CS 378 (Spring 2003)

Linux Kernel Programming

Yongguang Zhang

(ygz@cs.utexas.edu)

This Lecture

- Linux File System
  - Overview
  - Four basic data structures (superblock, inode, dentry, file)

- Questions?

Using File Systems in UML (1)

- You need a block device
  - In UML: a block device can be emulated by a file
  - Create a file (say, 4M in size)

    dd if=/dev/zero of=new_fs bs=1M count=4
- Run UML with the new block device
  - With command line argument ubd1=

    ./linux ubd0=... ubd1=new_fs ...
  - Look at /dev/ubd/:

```
usermode:~# ls -l /dev/ubd
total 0
brw-rw----    1 root     root      98,   0 Dec 31  1969 0
brw-rw----    1 root     root      98,   1 Dec 31  1969 1
```

Using File Systems in UML (2)

- Create a new file system on the new block device
  - Make a MSDOS file system:

    /projects/cs378.ygz/bin/mkdosfs -S 1024 -v /dev/ubd/1
  - Make a MINIX file system

    mkfs.minix /dev/ubd/1
  - Make a EXT2 file system

    mkfs.ext2 /dev/ubd/1
  - Disaster aware: never mistype as /dev/ubd/0

Using File Systems in UML (3)

- You need a mount point
  - Usually under /mnt

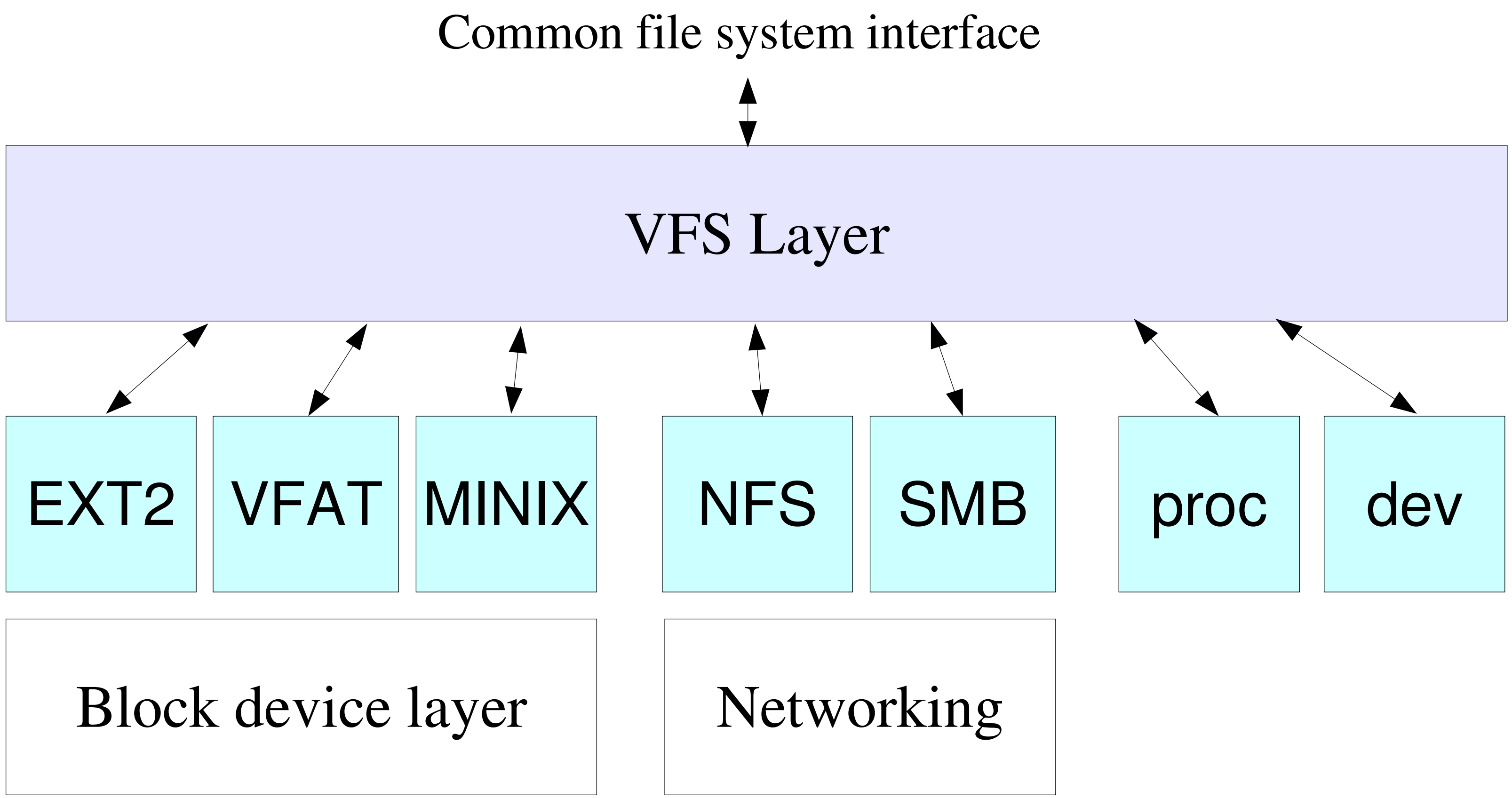
    mkdir /mnt/test
- Mount the new block device (with filesystem)
  - Give the block device file, and the mount point

