

# I/O

**Anne Bracy**

**CS 3410**

Computer Science

Cornell University

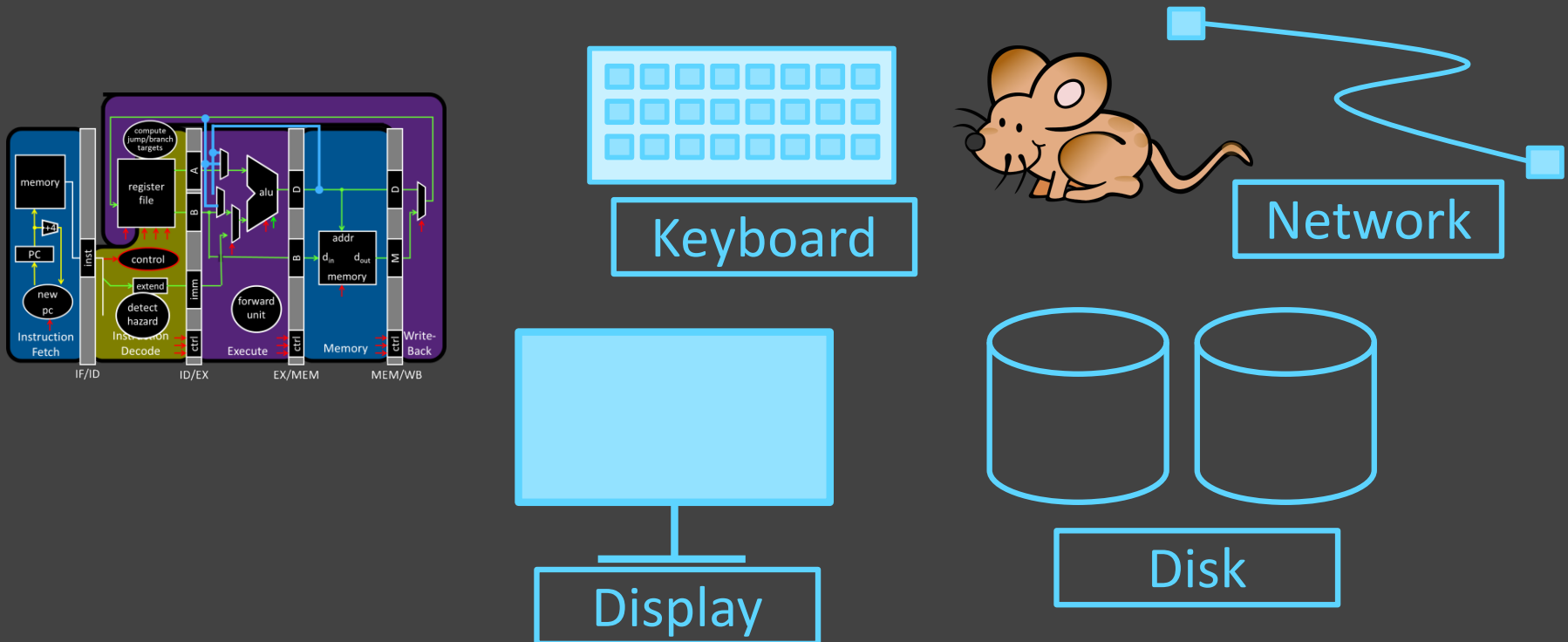
The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.

# Big Picture: Input/Output (I/O)

How does a processor interact with its environment?

