

Operating Systems (Honor Track)

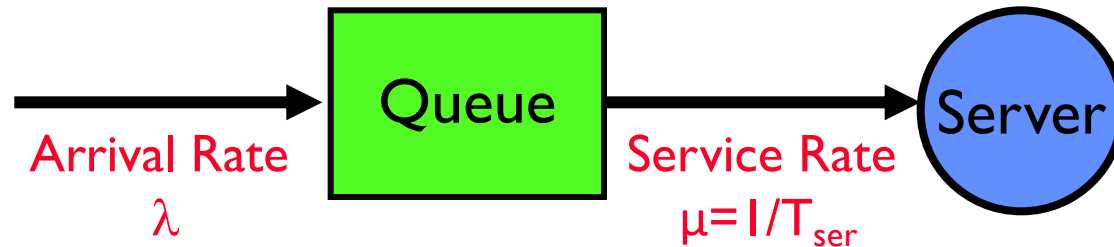
File System 2: File System Case Studies, Buffering

Xin Jin

Spring 2026

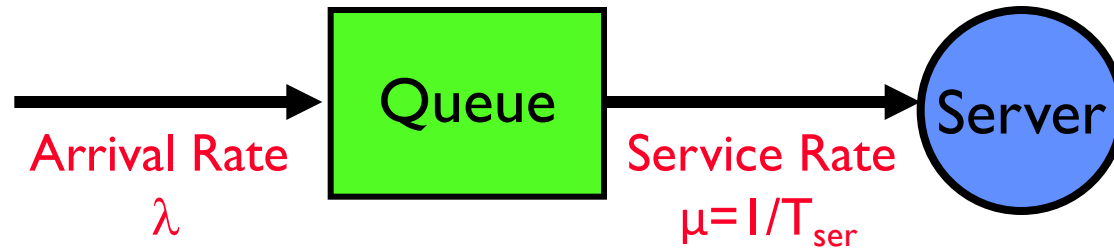
Recap: A Little Queuing Theory: Some Results (1/2)

- 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 (“m”)
 - 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 the queue
 - L_q : Length of queue = $\lambda \times T_q$ (by Little’s law)

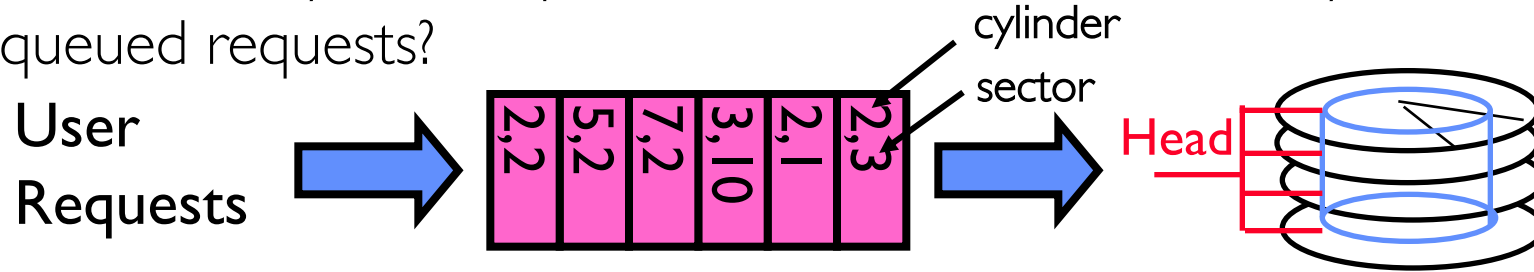
Recap: A Little Queuing Theory: Some Results (2/2)



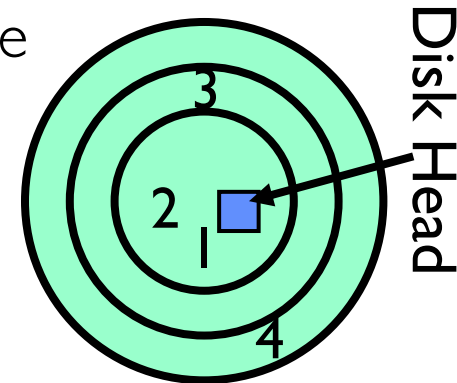
- Parameters that describe our system:
 - λ : mean number of arriving customers/second $\lambda = 1/T_A$
 - T_{ser} : mean time to service a customer (“m”)
 - C : squared coefficient of variance = σ^2/m^2
 - μ : service rate = $1/T_{ser}$
 - u : server utilization ($0 \leq u \leq 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
 - T_q : Time spent in the queue
 - L_q : Length of queue = $\lambda \times T_q$ (by Little’s law)
- **Results** (**M**: Poisson arrival process, **1** server):
 - **M**emoryless service time distribution ($C = 1$): Called an **M/M/1** queue
 - » $T_q = T_{ser} \times u/(1 - u)$
 - **G**eneral service time distribution (no restrictions): Called an **M/G/1** queue
 - » $T_q = T_{ser} \times \frac{1}{2}(1 + C) \times u/(1 - u)$

Recap: Disk Scheduling (1/3)

- Disk can do only one request at a time; What order do you choose to do queued requests?

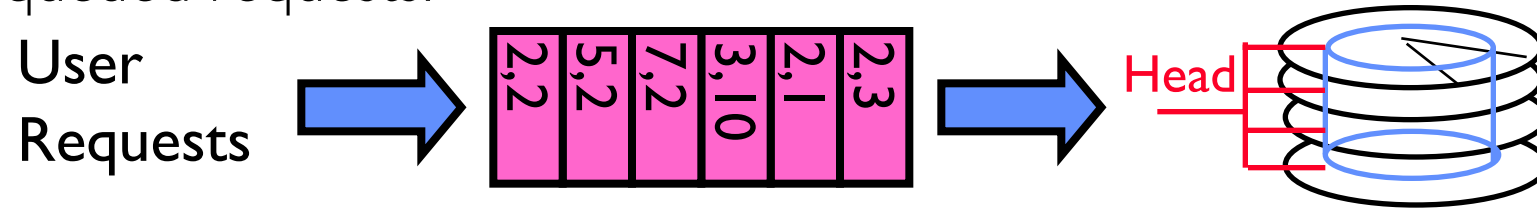


- FIFO Order
 - Fair among requesters, but order of arrival may be to random spots on the disk \Rightarrow Very long seeks
- SSTF: Shortest seek time first
 - Pick the request that's closest on the disk
 - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
 - Con: SSTF good at reducing seeks, but may lead to starvation

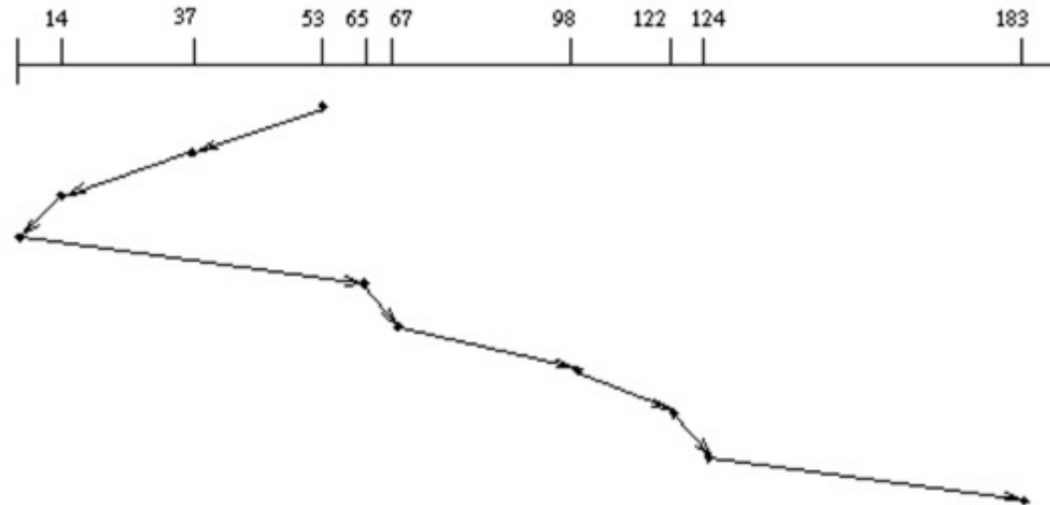


Recap: Disk Scheduling (2/3)

- Disk can do only one request at a time; What order do you choose to do queued requests?

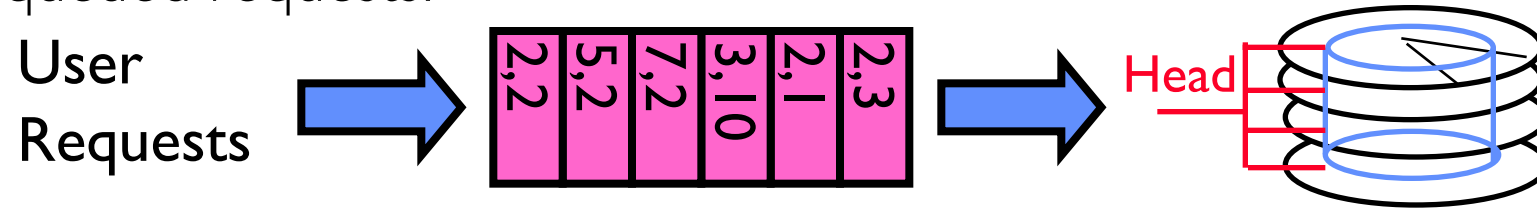


- SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
 - No starvation, but retains flavor of SSTF

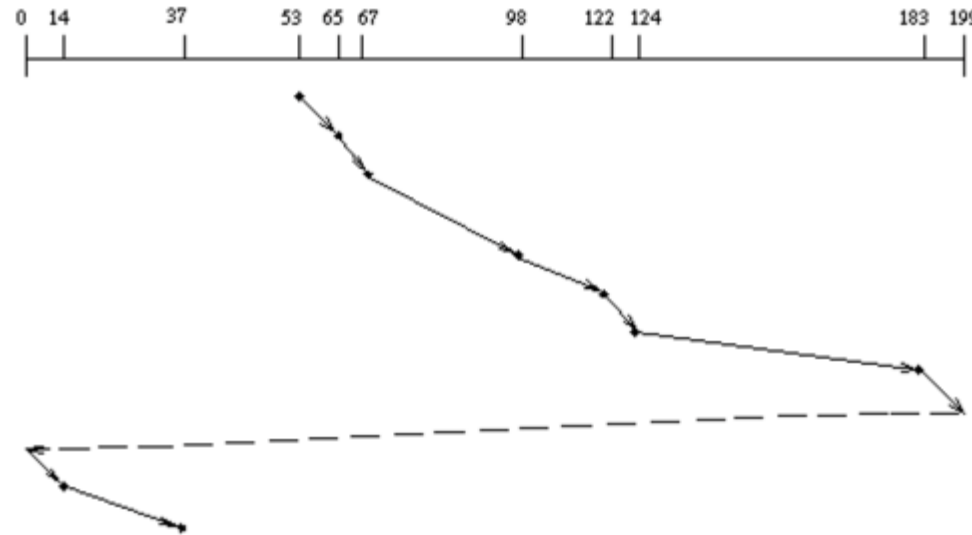


Recap: Disk Scheduling (3/3)

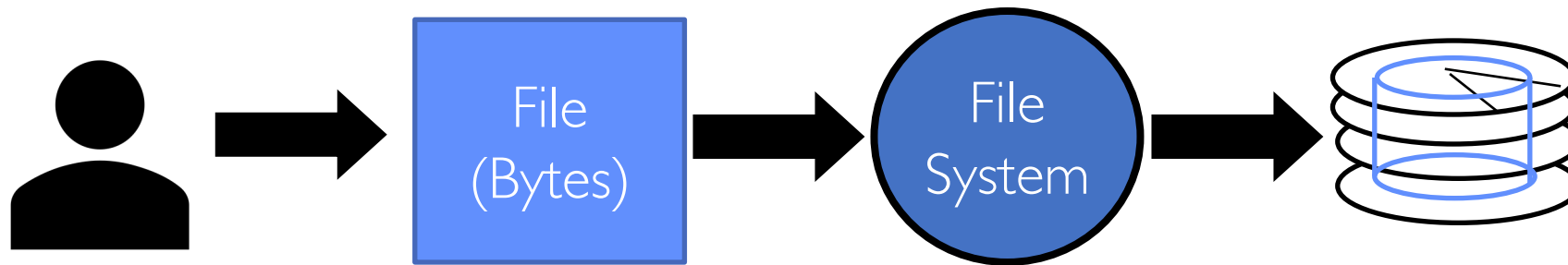
- Disk can do only one request at a time; What order do you choose to do queued requests?



