

文件的系统调用

- 系统调用 **open**
- 系统调用 **creat**
- 系统调用 **close**
- 系统调用 **link**和 **unlink**
- 系统调用 **read** 和 **write**
- 其它系统调用

关于系统调用和库函数

C语言支持一系列的库函数的调用，其中最基本的是**studio**库函数。事实上，**库函数**只是**C**语言在**较高层次**上调用的方式，**系统调用**是**更低层次**的与**C**语言的界面，是**内核提供**给用户调用的函数。

C程序使用系统调用的语句和调用库函数的语句形式完全相同，但是二者运行的环境和运行机制截然不同。

从运行环境来看，库函数使用依赖于所运行的用户环境，程序调用库函数时，它运行的目标代码是属于程序的，程序处于“用户态”执行；而系统调用的使用不依赖于它运行的用户环境，是UNIX内核提供的低层服务，系统调用时所执行的代码是属于内核的，程序处于“核心态”执行。

系统调用和库函数对于出错的处理也不尽相同。

对于**studio**库中的函数，出错时会返回一个预定义的常量**EOF**或**NULL**；许多库函数在出错时常常返回 **0** 或 **-1**；有些库函数则返回某种出错代码。

系统调用的出错处理比较简单，每个系统调用在出错时都返回 **-1**，在调用成功时返回 **0** (或某些其他具有特定意义的整数值)。Linux系统调用把出错代码放在一个名为**error**的外部变量中，在程序中包含头文件**error.h**，便可以得到**error**的错误代码。

程序库函数的调用最终还是要通过系统调用来实现，库函数一般执行一条指令，该指令（操作系统陷阱operating system trap）将进程执行方式变为核心态，然后使内核为系统调用执行代码。

编写一个高效的程序，不依靠系统调用的支持是不可想象的。甚至可以这么说，对Linux系统调用熟悉的程度决定了在Linux环境下编程水平的高低。

1 系统调用 open

- **open**是进程存取一个文件中的数据必需首先做的系统调用，打开文件。
- **open**系统调用的声明格式如下：
 - **int open(const char *pathname, int oflag);**
 - **int open(const char *pathname, int oflag, mode_t mode);**
- 系统调用的头文件：
 - **#include <sys/types.h>**
 - **#include <sys/stat.h>**
 - **#include <fcntl.h>**

返回：若成功为文件描述符，若出错为 **-1**。

pathname是要打开或创建的文件的名字。**oflag**参数可用来说明此函数的多个选择项。用下列一个或多个常数进行或运算构成**oflag**参数(这些常数定义在<fcntl.h>头文件中):

- **O_RDONLY** 只读打开。
- **O_WRONLY** 只写打开。
- **O_RDWR** 读写打开。

这三个常数应只指定一个。下列常数则是可选择的:

- **O_APPEND** 每次写时都加到文件的尾端。
- **O_CREAT** 若此文件不存在则创建它。使用此选择项时, 需同时说明第三个参数**mode**, 用其说明该新文件的存取许可权位。

- **O_EXCL** 如果同时指定了**O_CREAT**，而文件已经存在，则出错。这可测试一个文件是否存在，如果不存在则**创建**此文件成为一个**原子操作**。
- **O_TRUNC** 如果此文件存在，而且为只读或只写成功打开，则将其长度**截短为0**。

open实例:

- #include <sys/types.h>
 - #include <sys/stat.h>
 - #include <fcntl.h>
 - #include <errno.h>
 - #include <stdio.h>
 - extern int errno;
 - int main()
 - { int fd;
 - if ((fd = open("test.c", O_CREAT|O_EXCL, S_IRWXU)) == -1)
 - { printf("Open Error\n Error No = %d\n", errno);
 - if (errno == EEXIST)
 - printf("Open error, because file already exist!\n");
 - } else printf("Success\n");
 - return 0;
 - }
- O_CREAT** : 当文件不存在时, 将建立该文件。
- **O_EXCL** : **O_CREAT**一起用
 - **S_IRWXU**: 文件主有读、写、执行的权力。
 - **EEXIST**: 上述标示的文件名已经存在。

