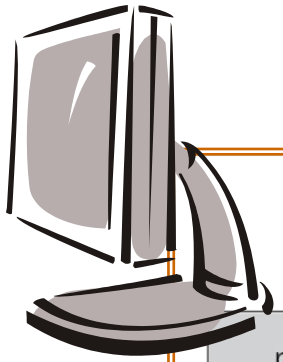
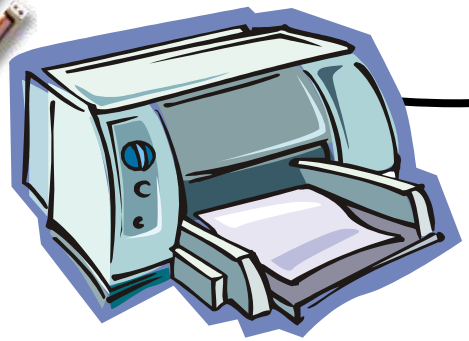
CSL373: Lecture 16
I/O and Disks

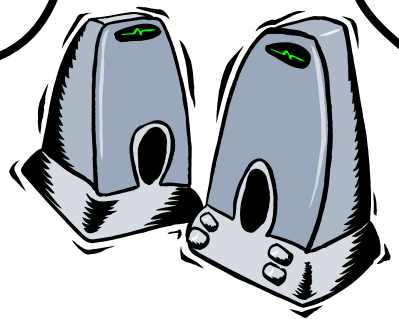
The Requirements of I/O

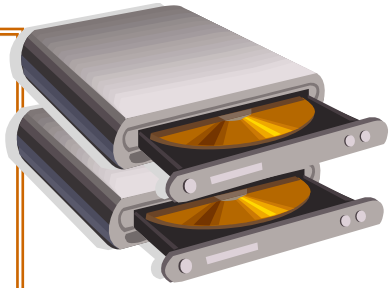
- So far in this course:
  - We have learned how to manage CPU, memory
- What about I/O?
  - Without I/O, computers are useless (disembodied brains?)
  - But… thousands of devices, each slightly different
    - How can we standardize the interfaces to these devices?
  - Devices unreliable: media failures and transmission errors
    - How can we make them reliable???
  - Devices unpredictable and/or slow
    - How can we manage them if we don't know what they will do or how they will perform?
- Some operational parameters:
  - Byte/Block
    - Some devices provide single byte at a time (*e.g.* keyboard)
    - Others provide whole blocks (*e.g.* disks, networks, etc)
  - Sequential/Random
    - Some devices must be accessed sequentially (*e.g.* tape)
    - Others can be accessed randomly (*e.g.* disk, cd, etc.)
  - Polling/Interrupts
    - Some devices require continual monitoring
    - Others generate interrupts when they need service

