Lecture 25: File System Implementation

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

```
While (STATUS == BUSY)
     ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
     (starts the device and executes the command)
While (STATUS == BUSY)
     ; // wait until device is done with your request
```

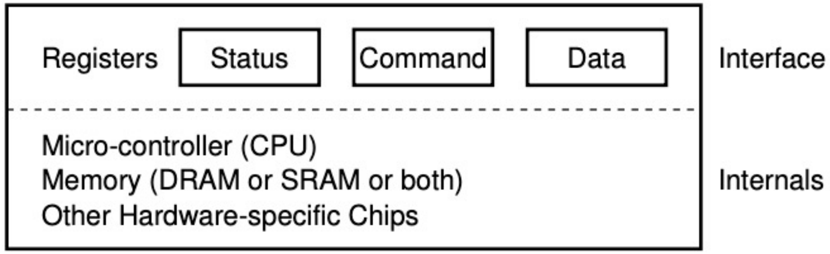
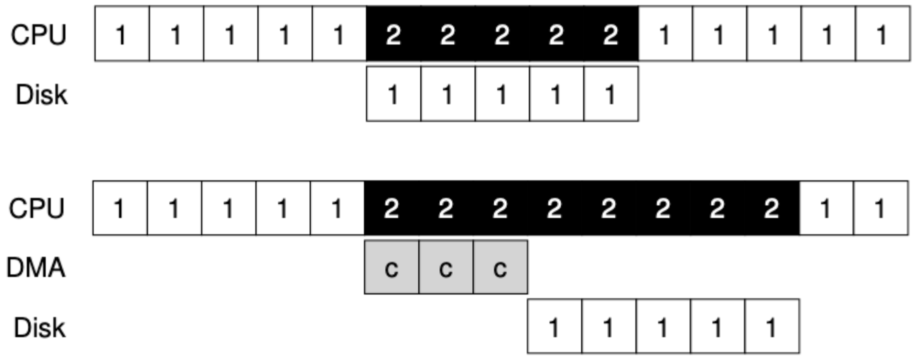
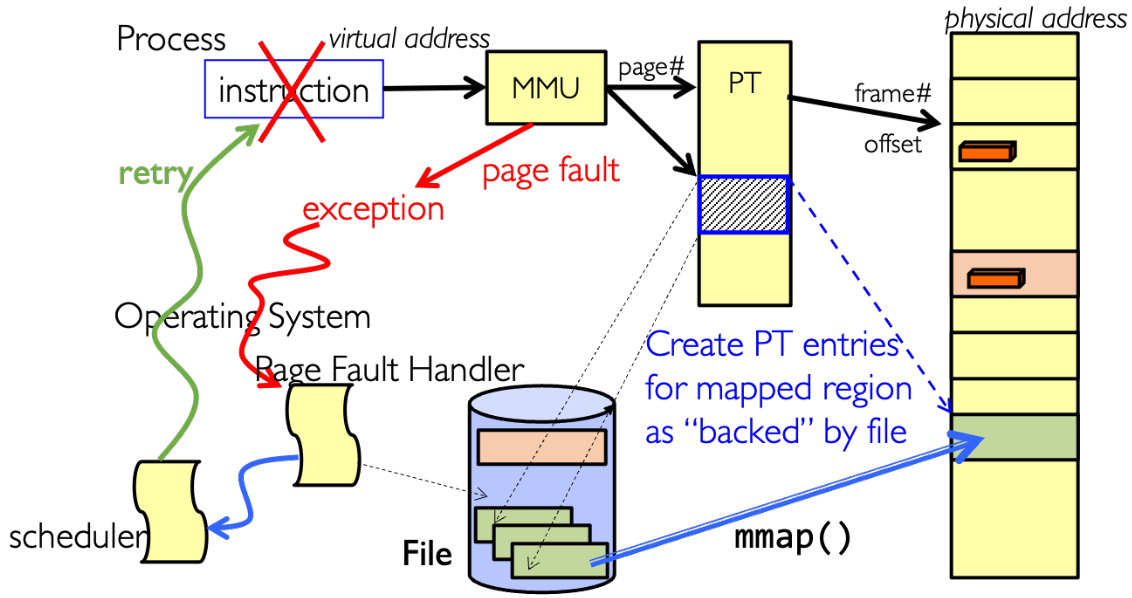


Figure 36.3: **A Canonical Device**



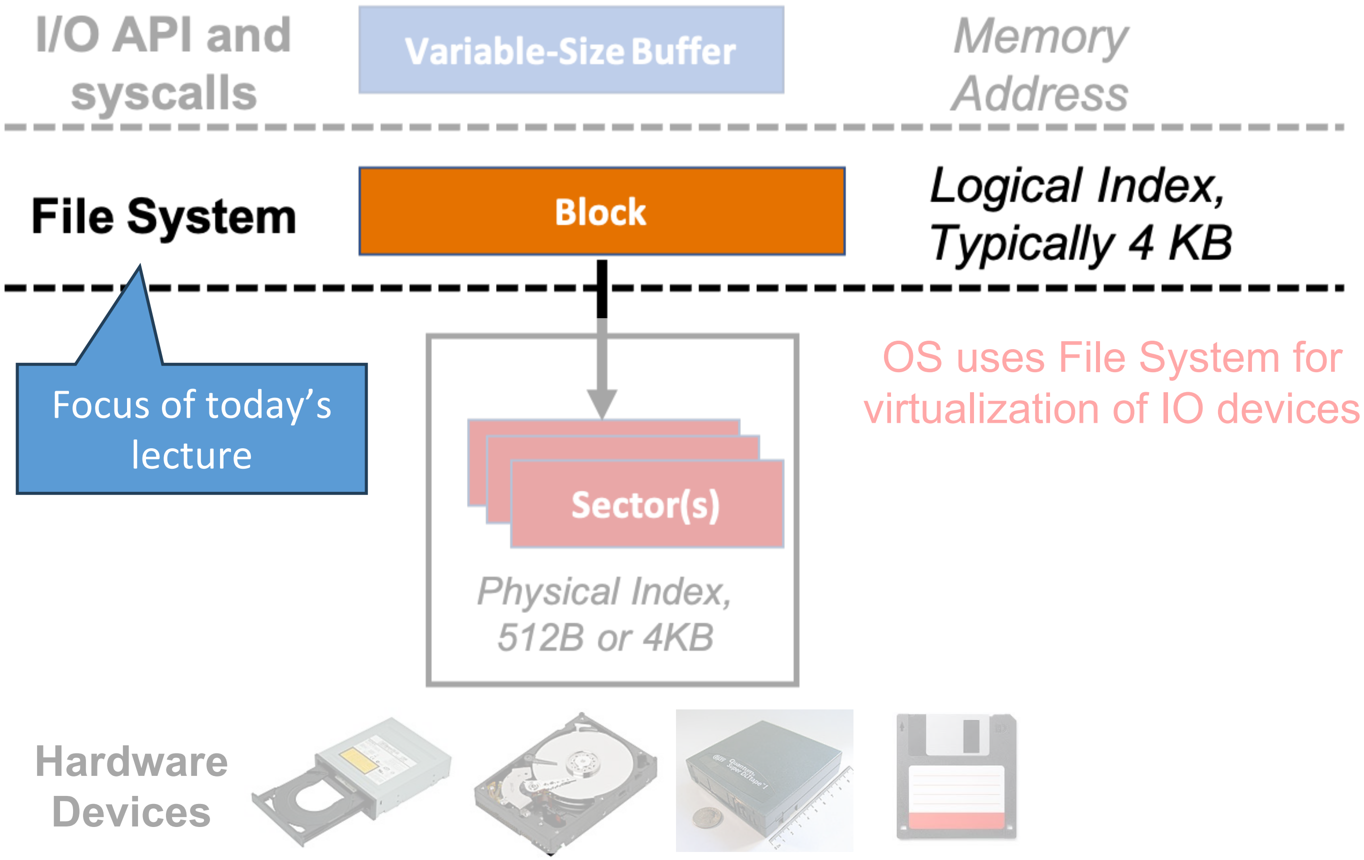


- Prototype of a simple IO device
- Using interrupts to avoid polling overheads
- Using DMA to avoid wastage of CPU cycles during copying of data
- Paging with memory mapped files

