SOLID-STATE DRIVES

Kai Mast

CS 537

Fall 2022

FLASH DISKS: SOLID STATE STORAGE

- A NAND flash block is a grid of cells
  - Single Level Cell (SLC) = 1 bit per cell (faster, more reliable)
  - Multi Level Cell (MLC) = 2 bits per cell (slower, less reliable)
  - Triple Level Cell (TLC) = 4 bits per cell (even more so)

- Cells are grouped into pages
  - e.g., 4kB in size

- Pages are grouped into blocks
  - e.g., 256 KB in size

Solid-State Storage Devices (SSDs)

Unlike hard drives, SSDs have no mechanical parts

- SSDs use transistors (just like DRAM), but SSD data persists when the power goes out
- NAND-based flash is the most popular technology, so we'll focus on it

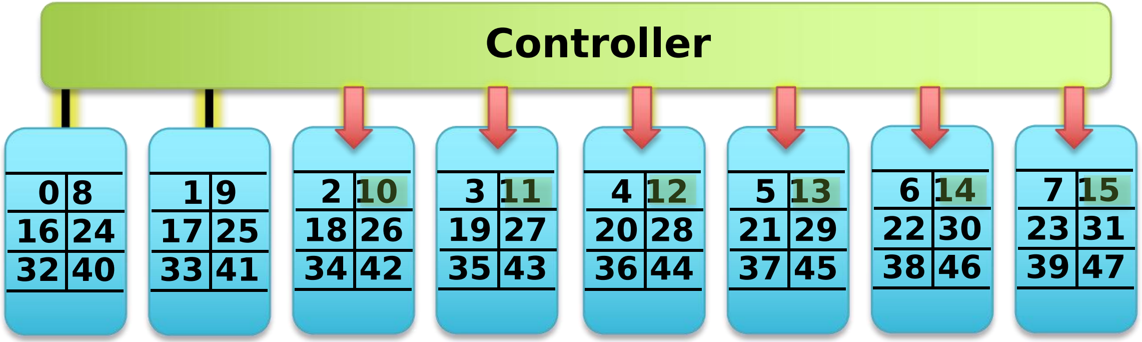
High-level takeaways

1. SSDs have a higher $/bit than hard drives, but better performance (no mechanical delays!)
2. SSDs handle writes in a strange way; this has implications for file system design

INTERNAL PARALLELISM OF SSDS

Block addresses striped across flash packages like RAID 0

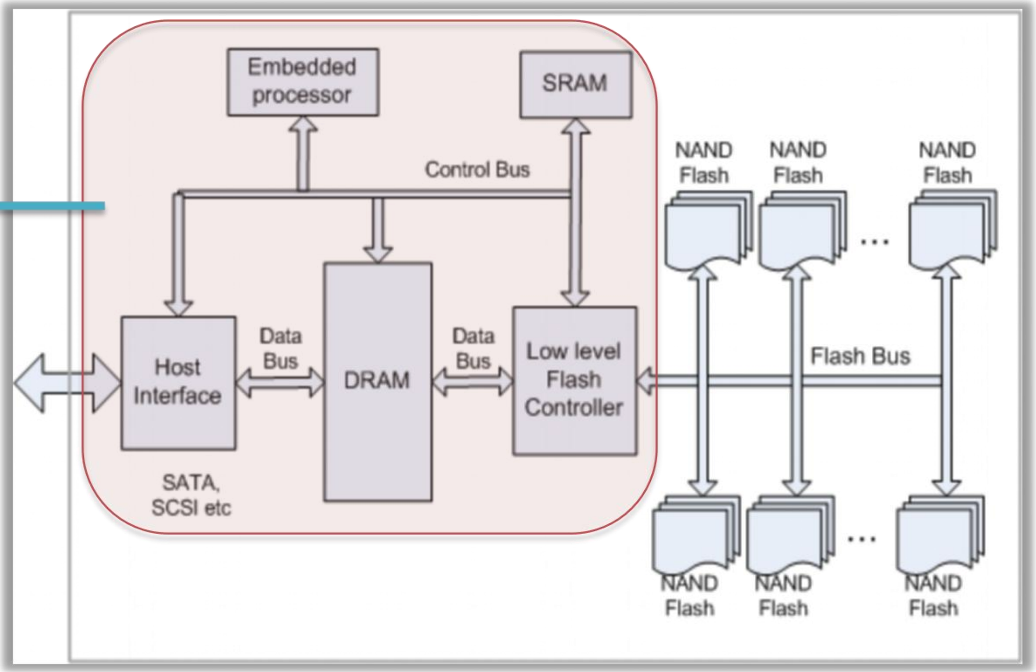
- Single request can span multiple chips
- Natural load balancing



