# Linux系统编程之文件与IO

## [linux系统编程之文件与IO（一）：文件描述符、open，close](http://www.cnblogs.com/mickole/p/3181086.html)

1. 什么是IO？

* 输入/输出是主存和外部设备之间拷贝数据的过程

          设备->内存（输入操作）

           内存->设备（输出操作）

* 高级I/O

          ANSI C提供的标准I/O库称为高级I/O，通常也称为带缓冲的I/O

* 低级I/O

          通常也称为不带缓冲的I/O

2. 文件描述符：fd

* 对于Linux而言，所有对设备或文件的操作都是通过文件描述符进行的。
* 当打开或者创建一个文件的时候，内核向进程返回一个文件描述符（非负整数）。后续对文件的操作只需通过该文件描述符，内核记录有关这个打开文件的信息。
* 一个进程启动时，默认打开了3个文件，标准输入、标准输出、标准错误，对应文件描述符是0（STDIN\_FILENO）、1（STDOUT\_FILENO）、2（STDERR\_FILENO）,这些常量定义在unistd.h头文件中。C库函数中与之对应的是：stdin,stdout,stderr,不过这三个是FILE指针类型。

3.文件描述符与文件指针相互转换

可以通过以下两个函数实现：

* fileno：将文件指针转换为文件描述符

            #include <stdio.h>

            int fileno(FILE \*stream)

测试程序：

[复制代码](javascript:void(0);)

#include <stdlib.h>

#include <stdio.h>

int main(void)

{

printf("fileno(stdin) = %d\n", fileno(stdin));

printf("fileno(stdout) = %d\n", fileno(stdout));

printf("fileno(stderr) = %d\n", fileno(stderr));

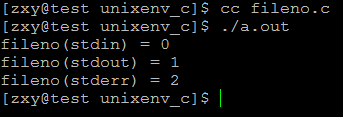
return 0;

}

测试结果：

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

[](http://images.cnitblog.com/blog/529981/201307/10123416-0917bc2bbc4b4bf9887af6a24c77fc5b.png)

[复制代码](javascript:void(0);)

* fdopen：将文件描述符转换为文件指针

#include <stdio.h>

        FILE \*fdopen(int fd, const char \*mode)    //mode :r,w,r+,w+,a,a+

4.文件系统调用

* 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。最常用的是使用open（）（打开）系统调用

函数原型

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

int open(const char \*pathname, int flags);   
int open(const char \*pathname, int flags, mode\_t mode);

参数

path ：文件的名称，可以包含（绝对和相对）路径

flags：文件打开模式

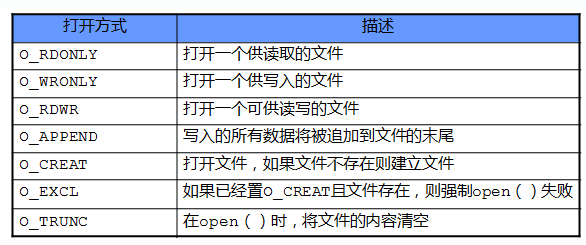
mode：用来规定对该文件的所有者，文件的用户组及系 统中其他用户的访问权限

返回值

打开成功，返回文件描述符；

打开失败，返回－1

文件打开方式：

[](http://images.cnitblog.com/blog/529981/201307/09230841-09a82e0122cf4de993c3e017cc8c34c5.png)

O\_EXCL表示：当O\_EXCL|O\_CREAT时，若文件存在，则打开失败，不存在，则打开成功

访问权限：

[](http://images.cnitblog.com/blog/529981/201307/09230842-52df2d181fab4ef4b06a43119336e8ea.png)

open系统调用的几点说明：

可以利用按位逻辑加(bitwise-OR)(|)对打开方式的标志值进行组合。

如打开一个新文件：

＃define NEWFILE  (O\_WRONLY|O\_CREAT|O\_TRUNC)

对访问权限位进行访问所用到的标识符，均可以通过

＃include <sys/stat.h> 访问到,同样可以通过|运算来对访问权限进行组合也可以直接给出数字表示如0655

#define MODE755 (S\_IRWXU|S\_IRGRP|S\_IXGRP|S\_IROTH|S\_IXOTH)

注：文件的访问权限是根据：umask&~mode得出来的，例如umask=0022,mode = 0655 则访问权限为：644

测试程序：

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

int main(void)

