File Systems

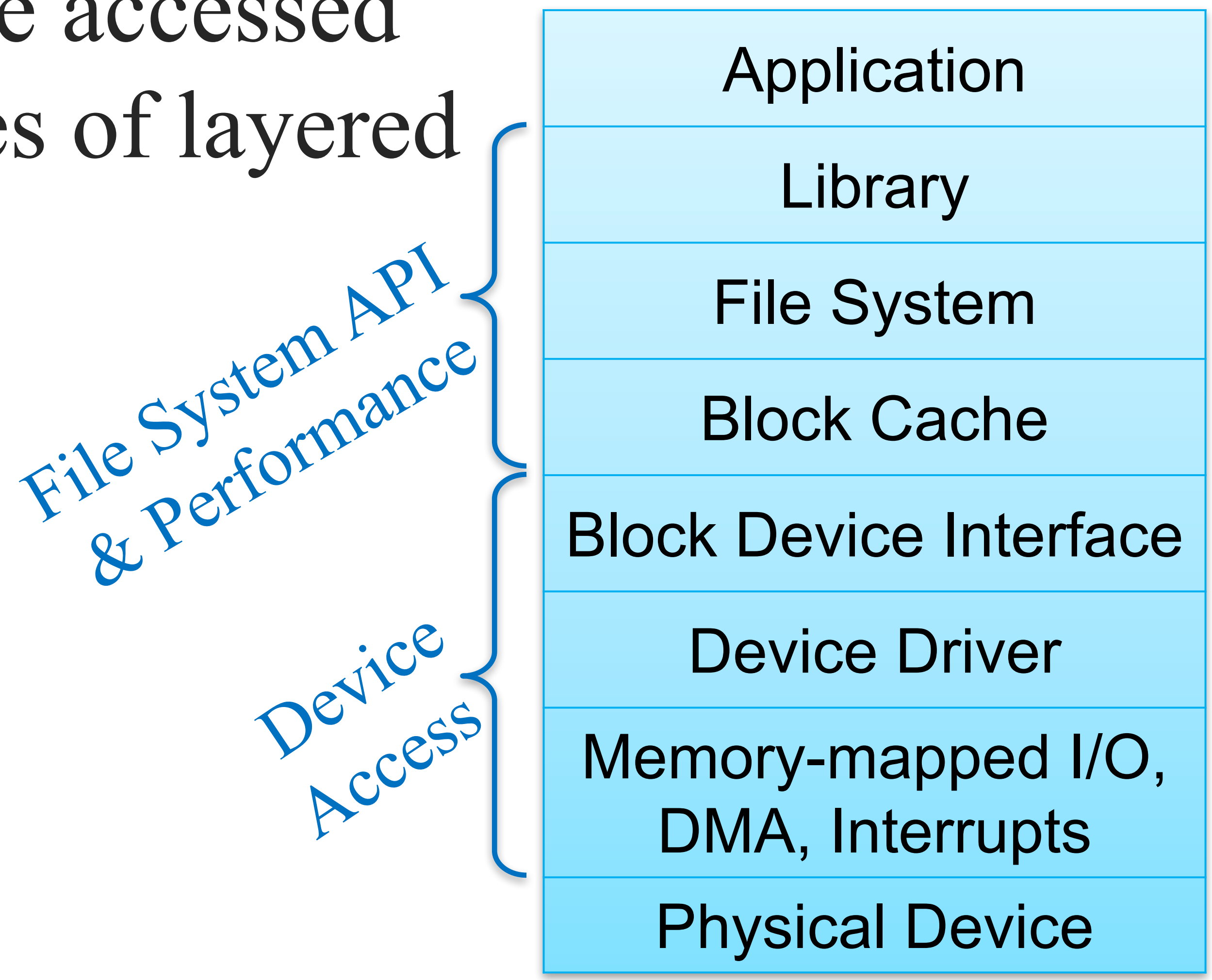
CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

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

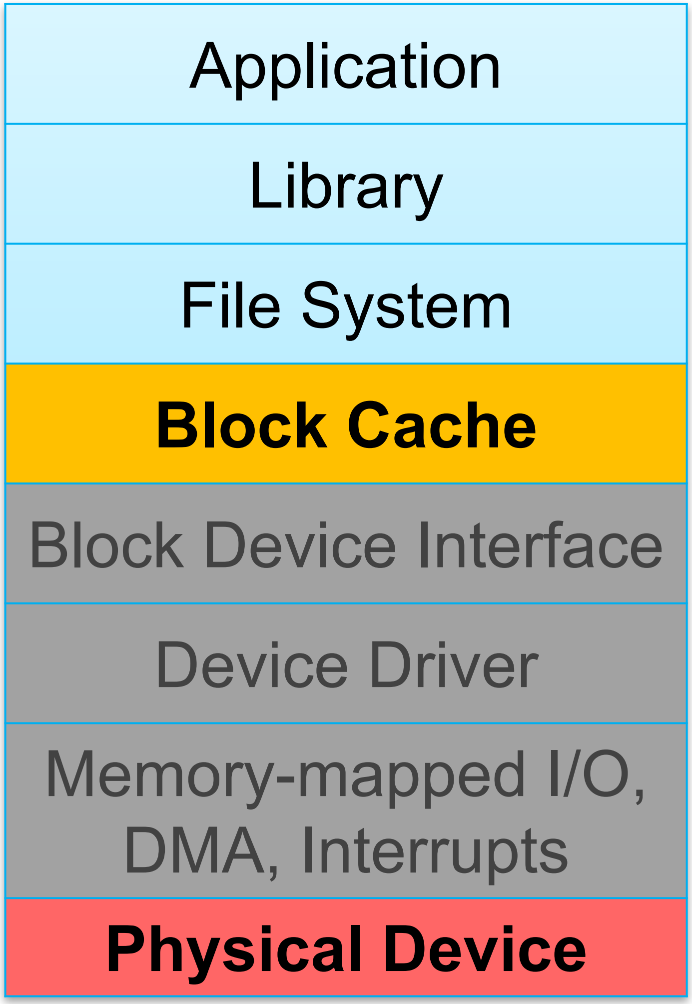
The abstraction stack

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
- serves as synchronization point (ensures a block is only fetched once)