- C-SCAN: Circular-Scan: only goes in one direction
 - Skips any requests on the way back
 - Fairer than SCAN, not biased towards pages in middle



Recap: Translation from User to System View



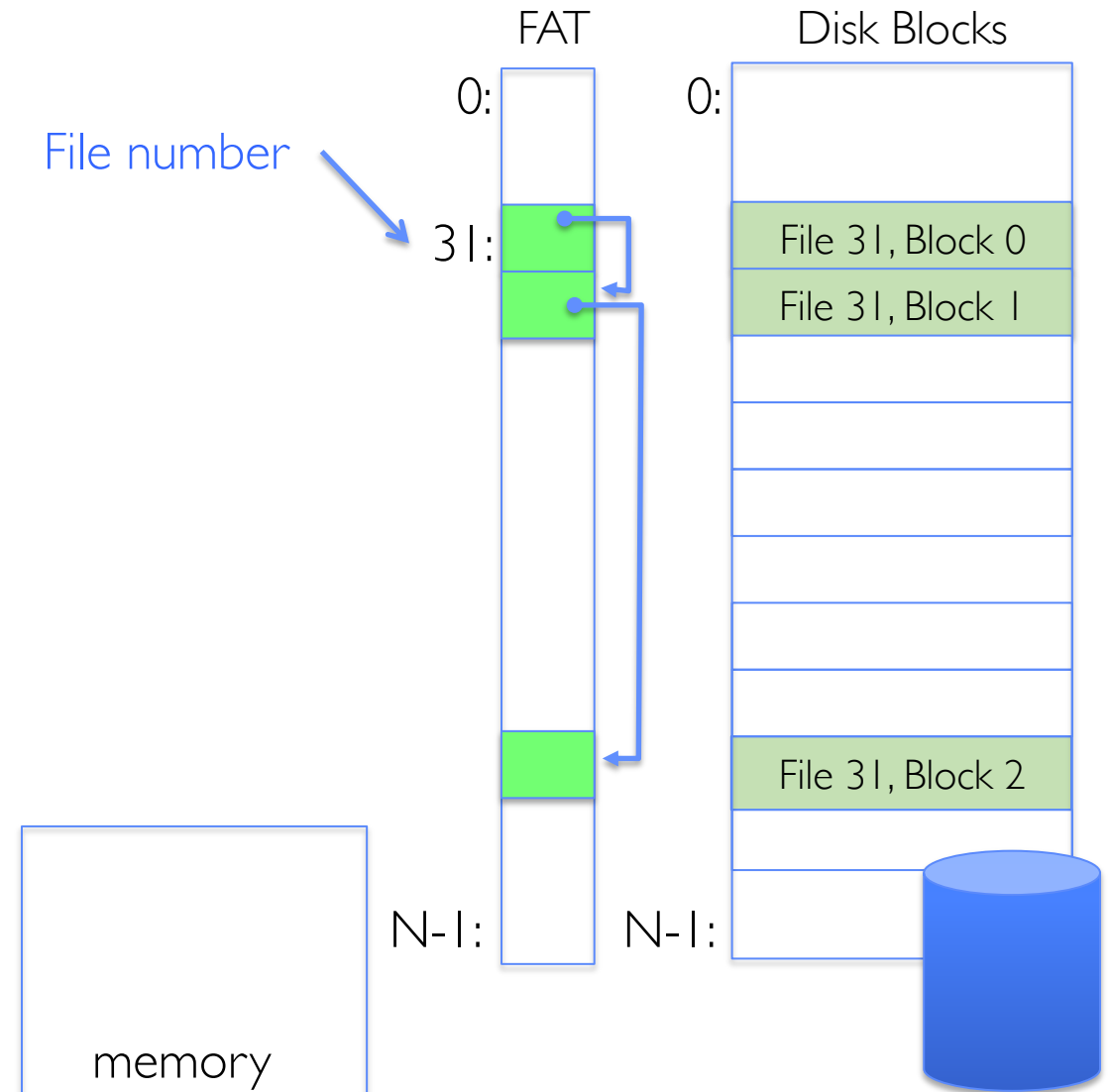
- What happens if user says: “give me bytes 2 – 12?”
 - Fetch block corresponding to those bytes
 - Return just the correct portion of the block
- What about writing bytes 2 – 12?
 - Fetch block, modify relevant portion, write out block
- Everything inside file system is in terms of whole-size blocks
 - Actual disk I/O happens in blocks
 - read/write smaller than block size needs to translate and buffer

CASE STUDY: FAT: FILE ALLOCATION TABLE

- MS-DOS, 1977
- Still widely used!

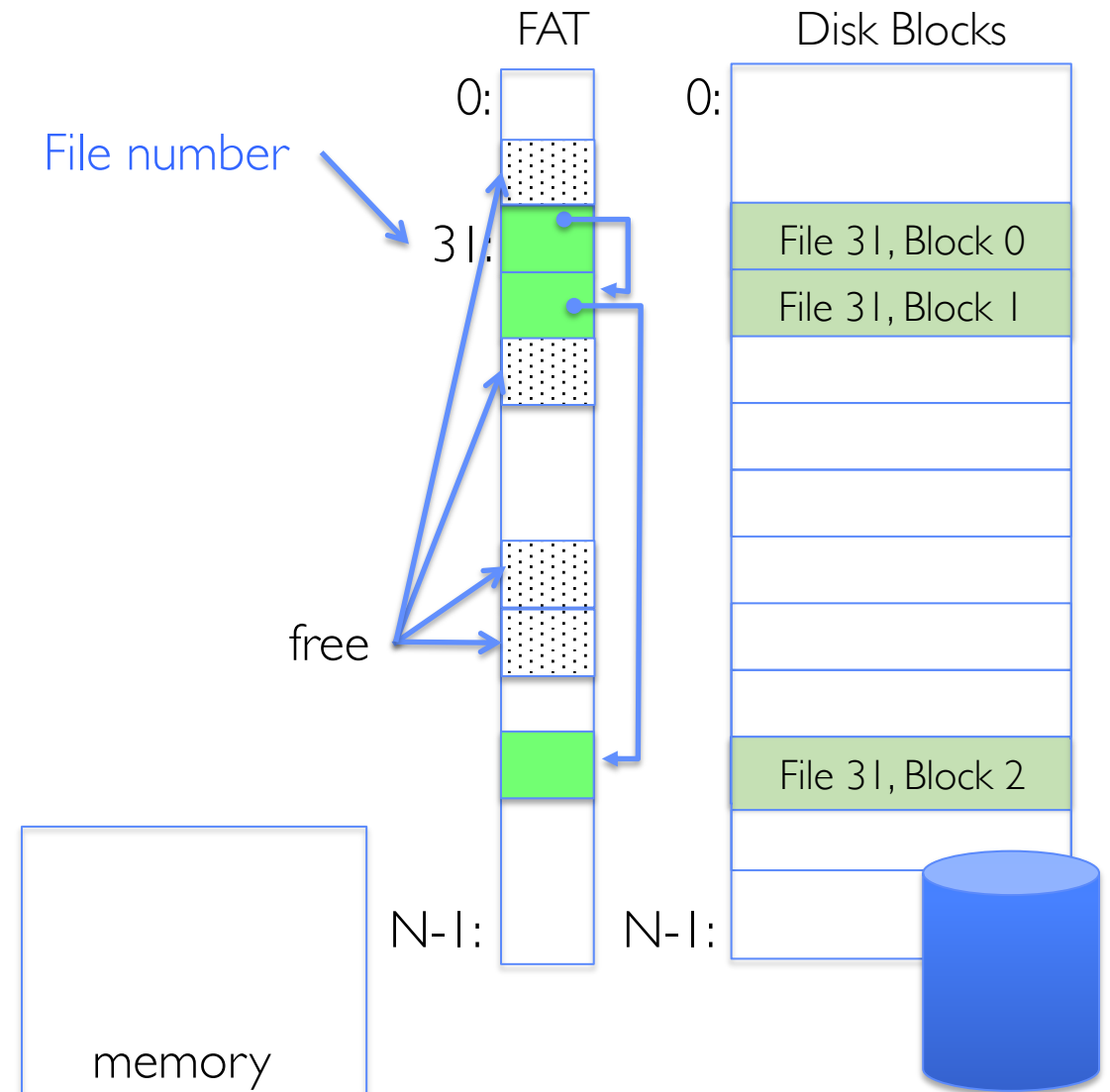
FAT (File Allocation Table)

