Universidade Estadual de Campinas
Instituto de Computação

MC504 Sistemas Operacionais

T25 I/O Devices

Referência principal
Ch.36 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 29 de outubro de 2018

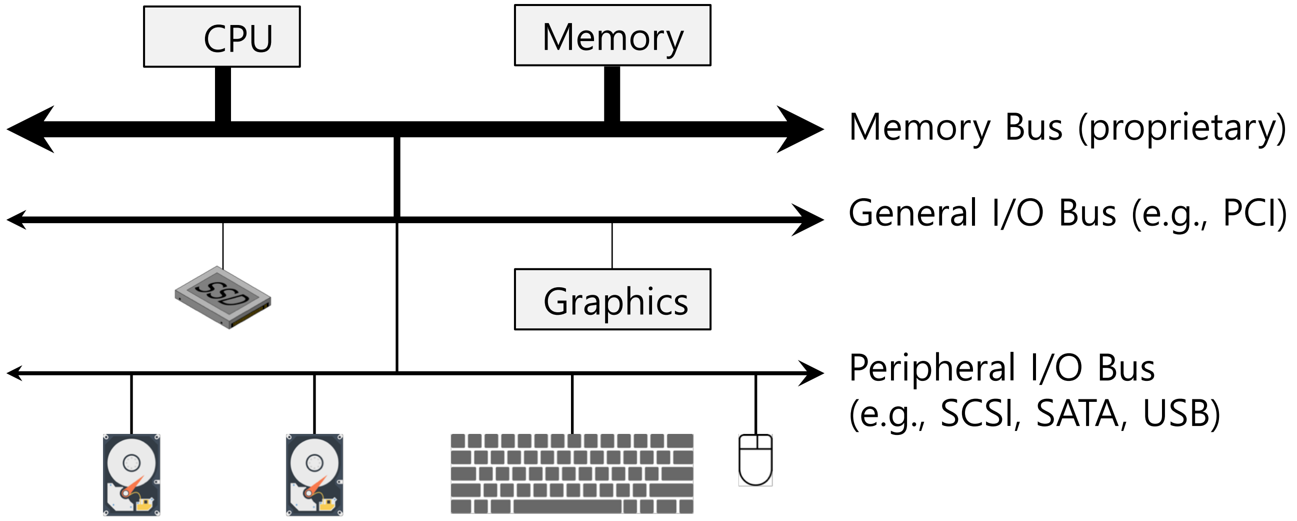
Arthur João Catto, PhD

2º semestre de 2018

Since I/O is quite critical to computer systems. . .

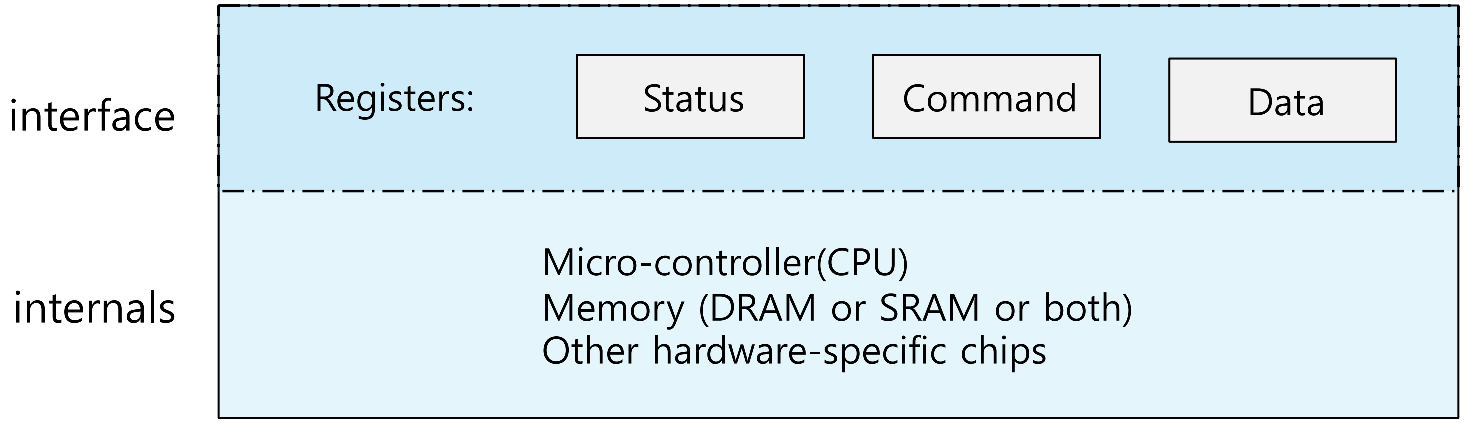
- How should I/O be integrated into systems?
- What are the general mechanisms?
- How can we make them efficient?

