CS3281 / CS5281

I/O Devices

Will Hedgecock
Sandeep Neema
Bryan Ward

**Some lecture slides borrowed and adapted from Andrea Arpaci-Dusseau*

ISIS
Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

VANDERBILT UNIVERSITY

Motivation

What good is a computer without any I/O devices?

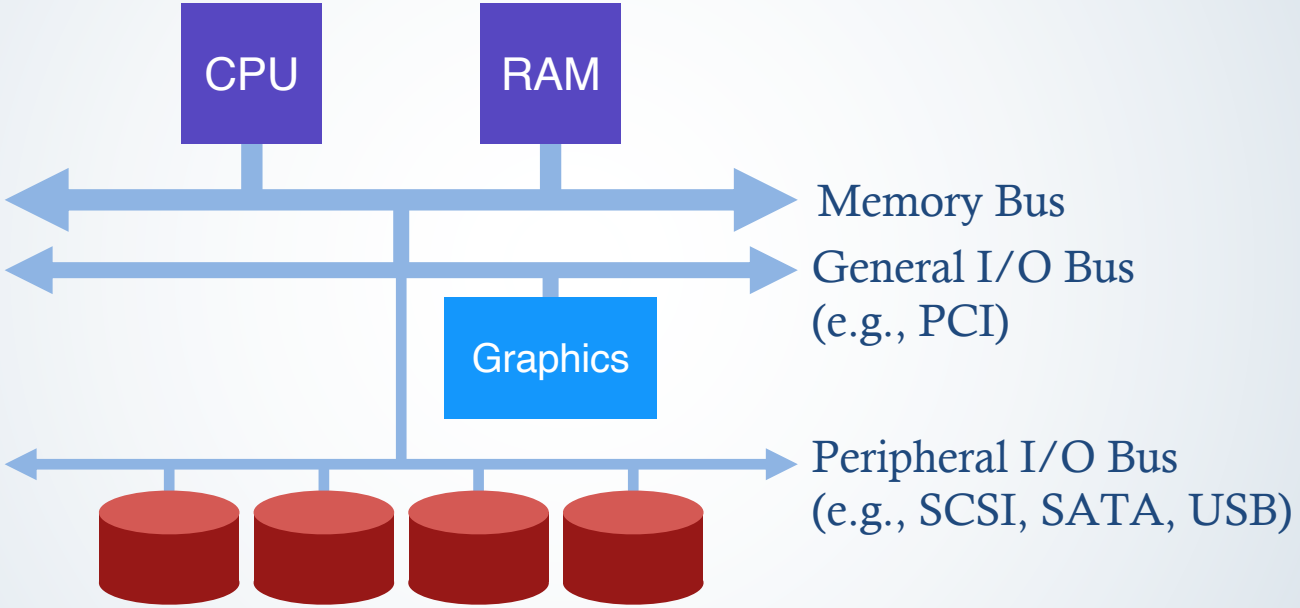
- e.g., keyboard, display, disks

We want:

- **H/W** that will let us plug in different devices
- **OS** that can interact with different combinations

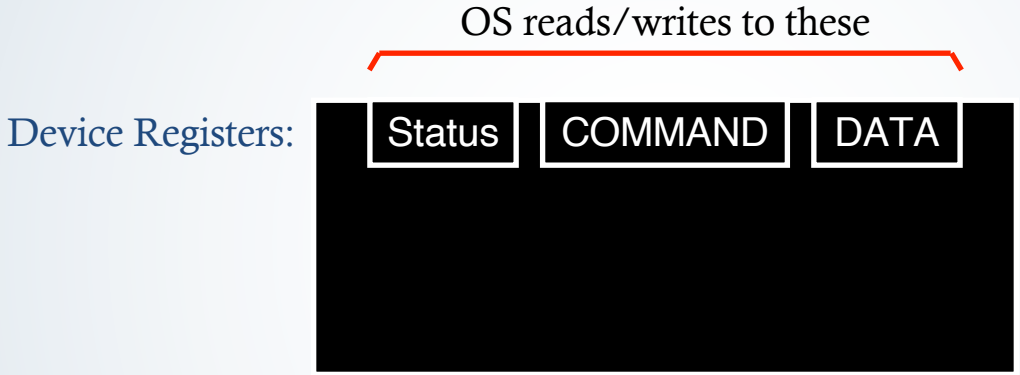
- I/O also allows for *persistence*
  - RAM is *volatile*, i.e., contents are lost when the machine restarts

