Operating Systems
(1DT020 & 1TT802)

Lecture 12

File System Implementation (continued)
I/O systems

May 12, 2008

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt08

Review: Disk Management Policies

- **Disk Performance:**
  - Queuing time + Controller + Seek + Rotational + Transfer
  - Rotational latency: on average ½ rotation
  - Transfer time: spec of disk depends on rotation speed and bit storage density
- **Basic entities on a disk: Files & directories**
    - » Directories represented as files
    - » File Header tracks which blocks belong to a file at which offsets within the logical file structure
- **Disk accessed using Logical Block Addressing (LBA).**
  - Every sector has integer address from zero up to max number of sectors.
  - Controller translates from address ⇒ physical position
    - » OS/BIOS must deal with bad sectors, hardware shields OS from structure of disk
- **Bitmap used to represent free space on disk**
- **Optimize placement of files' disk blocks to match access and usage patterns**
  - Access patterns: Sequential access or random access
    - » Databases are built on top of disk access to provide content based access
  - ☛Usage patterns

Goals for Today

- File System implementation
  - How to organize files on a disk
  - File system caching
  - Durability
- I/O Systems
  - Hardware Access
  - Device Drivers

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)

Designing the File System: Usage Patterns

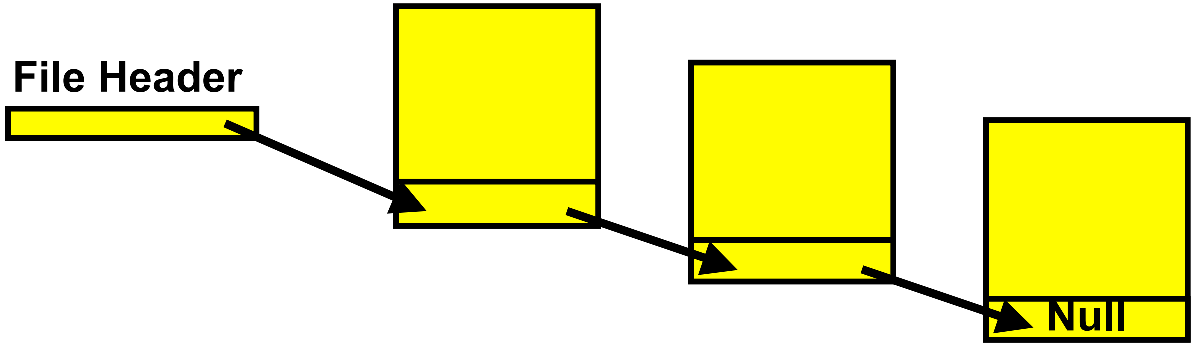
- Most files are small (for example, .login, .c files)
  - A few files are big – core files, etc.; executable are as big as all of linked object modules and statically linked library functions combined
  - However, most files are small – .java, .class’s, .o’s, .c’s, etc.
- Large files use up most of the disk space and bandwidth to/from disk
  - May seem contradictory, but a few enormous files are equivalent to an immense # of small files
- Although we will use these observations, beware usage patterns:
  - Good idea to look at usage patterns: beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns. Maybe UNIX has lots of small files because big files are really inefficient?
- Digression, danger of predicting future:
  - In 1950’s, marketing study by IBM said total worldwide need for computers was 7!

How to organize files on disk

- Goals:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, etc)
- First Technique: Continuous Allocation
  - Use continuous range of blocks in logical block space
    - Analogous to base+bounds in virtual memory
    - User says in advance how big file will be (disadvantage)
  - Search bit-map for space using best fit/first fit
    - What if not enough contiguous space for new file?
  - File Header Contains:
    - First block/LBA in file
    - File size (# of blocks)
  - Pros: Fast Sequential Access, Easy Random access
  - Cons: External Fragmentation/Hard to grow files
    - Free holes get smaller and smaller
    - Could compact space, but that would be *really* expensive
- Continuous Allocation used by IBM 360
  - Result of allocation and management cost: People would create a big file, put their file in the middle

