广嵌教育

Linux下的文件操作

www.gec-edu.org

目录

▲基本概念

▲文件

▲系统调用

▲库函数

▲系统命令

文件

- 文件：一组相关数据的有序集合

- Linux 中，一切（几乎一切）都是文件。

  Linux 环境中的文件具有特别重要的意义，因为它们为操作系统服务和设备提供了一个简单而统一的接口。

  设备对操作系统而言也可以看做是文件，通常程序完全可以像使用文件那样使用磁盘文件、串口、打印机和其他设备。

  目录也是一种文件，但它是一种特殊类型的文件。

  可以这么说Linux 中的任何事物都可以用一个文件代表，或者可以通过特殊的文件进行操作。当然，它们会与我们熟悉的传统文件有一些细微的区别，但两者的基本原则是一致的

- Linux常见的文件类型有5种:普通文件、目录文件、链接文件、设备文件和管道文件。
- Linux文件权限可分四种：可读取、可写入、可执行和无权限。分别用r、w、x和-表示。

系统调用

- 系统调用

  是指操作系统提供给用户程序调用的一组“特殊”接口，用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务。

  例如用户可以通过进程控制相关的系统调用来创建进程、实现进程调度、进程管理等。

- 为什么用户程序不能直接访问系统内核提供的服务呢?

  这是由于在Linux中，为了更好地保护内核空间，将程序的运行空间分为内核空间和用户空间（也就是常称的内核态和用户态），它们分别运行在不同的级别上，在逻辑上是相互隔离的。因此，用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。

- 但是，在有些情况下，用户空间的进程需要获得一定的系统服务（调用内核空间程序），这时操作系统就必须利用系统提供给用户的“特殊接口”——系统调用规定用户进程进入内核空间的具体位置。进行系统调用时，程序运行空间需要从用户空间进入内核空间，处理完后再返回到用户空间。

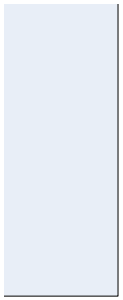
- Linux系统调用部分是非常精简的系统调用（只有250个左右）），它继承了UNIX系统调用中最基本和最有用的部分。这些系统调用按照功能逻辑大致可分为进程控制、进程间通信、文件系统控制、系统控制、存储管理、网络管理、socket控制、用户管理等几类。

▲库函数

- 前面讲到的系统调用并不是直接与程序员进行交互的,在实际使用中程序员使用的通常是库函数.
- 这些系统调用编程接口主要通过C库(libc)实现的.
- 并不是所有库函数都一一对应一个系统调用.有时,一个库存函数会需要几个系统调用来共同完成函数的功能;有一些库存函数不需要调用任何系统调用(因其不是完成内核提供的服务).

▲系统命令

- 系统命令相对库函数更高一层，它实际上是一个可执行程序，它的内部引用了库函数来实现相应的功能.比如：ls、cd等命令



关系图

文件I/O基本操作

Linux系统中文件操作主要有:

- 底层IO操作
- 标准IO操作

▲底层IO操作

底层文件I/O

① 不带缓冲

- 不带缓冲指的是每个文件操作动作都调用内核中的相应函数

② 通过文件描述符来访问文件

▲文件描述符

文件描述符是一个非负的整数，它是一个用于描述被打开文件的索引值。

当打开一个现存文件或创建一个新文件时，内核就向当前运行程序返回一个文件描述符；当需要读写文件时，也需要把文件描述符作为参数传递给相应的函数

如何区分和引用指定的文件呢?

