File Systems

CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

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

Where shall we store our data?

- Process Memory? *(why is this a bad idea?)*

File Systems 101

Long-term Information Storage Needs

- large amounts of information
- information must survive processes
- need concurrent access by multiple processes

Solution: the File System Abstraction

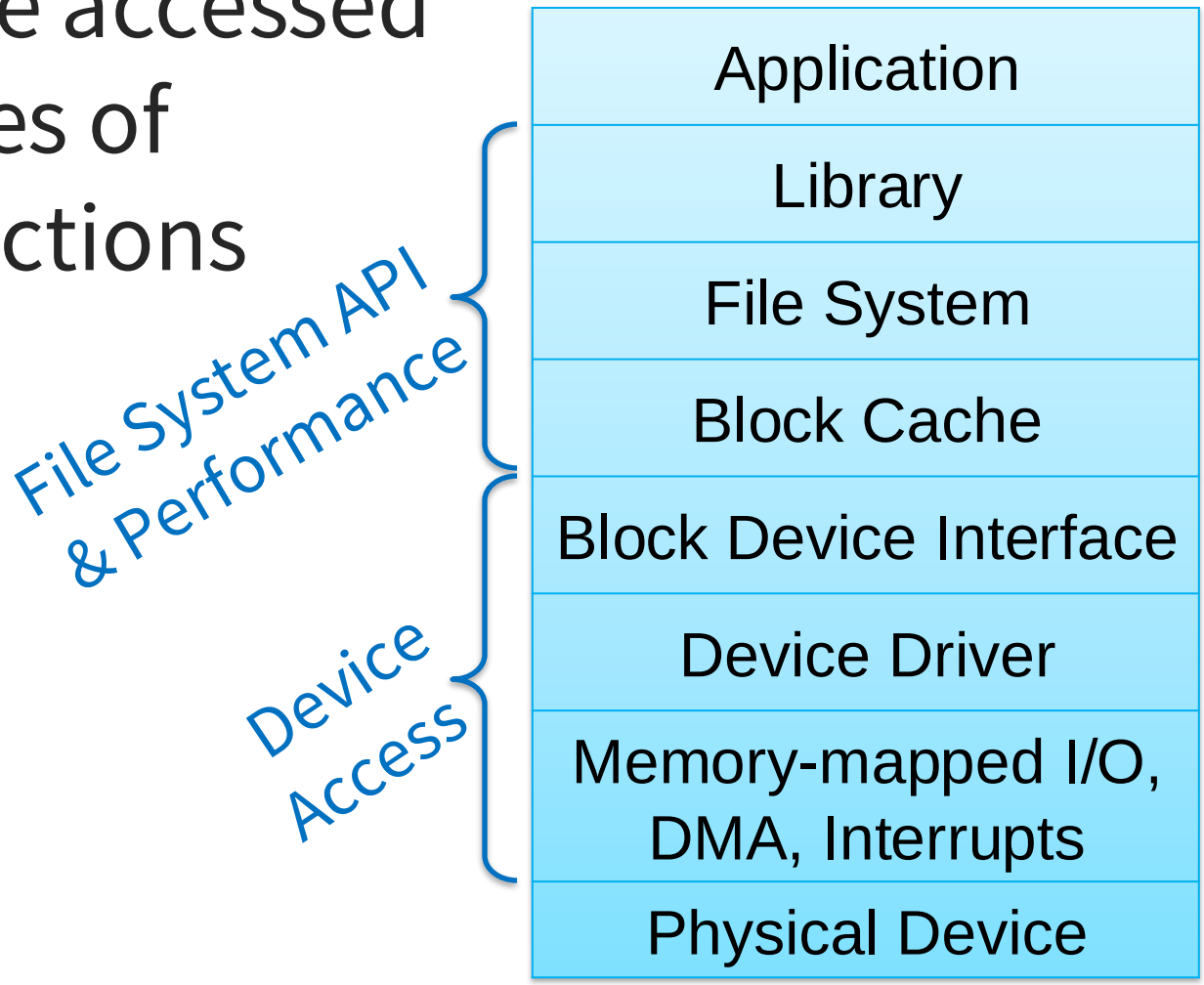
- Presents applications w/ **persistent**, **named** data
- Two main components:
  - Files
  - Directories

The File Abstraction

- File: a named collection of data
- has two parts
  - data – what a user or application puts in it
    - typically an array of bytes
  - metadata – information added and managed by the OS
    - name, size, owner, security info, modification time

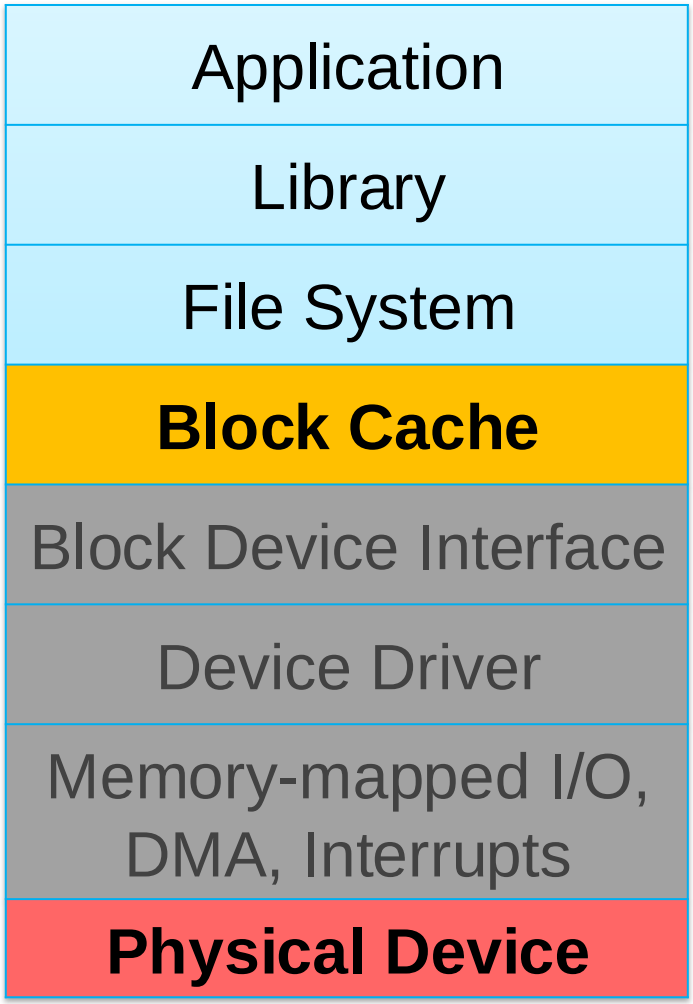
The abstraction stack

I/O systems are accessed through a series of layered abstractions



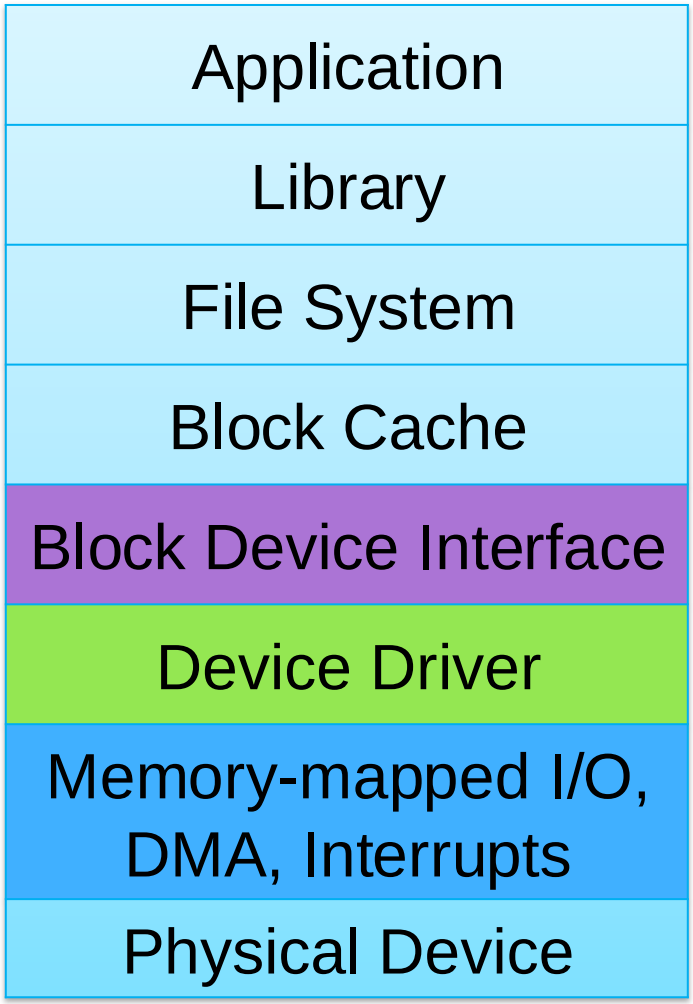
The Block Cache

- a **cache** for the **disk**
- caches recently read blocks
- buffers recently written blocks



More Layers

- allows data to be read or written in fixed-sized blocks
- uniform interface to disparate devices
- translate between OS abstractions and hw-specific details of I/O devices
- Control registers, bulk data transfer, OS notifications



First things first: Name the File!