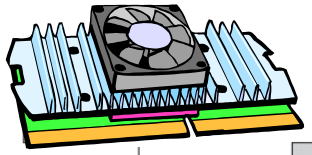
Modern I/O Systems

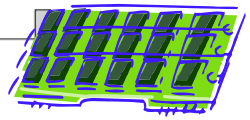




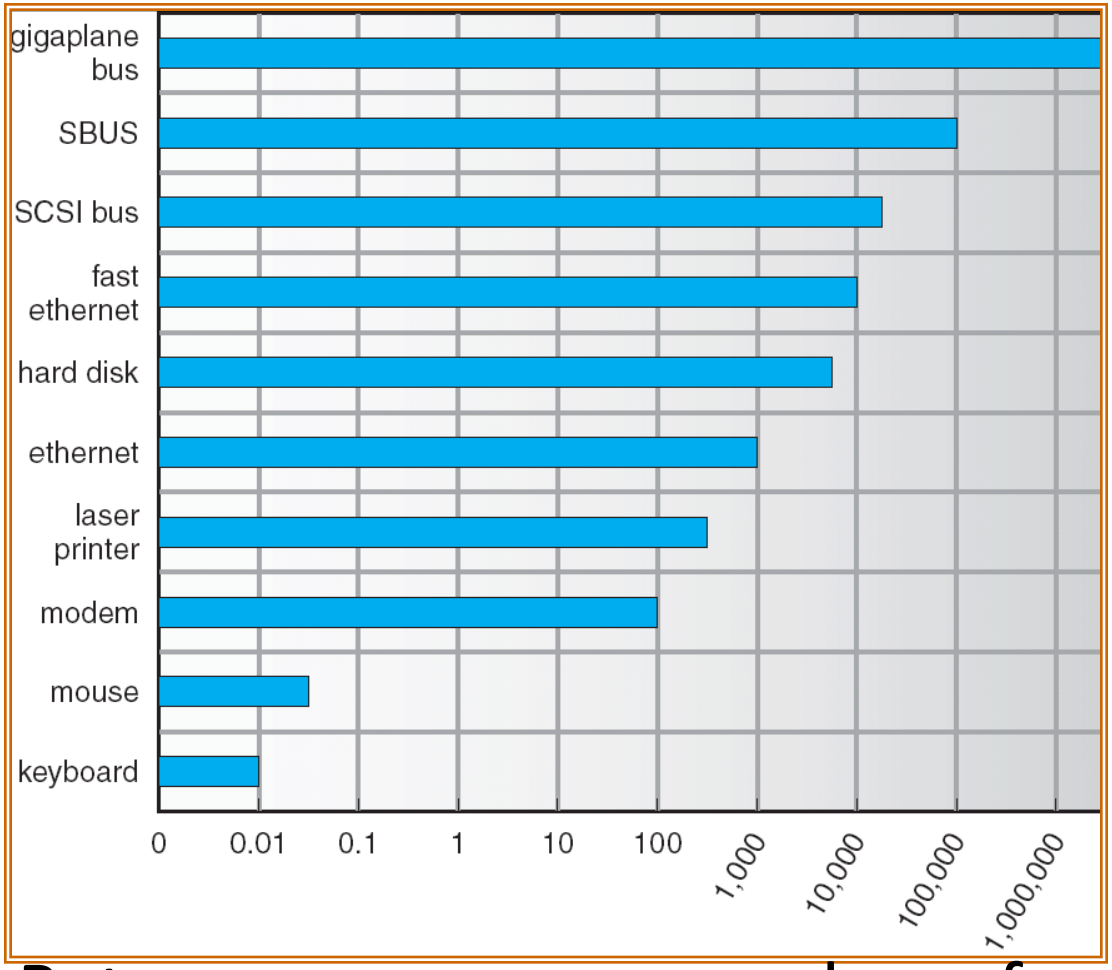








Example Device-Transfer Rates (Sun Enterprise 6000)



- 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.
- We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  - Can only scratch surface!

