Lecture 22: File Systems (cont'd)

CS 105

December 3, 2019

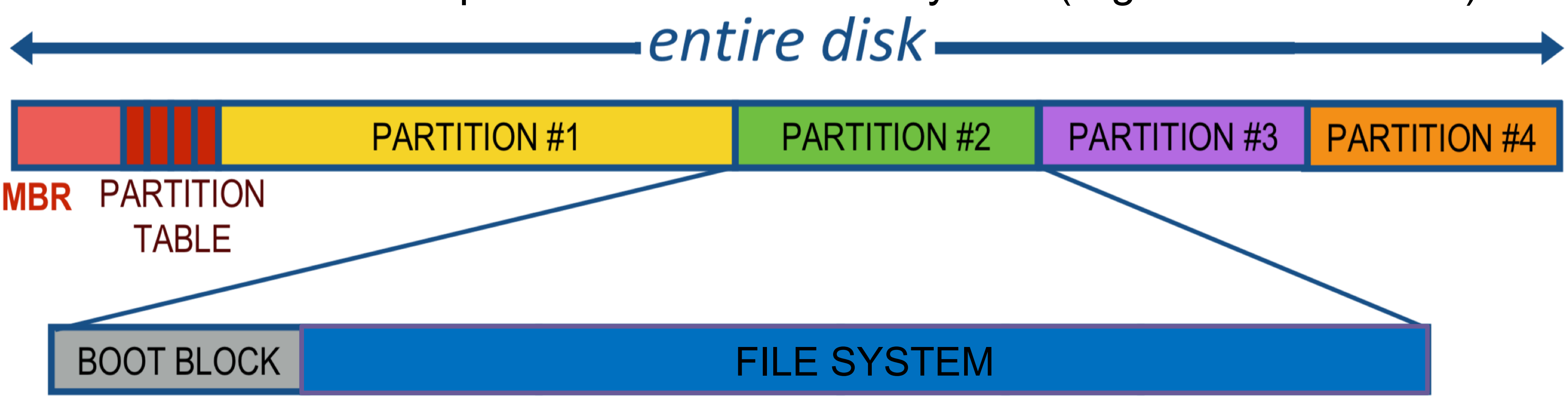
Review: File Systems 101

- Long-term information storage goals
  - **Persistence:** maintain/update user data + internal data structures on persistent storage devices
  - **Flexibility:** need to support diverse file types and workloads
  - **Performance:** despite limitations of disks
  - **Reliability:** must store data for long periods of time despite OS crashes or hardware malfunctions

- Solution: the File System Abstraction
  - interface that provides operations involving
    - files
    - directories (a special kind of file)

Review: File System Layout

- File systems are stored on disks
  - disks can be divided into one or more partitions
- Sector 0 of disk called Master Boot Record
  - executable boot loader
  - end of MBR: partition table (contains partitions' start & end addr.)
- Remainder of disk divided into partitions
  - First block of each partition is boot block (loaded by MBR on boot)
  - The rest of the partition stores the file system (organized as blocks)



Review: Storing Files

Possible ways to allocate files:

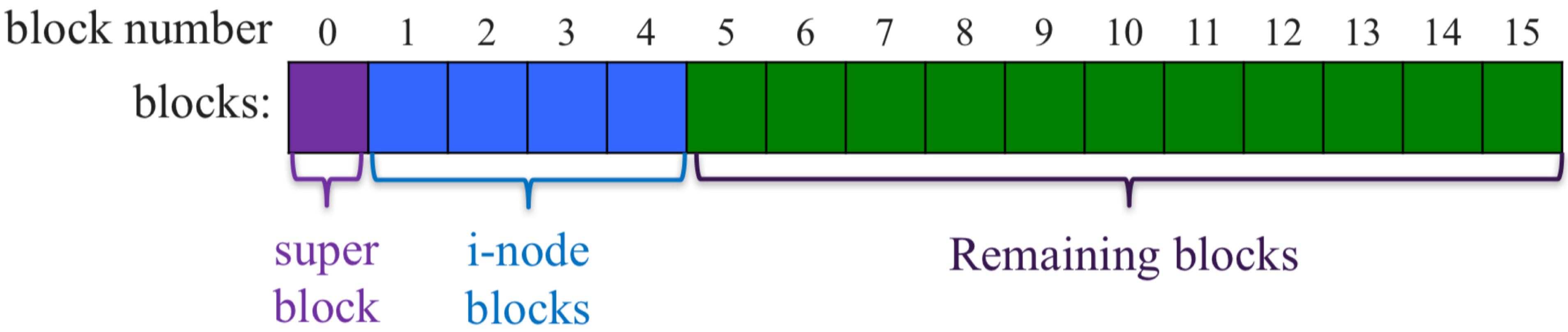
- **Continuous allocation:** all bytes together, in order
  - simple, efficient read, but low utility (external fragmentation) and usability
- **Linked structure:** each block points to the next block
  - still simple, better utilization, slower read (random access), ptr overhead
- **Indexed structure:** index block points to many other blocks

- **Log structure:** sequence of segments, each containing updates

Indexed Allocation: Fast File System (FFS)

