Lecture 16: OS Design and RAID

CS343 – Operating Systems

Branden Ghena – Fall 2020

Some slides borrowed from:
Stephen Tarzia (Northwestern)

Northwestern

Today’s Goals

- Discuss principles guiding OS design.

- Describe several classes of OS kernel.

- Also:
- Explore topic of RAID – redundancy in disks.

Outline

- **OS Design Principles**
- Kernel Designs
- RAID

Disclaimer

- This is the most abstract part of the class

- These ideas can take a long time to sink in
  - Best to introduce them and repeat them

Interface design

- Three guiding principles per Tanenbaum

  1. Simplicity
  2. Completeness
  3. Efficiency

1. Simplicity

- Simple interfaces are easier to understand and use

"Perfection is reached not when there is no longer anything to add,
but when there is no longer anything to take away."
- Antoine de St. Exupéry (French writer and aviator)

- Fork() is a great example

Simplicity means avoiding premature optimization

- Do not let perfect stand in the way of good
  - It’s more important to have something working than nothing at all

- First step is to make it work “good enough”
  - Optimization can come later based on usage
  - Otherwise you might be failing Amdahl’s Law

Simplicity reduces bugs

- Interfaces that match expectations end up being used correctly

- Features are the source of mistakes
    - Code that doesn’t exist has no bugs

2. Completeness

- Interfaces should make it possible to do everything users need to
- Simple things should be simple, difficult things should be possible
- For system design, this usually means moving extra functionality into userspace libraries
  - Managing heap memory is a good example

OSes have a long lifetime

- OSes in the real world have a tendency towards very long lifetimes
  - Planning for the future is difficult
  - Maintaining support for the past is crippling

- POSIX standard was created in 1988
  - 3 years before the first webcam
  - 10 years before WiFi

- eSata port example

3. Efficiency

- Implementations should be efficient
  - Given simplicity and completeness goals first

- Efficiency should meet user’s expectations as well
  - Which is faster: seek or read?

  - If it’s not seek, developers are going to write bad code
    - Also, seek should be cut from the interface (simplicity)

Other OS principles

- Two others I want to pull out of the chapter
  - See textbook for many more lessons and examples

1. Separate policy and mechanism

2. Project management is hard

Separation of policy and mechanism

- Exemplified in schedulers and virtual memory
  - Mechanism for switching threads
  - Policy for when to do so

  - Could swap out either without changing the other

- Be explicit about which is which

Project management is hard

- Mythical Man Month
  - Large projects involve an enormous amount of planning and testing
  - They are NOT highly parallelizable

- The person-month idea is that time and number of developers are exchangeable
  - In the general case, this is false

- **Brook's Law:** "Adding manpower to a late software project makes it later."

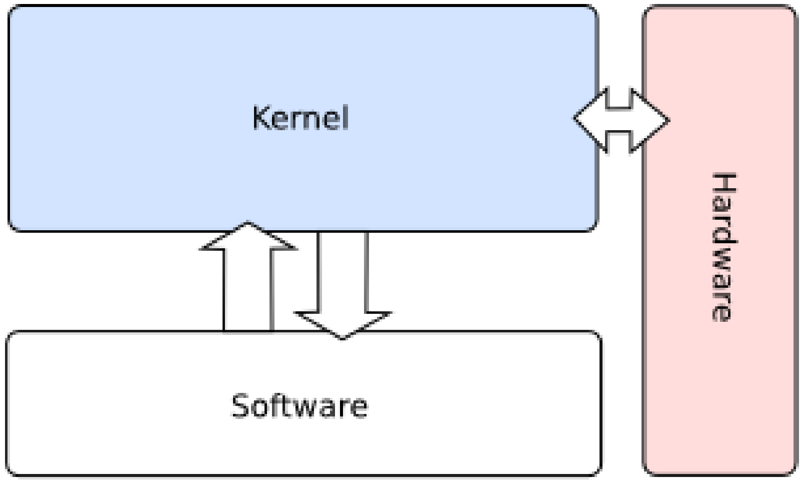
Outline

- OS Design Principles
- **Kernel Designs**
- RAID

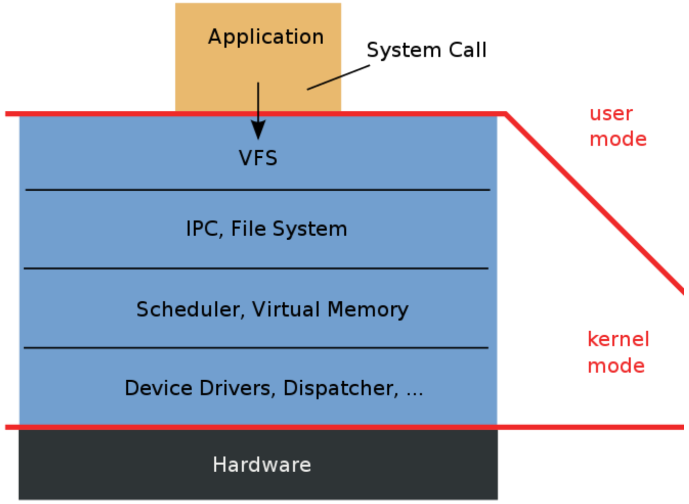
Monolithic kernel

- This is the model we have been learning

- All OS services occur within the kernel
- Applications request service from the kernel
- Hardware can only be accessed by the kernel