Want Standard Interfaces to Devices

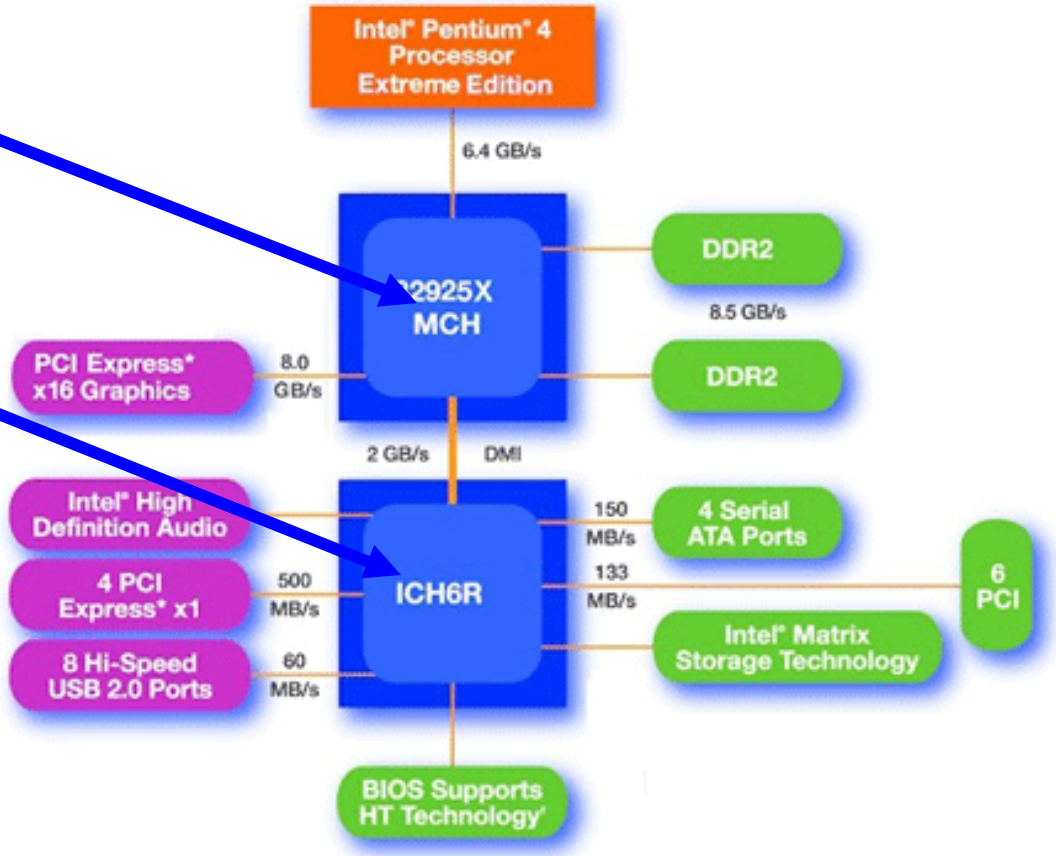
- Block Devices: *e.g.* disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character Devices: *e.g.* keyboards, mice, serial ports, some USB devices
  - Single 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?

