Lecture 15:
I/O Subsystems

Ack: the slides were borrowed from Prof. Kubiatowicz's CS162 at UC Berkeley and Prof. Weatherspoon's CS3410 at Cornell with their permission to use.

Goals for Today

Computer System Organization

How does a processor interact with its environment?

- I/O Overview

How to talk to device?

- Programmed I/O or Memory-Mapped I/O

How to get events?

- Polling or Interrupts

How to transfer lots of data?

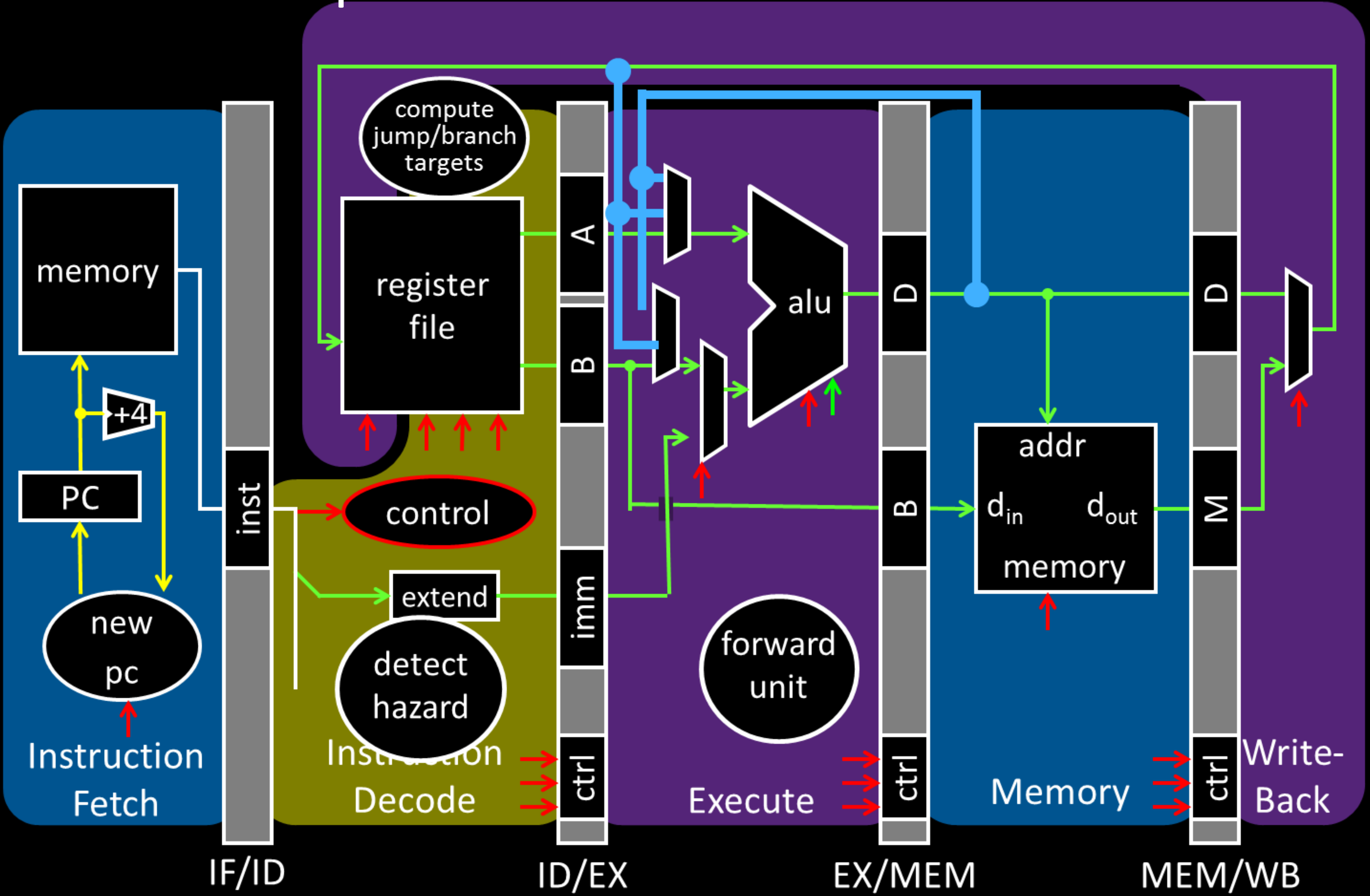
- Direct Memory Access (DMA)

Next Goal

How does a processor interact with its environment?

Big Picture: Input/Output (I/O)

How does a processor interact with its environment?



Big Picture: Input/Output (I/O)

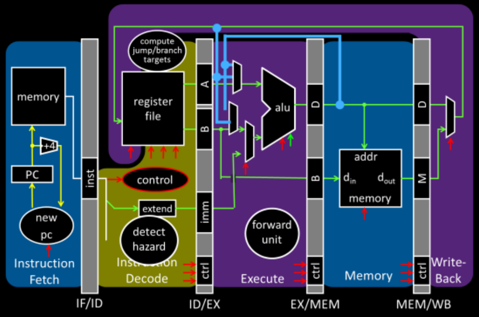
How does a processor interact with its environment?

Computer System Organization =

Memory +

Datapath +

Control +



Input +

Output

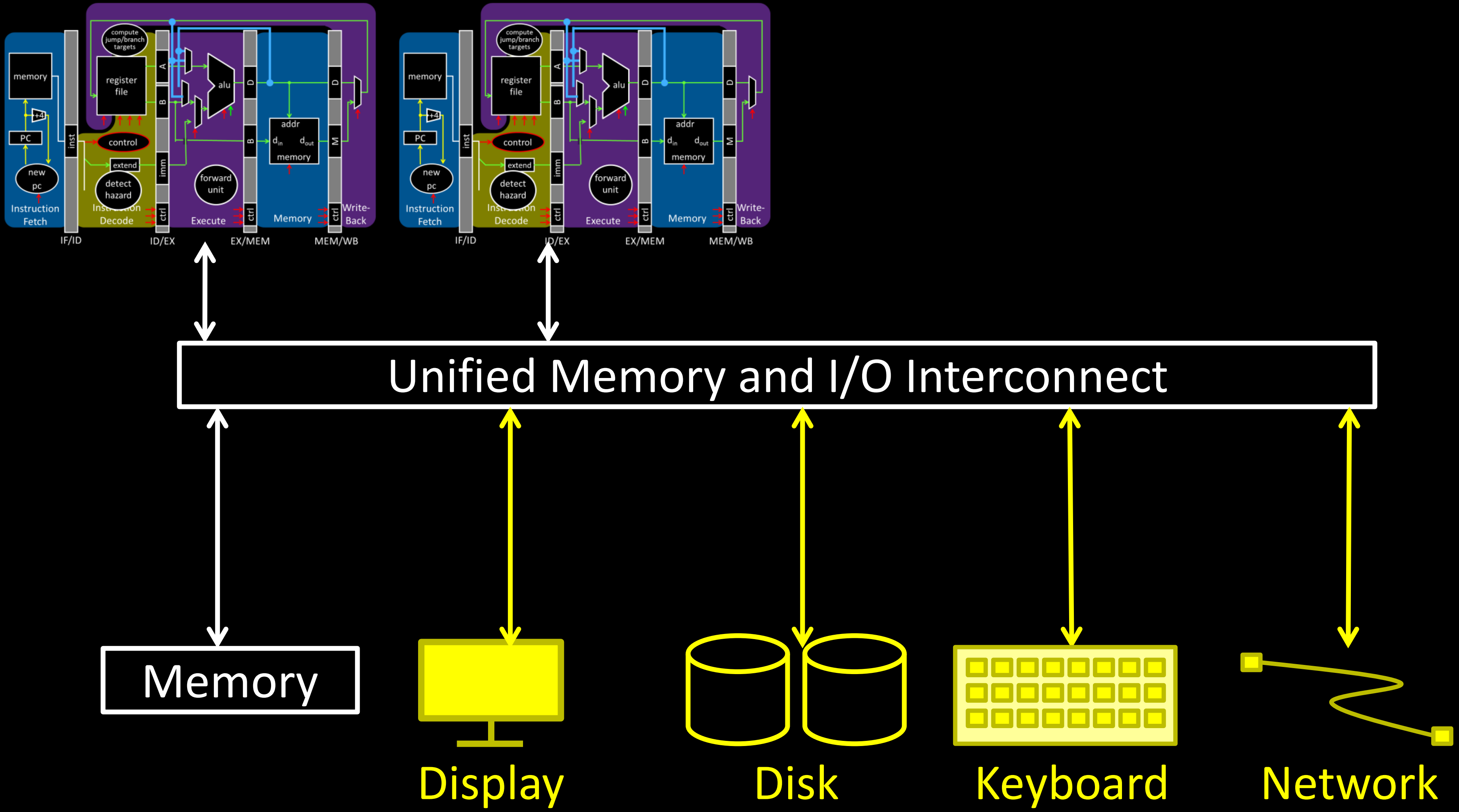
I/O Devices Enables Interacting with Environment

Device	Behavior	Partner	Data Rate (b/sec)
Keyboard	Input	Human	100
Mouse	Input	Human	3.8k
Sound Input	Input	Machine	3M
Voice Output	Output	Human	264k
Sound Output	Output	Human	8M
Laser Printer	Output	Human	3.2M
Graphics Display	Output	Human	800M – 8G
Network/LAN	Input/Output	Machine	100M – 10G
Network/Wireless LAN	Input/Output	Machine	11 – 54M
Optical Disk	Storage	Machine	5 – 120M
Flash memory	Storage	Machine	32 – 200M
Magnetic Disk	Storage	Machine	800M – 3G

