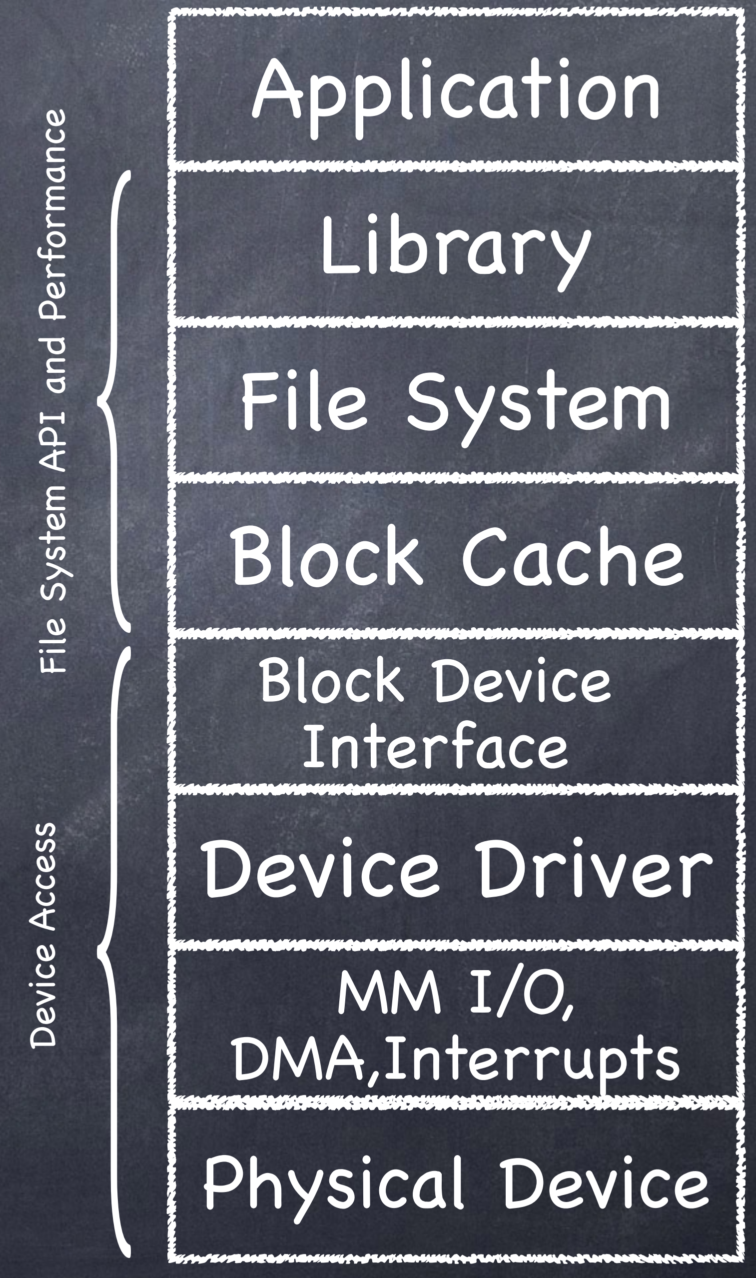
Storage stack:
File Systems:
Storing Files

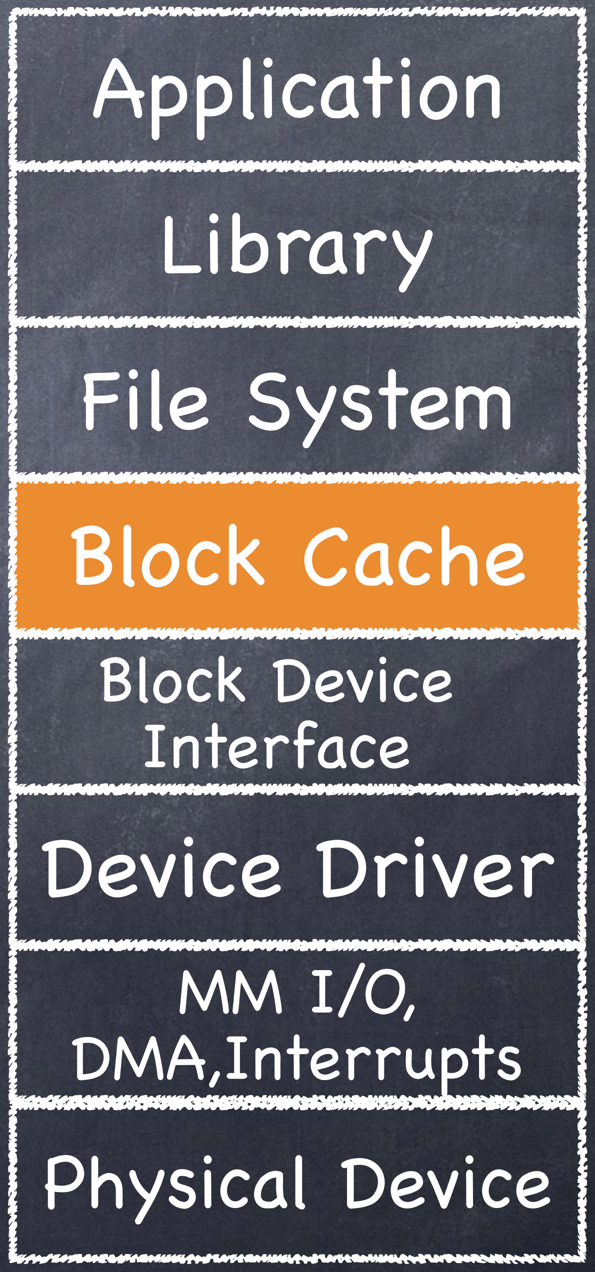
Recall: The Storage Stack

- I/O systems are accessed through a series of layered abstractions



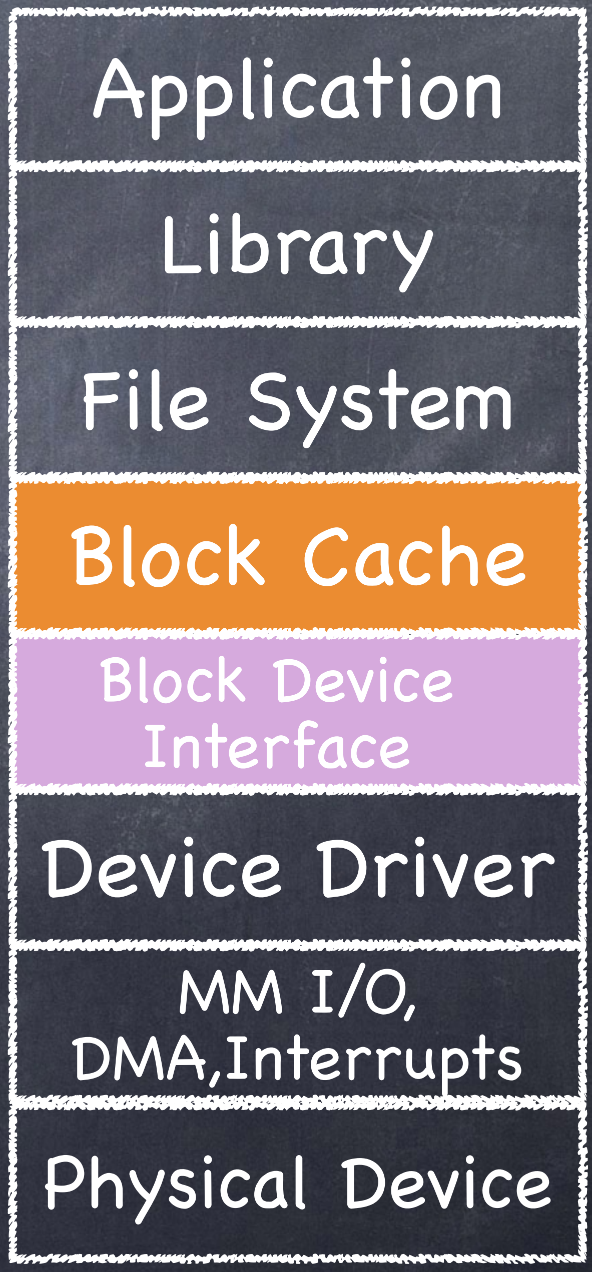
Recall: The Storage Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches blocks recently read from disk
  - Buffers recently written blocks



