CS61C: Great Ideas in Computer Architecture (Machine Structure)

Instructors: Rosalie Fang, Charles Hong, Jero Wang

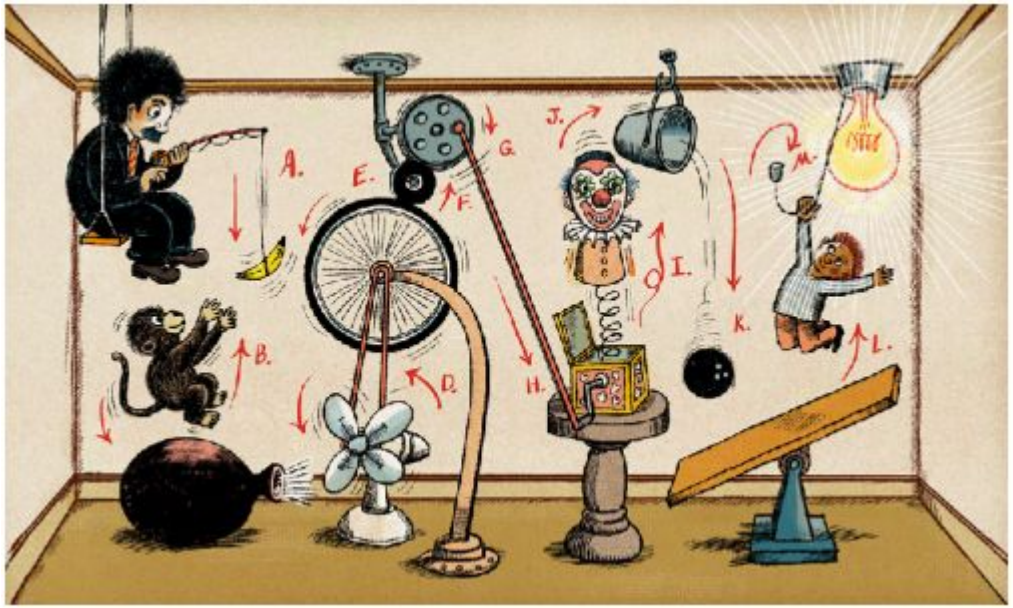
Lecture feedback:
https://tinyurl.com/fyr-feedback

Come sit at the front 🥺🥺🥺

Dependability

Introduction: Rube Goldberg Machines

- The big problem with Rube Goldberg Machines is that every step needs to go exactly right for the entire machine to work
- Each step adds some (ideally small) chance of failure
- 20 steps at 95% success rate gives ~37% chance of success overall
- 100 steps at 95% success rate gives ~1/168 chance of success



Dependability

- Computers are effectively electrical Rube Goldberg machines
- Modern systems are fairly reliable, but errors still happen occasionally
  - Network systems: ~10^{-9} chance of bit flip
  - DRAM: Cosmic rays at Earth sea level give a ~10^{-13} chance of a specific bit flipping per second (2009).
  - Error rates are higher for more modern systems (due to there being lower difference between a 1 bit and a 0 bit), and generally higher in space/high altitude environments.
  - Larger failures could damage the entire disk, with probability ~1% per year.
- Seems like a small amount, but we often deal with GiB at a time.
  - Downloading a 2 hour movie would have ~2 GiB * 10^{-9} = 10-20 bit errors in the download
- Conclusion: Failure is not an option. It is mandatory.
- We need a way to mitigate the effects of failure.

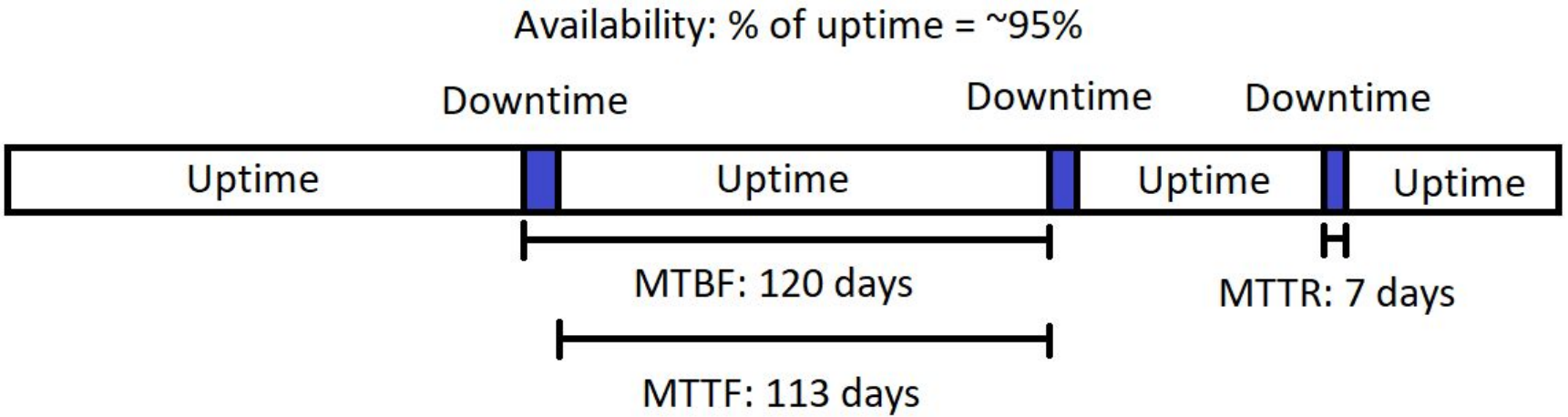
Availability: Definitions

- Motivating example: You have some tool/service. Every now and then it fails, so it's unavailable until you fix it.
- Mean Time to Failure (MTTF): Average time it takes for an entirely new system to fail.
- Mean Time to Repair (MTTR): Average time it takes to fix the tool once it breaks.
- Mean Time Between Failures (MTBF): Average time to complete one "cycle" of working→broken→working.
  - Equal to MTTF+MTTR
- Availability: Percentage of the time that the service is available
  - Equal to MTTF/MTBF