Hardware support for I/O

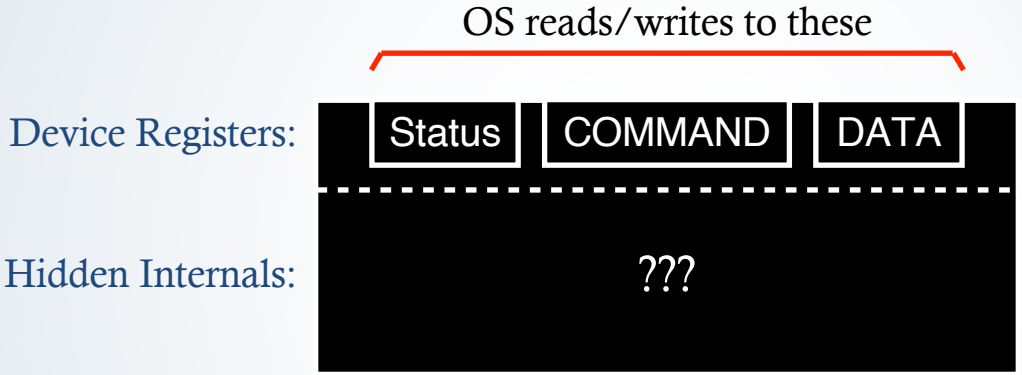


Why use hierarchical buses?

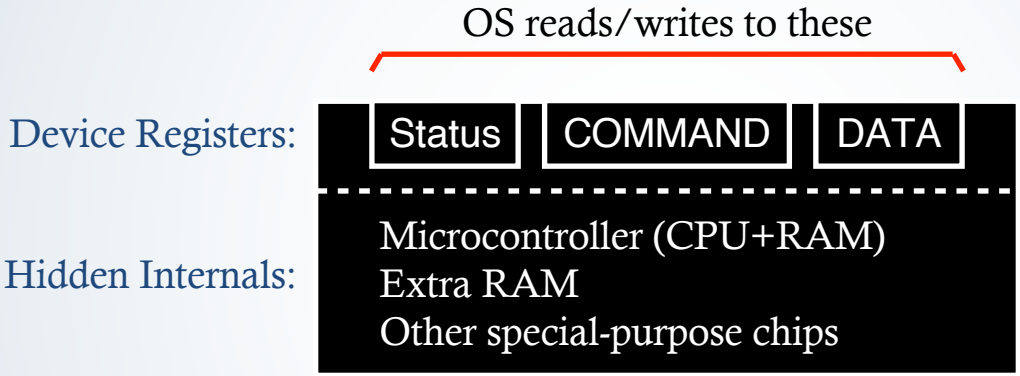
Canonical Device



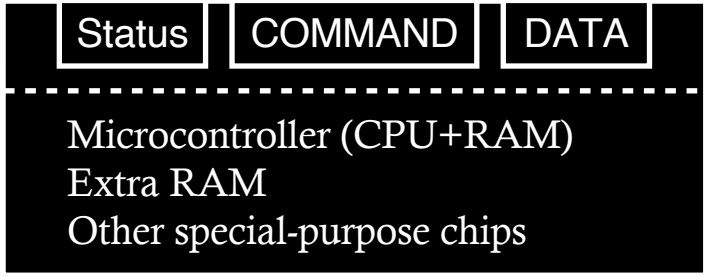
Canonical Device



Canonical Device



Example Write Protocol




```
while (STATUS == BUSY)
  ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
  ; // spin
```

This is called polling when the processor "asks" what the hardware is doing, often continuously

CPU:

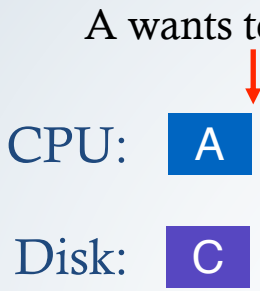
Disk:

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```

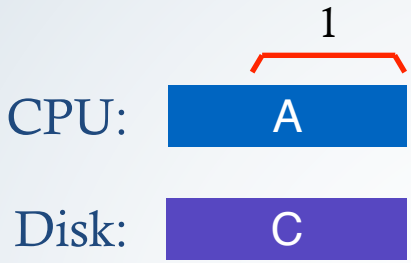
CPU: A

Disk: C

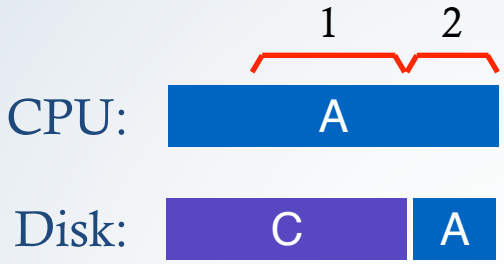
```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



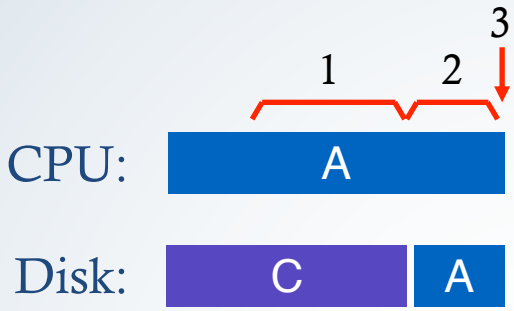

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



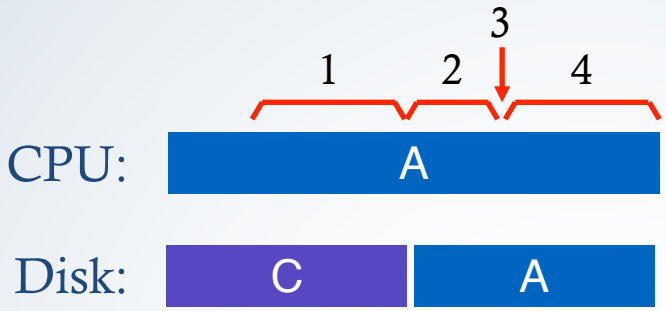

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



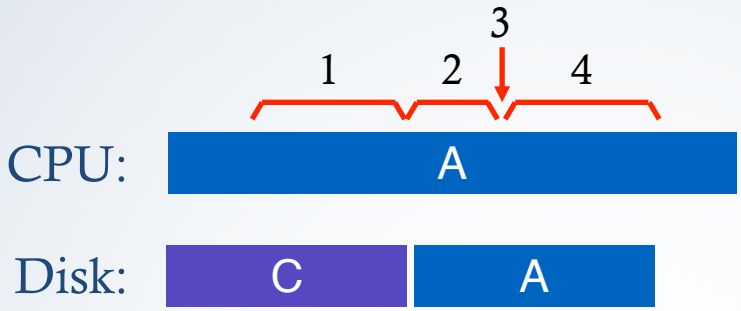

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```




