地址是 195.123.48.255。

gateway address 网络连接机(网关)地址。这是连接到网络的一台机器的 IP 地址,通过这台机器可以访问其他网络上的机器。网络连接机只少连到两个网络上。必要时,它能在网络之间转移网络通信量(network traffic)。如果你的机器连在已有的网络上,应从网络管理员那里得到这个地址。

name server address 域名服务器是一台机器。将每对域名和主机名转换为 IP 地址是域名服务器提供的功能之一。小型网络的设置用不着域名服务器。增加一台机器到已有网络上,可以从本地管理员那里取得域名服务器的 IP 地址。

有了这些名称和数字以后,就可以进行网络设置。如果有 netconfig 程序,用它进行设置是最简单的选择。当它提问时,提供必要的细节就可以了。如果没有 netconfig 程序,自己来配置网络也并不困难。

11.2.1 手工配置网络

注意,当将网卡加到机器上时,需要重建 Linux 内核。并将使用的网卡类型加到内核配置中去,在运行 make config 时提供网卡的类型信息。

如果需要运行连网软件,当第一次引导系统时,需要启动连网系统(networking system)。

这包括用 ifconfig 命令设置机器的网络接口，用 route 命令设置内核路由表，然后执行提供各种网络服务的守护程序。

遗憾的是：当设置启动连网系统的文件时，各种 Linux 发布没有统一的标准。如这些文件叫什么名称，它们放在什么位置等。但是多数 Linux 发布将这些存放在 /etc 目录下的某个位置。并对有关文件给以类似下面的一些名称，如 rc.net，rc.inet 或 rc.inet1 和 rc.inet2。

在引导系统时，执行这些文件和其他 rc 文件。系统中第一个运行的进程是称为 init 的 1 号进程。这一进程周期地检查 /etc/inittab 文件，从中找出在不同的环境下它要做的工作。执行相应的 rc 文件也是它的任务之一。有时，将 init 配置成直接执行网络配置文件。有时，init 只执行另一个 rc 文件，并由这一 rc 文件执行网络设置文件。不管用那一种方式，当这些文件被执行，要完成一些特殊的任务。

11.2.2 设置主机名

第一项任务是设置你的机器的主机名。用 hostname 命令完成这一任务。需要在每次引导机器时执行这条命令。一般的做法是：由 init 执行一个 rc 文件，并通过 rc 文件执行这条命令。命令的格式为：

```
# hostname grunthos
```

其中 grunthos 是要设置的主机名。也可以不直接给出主机名，一些 Linax 发布使用下面的方式设置主机名：

```
# hostname 'cat /etc/HOSTNAME | cut -f1 -d.'
```

在 /etc/HOSTNAME 文件则存有完整的网络名：

```
$ cat /etc/HOSTNAME
grunthos.staffs.ac.uk
```

对自动配置的程序讲，这是比较简单的安排方式。当准备主机名时，只要重写/etc/HOSTNAME 文件的内容，而不必去修改配置文件中带主机名的行。

11.2.3 连网启动文件

从现在开始，在下面的讨论中假定系统使用下列网络启动文件：

```
/etc/rc.d/rc.inet1
/etc/rc.d/rc.inet2
```

即使你的文件不用这些名称，它们要完成任务还是一样的。

在 rc.inet1 文件中要完成的第一项任务是用下面的一组命令配置回送设备(loopback device)，其中包括了 ifconfig 和 route 命令

```
# ! /bin/sh

/sbin/ifconfig  lo  127.0.0.1
/sbin/route  add  -net  127.0.0.0
```

如果 ifconfig 和 route 命令不放在 /sbin 目录下，需要修改这些行来适应你的目录结构。

在上例中，第一行说明用什么 shell 程序来执行其余的部分，ifconfig 用来指定本地回送接口的名称（lo）和它的 IP 地址（127.0.0.1）。route 命令将网络地址加到内核路由表中。这是本地回送地址可属的网络（127.0.0.0）。这样 IP 数据包能正确送到本地回送地址。

如果你安装了以太网卡，也需要对网卡进行配置。相应地在 rc.inet1 文件中应增加类似下面的一些行：

```
IPADDR = "ip.ip.ip.ip"
NETMASK = "nm.nm.nm.nm"
NETWORK = "nw.nw.nw.nw"
BROADCAST = "bc.bc.bc.bc"
GATEWAY = "gw.gw.gw.gw"

/sbin/ifconfig eth0 ${IPADDR} broadcast ${BROADCAST} netmask ${NETMASK}
/sbin/route add -net ${NETWORK} netmask ${NETMASK}

/sbin/route add default gw ${GATEWAY} metric 1
```

很明显，你需要将上面的 ip，nm，nw，bc 和 gw 的值用你自己的正确地址值代替。

其中 ifconfig 命令将第一个以太网地址 eth0 和 IP 地址结合起来。第一条 route 命令在内核路由表中增加一项，说明本地网的网络业务边界（traffic bound）应送过这一接口。最后一行说明本地网以外的所有网络业务的路由信息。如果只有本地网，没有网络连接机，就可以省去最后一行。

设置了网络接口和路由表后，下一步是执行与网络应用有关的守护程序和服务程序。在我们的例子中，这些事用 rc.inet2 文件来实现。

在简单的情况下，只有两个服务程序要通过 rc.inet2 文件来执行。它们是 syslogd 和 inetd。前者用来记录系统信息和错误信息。这些信息放在 /var/log/message 文件中。后者使你的机器能提供一系列服务：包括 telnet，ftp，finger 和 rlogin 等。在后面会讲到这些。

运行这两个守护程序的 rc.inet2 文件的内容如下：

```
# Start the syslogd daemon.
if [  -f /usr/sbin/syslogd ]
then
   /usr/sbin/syslogd
fi
```

```
# Start the INET SuperServer
if [ -f /usr/sbin/inetd ]
then
   /usr/sbin/inetd
fi
```

在一些 Linux 发布中，syslogd 和 inetd 程序可能不放在 /usr/bin 目录下。因此要对 rc.inet2 文件的内容作相应的修改。

还要建立或使用一些文件来指定 IP 名称和地址。如果你有并使用 netconfig 程序，它将为你建立和设置这些文件。这 4 个文件是：

```
/etc/networks
/etc/hosts
/etc/host.conf
/etc/resolve.conf
```

下面依次说明这些文件的用途和它们的典型内容。

11.2.4 /etc/networks

这一文件包含网络名称和它的网络地址的清单。当需要将一个完整的网络的路由加到内核路由表中去，你给的是网络名而不是网络地址。在这种情况下，route 命令就要从这个文件中找对应的网络地址。这一文件的典型内容如下：

```
loopback        127.0.0.0
phil-net         194.61.21.0
```

如果在 route 命令中直接使用网络地址，就没有必要将相应的项加到 networks 文件中。

11.2.5 /etc/hosts

这个文件包含机器的 IP 地址和 IP 地址名称的清单。对大型专用网和 Internet，网络主机名称由域名服务器转换为相应的 IP 地址。但是，对没有域名服务器的小型网，或在引导系统时，大型网的域名服务器还未运行之前，都要用到 hosts 文件。从中找到 IP 名称和实际地址之间的转换。文件的典型的内容如下：

```
127.0.0.1       localhost
194.61.21.50    grunthos.sfatfs.ac.uk   grunthos
```

这是说：localhost 主机的 IP 地址是 127.0.0.1，而 grunthos.stafffs.ac.uk 主机的 IP 地址是 194.61.21.50。在文件的这些行中，还允许给出主机的别名。第 2 行说明：grunthos 别名可以作为缩写的主机名使用。

对小型独立网络不需要域名服务器。网上的每台主机将用它自己的 hosts 文件进行主机

名到 IP 地址的转换。

11.2.6 /etc/host.conf

这一文件用来控制系统如何使用和以什么顺序使用不同的域名服务。/etc/host.conf 文件的典型内容如下：

```
order  hosts,  bind
multi  on
```

第一行说明：当一个进程请求将主机名转换为 IP 地址时，这一请求将按下面的顺序进行处理。先查阅 /etc/hosts 文件。当在这一文件中没有找到所需的信息时，才使用 bind 服务。也就是用本地域名服务程序来查找所需的信息。本例中第 2 行表示：允许在/etc/hosts 文件中列出的主机名有多个 IP 地址。

11.2.7 /etc/resolv.conf

这一文件主要用来指定域名服务器清单。这些服务器用来将主机名转换为 IP 地址。同时指定附加在主机名后面的域名。当 bind 第一次搜索没有找到相应的 IP 地址时，将域名和主机名放在一起再次进行搜索。这一功能有助于在本地网上只用主机名也能找出正确的 IP 地址。

典型的文件内容如下：

```
domain  staffs.ac.uk
nameserver  194.61.21.1
```

当指定多个服务器时，每个服务器在 /etc/resolv.conf 文件中各占一行。行的顺序决定使用服务器的先后顺序。

不能指望本节包含设置网络的全部知识。但前面的说明适用于多数简单的网络配置。为了将你的机器加到更复杂的网上去，应请教网络专家。

11.3 PPP

除了用本地回送和以太网进行联网外，还有其他的点对点的链路方案。在使用 RS232 串行接口或 centronic 并行接口的一对机器之间进行连接。

第一个讨论的方案，也许是最好的点对点通信方案。它通过标准的 RS232 串行链路进行操作。PPP (Point to Point Protocol) 表示点对点协议。它是标准的 Internet 协议。

PPP 软件分成两个部分：第一部分建在内核中。就是说链路两端的机器都要在内核中配置 PPP 软件。这一点只需要在运行 make config 时对 PPP 选项回答 yes 就行了。第二部分是一个称为 pppd 的守护程序。这是 PPP 链路建立后需要分别在两台机器上运行的程序。

最常用的点对点通信链路是通过调制解调器的拨号连接。它的设置将在下面讨论。

假定你家中有一台 Linux 机器，需要拨号接通一台远程的，接入 Internet 网的 Linux 机器。

当链路接通后，就能从家中对 Internet 进行全面的访问。

11.3.1 远程站的设置

先考虑远程站的设置。最简单的做法是在远程的机器专为拨号连接建立 ppp 登录项。开始，先在口令文件中按下面的形式增加一新行：

```
ppp: off :700:700:ppp acount:/home/ppp:home/ppp/ppplogin
```

应该用自己的 ppp 帐号名称，uid，gid 和起始目录来填写这一行。然后，为这一帐号建立起始目录 /home/ppp（可以用自己的名称来建立起始目录），然后改变目录的所有权。

```
#  mkdir  /home/ppp
#  chown  ppp .  /home/ppp
```

注意在新加的行中，将下列程序作为登录的 shell：

```
/home/ppp/ppplogin
```

实际上，它不是 shell 程序。而是在远程机上用来启动 pppd 守护程序的 script。它的典型形式如下：

```
#!  /bin/sh
exec  /usr/sbin/pppd  38400  modem  asyncmap  0  proxyarp \
      194.61.21.2:194.61.21.49。
```

需要对上面内容作些解释。第 1 行说明 script 的其余部分必须用/bin/sh 来执行。第 2 行最后的 \ 字符表示下一行是本行的继续。运行 exec 命令，它将当前的进程用指定的 pppd 进程代替。其余部分是 pppd 的命令行参数。

第 1 个参数（38400）是 pppd 驱动串行口到调制解调器的速率。modem 参数规定：本地端挂断电话时，pppd 应该用 RS232 接口的调制解调器控制线彻底断开链路。

下一个参数是 asyncmap 0，取决于对远程机的串行连接的具体细节。连接可能将 ASCII 控制码（0～31）用于特殊的用途。所以，这些代码不能直接作为数据包的一部分进行传送。这种情况下，需要告诉 pppd 不要直接用这些控制码，而用特殊的两字符的转义（escape）序列来代替它们。在 asyncmap 关键字后面的数字是一个用十六进制表示的 32 位数。每一位对应一个 ASCII 控制码。任何位置 1 表示不能使用相应的控制码。如果串行链路没有用控制码作特殊用途，那么 asyncmap 0 表示不需要对任何控制码进行转义处理。如果不给出 asyncmap 参数，默认值是对 32 个控制码都进行转义处理。

连接 TCP/IP 网络的以太网并不直接利用 IP 地址，而是利用以太网硬件地址来传递数据包。将 IP 地址转换为以太网硬件地址的协议称为地址解析协议（Address Resolution Protocol，ARP）。可以用 arp 命令显示机器的 ARP 表。它的典型内容如下：

```
# arp -a
Address          HW type          HW address          Flags   Mask
194.61.21.1      10Mbps Ethernet  08:00:2B:F7:E2:44   C       *
194.61.21.6      10Mbps Ethernet  08:00:2B:57:89:CE   C       *
194.61.21.252    10Mbps Ethernet  AA:00:04:40:0A:0C   C       *
194.61.21.50     10Mbps Ethernet  00:40:95:85:0C:B5   CMP     *
194.61.21.49     10Mbps Ethernet  00:40:95:85:0C:B5   CMP     *
```

送给 pppd 的 proxyarp 参数告诉它在远程机的 ARP 表中增加一项，列出本地机的 IP 地址和运程机的以太网硬件地址。当远程机器发现对你的机器的以太网硬件地址的请求时，它用自己的以太网硬件地址作出响应。它捡出送给你的机器的 IP 数据包，并通过点对点链路送给你的机器。上例中的 arp 表的最后两项是 proxyarp 项，两者都用远程机的以太网硬件地址。但是，分别对应点对点链路本地端的两台机器的 IP 地址。

送给 pppd 命令的最后参数给出远程机和本地机的 IP 地址。它们之间用冒号分开。在运行 pppd 时还可以指定其他选项。例如为了更加安全起见，可以使它具有验证功能。这些选项的细节请参阅 pppd 的手册页。

在远程机上最后要做的事是对 ppp 帐号设置合适的口令。

11.3.2 本地站设置

本地机器的设置比较简单。需要写一个 shell script，用在本地机器上运行 pppd 守护程序。然后，通过调制解调器接通远程机，并登录到 ppp 帐号，这件事的最简单做法是使用 pppd 和拨号程序 chat。

从 Linux 发布安装 ppp 软件包时，pppd 和 chat 以及其他有用的程序和 script 一起自动被装入。

连接远程机的典型 script 的内容如下：

```
/usr/sbin/pppd connect  '/usr/sbin/chat "" ATDT334566
CONNECT "" ogin: ppp word: PA55word' /dev/modem 38400
modem defaultroute -ip 194.61.21.49:194.61.21.233
```

这一 script 应该作为单独的长行送入文本文件，并用 chmod 命令使它成为可执行文件。这一 script 只运行 pppd 命令并传送给它不同的参数值。送给 pppd 命令的第一参数是 connect cmd 其中 cmd 表示运行建立调制解调器连接的命令或 script。

在本例中，cmd 是两个单引号之间的所有内容。其中包括：调用 chat 程序并将它运行时所需的参数传给它。送给 chat 程序的参数具体说明在连接调制解调器的串行线路上进行对话的内容。对话采用一对预期和发送(expect and send)字符串表示。作为预期的字符串，一对空的双引号表示没有预期什么。后面跟的发送字符串(ATDT334566)是 Hayes 兼容的调制解调器命令，告诉本地调制解调器拨通远程调制解调器的电话号。

下一对(pair)预期/发送字符串是 CONNECT "" 。这是说，预期接收 CONNECT 字符串，当收到这一字符串后，在本地机器上换行（作为发送字符串，空的双引号表示在本地机器上换行)。远程调制解调器在回答本地调制调器的振铃和协调运行速率后送出 CONNEC 字符串。

远程机应在新行上给我们送来 login: 提示符，并等待我们送出登录名。所以，下一对预期/发送字符串是 ogin: ppp。使用 ogin: 允许登录提示符用大写或小写字母开头。所以，这对字符串表示预期从远程机器上接受以 ogin: 结尾的字符串。当收到这一字符串后，送出 ppp 登录名作出响应。

送出登录名后，远程系统应要求相应的口令。因此，最后一对字符串表示预期从远程机接受以 word: 结尾的字符串，并给 ppp 登录帐号送去口令（本例中为 PA55word）作为响应。

如果一切按计划顺利进行，远程机现在应该运行 ppplogin script 了。正像你已经看到的，它将在远程机上运行 pppd 守护程序。

在本地的启动 script 中送给 pppd 的后面三个参数具体说明本地调制解调器使用的设备特殊文件，调制解调器的对话速率以及在串行线路上使用调制解调器控制线的指令。

下一个参数（defaultroute）告诉 pppd 在内核路由表中增加默认的路由，将没有在路由表中说明的任何机器的网络数据包通过点对点链路送给远程机。只在内核路由表中尚未包含默认项时，defaultroute 才起作用。这表示为了利用这一选项，rc.inetd 文件中不应包含上例中的最后一行。它对网络连接机设置默认的路由。在点对点链路中，远程机将作为连接外部世界的网络连接机。当建立了链路后，pppd 将默认的路由设为这台机器。送给 pppd 的最后一个参数（-ip 194.61.21.49:194.61.21.233）告诉 pppd 不用远程机的 IP 地址，而用指定的一对 IP 地址，它们分别对应本地和远程机器的 IP 地址。

一旦建立起点对点的链路后，你的机器就好像直接和 Internet 连接一样进行操作。当你用完链路后，只要执行 ppp-off script 就能断开链路。系统恢复到原来的状态，为下次建立链路作好准备。

在 pppd 和 chat 的手册中和 Linux 的 ppp - HOWTO 文档中还可以找到点对点链路的许多其他信息。

11.4 其他连接方法

在我们已经考虑的连网方式外，还有其他的方式，虽然它们没有前面讲到的方式那么重要。

11.4.1 SLIP

SLIP 是串行线路的 IP 协议，它逐渐被 ppp 取代。主要原因是 ppp 是 Internet 的标准协议，而 SLIP 则不是。从概念上讲，SLIP 和 ppp 非常类似，但 SLIP 并不完善而且容易出错。

为了使 Linux 支持 SLIP，在建立内核时，应对它进行配置，使它建在内核中。通常这只要在设置内核配置时，对 SLIP 选项回答 yes 就行了。

在内核中建立了 SLIP 软件后，可以运行 dip 程序。它对 SLIP 的作用就像 pppd 和 chat 的组合对 PPP 的作用一样。

使用 dip 的方法是建立 script 文件，在文件中具体说明需要 dip 完成的操作。在 dip 的手册页中可以找到建立和修改 script 文件的细节。

11.4.2 PLIP

PLIP 是并行线路的 IP 协议。它可以通过标准的并行打印机口连接两台计算机。这种方

案对一些场合下可能有用。例如，你有一台没有标准网络接口的笔记本或膝上型计算机，你想将它连到家中或办公室中的台式机上时，就用得上它。

PLIP 也是 Linux 发布的选项，需要将它编译到内核中去才能使用它。一旦它可以在内核中使用，设置软件的工作非常简单。最麻烦的事还是弄到连接两台机器的电缆，你可以购置电缆，也可以自己焊接电缆。

如果你自己做电缆，需要买一对 25 针的 D 型插头，插到打印机并行口上和一条长度适当的多芯电缆（只少要 12 芯的），并按下表进行连接：

```
Plug A      Plug B
    2  —  15
    3  —  13
    4  —  12
    5  —  10
    6  —  11
   10  —  5
   11  —  6
   12  —  4
   13  —  3
   15  —  2
   17  —  17
   25  —  25
```

两个插头上的其他插针不连接。

将两台机器连在一起后，为了使 PLIP 链路能够工作，应该在 rc.inet1 文件的后面增加两行：

```
/sbin/ifconfig plip0 la.la.la.la pointopoint ra.ra.ra.ra up
/sbin/route add -host ra.ra.ra.ra plip0
```

其中 la.la.la.la 是本地机的 IP 地址，而 ra.ra.ra.ra 是远程机的 IP 地址。

11.4.3 UUCP

在 Internet 出现之前，机器之间唯一的实际通信联系是在串行链路上用慢速调制解调器实现的。即使在那种情况下，仍然要求不同机器的用户之间传送 e-mail，仍然需要上载和下载文件。

UUCP 用来满足这些早期的要求。它在普通电话线上用调制解调器实现了简单的拨号网络。虽然 UUCP 使用了许多年，但仍然有大量的机器在使用它。

设置 UUCP 需要建立和配置一些文件。这些细节超出了本书叙述的范围。建议阅读 Linux 网络管理员指南中的 UUCP 设置基础。它已经成为 Linux 文档项目的一部分，许多 Linux 发布中都能看到这部分材料。

11.4.4 Kermit 和 seyon

除了网络通信接口外，还有许多程序为你提供连接和登录到远程机器的能力，并在你的登录对话过程中扮演哑终端模拟程序的角色。作为附加的特征，这类程序还提供上载和下载文件的能力。一般讲，上载文件表示将文件内容从本地机传送到远程机。下载文件则完成相反方向的传送，从远程机到本地机。

Linux 系统中容易取得的两个程序是 Kermit 和 seyon。kermit 程序直接在标准终端屏幕上运行，而 seyon 则需要 X-windows 环境才能运行。

从概念上讲，这两个程序非常相似。但它们的用户接口差别很大。为了体会这些程序的功能，下面将对 kermit 程序进行详细的讨论。

可以用几种不同方式使用 kermit 程序。作为开始，可以用交互方式使用 kermit。在这种方式下，它将重复显示提示符，接受用户命令并执行命令。

对一些选项，也可以用命令行方式运行 kermit。在这种情况下，运行程序并用参数指定需要程序完成的操作。kermit 完成指定的操作后回到 shell 提示符。

用它传送文件时，需要在两台机器运行 kermit 程序，两份拷贝都在你的本地机控制下运行。

为了做到这一点，在远程机上运行的 kermit 程序将被设为远程服务方式。它通过两台机器之间的通信链路接受和执行命令。

不管用那一种方式使用 kermit 程序，它都能读起始目录下的 .kermrc 命令文件。它通常用来设置 kermit 运行环境的默认参数。

通过调制解调器拨号连接从远程机将二进制文件传送到本地机的典型过程如下：

- 在本地机上用下面的命令运行 kermit：

```
kermit
```

  需要在 PATH 环境变量中增加包含 kermit 程序的目录名称

- 对远程机进行拨号连接：

```
set  modem  hayes
set  line  /dev/modem
set  speed  38400
dial  334566
connect
```

  这组命令不用解释。在 connect 命令执行后，将得到远程机的登录提示符。顺便提一下，当送入 connect 命令后，送入转义字符组（通常使用 ctrl-\ c）将使你退出终端模拟状态，回到 kermit 提示符。

- 登录到远程机后，找出要转送的文件，然后在远程机上运行 kermit。现在将从远程机上得到 kermit > 提示符。
- 为响应远程机的提示符，应送入下列命令：

```
server
```

这将使远程 kermit 进入服务程序方式，为传递文件作好准备。

- 现在应该送入本地的 kermit 转义字符组，它使你回到 kormit > 提示符。但这次来自本地的 kermit 程序。
- 现在可以发出传送文件的命令：

```
set file type binary
get datafile
```

第一条命令告诉 kermit 希望转送的文件的类型。kermit 允许传送两种类型文件：二进制（binary）数据文件和文本（text）文件。你要选择正确的文件类型。如果你没有把握确定文件的类型，就用 binary 类型，它的传送时间会长些。很明显，你要用实际传送的文件名取代 datafile。为了执行这些命令，本地 kermit 将命令传送给远程 kermit 服务程序。它将为你传送你要的文件。

- 当这一文件送结束后，你可以发出更多的 get 命令请求传送更多的文件。当你取得所有文件后，可以用下面的命令将远程 Kermit 从服务程序方式切换为交互方式：

```
finish
```

当服务程序停止运行后，将从本地 kermit 得到 kermit > 提示符。如果此时发出 connect 命令，它将使本地 kermit 回到终端模拟方式。将得到另一个 kermit > 提示符，但这次来自远程机。

- 送入下面的命令：

```
quit
```

将结束远程 kermit 的运行，远程机回到 shell 提示符。

- 当结束远程机上的终端对话过程时，应从远程机退出登录。然后，在本地机上送入转义字符组，回到 Kermit 提示符。最后送入 quit 命令，结束本地 kermit 的运行，并使你回到 shell 提示符。

上面的例子用来说明 kermit 程序的基本概念。kermit 还有许多选项。如果希望得到更多的信息，可以利用 kermit 的帮助功能。。从 kermit 提示符后送入 help 即能得到帮助信息。

第十二章　Internet

Internet 利用网际协议（IP）将世界范围内的许多独立的网络相互连接在一起形成单一的逻辑网络。

它为用户提供大量的服务，主要的服务包括：

e-mail	电子邮件使 Internet 用户可以在一对一的基础上和网上任何其他用户相互进行通信。
telnet	telnet 程序允许你在 internet 上任何一台你有帐号和口令的机器上登录。
ftp	文件传输协议（ftp）程序允许你在不同的机器之间传送文件。
archie	有助于找出在许多 ftp 网点上归档的文件的类型信息。
usenet news	相当于世界范围的电子公告牌系统。拥有数以千计的有趣的新闻组。几乎涉及任何话题。
gopher	这是另一项寻找和检索文档的服务。它的搜索程序称为 veronica。gopher 相当于 archie 和 ftp 的组合，但具有更复杂的功能。现在 gopher 服务在很大程度上已为 www 服务可代替。
world wide web	提供建立和显示超文本文档的准则。这些文档可以包含转移到其他文档的超文本连接，而后者可以位于 Internet 任何其他 www 服务器上。

本章将简单讨论每项主要服务。

12.1　e-mail

电子邮件是用户在一对一基础上进行通信的主要方式。这些用户可以在同一台机器上，也可以通过网络分散在不同的机器上。

从概念上讲，发送 e-mail 和发送普通邮件非常相似。任何人要想收到 e-mail 必须有 e-mail 地址，就像要收到信件必须有邮寄地址一样。

邮寄地址的全称包括：国家，地区，城市，街道，住房号和姓名。地址提供了传递邮件所必要的层次信息，使它越来越接近传递的对象。与此相似，e-mail 地址也有类似的层次结构。例如，我的 e-mail 地址是：

```
pc@soc.staffs.ac.uk
```

从层次的意义讲，这一信息应从右面读到左面。右边的第一项是国家代码。上例中，uk 表示联合王国（或英国）。除美国外，所有 e-mail 地址都有国家代码。美国则不同。没有国家代码，它就是默认的国家。在英国，我的 e-mail 地址在科学社区（ac）的 staffordshire 大学（staffs）内，而在大学内，我又在计算机学院（soc）。

当有人向我发送 e-mail 时，我的本地端的机器就像一个邮局，在那里接收和保存寄给我的任何邮件，直到我取走为止。与此相似，对发送信息的人讲，发送电子邮件的远程机也像一个

邮局。

从源机器到目的地的机器，e-mail 信息在传递过程中可能经过其他的机器，由于路径选择是自动的，信息发送者并不需要知道这些中间的机器。这些概念表示在图 12.1 中。

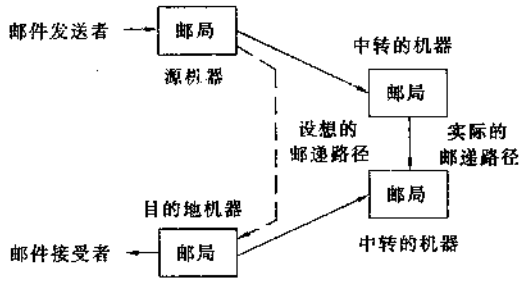

图 12.1　e-mail 从机器到机器的传送

一般讲，e-mail 地址具有下面的格式：

```
user@destination
```

发送者一般不需要知道和控制信息传递的路径。然而，有时愿意具体说明通过一组特定的机器来传递邮件。这可以通过指定目的地址（destination）和一台或几台接力的机器（relay）来实现

```
user%destination%relay1@relay2
```

按照上面的语法，送给 destination 机器上的用户 user 的 e-mail 信息，将先经过 relay2，再经过 relay1 机器。对机器的访问顺序自右至左，而且其中只包含一个@符号。邮件到达的下一台机器是@符号右边的机器。当邮件到达那里后，@符号及其右边的机器名称将从邮递途径中去掉。然后将最右边的%符号改为@符号，继续向前传送邮件。

到现在为止，我们说明了 e-mail 投递的传送部分：e-mail 如何从一台机器传到另一台机器。在 Linux 系统中，e-mail 的传送由 smail 或 sendmail 程序完成。对全面的 Internet 连接讲，这两个程序的配置相当费事。所幸的是，一些 Linux 发布带有简单的，可以用来配置邮件传送程序的 script。使这些程序能适应多数简单的运行环境。

如果要为自己设置 e-mail 传送环境，建议你使用这些用于配置的 script。也应查阅 Linux Mail-HOWTO 文档。其中有许多关于这些传送程序的常见问题和解决办法的说明。

除了传送 e-mail 的程序外，还要有阅读和投递（post）邮件信息手段，也就是 e-mail 的用户接口程序或邮件阅读程序。在 Linux 系统中，可以运行许多邮件阅读程序。主要的 Linux 发布提供的邮件阅读程序包括：

- elm;
- deliver;
- pine;
- mailx;

·metamail。

本人倾向于使用 elm,主要由于熟悉的缘故。其实,所有邮件阅读程序都完成相同的任务。

12.2 telnet

telnet 程序允许用户使用 TELNET 协议在机器之间进行通信。允许用户通过网络在远程机上登录是 telnet 的最普遍的应用。telnet 命令的一般形式如下:

```
$  telnet  host.domain
```

其中 host.domain 是要连接的系统的主机名的全称。如果这一命令执行成功,将从远程机得到 login: 提示符。

一旦登录到远程机上,telnet 程序的行为就像哑终端模拟程序。送入的任何字符都直接送到远程机上,这就是说,除非送入 telnet 的转义字符组,远程终端对话过程才被中断。telnet 的默认转义字符组是 ctrl-]。如果送入这一字符组,将从本地 telnet 程序得到下列提示符:

```
telnet>
```

提示符后只按 Enter 键,将重新连接到远程机上,使远程对话过程继续进行下去。

在 telnet 提示符后面可以送入许多命令,分别用来控制 telnet 对话过程的不同方面,telnet 手册页中对这些命令有详细的说明。例如,想改变转义字符组,可以送入下面的命令:

```
telnet>   set  escape  ^X
escape  character  is  '^X'.
```

在送入 set escape 命令后,只要送入以后使用的转义字符组即可(本例中为 ctrl-X)。telnet 将确认你的请求,然后继续远程对话过程。

所有的 Internet 服务都有服务程序使用的默认端口号。当客户程序请求特定的服务时,必需连接到相应的端口。可以从下面文件中找到默认端口号的清单:

```
/etc/services
```

telnet 的默认端口号是 23,telnet 程序在连接远程机的 telnet 服务时,会自动使用这一端口。然而,可以用本地的 telnet 程序对远程机上的任何其他端口进行哑终端连接,只要在最初的 telnet 命令中加上所要的端口就可以了:

```
$  telnet  host.domain  port
```

其中 port 将用所要的端口号代替

下面的例子中，telnet 直接与远程机上的邮件传送程序通信，以此来传递邮件信息。邮件传送程序在默认的 e-mail 端口上，用的是 25 号端口：

```
$ telnet grunthos 25
mail from: <pc>
250 <pc>... Sender ok
rcpt to: <saw>
250 <saw?... Recipient ok
data
354 Enter mail, end with "." on a line by itself
Hi
This is just a test message...

250 WAA00321 Message accepted for delivery
```

前面带行号的行是来自远程机的邮件传送程序的确认信息和提示信息，其余各行则是从本地机输入的命令和信息正文。

12.3 ftp

ftp 命令是标准的文件传输协议的用户接口。ftp 的用途是在本地机和远程机之间传送文件。运行 ftp 命令的格式如下：

```
$ ftp host.domain
```

其中 host.domain 是所要的远程机的主机名的全称。在命令行中，主机名的说明属于选项。如果指定主机名，ftp 将试图与远程机的 ftp 服务程序进行连接。如果没有指定主机名，ftp 将发出提示符，等待用户输入命令：

```
$ ftp
ftp>
```

在 ftp> 提示符后面送入 open 命令再跟以主机名，它将试图连接指定的主机。

不管用那一种方法，如果连接成功，需要在远程机上登录。如果在远程机上有帐号，可以通过 ftp 使用这一帐号并提供帐号的口令。在远程机上就有你使用的帐号的读和写的权限。这些权限决定了在远程机上能下载什么文件和将上载文件送到什么目录中去。

如果没有远程机的专用登录帐号，那么，许多 ftp 网点设有可以使用的特殊帐号。这个帐号的登录名为 anonymous（也称为匿名 ftp）。当使用这一帐号时，必须送入 e-mail 地址作为口令。

如果远程系统提供匿名 ftp 服务，使用这项服务将使你登录到特殊的，供公开使用的文件

读、写区。它专门提供了两个目录:pub 目录和 incoming 目录。前者包含远程网点供公众使用的所有文件,可以将供献给远程网点的文件上载到后一个目录中去。

一旦 ftp 在远程网点上登录成功,将从本地 ftp 程序得到 ftp > 提示符。现在可以自由使用 ftp 提供的命令。可以用 help 命令取得可供使用的命令清单。也可以在 help 命令后面指定具体的命令名称,来取得这条命的更多的细节。

最常用的命令有:

ls	列出远程机的当前目录;
cd	在远程机上改变工作目录;
lcd	在本地机上改变工作目录;
ascii	传送 ascii 文本文件;
binary	传送二进制文件;
hash	每传送一个 1024 字节的数据块显示一个 # 号;
get	从远程机传送指定文件到本地机;
put	从本地机将指定文件传送到远程机;
quit	断开与远程机的连接并退出 ftp。

从远程机下载二进制数据文件的典型对话过程如下:

```
$ ftp grunthos
Connected to grunthos
220 grunthos FTP server
Name (grunthos:pc): anonymous
331 Guest login ok, send your complete e-mail address as password.
Password:
230 Guest login ok, access restrictions apply.
Remote system type is UNIX.
ftp> cd pub
250 CWD command successful.
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 114
drwxr-xr-x       2 root    whell        1024 Oct 31 23:45 .
drwxr-xr-x       8 root    whell         1024 Mar 28 1995 ..
-rw-r--r--       1 root    root         14684 Oct 31 23:45 rog1
-rw-r--r--       1 root    root         96547 Oct 31 23:45 rog2
226 Transfer complete.
ftp> binary
200 Type set to I.
ftp> hash
Hash mark printing on (1024 bytes/hash mark).
ftp> get rog1
```

```
200 PORT command successful.
150 Opening BINARY mode data connection for rog1 (14684 bytes).
# # # # # # # # # # # # # #
226 Transfer complete.
14684 bytes received in 0.0473 secs (3e + 02 Kbytes/sec)
ftp> quit
221 Goodbye.
$
```

对话过程从运行 ftp 开始，并请求与 grunthos 主机连接。在收到一些信息后，接着收到提供登录名的邀请。如果在这时候只按 Enter 键，括弧中的内容（grunthos:pc）就构成登录名，可是我们要进行匿名 ftp 传送，所以用 anonymous 登录。接着要求我们提供口令，并提示用 e-mail 地址作为口令。屏幕上不回送口令，所以看不到送入的字符。在本例中，送入了下列内容：

```
password: pc@soc.staffs.ac.uk
```

成功地登录到远程系统后，我们改变到 pub 目录。然后列出它的内容。在决定下载 rog1 文件后，将 ftp 设为传送二进制文件，并在收到一个 1024 字节的数据块后显示一个 # 号标志。在下载文件后，退出 ftp。

如果要在自己的机器上运行匿名 ftp 服务，要注意保证系统的安全。一些 Linux 发布预装了匿名 ftp 软件，在系统安装后即能使用它。如果要用手工安装它，需要在 /etc/passwd 文件中增加 ftp 登录项。其中口号字段用 * 号表示。登录 shell 字段用/dev/false 表示，这样做为了防止用 ftp 帐号作为普通的登录。你还需要建立 ftp 的起始目录和一组子目录，如 pub 和 incoming 等。详细的情况请参阅 Linux NET-2-HOWTO 文档。

12.4 archie

ftp 为你提供了强有力的工具。但仍然留下一个基本问题，在世界什么地方才能找到你要的特定文件。这就是为什么要有 archie 的原因。

世界上许多网点设有 archie 服务程序。实际上，这些网点包含他们所接触到的所有 ftp 网点上的所有文件名称的数据库。如果你对其中一个数据库进行 archie 查询，你将得到包含所需文件的所有 ftp 网点以及在这些网点上到达这一文件的路径名清单。然后就可以用标准的 ftp 对话过程来检索你所要的文件。

世界上有许多 archie 网点，下面是一些网点的主机名和可在国家名称清单：

```
archie.au               Australia
archie.univie.ac.at     Austria
archie.belnet.be        Belgium
archie.bunyip.com       Canada
archie.cs.mcgill.ca     Canada
archie.funet.fi         Finland
```

```
archie.univ-rennes1.fr   France
archie.th-darmstadt.de   Germany
archie.ac.il             Israel
archie.unipi.it          Italy
archie.wide.ad.jp        Japan
archie.hana.nm.kr        Korea
archie.uninett.no        Norway
archie.icm.edu.pl        Poland
archie.rediris.es        Spain
archie.luth.se           Sweden
archie.switch.ch         Switzerland
archie.ncu.edu.tw        Taiwan
archie.doc.ic.ac.uk      UK
archie.hensa.ac.uk       UK
archie.sura.net          USA (MD)
archie.unl.edu           USA (NE)
archie.internic.net      USA (NJ)
archie.rutgers.edu       USA (NJ)
archie.ans.net           USA (NY)
```

为了利用这些网点,只要用 archie 作为登录名在表中列出的一台主机上进行远程登录即可。archie 服务程序将送回一些简单信息。对刚开始使用 archie 的人讲,最有用的是下面的命令:

```
archie> help ?
```

这将给出所有可用的命令清单。如果对清单中的一条命令请求帮助,它将给出更详细的解释。使用 help 命令后,可以按 ctrl-c 或 ctrl-d 回到 archie> 提示符。

用下面的简单命令就能使 archie 找到特定的文件:

```
archie>  find  posix.1
```

经过短时间搜索后,它将提供一个清单。其中包含已知的包含所需文件的所有网点以及在这些网点中到达所需文件的路径名。再用传统的匿名 ftp 找到所需文件就简单了。

注意:每个 archie 网点上的数据库只是这个网点对于世界上有什么文件的写照。因此,一个特定的文件不会在所有网点上都有记录。

12.5 Usenet 新闻

许多人用过公告牌系统(BBS)作为广泛的专题的论坛。usenet 新闻的设想是:它提供公告牌系统的功能。但是它必须在世界范围内,在更广泛的专题的基础上进行这种讨论。

讨论的专题称为新闻组(news groups)。利用新闻阅读程序,可以将稿件提交新闻组,也

投递稿件的总数,大多数新闻组都有一个经常更新的文档。而且定时进行投递(每半个月或一个月一次)。这个文档称为 FAQ,表示经常问到的问题。在投递任何问题给新闻组之前,你应经常查阅问题的答案是否已经包括在新闻组的 FAQ 文档中,或者在这领域的常用文档中。如果你失于检查,你将收到大量的邮件(以激动的语言)解释你的错误行为方式并建议你 RTFM(阅读好的手册)。

12.6 gopher

gopher 服务是很高级的。它将许多功能集中在一个软件包中,使你便于访问 Internet 各种不同的服务。

遗憾的是:很少 Linux 发布包含 gopher 客户程序。所以,要另外去找一份拷贝,才能使用它。这不像听上去那么严重,因为许多大的 Internet 的 Linux 镜像网点肯定有它的拷贝。例如,可从下面的网点用 ftp 取得它的拷贝。

```
src.doc.ic.ac.uk
```

也可以从 sunsite(sun 公司的网点)在下面的目录中找到 Linux 系统使用的 gopher 版本:

```
/unix/Linux/sunsite.unc-mirror/system/Network/info-systems
```

gopher 服务本身是由服务器网络提供的。每个服务器保存它自己的可在服务器上使用的文件和信息清单以及访问其他服务器资源的一组连接。

当你在本地机上运用 gopher 客户程序时,指定服务器的主机名称,这将形成进入 gopher 网络的起点。例如:

```
$  gopher  gopher.doc.ic.ac.uk
```

执行这条命令后,将从你选择的 gopher 服务器得到它的主菜单。它的典型的形式是一个带编号的选项的菜单,在其中一个选项旁边还有一个高亮度的指示箭头:

```
Root gopher server: gopher.doc.ic.ac.uk

      1. Welcome to the src.doc.ic.ac.uk gopher server.
-->   2. Gopher Services/
      3. astronomy/
      4. biology/
      5. computing/
      6. gnu/
      7. ic.doc/
      8. info/
      9. packages/
```

```
10. rfc/
11. science/
12. ukuug/
13. unix/
14. usenet/
15. weather/
Press? for Help,q to Quit, u to go up a menu     page:1/1
```

使用时，上、下箭头键使指示箭头在菜单中上，下移动，按 Enter 键选择箭头所指的菜单项，这将使你进入下一个菜单。从这一 gopher 服务器的主菜单选择第 2 项，就得到下面的子菜单：

```
Gopher Services

     1. Hosts and people/
-->  2. Other gopher servers/
     3. Useful college services/

Press ? for Help, q to Quit, u to go up a menu       Page: 1/1
```

从服务器上的菜单开始，沿着到其他服务器菜单的连接前进，直到找到你感兴趣的专题为止。在上面的菜单中，菜单选项后面的斜杠表示这是菜单选项。如果选择它，将使你进入子菜单。其他项是文件选项，选择它将检索这一文件。

即使有了在 gopher 空间（即 gopher 数据库网络）上航行的菜单系统，仍然存在到那里去找特定的专题的问题。正像 ftp 用 archie 来找特定的文件一样，gopher 也有它自己的搜索程序 veronica。使用 veronica 时，送入一组关键字作为查询条件，这一程序搜索大量的文件数据库和菜单标题，找出相应的项目。然后可以根据需要作进一步的检索。

由于 www 的强大功能和灵活性，它不仅提供 gopher 的所有功能，而且还提供大量其他功能。它正在快速地超越 gopher。

12.7 world wide web

world wide web (www) 与 gopher 一样，是由数以千计的服务器组成的网络。网络存有大量称为 web 页 (web pages) 的文档，几乎覆盖任何能想到的专题。

web 页包含用于显示的正文和图形。此外，web 页还包含超文本连接。它允许从 web 页中访问在网上任何位置的其他 web 页。

为了访问 www，你需要 web 浏览程序(web browser)。Linux 发布一般不提供 web 浏览程序。如果你要使用 www 服务，需要另外下载一个浏览程序。当前最流行的 web 浏览程序是 netscape。它的不同版本可以从几个主要的 Linux 网点或它的镜像网点中找到。

装了 netscape 以后，要启动 X-windows 才能运行它。netscape 运行时将显示它的正页(front page)。正页中有一框标为 Location:，框内包含当前浏览的 web 页的网址。web 页用超文本标记语言 (html) 写成。它用简单的方法标记文档正文的各部分：如文档的标题，普通的正文，到

其他 web 页的连接，或到当前 web 页的其他位置的连接等。

在 web 上的每一文档都有一个连接和它结合在一起。这个连接称为它的统一资源定位符（URL）。文档的 URL 具有如下的格式：

```
http://www.server.hostname/pathname/on/server
```

URL 包含 3 个部分：

1. 到第一个冒号为止的部分（这里是 http:），这部分指定用于传输文件的通信协议；
2. 从双斜杠（//）到第一个单斜杠（/）（这里用的是 www.server.hostname），它表示请求提供服务的主机名；
3. 从第一个单斜杠到行的结束（这里是/pathname/on/server），它表示在主机上到所需文档的路径名。

web 和它的浏览程序的巨大吸引力在于：它可以使用许多不同的协议，除 http 外，还可以应用 email，news，ftp，gopher 和 wais 等。

当你阅读 html 文档时，将会发现正文中一些用颜色或下划线加以强调的字。这些被突出的字带有超文本连接。用鼠标选择这些字，将导致浏览程序跟踪连接并显示连接所指的文档的内容。

好象 ftp 和 gopher 一样，www 也有自己的搜索工具。用来找出与特定专题有关的文档的统一资源定位符（URL）。这些搜索工具包括：

```
yahoo       http://www.yahoo.com/
lycos       http://www.lycos.com/
excite      http://www.excite com/
altavista   http://altavista.digital.com/
webCrawler  http://webcrawler.com/
inktomi     http://inktomi.berkeley.edu/
```

这些系统的搜索方法非常灵活。一般做法是：它们从你那里取得一组关键字，搜索它们的数据库，找出包含尽可能多的关键字的文档，然后将检索到的文档按照实用的准则进行排序，每次给你列出其中的一部分。列出的标题对文档本身形成超文本链接。用鼠标选择任何你感兴趣的标题将检索你选择的文档的内容。

除了运行 www 客户程序外，如果你的机器的全部时间都和 Internet 连接，也可以在 Linux 系统上运行自己的 web 服务程序。大多数 Linux 发布不提供 web 服务程序。只要对 cern-httpd 或 NCSA-httpd 进行一次 archie 搜索，能很快找到这两个 web 服务程序的副本。然后再用 ftp 下载和进行安装。完整的安装和运行说明也包含在下载的服务器软件的 tar 文件中。

第十三章　运行 DOS

尽管投身于 Linux 开发工作的人们作出了巨大的努力，仍然会有人认为没有 DOS 不行。这是可以理解的。因为还有一些 DOS 的应用软件包，自由软件的队伍尚未提供相应的产品。

在 Linux 环境中运行 DOS 软件可能会有许多问题。其中最主要的问题是由 CPU 运行模式不同引起的。在 386 和 386 以上的处理器上运行 DOS 时，CPU 用的是与 8086 向下兼容的实地址模式（real-mode）。当在同一处理器上运行 Linux 时，它将 CPU 切换为完全不同的模式，即所谓保护模式（Protected mode）。在两种不同模式下，处理器的许多内部寄存器有不同的含义。

13.1　DOSEMU

在 Linux 系统上运行的 DOSEMU 软件包使你能在 Linux 环境中运行许多 DOS 软件。从名称容易使人想到：DOSEMU 是 DOS 的模拟程序，但事实并非如此。DOSEMU 只模拟 DOS 运行的环境。在这一环境上才真正运行实际的 DOS 软件。由于各种版本的 DOS 软件都是专有产品，所以 DOSEMU 软件包中不包含 DOS 软件。你应该去购买或使用你已有 DOS 的软件。

DOSEMU 利用 80386 处理器的虚 8086 模式提供模拟的硬件环境，使许多实地址模式的软件可以在上面运行。而处理器本身则继续在保护模式下运行。这一方案允许在运行 Linux 系统的同时运行实地址模式软件。

DOSEMU 还提供 PC BIOS 的功能，包括它的所有系统调用和入口点。这表示 DOSEMU 库利用 Linux 功能模拟了 PC 的只读存储器中的 BIOS 功能：

```
int10  显示驱动程序；
int13  磁盘驱动程序；
int14  串行口驱动程序；
int15  输入/输出子系统扩充；
int16  键盘驱动程序；
int17  并行打印机驱动程序；
int1a  CMOS 实时钟
```

只要模拟足够逼真，在模拟环境中运行的软件（譬如说 DOS）将和在实际环境中运行一样。在 DOS 上运行的其他应用软件也可以在 DOS 和模拟的 BIOS 系统调用的基础上运行。图 13.1 表示：DOSEMU 模拟的，符合 DOS 及其应用程序要求的内存映像。

13.2　安装

注意：DOSEMU 处于不断进行修改和升级的过程中。它在不断增加新功能，排除隐错和提高它的可靠性。同时，它也需要跟上 Linux 本身的发展和改进的步伐。如果你要安装

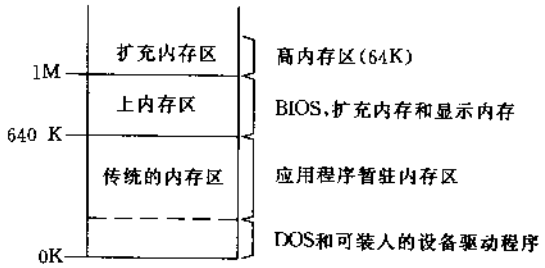

图 13.1 DOSEMU需要模拟 PC AT的内存映像

DOSEMU,最好的做法是从 FTP 档案中取得它的最新版本。

有两种方法使用这一软件包。一种方法是用现存的已经建好的软件包。另一方法是从源程序开始,自己进行编译、安装和配置。一些 Linux 发布将 DOSEMU 作为它的标准的和可选的软件包。在这种情况下,它是预先建好的软件包。安装比较方便。但是你不能得到它的最新的版本。

假定你手头有 DOSEMU 的源程序,建立和安装过程大致可按下面的步骤进行:

- 虽然 DOSEMU 是用户级的程序(它本身不在内核中运行)。为了运行它,仍然要建立新内核。这是因为它使用了 System V 的 IPC 软件包的共享内存和信号机进程间通信(semaphore interprocess communication)机制。必须建立和配置包含 IPC 软件包的新内核(如果你的内核已配置这一软件包就不用再做这一步)。还应保证内核版本足够高,以适合于安装 DOSEMU 版本。
- 应该用 root 帐号登录,否则就不能进行安装工作。现在将档案中的软件解压缩,并将它放在适当的位置。大约需要 2Mb 的磁盘空间用来存储源程序和二进制文件。
- 建立软件,产生文档和完成安装。将目录改变到解压缩文件所在的目录,并送入命令:

```
#  make  doeverything
```

相信我,它做好了。如果安装了 TeX 正文处理软件包,make 命令还能产生相应的手册。如果没有安装 TeX 软件包,可以用下面的命令来代替:

```
# make  most
```

应该将形成最初的 DOS C:驱动器的有关文件复制到相应的位置,以此来完成安装工作:

```
#  cp  ./hdimage.dist  /var/lib/dosemu/hdimage
```

- 建立配置文件,对软件包进行配置:

```
/etc/dosemu.conf
```

有两种方法来完成配置工作,一种方法是运行配置 script,并回答一系列有关机器配置的问题。另一种方法像下面那样复制实例配置文件:

```
#  cp  ./examples/config.dist  /etc/dosemu.conf
```

然后，按照文件中的说明并结合具体配置对文件的内容进行修改。还应该建立下面的文件：

```
/etc/dosemu.users
```

将有权使用 dosemu 软件包的用户的登录名清单写到文件中去。

现在可以装入 DOS 软件了。要有一张可引导的 DOS 软盘，还应将下面两个文件复制到软盘上：

```
FDISK.EXE
SYS.COM
```

将软盘插入 A:驱动器，然后送入命令：

```
#  dos  -A
```

这将引导软盘上的 DOS，并看到熟悉的 A>提示符。输入 dir c:，检查能否列出 hdimage 文件的内容。DOS 模拟程序用这一 Linux 文件模拟 C:驱动器。如果看到模拟的 C:驱动器工作正常，就可以用下面的命令将 DOS 的副本装到 DOSEMU 的 C:驱动器中，并使它成为可引导的：

```
A>  fdisk  /mbr
A>  sys  c:
A>  c:\exitemu
```

最后一条命令将结束 DOSEMU 的运行，并回到 Linux shell 提示符，现在不用再在模拟程序下用软盘引导 DOS。可以直接在 shell 提示符下输入：

```
# dos
```

安装了 DOSEMU 并使它开始工作后，应该花些时间看看文档和手册。然后对配置文件作进一步修改，将 DOS 模拟环境设置得更完善。

第十四章　安　　全

UNIX 系统过去在系统安全方面声誉不佳。原因之一是它的源程序可供研究和实验，因而不可避免地导致不同类型的安全脆弱点的暴露。我们认为这不是问题，而正是它的优势。因为这使得脆弱点能够及时发现并加以克服。其他操作系统也存在类似的问题。只是由于其他操作系统没有这样的机会进行如此深入的探索，尚未很好地排除隐患而已。

在现代 UNIX 或 Linux 系统中遇到的安全问题大多由于系统管理不当引起的。一些可以使用的安全措施没有用上，或者没有正确设置，或者有意牺牲系统的安全来换取用户使用的方便。

14.1　文件的权限

在本节中，将讨论文件的权限设置。为了保证系统的安全，应该如何设置 Linux 目录层次结构中的不同的文件和目录的权限。

总的说来，root 帐号必须拥有所有标准的 Linux 命令（也就是在 /bin 和/usr/bin 目录下的所有可执行文件）。如果它们是可执行的 2 进制文件，对这些文件的权限设置不能超越rwx--x--x；如果它们是可执行的 shell script 不能超越 rwxr-xr-x。这些权限位设置禁止 root 以外的用户去写文件。所以，普通用户不能修改任何标准命令。好的做法是：除了完成功能所必需的权限外，尽少设置文件的权限位。

在通常情况下，当执行一个程序时，相应的进程具有执行这一程序的用户的权限。有时，这并没赋予进程正确完成任务所必要的权限。在这种情况下，可以设置程序的 SUID 或 SGID 位，使得用户分别以程序拥有者或用户组的权限，而不是它自己的权限去运行程序。这是使一些程序（如 passwd 程序）能够正确运行的唯一的办法。如果这一功能落入非法用户之手，将导致滥用这一功能的一切可能。例如，作为 root 帐户登录，短时间留下终端无人看管。离开时，黑客(hacker)只要化很短的时间送入下列命令：

```
#  cp  /bin/bash  /home/hacker/.innocuous.file
#  chmod  4755  /home/hacker/.innocuous.file
#  clear
```

你回来时，什么也没有发现。而黑客实际隐藏了一个设置 SUID 的 root shell。他随时可以使用，只要他自己不做蠢事，不必担心被发现。

有时，系统中的一组用户要能修改一些系统数据文件的内容。在这种情况下，设立特定的用户组，将有关的文件的 GID 设为这一组用户的 GID。然后，将这组文件的权限位设为 rw-rw-r--。这就不需要 root 的权限就能使组的成员读写这一组文件，并给系统中其他用户读文件的权限。用这种方式减少使用 root 帐号的机会也出于重要的安全考虑。

作为附加安全措施，还可以用 umask 命令屏蔽文件建立时设置的权限位。

与正确设置数据文件及可执行文件的权限位一样，正确设置目录的权限位也非常重要。这一点常常被忽视。特别是，除了 /tmp 和 /usr/tmp 目录外，普通用户没有写系统目录的权限。注意，目录的写权限表示：对目录下的文件增加新的连接，也能删除已有的连接。

目录的读权限表示：读目录下的文件的名称，而不是读文件本身的内容。目录的执行权限表示：目录可以作为路径名来访问这一目录下的文件。将两者结合在一起，目录的读和执行权限可以这样设置：使用户不能列出目录的的内容，但能访问目录下已知的文件。相应的目录权限位应设为：

```
rwx--x--x
```

这样的设置使目录拥有者具有全面访问目录和它的内容的权限，其他用户只能访问此目录中已知的文件。

14.2 访问设备

因为设备特殊文件直接访问机器的硬件，需要慎重对待设备特殊文件的权限。如果你的用户中有知识渊博的人，对这些文件的权限设置不当，可能导致严重的安全问题。

要特别注意属于内存及磁盘驱动器的设备特殊文件。这些文件的名称是：

```
/dev/hda    /dev/hda1    /dev/hda2    etc.
/dev/hdb    /dev/hdb1    /dev/hdb2    etc.
/dev/sda    /dev/sda1     etc.
/dev/sdb    etc.
/dev/mem    /dev/kmem
```

