

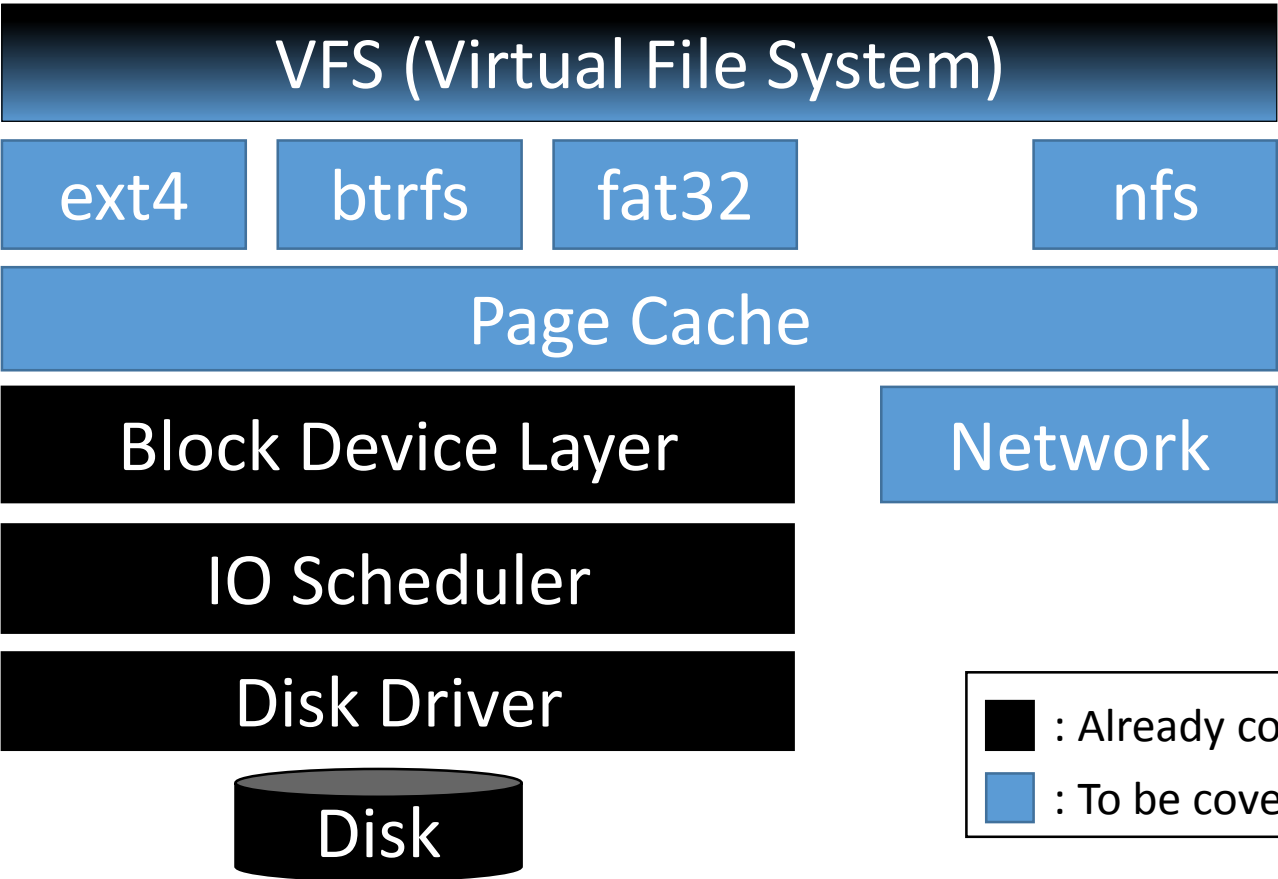
# Basic FS Implementation

Nima Honarmand

# A Typical Storage Stack (Linux)

User

Kernel



■ : Already covered  
■ : To be covered

# A Typical Storage Stack (Linux)

- **Block layer** and those underneath it hide disk details from the rest of storage stack
- **ext4, btrfs, fat32, nfs** are examples of “actual file systems”
  - The layer that determines how disk blocks are used to store the file system data and metadata
  - nfs (Network File system) is different; it does not use disk
- **VFS** hides the FS-specific details and works in terms of generic inodes, dentries and superblocks
  - It calls FS-provided functions to access on-disk inode, dentry, superblock and file data
  - It also caches inodes and dentries to reduce disk accesses
- **Page cache** is the main layer that caches FS data in the memory
  - It interacts with most other layers