Today's Class

- Design of FAT and FFS file systems
- Buffer cache
- Dealing with system crashes

File System



- User's view of File:
  - Durable Data Structures
- System's view of File (system call interface):
  - Collection of Bytes (UNIX)
  - Oblivious to specific data structures user wants to store
- System's view of file (inside OS):
  - File is a collection of blocks

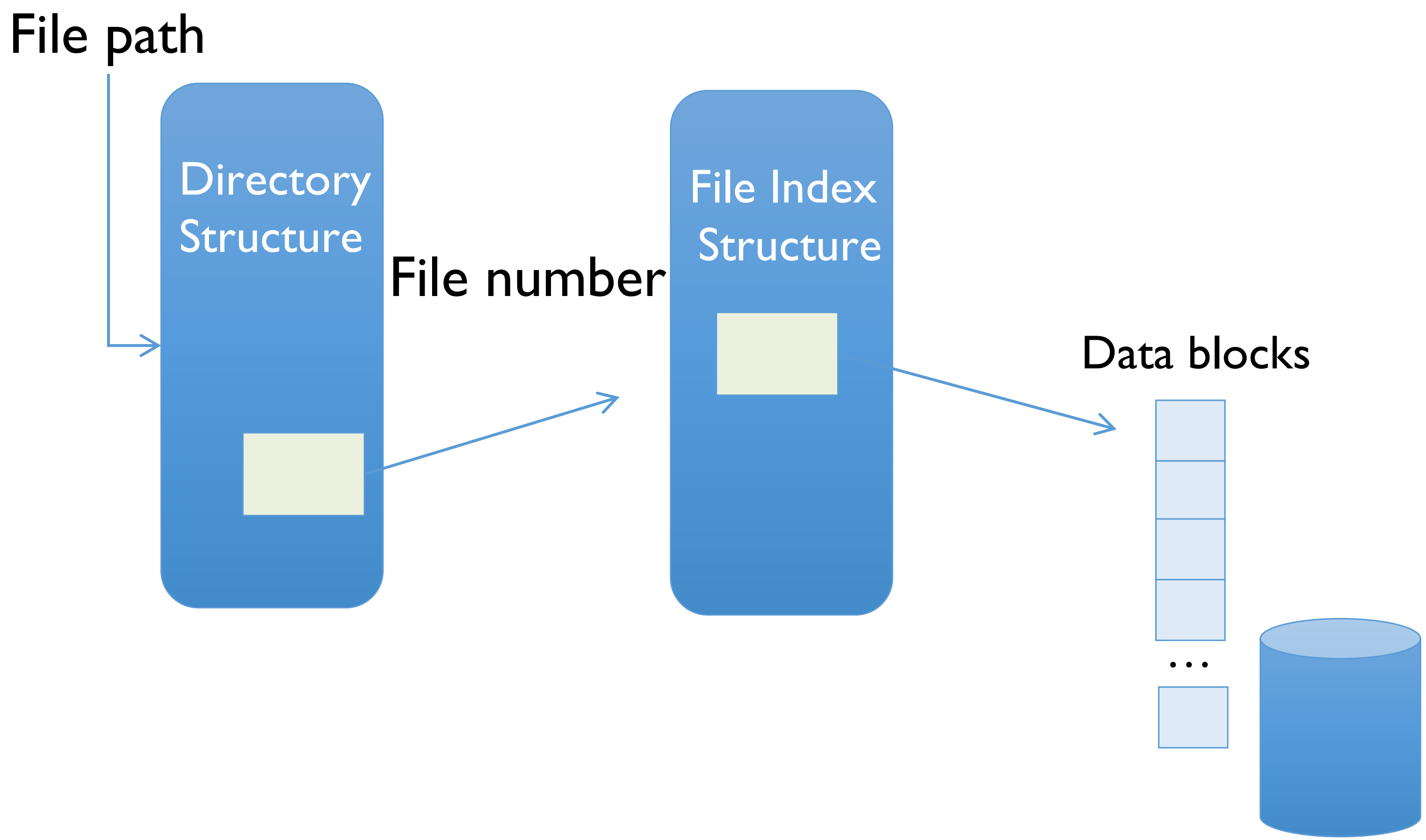
Building a File System

- File organization
  - Organize files by names with directories
- Protection
  - Provide access restriction
- Fast access
  - Access to disk is several orders of magnitude slower than DRAM access
- Reliability
  - Keep files intact despite crashes, hardware failures, etc.

What Does the File System Needs?

- Track free disk blocks
  - Need to know where to put newly written data
- Track which blocks contain data for which files
  - Need to know where to read a file from
- Track files in a directory
  - Find list of file's blocks given its name
- Where do we maintain all of this?
  - Somewhere on disk

Basic File System Components



Components of a File System

file name → *file number* → *Storage block*

directory *offset* index structure

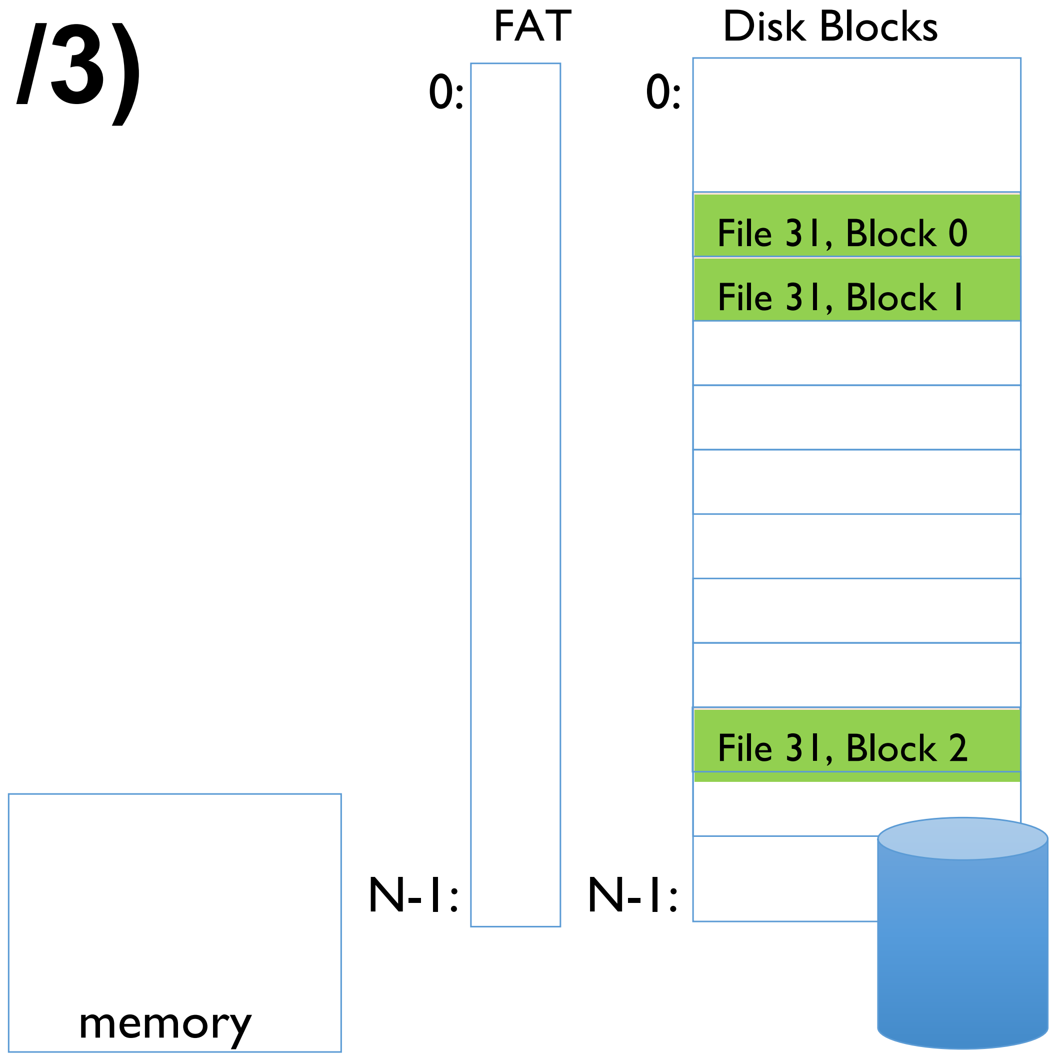
- Open performs **Name Resolution**
  - Translates pathname into a "file number"
    - Used as an "index" to locate the blocks
  - Creates a file descriptor in PCB within kernel
  - Returns a file descriptor (int) to user process
- Read and Write operation on the file number
  - Use file number as an "index" to locate the blocks on disk