Availability Example

- My laptop broke three times last year (right after the warranty expired...)
- Each failure prevented me from using the laptop for ~1 week (needed to find parts online, have them shipped, borrow a repair kit, etc.)
- MTBF: 1 year / 3 (failures) = 1/3 year or **~120 days**
- MTTR: **7 days**
- MTTF: 120 days between failures, but the laptop wasn't in use for 7 of those days. In terms of the actual time I was using the laptop, it failed **~113 days** after it got fixed.
- Availability: 113 days / 120 days = 113/120 or about **95%**

Availability Example



Availability

- Common shorthand is "9s of reliability/availability"
- 1 nine of availability → 90% available
- 2 nines of availability → 99% available
- 3 nines → 99.9%
- 5 nines of availability →99.999% availability = 5 minutes of repair/year
  - Extremely expensive to maintain
- Another common metric is Annualized Failure Rate: average number of failures per year.
  - Often around 1-10%, for individual drives, depending on the age of the drive.

Availability

- What happens if we have multiple components?
- If all our components are critical, then we expect the overall system to fail *faster* than any individual component
  - If your battery breaks every two years and your CPU breaks every two years, one of them will break (on average) every year.
- On the other hand, if we have two copies of the same system and only need one of them to work, it's extremely unlikely to have both systems fail simultaneously
  - If two different batteries have an availability of 99%, at least one battery will be working 99.99% of the time.
    - On the other hand, you'll be at half efficiency 2% of the time.
- Major principle in engineering: Add redundancy to systems to reduce the chance of failure
  - Eliminate all single points of failure.

Great Idea #6: Dependability Through Redundancy

- This ends up showing up a lot in various engineering components
  - From a hardware perspective, laptops are designed to "fail slowly" by shunting work to any remaining working components
    - This is why laptops tend to get slower as they age
- Today, we'll discuss two specific cases of data redundancy:
- Error Correcting Codes
  - Recover correct binary data even if some bits were corrupted
- RAID
  - Connect multiple hard drives together so that even if one fails, all data is recoverable