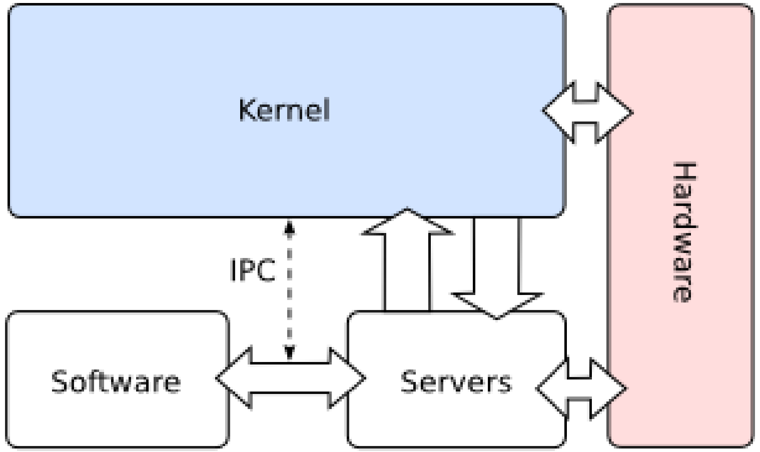


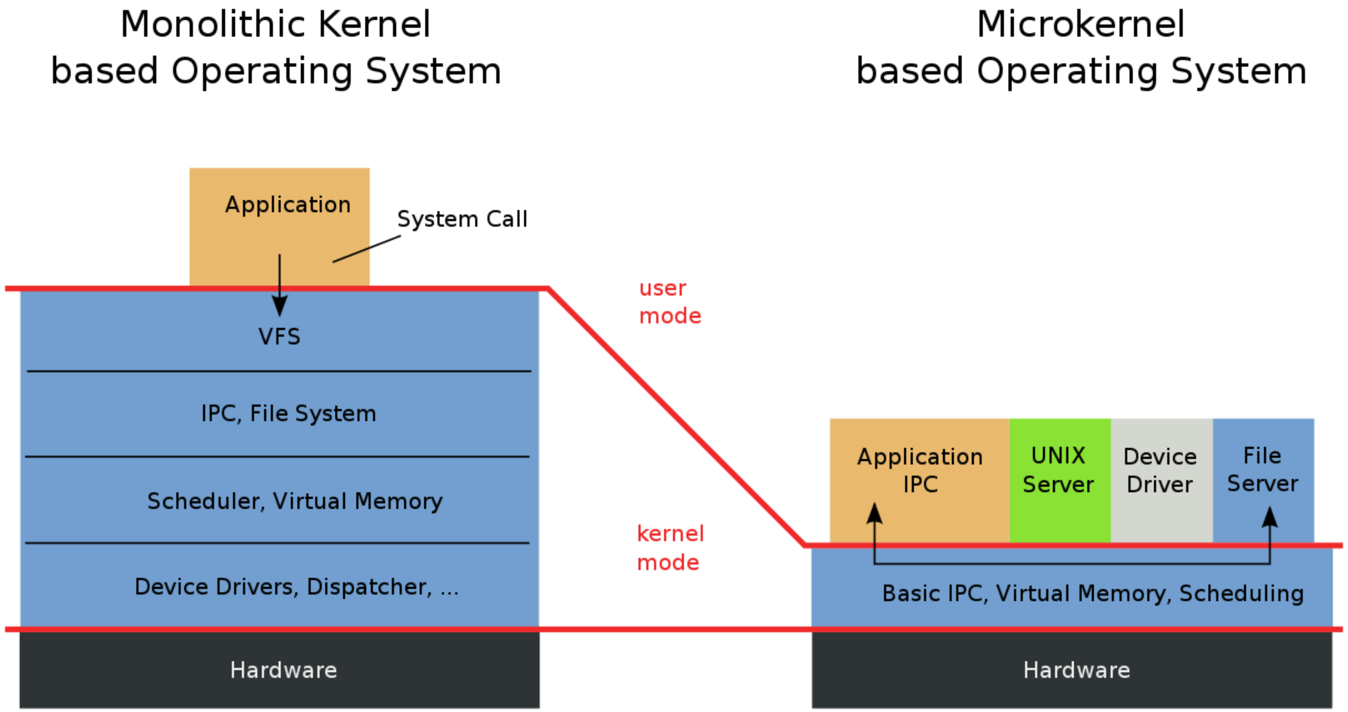
Monolithic Kernel
based Operating System



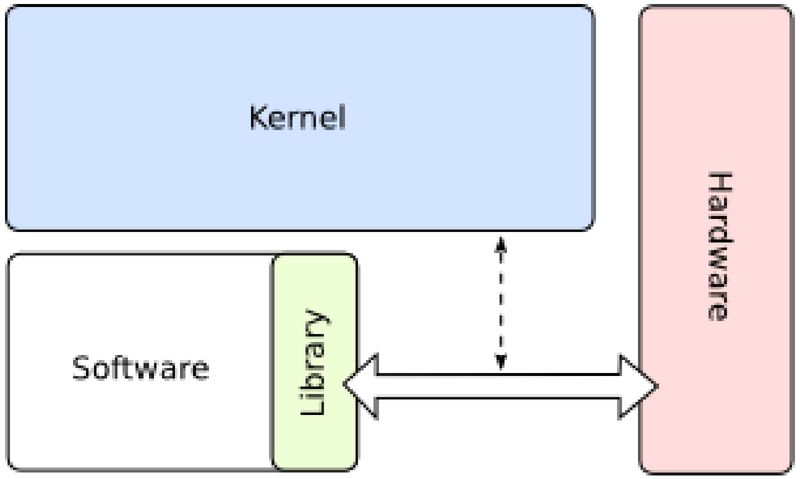
Microkernel

- Most services are userspace programs
- OS kernel implements minimum features to support them
    - Requests for services are often Inter-Process Communication (IPC)





