Operating System (OS)
CS232

Persistence: I/O Devices

Dr. Muhammad Mobeen Movania
Mr. Abid Butt

Outlines

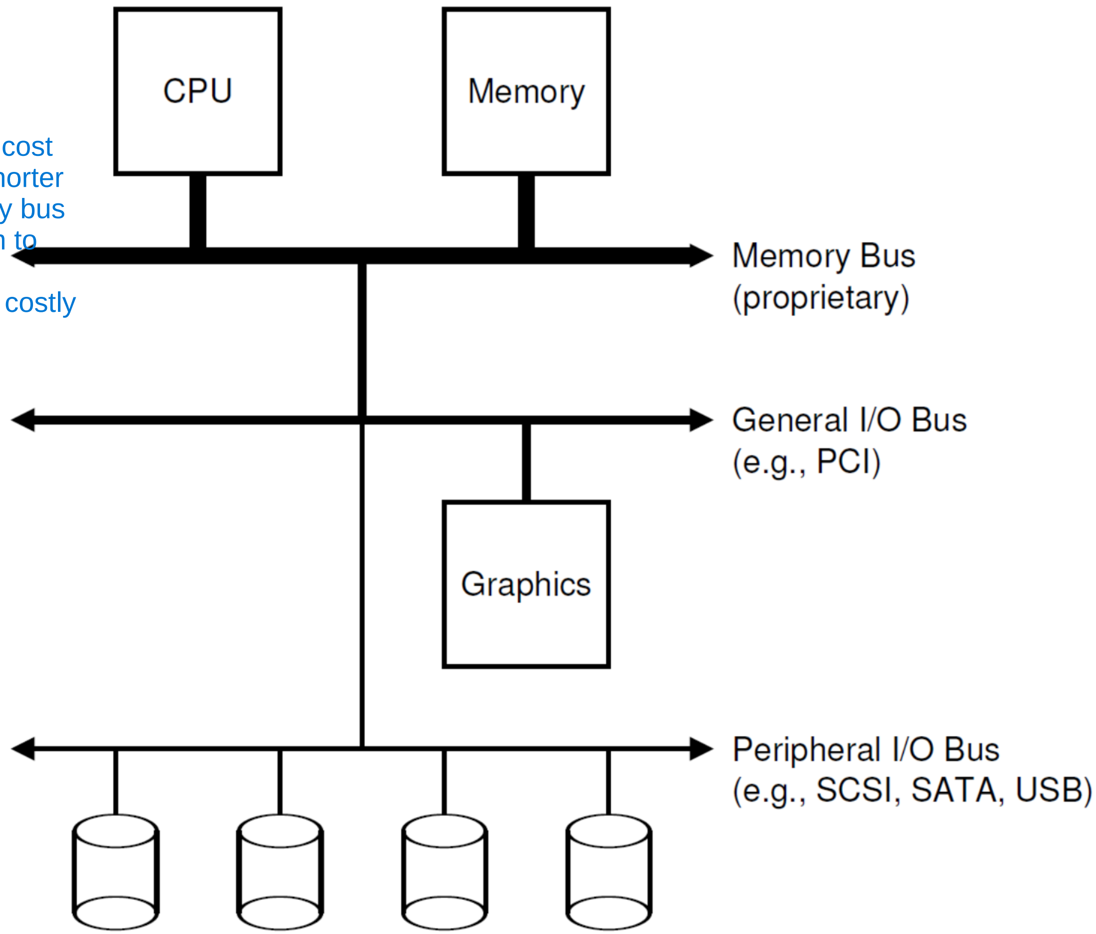
- Significance of I/O
- I/O interfaces in a prototypical and modern system
- Typical I/O device
- Typical protocol
- Communicating with devices
- Summary

Significance of I/O

- No serious program can be considered complete without I/O
- Its important to vary processing of data based on inputs and outputs
- Challenge is to support a wide range of I/O products in as general an abstraction as possible

