**VFS文件打开&&路径名解析**

**VFS路径搜索重要内存数据结构：**

/\*用来存放相关路径名查找结果\*/

struct nameidata {

struct path path;/\*将目录结构和mount结构封装在path结构中\*/

struct qstr last;

struct path root;

unsigned int flags;/\*对应搜索的标志\*/

int last\_type;//路径名最后一个分量类型，LOOK\_PARENT设置时使用

unsigned depth;//符号链接的嵌套级别，必须小于6

char \*saved\_names[MAX\_NESTED\_LINKS + 1];

//与符号链接相关联的路径名数组

/\* Intent data \*/

union {

struct open\_intent open;//单个成员结构体，指定如何访问文件

} intent;

};

/\*目录项名字数据结构，用来存放当前节点的哈希值以及节点名的长度\*/

struct qstr {

unsigned int hash; //哈希值

unsigned int len; //节点名长度

const unsigned char \*name; //节点名

};

struct path {

struct vfsmount \*mnt; //已安装文件系统对象

struct dentry \*dentry; //最后一个路径分量的目录项对象

};

**文件的打开操作大致流程：**

1)在当前进程的文件描述表fdtale中分配一个空的文件描述符fd

2)在filp\_cachep中创建一个file struct

3)调用do\_path\_lookup()找的文件的inode ,取出inode的文件操作方法file\_operations赋给file struct ,并调用f->f\_op->open()执行打开的操作

4)根据根据文件描述符fd将file安装在当前进程文件描述表fdtable对应的位置.

这样在以后读写文件操作时,只需要根据文件描述符fd,就可以在当前进程的描述表中找到该文件的file对象,然后调用f->f\_op操作方法对文件进行操作即可。因此文件打开操作是读写操作的前提。

open（）系统调用的服务例程为do\_sys\_open（）函数，该函数接受的参数为：要打开的路径名filename，访问模式标志flags，以及如果该文件被创建所需要的许可权位掩码mode。如果该系统调用成功，就返回一个文件描述符，也就是指向文件对象的指针数组current->files->fd中分配给新文件的索引，否则返回-1

**文件的打开操作（open）实际上包含了文件的创建操作（create）：**

sys\_open()：

SYSCALL\_DEFINE2(creat, const char \_\_user \*, pathname, int, mode)

{

return sys\_open(pathname, O\_CREAT | O\_WRONLY | O\_TRUNC, mode);

}

Note：

O\_CREAT：若文件不存在，则创建该文件；

O\_WRONLY：为写而打开；

O\_TRUNC：截断文件(删除所有现在的内容)；

**VFS的open过程**

get\_unused\_fd\_flags()

->files\_fdtable()

//获取fdtabel入口

->find\_next\_zero\_bit()

//查找空fd

->expand\_files()

//fd数组不够用时进行拓展

//此函数在current->files->fd中查找一个空的位置，相应索引放在fd局部变量中

**VFS路径名解析**

do\_sys\_open()

-> getname()

//将文件名从用户空间拷贝到内核空间

-> get\_unused\_fd\_flags()

-> do\_filp\_open()

-> fsnotify\_open()//将文件加入监控系统

-> fd\_install()

//将file对象加入fd索引位置

path\_walk ()

……

current->total\_link\_count = 0;

//与符号链接查找相关

result = link\_path\_walk(name, nd);

//路径名查找的核心

path\_lookup()

->do\_path\_lookup()

do\_path\_lookup()

-> path\_init(dfd, name, flags, nd);//找到搜索起点

-> path\_walk(name, nd);//从起点开始搜索

path\_init ()

……

nd->last\_type = LAST\_ROOT;

//此值随搜索结果而改变，若成功找到目标文件，为LAST\_NORM；若停留在了一个“**.**”上，则变成LAST\_DOT;若是“**..**”,则为LAST\_DOTDOT

if (\*name=='/') {//绝对路径

set\_root(nd);//设置nd的root为当前root

nd->path = nd->root;//保存根目录

path\_get(&nd->root);//递增引用计数

} else if (dfd == AT\_FDCWD) {//相对路径

struct fs\_struct \*fs = current->fs;

read\_lock(&fs->lock);

nd->path = fs->pwd;//保存当前路径

path\_get(&fs->pwd);//递增引用计数

read\_unlock(&fs->lock);