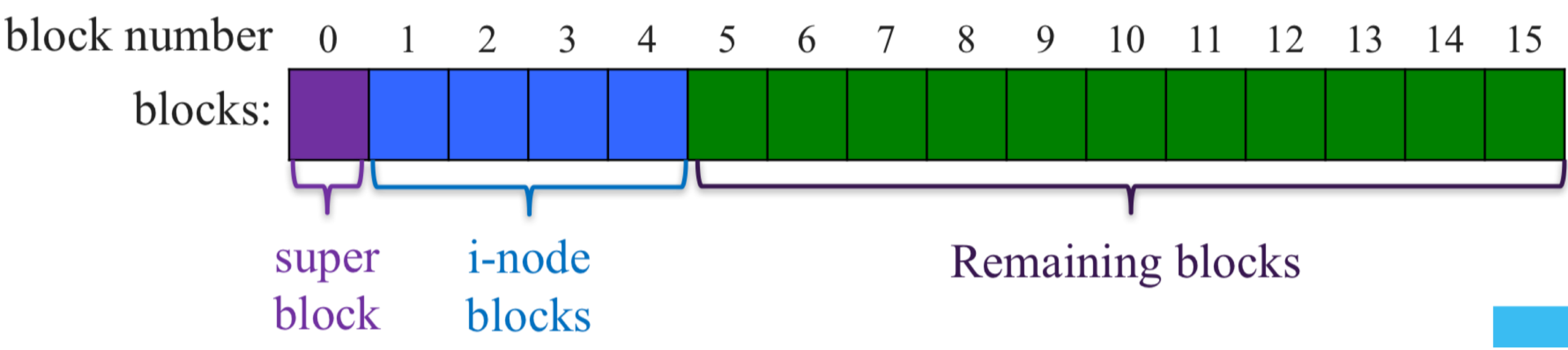
- tree-based, multi-level index

- **superblock** identifies file system's key parameters
- **inodes** store metadata and pointers
- **datablocks** store data



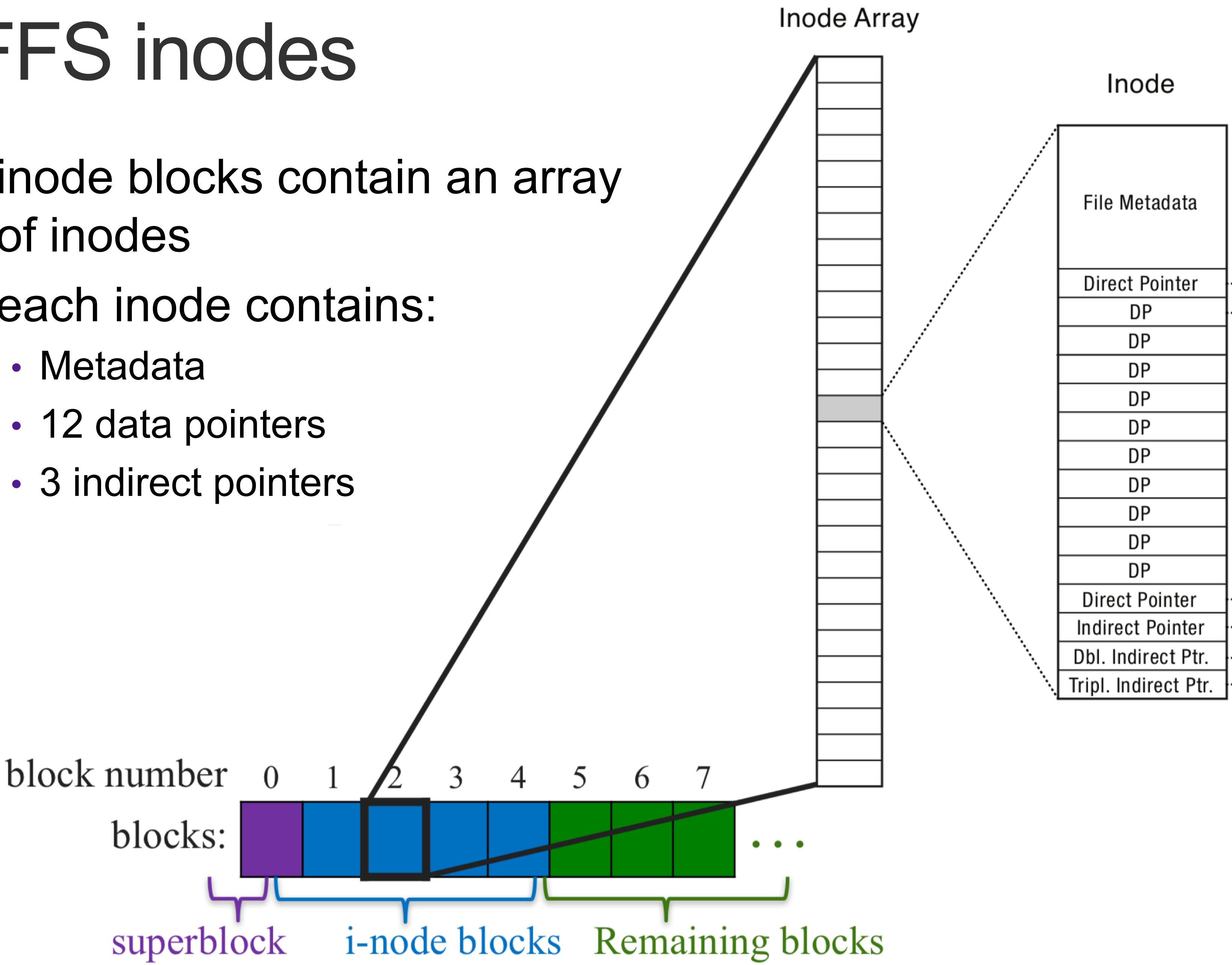
FFS Superblock

- Identifies file system's key parameters:
  - type
  - block size
  - inode array location and size
  - location of free list