Error Correcting Codes

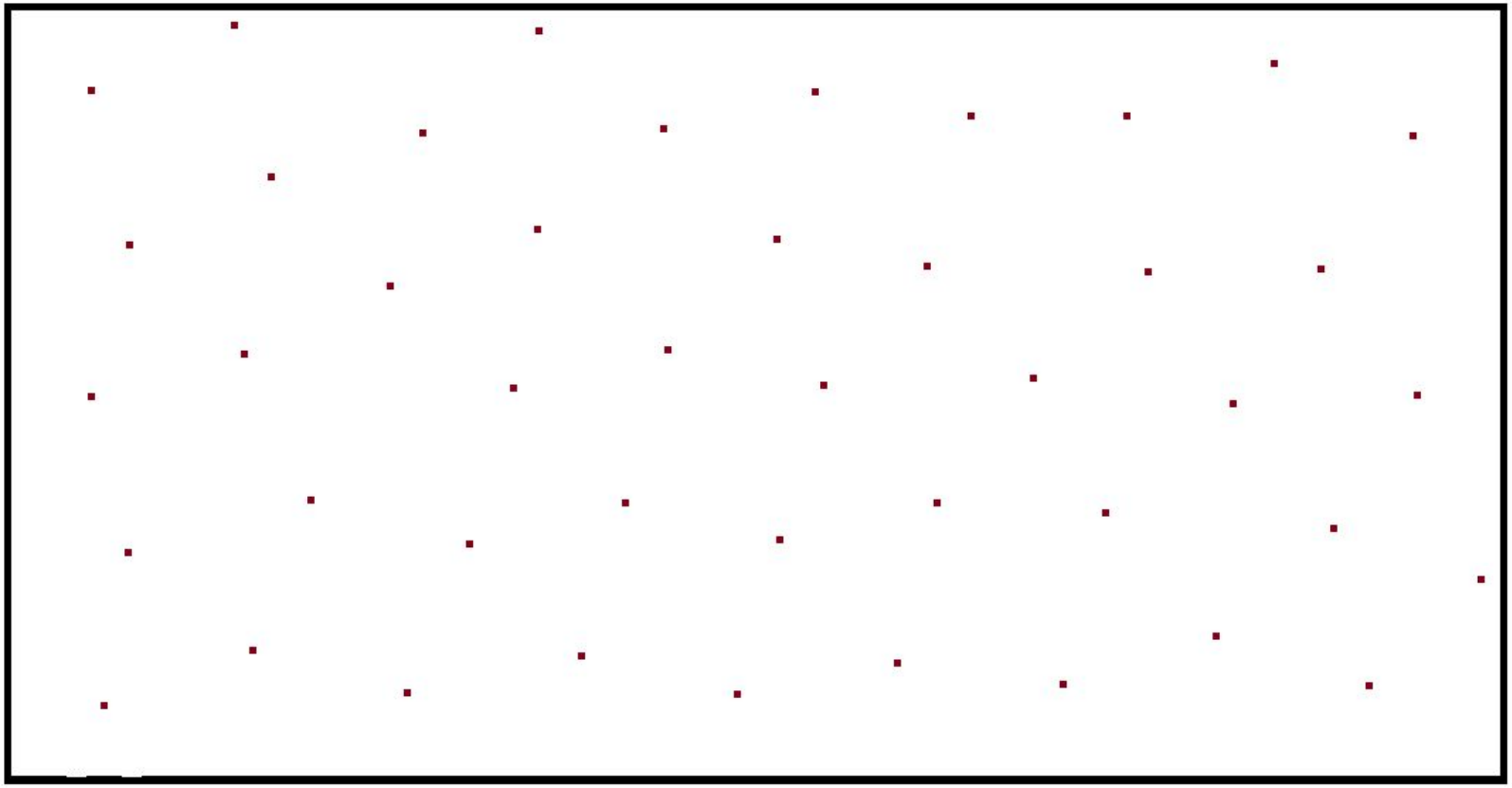
Error Correcting Codes

- When storing data, there's a very low (but still possible) chance that some of our bits get flipped by cosmic rays. https://www.exploratorium.edu/exhibits/cloud-chamber
- In order to detect these, we need to store more bits than the initial data we received.
- Two major goals:
  - Detect if an error occurred
    - Even if we can't fix the error, we can try again/inform the user that something wrong happened.
  - Correct an error that happened
    - Generally harder than just detecting an error, but lets you continue running the program even if an error happens.
- General assumption is that there's either a fixed number of errors (since it's rare to have multiple errors at once), or a fixed percentage of errors.
  - Ex. A Hamming Code might be able to detect 2-bit errors, and correct 1-bit errors.
  - Ex. QR codes use a Reed-Solomon code (out of scope) that can correct up to 7-30% errors, depending on the correction rate requested.

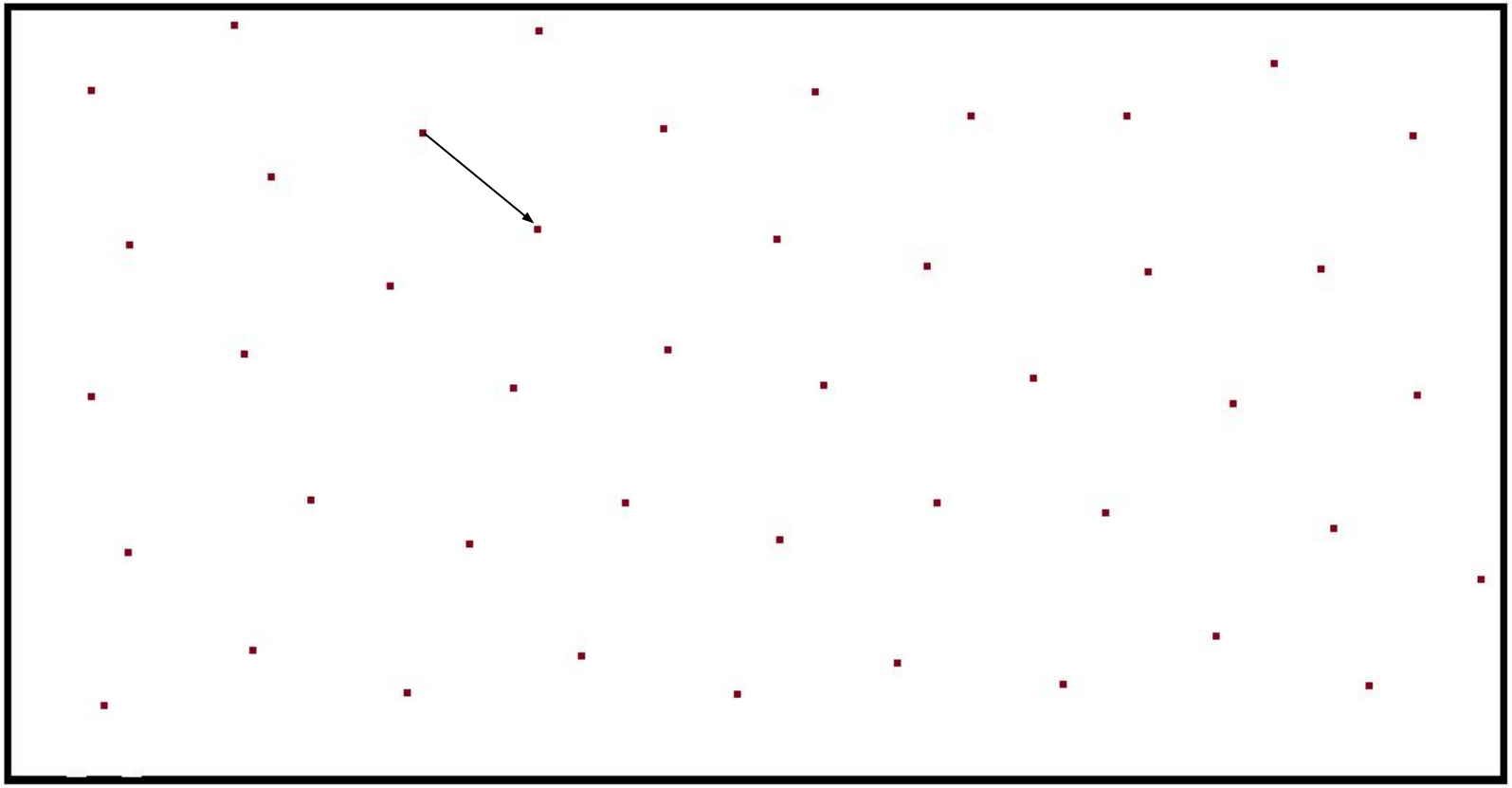
Error Correcting Codes

- Example: We start with 4 data bits (16 possible data values)
- We use some function f to convert from 4 data bits to a 7-bit codeword
  - The 16 possible outputs of f are considered the "valid" codewords, because they signify a lack of corruption.
- Someone takes that codeword and flips up to 1 bit.
  - Defensive programming: Assume that the bit flip is malicious and actively trying to make your system fail.
- If we are guaranteed that a corruption doesn't get us to another valid codeword, we can detect 1-bit errors
- We receive those 7 bits (with potentially one corruption) and use some function g to convert back to 4 data bits.
- If we are guaranteed to get back our old data, we can correct 1-bit errors

