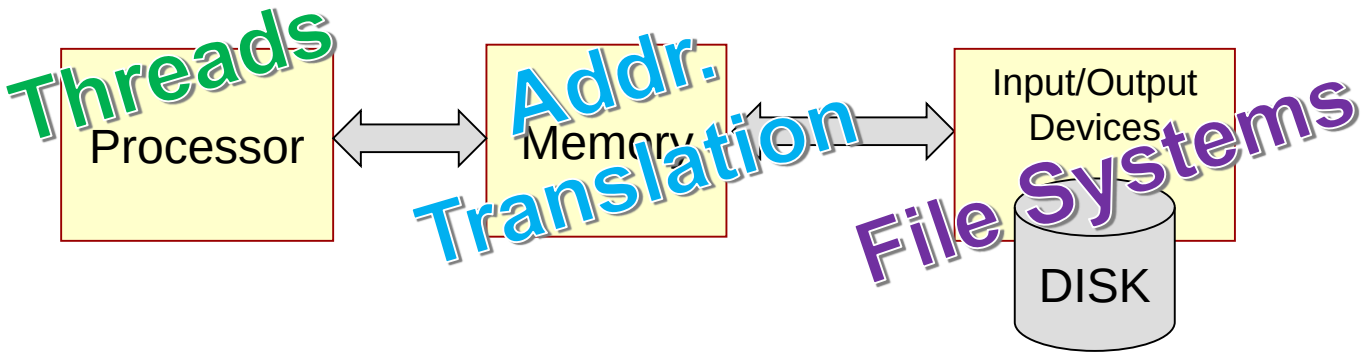
CSCI 350
Ch. 11 – File Systems

Mark Redekopp
Michael Shindler & Ramesh Govindan

Abstracting Persistent Storage

- Thread = Abstraction of the processor
- Address translation => Abstraction of memory
- What about abstracting storage I/O?
  - **File Systems**
- **File systems provide persistent, named data capabilities**
  - **Persistent: Contents retained until explicitly deleted even when power is off**
  - **Named: Use of human-friendly (human-chosen) named files & directories**
    - **Example: /home/student/cs350/pintos/src/threads/thread.c**



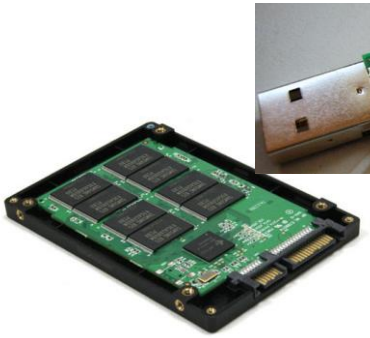
File System Requirements

- Reliability
- High-capacity
- Fast access
- Named data
- Controlled sharing (security)

Hardware Components

- Non-volatile storage
  - Non-volatile means contents are retained even when power is not supplied
  - By contrast, DRAM (main memory and possibly lower cache levels) and SRAM (generally used in cache) lose their contents when power is off
- Types: Tape, magnetic disk, and flash (solid state drives)





https://www.backblaze.com/blog/hdd-versus-ssd-whats-the-diff/
http://dis-dpcs.wikispaces.com/6.2.1+Blocking%2C+Sectors%2C+Cylinders%2C+Heads

Requirements Met by HW

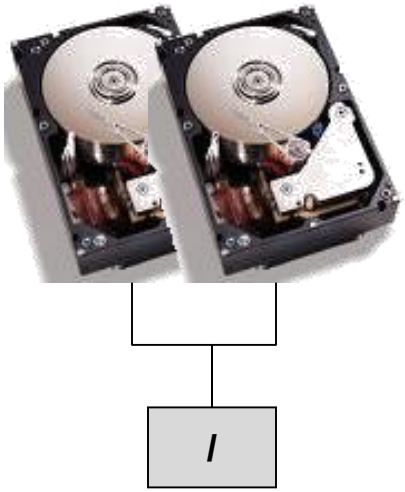
Requirement	HW Ability	HW Disability
Reliability	Generally long lifespan	• Disk (mechanical devices) drives fail (e.g. head crash) • Flash memory has a fixed number of writes/erasures before it will stop functioning
High Capacity	✓	
Fast Access	Some drives provide on-board cache	Generally slow • Tape access time (sec) • Magnetic disks access time (ms) • Flash memory access time (us)
Named Data	None	• Magnetic disks use "head/sector/track" addressing • Flash use sector/block addressing
Controlled Sharing	Generally none	

Requirements Enabled by the OS

Requirement	OS File System Design Approaches
Reliability	• Since a crash can occur at anytime, use "transactions" to make updates appear atomic • Use redundancy to detect and correct failures • Move data to even the "wear" on disks and Flash drives
Fast Access	• Organize data so that access can be as "sequential" as possible • Provide memory caching • (Note: file systems generally optimize for sequential read and append write. Writing to the middle of a file may require rewriting all of its contents. Reading from random locations may be extremely time consuming.)
Named Data	• Provide abstraction of named files and directories
Controlled Sharing	• Include access-control metadata with files (R,W,X permissions, user, group, all permissions), etc.

