Department of Computer Sciences THE UNIVERSITY OF TEXAS AT AUSTIN

CS 378 (Spring 2003)

Linux Kernel Programming

Yongguang Zhang

(ygz@cs.utexas.edu)

This Lecture

- Linux File System
  - Mounting a filesystem
  - Walking a path
  - Tracing open(), read()

- Questions?

VFS Data Structure Review

- Four basic data structures
  - superblock: about a file system instance
  - inode: about a particular file
  - dentry: about directory tree structure
  - file: about an open file handle by a process
- Each filesystem implementation provides the set of operations for each object
- Other data types
  - Filesystem type: about different filesystem implementations

Mounting a Filesystem

- Filesystem mounting
  - Each filesystem instance is a tree
  - Mounting: graft the root of one filesystem tree to the leaf of another to make a bigger tree
- Terminologies
  - Mount point
  - Root directory of a mounted filesystem
  - Root filesystem
- Data structure: struct vfsmount
  - Represent a mounted filesystem instance

struct vfsmount

- Defined in include/linux/mount.h
- Important fields:
  - mnt_parent: pointer to parent vfsmount
  - mnt_mountpoint: Dentry for the mount point
  - mnt_root: Dentry for the root directory (of this fs)
  - mnt_sb: Superblock for this filesystem instance
  - Various lists: mnt_hash, mnt_mounts, mnt_child, mnt_list
  - Flags, device name, etc.: mnt_flgs, mnt_count, mnt_devname

Walking a Path

- Find the inode from a given pathname
  - Common VFS procedure, used frequently
- Starting point dentry:
  - Pathname starts with / : current->fs->root
  - Otherwise: current->fs->pwd
- Special handling in walking the path
  - Symbolic links (and check looping)
  - Access permission
  - Crossing a mount point into a different filesystem

Path Walking Procedures

- Call the following three functions (in this order):
  - path_init(name,flags,nd)
  - path_walk(name,nd)
  - path_release(nd)
- A "context" used in path walking
  - nd is of type struct nameidata (in include/linux.fs.h)
  - Field dentry: the "current" dentry used in the walk
  - Field mnt: object of type vfsmount for this filesystem
- Code:
  - See fs/namei.c

path_init()

- Set up the nameidata object before a walk
  - Set dentry and mnt to the starting point
  - Initialize flags and other fields
- If name starts with /
  - Call walk_init_root(), which does

```
nd->mnt = mntget(currnt->fs->rootmnt);
nd->dentry = dget(current->fs->root);
```

- If name does not start with /,
  - Do

```
nd->mnt = mntget(currnt->fs->pwdmnt);
nd->dentry = dget(current->fs->pwd);
```

path_walk()

- Given the pathname and the initialized nd object
  - Actual work done in link_path_walk()
- For each real path component
  - Check for permission: call permission()
  - Calculate the hash value
  - Check special component name (like ., ..)
  - Lookup from the dcache: call cached_lookup()
  - Lookup from the disk: call real_lookup()
  - Check mountpoint, symbolic links, errors, etc.
  - Set the dentry in nd down to this component

cached_lookup(parent,name,flgs)

- Look up the dentry in dcache (given parent dentry and filename)
  - Call d_lookup() to return the dentry
  - Call dentry->d_op->d_revalidate() if defined (usually used in networked file system for stale files)
- Routine d_lookup()
  - Find the hash collision list with d_hash() call
  - Search the list for matching parent and filename
  - Use parent->d_op->d_compare() (if defined) to match the filename

real_lookup(parent,name,flags)

- Load dentry from the disk
  - Called when cached_lookup() fails to return the dentry
- Essentially,
  - Get a free dentry (from dcache) and set the filename

    ```
    struct dentry *dentry = d_alloc(parent, name);
    ```
  - Call parent inode's lookup method to fill the dentry

    ```
    struct inode *dir = parent->d_inode;
    dir->i_op->lookup(dir, dentry);
    ```
  - Filesystem-specific lookup() involves searching the directory content, and perhaps loading in a new inode

An Example Lookup()

- For EXT2: ext2_lookup() in fs/ext/namei.c:

```
ino = ext2_inode_by_name(dir, dentry)
if (ino) {
        inode = iget(dir->i_sb, ino);
        ...
}
d_add(dentry, inode);
```

  - ext2_inode_by_name(): search directory content (on disk) for the file named in dentry, return inode number
  - iget(): getting the inode object (back to VFS layer)

Getting inode with iget()

- iget(sb,ino): given superblock and inode number, return the inode object
  - Calls iget4(sb,ino,...)
  - Which first try the hash table

    ```
    struct list_head *head = inode_hashtable + hash(sb,ino);
    inode = find_inode(sb, ino, head, ...);
    ```

  - If not, load from disk

    ```
    return get_new_inode(sb, ino, head, ...);
    ```

- get_new_inode():
  - Allocate new inode object (from slab inode_cachep)
  - Call sb->s_op->read_inode(inode)

