VIRTUAL FILE SYSTEM AND FILE SYSTEM CONCEPTS

2016 Operating Systems Design
Euiseong Seo (euiseong@skku.edu)

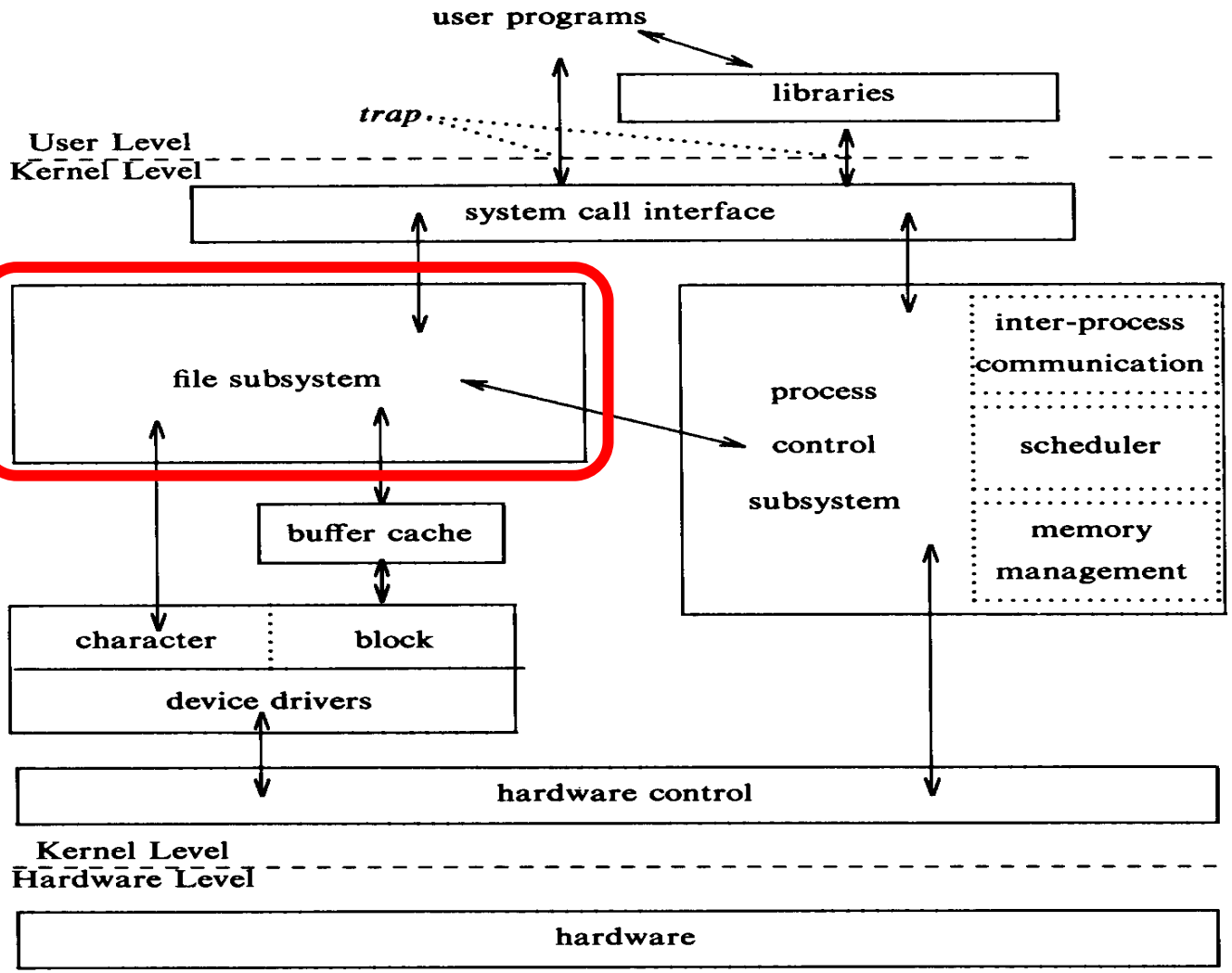
File Layout

- An entity that separates and isolates data
- Files have meanings only to applications
- Operating systems may be aware of data contexts
  - Structured file format
- Let applications handle internals
  - Unstructured file format
  - Just treat a file as a stream of bytes
- All UNIX-variants see a file as a stream of bytes

File Systems

- An OS subsystem that controls how data is stored and retrieved
- Aspects of a file system
  - Space management
  - File management
  - Directory management
  - Utilities
  - User-interface
  - Reliability and security provision

