10.系统级I/O

10.1 Unix I/O

一个Linux文件就是一个m个字节的序列：$B_0，B_1，\cdots，B_k，\cdots，B_{m-1}$

所有的I/O设备都被模型化为文件，所有的输入和输出都被当作对应文件的读和写来执行。将设备映射为文件的方式，允许Linxu内核引用出一个简单、低级的应用接口，称为Unix I/O，是的所有的输入和输出都能以一种统一且一致的方式来执行

- 打开文件：一个应用程序通过要求内核打开相应的文件，来宣告他想访问一个I/O设备。内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件。内核记录有关这个打开文件的所有信息。应用程序只需记住这个描述符
- Linux shell创建的每个进程开始时都有三个打开的文件：标准输入（描述符0），标准输出（1），和标准错误（2）。头文件< unistd.h> 定义了常量 STDIN_ FILENO、STDOUT_FILENO 和 STDERR_FILENO，它们可用来代替显式的描述符值。
- 改变当前的文件位置。对于每个打开的文件，内核保持着一个文件位置k，初始为0，这个文件位置是从文件开头起始的字节偏移量。应用程序能够通过执行seek操作，显式地设置文件的当前位置k。
- 读写文件。一个读操作就是从文件复制n>0个字节到内存，从当前文件位置k开始，然后将看增加到k+n。给定一个大小为m字节的文件，当k》m时执行读操作会触发一个end of file（EOF)的条件。类似的，写操作就是从内存复制n>0个字节到一个文件，从当前位置k开始，然后更新k。
- 关闭文件。应用程序完成对文件的访问后，通知内核关闭文件。作为响应，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中。

10.2 文件

每个文件都有一个type类型：

- 普通文件，包含任意数据。通常要区分文本文件和二进制文件，文本文件是只含有ASCII或Unicode字符的普通文件；二进制文件是所有其他的文件。
  文本文件包含了一个文本行序列，其中每一行都是一个字符序列，以一个新行符（"\n")结束，数字值为0x0a.
- 目录是包含一组链接的文件，每个链接将一个文件名映射到一个文件。每个目录至少包含两个条目：“.”是到该目录自身的链接，以及“..”是到目录层次中父目录的链接。
- 套接字是用来与另一个进程进行跨网络通信的文件
- 其他文件类型包括命名通道、符号链接、，以及字符和块设备。
  Linux内核将所有文件都组织成一个目录层次结构。

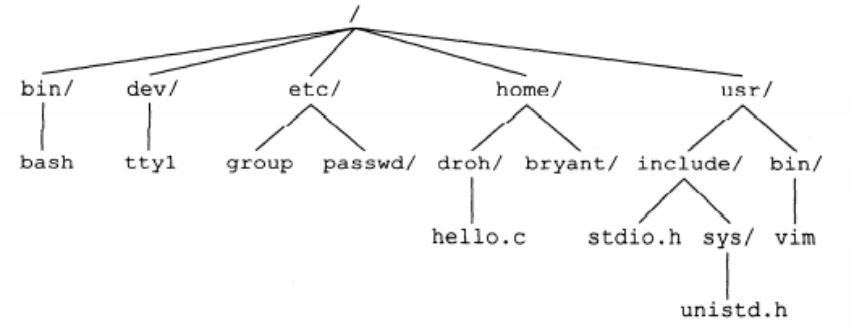

图 10-1 Linux 目录层次的一部分。尾部有斜杠表示是目录

绝对路径：以一个/开始，表示从根节点开始的路径
相对路径：以文件名开始，表示从当前工作目录开始的路径。

10.3 打开和关闭文件

进程是通过调用 open 函数来打开一个已存在的文件或者创建一个新文件的：

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

int open(char *filename, int flags, mode_t mode);
                                        返回：若成功则为新文件描述符，若出错为－1。
```

open函数将filename转换为一个文件描述符，并且返回描述符数字。返回的描述符总是在进程中当前没有打开的最小描述符。

flags 参数指明了进程打算如何访问这个文件：

- O_RDONLY：只读。
- O_WRONLY：只写。
- O_RDWR：可读可写。

flags 参数也可以是一个或者更多位掩码的或，为写提供给一些额外的指示：

- O_CREAT：如果文件不存在，就创建它的一个截断的(truncated)(空)文件。
- O_TRUNC：如果文件已经存在，就截断它。
- O_APPEND：在每次写操作前，设置文件位置到文件的结尾处。

例如，下面的代码说明的是如何打开一个已存在文件，并在后面添加一些数据：

