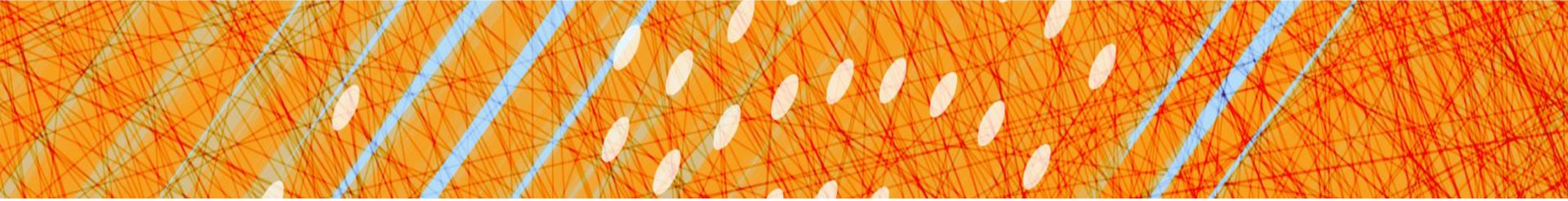
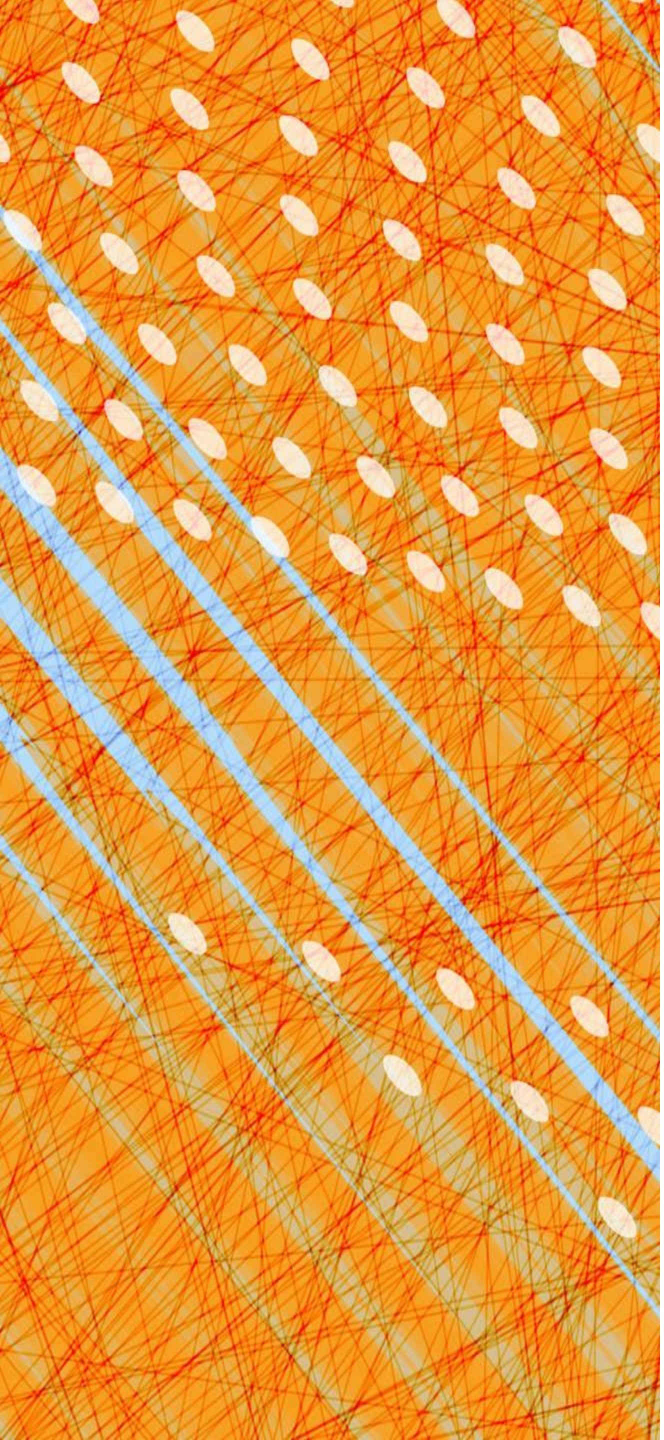


File Systems

CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[R. Agarwal, L. Alvisi, A. Bracy, M. George, E. Sirer, R. Van Renesse]



Tree-Structured Filesystems

Cornell CIS

COMPUTING AND INFORMATION SCIENCE

How is FAT Bad?

- Poor locality
- Many file seeks unless entire FAT in memory: *Example:* 1TB (2^{40} bytes) disk, 4KB (2^{12}) block size, FAT has 256 million (2^{28}) entries (!) 4 bytes per entry ➜ 1GB (2^{30}) of main memory required for FS (a sizeable overhead)

- Poor random access
- Limited metadata
- Limited access control
- Limitations on volume and file size
- No support for reliability techniques

[mid 80’s]

Fast File System (FFS)

UNIX Fast File System

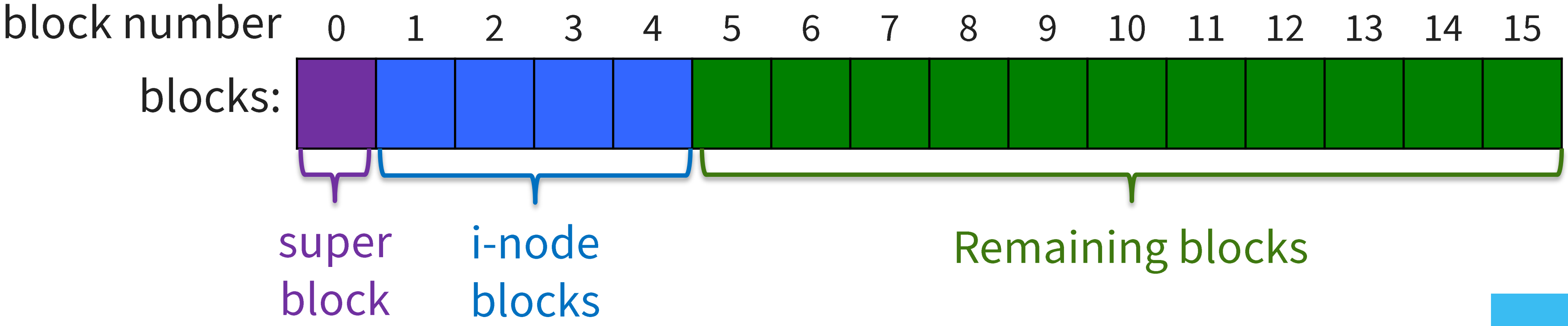
- Later became Linux ext2 and ext3 FS

Tree-based, multi-level index

FFS Superblock

Identifies file system’s key parameters:

- type
- block size
- inode array location and size
  (or analogous structure for other FSs)
- location of free list



block number 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

blocks:

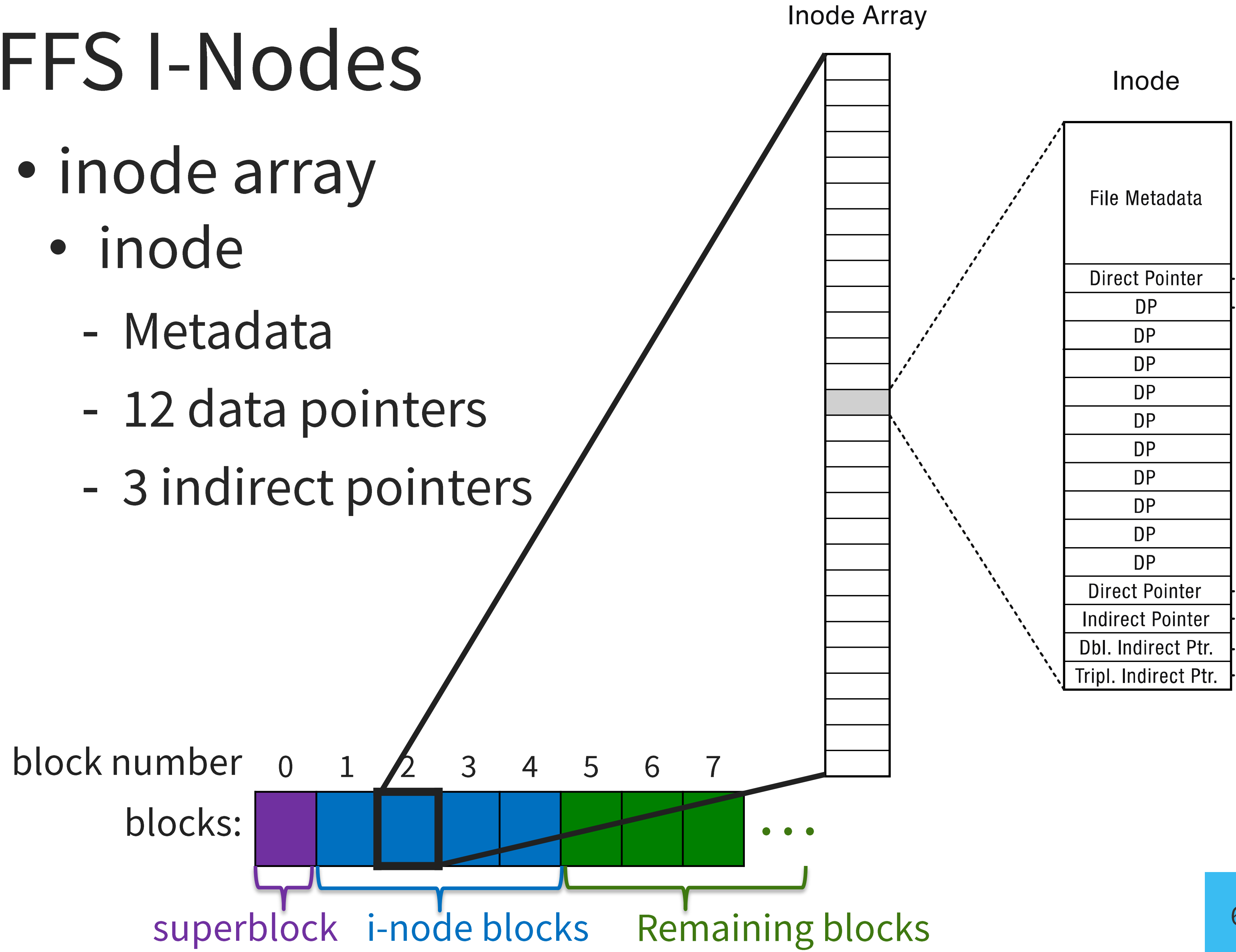
super block

i-node blocks

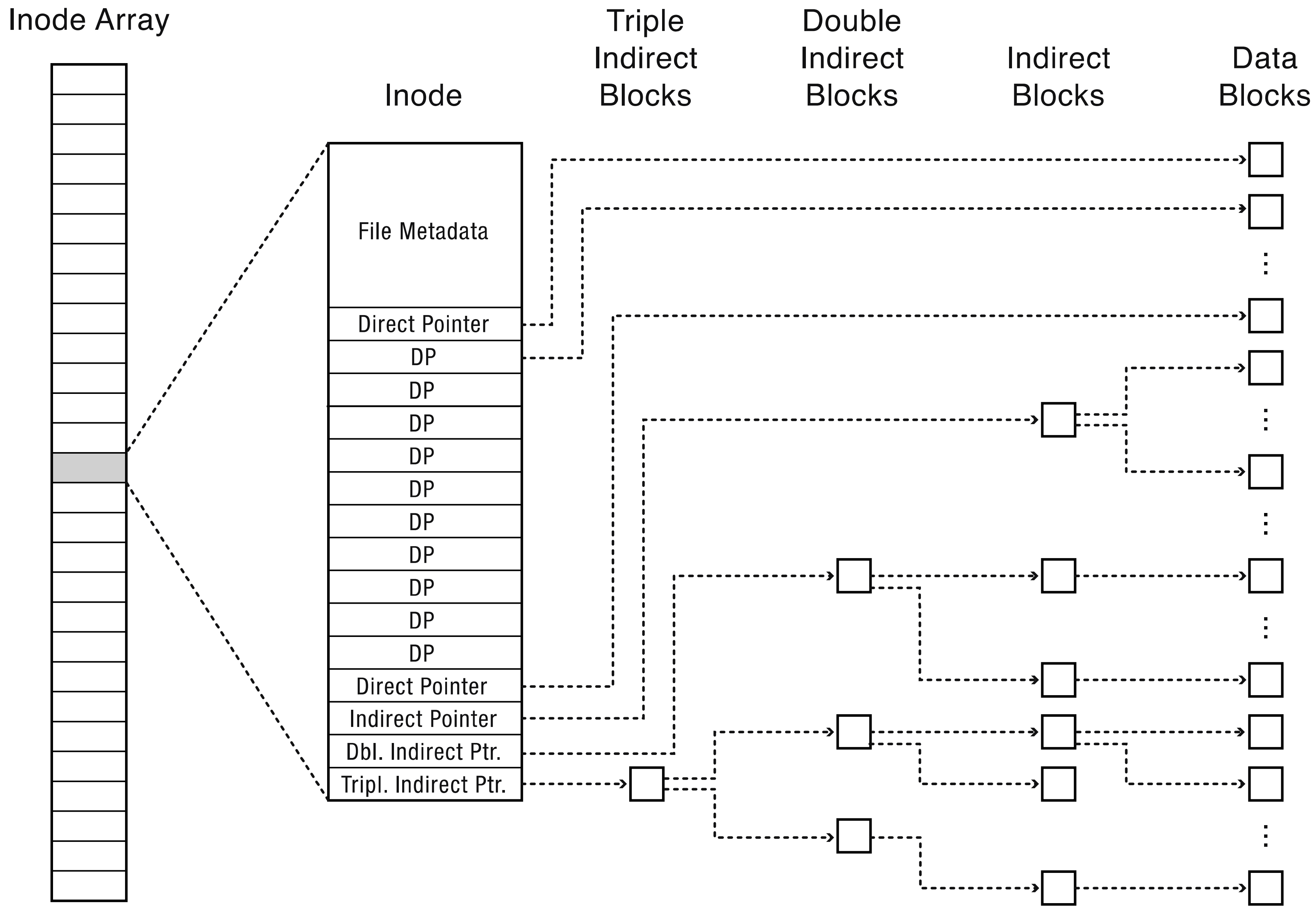
Remaining blocks

FFS I-Nodes

- inode array
  - inode
    - Metadata
    - 12 data pointers
    - 3 indirect pointers

