CS162
Operating Systems and Systems Programming
Lecture 19

General I/O, Storage Devices

November 7th, 2024

Prof. Ion Stoica

http://cs162.eecs.Berkeley.edu

Reverse Page Mapping (Sometimes called "Coremap")

- When evicting a page frame, how to know which PTEs to invalidate?
  - Hard in the presence of shared pages (forked processes, shared memory, …)
- Reverse mapping mechanism must be very fast
  - Must hunt down all page tables pointing at given page frame when freeing a page
  - Must hunt down all PTEs when seeing if pages "active"
- Implementation options:
  - For every page descriptor, keep linked list of page table entries that point to it
    - » Management nightmare – expensive
  - Linux: Object-based reverse mapping
    - » Link together memory region descriptors instead (much coarser granularity)

Allocation of Page Frames (Memory Pages)

- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory? Different fractions?
  - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - » instruction is 6 bytes, might span 2 pages
    - » 2 pages to handle *from*
    - » 2 pages to handle *to*
- Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Local replacement – each process selects from only its own set of allocated frames

Fixed/Priority Allocation

- Equal allocation (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes → process gets 20 frames
- Proportional allocation (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:

    s_i = size of process p_i and $S = \sum s_i$

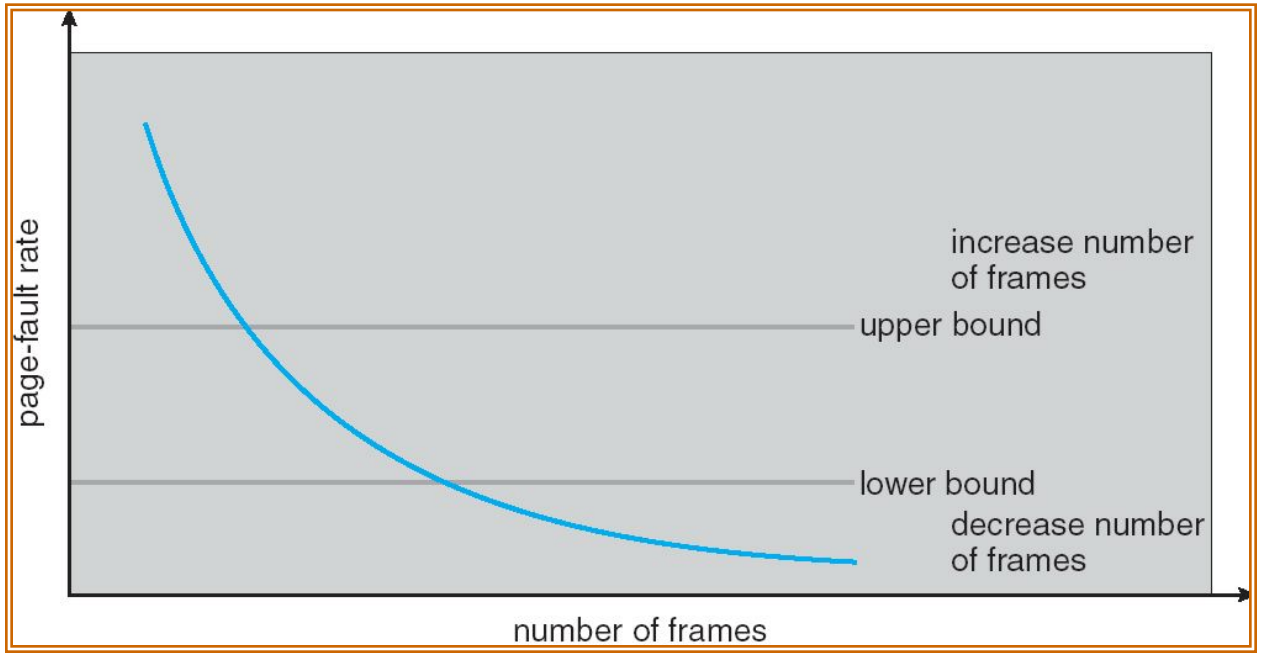
    m = total number of physical frames in the system

    a_i = (allocation for p_i) $= \frac{s_i}{S} \times m$
- Priority Allocation:
  - Proportional scheme using priorities rather than size
    - » Same type of computation as previous scheme
  - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number

- Perhaps we should use an adaptive scheme instead???
  - What if some application just needs more memory?

Page-Fault Frequency Allocation

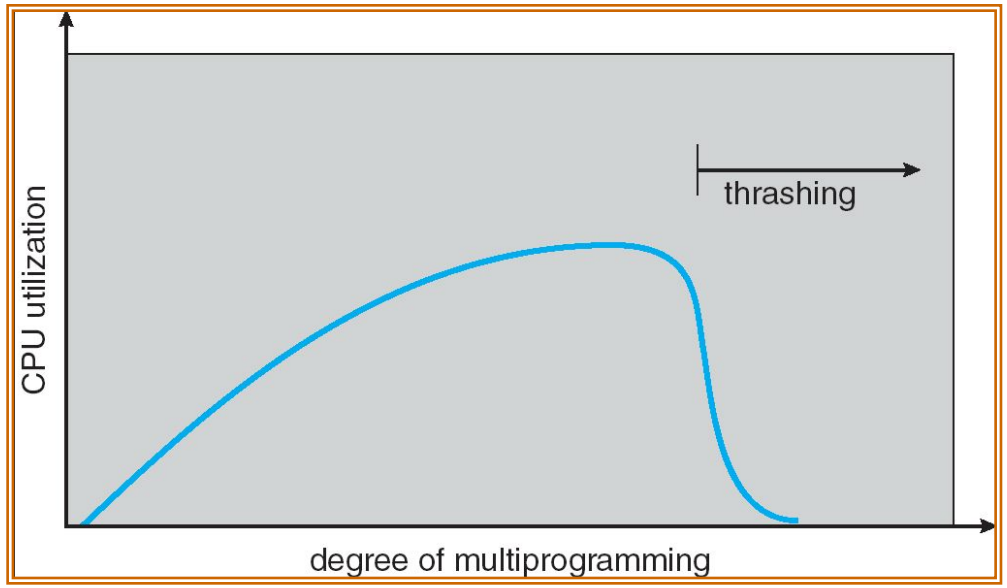
- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Question: What if we just don't have enough memory?

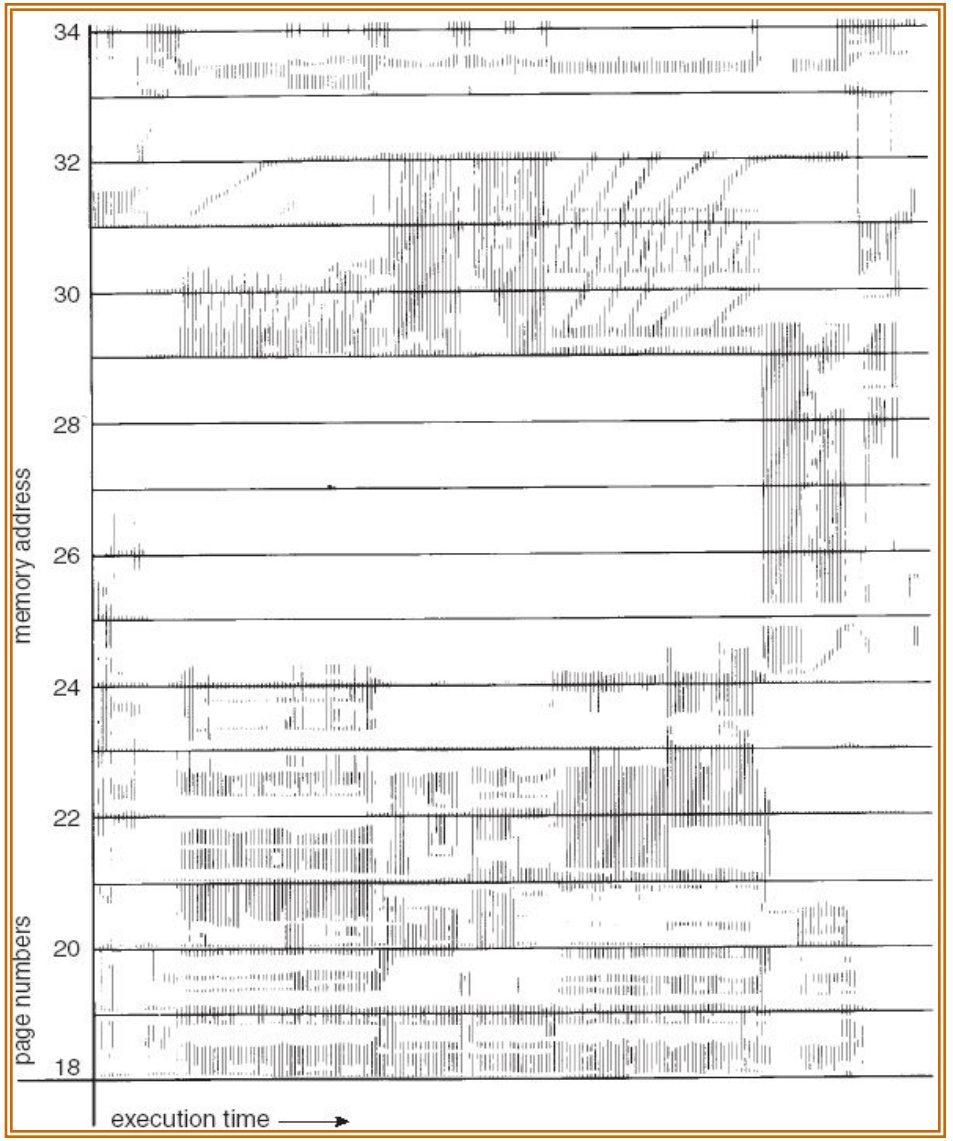
Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high.
  This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out with little or no actual progress
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

