# Linux VFS

COMS W4118

Prof. Kaustubh R. Joshi

krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

**References:** Operating Systems Concepts (9e), Linux Kernel Development, Understanding the Linux Kernel 3rd edition (Bovet and Cesati), previous W4118s

# File Systems

- old days – "the" filesystem!
- now – many filesystem types, many instances
  - need to copy file from NTFS to Ext3
- original motivation – NFS support (Sun)
- idea – filesystem op abstraction layer (VFS)
  - Virtual File System (aka Virtual Filesystem Switch)
  - File-related ops determine filesystem type
  - Dispatch (via function pointers) filesystem-specific op

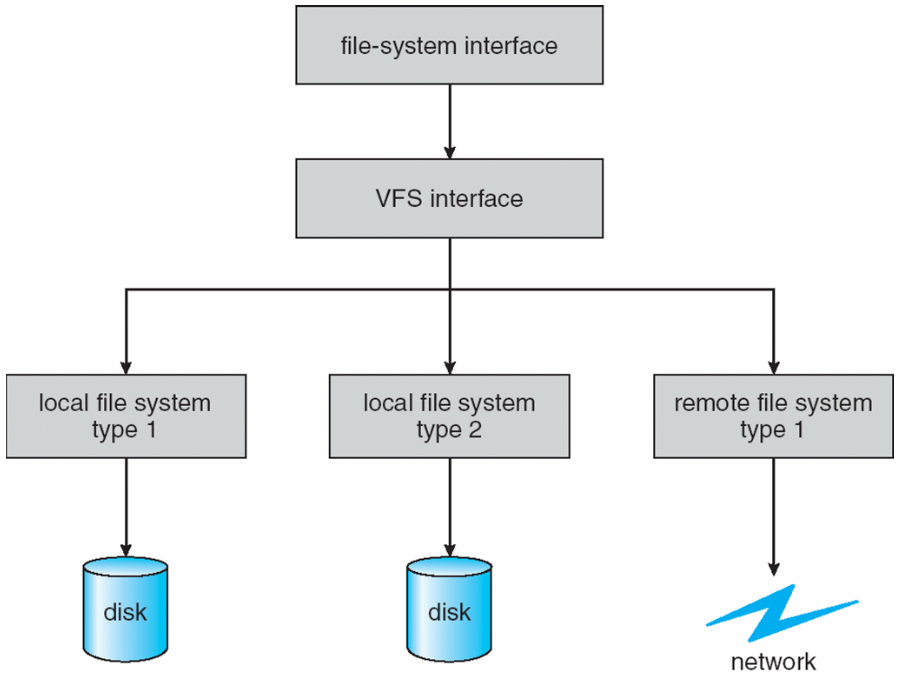
# File System Types

- lots and lots of filesystem types!
  - 2.6 has nearly 100 in the standard kernel tree
- examples
  - standard: Ext2, ufs (Solaris), svfs (SysV), ffs (BSD)
  - network: RFS, NFS, Andrew, Coda, Samba, Novell
  - journaling: Ext3, Veritas, ReiserFS, XFS, JFS
  - media-specific: jffs, ISO9660 (cd), UDF (dvd)
  - special: /proc, tmpfs, sockfs, etc.
- proprietary
  - MSDOS, VFAT, NTFS, Mac, Amiga, etc.
- new generation for Linux
  - Ext3, ReiserFS, XFS, JFS

# (VFS) Virtual File System

- Object-oriented way of implementing FSs
- Same API for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
  - Then dispatches operation to appropriate file system implementation routines
- Syscalls program to VFS API rather than specific FS interface

# Linux Virtual File System (VFS)




- Very flexible use cases:
  - User files remote and system files local? No problem.
  - Boot from USB? Network? RAM? No problem.
  - Boot from another file? No problem.
  - Interesting FSes: sshfs, gmailfs, FUSE (user space FS)

# VFS Stakeholders

- VFS Objects
  - inode, file, superblock, dentry
  - VFS defines which ops on each object
  - Each object has a pointer to a function table
    - Addresses of routines to implement that function on that object
- VFS Users
  - System calls that provide file related services
  - Use VFS function pointer and objects only
- VFS Implementers
  - File systems that translate VFS ops into native operations
  - Store on disk, send over network, etc.
  - Provide the functions pointer to by function pointers

