File Systems

CS 4410, Operating Systems

Fall 2016
Cornell University

Rachit Agarwal
Anne Bracy

See: Ch 13 in OSPP textbook

The slides are the product of many rounds of teaching CS 4410 by Professors Sirer, Bracy, Agarwal, George, and Van Renesse.

File Systems 101

Long-term Information Storage Needs

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

Solution

- Store information on disks in units called *files*
  - persistent, only owner can delete
  - managed by the OS

File Systems: How the OS manages files!

Challenges for File System Designers

- **Performance:** despite limitations of disks
  - leverage spatial locality

- **Flexibility:** need jacks-of-all-trades, 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 crashes or malfunctions

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

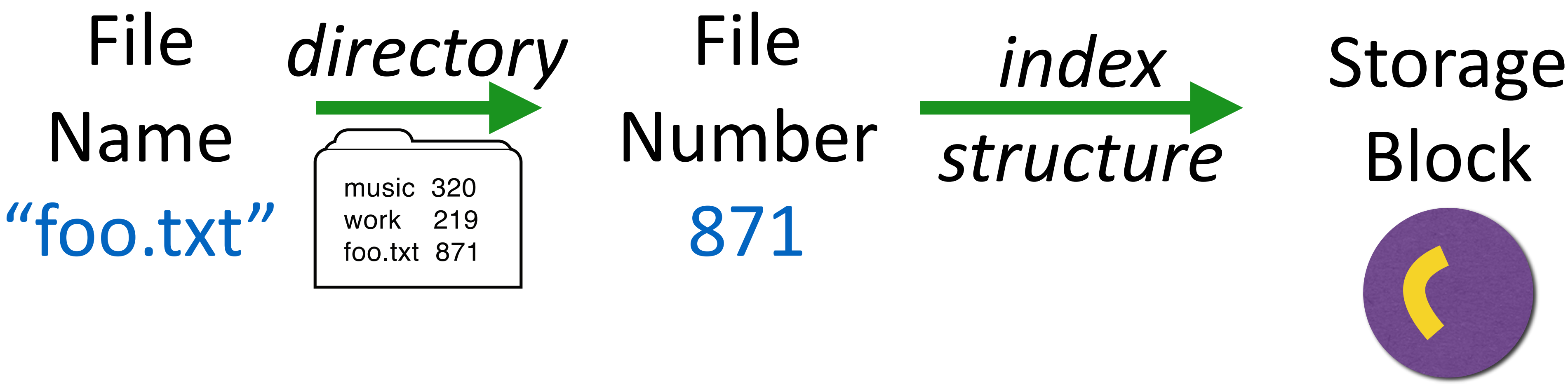
- OS dependent
  Windows not case sensitive, UNIX is
- Usually ok up to 255 characters

File Extensions

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

Directory

Maps human readable name to file number



Path Names

- Absolute: path of file from the root directory

  e.g., /home/pat/projects/test.c
- Relative: path from the current working directory (current work dir stored in process' PCB)

2 special entries in each UNIX directory:

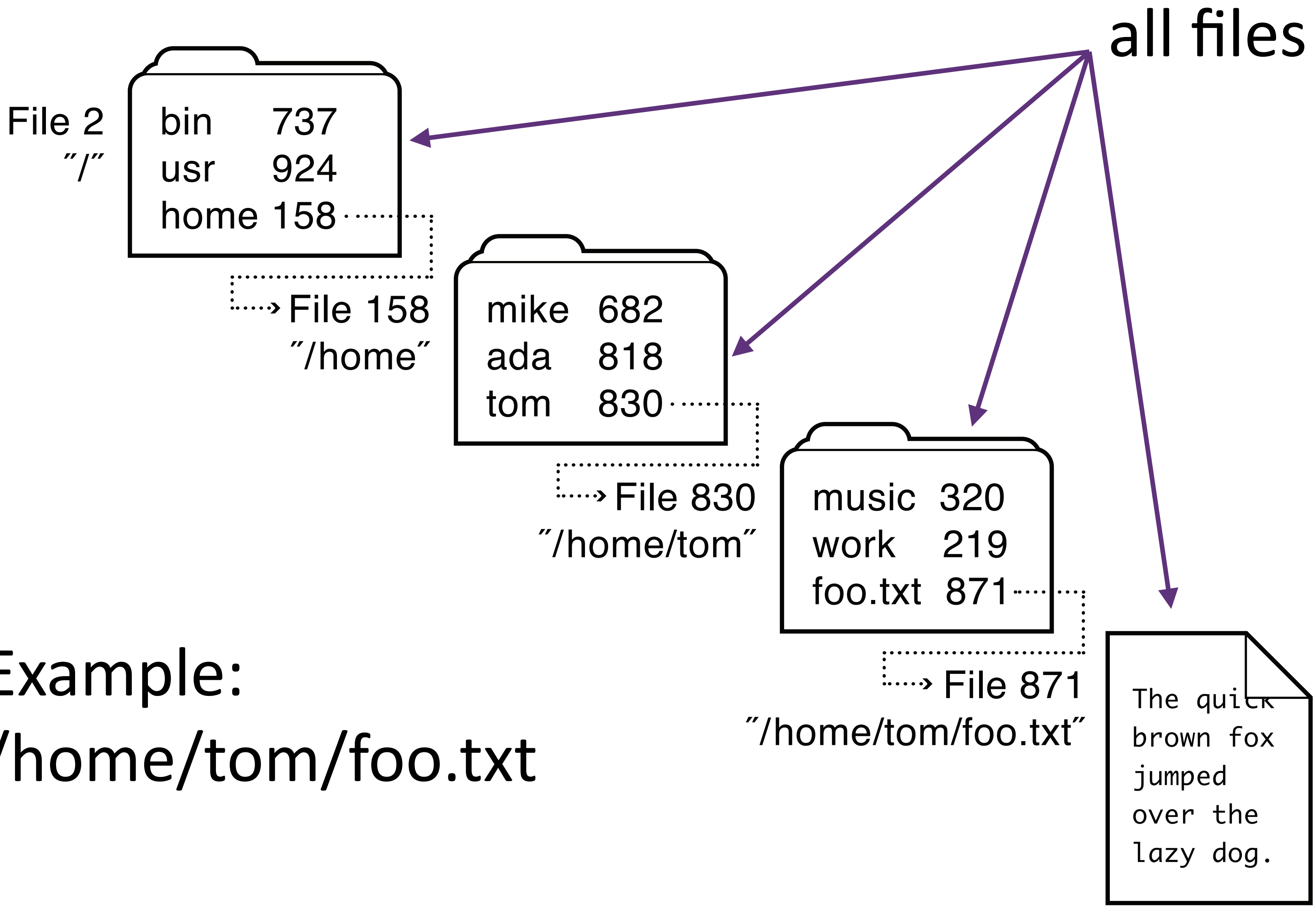
"." current dir

".." for parent

To access a file:

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

Paths in action!



Example:
/home/tom/foo.txt

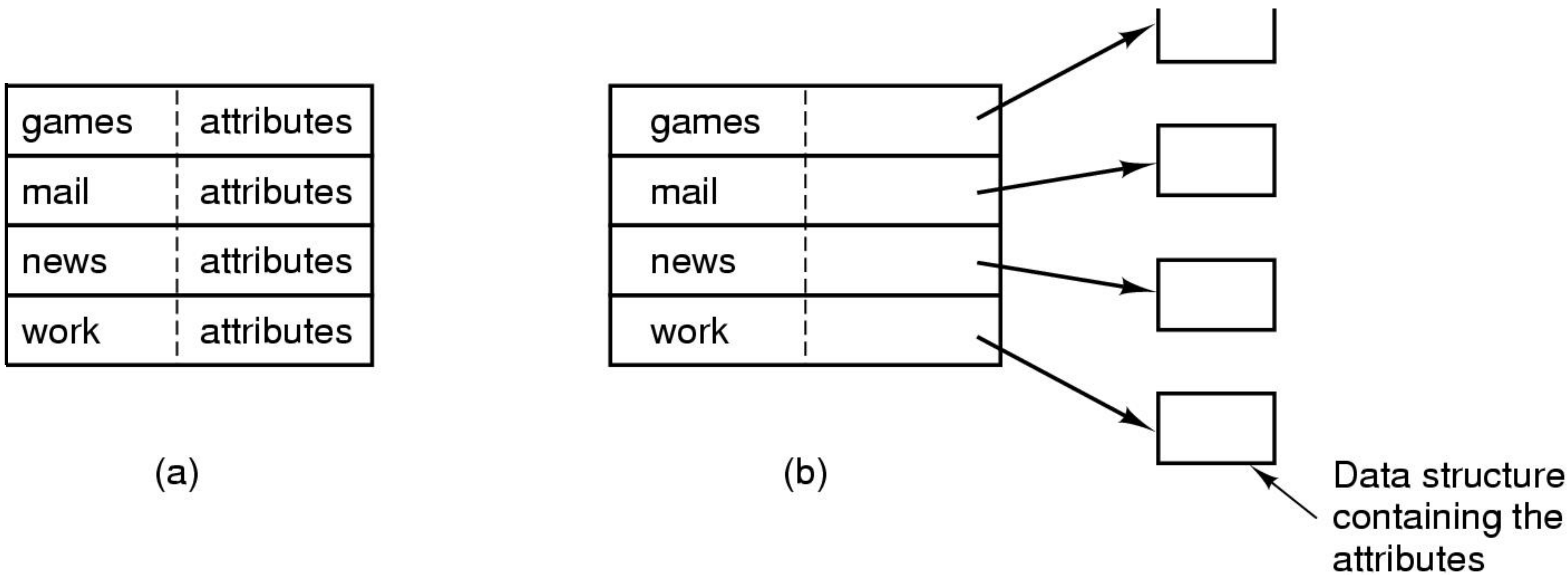
Implementing Directories

When a file is opened, OS uses path name to find dir

- Directory has information about the file's disk blocks
- Directory also has attributes of each file

