Operating Systems

Drivers and IO

Hongliang Liang, BUPT

Sep. 2023

Topics covered in this lecture

- How does the OS interact with devices
- Design of device drivers
- Types of IO devices

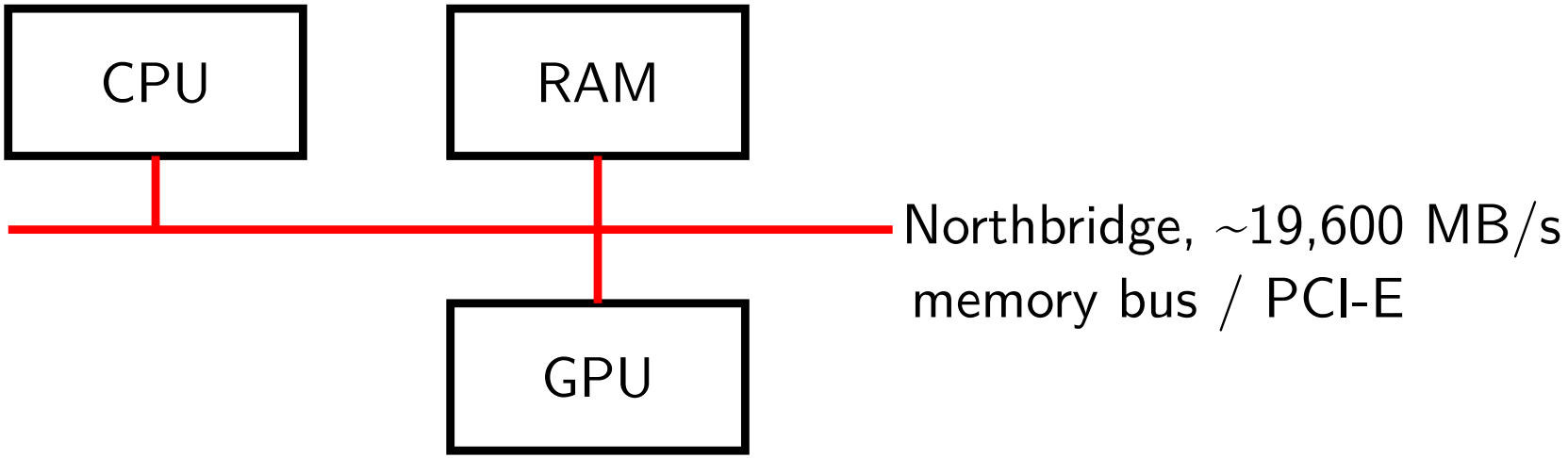
Motivation

- So far we have talked about the CPU and about RAM
- How do we get data into RAM?
  - Load programs and data from storage
  - Read and write packets from the network (maybe even streams?)
  - Write data to a terminal or the screen
  - Read data from input devices such as keyboard/mouse/camera
- Devices provide input/output (IO) to a system
- IO allows information to *persist* (RAM is volatile)!
- Enables interesting computation!

