I/O

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

See: P&H Chapter 6.5-6

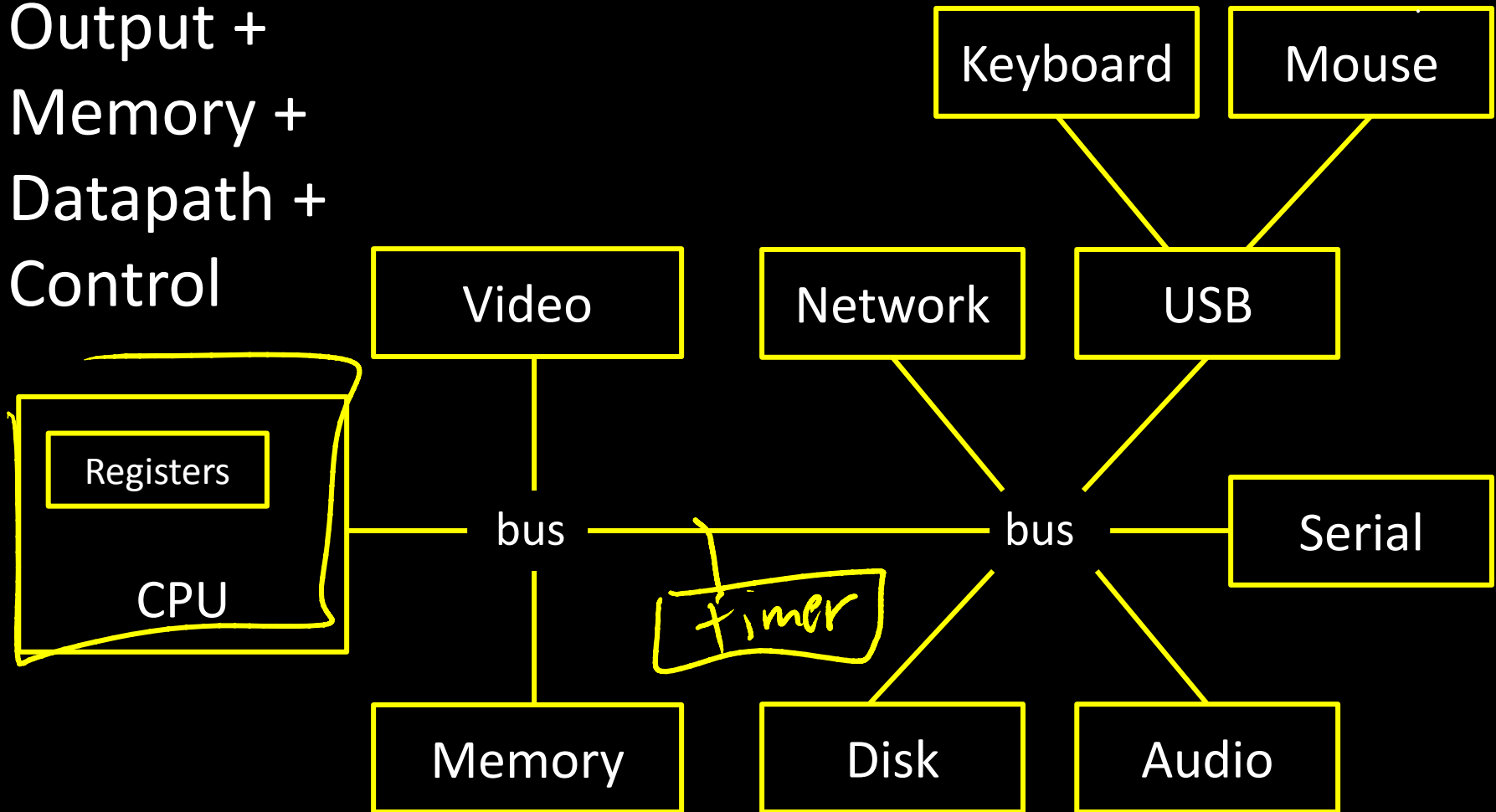Computer System =
Input +
Output +
Memory +
Datapath +
Control

Keyboard

Mouse

Video

Network

USB

Registers

bus

timer

bus

Serial

CPU

Memory

Disk

Audio

How do we interface to other devices

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

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 ?!

