第1章　UNIX基础知识

1.1　引言

所有操作系统都向它们运行的程序提供服务。典型的服务有执行新程序、打开文件、读文件、分配存储区、获得当前时间等等，本书集中阐述了UNIX操作系统各种版本所提供的服务。

以严格的步进方式、不超前引用尚未说明过的术语的方式来说明UNIX几乎是不可能的(可能也会是令人厌烦的)。本章从程序设计人员的角度快速浏览UNIX，并对书中引用的一些术语和概念进行简要的说明并给出实例。在以后各章中，将对这些概念作更详细的说明。本章也对不熟悉UNIX的程序设计人员简要介绍了UNIX提供的各种服务。

1.2　登录

1.2.1　登录名

登录UNIX系统时，先键入登录名，然后键入口令。系统在其口令文件，通常是/etc/passwd文件中查看登录名。口令文件中的登录项由7个以冒号分隔的字段组成：登录名，加密口令，数字用户ID(224)，数字组ID(20)，注释字段，起始目录(/home/stevens)，以及shell程序(/bin/ksh)。

很多比较新的系统已将加密口令移到另一个文件中。第6章将说明这种文件以及存取它们的函数。

1.2.2　shell

登录后，系统先显示一些典型的系统信息，然后就可以向shell程序键入命令。shell是一个命令行解释器，它读取用户输入，然后执行命令，用户通常用终端，有时则通过文件　(称为shell脚本)向shell进行输入。常用的shell有：

- Bourne shell, /bin/sh
- C shell, /bin/csh
- KornShell, /bin/ksh

系统从口令文件中登录项的最后一个字段中了解到应该执行哪一个shell。

自V7以来，Bourne shell得到了广泛应用，几乎每一个现有的UNIX系统都提供Bourne shell。C shell是在伯克利开发的，所有BSD版本都提供这种shell。另外，AT&T的系统V/386 R3.2和SVR4也提供C shell（下一章将对这些不同的UNIX版本作更多说明）。KornShell是Bourne shell的后继者，它由SVR4提供。KornShell在大多数UNIX系统上运行，但在SVR4之前，通常它需要另行购买，所以没有其他两种shell流行。

本书将使用很多shell实例，以执行已开发的程序，其中将应用Bourne shell和KornShell都具有的功能。

Bourne shell是Steve Bourne在贝尔实验室中开发的，其控制流结构使人想起Algol 68。C shell是在伯克利由Bill Joy完成的，其基础是第6版shell（不是Bourne shell）。其控制结构很像C语言，它支持一些Bourne shell没有提供的功能，如作业控制，历史机制和命令行编辑。 KornShell是David Korn在贝尔实验室中开发的，它兼容Bourne shell，并且也包含了使C shell非常流行的一些功能，如作业控制、命令行编译等。

本书将使用这种形式的注释来描述历史，并对不同的UNIX实现进行比较。当我们了解了历史缘由后，采用某种特定实现技术的原因将变得清晰起来。

1.3 文件和目录

1.3.1 文件系统

UNIX文件系统是目录和文件的一种层次安排，目录的起点称为根（root），其名字是一个字符/。

目录（directory）是一个包含目录项的文件，在逻辑上，可以认为每个目录项都包含一个文件名，同时还包含说明该文件属性的信息。文件属性是：文件类型，文件长度，文件所有者，文件的许可权（例如，其他用户能否能访问该文件），文件最后的修改时间等。stat和fstat函数返回一个包含所有文件属性的信息结构。第4章将详细说明文件的各种属性。

1.3.2 文件名

目录中的各个名字称为文件名（filename）。不能出现在文件名中的字符只有两个，斜线(/)和空操作符（null）。斜线分隔构成路径名(在下面说明)的各文件名，空操作符则终止一个路径名。尽管如此，好的习惯是只使用印刷字符的一个子集作为文件名字符 (只使用子集的理由是：如果在文件名中使用了某些shell特殊字符，则必须使用shell的引号机制来引用文件名)。

当创建一个新目录时，自动创建了两个文件名： .(称为点)和..(称为点-点)。点引用当前目录，点-点则引用父目录。在最高层次的根目录中，点-点与点相同。

某些UNIX文件系统限制文件名的最大长度为14个字符，BSD版本则将这种限制扩展为255个字符。

1.3.3 路径名

0个或多个以斜线分隔的文件名序列(可以任选地以斜线开头)构成路径名（pathname），以斜线开头的路径名称为绝对路径名（ absolute pathname），否则称为相对路径名（ relative pathname）。