```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```

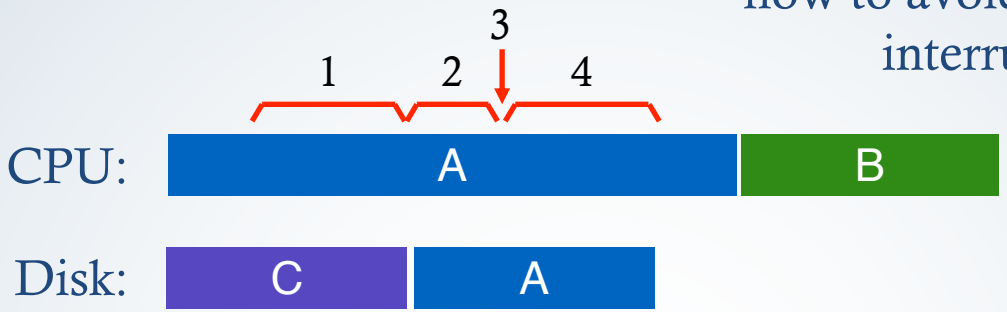



```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



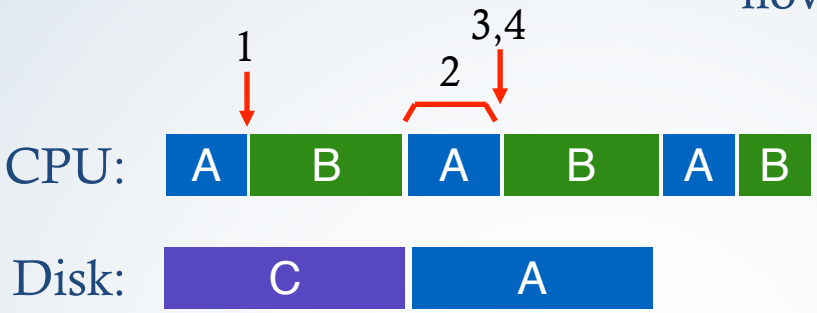

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```

how to avoid spinning?
interrupts!




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

how to avoid spinning?
interrupts!




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

Interrupts vs. Polling

Are interrupts ever worse than polling?

Fast device: Better to spin than take interrupt and context-switching overhead

- Device time unknown? Hybrid approach (spin then use interrupts)

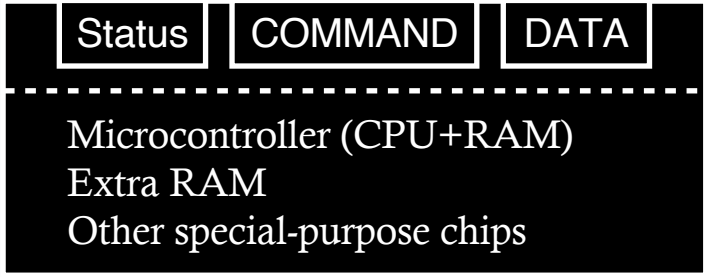
Flood of interrupts arrive

- Can lead to livelock (always handling interrupts)
- Better to ignore interrupts while making some progress handling them

Other improvement

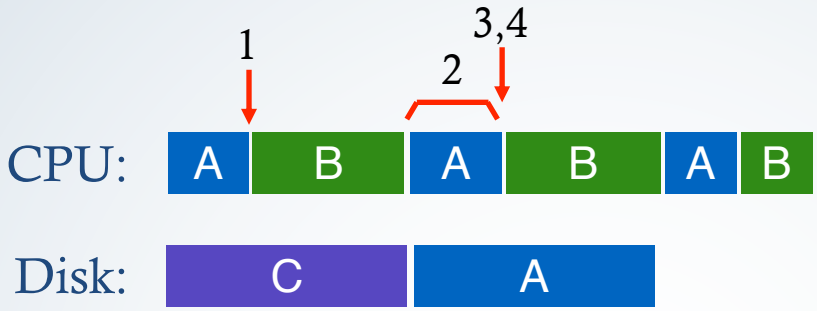
- Interrupt coalescing (hardware batches together several interrupts)
- Split interrupt handling
  - Top half: respond to the interrupt and store relevant data
  - Bottom half: do the real interrupt-handling work at a later point

Protocol Variants



- **Status checks**: polling *vs.* interrupts
- **Data**: programmed I/O (PIO) *vs.* direct memory access (DMA)
- **Control**: special instructions *vs.* memory-mapped I/O

ISIS
Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu
VANDERBILT UNIVERSITY




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

what else can we optimize?

data transfer!

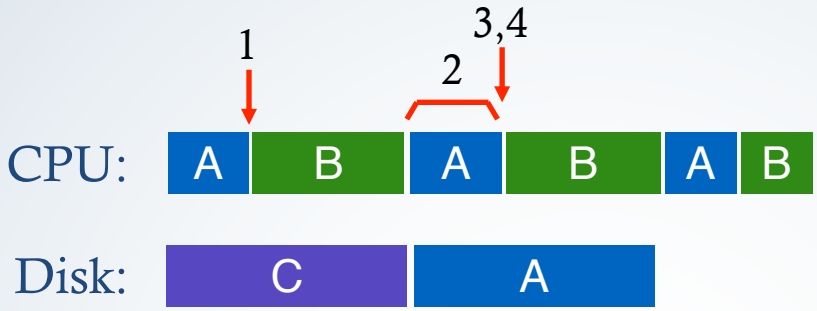
Programmed I/O vs. Direct Memory Access

PIO (Programmed I/O):

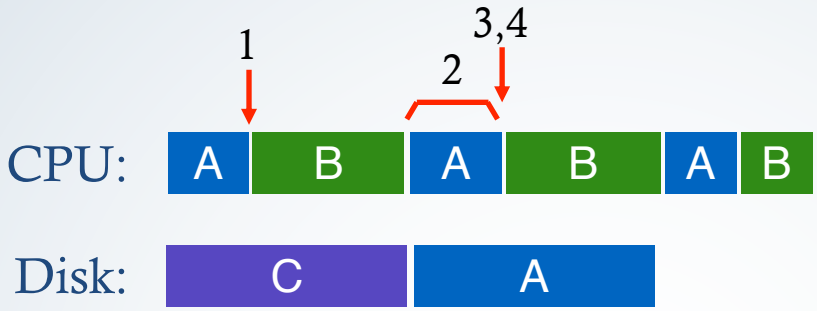
- CPU directly tells device what the data is
- One instruction for each byte/word
- Efficient for a few bytes/words, but *scales terribly*

DMA (Direct Memory Access):

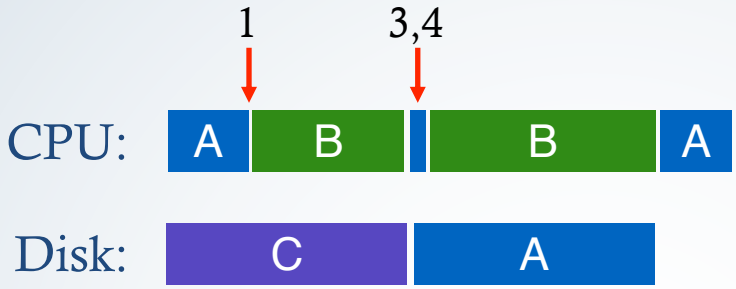
- CPU leaves data in memory
- Device reads/writes data directly from/to memory
- One instruction to send a pointer to the data to send
- Efficient for large data transfers




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

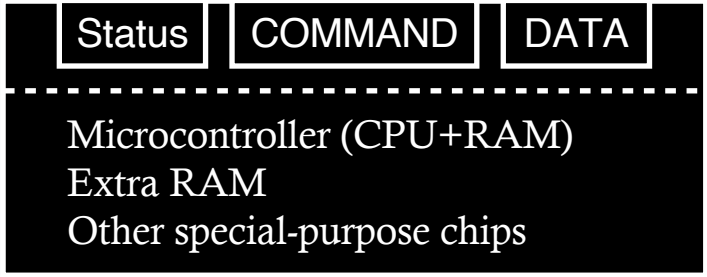