BACKGROUND: SSD STRUCTURE

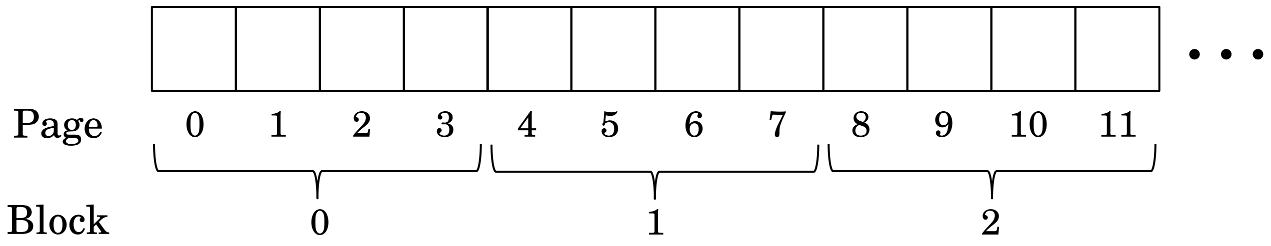
Simplified block diagram of an SSD

Flash Translation Layer (FTL)

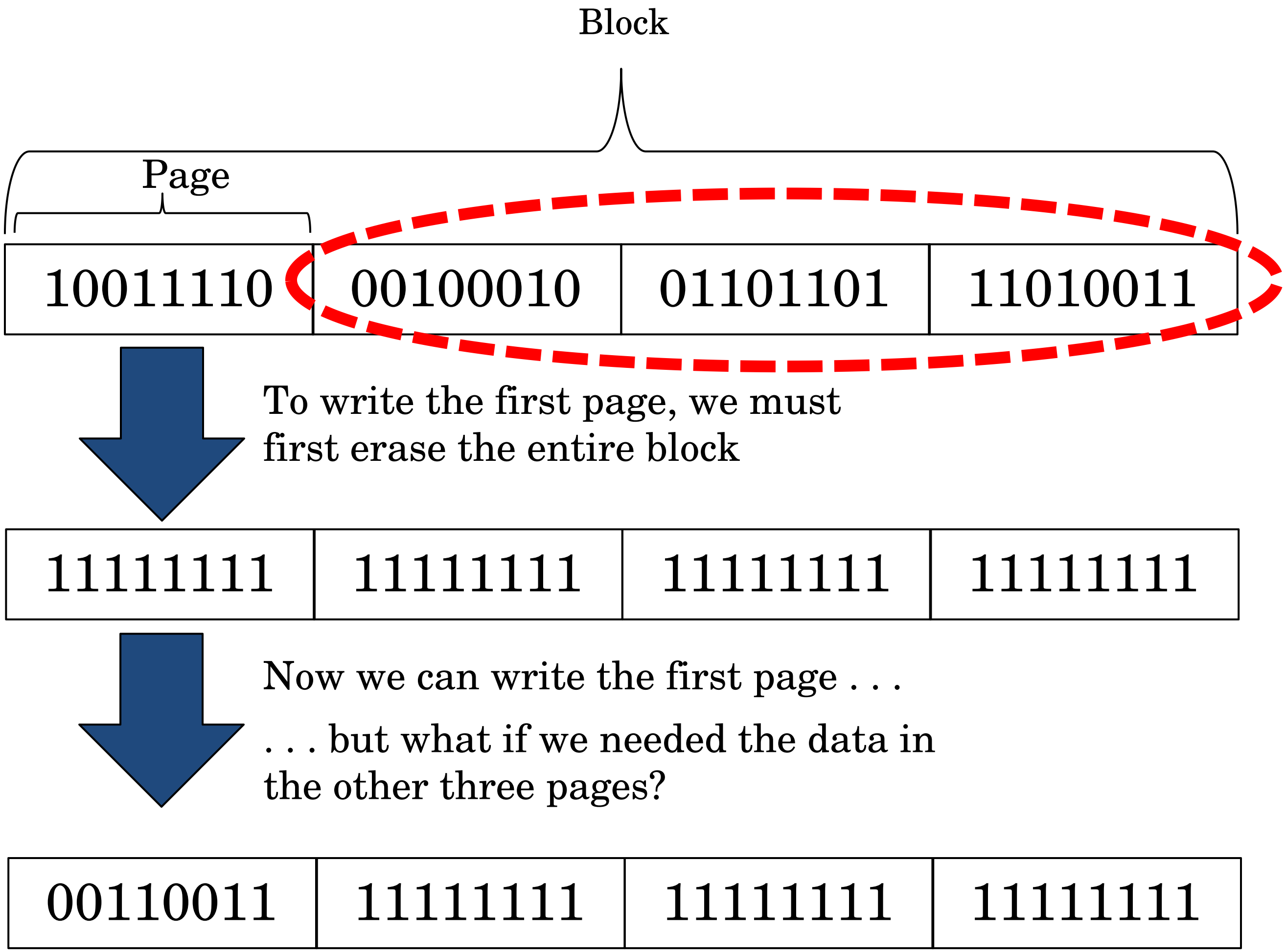


SOLID-STATE STORAGE DEVICES (SSDS)

