**文件描述符**

内核把所有打开的文件通过描述符引用,文件描述符是一个非负整数,当打开一个现有文件或者创建一个新文件时,内核向进程返回一个文件描述符

**open函数**

int open(const char \*pathname,int oflag,...);

patchname表示要打开或者创建的文件名

oflag表示对打开的文件后续操作权限及操作属性

如下三者选一

O\_RDONLY 只读打开

O\_WRONLY 只写打开

O\_RDWR 读写打开

如下可多选

O\_APPEND 每次写时追加到文件的尾端

O\_CREAT 若文件不存在则创建

第三个参数只有在文件创建时才会用到,用来表示创建文件的访问权限

返回值:成功返回文件描述符,失败返回-1

open返回的文件描述符一定是最小的未使用的文件描述符

**creat函数**

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

等价于

open(pathname,O\_WRONLY | O\_CREAT | O\_TRUNC, mode);

早起的open不能打开不存在的文件,现在的可以,所以creat没有必要再使用

返回值:成功返回文件描述符,失败返回-1

**close函数**

int close(int fileds);

返回值:成功返回0,失败返回-1

当进程终止时,内核会自动关闭其所打开的文件,所以很多程序都不显示调用close关闭文件

**lseek函数**

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

每一个打开的文件都有一个与其关联的”**文件偏移量**”通常为非负整数(某些设备文件可以允许其为负数,但是普通文件一定是非负数),表示从文件开始处的字节数,就是标明在文件的哪个位置,这个在读写文件接口都是从这个位置开始读写文件,当读写完毕,文件偏移量会加上读写的字节数.打开一个文件时,除非指定O\_APPEND选项,否则文件偏移量被设置为0

lseek函数可以显式指定一个打开的文件的偏移量

若whence是SEEK\_SET ,则表示将文件偏移量设置为距文件开始处的offset个字节

若whence 是SEEK\_CUR,则表示将该文件的偏移量设置为当前值加上offset,offset可为正负

若whence是SEEK\_END,则表示将该文件的偏移量设置为文件长度加上offset,offset可为正负

返回值:成功则返回最新的文件偏移量,错误返回-1

修改偏移量,则下次读写时会生效,文件偏移量可以大于文件的长度,此时如果写文件,则文件上次结尾到这次开始之间的为文件的空洞,空洞不占用磁盘,读出时表现为0

可用od -c命名以字符方式打印文件

**read函数**

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

返回值:成功则返回读到的字节数,失败返回-1

读取指定文件nbytes个字节,放入buf指定的内存中,有如下几种情况读成功,但是读不到指定的字节数

1:当前偏移到文件结尾无nbytes个字节

2:当从终端读设备时,最多一次读一行

**write函数**

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

返回值,成功返回写入的字节数,失败返回-1

通常写nbytes则返回值就是nbytes,否则出错

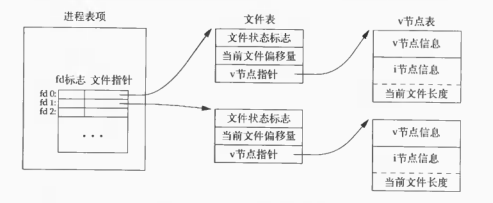
**I/O效率**

讨论的是read write中的nbytes多少的时候读写效率较高,

**文件共享**

unix支持多个进程共享一个文件

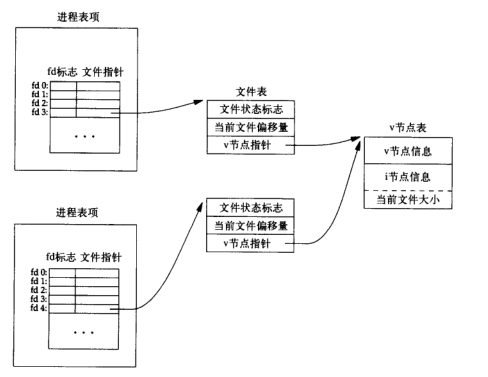
每个进程的描述符中有一个记录项,其中包含一个打开的文件描述符表,并包含每个描述符指定文件的文件表,文件表示内核为打开的文件维护的一个表,其中的v节点(v-node)比较重要(**每个文件只有一个v节点表**),v节点的中的**i节点(i-node) 有文件的所有者,文件长度,文件实际数据块在的磁盘等信息**.



