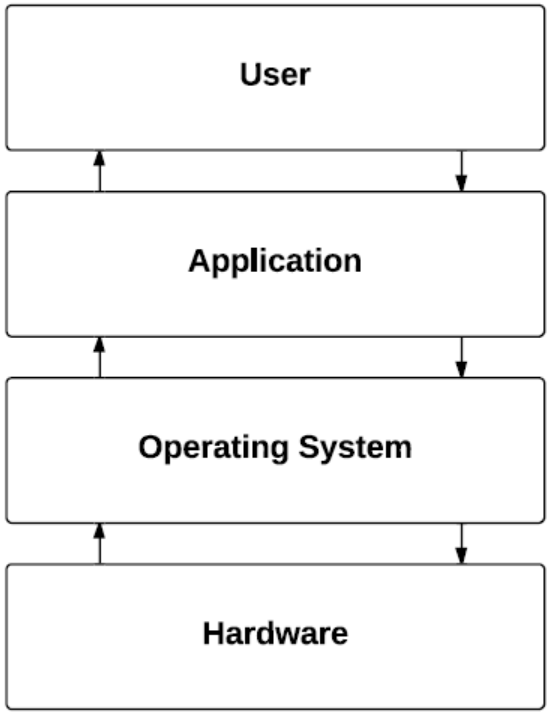
# CSE 330: Operating Systems

Adil Ahmad

**Lecture #14:** Device communication and protocols

# Recall the OS does the **plumbing** in modern computers

Provides an abstraction for the messy hardware




Manipulating hardware requires: (a) programming knowledge and (b) understanding of the hardware

**Plumbing analogy:**

With the OS' abstraction, users do not care about pipes, rather they can simply turn the taps on or off.

# What kind of hardware plumbing have we discussed thus far?

Provided below are some examples:

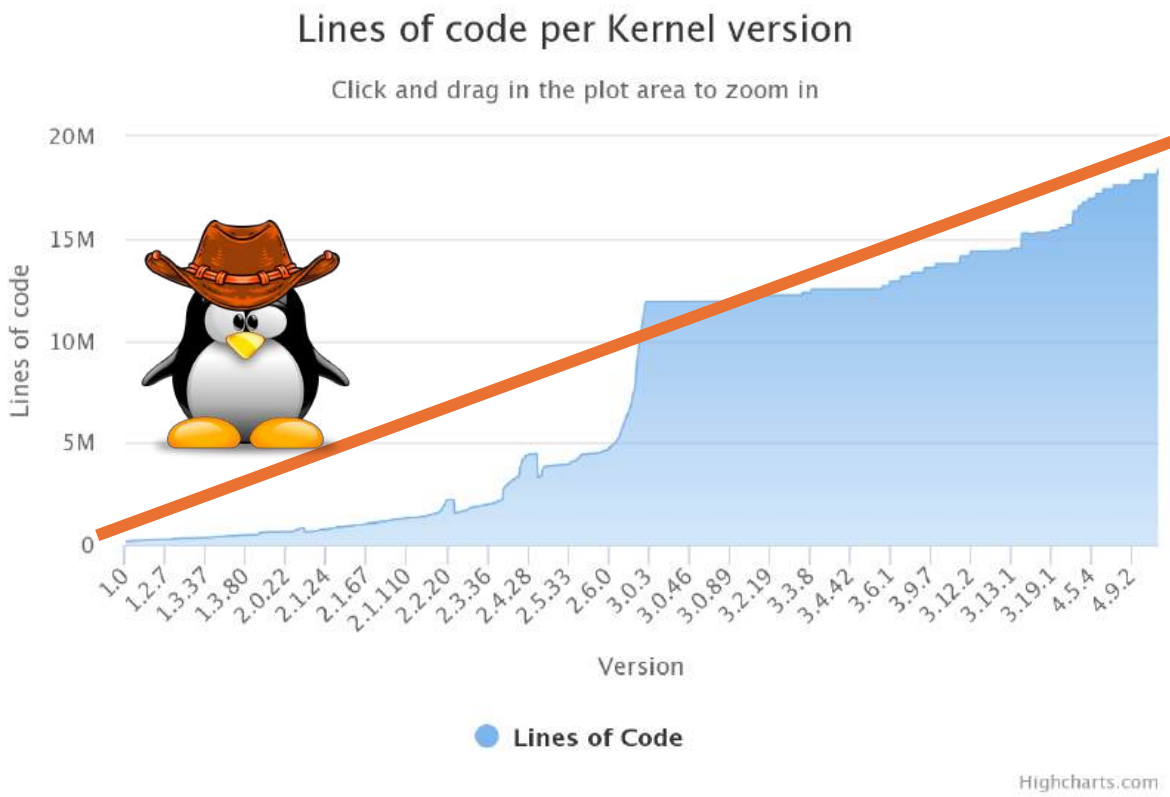
- **Memory management:** OS abstracted away aspects of
  - Memory allocation and deallocation from a user process

- **CPU management:** OS abstracted away aspects of
  - Translation Lookaside Buffer (TLB) management
  - CPU cache management

# External *hardware device management*

- This is another core operating system **plumbing** duty
- Provide a clean, easy-to-use way for external “connected” devices and processes in your computer to communicate
- Device management is a highly-complicated task for several reasons:
  - Devices are very complex and diverse in their functions
  - Devices can be connected to your computer “dynamically”
  - Many processes want to access devices at the same time
  - Many devices want to communicate with your computer at the same time

# *Remember* when I showed you this graph of Linux code




More than 31 million code lines today!

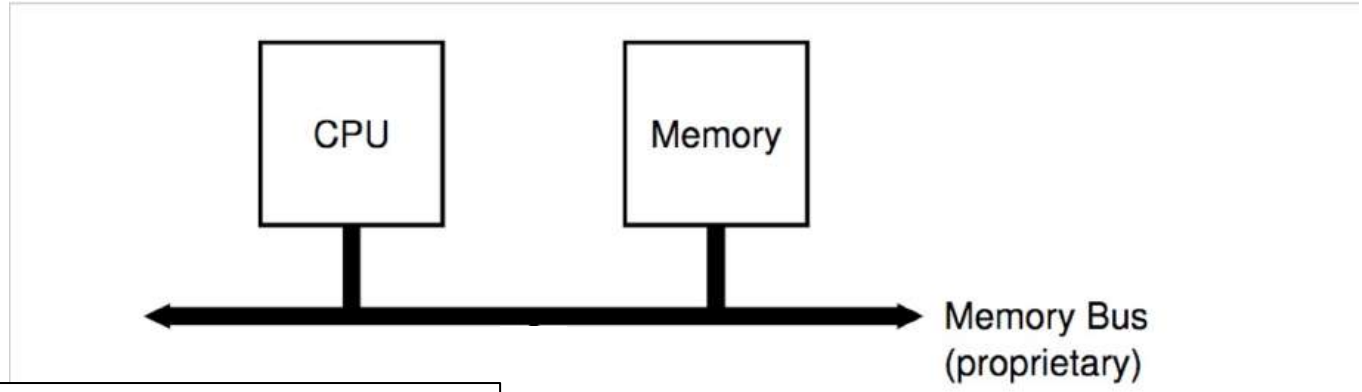
- A large chunk of the new lines (and bugs!) come from the ever-increasing list of devices that Linux is trying to support! ☺

# Let’s dive into *Input/Output (I/O)* devices

- **Definition:** a set of external connected interfaces that send and receive data (or information) to your computer

- **What are example of I/O devices connected to your computer?**
  - Hard disk drives (HDDs)
  - Solid state drives (SSDs)
  - Keyboard
  - Mouse
  - Graphics Processing Units (GPUs)
  - USB drives, etc.

