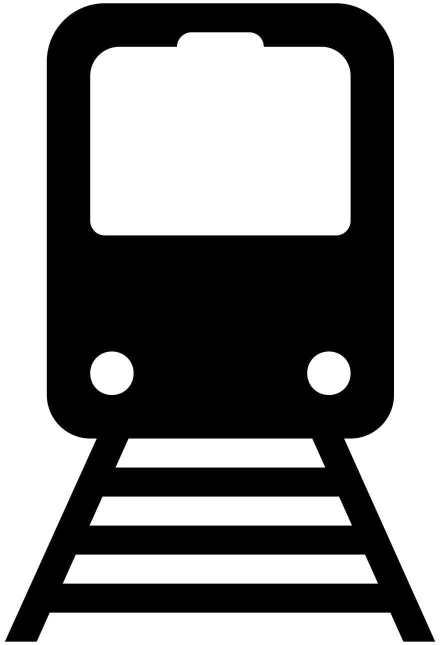
CS162
Operating Systems and Systems Programming
Lecture 17

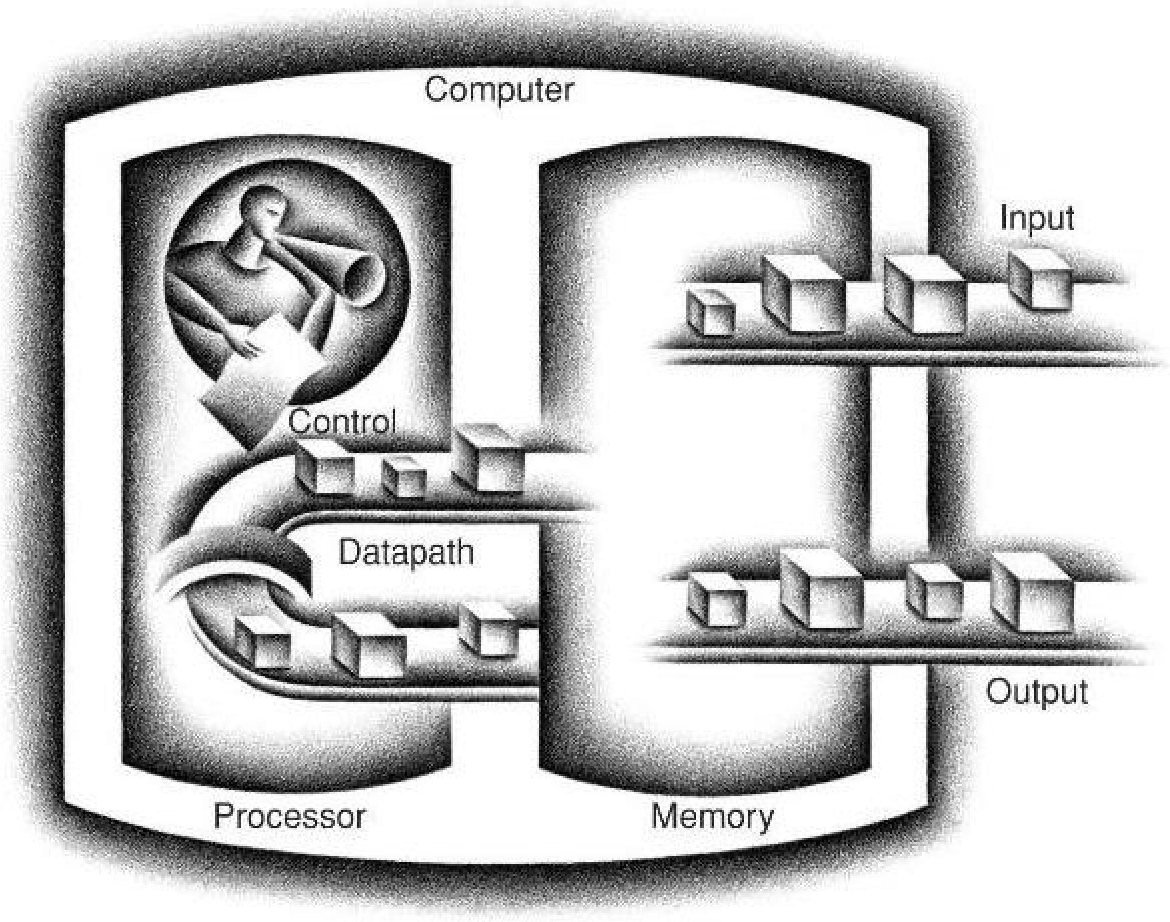
General I/O, Storage Devices

Course Map

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

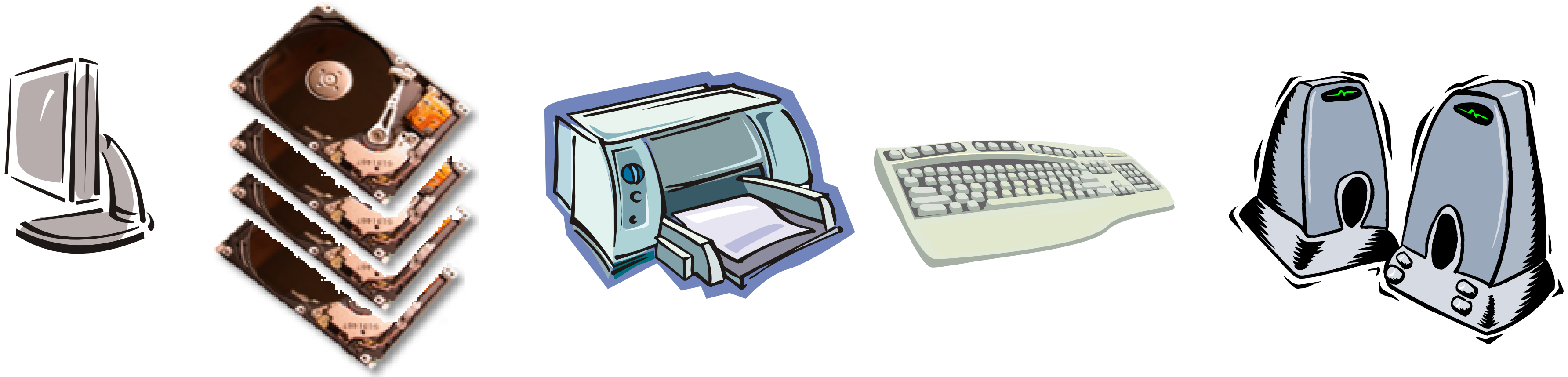


Recall: Five Components of a Computer



CPU: You need to get out more!