Computer System =

Memory + Datapath + Control + **Input + Output**

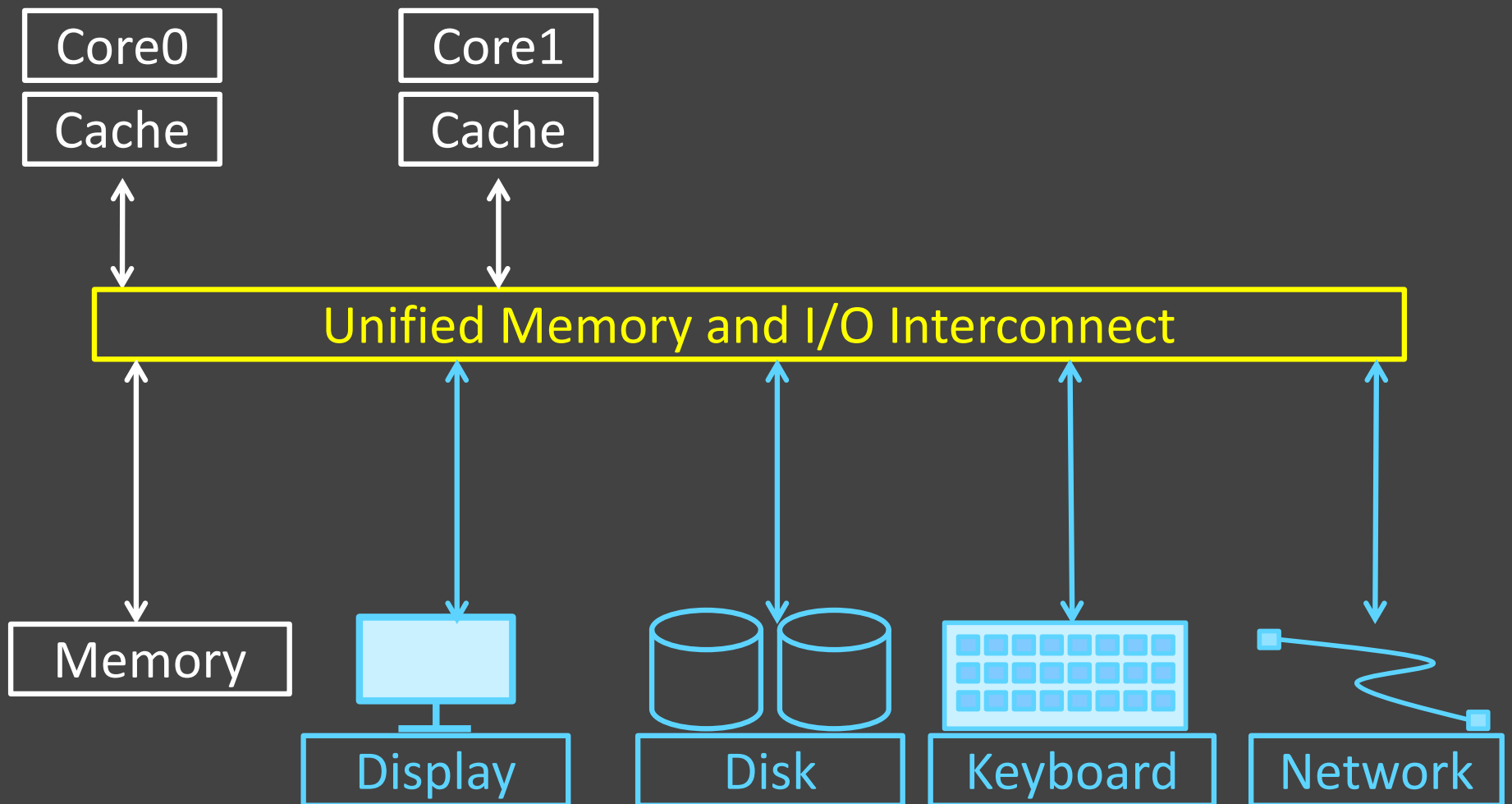


# I/O Devices Enables Interacting w/ Environment

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