Exokernel



- Goal: separate security and abstraction
  - OS should provide security only
  - Everything else goes in applications
- If an application is allowed to access a certain region of the disk
  - Monolithic: constrain it to a particular filesystem included in the kernel
  - Exokernel: give it raw access to those disk blocks and it can decide
    - Application libraries can implement filesystem stuff
- Upside: applications can be more efficient
- A downsides: surrenders "look and feel" of the system

Hybrid kernels

- Most real operating systems are not any of these extremes
  - But exhibit qualities for each as desirable
- Often some lowest level drivers are in the kernel
  - But higher level stuff are userspace services
- Example: the OS needs to support USB
  - But printers can be run as services
- Example: heap region is given to programs to manage
  - Libraries like malloc can manage it in userspace

Outline

- OS Design Principles
- Kernel Designs
- **RAID**

Failure rates for disks are a serious problem

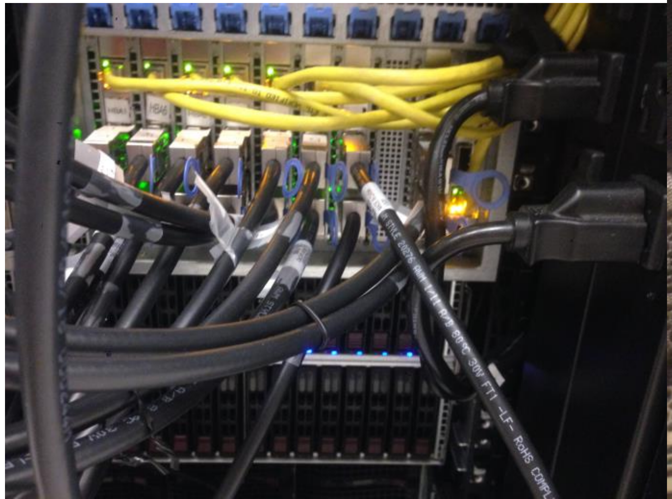
- Problem: disks fail
  - HDDs have physical actuators that wear out
  - SSDs have limited numbers of writes

- Big problem: servers have many disks
  - Assume rate of failure per year of disk is 1%
    - And failures aren't correlated
  - And a server has 264 disks
  - What are the odds that a disk will fail this year?

  - $1 - (1 - 0.1)^{264}$ = 93% odds that at least one disk will fail

Database server at Northwestern

- 264 fast (10k RPM) magnetic disks (for production)
- 56 slow (7200 RPM) magnetic disks (for backup)
- ~150 TB storage capacity
- Comprised of 6 physical chassis (boxes) in one big cabinet, about the size of a coat closet.



Redundant Array of Independent Disks (RAID)

- Observation in 1988 (Patterson, Gibson, Katz)
  - Servers could use high-quality mainframe disk drives

    OR
  - Servers could use several redundant consumer disk drives

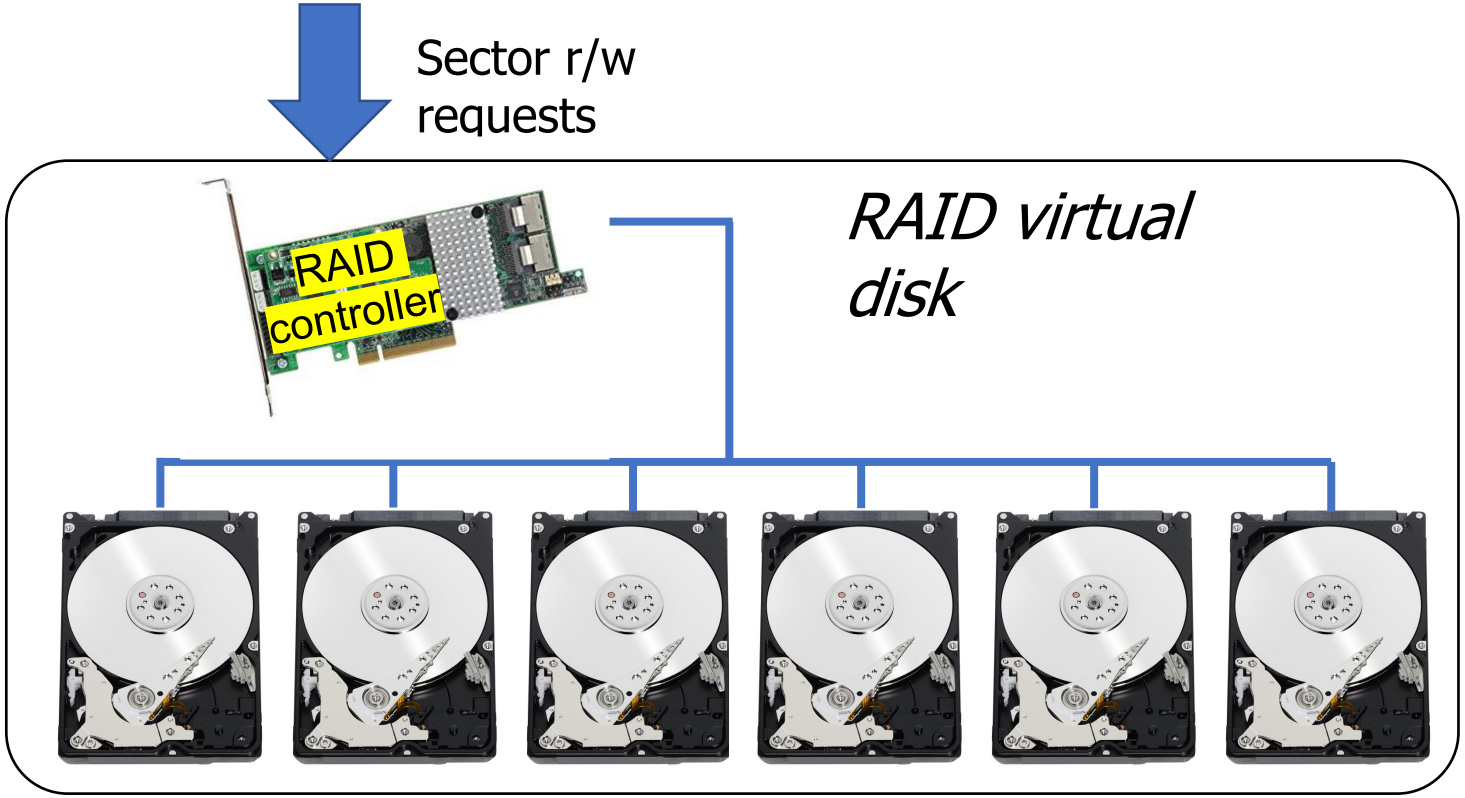
- Furthermore array of disks improves multiple things at once
  - Reduce impact of a ***failure*** by storing data redundantly on multiple disks.
  - Increase ***capacity*** by making multiple disks available to store data.
  - Increase ***throughput*** by accessing data in *parallel* on multiple disks.

