

Lecture 22: File Systems

CS 105

Fall 2024

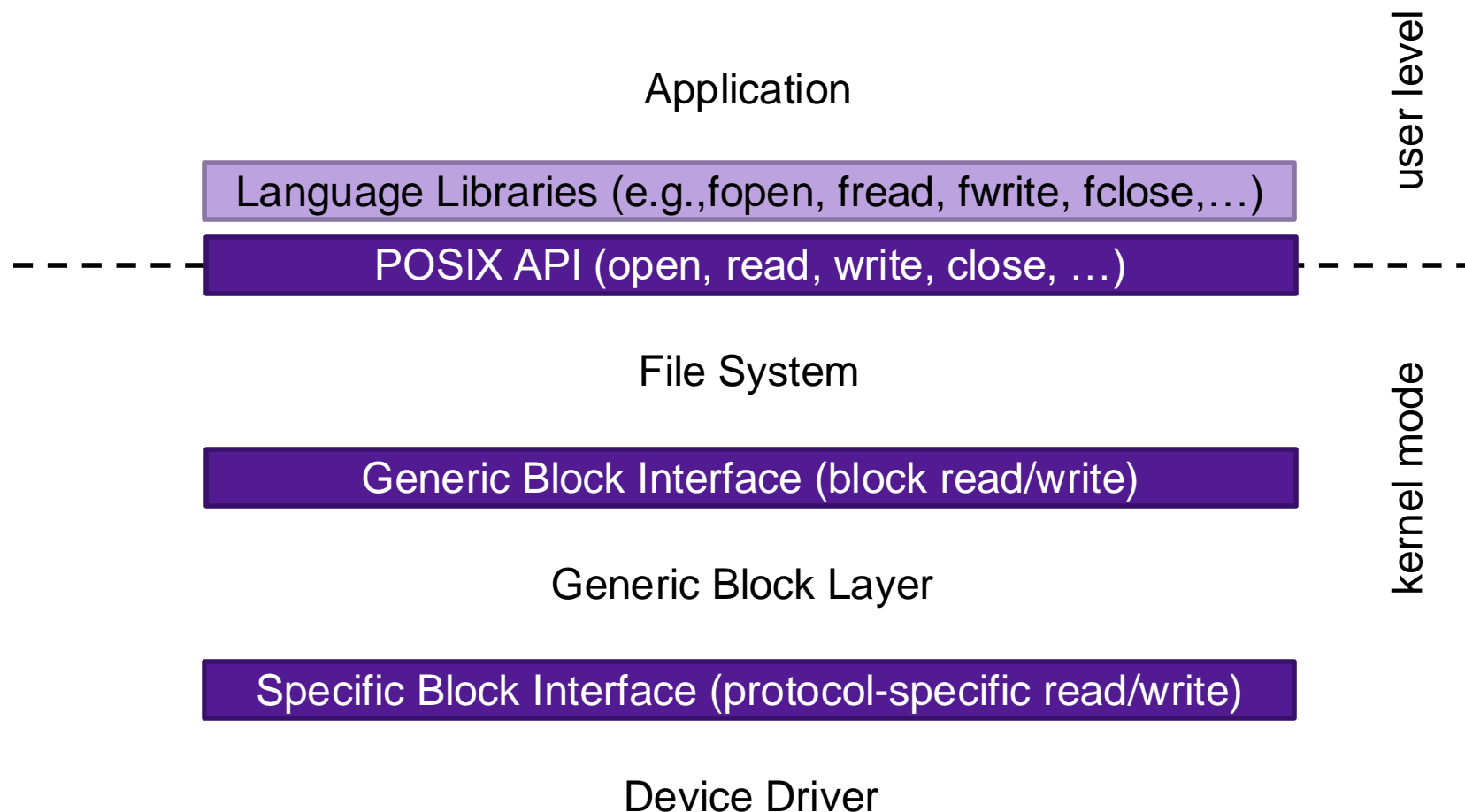
Review: File Systems 101

- 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
 - directories (a special kind of file)

Review: 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 is comprised of 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)

