

Linux Virtual File System

Wu Chia-Chih

Linux Kernel Trace Seminar

2002.6.3

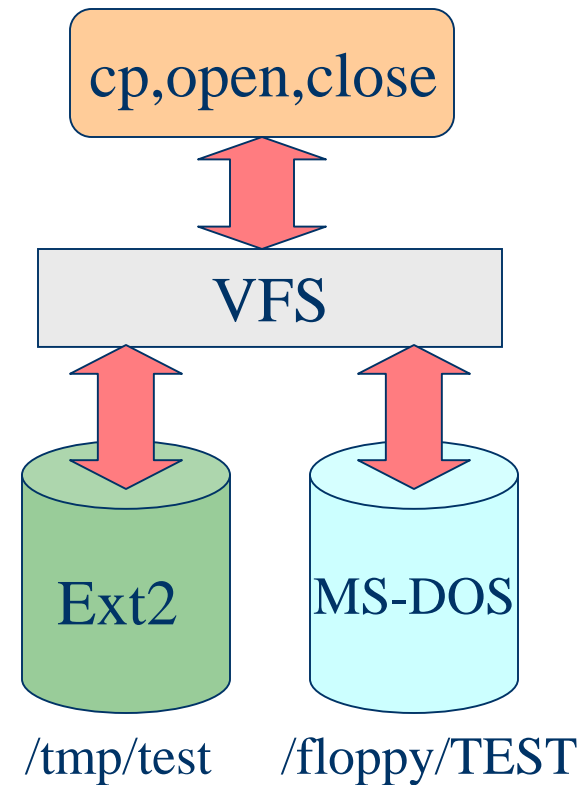


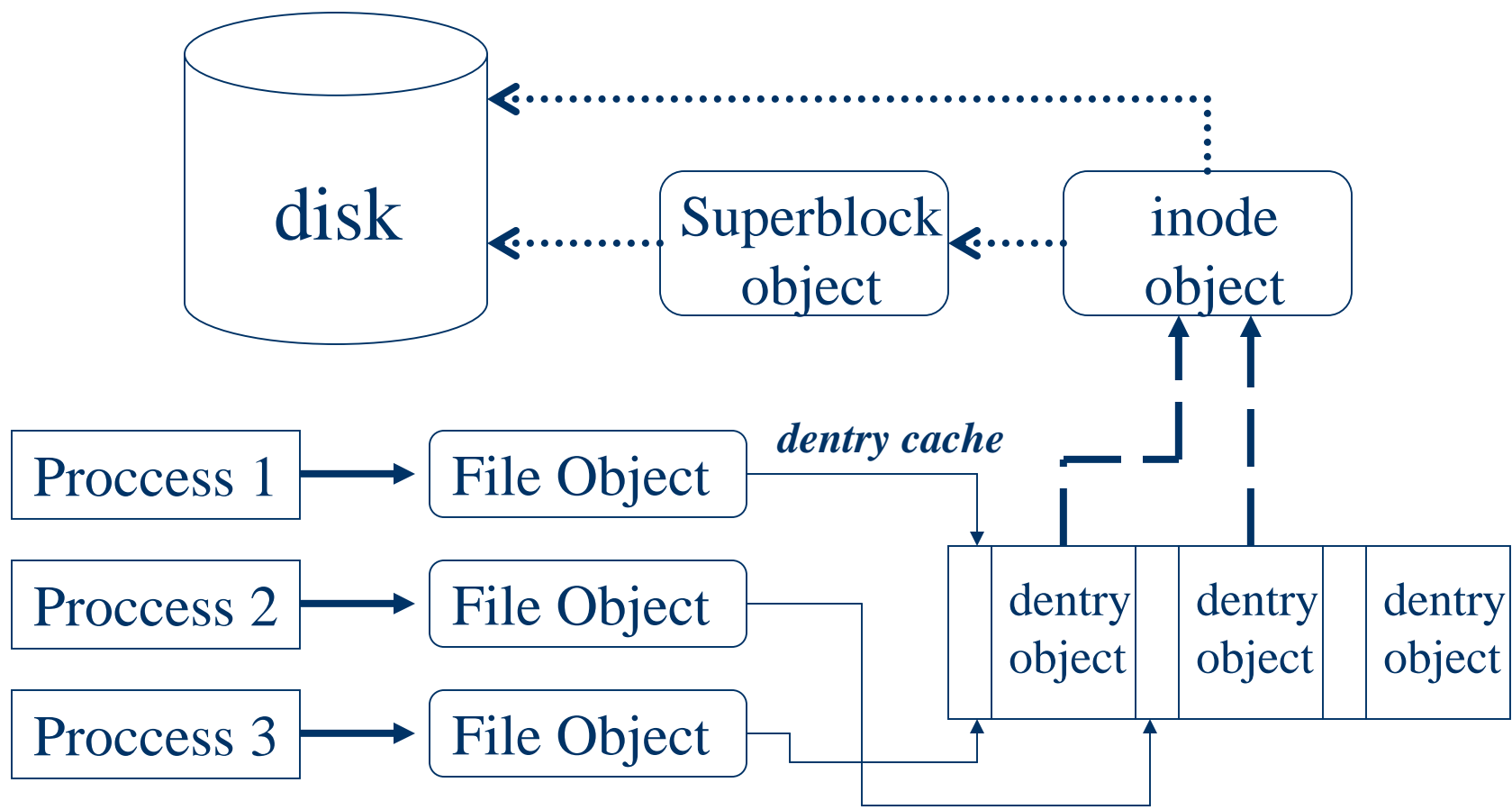
Outline

- | Introduction
- | Open a file
- | Mount filesystem
- | Write a new filesystem

Introduction

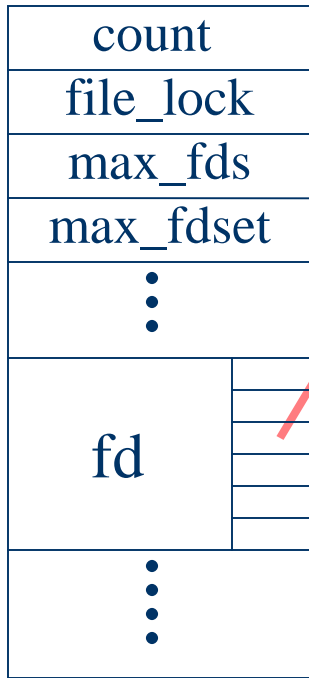
- Manages kernel level file abstractions in one format for all file systems
- Receives system calls from user level(e.g. write, open, close)