当两个进程打开同一个文件时,内核中有如下实现:

同一文件被不同进程打开都会有一个文件表,同一个文件的文件偏移量对不同进程是不一样的.当然在不同进程打开同一个文件时,可以让他们都指向同一个文件表,如fork创建父子进程时,父子进程对于每一个打开的文件描述符共享同一个文件表

同一个文件不管被多少进程打开只有一个v节点



dup和dup2函数

int dup(int filedes);

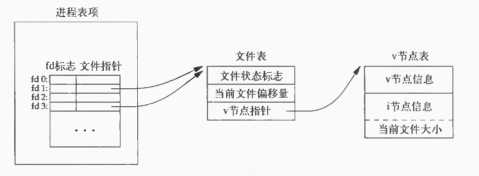
int dup(int filedes,int filedes2);

返回值:成功返回新的文件描述符,出错返回-1

这两个函数可用来复制一个现存的文件描述符,dup返回的文件描述符是可用文件描述符中的最小数值,dup2则可以指定filedes2,若filedes2已经打开,则将其先关闭.

内核中实现如下:

新的描述符和老的描述符的对应**文件表**是同一个



**sync fsync和fdatasync函数**

int fsync(int filedes);

int fdatasync(int filedes);

void sync(void);

返回值:成功返回0;失败返回-1

unix中设有缓冲区,大多数的磁盘I/O都是通过缓冲进行的,当数据写入文件时,内核通常先将该数据复制到一个缓冲区中,如果缓冲器未写满,则不将其排入输出队列(当输出队列到达队首时,进行实际的I/O操作),这种输出方式称为延时写.当系统发生故障时,延时写可能导致数据都是,unix提供以上三个接口让数据尽快进行实际I/O操作.

sync函数是将**所有缓冲区**排入输出队列,然后返回,内核线程update会周期性(一般30s)的调用sync函数

fsync函数**对指定的缓冲区**起作用,且等待磁盘操作结束才返回.其同步的是文件的数据及文件的属性

fdatasync函数类似fsync,但只影响数据部分.

fcntl函数

int fcntl(int filedes,int cmd,...);

返回值:成功则依赖cmd.失败返回-1

用于改变已经打开文件的性质,主要有5种功能

1:**复制一个现有的描述符** cmd= **F\_DUPFD**

2:获得/设置**文件描述符标记** cmd=**F\_GETFD**或**F\_SETFD**

3:获得/设置**文件状态标志** cmd=**F\_GETFL**或**F\_SETFL**

...

**ioctl函数**

int ioctl(int fileds,int request,...);

I/O操作的杂物箱

返回值：成功返回其他值，失败返回-1

**stat fstat和lstat函数**

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

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

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

返回值：成功返回0，出错返回-1

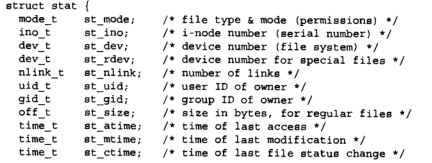
stat函数返回指定文件（如果是符号链接则是其指向的文件）的信息结构

fstat函数获取指定描述符对应文件的信息结构

lstat类似stat，只是当文件是符号链接时，返回符号链接的信息结构

例如使用ls -l命令是就会调用此函数

其中返回的文件信息结构为：





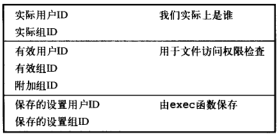
**文件类型**

unix系统包含的文件类型有7种，stat结构中的st\_mode表示文件类型，其值有如下几种



**设置用户ID和设置组ID**

**一个进程自身会有6种ID**



实际用户ID和实际组ID:取自登陆时的口令文件中的登陆项，也就是登陆的用户名等，一般登陆期间不会改变，超级用户可以修改它们。

有效用户ID和有效组ID和附加组ID:决定我们对文件的访问是否有权限

保存的设置用户ID和保存的设置组ID:**由exec保存进程执行exec命令时加载的可执行文件的设置用户ID和设置用户组ID**

**每个文件有两个所有者ID**

文件**所有者**ID(stat.st\_uid)