    mount /dev/ubd/1 /mnt/test
- Sysadmin tools for file systems
  - mount
  - umount /mnt/test

Linux File Systems

- The kernel subsystem that manage file system directories in kernel and external memory
  - It's about mapping file name to blocks in storage device
- Linux supports many file system types
  - Each type is a different filesystem implementation
  - Usually each type in a loadable module
  - Some important one are builtin
- Linux support many file system instances
  - Each instance is a mounted file system
  - Every linux system has a root file system

File System Software Architecture



Linux File System Types

- Real device-based filesystems
  - Build on top of a block device (disk, storage, …)
  - Example: EXT2, VFAT, …
- Network filesystems
  - Special module to deal with network protocols
  - Example: NFS, SMB, …
- Virtual/special filesystems
  - /proc, /dev (devfs), ...

VFS Layer

- Purpose
  - Provide the same interface to user mode processes
  - Provide a kernel abstraction for all different file system implementations
- Functions
  - Service file and file system related system calls
  - Manage all file and file system related data structures
  - Routines for efficient lookup, traverse the file system
  - Interact with specific file system modules

Four Basic VFS Data Structures

- superblock: about a file system
  - Each mounted file system has a superblock object
- inode: about a particular file (Unix vnode)
  - Every file is represented by an inode record on disk
  - Some loaded in the kernel memory as inode objects
- dentry: about directory tree structure
  - Each entry in a directory is represented by a dentry
  - For pathname-to-inode mapping purpose
- file: about an open file handle by a process
  - Each task tracks open files by the file handles

VFS Object Relationships

- Dentry and inode
  - Dentry resides in kernel memory only
  - Inode resides on disk, but loaded into memory for access (any change should be committed back to disk)
  - One-to-one mapping between file and inode
  - A file (inode) can have many dentries (e.g., hardlinks)
- Many lists (double linked)
  - Chain objects of the same type through a field of type `struct list_head *`
  - List header is a variable or a field in an object of another type