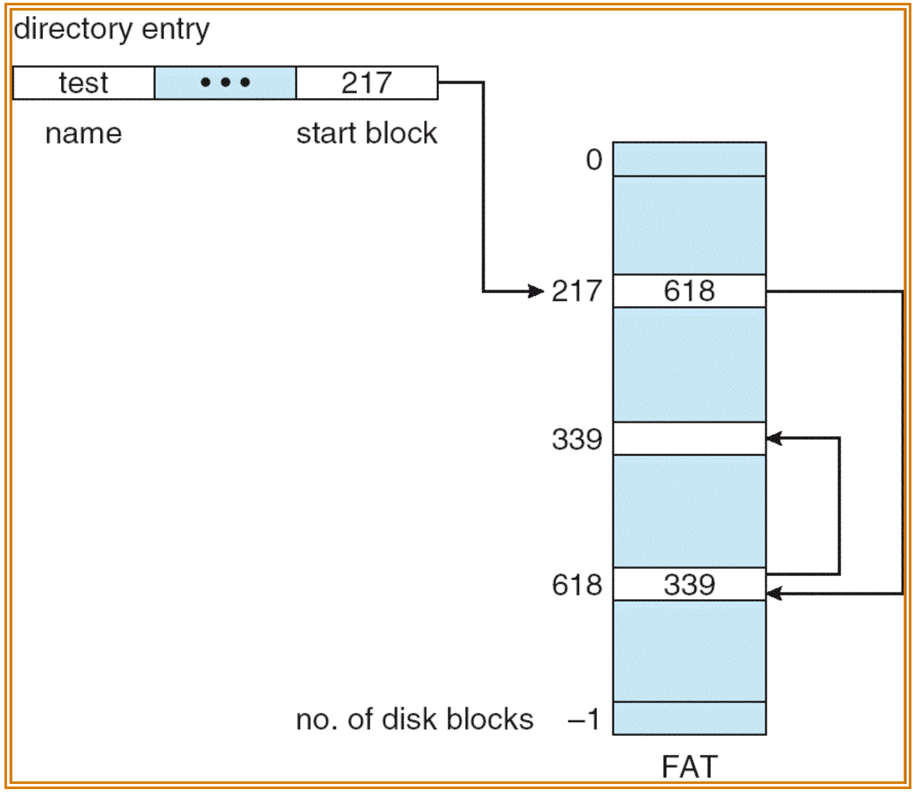
Linked List Allocation

- Second Technique: Linked List Approach
  - Each block, pointer to next on disk



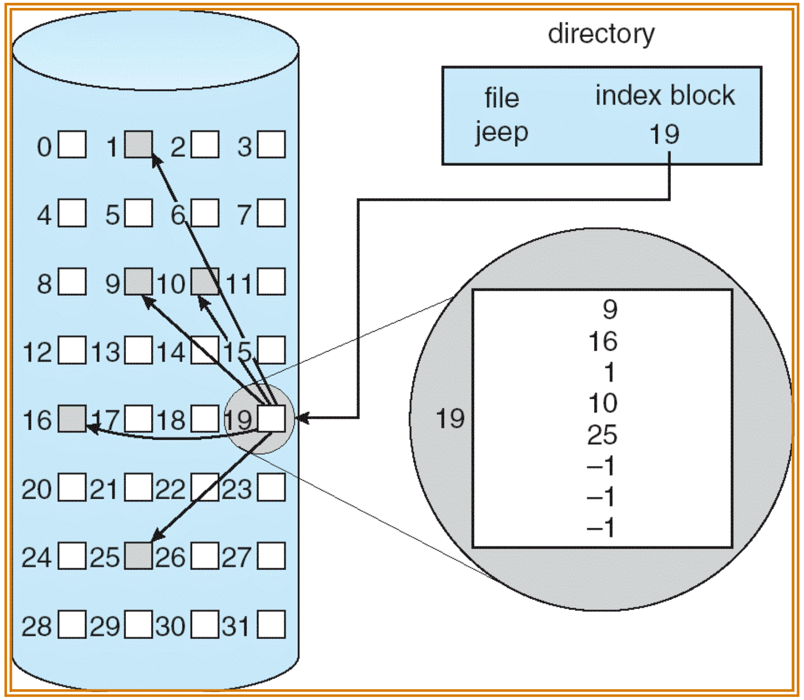
  - Pros: Can grow files dynamically, Free list same as file
  - Cons: Bad Sequential Access (seek between each block), Unreliable (lose block, lose rest of file)
  - Serious Con: Bad random access!!!!
  - Technique originally from Alto (First PC, built at Xerox)
    - » No attempt to allocate contiguous blocks

Linked Allocation: File-Allocation Table (FAT)



- **MSDOS links blocks together to create a file**
  - **Links not in blocks, but in the File Allocation Table (FAT)**
    - **FAT contains an entry for each block on the disk**
    - **FAT Entries corresponding to blocks of file linked together**
  - **Access properties:**
    - **Sequential access expensive unless FAT cached in memory**
    - **Random access expensive always, but *really* expensive if FAT not cached in memory**

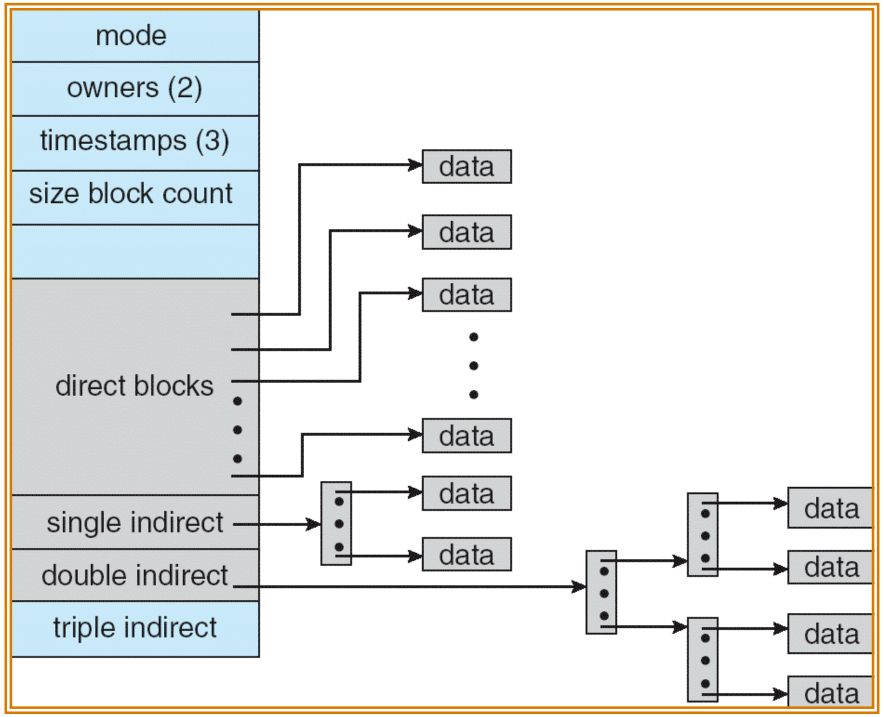
Indexed Allocation



- **Third Technique: Indexed Files (VMS)**
  - **System Allocates file header block to hold array of pointers big enough to point to all blocks**
    - **User pre-declares max file size;**
  - **Pros: Can easily grow up to space allocated for index**
    Random access is fast
  - **Cons: Clumsy to grow file bigger than table size**
    Still lots of seeks: blocks may be spread over disk

Multilevel Indexed Files (UNIX 4.1)

- **Multilevel Indexed Files: Like multilevel address translation (from UNIX 4.1 BSD)**
  - **Key idea: efficient for small files, but still allow big files**



