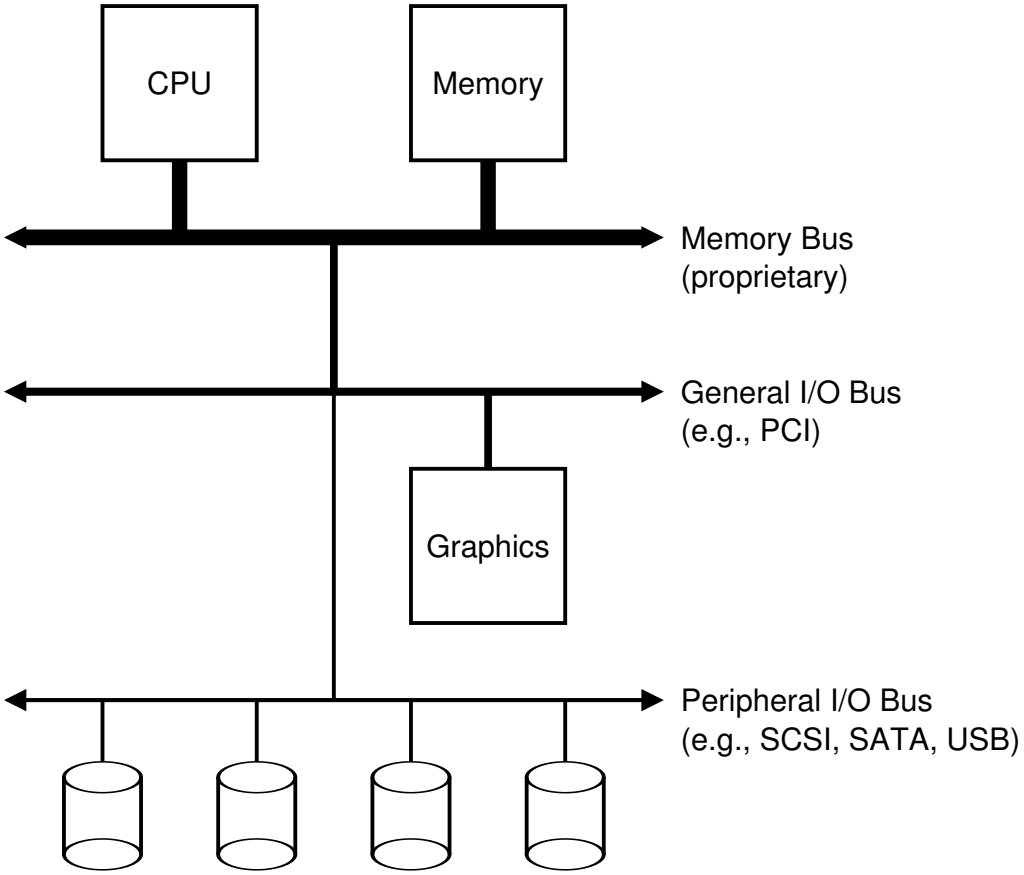
I/O Devices

ICS332
Operating Systems

Henri Casanova (henric@hawaii.edu)

I/O and the OS

- Every program performs some I/O
- The OS is in charge of interaction with I/O devices



High-performance devices
(GPU)
few of them, close to the CPU

Slower devices
(disks, mice, keyboards)
many of them, far from the CPU

Figure 36.1: **Prototypical System Architecture**

I/O and the OS

- The “traditional” picture on the previous slide, nowadays is more complicated with some specialized I/O chips

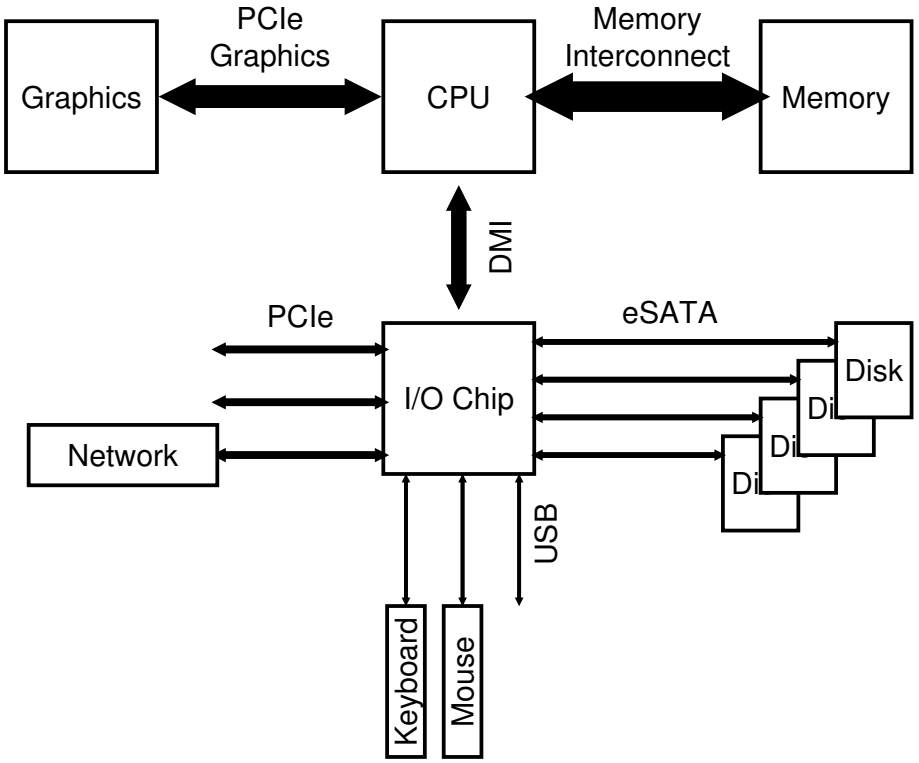


Figure 36.2: **Modern System Architecture**

- OSTEP shows this figure for an Intel Architecture as an example
- The goal is to improve performance
  - e.g., Graphics are treated special for better gaming

What Do Devices Look Like?

- A device has all kinds of internal hardware components (obviously), and controls these components with firmware (software stored in the device, typically upgradable)
- A device exposes a hardware interface
- This interface consists of registers
  - Writing into registers causes the device to do something
  - Reading from registers makes it possible to retrieve data from the device
- The OS uses this hardware interface on behalf of user programs, thereby “virtualizing I/O”
- OSTEP uses a simple “canonical” device example…

A Canonical Device

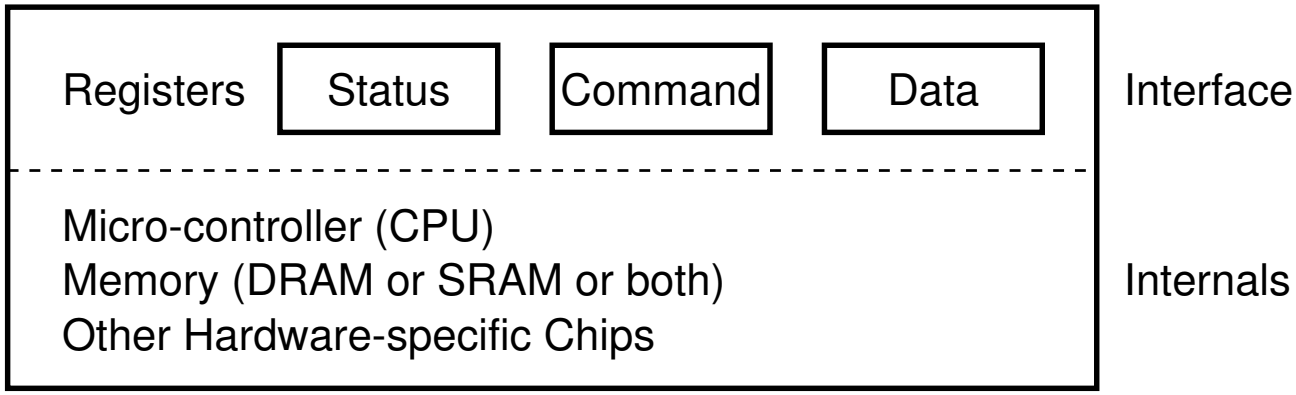


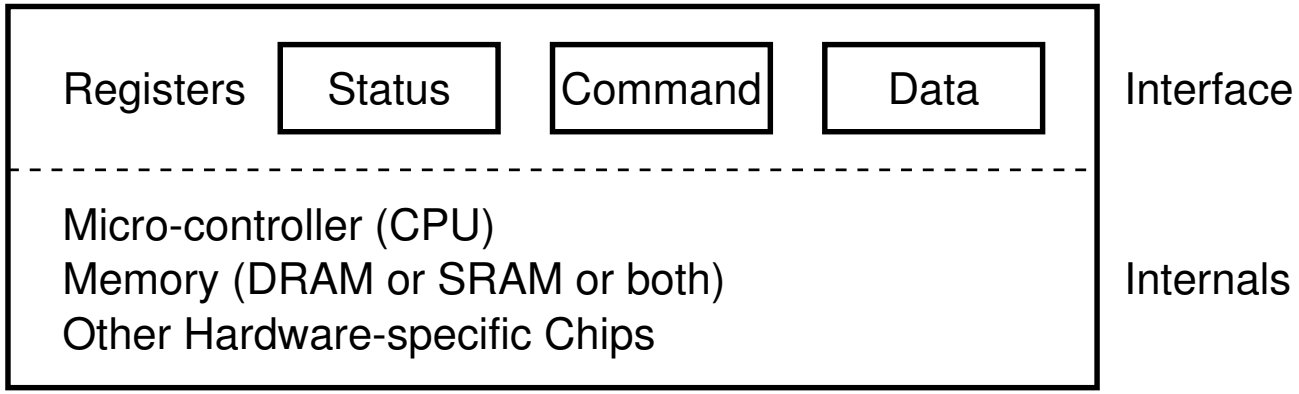
Figure 36.3: **A Canonical Device**

- A 3-register interface:
  - Status: can be read to find out what's going on on the device
  - Command: can be written to tell the device to do something
  - Data: can be read/written to get/put data from/to the device
- This is the bare minimum that an actual device would provide, and makes it possible for software to interact with it
- A device driver is the part of the OS code that interacts with the device using its hardware interface

Device Interactions: Polling

- One option is to write code that interacts synchronously with the device
- Example from OSTEP:

```
// The Polling Approach
While (STATUS == BUSY) {} // Wait for device to be idle
Write data to DATA        // Give data to device
Write command to COMMAND  // Tell device what to do
While (STATUS == BUSY) {} // Wait for device to be done
```



Device Interactions: Interrupts

- Polling is straightforward, but for a “slow” device it wastes a lot of CPU cycles
- The alternative is to use interrupts, as we’ve seen:
  - If the device supports it, its possible to send it requests and ask it to reply via an interrupt
  - There is an interrupt controller hardware component that manages all this and relays interrupts generated by devices to the CPU
- With interrupts, a program that does I/O is kicked out of the CPU so that other processes can run
  - This is what we’ve always assumed so far (Scheduling module)
  - This is also what happens with DMA
- But using interrupts with a “fast” device has a high overhead
- In practice, both polling and interrupts can be used or even mixed (poll for a little bit, and then resort to interrupts)
  - Very reminiscent of spin locks, blocking locks, and hybrid locks!

Interaction with the Device

- So far we've said "read/write values into the device's registers"
- But how doe the OS do this in practice?
- Option #1: Port-mapped I/O
  - Special I/O instructions built into the CPU
    - The `in` / `out` privileged instructions on x86
    - Specify the device's name or "port"
- Option #2: Memory-mapped I/O
  - The device's registers are "in RAM"
    - Address ranges are reserved for I/O devices
  - That is, reading/writing specific RAM addresses causes reading/writing of the device's register
  - Nice because you don't need special instructions and can use all "normal" instructions
- Options #2 is generally preferred, but both exist (and some devices support both)

Device Drivers

- A device driver interacts with a device's hardware interface and exposes a generic API to the kernel
- Linux defines several generic device driver APIs:
  - "Character" device drivers (mouse, keyboard, controller, sound cards, …)
    - Stream of bytes, in sequence (no jumping around)
  - "Block" device drivers (disks, CDs, tapes)
    - Read/Write fixed-size blocks of data, anywhere on the device, memory-mappable
  - "Network" device drivers (network cards)
    - Sort of like "block" but device can asynchronously push data to the kernel, and deal with special network stuff (network addresses, traffic monitoring, etc.)
- So, for instance, a device driver developer for a particular disk hardware interface will implement the "block device" API, and then the Kernel will be able to use the device
- Device hardware interfaces typically follow some standards
  - e.g., there is a single SATA device driver that can deal with all SATA drives

Linux and “Devices”

- In Linux, a lot of things are viewed by the OS as devices, but do not correspond to actual hardware devices
- Many things in Linux are viewed by the OS as “files” but are not files at all, and this includes devices
  - e.g., a process’s stdout stream, a pipe, a socket,
- Example: **`/dev/drandom`**
  - On a Linux system you can see the above “file”
  - But it’s not a file, it’s a device
  - But it’s not a device, it’s a software component of the OS that generates pseudo-random numbers based on ambient “noise”
- Other example: **`/dev/null`**
  - Just a byte eater
- This “everything looks like a file, including devices” philosophy has been central to UNIX/Linux from the beginning

Main Takeaways

- Hardware devices comply with different interface standards
- Hardware interfaces consists of registers that are read/written
  - Either via special instructions
  - Or via memory-mapping
- All interaction with slow devices is interrupt-based
- Device drivers are kernel code (or really kernel modules) that know how to interact with devices using these interface standards
- Linux implements all kinds of “fake” devices for convenient purposes

Conclusion

- Device drivers represent a LOT OF CODE in the Kernel
  - According to OSTEP, about 70% of your kernel code on your machine is device drive code
- We can't really go much deeper here
  - What's really interesting about device drivers is to implement one from scratch or fix/modify an existing one
  - There are entire books dedicated to device driver implementation