- 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



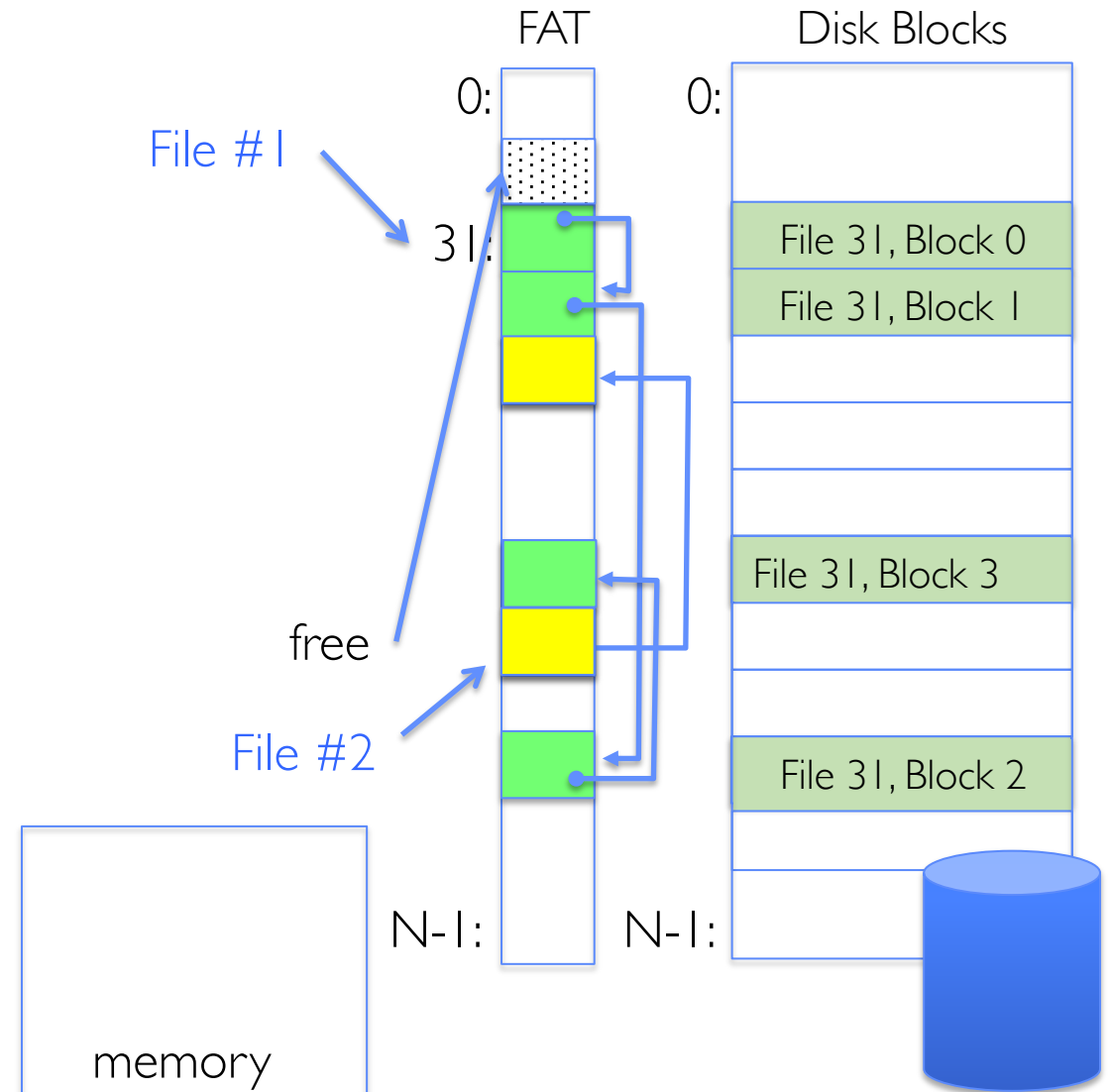
FAT (File Allocation Table)

- File is a collection of disk blocks
- FAT is a linked list 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

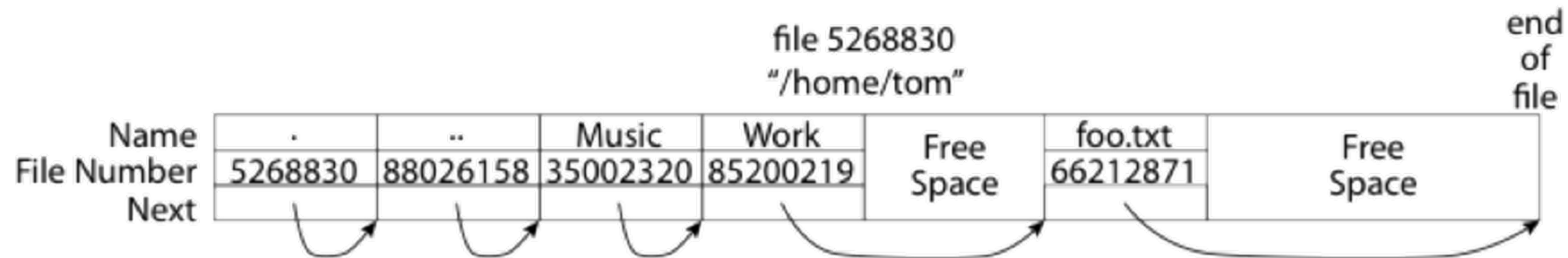


FAT (File Allocation Table)

- 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

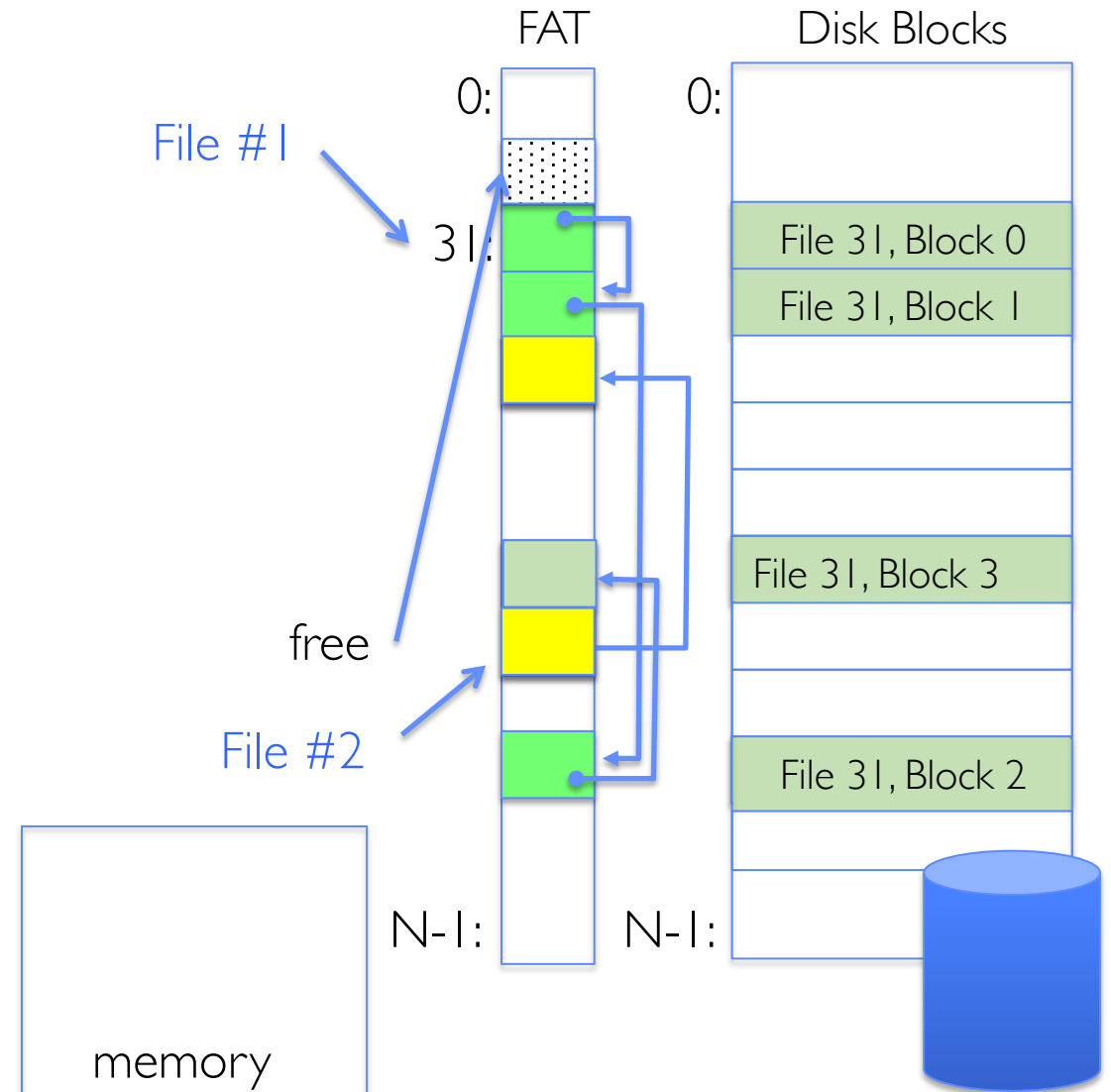


- A directory is a file containing <file_name: file_number> mappings
- Free space for new/deleted entries
- In FAT: file attributes are kept in directory (!!!)
 - Not directly associated with the file itself
- Each directory is a linked list of entries
 - Requires linear search of directory to find particular 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)

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?

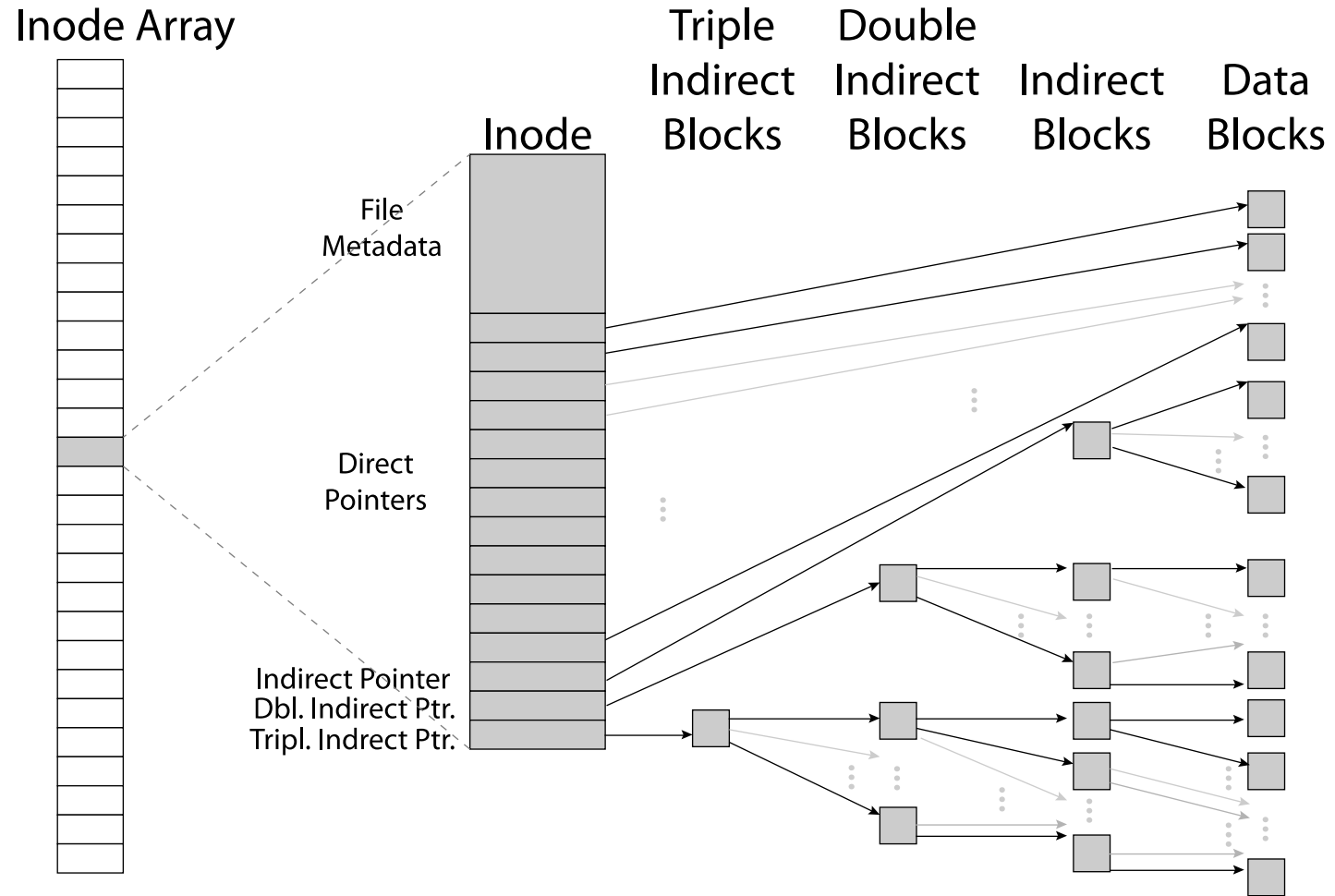


CASE STUDY: UNIX FILE SYSTEM (BERKELEY FFS)