Simple Prototypical System

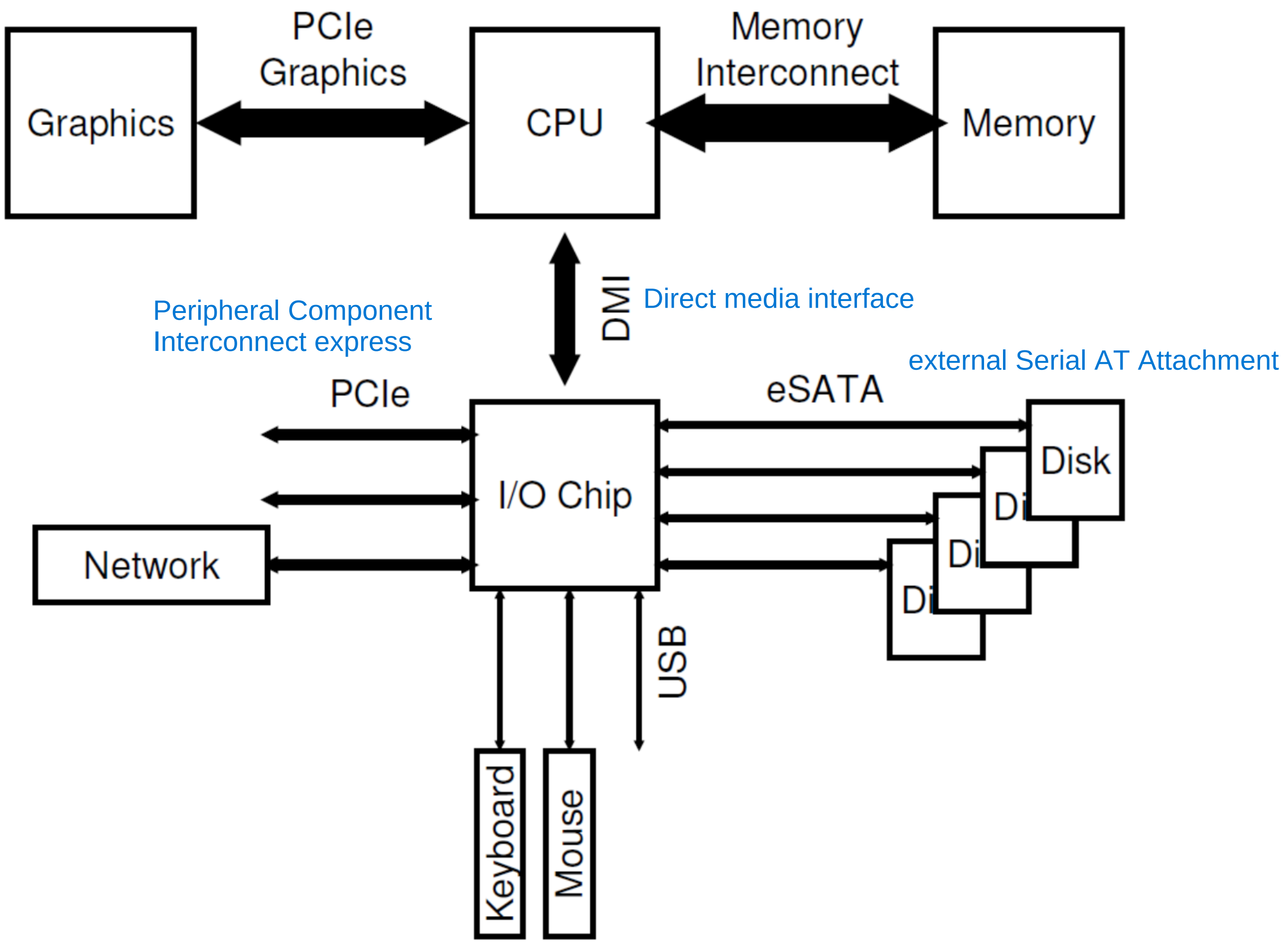
- why hierarchical: physics and cost
- physics: faster bus must be shorter
    - high-performance memory bus does not have much room to plug devices
- cost: high-performance bus is costly



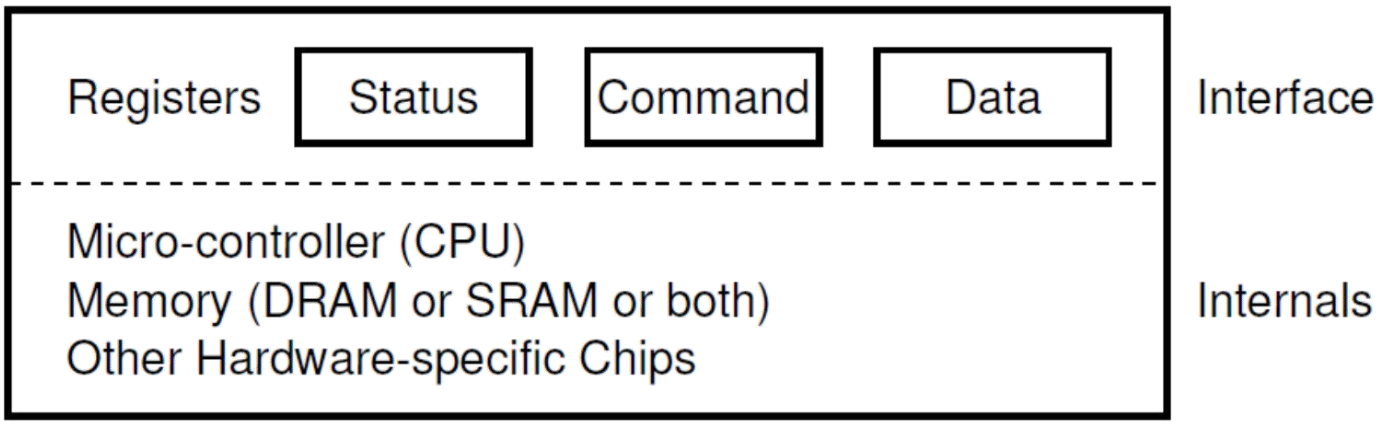
disks, mice, keyboards (slower devices)

