Persistence:
I/O Devices

OSTEP Chapter 36:
http://pages.cs.wisc.edu/~remzi/OSTEP/file-devices.pdf

Jan Reineke
Universität des Saarlandes

Motivation

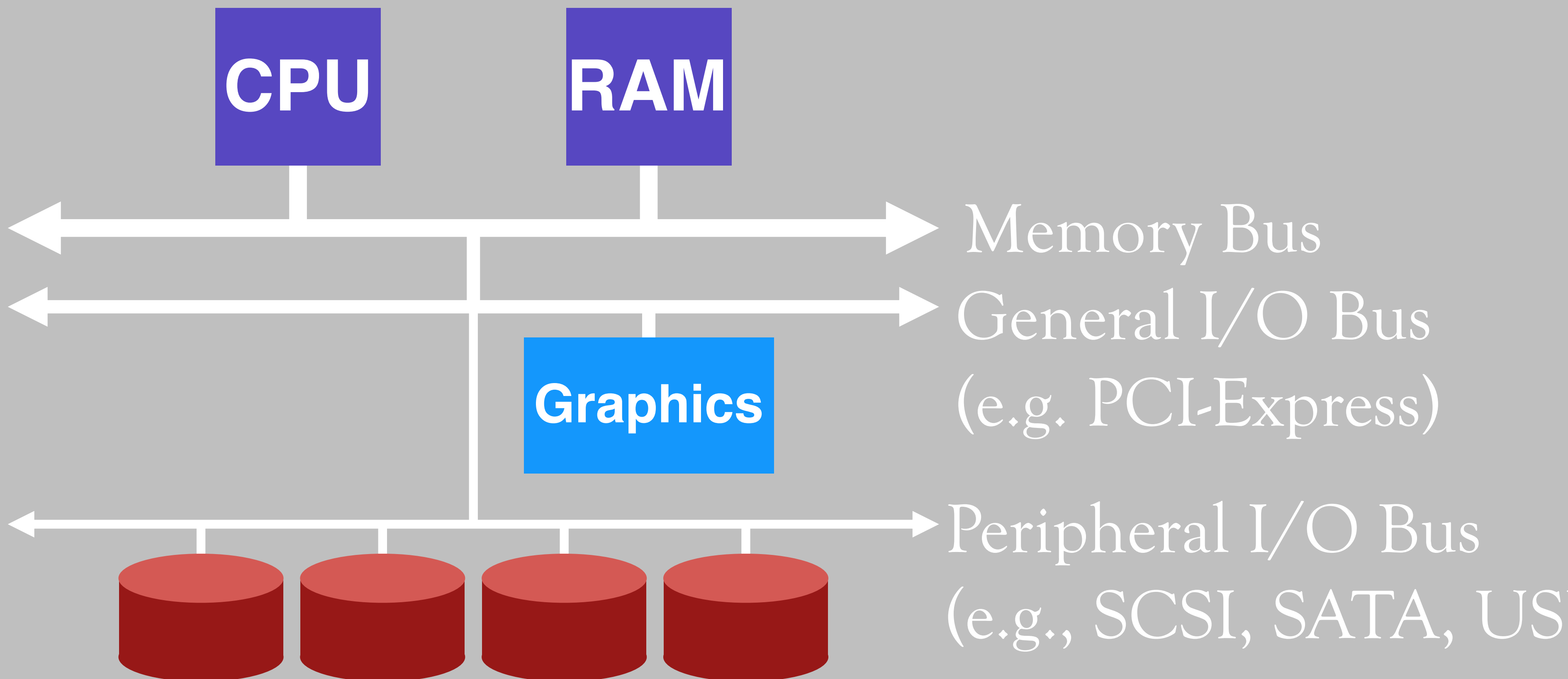
What good is a computer without any I/O devices?

- touchscreen, display, keyboard, hard disk, …
  → little ;-)

We would like:

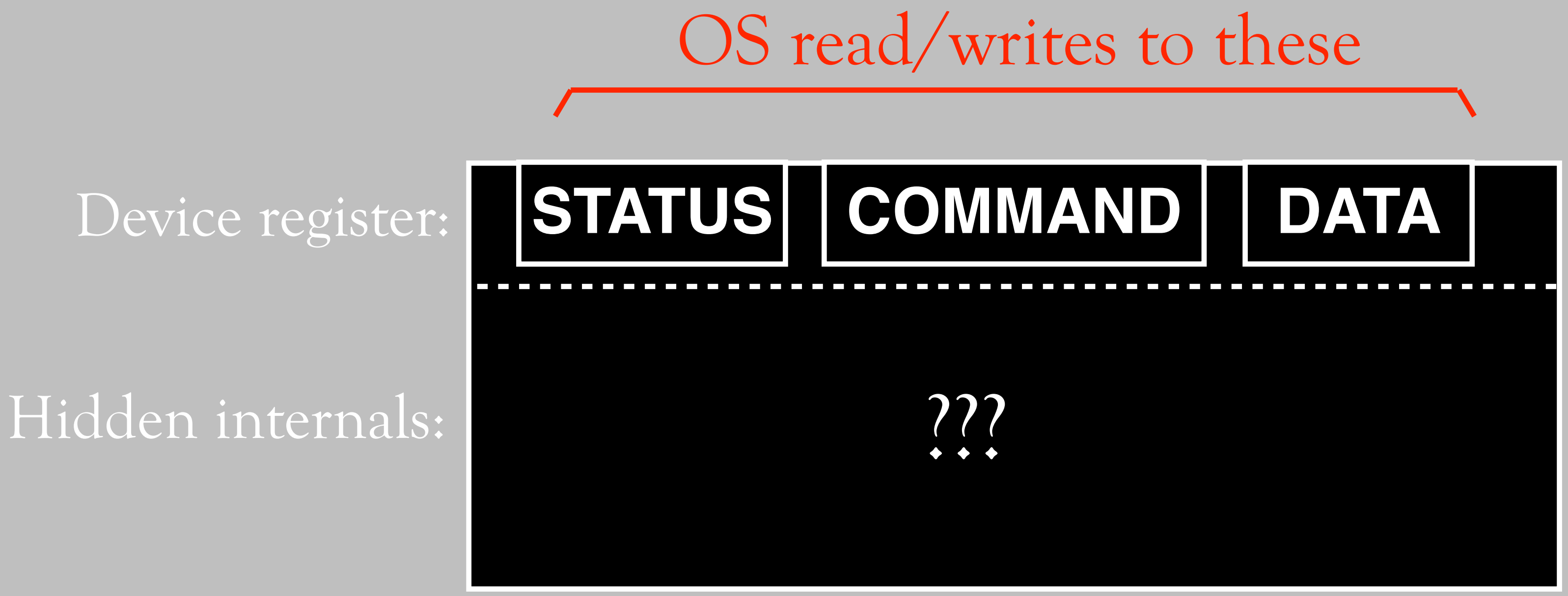
- **HW** that will let us plug in different devices
- **OS** that can interact with many combinations