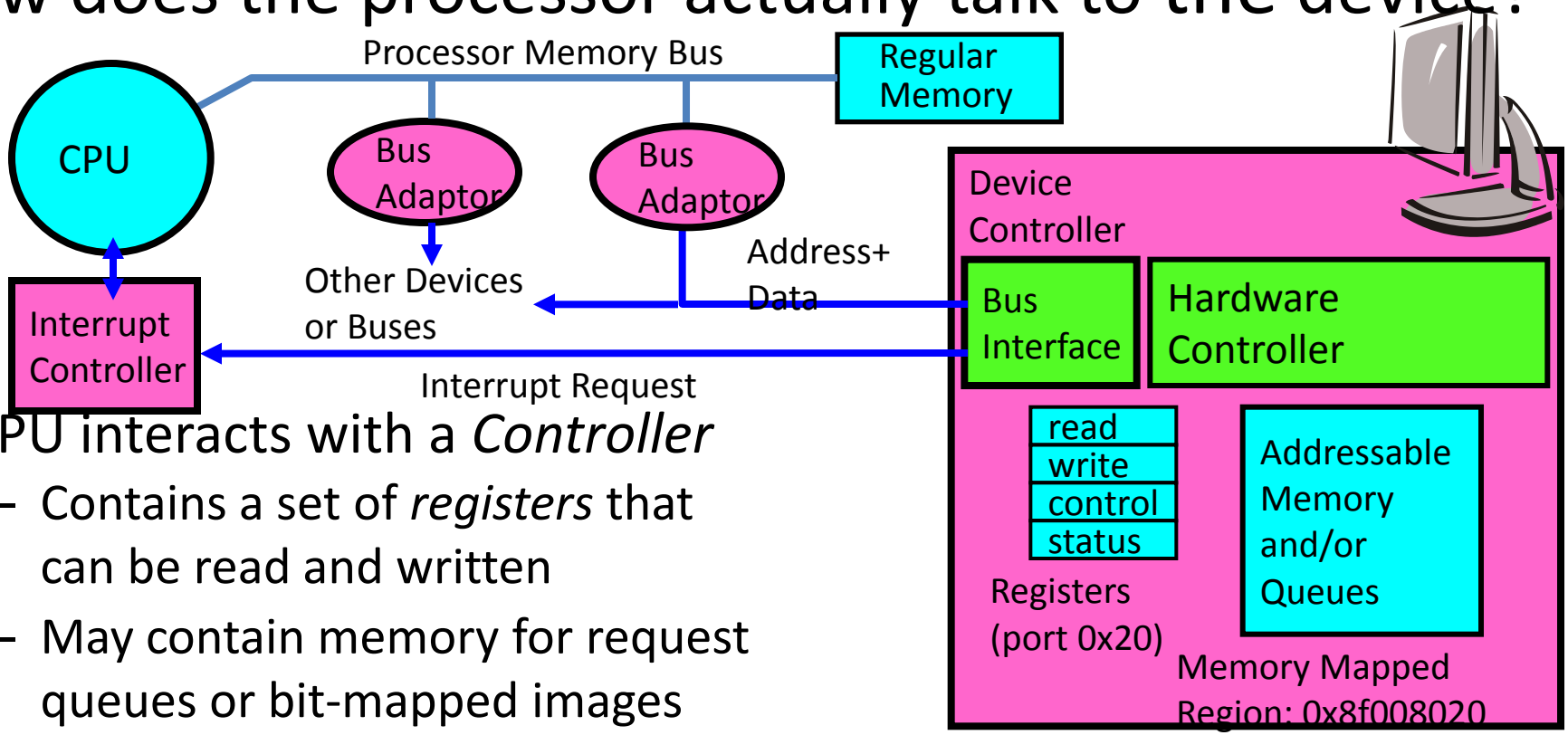
- Blocking Interface: "Wait"
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data
- Non-blocking Interface: "Don't Wait"
  - Returns quickly from read or write 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?