Attempt#1: All devices on one interconnect

Replace ***all*** devices as the interconnect changes

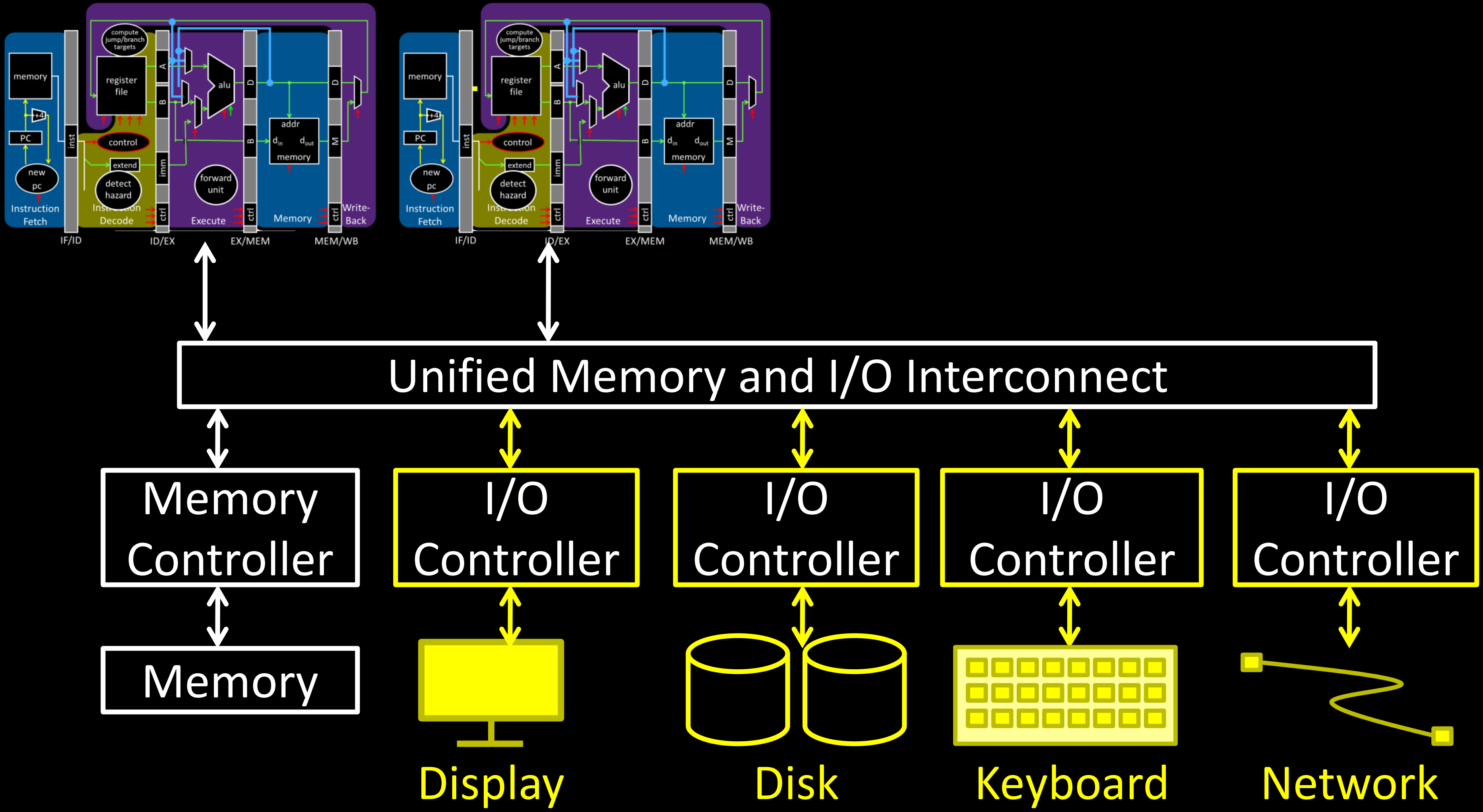
e.g. keyboard speed == main memory speed ?!



Attempt#2: I/O Controllers

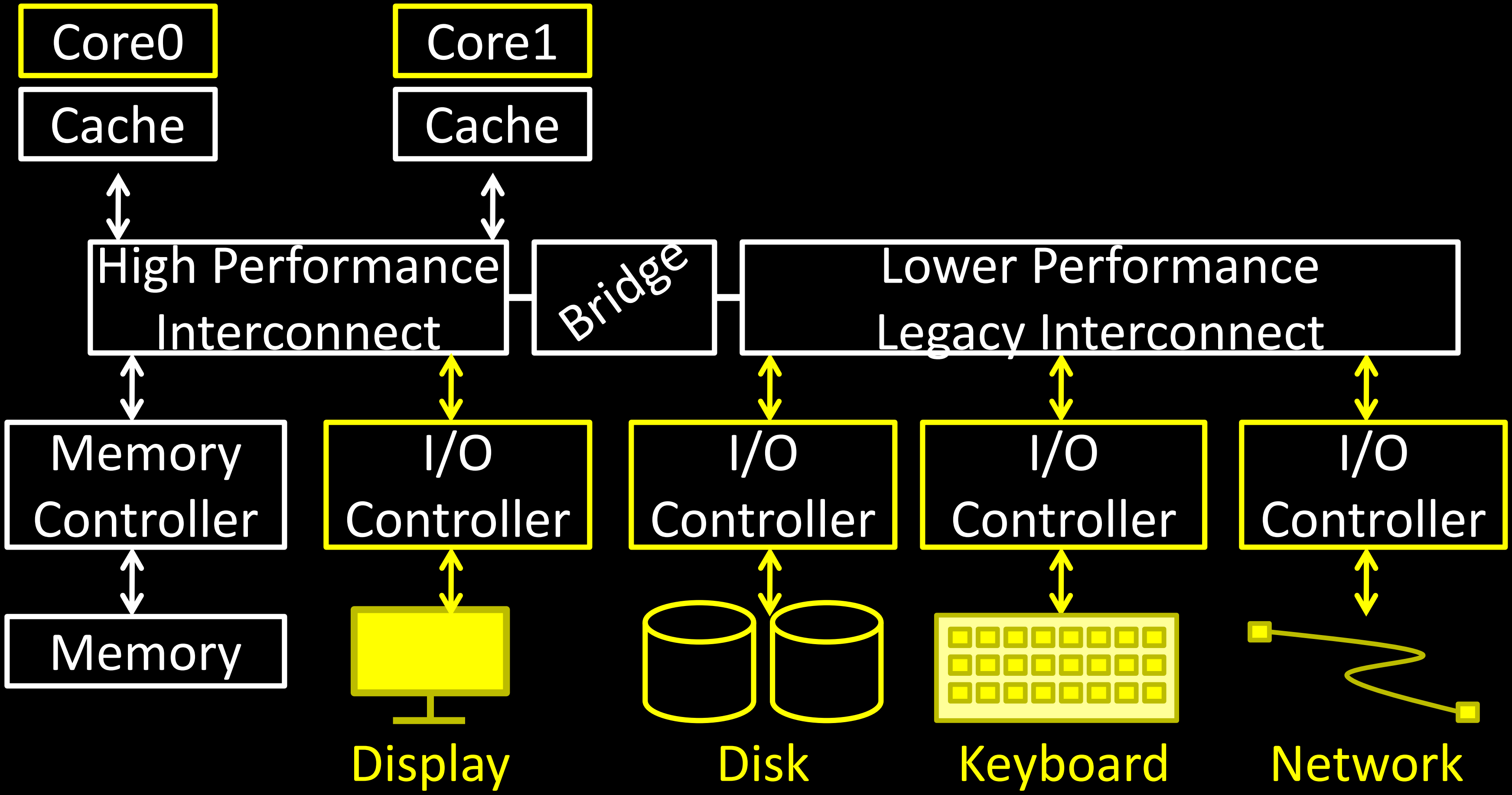
Decouple I/O devices from Interconnect

Enable smarter I/O interfaces



Attempt#3: I/O Controllers + Bridge

Separate high-performance processor, memory, display interconnect from lower-performance interconnect



Bus Parameters

Width = number of wires

Transfer size = data words per bus transaction

Synchronous (with a bus clock)
or asynchronous (no bus clock / “self clocking”)

Bus Types

Processor – Memory ("Front Side Bus". Also QPI)