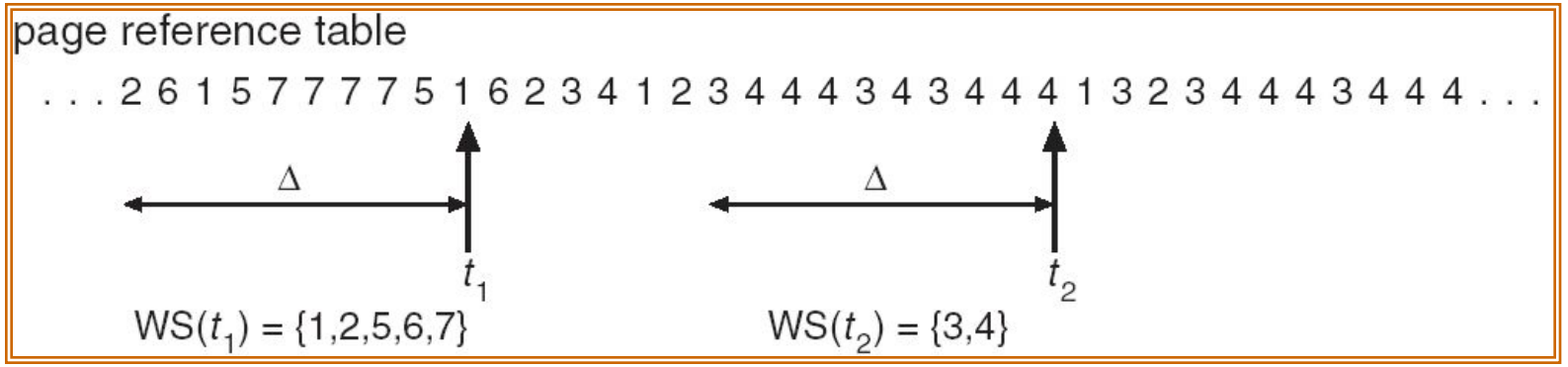


Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the "Working Set"
  - Working Set defines minimum number of pages for process to behave well
- Not enough memory for Working Set ⇒ Thrashing
  - Better to swap out process?



Working-Set Model Take 2



- Δ ≡ working-set window ≡ fixed number of page references
  - Example: 10,000 instructions
- WSi (working set of Process Pi) = total set of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program
- D = Σ|WSi| ≡ total demand frames
- if D > m ⇒ Thrashing
  - Policy: if D > m, then suspend/swap out processes
  - This can improve overall system behavior by a lot!

What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in
- Clustering:
  - On a page-fault, bring in multiple pages "around" the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set

Recall: Five Components of a Computer

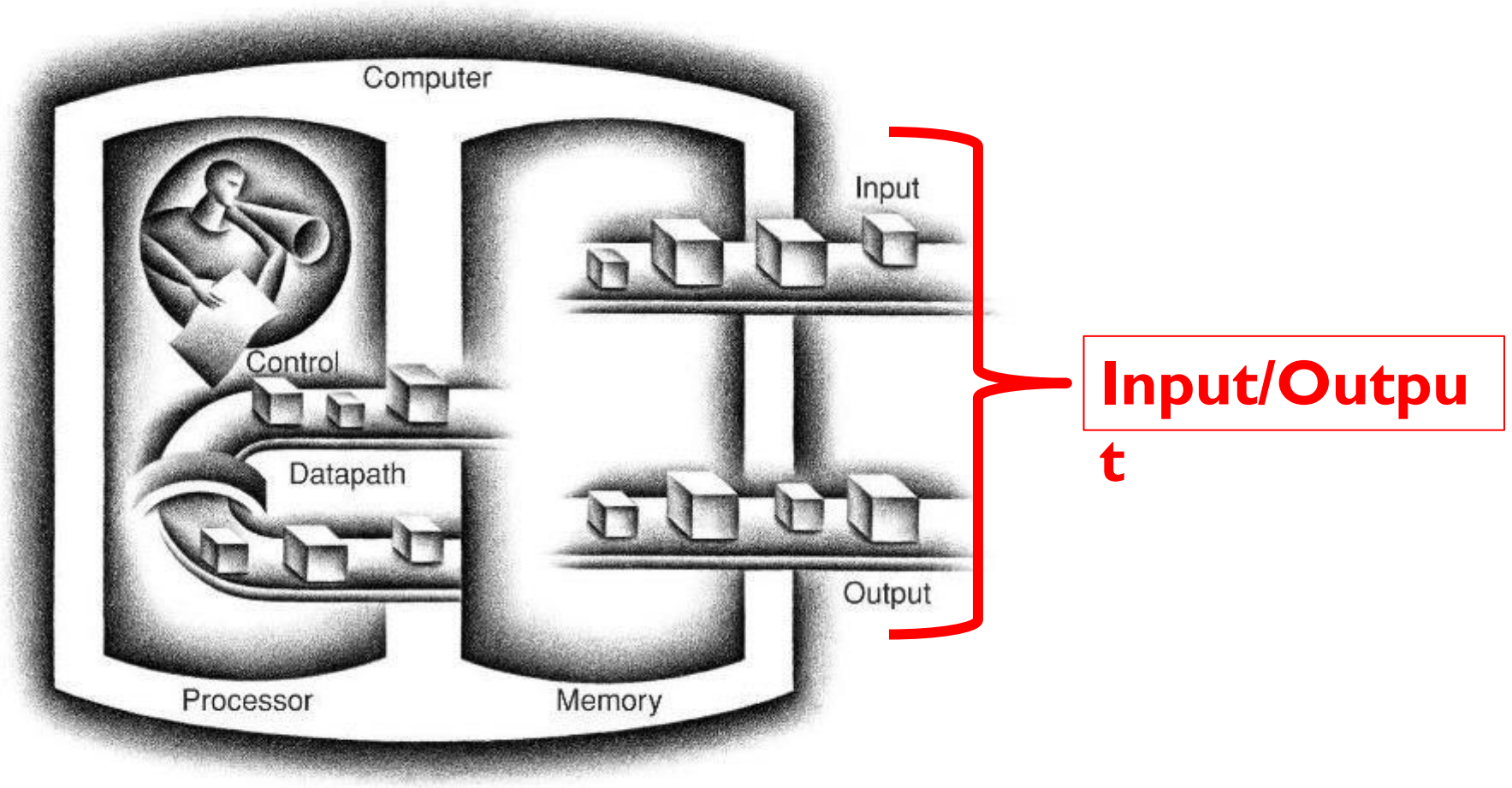
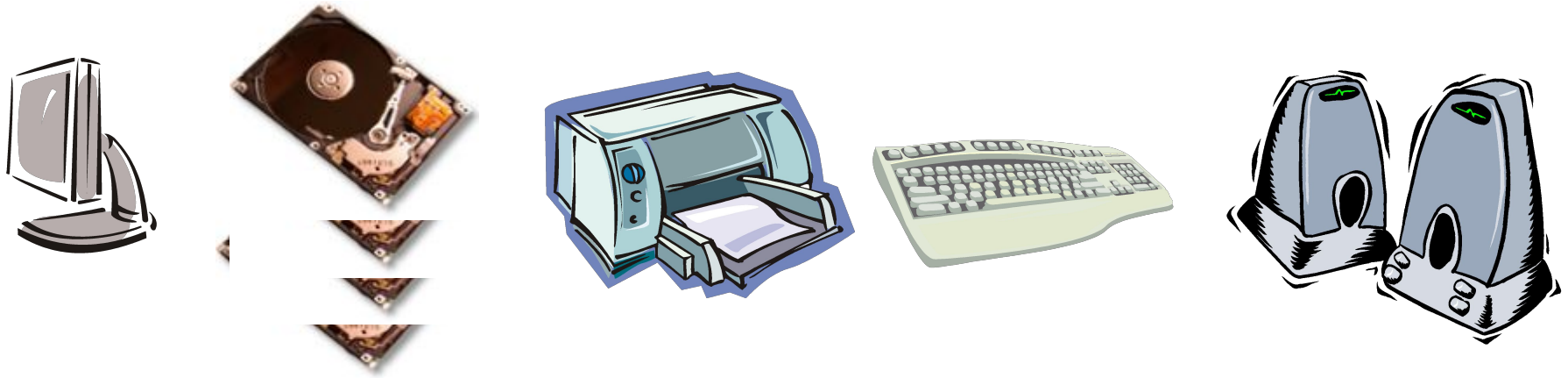