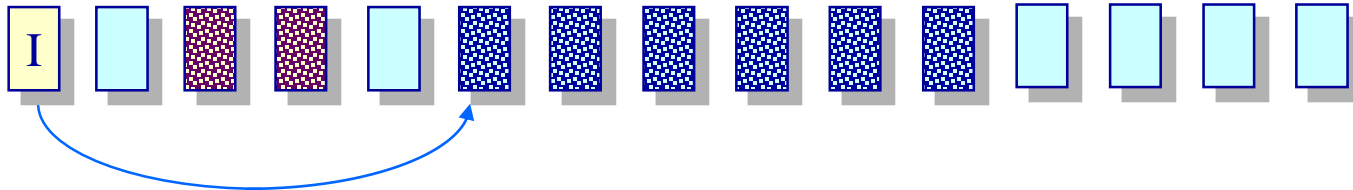
# File Allocation Methods

- Given a file's inode, how to find its data blocks?
  - inode somehow stores data block locations
- Many different approaches
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation
  - Multi-level indexed allocation
  - Extents
  - etc.

# File Allocation Considerations

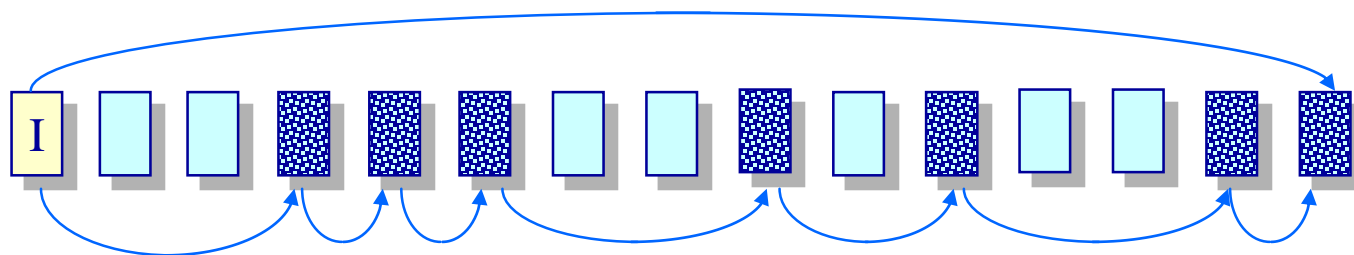
- Amount of fragmentation (internal and external)
  - Free space that can't be used
- Ability to grow file over time
- Performance of sequential accesses
- Performance of random accesses
  - Speed to find data blocks for random accesses
- Wasted space for meta-data overhead
  - Meta-data must be stored persistently too

# Contiguous Allocation



- Allocate each file to contiguous sectors on disk
- Inode specifies starting block & length
- Placement/Allocation policies
  - First-fit, best-fit, ...
- Fragmentation?
  - Awful external fragmentation
- Sequential access?
  - + Very good
- Random access?
  - + Easy to find block
- File growth?
  - Not easy; might need to move file
- Metadata overhead?
  - + Very low

# Linked Allocation



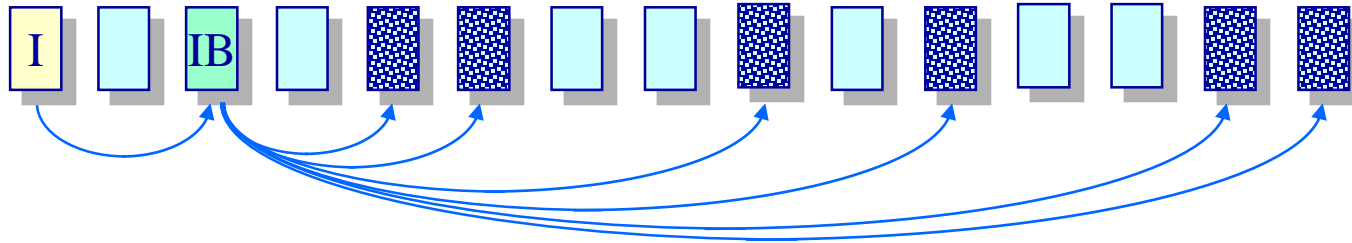
- File stored as a linked list of blocks
- Inode contains pointers to first and last data blocks
- Each block contains pointer to the next block
- Fragmentation? + No external fragmentation
- Sequential access? +/- Depends on block placement
- Random access? - Awful; has to traverse list to find
- File growth? + Easy and fast
- Metadata overhead? - One pointer per block