Decouple via I/O Controllers and "Bridges"
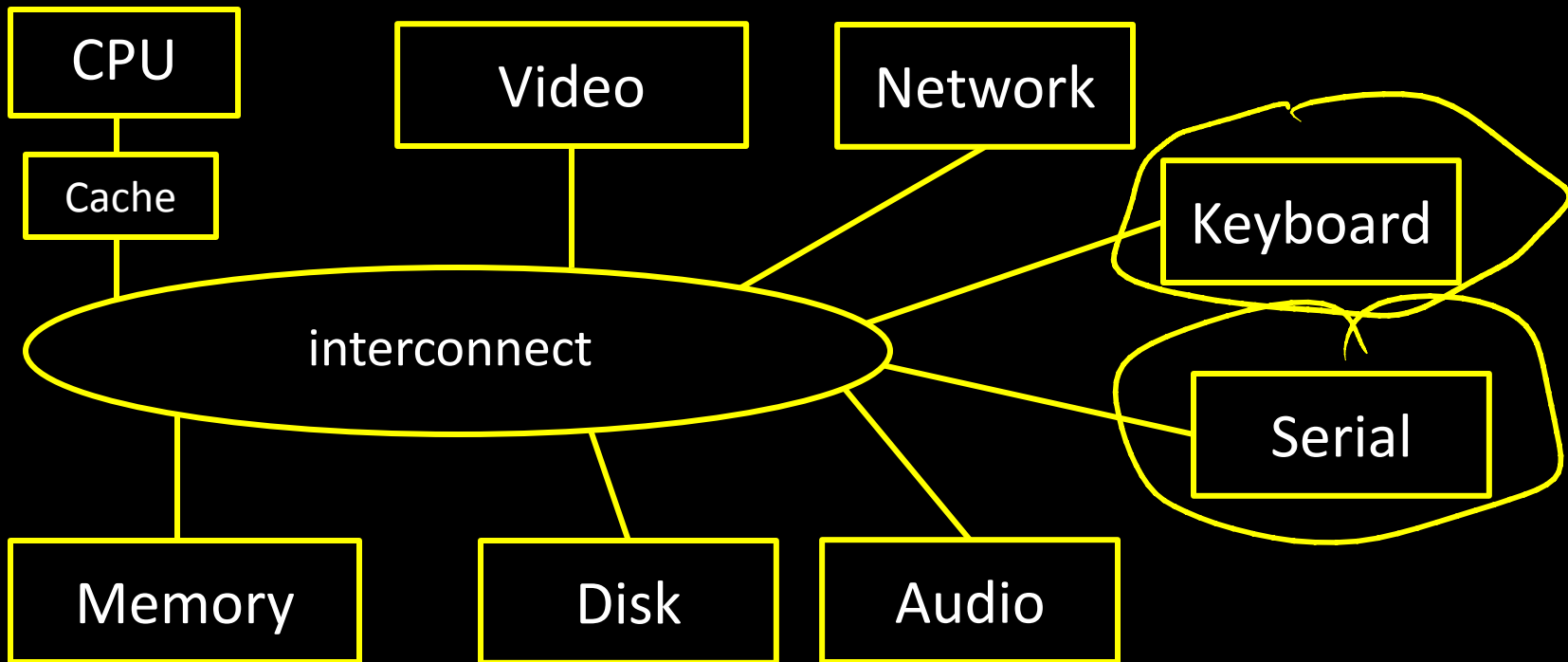
- fast/expensive busses when needed; slow/cheap elsewhere
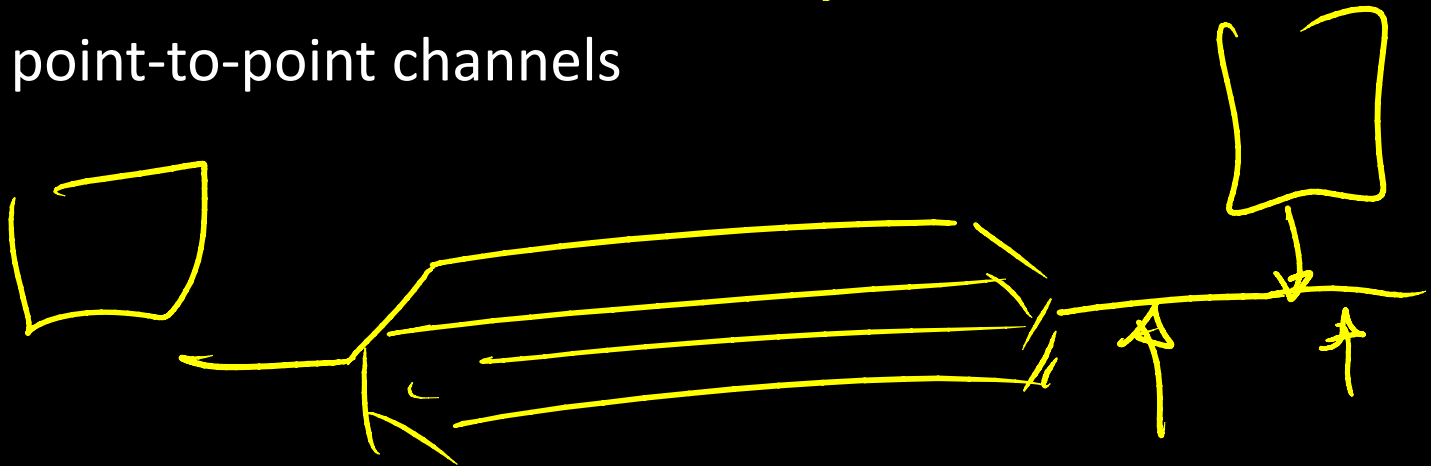- I/O controllers to connect end devices