File System in Kernel



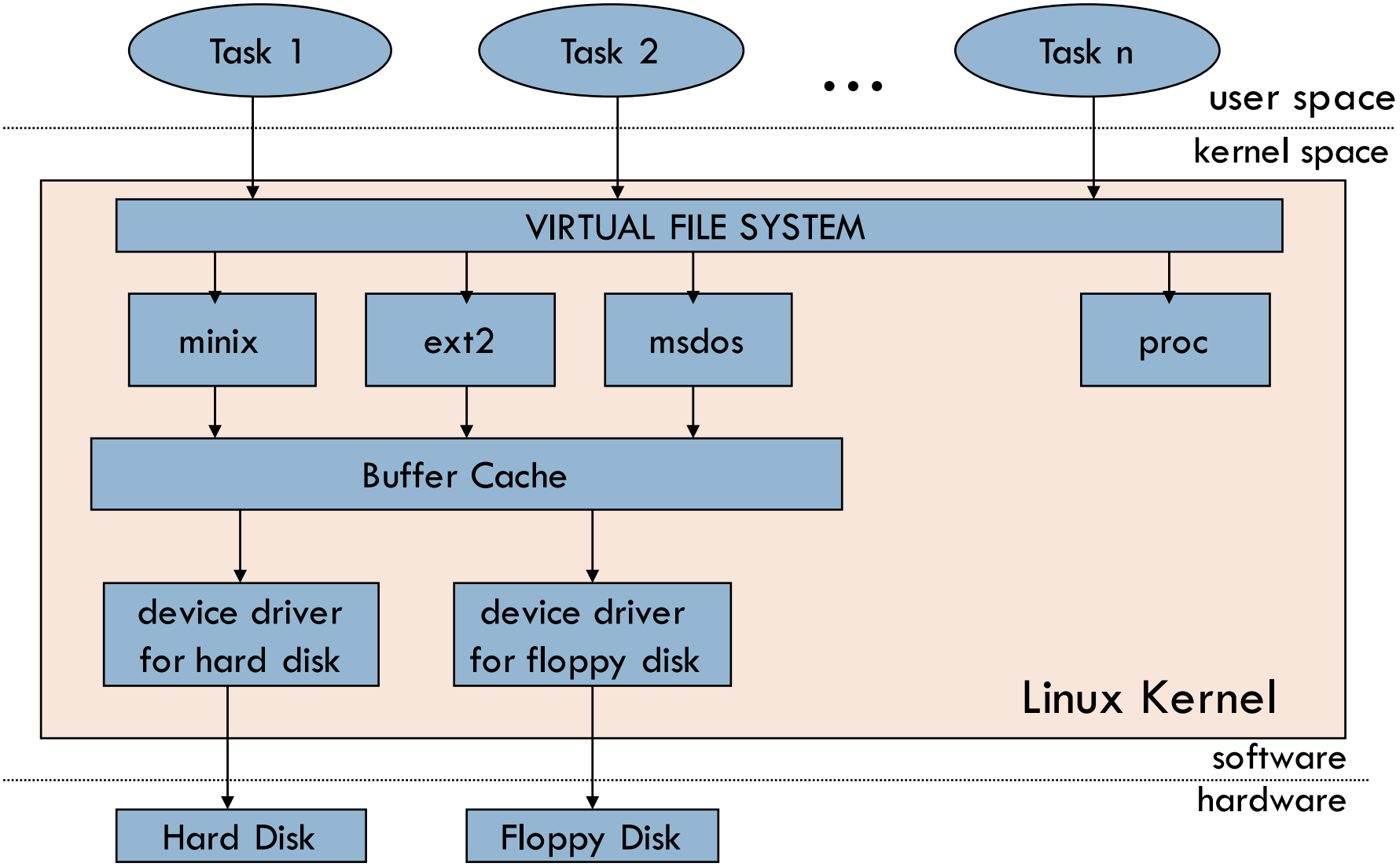
General Purpose File System Variations

- Many kinds of file systems for diverse purposes
- Disk-based file systems
  - Ext4, Ext3, FAT-32 and so on
- Flash-based file systems
  - UBIFS, F2FS and so on
- Tape file systems
- Network file systems
  - NFS, AFS, SMB and so on
- Special file systems
  - Device file system
  - Proc file system

Virtual File System Idea

- Multiple file systems need to coexist
- But file systems share a core of common concepts and high-level operations
- So can create a filesystem abstraction
- Applications interact with this VFS
- Kernel translates abstract-to-actual

Virtual File System



Common File Model

- VFS introduces a common file model to represent all supported filesystems
- Common file model is specifically geared toward Unix filesystems
  - All other filesystems must map their own concepts into the common file model
  - For example, FAT filesystems do not have inodes
- The main components of common file model are
  - superblock (information about mounted filesystem)
  - inode (information about a specific file)
  - file (information about an open file)
  - dentry (information about directory entry)