```
fd = Open("foo.txt", O_WRONLY|O_APPEND, 0);
```

mode 参数指定了新文件的访问权限位。这些位的符号名字如图 10-2 所示。

掩码	描述
S_IRUSR	使用者（拥有者）能够读这个文件
S_IWUSR	使用者（拥有者）能够写这个文件
S_IXUSR	使用者（拥有者）能够执行这个文件
S_IRGRP	拥有者所在组的成员能够读这个文件
S_IWGRP	拥有者所在组的成员能够写这个文件
S_IXGRP	拥有者所在组的成员能够执行这个文件
S_IROTH	其他人（任何人）能够读这个文件
S_IWOTH	其他人（任何人）能够写这个文件
S_IXOTH	其他人（任何人）能够执行这个文件

图 10-2　访问权限位。在 sys/stat.h 中定义

mode 参数指定了新文件的访问权限位。这些位的符号名字如图 10-2 所示。

作为上下文的一部分，每个进程都有一个 umask，它是通过调用 umask 函数来设置的。当进程通过带某个 mode 参数的 open 函数调用来创建一个新文件时，文件的访问权限位被设置为 mode & ~umask。例如，假设我们给定下面的 mode 和 umask 默认值：

```
#define DEF_MODE   S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
#define DEF_UMASK  S_IWGRP|S_IWOTH
```

接下来，下面的代码片段创建一个新文件，文件的拥有者有读写权限，而所有其他的用户都有读权限：

```
umask(DEF_UMASK);
fd = Open("foo.txt", O_CREAT|O_TRUNC|O_WRONLY, DEF_MODE);
```

10.4 读和写文件

应用程序是通过分别调用 read 和 write 函数来执行输入和输出的。

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t n);
                                    返回：若成功则为读的字节数，若 EOF 则为 0，若出错为－1。
ssize_t write(int fd, const void *buf, size_t n);
                                    返回：若成功则为写的字节数，若出错则为－1。
```

某些情况下，read和write传送的字节比应用程序要求的要少。这些不足值不表示有错误。

出现这样情况的原因有：

- 读时遇到EOF。
- 从终端读文本行：如果打开文件是与终端相关联的(如键盘和显示器），那么每个read函数将一次传送一个文本行，返回的不足值等于文本行的大小。
- 读和写网络套接字，内存的缓冲约束和较长的网络延迟会引起read和write返回不足值。Linux管道（pipe）调用read和write时，也可能出现不足值。
  实际上，除了EOF，在你读磁盘文件时不会遇到不足值，而且在写磁盘文件时，也不会遇到不足值。

10.5 用RIO包健壮地写

RIO提供了两类不同的函数：

- 无缓冲的输入输出函数。对于将二进制数据读写到网络和从网络读写二进制数据尤其有用
- 带缓冲的输入函数。

10.5.1 RIO的无缓冲的输入输出函数

```
#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
                返回：若成功则为传送的字节数，若 EOF 则为 0(只对 rio_readn 而言)，若出错则为－1。
```

实际就是将读写时可能遇到的错误包含在函数中考虑。并且每个函数在遇到中断时会手动地重启。

10.5.2 RIO的带缓冲的输入函数

rio_read为read的带缓冲区版本，rio _readnb为调用rio _read的包装函数，会多次调用rioread函数来达到读取用户要求的字节数。

10.6 读取文件元数据

应用程序能够通过调用stat和fstat函数，检索到关于文件的信息（有时也称为文件的元数据）。

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

int stat(const char *filename, struct stat *buf);   以文件名输入
int fstat(int fd, struct stat *buf);
                以文件描述符输入                    返回：若成功则为 0，若出错则为－1。
```

statbuf.h (included by sys/stat.h)

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;     /* Device */
    ino_t         st_ino;     /* inode */
    mode_t        st_mode;    /* Protection and file type */
    nlink_t       st_nlink;   /* Number of hard links */
    uid_t         st_uid;     /* User ID of owner */
    gid_t         st_gid;     /* Group ID of owner */
    dev_t         st_rdev;    /* Device type (if inode device) */
    off_t         st_size;    /* Total size, in bytes */
    unsigned long st_blksize; /* Block size for filesystem I/O */
    unsigned long st_blocks;  /* Number of blocks allocated */
    time_t        st_atime;   /* Time of last access */
    time_t        st_mtime;   /* Time of last modification */
    time_t        st_ctime;   /* Time of last change */
};
```

文件访问许可位

statbuf.h (included by sys/stat.h)

图 10-9　stat 数据结构

S_ISREG(m)。这是一个普通文件吗?
S_ISDIR(m)。这是一个目录文件吗?
S_ISSOCK(m)。这是一个网络套接字吗?