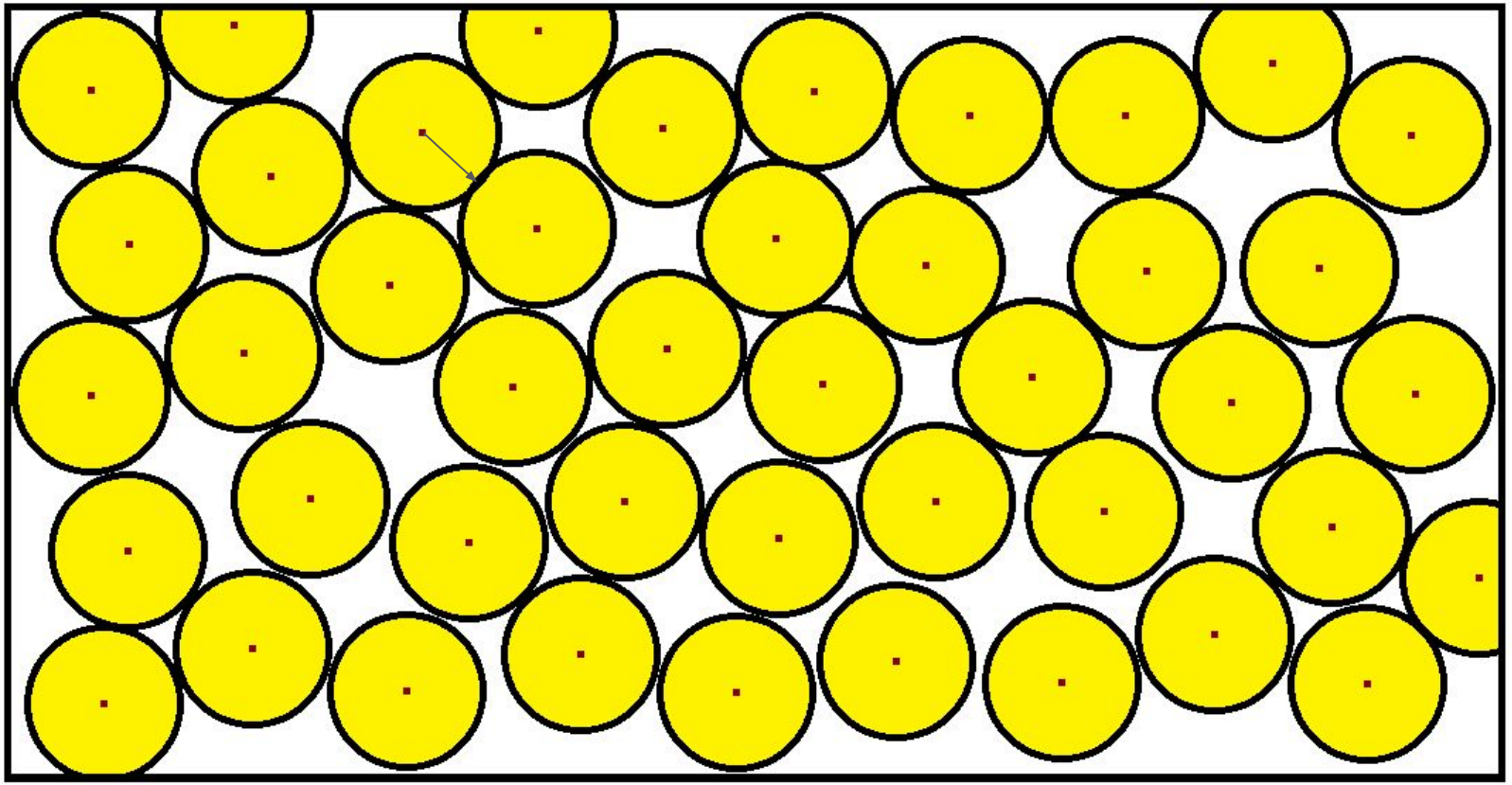
Error Correcting Codes: Graphic



Error Correcting Codes: Maximum to Detect Error

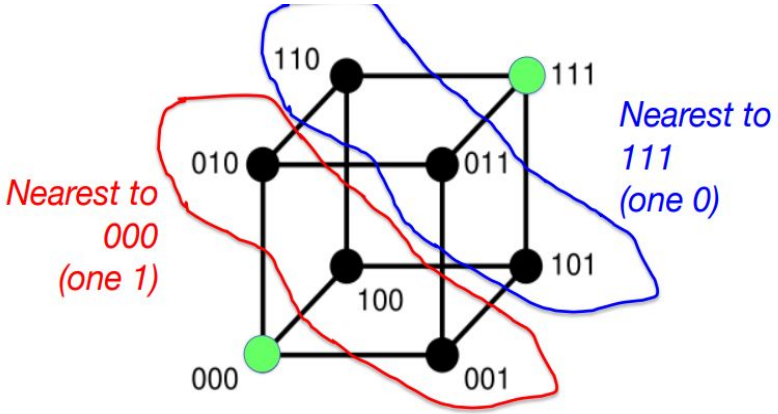


Error Correcting Codes: Maximum to Correct Error



Error Correcting Codes

- Generally, you can detect about twice as many errors as you can correct.
- As long as your bit space is large enough, you can create an ECC. The goal is to try to find the smallest possible "box" (bit count) that lets us fulfill a given detection/correction requirement.
- Reality is more discrete, and n-dimensional, but otherwise the same as the 2-D version (the same concepts work as long as we use a metric space)
- Hamming Distance: Number of 1-bit corruptions needed to move from one bitstring to another
- Limited by the two closest codewords



