CS310 Operating Systems

Lecture 35 : File System - 3

Ravi Mittal

IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
    - Book: Modern Operating Systems by Andrew Tanenbaum and Herbert Bos,
        - Chapter 4
    - Book: Linux System Programming: talking directly to the kernel and C library, by Robert Love
    - Book: Linux – The Textbook, by Sarwar, Koretsky
    - Book: Advanced Programming in the Unix Environment, by W Richard Stevens and Stephen Rogo
    - Lecture Notes: IC221 Systems Programming,
        - https://www.usna.edu/Users/cs/wcbrown/courses/IC221/classes/L09/Class.html

Read the following:

- Book: Modern Operating Systems, by Andrew Tanenbaum and Herbert Bos
  - Chapter 4
- Book: Linux System Programming: talking directly to the kernel and C library, by Robert Love
- Book: Linux – The Textbook, by Sarwar, Koretsky
- Book: Advanced Programming in the Unix Environment, by W Richard Stevens and Stephen Rago

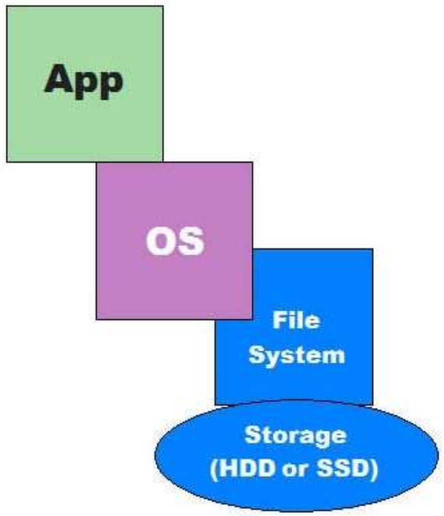
Today’s Class

- File System – Introduction (previous class)
- File System –Data Structures
- File System - Name Space
- Virtual File System
- Basic File System APIs

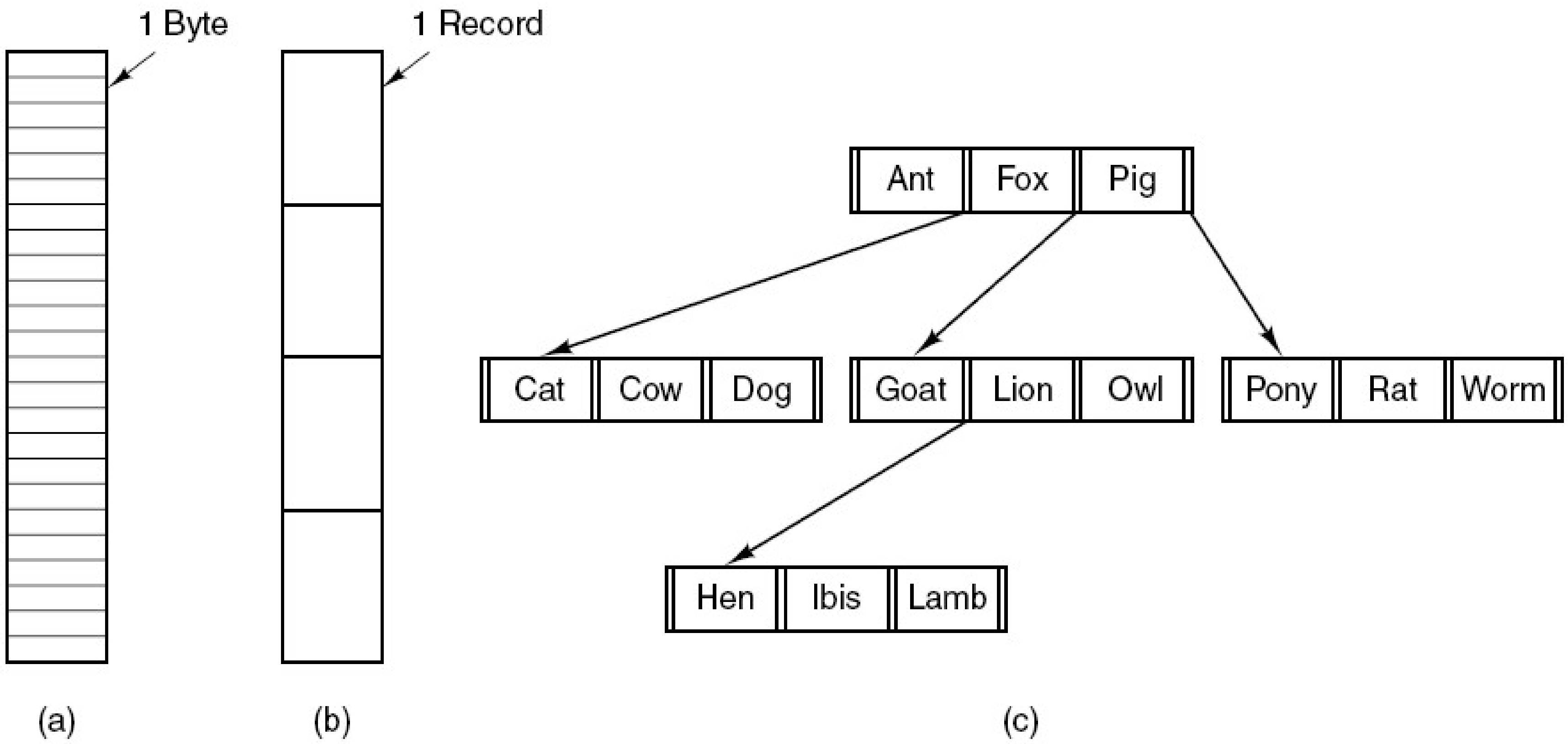
Previous Classes

File System

- Files are logical unit of information – Created by Processes
- A Disk may contain thousands of files – usually independent of each other
- Processes can read existing files and create new files
- Files remain in existence not affected by process creation and termination
- File must disappear only when the owner explicitly deletes it
- File are managed by Operating system
- File System deals with how files are
  - Structured
  - Named
  - Accessed
  - Used
  - Protected
  - Implemented, and
  - Managed