实例

不难列出一个目录中所有文件的名字，程序1-1是ls(1)命令的主要实现部分

程序1-1 列出一个目录中的所有文件

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

```
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("a single argument (the directory name) is required");

    if ( (dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);

    while ( (dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

ls(1)这种表示方法是UNIX的惯用方法，用以引用UNIX手册集中的一个特定项。它引用第一部分中的ls项，各部分通常用数字1至8表示，在每个部分中的各项则按字母顺序排列。假定你有一份所使用的UNIX系统的手册。

> 早期的UNIX系统把8个部分都集中在一本手册中，现在的趋势是把这些部分分别安排在不同的手册中：有用户专用手册，程序员专用手册，系统管理员专用的手册等等。
>
> 某些UNIX系统把一个给定部分中的手册页又用一个大写字母进一步分成若干小部分，例如，AT&T〔1990e〕中的所有标准I/O函数都被指明在3S部分中，例如fopen(3S)。
>
> 某些UNIX系统，例如以Xenix为基础的系统，不是采用数字将手册分成若干部分，而是用C表示命令(第1部分)，S表示服务(通常是第2、3部分)等等。

如果你有联机手册，则可用下面的命令查看ls命令手册页：

```
man 1 ls
```

程序1-1只打印一个目录中各个文件的名字，不显示其他信息，如若该源文件名为 myls.c，则可以用下面的命令对其进行编译，编译的结果送入系统默认名为 a.out的可执行文件名：

```
cc myls.c
```

某种样本输出是：

```
$ a.out /dev
.
..
MAKEDEV
console
tty
mem
kmem
null
```

此处略去多行

```
printer
$ a.out /var/spool/mqueue
can't open /var/spool/mqueue:Permission denied
$ a.out /dev/tty
can't open /dev/tty:Not a directory
```

本书将以这种方式表示输入的命令以及其输出：输入的字符以粗体表示，程序输出则以另一种字体表示。如果欲对输出添加注释，则以中文宋体表示，输入之前的美元符号 ($)是shell打印的提示符，本书将shell提示符显示为$。

注意，列出的目录项不是以字母顺序排列的， ls命令则一般按字母顺序列出目录项。

在这20行的程序中，有很多细节需要考虑：

• 首先，其中包含了一个头文件ourhdr.h。本书中几乎每一个程序都包含此头文件。它包含了某些标准系统头文件，定义了许多常数及函数原型，这些都将用于本书的各个实例中，附录B列出了常用头文件。

• main函数的说明使用了ANSI C标准所支持的新风格（下一章将对 ANSI C作更多说明）。

• 取命令行的第1个参数argv〔1〕作为列出的目录名。第 7章将说明main函数如何被调用，程序如何存取命令行参数和环境变量。

• 因为各种不同UNIX系统的目录项的实际格式是不一样的，所以使用函数 opendir, readdir和closedir处理目录。

• opendir函数返回指向DIR结构的指针，并将该指针传向 readdir函数。我们并不关心DIR结构中包含了什么。然后，在循环中调用 readdir来读每个目录项。它返回一个指向 dirent结构的指针，而当目录中已无目录项可读时则返回 null指针。在dirent结构中取出的只是每个目录项的名字 (d_name)。使用该名字，此后就可调用 stat函数(见4.2节)以决定该文件的所有属性。

• 调用了两个自编的函数来对错误进行处理：err_sys和err_quit。从上面的输出中可以看到，err_sys函数打印一条消息（“ Permission denied(许可权拒绝) ”或“ Not a directory(不是一个目录) ”），说明遇到了什么类型的错误。这两个出错处理函数在附录 B中说明， 1.7节将更多地叙述出错处理。这两个出错处理函数在附录 B中说明1.7节将更详细地叙述出错处理。

• 当程序将结束时，它以参数 0调用函数exit。函数exit终止程序。按惯例，参数0的意思是正常结束，参数值1～255则表示出错。8.5节将说明一个程序(例如shell或我们所编写的程序)如何获得它所执行的另一个程序的exit状态。

1.3.4 工作目录

每个进程都有一个工作目录 (working directory，有时称为当前工作目录 (current working directory))。所有相对路径名都从工作目录开始解释。进程可以用 chdir函数更改其工作目录。

例如，相对路径名doc/memo/joe指的是文件joe，它在目录memo中，而memo又在目录doc中， doc则应是工作目录中的一个目录项。从该路径名可以看出， doc和memo都应当是目录，但是却不清楚joe是文件还是目录。路径名 /urs/lib/lint是一个绝对路径名，它指的是文件 (或目录) lint，而lint在目录lib中，lib则在目录usr中，usr则在根目录中。

1.3.5 起始目录

登录时，工作目录设置为起始目录(home directory)，该起始目录从口令文件（见1.2节）中

的登录项中取得。

1.4 输入和输出

1.4.1 文件描述符

文字描述符是一个小的非负整数，内核用以标识一个特定进程正在存访的文件。当内核打开一个现存文件或创建一个新文件时，它就返回一个文件描述符。当读、写文件时，就可使用它。

1.4.2 标准输入、标准输出和标准出错

按惯例，每当运行一个新程序时，所有的shell都为其打开三个文件描述符：标准输入、标准输出以及标准出错。如果像简单命令ls那样没有做什么特殊处理，则这三个描述符都连向终端。大多数shell都提供一种方法，使任何一个或所有这三个描述符都能重新定向到某一个文件，例如：

