CS162
Operating Systems and Systems Programming
Lecture 17

General I/O, Storage Devices

Professor Natacha Crooks & Matei Zaharia

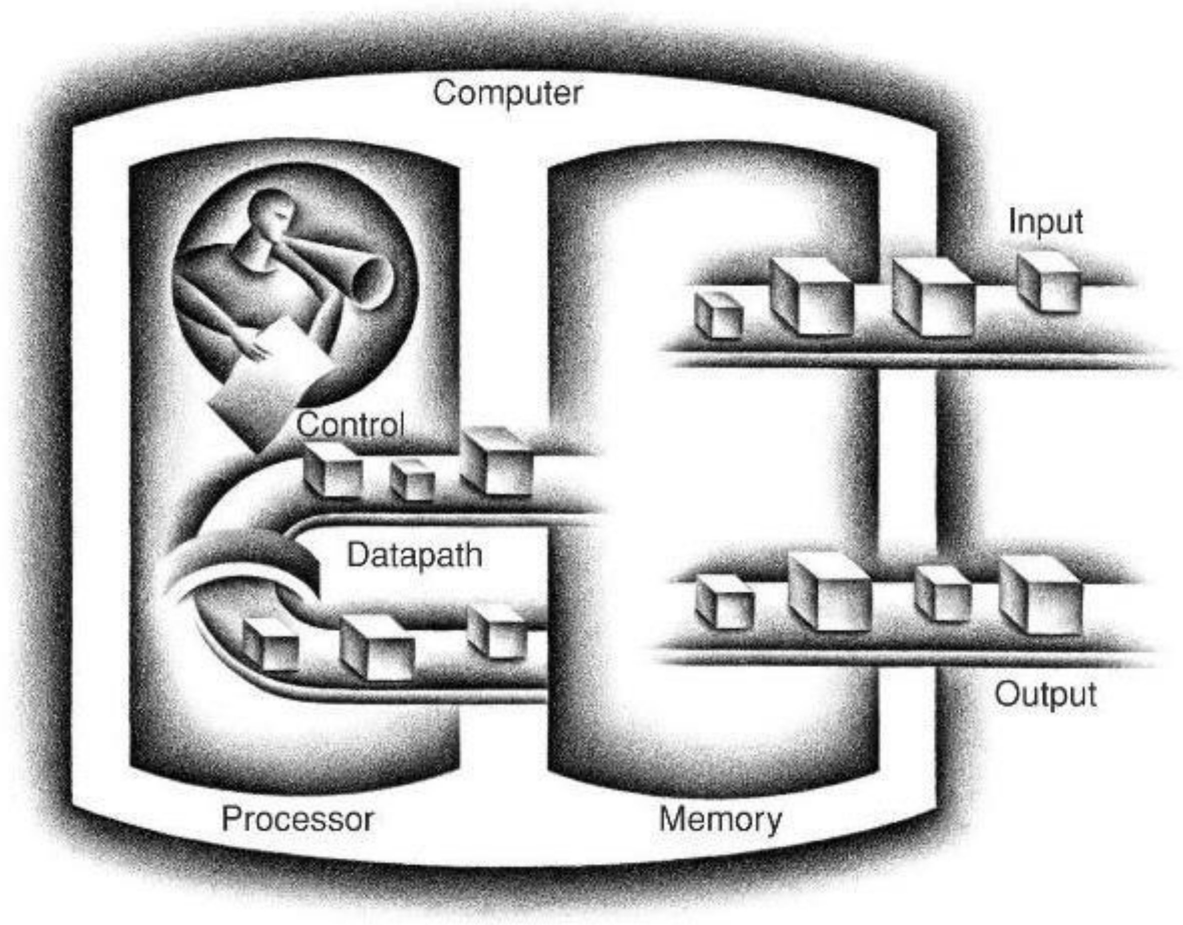
https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Course Map

- Introduction
- OS Concepts
- Concurrency
- Scheduling
- Memory Management
- Devices and file systems
- Reliability, networking and cloud

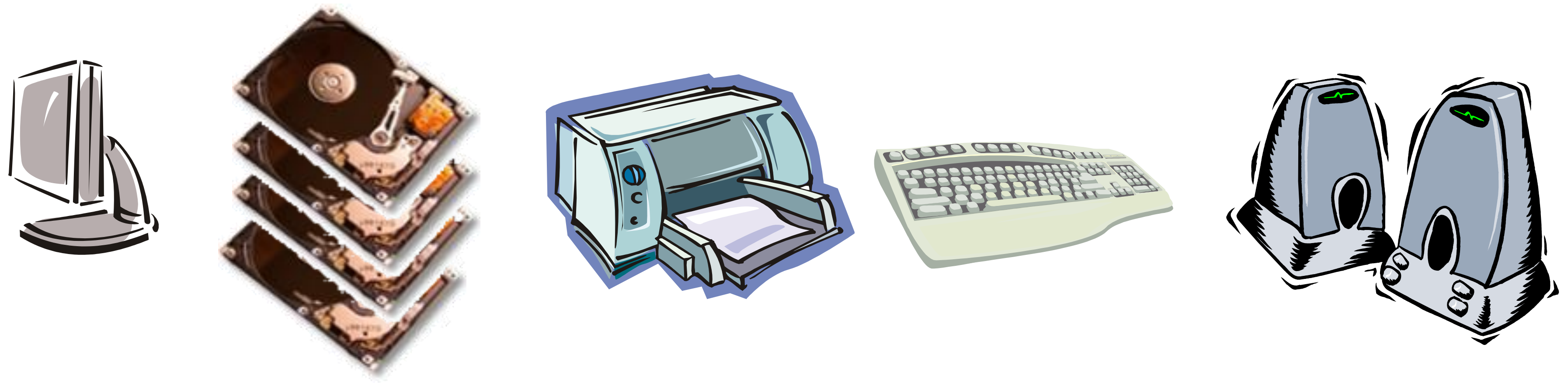
Recall: Five Components of a Computer



From "Computer Organization and Design" by Hennesy & Patterson

CPU: You need to get out more!