Hardware support for I/O

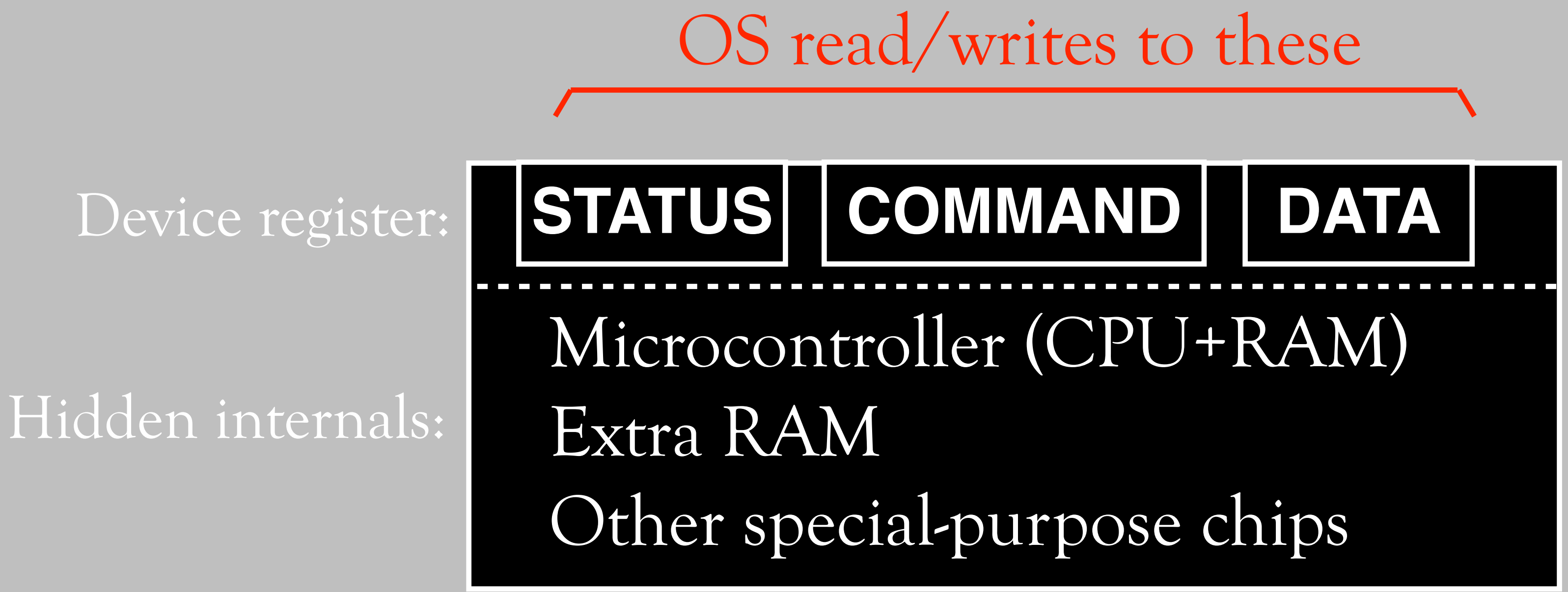


Why use hierarchical buses?

