File System Structure

Kevin Webb

Swarthmore College

March 31, 2020

Today’s Goals

- Characterizing disks and storage media
- File system: adding order and structure to storage
- FS abstractions (files, directories, paths) and their implementation with data structures (inodes, block pointers, directory entries)

Motivation for File Systems

- Long-term storage is needed for
  - user data: text, graphics, images, audio, video
  - user programs
  - system programs (including OS kernel) and data
- Persistent: remains “forever”
- Large: “unlimited” size
- Sharing: controlled access
- Security: protecting information

Using Disks for Storage

- Why disks: persistent, random access, cheap

SEAGATE ●●●●○ (9)

Seagate Exos 16TB Enterprise HDD X16 SATA 6Gb/s 512e/4Kn 7200 RPM 256MB...

$399.99 (20 Offers)

Sale Ends in 2 Days (Tue)

Free Shipping

VIEW DETAILS ▸

☐ Compare

SEAGATE ●●●●○ (219)

Seagate IronWolf 16TB NAS Hard Drive 7200 RPM 256MB Cache SATA 6.0Gb/s 3.5"...

$507.99 (10 Offers)

Free Shipping

VIEW DETAILS ▸

☐ Compare

SEAGATE ●●●●○ (9)

Seagate Exos X16 ST14000NM001G 14TB 7200 RPM 256MB Cache SATA...

$334.99 (13 Offers)

Free Shipping

VIEW DETAILS ▸

☐ Compare

SEAGATE ●●●●○ (32)

Seagate IronWolf Pro 16TB NAS Hard Drive 7200 RPM 256MB Cache SATA 6.0Gb/s...

$521.99 (6 Offers)

Free Shipping

VIEW DETAILS ▸

☐ Compare

WD ●●●●○ (1,597)

WD Red WD140EFFX 14TB 5400 RPM 512MB Cache SATA 6.0Gb/s 3.5" Internal Hard...

$504.99 (11 Offers)

WD ●●●●○ (54)

WD Gold 14TB Enterprise Class Hard Disk Drive - 7200 RPM Class SATA 6Gb/s 512M...

$489.99 (16 Offers)

WD ●●●●● (24)

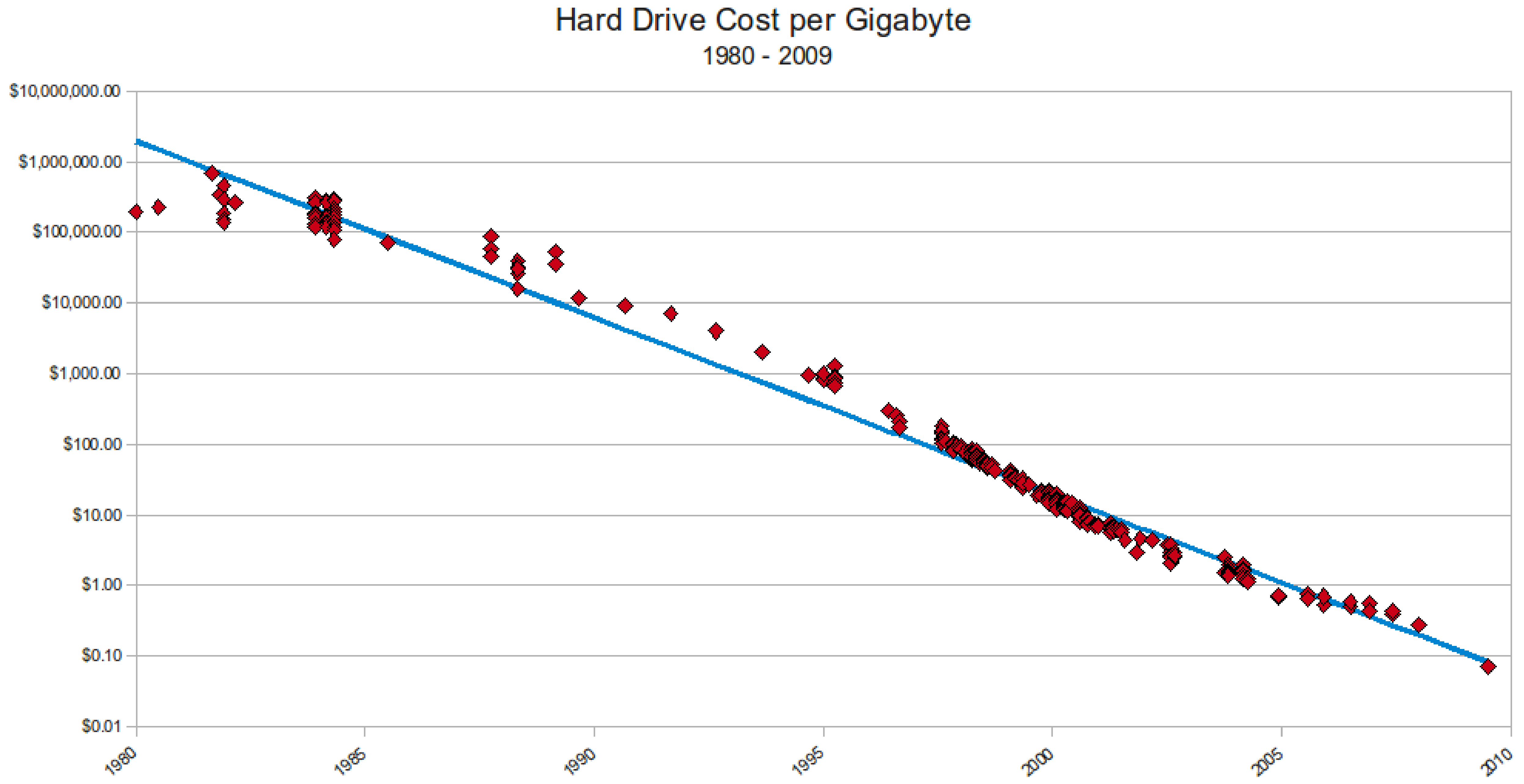
WD Red Pro WD141KFGX 14TB 7200 RPM 512MB Cache SATA 6.0Gb/s 3.5" Internal Hard...