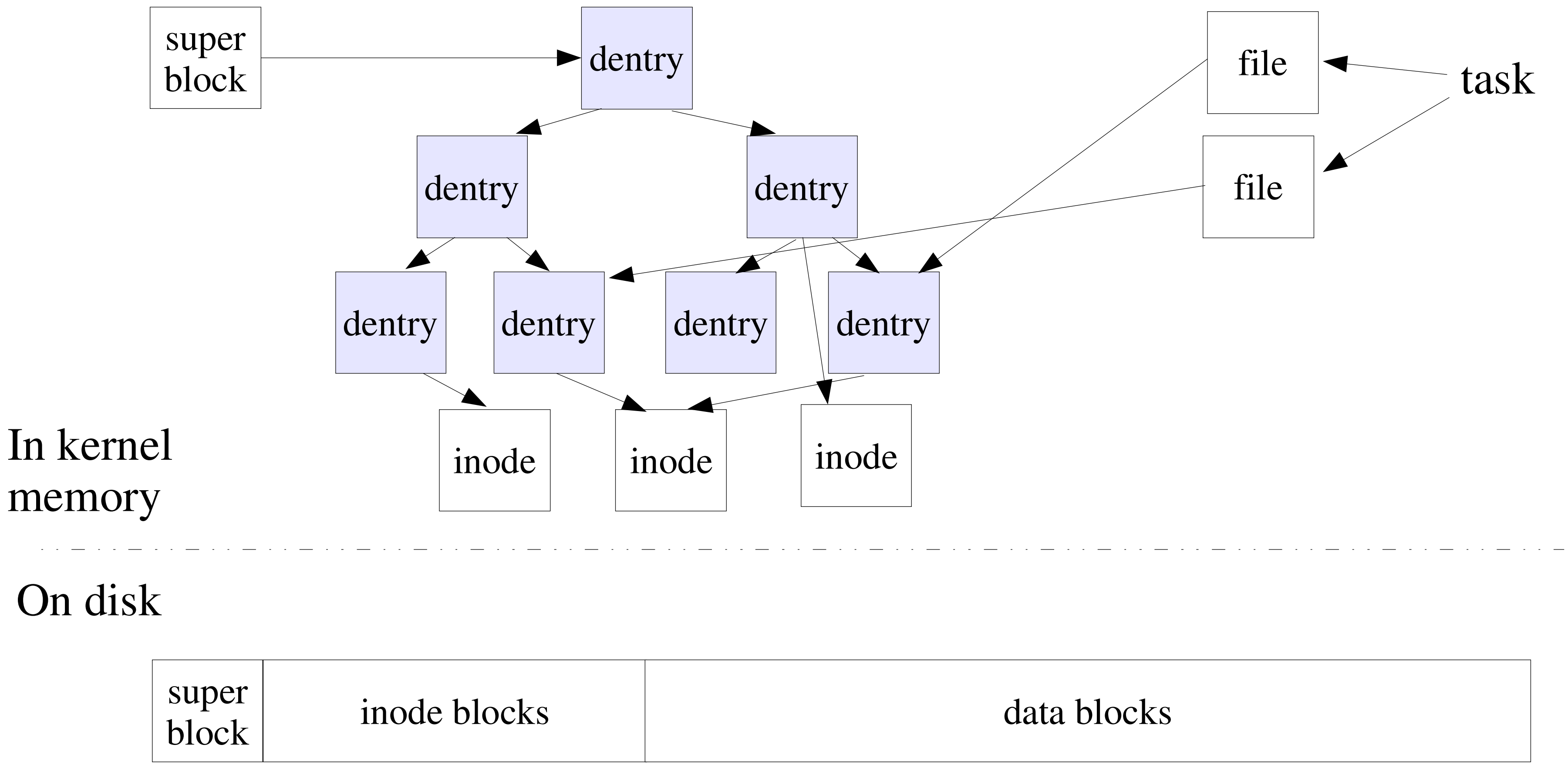
Object Methods

- An operation table for each VFS object
  - Each object provides a set of operations (function pointers)
  - Usually filesystem-type-dependent, but sometimes filesystem-specific, or file-specific (e.g., char dev)
- Interface b/w VFS layer and specific filesystem
  - VFS layer calls these functions for operations provided by the specific filesystem module
  - Actual functions implemented in filesystem modules
  - Operation table populated when the VFS object is loaded or initialized

Memory Management

- Memory for VFS objects are slab-allocated
  - Each VFS data type is a special slab
- Each slab is also a cache
  - A released (unused) data object is returned to slab uncleaned (fields not erased)
  - VFS still maintains pointers to a released object
  - It can be reclaimed and reused instancely