Canonical device

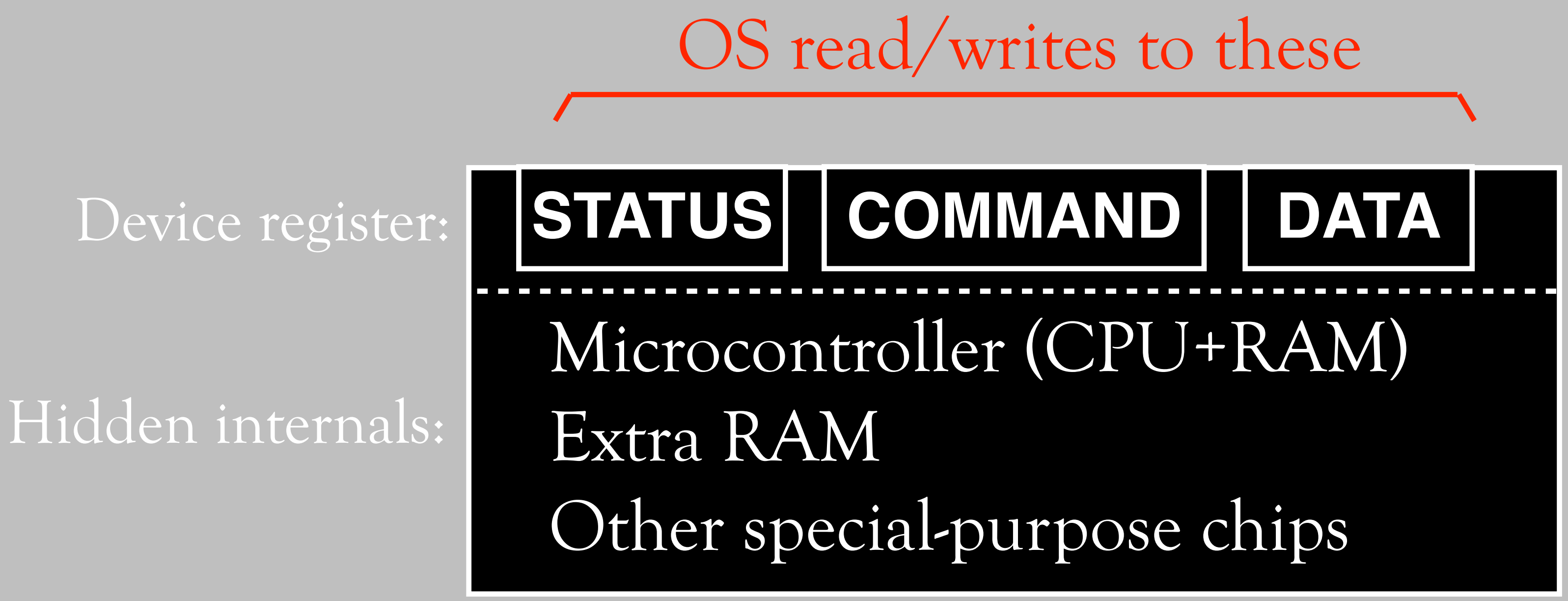


Canonical device



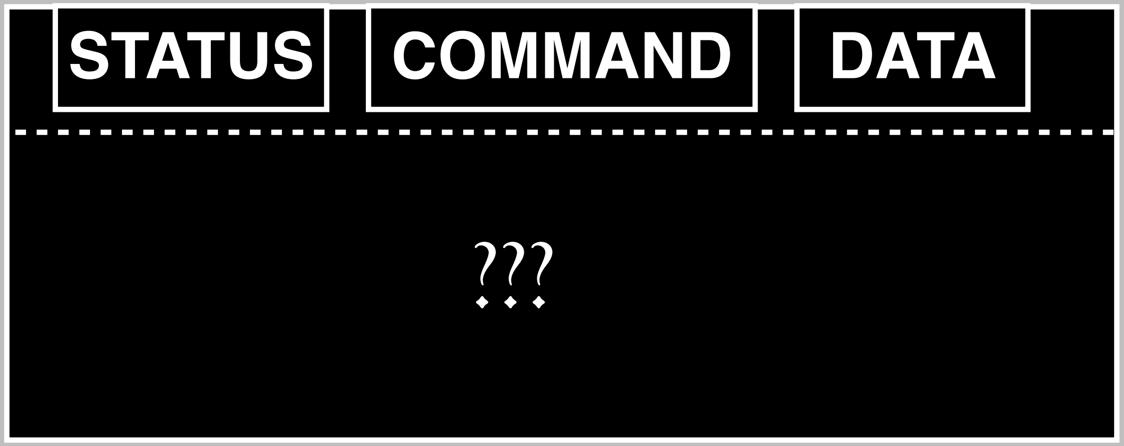
Some devices have a combined STATUS/COMMAND register

Canonical device



Some devices have a combined STATUS/COMMAND register
→ Project 2

Example Write protocol




```
while (STATUS == BUSY)
    ;

Write data to DATA register

Write command to COMMAND register

while (STATUS == BUSY)
    ;
```

CPU:

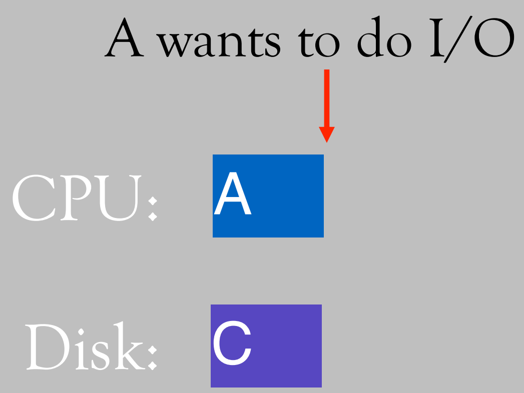
Disk:

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

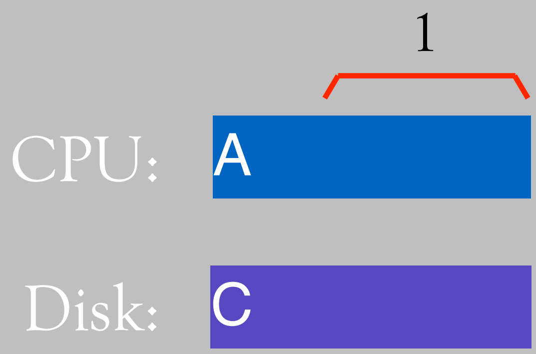
CPU: A

Disk: C

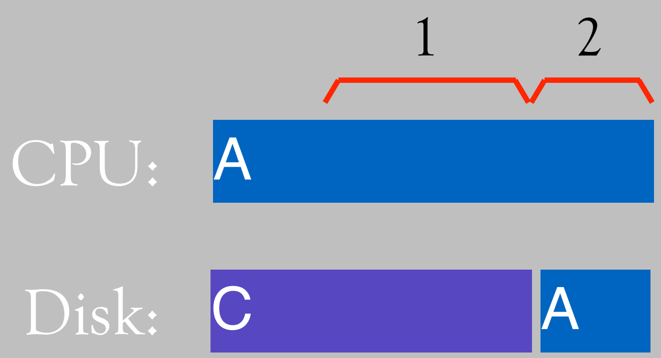
```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



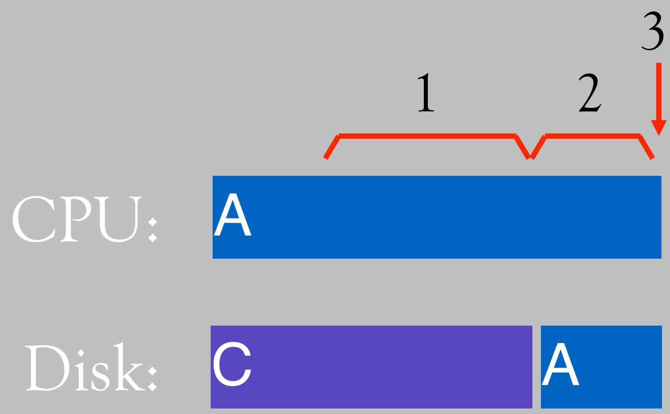