{

umask(0);

int fd;

fd = open("test.txt", O\_WRONLY | O\_CREAT, 0666);

if (fd == -1)

ERR\_EXIT("open error");

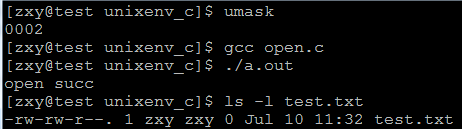
printf("open succ\n");

return 0;

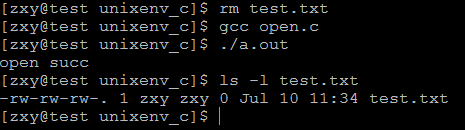
}

[复制代码](javascript:void(0);)

测试结果一：采用默认的umask值

[](http://images.cnitblog.com/blog/529981/201307/10123417-65406f9e0765415ea04440403034f510.png)

测试结果二：重新设置umask值

[](http://images.cnitblog.com/blog/529981/201307/10123417-1dc54cb688824267af24e6ca556f35d5.png)

* + 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

函数原型：

#include <unistd.h>

int close(int fd);

函数参数：

-fd ：要关闭的文件的文件描述符

返回值

如果出现错误，返回-1

调用成功返回0

注：若没有显示调用close（），当程序退出时也会关闭文件

* + 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()。现代的linux内核很少采用creat创建文件，因为open可以完成创建功能

函数原型：

int creat(const char \*path, mode\_t mode)；

参数

path ：文件的名称，可以包含（绝对和相对）路径

mode: 用来规定对该文件的所有者，文件的用户组及系 统中其他用户的访问权限

返回值

打开成功，返回文件描述符；

打开失败，返回－1

       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。

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)；

## [linux系统编程之文件与IO（二）：系统调用read和write](http://www.cnblogs.com/mickole/p/3181146.html)

* read系统调用

一旦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，就可以用read()系统调用从该文件中读取字节

函数原型：

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

参数

fd ：想要读的文件的文件描述符

buf ： 指向内存块的指针，从文件中读取来的字节放到这个内存块中

count ： 从该文件复制到buf中的字节个数

返回值

如果出现错误，返回-1

读文件结束，返回0

否则返回从该文件复制到规定的缓冲区中的字节数

否则返回从该文件复制到规定的缓冲区中的字节数

* write系统调用

用write()系统调用将数据写到一个文件中

函数原型：

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

函数参数：

-fd：要写入的文件的文件描述符

-buf：指向内存块的指针，从这个内存块中读取数据写入 到文件中

-count：要写入文件的字节个数

返回值

如果出现错误，返回-1

注:write并非真正写入磁盘,而是先写入内存缓冲区,待缓冲区满或进行刷新操作后才真正写入磁盘,若想实时写入磁盘可调用

int fsync(int fd);或在open时flags加上O\_SYNC

* 利用read和write进行文件拷贝

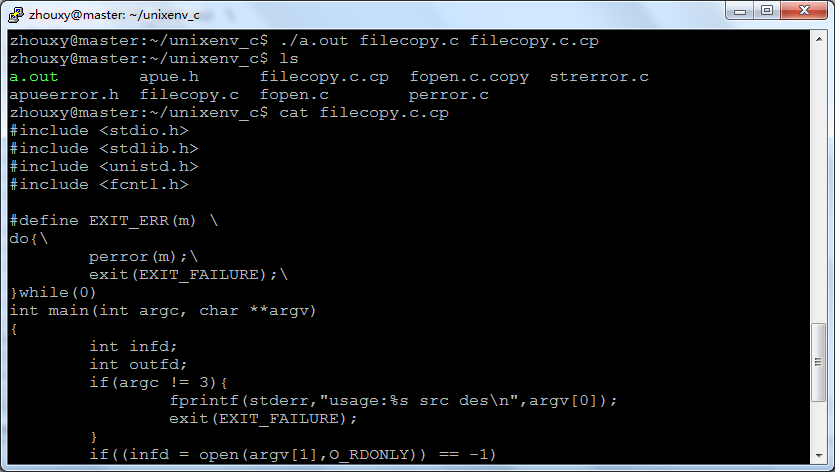
程序代码:

* [复制代码](javascript:void(0);)
* #include <stdio.h>
* #include <stdlib.h>
* #include <unistd.h>
* #include <fcntl.h>
* #define EXIT\_ERR(m) \
* do{\
* perror(m);\
* exit(EXIT\_FAILURE);\
* }while(0)
* int main(int argc, char \*\*argv)
* {
* int infd;
* int outfd;
* if(argc != 3){
* fprintf(stderr,"usage:%s src des\n",argv[0]);
* exit(EXIT\_FAILURE);
* }
* if((infd = open(argv[1],O\_RDONLY)) == -1)
* EXIT\_ERR("open error");
* if((outfd = open(argv[2],O\_WRONLY|O\_CREAT|O\_TRUNC,0644)) == -1)
* EXIT\_ERR("OPEN ERROR");
* char buf[1024];
* int n;
* while((n = read(infd, buf, 1024)) > 0 ){
* write(outfd, buf, n);
* }
* close(infd);
* close(outfd);
* return 0;

}

[复制代码](javascript:void(0);)

测试结果:

[](http://images.cnitblog.com/blog/529981/201307/10000342-205bf1dd21fd4519af091718c599e9bf.png)

## [linux系统编程之文件与IO（三）：利用lseek()创建空洞文件](http://www.cnblogs.com/mickole/p/3182033.html)

一、lseek（）系统调用

功能说明：

通过指定相对于开始位置、当前位置或末尾位置的字节数来重定位 curp，这取决于 lseek() 函数中指定的位置

函数原型：

#include <sys/types.h>   
#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence);

参数说明：

fd：文件描述符

offset：偏移量，该值可正可负，负值为向前移

whence：搜索的起始位置，有三个选项：

(1).SEEK\_SET: 当前位置为文件的开头，新位置为偏移量大小   
(2).SEEK\_CUR: 当前位置为文件指针位置，新位置为当前位置加上偏移量大小   
(3).SEEK\_END: 当前位置为文件结尾，新位置为偏移量大小

返回值：文件新的偏移值

二、利用lseek（）产生空洞文件（hole）

说明：

The lseek() function allows the file offset to be set beyond the end of the file (but this does not change the size of the file).  If  data  is later written at this point, subsequent  reads of the data in the gap (a "hole") return null bytes ('\0') until data is  actually  written  into the gap.

程序代码：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

int main(void)

{

int fd;

int ret;

fd = open("hole.txt",O\_WRONLY|O\_CREAT|O\_TRUNC,0644);

if(fd == -1)

ERR\_EXIT("open error");

write(fd,"hello",5);

ret = lseek(fd,1024\*1024\*1024,SEEK\_CUR);

if(ret == -1)

ERR\_EXIT("lseek error");

write(fd,"world",5);

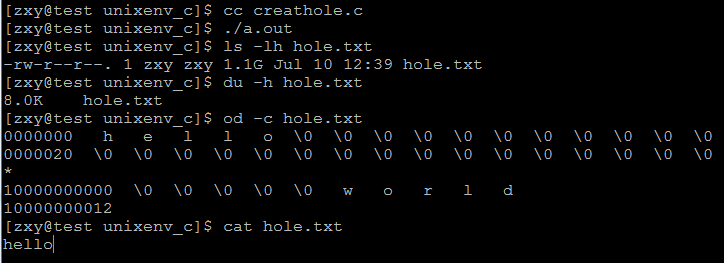
close(fd);

return 0;

}

[复制代码](javascript:void(0);)

测试结果：

[](http://images.cnitblog.com/blog/529981/201307/10134536-9dbf7fdd156e4c47a3592797a5a9485e.png)

程序创建一个hole文件，然后写入”hello”字符，在利用lseek系统调用从当前位置偏移到1024\*1024\*1024之后再写入”world”字符，ls显示文件大小为1.1G，实际上它并没有占用这么多的磁盘空间，du表明hole文件只有8k，系统只是存储一些信息，用它来表示有多少个\0，当我们用cat查看文件内容时只看到hello，由于文件太大最后的world太后以致看不到，但当我们用od命令查看时，可以发现有好多个\0。

## [linux系统编程之文件与IO（四）：目录访问相关系统调用](http://www.cnblogs.com/mickole/p/3182205.html)

1. 目录操作相关的系统调用   
    1.1 mkdir和rmdir系统调用   
    1.1.1 实例   
    1.2 chdir, getcwd系统调用   
    1.2.1 实例   
    1.3 opendir, closedir, readdir,   
    1.3.1 实例:递归便利目录

