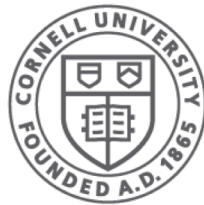




File Systems

CS 4410
Operating Systems



Cornell CIS
COMPUTING AND INFORMATION SCIENCE

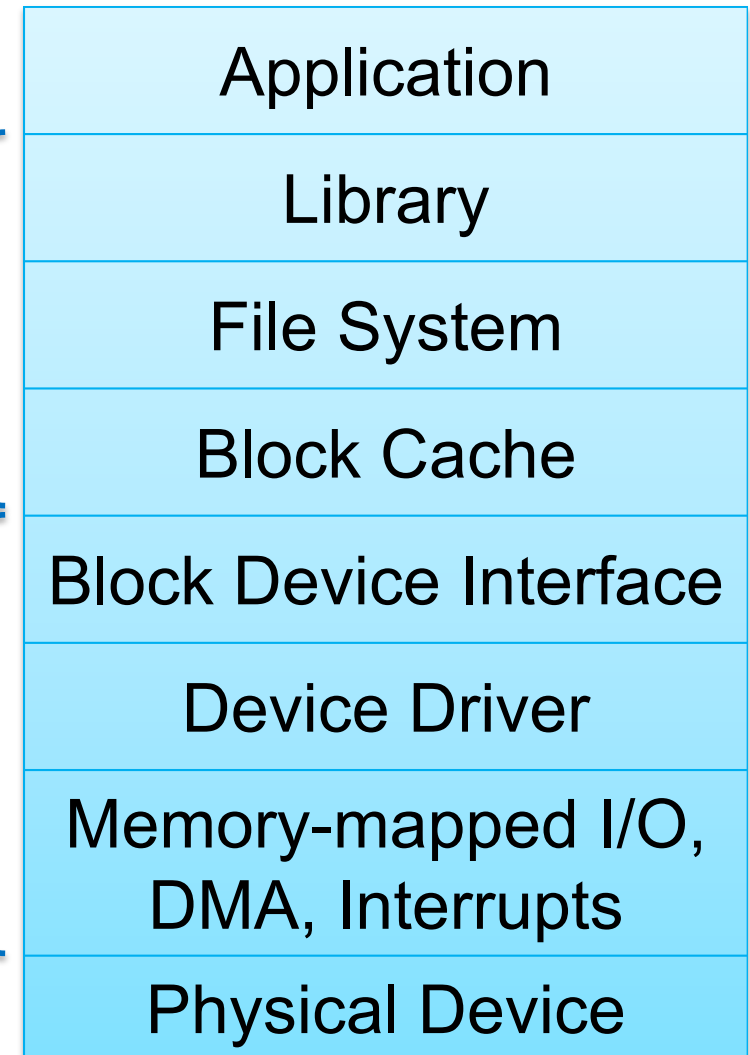
[R. Agarwal, L. Alvisi, A. Bracy, M. George, Kurose, Ross, E. Sirer, R. Van Renesse]