```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

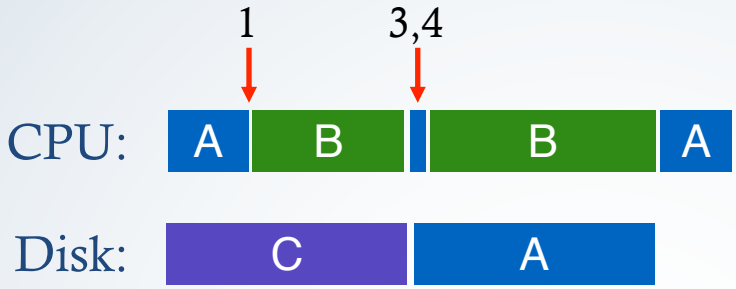
Protocol Variants



Status checks: polling *vs.* interrupts

Data: PIO *vs.* DMA

Control: special instructions *vs.* memory-mapped I/O




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

how does OS read and write registers?

Special Instructions vs. Mem-Mapped I/O

Special instructions

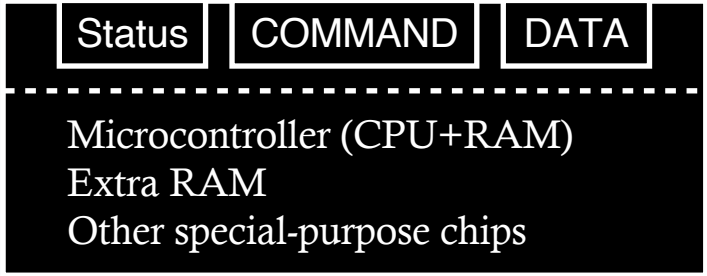
- each device has a port
- in/out instructions (x86) communicate with device

Memory-Mapped I/O

- H/W maps registers into address space
- loads/stores sent to device

Doesn't matter much (both are used)

Protocol Variants



Status checks: polling *vs.* interrupts

Data: PIO *vs.* DMA

Control: special instructions *vs.* memory-mapped I/O

Variety is a Challenge

Problem:

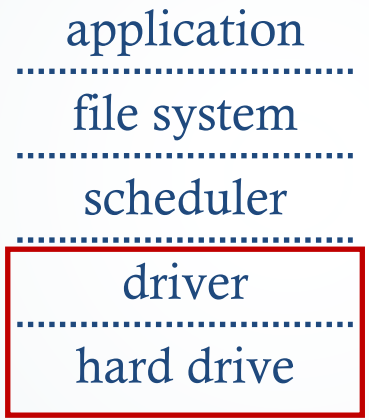
- many, many devices
- each has its own protocol

How can we avoid writing a slightly different OS for each H/W combination?

Write device driver for each device

Drivers are **70%** of Linux source code

Storage Stack



build common interface
on top of all HDDs