Basic idea of RAID

- Combine many disks to create one *superior* virtual disk.
- The RAID array provides the same interface as a single disk.

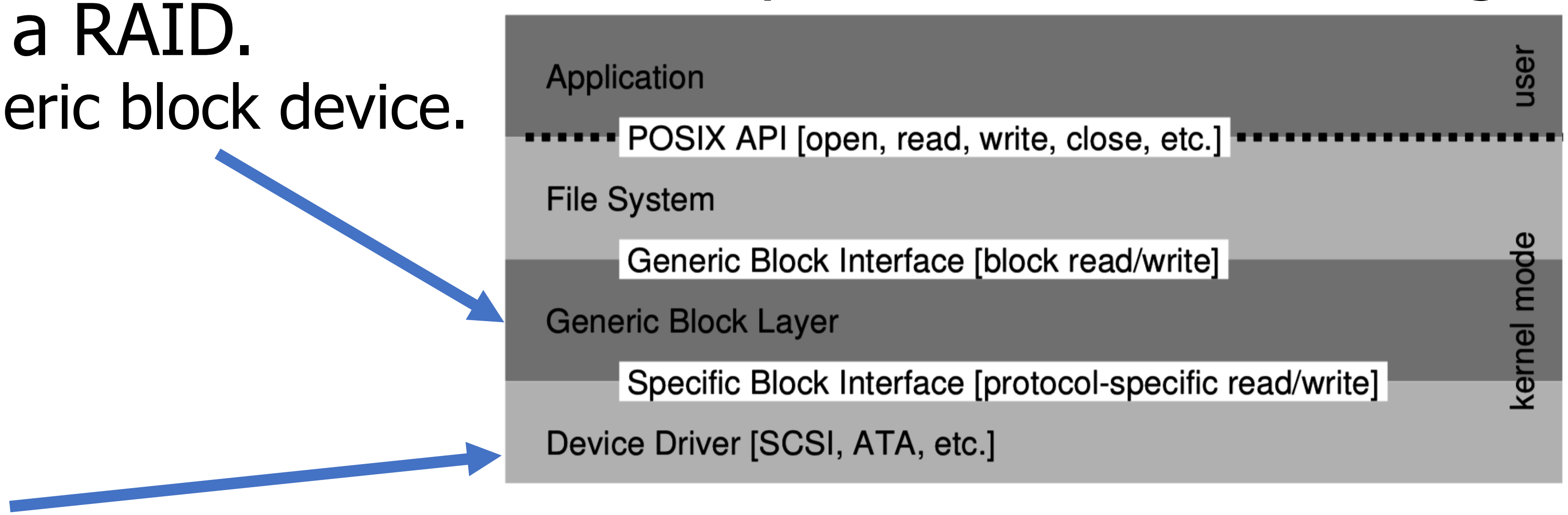
OS thinks it's dealing with this:

But it's just an illusion. The reality is:



How does RAID fit into the OS?

- RAID can be implemented in software or hardware
- ***Software RAID*** means that the OS is responsible for assembling multiple disks into a RAID.
  - Implements a generic block device.