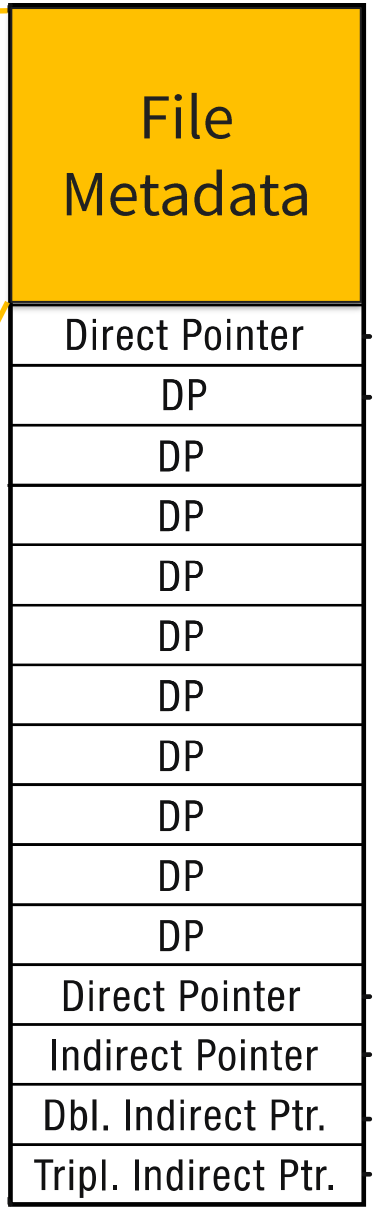


FFS: Index Structures



What else is in an inode?

- Type
  - ordinary file
  - directory
  - symbolic link
  - special device
- Size of the file (in #bytes)
- # links to the i-node
- Owner (user id and group id)
- Protection bits
- Times: creation, last accessed, last modified



File Metadata

Direct Pointer

DP

DP

DP

DP

DP

DP

DP

DP

DP

DP

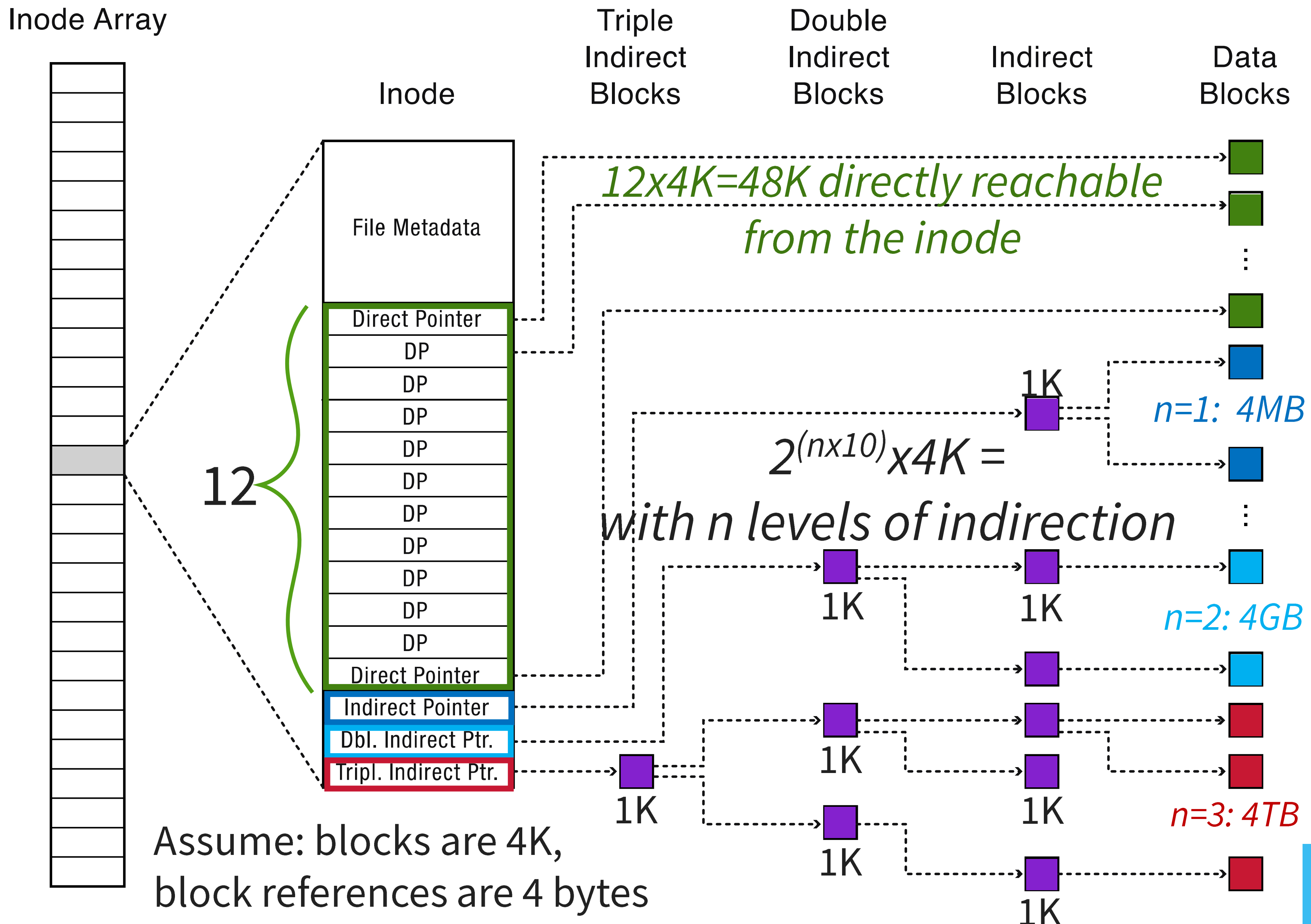
Direct Pointer

Indirect Pointer

Dbl. Indirect Ptr.

Tripl. Indirect Ptr.

