CS 61C

Great Ideas in Computer Architecture

Dependability: Parity, ECC, RAID

Instructor: Sean Farhat

This lecture is sponsored by:

RAID SHADOW LEGENDS

(for legal purposes, it's actually not)

Systems & Fault Tolerance

- So far in this course:
  - How software works
  - How hardware works
  - How computers combine software/hardware
- Now:
  - What happens when the real world gets involved!
  - Things break in mysterious ways
  - We can “count on” things breaking!

How can we ensure the systems we’ve developed function reliably (or predictably) in the face of failure?

Measurements of Fault Tolerance, Mitigations

- Dependability
  - Avoiding failure! Build “strong” systems :)
- Redundancy
  - Replication!
  - Make copies of your important state/info so that if it breaks, you can use a replica
- Error Correcting
  - If we do incur some damage, how can we:
    - Find the problem? (Error detection)
    - Fix the problem? (Error correction)

Agenda

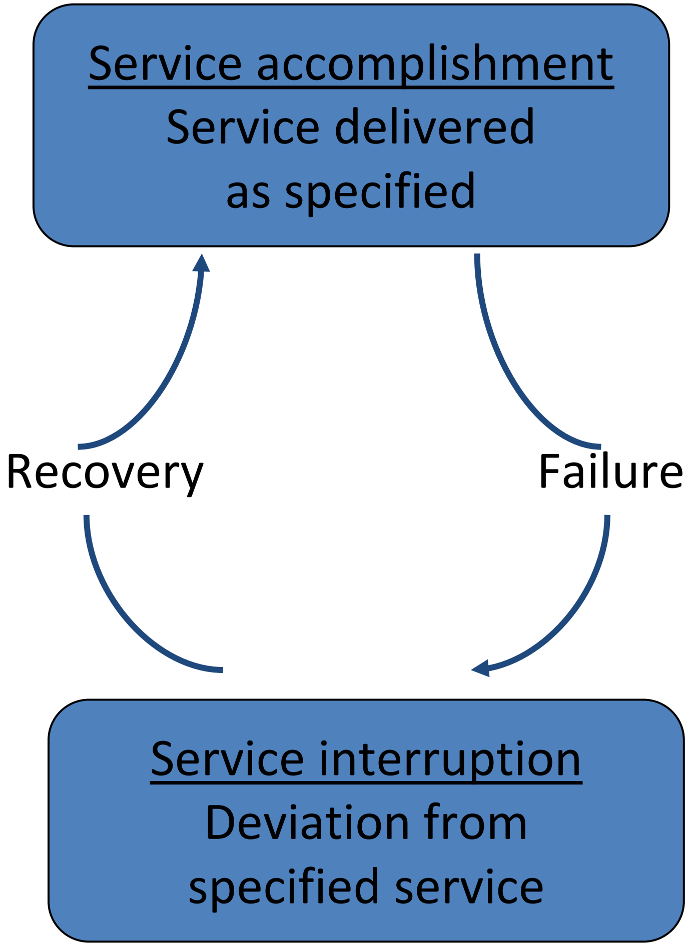
- Dependability
- Error Correcting Codes (ECC)
- RAID

What is Dependability?

- A component, or system, is considered highly dependable if it is highly available, or has a low probability of service outages.
  - Unlikely to encounter failures
  - Recovers quickly when failures occur

- The dependability of a system is determined by the overall dependability of its components

Dependability



- **Fault:** failure of a component
  - May or may not lead to system failure
  - Applies to *any* part of the system
- **Repair**: the act of restoring normal service after a fault

Dependability Measures

- *Reliability:* Mean Time To Failure (MTTF)
- *Service interruption:* Mean Time To Repair (MTTR)
- Mean Time Between Failures (MTBF)
  - MTBF = MTTR + MTTF

- Availability = $\dfrac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} = \dfrac{\text{MTTF}}{\text{MTBF}}$

- Improving Availability
  - Increase MTTF: more reliable HW/SW + fault tolerance
  - Reduce MTTR: improved tools and processes for diagnosis and repair

Reliability Measures

1) MTTF, MTBF measured in hours/failure
   – e.g. "average HDD MTTF is 100,000 hr/failure"
2) Annualized Failure Rate (AFR)
   – Average rate of failures per year (%)

$$\mathrm{AFR} = \underbrace{\left(\frac{\mathrm{Disks}}{\mathrm{MTTF}} \times 8760\,\frac{\mathrm{hr}}{\mathrm{yr}}\right)}_{\text{Total disk failures/yr}} \times \frac{1}{\mathrm{Disks}} = \frac{8760\ \mathrm{hr/yr}}{\mathrm{MTTF}}$$

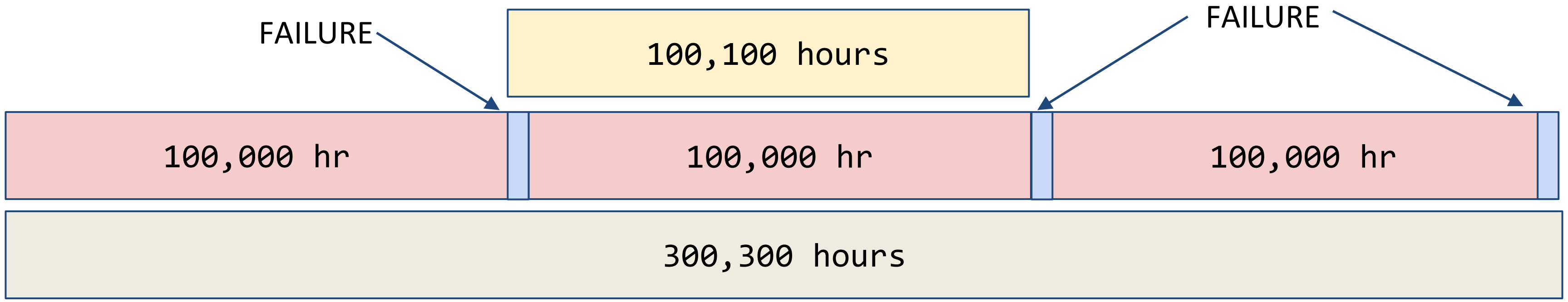
Rate that disks fail per hour

Availability Measures

- Availability = MTTF / (MTTF + MTTR) usually written as a percentage (%)
- Want high availability, so categorize by “number of 9s of availability per year”
  - 1 nine: 90% => 36 days of repair/year
  - 2 nines: 99% => 3.6 days of repair/year
  - 3 nines: 99.9% => 526 min of repair/year
  - 4 nines: 99.99% => 53 min of repair/year
  - 5 nines: 99.999% => 5 min of repair/year

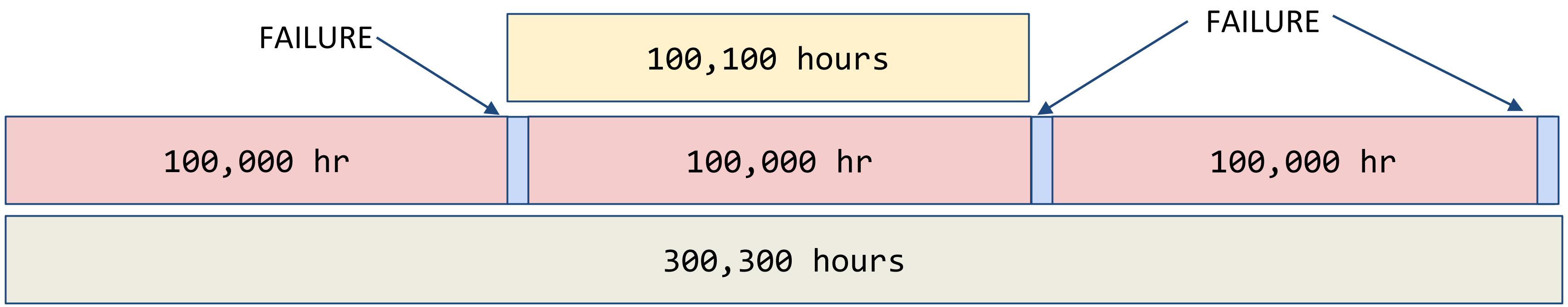
Dependability Example

- 1000 disks with **MTTF = 100,000 hr** and **MTTR = 100 hr**
  - **MTBF**= MTTR + MTTF = 100,100 hr



Dependability Example

- 1000 disks with MTTF = 100,000 hr and MTTR = 100 hr
  - MTBF = MTTR + MTTF = 100,100 hr
  - Availability = **MTTF**/**MTBF** = 0.9990 = 99.9%



Calculating MTTR Example

- Faster repair to get 4 nines of availability?
- GOAL: Closer to x/x == 1, smaller MTTR!
  - $0.9999 = MTTF / (MTTF + MTTR)$
  - $0.9999 \times (MTTF + MTTR) = MTTF$
  - $0.9999 \times MTTF + 0.9999 \times MTTR = MTTF$
  - $0.9999 \times MTTR = 0.0001 \times MTTF$
    - Plug in MTTF = 100,000 hr
  - MTTR = 10.001 hr

Dependability Design Principle

- No single points of failure
  - "Chain is only as strong as its weakest link"
- Dependability Corollary of Amdahl's Law
  - Doesn't matter how dependable you make *one* portion of system because dependability is limited by the part you do not improve
- In 2013, Google had 99.978% availability for their cloud clusters!

Question

- There's a hardware glitch in our system that makes the Mean Time To Failure (MTTF) *decrease*. Are the following statements TRUE or FALSE?

1) Our system's Availability will *increase*.

2) Our system's Annualized Failure Rate (AFR) will *increase*.

	12
(A)	**FF**
(B)	**FT**
(C)	**TF**
(D)	**TT**

Question

- There's a hardware glitch in our system that makes the Mean Time To Failure (MTTF) *decrease*. Are the following statements TRUE or FALSE?

1) Our system's Availability will *increase*.

2) Our system's Annualized Failure Rate (AFR) will *increase*.

	12
(A)	**FF**
(B)	**FT**
(C)	**TF**
(D)	**TT**

Question

- There's a hardware glitch in our system that makes the Mean Time To Failure (MTTF) *decrease*. Are the following statements TRUE or FALSE?

1) Our system's Availability will *increase*.

