

# Lecture 19: File System Implementation

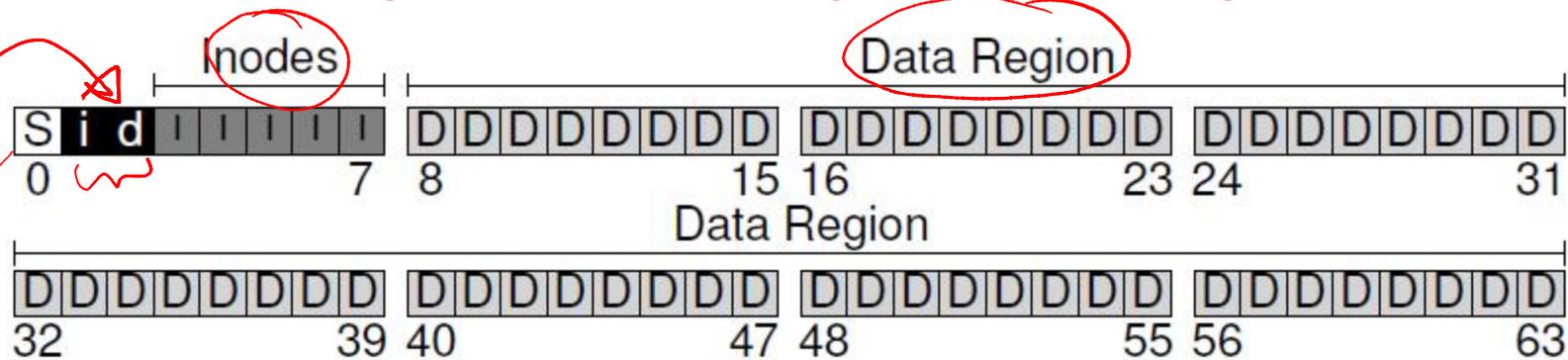
Mythili Vutukuru

IIT Bombay

# File System

- An organization of files and directories on disk
- OS has one or more file systems
- Two main aspects of file systems
  - Data structures to organize data and metadata on disk
  - Implementation of system calls like open, read, write using the data structures
- Disks expose a set of blocks (usually 512 bytes)
- File system organizes files onto blocks
  - System calls translated into reads and writes on blocks

# Example: a simple file system

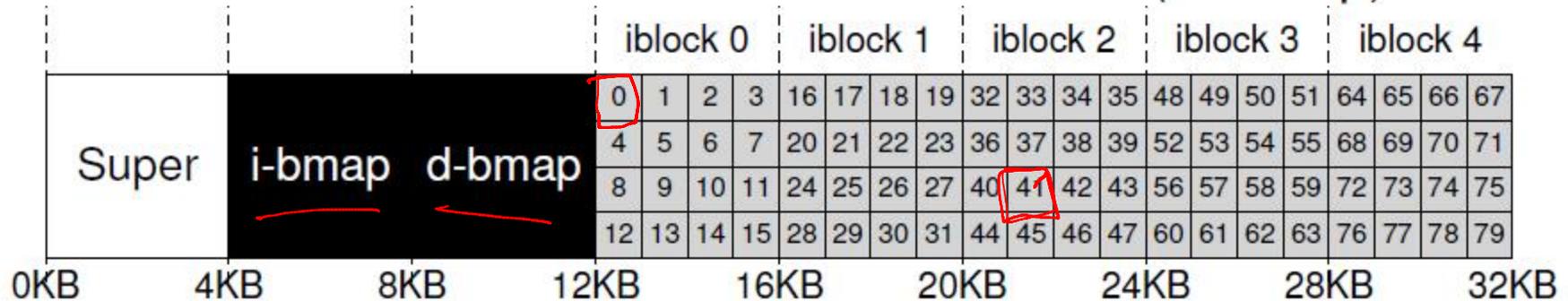


- Data blocks: file data stored in one or more blocks
- Metadata about every file stored in inode
  - Location of data blocks of a file, permissions etc.
- Inode blocks: each block has one or more inodes
- Bitmaps: indicate which inodes/data blocks are free
- Superblock: holds master plan of all other blocks (which are inodes, which are data blocks etc.)

# Inode table

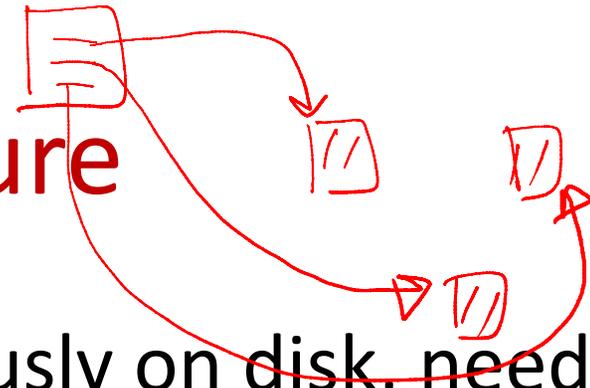
- Usually, inodes (index nodes) stored in array
  - Inode number of a file is index into this array

The Inode Table (Closeup)



- What does inode store?
  - File metadata: permissions, access time, etc.
  - Pointers (disk block numbers) of file data

# Inode structure

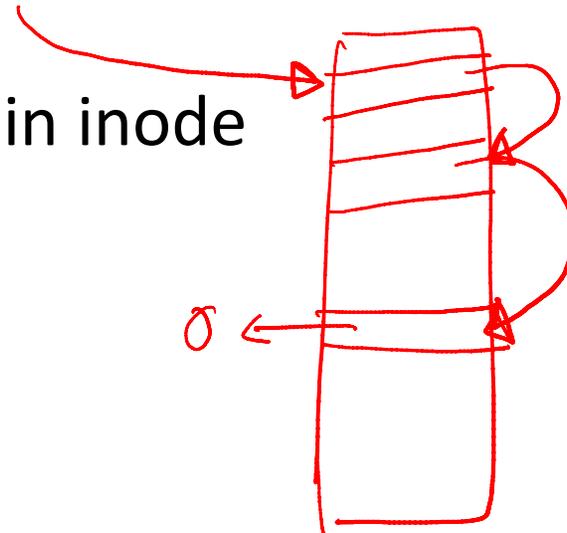


- File data not stored contiguously on disk, need to track multiple block numbers of a file
- How does inode track disk block numbers?
  - Direct pointers: numbers of first few blocks are stored in inode itself (suffices for small files)
  - Indirect block: for larger files, inode stores number of indirect block, which has block numbers of file data
  - Similarly, double and triple indirect blocks (multi-level index)

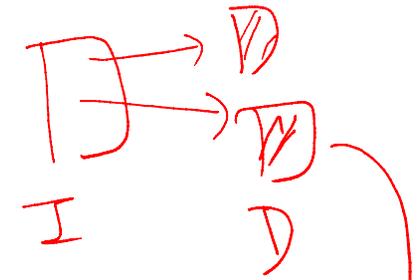


# File Allocation Table (FAT)

- Alternate way to track file blocks
- FAT stores next block pointer for each block
  - FAT has one entry per disk block
  - Entry has number of next file block, or null (if last block)
  - Pointer to first block stored in inode



# Directory structure



- Directory stores records mapping filename to inode number, e.g., as shown below

<u>inum</u>	<u>reclen</u>	<u>strlen</u>	<u>name</u>
<u>5</u>	12	2	.
<u>2</u>	12	3	..
12	12	4	foo
13	<u>12</u>	<u>4</u>	<u>bar</u>
24	<u>36</u>	28	foobar_is_a_pretty_longname

- Linked list of records, or more complex structures (hash tables, binary search trees etc.)
- Directory is a special type of file and has inode and data blocks (which store the file records)

# Free space management

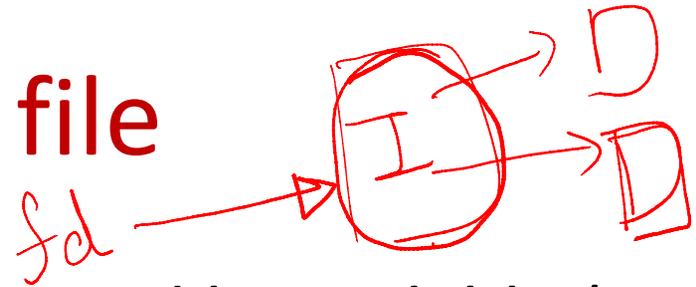
- How to track free blocks?



- Bitmaps, for inodes and data blocks, store one bit per block to indicate if free or not
- Free list, super block stores pointer to first free block, a free block stores address of next block on list
- More complex structures can also be used



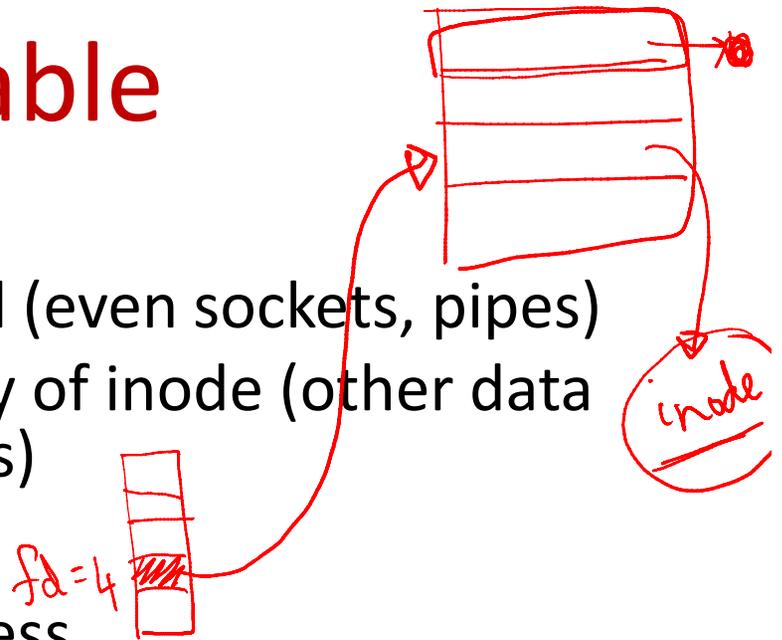
# Opening a file



- Why open? To have the inode readily available (in memory) for future operations on file
  - Open returns fd which points to in-memory inode
  - Reads and writes can access file data from inode
- What happens during open? /a/b/c.txt
  - The pathname of the file is traversed, starting at root
  - Inode of root is known, to bootstrap the traversal
  - Recursively do: fetch inode of parent directory, read its data blocks, get inode number of child, fetch inode of child. Repeat until end of path
  - If new file, new inode and data blocks will have to be allocated using bitmap, and directory entry updated

# Open file table

- Global open file table
  - One entry for every file opened (even sockets, pipes)
  - Entry points to in-memory copy of inode (other data structures for sockets and pipes)
- Per-process open file table
  - Array of files opened by a process
  - File descriptor number is index into this array
  - Per-process table entry points to global open file table entry
  - Every process has three files (standard in/out/err) open by default (fd 0, 1, 2)
- Open system call creates entries in both tables and returns fd number



# Reading and writing a file

- For reading/writing file
  - Access in-memory inode via file descriptor
  - Find location of data block at current read/write offset
  - Fetch block from disk and perform operation
  - Writes may need to allocate new blocks from disk using bitmap of free blocks
  - Update time of access and other metadata in inode

# Virtual File System

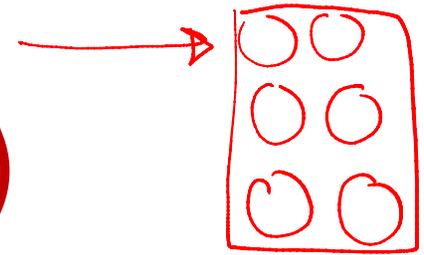
Sys Call

VFS

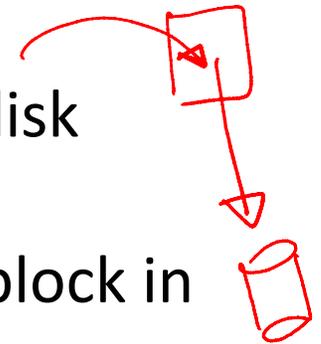
FS

- File systems differ in implementations of data structures (e.g., organization of file records in directory)
- Linux supports virtual file system (VFS) abstraction
- VFS looks at a file system as objects (files, directories, inodes, superblock) and operations on these objects (e.g., lookup filename in directory)
- System call logic is written on VFS objects
- To develop a new file system, simply implement functions on VFS objects and provide pointers to these functions to kernel
- Syscall implementation does not have to change with file system implementation details

# Disk buffer cache (1)



- Results of recently fetched disk blocks are cached
  - LRU to evict if cache is full
- File system issues block read/write requests to block numbers via buffer cache
  - If block in cache, served from cache, no disk I/O
  - If cache miss, block fetched to cache and returned to file system
- Writes are applied to cache block first
  - Synchronous/write-through cache writes to disk immediately
  - Asynchronous/write-back cache stores dirty block in memory and writes back after a delay



## Disk buffer cache (2)

- Unified page cache in OS
  - Free pages allocated to both processes and disk buffer cache from common pool
- Two benefits
  - Improved performance due to reduced disk I/O (one disk access for multiple reads and writes)
  - Single copy of block in memory (no inconsistency across processes)
- Some applications like databases may avoid caching altogether, to avoid inconsistencies due to crashes: direct I/O