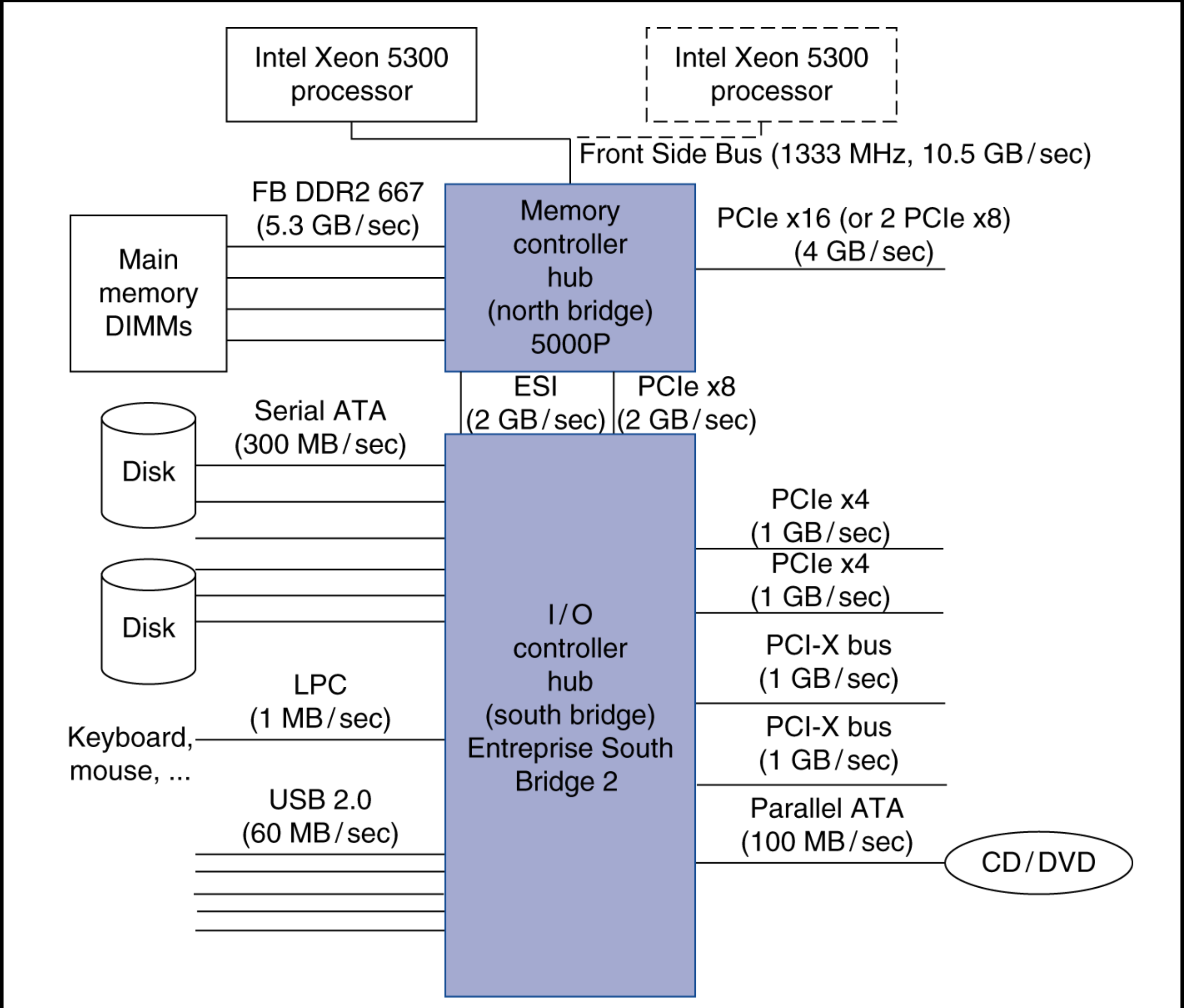
- Short, fast, & wide
- Mostly fixed topology, designed as a "chipset"
  - CPU + Caches + Interconnect + Memory Controller

I/O and Peripheral busses (PCI, SCSI, USB, LPC, …)

- Longer, slower, & narrower
- Flexible topology, multiple/varied connections
- Interoperability standards for devices
- Connect to processor-memory bus through a bridge

Attempt#3: I/O Controllers + Bridge

Separate high-performance processor, memory, display interconnect from lower-performance interconnect



Intel Xeon 5300 processor
Intel Xeon 5300 processor
Front Side Bus (1333 MHz, 10.5 GB/sec)
FB DDR2 667 (5.3 GB/sec)
Main memory DIMMs
Memory controller hub (north bridge) 5000P
PCIe x16 (or 2 PCIe x8) (4 GB/sec)
ESI (2 GB/sec)
PCIe x8 (2 GB/sec)
Serial ATA (300 MB/sec)
Disk
Disk
PCIe x4 (1 GB/sec)
PCIe x4 (1 GB/sec)
PCI-X bus (1 GB/sec)
PCI-X bus (1 GB/sec)
Parallel ATA (100 MB/sec)
CD/DVD
I/O controller hub (south bridge) Entreprise South Bridge 2
LPC (1 MB/sec)
Keyboard, mouse, ...
USB 2.0 (60 MB/sec)

Example Interconnects

Name	Use	Devics per channel	Channel Width	Data Rate (B/sec)
Firewire 800	External	63	4	100M
USB 2.0	External	127	2	60M
Parallel ATA	Internal	1	16	133M
Serial ATA (SATA)	Internal	1	4	300M
PCI 66MHz	Internal	1	32-64	533M
PCI Express v2.x	Internal	1	2-64	16G/dir
Hypertransport v2.x	Internal	1	2-64	25G/dir
QuickPath (QPI)	Internal	1	40	12G/dir

Interconnecting Components

Interconnects are (were?) busses

- parallel set of wires for data and control
- shared channel
  - multiple senders/receivers
  - everyone can see all bus transactions
- bus protocol: rules for using the bus wires

e.g. Intel Xeon

Alternative (and increasingly common):

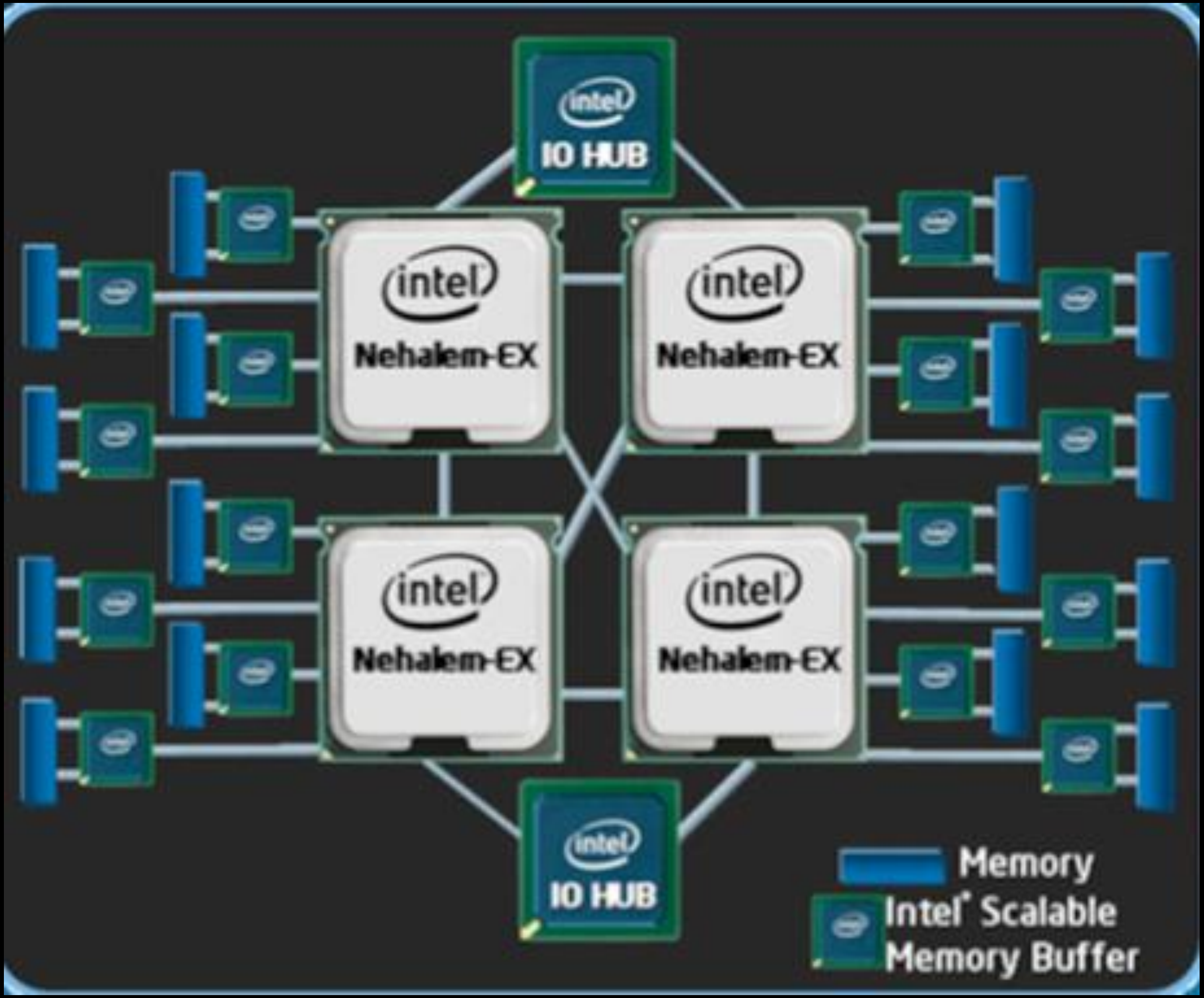
- dedicated point-to-point channels

e.g. Intel Nehalem

Attempt#4: I/O Controllers+Bridge+ NUMA

Remove bridge as bottleneck with Point-to-point interconnects

E.g. Non-Uniform Memory Access (NUMA)



Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Next Goal

How does the processor interact with I/O devices?

I/O Device Driver Software Interface

Set of methods to write/read data to/from device and control device

Example: Linux Character Devices

```
// Open a toy " echo " character device
int fd = open("/dev/echo", O_RDWR);

// Write to the device
char write_buf[] = "Hello World!";
write(fd, write_buf, sizeof(write_buf));

// Read from the device
char read_buf [32];
read(fd, read_buf, sizeof(read_buf));

// Close the device
close(fd);

// Verify the result
assert(strcmp(write_buf, read_buf)==0);
```

I/O Device API

Typical I/O Device API

- a set of read-only or read/write registers

Command registers

- writing causes device to do something

Status registers

- reading indicates what device is doing, error codes, …

Data registers

- Write: transfer data to a device
- Read: transfer data from a device

Every device uses this API

Communication Interface

Q: How does ~~program~~ ~~OS~~ code talk to device?