Diagram from "Computer Organization and Design" by Patterson and Hennessy

CPU: You need to get out more!

Input/output is the mechanism through which the computer communicates with the outside world



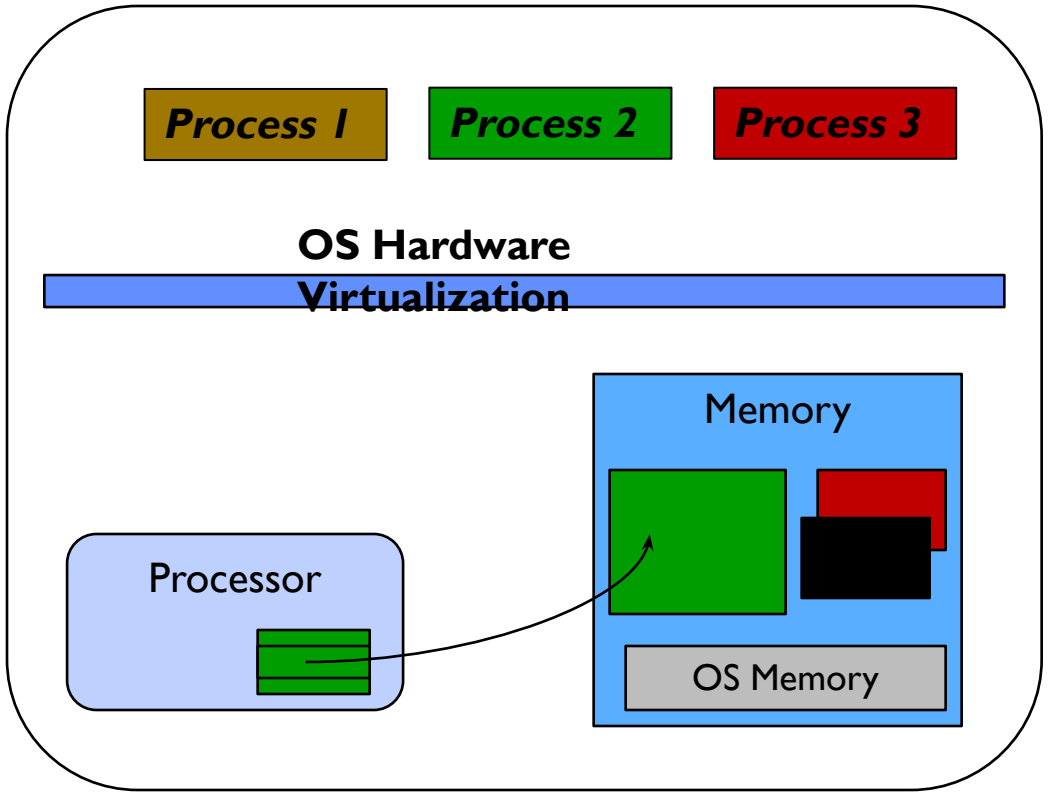
Want Standard Interfaces to Devices

- 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

- Character Devices: *e.g.* keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`

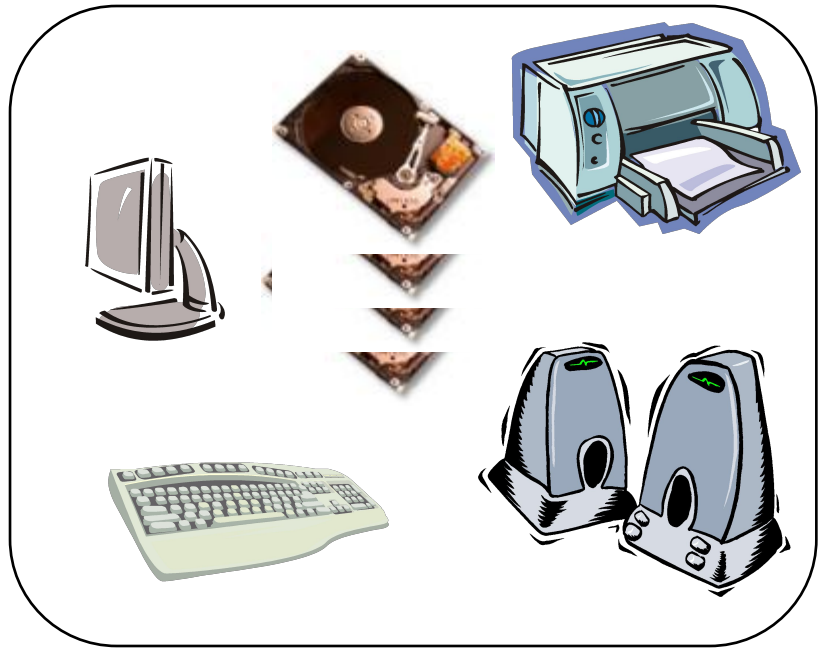
- Network Devices: *e.g.* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include socket interface
    - » Separates network protocol from network operation
    - » Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