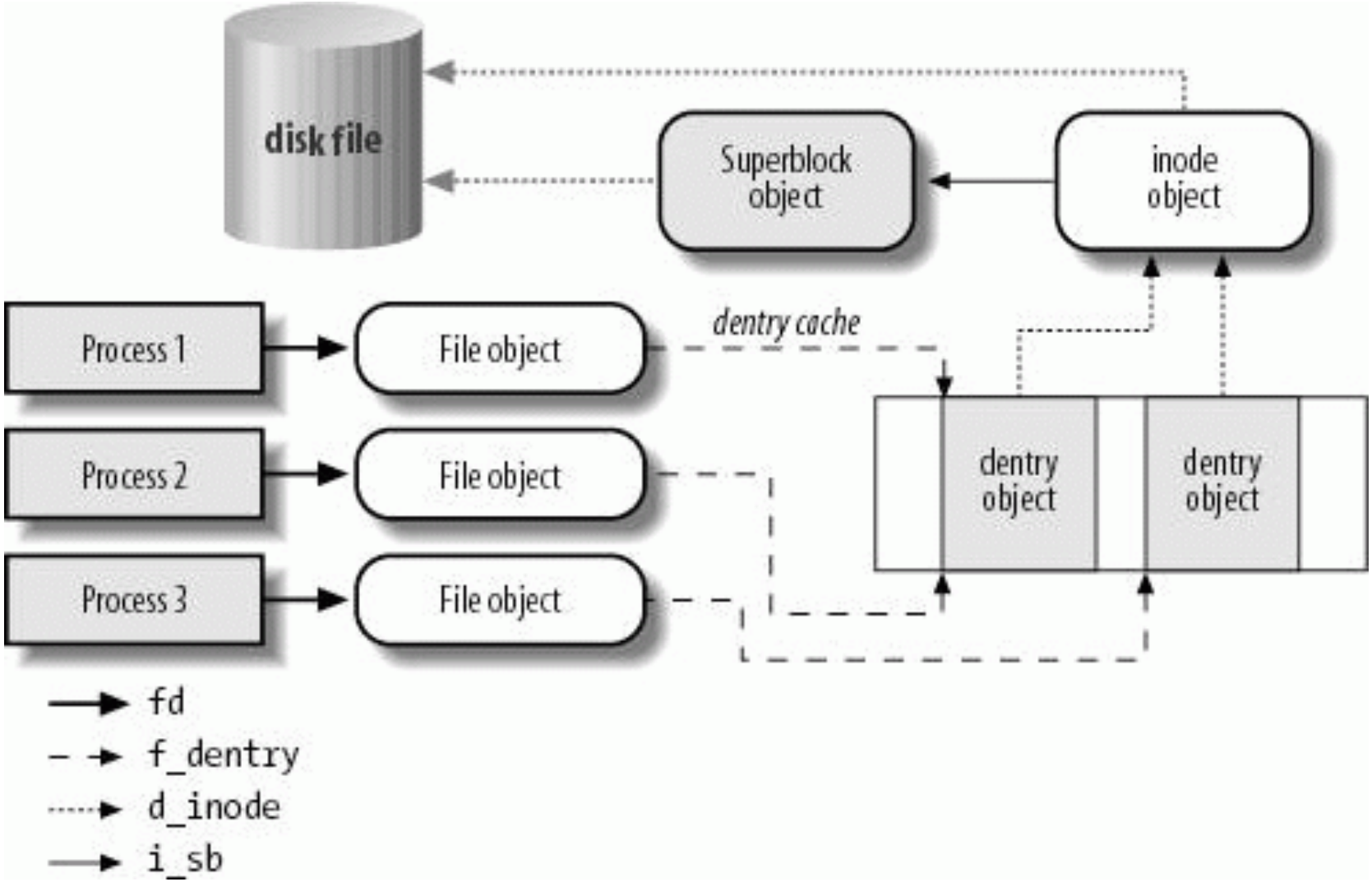
Object-Oriented Approach of VFS

- Each concept object has a set of defined operations that can be performed on the object (i.e., methods)
- VFS provides certain generic implementations for some operations
- Specific filesystem implementations must provide implementation specific operations definitions (i.e., inheritance and method overloading)
- There are no objects in C, though, so a table of function pointers is used for each object to provide its own version of the specific operations

VFS Call Path Example

- read() system call
  - read() invokes sys_read()
  - A open file is represented by a *file* data structure
  - File data structure contains a field, *f_op*, that contains pointers to functions specific to file system files
  - *file->f_op->read()*

VFS Call Path Example



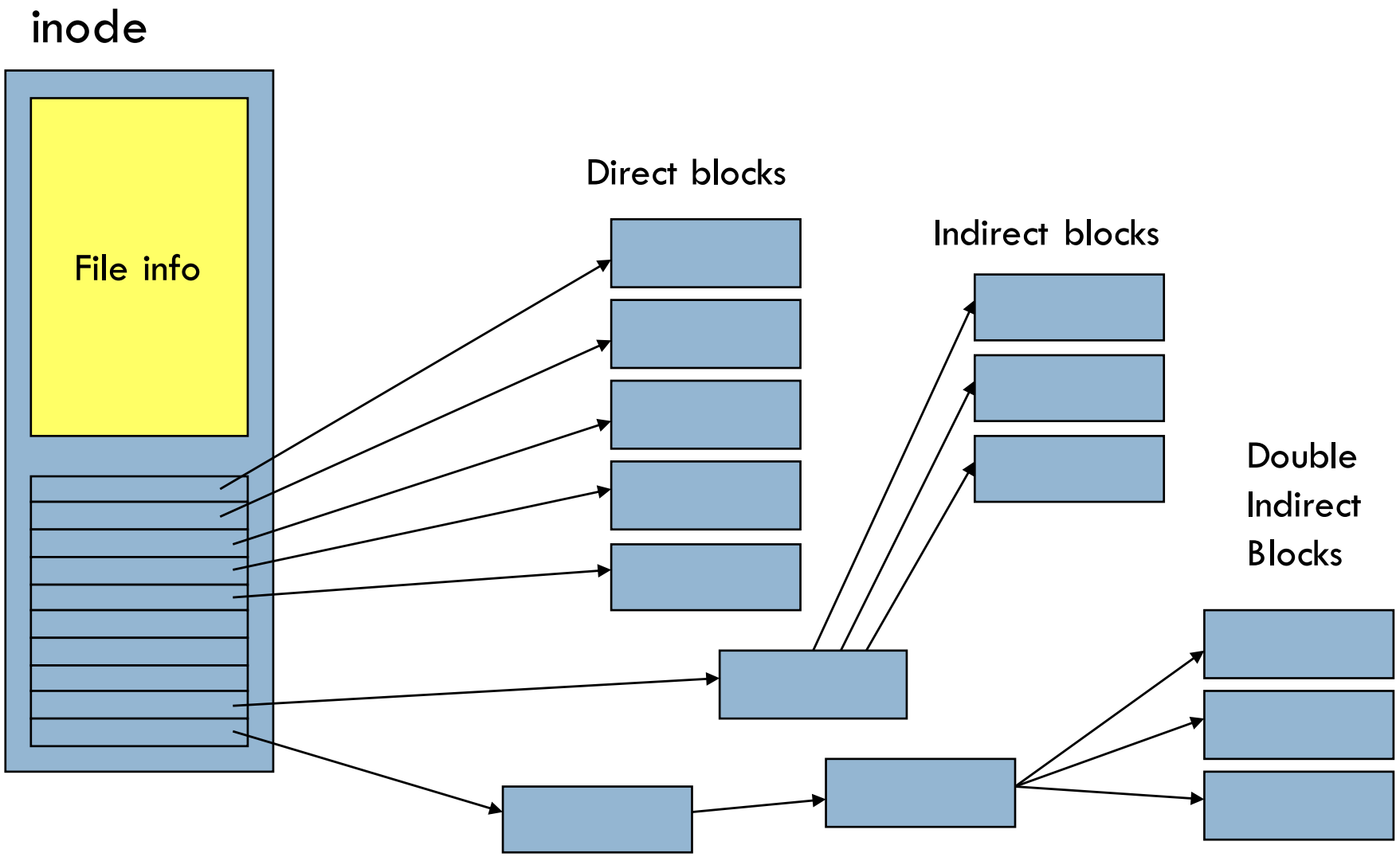
fstatfs Example

```
sys_fstatfs(fd, buf) {                  /* for things like “df” */
    file = fget(fd);
    if ( file == NULL ) return -EBADF;

    superb = file->f_dentry->d_inode->i_super;

    rc = superb->sb_op->sb_statfs(sb, buf);
    return rc;
}
```

