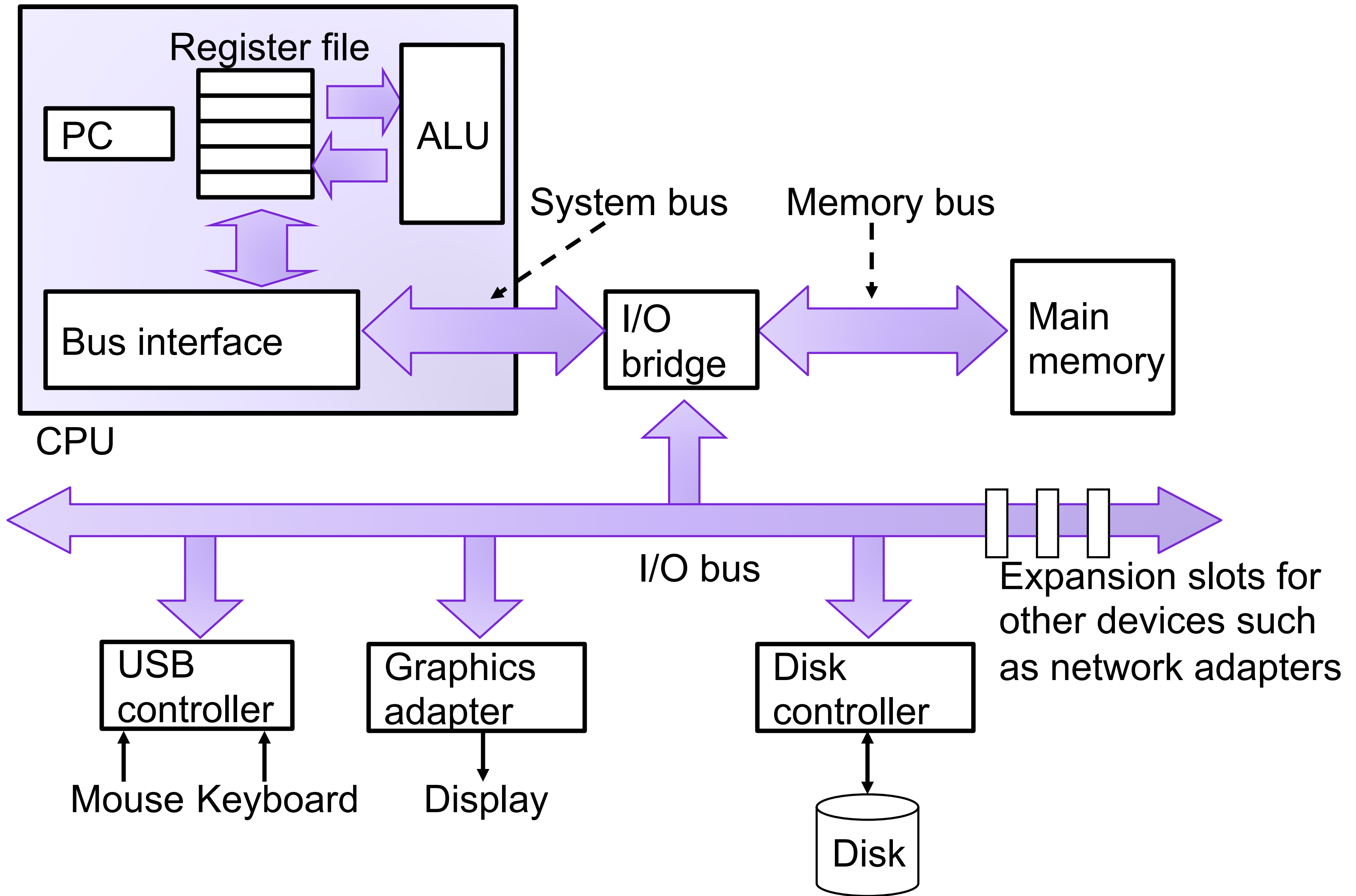
Lecture 29: File Systems (cont'd)

CS 105

May 1, 2019

Input and Output



File System 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

Storing Files

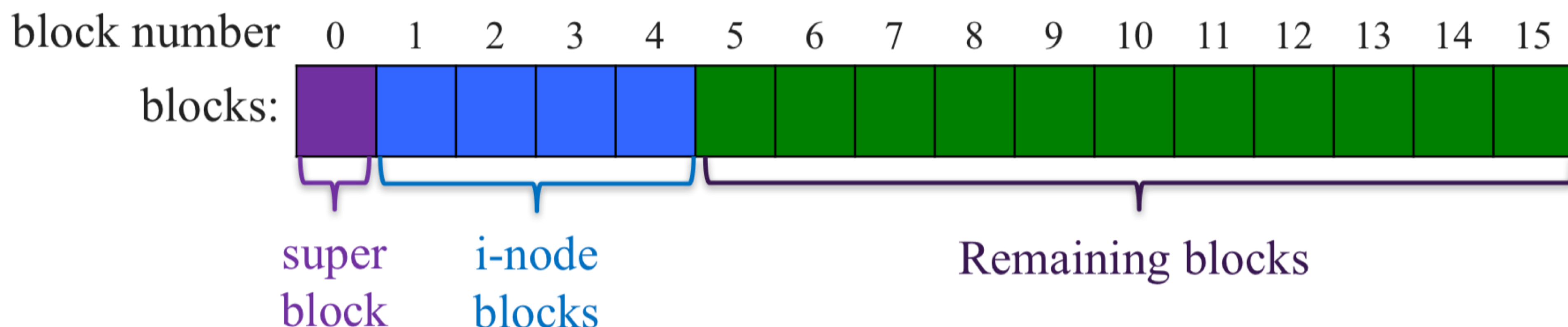
Possible ways to allocate files:

- **Continuous allocation:** all bytes together, in order
- **Linked structure:** each block points to the next block
- **Indexed structure:** index block points to many other blocks

Indexed Allocation: Fast File System (FFS)

- tree-based, multi-level index

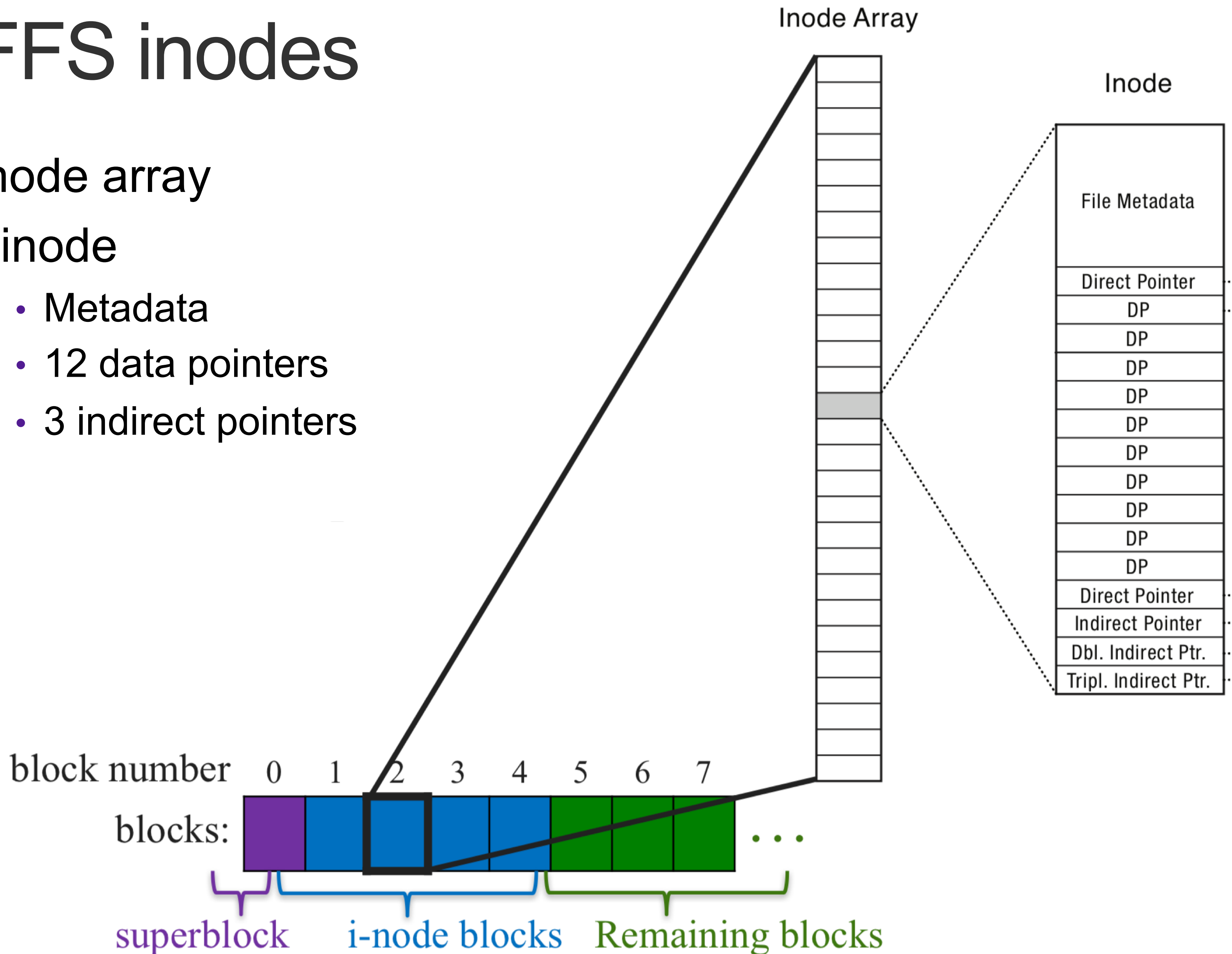
- **superblock** identifies file system's key parameters
- **inodes** store metadata and pointers
- **datablocks** store data



