I/O

Hakim Weatherspoon
CS 3410, Spring 2011
Computer Science
Cornell University

See: P&H Chapter 6.5-6

Goals for Today

Computer System Organization

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)

Computer System Organization

Computer System =

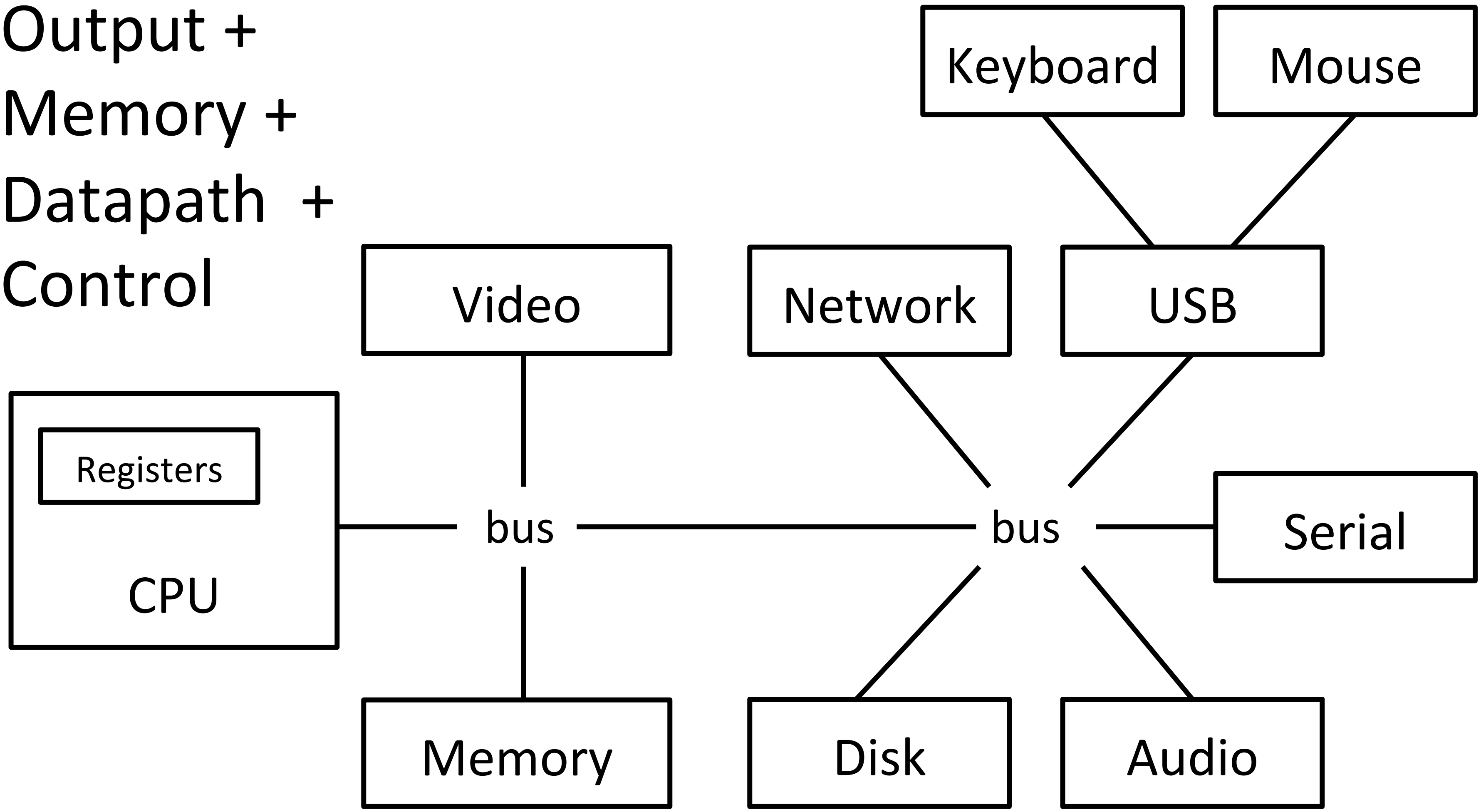
Input +

Output +

Memory +

Datapath +

Control



Challenge

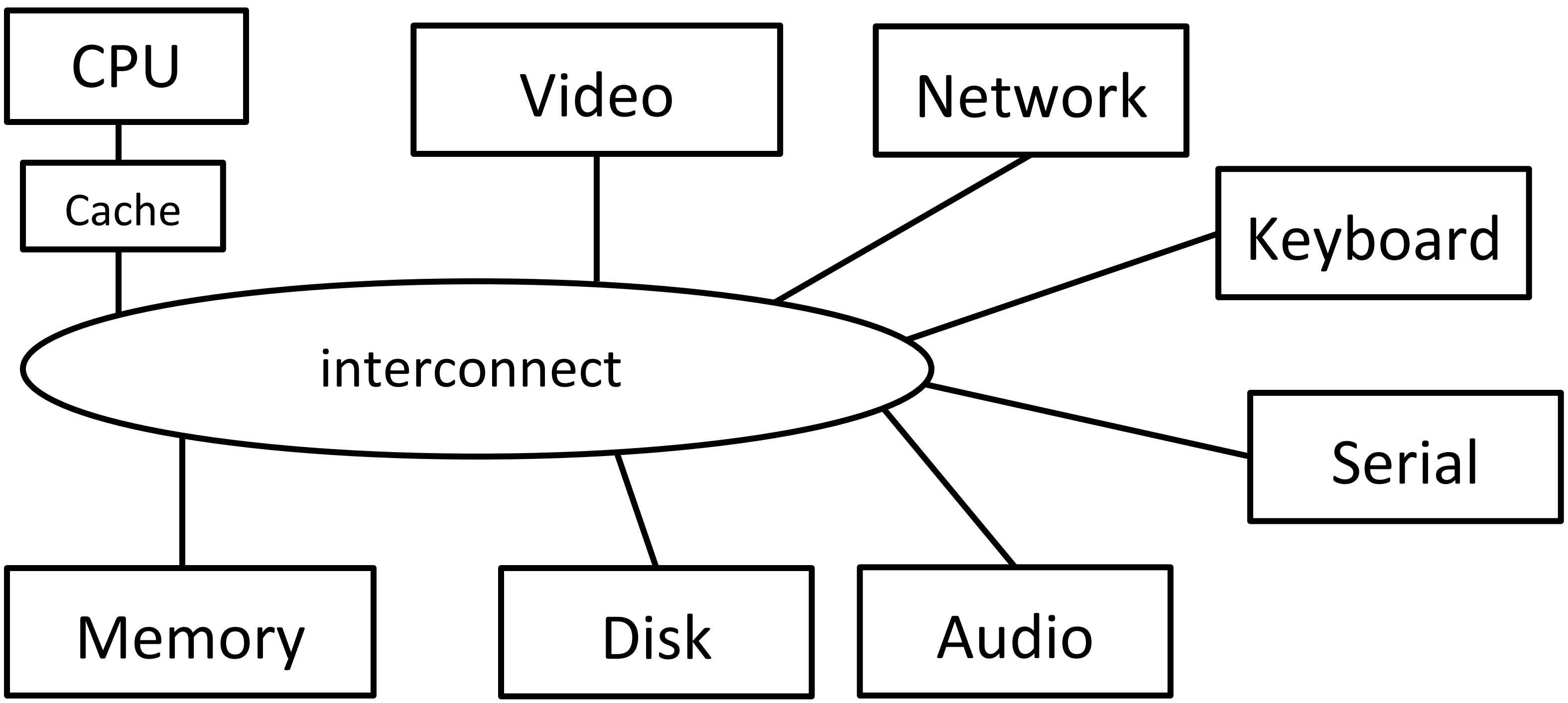