}

//函数主要功能就是为搜索做准备，设置搜索起点

link\_path\_walk()

//以“/”为分隔符，解析路径名分量

对于每个不是最后一个分量的分量

1. 检查索引节点i\_mode字段的访问模式和运行进程的特权
2. 用路径分量名计算哈希值
3. 如果是“.”,表示当前目录，直接跳过；如果是“..”,表示上一级目录，调用函数follow\_dotdot(nd)回到父目录；否则

-> do\_lookup()//搜索dentry

4) 如果返回的dentry是链接文件dentry，调用do\_follow\_link（）；如果返回的dentry不是目录dentry，返回错误；否则将path中的相关内容转化到nd中:

-> path\_to\_nameidata(&next, nd);

继续下一个分量的分析

//对于路径的最后一个分量

5）如果设置了LOOKUP\_PARENT标志位，查找到最后一个分量的前一个就可以了，此时查找操作完成。根据最后一个分量是“.”还是“.”,还是其他,设置nd->last\_type

6）如果设置了LOOKUP\_PARENT标志位，执行过程2）、3），如果找到的dentry是目录dentry，返回错误；如果返回的dentry是链接文件dentry，调用do\_follow\_link（）；否则调用

->path\_to\_nameidata(&next, nd);

函数返回；

do\_lookup()

主要流程：

1)若底层文件系统自定义哈希函数，则重新计算哈希值

2）在目录高速缓存中查找：

-> \_\_d\_lookup()//搜索目录高速缓存，dcache接口

3)若找到,设置path的mnt和dentry域,调用\_\_follow\_mount(path)函数，检查刚解析的分量是否指向某个系统安装点的目录，使之指向挂载在这个目录下的最上层文件系统的目录对象和已安装文件系统对象。函数返回。

4）如果没找到：

parent = nd->path.dentry;//父目录dentry

dir = parent->d\_inode;//父目录的inode

查找我们要找的dentry是否在等待目录锁信号期间已经被创建d\_lookup(parent, name)；

5）如果依然没找到，分配dentry并初始化struct dentry \*new= d\_alloc(parent, name);并且调用具体文件系统的lookup函数进行查找：

dentry = dir->i\_op->lookup(dir, new, nd);

simple\_lookup()

Const struct inode\_operations ramfs\_dir\_inode\_operations = {

……

.create = ramfs\_create,

.lookup = simple\_lookup,

.……,

};

do\_filp\_open()

……

->open\_to\_namei\_flags()

//把访问模式标志位拷贝到namei\_flags标志中

……

if (!(flag & O\_CREAT)) {

//如果没有设置O\_CREATE标志位

filp = **get\_empty\_filp()**;

//从file的slab缓冲区filp\_cachep中分配一个file结构

……

->do\_path\_lookup()

//完成路径名中最后一个分量的查找

……

有误退出，无误则进入OK执行

}

//如果设置了O\_CREATE标志位，需要找到父节

点。path\_init为查找作准备工作，path\_walk真正上路查找,这两个函数联合起来查找路径名对应的dentry.

->path\_init(，，，LOOKUP\_PARENT，)

->path\_walk()//到这里，找的是父dentry

-> get\_empty\_filp()

//根据父节点的dentry结构和记录搜索结果的nd

的last结构，可以找到目标文件的dentry结构

//如果找到的dentry为negative状态，创建file

-> lookup\_hash()

if (!path.dentry->d\_inode) {

……

-> \_\_open\_namei\_create()

-> nameidata\_to\_filp()

返回filp;}

//OK：若找到的dentry正常，即要打开的文件存在

-> path\_to\_nameidata ()//path转换成nd

->may\_open()//检查文件的打开权限

->nameidata\_to\_filp()

->\_\_dentry\_open()

//将nd转换成filp对象返回filp}

Note: O\_CREARE标志表示如果文件不存在则创建

/

lookup\_hash()

-> \_\_lookup\_hash();

\_\_open\_namei\_create()

-> vfs\_create()

//调用具体fs创建inode的方法

dir->i\_op->create(dir, dentry, mode, nd)