A: special instructions to talk over special busses

Programmed I/O ← Interact with cmd, status, and data device registers directly

- inb $a, 0x64 ← kbd status register
- outb $a, 0x60 ← kbd data register
- Specifies: device, data, direction
- $\boxed{\text{Protection: only allowed in kernel mode}}$

Kernel boundary crossinging is expensive

*x86: $a implicit; also inw, outw, inh, outh, …

Communication Interface

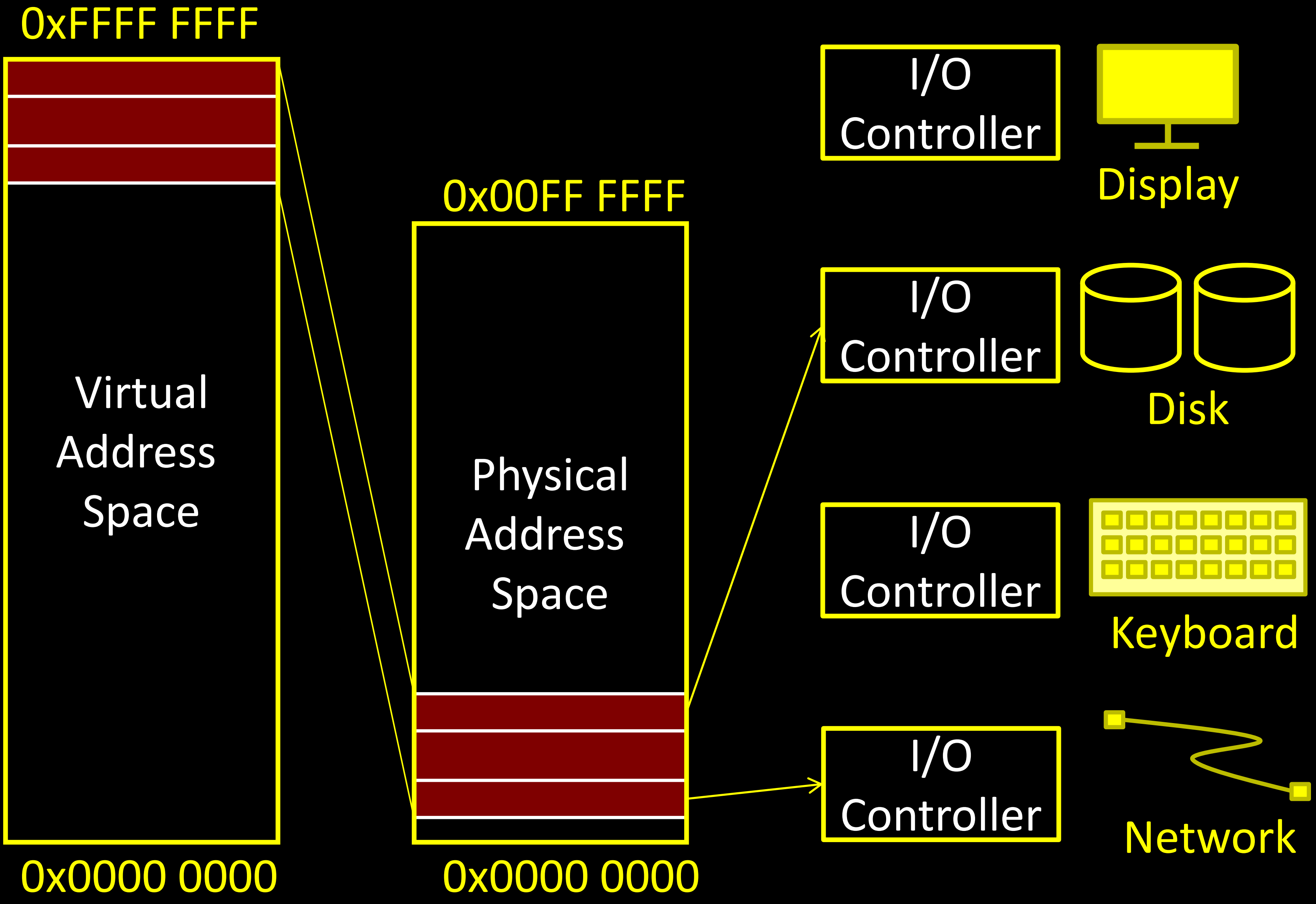
Q: How does ~~program~~ ~~OS~~ code talk to device?

A: Map registers into virtual address space

Memory-mapped I/O $\longleftarrow$ Faster. Less boundary crossing

- Accesses to certain addresses redirected to I/O devices
- Data goes over the memory bus
- Protection: via bits in pagetable entries
- OS+MMU+devices configure mappings

Memory-Mapped I/O



Device Drivers

Programmed I/O

Polling examples, But mmap I/O more efficient

```
char read_kbd()
{
do {
     sleep();
     status = inb(0x64);
  } while(!(status & 1));

  return inb(0x60);
}
```

syscall

Memory Mapped I/O

```
struct kbd {
   char status, pad[3];
   char data, pad[3];
};
kbd *k = mmap(...);
```

syscall

```
char read_kbd()
{
   do {
      sleep();
      status = k->status;
   } while(!(status & 1));
   return k->data;
}
```

NO syscall

Comparing Programmed I/O vs Memory Mapped I/O

Programmed I/O

- Requires special instructions
- Can require dedicated hardware interface to devices
- Protection enforced via kernel mode access to instructions
- Virtualization can be difficult

Memory-Mapped I/O

- Re-uses standard load/store instructions
- Re-uses standard memory hardware interface
- Protection enforced with normal memory protection scheme
- Virtualization enabled with normal memory virtualization scheme

Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.

Next Goal

How does the processor know device is ready/done?

Communication Method

Q: How does program learn device is ready/done?

A: Polling: Periodically check I/O status register

- If device ready, do operation
- If device done, …
- If error, take action

```
char read_kbd()
{
do {
    sleep();
    status = inb(0x64);
  } while(!(status & 1));

  return inb(0x60);  }
```

Pro? Con?

- Predictable timing & inexpensive
- But: wastes CPU cycles if nothing to do
- Efficient if there is always work to do (e.g. 10Gbps NIC)

Common in small, cheap, or real-time embedded systems

Sometimes for very active devices too…

Communication Method

Q: How does program learn device is ready/done?

A: Interrupts: Device sends interrupt to CPU

- Cause register identifies the interrupting device
- interrupt handler examines device, decides what to do

Priority interrupts

- Urgent events can interrupt lower-priority interrupt handling
- OS can ~~disable~~ defer interrupts

Pro? Con?

- More efficient: only interrupt when device ready/done
- Less efficient: more expensive since save CPU context
  - CPU context: PC, SP, registers, etc
- Con: unpredictable b/c event arrival depends on other devices' activity

Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.

Interrupt-based I/O avoids the wasted work in polling-based I/O and is usually more efficient

Next Goal

How do we transfer a ***lot*** of data ***efficiently***?

I/O Data Transfer

How to talk to device?

- Programmed I/O or Memory-Mapped I/O

How to get events?

- Polling or Interrupts

How to transfer lots of data?

```
disk->cmd = READ_4K_SECTOR;
disk->data = 12;
while (!(disk->status & 1) { }
for (i = 0..4k)
 buf[i] = disk->data;
```

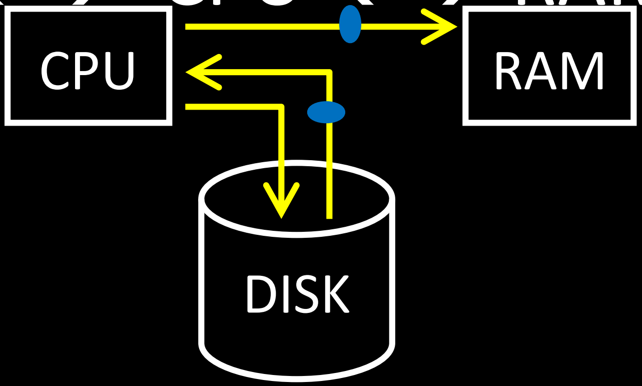
Very Expensive

I/O Data Transfer

Programmed I/O xfer: Device ←→ CPU ←→ RAM

for (i = 1 .. n)

- CPU issues read request
- Device puts data on bus & CPU reads into registers
- CPU writes data to memory
- ***Not*** efficient



Read from Disk
Write to Memory
Everything interrupts CPU
Wastes CPU

I/O Data Transfer

Q: How to transfer lots of data ***efficiently***?

A: Have device access memory directly

Direct memory access (DMA)

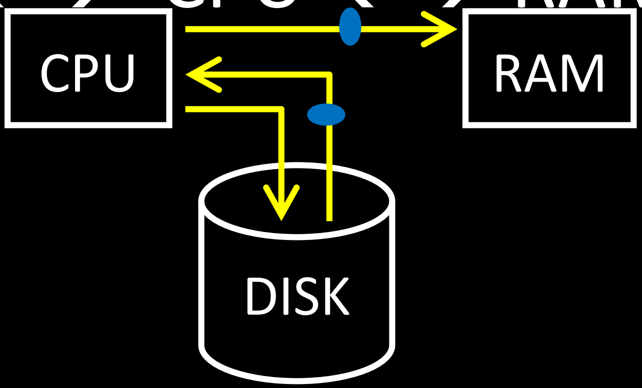
- 1) OS provides starting address, length
- 2) controller (or device) transfers data autonomously
- 3) Interrupt on completion / error

DMA: Direct Memory Access

Programmed I/O xfer: Device ←→ CPU ←→ RAM

for (i = 1 .. n)

- CPU issues read request
- Device puts data on bus & CPU reads into registers
- CPU writes data to memory