```
ls > file.list
```

执行ls命令，其标准输出重新定向到名为file.list的文件上。

1.4.3 不用缓存的I/O

函数open、read、write、lseek以及close提供了不用缓存的I/O。这些函数都用文件描述符进行工作。

实例

如果愿意从标准输入读，并写向标准输出，则程序1-2可用于复制任一UNIX文件。

程序1-2 将标准输入复制到标准输出

```
#include    "ourhdr.h"

#define BUFFSIZE    8192

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

头文件<unistd.h>(ourhdr.h中包含了此头文件)及两个常数STDIN_FILENO和STDOUT_FILENO是POSIX标准的一部分（下一章将对此作更多的说明）。很多UNIX系统服务的函数原型，例如我们调用的read和write都在此头文件中。函数原型也是ANSI C标准的一部分，本章的

稍后部分将对此作更多说明。

两个常数STDIN_FILENO和STDOUT_FILENO定义在<unistd.h>头文件中，它们指定了标准输入和标准输出的文件描述符。它们的典型值是0和1，但是为了可移植性，我们将使用这些新名字。

3.9节将详细讨论BUFFSIZE常数，说明各种不同的值将如何影响程序的效率。但是不管该常数的值如何，此程序总能复制任一UNIX文件。

read函数返回读得的字节数，此值用作要写的字节数。当到达文件的尾端时， read返回0，程序停止执行。如果发生了一个读错误， read返回－1。出错时大多数系统函数返回－1。

如果编译该程序，其结果送入标准的a.out文件，并以下列方式执行它：

```
a.out > data
```

那么，标准输入是终端，标准输出则重新定向至文件 data，标准出错也是终端。如果此输出文件并不存在，则shell创建它。

第3章将更详细地说明不用缓存的I/O函数。

1.4.4 标准I/O

标准I/O函数提供一种对不用缓存的I/O函数的带缓存的界面。使用标准I/O可无需担心如何选取最佳的缓存长度，例如程序1-2中的BUFFSIZE常数。另一个使用标准I/O函数的优点与处理输入行有关(常常发生在UNIX的应用中)。例如， fgets函数读一完整的行，而另一方面， read函数读指定字节数。

我们最熟悉的标准I/O函数是printf。在调用printf的程序中，总是包括<stdio.h>(通常包括在ourhdr.h中)，因为此头文件包括了所有标准I/O函数的原型。

实例

程序1-3的功能类似于调用read和write的前一个程序1-2，5.8节将对程序1-3作更详细的说明。它将标准输入复制到标准输出，于是也就能复制任一UNIX文件。

程序1-3 用标准I/O将标准输入复制到标准输出

```
#include    "ourhdr.h"

int
main(void)
{
    int     c;

    while ( (c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

函数getc一次读1个字符，然后putc将此字符写到标准输出。读到输入的最后 1个字节时，getc返回常数EOF。标准输入、输出常数stdin和stdout定义在头文件<stdio.h>中，它们分别表示标准输入和标准输出文件。

1.5 程序和进程

1.5.1 程序

程序（program）是存放在磁盘文件中的可执行文件。使用6个exec函数中的一个由内核将程序读入存储器，并使其执行。8.9节将说明这些exec函数。

1.5.2 进程和进程ID

程序的执行实例被称为进程（process）。本书的每一页几乎都会使用这一术语。某些操作系统用任务表示正被执行的程序。

每个UNIX进程都一定有一个唯一的数字标识符，称为进程ID（process ID）。进程ID总是一非负整数。

实例

程序1-4用于打印进程ID。

程序1-4 打印进程ID

```
#include    "ourhdr.h"

int
main(void)
{
    printf("hello world from process ID %d\n", getpid());
    exit(0);
}
```

如果要编译该程序，其结果送入a.out文件，然后执行它，则有：

