File Systems

System I/O as a Uniform Interface

Operating systems use a uniform system I/O interface for all I/O devices

Commands to read and write to a file descriptor are the same no matter what type of "file"

Types of files include

- File (input/output)
- Keyboard (input)
- Screen (output)
- Pipe (input/output)
- Network (input/output
- Etc.

Our First I/O System: File Systems

Long-term information storage goals

- Should be able to store large amounts of information
- Information must survive processes, power failures, etc.
- Processes must be able to find information
- Needs to support concurrent accesses by multiple processes

Solution: the file system abstraction

- Interface that provides operations involving: Files and Directories
    - Directories are just a special kind of file

The File System Abstraction

Interface that provides operations on data stored long-term on disk

A file is a named sequence of stored bytes

- Name is defined on creation
- Processes use name to subsequently access that file

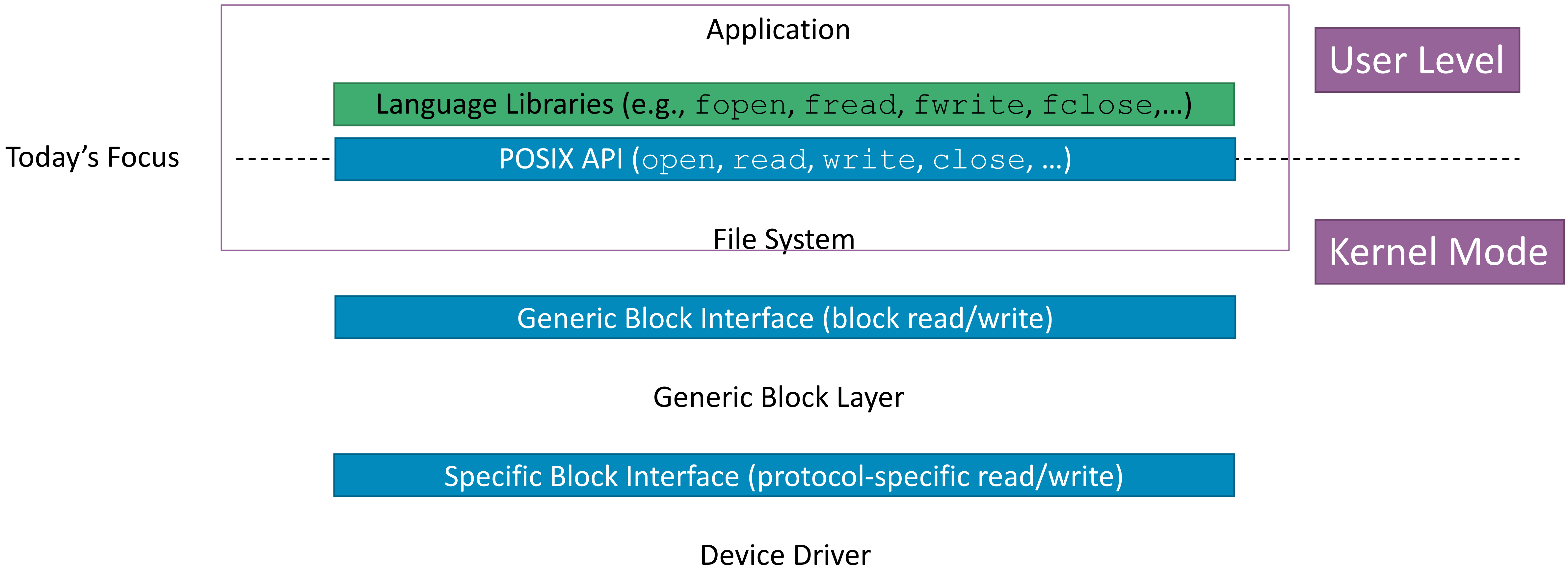
A file comprises two parts:

- Data: information a user or application puts in a file (an array of untyped bytes)
- Metadata: information added and managed by the OS (e.g., size, owner, security info, modification time)

Two types of files

- Normal files: data is an arbitrary sequence of bytes
- Directories: a special type of file that provides mappings from human-readable names to low-level names (i.e., File numbers)

The File System Stack



File System Challenges

- **Performance:** despite limitations of disks
- **Flexibility:** need to support diverse file types and workloads
- **Persistence:** store data long term
- **Reliability:** resilient to OS crashes and hardware failures

Common File System Properties

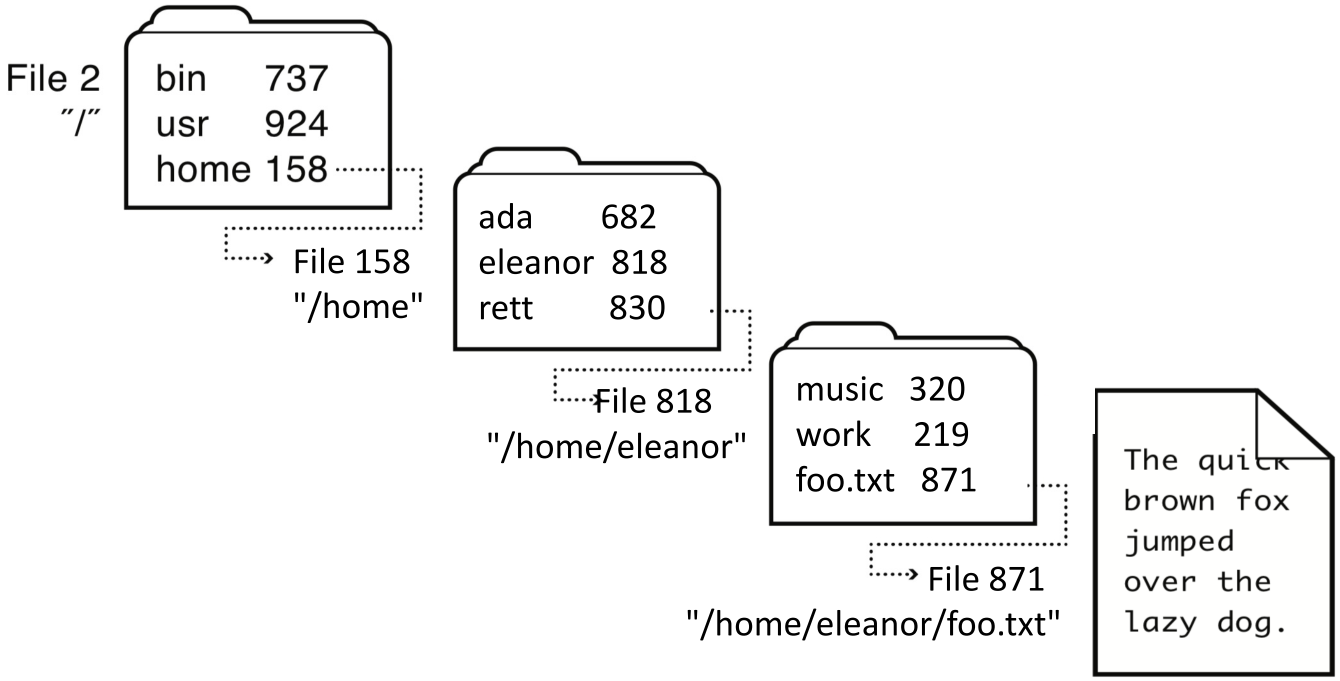
- Most files are small
  - Need strong support for small files (optimize the common case)
  - Block size can't be too big, or we’ll waste space
- Directories are typically small
  - Usually, 20 or fewer entries
- Some files are very large
  - Must handle large files
  - Large file access should be reasonably efficient
- File systems are usually about half full

