CSE 451: Operating Systems
Winter 2026

Module 17
File Systems

Gary Kimura

High Level View

Application Program (files and directories)

I/O (programming interface)

I/O (File Systems)
block I/O
LBN (Logical Block Number)

I/O (device interface) (blocks of data)

HW Devices

Slow – Fast
Read/Write – Read only – Write only
Byte or block I/O
Polling – Interrupt
DMA

HDD
SSD

Where File Systems Fit In

- The File System interacts with storage by reading/writing blocks (sectors) on a per volume or single disk basis.
- A computer can have multiple volumes each formatted with a different file system. For example. C: might be FAT and D: might be NTFS
- Any interaction between C: and D: is typically above the level of the file system.

Main Points

- Programming Interface
  - Naming. What the typical programmer sees are Files and Directories
  - Basic operations
- On-disk Structure
  - First general design issues and then a look at Microsoft's FAT file system, Unix, and NTFS.
- Journaling and Recovery

File System mission

- The concept of a file system is simple
  - the implementation of the abstraction for secondary storage
    - abstraction = files
  - logical organization of files into directories
    - the directory hierarchy
  - sharing of data between processes, people and machines
    - access control, consistency, …
- The discussion on file systems often center around two concepts
  - There is the on-disk structure (i.e., how is the data persistently stored on secondary storage)
  - There is the software component that manages the storage and communicates with the user to store and retrieve data (hopefully without any loss of information)

Files

- A file is a collection of data with some properties
  - contents, size, owner, last read/write time, protection …
- Files may also have types
  - Some understood by file system
    - device, directory, symbolic link
  - Some understood by other parts of OS or by runtime libraries
    - executable, dll, source code, object code, text file, …
- Type can be encoded in the file's name or contents
  - Windows encodes type in name (and contents)
    - .com, .exe, .bat, .dll, .jpg, .mov, .mp3, …
  - Old Mac OS stored the name of the creating program along with the file
  - Unix does both as well
    - in content via magic numbers or initial characters (e.g., #!)

Programming Interface

- The usual APIs plus maybe a few surprises
  - Open, close, read, write, …
- Files and Directories, the object we play with
- Finding and Enumerating entries in a directory
- Watching for changes
- How do we delete a file?
- Renaming or moving files
- Sequential access versus random access. Who remembers the last access point?
- Shared opens and files locks

Basic operations

Unix

- create(name)
- open(name, mode)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)
- rename(old, new)

Windows

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, …)
- WriteFile(handle, …)
- FlushFileBuffers(handle, …)
- SetFilePointer(handle, …)
- CloseHandle(handle, …)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

File access methods

- Some file systems provide different access methods that specify ways the application will access data
  - sequential access
    - read bytes one at a time, in order
  - direct access
    - random access given a block/byte #
  - record access
    - file is array of fixed- or variable-sized records
  - indexed access
    - FS contains an index to a particular field of each record in a file
    - apps can find a file based on value in that record (similar to DB)
- Why do we care about distinguishing sequential from direct access?
  - what might the File System do differently in these cases?

Directories

- Directories provide:
  - a way for users to organize their files
  - a convenient file name space for both user and File System
- Most file systems support multi-level directories
  - naming hierarchies (c:\, c:\DocumentsAndSettings, c:\DocumentsAndSettings\User, …)
- Most file systems support the notion of current directory
  - absolute names: fully-qualified starting from root of File System
    - `C:\>` **`cd c:\Windows\System32`**
  - relative names: specified with respect to current directory
    - `C:\>` **`c:\Windows\System32`** (absolute)
    - `C:\Windows\System32>` **`cd Drivers`**
    - (relative, equivalent to cd **`c:\Windows\System32\Drivers`**)

Directory internals

- A directory is typically just a file that happens to contain special metadata
  - directory = list of (name of file, file attributes)
  - attributes include such things as:
    - size, protection, location on disk, creation time, access time, …
  - the directory list can be unordered (effectively random)
    - when you type "ls" or "dir /on" , the command sorts the results for you.
    - some file systems organize the directory file as a BTree, giving a "natural" ordering
      - What case to use for sort?
      - What about international issues?

Back to some of the more unexpected functions

- Finding and Enumerating entries in a directory
- Watching for changes
- How do we delete a file?
- Renaming or moving files. What if someone else has the file open?
- Relative opens
- Shared opens and files locks
- Tunnelling, version control and files attributes

A deeper look into File Systems

- Design Constraints and options
- On-Disk structure

File System Design Constraints

- For small files:
  - Small blocks for storage efficiency
  - Files used together should be stored together (is this a holdover from HDD?)
- For large files:
  - Contiguous allocation for sequential access (another holdover?)
  - Efficient lookup for random access
- May not know at file creation
  - Whether file will become small or large

File System Design

- Data structures
  - Directories: file name -> file metadata
    - Store directories as files
  - File metadata: how to find file data blocks
  - Free map: list of free disk blocks
- How do we organize these data structures?
  - HDD have non-uniform performance that differs from SSD

Design Challenges

- Where to store the file's data?
  - Most often within a block(s) aka cluster(s)
  - Disk is divided into equal sized blocks, numbered from 0 to N.
- Index structure
  - How do we locate the blocks of a file? Using block numbers
- Index granularity
  - What block size do we use? Often a multiple of the disk sector size
- Free space
  - How do we find unused blocks on disk? Often a bitmap, but other options are available
- Locality
  - How do we preserve spatial locality? An HDD issue, fragmentation is tied into this.
- Reliability
  - What if machine crashes in middle of a file system op?

File System Design Options

	FAT	FFS	NTFS
Index structure	Linked list	Tree (fixed, assym)	Tree (dynamic)
granularity	block*	block*	extent
free space allocation	FAT array	Bitmap (fixed location)	Bitmap (file)
Locality	defragmentation	Block groups + reserve space	Extents Best fit defrag

* Really a collection of one or more of logical blocks, commonly referred to as a cluster.

Microsoft's File Allocation Table (FAT)

- Introduced in DOS in the early 1980's
- Linked list index structure
  - Simple, easy to implement
  - Still widely used (e.g., thumb drives)
- File table:
  - Linear map of all blocks on disk
  - Each file a linked list of blocks
- Allocation granularity (cluster size)

Legacy FAT

Attribute	FAT12	FAT16	FAT32
Used For	Floppies; small hard drives	Small to large hard drives	Large to very large hard drives
Size of Each FAT Entry	12 bits	16 bits	28 bits
Maximum Number of Clusters	~4,096	~65,536	~268,435,456
Supported Cluster Sizes	512 B to 4 KB	2 KB to 32 KB	4 KB to 32 KB
Maximum Volume Size	16,736,256 B (16 MB)	2,147,123,200 B (2 GB)	~2^{41} B (2 TB)

- FAT 8
  - 1977 Bill Gates and Marc McDonald
  - Floppy based

- FAT 12
  - 1980
- FAT 16
  - 1984 with release of PC/AT & MS DOS 3
- FAT 16B
  - 1987 Compaq DOS 3.31
- FAT 16X
  - 1995 PC DOS 7.0/Win 95 – LBA Addressing
- FAT 32
  - 1996 Windows 95 OSR2, 98, ME, MS DOS 7.1 – CHS Addressing
- FAT 32X
  - LBA Addressing

FAT disk layout

- Reserved Area (Boot sector and FAT)
- Root Directory Area
- Data Region

Dirents: contain: Filename, Attributes, Times (creation, last access, write), First cluster of file, Filesize, and a few more things

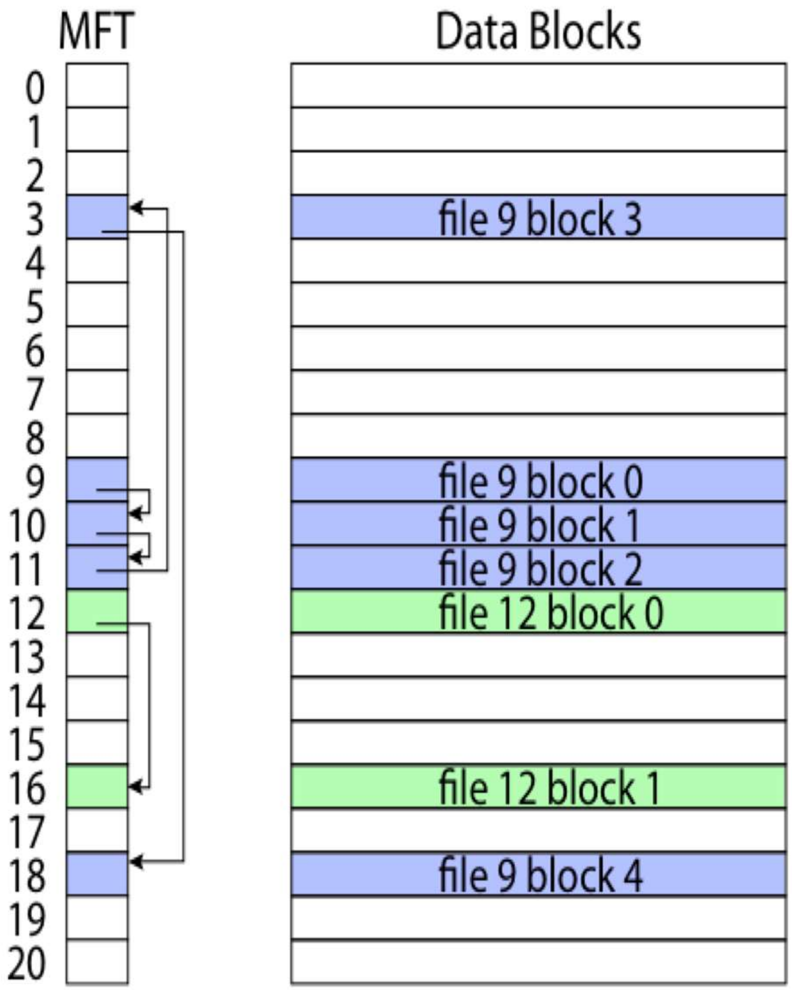
FAT

FAT On-disk structure

FAT File System Structure

FAT32 File System Structure

FAT


MFT
Data Blocks
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
file 9 block 3
file 9 block 0
file 9 block 1
file 9 block 2
file 12 block 0
file 12 block 1
file 9 block 4

FAT

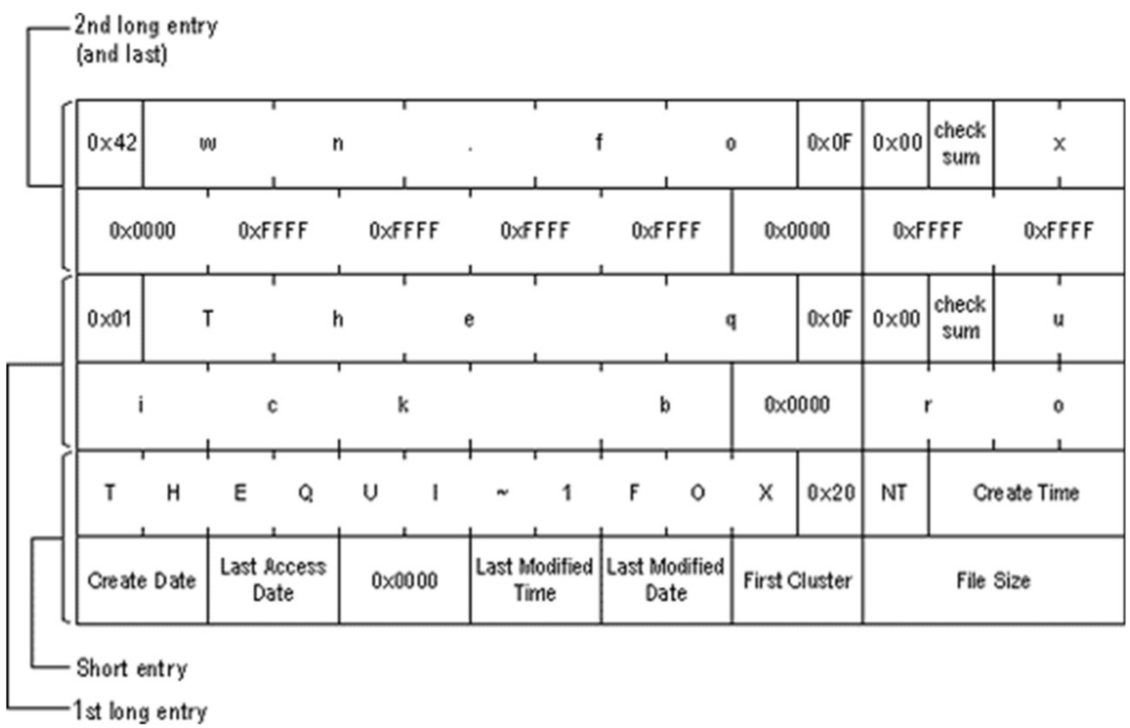
FAT Dirent
8.3 and long filename

Originally only had 8.3 names

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0x00	Name								Extension			Attr	Reserved		Create Time	
0x10	Created Date		Last Accessed Date		**Starting Cluster Hi**		Last Written Time		Last Written Date		**Starting Cluster Low**		**File Size**			

Later extended to Long Filenames with backwards compatibility

(like coughing up a hairball



FAT

- Evolution:
  - Floppy disk and 12-bit FAT
  - Hard drives and 16-bit FAT with subdirectories
  - Larger drives and 32-bit FAT
- Pros:
  - – Easy to find free block
  - – Easy to append to a file
  - – Easy to delete a file
- Cons:
  - – Small file access is slow
  - – Random access is very slow
  - – Fragmentation
    - File blocks for a given file may be scattered
    - Files in the same directory may be scattered
    - Problem becomes worse as disk is used

The original Unix file system

- Dennis Ritchie and Ken Thompson, Bell Labs, 1969
- "UNIX rose from the ashes of a multi-organizational effort in the early 1960s to develop a dependable timesharing operating system" – Multics
- Designed for a "workgroup" sharing a single system
- Did its job exceedingly well
    - Although it has been stretched in many directions and made ugly in the process
- A wonderful study in engineering tradeoffs

All disks are divided into five parts …

- Boot block
  - can boot the system by loading from this block
- Superblock
  - specifies boundaries of next 3 areas, and contains head of freelists of inodes and file blocks
- i-node area
  - contains descriptors (i-nodes) for each file on the disk; all i-nodes are the same size; head of freelist is in the superblock
- File contents area
  - fixed-size blocks; head of freelist is in the superblock
- Swap area
  - holds processes that have been swapped out of memory

So …

- You can attach a disk to a dead system …
- Boot it up …
- Find, create, and modify files …
    - because the superblock is at a fixed place, and it tells you where the i-node area and file contents area are
    - superblock also contains i-node number of root directory

The flat (i-node) file system

- Each file is known by a number, which is the number of the i-node
  - seriously – 0, 1, 2, 3, etc.!
  - why is it called "flat"?
- Files are created empty, and grow when extended through writes

The tree (directory, hierarchical) file system

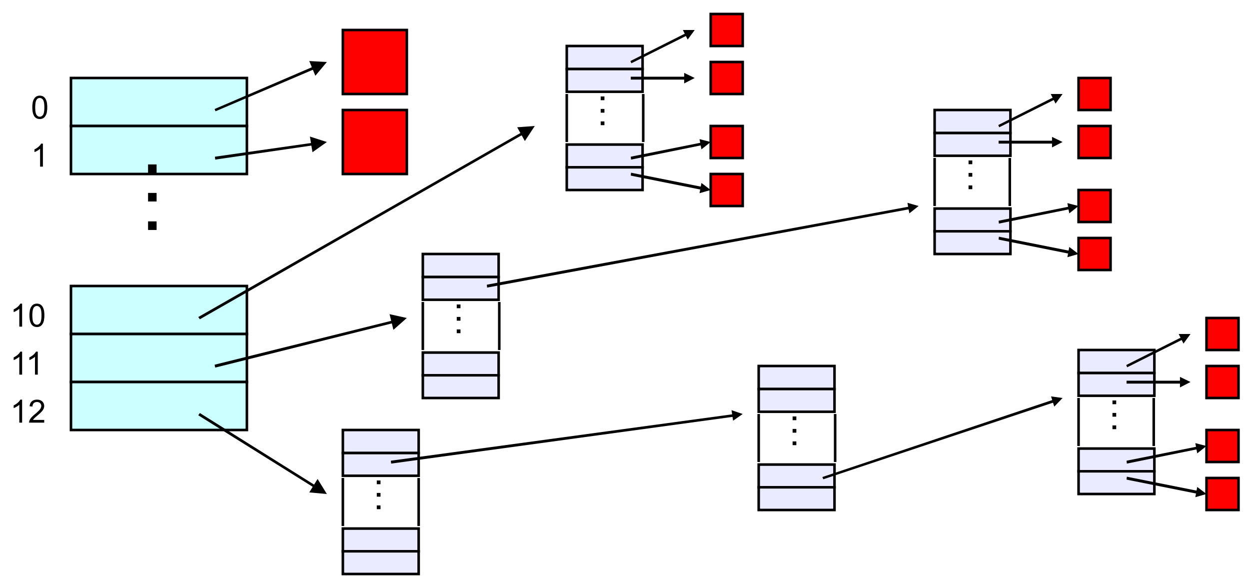
- A directory is a flat file of fixed-size entries
- Each entry consists of an i-node number and a file name

i-node number	File name
152	.
18	..
216	my_file
4	another_file
93	oh_my_god
144	a_directory

- It's as simple as that!

The "block list" portion of the i-node (Unix Version 7)

- Points to blocks in the file contents area
- Must be able to represent very small and very large files. How?
- Each inode contains 13 block pointers
  - first 10 are "direct pointers" (pointers to 512B blocks of file data)
  - then, single, double, and triple indirect pointers



Unix

Protection

- Objects: individual files
- Principals: owner/groups/everyone
- Actions: read/write/execute

- This is pretty simple and rigid, but it has proven to be about what we can handle!

File system consistency

- Both i-nodes and file blocks are cached in memory
- The “sync” command forces memory-resident disk information to be written to disk
  - system does a sync every few seconds
- A crash or power failure between sync’s can leave an inconsistent disk
- You could reduce the frequency of problems by reducing caching or via write-through, but performance would suffer big-time

Consistency of the Flat file system

- Is each block accounted for?
  - Belongs to precisely one file or is on free list
  - What to do if in multiple files?
- Mark-and-sweep garbage collection of disk space
  - Start with bitmap (one bit per block) of zeros
  - For every inode, walk allocation tree setting bits
  - Walk free list setting bits
  - Bits that are one along the way?
  - Bits that are zero at the end?

Consistency of the directory structure

- Verify that directories form a tree
- Start with vector of counters, one per inode, set to zero
- Perform tree walk of directories, adjusting counters on every name reference
- At end, counters must equal link count
  - What do you do when they don't?

Journaling File Systems

- Became popular ~2002, but date to early 80's
- There are several options that differ in their details
  - Ntfs (Windows), Ext3 (Linux), ReiserFS (Linux), XFS (Irix), JFS (Solaris)
- Basic idea
  - update metadata, or all data, *transactionally*
    - *"all or nothing"*
    - *Failure atomicity*
  - if a crash occurs, you may lose a bit of work, but the disk will be in a consistent state
    - more precisely, you will be able to quickly get it to a consistent state by using the transaction log/journal – rather than scanning every disk block and checking sanity conditions

Why are journaling file systems so popular?

- In any file system buffering is necessary for performance
- But suppose a crash occurs during a file creation:
    - Allocate a free inode
    - Point directory entry at the new inode
- In general, after a crash the disk data structures may be in an inconsistent state
    - metadata updated but data not
    - data updated but metadata not
    - either or both partially updated
- fsck (i-check, d-check) are *very* slow
    - must touch every block
    - worse as disks get larger!

Where is the Data?

- In the file systems we have seen already, the data is in two places:
  - On disk
  - In in-memory caches
- The caches are crucial to performance, but also the source of the potential "corruption on crash" problem
- The basic idea of the solution:
  - Always leave "home copy" of data in a consistent state
  - Make updates persistent by writing them to a sequential (chronological) journal partition/file
  - At your leisure, push the updates (in order) to the home copies and reclaim the journal space
  - Or, make sure log is written before updates

• Undo/Redo log

- Log: an append-only file containing log records
  - <start t>
    - transaction t has begun
  - <t,x,v>
    - transaction t has updated block x and its new value is v
      - Can log block "diffs" instead of full blocks
      - Can log *operations* instead of data (operations must be idempotent and undoable)
  - <commit t>
    - transaction t has committed – updates will survive a crash
- Committing involves writing the records – the home data needn't be updated at this time

If a crash occurs

- Open the log and parse
  - <start> <commit> => committed transactions
  - <start> no <commit> => uncommitted transactions
- Redo committed transactions
  - Re-execute updates from all committed transactions
  - Aside: note that update (write) is *idempotent*: can be done any positive number of times with the same result.
- Undo uncommitted transactions
  - Undo updates from all uncommitted transactions
  - Write "compensating log records" to avoid work in case we crash during the undo phase

Managing the Log Space

- A cleaner thread walks the log in order, updating the home locations (on disk, not the cache!) of updates in each transaction
  - Note that idempotence is important here – may crash while cleaning is going on
- Once a transaction has been reflected to the home blocks, it can be deleted from the log

Impact on performance

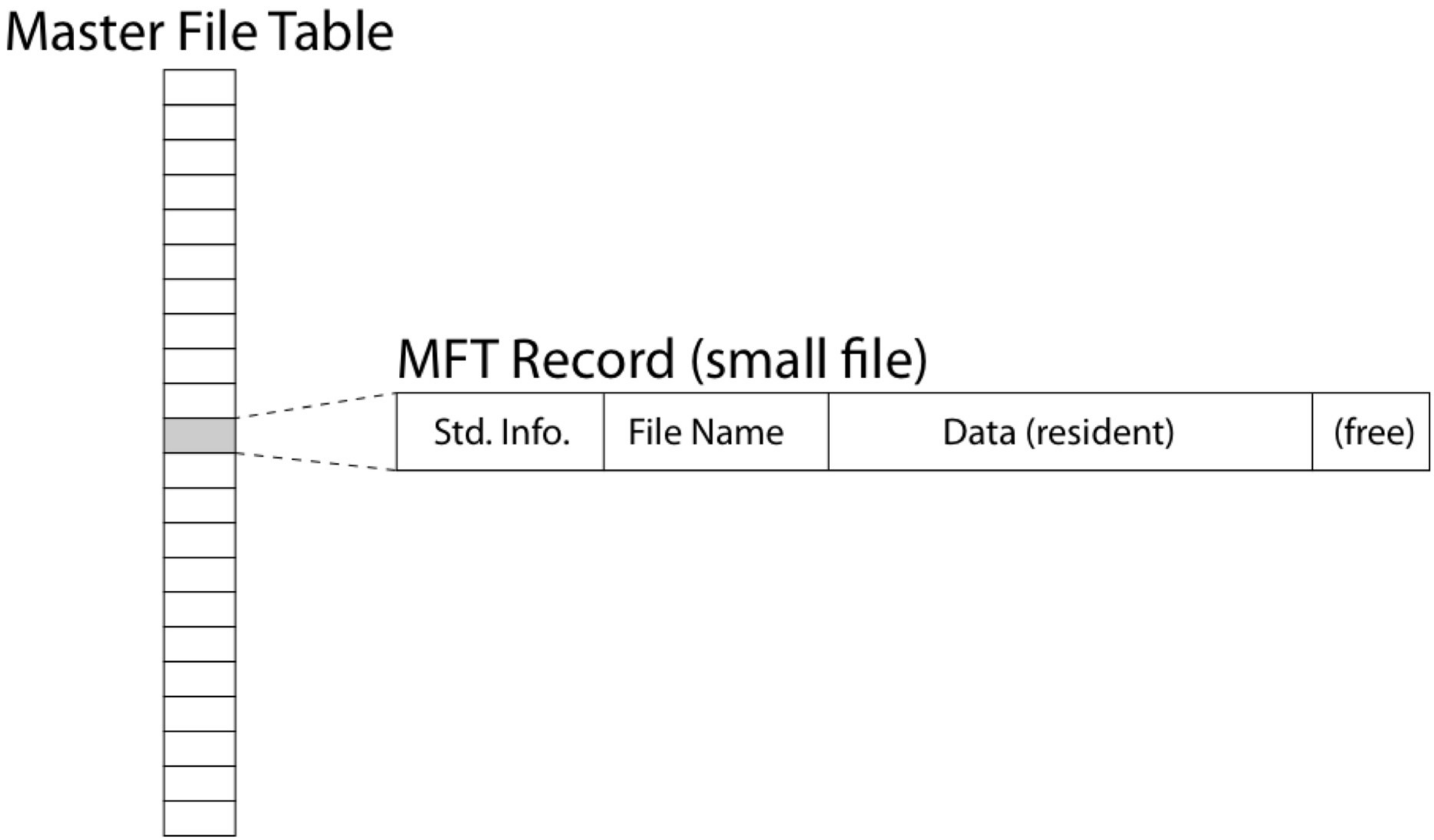
- The log is a big contiguous write
  - very efficient, but it IS another I/O
- And you do fewer scattered synchronous writes
  - very costly in terms of performance
- So journaling file systems can actually improve performance (but not in a busy system!)
- As well as making recovery very efficient

NTFS

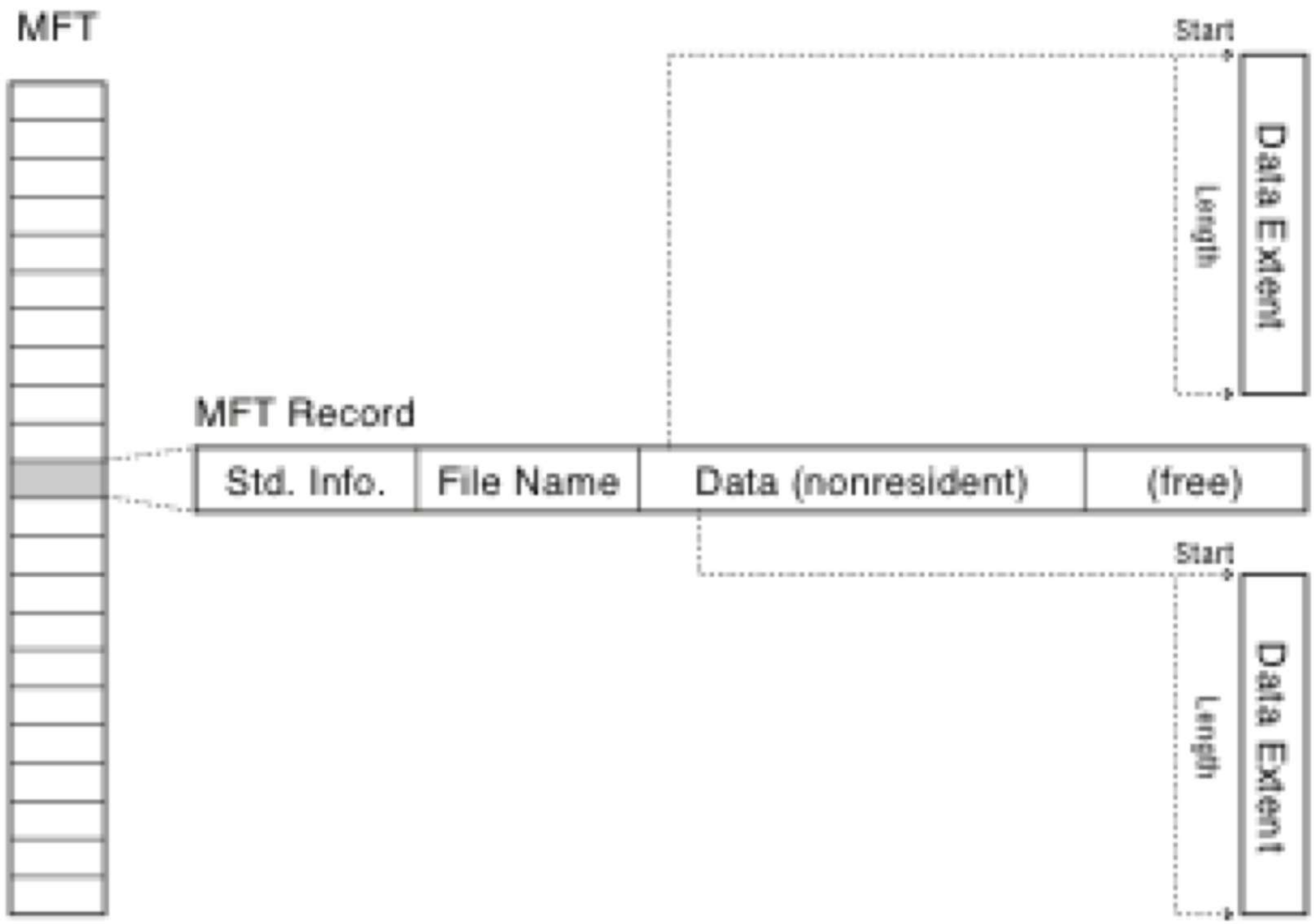
- Developed for Windows NT in the early 1990's
- Master File Table
  - Flexible 1KB storage for metadata and data
- Extents
  - Block pointers cover runs of blocks
  - Similar approach in linux (ext4)
  - File create can provide hint as to size of file
- Journalling for reliability
- A basic underlying design principle: Everything on the disk is represented as a file and accessible through the usual file operations (read, write, etc.)

NTFS disk layout

NTFS Small File

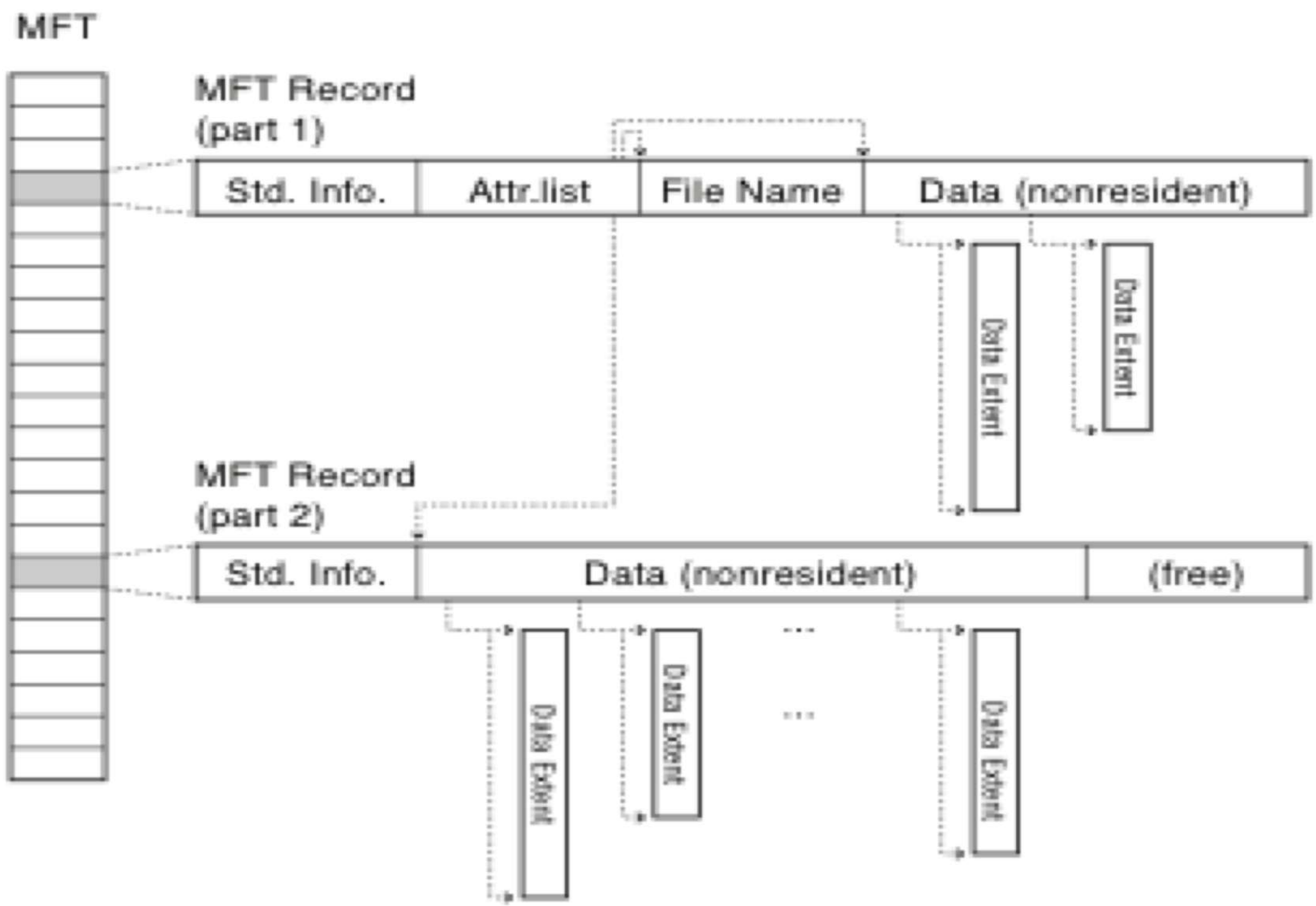


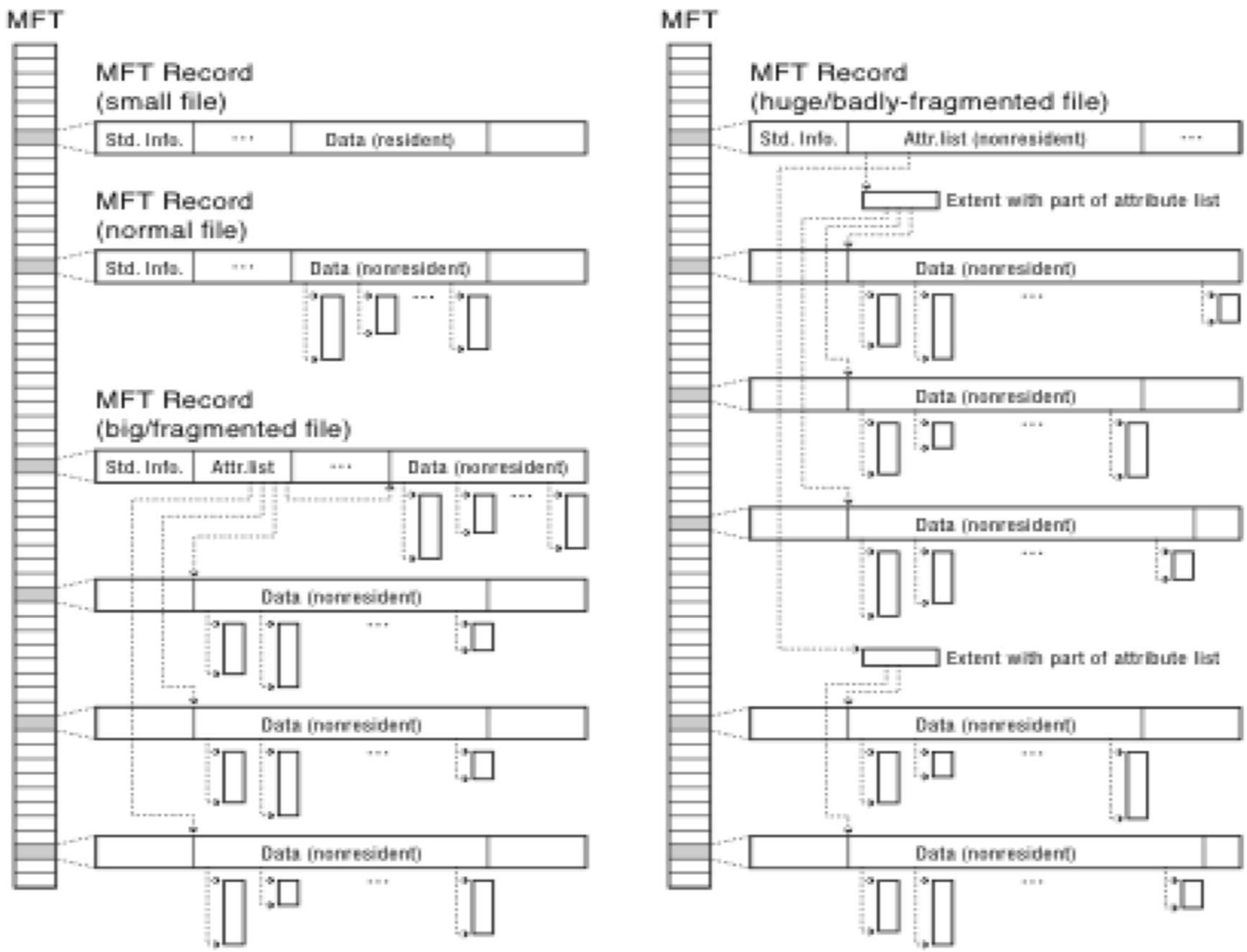
NTFS Medium-Sized File



NTFS

NTFS Indirect Block




MFT
MFT Record
(small file)
Std. Info.
Data (resident)
MFT Record
(normal file)
Std. Info.
Data (nonresident)
MFT Record
(big/fragmented file)
Std. Info.
Attr.list
Data (nonresident)
Data (nonresident)
Data (nonresident)
Data (nonresident)
MFT
MFT Record
(huge/badly-fragmented file)
Std. Info.
Attr.list (nonresident)
Extent with part of attribute list
Data (nonresident)
Data (nonresident)
Data (nonresident)
Extent with part of attribute list
Data (nonresident)
Data (nonresident)

NTFS

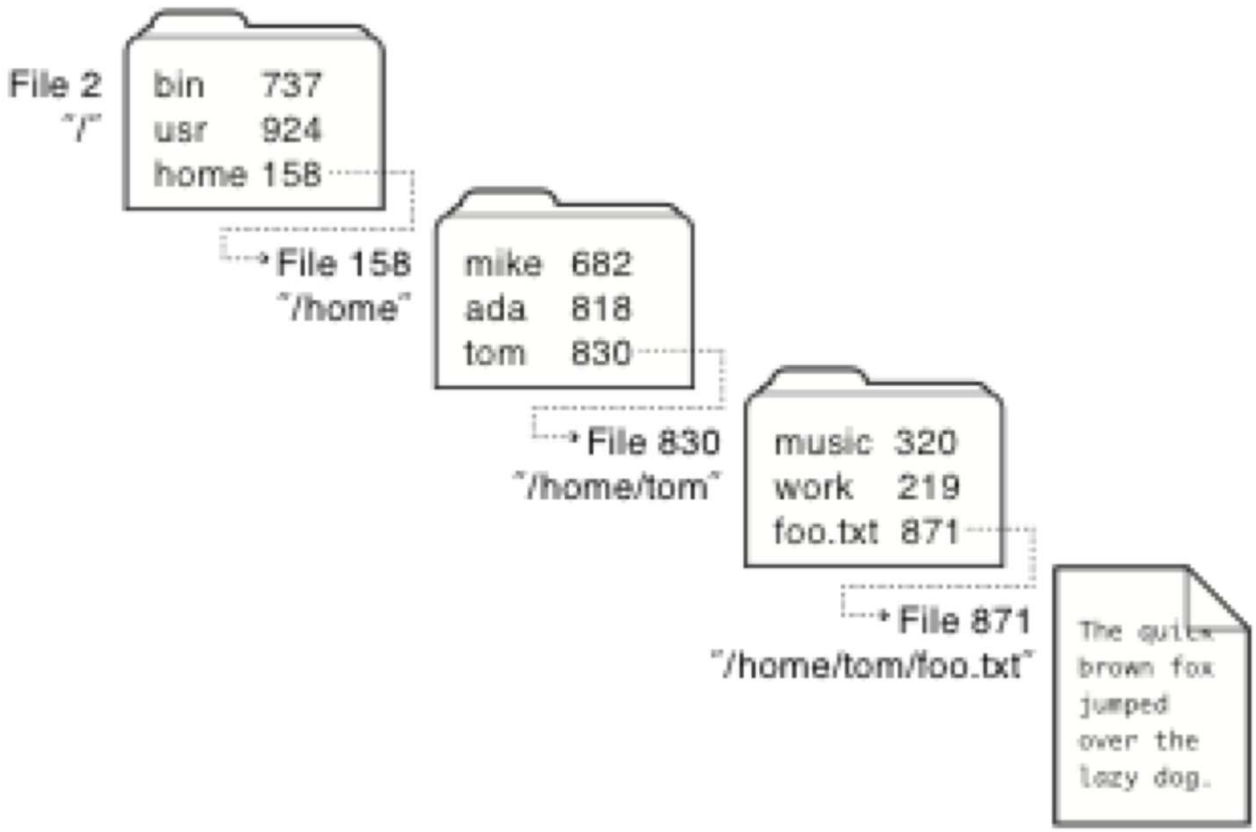
First few MFT Records

$MFT
$MFTMirr
$LogFile
$Volume
$AttrDef
. (root directory)
$Bitmap
$Boot
$BadClus
$Secure
$UpCase

Directories Are Files

NTFS

Recursive Filename Lookup



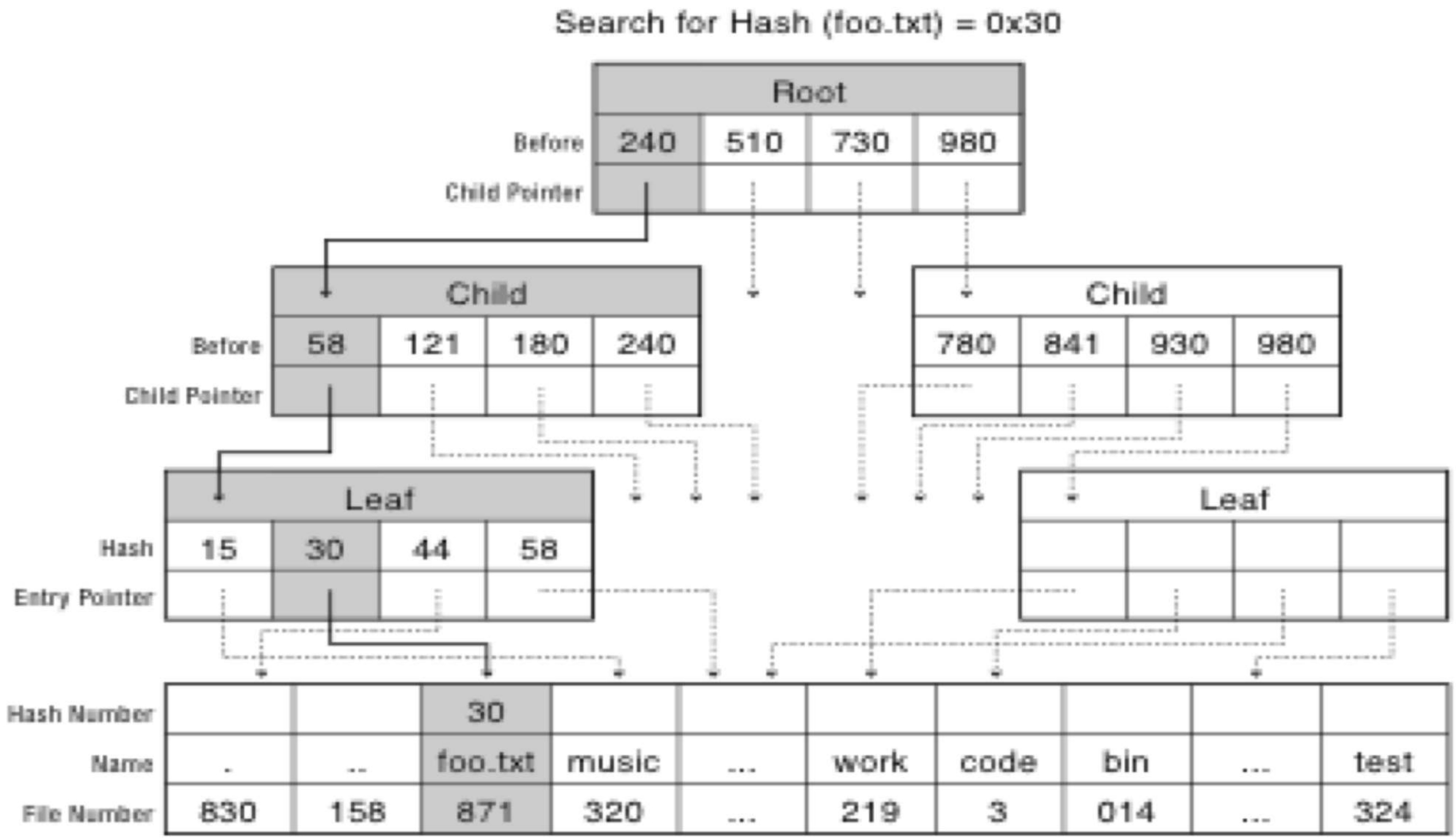
NTFS

Directory Layout

- Directory stored as a file
- Linear search to find filename (small directories)

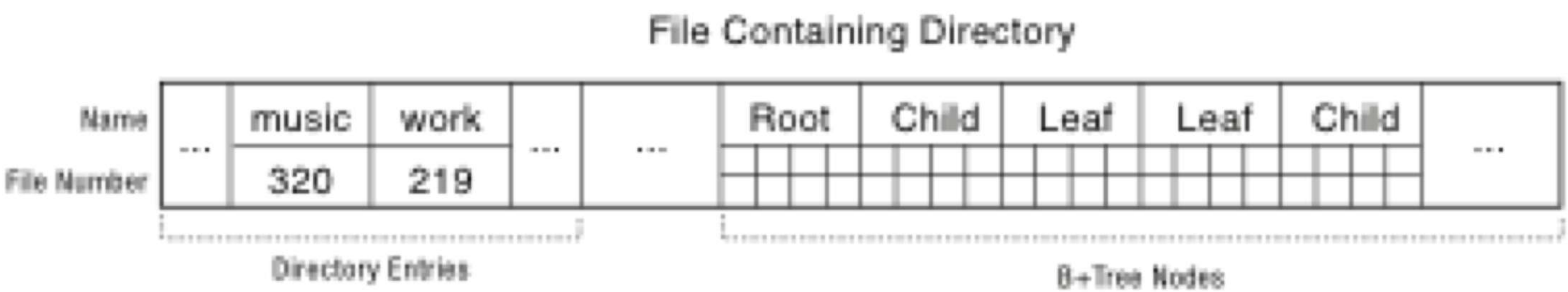
NTFS

Large Directories: B Trees



NTFS

Large Directories: Layout



NTFS

NTFS Features

- Journaling (logging) for quick recovery
- Individual lossless file compression and sparse files
- Symbolic links and hard links
- Unicode Filenames with accompanying collation table
- Random and sequential access
- Able to extend (i.e., add disks) to a volume.
- Fragmentation
- An obscure feature (?) to handle legacy apps that used short 8.3 (Eight Dot Three) names.
- Access Control Lists
- Alternate Data Streams