IO Subsystem: Abstraction, abstraction, abstraction



Virtual Machine Abstraction

IO Layer



IO Devices

IO Subsystem: Abstraction, abstraction, abstraction

- This code

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

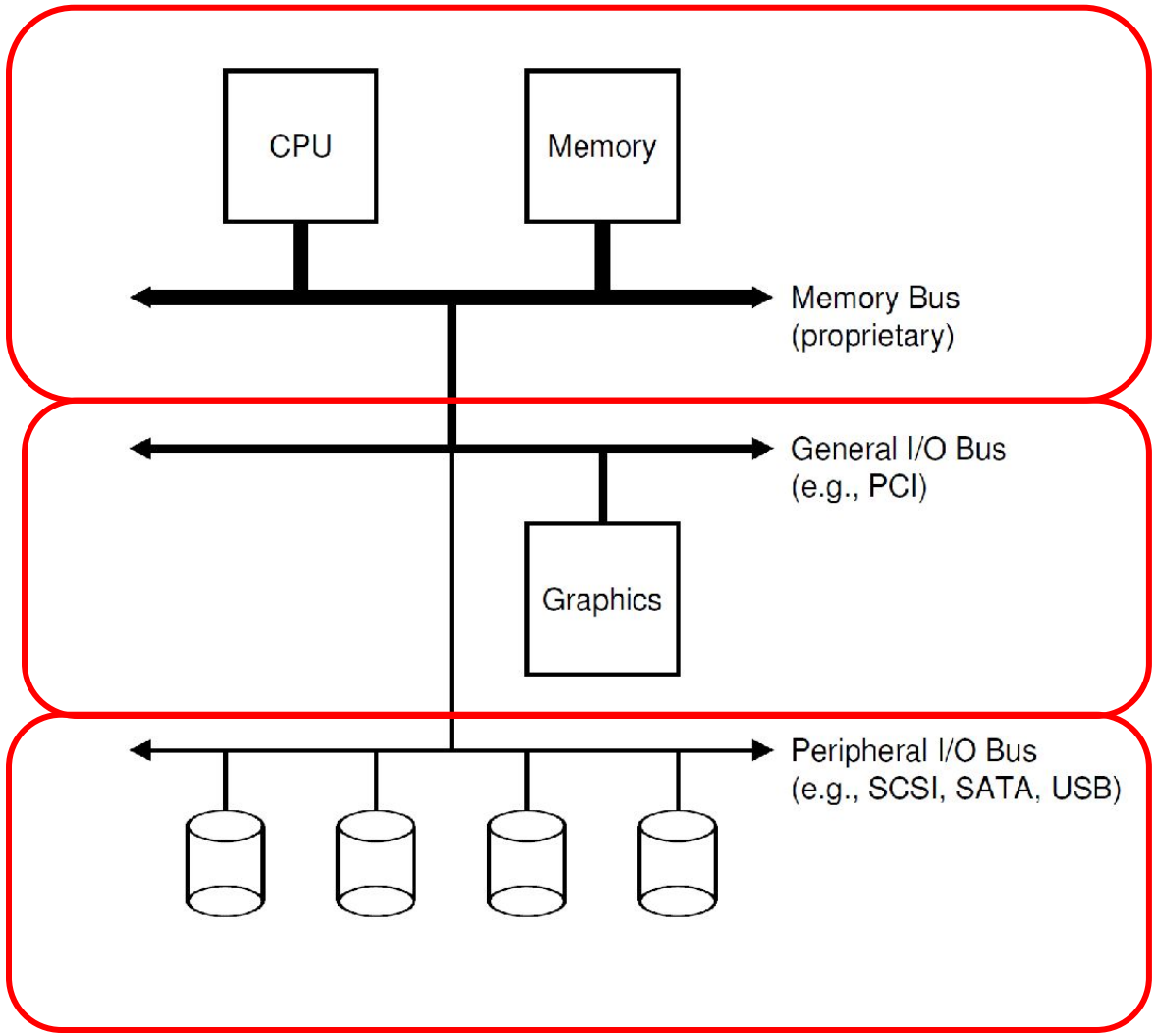
  - Why? Because code that controls devices ("device driver") implements standard interface

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

Requirements of I/O layer

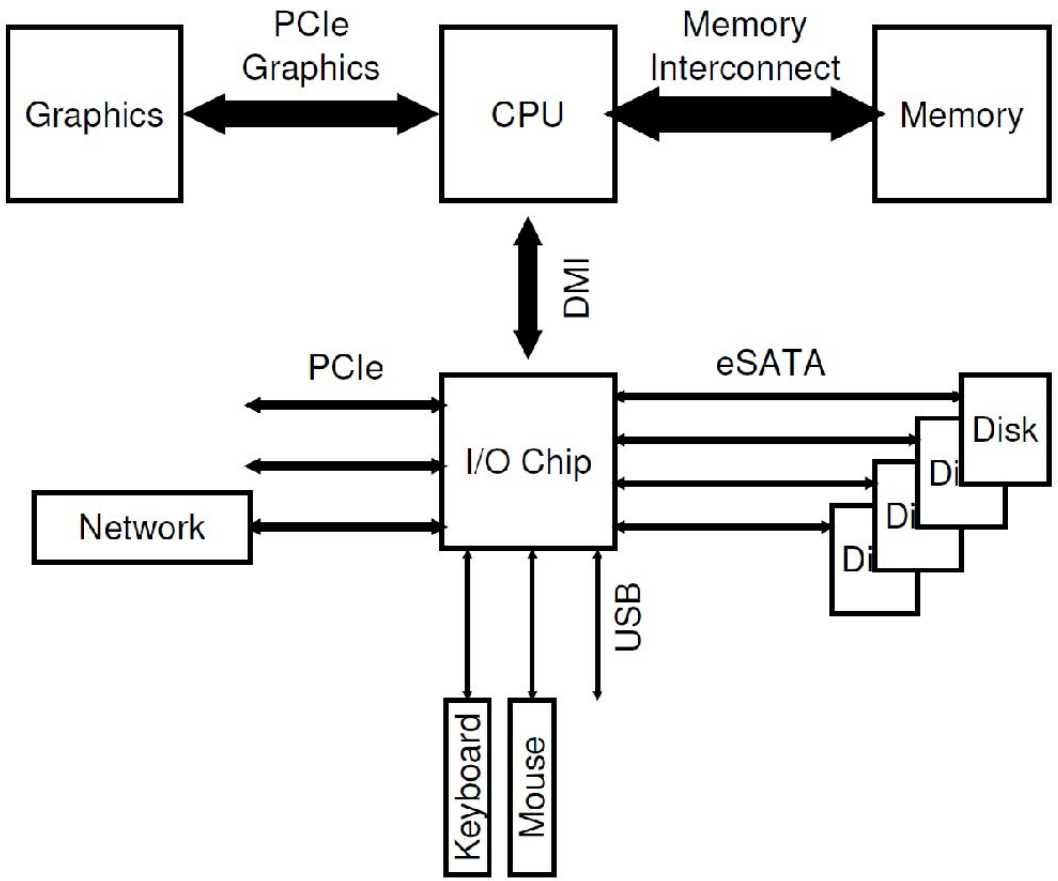
- But… thousands of devices, each slightly different
  - OS: How can we **standardize** the interfaces to these devices?
- Devices unreliable: media failures and transmission errors
  - OS: How can we make them **reliable**???
- Devices unpredictable and/or slow
  - OS: How can we **manage** them if we don't know what they will do or how they will perform?

Simplified IO architecture

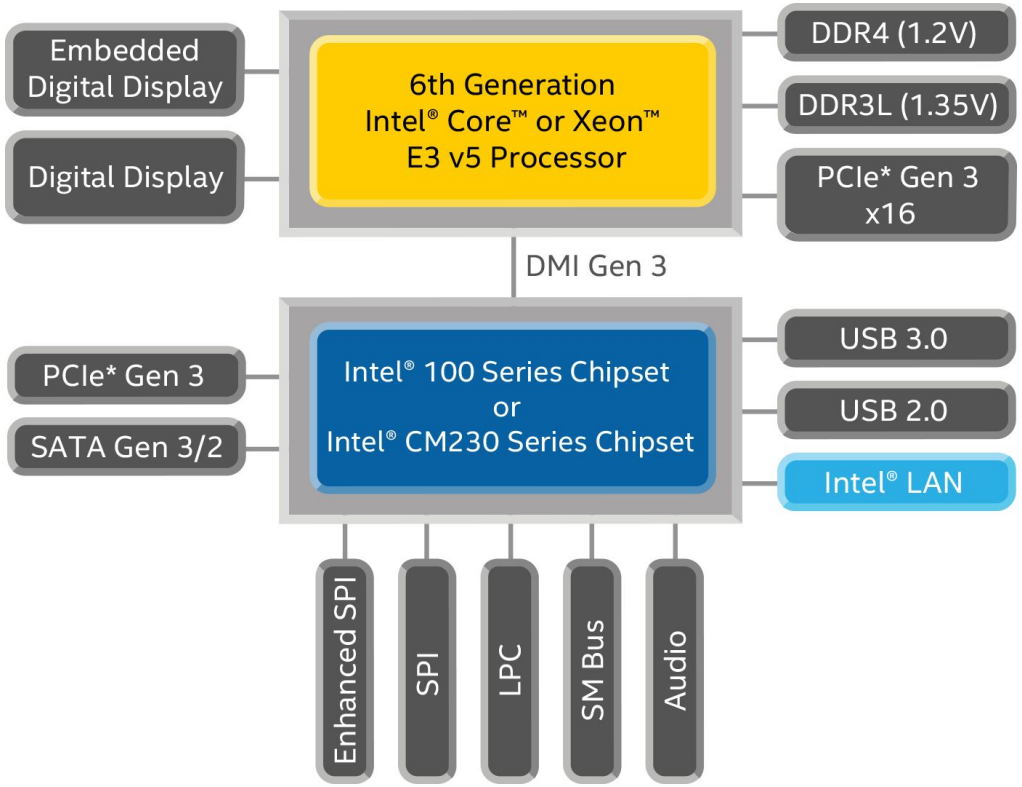


Follows a hierarchical structure because of cost: the faster the bus, the more expensive it is