# Linux File System Model

- basically UNIX file semantics
  - File systems are mounted at various points
  - Files identified by device inode numbers
- VFS layer just dispatches to fs-specific functions
  - libc read() -> sys_read()
    - what type of filesystem does this file belong to?
    - call filesystem (fs) specific read function
    - maintained in open file object (file)
  - example: file->f_op->read(…)
- similar to device abstraction model in UNIX

# VFS Users

- fundamental UNIX abstractions
  - files (everything is a file)
    - ex: /dev/ttyS0 – device as a file
    - ex: /proc/123 – process as a file
  - processes
  - users
- lots of syscalls related to files! (~100)
  - most dispatch to filesystem-specific calls
  - some require no filesystem action
    - example: lseek(pos) – change position in file
  - others have default VFS implementations

# VFS System Calls

- filesystem ops – mounting, info, flushing, chroot, pivot_root
- directory ops – chdir, getcwd, link, unlink, rename, symlink
- file ops – open/close, (p)read(v)/(p)write(v), seek, truncate, dup fcntl, creat,
- inode ops – stat, permissions, chmod, chown
- memory mapping files – mmap, munmap, madvise, mlock
- wait for input – poll, select
- flushing – sync, fsync, msync, fdatasync
- file locking – flock

# VFS-related Task Fields

- task_struct fields
  - fs – includes root, pwd
    - pointers to dentries
  - files – includes file descriptor array fd[]
    - pointers to open file objects