```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```



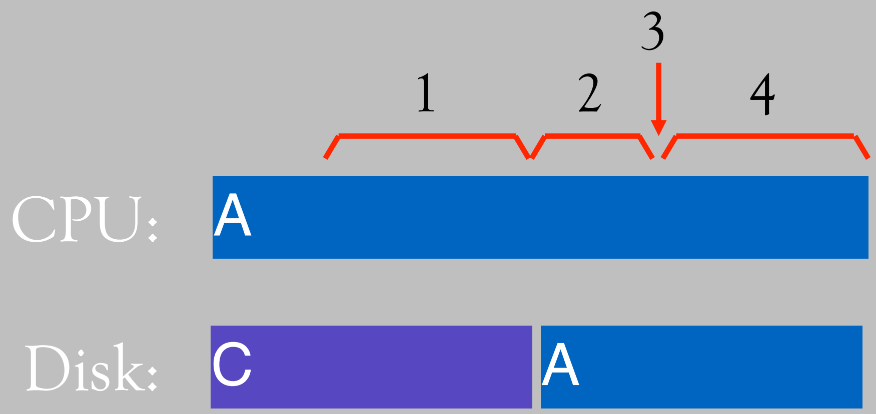

```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```



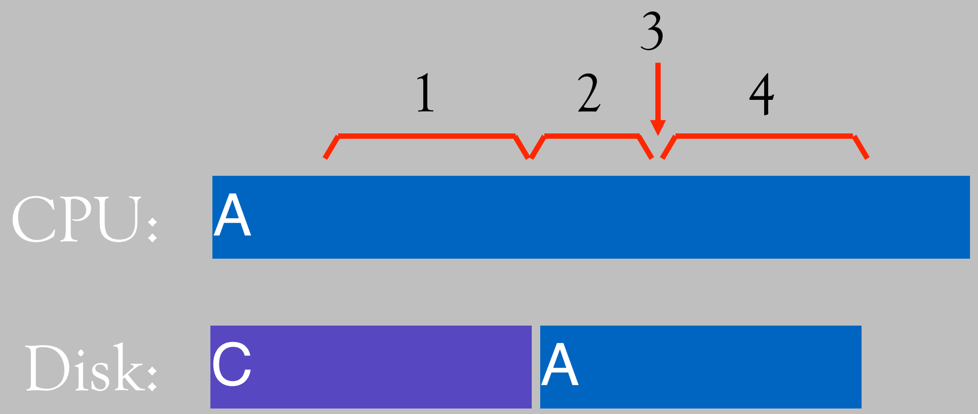

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



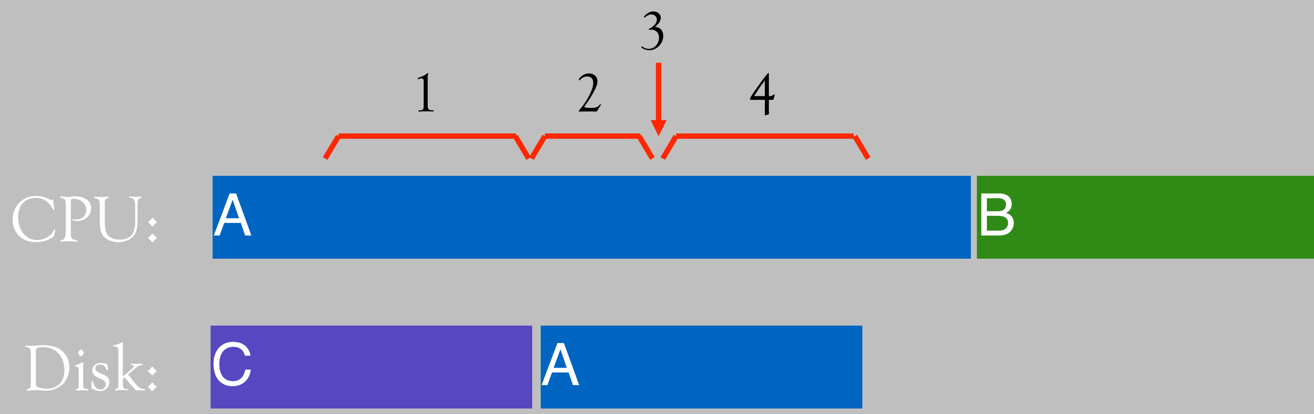

