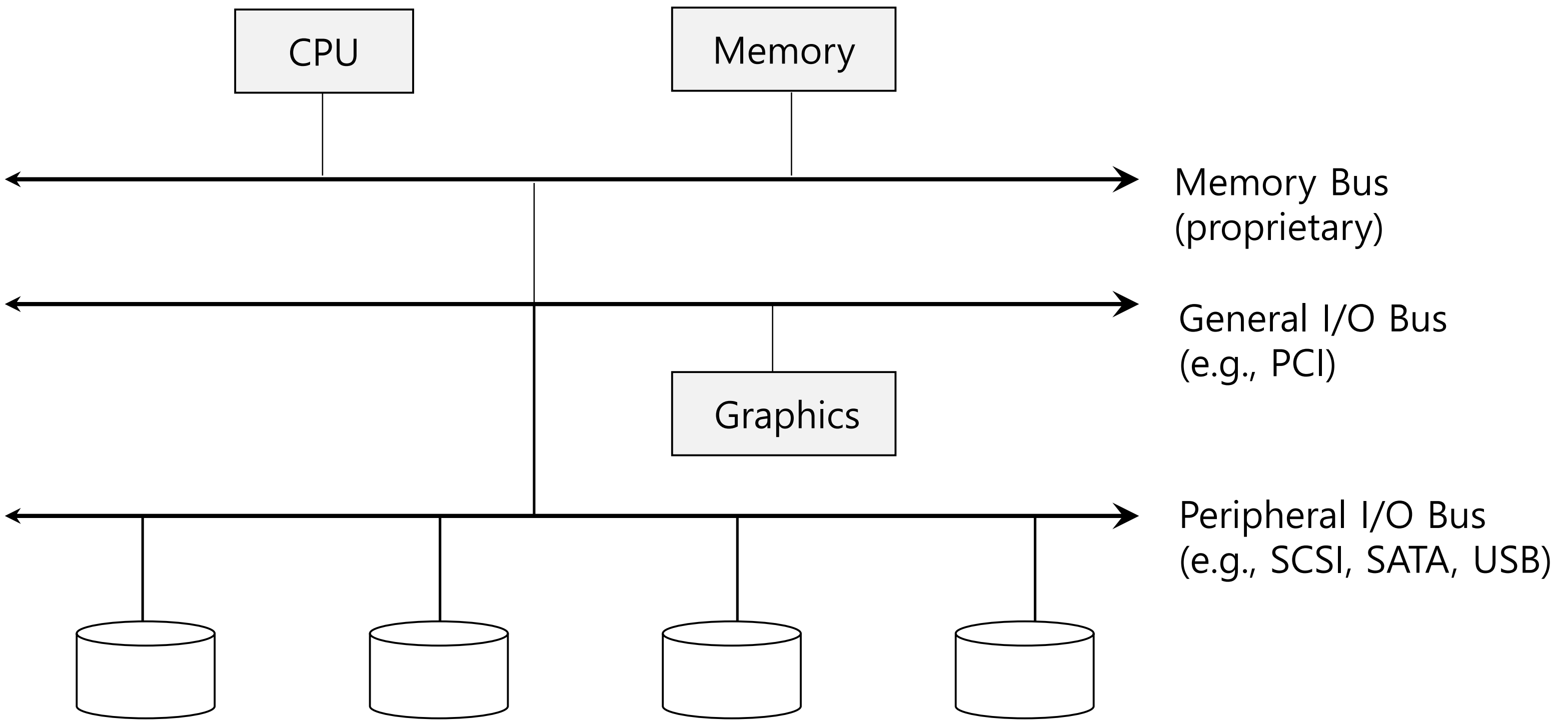
Persistence: I/O Devices

Sridhar Alagar

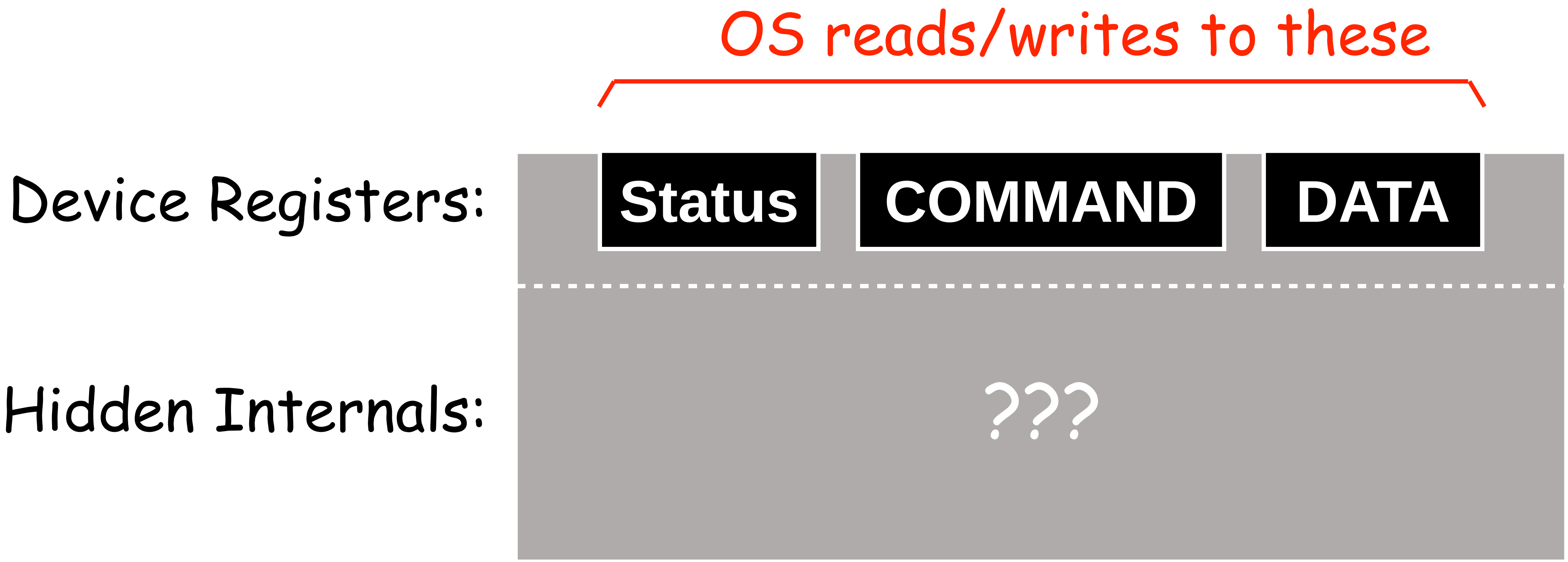
Motivation

- What good is a computer without any I/O devices?
  - keyboard, display, disks
- We want:
  - H/W that will let us plug in different devices
  - OS that can interact with different combinations

