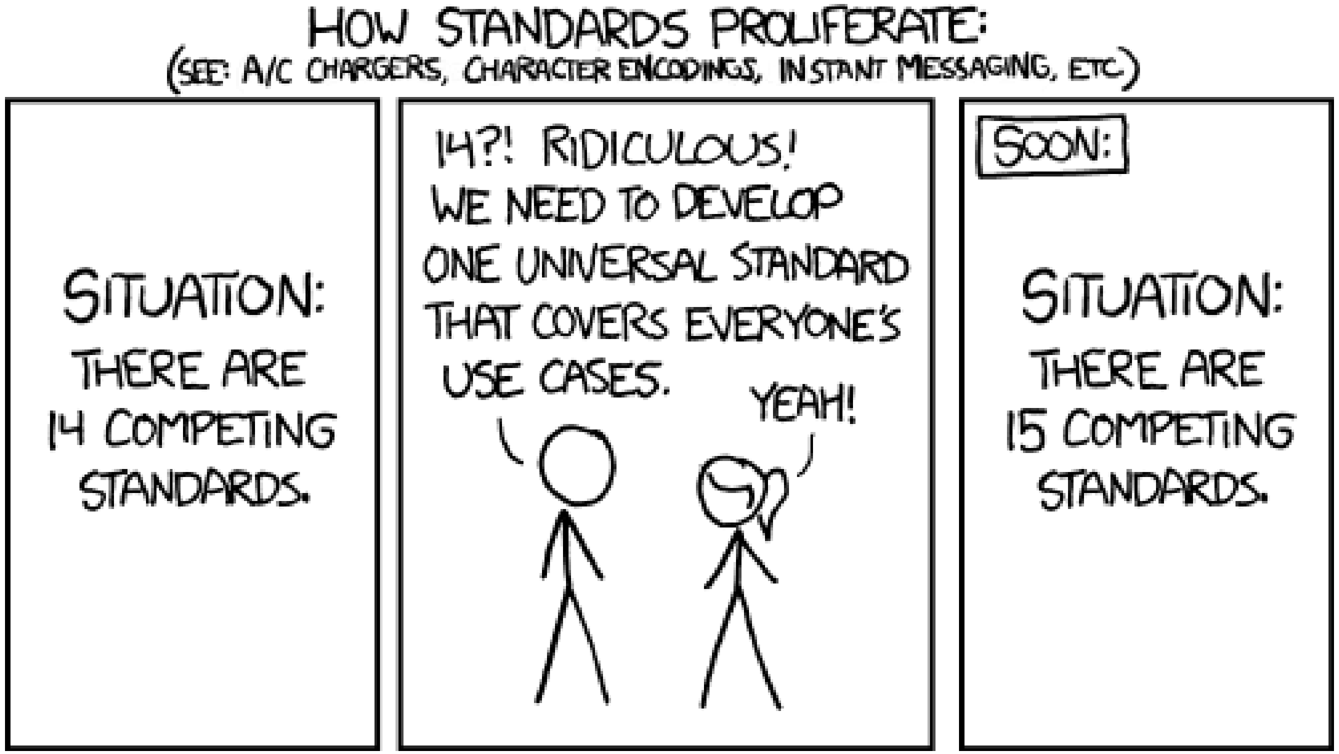
Input / Output

Kevin Webb
Swarthmore College
April 11, 2024

xkcd #927



Fortunately, the charging one has been solved now that we've all standardized on mini-USB. Or is it micro-USB?

Today's Goals

- Characterize devices the landscape of I/O devices.
- Mechanisms for data transfer, interacting with devices, and initiating I/O
- Device drivers and their place in OS structure.
- I/O interfaces for userspace applications.

Device Diversity

- Thus far: lots of focus on one specific I/O type: files & disk.

- Devices we've seen so far are the big / important ones:
  - CPU: Differences in ISA, but overall behavior similar: execute instructions.
  - Memory: Stores data when powered.
  - Disks: have some interesting variations (spinning vs. SSD)

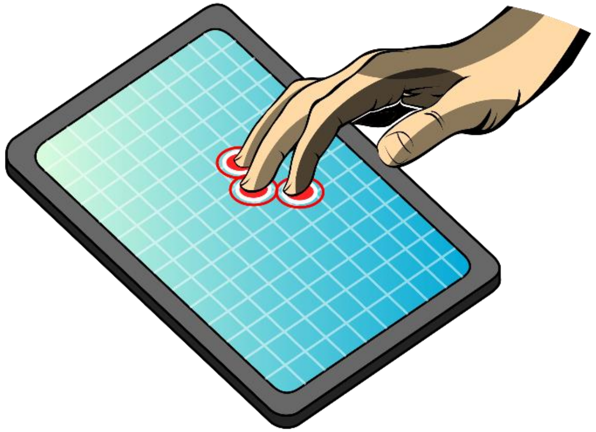
- General I/O takes this to another level!

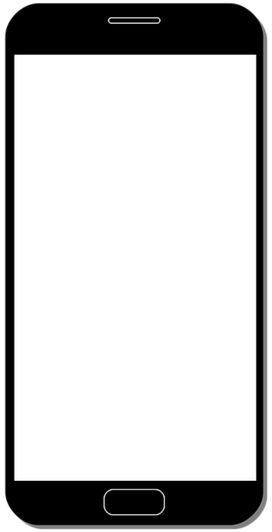
Devices for Machines

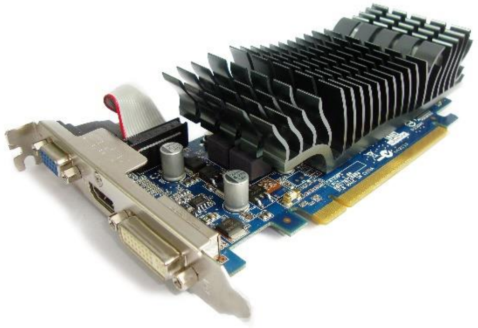
- Some I/O helps machine to talk to other machine devices:

- Most devices are for PEOPLE!

Devices for People



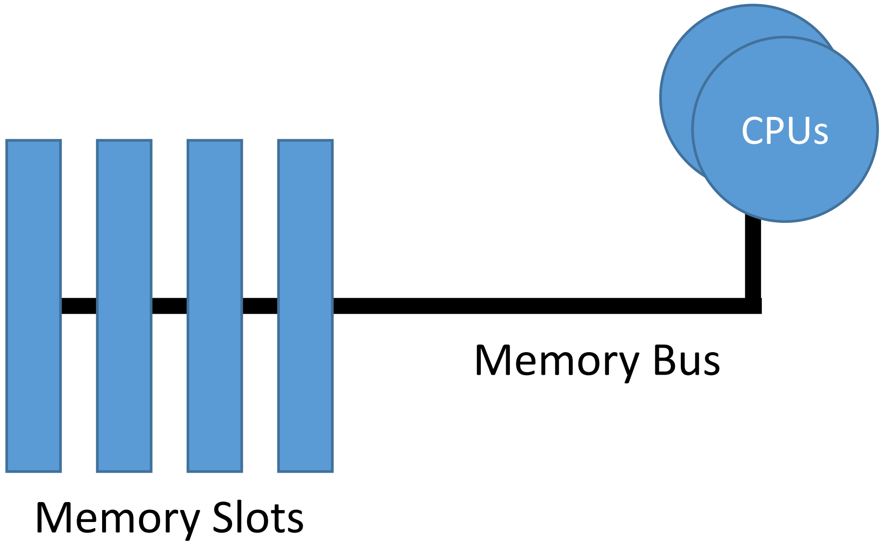




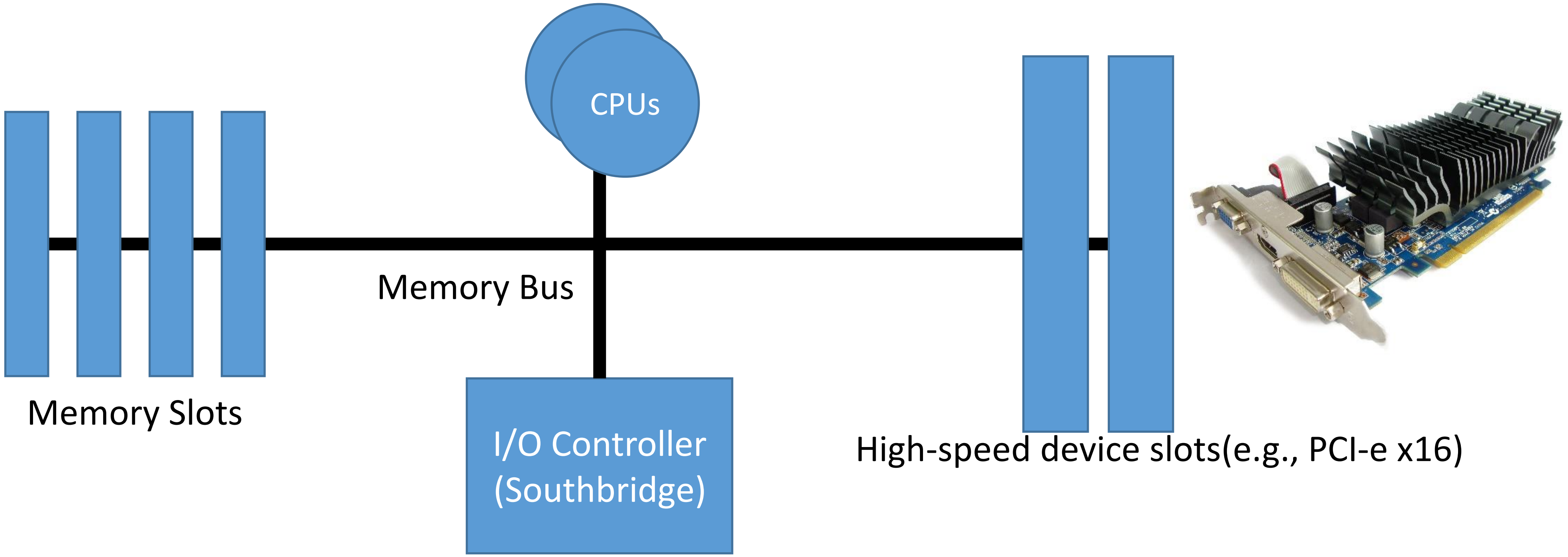
I/O Device Data Rates

Device	Transfer Rate
Keyboard	~ 10 Bytes / sec
Mouse	~ 100 Bytes / sec
…	…
Spinning Hard Disk	~ 100 Megabytes / sec
…	…
Fast Network Card (10 GBE)	~ 1.2 Gigabytes / sec
Graphics Card / GPU	Up to 16 Gigabytes / sec

