Disks

▶ **Scheduling the disk** - you have a bunch of blocks you want to write to the disk, say 0, 12, 10, 1, 6 - those are written to a buffer cache first, so you have an opportunity to schedule them in a different order:

- **FCFS** - not ideal, because it does not try to minimize seeks, not used often - bad performance
- **Shortest Seek Time First** - satisfy the block that is closest to your current position. also not used often - starves distant blocks, starvation.
- **Elevator Algorithm** - go in one direction, servicing blocks, till end of disk (all platter), then go in opposite direction till other end, repeat.
- **Scan Algorithm** - elevator algo that always goes from end to end of disk
- **Look Algorithm** - only go to last request cylinder in each direction (an optimized elevator algo)
- **C-Scan** - Scan with CIRCULAR LOGIC that goes in one direction only and then resets the head to the opposite end
- **C-Look** - Look with the same circular logic as C-Scan.

(variations / same)

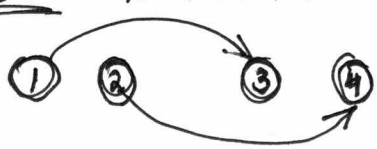
Theoretically, the seek distance for C-Scan and C-Look may not be better than Scan and Look, but in practice there may be mechanisms for fast head reset.

▶ **How do you get a lot of storage on a disk** - for the same density, the larger the disk, the more storage: ◯ > ○, but the bigger disk is more expensive, and seek time may be worse, and the yield rates are worse.

- Instead, we could have a bunch of small disks: ◯ = ○ + ○ + ○ + ○ + ○ This is called an **array of disks**.
- Problem with arrays is **MTBF** (mean time to failure): If a single disk failure rate is every 3 years, for the large disk, MTTF = 3 years If we have, say, 3 disks in an array, on average MTTF = 1 year for the array
- **RAID** = redundant array of inexpensive/independent disks (FUSION is SSD + HDD) is the solution to the increased MTTF of a disk array.

We use redundancy to solve the reliability problem by allowing to recover from (one, most often) disk failure.

▶ **RAID 1** - mirrors the disks with full redundancy, but needs double the number of disks, so expensive, we use 2N disks to store N disk fulls of data.



▶ **RAID 3** – uses a parity disk in lieu of complete replication, so we use N+1 disks instead of 2N (as in RAID1) to store N disk fulls of data.

①	②	③	④ parity disk
1	1	1	→ 1 odd
0	1	0	→ 1 odd
1	1	0	→ 0 even

← you can recover if you only use **any one** disk (the parity disk can be recovered also, trivially)

- You need to calculate parity for each change (only b/n the data and the parity disk, though) and potentially change it on the parity disk.
- The read performance is improved, since many disks can be read in parallel
- The write performance suffers, since the parity disk is a bottleneck
- Striping is used to distribute a file across disks and further increase performance – for RAID 3 it only helps with read performance

▶ **RAID 5** – spread the parity bits across, stripe them across the disks

①	②	③	④
0	1	1	0 parity
1 parity	1	1	1
0	1 parity	0	1

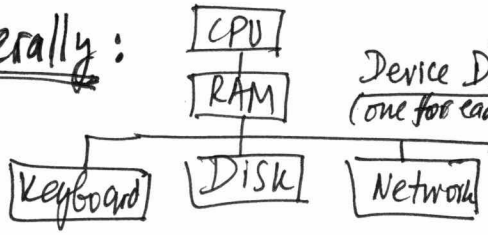
You still update a parity disk for each write, but that is now striped across many disks. Uses N+1 like RAID 3 but with better performance because you can access the disks in parallel.

▶ **RAID 0** – Files striped across all disks w/o parity or mirroring. Losing a disk is fatal but performance is improved (both read and write).

▶ **RAID 6** – RAID 5 + an extra parity disk that protects against 2 failures.

● Input/Output (I/O) Generally:

How does the OS support all the I/O devices:



OS – I/O device interface

Device Driver talks to OS thru (one for each device)

uses kernel (privileged) code

You could move the device drivers to separate device driver space, but that penalizes performance. ← potential problem, since bug in device driver will bring the whole OS down

▶ What happens from an I/O perspective?

EX:

Switch to app in user space to receive data ← OS puts it in a buffer for app

and back to client where the SSH app is probably blocked waiting for reply

Server network Card → Interrupt to device driver (interrupt serviced by the device driver)

I/O is complicated and SLOW!

System call (write)

software interrupt to run call

Get app data → identify device driver to handle data going out → call network card

network card sends data to server

Block waiting for I/O to complete

▶ **Interrupt Driven I/O** - shown in EX above, potentially expensive in CPU time.

- Some hardware has a way to buffer up the I/O and only send the interrupt every so often (when the buffer fills or every so many packets, etc.)

▶ **Polling I/O** - no interrupts, the thing that needs the information can periodically check. Downside is to determine how often to poll.

▶ **Blocking I/O** - call to I/O blocks till the I/O is performed (not necessarily to disk)

▶ **Non-Blocking I/O** - call to I/O returns immediately, more complicated programatically since you need to check when complete.

● **I/O Performance:**

- ▶ Reduce interrupts by coalescing (buffering I/O)
- ▶ Reduce context switching
- ▶ Reduce copying (ex: user space → OS → device driver → device) by passing a pointer and allowing access from different layers

---

● **UNIX View: 2 I/O devices** (2 different types of device drivers)

- ▶ **Block devices** - I/O is blocks of fixed size, randomly accessible, there is some cache in memory for blocks (ex. disk, CDROM), cache managed by OS w/ disk scheduling
- ▶ **character devices** - unit of transfer is not fixed size, usually sequential access (ex. keyboard, tape)
- ▶ **Raw devices*** - are used sometimes when an application wants more control (ex. to disable caching from the OS). * they are not exactly a 3rd type of device, but a narration of a block device or character device.

■