#### 基本概念

该文档主要关注regular file；

###### file descriptor

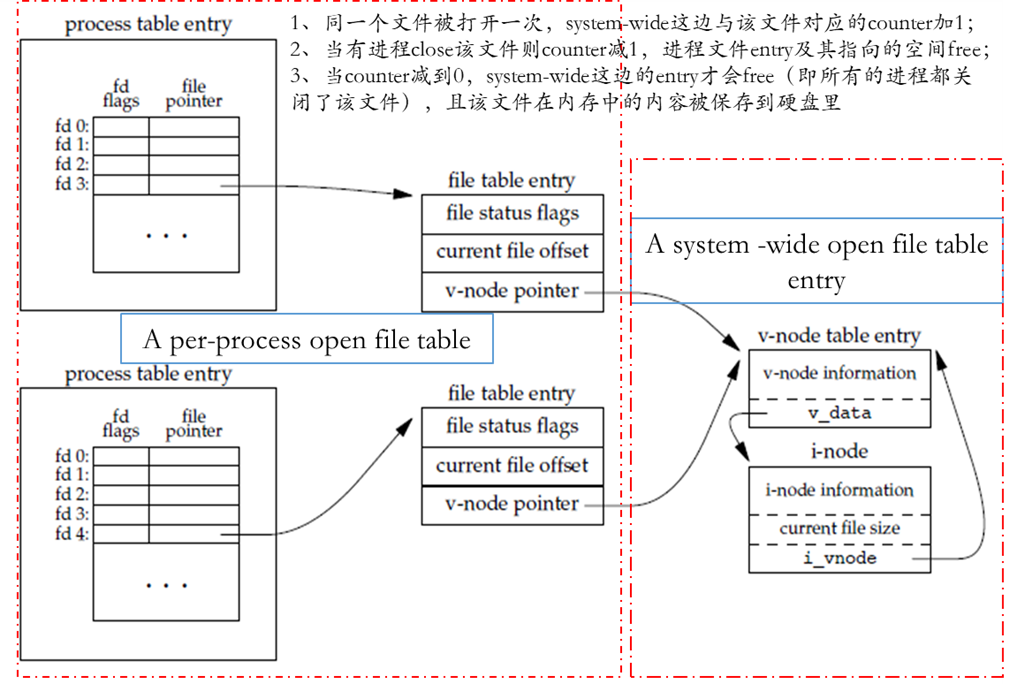
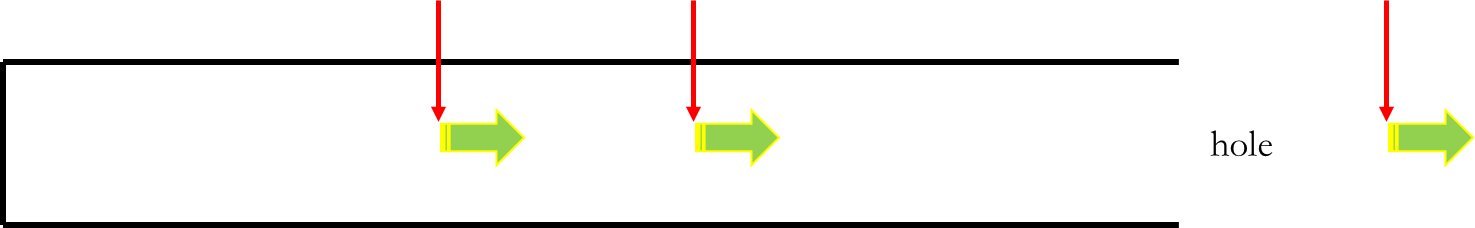


图1

1. 文件描述符是进程操作文件的索引，在进程内是唯一的：
   1. 第一次打开（可多次打开）一个文件时，内核会从硬盘导入文件相关信息（如inode），之后的文件操作一般以fd为参数；
2. 当打开或者创建时，内核会返回一个非负整数作为进程的文件描述符fd（默认是未使用且最小的number）；
3. 在一个进程中也可以多次打开同一个文件并得到各自不同的文件描述符，每个文件描述符也指向不同的file table entry；
4. 标准输入，标准输出以及标准错误输出的文件描述符一般是0，1，2：
   1. 很多application都遵循这个惯例；
   2. shell打开的时候把这3个描述符都定向到terminal（terminal也是文件，一切皆文件，也能以/dev/xxx/…文件名的形式打开）；
   3. 头文件unistd.h中的宏STDIN\_FILENO，STDOUT\_FILENO以及STDERR\_FILENO分别代表0，1和2（shell已经打开了，而其他进程相当于shell的子进程（fork\_&\_exec），子进程可以使用父进程的文件描述符）；