读取案例:

code/io/statcheck.c

```
1   #include "csapp.h"
2
3   int main (int argc, char **argv)
4   {
5       struct stat stat;
6       char *type, *readok;
7
8       Stat(argv[1], &stat);
9       if (S_ISREG(stat.st_mode))     /* Determine file type */
10          type = "regular";
11      else if (S_ISDIR(stat.st_mode))
12          type = "directory";
13      else
14          type = "other";
15      if ((stat.st_mode & S_IRUSR)) /* Check read access */
16          readok = "yes";
17      else
18          readok = "no";
19
20      printf("type: %s, read: %s\n", type, readok);
21      exit(0);
22  }
```

code/io/statcheck.c

图 10-10　查询和处理一个文件的 st_mode 位

10.7 读取目录内容

应用程序可以用 readdir 系列函数来读取目录的内容。

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

DIR *opendir(const char *name);
```

返回：若成功，则为处理的指针；若出错，则为 NULL。

以路径名作为参数，返回指向目录流的指针。流是对条目有序列表的抽象，在这里是指目录项的列表。

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

返回：若成功，则为指向下一个目录项的指针；若没有更多的目录项或出错，则为 NULL。

每次对 readdir 的调用返回的都是指向流 dirp 中下一个目录项的指针，或者，如果没有更多目录项则返回 NULL。每个目录项都是一个结构，其形式如下：

```
struct dirent {
    ino_t d_ino;       /* inode number */
    char  d_name[256]; /* Filename */
};
```

如果出错，readdir也返回NULL并设置error。唯一能区分错误和流结束情况的方法是检查自调用readdir以来errno是否被修改过。