```
$ a.out
hello world from process ID 851
$ a.out
hello world from precess ID 854
```

此程序运行时，它调用函数getpid得到其进程ID。

1.5.3 进程控制

有三个用于进程控制的主要函数： fork、exec和waitpid（exec函数有六种变体，但经常把它们统称为exec函数）。

实例

程序1-5 从标准输入读命令并执行

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
```

```
{
    char    buf[MAXLINE];
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0;    /* replace newline with null */

        if ( (pid = fork()) < 0)
            err_sys("fork error");

        else if (pid == 0) {         /* child */
            execlp(buf, buf, (char *) 0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

UNIX的进程控制功能可以用一个较简单的程序（见程序 1-5）说明，该程序从标准输入读命令，然后执行这些命令。这是一个类似于 shell程序的基本实施部分。在这个 30行的程序中，有很多功能需要思考：

• 用标准I/O函数fgets从标准输入一次读一行，当键入文件结束字符（通常是 Ctrl-D）作为行的第1个字符时，fgets返回一个null指针，于是循环终止，进程也就终止。第 11章将说明所有特殊的终端字符(文件结束、退格字符、整行擦除等等)，以及如何改变它们。

• 因为fgets返回的每一行都以新行符终止，后随一个 null字节，故用标准C函数strlen计算此字符串的长度，然后用一个 null字节替换新行符。这一操作的目的是因为 execlp函数要求的是以null结束的参数，而不是以新行符结束的参数。

• 调用fork创建一个新进程。新进程是调用进程的复制品，故称调用进程为父进程，新创建的进程为子进程。fork对父进程返回新子进程的非负进程ID，对子进程则返回0。因为fork创建一新进程，所以说它被调用一次(由父进程)，但返回两次(在父进程中和在子进程中)。

• 在子进程中，调用 execlp以执行从标准输入读入的命令。这就用新的程序文件替换了子进程。fork和跟随其后的exec的组合是某些操作系统所称的产生一个新进程。在 UNIX中，这两个部分分成两个函数。第8章将对这些函数作更多说明。

• 子进程调用 execlp执行新程序文件，而父进程希望等待子进程终止，这一要求由调用 waitpid实现，其参数指定要等待的进程 (在这里，pid参数是子进程ID)。waitpid函数也返回子进程的终止状态(status变量)。在此简单程序中，没有使用该值。如果需要，可以用此值精确地确定子进程是如何终止的。

• 该程序的最主要限制是不能向执行的命令传递参数。例如不能指定列出的目录名，只能对工作目录执行ls命令。为了传递参数，先要分析输入行，然后用某种约定把参数分开 (很可能使用空格或制表符)，然后将分隔后的各个参数传递给 execlp函数。尽管如此，此程序仍可用来说明UNIX的进程控制功能。

如果运行此程序，则得到下列结果。注意，该程序使用了一个不同的提示符 (%)。

```
$ a.out
% date
Fri Jun 7 15:50:36 MST 1991
% who
stevens console Jun 5 06:01
stevens ttyp0   Jun 5 06:02
% pwd
/home/stevens/doc/apue/proc
% ls
Makefile
a.out
shelll.c
% ^D                          键入文件结束符
$                             输出常规的shell提示符
```

1.6 ANSI C

本书中的所有实例都用ANSI C编写。

1.6.1 函数原型

头文件<unistd.h>包含了许多UNIX系统服务的函数原型，例如已调用过的 read，write和getpid函数。函数原型是ANSI C标准的组成部分。这些函数原型如下列形式：

```
ssize_t  read(int, void *, size_t);
ssize_t  write(int, const void *, size_t);
pid_t    getpid(void);
```

最后一个的意思是：getpid没有参数(void)，返回值的数据类型为pid_t。提供了这些函数原型后，编译程序在编译时就可以检查在调用函数时是否使用了正确的参数。在程序 1-4中，如果调用带参数的getpid(如getpid(1))，则ANSI C编辑程序将给出下列形式的出错信息：

```
line 8: too many arguments to function "getpid"
```

另外，因为编译程序知道参数的数据类型，所以如果可能，它就会将参数强制转换成所需的数据类型。

1.6.2 类属指针

从上面所示的函数原型中可以注意到另一个区别： read和write的第二个参数现在是void *类型。所有早期的UNIX系统都使用char *这种指针类型。作这种更改的原因是： ANSI C使用void *作为类属指针来代替char *。

函数原型和类属指针的组合消去了很多非ANSI C编辑程序需要的显式类型强制转换。

例如，给出了write原型后，可以写成：

```
float data[100];
write (fd, data, sizeof(data))
```

若使用非ANSI编译程序，或没有给出函数原型，则需写成：

```
write(fd, (void *)data, sizeof(data));
```

也可将void *指针特征用于malloc函数(见7.8节)。malloc的原型为：