Review: The File System Stack

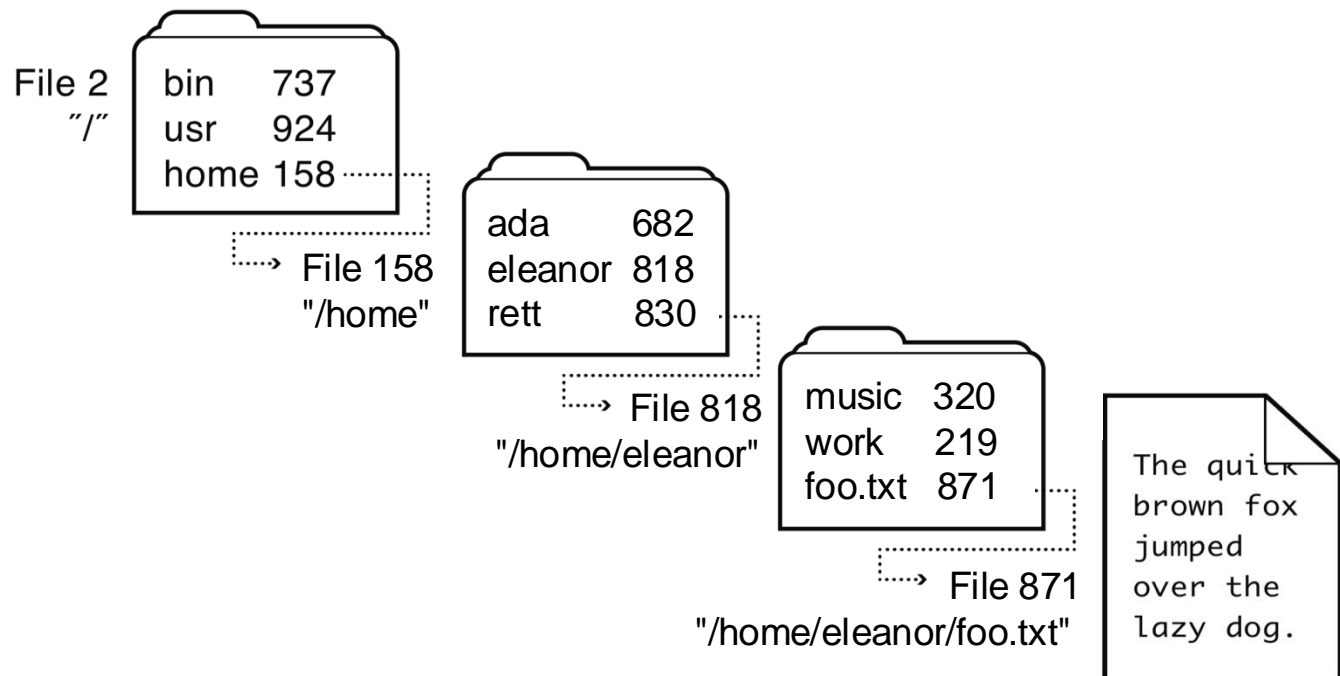


Implementation Basics

- Directories: file name -> low-level names (i.e., file numbers)

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 or directory
- OS uses path name to find directories and files



Exercise 1: Linked Allocation

- How many disk reads would be required to read (all of) a file named `/foo/bar/baz.txt`?
 - assume all files can be read with one disk read

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)

Implementation Basics

- Directories: file name -> low-level names (i.e., file numbers)
- File system index structures: file number -> block(s)

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

File System Properties

- Most files are small
 - need strong support for small files (optimize the common case)
 - block size can't be too big
- 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

Storing Files

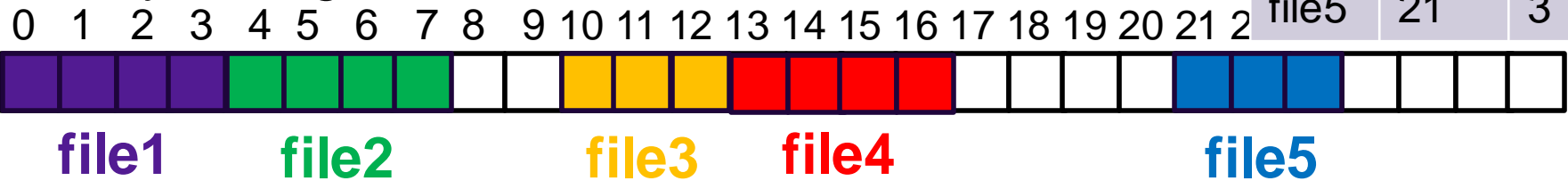
Possible ways to allocate files:

- **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

Continuous Allocation

	start	size
file1	0	4
file2	4	4
file3	10	3
file4	13	4
file5	21	3

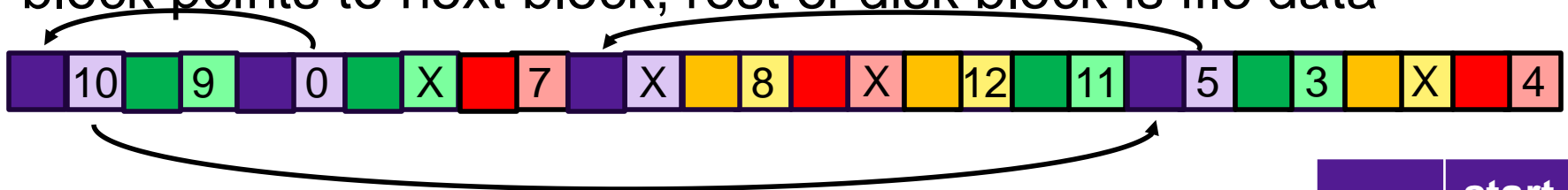
All bytes together, in order



- + **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

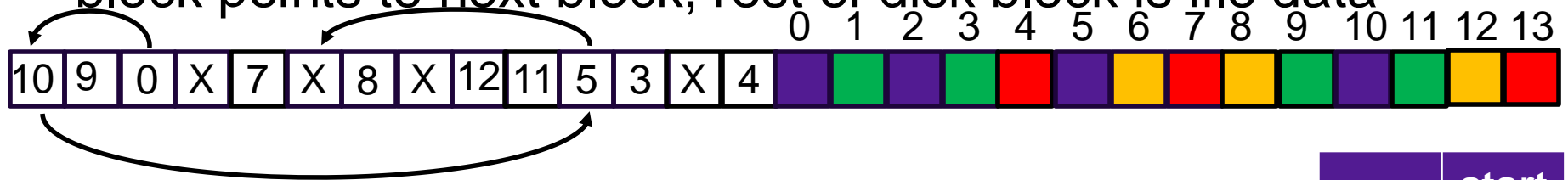
Each file is stored as linked list of blocks: One word of each block points to next block, rest of disk block is file data



	start
file1	2
file2	9
file3	6
file4	13
file5	15

Decoupled 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



	start
file1	2
file2	9
file3	6
file4	13
file5	15

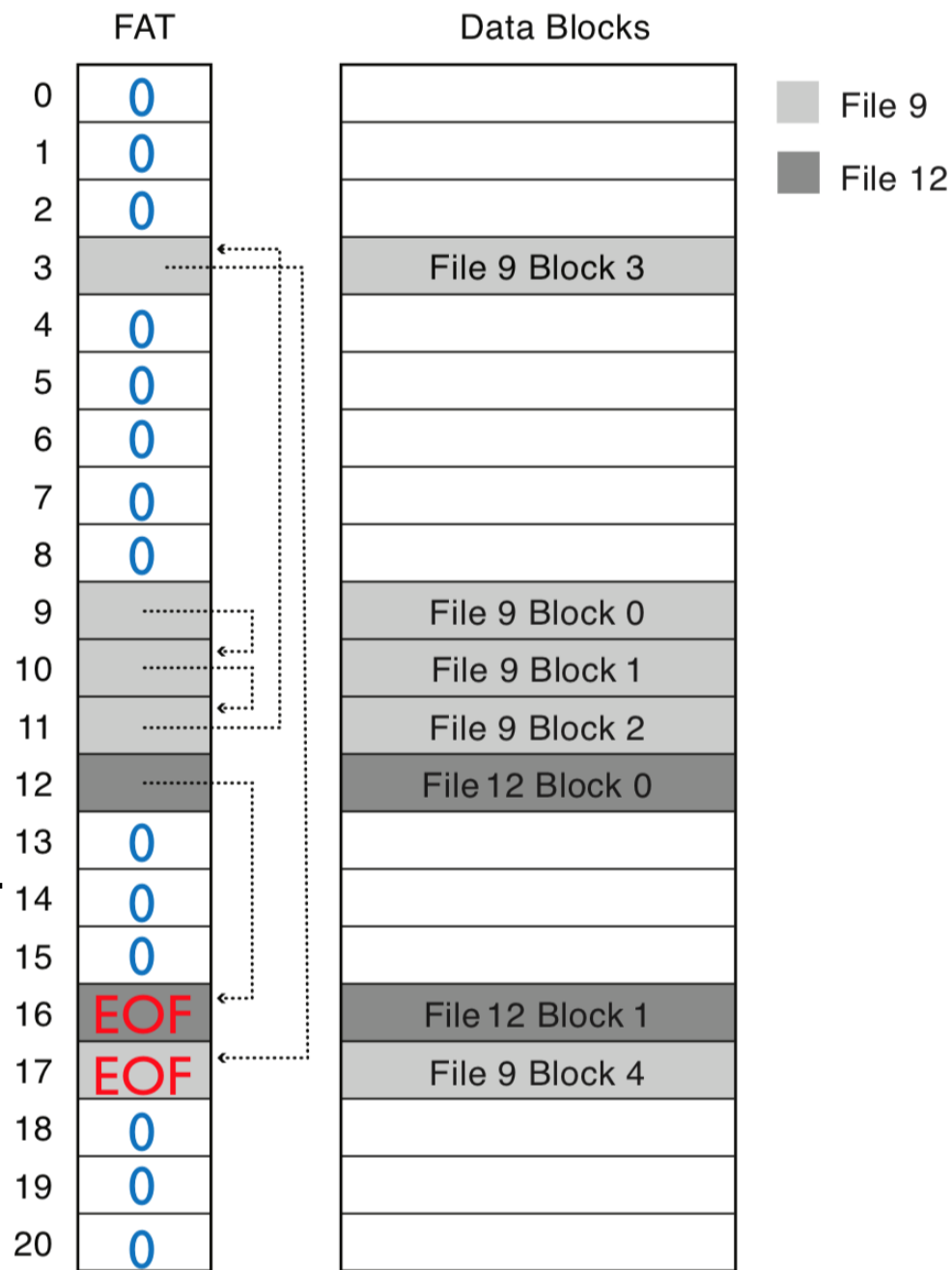
Exercise 2: (Decoupled) Linked Allocation