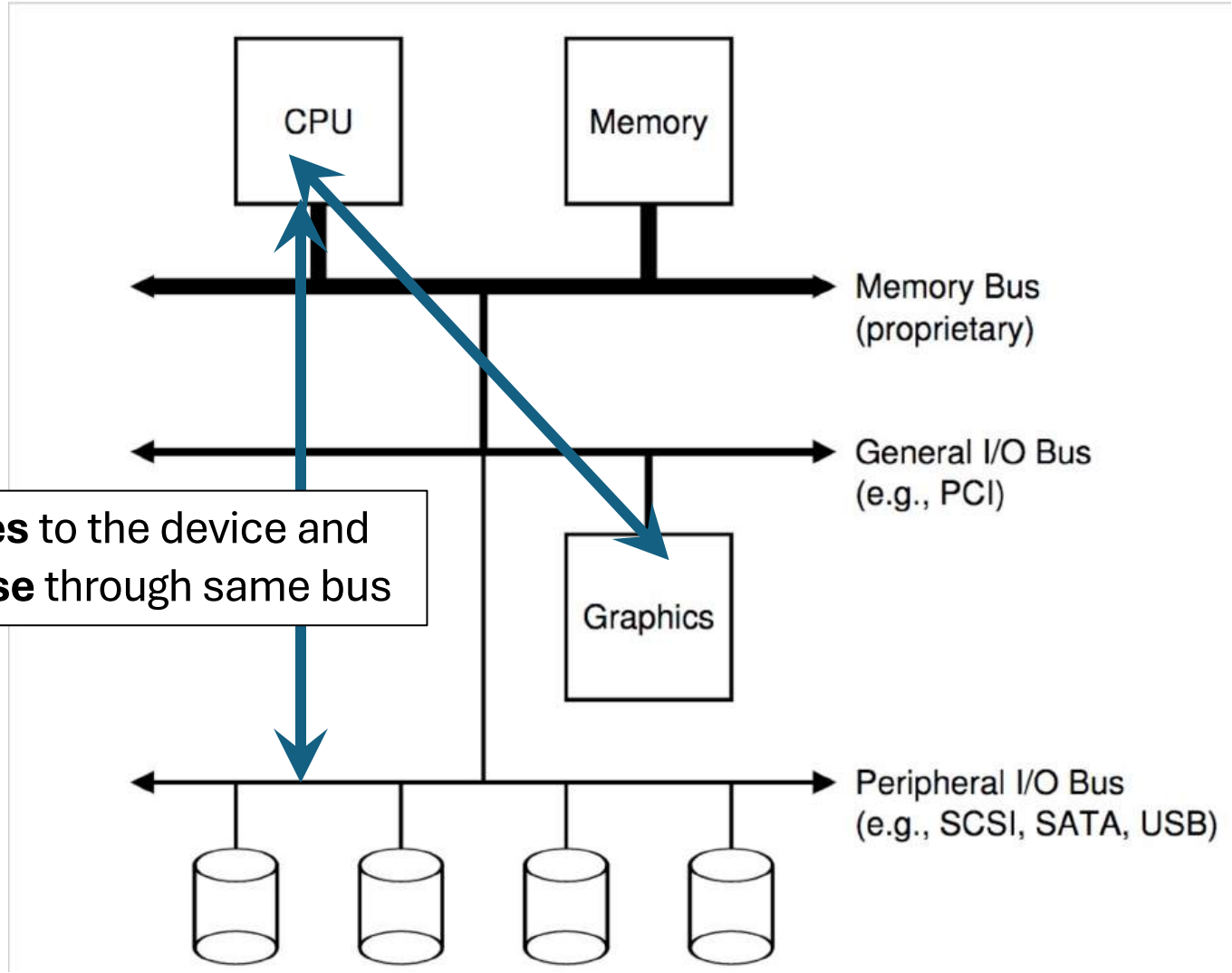
# Let’s see how devices are connected to a computer




Devices that need **fast communication** are directly connected to the **main bus**

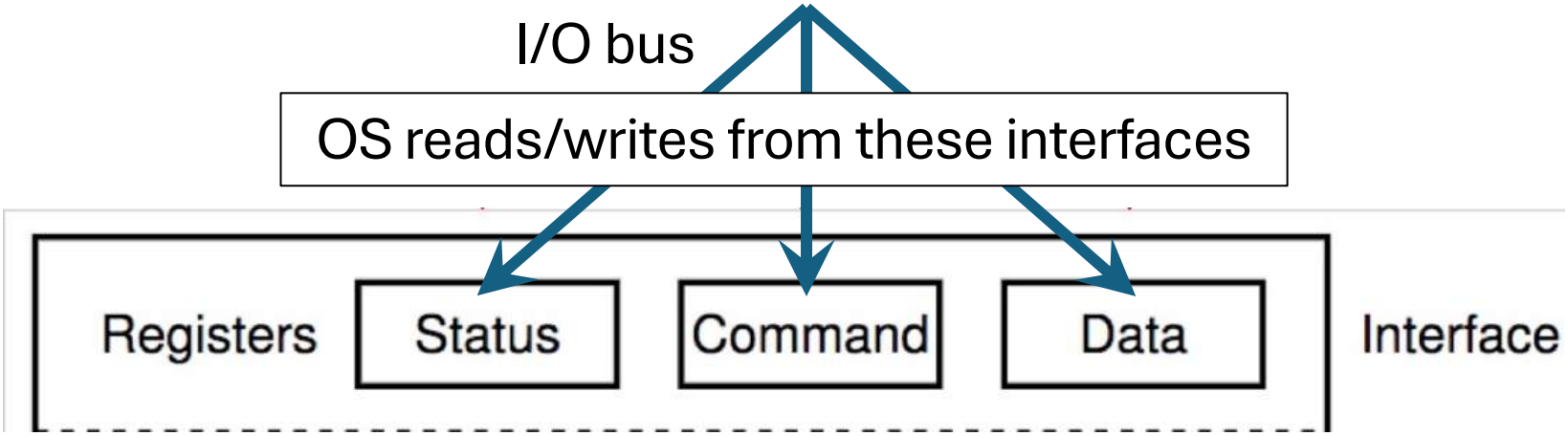
Rest are provided a “peripheral” bus connected to main using motherboards

