ILLINOIS TECH

College of Computing

CS 450 Operating Systems
File System Implementation

Yue Duan

Recap: File Storage Layout Strategies

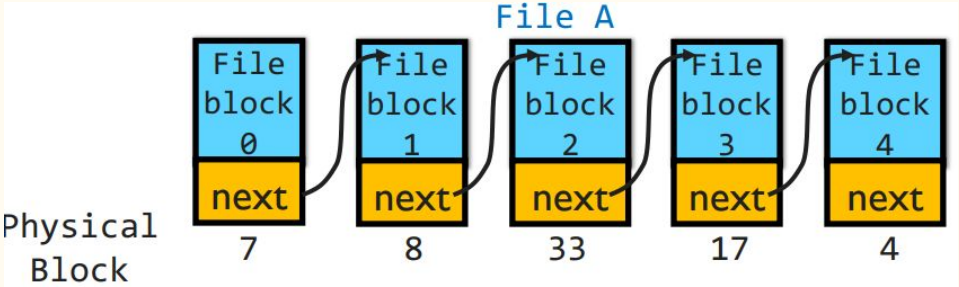
- Files span multiple disk blocks
  - contiguous allocation
    - all bytes together, in order
  - Linked structure
    - each block points to the next, directory points to the first
  - Indexed structure
    - an **index block** contains pointers to many other blocks
    - may need multiple index blocks (linked together)

Recap: Contiguous Allocation

- All bytes of file are stored together, in order
  - + simple: state required per file: start block & size
  - + efficient: entire file can be read with one seek
  - – fragmentation: external fragmentation is bigger problem
  - – usability: user needs to know size of file at time of creation

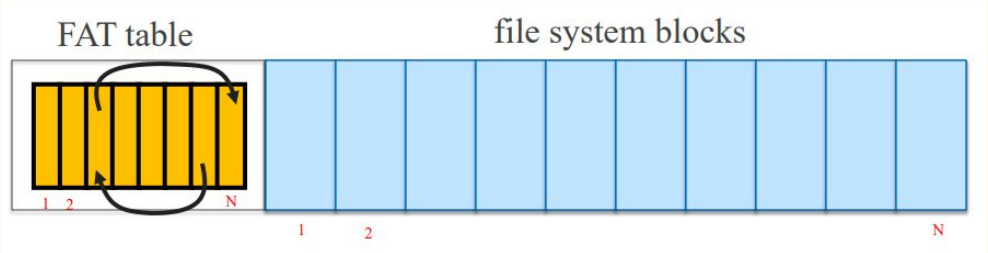
Recap: Linked-List File Storage

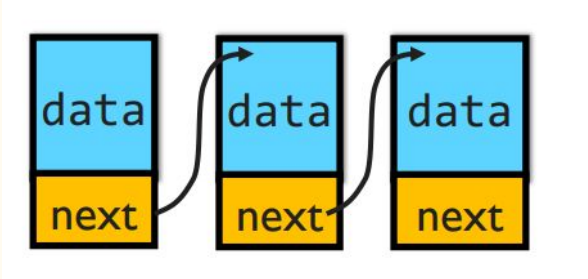
- Each file is stored as linked list of blocks
  - first word of each block points to next block
  - rest of disk block is file data
  - + space utilization: no space lost to external fragmentation
  - + simple: only need to store 1st block of each file
  - – performance: random access is slow
  - – space utilization: overhead of pointers



Recap: FAT File System

- File Allocation Table (FAT)
  - Used in MS-DOS, precursor of Windows
  - Still used (e.g., CD-ROMs, thumb drives, camera cards)
  - FAT-32, supports 2^{28} blocks and files of 2^{32}-1 bytes
  - The FAT Table
    - a linear map of all blocks on disk
    - each file is a linked list of blocks
      - with metadata located in the first block of the file