所有这些文件必须为 root 所有。它们的权限位设置应尽可能限制在最小范围内。

在任何情况下，都不允许 root 以外的任何其他用户拥有对这些文件的读、写权限。假如一个用户掌握了磁盘或磁盘分区的读、写权，他就能在磁盘或分区上绕过所有标准的文件和目录的权限设置。因为他可以不管系统在上面建立的目录结构，就像存取数据文件一样，直接存取驱动器上的字节。无可否认，用户必须将数据区分开来，用人工将它们分为文件和目录。但这并不像听上去那么困难。特别是，找出存在磁盘上的文件系统是能够做到的。

如果用户拥有内存特殊文件的读、写权限，也会产生类似的问题，因为这些文件直接映射到机器的内存区，因而用户能够看到和修改运行系统的内存区。

在 Linux 系统中，访问机器内存以及与每个进程结合的内存的另一种办法是通过 /proc 目录的内容。虽然，可以用 ls /proc 命令列出它的内容。事实上，/proc 和它的内容实际并不出现在你的硬盘上。这一目录和它的内容是由内核动态地放进来的。

很明显，如果运行中进程的内存区可以由具有足够权限的用户进行存取，安全就成问题，特别是，如果有人闯入了你的 root 帐号。

大多数 Linux 系统通过网络或机器的控制台终端和虚终端进行访问。然而，有些系统通过串行口连接了其他终端。这可能成为问题的另一个来源。传统的做法允许任何人写终端特

殊文件,因而允许用户之间用 write 命令传送简单的信息。对一些串行终端讲,这可能就是问题的来源。

对许多终端讲,可以对送入的简单的转义字符和命令字符组进行编程,使它们完成简单的操作。其中的一种操作是将一些字符从终端返回主机,就像直接从键盘送入一样。现在你可以想象:如果一个黑客从串行终端上用 write 命令将一串字符送到你的终端,这些又使终端将编程的命令送回主机,这些命令就像你自己从键盘上送入的一样由主机执行。假如,此时你使用 root 帐号登录,你就能想象可能出现的后果。

唯一防止受到这类攻击的办法是:从你的终端上去掉其他用户的写权限。这可以用下面的命令实现:

```
$ mesg n
```

遗憾的是:防止出现安全问题的可能性,同样也使其他用户不能用 write 命令向你传送信息,但是你无法同时兼顾两个方面。

14.3 口令

口令是保证系统安全的最有力措施。当一个用户试图登录到你的系统时,首先就要求他提供口令。对送入的口令进行加密,然后与原先保存在 /etc/passwd 文件中的加密口令进行比较。如果两者相同,就允许这一用户登录到系统中去。由于口令文件中的信息有多种用途,也由于安全地对口令进行了加密,普通用户都可以读这一文件的内容。

这些都不成问题,但要考虑人为的因素。多数用户希望口令便于记忆,因而愿意从他们认为有意义的事情中找口令。如他们双亲的名字,他们的生日等等。这样选择的口令容易招来麻烦。任何对你和你的朋友有一定了解的人能猜出这一类口令。当你选择口令时,应该包含大、小写字母,数字和标点符号的组合,因而不容易被别人猜出。当安全问题特别重要时,应经常改变重要的系统口令,而且要严格限止掌握这些口令的人数。

除了用户口令外,在 Linux 系统中还可以在 /etc/group 文件中设置用户组口令。允许任何掌握组口令的用户用 newgrp 命令进入这一用户组。但是从安全的角度考虑,不设置组口令更为安全。只允许用改变 /etc/group 文件中组内用户名称清单的办法,使用户从一个组转到另一个组。

14.4 root 帐号

使用 root 帐号的第一条规则是:如果你有 root 帐号,除非绝对必要时,就不要轻易使用它。第二条规则是:在系统中尽量少设 root 帐号。只要有可能,尽量采用由用户组来完成系统管理任务的代替方案,而尽少分配 root 帐号。这一点可以通过分配用户组,并指定合适的组访问权限来实现。

下面是作为 root 帐号登录时,需要小心谨慎的例子。当你以普通用户身份登录时,shell 的环境变量 PATH 中包含用冒号分开的目录名清单。shell 按清单中给定的顺序搜索这些目录,找出运行的命令的可执行文件。PATH 变量的典型内容如下:

```
$ echo $ PATH
. :/usr/local/bin:/bin:/usr/bin:/usr/X11/bin
```

本例中,在搜索路径中指定了5个目录。从当前目录(.)开始搜索。先搜索当前目录是很方便的,因为有可能在无意之中建立了一个与系统命令同名的程序文件。在这种情况下,当从当前目录下运行新命令时,它将比同名的系统命令先被搜索到而执行。如果不搜索当前目录,将执行同名的系统命令而引起混淆。

我曾遇到这种情况。我写过一个简单程序来检验我的新想法,习惯地将它称为test。当然,这也是标准的系统命令。不巧的是:当运行不带参数的test命令时,只回送shell提示符而不给出任何信息,还以为我的test程序有问题。

作为普通用户,将你的PATH变量设为从当前目录开始搜索是可取的。作为root帐号,这样做肯定是错误的。为了取得最大程度的安全,当前目录就不应该在root的PATH变量中出现。为什么不能这样做,请考虑下面的可能性。一个用户请你过去,要求你在他的终端上履行系统管理员的职能。你过去了,在他的终端上用su命令将用户帐号改为root帐号。并按用户要求执行所需的系统命令,然后作为root帐号退出登录。继续进行你原先的工作。你不会想到你的用户是一个黑客,利用机会建立一个篡改的命令存在当前目录下,让你作为root去执行。

如果你的root帐号的RATH变量从当前目录开始,当你为用户完成管理任务时,运行的将是用户篡改的命令而不是你自己的命令。作为root帐户运行用户的命令将给他做任何事情的权限。一个聪明的黑客能用篡改的命令在安全的位置建立root拥有的设置SUID的shell。然后再执行实际的命令,使你什么也没有发现。你需要对此有所警惕。

还有许多其他的安全问题,限于篇幅不在这里一一说明。但是,本章的说明有助于从正确方向去考虑安全问题。

第三篇

系统程序设计

第十五章　文　　件

本书第三篇的主题是 Linux 系统调用。这是应用程序和内核(kernel)提供的各种服务之间的主要接口。

Linux(事实上也是所有 UNIX)的基本原理之一是:系统试图使它对所有各类设备输入和输出,看上去就像对普通的文件的输入和输出一样。这些设备包括磁盘、光盘、终端、打印机等。

这一点在系统调用的使用上反映最为明显。当我们考察系统驱动不同类型的设备时,用于普通文件的系统调用将会反复出现。

15.1　顺序文件

Linux 的普通文件是数据字节的集合。这些数据字节一个接一个顺序地存储在该文件中。所以也称顺序(Sequential)文件。Linux 并不在数据上增加任何类型的内部结构。就 Linux 来说,它并不把数据分解成记录或字段。如果一项特定的应用要求使用某种类型的内部数据结构,则由作为程序员的你在数据上增加相应的结构。这使得 Linux 对文件的接口变得特别简单。

15.1.1　文件描述符

在 Linux 系统中,通过文件描述符(file descriptors,fd)访问文件。每个进程最多能够同时使用 OPEN_MAX 个文件描述符。通过标准头文件(header file) < limits.h > 访问 OPEN_MAX 的值。对 Linux 讲,它的值为 256。

OPEN_MAX 的定义实际上是在 < linux/limits.h > 中。但是为了使程序具有可移植性,不应该在代码中直接包含这个头文件。因为在代码中包含 # include < limits.h > 语句时,它就自动地被包含。

传统的做法是:当启动一个进程时,文件描述符 0、1 和 2 已经被分配并可供使用。文件描述符 0 被用作标准输入设备,文件描述符 1 被用作标准输出设备,文件描述符 2 被用作标准错误输出设备。

每个分配的文件描述符与一个打开的文件描述(open file description)相联系。打开的文件描述正是与该文件有关的信息结构。信息包含偏移值(offset)、文件访问模式(mode)和其它有关标志(flag)。偏移值用来规定下次存取文件的具体位置,文件访问模式规定文件用作输入或输出、或同时用作输入和输出。

文件描述符和打开的文件描述之间不一定是一对一的,它也可以是多对一的。这就是说,有可能属于不同进程的几个文件描述符指向同一个打开的文件描述。存储在打开的文件描述的数据结构中的信息将由所有指向它的文件描述符所共享。

事实上,我们将在后面见到,在文件描述符结构中也存有某些信息。这意味着,虽然两个或多个文件描述符全都指向同一个打开的文件描述。但这些信息可能有不同的值。

15.1.2 open 系统调用

有几种方法可以获得允许访问文件的文件描述符。最常用的方法是使用 open()(打开)系统调用:

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

int open(const char *path, int flags);
int open(const char *path, int flags, mode_t mode);
```

注意,open()系统调用具有两种不同的形式:第一种取两个参数,第二种取三个参数。实际上,将常常使用两个参数的形式。除非试图打开的文件不存在,并且希望创建该文件。在这种情况下,才需要使用三个参数的形式。

参数 path 是指向所要打开的文件的路径名(pathname)的指针。标志参数 flags 规定如何打开该文件,它必须包含下列三个值之一:

O_RDONLY 为只读访问打开文件;
O_WRONLY 为只写访问打开文件;
O_RDWR 为读和写访问打开文件。

此外,还可以包含利用按位逻辑加(bitwise-OR)(|)对下列标志值进行任意组合:

O_CREAT 打开文件,如果文件不存在则建立文件;
O_EXCL 如果已经置 O_CREAT 并且文件存在,则强制 open()失败;
O_TRUNC 在 open()时,将文件的长度截至 0;
O_APPEND 强制 write()写在文件的结束处。

还可以规定另外一些标志,但是在普通文件中并不使用它们。当后面遇到时,我们再讨论这些标志。所有这些标志值的符号名称可以通过 # include <fcntl.h> 访问。

如果使用 open()创建一个文件,参数 mode 则用来规定对该文件的所有者、文件的用户组及系统中所有其他用户的访问权限位(permission bits)。应该用按位逻辑加对下列符号常量建立所需的组合(这些符号常量通过 <sys/stat.h> 定义):

S_IRUSR 文件所有者的读权限位;
S_IWUSR 文件所有者的写权限位;
S_IXUSR 文件所有者的执行权限位
S_IRGRP 文件用户组的读权限位;
S_IWGRP 文件用户组的写权限位;

S_IXGRP 文件用户组的执行权限位；
S_IROTH 文件其他用户的读权限位；
S_IWOTH 文件其他用户的写权限位；
S_IXOTH 文件其他用户的执行权限位。

三个有用的按位逻辑加组合定义如下：

S_IRWXU 定义为(S_IRUSR | S_IWUSR | S_EXUSR)
S_IRWXG 定义为(S_IRGRP | S_IWGRP | S_IXGRP)
S_IRWXO 定义为(S_IROTH | S_IWOTH | S_IXOTH)

此外，还可以用下列的常量值规定设置 set-uid 位和 set-gid 位。

S_ISUID　　置 set-uid 位；
S_ISGID　　置 set-gid 位.

不要忘记，当一个进程代表你创建文件时，其权限位由与该进程相联系的 umask 值所修改。进程利用下列公式决定被创建的文件的实际权限位：

```
mode & (~umask)
```

这表示，如果程序所创建的文件上所得到的权限位和在 open()调用中所要求的 mode 参数不一致，不应该感到惊奇。如果安全问题变成重要，我们将利用 umask 的值做某些事(见 21.1 节)。

下面是利用各种参数值调用 open()的例子：

```
open("xfile", O_RDONLY);
open("yfile", O_RDWR | O_TRUNC);
open("zfile", O_WRONLY | O_CREAT | O_EXCL, S_IRWXU | S_IXGRP);
```

第一个例子打开文件 xfile 用于读访问。如果 xfile 不存在，则 open()返回一个错误。第二个例子打开文件 yfile 用于读和写访问。该文件必须已经存在，并且将它的长度截为 0，因而，将丢失文件中以前的内容。第三个例子打开文件 zfile 用于写访问。open()调用期待创建该文件，如果文件已经存在，则返回一个错误。因为 zfile 是新创建的文件，需要规定其权限位的组合，这里用的是 rwx--x---。

有时阅读一个 open()调用可能稍难一些。特别是，如果对许多标志使用了逻辑加操作。在这种情况下，在程序前面用 # define 语句定义适当的符号常量是好的做法。为了给出文件权限位组合 rwxr-xr-x，可以定义常量 MODE755 如下：

```
#define MODE755 (S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH)
```

当 open()检测到错误时,由返回 -1 值指示。因为存在许多不同的产生错误的条件,由内核赋予外部整型变量 errno 相应的值,以此来指示发生了什么错误。open()调用的主要错误有:

ENOENT	没有规定 O_CREAT 就试图打开一个不存在的文件。
EEXIST	利用 O_CREAT|O_EXCL 强制创建一个已经存在的文件。
EACCES	没有该文件所要求的访问权限,或者没有文件路径名中包含的目录的搜索权限。

这些符号通过 < errno.h > 访问。

如果没有出现错误,从 open()调用的返回值是一个文件描述符。在随后对该文件的所有操作中将使用这个文件描述符。文件描述符本身是一个小的非负整数,它实际是一个指向打开文件描述的指针数组的索引。当一个文件描述符被分配时,将搜索这个数组以找到第一个空闲的元素。其结果是当打开一个文件时,Linux 总是分配最低编号的空闲文件描述符。以后你将看到,在我们讨论如何实现输入/输出重新定向到文件和管道(pipes)时,这将得到很好的利用。

有时试图打开一个特定的文件之前,可能希望检验进程是否已经访问过该文件。这可以用 access()系统调用来完成。access()系统调用的一般形式是:

```
#include <unistd.h>

int access(char *pathname, int mode);
```

其中,pathname 是你希望检验的文件名,mode 是包含在文件 < unistd.h > 中的下列值之一:

R_OK	检验调用进程是否有过读访问;
W_OK	检验调用进程是否有过写访问;
X_OK	检验调用进程是否有过执行访问;
F_OK	检验规定的文件是否存在。

15.1.3 creat 系统调用

为了维持与早期的 UNIX 系统的向后兼容性,Linux 也提供可选的创建文件的系统调用,它称为 creat()。

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

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

在 UNIX 的早期版本中,open()系统调用仅仅存在两个参数的形式。如果文件不存在,它就不能打开这些文件。文件的创建则由单独的系统调用 creat()完成。在 Linux 及所有 UNIX 的近代版本中,creat()系统调用是多余的。事实上,creat()调用

```
fd = creat(file, mode);
```

完全等价于近代的 open()调用

```
fd = open(file, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

15.1.4 read 系统调用

一旦你有了与一个打开文件描述相连的文件描述符,只要该文件是用 O_RDONLY 或 O_RDWR 标志打开的,你可以用 read()系统调用从该文件中读取字节:

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

int read(int fd, void *buf, size_t nbytes);
```

其中 fd 是想要读的文件的文件描述符,buf 是指向内存块的指针。read()将从该文件中取来的字节放到这个内存块中。nbytes 是从该文件复制到 buf 中的字节个数的计数。

read()系统调用的函数原型在 < unistd.h > 中给出,数据类型 size_t 通过 < sys/types.h > 访问,目前定义为无符号整型(unsigned int)。在 PC 的 Linux 中,它取 32 位值。

read()操作从当前文件位置开始,这个位置由包含在相应的打开文件描述中的偏移值(offset)给出,并且继续操作直到所需的字节数已经从该文件读出为止,或者直到遇到文件结束符为止。你的责任是保证为 read()规定的数据缓冲区充分大,能够存放所要求的数据字节数。因为,内核不进行检查,它将只顾复制其数据。

在 read()操作结束时,文件偏移值将被加上刚刚复制过的数据量,到达新的位置为下一次 read()操作做好准备。

如果出现错误,read()返回 -1 值。如果文件偏移值是在文件结束(end-of-file)处,read()返回 0 值,否则它返回从该文件复制到规定的缓冲区中的字节数。通常这个字节数和请求的字节数相同。除非请求的字节数超过剩余的字节数。在这种情况下,将返回一个小于请求的字节数的数字。如果在此期间文件的容量不变,下一次 read()将返回 0 值(end-of-file)。

实际的磁盘读操作通常全部是在磁盘数据块中进行的,然后将它们存放在内存缓冲区中。为了改善系统性能,内核存在这样的机制,如果顺序地读一个文件(即读相邻的磁盘块),它可以预料进一步的数据请求,并且利用提前读技术使数据预先准备就绪。

15.1.5 write 系统调用

用 write()系统调用将数据写到一个文件中。从表面看,很容易将 write()系统调用想象为 read()系统调用相反的操作。它具有形式:

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

```
int  write(int fd, void *buf, size t nbytes);
```

它从 buf 所指的内存块将 nbytes 个字节写到文件描述符 fd 所指的文件中。

从当前文件偏移值位置开始将数据写到文件中,自动地将偏移值加上实际写入的字节数,为下一次 write()调用作好准备。如果文件用 O_APPEND 标志打开的,则在 write()发生之前将文件偏移值自动地增加到文件结束位置。借助 O_APPEND 标志来防止覆盖文件中已有的数据,在创建日志(log)文件和审计跟踪时很有用。

正如 read()一样,内核使用内存缓冲区保存进程和磁盘之间的数据。但是这里有细微的差别不容易被发现。在 read()调用的情况下,如果在内存中没有可用数据,则在 read()调用可以返回之前,进程不得不等待实际磁盘读的发生。如果在读数据时出现任何磁盘错误,则从 read()的返回值和 errno 的值可以反映此错误的细节。

在 write()调用情况下,数据只不过保存在内存缓存区中,然后 write()调用立即返回。在这过程中内核作出允诺,数据最终将放到磁盘上去。任何随后的读这些你刚写入的数据,将取自内存缓存区而不是取自磁盘。这意味着,除了读写内存中的数据要比实际磁盘读写快几个数量级以外,用户程序并不知道实际发生的情况。

问题在何处? 不管内核如何作出允诺,问题在于实际写磁盘可能永不发生。如果在从内存缓存区将数据写到磁盘时出现磁盘错误,则那些数据将被丢失。并且更坏的是,没有任何机制能够让写数据的进程知道这个问题。在实际读磁盘期间,错误由 read()的返回值报告给读进程。对于写数据可不行,在数据被实际转移到磁盘以前 write()调用可能早就返回。事实上,在出现写磁盘错误以前,写进程甚至于可能已经结束。磁盘硬件失效不是问题的唯一来源,任何阻碍写磁盘的情况的发生都可能引起麻烦,包括诸如机器电源失效,按复位(reset)按钮,没有使用卸下(umount)命令而移去一个已装配的(mounted)软盘,偶然关机等。

一般地说,当代硬件是如此可靠,以至于性能大为改善补偿了丢失数据的轻微的风险。但是,在最轻微的风险都不能接受的情况下,需要另外作出安排。

如果你正在使用带有 Linux 第二扩展文件系统的新 Linux 内核,则在 open()系统调用中还可以规定另外一个标志 O_SYNC:

```
fd = open(file, O_WRONLY | O_SYNC);
```

这个标志的效果是使对特定的文件描述符(这里是 fd)的 write()系统调用和相应的实际写磁盘同步。事实上,直到实际写磁盘已经完成,write()系统调用才返回。

注意,如果使用这个标志,则代码可能不能移植到其它操作系统。事实上,它甚至不能移植到 Linux 中的其它文件系统类型的文件上。你必须知道这点,因为,如果偶然地在其它文件系统类型的文件上使用了 O_SYNC,将得不到任何错误指示,系统将对执行所期待的同步失效保持沉默。

除了数据缓存问题之外,write()调用类似于 read()。所以,保证写的数据字节数和数据缓冲区容量一致,像这样的事情还是你的责任。

write()在错误时返回 -1 值,并相应地置 errno 值。或者返回实际写到文件的字节个数。在普通文件的情况下(不是某些其它的文件类型),如果 write()返回的值和期望写的字节数的

值不相同,则这表示发生了问题(例如,超出了最大的文件容量的限制)。无论如何,应该把它当作错误对待。

15.1.6 关闭(close)系统调用

为了重新利用文件描述符,用 close()系统调用释放打开的文件描述符:

```
#include <unistd.h>

int  close(int  fd);
```

每次分配一个文件描述符时,在打开文件描述中的引用计数(reference count)值被加 1,使得每个打开文件描述知道有多少个文件描述符与之相连。每当在一个文件描述符上调用 close()时,所属的打开文件描述的引用计数值被减 1。最终,一次对 close()的调用将使引用计数值为零。在这种情况下,close()调用不仅释放该文件描述符而且也释放该打开文件的描述。

close()调用在成功时返回 0 值,在错误时返回 -1 值,并且由 errno 存放错误编号。事实上,close()调用仅能返回一个错误,它是:

```
EBADF   参数不是一个有效的打开的文件描述符
```

事实上,用一个不和打开的文件描述相连的文件描述符作为参数调用 close()没有任何危险,因而,极少为了错误而检查从 close()返回的值。

虽然当一个进程结束时,任何打开的文件将被自动地关闭,明显地关闭任何打开的文件是良好的程序设计习惯。

15.1.7 用户缓冲(Buffering)

系统调用接口是普通应用程序对操作系统服务的最低层的访问。不应该由此得出这样的想法,即它们的使用必定是做某些事情的最有效的方法,举数据缓冲作为例子来说明这一点。

下面是一个非常简单的函数,它取两个指向文件名的指针作参数,并且将第一个参数所指的文件的内容复制到第二个参数所指的文件中:

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

#define  NEWFILE  (O_WRONLY | O_CREAT | O_TRUNC)
#define  MODE600  (S_IRUSR | S_IWUSR)
#define  SIZE  1

void  filecopy(char * infile, char * outfile)
{
   char buf[SIZE];
   int infd, outfd, count;
```

```
    if ((infd = open(infile, O_RDONLY)) == -1)
        fatal("opening infile");

    if ((outfd = open(outfile, NEWFILE, MODE600)) == -1)
        fatal("opening outfile");

    while ((count = read(infd, buf, sizeof(buf))) > 0)
        if (write(outfd, buf, count) != count)
          fatal("writing data");

    if (count == -1)
      fatal("reading data");

    close(infd);
    close(outfd);
}
```

这里没有列出用户函数 fatal(),它所需做的只是输出它所包含的字符串作为错误信息,然后结束进程的运行。

你已经知道系统使用大量的内存作为读写磁盘文件的高速缓冲区。而且从内存读写字节是非常迅速的操作。但是,在 filecopy()函数情况下,当它工作时,它执行所要求的复制,但它不是很有效。运行一个系统调用有一定的时间开销,这个函数每复制一个字节要进行两次系统调用(一次 read()和一次 write())。

利用所列的 filecopy()函数,我进行了少量的简单计时测试(利用 time 命令)。在我的系统上发现,我可以用每秒约 6.4K 字节速率复制文件。

现在,为了减少包含在复制操作中系统调用次数,你只要增加常量 SIZE 的值,这是相当简单的事。其效果是以较大容量的块和以较少次数来复制文件。选择一个自然的 SIZE 的值应该与磁盘数据块的容量有关。

在将 SIZE 值改变为 1024 后(不作其它改变)重复这个试验,文件的复制速率增加到了一个使人震惊的(无论如何,也给我以深刻的印象)每秒 1.3M 字节。速率的改善超过了 200 倍。

虽然速率得到了改善,以这样大的块读数据有时不很方便。在此情况下,可以在用户内存空间做自己的小的数据缓冲区,可以达到既快又方便。

下列的程序表明,对 filecopy()函数作小的改变,使得它调用函数 my_read()和 my_write(),除了它们使用内部的 1024 字节缓冲区以减少使用系统调用的次数之外,这两个函数和系统调用 read()和 write()完全相同。为了在关闭文件之前,刷新 write_buf[ ]的内容,利用 write_close()而不用 close()系统调用来关闭输出文件是必要的。

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

#define  NEWFILE  (O_WRONLY | O_CREAT | O_TRUNC)
```

```
#define  MODE600  (S_IRUSR | S_IWUSR)
#define  SIZE  1
#define  BUF_SIZE 1024

void filecopy(char * infile, char * outfile)
{
   char  buf[SIZE1;
   int infd, outfd, count;

   if ((infd = open(infile, O_RDONLY)) == -1)
       fatal("opening infile");

   if ((outfd = open(outfile, NEWFILE, MODE600)) == -1)
       fatal("opening outfile");

   while ((count = my_read(infd, buf, sizeof(buf))) > 0)
       if (my_write(outfd, buf, count)! = count)
          fatal("uriting data");

   if (count == -1)
     fatal("reading data");

   close(infd);
   write_close(outfd);
}

int my_read(int fd, char * buf, size_t count)
{
   static char read_buf [BUF_SIZE] ;
   static int read_offset = BUF_SIZE;
   static int read_max = BUF_SIZE;
   int i;

   for  (i = 0;  i < count; ++i)
   {
     if  (read_offset == read_max)
     {
       read_offset = 0;
       read_max = read(fd, read_buf, sizeof(read_buf));

       if (read_max == -1)
         return -1;

       if (read_max == 0)
```

```
            return i;
        }

        *buf++ = read_buf[read_offset++];
    }

    return i;
}


static char write_buf[BUF_SIZE];
static int write_offset = 0;

int my_write(int fd, char *buf, size_t count)
{
    int i, n;

    for (i = 0; i<count; ++i)
    {
        write_buf[write_offset++] = *buf++;

        if (write_offset == BUF_SIZE)
        {
            write_offset = 0;
            n = write(fd, write_buf, sizeof(write_buf));

            if (n! = BUF_SIZE)
                return -1;
        }
    }
    return i;
}

int write_close(int fd)
{
    if(write_offset>0)
        write(fd, write_buf, write_offset);

    write_offset = 0;
    return close(fd);
}
```

这段代码仅仅让你体会用户缓冲技术的特色,因为通常不必写自己的代码,在标准 C 库中有一个完善的使用用户缓冲技术的文件 I/O 软件包(fopen(), fclose()等)。

注意,在写文件时使用的用户缓冲技术,将遇到我们在 write()系统调用本身曾经遇到过的,类似的关于报告错误信息的问题。

使用用户缓冲技术和用一个字节的读和写测试 filecopy()函数的数据传输率,给出每秒 330K 字节的速率。如你所期望的,由于调用额外的函数的开销,不如直接用大的缓冲区调用 read()和 write()那样快,但是它仍然表明,它比直接一个字节的读和写速率改善超出 50 倍以上。

15.2 随机文件

到此为止的所有文件访问都默认为是顺序访问。这是因为所有的读和写都从当前文件的偏移位置开始,然后文件偏移值被自动地增加到刚好超出读或写结束时的位置,使它为下一次访问作好准备。

有了这一说明,在 Linux 系统中,随机(Random)访问就很简单。你所需做的只是将当前文件偏移值改变到有关的位置,它将自动地强迫下一次 read()或 write()发生在这一位置(当然,除非文件被 O_APPEND 打开,在这种情况下,任何 write()调用仍将发生在文件结束处)。

执行这项任务所要求的系统调用是 lseek():

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

off_t lseek(int fd, off_t offset, int base);
```

参数 fd 是文件描述符,它所指的打开文件描述将被该调用修改。

当对文件偏移值规定一个新位置时,只需给一个数作为新值。这相当于提供一个相对于文件的开始位置的偏移值。另外两个可能性是需要给出相对于当前的文件位置的偏移值,或者希望给出相对于文件结束的偏移值。这三个可能性以及用于选择它们的符号常量表示在图 15.1 中。

图 15.1 lseek()系统调用的偏移值的三种基本位置

参数 offset 是一个相对值,它被加到所选的基地址上给出新的文件偏移值。参数 base 可以取下列三个值中的一个(这些值定义在 <unistd.h> 中):

SEEK_SET	从文件的开始处计算偏移;
SEEK_CUR	从当前的文件偏移值计算偏移;
SEEK_END	从文件的结束处计算偏移。

参数 offset 和 lseek()自身的返回值的数据类型都是 off_t。实际上,这里并没有给出值的类型的任何有效指示,但在 <sys/types.h> 中 off_t 的类型定义为:

```
typedef long off_t;
```

下面一些简单的例子将帮助说明 lseek()的使用：

```
lseek(fd, (off_t) 0, SEEK_SET);
lseek(fd, (off_t) -50, SEEK_CUR);
lseek(fd, (off_t) 5000, SEEK_END);
```

第一件要注意的事是，为了从长远考虑，把参数 offset 的数据类型安排为 off_t 是必需的。这允许 offset 参数现在或将来在 Linux 的不同端口上有不同的基本数据类型。

在第一个例子中，lseek()调用把文件描述符 fd 所指的打开的文件描述中的文件偏移值设置成参数 offset 的值(0)和 SEEK_SET 所指定的文件开始的偏移值(也为 0)之和。所以这个调用设置文件偏移值为 0，即文件的开始。

在第二个例子中，把文件的偏移值从它的当前位置(SEEK_CUR)向后移动 50 个字节。不要忘记，在写文件时将自动地把文件的偏移值向前移以准备好下一次读或写操作。因此，如果你想一次一个字节地向后工作，当你写新的字节值时，需要的 lseek()偏移值将是 -2，而不是你最初期望的 -1。

在第三个例子中，把文件偏移值从文件结束的位置(SEEK_ END)向前移动 5000 个字节。把文件偏移值像这样移过文件结束的位置，初看起来似乎非常奇怪。但是，这正是在这里所发生的。如果现在你在这个新位置写数据，则文件容量将变成新的文件结束处的文件偏移值。在旧的文件结束和新的文件结束之间的间隙中出现了什么情况？我很高兴你问这问题。事实上，什么都未发生，甚至在磁盘上也没有分配任何数据块(即使最终将数据写到磁盘时也没有)。结果是一个带有间隙的文件，它的逻辑容量(由命令 ls -l 所报告的)是文件的结束处的文件偏移值，而实际容量(它在磁盘上所占据的数据块的数目)可能非常少。

容易使你自己相信这是可能的。你只要写一个程序创建一个新文件，然后使用 lseek()在文件中将文件偏移值移动一个长的距离。为了真实地使你信服，试用一个比存放该文件的文件系统的容量更大的值。然后写一个字节并且关闭这个文件。现在使用 ls -l 命令来看你刚刚创建的文件的容量，它比文件系统容量还要大。

如果以后某个时候你再次打开该文件并且寻找和写到间隙中某一个位置，则数据块将按需要分配给该文件用来存放新的数据。这将增加文件的实际容量而不是它的逻辑容量。在任何时候如果企图从该文件的间隙中读数据，则 Linux 将不为读该位置而分配数据块，而只返回零字节值。

当 lseek()移动文件偏移值时，唯一的限制是它不能移到文件开始之前(即最终的文件偏移值不能是负的)。

lseek()在出错时，返回 -1 值，并将其错误值放在 errno 中；或者返回文件偏移位置的新值(从文件开始位置算起的字节数)。例如：

```
pos = lseek(fd, (off_t) 0, SEEK_CUR);
```

将 pos 置为当前文件偏移值，并且

```
pos = lseek(fd, (off_t) 0, SEEK_END);
```

将 pos 置为文件的长度(有另外的方法来完成这一点，而且不必改变当前文件偏移值)。

15.3 终端输入/输出

在 Linux 系统中，所有的输入和输出都尽可能地做成像从普通文件的输入和输出一样。但是，终端输入/输出(Terminal I/O)是比较困难的情况之一。如果把终端处理成像顺序访问文件一样进行输入和输出，则可以使用 open()、close()、read()和 write()系统调用。即使如此，仍然存在某些差别。

通常，一个进程通过从它的双亲(parent)进程继承文件描述符 0、1 和 2，被自动地附属于一个终端。如果是这种情况，则 open()调用已经由进程完成，并且所附属的终端将是该进程的控制终端(稍后我们将看到控制终端的含义)。

如果需要为自己打开一个终端，则需要将一个文件名作为参数传给 open()。基本上有两种可能性：或者可以使用表示你的控制终端的特殊文件名/dev/tty(如果有的话)；或者需要知道一个特殊终端设备的特殊文件名，诸如虚拟的控制终端名 dev/tty1、/dev/tty2 等等。

思考一下你将明白，O_CREAT、O_EXCL、O_TRUNC 和 O_APPEND 标志用到终端文件时没有任何意义。但是，有两个新的标志有用：

O_NOCTTY	停止这个终端作为控制终端；
O_NONBLOCK	使 open()、read()和 write()不被阻塞。

为了与某些其它的 UNIX 系统兼容还提供了 O_NDELAY 标志。在 Linux 下，O_NDELAY 和 O_NONBLOCK 完全一样，虽然，它的语义实际上稍有不同。

稍后，我们将再一次讨论控制终端。但是，阻塞是什么意思？假定在一个终端键盘上进行 read()调用，但用户不敲任何键，将会发生什么情况？

事实上，当你正在从终端读的时候没有按任何键，read()调用将不返回。结果是，进程被阻塞或者挂起，直到 read()请求可以以某种方法得到满足为止。

在 Linux 内核中，每个终端文件都有数据缓冲区。当字符被发送给终端设备文件和从终端设备文件接收字符时，这些数据缓冲区的默认作用是存储字符，直到遇到一个换行字符(\n)为止。在键盘缓冲区的情况下，仅当已收到换行字符时，阻塞的 read()调用才被释放并且这些字符可供读进程使用。不管请求多少个字节，read()调用不可能返回比缓冲区中到第一个换行字符为止的更多的字符。如果请求的字节个数少于这个数，则 read()将只返回所请求的字节个数。在这两种情况下，从 read()返回的值将是实际读出的字符个数。

如果在 open()中置了 O_NONBLOCK 标志，则 read()的阻塞行为被抑制。现在如果当读终端时数据未准备好，则 read()调用返回 -1 值，并将 errno 置为 EAGAIN 值，指示应该重试 read()调用。为了和 UNIX 兼容，也定义了 errno 的错误值 EWOULDBLOCK，它和 EAGAIN 的值完全一样。

在用 write()调用写终端设备的情况下,不会产生阻塞,所写的字符只是被缓冲。直到发出一个换行字符为止,此时缓冲区的内容将被显示。

对终端设备的 close()调用好像对普通文件的 close()调用一样,释放文件描述符以供将来使用。

15.3.1 终端控制函数

如果你要终端设备做的只是使用 open()、close()、read()和 write(),则对终端的访问非常类似于对普通文件的访问。但是,你可能有许多要做的事情,如设置终端接口,就不适合由这些调用来完成。例如,不能使用 read()和 write()调用来设置终端线路上的波特(baud)率。

对于这样的任务,要使用称为 ioctl()的通用 I/O 控制系统调用(即终端控制函数,Terminal Control Functions):

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

int ioctl(int fd, int cmd, int arg)
```

其中,fd 是文件描述符,cmd 是你想要 ioctl()执行的命令函数,arg 是由 cmd 使用的可选参数。事实上,arg 参数可以是任意 4 字节的数。典型地是一个整型数或一个指向内存块(通常是一个结构)的指针,该内存块含有指定的 cmd 所要求的任何参数数据。

ioctl()调用不仅仅在终端上使用,还可供其他许多 I/O 设备使用。因而,每一类利用 ioctl()的 I/O 设备有它自己的传递给 ioctl()调用的 cmd 和 arg 参数值的集合。由于 POSIX 委员会感到工作过重,在 POSIX.1 标准中,用于终端 I/O 的 ioctl()调用已被一个单独的函数集合所代替。

一般地说,Linux 试图与尽可能多的不同的 UNIX 特点兼容,但总有优先的考虑。在系统之间发生直接冲突的情况下,首先考虑 POSIX 兼容性,然后再考虑最普遍接受的特点。

在终端 I/O 的情况下,已证明可同时提供 POSIX.1 功能及 ioctl()系统调用。因为,两个系统存放终端特征信息的基本数据结构本质上是相同的。

在 Linux 中,有一个称为 termios 的数据结构。它通过 <termios.h> 访问(为了兼容性也可通过 <termio.h> 访问 termio 结构)这些结构有下列基本格式:

```
#define  NCCS  19

struct termios
{
    tcflag_t c_iflag;       /* 输入模式标志(input mode flags)*/
    tcflag_t c_oflag;       /* 输出模式标志(output mode flags)*/
    tcflag_t c_cflag;       /* 控制模式标志(control mode flags)*/
    tcflag_t c_lflag;       /* 局部模式标志(local mode flags)*/
    cc_t c_line;            /* 行的控制(line discipline)*/
    cc_t c_cc[NCCS];        /* 控制字符(control characters)*/
};
```

每个终端设备都有一个这样的结构,包含在这一结构中的标志集合控制终端接口的各种特征。

termios 结构中的每个标志所完成的主要功能是:

c_iflag　在这个集合中的标志允许在输入时进行字符映射。例如,像大写字母转换成小写字母,\r转换成\n,及8位输入转换成7位。这些标志也允许控制输入奇偶检验以及I/O流控制(x-on,x-off)。

c_offag　这些标志允许在输出时进行字符映射。例如将 \n转换成 \r\n,或者制表符(tab)转换成空格(spaces)。它也可能对换页符、\n、水平制表符和垂直制表符等字符增加传输延迟。这适用于旧的慢速打印终端,这些设备需要有足够时间执行某些控制功能。

c_cflag　这些标志控制改变终端线路的波特率,字符位数(每个字符5、6、7或8位)并且允许控制数据奇偶位的生成和检验。

c_lflag　这个特殊的标志集合有一些非常强有力的功能,其中最重要的是:

* 禁止或允许键盘产生信号——是否在输入字符中搜寻 Ctrl-c 和 Ctrl-\。

* 禁止或允许字符的回送功能——回送自动地把输入字符送回输出显示。随着回送功能的关闭,用户可以隐蔽输入的口令等。

* 禁止或允许典型输入——当允许典型输入时,输入字符被自动地输入到缓冲区中以\n结束的行中,并且在缓冲区中退格和删行操作都有效。当禁止典型输入时,read()调用将不等待敲入换行字符,立即将输入的字符送回。

c_cc　这是一个数值的数组,它的功能依赖于是否允许典型输入。如果允许典型输入(即输入是按行缓冲的),则该数组包含诸如文件结束、中断、退出和停止(默认地,这些分别用 Ctrl-d、Ctrl-c、Ctrl-\ 和 Ctrl-z 表示)。如果禁止典型输入,则其中的2个值改变含义,并且一个变为超时(timeout),另一个变为最小字符计数。在 read()返回之前,必须满足其中的一个。通常将这两个值都置为零,这允许输入字符后,这些字符立即被返回 。

在 termios 结构中的所有标志的符号常量通过 < termios.h > 访问。这个文件包含 < linux/termios.h > ,它是这些常量实际被定义的地方。

访问和修改 termios 结构内容的主要 POSIX 函数是:

```
#include <termios.h>

int tcgetattr(int fd, struct termios *tptr)
int tcsetattr(int fd, int action, struct termios *tptr)
```

其中,fd 是文件描述符,tptr 是指向一个 termios 结构的指针,它由 tcgetattr()函数从终端结构中取得,由 tcsetattr()函数将它填入终端结构。tcsetattr()函数的 action 参数的可能值及其含义是:

TCSANOW　立即填入终端结构;

TCSADRAIN　在当前输出缓冲区的内容已送到终端后填入终端结构;

TCSAFLUSH　和 TCSADRAIN 一样,但同时丢弃未读的输入缓冲区的内容。

用 TCGETA 和 TCSETA 作为 cmd 值调用 ioctl(),利用一个指向 termios 结构的指针作为它们的第三个参数也可以得到同样的功能。

下列代码是一对简单函数的例子,使你能够切换标准输入和标准输出(如果它是一个终端)到'扫描(scan)'模式以及相反(即禁止回送、禁止信号、禁止典型模式和设置成从 read()立即返回字符):

```
#include  <termios.h>

struct termios tsave;

void scan_mode(void)
{
  struct termios tbuf;

  if  (!isatty(0))
       fatal("standard input is not a terminal");

  if (tcgetattr(0, &tbuf) == -1)
      fatal("getting terminal attributes");

  tsave = tbuf;
  tbuf.c_lflag &= ~(ECHO | ICANON | ISIG);
  tbuf.c_cc[VMIN] = tbuf.c_cc[VTIME] = 0;

  if (tcsetattr(0, TCSANOW, &tbuf) == -1)
      fatal("setting terminal attributes");
}

void restore_mode(void)
{
  if (tcsetattr(0, TCSANOW, &tsave) == -1)
      fatal("restoring terminal attributes");
}
```

注意,在 scan_mode()中 isatty()函数的用处。isatty()函数的原型为:

```
#include <unistd.h>

int isatty(int fd);
```

其中,fd 是文件描述符。如果 fd 与一个打开的终端相连,则 isatty()返回 1,否则它返回 0。

有一个可供使用的标准命令,它将允许你全面控制 termios 结构中的所有标志;它被称为 stty,并且它的手册页给出了一个完全的标志清单和他们的用法。例如,在前面的代码中所设

置的 scan_mode()可以用一个 stty 命令设置如下：

```
$ stty -echo -icanon -isig min 0 time 0
```

15.3.2 控制终端

每个进程是进程组的一个成员，每个进程组是对话过程(session)的成员。在每一个进程组中有一个进程，它的进程识别号(PID)和它的进程组识别号(PGID)相同。这个进程是它所属的进程组的领头进程(leader)。类似地，每个对话过程有一个进程，它扮演该对话过程的领头进程(见图 15.2)。

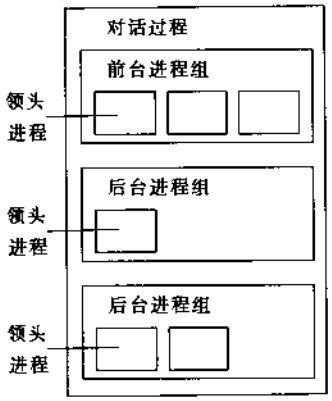

图 15.2 对话过程，进程组和进程

在一个对话过程中的进程通过对话过程的领头进程可以和一个终端相连，该终端扮演这个对话过程的控制终端。一个对话过程仅可以有一个控制终端并且任何终端至多仅可以控制一个对话过程。

如果一个对话过程有一个控制终端，则是这样的终端，从它的键盘所生成的信号(如 Ctrl-c，Ctrl-\ 等)可以被发送到该对话过程中的进程。

在一个对话过程内进程组之一将是前台进程组，在该对话过程内的任何其它进程组将是后台进程组。

仅仅在前台进程组内的进程才被允许从对话过程的控制终端取输入。控制终端的输出也被限于这个进程组使用。

这一切听起来很复杂，但是，当你登录(login)时，这些事全都自动为你设置好。而且一般讲，没有任何要你去改变它们的理由。

如果确实需要设置自己的对话过程和进程组，则有两个函数可以为此目的而使用。第一个称为 setsid()允许设置自己的对话过程：

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

pid_t  setsid(void);
```

一个成功的 setsid()调用创建一个新的对话过程，它包含一个新的进程组，它又包含当前进程作为新的对话过程和新的进程组两者的领头进程。但是，setsid()不能被一个已经是进程组的领头进程的进程成功地调用。在 22.2.3 中我们将看到这个问题的解。

由于一个终端不能用作多于一个对话过程的控制终端，因此，setsid()的副作用是新的对话过程将没有控制终端。因为，如果以前它有一个，那么它应该已经留给旧的对话过程。

如果你想要新的对话过程有一个控制终端，则你所要做的是使该对话过程的领头进程打开(open())一个终端，这个终端应该是不属于其他对话过程的控制终端，并且新对话过程将自动地取该终端作为它的控制终端。

如果新的对话过程需要打开一个终端，但是，你不想要该对话过程取这个终端作为它的控

制终端,则应当在 open()调用中使用 O_NOCTTY 标志。

第二个函数称为 setpgid(),它允许你操作进程组。一般说来,一个进程仅可以改变它自己的 PGID 或者它的子进程之一的 PGID。setpgid()的原型是:

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

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

参数 pid 和 pgid 分别是进程识别号和一个进程组识别号。考虑这些参数值的三种情况:

* pid 和 pgid 都是非零值——pid 进程被移入 pgid 进程组中。pgid 的值必须或者指的是当前对话过程中的一个进程组,或者是和 pid 相同,在后一种情况下,在当前对话过程中创建一个新的进程组,pid 进程将作为它的领头进程。
* pid 是零——除了用调用进程的进程识别号来代替 pid 外,和前面的情况相同。
* pgid 是零——除了用调用进程的进程识别号来代替 pgid 外,也和第一种情况相同。为了和其它 UNIX 特点兼容,还提供另一种调用来改变调用进程的进程组:

```
#include <unistd.h>

int setpgrp(void);
```

在 Linux 下,setpgrp()调用完全等价于下列 setpgid()调用:

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

15.4 索引节点

你已经看到在 Linux 系统中的所有文件和设备都有一个索引节点(inode 即 index node)与它们相连,通过索引节点来访问它们。索引节点本身包含相当多的关于该文件的信息,其中包括:

* 文件使用的设备和索引节点号给出唯一的文件识别(identity);
* 文件的类型(普通文件,目录,特殊文件等);
* 文件的访问权限位;
* 连接计数(连接到这个文件的目录项的个数);
* 文件的用户识别号(UID)和组识别号(GID);
* 设备特殊文件的主和辅助设备号;
* 以字节为单位的文件的容量;
* 文件的最后一次访问/修改/状态改变的时间;
* 访问包含该文件的数据的磁盘块的有关信息。

如果进程知道该文件的路径名,或者如果它有一个指向打开的文件描述的文件描述符,那么,除了最后一项外,所有这些属性毫无疑问都可供进程使用。

为了访问它们,可供进程使用的状态属性被拷贝到 stat 结构中,stat 结构通过 <sys/stat.h> 定义。在这个结构中主要的域是:

```
dev_t       st_dev;     /* 设备号 */
ino_t       st_ino;     /* 索引节点号 */
umode_t     st_mode;    /* 文件类型和权限 */
nlink_t     st_nlink;   /* 连接计数 */
uid_t       st_uid;     /* 文件所有者的识别号(owner ID) */
gid_t       st_gid;     /* 文件组的识别号(group ID) */
dev_t       st_rdev;    /* 设备文件的设备号 */
off_t       st_size;    /* 以字节为单位的文件的容量 */
time_t      st_atime;   /* 最后一次访问的时间 */
time_t      st_mtime;   /* 最后一次修改的时间 */
time_t      st_ctime;   /* 最后一次状态改变的时间 */
```

在这个结构中为 st_mode 域给定值的符号常量通过 <sys/stat.h> 定义。

在这个结构中的信息可以利用三个系统调用 fstat()、stat()或 lstat()之一来恢复:

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

int fstat(int fd, struct stat *sbuf);
int stat(char *pathname, struct stat *sbuf);
int lstat(char *pathname, struct stat *sbuf);
```

对于 fstat(),参数 fd 是指向打开的文件描述的文件描述符,sbuf 是指向 stat 结构的指针,stat 结构将由该调用填入适当的状态细节。

除了第一个参数是文件的路径名而不是文件描述符外,stat()调用和 fstat()相同。

如果文件是一个符号连接,lstat()将给出关于连接文件自身的状态细节,而 stat()将沿着连接并且给出连接所指向的文件的细节。除了这一点外,lstat()调用和 stat()相同。

所有这三个系统调用在成功完成时返回 0 值。在错误时返回 -1 值,并将 errno 置一个相应的值。

下列代码是一个简单的程序,它将以人们可读的形式显示任意给定的文件路径名有关的 stat 结构中的信息:

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

#define  MAJOR(a)  (int)((unsigned short)(a) > > 8)
```

```
#define  MINOR(a)  (int)((unsigned short)(a) & 0xFF)

main(int argc, char **argv)
{
  struct stat sbuf;
  int dev_flag;

  for (; argc > 1; --argc)
  {
    printf("\nFile name: %s\n", argv[argc-1]);

    if (stat(argv[argc-1], &sbuf) == -1)
        fatal("obtaining status details");

    dev_flag = file_type(&sbuf);
    printf("Permission bits: %o\n", sbuf.st_mode & 07777);
    printf("File numbers: ");
    printf("major %d, ", MAJOR(sbuf.st_dev));
    printf("minor %d, ", MINOR(sbuf.st_dev));
    printf("inode %d\n", sbuf.st_ino);
    printf("link count: %d\n", sbuf.st_nlink);

    if (dev_flag)
    {
      printf("Device numbers: ");
      printf("major %d, ", MAJOR(sbuf.st_rdev));
      printf("minor %d\n", MINOR(sbuf.st_rdev));
    }
    else
      printf("File size: %ld\n", sbuf.st_size);

    printf("File owner  ID: %d\n", sbuf.st_uid);
    printf("File group  ID: %d\n", sbuf.st_gid);
  }
}

file_type(struct stat *sbufp)
{
  printf("File type: ");

  switch (sbufp->st_mode & S_IFMT)
  {
  case S_IFREG:
    printf("Ordinary file\n");
```

```
    return 0;

  case S_IFDIR:
    printf("Directory\n");
    return 0;

  case S_IFIFO:
    printf("Named pipe (FIFO)\n");
    return 0;

  case S_IFBLK:
    printf("Block device special file\n");
    return 1;

  case S_IFCHR:
    printf("Character device special file\n");
    return 1;

  default:
    printf("unknoun...\n");
    exit(1);
  }
}

fatal(char *mess)
{
  fprintf(stderr, "Error: %s\n", mess);
  exit(1);
}
```

注意,这个程序能够处理命令行上的多个文件名。从这个程序的典型的输出显示如下:

```
$ file_status /etc ./book.ps /dev/modem

File name: /dev/modem
File type: Character device special file
Permission bits: 660
File numbers: major 3, minor 65, inode 47638
Link count: 1
Device numbers: major 5, minor 64
File owner ID: 0
File group ID: 14

File name: ./book.ps
```

```
File type: Ordinary file
Permission bits: 644
File numbers: major 3, minor 66, inode 34876
Link count: 1
File size: 1049173
File owner ID: 500
File group ID: 500

File name: /etc
File type: Directory
Permission bits: 755
File numbers: major 3, minor 65, inode 45701
Link count: 8
File size: 2048
File owner ID: 0
File group ID: 0
```

15.5 改变文件属性

在所有能收集到的关于文件的信息中,仅有少量信息可以改变。本节将讨论你能做的大部分事情。

15.5.1 chmod 和 fchmod 系统调用

第一个讨论的是文件权限位。用于改变它们的系统调用称为 chmod()和 fchmod()。可以由这些系统调用改变的位完全和那些可以由 chmod 命令改变的位相同。这包括:文件所有者、用户组和其他用户类的读、写和执行位。它还包含 setuid 位和 setgid 位。chmod()和 fchmod()的原型是:

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

int chmod(char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

chmod()调用用来改变给定路径名 pathname 的文件的权限位,而 fchmod()用来改变给定文件描述符 fd 的文件的权限位。

注意,权限位参数 mode 使用 POSIX 数据类型 mode_t,因此,为了真正做到可移植,应当使用在 <sys/stat.h> 中定义的权限位掩码(mask)的符号名而不使用八进制数表示。但是,两种方式 Linux 都能接受。下列两个 fchmod()调用完全执行同样的功能:

```
fchmod(fd, S_ISUID | S_IRWXU | S_IXGRP | S_IXOTH);
```

```
fchmod(fd, 04711);
```

如果执行成功,chmod()和 fchmod()这两个系统调用都返回 0 值。在错误时,都返回 -1 值,并在 errno 中置相应的错误值。

15.5.2 umask 系统调用

当改变一个文件的权限位时,不要忘记,你所规定的位将自动地由当前的 umask 值按照下列公式加以修改:

```
mode & (~umask)
```

以前见到的 umask 命令的功能在系统程序设计中是由 umask()系统调用提供的:

```
#include <unistd.h>

mode_t umask(mode_t mask);
```

和 chmod() 一样,mask 参数用掩码(mask)的符号名集合加以规定。

从 umask()调用返回的值是用你的新 mask 参数所代替的 umask 当前值。这就允许你用如下的方式找出当前的 umask 值:

```
oldmask = umask(0);
(void) umask(oldmask);
```

第二个 umask()调用用来恢复被找出 umask 值。

15.5.3 chown 和 fchown 系统调用

chown()和 fchown()系统调用用来改变文件所有者的识别号(owner ID)或者它的用户组识别号(group ID)。这些调用的原型是:

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

int chown(char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

chown()调用给路径名为 pathname 的文件赋予新的所有者识别号和组识别号。fchown()调用给由文件描述符 fd 所指定的文件赋予新的所有者识别号和组识别号。

在 Linux 下,只有 root 帐号可以使用 chown()和 fchown()系统调用。

15.5.4 fcntl 系统调用

另一项可以加以检验和改变的是由 open()调用的第二个参数所设置的标志。做这件事的系统调用称为 fcntl()。显然,由于它对文件被打开时所设置的标志进行操作, fcntl()调用仅有使用文件描述符的形式,而没有使用路径名的形式。fcntl()的原型是:

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

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long setval);
```

fcntl()调用可以提供各种文件控制功能的完整集合。这些功能通过文件描述符 fd 对指定文件进行操作。这里我们将仅讨论其中的两个(在后面的各章中,我们再讨论其余的功能)。

fcntl()调用取 2 个或 3 个参数,这取决于命令(cmd) 使用的参数值。一般讲,取某些属性或标志值的命令使用 fcntl()的两个参数的形式,并且提供所要求的值作为它的返回值。类似地,设置某些属性或标志值的命令典型地使用 fcntl()的 3 个参数形式,并且用 setval 参数设置属性或标志。

用于取和设置 open()某些标志的两个命令是:

F_GETFL 用 fd 取有关的标志和访问模式;
F_SETFL 用 fd 设置有关的标志。

F_GETFL 命令可以返回任何由 open()调用所设置的标志和访问模式(O_RDONLY、O_WRONLY、O_APPEND 等)。但是,F_SETFL 命令仅可以设置或者重置 O_APPEND 和 O_NONBLOCK 标志。

由 F_GETFL 命令返回的访问模式值不是一位(single-bit)的标志。测试一个特定的访问模式值是否设置的最安全的方法是使用下列形式的测试(在这个例子中测试 O_RDONLY)

```
if ((fcntl(fd, F_GETFL) & O_ACCMODE) == O_RDONLY)
    /* fd是否为只读打开的代码 */
```

其中,O_ACCMODE 是专门用来从 fcntl()的返回值中取出访问模式位的掩码。

15.5.5 truncate 和 ftruncate 系统调用

本节中最后的系统调用可以用于将文件截成规定的长度,文件用路径名或者文件描述符加以指定:

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

```
int truncate(char *pathname, size_t len);
int ftruncate(int fd, size_t len);
```

如果在这些调用中所指定的文件长度超过 len 个字节,则它将被截断成规定的长度,并且将丢失多余的字节。

为了使 ftruncate()调用成功地进行操作,fd 应当是指向为了写(即 O_WRONLY 或者 O_RDWR)而打开的打开文件描述。

15.6 文件系统的层次结构

为了创建、修改和维护 Linux 文件和目录的层次结构,必须能够创建和移动目录以及在目录中创建和移动对文件的连接(links)。所有这些事情很容易完成,特别因为它们都有它们自己的系统调用。

15.6.1 mkdir 和 rmdir 系统调用

Linux 进程可以用 mkdir()系统调用创建一个目录。mkdir()调用的原型是:

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

int mkdir(char *pathname, mode_t mode);
```