FAT (File Allocation Table)

- FAT is a file system architecture which is really simple and robust
  - Used in several places (MS-DOS, Windows (sometimes) and External drives)
- It uses linked allocation, where each file is a linked list of blocks on disk
  - These blocks could be scattered any where on the disk
- What is benefit of linked allocation v/s contiguous allocation of blocks on disk?
  - Each file occupying a set of contiguous blocks on the disk is really simple to implement but doesn't allow file growth
  - To avoid file growth, extra space must be allocated upfront leading to space wastage (disk fragmentation)

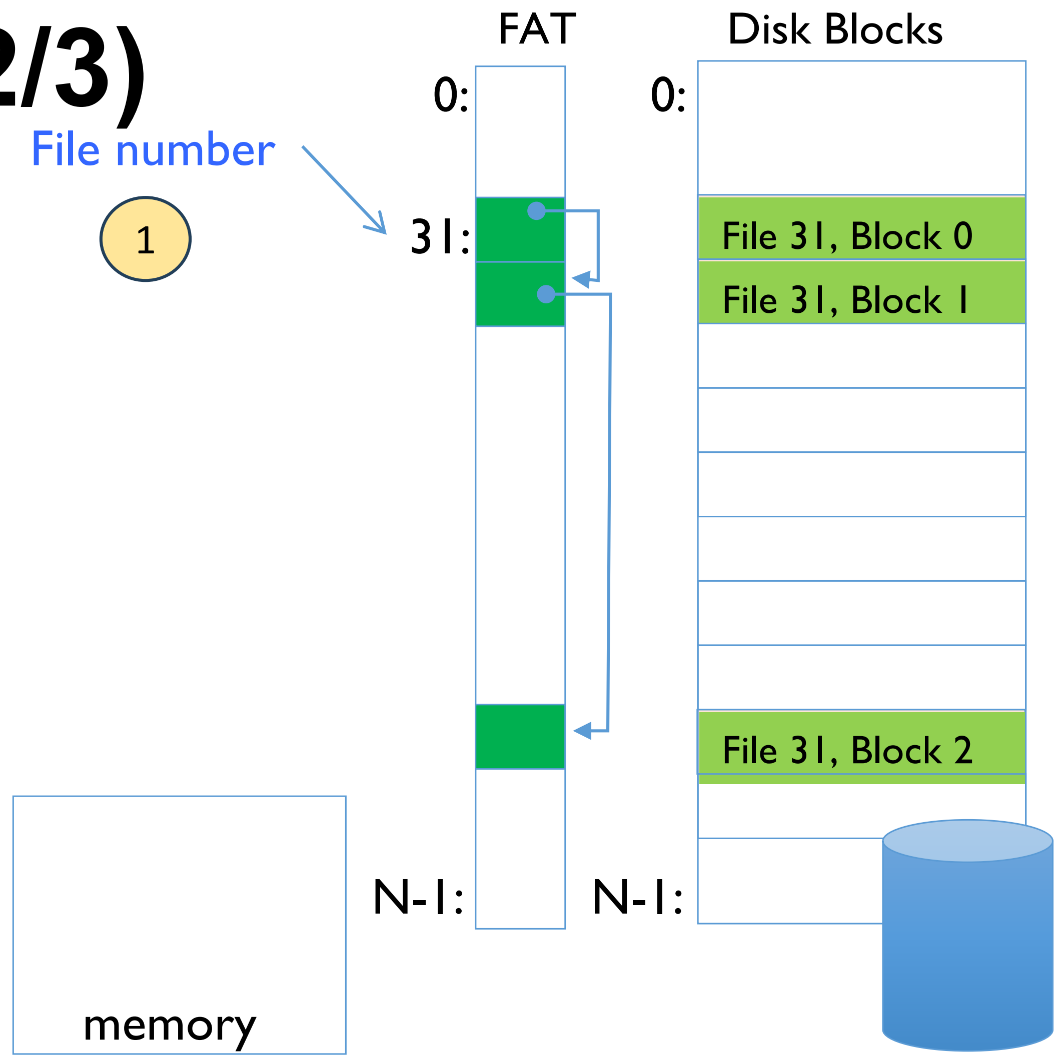
FAT: Reading File (1/3)

- Example: read file 31, block 2 (4KB blocks)



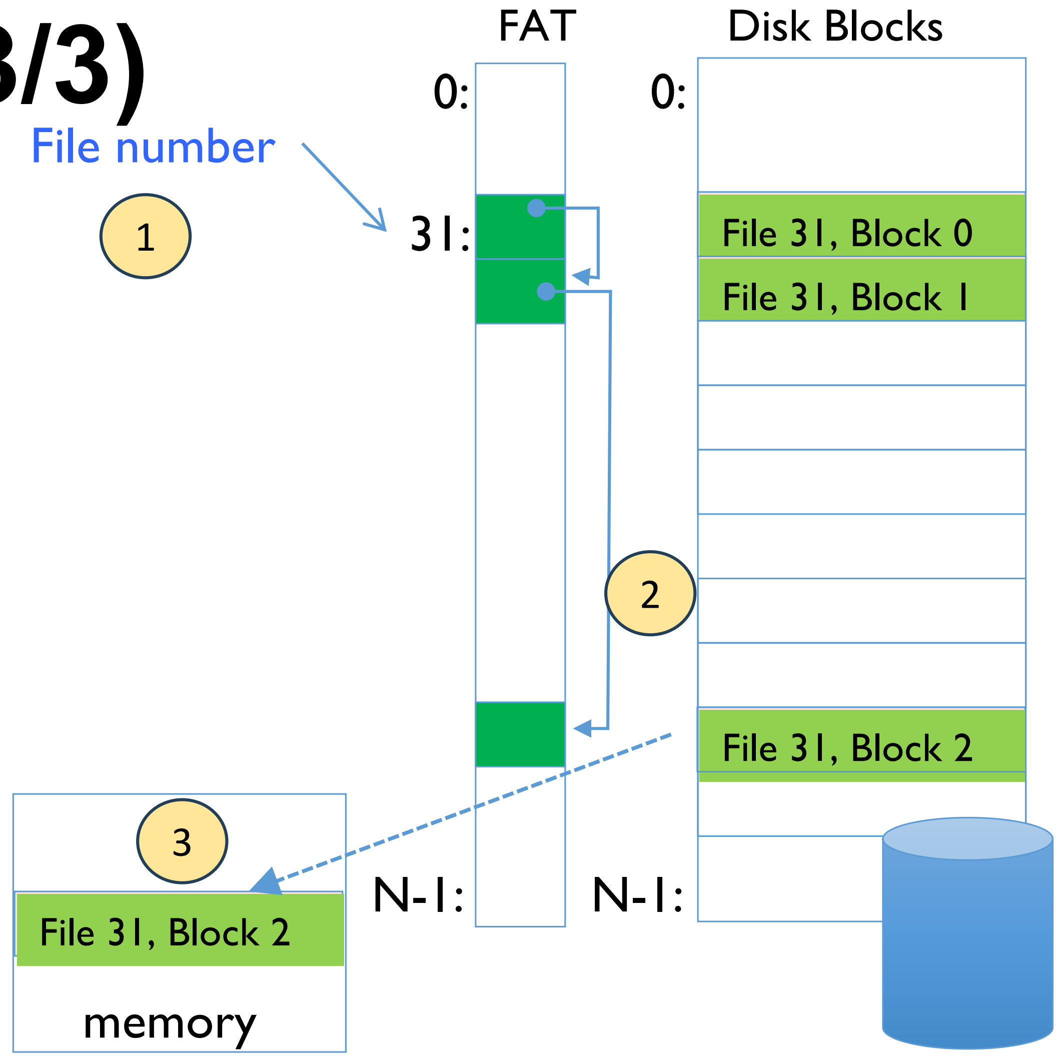
FAT: Reading File (2/3)