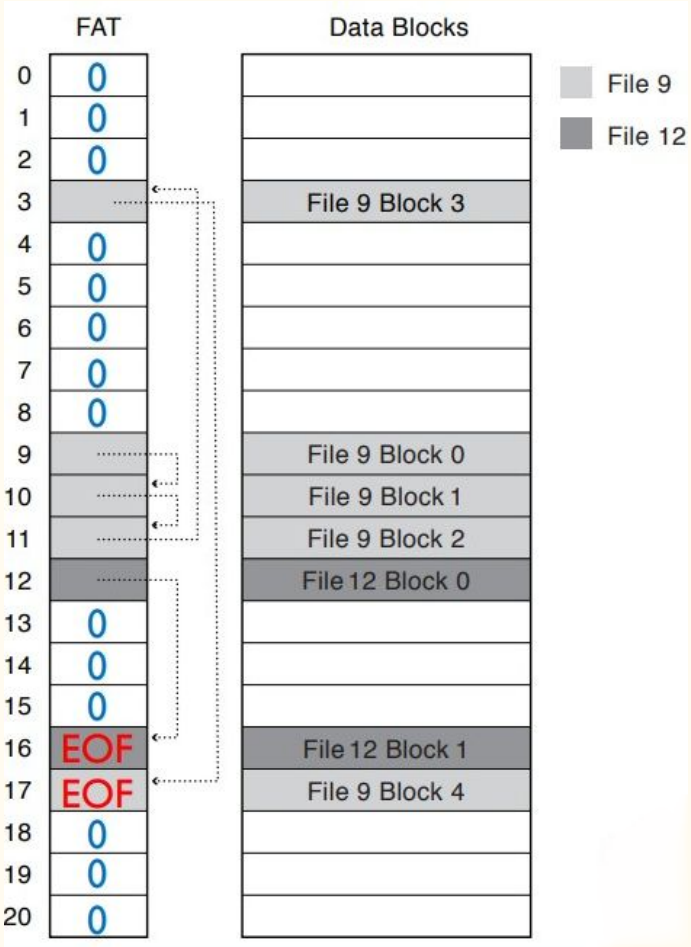




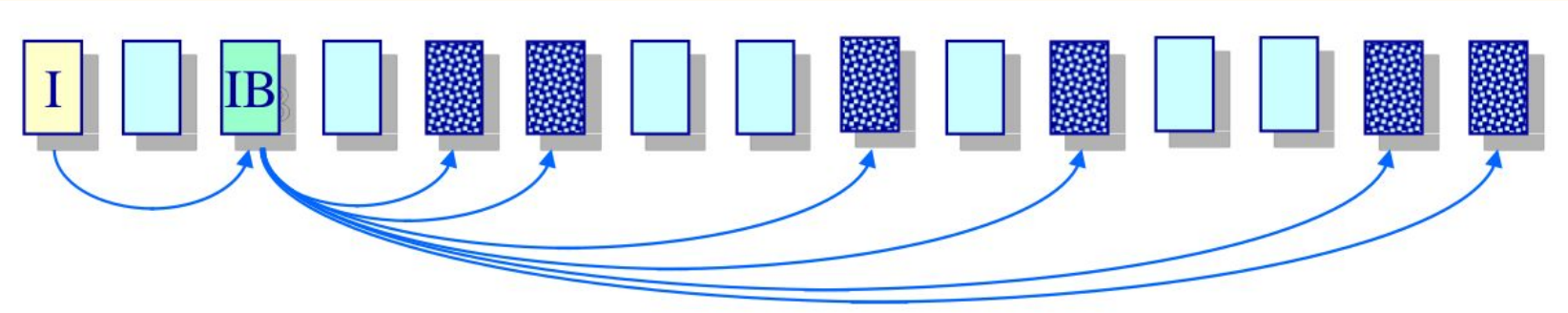
Recap: FAT File System

- 1 entry per block
- EOF for last block
- 0 indicates free block
- directory entry maps name to FAT index