假定你的进程有合适的权限,一个 mkdir()调用将创建一个称为 pathname 的新目录,它的权限位置成 mode。mode 参数的值由当前的 umask 值以通常的方式加以修改。新目录将为调用进程的有效用户识别号所拥有,并且它将自动地被初始化为包含点(dot)和点点(dot-dot)项目(entries)。

如果一个目录是空的(除去点和点点项目),则它可以用 rmdir()系统调用从目录层次结构中移去:

```
#include  <unistd.h>

int rmdir(char *pathname);
```

15.6.2 目录访问(Directory Access)

由于目录在文件系统层次结构中的特殊重要性,要求用专门的系统调用来访问它们的内容。为了打开一个目录,你需要使用 opendir()系统调用,其原型为:

```
#include  <dirent.h>
```

```
DIR  *opendir(char  *pathname);
```

这个调用打开指定的目录,并且如果打开成功,则返回一个目录指针。如果错误则返回零值。目录指针被传递给有关的系统调用,主要是 readdir()和 closedir()。

readdir()系统调用有原型:

```
#include  <dirent.h>

struct dirent  *readdir(DIR  *dirptr);
```

这个调用返回一个指向 dirent 结构的指针,它包含指定目录中下一个连接的细节。在该连接中的文件名被存放在这个结构的 d_name 域中,它是一个字符数组。重复地调用 readdir(),将导致在目录中顺序访问所有连接。当没有更多的连接时,返回零值。

一旦你已经结束一个目录,可以用 closedir()系统调用关闭它。该调用的原型是:

```
#include  <dirent.h>

int closedir(DIR  *dirptr);
```

15.6.3 link 和 unlink 系统调用

文件的一个目录项目称为一个连接(link)。当文件被首次创建时,从它的给定路径名自动地产生一个文件的连接。因此,对这个文件的其他连接可以由 link()系统调用产生:

```
#include  <unistd.h>

int link(char  *pathname1, char  *pathname2);
```

一个称为 pathname2 的新的目录连接被创建,它指向已有的 pathname1 文件。为了使link()调用正常工作,pathname1 和 pathname2 两者都必须在同一个文件系统中,并且调用进程必须有写到目标目录的权限。

文件的连接通常可以用 unlink()系统调用移去。

```
#include <unistd.h>

int unlink(char  *pathname)
```

这个调用移去指定的目录连接,并且在它的索引节点中对该文件的连接计数减 1。而且,如果连接计数已经变成零,则索引节点和文件的数据块全都释放给系统。如果当它为了移去目录连接而被调用时,某个进程必须使该文件打开,则索引节点和文件的数据块的释放将被延迟到该进程结束为止。但是,该目录连接仍然将被移去,没有其它进程可以再打开该文件和进一步延长它的生命。

练习

1. 写一个简化的 ls 命令，它只取单个目录名作为它的唯一命令行参数，并列出该目录中的所有的文件名，每行一个文件名。如果你不打算对文件名清单进行排序(sort)，它将是比较简单的。
2. 写一个程序，它取得与你的终端有关的 termios 结构的副本，并且以适合人们阅读的形式显示它所发现的设置值。
3. 写一个 C 程序，它提示用户送入用户名和口令，并且从用户响应提示的输入中提取字符串值。确保输入口令时抑制字符回送，并且在你的程序打印它所得到的值之前恢复回送，然后结束。
4. 写一个简单的 rename 命令，它将允许你重新命名在当前的目录中你所拥有的任何文件。该命令的格式应当是：

```
$ rename <oldname> <newname>
```

解答

1. 下列的代码非常简单，部分由于它不进行任何通常要求的检验，如命令行参数的个数和类型，以及检验系统调用的操作的正确性等。但是，它确实使代码非常容易阅读：

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

main(int argc, char **argv)
{
  DIR *dp;
  struct dirent *link;

  dp = opendir(argv[1]);

  while ((link = readdir(dp))! =0)
      printf("%s\n", link->d_name);

  closedir(dp);
}
```

2. 解决这个问题的方法是使用 ioctl()或者 tcgetattr()取与该终端有关的结构的内容。所需要的关于在该结构中的每个标志和字节的意义和值的信息包含在/usr/include/linux/termios.h 头文件中，某些其他的信息在 stty 命令的手册页中。从这些信息可以相当容易地找出哪些标志被设置和未被设置，以及它们的效果是什么。然后，可以以任何适当的形式显示出来。
3. 下列的 shell script 执行所要求的任务，并且作为伪码程序，有助于你写出 C 语言程序：

```
echo -n "enter name: "
read name
stty -echo
echo -n "enter passuord: "
read password
```

```
stty echo
echo
echo name: $name
echo password: $password
```

4. 这个问题的伪码程序如下：

```
main(argc, argv)
    if argc! = 3
        give error and usage message and exit
    if argv[1] or argv[2] outside current directory
        i. e. if they contain a '/', give error and exit
    if argv[1] not in current directory
        give error and exit
    link(argv[1], argv[2])
    unlink(argv[1])
```

C程序的解应该遵循这个基本模式。试图处理你能找到的尽可能多的错误条件，包括检测从系统调用返回的错误。

第十六章　进 程 控 制

Linux 是真正的多任务操作系统。如果你有一项应用,需要几个进程并发地协调去完成,则 Linux 可以安排它们并发运行。并且为你完成进程的全部调度和管理任务。事实上,你已经看到这种情况。当使用 shell 运行像下面的管道行时:

```
$ ls | wc -w
```

它实际包含多个命令(在这个例子中是 ls 和 wc)并发运行,而且在它们之间带有通信通道,使得它们可以协调一致地去完成单一的任务。当 shell 为你安排并发进程的运行时,它仅使用一些普通用户的功能(facilities),在自己的程序中也能自由使用这些功能。这些功能构成了本章的主题。

16.1　进程识别号

在 Linux 系统上运行的每个进程都有一个唯一的进程识别号(Process Identity Numbers, PID)。从 1 号进程(init)的后面开始,每个进程都有一个双亲(parent)进程,它也可以访问双亲进程的进程识别号。

每个进程属于一个进程组,进程组具有进程组的识别号,它正好是该进程组的领头进程的识别号。

当解决进程访问不同的文件具有什么样的权限时,进程使用另外 4 个识别号的集合。这些是实际用户识别号(real user ID)和实际组识别号(real group ID)以及有效用户识别号(effective user ID)和有效组识别号(effective group ID)。一个进程的实际用户识别号和实际组识别号正好是运行该进程的用户的用户识别号(UID)和组识别号(GID)。除了程序设置 setuid 或 setgid 位的情况以外,有效识别号(effective IDs)通常和实际识别号相同。如果 setuid 位和 setgid 位之一或者两者都被设置,则相应的有效用户识别号或有效组识别号将被设置成进程正在运行的程序文件的文件所有者识别号或文件用户组识别号。

这些听起来很复杂,所以用一个例子来说明。假设一个具有 UID 为 200 和 GID 为 20 的用户运行/usr/bin/passwd 程序。这个程序的文件所有者(root)的识别号为 0,文件组识别号为 1(bin),同时它设置了 setuid 位。

当 passwd 程序运行时,进程的实际用户识别号为 200,实际组识别号为 20。因为 setuid 位被设置,有效用户识别号为 0。而有效组识别号仍为 20。

实际识别号用于识别正在运行该进程的用户。有效识别号则用于按照下列算法(从规则 1 开始)解决访问文件时进程具有的优先权(privileges)和权限:

规则 1　如果进程的有效用户识别号是 0(root),则进程被自动地授予做最多事情的权限。如果不是,则转规则 2。

规则 2 如果进程的有效用户识别号和文件的用户识别号相同，则授予进程按照文件的所有者权限位访问该文件。如果不同，则转规则 3。

规则 3 如果进程的有效组识别号是和文件的组识别号相同，则授予进程按照文件的组权限位访问该文件。如果不同，则转规则 4。

规则 4 授予进程按照文件的其他用户权限位访问该文件。

所有与进程有关的不同的识别号可以通过下列的‘get ID’系统调用系列进行访问：

```
uid_t getuid(void)      /* 获得实际用户识别号 */
uid_t getgid(void)      /* 获得实际组识别号 */
uid_t geteuid(void)     /* 获得有效用户识别号 */
uid_t getegid(void)     /* 获得有效组识别号 */
pid_t getpid(void)      /* 获得进程识别号 */
pid_t getppid(void)     /* 获得双亲进程识别号 */
pid_t getpgrp(void)     /* 获得进程组识别号 */
```

16.2 用 fork 创建进程

为了进行讨论，我们将考虑进程由三个独立的部分组成(如图 16.1 所示)。

正文段存放被执行的机器指令。这个段是只读的(所以，在这里不能写自己能修改的代码)，它允许系统中正在运行同样程序的两个或多个进程之间能够共享这一代码。例如，如果几个用户全都在运行 bash 作为它们的 shell，在内存中仅需要该程序指令的一个副本，他们全都共享这一副本。

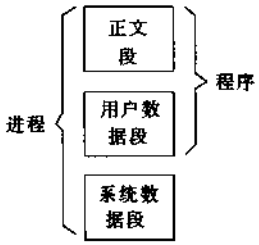

图 16.1 程序和进程的组成部分.

用户数据段存放进程在执行时直接进行操作的所有数据，包括进程使用的全部变量在内。显然，这里所包含的信息可以被改变。虽然，进程之间可以共享正文段，每个进程需要有它自己的专用用户数据段。

系统数据段有效地存放程序运行的环境。事实上，这正是程序和进程之间有区别的地方。程序是在磁盘上由一组指令和数据组成的静态事物，它们是进程最初使用的正文段和用户数据段。进程是动态事物，执行进程的环境要求将正文段、用户数据段和系统数据段的信息全都相互交织在一起。

在 Linux 系统中，一个已有的进程只有一种方法可以为你启动一个新的进程，这就是使用 fork()系统调用：

```
#include  <unistd.h>

pid_t fork(void);
```

从概念上讲，fork()调用的作用是使调用 fork()的进程变成新创建的进程的双亲。两个进程的正文段、用户数据段的内容完全相同，并且它们的系统数据段的内容也几乎完全相同。进

程之间的唯一差别是少量属性必须不同(例如,每个进程的 PID 必须是唯一的)。一旦子女进程已经被创建,则双亲和子女两个进程都从 fork()调用内部继续执行。这意味着两个进程的下一个动作是带有它的返回值从 fork()返回。

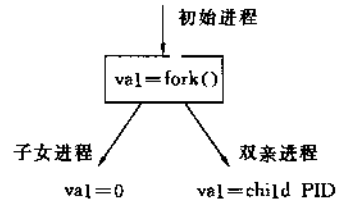

图 16.2 fork()系统调用的作用

除非有某种方法使它们接着执行不同的动作,似乎没有办法使得两个几乎完全相同的进程运行。fork()给两个进程返回不同的值使这件事变得相对地简单。对双亲进程它返回新创建的子女进程的进程识别号(PID),而对子女进程它返回值 0。因为正常的进程识别号从 init 进程所具有的 1 号开始编号,fork() 系统调用不可能给双亲进程返回 0 值作为新创建的子女进程的进程识别号(PID)。因此,如果 fork()确实返回一个 0 值,则它必须给新的子女进程。如果它返回一个非零值,则它必须是返回给双亲进程的子女进程的进程识别号(PID)(或者在错误时,返回 -1 值)。这一概念表示在图 16.2 中。

下面是一个说明 fork()调用的作用的简单程序:

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

main()
{
    pid_t val;

    printf("PID before fork(): %d\n", (int) getpid());

    if (val = fork())
       printf("Parent PID: %d\n", (int) getpid());
    else
        printf("Child PID: %d\n", (int) getpid());
}
```

当这个程序运行时,它应当产生 3 行输出。第一行将显示在 fork()调用被执行前进程的进程识别号(PID),其余两个输出行将在 fork()调用被执行后由双亲进程和子女进程产生,给出每个进程的进程识别号(PID)。这个程序的一个典型的输出可能是:

```
$ forktest
PID before fork(): 490
Parent PID: 490
Child PID: 491
```

在 fork()调用执行后,双亲进程的大部分属性不变,仍然可供子女进程使用。主要的不变的属性有:

* 对话过程和进程组成员；
* 控制终端(如果有的话)；
* 实际和有效的用户识别号和组识别号；
* 当前工作目录；
* 文件权限位创建掩码(umask)。

除此以外，在双亲进程中指向打开文件描述的所有文件描述符将被复制到子女进程中。这意味着子女进程的文件描述符和双亲进程的文件描述符全都指向同样的打开文件描述(如图 16.3 所示)。

这是一个非常重要的概念。我们稍后将看到，由于它允许进程事先打开文件，子女进程知道文件将被预先打开(pre-opened)，在 fork()调用后可以立即使用它们。

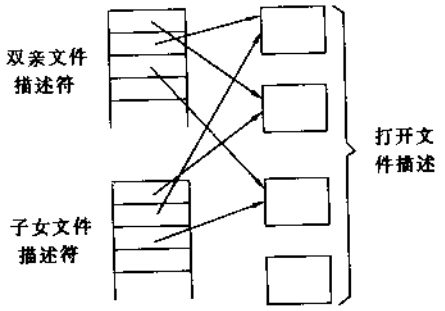

图 16.3 双亲进程和子女进程共享打开文件描述

16.3 exec 系统调用

当你用 shell 运行一条命令(譬如 ls)时，则在某一时刻，shell 将执行 fork()调用使一个新进程运行。做完以后，shell 如何使 ls 在新进程中运行以代替在 fork()调用后立即运行 shell 的副本?

在此情况下，相应使用 exec()系统调用来解决问题。事实上，有若干不同形式的 exec()调用。但从实质讲，它们全都执行同样的任务。在 Linux 系统中，使用 exec()系统调用是使程序执行的唯一方法。具体做法是：在使用 exec()系统调用时，将程序文件的名称作为参数传递给 exec()。然后，它将包含在程序文件中的正文和用户数据段来代替执行 exec()调用的进程的正文和用户数据段。可能最好用一个简单的例子加以进一步说明。

在我们进一步说明之前，需要先解释 exec()调用。当你从 shell 执行一个程序时，需要在命令行中为该程序指定参数和开关。从 C 语言的知识，你也知道这些命令行的值是通过 main()函数的参数 argc 和 argv 提供给程序使用。shell 需要能以某种方式取你的命令行的值，并且把它们作为 argc 和 argv 传递给为你运行的程序。这一任务由 exec()调用来完成。将你的命令行的值以适当的形式传递给它，它将安排它们作为将要运行的新程序的 argc 和 argv。

可以用 exec()调用的最简单的形式 execl()来说明这一点。其原型是：

```
#include  <unistd.h>

int execl(char *pathname, char *arg0, ...);
```

其中 pathname 是要执行的命令的完整路径名，后面是可变长的指向字符串的指针清单。这些将成为在新程序中由 argv 所指向的数组的内容。在 execl()中的指针清单应该以一个 NULL 指针(0)结束。下列的程序使用 execl()来执行简单的 ls -l 命令：

```
#include  <stdio.h>

main()
{
    execl("/bin/ls", "ls", "-l", 0);
    printf("Can only get here on error \n");
}
```

在这个例子中,execl()的第一个参数是 ls 命令的完整路径名。假如进程对该文件有执行权限,这一文件的内容将被运行。新程序中的 argv 数组元素所指向字符串将由 execl()的其余参数提供。在这个例子中,ls 程序将看见由它的 argv[0] 元素所指向的字符串 ls,和由它的 argv[1] 元素所指向的字符串 -l。

通常,exec()调用并不返回。一般来说,它们不能返回。因为它们的功能是用某个其它程序的正文和用户数据段代替调用它们的进程的正文和用户数据段,因此,将没有什么东西需要返回。

但是,如果对 exec()调用由于任何原因而失效(通常由于你试图执行的文件不存在,或者你没有执行它的权限),则它们将返回,使你能有机会对错误进行处理。

除了使所有这些参数可供新程序使用之外,exec()调用也为下列变量传递一个值:

```
extern char **environ;
```

除了通过 environ 传递的项是在进程的环境中的值(像任何输出的 shell 变量一样)而不是命令行参数外,这个变量和 argv 变量具有同样的格式。在 execl()调用的情况下,在新程序中的 environ 变量的值将是在调用进程中这个变量值的拷贝。

在你可以像前面的例子那样明显地列出所有参数的情况下,使用 execl()调用是好的。现在假设你要写一个程序,它将不仅仅运行 ls,而且将运行任何你希望运行的程序,并且传递任意个适当的命令行参数。显而易见,execl()将不能做这个工作。

下面是实现这一要求的程序的例子,说明 execv()系统调用将能完成所要求的任务:

```
#include  <stdio.h>

main(int argc, char **argv)
{
    if (argc == 1)
    {
        printf("Usage: run <command> [<parameters>]\n");
        exit(1);
    }

    execv(argv[1], &argv[1]);
    printf("Sorry… couldn't run that! \n");
}
```

execv()的原型说明它仅取两个参数,第一个是要执行的命令的完整路径名,第二个是你传递给新程序的 argv 值。在前面的例子中这个值来源于传递给 run 命令的 argv 值。使得 run 命令可以取你传递给它的命令行参数值,并且将它们传递给 execv()调用。

下面是一个典型的利用 run 命令的例子:

```
$ run ls -l mtos
Sorry... couldn't run that!
$ run /bin/ls -l mtos
total 2
drwxr-xr-x  2   pc   book      1024  Apr  2  20:11  tdd
drwxr-xr-x  2   pc   book      1024  Apr  2  20:11  tsh
```

注意,第一次运行没有成功。这是由于你使用 execv()(或 execl())调用时,没有指定你想运行的命令的完整路径名。第二次运行指定了完整路径名运行就取得成功。

这里问题是到目前为止用到的两种形式的 exec()调用在搜寻你所指定的命令时,都没有使用你的 PATH 环境变量中值。但是,正如你可能猜到的,存在另一种形式的 exec()调用。它们被称为 execlp()和 execvp()调用,它们和第一对完全相同,不过它们使用 PATH 变量的值来找出所指定的命令。

最后两种形式的 exec()调用和头两种相同(即它们不使用 PATH 变量的值),但是,它们允许你手工地规定在新程序的 environ 变量中所出现的值,而不是自动接受默认值。它们被称为 execle()和 execve()。下列是全部 6 种形式的 exec()调用以及它们所取的参数个数和类型的清单:

```
int execl(pathname, arg0, ..., argn, 0);
int execv(pathname, argv);
int execlp(cmdname, arg0, ..., argn, 0);
int execvp(cmdname, argv);
int execle(pathname, arg0, ..., argn, 0, envp);
int execve(pathname, argv, envp);

char *pathname, *cmdname;
char *arg0, ..., *argn;
char **argv, **envp;
```

正如 fork()调用一样,进程的大多数属性在通过 exec()调用时仍然被保留。这是由于在执行 exec()调用期间,系统数据段继续保持完整,仅仅改变了正文和用户数据段。最重要的是在调用之前文件描述符所指的打开文件描述在调用后通常仍然可供使用。例外的情况是在每个文件描述符(不是打开文件描述)中存在一个标志,它被称为在执行时关闭(close on exec)标志(猜一猜它用来干什么)。

由 fcntl()调用所提供的功能之一设置在执行时关闭(close on exec)标志。在一个特定的文件描述符(fd)上设置在执行时关闭标志的调用形式是:

fcntl(fd, F_SETFD, FD_CLOEXEC);

回顾一下,当你希望一个运行的进程组织另一个程序的执行作为它的操作的一部分时,具体步骤如下:

* 运行的进程取得或产生要执行的命令。
* 运行的进程执行 fork()系统调用。它启动一个新进程,它是原来的进程的拷贝,并且与原来的进程并发地运行。原来的进程被称为双亲进程,新进程被称为子女进程。
* 现在子女进程执行 exec()调用,它去掉子女进程的正文和用户数据段并且用将要运行的在命令文件中的正文和用户数据段代替它们。

在讨论这一情况时,有两个明显的问题。首先,用双亲进程的完整的复制品来创建子女进程,仅仅为了立即去掉它并用新程序代替它是否是一种浪费?其次,当子女进程正在执行时,双亲进程又在干什么?

我们依次讨论这两个问题。Linux 以一种非常有效的方式实现它的 fork()系统调用。事实上,Linux 实际并没有这么去做。它并没复制进程的完整拷贝,它只用两组指针,每个进程一组,全都指向同一实际的正文和数据段。在正文段情况下,这无关紧要,因为该段是只读的。对数据段来说,它就关系重要,因为这是一对独立的进程。但是,只要两个进程仅仅从数据段读数据段而不写任何新值,就不会发生问题。为了使这一方案能完整地工作,Linux 必须做的只是:当两个进程之一企图将数据写到数据段之一时能发现它,并且按需要对数据段的这一区域进行复制。这一技术被称为在写时复制(copy on write)。

从原则上讲,由任一个进程写的任何的区域将得到复制,使得两个进程都有它们自己的这些区域的副本。如果子女进程立即执行 exec()调用,则在执行 exec()之前只有非常少量的共享数据段被复制,因而大量节省了时间和资源。

16.4 wait 和 exit 系统调用

当子女进程运行时,双亲进程在干什么?这个问题的回答是相当简单的,或者它等待子女进程的结束,或者它继续做它需要做的任何事情。

例如,在 shell 的情况下,由用户来进行选择。通常,如果你只给 shell 送入一条命令,它在给你送回另一个提示符以前将等待该命令的结束。如果你在命令行的后面加一 & 字符,则 shell 将不等待命令的结束,立即送回一个新提示符。

为了等待子女进程的结束,双亲进程将只执行 wait()系统调用。这个调用将挂起双亲进程,直到它的任何子女进程结束,此时 wait()调用返回,并且双亲进程可以继续。

wait()调用的原型是:

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

pid_t wait(int *status);
```

从 wait()返回值是结束的子女进程的进程识别号(PID)。wait()的参数是指向一个位置的

指针，当子女进程结束时将在这一位置接收它的退出状态值。

当一个进程结束时，它或者直接地在它自己的程序中或者间接地通过库程序执行一个 exit()系统调用。exit()调用的原型是：

```
#include  <stdlib.h>

void exit(int status);
```

当调用 exit()的进程结束时，exit()调用没有返回值，所以无论如何不可能收到一个值。但是，注意 exit()确实取一个 status 参数值。除了使等待的双亲进程恢复执行之外，exit()也通过 wait()参数所指向的位置返回 status 参数值给双亲进程。

事实上，wait()可以通过 status 参数所指的值返回几个不同的信息。因而，提供了一个被称为 WEXITSTATUS()的宏(通过 < sys/wait.h > 访问)，它可以抽取并返回子女进程的退出状态。下列的代码段说明了它的用法：

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

int statval, exstat;
pid_t pid;

pid = wait(&statval);
exstat = WEXITSTATUS (statval) ;
```

现在能够画出一个子女进程的生命周期图。从由 fork()创建开始，当双亲进程用 wait()被挂起时，通过 exec()而执行，然后用 exit()结束它的生命。这个过程表示在图 16.4 中。

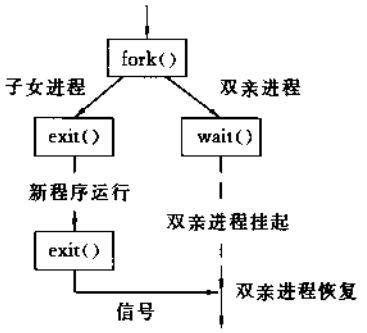

图 16.4　一个子女进程的生命周期

事实上，我们刚刚看到的 wait()调用仅仅是 Linux 系统中可供使用的最简单的形式。新的 POSIX 版本中被称为 waitpid()。waitpid()的原型是：

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

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

其中 pid 用来指定等待什么，status 和简单的 wait()调用的参数相同，如果没有子女进程准备好报告它的退出状态，options 允许你规定对 waitpid()的调用不挂起双亲进程。

pid 参数的各种可能值是：

pid	具体意义

< -1	等待进程组识别号(PGID)为 - pid 的子女进程;
-1	相当于标准的 wait()调用;
0	等待进程组识别号(PGID)和调用进程的进程组识别号相同的的子女进程;
> 0	等待进程识别号(PID)为 pid 的子女进程。

标准的 wait()调用现在是多余的,因为,它和下列 waitpid()调用是完全等价的:

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

int statval;
pid_t pid;

pid = waitpid(-1, &statval, 0);
```

有可能一个子女进程的执行时间非常短,在它的双亲进程有机会等待它的退出状态之前已经结束。在这种情况下,子女进程将进入所谓的僵死(zombie)状态。在此状态下,除了保存它的退出状态的进程数据结构外,它的所有其他资源全部释放给系统。当双亲进程最终使用 wait()调用等待子女进程时,退出状态立即被传递,然后该进程数据结构也被释放给系统。

练习

1. 写一个 C 程序用来确定它是否是进程组的领头进程,并打印适当的信息。
2. 写一个 C 程序用来确定它是否正在运行设置 UID 的程序,并打印适当的信息。
3. 用 fork()写一个简单测试程序,从双亲和子女进程打印信息。信息应该包括字 parent 和 child,也应该包括打印进程的进程识别号(PID)。执行程序若干次。看两个信息是否始终以同样的次序打印?
4. 把 wait()和 exit()系统调用加到前一个练习的程序中,使子女进程返回退出状态给双亲进程,并将它包含在双亲进程的打印信息中。执行程序若干次。看信息的打印次序是否和前一个练习不同?是否两个信息始终以同样的次序打印?
5. 写一个 C 程序,它将执行下列 Linux 命令序列:

```
cp /etc/fstab
sort fstab -o myfstab
cat myfstab
```

解答

1. 你的解将使用 getpid()和 getpgrp()得到进程的进程识别号(PID)和进程组识别号(GID)。如果这些是相同的,则该进程是一个进程组的领头进程,否则就不是。
2. 你的解将使用 getuid()和 geteuid()得到进程的实际用户识别号和有效用户识别号。如果这些是不相同的,则该进程正在运行设置 uid 的程序,否则就不是。
3. 这个问题的程序非常容易写:

```
main()
{
  if (fork())
```

```
        printf("parent PID = %d\n", getpid());
    else
        printf("child PID = %d\n", getpid());
}
```

一般说来，一些UNIX版本将首先运行双亲进程，其它的将首先运行子女进程，还有一些没有固定的次序。在Linux中，当前的fork()源代码表明它始终总首先运行双亲进程，你的实验应当证实这一点。

4. 把exit()和wait()加到程序的全部含义是强迫双亲进程在它继续执行之前，等待子女进程执行到结束。新的程序如下：

```
#include <sys/uait.h>
main()
{
    int status;
    if (fork())

    {
      wait(&status);
      printf("parent PID = %d, child ", getpid());
      printf("exit status = %d\n", WEXITSTATUS(statval));
    }
    else
    {
      printf("child PID = %d\n", getpid());
      exit(55);
    }
}
```

当这一程序运行时，使用exit()和wait()保证将始终首先显示子女进程信息，这不仅适于Linux，也适用于任何UNIX版本。

5. 保证这三个命令按既定次序执行的最好方法是依次用fork()创建执行每条命令的子女进程，使双亲进程在执行下一条命令之前，等待前一个子女进程的结束：

```
main()
{
    if (! fork())
    {
        execlp("cp", "cp", "/etc/fstab", ".", 0);
        printf("error executing cp\n");
        exit(1);
    }

    wait(0);
```

```
    if (! fork())
    {
        execlp("sort", "sort", "fstab", "-o", "myfstab", 0);
        printf("error executing sort \n");
        exit(1);
    }

    wait(0);
    execlp("cat", "cat", "myfstab", 0);
    printf("error executing cat \n");
    exit(1);
}
```

为了简化代码,这里没有检验 fork()的返回值是否是一个错误值。检查返回给 wait()退出状态值是否是一个错误值并且适当地处理它,这是一个好的想法。我把这件事留给你去做。

第十七章　进程间通信

为了进程能在同一项任务上协调工作，它们彼此之间必须能够进行通信。Linux 支持许多不同形式的进程间通信(IPC)机制。在特定的情况下，它们各有自己的优缺点。这一章将讨论最有用的进程间通信机制，使你在写协调工作的并发进程时可以作出明智的选择。

在进程之间通信的最简单的方法是通过一个文件。一个进程写文件，另一个进程从这个文件中读。这是出人意料地简单的机制，它有几个优点：

* 它允许任何对该文件具有访问权限的一对进程用它来进行通信。
* 它允许在进程之间传递非常大量的数据。

尽管有这些优点，使用文件作为主要的进程间通信机制有两大缺点：

* 为了确保读进程有机会看到所有送给它的数据，写进程必须仅仅把新的数据加到文件的结尾。这表示对长时间存在的进程来说，文件可能变得非常大。
* 如果读进程执行它的任务比写进程快，则它将经常赶上写进程，并且读不到任何东西。这表示读进程将不断地读文件的结尾，并且不知道是否这是写进程已经结束并关闭了文件，还是写进程仍旧在计算传给读进程的值，后面还有更多的数据。

克服使用文件的固有问题的最简单的进程间通信机制是管道(Pipe)。

17.1　管道

你在前面已经遇到过管道，即使你还没有写任何使用管道的程序。shell 一直在使用它们。当你用它执行管道行命令时，其中一个进程的输出成为另一个进程的输入：

```
$ ls | wc -w
```

使用管道克服了使用文件进行通信的两个问题。首先，管道是一个固定大小的缓冲区(在 Linux 系统中为 4096 字节)，使得它的大小不像文件那样不加检验地增长。

使用单个固定大小的缓冲区的后果是在写管道时可能变满。当这种情况发生时，随后对管道的 write()调用将默认地被阻塞，等待某些数据被读取，以便腾出足够的空间供 write()调用写。注意，从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃，释放空间以便写更多的数据。

其次，读进程也可能工作得比写进程快。当所有当前数据已经被读时，管道变空。当这种情况发生时，一个随后的 read()调用将默认地被阻塞，等待某些数据被写入。这解决了 read()调用返回文件结束的问题。

管道是利用 pipe()系统调用而不是利用 open()创建的。因为，要求使用不同的参数。而

且,在此情况下,open()调用不适合进行这种服务。pipe()调用的原型是:

```
#include  <unistd.h>

int  pipe(int  fd[2]);
```

有两个文件描述符与管道结合在一起,一个文件描述符用于管道的 read()端,一个文件描述符用于管道的 write()端。由于一个函数调用不能返回两个值,pipe()的参数是指向两个元素的整型数组的指针,它将由 pipe()调用用两个所要求的文件描述符填入。fd[0]元素将含有管道 read()端的文件描述符,而 fd[1] 含有管道 write()端的文件描述符。

注意,没有给出路径名作为 pipe()调用的参数。这表示,当创建一个管道时,没有为它创建一个目录连接。因此,其它现存的进程无法得到该管道的文件描述符,从而不能访问它。然而,两个进程如何使用一个管道来通信?

回顾一下,fork()和 exec()系统调用可以保证文件描述符的复制品既可供双亲进程使用,也可供它的子女进程使用。这一信息给出了所要求的机制。具体做法是一个进程用 pipe()系统调用创建管道,然后用 fork()调用创建一个或多个进程,管道的文件描述符将可供所有这些进程使用。

这里的含义是:一个普通的管道仅可供具有共同祖先的两个进程之间共享,并且这个祖先必须已经创建了供它们俩使用的管道。

在管道中的数据始终以和写数据相同的次序来进行读。这表示 lseek()系统调用对管道无作用。

在两个进程之间设置和使用管道的简单的程序如下:

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

main()
{
    int  fda[2];
    char  buf[1];

    if (pipe(fda) == -1)                    /* 创建管道 */
        fatal("creating pipe");

    switch (fork())
    {
    case -1:
        fatal("forking child");
        break;

    case 0:                                 /* 子女进程是管道的读进程 */
        close(fda[1]);                      /* 关闭管道的写端 */
```

```
            read(fda[0], buf, 1);
            printf("%c\n", buf[0]);
            break;

        default:                                /* 双亲进程是管道的写进程 */
            close(fda[0]);                      /* 关闭管道的读端 */
            write(fda[1], "a", 1);
            break;
        }
}

fatal(char  *mess)
{
    fprintf(stderr, "Error: %s\n", mess);
    exit(1);
}
```

注意,在这个例子中,这两个进程的每一个都关闭它所不需要的管道端。这是重要的,使得写进程完全关闭该管道时,文件结束的条件被正确地传递给读进程。

阻塞读和写分别成为对空或满的管道的默认操作。这些默认操作可以用 fcntl()系统调用对管道文件描述符设置 O_NONBLOCK 标志而被忽略:

```
#include  <fcntl.h>

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

17.2 输入/输出重新定向

由于一项附加的系统调用的帮助,所有可以通过 shell 使用的输入/输出重新定向(I/O Redirection)功能都可以编入你自己的程序中 。这个附加的系统调用称为 dup()。dup()的原型是:

```
#include <unistd.h>

int dup(int fd);
```

dup()调用取一个文件描述符作为它的参数,并且返回另 一个文件描述符,它是第一个文件描述符的复制品。它表示这两个文件描述符都指向同一个打开文件描述。dup()调用从头开始搜索文件描述符数组,并且在它找到的第 一个空闲文件描述符时完成它的复制。这一概念表示在图 17.1 中,它表示对文件描述符 2 使用 dup() 系统调用的效果。文件描述符被复制到文件描述符数组的第一个空闲元素中。在这个例子中,文件描述符 0 是空闲的。它将被利用,并且 dup()调用将返回 0 值。

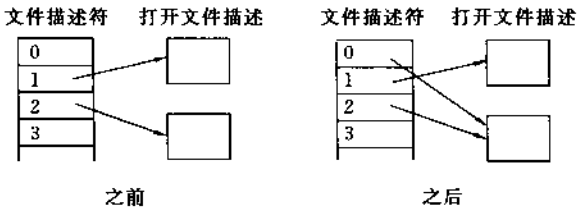

图 17.1 执行系统调用 dup(2)的效果

在进程之间的输入/输出重新定向和管道,正像 shell 所使用的,现在很容易设置。先看管道的情况,你送入管道行作为 shell 的命令,像:

```
$ ls | wc -w
```

回忆一下,ls 和 wc 命令不需要知道发生了重新定向;它们只继续使用它们的标准输入和标准输出文件来满足所有的输入和输出请求。

这表示 shell 必须创建一个管道,然后用 fork()系统调用创建两个子女进程,操作它们的文件描述符,使得管道对应于管道行的第一个进程的输出和第二个进程的输入。

下列清单是一个简单的程序,它执行前面例子中的管道行 ls | wc -w:

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

main()
{
    int fda[2];

    if (pipe(fda) == -1)                    /* 创建管道 */
        fatal("creating pipe");

    switch (fork())                         /* 创建子女进程 */
    {
    case -1:
        fatal("forking child");
        break;

    case 0:                                 /* 在子女进程中运行 ls */
        close(1);                           /* 关闭标准输出 */
        dup(fda[1]);                        /* 复制管道写端 */
        close(fda[1]);                      /* 关闭管道写端 */
        close(fda[0]);                      /* 关闭管道读端 */
        execlp("ls", "ls", 0);              /* 执行 ls 命令 */
        fatal("trying to exec ls");
```

```
        break;

    default:                                    /* 在双亲进程中运行 wc */
        close(0);                               /* 关闭标准输入 */
        dup(fda[0]);                            /* 复制管道读端 */
        close(fda[0]);                          /* 关闭管道读端 */
        close(fda[1]);                          /* 关闭管道写端 */
        execlp("wc", "wc", "-w", 0);            /* 执行 wc 命令 */
        fatal("trying to exec wc");
        break;
    }
}

fatal(char *mess)
{
    fprintf(stderr, "Error: %s\n", mess);
    exit(1);
}
```

注意,在使用 exec()调用执行 ls 和 wc 命令之前,重新定向到管道及从管道输出是如何完成的。

子女进程将被用于执行 ls 命令。在此情况下,有必要做一些准备使得当 ls 写到标准输出设备时,它将实际写到管道上。为了做到这一点,第一步用 close()调用关闭当前的标准输出设备,然后使用 dup()调用复制管道的写端。记住,dup()调用将在编号最小的空闲文件描述符中进行它的复制。在现在的情况下,它将是我们刚刚关闭的标准输出文件描述符。因为现在标准输出是和管道的写端相连,任何对这个文件描述符的 write()调用将自动地发送数据到这个管道中。

以后,该管道的原来的两个文件描述符可以被关闭。文件描述符的这种安排,将由 execlp()调用传递给 ls 命令。

类似地,双亲进程将被用于执行 wc 命令,这表示必须进行重新安排使 wc 命令从管道读取它的输入。利用类似方法来完成这一安排。当前的标准输入文件描述符(0)被关闭,利用 dup()调用将它重新分配作为管道的读端。接着,关闭该管道的原来的两个文件描述符。然后,按这种文件描述符安排使用 execlp()调用执行 wc 命令。

以后,ls 和 wc 命令都将像平常一样地执行,读和写它们的标准输入和输出文件描述符,而不知道它们正在通信。

正如下面的例子所示,标准输入和输出重新定向到文件可以以类似的方法实现。该程序用来执行下列命令:

```
$  cat  <file1 >file2
```

在现在的情况下,程序将依次关闭它的标准输入和输出文件描述符,并且使用 open()调用打开所需要的文件使它们和标准输入和输出文件描述符相连:

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

#define  WRFLAGS  (O_WRONLY | O_CREAT | O_TRUNC)
#define  MODE600  (S_IRUSR | S_IWUSR)

main()
{
    close(0);                                   /* 关闭标准输入 */

    if (open("file1", O_RDONLY) == -1)
        fatal("opening input file");

    close(1);                                   /* 关闭标准输出 */

    if (open("file2", WRFLAGS, MODE600) == -1)
        fatal("opening output file");

    execlp("cat", "cat", 0);                    /* 执行 cat 命令 */
    fatal("trying to exec cat");
}

fatal(char  *mess)
{
    fprintf(stderr, "Error: %s\n", mess);
    exit(1);
}
```

有另外两个系统调用,它们可以复制文件描述符。第一个是 dup2(),它的原型是:

```
#include  <unistd.h>

int  dup2(int  oldfd, int  newfd);
```

这个调用使得新文件描述符 newfd 成为旧文件描述符 oldfd 的复制品。在复制操作前,如果 newfd 已经被打开,它将先被关闭。

复制文件描述符的另一种方法是使用 fcntl()系统调用的另一种功能:

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

```
newfd = fcntl(oldfd, F_DUPFD, minfd);
```

它提供了介于 dup()和 dup2()之间的另一功能。fcntl()调用从指定的 minfd 开始搜索空闲文件描述符。当它找到一个时,就利用它复制旧文件描述符 oldfd。然后,复制的文件描述符由 fcntl()作为 newfd 返回。

17.3 FIFOs

匿名管道(anonymous pipe)克服了利用文件进行进程间通信的主要问题。但是,管道有一个文件所没有的缺陷,它们只可以由有共同祖先的进程所利用,它为子女进程创建了管道并且把它的文件描述符传递给它们。

先进先出(FIFO)文件(也称为命名管道)解决了这个问题,而且仍然保持匿名管道的所有优点。先进先出文件解决这个问题的方法是它们有文件名称(即目录连接),所以能够被任何进程打开和使用,只要它们对该文件有相应的访问权限,而不管它们是否有关系。

为了创建先进先出文件,可以从 shell 提示符使用 mknod 命令或者可以在程序中使用 mknod()系统调用。mknod 命令和系统调用两者除了都可以用来创建先进先出文件外还有更多的功能。当我们讨论设备驱动程序时,将看到 mknod 的其它用法。

从 shell 提示符使用 mknod 命令创建先进先出文件是容易的。命令具有如下格式:

```
$  mknod  filename  p
```

这个命令将创建一个称为 filename 的先进先出文件,它的访问权限是 666 减去 umask 的当前值。

mknod()系统调用的原型为:

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

int mknod(char *pathname, mode_t mode, dev_t dev);
```

其中 pathname 是被创建的文件名称,mode 表示将在该文件上设置的权限位和将被创建的文件类型(在此情况下为 S_IFIFO),dev 是当创建设备特殊文件时使用的一个值。因此,对于先进先出文件它的值为零。例如:

```
#define  MYMODE  (S_IFIFO | S_IRUSR | S_IWUSR)

mknod("myfifo", MYMODE, 0);
```

一旦先进先出文件已经被创建,它可以由任何具有适当权限的进程利用标准的 open()系

统调用加以访问。当用 open()调用打开时，一个先进先出文件和一个匿名管道具有同样的基本功能。即当管道是空的时候，read()调用被阻塞。当管道是满的时候，write()调用被阻塞。并且当用 fcntl()设置 O_NONBLOCK 标志时，将引起 read()和 write()调用立即返回。在它们已经被阻塞情况下，带有一个 EAGAIN 错误值。

由于它们可以被许多无关系的进程同时访问，FIFO 在有多个读进程和/或多个写进程的应用中是有用的。在这种情况下，需要有一个规则来决定如果两个写进程企图同时写这个 FIFO 该如何处理。这个规则是：一个 write()调用可以写管道能容纳(Linux 为 4096 字节)的任意个字节，这些将被保证是分开的。这表示多个写操作的数据在 FIFO 文件中并不混合而将被维持成分离的信息。同样的规则完全适用于匿名管道，但是它们较少用于这样类型的应用中。

17.4 system V 的进程间通信机制

为了提供与其它系统的兼容性，Linux 也支持三种 system V 的进程间通信机制：共享内存，消息和信号量(semaphores)，这是可供选择的项目。这些选项中的每一项都具有同样类型的接口，因此，我们将只简短地讨论其中的一项——共享内存。

管道和 FIFO 的使用仍然有一个缺点，这就是复制数据的数量，以及在进程传递数据过程中的环境切换。

共享内存克服了这个问题，它在两个或多个进程都能使用的地址空间中使用同一块内存。这表示只要一个进程已经把某些信息写入这一内存块，就立刻可供其它共享这一内存块的进程所使用，而不用再进行任何进一步的复制操作。

在任何进程可以使用共享内存段之前，首先需要创建它。这由 shmget()调用完成，它的原型如下：

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

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

其中 size 参数是所要求容量的字节数(四舍五入到内存页的最小整数)；key 参数是一个数值(对共享内存段来说，它具有类似于文件名的作用)；flags 参数是位的掩码，它用来规定访问权限以及共享内存段被创建时的创建标志，这些用逻辑加(OR)加以组合。(这有点像 open()系统调用的第 2 和第 3 个参数的组合)。

如果在 flags 参数中包含 IPC_CREAT 标志，当共享内存段不存在时，将创建和 key 参数结合的共享内存段。如果 flags 参数中包含 IPC_EXCL 标志，当与指定的 key 参数结合的共享内存段已经存在时，将导致 shmget()调用失败。

如果在 key 参数中使用 IPC_PRIVATE 标志，在创建共享内存段时，就不必担心该段是否已经存在。

从 shmget()返回的值是共享内存段的识别号。在概念上它类似于从 open()返回的文件描述符，这个值需要传递给和该内存段有关的其它调用。但是，主要的差别在于任何具有共享内存段识别号的进程都可以访问该内存段，不像使用文件描述符，只能由后裔进程共享。

下面是创建共享内存段的样本程序：

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

#define  SHM_SIZE  4096
#define  SHM_FLAGS  IPC_CREAT | 0644

main()
{
    int shmid;

    shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_FLAGS);

    if (shmid == -1)
        fatal("creating shared memory segment");

    printf("shared memory segment ID = %d\n", shm_id);
}

fatal(char *mess)
{
    fprintf(stderr, "Error: %s\n", mess);
    exit(1);
}
```

任何 system V 的进程间通信资源的状态一旦被建立，就可以用 ipcs 命令进行检查。命令的 -m 开关给出关于共享内存段的信息，-s 开关列出信号量数组，而 -q 开关则是列出消息队列。如果没有指定开关，则将显示所有三种资源列表。

如果前面所列的程序称为 make_shm，则它可以被使用，并且它的结果可以用下列命令序列考察：

```
$  make_shm
shared memory segment ID(共享内存段识别号) = 1408
$ ipcs

------ Shared Memory Segments(共享内存段) --------
shmid      owner      perms      bytes      nattch     status
1408       pc         644        4096       0

------ Semaphore Arrays(信号量数组) --------
semid      owner      perms      nsems      status
```

```
------ Message queues(消息队列) - - - - - - - -
msqid      owner      perms      used-bytes messages
```

一旦共享内存段已经存在,它可以用 shmat()调用附加到一个运行进程的内存。shmat()原型是:

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

char *shmat(int shmid, char * shmaddr, int flags);
```

其中 shmid 是由 shmget()调用返回的共享内存段识别号,shmaddr 是你希望共享内存段附加的地址,flags 允许你规定希望所附的段为只读(利用 SHM_RDONLY 标志)以代替读写。通常,并不要规定你自己的 shmaddr,可以用传递参数值零使得系统为你取一个地址。

shmat()返回值是一个指向共享内存段的指针(C 类型指针),然后,可以就像指向任何内存块的指针一样地使用它。

当共享内存段使用完毕时,你可以用 shmdt()调用使它和你的进程分开:

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

int shmdt (char *shmaddr);
```

将由相应的 shmat()调用返回的内存地址作为 shmaddr 参数。

即使没有其它进程和共享内存段结合在一起,shmdt()调用并不取消它。事实上,任何写到该段的内容将仍然保持不变。所以,容许随后的 shmat()调用将仍能访问该内存段和它所含的数据。

如果你希望实际取消一个共享内存段(当真正使用完毕时,应该这样做),这可以用一个称为 shmctl()的控制函数来完成:

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

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

从概念上讲,这个函数类似于 ioctl()和 fcntl()。它可以对由 shmid 指定的有关共享内存段的信息结构进行读和写访问。参数 cmd 规定实际执行的操作,buf 是指向共享内存结构的指针,它用来传递所要求的数据。下列 cmd 值可供使用:

IPC_STAT　　拷贝共享内存结构到 buf;

IPC_SET　　从 buf 设置共享内存结构;

IPC_RMID　　取消由 shmid 给定的共享内存段。

取消共享内存段的另一种方法是使用 ipcrm 命令:

```
$ ipcrm shm shmid
```

其中 shmid 是在 ipcs 命令的显示中给出的共享内存段识别号。例如:

```
$ ipcrm shm 1408
```

下面是一个简短的程序,它说明如何把识别号为 1408 的内存段连接到进程,写一些数据到该段,然后将它和进程分开,等待某个其它进程读:

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

#define  SHM_ID  1408

main()
{
    char *shmaddr;

    if ((shmaddr = shmat(SHM ID, 0, 0)) == (char *) -1)
        fatal("attaching shared memory segment");

    strcpy(shmaddr, "Some test message");

    if (shmdt(shmaddr) == -1)
        fatal("detaching shared memory segment");
}

fatal(char *mess)
{
    fprintf(stderr, "Error: %s\n", mess);
    exit(1);
}
```

注意,为了与系统调用统一,在错误时 shm 函数全都返回 -1 值。因为 shmat()实际上返回指向内存块的指针,为了进行比较有必要把 -1 值转换(cast)成 char *。

虽然它们操作在不同的数据结构上，其余的 system V 进程间通信机制(信号量和消息)在概念和用法上和共享内存非常相似，它们每一个都首先用一个调用来创建资源；它们也要调用访问和控制操作：

	共享内存	信号量	消息
创建	shmget()	semget()	msgget()
访问	shmat()	semop()	msgsnd()
	shmdt()		msgrcv()
控制	shmctl()	semctl()	msgctl()

17.5 Sockets

到此为止，所有讨论过的进程间通信机制都仅仅适用于在同一台机器上的进程之间进行通信。随着现代向网络发展的趋势，Linux 支持完全网络兼容的进程间通信机制是很重要的。Sockets 就是这样的机制。

Sockets 第一次出现在 BSD UNIX 的早期版本中。设计者试图建立非常通用的机制，它能够适应许多网络协议。在很大程度上，设计者们是成功的，但是，为了取得所有必要的灵活性，用 Sockets 进行程序设计需要对若干可变的参数值的详细说明。为了使某些参数含义更为明确，有必要首先讨论基本的网络原理。

互连网协议(Internet Protocol，简记为 IP)建立在较低层的软件协议和网络硬件的基础上，并从应用软件中掩盖掉这些网络结构的特点。这表示应用层不需要知道它们正在使用何种网络硬件(以太网，串行线路还是并行线路)。

IP 被称为无连接服务，这意味着网络在特定的机器之间没有任何固定的连接。只不过是在概念上大量的机器全都连接在一条公共的高速公路上。为了使信息投递路径的选择(routing)成为可能，网络上每台主机的每个接口被给予一个唯一的 IP 地址。每个地址正好是一个 32 位数，通常用 4 个 8 位值给出，以十进制数加句点的方式印出(例如 194.61.21.6)。如果你仅仅使用一台单独的机器，或者在你自己的本地网上，则你可以自由选择你自己的 IP 地址。但是，如果你将把你的机器连接到具有全球 Internet 访问的网上，你将需要由你的网络管理员分配给你一个全球注册的 Internet 地址，。在网络上传输的每个 IP 数据包(称为数据报(datagram))带有它的源 IP 地址和目的地 IP 地址信息以及数据自身，并且可能经过若干台机器才到达它的目的地。由于每个 IP 数据报被分别地传送，它们可能经过不同的路线，无连接服务不保证这些数据包将以它们发送时的同样顺序到达目的地，或者甚至根本不保证它们将最终到达目的地。

17.5.1 用户数据报协议

在 IP 层，数据报中的地址仅仅规定机器接口的 IP 号；它们不规定在这些机器上的任何特定的进程或应用。为了克服这个问题，在基本的 IP 层的上面增加了用户数据报协议(User Datagram Protocol，简记为 UDP)。它提供一套称为端口(ports)的通信端点(communication end point)，每个端口由一个小的整数值所标识。UDP 信息包含有源和目的地端口号，它们被目的地的 UDP 软件用来将进来的信息投递给正确的进程，并且使那个进程能够返回一个答复给原

始的发送者。

利用 UDP,每个数据包是一个独立自足的实体。这意味着每个数据包也传送所有投递它所必需的信息,使得从特定的进程通过给定的 UDP 端口传输的数据包序列不需要全都有同样的目的地。

UDP 简单地使用底层的 Internet 协议发送它的信息,所以它也像 IP 一样提供不可靠的无连接的信息传递。这里所说的不可靠并不表示信息不可能到达,而只不过是说它不保证信息一定到达。

17.5.2 传输控制协议

在应用程序层,在两个确定的进程之间经常需要传递大量的数据。在这种情景下,使用不可靠的无连接的协议意味着应用程序将不得不为其建立复杂的错误检测和纠正机制。为了在设计和建立应用程序时避免不必要的开销,要求使用可靠的协议,它保证在两个确定的进程之间数据的传递,同时也保证将以发送数据的同样的顺序接收到它们。传输控制协议(Transmission Control Protocol,简记为 TCP)是标准的解决方案。

像 UDP 一样,TCP 也在基本的 IP 层的基础上工作,并且提供协议端口,允许在单台机器上运行多目的地的进程。但是不像 UDP,TCP 是一个基于连接的协议。这表示在可以交换任何数据之前,通信链路两端的进程必须同意参与。一旦建立了连接,它们之间可以交换任意大小的数据,然后结束通信。协议本身无需任何种类的信息边界。因此,例如利用 read()和 write()系统调用由发送者传送 100 字节的信息,可以合理地在目的地被作为 4 个分开的 25 字节的信息被接收。这表示需要在连接两端的进程中对传递的数据增加所需的结构,不一定由 TCP 保存。

TCP 和 IP 是作为整体设计的,在实现上也是互相配合和互相补充的。TCP 解决 IP 没有解决的问题,而不去重复 IP 的工作。IP 提供了将数据包从源传递到目的地的方法,TCP 则解决了如数据包丢失和传递顺序混乱等问题。两者的结合,提供了可靠的数据传输方法。

Sockets 是一个通信端点(communication end point),并且可以被相当好地映射到一个 UDP 或 TCP 端口。虽然这些不是唯一可能的映射,但它们是到目前为止最普通使用的,也是这里最详细考虑的。

17.5.3 客户程序和服务程序

一个服务程序是一个进程,它给其它进程提供某种服务,客户程序是服务程序的顾客进程。在客户程序和服务程序利用 TCP 通信的情况下,服务程序通常将打开它的连接端并且被动地等待连接到来。另一方面,客户程序在打开到服务程序的连接中将扮演主动的角色。

一旦在两个进程之间的 TCP-Sockets 连接已经建立,连接的端点将被作为像普通的 Linux 文件一样对待,使得像 read()和 write()标准系统调用可以用来传递数据。为了使这一点成为可能,类似于普通文件有文件描述符和它结合在一起一样,Sockets 也有描述符和它们结合在一起。Sockets 描述符和文件描述符之间的主要差别是,当 Sockets 被创建时,Sockets 描述符并不自动地与规定的目的地地址联系在一起,而文件描述符则与由 open()系统调用所规定的文件名称结合在一起。为了使 read()和 write()系统调用能被用于这两种情况,Linux 有必要确保 Sockets 和文件描述符是在同样的数值范围内,并且在一个给定的应用中没有重复的数值。这由从同一进程表分配它们俩来做到。

17.5.4 创建一个 Sockets

Sockets 是利用 socket()系统调用来创建的。socket()调用取三个参数:地址类(family),类型(type)和协议(protocol),并且返回一个整型 Sockets 描述符作为它的返回值。该调用的一般形式是:

```
sd = socket(family, type, protocol)
```

地址类(family)参数指定该 Sockets 使用的地址类。地址类规定给定的地址的格式。两个主要的类是 AF_INET,它们的地址被给定为 32 位的 IP 地址(标准 Internet 格式)和 AF_UNIX1,其中地址是在 Linux 文件系统中的路径名。

类型(type)参数规定所请求的通信类型。主要的选择有两类:被称为 SOCK_DGRAM 的非连接数据报(datagram)服务(UDP 类型)和被称为 SOCK_ STREAM 的可靠的基于连接的服务(TCP 类型)。如果仅从简化和可靠的观点出发,大部分本地应用将利用 SOCK_STREAM 来写。

在大多数情况下,规定地址类和 Socket 类型足以决定在新的 Socket 上所使用的通信协议。在这些情况下,协议(protocol)参数被给予零值,使得默认的协议将自动地被选取。非零协议值通常仅用于直接访问低层硬件接口。

如果 socket()调用成功,返回值是一个整型 Sockets 描述符。在错误时返回 -1 值。在建立 Sockets 连接时,Sockets 描述符被用作其它调用的一个参数。

在对话过程(session)结束时,以该 Sockets 描述符为参数,使用标准的 close()系统调用来关闭它。

17.5.5 连接本地地址

当 Sockets 被初次创建时,它没有与规定的地址相连,所以远程进程无法引用它。对于特定的 Sockets 的地址格式将依赖于该 Sockets 被创建时所规定的地址类。例如,在 Internet 地址类(AF_INET)的情况下,Sockets 地址应当是一台机器的接口的 IP 地址和那台机器的一个端口号。

通常,客户进程并不在乎赋予它的 Sockets 什么端口号,并且将允许系统为它挑选。另一方面,服务程序的进程需要对系统规定它们的地址,因为它们通常将运行在一个固定的端口号上,该端口号对它们的客户进程是已知的,并且客户进程将需要使用这个端口号来建立连接。

为了把一个规定的地址和一个 Sockets 结合,一个服务程序的进程使用如下的 bind()系统调用:

```
bind(sd, address, addrlen)
```

sd 参数正好是由 socket()调用返回的 Sockets 描述符。address 参数是指向一个结构的指针,该结构包含与 Socket 结合的相应类型的地址,而 addrlen 参数规定地址的字节长度。为了允许在不同的地址类之间可能使用各种各样地址格式,这样安排是必要的。