- An SSD contains blocks made of pages
  - A page is a few KB in size (e.g., 4 KB)
  - A block contains several pages, is usually 128 KB or 256 KB



- To write a single page, you must erase the entire block first
- A block is likely to fail after a certain number of erases (~1000 for slowest-but-highest-density flash, ~100,000 for fastest-but-lowest-density flash)


Block
Page
10011110
00100010
01101101
11010011
To write the first page, we must first erase the entire block
11111111
11111111
11111111
11111111
Now we can write the first page . . .
. . . but what if we needed the data in the other three pages?
00110011
11111111
11111111
11111111

SSD API

Reading

- read: retrieve contents of a single page

Writing

- erase: reset and entire block
- program: write contents of one page

SSD OPERATIONS (LATENCY)

- Read a page: Retrieve contents of entire page (e.g., 4 KB)
  - Cost is 25—75 microseconds
  - Cost is independent of page number, prior request offsets
- Erase a block: Resets each page in the block to all 1s
  - Cost is 1.5—4.5 milliseconds
  - Much more expensive than reading!
  - Allows each page to be written
- Program (i.e., write) a page: Change selected 1s to 0s
  - Cost is 200—1400 microseconds
  - Faster than erasing a block, but slower than reading a page

Hard disk: 4—15ms avg. seek latency
2—6ms avg. rotational latency

FLASH TRANSLATION LAYER (FTL)

Goal 1: Translate reads/writes to logical blocks into reads/erases/programs on physical pages+blocks

- Allows SSDs to export the simple "block interface" that hard disks have traditionally exported
- Hides write-induced copying and garbage collection from applications

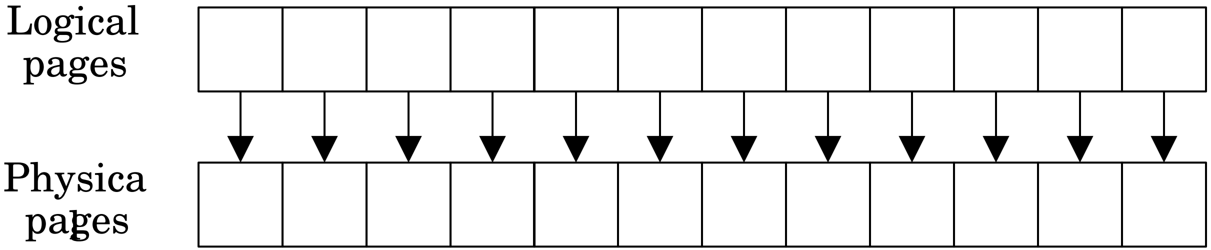
Goal 2: Reduce write amplification (i.e., the amount of extra copying needed to deal with block-level erases)

Goal 3: Implement wear leveling (i.e., distribute writes equally to all blocks, to avoid fast failures of a "hot" block)

FTL is typically implemented in hardware in the SSD, but is implemented in software for some SSDs

FTL Approach #1: Direct Mapping

- Have a 1-1 correspondence between logical pages and physical pages



- Reading a page is straightforward
- Writing a page is trickier:
  - Read the entire physical block into memory
  - Update the relevant page in the in-memory block
  - Erase the entire physical block
  - Program the entire physical block using the new block value

DIRECT MAPPING: LIMITATIONS

Problem 1: Write amplification

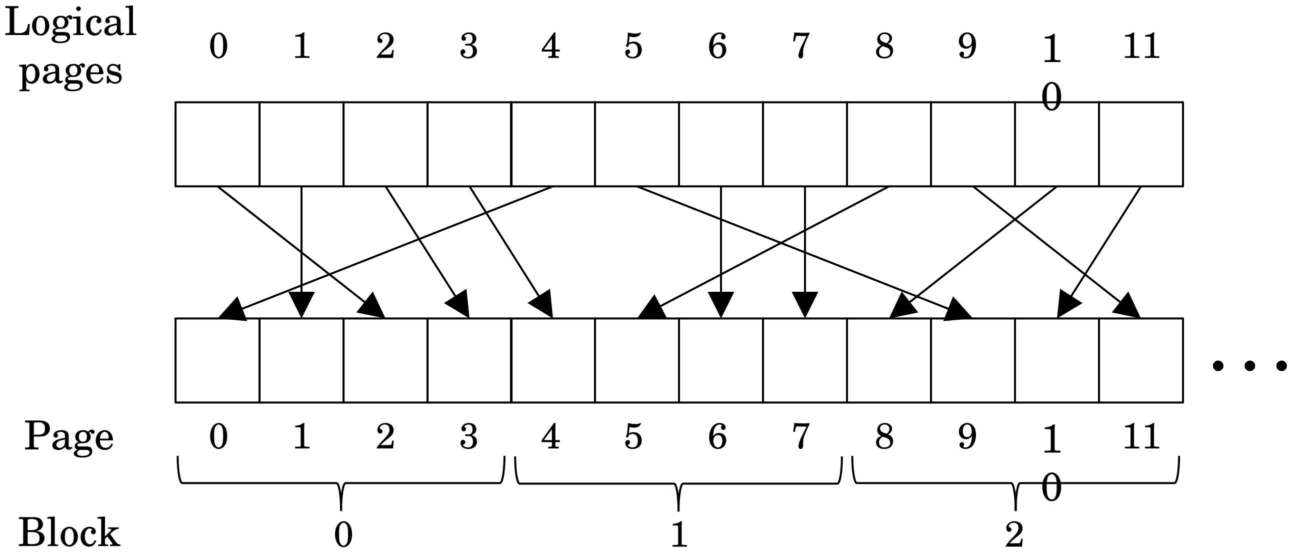
- Writing a single page requires reading and writing an entire block