Parity Bit

- Attempt 1: Store one extra bit so that the parity (number of 1 bits) of the string is even
  - Ex. If we want to store data 0b1001 1000, our parity bit will be 1, so we store 0b 1 0011 0001
  - Ex. If we want to store data 0b1001 1001, our parity bit will be 0, so we store 0b 1 0011 0010
- To convert back, we can just cut the parity bit.
- Can we detect 1-bit errors?
  - Yes. If one bit gets corrupted, our parity will become odd instead
- Can we detect 2-bit errors/correct 1-bit errors?
  - No: The two example codewords are Hamming distance 2 apart, so if we had two corruptions, we could go from one codeword to another.
- Is this optimal to detect 1-bit errors?
  - Yes. We can't do better than "1 extra bit", since "0 extra bits" would have no invalid codewords.

Hamming Code

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	✓		✓		✓		✓		✓		✓		✓		✓		✓		✓		
	p2		✓	✓			✓	✓			✓	✓			✓	✓			✓	✓		
	p4				✓	✓	✓	✓					✓	✓	✓	✓					✓	
	p8								✓	✓	✓	✓	✓	✓	✓	✓						
	p16																✓	✓	✓	✓	✓	

- Main idea of Hamming Codes: Include multiple parity bits (that look at a subset of all bits), so we can pinpoint which bit got corrupted
- In the above, d1, d2, … refer to the first, second, … data bits, and p1, p2, … are the parity bits.
- Parity bits are chosen such that the bits checked on each row have even parity
- Note that no row has multiple parity bits checked, so each parity bit is uniquely determined

Hamming Code: Encoding

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15
Parity bit coverage	p1	✓		✓		✓		✓		✓		✓		✓		✓		✓		✓	
	p2		✓	✓			✓	✓			✓	✓			✓	✓			✓	✓	
	p4				✓	✓	✓	✓					✓	✓	✓	✓					✓
	p8								✓	✓	✓	✓	✓	✓	✓	✓					
	p16																✓	✓	✓	✓	✓

- Example: Encode 0b 011 0111 0110 as a Hamming code
- __0_110_1110110
- _ _ 0 _ 1 1 0 _ 1 1 1 0 1 1 0 -> Even parity, so p1=0
- 0 _ 0 _ 1 1 0 _ 1 1 1 0 1 1 0 -> Even parity, so p2=0
- 0 0 0 _ 1 1 0 _ 1 1 1 0 1 1 0 -> Even parity, so p4=0
- 0 0 0 0 1 1 0 _ 1 1 1 0 1 1 0 -> Odd parity, so p8=1
- 000011011110110

Hamming Code

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	✓		✓		✓		✓		✓		✓		✓		✓		✓		✓		...
	p2		✓	✓			✓	✓			✓	✓			✓	✓			✓	✓		
	p4				✓	✓	✓	✓					✓	✓	✓	✓					✓	
	p8								✓	✓	✓	✓	✓	✓	✓	✓						
	p16																✓	✓	✓	✓	✓	

- To decode a Hamming Code:
- Check the parity of each row. If all the rows are correct, no error occurred
- If one bit gets corrupted, that'll flip the parities on its column.
- Note: Each bit (including parity bits) has a unique set of parity bits it affects, so we can always identify which bit got corrupted
- Using that information, we can figure out fix the corrupted bit, and return the correct data
- Useful note: If we label bits from 1-n (left to right), the corrupted bit is exactly the binary representation of our parities

Hamming Code: Decoding

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	✓		✓		✓		✓		✓		✓		✓		✓		✓		✓		...
	p2		✓	✓			✓	✓			✓	✓			✓	✓			✓	✓		
	p4				✓	✓	✓	✓					✓	✓	✓	✓					✓	
	p8								✓	✓	✓	✓	✓	✓	✓	✓						
	p16																✓	✓	✓	✓	✓	

- Example: Decode 0b 000 0110 1110 0110 as a Hamming code
- 000011011100110-> Odd parity, so 1
- 000011011100110-> Odd parity, so 1
- 000011011100110-> Even parity, so 0
- 000011011100110-> Odd parity, so 1
- Overall parity is 0b1011 = 11, so bit 11 is corrupted
- 000011011100110
- 0 110 1100110 (Remove parity bits)

Hamming Code

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	✓		✓		✓		✓		✓		✓		✓		✓		✓		✓		
	p2		✓	✓			✓	✓			✓	✓			✓	✓			✓	✓		
	p4				✓	✓	✓	✓					✓	✓	✓	✓					✓	
	p8								✓	✓	✓	✓	✓	✓	✓	✓						
	p16																✓	✓	✓	✓	✓	

- Thus, allows for correcting 1-bit errors
- Can we detect 2-bit errors?
  - Yes, since no two bits flip all parities an even number of times
- Can we detect 3-bit errors?
  - No; if bits 1,2,3 were corrupted, all our parities would be even.
- Is this optimal to correct 1-bit errors?
  - Yes! We won't go into details tho... math is hard
  - Note: CS 70 describes an ECC that can correct 1 error with 2 extra packets (Berlekamp-Welch). This doesn't apply here because Berlekamp-Welch allows only up to n packets sent at once, where n is the number of letters in our alphabet. For binary, Berlekamp-Welch would only be able to send at most 2 bits at a time.