Modern device interface

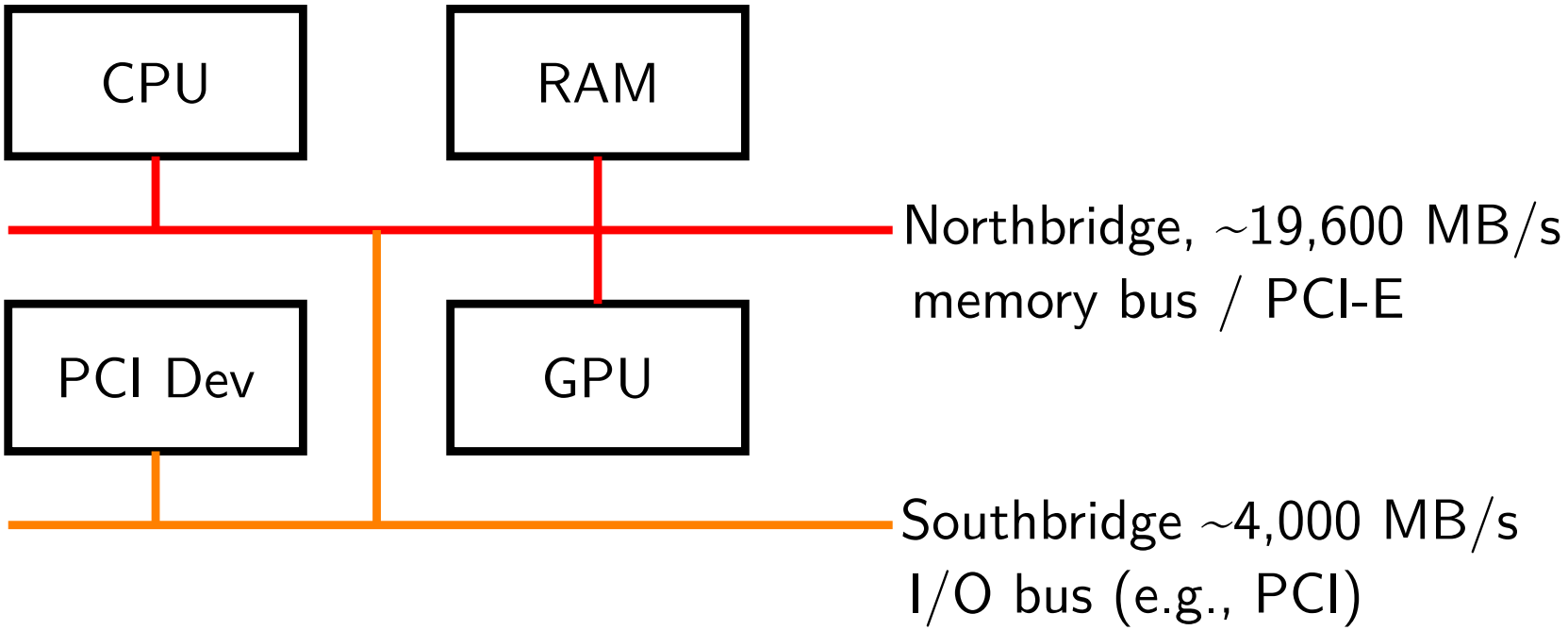
- The OS handles device management (and access)
- OS exposes a *uniform* interface to applications
- IO is **interrupt** driven

Hardware support for devices: Northbridge



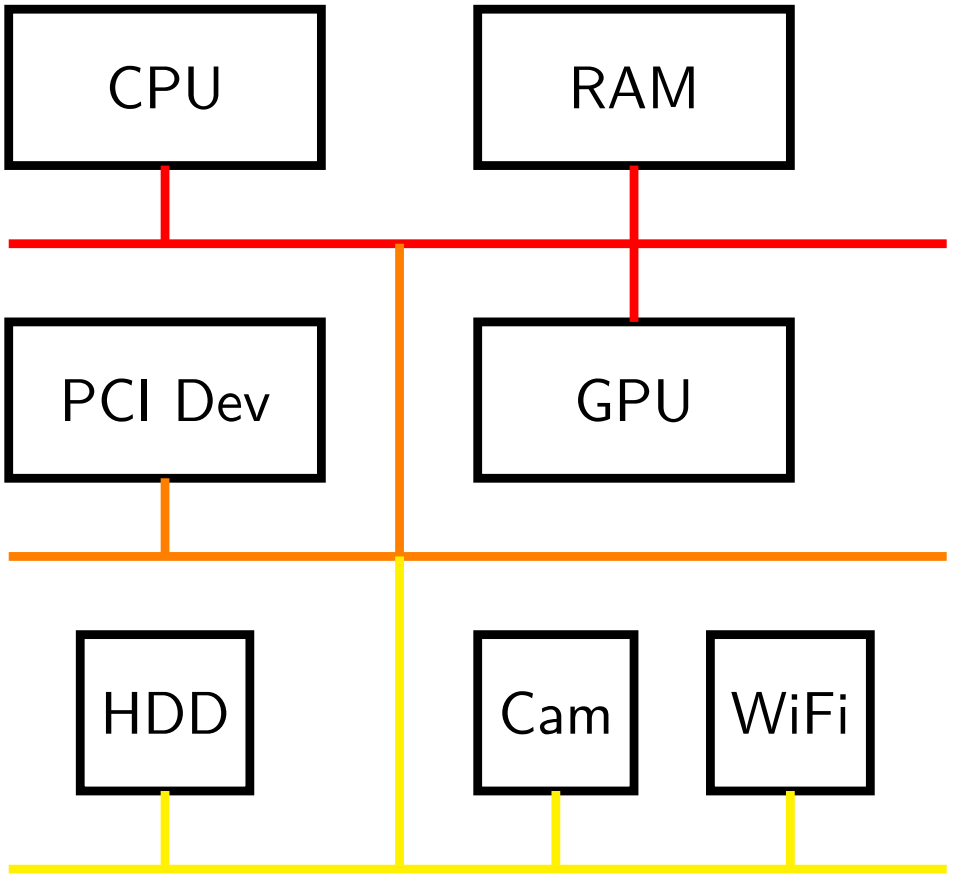
What about other devices?

