

**ECE 350**  
**Real-time**  
**Operating**  
**Systems**



# Lecture 11: File Systems

---

Prof. Seyed Majid Zahedi

<https://ece.uwaterloo.ca/~smzahedi>

# Outline

---

- File systems: files and directories
- File Allocation Table (FAT)
- Unix Fast File System (FSS)
- New Technology File System (NTFS)
- Buffer caches

# Building File Systems

---

- File system: OS **abstraction** that provides **persistent, named data**
  - Data is stored until it is **explicitly deleted**, even with crashes or power loss
  - Data can be accessed via **human-readable** identifier
- Goal: transform block interface of devices to interface with
  - **Naming**: interface to find files by name, not by blocks
  - **Date-storage management**: organize data-storage blocks into files
  - **Protection**: keep data secure and isolated
  - **Reliability/durability**: keep files durable despite crashes, failures, attacks, etc.

# Files

---

- Named set of blocks in file system
  - **Data**: information put by user or program
  - **Metadata**: information about file understood and managed by OS
    - Owner, size, modification time, permissions, ...
- User's view
  - Durable data structures
- System's view
  - System-call interface: array of untyped bytes (UNIX)
    - Doesn't matter what kind of data structures is store on disk!
    - Occasionally might need to parse file's data (e.g., ELF, **a.out**, bash scripts)
  - File system: collection of blocks
    - Block (logical transfer unit) size  $\geq$  sector (physical transfer unit) size

# File Access Patterns

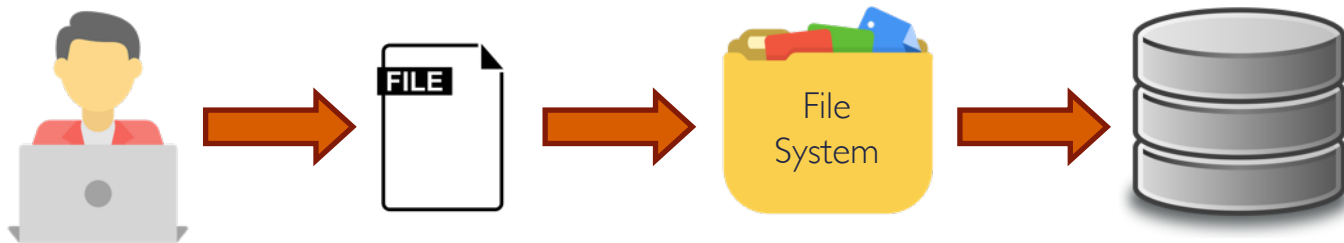
---

- **Sequential access:** bytes read in order
  - E.g., “give me the next X bytes, then give me next, etc.”
  - Most of file accesses are of this flavor
- **Random access:** read/write element out of middle of array
  - E.g., “give me bytes from i to j”
  - E.g., mem. page from swap file
  - Less frequent, but still important, want this to be fast
    - Don’t want to have to read all bytes to get to middle of file
- **Content-based access**
  - E.g., “find me 100 bytes starting with RTOS”
  - E.g., find employee X’s records and increase their salary by a factor of 2
  - Many systems don’t provide this; instead, build DBs to index content (still requires efficient random access)
    - E.g., Mac OSX Spotlight search



# From User to System View

---



- What happens if user asks for bytes 2 to 12?
  - File system fetches block corresponding to those bytes
  - System call returns just the correct portion of the block
- What about writing bytes 2 to 12?
  - Same as before, fetch block, modify relevant portions, write out block
- Everything inside file system is handled in terms of blocks
  - E.g., **getc()** and **putc()** buffer 4KiB only to access one byte at a time
  - Physical storage devices might work with sectors

# Characteristics of Files

## A Five-Year Study of File-System Metadata

NITIN AGRAWAL

University of Wisconsin, Madison

and

WILLIAM J. BOLOSKY, JOHN R. DOUCEUR, and JACOB R. LORCH

Microsoft Research

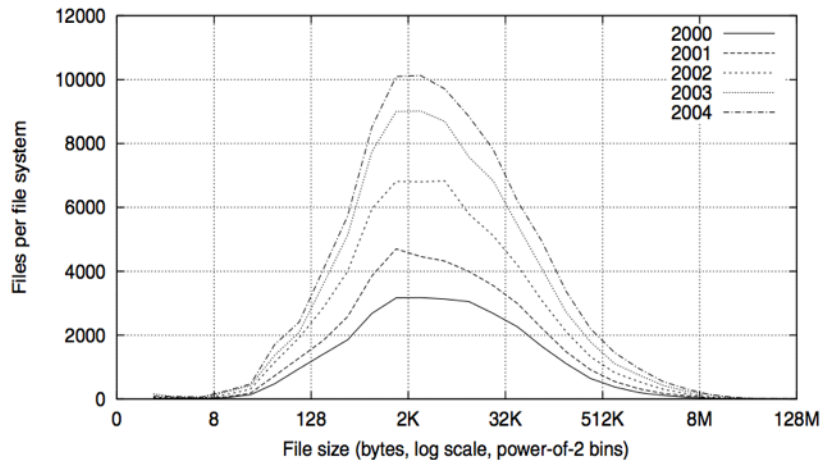


Fig. 2. Histograms of files by size.

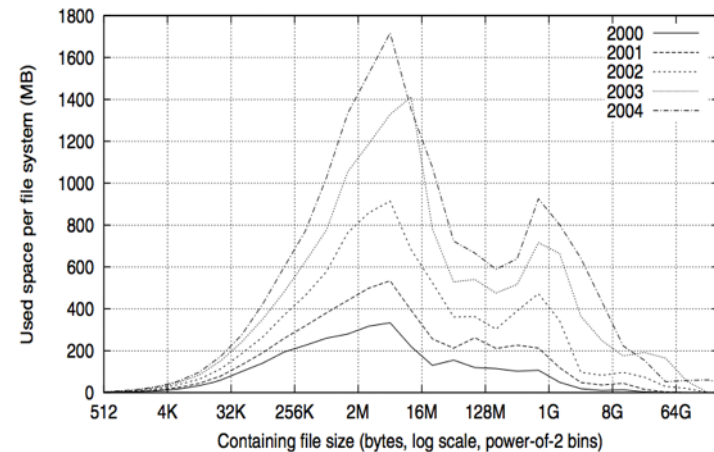


Fig. 4. Histograms of bytes by containing file size.



# Characteristics of Files (cont.)

---

- Most files are small, growing numbers of files over time

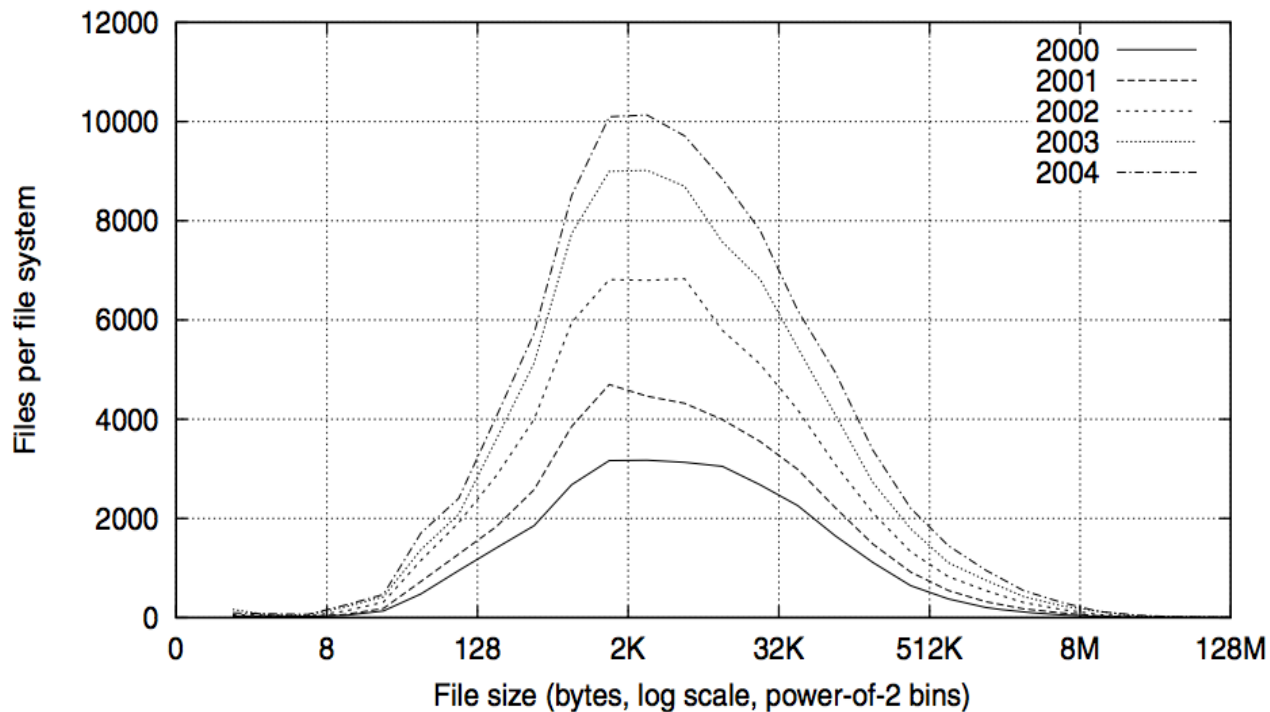


Fig. 2. Histograms of files by size.