Directory	
bart.txt	9
maggie.txt	12

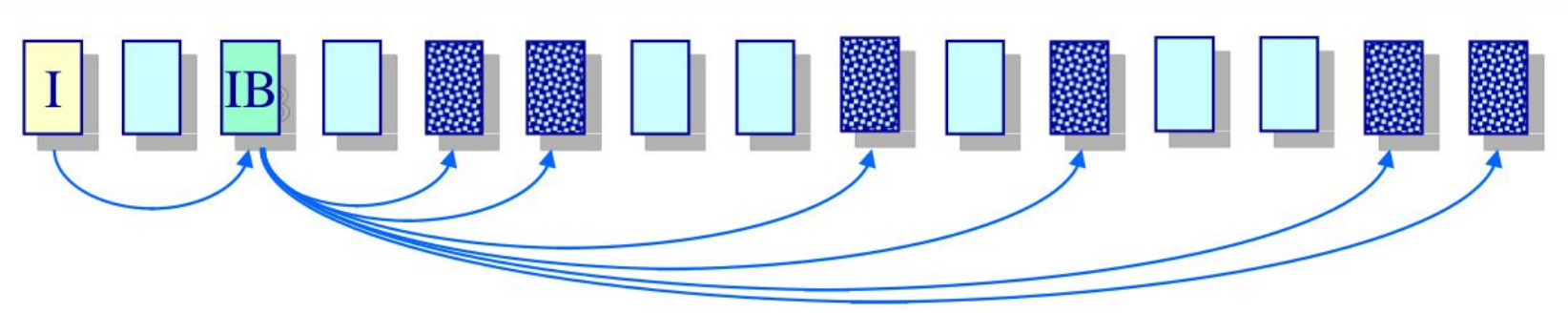


Indexed Allocation



- **Inode** points to **Index Block**
- **Index block** is an array of pointers to all blocks in the file
  - metadata: array of block numbers
  - allocate space for pointer at file creation time

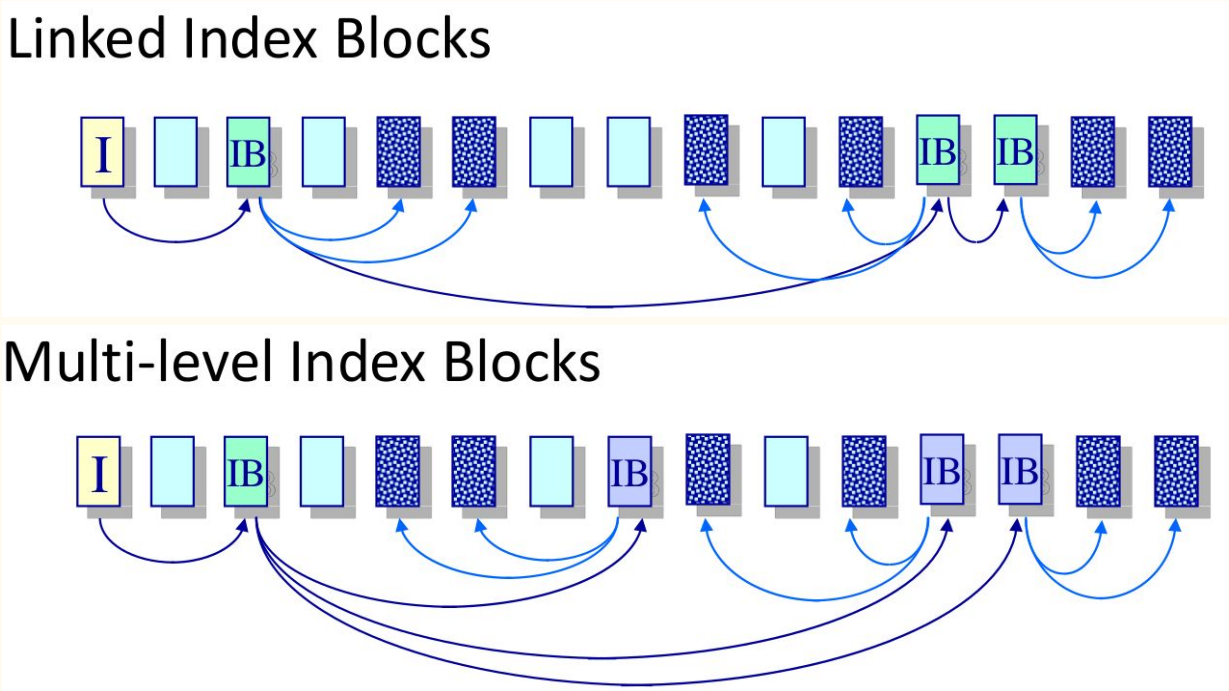
Indexed Allocation



- Fragmentation? + no external fragmentation
- Sequential access? +/- depends on block placement
- Random access? + easy to find block number
- File growth? +/- easy up to max size; but max is small
- Metadata overhead? - high, especially for small files

