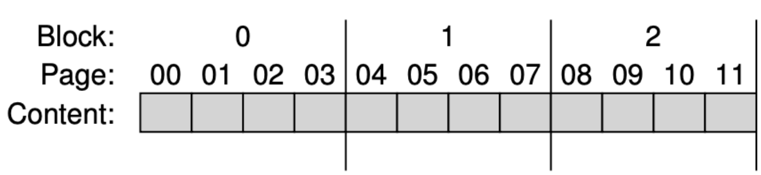
Persistence

- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories
- 40 - File System Implementation
- 41 - Locality and the Fast File System
- 42 - Crash Consistency and Journaling
- 43 - Log-structured (and other) File Systems
- 44 - Flash-based SSD
- 45 - Data Integrity and Protection

SSDs

- non-volatile storage
    - we will assume NAND flash, though rapidly evolving
- terminology
    - a flash chip implements one or more *banks* (or *planes*)
    - a bank contains some number of (erase) blocks
        - might be 128 KB or 256 KB
    - a block contains some number of pages
        - maybe 4 KB



SSDs *operations*

- reads
  - any page can be read, same cost
  - very fast, low microseconds
- erase
  - before writing, a page's entire block must be erased
  - slow, milliseconds
  - needs to be done in advance, usually asynchronously
- program (write)
  - entire page written
  - slower, 100's of usec

- tech constantly evolving, but generally costs follow:
  - read << write << erase

SSDs *example*

- Unrealistically small for example. All start as valid:

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

- If we want to write page 0, must first erase:

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

- Now we can program page 0:

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

- But, but…pages 1-3 are gone….

SSDs *deets*

Device	Read (μs)	Program (μs)	Erase (μs)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

- Reliability
  - no head crashes
  - erasure causes blocks to wear out
  - NANDs leak
    - not good for archival storage

SSDs *from flash*

- SSD contain
  - some amount of RAM for mapping tables
  - FLASH
  - control logic
- flash translation layer (FTL)
  - maps logical blocks to physical pages
  - handles erasures asynchronously
  - modifies mappings as needed
    - because of erasures (we don't write in place)
    - failures
  - wear leveling
- *log-structured…*

SSDs *ftl*

- log structure
  - in storage device
  - also in file system above
  - keeps *mapping table*
- Assume:
  - externally a disk w/ 512-byte sectors
  - client is reading/writing 4k blocks
  - SSD has many 16-KB blocks, w/ 4-KB pages

SSDs *example*

Write a1 to logical block 100, a2 → 101, b1 → 2000, b2 → 2001

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	i	i	i	i	i	i	i	i	i	i	i	i

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	E	E	E	E	i	i	i	i	i	i	i	i

Table: 100 → 0

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a1											
State:	V	E	E	E	i	i	i	i	i	i	i	i

Table: 100 → 0 101 → 1 2000 → 2 2001 → 3

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a1	a2	b1	b2								
State:	V	V	V	V	i	i	i	i	i	i	i	i

Rewrite c1 → 100, c1 → 101

Table: 100 → 4 101 → 5 2000 → 2 2001 → 3

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a1	a2	b1	b2	c1	c2						
State:	V	V	V	V	V	V	E	E	i	i	i	i

Garbage collect

Table: 100 → 4 101 → 5 2000 → 6 2001 → 7

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:					c1	c2	b1	b2				
State:	E	E	E	E	V	V	V	V	i	i	i	i

SSDs *hybrid mapping*

- direct all writes at a few empty blocks (log blocks)
  - log table : per-page mappings (checked first)
  - data table : per-block mapping (checked next)

Say: *a → 1000, b → 1001, c → 1002, d → 1003*

Log Table:
Data Table: 250 → 8

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:									a	b	c	d
State:	i	i	i	i	i	i	i	i	V	V	V	V

Log Table: 1000 → 0 1001 → 1 1002 → 2 1003 → 3
Data Table: 250 → 8

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a'	b'	c'	d'					a	b	c	d
State:	V	V	V	V	i	i	i	i	V	V	V	V

Log Table:
Data Table: 250 → 0

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a'	b'	c'	d'								
State:	V	V	V	V	i	i	i	i	i	i	i	i

But what if re-write 1000, 1001?