17.5.6 客户进程连接到服务进程

在基于连接的(TCP 类型)通信链路中,一个客户进程需要对服务进程的连接进行初始化。

为了做到这一点,客户进程需要知道服务器将用来为它提供服务的地址和端口号。客户进程用这些信息调用 connect()系统调用:

```
connect(sd, address, addrlen)
```

sd 参数是客户的本地 Sockets 的 Sockets 描述符。address 参数规定在目的机器上服务进程正在使用的地址,addrlen 是规定的地址的字节长度。

当作 connect()调用时,如果客户的 Sockets 还没有结合到一个规定的本地地址,则系统将自动地挑选一个本地地址和一个适当的本地端口号,并且把它们和该客户的 Sockets 相结合。

如果一个连接被成功地建立,connect()调用返回 0 值,或者在错误时返回 - 1 值。一旦客户已成功地建立了一个连接。该 Sockets 可以用于发送和接收数据。这里存在几种可能性,最普通的情况是使用 read()和 write()系统调用。这些是和在文件中所使用的完全同样的调用,但是本地 Sockets 描述符被作为它们的第一个参数。

17.5.7 设置服务程序

一旦服务进程已经用 socket()和 bind()系统调用来创建它自己的一个 Socket 并且把一个本地地址和端口和该 Socket 相结合,则它需要设置 Socket 使它能接受进来的连接请求。listen()系统调用被用来执行这项功能。同时,这个系统调用也用于设置一个队列的大小,Linux 将使用该队列将来自多个客户的连接请求进行排队。listen()系统调用的一般形式是:

```
listen(sd, qlen)
```

其中 sd 参数是由 socket()调用返回的 Sockets 描述符,qlen 是许可的队列大小,被用来决定客户连接请求的最大个数。

一旦 Socket 已经以这种方式设置,现在服务器仅需要等待一个连接。为了做这一点,它使用如下的 accept()系统调用:

```
newsd = accept(sd, address, addrlen)
```

其中 sd 参数是和服务器 Socket 相连的 Sockets 描述符。address 参数是指向一个 Socket 地址结构的指针,它将由 accept()调用填入。当建立连接时,和客户的 Socket 结合的地址和端口号将被填入这一地址结构。addrlen 参数是指向整数的指针,它也由 accept()调用填入,它给出客户的地址长度的字节数。

从 accept()的返回值是一个新的 Sockets 描述符,它具有所建立连接的客户作为它的目的地,并且它将被用于今后和那个客户的通信。原来作为 accept()一个参数的 Sockets 描述符在这一点仍是打开的,并且可以被再次用于对 accept()另一次调用来建立与另一个客户进程的连接。

一旦 Sockets 描述符已由 accept()返回,服务器可以使用通常的 read()和 write()系统调用与新的客户通信。

17.5.8 阻塞和非阻塞 Sockets

默认地,如果无客户进程正等待建立连接,则对 accept()的调用将被阻塞。这意味着 accept()调用不返回,直到有一个客户准备好通信。因而,服务器进程不能处理任何其它任务,直到建立连接为止。

有时,要求服务器仅注意检查是否有客户正在等待连接,如果有就接受连接;否则如果没有挂起的连接,它就继续做其他事情。这由非阻塞 Socket 来完成。即使没有客户在等待,使得 accept()调用立即返回。可以使用 fcntl() 系统调用对有关的 Socket 描述符设置或清除 O_NDELAY 标志来得到这个结果。首先获得与 Sockets 描述符相结合的标志:

```
flags = fcntl(sd, F_GETFL);
```

然后置 O_NDELAY 如下:

```
fcntl(sd, F_SETFL, flags | O_NDELAY);
```

或者清除 O_NDELAY 如下:

```
fcntl(sd, F_SETFL, flags&~O_NDELAY);
```

17.5.9 非连接通信

除了基于连接的通信之外,也支持基于 UDP 数据报 Socket 的无连接方案。除了在 socket()调用中用类型参数 SOCK_DGRAM 代替 SOCK_STREAM 之外,这些 Socket 和基于流(stream)的 Socket 以同样方式被建立。一旦建立了数据报 Socket,如果一个特定本地地址需要利用它,可以用 bind()系统调用将它和该 Socket 结合。如果使用 bind(),则必须在使用 Socket 传输数据之前进行。如果不使用 bind(),则在首次发送数据时系统将自动地分配一个本地地址和端口。

进程应该使用 sendto()和 recvfrom()系统调用发送和接收数据,因为在它们的参数列表中包含一个参数用来存放通信伙伴的地址和端口号。sendto()调用的一般形式是:

```
sendto(sd, buf, len, flags, addr, addrlen)
```

其中 sd,buf 和 len 与 write()系统调用中的参数具有同样的类型和作用。flags 参数用于若干特殊的信息选项,它们的值在头文件 < linux/socket.h > 中定义。addr 参数是指向包含目的地址和端口的结构的一个指针,addrlen 是地址结构的字节长度。

recvfrom()系统调用具有下列一般形式:

```
recvfrom(sd, buf, len, flags, addr, lenptr)
```

它的参数列表类似于 sendto()的参数列表,参数 sd、buf 和 len 像 read()系统调用中的参数一样。参数 addr 和 lenptr 分别是指向一个地址结构和一个整数的指针,并且由 recvfrom()调用

填入有关通信伙伴的相应信息。

数据报 Socket 也可以使用 connect()系统调用。在此情况下,并不试图与另一个进程建立一个固定的连接;connect()调用只是用来规定一个数据报 Socket 和一个特定的目的地址和端口之间的联系。一旦使用了 connect(),任何在该 Socket 上所发送的数据被自动地选择路由到达目的地。这允许使用 send()和 recv()系统调用来传递数据。这些调用的一般形式是:

```
send(sd, buf, len, flags)
recv(sd, buf, len, flags)
```

这两个调用完全和 sendto()及 recvfrom()相同,但是没有最后两个参数,因为这些信息已经由 connect()设置好了。

send()和 recv()调用也可以用在基于连接的 Sockets 上通信,因为进程和它的通信伙伴已经建立了固定的连接。

17.5.10 Sockets 支持的调用

除了你现在已经见到的和 Sockets 使用有关的主要系统调用集合之外,还需要提及若干支持函数。这些函数和主机名、端口号、标准网络服务及网络的整数字节顺序有关。

gethostname()系统调用允许进程访问本地主机名。其形式是:

```
gethostname(name, len)
```

其中 name 是指向存放主机域名的一个字符数组的指针,len 是该数组可能存放名称的最长长度。对于特权进程(EUID = 0),sethostname()可以用于将本地主机名设置为规定的字符串。

一旦你有一个主机域名可供使用,你需要能够得到它的 IP 地址或者相反。库函数 gethostbyname()和 gethostbyaddr()可以执行这些任务。这些调用的一般形式是:

```
hostent = gethostbyname(name)
hostent = gethostbyaddr(addr, len, type)
```

参数 name 是指向含有主机域名的一个字符数组的指针。参数 addr 是指向一个长度为 len,地址类型为 type 的主机地址的指针。两个调用都返回指向在 < netdb.h > 中定义的 hostent 结构的指针,hostent 包含所要求的主机的名称、地址和地址类型信息。

对于所有的标准网络服务有一组已知的端口。为了找到与特定的服务名和协议有关的端口。有一个标准库函数称为 getservbyname()。它具有下列一般形式:

```
servent = getservbyname(name, protocol)
```

参数 name 是指向含有所请求的服务名称的字符串指针,参数 protocol 是指向含有相应通信协议(udp 或 tcp)的字符串指针。这个函数在/etc/services 文件中搜寻所要求的信息并返回

指向 servent 结构的指针。servent 结构在 < netdb.h > 中被说明。

具有不同的处理机体系结构的机器以若干不同的方式存储整值。联网所涉及的最重要的差别是在整数中字节存储的顺序。某些处理机,包括 80x86 系列在一个整数中首先存储低字节。这意味着在这些机器上指向一个整数的指针实际上指向在多字节值中最小有效数字字节。其它处理机在一个整数中首先存储高字节,使得在此情况下,一个整数指针指向在多字节值中最高有效数字字节。

如果不采取步骤避免这个问题,则不同类型的机器在网络上彼此通信整型值将有麻烦。这个问题的解是所有的机器将以一种方式彼此发送多字节值。事实上,一致的标准是首先发送高字节。这意味着首先存储低字节的机器(包括 80x86 和 Linux)每当从网络输入整数或把整数输出到网络时都需要颠倒它们的字节顺序。为了进行这一转换,提供了 4 个函数,两个供 16 位值(short)的输入和输出,两个供 32 位值(long)的输入和输出。这些函数是:

```
netval = htonl(hostval) - 主机到网络(long);
netval = htons(hostval) - 主机到网络(short);
hostval = ntohl(netval) - 网络到主机(long);
hostval = ntohs(netval) - 网络到主机(short)。
```

所有的机器都提供这些函数。在已经使用正确的字节顺序的机器上,它们只是一些伪(dummy)函数或不做任何事的宏。但是,它们仍然被提供,使得在一个机器上写的代码可以移植到字节顺序颠倒的处理机体系结构的机器上。

显然,如果你仅仅想传输字节流(例如字符串)则不需要使用这些函数。

这些函数和有关的调用的更多信息在手册页的第 2 和第 3 节中提供。

练习

1. 写一个称为 mydup2()的函数,它提供 dup2()系统调用的功能,但是仅仅利用 dup()写出。
2. 设计和运行一个试验程序来发现 Linux 中一个管道的容量。
3. 写一个简单的程序,它创建一个管道,然后,执行一个 fork()调用,使得两个独立的进程在运行。然后,这两个进程应当使用这个管道把一个大文件的内容从一个进程传递给另一个进程。
4. 重写前面问题中的程序,在两个进程之间使用共享内存传递文件内容。在使用这种方法时,你会遇到什么样的额外问题? 你认为哪一个程序会运行得较快,实际哪一个程序运行得较快?

解答

1. 为了解决这个问题,在你到达你想要的文件描述符之前,你将需要追踪你用于服务的那些文件描述符,使得你以后可以再次释放它们。基本的函数如下:

```
#include < linux/limits.h >

int mydup2(int oldfd, int newfd)
{
    int i, fd, fdtable[OPEN_MAX];

    if (newfd < 0 || newfd > OPEN_MAX)
```

```
        return -1;

    if (oldfd < 0 || oldfd > OPEN_MAX)
        return -1;

    if (newfd == oldfd)
        return newfd;

    for (i = 0; i < OPEN_MAX; ++i)
        fdtable[i] = 0;

    close(newfd);

    while ((fd = dup(oldfd)) != newfd && fd != -1)
        fdtable[fd] = 1;

    for (i = 0; i < OPEN_MAX; ++i)
        if (fdtable[i])
            close(i);
    return fd;
}
```

2. 这只要建立一个管道，设置它的 O_NONBLOCK 标志，然后计数并把字节写到它里面，直到 write()调用的返回值指示字节不再被存储到该管道中为止。检查计数，一个管道的容量应该是 4K。
3. 执行这个任务的一个伪码程序可能是：

```
create pipe
if (fork())
    In parent, open() file
    while not end of file
        read a character and write it to the pipe
else
    In child
    while not end of pipe data
        read a character from pipe and discard
```

你应该运行这个程序若干次，以确保你正在复制的文件全都存储在高速缓冲区(cache)中——从内存中读它比从磁盘中读快得多。

4. 除了你将创建共享内存段以代替管道，在 fork()调用之后，在双亲和子女进程内使用 shmat()之外，这个程序将类似于前一个程序，使用共享内存遇到的额外问题是在两个进程之间你将需要某种形式同步，使得读进程进程在读到传输的字符之前写进程进程将不覆盖它，以及相反。做这件事的明显方法是使用信号量(semaphore)，你可以自己用共享内存实现，或者使用所提供的信号量 IPC 机制。我留给你自己做实验去解决这个问题。哪一种解决方法比较快，是管道还是共享内存。

第十八章　极小的 socket 库——实例研究

鉴于我们已经讨论了许多使用 socket 设置网络通信的理论，现在是将它用于实践的时候了。在大多数情况下，使用可靠的基于流(stream)的客户/服务器协议来编写 socket 程序。这表示每当程序员第一次写 socket 应用时，将包含通过搜寻手册页来找出所有有关的系统调用，然后试图理解每个调用可供使用的所有可选项，仅取得本质上和别人相同的解决方案。虽然，这可能有一点过分一般化，但令人惊奇的是许多程序员显然完全在重复 socket 通信问题的解决方案。

18.1　库函数

所要的是某种简单的函数库(library)，它将提供标准可靠的基于流的客户程序/服务程序的通信功能。仅仅当你愿意在通常范围以外做某些事情时，你才需要深入钻研手册页中的可选项。这有一点像建立文件的标准 I/O 库函数(fopen()，fclose()等)的思路。

最简单的安排是：如果你要设置一个服务程序，只要调用一个函数；如果你要设置客户程序就调用另一个函数。另外两个调用将用来创建和取消服务程序和客户程序赖以通信的 socket。除此之外，你也愿意指定尽可能少的参数，尽可能多地选取实用的默认值。

下列程序是一个极小的 socket 库，可以将它包含在你自己的程序中，它提供的就是这一功能。随后的简单应用用来演示该库的使用。

这个库本身由两个文件组成：一个源代码文件，你可以编译它并连接到你的应用中；一个头文件，你可以用 #include 语句将它包含在你的代码中。它说明某些符号常量、一个数据结构和库函数本身的原型。

库中的函数是 sopen()，sclose()，sserver() 和 sclient()。本质上，一个服务进程将调用 sopen()函数，该函数创建一个 socket，并且存储该 socket 描述符以及一些管理信息到一个结构中。这个函数返回指向该结构的指针作为它的返回值。然后传递 socket 结构的指针以及服务程序将用来提供与客户程序连接的端口号作为 sserver()调用参数。当一个客户程序和服务程序连接时，sserver()调用返回一个 socket 描述符。然后，在诸如 read()和 write()调用中利用该 socket 描述符来和客户程序进行通信。可以用同样的 socket 结构指针和端口号进一步调用 sserver()完成对其它客户程序的连接。当客户程序通信结束时，由 sserver()返回的 socket 描述符应该用 close()释放。当服务程序关闭时，由 sopen()返回的 socket 指针应该用 sclose()释放。

在客户程序端，对 sopen()的调用将同前面一样创建一个 socket，并且返回一个 socket 指针。然后这个指针和服务程序的地址及端口号可以被传递给 sclient()。当连接被建立时，sclient()返回一个和服务程序通信的 socket 描述符。如同在服务程序情况一样，当客户程序结束它的任务时，应该使用 close()和 sclose()。

18.1.1 sopen() 函数

```
/* 1 */
SOCKET *sopen(void)
{

    /* 2 */
    SOCKET *sp;

    /* 3 */
    if ((sp = (SOCKET *)malloc(sizeof (SOCKET)) ) == 0)
        return 0;

    /* 4 */
    if ((sp -> sd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        free(sp);
        return 0;
    }

    /* 5 */
    sp -> sinlen = sizeof(sp -> sin);
    sp -> bindflag = S_RESET;
    /* 6 */
    return sp;
}
```

依次对编号的注释说明如下：

1. sopen()函数被调用来创建一个 socket。socket 描述符以及一些内部管理信息被存储在一个 SOCKET 结构中(它定义在 socklib.h 中)。如果调用成功，由 sopen()返回指向该 SOCKET 结构的指针(出错时返回 0)。这个指针应该被保存，并传递给其它 socket 库函数。SOCKET 结构定义如下：

```
typedef  struct
{
    struct  sockaddr_in  sin;
    int  sinlen;
    int  bindflag;
    int  sd;
} SOCKET;
```

其中 sd 是当 socket 被创建时所返回的 socket 描述符，sin 和 sinlen 是服务程序 sock-

et 的地址和长度信息,bindflag 标志被用于服务程序,确保 bind()系统调用在服务程序的 socket 上仅使用一次。

2. 说明指向 SOCKET 的指针,它的值最终将成为从 sopen()的返回值。
3. 对 malloc()的调用为了得到一块足够大的内存来存储 SOCKET 结构。如果 malloc()失败,则 sopen() 返回 0 值。
4. 这段代码在 IP 地址类(AF_INET)中创建一个 TCP socket (SOCK_STREAM)。如果没有错误,由 socket()调用所返回的 socket 描述符被存储在前面 malloc()调用所得到的结构的 sd 域中。否则,所分配的内存用 free()释放,并且返回一个错误值 0。
5. 一旦成功地创建了一个 socket,所分配的结构的 sinlen 和 bindflag 域被初始化。
6. 最后由 sopen()返回指向经过初始化的 SOCKET 结构的指针将被传递给其它的 socket 库调用。

18.1.2 sclose()函数

```
/*1 */
sclose(SOCKET *sp)
{
    int sd;

    /* 2 */
    sd = sp->sd;
    free(sp);

    /* 3 */
    return close(sd);
}
```

1. 函数 sclose()关闭和给定的 SOCKET 指针相联的 socket。该指针作为参数被传递给函数。如果调用成功返回 0 值,否则,在错误时返回 -1 值。
2. 在 socket 指针结构的 sd 域中的 socket 描述符被保存在一个局部变量中,使得由 sp 所指向的内存块可以用 free()安全地释放。
3. 被保存的 socket 描述符现在可以用 close()系统调用关闭。这个系统调用在成功时返回 0,在错误时返回 -1,完全和 sclose()返回值一样。所以这些全可以在一行中完成。

18.1.3 sserver()函数

```
/* 1 */
sserver(SOCKET *sp, int port, int sync)
SOCKET *sp;
int port, sync;
{
```

```
int flags;
struct hostent *hostent;
char localhost[S_NAMLEN + 1];

/* 2 */
if (sp - >bindflag == S_RESET)
{
   /* 3 */
    if (gethostname (localhost, S_NAMLEN) == - 1
    | | (hostent = gethostbyname(localhost)) == 0)
        return - 1;

    /* 4 */
    sp -> sin. sin_family = (short)hostent -> h_addrtype ;
    sp -> sin.sin_port = htons((unsigned short)port);
    sp -> sin. sin_addr. s_addr = *(unsigned long *)hostent -> h_addr;

    /* 5 */
    if (bind(sp -> sd, (struct sockaddr *)&sp -> sin, sp -> sinlen) == - 1
    | | listen(sp -> sd, 5) == - 1)
        return - 1;

    /* 6 */
    sp -> bindflag = S_SET;
}

/* 7 */
switch (sync)
{
case S_DELAY:
    if ((flags = fcntl(sp -> sd, F_GETFL)) == - 1
    | | fcntl(sp -> sd, F_SETFL, flags&~O_NDELAY) == - 1)
        return - 1;
    break;

case S_NDELAY:
    if ((flags = fcntl(sp -> sd, F_GETFL)) == - 1
    | | fcntl(sp -> sd, F_SETFL, flags|O_NDELAY) == - 1)
        return - 1;

    break ;

default:
    return - 1;
```

```
    }

    /* 8 */
    return accept(sp->sd, (struct sockaddr *)&sp->sin, &sp->sinlen);
}
```

1. sserver()函数将调用的进程设置为一个网络服务程序,并且建立与客户程序的连接。参数 sp 是一个由前面的 sopen()返回的 SOCKET 指针。参数 port 是端口号,它应该把本地主机地址和打开的 socket 结合。参数 sync 规定:如果没有客户程序正在等待建立连接,是否对 sserver()的调用进行阻塞。sync 的两个可能值是 S_DELAY 和 S_NDELAY,它们定义在 socklib.h 中。如果出现错误,或者如果规定了 S_NDELAY 并且没有客户程序正在等待接受连接,对 sserver()的调用将返回 -1 值。在后一种情况下,说明为外部的整型变量 errno 将被设置 EWOULDBLOCK 值。如果建立了与客户程序的连接,则从 sserver()返回的值将是一个 socket 描述符,它可以用在 read()、write()中与新的客户程序进行通信。
2. 下一个任务是把一个地址和 socket 结合在一起,然后设置一个队列用来监听连接请求。但是,这应该仅仅在第一次调用 sserver()时完成。为了确保做到这一点,一个包含在指针 sp 所指的 SOCKET 结构中的 bindflag 标志被测试,仅当它是复位(reset)时,if 结构体内的 bind()和 listen()调用将被执行。
3. 利用 gethostname()函数,本地机器的主机名被复制到 localhost[ ]数组中。然后被传递给 gethostbyname(),它返回指向填入相应信息的 hostent 结构的指针。如果在这些函数调用中出现错误,则返回 -1。
4. 现在将参数 port 和从 hostent 结构中取得的地址类、端口号和 IP 地址复制到 socket 地址结构中,它被存储在由参数 sp 所指向的 SOCKET 中。
5. 当设置好前面各项时,调用 bind()和 listen()。如果这些调用中哪一个产生了错误,则返回 -1。
6. 为了保证前面的代码段对每个 socket 仅执行一次,现在设置 bindflag 标志。
7. 下一个任务是区分参数 sync。如果 sync 设为 S_DELAY 值,则 fcntl()被用来复位(reset)在有关的 socket 描述符中的 O_NDELAY 标志。如果 sync 设为 S_NDELAY 值,用来设置有关的 O_NDELAY 标志。如果 sync 设为任何其它值,将执行 default;并且返回 -1 作为错误值。
8. 最后,调用 accept()建立与任何正在等待的客户程序的连接。成功时,accept()返回一个连接该客户程序的 socket 描述符并且准备好运行。这个值也由 sserver()返回。如果没有客户程序等待和 socket 被设置为阻塞,accept()调用将等待直到一个客户程序提出连接请求时才返回,如果没有客户程序等待和 socket 是非阻塞的,则 accept()将返回 -1,并且将 errno 置为 EWOULDBLOCK 值。这个值也由 sserver()返回。

18.1.4 sclient() 函数

```
/* 1 */
sclient(SOCKET *sp, char *name, int port)
{
    struct hostent *hostent;

    /* 2 */
    if ((hostent = gethostbyname(name)) == 0)
        return -1;

    /* 3 */
    sp->sin.sin_family = (short)hostent->h_addrtype;
    sp->sin.sin_port = htons((unsigned short)port);
    sp->sin.sin_addr.s_addr = *(unsigned long *)hostent->h_addr;

    /* 4 */
    if (connect(sp->sd, (struct sockaddr *)$sp->sin, sp->sinlen) == -1)
        return -1;

    /* 5 */
    return sp->sd;
}
```

1. sclient()函数试图连接在给定的机器上的指定的服务程序。它的三个参数分别是:sp,它是由 sopen()返回的 socket 指针;name,它是服务程序的机器名;和 port,它是服务程序在机器上使用的端口号。在建立和服务程序的连接之前函数将等待,然后返回连接服务程序的 socket 描述符,或者在错误时返回 -1。
2. 在给定服务程序的主机名时,gethostbyname()函数用来填写 hostent 结构。
3. 从 hostent 结构和 port 参数填写与 SOCKET 相联的 socket 地址结构。
4. 现在用 connect()系统调用把本地地址和本地 socket 结合,并且试图与所选择的服务程序联系。如果服务程序已经进行了 listen()调用,则客户程序的 connect()请求将在服务程序上排队等待它的 accept()调用。
5. 一旦 connect()调用无错误地返回和建立了连接,sclient()函数返回和客户程序 socket 结合的 socket 描述符。

18.1.5 库 Preamble

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

```
#include   <sys/socket.h>
#include   <netdb.h>
#include   <netinet/in.h>

#define S_LIBRARY
#include "socklib.h"
```

这是用于 socket 库应用的标准的 # include 文件的集合;还定义了一个符号,用来向 socket 库头文件示意,表示它将被包含在 socket 库代码内。这将使一些仅仅被定义和用于库子程序本身的附加符号定义在 socket 库头文件中。

18.1.6 socklib.h 头文件

```
#include <netinet/in.h>

/* 1 */
#define S_DELAY  0
#define S_NDELAY  1

/* 2 */
#ifdef S_LIBRARY
#   define S_RESET  0
#   define S_SET  1
#   define S_NAMLEN  64
#endif

/* 3 */
typedef struct
{
    struct sockaddr_in sin;
    int sinlen;
    int bindflag;
    int sd;
} SOCKET;

/* 4 */
SOCKET *sopen(void);
int sclose(SOCKET *);
int sserver(SOCKET *, int, int);
int sclient(SOCKET *, char *, int);
```

1. 这些是符号常量 S_DELAY 和 S_NDELAY 的定义,用来选择阻塞或不阻塞服务程序操作。

2. 仅仅定义在库本身的符号。

3. 在 socket 库中的 SOCKET 结构的定义，它等价于标准 I/O 库中的文件结构。在文件前面的 # include <netinet/in.h> 行是用来定义 SOCKET 结构内的 sockaddr_in 结构。

4. 在 socket 库中的函数的 ANSI 函数协议，编译程序可以在应用程序中用来检查参数个数和类型的正确用法。

18.2 样本服务程序

在两台机器之间传递文件有许多方法；每种方法各有其优缺点。大部分要求用户在文件传递前进行登录(login)。下列 C 代码段演示了使用 socket 和极小的 socket 库来实现一个简单的服务程序，它允许客户程序作出请求用一个较简单的协议下载文件。

在服务程序的机器上，所有远程用户可以访问的文件被放入单个目录中。然后从那个目录中执行服务程序。随后的客户程序对那个目录中的任何文件的请求将由该服务程序提供而无须任何种类的登录过程或作出安全检查。换句话说，该服务程序为任何人自由访问这组文件提供服务，只要他们有客户程序软件的拷贝，并且连接到运行该服务程序的机器所使用的网络上。

由于下载大的文件可能花相当长时间，如果当前请求仍旧在进行时，需要某种机制处理随后的客户程序请求。下面给出一个样本服务程序，它在无限循环中调用 sserver()用来处理这种情况。当一个客户程序连接一建立，sserver()就返回一个 socket 描述符。这个服务程序使用 fork()系统调用创建一个子女进程。由于子女进程能访问双亲进程打开的所有文件和 socket，故当双亲进程循环等待后续的客户程序连接时，它可以处理文件传递请求。大量的客户程序可以用这种方法同时传递文件，每一个客户程序都有专门为它创建的子女进程来处理它的要求。

样本客户程序和服务程序使用的简单的请求/响应协议：客户程序给服务程序指定一个文件名，然后服务程序发送那个文件的内容给该客户程序。给予服务程序的文件名分为两个部分：首先发送一个字节给服务程序指定文件名的长度，然后发送文件名本身的字节。服务程序对文件名的合法性和有效性做一些简单的检查，如果这些通过了，则服务程序发送文件字节给客户程序。在传递结束时，服务程序(子女)进程结束与这个客户程序的连接。

```
/*****************************************************
EXAMPLE SOCKET SERVER(样本 socket 服务程序)
*****************************************************/

#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <fcntl.h>
#include "socklib.h"

/* Server's communication port number(服务程序的通信端口号) */
```

```
#define PORT 2121

/*******************************************
MAIN - A very simple file transfer program, which sets up a concurrent socket server
       on the host machine on port number PORT to transfer any files from the current
       directory on request.
       (这是一个非常简单的文件传递程序,它在主机的 PORT 端口号上设置并发的 socket 服务程
       序,在请求时从当前目录中传递任何文件。)
*******************************************/
main(void)
{
    SOCKET *sp;
    int sd;

    /* Set SIGCHLD for no zombies(为了不产生 zombie 设置 SIGCHLD) */
    signal(SIGCHLD, SIG_IGN);

    /* Create server's socket(创建服务程序的 socket) */
    if ((sp = sopen()) == 0)
        /* Exit server if this fails(如果失效,退出服务程序) */
        fatal("sopen()");

    /* Repeatedly accept and service client requests(反复接受和服务客户程序请求) */
    for (;;)
    {
        /* Connect to client(连接客户程序) */
        if ((sd = sserver(sp, PORT, S_DELAY)) == -1)
        fatal("sserver()");

        /* Fork() a child process(用 Fork()创建一个子女进程) */
        switch (fork())
        {
        /* Deal with client request in child(在子女进程中处理客户程序请求) */
        case 0:
            do_service(sd);

        /* Deal with fork() failure(处理 fork()失败) */
        case -1:
            fatal("fork()");
        }

        /* In parent, close client and loop for next connection
        (在双亲进程中关闭客户程序并进入下一连接循环) */
        close(sd);
```

```
    }
}

/*********************************************
FATAL - This routine is called when an error condition is returned to the daemon,us-
        ually from one of the system calls it makes. The function first prints the
        message pointed to by <text> and then terminates the process.
        (通常当错误条件从系统调用返回给守护进程时，调用这一子程序，这个函数首先打印由
        <text>所指向的信息,然后结束该进程。)
*********************************************/
fatal(char *text)
{
    fprintf(stderr, "Error in %s\n", text);
    exit(1);
}

/*********************************************
DO SERVICE - This function is called in the child process that is created when a cl-
        ient makes a successful connection. Its task is to service the client's re-
        quest and supervise the required file transfer. The parameter <sd> is the
        socket descriptor to use for the communication
        (当一个客户程序进行了成功的连接时,在所创建的子女进程中调用这个函数。它的任务是
        服务客户程序的请求,并且监督所请求的文件的传递。参数<sd>是用于通信的 socket
        描述符。)
*********************************************/
do_service(int sd)
{
    int i, fd;
    char *name;
    char c, namlen;

    /* Get length of file name(取得文件名的长度)*/
    if (read(sd, &namlen, 1)! = 1)
        fatal("namlen read()");

    /* Space for name + " open()"(文件名和" open()"使用的空间) */
    if ((name = (char *)malloc(namlen + 8)) == 0)
        fatal("malloc()");

    /* Get required file name into name[]( 把所要求的文件名放人 name[]中) */
    for (i = 0; i<namlen; ++i)
        if (read(sd, &name[i] , 1)! = 1)
            fatal("file name read()");
```

```
    /* Make name[] a string(使 name[]成为一字符串) */
    name[i] = '\0';

    /* Check name[] is in current directory(检查 name[]是否在当前目录中) */
    if (strchr(name, '/'))
        fatal("illegal file name");

    /* Open specified file(打开指定的文件) */
    if ((fd= open(name, O_RDONLY))==-1)
        fatal(strcat(name, " open()"));

    /* Transfer contents to client(传递文件内容给客户程序) */
    while ((i = read(fd, &c, 1))! =0)
        if (i==-1)
            fatal("file read()");
        else if (write(sd, &c, 1)! =1)
            fatal("write() ");

    /* Terminate child and with it the connection to this client
    (结束子女进程以及与这个客户程序的连接) */
    exit(0);
}
```

即使它能工作,当前使用的服务程序的编码方法是不完善的。但是在我们讨论守护进程之后,我们将再次回到这段代码。

18.3 样本客户程序

下面给出的 C 代码是和刚才说明的服务程序配套的客户程序。假设这个客户程序被称为 getfile 调用,则它应该按如下方式使用:

```
getfile host remote_file local_file
```

其中 host 是运行服务程序的机器的主机名,remote_file 是从服务程序获得的文件名,客户程序用 local_file 作为文件的拷贝的名称。

```
/*****************************************************
EXAMPLE SOCKET CLIENT(样本 socket 客户程序)
*****************************************************/

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

```
#include "socklib.h"

/* Server's communication port number(服务程序的通信端口号)*/
#define PORT 2121

/*************************************************************
MAIN - This program implements the client end of a simple file transfer system
       using the tiny socket library. It should be invoked with:
       getfile <machine_name> <remote_file_name> <local_file_name>
       (这个程序利用极小 socket 库实现简单的文件传递系统的客户程序。
       它应该按下列方式使用:
       getfile <machine_name> <remote_file_name> <local_file_name>)
*************************************************************/
main(int argc, char **argv)
{
    SOCKET *sp;
    FILE *fp;
    int sd, i;
    char namlen, c;

    /* Check correct usage(检查正确的用法)*/
    if (argc! = 4)
        fatal("command line parameters");

    /* Open local file(打开本地文件)*/
    if ((fp = fopen(argv[3], "w")) == 0)
        fatal("opening local file");

    /* Create client's socket(创建客户程序的 socket)*/
    if ((sp = sopen()) == 0)
        fatal("sopen()");

    /* Make contact with server(和服务程序联系) */
    if ((sd = sclient (sp, argv[1], PORT)) == - 1)
        fatal("sclient()");

    /* Get length of remote file name(获得远程文件名长度) */
    namlen = strlen(argv[2]);

    /* Send name length to server(把文件名长度发送给服务程序)*/
    if (write(sd, &namlen, 1)! = 1)
        fatal("sending namlen");

    /* Send name to server(把文件名发送给服务程序)*/
```

```
    for (i = 0; i < namlen; ++i)
        if (write(sd, &argv[2][i], 1)! = 1)
            fatal("sending file name");

    /* Read file bytes from server(从服务程序读文件字节) */
    while ((i = read(sd, &c, 1))! = 0)
        if (i == -1)
            fatal("reading remote file");
        /* Write bytes to local file (写字节到本地文件) */
        else if (putc(c, fp) == EOF)
            fatal("writing local file");

    /* Tidy up open files and sockets(清理打开的文件和 socket) */
    close(sd);
    fclose(fp);
    sclose(sp);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
FATAL - Function to print a fatal error message to the standard error
        device and then terminate the client process.
        (函数在标准错误设备上打印致命错误信息,然后结束客户进程。)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
fatal(char *text)
{
    fprintf(stderr, "Error in %s\n", text);
    exit(1);
}
```

练习

1. 写一个简单的程序,用来确定在单个进程内可以同时打开多少个 socket。
2. 利用极小 socket 库写简单的客户程序和服务程序,用来给客户程序提供有关运行服务程序的系统的某些信息(例如当前登录的用户数,或者在口令文件中 root 以外的用户识别号的个数,等等)。
3. 利用极小 socket 库写一个简单的 chat 服务程序,它将允许多个客户程序连接并且发送正文信息给服务程序。当该服务程序从一个客户程序接收一条信息时,它将把这条信息广播给所有当前连接的客户程序。用户应该使用 telnet 作为客户程序。

解答

1. 假定同时打开的 socket 的数目不大于 1000, 下列程序段将显示一个数目的清单,最后就是你要的数目:

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

```
#define  MAX  1000
main()
{
    SOCKET * sp[MAX];
    int i, j;

    for (i = 0; (sp[i] = sopen())! = 0 &&  i < MAX; ++ i)
        printf("%d\n", i);

    for (j = 0; j < i; ++ j)
    sclose(sp[j]);
}
```

2. 这是一个简单的练习，几乎可以直接使用本章的样本客户程序和服务程序得到它的答案。
3. 为了使这个程序在单个进程内工作，需要对 socket 使用 O_NDELAY 标志。当没有可个供使用的正文时，从客户程序的读将不阻塞服务程序。这使得服务程序能继续对其他客户程序进行检查，看是否已有任何准备好的正文。下面的程序是问题的答案。程序前面的 3 条 #defines 语句用来指定服务程序使用的端口号，接受行的最大长度以及同时连接的最大客户程序数：

```
#include  <stdio.h>
#include  <string.h>
#include  <fcntl.h>
#include  <errno.h>
#include  "socklib.h"

#define  PORT 2121
#define  MAX_LEN  80
#define  MAX_USERS  10

#define  LINE_READY  1
#define  LOGGED_OUT  2

struct client
{
    int sd;
    char in_use;
    char next;
    char text[MAX_LEN + 1];
} user[MAX_USERS];

main()
{

    SOCKET * sp;
```

```
    int i, sd, flags;

    if ((sp = sopen()) == 0)
        fatal("sopen()");

loop:
    if ((sd = sserver(sp, PORT, S_NDELAY))! = -1)
        grab_user_slot(sd);
    else if (errno! = EWOULDBLOCK)
        fatal("sserver()");

    for (i = 0; i < MAX_USERS; ++i)
        if (user[i].in_use)
            switch(user_state(i))
            {
            case LINE_READY:
                broadcast_line(i);
                break;

            case LOGGED_OUT:
                free_user_slot(i);
                break;
            }

    goto loop;
}

user_state(int slot)
{
    char *ptr;
    int n;

    ptr = &user[slot].text[user[slot].next];

    while ((n = read(user[slot].sd, ptr, 1)) == 1)
    {
        if (*ptr == '\n')
        {
            *(ptr + 1) = '\0';
            user[slot].next = 0;
            return LINE_READY;
        }

        if (++user[slot].next == MAX_LEN)
```

```
            --user[slot].next;
        else
            ++ptr;
    }

    if (n == 0)
        return LOGGED_OUT;

    return 0;
}

broadcast_line(int slot)
{
    int i, n;

    for (i = 0; i < (MAX_USERS; ++i)
        if (user[i].in_use && i! = slot)
        {
            n = strlen(user[slot].text);
            write(user[i].sd, user[slot].text, n);
        }
}

grab_user_slot(int sd)
{
    int i, flags;

    for (i = 0; i < MAX_USERS; ++i)
    {
        if (!user[i].in_use)
        {
            user[i].in_use = 1;
            user[i].sd = sd;
            user[i].next = 0;

            if ((flags = fcntl(sd, F_GETFL)) == -1
            || fcntl(sd, F_SETFL, flags | O_NDELAY) == -1)
                fatal("fcntl()");

            break;
        }
    }
    if (i == MAX_USERS)
    {
```

```
        write(sd, "Too many users - sorry...\n", 26);
        close(sd);
    }
}

free_user_slot(int slot)
{
    user[slot].in_use = 0;
    close(user[slot].sd);
}

fatal(char *mess)
{
    printf("Error: %s\n", mess);
    exit(1);
}
```

第十九章　信　　号

信号是一种机制，进程通过信号知道系统中正在出现的事件。信号的一个重要特点是它们是异步的，这表示进程在它的执行期间的任何时刻都可能接到信号，甚至可能当进程正在执行系统调用时接到信号。因而它必须随时为响应信号做好准备。一般讲，如果系统调用被信号所中断，则它将返回错误值，并且将 errno 置为 EINTR 值。重新继续被中断的调用应该是收到信号的进程的责任。

每一种信号类型都给予一个符号名，当前有三十多个，通过 <signal.h> 头文件进行访问。最常用的信号值和它们的符号名是：

值	符号	用途
1	SIGHUP	在控制终端上发出的结束信号
2	SIGINT	从键盘来的中断(Ctrl-c)信号
3	SIGQUIT	从键盘来的退出(Ctrl-\)信号
8	SIGFPE	浮点异常(例如被 0 除)
9	SIGKILL	结束接收信号的进程
14	SIGALRM	alarm()系统调用结束时的暂停(timeout)
15	SIGTERM	默认的 kill 命令信号
17	SIGCHLD	表示子女进程停止或结束的信号
19	SIGSTOP	从键盘来的停止执行(Ctrl-z)的信号

每种信号类型都有相关的操作。当进程发送信号时，内核将代表该进程执行这个操作。对于大多数信号的默认操作是结束接收信号的进程；然而，一个进程通常可以请求系统采取某些代替的操作。各种代替操作是：

* 忽略信号。随着这一选项的设置，进程将忽略信号的出现。有两个信号不可以被忽略：SIGKILL，它将结束进程；和 SIGSTOP，它是作业控制机制的一部分，将挂起作业的执行。
* 恢复信号的默认操作。
* 执行一个预先安排的信号处理函数。进程可以登记特殊的信号处理函数。当进程收到信号时，信号处理函数将像中断服务子程序一样被调用。当从该信号处理函数返回时，控制被返回给主程序，并且继续正常进程的执行。

19.1　signal 系统调用

在 Linux 系统中存在几个系统调用，允许在单个信号的基础上设置各种操作。为了和其它 UNIX 风格兼容，除了稍后我们将看到的 POSIX 规定的调用外，Linux 支持 signal()系统调用。signal()调用的原型是：

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

void (*signal(int signum, void (*handler)(int)))(int);
```

这个原型看上去极其复杂,但是它所要说的全部就是 signal()是一个取两个参数的调用:第一个(signum)是在前面清单中的信号值,它正被设置一项操作。第二个(handler)是指向一个函数的指针。该函数取单个整型参数,并且不返回任何内容(void)。signal()的返回值本身是指向一个函数的指针,该函数取单个整型参数并且不返回任何内容(void)。一段简单的代码将有助于说明这一点:

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

int  ctrl_c_count = 0;
void (* old_handler)(int);
void ctrl_c(int);

main()
{
   int c;

   old_handler = signal(SIGINT, ctrl_c);

   while ((c = getchar())! = '\n');

   printf("ctrl-c count = %d\n", ctrl_c count);

   (void) signal(SIGINT, old_handler);

   for (; ;);
}

void ctrl_c(int signum)
{
   (void) signal(SIGINT, ctrl_c);
   + + ctrl_c_count;
}
```

这个程序事实上执行平常的获得字符的操作,直到敲入一个新行字符(newline)为止,然后进入无限循环中。通常用键盘产生的中断信号(例如 Ctrl-c)可以打断这样一个程序。但是在此情况下,程序安排捕获 Ctrl-c 信号(SIGINT),并且利用它来执行 一个称为 ctrl_c()的信号处

理函数。仅在键盘上敲入一个新行字符(newline)后,SIGINT 的原来的操作(很可能是默认操作)才被恢复。由在 main()函数中的第一个语句完成设置信号处理程序:

```
old_handler = signal(SIGINT, ctrl_c);
```

signal()的两个参数是:信号值,它的操作被设置(这里是 SIGINT,键盘中断信号);以及一个指向函数的指针,当这个中断信号出现时,将调用该函数。signal()调用返回旧的信号处理程序的地址。在此情况下,它被赋给变量 old_handler,使得原来的信号处理程序稍后可以被恢复。

一旦信号处理程序放在应放的位置,进程收到任何中断(SIGINT)信号将引起信号处理函数的执行。这个函数增加 ctrl_c_count 变量的值以保持对 SIGINT 事件(events)出现次数的计数。注意信号处理函数也执行另一个 signal()调用,它重新建立 SIGINT 信号和 ctrl_c() 函数之间的联系。这是必需的,因为当信号出现时,用 signal()调用设置的信号处理程序被自动地复位,使得随后的同一类型信号将只执行信号的默认操作。

这是早期 UNIX 的 signal()调用的主要问题(以及为什么 POSIX 信号不同的原因)。问题是当信号出现时,信号的默认操作立即被恢复。处理程序重新建立信号和函数之间的联系需要一段短的(但实际存在的)时间间隔。在这段时间间隔内,如果收到另一个同样类型的信号,则将发生默认操作,通常它将结束该进程。

19.2 sigaction 系统调用

POSIX 委员会的信号处理版本比早期的 signal()调用稍许复杂些,但是它确实规定了克服早期问题的信号语义。

在 POSIX 下,每个进程有一个信号掩码(signal mask),这是一组当前被阻塞传递的信号。如果发送一个被阻塞的信号给该进程,它将被加到该进程中挂起的信号集合中。当阻塞被移开时,它将被传递。

当一个信号被发送时,它被自动地加到接收进程的信号掩码中,使得在当前的信号正在被服务时,下一个同类信号的出现将被阻塞。当信号处理程序正常返回时,则信号掩码被恢复到它的先前值。

设置信号处理程序的主要系统调用称为 sigaction(),它的原型是:

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

int sigaction(int signum, struct sigaction *new, struct sigaction *old);
```

改变了只规定一个信号处理函数的做法,sigaction()调用取指向 sigaction(new)结构的指针作为一个参数。sigaction 结构的格式是:

```
struct sigaction
```

```
{
  void (* sa_handler)(int);
  sigset_t sa_mask;
  unsigned long sa_flags;
  void (*sa_restorer)(void);
};
```

如同你见到的那样,指向处理函数的指针已经被移到这个结构的 sa_handler 域中。在这个结构中的 sa_mask 域是一个额外的信号掩码,当该信号出现时,它被逻辑或(OR)到该进程的信号掩码中。当信号处理程序执行时,这个额外的掩码保持有效。sa_flags 域是几个位标志的逻辑或(OR)组合,其中两个主要的标志是:

SA_ONESHOT	当信号出现时,将信号操作置为默认操作;
SA_NOMASK	忽略 sigaction 结构的 sa_mask 域。

如果你使用 signal()调用而非 sigaction()调用,这两个标志全都被默认地设置。

sigaction()的返回值不像 signal()调用那样包含指向旧的信号处理程序的指针,代替的是另一个参数 old 被传递给 sigaction()。这是指向另一个 sigaction 结构的指针,它将由 sigaction()调用用旧的 sigaction 结构细节填入。如果需要的话,它允许稍后恢复旧的值。

19.2.1 特殊的 SIGCHLD 反应

有时你希望运行一个产生大量子女进程而又不想使用 wait()等待它们结束的进程。默认地,处于这种状态下的子女进程将变成 zombie(僵死的进程),直到将来某个进程(可能是 init 进程)使用 wait()等待它们为止。如果双亲进程是一个活得非常长的进程,则这些 zombie 将长时间留在系统中。在这种情况下,创建 zombie 只是浪费系统的资源。在 Linux 下有几种方法解决这个问题,最简单的做法是像下面那样把 SIGCHLD 信号的操作置为 SIG_IGN:

```
signal(SIGCHLD, SIG_IGN);
```

因为 SIGCHLD 的默认操作是被忽略,设置这个信号为 SIG_IGN 并无其它有用的目的。因此在 Linux 中被用作一项特殊功能,告诉内核不产生调用进程的子女进程的 zombie。

19.2.2 信号掩码

由一个专门的函数集合来执行设置和修改信号掩码,它们的原型是:

```
#include  <signal.h>

int sigemptyset(sigset_t *mask);
int sigfillset(sigset_t *mask);
int sigaddset(sigset_t *mask, int signum);
```

```
int sigdelset(sigset_t *mask, int signum);
int sigismember(sigset_t *mask, int signum);
```

sigemptyset()函数取指向类型为 sigset_t 的信号掩码的指针(mask)作为参数,并且使它的值为零,以保证没有任何信号被标志为阻塞。函数 sigfillset()填信号掩码以保证所有的信号被标志为阻塞。sigaddset()和 sigdelset()函数允许分别对个别信号(signum)增加和删除信号阻塞。最后一个函数 sigismember()允许你用来确定特定的信号是否在信号掩码中被标志为阻塞。.

进程也可以利用 sigprocmask()系统调用改变和检查它自己的信号掩码的值:

```
#include  <signals.h>

int sigprocmask(int fcn, sigset_t *mask, sigset_t *old);
```

mask 参数指向信号掩码值,它将按照 fcn 规定用作进程的信号掩码。各种不同的 fcn 值及其含义如下:

```
SIG_BLOCK     mask 规定附加的阻塞信号;
SIG_UNBLOCK   mask 规定一组不予阻塞的信号;
SIG_SETMASK   mask 变成新进程的信号掩码。
```

SIGKILL 和 SIGSTOP 信号不能被阻塞,并且任何这样做的企图将被沉默地忽略。

参数 old 是指向信号掩码的指针,在调用后它将包含以前的信号掩码值,使得如果要求的话,以后可以恢复它。

进程可以调用 sigpending()系统调用来检查是否有挂起的阻塞信号:

```
#include  <signals.h>

int sigpending(sigset_t *mask);
```

由 mask 所指的信号掩码被设置为指示任何挂起的信号。

19.3 kill 系统调用

到此为止所考虑的大部分信号或者自动地由一个进程改变状态及由类似浮点异常的硬件错误产生,或者它们由用户在键盘上用中断和退出信号干涉而产生。除此之外,如果进程有适当的权限它也可能故意发送任一信号给另一个进程。这可以用 kill()系统调用来完成:

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

int kill(pid_t pid, int sig);
```

参数 sig 规定发送哪一个信号，参数 pid 规定它发送到何处。pid 各种不同值具有下列意义：

pid > 0　　这将发送信号 *sig* 给进程识别号为 *pid* 的进程。

pid = 0　　发送信号 *sig* 给所有和调用进程具有相同进程组识别号的进程。

pid = -1　　发送信号 *sig* 给系统中除去 *init* 进程和调用进程以外的所有进程(这是 *Linux* 的规定)。

pid < -1　　发送信号 *sig* 给进程组识别号和 *pid* 的绝对值相同的进程组(即进程组-pid)中的所有进程。

为了用 kill()发送信号，调用进程的有效用户 ID 必须是 root，或者它必须和接收进程的实际或有效用户 ID 相同。

19.4 pause 系统调用

有时你可能希望写一个程序，它需要和另一个程序协作来完成它的功能。例如，假设两个进程通过一个 FIFO 文件通信。这两个进程之一将创建该 FIFO，然后它们俩都将利用它来进行通信。

显然，没有创建该 FIFO 文件的进程在它可以继续之前，需要等待另一个进程创建该 FIFO 文件。达到进程同步的一种方法是使用 pause()系统调用：

```
#include  <unistd.h>

int pause(void);
```

这个系统调用除了挂起作出该调用的进程之外不做任何事。如果该进程随后接收一个信号，它有一个信号处理程序，则当该处理程序已经执行时，pause()调用将返回一个错误值，并且将 errno 置成 EINTR。这个返回值和错误条件可以被忽略，这是从 pause()返回的唯一方法。

结论是当一个进程需要等待另一个进程完成某项操作时，它将执行 pause()调用。当这项操作已被完成时，另一个进程可以发送一个预先约定的信号给这一暂停的进程，它将强迫 pause()返回，并且允许收到信号的进程恢复执行，知道它正在等待的事件现在已经出现。

19.5 alarm 系统调用

最后一个和信号有关的系统调用称为 alarm()。这个调用的原型是：

```
#include  <unistd.h>

long alarm(long secs);
```

每个进程都有一个闹钟计时器与之相联，在经过预先设置的时间后，进程可以用它来给自身发送 SIGALRM 信号。alarm()调用只取一个参数 secs，它是在闹钟关闭之前所经过的秒数。如果传递一个 0 值给 alarm()，这将关闭任何当前正在运行的闹钟计时器。

alarm()返回值是以前的闹钟计时器值。如果当前没有设置任何闹钟计时器，这将是零，或者是当作出该调用时，闹钟的剩余时间。

练习

1. 写一个 C 程序，它捕获所有可以被捕获的信号，并且当该进程接收到任何信号时，显示一条合适的信息。
2. 写一个程序，它读取和显示在键盘上敲入的字符。启动一个 alarm()，如果在一段短时间内无字符敲入，它将打印一条信息。
3. 基于下列 shell script，写一个程序发送 SIGKILL 信号给除去你的当前的 shell 和当前进程以外所有你正在运行的进程：

```
pidlist=`ps -aux | grep "^$LOGNAME" | grep -v $$ | \
  tr -s " " | cut -f 2 -d " "`
for pid in pidlist
do
  kill -9 $pid >/dev/null 2>/dev/null
done
```

解答

1. 这可以用单个信号处理程序来完成，接收到的信号值作为一个参数被传递给信号处理程序：

```
void trapper(int);

main()
{
  int i;

  for (i = 0; i<32; ++i)
    signal(i, trapper);

  for(; ;);
}

void trapper(int sig)
{
  signal(sig, trapper) ;
  printf("Just received signal number: %d\n", sig);
}
```

2. 在下列代码中，while 循环用于从键盘读字符，并把它们显示在屏幕上。当无字符正在被敲入时，该

循环被阻塞在 read()调用。在循环开始之前正好启动一个 20 秒闹钟。每当写一个字符时,该闹钟被重置回 20 秒。当 20 秒内没有写任何字符时,闹钟关闭,中断 read()和调用 trapper()函数:

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

void trapper(int);

main()
{
  char c;

  signal(SIGALRM, trapper);
  alarm(20L);

  while (read(0, &c, 1) == 1)
  {
    write(1, &c, 1);
    alarm(20L) ;
  }

  printf("read() interrupted... \n");
}

void trapper(int sig)
{
  printf("You stopped typing... \n");
}
```

3. 这个问题可以容易地把 shell script 翻译成 C 代码利用 pipe()、dup()、fork()和 exec()系统调用加以解决。当你把它翻译成 C 代码时,在 shell script 对 shell 的 PID 进行 grep -v 处理处,你也需要对你自己的 PID 进行 grep -v 处理。这样做当程序运行时,就不会结束你自己正在运行的进程。你可以在这个程序的开始创建所有的管道,使得当需要时它们全都可供使用。不要忘记,在做过 I/O 重新定向后和对每个命令调用 exec()之前,你应该关闭所有该进程不需要的文件描述符(除去 0,1 和 2 以外),否则你将得不到沿着管道行(pipeline)传送的文件结束符(end-of-file)。

第二十章　极小的 shell——实例研究

现在你已经看到许多系统调用，足以理解当 shell 执行简单的命令管道行(pipeline)时需要做些什么。我们将不去考察实际的 shell 代码，它太复杂以至无法在这里以任何切合实际的方法来讨论它，我们将研究一个极小的 shell，它正好具有最低限度必不可少的内容，足以说明某些基本原理。

这个极小的 shell 将给你显示一个提示符，你可以敲入任何简单的命令后随该命令所要求的参数来响应它。这个极小的 shell 将执行你的命令，然后返回另一个提示符。应该注意：这个极小的 shell 不能处理实际 shell 能接受的正则表达式。

除了这种简单的情况外，这个极小的 shell 也允许利用熟悉的 | 、< 、> 和 > > 运算符使用管道和 I/O 重新定向，并允许用 & 运算符在后台执行管道行。

20.1　数据结构

为了理解这个极小的 shell 的操作，要研究的第一件事是所包含的数据结构。为了简化这个极小的 shell 的内部结构，它的数据结构全是在编译时静态分配的。例如这意味着将利用数组来存储数据而不是动态地分配连接列表(linked lists)。这样做的结果是这个极小的 shell 对某些容量施加了固定的限制，而实际的 shell 无须施加这样的限制。特别是，下列事物是固定的，需要重新编译这个极小的 shell 来改变它们：在 shell 提示符后面输入的行的最大长度，用于 I/O 重新定向的文件名的最大长度，每个简单的命令的命令行参数的最多个数和在一个管道行中简单的命令的最多个数。

所有这些全局变量和数据结构的定义和说明出现在文件 def.h 中，如下所示：

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

#define  TRUE  1
#define  FALSE  0
#define  OKAY  1
#define  ERROR  0
#define  MAXLINE  200          /* 输入行的最大长度 */
#define  MAXARG  20            /* 每个简单命令的参数的最多个数 */
#define  PIPELINE  5           /* 在一管道行中简单命令的最多个数 */
#define  MAXNAME  100          /* I/O 重新定向的文件名的最大长度 */

char  line[MAXLINE+1];         /* 用户敲入的输入行 */
```

```
char   *lineptr;              /* 指向 line[]中当前位置的指针 */
char   avline[MAXLINE+1];     /* 取自 line[]的 Argv 串 */
char   *avptr;                /* 指向 avline[]中当前位置的指针 */
char   infile[MAXNAME+1];     /* 输入重新定向文件名 */
char   outfile[MAXNAME+1];    /* 输出重新定向文件名 */

int   backgnd;                /* 如果用 & 字符结束管道行为 TRUE,否则为 FALSE */
int   lastpid;                /* 在管道行中最后简单命令的 PID */
int   append;                 /* 对于附加重新定向符(>>)为 TRUE,否则为 FALSE */

struct cmd
{
char   *av[MAXARG];
int   infd;
int   outfd;
} cmdlin[PIPELINE];           /* Argvs 和 fds,每个简单命令一个 */
```

当你敲入一条命令行给这个极小的 shell 以响应它的提示符时,你敲入的字符被直接地存入 line[]数组中。假定你送入命令:

```
tsh: echo hello world
```

你已经知道为了执行这条命令,极小的 shell 需要用 fork()建立一个子女进程,然后在此子女进程中用 exec()执行 echo 命令。因为 echo 命令取参数,必须从命令行(在 line[]数组中)提取它们,并且把它们打包放到一个 argv 类型数据结构中传给 exec()(在这种情况下为 execvp()系统调用)。

在极小的 shell 中,在命令行中的分开的字被提取,并且被存放在数组 avline[]中作为一系列分开的串。然后对于在命令行中每个简单命令,一个指针数组被设置指向那条命令的相应的单字。该指针数组被包含在 cmd 结构中,其中每一个可供管道行中每个简单命令使用。这组结构本身被存放在称为 cmdlin[]的数组中。

图 20.1 用来说明当送入下列命令管道行时 line[]、avline[]和 cmdlin[]数组的内容:

```
tsh: ls | wc -w
```

20.2 代码

在看过基本的数据结构和它们的使用后,现在让我们来考虑代码(Code)。这个极小的 shell 基本上按由顶向下方式并且以我们将对它进行讨论的顺序设计的。

20.2.1 main 函数

任何解释程序的顶层设计基本上是一个循环,它只是取得要做的某些事,然后离开循环去

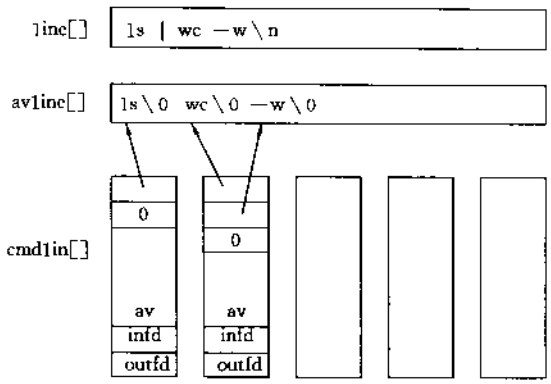

图 20.1 line[]、avline[]和 cmdlin[]数组的使用

完成它。这个极小的 shell 也不例外，所以 main()函数特别简单：

```
#include "def.h"
main(void)
{
    int i;

    initcold();

    for(;;)
    {
        initwarm();

        if(getline())
            if(i = parse())
                execute(i);
    }
}
```

第一件事是 main()调用一个称为 initcold()的一次性的初始化函数。随后进入一个无限循环，它重复打印提示符，获得输入行，对它们进行语法分析来提取简单命令和它们的参数，然后执行每个简单命令，处理 I/O 重新定向和管道。

详细地说，循环体首先调用 initwarm()来对每个输入命令行进行初始化和显示 tsh:提示符。后随 getline()调用，它把用户的命令行拷贝到 line[]数组中，如果送入的行的长度小于 MAXLINE，则返回一个非零值。

如果 getline()返回一个非零值，则接着调用 parse()函数。这个函数具有对 line[]数组的内容进行语法分析的任务，并且从中提取有关信息。这包括简单命令的名称和它们的有关的参数，任何 I/O 重新定向的输入和输出文件名以及管道行后台运算符(&)(如果规定的话)。

假设没有错误，parse()函数返回一个正值，它是输入命令管道行中的简单命令的个数。出错时 parse()返回 0 值。如果到目前为止还没有错误，则最终调用 execute()函数，由 parse()返

回的计数作为它的一个参数。execute()函数利用 fork()和 exec()依次执行每个简单命令,按要求创建管道(pipes)和打开 I/O 重新定向文件。

20.2.2 初始化函数

现在对初始化函数 initcold()的内容加以注释,实际上此函数不做任何事。这是因为这个极小的 shell 没有像实际 shell 那样的内部命令。这意味着像终止实际 shell 的 exit 命令在这个极小的 shell 中不能以同样的方式使用。因此,终止极小的 shell 的最容易的方法是从键盘发送一个中断(Ctrl-c)或退出(Ctrl-\)信号给它。显然,如果对 initcold()的内容不加注释,则这个极小的 shell 更难终止。

```
initcold(void)
{
/*
  signal(SIGINT, SIG_IGN);
  signal(SIGQUIT, SIG_IGN);
*/
}

initwarm(void)
{
  int i;

  backgnd = FALSE;
  lineptr = line;
  avptr = avline;
  infile[0] = '\0';
  outfile[0] = '\0';
  append = FALSE;

  for (i = 0; i < PIPELINE; ++i)
  {
    cmdlin[i].infd = 0;
    cmdlin[i].outfd = 1;
  }

  for (i = 3; i < OPEN_MAX; ++i)
    close(i);

  printf("tsh: ");
  fflush(stdout);
}
```

initwarm()函数在'得到一条命令,执行一条命令'循环的每次迭代开始处被执行。本质上,它只初始化全局变量,设置在 cmdlin[]数组中的每个简单命令结构的 infd 和 outfd 文件描述符为它们的默认值,关闭除了标准输入、输出和错误输出设备以外所有文件描述符,最后显示 tsh:提示符。

20.2.3 getline 函数

```
getline(void)
{
  int i;

  for (i = 0; (line[i] = getchar())! = '\n' && i < MAXLINE; ++i);

  if (i == MAXLINE)
  {
    fprintf(stderr, "Command line too long \n");
    return(ERROR) ;
  }

  line[i + 1] = '\0';
  return(OKAY);
}
```

这个函数从标准输入设备取字符并且把它们拷贝到 line[]数组中,直到或者遇到一个新行字符(\n)或者已经送入了 MAXLINE 个字符为止,在后一种情况下 line[]数组是满的。

如果已经送入了 MAXLINE 个字符,则函数返回 ERROR;否则一个串终止符(\0)被加到送入的最后字符后面,并且返回 OKAY 值。

20.2.4 parse 函数

一条命令管道行的句法如下:

```
cmd [ < filename ] [ | cmd ]... [ or filename ] [ & ]
```

这里 cmd 表示任何简单命令包括它的参数,or 是输出重新定向操作符(>或>>),方括号表示可选项,省略号(…)表示前面括号内的内容可以重复零次或多次。

把它全都放在一起,这说明一个极小的 shell 命令管道行开始于一个简单命令。然后可选择地后面随以一个输入重新定向操作符及有关的文件名。依次可以可选择地和重复地后面随以管道操作符,然后另一个简单命令。在最后的简单命令之后,也可规定可选的输出重新定向操作符及有关文件名(如果没有使用管道,最后的简单命令也就是第一个简单命令)。最后,整个管道行可以可选择地规定 & 操作符而使它在后台运行。

parse()函数需要确保它的输入的句法符合这个规范说明,同时提取属于每个简单命令的

句法记号(tokens)和任何可能指定的输入或输出重新定向的文件名。

```
parse(void)
{
  int i;

  /* 1 */
  command(0) ;

  /* 2 */
  if (check("<"))
    getname(infile) ;

  /* 3 */
  for (i = 1; i < PIPELINE; ++ i)
    if (check("|"))
      command(i) ;
    else
      break;

  /* 4 */
  if (check(">"))
  {
    if (check(">"))
      append = TRUE;

    getname (outfile) ;
  }
  /* 5 */
  if (check("&"))
    backgnd = TRUE;

  /* 6 */
  if (check("\n"))
    return(i);
  else
  {
    fprintf(stderr, "Command line syntax error\n");
    return(ERROR) ;
  }
}
```

在这个代码清单中的编号注释说明如下：

1. 首先调用 command()函数。稍后我们将看到,它知道如何从一个简单命令的命令行中提取单字。这些字将作为单独的字符串被拷贝到 avline[]数组中,在相应的 cmd 结构中的 av[]指针将被设置成指向这些字符串。传递给 command()函数的参数用来告诉它使用 cmdlin[]数组中的哪一个 cmd 结构。
2. 对第一条命令进行语法分析后,接下来利用 check()函数寻找可选的输入重新定向操作符。如果 check()函数在 line[]数组的当前位置找到匹配的参数串,它将返回一个真值,然后将调用 getname()函数提取后随的文件名,并将它放到 infile[]数组中。
3. 简单命令 0 已经被语法分析,所以下面的代码段寻找可选的管道操作符和后随的简单命令 1、2 等等。在 for 循环内如果发现一个管道操作符,则 command()函数被再次调用对下一个简单命令进行语法分析。此时 command()的参数由循环控制变量(i)提供。for 循环一直重复到没有发现更多的管道操作符或者达到'在管道行(pipeline)中简单命令的最大个数'限制为止。
4. 这一代码段检查输出重新定向操作符,然后寻找第二个 > 符号来检查是否是附加(append)输出重新定向操作符。如果是,则 append 标志被设置为 TRUE(在 initwarm()中被默认地设置为 FALSE)。如果出现任一输出重新定向操作符,则 getname()函数被用来拷贝有关的文件名到 outfile[]数组中。
5. 规范说明允许选用 & 操作符,接下来进行这一项检查。如果该操作符出现,则 backgnd 标志将被设置为 TRUE(在 initwarm()中被设置为 FALSE 值)。
6. 通过前面的代码的检查,输入行的语法分析现在应该已经完成,所剩下的应该是在行结尾的新行字符(\n)。如果是,则 parse()返回变量 i 的值,它给出了在命令行中找到的简单命令的个数。如果下一个字符不是新行字符,则显示错误信息,并且 parse()返回 ERROR 值。

20.2.5 command 函数

基本上,这个函数每次从 line[]数组中提取一个字,将命令名和它们的参数拷贝到 avline[]数组中。每当拷贝一个字,在数据结构的 cmdlin[]数组的 av[]数组中设置一个指向字的开始的指针:

```
command(int i)
{
  int j, flag, inword;

  for (j = 0; j < MAXARG - 1; ++j)
  {
    while (*lineptr == ' ' || *lineptr == '\t')
      ++lineptr;

    cmdlin[i].av[j] = avptr;
    cmdlin[i].av[j+1] = NULL;

    for (flag = 0; flag == 0;)
    {
      switch (*lineptr)
      {
      case '>':
```

```
        case '<':
        case '|':
        case '&':
        case '\n':
          if (inword == FALSE)
            cmdlin[i].av[j] = NULL;

          *avptr++ = '\0';
          return;
        case ' ':
        case '\t':
          inword = FALSE;
          *avptr++ = '\0';
          flag = 1;
          break;
        default:
          inword = TRUE;
          *avptr++ = *lineptr++;
          break;
        }
      }
    }
}
```

在这个极小的 shell 中,特别是在这个函数中,变量 lineptr 指向 line[]数组中当前的有关位置,变量 avptr 指向 avline[]数组中下一个要使用的字符位置。

command()函数由一对嵌套的 for 循环组成。内循环从 line[]数组拷贝一个字的全部字符到 avline[]数组中,并且用一个 \0 字节作为在 avline[]数组中的字的结束。外循环负责在 av[]数组的有关元素中设置指向该字的指针。

20.2.6 execute 函数

在每一完整的管道行中仅可以指定一个输入重新定向文件,附属于第一条简单命令;并且仅可以指定一个输出重新定向文件,附属于最后一条简单命令。显然,如果在该管道行中仅有一条简单命令,则它既是行中第一条命令又是最后一条简单命令,因而,它可以有它的输入和输出重新定向文件。

如果在管道行中的简单命令多于一条,例如,有 j 条简单命令,则需要创建 j - 1 个管道把这些简单命令连接在一起。记住这些点,现在我们可以研究 execute()函数的代码:

```
execute(int j)
{
  int i, fd, fds[2];
```

```
    /* 1 */
    if (infile[0]! = '\0')
      cmdlin[0].infd = open(infile, O_RDONLY) ;

    /* 2 */
    if (outfile[0]! = '\0')
      if (append == FALSE)
        cmdlin[j-1].outfd = open(outfile, O_WRONLY | O_CREAT | O_TRUNC, 0666) ;
      else
        cmdlin[j-1].outfd = open(outfile, O_WRONLY | O_CREAT | O_APPEND, 0666);

    /* 3 */
    if (backgnd == TRUE)
      signal(SIGCHLD, SIC_IGN) ;
    else
      signal(SIGCHLD, SIC_DFL) ;

    /* 4 */

    for (i = 0; i<j; ++i)
    {
      /* 5 */
      if (i<j-1)
      {
        pipe(fds);
        cmdlin[i].outfd = fds[1];
        cmdlin[i+1].infd = fds[0];
      }

      /* 6 */
      forkexec(&cmdlin[i]);

      /* 7 */
      if ((fd = cmdlin[i].infd)! =0)
        close(fd);

      if ((fd = cmdlin[i].outfd)! =1)
        close(fd);
    }
    /* 8 */
    if (backgnd = = FALSE)
      while (wait(NULL)! = lastpid);
}
```

回忆一下，execute()函数取一个整型参数，它是在命令管道行中简单命令的个数。这个值被赋给变量 j。对代码清单中的编号注释解释如下：

1. 如果已经指定了一个输入重新定向文件，则用 open()打开该文件，并且把所得到的文件描述符赋给简单命令 0(即命令行中第一条简单命令)的 cmd 结构中的 infd 域。注意，这个极小的 shell 程序犯了 Linux 程序设计中的一个最大致命的错误，它不对它使用的大多数系统调用的返回值进行检查，来找出错误的返回值。在代码中增加检查是一项非常容易的任务(留给读者作为一个练习)，为了使代码易于阅读，已简单地省略了它。
2. 类似地，如果已经指定了一个输出重新定向文件，则它将被 open()打开，并且把所得到的文件描述符赋给命令行中最后的(j-1)简单命令的 cmd 结构中的 outfd 域。这里的问题是存在两类输出重新定向(复盖和附加输出重新定向)，它需要以两种不同方式打开文件。在此情况下，正好由测试 append 标志的值来作出选择。
3. 下一代码段包含 zombie 子女进程。如果命令行运行在后台(backgnd == TRUE)，则 SIGCHLD 被设置，使得当子女进程结束时，zombie 子女进程将不被创建。这里这是正确的行动，因为 shell 将不使用 wait()等待它们。如果命令行运行在前台，则 SIGCHLD 被设置为它的默认操作，当子女进程僵死时，仍使系统从它们产生 zombie，直到这个 shell 使用 wait()等待它们为止。
4. 下一个代码段是一个循环，该循环体将对命令行中的每条命令执行一次，变量 i 确定下一次执行那一条简单命令。
5. 值 j-1 给出命令行中最后的简单命令的命令号。这个值大于零，它表示在管道行中存在多于一个简单命令。对于号码小于 j-1 的每个简单命令，需要创建一个管道，把 i 号命令的输出(outfd)通过该管道作为 i+1 号命令的输入(infd)。
6. i 号命令所需的所有文件描述符和命令行参数现在都可供使用，并且都已准备好，所以现在可以执行简单命令 i。forkexec()函数正好做它的名字所蕴涵的事，它创建一个子女进程来运行一条简单命令。因此，forkexec()函数仅在双亲进程中返回。
7. 如果任何包含 I/O 重新定向或管道的简单命令已经启动，则在这里关闭双亲进程中的文件描述符的拷贝。
8. 最后，当所有的简单命令已被启动，如果这一命令行正在前台运行，则双亲进程(极小的 shell)进入一个循环使用 wait()等待在管道行中的最后进程的结束，最后进程的进程 ID 已由 forkexec()函数设置在 lastpid 中。

20.2.7 forkexec 函数

forkexec()函数具有用 fork()系统调用创建一个子女进程的任务。然后在该子女进程中在利用 execvp()执行所要求的命令之前，它执行任何要求的 I/O 重新定向。

```
forkexec(struct cmd *ptr)
{
  int i,pid;

  /* 1 */
  if (pid = fork())
  {
    /* 2 */
    if (backgnd == TRUE)
```

```
        printf("%d\n", pid);
      lastpid = pid;
    }
    else
    {
      /* 3 */
      if (ptr->infd == 0 && backgnd == TRUE)
        ptr->infd = open("/dev/null", O_RDONLY);

      /* 4 */
      if (ptr->infd != 0)
      {
        close(0);
        dup(ptr->infd);
      }

      if (ptr->outfd != 1)
      {
        close(1);
        dup(ptr->outfd);
      }

      /* 5 */
      if (backgnd == FALSE)
      {
        signal(SIGINT, SIG_DFL);
        signal(SIGQUIT, SIG_DFL);
      }

      /* 6 */
      for (i = 3; i < OPEN_MAX; ++i)
        close(i);

      /* 7 */
      execvp(ptr->av[0], ptr->av);
      exit(1);
    }
}
```

forkexec()函数取指向正要执行的简单命令的 cmd 结构的指针作为它的参数。回忆一下这个结构包含该命令将要使用的的标准输入和标准输出的文件描述符(虽然尚未发生重新定向),它也包含指针数组,指向即将成为命令的 argv 参数的命令行参数(包括命令名称本身)。

在这个函数中的编号注释说明如下:

1. 第一项任务是使用 fork()创建一个子女进程 并且存贮它返回的进程 ID。
2. 如果这一命令行在后台运行,则当每个子女进程被创建时在双亲进程中将显示其进程 ID。每个子女进程的进程 ID 也被存储在变量 lastpid 中。因为命令行中的简单命令从左至右顺序地被执行,lastpid 的最终值将是最后被创建的简单命令(即最右边的简单命令)的进程 ID。如果这是在前台执行的命令行,这将是 wait()将要寻找的进程 ID。
3. 在子女进程中第一件要做的事是进行一项相当微妙的测试。如果这是在后台执行的命令行,则必须防止它从键盘取输入。事实上,在一个管道行中仅仅第一条简单命令可以从键盘取它的输入,因为所有其余的命令通过重新定向将从管道取它们的标准输入。所以,如果这是后台命令行,而且当前的简单命令的标准输入的文件描述符(infd)仍被设置为键盘(infd == 0),则它必定是管道行中的第一条简单命令。需要将它的标准输入从键盘移开,从而不给它试图从键盘读取的任何机会。在此情况下,做这件事的最简单和最安全的方法是设置自动的输入重新定向。用于这一重新定向的合适的文件就是标准文件 /dev/null,如果进程试图用 read()读它的内容,它将立即返回 EOF 值。利用这个简单的机制,使这个极小的 shell 避免了包含任何作业控制函数的必要。
4. 如果需要改变文件描述符的默认值,下一步实际上就是对这些文件描述符进行操作来实现输入和输出重新定向。
5. 如果这是前台的管道行,则将当前的简单命令设置成可以从键盘接受中断或退出信号来结束它的运行,除非它随后作出相反的规定。
6. 下一步是关闭除去标准输入、输出和错误输出以外的所有文件描述符。使得这个进程除了通过标准文件描述符 0,1 和 2 外不能干涉 shell 已经设置的的任何管道或 I/O 重新定向文件。
7. 最后,利用在 av[]数组中的值作为命令名和 argv 指针由 execvp()系统调用执行这条简单命令。

20.2.8 check 函数

```
check(char *ptr)
{
  char *tptr;

  while ( * lineptr == ' ')
    lineptr ++ ;

  tptr = lineptr;

  while ( *ptr! = '\0' && * ptr == * tptr)
  {
    ptr ++ ;
    tptr ++ ;
  }

  if ( * ptr! = '\0')
    return(FALSE);
  else
  {
    lineptr = tptr;
```

```
        return(TRUE);
    }
}
```

check()函数将传给它作为参数的字符串和 line[]数组中从 lineptr 位置开始的字符串相比较。如果找到匹配的字符串,则 lineptr 移过所匹配的字符串,并且返回 TRUE 值。如果 check()找不到匹配的字符串,则返回 FALSE 值,并且使 lineptr 保持不变,使得可以在 line[]数组中同样的位置开始对其它字符串进行检查。

20.2.9 getname 函数

```
getname(char *name)
{
    int i;

    for (i = 0; i<MAXNAME; ++i)
    {
      switch (*lineptr)
      {
      case '>':
      case '<':
      case '|':
      case '&':
      case ' ':
      case '\n':
      case '\t':
        *name = '\0';
        return;

      default:
        *name++ = *tlineptr++;
        break;
      }
    }
    *name = '\0';
}
```

这个函数仅将 MAXNAME 个字符长的文件名拷贝到由它的 name 参数所指的字符数组中。

练习

1. 利用 bash 和 tsh 送入下列含有错误的命令行,看从 shell 将会得到什么结果。所观察到的是否就是你所期待的?想要的是什么?试试其它的命令和其它的 shell(如果你有它们的话),现在又怎么想?

```
$ cat /etc/passwd > output1 > output2
$ cat </etc/passwd </etc/motd
$ cat /etc/passwd | | grep root
$ cat /etc/passwd > output3 | grep root
```

2. 给极小的 shell 增加一种简单机制，使它能执行内部的(built-in)命令。
3. 给极小的 shell 增加 cd 和 history 内部命令，并且使它能用！操作符重复执行历史表中前面的命令。

解答

1. 假设你认为错误信息很自然，标准的 shell 做得还可以，是不是？
2. 增加内部命令的简单方法是在 parse()函数的前面增加下列代码：

```
parse()
{
  int i;

  if (builtin())
    return 0;

  /* 1 */
  command(0);
/* etc... */
```

然后写一个 builtin()函数检查内部命令的关键字，并且当它发现它们时，执行任何要求的操作。如果 builtin()函数返回一个真(非零)值说明找到了一条内部命令，则 parse()将在那一点返回，并且不再试图进一步地执行该命令行：

```
builtin()
{
  if (check("exit"))
    exit(0);
  else
    return 0;
}
```

3. 为了增加这三个额外的内部命令，builtin()函数可以被修改如下：

```
builtin()
{
  if (check("exit"))
    exit(0);
  else if (check("cd"))
    return do_cd();
```

```
else if (check("history"))
  return do_history();
else if (check("!"))
  return do_repeat_command();
else
  return 0;
}
```

然后 do_cd()函数可以从 line[]数组中提取一个目录名,并且使用 chdir()系统调用来改变目录。对于！和 history 命令,你需要修改 getline()函数。使得从键盘送入命令行时,将它们存储起来。然后 history 命令可以列出存储的所有命令,以及！命令可以把相应的存储的命令拷贝回 line[]数组中。然后 do_repeat_command()可以从 builtin()返回一个零值,使得 line[]数组将被正常地进行语法分析。

第二十一章　各种系统调用

实际上还有许多系统调用(目前约有一百四十多个)。限于篇幅不能在这里进行详细讨论。但是,在我们结束之前,仍有少量的系统调用应该加以讨论。

所有可以从 shell 命令提示符执行的命令,只能调用某些基本的系统调用从操作系统取得所需的系统服务,才能完成它们的操作。在若干情况下,这表示一个新的系统调用专门提供一项特定的服务。下面的系统调用集合属于这种类型。

21.1　umask 系统调用

你将回想起 umask 命令用来设置文件创建时使用的掩码。每当建立文件时,umask 的值被用作一种安全措施,用来保证你运行的进程所创建的新文件的访问权限不会超出你想要给予的权限。它将为你创建的文件的权限位根据 umask 的值重新进行设置。

在内部,umask 命令调用相同名称的基本系统调用加以实现:

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

mode_t umask(mode_t mask);
```

调用 umask()系统调用将文件创建时使用的掩码设置为指定的 mask 值。只有在 mask 值中设置的权限位被保留,其余的权限位将被清除。

umask()的返回值是文件创建掩码的先前值,所以这个系统调用既可以用来获得又可以用来设置所要求的值。但是,遗憾的是不能在不设置新的掩码值的情况下直接获得旧的掩码值。

这表示为了看当前的掩码值,必须执行如下的一段代码:

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

mode_t get_umask()
{
   mode_t old;

   old = umask((mode_t)0);
   (void) umask(old);
   return old;
}
```

这个函数的第一行将掩码值设为零，作为副作用它将返回你感兴趣的掩码值，然后将它赋给变量 old。第二行将掩码恢复为它的原来值。最后一行将所要求的掩码的当前值返回给调用的程序。

21.2 mount 系统调用

实际的磁盘常常被划分成独立的分区(partitions)，并且每个文件系统记录在一个单独的分区上。每个这样的文件系统可以被装配到系统总的目录层次结构中，因此变成它的一个部分。装配文件系统的操作由 mount 命令完成，mount 命令本身使用基本的 mount()系统调用：

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

int mount(char *fspath, char *dir, char *fstype, unsigned long flags, void *data);
```

很少直接调用 mount()系统调用，特别是因为它仅可以供 root 用户使用。

参数 fspath 是你要装配的文件系统的磁盘分区的路径名(例如/dev/hdb2)，参数 dir 给出装配该文件系统的目录名。参数 fstype 规定磁盘分区上的文件系统类型(例如第二扩充文件系统，minix 或 DOS 等)，参数 flags 规定该文件系统将如何被装配(例如只读或只写)，参数 data 则是指向任意的信息结构的指针，它的内容(或者是否使用)依赖于被装配的特定的文件系统类型。

21.3 umount 系统调用

一个文件系统一旦被装配，可以用 umount 命令再次从目录层次结构中移去。这可以利用基本的 umount() 系统调用：

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

int umount(char *path);
```

参数 path 或者是被装配的磁盘分区的路径名，或者是装配文件系统的目录的路径名。这个系统调用也只能由 root 使用。

21.4 sync 系统调用

你已经知道当数据被写入一个文件时，内核(kernel)将把数据写入内存缓冲区，而不直接写入磁盘。你也已经看到可以使用 sync 命令强制将数据写到磁盘上。sync 命令使用基本的 sync()系统调用：

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

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

应该注意如果你打算使用这个系统调用,它只是对将要写到磁盘上的磁盘数据进行调度;它并不等待实际写磁盘操作的发生。这表示你不应当假定,当 sync()调用返回你的程序时,实际已经发生了磁盘写操作,而只是它们已经被调度,使得它尽可能快地发生。

事实上,你很少需要亲自调用 sync()。因为有一对称为 update 和 bdflush 的程序,大部分 Linux 发布在系统引导(boot)时自动执行这些程序。在系统运行的整个期间,这些命令一直在活动,并且它们正常每隔 30 秒(对于重要的数据间隔时间更短)用磁盘高速缓冲区的数据刷新(flush)磁盘。在这种方式下,即使机器崩溃,也只损失不多于 30 秒的有价值的工作。

不像 mount()和 umount()那样,sync()系统调用可以由任何用户使用。

21.5 显示时间

许多程序需要计量时间。这可以是当前的时钟时间,或者是进程使用了多少 CPU 时间。在 Linux 系统中,提供这两种意义下的时间计量的系统调用。

21.5.1 time 系统调用

在个人计算机的硬件中包含一个系统时钟。Linux 在引导系统(boot)时读该时钟,然后维持它自己的时钟计数器,利用时钟计数器对系统内的事件,如文件的创建或用户登录等给出时间记录。时钟值也可以利用 time()系统调用供你自己的程序使用:

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

time_t time(time_t *loc);
```

这个系统调用返回一个 time_t 值,如果参数 loc 是指向这种类型的变量的指针,则返回值也被拷贝到这个指针所指的变量中。

数据类型 time_t 实际上由 typedef 定义,可以通过 < time.h > 访问,它的基本数据类型是 long。因此,time()系统调用返回一个长型整数作为当前时钟的时间值,但是,这是一种奇怪格式,即从格林威治时间 1970 年 1 月 1 日午夜起所经过的秒数(我假定这个时钟必定在某一时刻开始...)。

有一个枝节问题是:它用一个长型整数可以记录多长时间? 如果限制在 32 位长的正整数范围内(负值对时间没有任何意义!),则它正好超过 68 年,或者说直到 2038 年初(还有充裕的时间想出一种替换办法)。

这种时间格式实际上相当有用,如果你想要解决两个事件之间(例如你的登录时间和退出系统的时间)经过了多长时间,你只要从较早时间减去较晚时间就得到两个事件之间的秒数。

但是,在你想要显示日、月、年或者当天的时间的情况下,这种时间格式就不方便。为了简化这些应用,标准库中包含一些有用的函数为你进行转换。两个这样的函数是:

```
#include  <time.h>

struct tm *gmtime(time_t *loc);
struct tm *localtime(time_t *loc);
```

这两个函数都取一个参数 loc,它是指向包含'自 1970 以来的秒数'值的变量的指针。这两个函数都返回指向一个结构的指针,该结构的域包含所有要求的信息:

```
struct tm {
        int tm_sec;
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;
        int tm_yday;
        int tm_isdst;
};
```

域 tm_hour、tm_min 和 tm_sec 以 24 小时时钟格式给出时钟时间,tm_mday 是在 1 到 31 范围内的该月的日子,tm_mon 是范围为 0 到 11 月数,一月(January)作为 0,tm_year 是自 1900 以来的年数,tm_wday 是范围为 0 到 6 的一周的日子,星期日(Sunday)作为 0,tm_yday 是范围为 0 到 365 该年的日子,一月一日(1 January)作为 0,tm_isdst 是一个标志规定夏时制是否有效(如果该信息可供使用)。

这两个函数的主要差别是 gmtime()给出它的相对于格林威治时间(Greenwich Mean Time (GMT))的时间信息,而 localtime()给出它的相对于你的本地时区的时间信息(在系统安装和配置期间设置本地时区)。

21.5.2 times 系统调用

其它可能感兴趣的是那些和进程以及它们的 CPU 使用有关的时间。这个信息可以使用 times()系统调用来得到:

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

clock_t times(struct tms *buf);
```

这里 times()的参数 buf 是指向将被该调用填入的 tms 结构的指针。这个结构的定义是:

```
struct tms {
        clock_t tms_utime;
```

```
    clock_t tms_stime;
    clock_t tms_cutime;
    clock_t tms_cstime;
}
```

一个进程的 CPU 时间被分解成两个部分:它在它自己的正文段中执行指令所化费的时间(用户时间)和通常通过系统调用在内核(kernel)中执行指令所化费的时间(系统时间)。

tms_utime 域是调用进程所使用的全部用户时间,tms_stime 是它的全部系统时间。其它两个域(tms_cutime 和 tms_cstime)是同样的,但是,这是对已成功地执行了一个 wait()调用的进程的包括所有子女进程使用时间在内的时间总数。

times()系统调用的返回值是自从最近启动系统以来所经过的时间量。

对于 Linux 系统讲,times()系统调用使用的时间单位是百分之一秒。

21.6 select 系统调用

本章最后一个系统调用是 select。在用得着的情况下,它具有非常有效地减少你的程序使用 CPU 时间的能力。

考虑下列情况。假设你希望写一个程序,它可能从终端键盘取输入,也可能从一个命名管道的读端取输入。如果能够保证从这两个源的输入将以严格的交替方式到达,使得可以先读第一个然后再读另一个。写这样的程序将是一件简单的事。但是,如果(更大的可能是)从这两个源的输入是以任意的次序到达,则就存在一个问题。因为当 read()调用发生时,如果数据未准备好,这两个输入设备都将阻塞在 read()调用上。

这表示如果输入未准备好,一个从键盘的 read()可能挂起该进程。甚至即使可能有大量从管道到达的输入,该进程也将不能对它做任何事。

对这个问题的明显解是在这两个输入设备上使用 O_NONBLOCK 标志,使得即使输入字符未准备好,它们也全都在执行 read()调用时立即返回。然后你可以写你的代码,依次反复地对这两个输入设备进行测试,当每个设备出现输入时分别处理它们。

这个方案能进行工作。但是以这种方式重复测试把一个连续的负载加到 CPU 上,除了当输入字符到达时以外,未做任何有效的工作。

select()系统调用提供这个问题的完全解,因为它允许你规定将你的进程本身挂起,而同时使内核监视所要求的一组文件描述符的任何活动。只要确认在任何被监视的文件描述符上出现活动,select()调用将返回指示该文件描述符已准备好使用的信息。

它允许为进程选出多种随机出现的输入,而不必由进程本身对输入进行测试而付出沉重的 CPU 开销。select()调用的原型是:

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

int select(int nfds, fd_set *rdfds, fd_set *wrfds, fd_set *exfds, struct timeval *
```

```
timeout);
```

其中 rdfds、wrfds 和 exfds 分别是被 select()监视的读、写和异常处理的文件描述符的集合，nfds 是需要检查的号码最高的文件描述符，参数 timeout 可以使 select()较早地返回，即使还没有文件描述符准备好。

由 rdfds、wrfds 和 exfds 所指向的 fd_sets 被该调用修改以表明那些文件描述符已准备好，并且 select()调用返回准备好的文件描述符的总数。

为了简化在 fd_set 中对相应的文件描述符位的设置、复位和测试，提供了一组宏(macros)定义如下：

```
FD_SET(fd, setptr)      设置由 setptr 所指的 fd 位，
FD_CLR(fd, setptr)      清除由 setptr 所指的 fd 位，
FD_ZERO (setptr)        清除由 setptr 所指的所有位，
FD_ISSET(fd, setptr)    测试由 setptr 所指的 fd 位。
```

将头文件 < types.h > 包含在你的代码中，就可以使用这些宏定义。

练习

1. 修改你在 socket 库实例研究的练习中所写的 chat 程序，发挥 select()系统调用的作用，而不是重复地测试所有已打开的 socket 描述符以找出已准备好读的 socket 描述符。
2. 实现你自己的标准 date 命令的版本，它以适当的格式显示当前时钟时间和日期。
3. 实现你自己标准 time 命令的版本，它取另一个命令和参数作为它自己的命令行参变量，执行给定的命令，然后显示该受控的命令在其执行期间所使用的内核(kernel)时间和用户时间。
4. 写一对简单命令，将它们设置为 setuid root，将允许普通用户在目录层次结构中的某一固定点(例如/floppy)装配(mount)和卸下(unmount)软盘。

解答

1. 为了做到这一点，你需要在 main()的无限循环的前面使用 select()系统调用。为了完全有效，必须包含由 sp 所指的 SOCKET 结构内部的 socket 描述符在内。当新的客户参加聊天(chat)和现有的客户离开时，在读文件描述符集合(rdfs)中相应的位必须设置和复位，使得 select()仅仅继续监视当前正在使用的 socket 描述符。
2. 这包括利用 time()系统调用获得自 1970 年 1 月 1 日以来以秒计算的当前的时间，然后对这个值使用一个转换函数进行转换，得到填入适当信息的 tm 结构。然后从这个结构中抽出所要求的值，并以正确的日期格式显示。
3. 实现这条命令的最容易的方法是用 fork()创建一个子女进程，它将使用 execvp()执行规定的命令，然后在双亲进程中在等待子女进程结束后，使用 times()调用得到访问子女进程的系统和用户时间信息：

```
#include < sys/times.h >
main(int argc, char **argv)
{
  struct tms tm;
```

```
    if (fork())
    {
      wait(0);
      times(&tm);
      printf("User CPU time = %d.%d secs\n", tm.tms_cutime/100, tm.tms_
          cutime%100);
      printf("System CPU time = %d.%d secs\n", tm.tms_cstime/100, tm.tms_
          cstime%100);
    }
    else
    {
        execvp(argv[1], &argv[1]);
        exit(1);
    }
}
```

4. 一个可能的解是按下列方式用适当的固定的参数调用 mount 和 umount 系统调用：

```
/* mountfd0 command - must be setuid root to work */
main()
{
  setuid(0);
  if (mount("/dev/fd0", "/floppy", "ext2", 0, 0))
    puts("mountfd0 failed!!!\n");
}

/* umountfd0 command - must be setuid root to work */
main()
{
  setuid(0);

  if (umount("/dev/fd0"))
    puts(umountfd0 failed!!!\n);
}
```

为了将该进程的实际用户识别号设置为 0，调用 setuid(0)是必要的。因为当 mount 系统调用进行检查以确保它正在被 root 用户运行时，它测试实际用户识别号而不是有效用户识别号。

第二十二章　守 护 进 程

守护进程(Daemons,发音为 dee-mon,而非 day-mon)是在后台运行而又无终端或登录 shell 和它结合在一起的进程。有许多标准的守护进程,其中的一些周期地运行来完成特定的任务(像 atrun,典型地由 cron 每五分钟执行一次),而其余的则连续地运行,等待处理某些特定的事件(像 inetd 和 lpd)。

22.1　原理

有几种启动守护进程的方法。最常用的是:

* 在引导系统时启动。在这时运行的守护进程通常在系统启动 script 的执行期间被启动,这些 script 典型地被存放在目录 /etc/rc.d 中。
* 手工地,从 shell 提示符启动。对任何具有相应的执行权限的用户可以用这种方法启动守护进程。
* 由 crond 守护进程启动。这个程序查询通常存放在 /var/spool/cron/crontabs 目录中的一组文件,它们规定了要执行的周期性任务。
* 由执行 at 命令启动。这将使一个程序在规定的日期和时间执行一次。

为了做到完全稳定可靠,一个守护进程应该能够在所有这些方式下被正确地执行。唯一问题是其中的一些启动方法使守护进程处于一种脆弱状态。它可能受到在它运行之前为它设置的环境的影响。你将要写的将普通程序(例如在 18.2 节中的 socket 服务程序)转换成守护进程的大部分代码将涉及把你的程序与这些环境影响隔离开。除此之外,准备作为守护进程使用的程序通常必须比普通的用户程序更可靠。你常常需要这样去编写程序,使得当它作为守护进程运行时,即使出现各种类型的系统错误,也不致于引起程序的崩溃。

22.2　实践

在启动守护进程的过程中,包含了比你能想象的更多的步骤,虽然它们全都可能只用极少量代码去完成。本节的其余部分将逐步介绍这一过程的更重要方面。

22.2.1　关闭文件描述符

第一项任务是关闭所有不必要的文件描述符。如果你的守护进程留下一个普通文件处于打开状态,这将阻止该文件被任何其它进程从文件系统中删除。它也阻止包含该打开文件的已装配的文件系统被卸下。在终端文件(通常是 stdin、stdout 和 stderr)的情况下,关闭不必要的连接甚至更为重要。因为当在该终端上的用户退出系统后,将执行 vhangup()系统调用,守护进程访问该终端的权利将被撤消。这表示守护进程虽有它认为处于打开状态的文件描述符,事实上它已不再能通过这些文件描述符访问该终端。

最简单的做法只是关闭所有的文件描述符,它将使守护进程和这些问题隔离开。将 close()系统调用用在没有打开的文件描述符上不存在任何问题,所以下列的代码段可以被使用:

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

for (i = 0; i<NOFILE; ++i)
  close(i);
```

符号常量 NOFILE 给出一个进程一次可以打开的文件的最大个数。

22.2.2 甩开控制终端

如果一个进程是从登录对话过程中被启动,则它将从对话过程中继承与它结合的控制终端。对于守护进程来说,其结果是它可以接收由该控制终端所产生的信号(诸如 SIGINT 和 SIGQUIT),如果这些信号不被捕获,将结束该进程。这个问题可以由守护进程忽略所有它可能忽略的信号而加以克服,但是这将阻止守护进程利用信号作为简单的进程间通信手段使用。一个较好的解决办法是使守护进程本身和控制终端分离,使得这些信号首先不传播到守护进程。

在 Linux 下这样做的一种方法是打开文件/dev/tty,并且使用 ioctl()在该文件上执行 TIOCNOTTY 命令。它使得每一个具有控制终端的进程通过文件/dev/tty 访问那个终端。

做这件事的简短的代码段如下:

```
if ((fd = open("/dev/tty", D_RDWR))>=0)
{
  ioctl(fd, TIOCNOTTY, 0);
  close(fd);
}
```

在 Linux 下,这不是使进程本身和它的控制终端分离的唯一的方法。事实上,在下一节中我们将看到做这件事情的更简单的方法。

22.2.3 脱离对话过程和进程组

进程从它的双亲进程获得它的对话过程和进程组识别号。由于它属于一个对话过程和一个进程组,一个进程将接收任何作为整体发送给该对话过程或进程组的信号。这类似于从控制终端接收信号的问题,并且实际的解决方法也是同样的,即将进程和这一环境影响分离开。

在 POSIX 中(因此也在 Linux 中)存在一个单一的系统调用,它将进程和它的当前的对话过程和进程组分离开,并且把它设置为一个新的对话过程的领头进程(leader)。这个系统调用设置对话过程识别号:

```
setsid()
```

由于一个控制终端仅可以和单一的对话过程相联,作为 setsid()调用的副作用,它也把进程和它的控制终端(如果它有控制终端的话)分离开。

setsid()的唯一问题是:只有执行它的进程不是对话过程的领头进程(leader)时,它才能发挥作用。在我们的情况下,容易假定该守护进程不是对话过程的领头进程,但是这不能保证,除非采取特殊步骤使它成为这样。你可以毫无疑问地这样说:如果一个特定的对话过程的领头进程执行了 fork()系统调用,则默认地这一子女进程应该不是对话过程的领头进程。这提供了确保执行 setsid()的守护进程不是对话过程的领头进程的机制。守护进程需要做的只是按如下所示执行 fork()调用,然后在双亲进程中执行 exit(),并且在子女进程中执行 setsid():

```
/* After this, child process is not a session leader
(在这以后,子女进程不再是对话过程的领头进程)*/
if (fork())
    /* Terminate parent process whatever its status
    (不管它的状态如何结束双亲进程)*/
    exit(0);

/* Disassociate child from session, process group and tty
(把子女进程和对话过程,进程组及 tty 分离)*/
setsid();
```

即使采取了这些措施,还没有彻底解决将守护进程和它的控制终端分离的全部问题。这是由于当一个没有控制终端的对话过程的领头进程(就像我们现在的情况)打开其本身还不是另一个对话过程的控制终端的终端设备时,该终端将自动地变成新的对话过程的控制终端。

但是注意,以这种方式获得控制终端仅可以由对话过程的领头进程完成。因此,明显的解决办法是要确保我们的进程不再是对话过程的领头进程。这可以由第二次执行 fork()及再次结束双亲进程来完成。这留下一个不属于其原始对话过程或进程组的子女进程,它没有控制终端,而且现在也不能再重新获得一个控制终端。所要求的代码段如下所示:

```
/* Start up new session, to lose old session and process group
(启动新的对话过程,脱离旧的对话过程和进程组)*/
if (fork())
    exit(0);

/* Disassociate process group and controlling terminal
(把进程组和控制终端分离)*/
setsid();

/* Become a NON-session leader so that a (变成一个非对话过程的领头进程)*/
/* control terminal can't be reacquired (使它不可能重新获得控制终端)*/
if (fork())
    exit(0);
```

22.2.4 改变工作目录

在进程存在期间,内核(kernel)保存系统中任何进程打开的当前工作目录。在正常情况下,这不成为一个问题。但是如果该进程在已装配的文件系统中有一个当前工作目录,则该文件系统被标志为'在使用(in use)'状态,而且它不可以被卸下。为了允许系统超级用户(superuser)卸下文件系统,守护进程可以执行:

```
chdir("/");
```

为了正确地进行操作,一个守护进程也可能必须用 chdir()改变到一个特定的目录。样本 socket 服务程序就是一个恰当的例子。其中服务程序可以提供的所有文件都放在守护进程的当前工作目录中。在这种情况下,你不得不接收限制,或者由系统管理员在根(root)文件系统中提供守护进程可以运行的一个位置。将当前工作目录改变到根(root)目录可能出现一个问题。如果出现守护进程结束运行和卸出一个核心(core)文件的情况,内核将试图打开在当前目录中的文件。如果这是根目录,则操作将失败,除非守护进程正在用超级用户的权限运行。为了克服这个问题,下面将使用 /tmp 作为当前工作目录。通常它只是根文件系统的一个子目录,并且所有进程都有写的权限:

```
chdir("/tmp");
```

22.2.5 重新设置文件的创建掩码

进程一开始,它继承它的双亲进程的文件创建掩码。典型地,它具有 022 值,表示由守护进程创建的任何文件将不把写权限给组或其它用户,不管守护进程本身规定什么权限位。依赖于守护进程的性质,这个操作可能或不可能成为问题。但是,将下列行包含在你的代码中,取消由双亲进程设置的文件创建掩码值产生的效果将是一件简单的事情,无论它当前是什么值:

```
umask(0);
```

即使到现在,仍然存在从双亲进程继承的环境中的一些特征。它们可以使守护进程产生某些问题。例如,nice()进程设置的优先权,或者用 alarm()设置的闹钟信号的留下的时间。但是,虽然这些事情是可能的,它应该是相当恶意的双亲进程给粗心大意的守护进程设置的陷阱。

22.2.6 处理 SIGCHLD 信号

有时守护进程被写成创建子女进程。将样本 socket 服务程序作为守护进程将演示这类代码。在此情况下,双亲进程保持循环接收更多客户连接,而子女进程服务客户的请求。注意,双亲进程不执行 wait()系统调用等待它的子女进程进程的结束。默认地,在这个情况下的子女进程将变成 zombie,直到将来某个进程(很可能是 init)等待它们为止。在这些情况下 zombie 的创建浪费系统资源。在 Linux 下有几种方法可以绕过这个问题。最简单的是按如下方式将 SIGCHLD 信号的操作设置为 SIG_IGN:

```
signal(SIGCHLD, SIG_IGN);
```

这是 SIGCHLD 信号的特殊的特征，它告诉内核(kernel)不从调用进程的子女进程中产生 zombie。

练习

1. 创建服务器的守护进程的伪码程序，它将允许在一个 socket 上并发地服务多个网络客户请求
2. 写一个新命令(launch)的程序，它可以用来将另一个程序作为守护进程投入运行。例如：

```
$ launch myprog param1 param2
```

在设置它的环境之后，将执行命令 myprog param1 param2，使得它作为一个守护进程运行。它将允许你以这样的方式写命令，使得你可以在你登录后作为普通命令运行它们，或者作为守护进程运行，即使在你退出系统之后它们将继续执行。

解答

1. 这实际上是在 22.2.6 节中所建议的方案，它给出所要求答案的暗示。你要做的是使该进程成为一个守护进程，设置 socket，然后进入一个循环接收客户程序的连接请求。每当你获得一个客户连接请求时，你用 fork 创建一个子女进程来处理它，同时双亲进程进入下一循环接收另一个客户连接请求。解答的伪码可能是：

```
Setup current process as a daemon (fork, setsid, fork etc.)
Create a socket on a well known port
while (true)
  Get a client connection
  if (fork() == 0)
    Close parent's copy of client connection
  else
    In child, deal with client request
    exit child process
  endif
endwhile
```

事实上，在下一章中你将看到这个解用在极小守护进程实例研究中。

2. 对这个问题的答案只要求你设置你的 launch 命令作为一个守护进程运行，然后使用 exec()将在守护进程中运行的程序改变为在命令行上给定的程序。这可以按如下方式来实现：

```
main(int argc, char **argv)
{
   Setup process as a daemon (fork, setsid, fork etc.)

   execvp(argv[1], &argv[1]);
}
```

第二十三章　极小的守护进程——实例研究

在讨论了创建守护进程(Daemons)的原理后,现在我们回到18.2节中的样本socket服务程序,并领会如何使它成为一个真正的守护进程,即使在你退出机器后,它将继续运行并且提供服务。

23.1　守护进程清单

我们对18.2节中的样本socket服务程序进行了修改,使它能作为健全的守护进程运行。下面是它的完整的清单。你将看到原来的服务程序代码本身已经作了一些修改,并增加了setup_daemon()函数。所有这些将在该清单后详细加以解释:

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
THE SOCKET SERVER DAEMON(SOCKET 服务程序的守护进程)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include  <stdio.h>
#include  <string.h>
#include  <signal.h>
#include  <fcntl.h>
#include  <setjmp.h>
#include  <sys/param.h>
#include  "socklib.h"

/* Server's communication port number  (服务程序的通信端口号)*/
#define PORT 2121

/* Values for the next_action parameter to fatal() (fatal()函数的 next_action 参数值)*/
#define  NA_EXIT  0
#define  NA_RESTART  1

/* Global environment store for setjmp/longjmp  (为 setjmp/longjmp 存储全局环境)
 */
jmp_buf env;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
MAIN    A very simple file transfer daemon,which sets up a concurrent socket server
        on the host machine on port number PORT to transfer any files from the current
        directory on request
```

```
    (一个非常简单的文件传输守护进程,它在主机的 PORT 端口号上创立并发的 socket 服务
    程序,按照请求从当前目录中传输任何文件。)
*****************************************************/
main(void)
{
  SOCKET * sp;
  int sd;

  /* Rearrange operating environment for daemon working
     (为守护进程工作重新安排操作环境) */
  setup_daemon();
  /* Set SIGCHLD for no zombies (为了无 zombie 设置 SIGCHLD)*/
  signal(SIGCHLD, SIG_IGN);

  /* Create server's socket (创建服务程序的 socket)*/
  if ((sp = sopen()) == 0)
    /* Exit server if this fails (如果失败,退出服务程序)*/
    fatal("sopen()", NA_EXIT);

  /* Setup the NA_RESTART (post error) entry point (创建 NA_RESTART 入口点)*/
  setjmp(env);

  /* Repeatedly accept and service client requests (重复地接收和服务客户的请求)*/
  for (;;)
  {
    /* Connect to client or restart on failure (连接客户或者在失败时重新启动)*/
    if ((sd = sserver(sp, PORT, S_DELAY)) == -1)
      fatal("sserver()", NA_RESTART);

    /* Fork() a child process (用 Fork()创建子女进程) */
    switch (fork())
    {
    /* Deal with client request in child (在子女进程中处理客户请求)*/
    case 0:
      do_service(sd);

    /* Deal with fork() failure (处理 fork()失败)*/
    case -1:
      close(sd);
      /* Restart for next client (重新启动下一个客户)*/
      fatal("fork()", NA_RESTART);
    }

    /* In parent, close client and loop for next connection
```

```
        (在双亲进程中,关闭客户并进入下一个连接循环)*/
    close(sd);
  }
}

/*********************************************
SETUP_DAEMON - This funtion is responsible for modifying the current working env-
        ironment of the process to that required by a daemon. I.e. it closes unnece-
        ssary file descriptions, disassociates itself from the current controlling
        terminal, session and process group and modifies the current file creation
        mask.
        (这个函数负责将进程的当前工作环境修改为守护进程所要求的工作环境。即它关闭不必要
        的文件描述符,将它本身和当前的控制终端、对话过程以及进程组分离开,并且修改当前的文
        件创建掩码。)
*********************************************/
setup_daemon(void)
{
  int i;

  /* Close all open file descriptors (关闭所有打开的文件描述符)*/
  for (i = 0; i < NOFILE; ++i)
    close(i);

  /* 1st fork() call(第一次 fork()调用)*/
  switch (fork())
  {
  /* Error in 1st fork()(在第一次 fork()调用中的错误) */
  case -1:
    fatal("setup_daemon(), 1st fork()", NA_EXIT);

  /* Exit 1st parent(退出第一个双亲进程)*/
  default:
    exit(0);

  /* continue in 1st child (2nd parent) (在第一个子女进程(第二个双亲进程)中继续)*/
  case 0:
    /* Start new session exit() on error (启动新的对话过程,在错误时调用 exit())*/
    if (setsid() == -1)
      fatal("setup_daemon(), setsid()", NA_EXIT);

    /* 2nd fork() call(第二次 fork()调用) */
    switch (fork())
    {
    /* Error in 2nd fork()(在第二次 fork()中出错)*/
```

```
    case -1:
      fatal("setup_daemon () , 2nd fork()", NA_EXIT);

    /* Exit 2nd parent(退出第二个双亲进程) */
    default:
      exit(0);

    /* Continue in 2nd child(在第二个子女进程中继续) */
    case 0:
      /* Reset file creation mask for ERROR.LOG(为 ERROR.LOG 复位文件创建掩码) */
      umask(0) ;
      /* and return with daemon set up(创建守护进程后返回) */
      return;
    }
  }
}

/*****************************************************
FATAL - This routine is called when an error condition is returned to the daemon,
        usually from one of the system calls it makes. The function first prints the
        messagepointed to by <text> and then either terminates the process or re-
        starts it with longjmp(), depending on the value of the <next_action>
        parameter.
        (通常,当从守护进程调用的系统调用中返回错误条件时,调用这个子程序。这个函数首先
        打印由<text>所指的信息,然后根据参数<next_action>的值,或者终止该进程,或者
        用 longjmp() 重新启动它。)
*****************************************************/
fatal(char *text, int next_action)
{
  FILE *fp;

  /* Open the error log file(打开错误日志文件) */
  if ((fp= fopen("ERROR.LOG", "a"))==0)
    /* exit() on failure here, with error unreported...(这里在失败时退出,不报告错误) */
    exit(1);

  /* Write the error message(显示错误信息) */
  fprintf(fp, "Error in %s\n", text);
  fclose(fp);

  if (next_action==NA_RESTART)
    /* Restart calling process(重新启动调用进程) */
    longjmp(env, 1);
  else
```

```
  /* Terminate calling process(结束调用进程) */
  exit(1);

/*****************************************************************
DO_SERVICE - This function is called in the child process that is created when a
      client makes a successful connection. Its task is to service the client's
      requestand supervise the required file transfer. The parameter <sd> is
      the socket descriptor to use for the communication.
      (在子女进程中调用这个函数。这个子女进程是当与一个客户进程作出成功的连接时创建的。
      它的任务是服务客户的请求和监督所请求的文件传输。参数<sd>是用于通信的 socket
      描述符。
*****************************************************************/
do_service(int sd)
{
  int  i,  fd;
  char  *name;
  char  c,  namlen;

  /* Get length of file name  (获得文件名的长度) */
  if (read(sd, &namlen, 1) != 1)
    fatal("namlen read()", NA_EXIT);

  /* Space for name + " open()"  (为 name + " open()"分配空间) */
  if ((name = (char *)malloc(namlen + 8) ) == 0)
    fatal("malloc()", NA_EXIT);

  /* Get required file name into name[]  (取得所要的文件名送入 name[]数组) */
  for (i = 0; i < namlen; ++i)
    if (read(sd, &name[i], 1)! = 1)
      fatal("file name read()", NA_EXIT);

  /* Make name[] a string  (使 name[]成为一字符串) */
  name[i] = '\0';

  /* Check name[] is in current directory  (检查 name[]是否在当前目录中) */
  if (strchr(name, '/'))
    fatal("illegal file name", NA_EXIT);

  /* Open specified file(打开指定的文件) */
  if ((fd = open(name, O_RDONLY)) == -1)
    fatal(strcat(name, " open()"), NA_EXIT) ;

  /* Transfer contents to client  (把文件内容传输给客户) */
  while ((i = read(fd, &c, 1))! = 0)
```

```
    if (i == -1)
      fatal("file read()", NA_EXIT);
    else if (write(sd, &c, 1)! =1)
      fatal("write()", NA_EXIT);

  /* Terminate child and with it the connection to this client (结束子女进程和对这个
                                                              客户的连接) */

  exit(0);
}
```

23.2 socket 服务程序增加的内容

对这个程序最明显的改变是增加了 setup_daemon()函数,它作为 main()函数的第一个语句被调用。这个函数用来干什么并没有使人意想不到的地方,但是它编程的方式不同于前面我们所讨论的构思方法。

第一项任务是关闭所有的文件描述符。不管它们是否打开都这样做。然后,下一段代码将该守护进程和它的控制终端、对话过程以及进程组分离开,并且阻止它将来重新获得另一个终端的可能。这些用通常的 fork(),setsid(),fork()序列完成,但嵌入到一对 switch()结构中,使得它易于处理从 fork()调用返回的任何可能的错误。fork()调用返回三个不同的值:-1 转向出错处理,0 转向子女进程,非零值转向双亲进程由 default:case 处理。最后,在第二个子女进程中调用 umask()将文件创建掩码设置为 0。这是因为在出现任何严重的错误时,该守护进程可能需要创建 ERRDR.LOG 文件,如果没有这个调用,当前的 umask()值可能是不适当的。