2) Our system's Annualized Failure Rate (AFR) will *increase*.

	12
(A)	FF
(B)	FT
(C)	TF
(D)	TT

Availability = MTTF / (MTTF + MTTR)

As MTTF shrinks, our fraction is outweighed by MTTR:

100/(100 + 50) → 66%
50/(50 + 50) → 50%
10/(10 + 50) → 16%

Availability decreases (less 9's, lower percentage)

Question

- There's a hardware glitch in our system that makes the Mean Time To Failure (MTTF) *decrease*. Are the following statements TRUE or FALSE?

1) Our system's Availability will *increase*.

2) Our system's Annualized Failure Rate (AFR) will *increase*.

	12
(A)	FF
(B)	$\boxed{\text{FT}}$
(C)	TF
(D)	TT

MTTF = "Mean time to Failure" = average time until something goes wrong!

If this number /decreases/, this means failures happen more often!
100 years until failure, 50 years until failure, 10 years until failure, etc.

Failures happening more often == an increased rate of failures per year

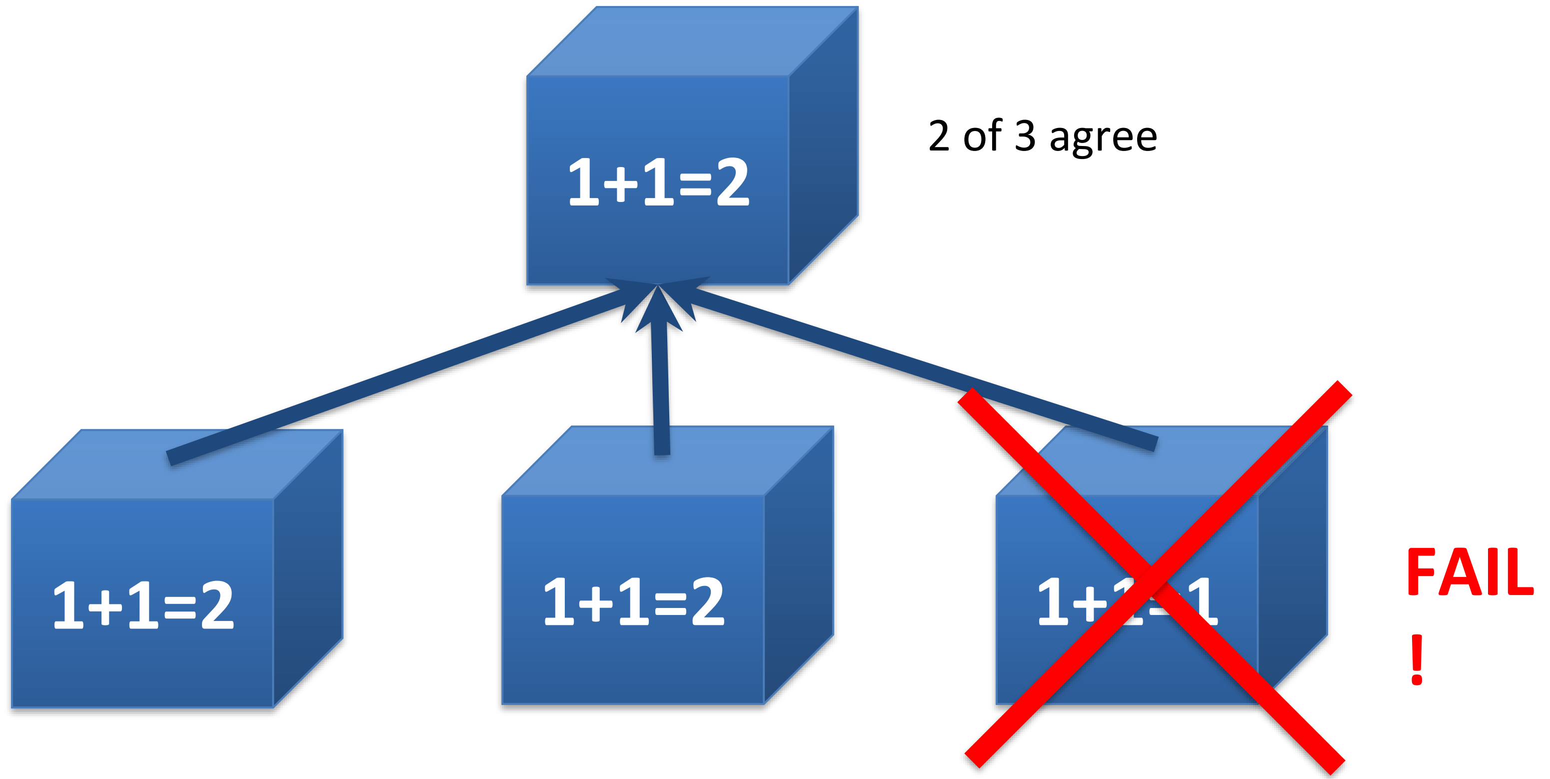
As MTTF decreases, AFR increases

Agenda

- Dependability
- Error Correcting Codes (ECC)
- RAID

Great Idea: Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail



Great Idea: Dependability via Redundancy

- Applies to everything from datacenters to memory
  - Redundant datacenters so that can lose 1 datacenter but Internet service stays online
  - Redundant routes so can lose nodes but Internet doesn't fail
  - Redundant disks so that can lose 1 disk but not lose data (Redundant Arrays of Independent Disks/RAID)
  - Redundant memory bits of so that can lose 1 bit but no data (Error Correcting Code/ECC Memory)

Error Detection/Correction Codes

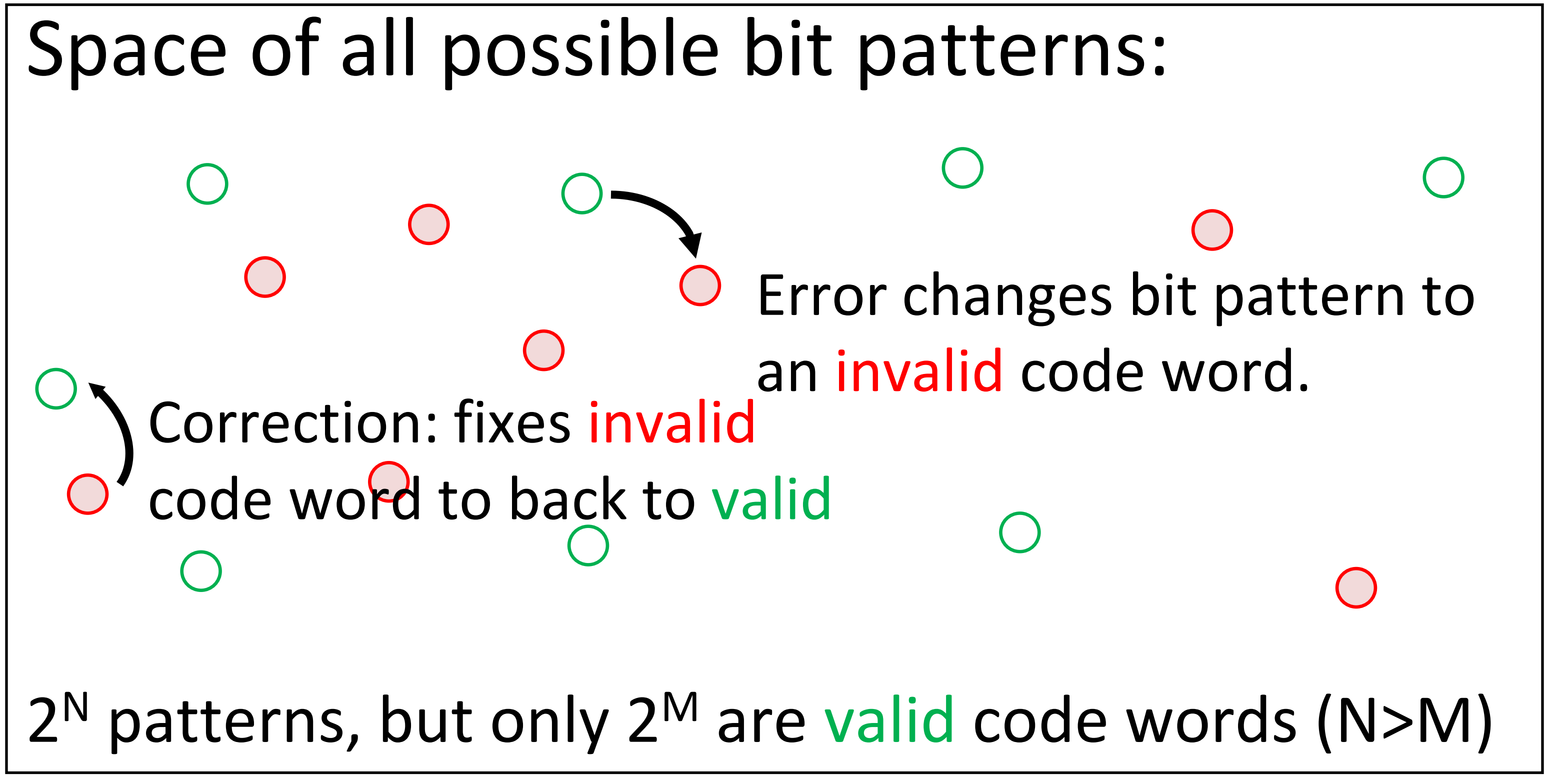
- Memory systems generate errors (accidentally flipped-bits)
  - DRAMs store very little charge per bit
  - “Soft” errors occur occasionally when cells are struck by alpha particles or other environmental upsets
  - “Hard” errors occur when chips permanently fail
  - Problem gets worse as memory systems get denser and larger

Error Detection/Correction Codes

- Protect against errors with EDC/ECC
- Extra bits are added to each M-bit data chunk to produce an N-bit "code word" (N>M)
  - Extra bits are a function of the data
  - Each data word value is mapped to a valid code word
  - Certain errors change *valid* code words to *invalid* ones (i.e. you can tell something is wrong)

Detecting/Correcting Code Concept

- *Detection:* fails code word validity check
- *Correction:* can map to nearest valid code word