->may\_open()

//检查文件的打开权限

\_\_lookup\_hash ()

1)在高速缓存中搜索

-> \_\_d\_lookup()//开锁查询

2)若没找到,检查开锁期间要找的dentry是否建立

if (!dentry)

dentry = d\_lookup(base, name);

3）如果仍返回空值，则新建dentry

struct dentry \*new= d\_alloc(base, name);

dentry = inode->i\_op->lookup(inode, new, nd);

4）返回dentry

ramfs\_create()

**Ramfs的create过程**

**simple\_lookup（）**

static const struct dentry\_operations simple\_dentry\_operations = {

.d\_delete = simple\_delete\_dentry,

};

if (dentry->d\_name.len > NAME\_MAX)

return ERR\_PTR(-ENAMETOOLONG);

dentry->d\_op = &simple\_dentry\_operations;

d\_add(dentry, NULL);

return NULL;

**simple\_delete\_dentry（）**

->return 1

如果lookup没有在目录项高速缓存中找到，说明这个目录项处于负状态，将d\_op置为删除操作，删除即可

**ramfs\_create（）**

->ramfs\_mknod(dir, dentry, mode | S\_IFREG, 0)

//在指定目录下创建文件

**ramfs\_mknod（）**

->ramfs\_get\_inode（）

//获取一个inode

……

if (inode) {

if (dir->i\_mode & S\_ISGID) {

inode->i\_gid = dir->i\_gid;

if (S\_ISDIR(mode))

inode->i\_mode |= S\_ISGID;

}

//如果mode带有GID，需要将GID传递给inode

-> d\_instantiate（）

//用于向dentry结构中填写inode信息

-> dget（）

//对dentry->d\_count加1

dir->i\_mtime = dir->i\_ctime = CURRENT\_TIME

//修改目录的访问时间、inode的修改时间

read（）和write（）系统调用非常相似，都需要三个参数：一个文件描述符fd，一个内存区的地址buf，以及一个数count（指定应该传送多少字节）。read（）把数据从文件传送到缓冲区，而write（）执行相反的操作。两个系统调用都返回成功传送的字节数，或者发送一个错误条件信号返回-1。

**ramfs\_get\_inode（）**

->new\_inode（）

//从内存中分配一个inode空间，调用kmem\_cache\_alloc函数

if（inode）{ //填充inode结构

inode->i\_mode = mode;

inode->i\_uid = current\_fsuid();

inode->i\_gid = current\_fsgid();

inode->i\_mapping->a\_ops = &ramfs\_aops;

inode->i\_mapping->backing\_dev\_info = &ramfs\_backing\_dev\_info;

mapping\_set\_gfp\_mask(inode->i\_mapping, GFP\_HIGHUSER);

**mapping\_set\_unevictable(inode->i\_mapping);**

inode->i\_atime = inode->i\_mtime = inode->i\_ctime = CURRENT\_TIME;

switch (mode & S\_IFMT) {

default: //处理特殊的inode，包括socket、fifo、块设备、字符设备

init\_special\_inode(inode, mode, dev);

break;

case S\_IFREG: //普通文件

inode->i\_op = &ramfs\_file\_inode\_operations;

inode->i\_fop = &ramfs\_file\_operations;

break;

case S\_IFDIR: //目录

inode->i\_op = &ramfs\_dir\_inode\_operations;

inode->i\_fop = &simple\_dir\_operations;

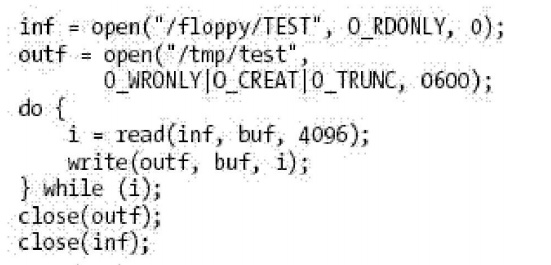
inc\_nlink(inode);

break;

case S\_IFLNK: 链接

inode->i\_op = &page\_symlink\_inode\_operations;

break;