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



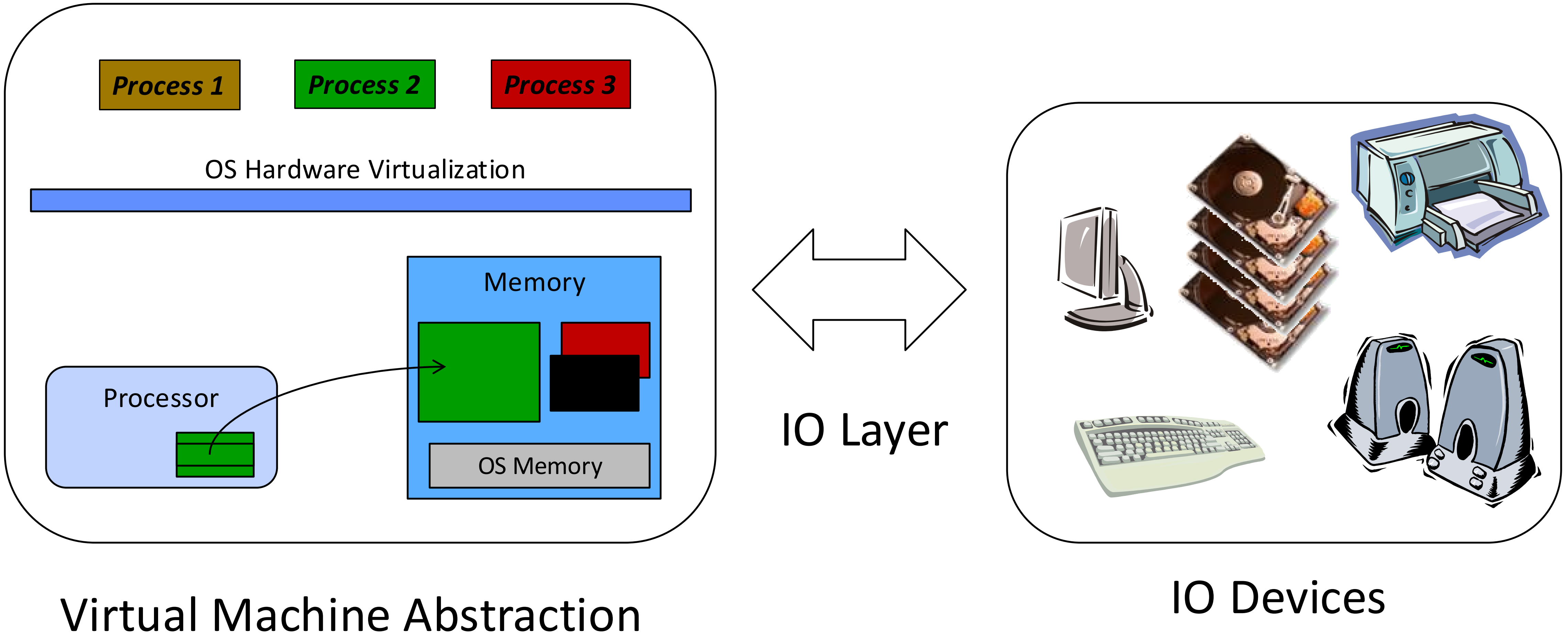
Want Standard Interfaces to Devices

- 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 polling and connection management functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

IO Subsystem: Abstraction, abstraction, abstraction



IO Subsystem: Abstraction, abstraction, abstraction

- This code works for pretty much any device

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

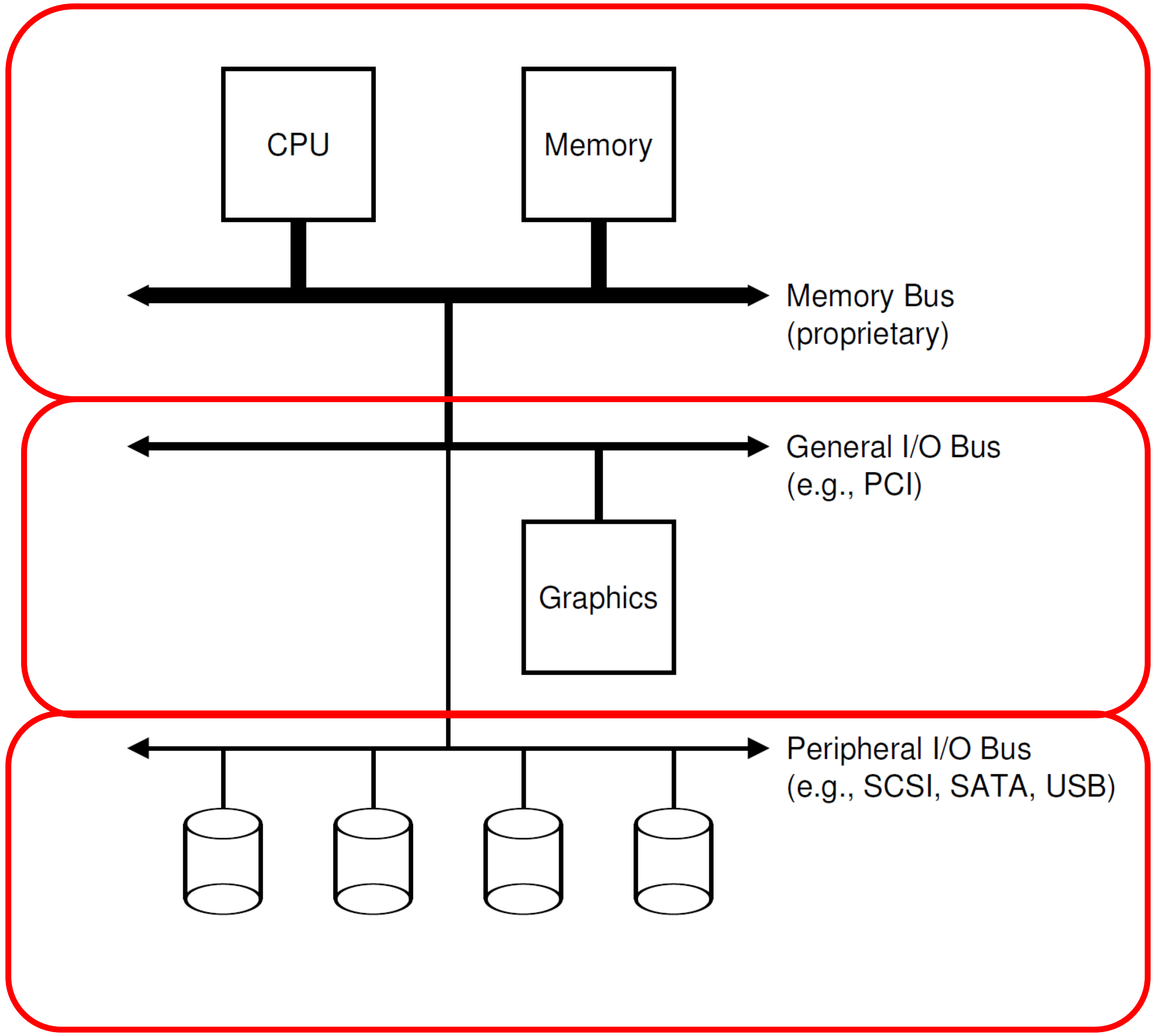
  – Why? Because code that controls devices (“device driver”) implements standard interface

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

Requirements of I/O layer

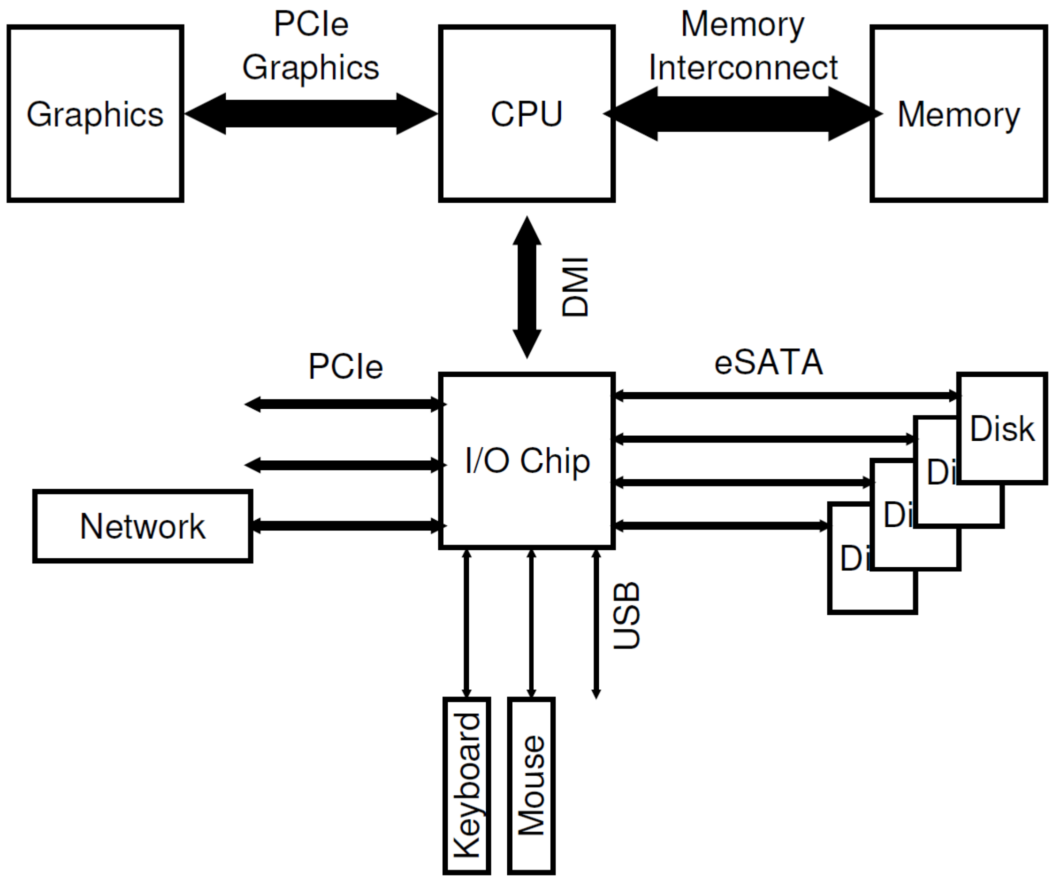
- But… thousands of devices, each slightly different
  - » OS: How can we **standardize** the interfaces to these devices?
- Devices unreliable: media failures and transmission errors
  - » OS: How can we make them **reliable**???
- Devices unpredictable and/or slow
  - » OS: How can we **manage** them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