- Example: read file 31, block 2
  1. Index into FAT with file num
     - Simple way to store blocks of a file: **Linked List** structure
     - File number is just the **first block**
     - One entry in table per data block
     - FAT contains pointer to **the next block** for each entry (or special END value)



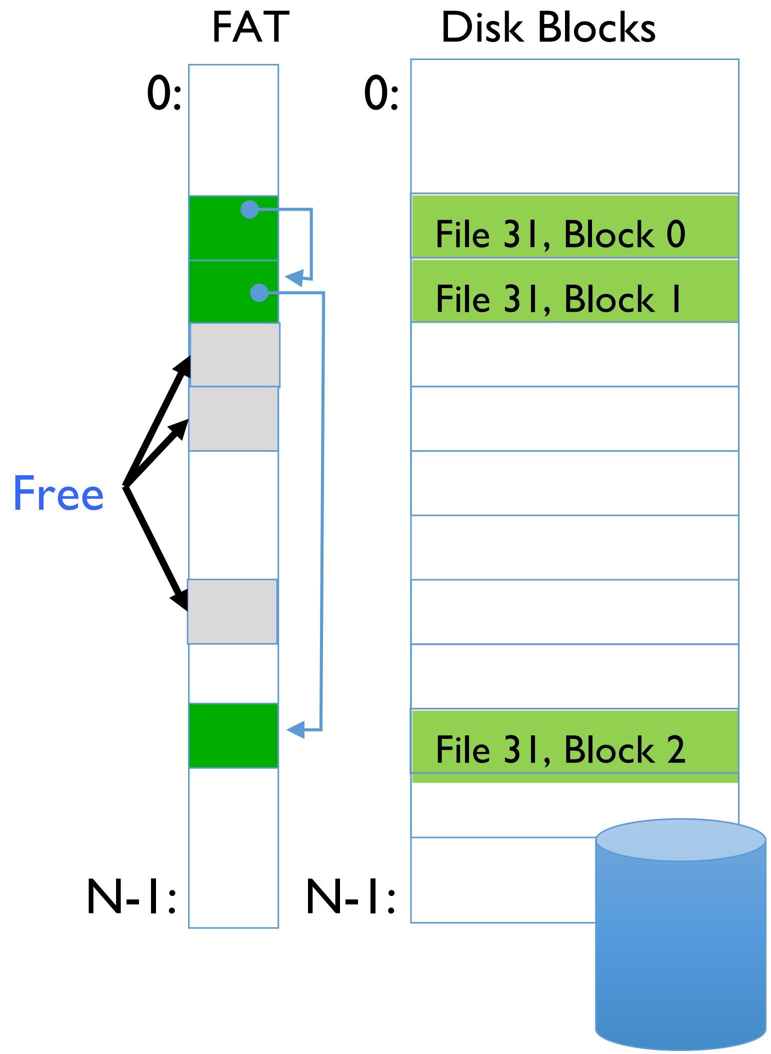
FAT: Reading File (3/3)

- Example: read file 31, block 2
  1. Index into FAT with file num
  2. Follow linked list to block 2
  3. Read block from disk into mem



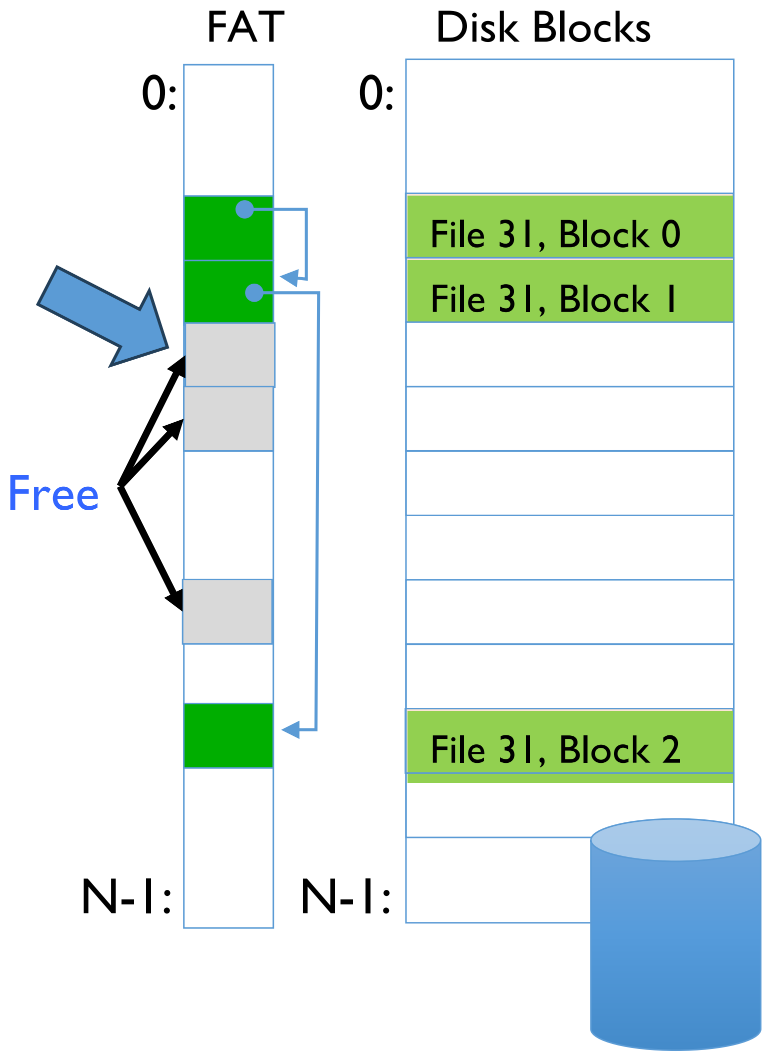
FAT: Expanding File (1/4)

- Entries in table corresponding to free block have value `0`
- Must scan through FAT to find free space



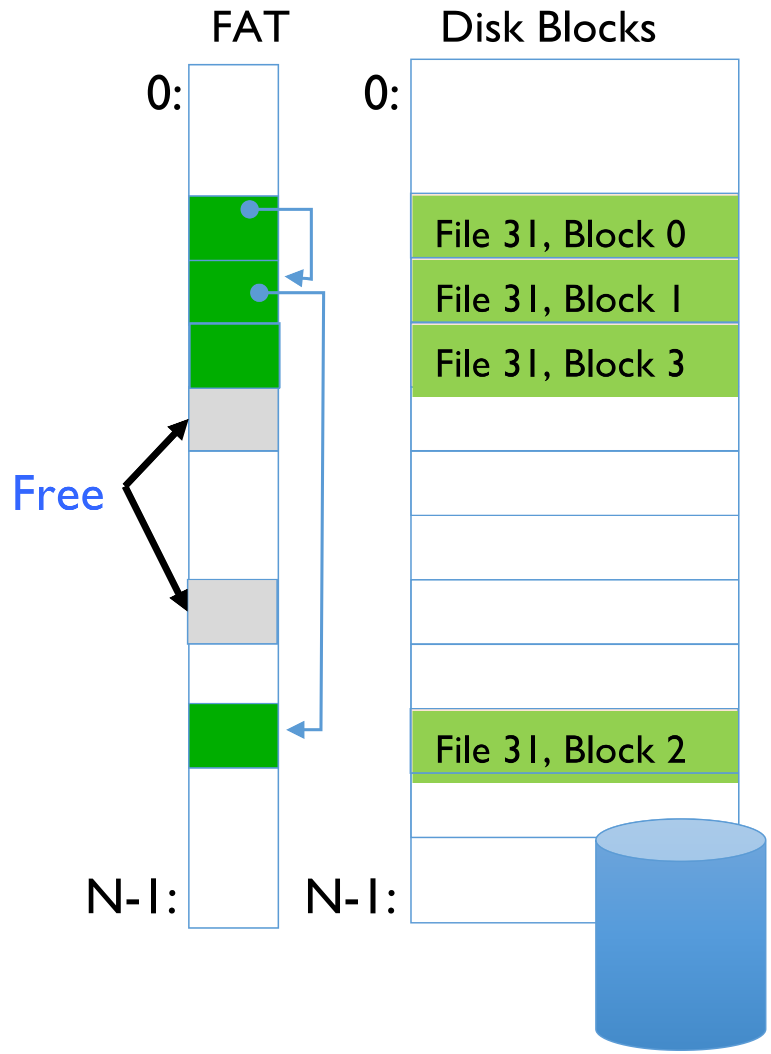
FAT: Expanding File (2/4)