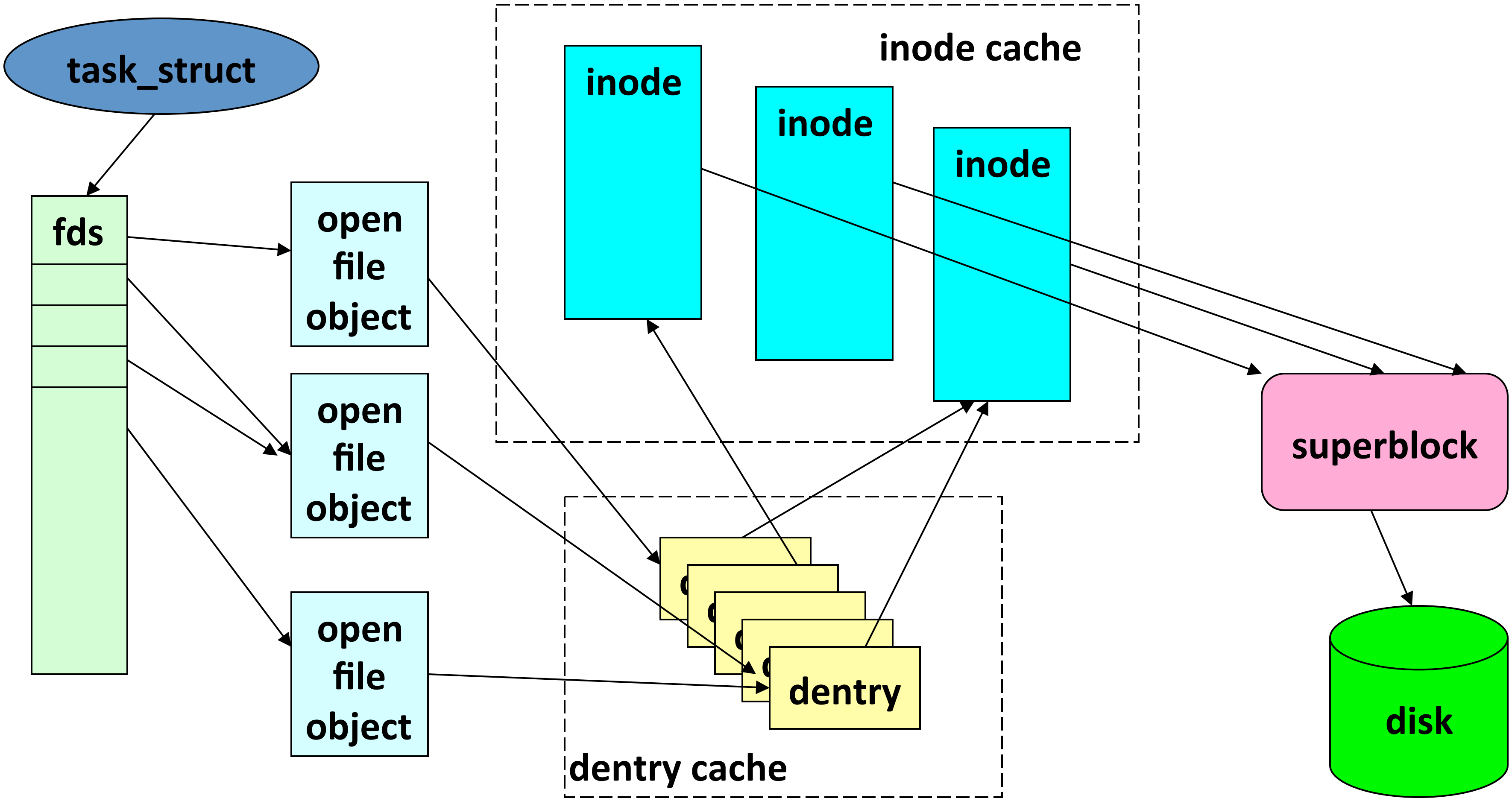
# VFS Objects: The Big Four

- **struct file**
  - information about an open file
  - includes current position (file pointer)
- **struct dentry**
  - information about a directory entry
  - includes name + inode#
- **struct inode**
  - unique descriptor of a file or directory
  - contains permissions, timestamps, block map (data)
  - inode#: integer (unique per mounted filesystem)
- **struct** superblock
  - descriptor of a mounted filesystem