RAID

RAID

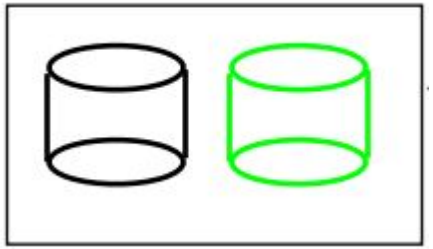
- Generally, the larger you make a disk, the faster it breaks. As such, it's useful to set up a system where you use many disks together
  - Goal: Ensure that even if some disks fail, you still have all your data
  - Goal: Maximize the amount of data you can store
- RAID: Redundant Array of Independent Disks
- 7 different formats (called levels) that can be used to save data (RAID 0 to RAID 6)
- RAIDs 2, 3, and 4 are mostly obsolete nowadays, but we'll still mention them (since they lead into RAID 5)
- Note: The data written on the disks get Hamming-encoded, so single-bit errors aren't a concern. If a disk fails, we'll know which disk failed, and will lose all data on it.
- We'll use for most of these examples a system where we have 4 1 TiB disks we want to put into a RAID array.

RAID 0

- No redundancy
- Split data over all disks
- Ex. 4 1TiB drives combine to store 4 TiB total
- Likely better than a single 4 TiB drive because each disk can output data independently (so 4x bandwidth)
- Generally useful if redundancy isn't needed
- On the other hand, if any disk fails, the entire array fails (because data was lost)

RAID 1

- Maximum redundancy
- Each disk stores an exact copy of the data
- Ex. 4 1 TiB drives combine to store 1 TiB, and three backups
- Very reliable (almost always available), but also extremely expensive
- Useful for when a system cannot be allowed to fail (ex. life support systems), or when memory use isn't too important.

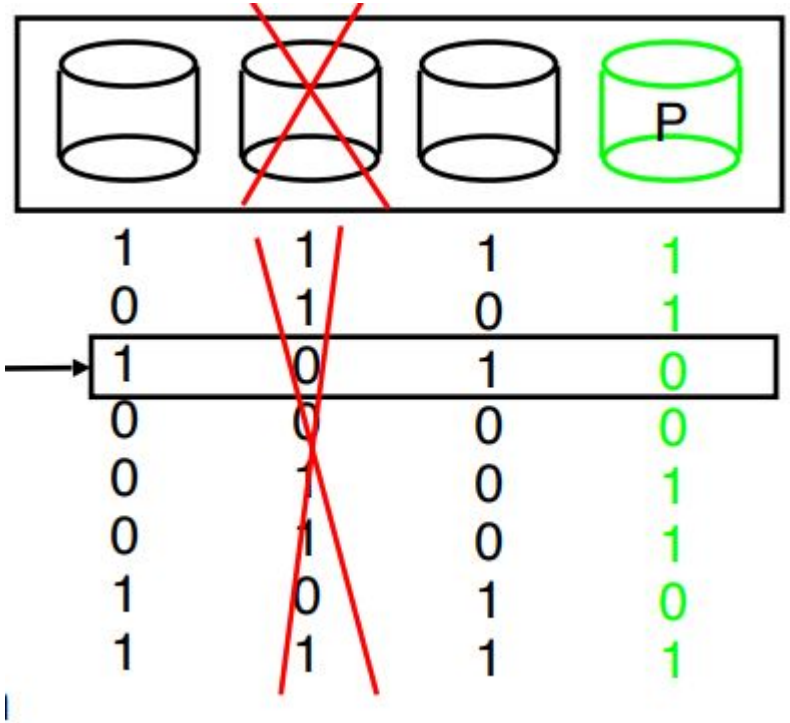


RAID 2

- Split the data into several data disks
- Each bit gets encoded with a Hamming Code, split across all disks
- Ex. If we had 7 1 TiB disks, we would save 4 TiB worth of data. The remaining 3 disks will be used to store parity bits such that the first bit of each disk form a valid 7-bit Hamming code, and so on.
- Basically never used; if you want to recover from two disk failures, use RAID 6