### 1. 目录操作相关的系统调用

#### 1.1 mkdir和rmdir系统调用

[code]   
filename: mk\_rm\_dir.c   
#include <sys/stat.h>   
int mkdir(const char \*path, mode\_t mode);   
return:   
    S    0   
    F    -1   
note:   
    mode权限至少要有执行权限。   
[/code]   
[code]   
#include <unistd.h>   
int rmdir(const char \*pathname);   
return:   
    S    0   
    F    -1   
note:   
    pathname目录必须是空目录。

##### 1.1.1 实例

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <stdio.h>

#include <assert.h>

#define MODE (S\_IRUSR | S\_IWUSR | S\_IXUSR | S\_IXGRP | S\_IXOTH)

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

{

char \*pname;

assert(argc == 2);

pname = argv[1];

assert(mkdir(pname, MODE) == 0);

printf("create %s successful!\n", pname);

assert(rmdir(pname) == 0);

printf("rm %s\n", pname);

return 0;

}

[复制代码](javascript:void(0);)

    测试：   
  
[qtlldr@qtldr editing]./mkrmdirtestdircreatetestdirsuccessful!rmtestdir[qtlldr@qtldrediting]./mkrmdirtestdircreatetestdirsuccessful!rmtestdir[qtlldr@qtldrediting]  

#### 1.2 chdir, getcwd系统调用

#include <unistd.h>   
int chdir(const char \*pathname);   
return:   
    S    0   
    F    -1   
  
#include <unistd.h>   
char \*getpwd(char \*buf, size\_t size);   
return:   
    S    buf   
    F    NULL   
  
    buf是缓冲地址，size是buf的长度。该缓冲必须有足够的长度以容纳绝对路径名加上一个null终止符。

##### 1.2.1 实例

[code]   
filename:ch\_get\_dir.c

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <stdio.h>

#include <string.h>

#include <assert.h>

#define BUFSIZE (50)

int main(void)

{

char buf[BUFSIZE];

memset((void \*)buf, '\0', sizeof buf);

assert(chdir("/tmp") == 0);

printf("chdir to /tmp successful\n");

assert(getcwd(buf, BUFSIZE) != NULL);

printf("now the directory is %s\n", buf);

return 0;

}

[复制代码](javascript:void(0);)

    测试：   
  
[qtlldr@qtldr editing]./chgetdirchdirto/tmpsuccessfulnowthedirectoryis/tmp[qtlldr@qtldrediting]./chgetdirchdirto/tmpsuccessfulnowthedirectoryis/tmp[qtlldr@qtldrediting]

### 1.3 opendir, closedir, readdir,

#include <sys/type.s>   
#include <dirent.h>   
DIR \*opendir(const char \*dirname);   
return:   
    S    DIR指针   
    F    NULL   
note:   
    DIR是一种目录结构，类似FILE。   
  
#include <sys/types.h>   
#include <dirent.h>   
struct dirent \*readir(DIR \*dirp);   
return:   
    S    一个指向保存目录流下一个目录项的dirent指针   
    F    NULL   
note:   
    struct dirent {   
        char    d\_name[NAME + 1]; /\* \0结尾的文件名 \*/   
    }   
    到达目录尾或出错返回NULL，但是到达目录尾不会设置errno，出错则设置。   
    如果在readir的同时有其他进程在目录中创建或者删除文件爱你，readdir不保证能列处该目录中所有文件。

#include <sys/types.h>   
#include <dirent.h>   
int closedir(DIR \*dirp);   
return:   
    S    0   
    F    -1

#### 1.3.1 实例:递归便利目录

filename:help.txt 帮助文档   
            本程序只为学习linux目录操作而写   
printdir   
    输出目录文件或者统计目录中的文件数目   
语法：   
    printdir [option] <files...>   
选项：   
    -l   
        输出目录下的文件名   
    -c   
        统计目录下的文件   
    -d n   
        指定最大层次，最大为30   
默认行为：   
    如果没有指定选项，那么只输出该目录下的文件名   
BUG:   
    -l与 -c选项不能同时使用，如果同时使用统计出错。(以后会修正）   
  
            本程序只为学习linux目录操作而写   
  
filename:printdir.c

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <dirent.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define INDENT\_DEPTH (4) /\* 列举文件时的缩进数 \*/

#define DEPTH\_MAX (30) /\* 递归便利的最大层次 \*/

#define HELPFILE ("help.txt")

typedef int count\_t;

struct nfiletype {

count\_t ndir;

count\_t nreg;

count\_t nchr;

count\_t nfifo;

count\_t nsock;

count\_t nchar;

count\_t nblock;

count\_t nlink;

count\_t ntotol;

count\_t nunknow;