$569.99 (14 Offers)

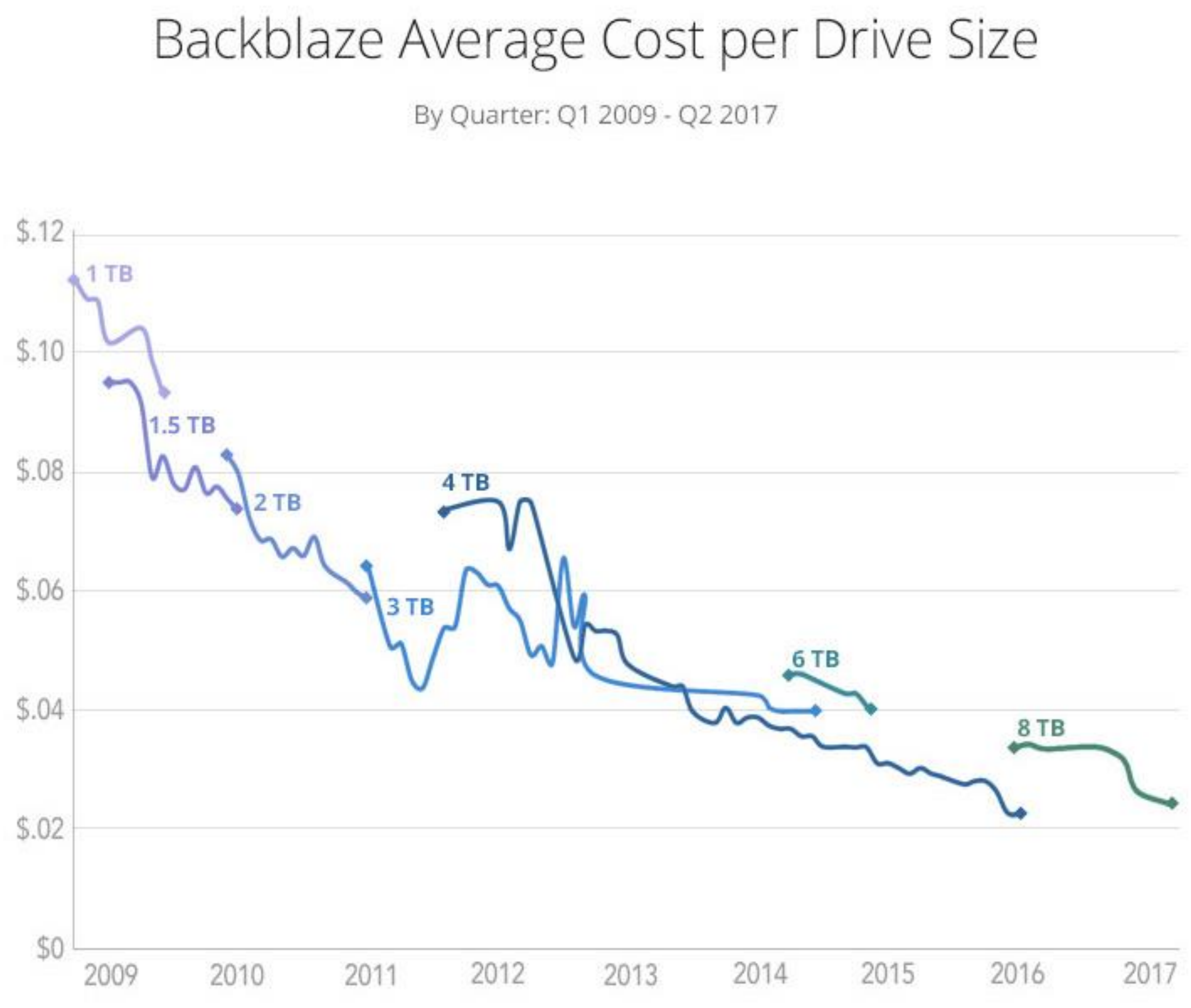
SEAGATE ●●●●○ (8)

Seagate Exos X14 14TB 7200 RPM 512e/4Kn SATA 6Gb/s 256MB Cache 3.5-Inch...

$339.99 (4 Offers)