- 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

FAT File System

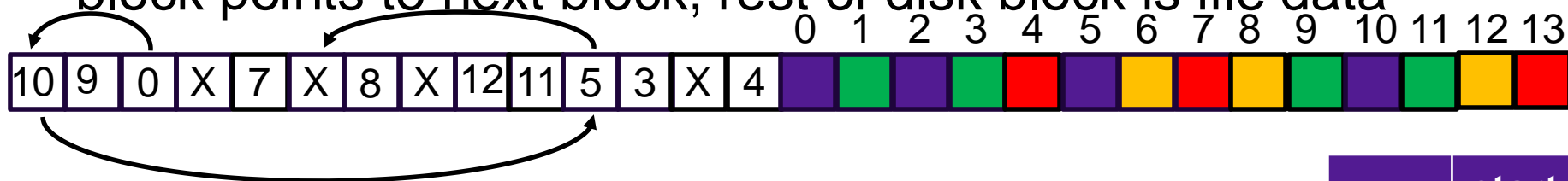
- Developed by Microsoft for MS-DOS
- decoupled linked allocation
- 1 FAT entry per block ("next pointer")
 - EOF for last block
 - 0 indicates free block
- low-level file name = FAT index of first block in file

Directory	
cecil.txt	9
eleanor.txt	12



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

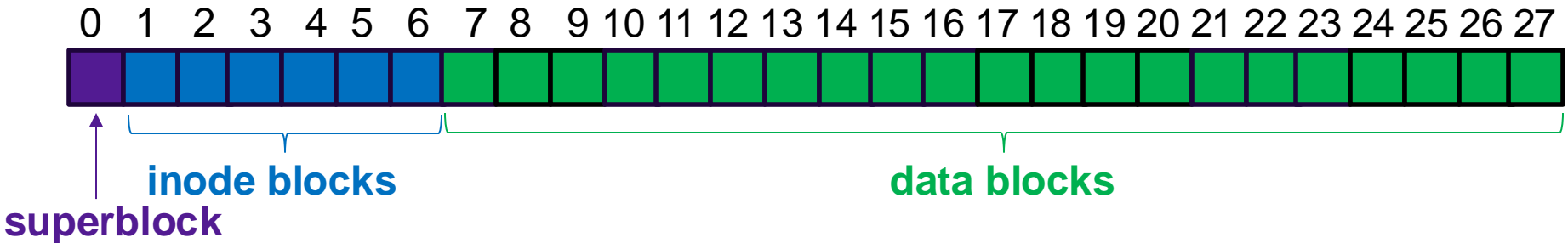


- + **Simple:** directory only need to store 1st block of each file
- + **Space Utilization:** no space lost to external fragmentation
- **Performance:** random access is slow
- ~ **Space Utilization:** overhead of pointers

	start
file1	2
file2	9
file3	6
file4	13
file5	15

Indexed Allocation: Fast File System (FFS)