Simplified IO architecture

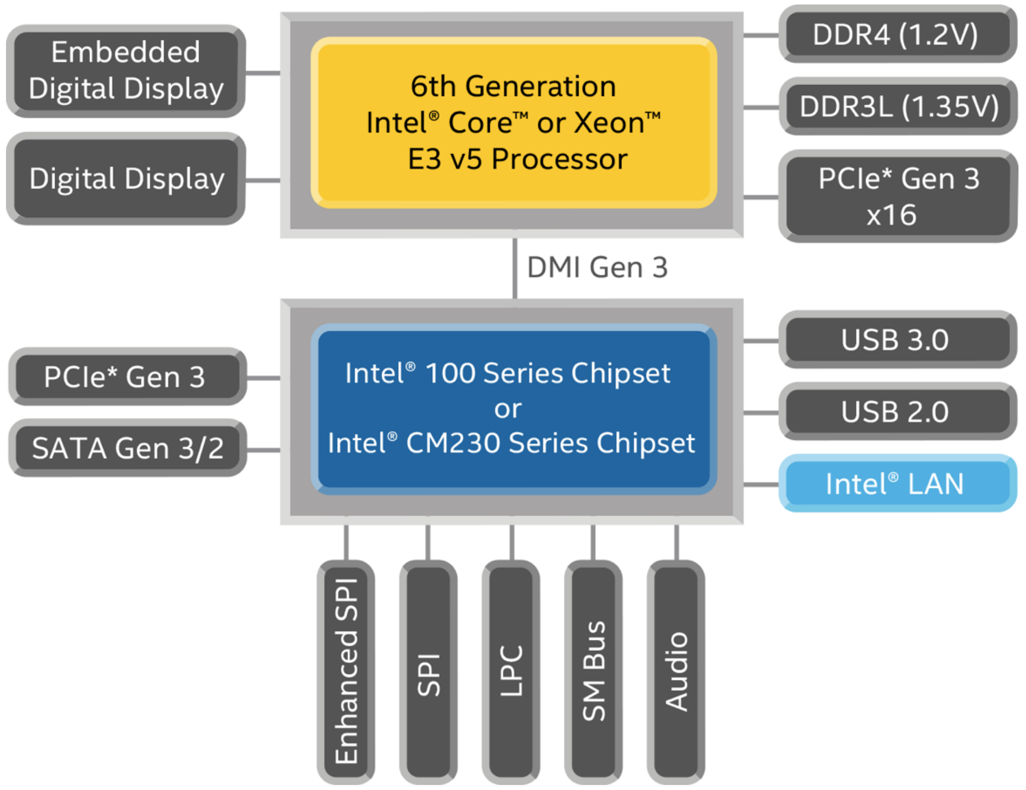

CPU
Memory
Memory Bus (proprietary)
General I/O Bus (e.g., PCI)
Graphics
Peripheral I/O Bus (e.g., SCSI, SATA, USB)

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)

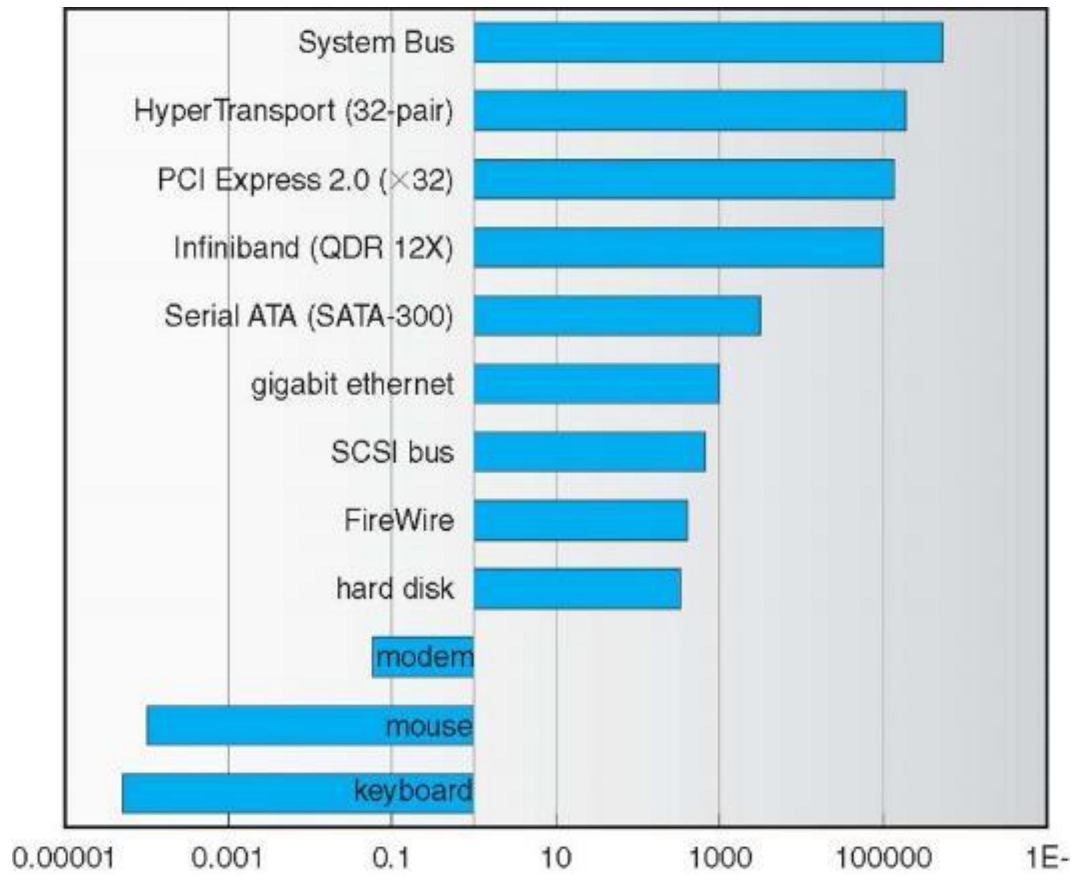
Recall: Range of Timescales

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)