# Bus interconnect allows two-way communication



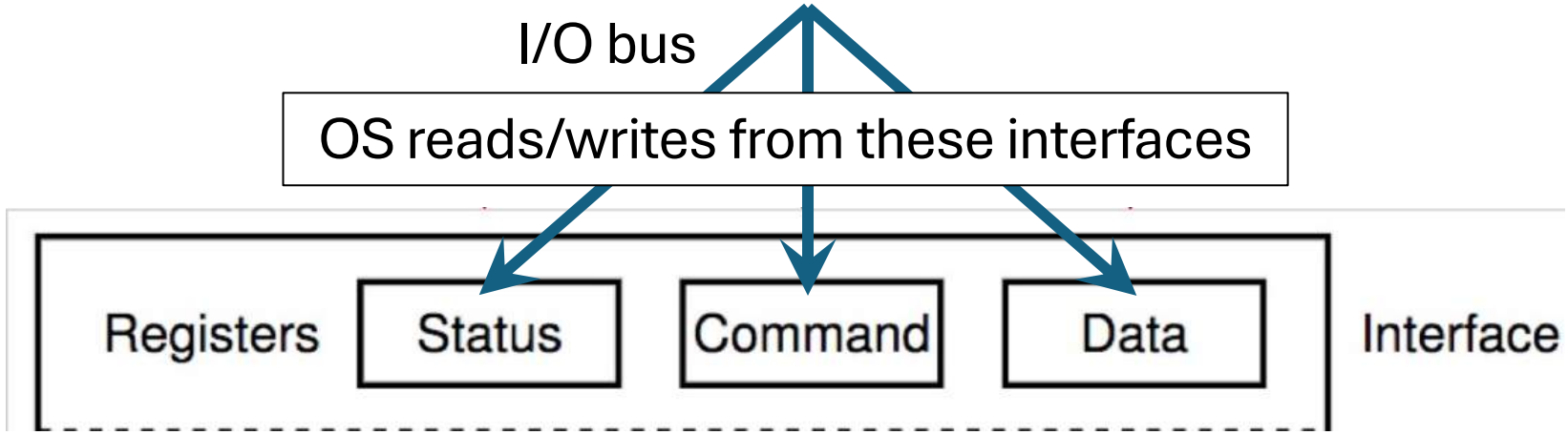

**Send messages** to the device and **receive response** through same bus

# Let’s see what a typical device looks like to the OS




- **Status:** Tells the OS whether the device is ready, busy, initialized, etc.
- **Command:** OS will write what it needs here, and if device is ready, it will perform the corresponding operation
- **Data:** OS will write any information the device needs to complete its task at this location

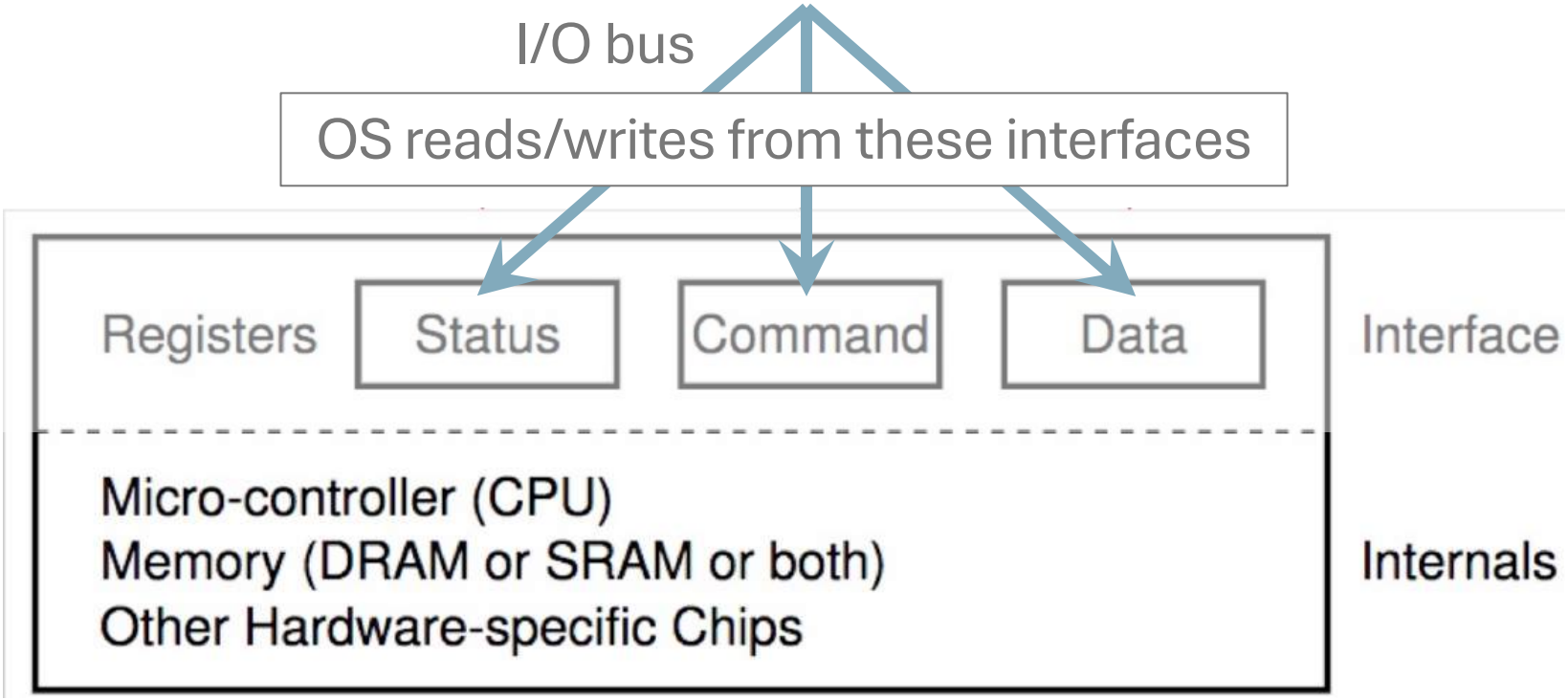
# How does the OS know how to talk to device registers?




| A2 | A1 | A0 | READ MODE | WRITE MODE |
|---|---|---|---|---|
| 0 | 0 | 0 | • Receive Holding Register | • Transmit Holding Register |
| 0 | 0 | 0 | N/A | • LSB of Divisor Latch when Enabled |
| 0 | 0 | 1 | N/A | • Interrupt Enable Register |
| 0 | 0 | 1 | N/A | • MSB of Divisor Latch when Enabled |
| 0 | 1 | 0 | • Interrupt Status Register | • FIFO control Register |
| 0 | 1 | 1 | N/A | • Line Control Register |
| 1 | 0 | 0 | N/A | • Modem Control Register |
| 1 | 0 | 1 | • Line Status Register | N/A |
| 1 | 1 | 0 | • Modem Status Register | N/A |
| 1 | 1 | 1 | • Scratchpad Register Read | • Scratchpad Register Write |