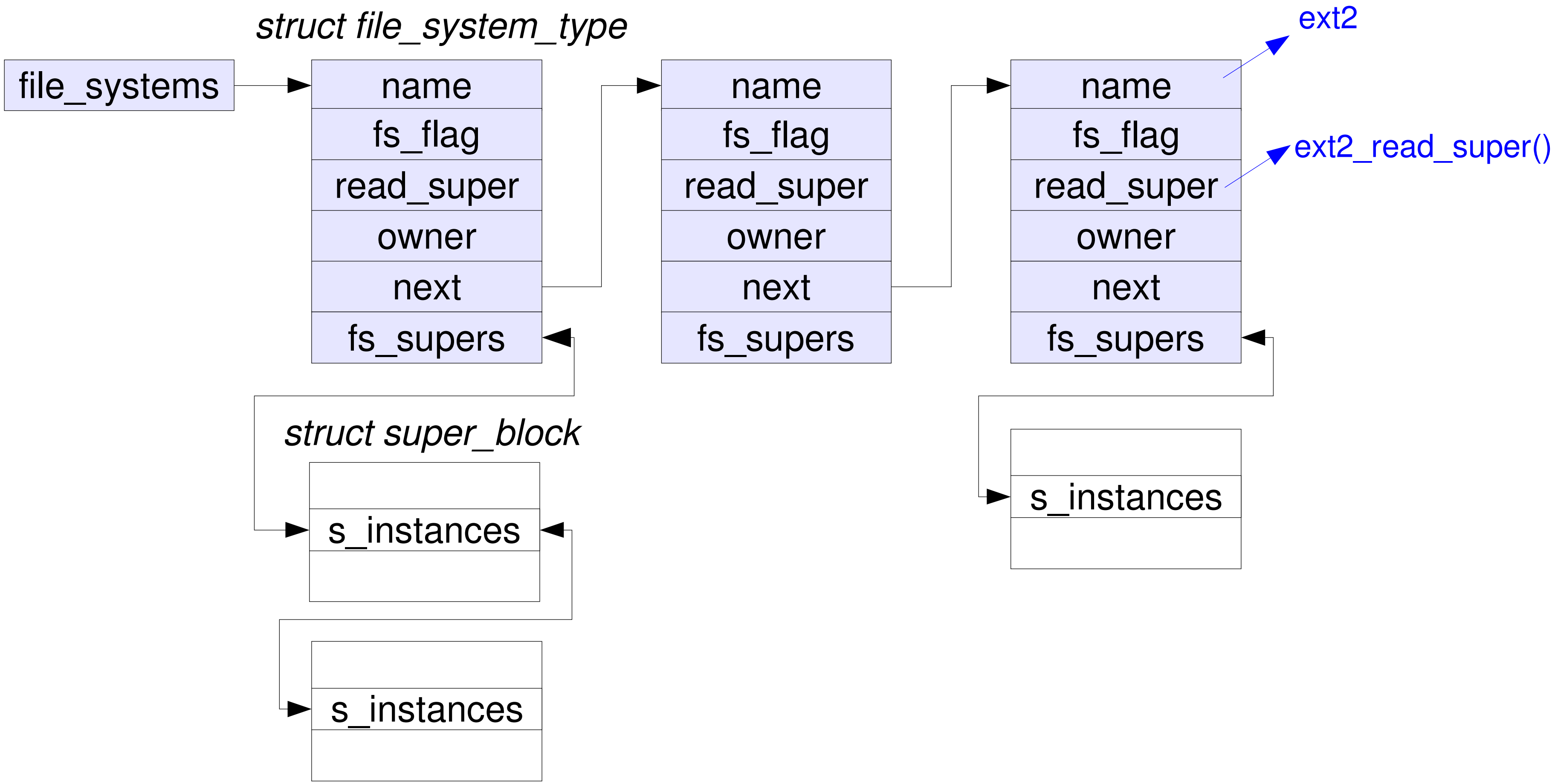
Abstract View



Filesystem Types

- Kernel record for a file system implementation
  - A list of all built-in or loaded filesystem modules
  - Type: struct file_system_type (include/linux/fs.h)
- Major fields:
  - name: name of the filesystem type, like "ext2"
  - read_super(): function on how to read the superblock
  - owner: module that implements this filesystem type
  - fs_flags: whether it requires a real device, etc.
  - fs_supers: a list of superblocks (for all mounted filesystem instances of this type)

Filesystem Type Illustrations



Filesystem Type Registration

- Each type of filesystem must register itself
  - Usually at module load time (i.e. with module_init(...))
  - Must unregister when the module is unloaded
- To register
  - Allocate a filesystem type object

    DECLARE_FSTYPE(var,type,read,flags)

    DECLARE_FSTYPE_DEV(var,type,read)
  - Write the read_super() function (filesystem type-dependent superblock reading function)
  - Call register_filesystem(struct file_system_type *)

Example FS Module Registration

- Look at end of fs/ext2/super.c

```
static DECLARE_FSTYPE_DEV(ext2_fs_type, "ext2", ext2_read_super);

static int __init init_ext2_fs(void)
{
	return register_file_system(&ext2_fs_type);
}

static int __exit exit_ext2_fs(void)
{
	unregister_file_system(&ext2_fs_type);
}

EXPORT_NO_SYMBOLS;

module_init(init_ext2_fs)
module_exit(exit_ext2_fs)
```


VFS Superblock

- Kernel data structure for a mounted file system
  - Data type: struct super_block (include/linux/fs.h)
- Important fields
  - File system parameters: s_blocksize, s_blocksize_bits, s_maxbytes, ...
  - Pointer to the file system type: s_type
  - Pointer to a set of superblock methods: s_op
  - Status of this record: s_dirt, s_lock, s_count, ...
  - Dentry object of the root directory: s_root
  - Lists of inodes: s_dirty, s_locked_inodes
  - A union of specific file system information: u

Superblock Methods

- A set of operations for this file system instance
  - read_inode(): load the inode object from disk
  - write_inode(): update inode object onto disk
  - put_inode(): unreference this inode object
  - delete_inode(): delete the inode and corresponding file
  - put_super(): unreference the superblock object
  - And more (see struct super_operations in include/linux/fs.h)
- Operations should be filesystem type dependent
  - Set up by read_super() of file_system_type

Filesystem-Specific Superblock Data

- Union u in struct super_block

```
#include <linux/minix_fs_sb.h>
#include <linux/ext2_fs_sb.h>
#include <linux/ext3_fs_sb.h>
...
struct super_block {
        ...
        union {
                struct minix_sb_info    minix_sb;
                struct ext2_sb_info     ext2_sb;
                struct ext3_sb_info     ext3_sb;
                ...
                void                    *generic_sbp;
        } u;
        ...
}
```

VFS dentry

- Kernel data structure for directory entry
  - About name to inode mapping (in a directory)
  - Encodes the filesystem tree layout
  - The way to look into the file space in the filesystem
  - Each dentry points to the inode
- Dentry cache (dcache)
  - Recently accessed (and released) dentry will be cached (in slab cache) for performance purpose

VFS dentry Data Structure

- Data structure
  - struct dentry (in include/linux/dcache.h)
- Important fields
  - Pointer to the associated inode: d_inode
  - Parent directory: d_parent
  - List of subdirectories: d_subdirs
  - Linking this object in various lists: d_hash, d_lru, d_child
  - Pointer to a set of dentry methods: d_op
  - Using this entry object: d_count, d_flags, ...