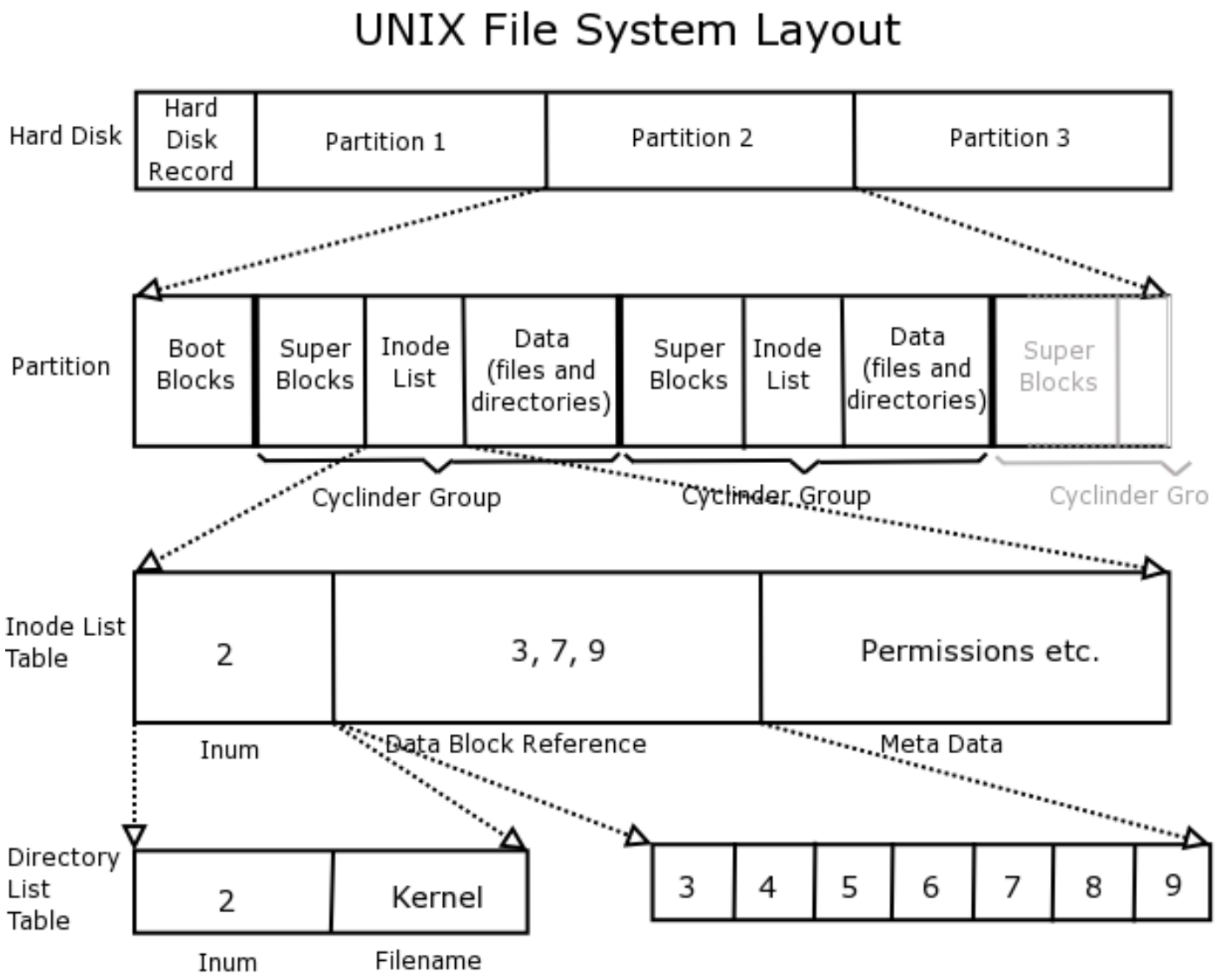
inode Details

- A structure that contains file's description:
  - Type
  - Access rights
  - Owners
  - Timestamps
  - Size
  - Pointers to data blocks
- Kernel keeps the inode in memory (open)