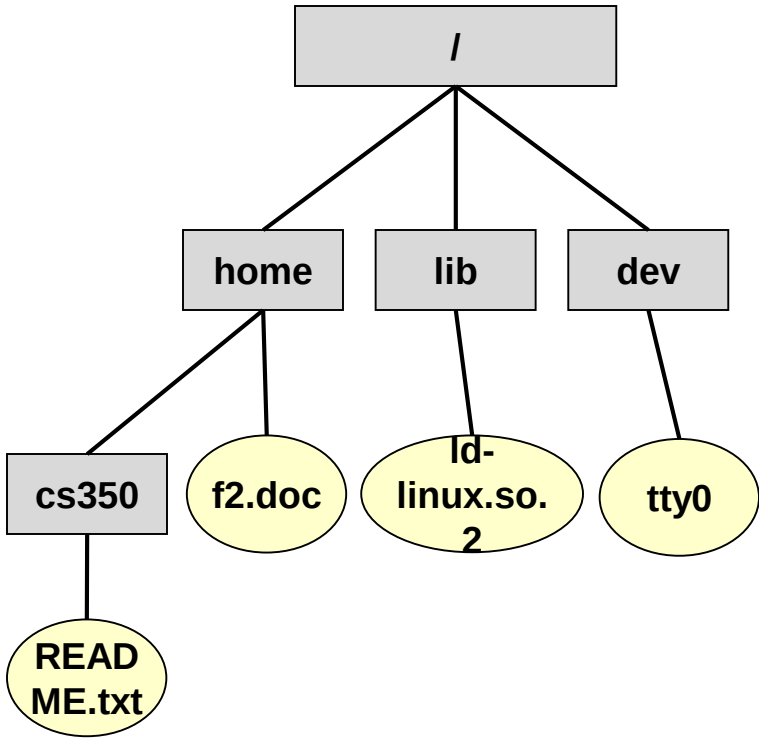
Volumes

- **Volume**: Logical storage system along with an instance of a file system
- Allows for arbitrary logical organization regardless of physical storage organization
  - 1 disks may contain multiple volumes (file systems) or partitions (e.g. C:\ and D:\)
  - 1 filesystem/volume may encompass several disks (e.g. servers)



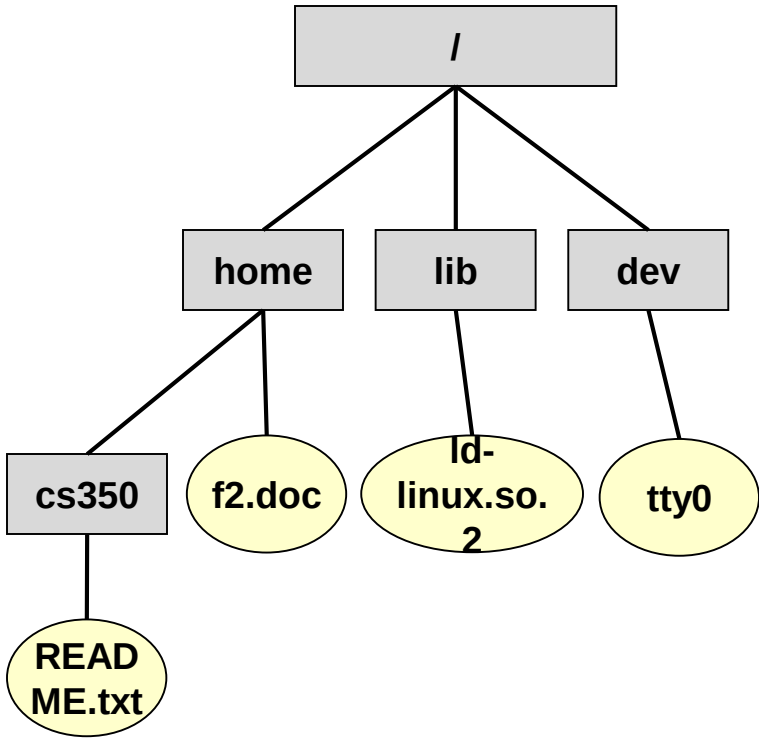
File Access and Naming

- Users generally access the file systems by
  - **Browsing**: Know the name of the file and want to navigate to it
  - **Searching**: Not sure of the name
    - Could search by name or content
    - Requires some kind of indexing for fast access
- To enable easy browsing file systems usually employ a hierarchical naming system (**tree** of directories [internal nodes] and files [leaves])



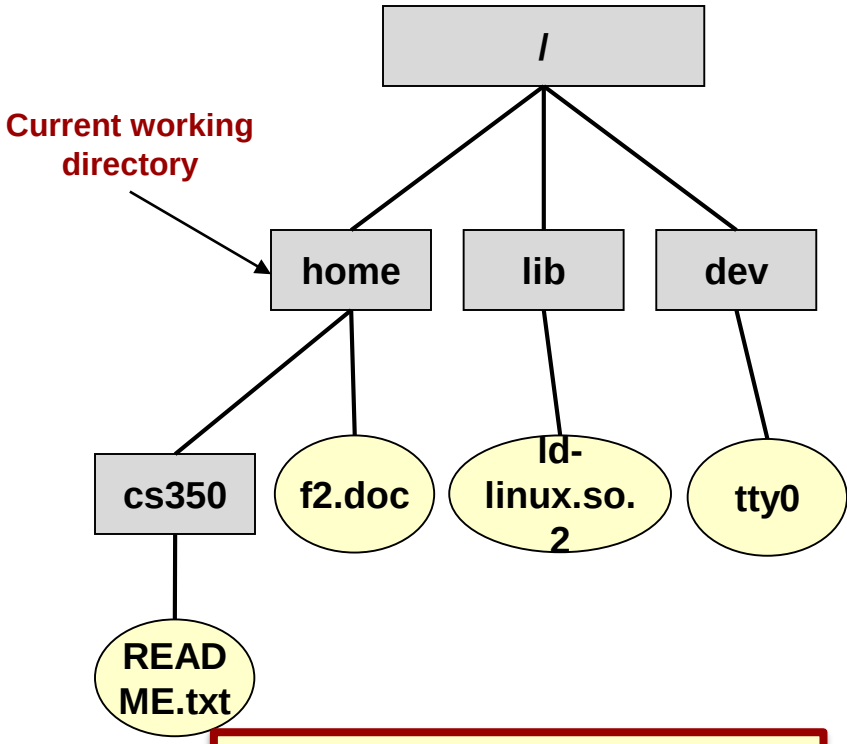
Special Directories

- **Root directory**: Starting point of the file sysem
  - Linux/Unix/Mac: /
  - Windows: C:\
- **Current working directory**: References/Paths to files or other directories will be interpreted as starting from this current location
  - Can be changed as needed (i.e. 'cd cs350';)
- **Home directory:** Starting point of a user upon login (/home/cs350)
  - Linux/Unix/Mac shortcut: ~