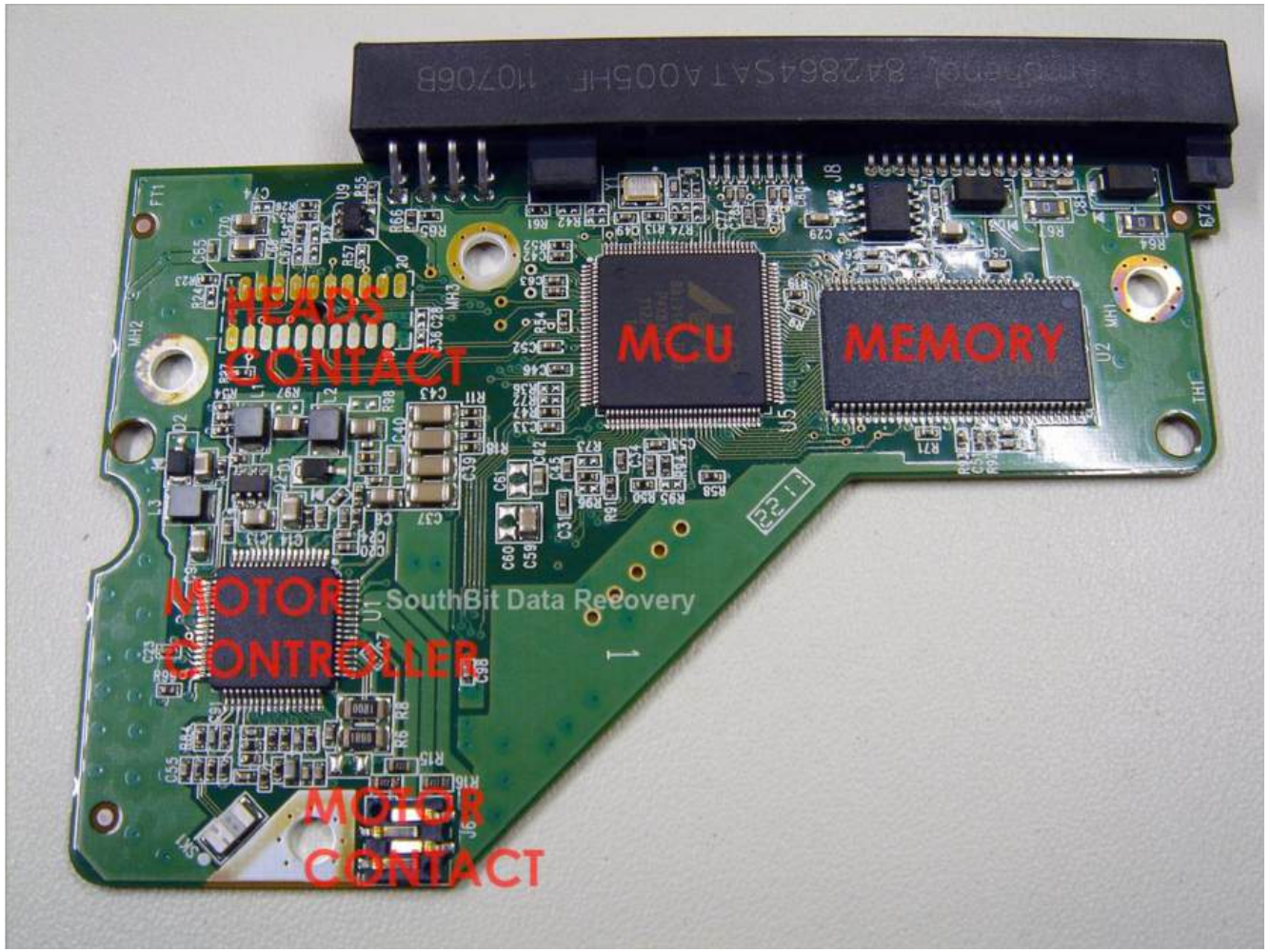
- Hardware manuals provided by hardware vendors explain the *"language"*!
- If you want to do OS research, get ready to read some badly written manuals! ☺

# The other “hidden” half of a typical device




- Device needs a **tiny CPU** (micro-controller) to perform computations
- Device (like CPU) will need some **tiny DRAM** for computation
- Some example of other chips:
  - Capacitor for power-backup
  - Hardware security chips, etc.

# An **example** of the hard drive disk PCB



# Understanding **I/O** communication

# Acknowledgement


- Most of the slide content in this section are taken directly from Prof. Zhicao Cao and Prof. Ming Zhao's CSE 330 classes

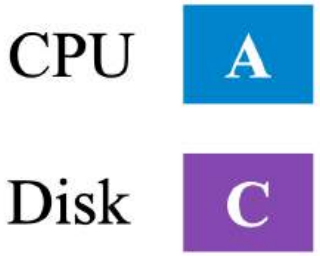
- These slides are fantastic!!

- Hence, I will present them mostly as is (with their permission, of course!)

# Basic I/O protocol from the OS perspective

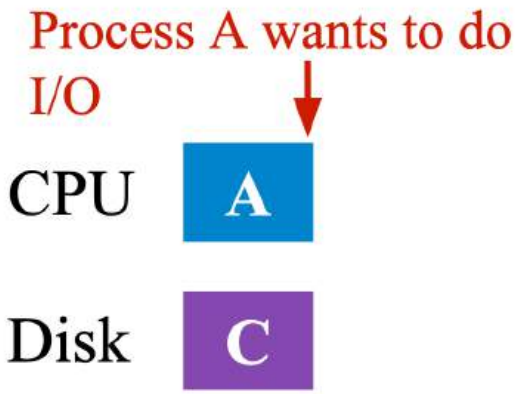
```
while (STATUS == BUSY)
   ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
   ; // spin
```

# Examining the workings of the basic I/O protocol



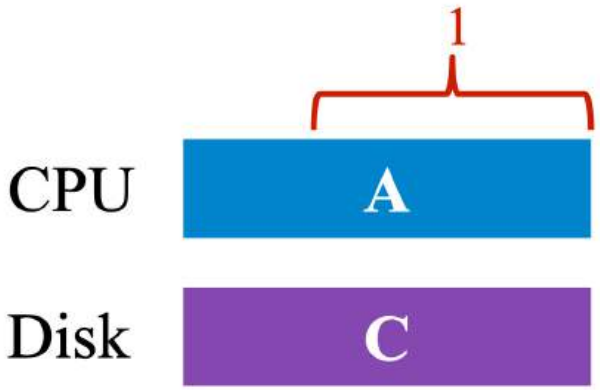

```
while (STATUS == BUSY)                  //1
   ; // spin
Write data to DATA register             //2
Write command to COMMAND register       //3
while (STATUS == BUSY)                  //4
   ; // spin
```

# Examining the workings of the basic I/O protocol



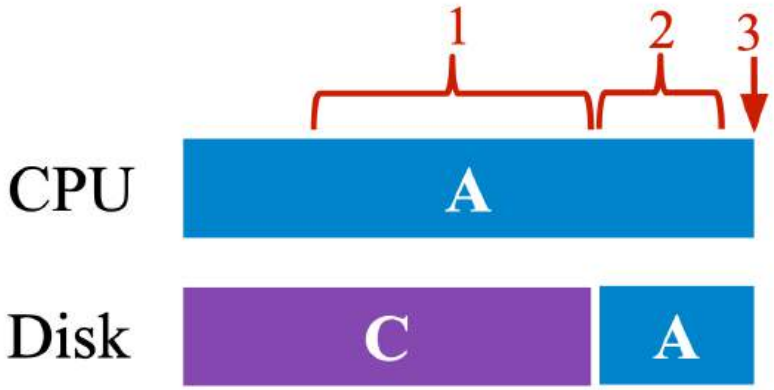

```
while (STATUS == BUSY)                  //1
   ; // spin
Write data to DATA register             //2
Write command to COMMAND register       //3
while (STATUS == BUSY)                  //4
   ; // spin
```