FFS: Index Structures



12x4K=48K directly reachable from the inode

n=1: 4MB

n=2: 4GB

n=3: 4TB

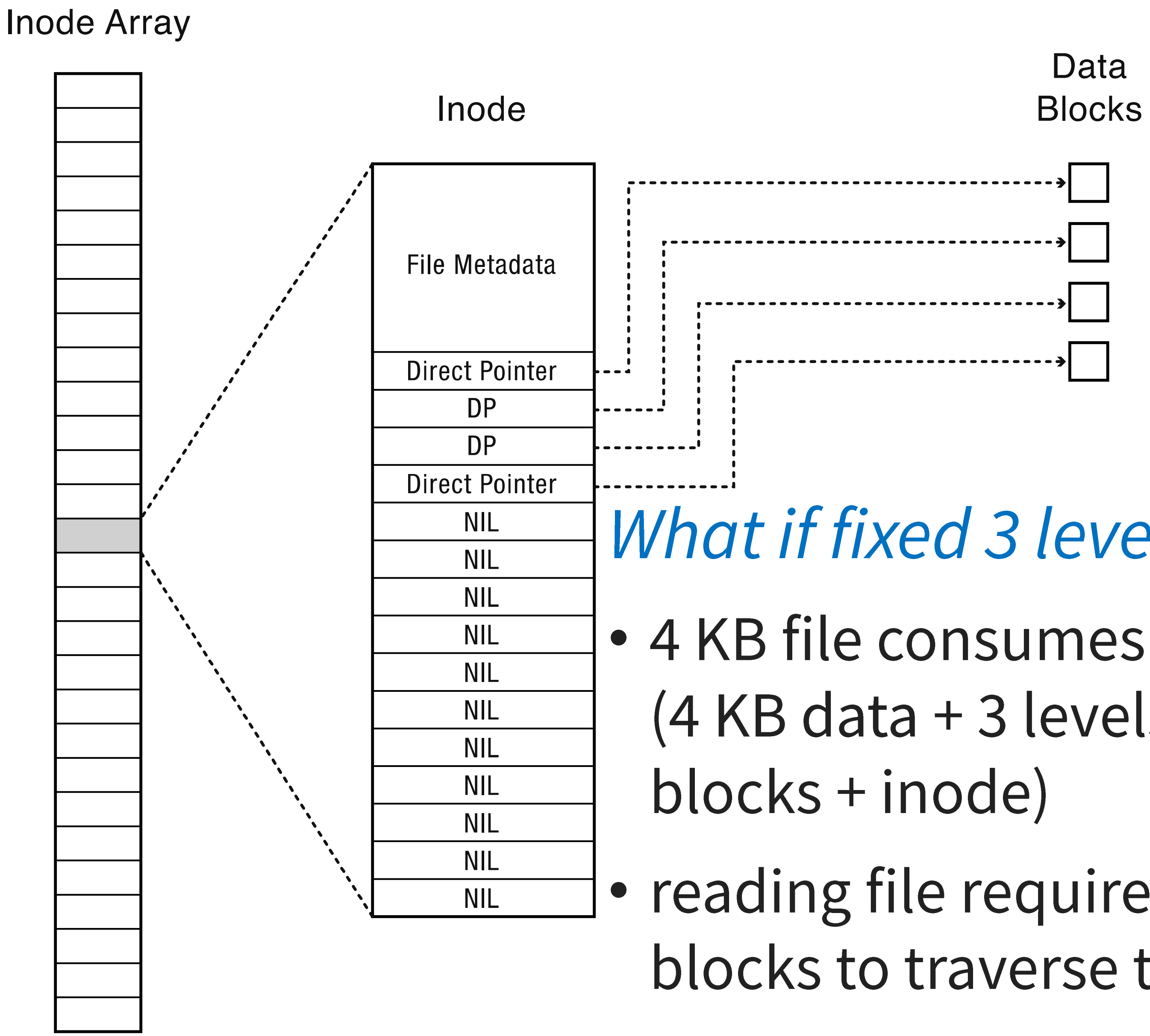
$2^{(n \times 10)} \times 4K =$ *with n levels of indirection*

Assume: blocks are 4K, block references are 4 bytes

4 Characteristics of FFS

1. Tree Structure
    - efficiently find any block of a file
2. High Degree (or fan out)
    - minimizes number of seeks
    - supports sequential reads & writes
3. Fixed Structure
    - implementation simplicity
4. Asymmetric
    - not all data blocks are at the same level
    - supports large files
    - small files don’t pay large overheads

Small Files in FFS



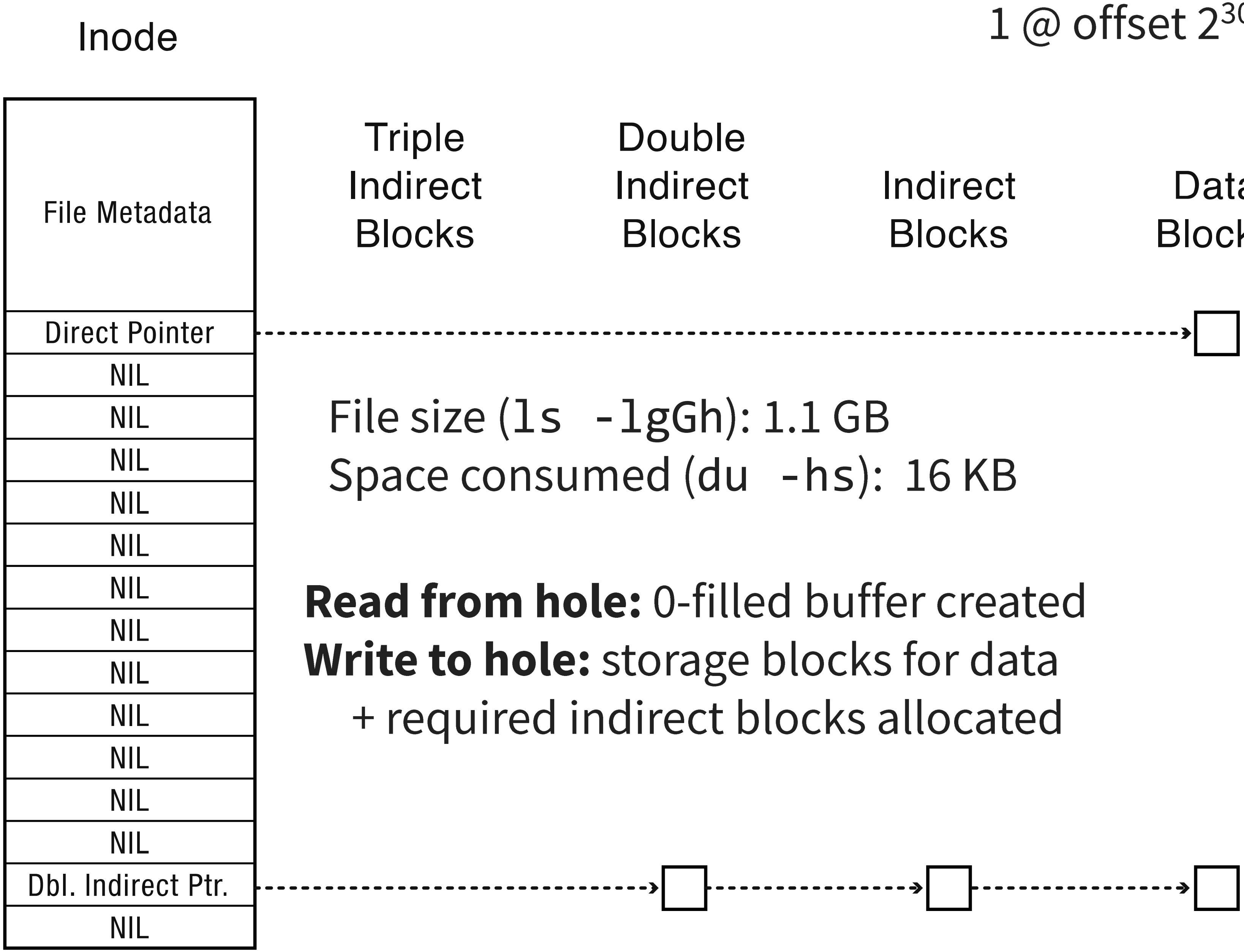
all blocks reached via direct pointers

What if fixed 3 levels instead?