1. Files are abstracted unit of information
2. Don’t care exactly where *on disk* the file is

➜ Files have human readable names

- file given name upon creation
- use the name to access the file

Name + Extension

Naming Conventions

- Some things OS dependent:

  Windows not case sensitive, Posix (typically) is

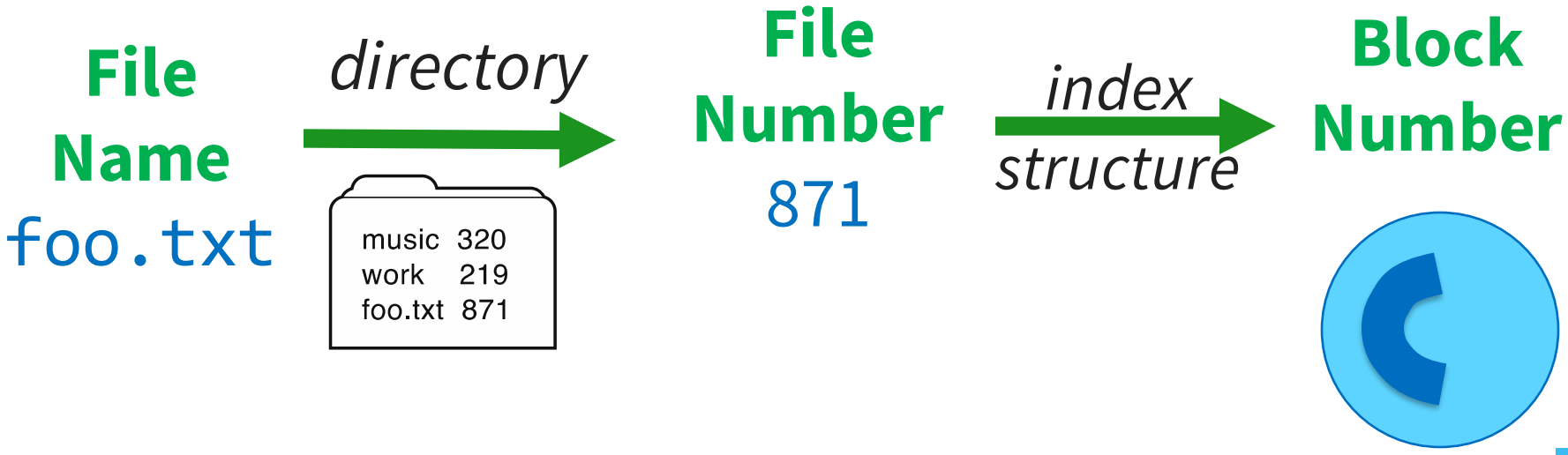
File Extensions, OS dependent:

- Windows:
  - attaches meaning to extensions
  - associates applications to extensions
- Posix:
  - extensions not enforced by OS
  - Some apps might insist upon them (.c, .h, .o, .s, for C compiler)

Directory

Directory: provides names for files

- Stored in a file
- A mapping from each name to a specific underlying file or directory



Path Names

Absolute: path of file from the root directory

`/home/ada/projects/babbage.txt`

Relative: path from the working directory

`projects/babbage.txt`

(current working dir stored in process' PCB)

2 special entries in each Posix directory:

"." current dir

".." for parent

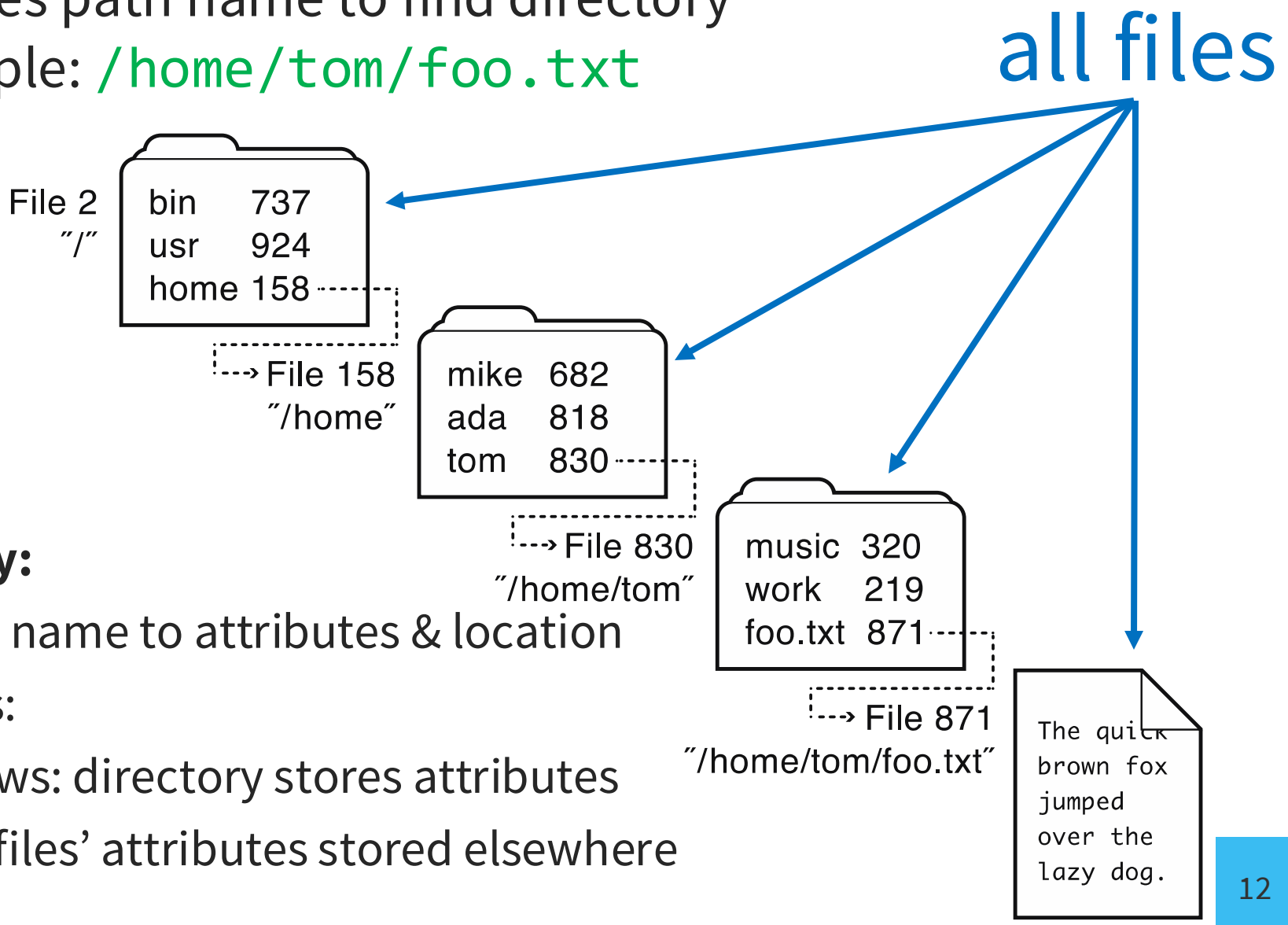
To access a file:

- Go to the folder where file resides —OR—
- Specify the path where the file is

Directories

OS uses path name to find directory

Example: `/home/tom/foo.txt`



Directory:

maps file name to attributes & location

2 options:

- Windows: directory stores attributes
- Posix: files' attributes stored elsewhere

Basic File System Operations

- Create a new file
- Open an existing file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Delete a file
- Truncate a file

Challenges for File System Designers

Performance: despite limitations of disks

- leverage spatial locality

Flexibility: need jacks-of-all-trades, diverse workloads, not just FS for application X

Persistence: maintain/update user data + internal data structures on persistent storage devices

Reliability: must store data for long periods of time, despite OS crashes or HW malfunctions

Security: file should have protection mechanisms

Implementation Basics

Directories

- file name → file number

Index structures

- file number + offset → block

Free space maps

- find a free block

Locality heuristics

- policies enabled by above mechanisms
  - group directories
  - prefetching
  - make writes sequential
  - keep blocks of a file close together

File System Properties

Most files are small

- need strong support for small files
- block size can’t be too big

Some files are very large

- must allow large files
- large file access should be reasonably efficient

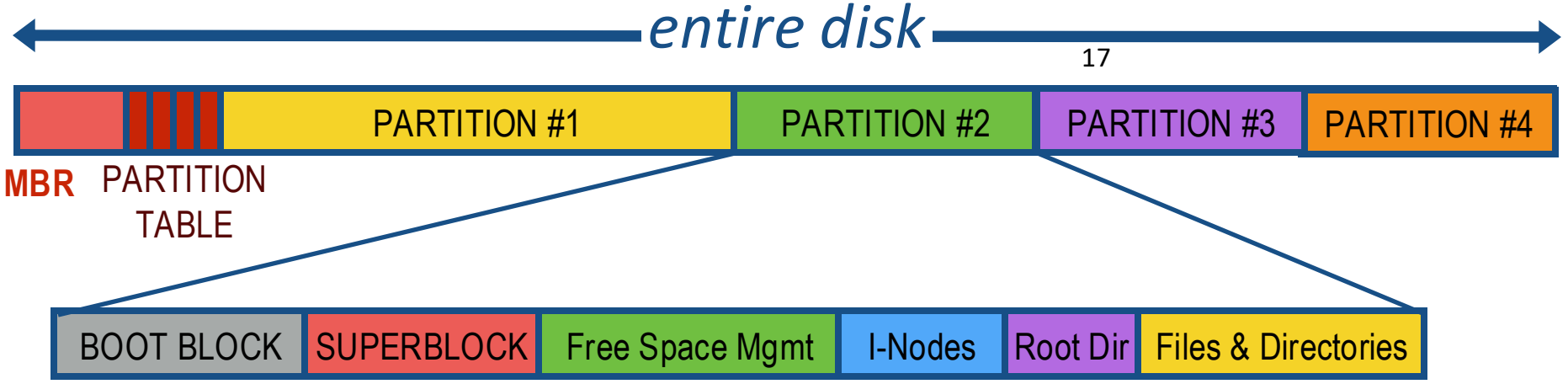
File System Layout

File System is stored on *disks*

- disk can be divided into 1 or more *partitions*
- Sector 0 of disk called Master Boot Record
  - Contains code for booting
- end of MBR: partition table (partitions' start & end addrs)

First block of each partition has *boot block*

- more code loaded by MBR and executed on boot



Storing Files

Files can be allocated in different ways:

- Contiguous allocation

  All blocks together, in order

- Linked Structure

  Each block points to the next block

- Indexed Structure

  Some kind of tree of blocks

Which is best?

- For sequential access? Random access?
- Large files? Small files? Mixed?

Contiguous Allocation

All blocks together, in order

+ **Simple:** state required per file: start block & size
+ **Efficient:** entire file can be read with one seek
– **External Fragmentation:** see next slide
– **Usability:** user needs to know size of file at time of creation

Used in CD-ROMs, DVDs

Fragmentation

Internal Fragmentation

- allocated file size (in blocks) may be larger than requested file size (in bytes); this size difference is wasted disk space

External Fragmentation

- total disk space exists to store a file, but it is not useful because the free blocks are not contiguous, and the file does not fit in any of the holes

Linked List Allocation

Each file is stored as linked list of blocks

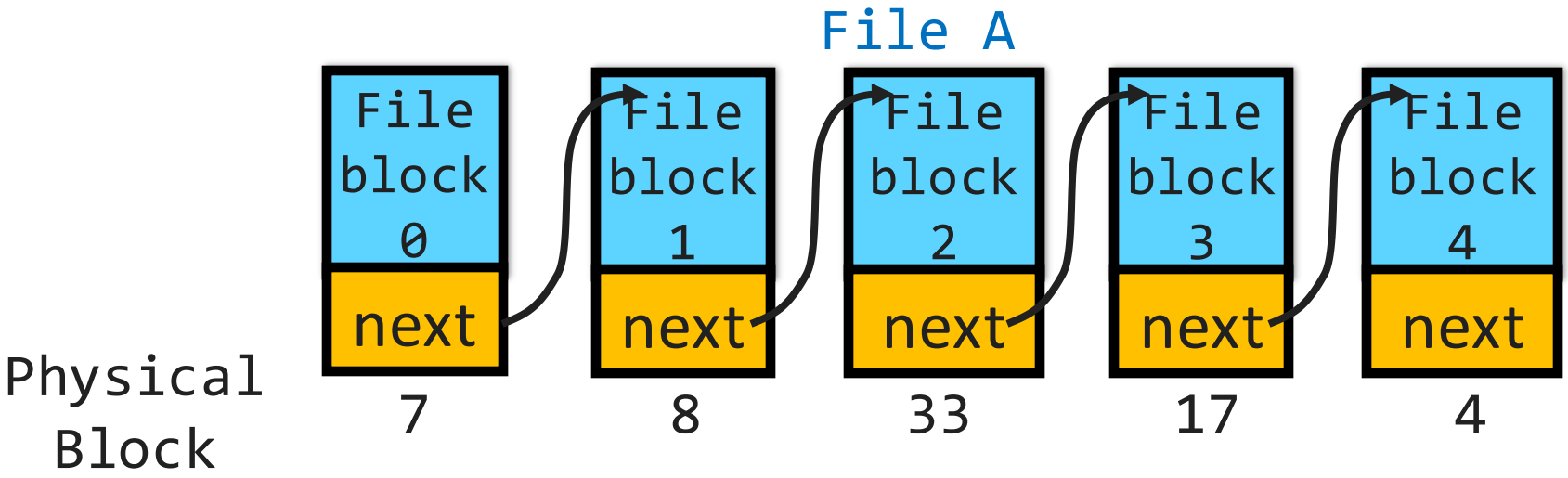
- First word of each block points to next block
- Rest of disk block is file data