2 系统调用 creat

- **creat**是进程新建一个文件时使用的系统调用。

新建文件的功能也可以由**open**调用实现。

- **creat**系统调用的声明格式如下：

- **int creat(const char *pathname, mode_t mode);**

- 系统调用的头文件：

- **#include <sys/types.h>**

- **#include <sys/stat.h>**

- **#include <fcntl.h>**

系统调用 **creat**

➤ **creat**系统调用中的参数:

pathname和**mode**的含义与系统调用**open**中的一样。

➤ 如果**pathname**指向的**文件不存在**，核心就以指定的**文件名和许可权**创建一个新文件；

➤ 如果**pathname**指向的**文件存在**，核心就将该文件**截断**，**释放**以前数据所占用的**磁盘块**。

返回：若成功为只写打开的文件描述符，若出错为-1。

此函数等效于：

open(*pathname*, **O_WRONLY** | **O_CREAT** | **O_TRUNC**, *mode*)

在早期的**UNIX**版本中，**open**的第二个参数只能是**0**、**1**或**2**。没有办法打开一个尚未存在的文件，因此需要另一个系统调用**creat**以创建新文件。现在，**open**函数提供了选择项**O_CREAT**和**O_TRUNC**，于是也就不再需要**creat**函数了。

3 系统调用 read 和 write

- **read**是从文件中读取指定长度的数据到内存中
- 其声明格式如下：
 - `ssize_t read(int fd, void *buf, size_t count);`
- 系统调用用的头文件：
 - `#include <unistd.h>`
- 功能：从文件描述符**fd**所指的文件中读取**count**个字节到**buf**所指向的内存缓冲中。

返回：读到的字节数，若已到文件尾为**0**，若出错为**-1**。

在**ANSI C**中，类型**void ***用于表示类属指针。其返回值必须是一个**带符号整数(ssize_t)**，以返回**正字节数**、**0**(表示文件尾端)或 **-1** (出错)。第三个参数在历史上是一个不带符号整数，以允许一个**16位**的实现可以一次读或写至**65534**个字节。在**1990 POSIX.1**标准中，引进了新的基本系统数据类型**ssize_t**以提供带符号的返回值， **size_t**则被用于第三个参数。

write系统调用

➤ **write**是将内存中的数据写入文件

➤ 其声明格式如下：

```
ssize_t write(int fd, const void *buf, size_t count);
```

➤ 系统调用加入头文件：

– **#include <unistd.h>**

➤ 功能：**write**将buf所指内存中的count个字节写入文件描述符fd所指的文件。

返回：若成功为已写的字节数，若出错为-1

write出错的一个常见原因是：磁盘已写满，或者超过了对一个给定进程的文件长度限制。

对于普通文件，写操作从文件的当前位移量处开始。如果在打开该文件时，指定了O_APPEND选择项，则在每次写操作之前，将文件位移量设置在文件的当前结尾处。在一次成功写之后，该文件位移量增加实际写的字节数。

//将标准输入复制到标准输出

- **#include <unistd.h>**
- **#define BUFSIZE 1024**
- **int main()**
- **{ int n;**
- **char buf[BUFSIZE];**
- **while ((n=read(STDIN_FILENO,buf,BUFSIZE))>0)**
- **if (write(STDOUT_FILENO,buf,n)!=n)**
- **printf("write error\n");**
- **if (n<0) printf("read error\n");**
- **exit(0);**
- **}**

关于该程序应注意下列几点：

- 它从标准输入读，写至标准输出，这就假定在执行本程序之前，这些标准输入、输出已由**shell**安排好。确实，所有常用的**UNIX shell**都提供一种方法，它在标准输入上打开一个文件用于读，在标准输出上创建(或重写)一个文件。
- 很多应用程序假定**标准输入**是**文件描述符0**，**标准输出**是**文件描述符1**。本例中则用两个在**<unistd.h>**中定义的名字**STDIN_FILENO**和**STDOUT_FILENO**。
- 考虑到进程终止时，**UNIX**会关闭所有打开文件描述符，所以此程序并不关闭**输入和输出文件**。
- 本程序对**文本文件**和**二进制代码**文件都能工作，因为对**UNIX**内核而言，这两种文件并无区别。