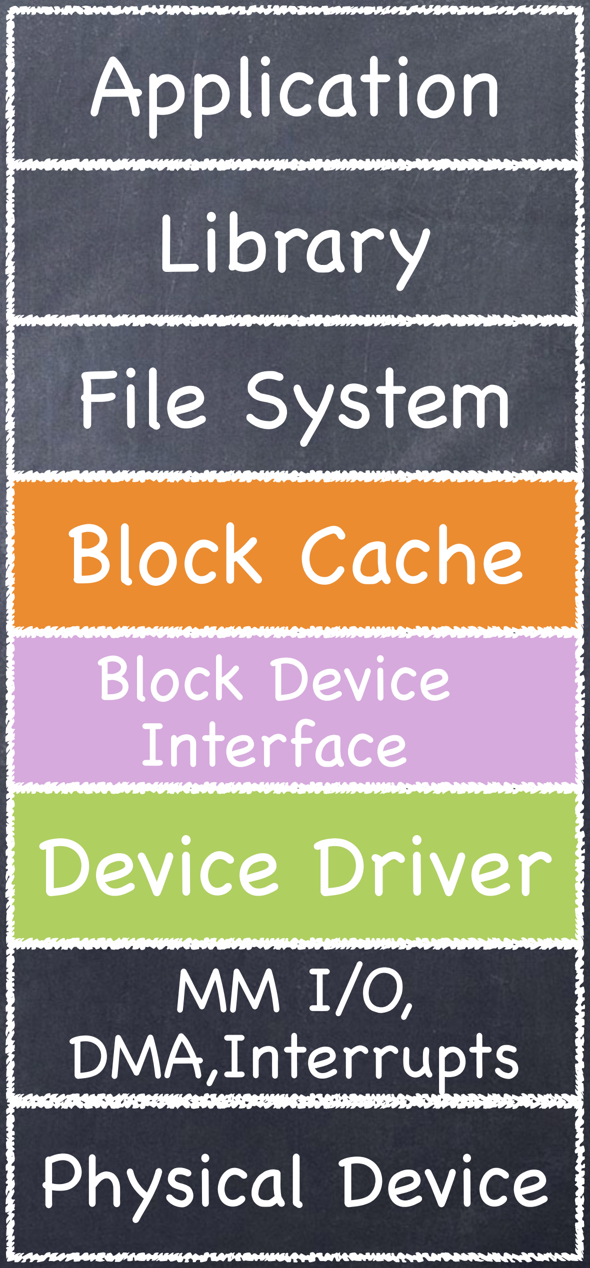
Recall: The Storage Stack

- 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



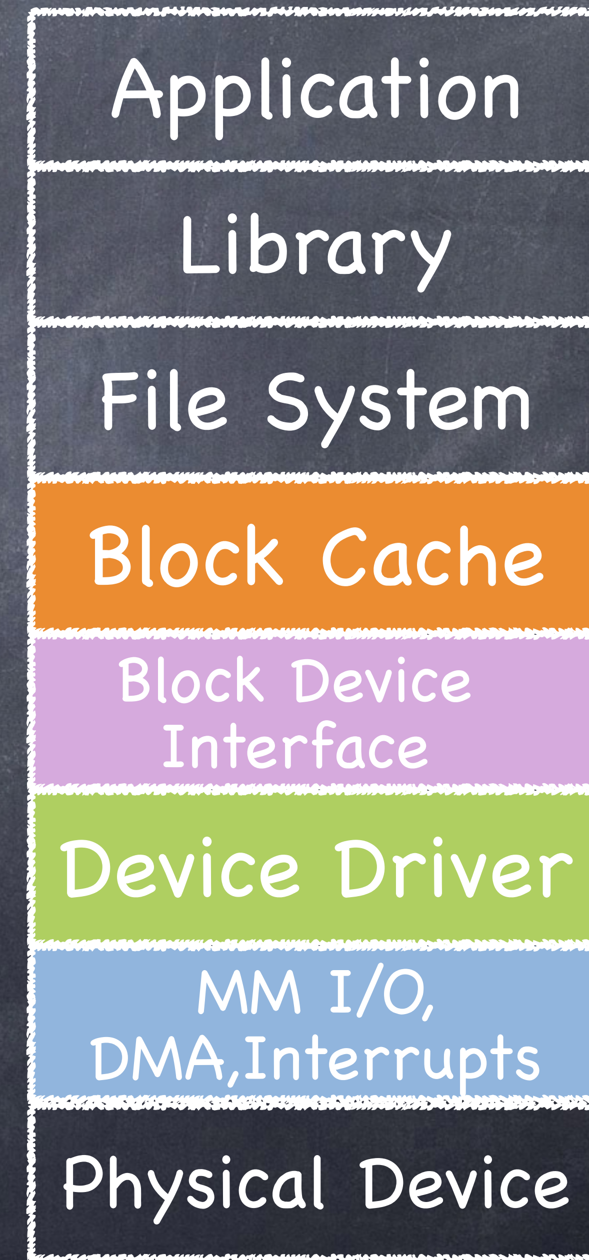
Recall: The Storage Stack

- 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



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



Recall: The File System Abstraction

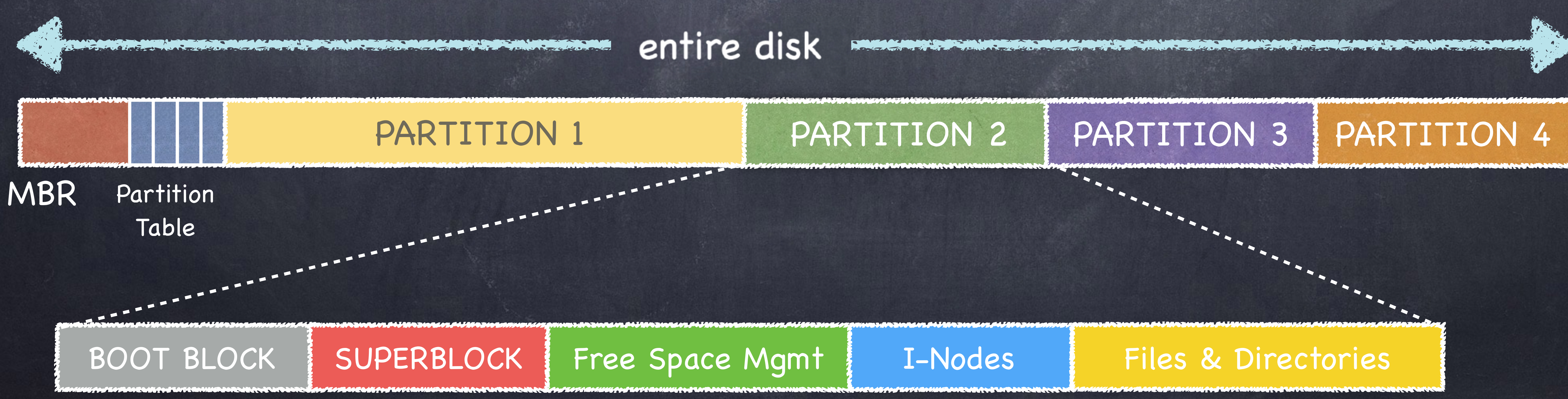
- Addresses need for long-term information storage:
  - store large amounts of information
  - do it in a way that outlives processes (RAM will not do)
  - can support concurrent access from multiple processes
- Presents applications with persistent, named data
- Two main components:
  - files
  - directories