Intel's Z270 Chipset



Sky Lake I/O: PCH



Sky Lake
System Configuration

- Platform Controller Hub
  - Connected to processor with proprietary bus
    - » Direct Media Interface
- Types of I/O on PCH:
  - USB, Ethernet
  - Thunderbolt 3
  - Audio, BIOS support
  - More PCI Express (lower speed than on Processor)
  - SATA (for Disks)

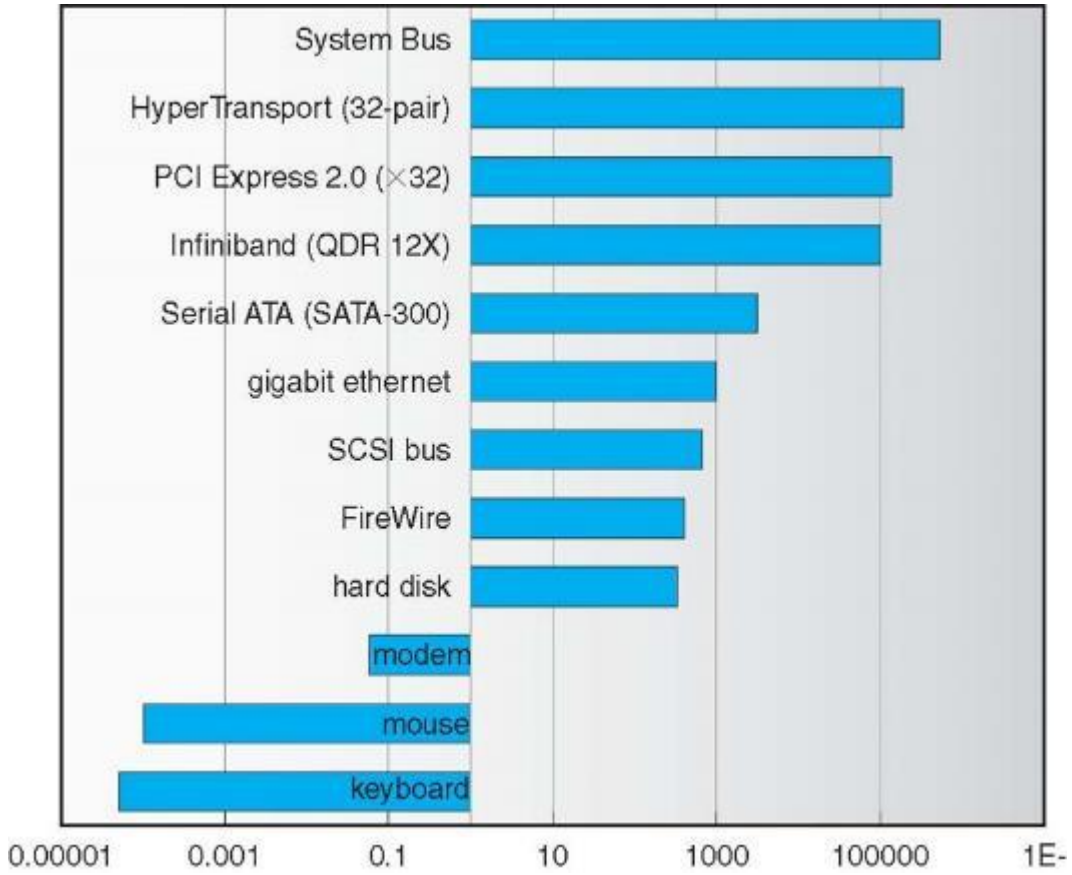
Recall: Range of Timescales

Jeff Dean: "Numbers Everyone Should Know"

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Example: Device Transfer Rates in Mb/s (Sun Enterprise 6000)

Device rates vary over 12 orders of magnitude!!!



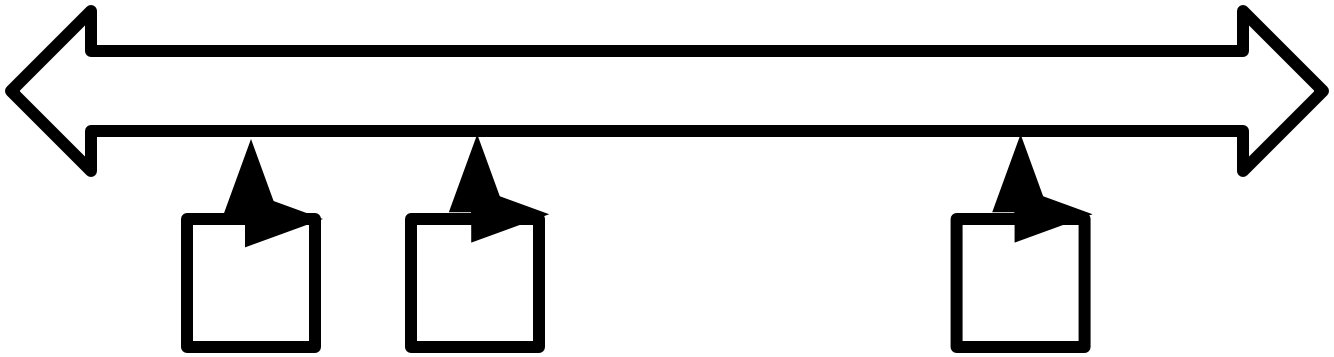
Administrivia

- Midterm 2 results
  - By next lecture (Tuesday, Nov 12)
- HW 4 due by Sunday, Nov 10
- Project 2 code and final report due by Wednesday, Nov 13

Two questions

- What is a bus?
- How does the processor talk to the devices?

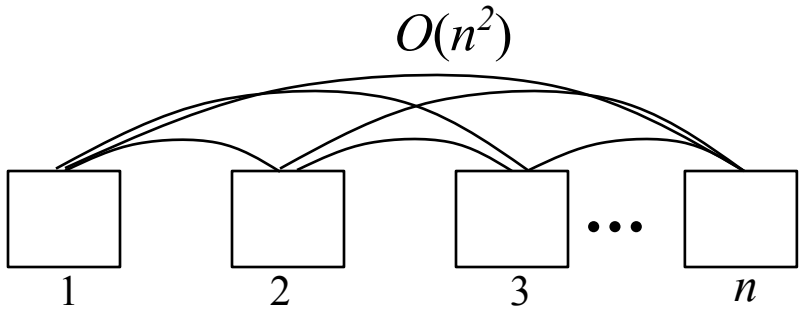
What's a bus?

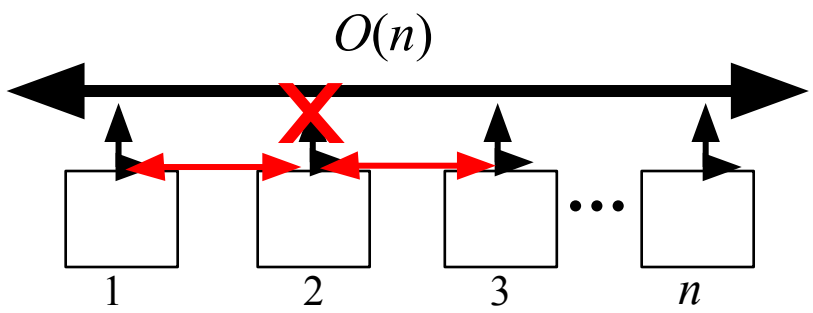


- Common set of wires for communication among hardware devices plus protocols for carrying out data transfer transactions
- Split into three parts: data bus, address bus, and control bus
- Protocol: initiator requests access, arbitration to grant, identification of recipient, handshake to convey address, length, data