Multiple human-readable names

- Many file systems allow a given file to have multiple names
- Hard links are multiple file directory entries that map different path names to the same file number
- Symbolic Links or soft links are directory entries that map one name to another (effectively a redirect)
- Directories: file name -> low-level names (i.e., file numbers or indices)

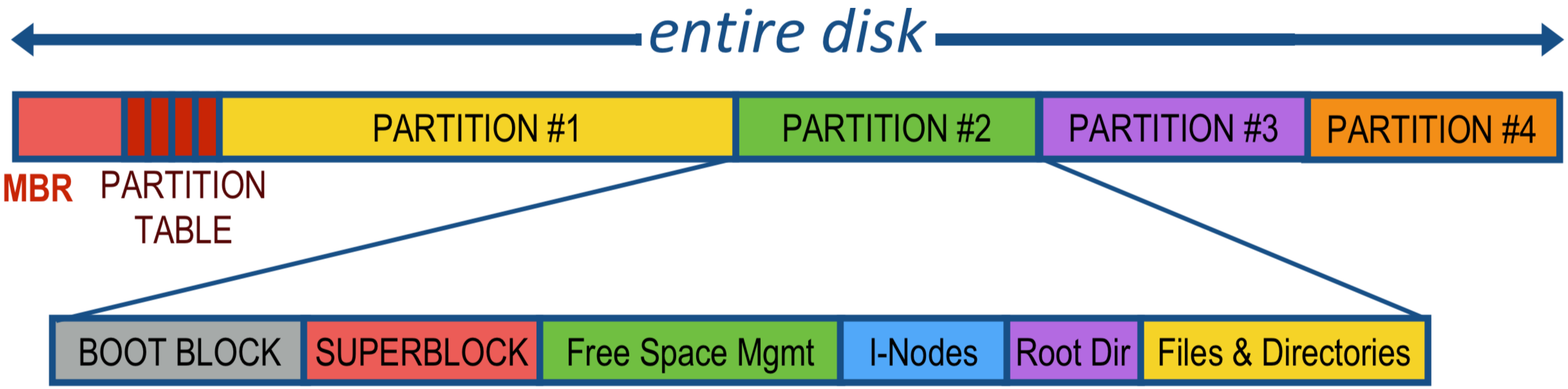
Directories

- A **directory** is a file that provides mappings from human-readable names to low-level names (i.e., file numbers):
  - A list of human-readable names
  - A mapping from each name to a specific underlying file (including subdirectories)
- OSs use path name to find directories and files



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



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
- Log structure: sequence of segments, each containing updates

Which is the best?

- For sequential access?
- For random access?
- For small files?
- For large files?

Continuous Allocation

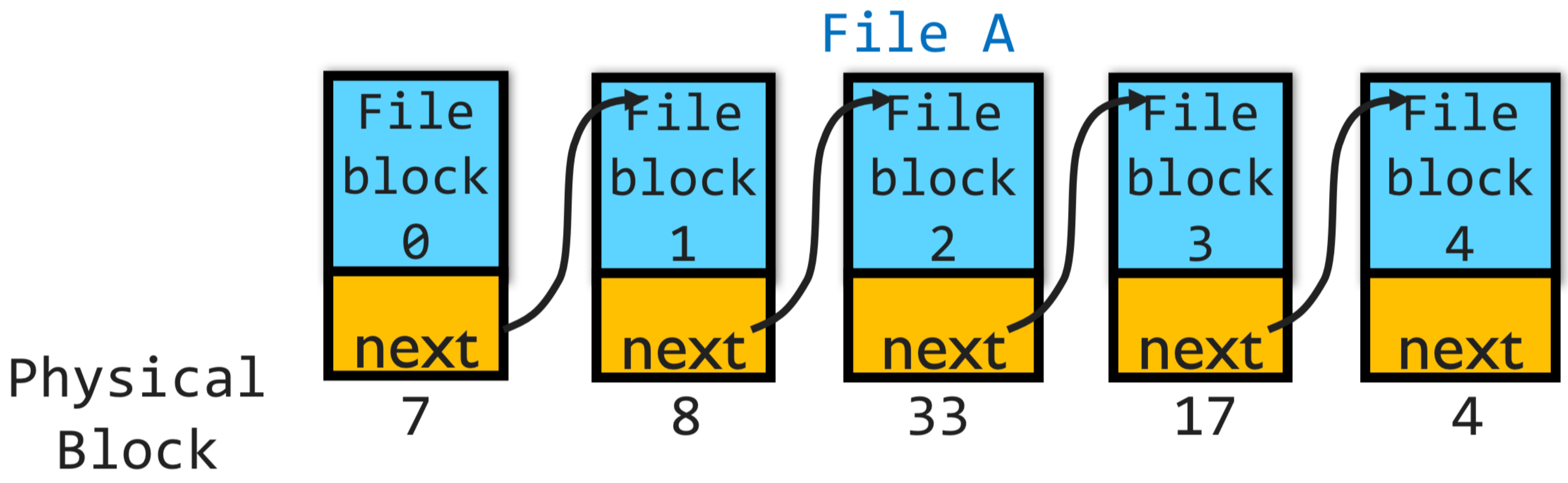
All bytes together, in order

- Simple: state required per file = start block & size
- Efficient: entire file can be read with one seek
- Fragmentation: external is bigger problem
- Usability: user needs to know size of file at time of creation