Paths

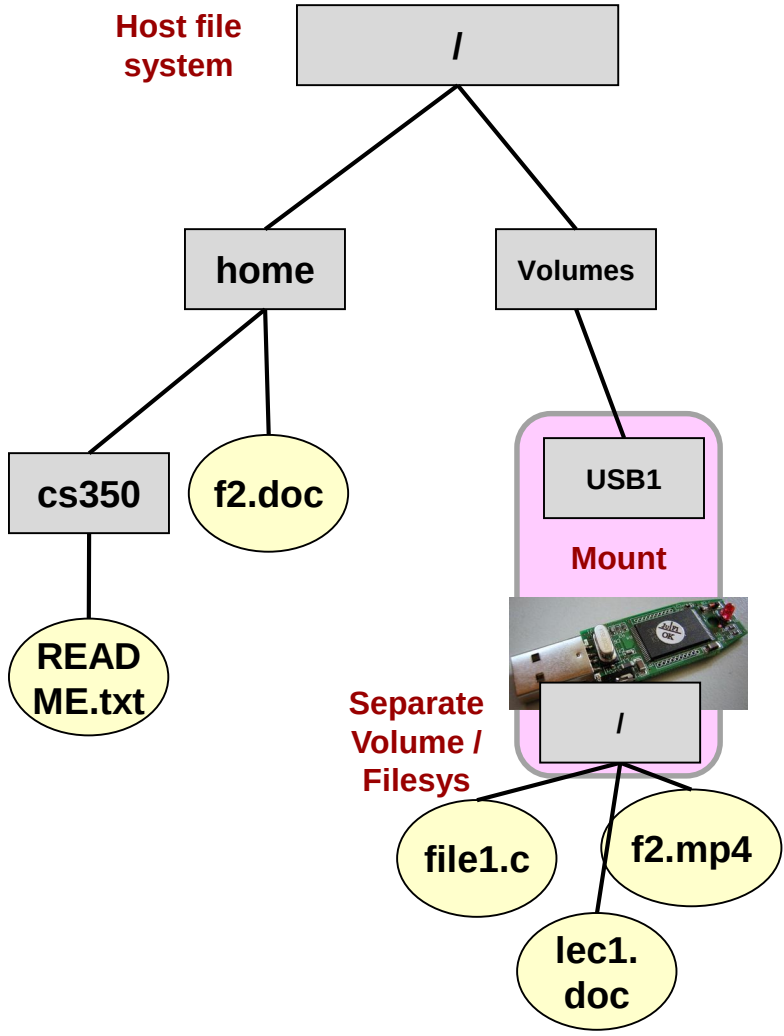
- Paths (as their name says) specify a path from one location in the file system to another
- **Absolute paths** start from the root directory
  - /home/cs350/README.txt
- **Relative paths** start from the current directory (assume '/home' is **cwd**)
  - cs350/README.txt
  - ../dev/tty0



Shortcuts:
. = Current directory
.. = Parent directory (up one)
~ = Home directory
Unix commands:
pwd = Print current working dir

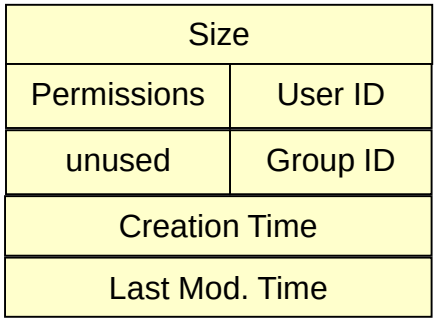
Mounting A Volume

- Multiple volumes can be made to co-exist in one logical hierarchy through a process known as mounting
- Mounting places a separate volume at a particular named location within another volume
  - CD Drives, USB Flash, etc.



File Concept

- Files consist of 2 parts
  - Metadata
  - Actual file data
- Metadata
  - Permissions, size, user ID, timestamp of creation and modification
    - And the filename too? No.
  - OSs may allow user-defined metadata (author, character encoding, etc.)
- Actual file data
  - Sequence of bytes whose interpretation (text, binary data, pixel data, etc.) is up to an application to interpret



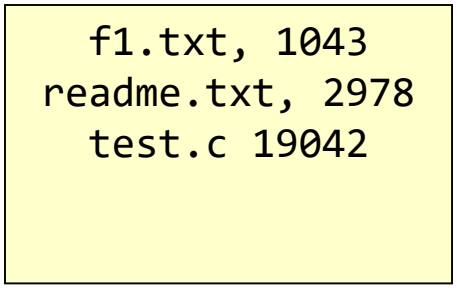
Metadata

```
00 0a 56 c4
81 e0 fa ee
39 bf 53 e1
b8 00 ff 22
```

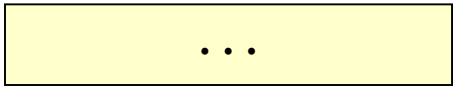
File Data

Directories (Folders)

- Usually "files" that hold lists of pairs:
  - (Human readable filename, file ID/#)
- Filenames are not stored with files because:
  - May have many names/links
  - Wouldn't be able to store just filename but full path since same filename may be used in multiple places on the volume