The abstraction stack

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

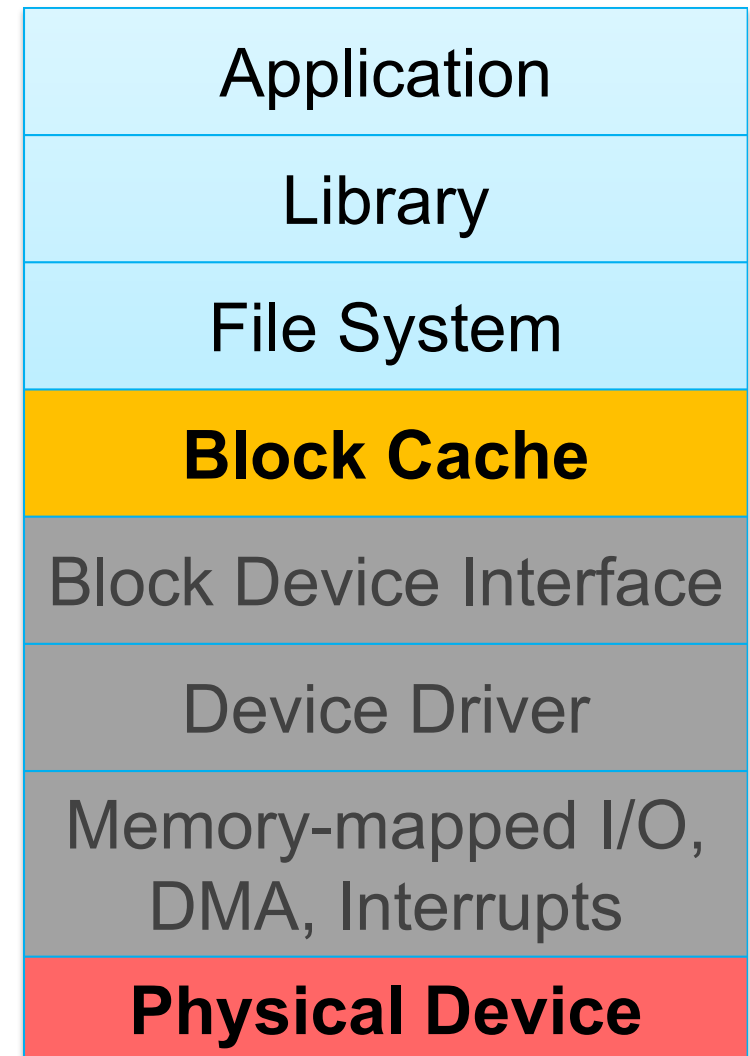
File System API
& Performance

Device
Access



The Block Cache

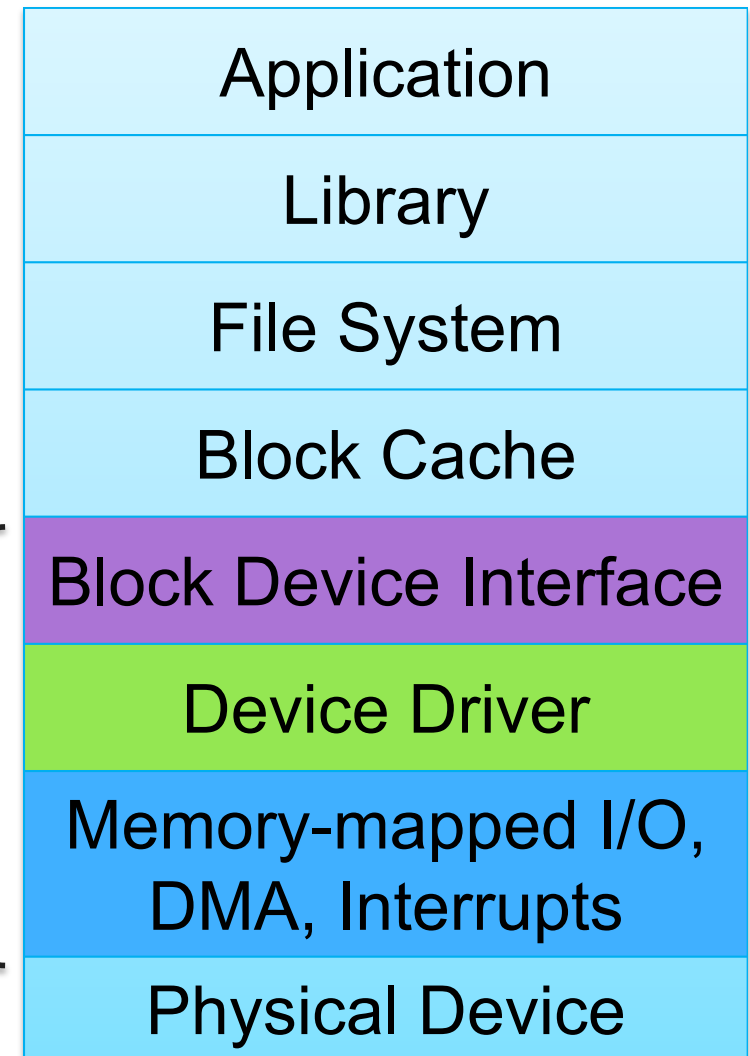
- a **cache** for the **disk**
- caches recently read blocks
- buffers recently written blocks
- serves as synchronization point (ensures a block is only fetched once)
- Big part of A4



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

ignored
in A4



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 to multiple processes

Solution: the File System Abstraction

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

The File

- **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

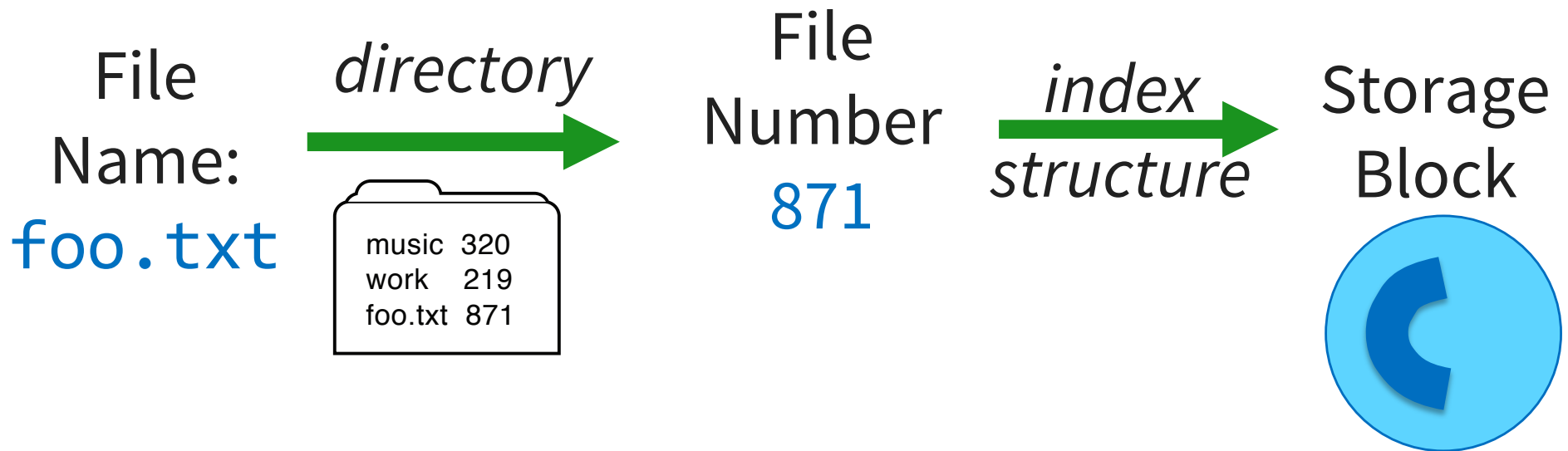
File Extensions, OS dependent:

- 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 work dir stored in process' PCB)

2 special entries in each UNIX directory:

“.” current dir

“..” for parent

To access a file:

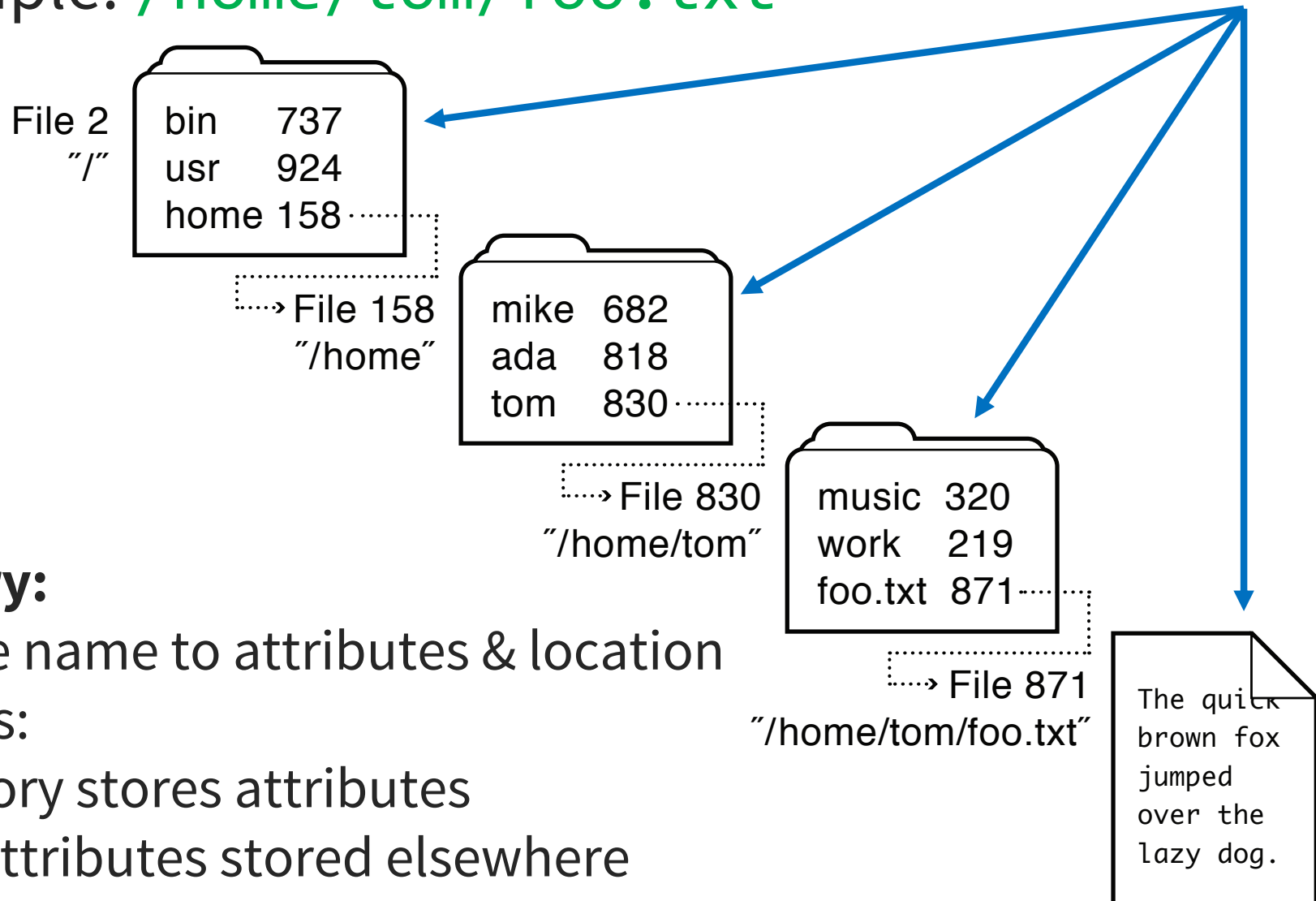
- Go to the folder where file resides —OR—
- Specify the path where the file is

Directories

OS uses path name to find directory

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

all files



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 name) to **values** (block number 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

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



Used in CD-ROMs, DVDs

Linked List Allocation

Each file is stored as linked list of blocks

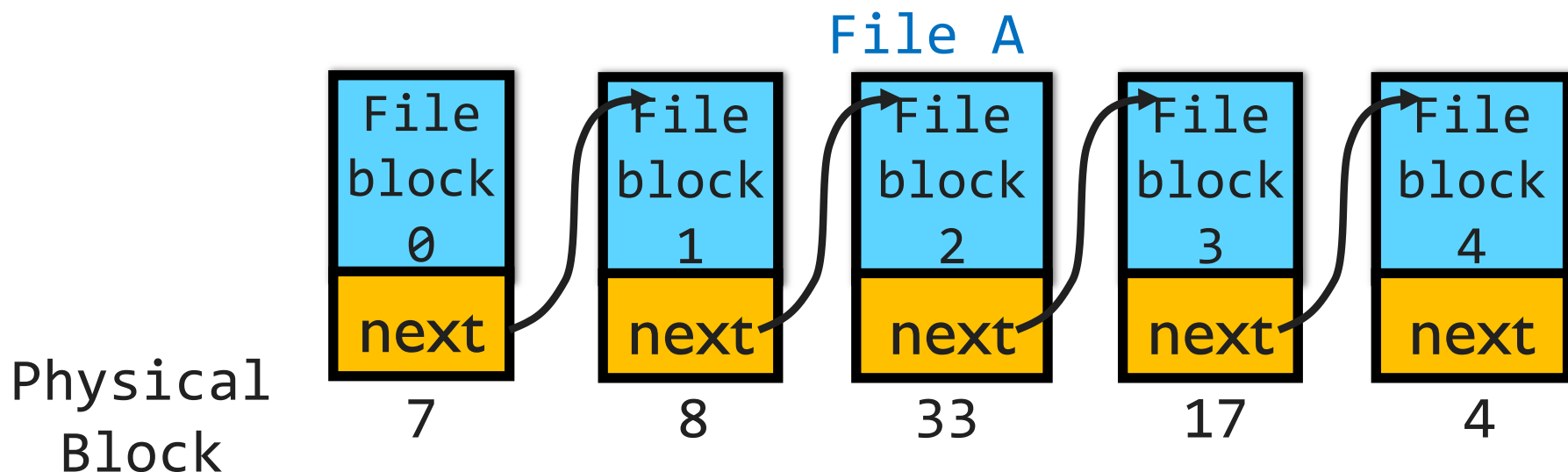
- First word of each block points to next block
- Rest of disk block is file data