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

Width = number of wires

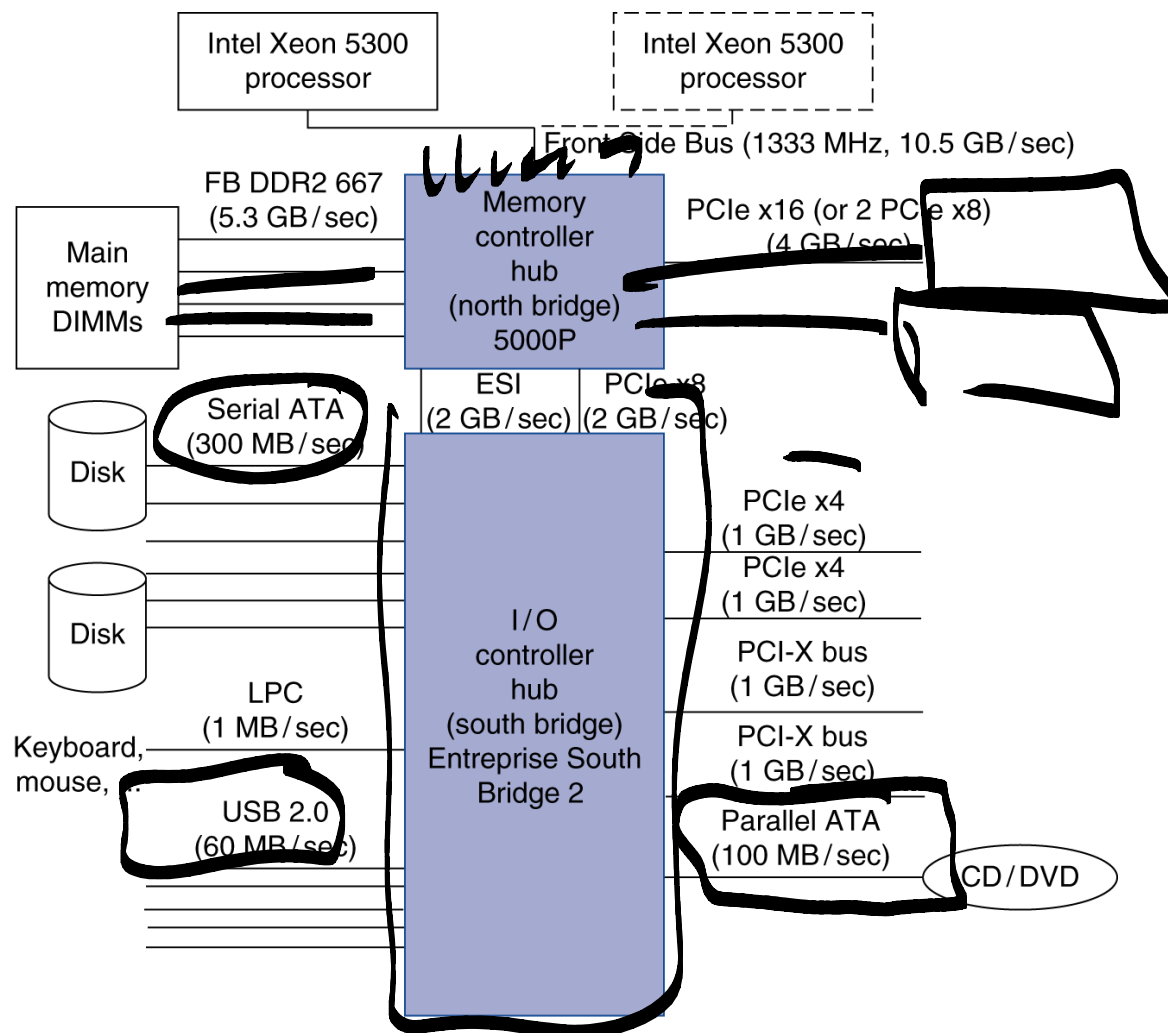Transfer size = data words per bus transaction