File Structure – Three Types



Byte sequence

Record sequence

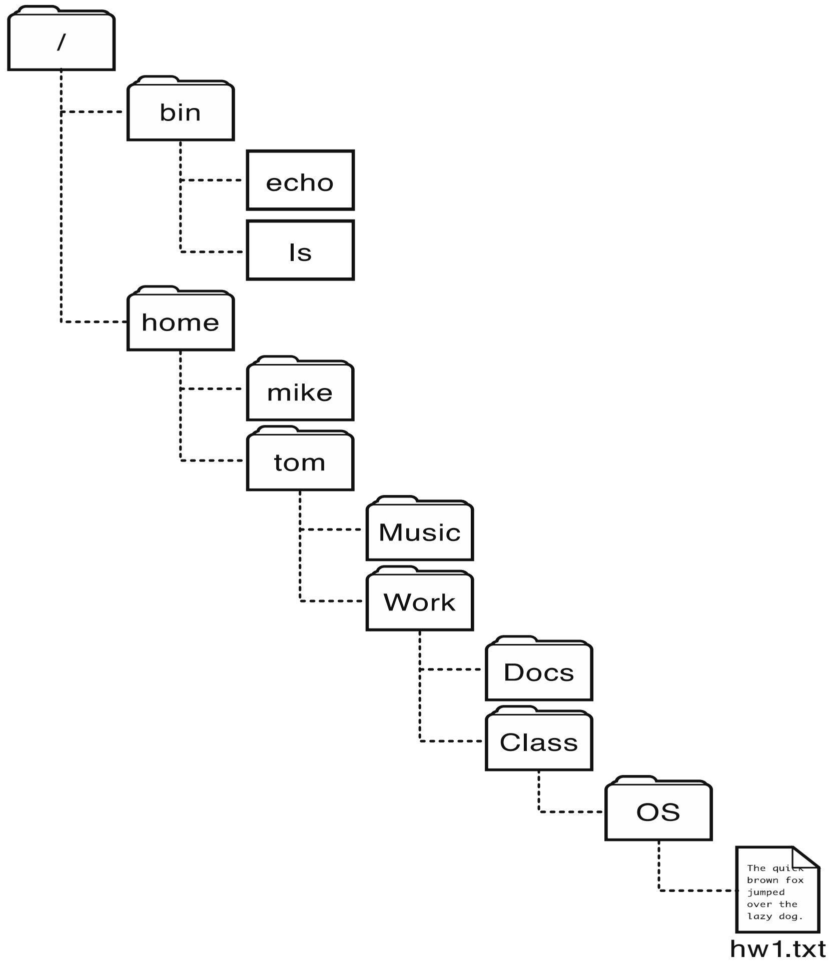
Tree

Regular Files (Linux)

- File reading or writing is done byte by byte
  - The file position increases
- File position may be set manually

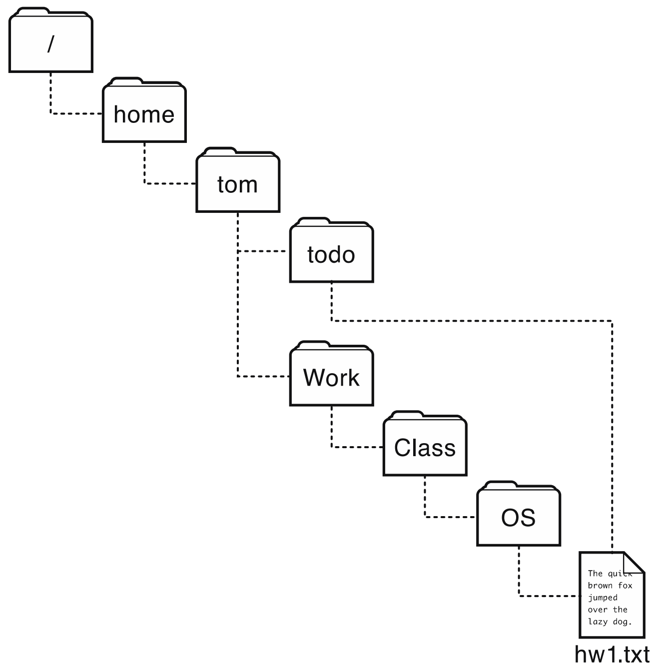
- File position starts at zero
- Writing a byte to the middle of a file overwrites over the previous byte
- It is not possible to expand a file by writing into middle of it

Hierarchical organization of files using directories



Directory is (not always) a tree

- Sometimes it is convenient for a file to appear in two directories



File System Data Structures (Unix)