考虑这个例子，用户发出了一条shell命令：把/floppy/TSET中的MS-DOS文件拷贝到/tmp/test中的ext2文件系统中，命令shell调用一个外部程序（如cp），我们假定cp执行上面的代码段

**VFS的read过程**

**generic\_file\_aio\_read()**

//generic\_file\_aio\_read()是所有文件系统实现同步和异步读操作所使用的通用例程

若设置了direct\_IO标志位，则绕过page cache，数据直接在用户地址空间的缓冲区和磁盘之间直接进行传输，这一般用于用户程序有自己缓存的方式，例如数据库等；注意到ramfs是基于page cache的内存文件系统，因此都是采用的page cache方式，都会执行do\_generic\_file\_read（）函数

//generic\_file\_aio\_read()是所有文件系统实现同步和异步读操作所使用的通用例程

若设置了direct\_IO标志位，则绕过page cache，数据直接在用户地址空间的缓冲区和磁盘之间直接进行传输，这一般用于用户程序有自己缓存的方式，例如数据库等；但是最常用的情况还是page cache方式.

**sys\_read ()**

->fget\_light()

//获取file对象

->file\_pos\_read()

//读取file对象的f->pos,并保持

->vfs\_read()

->file\_pos\_write()

//恢复file对象的f->pos域

->fput\_light()

//释放file对象

//读操作从read系统调用开始

**vfs\_read()**

1. 检查f->mode标志是否允许读权限，如果不允许，返回一个错误码-EBADF
2. 检查file是否有read()或aio\_read()操作，如果没有，则返回一个错误码-EINVAL
3. 调用access\_ok()粗略检查buf和count参数
4. 调用rw\_verify\_area()检查要访问的文件部分是否有冲突的强制锁，若有返回错误码
5. 调用底层文件系统自定义读操作方法：

file->f\_op->read()

//ramfs\_file\_operations->.read = do\_sync\_read,

1. 若底层文件系统没有自定义读操作方法，默认调用do\_sync\_read()
2. 文件系统读操作报告给notify系统

->fsnotify\_access()

增加任务操作字符计数add\_rchar()

8) 增加任务系统调用计数inc\_syscr()

9) 函数返回

do\_generic\_file\_read()

/\*函数思想：从文件指针\*ppos导出第一个请求字节所在页的逻辑号（地址空间中的页索引），并把它放在index中，把第一个请求字节在页内的偏移量放在offset局部变量中\*/

for(;;){

**find\_page**:

->find\_get\_page()//查找高速缓存以找到包含所请求数据的页描述

->若没找到，跳转**no\_cached\_page**；若找到，查看PG\_uptodate标志位。

->若PG\_uptodate标志位没置位，表示page不是最新，跳转**page\_not\_up\_to\_date**;否则继续

**page\_ok**:

->检查index是否超出文件的总页数（这种情况出现在正在读的文件已被其他进程删减的时候）

->计算应被拷贝入用户缓冲区的的页中字节数

->将PG\_referenced或PG\_active置位，从而表示该页正在访且不被换出

->将页中的数据拷贝到用户缓冲区，通过函数指针调用函数执行->file\_read\_actor ()

->查看ret == nr && desc->count，为真表示尚未读完，进入下一循环；否则跳转**out**

**page\_not\_up\_to\_date**:

->获取对页的互斥访问，成功则继续执行；失败则跳转**readpage\_error**

**page\_not\_up\_to\_date\_locked**:

->如果这个页已经被其他进程从page cache删除了；返回到循环开始出执行，否则继续

->查看PG\_uptodate标志位，若置位，表示其他进程已经读入了这个page，跳转**page\_ok**；否则**readpage**

**readpage**:

->调用底层文件系统的方法从磁盘读取数据mapping->a\_ops->readpage(filp, page);

->若读的过程中出现错误，跳转**readpage\_error;**等到PG\_uptodate被置位，解锁，跳转**page\_ok**

**readpage\_error**:

->在读操作描述符中报告错误，跳转out

**no\_cached\_page**:

->page\_cache\_alloc\_cold()//从伙伴系统分配一个页，并加入address\_space对应的基数树中

->add\_to\_page\_cache\_lru()//把新分配的页加入到LRU链中

->若上两个过程无误，跳转**readpage**；有误则结束循环，跳转到**out**

**out**：

->更新数据结构flip->ra来标记数据已经被顺序读入

->file\_accessed()//把当前时间放在inode的i\_actime字段，把它标记为脏并返回

}

do\_sync\_read()