Going Down in path_walk()

```
Inode = dentry->d_inode;
if (!inode)
        goto out_dput;                  /* no such file! */

if (!inode->i_op)
        goto out_dput;                  /* this is not a directory */

if (inode->i_op->follow_link) {
        /* this is a symbolic link, call do_follow_link(dentry,nd) */
} else {
        dput(nd->dentry);
        nd->dentry = dentry;            /* set nd down a path */
}

if (!inode->i_op->lookup)
        break;                          /* this is not a directory */
```

More about path_walk()

- Checking and calling dentry-specific methods
  - Example

```
if (nd->dentry->d_op && nd->dentry->d_op->d_hash) {
        err = nd->dentry->d_op->d_hash(nd->dentry, &this);
        ...
```

  - Similar code in many subroutines, e.g. In permission(), cached_lookup()
- Special handling
  - Going up a directory: call follow_dotdot(nd)
  - Going down across a mount point: call __follow_down(&nd->mnt, &dentry)

Path Walking Example

- After path_walk(name,nd) the dentry is found in nd.
- Example: sys_chdir(filename)

```
struct nameidata nd;

if (path_init(name, ...., &nd))
        error = path_walk(name, &nd);

set_fs_pwd(current->fs, nd.mnt, nd.dentry);

path_release(&nd);
```

Tracing open()

- User process:
  - fd=open(“xxx”,O_RDWR,0)
- System call service routine
  - long sys_open(filename,flags,mode) (in fs/open.c)
- Given filename, return open file object
  - struct file *filp_open(filename,flags,mode)
- Given dentry, return open file object
  - struct file *dentry_open(dentry,mnt,flags)
- Call filesystem-specific file open operation

sys_open(filename,flags,mode)

```
long sys_open(filename,flags,mode):

    tmp = getname(filename);
        /* getname() → do_getname() → strncpy_from_user() */

    fd = get_unused_fd();
        /* get free array index from current->files->fd[] */

    struct file *f = filp_open(tmp, flags, mode);

    fd_install(fd, f);                  /* current->files->fd[fd] = f */

    return (fd);
```

filp_open(filename,flags,mode)

- Open the file and return the open file object
  - Essentially

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

- Function open_namei()
  - Not to create: simply walk the path with path_walk()
  - For creating a new file: walk the path to find the parent and call the vfs_create()

dentry_open(dentry,mnt,flags)

- Do the following steps
  - Get an empty struct file object (first from free list free_list, then from slab cache filp_cachep)

    f = get_empty_filp();
  - Assign the file operations from the inode's default

    inode = dentry->d_inode;

    f->f_op = fops_get(inode->i_fop);
  - Move to the open file list of this filesystem

    file_move(f, &inode->i_sb->s_files);
  - Call the customized open operation:

    if (f->f_op && f->f_op->open) {

        error = f->f_op->open(inode,f);

How is a file's f_op Populated?

- From the inode's i_fop (default file operation)
  - Populated in dentry_open()
- Where is an inode's i_fop populated?
  - In the superblock's read_inode() method.
  - Example, in ext2_read_inode() (in fs/ext2/inode.c)

```
else if (S_ISREG(inode->i_mode)) {
        inode->i_fop = &ext2_file_operations;
} else if (S_ISDIR(inode->i_mode)) {
        inode->i_fop = &ext2_dir_operations;
}
```

Example file Operations

- EXT2: see fs/ext2/file.c:

```
struct file_operations ext2_file_operations = {
        llseek:         generic_file_llseek,
        read:           generic_file_read,
        write:          generic_file_write,
        ioctl:          ext2_ioctl,
        mmap:           generic_file_mmap,
        open:           generic_file_open,
        release:        ext2_release_file,
        fsync:          ext2_sync_file,
};
```

Tracing read()

- User process:
  - read(fd,buf,count)
- System call service routine
  - long sys_read(fd,buf,count) (in fs/read_write.c)
- Essentially
  - Get the open file object from the file descriptor

    structure file * file = fget(fd);
  - Call the customized read() function

    ssize_t (*read)(struct file *, char *, size_t loff_t *);

    if (file->f_op && (read = file->f_op->read) != NULL)

        ret = read(file, buf, count, &file->f_ops);

fget(fd)

- Given file descriptor, return the open file object
  - Essentially,

```
file = fcheck(fd);
if (file)
        get_file(file);
```

  - Function fcheck(fd)

```
struct file_struct *files = current->files
if (fd < files->max_fds)
        file = files->fd[fd];
```

  - Macro get_file(file)

```
#define get_file(x)          atomic_inc(&(x)->f_count)
```

generic_file_read()

- Set up read descriptor
- Call do_generic_file_read()
  - The main routine for file reading
  - Divide file into pages, operates on page boundary
  - Check page cache, read ahead, ...
  - Actual page read: invoke filesystem-specific readpage() operation

An Example readpage()

- EXT2: see fs/ext2/inode.c:

```
static int ext2_readpage(struct file *file, struct page *page)
{
        return block_read_full_page(page, ext2_get_block);
}
```

- Back to VFS layer: block_read_full_page()
  - Call submit_bh() to start the I/O
  - Which calls generic_make_request() to add to the request queue

Summary

- Next lecture: Linux Networking
- You need to have basic knowledgement about internetworking: protocols, IP address, subnet, routing, etc.
  - At the minimum, you should know this: http://www.netfilter.org/documentation/HOWTO/networking-concepts-HOWTO.html