23.3 Socket 服务程序的改变

对原有的样本 Socket 服务程序的所有其它改变直接和守护进程能够处理由系统调用和库调用返回的错误有关。这由重新启动守护进程的主循环来完成,而不仅仅是当出现任何错误时结束该守护进程。这种思想已经用在 fatal()函数中增加额外的参数来实现。在打印它的错误信息后应该做什么由 next_action 的值来决定。如果它取 NA_EXIT 值,结束调用的进程;或者如果它取 NA_RESTART 值,就使用 setjmp()和 longjmp()库子程序以非局部转移(non-local jump)方式跳回服务程序的主循环开始处。事实上,这个重新启动功能仅用于在双亲进程的主循环内检测到的错误。如果在为客户服务的子女进程内检测到任何错误,则同前面一样允许结束这个进程。

练习

1. 修改在 socket 库实例研究的练习中你所写的 chat 程序,使你的服务程序成为一个守护进程。
2. 增加一个内部命令到极小的 shell 中去,允许 shell 的用户为系统管理员存储简单的信息。这个新命令应该这样进行操作:取信息,把用户的登录名加在它的前面,并发送这个完整的字符串给已被创建的称为/tmp/messfifo 的命名管道中。然后写一个守护进程,它连续地监视命名管道的读端,并把它接收的任何信息和一个指示写信息的时间标记一起传输给/tmp/sysmess 信息文件。如果你在该机器上有足够的权力,你应该设置这个系统使得各个用户的 shell 不能直接写到这个信息文件中。

解答

1. 这只要把守护进程的启动代码加到前面的练习的开头。也可能利用机会使 chat 程序的错误检查更健全和功能更完善。例如,你可能喜欢使用户遵循某个登录协议,使他们的登录名可以被加到他们的所有信息中以便其他 chat 用户辨认信息源。
2. 在极小的 shell 中增加一个 message 命令,只包括在 builtin()中增加一个 do_message()函数调用。do_message()函数将从系统中提取用户的登录名(查阅 getenv("LOGNAME")或 getpwuid()),从 line[]数组中提取信息,并且把这些写到命名管道中。该守护进程将打开命名管道供读访问,在信息到达时从管道中取出信息,用 time()调用读时钟时间 ,并将信息和时间写到信息文件中。当你有 root 权限时,你可以使守护进程的程序,管道, 和信息文件为 root 所拥有。然后,给守护进程的程序对管道有读访问权和对信息文件有写访问权。而其他用户对管道有写访问权,但没有任何访问信息文件的权限。

第四篇

设备驱动程序

第二十四章　设备驱动程序基础

系统调用是操作系统内核和应用程序之间的接口，而设备驱动程序则是操作系统内核和机器硬件之间的接口。

为了使得从设备接受输入和将输出送到设备就像从文件接受输入和将输出送到文件一样，付出了巨大的努力。使得它们都能使用 open( ) ，close( ), read( ) 和 write( ) 系统调用。正是这些努的结果，使设备驱动程序本身具有文件的外在特征。

24.1　概述

操作系统内核需要访问两类主要设备。简单的字符型设备（下称字符设备），如打印机，键盘等；和数据块型设备（下称块设备），如软盘，硬盘，光盘等。从名称使人想到，字符设备在单个字符的基础上接收和发送数据。为了改进传送数据的速度和效率，块设备则在整个数据缓冲区填满时才一起传送数据。

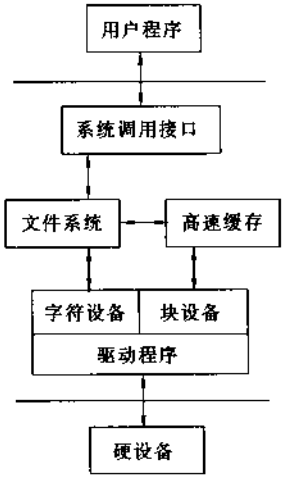

图 24.1　用户程序通过设备驱动程序访问硬件

与此对应，有两类设备驱动程序。分别称为字符设备驱动程序和块设备驱动程序。两者的主要差别在于：与字符设备有关的系统调用几乎直接和驱动程序的内部功能结合在一起。而读、写数据块设备则主要和快速缓冲存储区打交道。只在需要完成实际输入/输出时，才用到块设备驱动程序。图 24.1 将这两种功能结合在一起。

一般说来，每类需要加以控制的设备要有不同的设备驱动程序。如需要有控制软盘的设备驱动程序，它能和所有标准的不同容量的软盘打交道。事实上，如果系统中装了不止一个软盘驱动器，将用同一个设备驱动程序来控制它们。

Linux 设备驱动程序的主要功能有：

- 对设备进行初始化；
- 使设备投入运行和退出服务；
- 从设备接收数据并将它们送回内核；
- 将数据从内核送到设备；
- 检测和处理设备出现的错误。

为了使设备的存取能像文件的存取一样进行处理，所有的设备在目录层次结构中的适当位置必须有对应的文件名称。这样，才可能对它们使用 open( )和 close( )等系统调用。

这些文件分别称为字符设备特殊文件和块设备特殊文件。按习惯，它们都放在 /dev 目录下。下面是存在这一目录中的一些设备特殊文件：

```
brw-rw----   1 root   floppy   2,  0 Jul 18  1994 fd0
brw-rw----   1 root   floppy   2,  1 Jul 18  1994 fd1
brw-rw----   1 root   disk     3,  0 Jul 18  1994 hda
brw-rw----   1 root   disk     3,  1 Jul 18  1994 hda1
brw-rw----   1 root   disk     3,  2 Jul 18  1994 hda2
brw-rw----   1 root   disk     3,  3 Jul 18  1994 hda3
brw-rw----   1 root   disk     3,  4 Jul 18  1994 hda4
crw-rw----   1 root   daemon   6,  0 Jul 18  1994 lp0
crw-rw-rw-   1 root   sys      1,  3 Jul 18  1994 null
crw-rw-rw-   1 root   tty      5,  0 Jul 18  1994 tty
crw--w--w-   1 pc     book     4,  0 Jul 18  1994 tty0
crw--w--w-   1 pc     book     4,  1 Aug 30 15:16 tty1
```

注意:每行的第一个字符 c 或 b 给出文件的类型,分别表示字符或块设备特殊文件。同时注意,在日期字段前面有一对用逗号分开的数字。普通文件用它来表示文件的容量。作为特殊文件,两个数字中的第一个称为主设备号(major device number),第二个称为辅助设备号(minor device number)。

字符设备和块设备的主设备号用来指定特定的设备驱动程序,调用它的内部功能来处理这类设备的输入/输出请求。辅助设备号由设备驱动程序的子程序使用。主要用来确定当前的输入/输出设备请求与那一个具体设备有关。例如,在上面的清单中,软盘 0 和软盘 1(fd0 和 fd1)的主设备号都是 2。当发生对 /dev/fd0 或 /dev/fd1 的输入/输出请求时,用它来确定要调用的设备驱动程序。在驱动程序内部则用辅助设备号来区分具体设备(0 对应 fd0,1 对应 fd1)。

24.1.1 mknod 命令

用 mknod 命令(或 mknod( )系统调用)建立设备特殊文件。只有 root 帐号能建立这些文件。命令的格式为:

```
mknod  filename  type  major  minor
```

其中 filename 是待建立的设备特殊文件的路径名;type 的值取 c 或 b,分别对应字符设备或块设备特殊文件。major 和 minor 是与这一文件结合的主设备号和辅助设备号。例如:

```
# mknod  /dev/mytty1  c  4  1
```

用主设备号 4 和辅助设备号 1 建立字符设备特殊文件 /dev/mytty1。由于这一组合的主设备号和辅助设备号已经存在(dev/tty1),上面命令所做的只是建立能访问同一设备的备用设备特殊文件。

24.2 设备驱动程序子程序

当你试图打开(open( ))一个设备特殊文件时,将导致这一设备特殊文件的 open( )函数被

调用。因为每个设备驱动程序只是一组用来完成特定任务的函数的集合。每个驱动程序都有一个称为 file_operation 的数据结构，其中包含指向驱动程序内部大多数函数的指针。

当引导系统时，内核调用每个驱动程序的初始化函数。它的任务之一是将这一设备驱动程序使用的主设备号通知内核。同时，初始化函数还将驱动程序中的函数地址结构的指针送给内核。

file_operation 结构中包含许多元素，大多数驱动程序只利用其中的一部分。表示函数指针集合的完整结构如下：

```
Struct file_operations {
    int (*lseek) ();
    int (*read) ();
    int (*write) ();
    int (*readdir) ();
    int (*select) ();
    int (*ioctl) ();
    int (*mmap) ();
    int (*open) ();
    void (*release) ();
    int (*fsync) ();
    int (*fasync) ();
    int (*check_media_change) ();
    int (*revalidate) ();
};
```

其中有 open( )，read( )，write( ) 函数的入口点。其他入口点现在尚不明显。将在用到它们时详细说明这些入口点。

内核中有两张表。一张表用于字符设备驱动程序，另一张表用于块设备驱动程序。这两张表用来保存指向 file_operation 结构的指针，设备驱动程序内部的函数地址就保存在这一结构中。内核用主设备号作为索引访问 file_operation 结构，因而能访问驱动程序内的子程序地址。

24.3 设备驱动程序原理

到此为止，我们已经看到操作系统内核如何调用特定的设备驱动程序的内部函数。下面将讨论设备驱动程序的其他方面，包括它们与调用它们的用户进程之间的关系等。

当一个进程实际在 CPU 上运行时，Linux 将它作为当前进程对待。在当前进程执行系统调用时，可能导致执行驱动程序内部的函数。如果在驱动程序的子程序内部，当前进程的环境仍然可以使用。将数据从设备驱动程序的缓冲区复制到进程的缓冲区或者相反成为一项相对比较简单的任务。

有时候，调用的设备驱动程序函数要求从硬件（如磁盘驱动器）进行输入和输出。硬件操作速度一般要比 CPU 速度慢几个数量级。在这种情况下，如果 CPU 等待硬件完成输入/输出，将大大减慢整个系统的速度。事实上，CPU 并不等待硬件完成输入/输出。内核调度输入/输

出继续进行，而将当前进程挂起。在它等待输入/输出时，允许其他进程完成一些有用的工作。

很明显，需要提供一种机制。当输入/输出实际发生时，恢复挂起的进程的运行。这种机制称为中断。中断是由硬件的事件引起的。中断送给内核，而一般信号(signal)则送给用户进程。和信号一样，中断也能被捕捉，并导致执行特殊的服务子程序。用来处理引起中断的状态。

当进程等待的输入/输出发生了，将产生一个中断。并将执行有关的中断服务子程序。中断服务子程序的任务之一是恢复等待输入/输出完成而挂起的进程，使它继续运行。

注意，由于 CPU 转去执行其他进程，所以当中断发生和执行服务子程序时(严格讲，这是异步事件)，内核并没有在挂起的进程的环境中执行。这表示中断服务子程序现在不能马上去完成任何要用到挂起的进程的环境的任何操作，因为挂起的进程并不立即投入运行。

写设备驱动程序时，还要考虑驱动程序是作为操作系统整体的一部分运行的。在内核中没有使用抢先调度。这说明内核不像普通用户进程，内核中没有一种机制来强制一个内核子程序放弃对 CPU 的控制。结束内核程序的唯一办法是它完成功能后返回，或者挂起它们为之服务的进程，自愿放弃对 CPU 的控制，让其他进程使用。

将程序加到 Linux 内核中时，Linux 完全相信你的新程序。这表示如果程序进入死循环，等于将内核挂起，直到牵涉到其他进程为止。即使内核程序逻辑上是正确的，如果它在使用 CPU，内存或硬件的效率不高，也将导致系统性能的下降。

在我们讨论字符设备驱动程序之前，这些主要事情应牢记心头。

第二十五章　字符设备驱动程序

在本章中将讨论字符设备驱动程序的基础知识。我们先讨论字符设备驱动程序，因为写简单的字符设备驱动程序要比写块设备驱动程序容易得多。当你为新设备写串行设备驱动程序时，不管是什么类型，它们的复杂程度都差不多。由于串行设备驱动程序的操作和硬件关系密切，开始，并不特别适合作为例子。下面我们将更多地说明原理，而尽少涉及硬件的具体特点。

设备驱动程序只是一些完成不同任务的函数的集合。每个驱动程序都有一组实质上相同的函数。所以要有统一的命名规则，防止驱动程序之间名称的不协调。一条简单的规则使所有的名称都成为唯一的。只要为每个设备驱动程序选择简单唯一的前缀就能做到这一点。这个前缀将加到驱动程序中所有函数名称的前面。这就表示，虽然大多数设备驱动程序都提供open( )函数，每个驱动程序的前缀将使所有 open( ) 函数的名称在内核中是唯一的。

25.1　登记字符设备

在引导系统时，每个设备驱动程序都有机会对硬件和对它自身进行初始化。字符设备驱动程序用下面的办法实现这一点。每个驱动程序内部都设有初始化函数(init( ))，并将这一函数的调用放在内核的 chr_dev_init( )函数中。后者包含在下列文件中：

```
/usr/src/linux/drivers/char/mem.c
```

假定内核源码放在默认的 /usr/src/linux 目录下。

假定设备驱动程序的名称的前缀为 tdd_，那么它的 init( )函数应该称为 tdd_init( )。1.1.32版以前的内核，需要将下列行：

```
mem_start = tdd_init(mem_start);
```

放到 chr_dev_init( ) 函数的末尾，正好在 return mem_start; 行的前面。

1.1.33 版以后的内核，应将下列行：

```
tdd_init( );
```

加到 chr_dev_init( ) 函数的末尾，在 return 0; 行的前面。

驱动程序的 init( ) 函数的主要功能之一是在内核中登记设备驱动程序。包括：告诉内核这一驱动程序使用的主设备号，同时给内核提供指向驱动程序的 file_operation 结构的指针。内核将这一指针放在字符设备驱动程序表的相应表目中，通过调用下列的内核函数完成登记：

```
register_chrdev(major, name, file_op)
```

其中 major 是这一驱动程序使用的主设备号（或者用 0 表示，由内核分配空闲的主设备号）；name 是字符串，它给出驱动程序的名称；而 file_op 则是指向 file_operation 结构的指针。

如果登记失败，register_chrdev( ) 函数的返回值为负。如果登记成功，而且 major 的值为 0，函数将返回内核分配的主设备号。如果指定了主设备号，而且登记成功，返回值则为 0。

在下一节将看到，1.1.32 版以前的内核，驱动程序 init( ) 函数的返回值（mem_start）可以用来分配驱动程序在执行时使用的工作内存。

25.2 工作内存

设备驱动程序在执行时经常需要工作内存保存临时数据，工作内存可以从几个字节（只用来保存某些内部变量）到许多数据缓冲区（特别是，当若干设备特殊文件同时打开时，保存传送中的大量数据）。

在 Linux 系统中，驱动程序可以使用几种不同的数据缓冲技术。每种技术都有它自己的优点和弱点。

25.2.1 专用缓冲区

给设备驱动程序分配内存的最简单办法是在它的源程序中按所需的字节数加以说明。例如，由于某种原因你需要 800 字节的缓冲区用作工作内存，可以在源程序中说明数组：

```
static char buffer[800];
```

在任何函数外面使用关键字 static 表示缓冲区只能由同一文件中的代码所使用（即只适用于这一设备驱动程序内部）。

像上面那样使用专用缓冲区有几个缺点：

- 用这种方法分配的内存只能在驱动程序内部使用。
- 如果推测的容量比实际需要的容量少，无法增加工作内存的容量。
- 即使你不使用它，如硬设备没有连上，这块内存还是分配给你使用。

25.2.2 mem_start

上述问题中的最后一个问题可以用另一种内存分配方法（mem_start）加以克服。这是在引导 Linux 系统时使用的特殊的内存分配方法。在调用设备驱动程序的 init( )函数时使用这一方法。但 1.1.33 版以后的内核，不再提供这项功能。

mem_start 的值是指向可用内存起点的指针。这个值作为参数传送给你的设备驱动程序的 init( )函数。当 init( )函数结束时，期望返回同样的东西——可用内存起点的指针。这给 init( )函数一次机会，留下部分系统内存供驱动程序使用。只要将作为参数收到的 mem_start 值加上所需的内存容量即可。然后将新值返回给内核。

现在像下面那样进行这一过程。当 init( )函数成功地登记了设备驱动程序和主设备号以

后,它测试硬件是否存在。如果硬件存在,将驱动程序所需的缓冲区字节数和 mem_start 的值相加,然后将新的 mem_start 值返回给内核。从旧的 mem_start 地址开始,按所需容量留下的内存块将供这一设备驱动程序专用。

与直接保留专用内存的方法相比,这一方法的明显优点是:如果硬件不存在,或者驱动程序登记不成功就不分配内存块。这一结果一直保留到下次引导系统时为止。所以每次引导系统时,结果可能不一样。

使用这一技术仍有缺点。内存分配仍然是一次性操作,在初始分配的内存块以外,不能再增加内存。还有,即使驱动程序不再使用它,它也不能被释放。

25.2.3 kmalloc 函数

为克服这些问题,需要使用内核的动态分配内存的方案。对用户进程讲,这些功能由 malloc( ) 和 free( ) 库函数提供。在内核中,这些功能由 kmalloc( ) 和 kfree( ) 库函数提供。kmalloc( ) 函数的典型形式为:

```
void  *kmalloc(size t  size, int  priority);
```

其中 size 是你要分配的内存容量的字节数。返回值是指向已分配的内存块的起始地址的指针。在出错时,返回值为 0。

对 priority 参数要作一点解释。当申请特定的内存块时,kmalloc( ) 函数可能不能立即满足你的要求。如果准备让驱动程序在内存分配过程中休眠等待,kmalloc( ) 函数通过将一些内存页重新安排到交换区,最终总能满足你的要求。

如果准备在必要时进行等待,那么 priority 的值应该取 GFP_KERNEL。在不能接受不定长时间等待的情况下,priority 的值应该用 GFP_ATOMIC。在这种情况下,如果 kmalloc( ) 函数不能立即满足你的请求,它将立即返回 0 值。

在中断服务子程序内部调用 kmalloc( ) 时,必须用 GFP_ATOMIC 参数作为 priority 的值,这是因为作为一般规则,不能让中断服务子程序处于休眠状态。

要确保 kmalloc( ) 函数分配的内存块可以进行 DMA(直接内存存取)传送时,使用另一个选项。对许多机器讲,这不是一个问题,但在某些个人计算机上,DMA 传送只能在内存开始的 16MB 的范围内进行。为保证这一点不成问题,要使用 OR GFP_DMA 作为 priority 的值,kmalloc( ) 函数将送回可用于 DMA 传送的内存块。

使用 kmalloc( ) 函数最多能分配 128Kb 的内存块(由于管理的开销,实际分配的块要稍小些)。

用 kmalloc( )函数分配的内存块可以用 kfee( ) 函数释放,它的典型形式为:

```
viod  kfree_s(void  *ptr,  int  size);
```

其中 ptr 是 kmalloc( ) 函数返回的指针值,而 size 则是要释放的内存块以字节计的容量。size 的值必须和传送给 kmalloc( ) 函数的值相同。如果由于某种原因不知道内存块的容量,可以用 0 作为 size 的值调用 kfree_s( ) 函数。在这种情况下,它将用内部搜索功能找出内存块容量的大小。

在 size 的值为 0 时，也可以用 kfree( )代替 kfree_s( )函数。这是定义在 < linux/malloc.h > 中的 Linux 宏命令：

```
#define  kfree(n)  kfree_s( (n), 0)
```

25.2.4 vmalloc

由于某些硬件应用的需求，kmalloc( ) 函数分配的内存块实际上是连续的。当这一要求可以放宽时，内存分配工作可以用新的内存分配函数来取代，这个新函数称为 vmalloc( )：

```
void  * vmalloc(unsigned  long  size);
```

其中 size 是所要的内存块以字节计的容量。函数返回指向分配的内存块的指针，或者在出错时返回 0 值。

vmalloc( ) 函数分配的内存块不能用于 DMA 传送，也不能从中断服务子程序内部调用 vmalloc( ) 函数。

任何用 vmalloc( ) 分配的内存块可以用 vfree( ) 释放给系统：

```
void  vfree (void  * ptr);
```

其中 ptr 是 vmalloc()函数返回的指针值。

你能看出后面两种内存分配技术克服了专用缓冲区和 mem_start 分配技术的所有缺点，应该将它们优先用于你写的任何重要的设备驱动程序中。

25.3 基本入口点

我们已经看到如何在内核中登记字符设备驱动程序和如何分配内存供驱动程序内部使用。现在我们将注意力转向进入驱动程序的主要入口点。这些入口点地址包含在驱动程序的 file_operation 结构中。

用于字符设备或文件的主要系统调用有：open( )，close( )，read( )，write( )和 ioctl( )。如果将这些系统调用用于你的驱动程序，将导致相应的驱动程序的内部函数被调用。

25.3.1 open 函数

当用户进程对属于这一驱动程序的设备特殊文件执行 open( )系统调用时，将导致驱动程序的 open( )函数被调用，open( )函数的典型形式为：

```
int open(struct inode * inode, struct file * file);
```

其中 inode 参数是指向被访问的设备特殊文件的 inode(索引结点)结构的指针，而 file 则是指向这一设备的文件结构的指针。

open( )函数的典型操作包括：

·确定有关的硬件（如果有的话）可供使用而且已经联机。
·验证在 MINOR(inode -> i_rdev)中的辅助设备号是有效的（即在要求的范围内）。
·如果每次只允许一个进程打开设备，那么 open( ) 函数应测试和设置驱动程序的设备忙标志，如果这一标志已设置，则返回 -EBUSY。
·如果出现任何类型的错误，返回负值。
·成功时返回 0 值。

25.3.2 release 函数

当最后一个打开设备的用户进程执行 close( )系统调用时，才调用驱动程序的 release( )函数，它的典型形式为：

```
void release(struct inode * inode, struct file * file);
```

其中 inode 是指向设备特殊文件的索引结点结构的指针。而 file 是指向这一设备的文件结构的指针。

release( )函数用来完成下列功能：

·如果有未结束的输入/输出在进行，对它们完成清理操作。
·如果需要，释放硬件资源。
·对 open( ) 函数设置的任何排他访问标志进行复位。

25.3.3 read 函数

当对属于字符设备驱动程序的设备特殊文件执行 read( )系统调用时，将调用驱动程序的 read( )函数：

```
void read (struct inode *inode, struct file * file, char * buf, int count);
```

其中 inode 是指向设备特殊文件的索引结点结构的指针。file 是指向这一设备的文件结构的指针。buf 是指向用户空间的缓冲区的指针，它的地址由用户进程送给 read( )系统调用。而 count 则是用户进程要求读取的字节数，同样由用户进程送给 read( )系统调用。

read( )函数的功能是按要求的字节数 count 将数据从硬设备（如果存在的话）或已分配的内核内存中复制到用户内存区中由 buf 指定的缓冲区。

当完成复制任务时，必须注意 buf 中的指针是用户内存区的一个地址，这是不能直接从内核中使用的地址。为了正确完成复制工作，你必须使用专为这项用途提供的特殊复制函数。这些函数定义在 < asm/segment.h > 中：

```
void put_user_byte(char data_byte, char *u_addr);
void put_user_word(short data_word, short *u_addr);
void put_user_long(long data_long, long *u_addr);
void memcpy_tofs(void *u_addr, void *k_addr, unsigned long cnt);
```

对 put_user...( )类函数讲,其中 data_byte,data_word 和 data_long 分别表示要复制到用户空间地址 u_addr 的的字符,字和长整数值。对 memcpy_tofs( )函数讲,从内核空间地址 k_addr 复制 cnt 个字节到用户空间地址 u_addr。

为了与 0.99 版以下的旧内核向下兼容,提供了另外 3 个函数:

```
void put_fs_byte(char data_byte, char *u_addr);
void put_fs_word(short data_word, short *u_addr);
void put_fs_long(long data_long, long *u_addr);
```

这些函数与前面的 prt_usr_XXX 函数功能相同。

此外,如果你注意使用的指针类型,所有这些函数调用在新的内核中(高于 1.3 版)可以用单一的调用所取代:

```
void put_usr(type value, type *u_addr);
```

这一调用在 <asm/segment.h>中作为宏指令实现。它的扩展取决 u_addr 参数的数据类型。u_addr 指针指向的目标类型决定了传送到用户内存区的 value 的具体字节数。在你使用这些函数中的任何一个函数写到用户内存区之前,应该检验由用户进程传送给你的用户空间地址是否指向用户有写权限的内区存区。这可以用内核 verify_area( )函数来完成:

```
#include <linux/mm.h>

int verify_area(int access,  void *u_addr,  unsigned long size);
```

其中 access 取值 VERIFY_WRITE 或 VERIFY_READ,取决于你写到用户内存中去还是从用户用户内存中读出。u_addr 是用户空间的起始地址,而 size 则是存取的内存块的字节数。

如果指定的存取方式是容许的,verify_area( ) 函数返回 0 值。在出错时返回 -EFAULT 值。

25.3.4 write 函数

当对属于这一驱动程序的设备特殊文件执行 write( )系统调用时,将调用驱动程序的 write( )函数:

```
void write(struct inode *inode, struct file *file, char *buf, int count);
```

其中 inode 是指向特殊文件的索引结点结构的指针。file 是指向文件结构的指针。buf 是指向用户空间缓冲区的指针,由用户进程送入 write( )系统调用。用户的字符将写在这一缓冲区中。而 count 则是要传送的字节数,同样由用户进程传送给 write( )系统调用。

设备驱动程序的 write( )函数的用途是将数据按要求的字节数 count 从用户空间的缓冲区

buf复制到硬件或内核的缓冲区中。

和read( )函数一样,buf的值是用户空间的一个地址。同样应该用特殊函数完成复制工作:

```
unsigned char get_user_byte(char *u_addr);
unsigned short get_user_word(short *u_addr);
unsigned long get_user_long(long *u_addr);
void memcpy_fromfs(void *k_addr, void *u_addr, unsigned long cnt);
```

对get_user…( )类函数讲,从用户空间地址u_addr取回数据。取回的数据数分别由char,short或long的值指定。对memcpy_fromfs( )函数讲,将cnt个字节从用户空间地址u_addr复制到内核空间地址k_addr。

同样为了与旧的内核向下兼容,还提供了另外三个函数:

```
unsigned char get_fs_byte(char *u_addr);
unsigned short get_fs_word(short *u_addr);
unsigned long get_fs_long(long *u_addr);
```

它们和前面三个get_user_XXX函数的功能相同。

新的内核还支持单一的调用:

```
type get_user (type *u_addr);
```

它在<asm/segment.h>中作为宏指令实现。它的扩展取决于指针参数u_addr的数据类型。u_addr指针指向的目标类型用来决定从用户内存空间传送多少个字节的数据,也用来决定get_user( )调用本身的返回类型。

和设备驱动程序的read( )函数一样,应该在使用上面的函数从用户内存区读数以前先调用verify_area( )函数检验u_addr

25.4 特殊的控制函数

除了基本的open,close,read和write操作外,有时还需要传送控制信息给设备驱动程序,或者从它那里取得状态信息。例如,改变串行口的传输速率并未由基本的open,close,read和write提供。在这种情况(以及其他许多类似的情况)下,要用到ioctl( )系统调用。这一系统调用只是调用设备驱动程序中的ioctl( )函数来进行处理。

这样做是必要的,因为不同的设备驱动程序并不需要实现同样的功能。例如,对许多设备讲,改变传输速率对它们并没有任何意义。

设备驱动程序的ioctl( )函数的典型形式为:

```
int ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg);
```

其中 inode 和 file 的含义和前面的相同;cmd 是设备驱动程序要执行的命令的特殊代码;arg 是任何类型的 4 字节数(如一个整数或结构的指针),它为特定的 cmd 值提供参数。cmd 和 arg 参数从 ioctl( )系统调用中的第 2,3 项参数获得。

不要忘记,如果 arg 参数是指向用户空间的内存块的指针,必须使用前面提到的特殊复制函数将数据送到内存块或从内存块中取走。

一般的讲 ,除了 Linux 使用的 4 个特殊的值以外(这些值在调用 ioctl( )函数前由系统进行解释),你可以选用任何其他数字作为 cmd 的值,系统使用的 4 个值如下:

```
FIONBIO    0x5421  set/reset O_NONBLOCK from arg;
FIONCLEX   0x5450  clear close-on-exec flag;
FIOCLEX    0x5451  set close-on-exec flag;
FIOASYNC   0x5452  set/reset O_SYNC from arg.
```

如果你使用上面的 4 个命令值中的任何一个作为 ioctl( ) 函数的命令值,函数将不被调用,因为内核先对它们进行解释。

25.5 中断

除了通过 file_operation 结构调用的函数外,设备驱动程序还提供中断服务子程序 interrupt( )。当用到的硬设备能产生中断信号时,才需要中断服务子程序。

需要在内核中登记设备驱动程序,使得用主设备号访问设备特殊文件时,将调用驱动程序子程序。与此类似,也需要请求内核将特定的中断请求行和中断服务子程序联系在一起,当中断发生时,调用中断服务子程序。可以用 request_irq( ) 函数来实现请求:

```
#include  <linux/signal.h>

int request_irq(unsigned int irq, void (*handler)(int), unsigned long type, char *name);
```

其中 irq 是试图登记的中断请求号。handler 是指向中断服务子程序的指针。type 是一个标志用来确定这是正常中断还是快速中断。name 则是设备驱动程序的名称。

正常中断和快速中断的差别在于:从执行正常中断的子程序返回时,内核可以利用机会运行调度程序 (调用内核 schedule( )函数) 来检查是否有比当前进程更合适的进程要执行。如果有,就运行新进程取代当前的进程。在快速中断的情况下,当从中断服务程序返回时,内核不进行调度,立即恢复被中断程序的执行。

对正常中断 type 的值取为 0,对快速中断 type 的值取符号常数 SA_INTERRUPT。

可以用 free_irg( )函数撤销中断服务子程序的登记:

```
void free_irq(unsigned int irg);
```

其中 irq 是释放的中断请求号。

25.6 设备驱动程序安装

一旦写完了设备驱动程序,下一项任务是对它进行编译和装入可引导的内核。实际上这并不复杂,可以用下面的步骤来完成:

- 第一项任务是将 driver.c 和任何有关的 driver.h 文件复制到包含字符设备驱动程序源码的目录下,它放在 Linux 源程序目录下的 drivers/char 子目录中。
- 其次,你应该在 chr_dev_init( )函数的最后增加调用 init( )子程序的行。chr_dev_init( )函数在 drivers/char/mem.c 中。
- 现在编辑 drivers/char 目录中的 makefile,将 driver.o 的名称放在 OBJS 定义的后面,并将 driver.c 名称放在 SRCS 定义的后面。
- 现在将目录改变到 Linux 源程序目录的最上层,用 7.4 节中所述的步骤建立和安装你的新内核。

如果用 lilo 引导系统,最好将新内核作为试验项,在 lilo.conf 文件中另加一 Linux 引导段。这样,如果新内核由于某种原因不能正常运转,还能从旧内核引导系统。

作为一般性预防性措施,当改变内核的代码时,将机器上重要的内容作一次备份是明智的。一般说来,无论是新内核不工作,还是新内核导致工作混乱,还不致于导致永久性损坏。特别是你还能从旧内核重新引导机器。然而从理论上讲,有隐错的内核可能出现任何问题,包括破坏硬盘上的数据。确保安全以免造成不必要的遗憾!

第二十六章　微小设备驱动程序——实例研究

为了使我们在前面讨论的设备驱动程序的内容更加具体化，我们将讨论一个非常简单的字符设备驱动程序作为示范。虽然，对这一驱动程序的技术要求有点矫揉造作，但却让你看到一些有趣的想法如何付诸实施。

对这一设备驱动程序的技术要求如下：用两个设备特殊文件实现一个字符设备驱动程序，使得一对进程之间能相互传送短的可变长度的正文信息；不允许设备驱动程序进行多重读和多重写；此外，即使没有信息，不阻止 read( )操作；在下次 read( )操作前，不管写了多少信息也不阻止 write( )操作。

必须指出：最后的不限止写的能力的要求是不安全的。由于出现某种错误，使得使用数据的进程不能读任何数据的情况可能发生，因而使产生数据的进程不加检查和无休止地写信息，直到用完系统空间为止。事实上，对长期存在的进程讲，即使没有错误也能发生类似的情况。所要的条件只是：从平均的意义讲，产生数据的进程产生数据的速度高于使用进程消耗数据的速度。考虑到这一点，在设备驱动程序内部对存储信息的容量设置大的适当的限制是明智的，我留下安全检查作为练习，请你以后加上。

事实上，这一设备驱动程序只控制一些系统内存作为它的'硬设备'，并在已有的 IPC 机制上有效地提供另一种 IPC 机制。这一驱动程序提供的新的 IPC 和已有的 IPC 机制有很大的不同。

26.1　Header 信息

为实现上面的技术要求，第一项任务是确定设备驱动程序内部的数据结构。设备驱动程序内部应能存储许多在两个进程之间交换的短的正文信息。假定驱动程序要保存的信息数量没有限止（对实际计算机讲，没有限止也就是用完为止），但在能读这些信息之前实际上只保存了少量的信息。这表示你实际需要使用数量可变的动态分配的缓冲区。最好将这些缓冲区用连接表的形式建成先进先出（FIFO）的队列。这需要用到动态分配和释放内存块的内核机制。

注意：不能用标准的 malloc( ) 和 free( )库函数，因为不能从内核中使用它们。但另有完成 malloc( )同样功能的内核函数可供使用。

建立信息连接表的结构具有以下格式：

```
struct tdd_buf
{
    int buf_size;
    char buffer[MAX_BUF];
    struct tdd_buf * link;
};
```

其中 buffer[ ]是保存一条短信息的数组;but_size 说明 buffer[ ]数组中有多少个字符在使用。link 是指向下一个 tdd_buf 的连接表指针。符号常数 MAX_BUF 可以设为任何你认为是'短信息'的最长长度。默认值是 120 个字符。

注意:这一设备驱动程序使用前缀 tdd_,使它成为唯一的识别符。许多识别符说明为 static,它们不能在驱动程序的源文件以外使用。

这一设备驱动程序 header 信息如下:

```
/* 1 */
#define KERNEL

#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/signal.h>
#include <linux/errno.h>
#include <linux/malloc.h>

#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>
#include <asm/irq.h>

#include "tdd.h"

1* 2 */
static int tdd_trace;
static int write_busy;
static int read_busy;
static wtruct tdd_buf *qhead;
static wtruct tdd_buf *qtail;

/* 3 */
static int tdd_read(struct inode *, struct file *, char *, int);
static int tdd_write(struct inode *, struct file *, char *, int);
static int tdd_ioctl(struct inode *, struct file *, unsigned int, unsigned long);
static int tdd_open(struct inode *, struct file *);
static void tdd_release(struct inode *, struct file *);
extern void console_print(char *);

struct file_operations tdd_fops =
{
  NULL,
  tdd_read,
```

```
    tdd_write,
    NULL,
    NULL,
    tdd_ioctl,
    NULL
    tdd_open,
    tdd_release,
    NULl,
    NULl,
    NULl,
    NULL
};
```

下面是 header 信息中标有数字的注释：

1. 第一段包括所有有关的头文件。注意 KERNEL 符号的定义；它用在一些头文件中（包括 tdd.h）。当头文件用在内核代码中时，有条件地包括一些定义和附加项。普通用户代码中不定义这一符号，因此不包含这些附加项。
2. 这是在驱动程序内部使用的 static 变量。变量 tdd_trace 用作打开和关闭查错跟踪输出的标志；write_busy 和 read_busy 是用来防止多重读和多重写的标志；而 qhead 和 qtail 分别是信息连接表的头、尾指针。
3. 本段建立指向其他设备驱动程序子程序的 file_operation 结构。在引导系统时，由初始化子程序将指向这一结构的指针传送给内核。在其他地方不说明的内核函数也在这里说明。

除了上面的 header 代码（header code）外，还有称为 tdd.h 的头文件。它包含设备驱动程序和使用这一驱动程序的用户代码所需要的定义和结构说明：

```
#ifdef KERNEL                          /* If we're in kernel code */

#define TRACE_TXT(text) \
    { \
        if (tdd_trace) \
        { \
            console_print(text); \
            console_print("\n"); \
        } \
    }

#define TRACE_CHR(chr) \
    { \
        if (tdd_trace) \
            console_print(chr); \
    }
```

```
#define TDD_WRITE 0                    /* /dev/tddw minor device number */
#define TDD_READ 1                     /* /dev/tddr minor device number */

#endif

#define FALSE 0
#define TRUE 1
#define MAX_BUF 120                    /* Size of struct tdd_buf buffer */
#define TDD_TRON (('M' << 8)|0x01)     /* Trace on cmd for ioctl() */
#define TDD_TROFF (('M' << 8)|0x02)    /* Trace off cmd for ioctl() */

struct tdd_buf
{
    int buf_size;
    char buffer [MAX_BUF];
    struct tdd_buf * link;
};
```

对宏定义 TRACE_TXT( )和 TRACE_CHR( ) 需要作些解释。这些是为查错提供的非常简单的跟踪功能。在整个设备驱动程序代码中,一些重要的位置插入了这些宏命令的调用,所以能在屏幕上显示"到达这里"一类的信息。用内核函数 console_pnint( )在控制台屏幕上显示跟踪信息。标志变量的状态用来控制否显示跟踪信息。可以由 ioctl( ) 函数来设置和复位标志变量的状态。ioctl()中的两条命令由驱动程序用来控制标志变量的设置,TDD_TRON 用于设置跟踪标志,而 TDD_TROFF 则用于复位。这两条 ioctl( )命令都不带任何参数。

26.2 init 函数

我们先看初始化函数 tdd_init( ):

```
void tdd_init(void)
{
    tdd_trace = TRUE;

    if (register_chrdev(30, "tdd", &tdd_fops))
        TRACE_TXT("Cannot register tdd driver as major device 30")
    else
        TRACE_TXT("Tiny device driver registered successfully")

    qhead = 0;
    write_busy = FALSE;
    read_busy = FALSE;
    tdd_trace = FALSE;
    return;
```

```
}
```

在引导系统时执行这一子程序。应在 chr_dev_init( )函数中增加调用 tdd_init( )函数的行。chr_dev_init( )函数包含在下列文件中：

```
/usr/src/linux/drivers/char/mem.c
```

这里假定你的内核源程序存在/usr/src/linux 目录下。

当执行 tdd_init( )时，它将调用内核函数 register_chrdev( )。将驱动程序的 file_operation 结构的指针放在内核字符设备地址表中。利用查错跟踪功能，将操作成功与否的信息显示在机器的控制台屏幕上。然后，它对驱动程序的 static 变量进行初始化后返回。

26.3 open 函数

如果对属于这一驱动程序的两个设备特殊文件之一执行 open( )系统调用，将调用设备驱动程序的 open 函数（ tdd_open( ) ）：

```
static int tdd_open(struct inode * inode, struct file * file)
{
    TRACE_TXT("tdd_open")

    /* 1 */
    switch (MINOR(inode->i_rdev))
    {
    /* 2 */
    case TDD_WRITE:
        if (write_busy)
            return - EBUSY;
        else
            write_busy = TRUE;
            return 0;

    /* 3 */
    case TDD_READ:
        if (read-busy)
            return - EBUSY;
        else
            read_busy = TRUE;
            return 0;

    default:
        return - ENXIO;
    }
```

```
}
```

如果有任何硬件和设备驱动程序结合在一起,这一函数将使硬件投入服务。我们的驱动程序没有和硬件结合,只实现语义上的要求,对两个设备特殊文件各打开一次:

1. 第一项任务是为准备打开的设备特殊文件提取辅助设备号。这可以用宏命令 MINOR( )在 inode 参数所指的结构中在 i_rdev 字段中提取。
2. 如果设备特殊文件是 TDD_WRITE (即 /dev/tddw),则检查 write_busy 标志,看这一设备是否已经打开。如果是,返回 -EBUSY 错误信息。否则将标志设为 true,表示设备忙,阻止再次打开设备。返回 0 值表示打开成功。
3. 类似地,如要请求打开 /dev/tddr (使用 TDD_READ),则对 read_busy 标志进行检查。如果需要,设置这一标志,以保证对这一设备特殊文件进行排他性访问。

26.4 release 函数

当最后一个打开属于这一驱动程序的设备特殊文件的进程用 close( ) 系统调用关闭它时,才调用 tdd_release ( ) 释放函数。事实上,每次只有一个进程能打开这一驱动程序的每个设备特殊文件,所以从这一进程发出的 close( ) 调用将调用驱动程序的释放函数(tdd_release( )):

```
static void tdd_release(struct inode * inode, struct file * file)
{
    TRACE_TXT("tdd_release")

    switch (MINOR(inode->i_rdev))
    {
    case TDD_WRITE:
            write_busy = FALSE;
            return;

    case TDD_READ:
            read_busy = FALSE;
            return;
    }
}
```

tdd_release 所做的只是将读和写的忙标志复位。为下一个打开设备特殊文件的操作作好准备。

26.5 write 函数

当进程每次用 write( )系统调用对这一驱动程序的设备特殊文件的打开文件描述符(open file descriptor)进行操作时,将调用写函数 tdd_write( ):

```
static int tdd_write(struct inode *inode, struct file *file,
char *buffer, int count)
{
    int i, len;
    struct tdd_buf *ptr;
    TRACE_TXT("tdd_write")

    /* 1 */
    if (MINOR(inode->i_rdev)! = TDD_WRITE)
        return - EINVAL;

    /* 2 */
    if ((ptr = kmalloc(sizeof(struct tdd_buf), GFP_KERNEL)) == 0)
        return - ENOMEM;

    /* 3 */
    len = count < MAX_BUF? count:MAX_BUF;

    if (verify_area(VERIFY_READ, buffer, len))
        return - EFAULT;

    for (i = 0; i < count && i < MAX_BUF; ++i)
    {
        ptr->buffer[i] = get_user_byte(buffer + i);
        TRACE_CHR("w")
    }

    /* 4 */
    ptr->link = 0;

    if (qhead == 0)
        qhead = ptr;
    else
        qtail->link = ptr;

    qtail = ptr;
    TRACE_CHR("\n")

    /* 5 */
    ptr->buf_size = i;
    return i;
}
```

tdd_write( )中第3,4项参数是缓冲区和字符计数。它们由用户进程传送给 write( )系统调用。缓冲区的内容需要被复制到设备驱动程序内部的信息连接表。不能直接通过 tdd_write( ) 的 buffer 参数访问用户空间的内存;必须用一个特殊数据传送函数来完成此项任务。对清单中的编号注释说明如下:

1. 检验和保证只有打开 /dev/tddw 设备特殊文件的进程才能写信息。
2. 分配足以容纳单条信息的内核内存块,进行检验确保没有问题。如有问题返回出错信息。
3. 检验内核能从指定的内存块中读取。然后将信息从用户空间复制到内核分配的信息空间。实际复制的字符数由 count 或 BUF_MAX 的大小决定,取其中较小的一个作为字符计数。
4. 将新的信息结构连接到设备驱动程序的连接表的最后。
5. 在信息结构中设置实际信息长度,同时将这一值用 tdd_write( )返回。这一值并由 write( )系统调用返回给发出调用的用户进程。

26.6 read 函数

当一个用户进程调用 read( )系统调用从这一设备驱动程序控制的设备特殊文件中读取数据时,才调用 tdd_read( )函数:

```
static int tdd_read(struct inode * inode, struct file * file,
char * buffer, int count)
{
    int i, len;
    struct tdd_buf * ptr;

    TRACE_TXT("tdd_read")

    /* 1 */
    if (MINOR(inode -> i_rdev)! = TDD_READ)
        return - EINVAL;

    /* 2 */
    if (qhead == 0)
        return - ENODATA;

    /* 3 */
    ptr = qhead;
    qhead = qhead -> link;

    /* 4 */
    len = count < ptr -> buf_size? count:ptr -> buf_size;

    if (verify_area(VERIFY-WRITE, buffer, len))
        return - EFAULT;
```

```
    for (i = 0; i < count && i < ptr->buf_size; ++i)
    {
        put_user_byte(ptr->buffer[i], buffer + i);
        TRACE_CHR("r")
    }

    TRACE_CHR("\n")

    /* 5 */
    kfree_s(ptr, sizeof(struct tdd_buf));
    return i;
}
```

同样,tdd_read( )函数中的 buffer 和 count 参数分别是指向用户空间的缓冲区的指针和字符计数。它们由用户进程作为参数传送给 read( )系统调用。和 tdd_write( )不同之处在于缓冲区现在从设备驱动程序的信息结构中接受字符。编号的注释说明如下:

1. 只允许对 /dev/tddr 设备特殊文件进行 read( )调用。
2. 如果在队列中没有信息,并不阻止 read( )调用,只将 errno 变量的值设为 ENDATA,并返回 -1 值。
3. 将第一条信息从队列中取下,设置变量 ptr 使它指向这条信息。
4. 检验内核能否写指定的内存块,然后,按要求的字符数(count)或者信息缓冲区中的实际字符数 (ptr->buf_size) 将信息复制到用户空间的缓冲区 (buffer)。用其中较小的数字进行复制。用户有责任保证指定的缓冲区足以容纳请求的字符数,内核不进行此项检查。
5. 最后将旧的信息结构释放归还内核,并返回实际传送的字符数。

26.7 ioctl 函数

这一驱动程序的 ioctl( )函数非常简单,但足以说明一些概念。这一驱动程序提供了两个新的 ioctl( )调用,用来打开和关闭跟踪功能。分别用 TDD_TRON 和 TDD_TROFF 命令来完成:

```
static int tdd_ioctl(struct inode * inode, struct file * file,
unsigned int cmd, unsigned long arg)
{
    TRACE_TXT("tdd_ioctl")

    switch(cmd)
    {
    case TDD_TRON:
        tdd_trace = TRUE;
        return 0;
```

```
        case TDD_TROFF:
            tdd_trace = FALSE;
            return 0;

        default:
            return -EINVAL;
        }
    }
```

送给 tdd_ioctl( )的 cmd 和 arg 参数和用户进程送给 ioctl( )系统调用的值相同。在现在的情况下 arg 参数值不重要。对这一驱动程序讲,只有 TDD_TRON 和 TDD_TROFF 两个值中的一个用作 cmd 的值时才有意义。这些 ioctl( )命令用来设置和恢复 tdd_trace 标志。

第二十七章　块设备驱动程序

在字符设备程序中用到的许多概念可以直接用到块设备驱动程序中。在本章中，我将重点放在比较两者的差别。

总的说来，在 Linux 系统中建立块设备驱动程序是一项烦琐而又需要谨慎对待的任务。它在很大程度上依赖于使用掩盖内在复杂性的宏命令，这样做又使程序看上去像包含了许多“魔术的咒语”一样，使人一下不容易弄明白。

27.1　块设备驱动程序的概念

正像字符设备驱动程序一样，块设备驱动程序也由许多子程序或函数组成。调用这些子程序将在驱动程序控制的设备上完成不同的操作。

驱动程序使用的函数的清单包括下列的全部或一部分：open( )，release( )，ioctl( )，interrupt( )，init( )和 request( )。前面 4 个函数的格式和功能和字符设备驱动程序使用的函数相同。

在写块设备驱动程序时，要做的第一件事是为驱动程序选择块设备的主设备号，并把它加到下列文件中：

```
/usr/include/linux/major.h
```

在这个文件中包含标准内核使用的所有主设备号和每个主设备号的符号名称。应该在清单中加上主设备号和符号名称的定义。例如：

```
#define  MY_MAJOR  30
```

做完这件事后，就可以开始写设备驱动程序的代码。作为开始，应该使用类似下面的行：

```
#includde <linux/blkdev.h>

#define MAJOR_NR MY_MAJOR

#include "blk.h"
```

注意：MAJOR_NR 的定义指定用于这一设备驱动程序的主设备号。这一定义必须放在 #include "blk.h"行之前。你在后面将看到，这个符号用在 blk.h 文件中。blkdev.h 文件本身包含你刚才修改的 major.h 文件，将自动包括 MY_MAJOR 的定义。

27.1.1　高速缓冲存储

块设备的实际输入/输出是耗费时间的操作。应该尽可能减少这种操作。使用的方法是：

利用部分系统内存作为读、写进程和设备驱动程序之间的缓冲区，这种方法称为高速缓冲存储。

这说明块设备驱动程序不应包含 read( )和 write( )函数。当用户进程对驱动程序控制的设备特殊文件发出 read( )或 write( )系统调用时，将使用高速缓冲技术来处理它的请求。

仅当绝对必要时，才调用设备驱动程序完成实际的输入/输出操作。它将在设备驱动程序的请求队列中增加一项输入/输出请求，然后调用你的驱动程序的 reguest( )函数来处理这一请求队列。

27.1.2 request 函数

当 request 函数被调用时，它的任务是从请求的队列中依次读取挂起的输入/输出请求，并安排完成指定的读、写操作。请求队列中的每一个输入/输出请求保存在 request 结构中。这些结构在下列文件中定义：

/usr/include/linux/blkdev.h

在设备驱动程序中，不带中断服务子程序的 request( )函数是简单的，它的格式如下：

```
static void do_my_request(void)
{
loop:
INIT_REQUEST;

if (MINOR(CURRENT->dev) > MY_MINOR_MAX)
{
    end_request(0);
    goto loop;
}

if (CURRENT->cmd == READ)
{
    end_request(my_read());
    goto loop;
}

if (CURRENT->cmd == WRITE)
{
    end_request(my_write());
    goto loop;
}

    end_request(0));
    goto loop;
```

}

要说明的第一点是:CURRENT 是指向请求队列开头的 reguest 结构的指针。CURREN 在 blk.h 头文件中定义。

request( )函数从 INIT_REQUEST 宏命令开始(它也在 blk.h 中定义)。它对请求队列进行检查,保证请求队列中至少有一个请求在等待处理。如果没有请求(即 CURRENT==0),INIT_REQUEST 宏命令将使 request( ) 函数返回,它的任务结束。

假定队列中只少有一个请求,request( ) 函数现在应处理队列中的第一个请求,当处理请求后,request()函数将调用 end_request( ) 函数。

如果成功地完成了输入/输出操作。应该用参数值 1 调用 end_request( )。如果输入/输出不成功,以参数值 0 调用 end_request( ) 函数。

如果需要,end_reuest( ) 函数将记录错误信息。将处理了的请求从队列中移去。安排任何等待这一输入/输出请求完成的进程的启动。如果队列中还有其他请求,将 CURRENT 指针设为指向下一个请求。

执行 end_request( ) 函数后,request( ) 函数回到循环的起点,对下一个请求重复上面的处理过程。

在下一简单的例子中,请求本身的处理是非常简单的,大多数工作是围绕 CURRENT 请求的内容进行的。请求结构的格式如下:

```
struct request
{
    int dev;
    int cmd;
    int errors;
    unsigned long sector;
    unsigned long nr_sectors;
    unsigned long current_nr_sectors;
    char *buffer;
    struct semaphore *sem;
    struct buffer_head *bh;
    struct buffer_head *bhtail;
    struct request *next;
};
```

在这一结构中的主要字段和它们完成的任务如下:

dev	指定这一请求使用的实际设备;
cmd	要完成的命令(读或写);
sector	开始读、写的扇区号;
nr_sectors	读写的扇区数;
buffer	供读、写数据用的内核内存缓冲区。

在处理请求时,首先完成一些简单而明确的检查,以保证当前的请求指定了有效的实际设备。如果发现错误,以参数值 0 调用 end_request( ) 函数。

如果设备是有效的,用 CURRENT->cmd 的值来调用 my_read( ) 或 my_write( )函数。如果发生了错误,这两个函数的返回值必须是 0;如果一切进行顺利,返回值为 1。这些值直接送给 end_request( ) 函数,以相应的值结束当前的请求;

my_read( ) 和 my_write( ) 函数将使用当前请求结构中的 sector,nr_sector 和 buffer 字段的值按要求传送数据。

27.2 登记块设备驱动程序

登记块设备驱动程序要比登记字符设备驱动程序更为复杂。它既需要像字符设备驱动程序一样在引导内核时完成一定的工作,还需要在内核编译时增加一些内容。

27.2.1 引导时的登记

引导时的登记工作由驱动程序的 init( ) 函数完成。对 1.1.33 版以后的内核讲,它的典型形式为:

```
void my_init(void);
```

为了在引导内核时调用 init( )函数,需要在内核的 blk_dev_init( ) 函数中添加一行代码。这一函数包含在下列文件的末尾:

```
/usr/src/linux/drivers/block/ll_rw_blk.c
```

增加的一行代码应插在 blk_dev_init( ) 函数的 return 0; 语句的前面。增加的一行代码的格式为:

```
my_init( );
```

当在引导内核时执行 init( ) 函数时,它必须检查需要的硬件是否存在,并对它进行初始化。它也需要登记驱动程序使用的主设备号,同时将驱动程序的 file_operation 结构的指针传递给内核。

用于块设备驱动程序的 file_operation 结构类似于字符设备所用的结构,但在一些字段中插入了固定的地址。这样就能调用标准的内核子程序而不是调用设备驱动程序函数进行处理。这一结构的典型形式如下:

```
struct file_operations my_fops =
{
    0,
    block_read,          /* kernel function */
```

```
    block_write          /* kernel function */
    0,
    0,
    my_ioctl,            /* device driver function */
    0,
    my_open,             /* device driver function */
    my_release,          /* device driver function */
    block_fsync,         /* kernel function */
    0,
    0,
    0,
};
```

所有的块设备驱动程序中都调用内核的 block_read( ),block_write( ) 和 block_fsync( ) 函数,这表示不必要在块设备驱动程序中包含这些函数。本例同时表明:应该使用自己的设备驱动程序的 ioctl( ),open( ) 和 release( ) 函数。

一旦你确定了驱动程序使用的主设备号并建立了 file_operation 结构,就可以用 register_blkdev( ) 函数登记你的设备驱动程序:

```
if (register_blkdev(MY_MAJOR, "my_bdev", &my_fops))
{
    printk("MY_BDEV: Unable to register device \n");
    return; /* return mem_start; for pre 1.1.33 kernels */
}
```

register_blkdev( ) 函数第 1 项参数是这一驱动程序使用的主设备号;第 2 项参数是设备名称的字符串;第 3 项参数是指向 file_operation 结构的指针。如果一切进行顺利,函数将返回 0 值。如果出现错误,则返回负值。如果指定的主设备号为 0,register_blkdev( )将搜索空闲主设备号,并将它作为返回值提供给你。

你已经看到,当内核要完成实际输入/输出操作时,它将调用驱动程序的 request( )函数,但是这一函数的地址没有包含在 file_operation 结构中。所以,驱动程序的 init( ) 函数的另一项任务是将 request( ) 函数的地址告诉内核,这可以用包含下面一行标准代码来实现:

```
blk_dev [MY_MAJOR].request_fn = DEVICE_REQUEST;
```

这里假定你用了主设备号 MY_MAJOR。.符号名称 DEVICE_REQUEST 定义为驱动程序的 request( ) 函数的名称 (也就是它的地址) (后面我们将看到在什么地方给出这一定义)。

init( ) 函数要准备的最后一件事是告诉高速缓冲存储新的驱动程序控制的设备的数据块的容量。

```
my_block_size = 512;
blksize_size[MY_MAJOR] = &my_block_size;
```