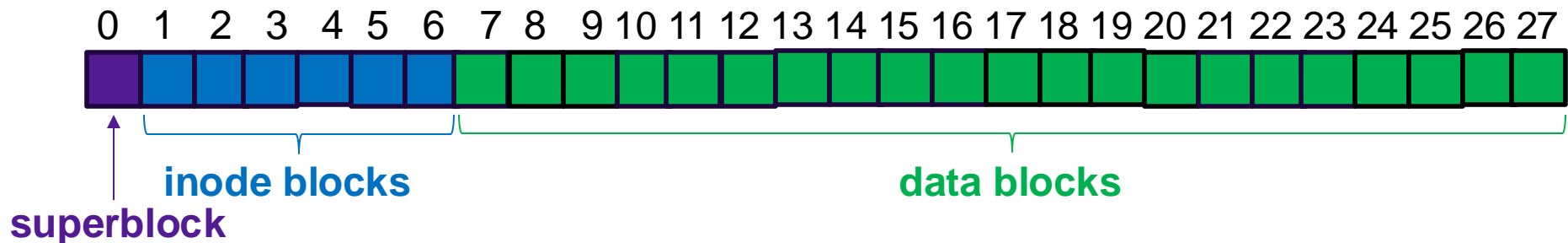
- tree-based, multi-level index



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

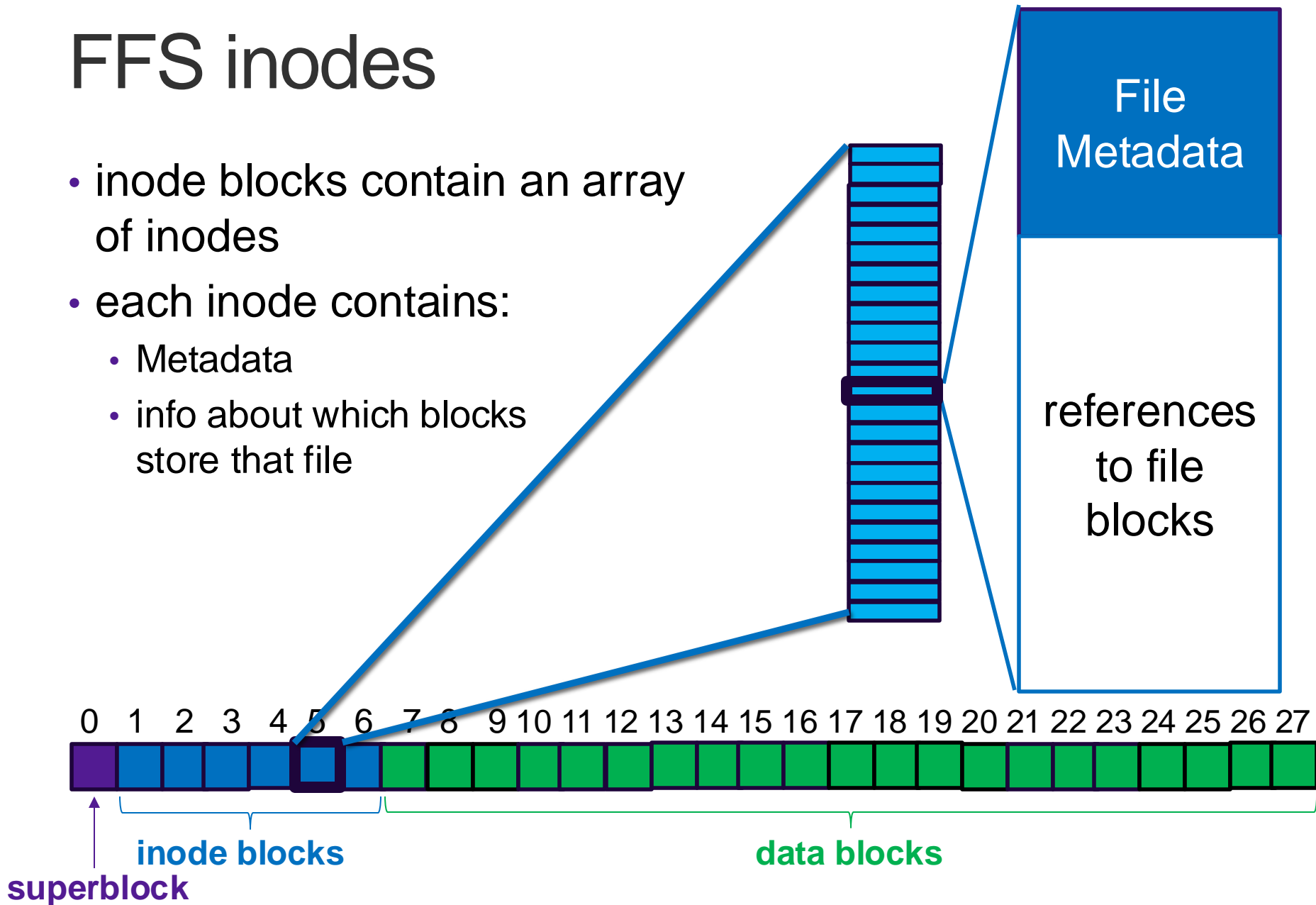
FFS Superblock

- 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
 - info about which blocks store that file



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

File
Metadata

references
to file
blocks

Each "Pointer" is a block number, not a memory address

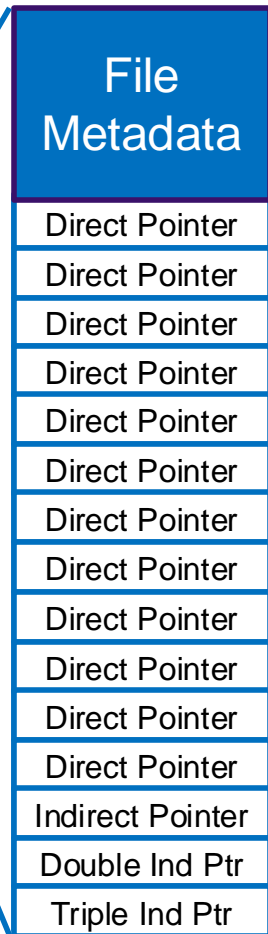
Indirect blocks contain arrays of block numbers