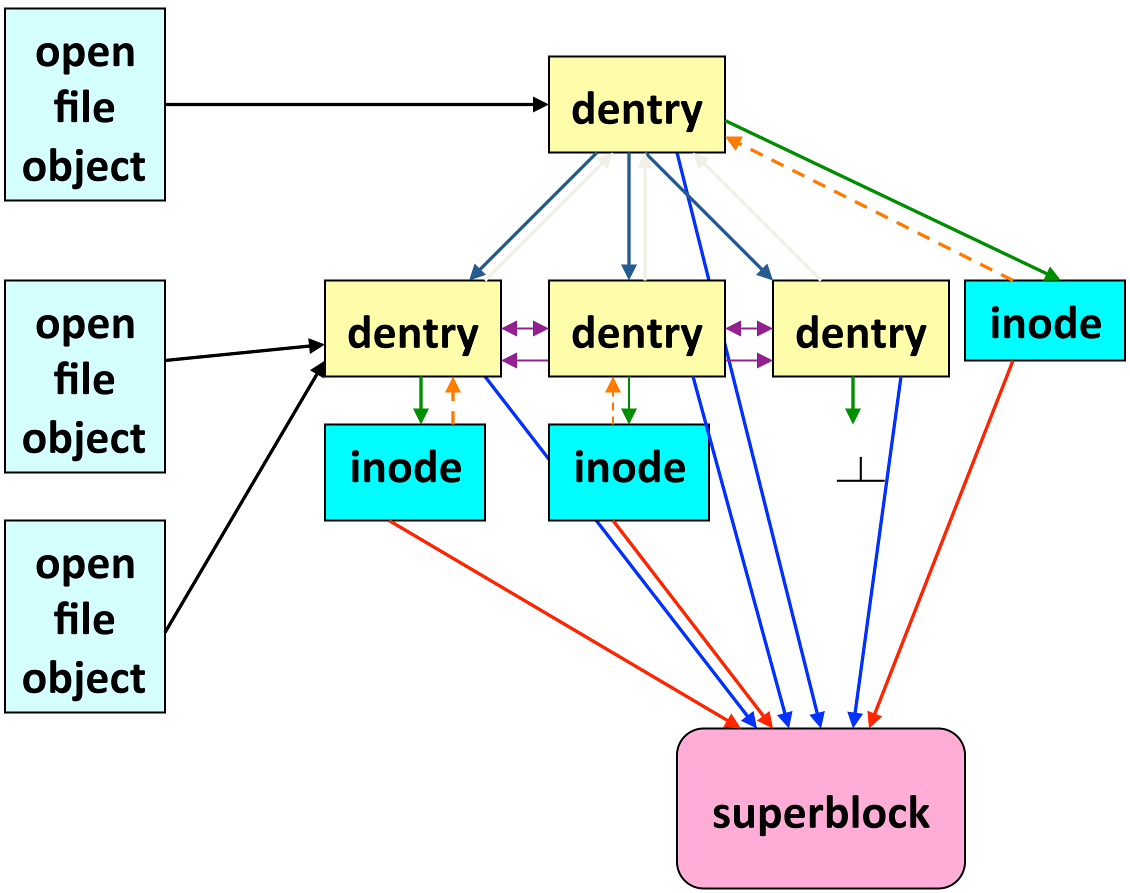
# Two More Data Structures

- struct file_system_type
  - name of file system
  - pointer to implementing module
  - including how to read a superblock
  - On module load, you call register_file_system and pass a pointer to this structure
- struct vfsmount
  - Represents a mounted instance of a particular file system
  - One super block can be mounted in two places, with different covering sub mounts
  - Thus lookup requires parent dentry and a vfsmount

# Data Structure Relationships



# Data Structure Relationships




| | |
|---|---|
| f_dentry | → |
| d_subdirs | → |
| d_inode | → |
| d_subdirs | ↔ |
| d_parent | → |
| i_sb | → |
| d_sb | → |
| i_dentries | ⇢ |

# Sharing Data Structures

- calling dup() –
  - shares open file objects
  - example: 2>&1
- opening the same file twice –
  - shares dentries
- opening same file via different hard links –
  - shares inodes
- mounting same filesystem on different dirs –
  - shares superblocks

# Superblock

- mounted filesystem descriptor
  - usually first block on disk (after boot block)
  - copied into (similar) memory structure on mount
    - distinction: disk superblock vs memory superblock
    - dirty bit (s_dirt), copied to disk frequently
- important fields
  - s_dev, s_bdev – device, device-driver
  - s_blocksize, s_maxbytes, s_type
  - s_flags, s_magic, s_count, s_root, s_dquot
  - s_dirty – dirty inodes for this filesystem
  - s_op – superblock operations
  - u – filesystem specific data

# Superblock Operations

- filesystem-specific operations
  - read/write/clear/delete inode
  - write_super, put_super (release)
    - no get_super()! that lives in file_system_type descriptor
  - write_super_lockfs, unlockfs, statfs
  - file_handle ops (NFS-related)
  - show_options

# Inode

- "index" node – unique file or directory descriptor
  – meta-data: permissions, owner, timestamps, size, link count
  – data: pointers to disk blocks containing actual data
    - data pointers are "indices" into file contents (hence "inode")
- inode # - unique integer (per-mounted filesystem)
- what about names and paths?
  – high-level fluff on top of a "flat-filesystem"
  – implemented by directory files (directories)
  – directory contents: name + inode

# File Links

- UNIX link semantics
  - hard links – multiple dir entries with same inode #
    - equal status; first is not "real" entry
    - file deleted when link count goes to 0
    - restrictions
      - can't hard link to directories (avoids cycles)
      - or across filesystems
  - soft (symbolic) links – little files with pathnames
    - just aliases for another pathname
    - no restrictions, cycles possible, dangling links possible

# Inode Fields

- large struct (~50 fields)
- important fields
  - i_sb, i_ino (number), i_nlink (link count)
  - metadata: i_mode, i_uid, i_gid, i_size, i_times
  - i_flock (lock list), i_wait (waitq – for blocking ops)
  - linkage: i_hash, i_list, i_dentry (aliases)
  - i_op (inode ops), i_fop (default file ops)
  - u (filesystem specific data – includes block map!)

# Inode Operations

- create – new inode for regular file
- link/unlink/rename –
  – add/remove/modify dir entry
- symlink, readlink, follow_link – soft link ops
- mkdir/rmdir – new inode for directory file
- mknod – new inode for device file
- truncate – modify file size
- permission – check access permissions

# (Open) File Object

- struct file (usual variable name - filp)
  - association between file and process
  - no disk representation
  - created for each open (multiple possible, even same file)
  - most important info: file pointer
- file descriptor (small ints)
  - index into array of pointers to open file objects
- file object states
  - unused (memory cache + root reserve (10))
    - get_empty_filp()
  - inuse (per-superblock lists)
- system-wide max on open file objects (~8K)
  - /proc/sys/fs/file-max

# File Object Fields

- important fields
  - f_dentry (directory entry of file)
  - f_vfsmnt (fs mount point)
  - f_op (fs-specific functions – table of function pointers)
  - f_count, f_flags, f_mode (r/w, permissions, etc.)
  - f_pos (current position – file pointer)
  - info for read-ahead (more later)
  - f_uid, f_gid, f_owner
  - f_version (for consistency maintenance)
  - private_data (fs-specific data)