- Find a new free block



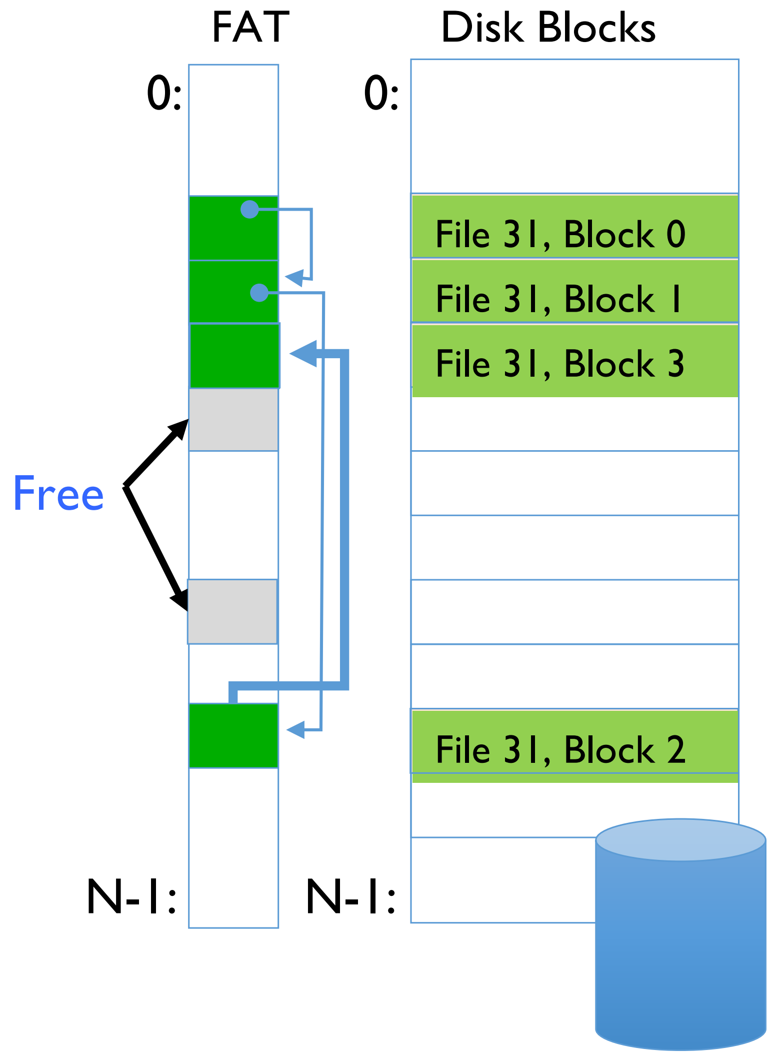
FAT: Expanding File (3/4)

- Add new content of the file at Block 3 corresponding to this FAT entry



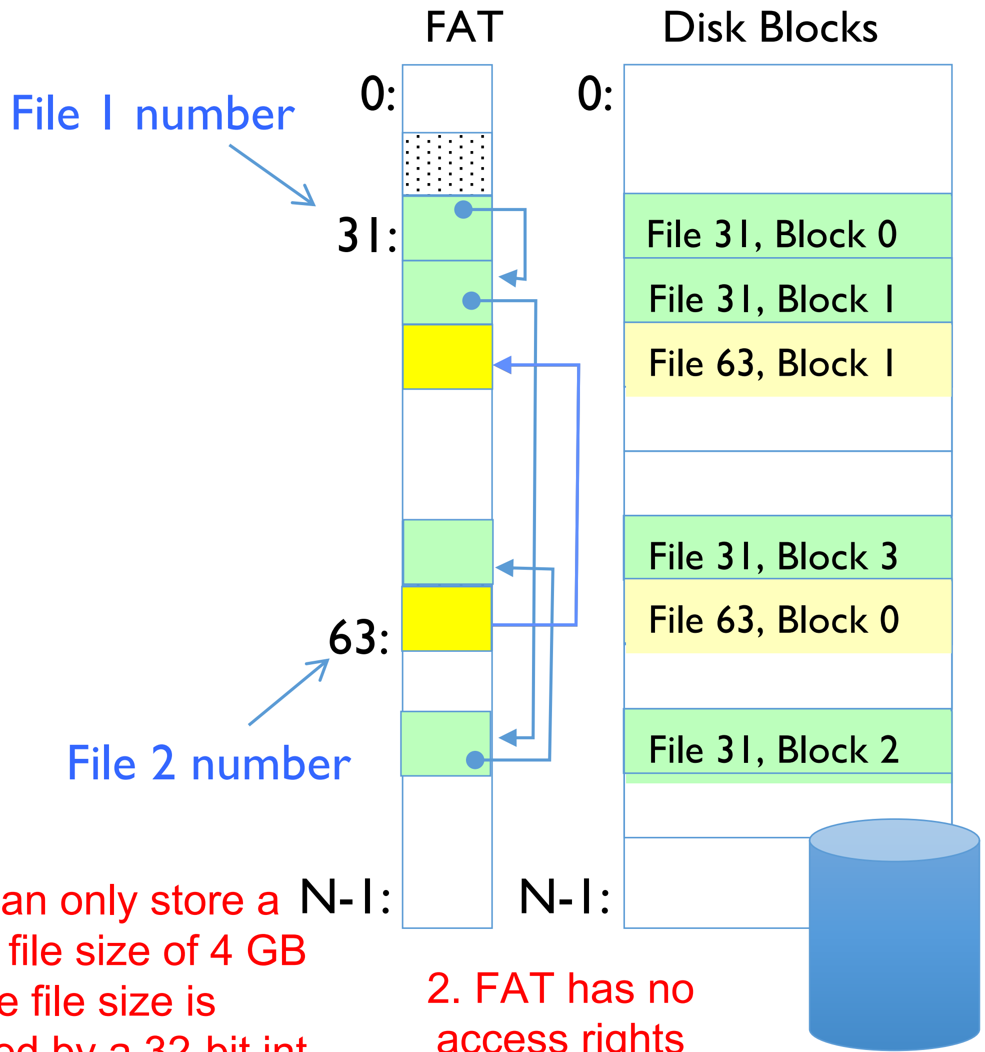
FAT: Expanding File (4/4)

- Link the new FAT entry with its predecessor



Storing the FAT

- Saved to disk when system is shut down
- **Copied into memory when OS is running**
  - Makes accesses, updates fast
  - Otherwise lots of random reads to locate the blocks of a file
- When drive is formatted, make all FAT entries 0