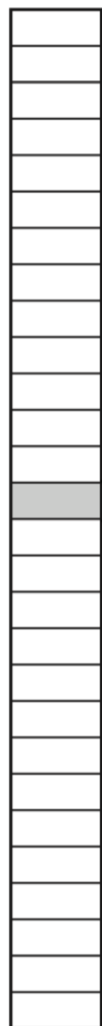
FFS inodes

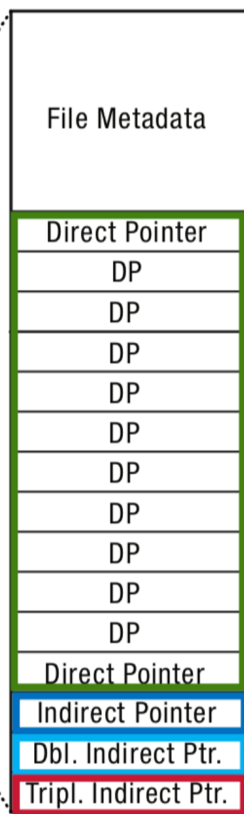
inode array

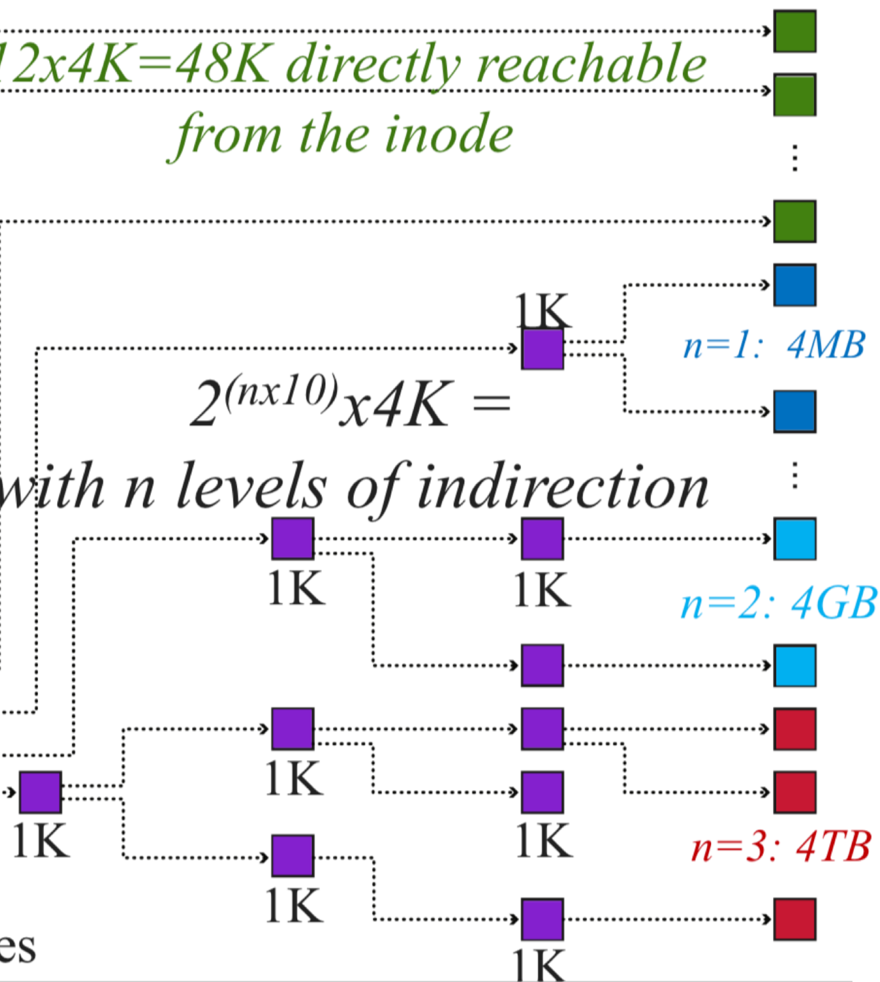
- inode
  - Metadata
  - 12 data pointers
  - 3 indirect pointers



FFS Index Structures







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

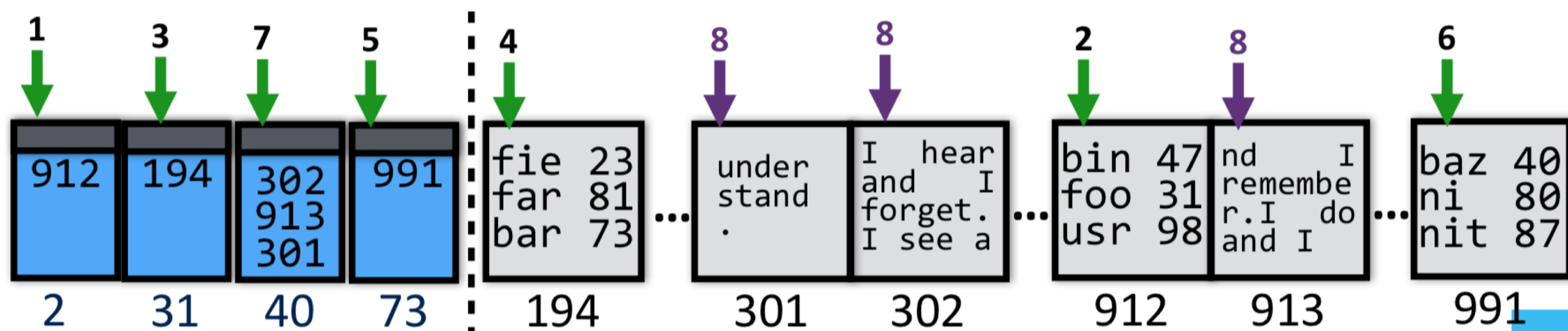
FFS Directory Structure

- Originally: array of 16 byte entries
  - 14 byte file name
  - 2 byte i-node number
- Now: linked lists. Each entry contains:
  - 4-byte inode number
  - Length of name
  - Name (UTF8 or some other Unicode encoding)
- First entry is ".", points to self
- Second entry is "..", points to parent inode

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, 913, 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