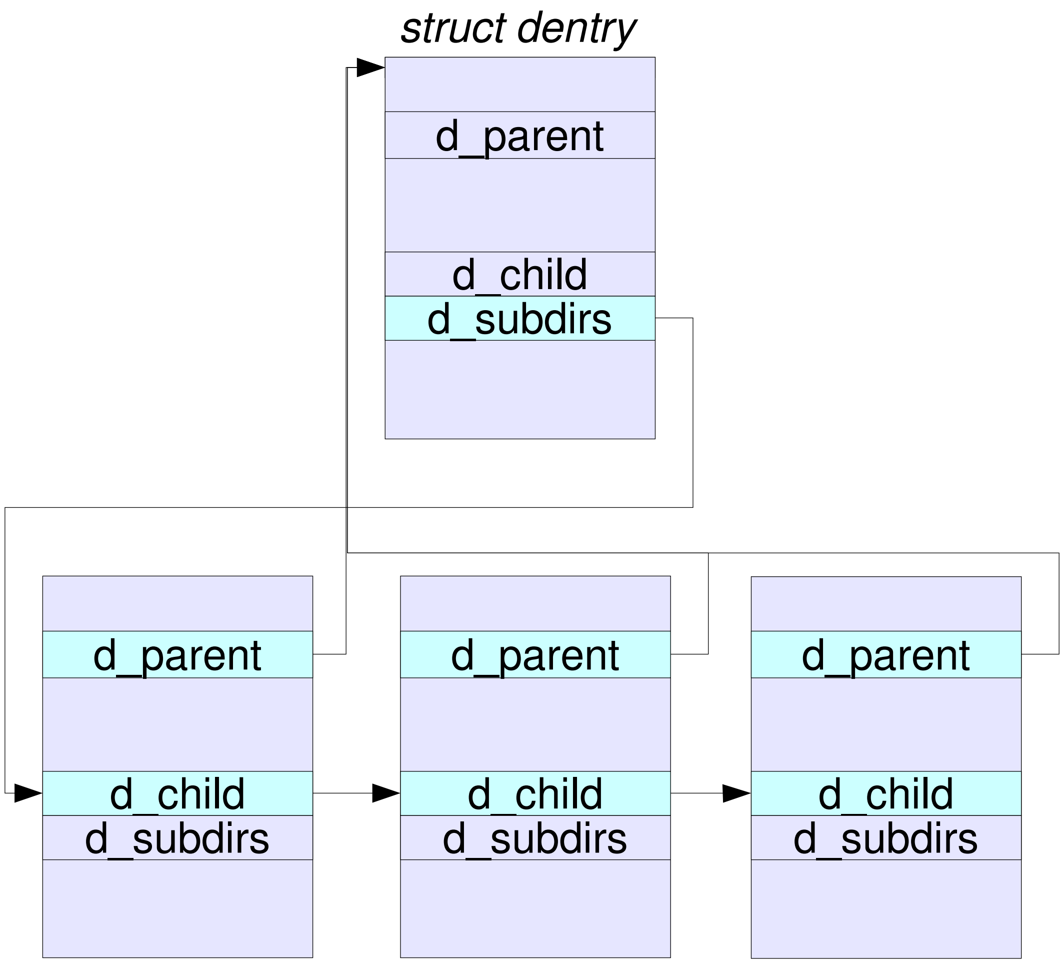
Dentry Methods

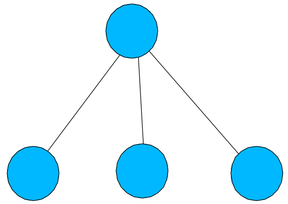
- A set of filesystem-dependent dentry operations
  - d_hash(): return a hash value for this dentry
  - d_compare(): filesystem-dependent filename compare
  - d_delete(): called when d_count becomes zero
  - d_release(): called when slab allocator really frees it
  - And more (see struct dentry_operations in include/linux/dcache.h)
- Common (filesystem-independent) dentry utilities
  - d_add(), d_alloc(), d_lookup(), ... (see include/linux/dcache.h)