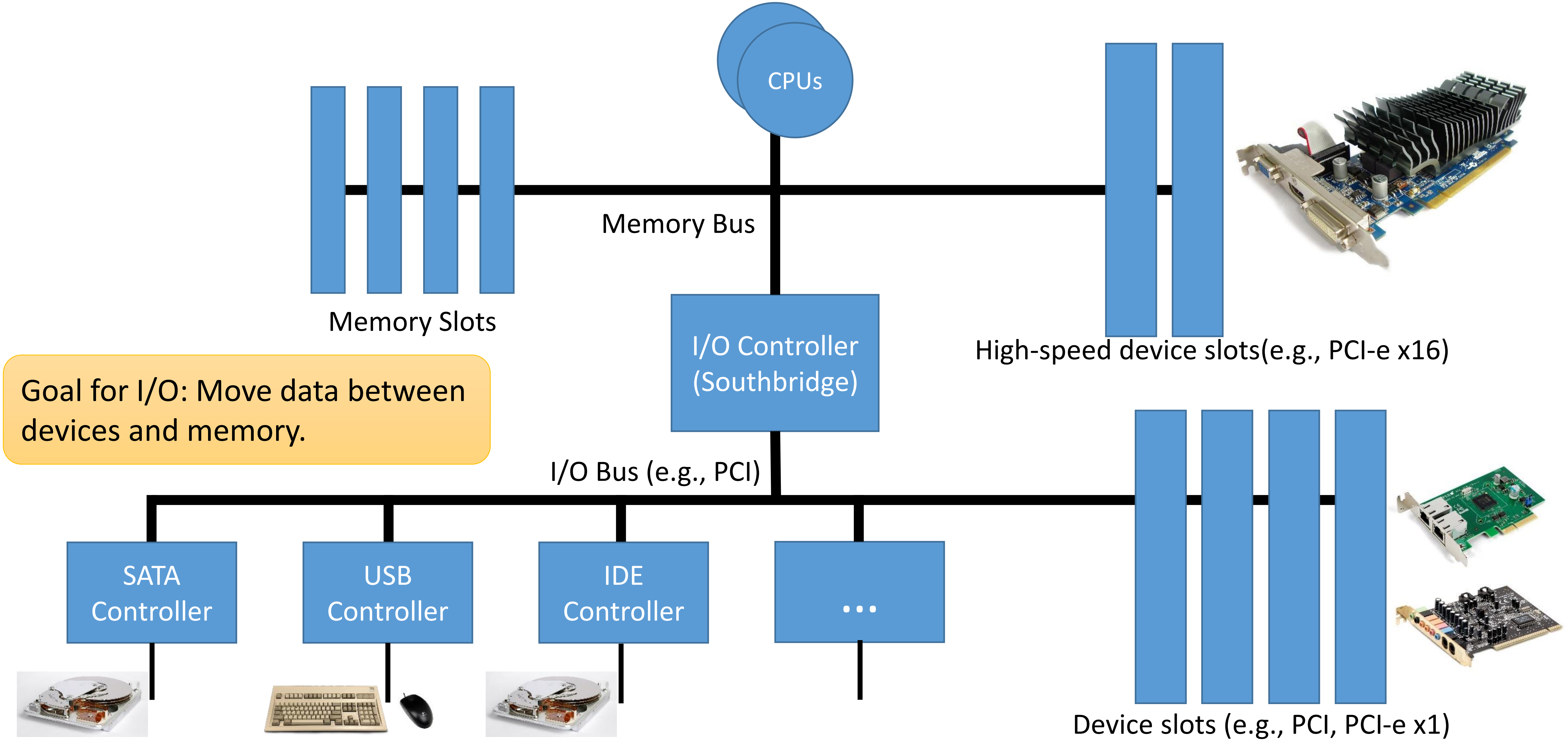
System Hardware and Connections



System Hardware and Connections



System Hardware and Connections



Goal for I/O: Move data between devices and memory.

Three Big I/O Questions

1. Which device should move data?

2. How does the OS communicate with a device (e.g., send it a request)?

3. How does the OS learn when a device has data available?

Which component should be responsible for moving data between the memory and device(s)? Why?

A. The CPU.

B. The memory.

C. The device that has data to move.

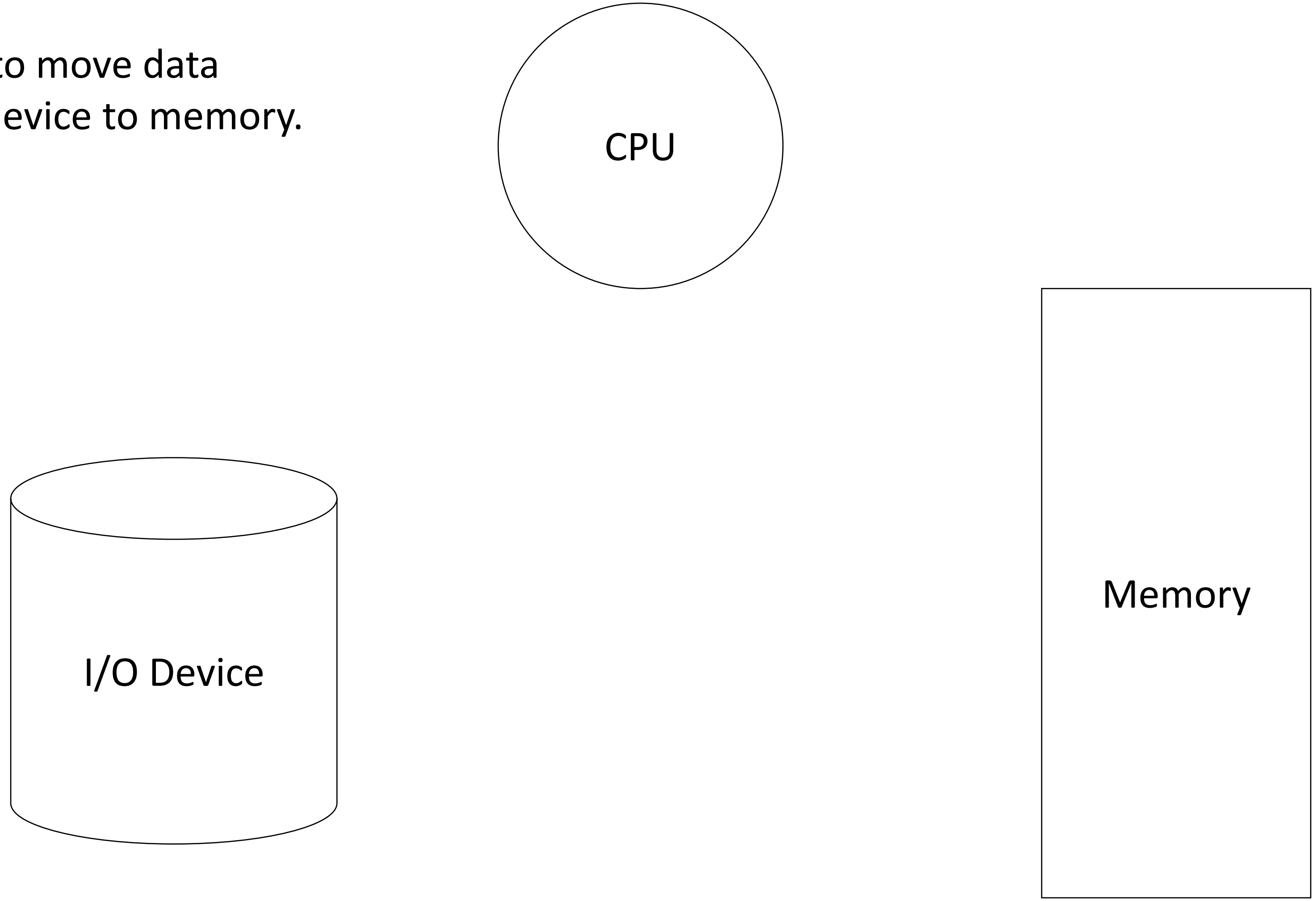
D. Some other component.

Programmed I/O vs. Direct Memory Access

- Programmed I/O (PIO)
    - CPU transfers data between device and memory.
- Direct Memory Access (DMA)
    - Device communicates directly with memory.
- We commonly use both of these methods!

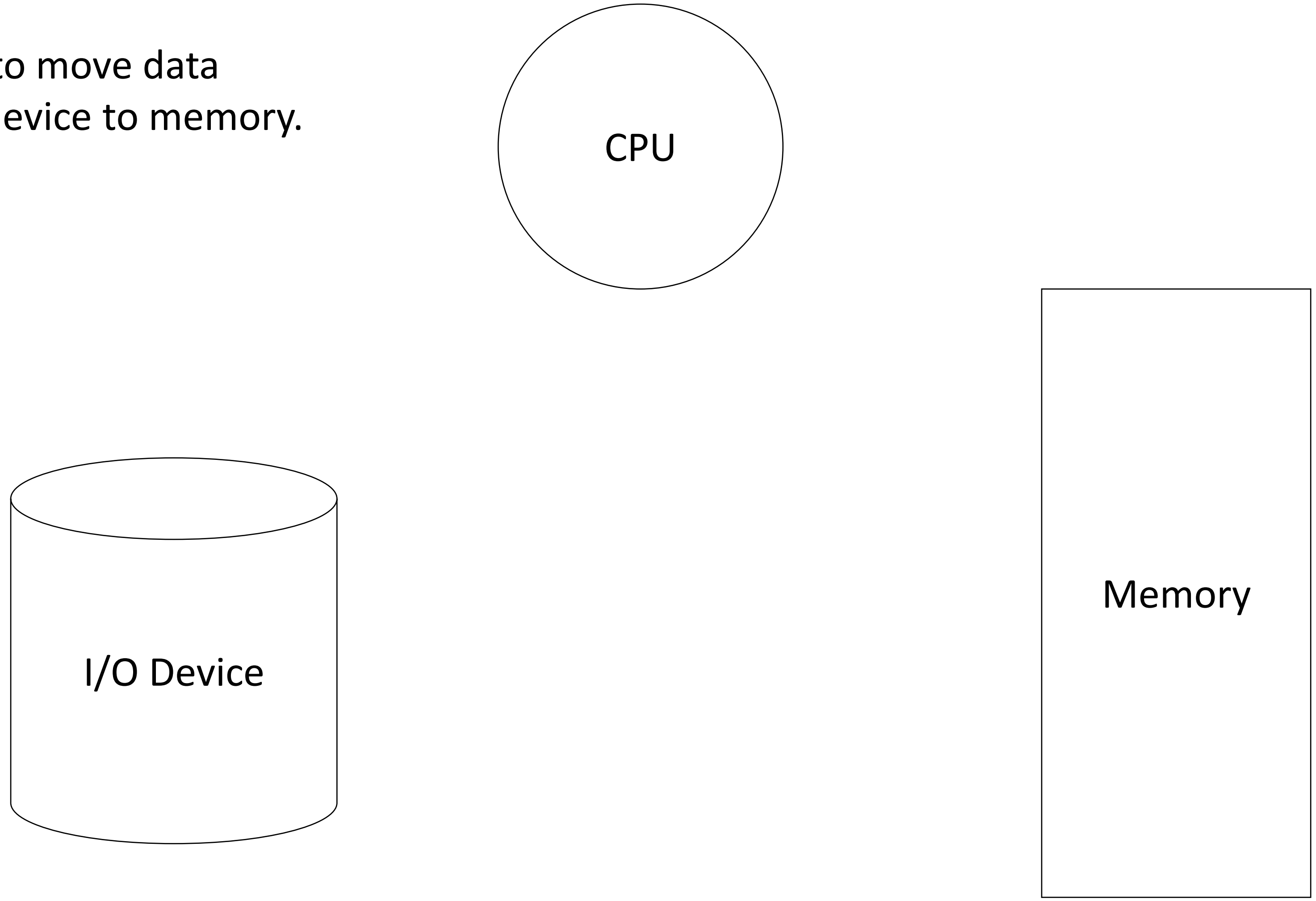
PIO vs. DMA

Want to move data
from device to memory.