How do we interface to other devices

- Keyboard
- Mouse
- Disk
- Network
- Display
- Programmable Timer (for clock ticks)
- Audio
- Printer(s)
- Camera
- iPod
- Scanner
- …

Interconnects

Bad Idea #1: Put all devices on one interconnect

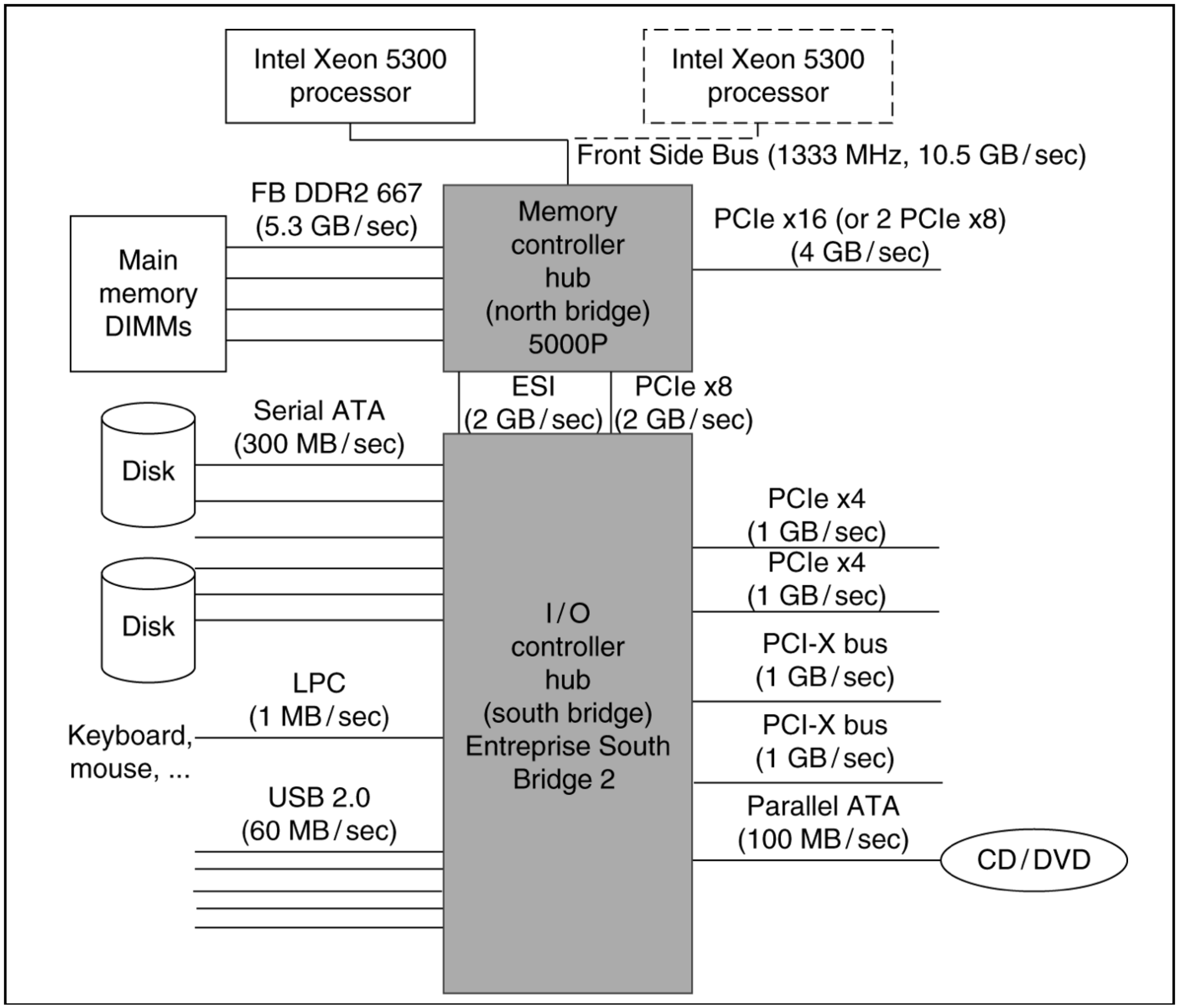
- We would have to replace all devices as we improve/ change the interconnect
- keyboard speed == main memory speed ?!