# Examining the workings of the basic I/O protocol



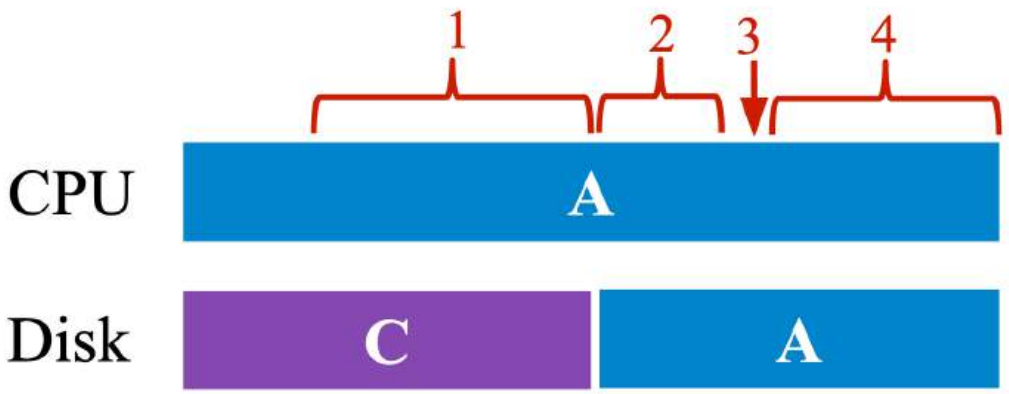

```
while (STATUS == BUSY)                    //1
   ; // spin
Write data to DATA register               //2
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
   ; // spin
```

# Examining the workings of the basic I/O protocol



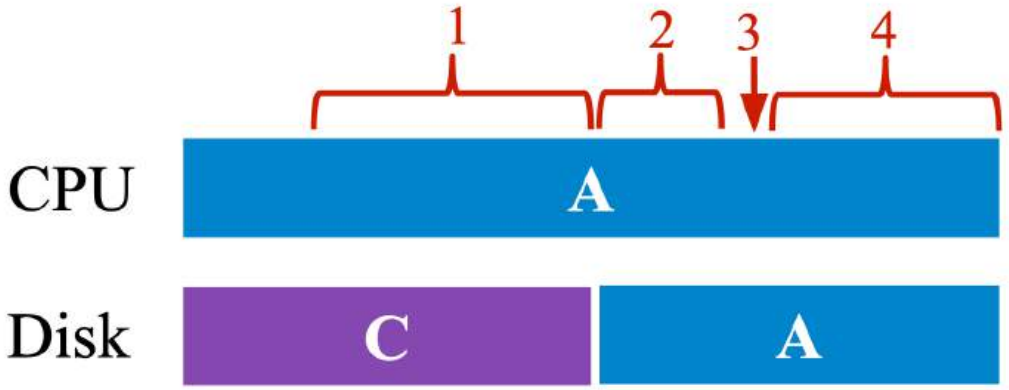

```
while (STATUS == BUSY)                    //1
   ; // spin
Write data to DATA register              //2
Write command to COMMAND register        //3
while (STATUS == BUSY)                    //4
   ; // spin
```

# Examining the workings of the basic I/O protocol



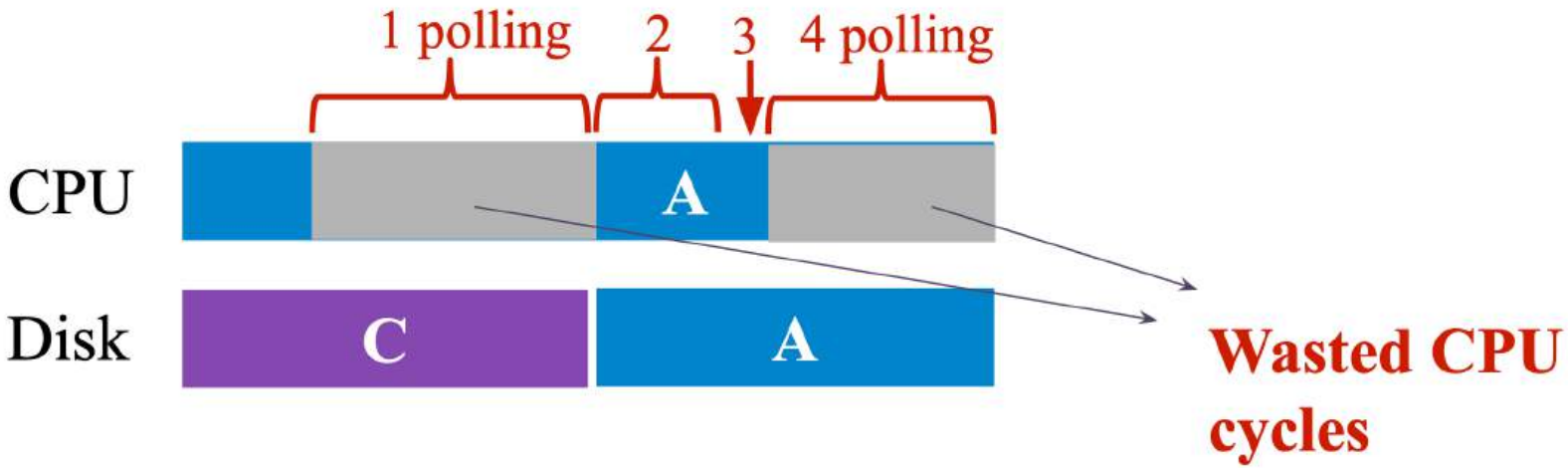

```
while (STATUS == BUSY)                    //1
   ; // spin
Write data to DATA register               //2
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
   ; // spin
```