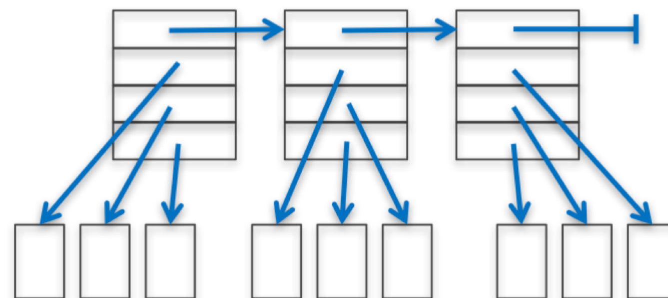
Free List

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

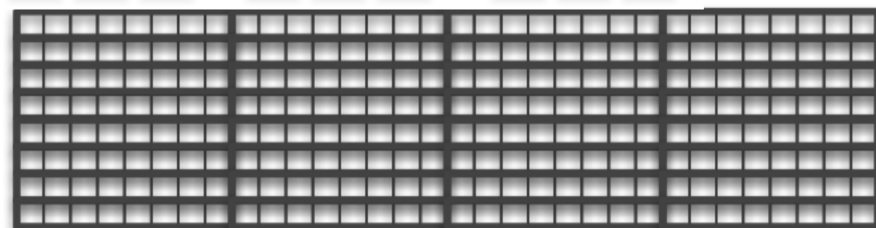
How to maintain?