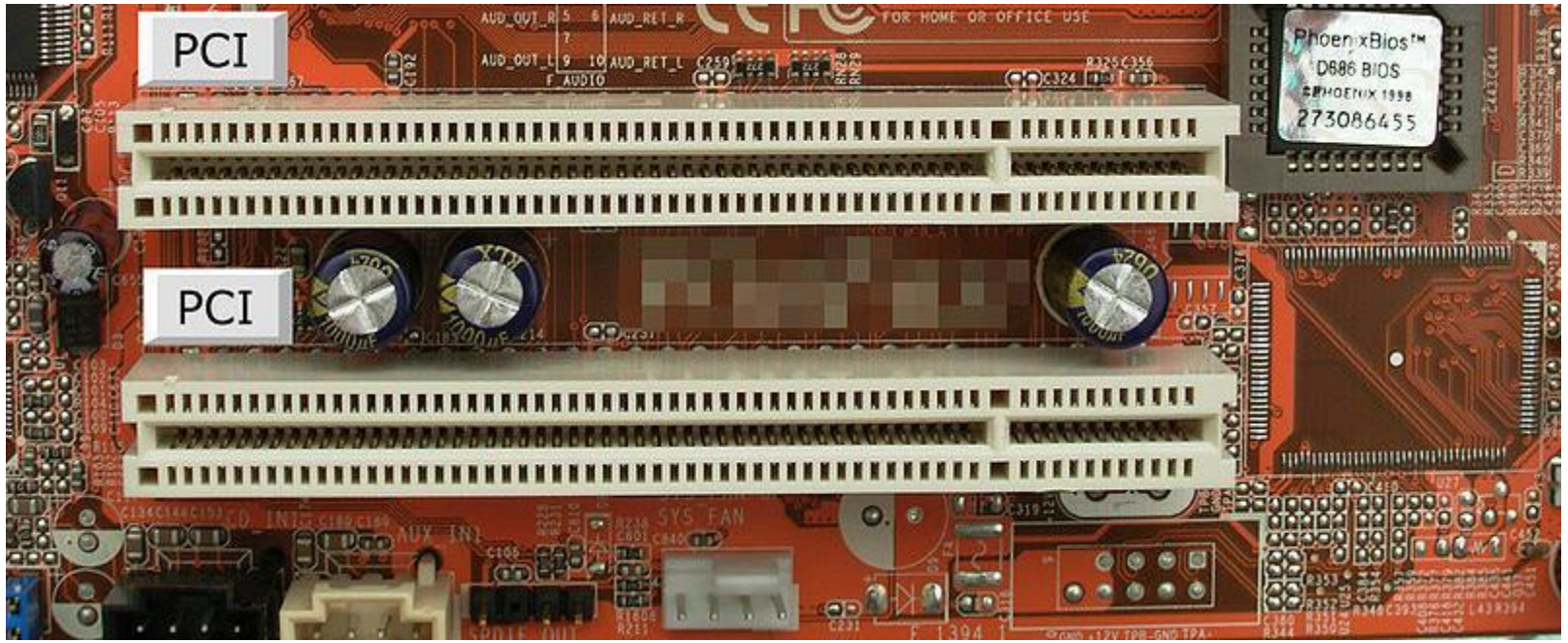
Why a Bus?

- Buses let us connect n devices over a single set of wires, connections, and protocols
  - $O(n^2)$ relationships with one set of wires (!)
- Downside: Only one transaction at a time
  - The rest must wait
  - "Arbitration" aspect of bus protocol ensures the rest wait





PCI Bus Evolution



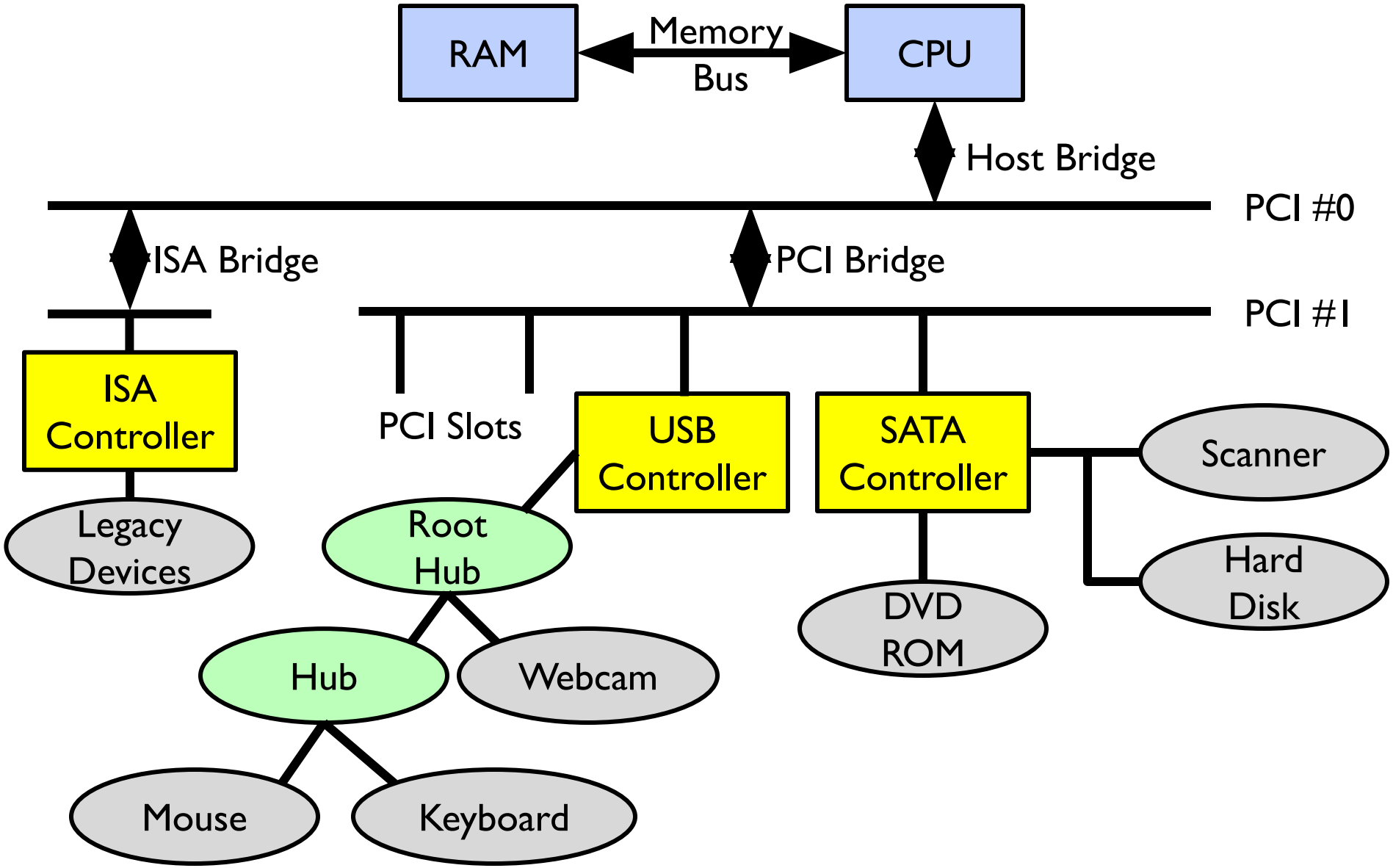
- PCI started life out as a parallel bus
- But a parallel bus has many limitations
  - Multiplexing address/data for many requests
  - Slowest devices must be able to tell what's happening (e.g., for arbitration)
  - **Bus speed is set to that of the slowest device**

PCI Express (PCIe) "Bus"

- No longer a parallel bus
- Really a **collection of fast serial channels** or "lanes"
- Devices can use as many serial channels as they need to achieve a desired bandwidth
  - 1X, 2X, 4X, 8X, 16X
- Slow devices don't have to share with fast ones