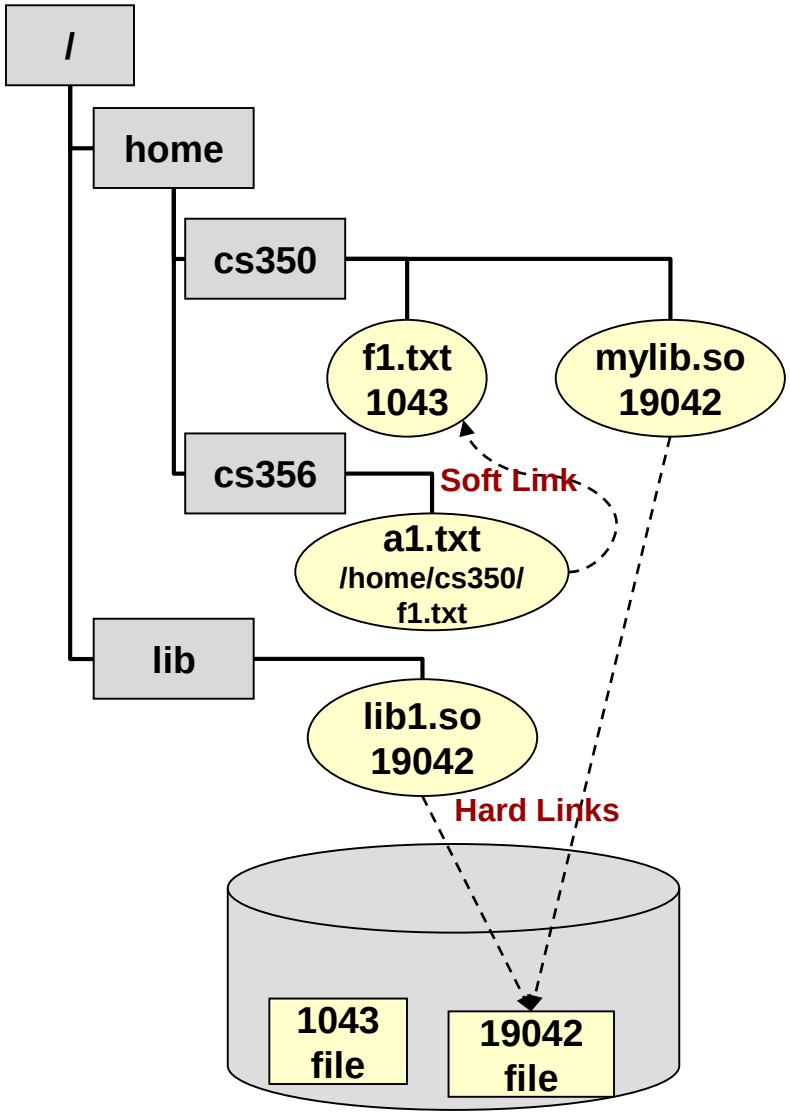
Directory (File) Data



Actual f1.txt known to the filesystem as file 1043 which can be "easily" indexed and found on the physical storage device

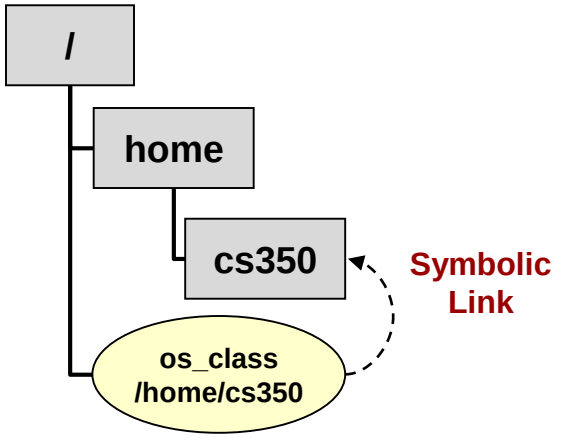
Links

- Hard link
  - A filename, file ID/# association
  - Same physical file can be known by different filenames (in different folders) but each reference the same physical file
  - Unlinking one doesn't affect the file or the other link
  - File maintains hard link count and file is only truly deleted from storage when last hard link is removed
- Symbolic (soft) link
  - One directory entry mapped to another
  - Removing actual file link (i.e. deleting file) may leave dangling soft links
  - Symbolic links can point to other directories or files on different volumes



Issues with Links

- Can have symbolic links to directories
- Interesting issue with symbolic links:
  - May no longer have a tree (one parent per node)
  - When we try to walk up the tree which "parent" do we return to
- Some shell applications track directory you came from and then return through that path
- No hard links to directories
  - Can create cycles



What is my cwd after this?

```
$ cd /os_class
$ cd ..
```

COMMON FILESYSTEM SYSCALLS

Creating & Deleting Files

Syscall	Description
create(pathname)	Creates a file
link(existingName, newName)	Creates a hard link to the underlying file referenced by existingName
unlink(pathName)	Remove the specified name for a file from its directory; if that is the last reference to a file, remove the file
mkdir(pathName)	Create a new directory with the specified name
rmdir(pathName)	Remove the directory with the specified name

- No remove/delete syscall (only unlink)

Open and Close

Syscall	Description
fd = open(fileName)	Finds and opens a file performing various checks (access permission) and initializing necessary kernel data structures to track access
close(fd)	Releases the resources associated with an open file

- **Q:** Why use a handle/file descriptor
  - You could just specify the filename when you call read/write etc.
- **A:** Avoids rechecking permissions, maintains state (current location in the file), etc.

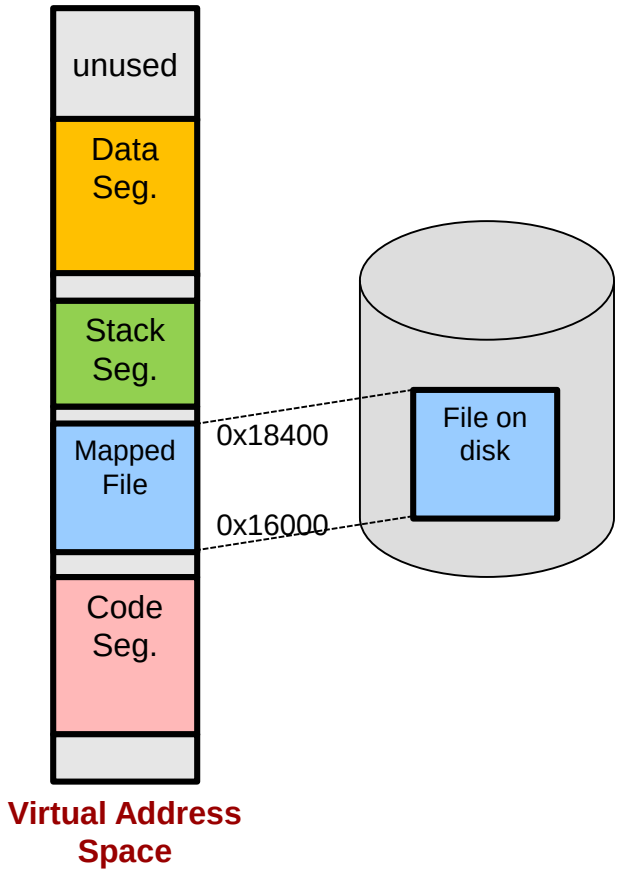
File Access

Syscall	Description
read(fd, buf, len)	Creates a file
write(fd, buf, len)	Creates a hard link to the underlying file referenced by existingName
seek(fd, offset)	Remove the specified name for a file from its directory; if that is the last reference to a file, remove the file
ptr = mmap(fd, off, len)	Set up a mapping between the data in the file (`fd`) from `off` to `off + len` and an area in the application's virtual address space from `ptr` to `ptr + len`. Writes are buffered and flushed periodically or when msync/munmap are invoked.
munmap(dataPtr, len)	Unmaps the file from the virtual address space
msync(dataPtr, len)	Flushes modified data from the given range back to the underlying file
fsync(fd)	Force modifications to a file to be flushed to disk