4 lseek 系统调用

- 每个打开文件都有一个与其相关联的“**当前文件位移量**”。它是一个**非负整数**，用以度量**从文件开始处**计算的字节数。
- **lseek**用来重新定位下一次**read/write**系统调用所用的**文件指针**（也叫**文件偏移量**：**file offset**）
- 进程可以使用系统调用**lseek**来指定**I/O**的位置，从而实现文件的随机存取。
- 也可以形成**空洞文件**

lseek声明格式如下：

- **off_t lseek(int fildes, off_t offset, int whence);**
- 系统调用加入头文件：
 - **#include <sys/types.h>**
 - **#include <unistd.h>**
- 作用：系统调用根据**whence**指定的位置将文件描述符**fildes**指向文件的**文件指针偏移****offset**长度的字节数。

返回：若成功为新的文件位移，若出错为-1。

对参数**offset** 的解释与参数**whence**的值有关。

- 若**whence**是**SEEK_SET**，则将该文件的位移量设置为距**文件开始处offset**个字节。
- 若**whence**是**SEEK_CUR**，则将该文件的位移量设置为其**当前值加offset**，**offset**可为正或负。
- 若**whence**是**SEEK_END**，则将该文件的位移量设置为**文件长度加offset**，**offset**可为正或负。

若**lseek**成功执行，则返回**新的文件位移量**，为此可以用下列方式确定一个**打开文件的当前位移量**：

```
Off_t currpos;
```

```
currpos = lseek(fd, 0, SEEK_CUR);
```

这种方法也可用来确定所涉及的文件是否可以设置位移量。

对于普通文件，则其位移量必须是非负值。因为位移量可能是负值，所以在比较**lseek**的返回值时应当谨慎，不要测试它是否小于**0**，而要测试它是否等于**-1**。

//创建一个具有空洞的文件

- `#include <sys/types.h>`
- `#include <sys/stat.h>`
- `#include <fcntl.h>`
- `#include <errno.h>`
- `#include <unistd.h>`
- `char buf1[]="abcdefghij";`
- `char buf2[]="ABCDEFGHIJ";`
- `int main()`
- `{ int fd;`
- `if ((fd=creat("file.hole",3)) < 0) /*3是FILE_MODE*/`
- `printf("creat error ");`

- if (**write**(fd,buf1,10)!=10)
- printf("buf1 write error ");
- /* offset now=10 */
- if (**lseek**(fd,40,SEEK_SET)==-1)
- printf("lseek error ");
- /* offset now=40 */
- if (**write**(fd,buf2,10)!=10)
- printf("buf2 write error "); /* offset now=50 */
- exit(0);
- }

➤ /* 输出结果:

➤ # a.out

➤ #od -c file.hole //以字符方式观察文件的实际内容

0000000 a b c d e f g h i j \0 \0 \0 \0 \0 \0

0000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

0000040 \0 \0 \0 \0 \0 \0 \0 \0 \0 A B C D E F G H

0000060 I J

0000062

➤ 每一行开始的一个七位数是以八进制形式表示的字节位移量。

5 系统调用 **close**

➤ **close**系统调用：当进程不再使用一个打开的文件时，就应该**关闭该文件**。

➤ 其声明格式如下：

– **int close(int **fd**);**

– 返回：若成功为**0**，若出错为**-1**

➤ **/*其中fd是一个已经打开的文件的文件描述符。*/**

➤ 系统调用的头文件：

#include <unistd.h>

当一个进程终止时，它所有的打开文件都由内核自动关闭。很多程序都使用这一功能而不显式地用**close**关闭打开的文件。

```
#include    "ourhdr.h"

#define BUFSIZE    8192

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

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

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

    exit(0);
}
```