- Interacts with a specific file system based on mount point traversal



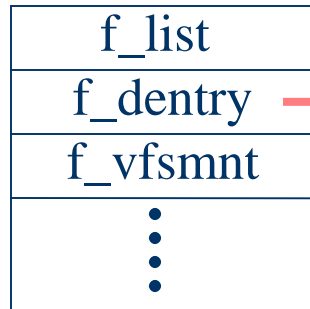


- fd* \longrightarrow
- f_dentr* \longrightarrow
- f_inode* \longrightarrow
- i_sb* $\cdots\longrightarrow$

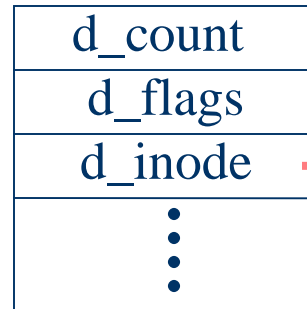
struct files_struct



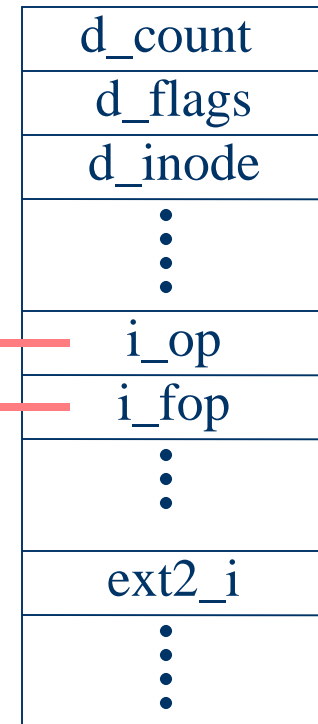
struct file



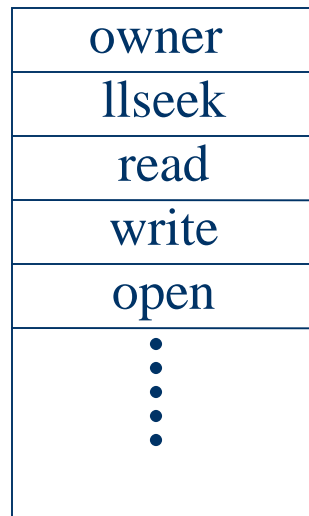
struct dentry



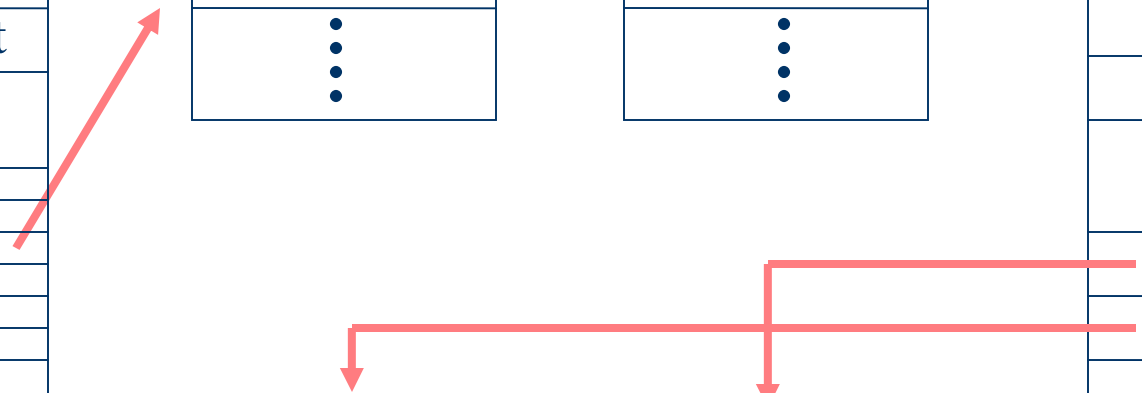
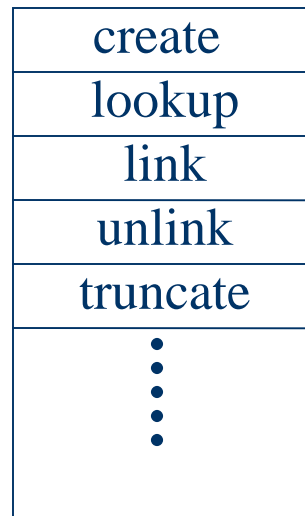
struct inode