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



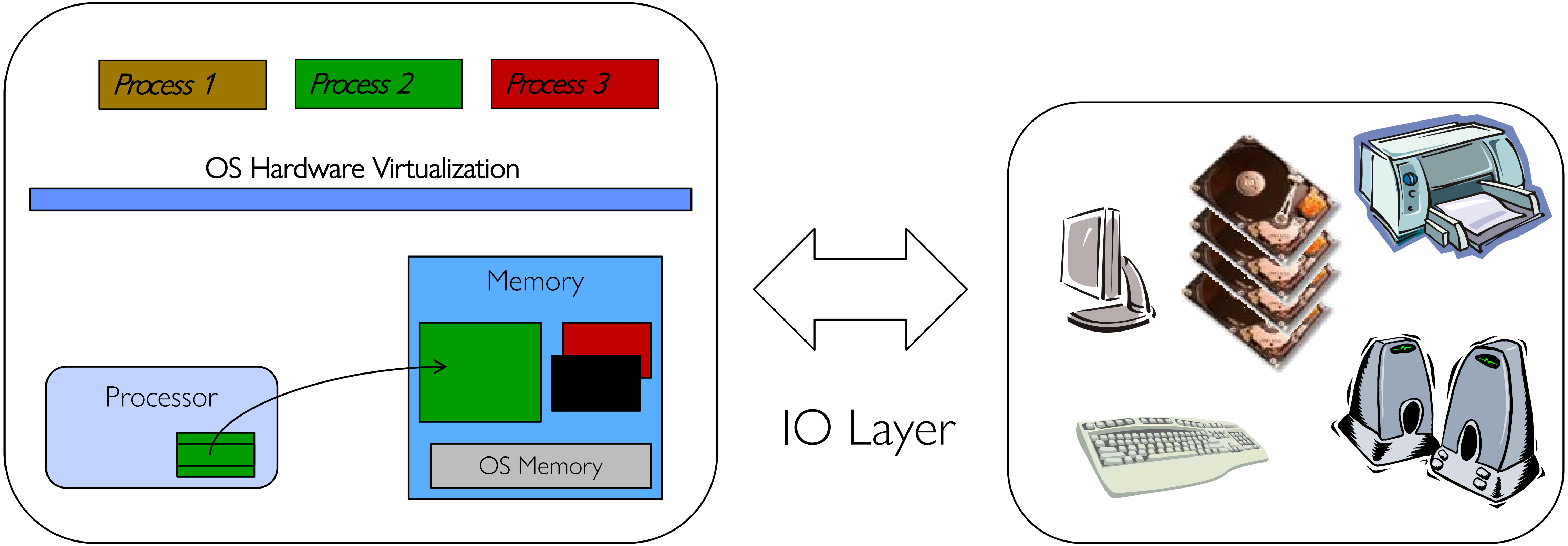
Want Standard Interfaces to Devices

- 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



Process 1

Process 2

Process 3

OS Hardware Virtualization

Memory

Processor

OS Memory

IO Layer

Virtual Machine Abstraction

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);
```

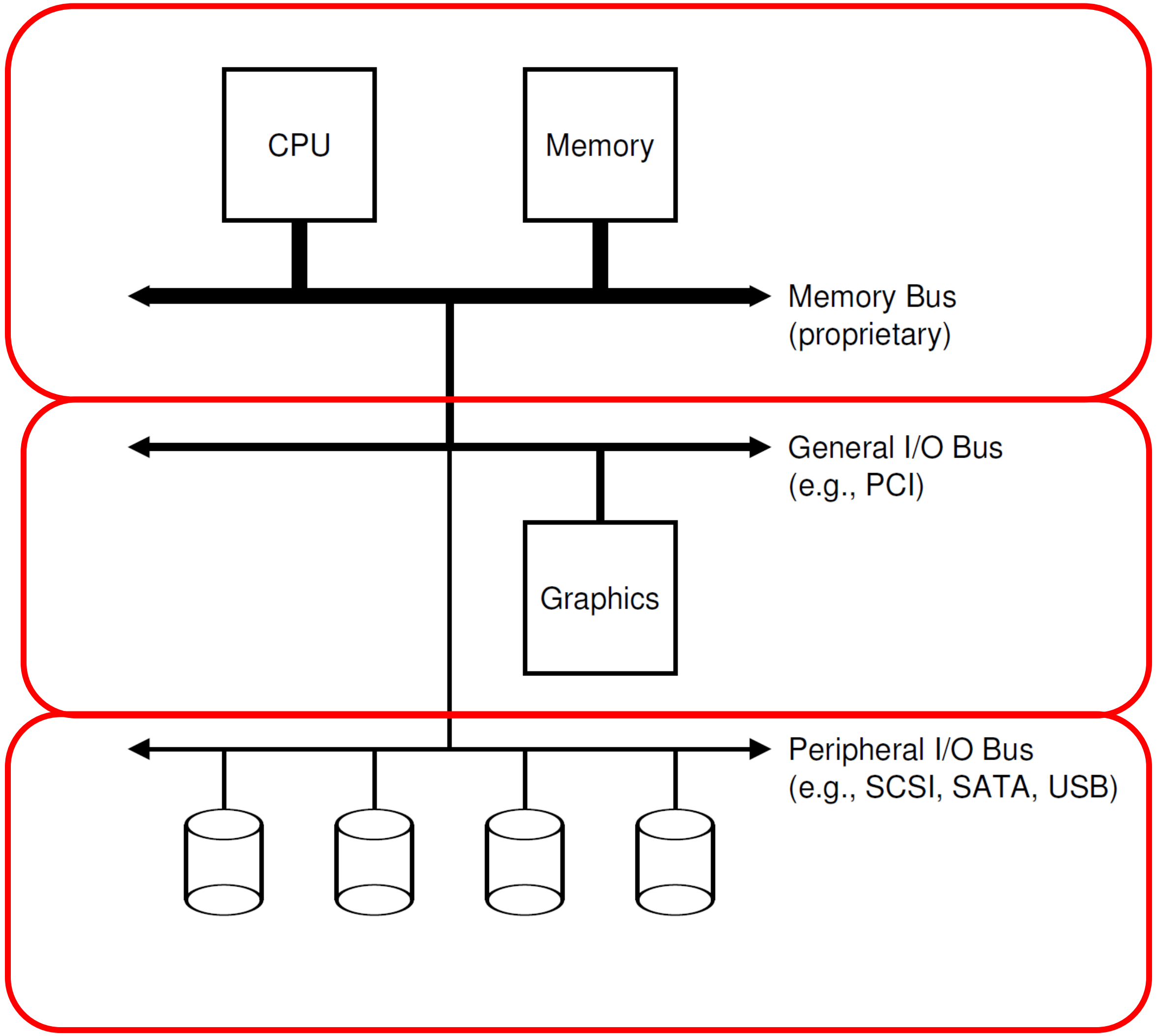
  - Why? Because code that controls devices (“device driver”) implements standard interface

- 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