Source: http://www.mkomo.com/cost-per-gigabyte



Source: https://www.backblaze.com/blog/hard-drive-cost-per-gigabyte/

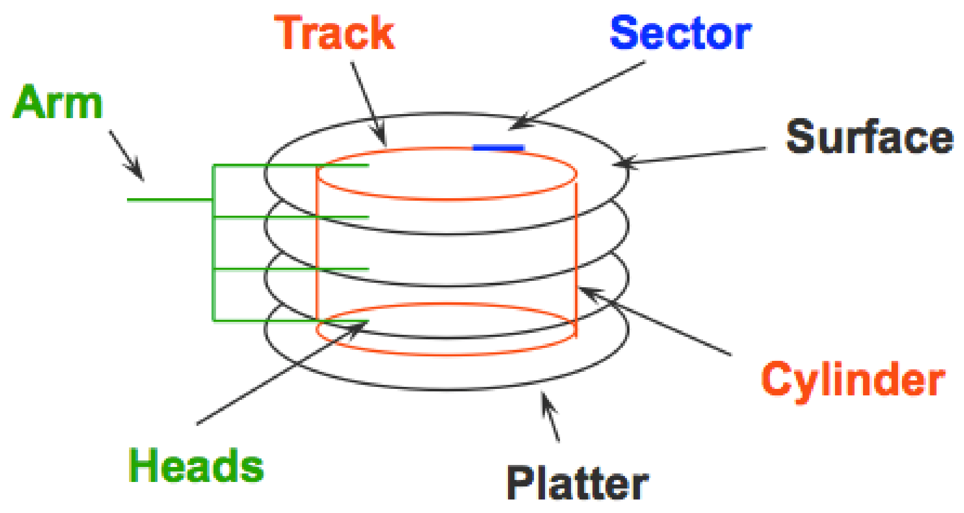
BACKBLAZE

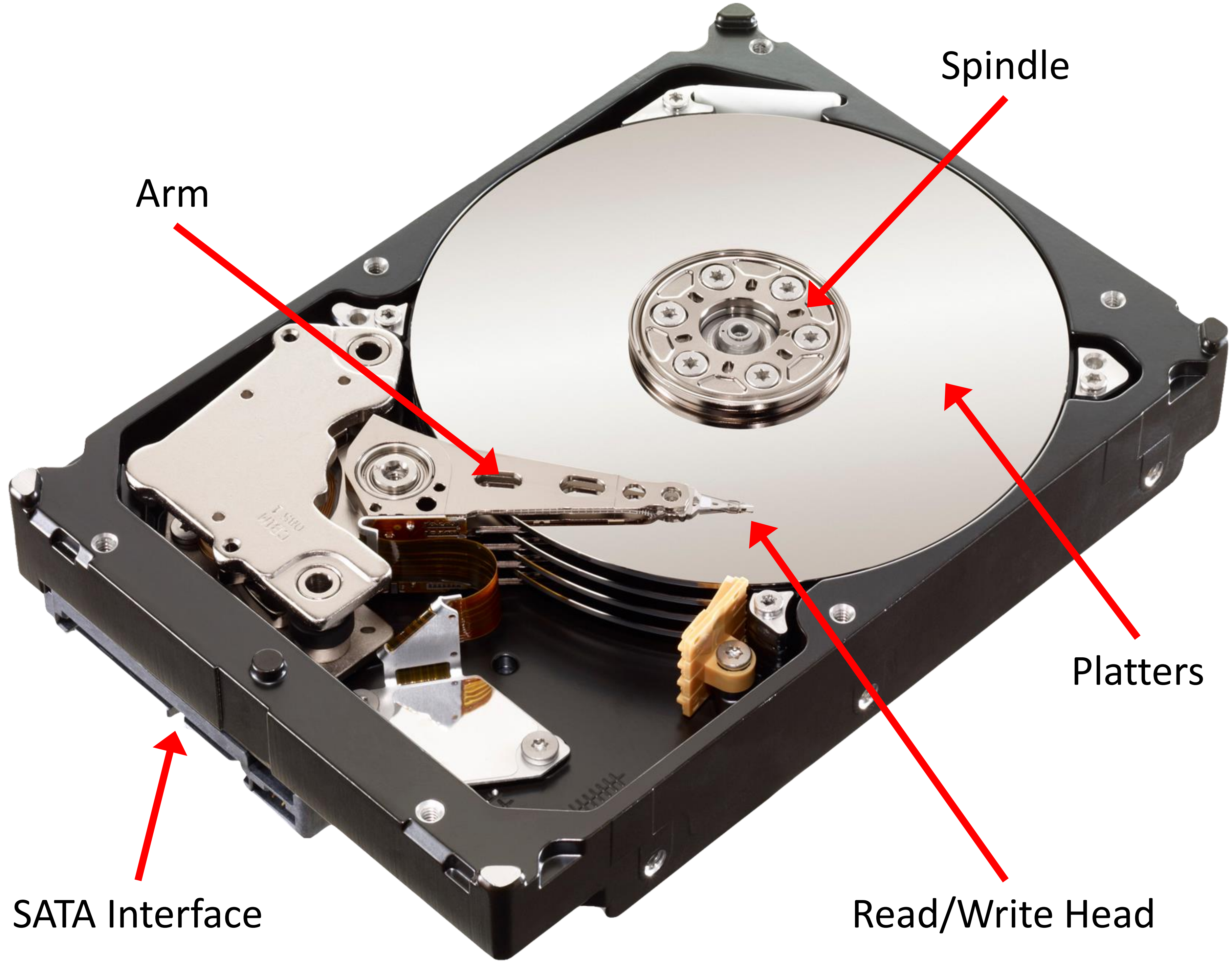
Using Disks for Storage

- Why disks: persistent, random access, cheap
- Biggest hurdle to OS: disks are slow

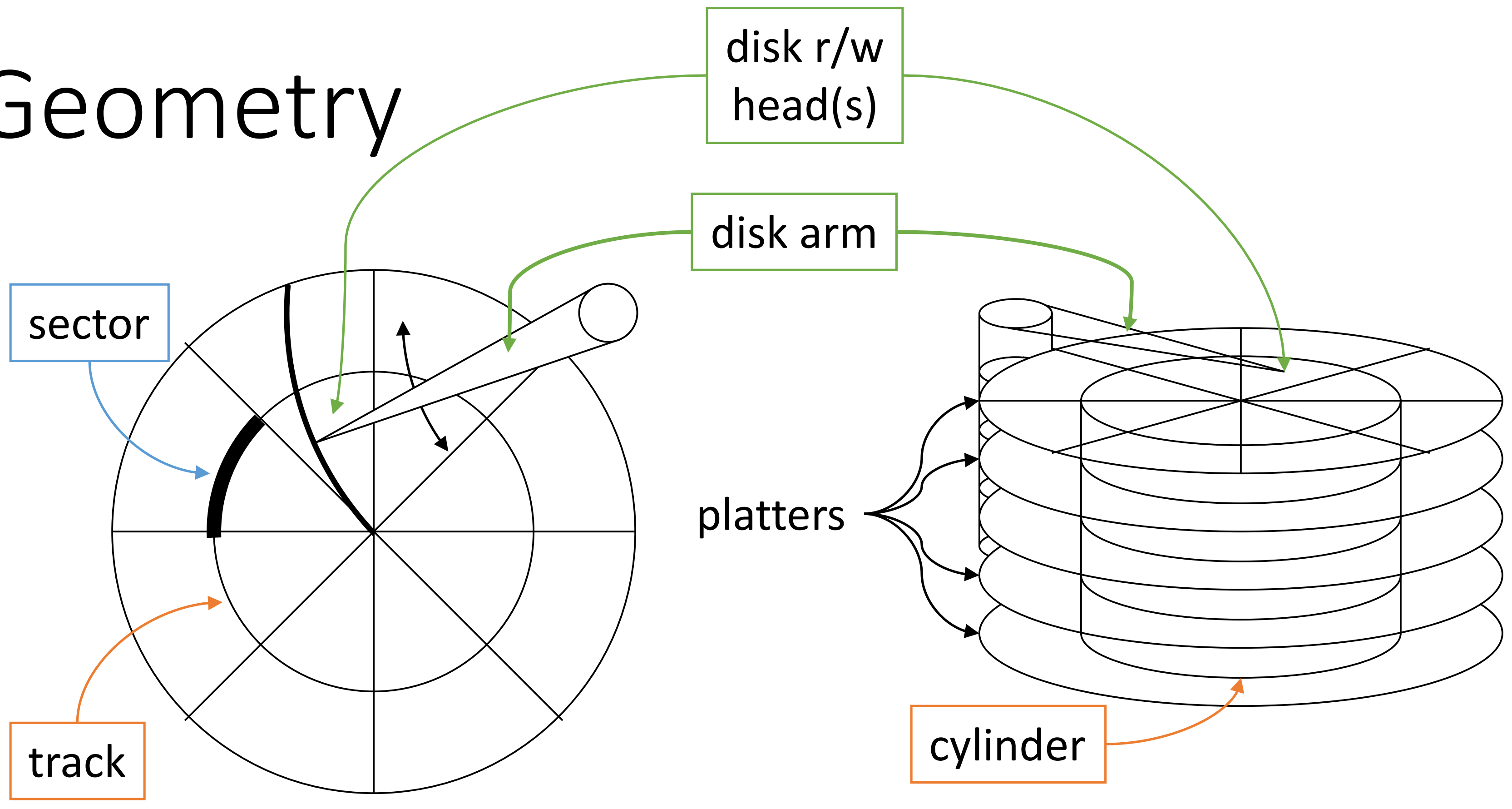
Disk Geometry

- Disk components
  - Platters
  - Surfaces
  - Tracks
  - Sectors
  - Cylinders
  - Arm
  - Heads