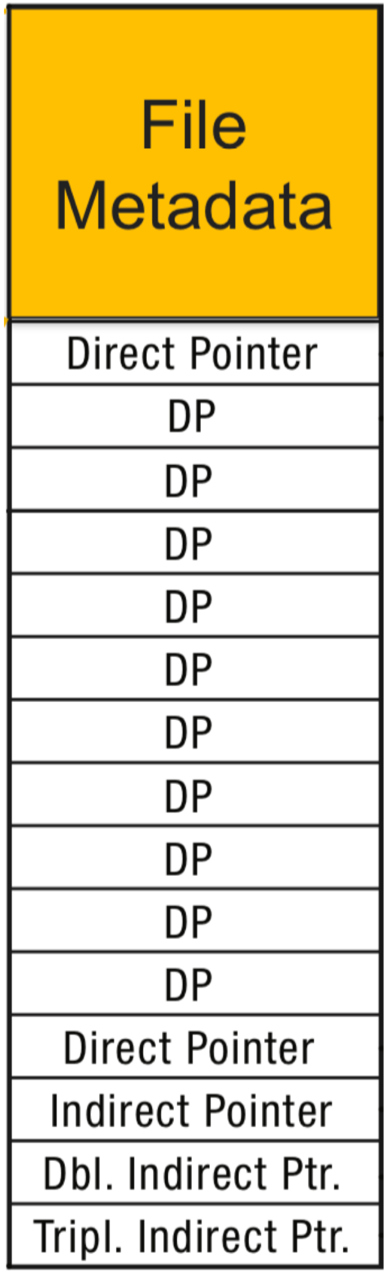
FFS inodes

- inode blocks contain an array of inodes
- each inode contains:
  - Metadata
  - 12 data pointers
  - 3 indirect pointers