+ Space Utilization: no space lost to external fragmentation

+ Simple: only need to find 1st block of each file

– Performance: random access is slow

– Implementation: blocks mix meta-data and data



File Allocation Table (FAT) FS

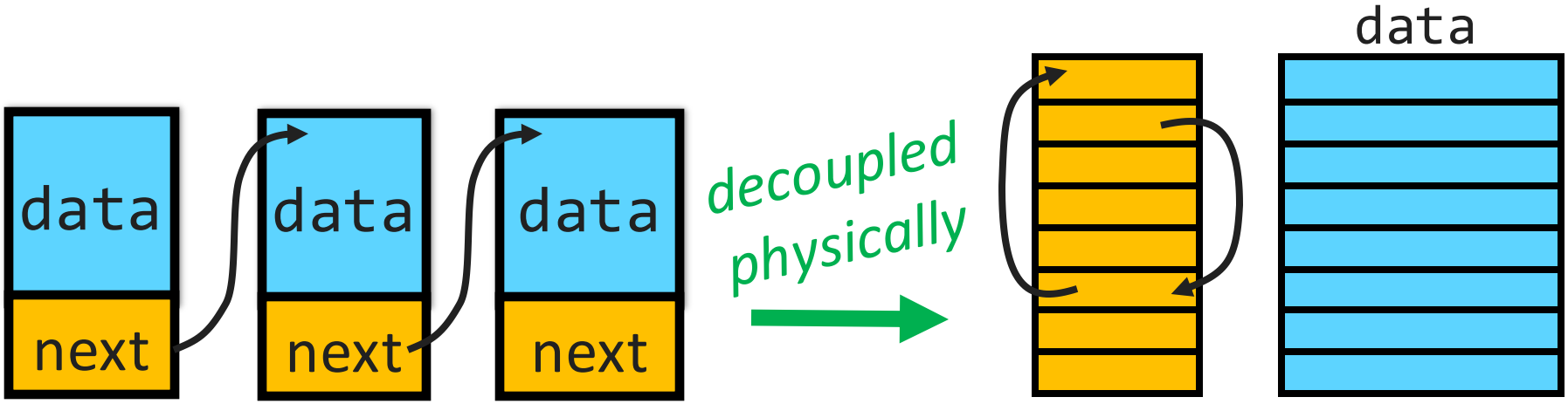
[late 70's]

Microsoft File Allocation Table

- originally: MS-DOS, early version of Windows
- today: still widely used (e.g., CD-ROMs, thumb drives, camera cards)

File table:

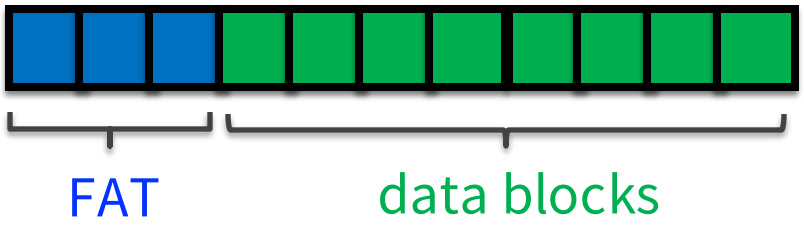
- Linear map of all blocks on disk
- Each file a linked list of blocks

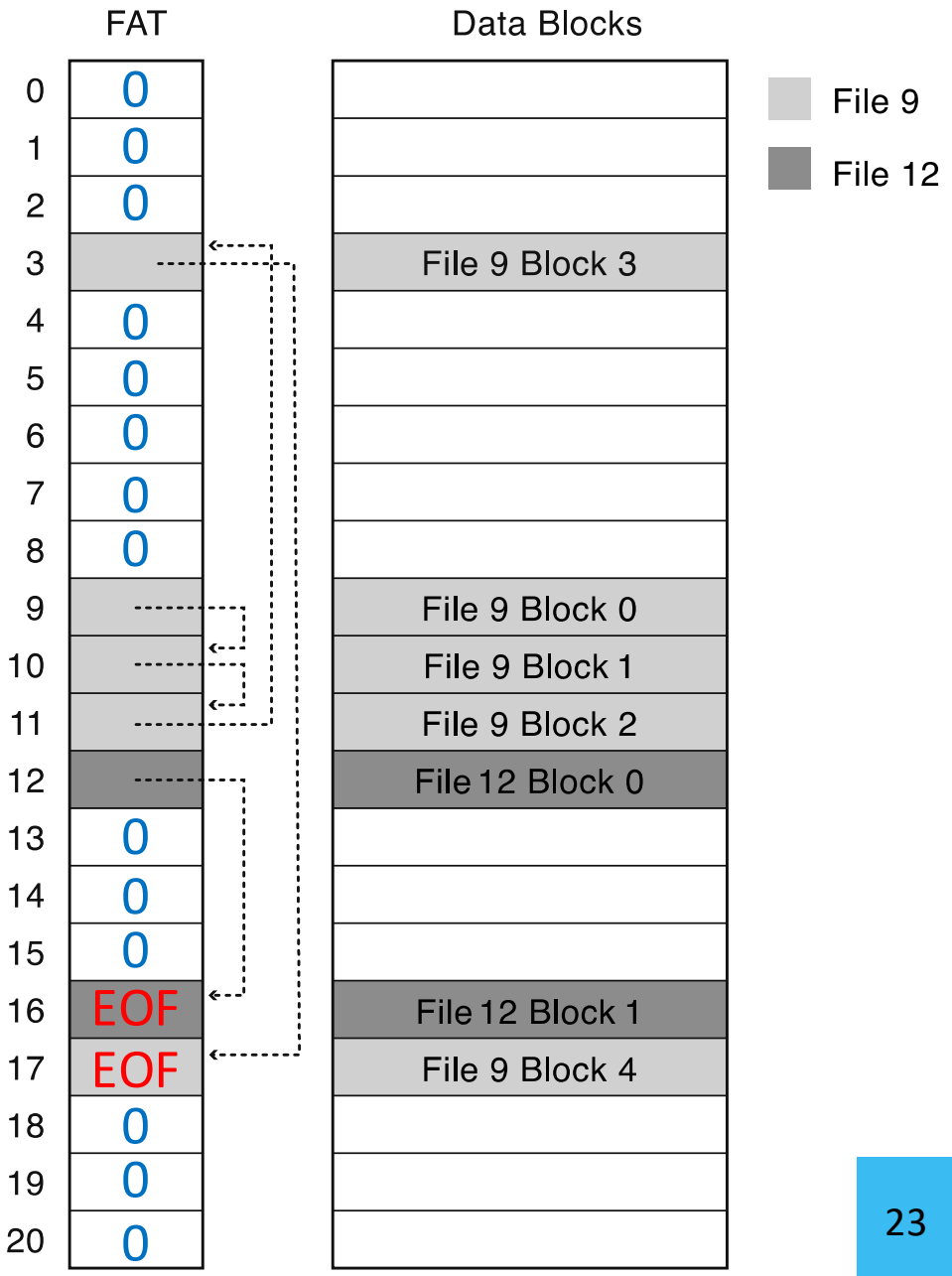


FAT File System

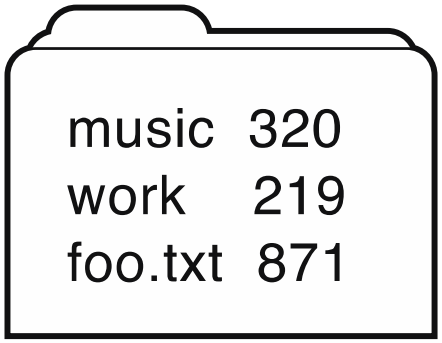
- 1 entry per block
- EOF for last block
- 0 indicates free block
- directory entry maps name to FAT index

Folder	
bart.txt	9
maggie.txt	12





FAT Directory Structure



Folder: a file with 32-byte entries

Each Entry:

- 8 byte name + 3 byte extension (ASCII)
- creation date and time
- last modification date and time
- first block in the file (index into FAT)
- size of the file
- Long and Unicode file names take up multiple entries

How is FAT Good?

+ Simple: state required per file: start block only
+ Widely supported
+ No external fragmentation
+ block used only for data

How is FAT Bad?