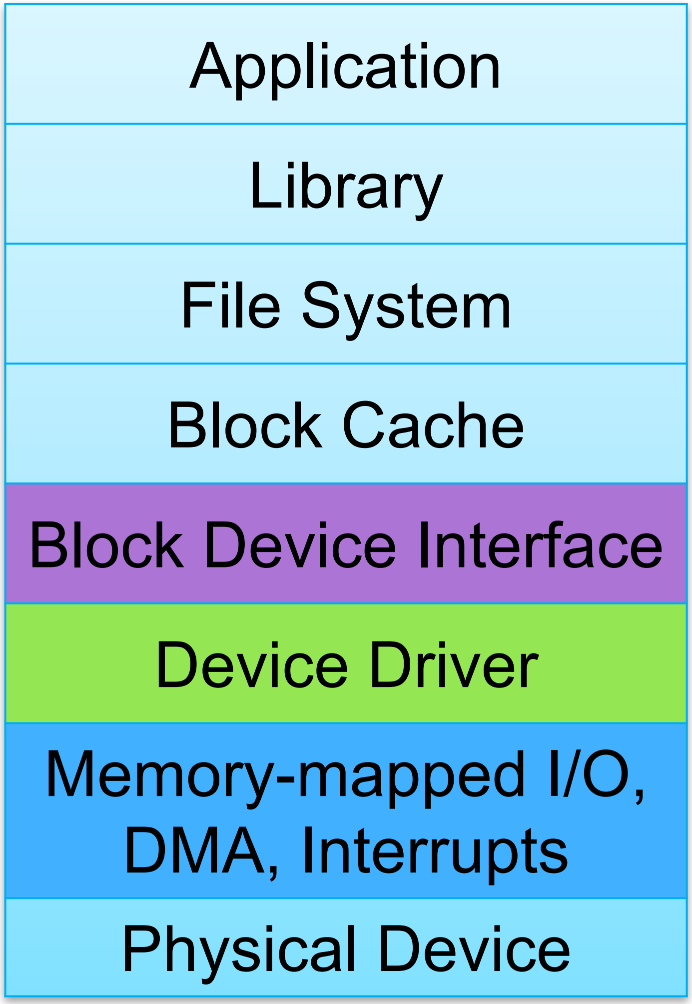


More Layers *(not a 4410 focus)*

- 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



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
    - array of untyped bytes
  - metadata – information added and managed by the OS
    - size, owner, security info, modification time

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, UNIX is
- Some things common:
  Usually ok up to 255 characters

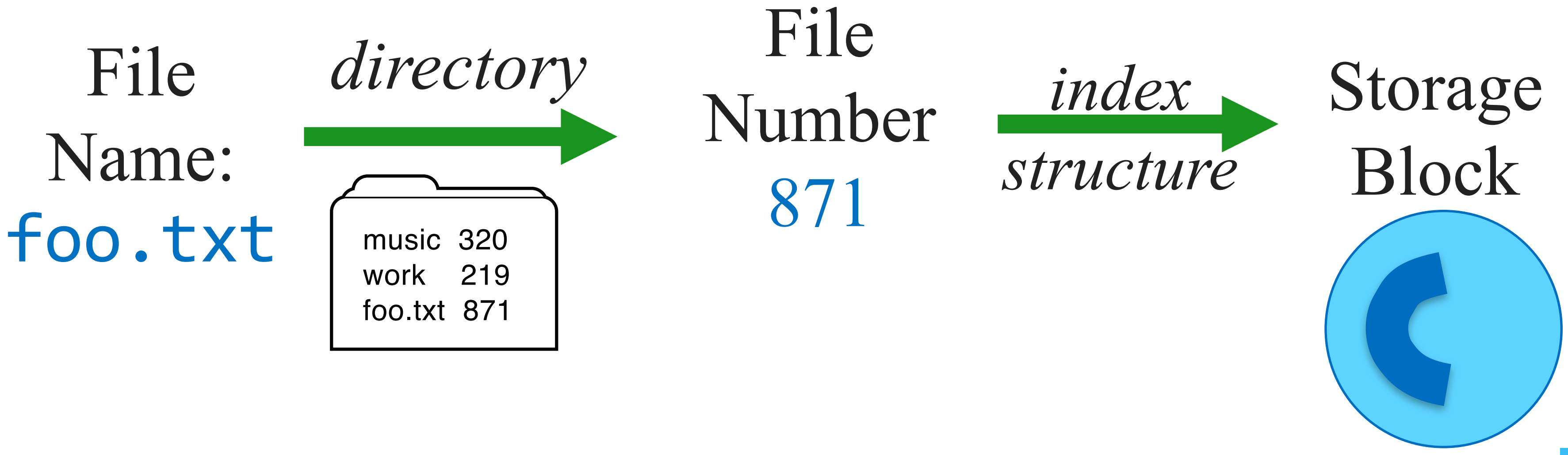
File Extensions, OS dependent:

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

Directory

Directory: provides names for files

- a list of human readable names
- 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 current working directory

(current working dir stored in process' PCB)

2 special entries in each UNIX directory:

"." current dir

".." for parent

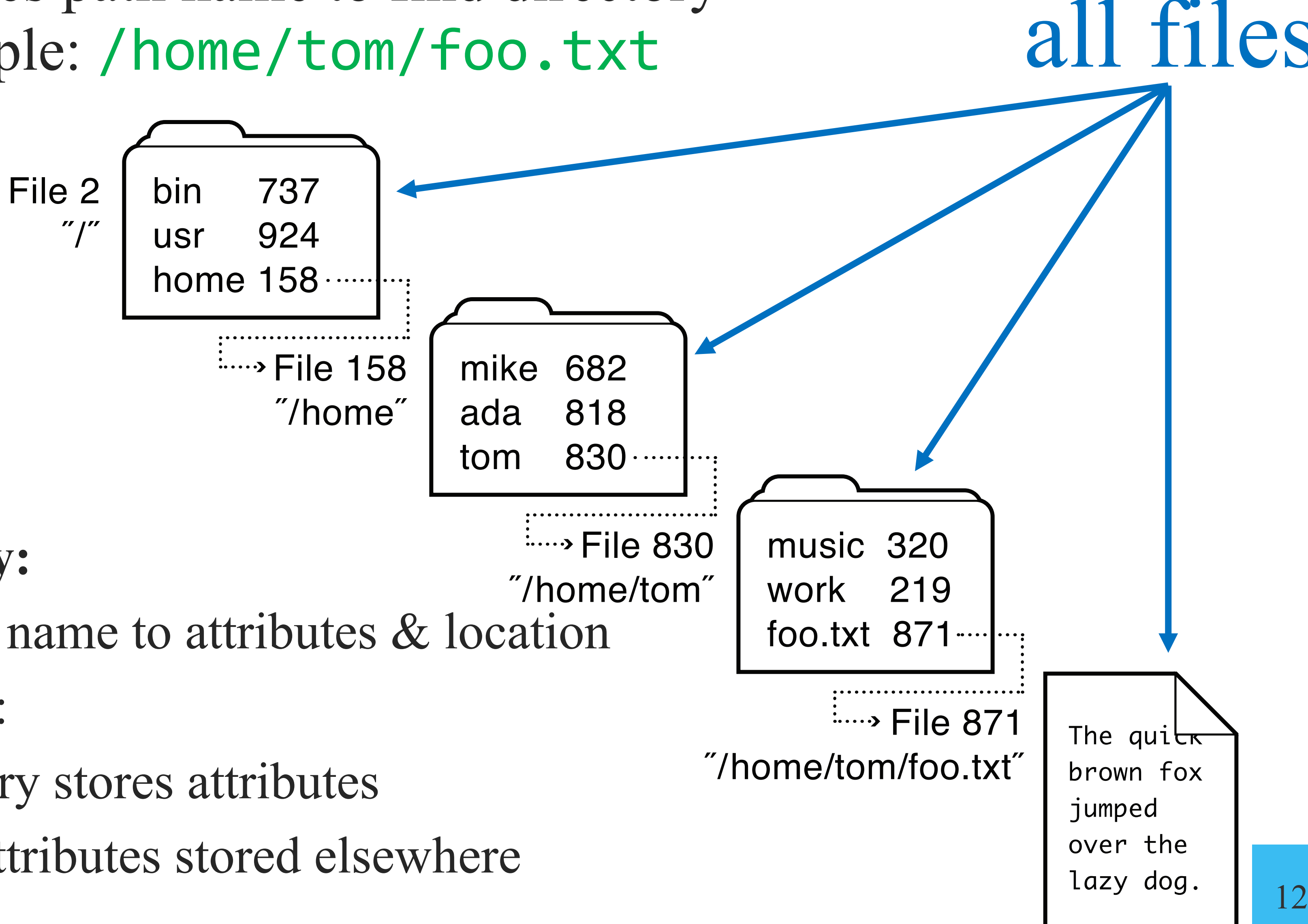
To access a file:

- 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:

- directory stores attributes
- files' attributes stored elsewhere

Basic File System Operations

- Create a file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Delete a file
- Truncate a file

How shall we implement this?

Just map keys (file names) to values (block numbers on disk)?

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 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

Implementation Basics

Directories

- file name ➜ file number

Index structures

- file number ➜ block

Free space maps

- find a free block; better: find a free block *nearby*

Locality heuristics

- policies enabled by above mechanisms
  - group directories
  - make writes sequential
  - defragment

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

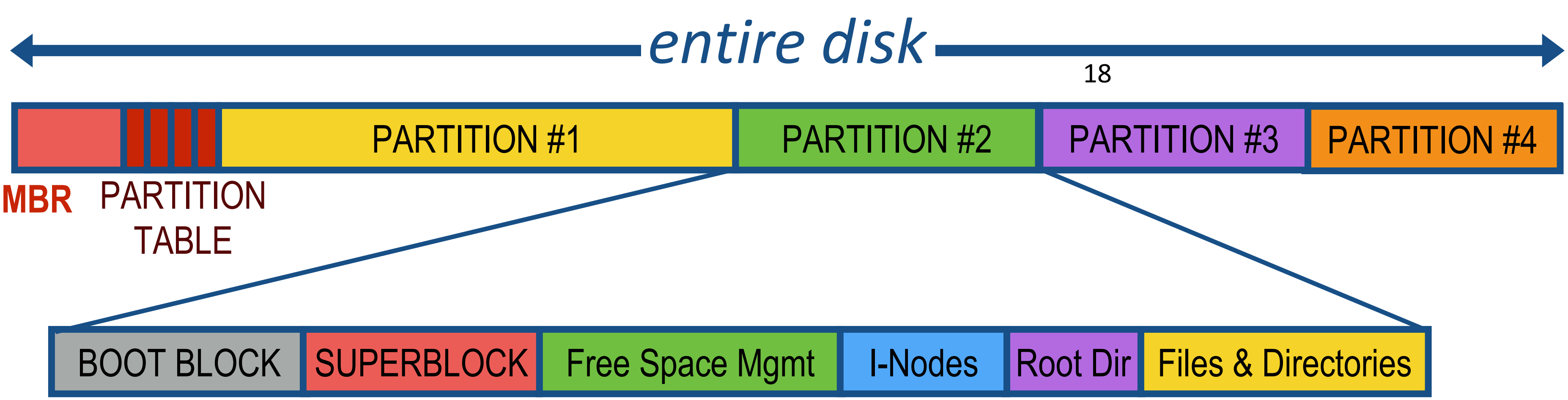
File System Layout

File System is stored on *disks*

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

First block of each partition has *boot block*

- loaded by MBR and executed on boot



Storing Files

Files can be allocated in different ways:

- Contiguous allocation

    All bytes together, in order

- Linked Structure

    Each block points to the next block

- Indexed Structure

    Index block points to many other blocks

Which is best?

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

Contiguous Allocation

All bytes together, in order

+ **Simple:** state required per file: start block & size

+ **Efficient:** entire file can be read with one seek

– **Fragmentation:** external is bigger problem

– **Usability:** user needs to know size of file at time of creation

Used in CD-ROMs, DVDs

Linked List Allocation

Each file is stored as linked list of blocks

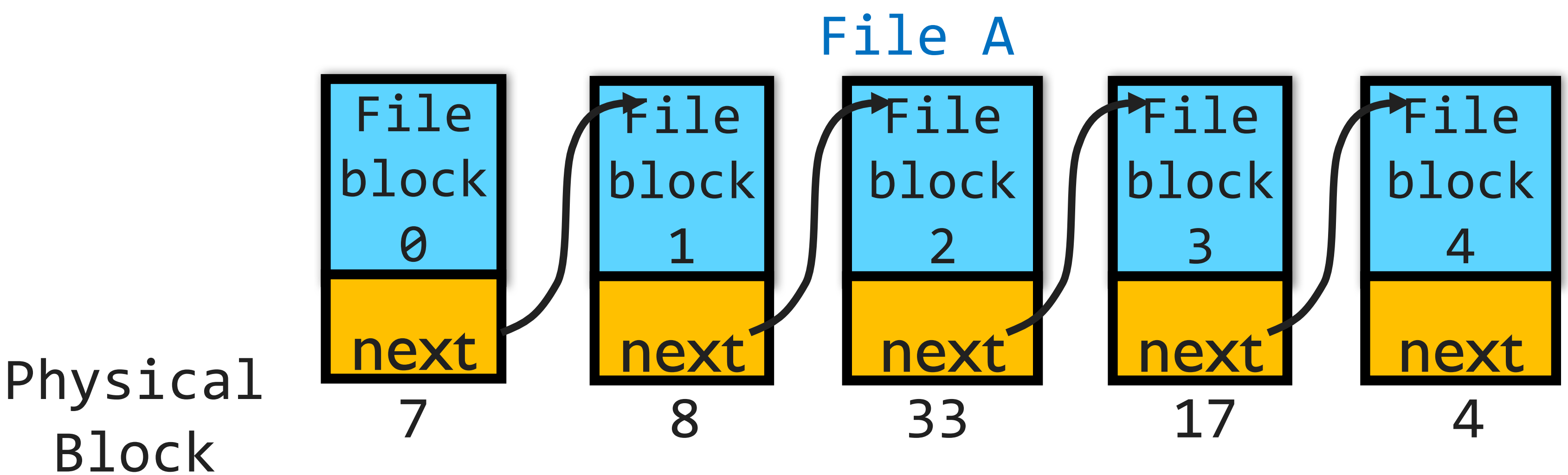
- First word of each block points to next block
- Rest of disk block is file data