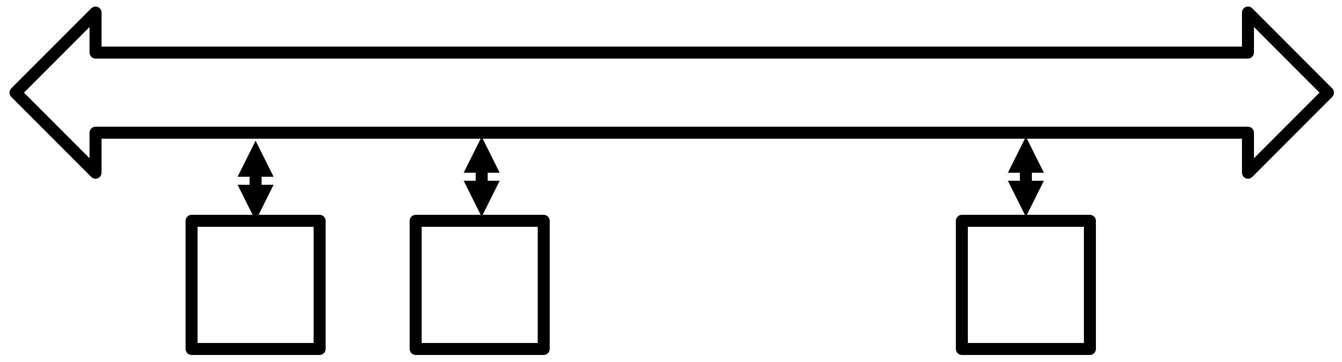
Device rates vary over 12 orders of magnitude!!!



Two questions

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

What’s a bus?

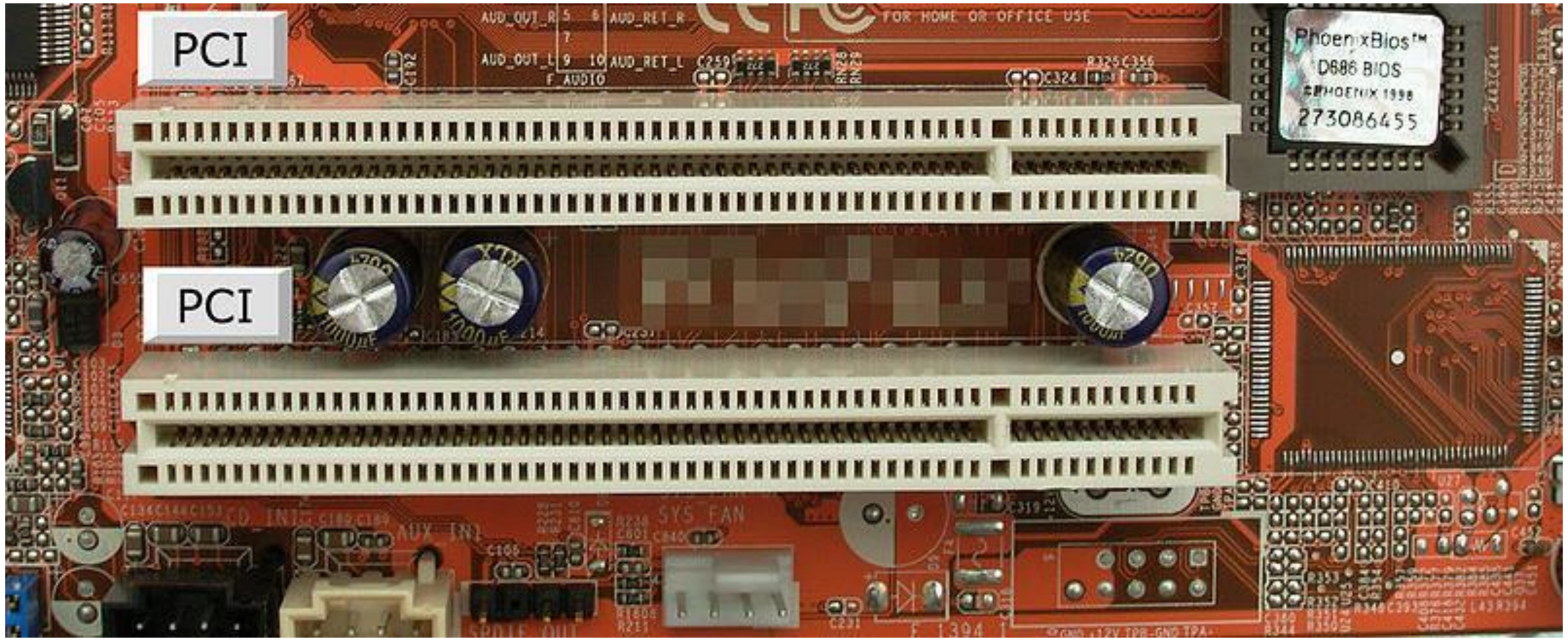


- Common set of wires for communication among multiple hardware devices plus protocols for carrying out data transfer transactions
- Split into three parts: control lines, address lines, and data lines
- Protocol: initiator requests access, arbitration to grant, identification of recipient, handshake to convey address, length, data
- High bandwidth close to processor, and slower but more flexible farther out

Why a Bus?

- Buses let us connect n devices over a single set of wires, connections, and protocols
  - $O(n^2)$ relationships with 1 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 (send bits on many wires in parallel)
- But a parallel bus has many limitations
  - Hard to keep all the wires sending/receiving in sync
  - 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”