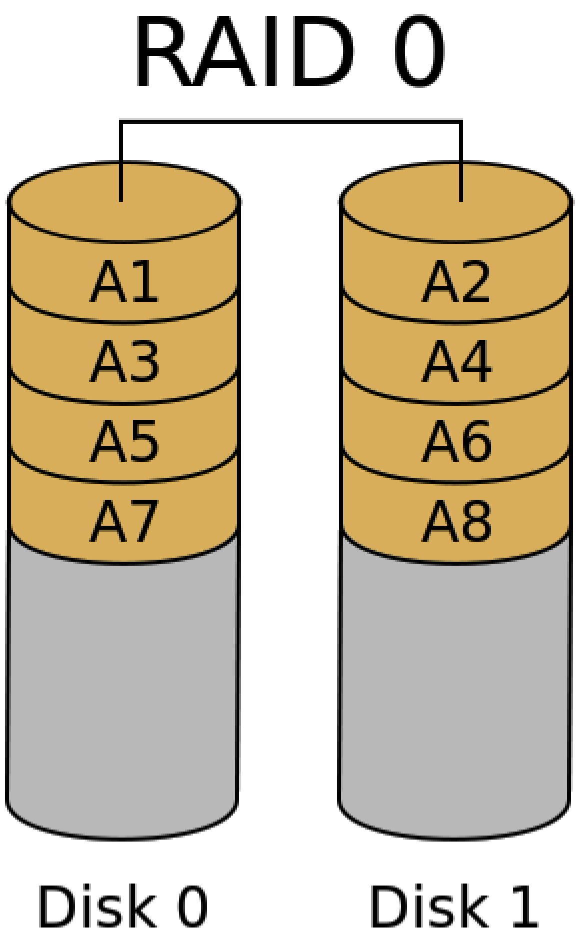


- ***Hardware RAID*** requires a specialized controller card that coordinates the multiple disks, presenting interface of one disk.
  - OS just needs a driver for the RAID controller, like any other disk controller.

RAID levels

- RAID 0 – ***Striping***:
  - Distribute data across 2 disks for twice the peak throughput.
- RAID 1 – ***Mirroring***:
  - Copy data onto 2 disks to tolerate failure of one.
- RAID 4/5/6 – ***Parity:***
  - Keep parity bits around for each block to check for errors and rebuild.
  - Typically involves 3+ disks.

RAID 0 – Striping *(for throughput and capacity)*

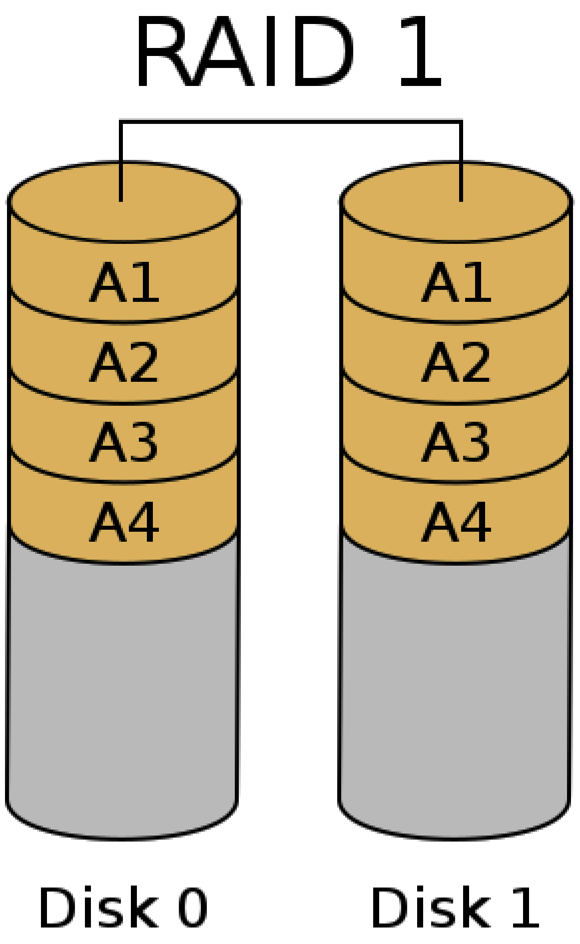


- Divide the logical disk into chunks
  (A1, A2, A3 …) 1 or more blocks in size
- Distribute the chunks regularly over two or more (*N*) physical disks.
- (+) Throughput for both random and sequential access scales with N.

  $T_{RAID0} = N * T_{disk}$
- (+) Capacity also scales by N.
- (+) Cost per byte is identical
- (–) But Mean Time To Failure is worse because failure of a single disk is catastrophic:

  $MTTF_{RAID0} = MTTF_{disk}/N$

RAID 1 – Mirroring *(for fault tolerance)*

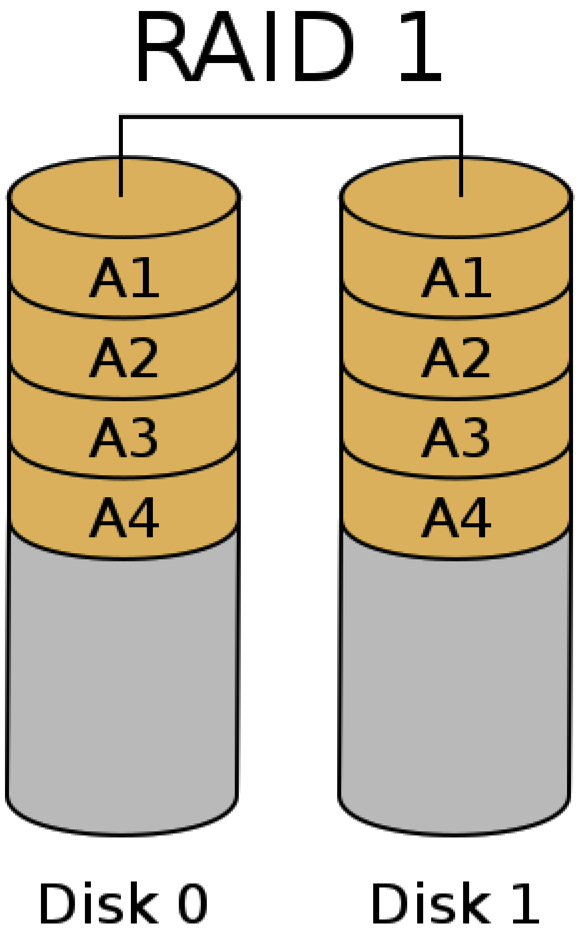


- Duplicate each chunk on each of N physical disks.