Recall: The File

- A file is a named collection of data. In fact, it has many names, depending on context:
  - i-node number: low-level name assigned to the file by the file system
  - path: human friendly name (HFN)—a string
    - must be mapped to inode number, somehow
  - file descriptor
    - dynamically assigned handle process a uses to refer to i-node
- The directory is just a special file

Recall: File System Layout

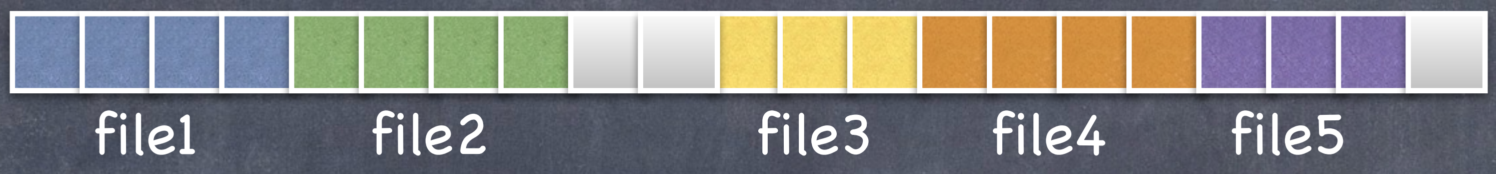
- File System is stored on disks
  - Storage device be divided into one or more partitions
  - At a known location: Master Boot Record (MBR). It contains:
    - bootstrap code (loaded and executed by firmware)
    - partition table (addresses of where partitions start & end)
  - First block of each partition has boot block
    - loaded by executing code in MBR and executed on boot



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?

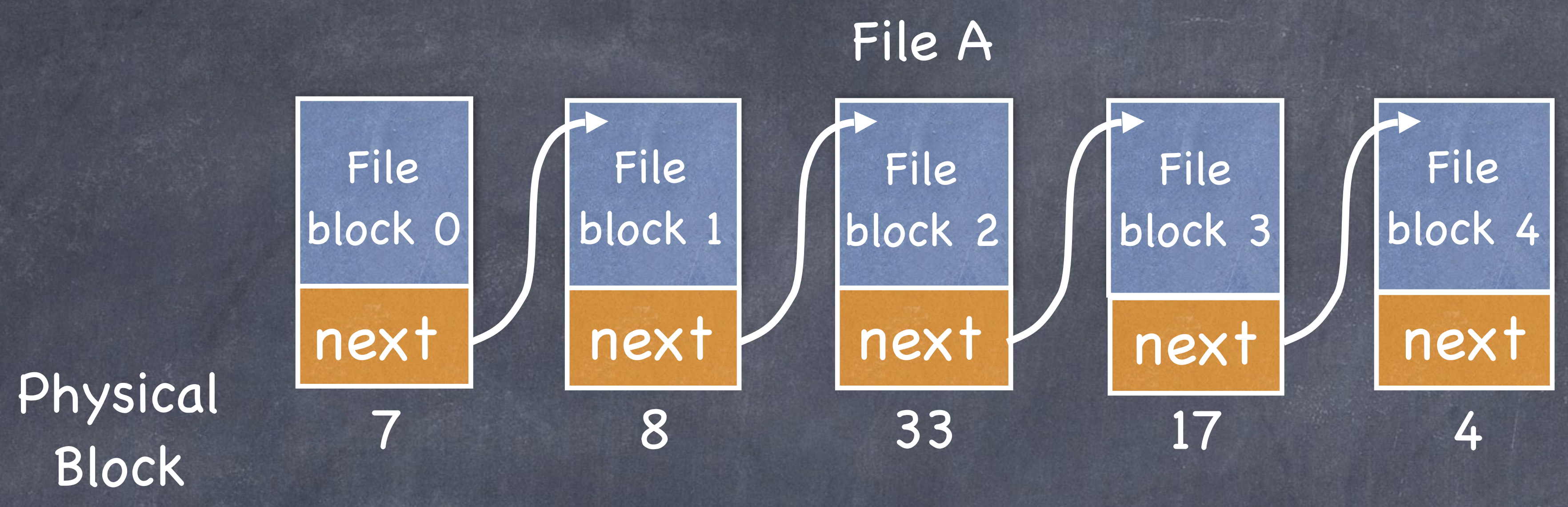
Contiguous Allocation



- All bytes together, in order
  - Simple: only need start block and size
  - Efficient: one seek to read entire file
  - Fragmentation: external, and can be serious
  - Usability: User need to know file's size at time of creation
    - Or, a lot of "moving files around" as file size increases