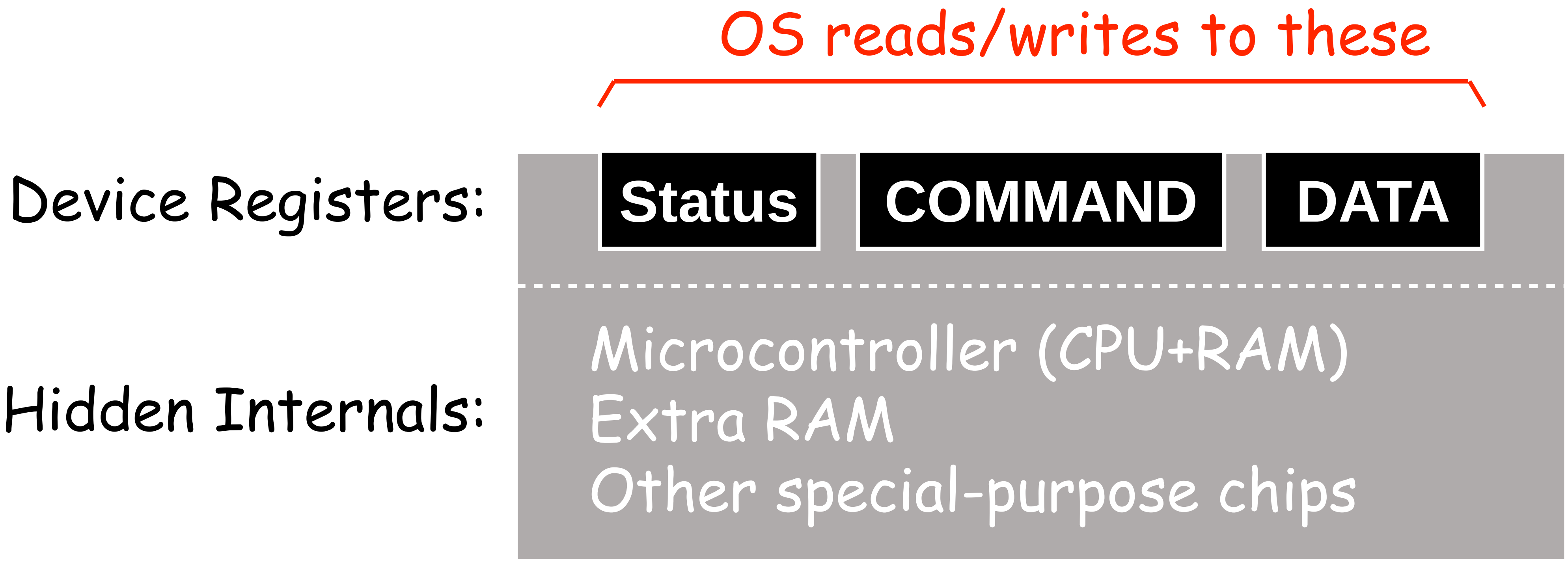
System Architecture



Canonical Device

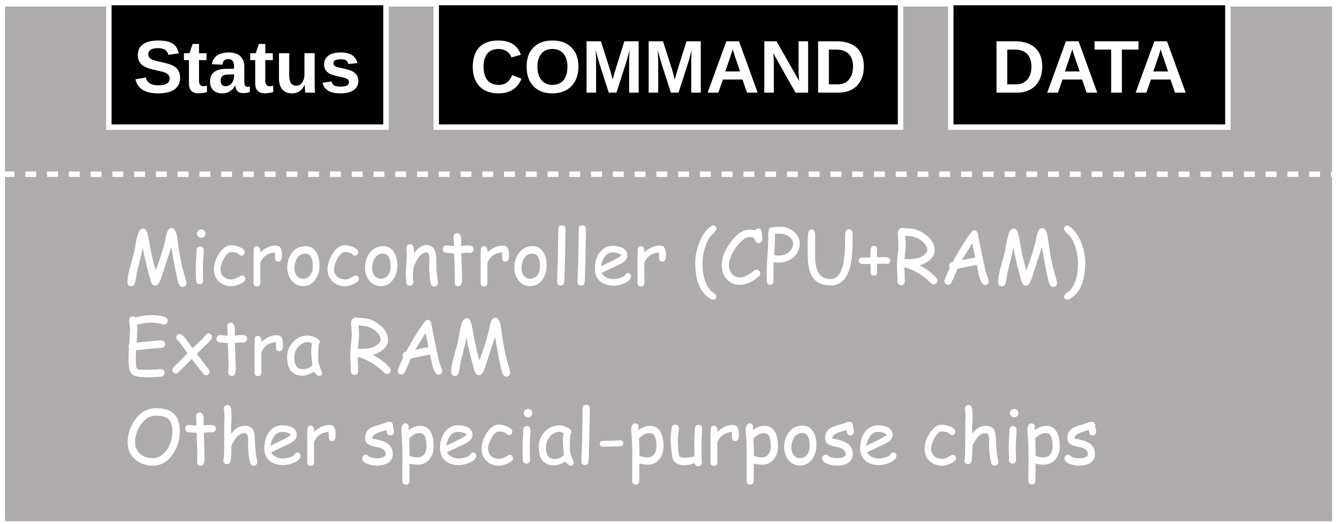


Canonical Device



Example Write Protocol

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



CPU: A

Disk: C

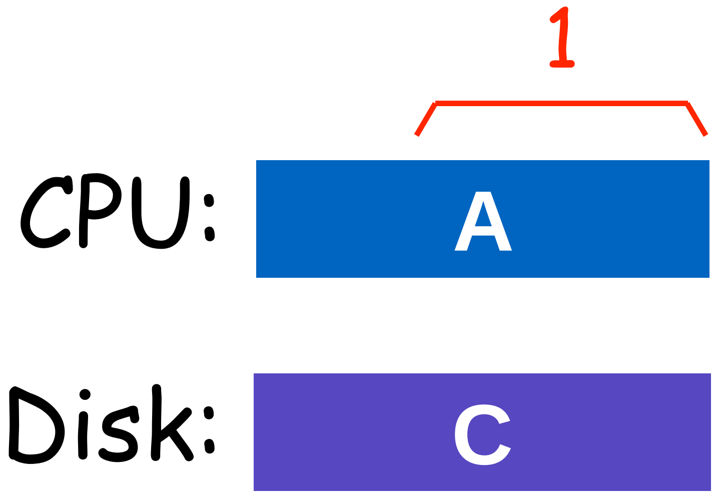
```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```

A wants to do I/O

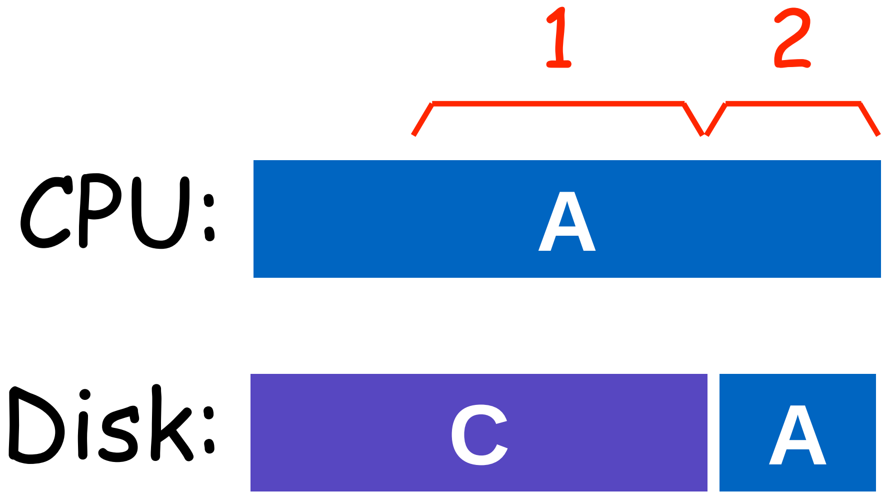
CPU: A

Disk: C

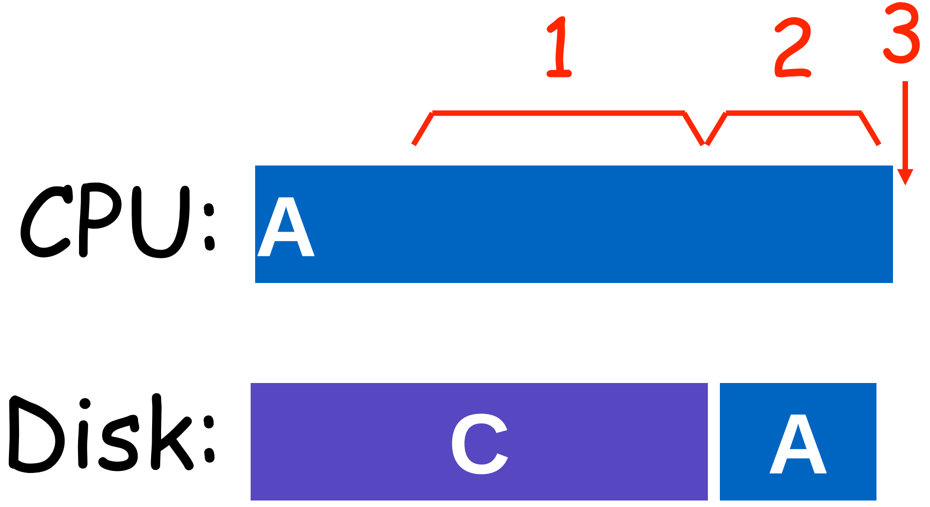
```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```



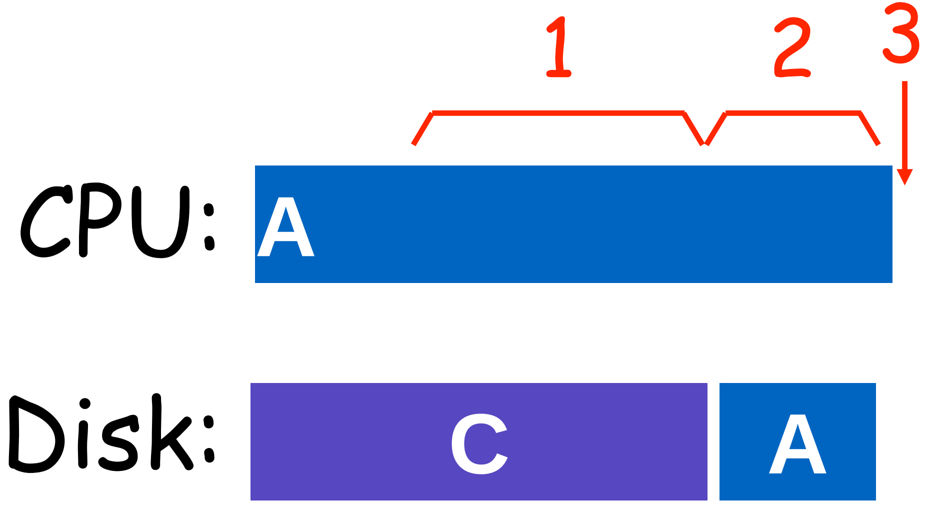

```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```



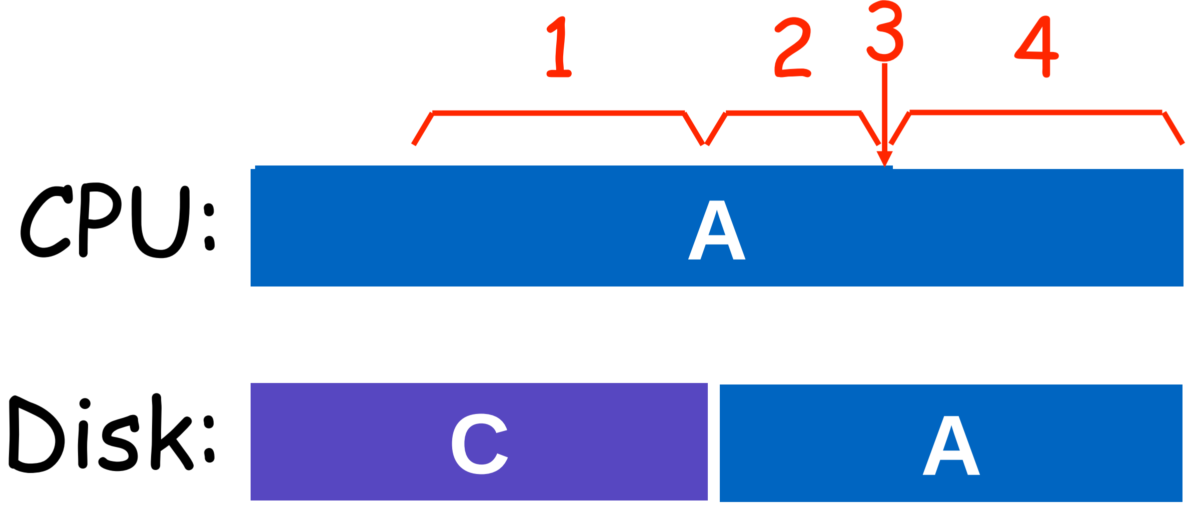