文件**用户组**ID(stat.st\_gid)

当以一个用户身份登陆系统时，在登陆期间启动的进程自身的有效用户ID是实际用户ID,有效组ID是实际组ID

设置文件的stat.st\_mode中的**设置用户ID**位S\_ISUID，可使“当执行此文件时（由进程调用exec函数加载此执行文件），将**进程的有效用户ID**设置为**文件的所有者ID**”

设置文件的stat.st\_mode中的**设置用户组ID**位S\_ISGID，可使“当执行此文件时（由进程调用exec函数加载此执行文件），将**进程的有效用户组ID**设置为**文件的组所有者ID**”

例如文件的所有者是超级用户，且设置了文件的设置用户ID,则当一个进程（不管这个进程的用户ID是什么）调用exec执行这个可执行文件时，新创建的进程(exec为fork后调用)或者当前的进程（在当前进程中直接调用exec,覆盖当前进程的进程地址空间）变得具有超级用户特权，

**文件的访问权限**

文件有一个**所有者(文件属于谁)**，所有者在一个组中，叫文件所属**用户组（文件属于哪个组），文件相当于一个物品**

进程有一个**有效用户ID（是谁）**和**有效组ID（进程所掌控的组）和一个附加组ID**,**相当于一个人**

进程（进程中的各种函数）**访问**一个文件，(进程ID为0的对所有文件都可访问)首先判断进程是不是文件的所有者，或者进程是不是文件的所有组主人，还是说进程对于文件来说是other其他人，确定进程对文件的身份后，再判断这个身份对于文件可以操作可读可写可执行中的哪些。

特别的，当进程是执行exec加载**可执行的文件**时，若文件的设置用户ID或者设置组ID被设置，则文件会设置进程的有效用户ID和有效组ID（在进程原有的基础上，让文件属于这个进程）

特别的如果文件的设置用户ID或者设置用户组ID被设置，则进程会的有效用户ID和有效组ID会变成文件的所有者和用户组ID

unix中有7种文件类型，每个文件都会三种权限（可读，可写，可执行）权限存在于stat\_st\_mode中，标明**文件所有者**，**文件用户组**，**其他人other**对这个文件的可读，可写，可执行权限

对于目录，想要打开目录，就是执行到其指向的文件或文件夹，必须对目录具有执行权限

**access函数**

如上所说当一个可执行的文件设置了设置所有组ID或设置组ID后，进程调用exec函数执行它时，它会让进程成为自己的主人，也就是让进程的有效用户组ID或有效组ID变成文件的所有者或用户组

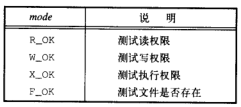
但是进程可能并不想改变自己的有效用户组ID或有效组ID，因为当用户已某个账号登陆后，登陆期间启动的进程都是与这个账号有关的，也就是进程的**实际ID**(实际用户ID和实际组ID)和**有效ID**(有效用户ID和有效用户组ID)是**这个账号的用户ID和组ID**,这样进程可以访问所有属于这个账号的用户ID和组ID的文件资源，当外界将进程的有效ID改变后，

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

返回值：成功返回0，失败返回-1

这个函数测试进程按实际ID是否能访问一个文件，注意能不能访问一个文件是由有效ID决定的，access只是一个测试功能

mode参数如下：



**新建文件的所有者和用户组**

新建的文件（7种类型）的所有者为进程，用户组为进程所掌控的组，也就是文件的所有组ID为进程有效用户ID,文件的用户组ID为进程的有效组ID(**有些规范让文件的用户组ID为它所在目录的组ID**)

但是创建时文件的文件所有者，文件用户组，other的具体读写执行怎么确定呢？这由进程的**文件模式创建屏蔽字umask值**来确定，这个文件模式屏蔽字可有umask函数设置