Indexed Allocation

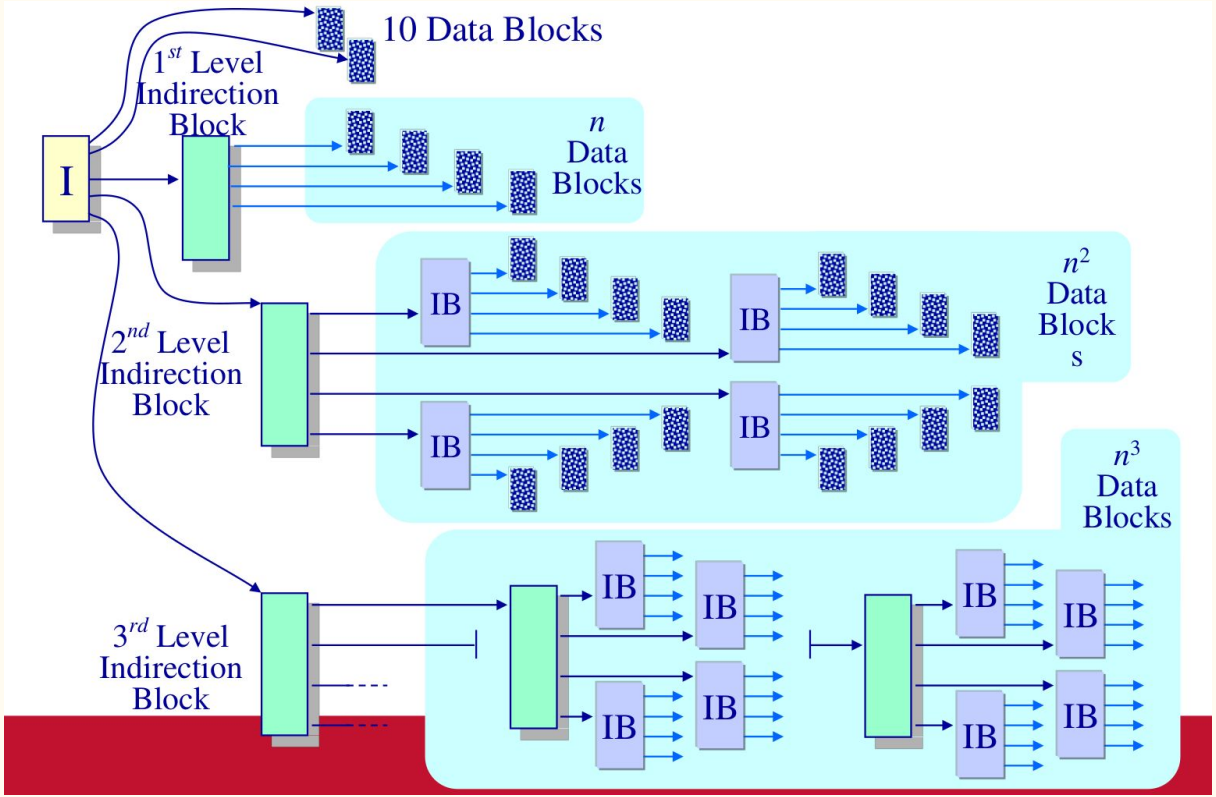
- How to support large files?



Multi-Level Indexing

- E.g., Unix FFS and ext2/ext3 file systems
- Inode contains N+3 pointers
  - N direct pointers to first N blocks in the file
  - 1 indirect pointer (points to an index block)
  - 1 double-indirect pointer (points to an index block of
  - index blocks)
  - 1 triple-indirect pointer (points to ...)

