Ausgewählte Betriebssysteme

Filesystem

File Systems

- Name space
  - Hierarchical tree structure
- Simple I/O API
  - `open, close, read, write`
- Uniform interface
  - Persistent storage
  - I/O devices
  - Interprocess communication
  - Kernel-user communication

File System types

File System Types (2)

- Persistent
  - Block device based (disk)
    - Ext2, VFAT ....
  - Network based
    - NFS, coda, AFS
- Virtual
  - Provide information through file API
    - procfs, sysfs, devfs, usbfs...

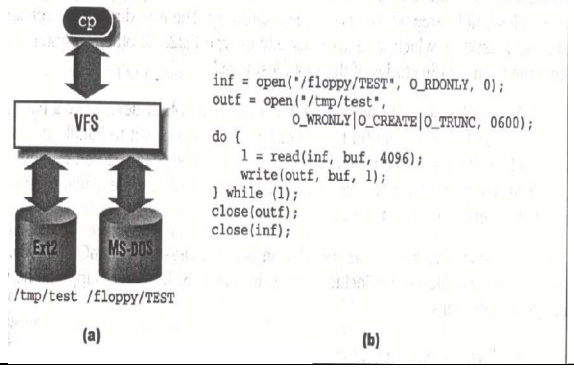
File systems (3)

```
peter@krypton:~> uname -a
Linux krypton 2.5.67 #1 SMP Tue Apr 8 00:17:05 CEST 2003 i686 unknown
peter@krypton:~> cat /proc/filesystems
nodev   sysfs
nodev   rootfs
nodev   bdev
nodev   proc
nodev   sockfs
nodev   usbfs
nodev   usbdevfs
nodev   futexfs
nodev   tmpfs
nodev   pipefs
nodev   eventpollfs
nodev   binfmt_misc
nodev   devpts
        ext3
        ext2
nodev   ramfs
        iso9660
nodev   nfs
nodev   nfsd
nodev   autofs
        reiserfs
nodev   oprofilefs
nodev   rpc_pipefs
```


VFS Layer

- Virtual Filesystem Switch
- Kernel abstraction for different file system implementations
- Framework
  - Define usage model
  - Implement common functionality
  - Provide hooks for specific implementations

User view




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

(b)

Common File Model

- Mirrors UNIX file model strictly
- Features not available in a particular FS must be emulated
  - Directories as files (UNIX) vs. special tables (FAT)
- Indirection layer associated with objects
  - Pointer to function pointer table

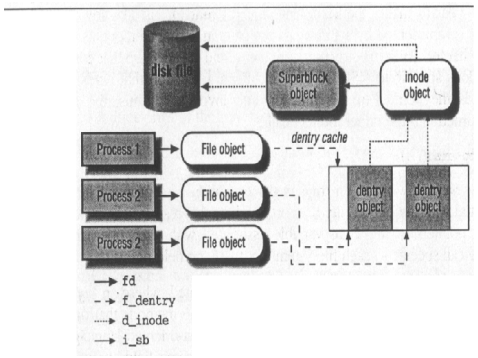
Commen File Model (2)

- Superblock (whole fs instance)
  - Represents a whole mounted file system
- Inode (unique entity for an object in a fs)
  - Manages properties of that particular object on disk
- Dentry (directory structure)
  - Represents an entry in a directory
  - Supports path-name to inode mapping
- File (file as seen by a task)
  - Session specific

Object indirection

- Each object has an function pointer table associated with it
- VFS framework calls customized function
  - `size_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);`
- Actual functions provided by file system implementation
- Operation table filled upon object creation or initialization

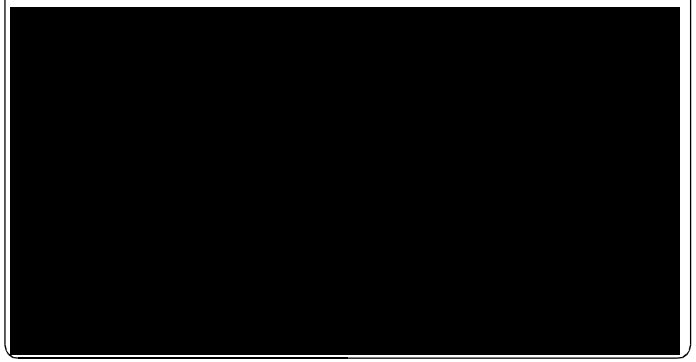
VFS objects



Memory objects

- VFS objects are cached in memory
- Read from disk when needed
- Slab allocator for each object type

