# VFS 文件打开&&路径名解析

## 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.

## 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：

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

## 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
};
```

## VFS 的 open 过程

### 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

### 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

### 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

### __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

### ramfs_create()

## VFS 路径名解析

### path_lookup()

->do_path_lookup()

### do_path_lookup()

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

### 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

### 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);

### lookup_hash()

-> __lookup_hash();

### __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

### 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

### 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);

### Const struct inode_operations ramfs_dir_inode_operations = {

……
.create = ramfs_create,
.lookup = simple_lookup,
.……,
};

### simple_lookup()

## Ramfs 的 create 过程

ramfs_create（）

->ramfs_mknod(dir, dentry, mode |

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

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;

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

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

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

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

## 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,
6) 若底层文件系统没有自定义读操作方法，默认调用 do_sync_read()
7) 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_sync_read()

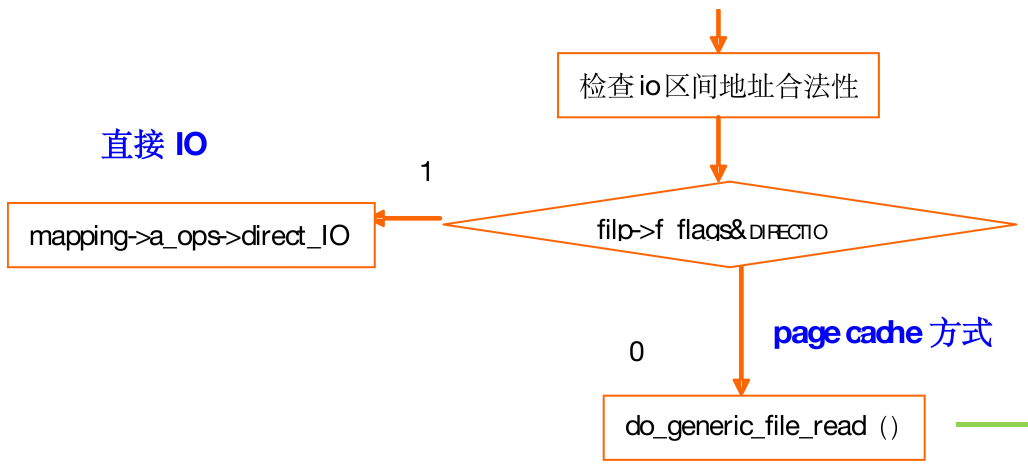
1) struct iovec iov 记录用户空间地址和长度；
   struct kiocb kiocb 记录 IO 完成状态，并初始化
2）采用异步读来完成同步读操作
   -> filp->f_op->aio_read（）
//ramfs_file_operations->.aio_read = generic_file_aio_read
3）异步操作需要等待
   -> wait_on_retry_sync_kiocb()

## 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

```
const struct address_space_operations
 ramfs_address_space_operations = {
      .readpage = simple_readpage,
      ……
};
```

## 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

## 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
`

## simple_ readpage ()

```
clear_highpage(page);
flush_dcache_page(page);
SetPageUptodate(page);
unlock_page(page);
return 0;
```

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

# VFS 的 write 过程

## 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

## 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,
6) 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

## 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()

```
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,
};
```

## 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_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

## 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

## 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

## 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

## 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

## 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_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