这里用到了文件描述符。对于Linux而言，所有对文件(设备)的操作都是使用文件描述符来进行的。

注意点：

通常，**一个程序一开始运行时**，都会打开3个文件：

- 标准输入
- 标准输出
- 标准出错处理

这3个文件分别对应文件描述符为0、1和2

（也就是宏替换STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO）。

▲底层文件I/O常用函数

① open()
② close()
③ read()
④ write()
⑤ lseek()

open和close函数

- open()函数是用于打开或创建文件

  在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。

- close()函数是用于关闭一个被打开的文件。

  当一个进程终止时，所有被它打开的文件都由内核自动关闭，很多程序都使用这一功能而不显示地关闭一个文件。

所需头文件	#include <sys/types.h> /* 提供类型 pid_t 的定义 */ #include <sys/stat.h> #include <fcntl.h>	
函数原型	int open(const char *pathname, int flags, int perms)	
函数传入值	pathname	被打开的文件名（可包括路径名）
	flag：文件打开的方式	O_RDONLY：以只读方式打开文件
		O_WRONLY：以只写方式打开文件
		O_RDWR：以读写方式打开文件
		O_CREAT：如果该文件不存在，就创建一个新的文件，并用第三个参数为其设置权限
		O_EXCL：如果使用 O_CREAT 时文件存在，则可返回错误消息。这一参数可测试文件是否存在。此时 open 是原子操作，防止多个进程同时创建同一个文件。
		O_NOCTTY：使用本参数时，若文件为终端，那么该终端不会成为调用 open()的那个进程的控制终端
		O_TRUNC：若文件已经存在，那么会删除文件中的全部原有数据，并且设置文件大小为 0。
		O_APPEND：以添加方式打开文件，在打开文件的同时，文件指针指向文件的末尾，即将写入的数据添加到文件的末尾。
	perms	被打开文件的存取权限。 可以用一组宏定义：S_I(R/W/X)(USR/GRP/OTH) 其中 R/W/X 分别表示读/写/执行权限 USR/GRP/OTH 分别表示文件所有者/文件所属组/其他用户。 例如 S_IRUSR|S_IWUSR 表示设置文件所有者的可读可写属性。八进制表示法中 600 也表示同样的权限。

- 在open函数中，flag参数可通过"|"组合构成，但前3个函数不能相互组合.
- close函数

所需头文件	#include <unistd.h>
函数原型	int close(int fd)
函数输入值	fd：文件描述符
函数返回值	0：成功 -1：出错

open和close函数的使用实例

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

int main(void)
{
int fd;
            /*调用open函数，以可读写的方式打开，注意选项可以用“|”符号连接*/
if((fd = open("/tmp/hello.c", O_CREAT | O_TRUNC | O_WRONLY , 0600 ))<0){
    perror("open:");
    exit(1);
}
else{
    printf("Open file: hello.c %d\n",fd);
}
if( close(fd) < 0 ){
    perror("close:");
    exit(1);
}
else
    printf("Close hello.c\n");
exit(0);
}
```

read、write和lseek函数

- read()函数:是用于将从指定的文件读出数据
- write()函数:是用于向文件写数据
- lseek()函数:是用于文件指针定位到相应的位置

read函数

❖read()函数是用于将从指定的文件描述符中读出的数据放到缓存区中，并返回实际读入的字节数。若返回0，则表示没有数据可读，即已达到文件尾。读操作从文件的当前指针位置开始。当从终端设备文件中读出数据时，通常一次最多读一行。

所需头文件	`#include <unistd.h>`
函数原型	`ssize_t read(int fd, void *buf, size_t count)`
函数传入值	fd：文件描述符
	buf：指定存储器读出数据的缓冲区
	count：指定读出的字节数
函数返回值	成功：读到的字节数 0：已到达文件尾 -1：出错

在读普通文件时，若读到要求的字节之前已到达文件的尾部，则返回的字节数会小于希望读出的字节数。

- 例: 读取20字节的数据到缓冲区