**umask函数**

mode\_t umask(mode\_t cmask);

cmask可以是如下常量“或“构成的



如umask(S\_IRUSR | S\_IROTH);//禁止用户读和其他读

返回值：返回以前的文件模式屏蔽字

**chmod 函数和fchmod函数**

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

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

这两个函数用于改变已存在的文件的文件模式屏蔽字

一个是通过文件路径，一个是通过打开的文件描述符

注意：只有超级用户或者文件的主人（文件的用户或者用户组是某个进程）才能改变文件的屏蔽字

mode由如下常量“或“构成的



S\_I**SUID** S\_I**SGID**

S\_I**R**USR S\_I**W**USR S\_I**X**USR S\_I**RWX**U

S\_I**R**GRP S\_I**W**GRP S\_I**X**GRP S\_I**RWX**G

S\_I**R**OTH S\_I**W**OTH S\_I**X**OTH S\_I**RWX**O

备注：chmod值修改文件的i节点信息，不修改文件内容

**chown fchown和lchown函数**

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

int fchown(int filedes,uid\_t owner,git\_t group)

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

返回值：成功返回0，失败返回-1

这些函数用于**修改文件(**给定的文件，给定打开文件的文件描述符，给定符号链接文件本身**)**的**用户ID**和**用户组ID**

如果不修改用户ID或者用户组ID则对应的参数放-1

一般只有超级用户和文件的主人才能成功执行此接口

**文件的长度**

**stat.st\_size**表示文件（只限于**普通文件，目录文件，符号链接文件**）以**字节为单位**的**文件长度**

对于有空洞的文件如何查看

ll可以查看文件大小，包含空洞，du命令则是查看文件所使用的磁盘空间总量，这是真实值

如：

说明core有很多空洞

**文件截短**

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

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

将文件截断到只有length

**文件系统**

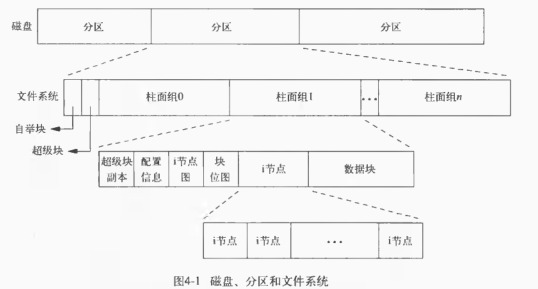
对于一块磁盘，想使用的话首先得分区，然后对分区进行格式化成系统能识别出的文件系统，**文件系统就相当于将一段给定的磁盘存储空间用抽象的文件概念来管理**，操作文件就相当于操作了磁盘，**逻辑块**是将分区格式化为文件系统是指定的最小存储单位

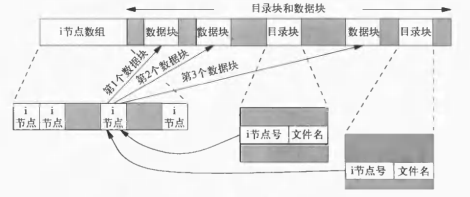
linux中最标准的文件系统ext2，这个文件系统中将文件内容分为两部分存储，一个是文件的属性，一个是文件的内容，属性用inode存储，内容用块存储（所有的逻辑块大致分为两大区域，一个用于放所有的indeo结点，一个用于放文件内容）

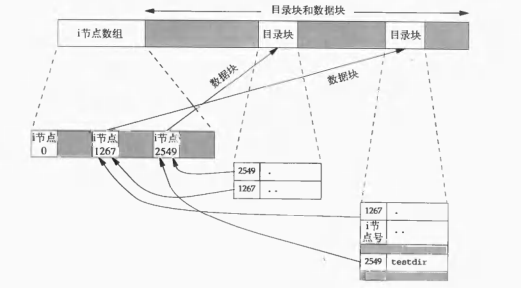
linux中创建一个普通文件时，会分配到至少一个inode（存放文件的相关属性及文件内容的块指针）和能放下文件内容的N个块（记录文件内容）

linux中创建一个目录文件时，会分配到一个inode（存放目录的相关属性及目录内容的块指针）和至少一个块（用于记录这个目录下的相关文件或目录的关联性，也就是此目录下文件或目录的inode结点位置在哪，**每一项成为一个目录项**）

inode记录文件或者目录的属性（**文件所有者，用户组，访问权限，文件设置用户及设置用户组位，文件类型，文件三个时间，文件大小，文件**），同时具有指针的作用，指向文件或目录内容放置的块