Directory: map ASCII file name to file attributes & location

2 options: entries have all attributes, or point to file I-node



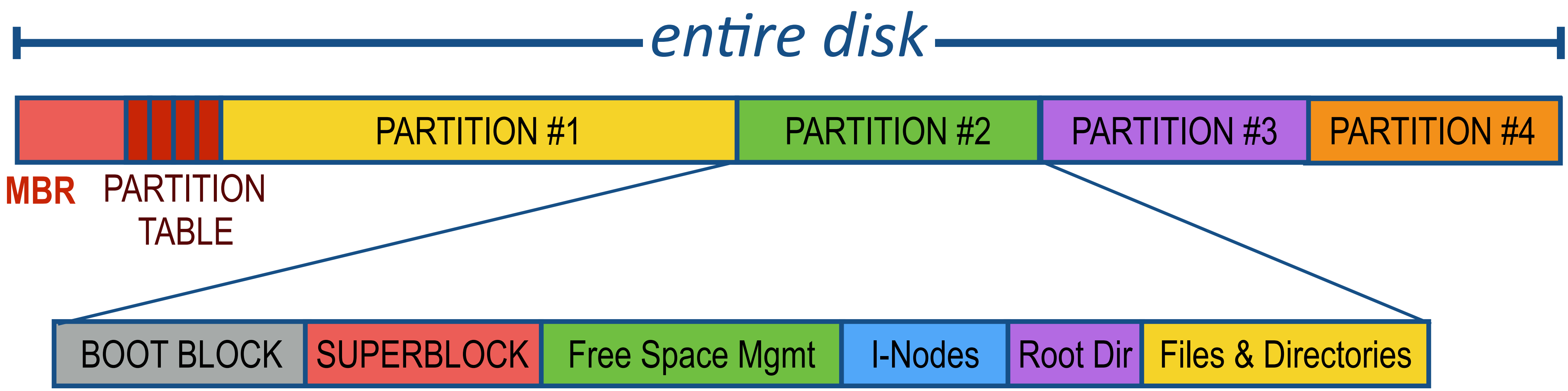
File System Layout

File System is stored on *disks*

- disk is 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

+ Performance:
    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

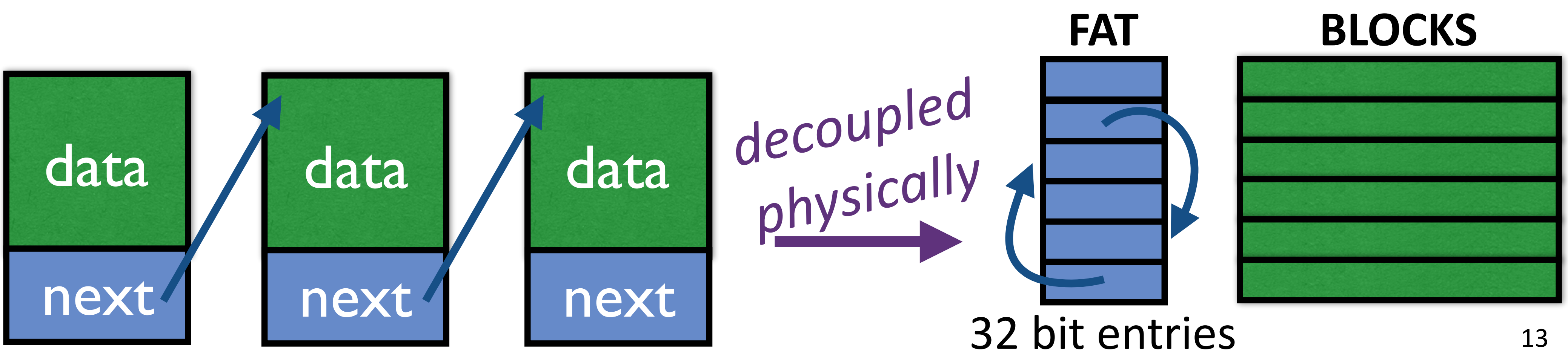
Case Study #1: File Allocation Table (FAT)

Microsoft File Allocation Table [late 70's]

- 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

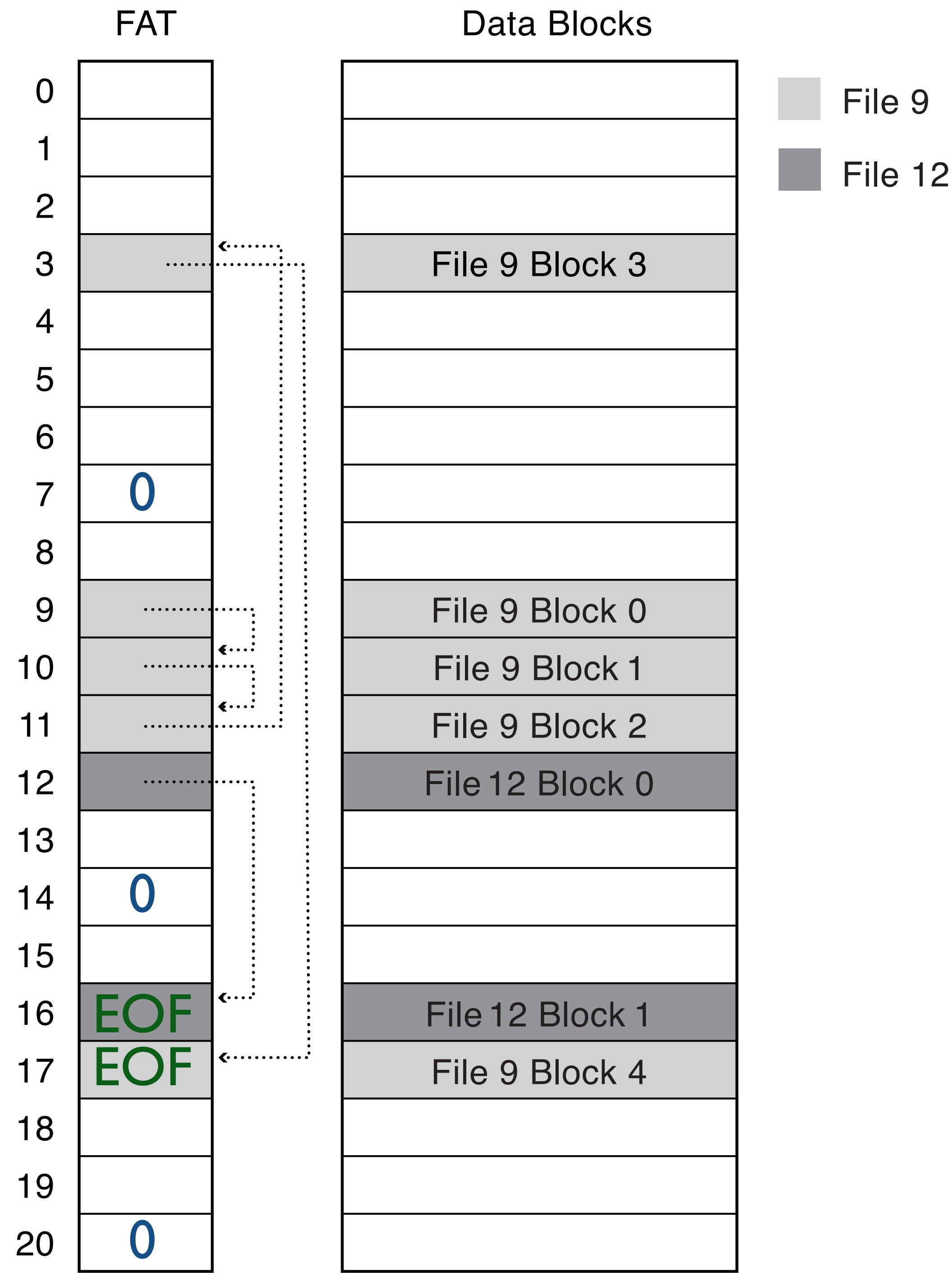


FAT File System

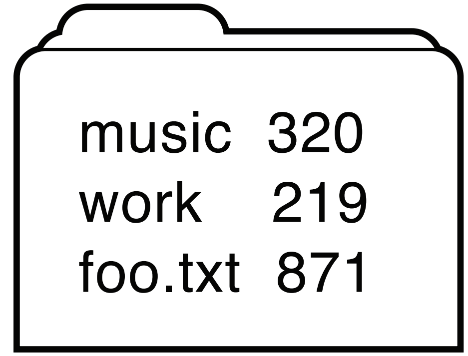
- 1 entry per block
- EOF for last block
- 0 indicates free block
- usually uses a simple allocation strategy (e.g. next-fit)
- directory entry maps name to FAT index

Directory

bart	9
maggie	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 Good is FAT?

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

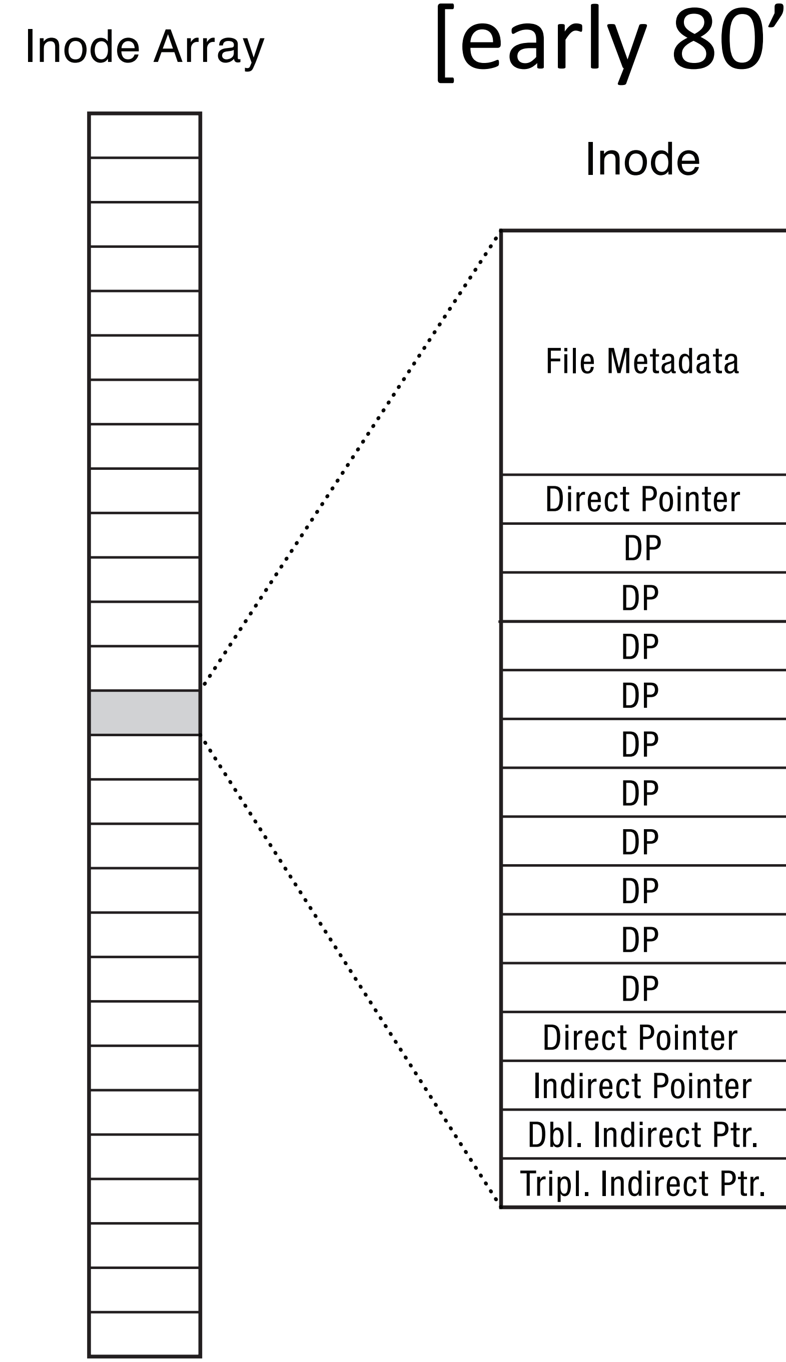
- Poor locality
- Many file seeks unless entire FAT in memory
- Poor random access
- Limited metadata
- Limited access control
- No support for hard links
- Limitations on volume and file size
- No support for reliability techniques

Case Study #2: Fast File System (FSS)

UNIX Fast File System

[early 80's]

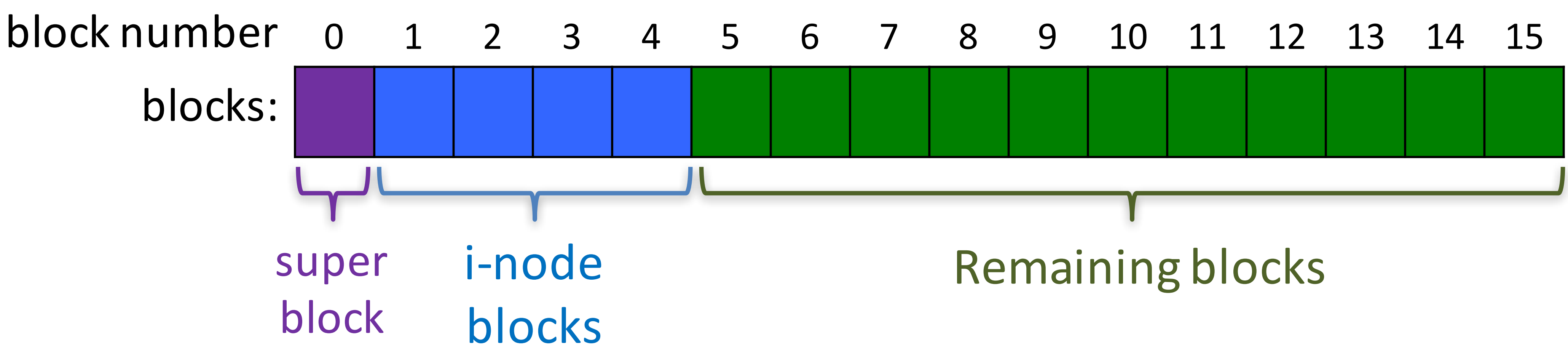
- inode table
    - Analogous to FAT table
- inode
    - Metadata
    - 12 data pointers
    - 3 indirect pointers



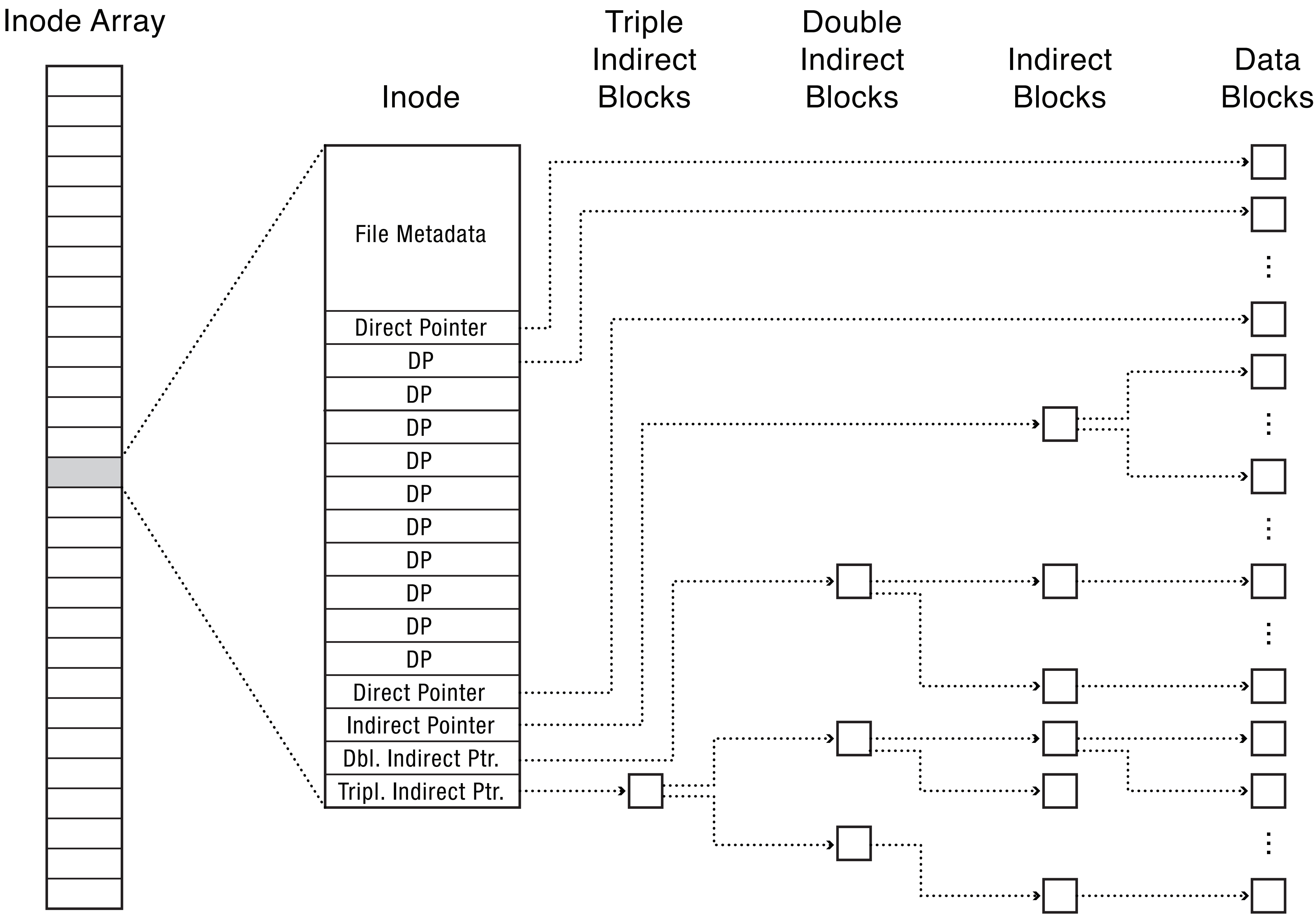
FFS Superblock

Identifies file system's key parameters:

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

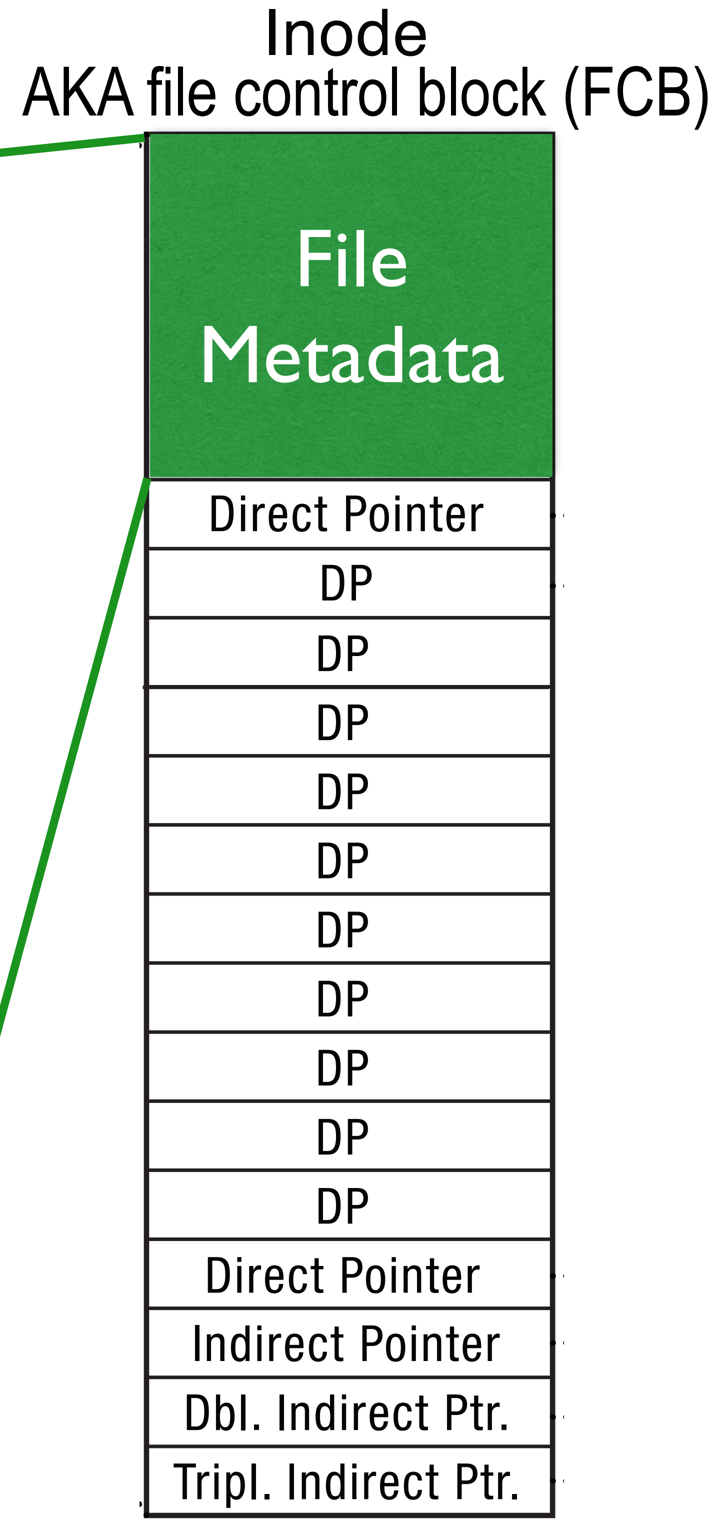


FFS: Index Structures

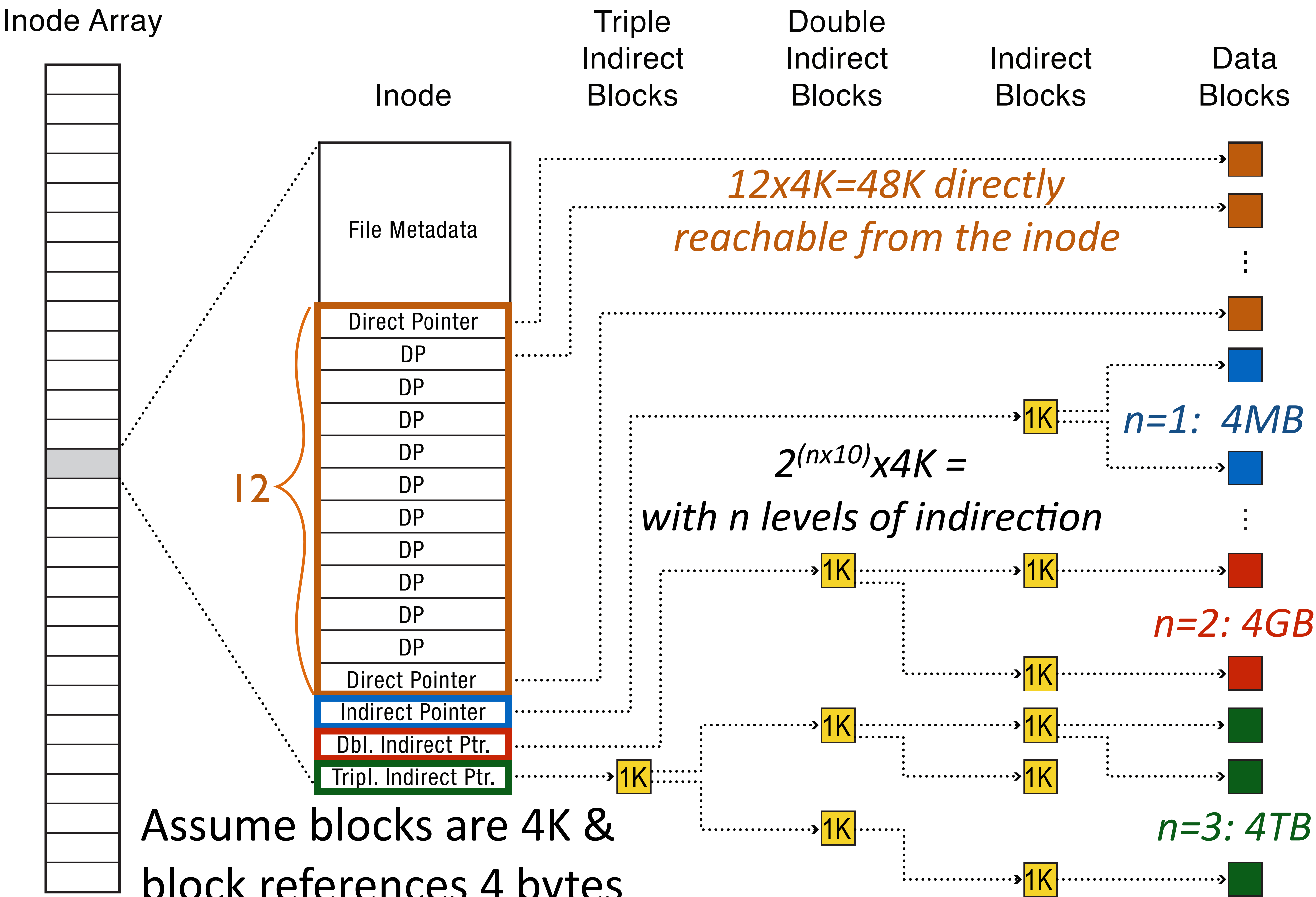


What else is in an Inode?