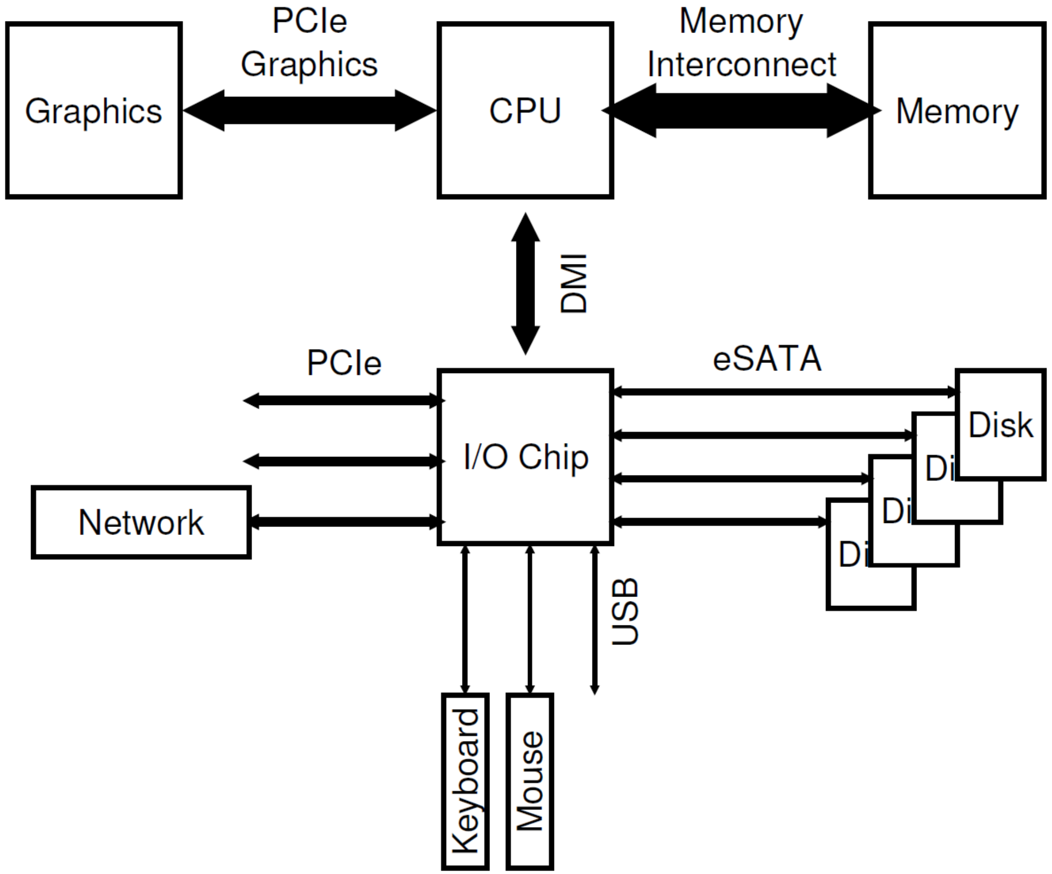
- But… thousands of devices, each slightly different
  - » OS: How can we **standardize** the interfaces to these devices?
- Devices unreliable: media failures and transmission errors
  - » OS: How can we make them **reliable**???
- Devices unpredictable and/or slow
  - » OS: How can we **manage** them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

Simplified IO architecture

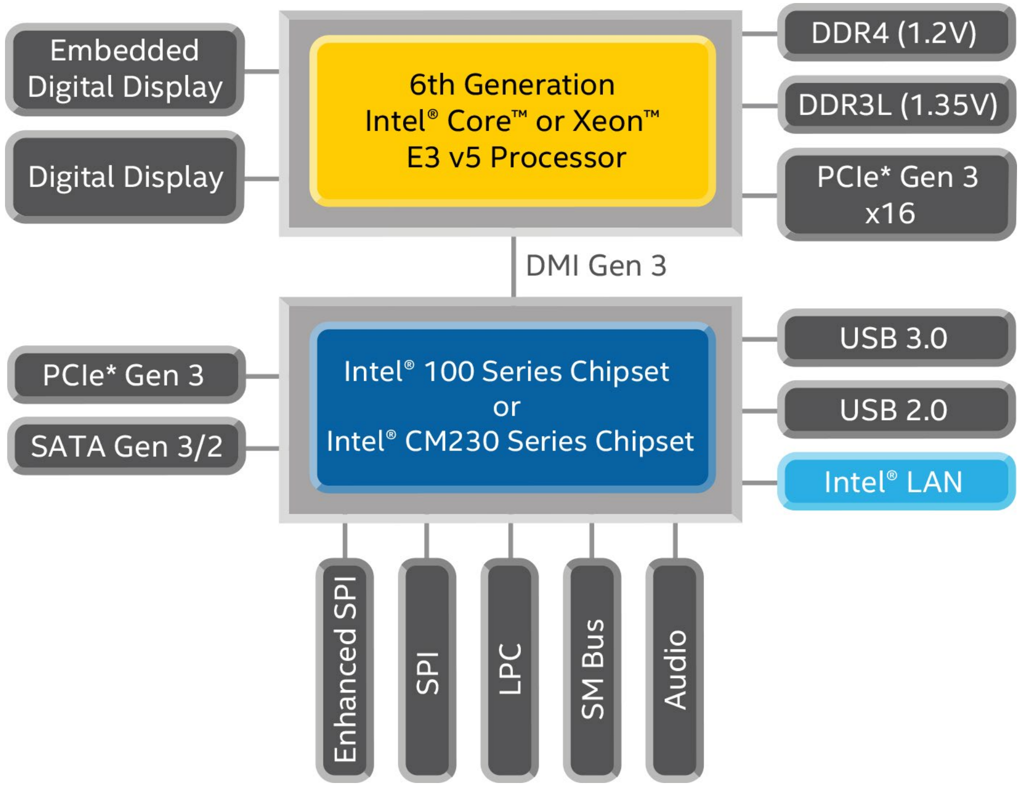


Follows a hierarchical structure because of cost: the faster the bus, the more expensive it is