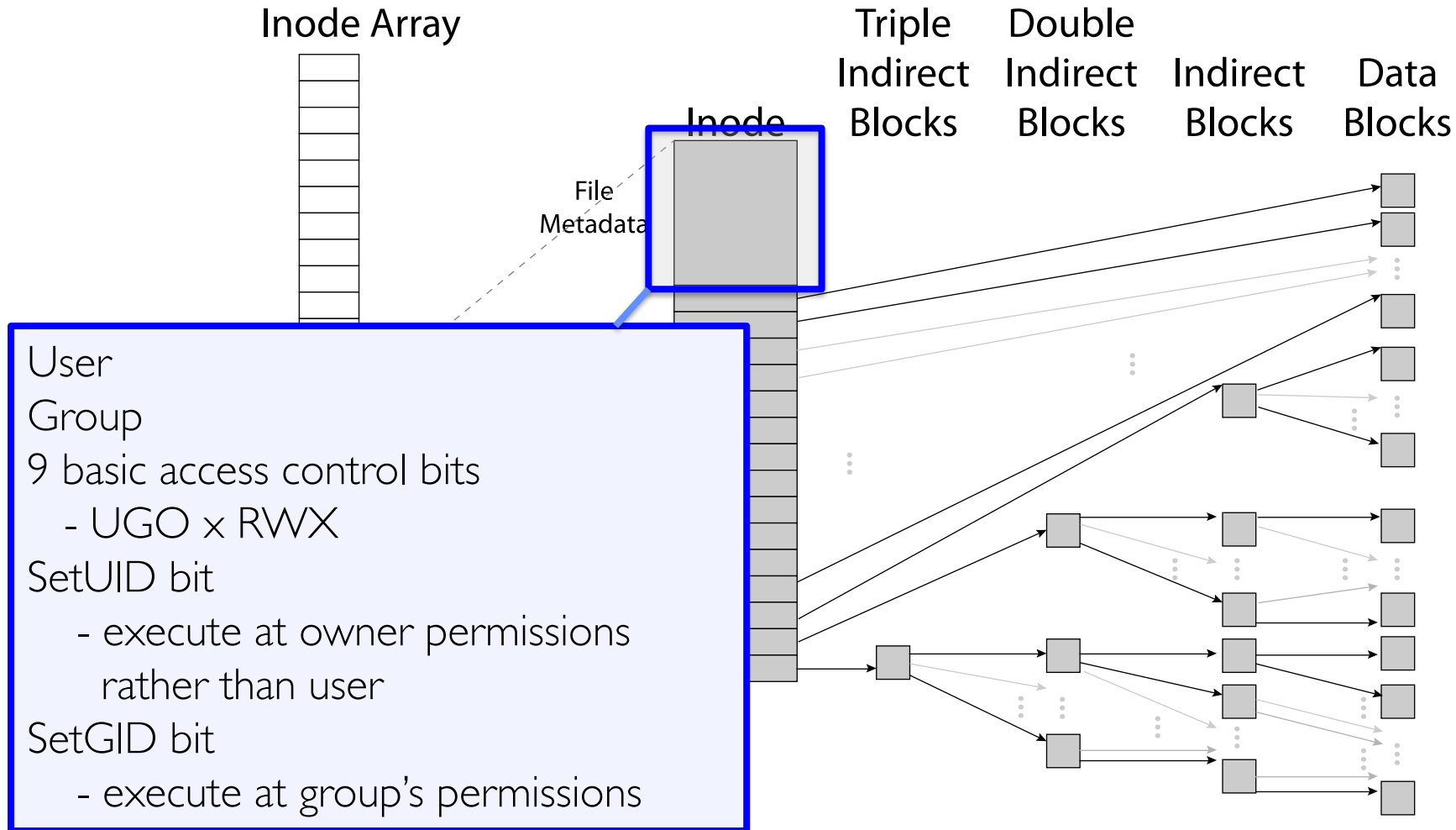
Inodes in Unix (Including Berkeley FFS)

- File Number (inumber) is index into an array of inodes (index structure)
- Each inode corresponds to a file and contains its metadata
 - So, things like read/write permissions are stored with *file*, not in directory (like in FAT)
 - 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!
 - Similar structure for Linux Ext 2/3