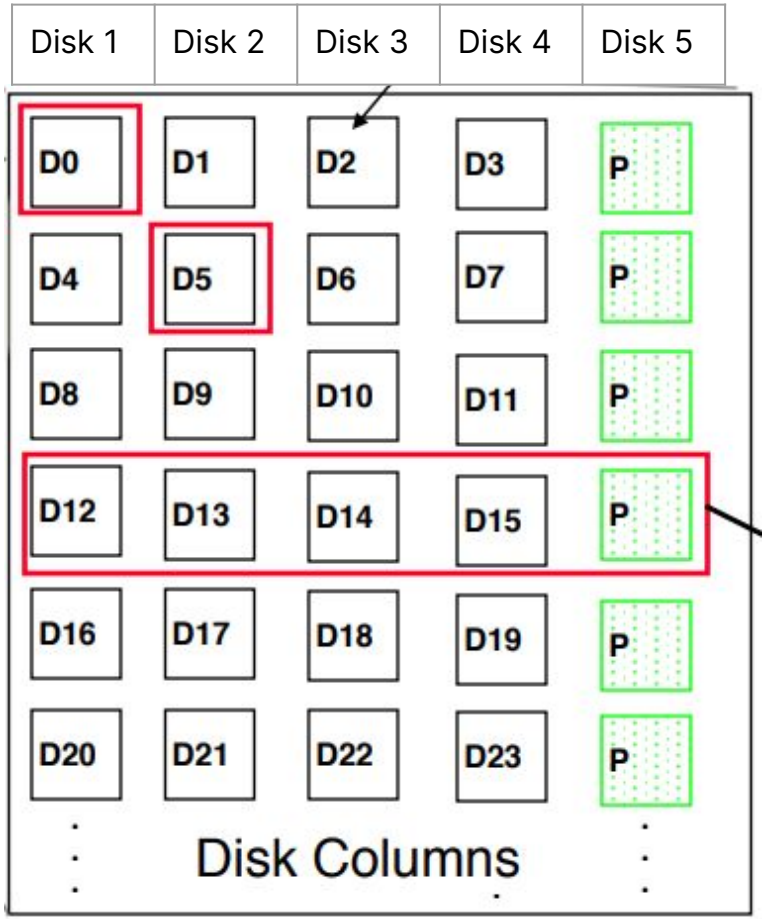
RAID 3

- Split the data into several data disks
- The last disk is a parity disk, and stores the parity bits of the remaining disks
- Ex. 4 1 TiB disks will have 3 TiB of storage and 1 parity disk
- If a disk fails, we can recover the old data by taking the parity of all remaining disks. Thus, we can recover from one disk failure.
- Never used, since worse than RAID 4



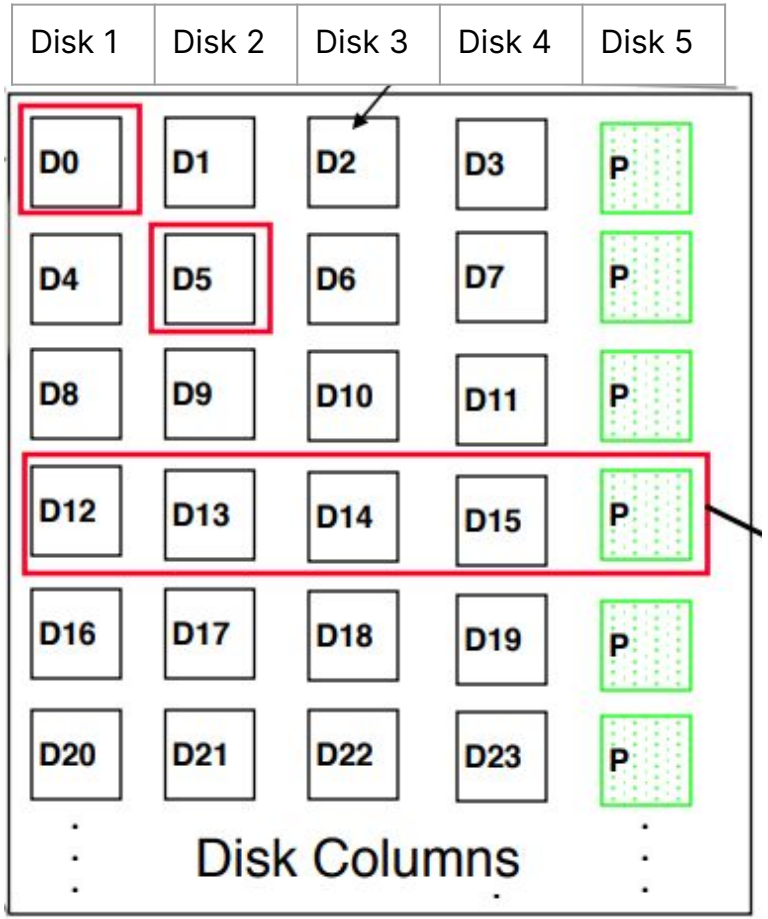
RAID 4

- Same as RAID 3, but we instead work at a block level
- Instead of writing one byte at a time, chunk memory into blocks of ~100 KiB, and read/write blocks as needed
- Use caches to combine multiple small reads/writes in a single operation