I/O Controllers

Decouple via I/O Controllers and "Bridges"

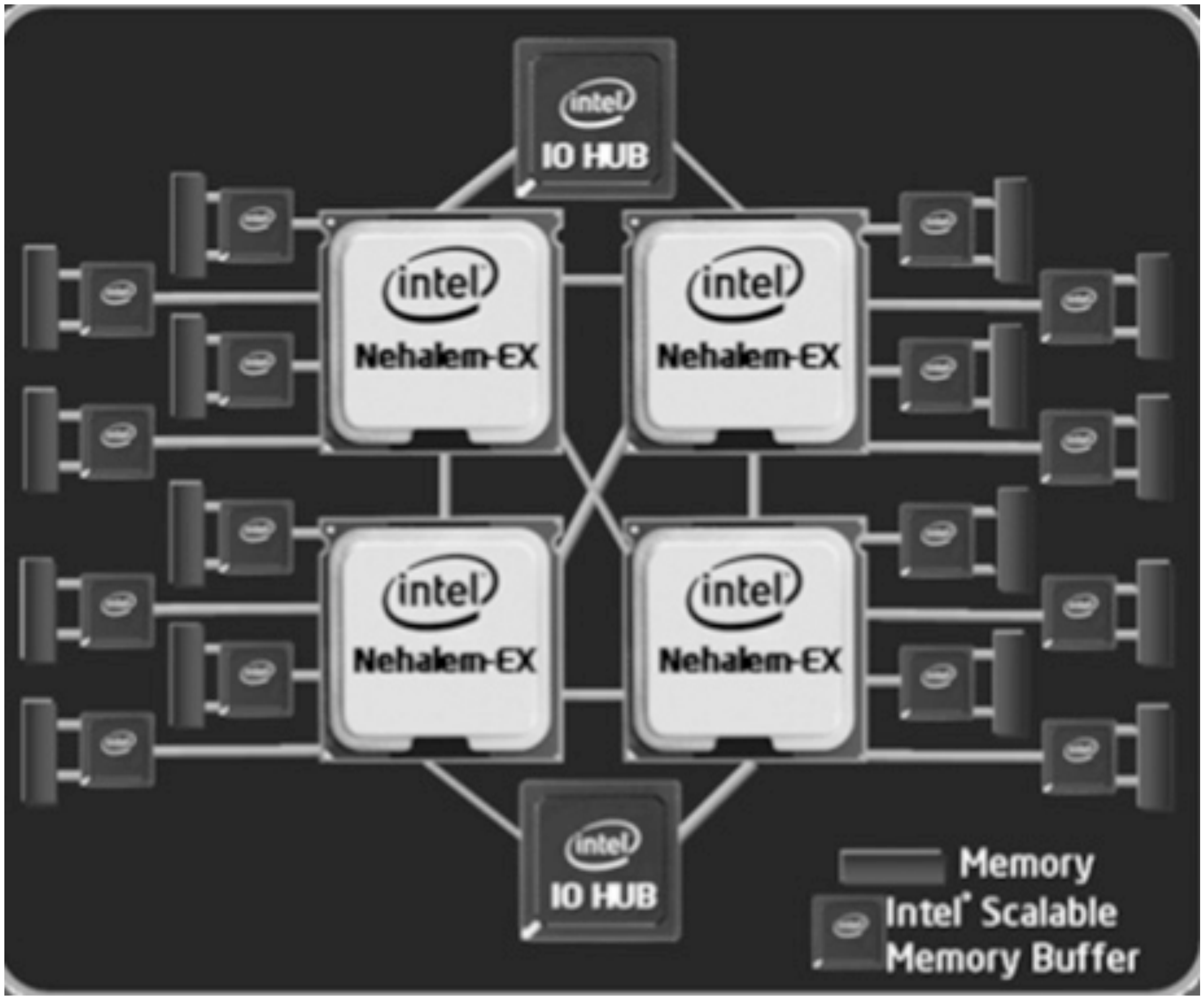
- fast/expensive busses when needed; slow/cheap elsewhere
- I/O controllers to connect end devices