```
#include <sys/types.h>
#include <unistd.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_read;
int fd;
...
nbytes = sizeof(buf);
bytes_read = read(fd, buf, nbytes);
...
```

➢本例中，调用read()后，需要检查返回的结果bytes_read，根据bytes_read进行相应的处理。

write()函数

- write()函数是用于向打开的文件写数据，写操作从文件的当前指针位置开始。对磁盘文件进行写操作，若磁盘已满则write()函数返回失败。

所需头文件	#include <unistd.h>
函数原型	ssize_t write(int fd, void *buf, size_t count)
函数传入值	fd：文件描述符
	buf：指定存储器写入数据的缓冲区
	count：指定读出的字节数
函数返回值	成功：已写的字节数 -1：出错

注意点:

- write()调用成功返回已写的字节数，失败返回-1。
- write()的返回值通常与count不同，因此需要循环将全部待写的数据全部写入文件。
- 对于普通文件，写操作从文件的当前位移量处开始，如果在打开文件时，指定了O_APPEND参数，则每次写操作前，将文件位移量设置在文件的当前结尾处，在一次成功的写操作后，该文件的位移量增加实际写的字节数。

❖ 例：将缓冲区的数据写入文件

```
#include <sys/types.h>
#include <string.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_written;
int fd;
...
strcpy(buf, "This is a test\n");
nbytes = strlen(buf);

bytes_written = write(fd, buf, nbytes);
...
```

➢本例中，调用write()后，需要检查返回的结果bytes_written，根据bytes_written进行相应的处理，如果bytes_written小于nbytes，则需要循环将未写的数据写入文件。

- lseek()函数
- lseek()函数是用于在指定的文件描述符中将文件指针定位到相应的位置。

  {它只能用在可定位（可随机访问）文件操作中。管道、套接字和大部分字符设备文件是不可定位的，所以在这些文件的操作中无法使用lseek()调用。}

所需头文件	#include <unistd.h> #include <sys/types.h>	
函数原型	off_t lseek(int fd, off_t offset, int whence)	
函数传入值	fd：文件描述符	
	offset：偏移量，每一读写操作所需要移动的距离，单位是字节，可正可负（向前移，向后移）	
	whence：当前位置的基点	SEEK_SET：当前位置为文件的开头，新位置为偏移量的大小
		SEEK_CUR：当前位置为文件指针的位置，新位置为当前位置加上偏移量
		SEEK_END：当前位置为文件的结尾，新位置为文件的大小加上偏移量的大小
函数返回值	成功：文件的当前位移 -1：出错	

- 例

把当前文件指针向后偏移20字节

(fd打开某文件返回的文件描述符)

```
lseek(fd,20,SEEK_CUR);
```

注意点:

- 每个打开的文件都有一个与其相关的“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。
- 通常，读/写操作都从当前文件位移量处开始，在读/写调用成功后，使位移量增加所读或者所写的字节数。
- lseek()调用成功返回为新的文件位移量，失败返回-1。
- lseek()只对常规文件有效，对socket、管道、FIFO等进行lseek()操作失败。
- lseek()仅将当前文件的位移量记录在内核中，它并不引起任何I/O操作。

- 练习

  创建一文件，接着在文件中写入”hello,I’m writing to this file!”,然后再把文件的前10个字节打印出来

▲标准IO操作

- 文件指针
  - FILE指针：每个被使用的文件都在内存中开辟一个区域，用来存放文件的有关信息，这些信息是保存在一个结构体类型的变量中，该结构体类型是由系统定义的，取名为FILE。
  - 标准I/O库是由Dennis Ritchie在1975年左右编写的

FILE在头文件/usr/include/libio.h中定义