```
while (STATUS == BUSY)               // 1
  ;
Write data to DATA register          // 2
Write command to COMMAND register    // 3
while (STATUS == BUSY)               // 4
  ;
```




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



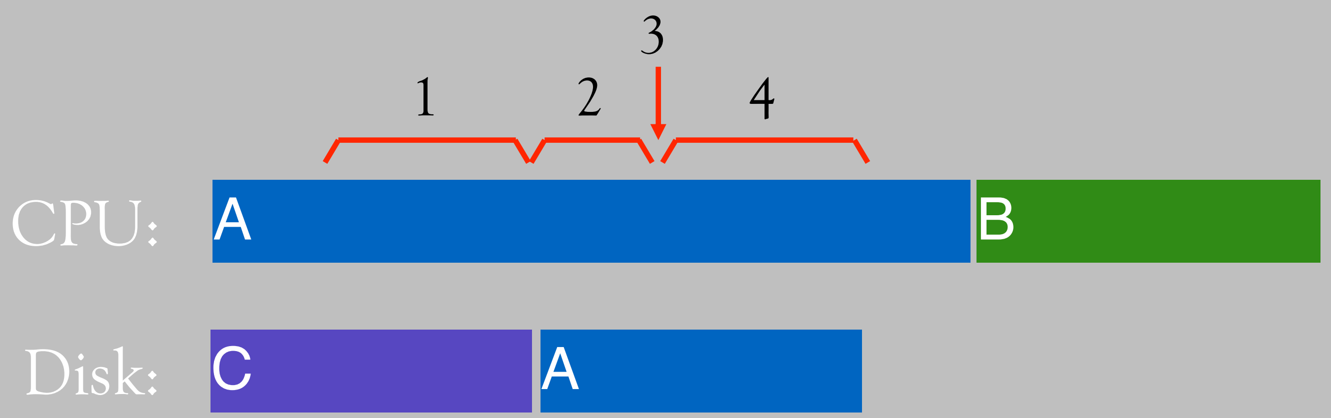

```
while (STATUS == BUSY)                    // 1
  ;
Write data to DATA register               // 2
Write command to COMMAND register         // 3
while (STATUS == BUSY)                    // 4
  ;
```




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

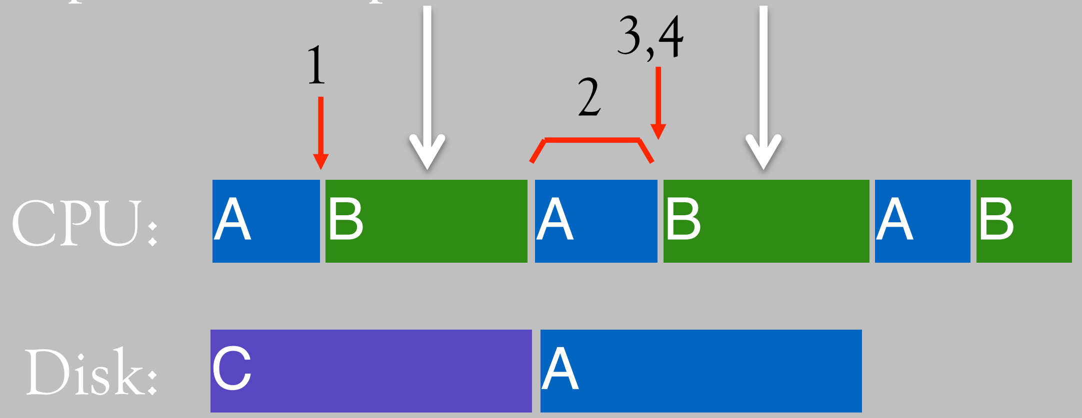
How to avoid “busy waiting” (“spinning”)?

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;
```

Overlap CPU computations and I/O via 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 every worse than polling?

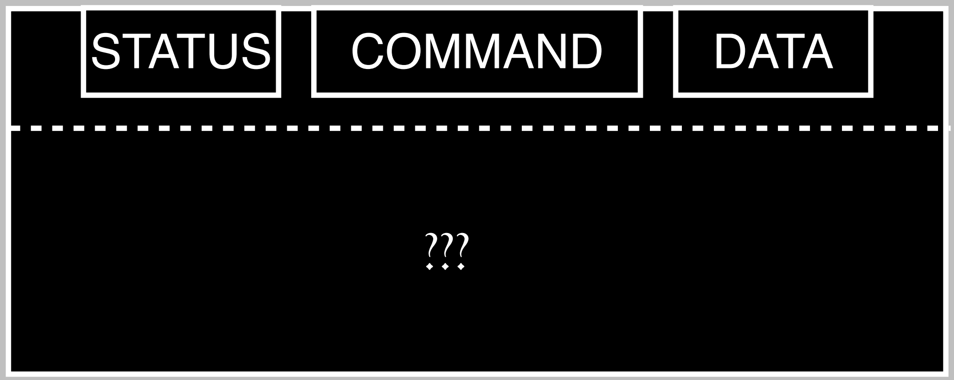
Fast device: Better to spin than take interrupt overhead

- 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
- "Interrupt coalescing"
  (batch together several interrupts)

Protocol variants



- **Status checks**: polling *vs.* interrupts
- **Data**: Programmed-IO *vs.* DMA
- **Control**: special instructions vs. memory-mapped I/O