I/O Controllers

Decouple via I/O Controllers and "Bridges"

- fast/expensive busses when needed; slow/cheap elsewhere
- I/O controllers to connect end devices



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

Alternative (and increasingly common):

- dedicated point-to-point channels

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

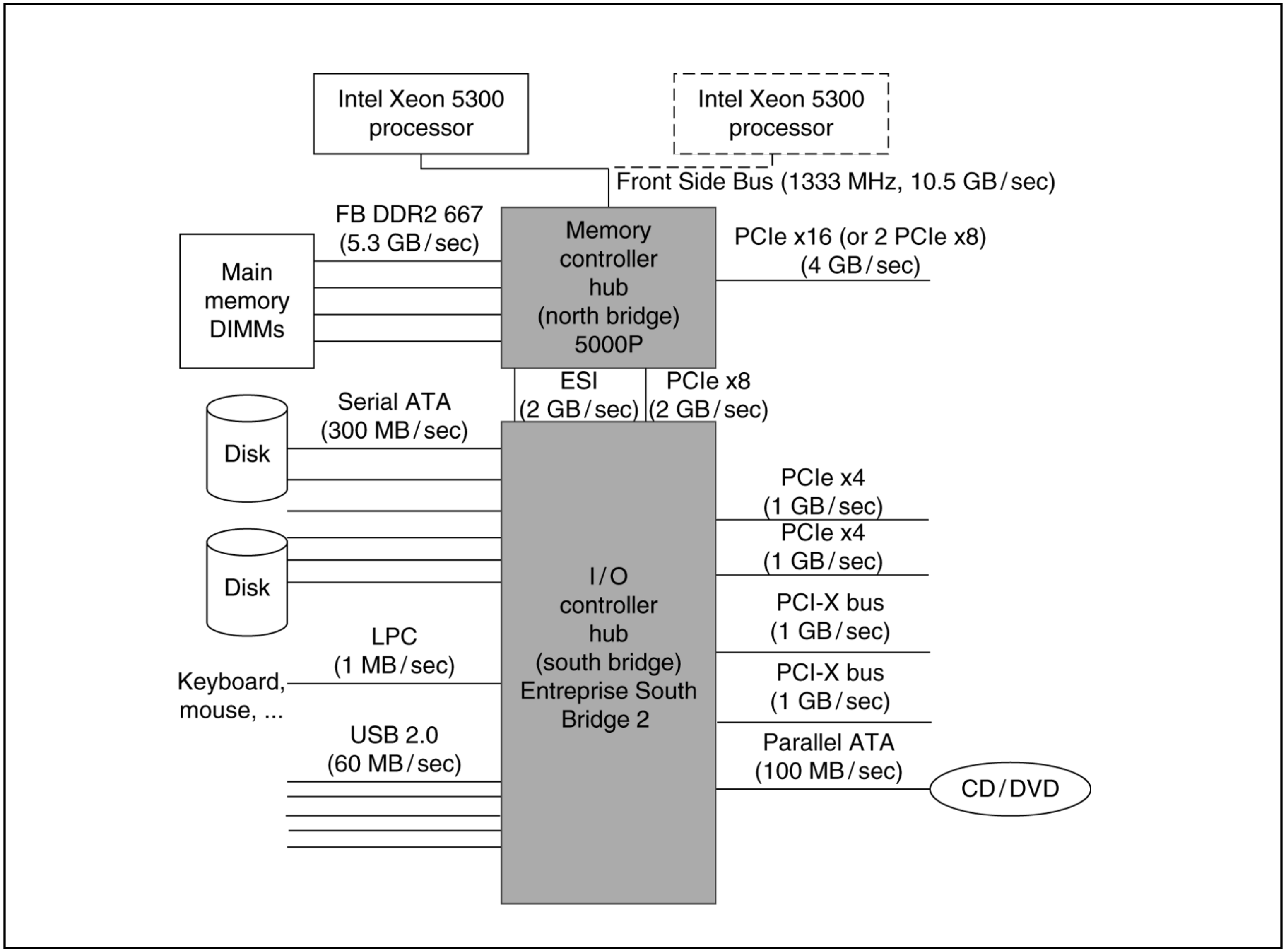
Processor – Memory ("Front Side Bus". Also QPI)