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

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



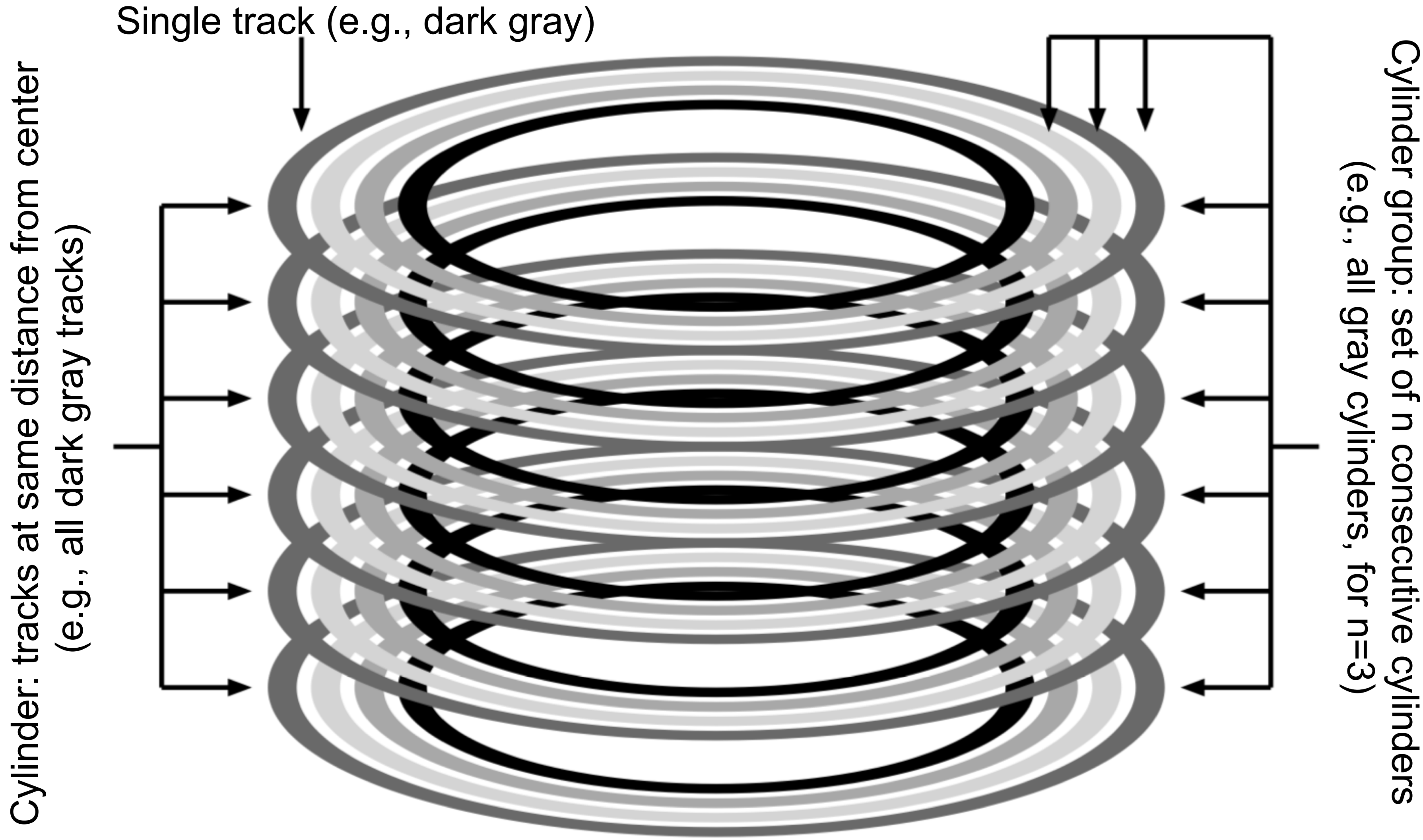
- bitmap
  - good because…



Problem 1: Poor Performance

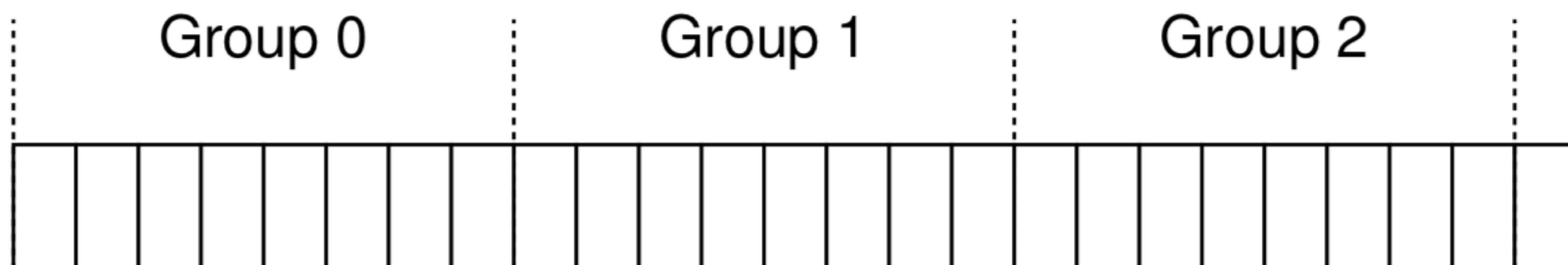
- In a naïve implementation of FFS, performance starts bad and gets worse
- One early implementation delivered only 2% disk bandwidth
- The root of the problem: poor locality
  - data blocks of a file were often far from its inode
  - file system would end up highly fragmented: accessing a logically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