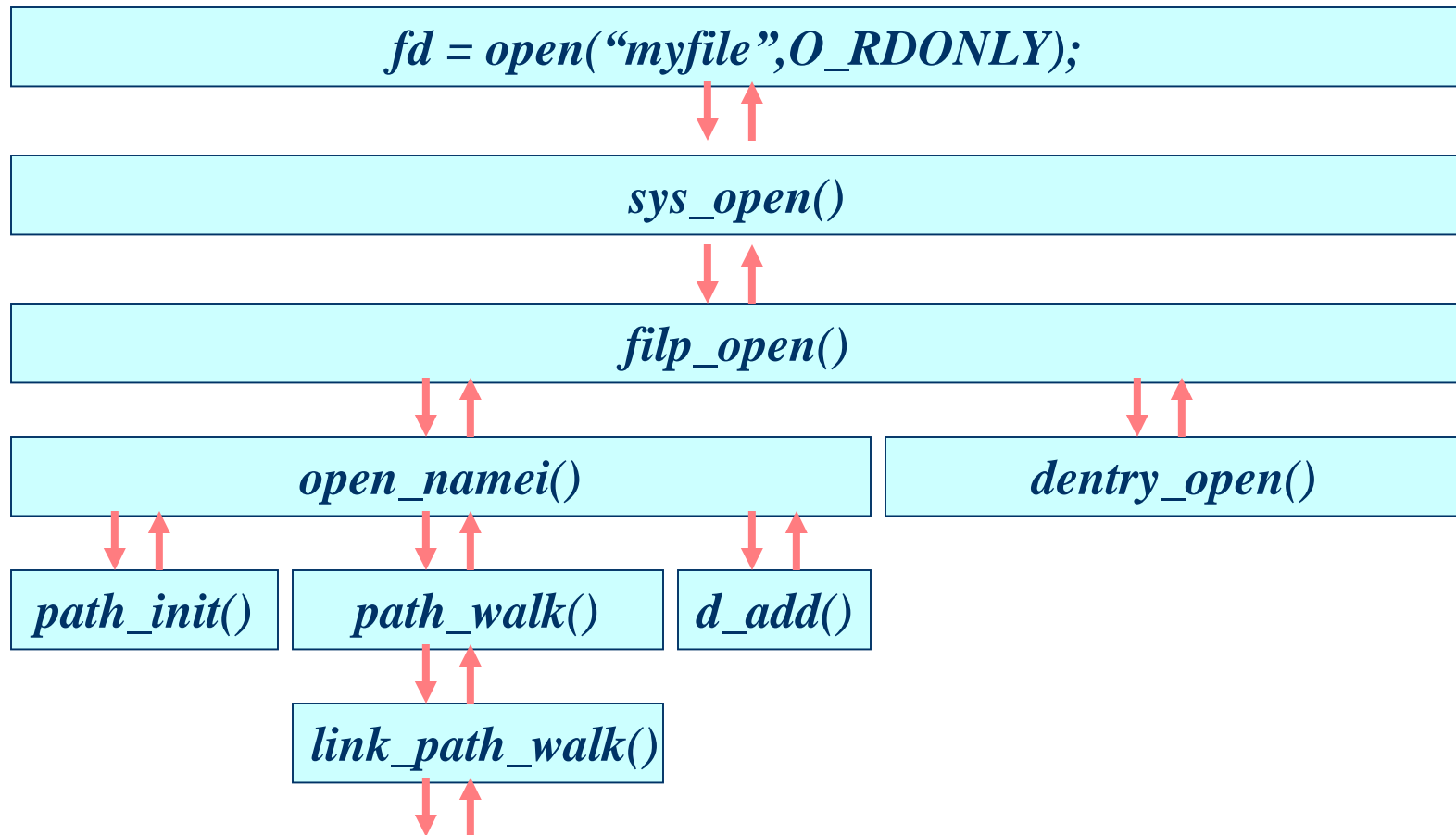
struct files_operations



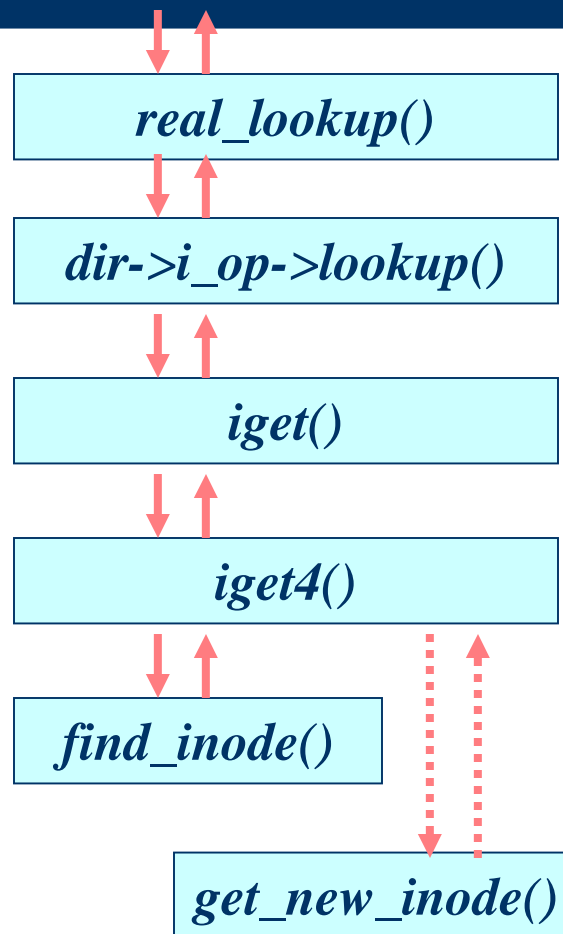
struct inode_operations



Opening a ext2 file via VFS



Opening a ext2 file via VFS(cont')