FFS Index Structures

Inode Array



Inode

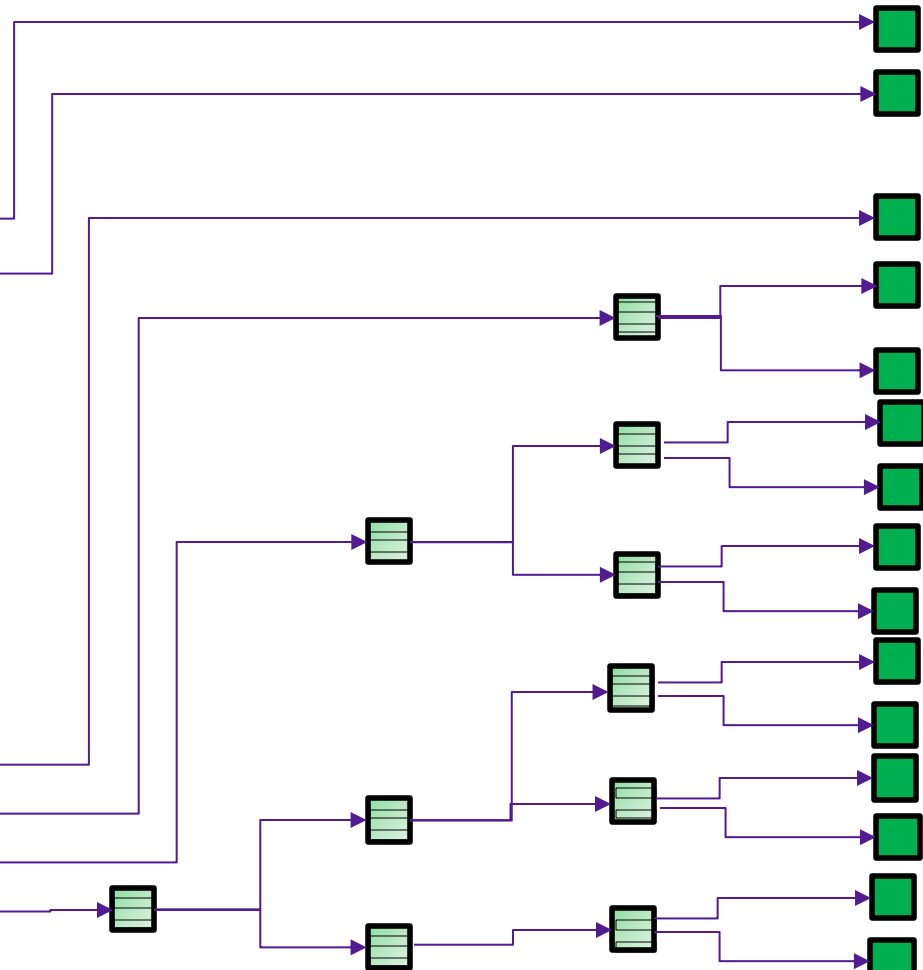


Triple Indirect Blocks

Double Indirect Blocks

Indirect Blocks

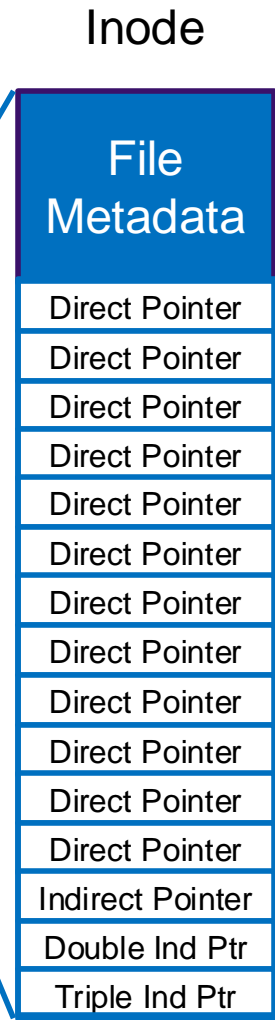
Data Blocks



Assume: blocks are 4KB (2^{12} bytes)
block numbers are 4 byte values

Max File Size

Inode Array



Triple Indirect Blocks

Double Indirect Blocks

Indirect Blocks

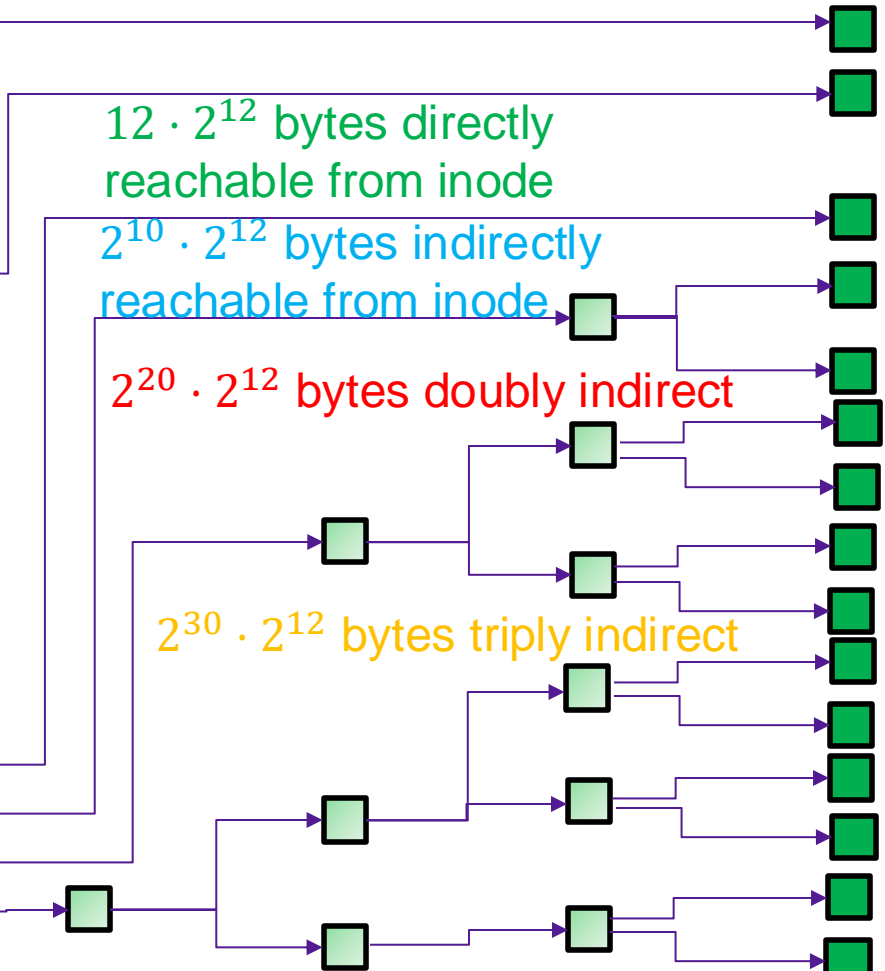
Data Blocks

$12 \cdot 2^{12}$ bytes directly reachable from inode

$2^{10} \cdot 2^{12}$ bytes indirectly reachable from inode

$2^{20} \cdot 2^{12}$ bytes doubly indirect