Intel's Z270 Chipset



Sky Lake I/O: PCH

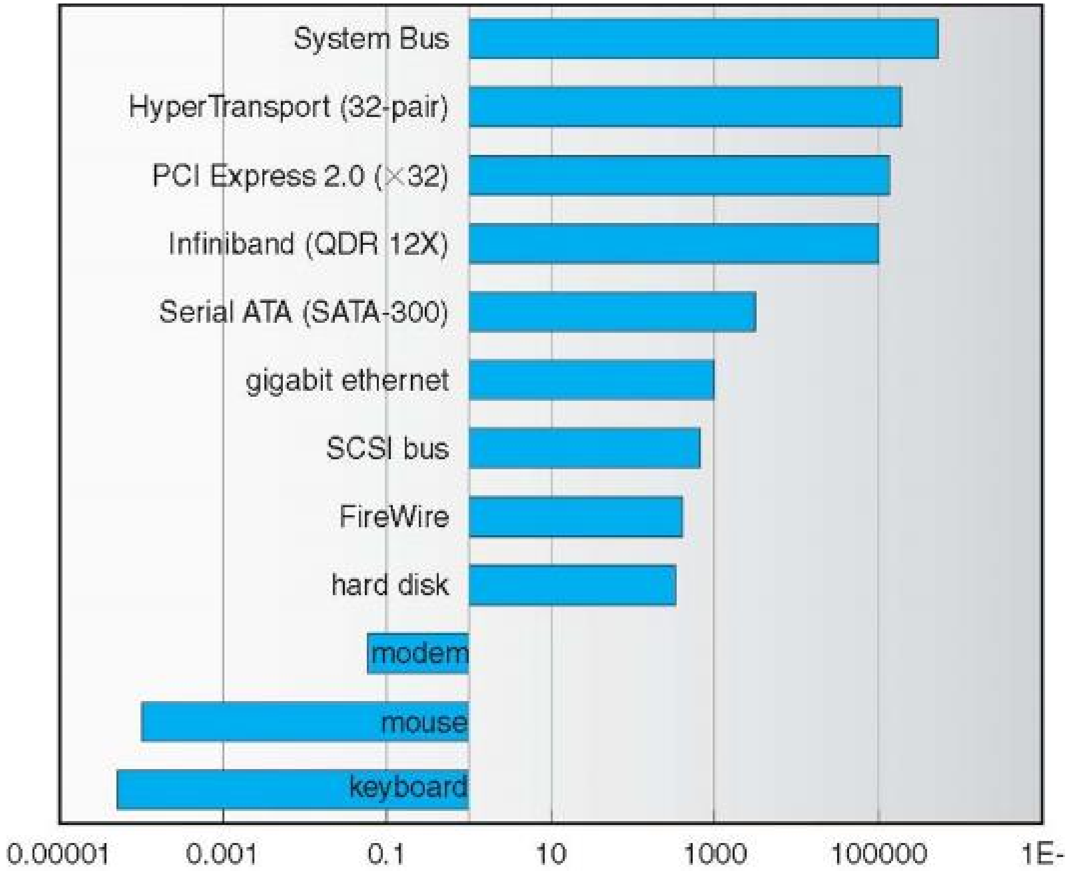


Sky Lake
System Configuration

- 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)

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

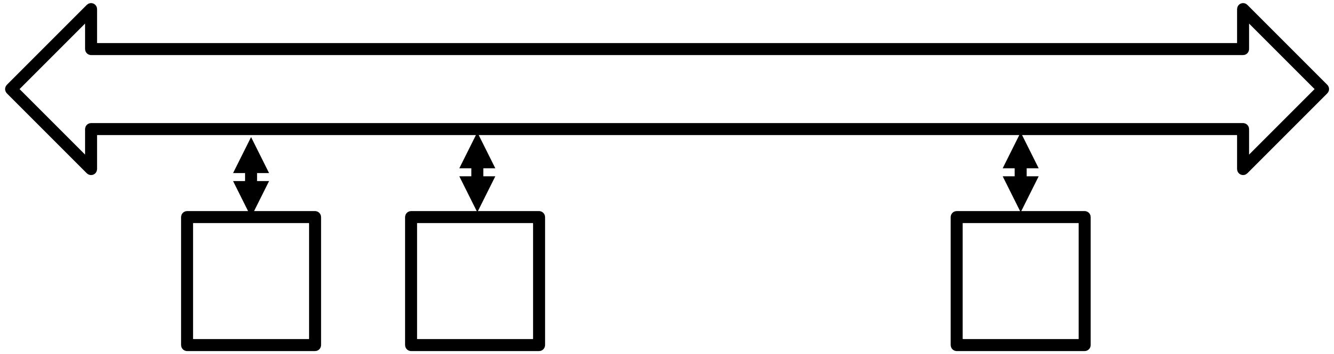
Device rates vary over 12 orders of magnitude!!!



Two questions

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

What's a bus?

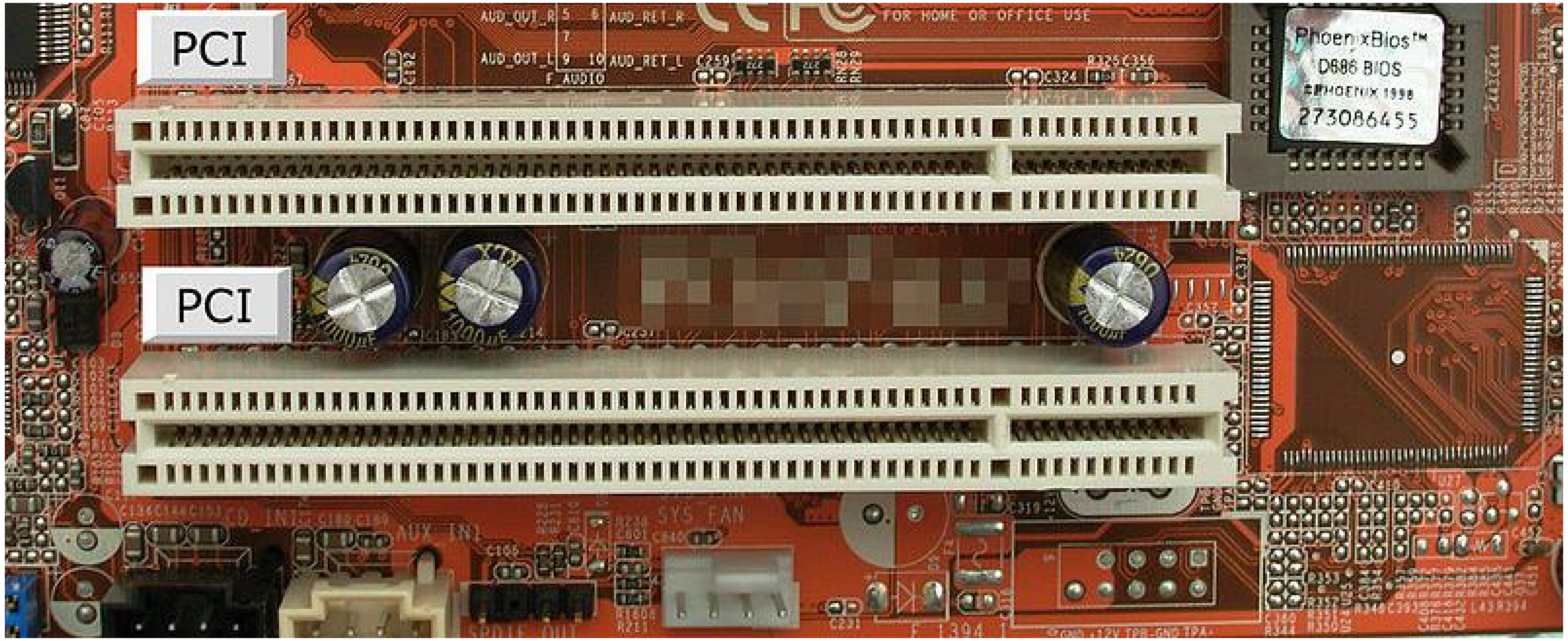


- Common set of wires for communication among hardware devices plus protocols for carrying out data transfer transactions