Disk and memory objects



File system types

- Structure for each particular file system
  - Populated during system startup or module loading
  - Upon mounting each registered file system probes partition unless the type is explicitly provided
- Important properties
  - `name` identifying name
  - `read_super` file system specific function
  - `fs_supers` all mounted file systems in a list
  - `owner` module that provides the implementation

Registered file systems

FS registration

```
/* file: fs/ext2/super.c */

static struct file_system_type ext2_fs_type =
{
      .owner      = THIS_MODULE,
      .name       = "ext2",
      .get_sb     = ext2_get_sb,
      .kill_sb    = kill_block_super,
      .fs_flags   = FS_REQUIRES_DEV,
};
```

```
static int __init init_ext2_fs(void)
{
      int err = init_ext2_xattr();
      if (err)
            return err;
      err = init_inodecache();
      if (err)
            goto out1;
        err = register_filesystem(&ext2_fs_type);
      if (err)
            goto out;
      return 0;
out:
      destroy_inodecache();
out1:
      exit_ext2_xattr();
      return err;
}
static void __exit exit_ext2_fs(void)
{
      unregister_filesystem(&ext2_fs_type);
      destroy_inodecache();
      exit_ext2_xattr();
}
module_init(init_ext2_fs)
module_exit(exit_ext2_fs)
```

Superblock

- Kernel data structure for a mounted fs
  - Data type: `struct super_block` (include/linux/fs.h)
- Important fields
  - fs parameters: `s_blocksize, s_maxsize`
  - fs type: `s_type`
  - Pointer to method array: `s_op`
  - File system specific data: `s_fs_info`
  - root dentry: `s_root`

Superblock (2)

- `struct super_operations` include/linux/fs.h
  - Load inode object from disk: `read_inode`
  - Write inode object to disk: `write_inode`
  - Decrement reference count: `put_inode`
  - Decrement superblock object: `put_super`
  - Delete inode (with file content): `delete_inode`

dentry

- Kernel data structure for directory entry
  - Associates name with inode object
    - Several dentries can refer to the same inode
  - No corresponding disk data structure
    - Directories are files with special interpreted content (UNIX)
- dentry cache
  - Recently used (looked up) dentry object will be kept in a slab cache

dentry lists

- Tree layout
  - Reflecting the directory layout
- Hash table
  - Fast lookup from filename to dentry object
- List of unused dentry objects
- List of aliases
  - Same inode, different dentries (hardlinks)

dentry tree

dentry lists (2)

inode

- Kernel data structure for a file (or directory)
  - Inode number is unique per fs
  - Names are not (hard links)
- To access a file
  - Allocate inode object in memory
  - Initialize it with data from disk
- Inode cache
  - Recently used objects (in memory) are kept for further reference

Inode data structure

- `Struct inode` (include/linux/fs.h)
- Important fields
  - Number, superblock: `i_ino, i_sb`
  - Reference counter: `i_count`
  - File information: `i_mode, i_nlink, i_uid, i_gid, i_size,i_atime, i_mtime, i_ctime, i_blksize, i_blocks`
  - Inode methods: `i_op`
  - Default file methods: `i_fop`
  - Inode lists: `i_hash, i_list`
  - Referring dentries: `i_dentry`

inode (2)

inode methods

- `struct inode_operations` (include/linux/fs.h)
- fs dependent operations on inode
  - Create a new inode (and a new file): `create`
  - Find by name: `lookup`
  - Create, remove hardlink: `link, unlink`
  - Create, remove directory: `mkdir, rmdir`
  - Special cases: `symlink, mknod`

file objects

- Kernel data structure for a file opened by a task
  - `struct file` (include/linux/fs.h)
- Important fields
  - dentry object: `f_dentry`
  - File operations: `f_op`
  - Current file pointer: `f_pos`
  - Reference count: `f_count`
  - List link: `f_list`
  - Device driver data: `private_data`