1. struct iovec iov记录用户空间地址和长度；

struct kiocb kiocb记录IO完成状态，并初始化

2）采用异步读来完成同步读操作

-> filp->f\_op->aio\_read（）

**//**ramfs\_file\_operations->.aio\_read = generic\_file\_aio\_read

3）异步操作需要等待

-> wait\_on\_retry\_sync\_kiocb()

4)更新pos参数，返回

file\_read\_actor ()

->kmap()//为处于高端内存中的页建立永久的内核映射

->\_\_copy\_to\_user() //把页中的数据拷贝到用户地址空间

->kunmap() //释放页的任一永久内核映射

->更新read\_descripter\_t描述符count、written等字段

**const struct address\_space\_operations**

**ramfs\_address\_space\_operations** = {

.readpage = simple\_readpage,

……

};

如果page的内容已经是最新，直接返回就可以；如果不是，读取这个页的数据之前，将cache的内容刷回这个page，保证数据一致性，然后返回这个页的内容，将page标志位设置为最新

**simple\_ readpage ()**

clear\_highpage(page);

flush\_dcache\_page(page);

SetPageUptodate(page);

unlock\_page(page);

return 0;

**VFS的write过程**

**generic\_file\_aio\_write() or generic\_file\_write\_iter()**

->mutex\_lock（）

//对索引节点信号量上锁

->\_\_generic\_file\_aio\_write（）

//调用\_\_generic\_file\_aio\_write函数完成异步写操作的执行

->mutex\_unlock（）

//解锁

->generic\_write\_sync（）

//如果异步写成功，则执行写同步

**sys\_write ()**

->fget\_light()

//获取file对象

->file\_pos\_read()

//读取file对象的f->pos,并保持

->vfs\_write()

->file\_pos\_write()

//恢复file对象的f->pos域

->fput\_light()

//释放file对象

//写操作从write系统调用开始，过程和sys\_read完全类似

**\_\_generic\_file\_aio\_write（）**

->generic\_segment\_checks（）

//在写操作之前做必要的检查工作

->generic\_write\_checks（）

//详细检查写操作

if（file->f\_flags & O\_DIRECT）

//判断是否为直接I/O，如果是则绕过page cache，数据直接在用户地址空间的缓冲区和磁盘之间直接进行传输

else generic\_file\_buffered\_write（）

//对于ramfs文件系统来说，不考虑直接I/O，直接跳转到else执行generic\_file\_buffered\_write（）函数，在缓存中执行写操作

**vfs\_write()**

1. 检查f->mode标志是否允许写权限，如果不允许，返回一个错误码-EBADF
2. 检查file是否有write()或aio\_write()操作，如果没有，则返回一个错误码-EINVAL
3. 调用access\_ok()粗略检查buf和count参数
4. 调用rw\_verify\_area()检查要访问的文件部分是否有冲突的强制锁，若有返回错误码
5. 调用底层文件系统自定义写操作方法：

file->f\_op->write()

//ramfs\_file\_operations->.write= do\_sync\_write,

1. 若底层文件系统没有自定义写操作方法，默认调用do\_sync\_write()

7）文件系统写操作报告给notify系统

->fsnotify\_access()

8） 增加任务操作字符计数add\_wchar()

9) 增加任务系统调用计数inc\_syscw()

10) 函数返回

**generic\_file\_buffered\_write（）**

-> generic\_perform\_write（）

//调用generic\_perform\_write（）函数完成在高速缓存里的写操作

if (likely(status >= 0)) {

written += status;

\*ppos = pos + status;

}

//如果写操作成功，返回的status表示写的字节数，更新written且作为函数的返回值，并更新指针ppos

**do\_sync\_write()**

1. struct iovec iov记录用户空间地址和长度；

struct kiocb kiocb记录IO完成状态，并初始化

2）采用异步写来完成同步写操作

-> filp->f\_op->aio\_write（）

**//**ramfs\_file\_operations->.aio\_write= generic\_file\_aio\_write

3）异步操作需要等待

-> wait\_on\_retry\_sync\_kiocb()