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

Processor – Memory ("Front Side Bus")

- 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

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

I/O
controller
hub
(south bridge)
Entreprise South
Bridge 2

PCIe x4
(1 GB/sec)

PCIe x4
(1 GB/sec)

PCI-X bus
(1 GB/sec)

PCI-X bus
(1 GB/sec)

LPC
(1 MB/sec)

Keyboard,
mouse, ..

USB 2.0
(60 MB/sec)

Parallel ATA
(100 MB/sec)

CD/DVD

9

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 ...

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

A: special instructions to talk over special busses

Programmed I/O

Handwritten annotation: Kbd Status Reg. (arrow to inb line), Kbd data Reg (arrow to outb line)

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

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

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

A: special instructions to talk over special busses

Programmed I/O

Kbd Stous Reg.
Kbd data Reg

- inb $a, 0x64

- outb $a, 0x60

- Specifies: device, data, direction

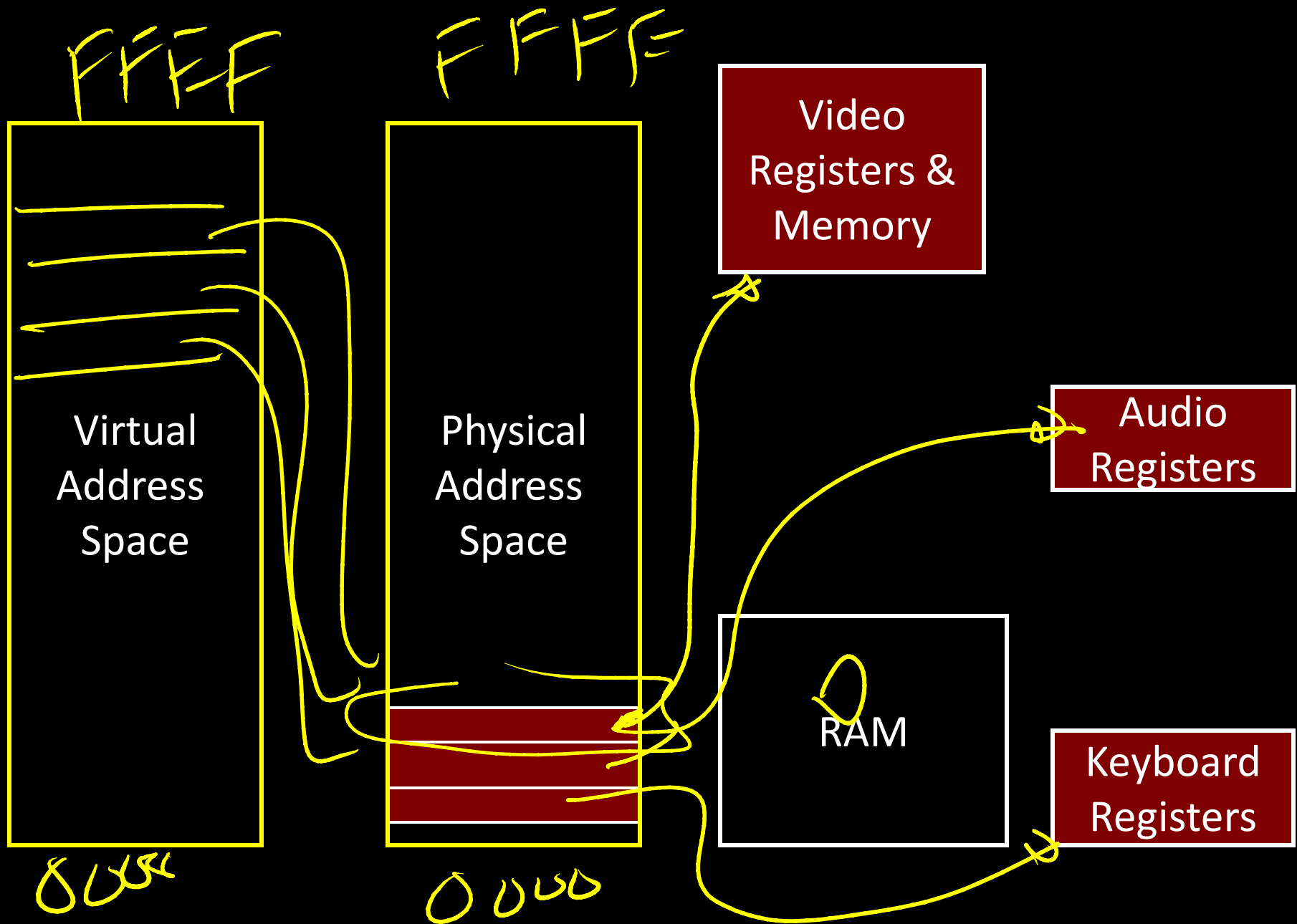- Protection: only allowed in kernel mode