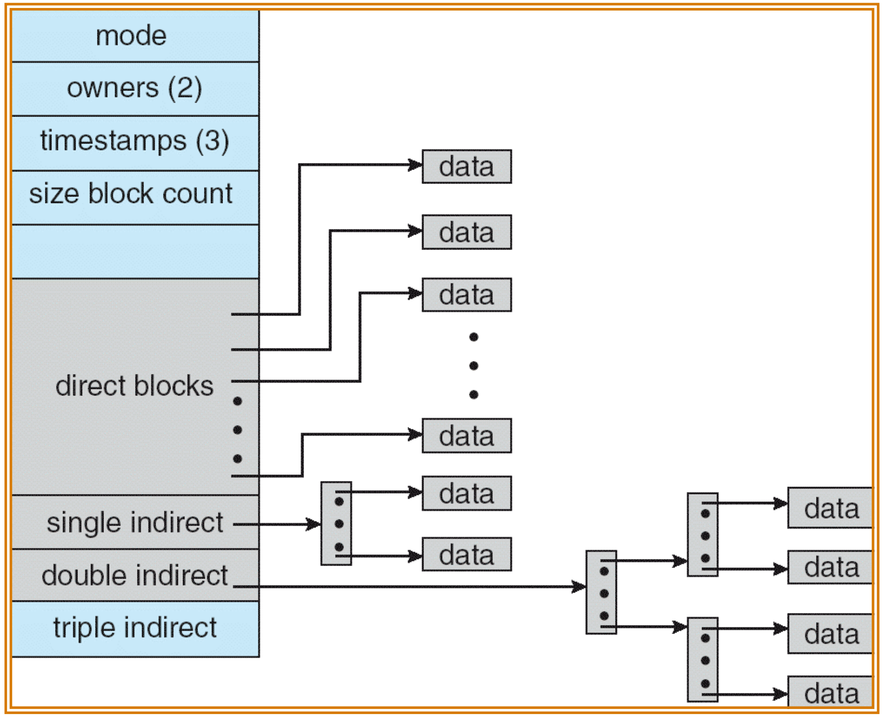
- **File hdr contains 13 pointers**
  - **Fixed size table, pointers not all equivalent**
  - **This header is called an "inode" in UNIX**
- **File Header format:**
  - **First 10 pointers are to data blocks**
  - **Ptr 11 points to "indirect block" containing 256 block ptrs**
  - **Pointer 12 points to "doubly indirect block" containing 256 indirect block ptrs for total of 64K blocks**
  - **Pointer 13 points to a triply indirect block (16M blocks)**

Multilevel Indexed Files (UNIX 4.1): Discussion

- **Basic technique places an upper limit on file size that is approximately 16Gbytes**
  - **Designers thought this was bigger than anything anyone would need. Much bigger than a disk at the time…**
  - **Fallacy: today, EOS producing 2TB of data per day**

- **Pointers get filled in dynamically: need to allocate indirect block only when file grows > 10 blocks**
  - **On small files, no indirection needed**

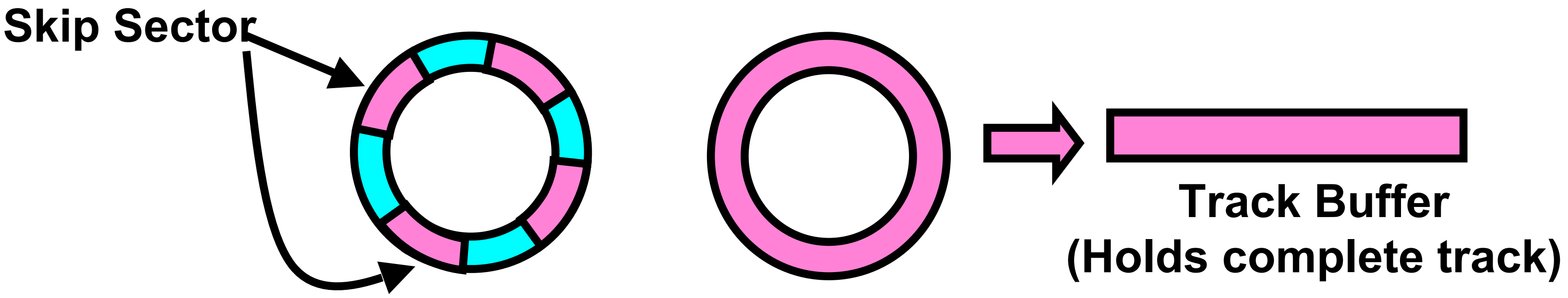
Example of Multilevel Indexed Files



- Sample file in multilevel indexed format:
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data
- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect blocks
    (four I/Os per block!)

Attack of the Rotational Delay

- Another problem: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



Skip Sector

Track Buffer
(Holds complete track)

-
  - Solution1: Skip sector positioning ("interleaving")
    - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
  - Solution2: Read ahead: read next block right after first, even if application hasn't asked for it yet.
    - » This can be done either by OS (read ahead)
    - » By disk itself (track buffers). Many disk controllers have internal RAM that allows them to read a complete track
- Important Aside: Modern disks+controllers do many complex things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

File System Caching

- **Key Idea: Exploit locality by caching data in memory**
  - **Name translations: Mapping from paths→inodes**
  - **Disk blocks: Mapping from block address→disk content**
- **Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations**
  - **Can contain "dirty" blocks (blocks yet on disk)**
- **Replacement policy? LRU**
  - **Can afford overhead of timestamps for each disk block**
  - **Advantages:**
    - » **Works very well for name translation**
    - » **Works well in general as long as memory is big enough to accommodate a host's working set of files.**
  - **Disadvantages:**
    - » **Fails when some application scans through file system, thereby flushing the cache with data used only once**
    - » **Example:** `find . -exec grep foo {} \;`
- **Other Replacement Policies?**
  - **Some systems allow applications to request other policies**
  - **Example, 'Use Once':**
    - » **File system can discard blocks as soon as they are used**

File System Caching (con't)

- **Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?**
  - Too much memory to the file system cache ⇒ won't be able to run many applications at once
  - Too little memory to file system cache ⇒ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
- **Read Ahead Prefetching:** fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
  - Elevator algorithm can efficiently interleave groups of prefetches from concurrent applications
  - How much to prefetch?
    - » Too many imposes delays on requests by other applications
    - » Too few causes many seeks (and rotational delays) among concurrent file requests

File System Caching (con’t)

- **Delayed Writes:** Writes to files not immediately sent out to disk
  - – Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
    - » Enabled by presence of buffer cache: can leave written file blocks in cache for a while
    - » If some other application tries to read data before written to disk, file system will read from cache
  - – Flushed to disk periodically (e.g. in UNIX, every 30 sec)
  - – Advantages:
    - » Disk scheduler can efficiently order lots of requests
    - » Disk allocation algorithm can be run with correct size value for a file
    - » Some files need never get written to disk! (e..g temporary scratch files written /tmp often don’t exist for 30 sec)
  - – Disadvantages
    - » What if system crashes before file has been written out?
    - » Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