```
#include <dirent.h>

int closedir(DIR *dirp);
```

返回：成功为 0；错误为 −1。

10.8 共享文件

内核用三个相关的数据结构来表述打开的文件

- 描述符表。（对于进程所打开的文件的描述，每个进程一个）每个进程都有他独立的描述符表，他的表项是由进程打开的文件描述符来索引的。每个打开的描述符表项指向文件表中的一个表项。
- 文件表。（对于打开文件情况的描述，每个打开文件一个）打开文件的集合是有一张文件表来表示的，所有的进程共享这张表。每个文件表的表项组成包括当前的文件位置，引用计数以及一个指向v-node表中对应表项的指针。
- v-node表。（每个文件一个，对文件更细节信息的描述）所有进程共享，每个表包含stat结构中的大多数信息。

图 10-12 展示了一个示例，其中描述符 1 和 4 通过不同的打开文件表表项来引用两个不同的文件。这是一种典型的情况，没有共享文件，并且每个描述符对应一个不同的文件。

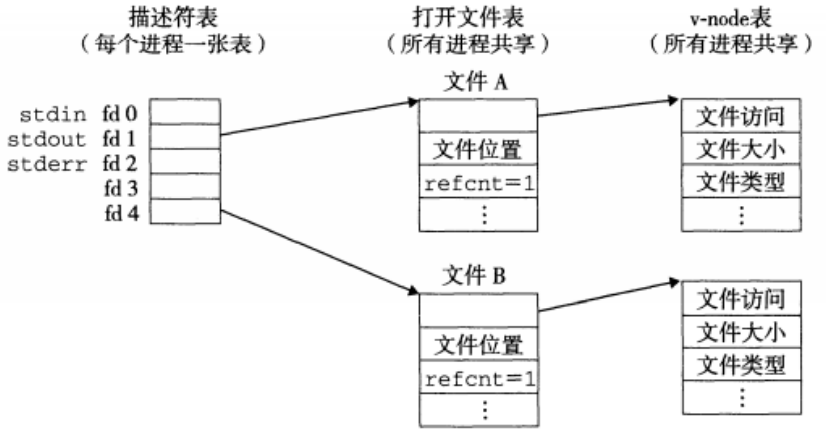

图 10-12 典型的打开文件的内核数据结构。在这个示例中，两个描述符引用不同的文件。没有共享

如图 10-13 所示，多个描述符也可以通过不同的文件表表项来引用同一个文件。例如，如果以同一个 filename 调用 open 函数两次，就会发生这种情况。关键思想是每个描述符都有它自己的文件位置，所以对不同描述符的读操作可以从文件的不同位置获取数据。

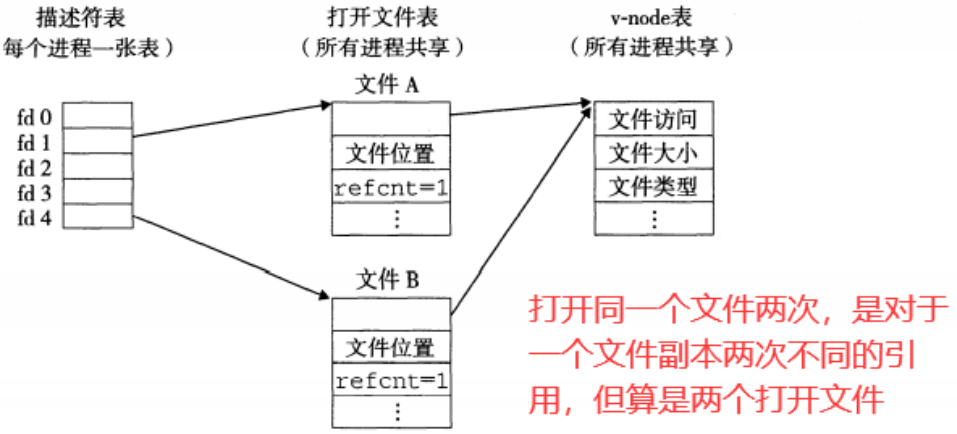

图 10-13 文件共享。这个例子展示了两个描述符通过两个打开文件表表项共享同一个磁盘文件

10.9 文件重定向

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

返回：若成功则为非负的描述符，若出错则为−1。

dup2函数复制描述符表项oldfd到描述符表表项newfd，覆盖描述符表表项newfd以前的内容。（就是让new指向old指向的文件描述符）

假设在调用 dup2(4,1)之前，我们的状态如图 10-12 所示，其中描述符 1(标准输出)对应于文件 A(比如一个终端)，描述符 4 对应于文件 B(比如一个磁盘文件)。A 和 B 的引用计数都等于 1。图 10-15 显示了调用 dup2(4,1)之后的情况。两个描述符现在都指向文件 B；文件 A 已经被关闭了，并且它的文件表和 v-node 表表项也已经被删除了；文件 B 的引用计数已经增加了。从此以后，任何写到标准输出的数据都被重定向到文件 B。

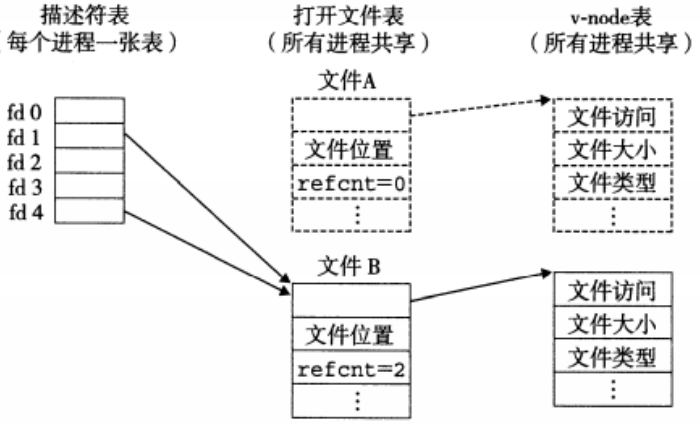

图 10-15　通过调用 dup2(4,1)重定向标准输出之后的内核数据结构。初始状态如图 10-12 所示

10.10 标准I/O

10.11 该使用何种I/O

使用哪个I/O函数的指导原则

- G0.只要有可能就使用I/O。（可能stat除外，因为标准库中没有与他对应的函数）
- G1.不要使用scanf或rio_readlineb来读二进制文件。他们是专门设计用来读文本文件的。
- G2.对网络套接字使用RIO函数

标准I/O的限制

- 一：跟在输出函数之后的输入函数，如果没有对fflush，fessk，fsetpos，rewind的调用，一个输入函数不能跟随在一个输出函数之后
- 二：跟在输入函数之后的输出函数，没有fseek，fsetpos或rewind的调用，一个输出函数不能跟随在一个输入函数之后，除非该输入函数遇到了一个文件结束。