Hamming Distance

- Hamming distance = # of bit changes to get from one code word to another
  - p = 011011,
    q = 001111, Hdist(p,q) = 2
  - p = 011011,
    q = 110001, Hdist(p,q) = 3
  - If all code words are valid, then *min Hdist between valid code words is 1*
    – Change one bit, at another valid code word

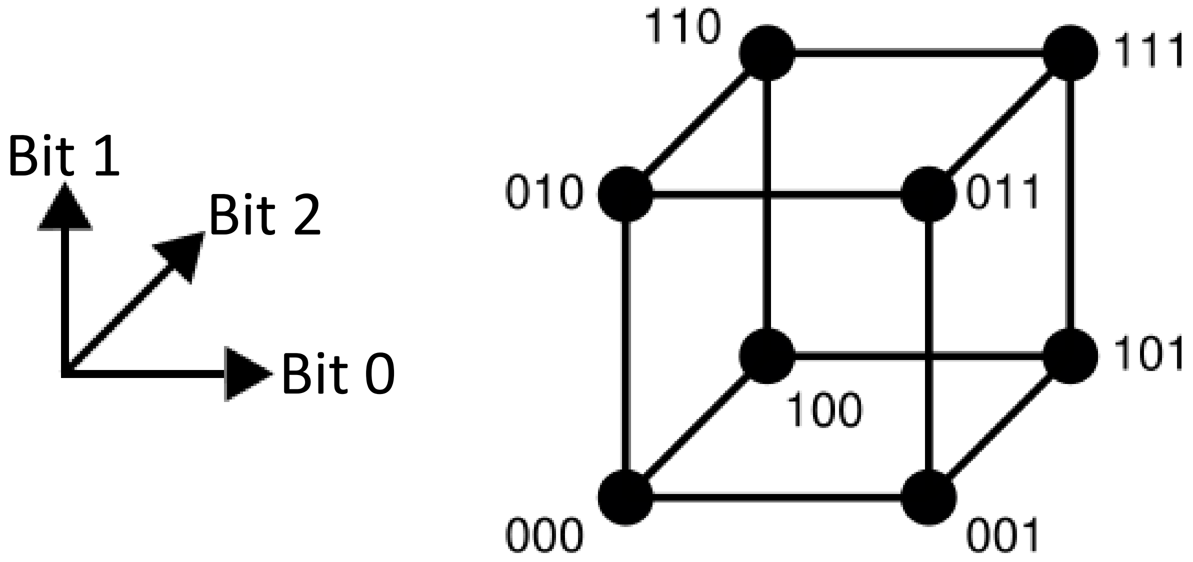
Richard Hamming (1915-98)
Turing Award Winner

Why does this matter?

- If the entire "space" of possible codewords is valid, we can't tell if our word has errors (maybe it morphed into another valid word!)
- If some words are valid, and others are invalid, we can detect errors
- If there are a small subset of valid words and a large subset of invalid words, we can trace/track our errors and revert them!
  - Wait really????

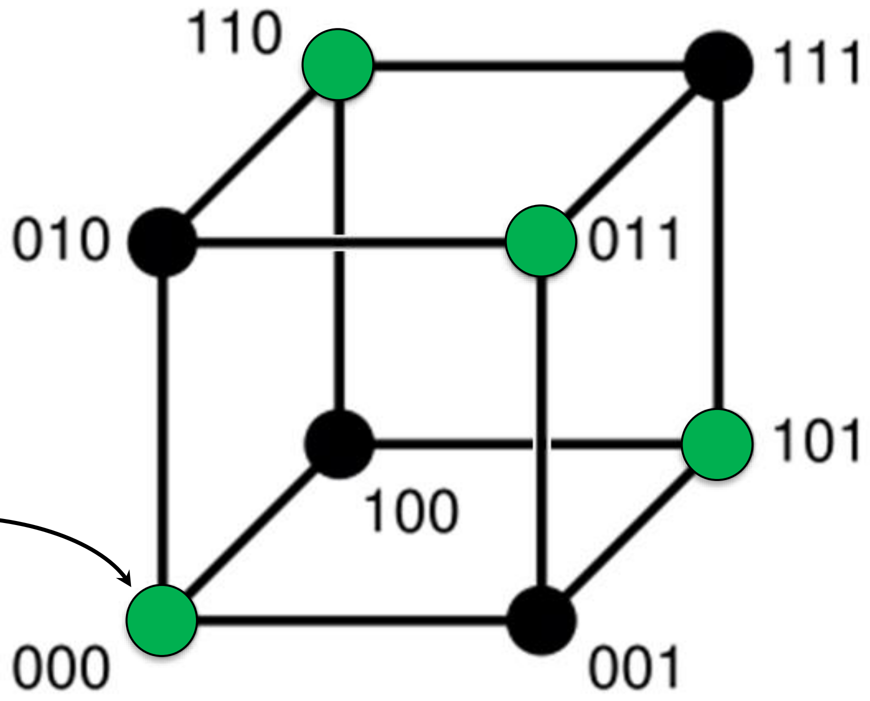
3-Bit Visualization Aid

- Want to be able to *see* Hamming distances
  - Show code words as *nodes*, Hdist of 1 as *edges*
- For 3 bits, show each bit in a different dimension:



Minimum Hamming Distance 2

Let 000
be valid



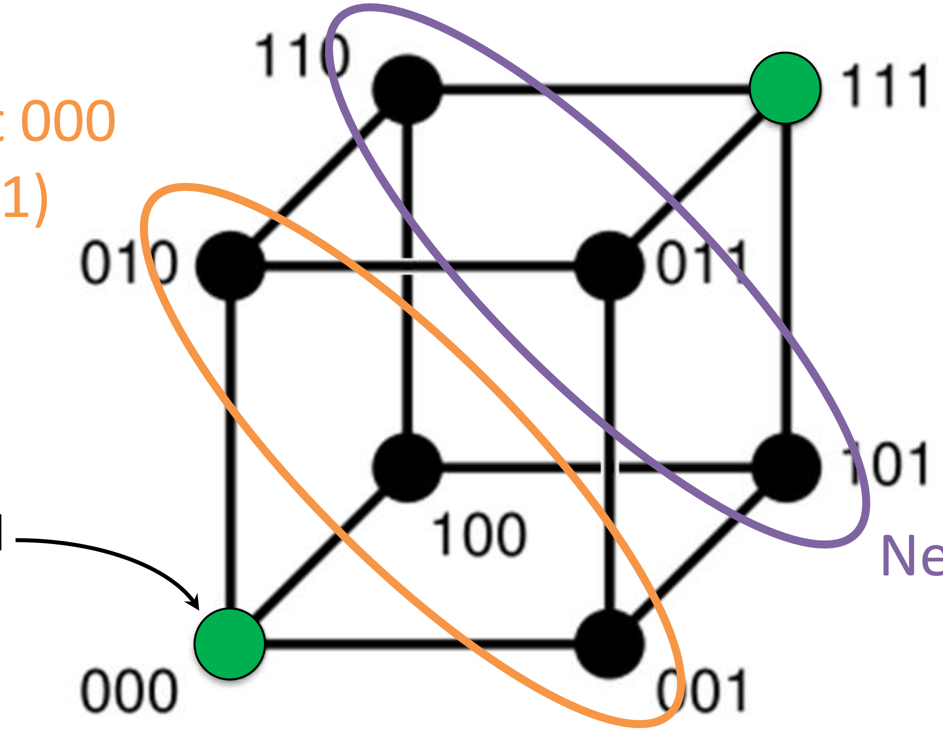
Half the available
code words
are valid

- If 1-bit error, is code word still valid?
  – No! So can *detect*
- If 1-bit error, know which code word we came from?
  – No! Equidistant, so cannot *correct*

Minimum Hamming Distance 3

Nearest 000
(one 1)

Let 000
be valid



Only a *quarter* of the available code words are valid

Nearest 111
(one 0)

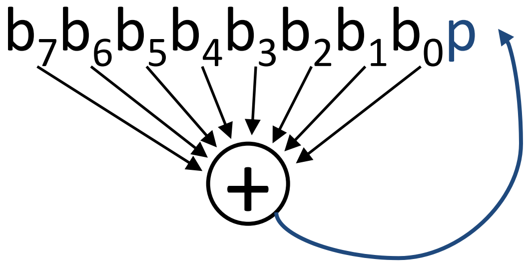
- How many bit errors can we detect?
  - Two! Takes 3 errors to reach another valid code word
- If 1-bit error, know which code word we came from?
  - Yes!

Parity Bit

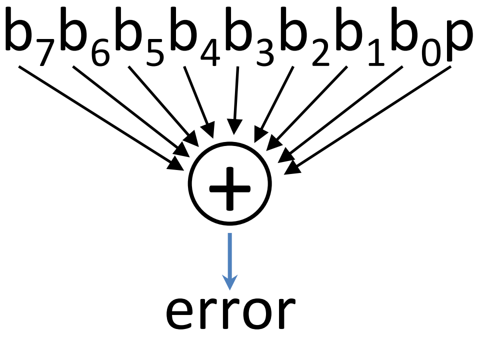
- Describes whether a group of bits contains an even or odd number of 1’s
  - Define 1 = odd and 0 = even
  - Can use XOR to compute parity bit!
- Adding the parity bit to a group will always result in an even number of 1’s (“even parity”)
  - 0b100 Parity: 1, 0b101 Parity: 0
- If we know number of 1’s must be even, can we figure out what a single missing bit should be?
  - 10?11 → missing bit is 1

Parity: Simple Error Detection Coding

- Add parity bit when writing block of data:



- Check parity on block read:
  - Error if odd number of 1s
  - Valid otherwise



- Minimum Hamming distance of parity code is 2
- Parity of code word = 1 indicates an error occurred:
  - 2-bit errors not detected (nor any even # of errors)
  - Detects an odd # of errors

Parity Examples

1) Data 0101 0101
   - 4 ones, even parity now
   - Write to memory
     0101 0101 0
     to *keep* parity even
2) Data 0101 0111
   - 5 ones, odd parity now
   - Write to memory:
     0101 0111 1
     to *make* parity even

3) Read from memory
   0101 0101 0
   - 4 ones → even parity, so no error
4) Read from memory
   1101 0101 0
   - 5 ones → odd parity, so error

- What if error in parity bit?
  - Can detect!

How to Correct 1-bit Error?