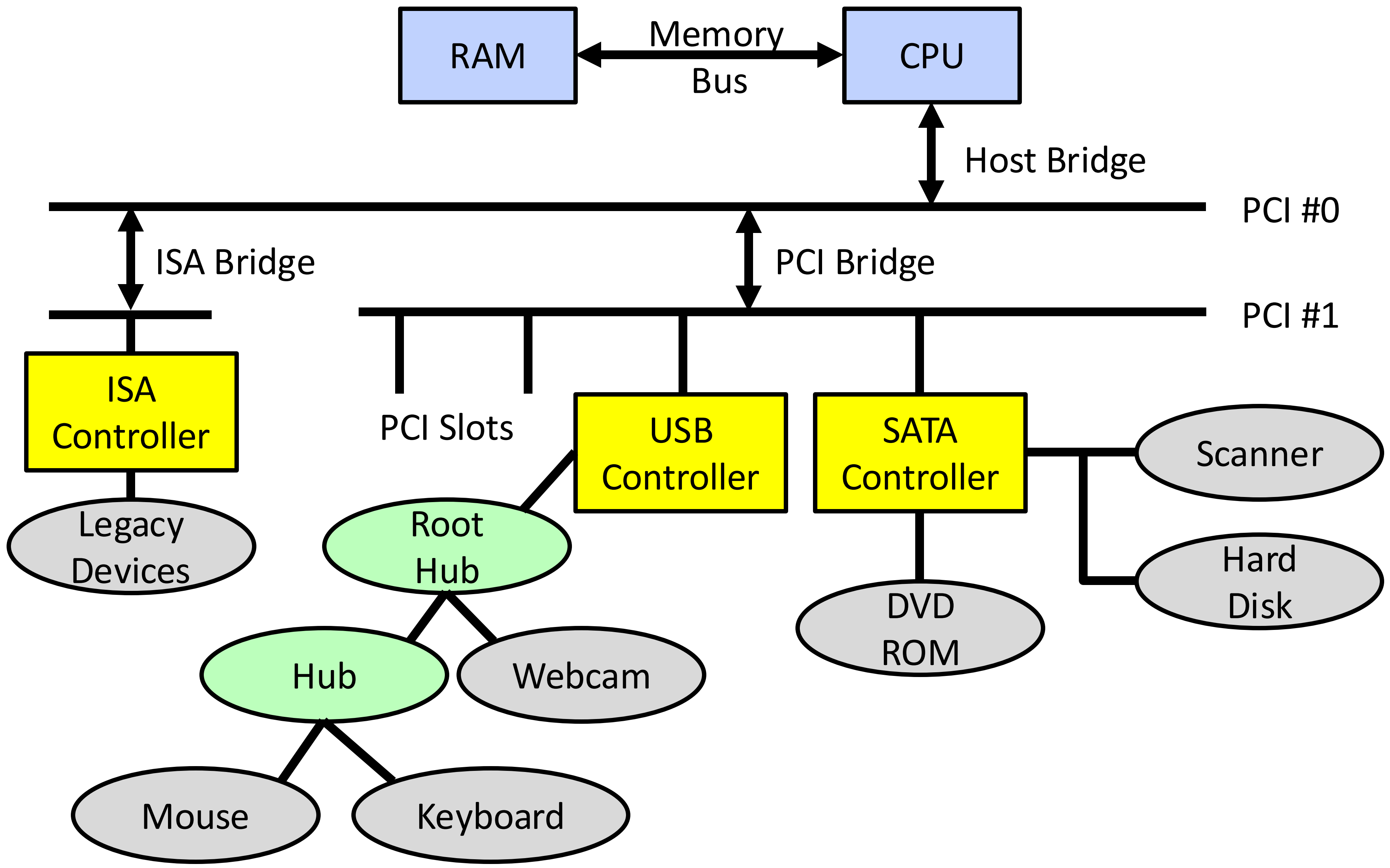
- No longer a parallel bus

- Really a **collection of fast serial channels** or “lanes”
- Devices can use as many as they need to achieve a desired bandwidth

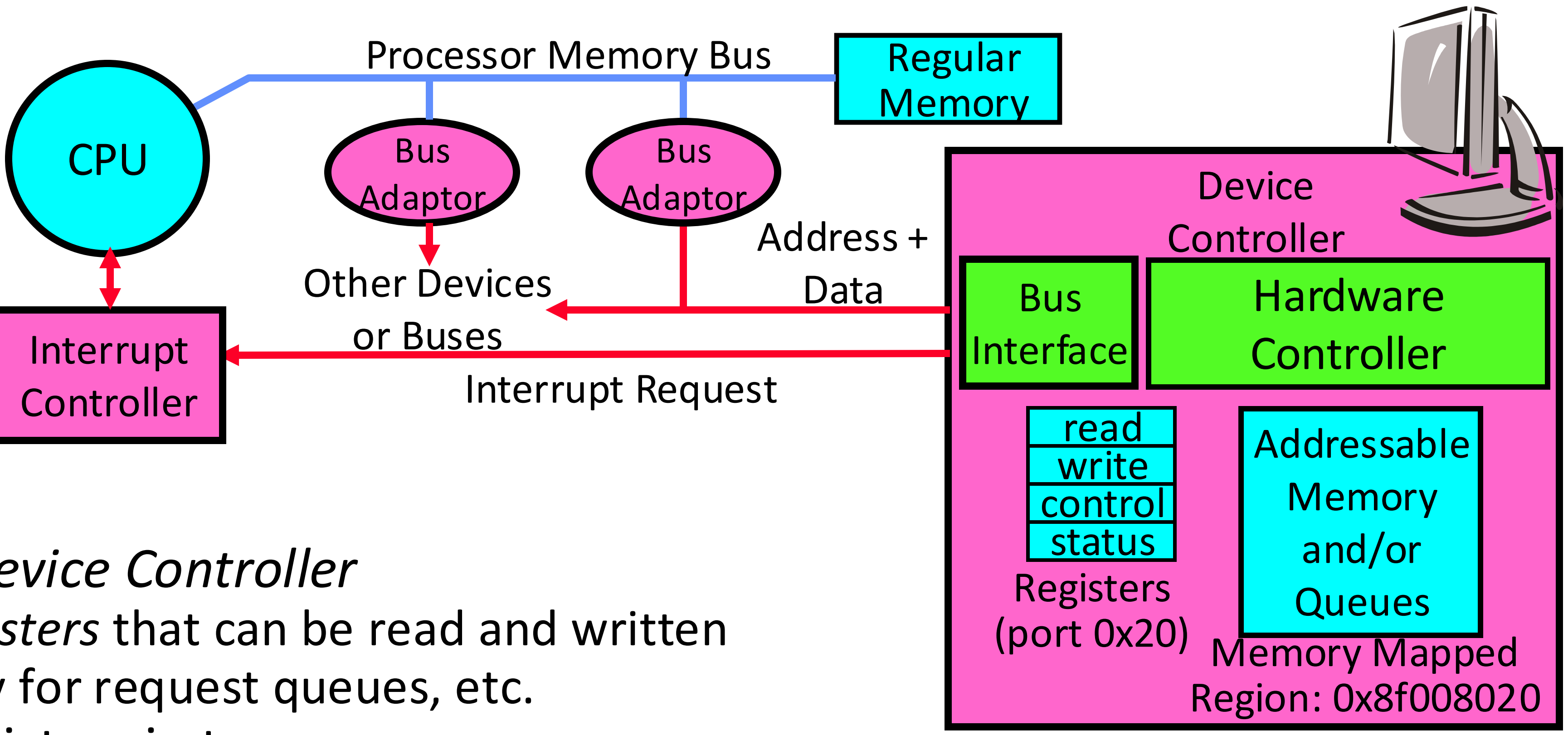
- Slow devices don’t have to share with fast ones

- 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



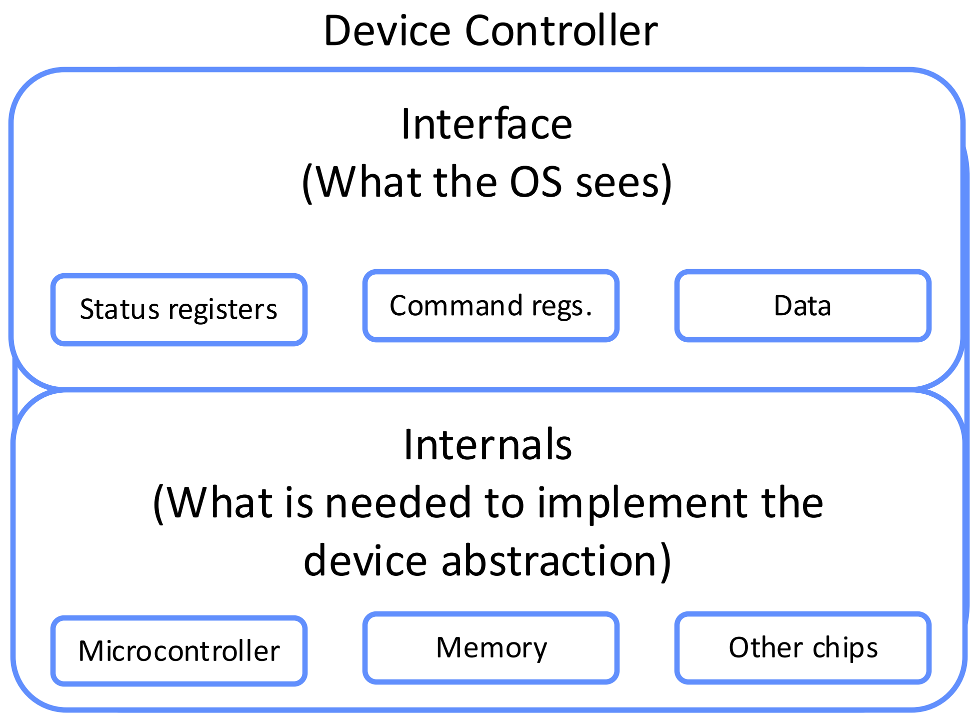
How does the Processor Talk to Devices?



- CPU interacts with a *Device Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues, etc.
- Processor accesses registers in two ways:
  - Port-Mapped I/O: 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

How does the processor talk to devices?