# Characteristics of Files (cont.)

- Most of disk space is occupied by rare big ones

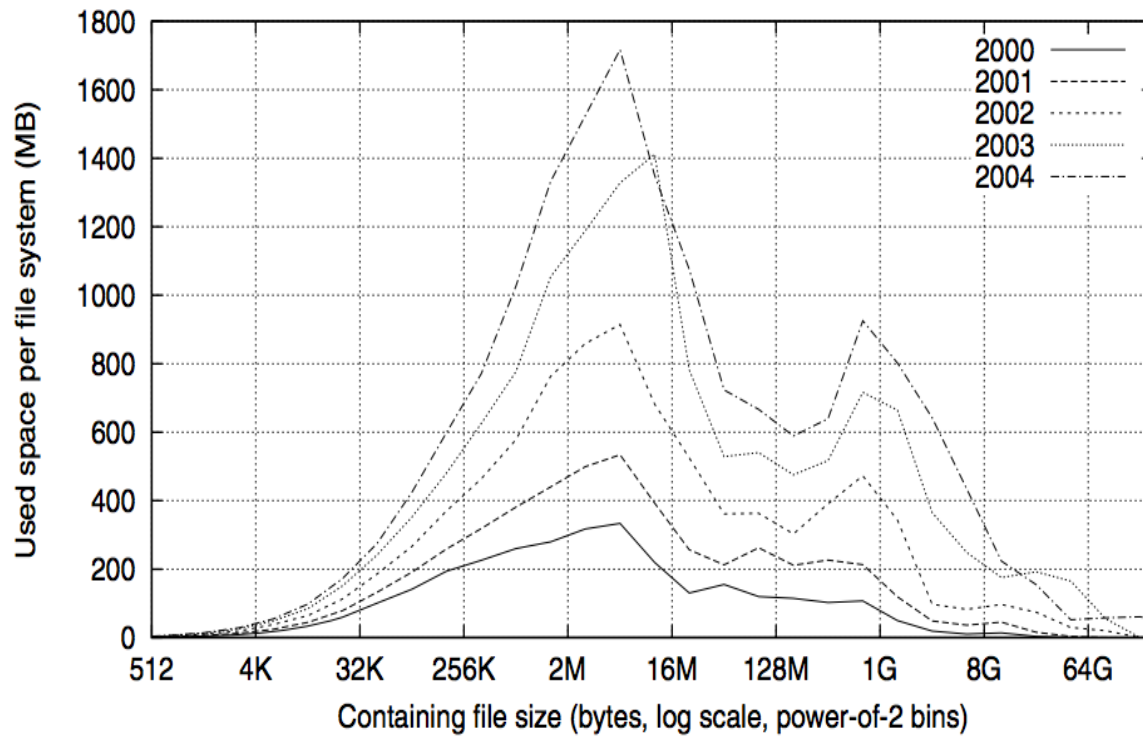


Fig. 4. Histograms of bytes by containing file size.

# Putting it Together: File Usage Patterns

---

- Observation 1: most files are small
- Observation 2: large files use up most of storage space (and devices BW)
  - May seem contradictory, but few big files  $\approx$  large # of small files
- Although we will use these observations, beware!
  - Good idea to look at usage patterns
    - Beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns.
    - Maybe UNIX has lots of small files because big files are handled inefficiently?

# Directory

---

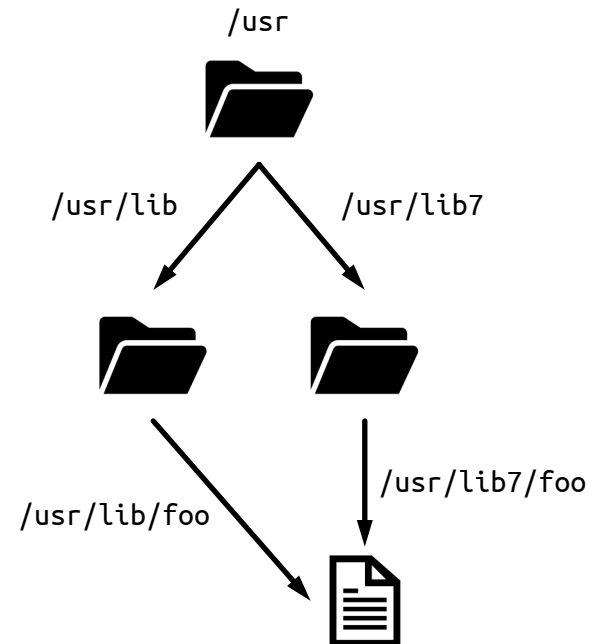


- List of human-readable names and mapping from each name to file or directory
  - Folder that contains documents (files) and other folders (directories)
  - Basically a hierarchical structure
- **Root directory**: thinking of directory as tree, root of tree is root directory
  - E.g., “/” in Unix
- **Path**: string that identifies file or directory
  - Absolute path: e.g., `/home/snz/foo.txt`
  - Relative path: e.g., from `/home` directory, `snz/foo.txt`

# Links

---

- **Hard link**: create another name (path) for given file
  - E.g., `ln /path/to/file /path/to/link` (in Unix)
  - If original file is deleted, hard link is still valid
- **Soft link**: map one name to another name
  - E.g., `ln -s /path/to/file /path/to/link` (in Unix)
  - If original file is deleted, soft link will point to nothing



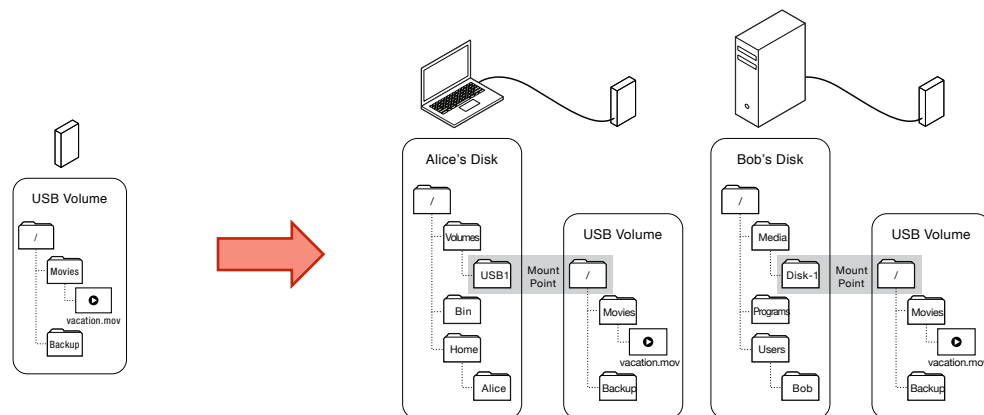
# Directory Access Pattern

---

- Directories are stored as files
  - Can be read from and written to
- System calls to access directories
  - `open/create` to traverse the structure
  - `mkdir/rmdir` to add/remove entries
  - `link/unlink` to link/unlink existing file to directory
- When can file be deleted?
  - Maintain ref-count of links to each file
  - Delete after last reference is gone
- `libc` support
  - `DIR * opendir (const char *dirname)`
  - `struct dirent * readdir (DIR *dirstream)`
  - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`

# Volume

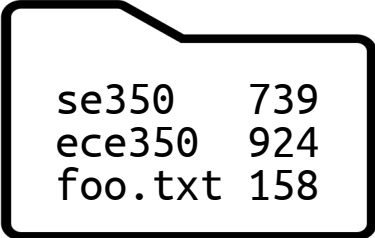
- Set of physical storage resources that form logical storage device
  - Each instance of file system manages files and directories for single volume
- Multiple volumes can be **mounted** on existing file system
  - Map path in file system to root directory of mounted volume's file system
  - Mounted file system controls mappings for all extensions of that path
  - Single machine can use multiple file systems stored on multiple volumes



# File System: Naming Data

---

- To access file, file system first translates file's name to its number
- Directories are files
- Each directory is linked-list of entries
- Each entry contains `<file_name, file_number>` mapping
- E.g., to find file number for file **foo.txt**, file system scans its directory **/home/snz**