同样,这些代码将值存入用主设备号索引的数组中。

27.2.2 编译时登记

到现在为止,除了 DEVICE_REQUEST 符号外,还没有告诉内核到那里去找你的 request( ) 函数。为了完成登记过程,需要将这一宏定义及其他宏定义加到 blk.h 文件中。blk.h 文件存在块驱动程序源码目录下:

```
/usr/src/linux/drivers/block
```

为了做到这一点,需要在文件中找到类似下面的行:

```
#endif  /*  MAFOR_NR == whatever  */
```

然后,在这一行的前面加入下列的宏定义:

```
#elif (MAJOR_NR == MY_MAJOR)

static void do_my_request(void);

#define DEVICE_NAME "My Block Driver"
#define DEVICE_REQUEST do_my_request
#define DEVICE_NR(device) (MINOR(device))
#define DEVICE_ON(device)
#define DEVICE_OFF(device)
```

因为它放在其他驱动程序的类似的宏定义块的后面,容易检查将这块代码放入了文件中的正确位置。

对不同的宏定义说明如下:

```
DEVICE_NAME     短的驱动程序名称;
DEVICE_REQUEST  指向驱动程序 request( ) 函数的指针;
DEVICE_NR(d)    根据辅助设备号计算实际设备号;
DEVICE_ON(d)    用于需要打开的设备;
DEVICE_OFF(d)   用于需要关闭的设备(软盘)。
```

27.3 中断

与字符设备驱动程序一样,利用 request_irq( ) 函数将块设备驱动程序的中断服务子程序和特定的中断请求行联系起来。

驱动程序的中断服务子程序的中断过程如下:

- 用户进程发出某种类型的输入/输出请求（譬如说，read( )系统调用），或许先通过高速缓冲存储，最终要求进行实际设备的输入/输出。
- 调用设备驱动程序的 read( )函数，或者 request( )函数，将完成输入/输出操作的指令送给硬设备，现在设备驱动程序等待操作的发生。
- 一些时间以后，硬设备准备好完成指定的操作，并产生中断信号标志事件的发生。
- 中断信号导致调用驱动程序的中断服务子程序，它将所要的数据从硬设备复制到设备驱动程序的内存缓冲区，并通知正在等待的 read( )（或 request( )）函数，数据现在已经可供使用。
- 在数据可供使用时，read( ) 或 request( )函数现在可将数据提供给用户进程。上述的中断过程是经过简化的，但反映了过程的主要方面。还有几点需要进一步说明：
- 如果中断服务子程序需要改变某些数据结构，而内核其他部分也要改变这些数据结构，将会发生什么情况？
- 当发生硬设备的输入/输出请求时，驱动程序的 read( )，write( ) 和 request( )函数又做些什么？
- 为什么中断服务子程序不直接将数据复制到用户空间，或者从用户空间读取数据？

27.3.1 关键性段落

当两段内核程序都要访问和修改特定的内核数据结构时，如果想避免出现灾难性后果，应该仔细协调它们的行为。当中断服务子程序作为两段内核程序中的一段修改数据结构时，问题就显得更为严重。

考虑下面的情况：

设想内核中某一程序要改变连接表的结构，准备将新结点加到连接表中。它在表中适当的位置断开连接，并将新元素加进去。在修改过程中的某一时刻，数据结构将处于不稳定状态。其中只有部分任务已完成，一个或几个指针可能指向不正确的位置。在一般情况下，这不会构成问题。因为再执行两、三条语句以后，增加结点的任务已经完成，一切都恢复到稳定的状态，其他程序使用连接表就不成问题。然而，如果连接表处于不稳定状态时发生了中断，而且如果中断服务子程序也要访问和修改同一个连接表，我想你能看到问题的严重性。

这是在关键性段落的总标题下出现的问题的性质。在这一情况下，关键性段落的概念涉及的不是数据结构，而是对数据结构进行关键性访问的程序段。很明显这里的概念是要保证数据的精确性和完整性。

内核中包含对某一内核数据结构的关键性段落的任何程序，在执行关键性段落时，必须有一种机制保护自己不受中断干扰。

当确定程序的关键性段落后，用来处理这一问题的标准技术是在关键性段落执行期间禁止中断。很明显，关键性段落应尽可能短，使得中断不致延时过长。必要时，在内核程序中，可以用两条特殊命令禁止和允许中断请求：

```
#include <asm/system.h>

sti( );
cli( );
```

sti( )和 cli( )都是在 < asm/system.h > 中定义的宏命令。sti( )调用用来允许中断请求,而 cli( )调用则用来禁止中断请求。

27.3.2 休眠和唤醒

当设备驱动程序函数对能产生中断的硬设备发出输入/输出请求时,这一函数必须等待,直到硬设备完成它的功能,产生中断,并由中断服务子程序将数据复制到内存为止。只在这时候,原先的设备驱动程序函数才能继续运行。很可能这一函数只是进入 while 循环,等待中断服务子程序完成它的任务。从处理机使用的角度看,这是极度低效的。特别是在请求的数据最终可供使用以前,消逝的时间间隔是不定的。

这里的实际问题是一旦一个内核程序控制了 CPU,它将一直保持控制直到自愿放弃为止。

不要忘记,设备驱动程序的函数是在用户进程发出了系统调用后才被调用的。虽然,这些函数在内核中执行,代表的却是用户进程。这表示直到它的输入/输出请求得到满足以前,用户进程那里什么事情也不会进一步发生。所以这些函数进程没有必要继续保持对 CPU 的控制。如果这是用户程序产生的进程,控制将自动被切换。如果是内核程序的进程,必须由它自己放弃控制,在它等待输入/输出完成时允许其他进程继续运行。

为了释放对 CPU 的控制,驱动程序函数将调用 sleep_on( )函数:

```
#include < linux/sched.h >

void sleep_on (struct wait_queue **ptr);
void interruptible_sleep_on (struct wait_queue **ptr);
```

从调用的角度看,对 sleep_on( ) 函数的调用和对任何其他函数的调用一样,但是直到调用函数等待的事件发生时,sleep - on( )函数才返回。

一个以上的进程等待同一事件的发生是可能的。在 Linux 系统中,将等待特定事件发生的所有进程放在单独的队列中。一个等待的队列实际是 wait_queue 结构的结点的循环连接表。在 sleep_on( ) 函数被调用时,由它自动建立这一连接表。

为了开始一个新的等待队列,需要说明指向等待队列结点的指针,并将它的初值置为 0,然后,将这一指针的地址传送给 sleep_on( ),它将当前进程加到等待队列,并将进程标记为休眠。使 Linux 调度程序下一轮调度中不选这一进程。然后,调用调度程序选择下一个要运行的进程:

```
#include< linux/sched.h >

static struct wait_queue *p = 0;

sleep_on(&p);
```

如果用了 sleep_on( ) 调用,那么,只有对同一等待队列使用 wake_up( ) 调用才能唤醒休眠的进程。除了可以送信号或超时信号唤醒休眠的进程外,可中断的休眠函数 interruptible_

sleep_on( )和 sleep_on( )函数功能相同。

当一个或多个进程等待的事件发生了(如输入/输出准备好),用特定的等待队列作为参数调用 wake_up( )函数,将导致这一队列中的所有进程都标记为可运行的进程。当调度程序再次被调用时,它将认为这些进程都有资格运行。

```
#include <linux/sched.h>

void wake_up(struct wait_queue **ptr);
void wake_up_interruptible(struct wait_queue **ptr);
```

对驱动程序来说,通常在中断服务子程序中将所需的数据复制到内核内存后,由它发出 wake_up( ) 调用。

注意:wake_up( ) 调用将所有在特定队列中休眠的进程恢复为可运行状态。这样做会产生一定的后果。例如,如果这些进程等待的事件是某一设备变为空闲,它们可以将数据写到设备上去。在这种情况下,新唤醒的进程中第一个投入运行的进程将占用这一设备。这说明这些进程被唤醒时设备是空闲的,但其中一个进程占用设备后,设备又处忙的状态。当可能出现这种情况时,在使用设备以前应该重新进行测试。如果它处于忙的状态,进程又回到休眠状态,即使这一进程刚从上一次休眠被唤醒也应该这么做。

wake_up( ) 函数唤醒特定队列中的所有进程,而 wake_up_interruptible( ) 函数只唤醒队列中可中断的进程。除此以外,两者的功能相同。

27.3.3 内存存取

你已经看到当一个进程在等待实际输入/输出发生时,可以进入休眠状态(由设备驱动程序调用 sleep_on( )),因而允许其他进程在这段时间使用 CPU。接下来,当发生中断时,中断服务子程序投入运行,但是中断服务子程序并没有在最终使用数据的进程的环境下运行。事实上,这一进程仍然处于休眠状态,等待中断服务子程序完成任务后去唤醒它。

由于当前运行的进程肯定不是最终使用数据的进程,所以在任何情况下,中断服务子程序都不应将数据直接从硬设备复制到用户空间。它应该做的是将数据复制到属于驱动程序的内核内存空间,并唤醒休眠的进程。这将导致原先的 sleep_on( )调用返回,进程再一次运行。由于现在程序运行在正确的环境中,将数据复制到用户空间是安全的。

27.4 计时程序

sleep_on( ) 和 wake_up( ) 通常用于 read( )/write( )/request( )和中断子程序中。然而一些不需要中断的驱动程序仍然需要休眠和在一定的时间延迟后被唤醒。

有一个简单的方法,只要写几行程序就能构成支持这一要求的函数:

```
#include <linux/sched.h>

extern unsigned long jiffies;
```

```
void my_delay(unsigned long jiffs)
{
    current -> state = TASK_INTERRUPTIBLE;
    current -> timeout = jiffies + jiffs;
    schedule();
}
```

其中 jiffies 是全程变量,也是一个计时程序。在引导系统时对它的值进行初始化,以后每秒钟增值 100 次(即每 10ms 增值一次)。my_delay( ) 中的 jiffs 参数同样用 10ms 为单位加以指定。

由于硬设备故障或类似的原因,没有出现期望的中断的情况是可能发生的,这表示一个休眠的进程不会被唤醒。在这种情况下,一个解决办法是通过一对函数 add_timer( ) 和 del_timer( ) 以及 timer_list 结构调用内核的计时程序:

```
struct timer_list
{
    struct timer_list *next, *prev;
    unsigned long expires, data;
    void (*function)(unsigned long);
};

void add_timer(struct timer_list *timer);
void del_timer(struct timer_list *timer);
```

add_timer( )调用设置成在特定时间(由 jiffies 给出)执行指定的函数。除非计时程序被撤销,它将在给定时间执行指定的函数。如果进程等待的中断发生了,则由中断服务子程序撤销计时程序(用 del_timer( ) 调用撤销)。如果由于某种原因中断没有发生,计时程序在指定时间达到后,调用指定的函数。通常这表示出现了错误,将利用 wake_up( ) 函数唤醒休眠的进程来处理这一情况。

第五篇

内　　核

第二十八章　进 程 调 度

作为多任务操作系统，进程调度是它的最基本的操作之一。希望在一台单处理器的机器上同时运行多个进程时，必须有某种形式的进程调度。这是明显的，因为在任何特定的瞬间，机器只能为一个进程执行一条指令。为了使机器上的若干进程同时取得进展，必须由准备好运行的进程共享 CPU 时间。调度程序的任务是选择下一个准备好运行的进程，将 CPU 时间分配给它。

28.1　背景

当调度程序完成调度任务时，它试图达到许多不同的目标。我们将看到一些目标对调度程序提出的要求是互相冲突的。调度程序的最重要的目标有：

- 使每个进程公平地共享 CPU 时间。
- 使 CPU 的空闲时间达到最少（即保持 CPU 处于忙碌状态）。
- 吞吐能力达到最高。这表示在给定时间内完成任务的进程数目达到最多。
- 使系统响应用户请求的时间达到最短。

似乎应该以某种方式优先考虑用户请求，但是这和所有进程公平共享 CPU 时间的目标明显发生冲突。

一般说来，调度程序直接面对的问题是：当它启动进程时，对进程了解甚少。如进程平均用多少 CPU 时间才停下来等待输入/输出；以及进程提出输入/输出请求后，平均用多长时间进行等待等都是未知数。

另一个问题是：对进行输入/输出前占用很长的 CPU 时间的进程应采取什么措施？能让它独占 CPU 一直运行下去？显然不能。否则这一进程可能使所有其他进程几乎无法取得任何进展。

这说明当前进程运行足够长的时间后，要有某种办法将 CPU 切换给另一个进程。但是从那里着手，又如何进行这样的调度。

这里有两种可能。第 1 种可能是让进程在 CPU 上运行一段时间后，自愿释放对 CPU 的控制。第 2 种可能是找出某种办法强制进程释放对 CPU 的控制。第 1 种称为非抢先调度。第 2 种称为抢先调度。

当我们讨论线程（thread）时将看到：运行由相互协调的程序组成的系统时，用非抢先调度进行切换是完全行得通的。然而对多用户环境下的进程调度讲，更安全的做法是：在最好的情况下，假定进程之间并不知道对方的存在。在最坏的情况下，假定进程之间相互竞争 CPU 的使用。结果是在多用户环境下几乎毫无例外地使用抢先调度，Linux 也不例外。

具体做法是：给每个进程分配一段最长的不间断的 CPU 时间，同时系统产生快速和周期性的时钟计时中断，用来决定进程什么时候拥有它的时间片。

当分配给当前进程的时间片消逝以后，强使调度程序投入运行。由它来决定是否还有准备好运行的进程，它是否比刚用完时间片的当前进程更有资格投入运行。如果有，由新的进程取代当前的前程。如果没有，让当前进程继续运行：

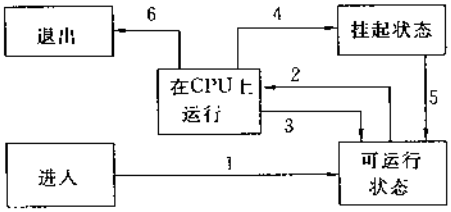

图 28.1 CPU 调度程序的简化状态转换图

从前面的讨论可以看到，进程有几种不同的状态。随着不同事件的出现，在这些状态间进行转换。图 28.1 表示在 CPU 调度程序控制下的简化的进程状态转换图，并且标出了在下列情况下发生的 6 种转换。

1. 启动（start）转换。当进程首次被启动时（在 Linux 系统中用 fork( ) 启动），并没有让它直接控制 CPU。而是将它置为可运行状态，和其他进程一起放在一个清单中。只要给它们分配 CPU 时间，就可以立即投入运行。
2. 将处于可运行状态的进程转换为运行状态。在可运行进程队列中的进程，最终将被调度程序选中，在 CPU 上执行一段时间。
3. 有几种方式使一个在 CPU 运行的进程转换为其他状态。转换 3 就是其中的一种，将当前运行的进程放回可运行的进程的清单中。当在处理器上运行的进程中用完了分配的时间片后，就发生这种转换，使其他进程有机会投入运行。
4. 另一种主要方式是当运行中的进程提出输入/输出操作的请求时，它将失去对 CPU 的控制。机器的硬设备对请求作出响应前会有一段时间延迟。在等待输入/输出完成时，即使 CPU 空闲，进程也不能运行。所以，当进程停下等待输入/输出完成时，它不能被放回可运行进程的清单中，而是通过转换 4 将它设置为挂起状态，等输入/输出的完成。
5. 当一个挂起的进程等待的输入/输出事件发生了。它再次成为有资格运行的进程。然而并不立即给这一进程提供 CPU 时间去处理它的输入/输出，只是将这它送到可运行的进程的清单中，等待再次被调度程序选中。这是转换 5 完成的工作。
6. 在简化图中，最后一种失去对 CPU 控制的方式发生在运行的进程结束时，转换 6 表示导致进程结束的事件。

也许出人意料，不管 CPU 调度程序采用什么算法决定下一个应该运行的进程，一般都能使用这个状态转换图。

CPU 调度程序可以使用许多可能的调度算去。也有许多关于在不同的环境下那种算法最佳的理论。这里不准备对调度算法进行一般性讨论，将集中于对 Linux 调度算法的细节做一些说明。

28.2 细节

在 Linux 进程的生存期间，它可能处于几种状态中的一种。下面是它的主要状态：

TASK_RUNNING 仅当进程处于这一状态时,调度程序才认为它是可以运行的。它等同于图 28.1 中的可运行状态。

TASK_INTERRUPTIBLE 处于这一状态的进程在休眠之中。但是,如果它接受一个信号,将使它转为 TASK_RUNNING 状态。这相当于图 28.1 的挂起状态。

TASK_UNINTERRUPTIBLE 除了不能用信号改变进程的状态外,类似上一个状态。通常需要用 wake_up( ) 函数来唤醒休眠中的进程。wake_up( ) 函数用来实现图 28.1 的转换 5。

Linux 系统中每个进程用 task_struct 结构来表示。其中包括进程运行环境的许多细节。在下列文件中说明这一结构:

```
/usr/src/linux/include/linux/sched.h
```

当进程刚建立时,由内核 do-fork( ) 函数将这一结构动态地分配给进程。do_fork( )函数包含在下列文件中:

```
/usr/src/linux/kernel/fork.c
```

task_struct 的成分之一称为 counter。它在进程调度中起重要作用。处于 TASR_RUNNING 状态的进程的 counter 的值越高,当调度程序运行时就越可能被选为下一次执行的进程。counter 的值一般在 0~70 范围内。在某些情况下,超出这一范围也是可能的。

在下列 4 种主要情况下,要求在 CPU 上运行的进程放弃对 CPU 的控制,使其他进程能够运行:

1. 当进程要求为它进行输入/输出操作时 (如磁盘的存取操作),而这一操作需要一定时间才能完成。在这种情况下,即使时间延迟只有几毫秒也要重新进行调度。
2. 不同信号引起的有关活动也能导致重新进行调度。如调用 pause( )或 sigsuspend( ) 系统调用或者从 ptrace( ) 接到一个信号等。
3. 当硬设备发出中断信号并唤醒为等待这一事件处于休眠状态的进程时,如果被唤醒的进程的 counter 值比当前进程的 counter 值高,就要重新进行调度。
4. 如果一个进程没有因为其他原因停下来,最终它将用完它的时间片,这也将导致重新进行调度。在 task_struct 结构中的 counter 值是分配给这一进程的剩下 CPU 时间的计数值 (以 10ms 的计时中断进行计数)。

从概念上讲,Linux 调度程序从它产生以来几乎没有变化。虽然 1.3 版以后的内核对代码进行了整理,并增加了对多处理器系统的支持。对单处理器的机器讲,进程调度是简单的,我们将进一步对它进行说明。

调度的任务本身是由内核中称为 schedule( ) 的单独的函数完成的。这一函数包含在下列文件:

```
/usr/src/linux/kernel/sched.c
```

下面的伪码 (pseudo code) 程序说明支持单处理器的内核的 schedule( )函数采取的调度

步骤：

```
Execute any functions on the scheduler task queue

FOR each process
    IF the process has a real time interval timer running
        IF the interval timer has expired
            Send a SIGALRM to the process
            Restart the interval timer if required
        END_IF
    END_IF

    IF the process state is TASK_INTERRUPTIBLE
        IF process has had a signal or has an expired timeout
            Set process state to TASK_RUNNING
        END_IF
    END_IF
END_FOR

FOR each process
    IF process state is TASK_RUNNING
        Remember the process with the highest counter value
    END_IF
END_FOR

IF all runnable processes have a counter value of zero
    FOR each process
        Recalculate process counter value from its priority
    END_FOR
END_IF

switch context to the remembered 'highest counter' process
```

正像你看到的，这是非常简单的调度算法。它在很大程度上依赖于 counter 值。用 counter 值来确定进程运行的先后顺序。一旦进程投入运行，又将它作为它的时间片的长度。

在支持多处理器的内核中，使用 counter 值来选择下一个运行的进程的方式稍有改变。除了使用 counter 值外，还增加了其他项目。使同一处理器上的进程具有某种优势(advantage)，同时也给当前进程一些小的优势。

有了这些说明和 Linux 的调度程序的源码，通过源码找出它如何进行工作应该不会太困难。

第二十九章　小型线程—实例研究

你已看到 Linux 系统中主要的并行操作单位是进程。为了从执行一个进程的环境切换到执行另一进程的环境有许多工作要做。这种环境切换的开销是必要的，因为 Linux 要保证任何一对进程之间互相都不干扰。特别是，这些进程可能属于不同的用户就更需要这样做。

你经常需要写一些程序，这些程序不仅知道对方的存在，而且还能协调一致完成共同的任务。解决这个问题的标准办法是进程间的通信（IPC）机制。

不管你选用哪一种 IPC 机制，当普通进程协同工作时，仍然会有环境切换的开销。

克服这一问题的办法要用到线程（thread）的概念。每个 Linux 进程都有它自己的正文，数据，系统的内存区，数据结构以及它自己的处理机时间片。所以，每个进程可以按自己的步调独立运行。线程的概念是每个进程可以有若干并行的线程共享进程的时间片。每个线程还可以共享进程的内存和数据结构。同一进程内的两个协同工作的线程的环境切换工作量远小于进程之间的环境切换工作量。主要因为进程内的线程共享进程的正文，数据，系统内存以及数据结构，所以不需要为重新安排这些事情而付出代价。

由于线程设计成能协调一致进行工作，它们之间不会发生冲突，内核不需要做什么事情来保护进程内部的线程，使它们互不干扰。

因此，线程可以看作是轻量级的进程。轻在它们之间的环境切换工作量，特别是和标准的（或重量级的）的进程的环境切换工作量相比更是如此。

一般可以用两种不同方式实现线程：

- 在内核中，作为内核的任务；
- 在进程本身，作为用户空间的任务。

当在内核中实现线程时，线程的调度可以是抢先的，类似于进程本身的调度。这就是说，不管线程当时在做什么，当分配给它的时间用完时，内核将中断它的执行。将控制转移给另一个线程，使它接下去运行。

在用户空间实现线程时，实现的方法要简单得多。现在线程的调度变为非抢先调度。这就是说，线程之间的环境切换只发生在线程自动放弃对 CPU 控制的时候。

在多用户的环境中通常避免使用非抢先调度，因为它允许一个用户拒绝放弃对 CPU 的控制，独占处理机不让其他用户使用。

在线程的情况下，非抢先调度不会出现这样的问题。因为假定用户进程的线程协调一致完成共同的任务，并非互相争夺处理机。

用户级线程实际发生的唯一问题是输入/输出阻塞（blocking I/O）。内核级线程可以避免发生这样的问题。使用非抢先调度，如果其中一个线程被阻塞在输入/输出上（例如，等待键盘的输入），这一线程不能释放对 CPU 的控制让其他线程运行，实际将导致一个进程中的所有线程都停止工作。直到输入/输出阻塞结束，控制才能被转移。这说明在使用用户级线程时必须谨慎，不让输入/输出阻塞线程的情况发生。

Linux 系统中有几个软件包可以用来实现线程。一般说来，它们比较复杂不适合我们当前的教学需要。所以，我提供一个新的，非常简单的用户级线程库。可以将它们结合在自己的程序中使用来体会一些线程的概念。

29.1 库调用

小型线程库实际非常简单，特别是它的用户接口。线程是根据实际需要在运行时动态地建立的。可以调用 new_thread ( ) 函数来建立线程：

```
int new_thread (int ( *start  addr) (void), int stack_size)
```

其中 start_addr 是指向一个函数的指针。这一函数对新建的线程的作用就象 main ( ) 函数对进程的作用一样。stack_size 参数用来指定线程作堆栈用的以字节计的内存空间。当 new_thread ( ) 函数被调用时，将建立新的线程结构，对它进行初始化，并将它加到双向连接的循环调度表中。当前的线程放弃对 CPU 的控制时，线程环境切换程序 (或调度程序) 用这个表来确定下一个执行的线程。

通常，从进程的 main ( ) 函数内部发出对 new_thread ( ) 函数的第一次调用。发出调用后，main ( ) 函数自身被挂起，然后启动线程调度程序。仅当在特定进程内部建立的所有线程的工作结束时，控制最终才回到 main ( ) 函数中 new_thread ( ) 调用下面的指令。

除第一次以外，所有对 new_thread ( ) 的调用都立即返回。所以，发出调用的线程可以继续向下执行。在出错时 (如 malloc ( )分配内存失败)，从 new_thred ( ) 函数返回的值为 0。成功时，返回值为 1。

另一个用到小型线程库的函数是 release ( )，它使线程放弃对 CPU 的控制，使控制回到线程调度程序：

```
int release(void);
```

调用这一函数后将执行线程调度程序，它又将 CPU 的控制权转移给由 new_thred ( ) 建立的循环调度表中的下一个线程。可以建立一个简单的预处理程序(preprocessor)，将 release ( ) 函数自动加到线程代码中所有必要的位置。虽然在这里没有包括这一内容；但做到这一点并不困难。从安全的角度考虑，应该将 release ( ) 函数加到线程中任何可能出现循环的位置。这样做即使出现无限循环，也不会妨碍对其他线程的调度。根据这一规则，需要加 release ( ) 函数的位置有：每个 while，for 和 do 循环的循环体内部；每个标号下面以及每个函数的开头。这样做即使遇到递归函数也不成问题。

在两个或多个线程需要相互通信的情况下，可以使用进程内的全程变量(因而对每个线程都适用)。注意：一个进程内的所有线程共享程序和数据存储区，每个线程唯一的专用存储区是它的堆栈，它不用来保存全程变量。

在线程之间经常用全程变量进行通信。但有不同步的问题。如果不采取特殊措施，一个提供数据的线程将数据写到全程变量后，无法知道使用数据的线程是否已经读取了数据。可能又将下一个数据写到全程变量中去。

为简化线程的编写工作，假定它们之间的通信是同步进行的。小型线程库在约会(rendezvous)概念的基础上提供了两个线程间的简化通信机制。约会的基本概念是：不管通信伙伴中那一个先到达约会点，它的执行将被挂起，直到另一伙伴到达同一点为止。这时候恢复被挂起的线程的执行，两个线程传送数据。在保证通信完成后，两个线程继续执行。

小型线程库提供了有关约会机制的3个函数调用：get_channel ( )，send ( ) 和 receive ( )。

get_channel ( ) 函数的格式如下

```
int get  channel (int number);
```

其中 number 是你愿意使用的通信通道号，返回值是通道描述符。它的值将向前传给其他与约会有关的调用。在出错时，返回0值。除了用通道号代替文件名以及返回值是通道描述符而不是文件描述符以外，这一函数在概念上和打开文件相似。

第1个对特定通道号的 get_channel ( ) 调用将动态地建立通道，并返回通道描述符。其后的对同一通道号的 get_channel ( ) 调用请求，只送回已有的通道描述符，不再建立通道。

通道的建立方式容许多个产生数据和使用数程的线程共享同一通道的存取。在这种情况下，线程库应该保证生产数据的线程对产生的信息正确地进行排队，而且每条信息只允许一个使用数据的线程进行读取，并以先进先出的方式将信息从通道中移走。

取得了通道描述符，信息就可以用 send ( ) 和 receive ( ) 通过通道进行通信：

```
int send (int cd, char *buf, int size);
int receive (nt cd, char *buf, int size);
```

其中 cd 是从 get_channel ( ) 调用返回的通道描述符；buf 是指向缓冲区的指针，缓冲区用于传送或接收数据；而 size 则是要传送的以字节计的数据容量。

如果一对 send ( ) 和 receive ( ) 调用指定的数据容量不相同，将使用其中较小的一个。send ( ) 和 receive ( ) 调用的返回值是实际传送的字节数。

29.2 调用 C 函数

为了了解小型线程库的操作，先需要了解 C 程序如何实现函数调用，以及它们如何处理传递的参数值。

每个 C 程序都从执行 main ( ) 函数开始。事实上，当编译和连接 C 程序时，一段系统专用的代码插在程序的前面。它用来处理任何命令行参数，并完成对程序的 main ( ) 函数的调用。函数调用和机器代码中的子程序调用一样，将返回地址推进机器的堆栈，然后将控制转移到被调用函数的起点。当函数结束运行时，堆栈顶部的返回地址用来返回调用程序中紧接着函数调用语句后面的指令。

事实上，堆栈不仅用于保存函数的返回地址，也用来传递函数的参数值，并用来保存函数内部说明的局部变量的值。所以必须小心对待堆栈中存放的值，因为任何类型的堆栈混乱将导致难以对付的隐错。

当调用一个函数时，要传送给函数的任何参数表达式必须先进行计算。然后将它的值推

进堆栈。其次将返回地址推进堆栈，并对函数发出机器代码的调用指令。在被调用函数的前面，指定了任何参数变量的名称。这些名称用来标记已经推入堆栈的参数值的位置。用这种方式，任何参数值将自动地赋予它们相应的参数变量。

一旦进入了函数内部，处理机的一个寄存器（扩充基地址指针寄存器，ebp）的值也被推入堆栈。这一寄存器将用作位置保存器。当函数结束时，用它的内容在堆栈中找到函数的返回地址。在进入函数时，ebp的值被推入堆栈中返回地址下面的一个单元中，并将当前堆栈的指针值（指向堆栈中刚推入的ebp值）复制到ebp寄存器，复盖它原有的，现在已安全保存在堆栈中的值。

在处理了ebp的值以后，将堆栈指针值减1，使堆栈留出空间用来存放函数体内说明局部变量。将堆栈指针值减1是在堆栈内留出空间的正确操作。因为堆栈是从内存的高地址向下增长的。当将数据推入堆栈时，堆栈的指针值减1。当将数据从堆栈中弹出时，堆栈的指针值加1。

当函数结束时，在堆栈中所有和函数有关的数据都应删去，使堆栈回到调用函数以前的状态。这由下面的4步来完成：

- 第1，将当前的ebp值复制到堆栈的指针，这将自动删除留给任何局部变量的堆栈空间，使堆栈指针指向保存的ebp值。
- 第2，将堆栈上托送入ebp寄存器，使ebp恢复保存的值，并使堆栈指针指向调用函数的返回地址。
- 第3，执行返回指令，并将执行的控制权转移给调用函数。堆栈指针指向任何传送给被调用函数的参数值。
- 第4，回到调用函数，正确提升堆栈的指针值，使它高于任何推入的参数的数目，这等效于从堆栈中删去所有的参数值。

为了说明执行程序时堆栈的操作，我们看堆栈的两个瞬态图。它们取自下面两个简单程序的执行过程：

```
main()
{
    int x, y;

    x = 6;
    /* snap-shot ont here */
    y = twice(x);
}

twice(int n)
{
    int r;

    r = 2 *n;
    /* shap-shot two here */
    return r;
```

第 1 张格式图（图 29.1）取自 main（）函数内部，在调用 twice（）函数之前。它表示调用 main（）函数后它的返回地址被推入堆栈，接着将 ebp 的原始值推入堆栈，并将 ebp 寄存器的值设为指向这一堆栈位置，下面是留给局部变量 X，Y 的位置。当前堆栈指针的位置，以及赋予变量 X 的值 6。

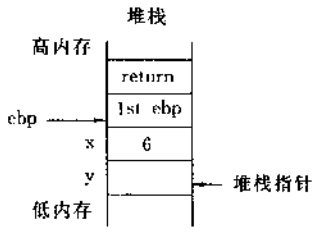

图 29.1　在 main（）函数内调用 twice（）函数前的堆栈格式

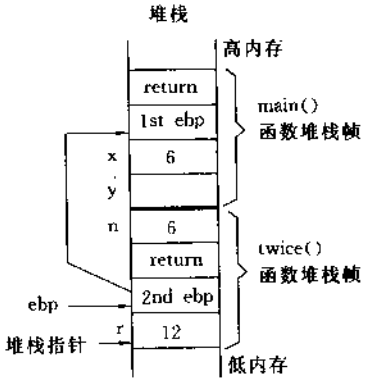

图 29.2　在 twice（）函数内返回前的堆栈格式

第 2 张格式图（见图 29.2）取自 twice（）函数内部，在执行 return 语句以前。它表示在 main（）函数的堆栈帧的下面，值 6 被推入堆栈中（作为参数值传送给 twice（）函数），并用名称 n 加以标记。在这以后，调用 twce（）函数的机器代码将另一个返回地址推入堆栈，接下去再次保存 ebp 的值，并使 ebp 的值指向第 2 次保存值的位置。用这种方式，ebp 寄存器等效于设置为指向所有保存在堆栈中的函数堆栈帧（stack frame）的连接表的头指针。最后则为变量 r 在堆栈上分配了位置，并在后面将计算值 12 赋值给 r。

29.3　线程调度

注意：一个进程内的所有线程共享程序和进程所有的全程环境。线程唯一的专用部分是它在执行时使用的任何重要的处理机寄存器和堆栈的内容。为了切换一个进程内的两个线程的环境，要做的只是交换少量寄存器的值。特别是程序计数器（或指令寄存器），ebp 寄存器和当前堆栈的指针。事实上，你很快将发现，所有实际需要保存的只有 ebp 的值，其他的值可以从它推算出来。

每个线程都有一个属于它的数据结构，当线程不是当前运行的线程时，它用来保存每个线程的环境。这一数据结构具有以下的形式：

```
struct context
{
    int ebp;
    char *stack;
    struct context *next;
    struct context *prev;
```

```
};
```

结构中的 next 和 prev 字段用来形成向前和向后的连接，将这一结构连接到环形的双向连接的结构表中，它就是循环调度表。stack 字段是指向字节数组的指针，这一数组是这一线程的专用堆栈空间。当 CPU 处理其他线程时，ebp 字段用来临时保存 ebp 寄存器的内容。

29.4 环境切换

我们讨论的线程库的第一段代码是 release ( ) 函数。当线程需要将 CPU 的控制权转移给循环调度表中下一个线程时，它将调用 release ( ) 函数。当你看这段代码时，注意 current 变量是指向当前运行的线程的 context (环境) 结构的指针：

```
release(.oid)
{
    /* 1 */
    if (thread_count <= 1)
        return 0;

    /* 2 */
    current = current->next;
    switch_context(current->prev, current);
    return 1;
}
static switch_context(struct context *from, struct context *to)
{
    /* 3 */
    __asm__
    (
        "movl 8(%ebp), %eax\n\t"
        "movl %ebp,(%eax)\n\t"
        "movl 12(%ebp), %eax\n\t"
        "movl (%eax), %ebp\n\t"
    );
}
```

注意 release ( ) 函数调用另一个函数 (switch_context( ))。它被说明为 static，所以它不能被线程库文件以外的任何程序调用。同时注意 switch_context ( ) 函数包含一些嵌入的汇编语言指令，它们用来直接访问特定的处理机寄存器。这是不能直接在 C 语言中使用的控制。这说明这一线程库不能移植到其他处理机上，因为它使用了 CPU 内部结构的特点。对编号的注释解释如下：

1. 如果循环调度表只有一个线程，就没有必要进行环境切换。在这种情况下，release ( ) 直接返回

当前线程继续执行，直到遇到下一个 release ( ) 函数为止。

2. 将 current 的值指向循环调度表中下一个线程，然后调用 switch_context ( ) 。用 current 的旧值和新值作为调用参数。
3. 在 switch_context ( ) 函数内部有 4 条 386 汇编语言指令。这些指令所做的工作是取 CPU 的 ebp 寄存器的内容，将它保存在 from 结构的 ebp 字段中。然后将 to 结构中的 ebp 字段的内容复制到 ebp 寄存器中，然后从 switch_context ( ) 函数返回。

为什么线程环境切换只围绕着变 ebp 的内容进行？为回答这一问题，需要回头看一下堆栈的情况。

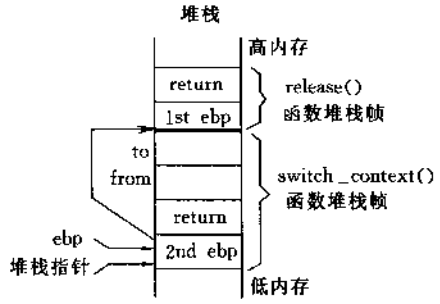

图 29.3 在环境切换前线程的堆栈格式

设想一个线程运行一段时间之后调用了 release ( ) 函数。出现的情况是这一函数的调用和接下来的 switch_contex ( ) 函数的调用都使用属于这一线程的堆栈。图 29.3 表示在执行 switch_context ( ) 函数的汇编指令前这一线程的部分堆栈的内容。

汇编指令部分将 CPU 的 ebp 寄存器的当前值保存在这一线程的 context 结构的 ebp 字段，然后将下一个要运行的线程的 context 结构的 ebp 字段的内容加载 ebp 寄存器。然后从 switch_context ( ) 函数返回。

最后，CPU 将执行了循环调度表中的每一个线程，直到遇到它的下一个 relaase ( ) 调用。所以，我们所观察的线程再次得到运行的机会。出现这种情况时，在循环调度表中我们前面的一个线程执行了 release ( ) 调用，这时 switch_context ( ) 的汇编指令已经执行，已用原先保存的我们线程的 ebp 寄存器值加载 CPU ebp 寄存器。注意，此时，堆栈的指针仍然指向前一个线程的堆栈，而程序计数器将指向 switch_context ( ) 函数后面的指令，而且将执行这些指令。

每个函数结束时，将堆栈指针设为 ebp 寄存器的值以删除任何局部变量。在现在的情况下，switch_context ( ) 函数没有局部变量，但仍然做了赋值工作，这样做的好处是在我们的线程的堆栈中将指针移回到正确位置。函数结束时的另一项操作是将 ebp 寄存器的值从当前堆栈(我们的线程的堆栈)上托，并利用堆栈中的返回地址返回调用函数。

这表示我们的线程从 release ( ) 调用正常返回并继续向下执行。事实上，任何线程常常调用 release ( ) 函数，然后从那里返回，允许这些线程继续往下执行。在调用 release ( ) 函数期间，线程本身并不意识到有什么不寻常的事情发生。

29.5 启动新线程

启动新线程包括建立线程的 context 结构，并分配一些内存作为它的堆栈。然后对这些数据结构进行初始化，并把它加到循环调度表中使它有机会运行。这听起来简单，但仍有一些难处理的地方。难在对线程堆栈的初始化。一些值需要仔细地送到空的堆栈中去，才能使用堆栈：

```
new_thread(int (*start_addr)(void), int stack_size)
{
    struct context *ptr;
    int esp;

    /* 1 */
    if (!(ptr = (struct context *)malloc(sizeof(struct context))))
        return 0;

    /* 2 */
    if (!(ptr->stack = (char *)malloc(stack_size)))
        return 0;

    /* 3 */
    esp = (int)(ptr->stack+(stack_size-4));
    *(int *)esp = (int)exit_thread;
    *(int *)(esp-4) = (int)start_addr;
    *(int *)(esp-8) = esp-4;
    ptr->ebp = esp-8;

    /* 4 */
    if (thread_count++)
    {
        /* 5 */
        ptr->next = current->next;
        ptr->prev = current;
        current->next->prev = ptr;
        current->next = ptr;
    }
    else
    {
        /* 6 */
        current = ptr->next = ptr->prev = ptr;
        switch_context(&main_thread, current);
    }
    return 1;
}
```

new_thread ( ) 函数取两个参数,第 1 个参数是这一线程开始执行的函数的地址,第 2 个参数是分配给新线程作堆栈用的以字节计的内存块的容量。对编号的注释说明如下:

1. 使用 malloc ( ) 函数为这一线程建立 context 结构。检验内存分配是否成功。出错时返回 0 值。
2. 再用 malloc ( ) 函数按指定的线程堆栈的容量分配供堆栈用的内存块。将新的 context 结构中的

堆栈指针设为指向新的堆栈。如果分配内存出错，new_thread ( ) 返回 0 值。

3. 将堆栈初始化，使它成为图 29.4 所示的状态。注意 context 结构的 stack 指针指向堆栈的最低的地址，而堆栈的操作则从高地址端开始。
4. 下一部分取决于这一线程是否是当前进程中的第 1 个线程。如果是，则 main 程序的执行将被挂起，将环境切换到新的线程使用的环境。如果不是，将新线程加到循环调度表中，等待运行的机会。
5. 将新线程加到已有的循环调度表有点复杂。因为循环表是双向连接的，需要设置向前和向后的连接指针。新线程以这样的方式插入循环调度表中，当当前的线程执行 release ( )调用时，它自动成为下一个运行的线程。
6. 如果当前没有线程在运行，需要第一次调用 switch_context ( ) 函数来建立循环调度表。为了保证 main 程序挂起，switch_context ( ) 调用将它的 ebp 寄存器的内容保存在称为 main_thread 的 context 结构中。这一结构不是循环调度表的一部分，所以，main 程序不会再执行。除非另有安排或者所有线程运行结束时再运行 main 程序。

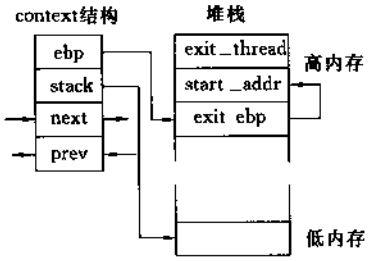

图 29.4 在 new_thread ( ) 函数内设置的数据结构

图 29.4 表明新线程的 context 结构中 ebp 元素指向线程的堆栈的一个单元，当切换到新线程的环境时，这一值将加载到 CPU ebp 寄存器。从switch_context( ) 函数返回时也使堆栈指针指向同一个堆栈地址。完成堆栈上托，用相应的值加载 bep 寄存器。同时使堆栈指针指向线程的 start 函数。最后从子程序返回，将堆栈上托将 start 地址送入程序计数器，因而完成到线程的 start 函数的转移。

当线程的 start 函数结束时，为了将线程从循环调度表中移去并释放分配给它的内存块还需要采取一些措施。这些操作由 exit_therad ( ) 函数完成。不必在每个线程结束时调用这一函数。转移到这个函数地址自动放在堆栈中。当 start 函数执行完毕后，用执行 start 函数同样的办法，自动执行 exit_thread 函数。

exit_thread ( ) 函数的代码如下：

```
static exit  thread(void)
{
    struct context dump, *ptr;

    /* 1 */
    if ( -- thread_count)
    {
        /* 2 */
        ptr = current;
        current -> prev -> next = current -> next;
        current -> next -> prev = current -> prev;
        current = current -> next;
        free(ptr -> stack);
        free(ptr);
```

```
        switch_context(&dump, current);
    }
    else
    {
        /* 3 */
        free(current->stack);
        free(current);
        switch_context(&dump, &main_thread);
    }
}
```

退出线程函数 exit_thread ( ) 结束 current 指针所指的线程的运行。由于 exit_thread ( ) 被说明为 static,防止了从线程外面调用这一函数。对编号的注释说明如下:

1. 如果 current 所指的不是循环调度表中最后一个进程,那么用调度程序将它从循环表中去掉,并将环境切换为下一线程的环境。否则由于没有更多的线程要执行,恢复执行 main 程序。
2. 复制指向当前线程的指针,将当前线程从连接表中去掉,然后使 current 指向下一个要执行的线程;释放运行结束的线程的内存块,最后用 switch_context ( ) 从当前线程环境切换到新的当前线程的环境。
3. 在这里,只要用 free ( ) 函数释放循环表中最后的线程的内存。然后用 switch_context ( ) 函数将环境切换为 main 程序的环境,然后恢复 main 程序的执行。

29.6 约会机制

两个新的数据结构用来实现约会机制。第 1 个是 channel 结构,每个信息通道一个。第 2 个是 message 结构,在分块的信息的基础上提供。这两种结构以及 context 结构和一些全程变量的说明等都由单独的头文件 def.h 提供,它应该用 # include 语句包含在程序的前面,def.h 的全部内容如下:

```
#include <string.h>

struct context                              /* One structure for each thread */
{
    int ebp;                          /* Base pointer (stack frame pointer) store */
    char *stack;                      /* Pointer to memory block for thread stack */
    struck context *next;                   /* Round robin circular list pointer */
    struck context *prev;                   /* Round robin circular list pointer */
};

struct channel                    /* One structure for each communication channel */
{
    int number;                                                /* Channel number */
    int sr_flag;                /* 0 = no queue, 1 = send queued, 2 = receive queued */
```

```
    struct channel *link;                    /* Link to next channel in list */
    struct message *message_list;                  /* Head of message queue */
    struct message *message_tail;                  /* Tail of message queue */
};

struct message                  /* One structure for each pending send/receive */
{
    int size;                                      /* Size of message in bytes */
    char *addr;                                  /* Pointer to start of message */
    struct message *link;                      /* Link to next message in queue */
    struct context *thread;                /* Which thread blocks on this struct */
};

static struct context main_thread;               /* Storage for main() details */
static struct context *current;                  /* Currently executing thread */
static int thread_count = 0;                    /* Number of threads to schedule */
static struct channel *channel_list = 0;               /* List of all channels */

static int switch_context(struct context *, struct context *);
static int exit_thread(void);
static int rendezvous(struct channel *, char *, int, int);
```

由于每建一个通道,分配一个 channel(通道)结构,并将它加到当前进程的所有通道的连接表中。当未配对的 send()或 recelve()对一通道进行操作时,分配一个 message(信息)结

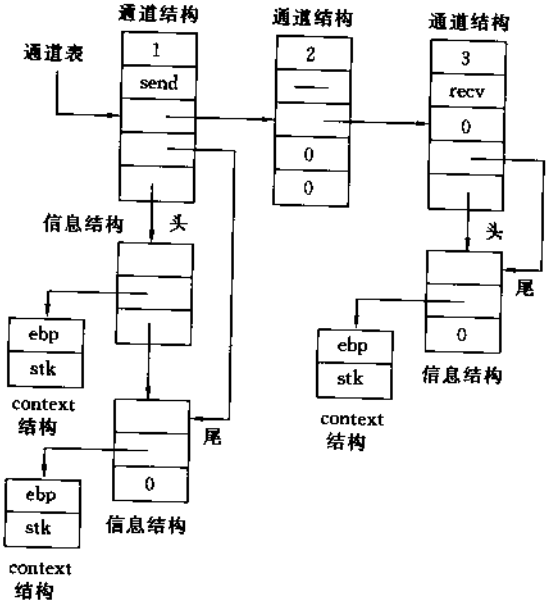

图 29.5 在约会过程中建立的结构连接

构,并将它加到挂在通道结构的 message 队列的最后,。当一个线程在约会过程由于等待通信伙伴而被挂起时,这一线程的结构从循环调度表中取出,并将它连接到它的 message 结构下面。图 29.5 表示当 send ( ) 和 receiv ( ) 操作暂时挂起时,不同的结构如何相互连接。

图 29.5 表明本例中有 3 个通道,它们的结构连接在一起。用 channel_list 作为指向表头的指针。对通道 1 进行了 send ( ) 操作,一个信息结构就加到这一通道结构的下面,执行 send ( )函数的线程的环境结构从循环调度表中取出,附加在信息结构下面。保证直到配对的 receive ( ) 函数对同一通道的操作执行以后,这一线程才能运行。这时,环境结构很容易找到,并将它重新连接到循环调度表中。

本图还表示另一个线程对同一通道进行了第 2 个 send ( ) 操作,它也在这一通道下排队,等待第 2 个 receie 操作。通道 2 建立了,但没挂起的信息队列。对通道 3 进行了 receive ( ) 操作,正在等待对这一通道的 send ( ) 操作。

29.7 建立通道

用 get_channel ( ) 函数建立新的通信通道:

```
get_channel(int number)
{
    struct channel *ptr;

    /* 1 */
    for (ptr = channel_list; ptr; ptr = ptr->link)
        if (ptr->number == number)
            return((int)ptr);

    /* 2 */
    if (!(ptr = (struct channel *)malloc(sizeof(struct channel))))
        return 0;

    /* 3 */
    ptr->number = number;
    ptr->message_list = 0;
    ptr->message_tail = 0;
    ptr->sr_flag = 0;
    ptr->link = channel_list;
    channel_list = ptr;
    return((int)ptr);
}
```

注意:get_channel ( ) 函数用通道号作为参数,返回通道描述符。如果通道还未建立,建立这一通道。编号的注释说明:

1. 从 channel_list 开始,向下搜索已有的通道连接表,检查指定的通道号是否已经存在。如果在表中找到了指定的通道号,将这一通道结构的地址送给一个整数变量,并将它作为通道描述符返回。
2. 如果指定的通道不存在,分配新的 channel 结构。在出错时,返回 0 值。
3. 在新的 channel 结构成功地建立后,对它的域进行初始化,并将这一结构加到通道表的前头,最后将新结构的地址送给整数变量,将它作为通道描符返回。

29.8 send 和 receive 函数

一旦取得了通道描述符,剩下的事是用 send ( ) 和 receive ( ) 函数在线程间传递信息。事实上,正像下面的程序所表明的,send ( ) 和 receive ( ) 是完全对称的操作。两者都在内部调用 rendezvous 函数,只是用另外的标志参数来指定数据的传送方向:

```
send(int chan, char *addr, int size)
{
    /* 1 */
    return(rendezvous((struct channel *)chan, addr, size, 1));
}


receive(int chan, char *addr, int size)
{
    /* 2 */
    return(rendezvous((struct channel *)chan, addr, size, 2));
}


static int rendezvous(struct channel *chan, char *addr,
int size, int sr_flag)
{
    struct message *ptr;
    int nbytes;

    /* 3 */
    if (sr_flag == 3 - chan->sr_flag)
    {
        /* 4 */
        ptr = chan->message_list;
        chan->message_list = ptr->link;
        ptr->thread->next = current->next;
        ptr->thread->prev = current;
        current->next->prev = ptr->thread;
        current->next = ptr->thread;
        ++thread_count;
```

```
    /* 5 */
    nbytes = (size < ptr -> size)? size:ptr -> size;
    ptr -> size = nbytes;

    /* 6 */
    if (sr_flag == 1)
        memcpy(ptr -> addr, addr, nbytes);
    else
        memcpy(addr, ptr -> addr, nbytes);

    /* 7 */
    if (! chan -> message_list)
        chan -> sr_flag = 0;

    return(nbytes);
}
else
{
    /* 8 */
    ptr = (struct message *)malloc(sizeof(struct message));

    if (! chan -> message_list)
        chan -> message_list = ptr;
    else
        chan -> message_tail -> link = ptr;
    chan -> message_tail = ptr;
    ptr -> link = 0;
    ptr -> size = size;
    ptr -> addr = addr;
    chan -> sr_flag = sr_flag;
    ptr -> thread = current;
    current -> prev -> next = current -> next;
    current -> next -> prev = current -> prev;

    /* 9 */
    if ( -- thread_count)
    {
        current = current -> next;
        switch_context(ptr -> thread, current);
    }
    else
        switch_context(ptr -> thread, &main_thread);

    /* 10 */
```

```
        nbytes = ptr->size;
        free(ptr);
        return(nbytes);
    }
}
```

send ( ) 和 receive ( ) 函数都用 3 个参数,第 1 个参数是通道描述符,第 2 个参数是发送或接收数据的缓冲区,第 3 个参数是传送数据的字节数,两个函数的返回值是实际传送数据的字节数。它对应于两个函数的第 3 个参数数值中较小的一个。编号注释的说明:

1. send ( ) 函数的参数值直接送给 rendezvous ( ) 函数,增加了第 4 个参数用来说明数据传输方向。
2. receive ( ) 函数的参数值直接送给 rendezvous ( ) 函数,增加了第 4 个参数用来说明数据传输方向。
3. 在 rendezvous ( ) 函数内部,首先检查是否有正确类型的通信伙伴在排队等候,如果有,将通信伙伴的 context 结构放回循环调度表中,并在两个线程恢复执行之前完成数据复制工作,如果没有伙伴在等待,将当前线程从循环调度表中撤下,并将通信请求加到这一通道的信息队列中。
4. 这段代码将线程的 context 结构重新加入循环调度表中,并将准备好运行的线程计数值加 1。
5. 这一计算用来确定传送数据的字节数,并将这一数值保存在适当位置,使通信伙伴运行时能取到它。
6. 现在进行数据传送,保证按正确方向传送数据。
7. 如果这是通道的队列中最后一条信息,将信息的类型标记复位,并将实际传送数据的字节数返回给调用程序。
8. 在这段代码中,由于还没有配对的伙伴,一条新的信息需要进行排队。分配新的 message 结构,并将它加到通道的信息队列中。等待通信伙伴的线程将被挂起。这段代码将这一线程的 context 结构从循环调度表中撤下,并将它放在信息结构队列的后面。
9. 将可运行的线程数减 1,对循环调度表中下一个线程完成环境切换。由于新挂起的线程需要找到通信伙伴,应该有其他线程接过控制权。如果找不到其他线程,那么将环境切换为 main 程序的环境,并恢复它的执行。
10. 在发生约会(rendezvous)以后,恢复并返回传送的字节数,释放信息结构,这一线程下次将得到调度。

29.9 无界缓冲区

作为应用小型线程库的一个简单的例子,下面的程序用简练的方式实现了无界缓冲区:

```
int buffer(void);
int start(void);

int ch_desc1, ch_desc2;

main(void)
{
    ch_desc1 = get_channel(1);
    ch_desc2 = get_channel(2);
    new_thread(start, 1024);
```

```
}

start(void)
{
    int i, n;
    new_thread(buffer, 1024);

    for (i = 0; i<10, ++i)
    {
        send(ch_desc1, &i, sizeof(int));
        release();
    }

    for (i = 0; i<10, ++i)
    {
        receive(ch_desc2, &n, sizeof(int));
        printf("i = %d n = %d \n", i, n);
        release();
    }
}

buffer(void)
{
    int i;

    receive(ch_desc1, &i, sizeof(int));
    new_thread(buffer, 1024);
    send(ch_desc2, &i, sizeof(int));
}
```

我将简单说几点,其余留给你自己去解释。你将发现,这里用了两个通道进行多重传送和接收。同时你将发现程序中 start( ) 函数只用一次,而多次运行 buffer( ) 函数。同时注意:这一解决方案如何根据需要动态地建立更多的缓冲区空间,又如何动态地清除这些空间。

第三十章　内 存 管 理

Linux 内存管理是内核最复杂的任务之一。主要因为它用到许多 CPU 提供的功能，而且和这些功能关系密切。因此，在我们讨论内存管理时，先要讨论 CPU 及其在内存管理中所起的作用。

30.1　受保护的编址方式

CPU 执行的许多指令需要访问内存。看起来很简单，在机器代码指令中指定要提取或修改的内存单元的地址就行了。如果事情是这么简单就好了。

当需在内存中同时运行许多进程时，使每个进程都好像在它们自己的机器上运行一样就方便了。做到这一点的办法是在 CPU 内部完成不同形式的地址转换，将指令指定的地址（称为逻辑地址或有效地址）转换为硬件对实际内存的访问。

进程按指令、数据和堆栈分成若干段。每段都有段描述符（segment descriptor）。每一段描述符包含 8 个字节。内容包括段开始的基地址，段的容量和段的访问权限。段的描述符集中放在段描述符表中。一个 CPU 内部寄存器保存访问段描述符表的基地址。

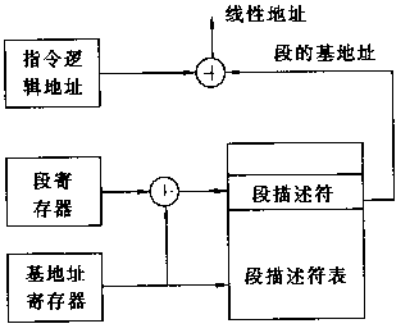

图 30.1　线性地址计算

除此以外，CPU 还包含一组段寄存器。每个寄存器指向描述符表中的一个段描述符项。如果发生了任何类型的内存访问，将选择一个 CPU 段寄存器（或者由访问内存的类型隐含说明，或者由访问内存的指令明确说明）用来进行地址计算。计算的结果得到线性地址。图 30.1 表示在给定逻辑地址和段寄存器的情况下如何进行线性地址计算。