# Is there some inefficiency (resource waste) problem in this example?



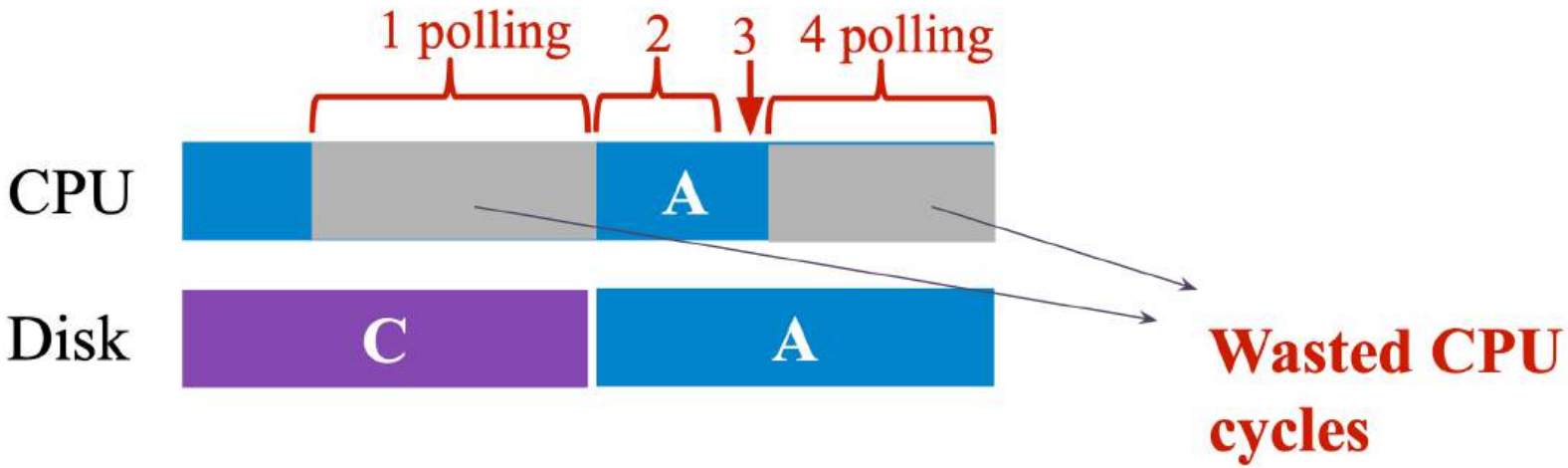

```
while (STATUS == BUSY)                    //1
   ; // spin
Write data to DATA register               //2
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
   ; // spin
```

# All this waiting (called *polling*) is not very efficient



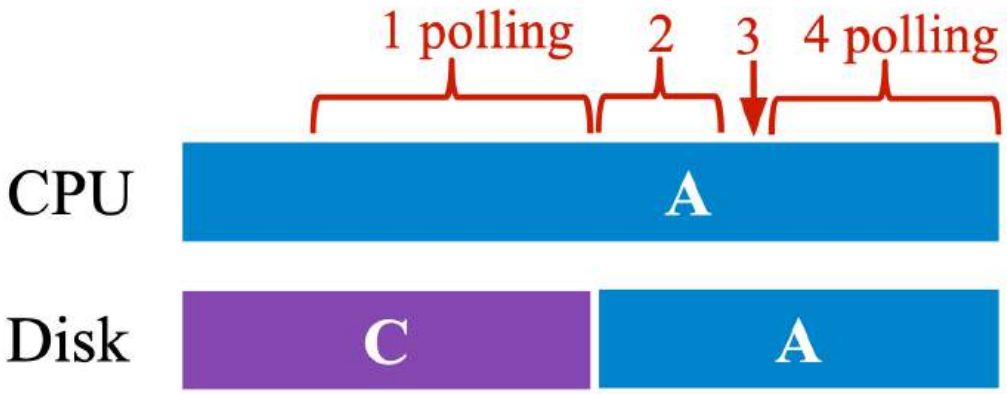

```
while (STATUS == BUSY)                    //1
   ; // spin
Write data to DATA register               //2
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
   ; // spin
```

# How would you solve this resource waste?



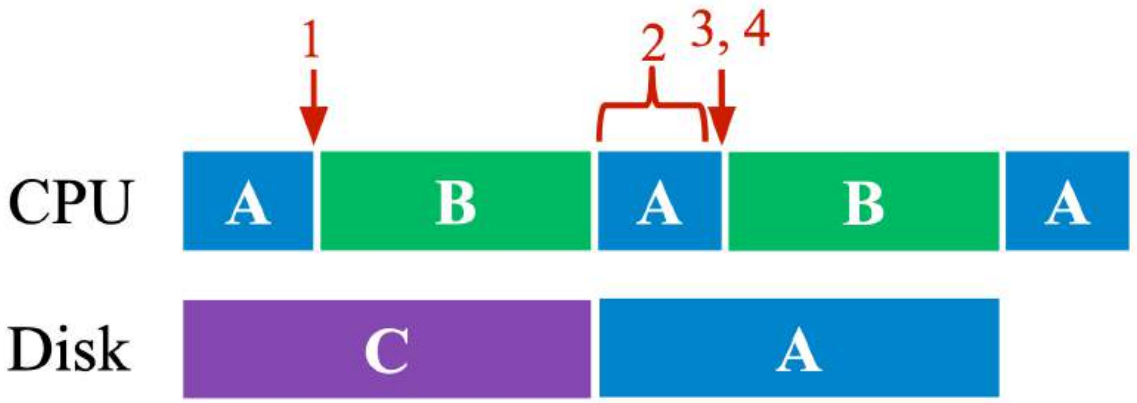

```
while (STATUS == BUSY)                    //1
   ; // spin
Write data to DATA register               //2
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
   ; // spin
```

# Solving the inefficiency problem using **interrupts**




```
while (STATUS == BUSY)                    //1
   wait for interrupt;
Write data to DATA register               //2
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
   wait for interrupt;
```

# Service other processes while the device is handling data




```
while (STATUS == BUSY)                    //1
   wait for interrupt;
Write data to DATA register               //2
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
   wait for interrupt;
```

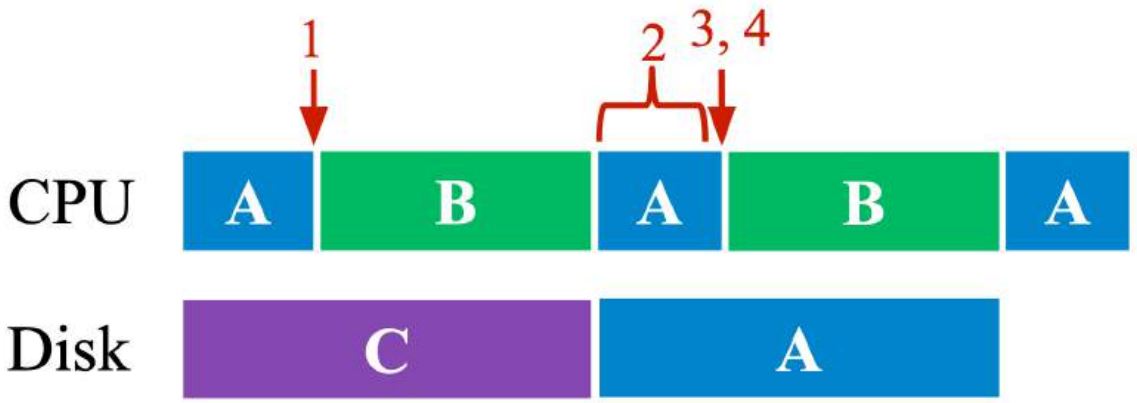
# Is there any disadvantage to using interrupts vs polling?