```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```



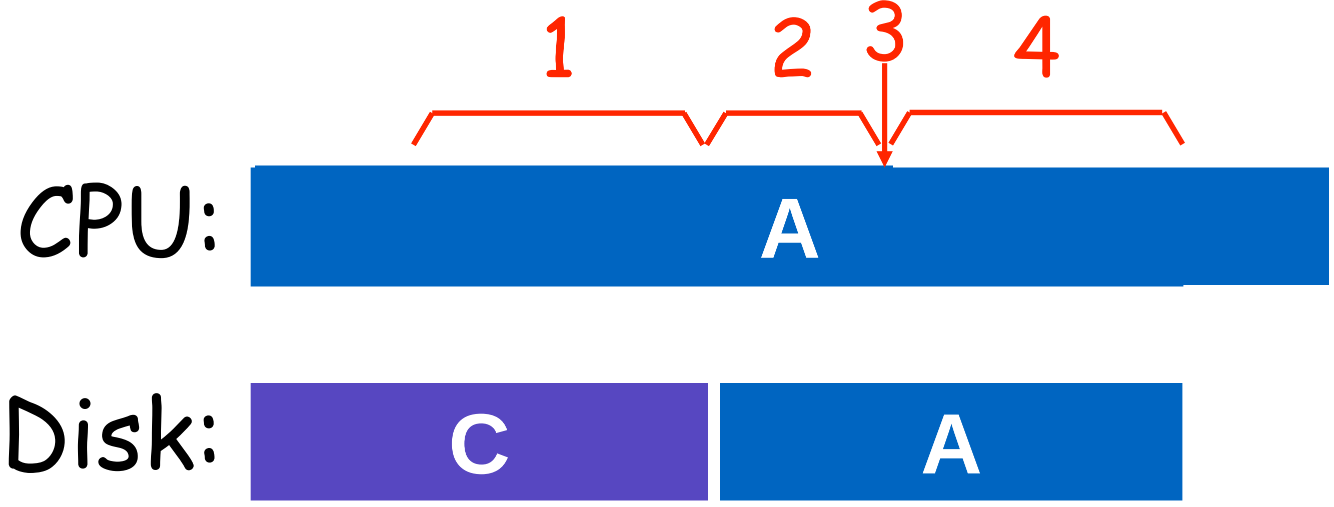

```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```



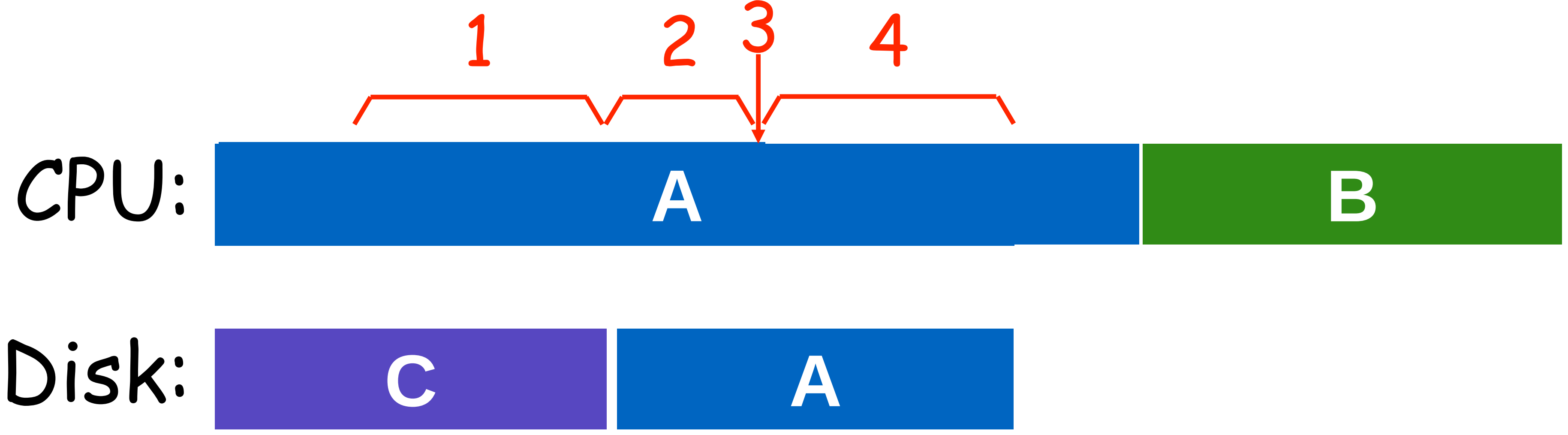

```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```




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

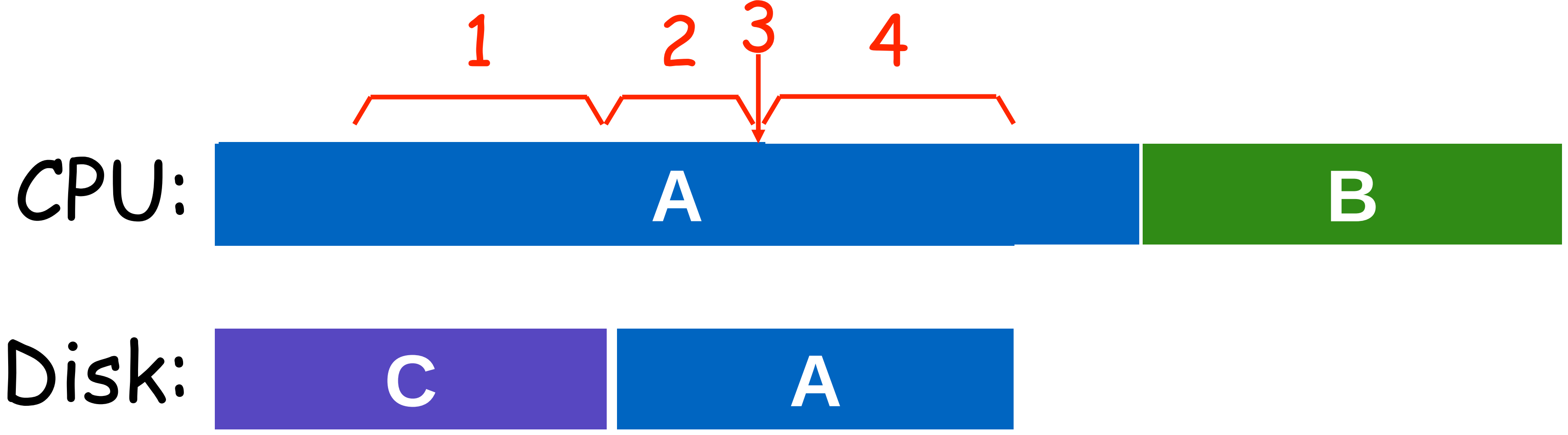



```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```

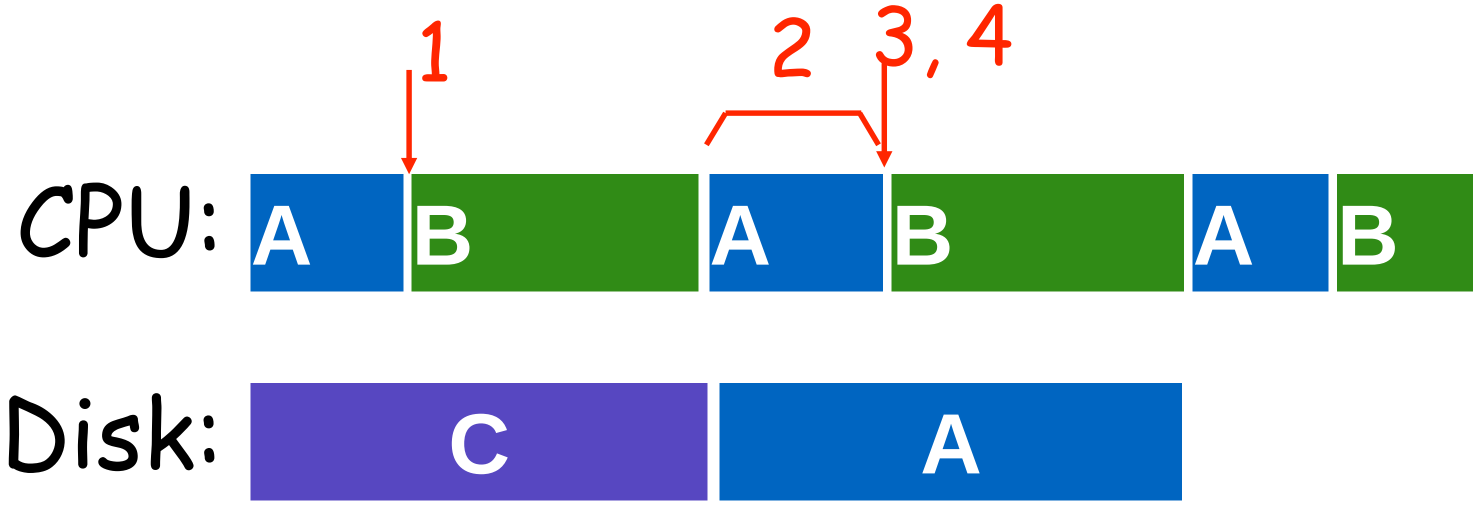