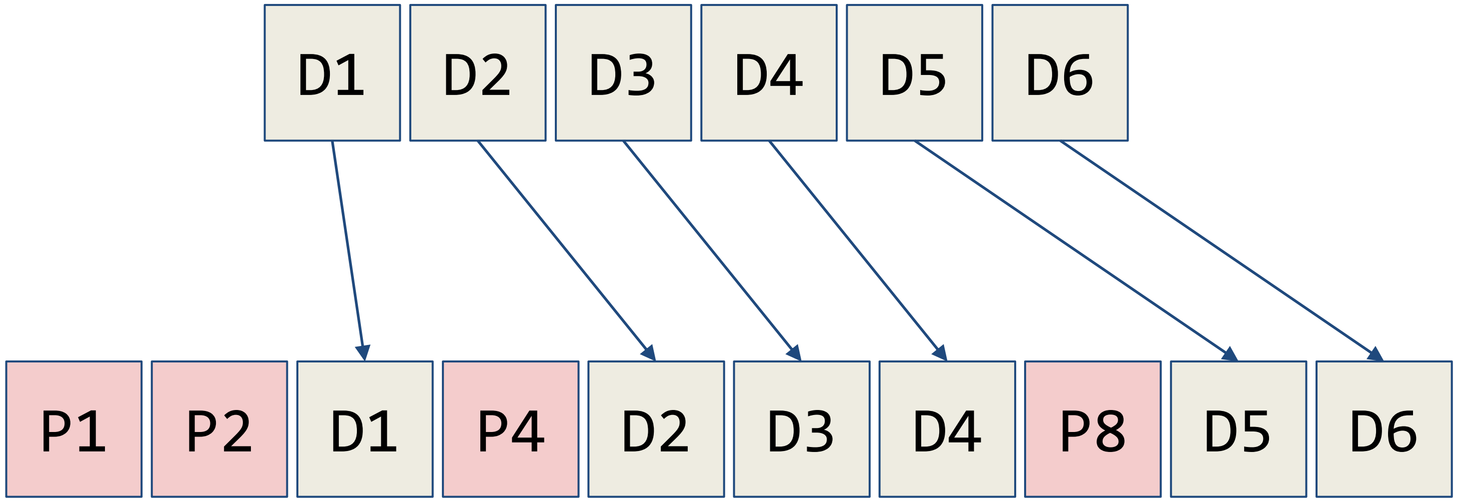
- **Recall:** Minimum distance for correction?
  - Three
- Richard Hamming came up with a mapping to allow Error Correction at min distance of 3
  - Called Hamming ECC for Error Correction Code

Hamming ECC (1/2)

- Use *extra parity bits* to allow the position identification of a single error
  - Interleave parity bits *within* bits of data to form code word
  - **Note:** Number bits starting at 1 from the left

1) Use *all* bit positions in the code word that are powers of 2 for parity bits (1, 2, 4, 8, 16, …)
2) All other bit positions are for the data bits (3, 5, 6, 7, 9, 10, …)

Hamming ECC

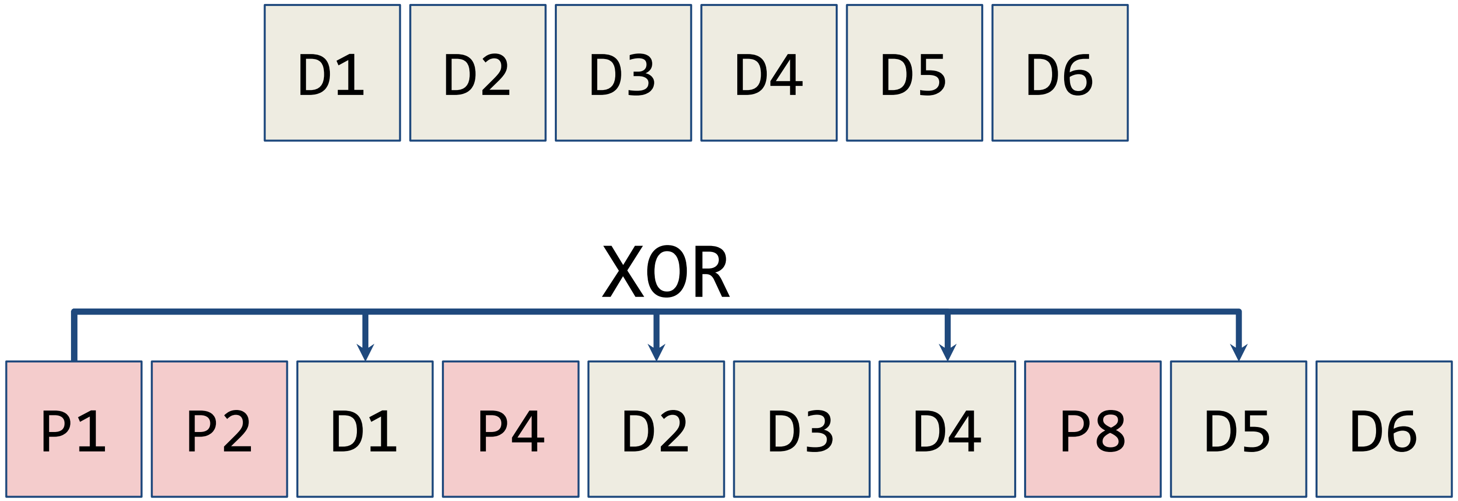


Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		...
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X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

Hamming ECC (2/2)

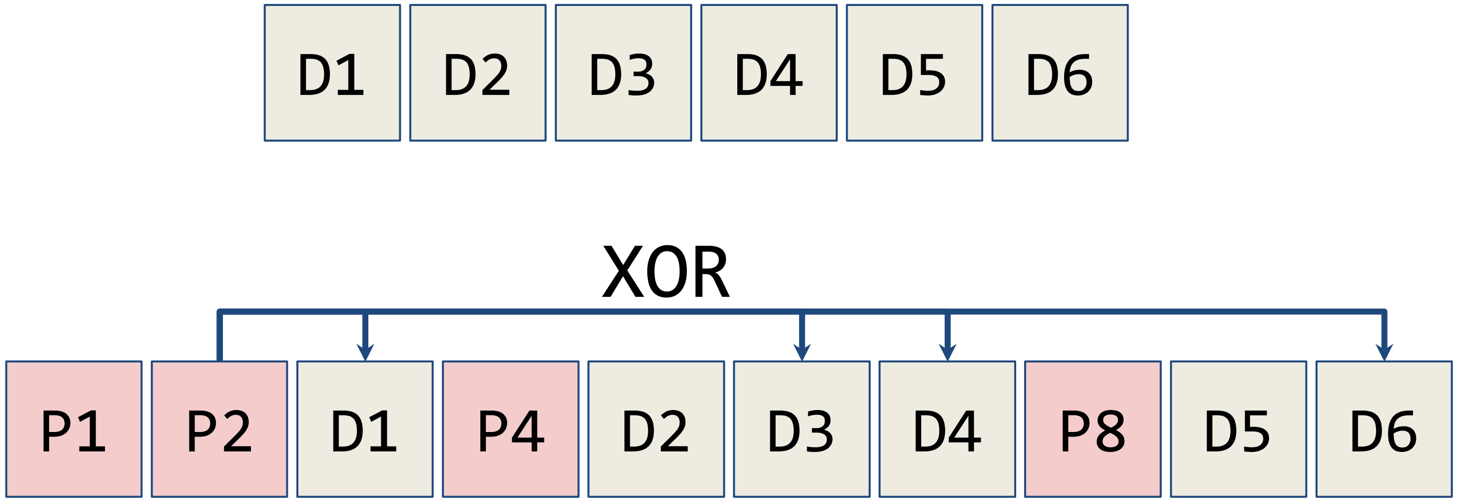
3) Set each parity bit to create even parity for *a group* of the bits in the code word
   - The position of each parity bit determines the group of bits that it checks
   - Parity bit p checks every bit whose position number in binary has a 1 in the bit position corresponding to p
     - Bit 1 (0001_2) checks bits 1,3,5,7, … ($XXX1_2$)
     - Bit 2 (0010_2) checks bits 2,3,6,7, … ($XX1X_2$)
     - Bit 4 (0100_2) checks bits 4-7, 12-15, … ($X1XX_2$)
     - Bit 8 (1000_2) checks bits 8-15, 24-31, … ($1XXX_2$)

Hamming ECC



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	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		...
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

Hamming ECC



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	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		...
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

Hamming ECC Example

- A byte of data: 10011010
- Create the code word, leaving spaces for the parity bits:

$\_\_{}_1\ \_\_{}_2\ 1_3\ \_\_{}_4\ 0_5\ 0_6\ 1_7\ \_\_{}_8\ 1_9\ 0_{10}\ 1_{11}\ 0_{12}$

Hamming ECC Example

- Calculate the parity bits:
  - Parity bit 1 group (1, 3, 5, 7, 9, 11):
    ? _ **1** _ **0** 0 **1** _ **1** 0 **1** 0 → 0
  - Parity bit 2 group (2, 3, 6, 7, 10, 11):
    ? ? 1 _ 0 **0 1** _ 1 **0 1** 0 → 1
  - Parity bit 4 group (4, 5, 6, 7, 12):
    ? ? 1 **? 0 0 1** _ 1 0 1 **0** → 1
  - Parity bit 8 group (8, 9, 10, 11, 12):
    ? ? 1 **?** 0 0 1 **? 1 0 1 0** → 0

Hamming ECC Example

- Valid code word: $\underline{01}1\underline{1}001\underline{0}1010$
- Original data: 1 001 1010

Suppose we read $0_{1}1_{2}1_{3}1_{4}0_{5}0_{6}1_{7}0_{8}1_{9}1_{10}1_{11}0_{12}$ instead – fix the error!

But how would we figure out where the error is if we *just* see the code word? Hmm….

- Let's examine the parity bits that are dependent on bit 10
  - Maybe we can figure out a pattern?