Multi-Level Indexing



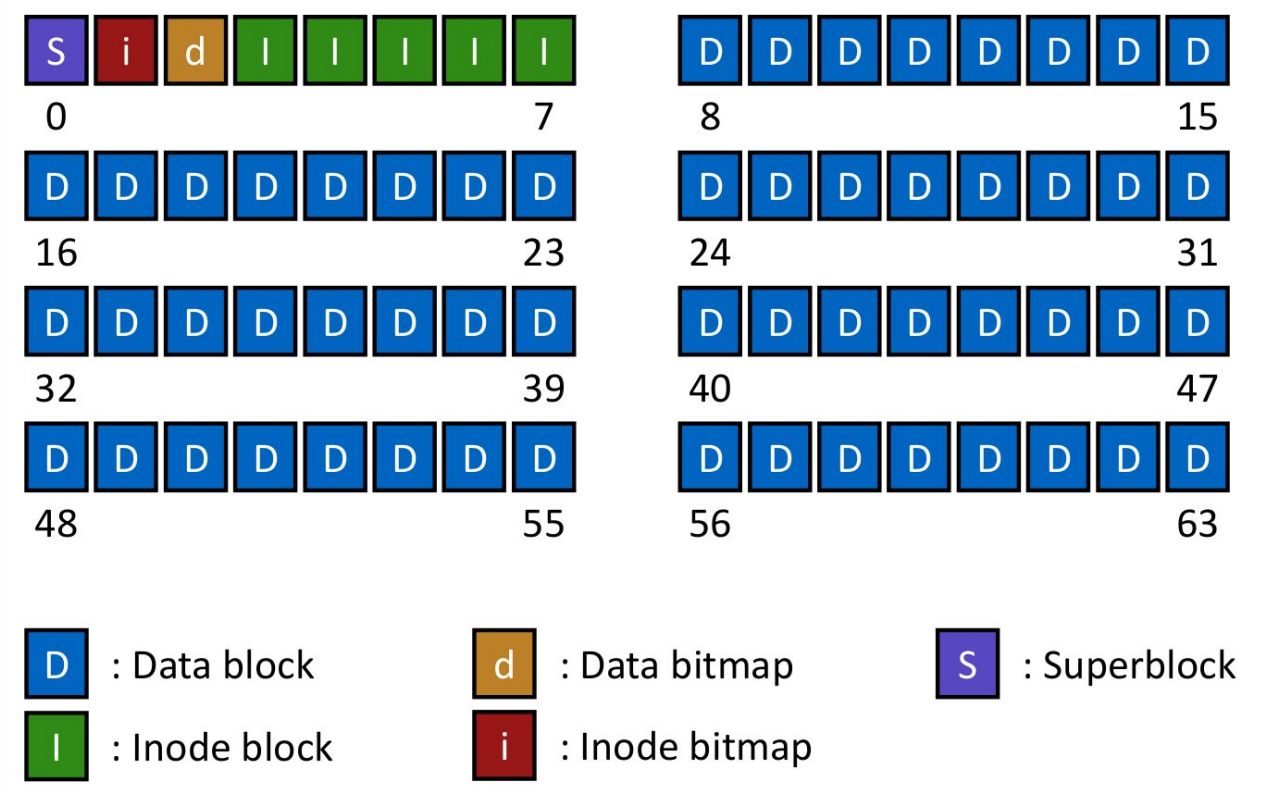
Multi-Level Indexing

- Why have N (10) direct pointers?
  - because most files are small
  - allocate indirect blocks only for large files
- Implications
  - +/- maximum file size limited (a few terabytes)
  - + no external fragmentation
  - + simple and supports small files well
  - + easy to grow files
  - +/- sequential access performance depends on block layout
  - +/- random access performance good for small files
    - for large files have to read multiple indirect blocks first

On-Disk FS Layout

- Consider a general scheme that forms basis of most FS
- Disk blocks are used to hold one of the following
  - **data blocks**
  - **inode table**
    - each block here stores a few inodes;
    - i-number determines which block in the table and which inode in the block
  - **indirect blocks**: often in the same pool as data blocks
  - **directories**: often in the same pool as data blocks
  - **data block bitmap**: to identify free/used data blocks
  - **inode bitmap**: to identify free/used inodes
  - **superblock**

Simple Layout



Inode Block

- Inodes are fixed size
  - 128-256 bytes
- Assume 4K blocks
  - i.e., each block is 8 sectors
- 16 inodes per inode block
  - easy to find block containing a given inode number

On-Disk inode Data

- Type
  - file, directory, symbolic link, etc.
- Ownership and permission info
- Size
- Creation and access time
- File data
  - direct and indirect block pointers
- Link count

Directories

- Common design:
  - a special file with its inode
  - store directory entries in data blocks
  - large directories just use multiple data blocks
- Various formats could be used to store dentries
  - lists
  - B-trees
  - different tradeoffs w.r.t. cost of searching, enumerating children, free entry management, etc