Programmed I/O vs. Direct Memory Access

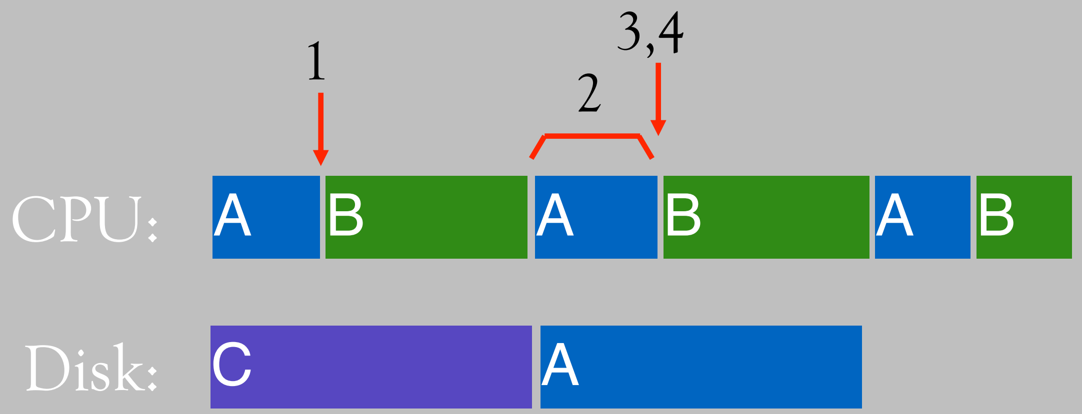
Programmed I/O (**PIO**):

– CPU directly tells device what the data is

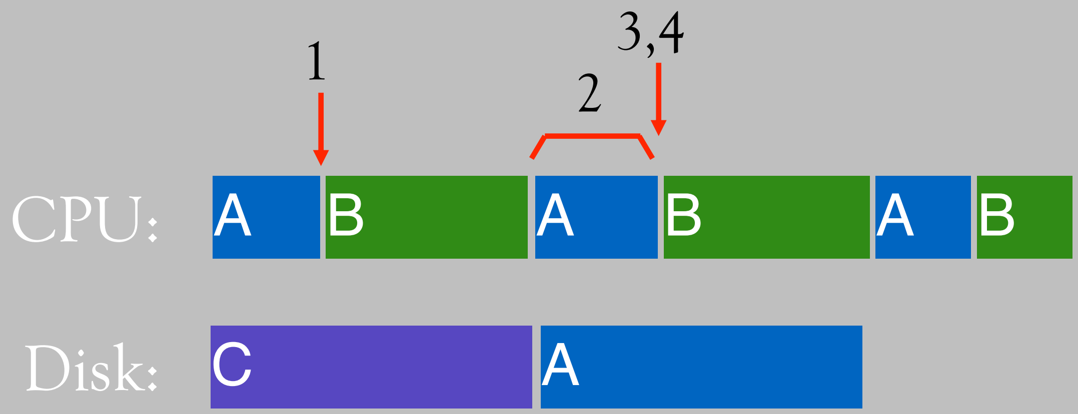
Direct Memory Access (**DMA**):

– CPU leaves data in memory

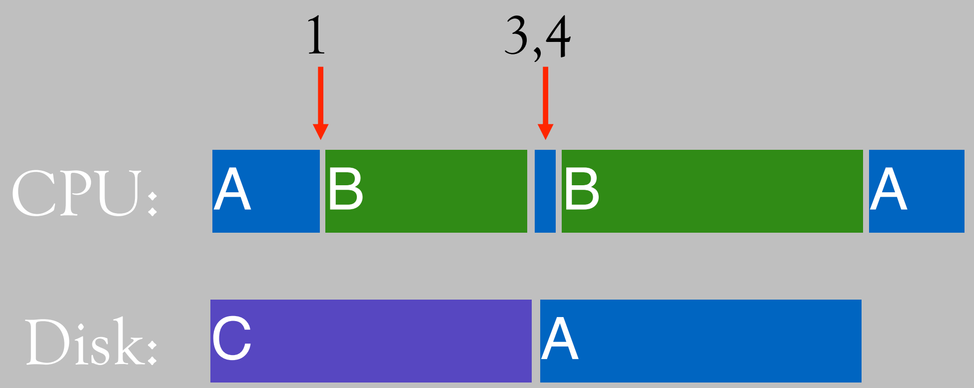
– Device reads data directly from memory