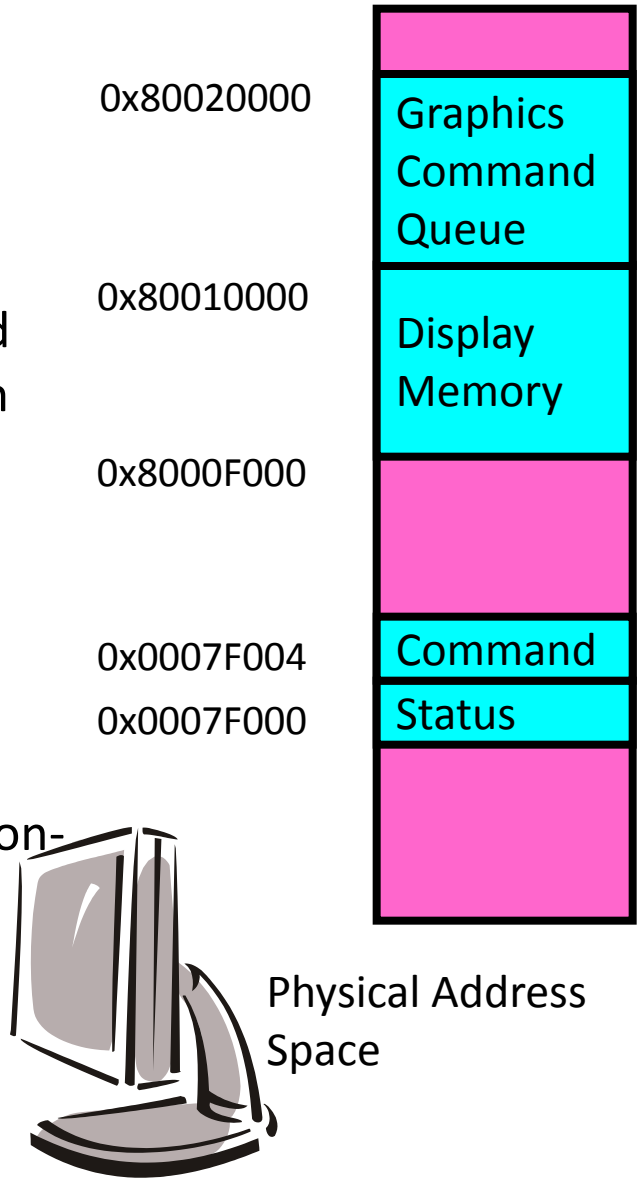


- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images

- 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

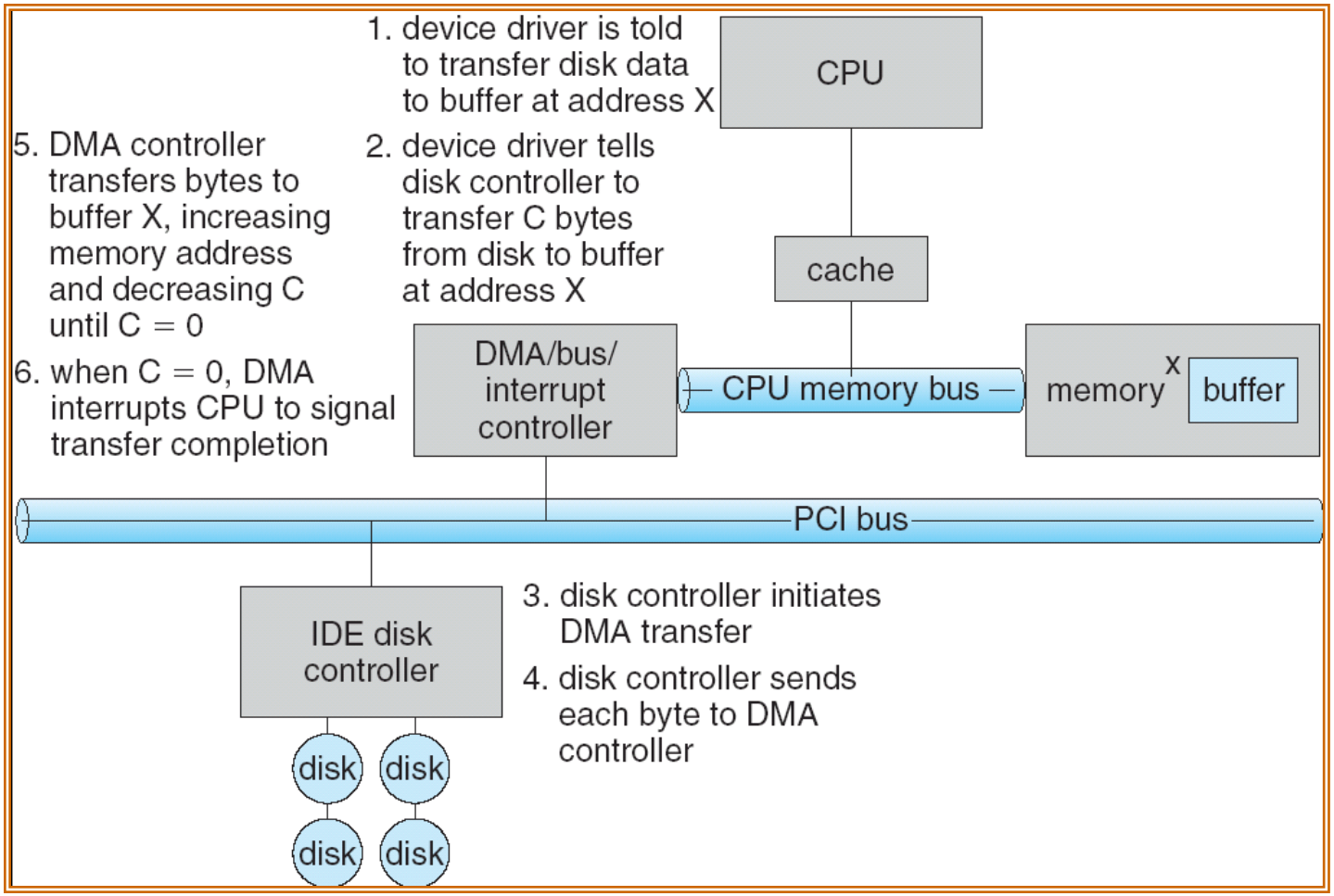


Transfering Data To/From Controller