###### file stat（inode的核心内容）

struct stat {

**mode\_t st\_mode; /\* file type & mode (permissions) \*/**

ino\_t st\_ino; /\* i-node number (serial number) \*/

dev\_t st\_dev; /\* device number (file system) \*/

dev\_t st\_rdev; /\* device number for special files \*/

nlink\_t st\_nlink; /\* number of links(指向该文件的hard link数量即directory entry: "i-node number + filename") \*/

**uid\_t st\_uid; /\* user ID of owner \*/**

**gid\_t st\_gid; /\* group ID of owner \*/**

off\_t st\_size; /\* size in bytes, for regular files \*/

struct timespec st\_atim; /\* time of last access \*/

struct timespec st\_mtim; /\* time of last modification \*/

struct timespec st\_ctim; /\* time of last file status change \*/

blksize\_t st\_blksize; /\* best I/O block size \*/

blkcnt\_t st\_blocks; /\* number of disk blocks allocated \*/

};

**9个文件权限位**：

1. 涉及文件本身：
   1. 调用open函数，参数O\_RDONLY需要读权限，O\_WRONLY需要写权限，O\_WDWR需要读和写权限，O\_TRUNC需要写权限；
   2. 调用exec相关函数执行可执行文件（regular file）需要可执行权限；
2. 涉及文件所在的目录：
   1. 创建/删除文件需要有所在目录的写和执行权限，与文件本身无关；
   2. 以文件名打开文件，需要具备各层级目录文件的执行权限；

**7种文件类型：**

1. Regular file：最常见的文件类型，具体内容解释由应用程序决定；
2. Directory file：存储“文件名：执行文件相关信息的指针”二元组的文件，对目录文件有读权限的进程可以查看目录文件二元组内容；
3. 块设备文件：提供固定大小访问方式的设备文件如硬盘驱动；
4. 字符设备文件：提供任意大小访问方式的设备文件（所有的设备文件要么是字符设备，要么块设备）；
5. FIFO/PIPE：用于进程间交互的文件类型；
6. Socket：既可以用作不同主机的进程之间通过网络通信的文件，也可以用作相同主机进程之间的通信；
7. Symbolic link：文件内容包含指向另一个文件的信息；

<sys/stat.h>

S\_ISREG() regular file

S\_ISDIR() directory file

S\_ISCHR() character special file

S\_ISBLK() block special file

S\_ISFIFO() pipe or FIFO

S\_ISLNK() symbolic link

S\_ISSOCK() socket

if(S\_ISREG(buf.st\_mode) == 1)

**鉴权步骤：**

1. If the effective user ID of the process is 0 (the superuser), access is allowed（获得所有权限）
2. If the effective user ID of the process equals the owner ID of the file, access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied.
3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied.
4. If the appropriate other access permission bit is set, access is allowed.Otherwise, permission is denied.

在调用open函数时，需要的权限在某一步获得后，之后的鉴权步骤不再进行。

###### file descriptor和file stat的比较

1. open函数打开文件的时候，针对进程操作文件权限的鉴权已经开始（根据鉴权结果确定open函数是否可以按照输入参数将file staus flags 某些bits置位）；
2. read和write函数执行时先去check file status flags相关位是否开启（这不是鉴权）；

#### 文件基本的IO操作

“Linux一切皆文件”意思是它们在读取，写入，创建等概念上是相通的（视为文件看待），但是具体的操作函数不是统一的（open函数不能打开directory file）；

###### open和openat

A file is opened or created by calling either the open function or the openat function.