Linked Allocation

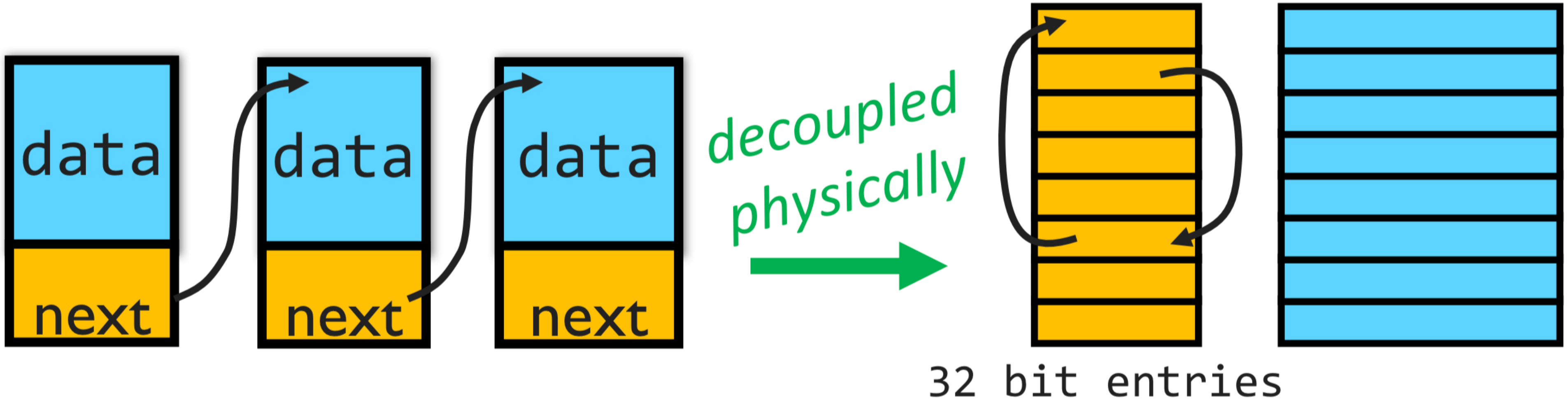
Each file is stored as linked list of blocks: First word of each block points to next block, rest of disk block is file data

- Simple: only need to store 1st block of each file
- Space Utilization: no space lost to external fragmentation
- Performance: random access inside a file is slow
- Space Utilization: overhead of pointers



Linked Allocation: File Allocation Table (FAT)

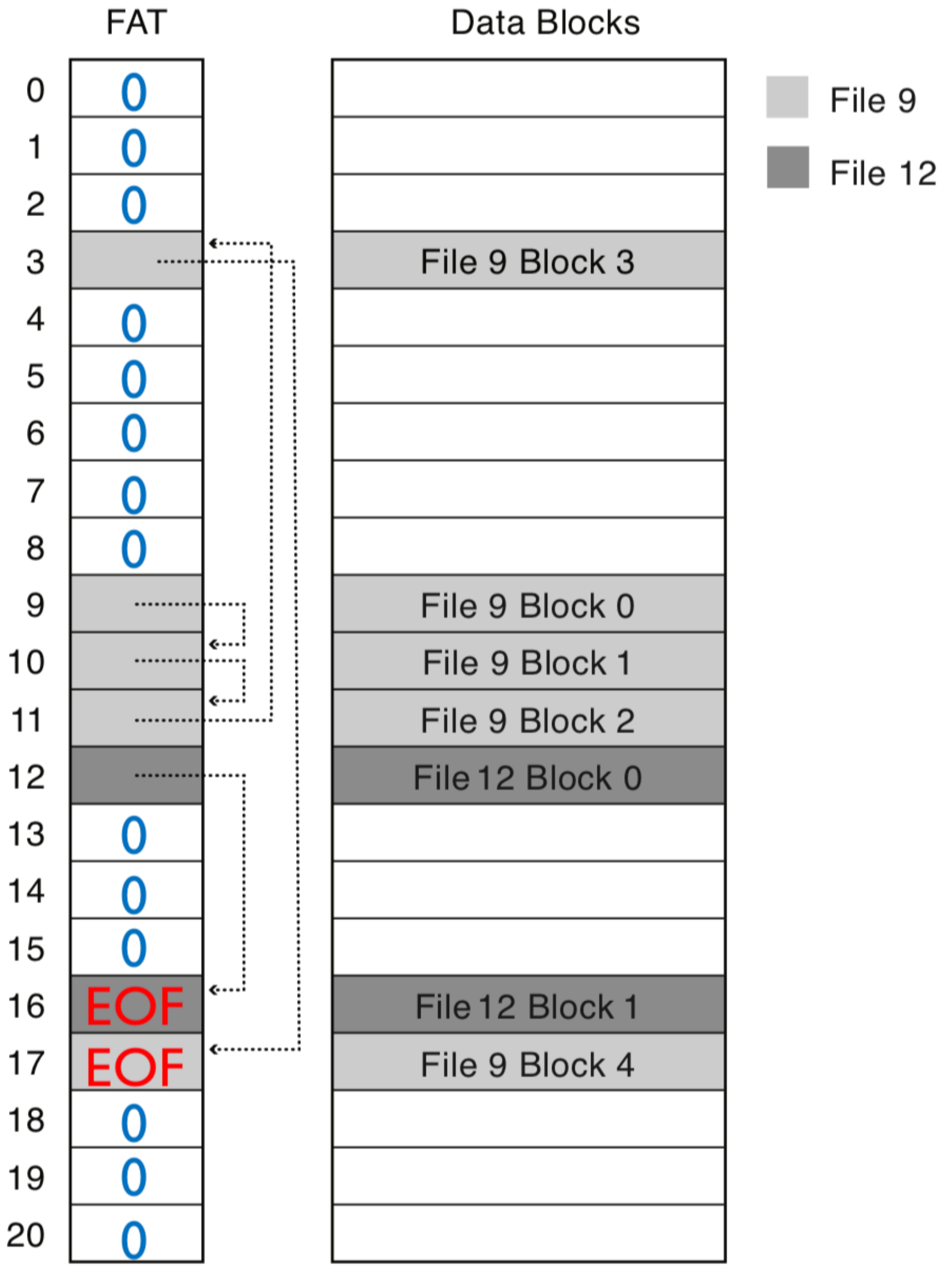
- Developed by Microsoft for MS-DOS
- Still widely used for flash drives, camera cards, etc.
- Fat-32 supports 2^{28} blocks and files of $2^{32} - 1$ bytes
- File table:
  - Linear map of all blocks on disk
  - Each file a linked list of blocks