- Can lead to **more frequent context switches** (e.g., context switch to a different process and back)

- In practice today, OS uses a hybrid approach:
  - short device computations → polling and wait
  - long device computations → interrupts

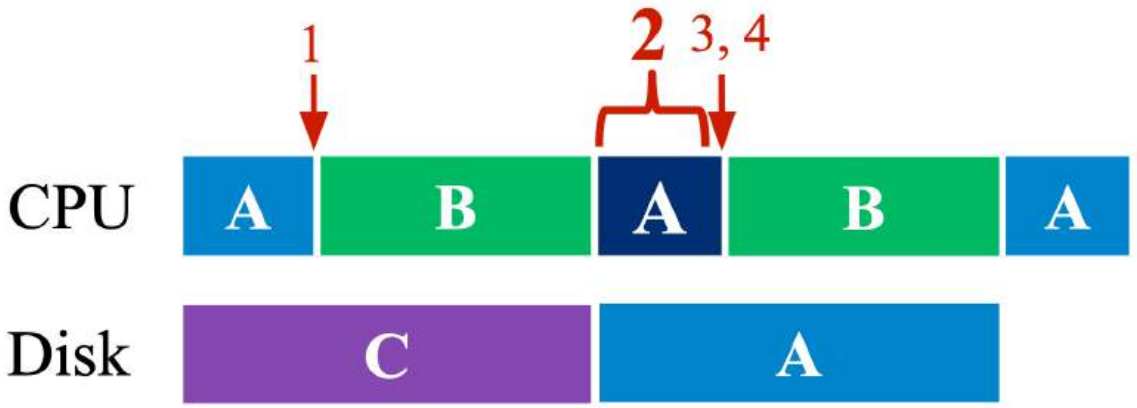
- Another hardware optimization: interrupt batching
  - CPU queues a sequence of interrupts, and sends them all at once

# Is there something else that could be expensive in this protocol?




```
while (STATUS == BUSY)                    //1
   wait for interrupt;
Write data to DATA register               //2
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
   wait for interrupt;
```

# Is there something else that could be expensive in this protocol?




```
while (STATUS == BUSY)                    //1
   wait for interrupt;
Write data to DATA register               //2
```

- A basic, menial operation that **peasants** can do *(copy data from one place to another)*
- Wasting your **expensive, precious, royal** CPU for this task is not ideal

```
wait for interrupt;
```

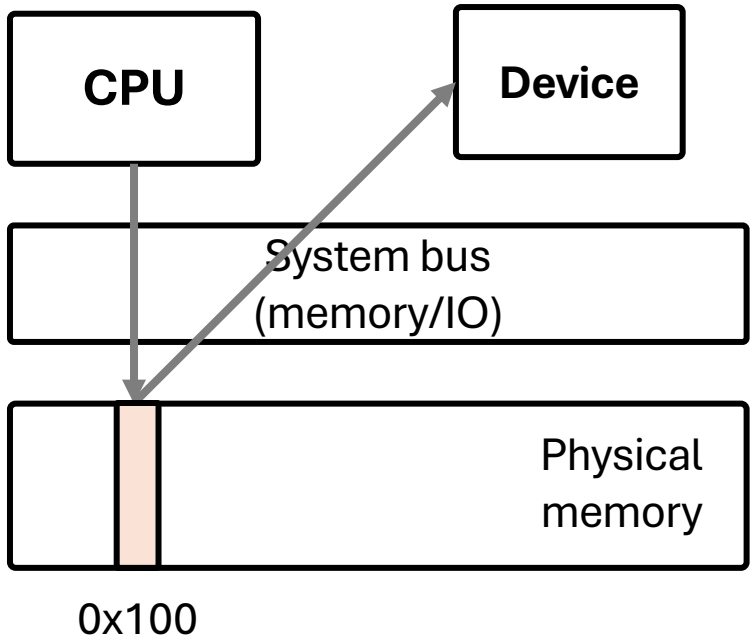
# Programmed I/O versus Direct Memory Access

- **Programmed I/O (PIO)**
  - This is the version we've discussed so far, and it is also called memory-mapped I/O (MMIO)
  - CPU directly tells device what data is needed
  - CPU is involved in the data transfer (e.g., writing to DATA register)

- **Direct Memory Access (DMA)**
  - CPU leaves (prepares) data in main memory
  - DMA hardware performs the data copy
  - CPU is free to complete other tasks in the background

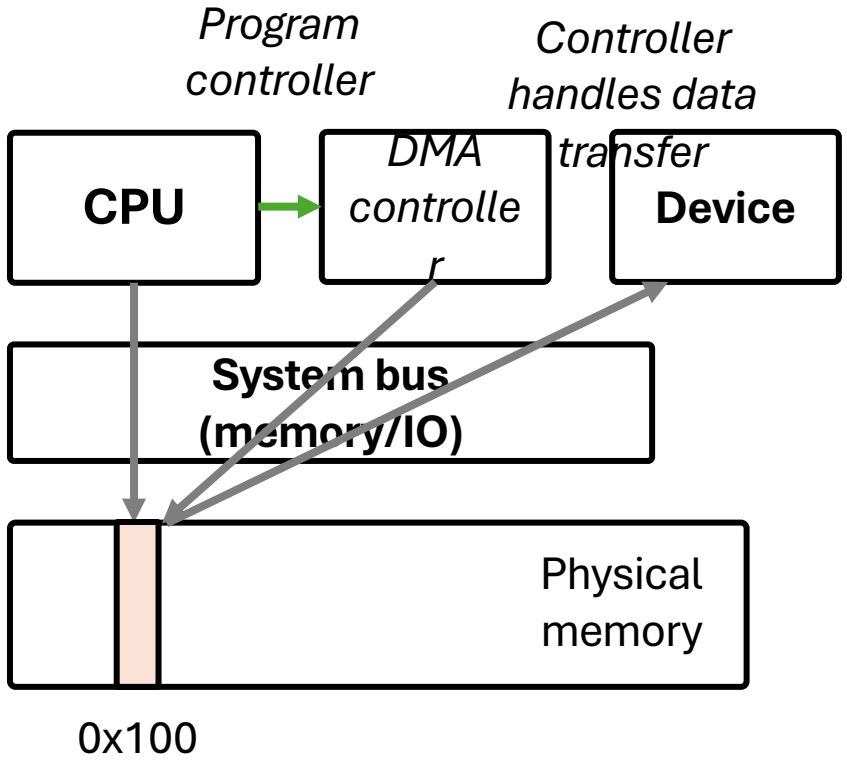
# *Understanding DMA:* memory access and PIO

- CPUs access memory by sending requests to a system bus
- Consider PIO: *writeReg(CMD, 0x100)*
  - Copy data at 0x100 to CMD register
  - System bus forwards data to device
- Depending on size to send, the CPU must:
  - Prepare data in DRAM
  - Write to MMIO address to copy data (e.g., byte-by-byte)