Spindle
Arm
Platters
SATA Interface
Read/Write Head

Disk Geometry



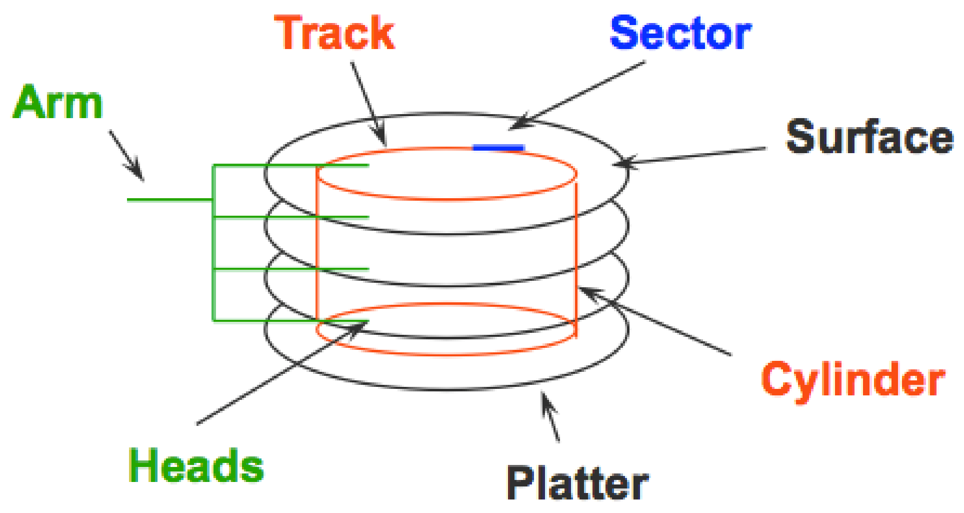
- Moving parts: spinning platters, disk actuator arm
- **seek time:** moving the arm to the desired track
- **rotational latency:** turning the platter so that the desired data is under the head

Implications of moving parts…

- https://www.youtube.com/watch?v=tDacjrSCeq4
- Disks have a MUCH higher failure rate than most other components
- PSA: back up your data!

Should the OS take these disk parameters into account when using a disk? Why or why not?

- Disk components
  - Platters
  - Surfaces
  - Tracks
  - Sectors
  - Cylinders
  - Arm
  - Heads



A. Yes B. No C. It depends (on?)

Disk Interaction

- In the old days: specifying disk requests required a lot of info:
  - Cylinder #, head #, sector # (CHS)
  - Disks didn’t have controller hardware built-in
- Very early OSes needed to know this info to make requests, but didn’t attempt to optimize data storage for it.
- ~mid 80’s: “fast file system” emerged, which took disk geometry into account.
  - Paper: “A Fast File System for Unix”
  - Example disk in paper is 150 MB

Disk Interaction

- Problem: storage increased rapidly, only 10 bits available for cylinder
  - Maximum size disk that could be represented: approx. 500 MB
- “Solution”: put a hardware controller on the disk
  - Controller lies to the OS about how many sectors and heads are available
  - Controller translates request from OS to actual >10-bit cylinder number
- More problems: disk complexity increases, new types of disk (SSD)

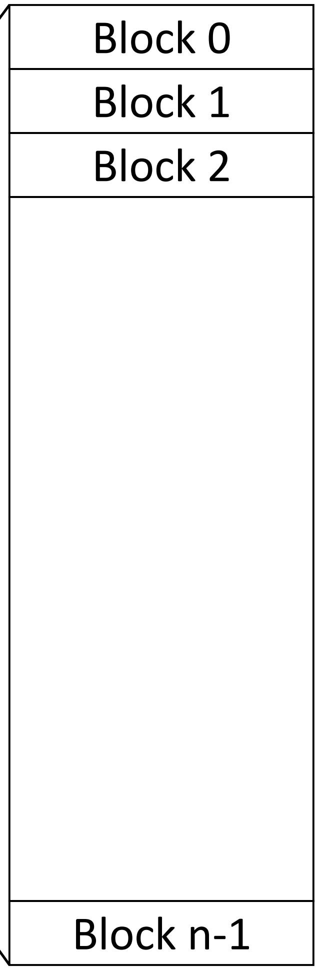
For history, see: http://jdebp.eu/FGA/os2-disc-and-volume-size-limits.html

Modern Disk Interaction

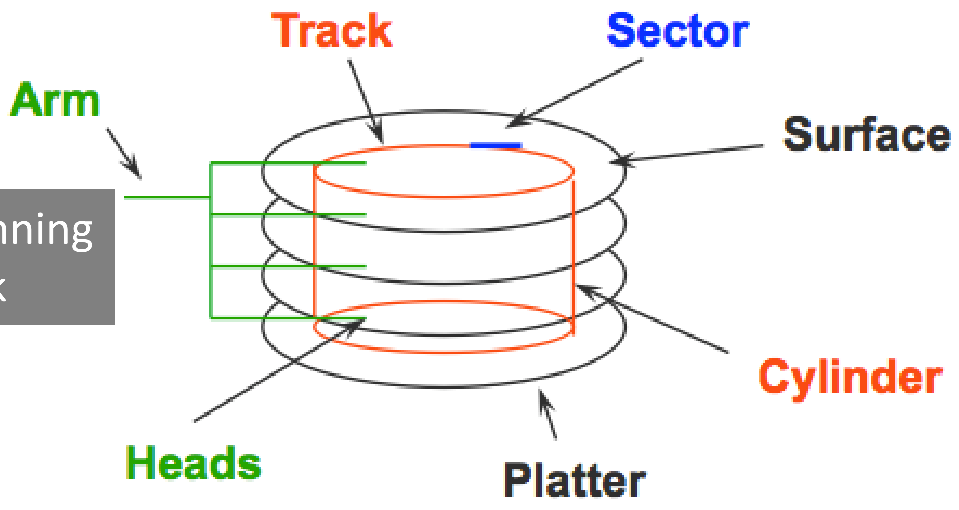
- Very simple block number interface:
  - disk is divided into N abstract blocks (traditionally 512 B, today often 4 KB)
  - read(block #)
  - write(block #, data)
- Trust the disk controller.
  - Convert block number to the “right” place in disk geometry.
  - For some disks (SSDs), this may not even be the same location every time!
- Significant research happening now in new types of storage.

Storage Abstraction for File System

Abstraction: Illusion of an array of blocks.

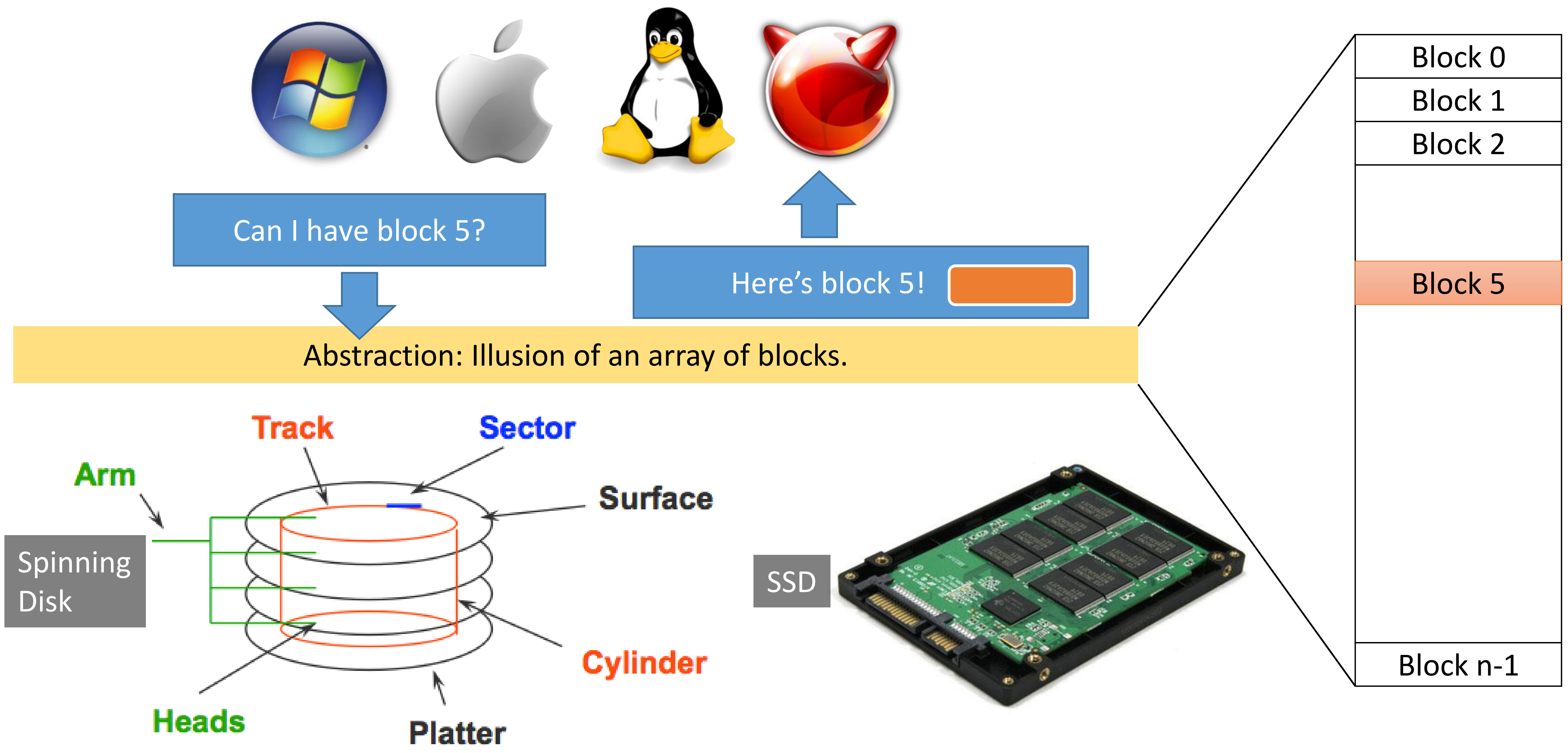


Spinning Disk



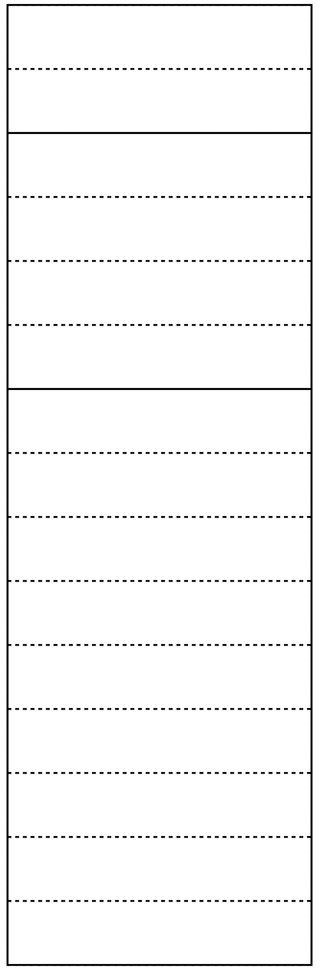
SSD

Storage Abstraction for File System

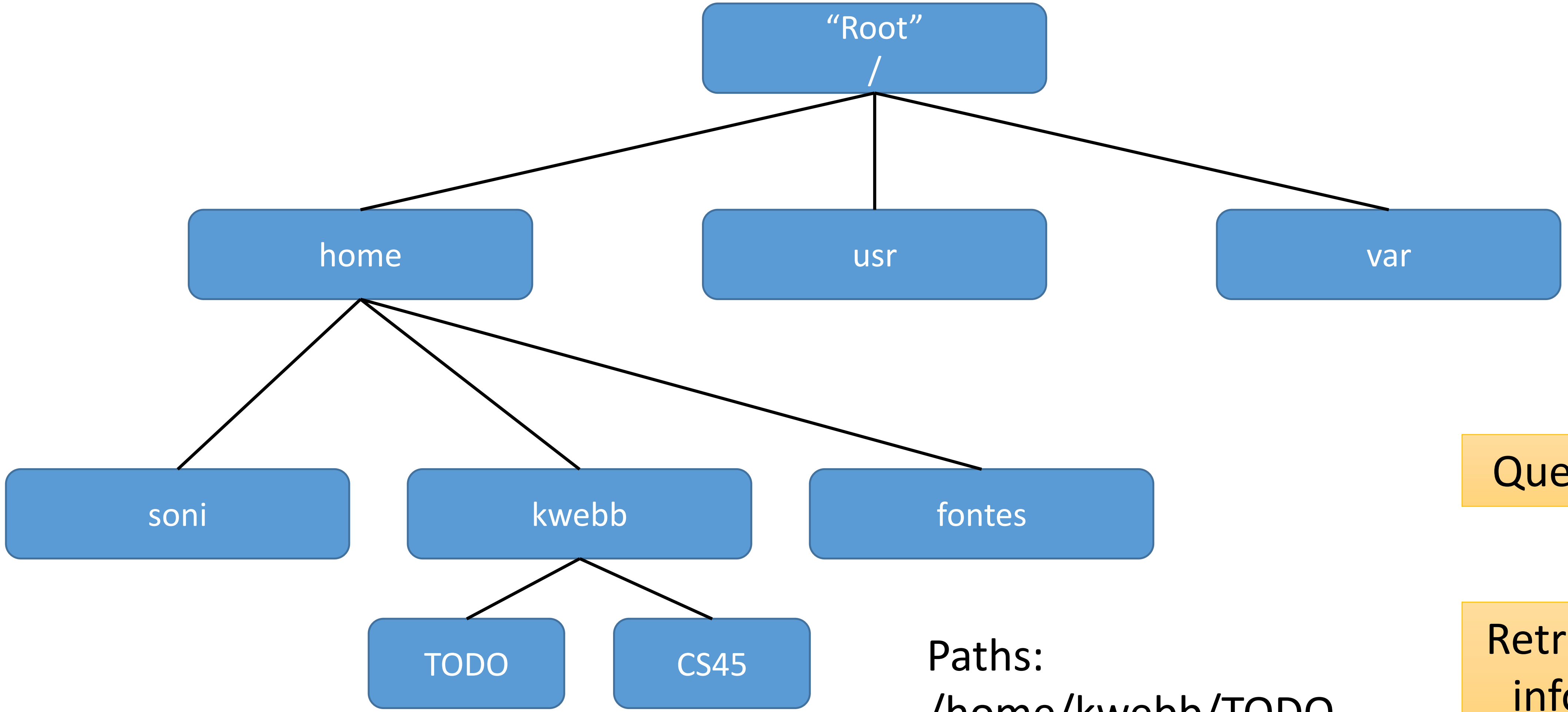


File Systems

- Disk “reality” to the OS: here’s a bunch of blocks, go crazy!
- File System: Add order and structure to the blocks.
  - Abstractions: files, directories, and others



FS Abstraction: Hierarchical Name Space - "Tree"



Query path

⬇

Retrieve file info/data

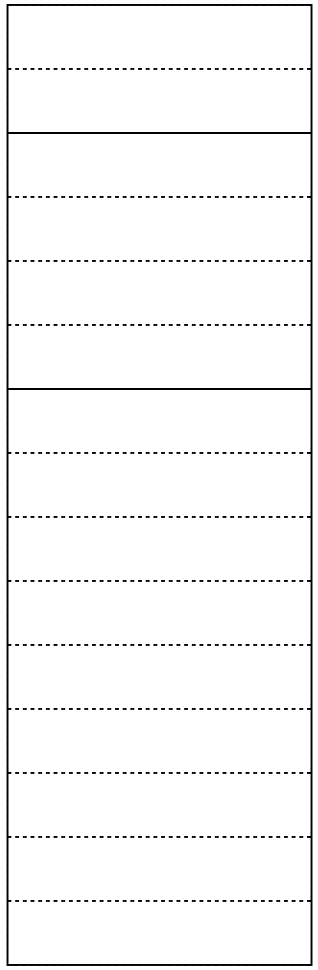
Paths:
/home/kwebb/TODO
/home/soni

File Systems

- Disk “reality” to the OS: here’s a bunch of blocks, go crazy!

- File System: Add order and structure to the blocks.
  - Abstractions: files, directories, and others
  - Control how data is stored and retrieved.
  - Translate high-level abstractions into low-level block requests.

- There are LOTS of ways to build file systems.
  - We’re going to mainly focus on “traditional” structure.



Data vs. Metadata

- **Data**: the files, directories, and other stuff you’re storing for the user.

- **Metadata**: information you’re storing about your data.
    - Example for entire FS: What type of FS is it? How large is it?
    - Example for one file: Where are the file’s blocks located on disk?

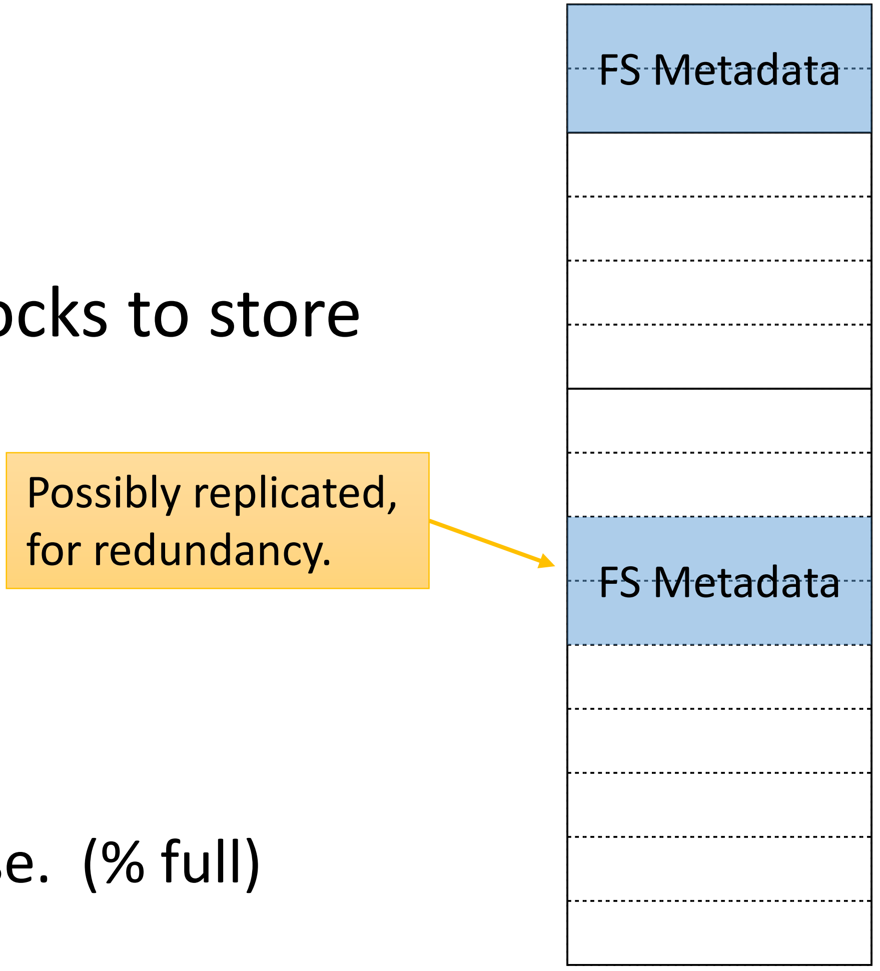
- Data: for the user or programs
- Metadata: for the system to make the file system work

- Both data and metadata stored together on disk!

File System Metadata

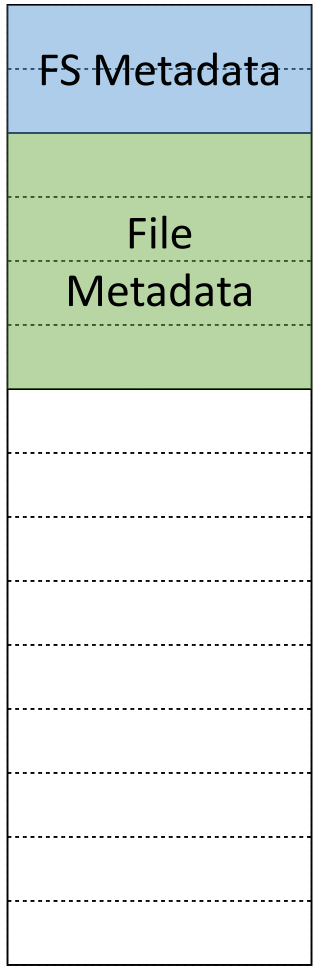
- Information about the FS as a whole.
- For most file systems, use the first few blocks to store global metadata.
- Includes:
  - Type of file system.
  - Block size.
  - Of the remaining blocks, which are free/in use. (% full)

First block: "superblock"



File Metadata

- Next region stores information about *files*.
- **NOT** the contents of the files!

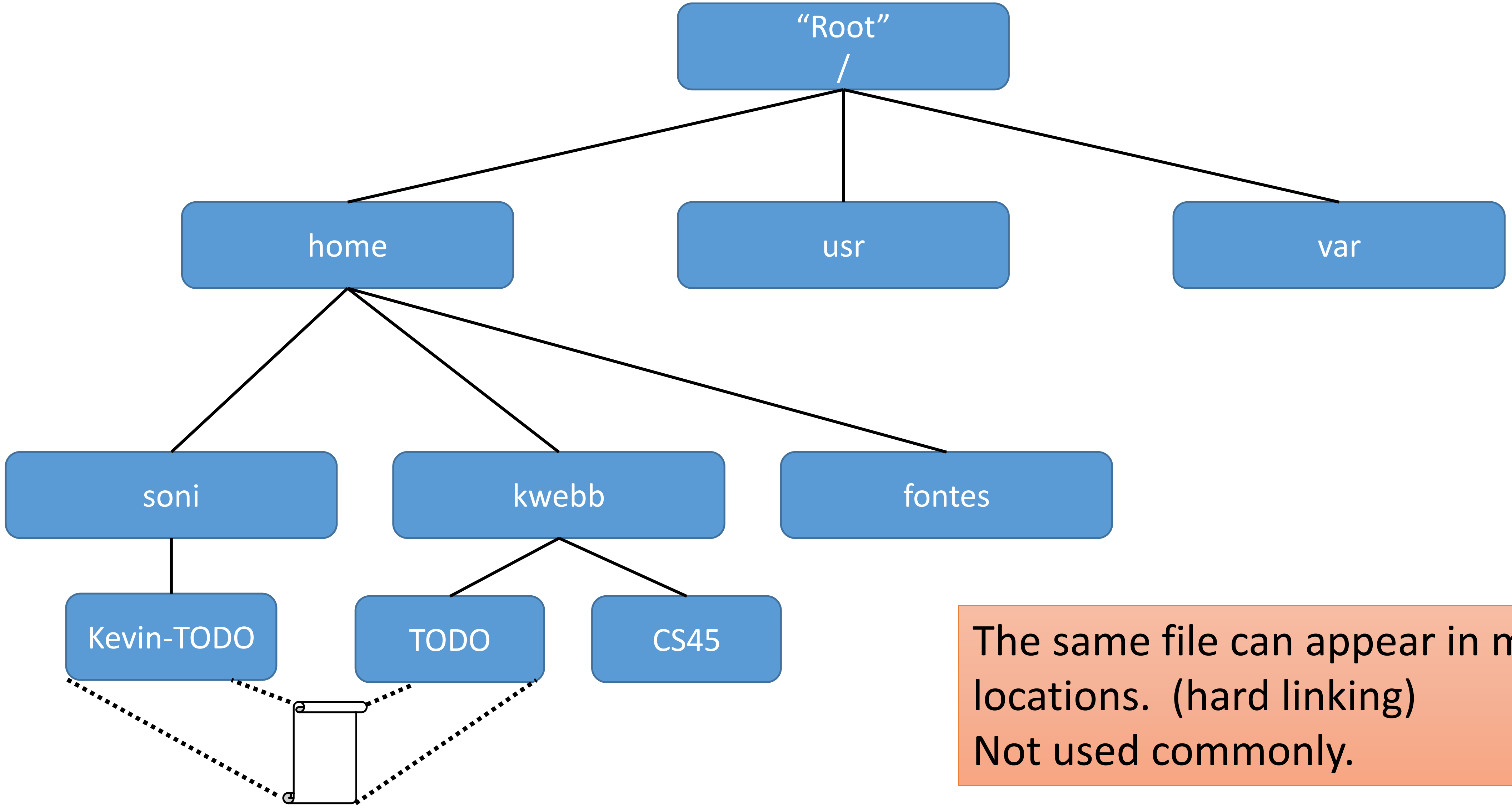


What is a file?