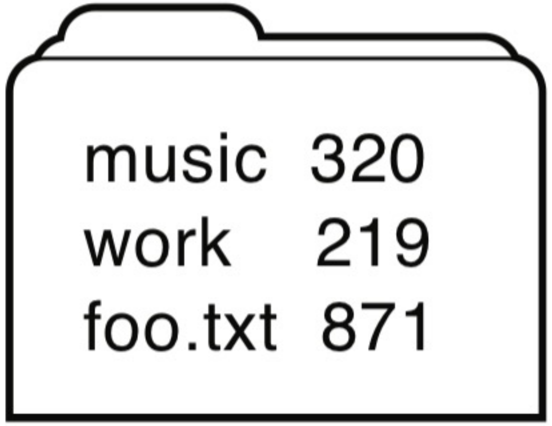
FAT File System

- 1 entry per block
- EOF for last block
- 0 indicates free block
- low-level name = FAT index of first block in file

Directory	
bart.txt	9
maggie.txt	12



FAT Directory Structure



Folder: a file with 32-byte entries

Each Entry:

- 8-byte name + 3-byte extension (ASCII)
- creation date and time
- last modification date and time
- first block in the file (index into FAT)
- size of the file

Exercise 1: Linked Allocation

- How many disk reads would be required to read (all of) a 2^{15} byte file named /foo/bar/baz.txt
  - assume 4096-byte (4 KB or 2^{12} byte) blocks
  - assume that all directories are small enough to fit in one block

Exercise 1: Linked Allocation

- How many disk reads would be required to read (all of) a 2^{15} byte file named /foo/bar/baz.txt
  - assume 4096-byte (4 KB or 2^{12} byte) blocks
  - assume that all directories are small enough to fit in one block

1. read / directory block, find foo's file number
2. read foo directory block, find bar's file number
3. read bar's directory block, find baz.txt's file number
4. read baz.txt's block 0
5. read ptr to block 1 in FAT
6. read baz.txt's block 1
7. read ptr to block 2 in FAT

…

15. read ptr to block 6 in FAT
16. read baz.txt's block 6
17. read ptr to block 7 in FAT
18. read baz.txt's block 7
19. read EOF ptr in FAT

Evaluating FAT

How is FAT good?

- Simple: state required per file: start block only
- Widely supported
- No external fragmentation
- block used only for data

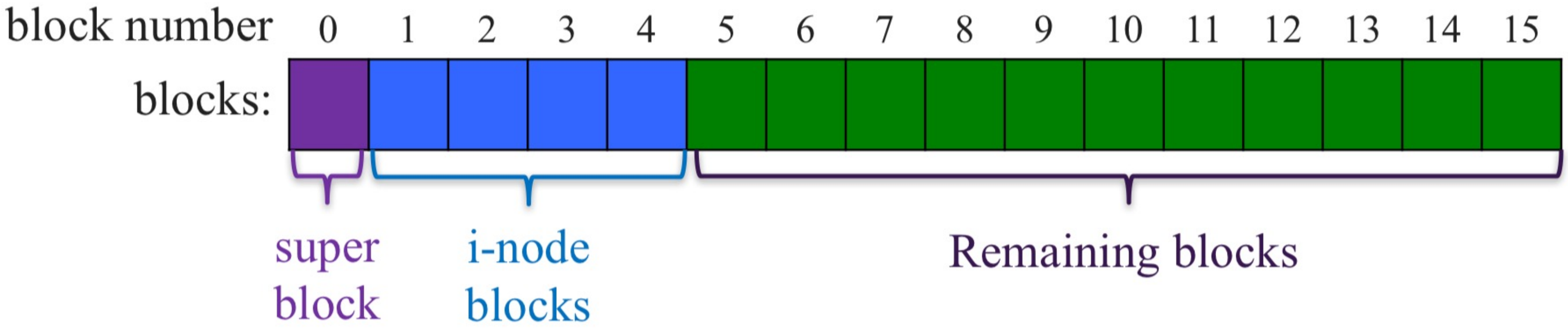
How is FAT bad?

- Poor locality
- Many file seeks (unless entire FAT in memory)
- Poor random access
- Limited metadata
- Limited access control
- Limitations on volume and file size
- No support for reliability techniques

Indexed Allocation: Fast File System (FFS)