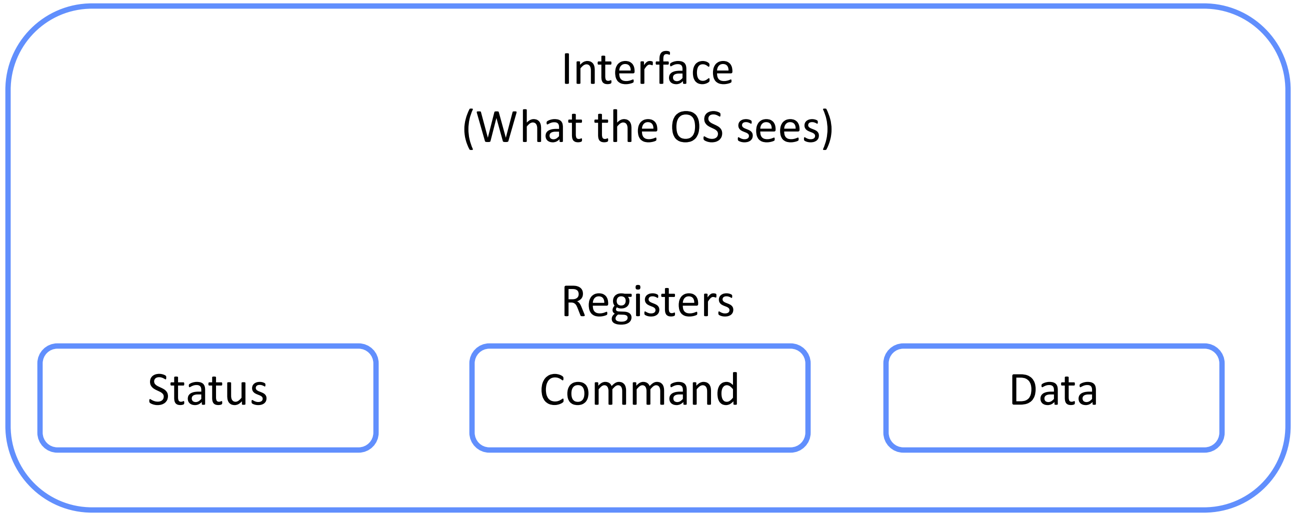
- Remember, it's all about abstractions!



Hardware interface device presents to OS

Device implementation

How does the processor talk to devices?



Port-Mapped I/O:
Privileged in/out instructions

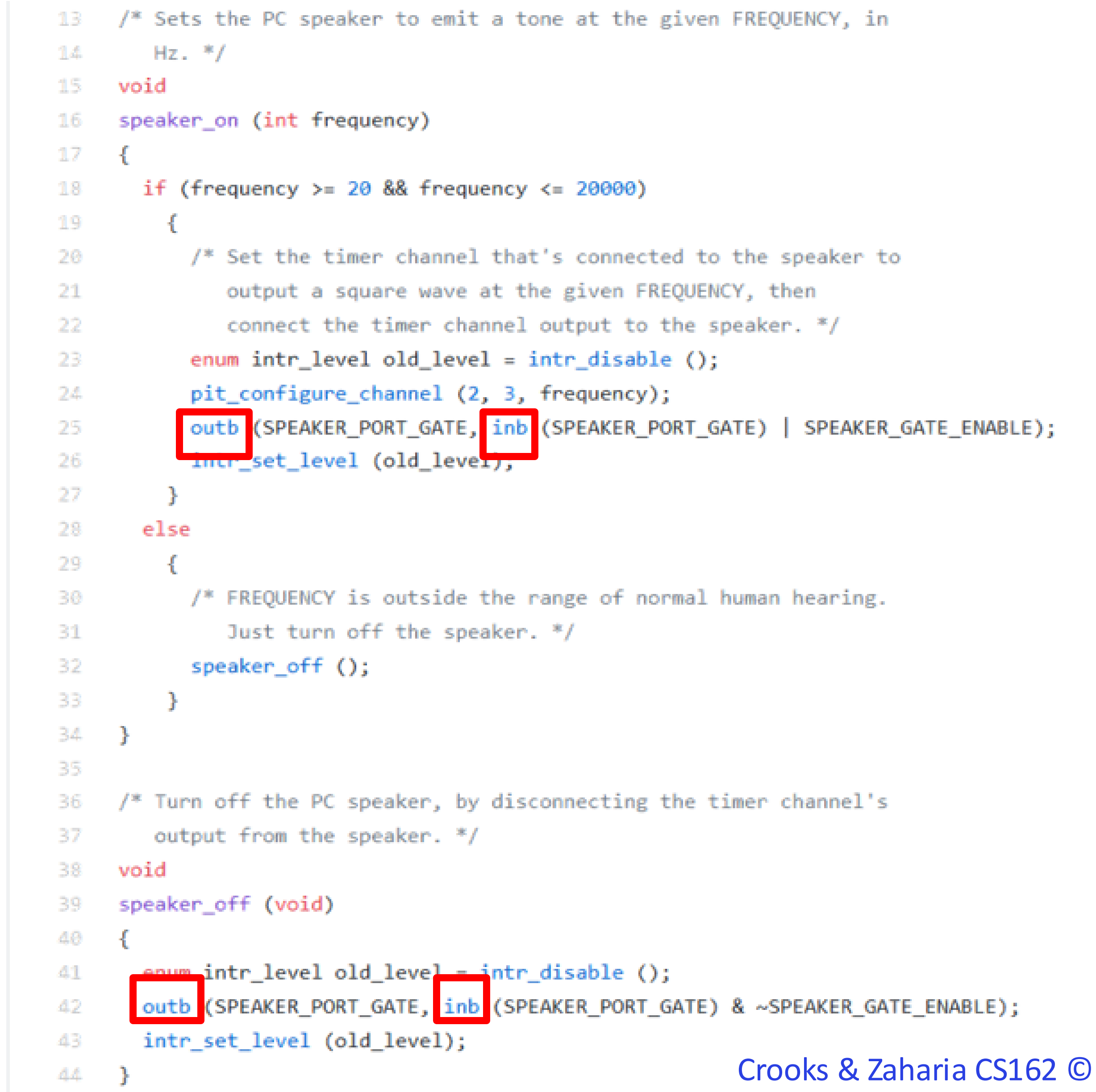
Example in Intel assembly: out 0x21,AL

Memory-mapped I/O: load/store instructions

Registers/memory appear in physical address space
I/O accomplished with load and store instructions