# Linked Allocation (cont'd)

- File Allocation Table (FAT)
  - A variant of linked allocation commonly used in older Windows, DOS and OS2
- Idea: Keep next-pointer information in a separate table
  - Table has one entry per disk block
  - The entry points to the next block in that file
- Advantage?
  - Table can be cached in memory (if small)
    - Can traverse linked list in memory
    - Improves random access performance



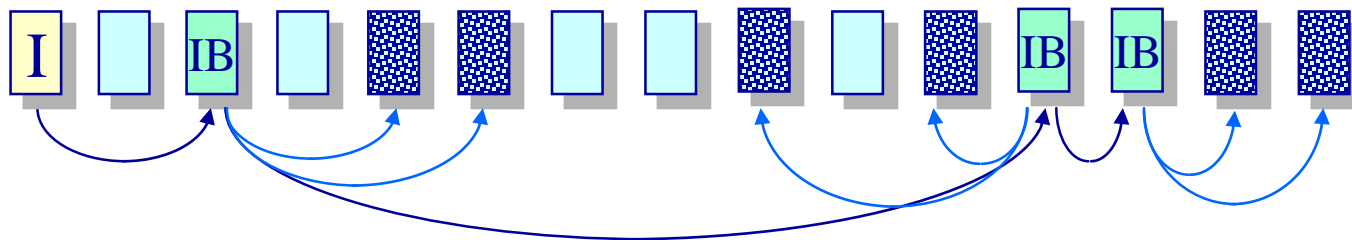
# Indexed Allocation



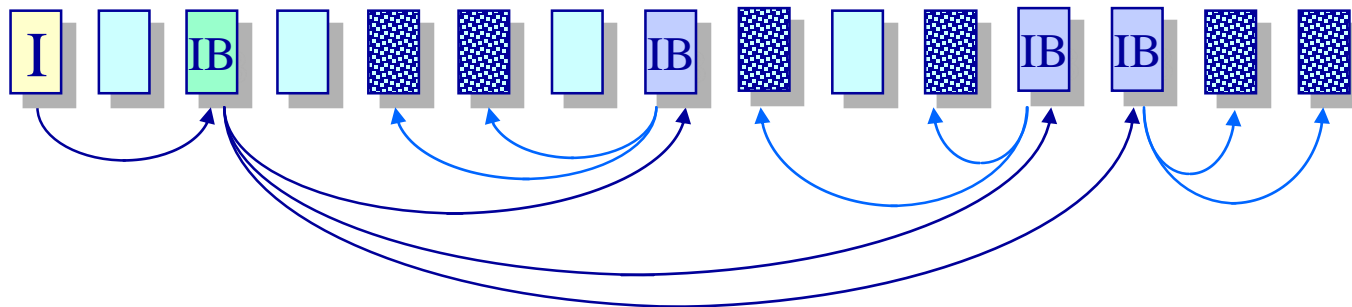
- Inode points to Index Block
- Index block is an array of pointers to all blocks in the file
  - Metadata: array of block numbers
  - Allocate space for pointer at file creation time
- Fragmentation? + No external fragmentation
- Sequential access? +/- Depends on block placement
- Random access? + Easy to find block number
- File growth? +/- Easy up to max size; but max is small
- Metadata overhead? - high, especially for small files