# File Object Operations

- f_op field – table of function pointers
  - copied from inode (i_fop) initially (fs-specific)
  - possible to change to customize (per-open)
    - device-drivers do some tricks like this sometimes
- important operations
  - llseek(), read(), write(), readdir(), poll()
  - ioctl() – "wildcard" function for per-fs semantics
  - mmap(), open(), flush(), release(), fsync()
  - fasync() – turn on/off asynchronous i/o notifications
  - lock() – file-locks (more later)
  - readv(), writev() – "scatter/gather i/o"
    - read/write with discontiguous buffers (e.g. packets)
  - sendpage() – page-optimized socket transfer

# Dentry

- abstraction of directory entry
  - ex: line from ls -l
  - either files (hard links) or soft links or subdirectories
  - every dentry has a parent dentry (except root)
  - sibling dentries – other entries in the same directory
- directory api: dentry iterators
  - posix: opendir(), readdir(), scandir(), seekdir(), rewinddir()
  - syscall: getdents()
- why an abstraction?
  - Local filesystems: directories are really files with directory "records”
  - Network filesystems: often have separate directory operations (e.g., NFS, FTP)
  - Having abstraction allows unification, caching

# Dentry (continued)

- not-disk based (no dirty bit)
  - dentry_cache – slab cache
- important fields
  - d_name (qstr), d_count, d_flags
  - d_inode – associated inode
  - d_parent – parent dentry
  - d_child – siblings list
  - d_subdirs – my children (if i'm a subdirectory)
  - d_alias – other names (links) for the same object (inode)?
  - d_lru – unused state linkage
  - d_op – dentry operations (function pointer table)
  - d_fsdata – filesystem-specific data

# Dentry Cache

- very important cache for filesystem performance
  - every file access causes multiple dentry accesses!
  - example: /tmp/foo
    - dentries for "/", "/tmp", "/tmp/foo" (path components)
- dentry cache "controls" inode cache
  - inodes released only when dentry is released
- dentry cache accessed via hash table
  - hash(dir, filename) -> dentry

# Dentry Cache (continued)

- dentry states
  - free (not valid; maintained by slab cache)
  - in-use (associated with valid open inode)
  - unused (valid but not being used; LRU list)
  - negative (file that does not exist)
- dentry ops
  - just a few, mostly default actions
  - ex: d_compare(dir, name1, name2)
    - case-insensitive for MSDOS

# Process-related Files

- current->fs (fs_struct)
  - root (for chroot jails)
  - pwd
  - umask (default file permissions)
- current->files (files_struct)
  - fd[] (file descriptor array – pointers to file objects)
    - 0, 1, 2 – stdin, stdout, stderr
  - originally 32, growable to 1,024 (RLIMIT_NOFILE)
    - complex structure for growing … see book
  - close_on_exec memory (bitmap)
    - open files normally inherited across exec

# Filesystem Types

- Linux must "know about" filesystem before mount
  - multiple (mounted) instances of each type possible
- special (virtual) filesystems (like /proc)
  - structuring technique to touch kernel data
  - examples:
    - /proc, /dev (devfs)
    - sockfs, pipefs, tmpfs, rootfs, shmfs
  - associated with fictitious block device (major# 0)
    - minor# distinguishes special filesystem types

# Registering a Filesystem Type

- must register before mount
  – static (compile-time) or dynamic (modules)
- register_filesystem() / unregister_filesystem
  – adds file_system_type object to linked-list
    - file_systems (head; kernel global variable)
    - file_systems_lock (rw spinlock to protect list)
- file_system_type descriptor
  – name, flags, pointer to implementing module
  – list of superblocks (mounted instances)
  – read_super() – pointer to method for reading superblock
    - most important thing! filesystem specific