Problem 2: Poor reliability

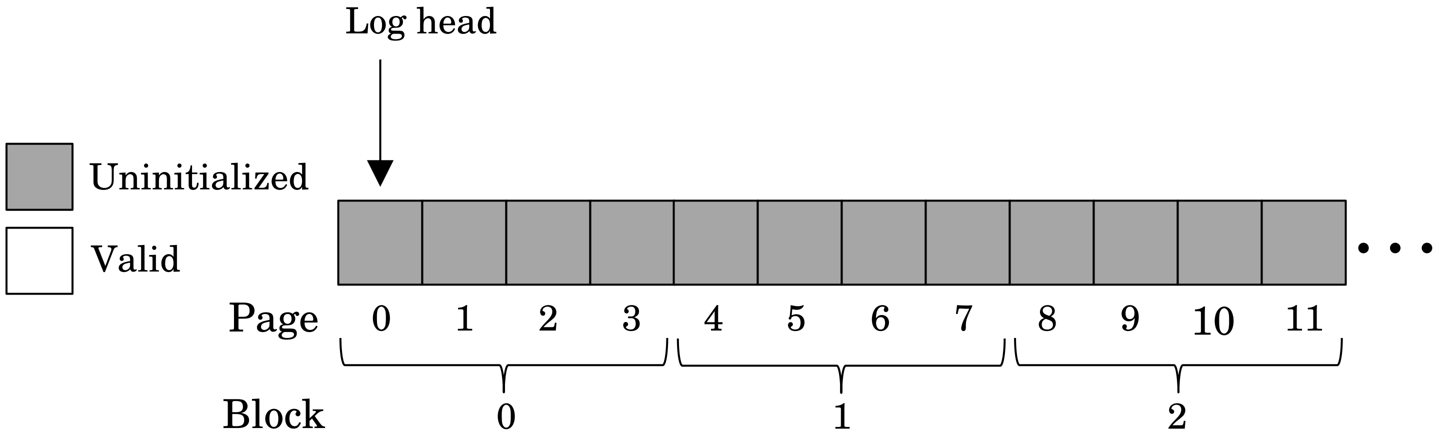
- If the same logical block is repeatedly written, its physical block will quickly fail
- Particularly unfortunate for logical metadata blocks
- If failure happens between erase + rewrite => data loss

FTL Approach #2: Log-Based Mapping

- Basic idea: Treat the physical blocks like a log
  - Send data in each page-to-write to the end of the log
  - Maintain a mapping between logical pages and the corresponding physical pages in the SSD



Logical-to-physical map



write(page=92, data=w0)

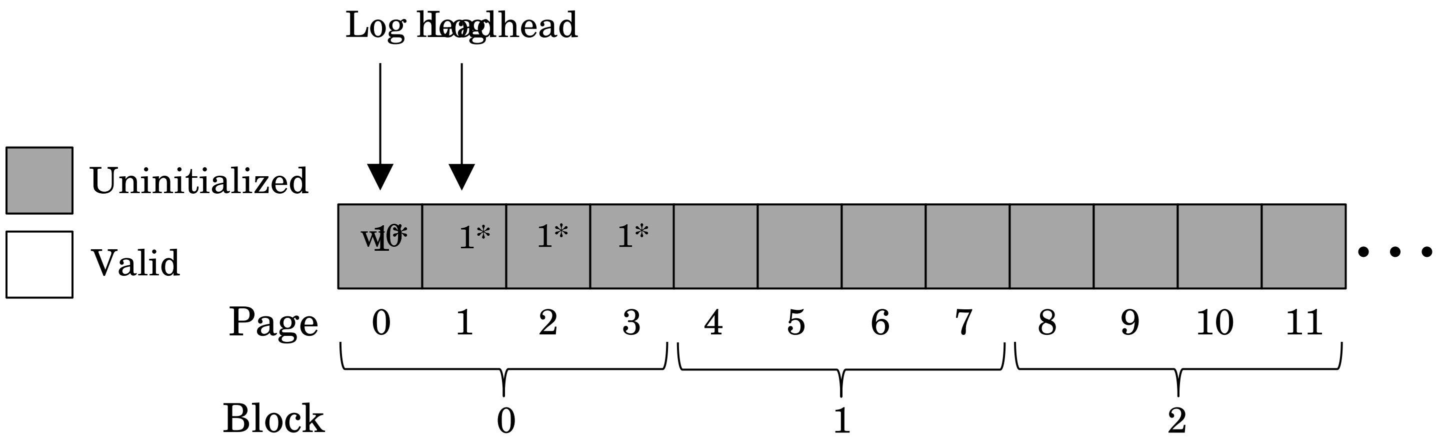
→ erase(block0)