};/\*记录各个类型文件的数目\*/

int DEPTH = 20; /\* 递归层级限制 \*/

int idepth\_count = 1;

int idepth\_print = 1;

static struct nfiletype \*count\_files(const char \*pathname,

struct nfiletype \*nfile);

static void printdir(const char \*pathname, int indent);

int main(int argc, char \*\*argv)

{

int opt;

int depth\_opt;

int count\_flag = 0;

int print\_flag = 0;

char \*parg = NULL;

struct nfiletype nfiles = {0};

int fd\_help;

char buf\_help[BUFSIZ];

int nread\_help;

char \*filename\_help = HELPFILE;

while ((opt = getopt(argc, argv, "lhd:c")) != -1) {

switch (opt) {

case 'l':

print\_flag = 1;

break;

case 'c':

count\_flag = 1;

break;

case 'd':

depth\_opt = strtol(optarg, NULL, 10);

DEPTH = depth\_opt <= DEPTH\_MAX ? depth\_opt: DEPTH;

break;

case ':':

printf("option needs a value\n");

break;

case '?':

printf("unknown option :%c\n", optopt);

break;

case 'h':

fd\_help = open(filename\_help, O\_RDONLY);

if (fd\_help != -1) {

while ((nread\_help = read(fd\_help, buf\_help, BUFSIZ)) > 0) {

write(1, buf\_help, nread\_help);

}

close(fd\_help);

} else {

fprintf(stderr, "open %s failed!\n", filename\_help);

}

return 0;

}

}

/\* 如果没有选项，那么默认是打印目录 \*/

if (!print\_flag && !count\_flag)

print\_flag = 1;

for( ; optind < argc; optind++) {

parg = argv[optind];

if (print\_flag) {

//printf("DEBUG-- printdir --%s\n", parg);

printdir(parg, 4);

}

if (count\_flag) {

memset((void \*)&nfiles, '\0', sizeof nfiles);

//printf("DEBUG-- count\_files--%s\n", parg);

count\_files(parg, &nfiles);

printf("In the %s there are :\n", parg);

printf(" directory %d\n", nfiles.ndir);

printf(" regular file %d\n", nfiles.nreg);

printf(" specal character file %d\n", nfiles.nchr);

printf(" special block file %d\n", nfiles.nblock);

printf(" fifo file %d\n", nfiles.nfifo);

printf(" sock file %d\n", nfiles.nsock);

printf(" link file %d\n", nfiles.nlink);

printf(" unknown file %d\n", nfiles.nunknow);

printf("Total %d\n", nfiles.ntotol);

}

}

return 0;

}

/\*

\*function: 对该目录下的文件类型进行统计

\* input arg:

\* pathname:目录名指针

\* nfile:记录文件类型数目的结构体指针

\* return：

\* 记录文件类型数目的结构体指针

\*/

static struct nfiletype \*count\_files(const char \*pathname,

struct nfiletype \*nfile)

{

DIR \*dp;

struct dirent \*entry;

struct stat statbuf;

//printf("DEBUG-- in count\_files -- %s\n", pathname);

/\* 层次控制 \*/

if (idepth\_count > DEPTH)

return NULL;

idepth\_count++;

if ((dp = opendir(pathname)) == NULL) {

fprintf(stderr, "can not open %s\n", pathname);

return NULL;

}

chdir(pathname);

while ((entry = readdir(dp)) != NULL) {

/\* 跳过 . 和 .. \*/

if (strcmp(entry->d\_name, ".") == 0 || strcmp(entry->d\_name, "..") == 0)

continue;

/\* 取得文件信息 \*/

if (lstat(entry->d\_name, &statbuf) == -1) {

fprintf(stderr, "can not test the %s's type\n", entry->d\_name);

return NULL;

}

/\* 统计文件数目 \*/

if (S\_ISDIR(statbuf.st\_mode)) { /\* 是目录就递归吧 \*/

//printf("DEBUG -- directory %s\n", entry->d\_name);

count\_files(entry->d\_name, nfile);

nfile->ndir++;

}

else if (S\_ISREG(statbuf.st\_mode)) {

//printf("DEBUG -- regular file %s\n", entry->d\_name);

nfile->nreg++;