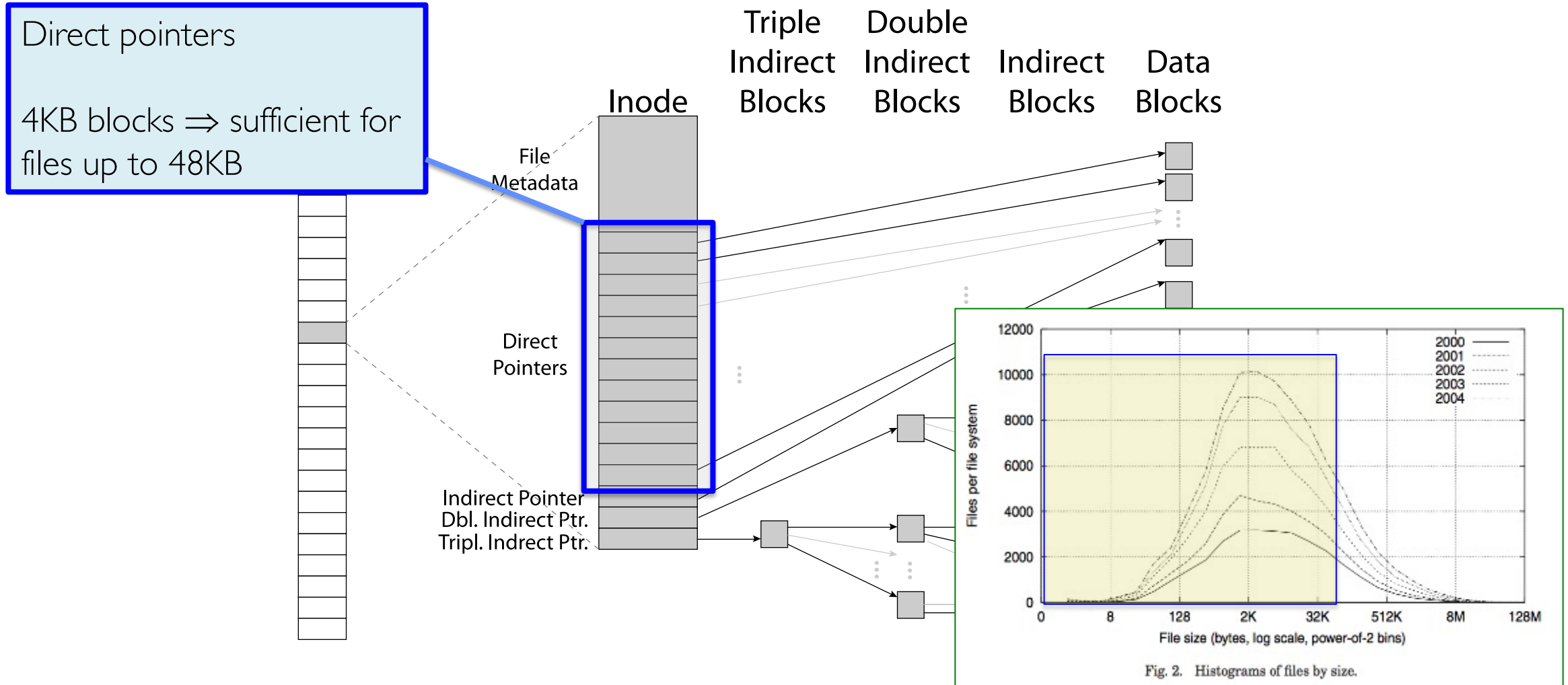
Inode Structure



File Attributes



Small Files: 12 Pointers Direct to Data Blocks



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

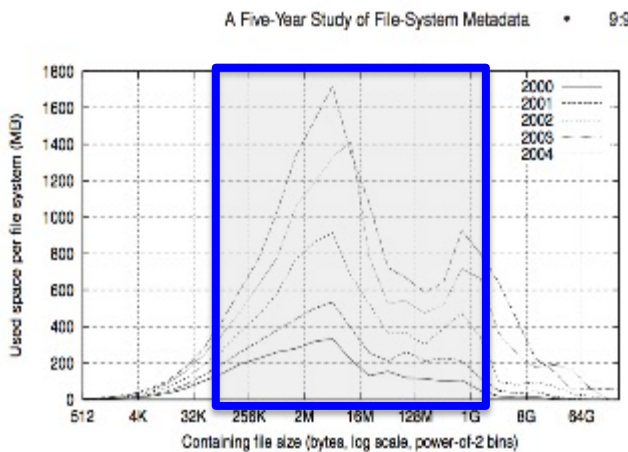
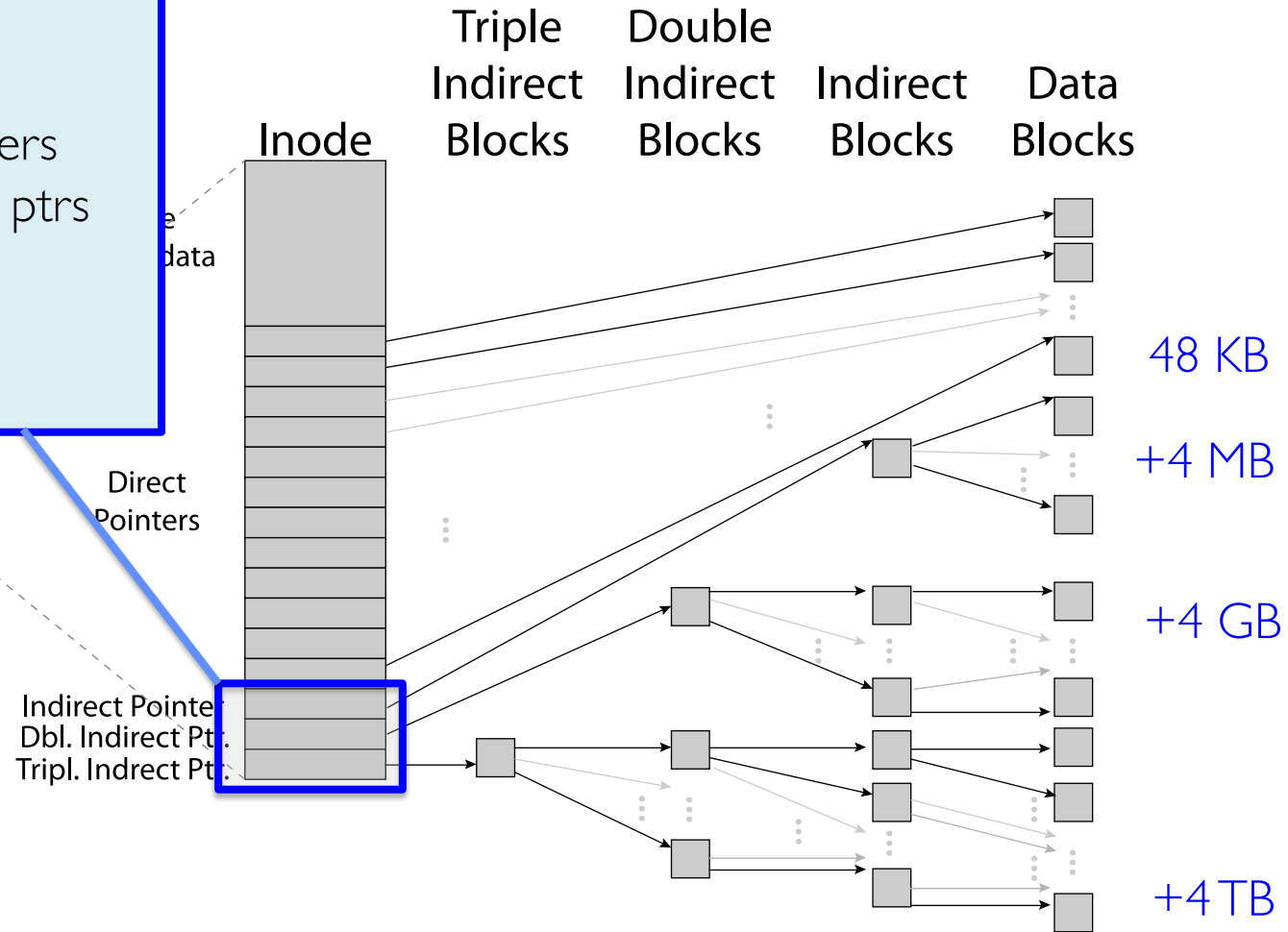
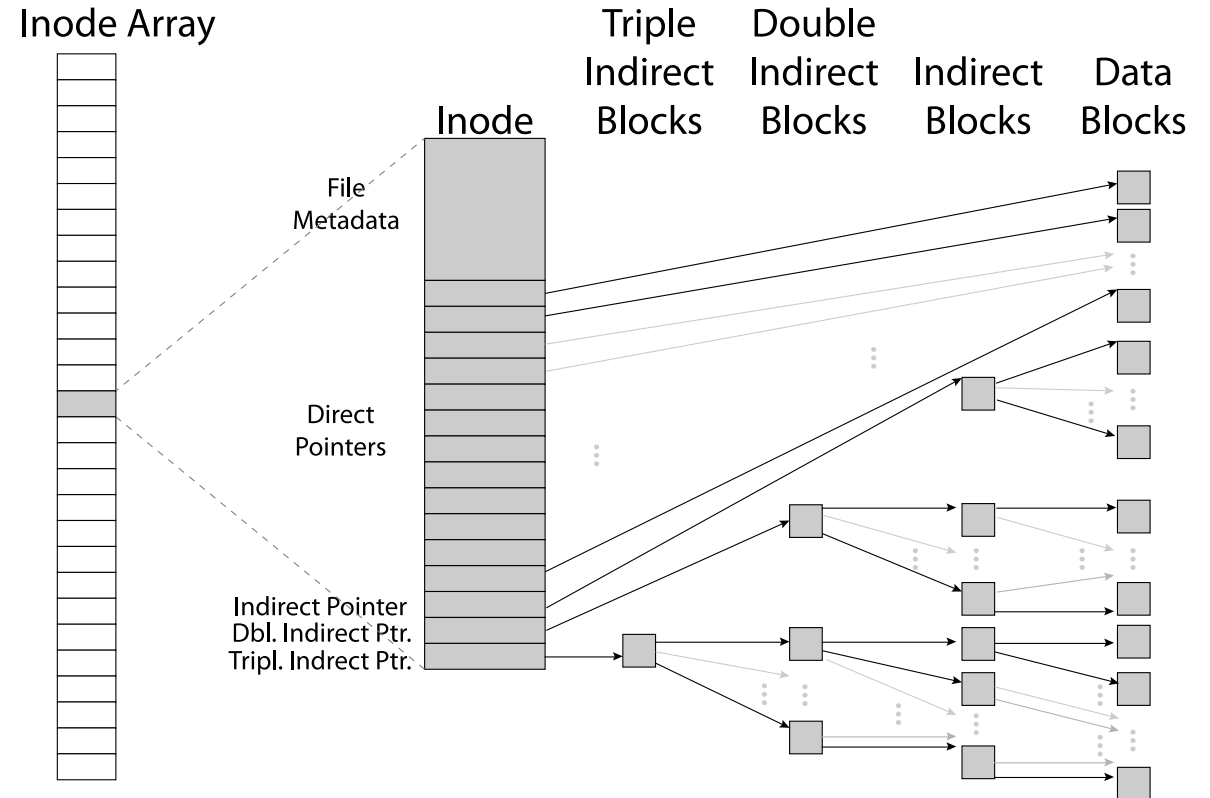


Fig. 4. Histograms of bytes by containing file size.



Putting it All Together: On-Disk Index

- Sample file in multilevel indexed format:
 - 10 direct ptrs, 1KB blocks
 - » 256 indirect blocks
 - » 256^2 double indirect blocks
 - » 256^3 triple indirect blocks
 - 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



CASE STUDY: BERKELEY FAST FILE SYSTEM (FFS)

Fast File System (BSD 4.2, 1984)

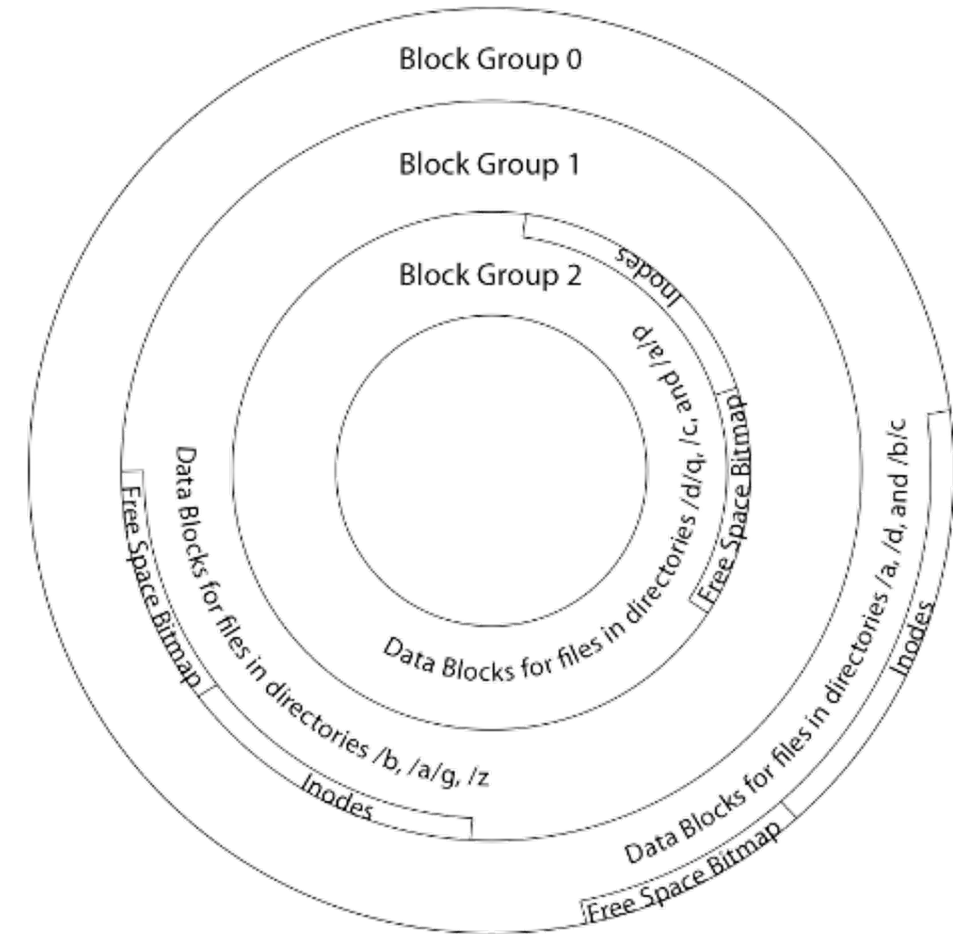
- Same inode structure as in BSD 4.1
 - Same file header and triply indirect blocks like we just studied
 - Some changes to block sizes from 1024 \Rightarrow 4096 bytes for performance
- Paper on FFS: “A Fast File System for UNIX”
 - Marshall McKusick, William Joy, Samuel Leffler and Robert Fabry
- Optimization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

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 to get 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