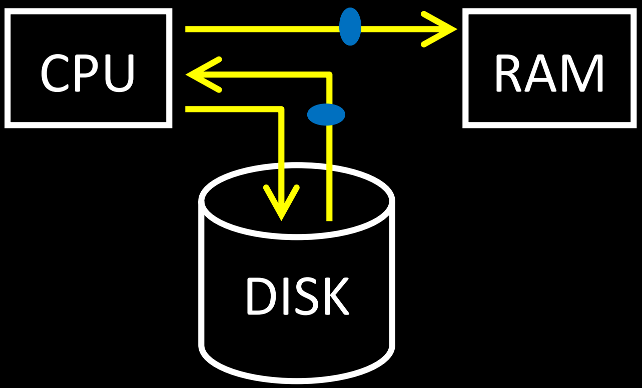


DMA: Direct Memory Access

Programmed I/O xfer: Device ←→ CPU ←→ RAM

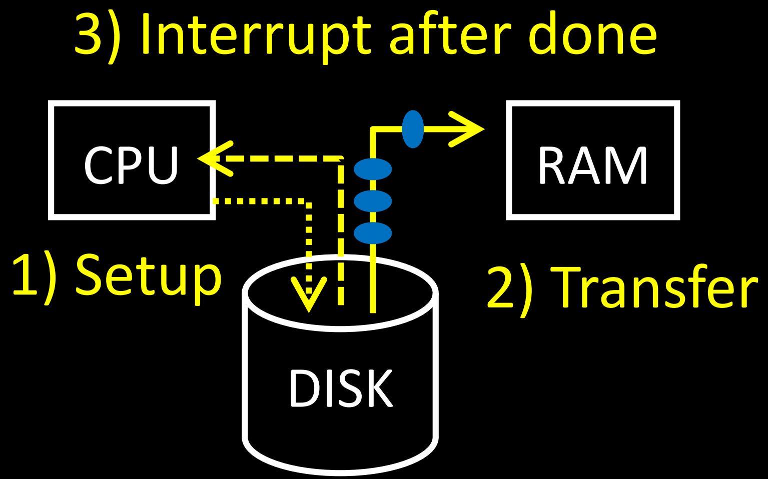
for (i = 1 .. n)

- CPU issues read request
- Device puts data on bus & CPU reads into registers
- CPU writes data to memory



DMA xfer: Device ←→ RAM

- CPU sets up DMA request
- for (i = 1 ... n)
  Device puts data on bus
  & RAM accepts it
- Device interrupts CPU after done



DMA Example

DMA example: reading from audio (mic) input

- DMA engine on audio device… or I/O controller … or …

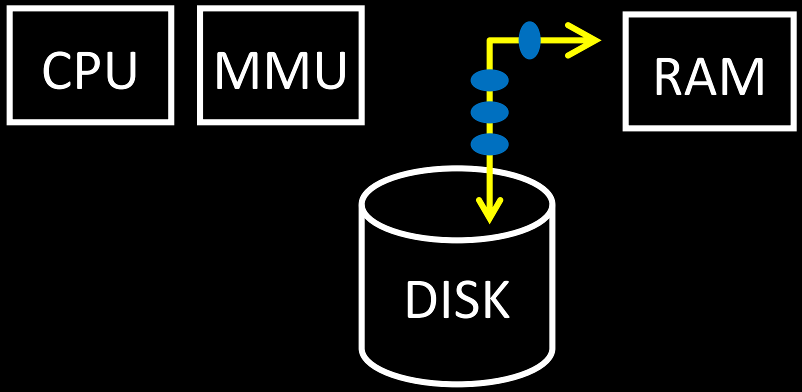
```
int dma_size = 4*PAGE_SIZE;
int *buf = alloc_dma(dma_size);
...
dev->mic_dma_baseaddr = (int)buf;
dev->mic_dma_count = dma_len;
dev->cmd = DEV_MIC_INPUT |
DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
```

DMA Issues (1): Addressing

Issue #1: DMA meets Virtual Memory

RAM: physical addresses

Programs: virtual addresses



Solution: DMA uses physical addresses

- OS uses physical address when setting up DMA
- OS allocates contiguous physical pages for DMA
- Or: OS splits xfer into page-sized chunks

  (many devices support DMA “chains” for this reason)

DMA Example

DMA example: reading from audio (mic) input

- DMA engine on audio device… or I/O controller … or …

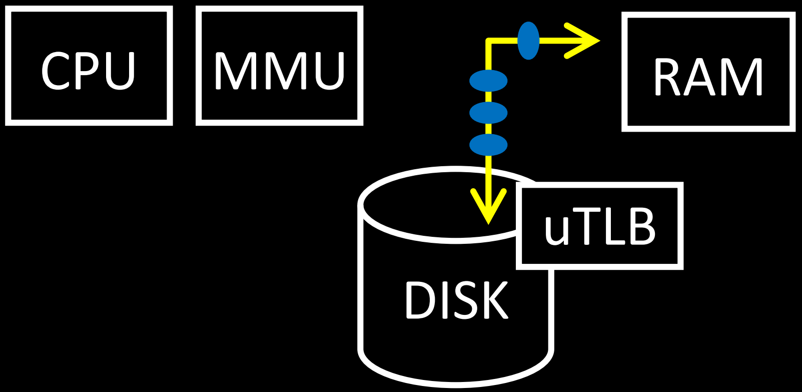
```
int dma_size = 4*PAGE_SIZE;
void *buf = alloc_dma(dma_size);
...
dev->mic_dma_baseaddr = virt_to_phys(buf);
dev->mic_dma_count = dma_len;
dev->cmd = DEV_MIC_INPUT |
DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
```

DMA Issues (1): Addressing

Issue #1: DMA meets Virtual Memory

RAM: physical addresses

Programs: virtual addresses



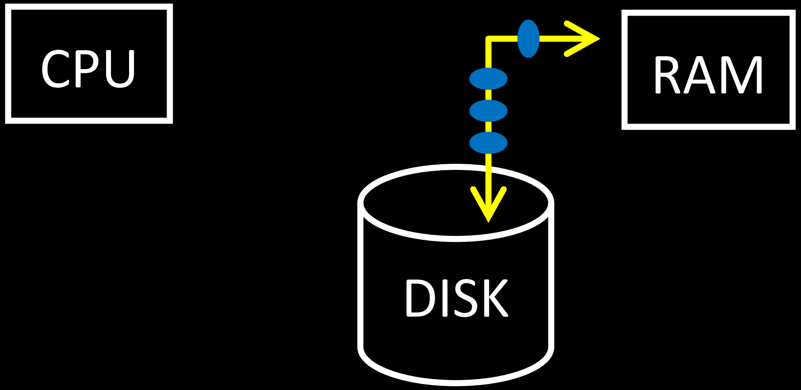
Solution 2: DMA uses virtual addresses

- OS sets up mappings on a mini-TLB

DMA Issues (2): Virtual Mem

Issue #2: DMA meets *Paged* Virtual Memory

DMA destination page may get swapped out



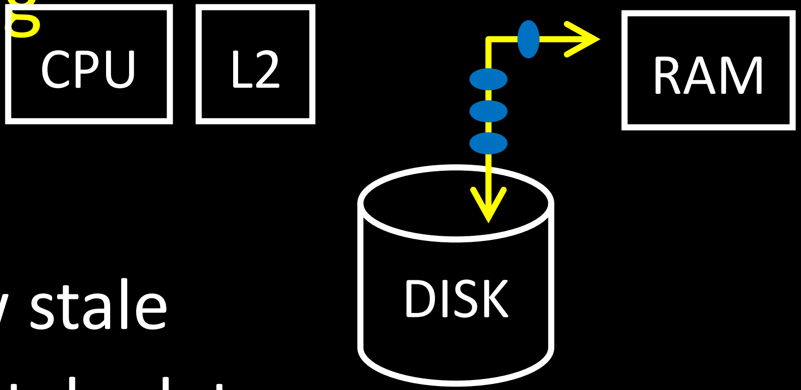
Solution: Pin the page before initiating DMA

Alternate solution: Bounce Buffer

- DMA to a pinned kernel page, then memcpy elsewhere

DMA Issues (4): Caches

Issue #4: DMA meets Caching



DMA-related data could be cached in L1/L2

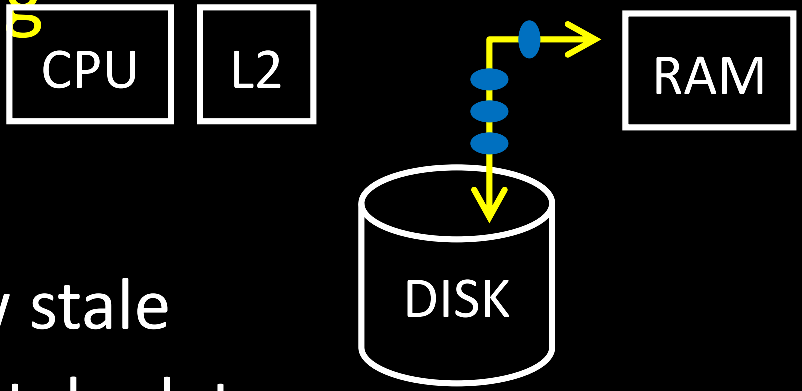
- DMA to Mem: cache is now stale
- DMA from Mem: dev gets stale data

Solution: (software enforced coherence)

- OS flushes some/all cache before DMA begins
- Or: don't touch pages during DMA
- Or: mark pages as uncacheable in page table entries
    – (needed for Memory Mapped I/O too!)

DMA Issues (4): Caches

Issue #4: DMA meets Caching



DMA-related data could be cached in L1/L2

- DMA to Mem: cache is now stale
- DMA from Mem: dev gets stale data

Solution 2: (hardware coherence aka snooping)

- cache listens on bus, and conspires with RAM
- DMA to Mem: invalidate/update data seen on bus
- DMA from mem: cache services request if possible, otherwise RAM services

Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.