- No rmove/delete syscall (only unlink)

mmap Example

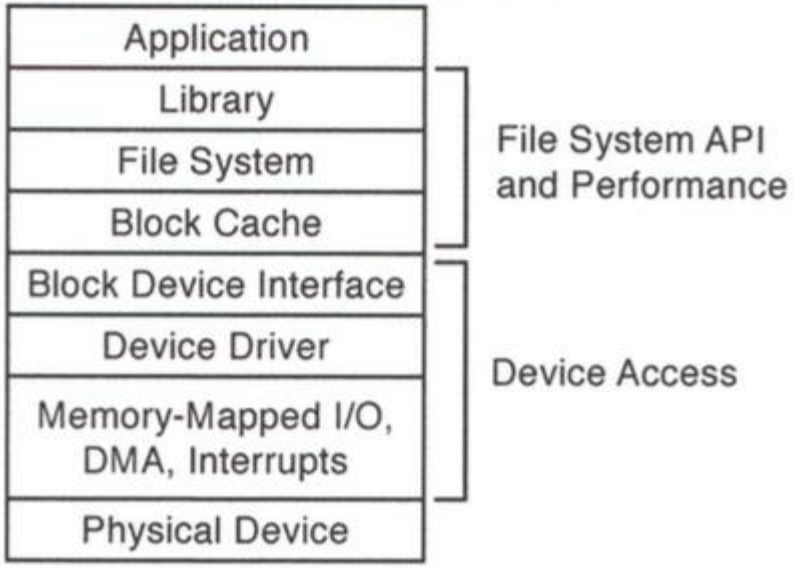
- Memory-mapped file I/O
- Provide efficient access when data from a file will be accessed multiple times
  - Memory access is far faster than disk access
  - Like an explicit caching of a file's data



APIS AND DEVICE ACCESS

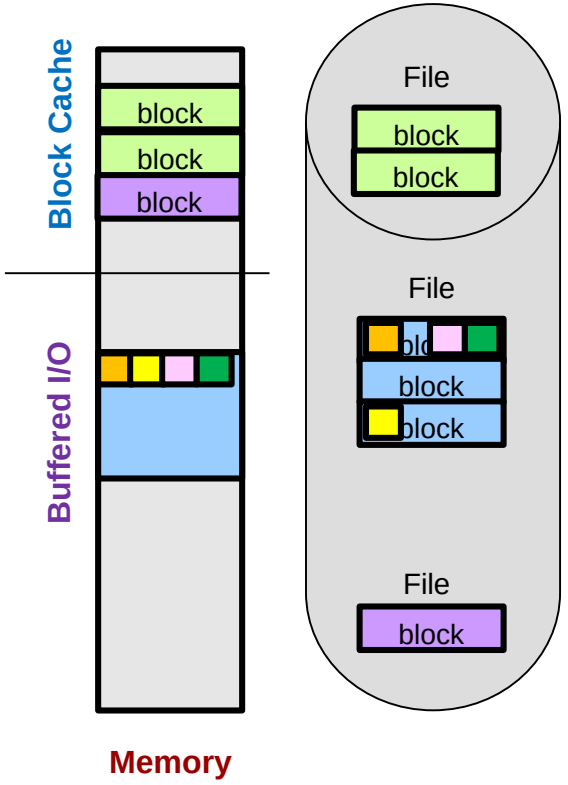
Software Layers

- File access consists of many layers of software
- API layers provide a simplifie interface to the developer
- Performance layers utilize caching methods
- Device access interfaces to the HW and utilizes HW-based performance enhancements



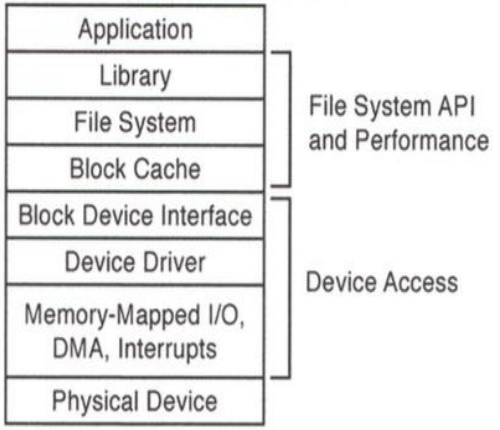
Performance Enhancements

- Buffered I/O
  - User level C library functions fwrite buffer writes in memory and writeback to disk periodically
    - Imagine multiple updates to the same data (only the last update need be written)
  - fread may bring in a whole block of data rather than the few bytes actually requested
- Caching
  - OS may maintain its own **block cache** of recently accessed disk blocks so that requests to the disk can be satisfied from the memory cache if possible
- Prefetching
  - When we request a block from the disk the OS may issue a request for the next block so that if it is needed it will be ready (soon)
  - Take care: can lead to issues of cache pressure, I/O contention, and wasted effort