→ program(page0, w0)

→ logHead++

Logical-to-physical map

92 --> 0



write(page=17, data=w1)

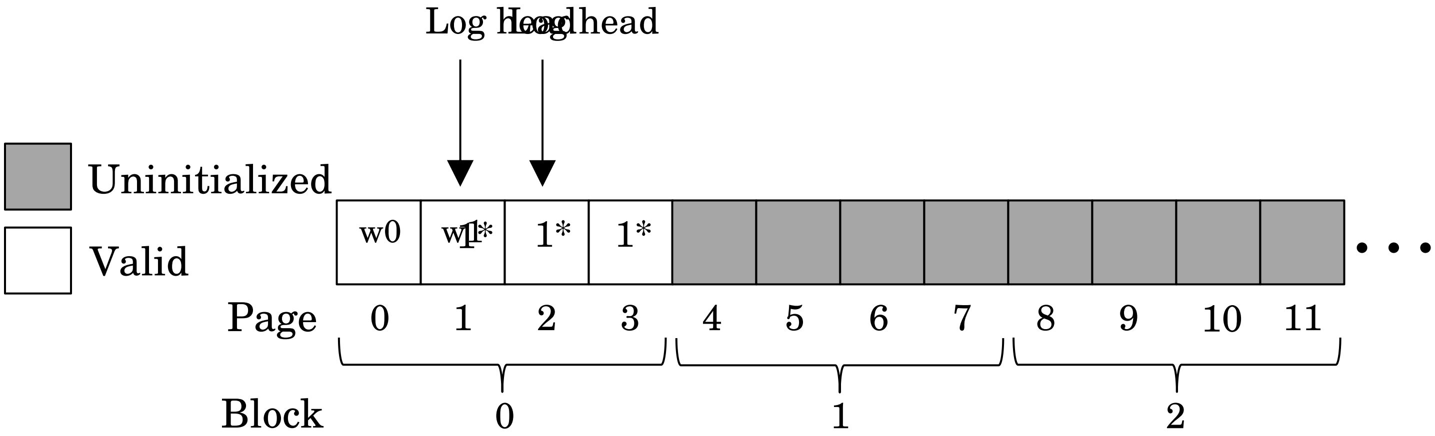
→ program(page1, w1)

→ logHead++

<u>Logical-to-physical map</u>

92 --> 0

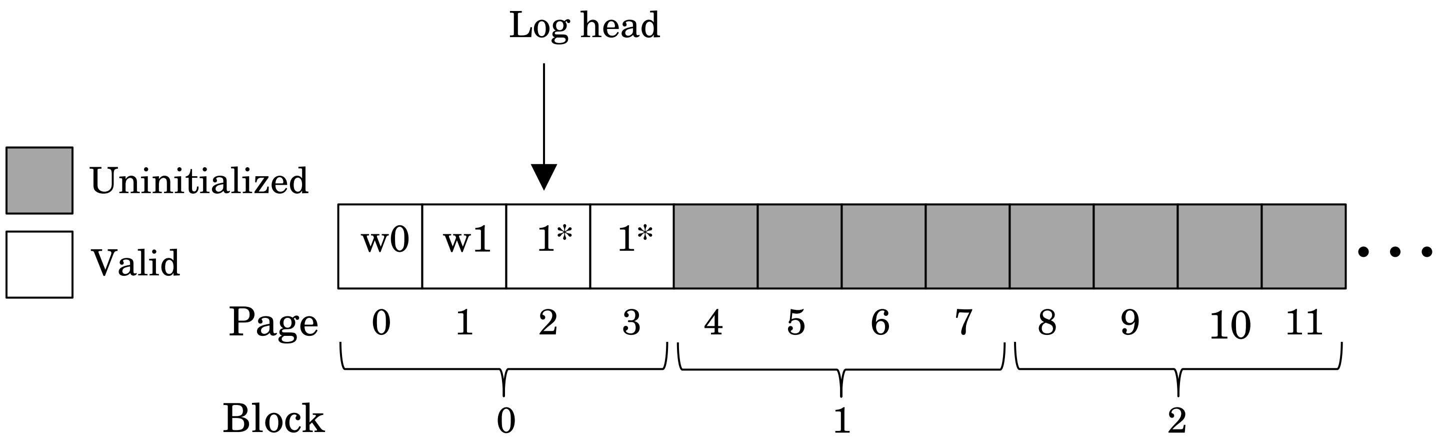
17 --> 1



Logical-to-physical map
92 --> 0
17 --> 1

Advantages w.r.t. direct mapping

- Avoids expensive read-modify-write behavior
- Better wear leveling: writes get spread across pages, even if there is spatial locality in writes at logical level