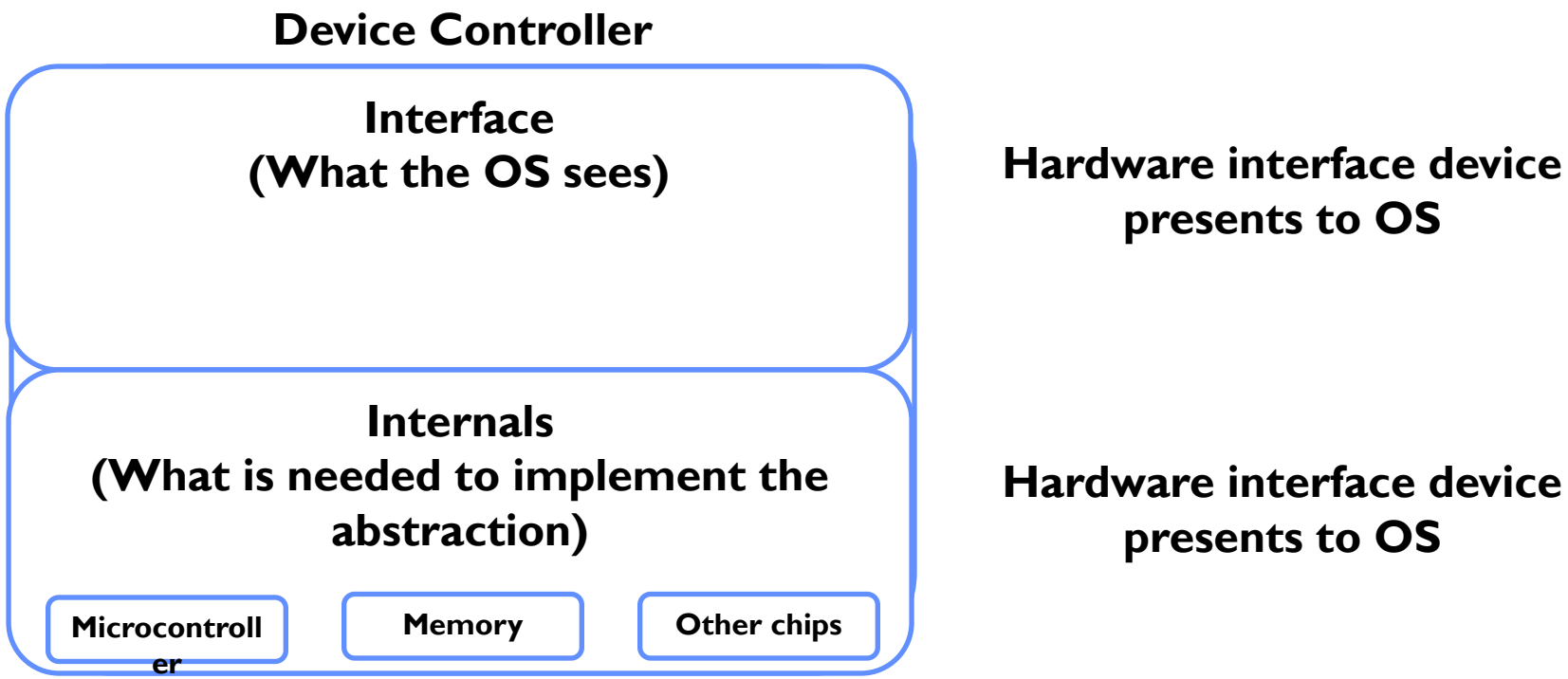
- One of the successes of device abstraction in Linux was the ability to migrate from PCI to PCI Express
  - The physical interconnect changed completely, but the old API still worked

Example: PCI Architecture

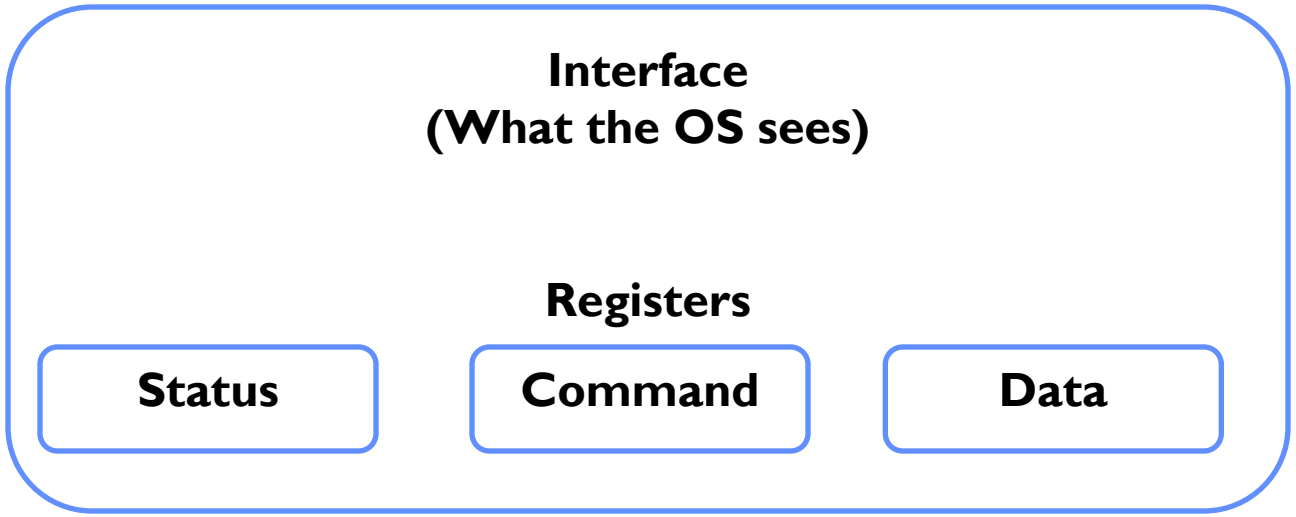


How does the processor talk to devices?

- Remember, it's all about abstractions!



How does the processor talk to devices?



Port-Mapped I/O:
Privileged 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

Port-Mapped I/O in Pintos Speaker Driver

Pintos: devices/speaker.c

```
13  /* Sets the PC speaker to emit a tone at the given FREQUENCY, in
14     Hz. */
15  void
16  speaker_on (int frequency)
17  {
18    if (frequency >= 20 && frequency <= 20000)
19      {
20        /* Set the timer channel that's connected to the speaker to
21           output a square wave at the given FREQUENCY, then
22           connect the timer channel output to the speaker. */
23        enum intr_level old_level = intr_disable ();
24        pit_configure_channel (2, 3, frequency);
25        outb (SPEAKER_PORT_GATE, inb (SPEAKER_PORT_GATE) | SPEAKER_GATE_ENABLE);
26        intr_set_level (old_level);
27      }
28    else
29      {
30        /* FREQUENCY is outside the range of normal human hearing.
31           Just turn off the speaker. */
32        speaker_off ();
33      }
34  }
35
36  /* Turn off the PC speaker, by disconnecting the timer channel's
37     output from the speaker. */
38  void
39  speaker_off (void)
40  {
41    enum intr_level old_level = intr_disable ();
42    outb (SPEAKER_PORT_GATE, inb (SPEAKER_PORT_GATE) & ~SPEAKER_GATE_ENABLE);
43    intr_set_level (old_level);
44  }
```

Pintos: threads/io.h

```
7   /* Reads and returns a byte from PORT. */
8   static inline uint8_t
9   inb (uint16_t port)
10  {
11    /* See [IA32-v2a] "IN". */
12    uint8_t data;
13    asm volatile ("inb %w1, %b0" : "=a" (data) : "Nd" (port));
14    return data;
15  }
```

```
64  /* Writes byte DATA to PORT. */
65  static inline void
66  outb (uint16_t port, uint8_t data)
67  {
68    /* See [IA32-v2b] "OUT". */
69    asm volatile ("outb %b0, %w1" : : "a" (data), "Nd" (port));
70  }
```

A simple protocol

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

Protocol does a lot of polling!

How can we lower this overhead?

CPU is responsible for moving data

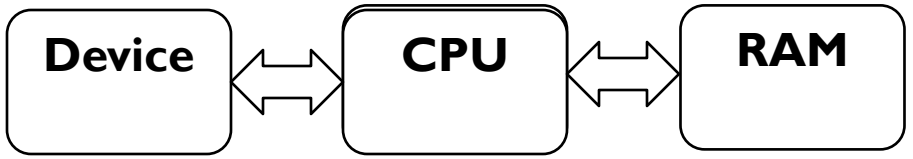
Polling vs Interrupt-driven IO

- Use hardware interrupts:
  - Allows CPU to process another task. Will get notified when task is done
  - Interrupt handler will read data & error code

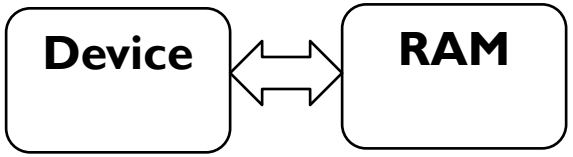
- Is it always better to use interrupts?