inode Details



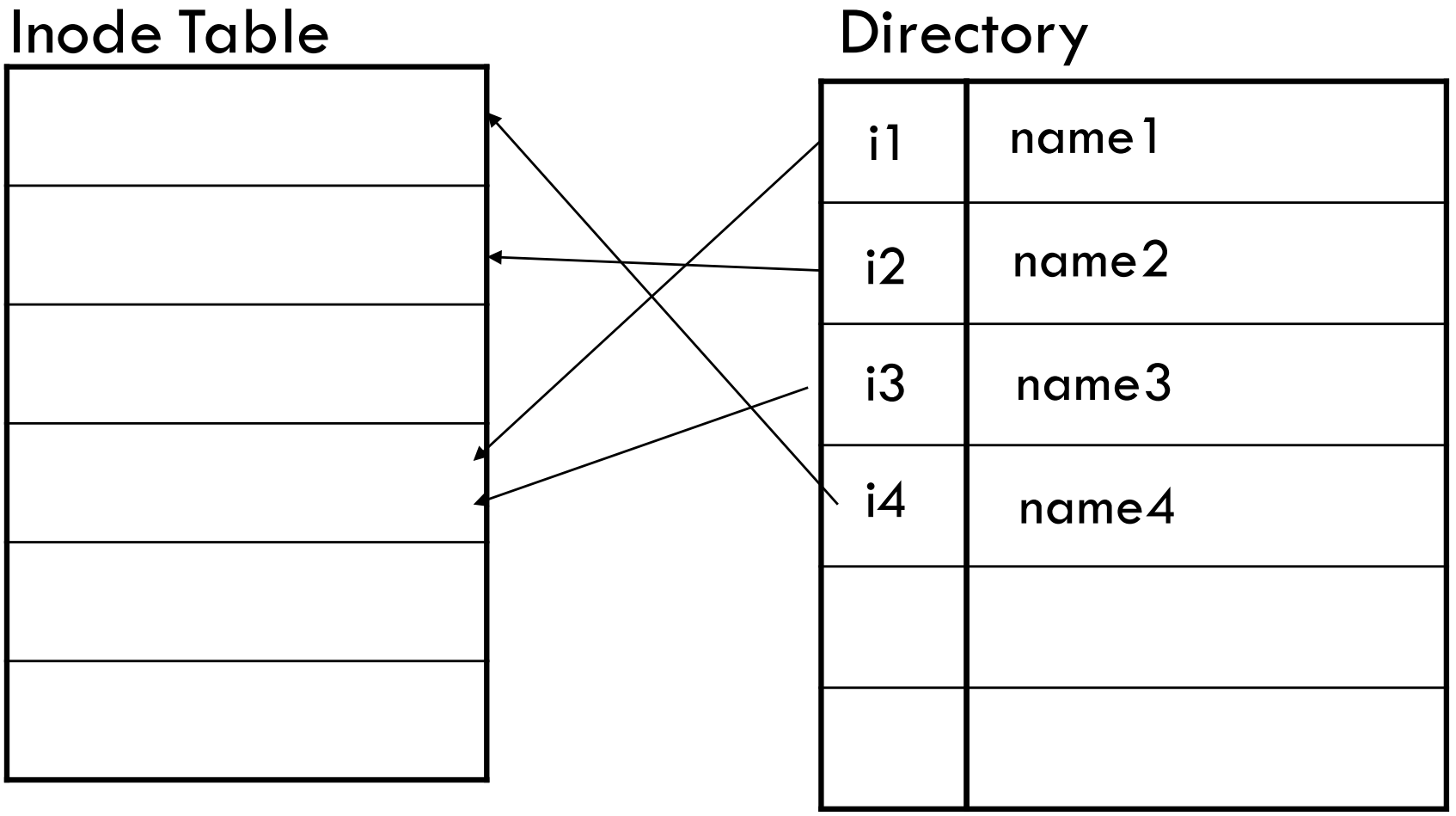
Brief File System Layout



Directory Details

- inodes are structured in a tree hierarchy
- Each can contain both files and directories
- A directory is just a special type of file
- Special user-functions for directory access
- Each dentry contains filename + inode number
- Kernel searches the directory tree and translates a pathname to an inode number

Directory Illustration



Links

- Multiple names can point to same inode
- The inode keeps track of how many links
- If a file gets deleted, the inode's link-count gets decremented by the kernel
- File is deallocated if link-count reaches 0
- This type of linkage is called a 'hard' link
- Hard links may exist only within a single FS
- Hard links cannot point to directories (cycles)

Symbolic Links

- Another type of file linkage ('soft' links)
- Special file, consisting of just a filename
- Kernel uses name-substitution in search
- Soft links allow cross-filesystem linkage
- But they do consume more disk storage

