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 File Systems
- 44 - Flash-based SSD
- 45 - Data Integrity and Protection

LFS Issues *need for a cleaner*

- no overwrite means
  - files, dirs, etc. become fragmented
  - parts of the log no long active:
    - all but most-recent versions of inodes
    - data that has been modified
    - out-of-date imap chunks
- *cleaner process* asynchronously copies live data
  - from *full segments* to clean new segments
  - cleaned segments are empty, can be used again
  - might use this opportunity to segregate by age, activity, etc.
    - segment full of rarely-changing data rarely needs cleaning

LFS *cleaning costs*

- Cleaner
  - read some number of live segments
  - copy live data out into fewer new segments
  - old segments are now free.
- But….write amplification! Let:
  - N : num segments to be cleaned
  - u : percent of these segments that is live
  - write cost (wc): *write amplification* of each new byte

```
write cost  = (#readSegs + #writeLive + #writeNew) / (#writeNew)
            = (N + N*u + (N*(1-u)) / (N * (1-u))
            = 2/(1-u)
```

  - if utilization low, say 10%: *wc = 2.22*
  - if utilization high, say 90%: *wc = 20.00*

LFS *cleaner notes*

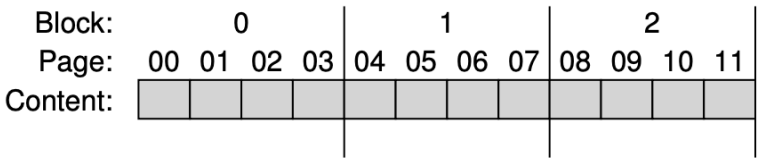
- advantages
  - asynchronous
  - can be done in bulk, i.e. fast
- opportunities
  - *older data less likely to be modified than new data*
  - can segregate data based on age for cleaner writes
- implemented on bare disk
  - log chunk to be written to disk is many pages long
  - LFS can report consistency check of all blocks back to OS
  - LFS guarantees that pg *i* written correctly before pg *i+1*

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

- Pages can be in one of several states:

		`iiii`	*Initial: pages in block are invalid* (`i`)
Erase()	→	`EEEE`	*State of pages in block set to erased* (`E`)
Program(0)	→	`VEEE`	*Program page 0; state set to valid* (`V`)
Program(0)	→	**error**	*Cannot re-program page after programming*
Program(1)	→	`VVEE`	*Program page 1*
Erase()	→	`EEEE`	*Contents erased; all pages programmable*

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…*

Why not direct mapped?

- Problems if FTL mapping LBA *N* to physical page *N*
  - performance
    - write to *N* requires:
      - read block
      - erase block
      - write block
  - reliability
    - hot spots in program cause some blocks to fail
    - no wear leveling

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 FS 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

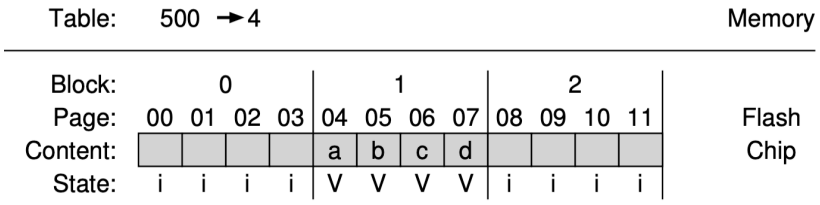
Table: 100 → 4 101 → 5 2000 → 6 2001 → 7

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

logical block (FS) to physical page *mapping table*

SSDs *mapping table size*

- assume 1TB drive:
  - if assume 4 bytes / 4k block, 1GB for table, in memory!
- block approach:
  - reduces size by factor $\frac{size_{block}}{size_{page}}$, but more complicated
  - assume page-level mappings $2000 \rightarrow 4$, $2001 \rightarrow 5$, $2002 \rightarrow 6$, $2003 \rightarrow 7$:
    - all have chunk *500*
    - offsets 0, 1, 2, 3
  - LBA translation:
    - get chunk from top-level bits
    - add offset to chunk —> page mapping

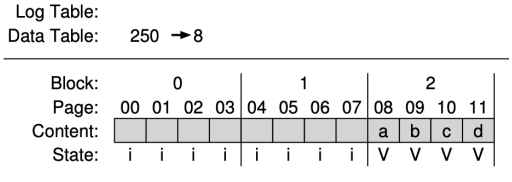


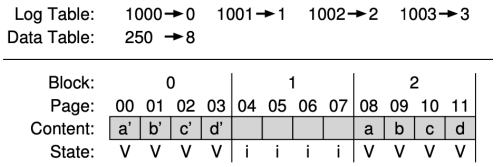
  - reads easy, but writes require prior reads and erases

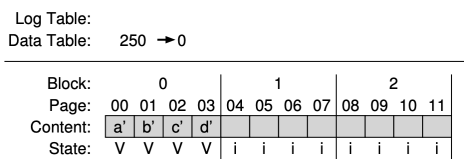
SSDs *hybrid mapping*

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

Say: $a \rightarrow 1000$, $b \rightarrow 1001$, $c \rightarrow 1002$, $d \rightarrow 1003$







switch merge

But what if re-write 1000, 1001?