Hardware support for devices: Southbridge



What about "slow" IO?

Hardware support for devices: Other IO



Northbridge, ~19,600 MB/s
memory bus / PCI-E

Southbridge ~4,000 MB/s
I/O bus (e.g., PCI)

Peripheral I/O bus, ~2,000 MB/s
(e.g., SATA, USB)

Terminology

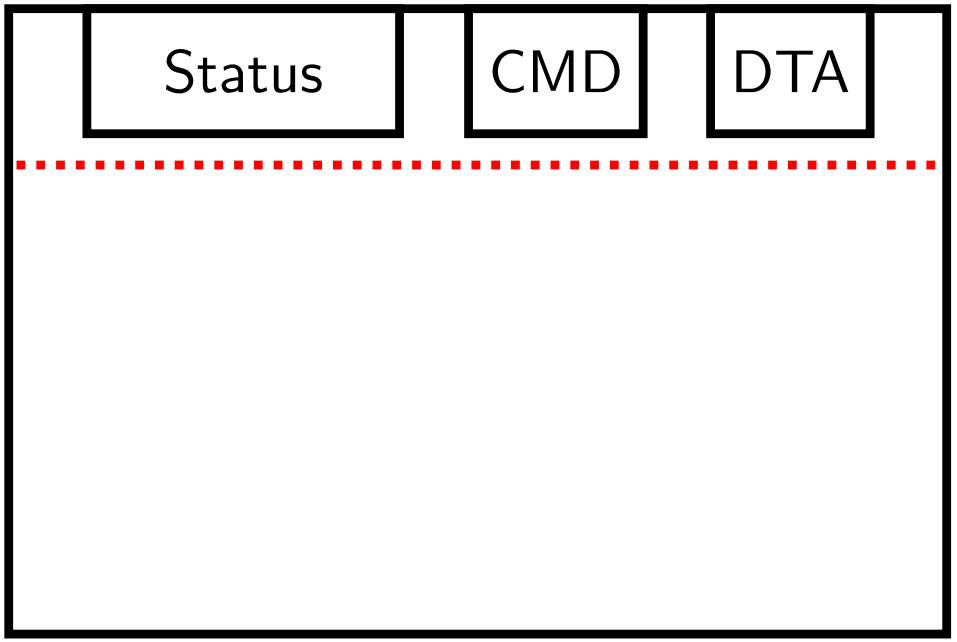
- Northbridge is also called memory controller hub
- Southbridge is also called I/O Controller Hub (ICH / Intel) or Fusion Controller Hub (FCH / AMD)
- The southbridge is connected to the CPU through the northbridge which has a direct connection

Terminology

- Northbridge is also called memory controller hub
- Southbridge is also called I/O Controller Hub (ICH / Intel) or Fusion Controller Hub (FCH / AMD)
- The southbridge is connected to the CPU through the northbridge which has a direct connection

But how do devices work?

Canonical device (1/2)

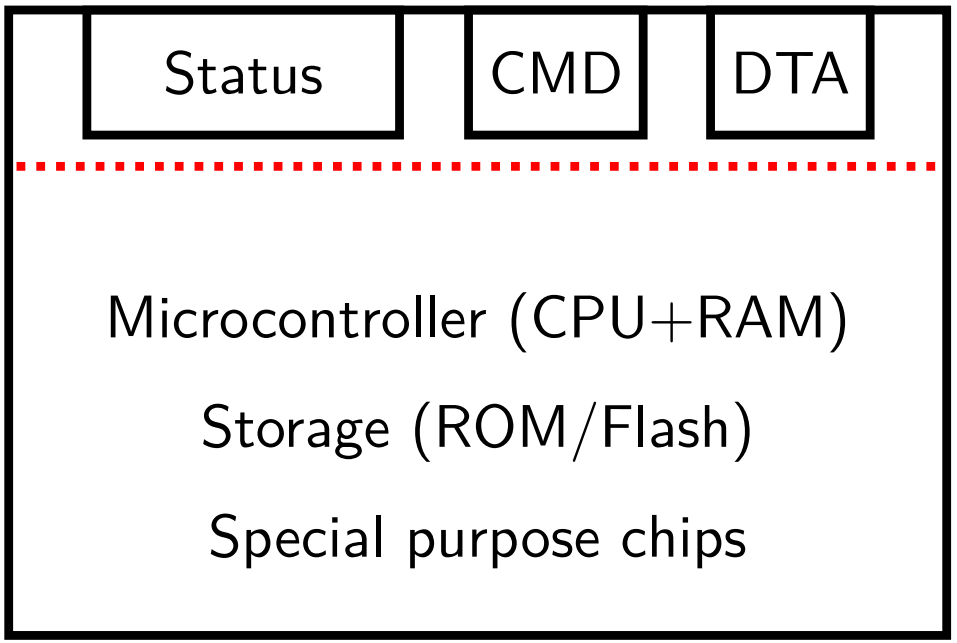


(Device registers)

(Device internals)

- OS writes device registers (by executing CPU instructions)
- Device internals are hidden (abstraction)

Canonical device (2/2)



(Device registers)

(Device internals)

- OS communicates based on agreed protocol (through "driver")
- Device signals OS through memory or interrupt

Device protocol (1/3)

```
while (STATUS == BUSY) ; // 1. spin
// 2. Write data to DTA register
*dtaRegister = DATA;
// 3. Write command to CMD register
*cmdRegister = COMMAND;
while (STATUS == BUSY) ; // 4. spin
```

- Wait until device is ready
- Set data and command (why send data first?)
- Wait until command has completed

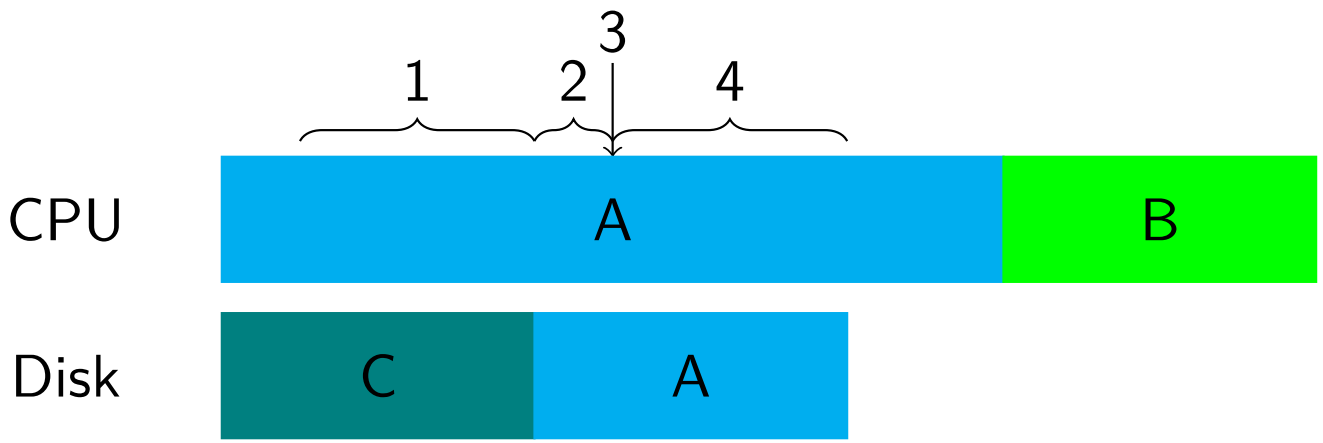
Device protocol (1/3)

```
while (STATUS == BUSY) ; // 1. spin
// 2. Write data to DTA register
*dtaRegister = DATA;
// 3. Write command to CMD register
*cmdRegister = COMMAND;
while (STATUS == BUSY) ; // 4. spin
```