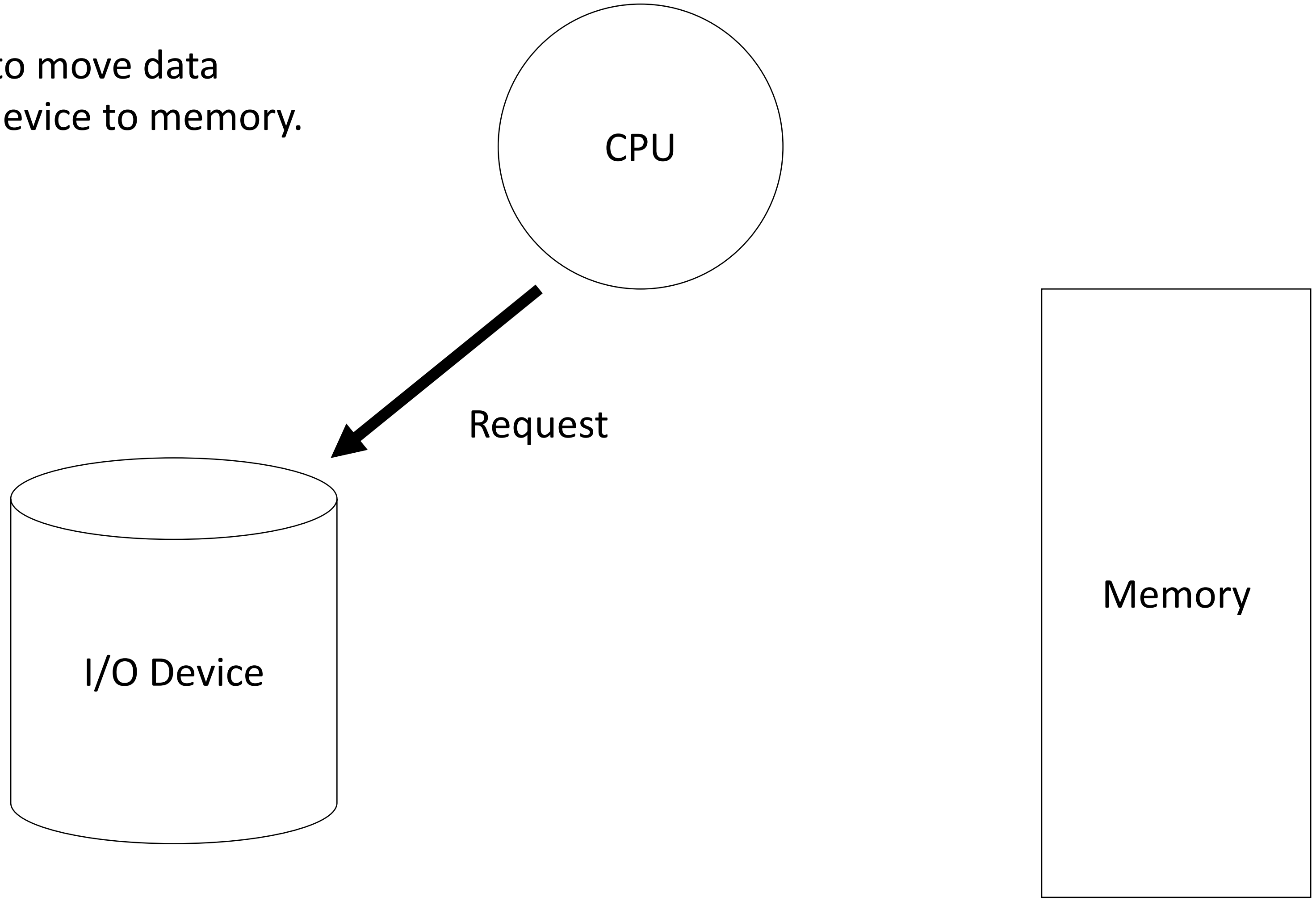
PIO vs. DMA

Want to move data
from device to memory.



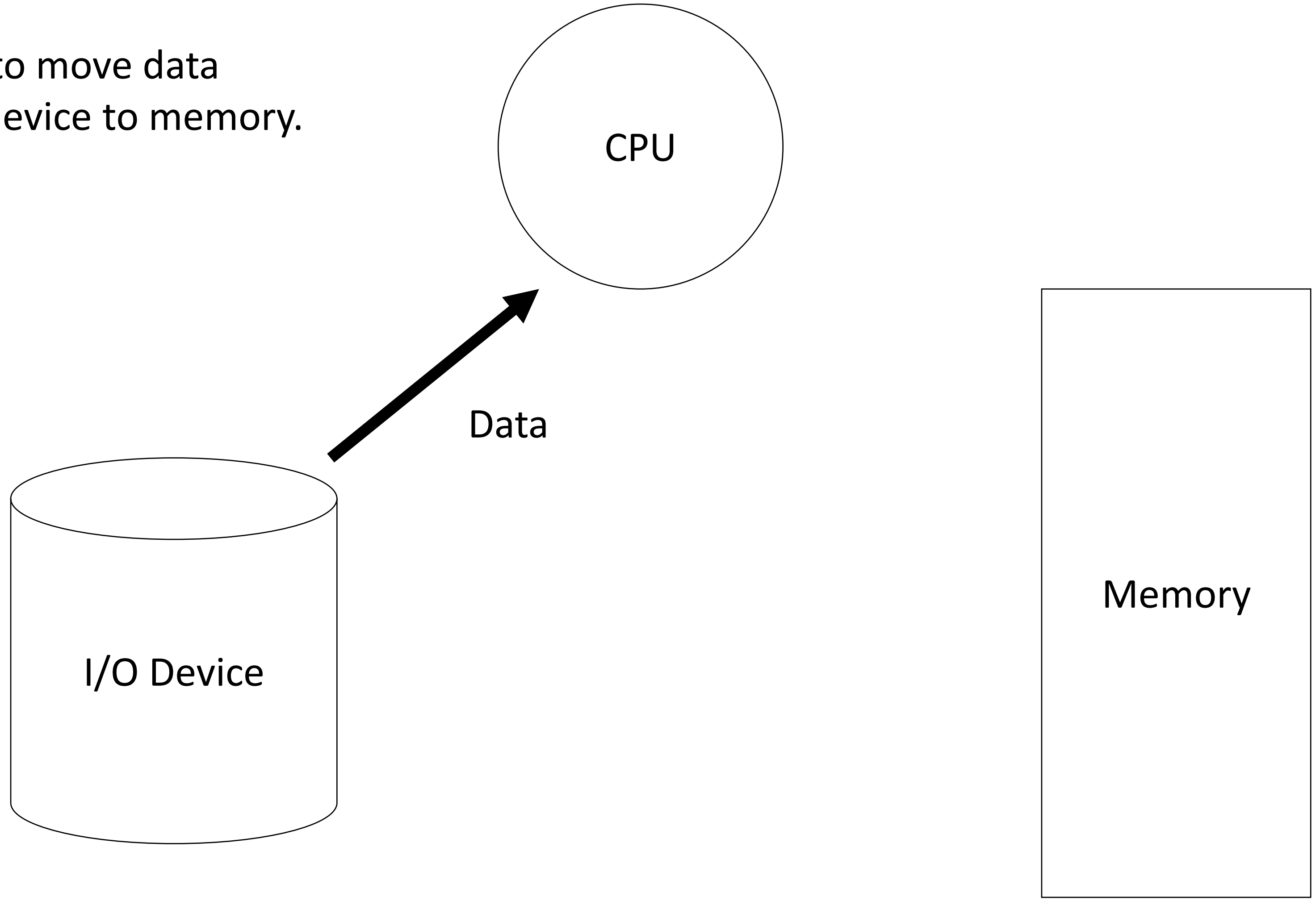
PIO vs. DMA

Want to move data
from device to memory.



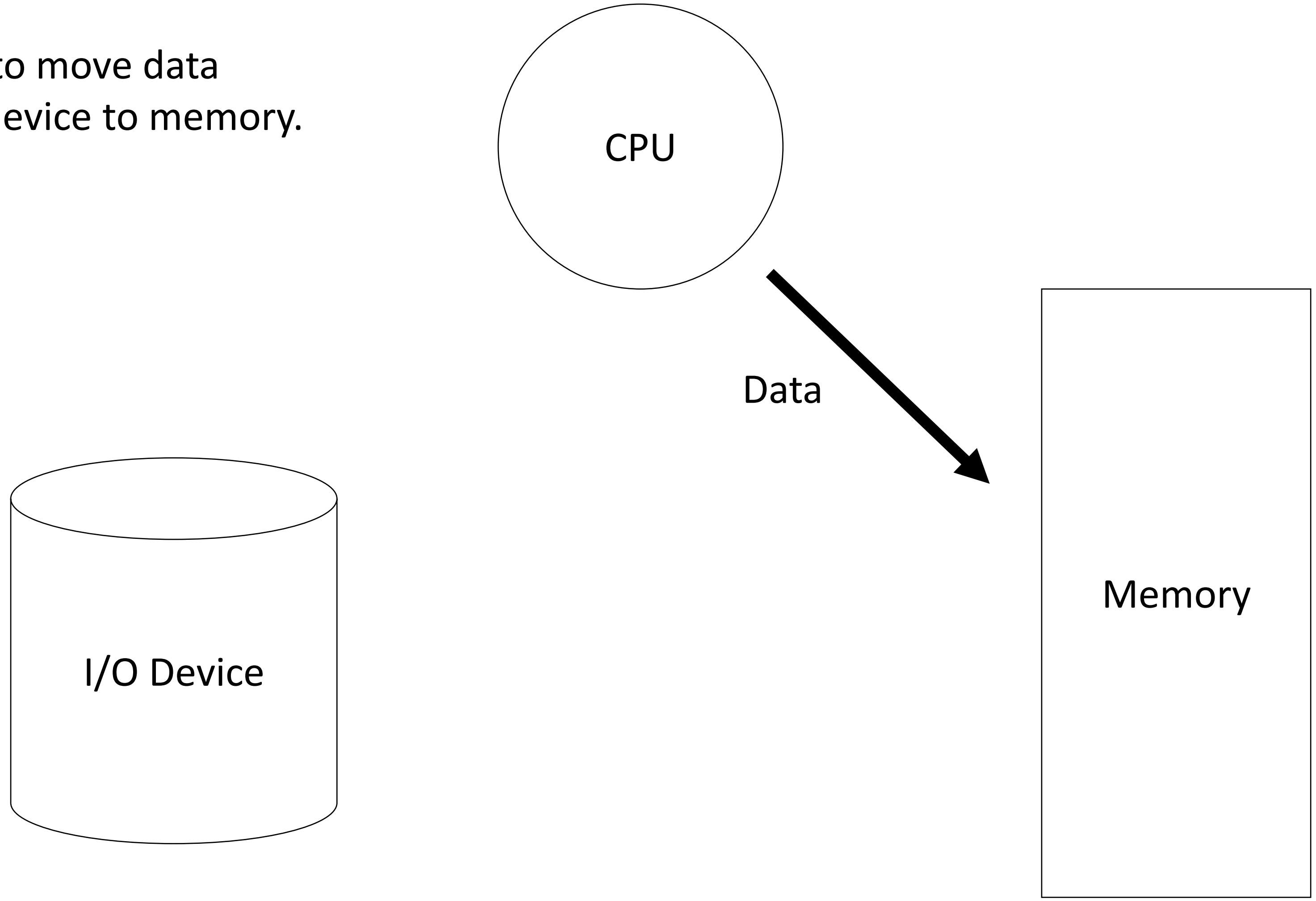
PIO vs. DMA

Want to move data
from device to memory.