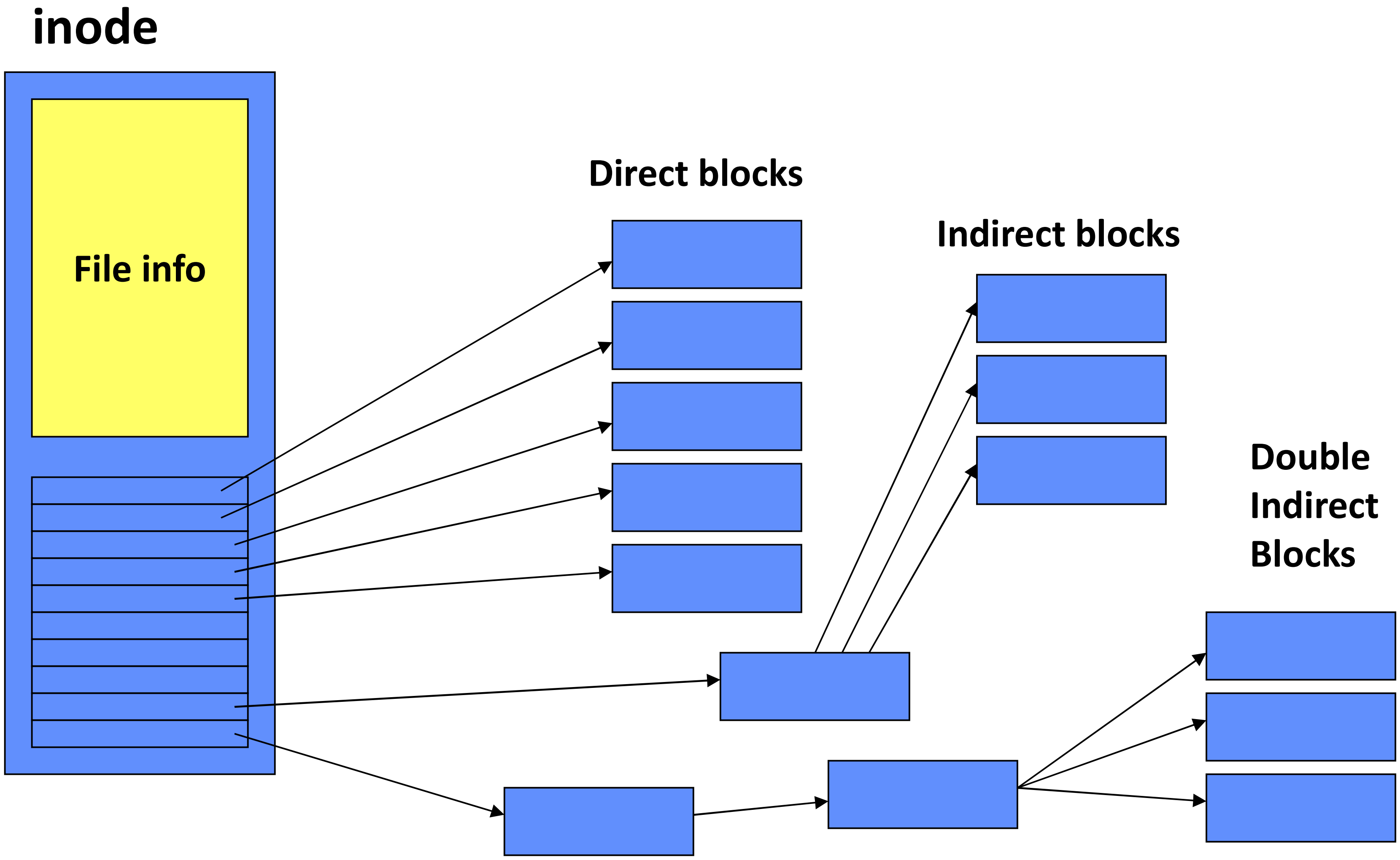
Contents of *inode*

Link Count
File Mode
User ID
Time Created
Time Last Updated
Access Permissions

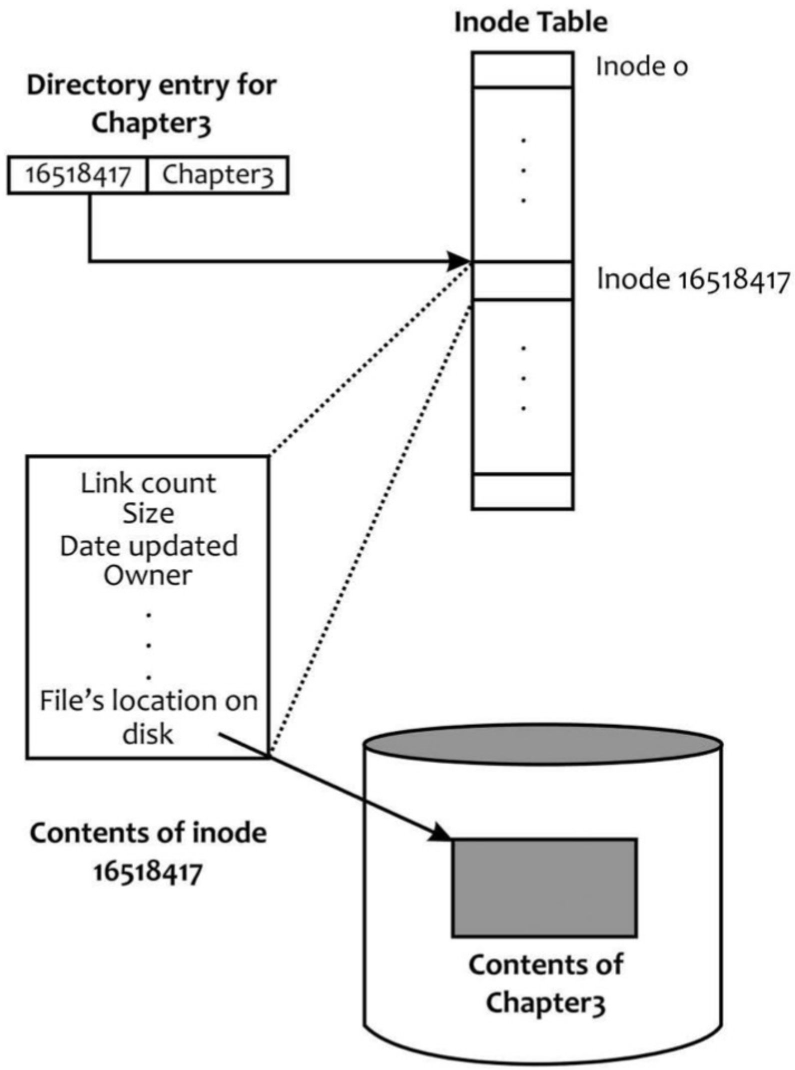
•
•
•

File's Location on Disk

Inode diagram

