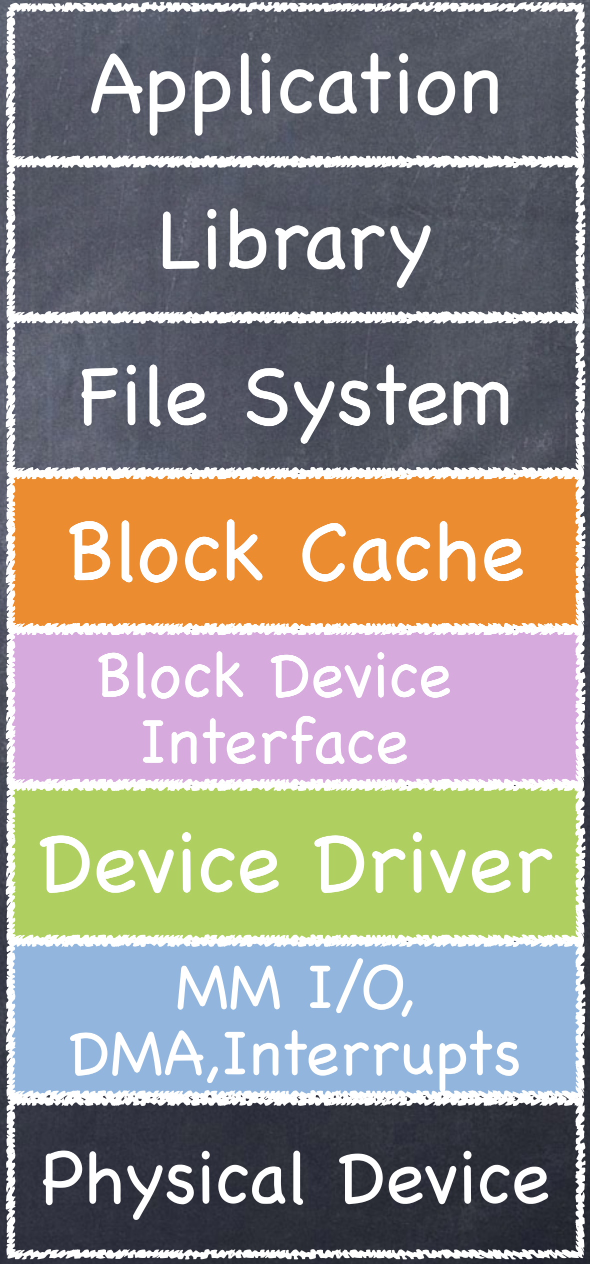
Storage stack:
File Systems:
Storing Files

Recall: The Storage Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches blocks recently read from disk
  - Buffers recently written blocks
  - Single interface to many devices, allows data to be read/written in fixed sized blocks
  - Translates OS abstractions and hw specific details of I/O devices
  - Control registers, bulk data transfer, OS notifications