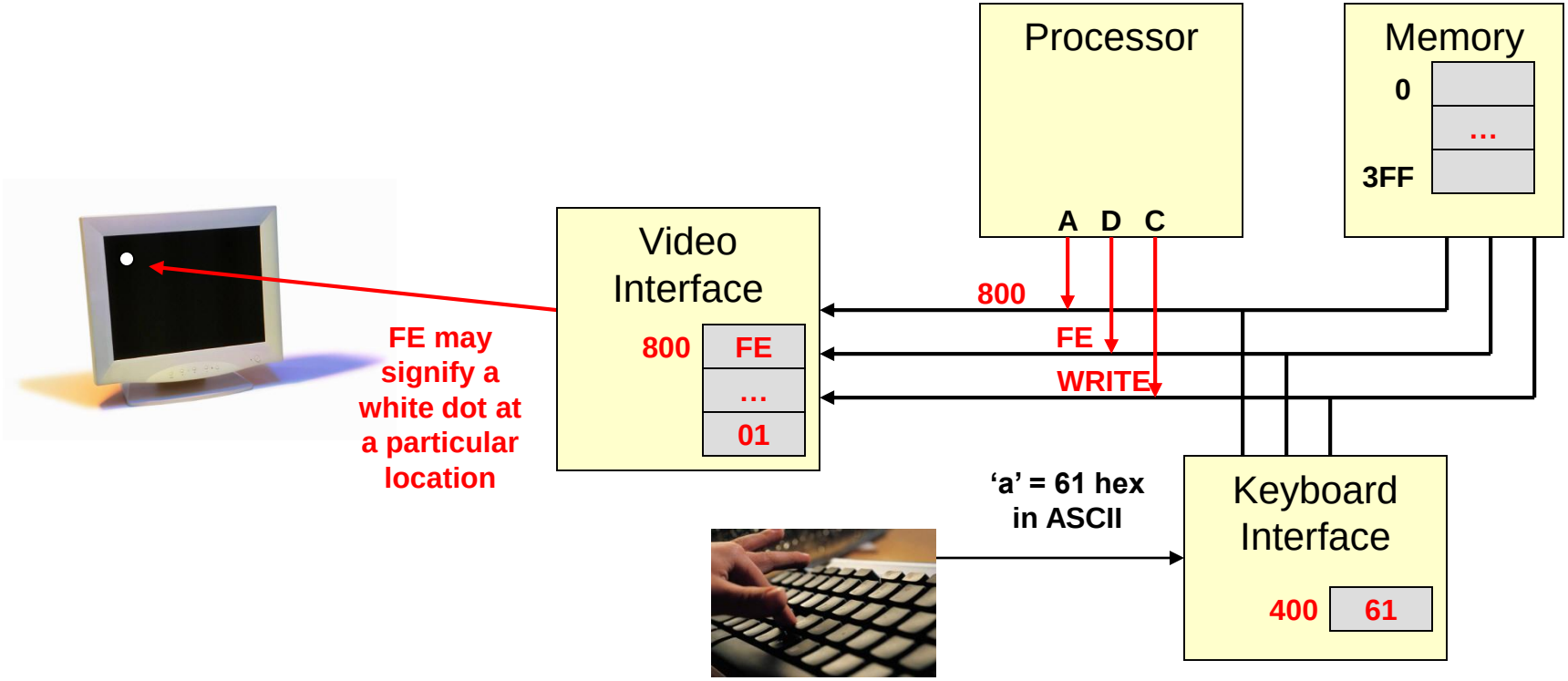
Device Driver Organization

- OS device drivers must meet a certain interface for certain classes of devices
- Byte-oriented (character devices)
  - Read and write data in units of bytes/characters
  - Data may be ephemeral
    - Writing to the console, reading from a serial port c keyboard
- Block-oriented
  - Read and write data in blocks (chunks) (e.g. 512 byte sectors at a time)
  - Used for devices that can host a file system
  - Well-known interface that all devices must implement (e.g. bread() and bwrite())
- Network interfaces



Memory Mapped I/O

- Processor performs reads and writes to communicate with I/O devices just as it does with memory
  - I/O devices have locations (i.e. **registers**) that contain data that the processor can access
  - These registers are assigned unique addresses just like memory

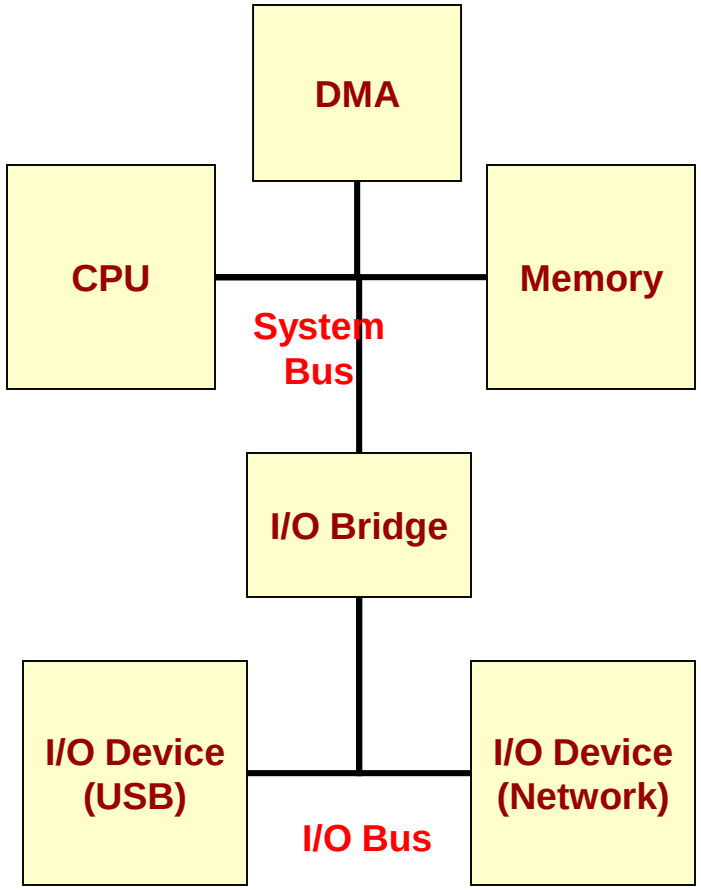


Device File Systems

- Devices themselves can be treated like files
- Physical devices have a name and for Linux/Unix/Mac live in the `/dev` directory
  - `tty, sda, usb, etc`
- Can be read or written to just as files (e.g. write to the terminal by just opening and writing to /dev/tty1)
- Specific I/O register access can be done with IOCTL operations
- Can expose information about the system via the file system
  - See a process' open FDs (`/proc/1000/fd/`) where 1000 is the pid of the process