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

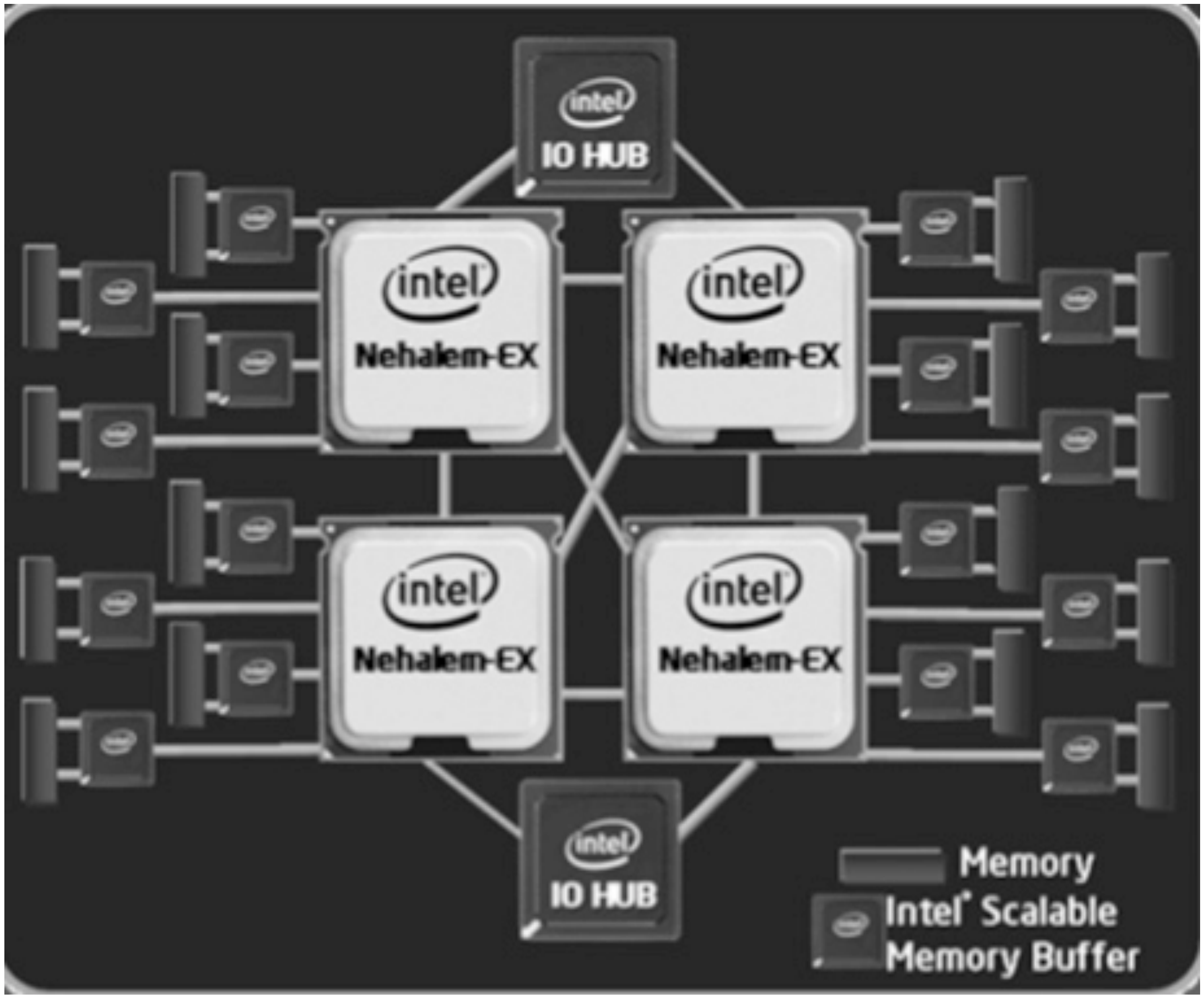
I/O and Peripheral busses (PCI, SCSI, USB, LPC, …)

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

Typical x86 PC I/O System



Typical x86 PC I/O System



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

Simple (old) example: AT Keyboard Device

8-bit Status:

PE	TO	AUXB	LOCK	AL2	SYSF	IBS	OBS

8-bit Cmd:

0xAA = “self test”

0xAE = “enable kbd”

0xED = “set LEDs”

…

8-bit Data:

scancode (when reading)

LED state (when writing) or …

Communication Interface

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

A: special instructions to talk over special busses

Programmed I/O

- inb $a, 0x64
- outb $a, 0x60
- Specifies: device, data, direction
- Protection: only allowed in kernel mode