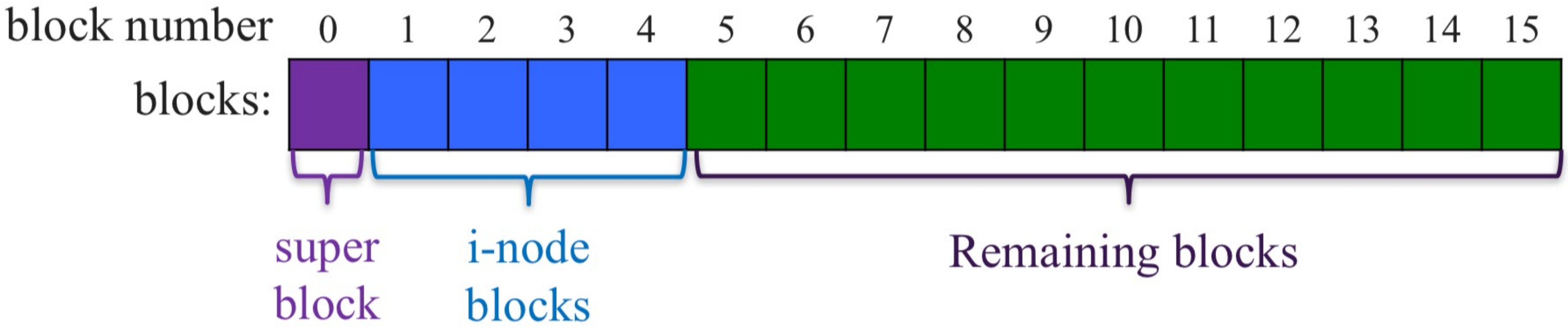
- tree-based, multi-level index

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



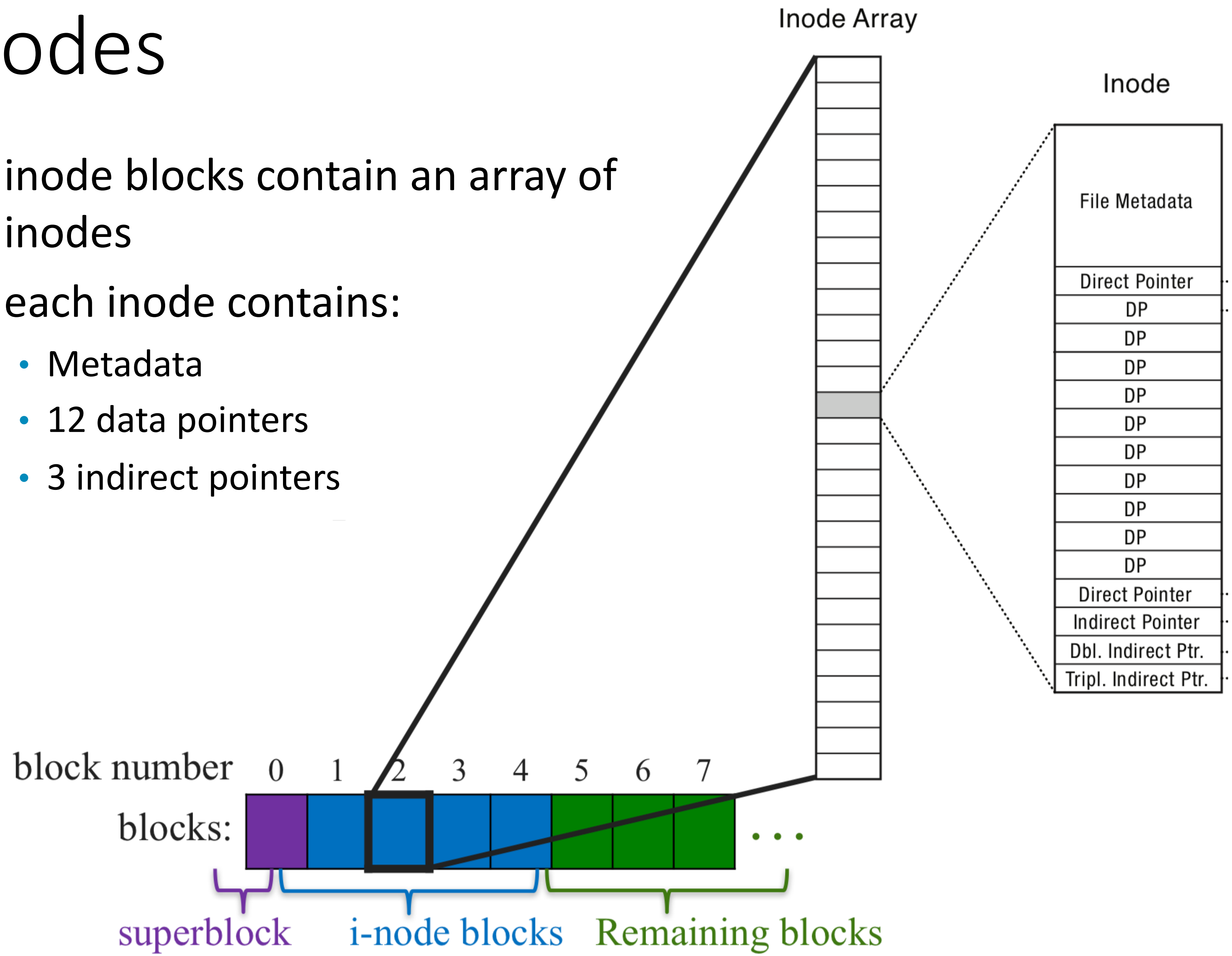
FFS Superblock

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



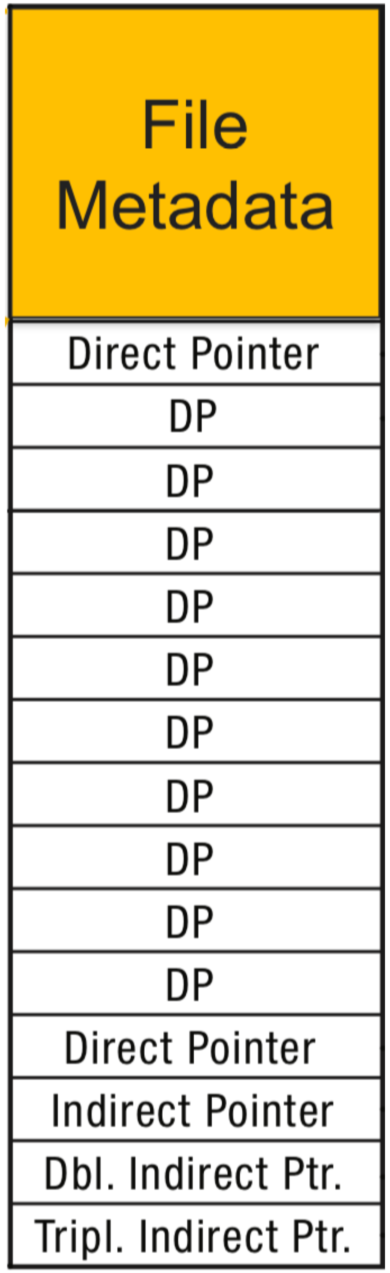
FFS inodes

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