# *Understanding DMA:* using controllers for speed-up

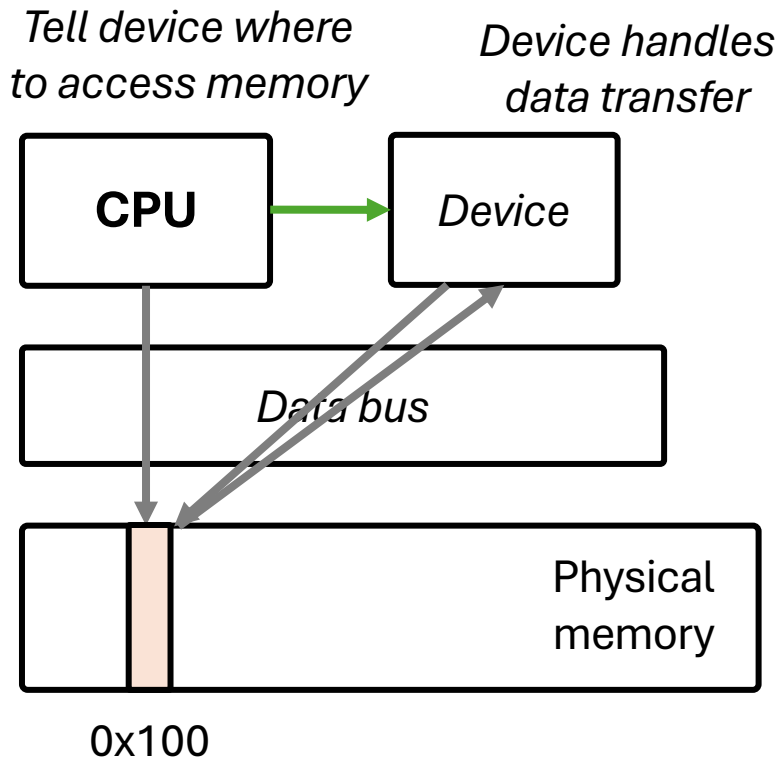
- DMA controllers:
  - Specialized hardware units that can only be programmed to read/write memory
- OS can then do the following:
  - Programs DMA controller to send data to the device in the background
  - Set up interrupts for when DMA transfers are completed
- OS can resume execution of other tasks




Is this approach still (somewhat) inefficient?

# DMA **version 2.0**: removing the controller

- Allow device to directly use the system bus (also called *bus mastering*)
- OS can then do the following:
  - Prepares memory region (e.g., 0x100) with a batch of data
  - Tell device (e.g., using PIO) where the data is and let device handle the rest
  - Interrupts enabled like before
- No need for synch. between the controller and device; hence, faster!



# Quick comparison: *controllers* and *bus mastering*

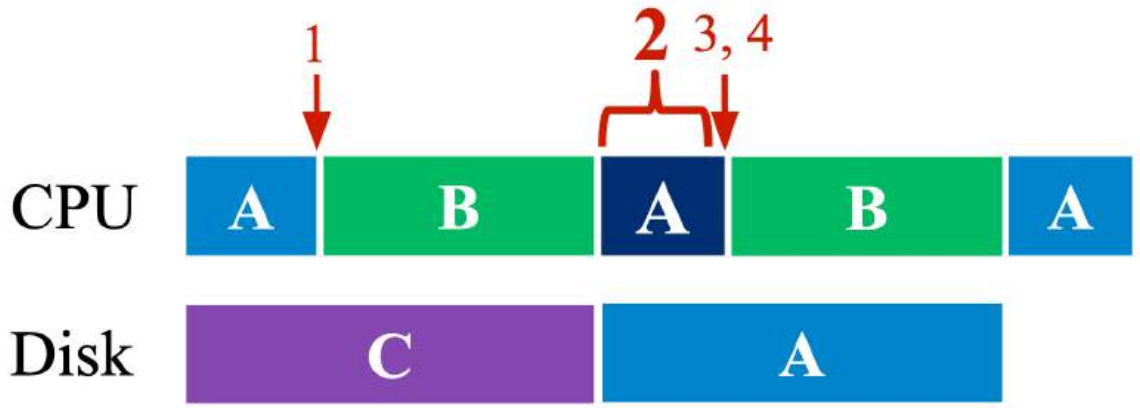
## Controller approach (also called *third-party DMA*)

- Leaves complex handling of system bus requests to the system (e.g., motherboard/chipset)
- Suitable for simpler devices
- Slower since it requires interactions b/w CPU-controller-device

## Bus mastering approach (also called *first-party DMA*)

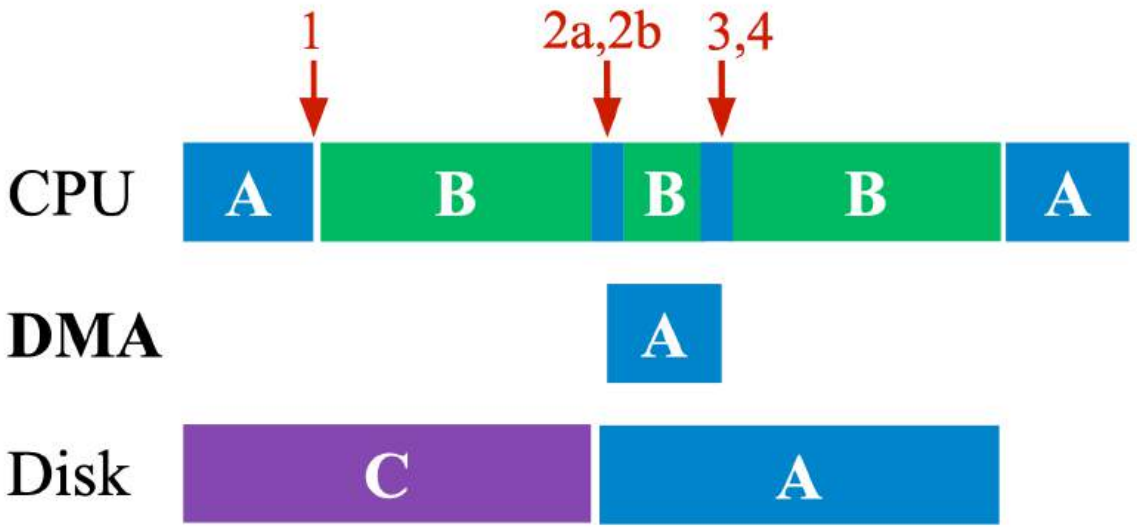
- Requires complex tasks be done by the device directly
- Suitable for more advanced devices (e.g., GPUs)
- Faster since it avoids having the middle-man

# Recap of code-level illustration of Programmed I/O (before)




```
while (STATUS == BUSY)                    //1
   wait for interrupt;
Write data to DATA register               //2
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
   wait for interrupt;
```

# Code-level illustration for Direct Memory Access (after)




```
while (STATUS == BUSY)                    //1
   wait for interrupt;
Initiate DMA transfer                     //2a
Wait for interrupt                        //2b
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
   wait for interrupt;
```

# Is there a security threat with bus mastering DMA?

- Sadly, yes! Can directly access physical memory region

- **Malicious devices can send DMA requests for any regions**
  - e.g., belonging to the OS or other programs etc.
  - Google the **BadUSB** attack! ☺

- **OSs can also be malicious and leverage DMA to steal data**
  - **Any example scenario?**
  - Consider the OS running inside your VirtualBox setups
  - If it is malicious, it can try to use a device (e.g., keyboard) to access data from your host OS using DMA

# Is there a similarity to this problem with processes?

- **Yes!** If processes directly access physical memory, they could also access memory regions belonging to other processes (lecture 4 – 7)
- **How did we solve that problem back then?**
- We leveraged a translation layer (aka *page tables*) to define and restrict regions of memory accessible to processes

# Adding *another* translation layer for **DMA protection**

- Enabled by a hardware unit inside the CPU called the **I/O memory management unit (IOMMU)**
- The OS sets *translation tables* for DMA and all access to the DRAM from devices occurs through I/O virtual address
- Like normal page tables, the OS can set permissions on mapped regions (e.g., **read-only**, **read-write**, etc.).

**Questions?** Otherwise, see you next class!