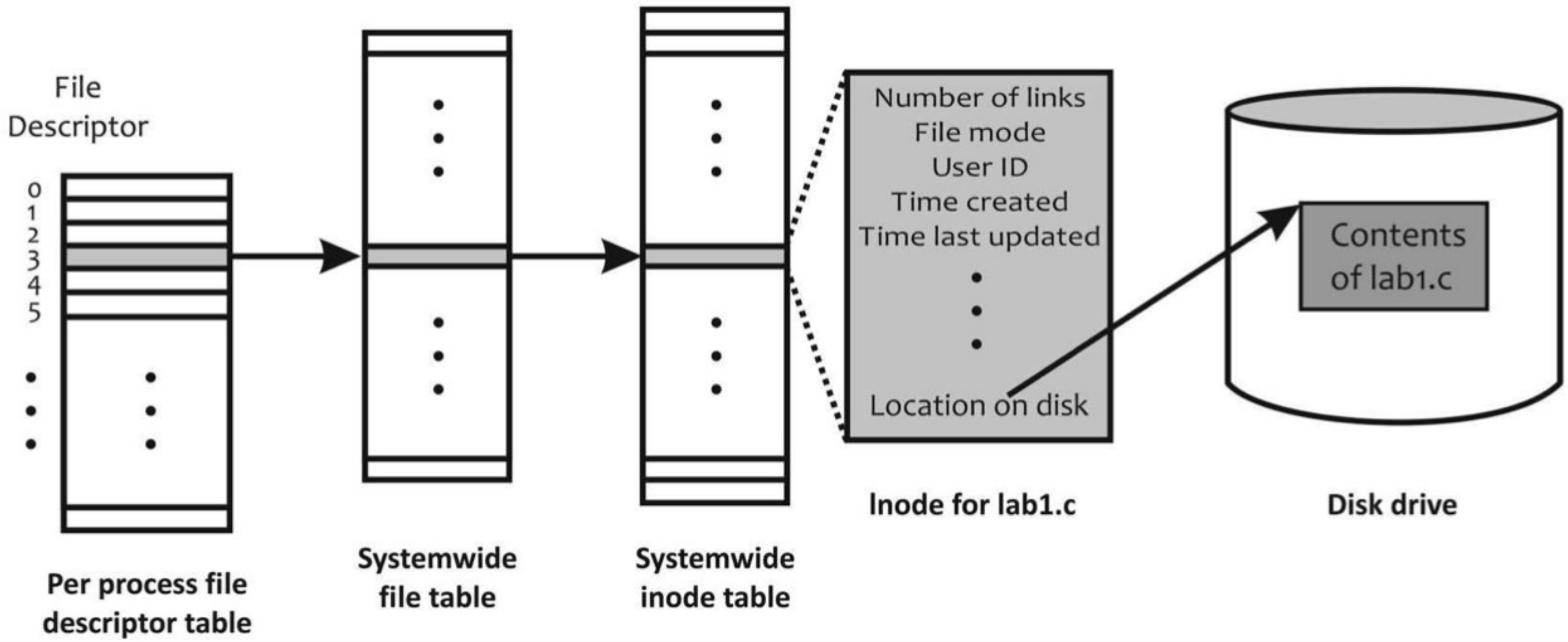


Directory Entry, inode, and file contents



Adapted from: Linux: The Textbook by Sarwar and koretsky

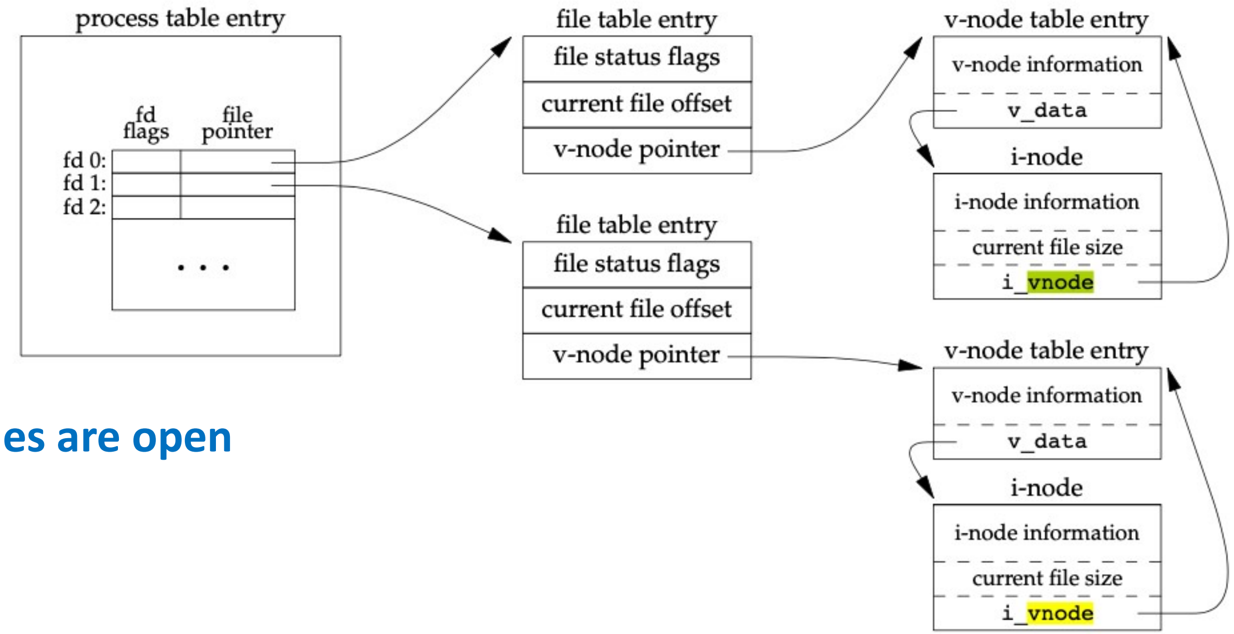
File descriptor, file table, inode, disk



vnode table (Unix)