6. 文件链接的概念

- 文件链接：在不同目录文件中的某些文件名所对应的**inode**都指向**同一个索引节点**。即：多个文件对应一个文件。叫**硬链接**。硬链接**不能跨文件系统**。在**inode**（磁盘索引节点）的结构中，**文件链接数**字段的作用：有多少文件名指向该文件（**>1 or =0**）。
- 系统调用的头文件：
 - **#include <unistd.h>**

系统调用 **link**

➤ 系统调用声明形式:

➤ **int link(const char ***oldpath**, const char ***newpath**);**

➤ 其中:

oldpath: **已存在**的文件。

newpath: **被链接**的新文件。

➤ 功能: 系统调用**link**对一个已经存在的文件创建一个新的链接, 是**硬链接**。给文件一个新的文件名字, **一个文件多个名字**, **链接文件不开新的空间**。

只有超级用户进程可以创建指向一个目录的新链接。其原因是这样做可能在文件系统中形成循环，大多数处理文件系统的公用程序都不能处理这种情况。

POSIX1.1允许支持跨越文件系统的链接的实现。

symlink系统调用

➤ 声明形式:

```
int symlink(const char *oldpath, const char *newpath);
```

返回：若成功则为0，若出错则为-1

➤ 系统调用头文件:

```
#include <unistd.h>
```

➤ 作用：系统调用symlink创建一个包含字符串oldpath的名为newpath的符号链接。

➤ 符号链接在运行时被解释，仿佛链接的内容被随后找到的文件或目录替代。符号链接中可以包含上级目录“..”，作为符号链接中路径的组成部分。

➤ 说明:

(1) 符号链接（软链接）：可以指向一个已存在的文件或还不存在的文件。

(2) 可以给一个目录做链接，可以跨文件系统作链接。

(3) 符号链接的许可权是没有意义的。其权限是由其链接的目标确定的。

➤ 注意：符号链接与硬链接不同，删除符号链接所指向的文件，将导致真正的删除。

符号链接是对一个文件的**间接指针**，它与硬链接有所不同，硬链接直接指向文件的i节点。引进符号链接的原因是为了**避免硬链接的一些限制**：

- (a) 硬链接通常要求**链接和文件位于同一文件系统中**，
- (b) 只有超级用户才能创建到**目录**的硬链接。

对**符号链接**以及它指向什么**没有文件系统限制**，任何用户都可创建指向目录的符号链接。符号链接一般用于将一个文件或整个目录结构移到系统中其他某个位置。

系统调用 **link**和**symlink**实例

➤ **Book3.c**

➤ **#include <unistd.h>**

➤ **#include <stdio.h>**

➤ **#include <errno.h>**

➤ **extern int errno;**

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

➤ **{ if (argc != 4)**

➤ **{**

➤ **printf("Usage: link_exam oldfile linkfn symlinkfn \n");**

➤ **exit(1);**

➤ **}**

```
➤ if (link(argv[1],argv[2])==-1)
➤ {   printf("link error\nErrno = %d\n", errno); }
➤ if (symlink(argv[1],argv[3])==-1)
➤ {   printf("symlink error\nErrno = %d\n",
➤                                             errno); }
➤ }
```


➤ **//命令: ./a.out test.c test.link test.symlink**

➤ **./a.out ttt ttt.link ttt.symlink**


结果:

➤ **ttt.link** => **error**

➤ 通过 **ls -li** 看link, **symlink**的作用不同。



```
306470 -rw-r--r-- 2 tjb root 477 Aug 3 11:26 test.c
306470 -rw-r--r-- 2 tjb root 477 Aug 3 11:26 test.link
306473 lrwxrwxrwx 1 tjb root 6 Aug 3 11:26 test.symlink->test.c
```

```
357818 drwxr-xr-x 2 tjb root 1024 Aug 3 11:31 ttt
306474 lrwxrwxrwx 1 tjb root 3 Aug 3 11:34 ttt.symlink->ttt
```