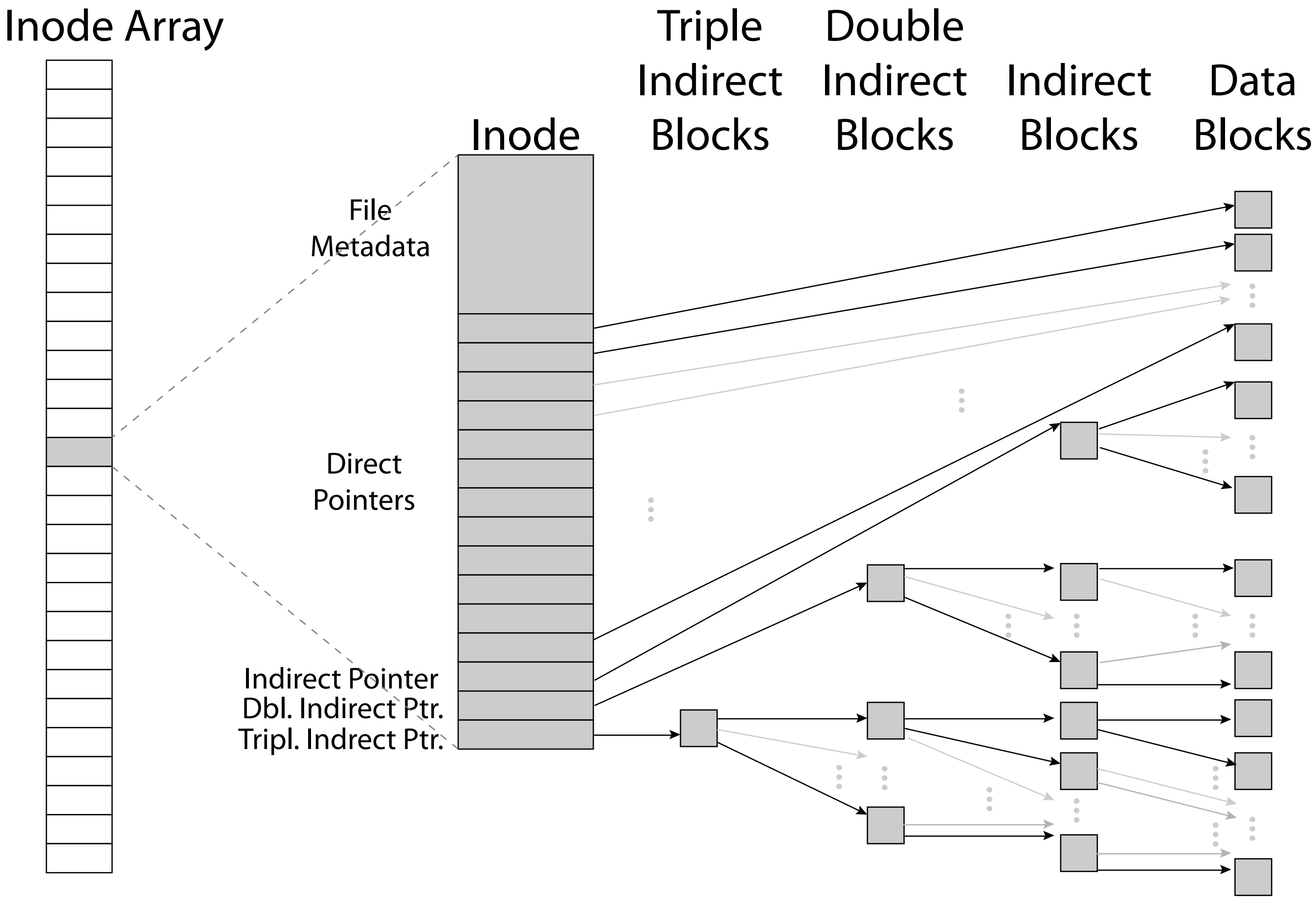
1. FAT can only store a maximum file size of 4 GB as the file size is represented by a 32-bit int

2. FAT has no access rights

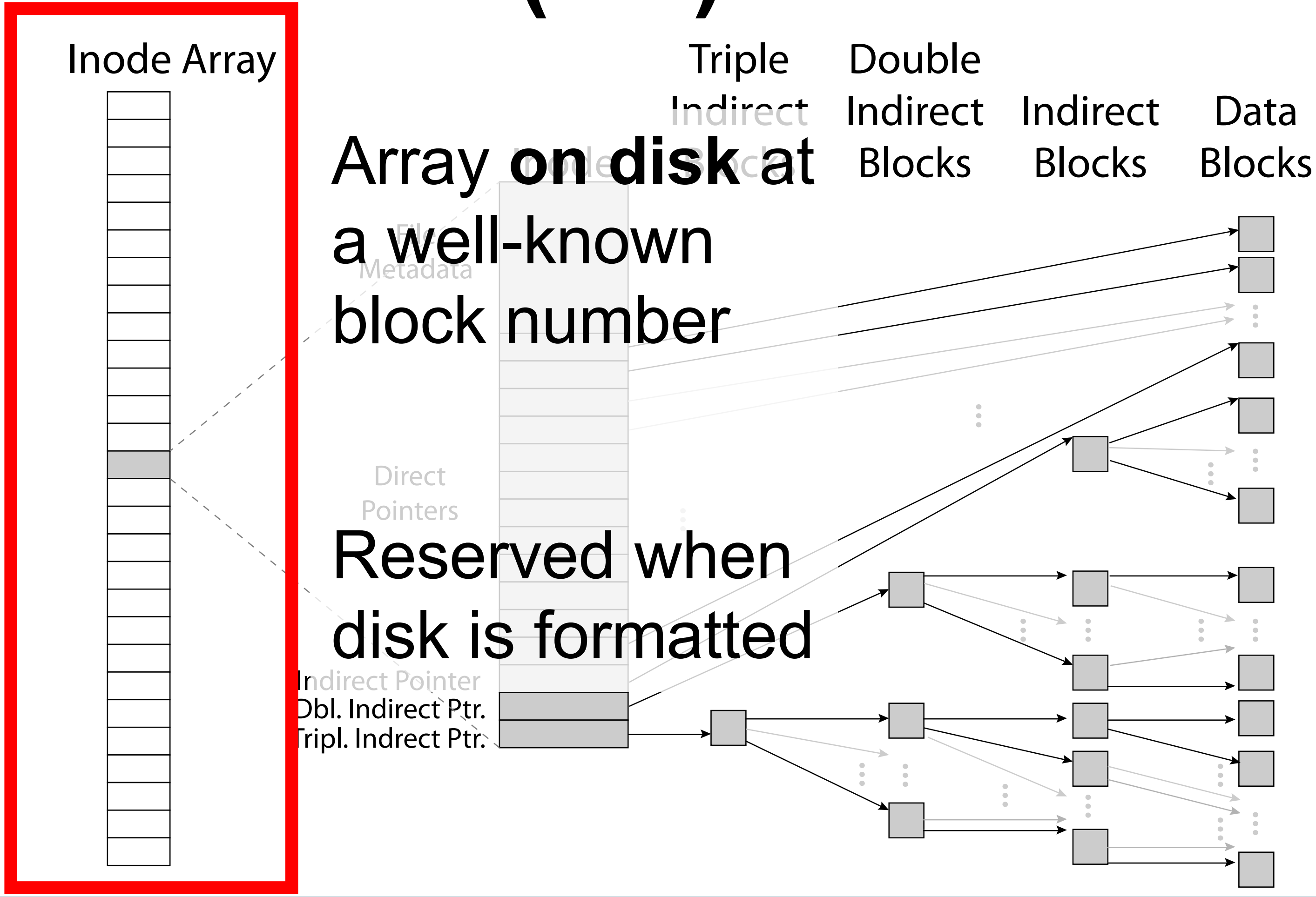
Inodes in Unix

- File number is index into a set of inode arrays
  - Suppose an Inode occupies 128 bytes
    - Byte offset of Inode-1000 = 128x1000 bytes
- Inode maintains a multi-level tree structure to find storage blocks for files
  - Great for little and large files
  - Asymmetric tree with fixed sized blocks
- Original **inode** format appeared in Berkeley Standard Distribution (BSD) Unix
  - Still used in modern Linux filesystems (e.g., ext4)

Inode Structure (1/5)