Example: Port-Mapped I/O in Pintos Speaker Driver

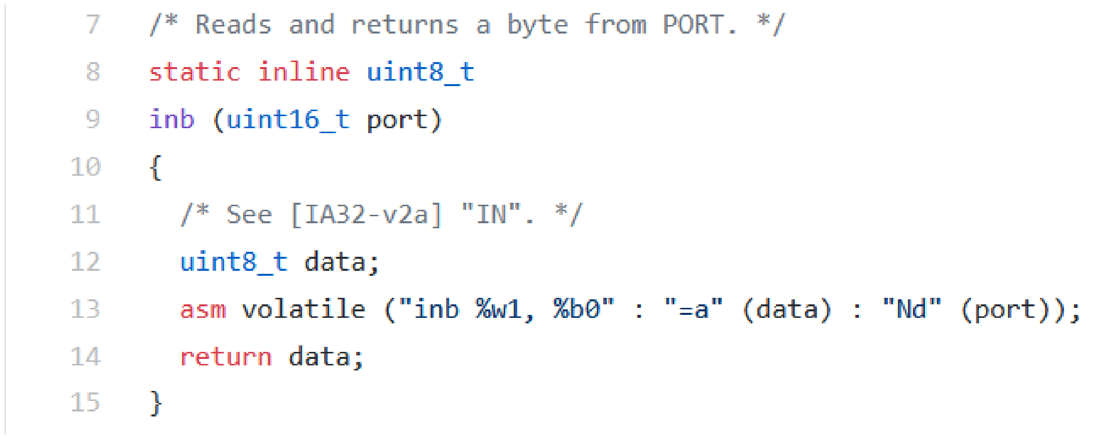
Pintos: devices/speaker.c



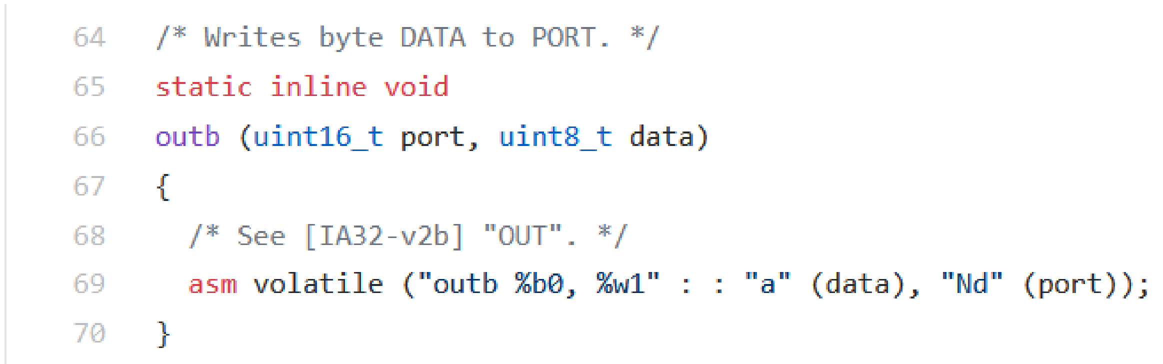
```
/* Sets the PC speaker to emit a tone at the given FREQUENCY, in
   Hz. */
void
speaker_on (int frequency)
{
  if (frequency >= 20 && frequency <= 20000)
    {
      /* Set the timer channel that's connected to the speaker to
         output a square wave at the given FREQUENCY, then
         connect the timer channel output to the speaker. */
      enum intr_level old_level = intr_disable ();
      pit_configure_channel (2, 3, frequency);
      outb (SPEAKER_PORT_GATE, inb (SPEAKER_PORT_GATE) | SPEAKER_GATE_ENABLE);
      intr_set_level (old_level);
    }
  else
    {
      /* FREQUENCY is outside the range of normal human hearing.
         Just turn off the speaker. */
      speaker_off ();
    }
}

/* Turn off the PC speaker, by disconnecting the timer channel's
   output from the speaker. */
void
speaker_off (void)
{
  enum intr_level old_level = intr_disable ();
  outb (SPEAKER_PORT_GATE, inb (SPEAKER_PORT_GATE) & ~SPEAKER_GATE_ENABLE);
  intr_set_level (old_level);
}
```

Pintos: threads/io.h



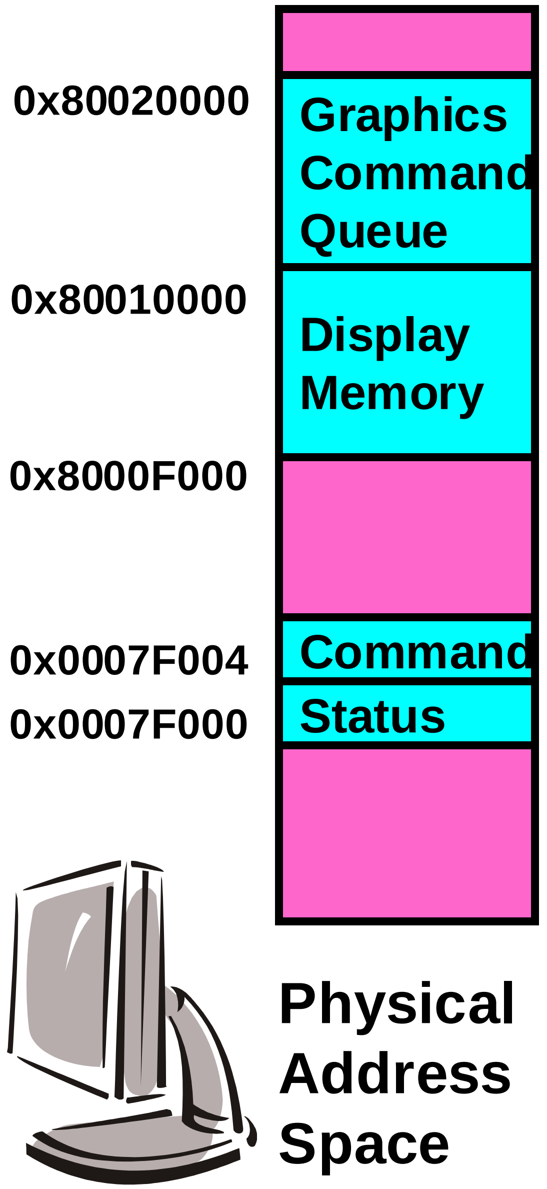
```
/* Reads and returns a byte from PORT. */
static inline uint8_t
inb (uint16_t port)
{
  /* See [IA32-v2a] "IN". */
  uint8_t data;
  asm volatile ("inb %w1, %b0" : "=a" (data) : "Nd" (port));
  return data;
}
```



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

Example: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - » Addresses set by HW jumpers or at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - » Addr: 0x8000F000 — 0x8000FFFF
  - Writing graphics description to cmd queue
    - » Say enter a set of triangles describing 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 address translation