Typical I/O Interfaces in a modern system

-specialized chipsets
-faster point-to-point interconnects



Typical I/O Device



Three types of I/O

When main CPU is involved with data movement

- Programmed I/O (aka Polling)
- Interrupt driven I/O
- DMA

Typical Protocol

- Uses polling

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

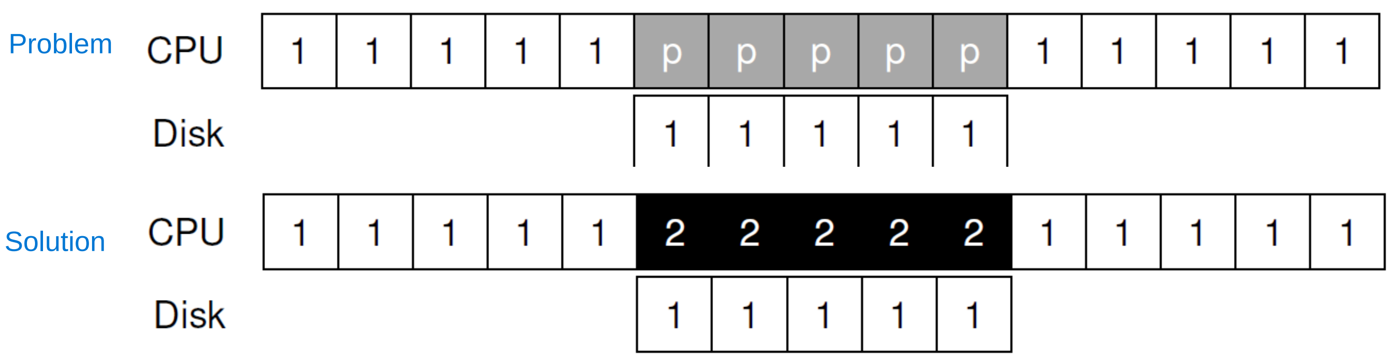
Positive aspect:
- Simple and working

Negative:
- inefficiencies (waiting wastes CPU time, could switch to something else)
- inconveniences

Interrupts

Not always the best solution:
- Imagine a device that performs its tasks very quickly
  - If fast, best to poll
  - If slow, use interrupts
  - If unknown speed: use hybrid (poll for a while, then use interrupt)
- Networks:
  - huge stream of incoming packets
  - each generating interrupt
  - can cause "livelock"
- Coalescing:
  - device to wait before issuing interrupt
  - can bunch together other requests that got done

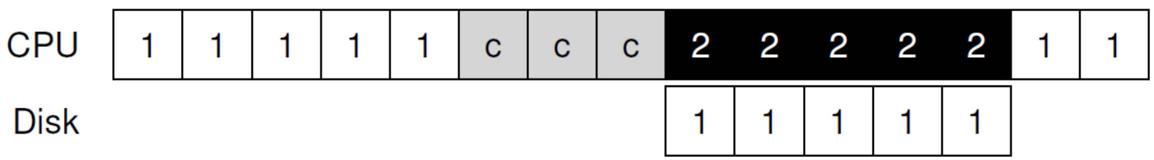
- Let computation and I/O overlap
  - OS writes the data and command
  - Puts the calling process to sleep and shifts to another task
  - After finishing the task, the device raises an interrupt
  - The ISR is called (interrupt service routine)
  - The OS reads the result of the I/O, wakes the sleeping process



Direct Memory Access (DMA)