+ **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

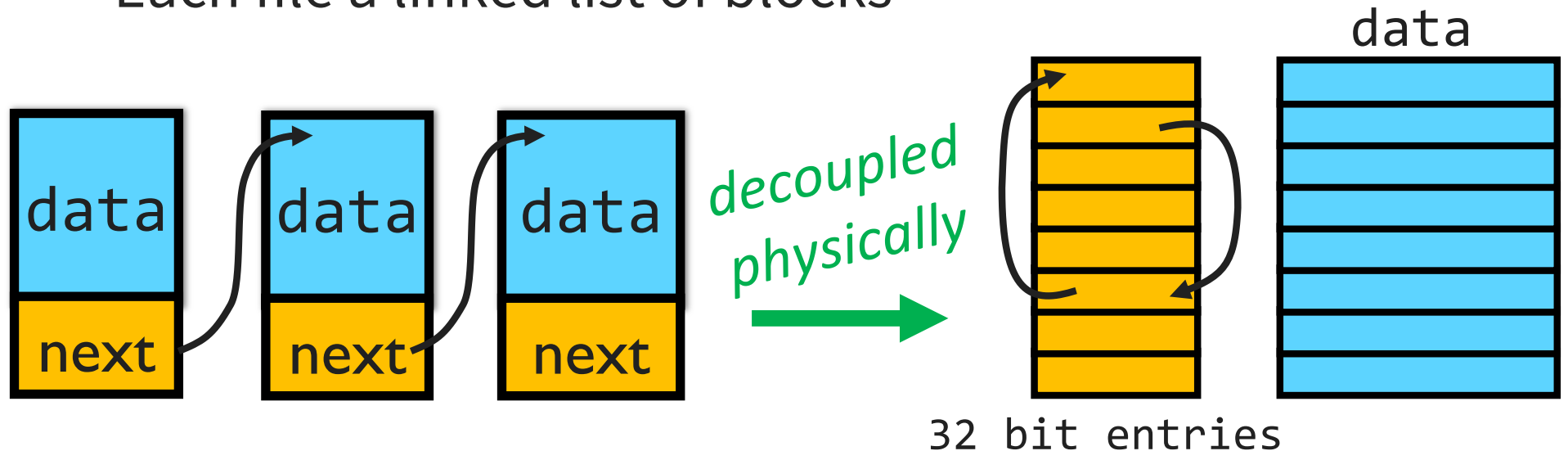
[late 70's]

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:

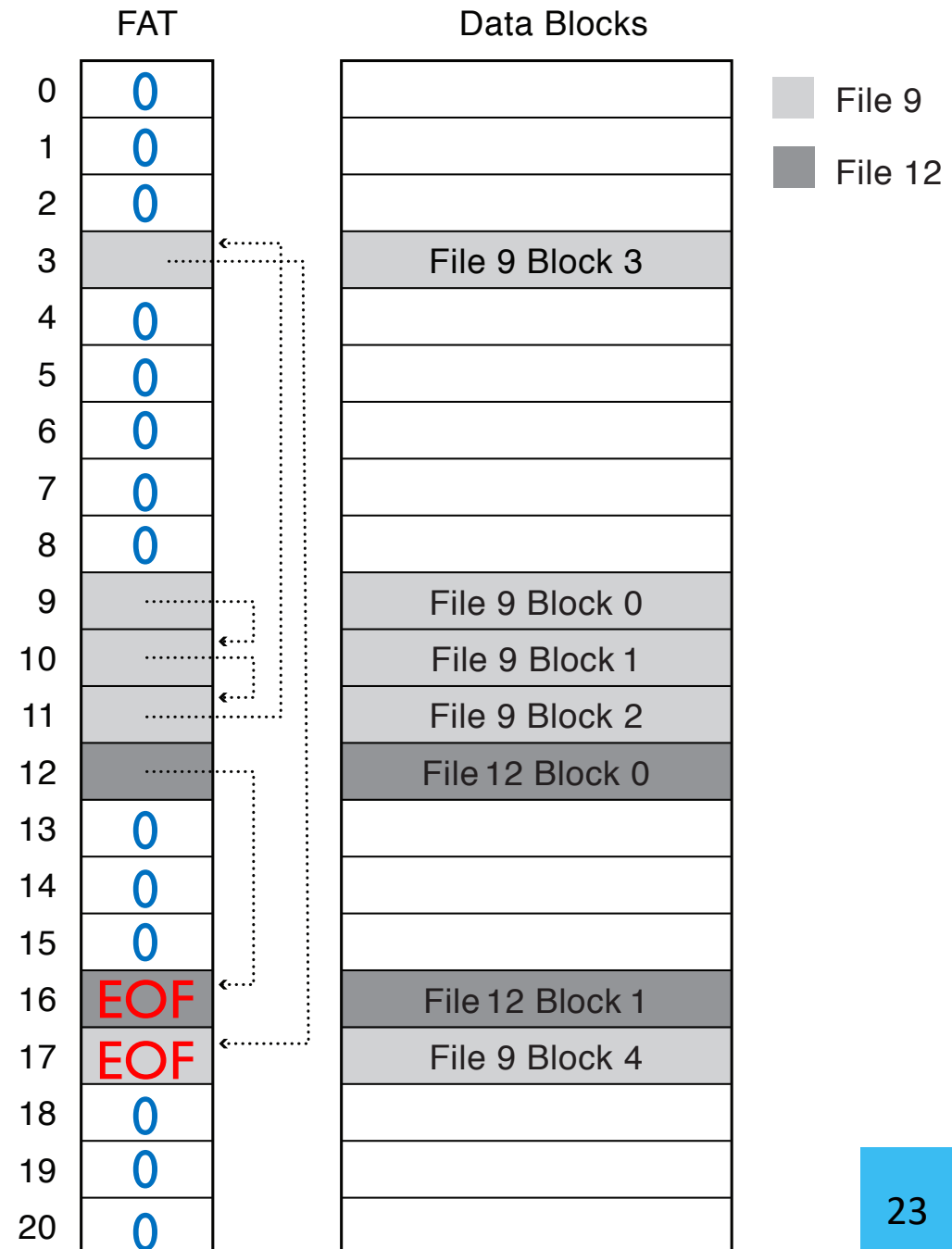
- Linear map of all blocks on disk
- Each file a linked list of blocks



FAT File System

- 1 entry per block
- **EOF** for last block
- **0** indicates free block
- usually uses a simple allocation strategy (e.g. next-fit)
- directory entry maps name to FAT index

Directory	
bart.txt	9
maggie.txt	12

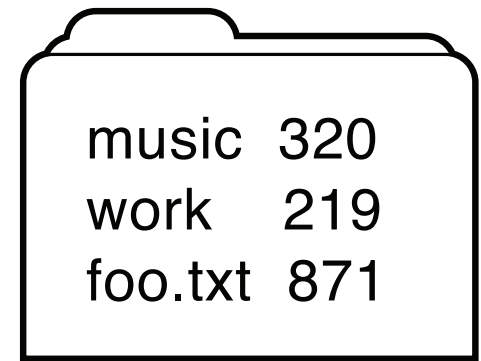


FAT Directory Structure

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



music	320
work	219
foo.txt	871

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 \rightarrow 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