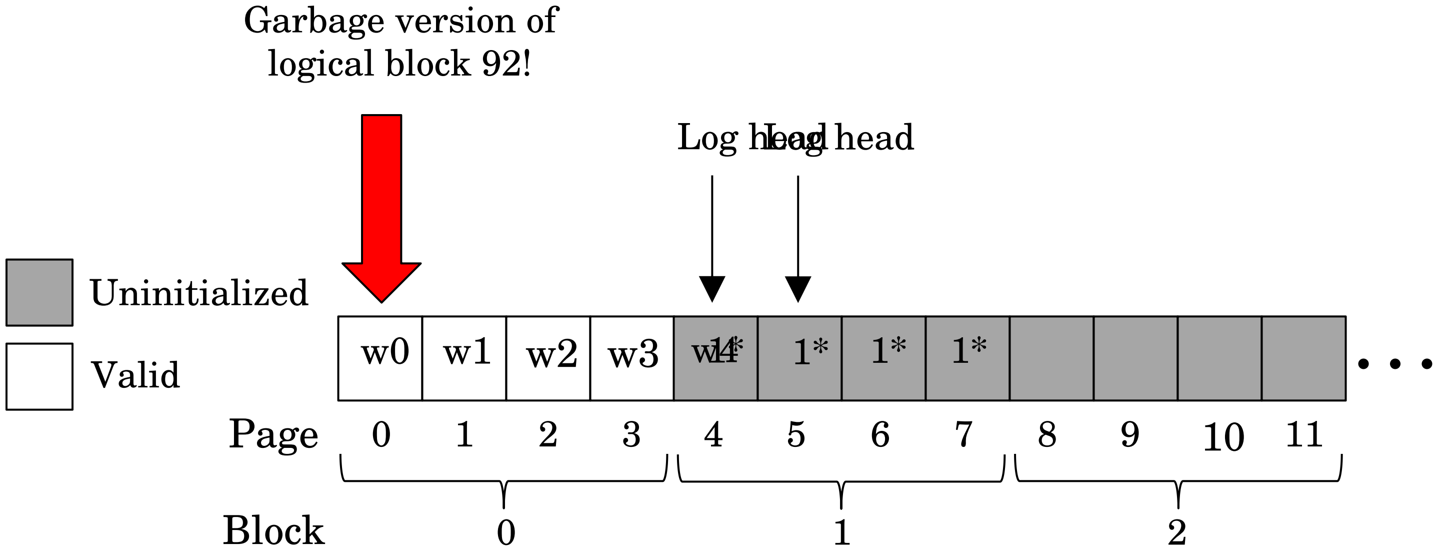


write(page=92, data=w4)
- erase(block1)
- program(page4, w4)
- logHead++

Logical-to-physical map

~~92 --> 0~~
17 --> 1
33 --> 2
68 --> 3

92 --> 4



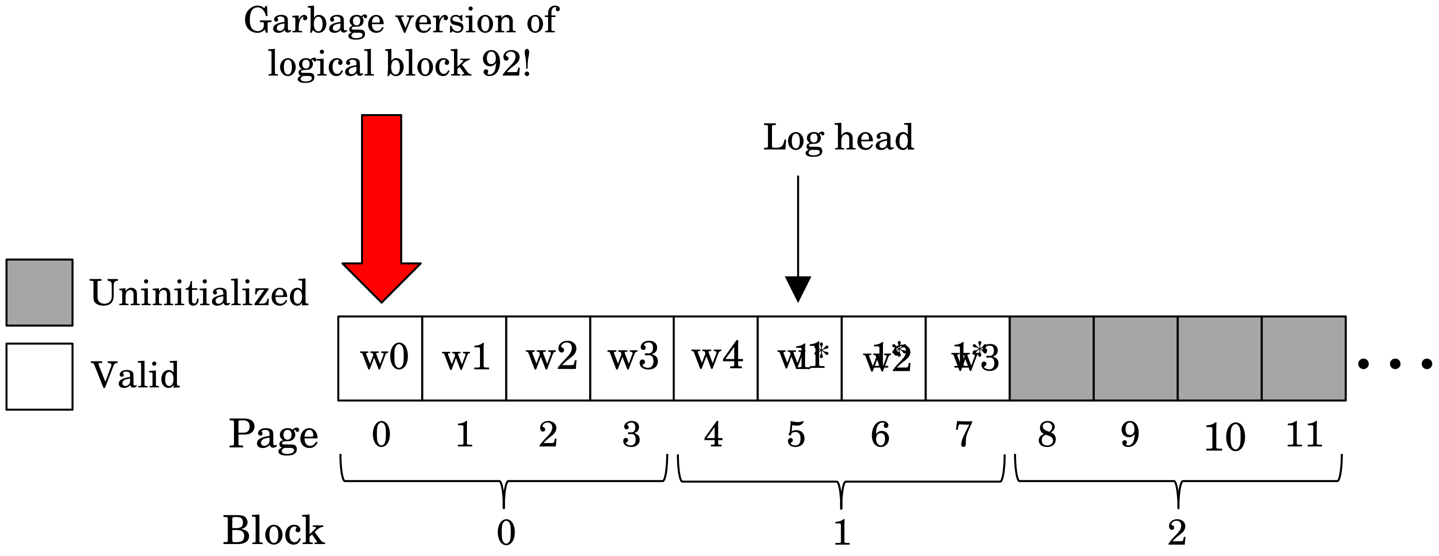
- Read all pages in physical block 0
- Write out the second, third, and fourth pages to the end of the log
- Update logical-to-physical map