- 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

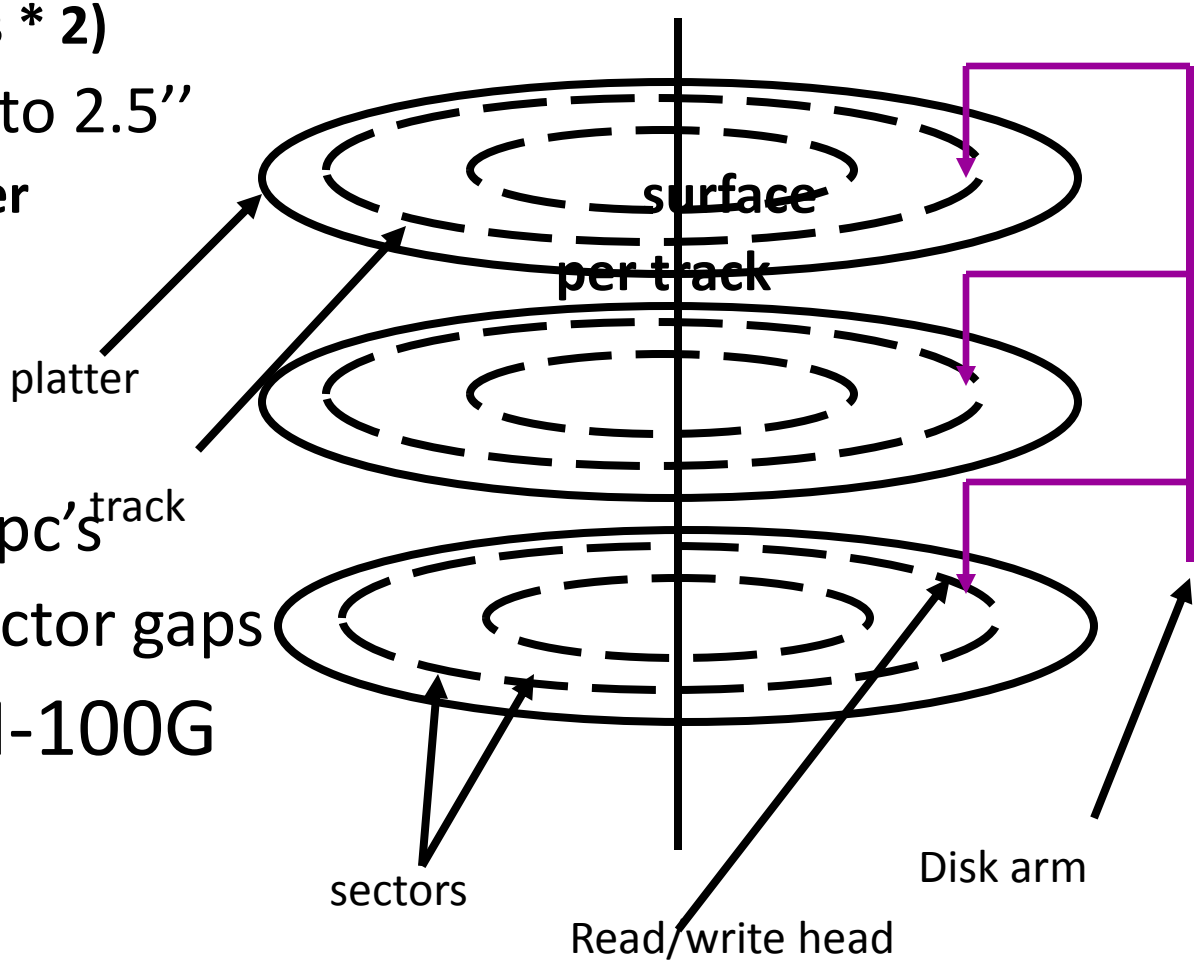
- Direct Memory Access:
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly

Sample interaction with DMA controller



What do disks look like?