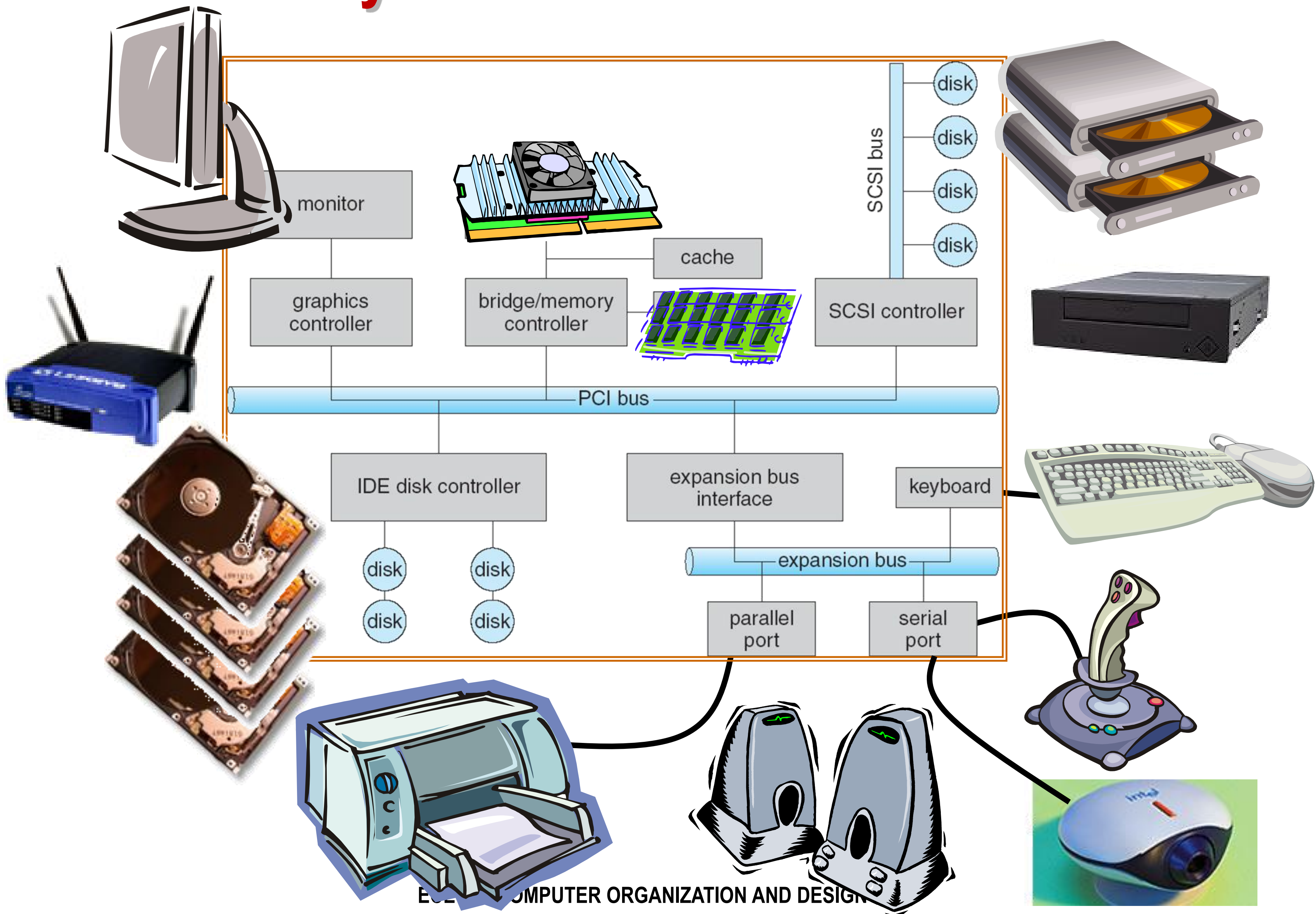
Interrupt-based I/O avoids the wasted work in polling-based I/O and is usually more efficient.

Modern systems combine memory-mapped I/O, interrupt-based I/O, and direct-memory access to create sophisticated I/O device subsystems.

Announcement

- today's lecture: I/O subsystem
  - ✓ Ch. 5.6 – 5.7 (HP1)
- no lecture on 11/21 and 11/23
- next lecture on 11/30
  - ✓ Impact of technology – handout
- MP assignment
  - ✓ MP3 check-point 4 due on 11/26 5pm

Modern I/O Systems

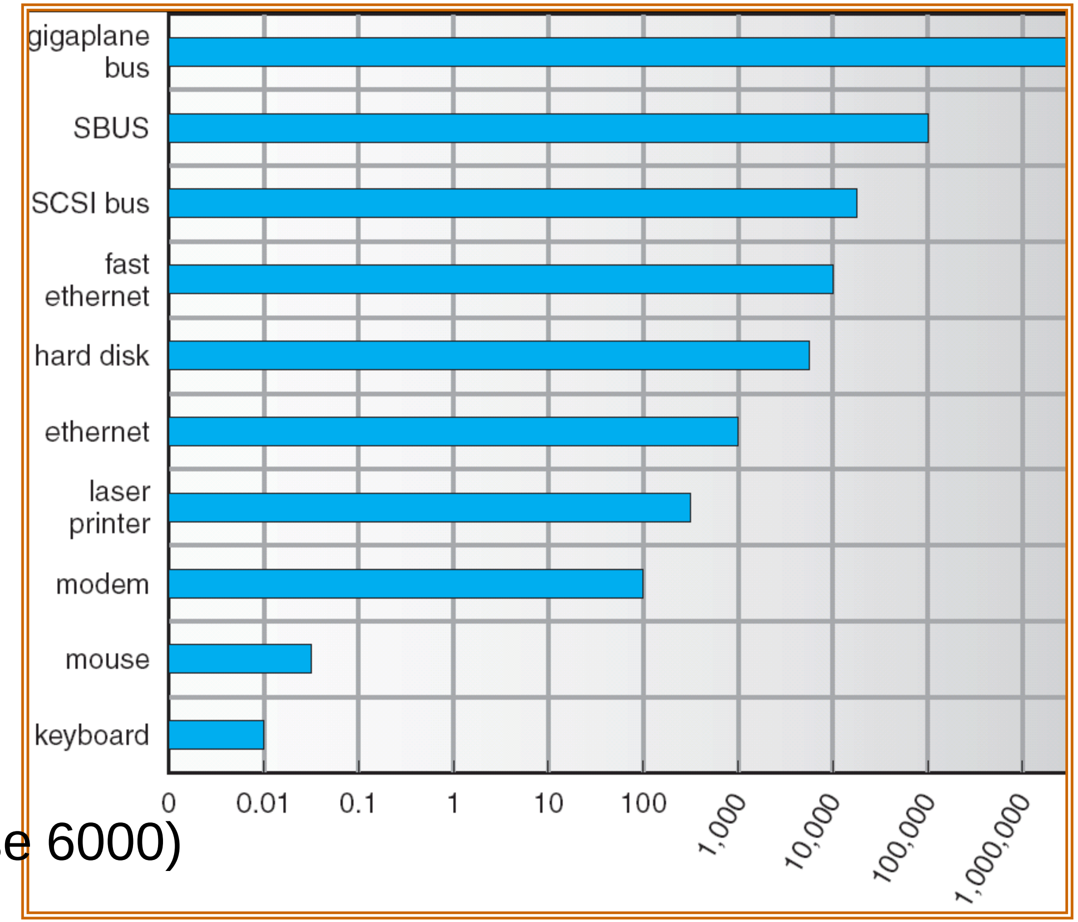


Requirement of I/O

- computers are useless (disembodied brains?) w/o I/O
  - ✓ 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 to manage them if we don't know what they'll do or how they'll 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

Example Device-Transfer Rates

- 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



(Sun Enterprise 6000)

Goal of I/O Subsystem

- provide uniform interfaces, despite wide range of different devices
  - ✓ this code works on many different devices:

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

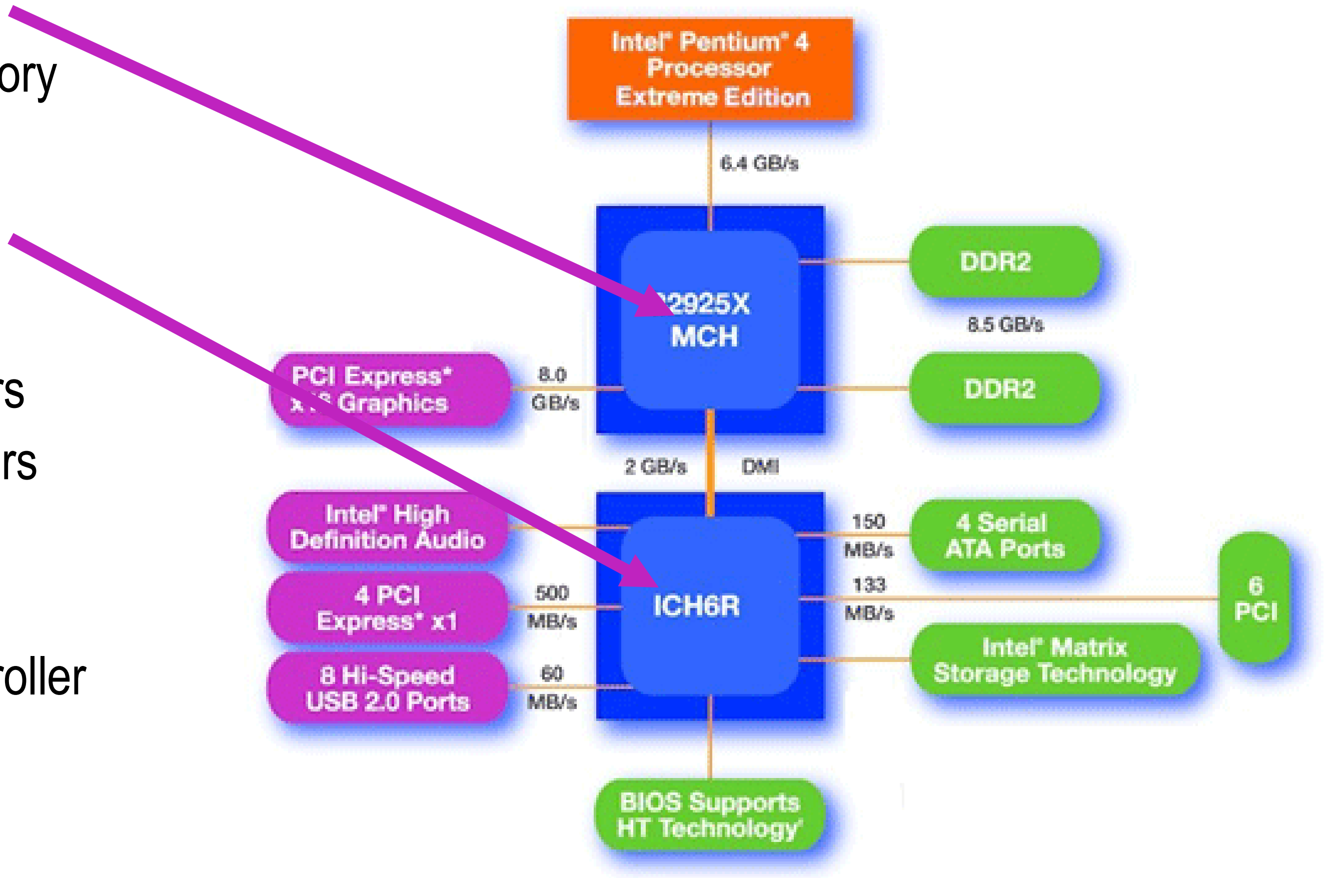
  - ✓ why? b/c code that controls devices ("device driver") implements standard interface
- we will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  - ✓ can only scratch surface!