- Poor locality
- Poor random access
  - 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

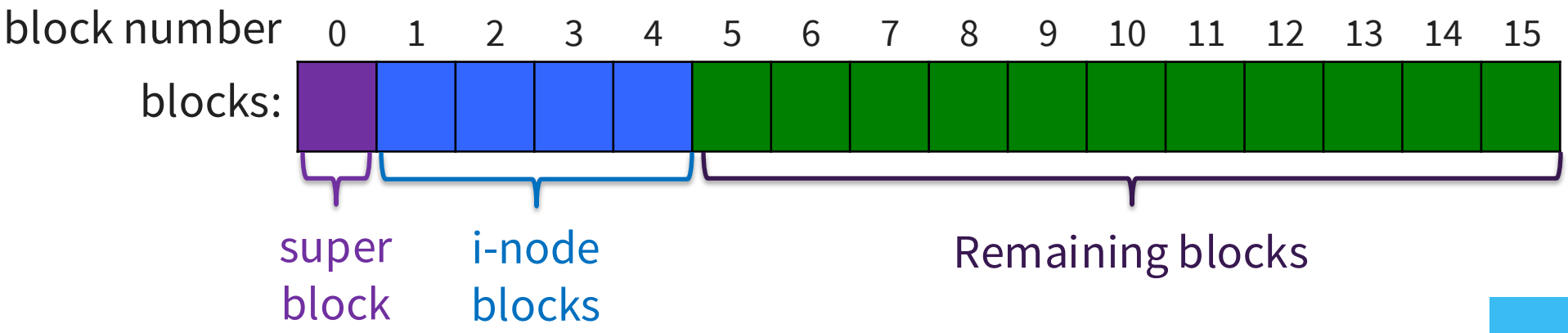
Unix File System (UFS)

Tree-based, multi-level index

UFS Superblock

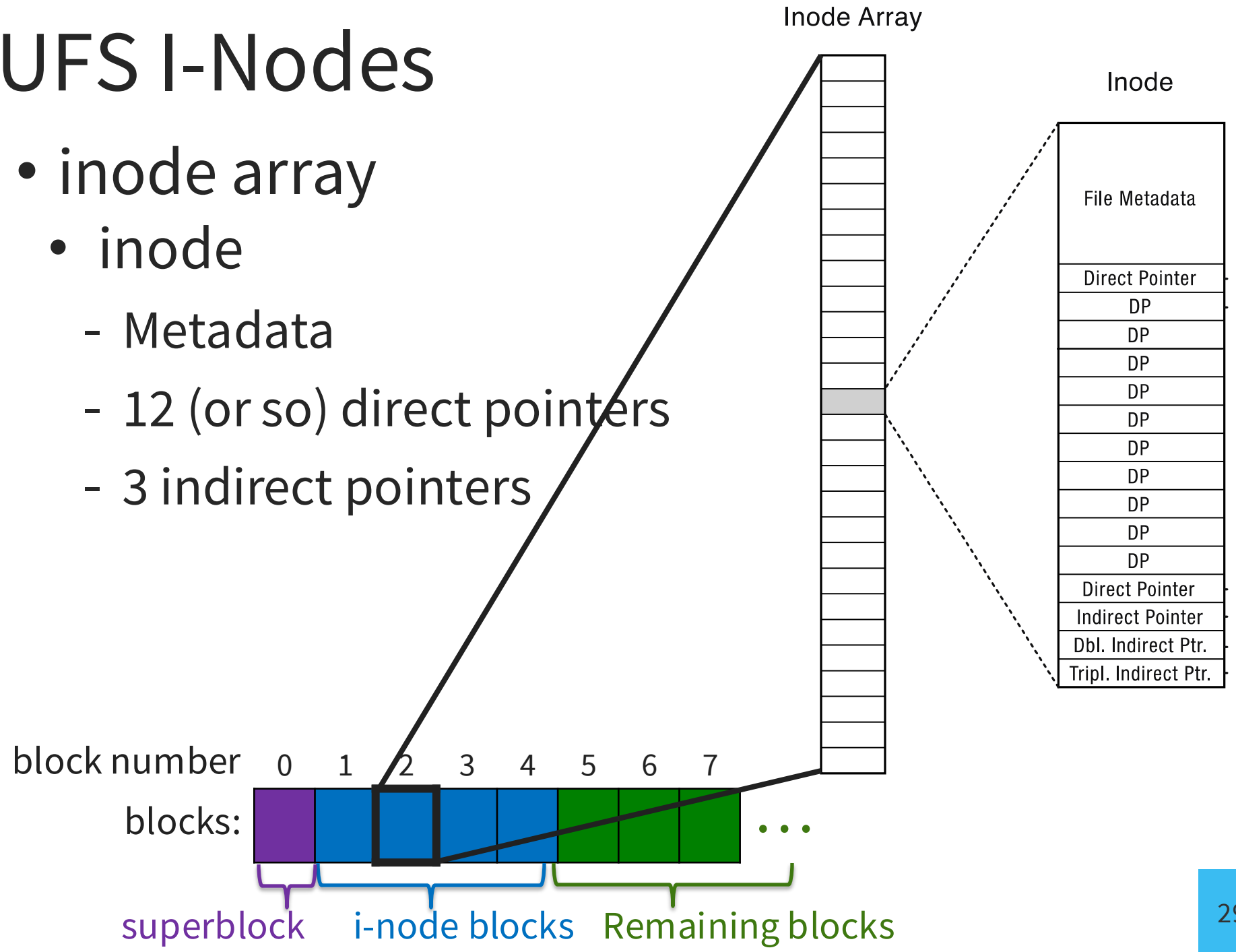
Identifies file system's key parameters:

- type
- block size
- inode array location and size
- location of free list

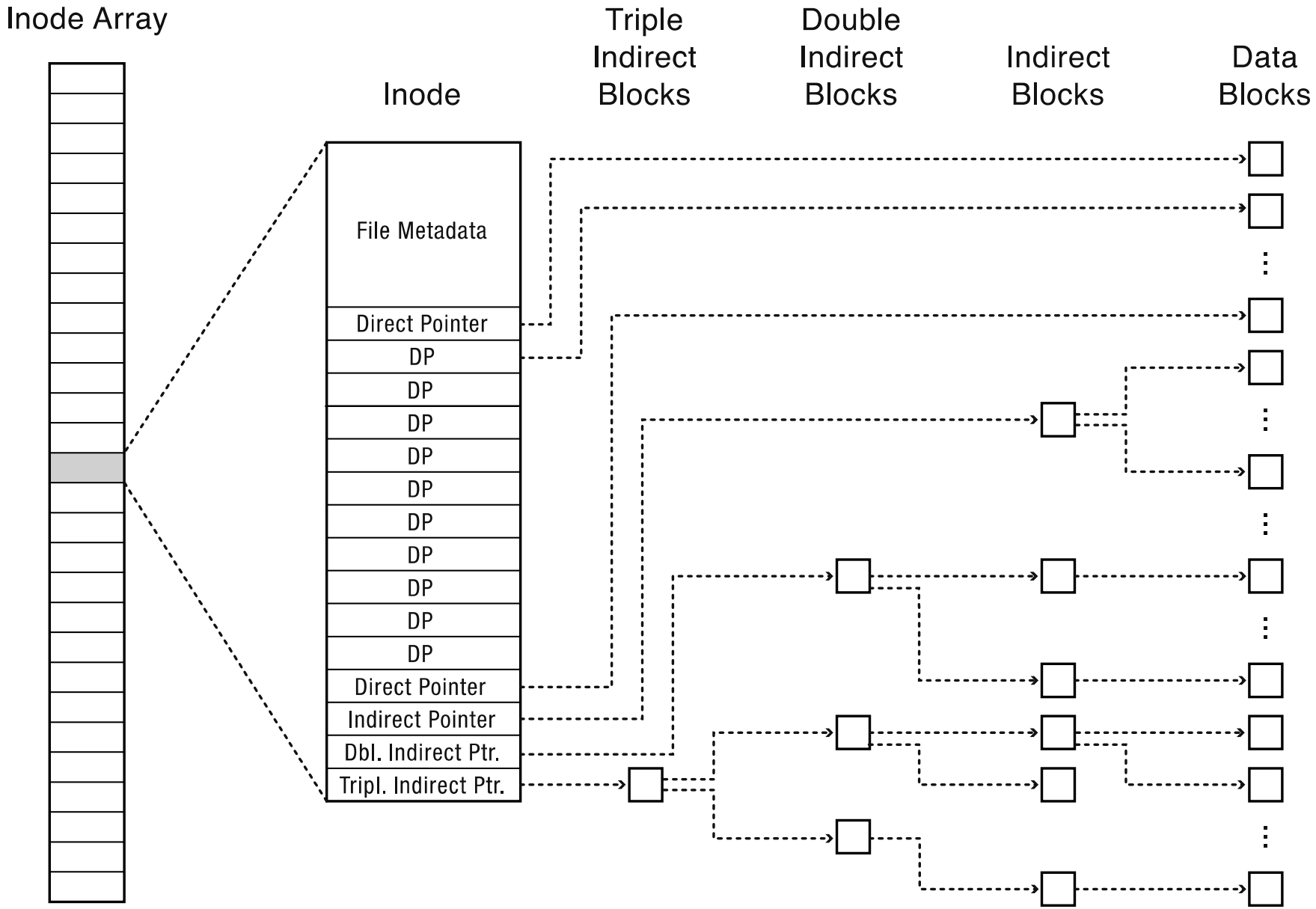


UFS I-Nodes

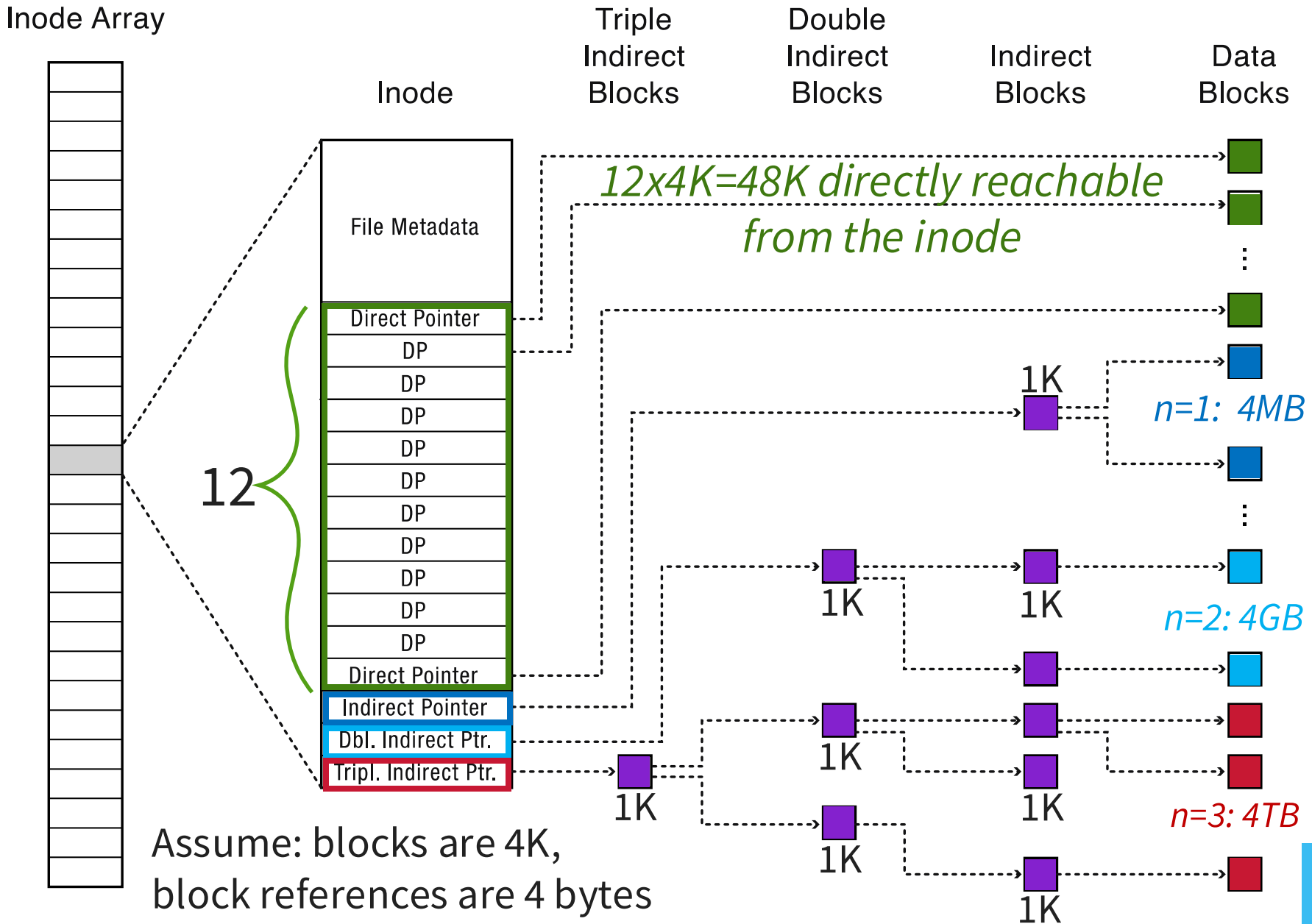
- inode array
  - inode
    - Metadata
    - 12 (or so) direct pointers
    - 3 indirect pointers



UFS: Index Structures

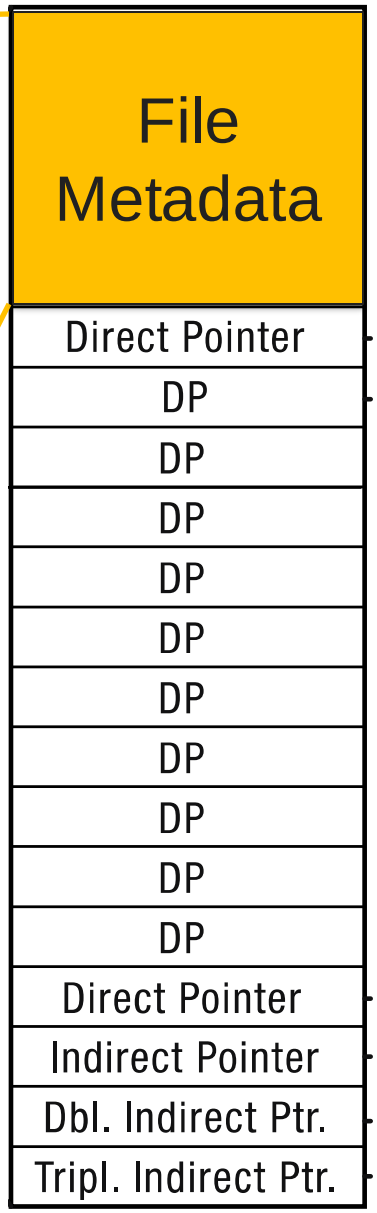


UFS: Index Structures



What else is in an inode?

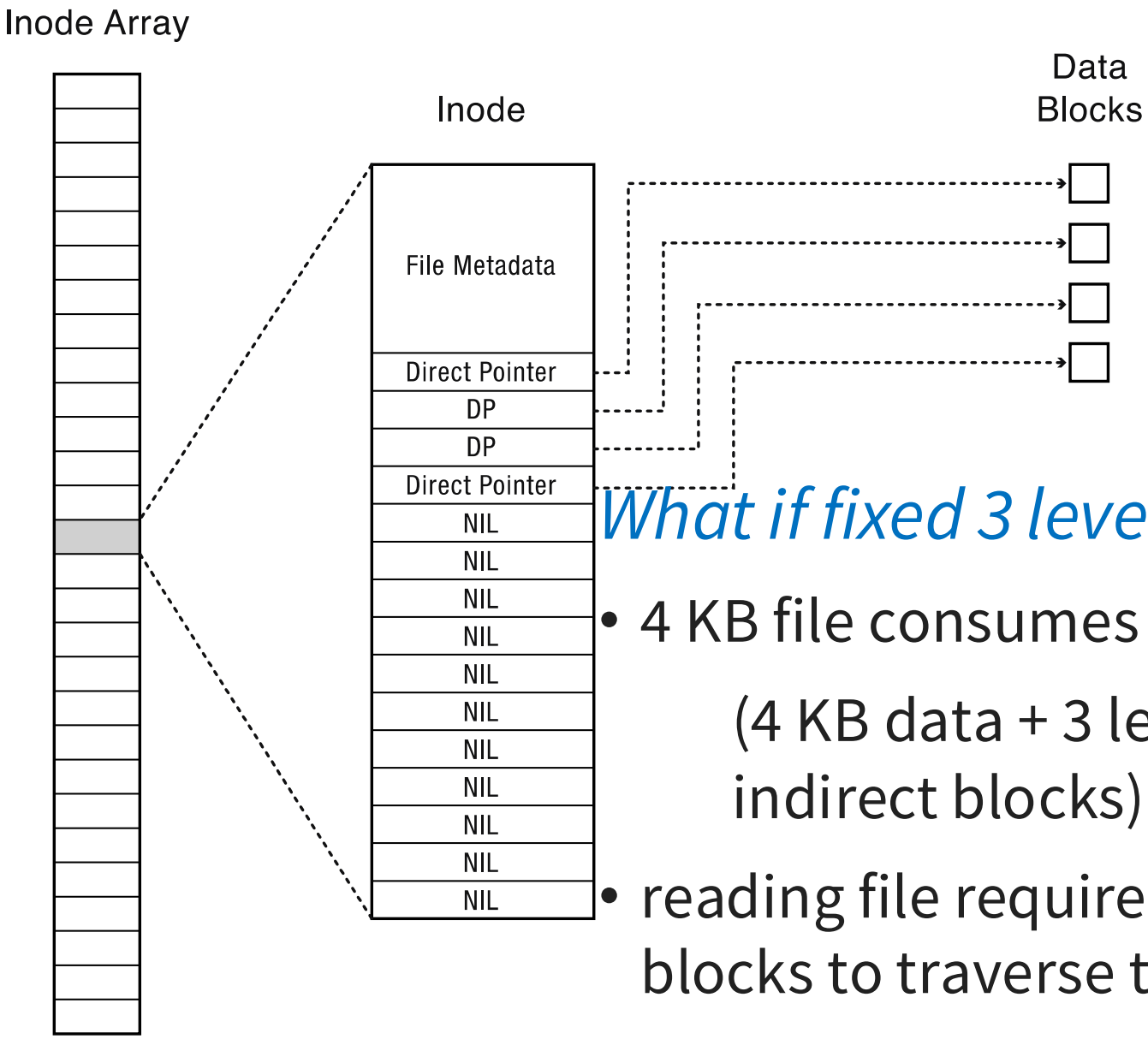
- Type
  - ordinary file
  - directory
  - symbolic link
  - special device
- Size of the file (in #bytes)
- # links to the i-node
- Owner (user id and igroupd)
- Protection bits
- Times: creation, last accessed, last modified



4 Characteristics of UFS

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 UFS



all blocks reached via direct pointers

What if fixed 3 levels instead?

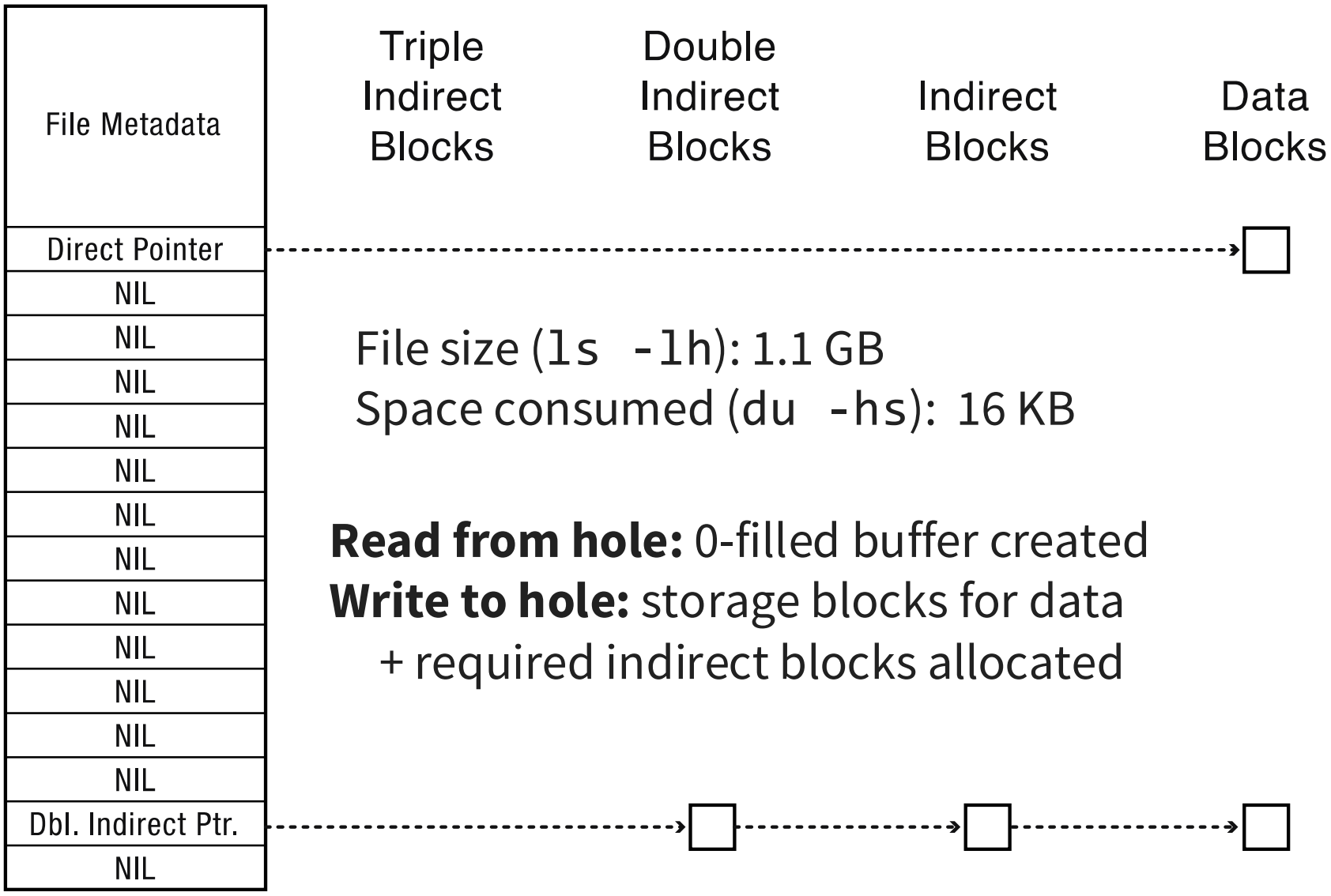
- 4 KB file consumes ~16 KB

  (4 KB data + 3 levels of 4KB indirect blocks)

- reading file requires reading 5 blocks to traverse tree

Sparse Files in UFS

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



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

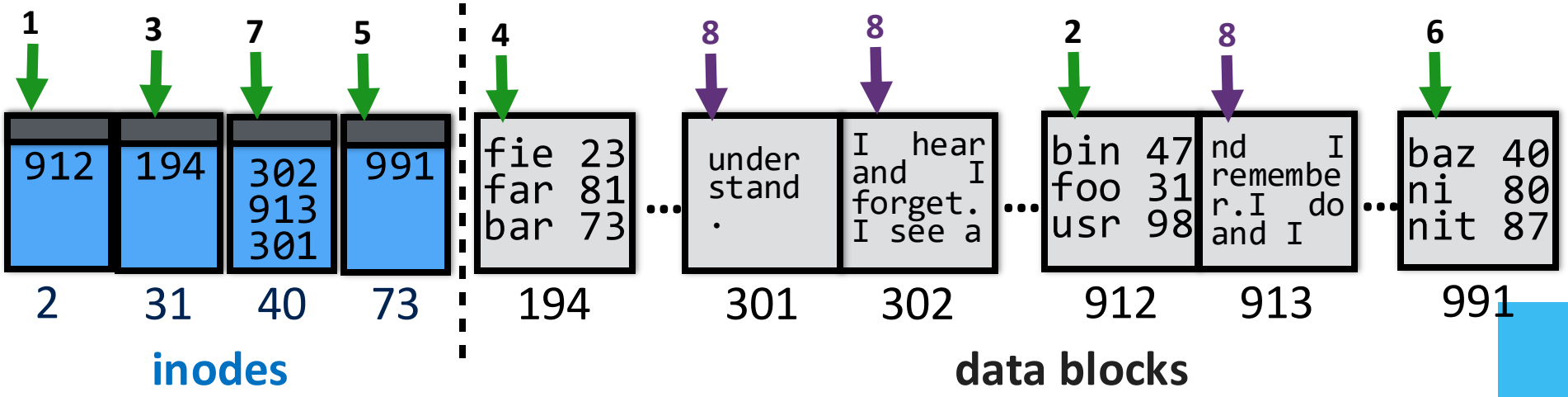
Read from hole: 0-filled buffer created
Write to hole: storage blocks for data
+ required indirect blocks allocated