+ **Space Utilization:** no space lost to external fragmentation

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

– **Performance:** random access is slow

– **Space Utilization:** overhead of pointers

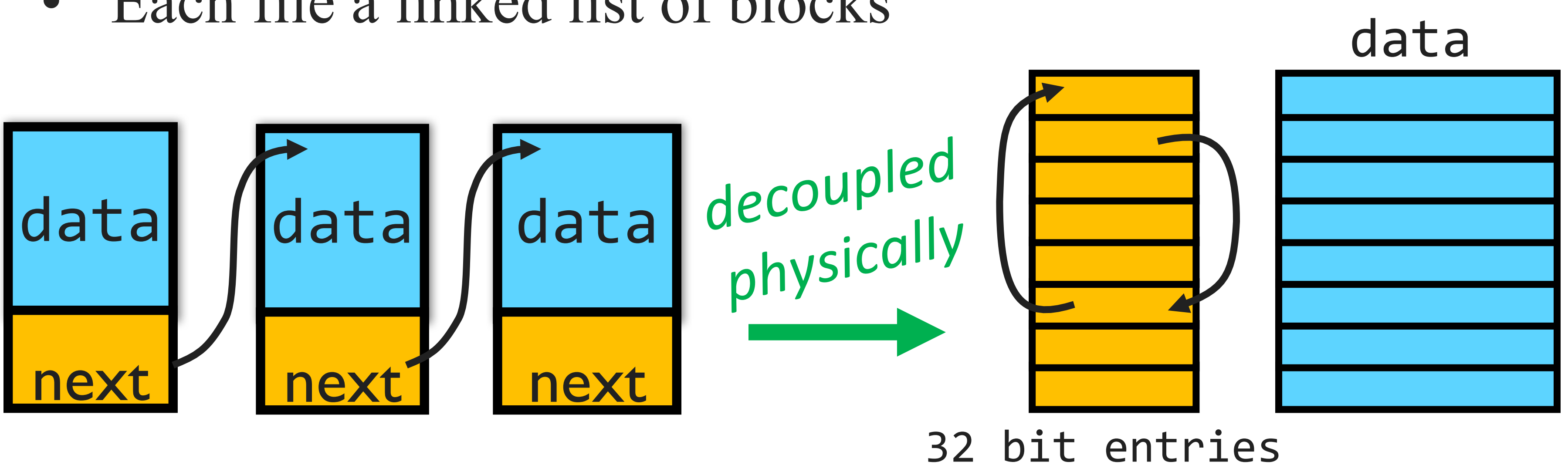


File Allocation Table (FAT) FS

[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)
- FAT-32, supports 2^{28} blocks and files of 2^{32}-1 bytes

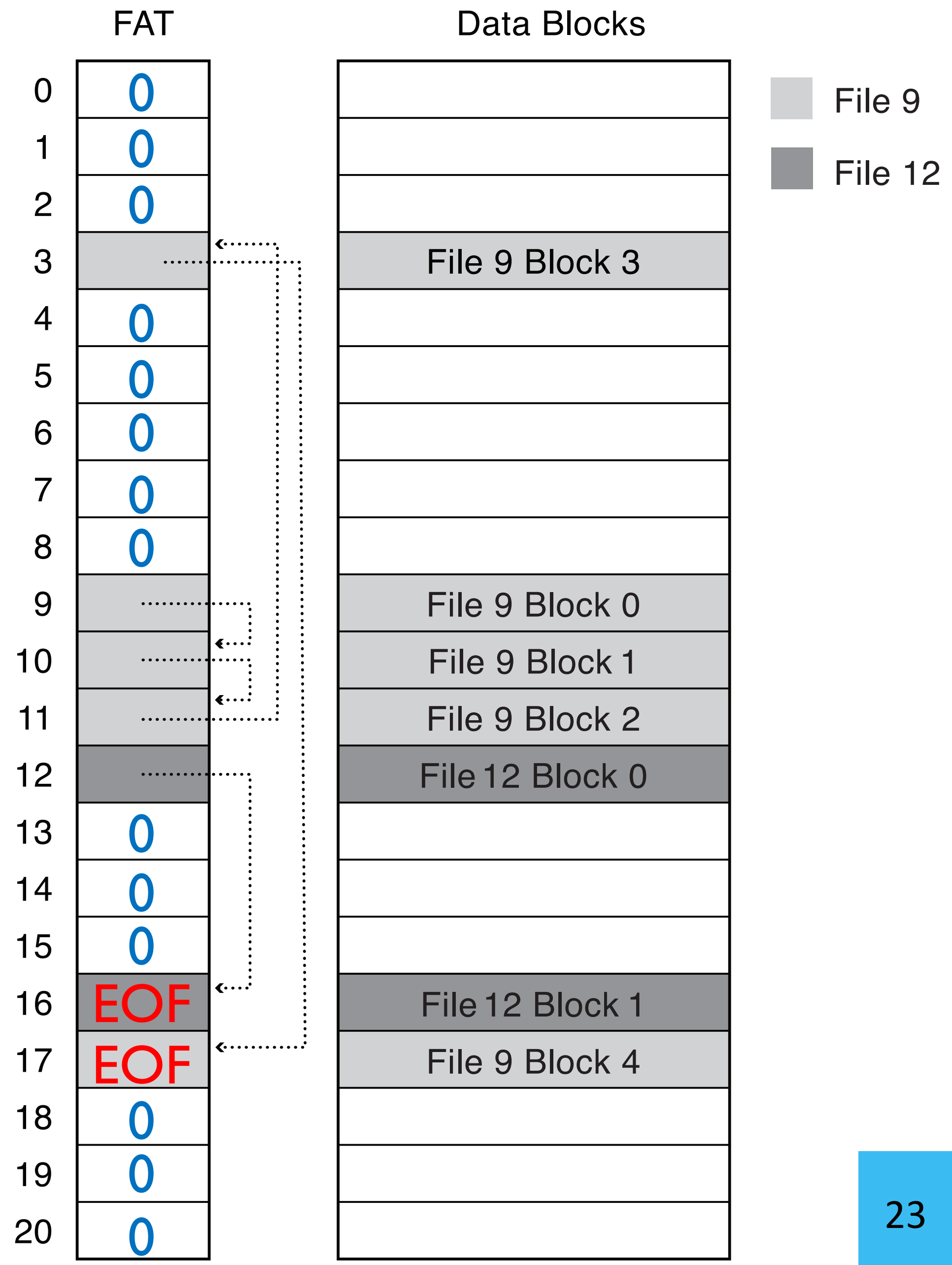
File table:
- Linear map of all blocks on disk
- Each file a linked list of blocks



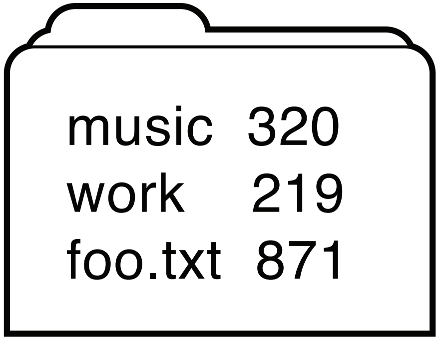
FAT File System

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

Directory	
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
- 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

Fast File System (FFS)

[mid 80's]