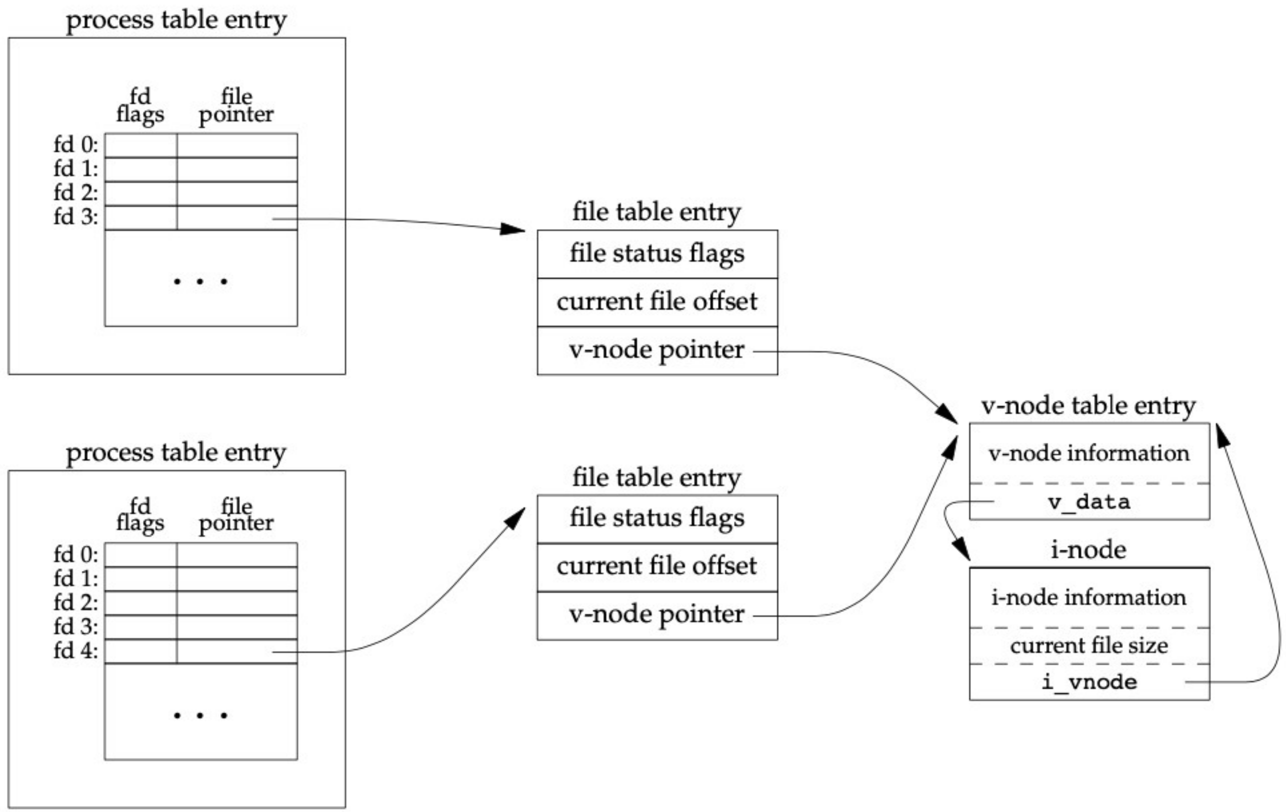
Kernel Data Structures for open files (Unix)