- 4 KB file consumes ~16 KB (4 KB data + 3 levels of 4KB indirect blocks + inode)
- reading file requires reading 5 blocks to traverse tree

Sparse Files in FFS

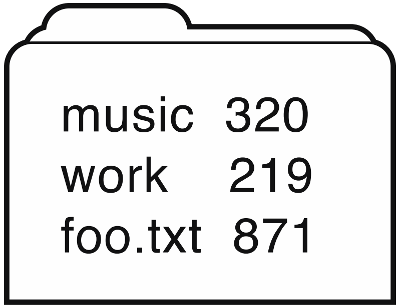
Example:
2 x 4 KB blocks: 1 @ offset 0
1 @ offset 2^{30}



File size (`ls -lgGh`): 1.1 GB
Space consumed (`du -hs`): 16 KB

Read from hole: 0-filled buffer created
Write to hole: storage blocks for data
+ required indirect blocks allocated

FFS Directory Structure



music	320
work	219
foo.txt	871

Originally: array of 16 byte entries

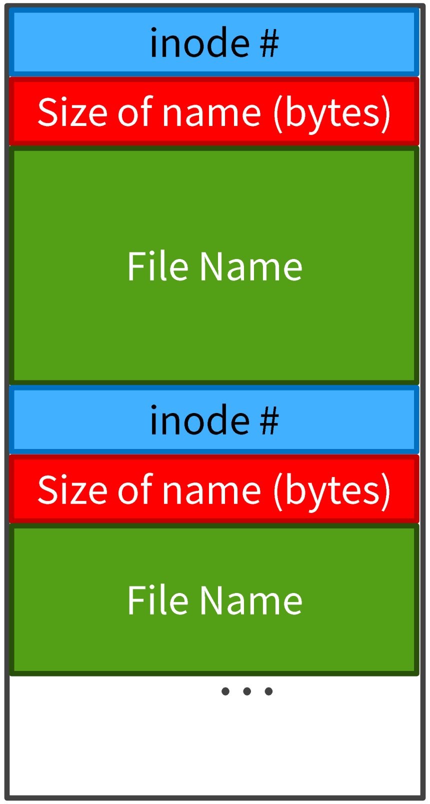
- 14 byte file name
- 2 byte i-node number

Now: linked lists. Each entry contains:

- 4-byte i-node number
- Length of name
- Name (UTF8 or some other Unicode encoding)

First entry is “.”, points to self (this directory’s inode)

Second entry is “..”, points to parent’s inode

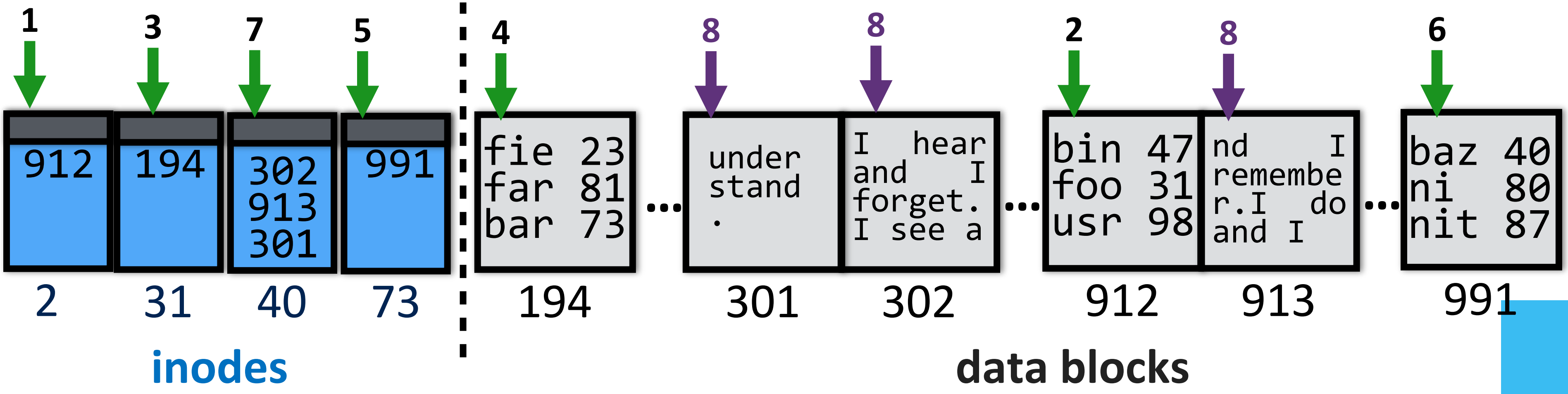


FFS: Steps to reading `/foo/bar/baz`

Read & Open:

(1) inode #2 (root always has inumber 2), find root’s blocknum (912)
(2) root directory (in block 912), find foo’s inumber (31)
(3) inode #31, find foo’s blocknum (194)
(4) foo (in block 194), find bar’s inumber (73)
(5) inode #73, find bar’s blocknum (991)
(6) bar (in block 991), find baz’s inumber (40)
(7) inode #40, find data blocks (302, 913, 301)
(8) data blocks (302, 913, 301)

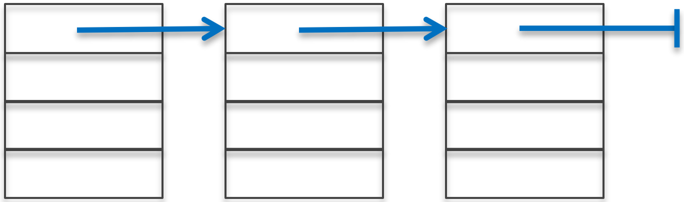
Caching allows first few steps to be skipped



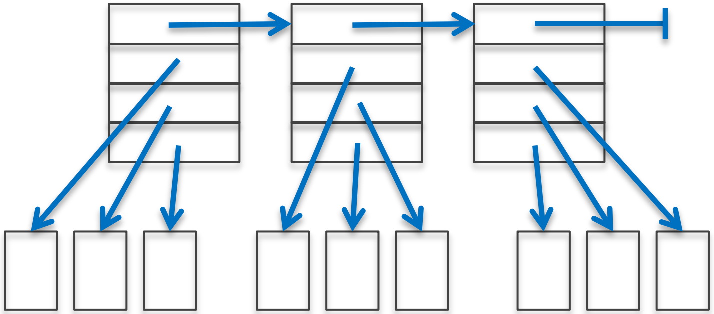
Free List

- List of blocks not in use
- How to maintain?

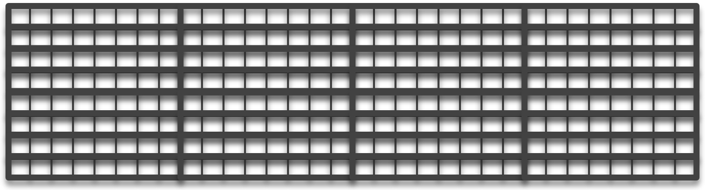
1. linked list of free blocks
    - inefficient (why?)



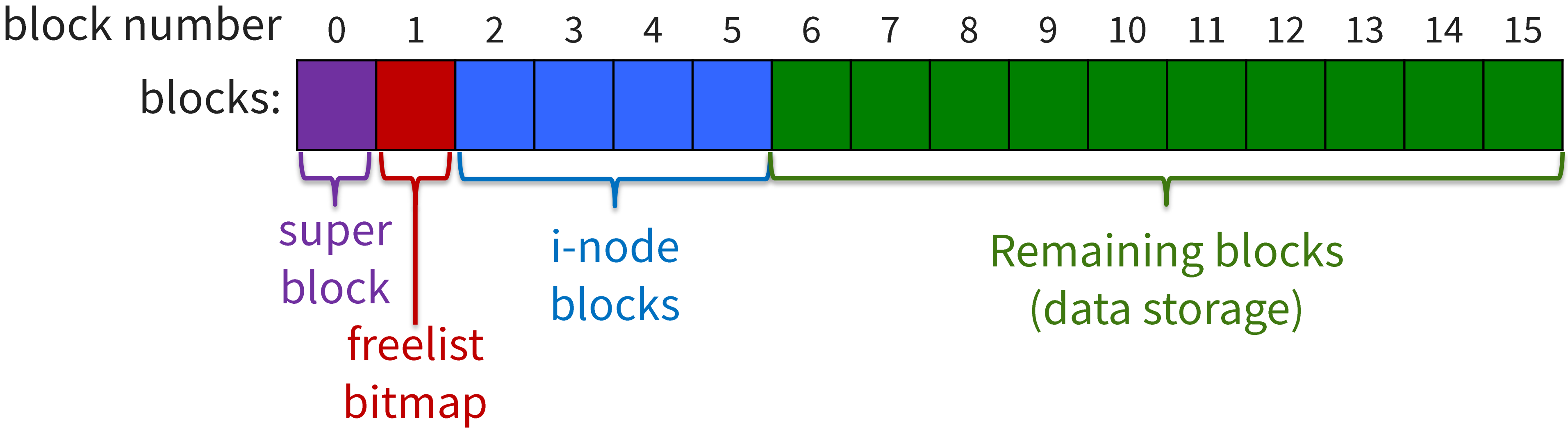
2. linked list of metadata blocks that in turn point to free blocks
    - simple and efficient