系统调用**unlink**

- **unlink**是清除文件的连接（一个目录表项）。
- 声明格式：
 - **int unlink(const char *pathname);**
 - 返回：若成功则为**0**，若出错则为**-1**
- 系统调用的头文件：
 - **#include <unistd.h>**
- 作用：**unlink**从文件系统中删除目录项，并将**pathname**所引用文件的链接计数减**1**。当链接计数为**0**并且打开此进程数为**0**时，此链接被删除。释放索引节点及其指向的数据块。
- 若**pathname**指符号链接，则该链接被删除。

只有当链接计数达到0时，该文件的内容才可被删除。另外只要有进程打开了该文件，其内容也不能被删除。关闭一个文件时，内核首先检查使该文件打开的进程计数。如果该计数达到0，然后内核检查其链接计数，如果也是0，那么就删除该文件的内容。

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

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");

    if (unlink("tempfile") < 0)
        err_sys("unlink error");

    printf("file unlinked\n");
    sleep(15);
    printf("done\n");

    exit(0);
}
```

运行该程序，其结果是：

\$ ls -l tempfile

查看文件大小

```
-rw-r--r--  1 stevens  9240990 Jul 31 13:42 tempfile
```

\$ df/home

检查空闲空间

Filesystem	kbytes	used	avail	capacity	Mounted on
/dev/sd0h	282908	181979	72638	71%	/home

\$ a.out &

在后台运行程序4-5

1364

shell打印其进程ID

\$ file unlinked

该文件是未连接的

ls -l tempfile

观察文件是否仍然存在

tempfile not found

目录项已删除

\$ df/home

检查空闲空间有无变化

Filesystem	kbytes	used	avail	capacity	Mounted on
/dev/sd0h	282908	181979	72638	71%	/home

\$ done

程序执行结束，关闭所有打开文件

df/home

磁盘空间有效

Filesystem	kbytes	used	avail	capacity	Mounted on
/dev/sd0h	282908	172939	81678	68%	/home

9.2M字节磁盘空间有效

unlink的这种特性经常被程序用来确保即使是在程序崩溃时，它所创建的临时文件也不会遗留下来。进程用**open**或**creat**创建一个文件，然后立即调用**unlink**。因为该文件仍旧是打开的，所以不会将其内容删除。只有当进程关闭该文件或终止时(在这种情况下，内核关闭该进程打开的全部文件)，该文件的内容才被删除。

7 stat系统调用

➤ **stat**系统调用**stat**、**lstat**，**fstat**都是取得文件状态的系统调用，返回指定文件的信息。

➤ 其声明格式如下：

```
int stat(const char *file_name, struct stat *buf);
```

```
int fstat(int fileds , struct stat *buf);
```

```
int lstat(const char *file_name, struct stat *buf);
```

➤ 系统调用**stat**将文件**file_name**的信息存放在参数**buf**所指向的结构中。

stat, lstat, fstat的不同

➤ stat, lstat的区别:

如果文件是**符号链接**，**lstat**返回的是符号链接**本身**的信息，**stat**返回的是符号链接**所指**文件的信息。

➤ stat, fstat的区别:

stat使用**文件名**指向文件；**fstat**使用**文件描述符**指向文件。

stat结构

- 这三个调用的返回值都存放在一个stat结构中，该结构声明如下：

➤ struct stat

```
{ dev_t      st_dev;    /* 文件所在设备号 */
  ino_t      st_ino;    /* 索引节点号 */
  — umode_t   st_mode;   /* 文件模式 */
  — nlink_t   st_nlink;  /* 与该文件硬链接的数量 */
  — uid_t     st_uid;    /* 属主的用户ID(UID) */
  — gid_t     st_gid;    /* 属主的组ID(GID) */
```

- **dev_t** **st_rdev;**
- */* 如果是一个设备文件则指出其所代表的设备号 */*
- **off_t** **st_size;** */* 以字节计算的文件长度 */*
- **unsigned long** **st_blksize;** */* 文件系统I/O的块尺寸 */*
- **unsigned long** **st_blocks;** */* 被定位块的数量 */*
- **time_t** **st_atime;** */* 最近一次访问的时间 */*
- **time_t** **st_mtime;** */* 最近一次修改的时间 */*
- **time_t** **st_ctime;** */* 最近一次状态改变的时间 */*
- **};**

stat取得文件的状态信息实例

- 系统调用头文件:

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