$2^{30} \cdot 2^{12}$ bytes triply indirect



Exercise 3: 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.

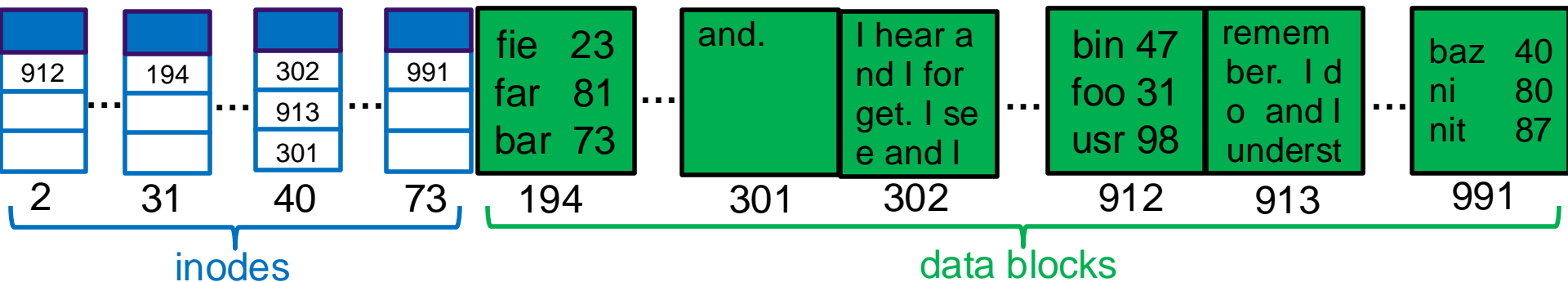
- Which pointers in the inode of a 32KB file would be non-null?
- Which pointers in the inode of a 47MB file would be non-null?

FFS Directory Structure

- Originally: directory was 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 4: Indexed Allocation

How many disk reads would be required to read (all of) file /foo/bar/baz?



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