Hamming ECC Example

- Calculate the parity bits for $0_1 1_2 1_3 1_4 0_5 0_6 1_7 0_8 1_9 1_{10} 1_{11} 0_{12}$
  - Parity bit 1 group (1, 3, 5, 7, 9, 11):
    0 + 1 + 0 + 1 + 1 + 1 = 4 (EVEN)
  - Parity bit 2 group (2, 3, 6, 7, 10, 11):
    1 + 1 + 0 + 1 + 1 + 1 = 5 (ODD)
  - Parity bit 4 group (4, 5, 6, 7, 12):
    1 + 0 + 0 + 1 + 0 = 2 (EVEN)
  - Parity bit 8 group (8, 9, 10, 11, 12):
    0 + 1 + 1 + 1 + 0 = 3 (ODD)

Graphic of Hamming Code

0 1 1 1 0 0 1 0 1 1 1 0

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11
Parity bit coverage	p1	X		X		X		X		X		X		X		X
	p2		X	X			X	X			X	X			X	X
	p4				X	X	X	X					X	X	X	X
	p8								X	X	X	X	X	X	X	X

- Looks like p2 and p8 were responsible
    - p2 & p8 will be the only incorrect parity bits IFF bit 10 is incorrect
    - Notice that 2 + 8 = 10
    - Seems like incorrect parity bits tell us where the error is... o:

Hamming ECC Example

- Valid code word: $\underline{01}1\underline{1}001\underline{0}1010$
- Recover original data: 1 001 1010

Suppose we see $0_1 1_2 1_3 1_4 0_5 0_6 1_7 0_8 1_9 1_{10} 1_{11} 0_{12}$ instead – fix the error!

How to figure out where the error is? Hmm….

- Check the parity bits responsible for bit 10
  - Parity bits 2 and 8 are incorrect
  - As 2+8=10, bit position 10 is the bad bit, so flip it!
- Corrected value: $011100101\underline{0}10$

Hold on...

Replace all the X's with 1's

Everywhere else, put a 0

Bit position		1	2	3	4	5	6	7	8	9
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5
Parity bit coverage	p1	X		X		X		X		X
	p2		X	X			X	X		
	p4				X	X	X	X		
	p8								X	X

OMFG

The parity bits form a number—the bit position in the code word!

- This tells you which bit of the code word is incorrect!

Bit position	Encoded data bits	Parity bit coverage p8	p4	p2	p1
1	p1	0	0	0	1
2	p2	0	0	1	0
3	d1	0	0	1	1
4	p4	0	1	0	0
5	d2	0	1	0	1
6	d3	0	1	1	0
7	d4	0	1	1	1
8	p8	1	0	0	0
9	d5	1	0	0	1
10	d6	1	0	1	0
11	d7	1	0	1	1
12	d8	1	1	0	0
13	d9	1	1	0	1
14	d10	1	1	1	0

Hamming ECC Example

Going in reverse:

We see $0_1 1_2 1_3 1_4 0_5 0_6 1_7 0_8 1_9 1_{10} 1_{11} 0_{12}$

Figure out which bit is wrong:

p_1: $0_1 1_3 0_5 1_7 1_9 1_{11}$ = even number of 1's : 0

p_2: $1_2 1_3 0_6 1_7 1_{10} 1_{11}$ = odd number of 1's : 1

p_4: $1_4 0_5 0_6 1_7 0_{12}$ = even number of 1's: 0

p_8: $0_8 1_9 1_{10} 1_{11} 0_{12}$ = odd number of 1's : 1

Incorrect code bit:

0b $p_8 p_4 p_2 p_1$ = 0b 1010 = 10! So flip bit 10

Is This Enough?

- For 1-bit error detection: parity bit
- For 1-bit error correction: Hamming ECC

- What about more bits?
  - For 2-bit error detection: modified Hamming ECC! (see bonus slides)
  - For 2-bit error correction: more advanced schemes such as Reed-Solomon codes (CS 70)

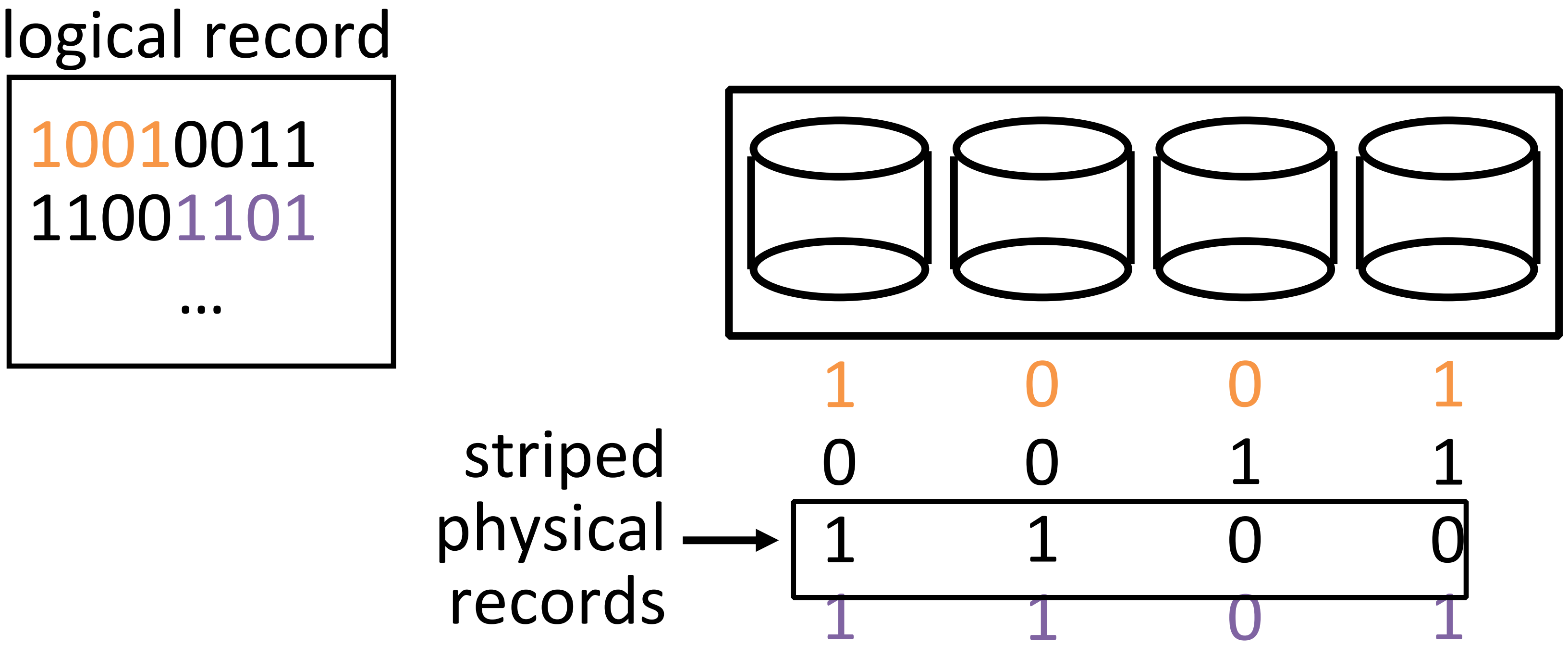
Agenda

- Dependability
- Error Correcting Codes (ECC)
- RAID

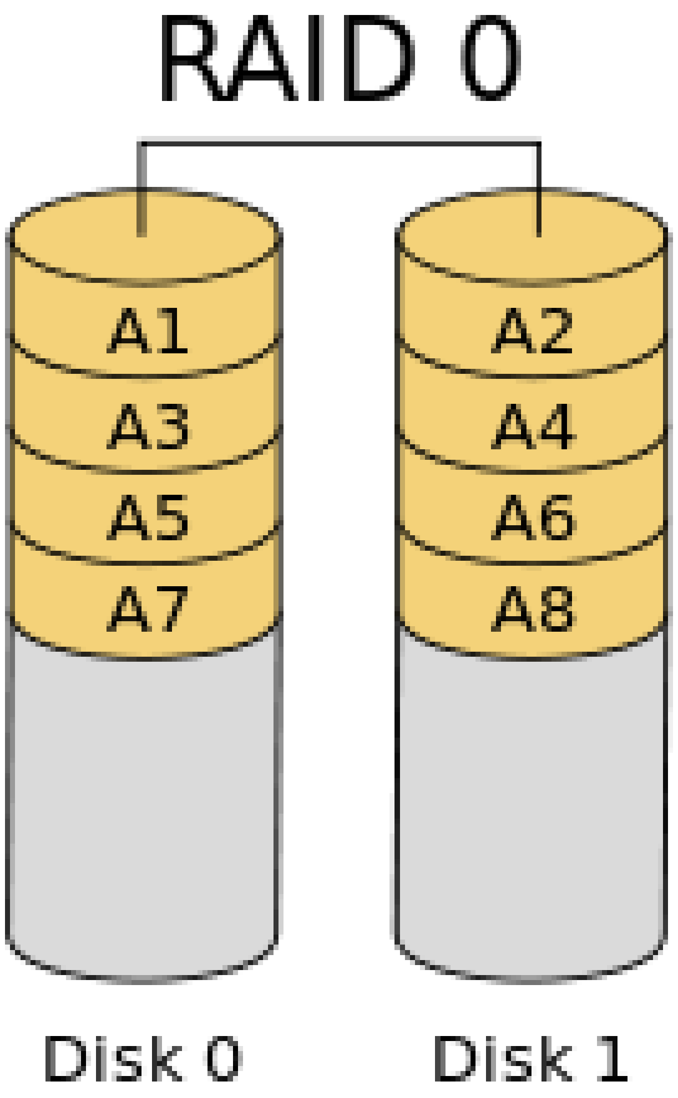
RAID: Redundant Array of Inexpensive/Independent Disks

- Files are "shared" across multiple disks
  - Concurrent disk accesses <u>improve throughput</u>
- Redundancy yields high data availability
  - Service still provided to user, even if some components (disks) fail
- Contents reconstructed from data redundantly stored in the array
  - Can detect when data is corrupted
  - Can fix data/restore correct version
  - Like before, but "bit" is now "disk"