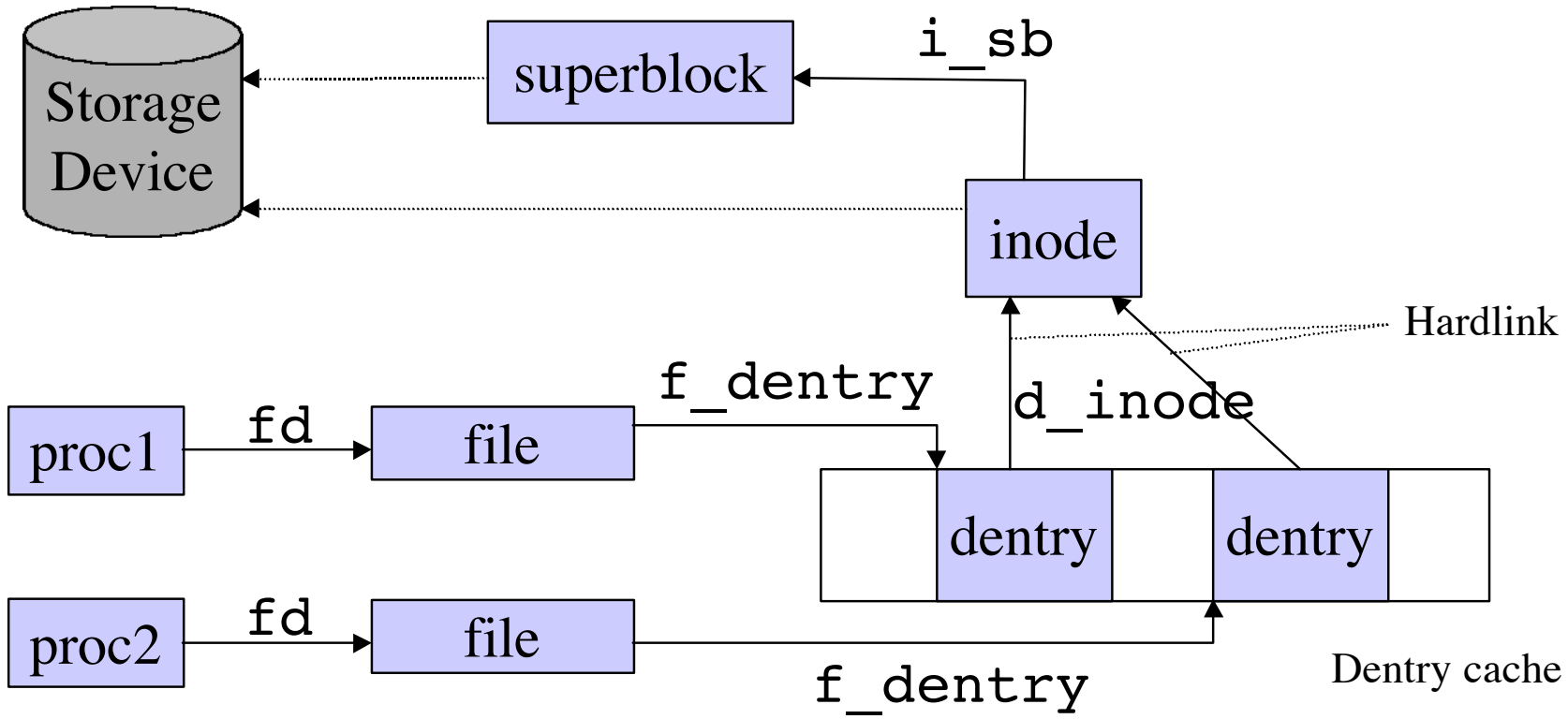
Dentry Cache

- Reading a directory entry requires considerable time
  - Dentry object = directory entry
- It is clever to keep dentry object that you've finished with but might need later
- Dentry cache
  - A set of dentry objects in the in-use, unused or negative state
  - A hash table to derive dentry object associated with a given filename

Stat System Call Example

```
sys_stat(path, buf) {
    dentry = namei(path);
    if ( dentry == NULL ) return -ENOENT;

    inode = dentry->d_inode;
    rc =inode->i_op->i_permission(inode);
    if  ( rc ) return -EPERM;

    rc = inode->i_op->i_getattr(inode, buf);
    dput(dentry);
    return rc;
}
```