- Split into three parts: data bus, address bus, and control bus
- Protocol: initiator requests access, arbitration to grant, identification of recipient, handshake to convey address,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

- No longer a parallel bus

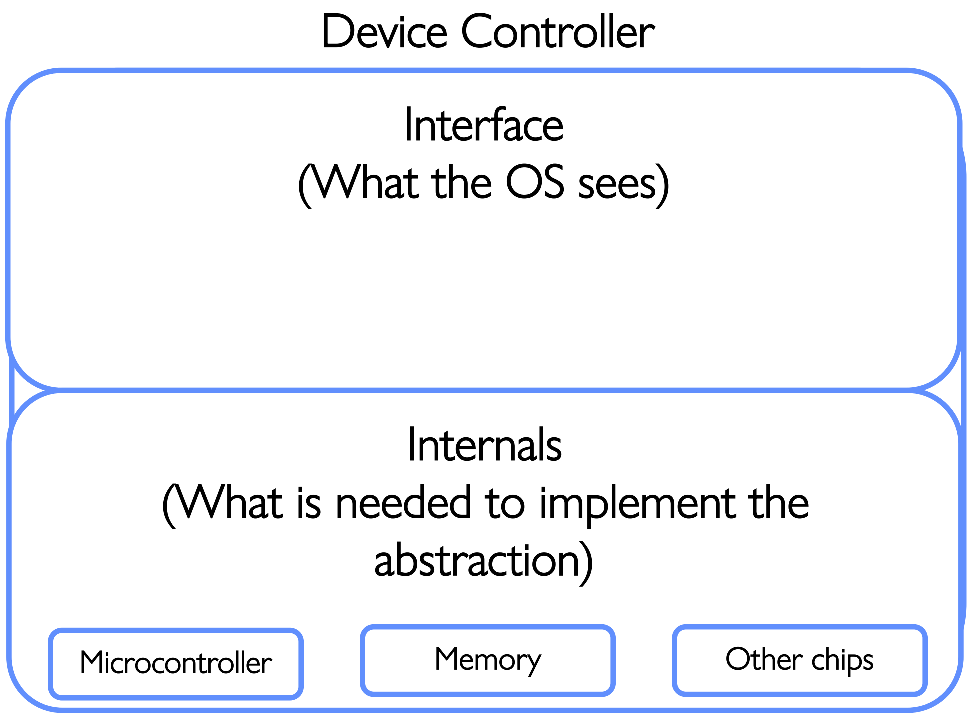
- Really a **collection of fast serial channels** or "lanes"
- Devices can use as many as they need to achieve a desired bandwidth

- 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

How does the processor talk to devices?

- Remember, it's all about abstractions!