- 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

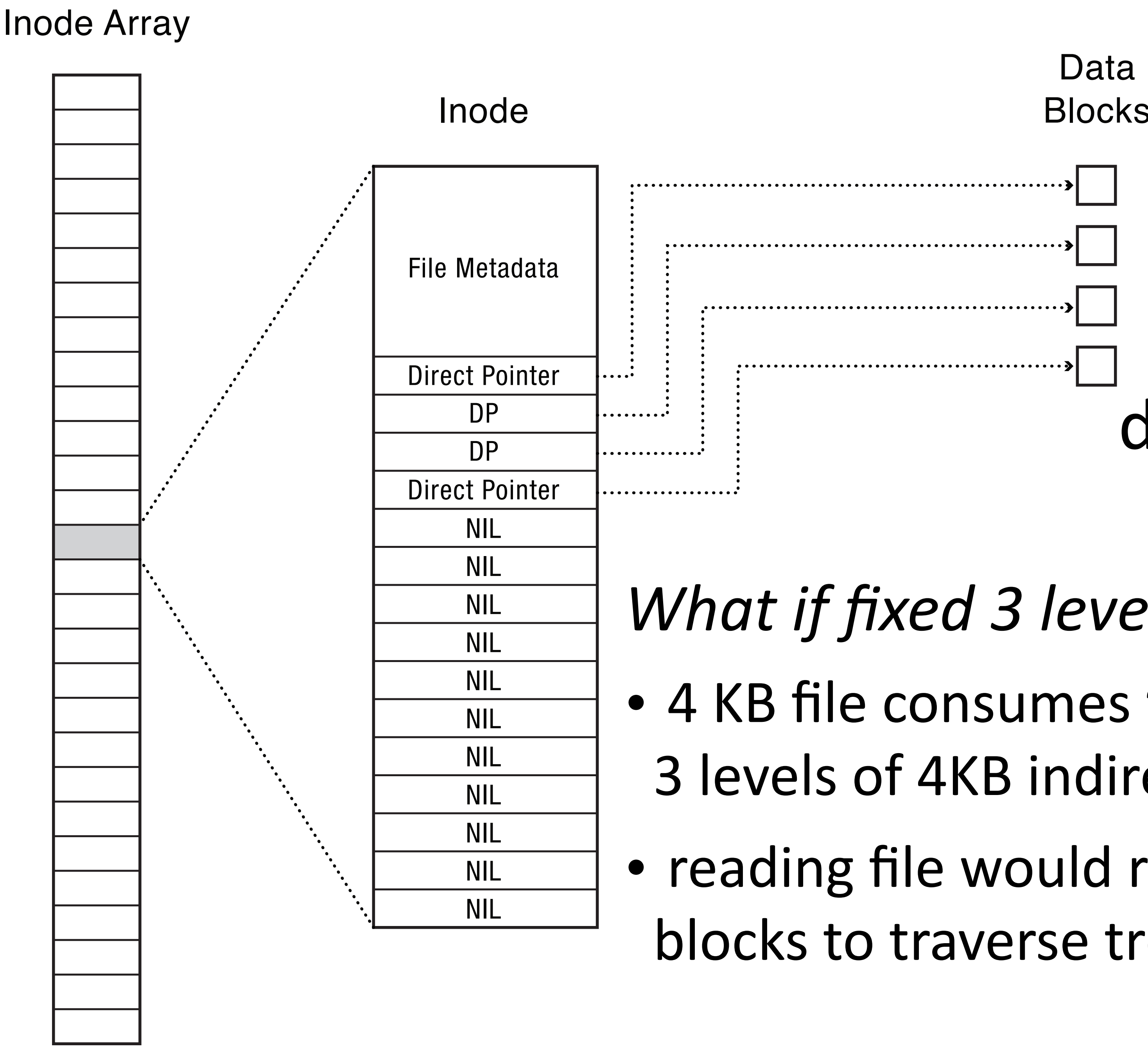


Assume blocks are 4K & block references 4 bytes

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
   - small files don't pay large overheads

Small Files in FFS



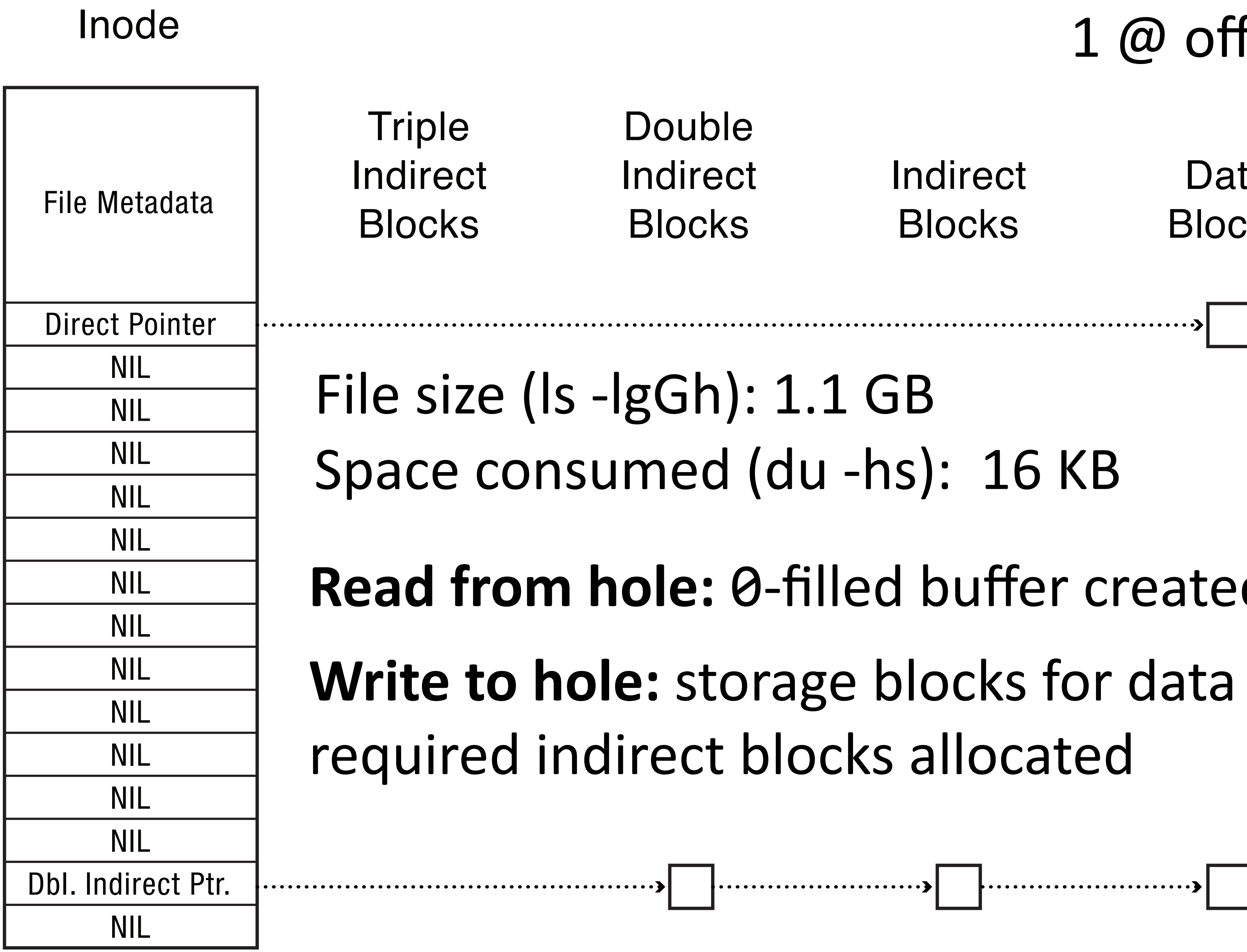
all blocks reached via direct pointers

What if fixed 3 levels instead?

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

Sparse Files in FFS

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



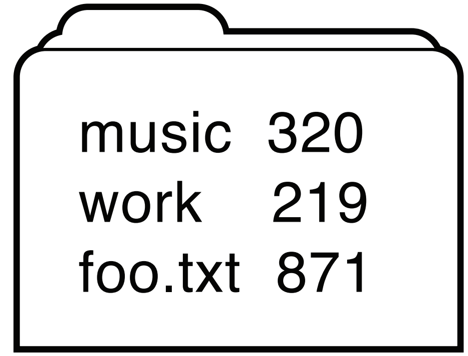
File size (ls -lgGh): 1.1 GB

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



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

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