A Solution: Disk Awareness

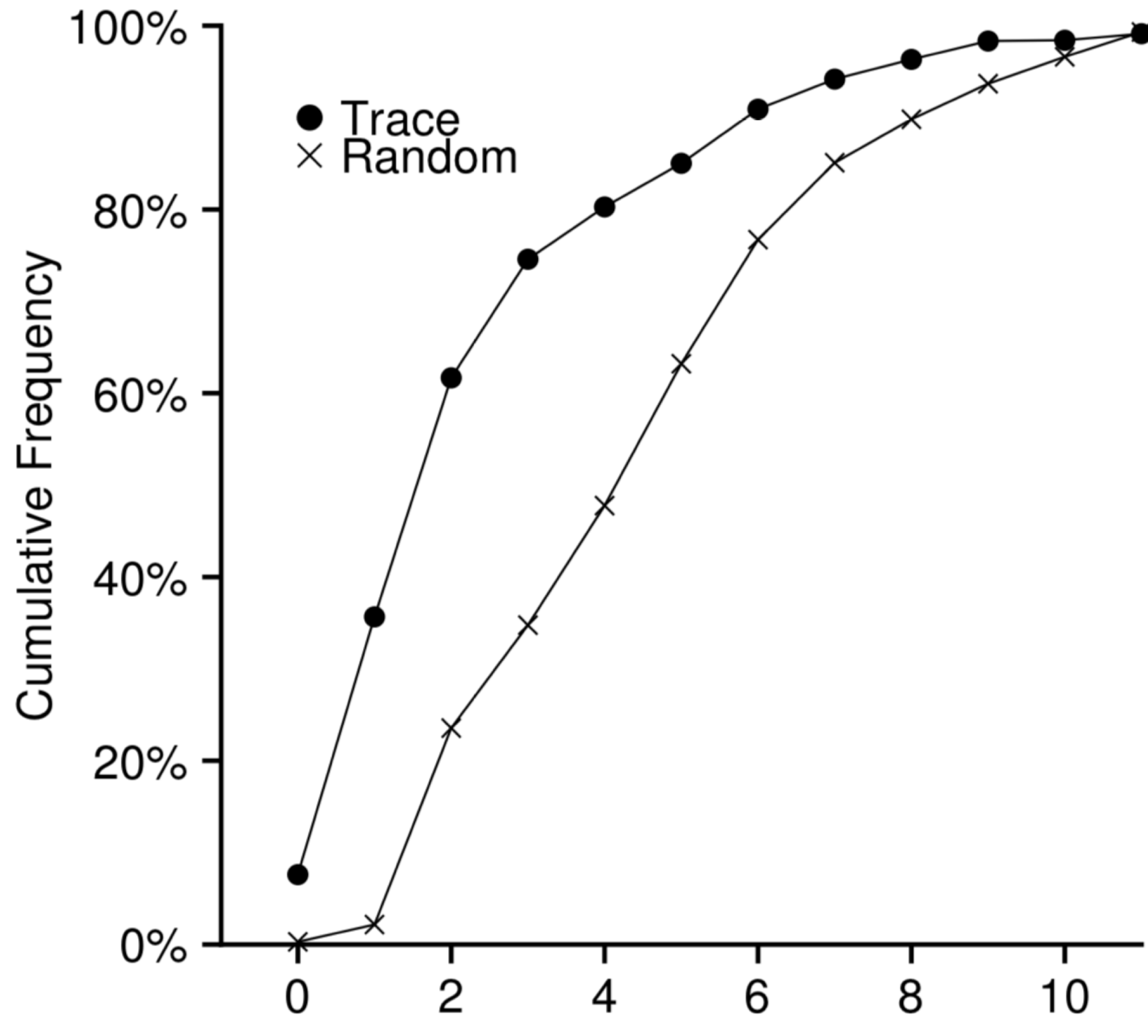


Abstracting Disk Awareness

- 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

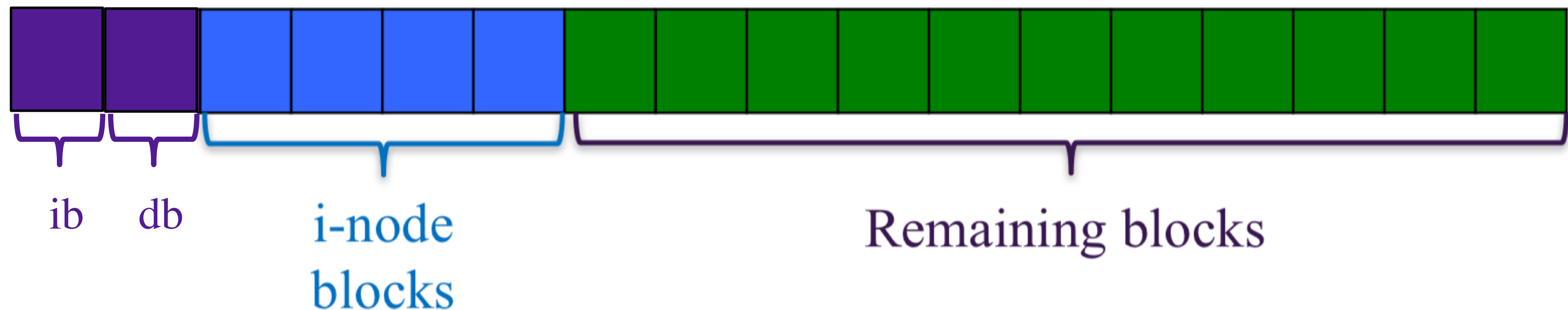


Locality in File System Accesses



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 files 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

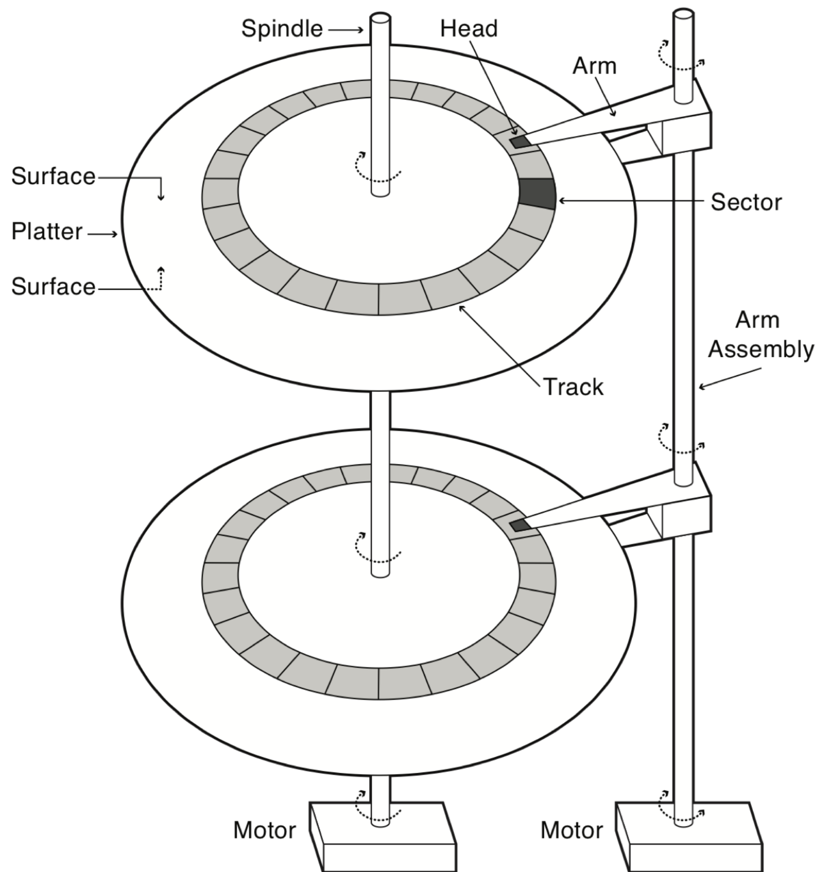
Page Cache

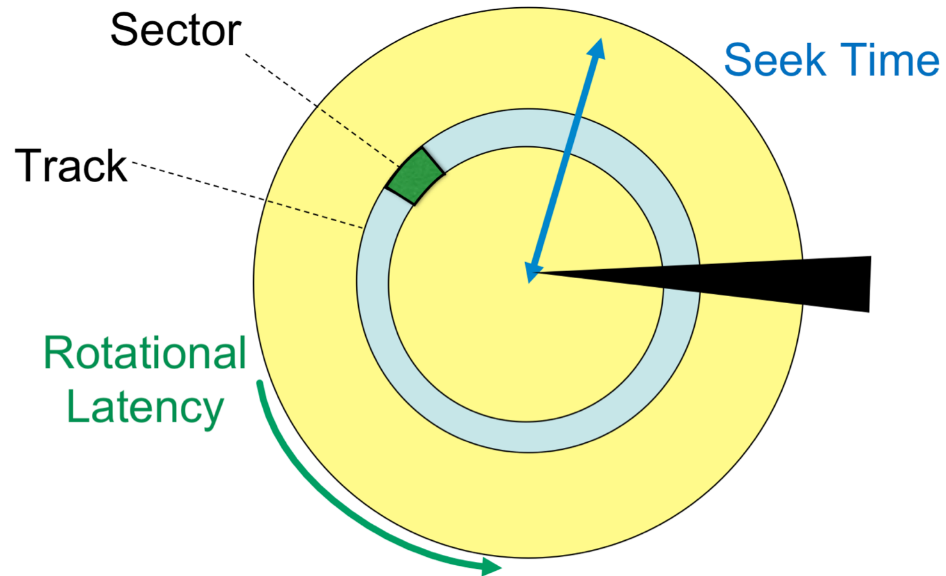
- To reduce costs of accessing files, most operating systems make aggressive use of caching
- page cache contains
  - heap and stack pages for each process
  - file data and metadata from devices (accessed with read() and write() calls)
  - memory-mapped files

What about writes?

- page cache tracks if each page is "dirty" (aka modified)
- dirty pages are periodically flushed to disk
- need to durably store data means writes often dominate performance
- small writes are expensive

Writing on Magnetic Disks

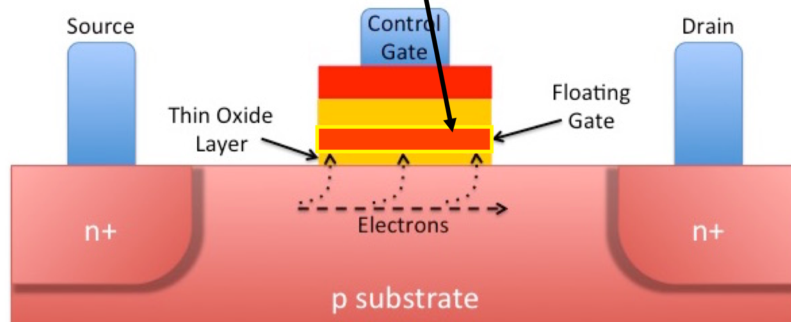


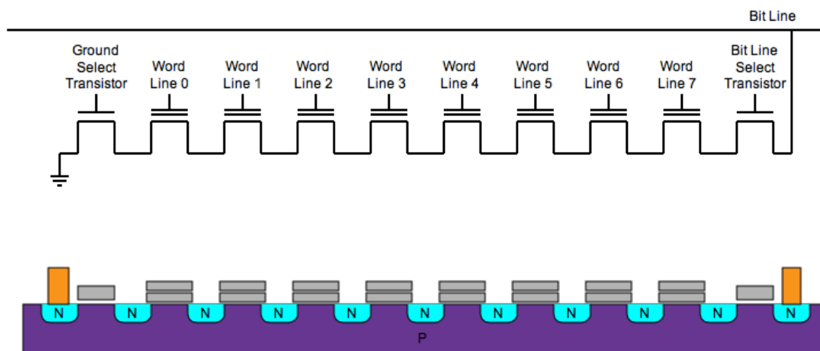


- **Seek:** to get to the track (1-15ms)
- **Rotational Latency:** to get to the sector (2-8ms)
- **Transfer:** get bits off the disk (.005ms/512-byte sector)

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

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.

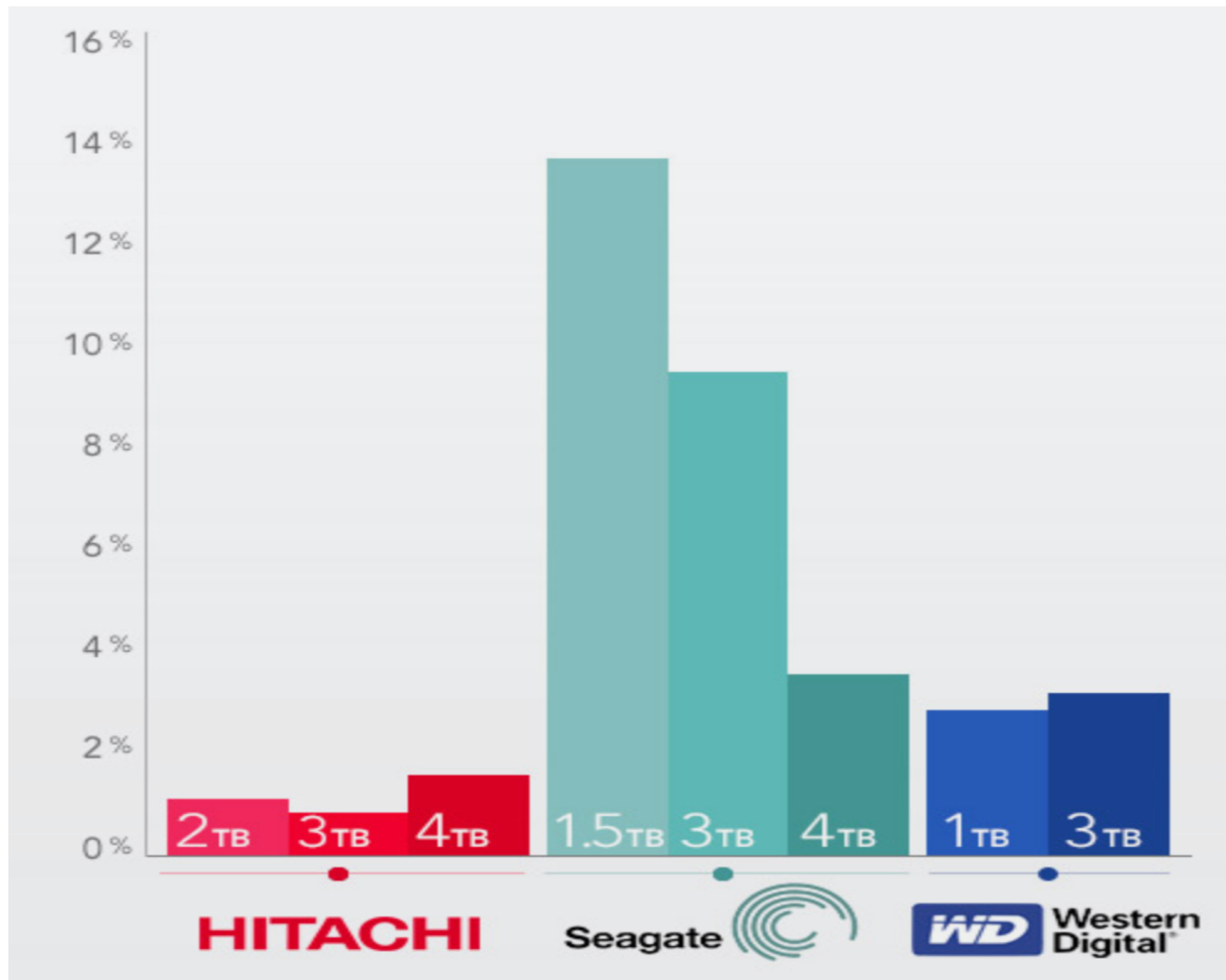
- also used to optimize copies from fork(), exec(), etc.