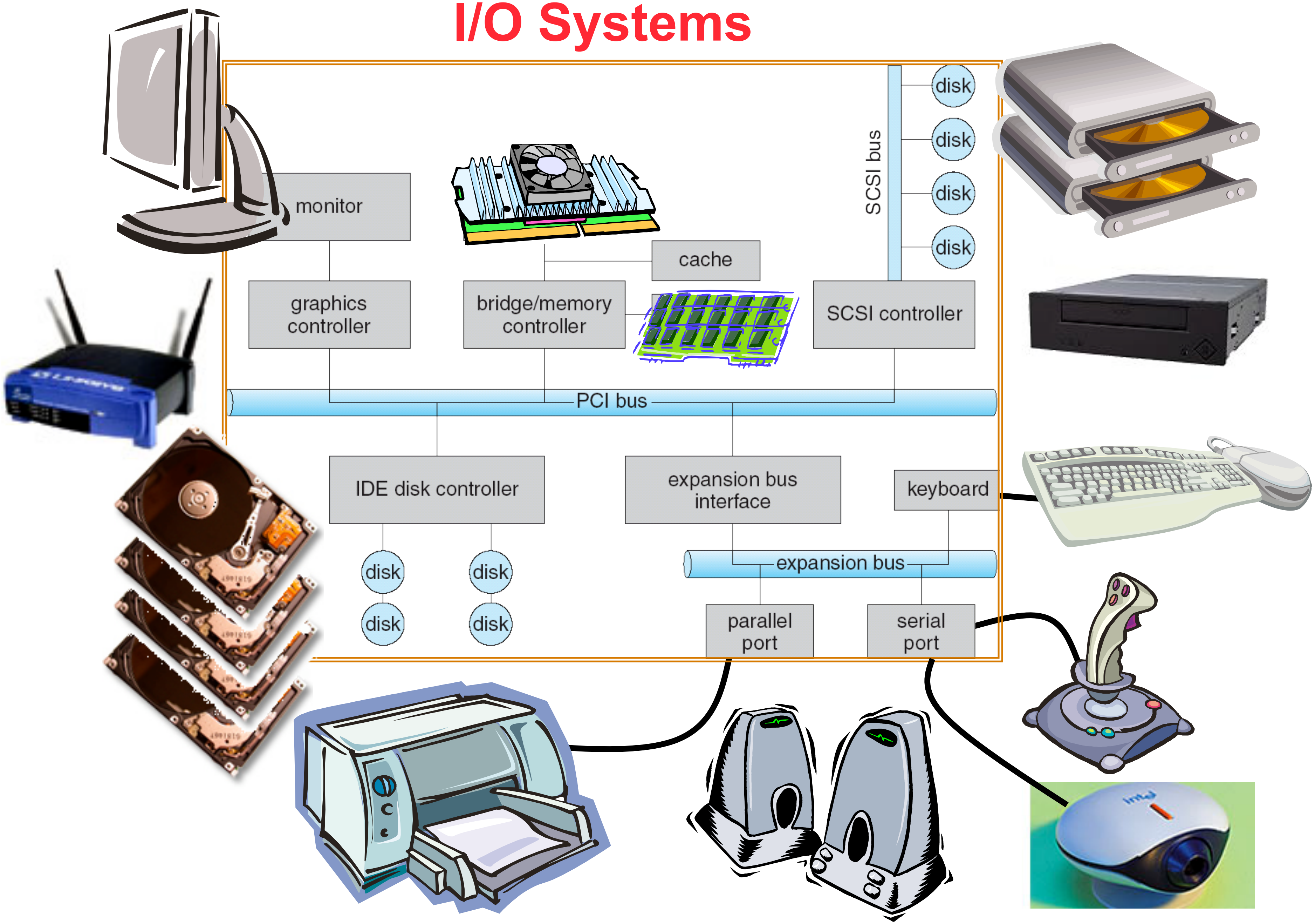
How to make file system durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache.
- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: independence of failure
    - » Could put copies on one disk, but if disk head fails…
    - » Could put copies on different disks, but if server fails…
    - » Could put copies on different servers, but if building is struck by lightning….
    - » Could put copies on servers in different continents…
- RAID: Redundant Arrays of Inexpensive Disks
  - Data stored on multiple disks (redundancy)
  - Either in software or hardware
    - » In hardware case, done by disk controller; file system may not even know that there is more than one disk in use

Log Structured and Journaled File Systems

- Better reliability through use of log
    - All changes are treated as *transactions*
        - A transaction either happens *completely* or *not at all*
    - A transaction is *committed* once it is written to the log
        - Data forced to disk for reliability
        - Process can be accelerated with NVRAM
    - Although File system may not be updated immediately, data preserved in the log
- Difference between “Log Structured” and “Journaled”
    - Log Structured Filesystem (LFS): data stays in log form
    - Journaled Filesystem: Log used for recovery
- For Journaled system:
    - Log used to asynchronously update filesystem
        - Log entries removed after used
    - After crash:
        - Remaining transactions in the log performed (“Redo”)
- Examples of Journaled File Systems:
    - Ext3 (Linux), XFS (Unix), NTFS (Windows)