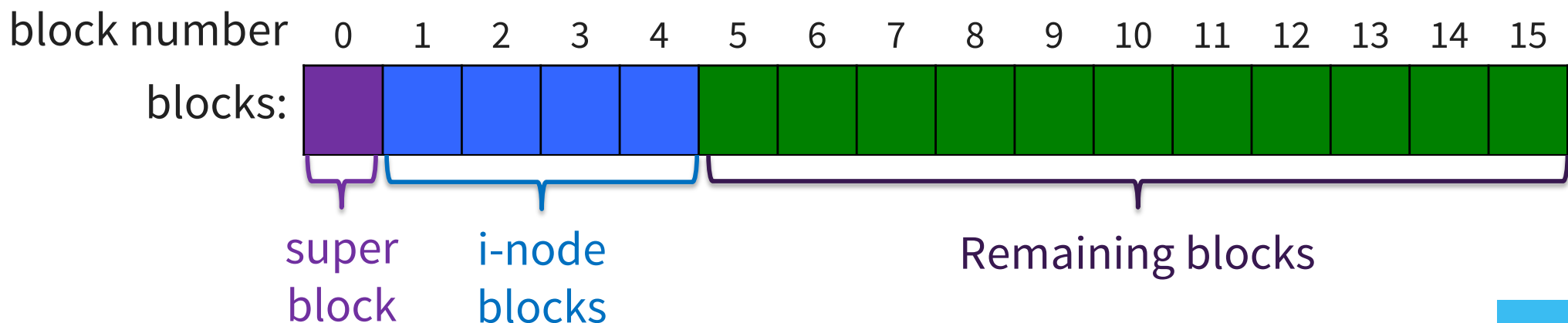
Tree-based, multi-level index

...but first... A4

FFS Superblock

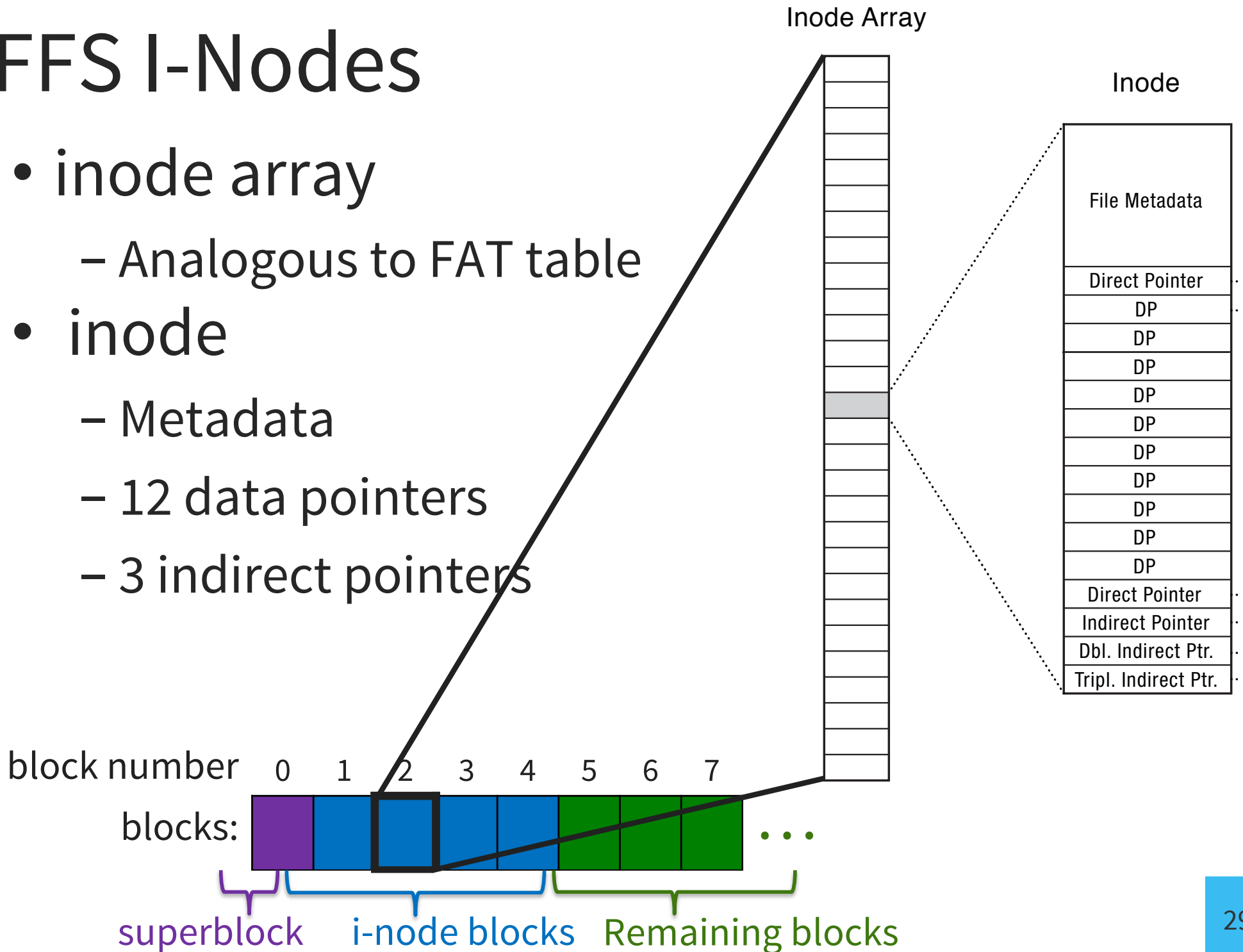
Identifies file system's key parameters:

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



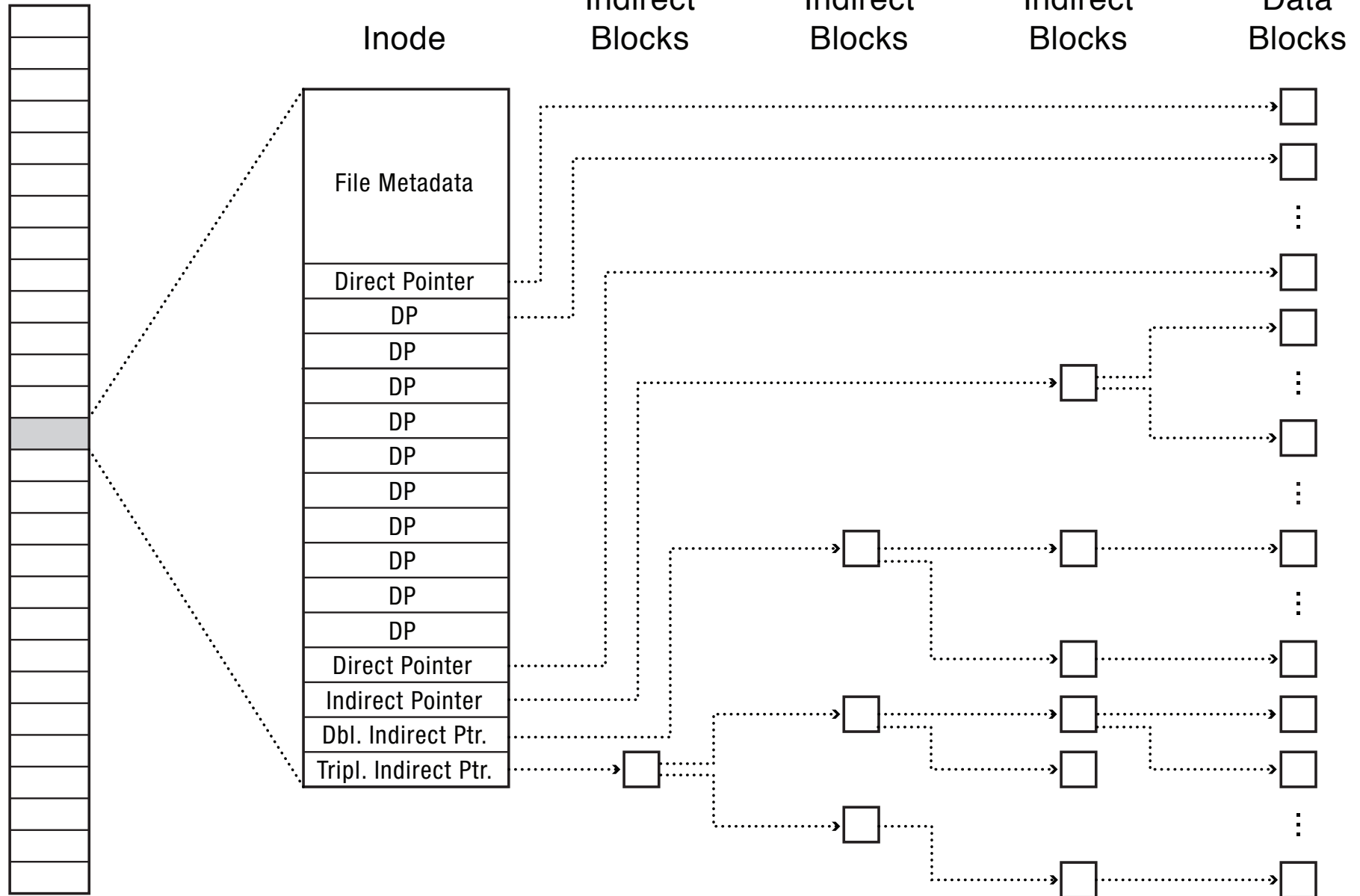
FFS I-Nodes

- inode array
 - Analogous to FAT table
- inode
 - Metadata
 - 12 data pointers
 - 3 indirect pointers



FFS: Index Structures

Inode Array



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	·
Direct Pointer	·
Indirect Pointer	·
Dbl. Indirect Ptr.	·
Tripl. Indirect Ptr.	·