Device Controller

Interface
(What the OS sees)

Internals
(What is needed to implement the abstraction)

Microcontroller

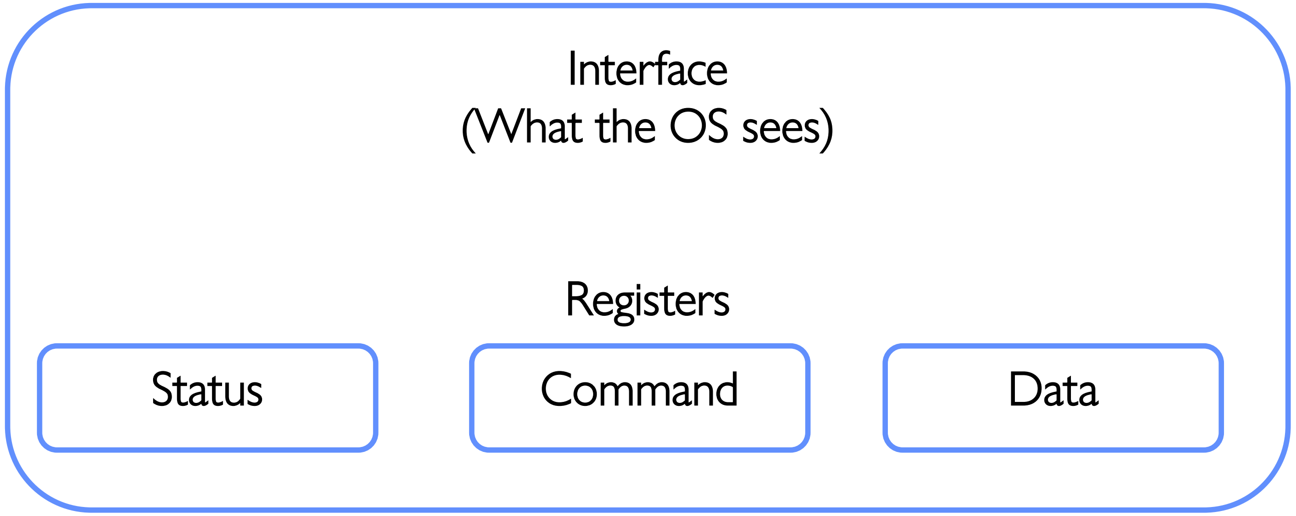
Memory

Other chips

Hardware interface device presents to OS

Hardware interface device presents to OS

How does the processor talk to devices?



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

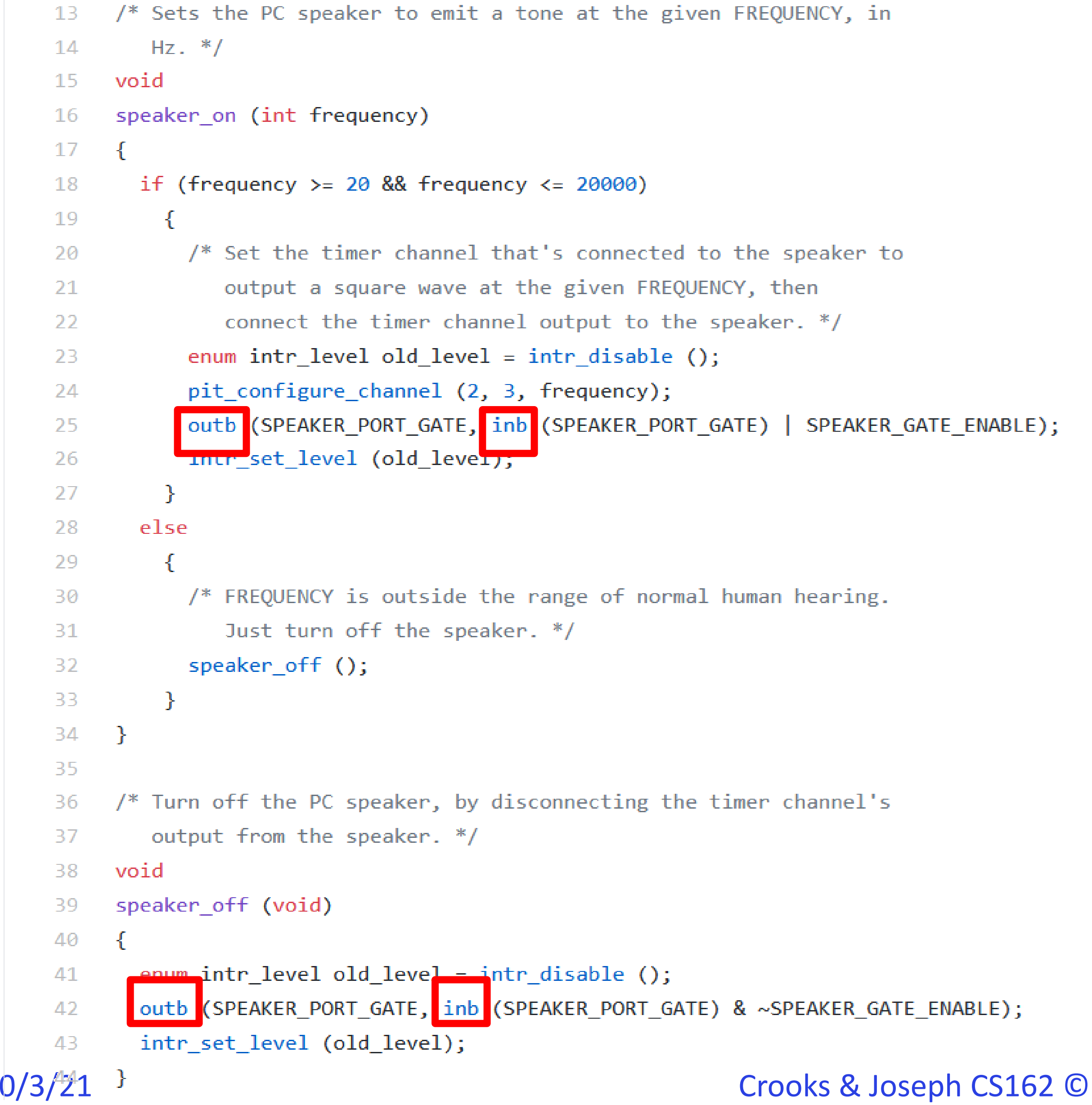
Example from the Intel architecture: out 0x21,AL

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

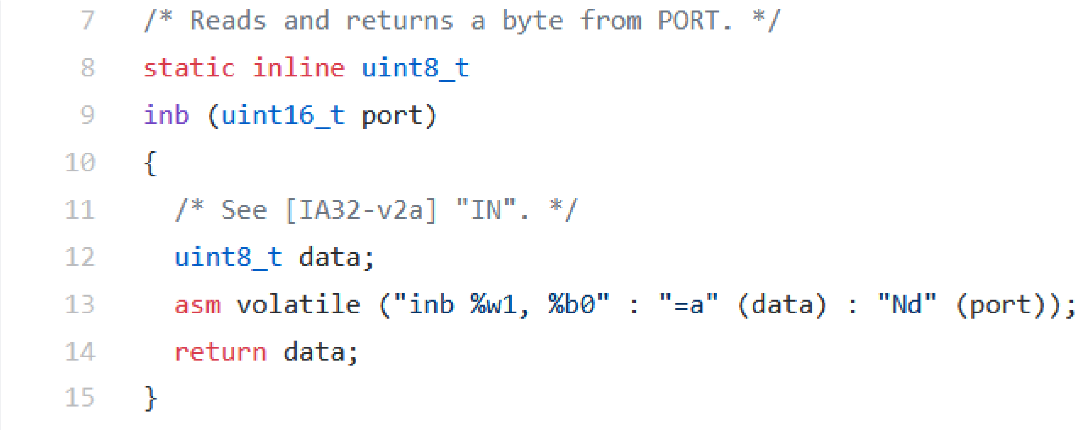