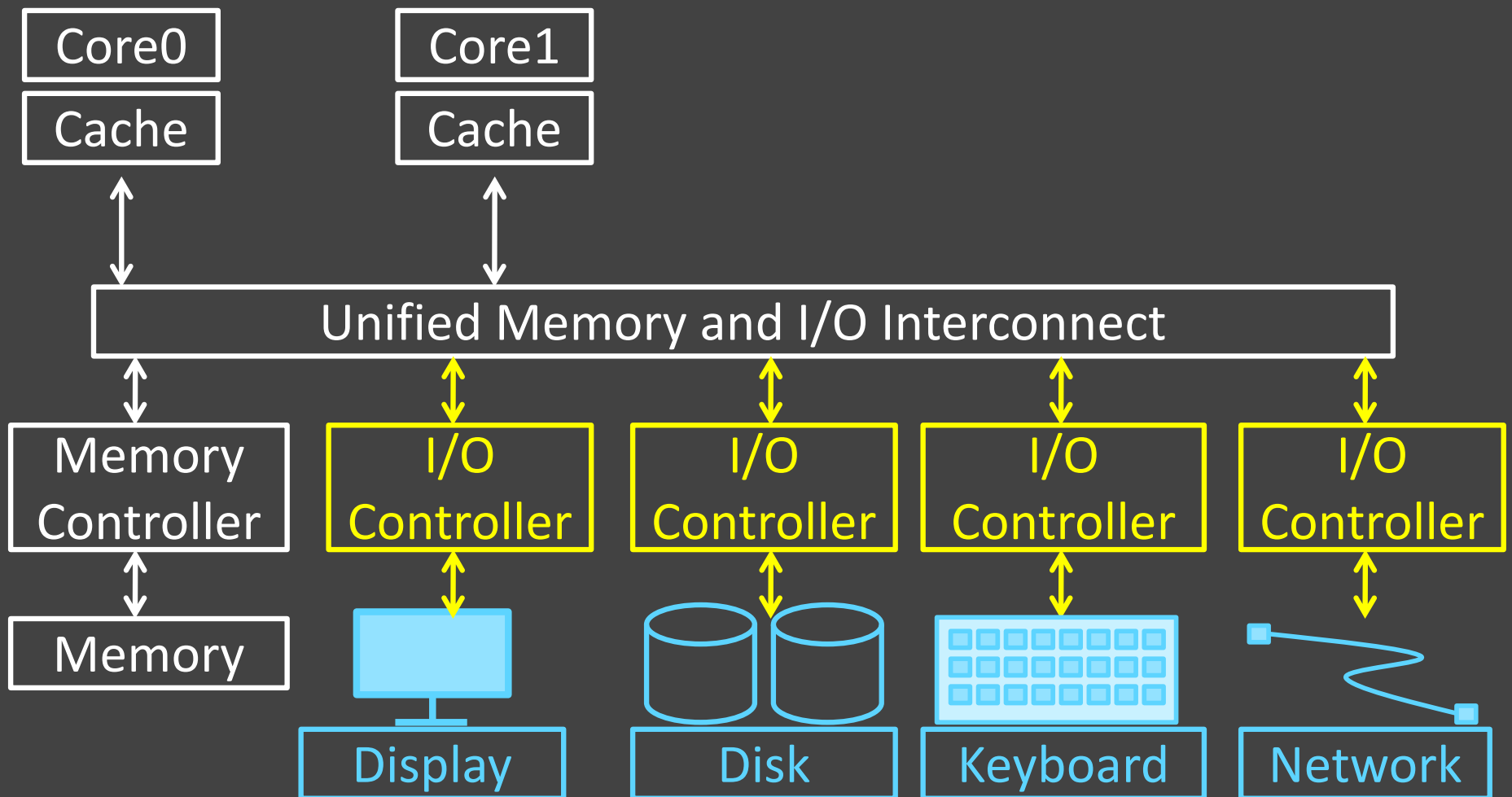
# Round 1: All devices on one interconnect

Replace *all* devices as the interconnect changes  
e.g. keyboard speed == main memory speed ?!