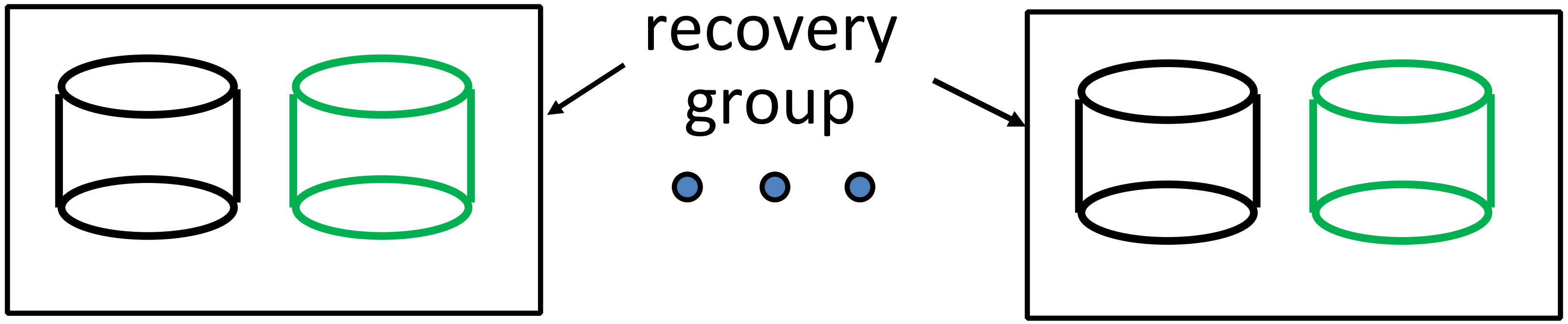
RAID 0: Data Striping



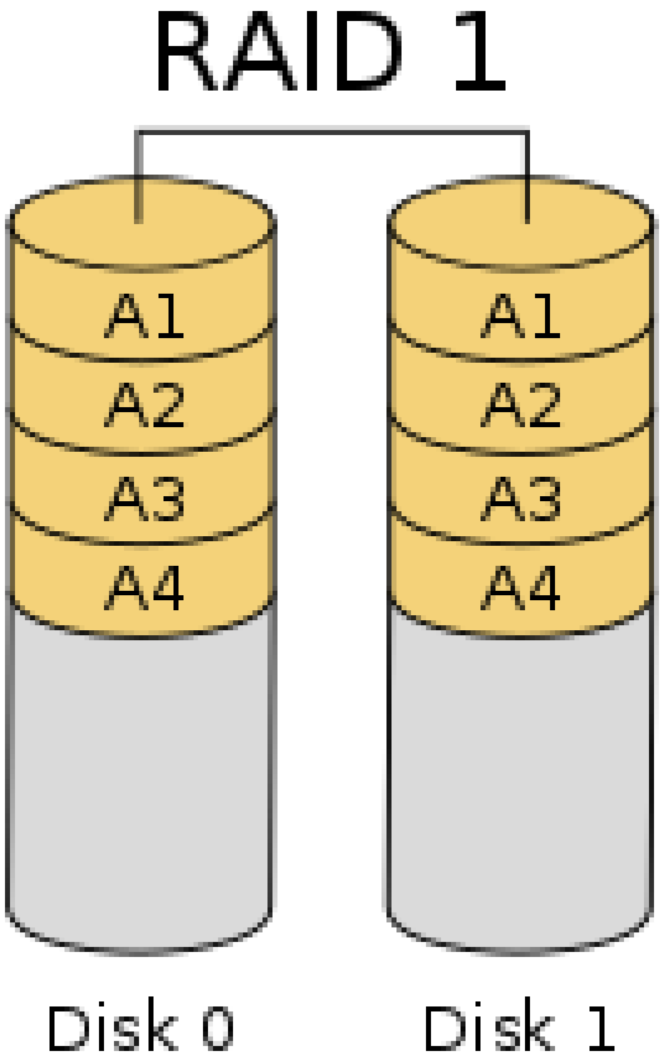
- "Stripe" data across all disks
  - Generally faster accesses (access disks in parallel)
  - No redundancy, cannot tolerate any failures
  - Bit-striping shown here, can do in larger chunks


RAID 0
A1
A2
A3
A4
A5
A6
A7
A8
Disk 0
Disk 1

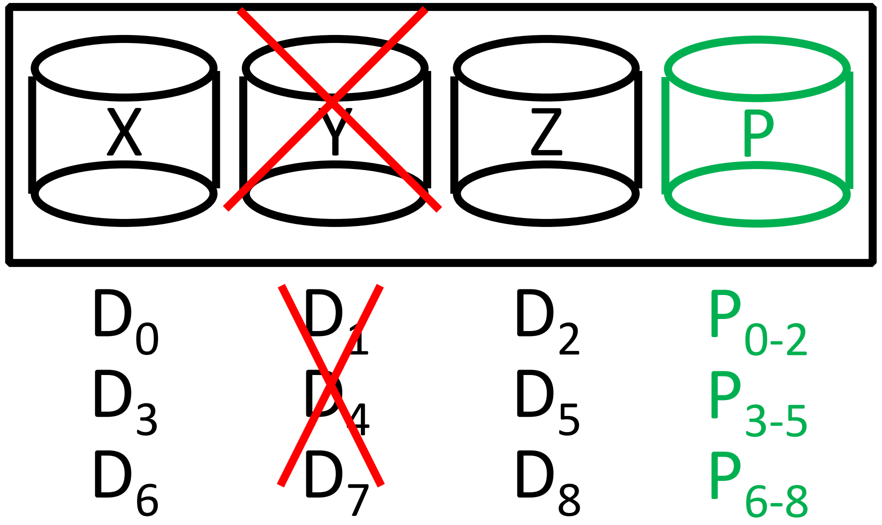
RAID 1: Disk Mirroring



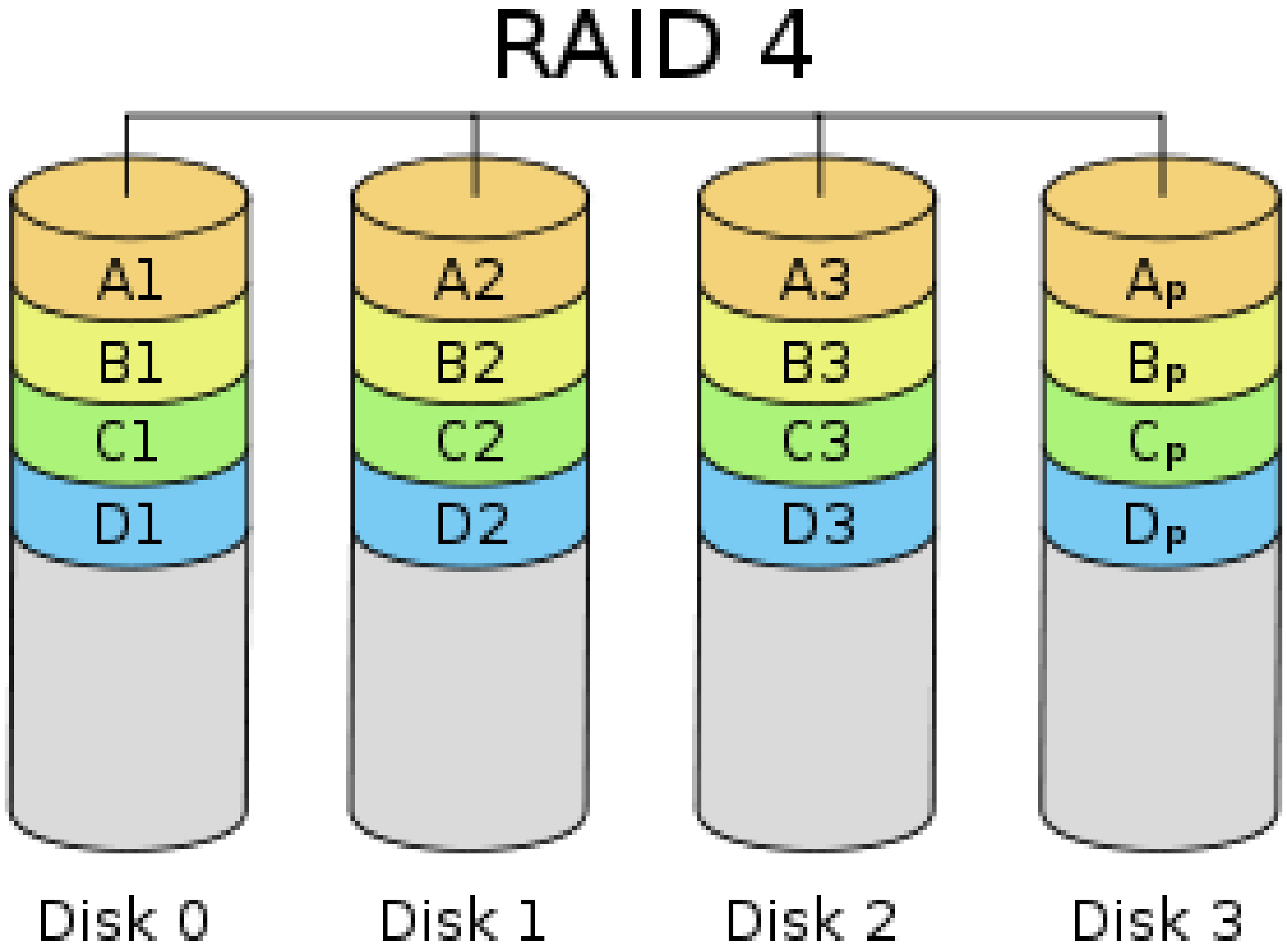
- Each disk is fully duplicated onto its "mirror"
  - Very high availability can be achieved
- Bandwidth sacrifice on write:
  - Logical write = two physical writes
  - Logical read = one physical read
- Most expensive solution: 100% capacity overhead