I/O Systems



monitor

graphics controller

bridge/memory controller

cache

SCSI controller

SCSI bus

disk

disk

disk

disk

PCI bus

IDE disk controller

expansion bus interface

keyboard

disk

disk

disk

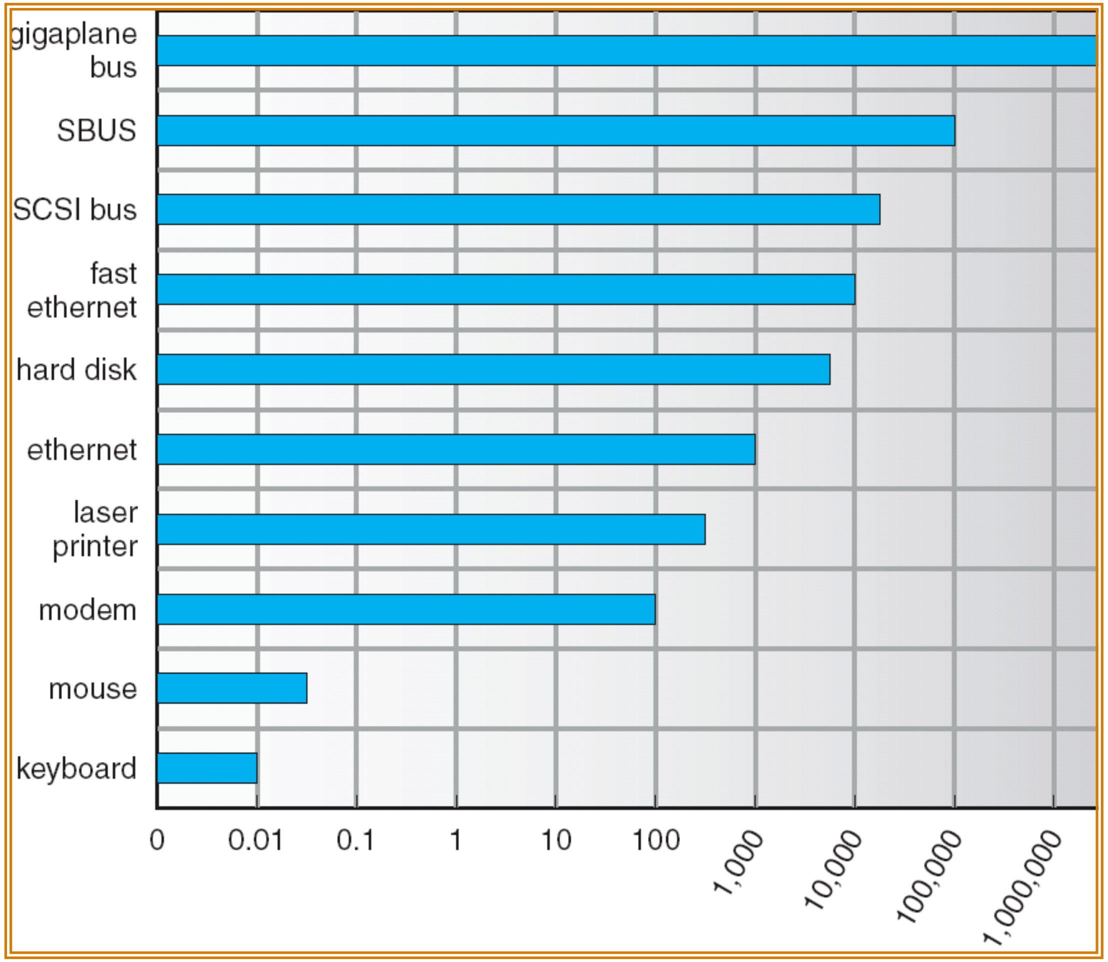
disk

expansion bus

parallel port

serial port

Example Device-Transfer Rates (Sun Enterprise 6000)



- **Device Rates vary over many orders of magnitude**
  - **System better be able to handle this wide range**
  - **Better not have high overhead/byte for fast devices!**
  - **Better not waste time waiting for slow devices**

The Goal of the I/O Subsystem

- **Provide Uniform Interfaces, Despite Wide Range of Different Devices**
  - This code works on many different devices:

```
FILE fd = fopen(“/dev/something”,”rw”);
for (int i = 0; i < 10; i++) {
   fprintf(fd,”Count %d\n”,i);
}
close(fd);
```

  - Why? Because code that controls devices (“device driver”) implements standard interface.

Want Standard Interfaces to Devices

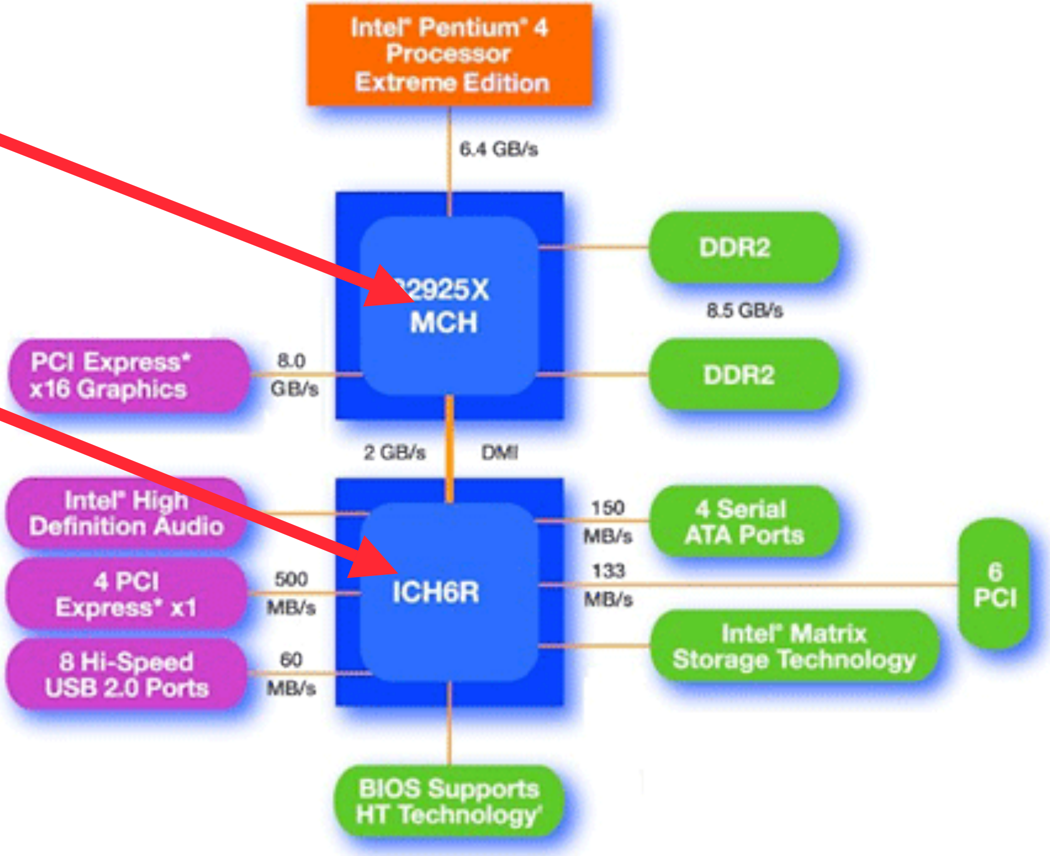
- **Block Devices:** *e.g.* disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- **Character Devices:** *e.g.* keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- **Network Devices:** *e.g.* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include socket interface
    - Separates network protocol from network operation
    - Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

How Does User Deal with Timing?