- (+) It is impossible to lose data unless all disks fail simultaneously.
  - i.e., failure window is reduced to the time it takes to replace a broken disk.
- (–) Write throughput is not improved
- (–) Capacity is the same as a single disk
- (–) Cost per byte is greater

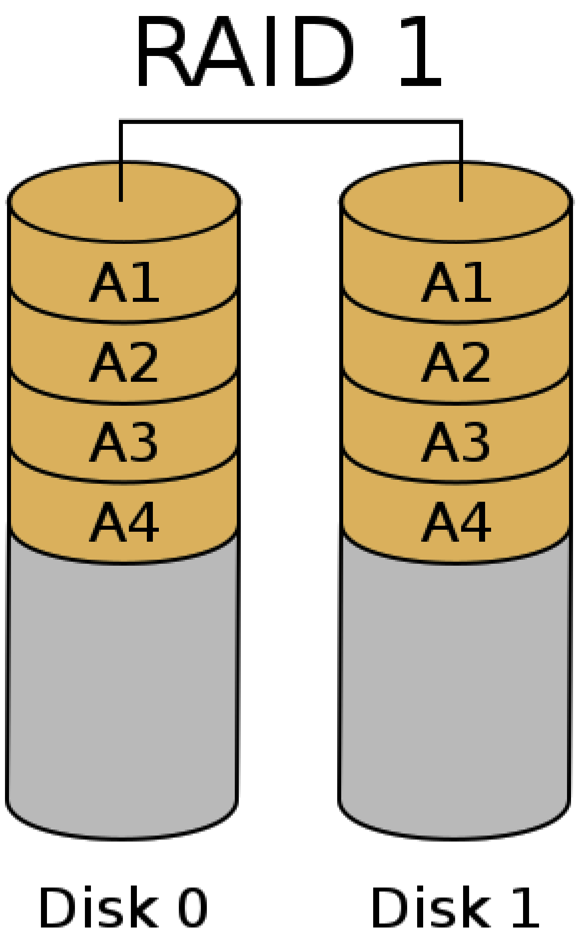
$$\$_{RAID1} = N * \$_{disk}$$

Check your understanding – RAID 1



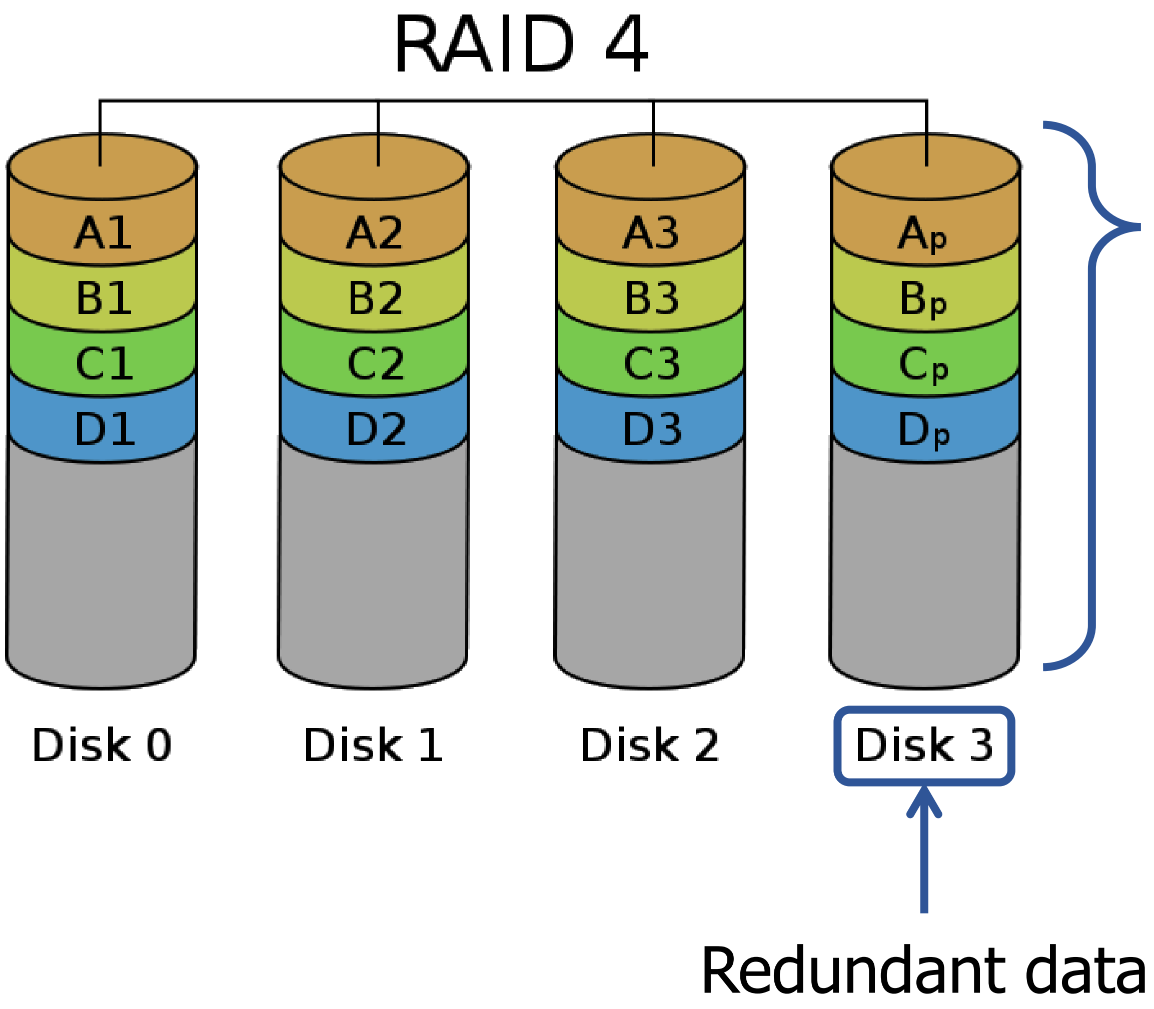
- (–) Write throughput is not improved
- Is write throughput reduced in RAID 1? Or is it the same as a single disk?
- What about read throughput?