RAID 1
A1
A2
A3
A4
A1
A2
A3
A4
Disk 0
Disk 1

RAID 2-4: Data Striping + Parity



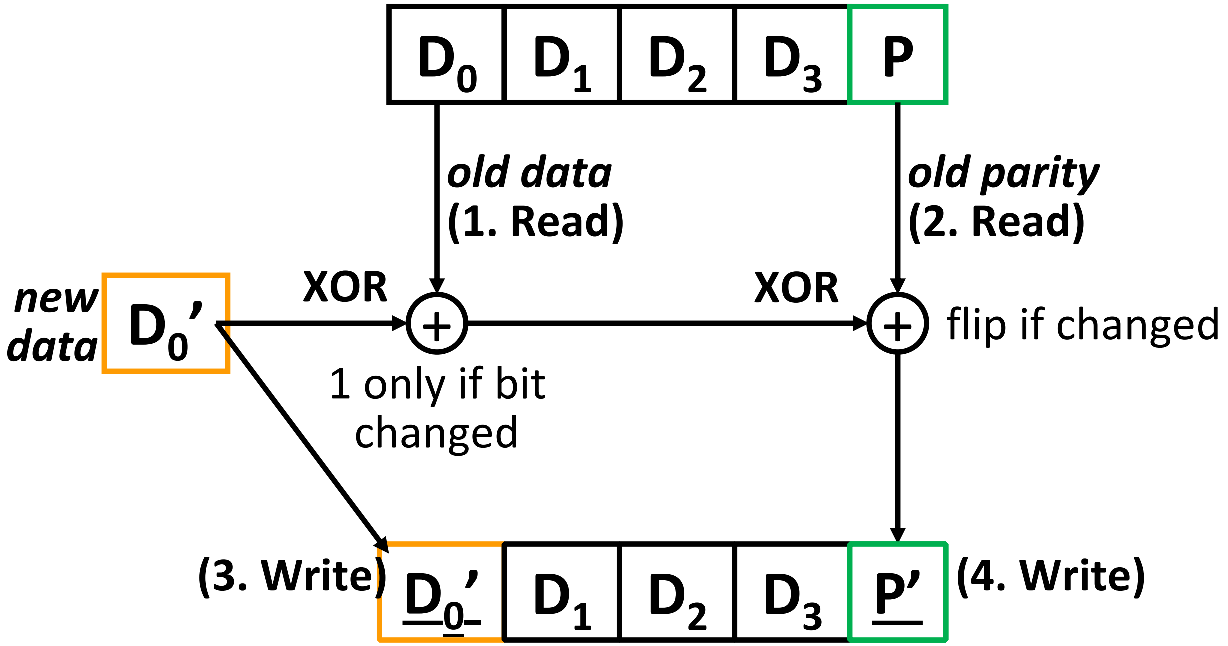
- Logical data is striped across disks
  - 2: bit, 3: byte, 4: block
- Parity disk P contains parity of other disks
- If any one disk fails, can use other disks to recover data!
  - We have to *know* which disk failed
- Must update Parity data on EVERY write
  - Logical write = min 2 to max N physical reads and writes
  - $parity_{new} = data_{old} \oplus data_{new} \oplus parity_{old}$


RAID 4
A1
A2
A3
Ap
B1
B2
B3
Bp
C1
C2
C3
Cp
D1
D2
D3
Dp
Disk 0
Disk 1
Disk 2
Disk 3

Updating the Parity Data

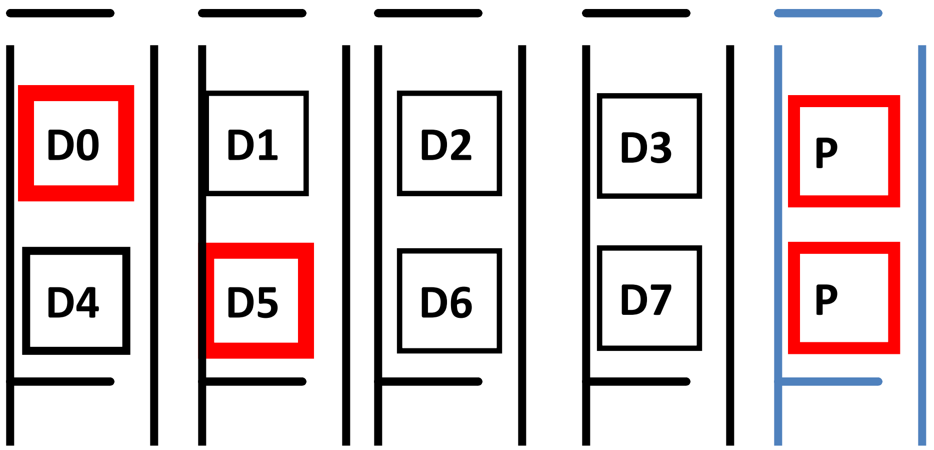
- Examine small write in RAID 3 (1 byte)
  - 1 logical write = 2 physical reads + 2 physical writes
  - Same concept applies for later RAIDs, too

D_0'	D_0	P	P'
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



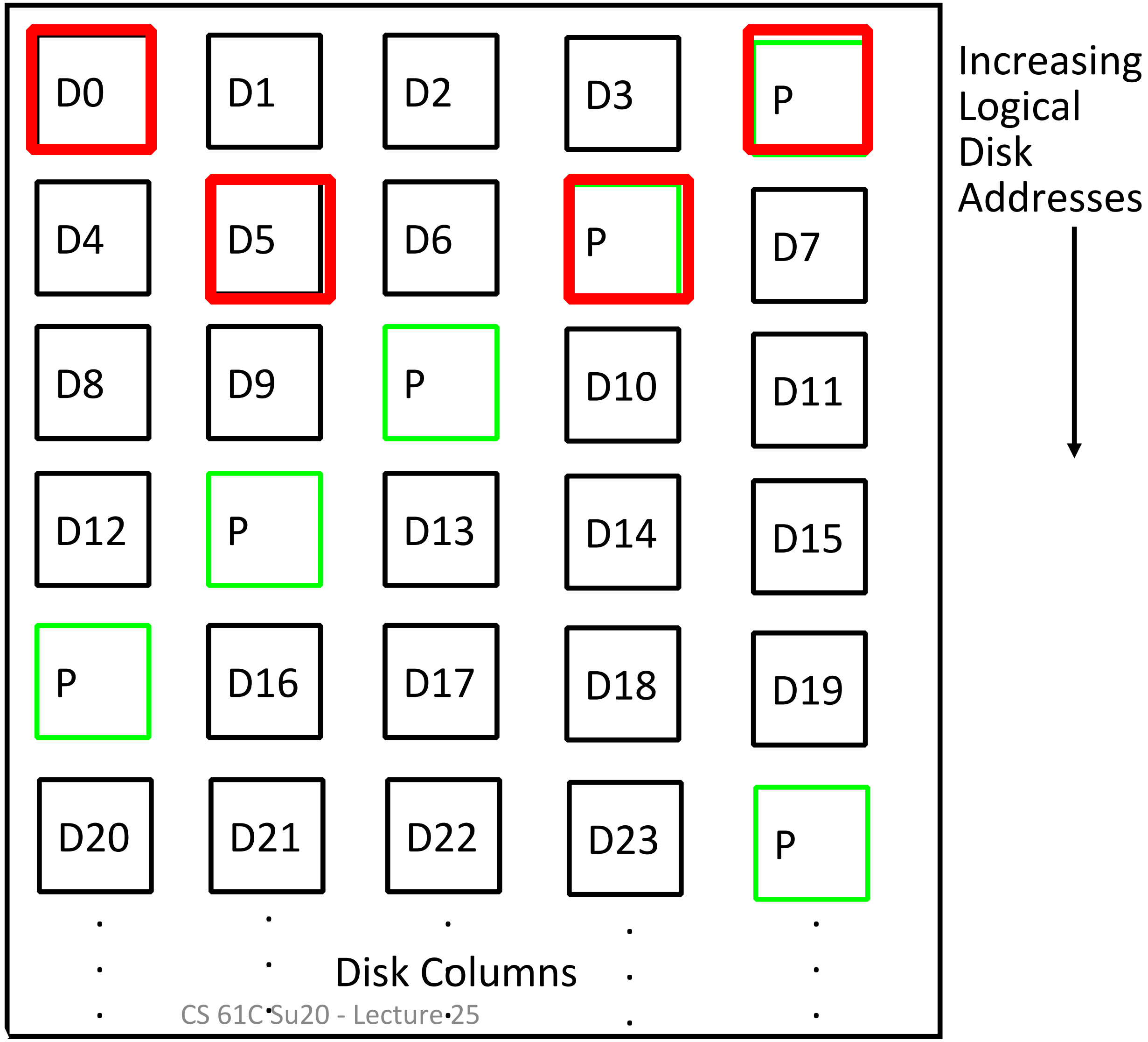
Inspiration for RAID 5

- When writing to a disk, need to update Parity
- Small writes are bottlenecked by Parity Disk: Write to D0, D5 but also write to P disk twice!