A Prototypical System Architecture



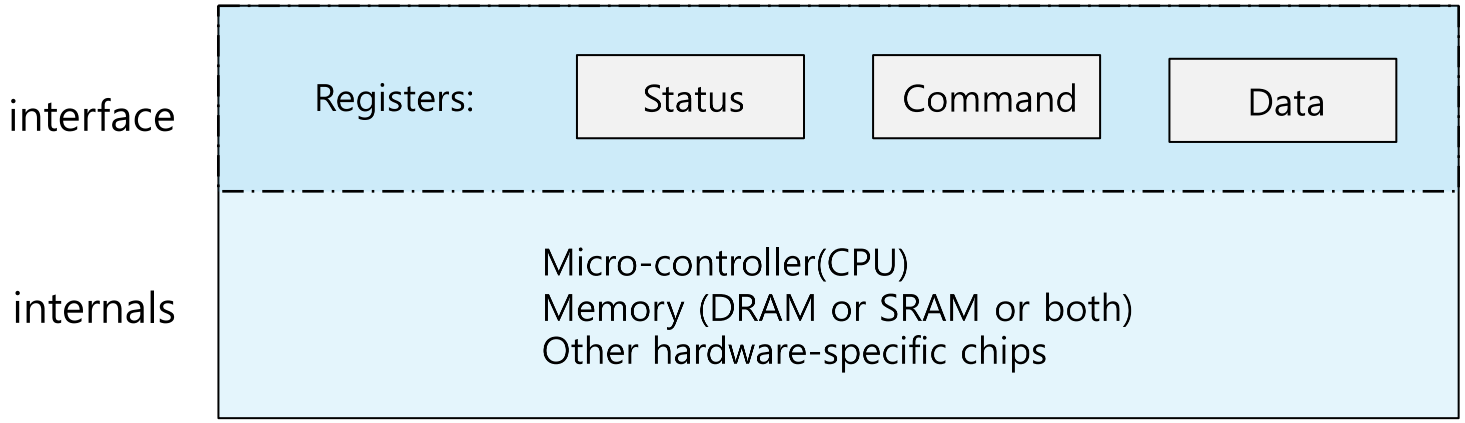
- A CPU is attached to main memory via some kind of proprietary bus.
- Graphics and some higher-performance I/O devices are connected to the system via a general I/O bus (e.g. PCI or one of its many derivatives).
- Finally, there are one or more peripheral buses, such as SCSI, SATA, or USB, connecting slower devices (e.g. hard disks, keyboard, mouse) to the system.

A Canonical Device: two important components



- **The hardware interface** it presents to allow the system software to control its operation.
  - All devices have some specified interface and protocol for typical interaction.
- **The device's internal structure** which is implementation specific and is responsible for creating the abstraction the device presents to the system.
  - Simple devices will have a few hardware chips to implement their functionality.
  - More complex devices will include a CPU, some general purpose memory, and other device-specific chips to get their job done.

The Canonical Device Interface



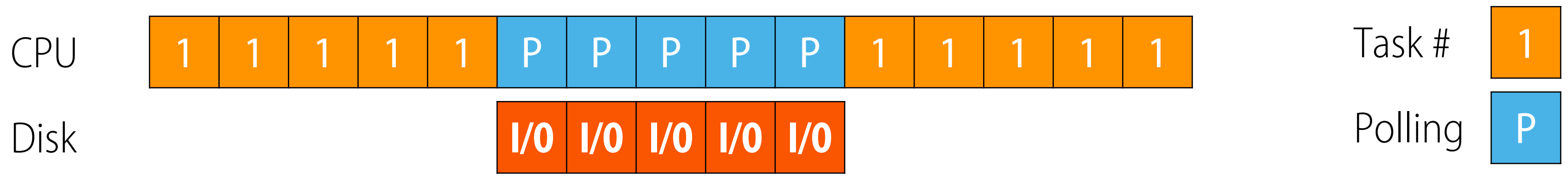
- The operating system can control the canonical device's behavior by reading and writing to three registers:
  - **Status register,** to get the current status of the device.
  - **Command register**, to tell the device to perform a certain task.
  - **Data register,** to pass data to the device, or get data from it.