Logical-to-physical map

~~92 --> 0~~
~~17 --> 1~~
33 --> 2
68 --> 3

At some point, FTL must:

92 --> 4



BACKGROUND TASKS

Garbage collection requires extra read+write traffic

Over-provisioning makes GC less painful

- SSD exposes a logical page space that is smaller than the physical page space
- By keeping extra, "hidden" pages around, the SSD tries to defer GC to a background task (thus removing GC from critical path of a write)

SSD will occasionally shuffle live (i.e., non-garbage) blocks that never get overwritten

- Enforces wear leveling

MAPPING TABLE SIZE

Example

- Disk size: 1TB (=1024GB)
- Page size: 4Kb
- Small-ish entry size: 4 bytes

How big does the mapping table get?

- ~268 Million Entries
- ~1Gb of FTL data

HYBRID MAPPING

Idea: Keep fine-grained mapping for recent data and coarse grained mapping for older data

Log-mapping: Per-page mapping for most recent log entries
Data-mapping: Per-block mapping for all other data

Need to merge data for log-mapping to data-mapping

- If an entire block was written in order we can move the block as is (**switch merge**)
- Otherwise, we have to merge with an old block (**partial merge**) or merge with multiple blocks (**full merge**)

CRASH RECOVERY

- Mapping table is stored in memory
- What do we do if there is a crash?

Possible Solution:

- Keep some extra out-of-band data with every page
- Scan the entire disk (similar to fsck) on restart

Faster approaches based on checkpointing and logging exist

SSDS VERSUS HARD DRIVES (THROUGHPUT)

Device	Random Reads (MB/s)	Random Writes (MB/s)	Sequential Reads (MB/s)	Sequential Writes (MB/s)
Samsung 960 Evo Plus SSD	85.5	244	2664	2508
Crucial BX100 SSD	24.5	73.5	466	392
Samsung 840 EVO SSD	44.1	112	502	494
Seagate Savio 15K.3 HD	2	2	223	223

Dollars per storage bit: Hard drives are 10x cheaper!

Source: "Flash-based SSDs" chapter of "Operating Systems: Three Easy Pieces" by the Arpaci-Dusseaus.

SSD SUMMARY

- Most local storage will be solid state
  - Faster, cheap enough
- Best for fast random access
- Relies on FTL for fast writes
  - Sequential and random writes closer cost
  - Garbage collection more expensive with random writes

PRESISTENCE REVISION

HDD

Overlap I/O operations and CPU instructions whenever possible

- Use interrupts, DMA

Disks: Linear array of sectors (512 bytes – read/write atomically)

- IO Time = Seek + Rotation + Transfer
- Sequential bandwidth >> Random

Disk Scheduling

- FCFS
- SSTF (shortest seek time first)
- Dealing with starvation
  - Elevator (Scan) or Circular Scan (C-Scan)

RAID LEVEL COMPARISONS

	Reliability	Capacity	Read Latency	Write Latency
RAID-0	0	C*N	D	D
RAID-1	1	C*N/2	D	D
RAID-4	1	(N-1) * C	D	2D
RAID-5	1	(N-1) * C	D	2D

RAID LEVEL COMPARISONS

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	N * S	N * S	N * R	N * R
RAID-1	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-4	(N-1)*S	(N-1)*S	(N-1)*R	R/2
RAID-5	(N-1)*S	(N-1)*S	N * R	N/4 * R