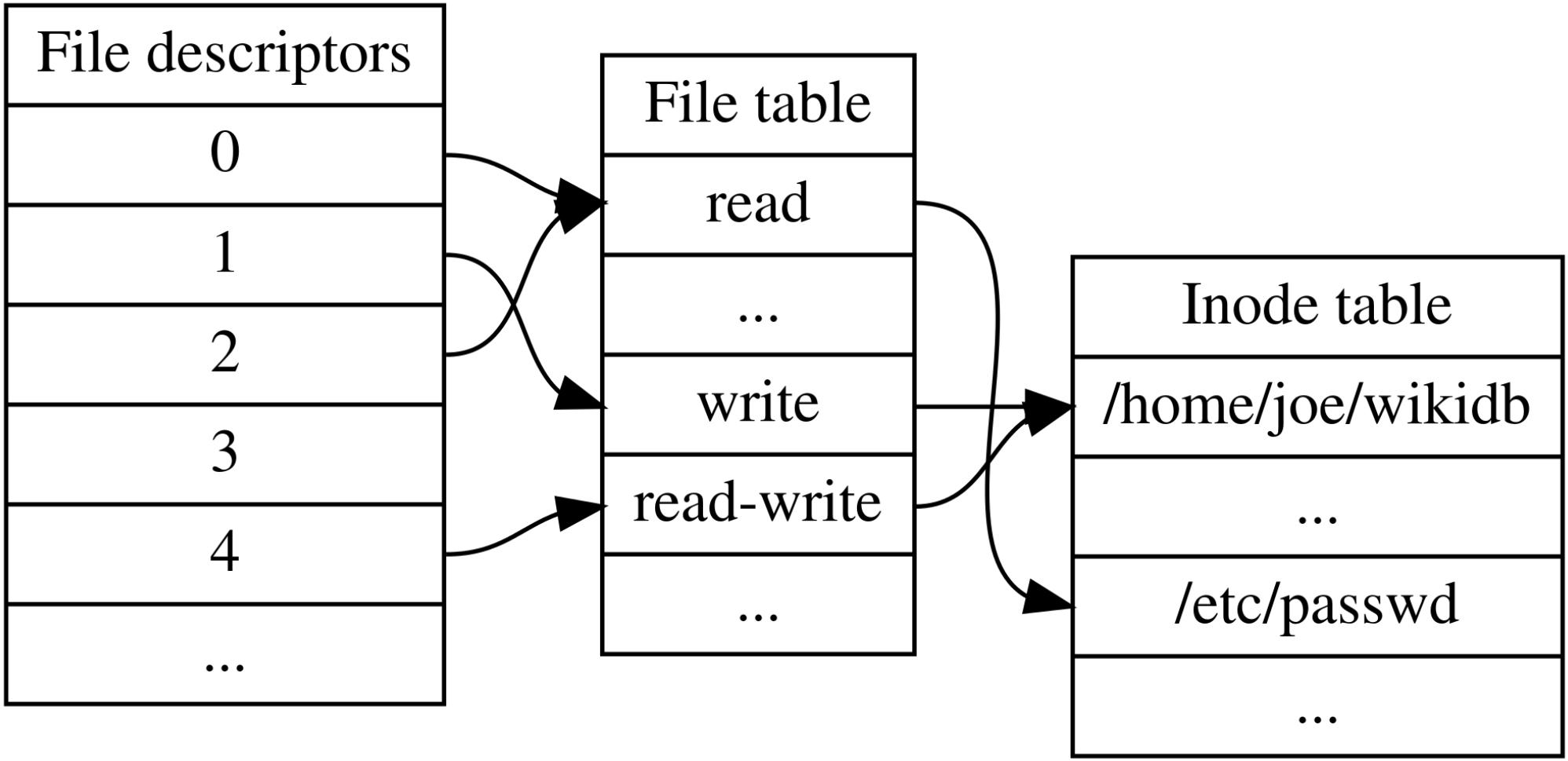
Interaction Among Objects



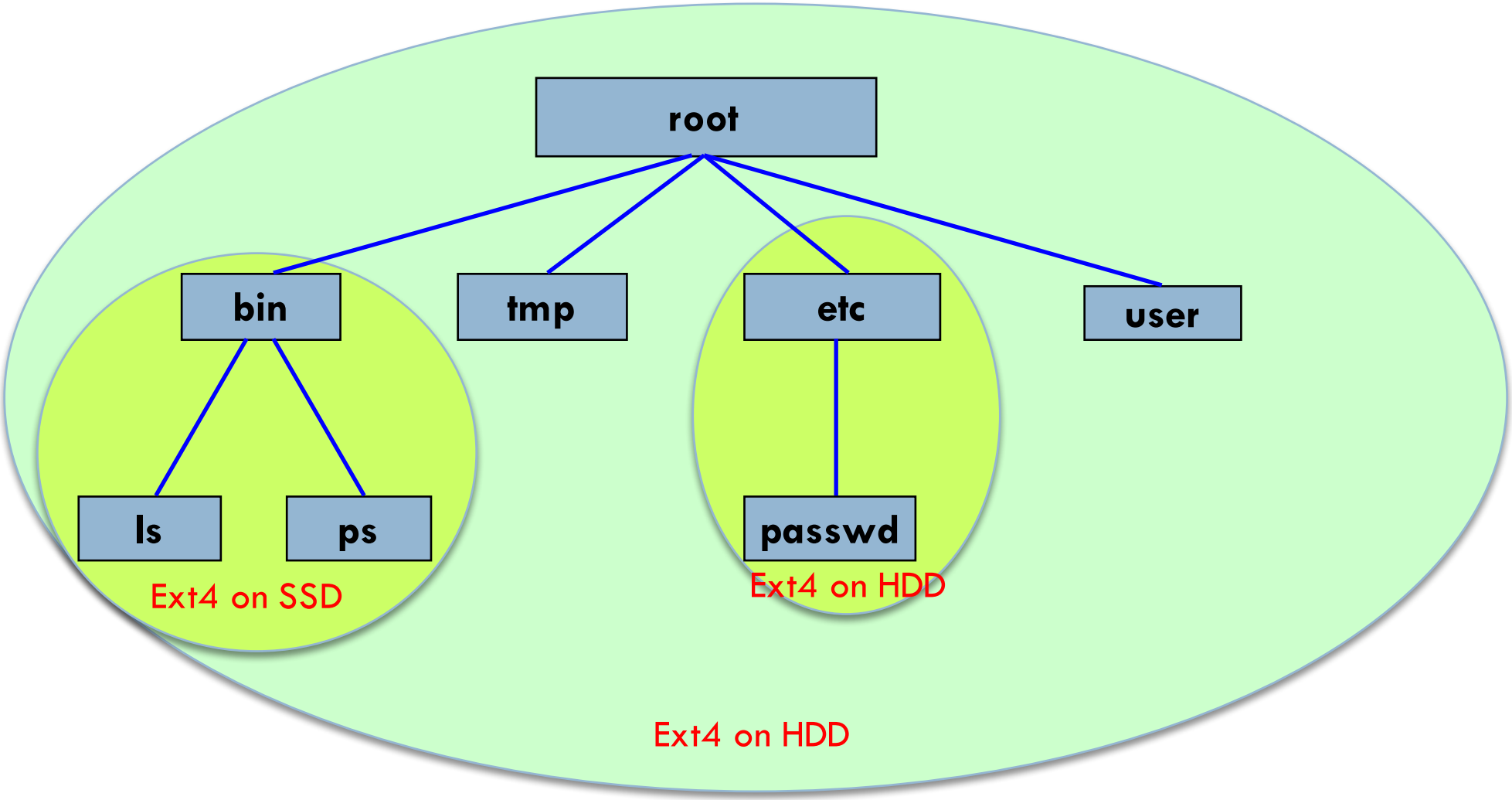
Files Associated with a Process

- Each process has its own current working directory and its own root directory, this is stored in an *fs_struct* in the *fs* field of the process descriptor
- The open files of a process are stored in a *files_struct* in the files field of the process descriptor
- When performing an *open()* system call, the file descriptor is actually an index into an array of the file objects in the *fd* array field of the process descriptors files field
  - For example, *current->files->fd[1]* is standard output for the process