这里说的是每一次内存访问，CPU 都用一个内部寄存器的内容找到描述符表的基地址。基地址和一个段寄存器的内容（它的最低 3 位的值为 0）相加。相加的结果是指向描述符表的一个 8 字节的项的指针。注意：段的描述符项包含段在内存中的基地址，段的容量和对本段的访问权限。

访问内存的指令本身也提供一个地址，这个地址是已取得段描述符的内存段内部的逻辑地址（或有效地址）。指令的逻辑地址和段的基地址相加得到线性地址。必须对线性地址进行检查，保证它落在段的范围内（即不超出段的容量），并检查是否允许在段内进行请求的访问类型（读、写或执行）。如果这些检查被通过，CPU 现在有了用来访问内存的地址。

我们已经看到的段描述符主表称为全程段描述符表或 GDT，而指向 GDT 起点的 CPU 寄存器称为 GDTR。在多任务的环境中，可能愿意每个进程访问它自己的内存段。为满足这一要求，CPU 也为每个进程提供另一个描述符表，称为局部描述符表或 LDT，它通过 GDT 的项和处理机内部寄存器 LDTR 进行访问。

如果每次内存访问都要查找描述符表（它本身又在内存中）的值，这将消耗大量的 CPU

时间。为了大大加速这一过程，CPU 内部有一些隐藏的，不能由程序直接访问的寄存器。这些寄存器用作快速缓存，保存当前每个段寄存器所指的段描述符的值。只在有关的段寄存器的值改变时，或者描述符表的 GDTR 或 LDTR 的基地址改变时，才用相应的段描符值加载隐藏的寄存器。

30.2 分页技术

在前面讨论的组织内存访问的功能的基础上，可以建立起一个完全切实可行的多任务系统。只要在开始运行时每个进程知道它实际需要的内存容量，一开始就可以为这一进程在合适的位置分配它所需的内存块。在进程存在期间也不用再改变内存的分配。然而，Linux 系统要比上面的简单模型更为灵活。例如，它允许进程不必考虑周围实际还有其他进程存在的现实而增加它的内存要求。Linux 系统还允许同时运行比实际内存能容纳的更多的进程。

Linux 之所以能这样做，因为它利用了处理机提供的分页（ paging ）功能。在 CPU 中是否进行调页由处理机控制寄存器中的 1 位来进行控制。如果调页位设置，32 位的线性地址将一分为二。高 20 位用作页号，而低 12 位用作页内的位移地址。这表示 32 位的线性地址可以用来访问 100 万 4KB 页。分页所做的主要工作之一是将线性地址中提供的 20 位页号转换为对应于机器实际的 4KB 内存页的另一个 20 位页号，结果是任何实际 4KB 内存页可以出现在处理机线性地址空间的 100 万 4KB 页边界内的任何位置。地址变换用内存中的对照表完成，其中线性地址的高 20 位用作大的 32 位元素的数组的索引。从数组取得的 32 位值提供所需的 20 位转换值，其余 12 位则用来指定类似下面的一些事项：

- 本页允许读、写还是只允许读访问？
- 本页是用户进程访问页还是管理进程访问页？
- 是否对任何页内地址进行了读或写访问？
- 是否对任何页内地址进行了写操作？
- 本页在内存中还是在磁盘上？

用两级页表取代单个大的页表可以不必用很大的连续线性地址空间来保存页表。两级页表将线性地址的高 20 位划分为两个独立的 10 位值。CPU 的 3 号控制寄存器包含内存中用作第一级页目录表的 4KB 页的地址。每一 4KB 页分成 1024 项，每项占 32 位。线性地址中页号的前 10 位正好用来选择页目录中的 1024 项。被选择的项中的前 20 位用来选择页表中的另一 4KB 页。这一页同样包含 1024 项，每项 32 位长。线性地址页号的后 10 位正好用来选择页表中的项。这样选出的 32 位包含 20 位实际页号，以及前面讨论过的 12 位。

图 30.2 表示从线性地址到实际地址的转换。同样，如果每一线性地址都要进行这样的转换，需要额外增加两次读页目录和页表的内存访问。这是非常耗费时间的操作。解决的办法和前面一样，用 CPU 寄存器保存最近的地址转换，使得查找工作快得多。

30.2.1 段的增长

当你希望从包围在中间的内存段（ segment ）中提高内存容量时，分页技术显得特别重要。图 30.3 说明这种情况。图中（a）表示两个由若干页组成的内存段映射到实际内存页的

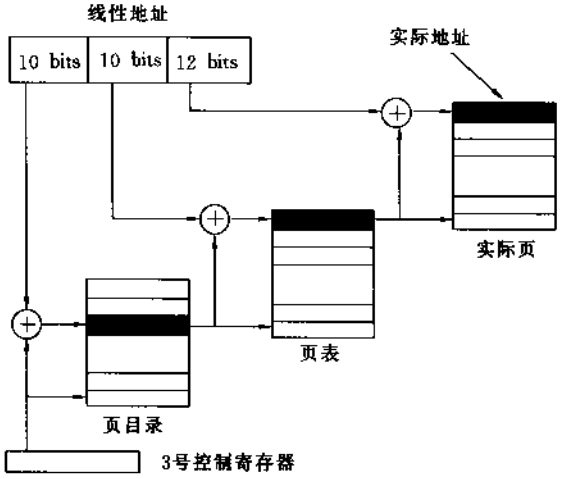

图 30.2　分页技术如何将线性地址转换为实际地址

情况。其中内存段 1 的页被内存段 2 的页所包围。不使用分页技术,就不能增加内存段 1 的容量,除非将整段移到足以容纳该段的新的连续的实际内存区。

图 (b) 表示使用分页技术,增加段的容量不成问题。因为在线性地址空间连续的段 1 的页面不需要在实际内存中连在一起。所以,可以分配任何空闲的实际内存页作为段 1 的新页 (线性页 3),只要正确地进行映射即可。

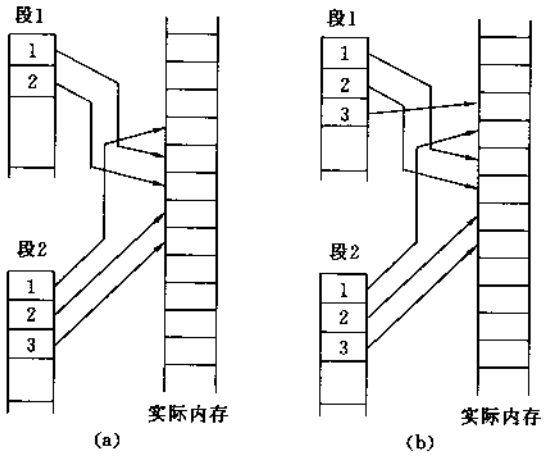

图 30.3　分页技术使内存段容易增长

30.2.2　交换

分页的另一个主要用途称为交换(swapping)。交换概念的实质在于允许在系统中同时运行比内存实际能容纳的更多的进程。这包括另外设置交换区 (通常在磁盘上) 作为内存的溢出区。当需要启动一个新进程而内存中又没有足够空间允许这样做时,将调用交换程序,它将选择内存中的一个或几个进程将它们写到交换区中,留出内存空间供新进程装入和运行。当进程运行一段时间后,一些进程因等待输入/输出而挂起,另一些进程将结束,这就给交换出

去的进程调回内存的机会（也可以将其他进程交换出去作为代价），使它们能继续执行下去。

利用分页技术实现交换功能，表示不需要将整个进程交换到磁盘上，只要留出足够的页面供新进程运行就行了。事实上，Linux 做得更好，因为它使用了称为按需调页技术（demand paging）。为了实现按需调页技术，页表中的描述符项需要提供说明本页的内容是否实际装入内存的信息。有了这一信息，当 CPU 试图访问的页不在内存时，将产生异常信息（或称页面故障）。这一异常信息将导致执行异常处理程序，它将所需的页面装入实际内存。导致页面故障的指令将再次被执行，这一次它的内存访问将是成功的。

利用分页技术将新进程装入内存是好的设想。Linux 实际也是这样做的。出现的情况是当运行的进程试图在没有装入内存的地址上执行指令时，将出现页面故障，对它进行处理后，进程又可以继续执行下去。

对执行中的程序进行的研究发现：对一个适度大小的程序讲，在任何特定的运行时刻，大部分代码并未得到执行。使用按需调页技术，它的含义在于当进程所需的页面调入后，它将以少量的页面故障稳定运行一段时间，虽然它的大多数页面完全没有装入内存。即使特定的程序在运行时访问所有的程序页，但在任何时刻，在一段时间内进程只用其中少量的页面，所以，从资源利用的角度看，分页技术能取得很大的效益。

30.3 Linux 特色

使用 fork( )系统调用是 Linux 产生新进程的唯一办法。它在开始时分配两页内存。一页用作新进程的任务结构，另一页当运行新进程时用作内核堆栈。

其次，从双亲进程的任务结构复制子进程的任务结构，并进行一些必要的修改。如进程的 ID 以及对话过程的标题等。最后，将所有的进程信息，包括内存映射复制到新的子进程。子进程有它自己的局部描述符表（LDT），因此有它自己的正文，数据和堆栈段。也有它自己的页表。开始时只是双亲页表的副本。从概念上讲，fork( )调用用复制双亲进程的正文，数据和堆栈段的办法来产生子进程。然而从时间的角度看，这是代价高昂的操作。特别是子进程肯定立即调用 exec( )系统调用，将这些段的内容换成属于新进程的内容。为克服这种浪费，Linux 只从双亲进程复制页表给子进程，而不是复制它们访问的页。

对正文段来讲，这项技术是好的，因为正文段标记为只读。只进行读操作不会影响它们共享程序，而数据和堆栈段则允许进行读和写。Linux 解决问题的办法称为在写时才进行复制（copy on write）。当其中一个进程要写数据到当前共享的数据或堆栈页时，这一页才被复制，并对页表作相修改，使得每个进程都有这一页的副本。所以一个进程的写操作不会对另一个进程产生影响。当用 exec( )系统调用来启动新进程时，使用按需调页技术来装入进程当前所需的页面。许多情况下，需要将新的内存页分配给进程，最后用完内存中空闲的页面。在这种情况下，Linux 试图从内存中其他地方巧妙地取得内存页（必要时将一些页写到交换区中）来满足对内存页的请求。页面的来源包括降低用于磁盘快速缓存的内存页数，因为快速缓存和普通进程使用的内存之间的边界是动态的。如果需要也可以从进程内存区取得以满足内存页的请求。

从某一进程的正文段中选取内存页有它的好处，因为正文段只用于读，所以它不会被改变，在释放它以前不必将它写到交换区中去，需要调入时；可以从原始的可执行文件中调入。

第三十一章　文 件 存 储

为了在 Linux 系统中访问文件系统，需要将文件系统装配到目录层次结构中。这是用 mount 命令完成的。Linux 系统可以使用许多类型的文件系统。这些文件系统的全部或任何一部分可以任意组合进行装配，几乎没有限制。

31.1　VFS

虚文件系统转换（Virtual Filesystem Switch，VFS）允许 Linux 装配和访问所有不同类型的文件系统。VFS 的主要任务是在 Linux 文件存储系统中有效地增加间接访问层，将文件系统的调用转换为对相应的子程序的调用，这些子程序用准确的文件系统类型的代码写成。

当前，Linux 通过 VFS 支持许多类型的文件系统，其中主要有：

minix	这是 Linux 最早支持的文件系统类型，也是它当时支持的唯一的文件系统。它的两个主要的缺点是：它只支持最大 64MB 的磁盘分区和最长 14 个字符的文件名称。
ext	Linux 很快增加了扩充文件系统（ext）。它克服了 minix 文件系统的缺点，最大支持 2GB 的磁盘分区，并允许使用长达 255 字符的文件名称。然而，它在功能和性能两方面各有它自己的问题，所以很快被其他文件系所超越。
xiafs	它的主要意图是取代 ext 文件系统。它是在 minix 文件系统的基础上发展起来的，它克服了 minix 文件系统的缺点，并在功能上作了扩充。但 xiafs 很快被功能和性能大为改善的文件系统所取代。
ext2	第 2 扩充文件系统（在我写本书的时候）几乎成为普遍接受的 Linux 标准文件系统。它可以移植到其他系统上。本系统允许磁盘分区的容量达到 4TB，文件名称长达 255 字符。
System V	这是 UNIX 系统早期支持的文件系统。现在也可以在 Linux 系统中使用，这是为了与已有的 System V 系统兼容。除非兼容性成为突出的问题，一般情况下用得不多。它也有 minix 文件系统同样的限制。
NFS	网络文件系统（NFS）原先由 SUN 公司开发。当计算机与网络连接时，允许将一台机器上的文件系统装配到网上另一台计算机的目录结构中。尽管速度有少量的损失，网络连接是透明的，并允许整个网络共享磁盘分区和文件。
ISO 9660	这是光盘使用的文件系统，它已在 Linux 系统中实现。一张光盘能存高达 650 MB 的数据，而且现在有许多有用的 Linux 光盘产品。标准的 ISO 文件系统具有类似 DOS 的文件名称格式，但对这一标准扩充后，允许使用常用的 UNIX 系统的长文件名称。
/proc	这是建在 Linux 内核中的特殊文件系统类型。它允许对系统的不同部分和它的进程进行访问。
msdos	标准的 DOS 磁盘和分区可以装配到 Linux 系统中，并可以对它们进行访问。系统力图使它的外在表现尽量类似 UNIX，但仍然受到一定限止。如文件名称和格式等。
UMSDOS	这一文件系统允许你不必重新对磁盘分区进行化格式就能在 DOS 的环境中使用 Linux。它允许你将普通的 Linux 文件放到 DOS 分区中，并能像在普通的 Linux 文件系统中一样使用这些文件。这是利用索引文件在标准的 Linux 文件名和有限制的 DOS 文件名之间提

供透明的对照表实现的。

为了理解文件系统设计过程中的问题和选择，我们将对 Linux 支持的一些文件系统作一些简单的说明，以便了解这一领域中发生的变化。

31.2 System V

标准的 UNIX System V 文件系统配置的磁盘分区的格式非常简单，见图 31.1。

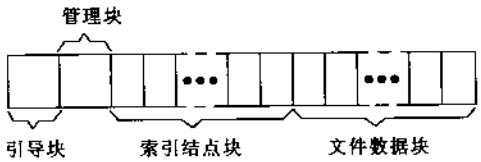

图 31.1 System V 磁盘分区格式

每个磁盘分区分成 4 个部分：

boot block（引导块） 如果这是用于引导系统的分区，它将包含引导机器的代码。否则，虽然分配引导块，但不使用它。

super bolck（管理块）本块保存这一分区的文件系统的管理信息，我们将在后面作更详细说明。

inode block（索引结点块） 这些磁盘块用来保存这一文件系统中的所有索引结点。一旦在这一分区建立了文件系统，分配给索引结点的磁盘块数（因而这一文件系统可用的索引结点数）是固定和不能改变的。

file data blocks（文件数据块） 分区中其余的磁盘块用来保存这一文件系统中的文件的实际数据。这些块按实际需要动态地分配给文件。

对索引结点和数据块来讲，需要用某种方法表示它是空闲的，正在等待分配；还是已经分配，在使用之中。system V 文件系统的管理块保存了空闲的索引结点和数据块的表。虽然两者的保存方式不同，但在需要时都可以很快进行分配。

31.2.1 空闲数据块表

一般说来，通过简单地检查数据块的内容来确定数据块是否在使用是行不通的，这是因为数据块的所有字节都用来保存文件的内容，而文件的内容可以是任何字节序列。这表示不能在数据块内保存特定的字节序列，用来指示这一块是空闲可供分配的。必须用某种数据结构列出空闲的数据块。

事实上，这是用连接表的形式实现的。连接表的前面部分保存在文件系统的管理块中。在磁盘分区中的数据块按顺序进行编号，每一数据块都有唯一的块号。当文件系统刚建立时，所有数据块都是空闲的，应该都出现在空闲数据块表（free data block list）中，然而，在管理块中用于空闲数据块表的空间有限，只能将表的很少一部分放在管理块中。表的其余部分存在一个空闲的数据块中。这一数据块的块号对应于管理块中空闲数据块表中最后的一个块号。重复使用这一概念，可以用若干空闲数据块来保存空闲数据块表。并使每个数据块保存的最

后一个块号指向下一用于空闲数据块表的数据块。这一概念表示在图 31.2 中。

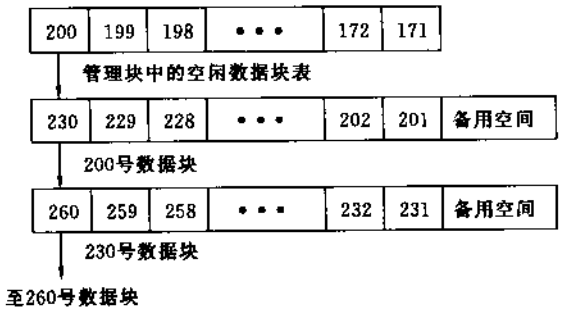

图 31.2　最初的空闲数据块表

使用这一方案,当请求新的数据块时,管理块中的空闲数据块表中下一空闲数据块将被分配（图 31.2 中,分配的顺序自右至左）。当请求更多的数据块时,重复这一过程,直到管理块中的空闲数据块表只剩最后一项为止。注意:管理块中的空闲表的最后一个数据块（200）包含下一部分空闲数据块表,因此在这一块被分配以前,应将它的内容先复制到管理块中。

图 31.3 表示当前管理块中最初的空闲数据块表中的所有空闲块都被分配时,在数据块 200 作为空闲块分配之前,将它的内容复制到管理块中。

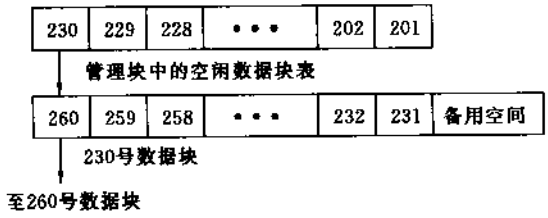

图 31.3　数据块 200 分配后的空闲数据块表

当文件被删除时,文件使用的数据块变为空闲,这些空闲块的块号将被加到管理块的空闲数据块表中。当它被填满时,另一数据块变为空闲（譬如说 188),此时已没有空间来保存这一块号,在这种情况下,将管理块中的空闲数据块表复制到新释放的数据块(188)。这时管理块中的空闲表变空,新释放的数据块(188)成为表中最左面的唯一的一项。

31.2.2　空闲索引结点表

由于一些原因索引结点的分配不同于磁盘数据块的分配。因此,空闲索引结点的保存格式不同于空闲数据块。不同的主要原因有:

- 索引结点不大,它的结构中没有足够的空间来保存大的空闲索引结点表。它和数据块的情况不一样,如果需要,整个数据块都可以用来保存空闲表。
- 作为数据块,无法用检查它的内容的办法来确定它是否空闲。这表示所有空闲的数据块的块号必须加以保存,否则就会丢失。作为索引结点,当它为空闲时,它有相应的内部标记。这表示只要对索引结点数组作一次线性扫描,总能找到空闲的索引结点。
- 一个文件通常有许多数据块,因此数据块的分配的频繁程度远高于索引结点。低效率寻找索引结点要

比低效率寻找数据块更容易被接受。

考虑到这些点，每次出现对新的索引结点的请求时，内核可以对索引点数组进行线性扫描来找出空闲的索引结点。然而如果在管理块内有一组空闲的索引结点号，搜索的效率就会高得多。和空闲数据块表不同，管理块中的空闲索引结点是一个独立的表。所以，可能有许多索引结点是空闲的，但在表中并没有反映出来。管理块中的空闲索引结点表的使用方法见图31.4的(A)到(F)。

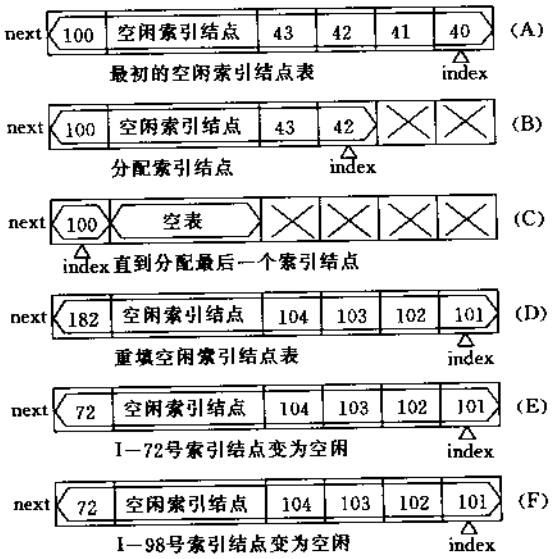

图31.4 空闲索引结点的分配和释放

(A) 表示管理块中空闲索引结点表的最初的状态。其中index指示用于下一次分配请求的索引结点。表中的空闲索引结点号是通过对索引结点数组的线性扫描找到的。在扫描期间按图中自右至左的顺序填写空闲索引结点表。所以，最左边的是找到的最大索引结点号。

(B) 其中两个索引结点（I-40和I-41）已经分配，index的位置已经作了调整。

(C) 在分配更多的索引结点后，最终将到达表中最后一项，当表中最后的索引结点被分配时，就对索引结点数组进行一次新的扫描，找出更多的空闲索引结点，并将它们重新填到管理块中的空闲索引结点表。对索引结点数组的扫描不必从头开始，因为我们知道到100为止的所有索引结点已经在使用。所以，原先的空闲索引结点表中最左面的索引结点号，应该作为下一次扫描的起点。

(D) 自右至左重新填满空闲索引结点表。最左边的索引结点号最后被分配。当需要更多的索引结点时，它将再次作为另一次扫描的起点。

(E) 这一方案对索引结点的分配不成问题。现在考虑索引结点释放的情况。假定某一文件被删除，它使用的72号索引结点被释放。这时，管理块中的空闲索引结点表是满的，没有位置存放新释放的索引结点号。这是一个问题，因为不能放着72号结点不管。如果我们这样做，下次从182结点开始搜索时就找不到它。为了保证不因此而丢失空闲索引结点，在管理块中NEXT所指的索引结点号被新释放的较小的索引结点号取代。这样做，保证下一次扫描索引结点数组查找更多空闲索引结点时，表的最左边的索引结点号总是正确的起点。

(F) 当管理块的空闲索引结点表填满时，如果释放的索引结点号大于NEXT所指的索引结点号，就不需采取特殊措施来记住这一结点号。因为下次从NEXT所指的索引结点号开始搜索时，自然能发

现这一空闲索引结点号。如果一个索引结点被释放，空闲索引结点表中还有空位，就将它加到空闲表中。

31.2.3 索引结点的数据块指针

有了分配新索引结点给文件以及找出磁盘数据块加到文件中去的方法以后，现在还需要知道在已知文件的索引结点号的情况下如何访问文件中的数据块。最明显的做法是将文件的数据块号列在索引结点中。遗憾的是，对任何大文件讲，这一办法行不通。因为这要在索引结点内部留很大空间。

考察用户建立的文件的平均容量表明，多数文件的容量为 1kb 或更小。当文件容量超出这一范围时，随着容量的增加，文件的数量随之减少。这一启示使人们倾向于采用访问小文件快而有效的方案。即使对大文件有某种程度的速度损失，也是一种可以接受的折衷方案。System V 实现的就是这样的方案。

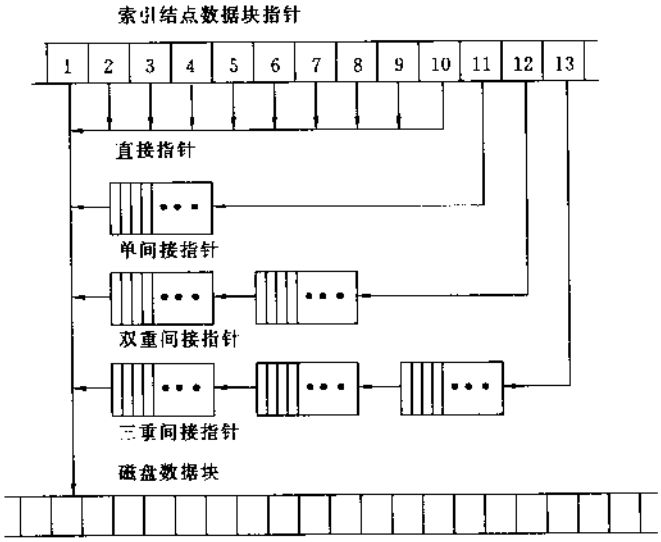

图 31.5 索引结点内的数据块指针

在 system V 索引结点内部有一个包含 13 个元素的数组，每个元素都能保存一个磁盘数据块的块号（见图 31.5）。数组的前 10 个元素用来保存分配给文件的前 10 个数据块的块号。假定每一数据块的容量为 1024 字节，这 10 个数据块的指针允许建立 10Kb 大小的文件。对大多数文件讲，可以用直接查找数据块的办法来访问文件中的数据。

按照这一方案，容量超过 10kb 的文件，不管它最终的容量有多大，索引结点中可供使用的数据块指针只留下 3 个。很明显要用新的办法，使剩下的 3 个数据块指针足以表达现实的文件容量，同时不致于使数据存取的时间变得太长。

这一目标是用间接数据块指针的概念实现的。具体讲，当需要给文件分配第 11 个数据块时，要用到索引结点的第 11 号数据块指针。但是它不直接指向文件使用的数据块，而是作为单间接指针使用。它指向另一个数据块，在这一数据块中包含直接指向文件使用的数据块的指针。

在我们的例子中，假定数据块的块号是 32 位的值。在单间接数据块中可以容纳 256 个数据块块号。这些块号将直接指向用于文件的其他 256 个数据块。所以，使用索引结点中的 11

个指针，使我们能建立文件的容量达到266kb(10+256)。

对容量大于266kb的文件，需要使用索引结点中的第12号指针，即双重间接指针。双重间接指针指向包含256个单间接指针的数据块，而每一单间接指针又指向包含256个直接指针的数据块，这就说明使用索引结点的第12号指针，允行文件访问另外的65536kb(256×256)的数据。

现在你能看出，当文件容量超过64Mb(实际是65802Kb)时，要用到第13号指针。现在它作为三重间接指针使用，允许文件访问另外的16Gb(256×256×256)的文件空间。单个文件达到16Gb听上去就很大了。前面的计算表明，用上面的索引结点结构达到这一容量不成问题的。然而，一些其他因素的限制，使文件实际容量要比这个值小一些，例如，以字节计的文件容量存放在索引结点的另一字段中。这一字段的类型是unsigned long。它是一个32位的数，因而将文件的最大容量限为4Gb，所以在索引结点中有13个数据指针已足够使用。

31.3 minix

minix文件系统是Linux使用的第一个文件系统。在它的最早阶段，Linux曾经需要一个minix文件系统才能对它的源程序进行编译和运行。

从用户的角度看，minix文件系统的设计和前面讲的system V文件系统一样，但是从内部看，两者还是有差别的。

minix内部使用了类似system V的索引结点结构。唯一的实际差别是它只用了7个直接数据块指针，它不支持三重间接块指针，在索引结点中总共用了9个指针。作为默认值，minix使用1kb的数据块和16位的数据块指针。这表示一个间接磁盘块可以保存512个指针，最大的文件容量可以超过256Mb（7个直接数据块，加上通过单间接块访问的512个数据块，再加上通过双重间接块访问的512×512个数据块）。

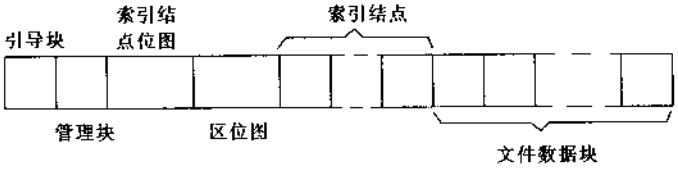

图31.6 minix文件系统的磁盘分区格式

minix文件系统使用的磁盘分区的格式要比systemV使用的更复杂些（见图31.6）。增加复杂性的部分原因是minix文件系统使用区（zone）作为数据块组的概念，作为默认值，区和数据块是一样的，两者的容量都是1kb。

另一个原因是它增加了索引结点位图（inode bitmap）和区位图（zone bitmap）。从实质讲，它们取代了system V管理块中的空闲索引结点表和空闲数据块表。位图的优点是它们使用较少的空间，因为只用1位表示每个索引结点或区是否在使用。得到的结果是：文件系统装配后整个位图都可以放在内存中。搜索内存中的位图的操作远远快于通过磁盘上的索引结点数组进行顺序搜索的操作或从磁盘读取数据块内容的操作。

用区位图而不是用数据块的位图的理由是磁盘按区进行分配。一个区将2的乘方的数据块组合在一起分配给文件，这表示区可以由1个，2个，4个或8个…数据块组成（默认值是1

个数据块)。按这种方案 minix 文件系统的文件容量可以超过 64mb,因为索引结点的指针可以指向容量大于 1kb 的区。

31.4 ext2

第二扩充文件系统几乎成为所有 Linux 发布和安装的标准文件系统类型。它非常灵活和有强大的功能,而且在性能上进行了全面的优化,我们可以从它和前面两个文件系统的差别中看到它的改进。

ext2 文件系统包含 15 个块指针,每个指针占 32 位。其中 12 个用作直接数据块指针,接着是单间接指针,双重间接指针和三重间接指针。

ext2 文件系统的文件名称也和 system V 及 minix 系统用的不一样,它不使用最长 14 个字符的定长记录。在 ext2 中目录项是变长的,所以不浪费空间。最长可达 255 个字符,足以满足任何应用的需要。

磁盘分区的文件系统的格式也作了改进,使得数据的读和写更快和更有效,图 31.7 表示分区的上层结构。

图 31.7 ext2 文件系统的磁盘分区格式

图中表示分区由引导块 (boot block) 和重复的块组 (block group)组成。和其他文件系统相比,块组的概念有助于改进文件系统的可靠性和提高存取速度。

每一块组包含了系统重要信息的副本,特别是管理块 (super block) 和文件系统描述符 (FS descriptors),因而使用块组提高了系统的可靠性。当这些数据的一个副本遭到破坏时,还可以利用备用的副本。

文件系统的其余信息,包括数据块位图 (block bitmap) 和索引结点的位图 (inode bitmap),索引结点数组以及文件的数据块分布在各个块组中,所以每个块组都有各种结构。图 31.8 表示块组的格式。

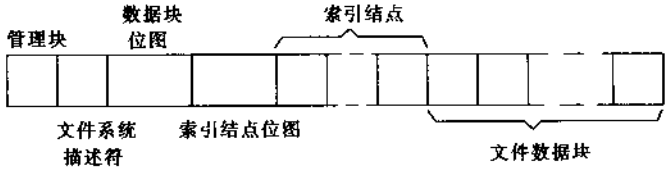

图 31.8 ext2 块组的格式

为了用这种方案提高数据存取速度,ext2 使用了一些最佳化技术,其中包括:

- 当给文件分配数据块时,ext2 力图从同一块组中分配数据块作为文件的索引结点。使得索引结点和数据块在磁盘上靠得很近,在访问文件和它的数据块时减少磁头移动的距离。
- 当你写文件时或给文件分配数据块时,ext2 试图预分配多达 8 块连续的数据块,使文件可以继续增长,用数据块位图做到这一点也是方便的,因为磁盘上连续的 8 块在位图中用相继的 8 位来表示,正好是一个字节。
- 当对文件进行顺序的读操作时,需要进行实际的磁盘读操作。ext2 可以利用连续写数据的优点,因为

在写时预分配了连续的数据块，将要读的数据很可能包含在连续的数据块中。当磁头处于正确的位置时，这就值得进行向前读几块数据，将若干块数据同时读入快速缓冲存贮区中。当需要这些数据时，它们已经在缓冲区中。

ext2 文件系统还提供许多其他功能，使它成为当前最好的通用 Linux 文件系统，它还具有进一步扩充的灵活性，并且仍在不断发展之中。

第三十二章　系 统 调 用

在第 2 部分中我们对 Linux 提供的重要系统调用进行了详细的说明，我们看到的并不是 Linux 系统调用的全部，也不可能是全部，因为系统调用的数目并不固定，随着需求的增长，它的数量还会进一步增加。

本章讨论 Linux 执行系统调用的机制。然后解释如何将自己的系统调用加到 Linux 内核中去。

32.1　中断和异常

大多数新的微处理器，包括 Linux 在上面运行的个人计算机的微处理器，它们的处理可以被比当前执行的程序具有更高优先级的事件所中断。

这些特殊事件有各自的特色，但总的说来，它们可以分为中断和异常（ Interrupts and exceptions ）。中断可以说是硬设备对处理器产生的“刺激”，它们是由外部事件引起的。这些事件包括硬盘和软盘的活动，键盘的输入，调制解调器的控制活动等。另一方面，异常则是响应某些系统错误引起的，也可以是响应某些可以在程序中执行的特殊机器指令引起的。

不管产生的是中断还是异常，处理器响应的方式是类似的，执行一段称为中断处理或异常处理的特殊程序。一般说来，在特定的系统中，对每一种中断和异常类型都有单独的处理程序。

对每一种中断和异常，都在 0 ~ 255 范围内给以一个唯一的数加以标记，当发生中断或异常时，这个数（ 称为向量号 ）用作进入指针数组（ 中断或异常向量表 ）的索引，表中每个指针指向用来处理特定中断或异常类型的子程序的地址。大多数中断和异常像函数调用一样进行处理。所以，当中断或异常处理程序结束时，被中断的程序将从发生中断时的断点继续向下执行。

32.2　系统调用机制

在 Linux 系统中，系统调用是作为一种异常类型实现的。它将执行相应的机器代码指令来产生异常信号。产生中断或异常的重要效果是系统自动将用户模式切换为内核模式来对它进行处理。这就是说，执行系统调用的异常指令时，将自动地将系统切换为内核模式，并安排异常处理程序的执行。它知道如何处理这一调用。

Linux 用来实现系统调用异常的实际指令是：

```
int     $0x80
```

这一指令使用中断/异常向量号 128（ 即 16 进制的 80 ）将控制权转移给内核。

为达到在使用系统调用时不必用机器指令编程，在标准的 C 语言库中为每一系统调用提

供了一段短的子程序，完成机器代码的编程工作。事实上，机器代码段非常简短。它所要做的工作只是将送给系统调用的参数值加载到 CPU 寄存器中，接着执行 int $0x80 指令。然后运行系统调用，系统调用的返回值将送入 CPU 的一个寄存器中，标准的库子程序取得这一返回值，并将它送回给你的程序。

为使系统调用的执行成为一项简单的任务，Linux 提供了一组预处理宏指令。它们可以用在程序中。这些宏指令取一定的参数，然后扩展为调用指定的系统调用的函数：

这些宏指令具有类似下面的名称格式：

```
_syscallN(parameters)
```

其中 N 用系统调用所需的参数数目代替，而 parameters 则用一组参数代替。这些参数使宏指令完成适合于特定的系统调用的扩展。例如，为了建立调用 setuid( ) 系统调用的函数，应该使用：

```
_syscall1(int, setuid, uid_t, uid)
```

_syscall N( ) 宏指令的第 1 个参数说明产生的函数的返回值的类型（这里是 int），第 2 个参数说明产生的函数的名称（这里是 setuid）。

后面是系统调用所需要的每个参数。这一宏指令后面还有两个参数分别用来指定参数的类型和名称（这里是 uid_t 和 uid）。

用作系统调用的参数的数据类型有一个限制，它们的容量不能超过 4 个字节。这是因为执行 int $0x80 指令进行系统调用时，所有的参数值都存在 32 位的 CPU 寄存器中。使用 CPU 寄存器传递参数带来的另一个限制是可以传送给系统调用的参数的数目。这个限制是最多可以传送 5 个参数。所以一共定义了 6 个不同的 _syscall N( ) 宏指令（从 _syscall 0( ) 到 _syscall5( )）。

一旦 _syscall N( ) 宏指令用特定系统调用的相应的参数进行了扩展，得到的结果是一个与系统调用同名的函数，它可以在用户程序中执这一系统调用。

32.3 增加新的系统调用

掌握了 Linux 程序执行系统调用的方法，将自己的系统调用加到内核中去就是一项容易的任务。

为了更好地理解这一过程，我们将实际开发一个普通的新的系统调用，并将它加到内核中去，要增加的系统调用是：

```
int printcons (char *mess, int len);
```

printcons( ) 调用取两个参数，其中 mess 是指向保存在字符数组中的信息的指针，而 len 则是数组中的字符数。新调用所做的工作只是在控制终端屏幕上显示字符数组中的信息。如果执行成功，返回 0 值。在出错时，返回 -1 值及相应的错误信息。

32.3.1 编写调用

第一项任务是编写加到内核中的程序。这是将要加到一个内核文件中去的一个函数，它类似于写设备驱动程序函数。printcons( ) 系统调用的程序如下：

```
asmlinkage int sys_printcons(char *mess, int len)
{
    char *kmess;

    /* 1 */
    if (verify_area(VERIFY_READ, mess, len))
        return -EFAULT;

    /* 2 */
    if ((kmess = vmalloc(len+1)) == 0)
        return -ENOMEM;

    /* 3 */
    memcpy_fromfs(kmess, mess, len);
    kmess[len] = '\0';
    console_print(kmess);

    /* 4 */
    vfree(kmess);
    return 0;
}
```

像这样一项在控制终端屏幕上显示正文的普通任务，这个程序看上去有点过份。但还是可以说明一些概念。假定你的内核的源程序放在标准的位置（/ust/src/linux）。最简单的做法是将上面的程序加到下列文件的最后：

```
/usr/src/linux/kernel/sys.c
```

对程序本身要注意的第一件事是：函数的名称应该是新的系统调用名称前面加上 sys_。并像函数一样说明系统调用使用的参数。

下面是程序中编号的注释的说明：

1. 这一系统调用使用 console_print( )内核函数在控制台屏幕上显示字符串的内容。由于正文信息的指针指向用户内存的地址，所以在显示前应将它复制到内核的内存中。首先要检查用户对指定的内存地址是否有合法的访问权。这项检验用 verify_area( ) 内核函数完成。如果不允许用户进行访问，这一系统调用将返回 EFAULT 错误信息。
2. 为了能复制任意长度的信息，下一步用 vmalloc( ) 内核函数动态地从内存中抓取一块内存，使用户

的正文信息能复制到内核内存中。注意,请求的内存块要比从用户空间复制的字符数多一个。这是为了有写入0字符的位置来结束这段信息,使它成为供显示用的C语言的字符串。由vmalloc( )返回的指针指向kmess。这里出现的任何错误将导致返回ENOMEM出错信息。

3. 下面的3行程序用来取用户信息。用memcpy_fromfs( )函数将它从mess复制到kmess。在信息的字符串后面加上0供显示用。然后用console_print( )内核函数将信息显示在屏幕上。
4. 最后两行程序完成清理工作。用vfree( )函数将已分配的内存释放给系统。最后一行给用户进程返回0值,表示调用成功。

32.3.2 连接新的系统调用

写了新的系统调用的程序后,下一项任务是使内核的其余部分知道这一程序的存在。然后重建包含新的系统调用的内核。

为了从已有的内核程序中增加到新的函数的连接,需要编辑两个文件。这取决于使用的内核版本,1.2版以前(包括1.2版)的内核要修改的第一个文件是:

```
/usr/src/linux/include/linux/unistd.h
```

1.3版及1.3版以上的内核,要修改的第一个文件是:

```
/usr/src/linux/include/asm-i386/unistd.h
```

这一个文件包含定义行的清单,用来给每个系统调用分配一个唯一的号码。文件中每行的格式如下:

```
#define  __NR_name      NNN
```

其中name用系统调用名称代替,而NNN则是这一系统调用对应的号码。应该将系统调用名称加到清单的最后,并给它分配序列中下一个可用的系统调用号。用于我们的例子的系统调用的定义行可能是:

```
#define  __NR_printcons    149
```

要修改的第2个文件同样与你使用的内核版本有关。1.1版(包括1.1版)以前的内核用下面的文件:

```
/usr/src/linux/include/linux/sys.h
```

你要做的是将新内核函数加到清单的最后。这一清单用来对sys_call_table[]数组进行初始化。数组包含指向内核中每个系统调用的指针。这样做就在数组中增加了到你的新的内核函数的指针。为使编译程序接受函数名称。还需要在变量说明清单中加上函数的说明(extern)。说明清单出现在上述数组定义的前面。

对1.2版和1.2版以后的内核讲,要修改的第2个文件是:

/usr/src/linux/arch/i386/kernel/entry.S

和前面的版本一样,它用于对指针数组的初始化。但现在用汇编语言而不是用 C 语言写成。要对它成功地进行修改并不一定要看懂它,要做的只是找到具有下列格式的汇编指令清单:

```
.long   _sys_name
```

它后面是具有下列格式的另一个汇编指令:

```
.space  (NR_syscalls-NNN)*4
```

其中 NNN 是系统调用的总数,要做的是用自己内核函数名称前面加下划字符在文件中增加新的 .long 行。然后将 NNN 的值加 1,使新调用也包括在总数内。对我们的例子讲,应该增加:

```
.long   _sys_printcons
.space  (NR_syscalls-149)*4
```

到此为止,剩下的事是重建和运行新内核,在前面的说明中一些文件的路径名用到了 i386,如果你在其他体系结构的机器上使用 Linux,应该用相应的名称代替它。

32.3.3 使用新的系统调用

现在要说明如何在你的程序中使用新的系统调用。注意,在标准的 C 语言库中没有新的系统调用的承接段,必须自己建立它。

利用 _syscall N( ) 宏指令,使用新的 printcons()系统调用的短的 C 程序如下:

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

_syscall2(int, printcons, char *, mess, int, len)
main()
{
    char message[] = "Print straight to console \n";

    if (printcons(message, strlen(message)) == -1)
        printf("Error occurred in printcons() call \n");
}
```

因为我们的系统调用中使用了两个参数,这里用的是 _syscall2( ) 宏指令。宏指令本身将在你的程序中扩展成名称为 printcons( ) 的函数,它在 main( ) 函数内部加以调用。在 printcons

( ) 函数中,预处理程序产生所有必要的机器代码指令,包括用系统调用参数值加载相应的CPU 寄存器,然后执行 int 0x80 异常指令等,当从异常处理子程序返回时,将对你的内核函数的返回值进行检查,并将相应的值送回 main( )函数中的代码。

GNU 通用公共许可证(GPL)

下面的正文是自由软件基金会 GNU 通用公共许可证原始文档的副本。Linux 操作系统以及与它有关的大量软件是在 GPL 的推动下开发和发布的。

你将看到:如果你打算为了发布的目的修改、更新或改进任何受通用公共许可证约束的软件,你的修改的软件同样必须受到 GNU 通用许可证条款的约束。

GNU 通用公共许可证

1991.6 第二版

序　言

大多数软件许可证决意剥夺你的共享和修改软件的自由。对比之下,GNU 通用公共许可证力图保证你的共享和修改自由软件的自由。——保证自由软件对所有用户是自由的。GPL 适用于大多数自由软件基金会的软件,以及由使用这些软件而承担义务的作者所开发的软件。(自由软件基金会的其他一些软件受 GNU 库通用许可证的保护)。你也可以将它用到你的程序中。

当我们谈到自由软件(free software)时,我们指的是自由而不是价格。我们的 GNU 通用公共许可证决意保证你有发布自由软件的自由(如果你愿意,你可以对此项服务收取一定的费用);保证你能收到源程序或者在你需要时能得到它;保证你能修改软件或将它的一部分用于新的自由软件;而且还保证你知道你能做这些事情。

为了保护你的权利,我们需要作出规定:禁止任何人不承认你的权利,或者要求你放弃这些权利。如果你修改了自由软件或者发布了软件的副本,这些规定就转化为你的责任。

例如,如果你发布这样一个程序的副本,不管是收费的还是免费的,你必须将你具有的一切权利给予你的接受者;你必须保证他们能收到或得到源程序;并且将这些条款给他们看,使他们知道他们有这样的权利。

我们采取两项措施来保护你的权利。(1)给软件以版权保护。(2)给你提供许可证。它给你复制,发布和修改这些软件的法律许可。

同样,为了保护每个作者和我们自己,我们需要清楚地让每个人明白,自由软件没有作保(no warranty)。如果由于其他某个人修改了软件,并继续加以传播。我们需要它的接受者明白:他们所得到的并不是原来的自由软件。由其他人引入的任何问题,不应损害原作者的声誉。

最后,任何自由软件经常受到软件专利的威胁。我们希望避免这样的风险,自由软件的再发布者以个人名义获得专利许可证,事实上,将软件变为私有。为防止这一点,我们必须明确:任何专利必须以允许每个人自由使用为前提,否则就不准许有专利。

下面是有关复制,发布和修改的确切的条款和条件。

GNU 通用公共许可证
有关复制,发布和修改的条款和条件

0. 此许可证适用于任何包含版权所有者声明的程序和其他作品,版权所有者在声明中明确说明程序和作品可以在 GPL 条款的约束下发布。下面提到的"程序"指的是任何这样的程序或作品。而"基于程序的作品"指的是程序或者任何受版权法约束的衍生作品。也就是说包含程序或程序的一部分的作品。可以是原封不动的,或经过修改的和/或翻译成其他语言的(程序)。在下文中,翻译包含在修改的条款中。每个许可证接受人(licensee)用"你"来称呼。

许可证条款不适用于复制,发布和修改以外的活动。这些活动超出这些条款的范围。运行程序的活动不受条款的限止。仅当程序的输出构成基于程序作品的内容时,这一条款才适用(如果只运行程序就无关)。是否普遍适用取决于程序具体用来做什么。

1. 只要你在每一副本上明显和恰当地出版版权声明和不承担保证的声明,保持此许可证的声明和没有保证的声明完整无损,并和程序一起给每个其他的程序接受者一份许可证的副本,你就可以用任何媒体复制和发布你收到的原始的程序的源代码。

你可以为转让副本的实际行动收取一定费用。你也有权选择提供保证以换取一定的费用。

2. 你可以修改程序的一个或几个副本或程序的任何部分,以此形成基于程序的作品。只要你同时满足下面的所有条件,你就可以按前面第一款的要求复制和发布这一经过修改的程序或作品。

   a) 你必须在修改的文件中附有明确的说明:你修改了这一文件及具体的修改日期。

   b) 你必须使你发布或出版的作品(它包含程序的全部或一部分,或包含由程序的全部或部分衍生的作品)允许第三方作为整体按许可证条款自由使用。

   c) 如果修改的程序在运行时以交互方式读取命令,你必须使它在开始进入常规的交互使用方式时打印或显示声明:包括适当的版权声明和没有保证的声明(或者你提供保证的声明);用户可以按此许可证条款重新发布程序的说明;并告诉用户如何看到这一许可证的副本。(例外的情况:如果原始程序以交互方式工作,它并不打印这样的声明,你的基于程序的作品也就不用打印声明。)

这些要求适用于修改了的作品的整体。如果能够确定作品的一部分并非程序的衍生产品,可以合理地认为这部分是独立的,是不同的作品。当你将它作为独立作品发布时,它不受此许可证和它的条款的约束。但是当你将这部分作为基于程序的作品的一部分发布时,作为整体它将受到许可证条款约束。准予其他许可证持有人的使用范围扩大到整个产品。也就是每个部分,不管它是谁写的。

因此,本条款的意图不在于索取权利;或剥夺全部由你写成的作品的权利。而是履行权利来控制基于程序的集体作品或衍生作品的发布。

此外,将与程序无关的作品和该程序或基于程序的作品一起放在存储体或发布媒体的同一卷上,并不导致将其他作品置于此许可证的约束范围之内。

3. 你可以以目标码或可执行形式复制或发布程序(或符合第 2 款的基于程序的作品),只要你遵守前面的第 1,2 款,并同时满足下列 3 条中的 1 条。

   a) 在通常用作软件交换的媒体上,和目标码一起附有机器可读的完整的源码。这些源码的发布应符合上面第 1,2 款的要求。或者

   b) 在通常用作软件交换的媒体上,和目标码一起,附有给第三方提供相应的机器可读的源码的书面报价。有效期不少于 3 年,费用不超过实际完成源程序发布的实际成本。源码的发布应符合上面的第 1,2 款的要求。或者

   c) 和目标码一起,附有你收到的发布源码的报价信息。(这一条款只适用于非商业性发布,而且你只收到程序的目标码或可执行代码和按 b)款要求提供的报价)。

   作品的源码指的是对作品进行修改最优先择取的形式。对可执行的作品讲,完整的源码包括:所有模块的所有源程序,加上有关的接口的定义,加上控制可执行作品的安装和编译的 script。作为特殊例外,发布的源码不必包含任何常规发布的供可执行代码在上面运行的操作系统的主要组成部分(如编译程序,内核等)。除非这些组成部分和可执行作品结合在一起。

   如果采用提供对指定地点的访问和复制的方式发布可执行码或目标码,那么,提供对同一地点的访问和复制源码可以算作源码的发布,即使第三方不强求与目标码一起复制源码。

4. 除非你明确按许可证提出的要求去做,否则你不能复制,修改,转发许可证和发布程序。任何试图用其他方式复制,修改,转发许可证和发布程序是无效的。而且将自动结束许可证赋予你的权利。然而,对那些从你那里按许可证条款得到副本和权利的人们,只要他们继续全面履行条款,许可证赋予他们的权利仍然有效。

5. 你没有在许可证上签字,因而你没有必要一定接受这一许可证。然而,没有任何其他东西赋予你修改和发布程序及其衍生作品的权利。如果你不接受许可证,这些行为是法律禁止的。因此,如果你修改或发布程序(或任何基于程序的作品),你就表明你接受这一许可证以及它的所有有关复制,发布和修改程序或基于程序的作品的条款和条件。

6. 每当你重新发布程序(或任何基于程序的作品)时,接受者自动从原始许可证颁发者那里接到受这些条款和条件支配的复制,发布或修改程序的许可证。你不可以对接受者履行这里赋予他们的权利强加其他限制。你也没有强求第三方履行许可证条款的义务。

7. 如果由于法院判决或违反专利的指控或任何其他原因(不限于专利问题)的结果,强加于你的条件(不管是法院判决,协议或其他)和许可证的条件有冲突。他们也不能用许可证条款为你开脱。在你不能同时满足本许可证规定的义务及其他相关的义务时,作为结果,你可以根本不发布程序。例如,如果某一专利许可证不允许所有那些直接或间接从你那里接受副本的人们在不付专利费的情况下重新发布程序,唯一能同时满足两方面要求的办法是停止发布程序。

   如果本条款的任何部分在特定的环境下无效或无法实施,就使用条款的其余部分。并将条款作为整体用于其他环境。

   本条款的目的不在于引诱你侵犯专利或其他财产权的要求,或争论这种要求的有

效性。本条款的主要目的在于保护自由软件发布系统的完整性。它是通过通用公共许可证的应用来实现的。许多人坚持应用这一系统,已经为通过这一系统发布大量自由软件作出慷慨的供献。作者/捐献者有权决定他/她是否通过任何其他系统发布软件。许可证持有人不能强制这种选择。

本节的目的在于明确说明许可证其余部分可能产生的结果。

8. 如果由于专利或者由于有版权的接口问题使程序在某些国家的发布和使用受到限止,将此程序置于许可证约束下的原始版权拥有者可以增加限止发布地区的条款,将这些国家明确排除在外。并在这些国家以外的地区发布程序。在这种情况下,许可证包含的限止条款和许可证正文一样有效。
9. 自由软件基金会可能随时出版通用公共许可证的修改版或新版。新版和当前的版本在原则上保持一致,但在提到新问题时或有关事项时,在细节上可能出现差别。

   每一版本都有不同的版本号。如果程序指定适用于它的许可证版本号以及"任何更新的版本"。你有权选择遵循指定的版本或自由软件基金会以后出版的新版本,如果程序未指定许可证版本,你可选择自由软件基金会已经出版的任何版本。
10. 如果你愿意将程序的一部分结合到其他自由程序中,而它们的发布条件不同。写信给作者,要求准予使用。如果是自由软件基金会加以版权保护的软件,写信给自由软件基金会。我们有时会作为例外的情况处理。我们的决定受两个主要目标的指导。这两个主要目标是:我们的自由软件的衍生作品继续保持自由状态。以及从整体上促进软件的共享和重复利用。

没 有 担 保

11. 由于程序准予免费使用,在适用法准许的范围内,对程序不作担保。除非另有书面说明,版权所有者和/或其他提供程序的人们"一样"不提供任何类型的担保。不论是明确的,还是隐含的。包括但不限于隐含的适销和适合特定用途的保证。全部的风险,如程序的质量和性能问题都由你来承担。如果程序出现缺陷,你承担所有必要的服务,修复和改正的费用。
12. 除非适用法或书面协议的要求,在任何情况下,任何版权所有者或任何按许可证条款修改和发布程序的人们都不对你的损失负有任何责任。包括由于使用或不能使用程序引起的任何一般的,特殊的,偶然发生的或重大的损失(包括但不限于数据的损失,或者数据变得不精确,或者你或第三方的持续的损失,或者程序不能和其他程序协调运行等)。即使版权所有者和其他人提到这种损失的可能性也不例外。

最后的条款和条件
如何将这些条款适用你的新程序

如果你开发了新程序,而且你需要它得到公众最大限度的利用。要做到这一点的最好办法是将它变为自由软件。使得每个人都能在遵守条款的基础上对它进行修改和重新发布。

为了做到这一点,给程序附上下列声明。最安全的方式是将它放在每个源程序的开头,以便最有效地传递拒绝保证的信息。每个文件至少应有"版权所有"行以及在什么地方能看到声

明全文的说明。

<用一行空间给出程序的名称和它用来做什么的简单说明>
版权所有(C)　　19XX　　<作者姓名>

这一程序是自由软件,你可以遵照自由软件基金会出版的 GNU 通用公共许可证条款来修改和重新发布这一程序。或者用许可证的第二版,或者(根据你的选择)用任何更新的版本。

发布这一程序的目的是希望它有用,但没有任何保证。甚至没有适合特定目的的隐含的保证。更详细的情况请参阅 GNU 通用公共许可证。

你应该已经和程序一起收到一份 GNU 通用公共许可证的副本。如果还没有,写信给:

The Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA

还应加上如何和你保持联系的信息。

如果程序以交互方式进行工作,当它开始进入交互方式工作时,使它输出类似下面的简短声明:

Gnomovision 第 69 版, 版权所有(C) 19XX, 作者姓名,

Gnomovision 绝对没有保证。要知道详细情况,请输入'show w'。

这是自由软件,欢迎你遵守一定的条件重新发布它,要知道详细情况,

请输入'show c'。

假设的命令'show w'和'show c'应显示通用公共许可证的相应条款。当然,你使用的命令名称可以不同于'show w'和'show c'。根据你的程序的具体情况,也可以用菜单或鼠标选项来显示这些条款。

如果需要,你应该取得你的上司(如果你是程序员)或你的学校签署放弃程序版权的声明。下面只是一个例子,你应该改变相应的名称:

Yoyodyne 公司以此方式放弃 James Harker
所写的 Gnomovision 程序的全部版权利益。
<Ty coon 签名>,1989.4.1
Ty coon 副总裁

这一许可证不允许你将程序并入专用程序。如果你的程序是一个子程序库。你可能会认为用库的方式和专用应用程序连接更有用。如果这是你想做的事,使用 GNU 库通用公共许可证代替本许可证。