For a single process

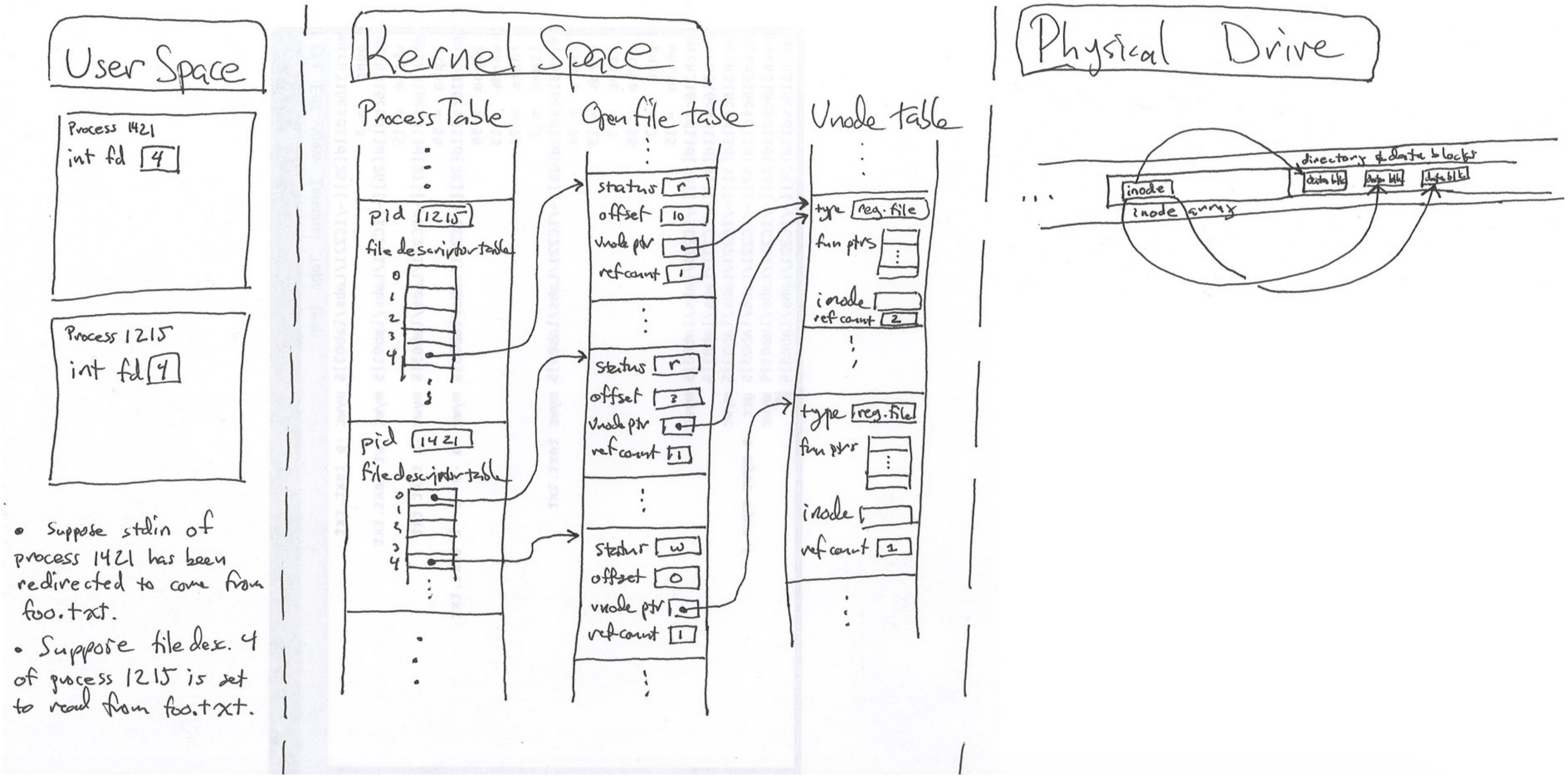


Two files are open

Two processes using the same opened file



Big Picture



User Space
Process 1421
int fd 4
Process 1215
int fd 4
• Suppose stdin of process 1421 has been redirected to come from foo.txt.
• Suppose file desc. 4 of process 1215 is set to read from foo.txt.
Kernel Space
Process Table
pid 1215
file descriptor table
pid 1421
file descriptor table
Open file table
status r
offset 10
vnode ptr
ref count 1
status r
offset 3
vnode ptr
ref count 1
status w
offset 0
vnode ptr
ref count 1
Vnode table
type reg. file
fun ptrs
inode
ref count 2
type reg. file
fun ptrs
inode
ref count 1
Physical Drive
directory & data blocks
inode
inode array

File descriptor

- Kernel and the user process refer to each open connection by a number (type int) called a *file descriptor*
- The kernel uses *file descriptors* to perform file operations
  - Example: file read
- The kernel uses a *file descriptor* to index the per-process file descriptor table
  - Obtains a pointer to the systemwide file table
- File descriptor Table (per process):
  - It resides in Process Table entry (PCB)
  - PCB contains all *file descriptors* currently in use by a process
  - Each *file descriptor* points to a file table entry

System Open File Table

- The system needs to keep track of each connection to a file
- Since the same file may have many open connections, this connection is truly distinct from the concept of a file
- Distinct processes may even share a connection
  - eg. two processes wanted to write to the same log file, each adding new data to the end of the file)
  - The connection information cannot be kept inside the process table
- Kernel keeps a data structure called the *system open-file table* which has an entry for each connection
- Entry contains
  - Connection Status: read or write
  - Current offset in the file
  - A pointer to vnode entry for the file
    - Vnode is data structure representing the file
    - Linux: vnode → generic inode

Vnode table

- Each open file has a vnode structure
- Vnode contains
  - Information about type of the file
  - Pointers to functions that operate of file
  - Copy of the *inode* for the file
    - Physical information about the file
  - This information (about *inode*) is read from the disk when the file is opened
  - Reference Count: gives the number of hard links
    - A file can't be removed from the file system unless all of those references are removed

Physical Device: inodes etc

- The *inode* contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk
- A file may be broken up into many data blocks, which may be widely distributed across the physical drive
- The *inode* for a file contains the locations of each of the data blocks comprising the file

Special Files

File Types

- Regular Files (most common)
  - Text
  - Binary
- Directories
  - A file that contains the names and locations of other files
- Character Special and Block Special Files
  - Terminals (character special) and disks (block special)
- FIFO (named pipe)
  - A file type used for inter-process communication
- Socket
  - A file type used for network communication between processes

Special Files

- *Special files* are kernel objects that are represented as files
- Linux supports four types of files
  - block device files,
  - character device files,
  - named pipes, and
  - Unix domain sockets
- Special files are a way to let certain abstractions fit into the filesystem
  - Everything-is-a-file paradigm
- Linux provides a system call to create a special file

Device Files

- I/O devices are treated as special files called *device files*
- Hence, device access (in Unix/Linux systems) is performed via *device files*
  - Act and look like normal files residing on the file system
- The same system calls used to interact with regular files on disk can be used to directly interact with I/O devices
- Device files may be opened, read from, and written to
- This allows users to access and manipulate devices on the system
- Each type of device has its own special device file
- Two types
  - Character Devices
  - Block Devices

Character Device - Files

- Character devices are accessed via a special file called a *character device file (node)*
- A character device is accessed as a linear queue of bytes
- The device driver places bytes onto the queue, one by one
- User space reads the bytes in the order that they were placed on the queue
- Character devices are accessed via *character device files*
- Character devices provide access to data only as a stream, generally of characters (bytes)
- Example: Keyboard, mice, printers etc

Block Device - Files

- A block device, in contrast, is accessed as an array of bytes
- The device driver maps the bytes over a seekable device
- User space is free to access any valid bytes in the array, in any order
  - Read byte 12, byte 7, byte 15 etc
- Block devices are generally storage devices
  - Hard disks, CD-ROM drives, Flash Memory
- They are accessed via *block device files*
- Generally mounted as a filesystem

File System Name Space

Volume

- A volume is a collection of physical storage resources that form a logical storage device
  - Each instance of a file system manages files and directories for a volume
- A volume is an abstraction that corresponds to a logical disk

File System and namespaces

- Some operating systems separate different disks and drives into separate namespaces
  - a file on a floppy disk might be accessible via the pathname *A:\plank.jpg*
  - the hard drive is located at *C:*
  - DOS and Windows, which break the file namespace up into drive letters
  - This breaks the namespace up among device and partition boundaries

- Linux (Unix) provides a *global* and unified *namespace* of files and directories
  - In Unix, that same file on a floppy might be accessible via the pathname */media/floppy/plank.jpg*
    - Alongside files from other media