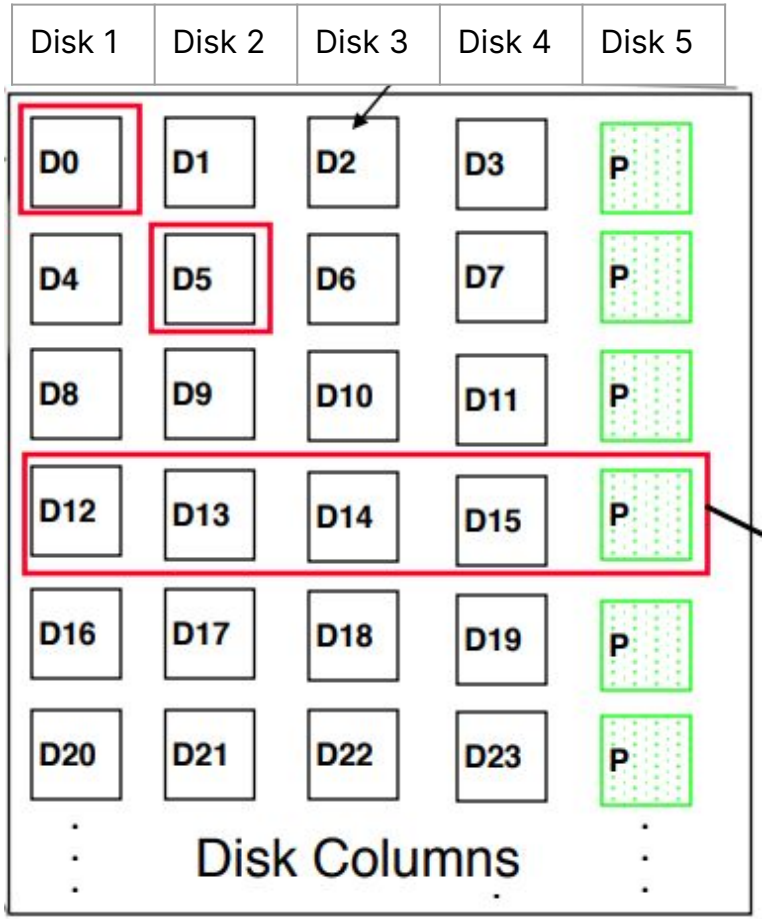
Problem with RAID 4

- Generally, it is possible to do one operation per disk (A single disk can do a single read/write at a time)
- It's also possible to do reads/writes from two different disks at the same time.
- Ex. In the image on the left (5 disks), if we want to read D0 and D5, we can do so at the same time (1 unit of time). If we wanted to read D0 and D4, though, it would take 2 units of time.
- Memory accesses are way longer than math operations, so we only really care about disk reads/writes



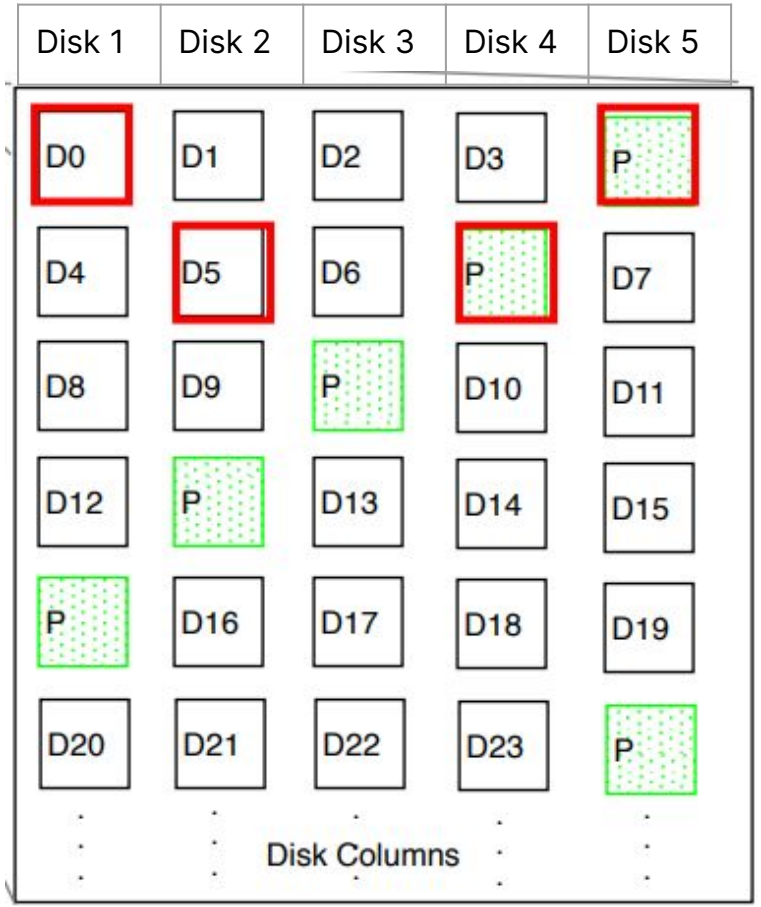
Problem with RAID 4

- What if we want to write to D0 and D5?
  - We need to update the parity blocks too
- We end up writing to D0, D5, and two different P blocks
- Disk 1 and 2 get 1 write each, Disk 5 gets 2 writes
  - Takes 2 units of time to finish
- What if we had 100 blocks to write?
  - Disks 1-4 need only 25 units of time, but Disk 5 takes 100 units of time to access blocks
- This ends up bottlenecking writes



RAID 5

- Same as RAID 4, but this time, we interleave parity blocks over all disks
- If we want to write to both D0 and D5, disks 1, 2, 4, and 5 all get one write each, so we can finish all writes in 1 unit of time
- If we want to write to 100 random blocks, each disk will get ~40 accesses each, so this evens up the work



RAID 6

- RAID 5's good if we only want to handle one disk failure at a time
  - As long as disk failures are independent, RAID 5 will likely work
- However, failures often happen in close proximity
  - If the Golden Gate Bridge collapses due to earthquake, there's a good chance the Hayward Bridge will also collapse due to earthquake fairly soon...
- RAID 6 introduces a second parity block, (so we get one fewer disk to store data).
- This allows us to handle up to two disk failures
  - Since we work with blocks instead of bits, the number of disks is less than the number of letters in our alphabet, so we don't need to do Hamming Encoding, and can just do Berlekamp-Welch or a derivative

Other RAID options

- We can nest RAID levels to fine-tune our array to the situation
  - Ex. If we have 4 disks, we can treat it as two sets of 2 disks each, with the disks in a set having identical data. This is called RAID 10, since it's two RAID 1 systems that combine in a RAID 0 format.
- We can also keep a spare disk entirely unused (but already installed), so we can replace it quickly when one disk fails, to reduce the chance that a third disk fails before we finish repairing the first one.
  - Called a "hot spare".

Conclusion

- Failures will occur
- However, we have ways to mitigate the damage
- Generally, there's a tradeoff between dependability and memory use/runtime, so this ends up being another factor to consider when making design decisions.