A Simple Protocol for Talking to a Device

```
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**

CPU is responsible for moving data

How can we lower this overhead?

Interrupt-driven I/O vs Polling

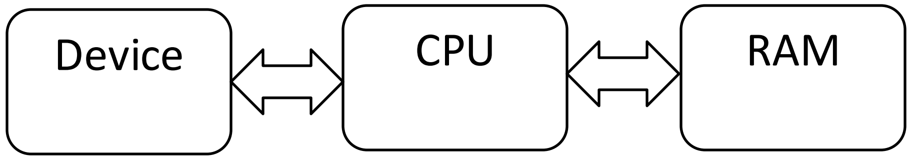
- Use hardware interrupts to avoid busy polling:
  - Allows CPU to process another task. Will get notified when task is done
  - Interrupt handler will read data & error code

- Is it always better to use interrupts?

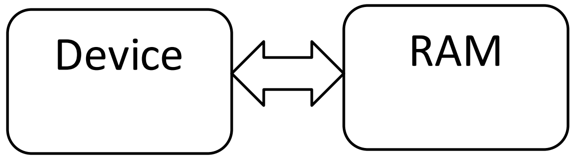
- Actual devices often support both polling and interrupts: e.g. wait for an interrupt from the network card to read the first packet, then poll its input queue memory space to look for other received packets

From programmed I/O to direct memory access

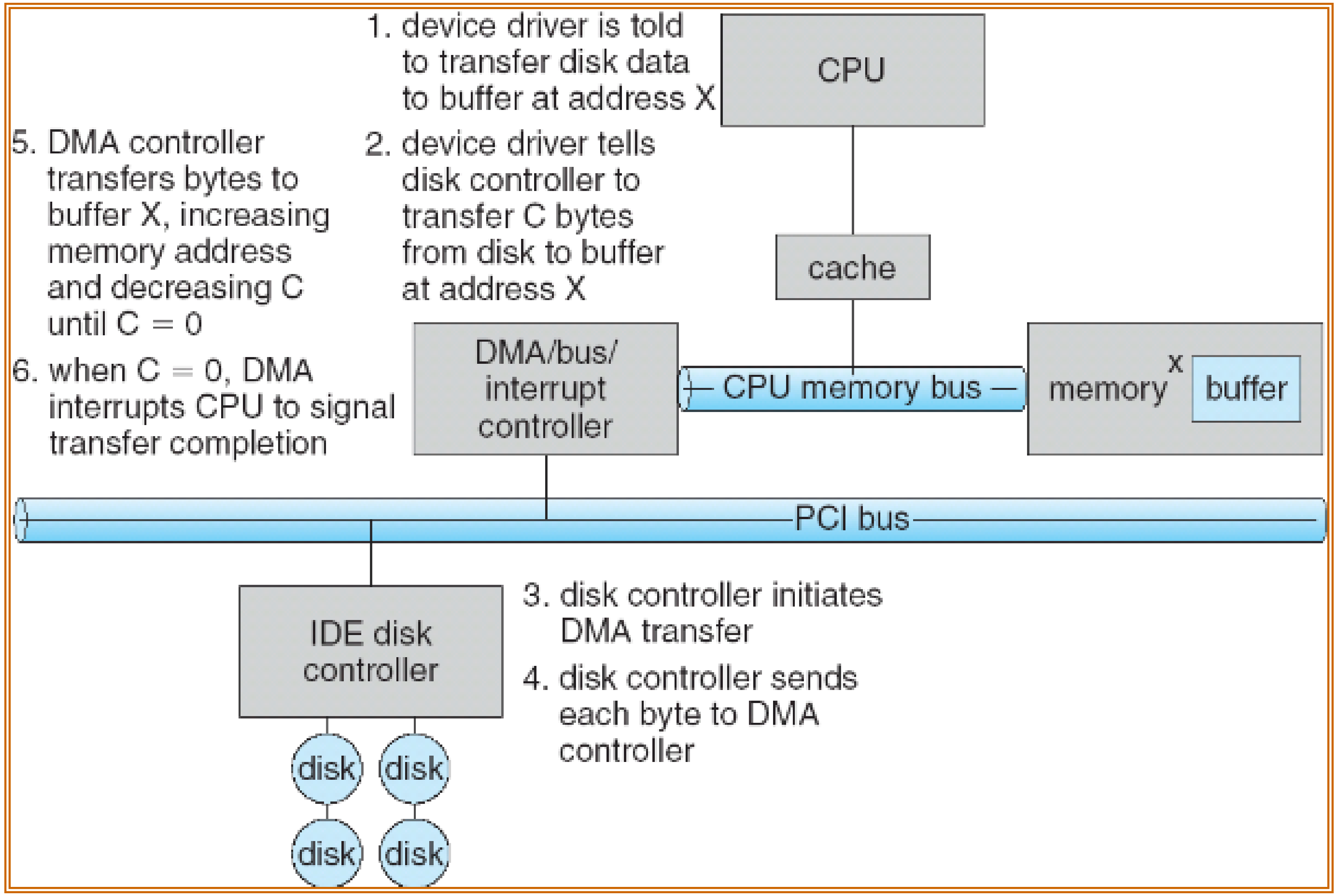
- With programmed I/O (our simple protocol):
  - CPU issues read request
  - Device interrupts CPU with data
  - CPU writes data to memory
  - Pros: simple hardware. Cons: Poor CPU is always busy!



- 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


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

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

- How do we fit devices with specific interfaces into the 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 that device hardware
  - Supports a standard, internal interface
  - Special device-specific configuration supported with the `ioctl()` system call

- Device Drivers are 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, your OS is 70% device-drivers

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

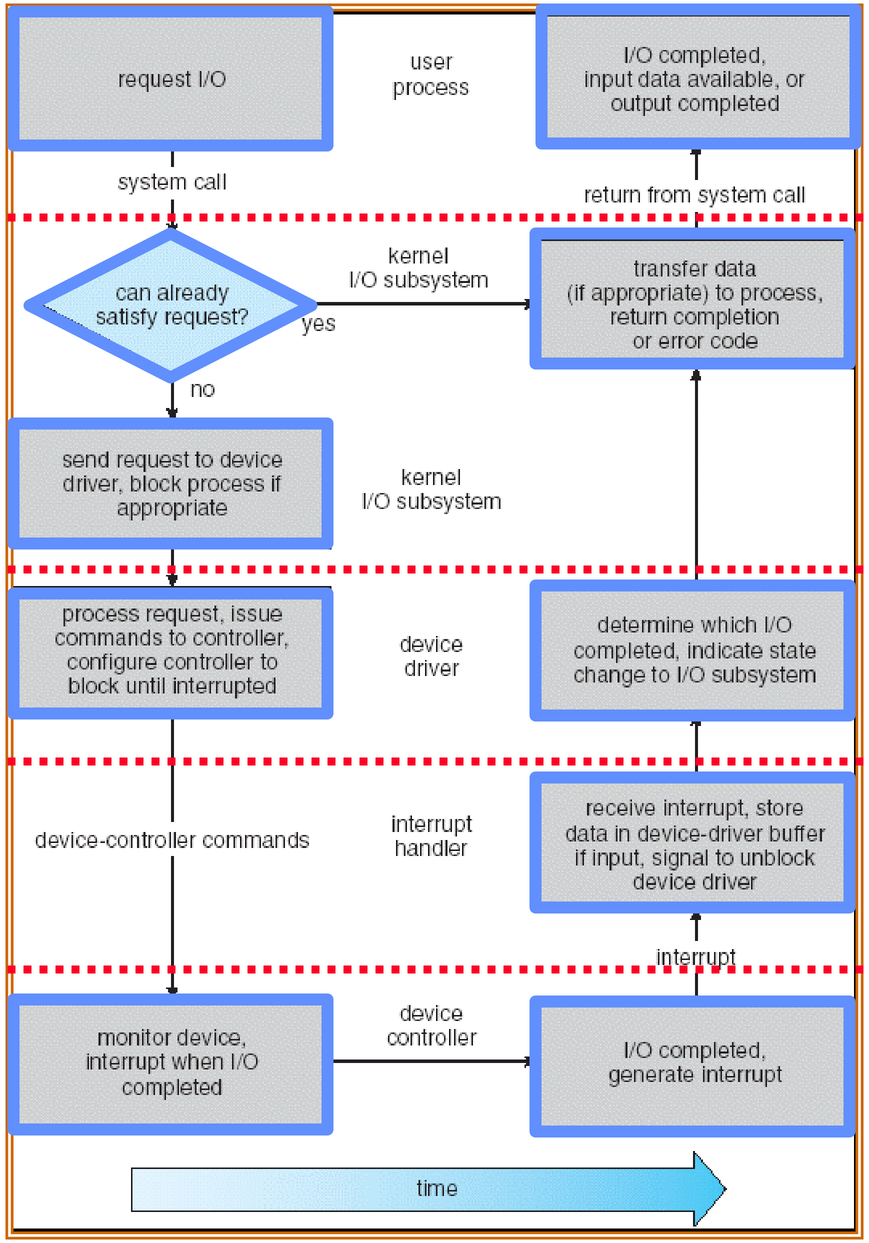
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
yes
kernel I/O subsystem
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

Conclusion

- I/O Devices Types:
  - Many different speeds (0.1 bytes/sec to 50+ GBytes/sec)
  - Different Access Patterns:
    - » Block Devices, Character Devices, Network Devices
- Device Controllers: Hardware that controls actual device
  - Processor Accesses through I/O instructions (port-mapped I/O) or load/store to special physical memory addresses (memory-mapped I/O)
- 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

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 (or lets user check/poll)
  - When send data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user (or lets user check/poll)

Storage Devices

Magnetic disks

– Storage that rarely becomes corrupted

– Large capacity at low cost

– Block level random access (except for SMR – later!)

– Slow performance for random access

– Better performance for sequential access

Flash memory

– Storage that rarely becomes corrupted

– Capacity at intermediate cost (5-20x disk)

– Block level random access

– Good performance for reads; worse for random writes

– Wear patterns issue