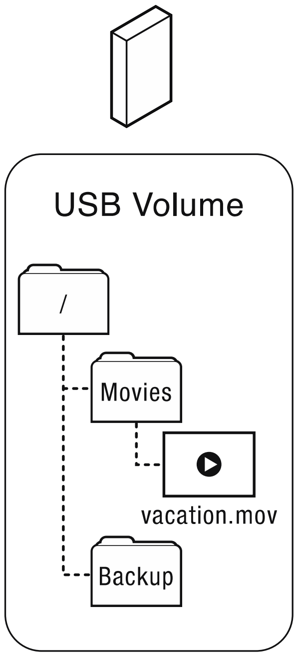
Mounting and Unmounting

- A *filesystem* is a collection of files and directories in a formal and valid hierarchy
- Filesystems may be individually added to and removed from the global namespace of files and directories
  - These operations are called *mounting* and *unmounting*
- Each filesystem is mounted to a specific location in the namespace, known as a *mount point*
- The root directory of the (added) filesystem is then accessible at this mount point
- For example, a CD might be mounted at */media/cdrom*, making the root of the filesystem on the CD accessible at */media/cdrom*

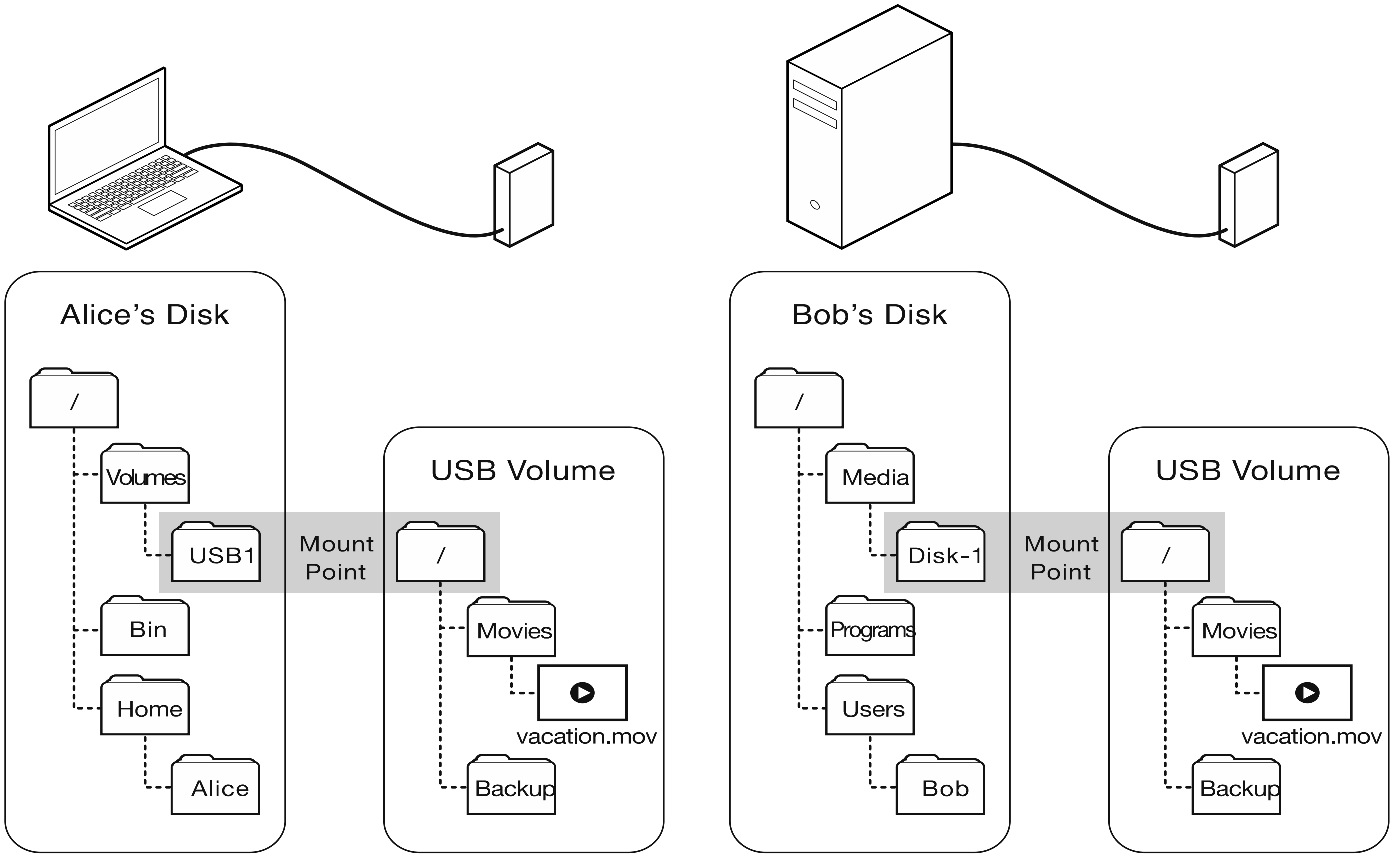
Mount (repeat)

- Mount allows multiple file systems on multiple volumes to form a single logical hierarchy
- Mounting a volume on an existing file system creates a mapping from some path in the existing file system to the root directory of the mounted volume's file system
- Mounting enables all mounted filesystems to appear as entries in a single tree

USB Volume



Mapping from some path in existing file system to the root directory of the mounted file system