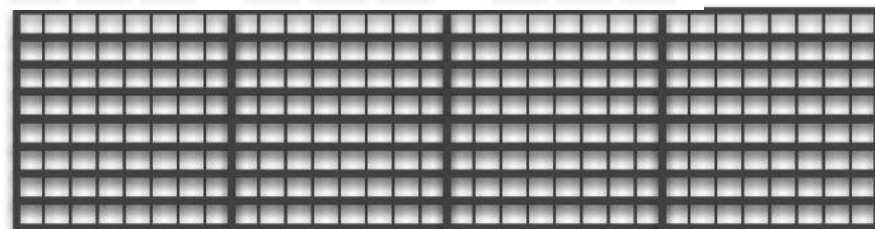
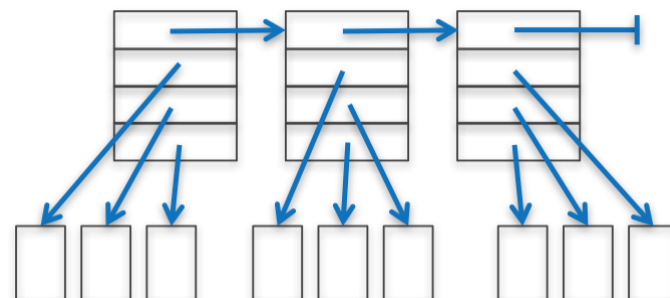
- Directories: file name -> low-level names (i.e., file numbers)
- Index structures: file number -> block
- Free space maps: find a free block (ideally nearby)

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)

Performance Optimizations

- **Grouped Allocation:** disk organized into groups that are (temporally) close, try to allocate all file blocks in same group
- **Defragmentation:** periodically rearrange files to improve locality
- **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