由上图得出两个结论：

1：文件名存放在目录的内容块中

2：看到的**文件夹名A**是上一级目录块中文件夹名，上一级目录块中还有指向这个文件夹A的inode指针

看到的**文件名**是上一级目录块中的文件名,上一级目录块中还有指向这个文件的inode指针

**linux中读取文件的内容过程**

读取一个普通文件/etc/www/wang.c的内容过程如下：

从根目录/的inode结点开始找，找到/下etc目录的inode节点，从ect的inode结点找到其下的www目录的inode节点，再从这个inode节点找到wang.c的inode节点，然后根据这个inode节点找到wang.c内容块。

备注：ext2文件系统建立后，inode节点数和填写内容的块数目就已经固定

当目录下的文件内容太多，导致一个块无法容纳所有关联数据，linux会给该目录多个块，用来记录关联数据

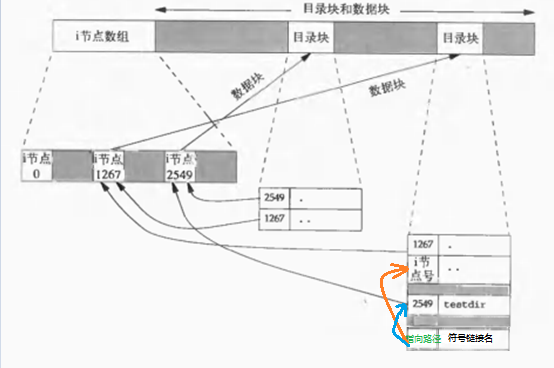
**符号链接**

个人的理解在当前目录的目录块中，增加一个目录项，目录项中存放**符号链接名**和其**指向的路径（某个目录项）**

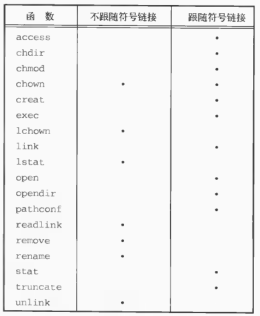
如符号链接名为link1，指向/home/wang,不管link1放到哪个目录，都链接到/home/wang

如符号链接名为link2，指向./wang，则不管link2放到哪个目录，都链接到./wang,如果没有这个文件则报错。

备注：符号链接没有文件系统的限制（各种文件系统想要在linux上使用，都得有目录项这些框架规范），



当一些文件IO函数的参数是一个文件，当这个文件是符号链接时，处理的是符号链接本身还是其指向的文件，以下列出常用接口操作



**link函数**

int link(const char \*existingpath,const char \*newpath);

返回值：成功返回0，失败返回-1

创建一个指向现有文件的**硬链接**，此函数创建一个新的目录项（也就是目录块中的一项），注意一般不支持创建目录的硬链接（超级用户可创建），主要是防止文件系统形成死循环

**symlink 和readlink函数**

int symlink（const char \*actualpath,const char \*sympath）；

该函数用于创建一个符号链接，

ssize\_t readlink(const char \* restrict pathname,char \*restrict buf,size\_t bufsize);

该函数用于打开一个符号链接本身

**unlink remove函数**

int unlink(const char \*pathname);

删除一个现有的目录项，可以是**文件，目录**，或者**符号链接**

删除现有的目录项，并将由pathname所引用文件的链接计数减一（符号链接不会增加文件的链接计数），当这个文件的链接计数为0时，系统会从磁盘删除文件块内容，特别的当进程打开一个文件时，即使链接计数为0，文件内容也不会删除，根据这个特性一般可以实现如下操作

**当进程崩溃时，其所创建的临时文件也要删除**，可以在用open或者creat创建一个文件后，立即调用unlink将这个文件的链接计数变为0，当进程崩溃时，系统会回收文件所占块。

备注：对于所给的文件如果是符号链接，则unlink删除该符号链接，不会删除其指向的文件

int remove(const char \*pathname);