# Indexed Allocation (cont'd)

- How to support large files?
- Linked Index Blocks



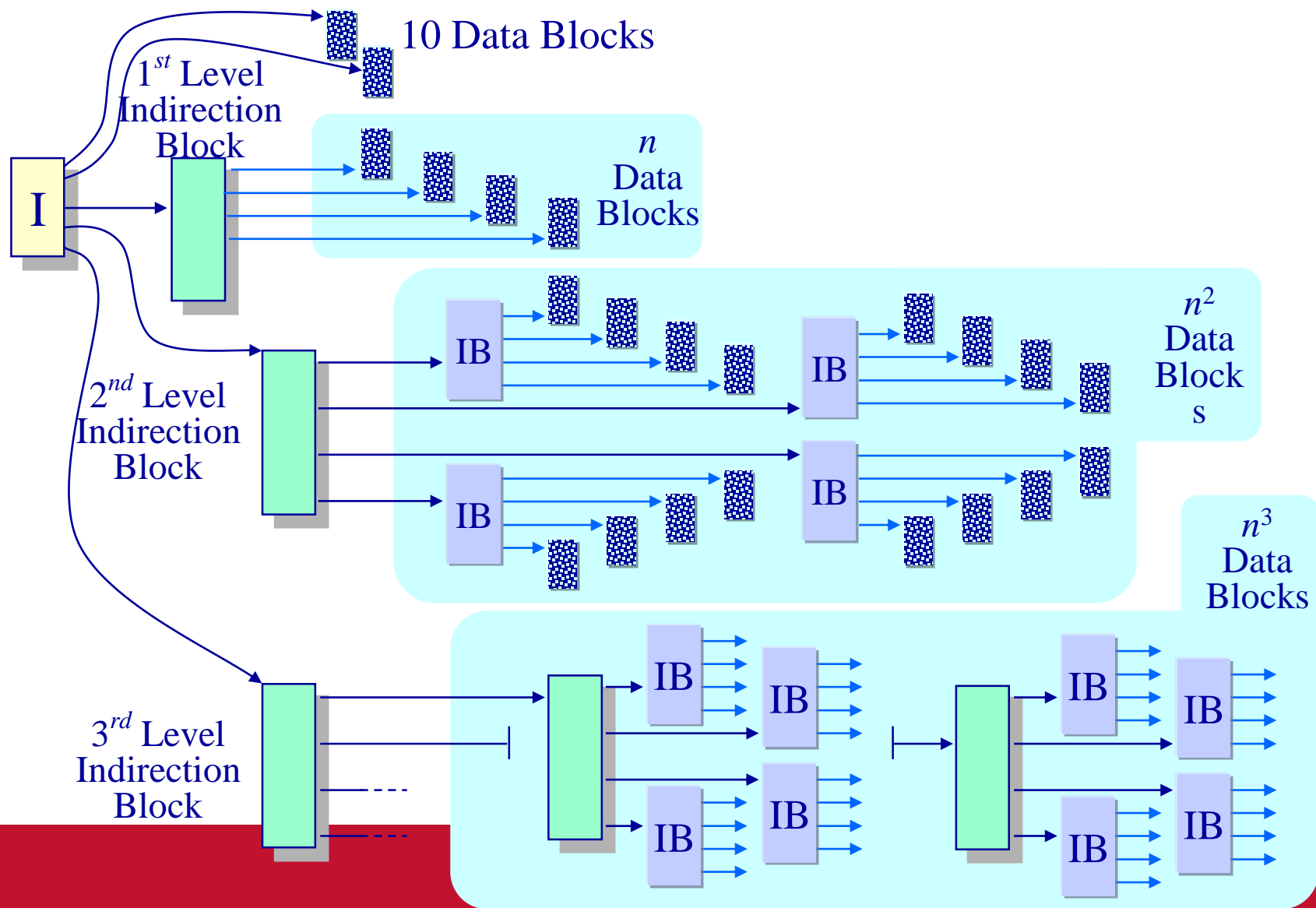
- Multi-level Index Blocks



# Multi-Level Indexing in Practice

- 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 in Practice



# Multi-Level Indexing in Practice

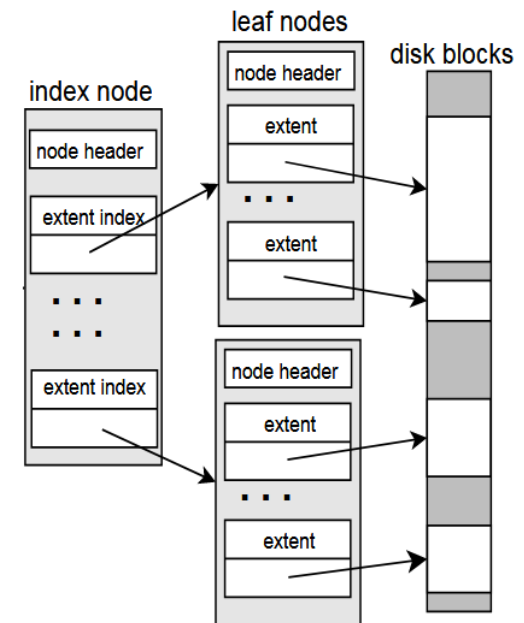
- Why have N (10) direct pointers?
  - Because most files are small  
→ allocate indirect blocks only for large files
- Implications
  - +/- Maximum file size limited (a few terabytes)
  - + No external fragmentation
  - + Simple and supports small files well
  - + Easy to grow files
  - +/- Sequential access performance depends on block layout
  - +/- Random access performance good for small files;  
for large files have to read multiple indirect blocks first