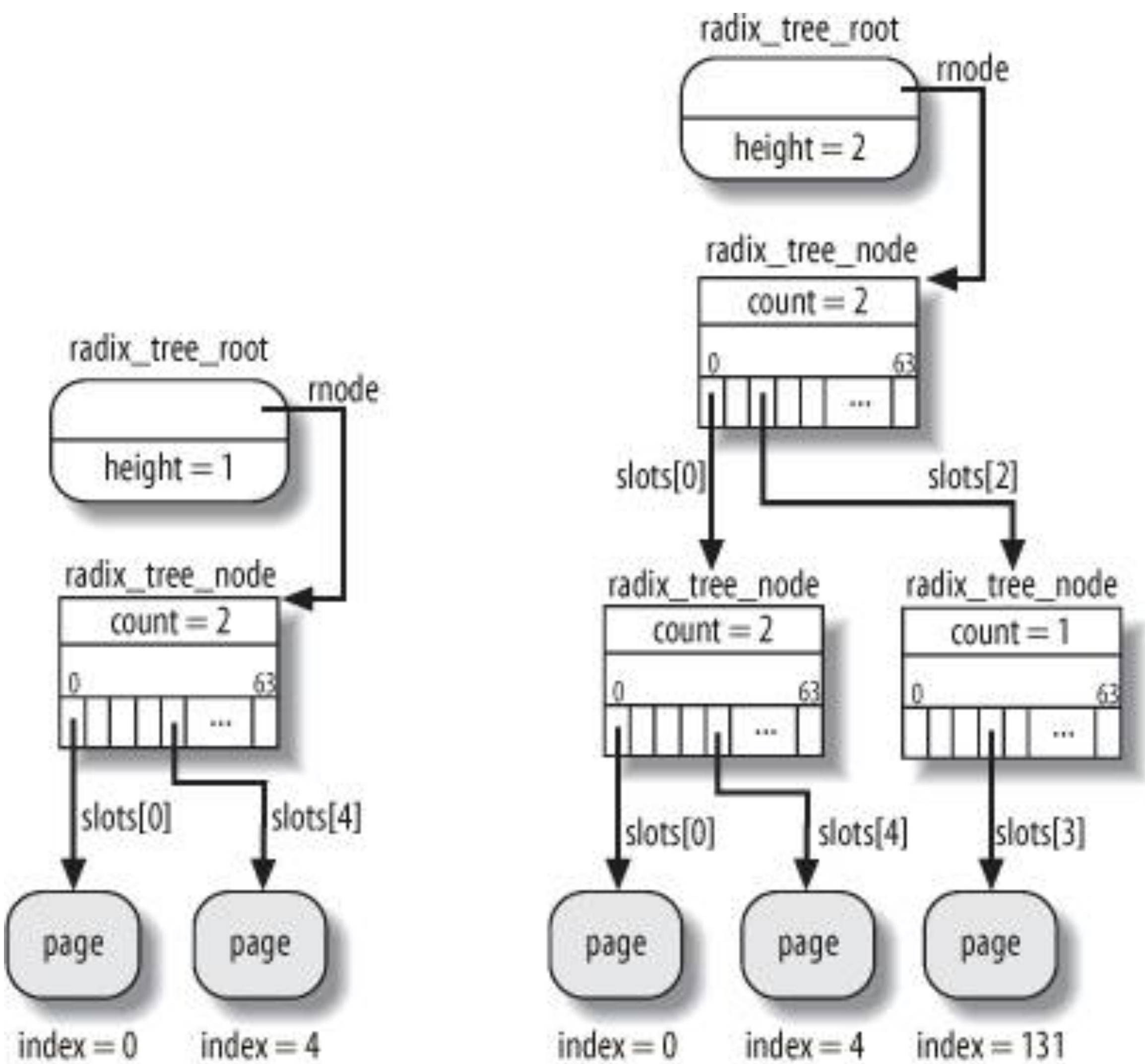
# Integration with Memory Subsystem

- The address_space structure
  - One per file, device, etc.
  - Mapping between logical offset in object to page in memory
  - Pages in memory are called "page cache"
  - Files can be large: need efficient data structure
- You don't have to use address_space for hw4. Use a simple array to maintain your offset->page mapping.