```
void * malloc(size_t)
```

这使得可以有如下程序段：

```
int * ptr;
ptr = malloc (1000 * sizeof(int));
```

它无需将返回的指针强制转换成int *类型。

1.6.3 原始系统数据类型

前面所示的getpid函数的原型定义了其返回值为 pid_t类型，这也是POSIX中的新规定。UNIX的早期版本规定此函数返回一整型。与此类似， read和write返回类型为ssize_t的值，并要求第三个参数的类型是size_t。

以_t结尾的这些数据类型被称为原始系统数据类型。它们通常在头文件 <sys/types.h>中定义(头文件<unistd.h>应已包括该头文件)。它们通常以C typedef说明加以定义。typedef说明在C语言中已超过15年了(所以这并不要求ANSI C)，它们的目的是阻止程序使用专门的数据类型(例如int,short或long) 来允许对于一种特定系统的每个实现选择所要求的数据类型。在需要存储进程ID的地方，分配类型为pid_t的一个变量(注意，程序1-5已对名为pid的变量这样做了)。在各种不同的实现中，这种数据类型的定义可能是不同的，但是这种差别现在只出现在一个头文件中。我们只需在另一个系统上重新编辑应用程序。

1.7 出错处理

当UNIX函数出错时，往常返回一个负值，而且整型变量errno通常设置为具有特定信息的一个值。例如，open函数如成功执行则返回一个非负文件描述符，如出错则返回－1。在open出错时，有大约15种不同的errno值(文件不存在，许可权问题等)。某些函数并不返回负值而是使用另一种约定。例如，返回一个指向对象的指针的大多数函数，在出错时，将返回一个null指针。

文件<errno.h>中定义了变量errno以及可以赋与它的各种常数。这些常数都以E开头，另外，UNIX手册第2部分的第1页， intro(2) 列出了所有这些出错常数。例如，若 errno等于常数EACCES，这表示产生了权限问题(例如，没有打开所要求文件的权限)。POSIX定义errno为：

```
extern int errno;
```

> POSIX.1中errno的定义较C标准中的定义更为苛刻。C标准允许errno是一个宏，它扩充成可修改的整型左值(lvalue) (例如返回一个指向出错数的指针的函数)。

对于errno应当知道两条规则。第一条规则是：如果没有出错，则其值不会被一个例程清除。因此，仅当函数的返回值指明出错时，才检验其值。第二条是：任一函数都不会将 errno值设置为0，在<errno.h>中定义的所有常数都不为0。

C标准定义了两个函数，它们帮助打印出错信息。

```
#include <string.h>

char *strerror(int errnum);
```

返回：指向消息字符串的指针

此函数将*errnum*(它通常就是errno值) 映射为一个出错信息字符串，并且返回此字符串的指针。

perror函数在标准出错上产生一条出错消息(基于errno的当前值)，然后返回。

```
#include <stdio.h>

void perror(const char *msg);
```

它首先输出由*msg*指向的字符串，然后是一个冒号，一个空格，然后是对应于 errno值的出错信息，然后是一个新行符。

实例

程序1-6显示了这两个出错函数的使用方法。

程序1-6 例示strerror和perror

```
#include    <errno.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));

    errno = ENOENT;
    perror(argv[0]);

    exit(0);
}
```

如果此程序经编译，结果送入文件a.out，则有：

```
$ a.out
EACCES: Permission denied
a.out: No such file or directory
```

注意，我们将程序名 (argv〔0〕，其值是a.out) 作为参数传递给 perror。这是一个标准的UNIX惯例。使用这种方法，如程序作为管道线的一部分执行，如：

```
prog1 < inputfile | prog2 | prog3 > outputfile
```

则我们就能分清三个程序中的哪一个产生了一条特定的出错消息。

本书中的所有实例基本上都不直接调用strerror或perror，而是使用附录B中的出错函数。该附录中的出错函数使用了ANSI C的可变参数表设施，用一条C语句就可处理出错条件。

1.8 用户标识

1.8.1 用户ID

口令文件登录项中的用户ID（user ID）是个数值，它向系统标识各个不同的用户。系统管理员在确定一个用户的登录名的同时，确定其用户ID。用户不能更改其用户ID。通常每个用户有一个唯一的用户ID。下面将介绍内核如何使用用户ID以检验该用户是否有执行某些操作的适当许可权。

用户ID为0的用户为根(root)或超级用户(superuser)。在口令文件中，通常有一个登录项，其登录名为root，我们称这种用户的特权为超级用户特权。我们将在第 4章中看到，如果一个进程具有超级用户特权，则大多数文件许可权检查都不再进行。某些操作系统功能只限于向超

级用户提供，超级用户对系统有自由的支配权。

实例

程序1-7用于打印用户ID和组ID（在下面说明）。

程序1-7 打印用户ID和组ID

```
#include    "ourhdr.h"

int
main(void)
{
    printf("uid = %d, gid = %d\n", getuid(), getgid());
    exit(0);
}
```

调用getuid和getgid以返回用户ID和组ID。运行该程序，产生：