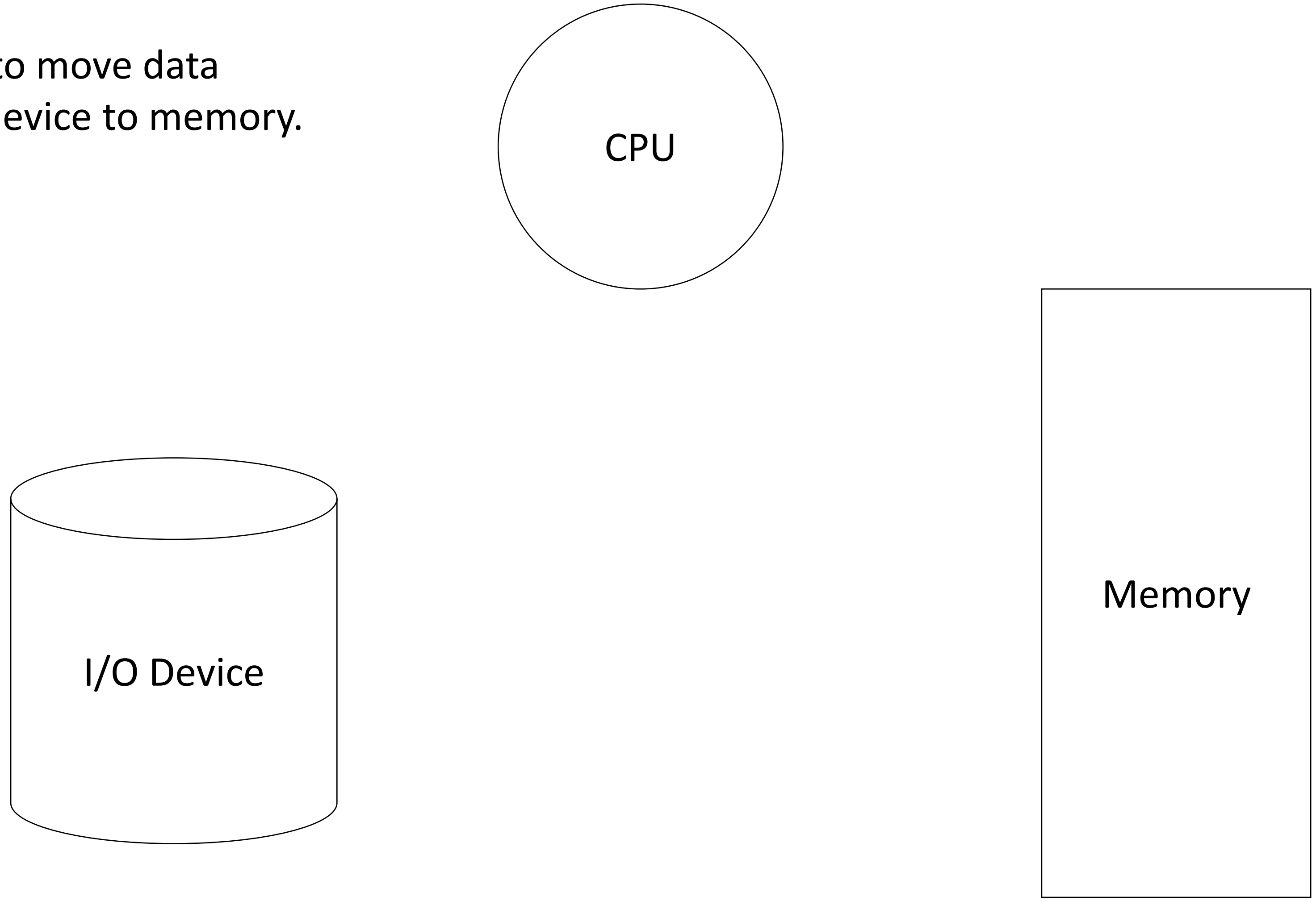
PIO vs. DMA

Want to move data
from device to memory.



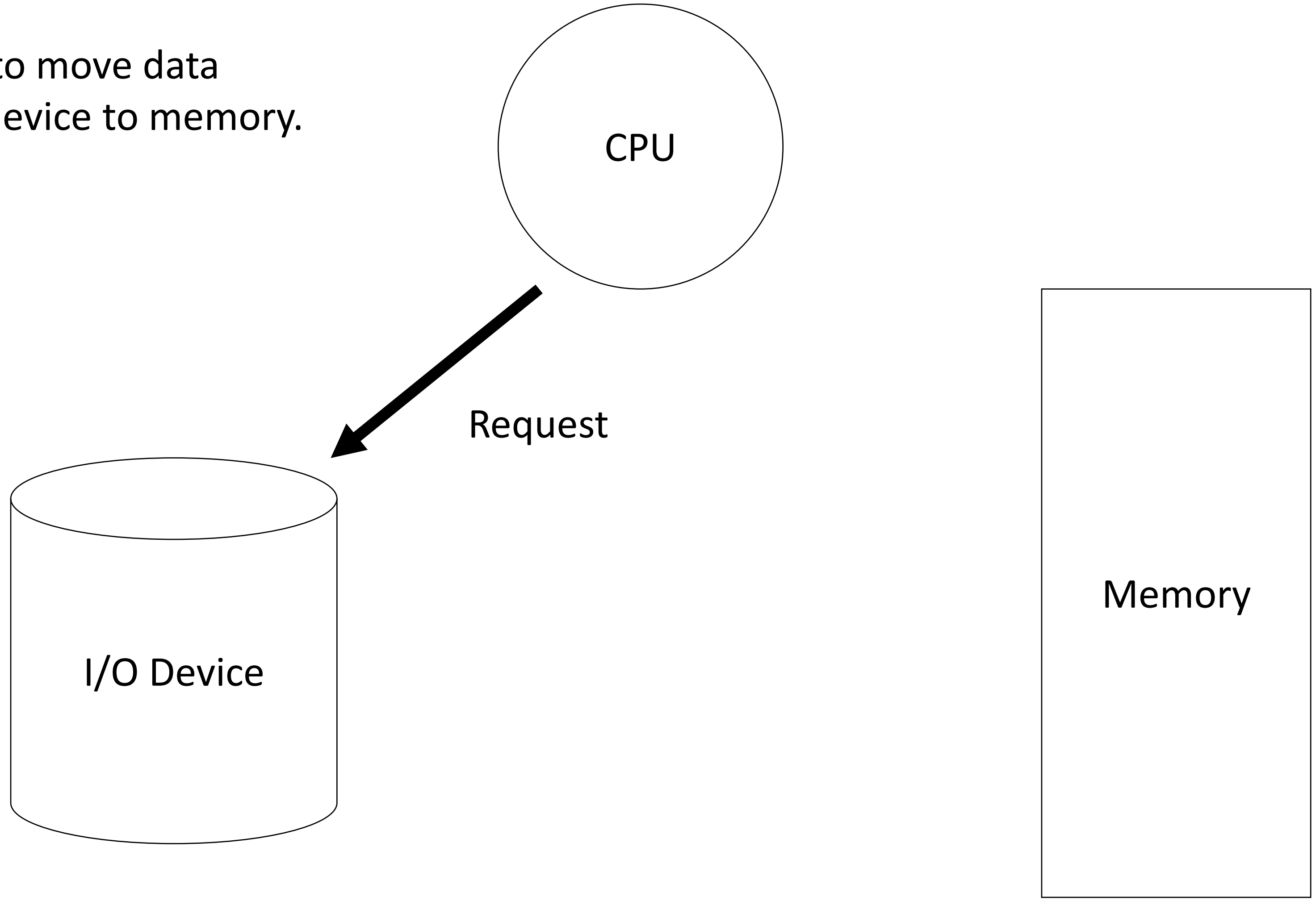
PIO vs. DMA

Want to move data
from device to memory.



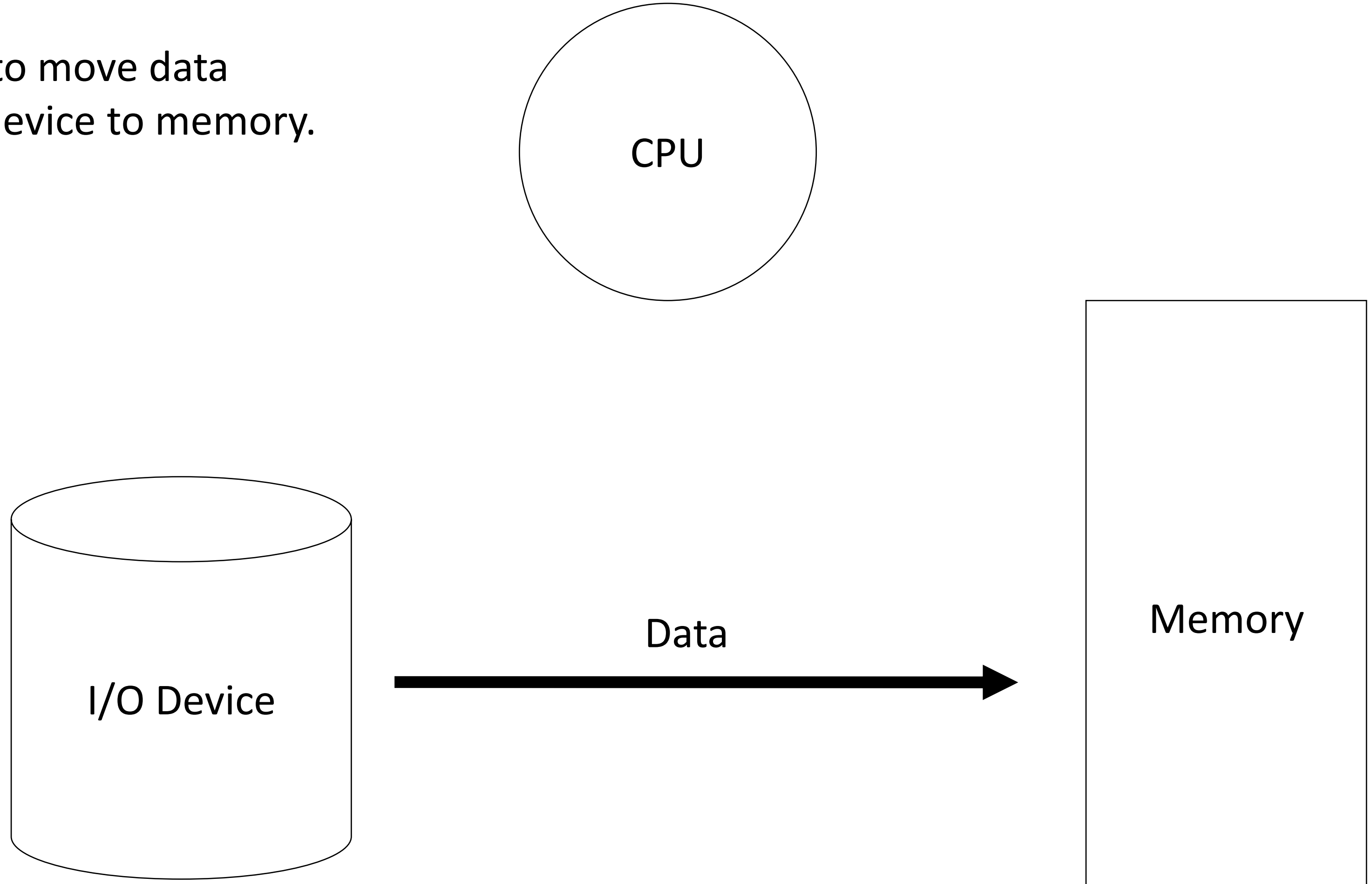
PIO vs. DMA

Want to move data
from device to memory.



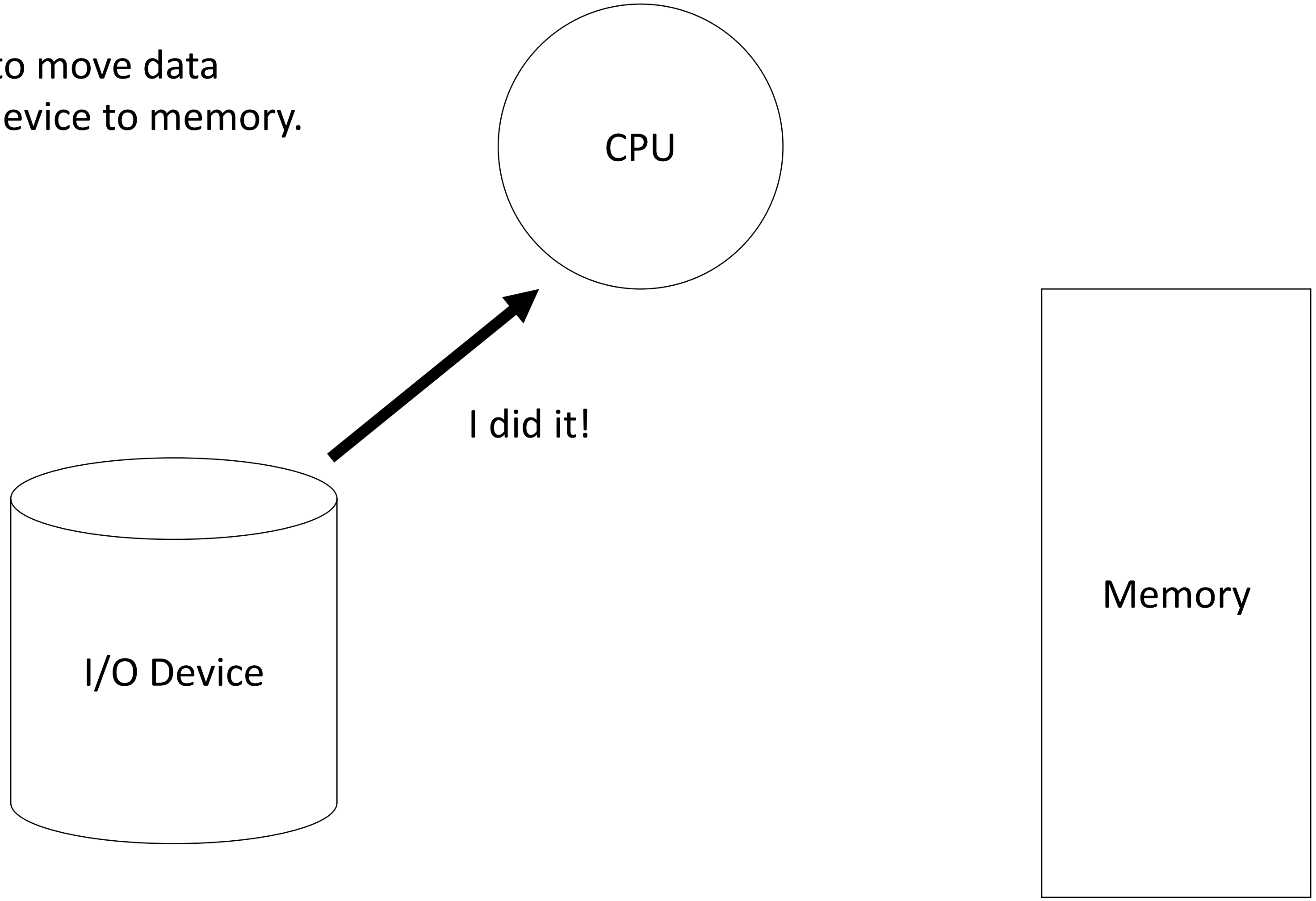
PIO vs. DMA

Want to move data
from device to memory.