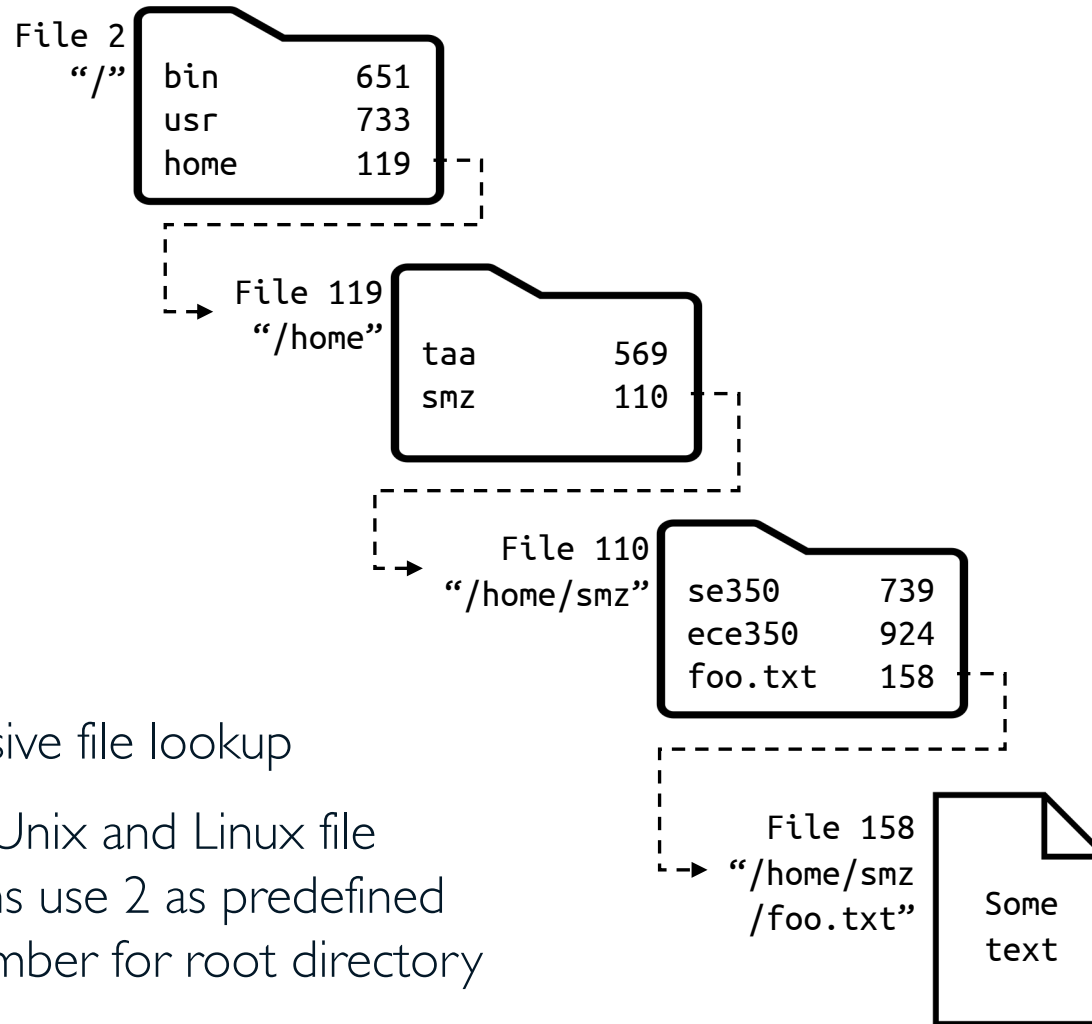


se350	739
ece350	924
foo.txt	158



# How Do We Find Directory?

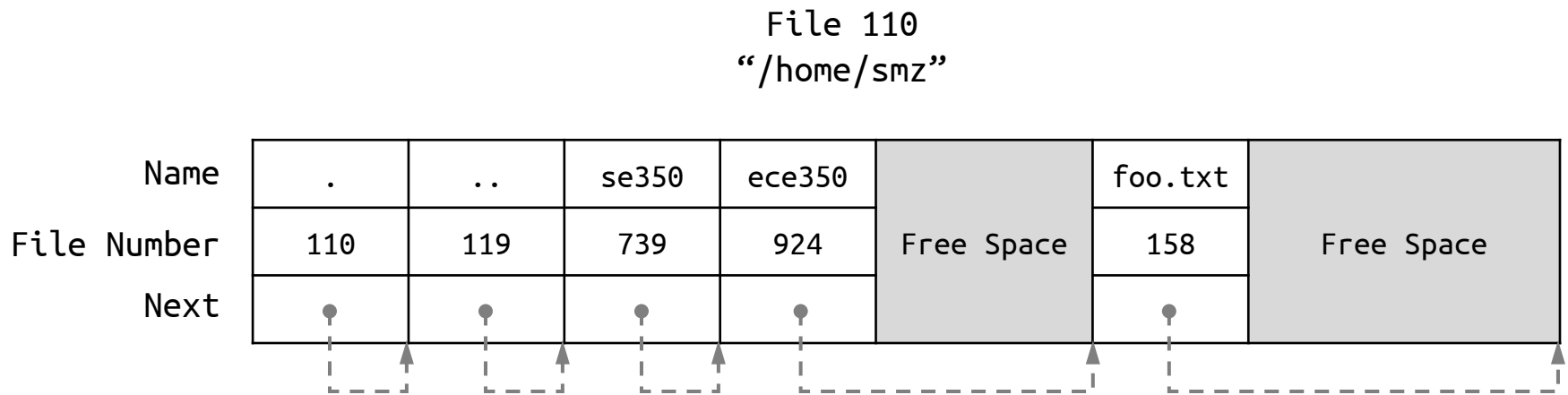
---



- Recursive file lookup
- Many Unix and Linux file systems use 2 as predefined file number for root directory

# Linked-list Directory Layout: Early Implementations

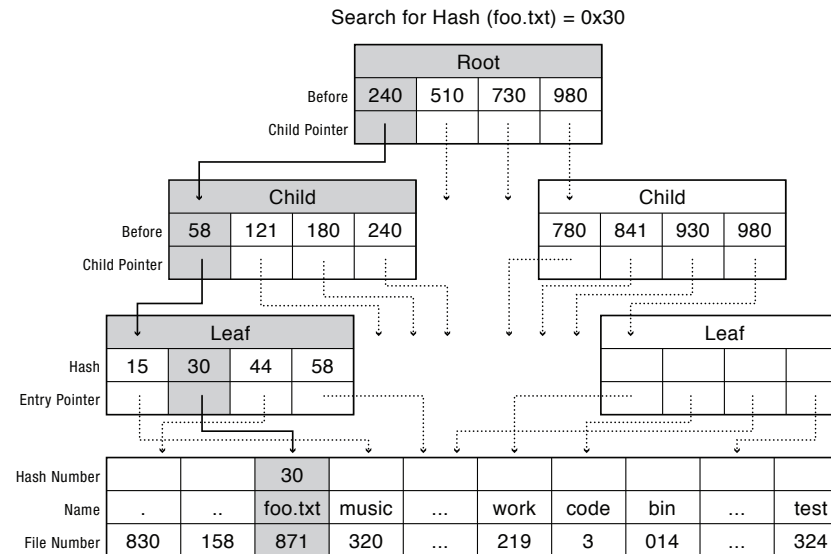
---



- Store linear lists of file name, file number pairs in directory files
  - E.g., original version of Linux Ext2
- + Simple
  - Work fine when the number of directory entries is small
- – Could become sluggish when there are thousands of files in directories

# Tree-based Directory Layout: Modern File Systems (Logical View)

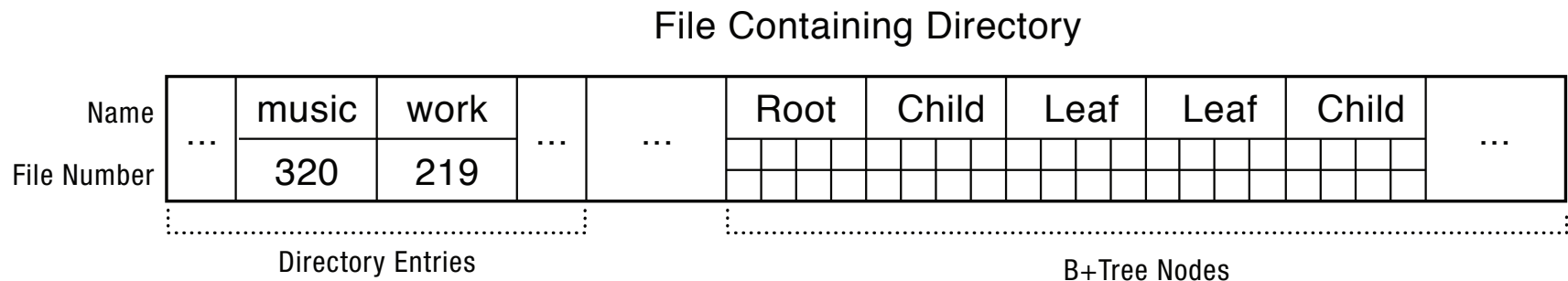
- Linux XFS, Microsoft NTFS, and Oracle ZFS organize directory's contents as tree
- Example: B+tree is indexed by hash of file's name



- Internal nodes contain array of sorted hash keys, each pointing to child node
- Child node's keys are smaller than parent's key entry but larger than parent's previous keys
- File system searches node for first entry with key larger than target
- File number at leaf nodes points to target directory entry