~~inb $a, 0x64~~

Syscall

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

13

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

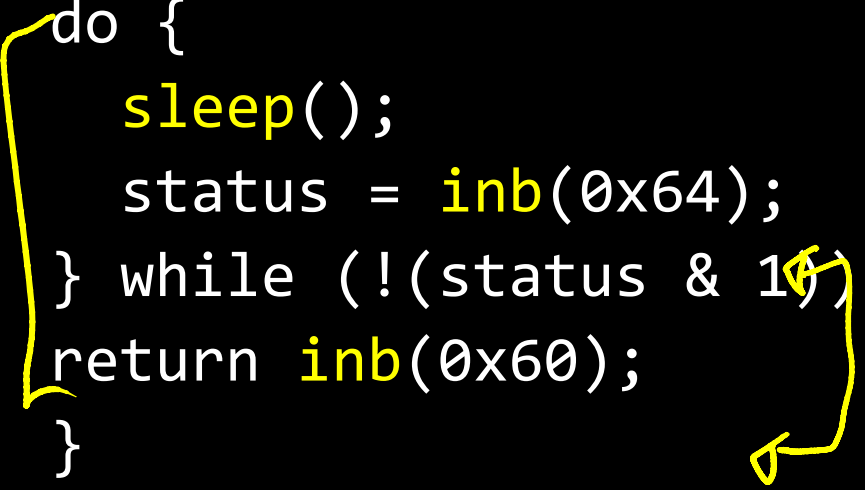A: Map registers into virtual address space