}

else if (S\_ISCHR(statbuf.st\_mode))

nfile->nchr++;

else if (S\_ISBLK(statbuf.st\_mode))

nfile->nblock++;

else if (S\_ISLNK(statbuf.st\_mode))

nfile->nlink++;

else if (S\_ISFIFO(statbuf.st\_mode))

nfile->nfifo++;

else if (S\_ISSOCK(statbuf.st\_mode))

nfile->nsock++;

else nfile->nunknow++;

nfile->ntotol++;

}

chdir("..");

closedir(dp);

return nfile;

}

/\*

nblock; \*function:列出目录中的文件

nlink; \*input arg:

ntotol; \* pathname: 目录名

\*return:

\* void

\*/

static void printdir(const char \*pathname, int indent)

{

DIR \*dp;

struct dirent \*entry;

struct stat statbuf;

/\* 层次控制 \*/

if (idepth\_print > DEPTH)

return ;

idepth\_print++;

if ((dp = opendir(pathname)) == NULL) {

fprintf(stderr, "can not open %s\n", pathname);

return ;

}

chdir(pathname);

while ((entry = readdir(dp)) != NULL) {

/\* 跳过 . 和 .. \*/

if (strcmp(entry->d\_name, ".") == 0 || strcmp(entry->d\_name, "..") == 0)

continue;

if (lstat(entry->d\_name, &statbuf) == -1) {

fprintf(stderr, "can not test the %s's type\n", entry->d\_name);

return ;

}

if (S\_ISDIR(statbuf.st\_mode)) { /\* 是目录就递归吧 \*/

printf("%\*s%s/\n", indent," ", entry->d\_name);

printdir(entry->d\_name, indent + INDENT\_DEPTH);

}

else {

printf("%\*s%s\n", indent," ", entry->d\_name);

}

}

chdir("..");

closedir(dp);

}

[复制代码](javascript:void(0);)

## [linux系统编程之文件与IO（八）：文件描述符相关操作-dup,dup2,fcntl](http://www.cnblogs.com/mickole/p/3184254.html)

本节目标：

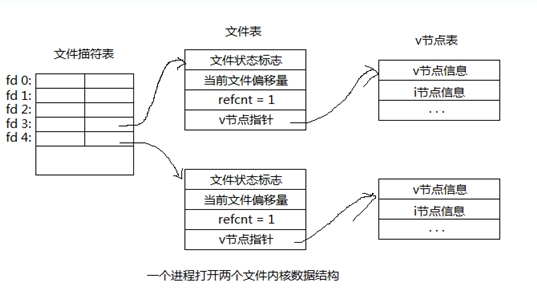
##### 1,文件共享

* 打开文件内核数据结构
* 一个进程两次打开同一个文件
* 两个进程打开同一个文件

##### 2,复制文件描述符（dup、dup2、fcntl）

一，文件共享

1，一个进程打开两个文件内核数据结构

[](http://images.cnitblog.com/blog/529981/201307/11163923-155e1ffab087400b849c91dd910f449f.png)

说明：

文件描述符表：每个进程都有一张，彼此独立，每个文件描述符表项都指向一个文件表，文件描述符0（STDIN\_FILENO）、1（STDOUT\_FILENO）、2（STDERR\_FILENO）,默认已经打开，分别表示：标准输入，标准输出，标准错误设备。

文件表：每打开一个文件就对应一张文件表，文件表可以共享，当多个文件描述符指向同一个文件表时，文件表中的

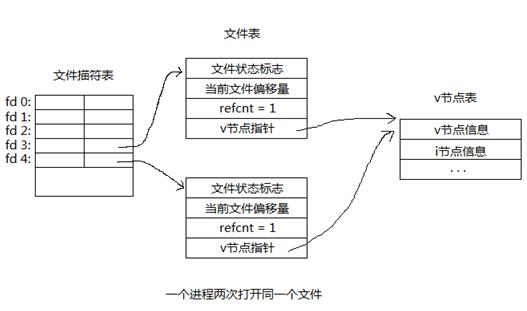
refcnt字段会相应变化。文件状态标识：文件的打开模式（R,W,RW,APPEND,NOBLOCK,等），当前文件偏移量，refcnt：被引用数量，

v节点指针：指向一个v节点表。

v节点表：每个文件对应一个，无论被被多少个进程打开都只有一个，它包括v节点信息（主要是stat结构体中的信息），i节点信息。

每个进程默认只能打开1024个文件描述符，当一个进程打开一个文件时，默认会从0开始查找未被使用的描述符，由于0,1,2默认被占用，所有一般从3开始使用。

2、一个进程两次打开同一个文件

[](http://images.cnitblog.com/blog/529981/201307/11163924-b2e1d6be3cc94b1cbb77dffb8060591b.png)