# Tree-based Directory Layout: Modern File Systems (Physical Storage)

---

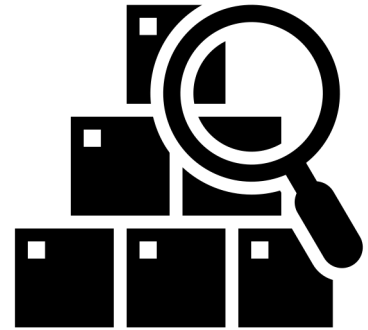


- Directory entries are usually stored in first part of directory file
- Tree's root is at well-known offset within file
- Fixed-size internal and leaf nodes are stored after root node
- Variable-size directory entries are stored at start of directory file
- Each tree node includes pointers to where in the file its children are stored

# File Systems: Finding Data

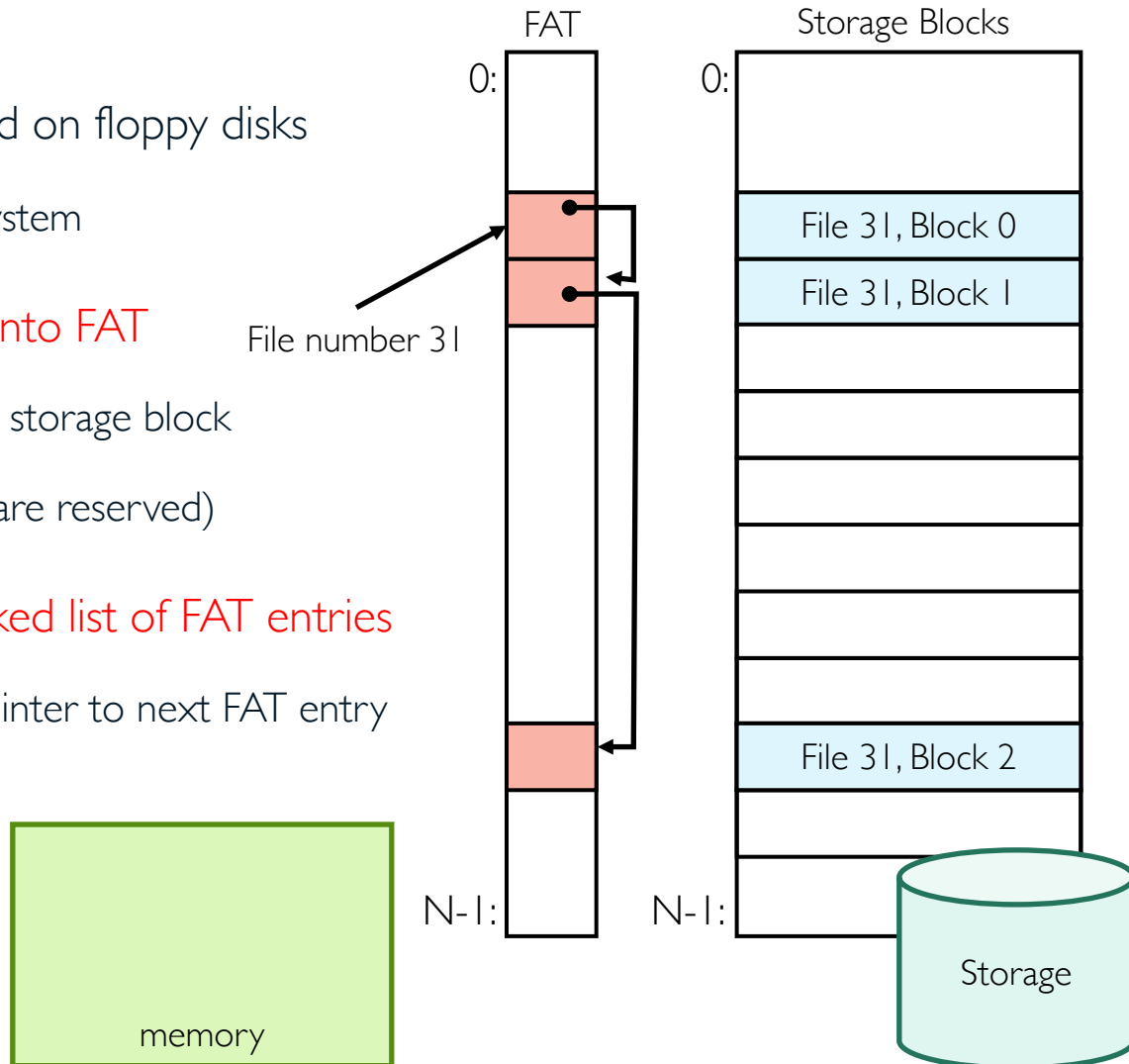
---

- Index structure
  - How do we locate blocks of a file?
- Index granularity
  - What block size do we use?
- Free space
  - How do we find unused blocks on storage device?
- Locality
  - How do we preserve spatial locality?
- Reliability
  - What if machine crashes in middle of file system operation?



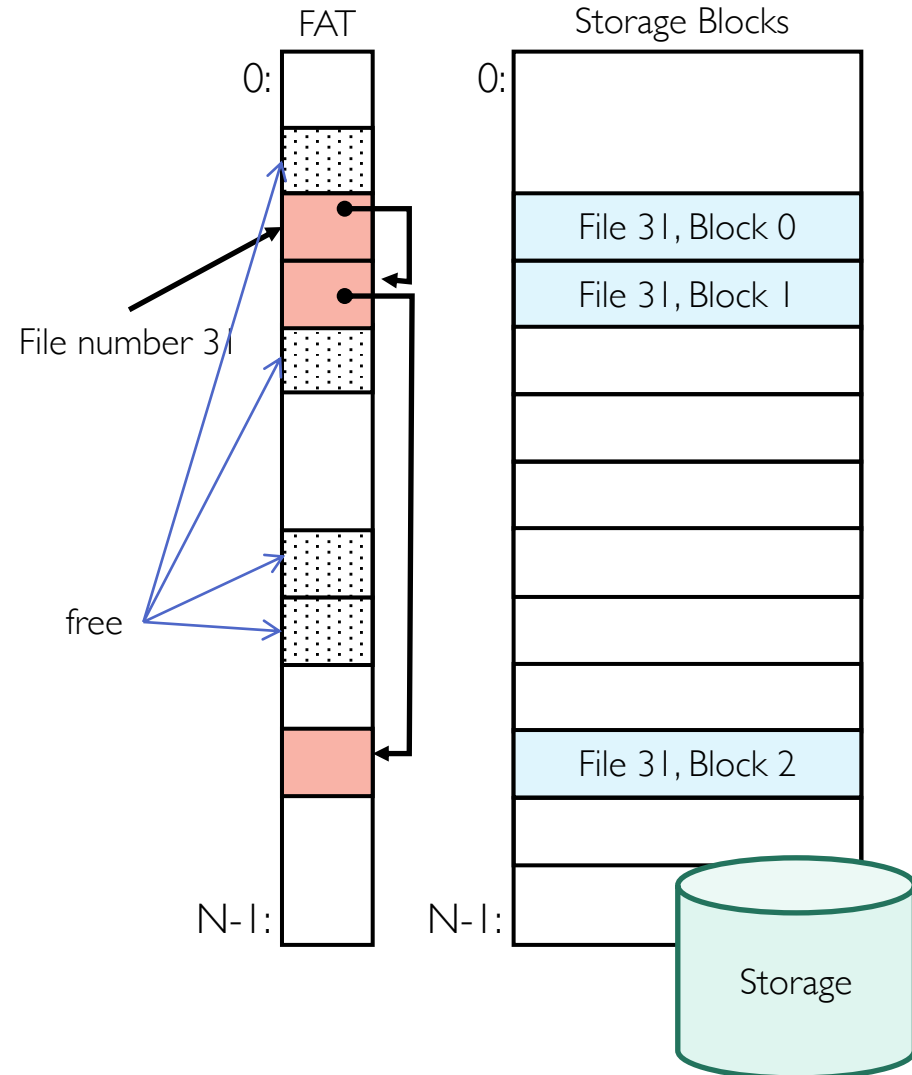
# FAT (File Allocation Table)

- Introduced in 1977 to be used on floppy disks
  - It is still a widely-used file system
- File number is used to **index into FAT**
  - There is one FAT entry per storage block
  - Each entry is 32 bit (4 bits are reserved)
- Each file is represented by **linked list of FAT entries**
  - Each FAT entry contains pointer to next FAT entry