#include <fcntl.h>

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

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

int openat(int dirfd, const char \*pathname, int flags);

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

Both return: file descriptor if OK, −1 on error；

**参数说明**：

1. mode\_t类型的mode参数一般在创建新文件时才会用到，它对应着文件stat结构的mode\_t类型成员，决定文件的“可读可写可执行权限”（结合umask函数一起使用）；
2. flags参数决定打开的形式：
   1. O\_RDONLY 只读的方式打开；
   2. O\_WRONLY 只写的方式打开；
   3. O\_RDWR 读写的方式打开；

上述三种标志位是互斥的，也就是不可同时使用，但可与下面的标志位利用OR(|)运算符组合，可读可写不能“O\_RDONLY | O\_WRONLY”而是“O\_RDWR”（**看的时候认真一点**）

* 1. O\_EXRC 只执行的方式打开；
  2. O\_SEARCH

以上是文件打开的必选参数（可选一个或者多个），下面是可选参数：

1. O\_APPEND 每次写的时候都是在文件末尾写；
2. O\_CREAT 创建新文件（需要指明参数mode\_t mode）；
3. O\_TRUNC 如果文件存在而且成功以只读或者只写的方式打开，则将文件内容“清零”（truncate its length to 0）
4. O\_NONBLOCK If path refers to a FIFO, a block special file, or a character special file, this option sets the nonblocking mode for both the opening of the file and subsequent I/O.

…

**其他说明**：

open函数要求输入绝对路径名或者相对路径名（相对路径名从当前目录下开始），而openat可以输入相对路径名（输入绝对路径名时，参数dirfd被无视）：

1. dirfd是打开一个目录文件得到的文件描述符，相对路径就是从该目录文件下开始；
2. 如果fd是宏AT\_FDCWD，则相对路径从当前目录开始（和open函数一样）；

创建新文件：open(path, O\_WRONLY | O\_CREAT | O\_TRUNC, mode);

###### read

Data is read from an open file with the read function.

#include <unistd.h>

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

Returns: number of bytes read, 0 if end of file, −1 on error

**其他说明**：

1. 从当前file offset开始read，成功后更新file offset；
2. 如果还剩30个字节，而进程要read 100个字节，那么read返回30，下次再read返回0（EOF）；

###### write

Data is written to an open file with the write function.

#include <unistd.h>

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

Returns: number of bytes written if OK, −1 on error

**其他说明**：对于regular file，从当前file offset开始write，成功之后更新file offset；

###### lseek

An open file's offset can be set explicitly by calling lseek.

#include <unistd.h>

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

Returns: new file offset if OK, −1 on error

**参数说明**：offset的效果取决于whence（改变偏移量并影响到下次的read\_&\_write）

1. If whence is SEEK\_SET, the file's offset is set to offset bytes from the beginning of the file.
2. If whence is SEEK\_CUR, the file's offset is set to its current value plus the offset. The offset can be positive or negative.
3. If whence is SEEK\_END, the file's offset is set to the size of the file plus the offset. The offset can be positive or negative.

**其他说明**：该函数涉及到进程file entry的current file offset, normally a non-negative integer that measures the number of bytes from the beginning of the file（见图1）

1. 当文件打开时，current file offset默认是0；；
2. read和write一方面以current file offset开始读或写，另一方面读写完会更新current file offset值；
3. 可以通过off\_t currpos = lseek(fd, 0, SEEK\_CUR)获得当前file offset；
4. 以O\_APPEND方式打开文件：
   1. 开始的file offset是0，直到调用write，file offset先移到文件末尾，写完并更新；
   2. 之后不管用lseek将offset移到那里，read会从offset处开始，但是一旦调用write，offset会先移到文件末尾后再写；

###### pread和pwrite

Calling pread is equivalent to calling lseek followed by a call to read, with the following exceptions (pwrite is the same).

#include <unistd.h>

ssize\_t pread(int fd, void \*buf, size\_t nbytes, off\_t offset);

ssize\_t pwrite(int fd, const void \*buf, size\_t nbytes, off\_t offset);

其他说明：

1. 这两个函数都是atomic operation，在操作文件时其他进程不能interfere；
2. 它们执行完后，file offset是不更新的（与write和read不一样）；

###### close

An open file is closed by calling the close function.

#include <unistd.h>

int close(int fd);

Returns: 0 if OK, −1 on error

**其他说明**：当进程关闭时，它打开的所有文件都自动被内核关闭（很多程序利用这一点往往不在程序里调用close）；

###### 多进程共享文件（write）

1. 当多个进程向同一个文件写时，可能会出现覆盖的现象，因为各自的file's current offset不一样（此时建议采用pwrite函数或者用O\_APPEND打开文件）；
2. 有的进程读有的进程写同一个文件时，抓住它们各自的file's current offset的变化便可理解所发生的现象；

###### fcntl

The fcntl function can change the properties of a file that is already open.