Problem 2: Poor Reliability

- Goal: must store data (correctly!) for long periods of time
- Reality: disks aren't perfect

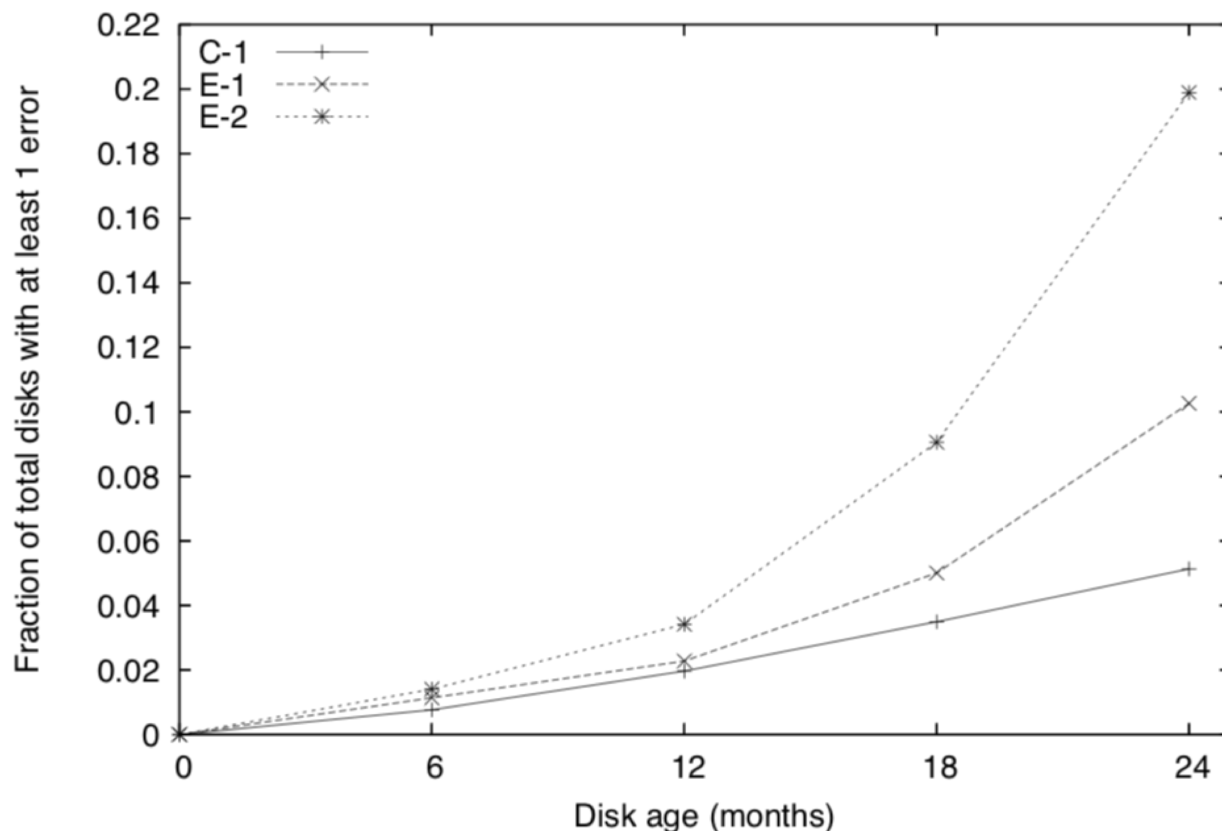
Full Disk Error

- Damage to disk head, electronic failure, wear out



Latent-Sector Errors

- latent-sector errors arise when a disk sector (or group of sectors) has been damaged in some way
- Example: head crash



Data Corruption

- data corruption can be caused by write interference, head height, leaked charge, cosmic rays, etc.
- approximately one sector will be corrupted per 10^14 bits read (about a 2% chance if you read a 2TB disk)

Error Correcting Codes

- an error-correcting code is a rendundant encoding of data that allows information to be recovered from a corrupted copy

- used by disks to automatically correct for disk errors

- balances storage overhead versus error rate

Checksums

- a checksum is the result of a function that takes a chunk of data (e.g., a 4KB block) and returns a short summary (e.g., 4 or 8 bytes)

- Example:
  - xor
  - cyclic redundancy check (CRC)

- File systems can store checksums for metadata and/or file contents

RAID

- a redundant array of inexpensive disks (RAID) is a system that spreads data redundantly across multiple disks in order to tolerate individual disk failures