```
#include <unistd.h>
```

- book2.c作用:

- 取得文件的状态信息

- **#include <time.h>**
- **#include <sys/stat.h>**
- **#include <unistd.h>**
- **#include <stdio.h>**
- **#include <errno.h>**
- **extern int errno;**
- **int main(int argc, char* argv[])**
- **{**
- **struct stat buf;**
- **if (argc!=2)**
- **{**
- **printf("Usage: stat_exam filename\n");**
- **exit(0);**
- **}**

- if (**stat**(argv[1], &buf) == -1)
- {
- printf("stat error\nerrno is %d\n",errno);
- exit(1);
- }
- printf("dev_t is %d\n",buf.**st_dev**);
- printf("ino_t is %d\n",buf.**st_ino**);
- printf("mode is %o\n", buf.**st_mode**);
- printf("nlink_t is %d\n",buf.**st_nlink**);
- printf("uid_t is %d\n",buf.**st_uid**);
- printf("gid_t is %d\n",buf.**st_gid**);

- `printf("rdev is %d\n",buf.st_rdev);`
- `printf("size is %d\n",buf.st_size);`
- `printf("blksize is %d\n",buf.st_blksize);`
- `printf("blocks is %d\n",buf.st_blocks);`
- `printf("last access on %s",ctime(&buf.st_atime));`
- `printf("last modified on %s",ctime(&buf.st_mtime));`
- `printf("last change on %s",ctime(&buf.st_ctime));`
- `}`

结果:

命令: **./a.out test.c**

- **dev_t** is **770**
- **ino_t** is **137864**
- **mode** is **100644**
- **nlink_t** is **2**
- **... ..**
- **last access** **on ...**
- **last modified** **on ...**
- **last change** **on ...**

例：用 **lstat** 查看某设备是**字符**设备还是**块**设备。

- **#include <sys/types.h>**
- **#include <sys/stat.h>**
- **#include <dirent.h>**
- **#include <unistd.h>**
- **int main(int argc, char *argv[])**
- **{ int i;**
- **struct **stat** buf;**

- `for (i=1;i<=argc;i++);`
- `{ printf("%s:",argv[i]);`
- `if (lstat(argv[1],&buf)<0)`
- `{ printf("lstat error \n");`
- `exit(0);`
- `};`
- `printf("dev=%d\n",buf.st_dev);`

```
➤ if (S_ISCHR(buf.st_mode)||S_ISBLK(buf.st_mode))
➤     {         printf("(%s) rdev = %d",
(S_ISCHR(buf.st_mode))?"character":"block",
                                buf.st_rdev);
➤     }         printf("\n")
➤ }
➤ exit(0);
➤ }
```

结果:

- **./a.out /dev/hdg6**
- **/dev/hdg6 :dev=773**
- **(block) rdev = 8710**

- **./a.out /dev/ttyu1**
- **/dev/ttyu1 :dev=773**
- **(character) rdev = 849**

- **//用 \$ ls -l /dev/hdg6 或 \$ ls -l /dev/ttyu1**
- **可分别验证 b 或 c 开头的标记。**

取命令行参数，然后针对每一个命令行参数打印其文件类型

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

int
main(int argc, char *argv[])
{
    int          i;
    struct stat  buf;
    char         *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
    }
}
```

```

        if      (S_ISREG(buf.st_mode)) ptr = "regular";
    else if (S_ISDIR(buf.st_mode)) ptr = "directory";
    else if (S_ISCHR(buf.st_mode)) ptr = "character special";
    else if (S_ISBLK(buf.st_mode)) ptr = "block special";
    else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
#ifdef S_ISLNK
    else if (S_ISLNK(buf.st_mode)) ptr = "symbolic link";
#endif
#ifdef S_ISSOCK
    else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
#endif
    else
        ptr = "** unknown mode **";
    printf("%s\n", ptr);
}
exit(0);
}