sys_open()

```
if (!IS_ERR(tmp)) {  
    fd = get_unused_fd();  
    if (fd >= 0) {  
        struct file *f = filp_open(tmp, flags, mode);  
        error = PTR_ERR(f);  
        if (IS_ERR(f))  
            goto out_error;  
        fd_install(fd, f);  
    }  
    .....
```

struct file *filp_open(*const char * filename, int flags, int mode*)

```
{  
    int namei_flags, error;  
    struct nameidata nd;  
    .....  
    error = open_namei(filename, namei_flags, mode, &nd);  
    if (!error)  
        return dentry_open(nd.dentry, nd.mnt, flags);  
  
    return ERR_PTR(error);  
}
```

struct nameidata

```
{  
    struct dentry *dentry;  
    struct vfsmount *mnt;  
    struct qstr last;  
    unsigned int flags;  
    int last_type;  
}
```

open_namei()

I Aims

- Fills the nameidata structure containing dentry and vfsmount structures

I Functions it used

- Call path_init() and path_walk() to walk through the path and find the “parent directory dentry” dir
- Do iget(dir->i_sb,ino) to get the corresponding inode
- Instantiate the dentry by d_add(dentry,inode)

open_namei()

```
if (!(flag & O_CREAT)) {  
    if (path_init(pathname, lookup_flags(flag), nd))  
        error = path_walk(pathname, nd);  
  
    if (error)  
        return error;  
  
    dentry = nd->dentry;  
    goto ok;  
}
```

path_init()

/*initialize the path walk*/

```
{  
    nd->last_type = LAST_ROOT; /* if there are only slashes... */  
    nd->flags = flags;  
    if (*name=='/')  
        return walk_init_root(name,nd);  
    read_lock(&current->fs->lock);  
    nd->mnt = mntget(current->fs->pwdmnt);  
    nd->dentry = dget(current->fs->pwd);  
    read_unlock(&current->fs->lock);  
    return 1;  
}
```

path_walk()

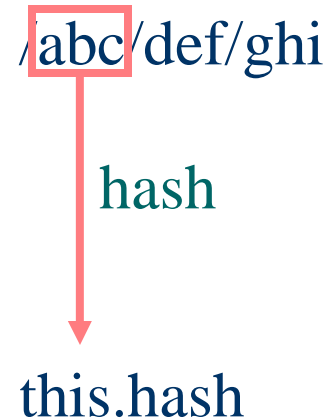
```
/*call link_path_walk()*/
```

```
{  
    current->total_link_count = 0;  
    return link_path_walk(name, nd);  
}
```

link_path_walk()

/* do the real walk */

```
for(;;){  
    ....  
    this.name = name;  
    c = *(const unsigned char *)name;  
    hash = init_name_hash();  
    do {  
        name++;  
        hash = partial_name_hash(c, hash);  
        c = *(const unsigned char *)name;  
    } while (c && (c != '/'));  
    this.len = name - (const char *) this.name;  
    this.hash = end_name_hash(hash);
```



link_path_walk()

```
/* do the real walk */
```

```
/*  
 * See if the low-level filesystem might want  
 * to use its own hash..  
 */  
if (nd->dentry->d_op && nd->dentry->d_op->d_hash) {  
    err = nd->dentry->d_op->d_hash(nd->dentry, &this);  
    if (err < 0)  
        break;  
}
```

link_path_walk()

```
/* do the real walk */
```

```
/* This does the actual lookups.. */
```

```
dentry = cached_lookup(nd->dentry, &this, LOOKUP_CONTINUE);
```

```
if (!dentry) {
```

```
    dentry = real_lookup(nd->dentry, &this, LOOKUP_CONTINUE);
```

```
    err = PTR_ERR(dentry);
```

```
    if (IS_ERR(dentry))
```

```
        break;
```

```
}
```

```
/* Check mountpoints.. */
```

```
while (d_mountpoint(dentry) && __follow_down(&nd->mnt, &dentry))
```

```
    ;
```

link_path_walk()

```
/* do the real walk */
```

```
if (inode->i_op->follow_link) {  
    .....  
} else {  
    dput(nd->dentry);  
    nd->dentry = dentry;  
}  
err = -ENOTDIR;  
if (!inode->i_op->lookup)  
    break;  
continue;  
/* here ends the main loop */
```

real_lookup()

Lookup the cache

```
down(&dir->i_sem);
result = d_lookup(parent, name);
if (!result) {
    struct dentry * dentry = d_alloc(parent, name);
    result = ERR_PTR(-ENOMEM);
    .....
    result = dir->i_op->lookup(dir, dentry);
    .....
    up(&dir->i_sem);
}
```

dir->i_op->lookup()

```
struct inode_operations ext2_dir_inode_operations = {  
    create:          ext2_create,  
    lookup:         ext2_lookup,  
    link:           ext2_link,  
    unlink:        ext2_unlink,  
    symlink:       ext2_symlink,  
    mkdir:         ext2_mkdir,  
    rmdir:         ext2_rmdir,  
    mknod:         ext2_mknod,  
    rename:        ext2_rename,  
};
```

ext2_lookup()

```
struct inode* inode;
ino_t ino;
.....
ino = ext2_inode_by_name(dir, dentry);
inode = NULL;
if (ino) {
    inode = iget(dir->i_sb, ino);
    if (!inode)
        return ERR_PTR(-EACCES);
}
d_add(dentry, inode);
```

```
static inline struct inode *iget(struct  
super_block *sb, unsigned long ino)
```

```
{
```

```
    return iget4(sb, ino, NULL, NULL);
```

```
}
```

```
struct inode *iget4(struct super_block *sb,  
unsigned long ino, find_inode_t find_actor, void  
*opaque)
```

```
struct list_head * head = inode_hashtable + hash(sb,ino);  
  
struct inode * inode;  
spin_lock(&inode_lock);  
  
inode = find_inode(sb, ino, head, find_actor, opaque);  
if (inode) {  
    __iget(inode);  
    spin_unlock(&inode_lock);  
    wait_on_inode(inode);  
    return inode;  
}  
spin_unlock(&inode_lock);  
  
return get_new_inode(sb, ino, head, find_actor, opaque);
```

Wait if other one
locked the inode

Why `head = inode_hashtable + hash(sb,ino)` ?