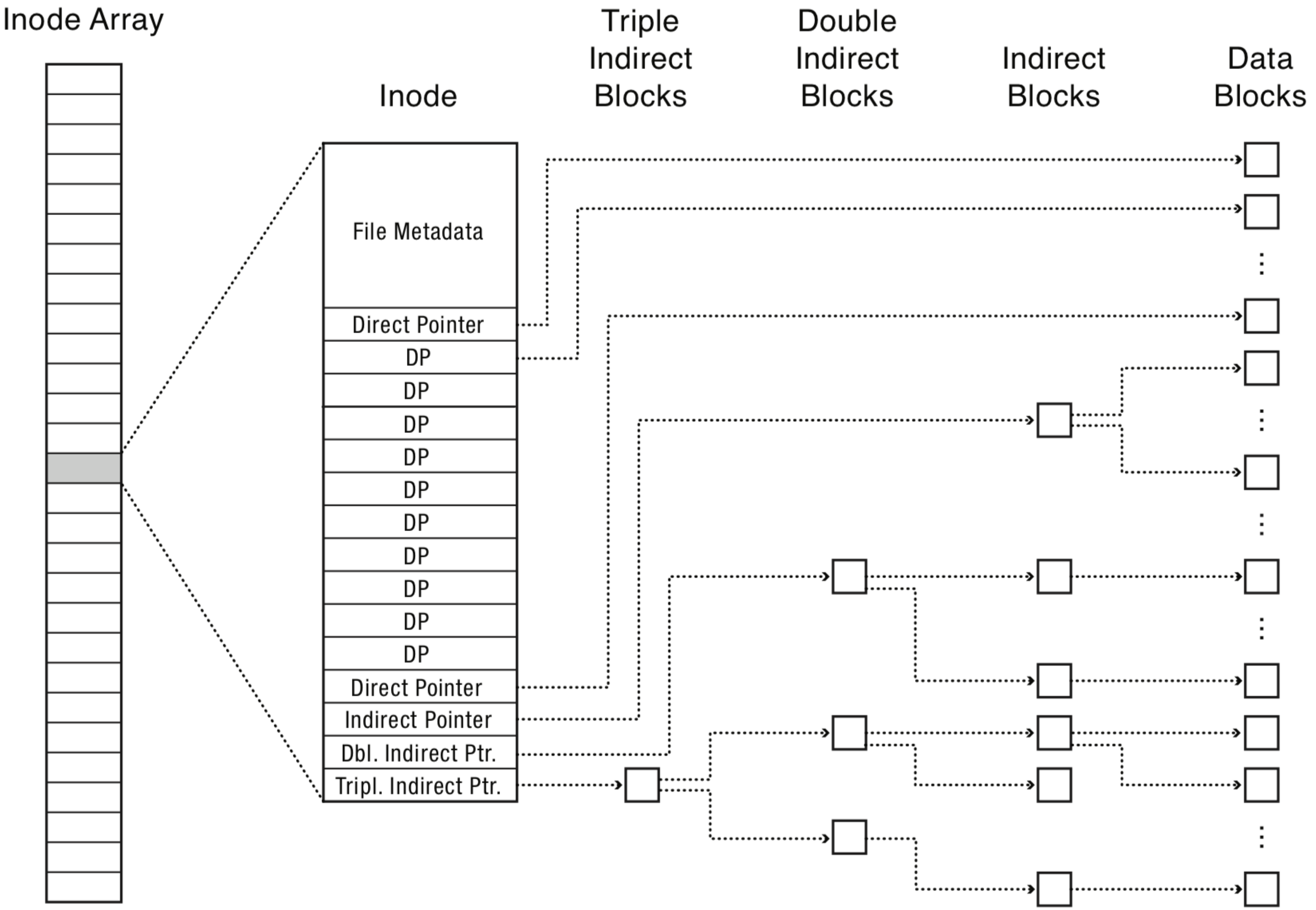


inode Metadata

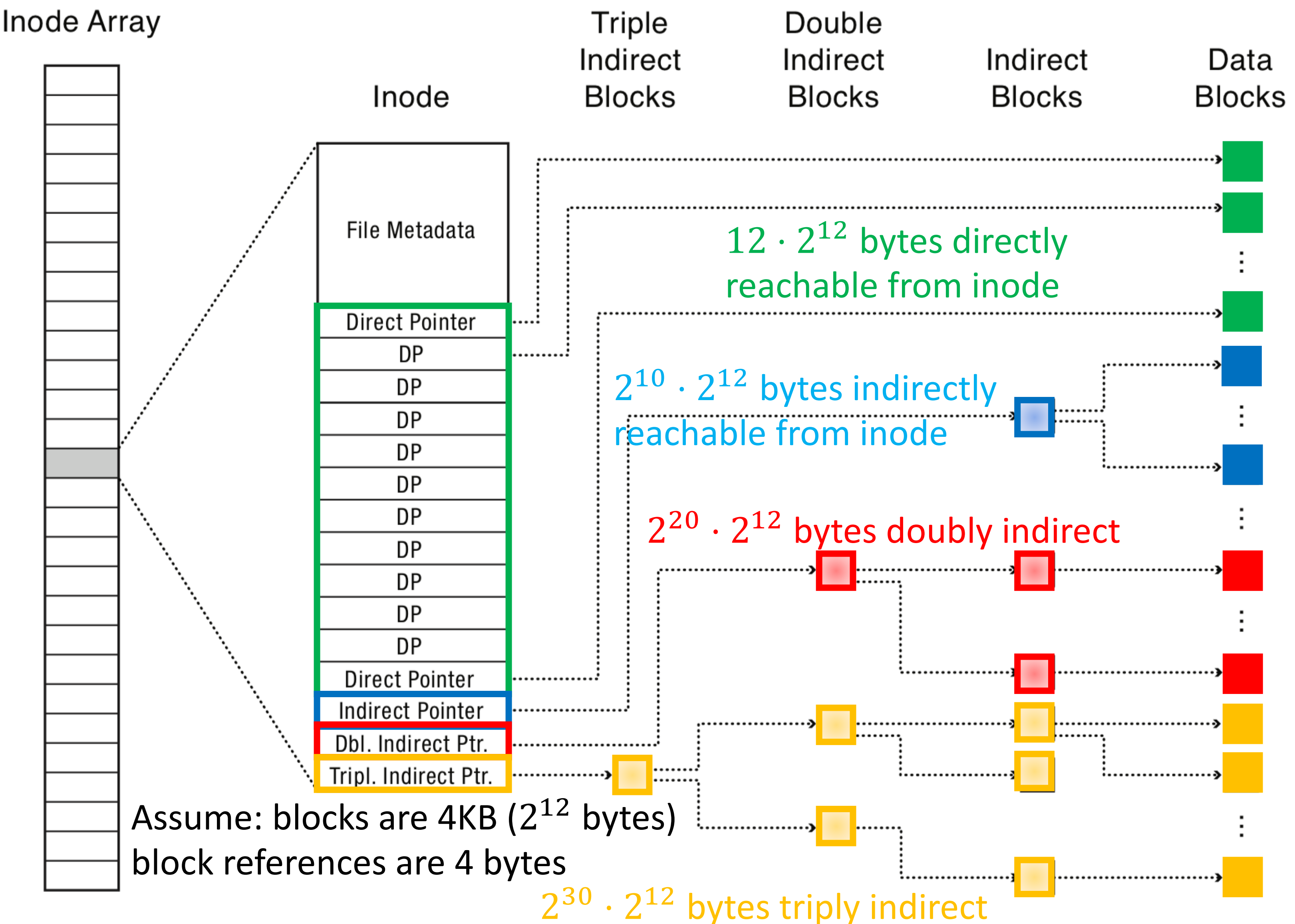
- Type
  - ordinary file
  - directory
  - symbolic link
  - special device
- Size of the file (in #bytes)
- # links to the i-node
- Owner (user id and group id)
- Protection bits
- Times: creation, last accessed, last modified