UFS: Steps to reading /foo/bar/baz

Read & Open:

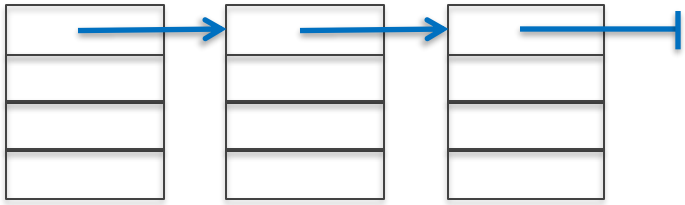
(1) inode #2 (root has inumber 2), find root's blocknum (912)
(2) root directory (in block 912), find foo's inumber (31)
(3) inode #31, find foo's blocknum (194)
(4) foo (in block 194), find bar's inumber (73)
(5) inode #73, find bar's blocknum (991)
(6) bar (in block 991), find baz's inumber (40)
(7)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

Caching often allows first few steps to be skipped

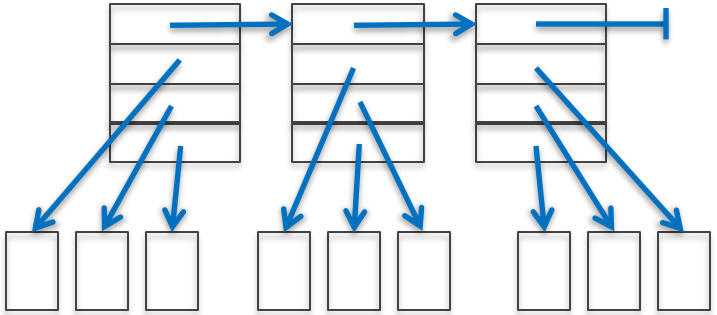


Free List

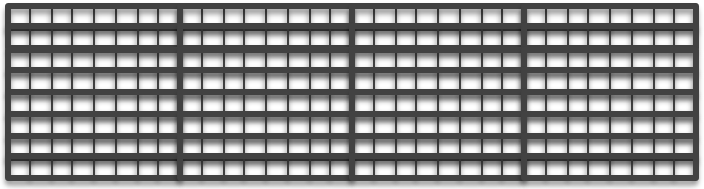
- List of blocks not in use
- How to maintain?
  1. linked list of free blocks
     - inefficient (why?)



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



  3. bitmap
     - good for contiguous allocation

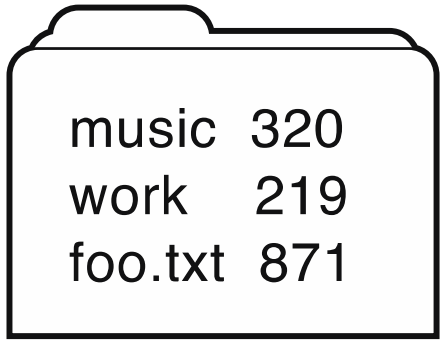


File System API: Creation

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

UFS Directory Structure



Originally: array of 16 byte entries

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

Now: linked lists. Each entry contains:

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

First entry is ".", points to self

Second entry is "..", points to parent inode

Hard & Soft Links

- a mapping from each name to a specific underlying file or directory (hard link)
- a **soft link** is instead a mapping from a file name to another file name
  - it's simply a file that contains the name of another file
  - use as *alias*: a soft link that continues to remain valid when the (path of) the target file name changes

File System Consistency

System crashes before modified files written back?

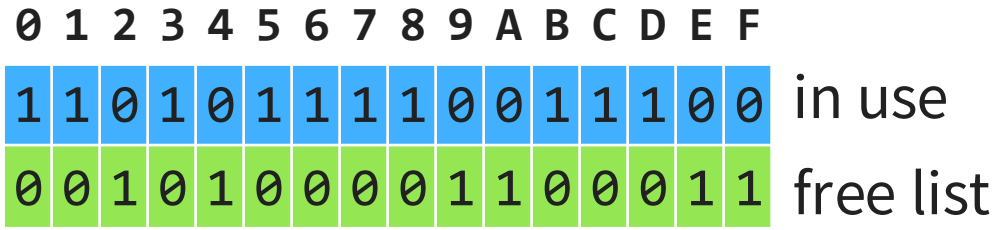
- Leads to inconsistency in FS
- fsck (UNIX) & scandisk (Windows) check FS consistency

Algorithm:

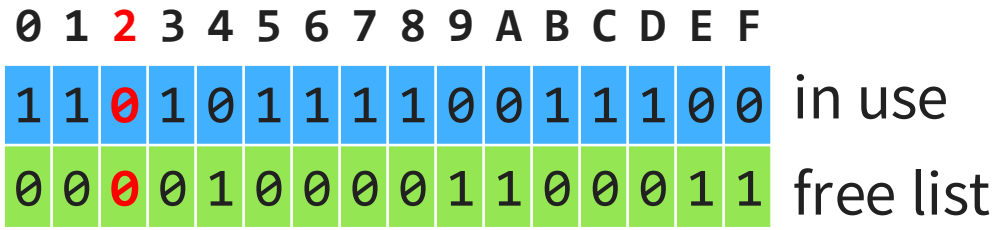
- Build table with info about each block
  - initially each block is unknown except superblock
- Scan through the inodes and the freelist
  - Keep track in the table
  - If block already in table, note error
- Finally, see if all blocks have been visited

Inconsistent FS Examples

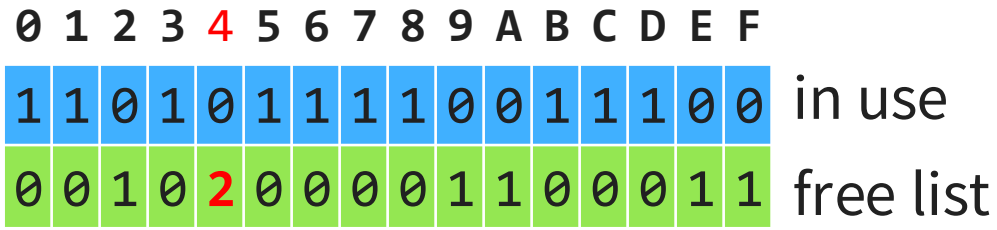
Consistent



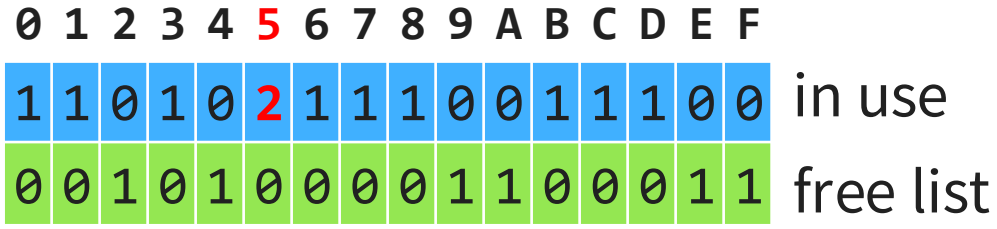
Missing Block 2
(add it to the free list)



Duplicate Block 4 in Free List
(rebuild free list)



Duplicate Block 4 in Data List (copy block and add it to one file)



Check Directory System

Use a per-file table instead of per-block

Parse entire directory structure, start at root

- Increment counter for each file you encounter
- This value can be >1 due to hard links
- Symbolic links are ignored

Compare table counts w/link counts in i-node

- If i-node count ≠ our directory count
  - Fix i-node count both larger than 0
  - If i-node count = 0, i-node is free
    - remove the corresponding directory entries
  - If directory-count = 0, no links to the i-node
    - add to “lost+found” directory under unique name