PIO vs. DMA

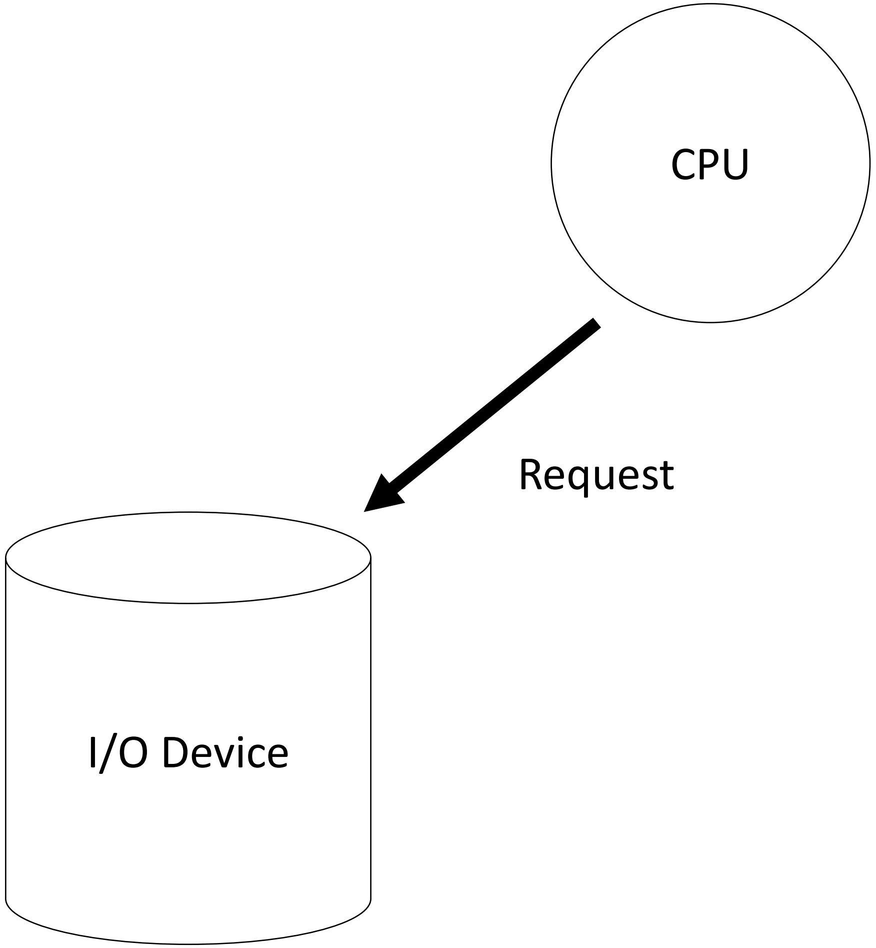
Want to move data
from device to memory.



What types of devices would you expect to use PIO? DMA? Why?

Initiating I/O

- PIO and DMA help us move data.
- How do we tell the device what to do?



How do we tell a device what to do?

- Example: suppose you have an IBM model M keyboard and you want to light up the caps lock light…

How do we tell a device what to do?

- The device has a controller, with some registers. One of the registers controls the status lights.

How can the OS issue a command to a device?

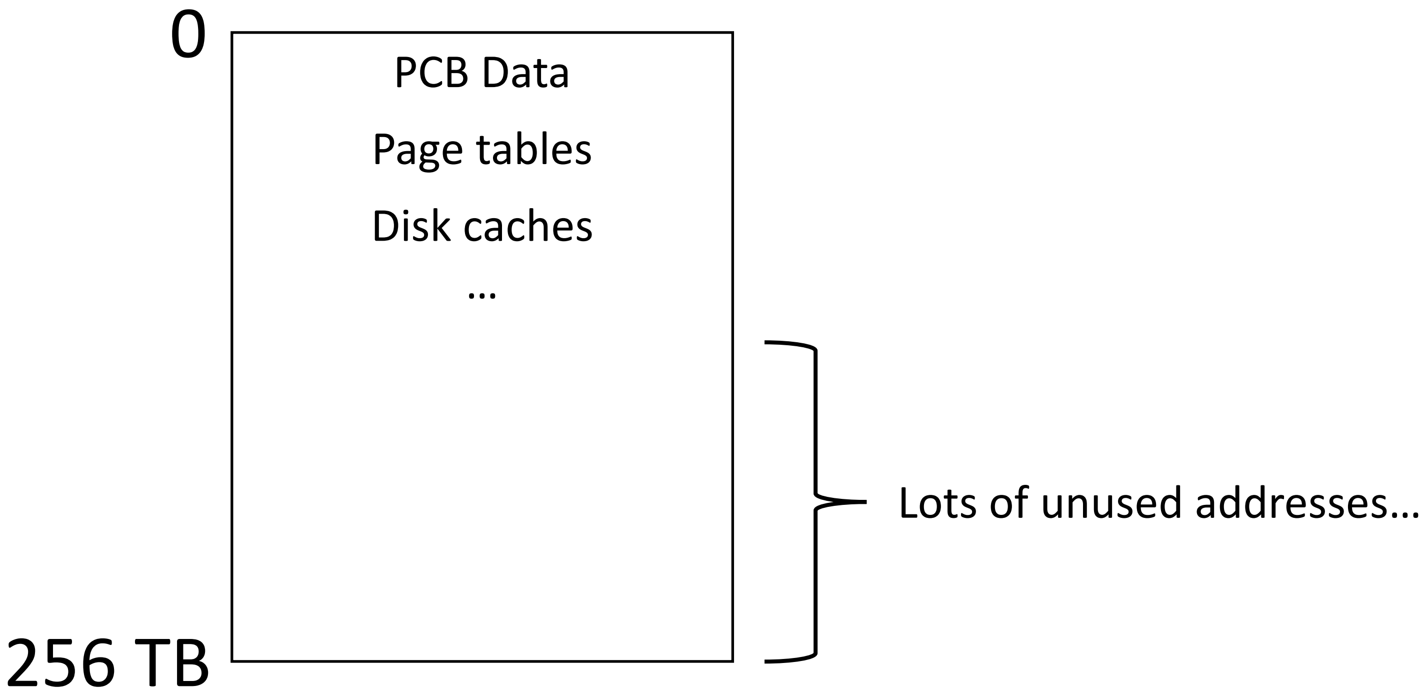
- Suppose you're writing the kernel code. What does it look like?
- How might we make reading/writing those registers available to kernel code?

Interacting with Devices

- Port-mapped I/O
  - Assign each device to a numbered I/O port.
  - Access device's registers through port-based CPU instructions.
- Memory-mapped I/O
  - Logically place device's registers into kernel's address space.
  - Access registers by issuing standard memory load / store instructions.
- Memory-mapped I/O is (almost?) exclusively used today.

Memory-Mapped I/O

- OS kernel's virtual address space is typically much larger than physical memory. (e.g., 48-bit VAS -> address 256 TB)



Memory-Mapped I/O

- OS kernel's virtual address space is typically much larger than physical memory. (e.g., 48-bit VAS -> address 256 TB)

0

PCB Data

Page tables

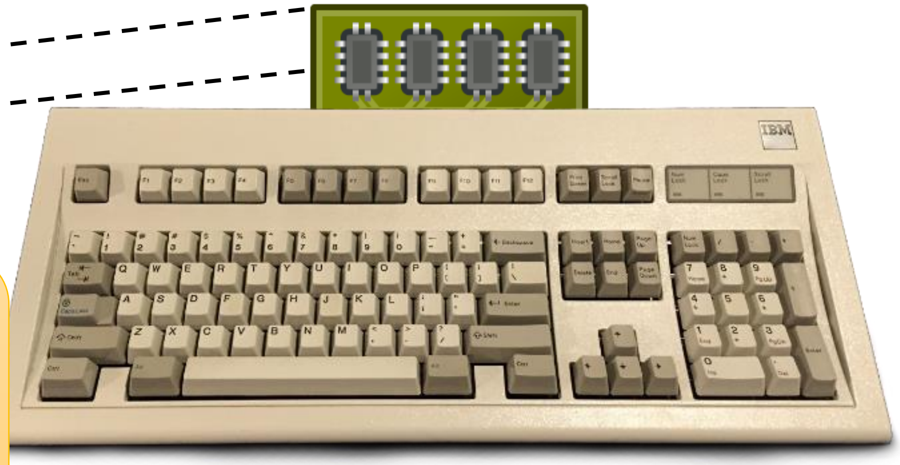
Disk caches

…

Device control registers

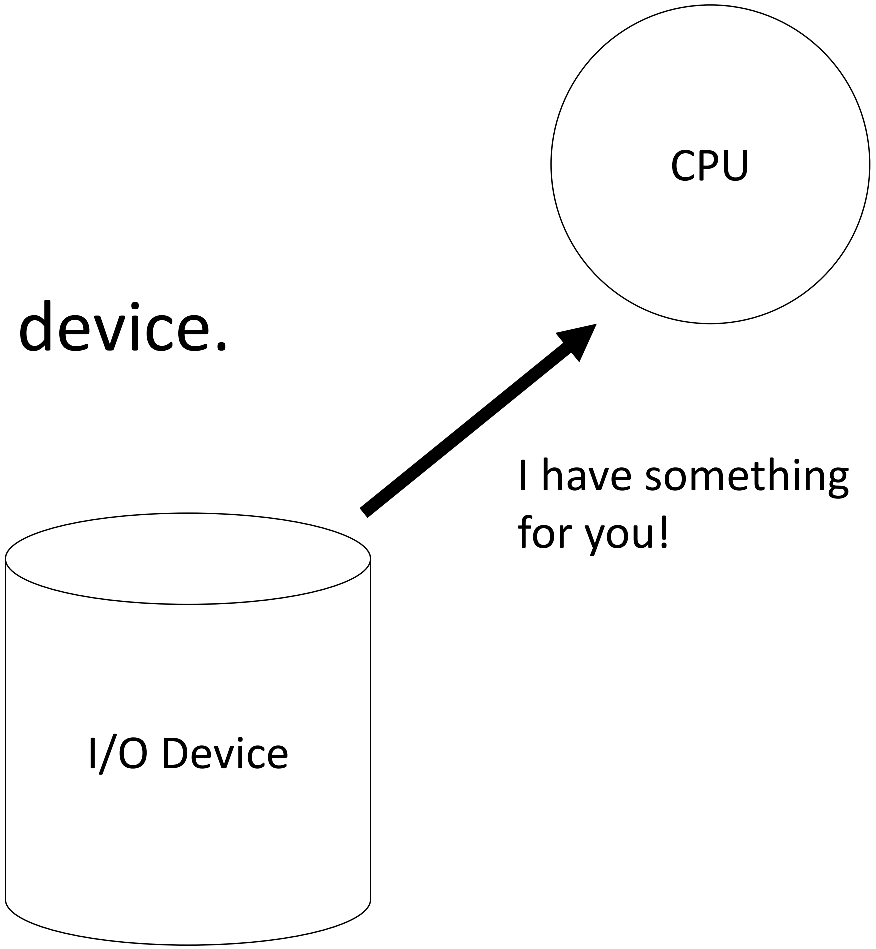
256 TB

Writing to this region of VAS does NOT touch system memory. It changes the device state!