```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```

how to avoid 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;
```




```
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;
```

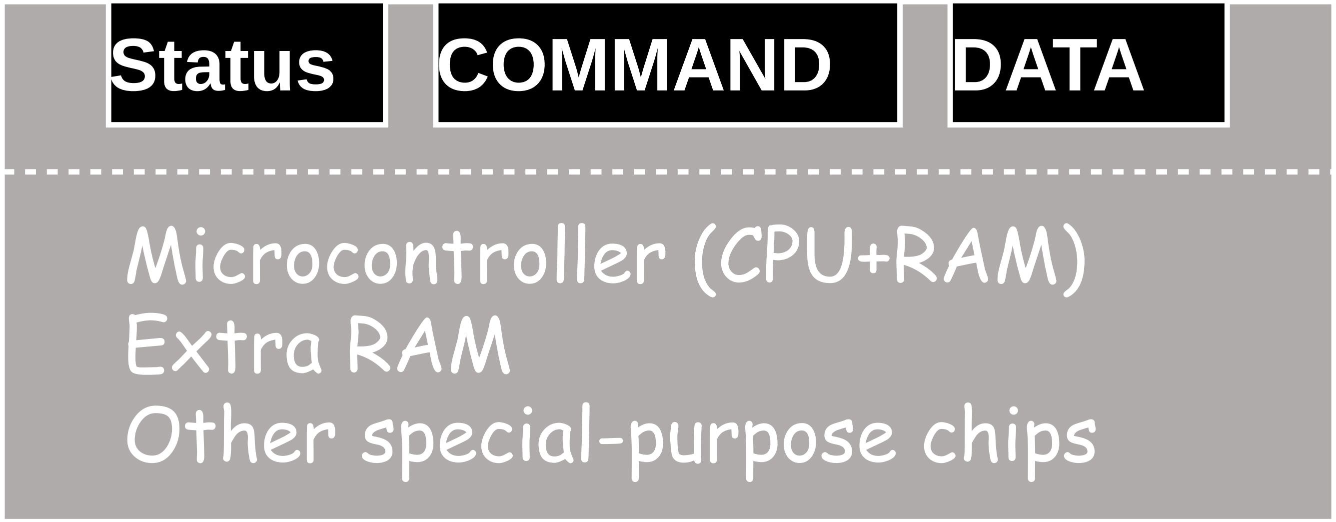
Interrupts vs Polling

- Are interrupts always better than polling?

- Fast device: Better to spin than take interrupt overhead
- Device time unknown?
  - Hybrid approach (spin then use interrupts)

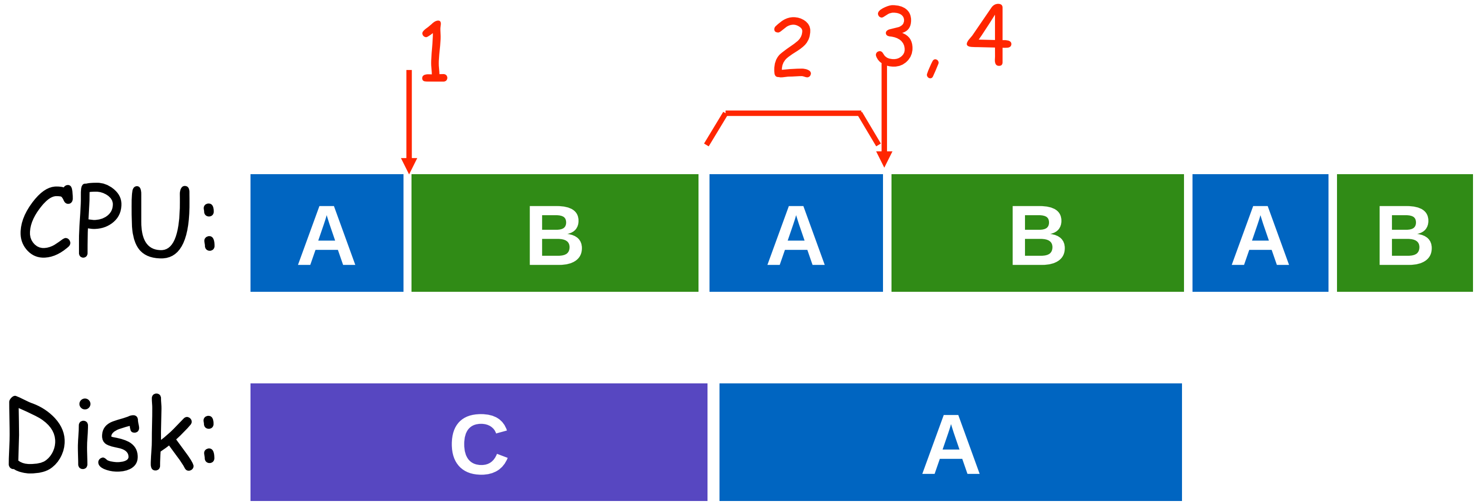
- Flood of interrupts arrive
  - Can lead to livelock (always handling interrupts)
  - Better to ignore interrupts while make some progress handling them

Protocol Variants

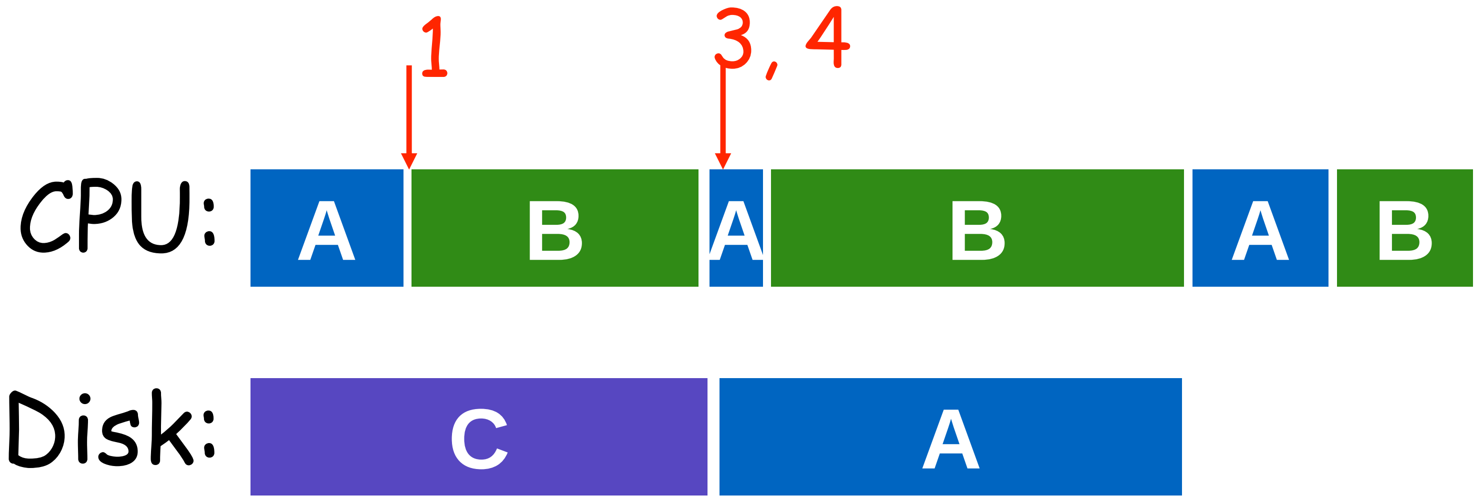


Status checks: polling vs interrupts

Data: PIO vs DMA




```
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;