```
$ a.out
uid = 224, gid = 20
```

1.8.2 组ID

口令文件登录项也包括用户的组ID（group ID），它也是一个数值。组ID也是由系统管理员在确定用户登录名时分配的。一般来说，在口令文件中有多个记录项具有相同的组ID。在UNIX下，组被用于将若干用户集合到课题或部门中去。这种机制允许同组的各个成员之间共享资源(例如文件)。4.5节将说明可以设置文件的许可权使组内所有成员都能存取该文件，而组外用户则不能。

组文件将组名映射为数字组ID，它通常是/etc/group。

对于许可权使用数值用户ID和数值组ID是历史上形成的。系统中每个文件的目录项包含该文件所有者的用户ID和组ID。在目录项中存放这两个值只需4个字节(假定每个都以双字节的整型值存放)。如果使用8字节的登录名和8字节的组名，则需较多的磁盘空间。但是对于用户而言，使用名字比使用数值方便，所以口令文件包含了登录名和用户ID之间的映射关系，而组文件则包含了组名和组ID之间的映射关系。例如UNIX ls-l命令使用口令文件将数值用户ID映射为登录名，从而打印文件所有者的登录名。

1.8.3 添加组ID

除了在口令文件中对一个登录名指定一个组ID外，某些UNIX版本还允许一个用户属于另外一些组。这是从4.2 BSD开始的，它允许一个用户属于多至16个另外的组。登录时，读文件/etc/group，寻找列有该用户作为其成员的前16个登记项就可得到该用户的添加组ID（supplementary group ID）。

1.9 信号

信息是通知进程已发生某种条件的一种技术。例如，若某一进程执行除法操作，其除数为0，则将名为SIGFPE的信号发送给该进程。进程如何处理信号有三种选择：

(1) 忽略该信号。有些信号表示硬件异常，例如，除以0或访问进程地址空间以外的单元等，

因为这些异常产生的后果不确定，所以不推荐使用这种处理方式。

(2) 按系统默认方式处理。对于0除，系统默认方式是终止该进程。

(3) 提供一个函数，信号发生时则调用该函数。使用这种方式，我们将能知道什么时候产生了信号，并按所希望的方式处理它。

很多条件会产生信号。有两种键盘方式，分别称为中断键 (interrupt key，通常是Delete键或Ctrl-C)和退出键(quit key，通常是Ctrl-\)，它们被用于中断当前运行进程。另一种产生信号的方法是调用名为kill的函数。在一个进程中调用此函数就可向另一个进程发送一个信号。当然这样做也有些限制：当向一个进程发送信号时，我们必需是该进程的所有者。

实例

回忆一下基本shell程序(见程序1-5)。如果调用此程序，然后键入中断键，则执行此程序的进程终止。产生这种后果的原因是：对于此信号 (SIGINT)的系统默认动作是终止此进程。该进程没有告诉系统核对此信号作何处理，所以系统按默认方式终止该进程。

为了更改此程序使其能捕捉到该信号，它需要调用 signal函数，指定当产生SIGINT信号时要调用的函数名。因此编写了名为 sig_int的函数，当其被调用时，它只是打印一条消息，然后打印一个新提示符。在程序1-5中加了12行构成了程序1-8(添加的12行以行首的+号指示)。

程序1-8 从标准输入读命令并执行

```
    #include    <sys/types.h>
    #include    <sys/wait.h>
+   #include    <signal.h>
    #include    "ourhdr.h"

+   static void sig_int(int);       /* our signal-catching function */
+
    int
    main(void)
    {
        char    buf[MAXLINE];
        pid_t   pid;
        int     status;

+       if (signal(SIGINT, sig_int) == SIG_ERR)
+           err_sys("signal error");
+
        printf("%% ");  /* print prompt (printf requires %% to print %) */
        while (fgets(buf, MAXLINE, stdin) != NULL) {
            buf[strlen(buf) - 1] = 0;   /* replace newline with null */

            if ( (pid = fork()) < 0)
                err_sys("fork error");

            else if (pid == 0) {        /* child */
                execlp(buf, buf, (char *) 0);
                err_ret("couldn't execute: %s", buf);
                exit(127);
            }

            /* parent */
            if ( (pid = waitpid(pid, &status, 0)) < 0)
                err_sys("waitpid error");
            printf("%% ");
        }
        exit(0);
+   }
+
```