# The address_space structure

```
struct address_space {
        struct inode            *host;          /* owner: inode, block_device */
        struct radix_tree_root  page_tree;      /* radix tree of all pages */
        spinlock_t              tree_lock;      /* and lock protecting it */
        unsigned int            i_mmap_writable;/* count VM_SHARED mappings */
        struct prio_tree_root   i_mmap;         /* tree of private and shared 
mappings */
        struct list_head        i_mmap_nonlinear;/*list VM_NONLINEAR mappings */
        spinlock_t              i_mmap_lock;    /* protect tree, count, list */
unsigned long           nrpages;        /* number of total pages */
        pgoff_t                 writeback_index;/* writeback starts here */
        const struct address_space_operations *a_ops;   /* methods */
        unsigned long           flags;          /* error bits/gfp mask */
        struct backing_dev_info *backing_dev_info; /* device readahead, etc */
}
```

# The Page Cache Radix Tree




(a) radix tree of height 1 (b) radix tree of height 2

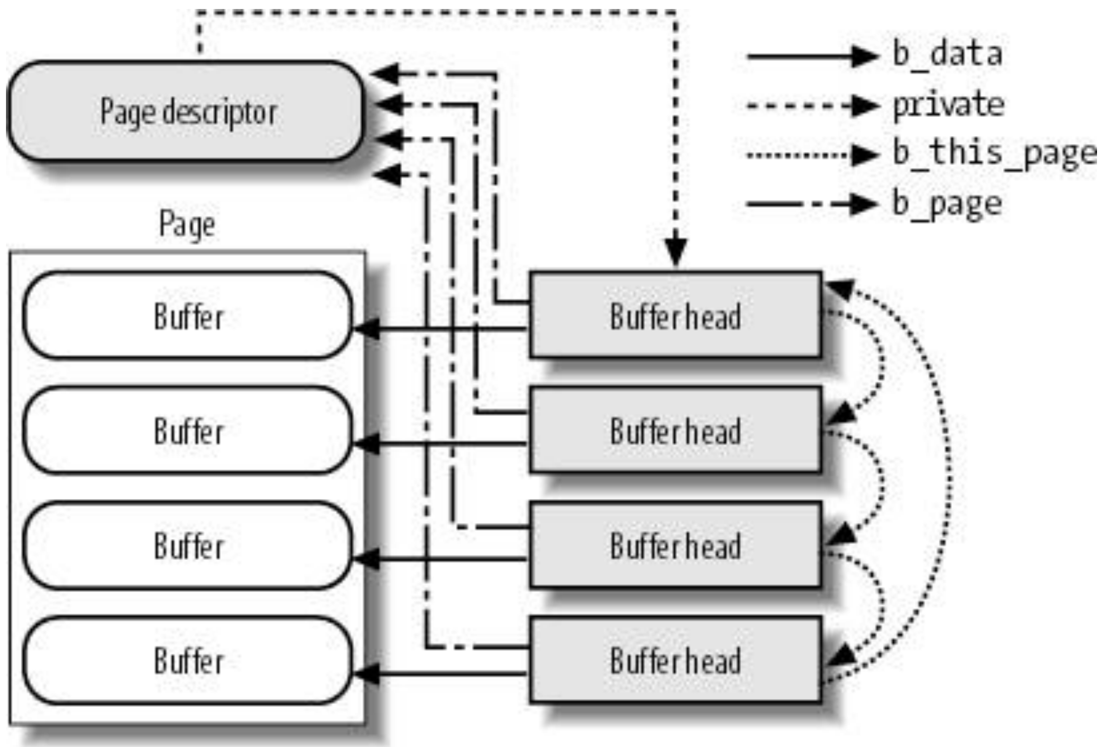
# address_space_operations structure

```
struct address_space_operations {
     int (*writepage)(struct page *page, struct writeback_control *wbc);
     int (*readpage)(struct file *, struct page *);

     int (*write_begin)(struct file *, struct address_space *mapping,
                              loff_t pos, unsigned len, unsigned flags,
                              struct page **pagep, void **fsdata);
     int (*write_end)(struct file *, struct address_space *mapping,
                              loff_t pos, unsigned len, unsigned copied,
                              struct page *page, void *fsdata);

     sector_t (*bmap)(struct address_space *, sector_t);
     void (*invalidatepage) (struct page *, unsigned long);
}
```

# Buffer Cache Descriptors



# Reverse Mapping for Memory Maps

- Problem: anon_vma is good for limited sharing
  - Memory maps can be shared by large numbers of processes
  - E.g., libc shared by everyone
  - I.e., need to do linear search for every eviction
  - Also, different processes may map different ranges of a memory map into their address space
- Need efficient data structure
  - Basic operation: given an offset in an object (such as a file), or a range of offsets, return vmas that map that range
  - Enter priority search trees
  - Allows efficient interval queries
- Note: you don't need this for hw4. Use anon_vma

# i_mmap Priority Tree

Part of struct address_space in fs.h