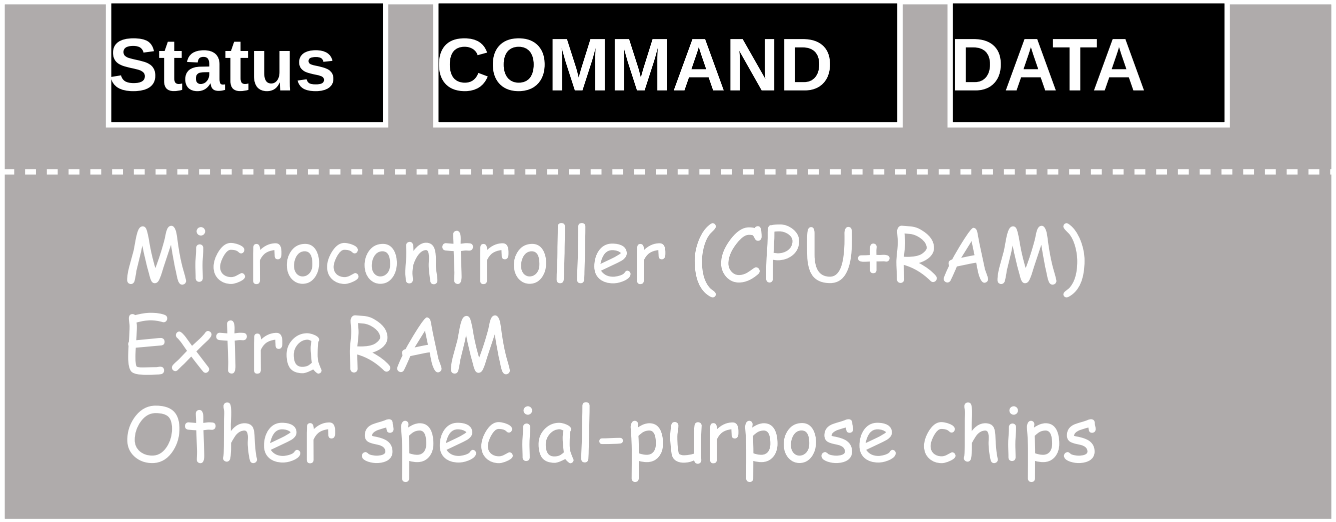
~~Write data to DATA register // 2~~

Write command to COMMAND register // 3

while (STATUS == BUSY) // 4

wait for interrupt;

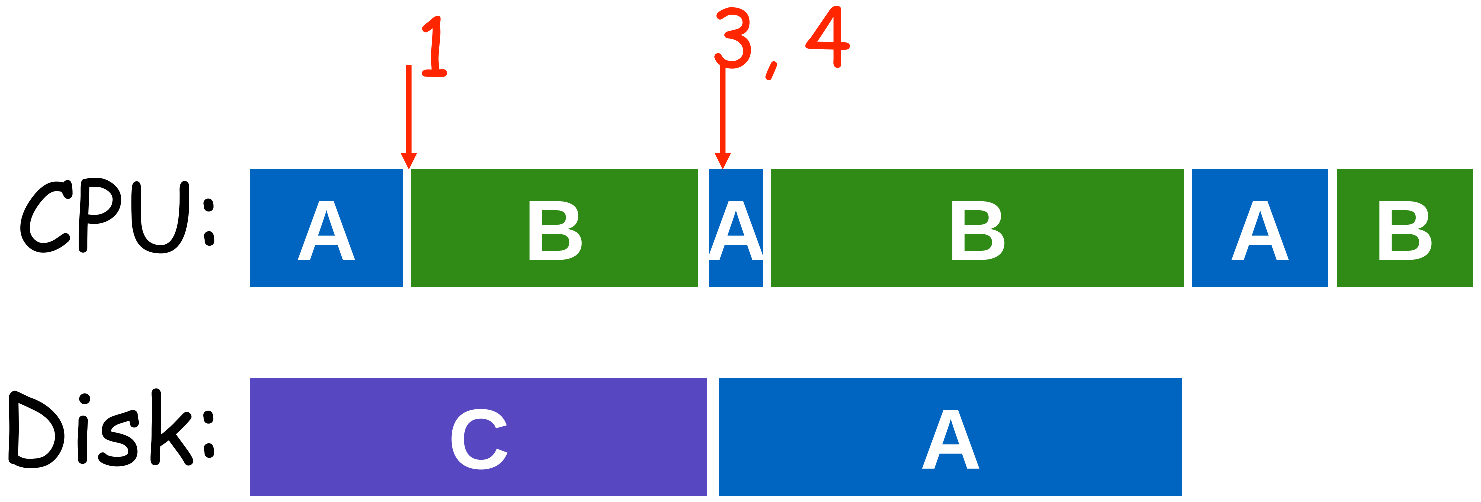
Protocol Variants



Status checks: polling vs interrupts

Data: PIO vs DMA

Access: special instruction 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 access device registers?

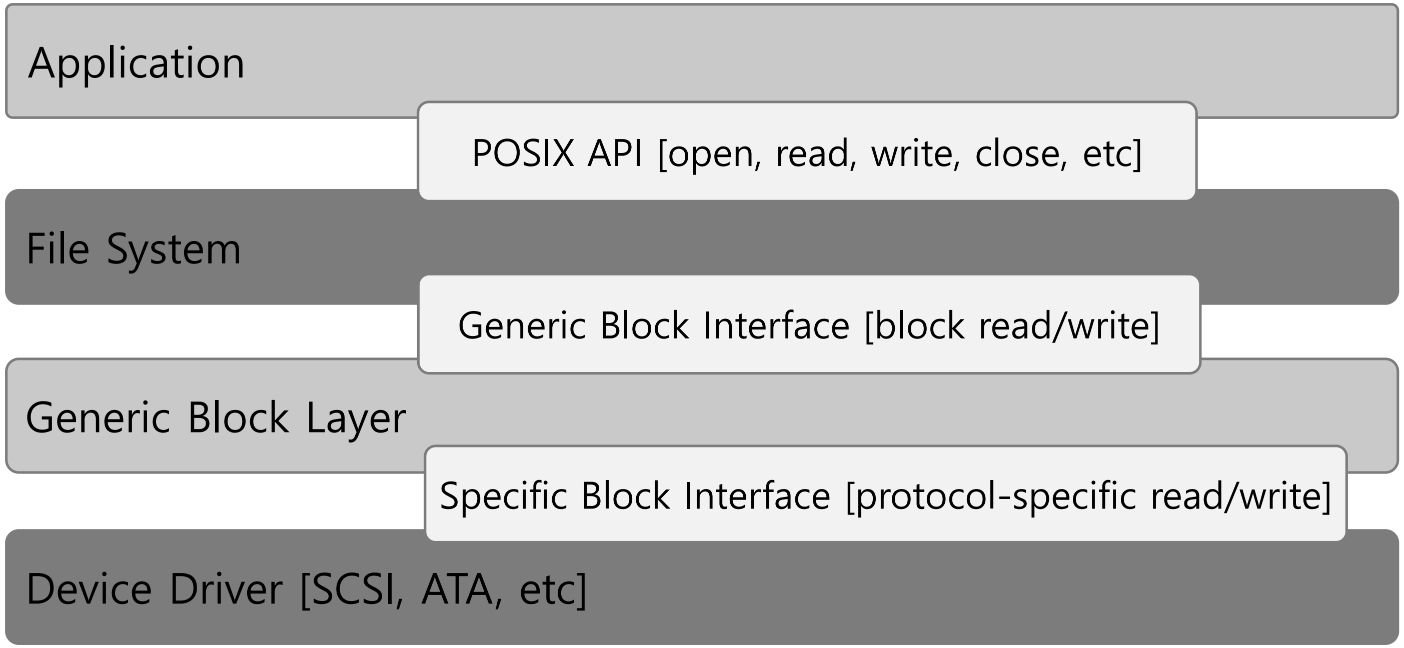
- Special instructions
  - each device has a port
  - in/out instructions (x86) communicate with device
- Memory-Mapped I/O
  - H/W maps registers into address space
  - loads/stores sent to device
- Doesn't matter much (both are used)

Variety is a Challenge

- Many devices
  - Various characteristics
  - Each with its own protocol
- Cannot write a different OS for each H/W
- Device driver provides the abstraction

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner	400 KB/sec
Digital camcorder	3.5 MB/sec
802.11g Wireless	6.75 MB/sec
52x CD-ROM	7.8 MB/sec
Fast Ethernet	12.5 MB/sec
Compact flash card	40 MB/sec
FireWire (IEEE 1394)	50 MB/sec
USB 2.0	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
SATA disk drive	300 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec

Storage Stack

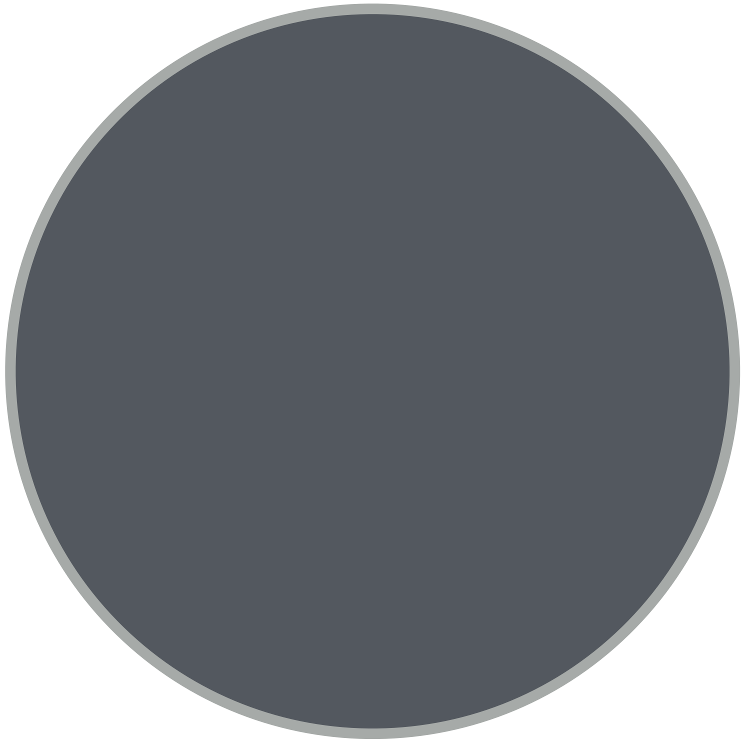


Hard Disks

Basic Interface

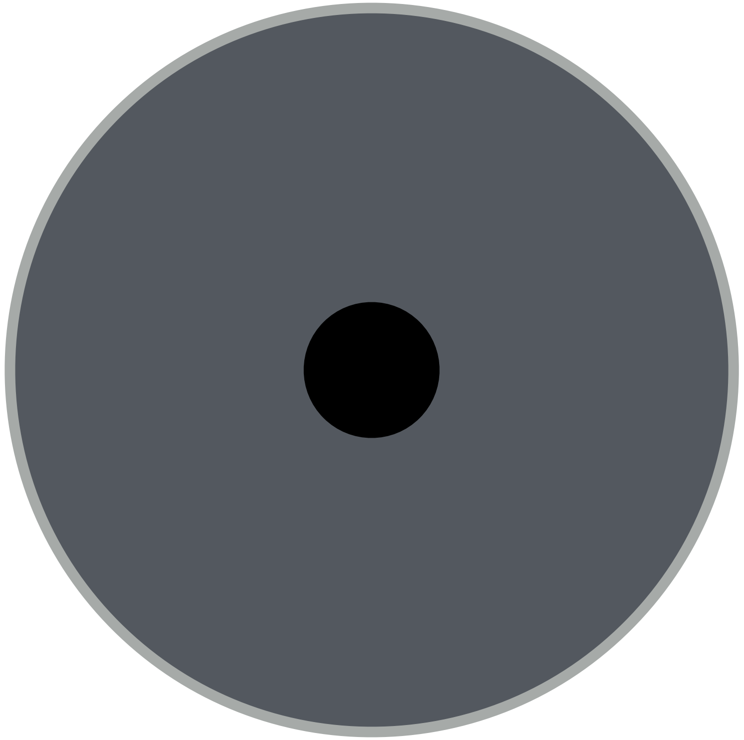
- Disk has a sector-addressable address space
  - Appears as an array of sectors
- Sectors are typically 512 bytes or 4096 bytes.
- Main operations: reads + writes to sectors
- Mechanical (slow) nature makes management "interesting"

Disk Internals - Platter

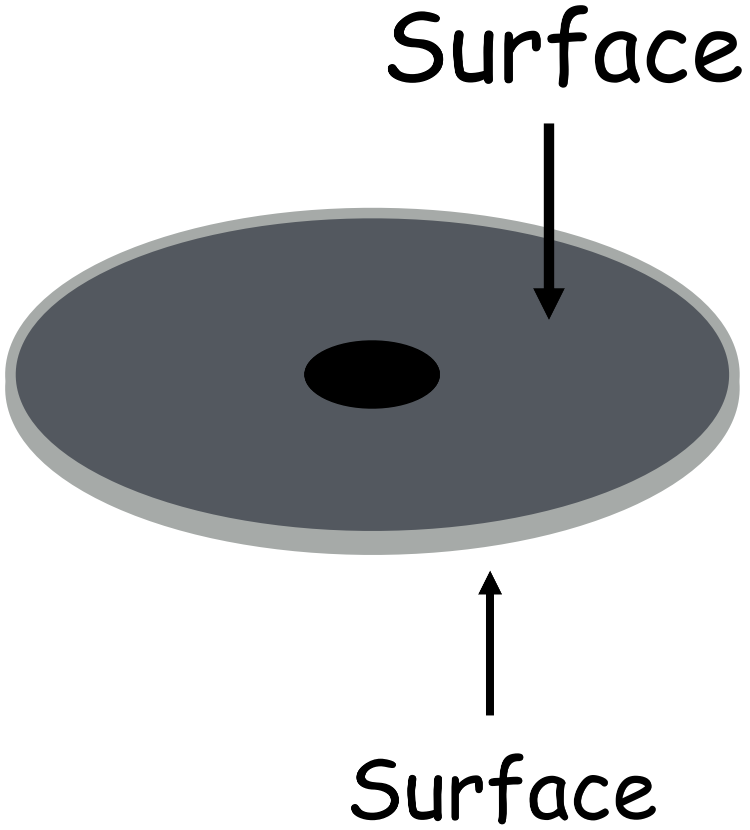


Platter is covered with a magnetic film.

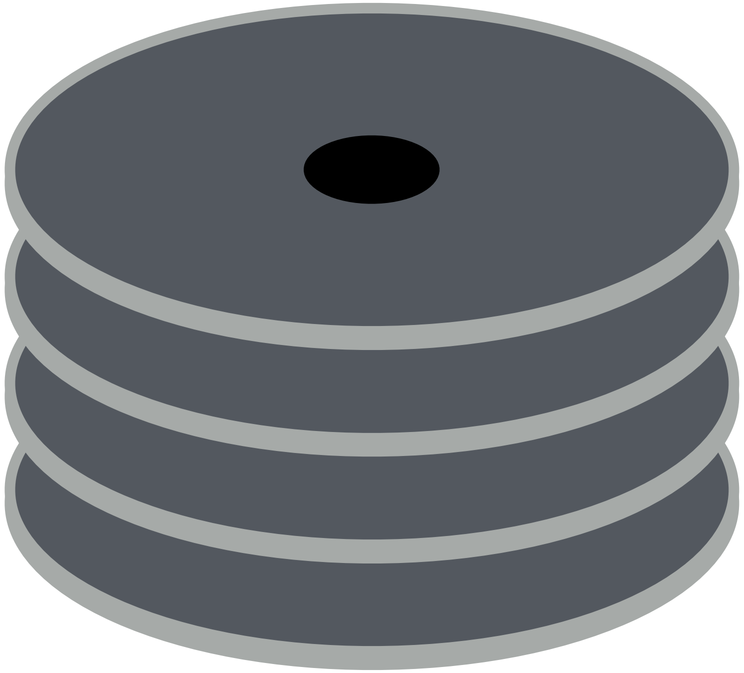
Disk Internals - Spindle



Disk Internals - Dual sided

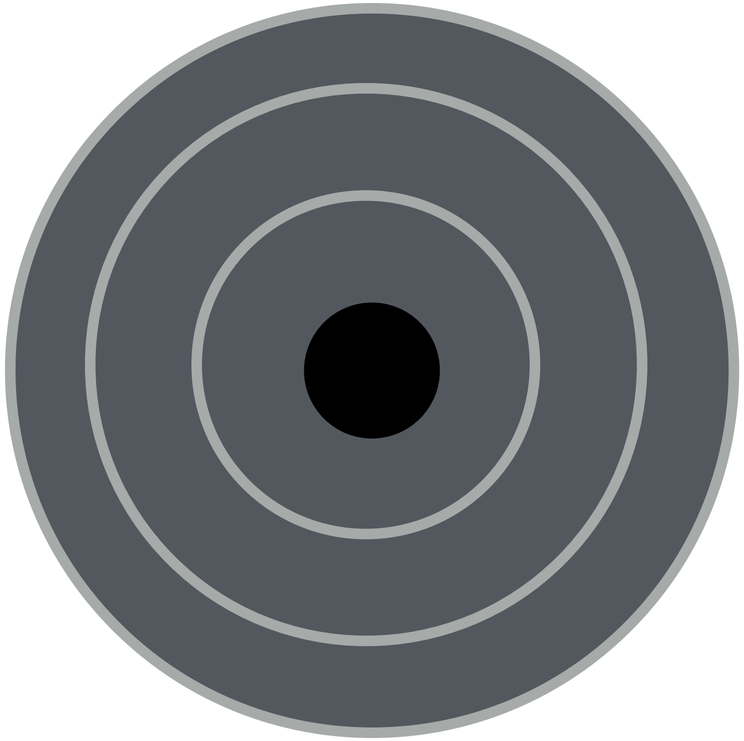


Disk Internals - many platters



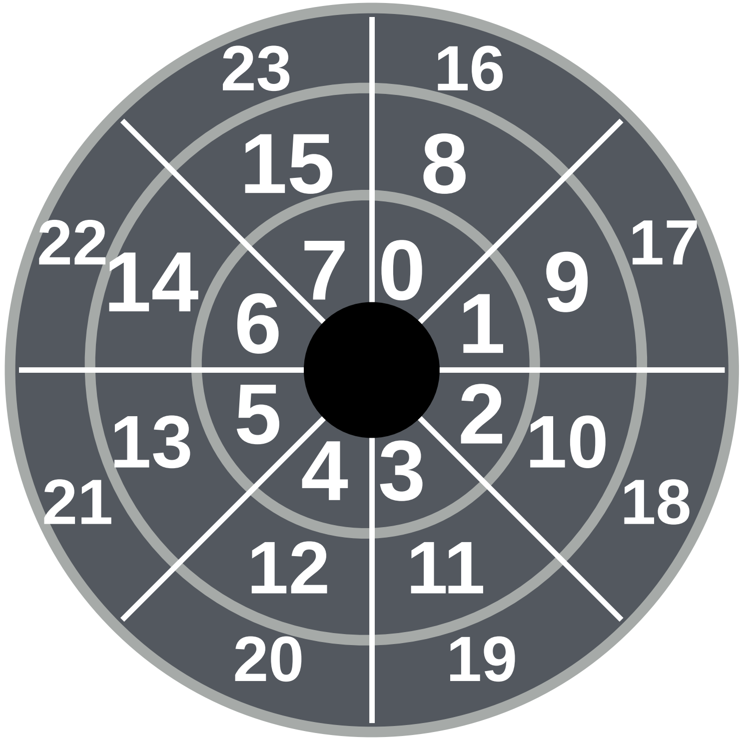
Disk Internals - tracks

- Each surface is divided into rings called tracks.
- A stack of tracks (across platters) is called a cylinder.