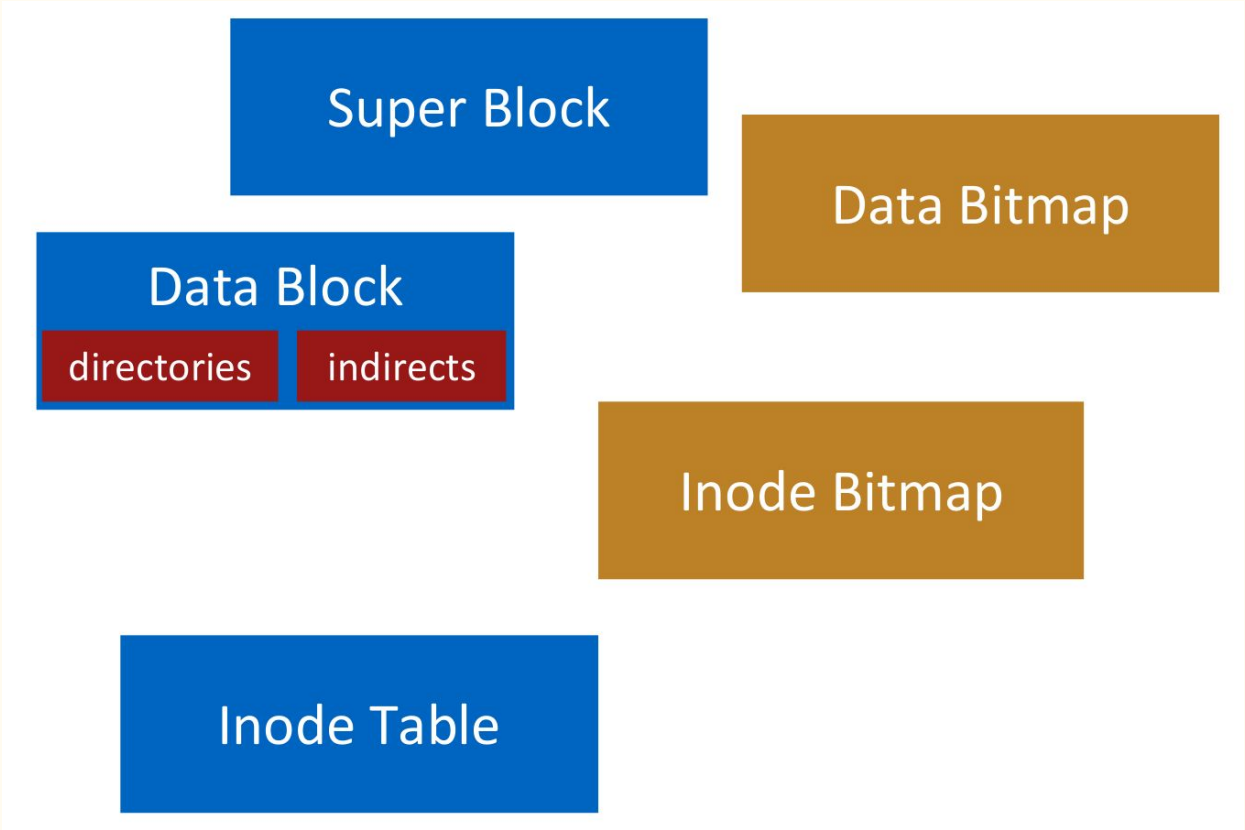
Free Space Management

- How do we find free data blocks or free inodes?
- Two common approaches
  - in-situ free lists
  - bitmaps (more common)