Application

Library

File System

Block Cache

Block Device Interface

Device Driver

MM I/O, DMA,Interrupts

Physical Device

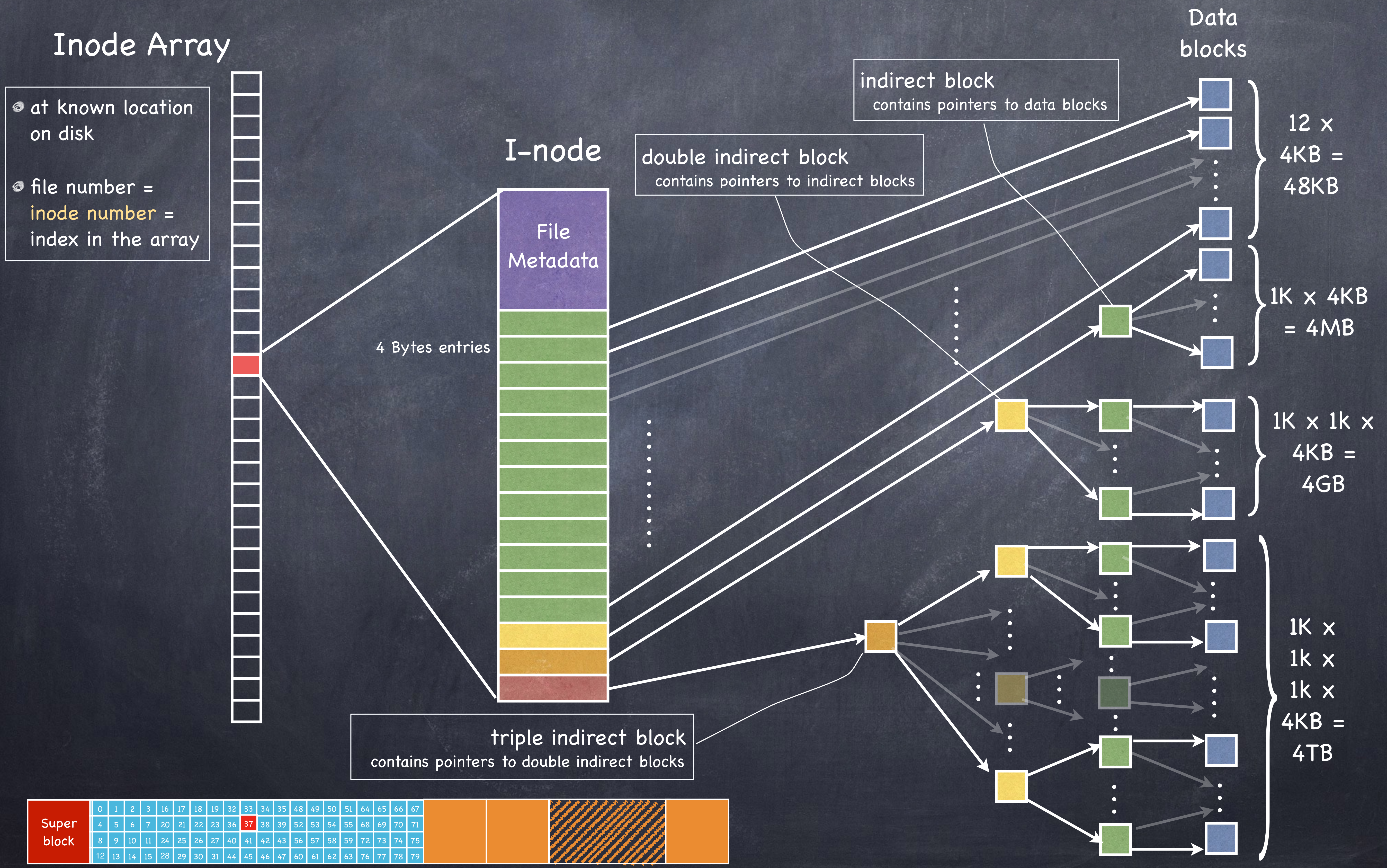
Recall: Storing Files

- Files can be allocated in different ways
  - Contiguous allocation
    - all bytes together, in order
  - Linked Structure
    - Each points to the next block
  - Indexed Structure
    - Index block, pointing to many other blocks
- Which is best?
  - For sequential access? Random access?
  - Large files? Small files? Mixed?

Recall: File structure

- Each file is a fixed, asymmetric tree, with fixed size data blocks (e.g. 4KB) as its leaves
- The root of the tree is the file's inode, containing
  - metadata (more about it later)
  - a set of 15 pointers
    - first 12 point to data blocks
    - last three point to intermediate blocks, themselves containing pointers…
      - #13: pointer to a block containing pointers to data blocks
      - #14: double indirect pointer
      - #15: triple indirect pointer (!)

Recall: Multilevel index



Inode Array

- at known location on disk
- file number = inode number = index in the array

I-node

indirect block
contains pointers to data blocks

double indirect block
contains pointers to indirect blocks

triple indirect block
contains pointers to double indirect blocks

Super block	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67
	4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71
	8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75
	12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79