Detecting Available Input

- PIO and DMA help us move data.
- MMIO gives us a mechanism for talking to device.
- How can we determine that a device has data for us?



How should the OS learn that a device has input available? Why? Under what conditions?

A. Ask the device if it has data.

B. Let the device signal the OS if it has data.

C. Learn about data via some other mechanism.

Polling vs. Interrupts

- Polling: periodically ask device "hey, do you have anything for me?"

- Interrupts: device signals CPU to stop what it's doing, context switch to OS so that it can initiate data transfer.

- Most devices use interrupts to avoid wasted polling time.

- In some cases, it might make sense to switch to polling:
  - "Eliminating Receive Livelock in an Interrupt-driven Kernel" (1997)

Interrupt Handlers

- If a device uses interrupts, the OS needs a ‘handler’ for it.
  - Disk: wake up the process that blocked requesting disk access
  - Network: copy newly-received data from device to kernel buffer

- When interrupt signal sent to CPU, determine which device sent it, and which handler to invoke.

- Analogous to system calls, OS keeps a table with unique numbers for each device. Maps device to handler code.

The story so far…

- Lots of decisions to be made in accessing a device:
  - PIO vs. DMA, ports vs. MMIO, polling vs. interrupts
  - Available options depend on hardware support, of course.

- In OS, control for each device implemented in *device driver.*

- *Drivers* often loaded as OS kernel modules.

OS Kernel (core services): Signal handling, I/O system, swapping, scheduling, page replacement, virtual memory	file system: ext4	device driver: USB disk	file system: fat32

Device Drivers

- Executes in OS kernel address space. (except for microkernels)

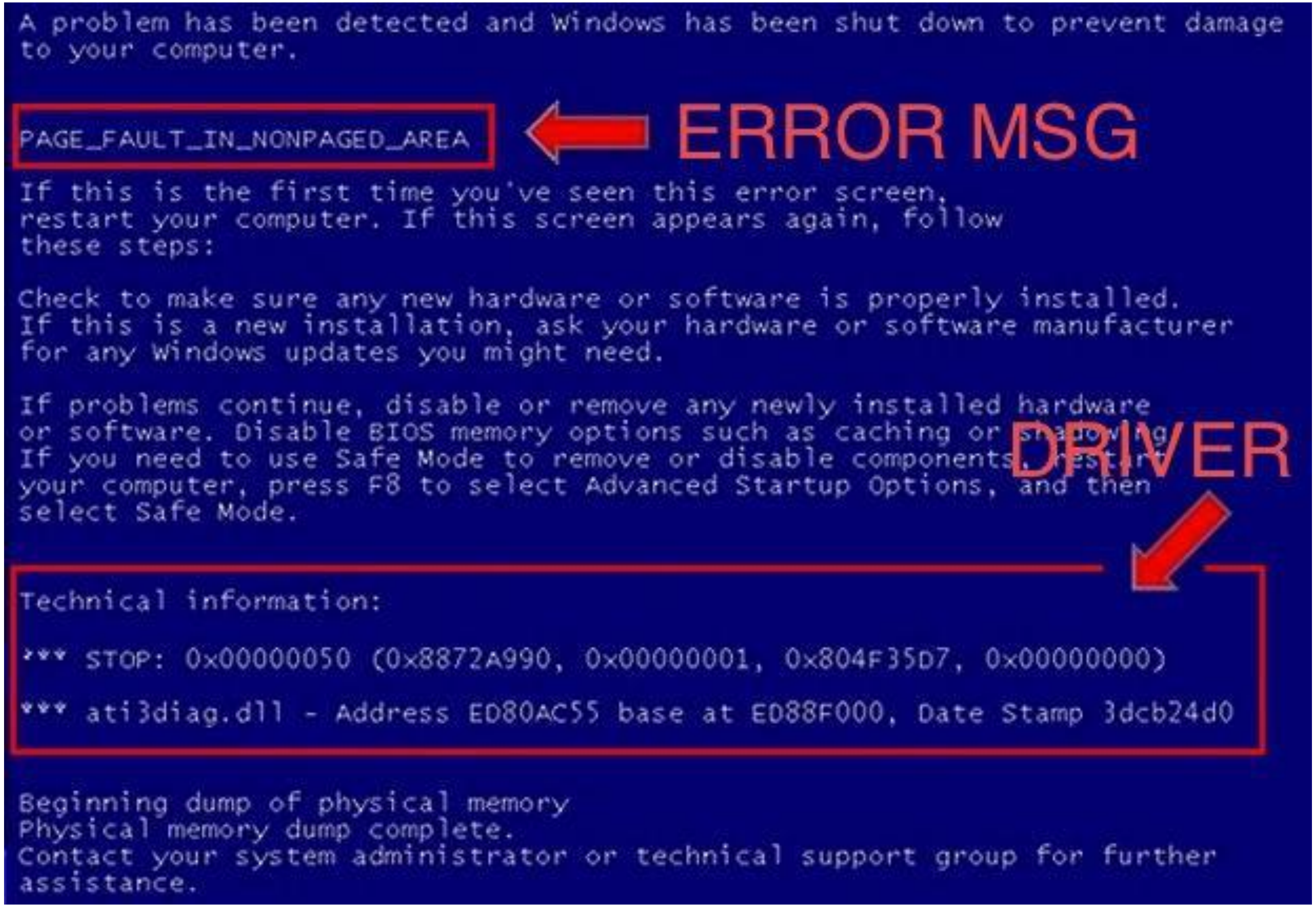
- Defines how to interact with device…
  - Where the device's registers are mapped in memory
  - What type of device it is (how users interact with it)

- Device types: character, block, network
  - Block: disk-like device, typically can do random access, transfers large chunks
  - Character: transfers a stream of characters, data typically consumed when read
  - Network: transfers small data chunks (packets)

Device Drivers

- Executes in OS kernel address space.
  - Hold on… we're extending the trusted kernel code?
- Who writes these…where do they come from?
  - Linux: almost all drivers provided by kernel devs, open source code.
  - Windows: some drivers provided by Microsoft, many by device maker.
  - OS X: somewhere in between. Most from Apple, occasionally device maker.
- Do we trust these drivers? What happens if something goes wrong?

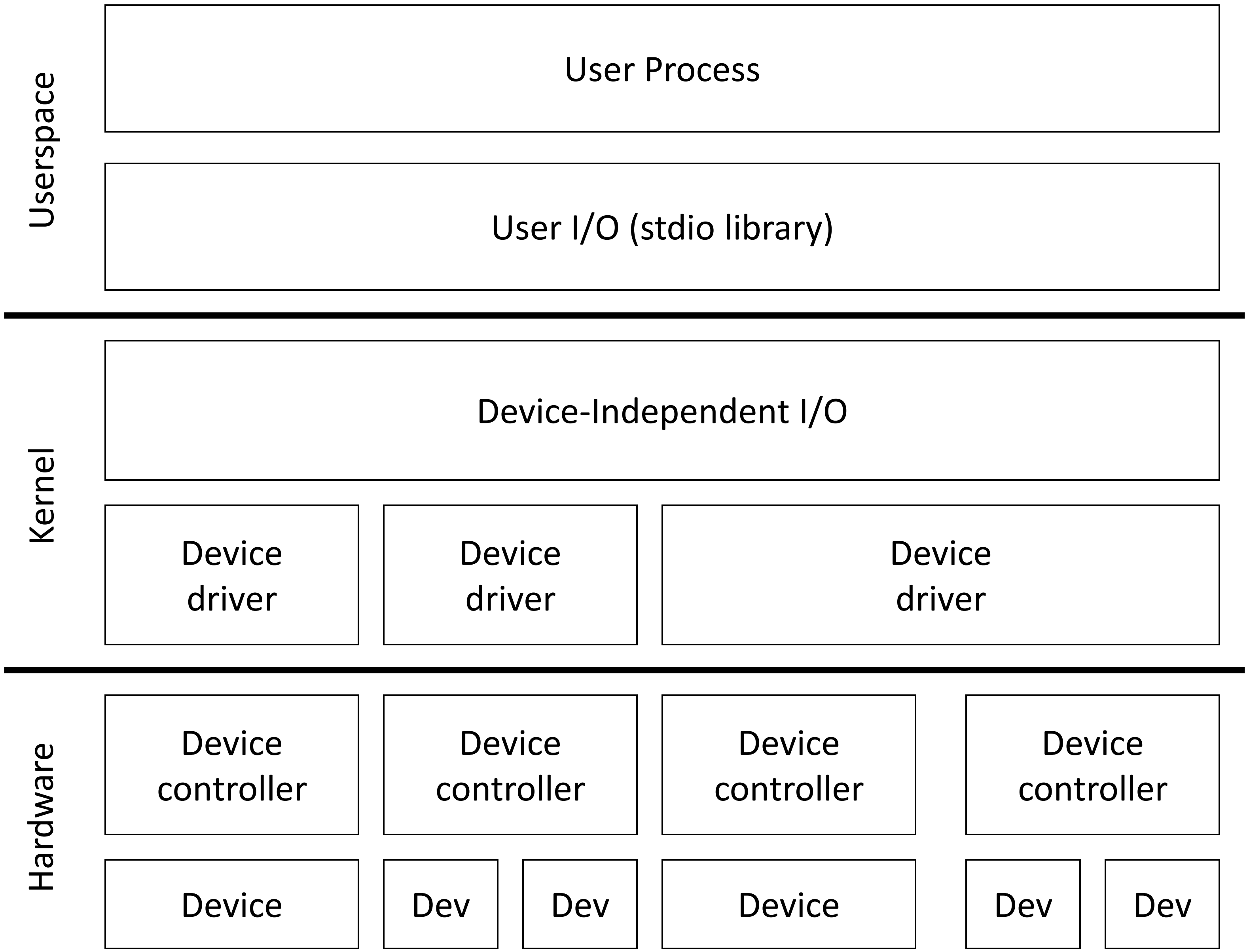
Windows BSOD



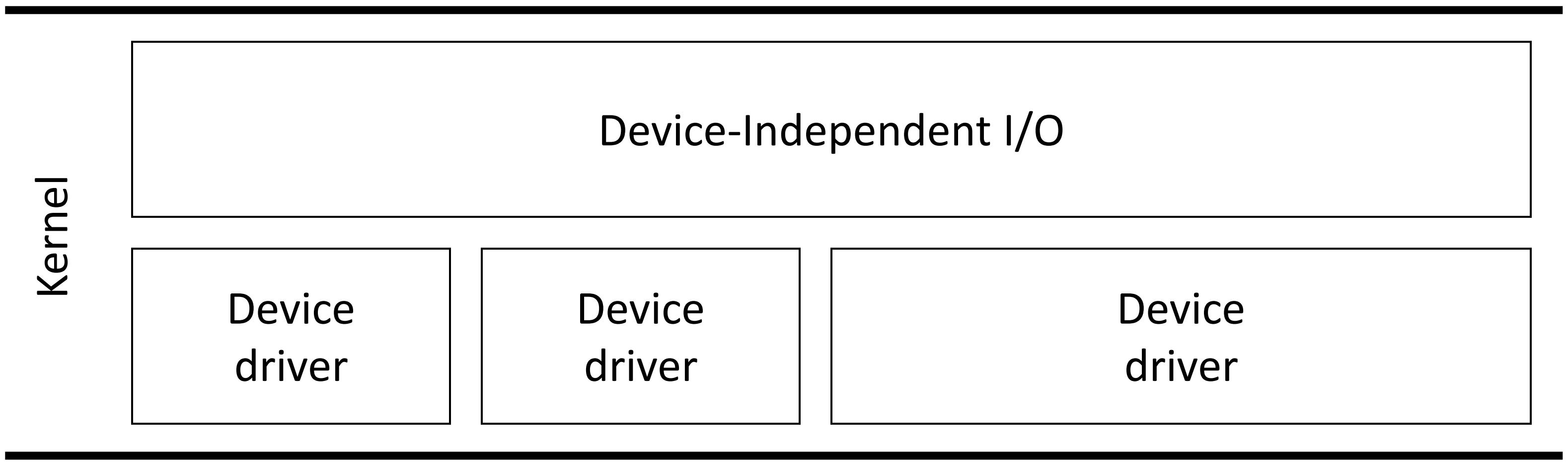
When drivers go bad…

Driver support was a major contributing factor to the Windows Vista "disaster", particularly at launch.

I/O Software Structure: Layered

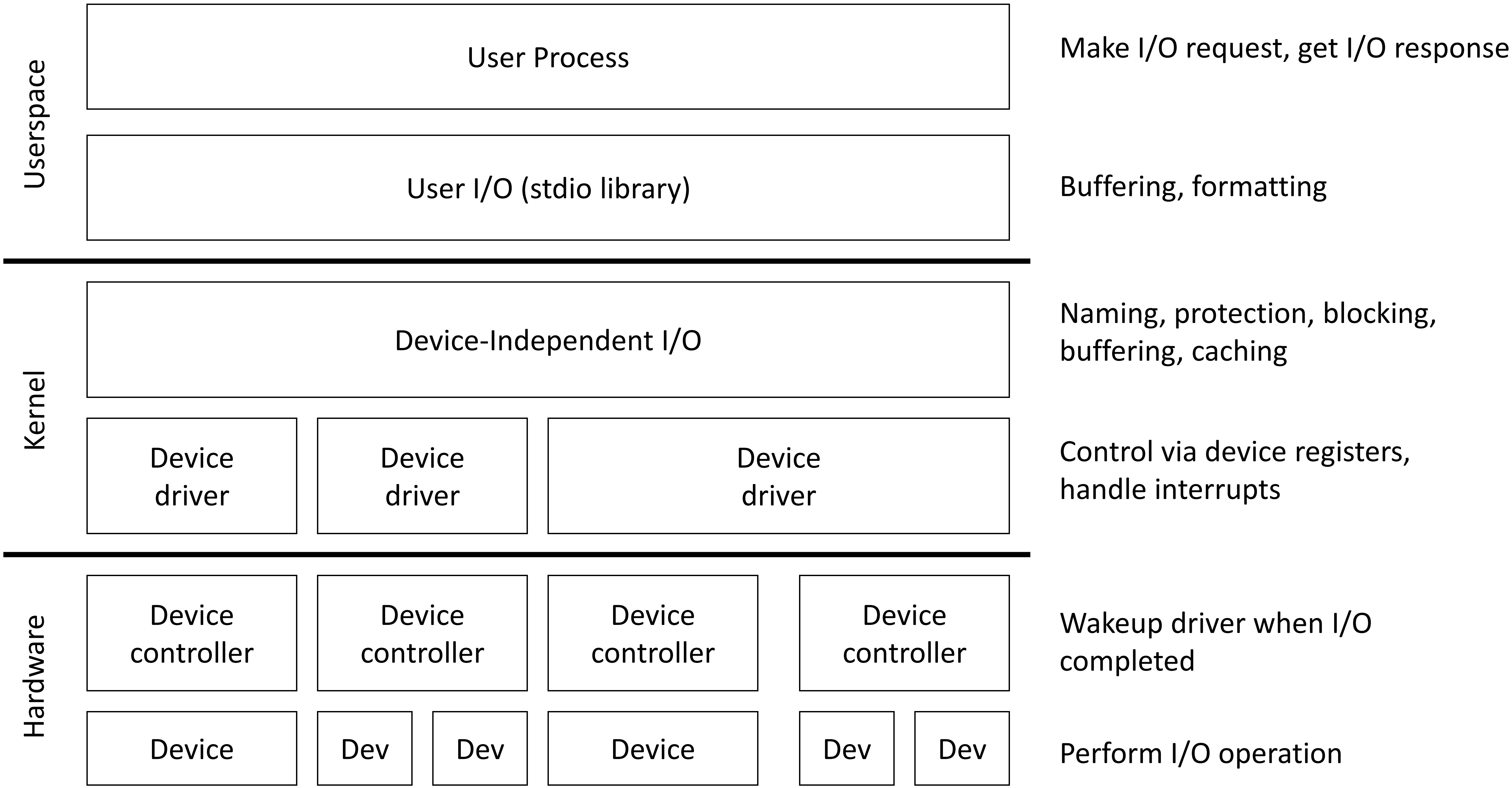


Where would you expect to find an interrupt handler? A caching implementation? Why?

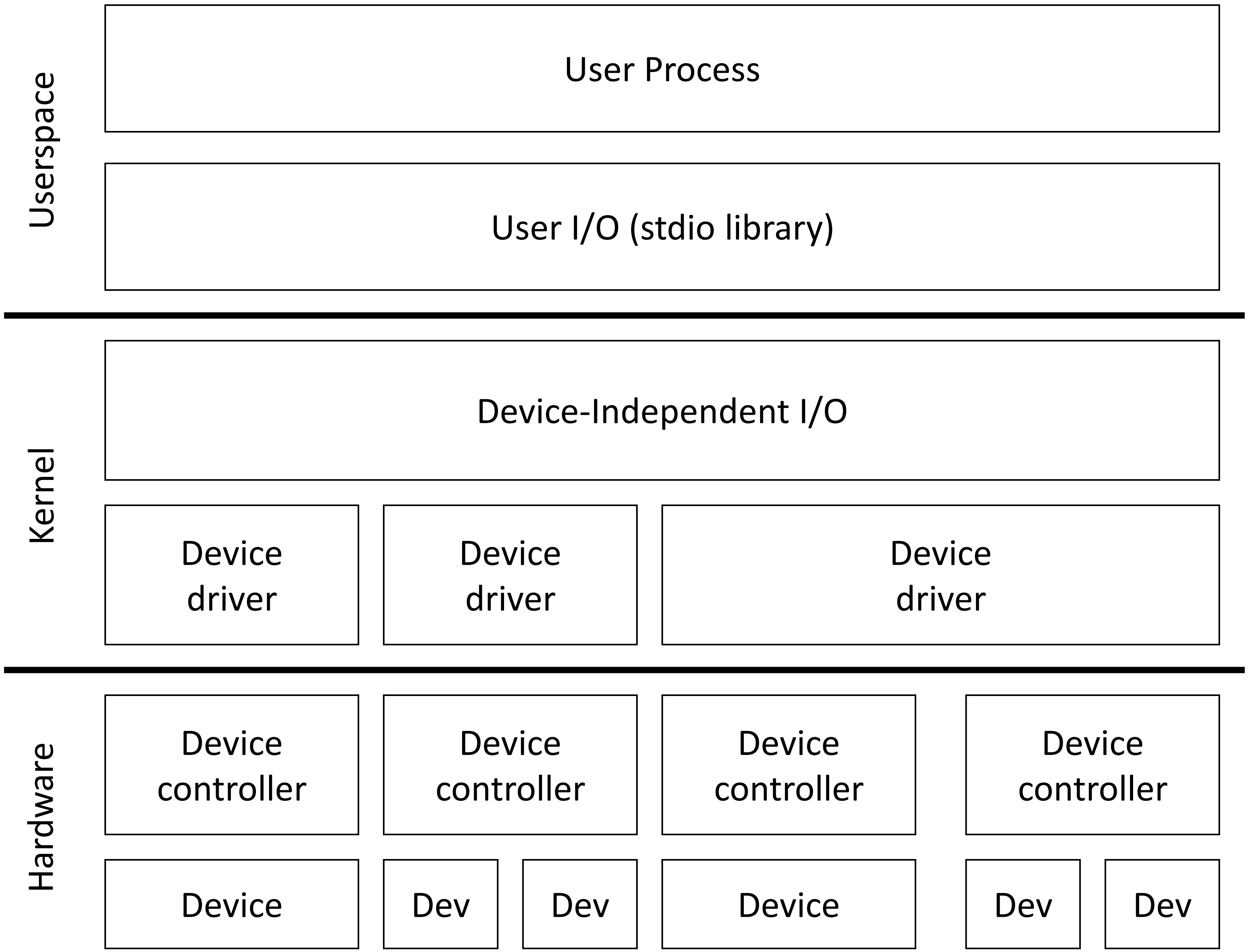


Answer Choice	Interrupt Handler	Caching
A	Device driver	Device driver
B	Device-independent	Device driver
C	Device driver	Device-independent
D	Device-independent	Device-independent

I/O Software Structure: Layered



I/O Software Structure: Layered



Make I/O request, get I/O response

Buffering, formatting

Naming, protection, blocking, buffering, caching

Control via device registers, handle interrupts

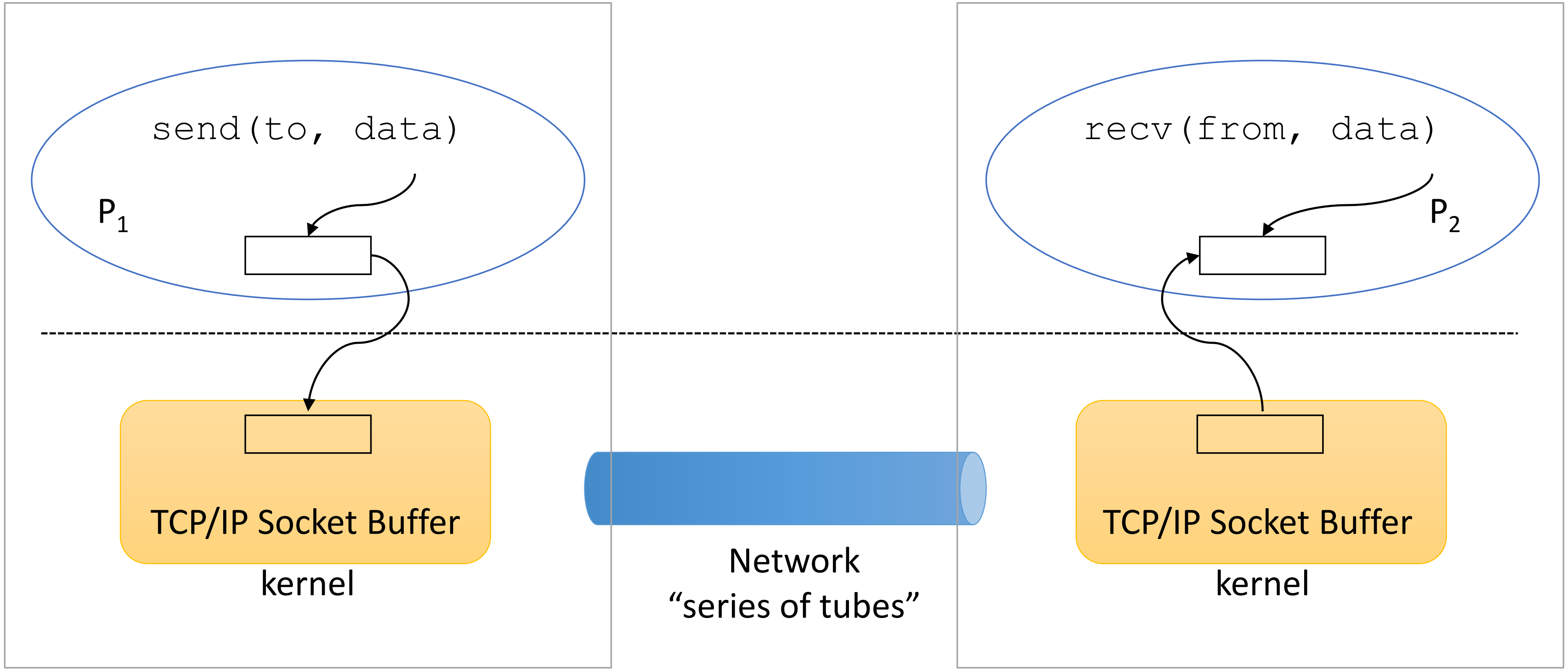
Wakeup driver when I/O completed

Perform I/O operation

(From IPC Slides)

Recall: OS Buffering

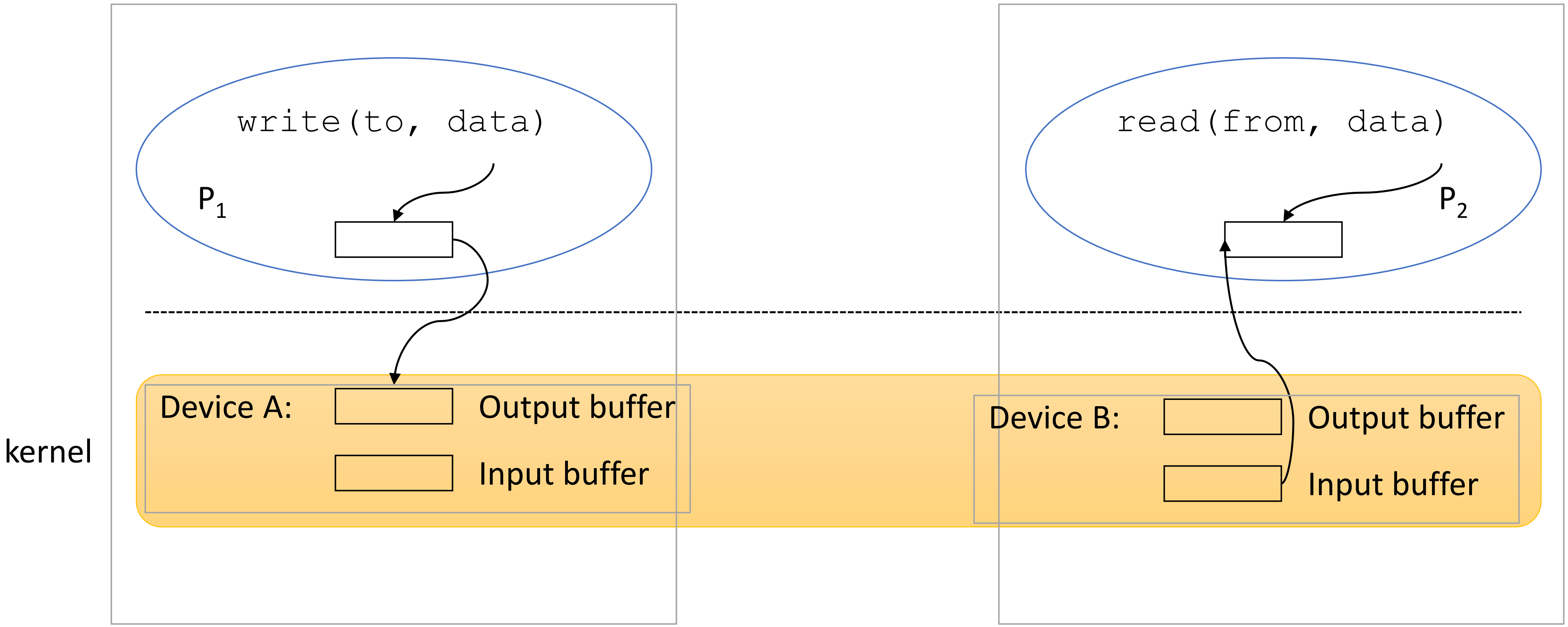
Kernel's buffers have finite storage space!



If sender fills buffer, OS will mark the process as blocked – can't be scheduled until space is free.

If the buffer is empty, OS will mark the receiver process as blocked – can't read data before it arrives!

General Device Buffering



Why should the OS buffer data to/from devices?

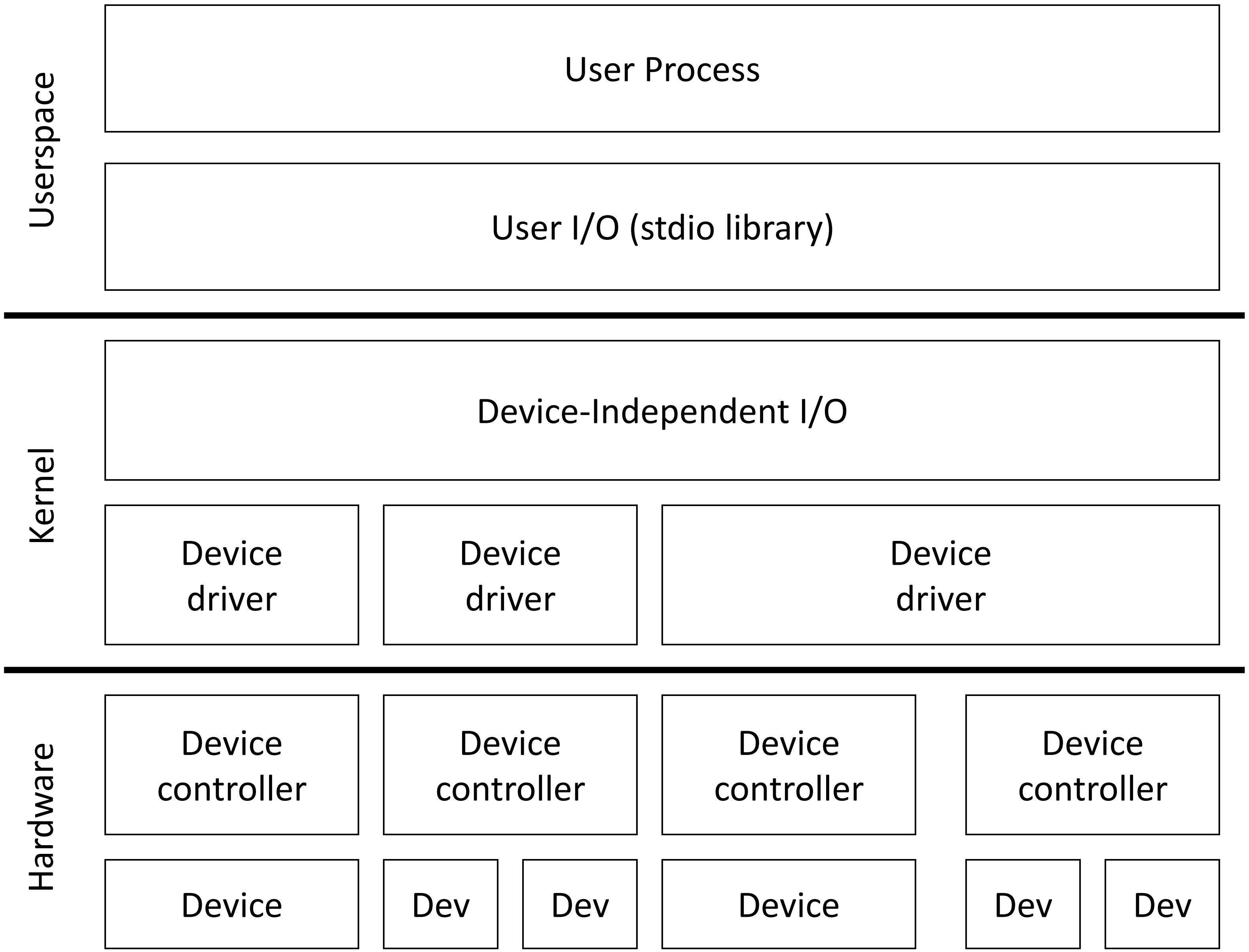
Why OS Buffering?

- Speed mismatch between device and user process.
  - analogous to producer/consumer problem
  - store produced items (data) in buffer, smooth out bursty requests

- Data transfer size mismatch between device and user process.
  - e.g., receive large chunk of data from network, user only wants a few bytes

- DMA requires large, aligned, contiguous chunks of reserved memory.
  - better not to rely on the user to set that up…

I/O Software Structure: Layered



Make I/O request, get I/O response

Buffering, formatting

Naming, protection, blocking, buffering, caching

Control via device registers, handle interrupts

Wakeup driver when I/O completed

Perform I/O operation

Unix: I/O System Calls

- For most devices, uniform access via file system interface:
  - `fd = open("/dev/devname", …);`
  - `bytes_read = read(fd, buf, count);`
  - `bytes_written = write(fd, buf, count);`
  - `ioctl(fd, request, …);`
  - `close(fd);`

- Notable exceptions:
  - Devices that userspace can't access directly (e.g., timers used for scheduling).
  - Network adapters – FS interface awkward due to addressing.

What is `ioctl()`?

- ioctl is a “catch-all” for all I/O commands that are not open/close/read/write.

- It takes a “request” parameter that the I/O device then translates into the actual command it executes.
  - e.g., asking a USB device its transfer rate (USB 1 vs. USB 2)
  - e.g., instructing CD-ROM device to eject the disc.

```
#include <sys/ioctl.h>
    int ioctl(int fd, unsigned long request, ...);
```

Common I/O Interface

- Processes can mix and match descriptors (e.g., shell redirection).
- OS can cache / buffer / protect data in device-independent way.
- Easier for humans to remember!

“Synchronous” I/O

- For most devices, uniform access via file system interface:
  - `fd = open(“/dev/devname”, …);`
  - `bytes_read = read(fd, buf, count);`
  - `bytes_written = write(fd, buf, count);`
  - `ioctl(fd, request, …);`
  - `close(fd);`
- The functions above are synchronous: when the user calls them, they need them to happen now. Can’t make progress until they’re done.

This is the most common form of I/O, and it’s what we’ve been assuming all along:
You perform I/O, which is slow, so you have to block while you wait!

Alternative: Asynchronous I/O

```
The POSIX AIO interface consists of the following functions:

aio_read(3)      Enqueue  a read request.  This is the asynchronous ana□
                 log of read(2).

aio_write(3)     Enqueue a write request.  This is the asynchronous ana□
                 log of write(2).

aio_fsync(3)     Enqueue a sync request for the I/O operations on a file
                 descriptor.  This  is  the  asynchronous  analog  of
                 fsync(2) and fdatasync(2).

aio_error(3)     Obtain the error status of an enqueued I/O request.

aio_return(3)    Obtain the return status of a completed I/O request.

aio_suspend(3)   Suspend the caller until one or more of a specified set
                 of I/O requests completes.

aio_cancel(3)    Attempt to cancel outstanding I/O requests on a  speci□
                 fied file descriptor.

lio_listio(3)    Enqueue  multiple  I/O requests using a single function
                 call.
```

Alternative: Asynchronous I/O

```
The POSIX AIO interface consists of the following functions:

aio_read(3)     Enqueue  a read request.  This is the asynchronous ana□
                log of read(2).

aio_write(3)    Enqueue a write request.  This is the asynchronous ana□
                log of write(2).
```

Issue a read or write request in the background, but don’t block waiting for it. In the mean time, process can continue working on other things.

Notification options when request completes:

1. Do nothing, my process will check later.
2. Send my process a signal.
3. Start a new thread in my process with the specified function.

Alternative: Asynchronous I/O

```
The POSIX AIO interface consists of the following functions:

aio_read(3)     Enqueue  a read request.  This is the asynchronous ana□
                log of read(2).

aio_write(3)    Enqueue a write request.  This is the asynchronous ana□
                log of write(2).

aio_fsync(3)    Enqueue a sync request for the I/O operations on a file
                descriptor.  This  is  the  asynchronous  analog  of
                fsync(2) and fdatasync(2).

aio_error(3)    Obtain the error status of an enqueued I/O request.

aio_return(3)   Obtain the return status of a completed I/O request.

aio_suspend(3)  Suspend the caller until one or more of a specified set
                of I/O requests completes.

aio_cancel(3)   Attempt to cancel outstanding I/O requests on a  speci□
                fied file descriptor.

lio_listio(3)   Enqueue  multiple  I/O requests using a single function
                call.
```

Summary

- Huge diversity in I/O devices, with many different characteristics.
- Many choices in interacting with devices:
  - Programmed I/O (PIO) vs. Direct Memory Access (DMA)
  - Port-mapped I/O vs. Memory-mapped I/O
  - Polling vs. Interrupts
- Devices controlled by OS driver code.
- I/O interface usually synchronous, alternatives for asynch exist.