Used in CD-ROm, DVDs

Linked List Allocation

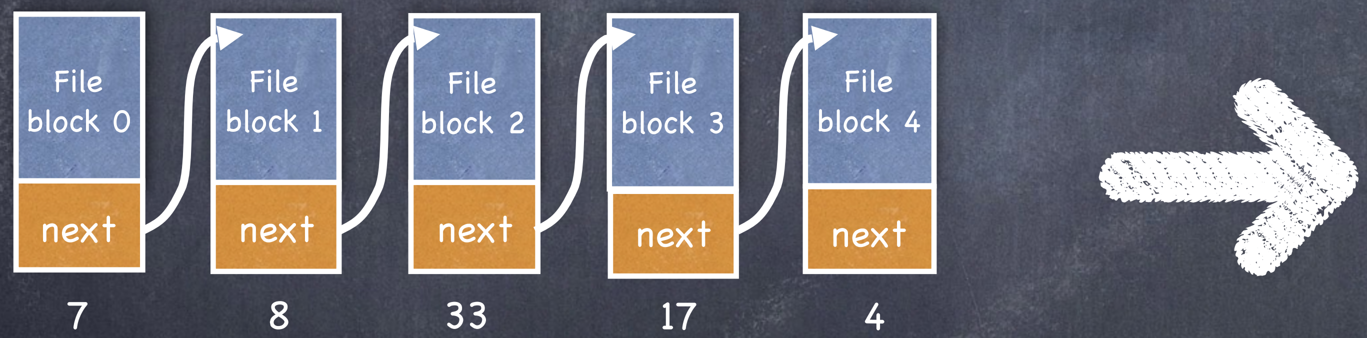


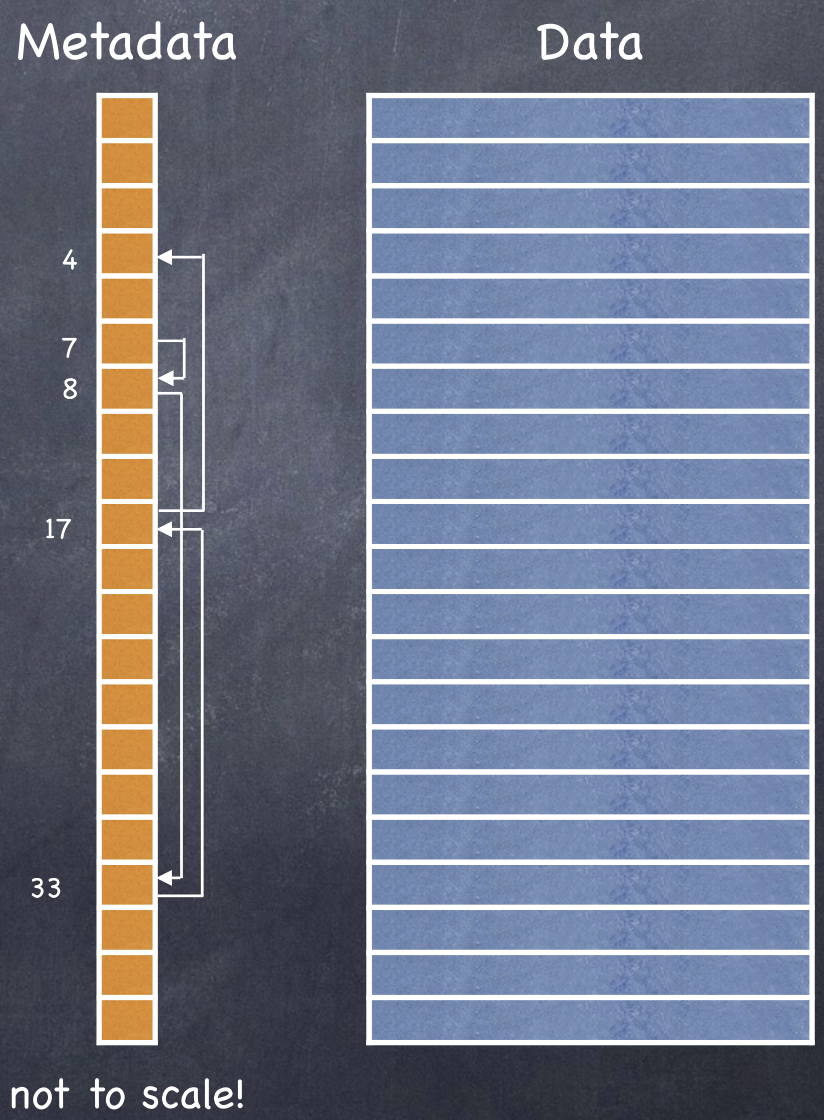
- Each file is stored as a linked list of blocks
  - first word of each block points to next block
  - the rest of the block is data

- Space utilization: no external fragmentation
- Simplicity: only need to find first block of each file
- Performance: random access is slow
  - Core problem?
  - Accessing a byte may require accessing many many blocks
- Implementation: blocks mix data and metadata

File Allocation Table (FAT) FS

- Decouple data and metadata
  - reduces seeks (and enables caching!)





Microsoft, late 70s

- still widely used today
  - thumb drives, camera cards, CD ROMs

FAT File system

Index Structures

File Allocation Table (FAT)

- array of 4-byte entries
- one entry per block
- file represented as a linked list of FAT entries
- file # = index of first FAT entry