# Extent-Based Allocation

- Sequential access performance dictated by on-disk contiguity of file data blocks
  - Most file systems try to keep file data in big chunks of consecutive disk blocks
  - Why not use this fact to reduce individual block pointers?
- **Extent**: a consecutive range of disk blocks
  - Identified by its first block and length
- Inode store file blocks as a set of extents (instead of pointers)
  - Organize extents into multi-level tree structure
  - Each leaf node: starting block and contiguous size
  - Minimizes meta-data overhead when have few extents
  - Allows growth beyond fixed number of extents

# Extent-Based Allocation

- Ext4 uses extents instead of direct/indirect pointers used by ext2/3



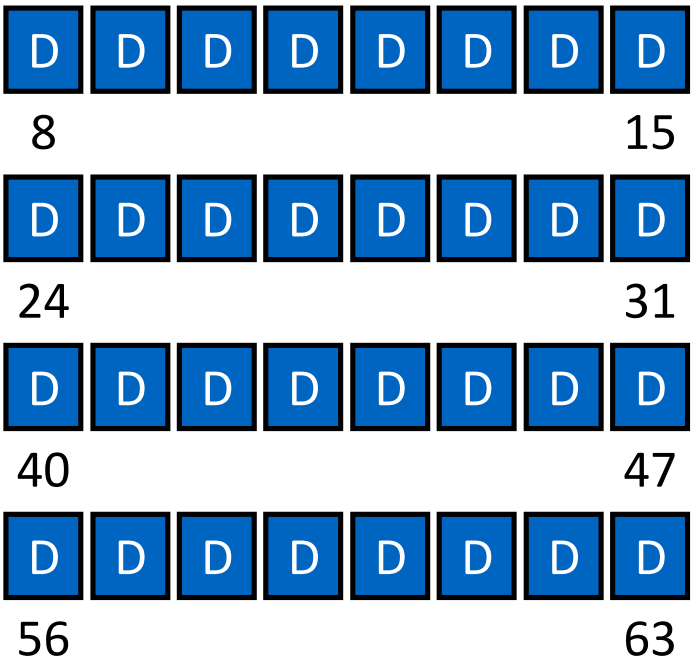
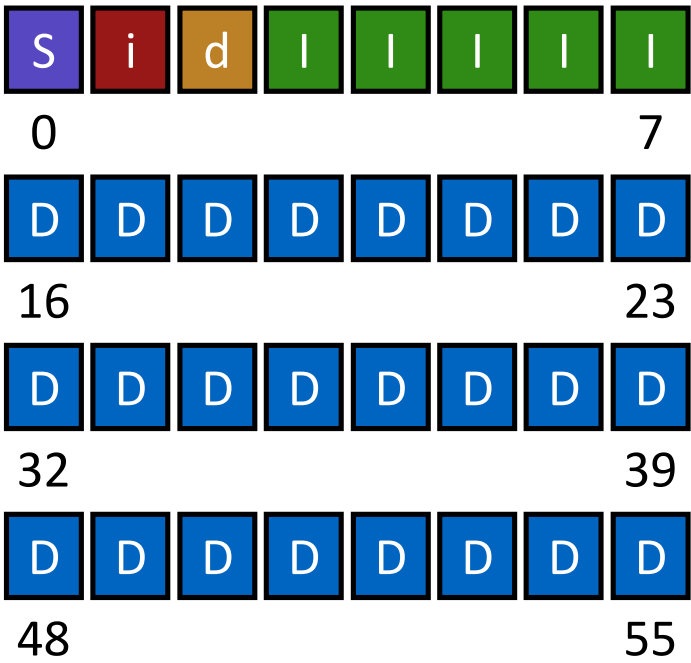
- Fragmentation?
  - Sequential access?
  - Random access?
  - File growth?
  - Metadata overhead?
- + No external fragmentation
  - + Good assuming few large extents
  - + Quick assuming a shallow extent tree
  - + Easy to grow
  - + low, assuming a few extents






# On-Disk FS Layout

- Varies from FS to FS; we 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



 : Data block	 : Data bitmap	 : Superblock
 : Inode block	 : Inode bitmap	

# One 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

inode 16	inode 17	inode 18	inode 19
inode 20	inode 21	inode 22	inode 23
inode 24	inode 25	inode 26	inode 27
inode 28	inode 29	inode 30	inode 31

# 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:
  - Directory is 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.