Check your understanding – RAID 1



- (–) Write throughput is not improved
- Is write throughput reduced in RAID 1? Or is it the same as a single disk?
  - Same as a single disk
  - Write can go to both disks in parallel
- What about read throughput?
  - Better than a single disk
  - Can read two different blocks at once!

RAID 4 – Parity *(for fault tolerance, capacity & throughput)*



- Distribute the chunks across the first (N-1) disks.
- On the N^{th} disk, store a corresponding ***parity*** chunk.
  - Parity block is redundant data about a set of chunk (a ***stripe***)
- Can tolerate loss of any one disk
- Parity disk becomes bottleneck for writes limiting throughput

How does parity work?

- ***Even parity*** – add a 0 or 1 such that the total number of 1's is even.
  - There also exists odd parity which makes the total number of 1's odd
- Examples (Even Parity):
  - 0b0000_0000 – zero ones -> parity bit = 0
  - 0b1111_1111 – eight ones -> parity bit = 0
  - 0b0110_1101 – five ones -> parity bit = 1
- If a single bit is lost, the parity bit allows us to infer the value of the lost bit

Check your understanding – Parity Recovery

- What are the values of the missing bits?

- [0, 0, 1, 0, **?**, 0, 1, 1] – Even Parity: 1

- [0, **?**, 1, 1, 1, 0, 0, 0] – Even Parity: 0

Check your understanding – Parity Recovery

- What are the values of the missing bits?

- [0, 0, 1, 0, **?**, 0, 1, 1] – Even Parity: 1
  - Value must be a 0
  - Because parity plus ones is already even

- [0, **?**, 1, 1, 1, 0, 0, 0] – Even Parity: 0
  - Value must be a 1
  - Because parity plus ones is not currently even

Parity can only fix a single error

- What if two bits are missing?

- [**?**, 0, 1, 0, **?**, 0, 1, 1] – Even Parity: 1
  - Could both be zeros
  - Could both be ones
  - Impossible to tell which

- More advanced “error correcting codes” are possible to detect/fix two or more errors
  - Hamming Code (single error correcting, double error detecting)

Parity chunk in RAID

Disk 0	Disk 1	Disk 2	**Disk 3 (parity)**
0001 0010 1100 1100	0000 1111 0000 1111	1101 1111 0011 0001	**1100 0010 1111 0010**
1111 1111 1111 1111	0001 0001 0001 0001	1101 1001 0110 0110	**0011 0111 1000 1000**
0000 0000 0000 0000	1101 1011 0011 0011	1111 0011 0011 1000	**0010 1000 0000 1011**

Useful storage capacity (Disks 0–2)

Redundancy overhead (Disk 3)

- Parity is computed bit-wise across corresponding chunks.
- Chunks are one or more blocks (multiple of 4 kB) in size
- Writing a small file will involve one disk *plus the parity disk.*
  - (parity disk can become a bottleneck)
- Writing a large file will involve all the disks.