Available
physical pages

```
.....  
mempages >>= (14 - PAGE_SHIFT);  
mempages *= sizeof(struct list_head);  
for (order = 0; ((1UL << order) << PAGE_SHIFT) < mempages; order++)  
    ;  
do {  
    unsigned long tmp;  
  
    nr_hash = (1UL << order) * PAGE_SIZE /  
    sizeof(struct list_head);  
    i_hash_mask = (nr_hash - 1);
```

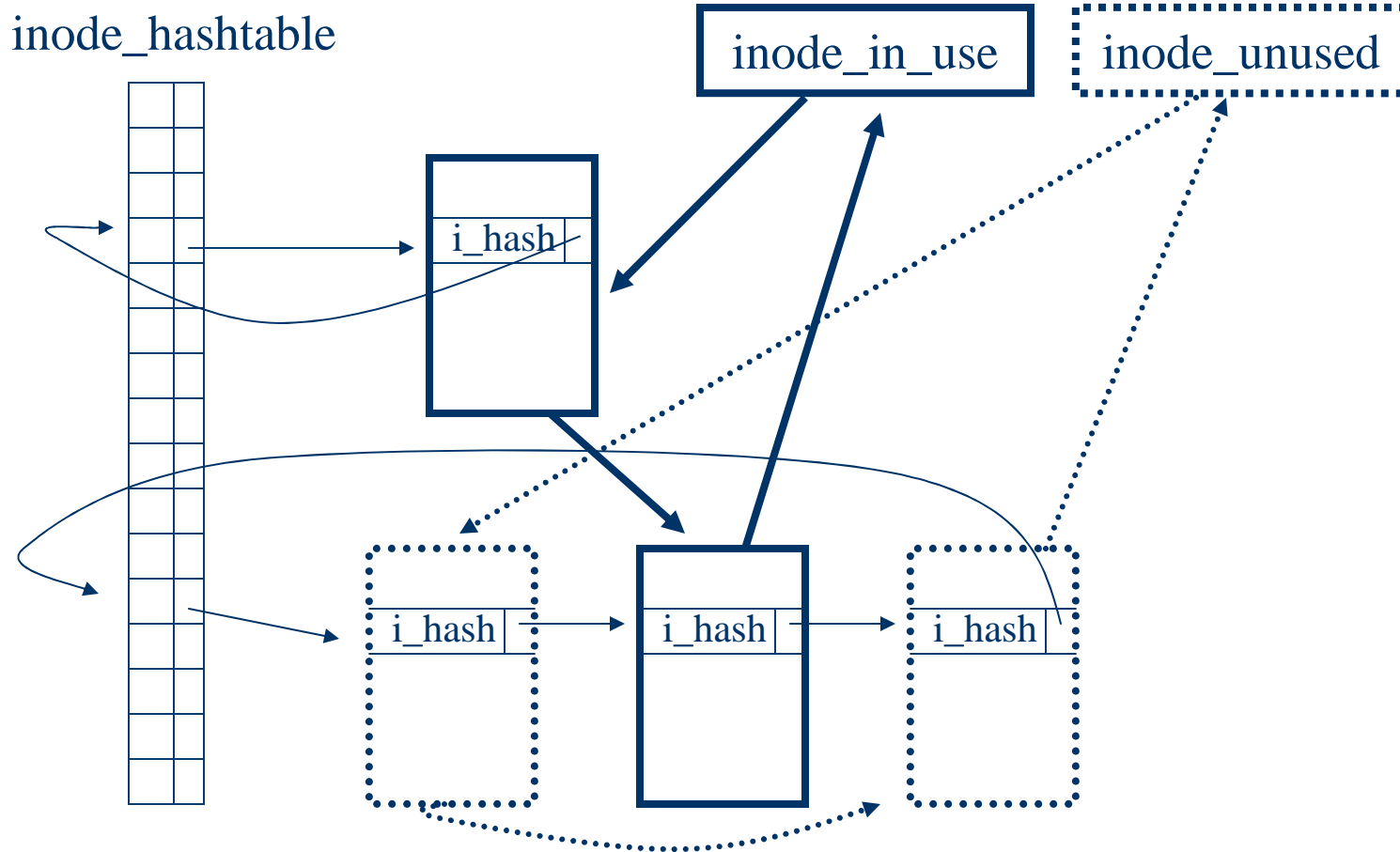
Ans : inode_hashtable is a ptr to a block of contiguous memory

```
tmp = nr_hash;
i_hash_shift = 0;
while ((tmp >>= 1UL) != 0UL)
    i_hash_shift++;

inode_hashtable = (struct list_head *)
    __get_free_pages(GFP_ATOMIC, order);
} while (inode_hashtable == NULL && --order >= 0);

printk("Inode-cache hash table entries: %d (order: %ld, %ld bytes)\n",
nr_hash, order, (PAGE_SIZE << order));
.....
```