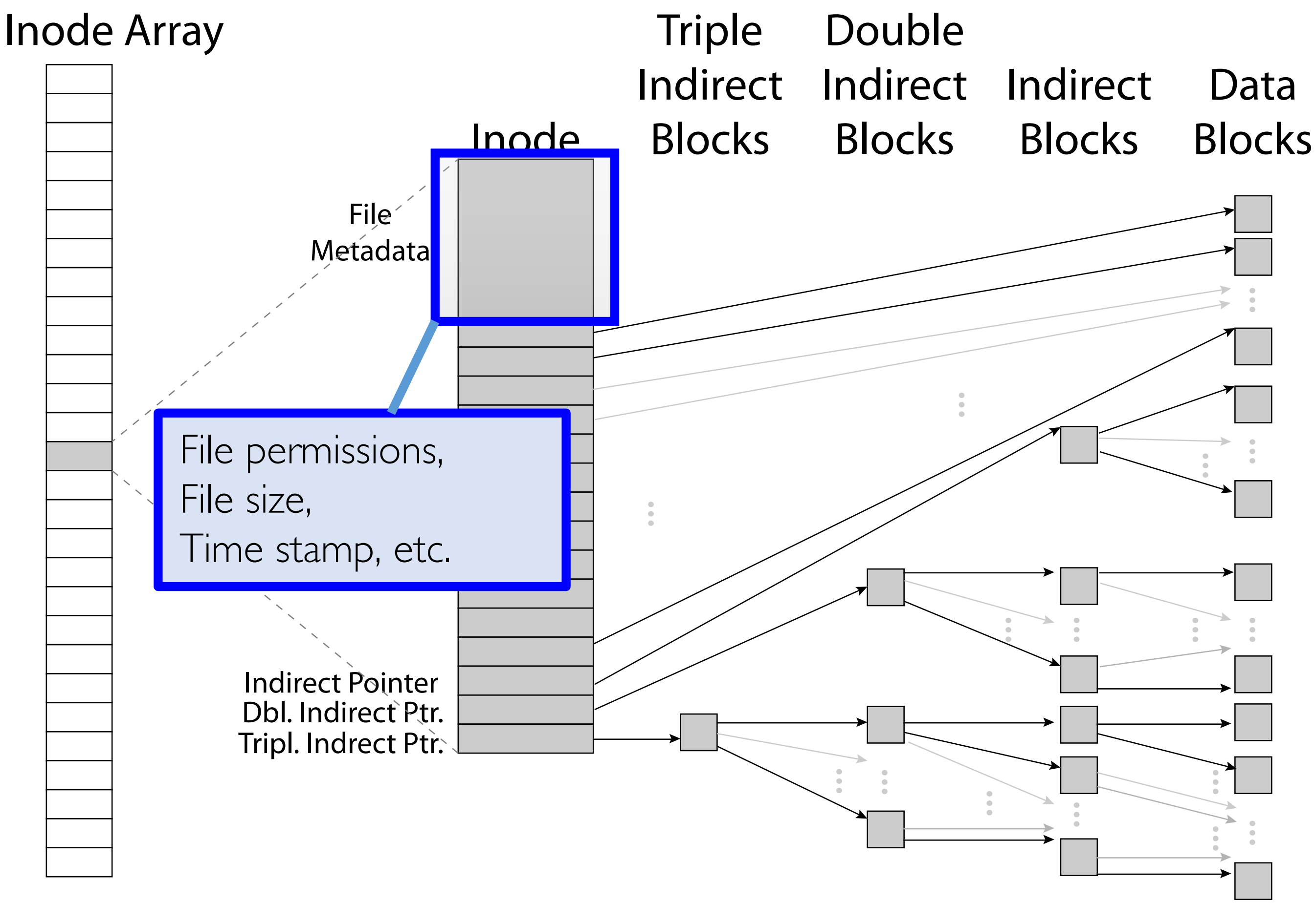
Inode Structure (2/5)



Array **on disk** at a well-known block number

Reserved when disk is formatted

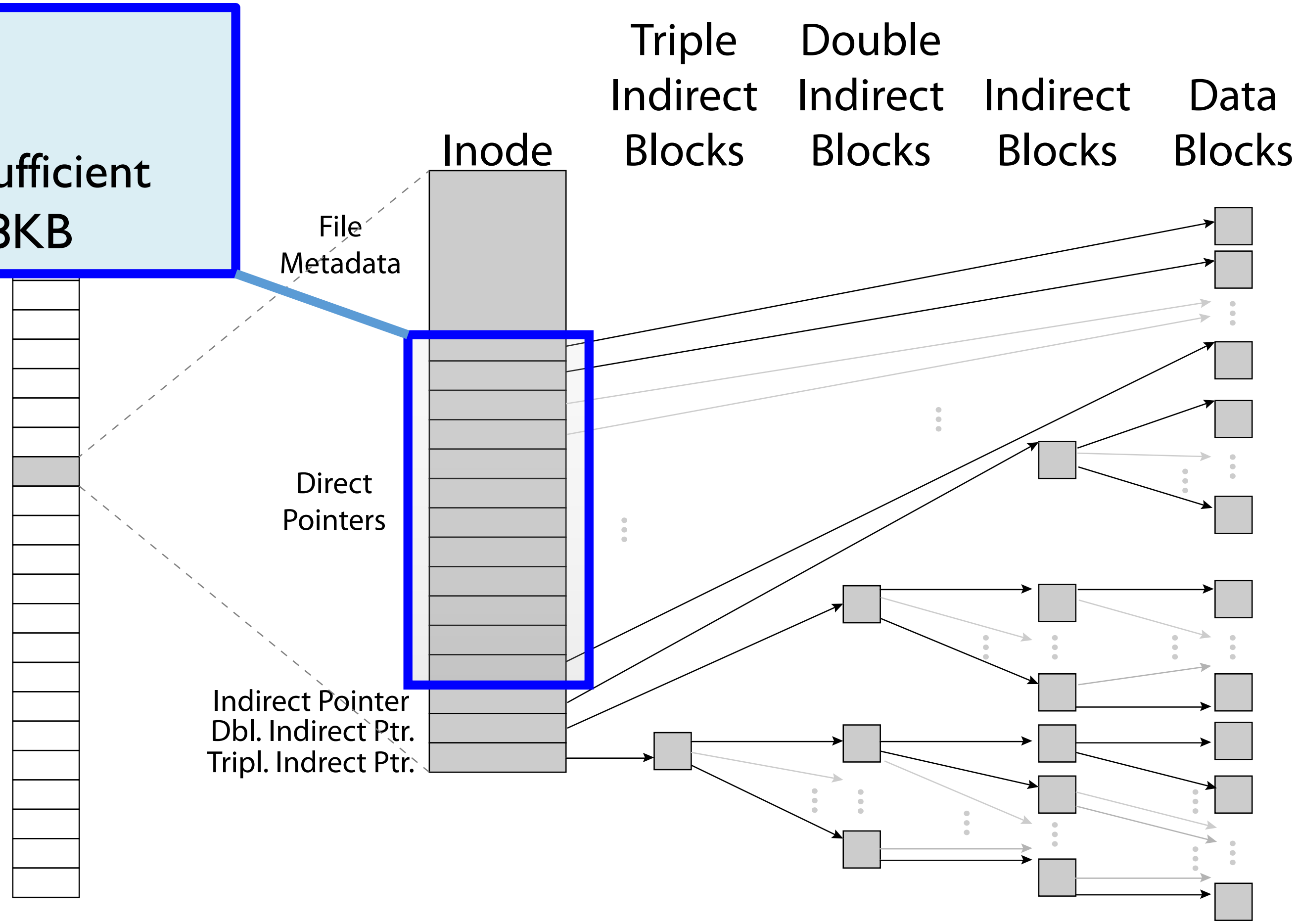
Inode Structure (3/5)



Inode Structure (4/5)