Direct Memory Access (DMA)

- Large buffers of data often need to be copied between:
  - Memory and I/O (video data, network traffic, etc.)
  - Memory and Memory (OS space to user app. space)
- DMA devices are small hardware devices that copy data from a source to destination freeing the processor to do “real” work

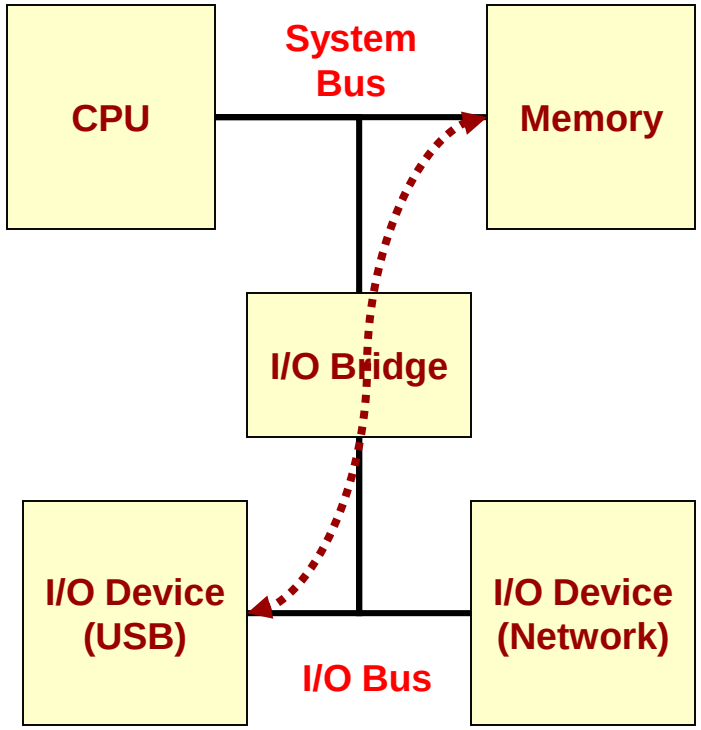


Data Transfer w/o DMA

- Without DMA, processor would have to move data using a loop
- Move 16Kwords pointed to by (%esi) to (%edi)

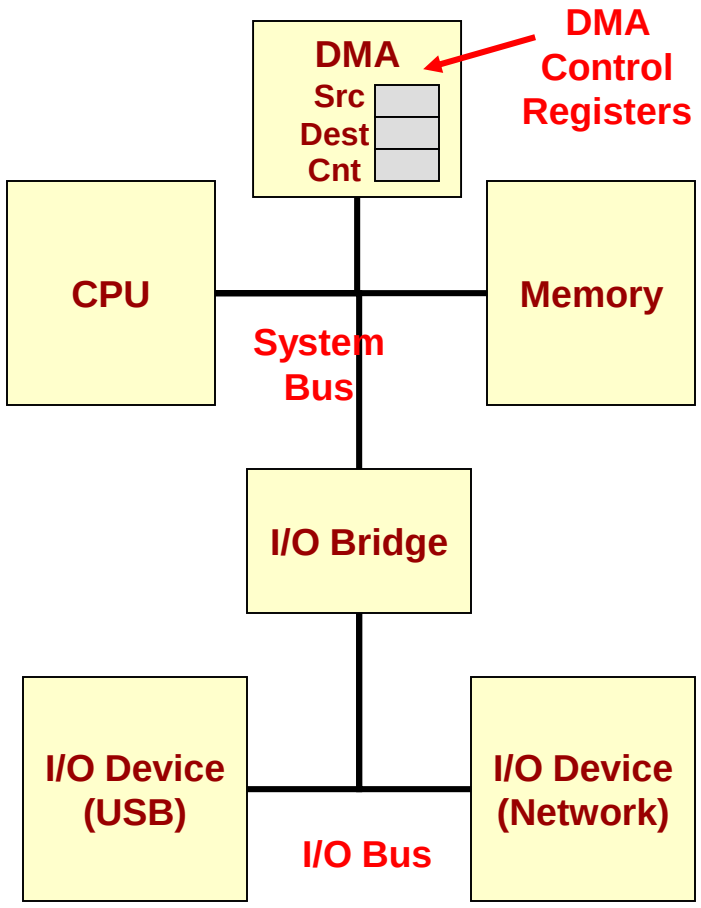
```
        move   $16384,%ecx
AGAIN:  move   (%esi),%eax
        move   %eax,(%edi)
        add    $4,%esi
        add    $4,%edi
        sub    $1,%ecx
        jnz    AGAIN
```

- Processor wastes valuable execution time moving data



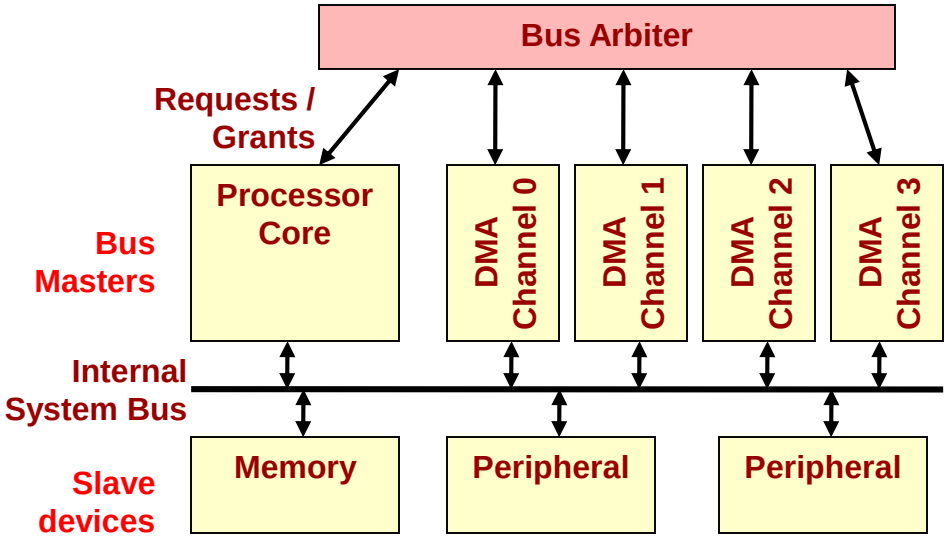
Data Transfer w/ DMA

- Processor sets values in DMA control registers
  - Source Start Address
  - Dest. Start Address
  - Byte Count
  - Control & Status (Start, Stop, Interrupt on Completion, etc.)
- DMA becomes "bus-master" (controls system bus to generate reads and writes) while processor is free to execute other code
  - Small problem: Bus will be busy
  - Hopefully, data & code needed by the CPU will reside in the processor's cache