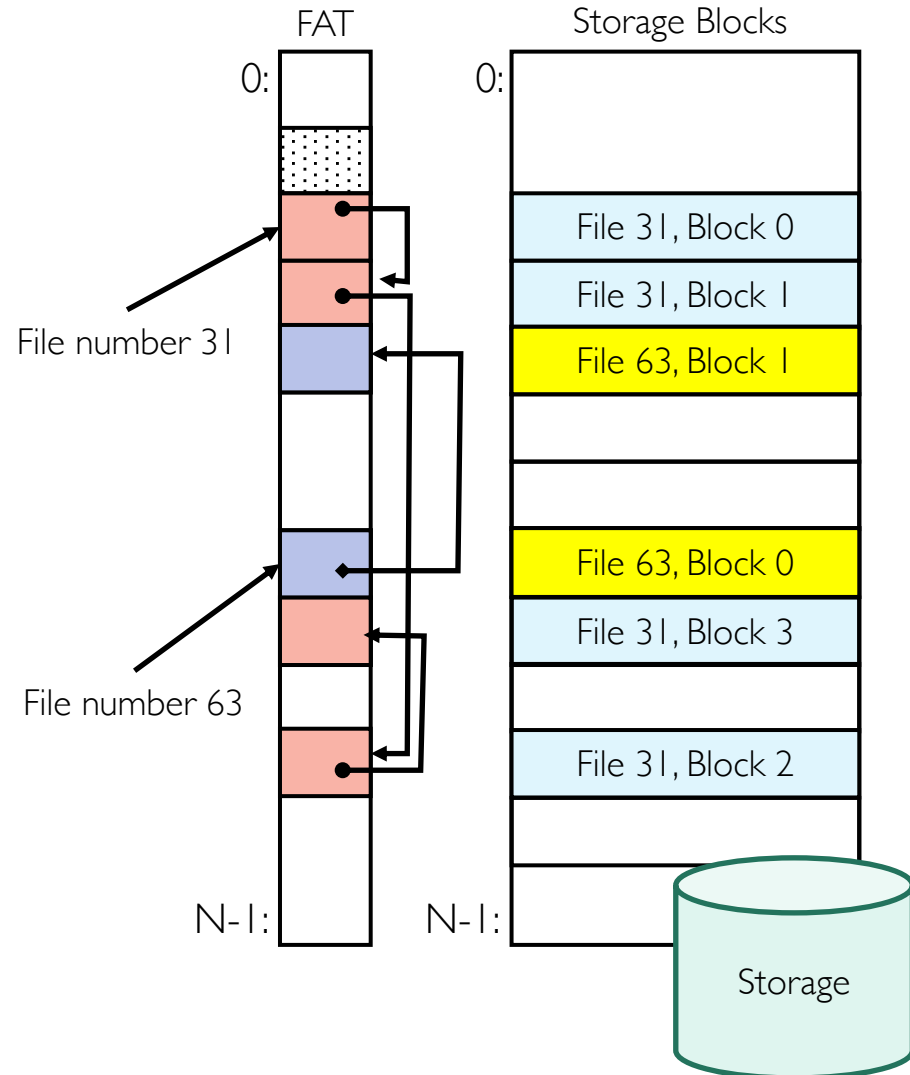
# FAT Properties

- File number is index of root of block list for each file
- Unused blocks are marked free (no ordering, must scan to find)



## FAT Properties (cont.)

- File number is index of root of block list for each file
- Unused blocks are marked free (no ordering, must scan to find)
- Grow file by allocating free blocks and linking them





# FAT Assessment

---

- Where is FAT stored?
  - On disk, on boot cache in memory, second (backup) copy on disk
- What happens when you format disk?
  - Zero all blocks, mark FAT entries “free”
- What happens when you quick format disk?
  - Mark all entries in FAT as free
- What is the most significant advantage of FAT? It is **simple**!
  - Easy to find free block
  - Easy to append to files
  - Easy to delete files

# Disadvantages of FAT

---

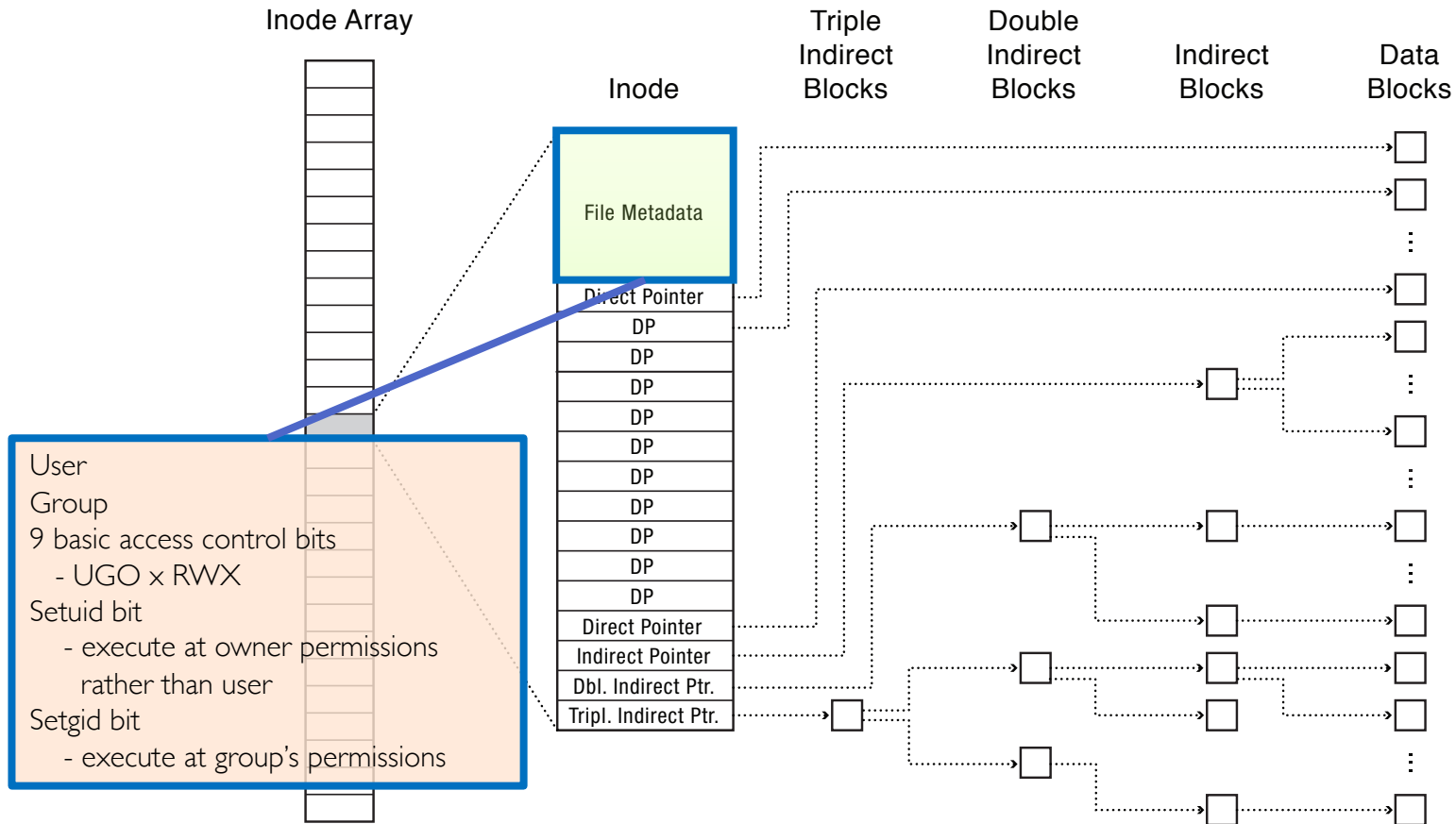
- Random access is very slow (traversing linked-list)
  - Finding random block in large file takes long time
- Poor locality
  - File blocks for given file may be scattered
  - Files in the same directory may be scattered
  - *Defragmentation* could be used to improve performance
- No support for hard links
  - File metadata is stored with directory entries with the file's name
  - Each file must be accessed via exactly one directory entry
- Limited volume and file sizes
  - FAT volume can have at most  $2^{28}$  entries  $\Rightarrow$  with 4KiB blocks, max volume size is 1 TiB
  - File size is encoded into 32 bit  $\Rightarrow$  max file size is  $2^{32} - 1$ , which is just under 4GiB
- Security and reliability vulnerabilities
  - FAT doesn't support access rights & transactional update methods (more on this later)