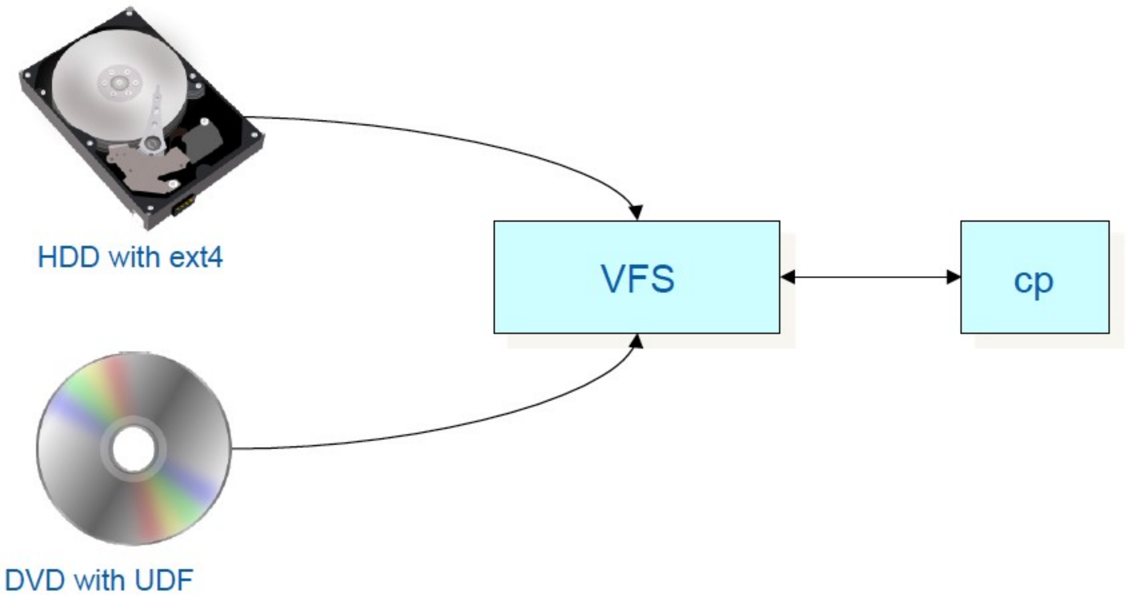
Alice can access the vacation.mov movie using the path /Volumes/usb1/Movies/vacation.mov

Bob can access the movie using the path /media/disk-1/Movies/vacation.mov

Virtual File System (Linux/Unix)

Virtual File System (Unix/Linux)

- The Virtual File System (VFS) implements a generic file system interface between the actual file system implementation (in kernel) and accessing applications
- Applications can access different file systems on different media via a homogeneous set of UNIX system calls



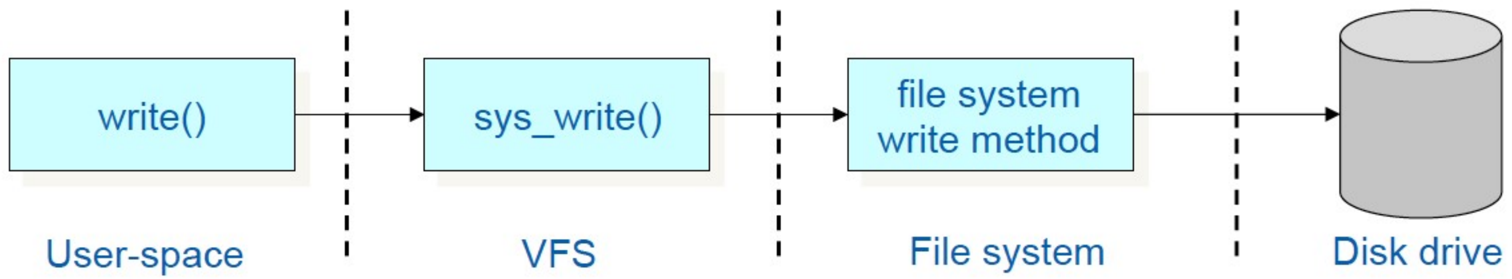
Adapted from: Linux System Programming : Robert Love

Virtual File System

- VFS is a glue that enables syscalls such as open(), read(), and write()to work regardless of the filesystem or underlying physical medium
- System calls work *between* these different filesystems and media
  - In older operating systems, such as DOS, this would never have worked
- Generic Interface
  - Kernel implements abstraction layer around low-level interfaces
  - Linux to support different filesystems, even if they differ in supported features or behavior
    - FAT, Windows NTFS, Linux and Unix file systems

Virtual File System

- To the user, each file system looks the same
- Example: `write (fd, &buf, len);`
  - Write `len` bytes in file with descriptor `f` from buffer buf
  - It is translated into a system call `sys_write()` which determines the actual file writing method for the filesystem that holds file corresponding to `fd`
  - The generic write system call is then invokes the method that is part of the filesystem implementation



Adapted from: Linux System Programming : Robert Love

Basic File System APIs

File System API – file creation

- Creating files
  - `create()` creates
    - a new file with some metadata, and
    - a name for that file in a directory
  - `link()` creates a hard link
    - a new name for the same underlying file
  - `unlink()` removes a name for a file from its directory
    - If a file has multiple names or links, unlink() only removes the specified name, leaving the file accessible via other names
    - If the specified name is the last (or only) link to a file, then unlink() also deletes the underlying file and frees its resources
  - `mkdir()` and `rmdir()` create and delete directories

File System API – Open and Close

- `Open()`
  - To get a file descriptor it can use to refer to the opened file
  - Path parsing and permission checking can be done just when a file is opened
    - Not repeated on each read or write
  - The OS creates a data structure that stores information about the file: file id, read/write/exec, pointer to the processes current position in the file
  - The file descriptor can thus be thought of as a reference to the operating system's per-open-file data structure that the operating system will use for managing the process's access to the file
- `Close()`
  - Releases the open file record in the operating system

File System API – File Access

- While a file is open, an application can access the file's data in two ways
  - `read()`
  - `Write()`
    - Call to read and write starts from files current position
- `Seek()`
  - Supports random access within a file
  - `Seek()` call changes a process's current position for a specified open file
- Other APIs
  - `munmap()`
  - `fsync()`

Lecture Summary

- In file system (Unix / Linux) the following data structures are used
  - File descriptor
  - File Table entry (Process Control Block)
  - Open File Table
  - Vnode or generic inode table
  - Inode
- Devices are accessed with special files: Character special and block special files
- File Name Space
- Mounting and Unmounting
- Virtual File System (Linux / Unix)
- File APIs