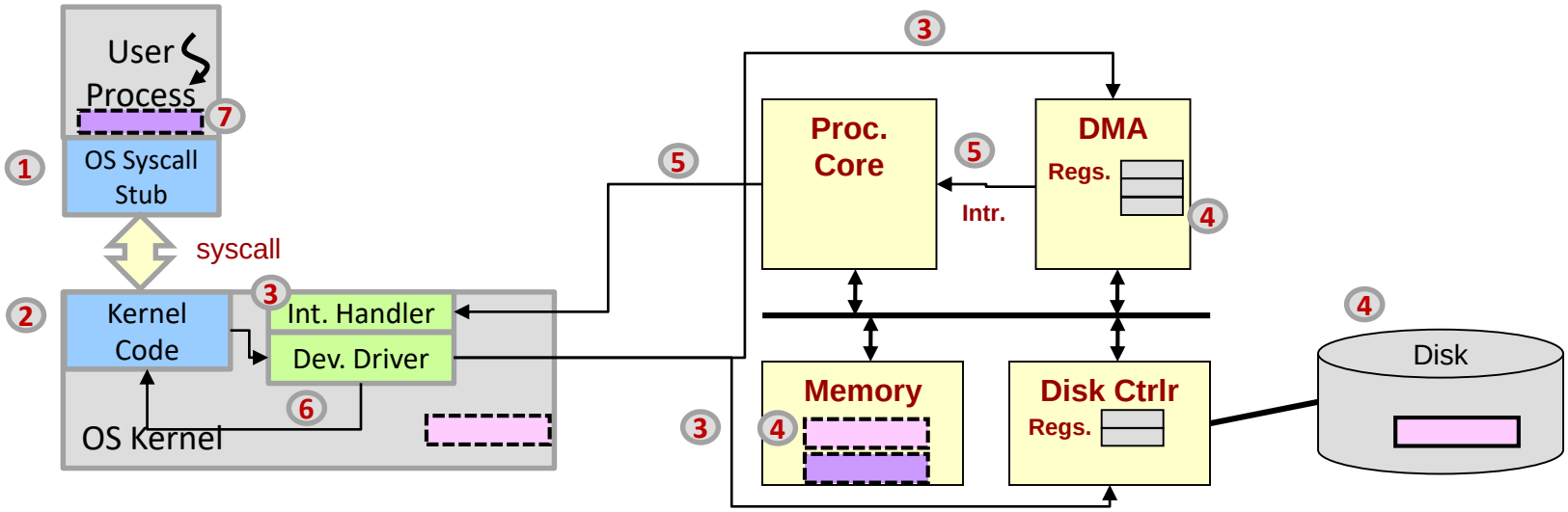
DMA Engines

- Systems usually have multiple DMA engines/channels
- Each can be configured to be started/controlled by the processor or by certain I/O peripherals
  - Network or other peripherals can initiate DMA's on their behalf
- Bus arbiter assigns control of the bus
  - Usually winning requestor has control of the bus until it relinquishes it (turns off its request signal)



Disk Access Sample Sequence

- 1) User process performs read syscall
- 2) Kernel invokes device driver
- 3) Dev. driver performs I/O control commands to disk controller and sets up DMA engine via memory mapped I/O reads/writes
  - Thread now blocks
- 4) Disk reads data and DMA transfers it to kernel area of memory (pink)
- 5) When done, processor is interrupted and the interrupt handler reschedules the blocked thread
- 6) Once awoken thread can copy the data from kernel to user space (purple)
- 7) Syscall returns and user process has its data available



SECURITY

Abstract View of Security

- Security assigns permissions to resources based on the principals involved
  - Principals are usually users or sometimes processes
  - Permissions indicate what actions are allowed
- Issues:
  - **Delegation**: Granting access to another
  - Escalate privileges to do some task
    - Take care! (Confused deputy problem)
  - Mandatory vs. Discretionary Access Control

Resources

Principals	File1	File2	ResourceA
User 1	R/W/X	R/W	R/W
User 2	R	R/X	R
User 3	R/W/X	R/W	R/W
Process X	R	R/X	R

Abstract View of Security

- Two system approaches
  - Access Control Lists (ACL): Store columns and then check permission when a user/process presents itself
  - Capability-based Systems: Each principal stores its row of permissions and presents it to the system when it attempts to access a resource
  - Essential choice is where do we store this security info (w/ resource or user)
- Which approach facilitates **delegation** most easily?

	Resources		
Principals	**File1**	**File2**	**ResourceA**
User 1	R/W/X	R/W	R/W
User 2	R	R/X	R
User 3	R/W/X	R/W	R/W
Process X	R	R/X	R

Unix/Linux/MacOS

- Uses ACL approach
  - Each resource belongs to a {user, group} pair
  - Permissions are maintained for user, group, all
  - Process is associated with the user at creation
  - When a file/resource is opened access is checked using the ACL
  - See output from 'ls -l' command
    - `-rw-rw-r--  1 redekopp bits-www    868 Jul 28  2015 README.md`

Syscall	Description
access(pathname, mode)	Checks if the current process has mode permission to access pathName
chown	Changes the owner and group of a file
chmod	Changes the permissions of a file
umask	Changes current process' default permissions for files it creates
setuid	Sets the effective user id of the current process