inode cache



```
429 struct inode {
430     struct list_head    i_hash;
431     struct list_head    i_list;
432     struct list_head    i_dentry;
433
434     struct list_head    i_dirty_buffers;
435     struct list_head    i_dirty_data_buffers;
436
437     unsigned long        i_ino;
438     atomic_t             i_count;
439     kdev_t               i_dev;
440     umode_t              i_mode;
441     nlink_t              i_nlink;
442     uid_t                i_uid;
443     gid_t                i_gid;
444     kdev_t               i_rdev;
445     loff_t               i_size;
446     .....
```

find_inode()

```
struct list_head *tmp;
struct inode * inode;

tmp = head;
for (;;) {
    tmp = tmp->next;
    inode = NULL;
    if (tmp == head)
        break;
```

find_inode()

```
inode = list_entry(tmp, struct inode, i_hash);
if (inode->i_ino != ino)
    continue;
if (inode->i_sb != sb)
    continue;
if (find_actor && !find_actor(inode, ino, opaque))
    continue;
break;
}
return inode;
```

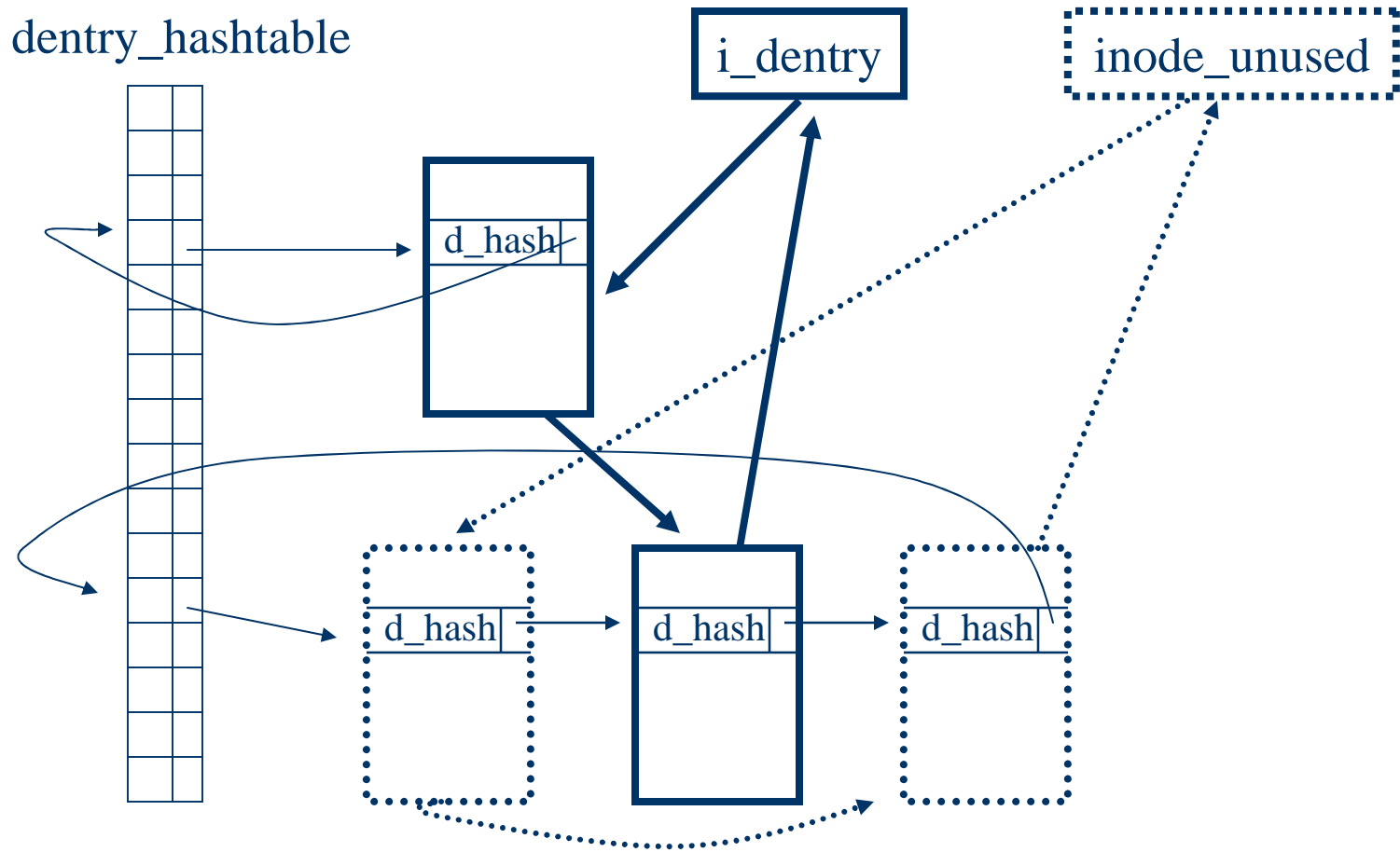
__iget()

```
{  
    if (atomic_read(&inode->i_count)) {  
        atomic_inc(&inode->i_count);  
        return;  
    }  
    atomic_inc(&inode->i_count);  
    if (!(inode->i_state & (I_DIRTY|I_LOCK))) {  
        list_del(&inode->i_list);  
        list_add(&inode->i_list, &inode_in_use);  
    }  
    inodes_stat.nr_unused--;  
}
```

d_add(*entry,inode*)

```
{  
    d_instantiate(entry,inode);  
    d_rehash(entry);  
}
```