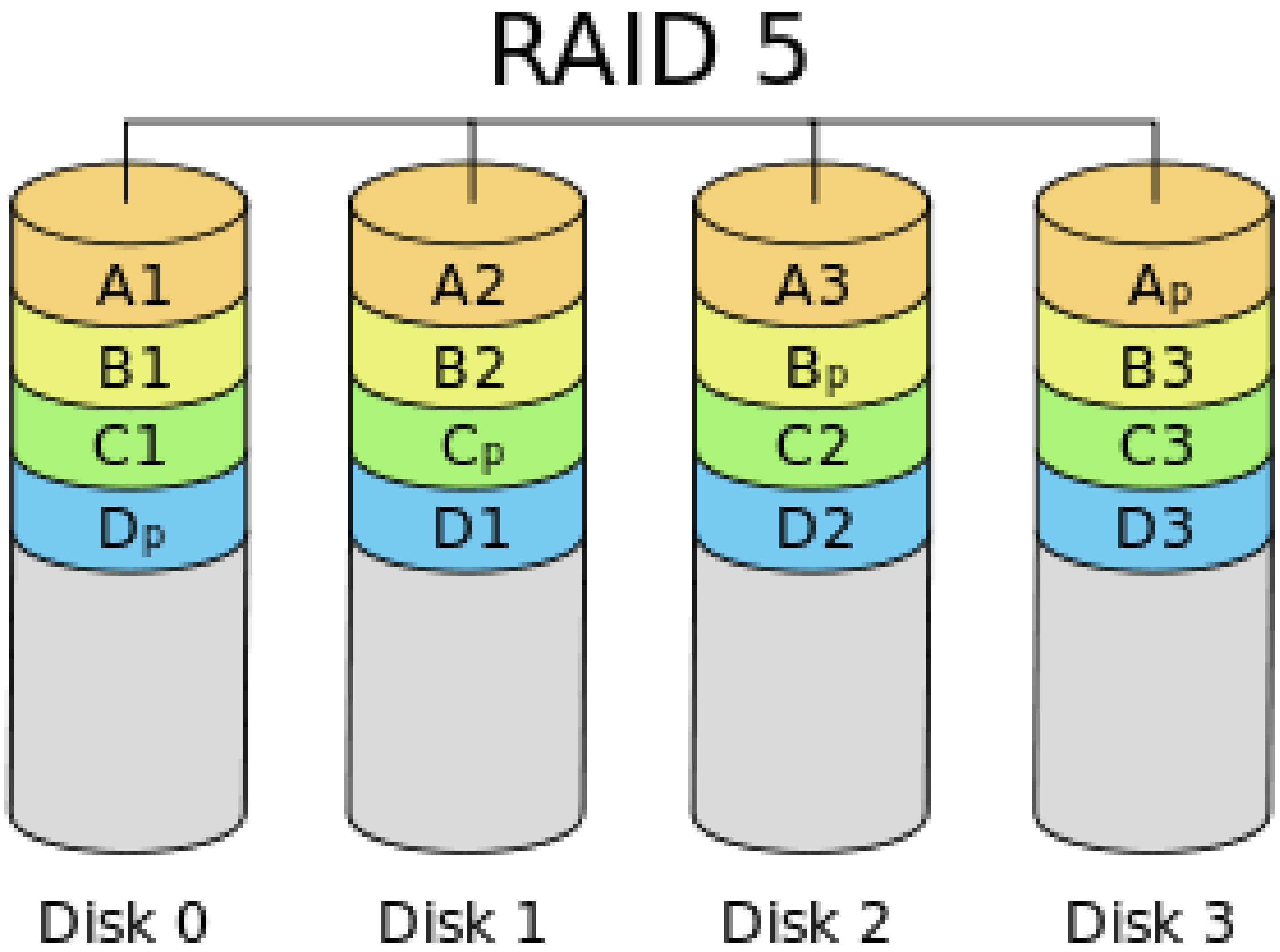


RAID 5: Interleaved Parity

Independent writes possible because of interleaved parity

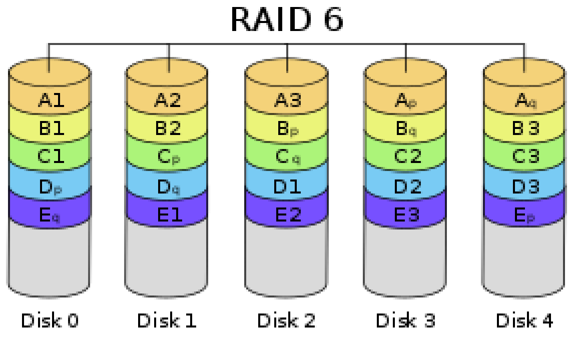


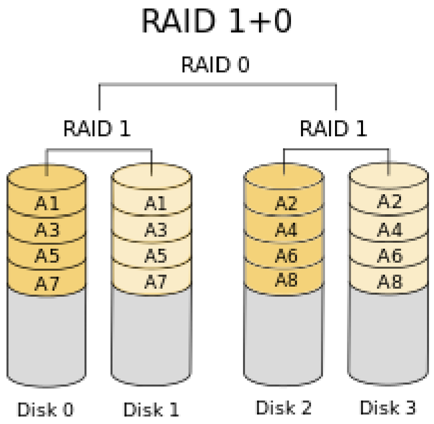
Example: write to D0, D5 uses disks 1, 2, 4, 5


RAID 5
A1
A2
A3
Ap
B1
B2
Bp
B3
C1
Cp
C2
C3
Dp
D1
D2
D3
Disk 0
Disk 1
Disk 2
Disk 3

Newer RAID

- RAID 6 – RAID 5 with more parity blocks, so can tolerate 2 failed disks
- RAID 10 – really just RAID 1 + 0,
- You can setup a RAID array on your computer (if you have enough disks) and choose which version you want!





Modern Use of RAID and ECC (1/2)

- RAID 0 has no redundancy
- RAID 1 is too expensive
- RAID 2 is obsolete due to on-disk ECC
- RAID 3 is not commonly used (bad I/O rates)

- Typical modern code words in DRAM memory systems:
  - 64-bit data blocks (8 B) with 72-bit codes (9 B)
  - $d = 64 \rightarrow p = 7$, +1 for DED

Modern Use of RAID and ECC (2/2)

- Common failure mode is bursts of bit errors, not just 1 or 2
  - Network transmissions, disks, distributed storage
  - Contiguous sequence of bits in which first, last, or any number of intermediate bits are in error
  - Caused by impulse noise or by fading signal strength; effect is greater at higher data rates
- **Other tools:** cyclic redundancy check, Reed-Solomon, Fountain Codes, LaGrange

Summary

- Great Idea: Dependability via Redundancy
  - Reliability: MTTF & Annual Failure Rate
  - Availability: % uptime = MTTF/MTBF
- Memory Errors:
  - Hamming distance 2: Parity for Single Error Detect
  - Hamming distance 3: Single Error Correction Code + encode bit position of error
  - Hamming distance 4: SEC/Double Error Detection
- RAID:
  - Many different flavors, all with their pros/cons
  - Generally: disks are cheap enough that many of them + some smart organization yields safe storage

Bonus Slides

How does 2-bit detection work for Hamming ECC?

Hamming ECC "Cost"

- Space overhead in single error correction code
  - Form $p + d$ bit code word, where p = # parity bits and d = # data bits
- Want the p parity bits to indicate either "no error" or 1-bit error in one of the $p + d$ places
  - Need $2^p \geq p + d + 1$, thus $p \geq \log_2(p + d + 1)$
  - For large d, p approaches $\log_2(d)$

Hamming Single Error Correction, Double Error Detection (SEC/DED)

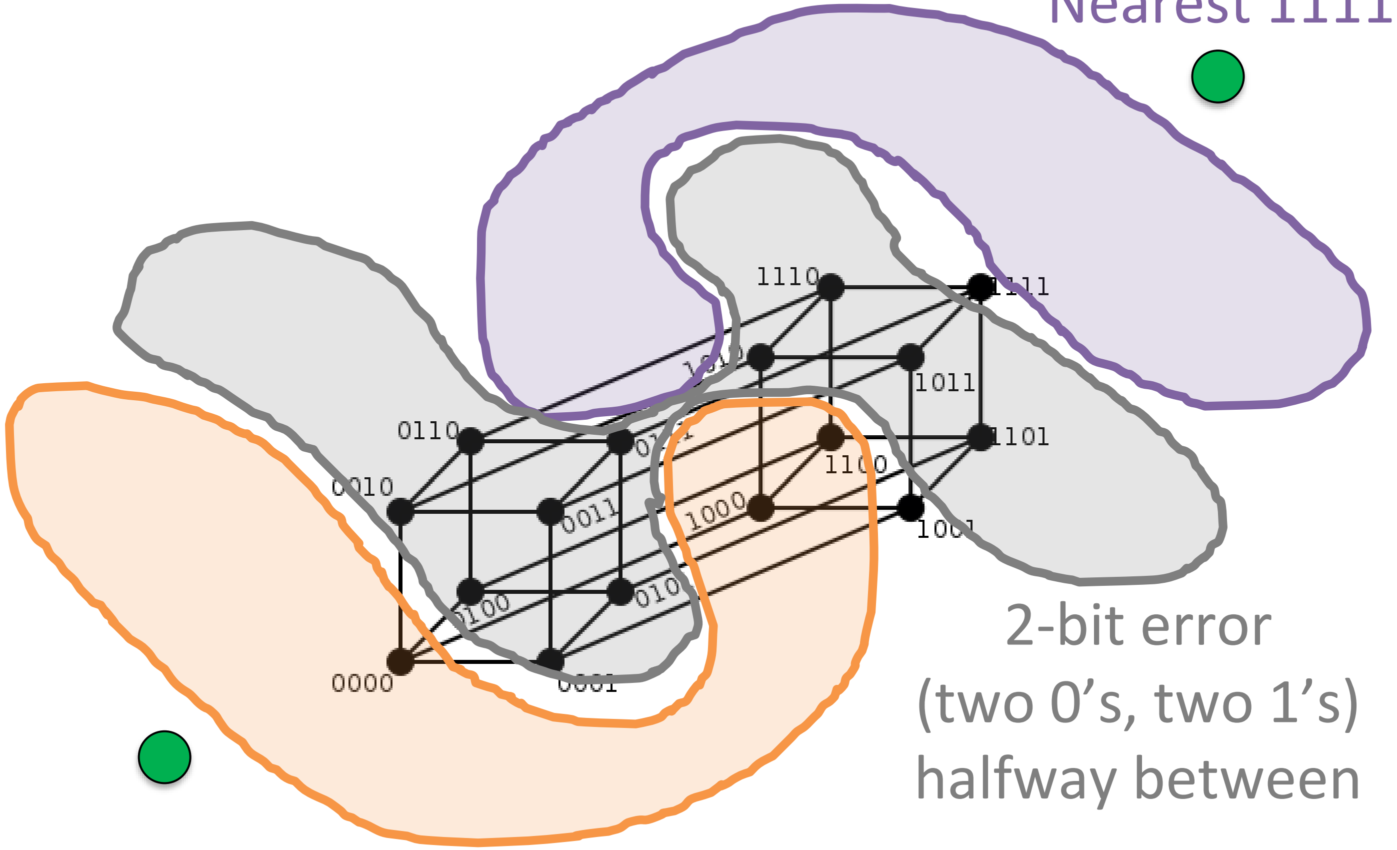
- Adding extra parity bit covering the entire SEC code word provides *double error detection* as well!

1	2	3	4	5	6	7	8
p_1	p_2	d_3	p_4	d_5	d_6	d_7	p_8

- Let H be the position of the incorrect bit we would find from checking p_1, p_2, and p_4 (0 means no error) and let P be parity of complete code word (p's & d's)
  - H=0 P=0, no error
  - H≠0 P=1, correctable single error (P=1 → odd # errors)
  - H≠0 P=0, double error detected (P=0 → even # errors)
  - H=0 P=1, an error occurred in p_8 bit, not in rest of word

SEC/DED: Hamming Distance 4

1-bit error (one 0)
Nearest 1111



2-bit error
(two 0’s, two 1’s)
halfway between

1-bit error (one 1)
Nearest 0000