# Berkeley Unix Fast File System (FFS)

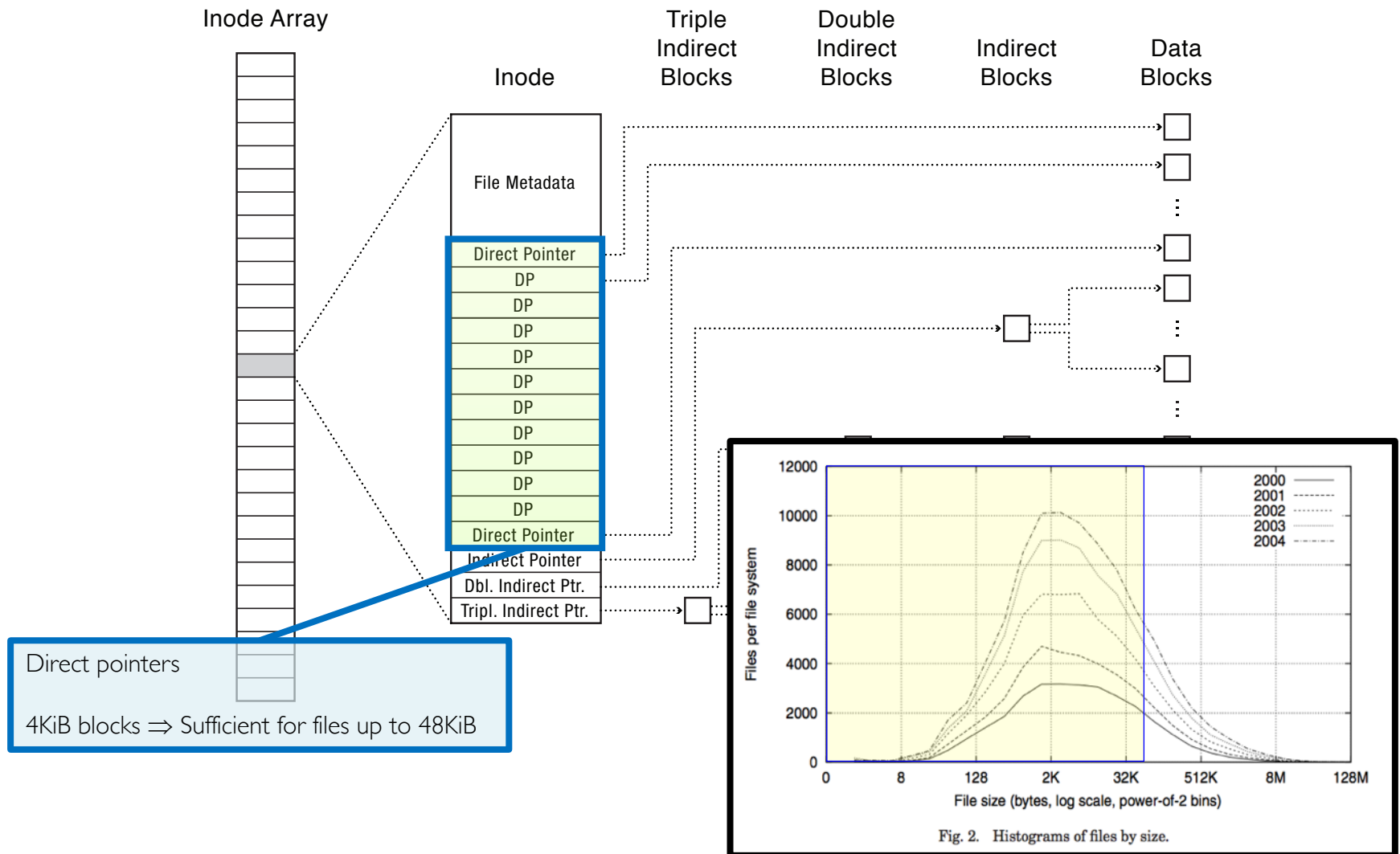
---

- Originally, *inode* format appeared in Berkeley Standard Distribution (BSD) 4.1
  - Similar structure for Linux Ext2/3
- File number (*inumber*) is used to index into inode array
  - Each inode tracks file blocks for single file
- Each inode has multi-level-index structure
  - Great for little and large files
  - Asymmetric tree with fixed sized blocks
  - Stores metadata associated with each file
- Free space is managed using bitmap with one bit per storage block
  - Placed in fixed position when file system is formatted

# Inode Structure



# Inode Structure (cont.)



# Inode Structure (cont.)

## Indirect pointers

- point to a disk block containing only pointers
- 4KiB blocks  $\Rightarrow$  1024 pointers
  - $\Rightarrow$  4MiB @ level 2
  - $\Rightarrow$  4GiB @ level 3
  - $\Rightarrow$  4TiB @ level 4

## Inode Array

## Inode

### File Metadata

### Direct Pointer

DP

DP

DP

DP

DP

DP

DP

DP

DP

DP

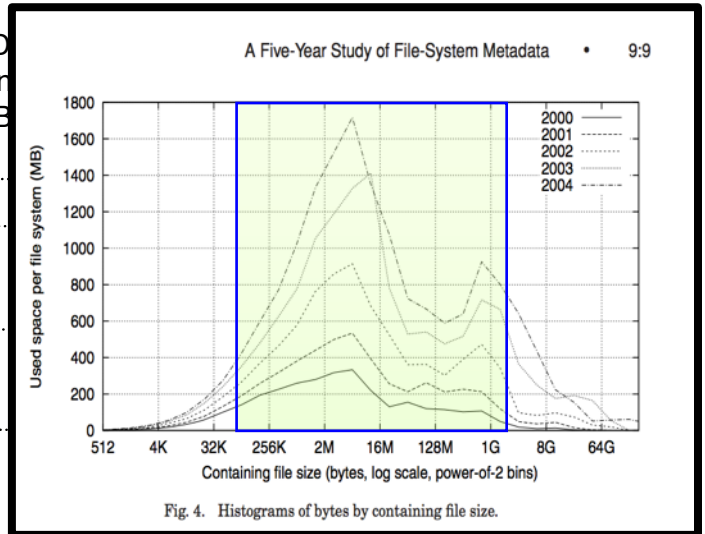
### Direct Pointer

### Indirect Pointer

### Dbl. Indirect Ptr.

### Tripl. Indirect Ptr.

## Triple Indirect Blocks



# Summary: Inode Details

---

- Metadata
  - File owner, access permissions, access times, ...
- Direct-block pointers
  - With 4KiB blocks  $\Rightarrow$  max size of 48KiB
- Indirect-block pointer
  - Pointer to on-disk indirect block which contains array of direct-block pointers
  - With 4KiB indirect block and 4B block pointer  $\Rightarrow 2^{10}$  direct-block pointers
  - With 4KiB blocks  $\Rightarrow$  max size of 4MiB (+ 48KiB)
- Doubly-indirect-block pointer
  - $2^{10}$  indirect-block pointers  $\Rightarrow 2^{20}$  direct-block pointers
  - With 4KiB blocks  $\Rightarrow$  max size of 4GiB (+ 4MiB + 48KiB)
- Triply-indirect-block pointer
  - $2^{10}$  doubly indirect blocks  $\Rightarrow 2^{30}$  direct-block pointers
  - With 4KiB blocks  $\Rightarrow$  max size of 4TiB (+ 4GiB + 4MiB + 48KiB)

# FFS Asymmetric Tree

---



- Small files: shallow tree
  - Efficient storage for small files
- Large files: deep tree
  - Efficient lookup for random access in large files
- Sparse files: only fill pointers if needed



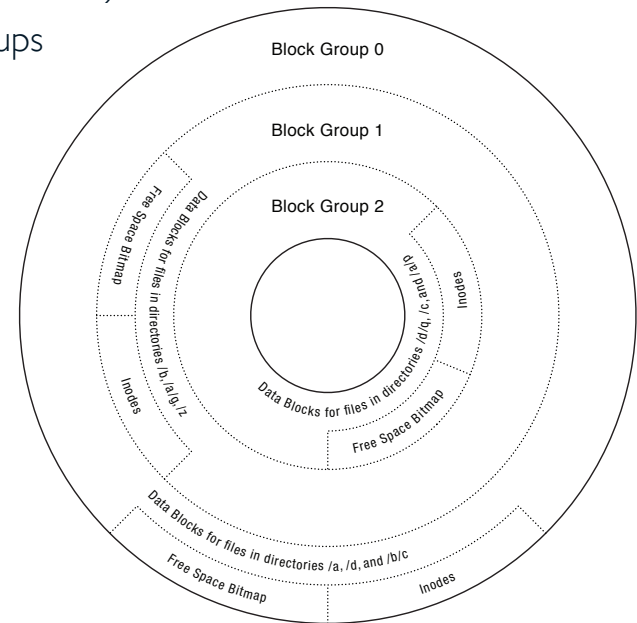
# Multi-level-indexed Files Example

---

- How many accesses for block #5 (assuming file header accessed on open)?
  - One: one for data block
- How many accesses for block #23?
  - Two: one for indirect-block pointer, one for data block
- How many accesses for block #340?
  - Three: double-indirect block, direct block, and data block
- ...