3. bitmap
    - good for contiguous allocation



FFS Layout, More Complete



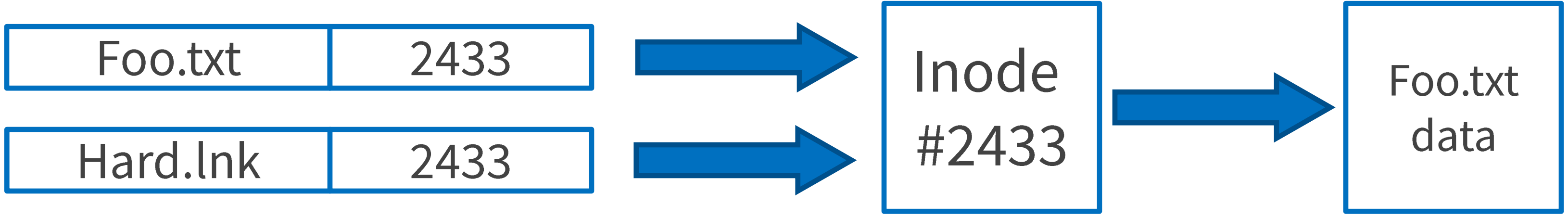
File System API

Creating and deleting files

- creat(): creates
  1. a new file with some metadata; and
  2. a name for the file in a directory
- link() creates a *hard link*–a new name for the same underlying file, and increments link count in inode
- unlink() removes a name for a file from its directory and decrements link count in inode. If last link, file itself and resources it held are deleted

Hard & Soft Links

- **Hard link**: a mapping from a name to a specific file or directory by inode #

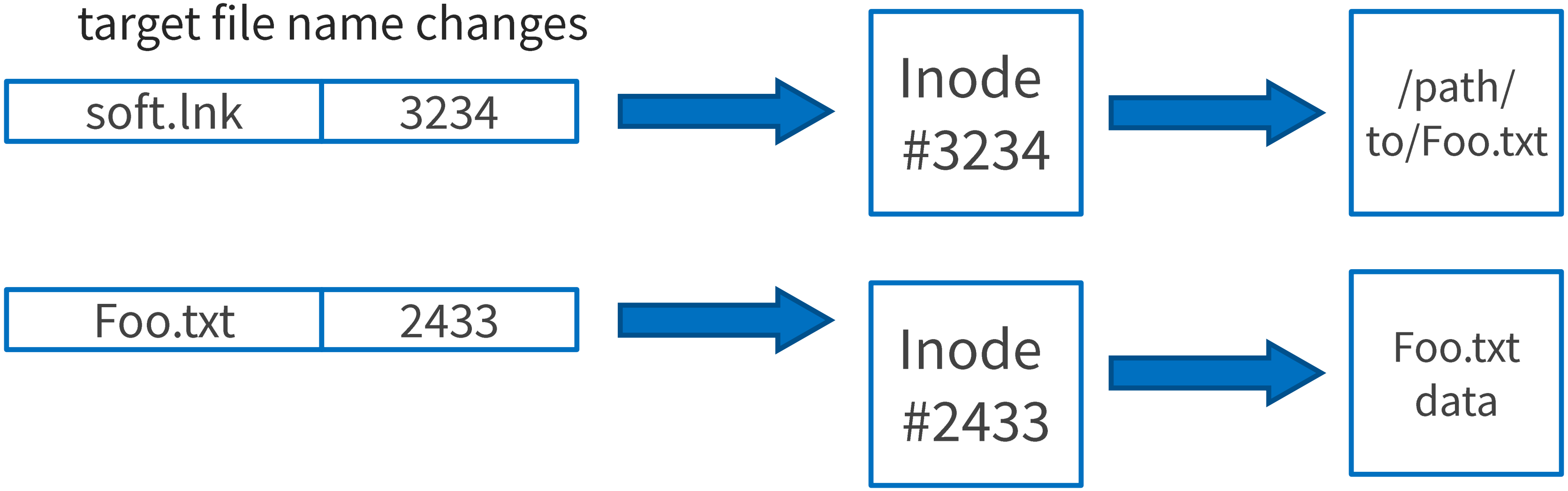


Foo.txt	2433
Hard.lnk	2433

Inode #2433

Foo.txt data

- **Soft link**: a mapping from a file name to another file name (just a file containing the name)
  - use as *alias*: a soft link continues to remain valid when the (path of) the target file name changes



soft.lnk	3234

Inode #3234

/path/to/Foo.txt

Foo.txt	2433

Inode #2433

Foo.txt data

FFS Pros and Cons

- Good:
  - Efficient storage for both small and large files
  - Locality for both small and large files
  - Locality for metadata
  - Fixed structure leads to simple implementation
- Bad:
  - Inefficient for tiny files: need both inode and data block
  - Inefficient encoding for mostly contiguous files
  - Needs 10%-20% unutilized disk space to prevent fragmentation

NTFS (NT File System)

[early 90’s]