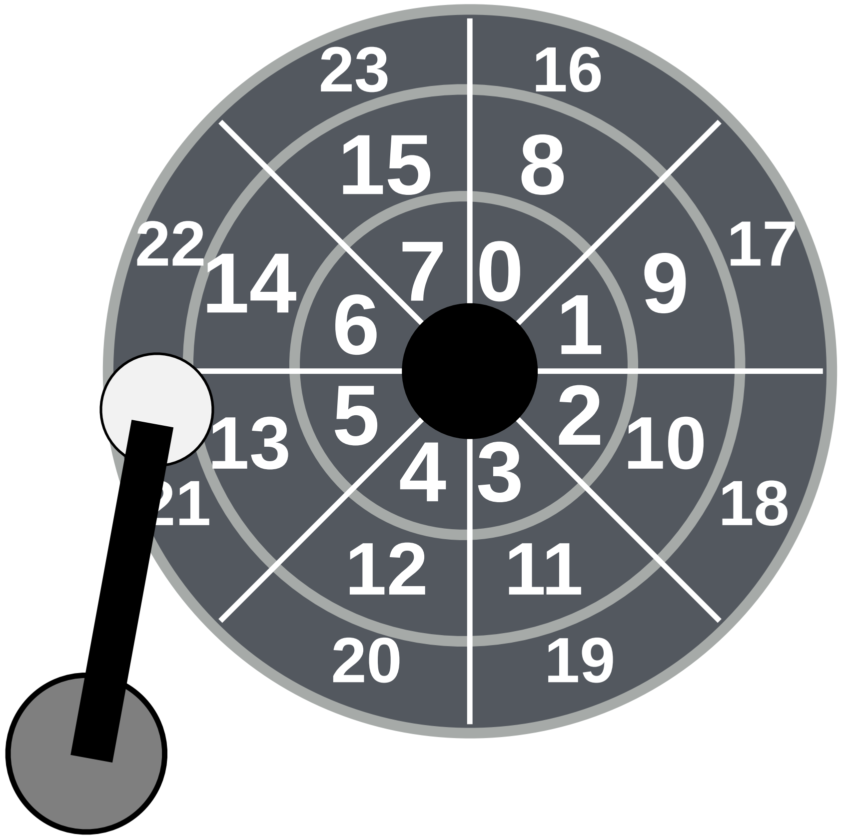
Disk Internals - sectors

The tracks are divided into numbered sectors.



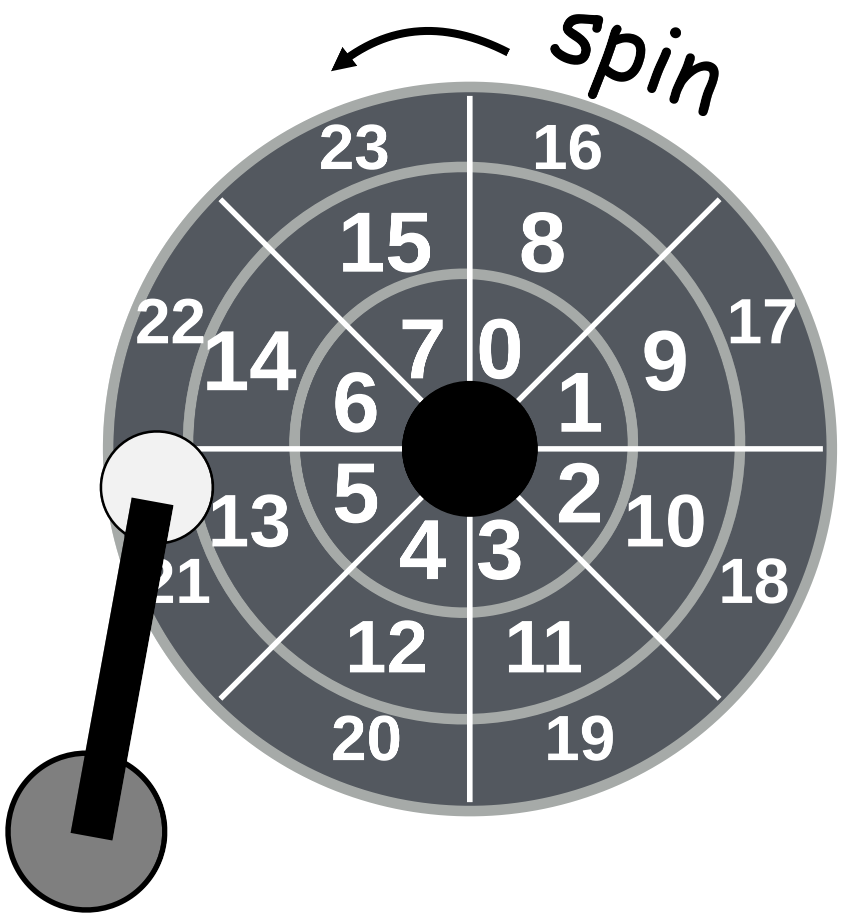
Disk Internals - heads

<u>Heads</u> on a moving <u>arm</u> can read from each surface.

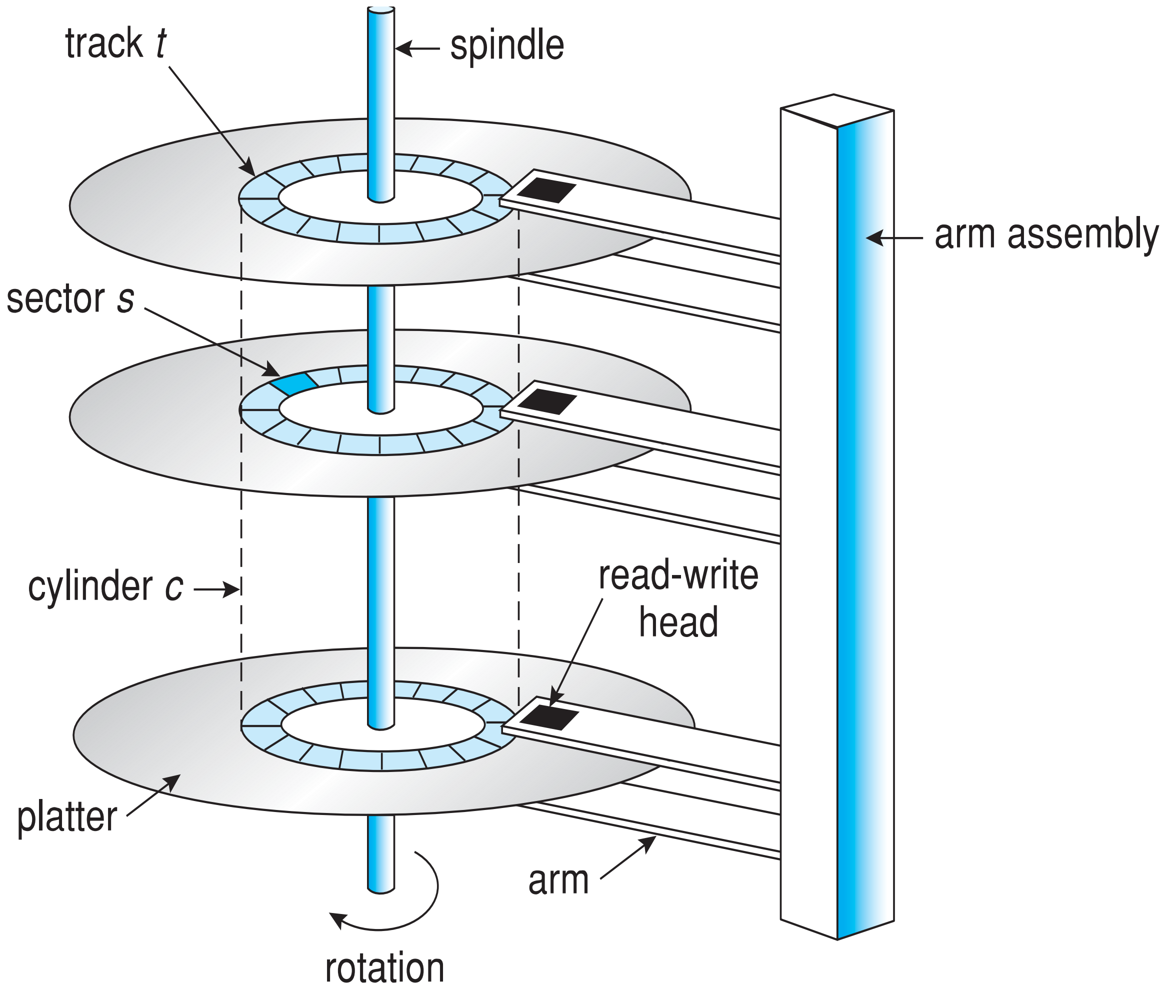


Disk Internals - heads

Spindle/platters spin.