dentry cache



```
66 struct dentry {
67     atomic_t d_count;
68     unsigned int d_flags;
69     struct inode * d_inode;      /* Where the name belongs to - NULL is negative */
70     struct dentry * d_parent;    /* parent directory */
71     struct list_head d_hash;    /* lookup hash list */
72     struct list_head d_lru;    /* d_count = 0 LRU list */
73     struct list_head d_child;    /* child of parent list */
74     struct list_head d_subdirs;  /* our children */
75     struct list_head d_alias;  /* inode alias list */
76     int d_mounted;
77     struct qstr d_name;
78     unsigned long d_time;        /* used by d_revalidate */
79     struct dentry_operations *d_op;
80     struct super_block * d_sb;    /* The root of the dentry tree */
81     unsigned long d_vfs_flags;
82     void * d_fsdata;             /* fs-specific data */
83     unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
84 };
```

d_instantiate(*struct dentry* entry, struct inode* inode*)

```
{  
    if (!list_empty(&entry->d_alias)) BUG();  
    spin_lock(&dcache_lock);  
    if (inode)  
        list_add(&entry->d_alias, &inode->i_dentry);  
        entry->d_inode = inode;  
    spin_unlock(&dcache_lock);  
}
```

d_rehash(struct dentry* entry, struct inode* inode)

```
{  
    struct list_head *list = d_hash(entry->d_parent, entry->d_name.hash);  
    if (!list_empty(&entry->d_hash)) BUG();  
    spin_lock(&dcache_lock);  
    list_add(&entry->d_hash, list);  
    spin_unlock(&dcache_lock);  
}
```

dentry_open()

I Aims

- Given a dentry and vfsmount, this function allocates a new struct file and links them together
- Invokes the filesystem specific `f_op->open` when inode was read in `open_namei()`

dentry_open(struct dentry *dentry, struct vfsmount *mnt, int flags)

```
struct file * f;  
struct inode *inode;  
  
.....  
f = get_empty_filp();  
if (!f)  
    goto cleanup_dentry;  
  
f->f_flags = flags;  
f->f_mode = (flags+1) & O_ACCMODE;  
inode = dentry->d_inode;  
  
.....  
f->f_dentry = dentry;  
f->f_vfsmnt = mnt;
```

dentry_open(struct dentry *dentry, struct vfsmount *mnt, int flags)

```
f->f_op = fops_get(inode->i_fop);  
file_move(f, &inode->i_sb->s_files);
```

Add to super
block file list

```
/* preallocate kiobuf for O_DIRECT */  
f->f_iobuf = NULL;  
f->f_iobuf_lock = 0;  
if (f->f_flags & O_DIRECT) {  
    error = alloc_kiovec(1, &f->f_iobuf);  
    if (error)  
        goto cleanup_all;  
}
```

dentry_open(struct dentry *dentry, struct vfsmount *mnt, int flags)

```
if (f->f_op && f->f_op->open) {  
    error = f->f_op->open(inode,f);  
    if (error)  
        goto cleanup_all;  
}  
f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);  
  
return f;
```

Mounting a File System via VFS

```
mount /dev/hdb1 /mnt -t ntfs -o ...
```

dev_name, dir_name, type, flag, data

sys_mount()

dev_page, dir_page, type_page, flags, data_page

do_mount()

&nd, type_page, flags, mnt_flags, dev_name, data_page

do_add_mount()

type, flags, name, data

struct vfsmnt* mnt

do_kern_mount()

asmlinkage long sys_mount(*char * dev_name, char * dir_name, char * type, unsigned long flags, void * data*)

```
.....  
retval = copy_mount_options (type, &type_page);  
if (retval < 0)  
    return retval;  
  
.....  
retval = copy_mount_options (dev_name, &dev_page);  
if (retval < 0)  
    goto out2;  
retval = copy_mount_options (data, &data_page);  
if (retval < 0)  
    goto out3;  
  
.....
```

asmlinkage long sys_mount(*char * dev_name, char * dir_name, char * type, unsigned long flags, void * data*)

```
.....  
lock_kernel();  
retval = do_mount((char*)dev_page, dir_page, (char*)type_page, flags,  
(void*)data_page);  
unlock_kernel();  
.....
```

long do_mount(*char * dev_name, char * dir_name, char *type_page, unsigned long flags, void *data_page*)

```
struct nameidata nd;
.....
/* Separate the per-mountpoint flags */
if (flags & MS_NOSUID)
    mnt_flags |= MNT_NOSUID;
if (flags & MS_NODEV)
    mnt_flags |= MNT_NODEV;
if (flags & MS_NOEXEC)
    mnt_flags |= MNT_NOEXEC;
flags &= ~(MS_NOSUID|MS_NOEXEC|MS_NODEV);

/* ... and get the mountpoint */
if (path_init(dir_name, LOOKUP_FOLLOW|LOOKUP_POSITIVE, &nd))
    retval = path_walk(dir_name, &nd);
```