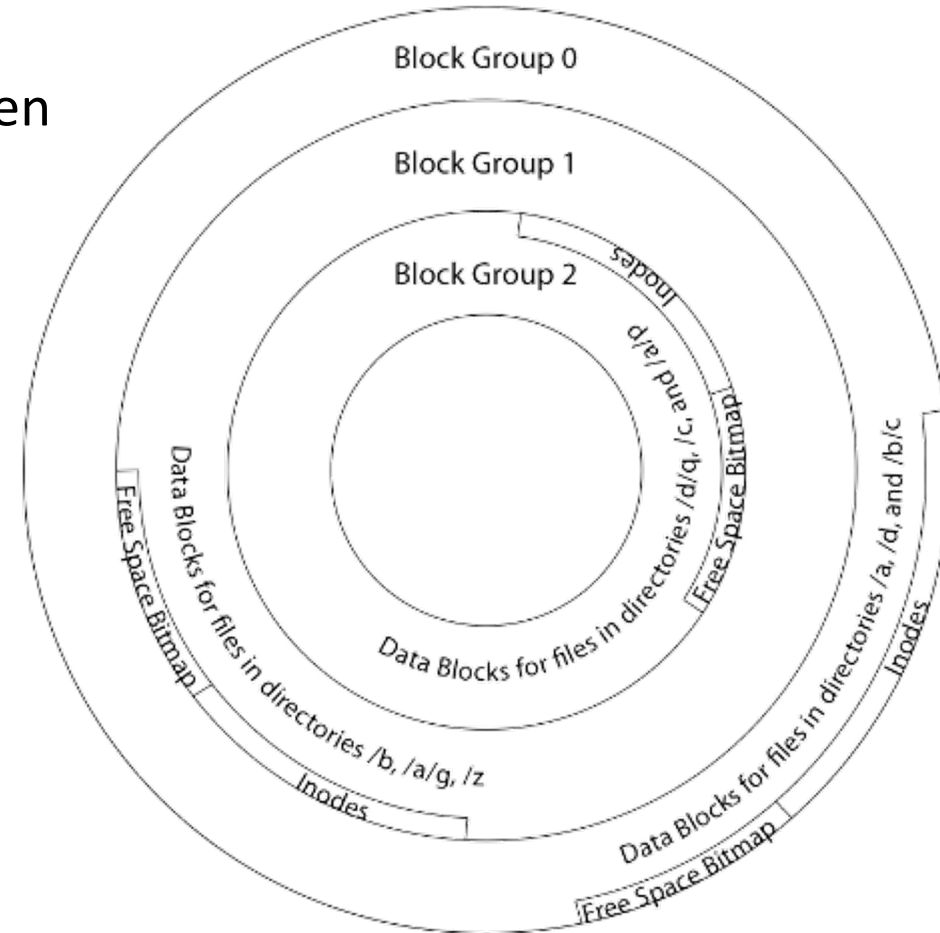
FFS Locality: Block Groups

- 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

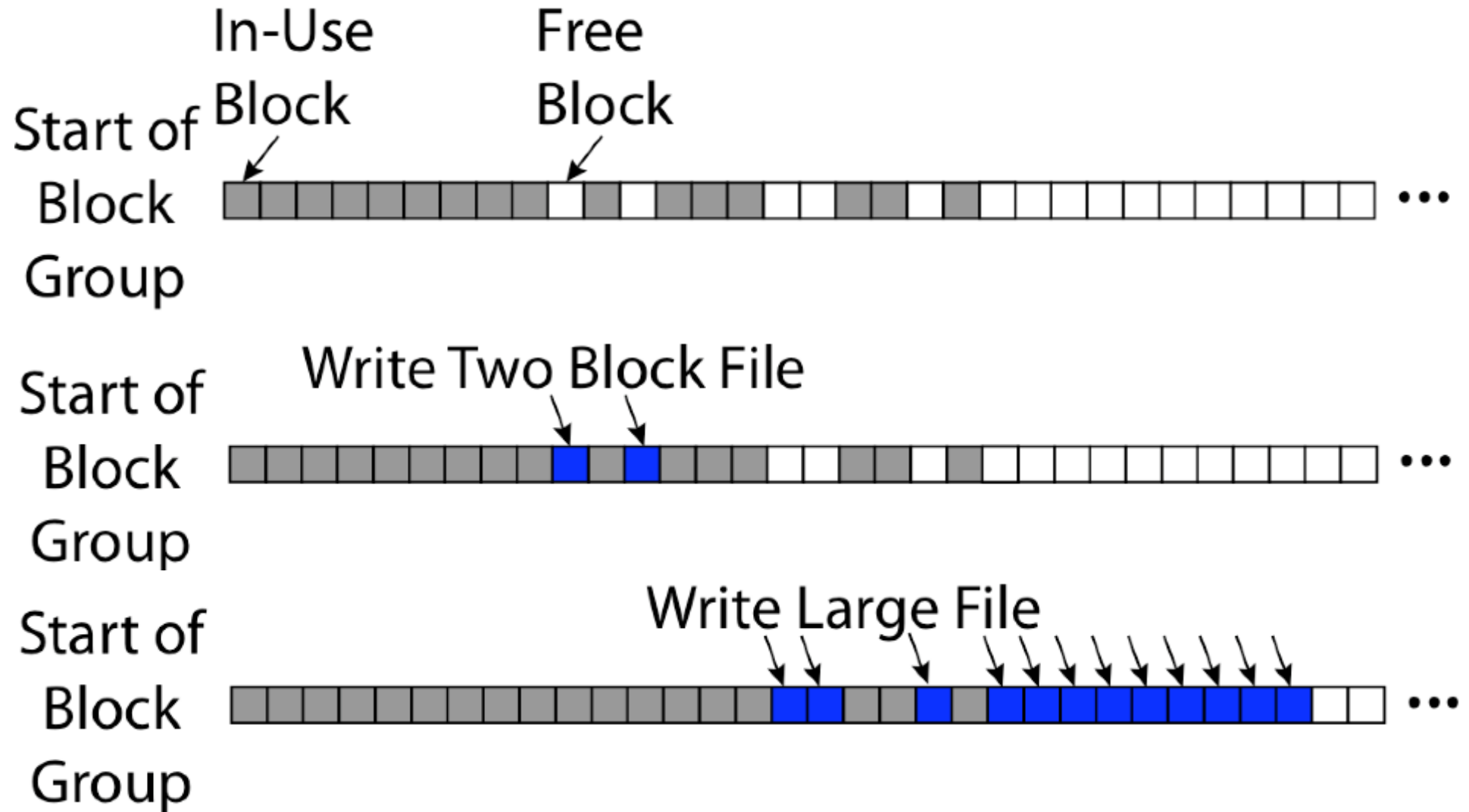


FFS Locality: Block Groups (Con't)

- 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)

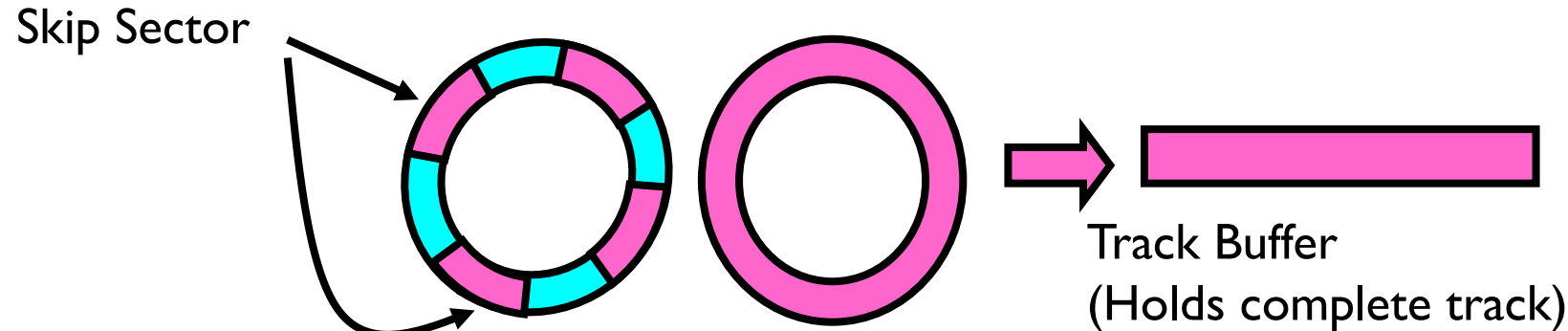


UNIX 4.2 BSD FFS First Fit Block Allocation



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!



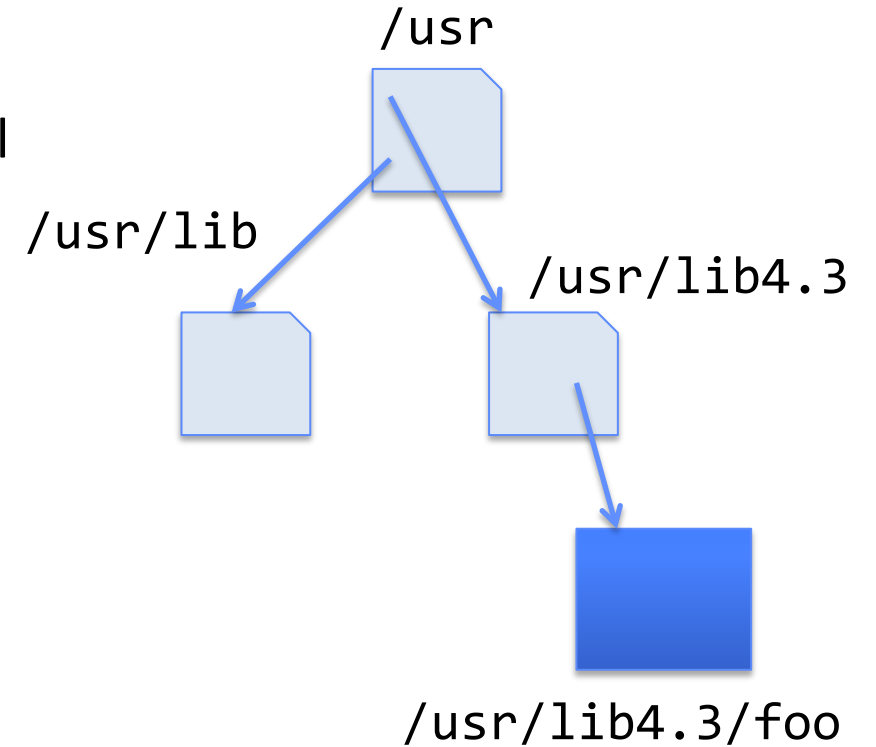
- Solution1: Skip sector positioning (“interleaving”)
 - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
 - » Can be done by OS or in modern drives by the disk controller
- Solution 2: Read ahead: read next block right after first, even if 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

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

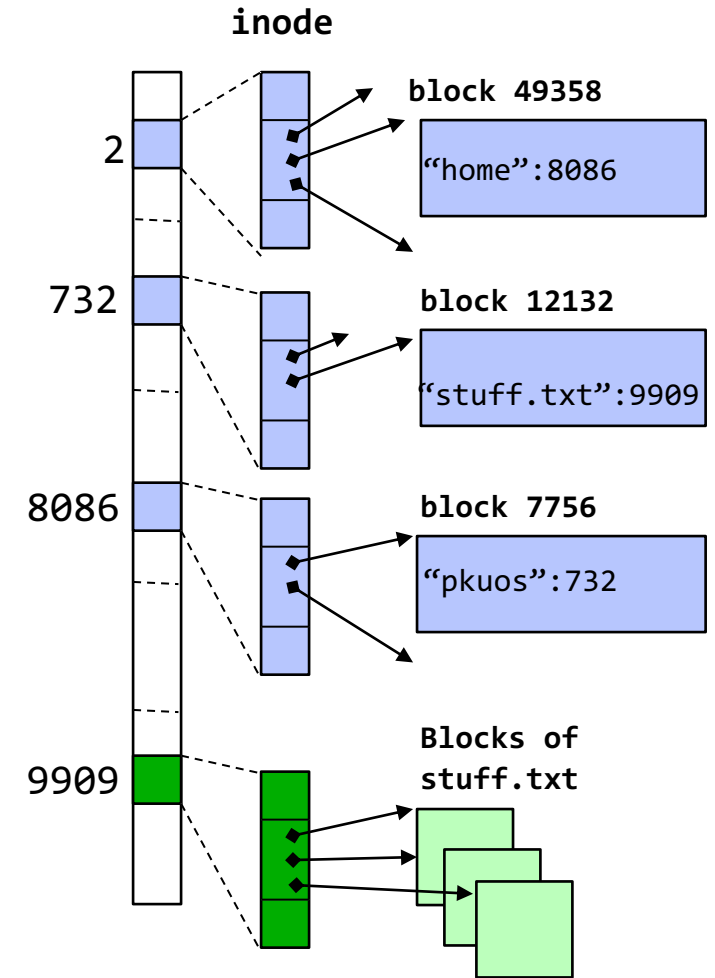


Soft Links (Symbolic Links)

- 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/pkuos/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 “pkuos” to get its inumber (say 732)
- Read inode 732 for /home/pkuos, extract its blocks, read block (say 12132), scan it for “stuff.txt” to get its inumber, say 9909
- Read inode 9909 for /home/pkuos/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...**



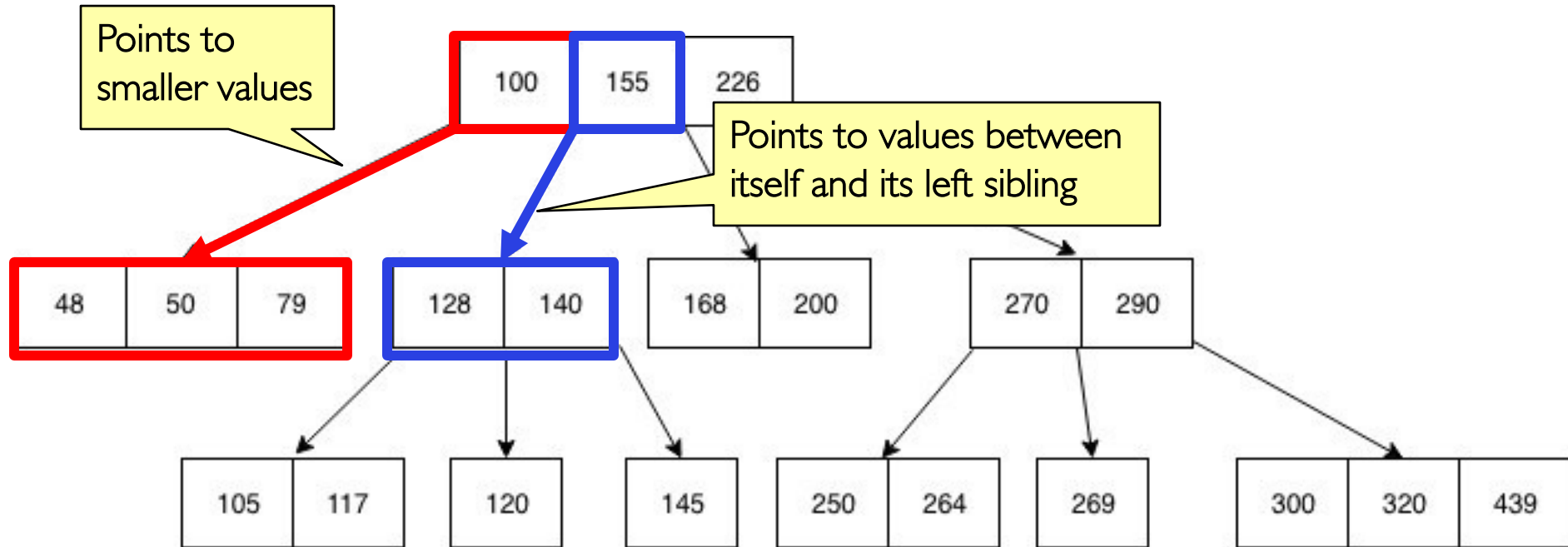
Large Directories

- Early file systems organize directories as:
 - List of <file_name, inode> entries
 - Array of <file_name, inode> entries
- Challenges
 - Linear search: expensive
 - Might need to read entire directory just to find a file: many disk accesses

Large Directories: B-Trees (dirhash)

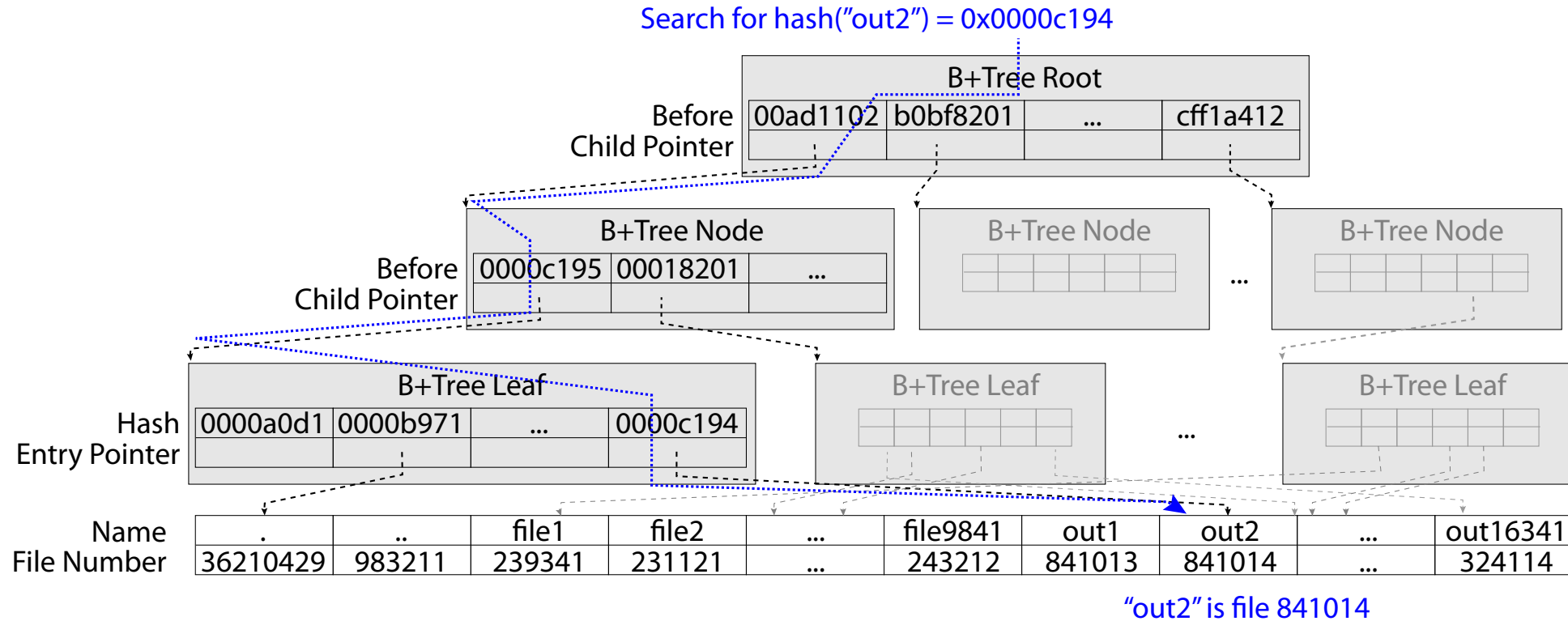
in FreeBSD, NetBSD, OpenBSD

Recall B-Trees data structure



Large Directories: B-Trees (dirhash)

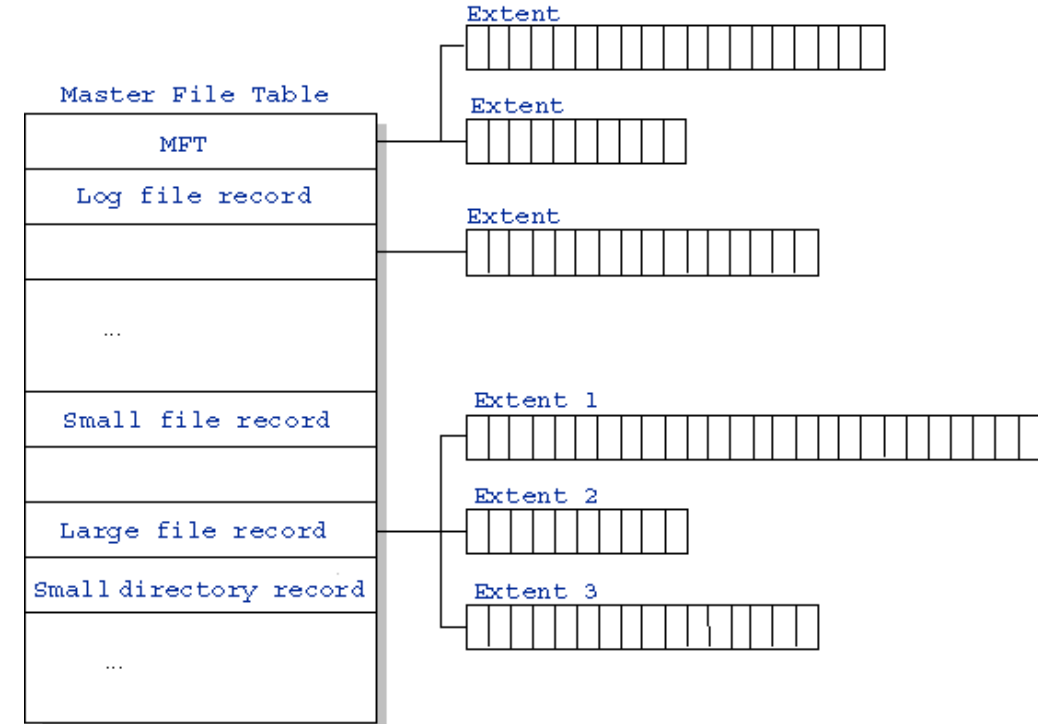
in FreeBSD, NetBSD, OpenBSD



CASE STUDY: WINDOWS NTFS

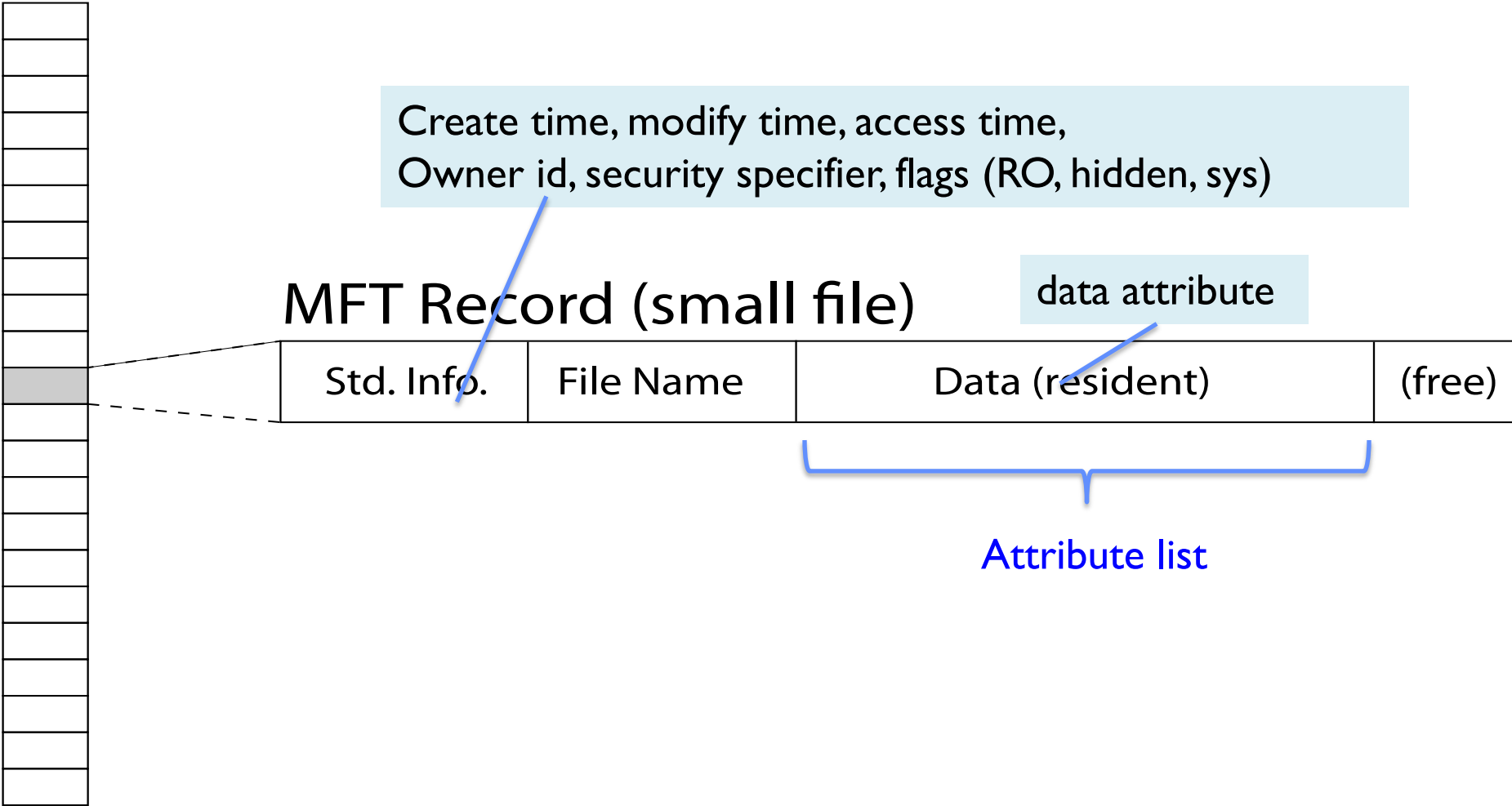
New Technology File System (NTFS)

- 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

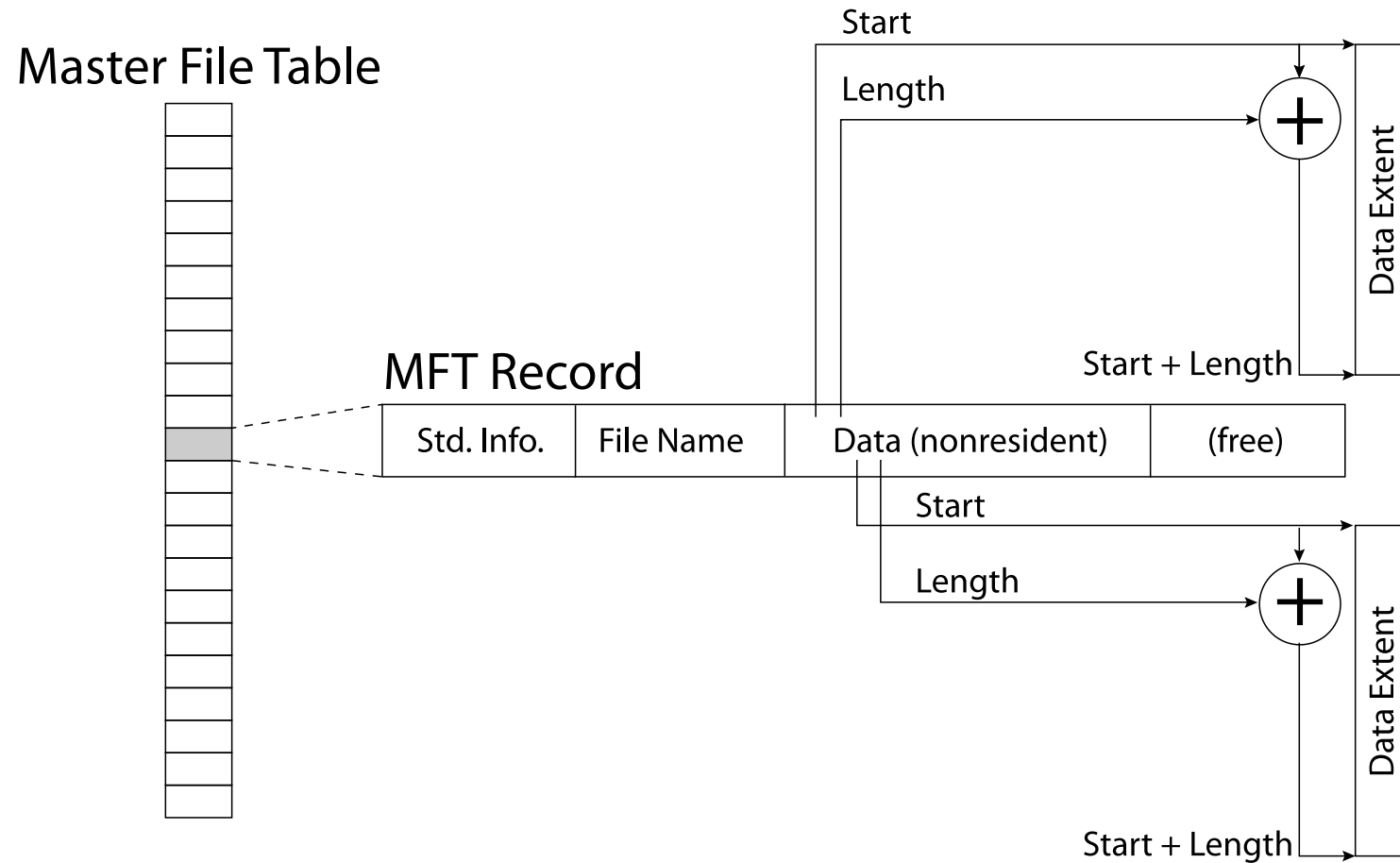


NTFS Small File: Data stored with Metadata

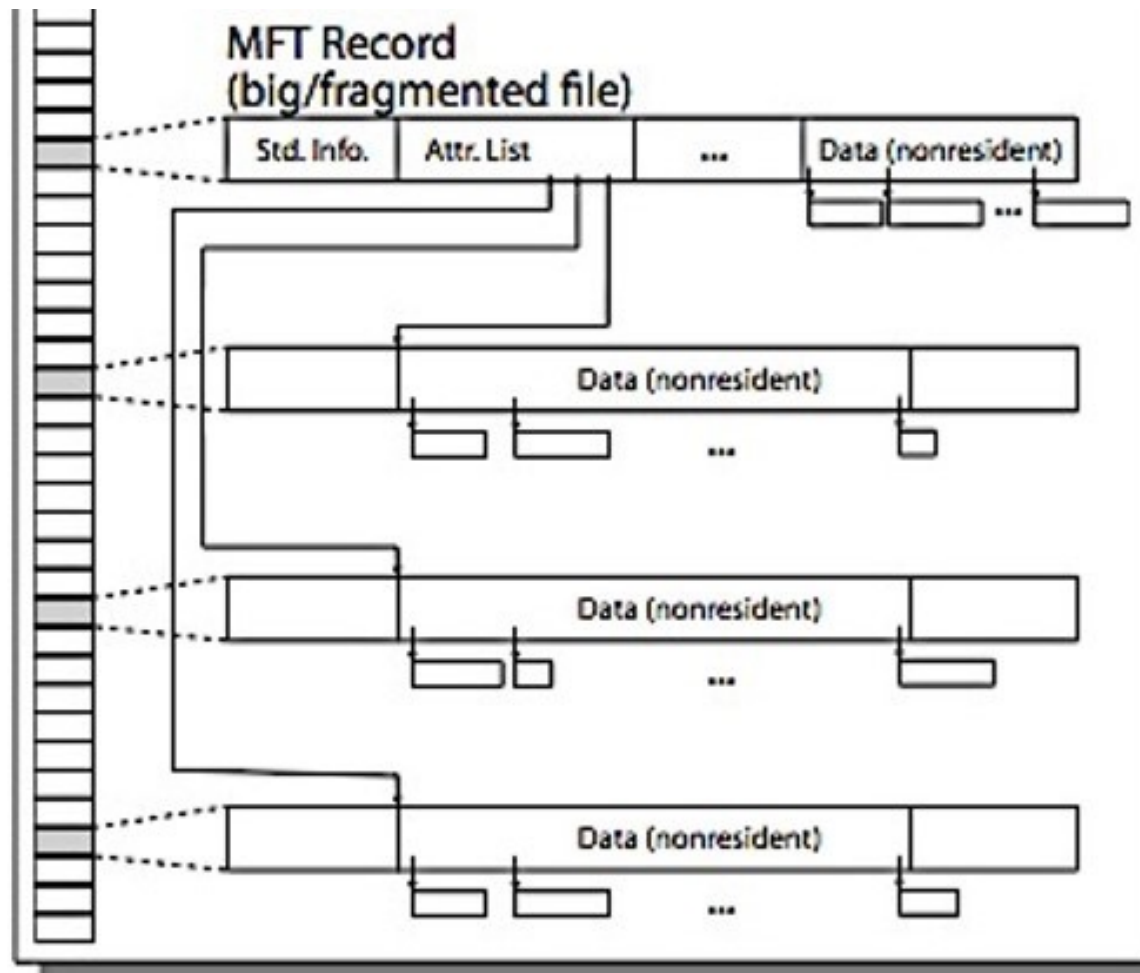
Master File Table



NTFS Medium File: Extents for File Data

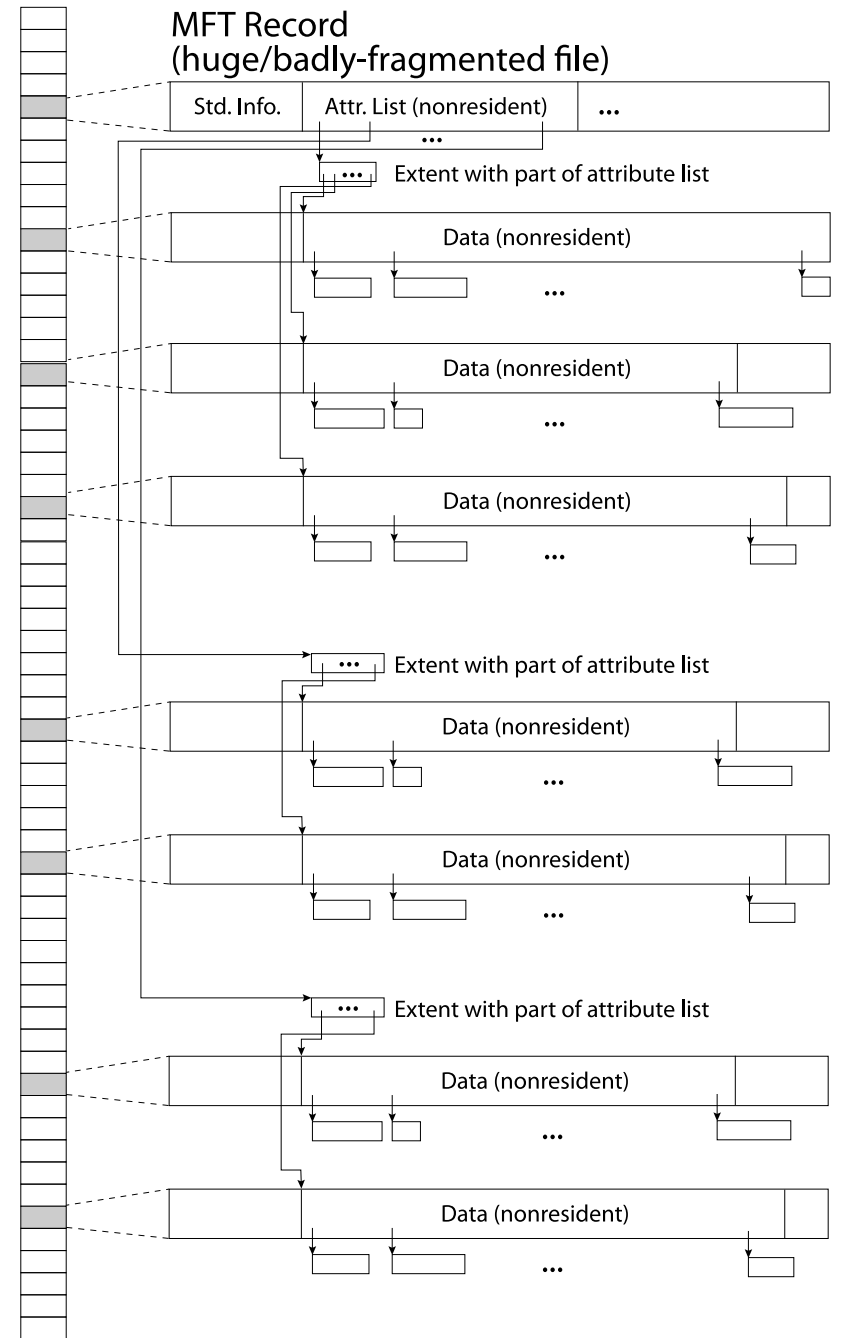


NTFS Large File: Pointers to Other MFT Records



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

File System Summary (1/2)

- File System:
 - Transforms blocks into Files and Directories
 - Optimize for size, access and usage patterns
 - Maximize sequential access, allow efficient random access
 - Projects the OS protection and security regime (UGO vs ACL)
- File defined by header, called “inode”
- 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
- 4.2 BSD Multilevel Indexed Scheme
 - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc.
 - NTFS: variable extents not fixed blocks, tiny files data is in header

File System Summary (2/2)

- File layout driven by freespace management
 - Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization
 - Integrate freespace, inode table, file blocks and dirs into block group