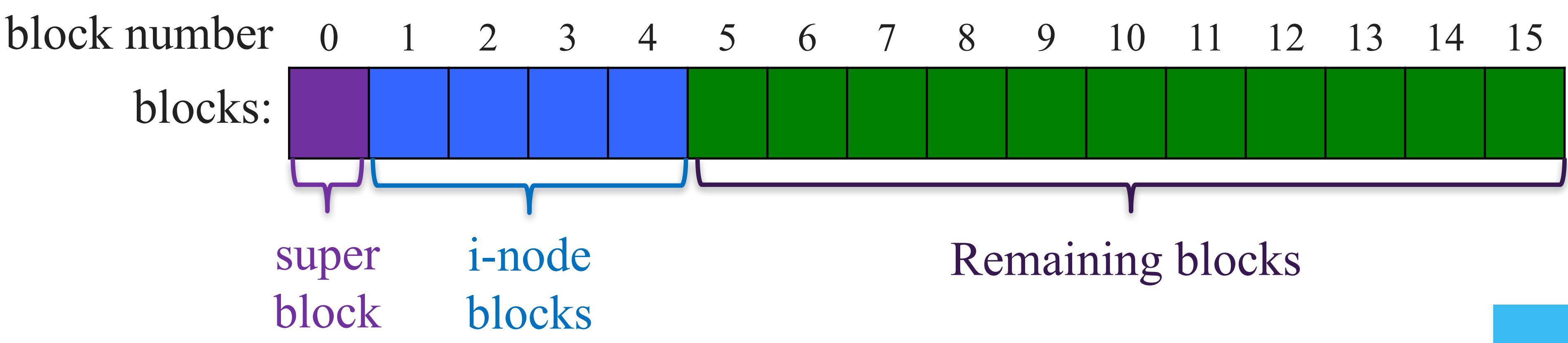
UNIX Fast File System

Tree-based, multi-level index

FFS Superblock

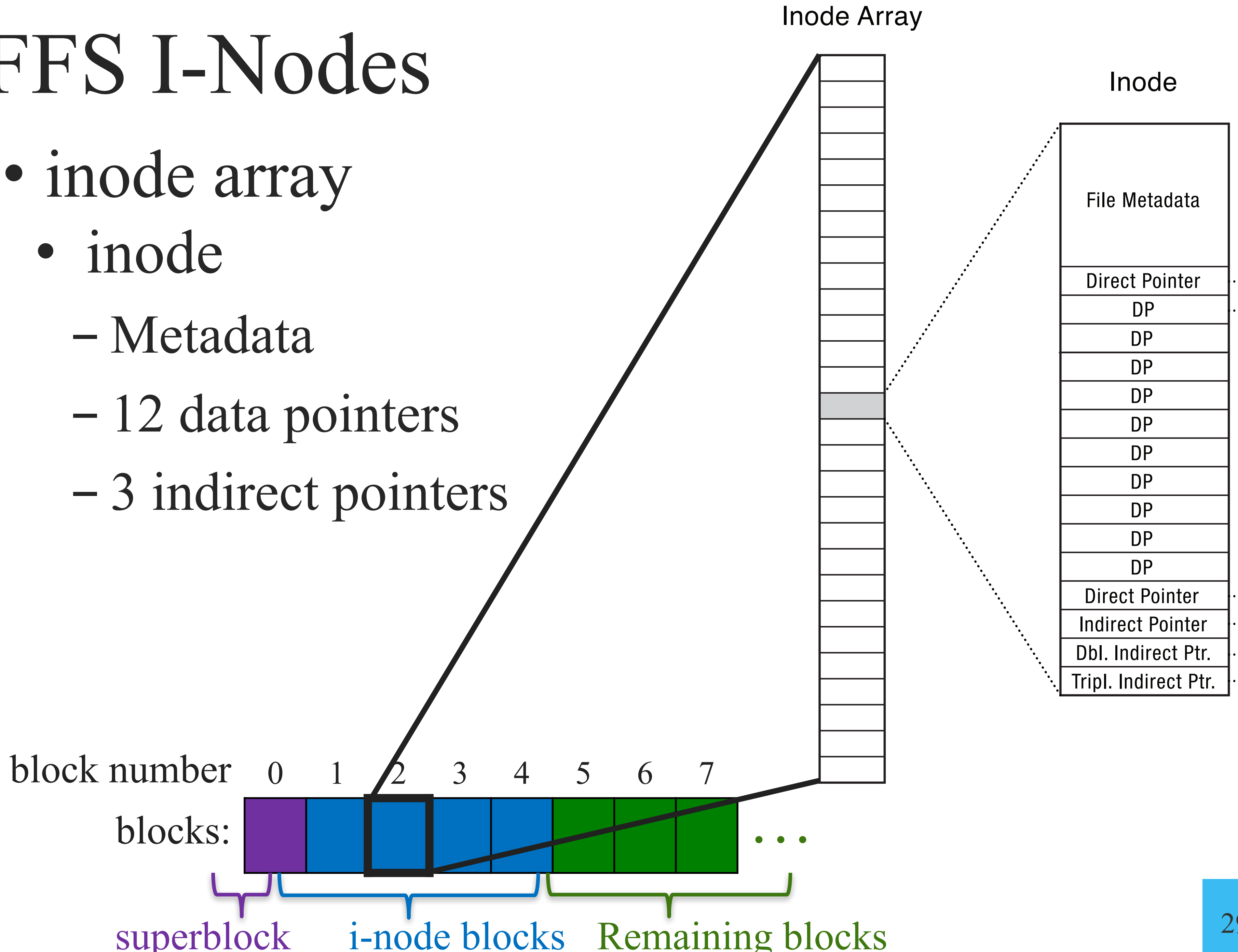
Identifies file system's key parameters:

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

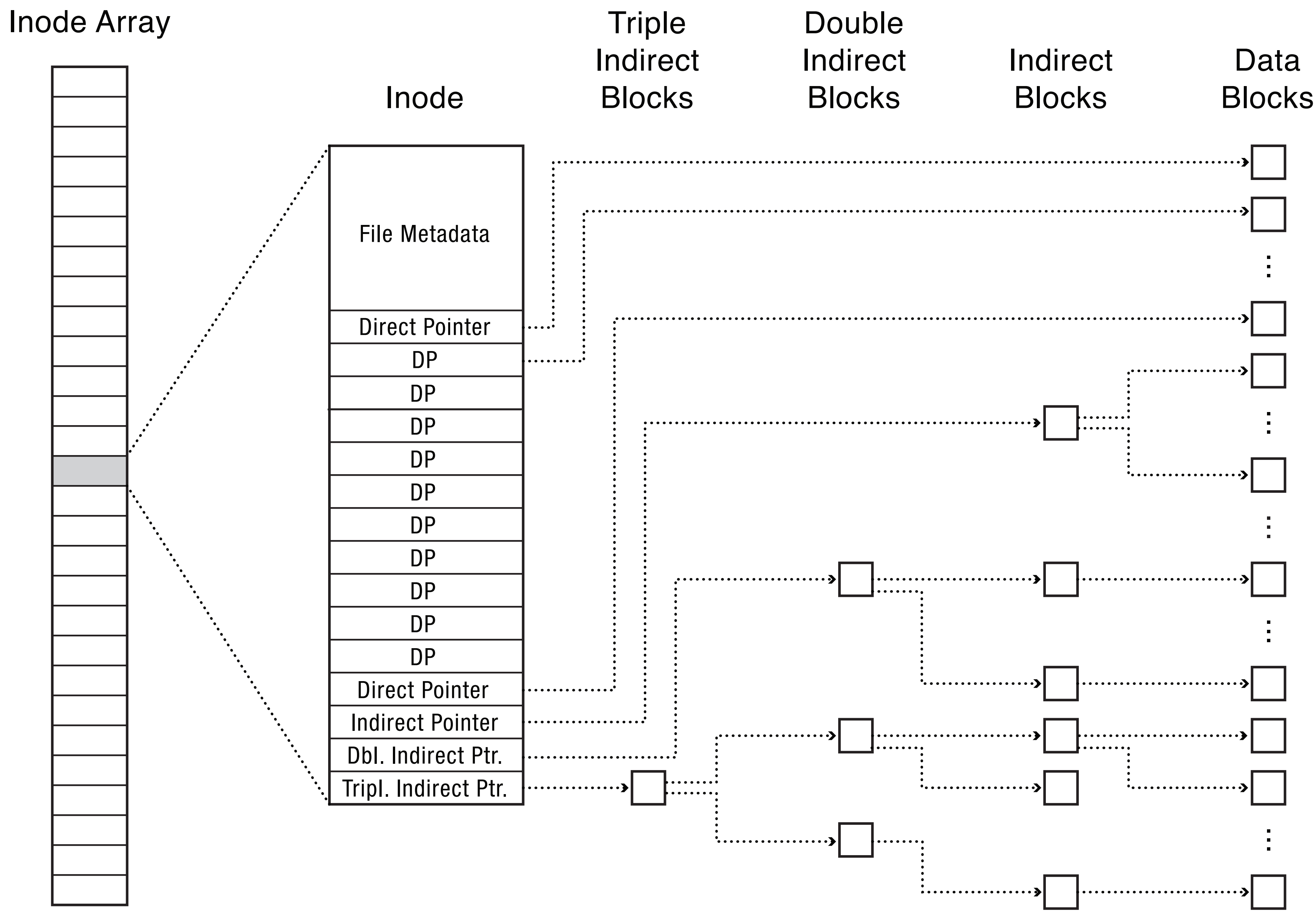


FFS I-Nodes

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

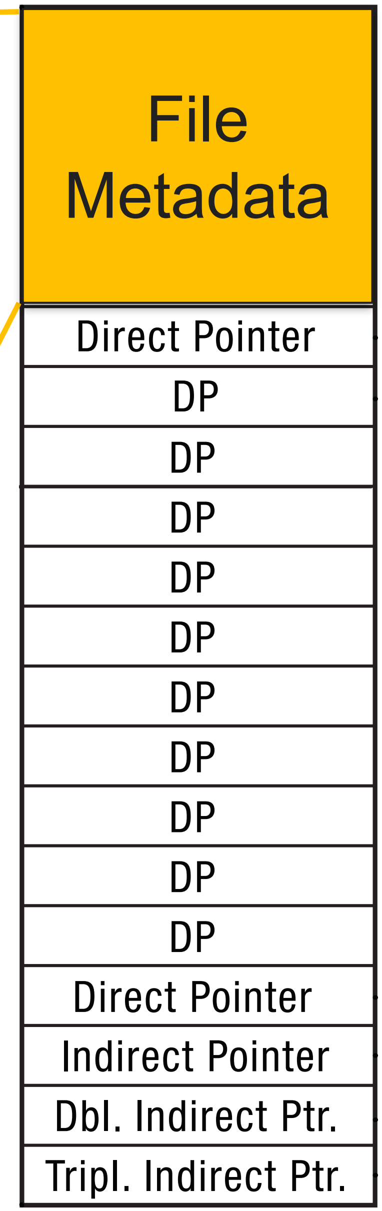


FFS: Index Structures

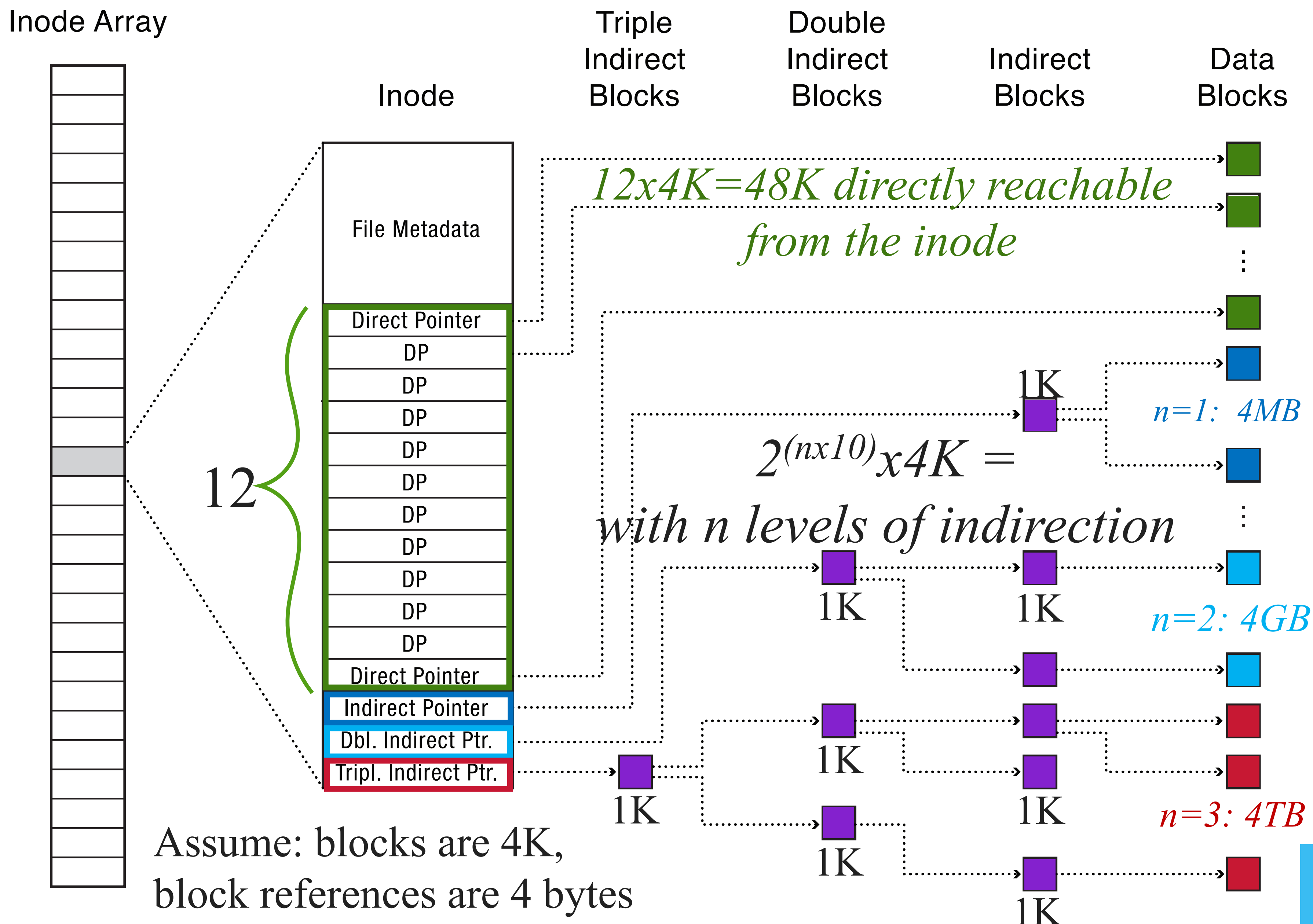


What else is in an inode?