```
struct _IO_FILE {
  int _flags;           /* High-order word is
_IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags
  /* The following pointers correspond to the C++
streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and
_IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get
area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support
backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-
current get area. */
  char *_IO_backup_base;  /* Pointer to
first valid
                                    character of
backup area */
  char *_IO_save_end; /* Pointer to
end of
non-current get area. */
  struct _IO_marker *_markers;
  struct _IO_FILE *_chain;
  int _fileno;
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used
to be
_offset but it's too small.  */
#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is
unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];
  /*  char* _save_gptr;  char*
_save_egptr; */
  _IO_lock_t *_lock;
};
```

- 标准I/O提供了三种类型的缓存
  - 全缓存
    - 当填满I/O缓存后才进行实际I/O操作
  - 行缓存
    - 当在输入和输出中遇到新行符('\n')时，进行I/O操作。

    当流遇到一个终端时，典型的行缓存。
  - 不带缓存
    - 标准I/O库不对字符进行缓冲，例如stderr。

- 标准I/O预定义3个流，他们可以自动地为进程所使用

标准输入	0	STDIN_FILENO	stdin
标准输出	1	STDOUT_FILENO	stdout
标准错误输出	2	STDERR_FILENO	stderr

标准IO操作函数

函数	作　用
fopen	打开或创建文件
fclose	关闭文件
fread	由文件中读取一个字符
fwrite	将数据成块写入文件流
fseek	移动文件流的读写位置

- 打开文件:

  有二个标准函数，分别为：fopen()、fdopen()。

  它们可以以不同的模式打开，但都返回一个指向FILE的指针，。此后，对文件的读写都是通过这个FILE指针来进行。

  其中fopen()可以指定打开文件的路径和模式，fdopen()可以指定打开的文件描述符和模式.

所需头文件	#include <stdio.h>
函数原型	`FILE *fopen(const char * path,const char * mode)`
函数传入值	path：包含要打开的文件路径及文件名
	mode：文件打开状态（后面会具体说明）
函数返回值	成功：指向 FILE 的指针 失败：NULL

所需头文件	#include <stdio.h>
函数原型	FILE * fdopen(int fd,const char * mode)
函数传入值	fd：要打开的文件描述符
	mode：文件打开状态（后面会具体说明）
函数返回值	成功：指向 FILE 的指针 失败：NULL

❖ 其中，mode定义打开文件的访问权限等:

r 或 rb	打开只读文件，该文件必须存在
r＋或 r＋b	打开可读写的文件，该文件必须存在
W 或 wb	打开只写文件，若文件存在则文件长度清为 0，即会擦写文件以前的内容。若文件不存在则建立该文件
w+或 w＋b	打开可读写文件，若文件存在则文件长度清为 0，即会擦写文件以前的内容。若文件不存在则建立该文件
a 或 ab	以附加的方式打开只写文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留
a+或 a＋b	以附加方式打开可读写的文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留

当给定“b”参数时，表示以二进制方式打开文件

打开一个标准I/O流的六种不同方式:

打开一个标准I/O流的六种不同的方式

限　制	r	w	a	r+	w+	a+
文件必须已存在	•			•		
擦除文件以前的内容		•			•	
流可以读	•			•	•	•
流可以写		•	•	•	•	•
流只可在尾端处写			•			•

- 关闭文件:

  关闭标准流文件的函数为fclose()，该函数将缓冲区内的数据全部写入到文件中，并释放系统所提供的文件资源.

所需头文件	#include <stdio.h>
函数原型	int fclose(FILE * stream)
函数传入值	stream：已打开的文件指针
函数返回值	成功：0 失败：EOF

•在调用fclose()关闭流后对流所进行的任何操作，包括再次调用fclose()，其结果都将是未知的。

- 例1:
  - 以读写方式打开文件file_3，如果该文件不存在，则创建。如果该文件已经存在，则长度截短为0。

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(int argc, char **argv)
{
    FILE *fp;

    if((fp = fopen("test_file_3", "w+")) == NULL)
    {
        fprintf(stderr, "fopen() failed: %s\n", strerror(errno));
        return -1;
    }

    fclose(fp);

    return 0;
}
```

调用fopen()成功打开流之后，可在三种不同类型的I/O中进行选择，对其进行读、写操作:

- **每次一定数量I/O。**fread()和fwrite()函数支持这种类型的I/O。每次I/O操作读或写某种数量的对象，而每个对象具有指定的长度。
- **每次一个字符的I/O。**使用fgetc()/fputc()一次读或写一个字符，如果流是带缓存的，则标准I/O函数处理所有缓存。
- **每次一行的I/O。**使用fgets()和fputs()一次读或写一行。每行都以一个新行符终止。当调用fgets()时，应说明能处理的最大行长。

❖ 读操作的函数为fread()：

所需头文件	#include <stdio.h>
函数原型	size_t fread(void * ptr,size_t size,size_t nmemb,FILE * stream)
函数传入值	ptr：存放读入记录的缓冲区
	size：读取的记录大小
	nmemb：读取的记录数
	stream：要读取的文件流
函数返回值	成功：返回实际读取到的 nmemb 数目 失败：EOF

- fwrite()函数是用于对指定的文件流进行写操作。

所需头文件	#include <stdio.h>
函数原型	size_t fwrite(const void * ptr,size_t size,size_t nmemb,FILE * stream)
函数传入值	ptr：存放写入记录的缓冲区
	size：写入的记录大小
	nmemb：写入的记录数
	stream：要写入的文件流
函数返回值	成功：返回实际写入到的 nmemb 数目 失败：EOF

- 实例

```
                /*  fwrite.c  */
#include <stdio.h>
int main()
{
 FILE *stream;
 char s[3]={'a','b','c'};
                /*首先使用fopen打开文件，之后再调用fwrite写入文件*/
 stream=fopen("what","w");
 i=fwrite(s,sizeof(char),nmemb,stream);
    printf("i=%d",i);
    fclose(stream);
}
```

运行结果如下所示：

```
[root@localhost file]#  ./write
i=3
[root@localhost file]# cat what
abc
```

例2: 读或写一个结构体，将一个结构体写至一个文件上

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#define NAMESIZE 16

struct
{
    short count;
    long total;
    char name[NAMESIZE];
}item;

int main(int argc, char **argv)
{
    FILE *fp;
    if((fp=fopen("test_file_5", "wb+")) == NULL){
        fprintf(stderr, "fopen file test_file_5 for binary write failed: %s\n", \
                strerror(errno));
        return 0;
    }
    memset(&item, 0, size(item));
    item.count = 9;
    item.total = 74;

    strncpy(item.name, "Richard Stallman", NAMESIZE);

    if(fwrite(&item, sizeof(item), 1, fp) != 1){
        fprintf(stderr, "Binary write data to file test_file_5 failed: %s\n", \
                strerror(errno));
        return 0;
    }

    fclose(fp);

    return 0;
}
```

育

fseek()/ftell()/rewind()函数原型：

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
void rewind(FILE *stream);
```

fseek()：用户设定stream流的文件位置指示，调用成功返回 0，失败返回-1(fseek()的whence参数：SEEK_SET/ SEEK_CUR/ SEEK_END)

ftell()：用于取得当前的文件位置，调用成功则返回当前文件读写位置，若出错则为-1

rewind()：用于设定流的文件位置指示为文件开始，该函数调用成功无返回值。rewind()等价于fseek(stream, 0, SEEK_SET)

例子

- 使用标准IO操作把已存在的文件imrc内容复制到当前文件test中

字符输入函数

所需头文件	#include <stdio.h>
函数原型	int getc(FILE * stream) int fgetc(FILE * stream) int getchar(void)
函数传入值	stream：要输入的文件流
函数返回值	成功：**返回读取的字符** 失败：EOF

☆三个函数的返回：若成功则为读取的字符，若已处文件尾端或出错则为EOF

☆函数getchar()等同于getc(stdin)

☆注意，不管是出错还是到达文件尾端，这三个函数都返回同样的值。为了区分　　这两种不同的情况，必须调用ferror()或feof()。

☆返回值为int类型。

字符输出函数

所需头文件	#include <stdio.h>
函数原型	int putc(int c,FILE * stream) int fputc(int c,FILE * stream) int putchar(int c)
函数返回值	成功：字符 c 失败：EOF

☆putchar(c)等价于putc(c,stdout)。
☆出错返回EOF。

下面这个实例结合fputc和fgetc，将标准输入复制到标准输出中去。

```
                                /*  fput.c  */
        #include<stdio.h>
        main()
        {
         int c;
                        /*把fgetc的结果作为fputc的输入*/
         fputc(fgetc(stdin),stdout);
        }
```