FFS: Index Structures

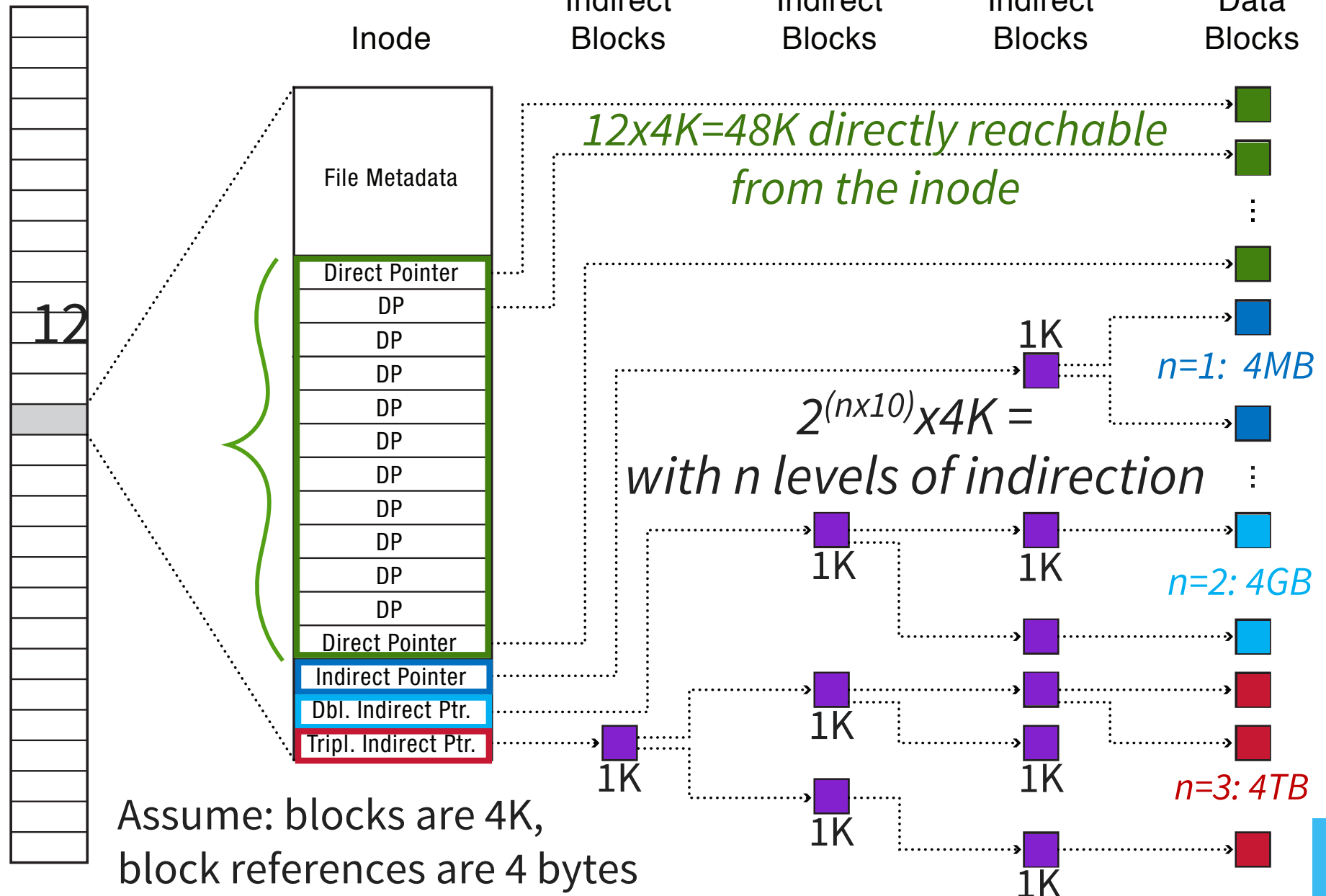
Inode Array

Triple
Indirect
Blocks

Double
Indirect
Blocks

Indirect
Blocks

Data
Blocks



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

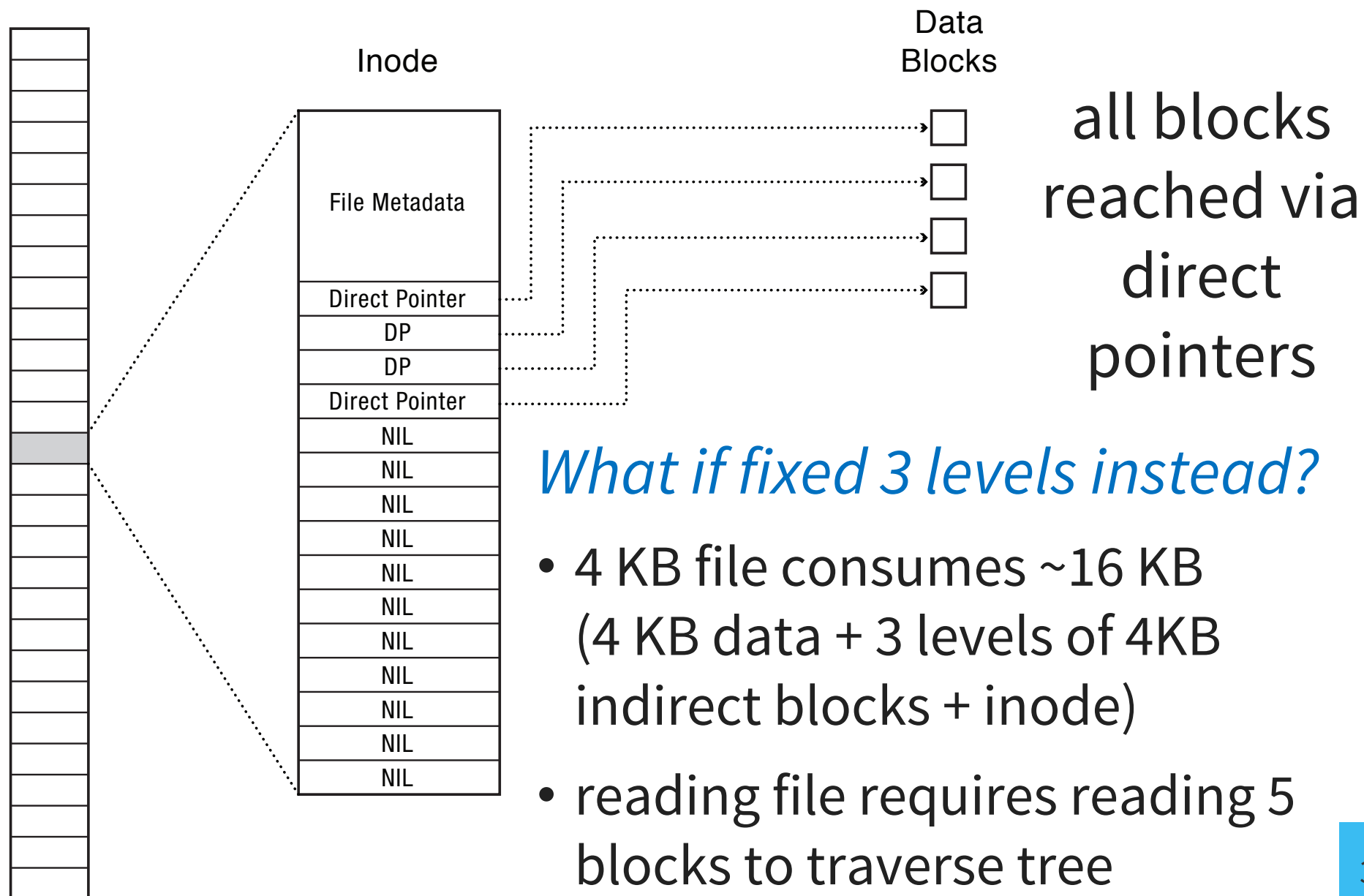
- implementation simplicity

4. Asymmetric

- not all data blocks are at the same level
- supports large
- small files don't pay large overheads