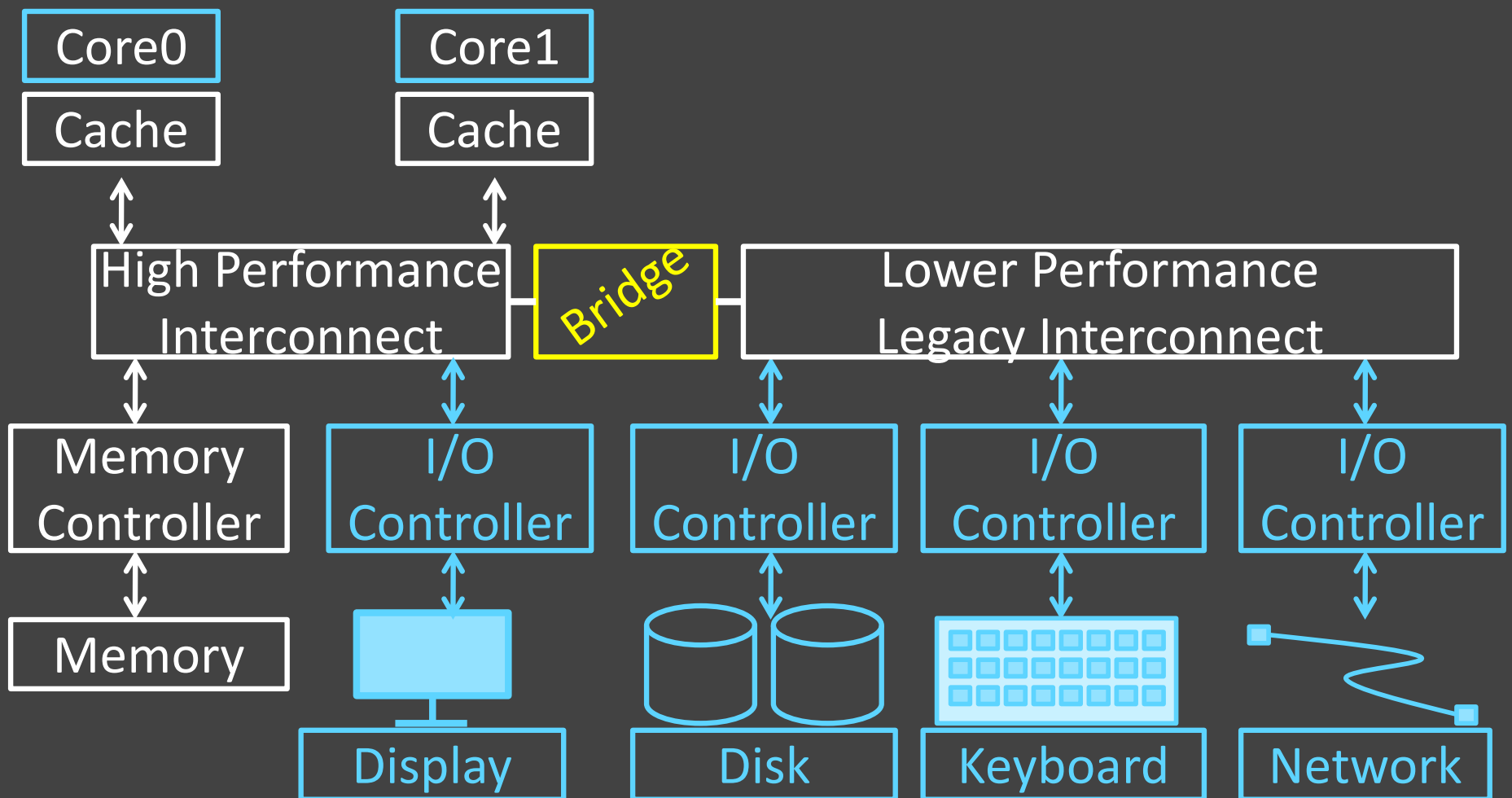
# Round 2: I/O Controllers

Decouple I/O devices from Interconnect  
Enable smarter I/O interfaces



# Round 3: I/O Controllers + Bridge

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



# 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

# 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
USB 3.0	External	127	2	625M
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



# 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

# I/O Device API

Simple (old) example: AT Keyboard Device



8-bit Status: 

PE	TO	AUXB	LOCK	AL2	SYSF	IBS	OBS
----	----	------	------	-----	------	-----	-----

8-bit Command:

0xAA = “self test”

0xAE = “enable kbd”

0xED = “set LEDs”

...