# FFS on Magnetic Storage

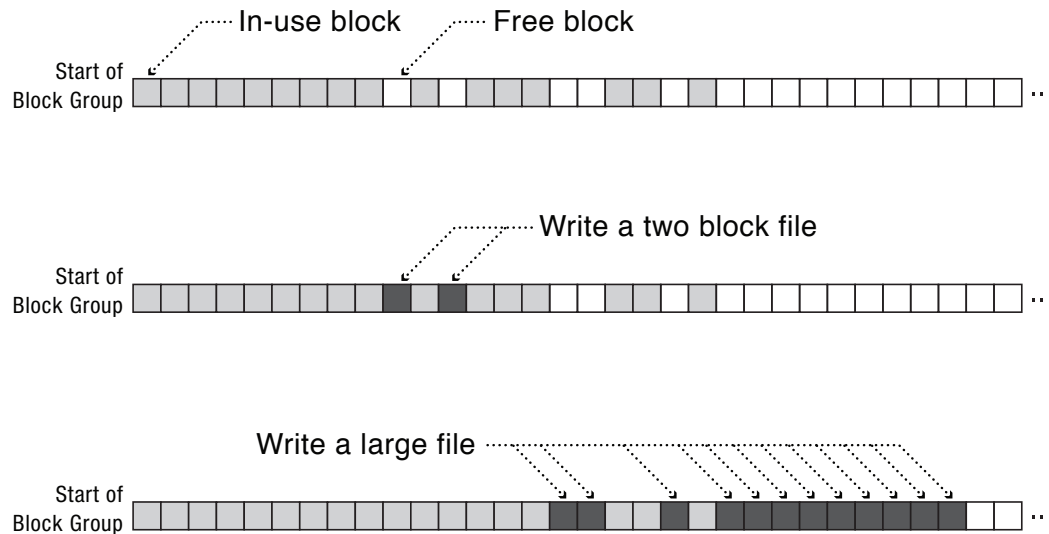
- In early FFS, inode array was stored at fixed location in outermost cylinders
  - At formatting time, fixed number of inodes are created
  - Each is given unique number (inumber)
  - To read small files, disk head had to seek to get header; seek back to data
- In later versions, disk is divided into groups (neighboring tracks)
  - Data blocks, inodes, and bitmaps are scattered across groups
  - Avoid seeks between user data and file system structure
  - Try to put directory and its files in the same block group
- FFS reserves some fraction space (e.g., 10%)
  - When most block groups become almost full, new writes will be scattered around disk
  - To avoid this, if actual free space falls below reserved fraction, FFS prevents new writes (i.e., they fail)
  - Only super user's processes can allocate new blocks, allowing admin to log in and clean things up



# Data Block Placement: First-free Heuristic

---

- To expand file, use first free blocks in group using the group's bitmap



- This might scatter sequential writes into holes near start of group
- But it leads to contiguous free space at the end of group

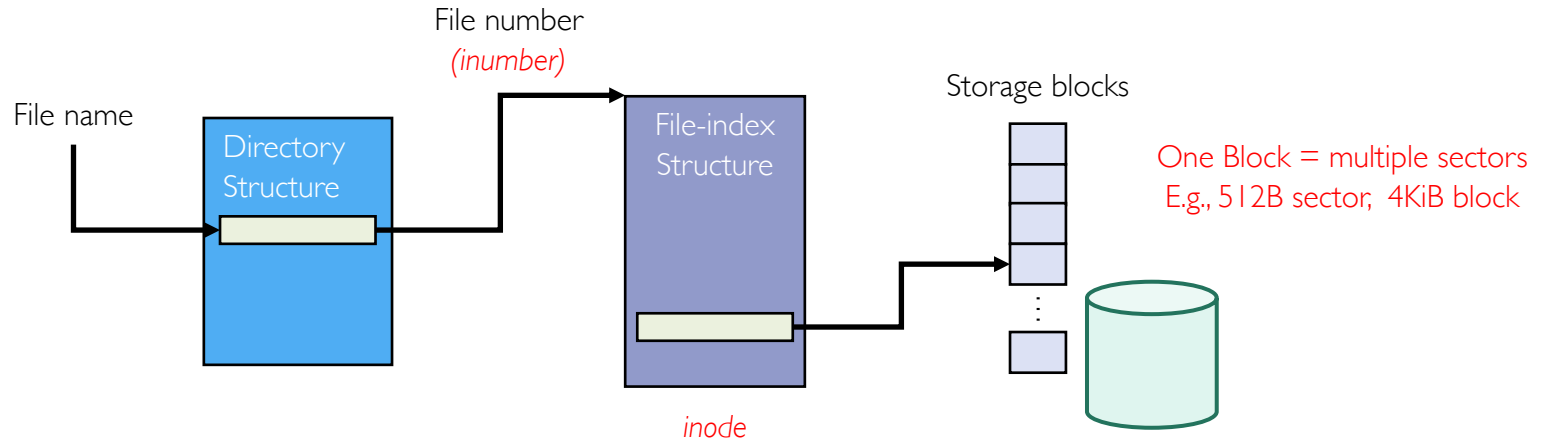
# FSS Analysis

---

- + Efficient storage for both small and large files
  - + Locality for both small and large files
  - + Locality for metadata and data
  - + No defragmentation necessary!
- 
- – Inefficient for tiny files (a 1 B file requires both inode and data block)
  - – Inefficient encoding when file is mostly contiguous on disk
  - – Need to reserve 10-20% of free space to prevent fragmentation

# Putting it Together: File-system API (Unix)

---



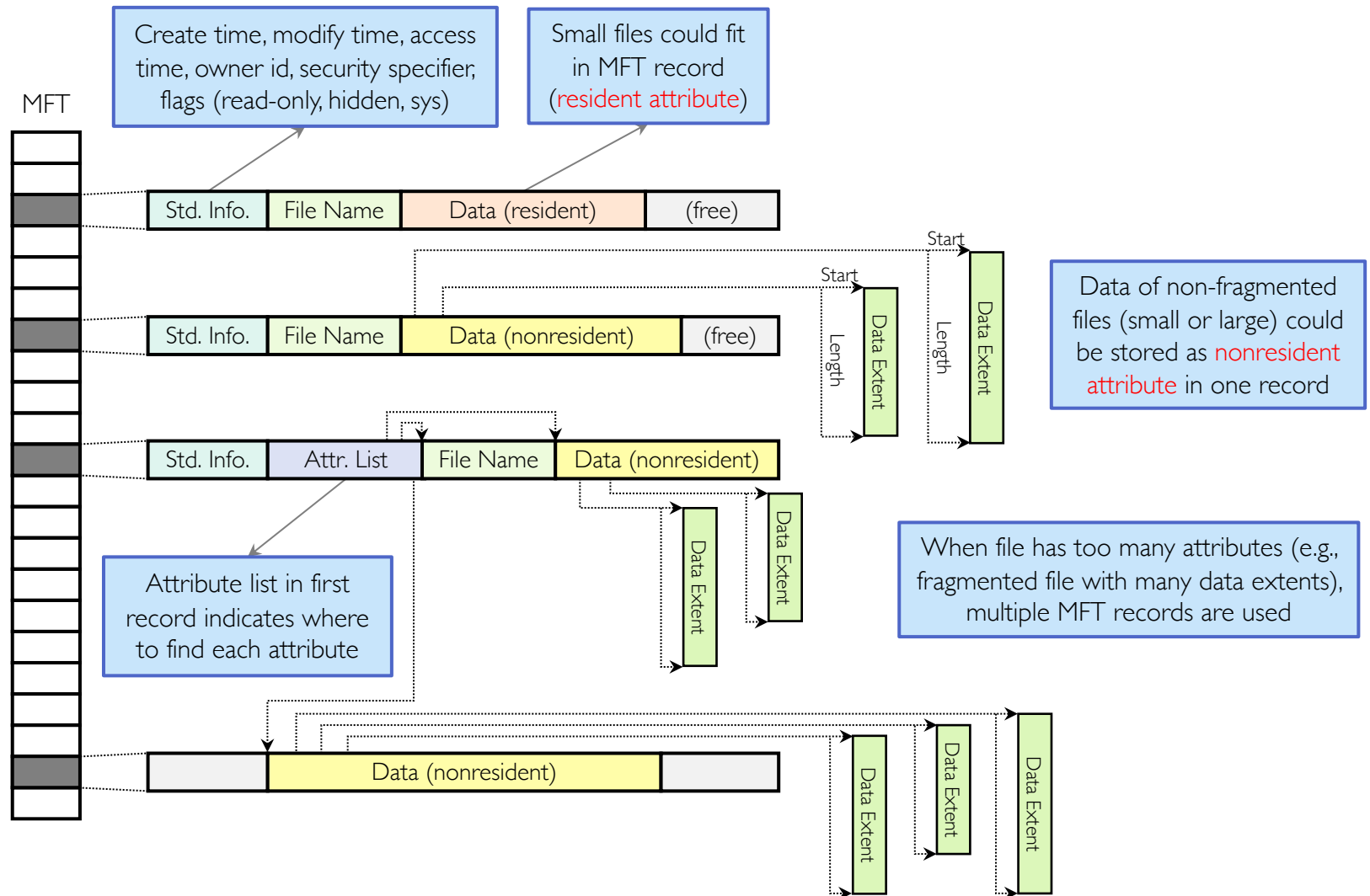
- **open()** performs **name resolution**
  - Resolves file name, finds inode
  - Creates **file descriptor** in PCB
  - Returns **file handle** (another integer) to user process
- **read()**, **write()**, **seek()**, and **sync()** operate on file handle to locate inode
  - Perform appropriate read, write, etc. operations