当一个进程多次打开同一个文件时，首先会在描述符表顺序查找未被使用的描述符，然后每打开一次建立一张文件表，但各文件表中的v节点指针都指向同一个v节点表。

示例程序：

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

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

{

int fd1;

int fd2;

char buf1[1024] = {0};

char buf2[1024] = {0};

fd1 = open("test.txt", O\_RDONLY);

if (fd1 == -1)

ERR\_EXIT("open error");

read(fd1, buf1, 5);

printf("buf1=%s\n", buf1);

fd2 = open("test.txt", O\_RDWR);

if (fd2 == -1)

ERR\_EXIT("open error");

read(fd2, buf2, 5);

printf("buf2=%s\n", buf2);

write(fd2, "AAAAA", 5);

memset(buf1, 0, sizeof(buf1));

read(fd1, buf1, 5);

printf("buf1=%s\n", buf1);

close(fd1);

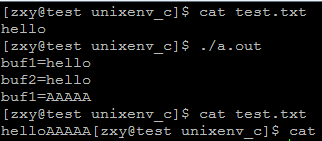
close(fd2);

return 0;

}

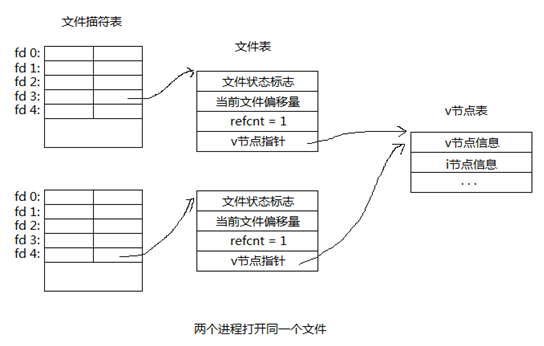
[复制代码](javascript:void(0);)

运行结果：

[](http://images.cnitblog.com/blog/529981/201307/11163925-c2d640fd17174558b3f89566600363de.png)

说明：先创建test.txt文件写入hello，再同一个进程两次打开该文件，可见每打开一次文件就参数一张文件表，不共享偏移量，都开始位置读取，之后利用第二个文件描述符写入AAAAA，在利用第一个描述符可以读取出，表明都指向同一个v节点表，操作同一个文件。

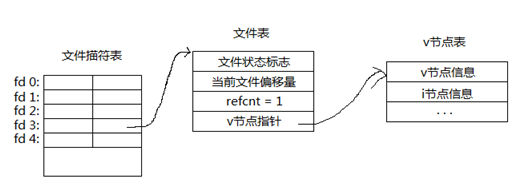
3，两个不同进程打开同一个文件

[](http://images.cnitblog.com/blog/529981/201307/11163926-69520508e0464d799eaeae22e991ac4a.png)

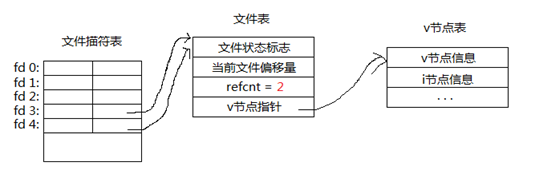
当不同进程打开同一个文件时，每个进程首先在它们各自的文件描述符表中顺序查找未被使用的描述符，最终获得的文件描述符可能相同也可能不同，每个fd指向各自的文件表，但同样，每个文件表中的v节点指针都指向同一个v节点表。

二、复制文件描述符

复制前：

[](http://images.cnitblog.com/blog/529981/201307/11163927-abd8b82be32a48a48ff8dfe68bb26055.png)

复制后：

[](http://images.cnitblog.com/blog/529981/201307/11163928-414d203e8ad846a0bb276258c9388148.png)

复制后，两个文件描述符都指向了同一个文件表，refcnt=2。

复制文件描述符有三种方法：

1，dup

2，dup2

#include <unistd.h>

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

DESCRIPTION   
       These system calls create a copy of the file descriptor oldfd.

       dup()  uses  the lowest-numbered unused descriptor for the new descriptor.

       dup2() makes newfd be the copy of oldfd, closing newfd first if  necessary, but note

                  the following:

       \*  If  oldfd  is  not a valid file descriptor, then the call fails, and newfd is not closed.

       \*  If oldfd is a valid file descriptor, and newfd has the same value as   
          oldfd, then dup2() does nothing, and returns newfd.

       After  a  successful return from one of these system calls, the old and new file descriptors may be used interchangeably.  They  refer  to  the same open file description (see open(2)) and thus share file offset and file status flags; for example, if the file offset is modified by using lseek(2)  on one of the descriptors, the offset is also changed for the other.

RETURN VALUE   
       On success, these system calls return the new descriptor.  On error, -1 is returned, and errno is set appropriately.

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

int main(void)

{

int fd;

fd = open("test.txt",O\_WRONLY);

if( fd == -1){

perror("open error");

exit(EXIT\_FAILURE);

}

int fd2;

fd2 = dup(fd);

if(fd2 == -1){

perror("dup error");

exit(EXIT\_FAILURE);

}

printf("oldfd = %d\n",fd);

printf("newfd = %d\n",fd2);

int fd3;

close(0); //或者close(STDIN\_FILENO)

fd3 = dup(fd);

if(fd3 == -1){

perror("dup error");

exit(EXIT\_FAILURE);

}

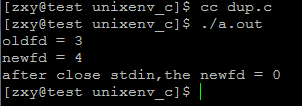
printf("after close stdin,the newfd = %d\n",fd3);

exit(EXIT\_SUCCESS);

}

[复制代码](javascript:void(0);)

运行结果：

[](http://images.cnitblog.com/blog/529981/201307/11163929-4ea77242686e45d0afaef12023c6b56f.png)

用dup进行文件描述符的复制时，顺序查找即从0开始查找可以文件描述符   
示例2：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

int main(void)

{

int fd;

fd = open("test.txt",O\_WRONLY);

if( fd == -1){

perror("open error");

exit(EXIT\_FAILURE);

}

int fd2;

fd2 = dup2(fd,0);

if(fd2 == -1){

perror("dup error");

exit(EXIT\_FAILURE);

}

printf("oldfd = %d\n",fd);

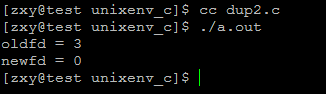
printf("newfd = %d\n",fd2);

exit(EXIT\_SUCCESS);

}

[复制代码](javascript:void(0);)

运行结果：

[](http://images.cnitblog.com/blog/529981/201307/11163930-80e37948a44d481e9a8f01915ca817b7.png)

用dup2进行文件描述符复制时，指定需要复制的新的描述符，如果该描述符已经被占用，则先关闭它在重新复制，类似于先调用close再dup

3，fcntl

功能：操纵文件描述符，改变已打开的文件的属性

#include <fcntl.h>

int fcntl(int fd, int cmd, ... /\* arg \*/ );

DESCRIPTION   
       fcntl() performs one of the operations described below on the open file   
       descriptor fd.  The operation is determined by cmd.

       fcntl() can take an optional third argument.  Whether or not this argu-   
       ment  is  required is determined by cmd.  The required argument type is   
       indicated in parentheses after  each  cmd  name  (in  most  cases,  the   
       required  type  is  long,  and  we identify the argument using the name   
       arg), or void is specified if the argument is not required.

由第二个参数指定操作类型，后面点的可变参数指定该命令所需的参数

这里我们进行文件描述符复制，可将cmd 设为： F\_DUPFD (long)，该命令表示：

Find the lowest numbered available file descriptor greater  than   
  or  equal to arg and make it be a copy of fd.  This is different   
from dup2(2), which uses exactly the descriptor specified.

  On success, the new descriptor is returned.

示例程序：

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

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

{

int fd;

fd = open("test.txt", O\_WRONLY);

if (fd == -1)

ERR\_EXIT("open error");

close(1);

if (fcntl(fd, F\_DUPFD, 0) < 0)

ERR\_EXIT("dup fd error");

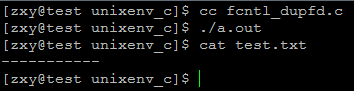
printf("-----------\n");//进行重定向，将不会显示在标准输出，

return 0;

}

[复制代码](javascript:void(0);)

运行结果：

[](http://images.cnitblog.com/blog/529981/201307/11163931-b455954234a0430a87c5e578890a943b.png)