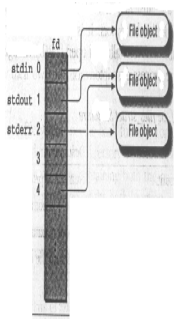
File operations

```
struct file_operations {
     struct module *owner;
     loff_t (*llseek) (struct file *, loff_t, int);
     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
     ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
     ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
     int (*readdir) (struct file *, void *, filldir_t);
     unsigned int (*poll) (struct file *, struct poll_table_struct *);
     int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
     int (*mmap) (struct file *, struct vm_area_struct *);
     int (*open) (struct inode *, struct file *);
     int (*flush) (struct file *);
     int (*release) (struct inode *, struct file *);
     int (*fsync) (struct file *, struct dentry *, int datasync);

     int (*aio_fsync) (struct kiocb *, int datasync);
     int (*fasync) (int, struct file *, int);
     int (*lock) (struct file *, int, struct file_lock *);
     ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
     ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
     ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
     ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
     unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
unsigned long);
};
```

Tasks and files

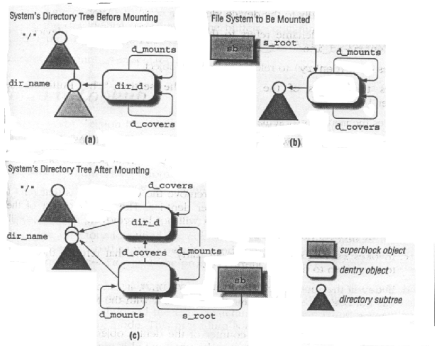
- Fields in `task_struct`
- `struct fs_struct *fs`
  - fs related information
    - `struct dentry *root`
    - `sturct dentry *pwd`
- `struct files_struct *files`
  - Currently open files
  - `t->files->fd[i]` to access file for file descriptor i



Mounting a file system

- Filesystem mounting
  - Each fs can be represented by a tree structure
  - Mounting: graft the root of one filesystem tree to the leaf of another to get a bigger tree
- Terminologies
  - Mount point: leave of a file system where the next fs gets appended
  - Root file system: root fs, by magic on the top
  - Root directory of a mounted fs
- Data structure: struct vfsmount
  - Representing a mounted fs instance

Mount anatomy



Walking a path

- Find the inode for a given pathname
  - Common problem, used frequently
- Starting point dentry:
  - Leading '/': current->fs->root
  - Otherwise: current->fs->pwd
- Special handling when walking a path
  - Symbolic links (loop detection)
  - Access permission
  - Crossing a mount point into another filesystem

Path Walking procedure

- Two relevant functions
  - Lookup path, lock final dentry: `path_lookup(name, flags, nd)`
  - Decrement reference counts: `path_release(nd)`
- `struct nameidata nd` is the context used for walking
  - Field `struct dentry *dentry`: the current (last used) dentry
  - Field `struct vfsmount *mnt`: current file system

`path_lookup` preparation

- Set up the nameidata object before walking
  - Set dentry and mnt to the starting point
  - Initialize flags and other fields
  - If name starts with '/'
    - `nd->mnt = mntget(current->fs->rootmnt);`
      `nd->dentry = dget(current->fs->root);`
  - If name does not start with '/'
    - `nd->mnt = mntget(current->fs->pwdmnt);`
      `nd->dentry = dget(current->fs->pwd);`

Walking further

- Actual work done in `link_path_walk`
- For each real path component
  - Check for permission: `permission`
  - Calculate hash value
  - Check for special component name (like '.', '..')
  - Lookup from the dcache: `cached_lookup`
  - Lookup from disk if not cached: `real_lookup`
  - Check mountpoint, symbolic links, errors etc.
  - Set the dentry in `nd` down to the new component

dentry cache lookup

- Look up the dentry in dcache
  - Call `d_lookup` to return the dentry
  - Call `dentry->d_op->d_revalidate` if defined
    - Usually in network fs for stale files
- Routine d_lookup
  - Find the hash bucket with `d_hash`
  - Search the list for matching parent and filename
  - Use `parent->d_op->d_compare` if defined to match the filename

Real lookup

- 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 inodes lookup method to file 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 a new inode