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

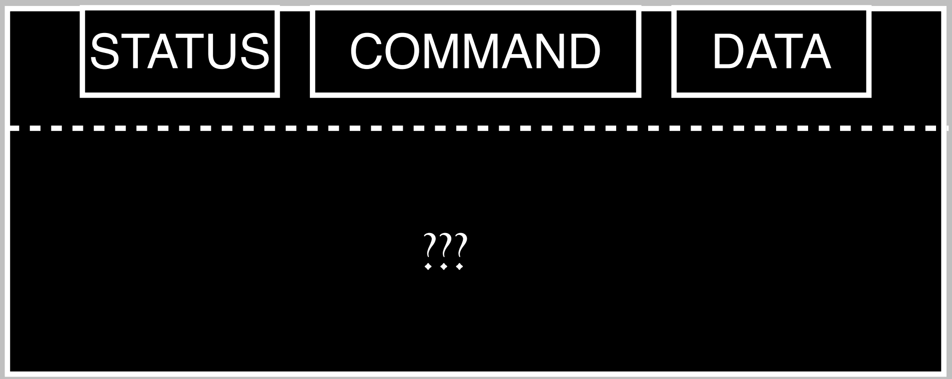



```
while (STATUS == BUSY)                    // 1
  ;
Write data to DATA register               // 2
Write command to COMMAND register         // 3
while (STATUS == BUSY)                    // 4
  ;
```

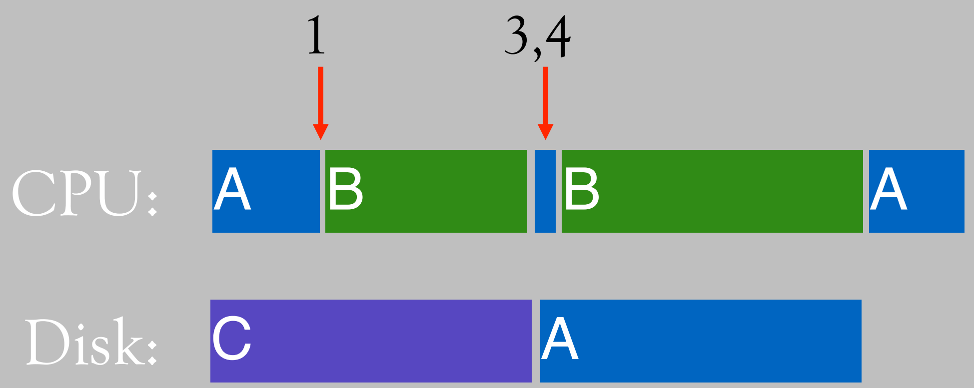



```
while (STATUS == BUSY)                  // 1
  ;
```
~~Write data to DATA register   // 2~~
```
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```

Protocol variants



- **Status checks**: polling *vs.* interrupts
- **Data**: Programmed-IO *vs.* DMA
- **Control**: special instructions vs. memory-mapped I/O



`while (STATUS == BUSY)                       // 1`

`    ;`

~~Write data to DATA register   // 2~~

`Write command to COMMAND register            // 3`

`while (STATUS == BUSY)                       // 4`

`    ;`

How does OS read and write registers?

Special instructions vs. Memory-Mapped I/O

Special instructions:

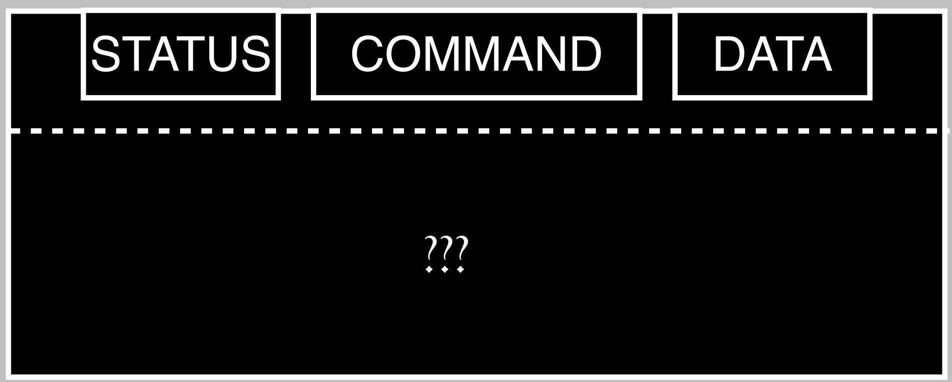
- each device has separate port
- in/out instructions (x86) communicate with device

Memory-Mapped I/O:

- HW maps registers into address space
- Loads and stores are forwarded to the respective devices

Doesn't matter much
(both are used)

Protocol variants



- **Status checks**: polling *vs.* interrupts
- **Data**: Programmed-IO *vs.* DMA
- **Control**: special instructions vs. memory-mapped I/O

Variety is a challenge

Problem:

many, many devices

each has its own protocol

New OS variant for each new device?

Better: new **driver** for each new device, but standardized interfaces

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

Example: Abstraction layers

Application

File system

Driver

Hard disk

fixed interface for all hard disks

new driver for each new device

Summary: I/O Devices

Overlap I/O and computations whenever possible:

- Interrupts
- Direct Memory Access