Crash Consistency

Caching and Consistency

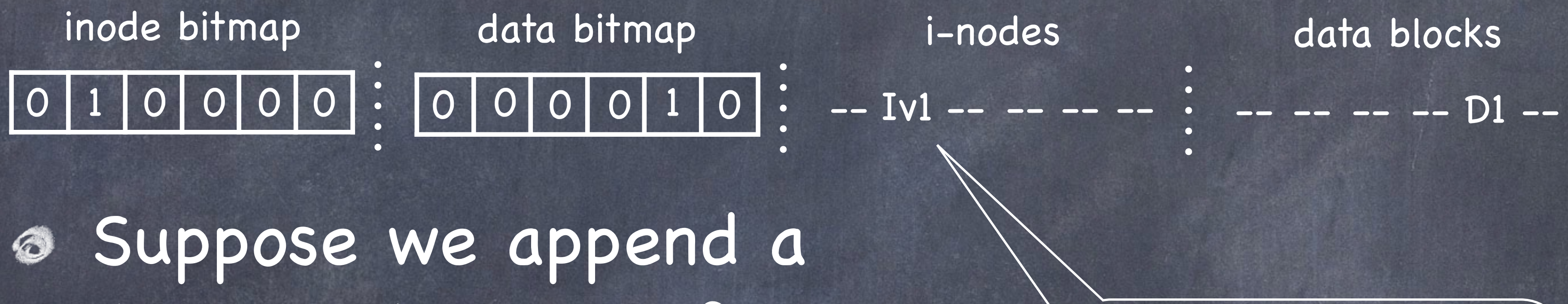
- File systems maintain many data structures
  - Bitmap of free blocks and inodes
  - Directories
  - Inodes
  - Data blocks
- Data structures cached for performance
  - works great for read operations…
  - …but what about writes?

Caching and consistency

- File systems maintain many data structures
  - Bitmap of free blocks and inodes
  - Directories
  - Inodes
  - Data blocks
- Data structures cached for performance
  - works great for read operations...
  - ...but what about writes?
- Write-back caches
  - delay writes: higher performance at the cost of potential inconsistencies
- Write-through caches
  - write synchronously but poor performance (fsync)
    - do we get consistency at least?