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

Communication Interface

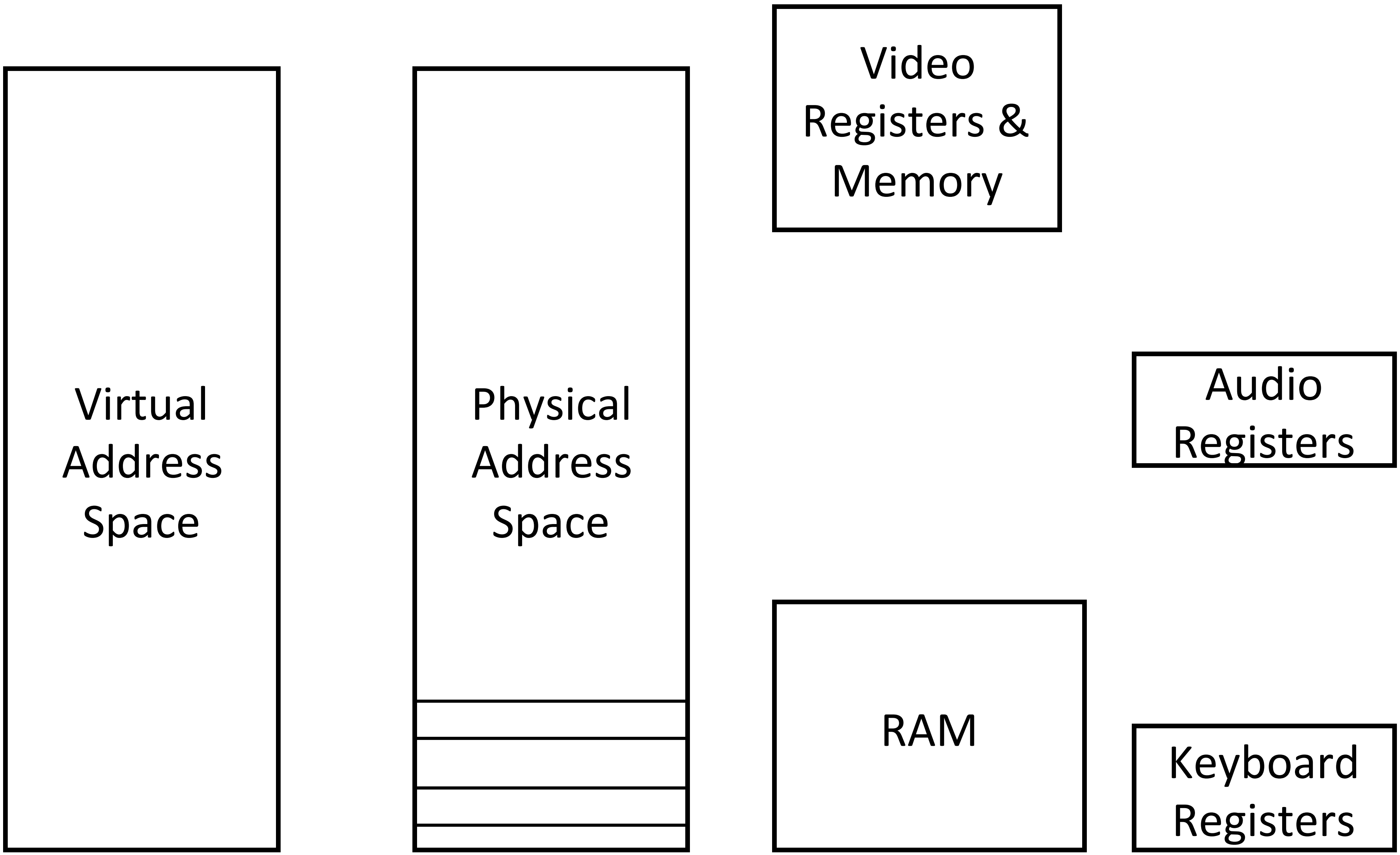
Q: How does ~~program~~ ~~OS~~ code talk to device?

A: Map registers into virtual address space

Memory-mapped I/O

- 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

```
char read_kbd()
{
do {
  sleep();
  status = inb(0x64);
} while (!(status & 1));
return inb(0x60);
}
```

Memory Mapped I/O

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

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

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

Pro? Con?

- Predictable timing & inexpensive
- But: wastes CPU cycles if nothing to do
- Efficient if there is always work to do

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?

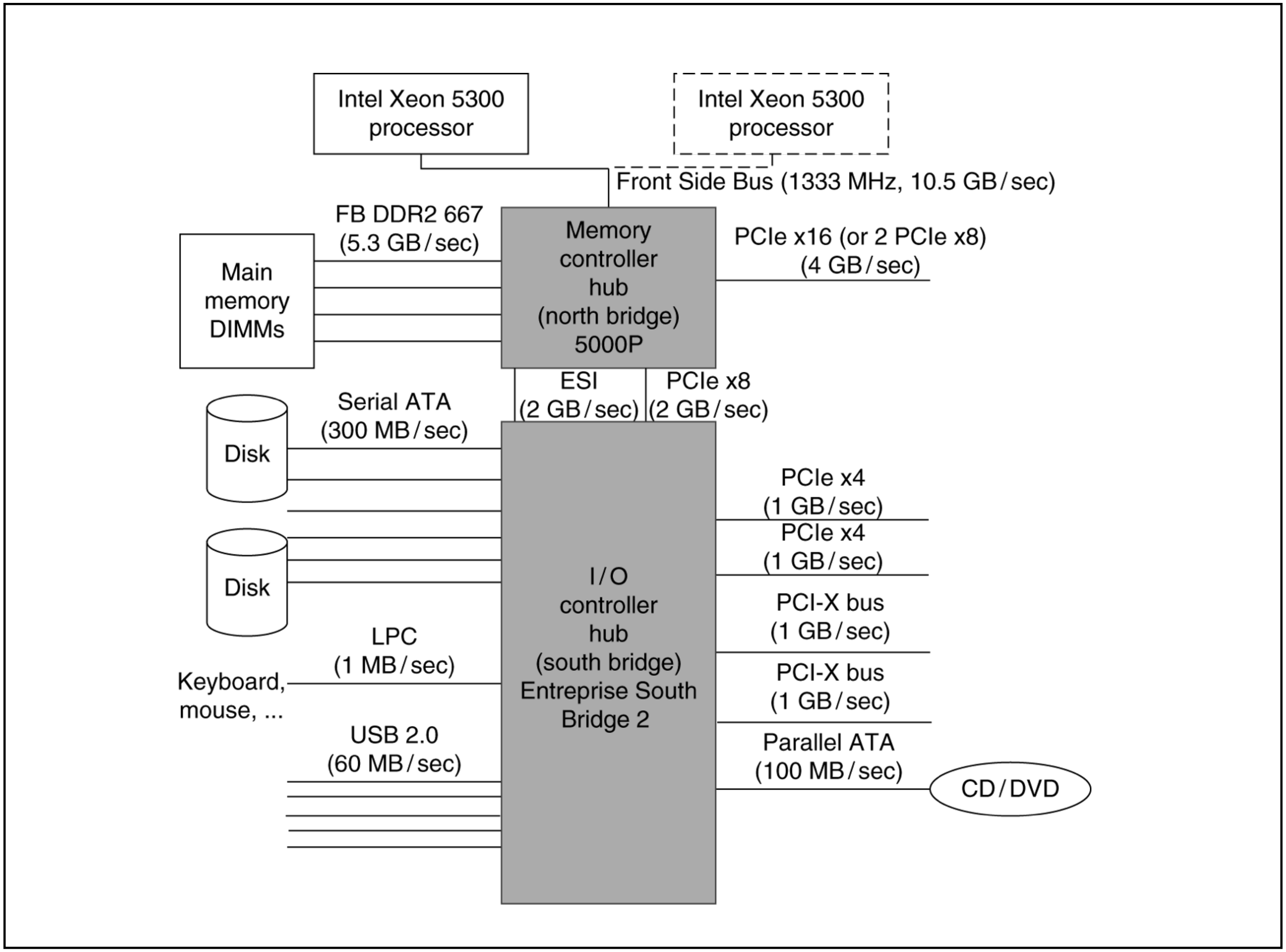
A: Interrupts: Device sends interrupt to CPU