#include <fcntl.h>

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

Returns: depends on cmd if OK (see following), −1 on error

**参数说明**：第3个参数的含义（int arg）取决于cmd，当cmd取：

1. F\_DUPFD：Duplicate the file descriptor fd，并返回新的file descriptor（该值遵循“最小未打开原则”）。新的fd和原来的fd共享file table entry（包括file status flag这些：RD\_ONLY，WR\_ONLY…），但file descriptor flag不一样（FD\_CLOEXEC清零）；
2. F\_DUPFD\_CLOEXEC：作用和F\_DUPFD基本一样，不同是FD\_CLOEXEC file descriptor flag置位；
3. F\_GETFD：返回fd对应的file descriptor flag（不是status flag，见图1），目前只定义了一种flag即FD\_CLOEXEC；
4. F\_SETFD：设置fd对应的file descriptor flag为arg（第3个参数）；
5. F\_GETFL：返回file status flags（返回int类型，flags涉及到位操作相关），这些flag是open函数用到的O\_RDONLY，OWRONLY，O\_APPEND…所涉及的相关位；
6. F\_SETFL：用第3个参数值设置file status flags，但只能设置O\_APPEND, O\_NONBLOCK, O\_SYNC, O\_DSYNC, O\_RSYNC, O\_FSYNC, 和O\_ASYNC（可以用或|设置多个）.

…

**其他说明：**开启 / 关闭文件属性的思路（关闭可以是与非）

void

set\_fl(int fd, int flags) /\* flags are file status flags to turn on \*/

{

int val;

if ((val = fcntl(fd, F\_GETFL, 0)) < 0)

err\_sys("fcntl F\_GETFL error");

val |= flags; /\* turn on flags \*/

//val &= ˜flags; /\* turn flags off \*/

if (fcntl(fd, F\_SETFL, val) < 0)

err\_sys("fcntl F\_SETFL error");

}

#### 文件stat结构查询与修改

mode\_t st\_mode和uid\_t st\_uid、gid\_t st\_gid

###### 总结

**进程修改new file和existing file属性：**

1. **new file**创建时9个权限位由umask函数（一般保持原样即002：只禁止其他人写）和open函数的mode\_t mode参数决定：mode = (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | …)；
2. **new file**的owner user ID和进程的effective user ID一样，owner group ID取决于所在的directory文件的set-group-ID置位情况（要是进程的effective group ID，要么是其所在directory的owner group ID）
3. **existing file**的9个权限位以及set-user-ID位、set-group-ID位的修改是调用chmod, fchmod 和 fchmodat
   1. 方法也是mode = (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | …)的形式；
   2. 可修改的前提条件是进程的effective user ID和文件owner user ID一样（或者进程拥有root权限）；
4. **existing file**的owner user ID和owner group ID修改是调用函数chown, fchown, fchownat, 和lchown：
   1. 可修改的前提条件是进程具有超级权限（effective user ID是0）；

new file的set-user-ID和set-group-ID设置没有具体函数可以直接设置，可以创建完成之后再调用chmod函数修改。

**mode\_t st\_mode成员变量中set-user-ID位和set-group-ID位**：

1. 对于可执行文件，这2位置位决定可执行文件执行时（进程）的effective user ID和effective group ID；
   1. 可执行文件作为文件有文件的属性，即有owner user ID和owner group ID，它变成进程后，有进程的属性即：
      1. real user ID和real group ID；
      2. effective user ID和effective group ID以及supplementary group ID；
      3. saved set-user-ID和saved set-group-ID
2. 对于directory文件，set-group-ID决定进程创建文件时，owner group ID是什么；

可以通过宏S\_ISUID和S\_ISGID检查st\_mode中的这两位是否置位；

###### stat, fstat, fstatat, and lstat

get file stat.

#include <sys/stat.h>

int stat(const char \*restrict pathname, struct stat \*restrict buf );

int fstat(int fd, struct stat \*buf );

int lstat(const char \*restrict pathname, struct stat \*restrict buf );

int fstatat(int fd, const char \*restrict pathname,

struct stat \*restrict buf, int flag);

All four return: 0 if OK, −1 on error

###### access, faccessat

(a processs want to) test to accessibility (read write and excute) based on the real user ID and real group ID (or effective user ID and effective group ID)

#include <unistd.h>

int access(const char \*pathname, int mode);

int faccessat(int fd, const char \*pathname, int mode, int flag);

Both return: 0 if OK, -1 on error

**参数说明：**

mode要么是F\_OK（检验文件是否存在），要么是R\_OK，W\_OK以及X\_OK中的一种或几种（这3个可以通过OR|组合）；如果flag是AT\_EACCESS，那么access checks用的是effective ID；