4）更新pos参数，返回

**generic\_perform\_write（）**

1. 开始执行一个循环，只要count+written的值不为0，即要写的数据没有写完。从参数pos（文件的指针）导出第一个请求字节所在的逻辑页号，存放在index变量里，页内偏移保存在offset变量里，要写的字节数保存在bytes中
2. 检查要写入的page是否允许被访问，如果不允许则终止循环返回一个错误值-EFAULT
3. 调用底层文件系统的地址空间操作方法，准备从用户缓冲区开始写文件页块：mapping->a\_ops->write\_begin
4. 如果mapping的i\_mmap\_writable字段（地址空间中共享内存映射的个数）不为0，即在用户空间页面被修改过，则把该data cache page写回到内存
5. 调用iov\_iter\_copy\_from\_user\_atomic函数把缓冲区的数据拷贝到page中，注意为了防止缺页，在拷贝前调用pagefault\_disable()函数，拷贝完毕后调用pagefault\_enable()函数。然后调用flush\_dcache\_page函数把CPU L2 cache 中内容写回到该page，使得page内容保持为最新。
6. 调用mark\_page\_accessed（）函数把该page标记为已访问过
7. 调用底层文件系统的地址空间操作方法，从用户缓冲区写文件页块完成：mapping->a\_ops->write\_end
8. 将write\_end函数的返回值传递给copied，即实际完成写操作的字节数，并调用cond\_resched（）检查当前进程的TIF\_NEED\_RESCHED标志，如果置位则调用schedule（）函数
9. 调用iov\_iter\_advance函数，更新用户缓冲区地址，并减少要写入自己数的值：i->iov\_offset += copied；i->count -= copied
10. 如果copied的值为0，即未完成写入操作，则将要写入的字节数bytes设置为一个单独的iov\_iter segment大小，跳转至3）
11. 更新pos的值，让它指向最后一个被写入的字符之后的位置，然后调用balance\_dirty\_pages\_ratelimited（）函数检查脏页是否需要写回，最后返回完成写操作字节数

**const struct address\_space\_operations ramfs\_aops** = {

.readpage = simple\_readpage,

.write\_begin = simple\_write\_begin,

.write\_end = simple\_write\_end,

.set\_page\_dirty = \_\_set\_page\_dirty\_no\_writeback,

};

**simple\_write\_end（）**

（1）如果实际完成写入的长度copied小于应该写入的长度，则把其他剩余的字段填充为0，然后调用flush\_dcache\_page把data cache page写回到内存中

（2）调用simple\_commit\_write

（3）写操作完成，解锁page并减少page的引用次数，返回实际完成的写入字节数copied

**simple\_write\_begin（）**

（1）从参数pos（文件的指针）导出第一个请求字节所在的逻辑页号，存放在index变量里，页内偏移保存在from变量里

（2）调用grab\_cache\_page\_write\_begin

//到pagecache中寻找index对应的页面，如果不存在则新建一个。如果调用失败，则返回一个错误值-ENOMEM

（3）调用simple\_prepare\_write

//返回simple\_prepare\_write的值作为simple\_write\_begin函数的最终返回值

**simple\_commit\_write（）**

（1）如果page不是最新的，置为最新

（2）如果文件写入终止位置大于原来的尺寸inode->i\_size，则更新i\_size

（3）将page置为脏：set\_page\_dirty

**grab\_cache\_page\_write\_begin（）**

（1）调用find\_lock\_page()函数，搜索index对应的page是否在mapping指向的页高速缓存里，如果找到了就调用lock\_page函数锁住页面。执行完毕后，如果page不为NULL就返回page

（2）如果page为NULL，即不在页高速缓存里，则调用\_\_page\_cache\_alloc函数创建一个

（3）调用add\_to\_page\_cache\_lru函数把新建的page添加到LRU链表里，如果失败则返回NULL

（4）返回page

**simple\_prepare\_write（）**

（1）如果page不是最新，且如果要写的字节数bytes不等于page cache size，则把page其他字段内容全部清零

（2）返回0