Microsoft’s New File System

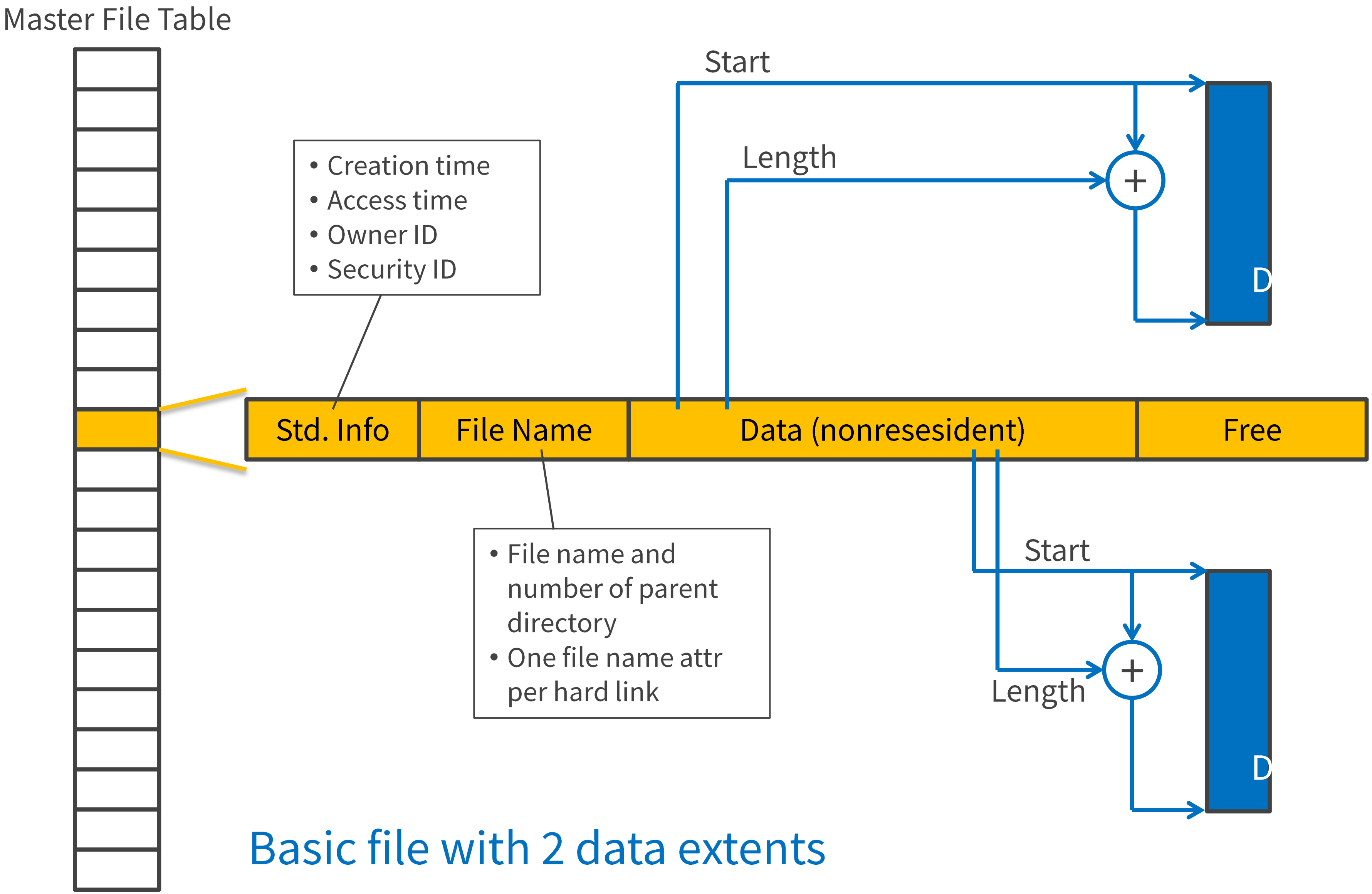
- Developed in 1990s to replace FAT
- Borrows ideas from FFS – tree structure, attributes stored with files
- Still used in modern Windows
- Linux ext4 has similar design

Flexible Tree Structure with Extents

NTFS Index Structure Design

- Extents
  - Track ranges of contiguous blocks rather than single blocks
- Flexible Tree
  - File represented by variable depth tree, depending on number of extents
- MFT (Master File Table)
  - Array of 1KB records holding trees' roots
  - Similar to inode table, but 1 file can have multiple MFT entries
  - Each record stores sequence of variable-sized **attribute records**
    - Both data and metadata are attributes
    - Attributes can be **resident** (fit in the record) or **nonresident**