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

FFFF

FFFF

Virtual
Address
Space

Physical
Address
Space

Video
Registers &
Memory

Audio
Registers

RAM

Keyboard
Registers

8000

0 000

Programmed I/O

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

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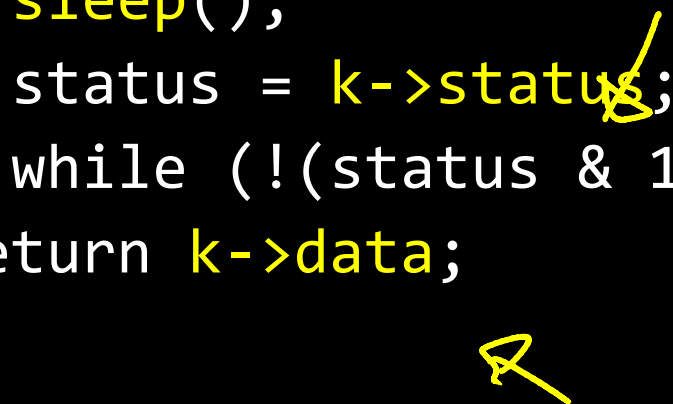
```
struct kbd {
  char status, pad[3];
  char data, pad[3];
};
kbd *k = mmap(...);

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

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
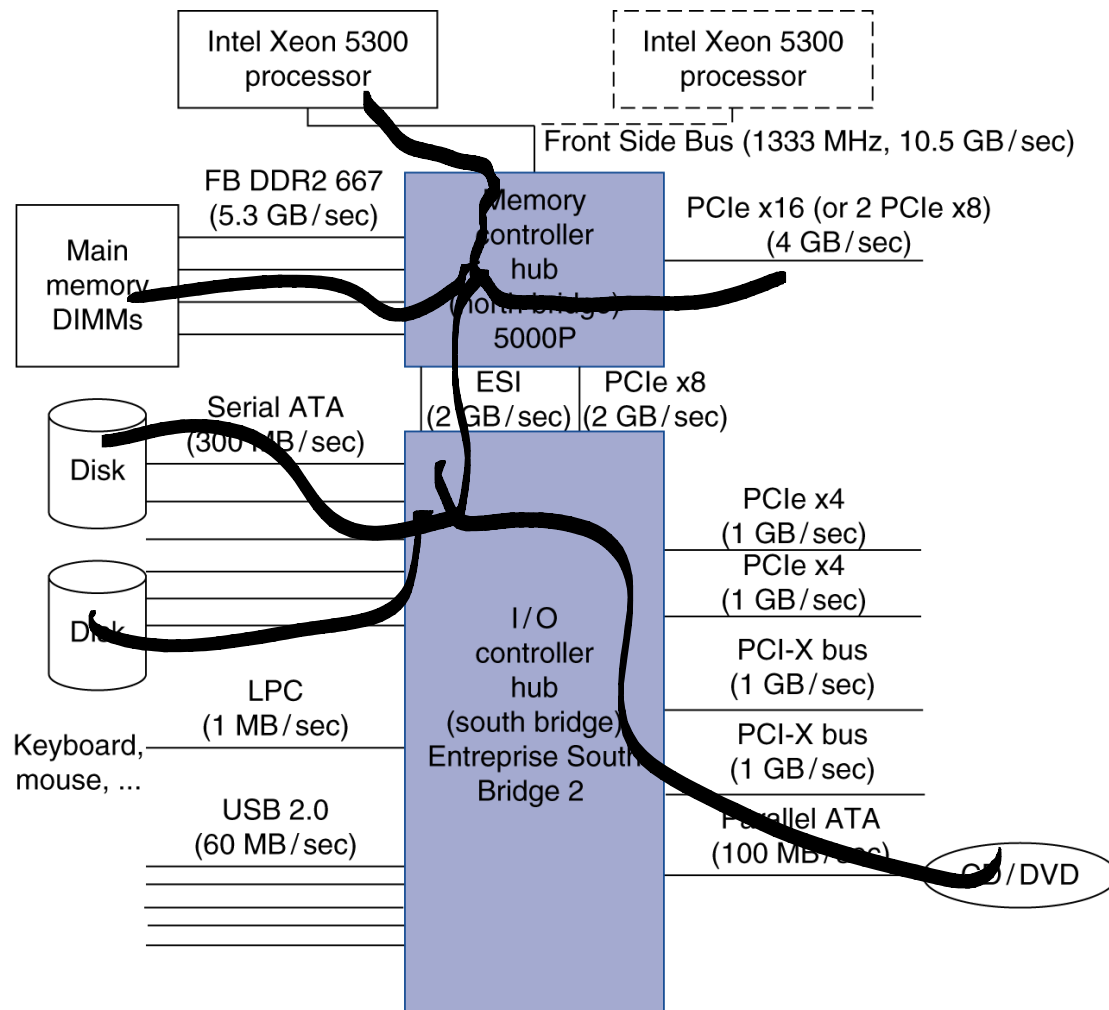
Sometimes for very active devices too…

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

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

LPC (1 MB/sec)

Keyboard, mouse, ...

USB 2.0 (60 MB/sec)

I/O controller hub (south bridge) Entreprise South Bridge 2

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

How to talk to device?

Programmed I/O or Memory-Mapped I/O
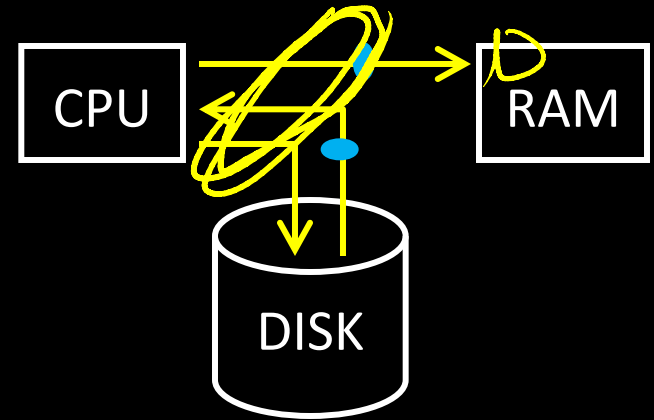
How to get events?

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;
```

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

CPU

RAM

DISK

Q: How to transfer lots of data efficiently?