Superblock

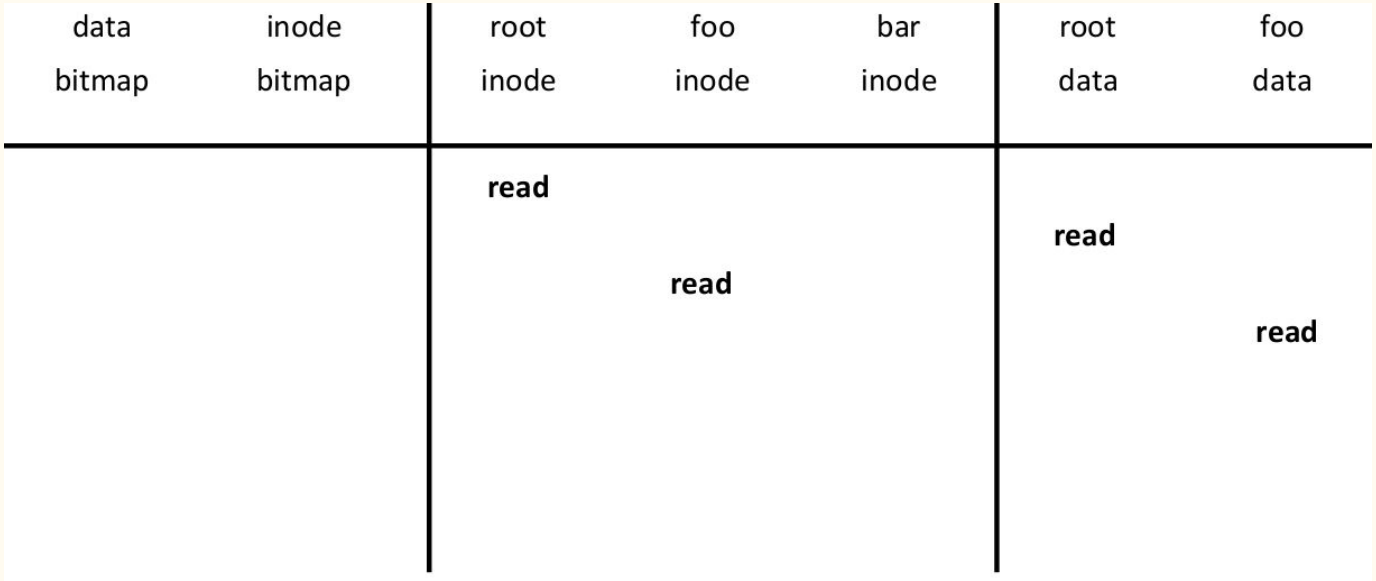
- Need to know basic FS configuration metadata, like:
  - FS type (FAT, FFS, ext2/3/4, etc.)
  - block size
  - # of inodes
  - location of inode table and bitmaps

Summary: On-Disk Structures



Example: create /foo/bar

- Step 1: traverse
  - verify that bar does not already exist



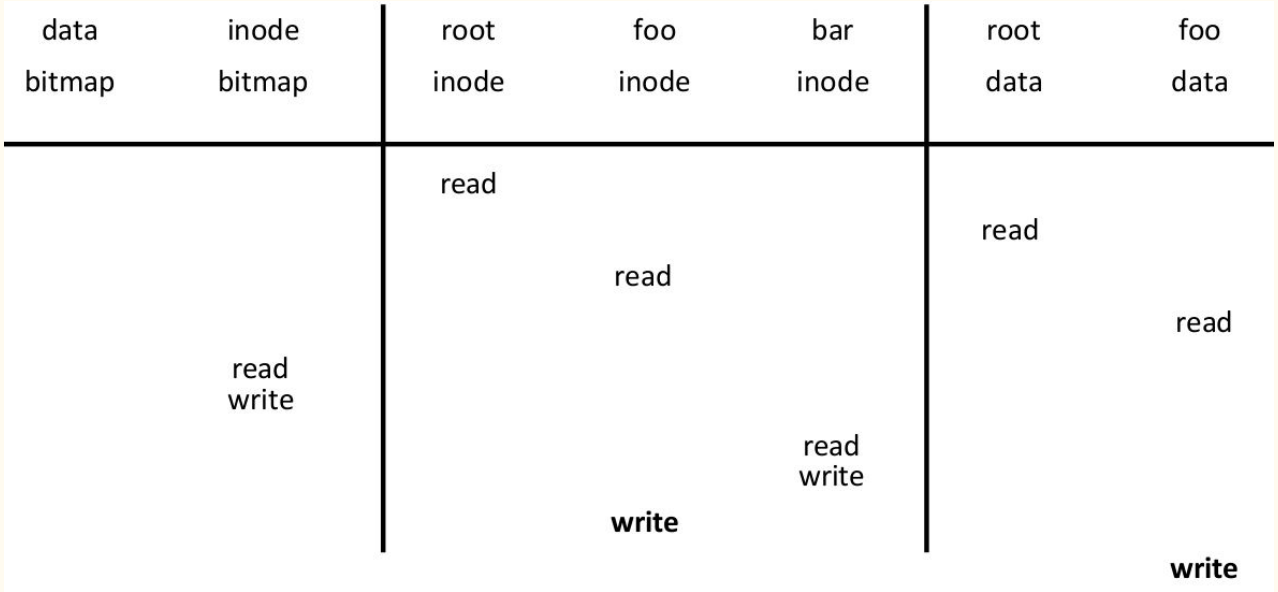
Example: create /foo/bar

- Step 2: populate inode
  - **Why must read bar inode block?**
  - **How to initialize inode?**