运行结果如下所示：

```
[root@localhost file]# ./file
w（用户输入）
w（屏幕输出）
```

检查文件出错函数:

```
#include <stdio.h>

int feof(FILE *stream);
int ferror(FILE *stream);
int clearerr(FILE *stream);
```

- 在大多数的FILE对象的实现中，保留两个标志:
- 出错标志。
- 文件结束标志。

行输入函数

所需头文件	#include <stdio.h>
函数原型	char * gets(char *s) char * fgets(char * s, int size, FILE * stream)
函数传入值	s：要输入的字符串 size：输入的字符串长度 stream：对应的文件流
函数返回值	成功：s 失败：NULL

- 两个函数返回：若成功则为buf，若已处文件尾端或出错则为null
- 这两个函数都指定了缓存地址，读入的行将送入其中。gets()从标准输入读，而fgets()则从指定的流读。

- 对于fgets()函数，必须指定缓存的长度n。此函数一直读到换行符为止，但是不超过n-1个字符，读入的字符被送入缓存。该缓存以空字符\0结尾。
- gets()是一个不推荐使用的函数，因为调用者在使用gets()时不能指定缓存的长度，这样就可能造成缓存越界（如若该行长于缓存长度），写到缓存之后的存储空间中，从而产生不可预料的后果。
- gets()与fgets()的另一个区别是，gets()并不将换行符存入缓存中。gets()函数用来从标准输入设备（键盘）读取字符串直到换行符结束，但换行符会被丢弃，然后在末尾添加'\0'字符。

行输出函数

所需头文件	#include <stdio.h>
函数原型	int puts(const char *s) int fputs(const char * s, FILE * stream)
函数传入值	s：要输出的字符串 stream：对应的文件流
函数返回值	成功：**非负整数** 失败：**EOF**

- 两个函数返回：若成功则为非负值，若出错则为EOF
- 函数fputs()将一个以空字符\0终止的字符串写到指定的流，终止符不写出。注意，这并不一定是每次输出一行，因为它并不要求在空字符之前一定是换行符。通常，在空字符之前是一个换行符，但并不要求总是如此。
- puts()将一个以空符终止的字符串写到标准输出，终止符不写出。但是，puts()然后又将一个换行符写到标准输出。
- puts()并不像它所对应的gets()那样不安全。但是我们还是应避免使用它，以免需要记住它在最后又加上了一个换行符。如果总是使用fgets()和fputs(),那么就会熟知在每行终止处我们必须自己加一个换行符。

例：循环从标准输入(stdin)逐行读入数据，并逐行字符显示到标准输出，每次读取的最大长度为20字节。

```c
#include <stdio.h>

#define MAXLINE 20

int main(void)
{
    char line[MAXLINE];
    while(fgets(line, MAXLINE, stdin) != NULL && line[0] != '\n'){
        fputs("result: ", stdout);
        fputs(line, stdout);
    }

    return 0;
}
```

```
seton@ubuntu:~$ ./a
01234567890123456789012345
result: 01234567890123456result: 9012345
```

I/O模型比较

I/O模型	文件I/O	标准I/O
缓冲方式	非缓冲I/O	缓冲I/O
操作对象	文件描述符	流(FILE *)
打开	open()	fopen()/fdopen()
读	read()	fread()/fgetc()/fgets()…
写	write()	fwrite()/fputc()/fputs()…
定位	lseek()	fseek()/ftell()/rewind()
关闭	close()	fclose()

▲文件和目录的维护

1. chmod

- chmod系统调用改变文件或目录的访问权限。

- 函数原型:
- #include<sys/stat.h>
- int chmod(const char *path,mode_t mode)
- path参数指定的文件被修改为具有mode参数给出的访问权限。

2.mkdir和rmdir

- mkdir和rmdir函数用来建立和删除目录

- 函数原型
- #include<sys/types.h>
- #include<sys/stat.h>
- Int mkdir(const char *path,mode_t mode);

- #include<unistd.h>
- Int rmdir(const char *path)
-