NTFS Index Structure Example

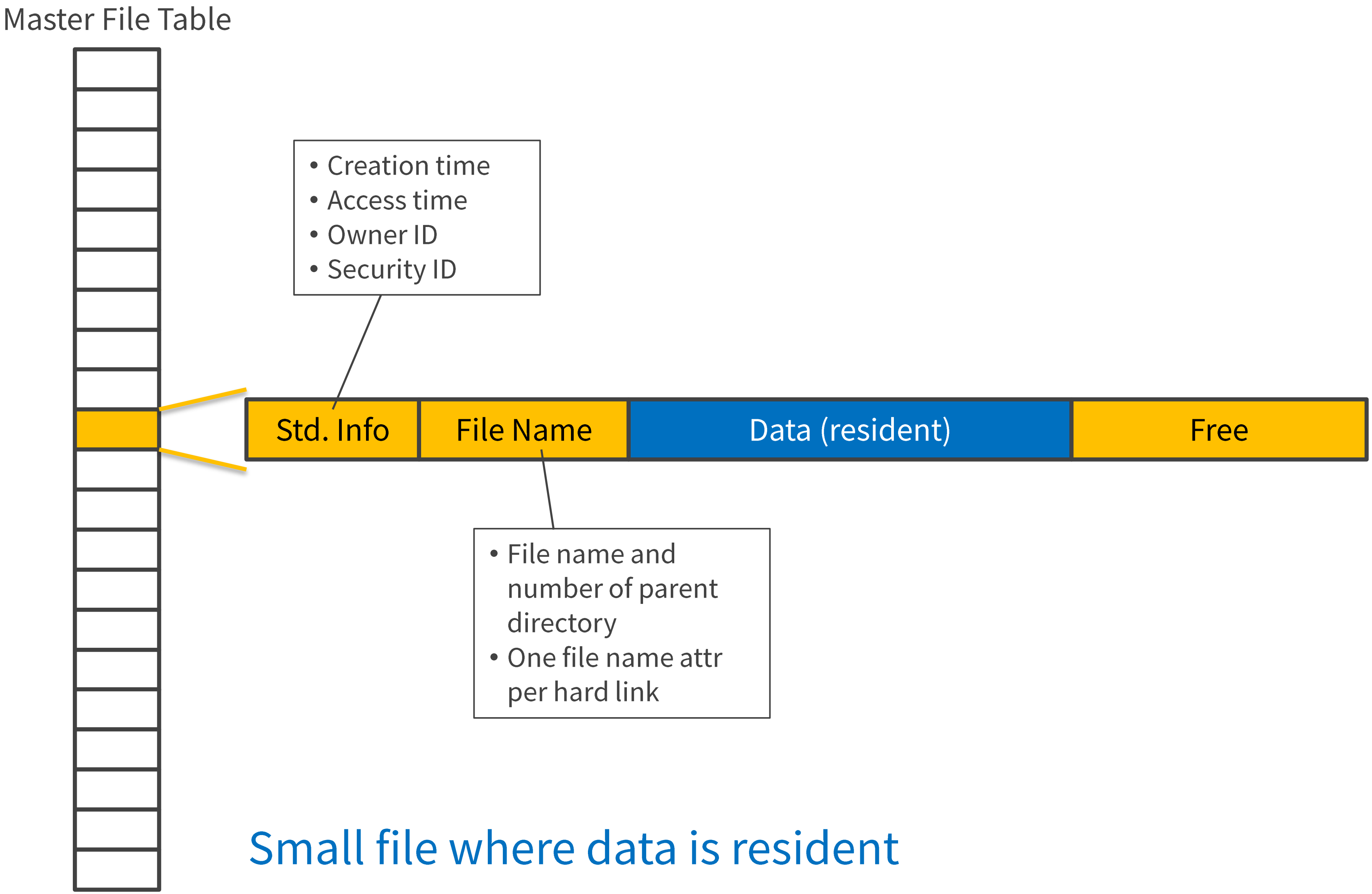


- Creation time
- Access time
- Owner ID
- Security ID

- File name and number of parent directory
- One file name attr per hard link

Basic file with 2 data extents

NTFS Index Structure Example

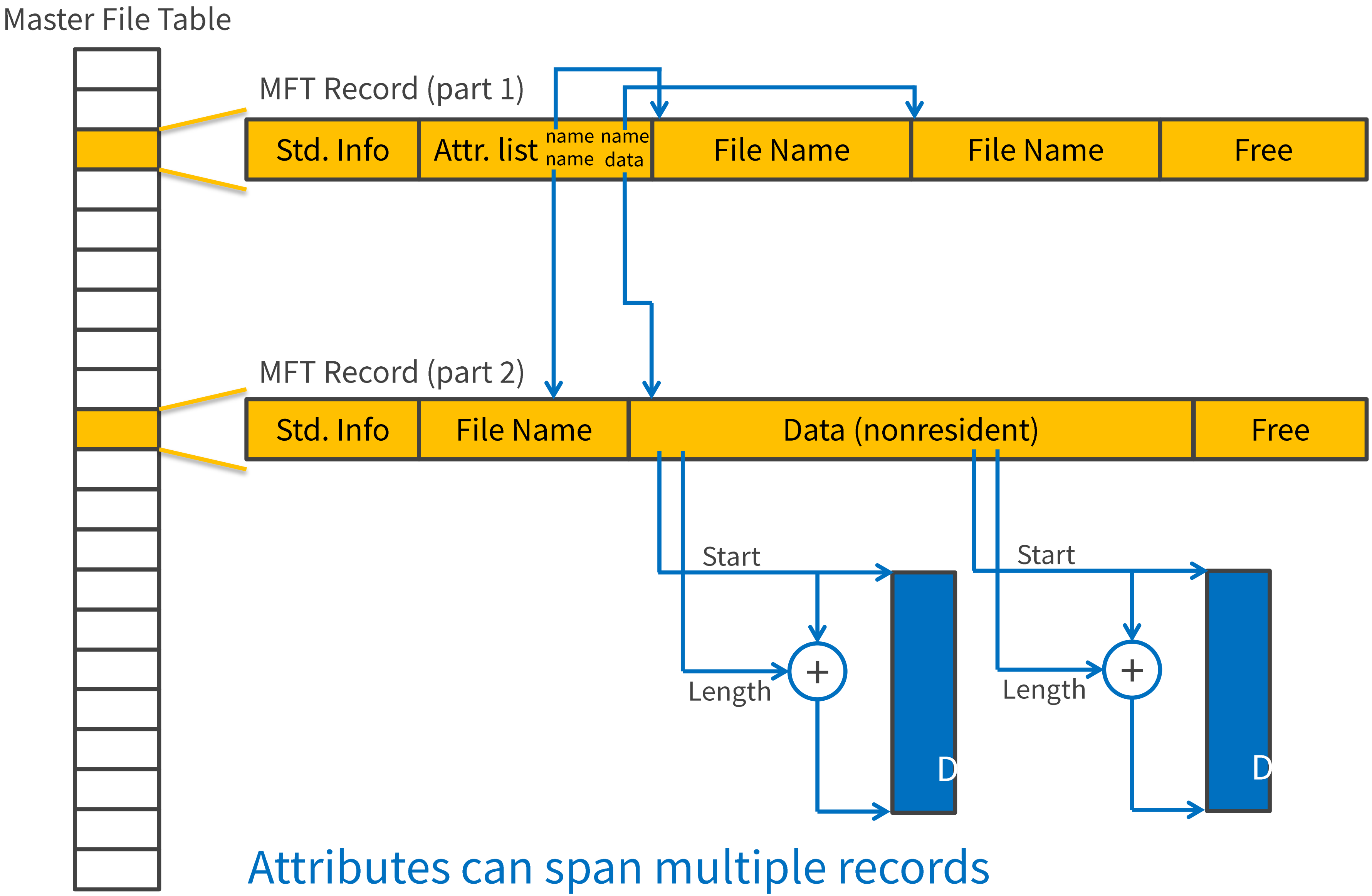


- Creation time
- Access time
- Owner ID
- Security ID

- File name and number of parent directory
- One file name attr per hard link

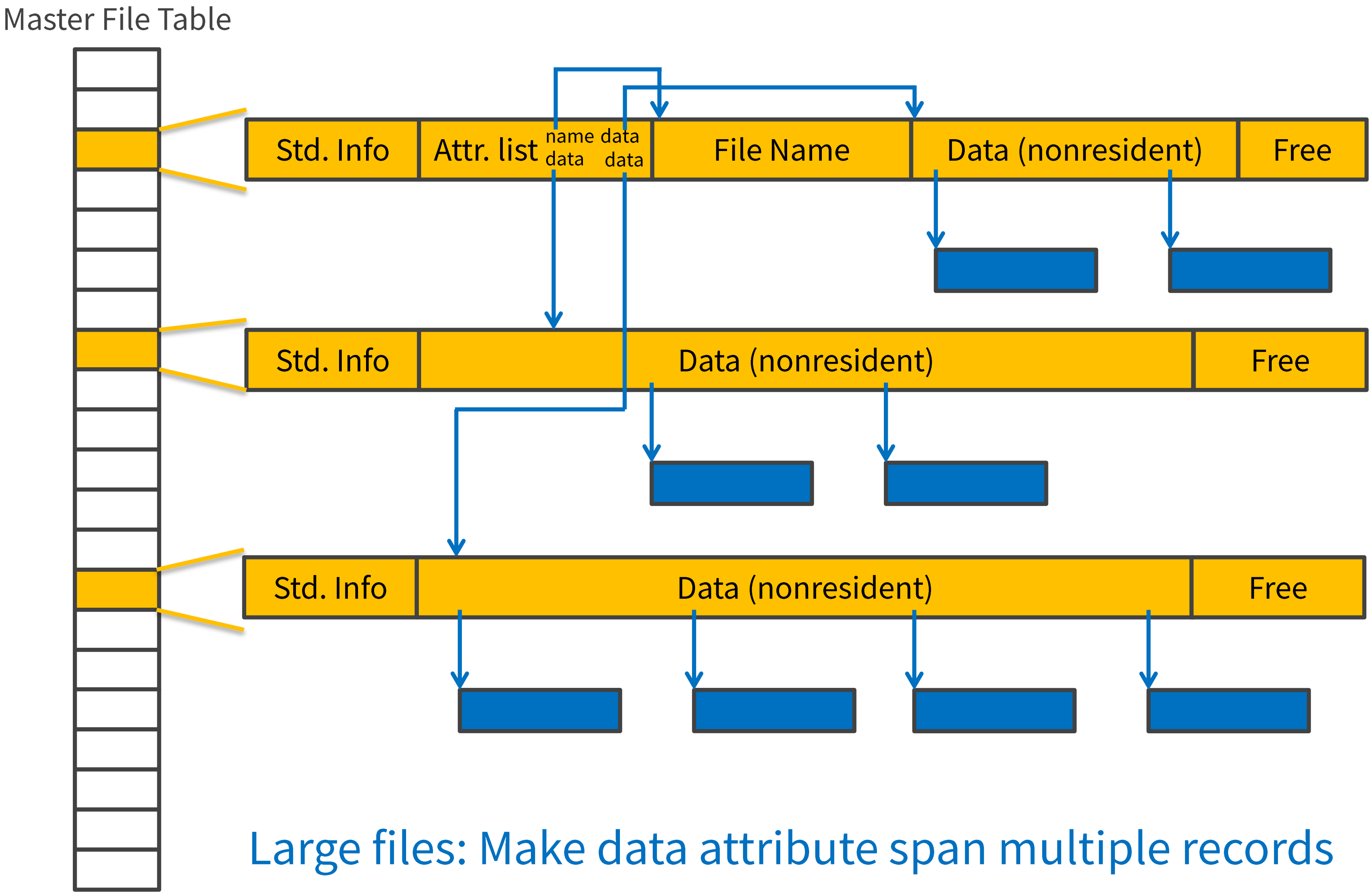
Small file where data is resident

NTFS Index Structure Example



Attributes can span multiple records

NTFS Index Structure Example



Large files: Make data attribute span multiple records