Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Spring 2024

# File System Implementation

Implementing a File System

- **On-disk structures**
  - How does file system represent files and directories?
  - How to manage various file system metadata?

- **Access methods**
  - What steps should be taken for various file system APIs?
 - `open()`, `read()`, `write()`, `close()`, ...

VSFS: Data Blocks

- **"Very Simple File System"**

  - Divide the disk into blocks (e.g., 4KB)

  - Block size is a multiple of sector size

  - Most of disk blocks are used for storing user data

  - A small portion of the disk is reserved for file system metadata

Data region
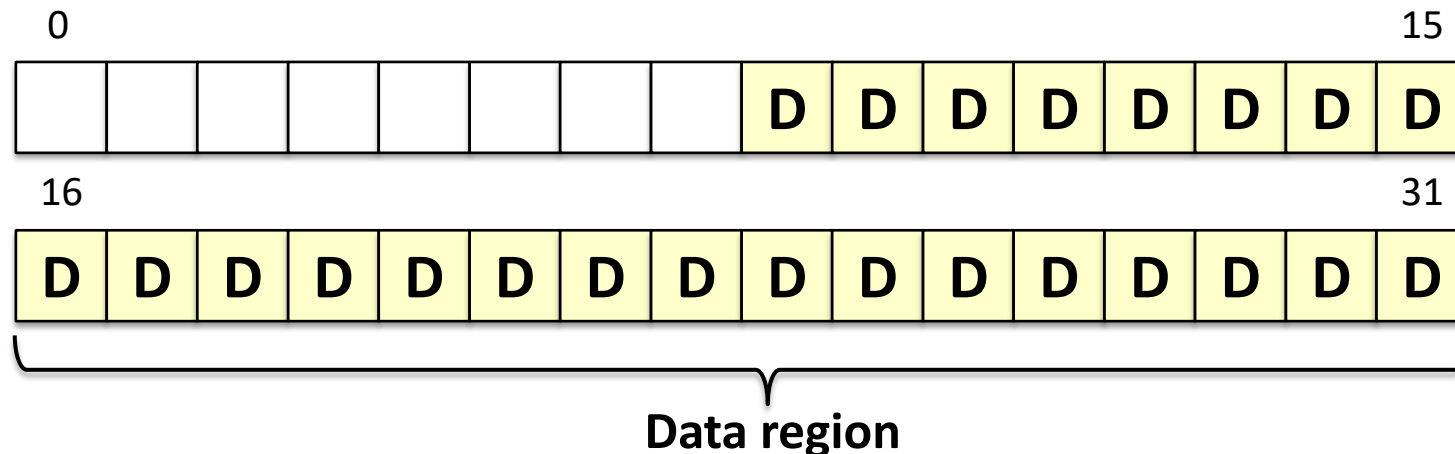
VSFS: Inodes

- **Each inode holds file metadata**
  - The size of an inode is fixed (typically, 128B ~ 256B)
  - For 256B per inode,
    a 4KB block can hold 16 inodes
  - The total 80 inodes with five inode blocks
    = the max # of files


Inode table
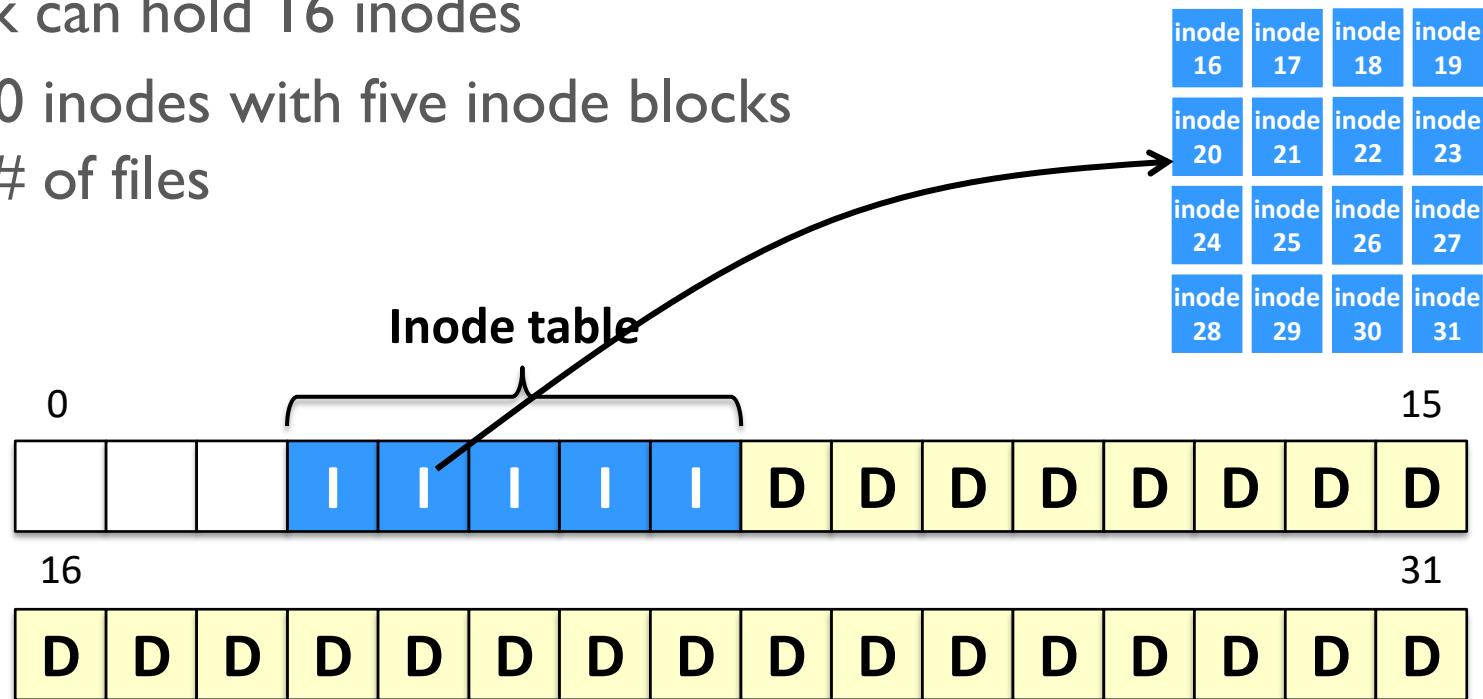
VSFS: Bitmaps

- Data bitmap & Inode bitmap
  - Each bit indicates whether the corresponding block/inode is free (0) or in-use (1)
  - One data bitmap (or inode bitmap) block can support up to 4096*8 data blocks (or inodes)
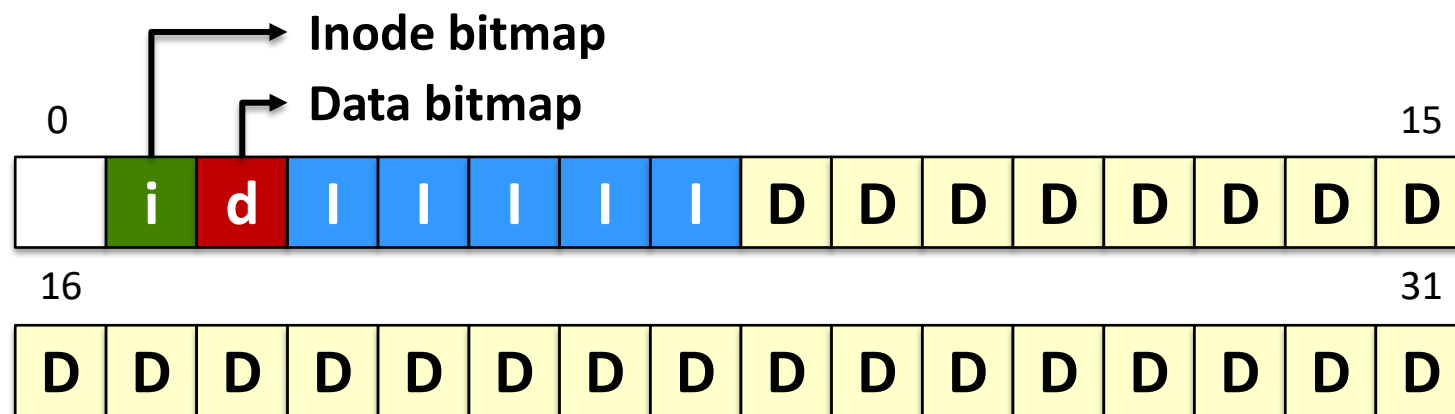
VSFS: Superblock

- **Superblock holds file system metadata**
  - File system type
  - Block size
  - Total number of blocks
  - Number of inodes
  - Number of data / inode bitmap blocks, …
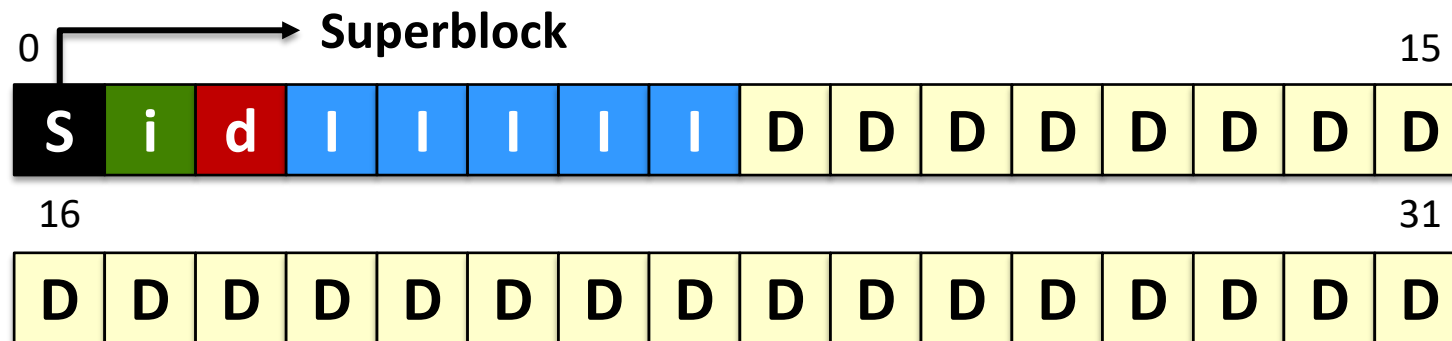
Allocation Strategies

- **How to map files to disk blocks?**
  - Similar to mapping variable-sized address spaces to physical memory
  - Same principle: map logical abstraction to physical resources

- **Issues**
  - The amount of fragmentation (mostly _____)
  - Ability to grow file over time
  - Performance of sequential accesses
  - Speed to find data blocks for random accesses
  - Metadata overhead to track data blocks

Contiguous Allocation

- Allocate each file to contiguous blocks on disk

  - Metadata: <starting block #, length>

  - Feasible and widely used for CD-ROMs

  - Example: IBM OS/360

		A	A	A		B	B	B	B	C	C	C			
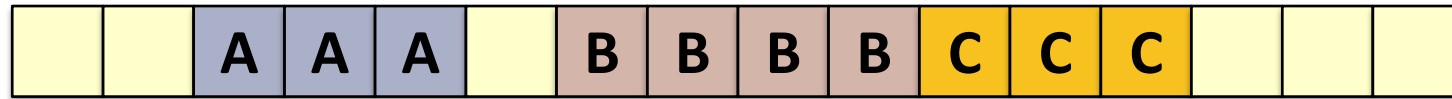
  - Horrible external fragmentation (needs periodic compaction)

  - May not be able to grow file without moving

  - Excellent performance for sequential accesses

  - Simple calculation to perform random accesses

  - Little overhead for metadata
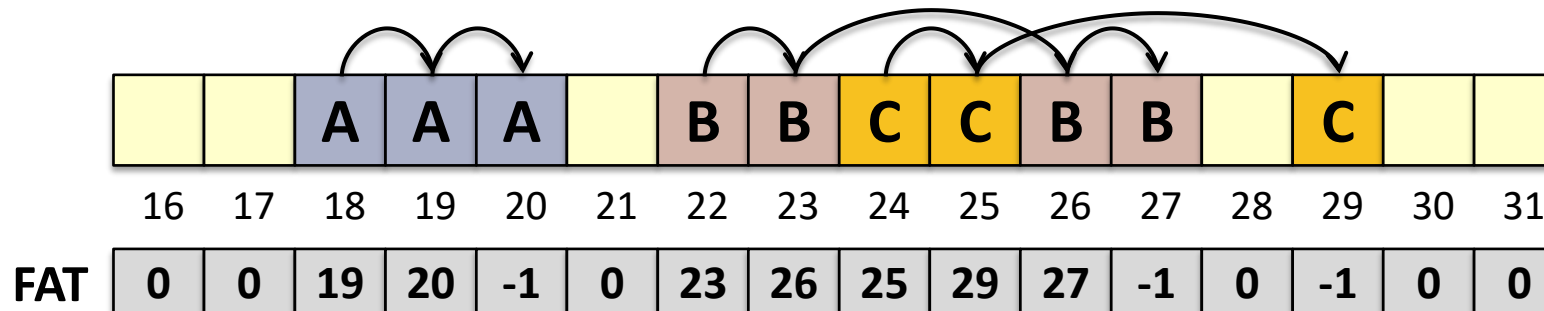
Linked Allocation

- **Allocate linked-list of fixed-sized blocks**
  - Metadata: \<starting block #\>
  - Each block contains pointer to next block
  - Example: TOPS-10, Alto
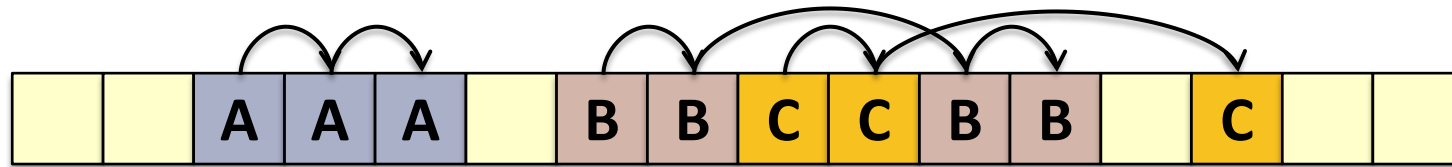
  - No external fragmentation
  - File can grow easily
  - Sequential access performance depends on data layout
  - Poor _____ access performance
  - Waste pointer per block (fragile -- it can be lost or damaged)

File Allocation Table (FAT)

- ■ Variation of linked allocation
  - Keep linked-list information for all files in on-disk FAT
  - FAT is cached in main memory to avoid disk seeks
  - Metadata: <starting block #> + FAT
  - Example: MS-DOS, Windows (FAT12, FAT16, FAT32)

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

FAT	0	0	19	20	-1	0	23	26	25	29	27	-1	0	-1	0	0

  - Improved random access performance
 - Scalability with larger file systems?

Indexed Allocation

- ## Allocate fixed-size blocks for each file

  - Metadata: An array of block pointers

  - Each block pointer points to the corresponding data block

  - No external fragmentation

  - File can grow easily up to max file size

  - Sequential access performance depends on data layout

  - Random accesses supported

  - Large overhead for metadata:
    wasted space for unneeded pointers
    (most files are small)
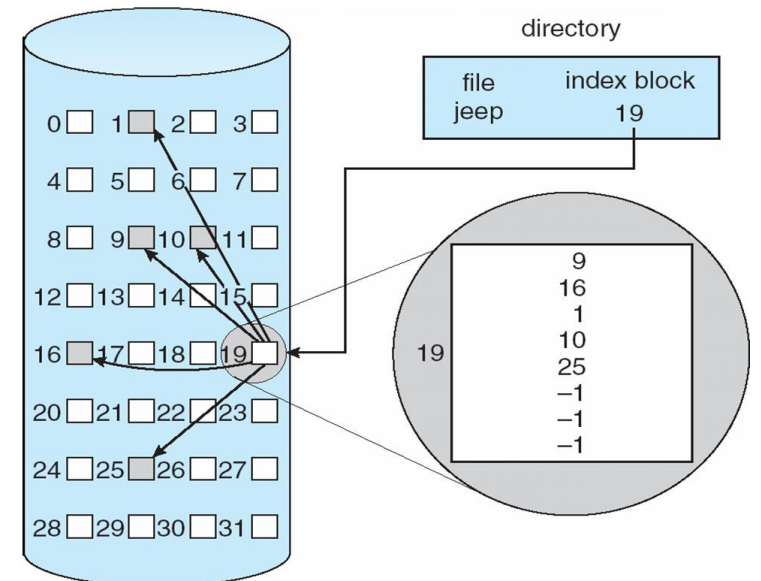
Multi-level Indexing

- **Variation of indexed allocation**

  - Dynamically allocate hierarchy of pointers to data blocks

  - Metadata: small number of direct pointers + indirect pointers

  - Example: Unix FFS, Linux Ext2/3

  - Does not waste space for unneeded pointers

  - Need to read indirect blocks of pointers
    to calculate addresses (extra disk read)
    - Keep indirect blocks cached in main memory
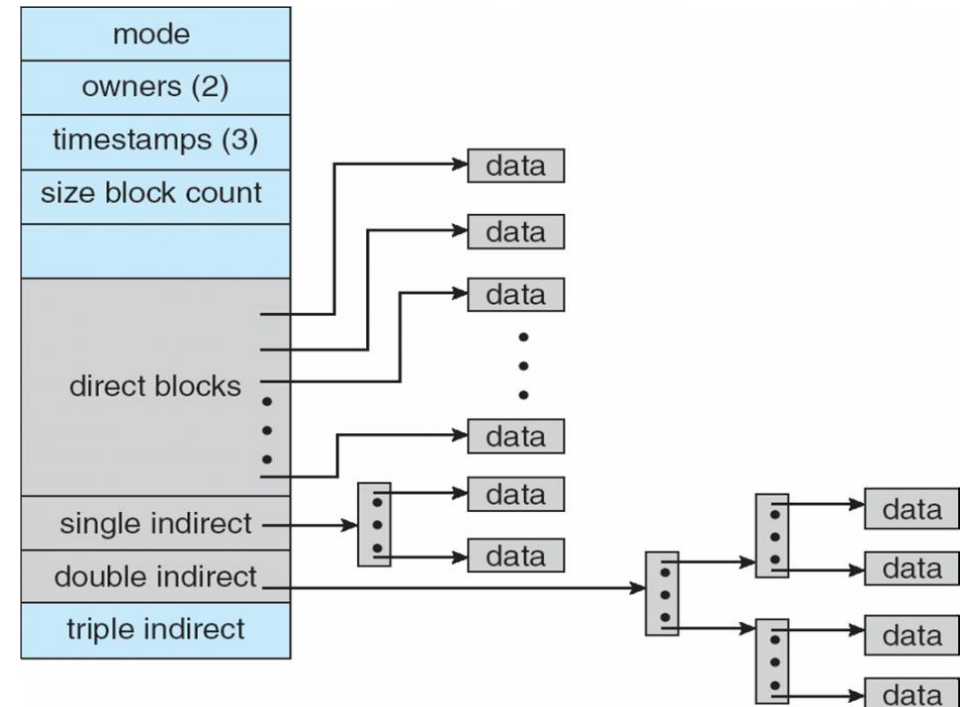
Multi-level Indexing in VSFS

- Configurations
  - An inode has 12 direct pointers and 1 single indirect pointer
  - 4-byte disk address: 1024 pointers per 4KB block
  - Max file size = (12 + 1024) * 4KB = 4144KB
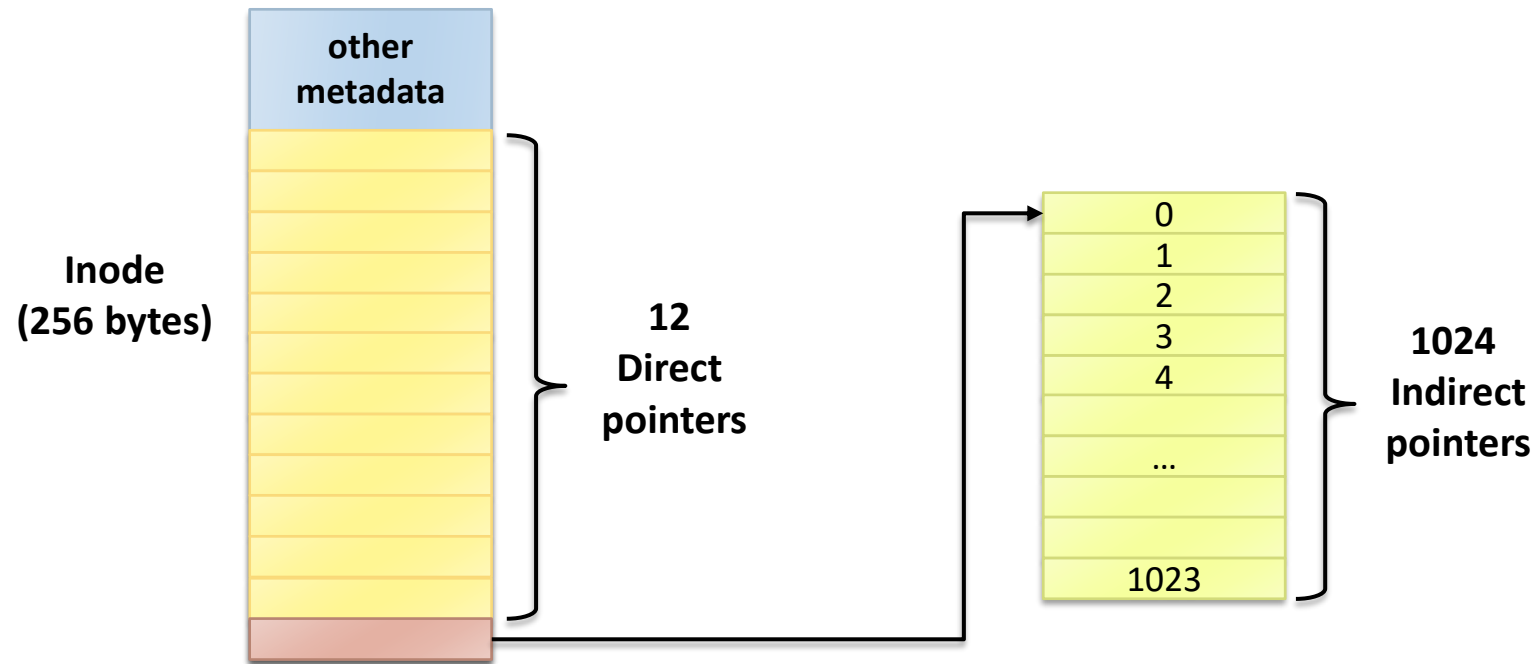
Extent-based Allocation

- **Allocate multiple contiguous regions (extents) per file**
  - Organize extents into multi-level tree structure (e.g., B+tree)
  - Each leaf node: <logical block #, physical block #, extent size>
  - Example: Linux Ext4

  - Reasonable amount of external fragmentation
  - Still good sequential performance
  - Some calculations needed for random accesses
  - Relatively small metadata overhead
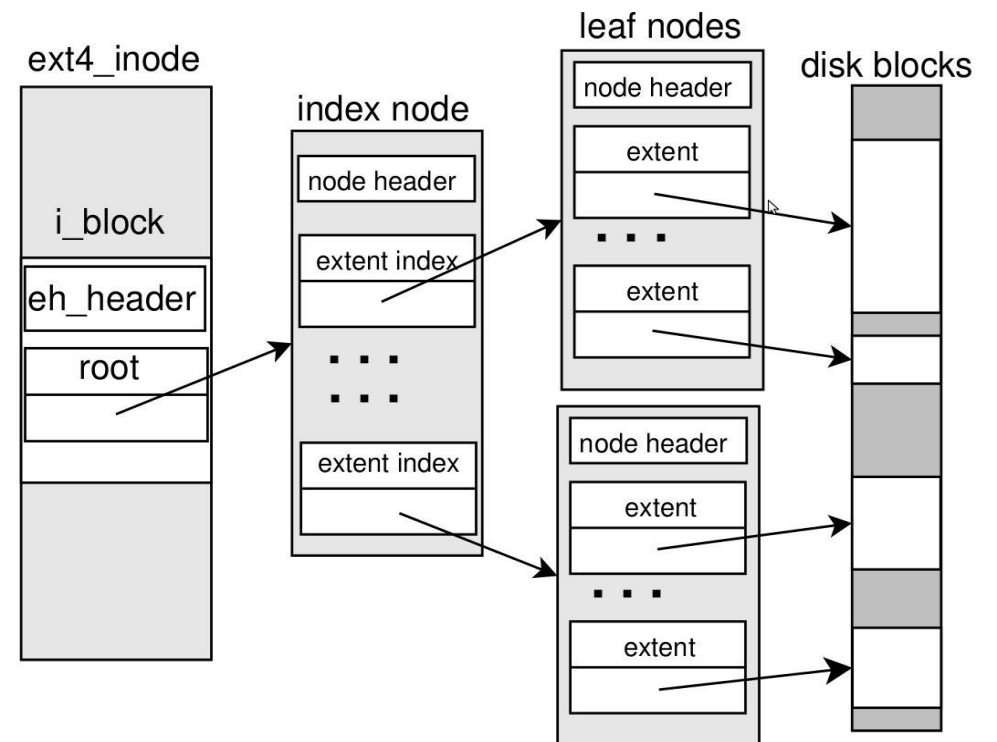
Directory Organization

■ **Common design**

- Directory is a special file containing directory entries

- Large directories just use multiple data blocks

- Use bits in inode to distinguish directories from files

■ **Table (fixed length entries) or linear list:**

- Requires a linear search to find an entry

■ **Tree:**

- Entries may be sorted to decrease the average search time and to produce a sorted directory listing easily

■ **Hash table:**

- Fast, but should be scalable as the number of files increases

VSFS: Directory

- **A linear list of <file name, inode number>**
  - Similar to Linux Ext2 directory
  - Supports variable-sized names
  - Example: `/dir`
    - Inode number for `/dir`?
    - Inode number for the root directory?

inode number	record length	name length	name						
5	12	2	.	\0	\0	\0			
2	12	3	.	.	\0	\0			
12	12	4	f	o	o	\0			
0	12	4	b	a	r	\0	*<deleted entry>*		
24	16	7	f	o	o	b	a	r	\0 \0

Reading a File

- Open /foo/bar and read three blocks

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read	read	read	read	read			
read()					read, write			read		
read()					read, write				read	
read()					read, write					read

Writing a File

- Create /foo/bar and write three blocks

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)		read write	read	read	read write write	read	read write	write		
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write