From programmed IO to direct memory access

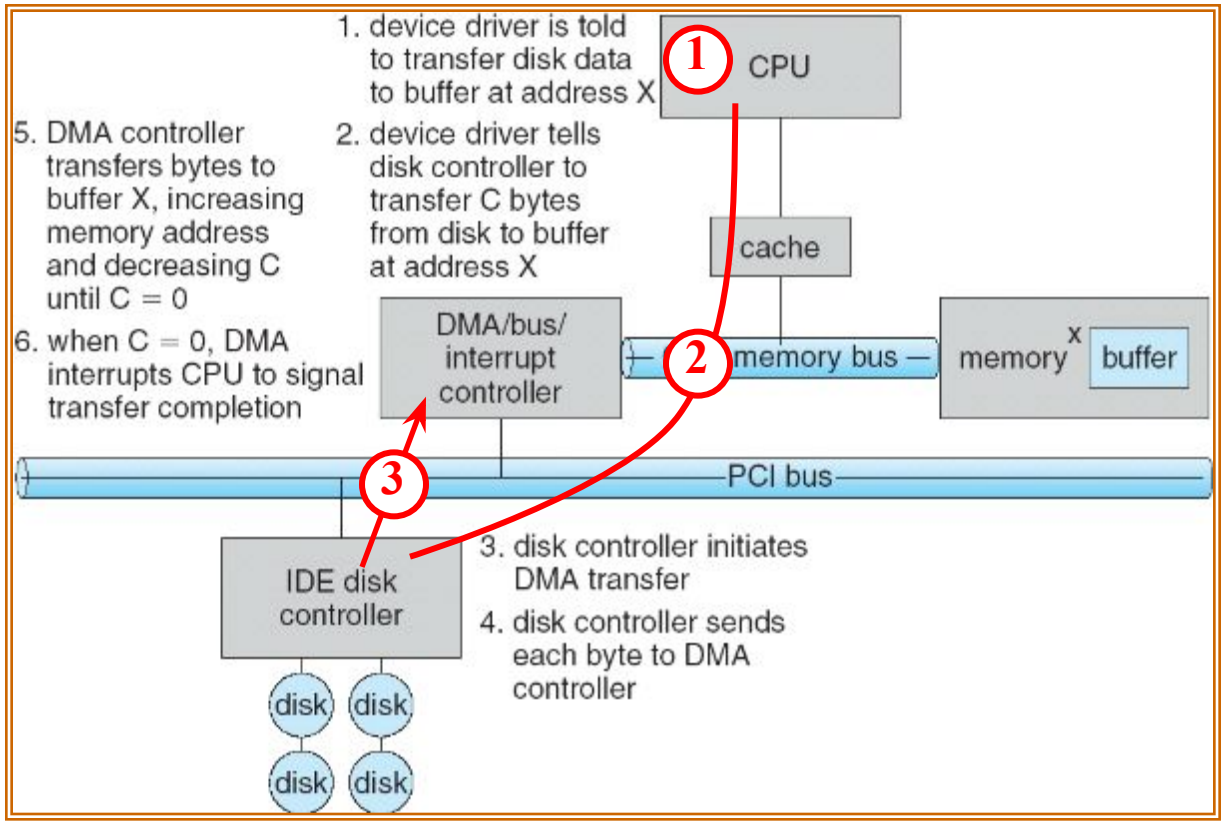
- With programmed IO (simple protocol):
  - CPU issues read request
  - Device interrupts CPU with data
  - CPU writes data to memory
  - Pros: simple hardware. Cons: Poor CPU is always busy!



- With direct-memory-access (DMA):
  - CPU sets up DMA request
    - » Gives controller access to memory bus
  - Device puts data on bus & RAM accepts it
  - Device interrupts CPU when done



DMA in more detail



How can the OS handle ~~one~~ all devices

- How do we fit devices with specific interfaces into OS, which should remain general?
  - Build a "device neutral" OS and hide details of devices from most of OS

- Abstraction to the rescue!
  - Device Drivers encapsulate all specifics of device interaction
  - Implement device neutral interfaces

Device Drivers

- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware and implements the OS abstraction
  - Supports a standard, internal interface
  - 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

- Your body is 90% water, the OS is 70% device-drivers

Putting it together: Life Cycle of An I/O Request

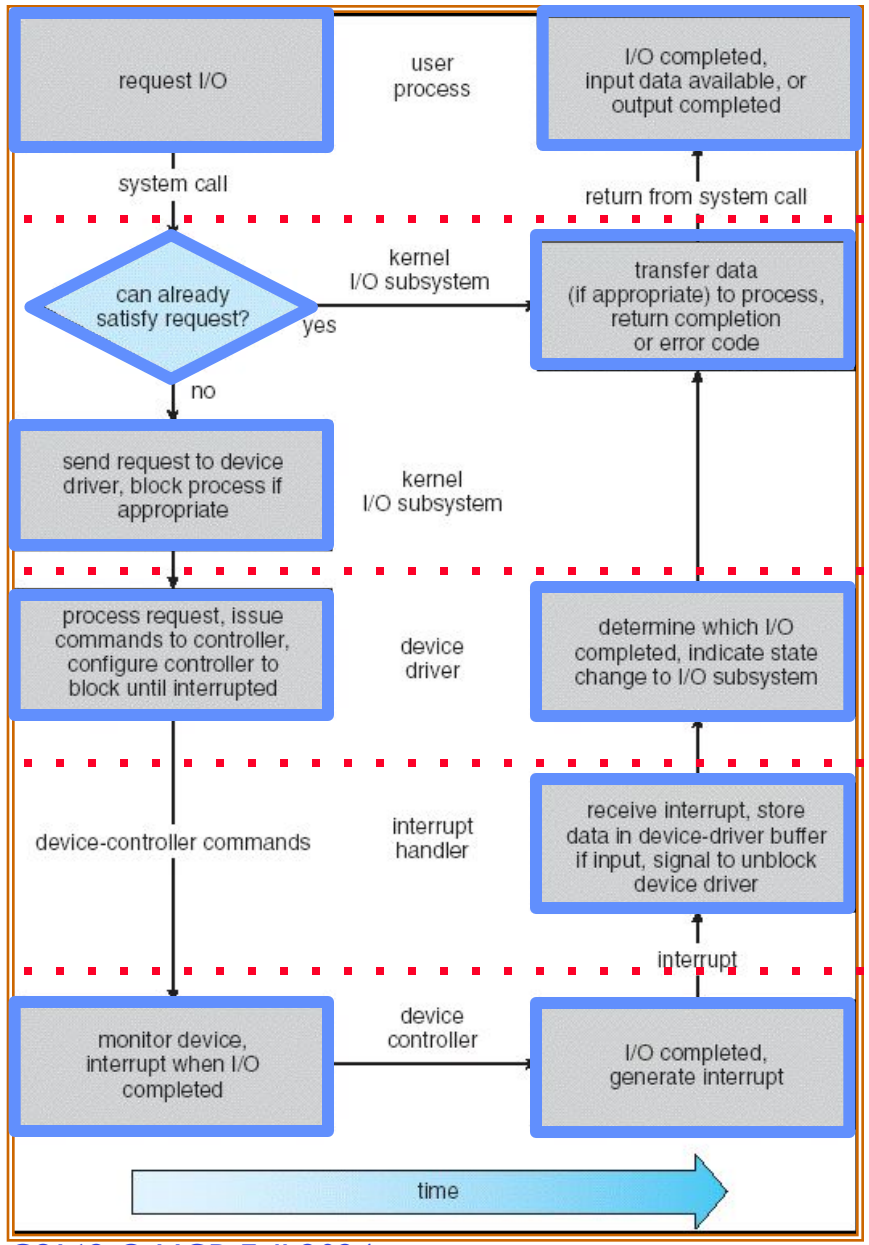
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware



Conclusion

- I/O Devices Types:
  - Many different speeds (0.1 bytes/sec to GBytes/sec)
  - Different Access Patterns:
    - » Block Devices, Character Devices, Network Devices
  - Different Access Timing:
    - » Blocking, Non-blocking, Asynchronous
- I/O Controllers: Hardware that controls actual device
  - Processor Accesses through I/O instructions, load/store to special physical memory
- Notification mechanisms
  - Interrupts
  - Polling: Report results through status register that processor looks at periodically
- Device drivers interface to I/O devices
  - Provide clean Read/Write interface to OS above
  - Manipulate devices through PIO, DMA & interrupt handling
  - Three types: block, character, and network

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