How Does User Deal w/ Timing?

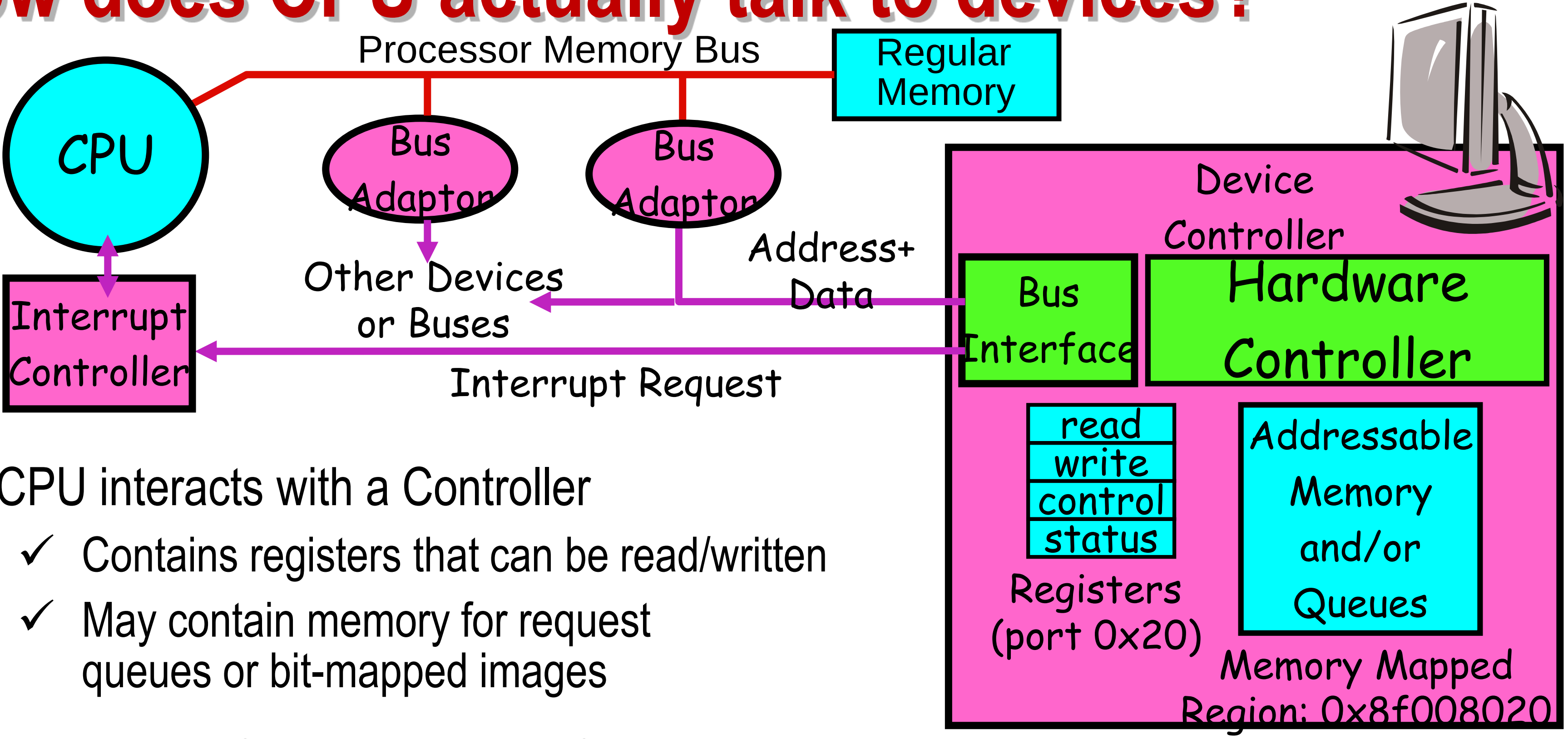
- block devices: e.g. disk drives, tape drives, cdrom
  - ✓ access blocks of data
  - ✓ commands include open(), read(), write(), seek()
  - ✓ raw I/O or file-system access
  - ✓ memory-mapped file access possible
- character devices: e.g. keyboards, mice, serial ports, some USB devices
  - ✓ single 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

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


Intel® Pentium® 4 Processor Extreme Edition
6.4 GB/s
925X MCH
DDR2
8.5 GB/s
DDR2
PCI Express* x16 Graphics
8.0 GB/s
2 GB/s
DMI
Intel® High Definition Audio
4 PCI Express* x1
500 MB/s
8 Hi-Speed USB 2.0 Ports
60 MB/s
ICH6R
150 MB/s
4 Serial ATA Ports
133 MB/s
6 PCI
Intel® Matrix Storage Technology
BIOS Supports HT Technology*

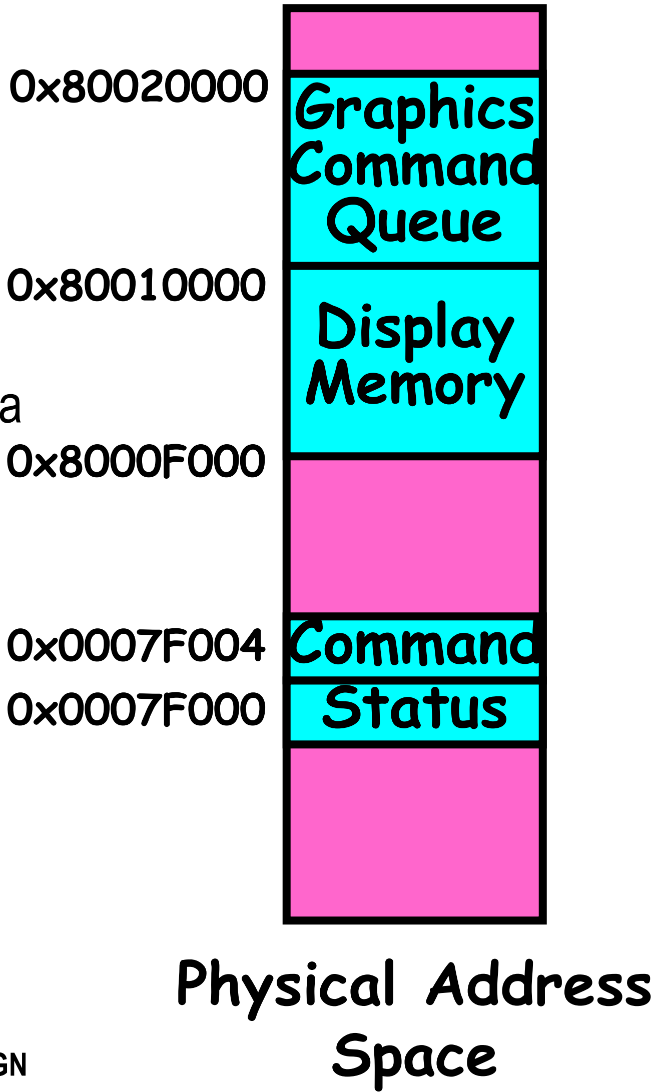
How does CPU actually talk to devices?