**其他说明**：

默认是用real ID去check，所以当real ID和effective ID不一样时，可能出现permission denied但open成功的情况（或者反过来）；check的步骤和鉴权步骤一致；

###### umask

set the file mode creation mask（9个权限位） for the process and return the previous value;

#include <sys/stat.h>

mode\_t umask(mode\_t cmask);

**参数说明**：

cmask的形式是S\_IRUSR | S\_IWUSR | …，即对应9个权限位的mask；

**其他说明**：

1. cmask某位是on，则对应的file's mode权限位是off，cmask是0代表所有位都不禁止；
2. shell常用的umask值是002（八进制表示：000 000 010），表示当自己创建新文件时，要关闭other对自己文件的写权限（shell打开的进程会继承它的umask值，除非进程自己再调用umask函数修改）；

###### chmod, fchmod, and fchmodat

change the file access permissions for an existing file

#include <sys/stat.h>

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

int fchmod(int fd, mode\_t mode);

int fchmodat(int fd, const char \*pathname, mode\_t mode, int flag);

**参数说明：**

mode Description

S\_ISUID set-user-ID on execution

S\_ISGID set-group-ID on execution

S\_ISVTX saved-text (sticky bit，这个不用管)

S\_IRWXU read, write, and execute by user (owner)

S\_IRUSR read by user (owner)

S\_IWUSR write by user (owner)

S\_IXUSR execute by user (owner)

S\_IRWXG read, write, and execute by group

S\_IRGRP read by group

S\_IWGRP write by group

S\_IXGRP execute by group

S\_IRWXO read, write, and execute by other

S\_IROTH read by other

S\_IWOTH write by other

S\_IXOTH execute by other

调用方法也是mode = (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | …)的形式

**其他说明：**

要改变existing file的权限，进程的effective user ID要等于文件的owner ID或者进程的effective user ID是０（即拥有root权限）；这3个函数需要的条件和“可读可写可执行”需要的权限是有区别的（后者对应4步鉴权）

**例程：**

struct stat statbuf;

if (stat("foo", &statbuf) < 0)

err\_sys("stat error for foo");

if (chmod("foo", (statbuf.st\_mode & ˜S\_IXGRP) | S\_ISGID) < 0) //turn on set-group-ID and turn off group-execute

err\_sys("chmod error for foo");

if (chmod("bar", S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH) < 0) //set absolute mode to "rw-r--r--"

err\_sys("chmod error for bar");

运行结果：

$ ls -l foo bar

-rw-r--r-- 1 sar 0 Dec 7 21:20 bar

-rw-rwSrw- 1 sar 0 Dec 7 21:20 foo

这个S处于execute权限的位置，代表set-user-ID和set-group-ID；

###### chown, fchown, fchownat, and lchown

change a file's user ID and group ID（改变uid\_t st\_uid和gid\_t st\_gid）

#include <unistd.h>

int chown(const char \*pathname, uid\_t owner, gid\_t group);

int fchown(int fd, uid\_t owner, gid\_t group);

int fchownat(int fd, const char \*pathname, uid\_t owner, gid\_t group, int flag);

int lchown(const char \*pathname, uid\_t owner, gid\_t group);

all four return 0 if OK, -1 on error

**其他说明：**

lchown和fchownat（flag参数是AT\_SYMLINK\_NOFOLLOW）是改变symbolic link文件本身而不是其指向的文件；Linux系统下，只有具备超级权限的进程才可以调用这4个函数；

###### truncate and ftruncate

truncate and existing file to length bytes

#include <unistd.h>

int truncate(const char \*pathname, off\_t length);

int ftruncate(int fd, off\_t length);

both return: 0 if OK, -1 on error

**其他说明**：

如果是“截长”了，那么原文件的末端到新的末端之间是0（hole）；如果是截短，那么这个过程是不可逆的，包括以O\_TRUNC的方式打开文件也是如此，这时候再将其“截长”回来，原来的内容不会恢复，而是0；

可以使用这两个函数截短文件，或者清空文件内容（将文件大小变0）；

#### 补充

1. wrtie(fd, buf, n)和read(fd, buf, n)，文件和buf之间数据往返都是低地址字节对应低地址字节，高地址对应高地址；
2. ram和文件的逻辑地址都是紧邻的（lseek偏移地址+1在逻辑上是移动到下一个相邻的高地址）；