```


输出:

```
$ a.out /vmunix /etc /dev/ttya /dev/sd0a /var/spool/cron/FIFO \  
> /bin /dev/printer  
/vmunix: regular  
/etc: directory  
/dev/ttya: character special  
/dev/sd0a: block special  
/var/spool/cron/FIFO: fifo  
/bin: symbolic link  
/dev/printer: socket
```

特地使用了`lstat`函数而非`stat`函数以便检测符号链接。如使用`stat`函数，则决不会观察到符号链接。

stat和fstat的使用及不同:

- **#include <sys/types.h>**
- **#include <sys/stat.h>**
- **#include <stdio.h>**
- **#include <fcntl.h>**
- **int main(int argc, char *argv[])**
- **{ int fd;**
- **struct **stat** buf;**

```
If(argc!=2){
```

```
    printf("usage:statfile filename\n");
```

```
    exit(1);
```

```
}
```

```
If((fd=open(argv[1],O_RDONLY))== -1){
```

```
    printf("cannot open %s\n",argv[1]);
```

```
    exit(1);
```

```
}
```

```
If(unlink(argv[1])== -1{
```

```
    printf("cannot unlink %s\n",argv[1]);
```

```
    exit(1);
```

```
}
```

```
if((stat(argv[1], &buf)==-1){  
    printf("stat %s fail!\n" , argv[1]);  
else  
    printf("stat %s succeed!\n" , argv[1]);  
if(fstat(fd , &buf)==-1)  
    printf("fstat %s fail!\n" , argv[1]);  
else  
    printf("fstat %s succeed!\n" , argv[1]);  
}
```

文件名在**unlink**之后，**stat**按原路径无法找到该文件，故无法获取该文件的信息；而文件描述符则因为文件仍打开而保留下来，**fstat**可获取该文件的信息。

8 access 系统调用

- 功能：系统调用**access**根据**mode**的值检查调用进程对文件**pathname**的许可权情况，即：是否具有**读、写或执行**的许可权。
- 声明格式：
- **int access(const char ***pathname**, int **mode**);**
- 在使用该系统调用的程序中要加入以下头文件：
- **#include <unistd.h>**

说明:

➤ 若**pathname**是符号链接，检查其所指的文件。

➤ 参数**mode**含义:

(**R_OK**可读; **W_OK**可写; **X_OK**可执行; **F_OK**是否存在)

04

02

01

00

用**access**检查用户文件的存取许可权

- **#include <sys/types.h>**
- **#include <fcntl.h>**
- **#include <unistd.h>**
- **int main(int argc, char *argv[])**
- **{ if (argc!=2)**
- **printf("usage:a.out <pathname>");**

- if (**access**(argv[1],**R_OK**)<0)
- printf("access error for %s\n",argv[1]);
- else printf("**read access OK!**\n");
- if (**open**(argv[1],**O_RDONLY**)<0)
- printf("open error for %s\n ",argv[1]);
- else printf("**open for reading OK**\n ");
- **exit**(0);
- }

结果:

➤ 程序中，**access**检查的结果 与 **open** 的方式应一致。

➤ **\$ ls -l a.out**

-rwxrwxr-x 1

➤ **\$./a.out a.out**

read access OK

open for reading OK

➤ **\$ ls -l /etc/uucp/Systems**

-rw-r----- 1 uucp 1441 Jul 18 15:05 /etc/uucp/Systems

➤ **\$ a.out /etc/uucp/Systems**

access error for /etc/uucp/Systems: Permission denied

open error for /etc/uucp/Systems: Permission denied