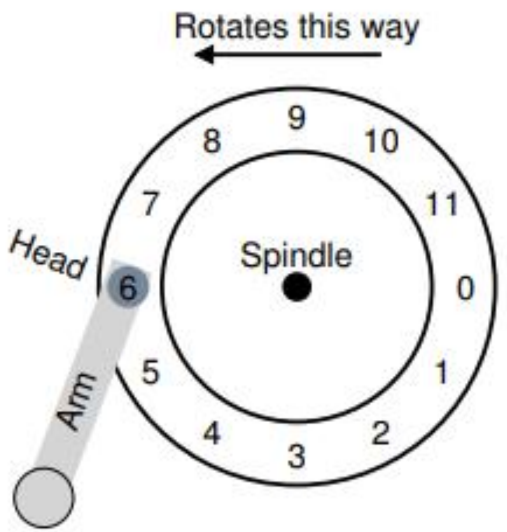


Disk - Full view



How to read a sector in the same track?

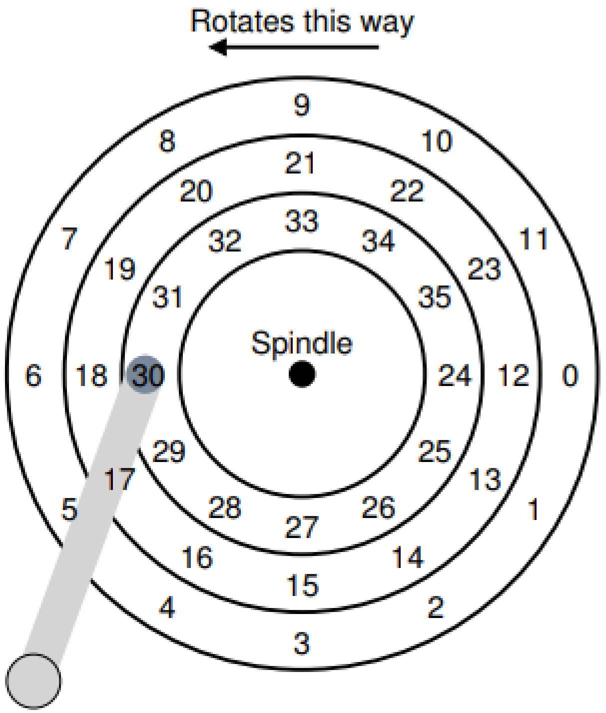
- How to read sector 0?
- Rotational delay
  - time for desired sector to rotate under the disk head

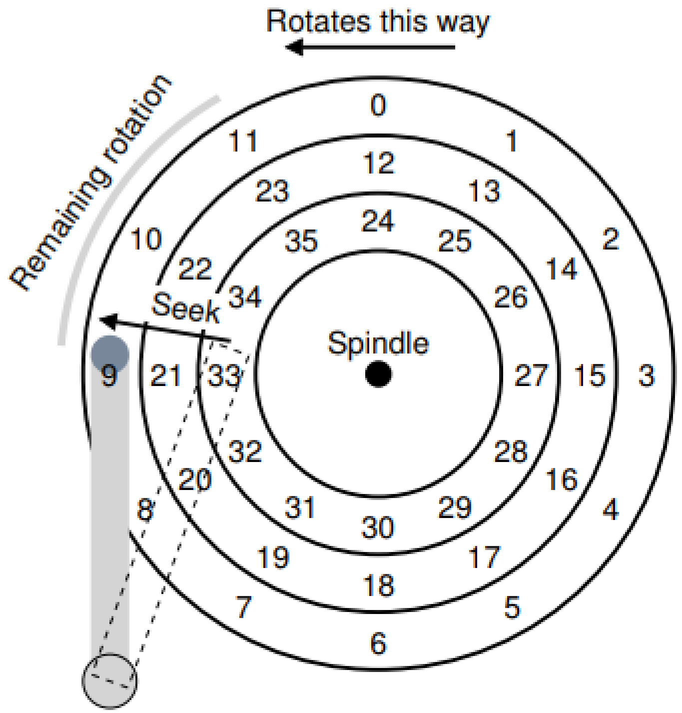


How to read any sector?

How to read sector 11?

- Move to desired track (seek time)
- Rotate till desired sector under head (rotational latency)
- Finally, transfer the data





I/O performance

- I/O time ($T_{I/O}$): $T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$

- The rate of I/O ($R_{I/O}$): $R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects Via	SCSI	SATA

Disk Drive Specs: SCSI Versus SATA

Performance

- **Random workload**: Issue 4KB read to random locations on the disk
- **Sequential workload**: Read 100MB consecutively from the disk

		Cheetah 15K.5	Barracuda
T_{seek}		4 ms	9 ms
$T_{rotation}$		2 ms	4.2 ms
Random	$T_{transfer}$	30 microsecs	38 microsecs
	$T_{I/O}$	6 ms	13.2 ms
	$R_{I/O}$	0.66 MB/s	0.31 MB/s
Sequential	$T_{transfer}$	800 ms	950 ms
	$T_{I/O}$	806 ms	963.2 ms
	$R_{I/O}$	125 MB/s	105 MB/s

Disk Scheduling

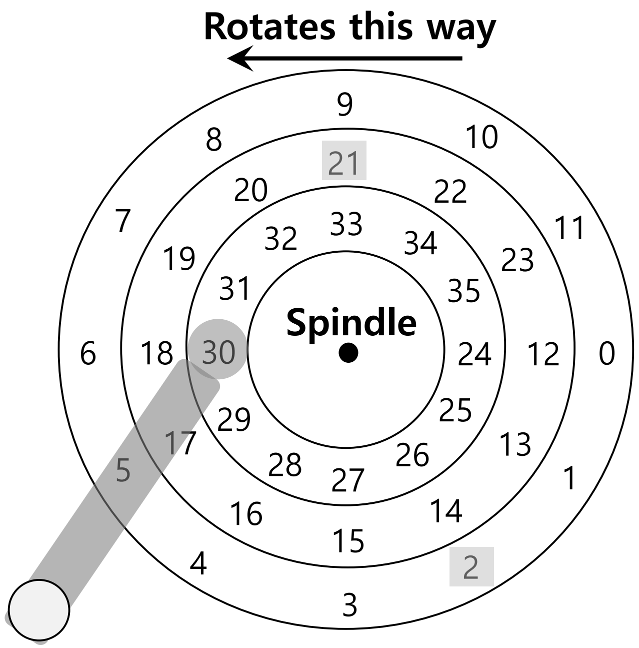
- **Disk Scheduler** decides <u>which I/O request</u> to schedule next
- Parameters to optimize:
  - Disk utilization: reduce seek and rotational delays
  - Response time: Queuing delay + seek + rotational + transfer

Disk Scheduling: FIFO

- Fair
- Does not optimize any parameters

Disk Scheduling

- **SSTF** (Shortest Seek Time First)
  - Order the queue of I/O request by track
  - Pick requests on the nearest track to complete first



SSTF: Scheduling Request 21 and 2

Issue the request to 21 → issue the request to 2

Problems with SSTF

- **Problem 1**: The drive geometry is not available to the host OS
  - Solution: OS can simply implement Nearest-block-first (NBF)
- **Problem 2**: Starvation
  - If there were a steady stream of request to the inner track, request to other tracks would then be ignored completely.
- High Throughput
- Average response time better than FIFO; but high variance

Elevator Algorithm: SCAN

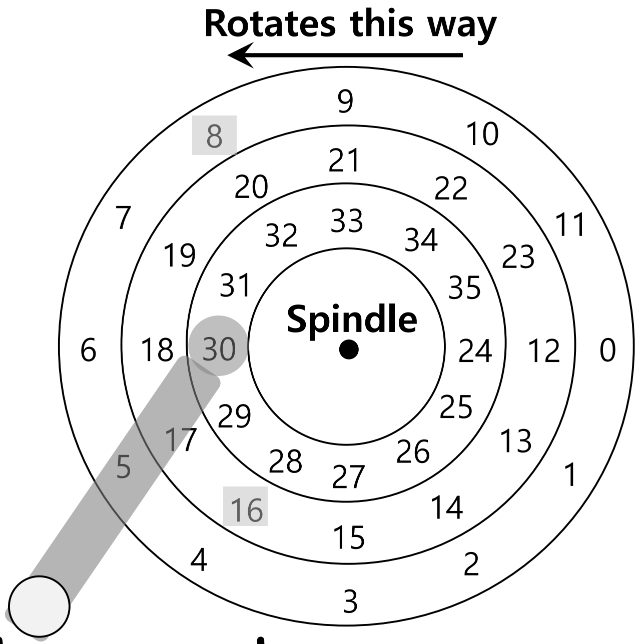
- Move across the disk from end to another servicing requests in the path. Reverse direction once reaching an end

- High throughput
- Average response time; but low variance

Elevator Algorithm: C-SCAN

- Instead of reversing after reaching the end, go back to the beginning

How to account for rotational delay?



- If rotation is faster than seek : request 16 → request 8
- If seek is faster than rotation : request 8 → request 16

On modern drives, both seek and rotation are roughly equivalent:
Thus, SPTF (Shortest Positioning Time First) is useful.

Disclaimer

- Some of the materials in this lecture slides are from the lecture slides by Prof. Arpaci, Prof. Youjip, and other educators. Thanks to all of them.