- CPU interacts with a Controller
  - ✓ Contains registers that can be read/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

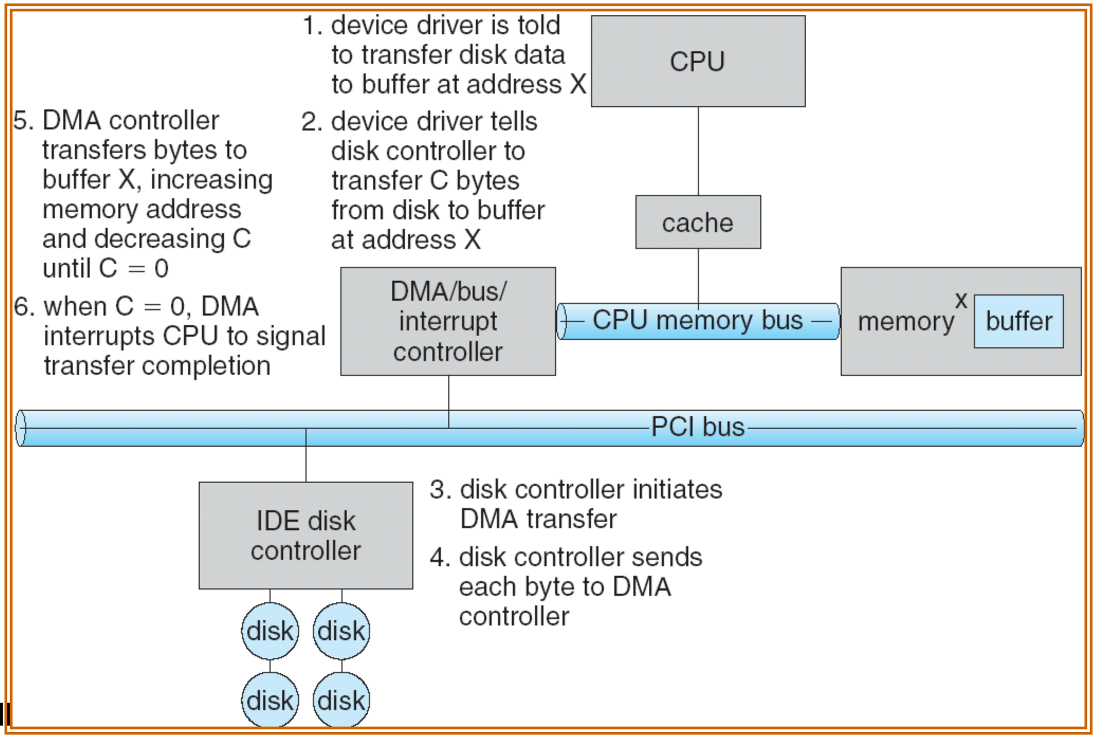
Memory-Mapped Display Controller

- 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 w/ page tables



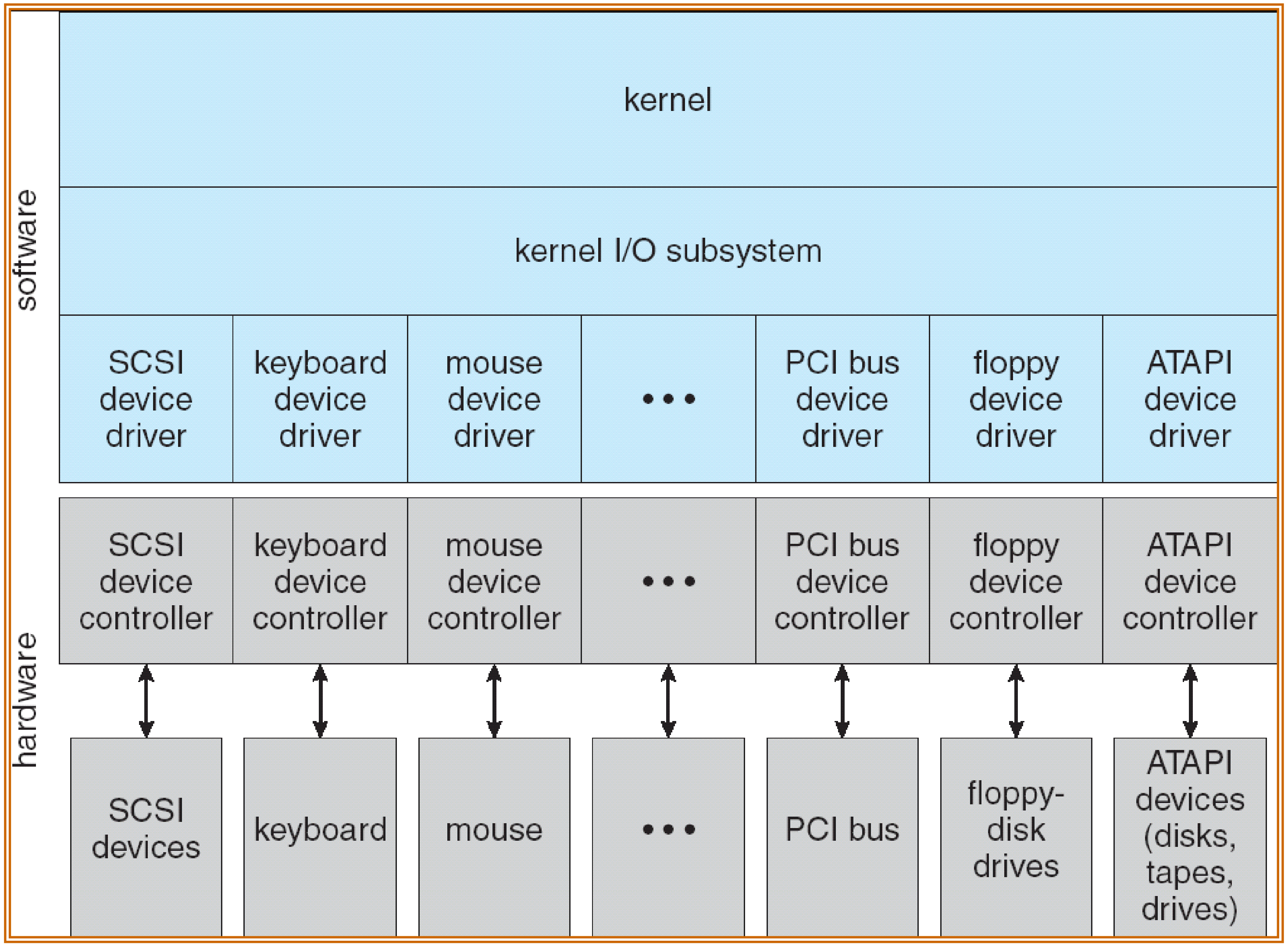
Transferring Data to/from Controller

- 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 w/ DMA controller (from book):



1. device driver is told to transfer disk data to buffer at address X
CPU
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
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

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

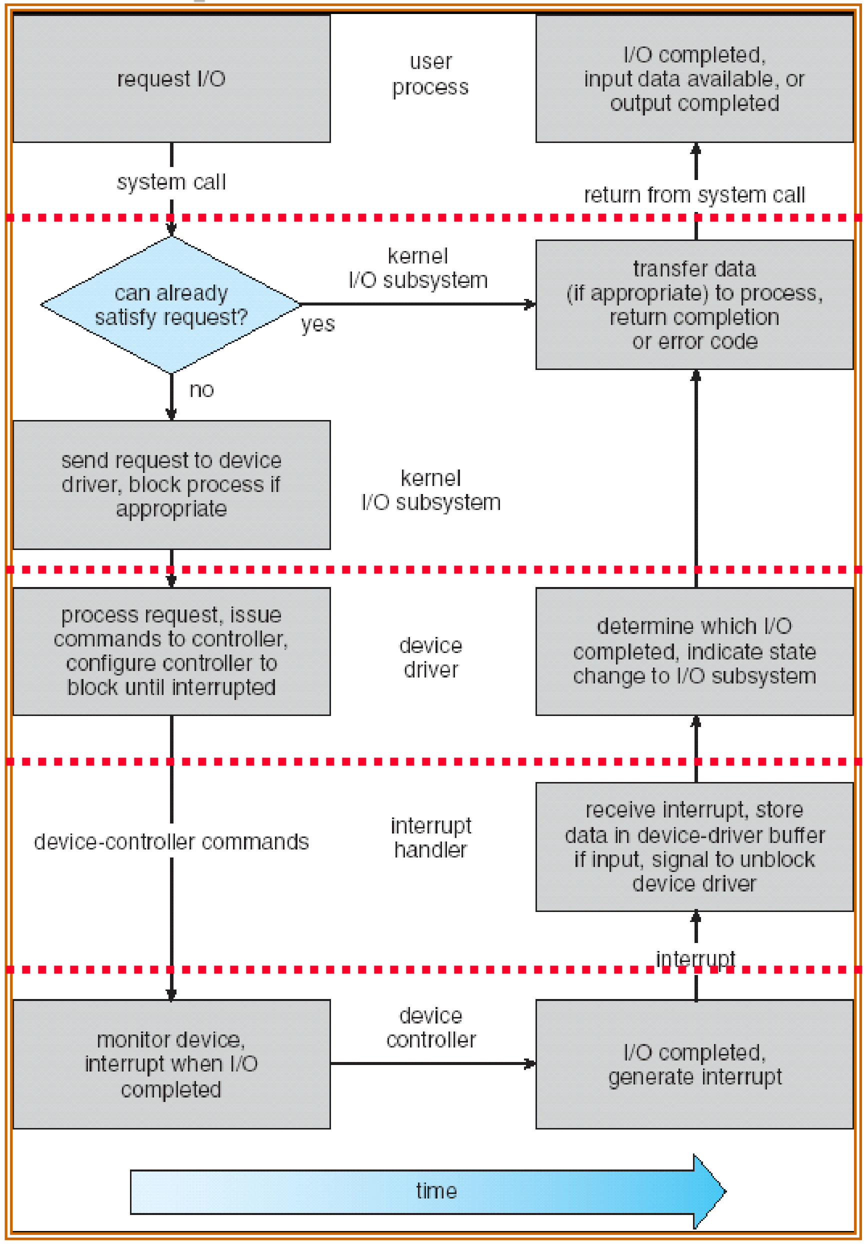
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
kernel I/O subsystem
yes
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

I/O Device Notifying the OS

- OS needs to know when I/O device has completed an operation
- I/O interrupt:
  - ✓ device generates an interrupt whenever it needs service
  - ✓ handled in bottom half of device driver by 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 ops
- 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