Example: a tiny ext2

- 6 blocks, 6 inodes

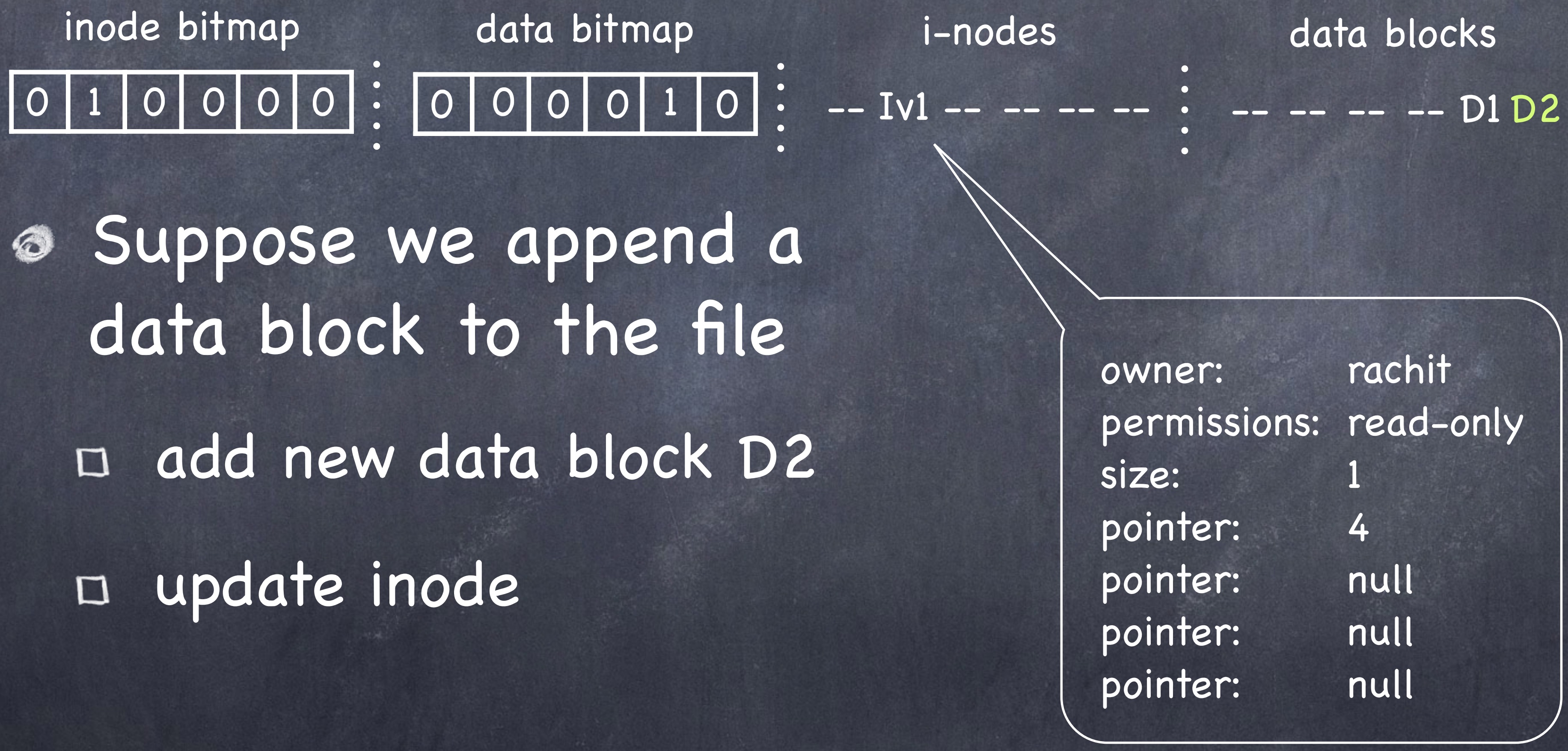


- Suppose we append a data block to the file
  - add new data block D2

owner:	rachit
permissions:	read-only
size:	1
pointer:	4
pointer:	null
pointer:	null
pointer:	null

Example: a tiny ext2

- 6 blocks, 6 inodes



owner:	rachit
permissions:	read-only
size:	1
pointer:	4
pointer:	null
pointer:	null
pointer:	null

- Suppose we append a data block to the file
  - ☐ add new data block D2
  - ☐ update inode

Example: a tiny ext2

- 6 blocks, 6 inodes

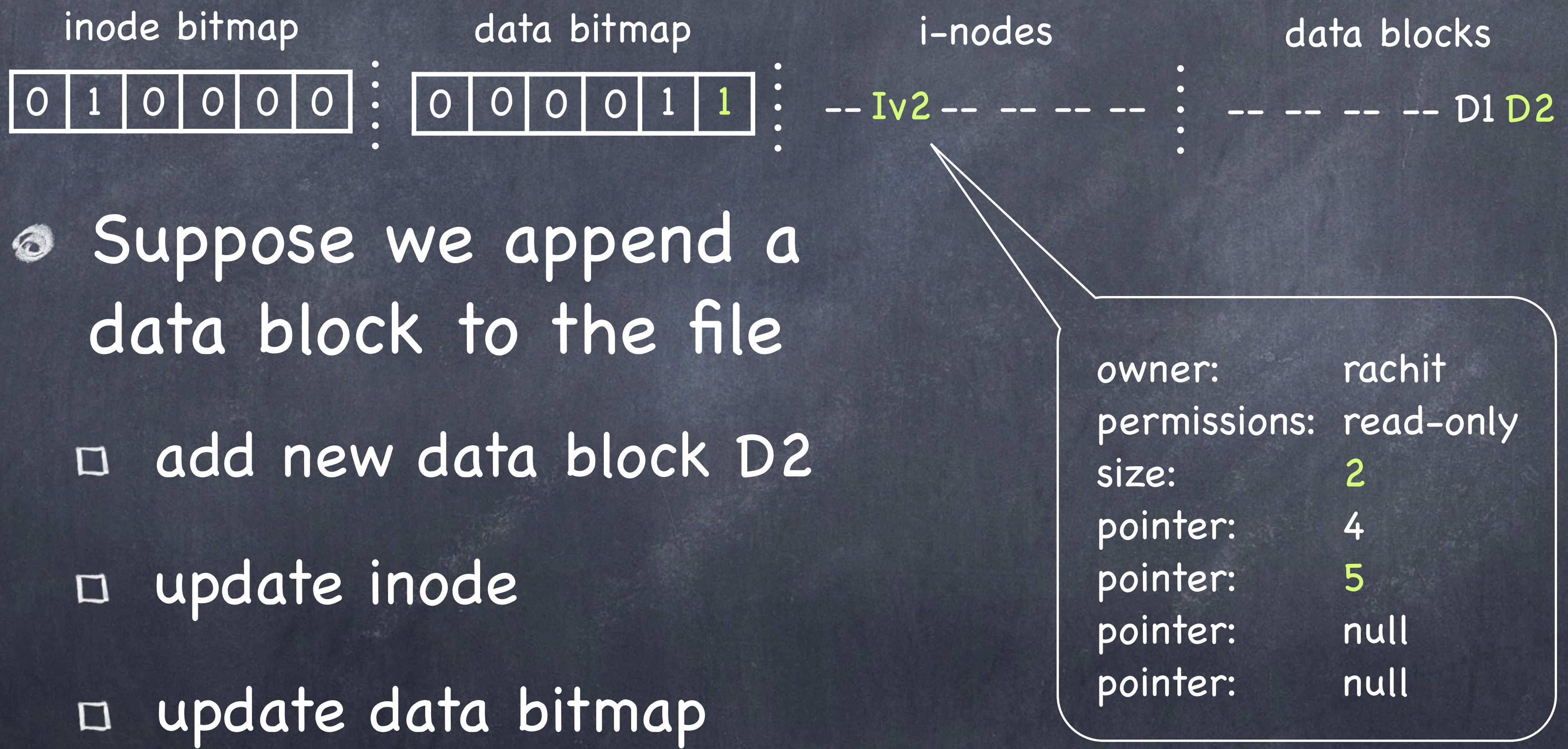
inode bitmap	data bitmap	i-nodes	data blocks
0 | 1 | 0 | 0 | 0 | 0	0 | 0 | 0 | 0 | 1 | 0	-- Iv2 -- -- -- --	-- -- -- -- D1 D2

- Suppose we append a data block to the file
  - ☐ add new data block D2
  - ☐ update inode
  - ☐ update data bitmap

owner:	rachit
permissions:	read-only
size:	2
pointer:	4
pointer:	5
pointer:	null
pointer:	null

Example: a tiny ext2

- 6 blocks, 6 inodes



inode bitmap

0	1	0	0	0	0

data bitmap

0	0	0	0	1	1

i-nodes

-- Iv2 -- -- -- --

data blocks

-- -- -- -- D1 D2

owner:	rachit
permissions:	read-only
size:	2
pointer:	4
pointer:	5
pointer:	null
pointer:	null

- Suppose we append a data block to the file
  - add new data block D2
  - update inode
  - update data bitmap

What if a crash or power outage occurs between writes?

If Only a Single Write...

- Just the data block (D2) is written to disk
  - Data is written, but no way to get to it - in fact, D2 still appears as a free block
  - Write is lost, but FS (meta)data structures are consistent
- Just the updated inode (Iv2) is written to disk
  - If we follow the pointer, we read garbage
  - File system inconsistency: data bitmap says block is free, while inode says it is used. Must be fixed
- Just the updated bitmap is written to disk
  - File system inconsistency: data bitmap says data block is used, but no inode points to it. The block will never be used. Must be fixed

If Two Writes…

- Inode and data bitmap updates succeed
  - Good news: file system is consistent!
  - Bad news: reading new block returns garbage
- Inode and data block updates succeed
  - File system inconsistency. Must be fixed
- Data bitmap and data block succeed
  - File system inconsistency
  - No idea which file data block belongs to!

The Consistent Update Problem

- Several file systems operations update multiple data structures
  - Create new file
    - update inode bitmap and data bitmap
    - write new inode
    - add new file to directory file
- Would like to atomically move FS from one consistent state to another

Solution 1:
File System Checker

- Ethos: If it happens, I'll do something about it
  - Let inconsistencies happen and fix them post facto
    - during reboot
- Classic example: fsck
  - Unix, 1986
- Fixing inconsistencies post facto can be VERY slow

Solution 2:
Ordered Updates

- Three rules towards a (quickly) recoverable FS:
  - Never reuse a resource before nullifying all pointers to it (e.g., nullify an i-node pointer to a data block before reallocating that block to another i-node)
  - Never point to a structure before it has been initialized (e.g., must initialize i-node before a directory entry references it)
  - Never clear last pointer to live resource before setting a new one (e.g., when renaming a file, do not remove old name for an i-node until after new name has been written)
- How?
  - A principled approach: Transactions

A principled approach: Transactions

- Group together actions so that they are
  - Atomic: either all happen or none
  - Consistent: maintain invariants
  - Isolated: serializable (schedule in which transactions occur is equivalent to transactions executing sequentially)
  - Durable: once completed, effects are persistent
- Transaction can have two outcomes:
  - Commit: transaction becomes durable
  - Abort: transaction never happened
    - may require appropriate rollback

Solution 3: Journaling (write ahead logging)

- Turns multiple disk updates into a single disk write
  - “write ahead” a short note to a “log”, specifying changes about to be made to the FS data structures
  - if a crash occurs while updating FS data structures, consult log to determine what to do
    - no need to scan entire disk!

Data Jounaling: an example

- We start with

- We want to add a new block to the file
- Three easy steps
  - Write to the log 5 blocks: TxBegin | Iv2 | Bv2 | D2 | TxEnd

    includes TxID and blocks' final addresses
    - write each record to a block, so it is atomic
  - Write the blocks for Iv2, Bv2, D2 to the FS proper [a.k.a checkpoint]
  - Mark the transaction free in the journal
- What if we crash before the log is updated?
  - if no commit, nothing made it into FS - ignore changes!
- What if we crash after the log is updated?
  - replay changes in log back to disk!

Journaling and Write Order

- Issuing the 5 writes to the log TxBegin | Iv2 | B2 | D2 | TxEnd sequentially is slow
    - Issue at once, and transform in a single sequential write!?
- Problem: disk can schedule writes out of order
    - first write TxBegin, Iv2, B2, TxEnd

Disk loses power →

    - then write D2
- Log contains: TxBegin | Iv2 | B2 | ?? | TxEnd
    - syntactically, transaction log looks fine, even with nonsense in place of D2!
- TxEnd must block until prior blocks are on disk
    - Transaction committed when TxEnd on disk

Log Structured File Systems

- Instead of adding a log to the existing FS disk layout, use all disk as a log
  - buffer all updates (including metadata!) into an in-memory data structure
  - Periodically, write to persistent storage in a long sequential transfer
- Never overwrite existing data
  - always write data to "next" free locations
  - Sequential writes: much improved throughput

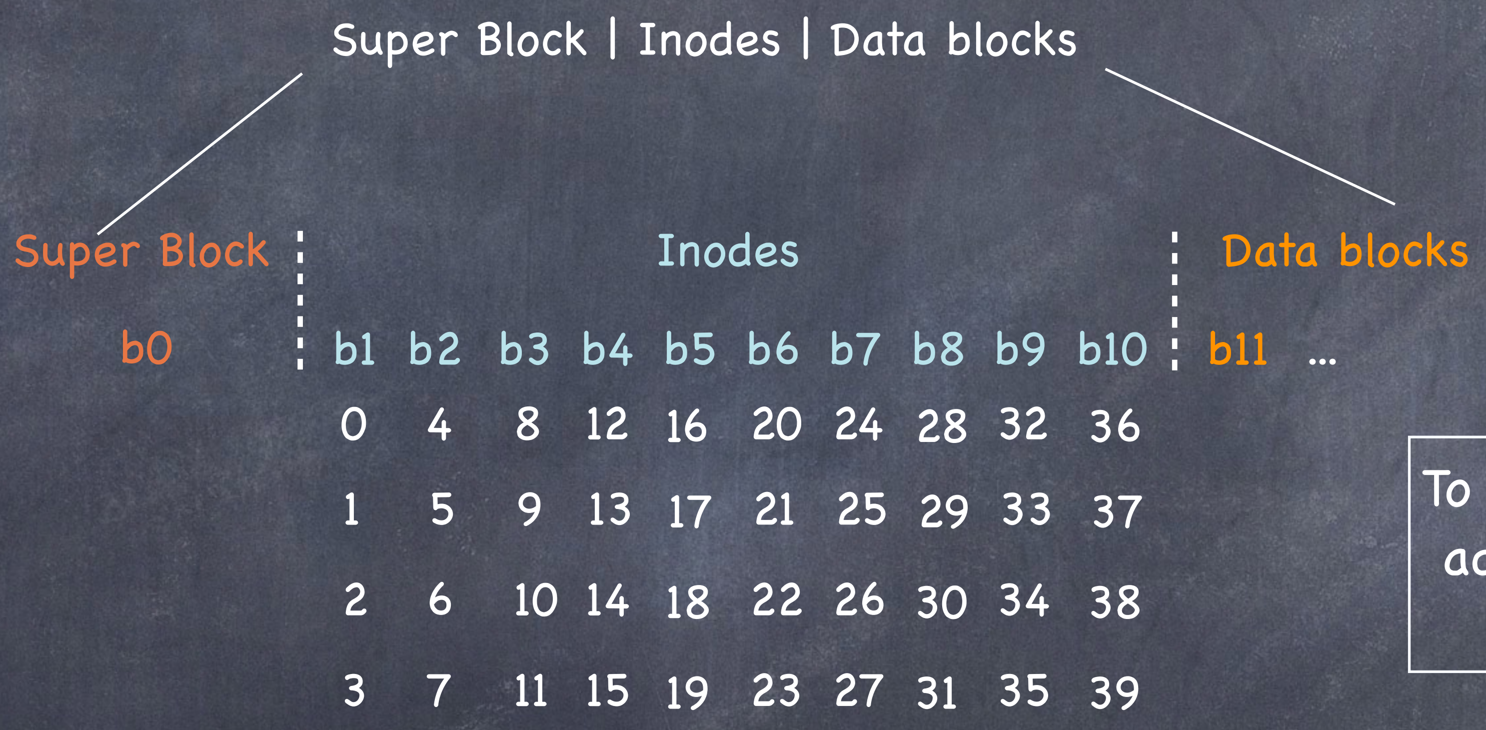
Log Structured File Systems

- But how does it work?
  - ☐ suppose we want to add a new block to a 0-sized file
  - ☐ not enough to write to log just the data block…
  - ☐ …we have to update the inode too!
- LFS places both data block and inode in-memory

- Leverages write buffering to write a chunk of updates all at once

Finding i-nodes

- in UFS, just index into inode array



Super Block | Inodes | Data blocks

Super Block	Inodes										Data blocks	
b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	...
	0	4	8	12	16	20	24	28	32	36		
	1	5	9	13	17	21	25	29	33	37		
	2	6	10	14	18	22	26	30	34	38		
	3	7	11	15	19	23	27	31	35	39		

512 bytes/block
128 bytes/inode

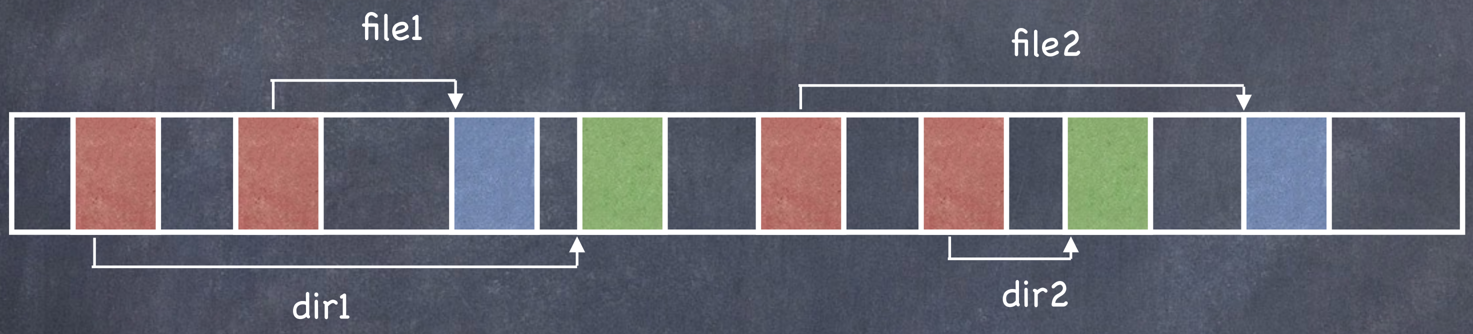
To find address inode 11:
addr(b1) + #inode x size(inode)

- FFS is the same, with i-nodes divided among block groups and stored at known locations
- But in LFS i-nodes are scattered everywhere on disk!

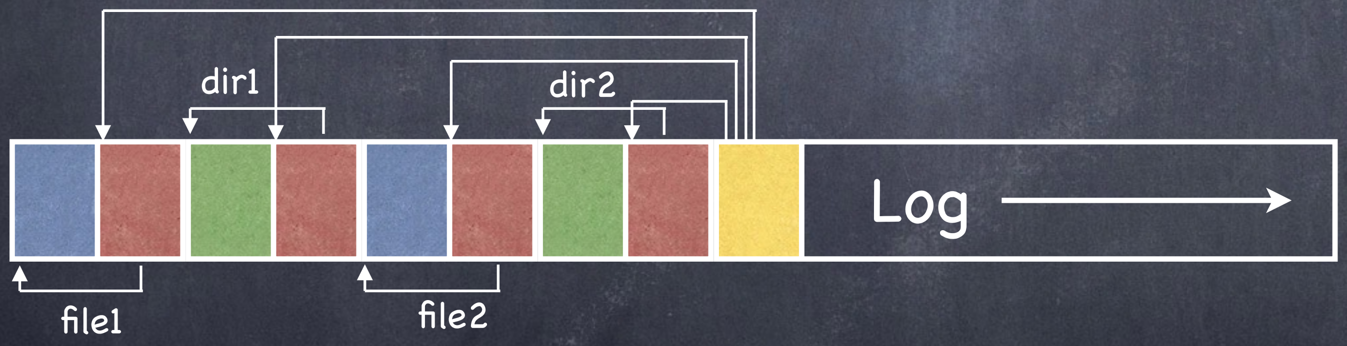
Finding inodes in LFS

- Inode map: a table indicating where each inode is on disk: Imap(i#) -> disk address of i#
- Keep it in memory, and periodically push it to disk
- In case of failures, reconstruct
  - e.g., by scanning data on disk

LFS vs UFS



Unix File System



Log-structured File System

🟥 inode

🟩 directory

🟦 data

🟨 inode map

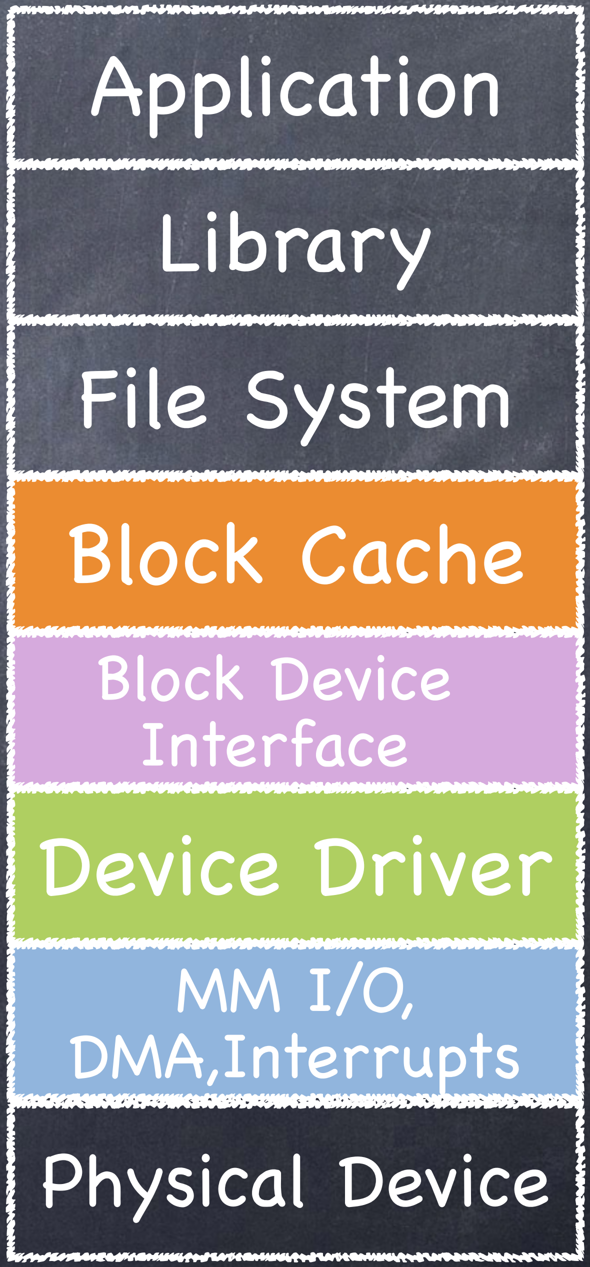
Blocks written to
create two 1-block files:
dir1/file1 and dir2/file2
in UFS and LFS

Garbage collection

- As old blocks of files are replaced by new ones, data in log become fragmented: live and dead.
- Cleaning used to produce contiguous space on which to write
  - compact M fragmented blocks into N new blocks, newly written to the log
  - free old M blocks
- Cleaning mechanism:
  - How can LFS tell which blocks are live and which dead?
- Cleaning policy
  - How often should the cleaner run?
  - How should the cleaner pick blocks?
- No one-size-fits-all solution. Different solutions, different tradeoffs
  - See the discussion for SSD log-structured storage for examples, and tradeoffs

Recall: The Storage Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches blocks recently read from disk
  - Buffers recently written blocks
  - Single interface to many devices, allows data to be read/written in fixed sized blocks
  - Translates OS abstractions and hw specific details of I/O devices
  - Control registers, bulk data transfer, OS notifications


Application
Library
File System
Block Cache
Block Device Interface
Device Driver
MM I/O, DMA,Interrupts
Physical Device