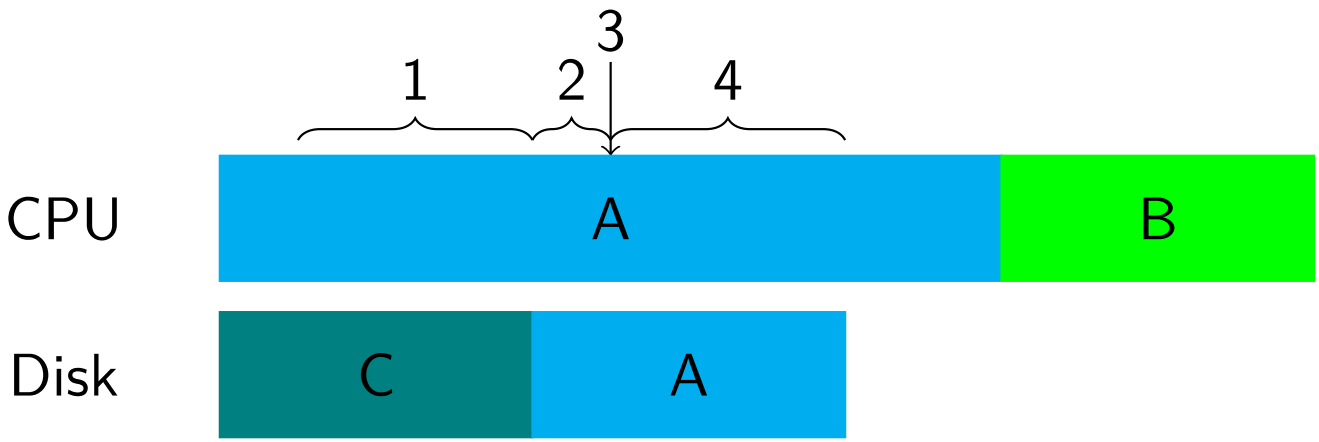
- Wait until device is ready
- Set data and command (why send data first?)
- Wait until command has completed

Where do you see problems?

Device protocol (2/3)

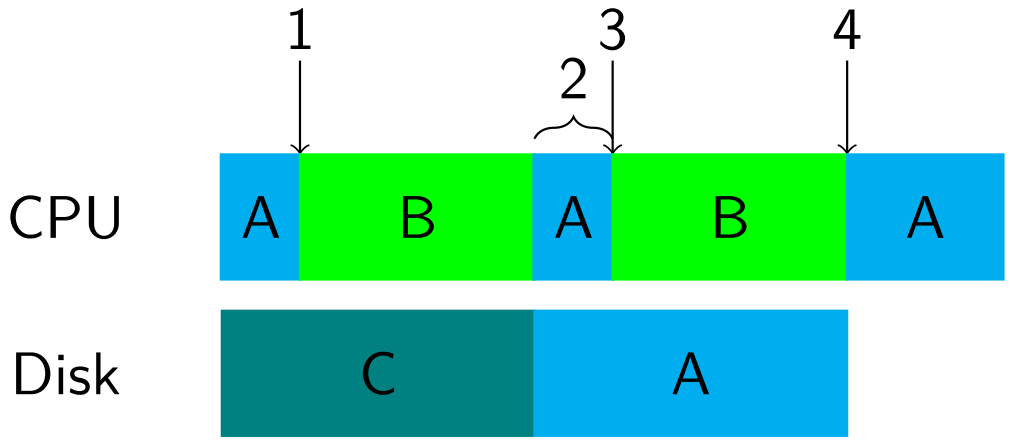


Device protocol (2/3)



- Busy waiting (1. and 4.) wastes cycles twice
- CPU should not need to wait for completion of command
- Solution: embrace asynchronous communication
  - Inform device of request (to get rid of 1.)
  - Wait for signal of completion (to get rid of 4.)

Device protocol (3/3): interrupts



- Instead of spinning, the OS (driver) waits for an interrupt
  - Interrupts are handled centrally through a dispatcher
  - On interrupt arrival, wake up kernel thread that waits on that interrupt

Interrupt performance

- Can interrupts lead to worse performance than polling?

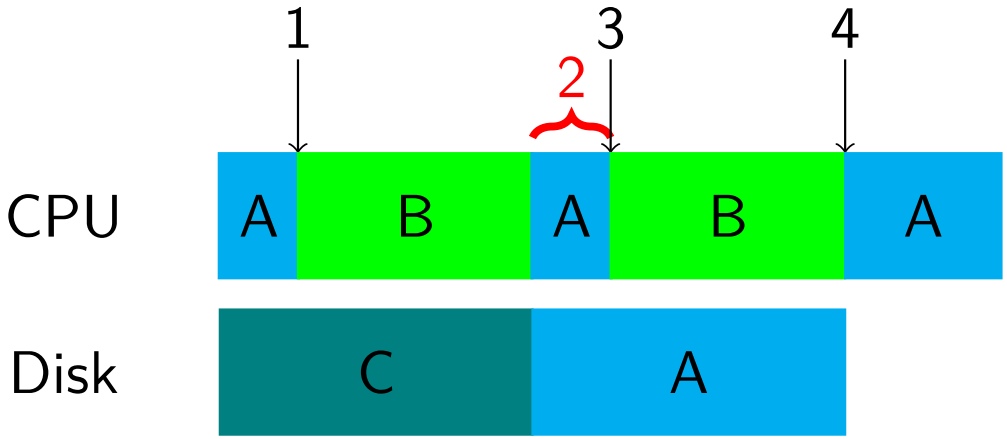
Interrupt performance

- Can interrupts lead to worse performance than polling?
- Yes: livelock (e.g., a flood of arriving network packets)
  - A livelock is *similar to a deadlock* (no process makes progress, resulting in starvation) with the difference that the states of the processes constantly change
  - For example: network packets arrive; interrupt handling and context switch is costly, prohibiting the application from reacting to the packets and then simply queuing up.

Interrupt performance

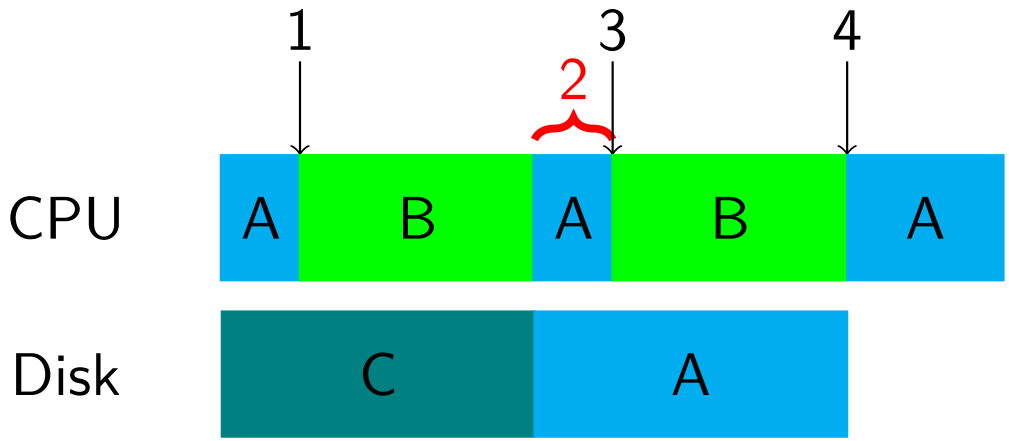
- Can interrupts lead to worse performance than polling?
- Yes: livelock (e.g., a flood of arriving network packets)
  - A livelock is *similar to a deadlock* (no process makes progress, resulting in starvation) with the difference that the states of the processes constantly change
  - For example: network packets arrive; interrupt handling and context switch is costly, prohibiting the application from reacting to the packets and then simply queuing up.
- Real systems therefore use a *mix* between polling and interrupts
  - Interrupts allow overlap between computation and IO, most useful for slow devices;
  - Use polling for fast devices (short bursts) or small amounts of data
- Another optimization is *coalescing*, i.e., the device waits for a bit until more requests complete, then batch sends everything.

Optimize data transfer



- **PIO (Programmed IO)**: CPU tells the device **what** data is
  - One instruction for each byte/word
  - Efficient for a few bytes, scales terribly

Optimize data transfer



- **PIO (Programmed IO)**: CPU tells the device **what** data is
  - One instruction for each byte/word
  - Efficient for a few bytes, scales terribly
- **DMA (Direct Memory Access)**: tell device **where** data *is*
  - One instruction to send a pointer
  - Efficient for large data transfers

How to transfer data?

- IO ports
  - Each device has an assigned IO port
  - Special instructions (`in`/`out` on x86) communicate with device
- Memory mapped IO
  - Device maps its registers to memory
  - Loads/stores interact with device

How to transfer data?

- IO ports
  - Each device has an assigned IO port
  - Special instructions (`in`/`out` on x86) communicate with device
- Memory mapped IO
  - Device maps its registers to memory
  - Loads/stores interact with device
- Both are used in practice.
  - Architectures now support both.
  - Differences are a matter of choice and preference.

Support for different devices

- Challenge: different devices have *different protocols*
- Drivers are specialized pieces of code for a particular device
  - Low end communicates with the device
  - High end exposes generic interface to OS

Support for different devices

- Challenge: different devices have *different protocols*
- Drivers are specialized pieces of code for a particular device
  - Low end communicates with the device
  - High end exposes generic interface to OS
- Drivers are an example of *encapsulation*
  - Different drivers adhere to the same API
  - OS only implements support for APIs based on device class
- Requirement: well-designed interface/API
  - Trade-off between versatility and over-specialization
  - Due to device class complexity, OS ends with layers of APIs

Complexity of API layers

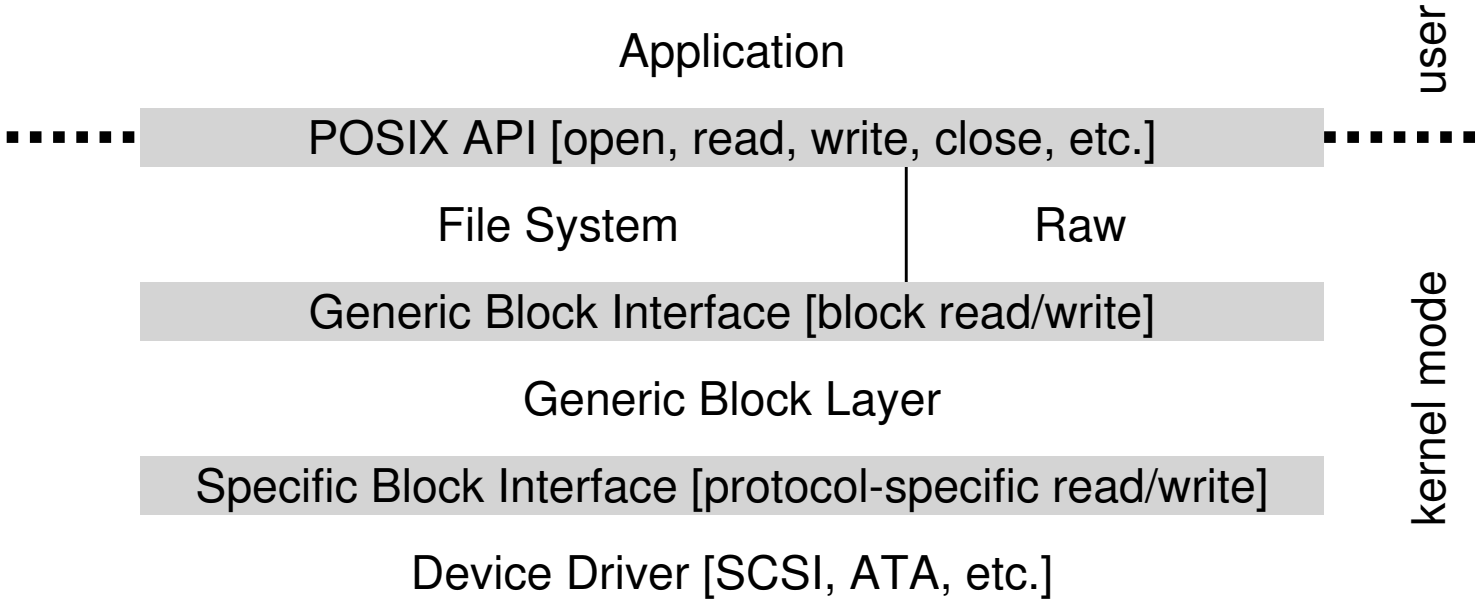


Figure 1: File system stack