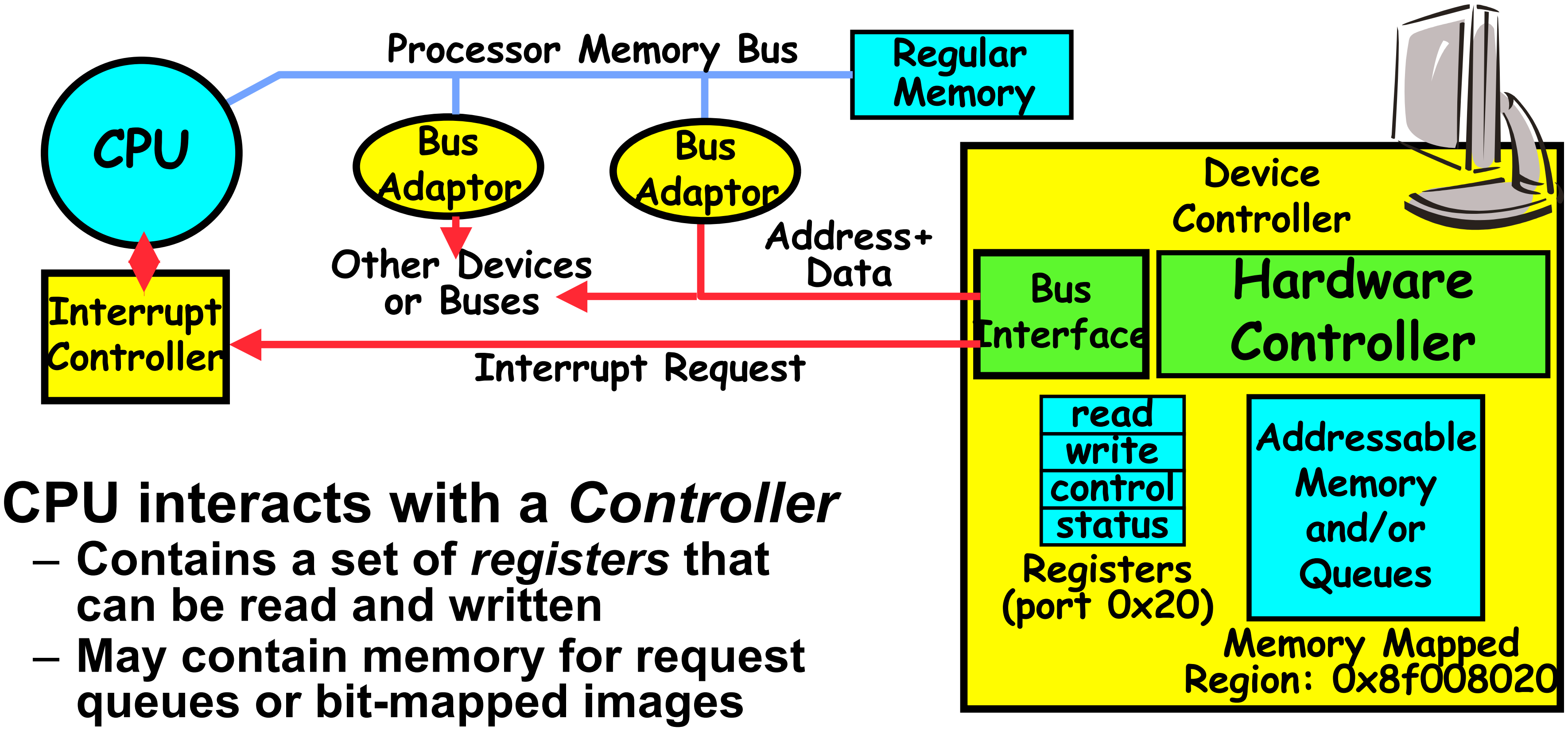
- **Blocking Interface:** “Wait”
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data
- **Non-blocking Interface:** “Don’t Wait”
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing
- **Asynchronous Interface:** “Tell Me Later”
  - When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user

Main components of Intel Chipset: Pentium 4

- **Northbridge:**
  - **Handles memory**
  - **Graphics**
- **Southbridge: I/O**
  - **PCI bus**
  - **Disk controllers**
  - **USB controllers**
  - **Audio**
  - **Serial I/O**
  - **Interrupt controller**
  - **Timers**



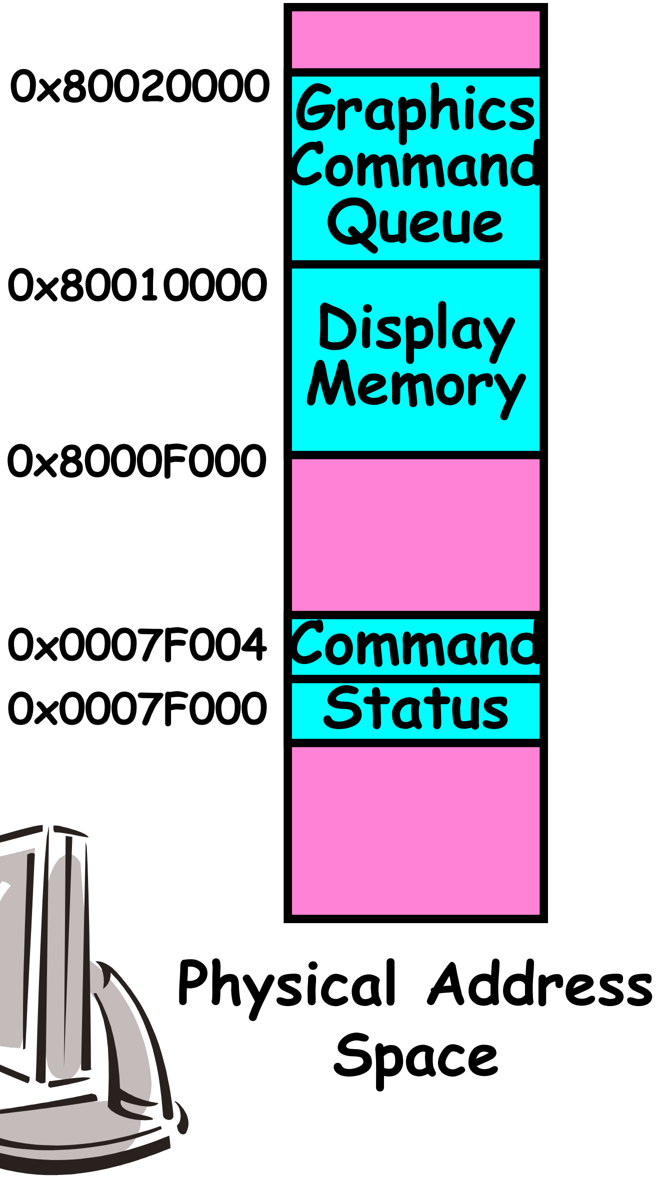
How does the processor actually talk to the device?



- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - I/O instructions: in/out instructions
    - » Example from the Intel architecture: `out 0x21,AL`
  - Memory mapped I/O: load/store instructions
    - » Registers/memory appear in physical address space
    - » I/O accomplished with load and store instructions

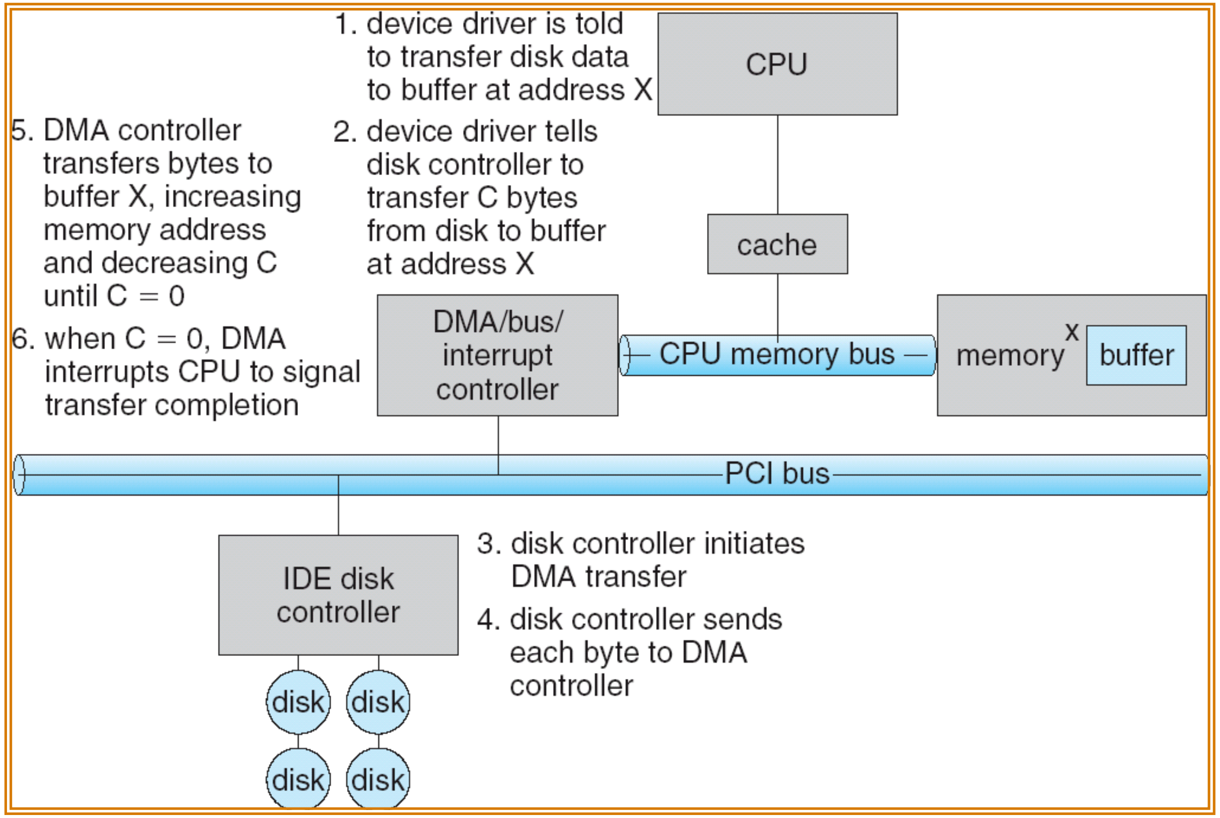
Example: Memory-Mapped Display Controller

- **Memory-Mapped:**
  - Hardware maps control registers and display memory into physical address space
    - » Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - » Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - » Say enter a set of triangles that describe some scene
    - » Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - » Say render the above scene
    - » Addr: 0x0007F004
- **Can protect with page tables**