# New Technology File System (NTFS)

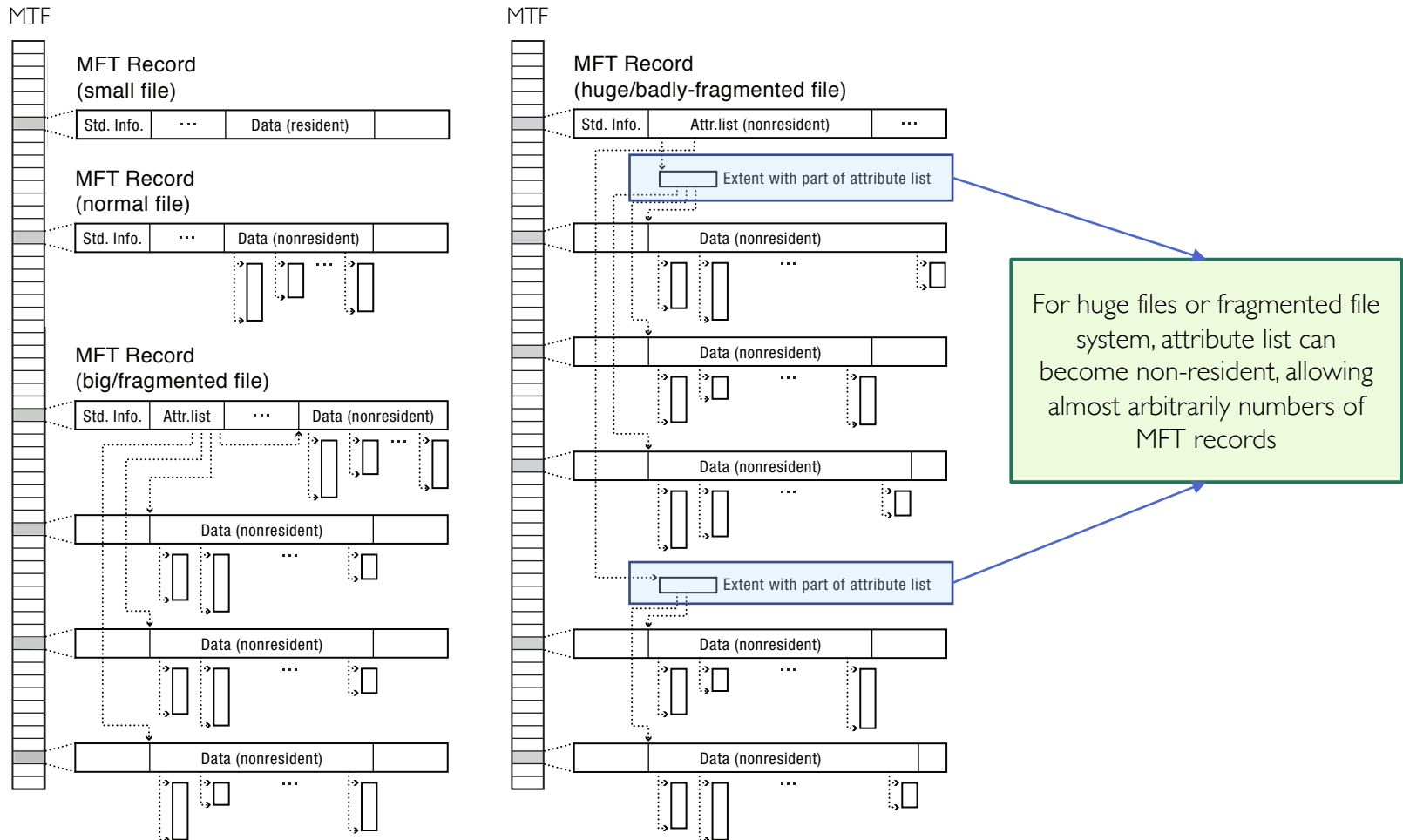
---

- NTFS is proprietary file system developed by Microsoft
- NTFS provides improvements over FAT
  - File metadata, security, and reliability (journaling)
- **Extents** represent regions of file stored in contiguous regions of storage device
  - Similar approach in Linux (Ext4)
- Files are represented by **trees** containing pointers to file's extents
  - File with small # of extents is stored in shallow tree even if file is large
  - Deeper trees are only needed for fragmented files
- Roots of trees are stored in **master file table (MFT)**
  - Like inode array
- Directories are organized in **B+tree structure**

# Master File Table (MFT)



# MTF Record Stages





# Some NTFS Details

---

- Almost all of metadata is stored as files with well-known file numbers
  - E.g., file number 5 for root directory and 6 for free space bitmap
- Security and access control information is stored in **\$Secure** file (file number 9)
  - Each unique access control has fixed-length unique key
  - Each individual file stores appropriate fixed-length key in its MFT record
- Master file table is stored in **\$MFT** file (file number 0)
- NTFS uses variation of **best-fit** allocation policy
- NTFS allows application to specify expected size of file at creation time
- To prevent **\$MFT** to become fragmented, NTFS reserves fraction of volume
- NTFS defragmentation rewrites fragmented files to contiguous regions of storage

# File Systems: Design Options

---

	FAT	FFS	NTFS
Index structure	Linked list	Tree (fixed, asymmetric)	Tree (dynamic)
Granularity	Block	Block	Extent
Free space tracking	FAT array scan	Bitmap (fixed location)	Bitmap (file)
Locality	Defragmentation	Block groups + reserve space	Extents + best fit + defragmentation

# Buffer Cache

---

- Kernel must copy disk blocks to main memory to access their contents and write them back if modified
  - Could be data blocks, inodes, directory contents, etc.
  - Possibly dirty (modified and not written back)
- Key idea: exploit locality by caching storage data in memory
  - Name translations: mapping from paths → inodes
  - Disk blocks: mapping from block address → disk content
- Buffer cache: memory used to cache kernel resources, including disk blocks and name translations
  - Implemented entirely in OS software (unlike memory caches and TLB)

# File-system Caching

---

- Replacement policy? **LRU!**
  - Because we can afford overhead of timestamps for each disk block
  - + Works very well for name translation
  - + Works well if memory is big enough to accommodate working set of file blocks
  - – Some apps scan through file system, flushing cache with data used only once
    - E.g., `find . -exec grep foo {} \;`
- Other replacement policies?
  - Some systems allow applications to request other policies
  - E.g., **use once**: file system can discard blocks as soon as they are used
- How much memory should OS allocate to buffer cache?
  - Too much memory  $\Rightarrow$  OS will have less memory to allocate to applications
  - Too little memory  $\Rightarrow$  applications may run slowly (inefficient disk caching)
  - Solution: dynamically adjust allocation to balance paging and file access time

# File-system Caching (cont.)

---

- **Read-ahead prefetching**: fetch sequential blocks early
  - Fast to access; file system tries to obtain sequential layout
  - Applications tend to do sequential reads and writes
- How much to prefetch?
  - Too many  $\Rightarrow$  delaying requests by other applications
  - Too few  $\Rightarrow$  missed opportunities to improve performance

# Delayed Writes

---

- Every `write()` copies data from user space to kernel buffer
  - Other apps read data from cache instead of disk
  - Cache is transparent to user programs
- Buffer cache is write-back cache
  - Writes are not immediately sent to disk
  - Buffer cache is flushed to disk periodically
    - E.g., in Linux, kernel threads flush buffer cache every 30 sec. in default setup
  - Some files never actually make it all the way to disk
    - There are many short-lived files
- But **what if system crashes** before buffer cache is flushed to disk?
  - And what if this was for a directory file?
    - Lose pointer to inode
- **File systems need recovery mechanisms**

# Important “abilities”

---

- **Reliability**: ability of system or component to perform its required functions under stated conditions for specified time
  - System is not only “up”, but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Data must survive system crashes, disk crashes, other problems
- **Availability**: probability that system can accept and process requests
  - Can build highly-available systems, despite unreliable components
    - Involves independence of failures and redundancy
  - Often as “nines” of probability. 99.9% is “3-nines of availability”
- **Durability**: ability of system to recover data despite faults
  - This idea is fault-tolerance applied to data
  - Doesn’t necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone

# Summary

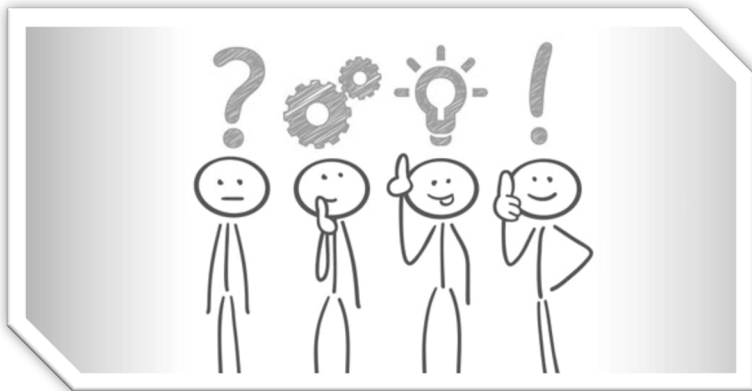
---

- File system
  - Abstraction that provides persistent, named data
  - Optimize for access and usage patterns
- File Allocation Table (FAT)
  - Linked-list approach, widely-used file system (e.g., cameras, USB drives, SD cards)
  - Simple to implement, but poor performance and no security
- Unix Fast File System (FFS)
  - Tracking files' blocks using inodes, indirect blocks, double indirect blocks, etc.
- New Technology File System (NTFS)
  - Variable extents, dynamic tree, tiny files data is in header
- Buffer caches
  - Exploiting locality by caching storage data in memory



# Questions?

---



# Acknowledgment

---

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny