

CS162
Operating Systems and
Systems Programming
Lecture 22

Filesystems 2: Filesystem Design (Con't),
Filesystem Case Studies

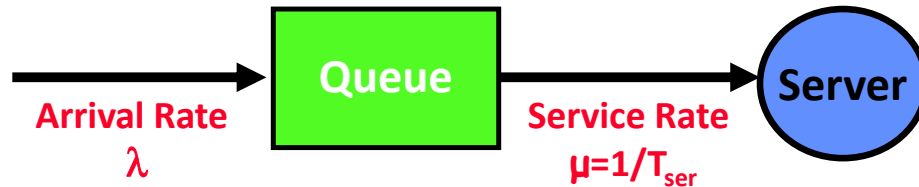
April 16th, 2026

Prof. John Kubiatowicz

<http://cs162.eecs.Berkeley.edu>

Recall: A Few Queuing Theory Results

- Assumptions:
 - System in equilibrium; No limit to the queue
 - Time between successive arrivals is random and memoryless

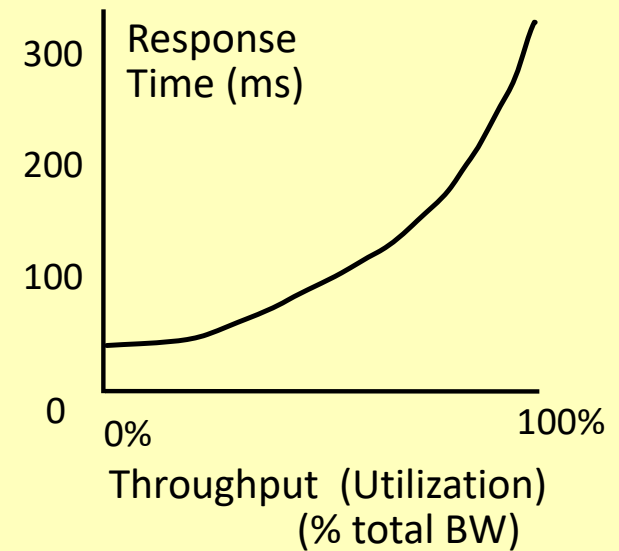


- Parameters that describe our system:
 - λ : mean number of arriving customers/second
 - T_{ser} : mean time to service a customer ("m1")
 - C : squared coefficient of variance = σ^2/m^2
 - μ : service rate = $1/T_{ser}$
 - u : server utilization ($0 \leq u \leq 1$): $u = \lambda/\mu = \lambda \times T_{ser}$

- Parameters we wish to compute:
 - T_q : Time spent in queue
 - L_q : Length of queue = $\lambda \times T_q$ (by Little's law)

- Results:
 - Memoryless service distribution ($C = 1$): (an "M/M/1 queue"):
 - $T_q = T_{ser} \times u/(1-u)$
 - General service distribution (no restrictions), 1 server (an "M/G/1 queue"):
 - $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1-u)$

Why does response/queueing delay grow unboundedly even though the utilization is < 1 ?



Recall: I/O and Storage Layers

Application / Service

High Level I/O

Streams

Low Level I/O

File Descriptors

Syscall

*open(), read(), write(), close(), ...
Open File Descriptions*

What we covered in early Lectures

File System

Files/Directories/Indexes

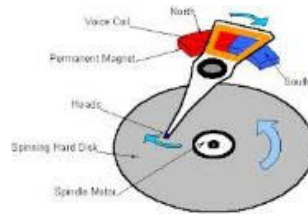
What we will cover next...

I/O Driver

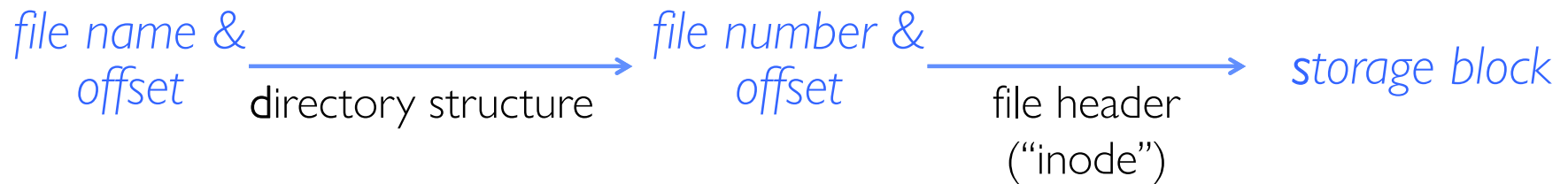
Commands and Data Transfers

Disks, Flash, Controllers, DMA

What we just covered...



Components of a File System

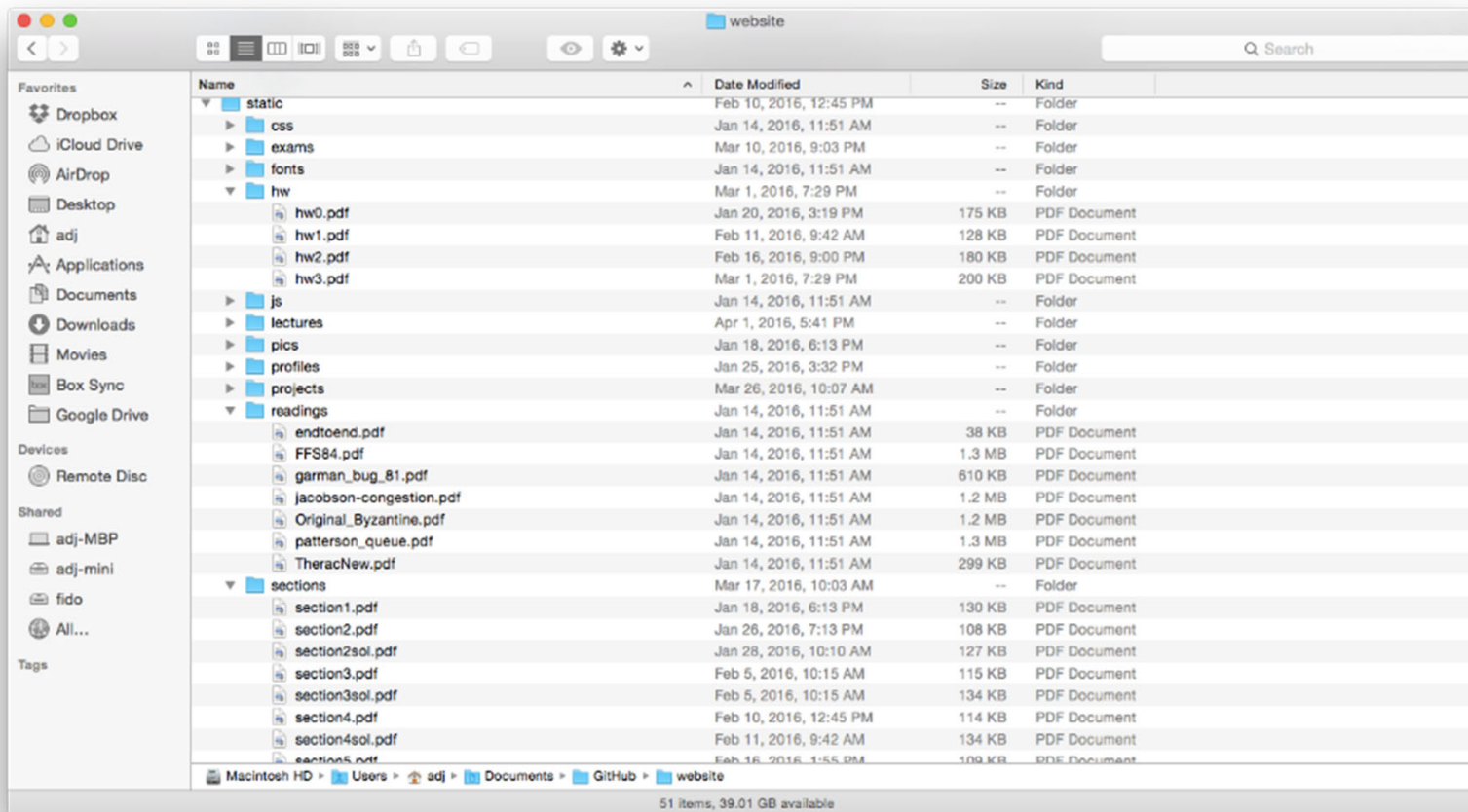


- Open performs *Name Resolution*
 - Translates path name into a *file number*
 - Checks access permissions (for file systems with permissions)
- File number is an index to *file header* structure describing the file
 - Often called an *inode*
 - File header gives us enough information to find all the blocks of the file – wherever they are on disk
- Read and Write operate on the file using the file header
 - Use file header as an “index” to locate the blocks
 - Much of the distinction between file systems is with respect to the structure of the file header and/or directories
- **4 components:**
 - **directory, file header, storage blocks, free space map**

How to get the File Number?

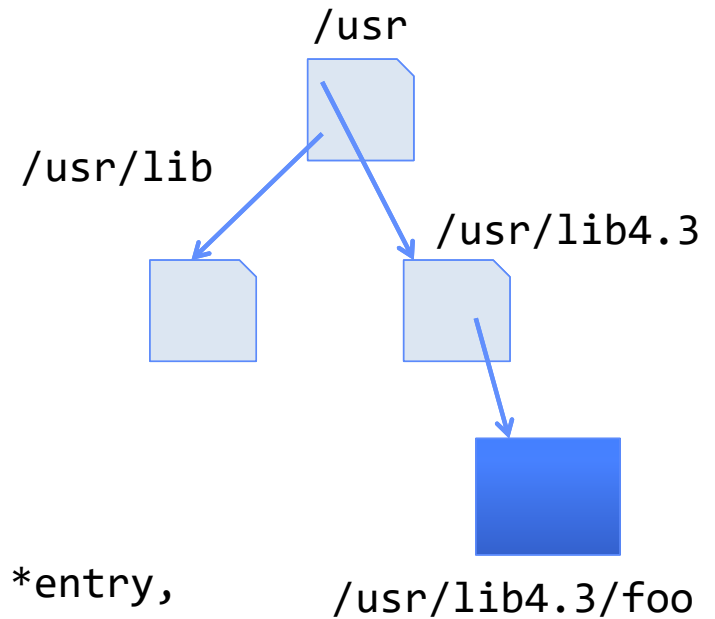
- Look up in *directory structure*
- A directory is a file containing <file_name : file_number> mappings
 - File number could be a file or another directory
 - Operating system stores the mapping in the directory in a format it interprets
 - Each <file_name : file_number> mapping is called a directory entry
- Process isn't allowed to read the raw bytes of a directory
 - The **read** function doesn't work on a directory
 - Instead, see **readdir**, which iterates over the map without revealing the raw bytes
- Why shouldn't the OS let processes read/write the bytes of a directory?

Directories



Directory Abstraction

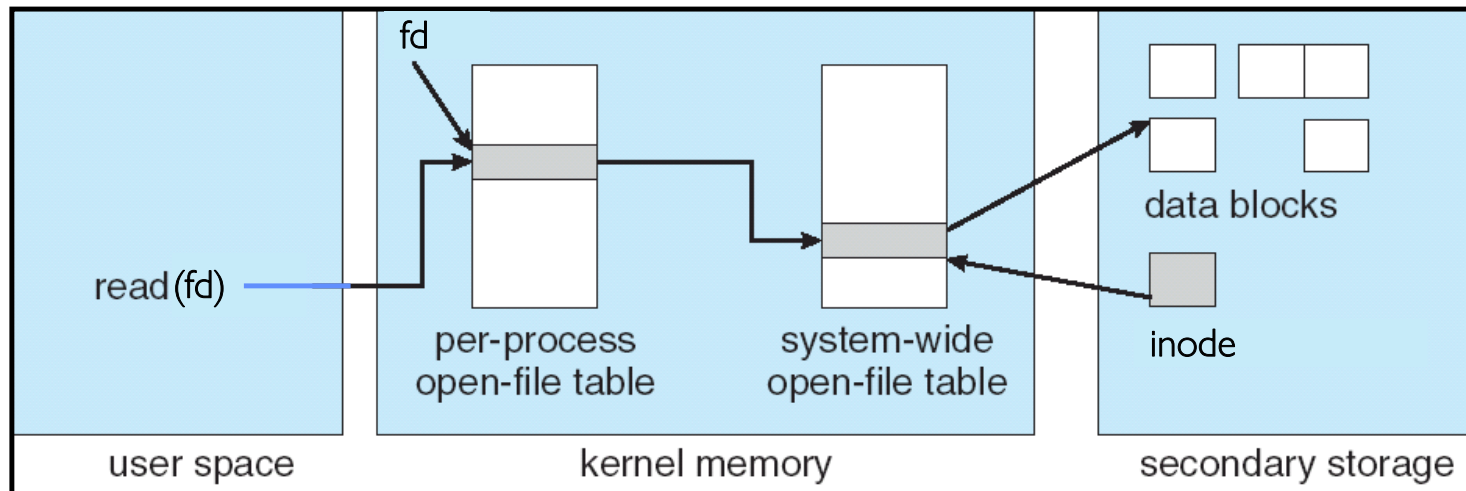
- Directories are specialized files
 - Contents: List of pairs <file name, file number>
- System calls to access directories
 - `open` / `creat` / `readdir` traverse the structure
 - `mkdir` / `rmdir` add/remove entries
 - `link` / `unlink` (`rm`)
- libc support
 - `DIR * opendir (const char *dirname)`
 - `struct dirent * readdir (DIR *dirstream)`
 - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`



Directory Structure

- How many disk accesses to resolve “/my/book/count”?
 - Read in file header for root (fixed spot on disk)
 - Read in first data block for root
 - » Table of file name/index pairs.
 - » Search linearly – ok since directories typically very small
 - Read in file header for “my”
 - Read in first data block for “my”; search for “book”
 - Read in file header for “book”
 - Read in first data block for “book”; search for “count”
 - Read in file header for “count”
- **Current working directory:** Per-address-space pointer to a directory used for resolving file names
 - Allows user to specify relative filename instead of absolute path (say CWD=“/my/book” can resolve “count”)

In-Memory File System Structures



- Open syscall: find inode on disk from pathname (traversing directories)
 - Create “in-memory inode” in system-wide open file table
 - One entry in this table no matter how many instances of the file are open
- Read/write syscalls look up in-memory inode using the file handle

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

Published in FAST 2007

Observation #1: Most Files Are Small

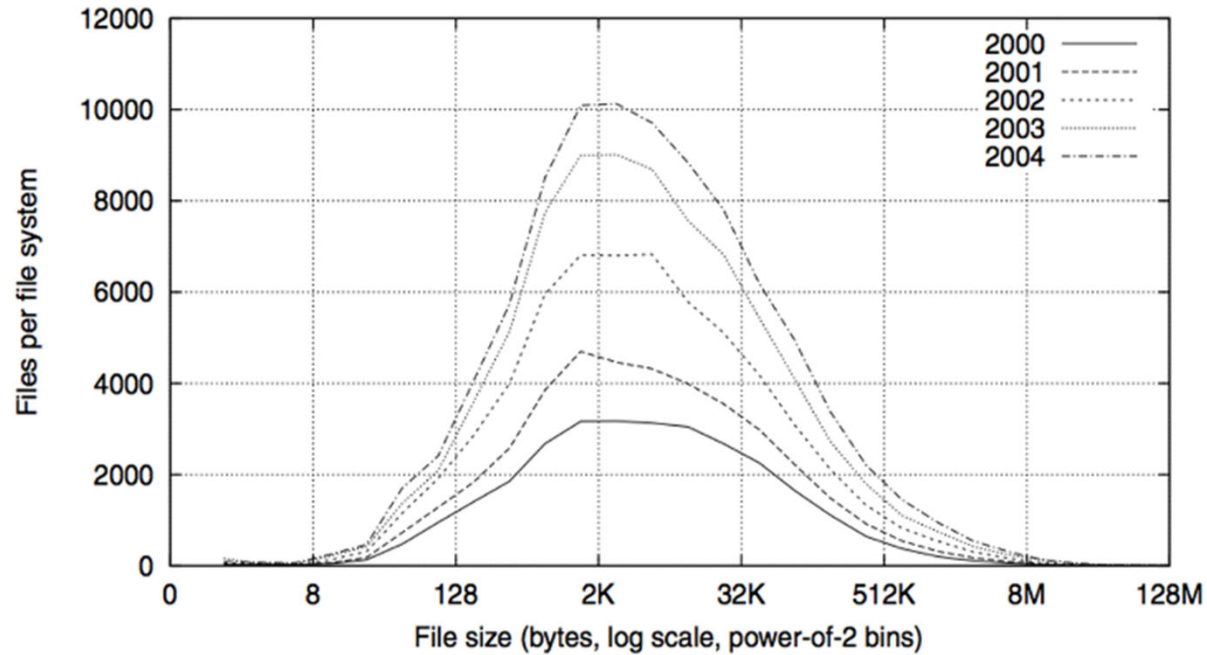


Fig. 2. Histograms of files by size.

Observation #2: Most Bytes are in Large Files

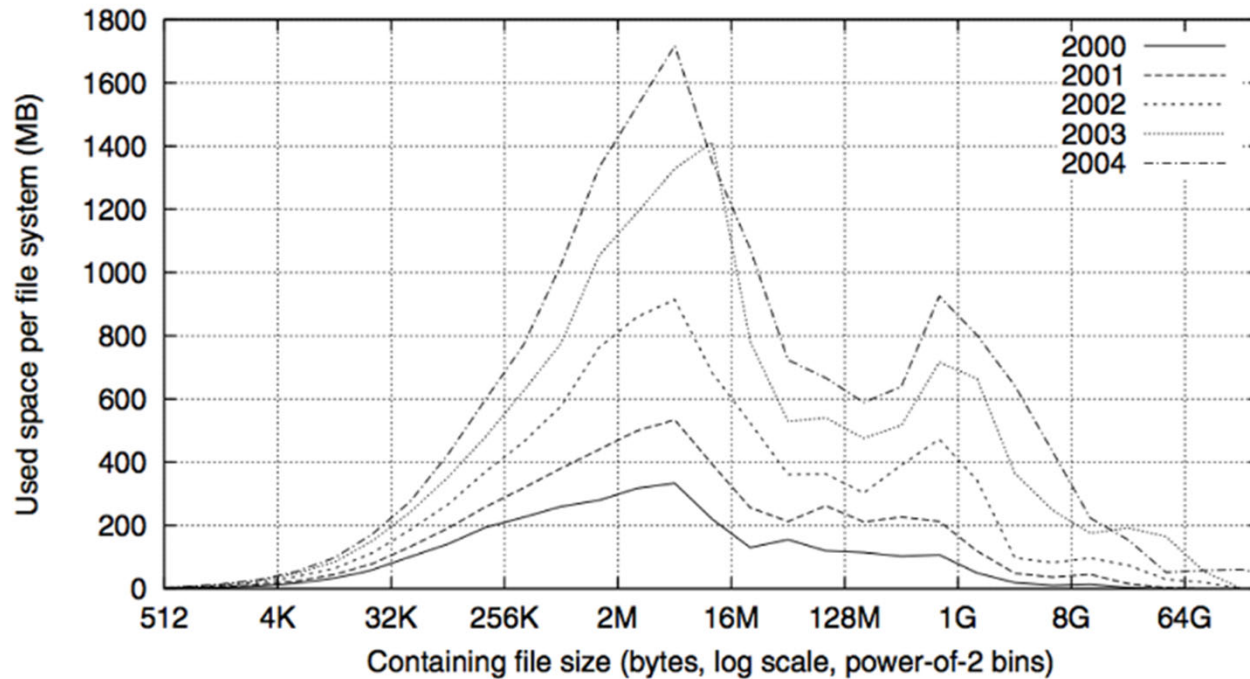
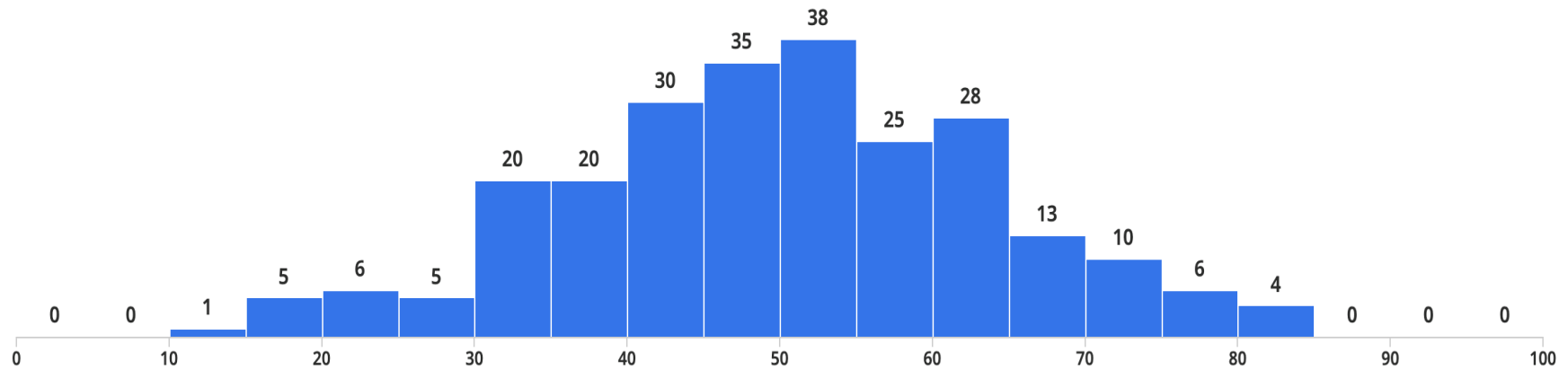


Fig. 4. Histograms of bytes by containing file size.

Administrivia

- Midterm 2 grading is done (**Hot off the presses!**)
- Watch for post on Ed about regrades.



Minimum

14.85

Median

50.35

Maximum

84.9

Mean

49.95

Std Dev [?](#)

14.18

Administrivia (con't)

- Midterm 3 on Thursday, April 30
 - All topics up to previous Tuesday (4/28) are in scope
 - Closed book, 3 pages, double-sided *handwritten* notes.
- Project 3 is out!
 - Get moving, since the end of term is approaching rapidly!
 - (That light you see is an approaching train!)
 - Design document due date moved to 4/18 😊
- Other deadlines:
 - HW5 Checkpoint moved to Saturday 4/20 😊
 - HW5 Due ⇒ Friday 4/24
 - HW6 Release ⇒ Saturday 4/25

Class Attendance:
4/16/2026

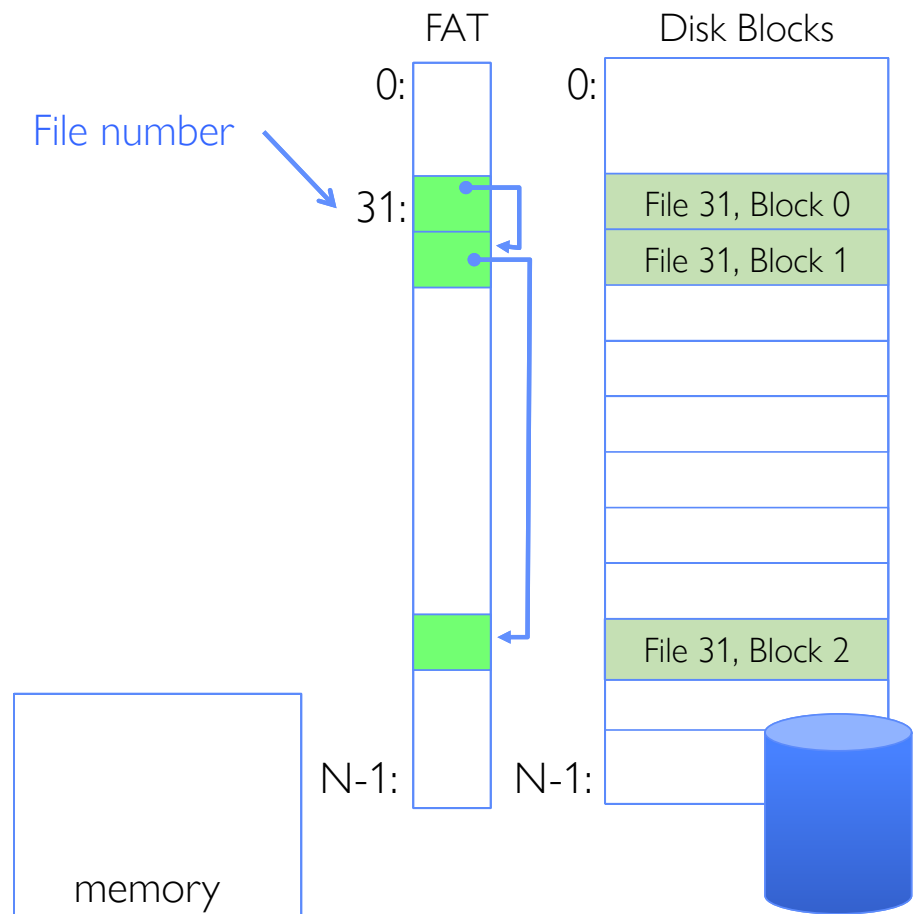


CASE STUDY: FAT: FILE ALLOCATION TABLE

- MS-DOS, 1977
- Still widely used!

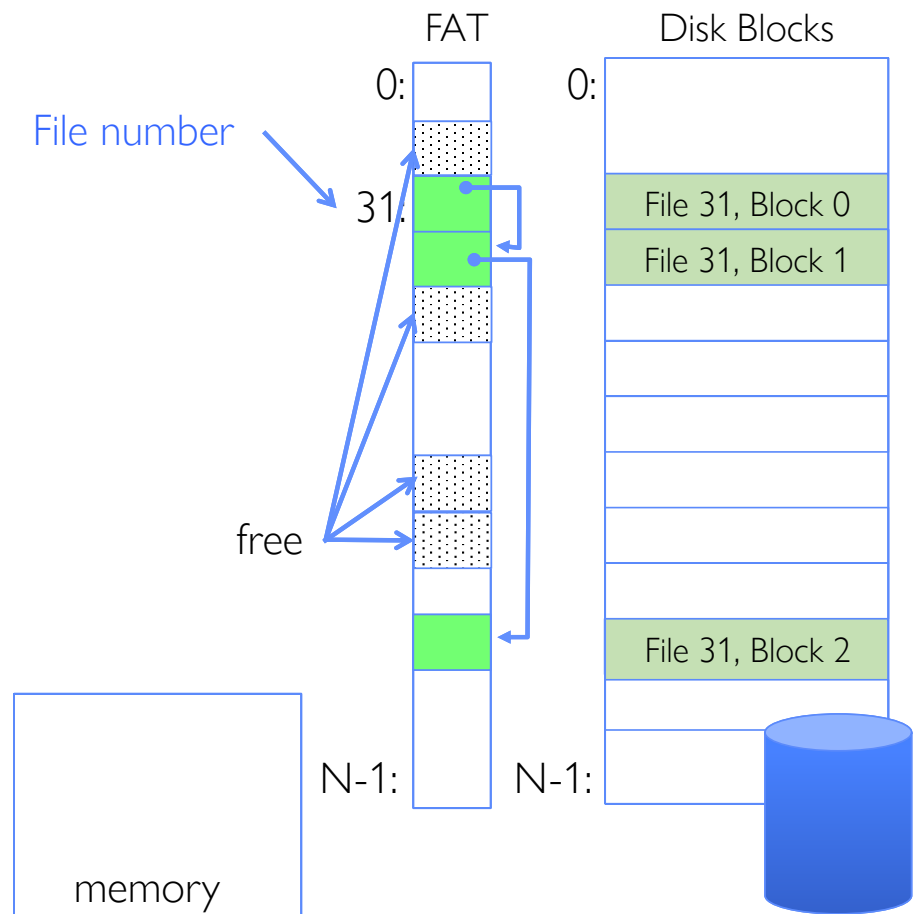
FAT (File Allocation Table): Example

- Assume (for now) we have a way to translate a path to a “file number”
 - i.e., a directory structure
- Disk Storage is a collection of Blocks
 - Just hold file data (offset $o = \langle B, x \rangle$)
- Example: `file_read 31, < 2, x >`
 - Index into FAT with file number
 - Follow linked list to block
 - Read the block from disk into memory



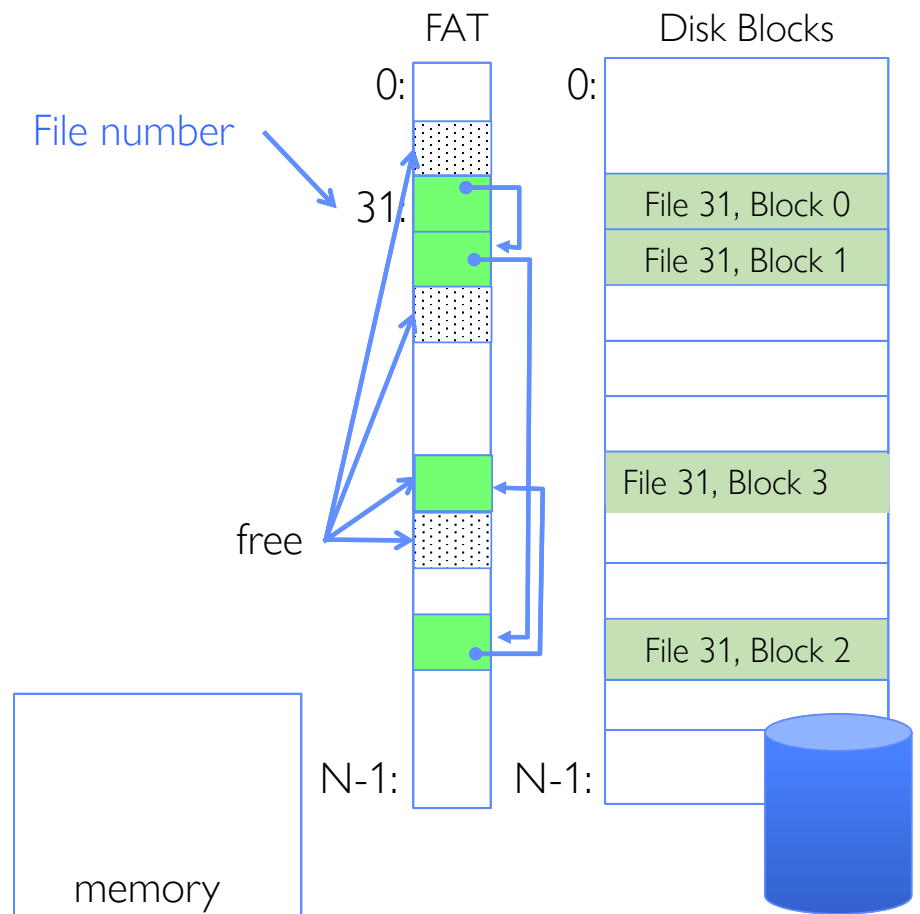
FAT: Details

- File is a collection of disk blocks
- FAT is linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get block number
- Unused blocks marked free
 - Could require scan to find
 - Or, could use a free list



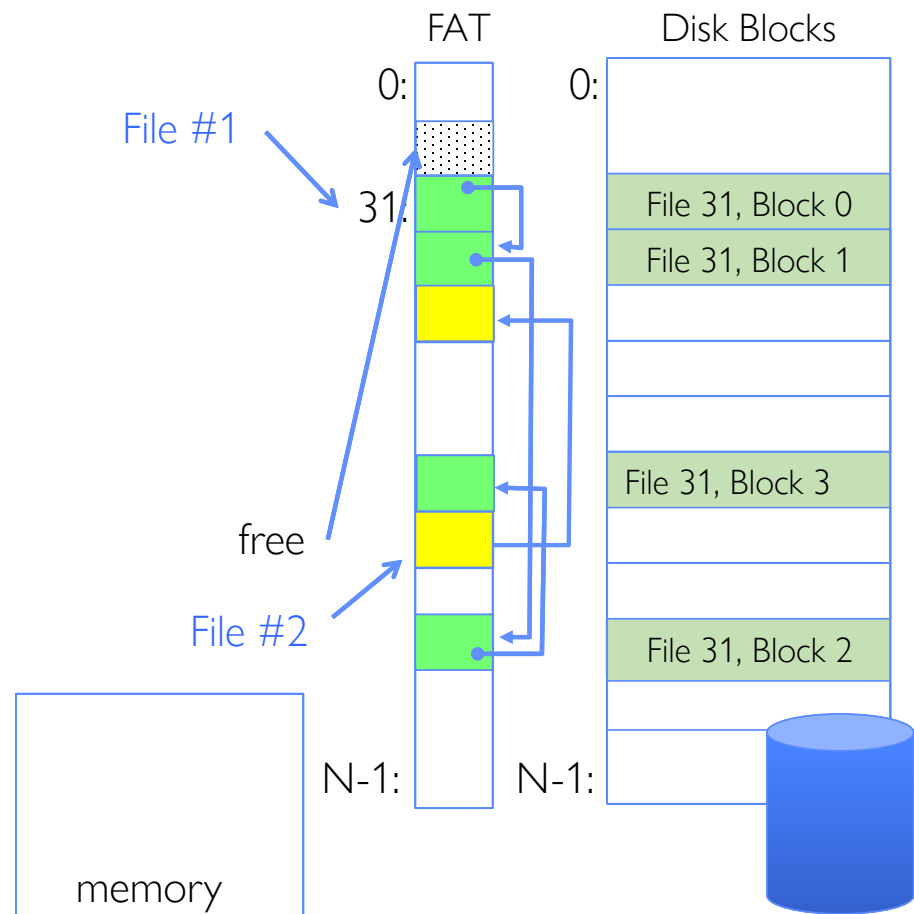
FAT: Example (Con't)

- Ex: `file_write(31, < 3, y >)`
 - Grab free block
 - Linking them into file
- Note that files have no locality by default
 - Files become “fragmented” over time

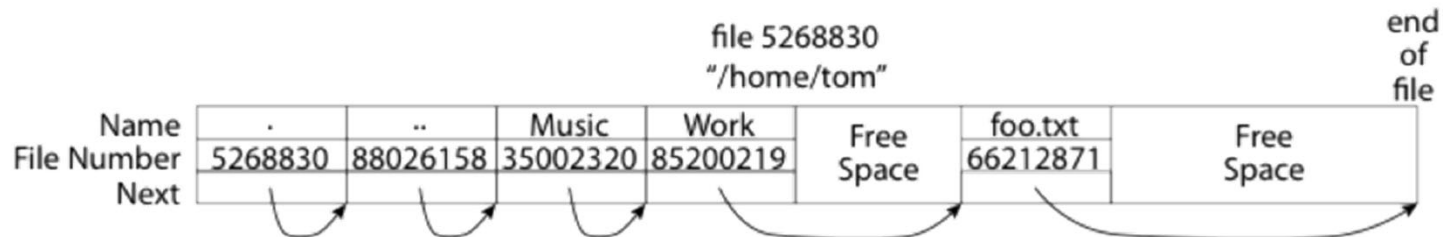


FAT: Other Details

- Where is FAT stored?
 - On disk
- How to format a disk?
 - Zero the blocks, mark FAT entries “free”
- How to quick format a disk?
 - Mark FAT entries “free”
- Simple: can implement in device firmware



FAT: Directories

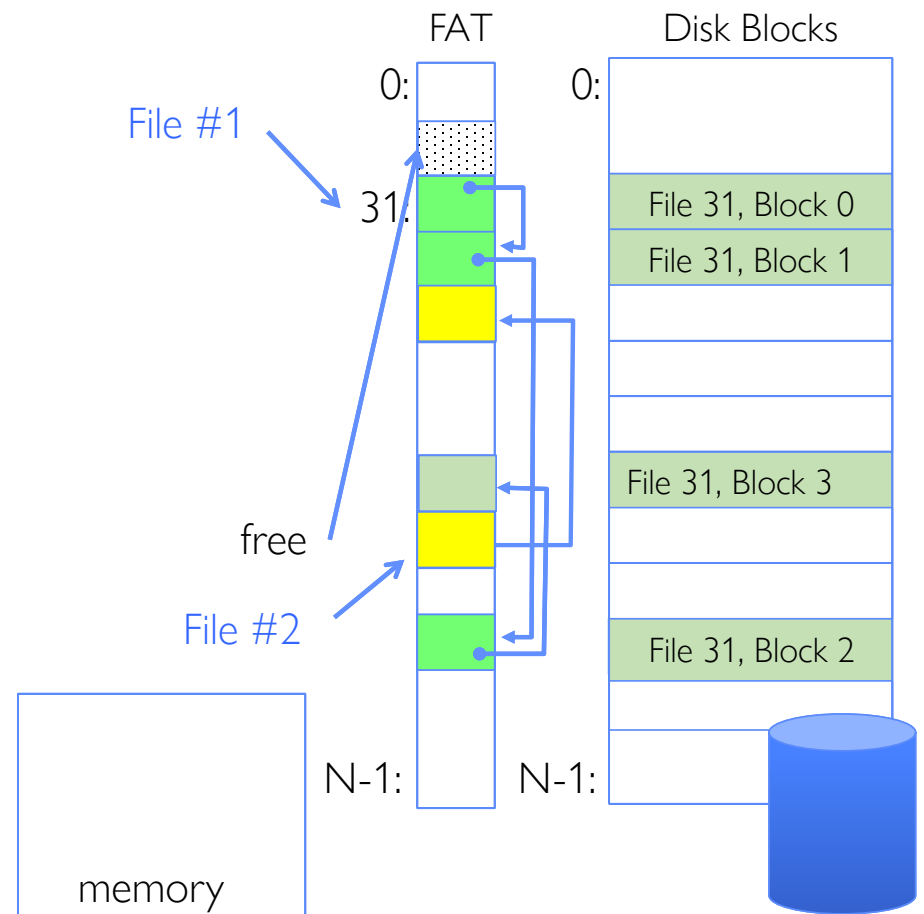


- A directory is a file containing <file_name: file_number> mappings
- Free space for new/deleted entries
- In FAT: file attributes are kept in directory (!!!)
 - Not directly associated with the file itself
- Each directory a linked list of entries
 - Requires linear search of directory to find particular entry
- Where do you find root directory (“/”)?
 - At well-defined place on disk
 - For FAT, this is at block 2 (there are no blocks 0 or 1)
 - Remaining directories

FAT Discussion

Suppose you start with the file number:

- Time to find block?
- Block layout for file?
- Sequential access?
- Random access?
- Fragmentation?
- Small files?
- Big files?



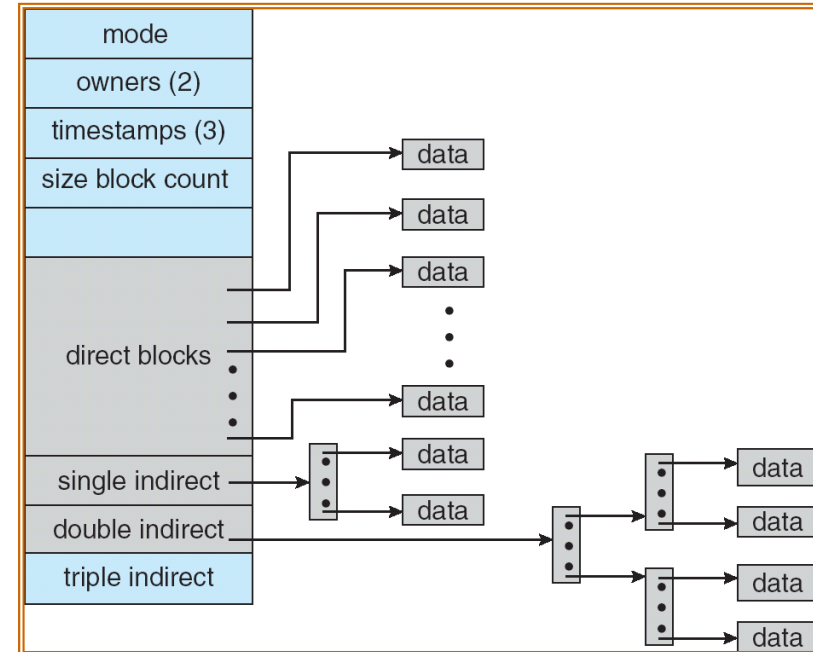
CASE STUDY: UNIX FILE SYSTEM (BERKELEY FFS)

Inodes in Unix (Including Berkeley FFS)

- Index structure is an array of *inodes*
 - File Number (inumber) is an index into the array of inodes
 - Each inode corresponds to a file and **contains its metadata**
 - » So, things like read/write permissions are stored with *file*, not in directory
 - » Allows multiple names (directory entries) for a file
- 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 BSD 4.1 (more following)
 - Berkeley Standard Distribution Unix!
 - Part of your heritage!
 - Similar structure for Linux Ext 2/3

Original Multilevel Indexed Files (4.1 BSD, 1981)

- Original 4.1BSD file system
 - 10 “direct pointers” to blocks (1KB blocks)
 - 1 “single indirect ptr” (to block with 256 ptrs to blocks)
 - 1 “double indirect ptr” (to block with 256 indirect ptrs)
 - Metadata associated with FILE, not directory entry (like FAT)
- Sample file in multilevel indexed format:
 - How many accesses for block #23?
(assume file header accessed on open)?
 - » Two: One for indirect block, one for data
 - How about block #5?
 - » One: One for data
 - Block #340?
 - » Three: double indirect block, indirect block, and data
- UNIX 4.1 Pros and cons
 - Pros: Simple (more or less)
Files can easily expand (up to a point)
Small files particularly cheap and easy, large files can be handled!
 - Cons: Lots of seeks
Very large files must read many indirect block (four I/Os per block!)

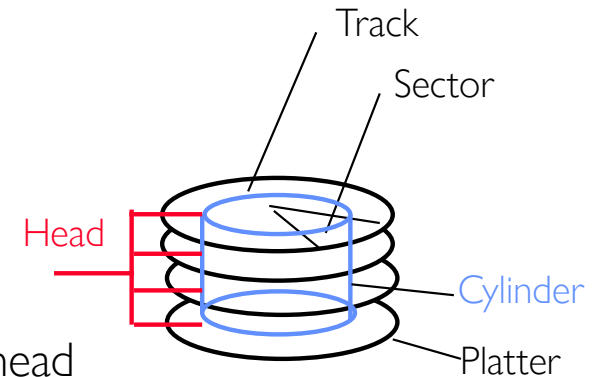


Recall: Critical Factors in File System Design

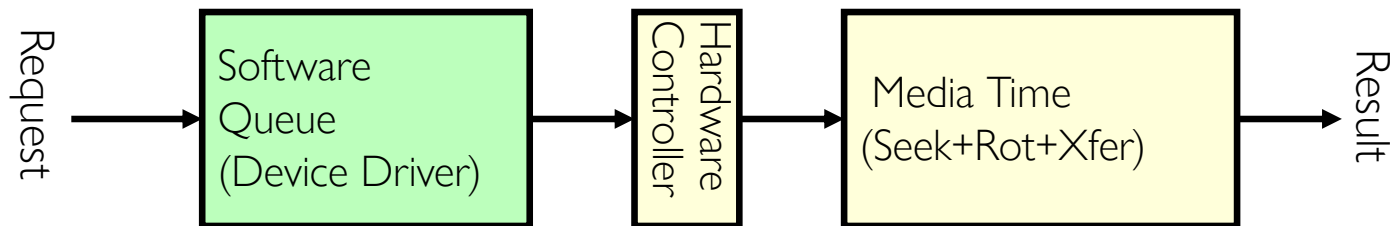
- (Hard) Disk Performance !!!
 - Maximize sequential access, minimize seeks
- Open before Read/Write
 - Can perform protection checks and look up where the actual file resource are, in advance
- Size is determined as they are used !!!
 - Can write (or read zeros) to expand the file
 - Start small and grow, need to make room
- Organized into directories
 - What data structure (on disk) for that?
- Need to carefully allocate / free blocks
 - Such that access remains efficient

Recall: Magnetic Disks

- **Cylinders:** all the tracks under the head at a given point on all surfaces
- Read/write data is a three-stage process:
 - **Seek time:** position the head/arm over the proper track
 - **Rotational latency:** wait for desired sector to rotate under r/w head
 - **Transfer time:** transfer a block of bits (sector) under r/w head



$$\text{Disk Latency} = \text{Queueing Time} + \text{Controller time} + \text{Seek Time} + \text{Rotation Time} + \text{Xfer Time}$$



Fast File System (BSD 4.2, 1984)

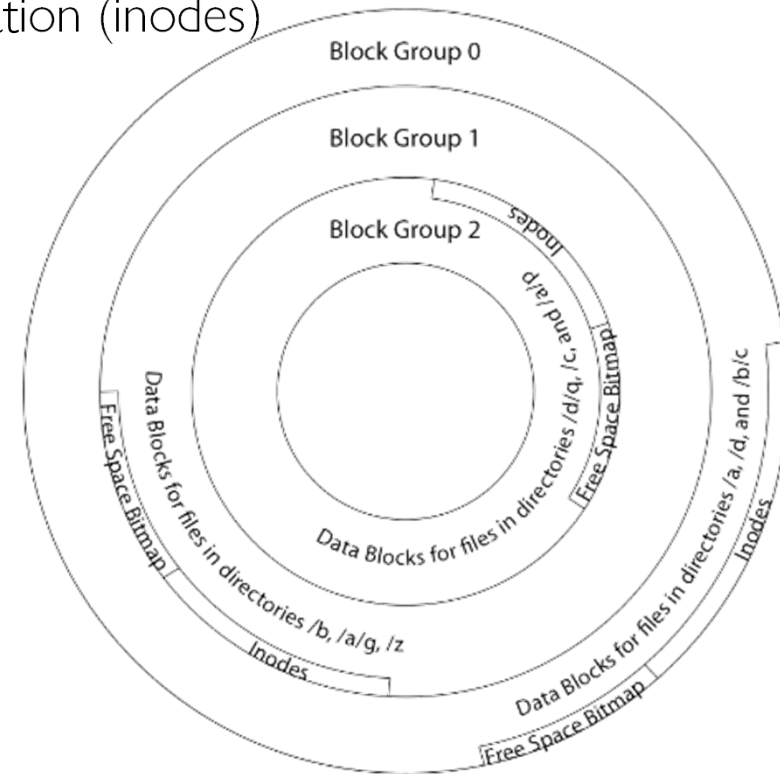
- Same inode structure as in BSD 4.1
 - same file header and triply indirect blocks like we just studied
 - Some changes to block sizes from 1024 \Rightarrow 4096 bytes for performance
- Paper on FFS: “A Fast File System for UNIX”
 - Marshall McKusick, William Joy, Samuel Leffler and Robert Fabry
 - Off the “resources” page of course website – Take a look!
- Optimization for Performance and Reliability:
 - Distribute inodes among different tracks to be closer to data
 - Uses bitmap allocation in place of freelist
 - Attempt to allocate files contiguously
 - 10% reserved disk space
 - Skip-sector positioning (mentioned later)

FFS Changes in Inode Placement: Motivation

- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
 - Fixed size, set when disk is formatted
 - » At formatting time, a fixed number of inodes are created
 - » Each is given a unique number, called an “inumber”
- Problem #1: Inodes all in one place (outer tracks)
 - Head crash potentially destroys all files by destroying inodes
 - Inodes not close to the data that they point to
 - » To read a small file, seek to get header, seek back to data
- Problem #2: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
 - How much contiguous space do you allocate for a file?
 - Makes it hard to optimize for performance

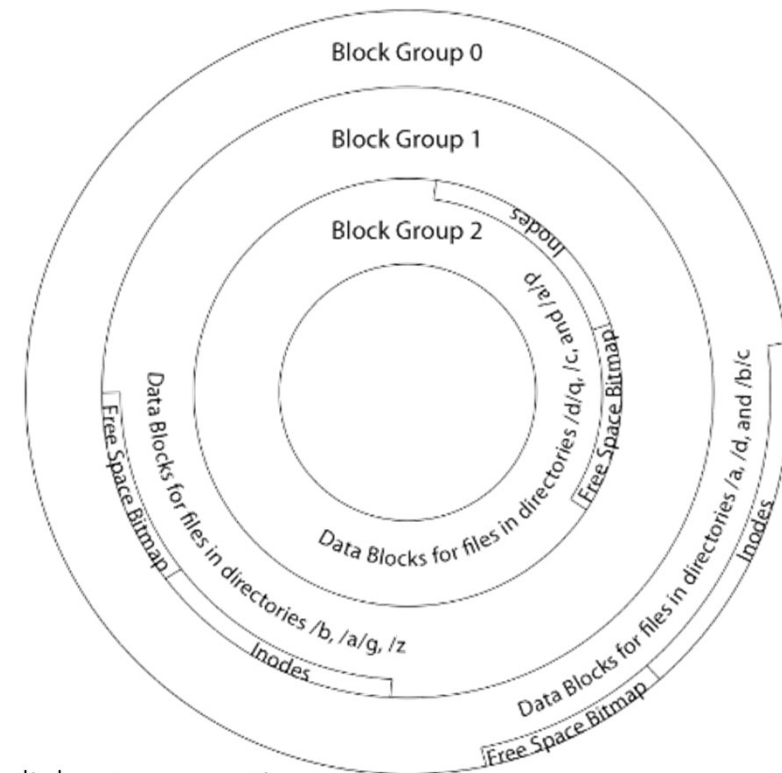
FFS Locality: Block Groups

- The UNIX BSD 4.2 (FFS) distributed the header information (inodes) closer to the data blocks
 - Often, inode for file stored in same “cylinder group” as parent directory of the file
 - makes an “ls” of that directory run very fast
- File system volume divided into set of block groups
 - Close set of tracks
- Data blocks, metadata, and free space interleaved within block group
 - Avoid huge seeks between user data and system structure
- Put directory and its files in common block group

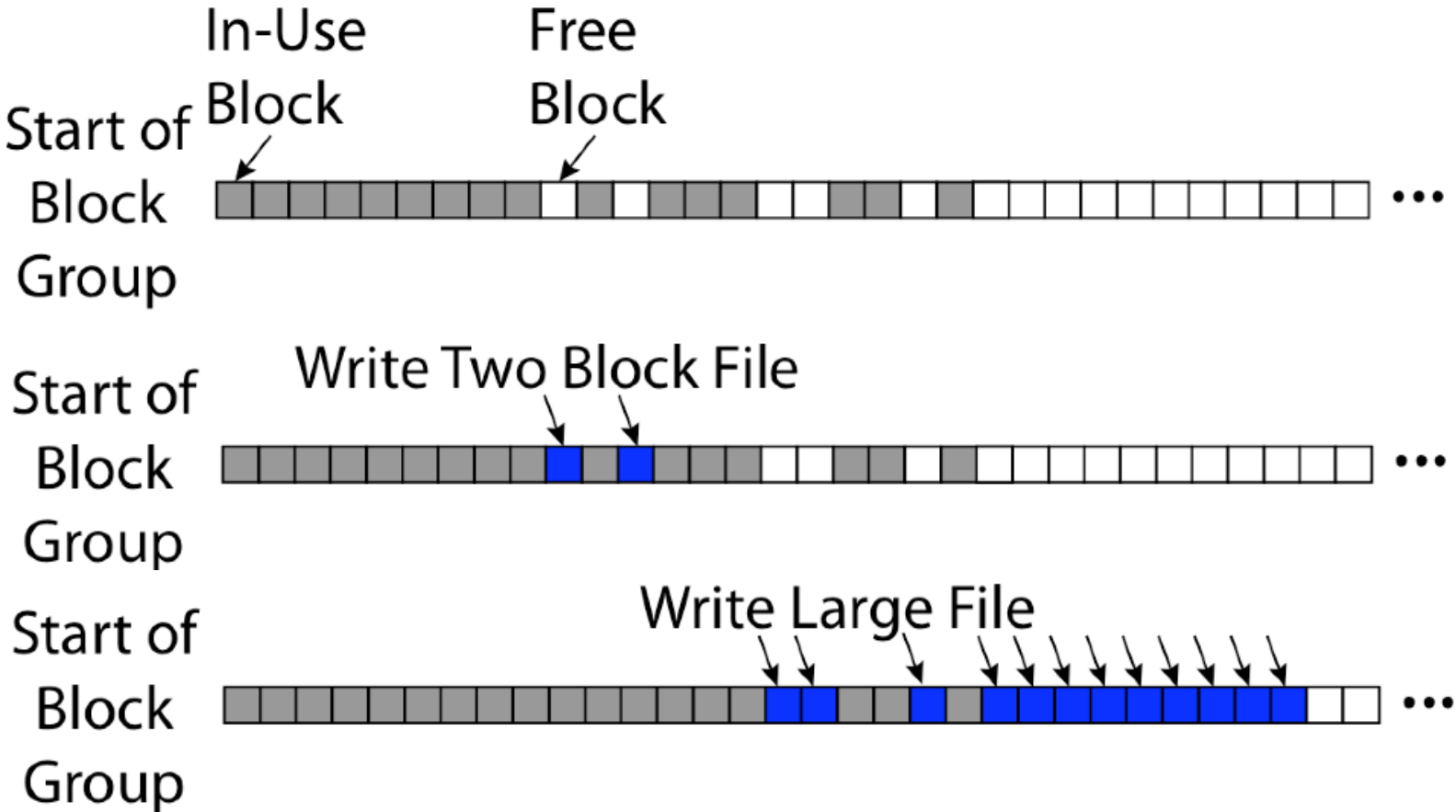


FFS Locality: Block Groups (Con't)

- First-Free allocation of new file blocks
 - To expand file, first try successive blocks in bitmap, then choose new range of blocks
 - Few little holes at start, big sequential runs at end of group
 - Avoids fragmentation
 - Sequential layout for big files
- Important: keep 10% or more free!
 - Reserve space in the Block Group
- Summary: FFS Inode Layout Pros
 - For small directories, can fit all data, file headers, etc. in same cylinder \Rightarrow no seeks!
 - File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
 - Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)

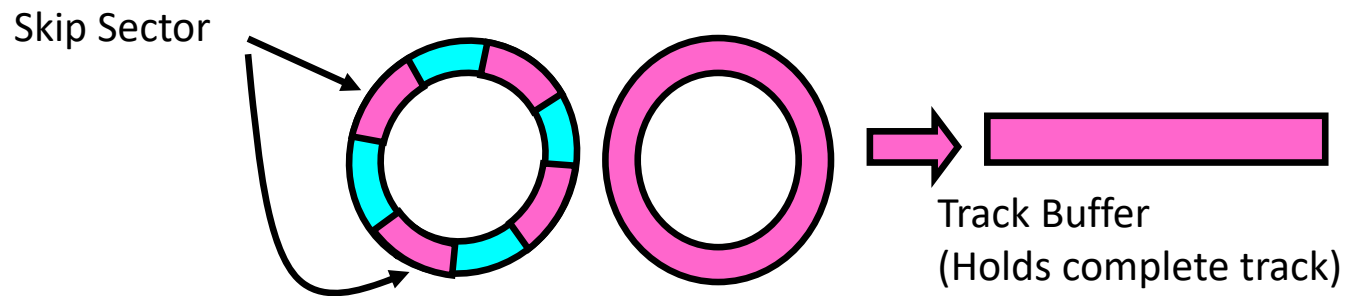


UNIX 4.2 BSD FFS First Fit Block Allocation



Attack of the Rotational Delay

- Problem 3: Missing blocks due to rotational delay
 - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



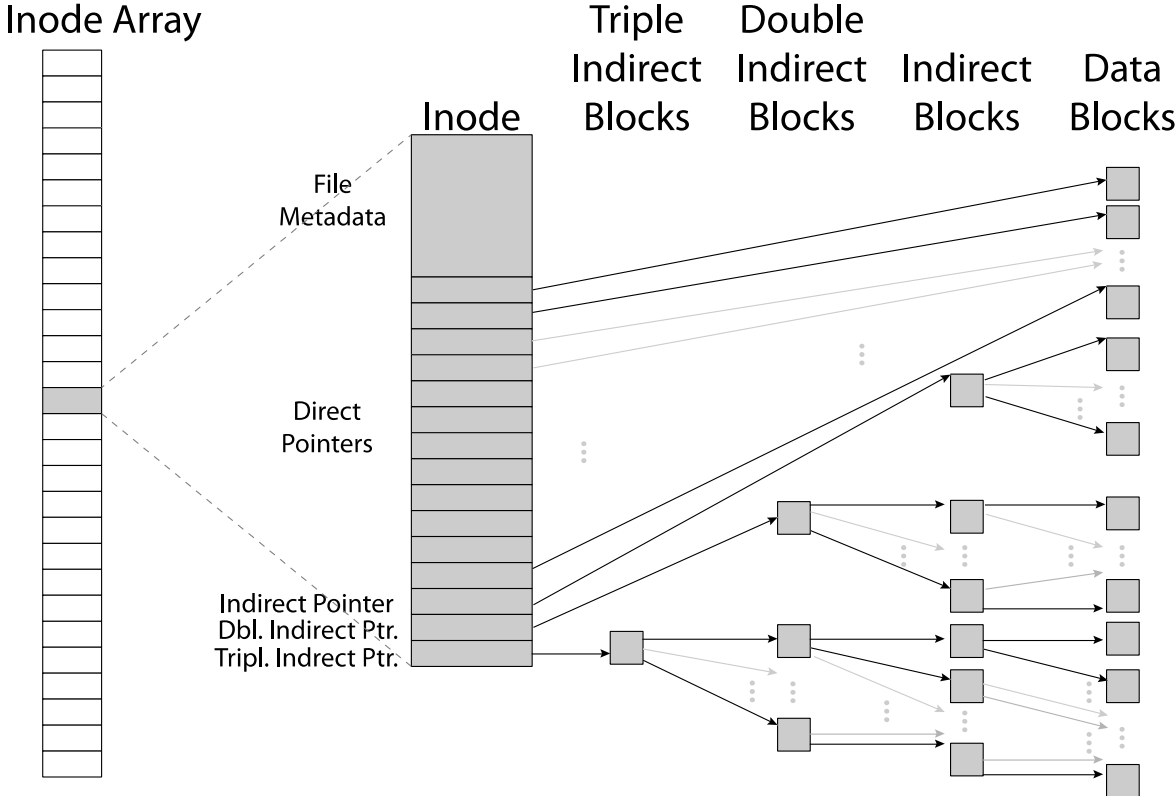
- Solution 1: Skip sector positioning (“interleaving”)
 - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
 - » Can be done by OS or in modern drives by the disk controller
- Solution 2: Read ahead: read next block right after first, even if application hasn’t asked for it yet
 - » This can be done either by OS (read ahead)
 - » By disk itself (track buffers) - many disk controllers have internal RAM that allows them to read a complete track
- Modern disks + controllers do many things “under the covers”
 - Track buffers, elevator algorithms, bad block filtering

UNIX 4.2 BSD FFS

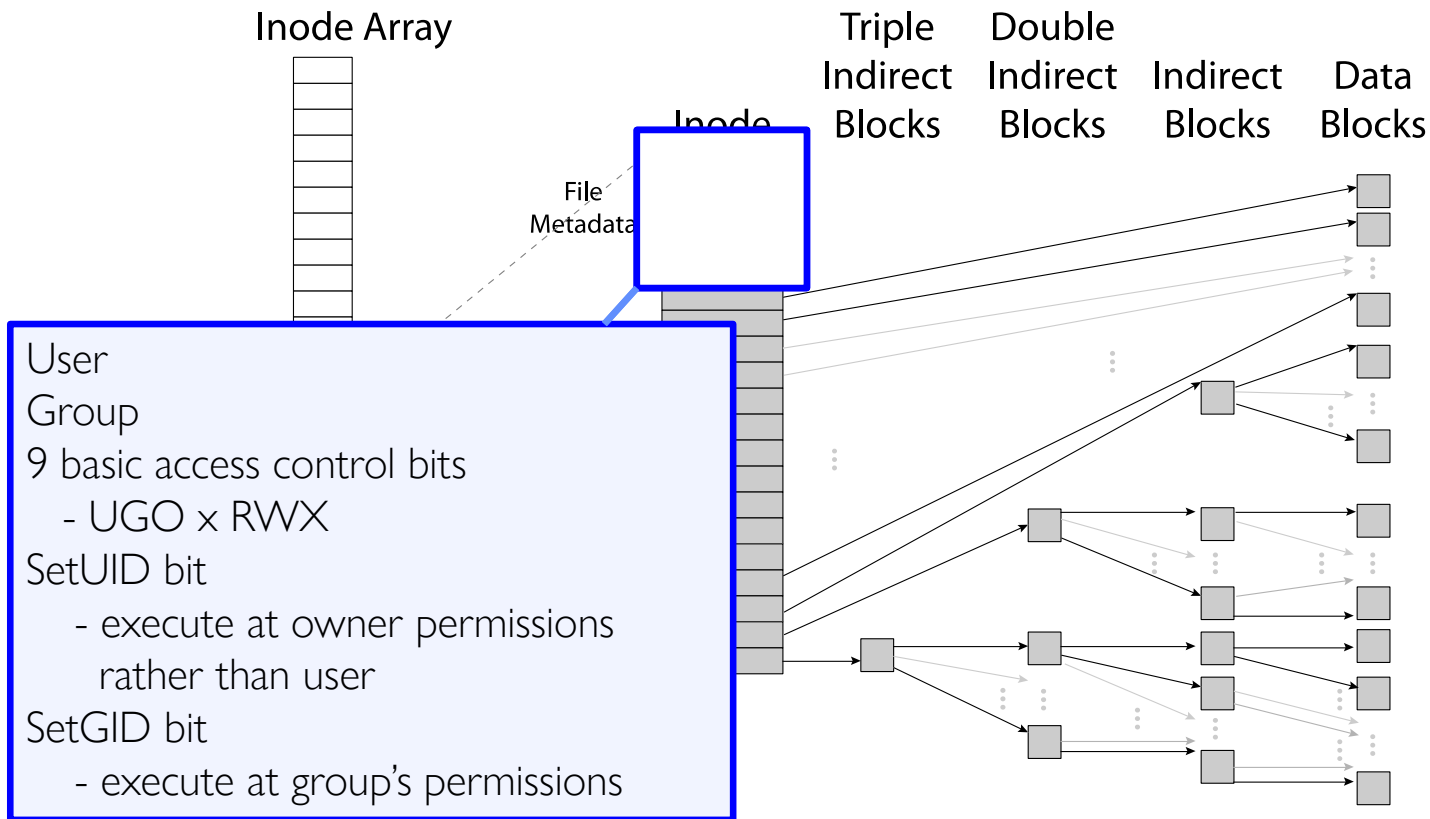
- Pros
 - Efficient storage for both small and large files
 - Locality for both small and large files
 - Locality for metadata and data
 - No defragmentation necessary!
- Cons
 - Inefficient for tiny files (a 1 byte file requires both an inode and a data block)
 - Inefficient encoding when file is mostly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

CASE STUDY: LINUX EXT2/EXT3 FILE SYSTEM

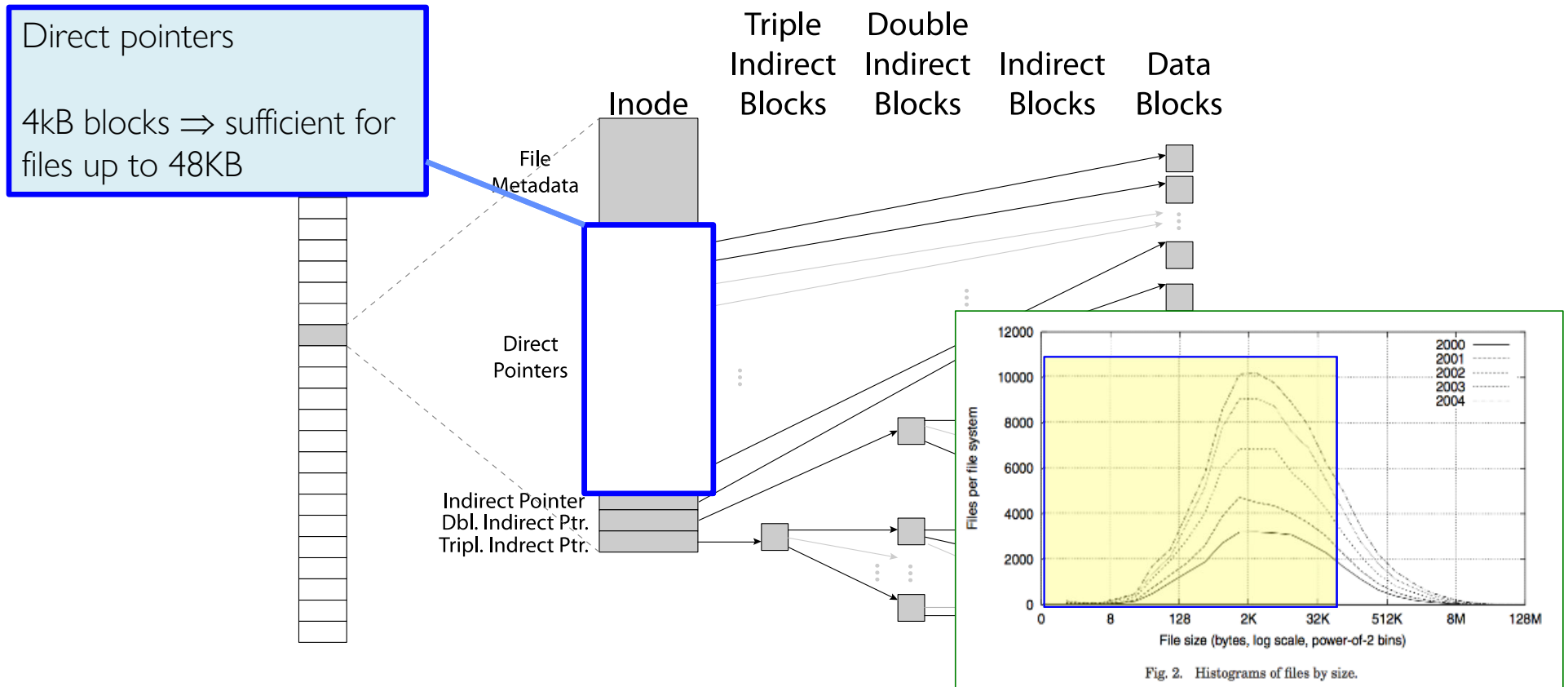
Ext3 (Linux) Inode Structure (128bytes)



Ext3 File Attributes/Metadata



Ext3 Small Files: 12 Pointers Direct to Data Blocks



Ext3 Large Files: 1-, 2-, 3-level indirect pointers

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

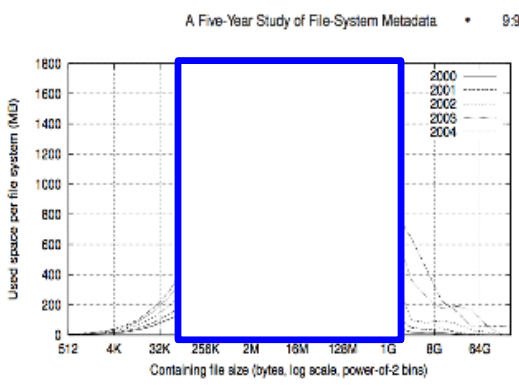
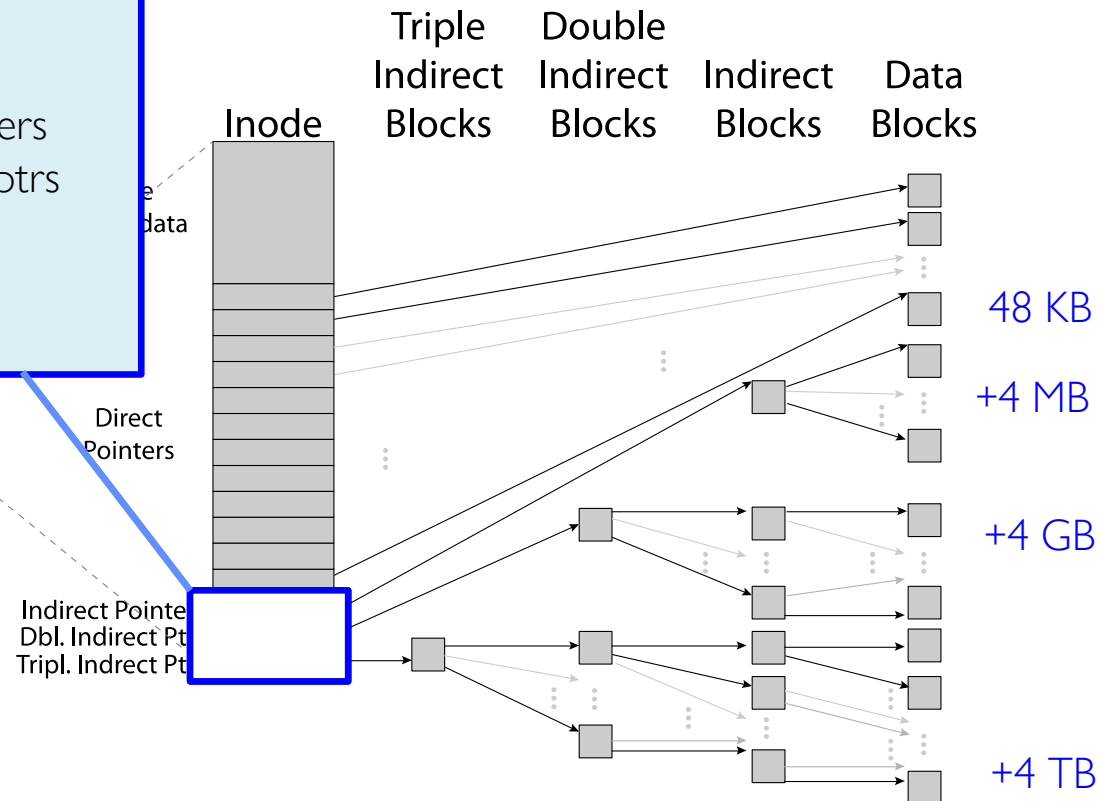
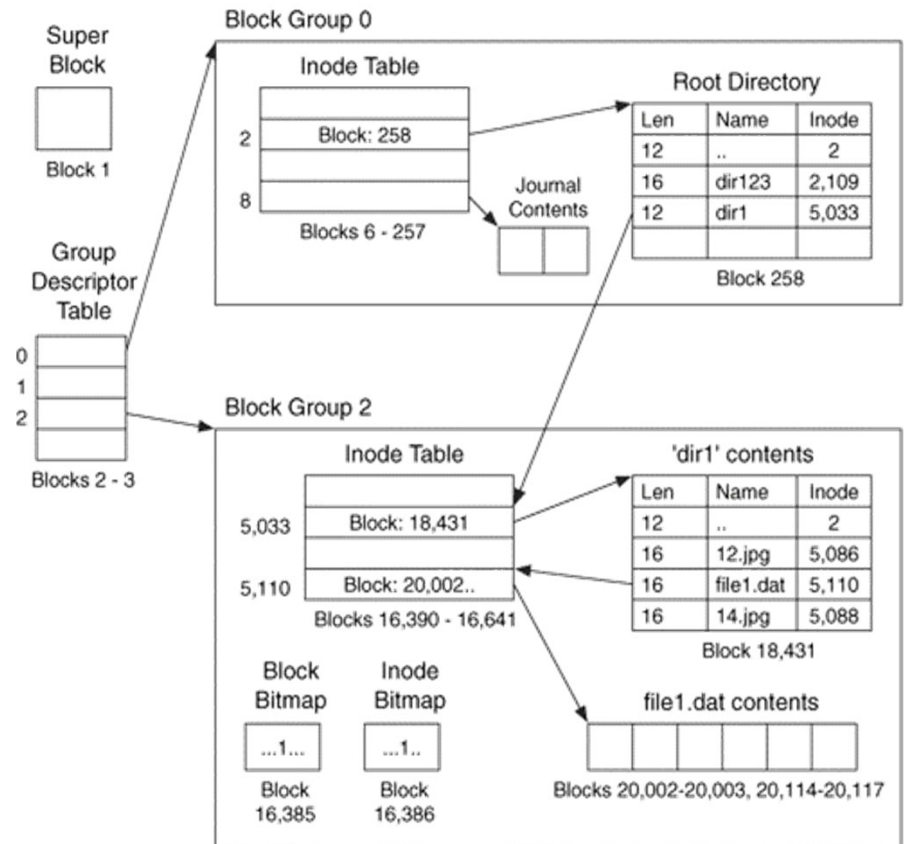


Fig. 4. Histograms of bytes by containing file size.

Linux Example: Ext2/3 Disk Layout

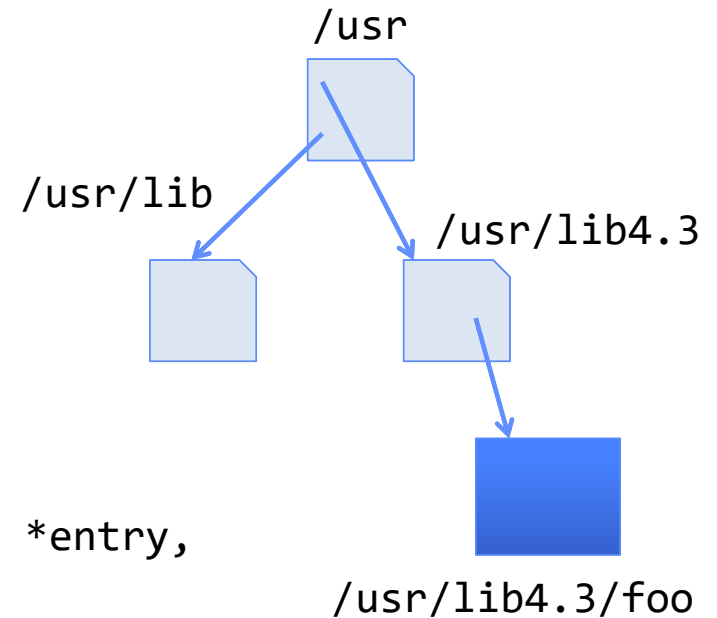
- Disk divided into block groups
 - Provides locality
 - Each group has two block-sized bitmaps (free blocks/inodes)
 - Block sizes settable at format time: 1K, 2K, 4K, 8K...
- Actual inode structure similar to 4.2 BSD
 - with 12 direct pointers
- Ext3: Ext2 with Journaling
 - Several degrees of protection with comparable overhead
 - We will talk about Journalling later



- Example: create a file1.dat under /dir1/ in Ext3

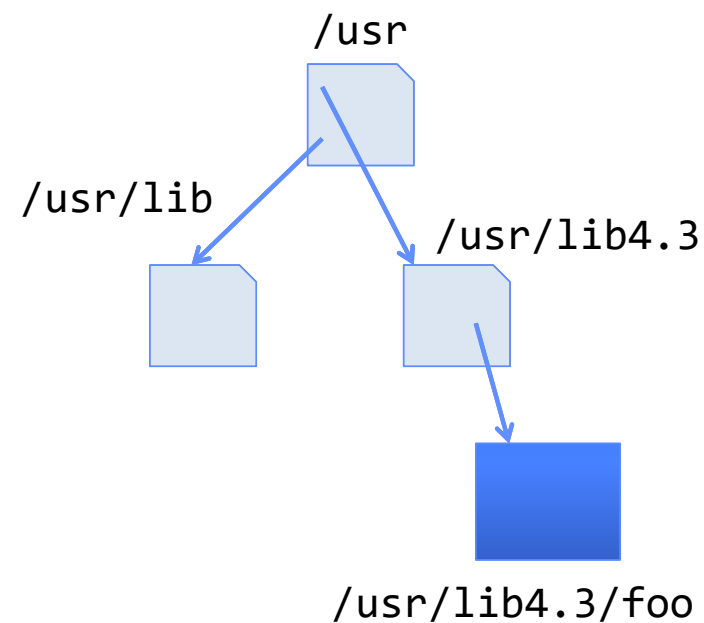
Recall: Directory Abstraction

- Directories are specialized files
 - Contents: List of pairs <file name, file number>
- System calls to access directories
 - **open / creat** traverse the structure
 - **mkdir / rmdir** add/remove entries
 - **link / unlink (rm)**
- libc support
 - `DIR * opendir (const char *dirname)`
 - `struct dirent * readdir (DIR *dirstream)`
 - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`



Hard Links

- Hard link
 - Mapping from name to file number in the directory structure
 - First hard link to a file is made when file created
 - Create extra hard links to a file with the `link()` system call
 - Remove links with `unlink()` system call
- When can file contents be deleted?
 - When there are no more hard links to the file
 - Inode maintains reference count for this purpose

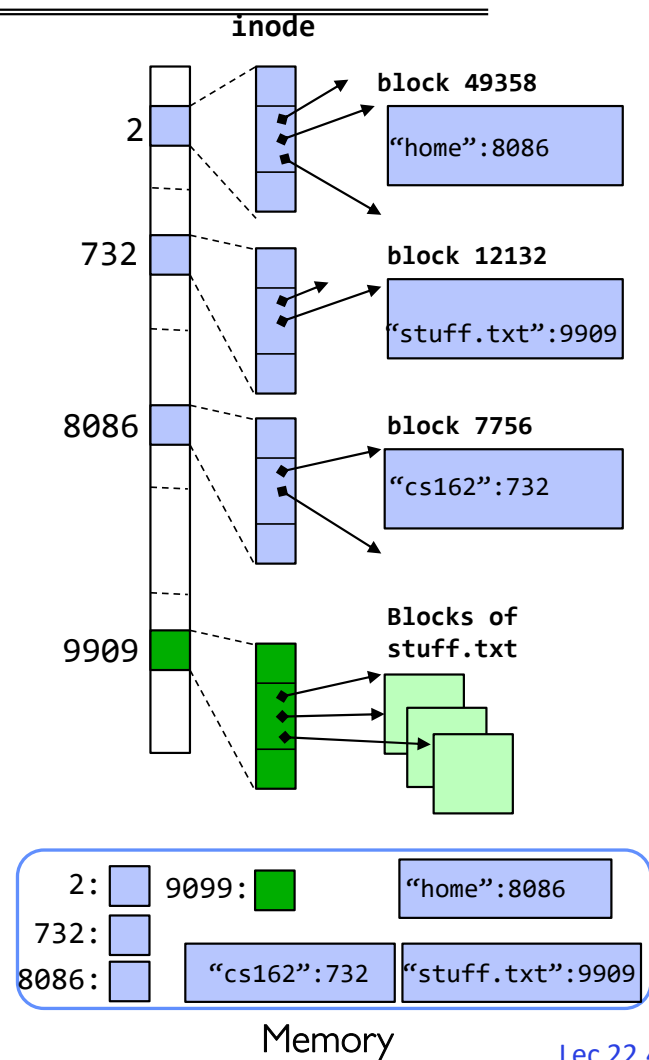


Soft Links (Symbolic Links)

- Soft link or Symbolic Link or Shortcut
 - Directory entry contains the path and name of the file
 - Map one name to another name
- Contrast these two different types of directory entries:
 - Normal directory entry: <file name, **file #**>
 - Symbolic link: <file name, **dest. file name**>
- OS looks up destination file name **each time** program accesses source file name
 - Lookup can fail (error result from **open**)
- Unix: Create soft links with **symlink** syscall

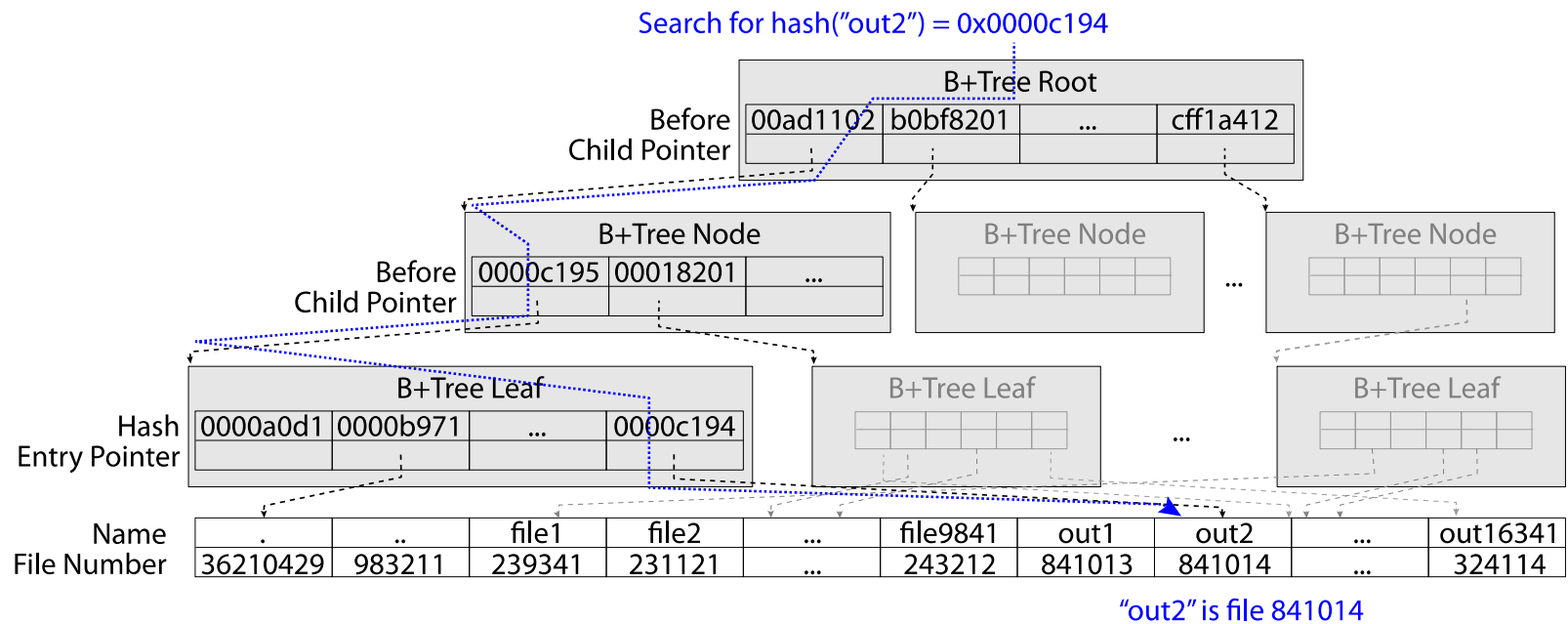
Directory Traversal

- What happens when we open `/home/cs162/stuff.txt`?
- `/` - inumber for root inode configured into kernel, say 2
 - Read inode 2 from its position in inode array on disk
 - Extract the direct and indirect block pointers
 - Determine block that holds root directory (say block 49358)
 - Read that block, scan it for “home” to get inumber for this directory (say 8086)
- Read inode 8086 for `/home`, extract its blocks, read block (say 7756), scan it for “cs162” to get its inumber (say 732)
- Read inode 732 for `/home/cs162`, extract its blocks, read block (say 12132), scan it for “stuff.txt” to get its inumber, say 9909
- Read inode 9909 for `/home/cs162/stuff.txt`
- Set up file description to refer to this inode so reads / write can access the data blocks referenced by its direct and indirect pointers
- **Check permissions on the final inode and each directory’s inode...**



Large Directories: B-Trees (dirhash)

in FreeBSD, NetBSD, OpenBSD





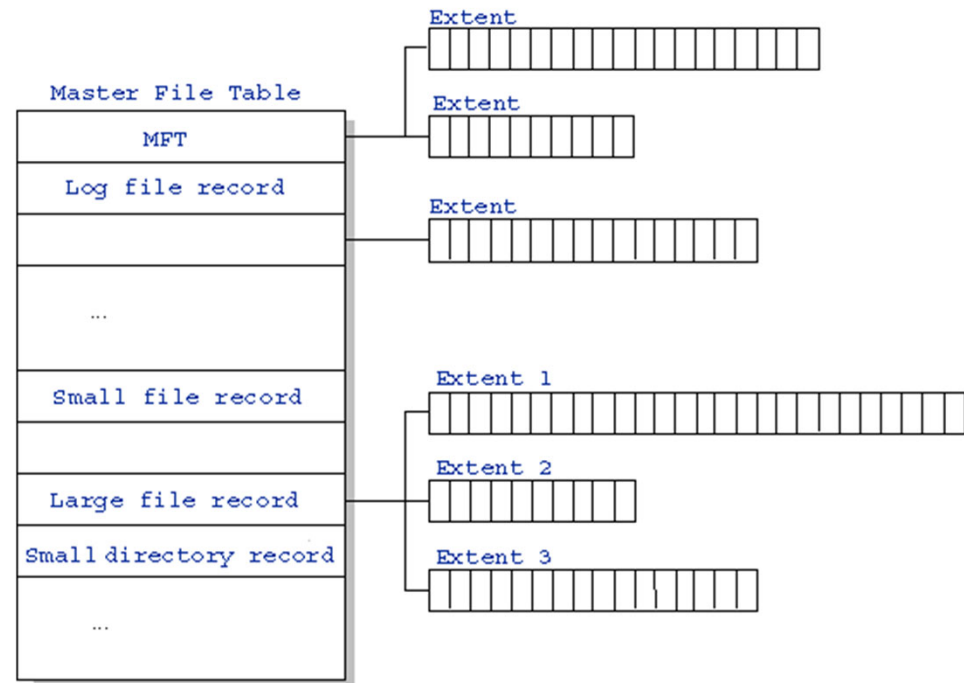
CASE STUDY: WINDOWS NTFS

New Technology File System (NTFS)

- Default on modern Windows systems
- Variable length extents
 - Rather than fixed blocks
- Instead of FAT or inode array: Master File Table
 - Like a database, with max 1 KB size for each table entry
 - Everything (almost) is a sequence of <attribute:value> pairs
 - » Meta-data and data
- Each entry in MFT contains metadata and:
 - File's data directly (for small files)
 - A list of *extents* (start block, size) for file's data
 - For big files: pointers to other MFT entries with *more* extent lists

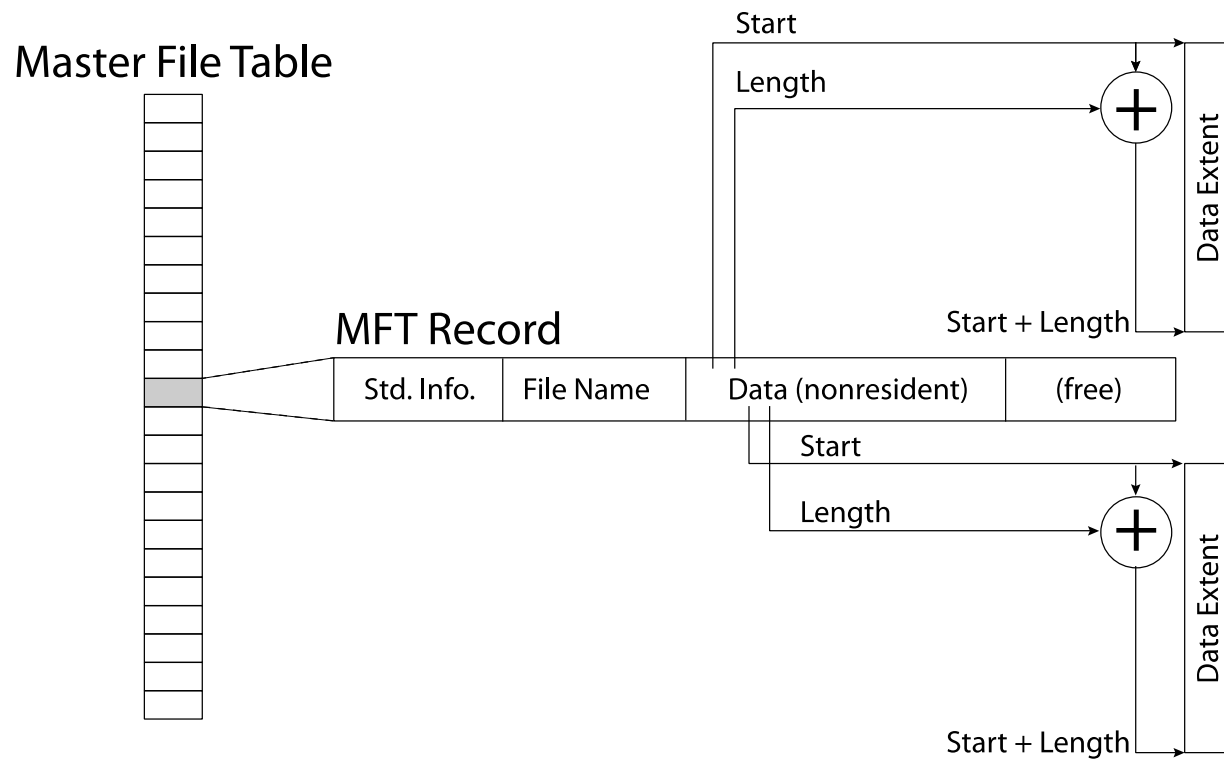
NTFS

- Master File Table
 - Database with Flexible 1KB entries for metadata/data
 - Variable-sized attribute records (data or metadata)
 - Extend with variable depth tree (non-resident)
- Extents – variable length contiguous regions
 - Block pointers cover runs of blocks
 - Similar approach in Linux (ext4)
 - File create can provide hint as to size of file
- Journaling for reliability
 - Discussed later

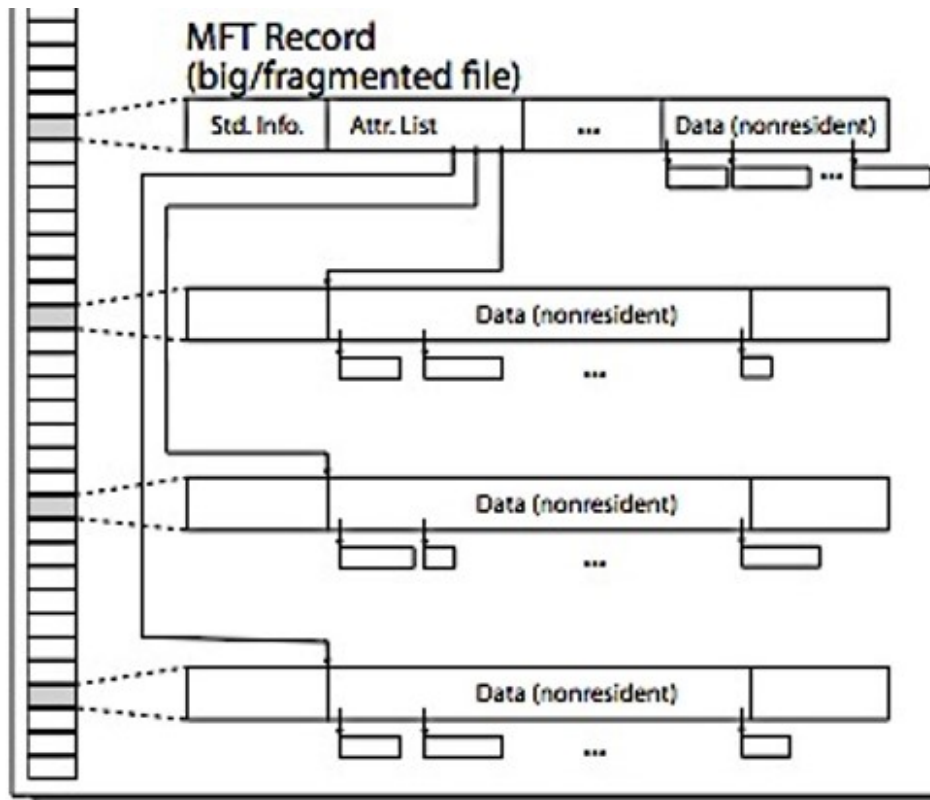


<http://ntfs.com/ntfs-mft.htm>

NTFS Medium File: Extents for File Data

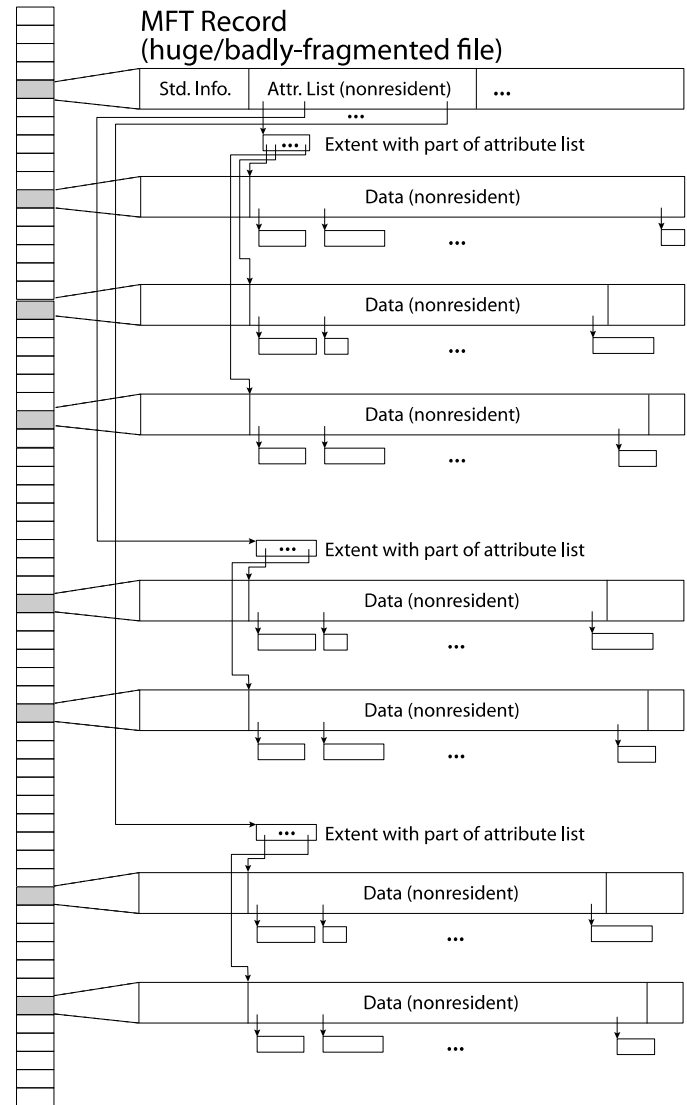


NTFS Large File: Pointers to Other MFT Records



NTFS Huge, Fragmented File: Many MFT Records

Master File Table



NTFS Directories

- Directories implemented as B Trees
- File's number identifies its entry in MFT
- MFT entry always has a file name attribute
 - Human readable name, file number of parent dir
- Hard link? Multiple file name attributes in MFT entry

Conclusion

- Naming: translating from user-visible names to actual sys resources
 - Directories used for naming for local file systems
 - Linked or tree structure stored in files
- Look at actual file access patterns
 - Many small files, but large files take up all the space!
- File Allocation Table (FAT) Scheme
 - Linked-list approach
 - Very widely used: Cameras, USB drives, SD cards
 - Simple to implement, but poor performance and no security
- 4.2 BSD Fast File System: Multi-level inode header to describe files
 - Inode contains ptrs to actual blocks, indirect blocks, double indirect blocks, etc.
 - Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization
- NTFS: Windows File System
 - Organized around a single Master File Table (like DataBase)
 - Extents for large contiguous regions of storage
 - Journaling for reliability (more later)