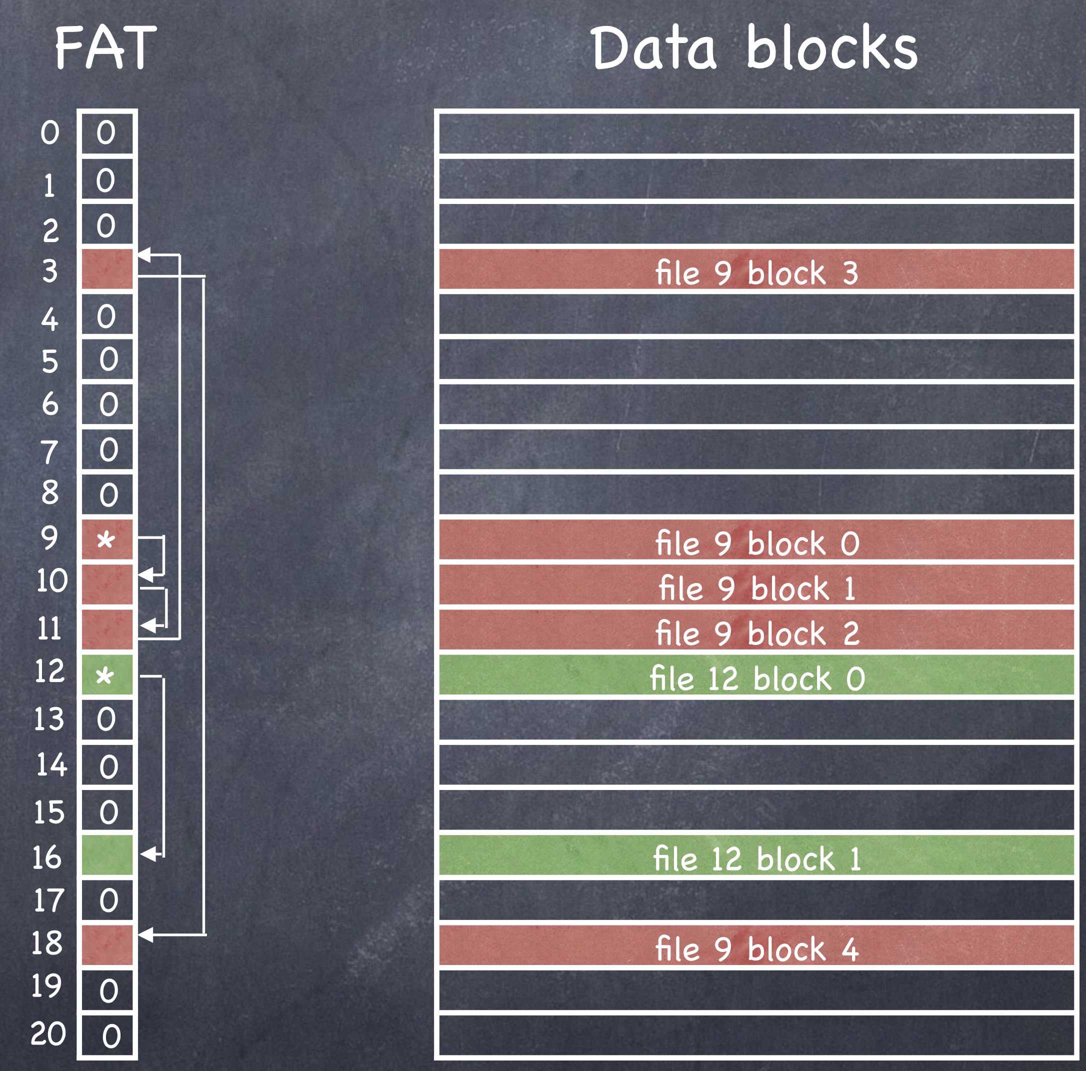
Free space map

- If data block i is free, then FAT[i] = 0
- find free blocks by scanning FAT

Directory

- Maps file name to FAT index

Directory	
jack.txt	12
jill.txt	9



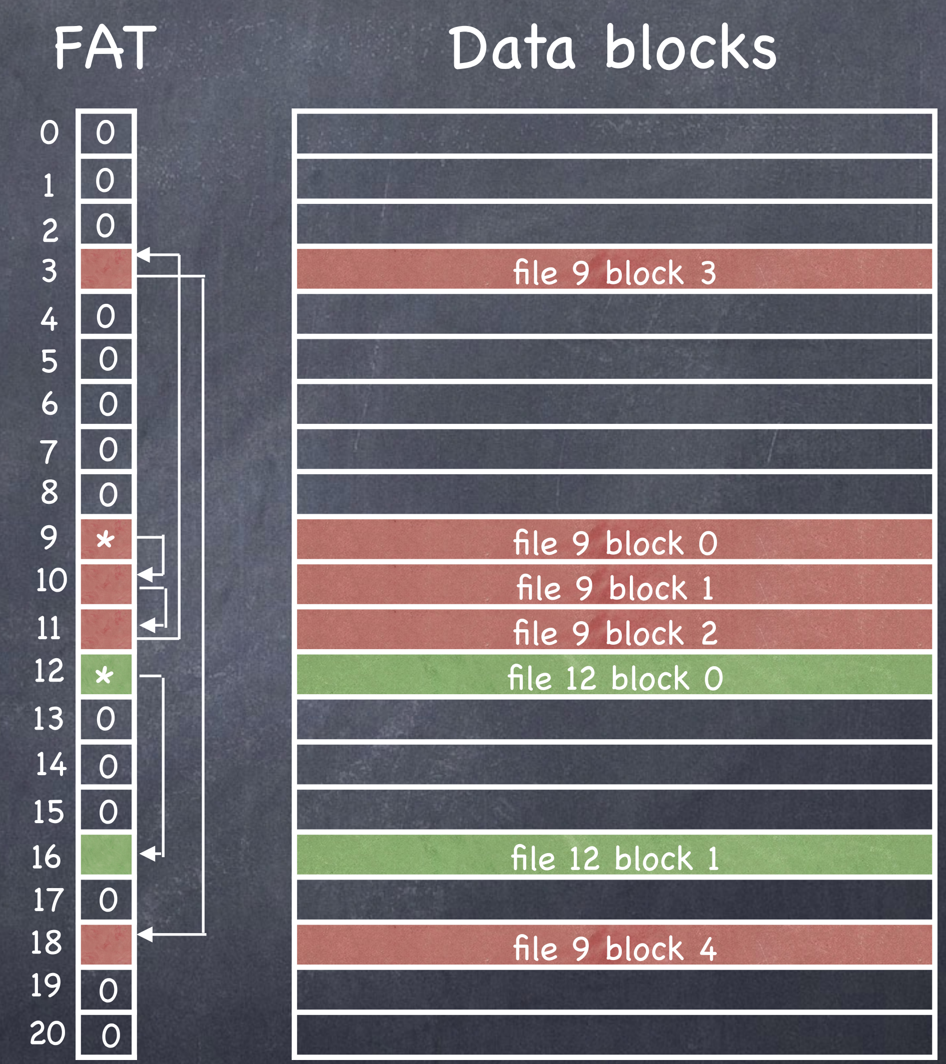
FAT File system

Advantages

- simple!
  - per file, needs only start block
- widely supported
- no external fragmentation
- no conflating data and metadata in the same block

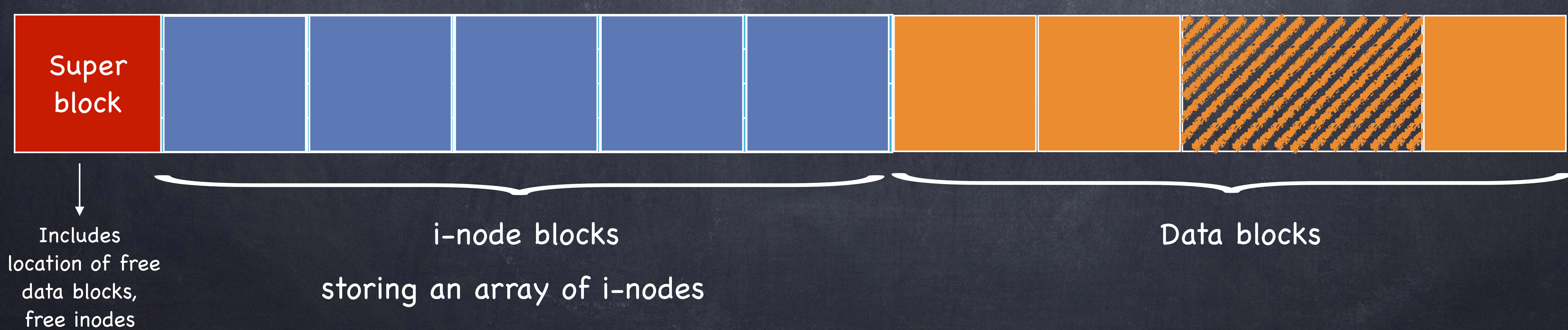
Disadvantages

- Poor locality
  - many file seeks unless entire FAT in memory
  - 1 TB (2^{40} bytes) disk, 4kB (2^{12} bytes) block, 2^{28} FAT entries; at 4B/entry, 1 GB (!)
- Poor random access
  - needs sequential traversal
- Volume and file size are limited
  - FAT entry is 32 bits, but top 4 are reserved
  - no more than 2^{28} blocks
  - with 4kB blocks, at most 1TB FS
  - file no bigger than 4GB
    - Directory also has 32 bit entries
- No support for advanced reliability techniques



Tree-based Multi-level Index

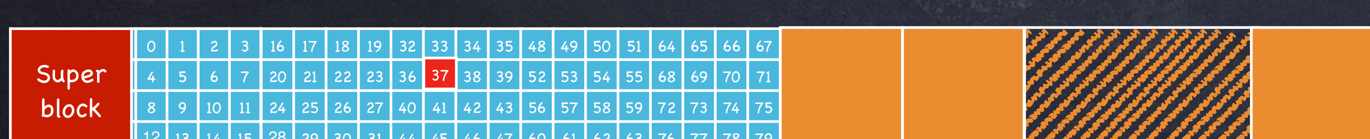
- UFS (Unix File System) (Ken Thompson, 1969)
- 4.2 BSD FFS (Fast File System) (McKusick, Joy, Leffler, Fabry, 1983)



Multilevel index

Inode Array

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

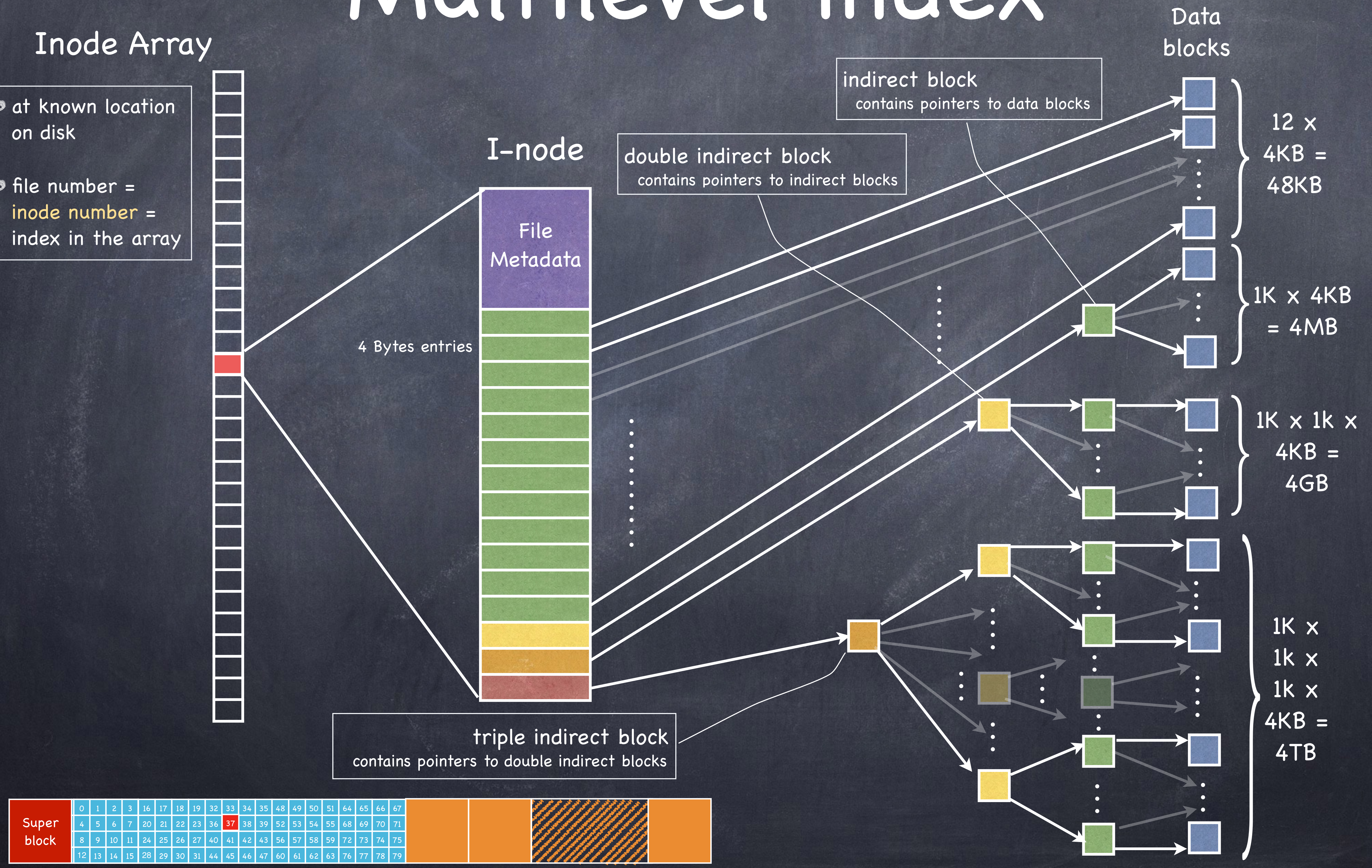


File structure

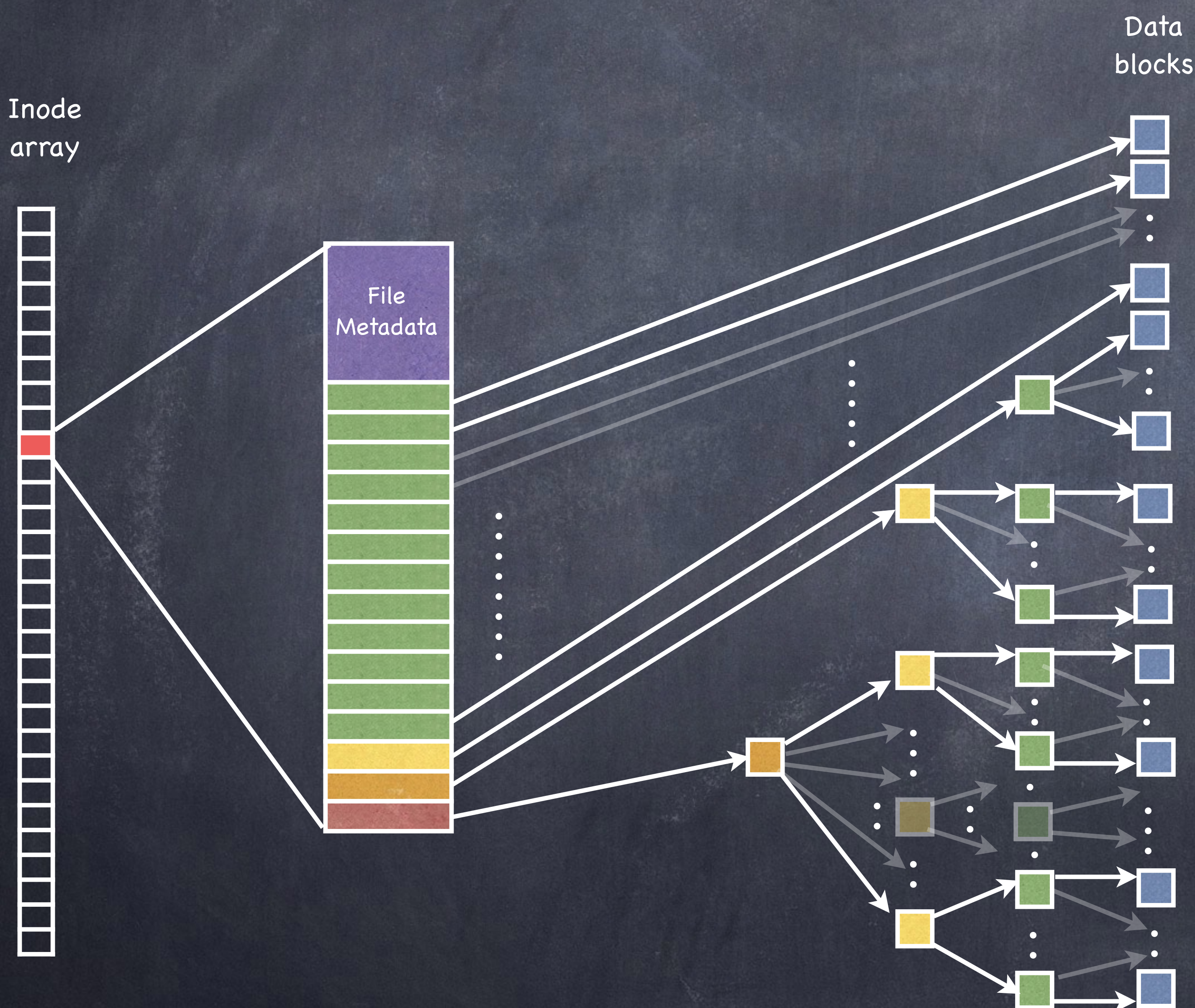
- Each file is a fixed, asymmetric tree, with fixed size data blocks (e.g. 4KB) as its leaves
- The root of the tree is the file's inode, containing
  - metadata (more about it later)
  - a set of 15 pointers
    - first 12 point to data blocks
    - last three point to intermediate blocks, themselves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

Multilevel index

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

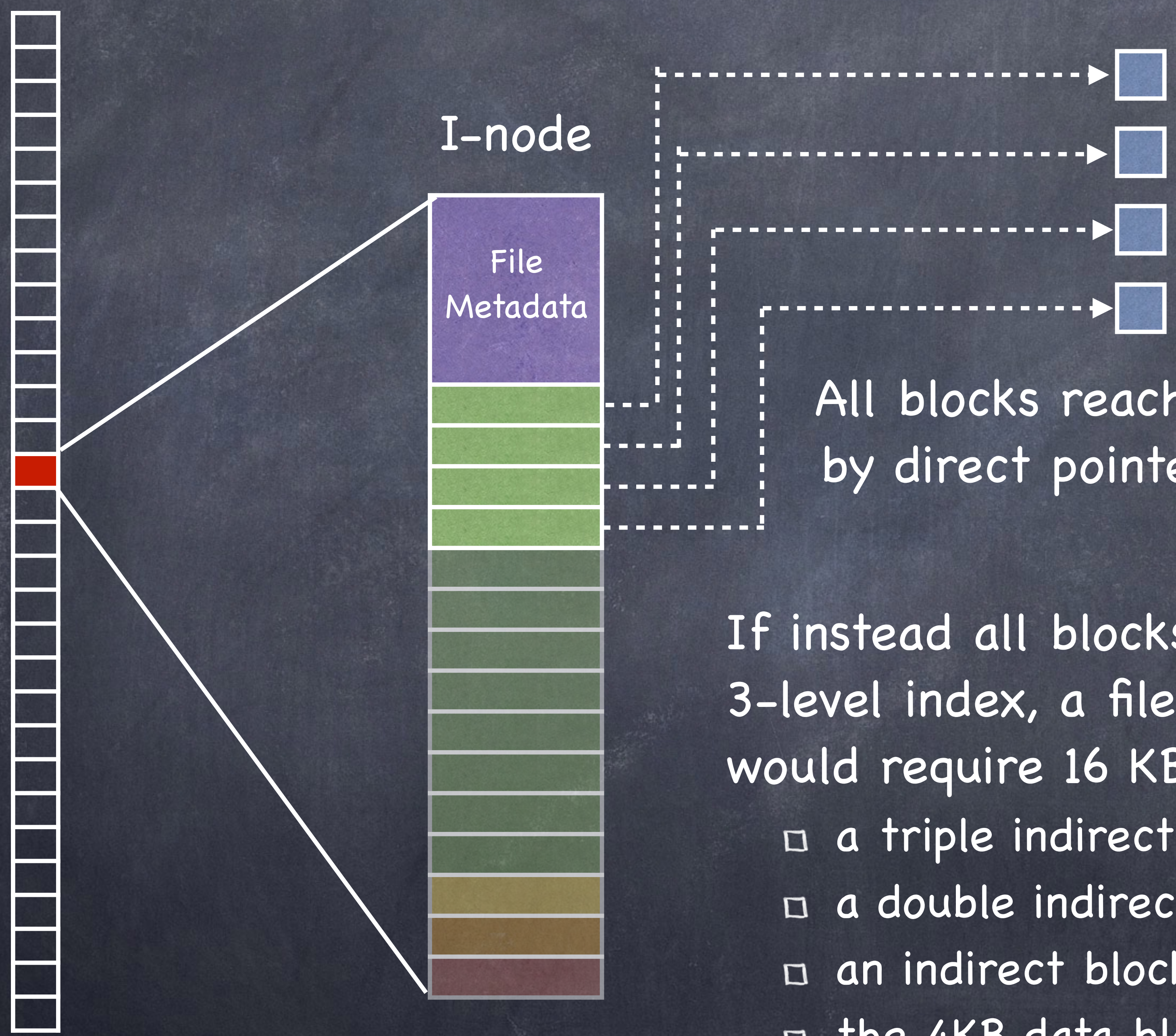


Multilevel index: key ideas



- Tree structure
  - efficient in finding blocks
- High degree
  - efficient in sequential reads
    - once an indirect block is read, can read 100s of data block
- Fixed structure
  - simple to implement
- Asymmetric
  - supports large files
  - small files don't pay large overheads

Good for small files...



All blocks reached by direct pointers

If instead all blocks were accessed through a 3-level index, a file occupying a single 4KB block would require 16 KB:

- a triple indirect block
- a double indirect block
- an indirect block
- the 4KB data block
- reading would require reading 5 blocks to traverse the tree

Why Unbalanced Trees? (and other fun facts)

- Most files are small
  - Roughly 2K is the most common size
- Average file size is growing
  - Almost 200K is the average
- Most bytes are stored in large files
  - A few big files use most of the space
- File systems contains lots of files
  - Almost 100K on average
- File systems are roughly half full
  - Even as disks grow, file system remains about 50% full
- Directories are typically small
  - Many have few entries; most have 20 or fewer

What else is in an i-node?

- Type
  - ordinary file
  - directory
  - symbolic link
  - special device
- Size of the file (in bytes)
- No. of links to the i-node
- Owner (user id & group id)
- Protection bits
- Times: creation, last accessed, last modified

Reading a File

- First, must open the file
  - Follow the directory tree, until we get to the file's inode
  - Read that inode
    - do a permission check
    - return a file descriptor fd
- Then, for each `read()` that is issued:
  - read inode
  - read appropriate data block (depending on offset)
  - update last access time in inode
  - update file offset in in-memory open file table for fd

Writing a File

- Must open the file, like before
- But now may have to allocate a new data block
  - each logical write can generate up to five I/O ops
    - reading the free data block bitmap
    - writing the free data block bitmap
    - reading the file's inode
    - writing the file's inode to include pointer to the new block
    - writing the new data block
- Creating a file is even worse!
    - read and write free inode bitmap
    - write inode
    - (read) and write directory data
    - write directory inode

and if directory block is full, must allocate another block

BSD FFS: Fast File System

- UFS treats disks as if they were RAM
  - files grab first free data block: seeks and fragmentation
- FFS optimizes file system layout for how disks work
- Smart locality heuristics
  - block group placement
    - optimizes placement for when a file data and metadata, and other files within same directory, are accessed together
  - reserved space
    - gives up about 10% of storage to allow flexibility needed to achieve locality

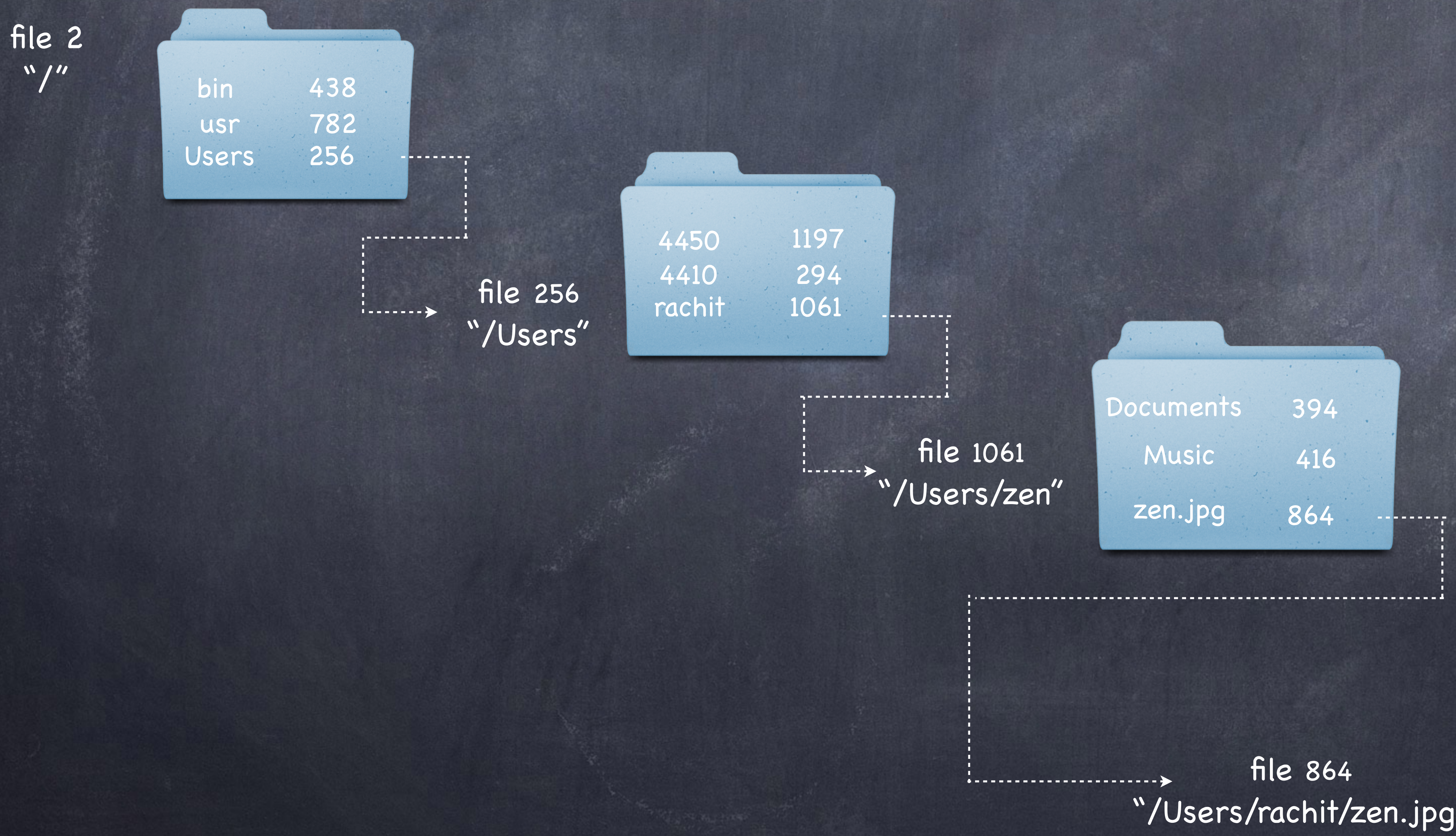
Directory

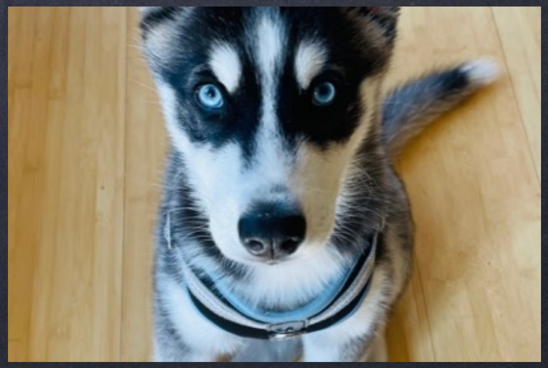
- A file that contains a collection of mapping from file name to file number

- To look up a file, find the directory that contains the mapping to the file number
- To find that directory, find the parent directory that contains the mapping to that directory's file number...
- Good news: root directory has well-known number (2)

Looking up a file

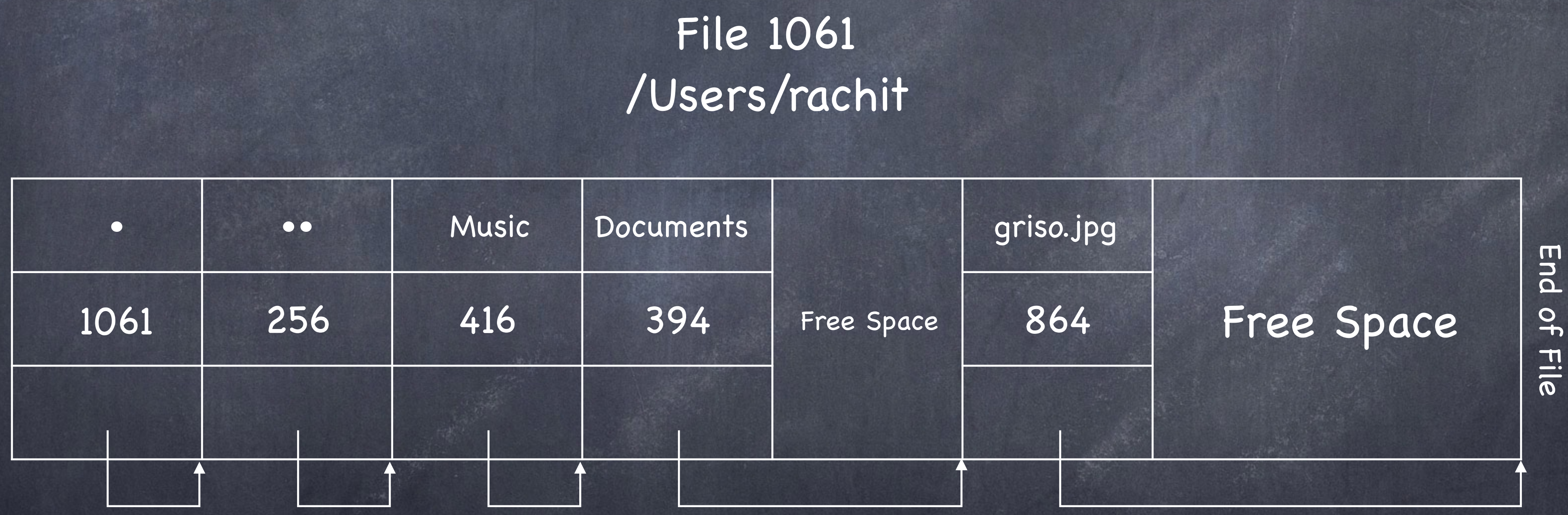
- Find file /Users/rachit/zen.jpg





Directory Layout

- Directory stored as a file
  - Linear search to find filename (small directories)



- Larger directories use B trees
  - searched by hash of file name

Questions?

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

inode bitmap	data bitmap	i-nodes	data blocks
0 1 0 0 0 0	0 0 0 0 1 0	-- Iv1 -- -- -- --	-- -- -- -- D1 --

- 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

inode bitmap	data bitmap	i-nodes	data blocks
0 1 0 0 0 0	0 0 0 0 1 0	-- Iv1 -- -- -- --	-- -- -- -- D1 D2

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: | 0 | 1 | 0 | 0 | 0 | 0 |

data bitmap: | 0 | 0 | 0 | 0 | 1 | 0 |

i-nodes: -- Iv2 -- -- -- --

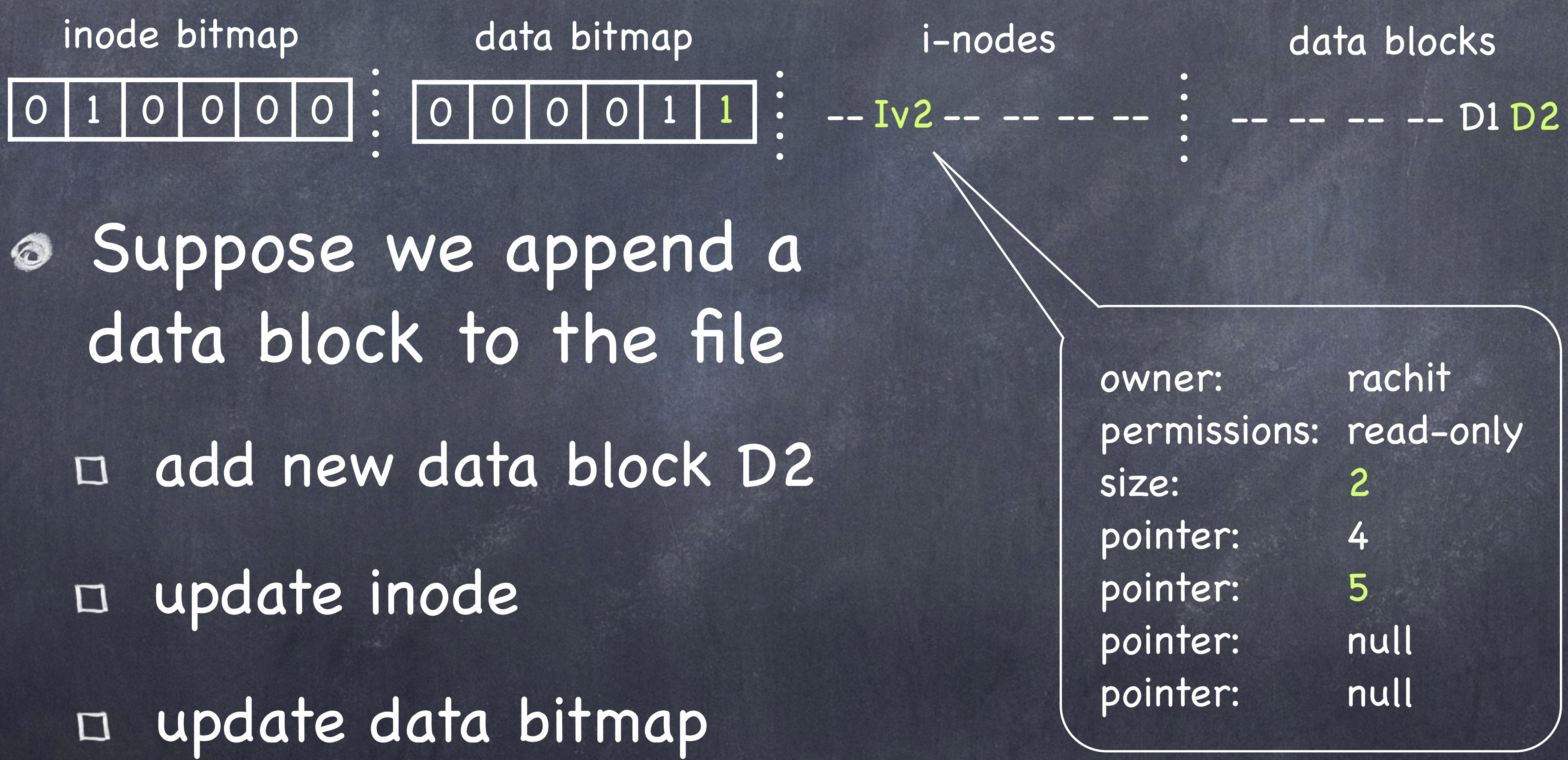
data blocks: -- -- -- -- D1 D2

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

Example: a tiny ext2

- 6 blocks, 6 inodes



- 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