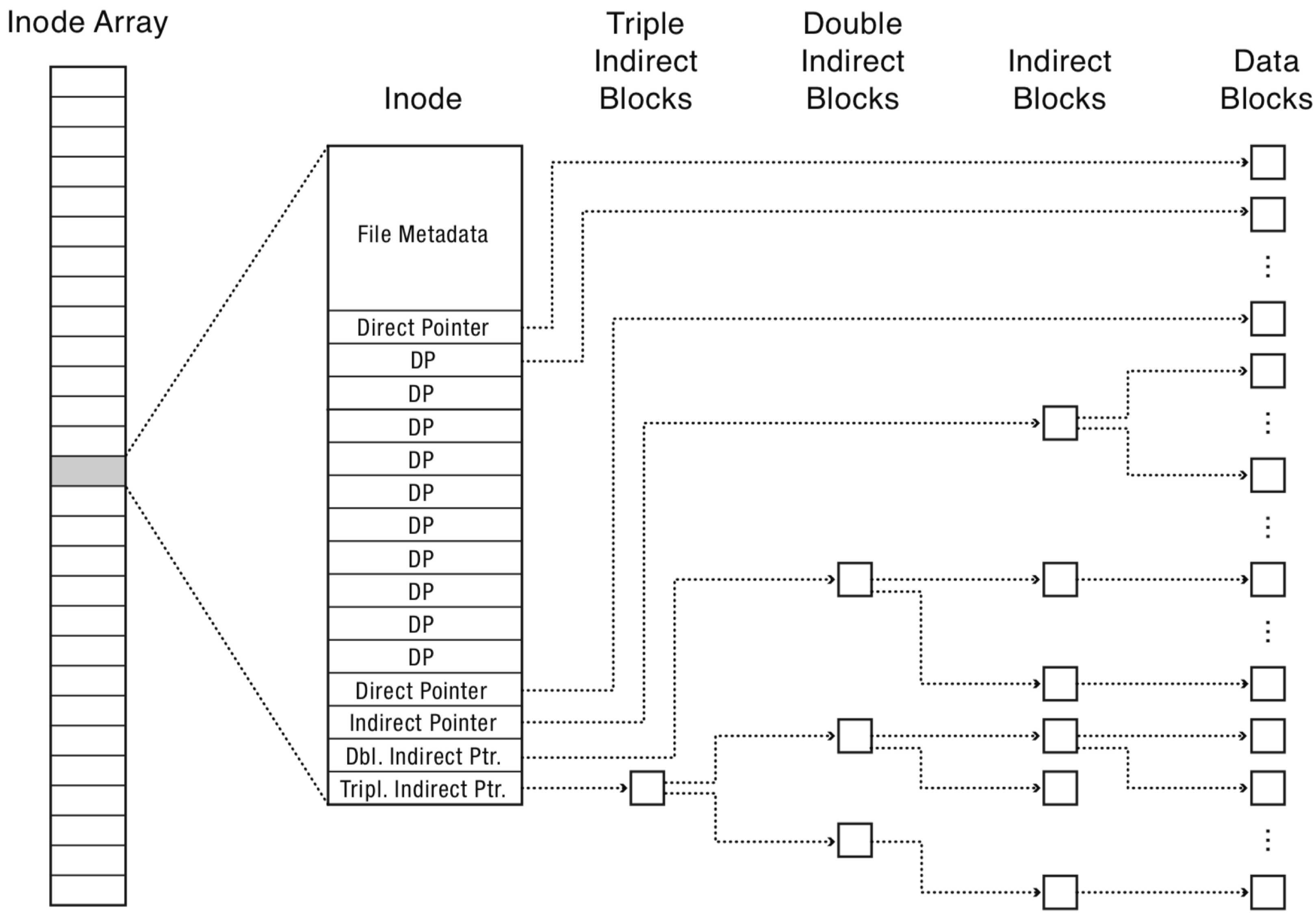


inode Metadata

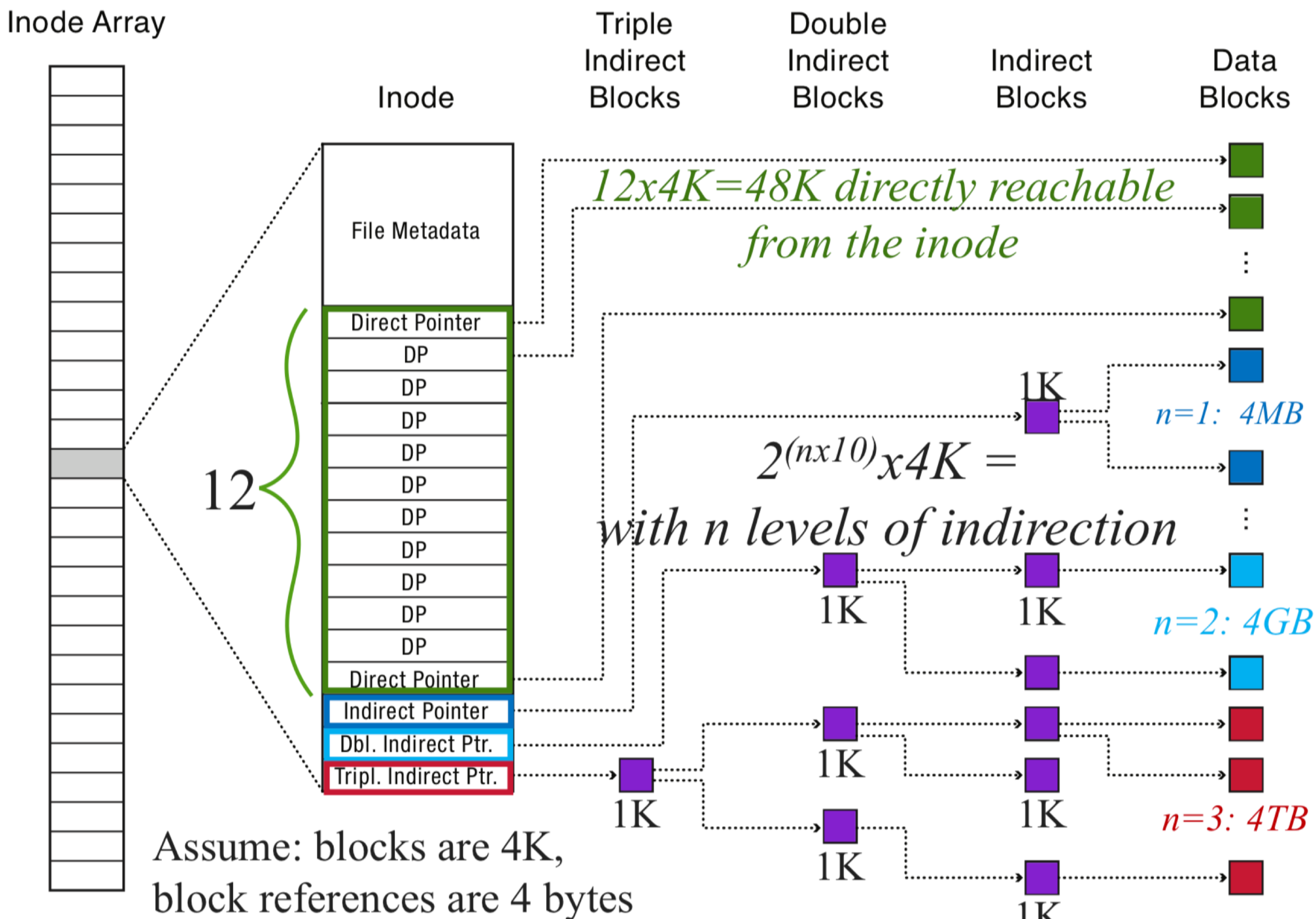
- Type
  - ordinary file
  - directory
  - symbolic link
  - special device
- Size of the file (in #bytes)
- # links to the i-node
- Owner (user id and group id)
- Protection bits
- Times: creation, last accessed, last modified



FFS Index Structures



FFS Index Structures



Assume: blocks are 4K,
block references are 4 bytes

Key Characteristics of FFS

- Tree Structure
  - efficiently find any block of a file
- High Degree (or fan out)
  - minimizes number of seeks
  - supports sequential reads & writes
- Fixed Structure
  - implementation simplicity
- Asymmetric
  - not all data blocks are at the same level
  - supports large files
  - small files don’t pay large overheads

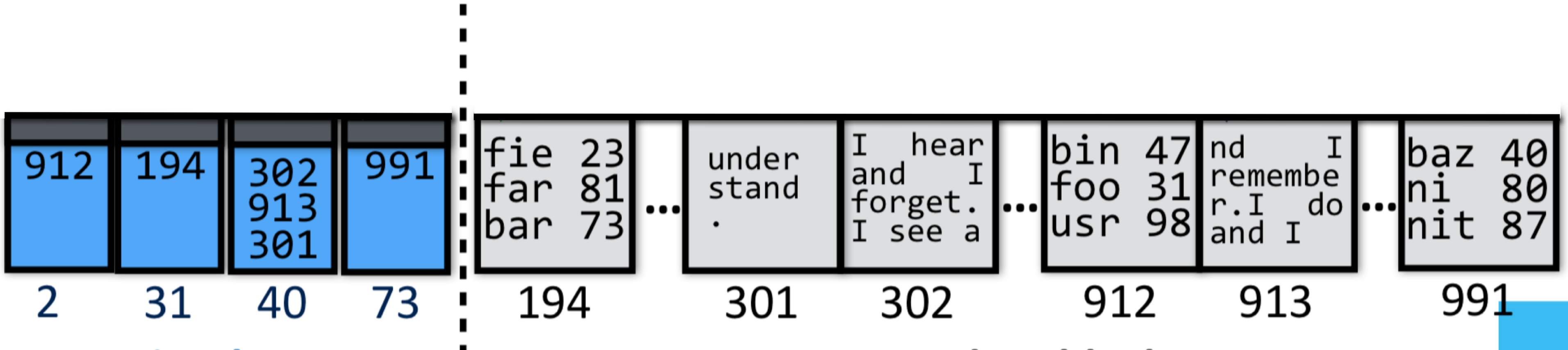
FFS Directory Structure

- Originally: array of 16 byte entries
  - 14 byte file name
  - 2 byte i-node number
- Now: implicit list. Each entry contains:
  - 4-byte inode number
  - Full record length
  - Length of filename
  - Filename
- First entry is ".", points to self
- Second entry is "..", points to parent inode