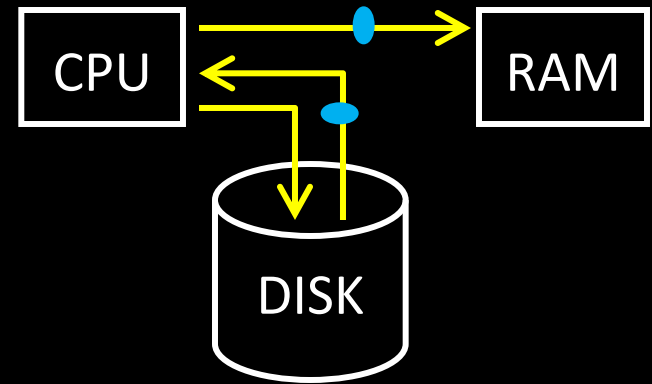A: Have device access memory directly

Direct memory access (DMA)

- OS provides starting address, length
- controller (or device) transfers data autonomously
- Interrupt on completion / error
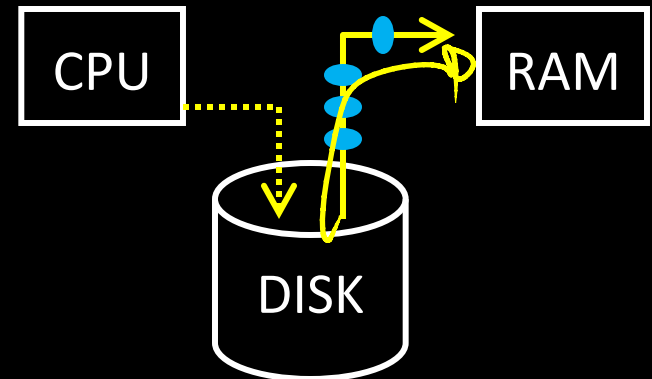
Programmed I/O xfer: Device ←→ CPU ←→ RAM

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: 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;
```
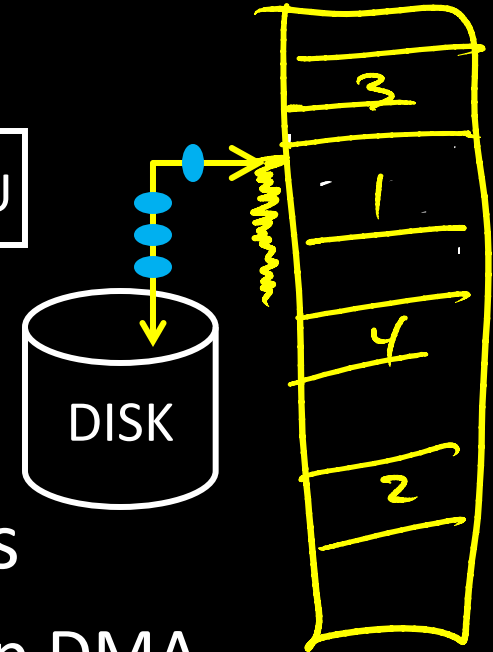
Issue #1: DMA meets Virtual Memory

RAM: physical addresses

Programs: virtual addresses

CPU   MMU

DISK

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)

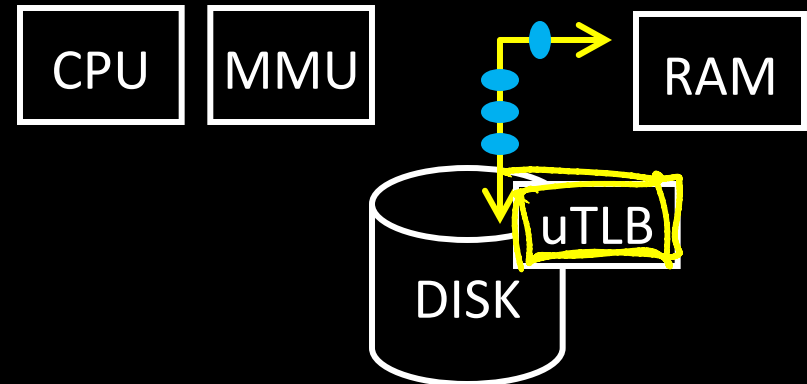# DMA example: reading from audio (mic) input

- 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;
```

Issue #1: DMA meets Virtual Memory

RAM: physical addresses

Programs: virtual addresses

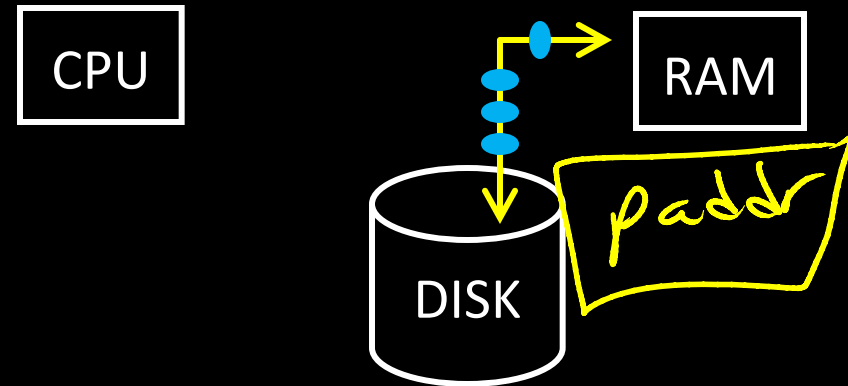| CPU | MMU |

RAM

uTLB

DISK

Solution 2: DMA uses virtual addresses

• OS sets up mappings on a mini-TLB

Issue #2: DMA meets *Paged* Virtual Memory

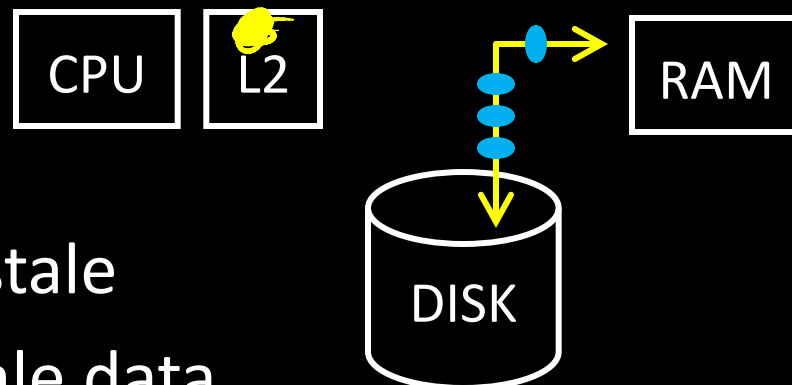DMA destination page
  may get swapped out

CPU

RAM

DISK

paddr

Solution: Pin the page before initiating DMA

Alternate solution: Bounce Buffer

- DMA to a pinned kernel page, then memcpy elsewhere

28

Issue #4: DMA meets Caching

DMA-related data could
  be cached in L1/L2

CPU   L2   RAM

DISK

- 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!)

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
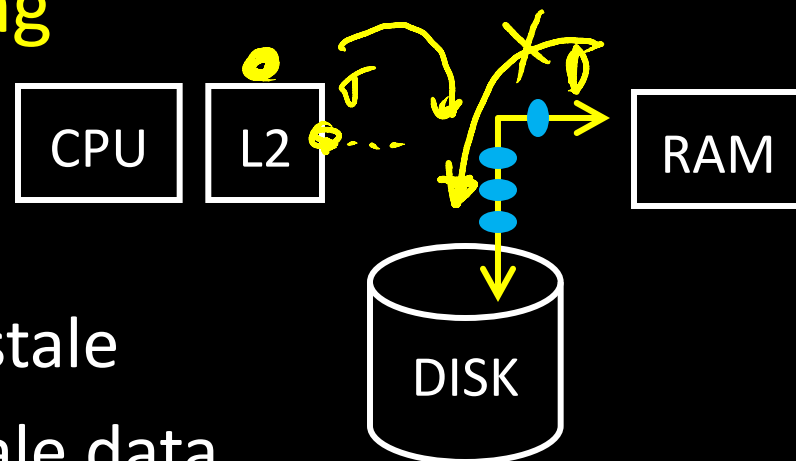
Solution 2: (hardware coherence aka snooping)

- 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

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