返回值：成功返回0，出错返回-1

删除对一个文件或者目录的链接

**rename****函数**

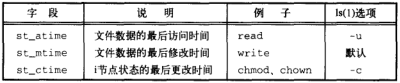
int rename(const char oldname,const char \*newname);

更改**文件，目录**，或者**符号链接**的名字

**utime函数**

int utime(const char \*pathname,const struct utimbuf \*times);

一个文件的时间有三种，



修改一个文件的时间(只能修改stat.st\_atime,stat.st\_mtime,调用utime函数后st\_ctime会自动更新，因为i节点中存放了这三个time数据)

返回值：成功返回0，失败返回-1

struct utimbuf{

time\_t actime,//访问文件的时间

time\_t modtime,//修改文件的时间

};

当times为NULL,则访问时间和修改时间都设置为当前时间

当times为非空,则文件的访问时间和修改时间设置为times的时间

**mkdir和rmdir函数**

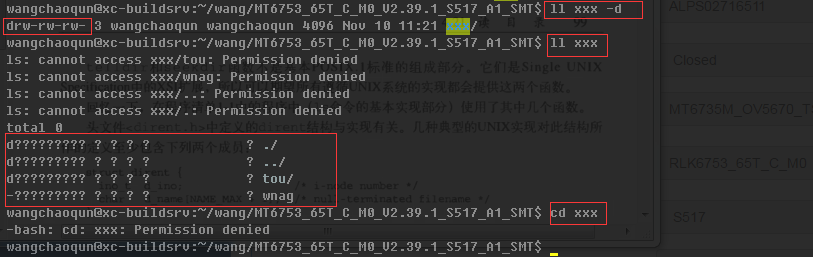
int mkdir(const char \*pathname,mode\_t mode);

int rmdir(const char \*pathname);

用于创建一个目录或者删除一个目录

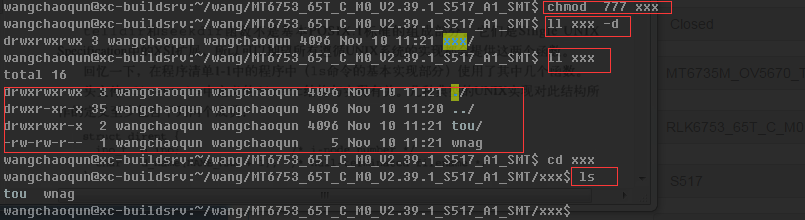
**读目录**

如果目录的权限位有r，则可以访问目录中的文件名，但各个文件的权限等属性看不到



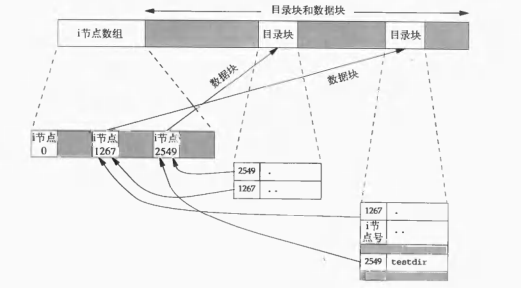
如果目录的权限位有wx，则可以进入目录，看到其下文件的属性等

（**为了文件系统的安全，只有内核才能写目录**）



本质原因如下：

我们所看到的目录testdir是其上一级目录的目录块中的一个目录项



对目录可读的意思是可以找到这个目录testdir的inode节点指针2549，进而找到其目录块，就能看到各文件名

但要看到各个文件的属性也就是inode节点，则必须要先进入指针2549所代表的目录块中，这需要对testdir有wx权限

目录项的结构至少包含如下部分:

struct dirent{

ino\_t d\_ino;//inode节点号

char d\_name[NAME\_MAX + 1] //文件名或目录名

};

有如下操作目录的接口

DIR \***opendir**(const char \*pathname);

struct dirent \***readdir**(DIR \*dp);

void **rewinddir**(DIR \*dp);

long **telldir**(DIR \*dp);

void **seekdir**(DIR \*dp,long loc);

DIR结构是保存当前正在读取的目录的有关信息，类似FILE结构

readdir读取目录中的第一个目录项，目录项中的顺序与实现有关，通常不安字母排序

20161110 wangchaoqun