- **2-30 heads (platters * 2)**
  - diameter 14’’ to 2.5’’
- **700-20480 tracks per**
- **16-1600 sectors**
- **sector size:**
  - 64-8k bytes
  - 512 for most pc’s
  - note: inter-sector gaps
- capacity: 20M-100G



Some modern disks drives

	Barracuda 180	Cheetah X15-26LP
Capacity	181GB	36.7GB
Platter/Heads	12/24	4/8
Cylinders	24,247	18,479
Sectors/track	~609	~485
Speed	7200RPM	15000RPM
Latency (ms)	4.17	2.0
Avg seek (ms)	7.4/8.2	3.6/4.2
Track-2-track(ms)	0.8/1.1	0.3/0.4

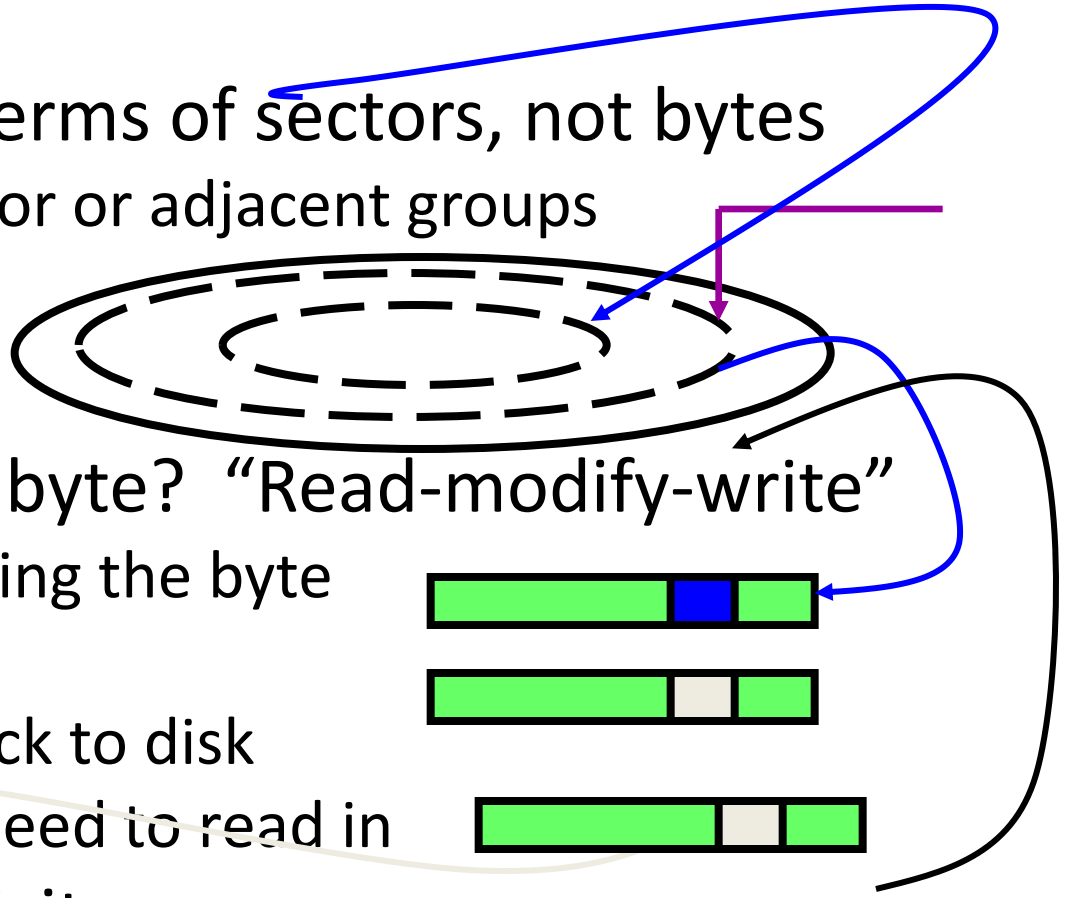
Disk vs. Memory

- Smallest write: sector
- Atomic write = sector
- Random access: 5ms
  - not improving
- Sequential access: 200MB/s
- Cost $.002MB
- Crash: doesn't matter ("non-volatile")