Log Table: 1000 → 0 1001 → 1
Data Table: 250 → 8

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a'	b'							a	b	c	d
State:	V	V	i	i	i	i	i	i	V	V	V	V

SSDs *hybrid mapping*

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a'	b'							a	b	c	d
State:	V	V	i	i	i	i	i	i	V	V	V	V

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a'	b'	c	d					a	b	c	d
State:	V	V	i	i	i	i	i	i	V	V	V	V

- This is a *partial merge*
  - could clean up by copying c, d to end of log (block 0)
- Might need to copy from many blocks (*full merge*)
  - assume blocks 0, 4, 8, 12 written
  - would need to write *0,1,2,3* and *4,5,6,7* and….
  - *avoid at all costs* ⟹ *cache only active portion of FTL*

SSDs *conclusion*

- Other issues
  - FTL can be expensive
  - wear leveling
  - cost

- But:

Device	Random Reads (MB/s)	Random Writes (MB/s)	Sequential Reads (MB/s)	Sequential Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Persistence

- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories
- 40 - File System Implementation
- 41 - Locality and the Fast File System
- 42 - Crash Consistency and Journaling
- 43 - Log-structured (and other) File Systems
- 44 - Flash-based SSD
- 45 - Data Integrity and Protection

Data Integrity *how to ensure our data is safe?*

- RAID
  - good, but assumes *fail-stop* failures
  - also need to worry about:
    - latent-sector errors (LSEs)
    - block corruption

	Cheap	Costly
LSEs	9.40%	1.40%
Corruption	0.50%	0.05%

  - over 3 years, 1.5 million drives

Data Integrity *handling latent sector errors*

Latent sector errors:

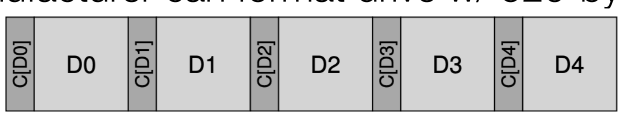
- causes
  - head crashes
  - cosmic rays
- hardware for the win….
  - in-disk error-correcting codes (ECC)
  - ECC fails lead to *disk returning an error* while reading
  - depending on the failure, and the type of ECC, disk might even be able to correct bit errors
- recover using RAID
  - but what if full-disk failure while attempting to recover a sector?
  - *use two parity blocks…*

Data Integrity *block corruption*

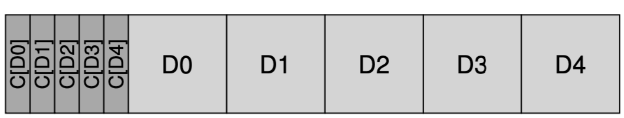
- problem:
  - disk might have incorrect block
  - but not be able to detect it.
- causes
  - buggy firmware might write block to wrong location
  - block corrupted on way to disk
- detection
  - file systems use checksums w/ various speeds and strengths:
    - XOR of all words
    - addition of all words
    - cyclic redundancy check (CRC)
  - but where to store checksums?

Data Integrity *misdirected blocks*

- Where to store checksums?
  - manufacturer can format drive w/ 520-byte sectors



  - consolidate checksums on another sector



- How do we use them?
  - compare checksums when reading, hope for a backup
- What if block b_x stored to sector *y* instead of *x*?
  - checksum would be valid
  - *include x in the checksum*

Distributed Systems

Distributed Systems

- 48 - Communication Basics
- 49 - NFS
- 50 - AFS
- Spore

Communication Basics

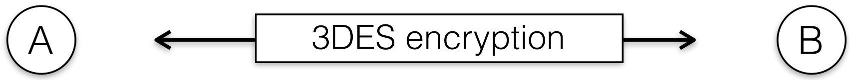
- Building distributed systems
  - all components fail
  - communication fails
  - how to build systems that *rarely* fail from components that do?
- Issues:
  - performance
    - especially with interconnects much slower than buses
  - security
    - systems span users, domains
    - the Internet is scary
  - communication
    - what are the right primitives?
    - what are the right types of applications?

Communication

"progress and correctness of distributed consensus algorithms is impossible to prove in asynchronous environments" - FLP theorem

- communication is fundamentally unreliable
  - packet loss
  - packet corruption
  - packet delays
- maybe don't rely on reliability
  - maybe add encryption to the link!
  - but….

End-to-End Argument *crypto is always good, right?*



- example of *end-to-end argument* says:
  - provided encryption might not be good enough
    - 3DES is ancient, maybe want to use *AES*, *blowfish*
  - provided encryption might be too expensive
    - might not need encryption at all, just adds overhead
  - app semantics might be needed
    - different app messages might have different needs

- but strong semantics in underlying layers *do* help