Pintos: devices/speaker.c



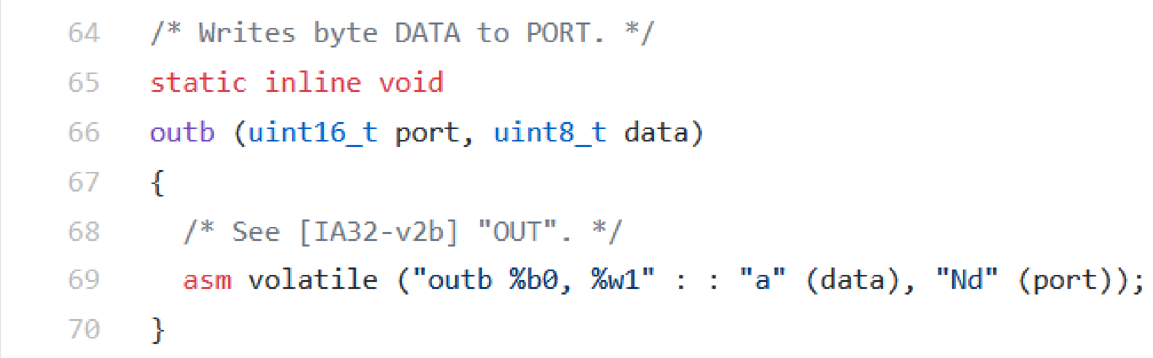
```c
/* Sets the PC speaker to emit a tone at the given FREQUENCY, in
   Hz. */
void
speaker_on (int frequency)
{
  if (frequency >= 20 && frequency <= 20000)
    {
      /* Set the timer channel that's connected to the speaker to
         output a square wave at the given FREQUENCY, then
         connect 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



```c
/* 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;
}
```



```c
/* 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));
}
```