Dentry Lists

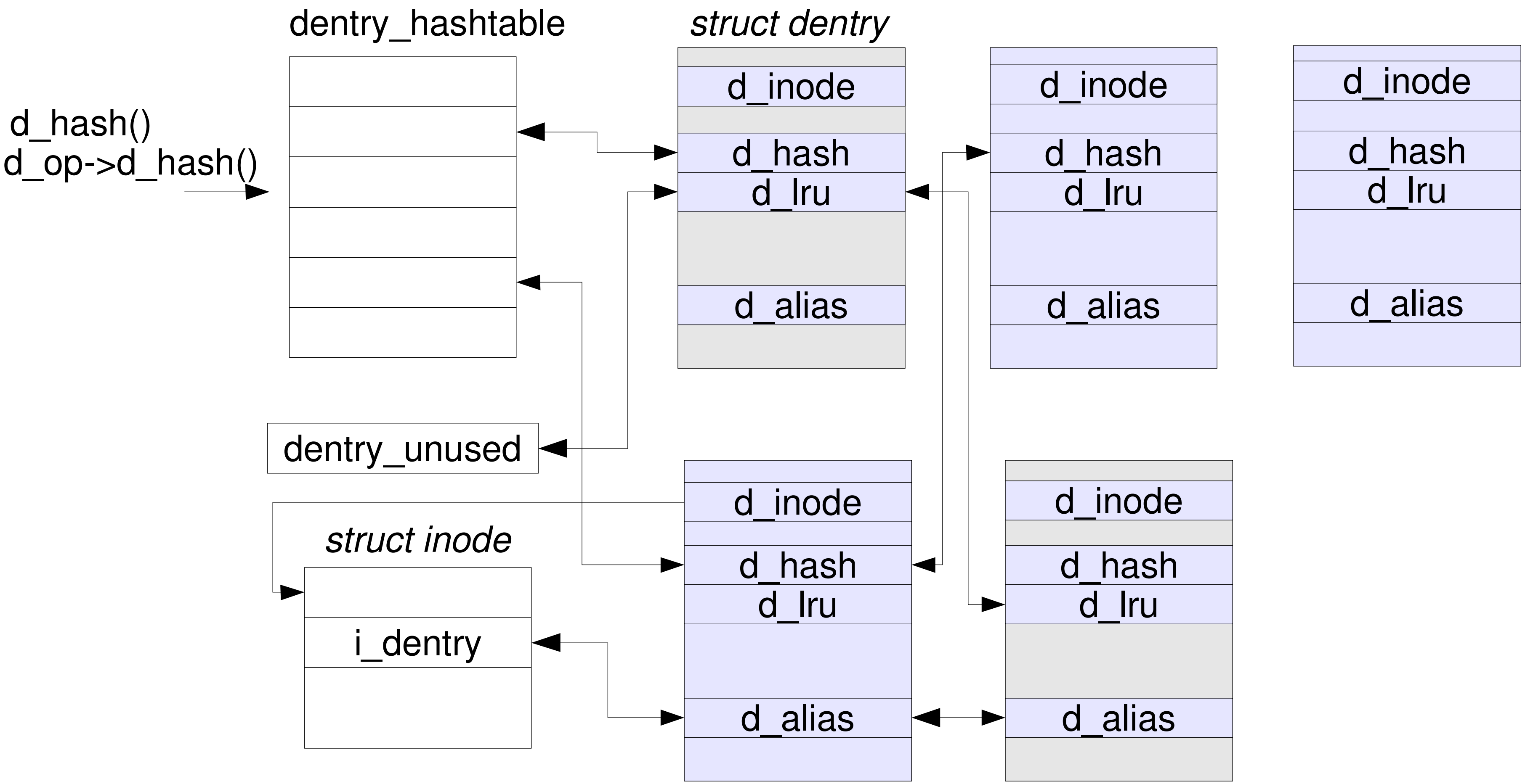
- Tree layout: parent pointer and children list
  - According to the directory layout
  - Through fields d_parent, d_subdirs, d_child
- Dentry hash table
  - For fast lookup from filename to dentry object
  - Hash collision list linked by d_hash field
- List of unused dentry (free list)
  - Through d_lru field
- List of aliases (same inode, different dentries)
  - Through d_alias field

Dentry Lists (Tree Layout)





Dentry Lists (Hash/Free/Alias)



VFS inode

- Kernel data structure for a file (or directory)
  - Each file/directory (reside on disk) is represented by one unique inode number and an inode record on disk
  - Inode number never change (during lifetime of the file)
- To access a file
  - Allocate a VFS inode object in kernel memory
  - Load from the inode record on disk
- Inode cache
  - Recently used (and released) inode will be cached (in slab cache) for performance purpose