Exercise: Reading Files with FFS

To read file /foo/bar/baz, Read & Open:

1. inode #2 (root always has inumber 2), find root's blocknum (912)
2. root directory (in block 912), find foo's inumber (31)
3. inode #31, find foo's blocknum (194)
4. foo (in block 194), find bar's inumber (73)
5. inode #73, find bar's blocknum (991)
6. bar (in block 991), find baz's inumber (40)
7. inode #40, find data blocks (302, 913, 301)
8. data blocks 302
9. data block 913
10. data block 301



Key Characteristics of FFS

- Tree Structure
  - efficiently find any block of a file
- High Degree (or fan out)
  - minimizes number of seeks
  - supports sequential reads & writes
- Fixed Structure
  - implementation simplicity
- Asymmetric
  - not all data blocks are at the same level
  - supports large files
  - small files don’t pay large overheads

Exercise: File organization

- Imagine you are trying to organize your (large) collection of photographs. You consider two possible ways to store your files

1. Put all the files in a director called /photos/
    - e.g., /photos/xyzstu.jpg, /photos/xeftuv.jpg, /photos/xywqre.jpg

2. Organize the photos into nested subdirectories organized by the first three characters of the filename
    - e.g., /photos/x/y/z/xyzstu.jpg, /photos/x/e/f/xeftuv.jpg, /photos/x/y/w/xywqre.jpg

- What are the tradeoffs between these two approaches, and which one would you adopt?

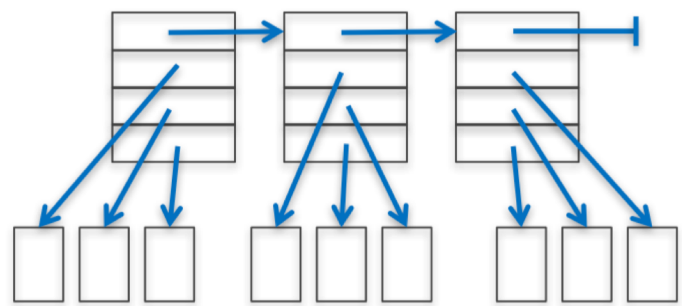
Free List

To write files, need to keep track of which blocks are currently free

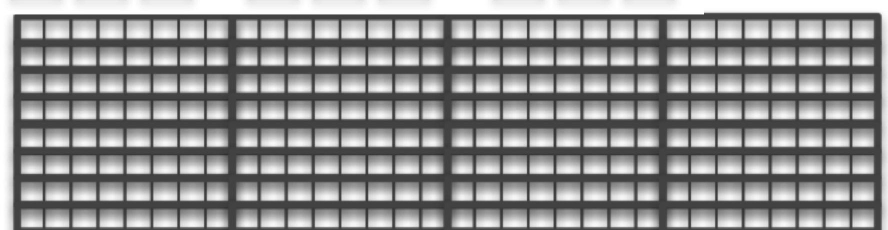
How to maintain?

- linked list of free blocks
  - inefficient (why?)

- linked list of metadata blocks that in turn point to free blocks
  - simple and efficient



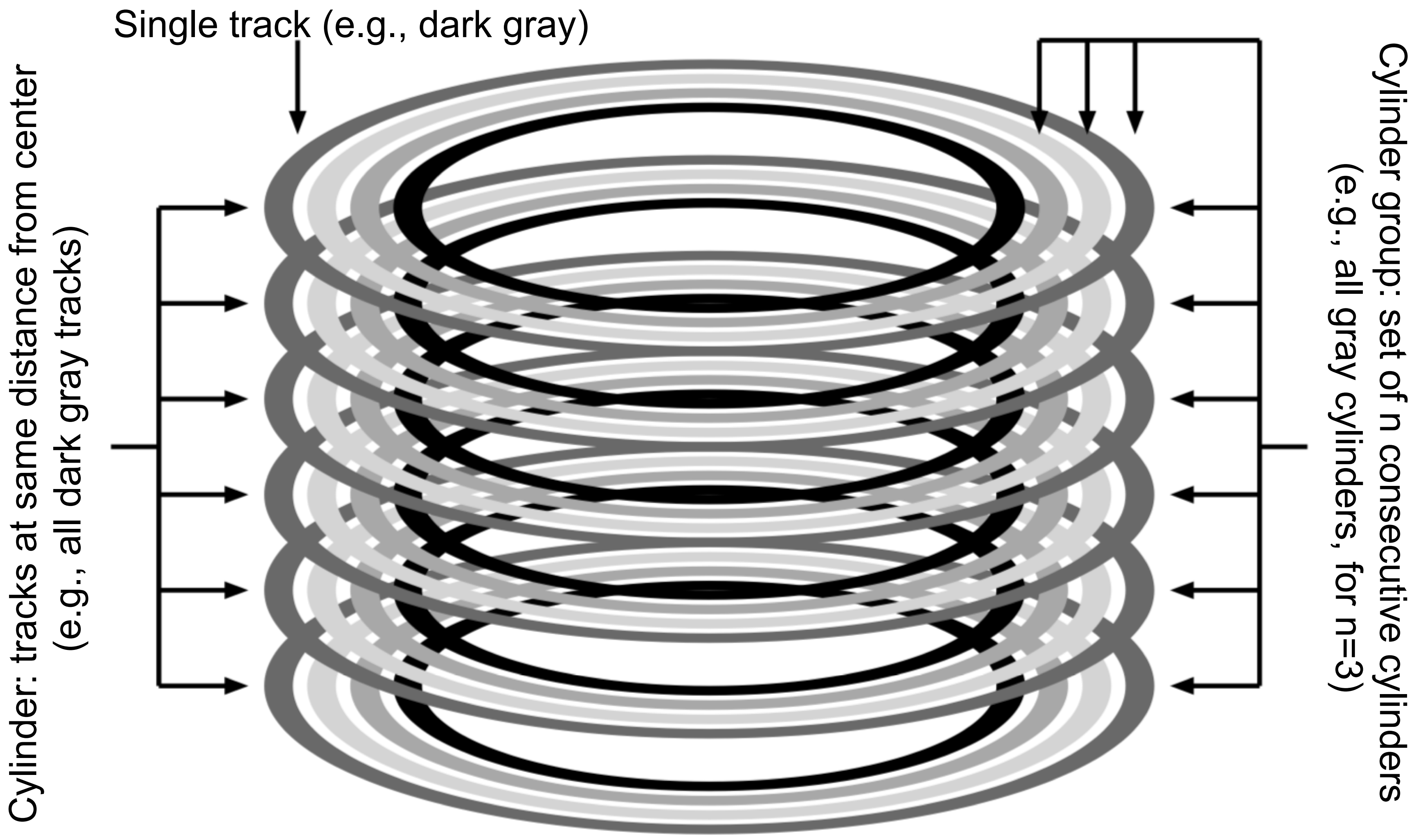
- bitmap
  - good because…



Problem 1: Poor Performance

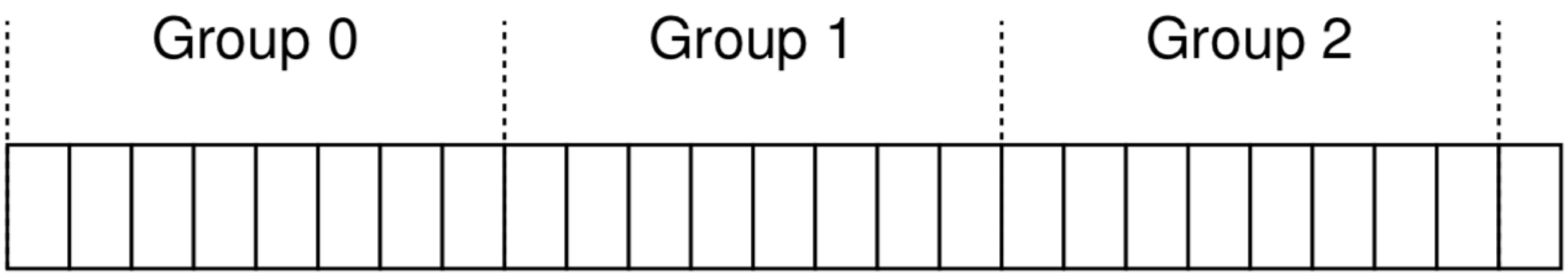
- In a naïve implementation of FFS, performance starts bad and gets worse
- One early implementation delivered only 2% disk bandwidth
- The root of the problem: poor locality
  - data blocks of a file were often far from its inode
  - file system would end up highly fragmented: accessing a logically continuous file would require going back and forth across the

Solution 1: Disk Awareness

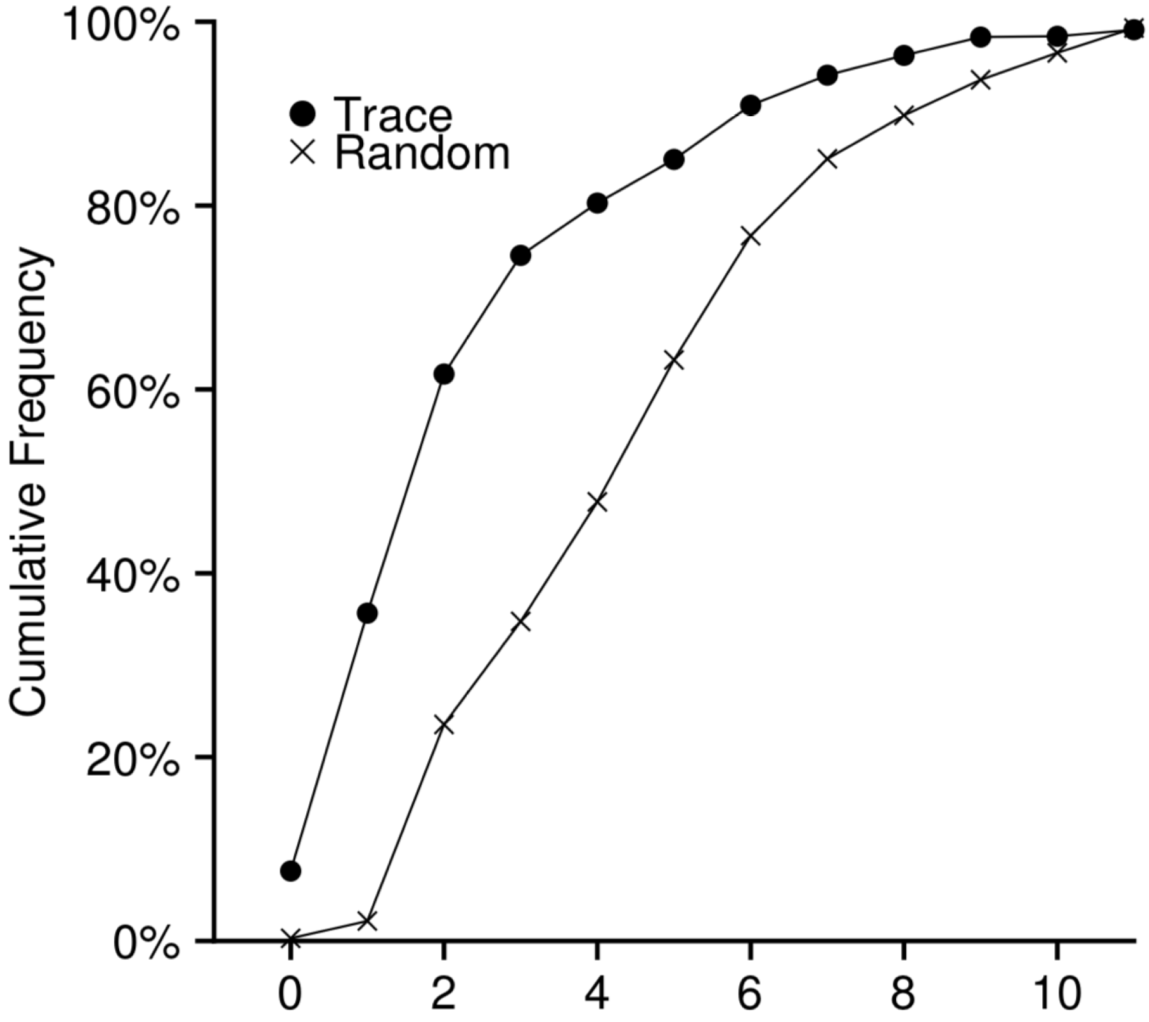


Abstracting Disk Awareness

- modern drives export a logical address space of blocks that are (temporally) close
- modern versions of FFS (ext2, ext3, ext4) organize the drive into block groups composed of consecutive portions of the disk's logical address space

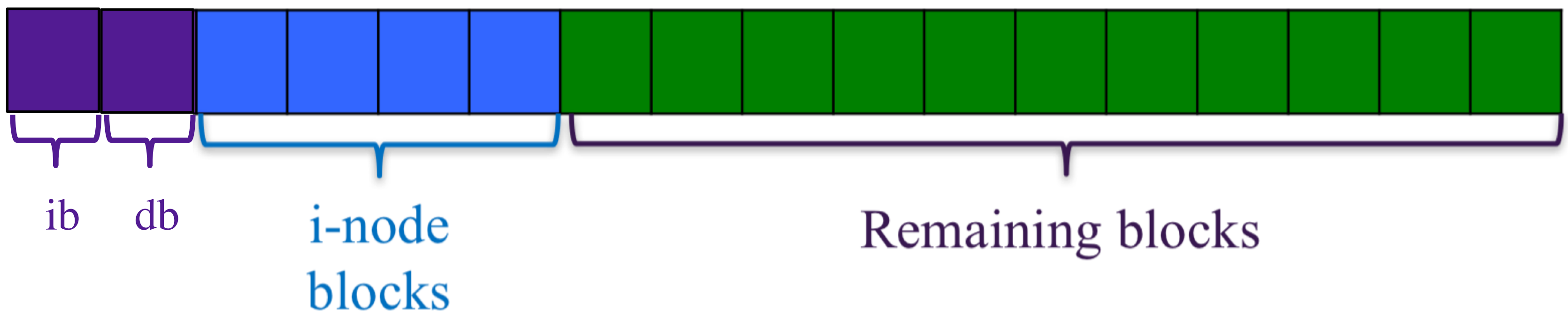


Locality in File System Accesses



Allocating Blocks

- FFS manages allocation per block group
- A per-group inode bitmap (ib) and data bitmap (db)



- Allocating directories:
  - find a group with a low number of allocated directories & high number of free inodes; put the directory data + inode there
- Allocating files:
  - place a file in the same group as the directory that contains it; allocate inode and data in same group
  - uses first-fit heuristic
  - reserves ~10% space to avoid deterioration of first-fit

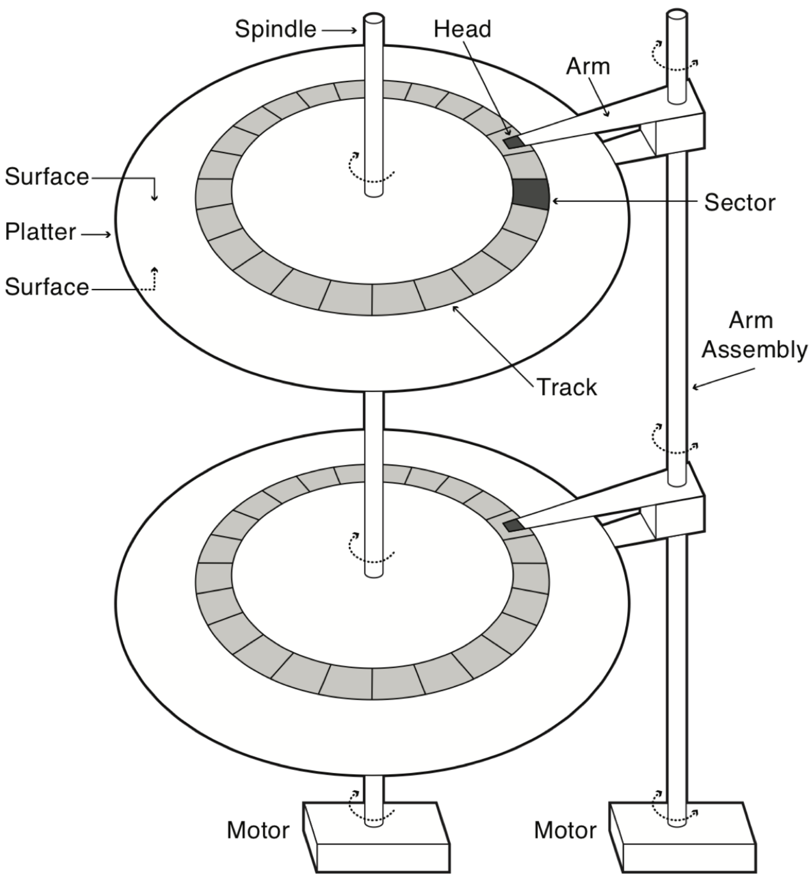
Solution 2: Page Cache

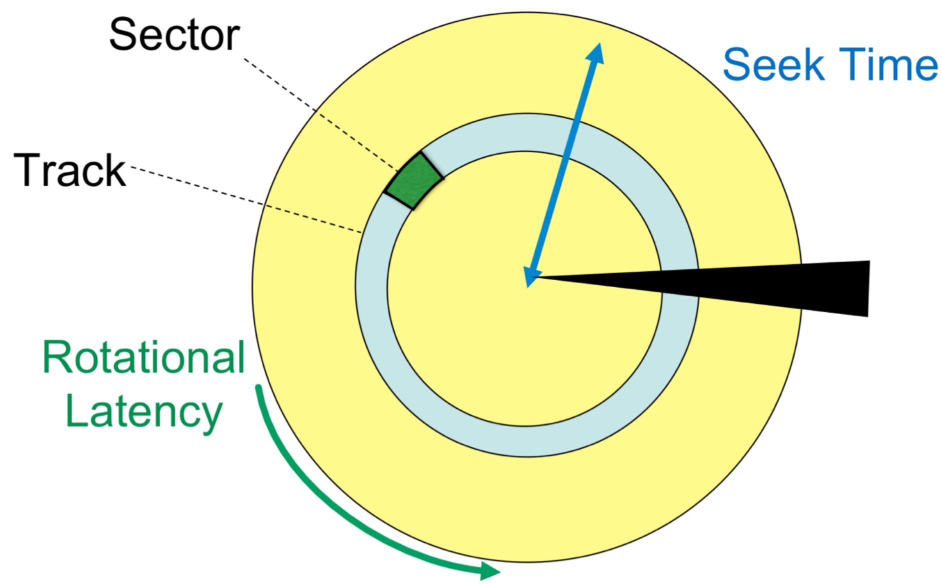
- To reduce costs of accessing files, most operating systems make aggressive use of caching
- page cache contains
  - heap and stack pages for each process
  - file data and metadata from devices (accessed with read() and write() calls)
  - memory-mapped files

What about writes?

- need to durably store data means writes often dominate performance
- small writes are expensive

Writing on Magnetic Disks



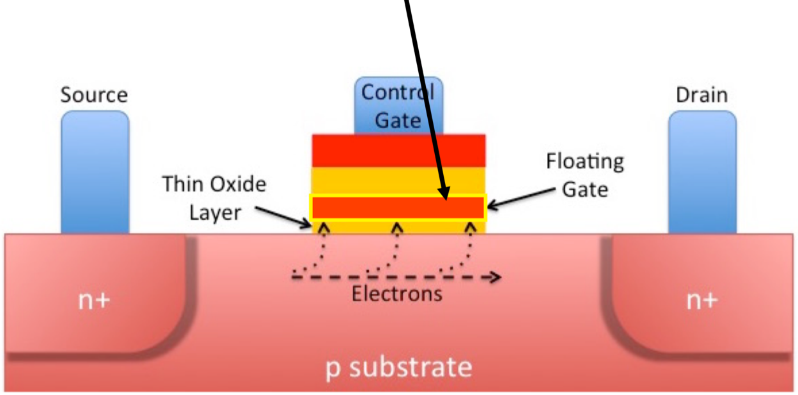


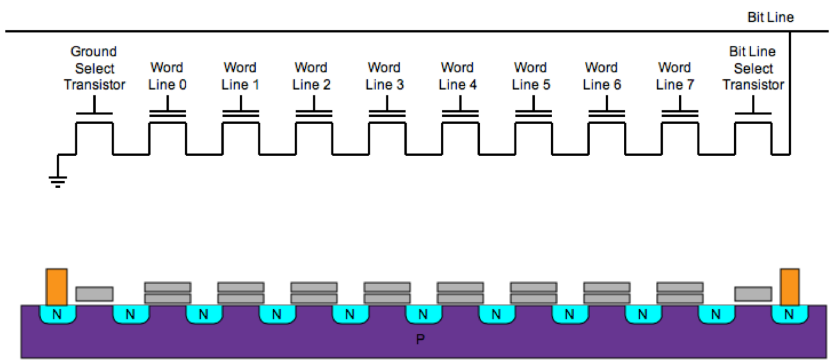
- **Seek:** to get to the track (1-15ms)
- **Rotational Latency:** to get to the sector (2-8ms)
- **Transfer:** get bits off the disk (.005ms/512-byte sector)

Writing on Flash Disks (SSDs)

Charge is stored in Floating Gate

(can have Single and Multi-Level Cells)





- can't write 1 byte/word (must write whole blocks)
- limited # of erase cycles per block (memory wear)
  - 103-106 erases and the cell wears out
- reads can "disturb" nearby words and overwrite them with garbage

Solution 1: Copy-on-write (COW)

- key idea: never overwrite files or directories in place; write new copy of updated version to previously unused location on disk.

- also used to optimize copies from fork(), exec(), etc.

Solution 2: write buffering

- page cache tracks if each page is "dirty" (aka modified)
- dirty pages are periodically flushed to disk

+ amortizes write overhead
+ allows re-ordering of disk accesses
- can introduce inconsistency in the event of crash failures