The Canonical Protocol

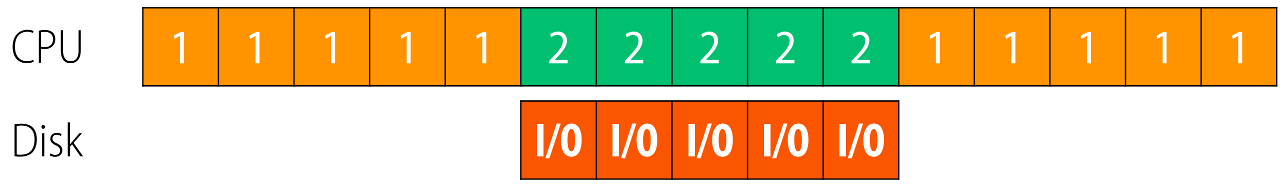
```
while (STATUS == BUSY)
    ; // wait until device is not busy
write data to DATA register;
write command to COMMAND register; // starts device and executes command
while (STATUS == BUSY)
    ; // wait until device is done with your request
```

- Although this basic protocol is simple and works, it also shows some inefficiencies and inconveniences.
- Polling seems inefficient because it wastes a lot of CPU time just waiting for the (potentially slow) device to complete its activity, instead of switching to another ready process and thus better utilizing the CPU.

Lowering CPU Overhead With Interrupts



- Here, the system spins while waiting for the completion of a disk I/O request by Process 1.



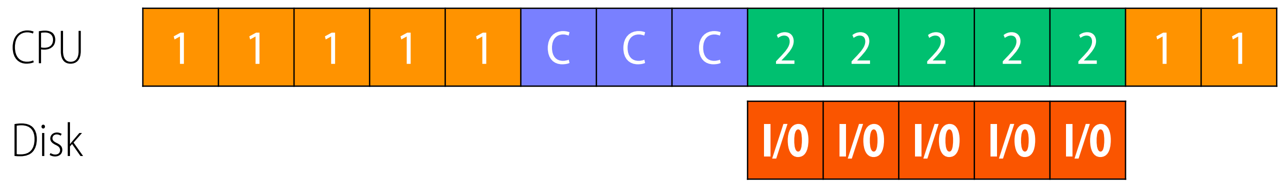
- When the system uses interrupts, the OS can schedule Process 2 to run while waiting for the disk.

WARNING: Interrupts Not Always Better Than PIO

- Interrupts allow for overlap of computation and I/O but they only really make sense for slow devices. Otherwise the cost of interrupt handling and context switching may outweigh their benefits.
- A flood of interrupts may also overload a system and lead it to **livelock**. In such cases, polling provides more control to the OS in its scheduling and thus is again useful.
- A hybrid approach that polls for a little while and then, if the device is not yet finished, uses interrupts can be used if the speed of the device is not known, or is sometimes fast and sometimes slow.
- Another interrupt-based optimization is **coalescing**, where a device first waits a little before delivering an interrupt to the CPU. While waiting, other requests may complete, and thus multiple interrupts can be coalesced into a single delivery, lowering the overhead of interrupt processing.

Another Issue On The Canonical Protocol

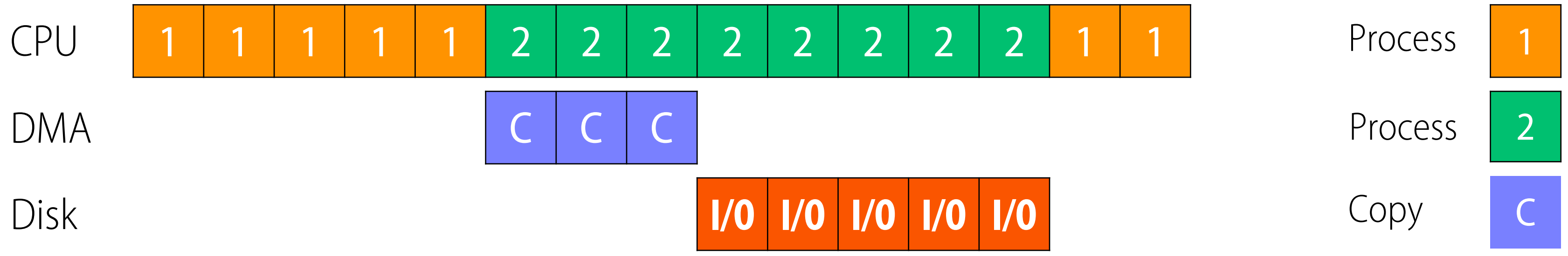
- When using programmed I/O (PIO) to transfer a large chunk of data to a device, the CPU wastes a lot of time and effort that could better be spent running other processes as shown by this timeline:



- Process 1 is running and then wishes to write some data to the disk.
- It then initiates the I/O, which must copy the data from memory to the device explicitly, one word at a time (marked “C” in the diagram).
- When the copy is complete, the I/O begins on the disk and the CPU can finally be used for something else (running Process 2, in this case).