- Cause 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

Typical x86 PC I/O System



I/O Data Transfer

How to talk to device?

  Programmed I/O or Memory-Mapped I/O

How to get events?

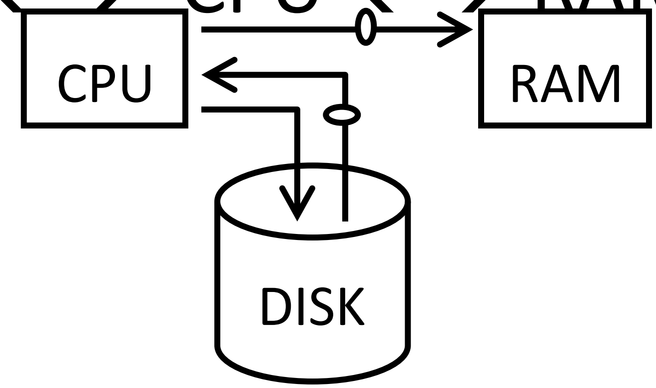
  Polling or Interrupts

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;
```

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

I/O Data Transfer

Q: How to transfer lots of data efficiently?

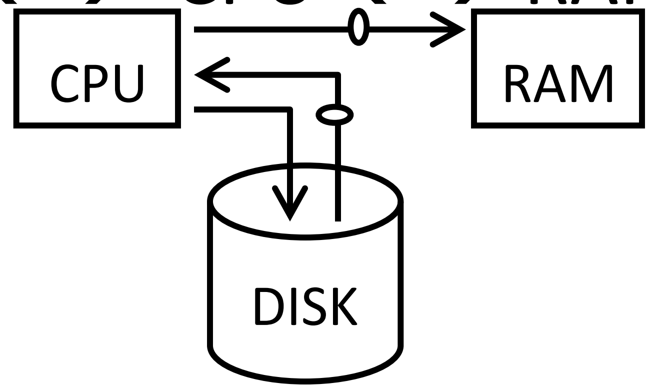
A: Have device access memory directly

Direct memory access (DMA)

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

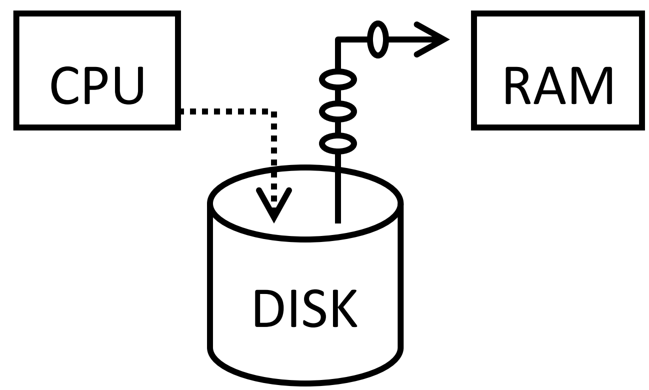
DMA: Direct Memory Access

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 xfer: Device ←→ RAM

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

DMA Example

DMA example: reading from audio (mic) input

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

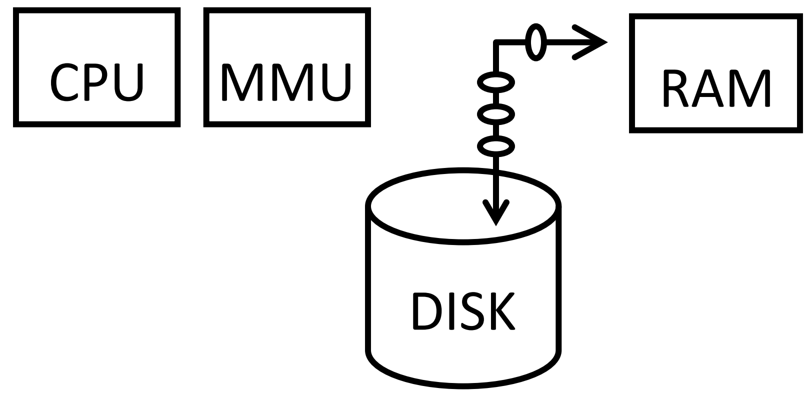
```
int dma_size = 4*PAGE_SIZE;
int *buf = alloc_dma(dma_size);
...
dev->mic_dma_baseaddr = (int)buf;
dev->mic_dma_count = dma_len;
dev->cmd = DEV_MIC_INPUT |
  DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
```

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 …

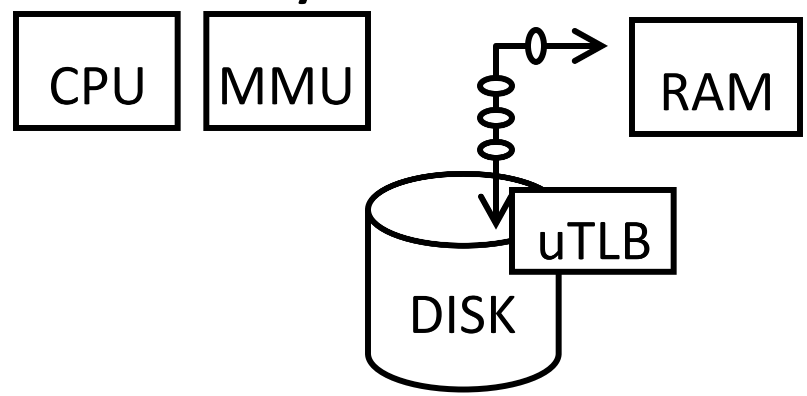
```
int dma_size = 4*PAGE_SIZE;
void *buf = alloc_dma(dma_size);
...
dev->mic_dma_baseaddr = virt_to_phys(buf);
dev->mic_dma_count = dma_len;
dev->cmd = DEV_MIC_INPUT |
  DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
```

DMA Issues (1): Addressing

Issue #1: DMA meets Virtual Memory

RAM: physical addresses

Programs: virtual addresses



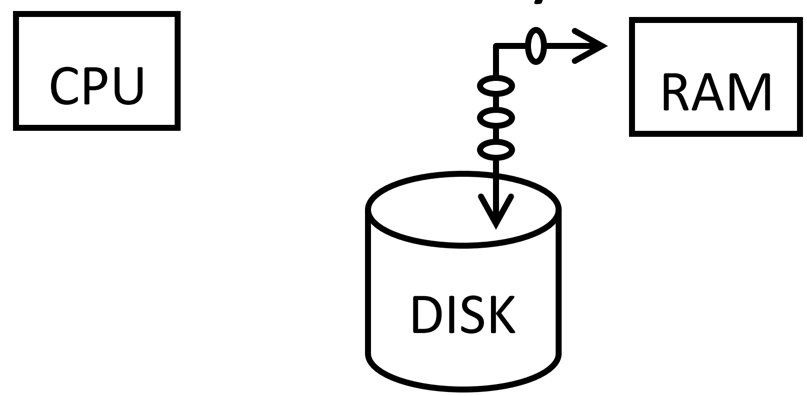
Solution 2: DMA uses virtual addresses

- 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

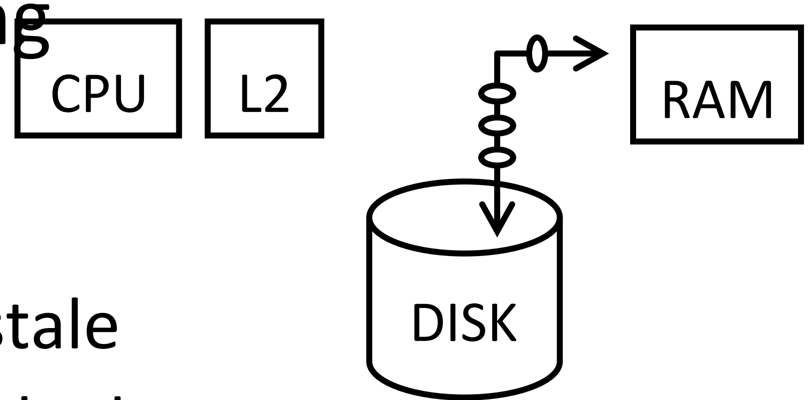


Solution: Pin the page before initiating DMA

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)

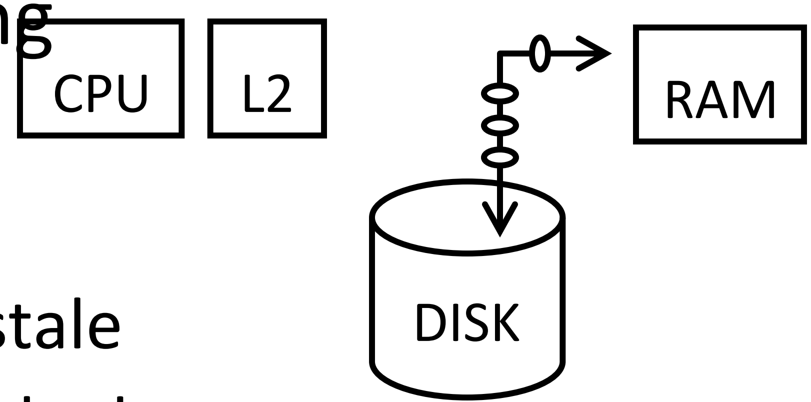
- OS flushes some/all cache before DMA begins
- Or: don't touch pages during DMA
- Or: mark pages as uncacheable in page table entries
  - (needed for Memory Mapped I/O too!)

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

I/O Summary

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?
- DMA