FFS Index Structures



FFS Index Structures



Exercise 2: Inode Structures

- Assume we are using the inode structure we just described and assume again that each block is 4K (2^{12}) and that each block reference is 4 bytes.

- Which pointers in the inode of a 32KB file would be non-null?

- Which pointers in the inode of a 47MB file would be non-null?

Exercise 2: Inode Structures

- Assume we are using the inode structure we just described and assume again that each block is 4K (2^{12}) and that each block reference is 4 bytes.

- Which pointers in the inode of a 32KB file would be non-null?

  the first 8 direct pointers

- Which pointers in the inode of a 47MB file would be non-null?

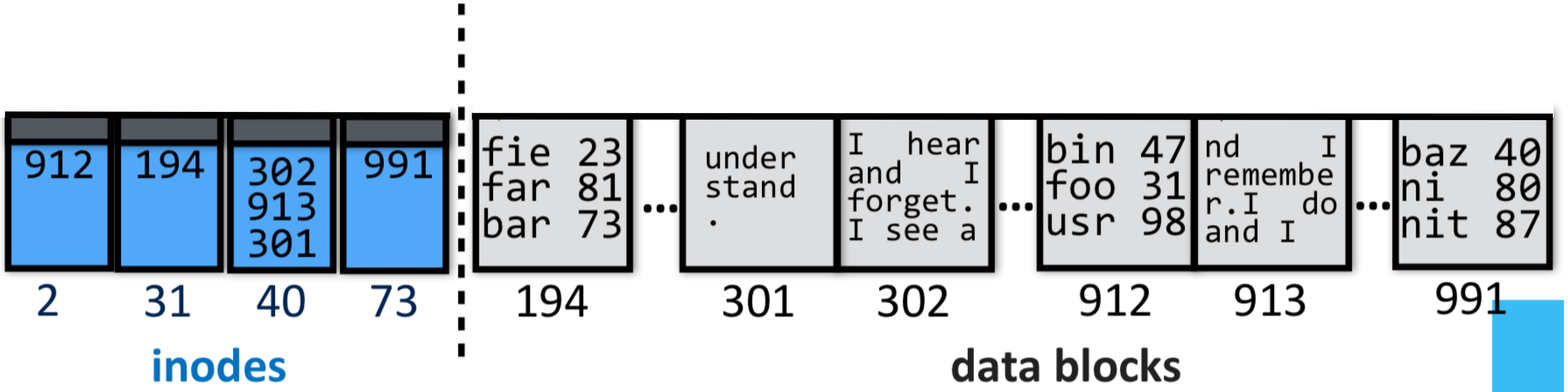
  all 12 direct pointers, the indirect pointer,
  and the doubly-indirect pointer

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 3: Indexed Allocation

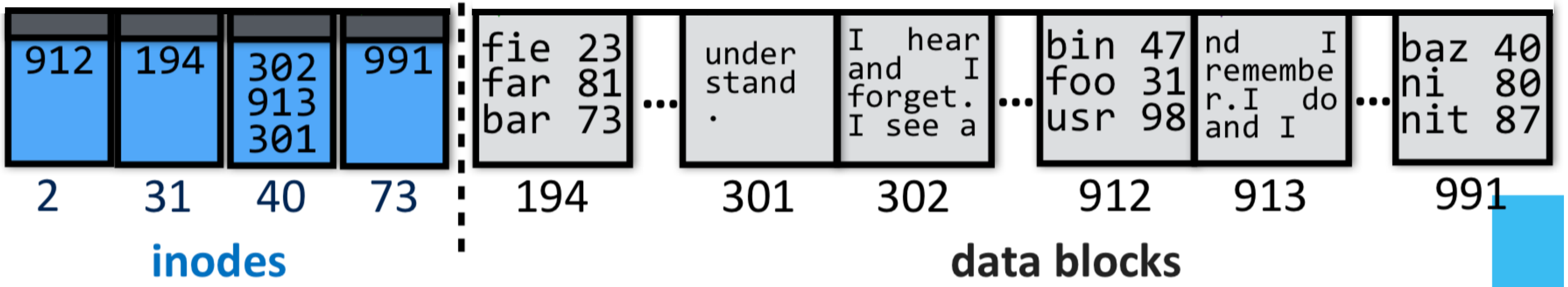
Which inodes and data blocks would need to be accessed to read (all of) file /foo/bar/baz?



Exercise 3: Indexed Allocation

Which inodes and data blocks would need to be accessed to read (all of) file /foo/bar/baz?