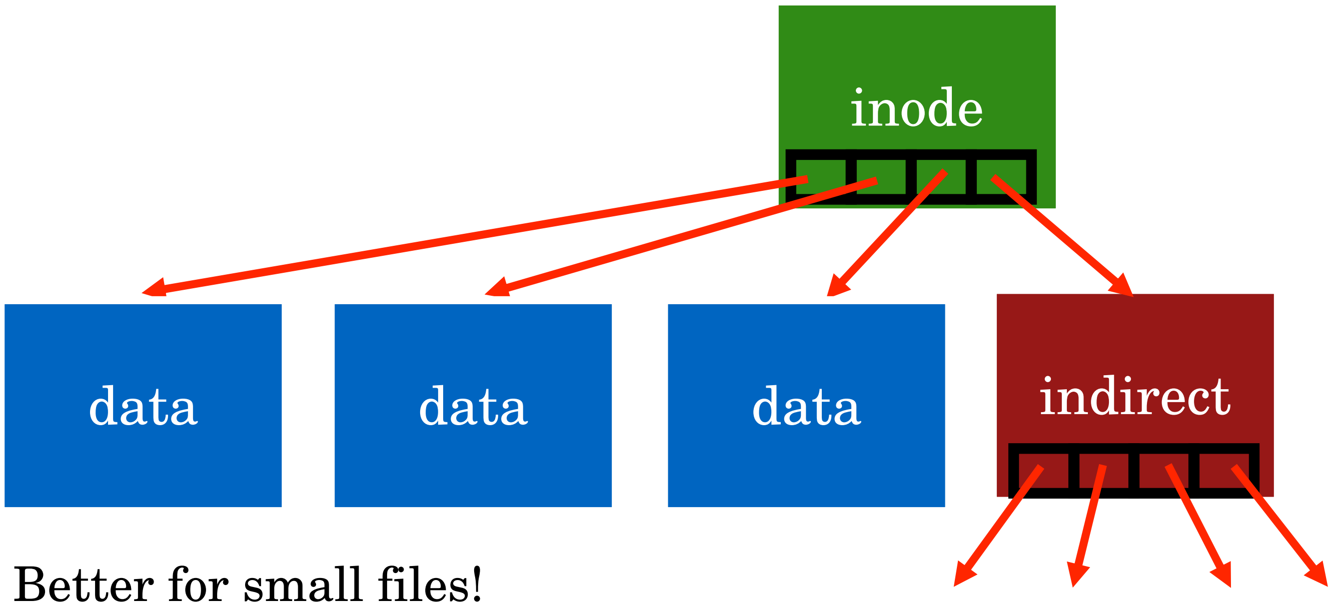
RAID-5 is strictly better than RAID-4

RAID-0 is always fastest and has best capacity (but at cost of reliability)

RAID-1 better than RAID-5 for random workloads

RAID-5 better than RAID-1 for sequential workloads

INODES



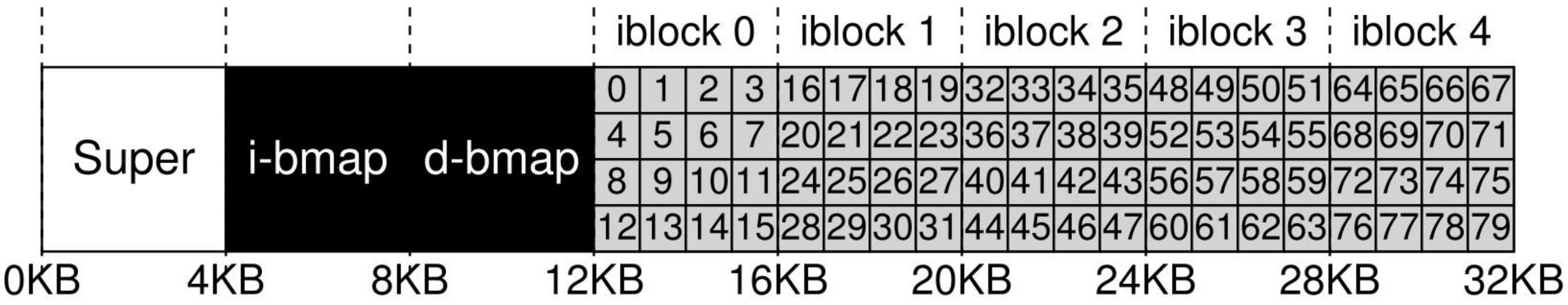
Better for small files!
How to handle even larger files?

Double indirect blocks
Triple indirect blocks

FILE SYSTEM ORGANIZATION

Very simple file system (vsfs) from the textbook

i-node table



Superblock: Parameters of the file system (e.g., how many inodes)

i-bitmap: Which inodes are in use?

d-bitmap: Which data blocks are in use?
(Data blocks are not shown on the above figure)

FS-API

Using multiple types of names provides convenience and efficiency
- inodes
- path names
- file descriptors

Directories: Special files that hold the name to inode-number mapping

Two types of linking
- Softlinks – Point to second path name having different inode number
- Hardlinks – Both path names use same inode number

Special calls (fsync, rename) let developers communicate requirements to file system