8-bit Data:

scancode (when reading)

LED state (when writing) or ...

Input  
Buffer  
Status



Output  
Buffer  
Status

# 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
- Protection: only allowed in kernel mode

Kernel boundary crossing is expensive

# Communication Interface

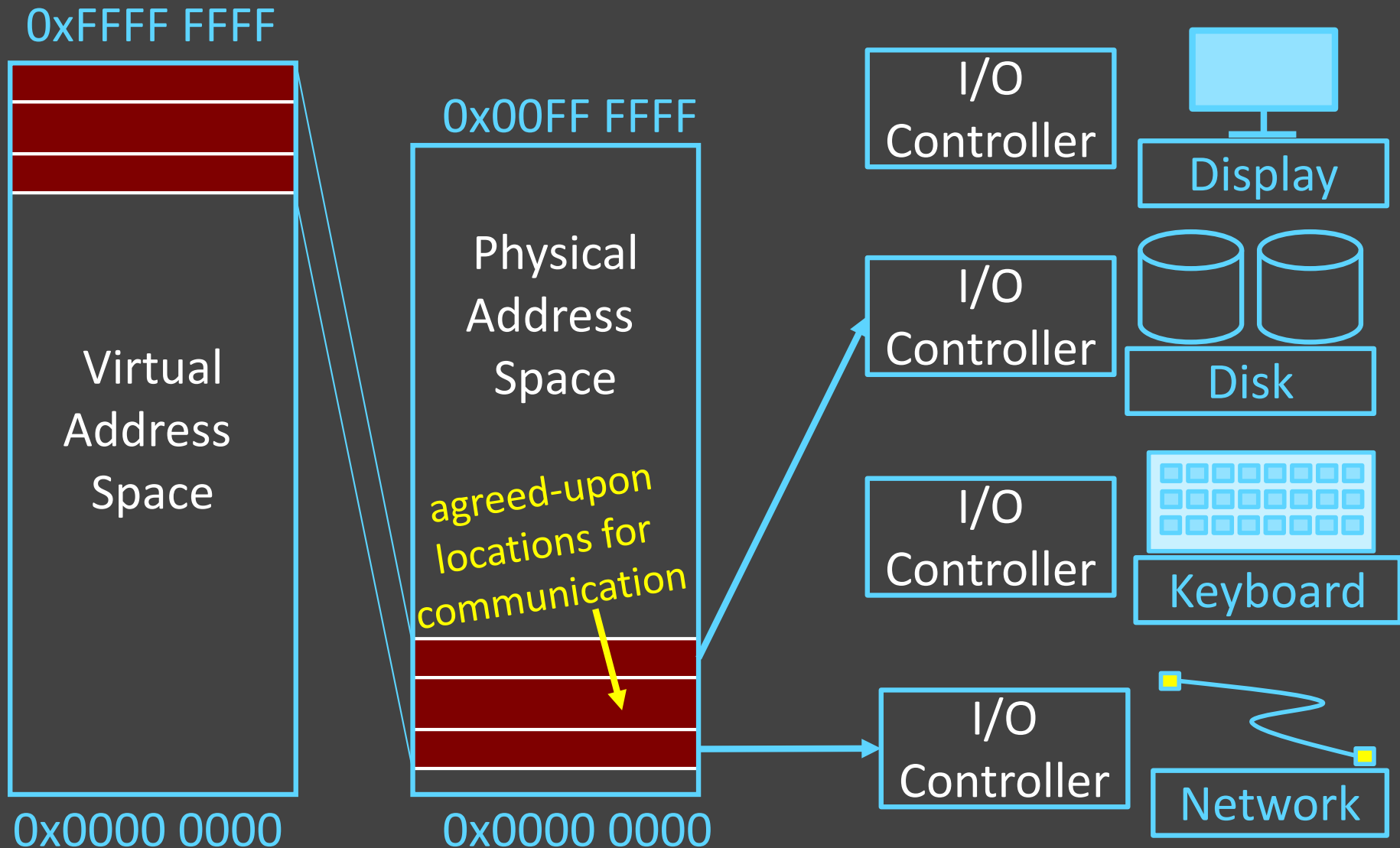
Q: How does ~~program~~ ~~OS~~ code talk to device?