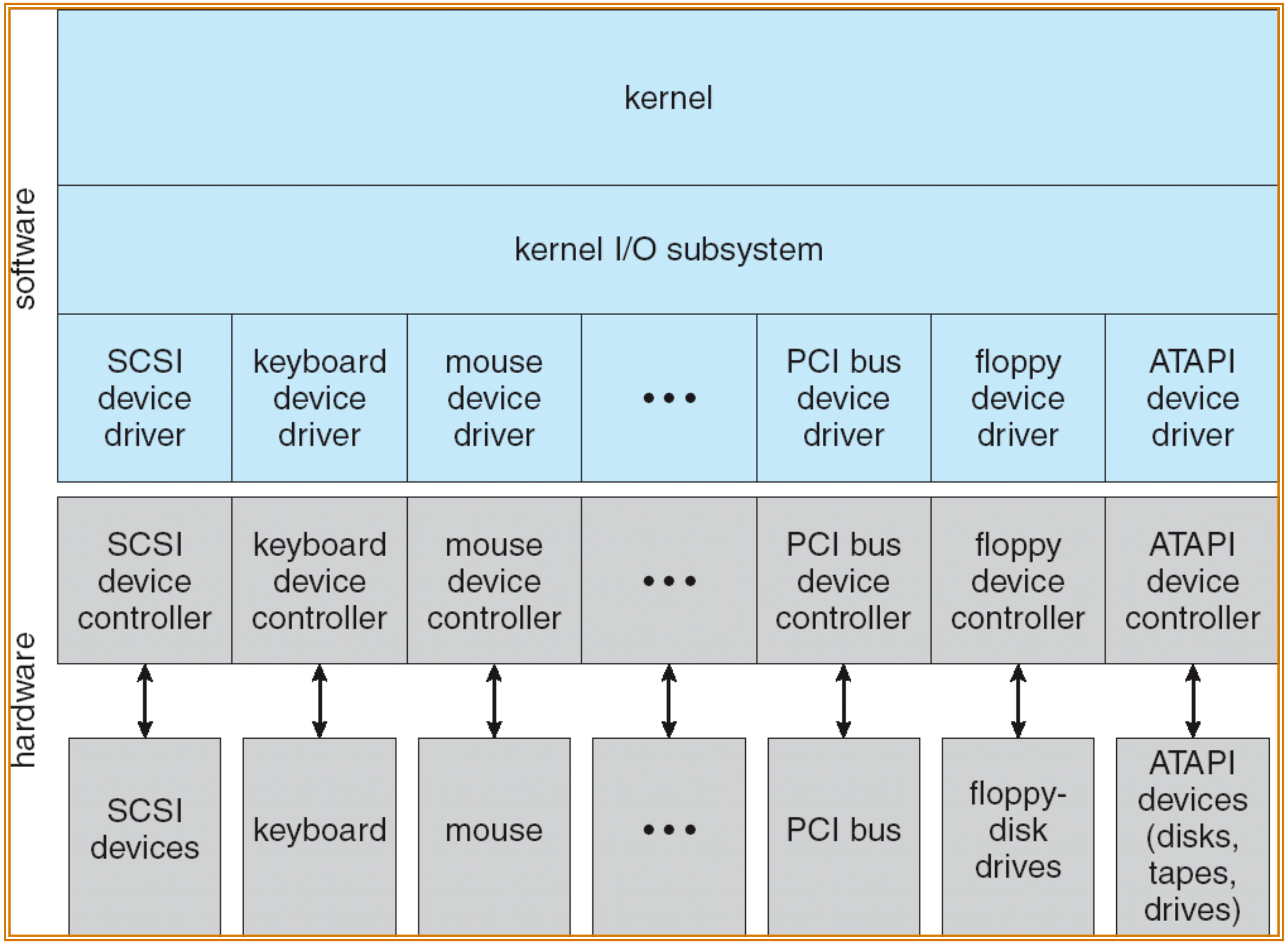
Transfering Data To/From Controller

- **Programmed I/O:**
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
- **Direct Memory Access:**
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly
- **Sample interaction with DMA controller (from book):**



1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion

A Kernel I/O Structure



Device Drivers

- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - » Implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - » This is the kernel's interface to the device driver
    - » Top half will *start* I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - » Gets input or transfers next block of output
    - » May wake sleeping threads if I/O now complete

Life Cycle of An I/O Request

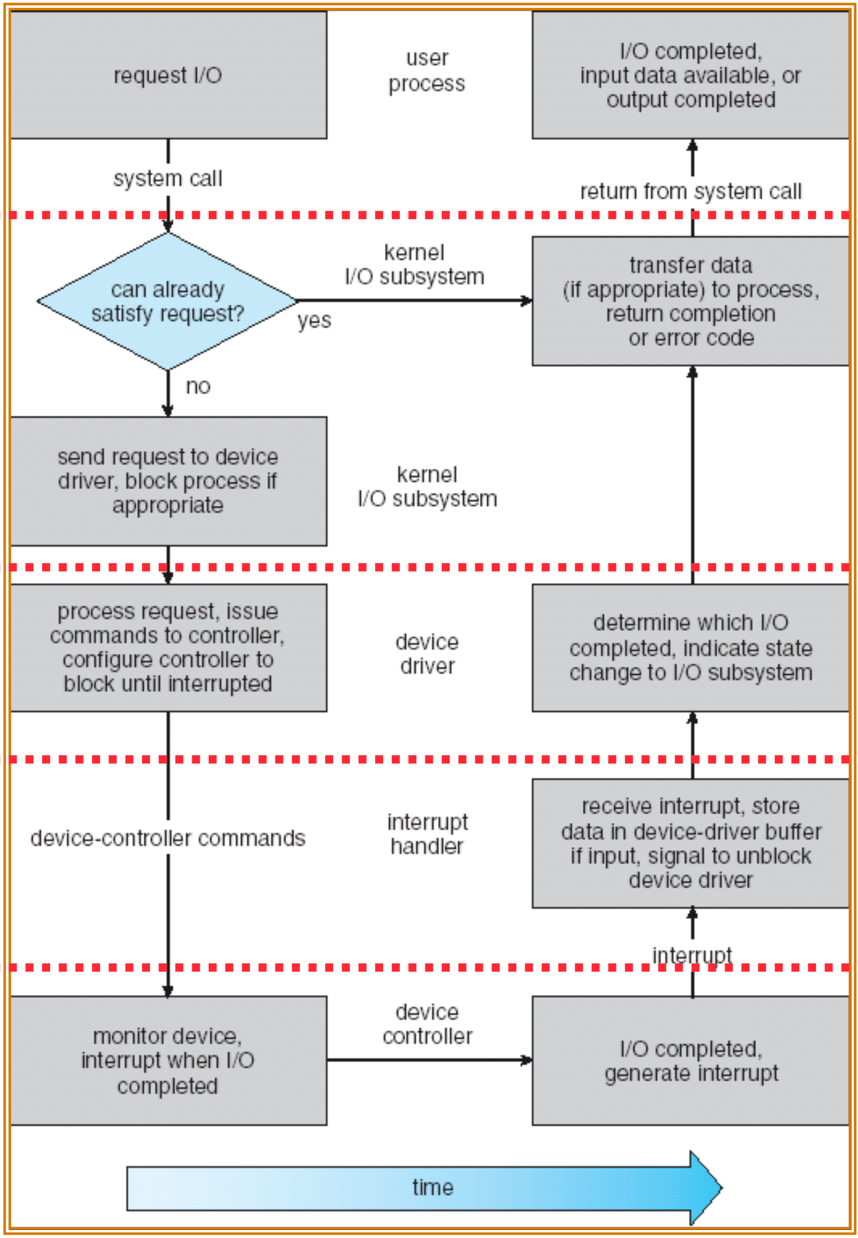
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware



I/O Device Notifying the OS

- **The OS needs to know when:**
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- **I/O Interrupt:**
  - Device generates an interrupt whenever it needs service
  - Handled in bottom half of device driver
    - » Often run on special kernel-level stack
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- **Polling:**
  - OS periodically checks a device-specific status register
    - » I/O device puts completion information in status register
    - » Could use timer to invoke lower half of drivers occasionally
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- **Actual devices combine both polling and interrupts**
  - For instance: High-bandwidth network device:
    - » Interrupt for first incoming packet
    - » Poll for following packets until hardware empty

Summary

- **Multilevel Indexed Scheme**
  - Inode contains file info, direct pointers to blocks,
  - indirect blocks, doubly indirect, etc..
- **Buffer cache used to increase performance**
  - Read Ahead Prefetching and Delayed Writes
- **I/O Devices Types:**
  - Many different speeds (0.1 bytes/sec to GBytes/sec)
  - Different Access Patterns: block, char, net devices
  - Different Access Timing: Non-/Blocking, Asynchronous
- **I/O Controllers: Hardware that controls actual device**
  - CPU accesses thru I/O insts, ld/st to special phy memory
  - Report results thru interrupts or a status register polling
- **Device Driver: Device-specific code in kernel**