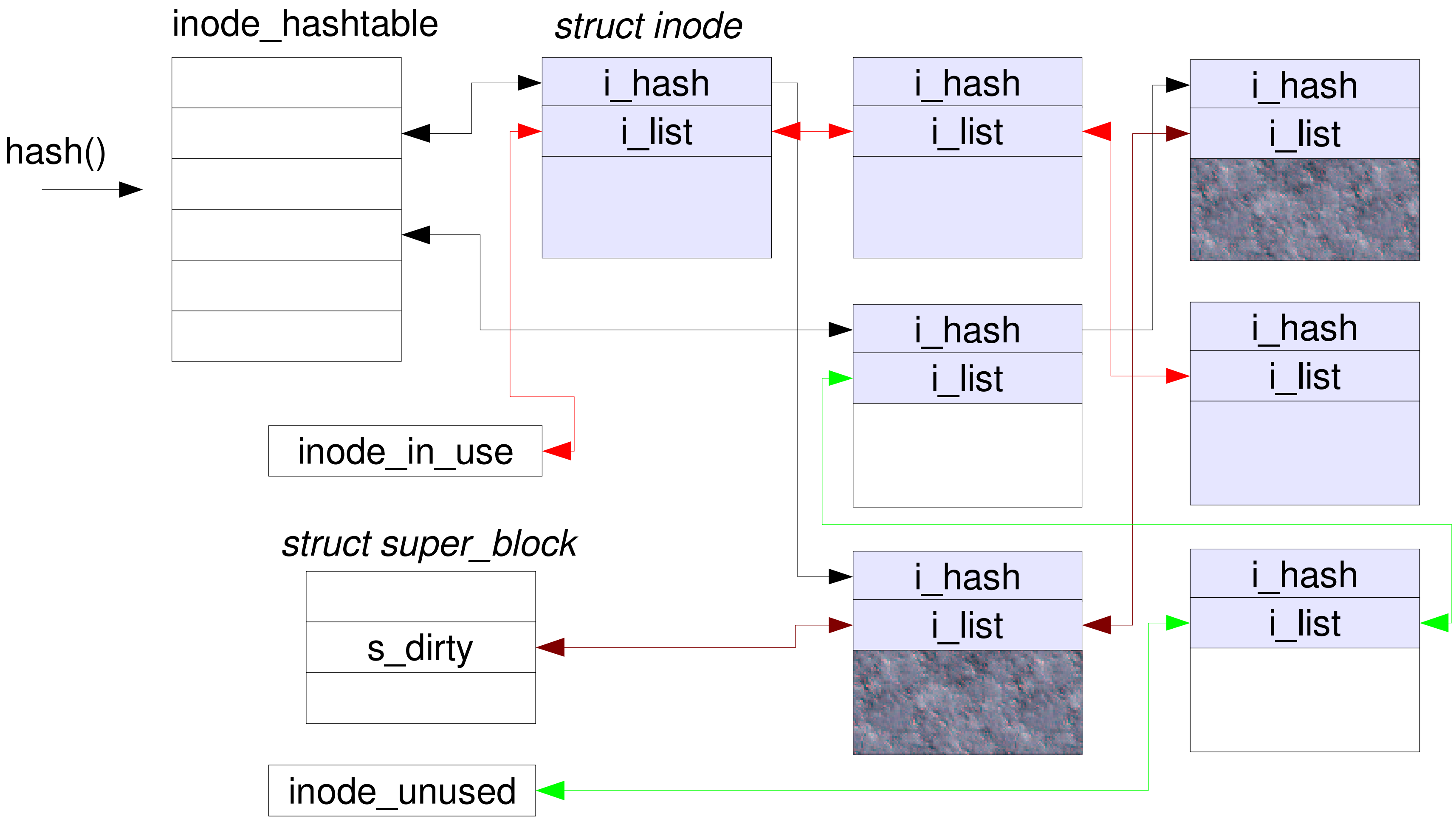
VFS inode Data Structure

- struct inode (in include/linux/fs.h)
- Important fields
  - Inode number & superblock pointer: i_ino, i_sb
  - Access counter (by how many processes): i_count
  - File information: i_mode, i_nlink, i_uid, i_gid, i_size, i_atime, i_mtime, i_ctime, i_blksize, i_blocks
  - Pointer to a set of inode methods: i_op
  - Default file operations: i_fop
  - Inode lists: i_hash, i_list, ...
  - List of dentry(-ies) for this inode: i_dentry

VFS inode Lists

- Inode hash table (for all inodes, in use or released)
  - For fast lookup from inode number to inode object
  - Each inode object is hashed by i_sb and i_ino
  - Hash collision list linked by i_hash field
- Each inode is in one of the three lists
  - In-use list: i_count > 0
  - Ditry list: i_count > 0 with dirty bits set in i_state
  - Unused list: i_count = 0
  - All lists linked by i_list field
  - List heads: inode_in_use, inode_unused (static variables in fs/inode.c) and superblock's s_dirty