Small Files in FFS

Inode Array



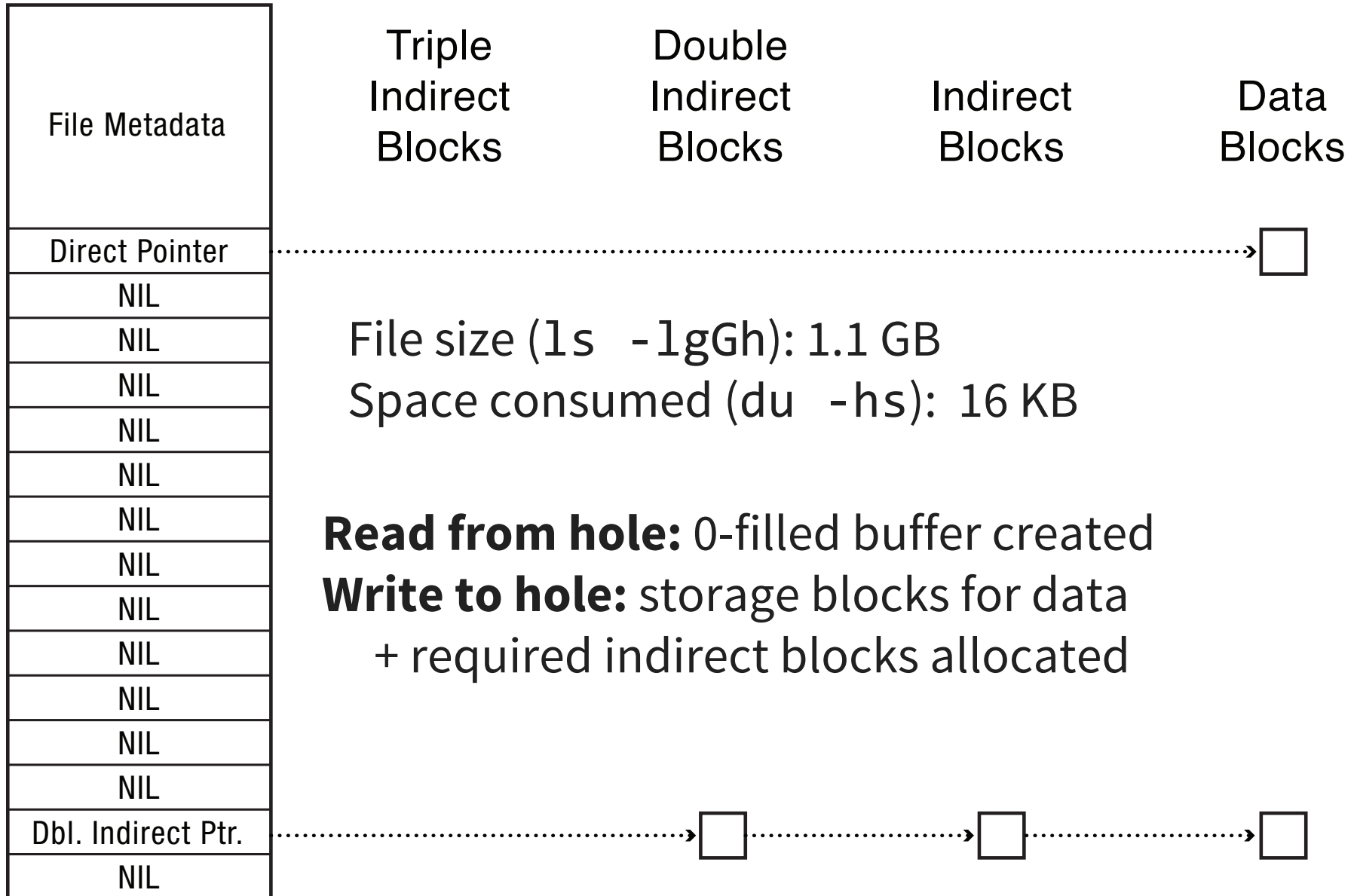
Sparse Files in FFS

Example:

2 x 4 KB bocks: 1 @ offset 0

1 @ offset 2^{30}

Inode

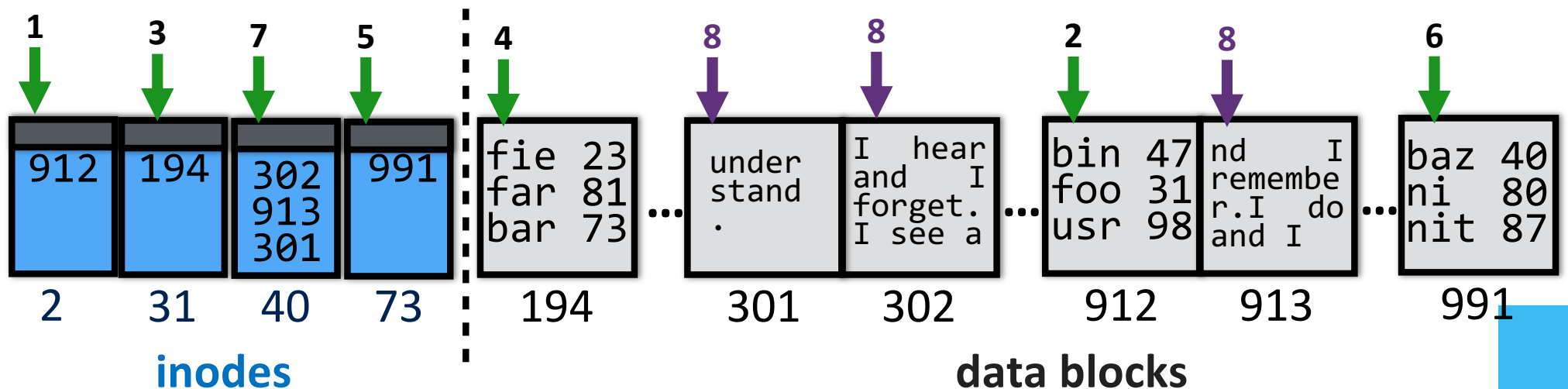


FFS: Steps to reading /foo/bar/baz

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*



File System Consistency

System crashes before modified files written back?

- Leads to inconsistency in FS
- fsck (UNIX) & scandisk (Windows) check FS consistency
- (also gets called in A4)

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

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)

Inconsistent FS Examples

Consistent

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

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	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	in use
0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1	free list

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	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	in use
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1	free list

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	
1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0	in use
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	free list