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

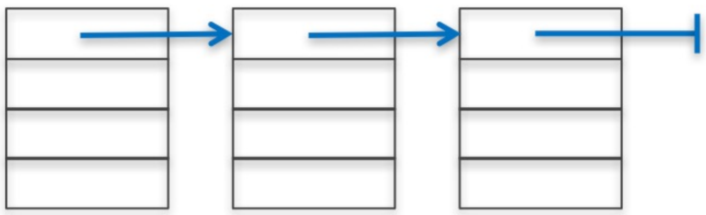
Implementation Basics

- Directories: file name -> low-level names (i.e., file numbers)
- Index structures: file number -> block
- Free space maps: find a free block (ideally nearby)

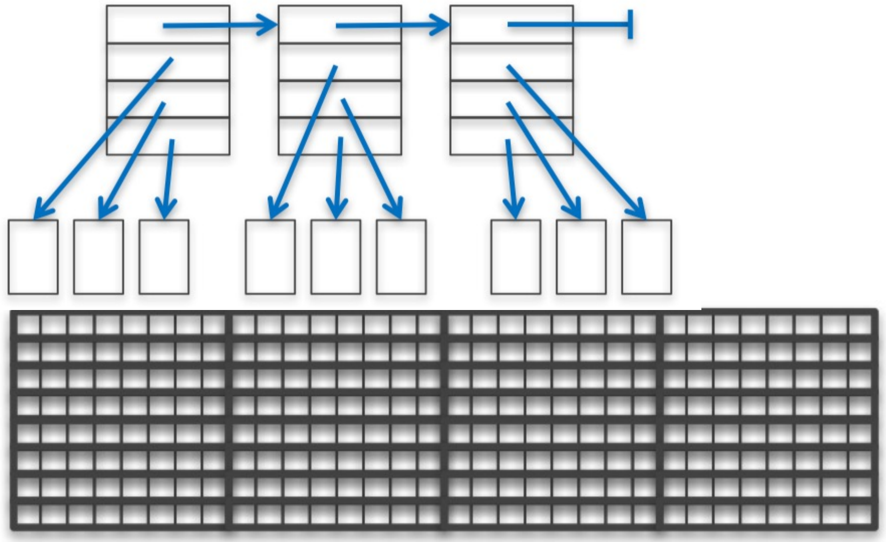
Free List

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
  - actually used

Problem: Poor Performance

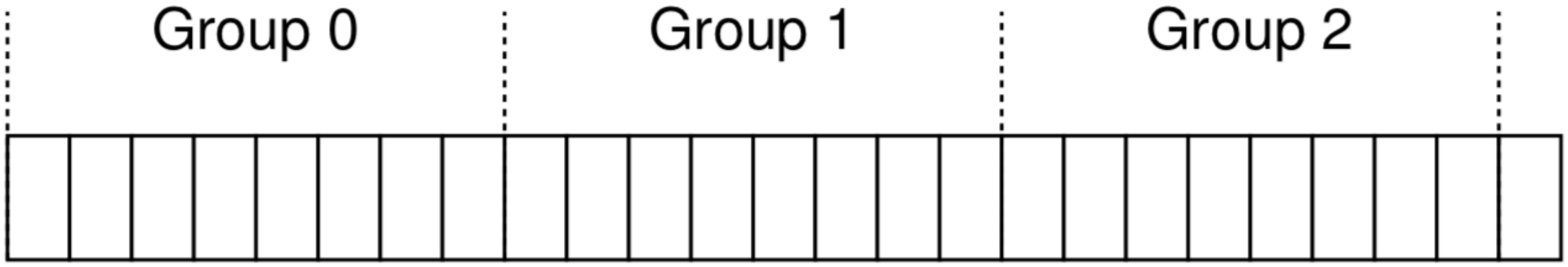
- In a naïve implementation of FFS, performance starts bad and gets worse
- One early implementation delivered only 2% disk bandwidth
- The root of the problem: poor locality
  - data blocks of a file were often far from its inode
  - file system would end up highly fragmented: accessing a logically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

Implementation Basics

- Directories: file name -> low-level names (i.e., file numbers)
- Index structures: file number -> block
- Free space maps: find a free block (ideally nearby)
- Performance optimizations (e.g., locality heuristics)

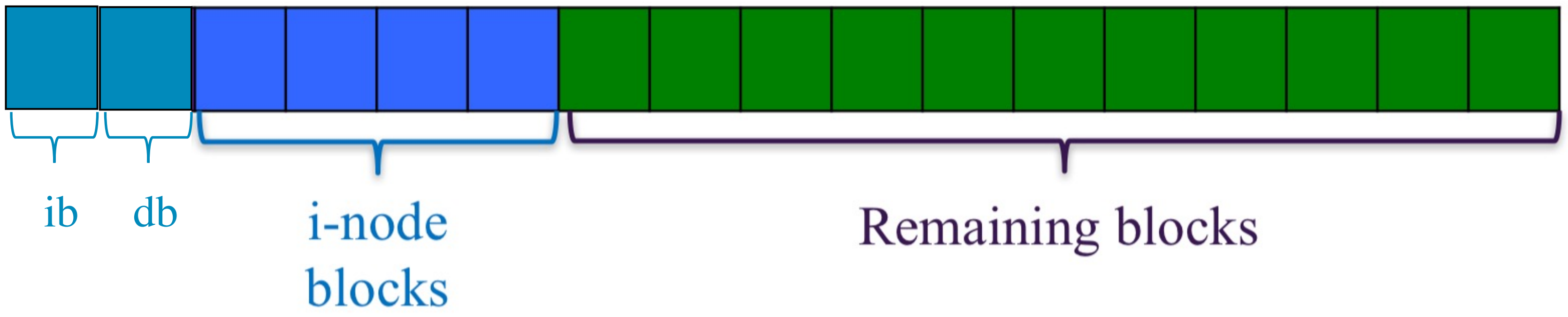
Solution 1: Disk Awareness

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

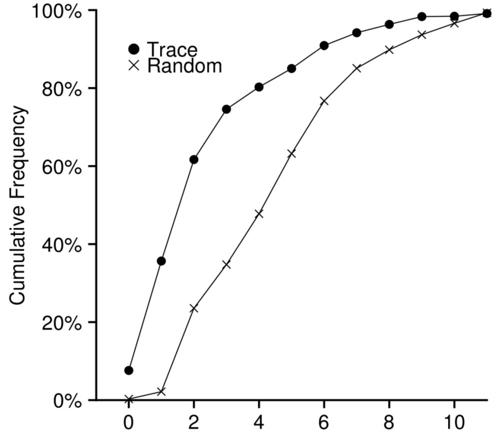


Allocating Blocks

- 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
  - OR group directories
- Allocating files:
  - place all file data in same group
  - uses first-fit heuristic
  - reserves ~10% space to avoid deterioration of first-fit
- Defragmentation



Other Solutions

- **Page Cache:** to reduce costs of accessing files, cache file contents in memory (e.g., device data, memory-mapped files)

- **Copy-on-write (COW):** create new, updated copy at time of update

- **Write Buffering:** buffer writes and periodically flush to disk