```
+ void
+ sig_int(int signo)
+ {
+     printf("interrupt\n%% ");
  }
```

因为大多数重要的应用程序都将使用信号，所以第10章将详细介绍信号。

1.10 UNIX时间值

长期以来，UNIX系统一直使用两种不同的时间值：

(1) 日历时间。该值是自1970年1月1日00:00:00以来国际标准时间（UTC）所经过的秒数累计值（早期的手册称UTC为格林尼治标准时间）。这些时间值可用于记录文件最近一次的修改时间等。

(2) 进程时间。这也被称为CPU时间，用以度量进程使用的中央处理机资源。进程时间以时钟滴答计算，多年来，每秒钟取为50、60或100个滴答。系统基本数据类型clock_t保存这种时间值。另外，POSIX定义常数CLK_TCK，用其说明每秒滴答数。（常数CLK_TCK现在已不再使用。2.5.4节将说明如何用sysconf函数得到每秒时钟滴答数。）

当度量一个进程的执行时间时（见3.9节），UNIX系统使用三个进程时间值：

- 时钟时间。
- 用户CPU时间。
- 系统CPU时间。

时钟时间又称为墙上时钟时间（wall clock time）。它是进程运行的时间总量，其值与系统中同时运行的进程数有关。在我们报告时钟时间时，都是在系统中没有其他活动时进行度量的。

用户CPU时间是执行用户指令所用的时间量。系统CPU时间是为该进程执行内核所经历的时间。例如，只要一个进程执行一个系统服务，例如read或write，则在内核内执行该服务所花费的时间就计入该进程的系统CPU时间。用户CPU时间和系统CPU时间的和常被称为CPU时间。

要取得任一进程的时钟时间、用户时间和系统时间很容易——只要执行命令time(1)，其参数是要度量其执行时间的命令，例如：