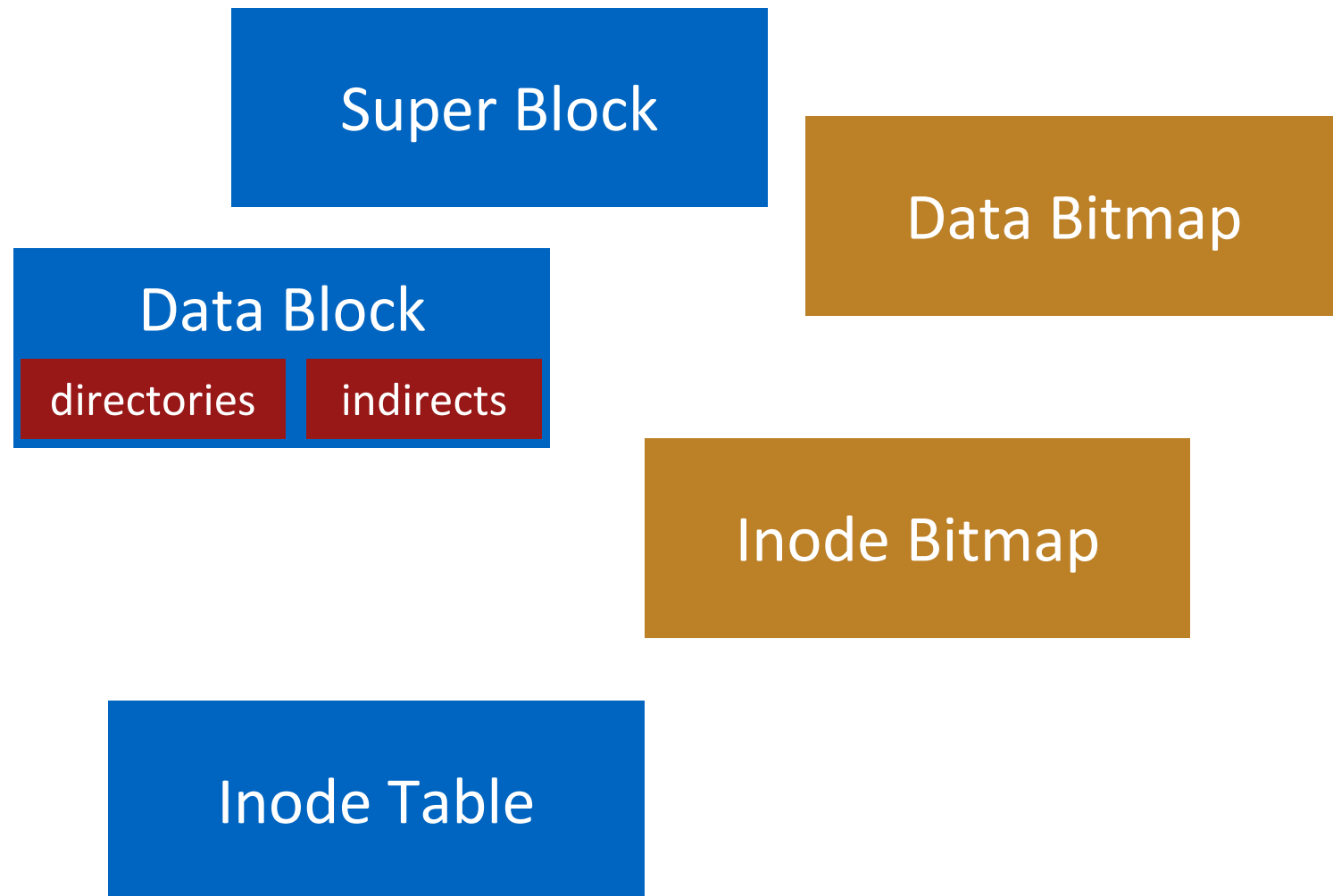
# Free Space Management

- 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
- Store this in superblock

# Summary: On-Disk Structures



# Example 1: create /foo/bar (1)

- Step 1: traverse

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

Verify that bar does not already exist



# Example 1: create /foo/bar (2)

- Step 2: populate inode

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

Why must **read** bar inode block?  
How to initialize inode?

# Example 1: create /foo/bar (3)

- Step 3: update directory

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

Update directory's inode (e.g., size) and data

# Synthesis Example: write to /foo/bar

- Assuming it's already opened

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

Need to allocate a data block assuming  
bar was empty

# Efficiency

- How to avoid so much IO for basic operations?
  - Answer: 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