Direct pointers

4kB blocks ⇒ sufficient for files up to 48KB

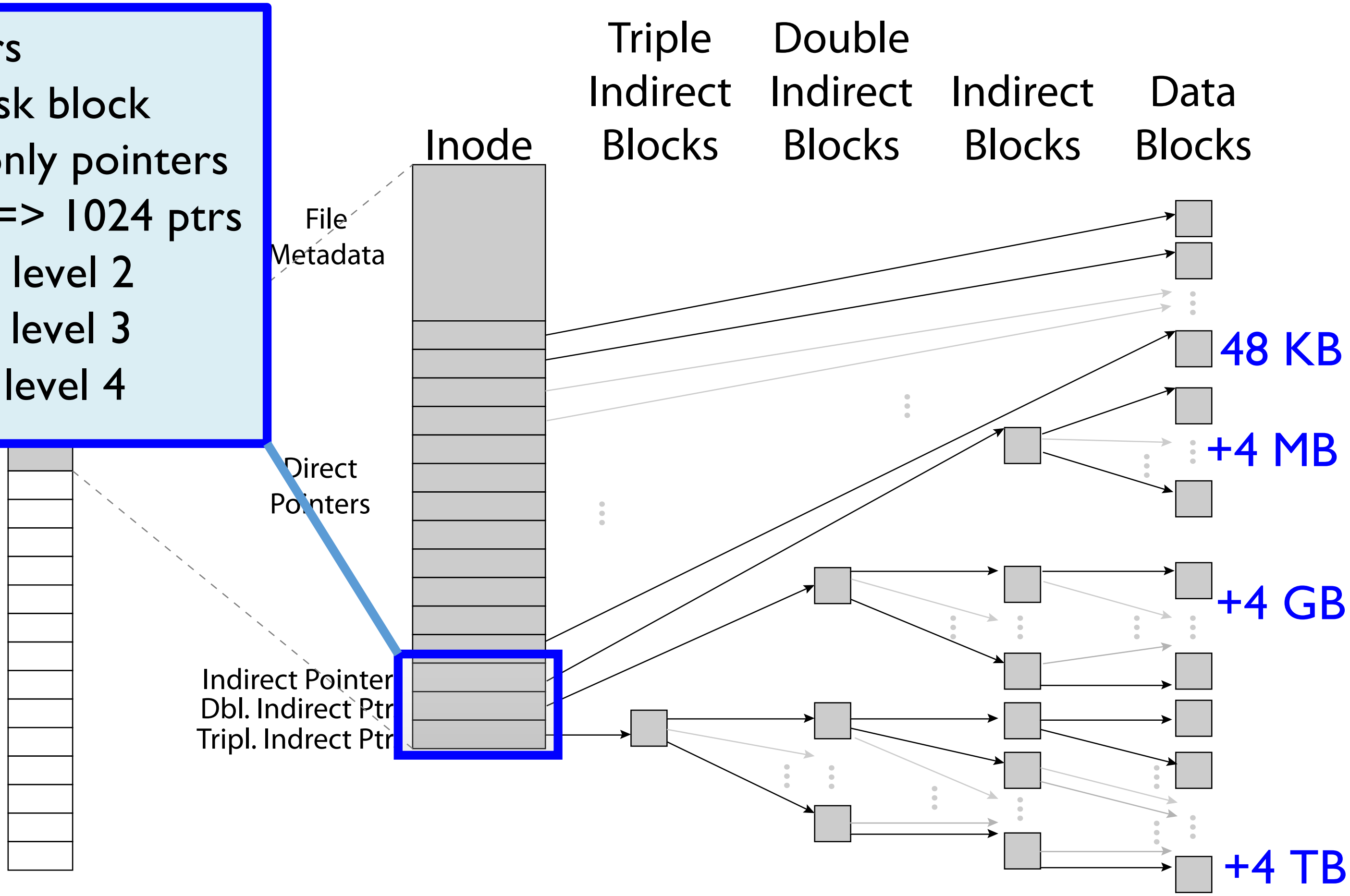


Small files: Level-1 has 12 pointers direct to data blocks (files up to 48KB)

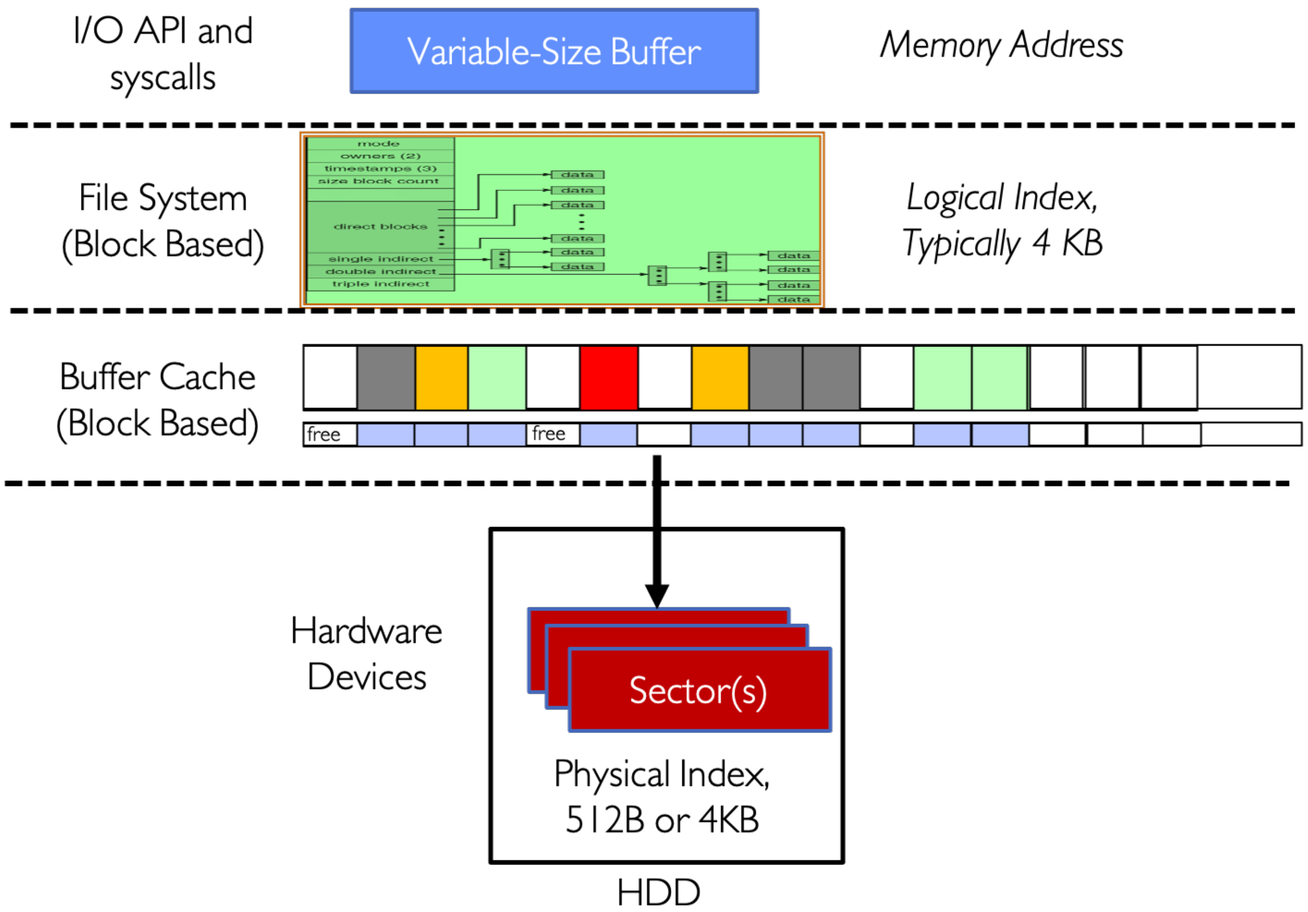
Inode Structure (5/5)

Indirect pointers
- point to a disk block containing only pointers
- 4 kB blocks => 1024 ptrs
  => 4 MB @ level 2
  => 4 GB @ level 3
  => 4 TB @ level 4

Large files: 2,3,4 level indirect pointers



Buffer Cache



Not block-sized or block-aligned access

Reuse of inodes, indirect blocks, data blocks

Speed up access to file system path and data

Apparent speed and flexibility is greater because of Buffer Cache

- Kernel *must* copy disk blocks to main memory (Buffer Cache) to access their contents and write them back when modified

Where are we as of now

- **CSE231 Post Conditions**
  1. Students are able to create a Unix shell with complete clarity about process creation and process execution
  2. Students are able to write multi- threaded applications with synchronization primitives and ability to analyze effects of concurrency on process execution and correctness
  3. Students are able to analyze the impact of OS concepts, e.g. virtual memory, concurrency, on program execution and ability to fine-tune the program to run efficiently on a given OS
  4. Students are able to demonstrate deeper understanding of the Unix-like OSes and kernel programming

Next Lecture

- End semester review