- (usually) bytes
- byte, word
- 50 ns
  - getting faster all the time
- 200-1000MB/s
- $.15MB
- contents gone ("volatile")

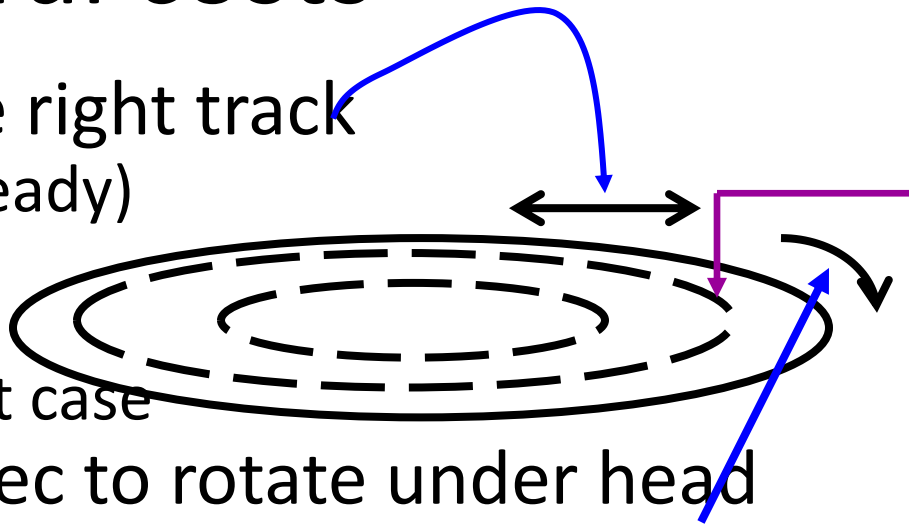
Some useful facts

- Disk reads/writes in terms of sectors, not bytes
  - read/write single sector or adjacent groups



- How to write a single byte? "Read-modify-write"
  - read in sector containing the byte
  - modify that byte
  - write entire sector back to disk
  - key: if cached, don't need to read in
- Sector = unit of atomicity.
  - sector write done completely, even if crash in middle
    - (disk saves up enough momentum to complete)
  - larger atomic units have to be synthesized by OS

Some useful costs



- Seek: move disk arm to the right track
  - best case: 0ms (on track already)
  - worst: ~30-50ms
    (move over entire disk)
  - average: 10-20ms, 1/3 worst case
- Rotational delay: wait for sec to rotate under head
  - best: 0ms (over sector)
  - worst: ~16ms (entire rotation)
  - average: ~8ms (1/2 worst case)
- Transfer bandwidth: suck bits off of device
- Cost of disk access? Seek + rotation + transfer time
  - read a single sector: 10ms + 8ms + 50us ~= 18ms
  - Cool: read an entire track? Seek + transfer! (why?)

Some useful trends

- Disk bandwidth and cost/bit improving exponentially
  - similar to CPU speed, memory size, etc.
- Seek time and rotational delay improving *very* slowly
  - why? require moving physical object (disk arm)
- Some implications:
  - disk accesses a huge system bottleneck & getting worse
  - bandwidth increase lets system (pre-)fetch large chunks for about the same cost as small chunk.
    - Result? trade bandwidth for latency if you can get lots of related stuff.
    - How to get related stuff? Cluster together on disk
  - Memory size increasing faster than typical workload size
    - More and more of workload fits in file cache
    - disk traffic changes: mostly writes and new data