```
$ cd /usr/include
$ time grep _POSIX_SOURCE */*.h > /dev/null

real    0m19.81s
user    0m0.43s
sys     0m4.53s
```

time命令的输出格式与所使用的shell有关。

8.15节将说明一个运行进程如何取得这三个时间。关于时间和日期的一般说明见6.9节。

1.11 系统调用和库函数

所有的操作系统都提供多种服务的入口点，由此程序向内核请求服务。各种版本的UNIX都提供经良好定义的有限数目的入口点，经过这些入口点进入内核，这些入口点被称为系统调用(system call)。系统调用是不能更改的一种UNIX特征。UNIX第7版提供了约50个系统调用，4.3+BSD提供了约110个，而SVR4则提供了约120个。

系统调用界面总是在《UNIX程序员手册》的第2部分中说明。其定义也包括在C语言中。这与很多早期的操作系统不同，这些系统按传统方式在机器的汇编语言中定义内核入口点。

UNIX所使用的技术是为每个系统调用在标准C库中设置一个具有同样名字的函数。用户进程用标准C调用序列来调用这些函数，然后，函数又用系统所要求的技术调用相应的内核服务。例如函数可将一个或多个C参数送入通用寄存器，然后执行某个产生软中断进入内核的机器指令。从应用角度考虑，可将系统调用视作为C函数。

《UNIX程序员手册》的第3部分定义了程序员可以使用的通用函数。虽然这些函数可能会调用一个或多个内核的系统调用，但是它们并不是内核的入口点。例如， printf函数会调用write系统调用以进行输出操作，但函数strcpy(复制一字符串)和atoi(变换ASCII为整数)并不使用任何系统调用。

从执行者的角度来看，系统调用和库函数之间有重大区别，但从用户角度来看，其区别并不非常重要。在本书中系统调用和库函数都以 C函数的形式出现，两者都对应用程序提供服务，但是，我们应当理解，如果希望的话，我们可以替换库函数，但是通常却不能替换系统调用。

以存储器分配函数 malloc为例。有多种方法可以进行存储器分配及与其相关的无用区收集操作 (最佳适应，首次适应等)，并不存在对所有程序都最佳的一种技术。UNIX系统调用中处理存储器分配的是sbrk(2)，它不是一个通用的存储器管理器。它增加或减少指定字节数的进程地址空间。如何管理该地址空间却取决于进程。存储器分配函数 malloc(3)实现一种特定类型的分配。如果我们不喜欢其操作方式，则可以定义自己的 malloc函数，它可能将使用sbrk系统调用。事实上，有很多软件包，它们实现自己的存储器分配算法，但仍使用 sbrk系统调用。图1-1显示了应用程序、 malloc函数以及sbrk系统调用之间的关系。

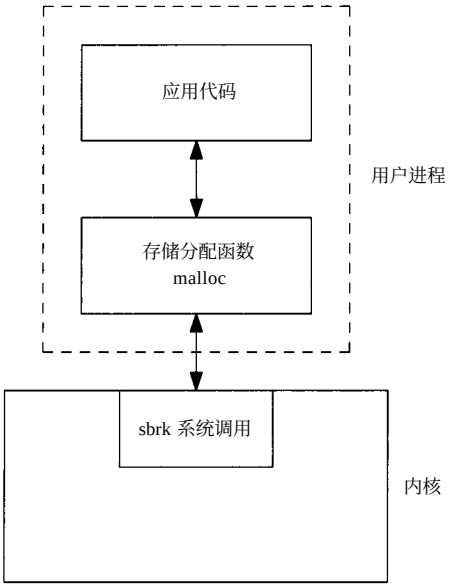

图1-1 malloc函数和sbrk系统调用

从中可见，两者职责不同，相互分开，内核中的系统调用分配另外一块空间给进程，而库函数malloc则管理这一空间。

另一个可说明系统调用和库函数之间的差别的例子是， UNIX提供决定当前时间和日期的界面。某些操作系统提供一个系统调用以返回时间，而另一个则返回日期。任何特殊的处理，例如正常时制和夏时制之间的转换，由内核处理或要求人为干预。 UNIX则不同，它只提供一条系统调用，该系统调用返回国际标准时间 1970年1月1日零点以来所经过的秒数。对该值的任何解释，例如将其变换成人们可读的，使用本地时区的时间和日期，都留给用户进程运行。在标准C库中，提供了若干例程以处理大多数情况。这些库函数处理各种细节，例如各种夏时制算法。

应用程序可以调用系统调用或者库函数，而很多库函数则会调用系统调用。这在图 1-2中显示。

系统调用和库函数之间的另一个差别是：系统调用通常提供一种最小界面，而库函数通

常提供比较复杂的功能。我们从 sbrk系统调用和malloc库函数之间的差别中可以看到这一点，在以后当比较不带缓存的 I/O函数（见第3章）以及标准I/O函数（见第5章）时，还将看到这种差别。

进程控制系统调用（fork, exec和wait）通常由用户的应用程序直接调用（请回忆程序1-5中的基本shell）。但是为了简化某些常见的情况，UNIX系统也提供了一些库函数；例如system和popen。8.12节将说明system函数的一种实现，它使用基本的进程控制系统调用。10.18节还将强化这一实例以正确地处理信号。

为使读者了解大多数程序员应用的UNIX系统界面，我们不得不既说明系统调用，只介绍某些库函数。例如若只说明sbrk系统调用，那么就会忽略很多应用程序使用的malloc库函数。

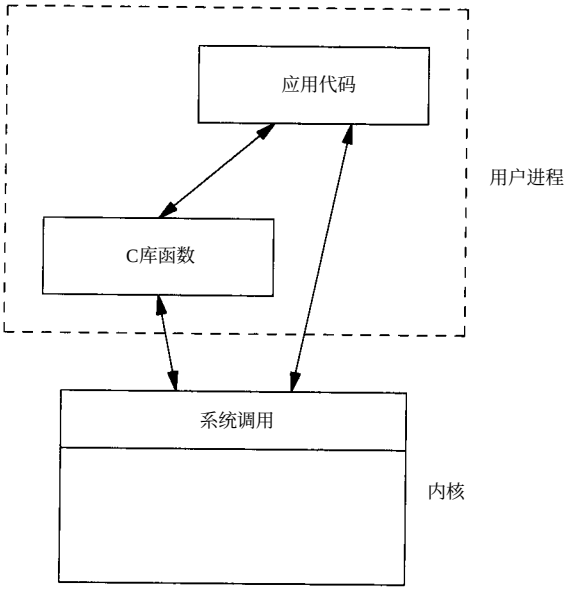

图1-2 C库函数和系统调用之间的差别

本书除了必须要区分两者时，都将使用术语函数（ function）来指代系统调用和库函数两者。

1.12 小结

本章快速浏览了UNIX。说明了某些以后会多次用到的基本术语，介绍了一些小的 UNIX程序的实例，从中可感知到本书的其余部分将会进一步介绍的内容。

下一章是关于UNIX的标准化，以及这方面的工作对当前系统的影响。标准，特别是 ANSI C标准和POSIX.1标准将影响本书的余下部分。

习题

1.1 在系统上查证，除根目录外，目录. 和.. 是不同的。

1.2 分析程序1-4的输出，说明进程ID为852和853的进程可能会发生什么情况？

1.3 在1.7节中，perror的参数是用ANSI C的属性const定义的，而rerror的整型参数则没有用此属性定义，为什么？

1.4 附录B包含了出错处理函数err_sys，当调用该函数时，保存了errno的值，为什么？

1.5 若日历时间存放在带符号的32位整型数中，那么到哪一年它将溢出？

1.6 若进程时间存放在带符号的32位整型数中，而且每秒为100滴答，那么经过多少天后该时间值将会溢出？