A: Map registers into virtual address space

Memory-mapped I/O ← 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



Less-favored alternative = Programmed I/O:

- Syscall instructions that communicate with I/O
- Communicate via special device registers

# Device Drivers

## Programmed I/O

*Both polling examples,  
But mmap I/O more efficient*

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

return inb(0x60);
}
```

syscall

**NO**  
syscall

## 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;
}
```

syscall

# 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,  
Very,  
Expensive

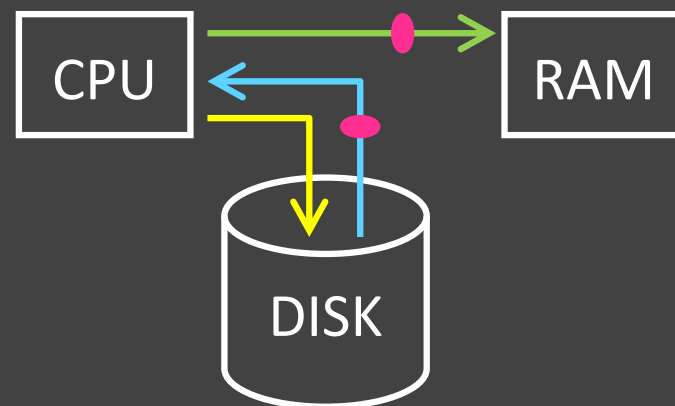


# Data Transfer

## 1. Programmed I/O: Device $\leftrightarrow$ CPU $\leftrightarrow$ RAM

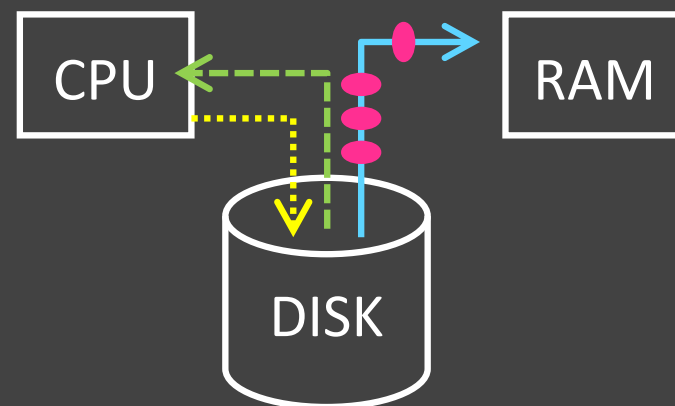
for ( $i = 1 \dots n$ )

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



## 2. Direct Memory Access (DMA): Device $\leftrightarrow$ RAM

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



*Which one is the winner? Which one is the loser?*

# 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 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;
```

# 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

# Polling vs. Interrupts

How does program learn device is ready/done?

## 1. Polling: Periodically check I/O status register

- Common in small, cheap, or real-time embedded systems
- + Predictable timing, inexpensive
- Wastes CPU cycles

## 2. Interrupts: Device sends interrupt to CPU

- Cause register identifies the interrupting device
- Interrupt handler examines device, decides what to do
- + Only interrupt when device ready/done
- Forced to save CPU context (PC, SP, registers, *etc.*)
- Unpredictable, event arrival depends on other devices' activity

*Which one is the winner? Which one is the loser?*

# I/O 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.