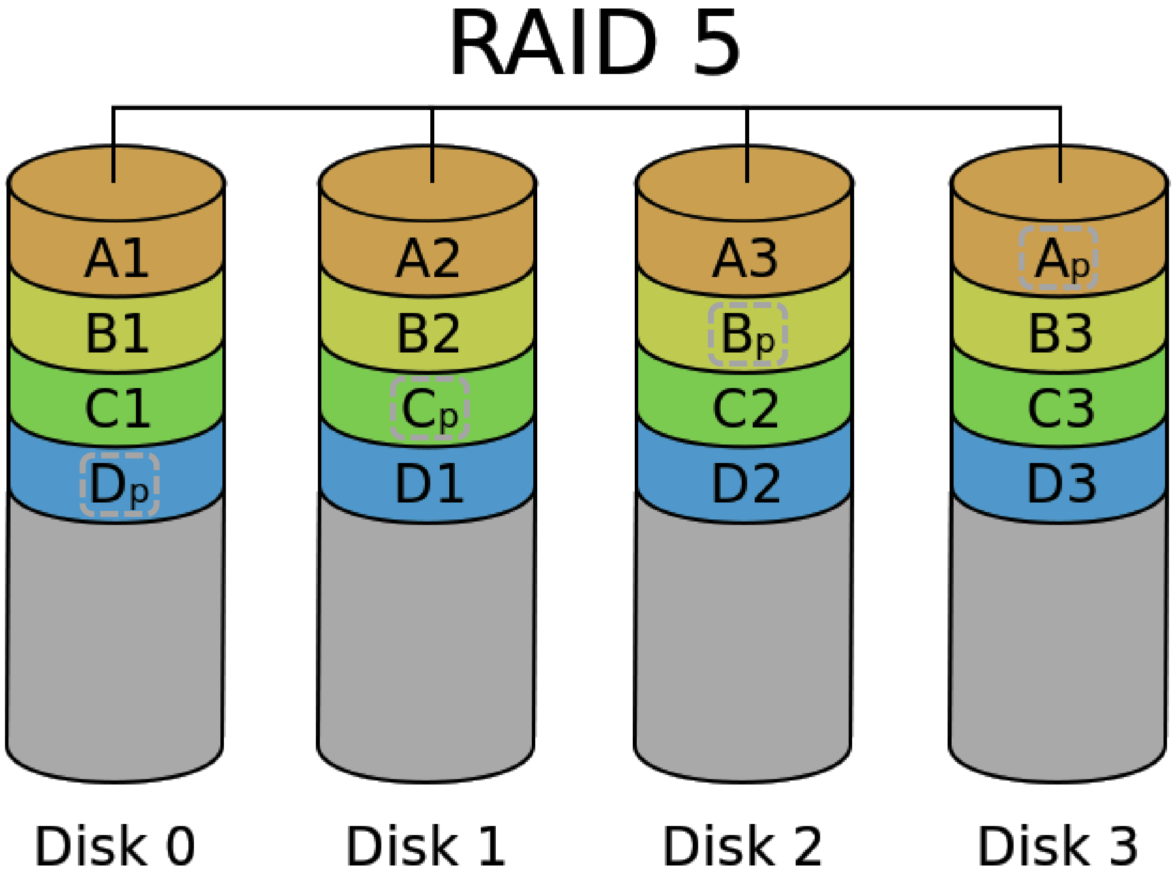
Rebuilding an array after failure

Disk 0	Disk 1	Disk 2	**Disk 3 (parity)**
0001 0010 1100 1100		1101 1111 0011 0001	**1100 0010 1111 0010**
1111 1111 1111 1111		1101 1001 0110 0110	**0011 0111 1000 1000**
0000 0000 0000 0000		1111 0011 0011 1000	**0010 1000 0000 1011**

Disk failed!

- If a disk fails, then we remove it and replace it with a working disk.
- Then scan through the entire array to compute and write missing data.
    - This is called "rebuilding" the array
    - We cannot tolerate another disk failure until rebuild completes.
    - Reads/writes can continue while array is rebuilding!

RAID 5 – Distributed Parity *(the winner in practice)*



- Distribute parity chunks across the disks, to avoid a small-write bottleneck
- (+) Failure of one disk is OK
- (+) Throughput is good

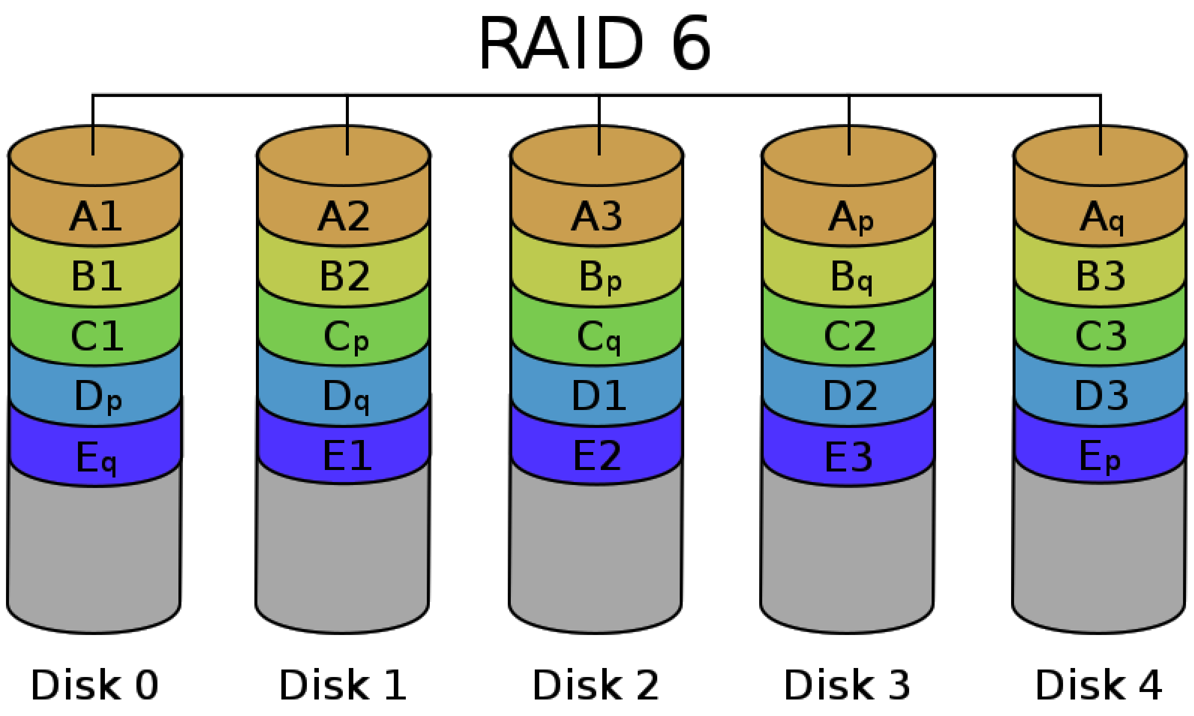
$$T_{RAID5} = (N-1) * T_{disk}$$

- (+) Cost per byte is good

$$\$_{RAID1} = N/(N-1) * \$_{disk}$$

- (–) High overhead for small N
- (–) Failure risk is high for large N
- N is typically 3 to 8

RAID 6 – Double Parity *(for large arrays)*



- Add another disk and keep two parity chunks per stripe
  - 2nd parity is computed differently
- (+) Failure of ***two*** disks is OK
- (~) Throughput is less:

$$T_{RAID5} = (N-2) * T_{disk}$$

- (~) Cost per byte is higher:

$$\$_{RAID1} = N/(N-2) * \$_{disk}$$

- Makes sense for larger N (>8)

Outline

- OS Design Principles
- Kernel Designs
- RAID