VFS inode Lists Illustrated



Inode Methods

- A set of filesystem-dependent operations on inode
  - create(): create an new inode on disk (for a new file)
  - lookup(): look up inode in a directory by filename
  - link(), unlink(): create/remove a hardlink
  - mkdir(), rmdir(): create/delete a directory
  - symlink(), mknod(): create special file
  - And many more (see struct inode_operations in include/linux/fs.h)

VFS file Objects

- Kernel data structure for a task's opened file
  - Data type: struct file (in include/linux/fs.h)
- Important fields
  - f_dentry: dentry object associated with this file
  - f_op: pointer to a set of file operations
  - f_pos: current file pointer (file position/offset)
  - f_count: usage counter (by how many tasks)
  - f_list: to link this file object in one of the several lists
  - private_data: for use in device driver

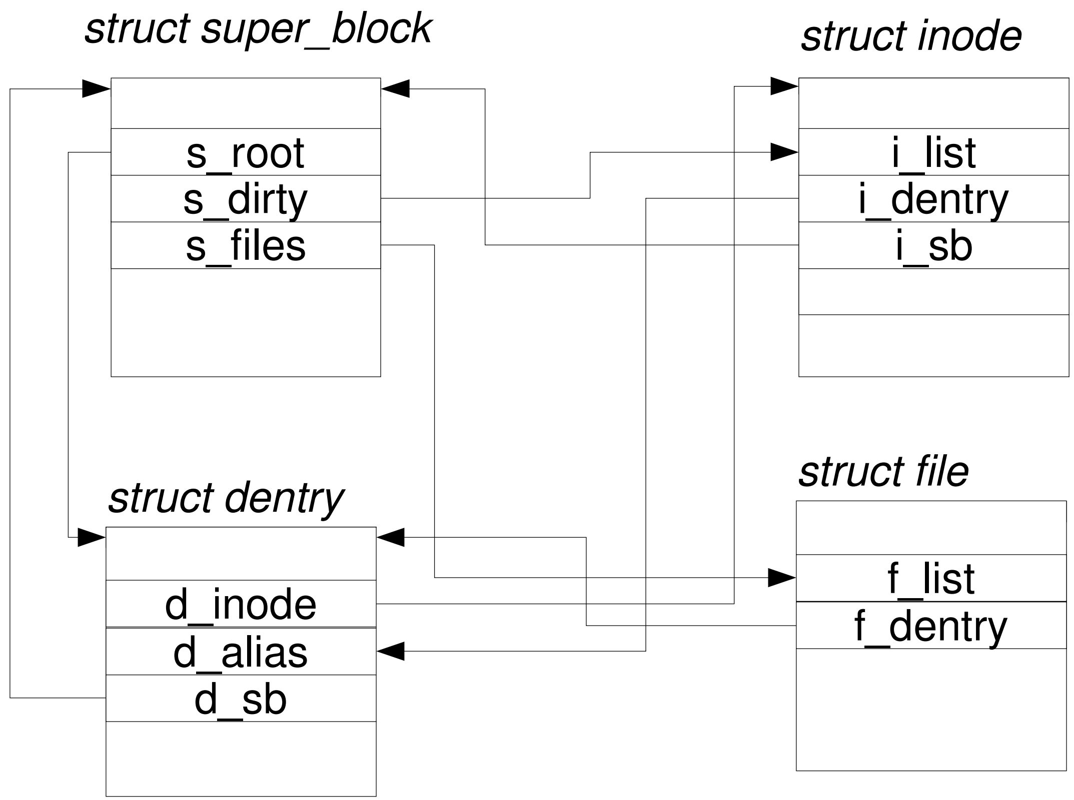
VFS file Lists

- A VFS file object is one of the several lists
  - Each list is chained through the f_list field
- Each superblock keep a list of opened files
  - So that it wouldn't umount if still opened file(s)
  - List header at superblock's s_files field
- Free list
  - Variable free_list in fs/file_table.c
- Anon list
  - When a new file object is created (but not yet opened)
  - Variable anon_list in fs/file_table.c

File Operations

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, 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 (*relase) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        …
}
```

List Review


struct super_block
s_root
s_dirty
s_files
struct inode
i_list
i_dentry
i_sb
struct dentry
d_inode
d_alias
d_sb
struct file
f_list
f_dentry

Access from Task Structure

- Fields in task_struct
- struct fs_struct *fs
  - About the file system this task is running on
  - Data type defined in include/linux/fs_struct.h
  - Has 2 important fields: struct dentry * root, * pwd
- struct files_struct *files
  - About the files this task has opened
  - Data type defined in include/linux/sched.h
  - Has important field: struct file ** fd;
  - To get to an open file from a task: t->files->fd[i]

Summary

- Linux File System:
  - LKP §6
  - ULK §12