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		read				
					read	
			read			
						read
	read write					
				read write		

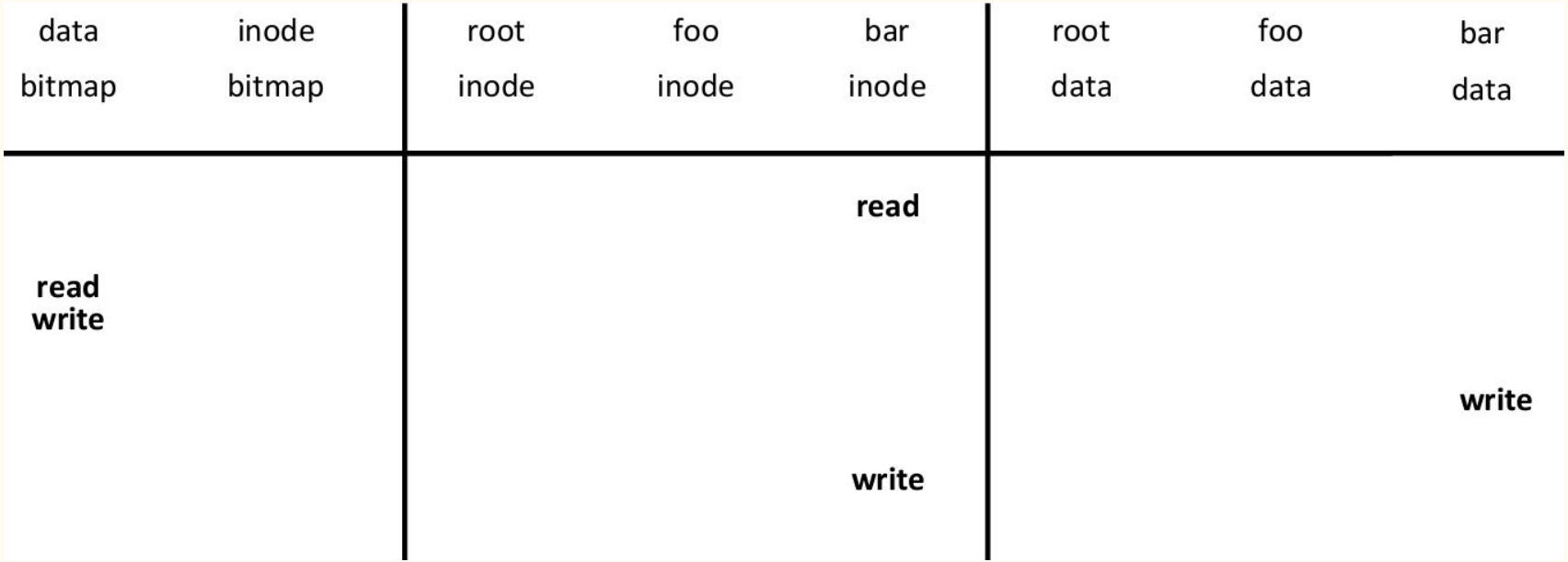
Example: create /foo/bar

- Step 3: update directory
  - Update directory's inode (e.g., size) and data



Example: write to /foo/bar

- Assuming it's already opened
  - Need to allocate a data block assuming bar was empty



Efficiency

- How to avoid so much IO for basic operations?
  - cache disk data aggressively
- What to cache?
  - everything
    - Inodes
    - Dentries
    - Allocation bitmaps
    - Data blocks
- Reads first check the cache; if not there, then access disk
- Modifications update the cached data (make them **dirty**)
  - Dirty data is written back to disk later in the background

Issues with Caching

- Many important decisions to make
  - how much to cache?
  - how long to keep dirty data?
  - how much to write back?
- What about crashes?
  - FS consistency issues

sync() System Calls

- In case an application needs cached data flushed to disk immediately
- **sync()**
  - flush all dirty buffers to disk
- **syncfs(fd)**
  - flush all dirty buffers to disk for FS containing fd
- **fsync(fd)**
  - flush all dirty buffers associated with this file to disk (including metadata changes)
- **fdatasync(fd)**
  - flush only dirty data pages for this file
  - don’t bother with inode metadata, unless critical metadata changed

THANK YOU!