More Efficient Data Movement With DMA

- A Direct Memory Access (DMA) engine can arrange transfers between devices and main memory without much CPU intervention.
- To transfer data to a device, for example, the OS programs the DMA engine by telling it the data address, how much data to copy, and the device to send it to.
- After that the OS is done with the transfer and can proceed with other work.
- When transfer is done, the DMA controller tells the OS by raising an interrupt.



- Now, Process 2 thus gets to use more CPU before Process 1 runs again.

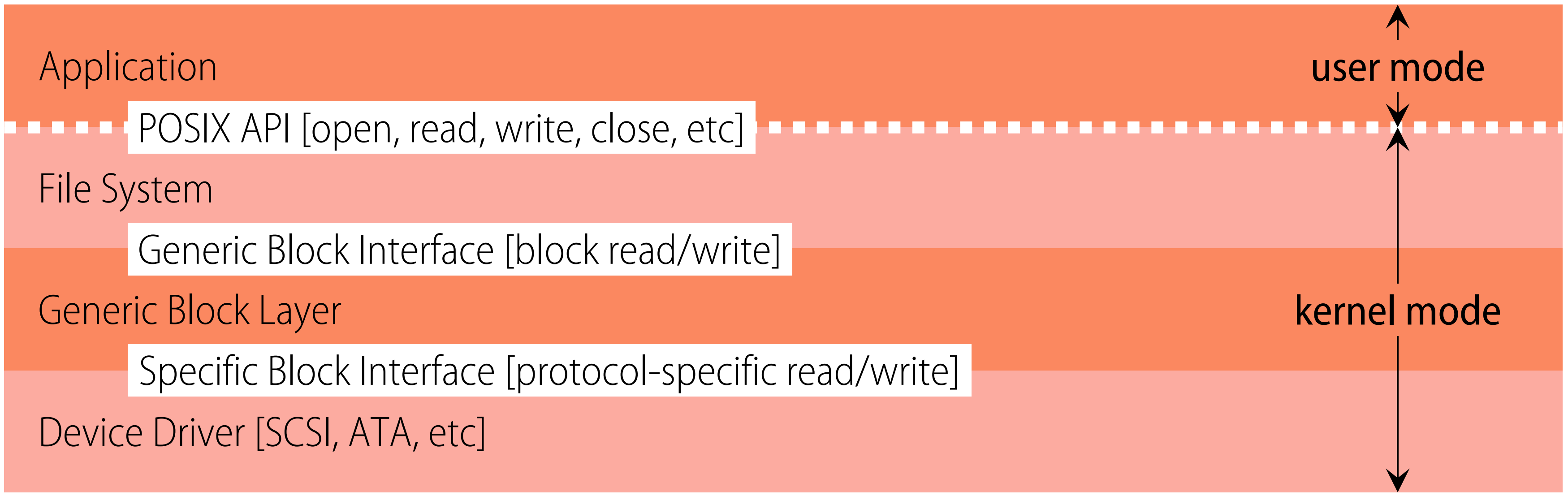
How Does The Hardware Communicate With Devices?

- Via explicit **privileged I/O instructions**
  - They specify a way for the OS to send data to specific device registers and thus allow the construction of the protocols just described.
- Via **memory-mapped I/O**
  - The hardware makes device registers available as if they were memory locations.
  - To access a particular register, the OS issues a load (to read) or store (to write) to its address; the hardware then routes the load/store to the device instead of main memory.
- **There is no best approach.**
  - The memory-mapped approach is nice in that no new instructions are needed to support it, but both approaches are still in use today.

How to Fit Specific Devices into a General-Purpose OS?

- A file system should work on top of a variety of disks (e.g. SCSI, IDE, USB pendrives, etc.).
- The file system should be able to ignore the details of how to issue a read or write request to each of these different types of drives.
- The problem is solved using **abstraction**.
  - At the lowest level in the OS must know in detail how a device works.
  - This piece of software is a **device driver** which encapsulates all the specifics of device interaction.

A File System Software Stack


Application
user mode
POSIX API [open, read, write, close, etc]
File System
Generic Block Interface [block read/write]
Generic Block Layer
kernel mode
Specific Block Interface [protocol-specific read/write]
Device Driver [SCSI, ATA, etc]

- An application and the file system are oblivious to the specifics of the disk class they are using.
  - The file system simply issues block read and write requests to the generic block layer, which routes them to the appropriate device driver, which handles the details of issuing the specific request.

Two Concerns About I/O Encapsulation

- When a device has many special capabilities, but has to present a generic interface to the rest of the kernel, those special capabilities will go unused.
- Because device drivers are needed for any device that might be plugged into a system, they have come to represent a huge part of kernel code.
  - In the Linux kernel, device drivers account for over 70% of the code.
  - As drivers are often written by "amateurs" (instead of full-time kernel developers), they tend to have many more bugs and thus are a primary contributor to kernel crashes.