- 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



FFS: Index Structures

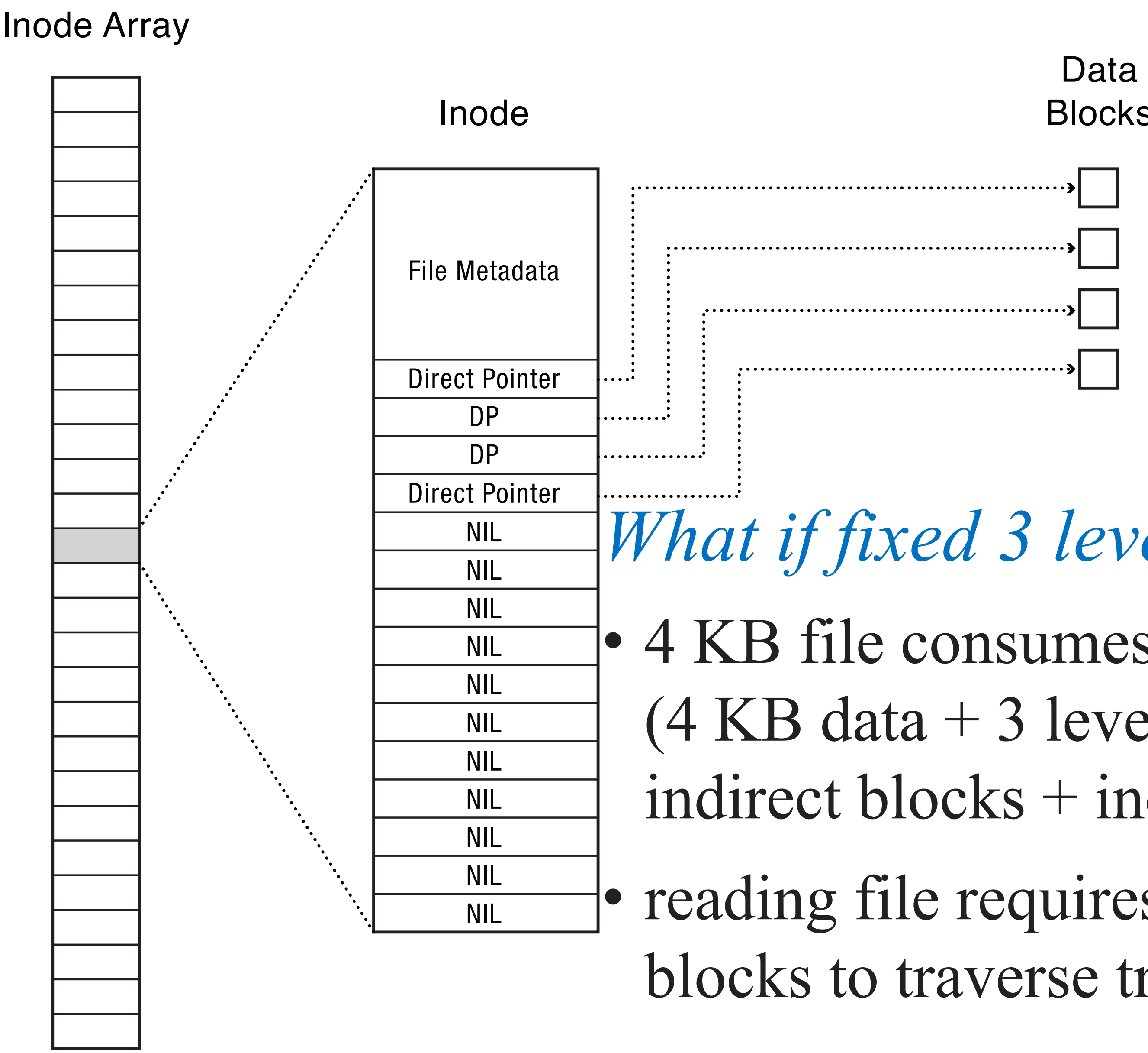


Assume: blocks are 4K, block references are 4 bytes

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

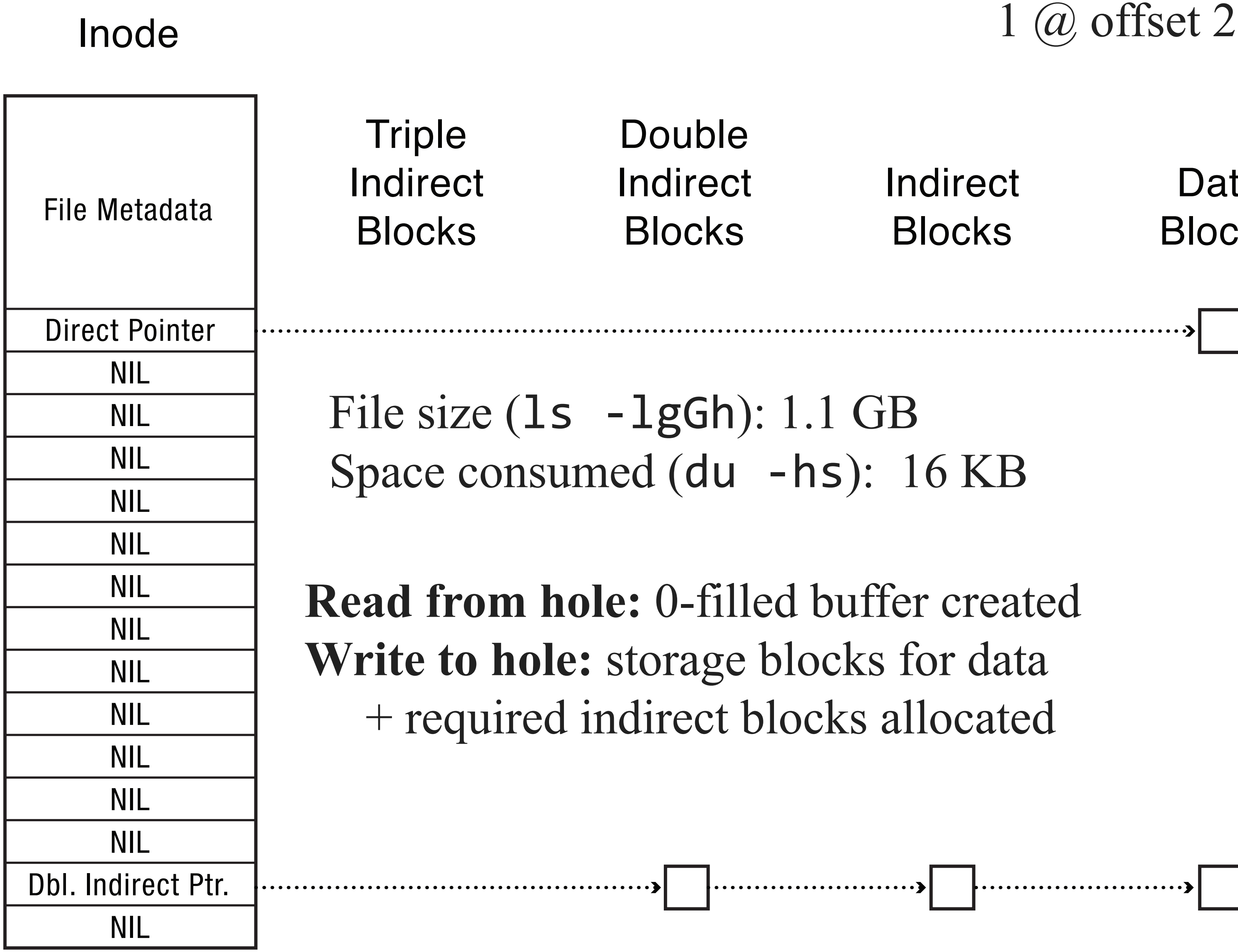
What if fixed 3 levels instead?