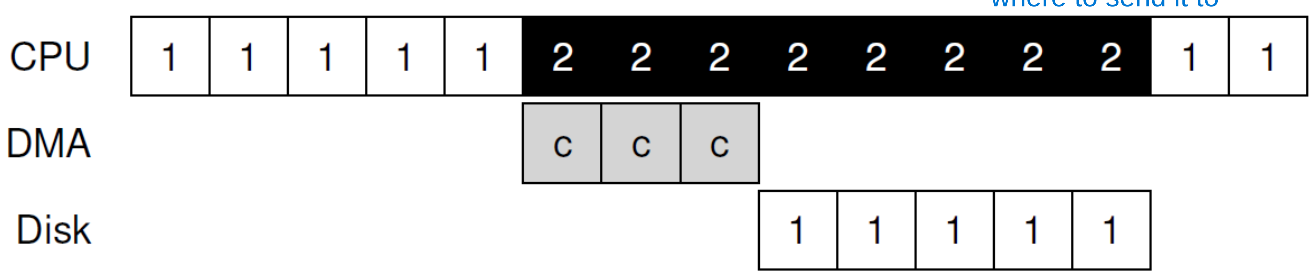
Data movement via ...

- Programmed I/O is also very expensive!



- DMA engine can transfer data b/w RAM and device w/o CPU intervention

OS would program the DMA engine by telling it:
- where the data lives in memory
- how much data to copy
- where to send it to



Communicating with devices

- Two main techniques:
  - Explicit I/O instructions
    - Intel has in and out instructions
  - Memory mapped I/O

- usage:
out reg, port

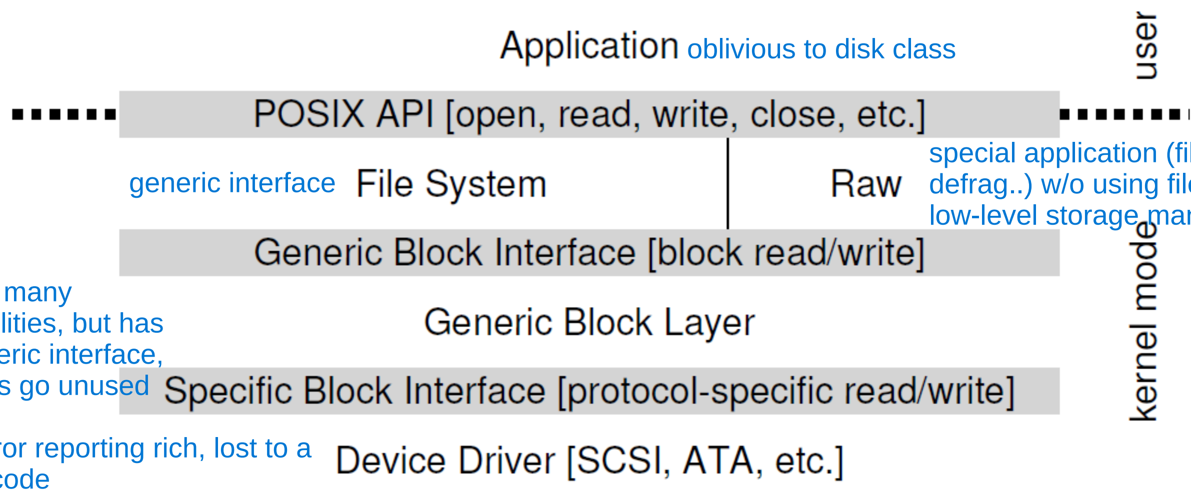
- privileged: only OS has access

- hardware makes device regs available as if they were memory locations
- OS issues a load or store for reading/writing the address
- hardware routes load/store to device instead of memory

OS is oblivious to device details

- Each device has its own characteristics
- How do we do it?
  - Abstraction!
- Device driver is a layer that hides the details

e.g. file system to work on top of SCSI, IDE, USD, and so forth
file system should be oblivious to all the details



special application (file-sys checker, disk defrag..) w/o using file abstraction.. to support low-level storage management applications

Downside:
- if device has many special capabilities, but has to stick to generic interface, special abilities go unused

- e.g. SCSI error reporting rich, lost to a generic error code

- drivers represent 70% of OS code
- millions of lines of device driver code, most inactive in any installation

Summary

- We have a basic understanding of how OS manages I/O devices
- Three types of I/O
  - Programmed I/O
  - interrupt driven I/O and
  - DMA
- Two approaches to accessing device registers
  - explicit I/O instructions and
  - memory-mapped I/O,
- Device driver has been presented, showing how the OS itself encapsulates low-level details and makes it easier to build the OS in a device-neutral fashion