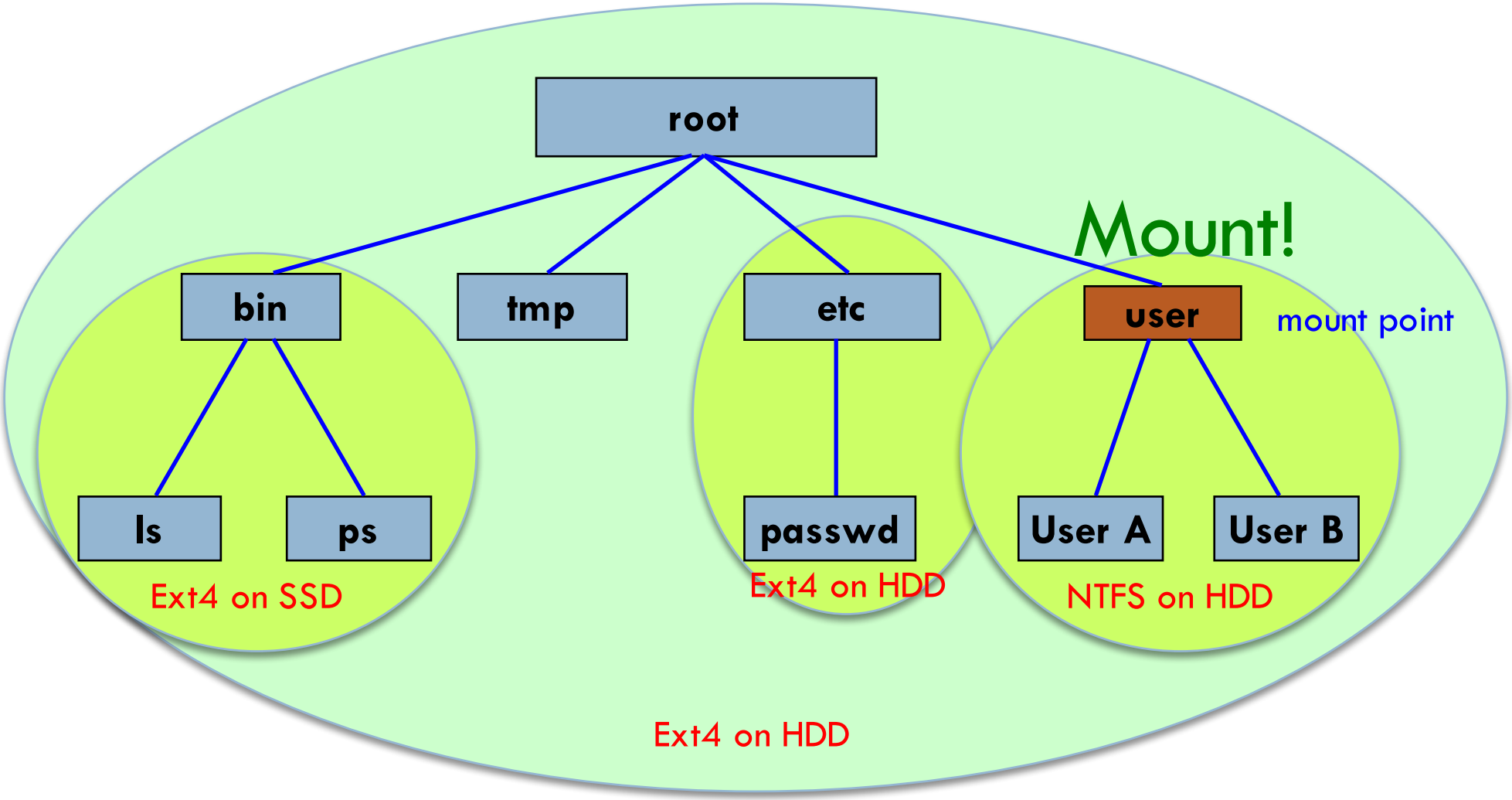
Files Associated with a Process



Mount and Unmount



Mount and Unmount



Mount

- A partition is a basic unit of mount
- A mount point is a normal directory
- Mount maps a mount point to the root of the newly attached filesystem
- Previous contents of the mount point become inaccessible as long as another filesystem is mounted there
- `mount -t ext4 /dev/sda3 /mnt`

Unmount

- Unmount detaches a mounted filesystem from its mount point
- Contents in mount point become accessible again after unmount
- umount /mnt
- Unmount is allowable only when no processes have open files in mounted filesystem
  - umount -l conducts lazy release

File Locking

- You can lock a portion of a file or an entire file for exclusive access by using lockf(3)
  - This is an advisory lock
  - An advisory lock requires all programs to use lockf() to enforce locking
- Prototype

```
#include <unistd.h>
int lockf(int fd, int cmd, off_t len);
```

- Locked section ranges from current offset to len
- If len is 0 then rest of file will be locked

Using fcntl() to Lock Files

- fcntl(2) can do many things
  - Duplicating a FD
  - Changing FD flags
  - Advisory locking
  - Mandatory locking (Non-POSIX)
- Prototype

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

File Locking Implementation

- File locking is associated with a file object
  - Owned by the process that opened the file
- Cloned processes will share the same lock
- Checking advisory locks VS mandatory locks
  - Mandatory locks work only when the file system is mounted with MS_MANDLOCK option and file has setgid without group executable permission
  - Kernel checks lock data structures when mandatory lock-enabled files