- 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

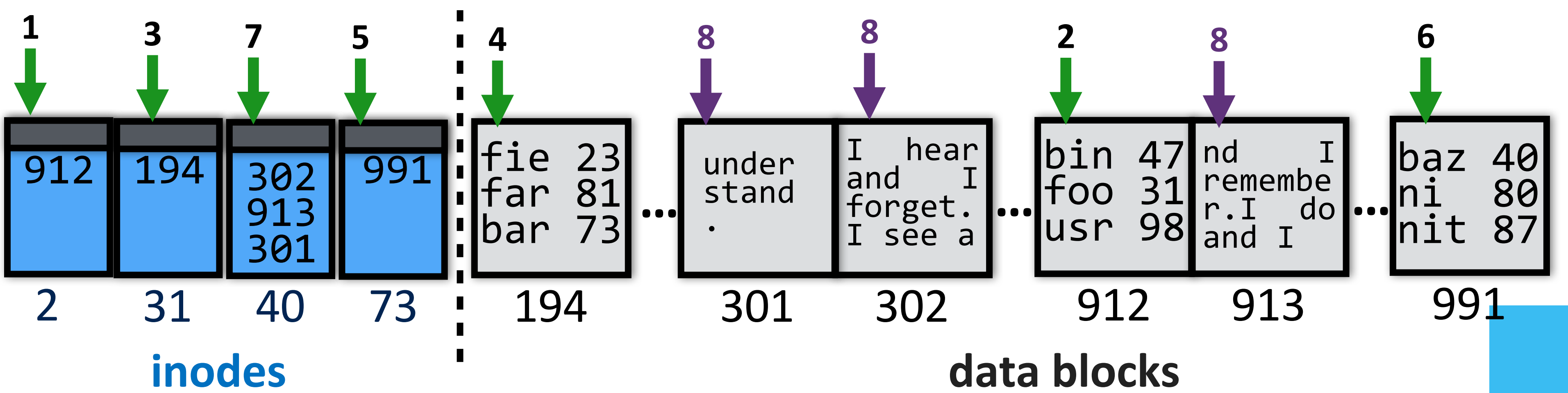
Read from hole: 0-filled buffer created
Write to hole: storage blocks for data
+ required indirect blocks allocated

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

Read & Open:

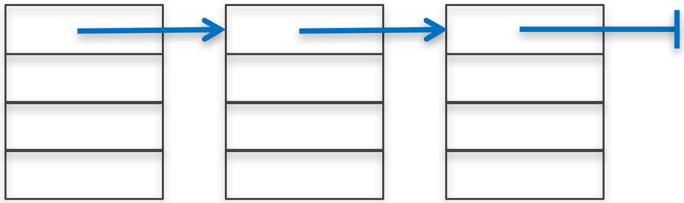
(1) inode #2 (root always has inumber 2), find root's blocknum (912)
(2) root directory (in block 912), find foo's inumber (31)
(3) inode #31, find foo's blocknum (194)
(4) foo (in block 194), find bar's inumber (73)
(5) inode #73, find bar's blocknum (991)
(6) bar (in block 991), find baz's inumber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

Caching allows first few steps to be skipped

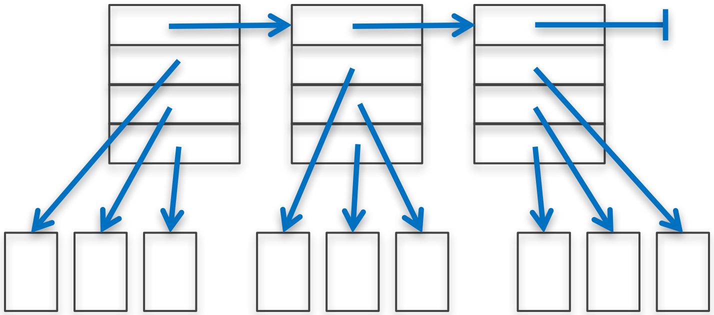


Free List

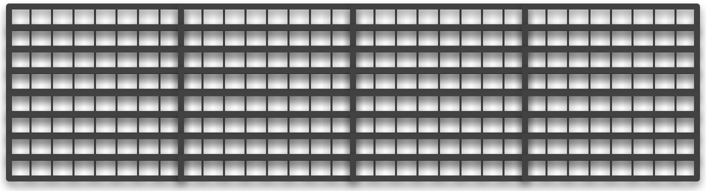
- List of blocks not in use
- How to maintain?
  1. linked list of free blocks
     - inefficient (why?)



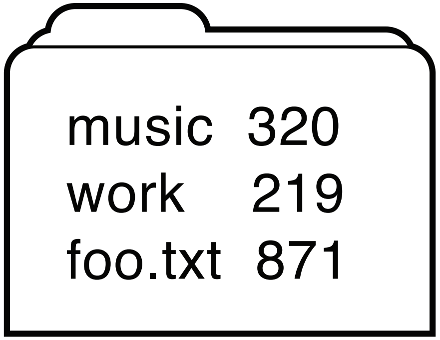
  2. linked list of metadata blocks that in turn point to free blocks
     - simple and efficient



  3. bitmap
     - good for contiguous allocation



FFS 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

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

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
free list	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

Missing Block 2
(add it to the free list)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
free list	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1

Duplicate Block 4 in Free List
(rebuild free list)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
free list	0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
in use	1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0
free list	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

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 (wastes space)
- If i-node count < our directory count (catastrophic)