long do_mount(*char * dev_name, char *
dir_name, char *type_page, unsigned long flags, void
data_page)

```
if (retval)
    return retval;

.....

if (flags & MS_REMOUNT)
    retval = do_remount(&nd, flags & ~MS_REMOUNT, mnt_flags, data_page);
else if (flags & MS_BIND)
    retval = do_loopback(&nd, dev_name, flags & MS_REC);
else if (flags & MS_MOVE)
    retval = do_move_mount(&nd, dev_name);
else
    retval = do_add_mount(&nd, type_page, flags, mnt_flags, dev_name, data_page);
path_release(&nd);
return retval;
```

```
static int do_add_mount(struct nameidata
*nd, char *type, int flags, int mnt_flags, char *name,
void *data)
```

```
struct vfsmount *mnt = do_kern_mount(type, flags, name, data);
```

```
int err = PTR_ERR(mnt);
```

```
if (IS_ERR(mnt))
```

```
    goto out;
```

```
down(&mount_sem);
```

```
/* Something was mounted here while we slept */
```

```
while(d_mountpoint(nd->dentry) && follow_down(&nd->mnt, &nd->dentry))
```

```
    ;
```

```
.....
```

```
static int do_add_mount(struct nameidata
*nd, char *type, int flags, int mnt_flags, char *name,
void *data)
```

```
/* Refuse the same filesystem on the same mount point */
```

```
err = -EBUSY;
```

```
if (nd->mnt->mnt_sb == mnt->mnt_sb && nd->mnt->mnt_root == nd->dentry)
```

```
    goto unlock;
```

```
mnt->mnt_flags = mnt_flags;
```

```
err = graft_tree(mnt, nd);
```

```
unlock:
```

```
up(&mount_sem);
```

```
mntput(mnt);
```

```
.....
```

struct vfsmount *do_kern_mount(*char *type, int flags, char *name, void *data*)

```
struct file_system_type * fstype;
struct vfsmount *mnt = NULL;
struct super_block *sb;
.....
/* we need capabilities... */
if (!capable(CAP_SYS_ADMIN))
    return ERR_PTR(-EPERM);

/* ... filesystem driver... */
fstype = get_fs_type(type);
.....
```

struct vfsmount *do_kern_mount(*char *type, int flags, char *name, void *data*)

```
/* ... allocated vfsmount... */  
  
mnt = alloc_vfsmnt();  
  
.....  
  
set_devname(mnt, name);  
  
/* get locked superblock */  
  
if (fstype->fs_flags & FS_REQUIRES_DEV)  
    sb = get_sb_bdev(fstype, name, flags, data);  
else if (fstype->fs_flags & FS_SINGLE)  
    sb = get_sb_single(fstype, flags, data);  
else  
    sb = get_sb_nodev(fstype, flags, data);
```

struct vfsmount *do_kern_mount(*char*
**type, int flags, char *name, void *data*)

```
.....  
mnt->mnt_sb = sb;  
mnt->mnt_root = dget(sb->s_root);  
mnt->mnt_mountpoint = mnt->mnt_root;  
mnt->mnt_parent = mnt;  
up_write(&sb->s_umount);  
fs_out:  
    put_filesystem(fstype);  
    return mnt;  
}
```

Filesystem Registration/Unregistration

- I Provides a mechanism for new filesystems to be written with minimum effort
- I Steps required to implement a filesystem
 - Fill in a struct `file_system_type` structure
 - Register it with the VFS using the `register_filesystem()` function

Example : bfs

```
static DECLARE_FSTYPE_DEV(bfs_fs_type, "bfs", bfs_read_super);  
static int __init init_bfs_fs(void)  
{  
    return register_filesystem(&bfs_fs_type);  
}  
static void __exit exit_bfs_fs(void)  
{  
    unregister_filesystem(&bfs_fs_type);  
}  
module_init(init_bfs_fs)  
module_exit(exit_bfs_fs)
```

struct file_system_type

```
{  
    const char *name;  
    int fs_flags;  
    struct super_block *(*read_super)(struct super_block *, void*, int);  
    struct module *owner;  
    struct vfsmount *kern_mnt; /*For kernel mount, if it's FS_SINGLE fs*/  
    struct file_system_type *next;  
}
```

human readable name, appears
in /proc/filesystems

linkage into singly-linked list register/unregister_filesystem()
modify it by linking and unlinking the entry from the list.

read_super()

- | Fills in the fields of the superblock
- | Allocate root inode
- | Initialize any fs-private information associated with this mounted instance of the file filesystem

bfs_read_super(*struct super_block * s, void * data, int silent*)

```
.....  
dev = s->s_dev;  
set_blocksize(dev, BFS_BSIZE);  
s->s_blocksize = BFS_BSIZE;  
s->s_blocksize_bits = BFS_BSIZE_BITS;
```

```
bh = sb_bread(s, 0);
```

```
if(!bh)
```

```
    goto out;
```

```
bfs_sb = (struct bfs_super_block *)bh->b_data;
```

Read the superblock from the device specified , using buffer cache

bfs_read_super(*struct super_block * s, void * data, int silent*)

```
.....  
259     if (bfs_sb->s_magic != BFS_MAGIC) {  
260         if (!silent)  
261             printf("No BFS filesystem on %s (magic=%08x)\n",  
262                 bdevname(dev), bfs_sb->s_magic);  
263         goto out;  
264     }  
265     .....
```

bfs_read_super(*struct super_block * s, void * data, int silent*)

```
301     for (i=BFS_ROOT_INO; i<=s->su_lasti; i++) {
302         inode = iget(s,i);
303         if (inode->iu_dsk_ino == 0)
304             s->su_freei++;
305         else {
306             set_bit(i, s->su_imap);
307             s->su_freeb -= inode->i_blocks;
308             if (inode->iu_eblock > s->su_lf_eblk) {
309                 s->su_lf_eblk = inode->iu_eblock;
310                 s->su_lf_sblk = inode->iu_sblock;
311                 s->su_lf_ioff = BFS_INO2OFF(i);
312             }
313         }
314         iput(inode);
```

Reference

- | Understanding the LINUX KERNEL - *O'reilly*
- | Cross-Reference Linux - *<http://lxr.linux.no>*
- | Linux Kernel 2.4 Internals
 - *<http://linux-study.cs.ccu.edu.tw/lki/lki.html>*