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
```

CPU is responsible for moving data

Protocol does a lot of polling!

How can we lower this overhead?

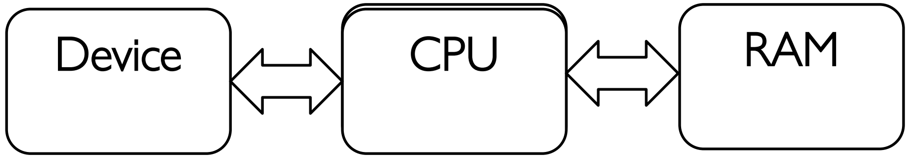
Polling vs Interrupt-driven IO

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

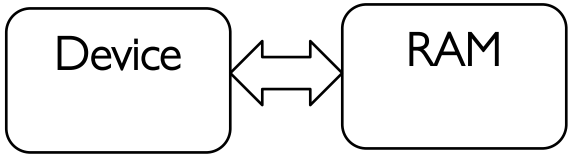
- Is it always better to use interrupts?

From programmed IO to direct memory access

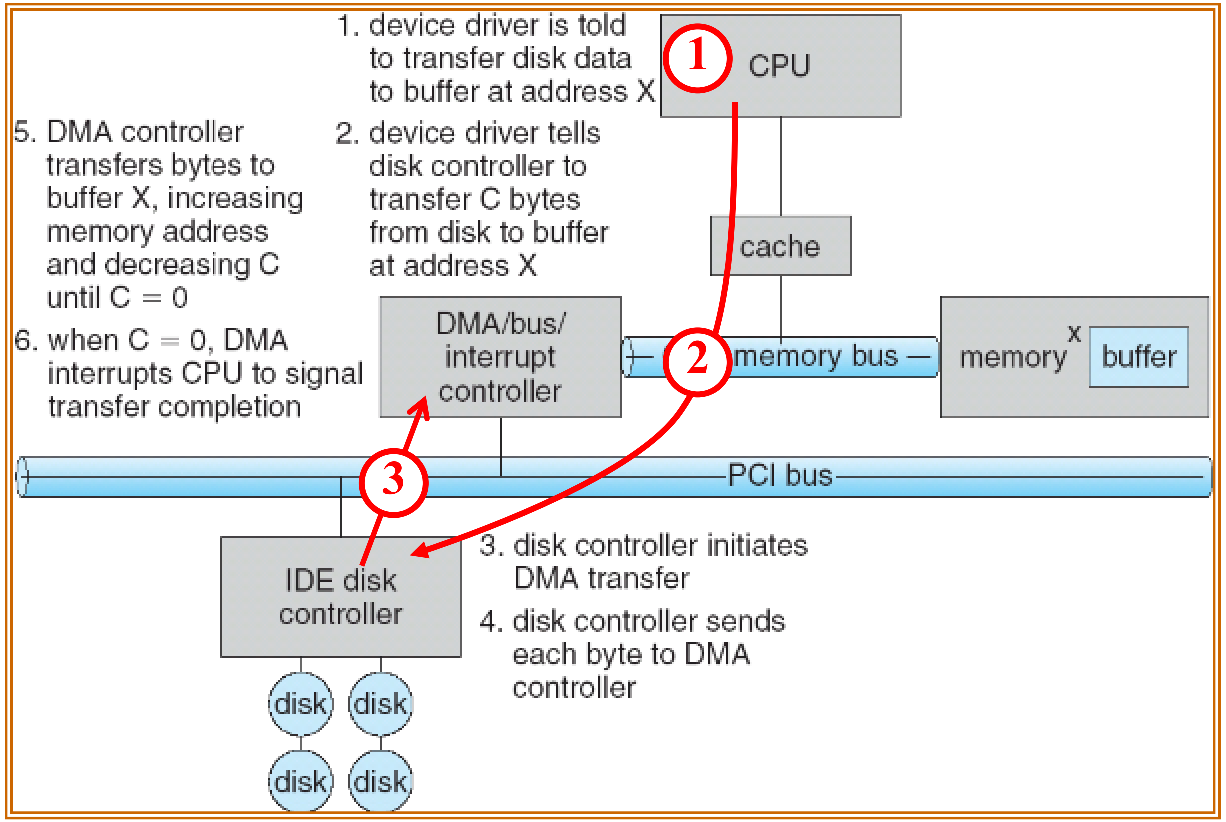
- With programmed IO (simple protocol):
  - CPU issues read request
  - Device interrupts CPU with data
  - CPU writes data to memory
  - Pros: simple hardware. Cons: Poor CPU is always busy!



- 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
  - 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

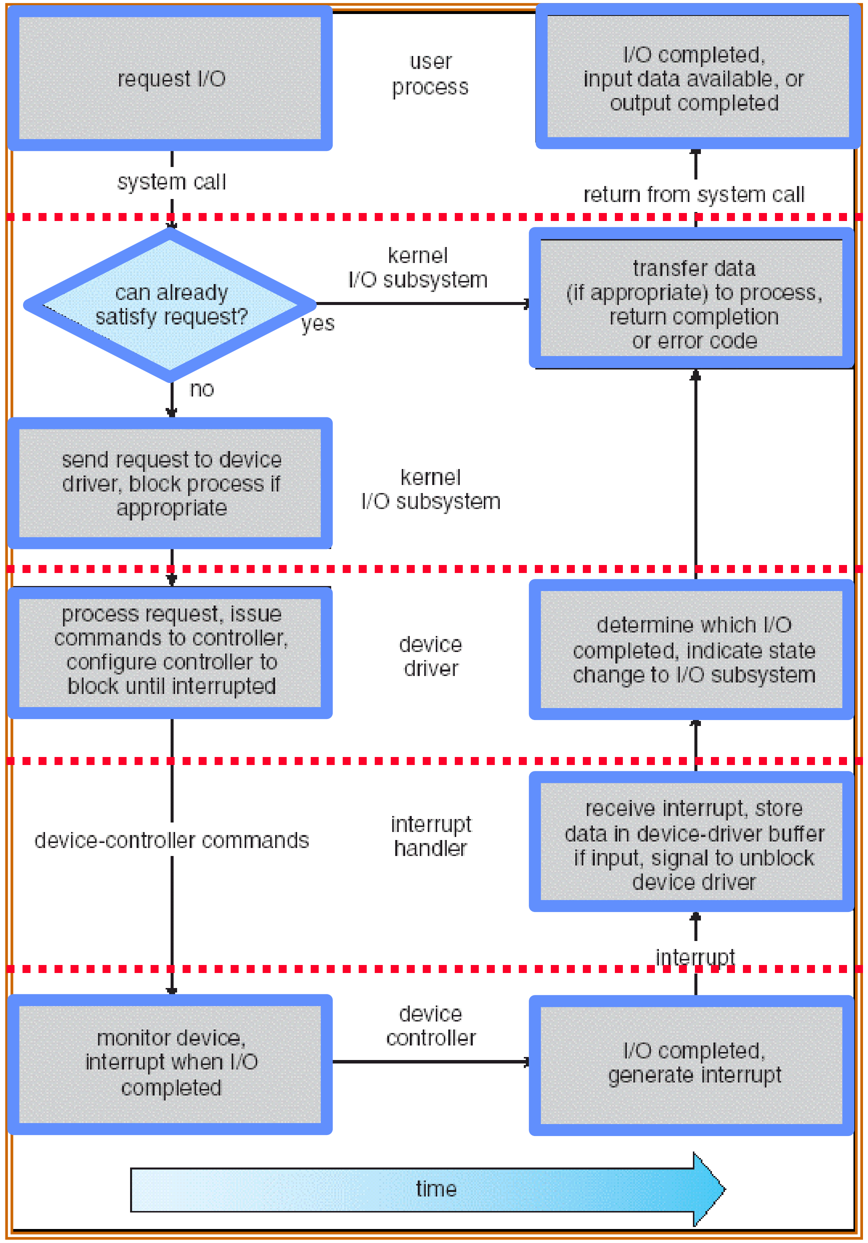
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware



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 GBytes/sec)
  - Different Access Patterns:
    - Block Devices, Character Devices, Network Devices
- 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