- To the FS, a collection of attributes:
  - Type
  - Times: creation, accessed, modified
  - Sizes: current size, maximum size
  - Structure: where is the content stored on disk?
  - Access control (permissions)
  - Others?

What about the file's name?

FS Abstraction: Hierarchical Name Space - “Tree”



The same file can appear in multiple locations. (hard linking)
Not used commonly.

What about the file's name?

- File name *is* metadata, but it's not associated with the file itself.

- File name comes from the directory it's linked into.

(More on directories later…)

What is a file?

- To the FS, a collection of attributes:
  - Type ⇦ ?
  - Times: creation, accessed, modified
  - Sizes: current size, maximum size
  - Structure: where is the content stored on disk?
  - Access control (permissions)
  - Others?

Regarding file types…

A. The OS distinguishes between file types.

B. The user distinguishes between file types.

C. Both the OS and users distinguish between file types.

D. Nobody distinguishes between file types, files are all just files.

File Type

- To a human user:
  - Is this a music file? text file? video? pdf?
  - To distinguish…
    - name the file with a suffix
    - add special “magic number” in beginning of file (e.g., Java: 0xCAFEBABE)
  - OS does NOT care what your files are. You ask for bytes, it delivers them.

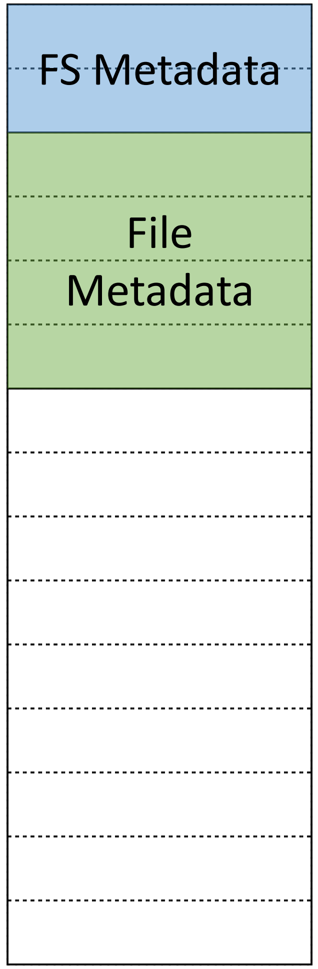
- To the OS:
  - How should I interpret these bytes?
  - regular file? directory? device? FIFO (named pipe)?

File Metadata

- Information about files.
  - Everything we know about a file encapsulated in **inode** structure.
  - Each file has an inode.

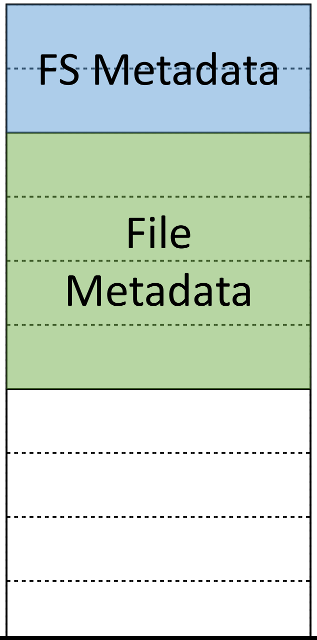
- Typically about 2-5% of blocks reserved for inodes.

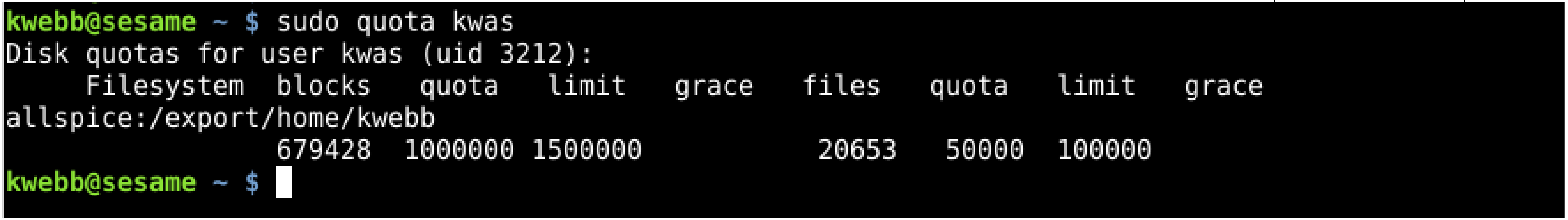
- Every file needs an inode. It includes:
  - Type of file (to the OS).
  - File size.
  - Most recent modification time.
  - Number of times file is hard linked.
  - Block(s) it's stored in on disk.
  - Many more…
  - NOT the name of the file. (it might be linked from multiple directories)



File Metadata

- Information about files.
  - Everything we know about a file encapsulated in **inode** structure.
  - Each file has an inode.
- Typically about 2-5% of blocks reserved for inodes.
  - inodes are a file system resource!



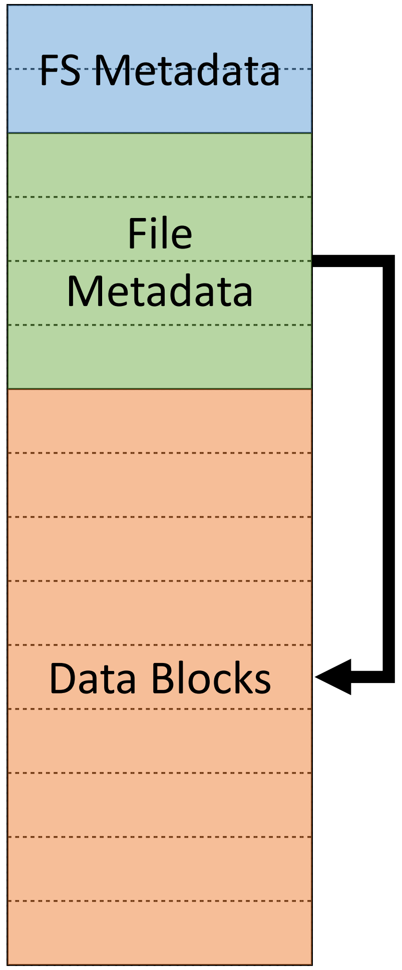


```
kwebb@sesame ~ $ sudo quota kwas
Disk quotas for user kwas (uid 3212):
     Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
allspice:/export/home/kwebb
                 679428  1000000 1500000           20653   50000  100000
kwebb@sesame ~ $ 
```

Data Blocks

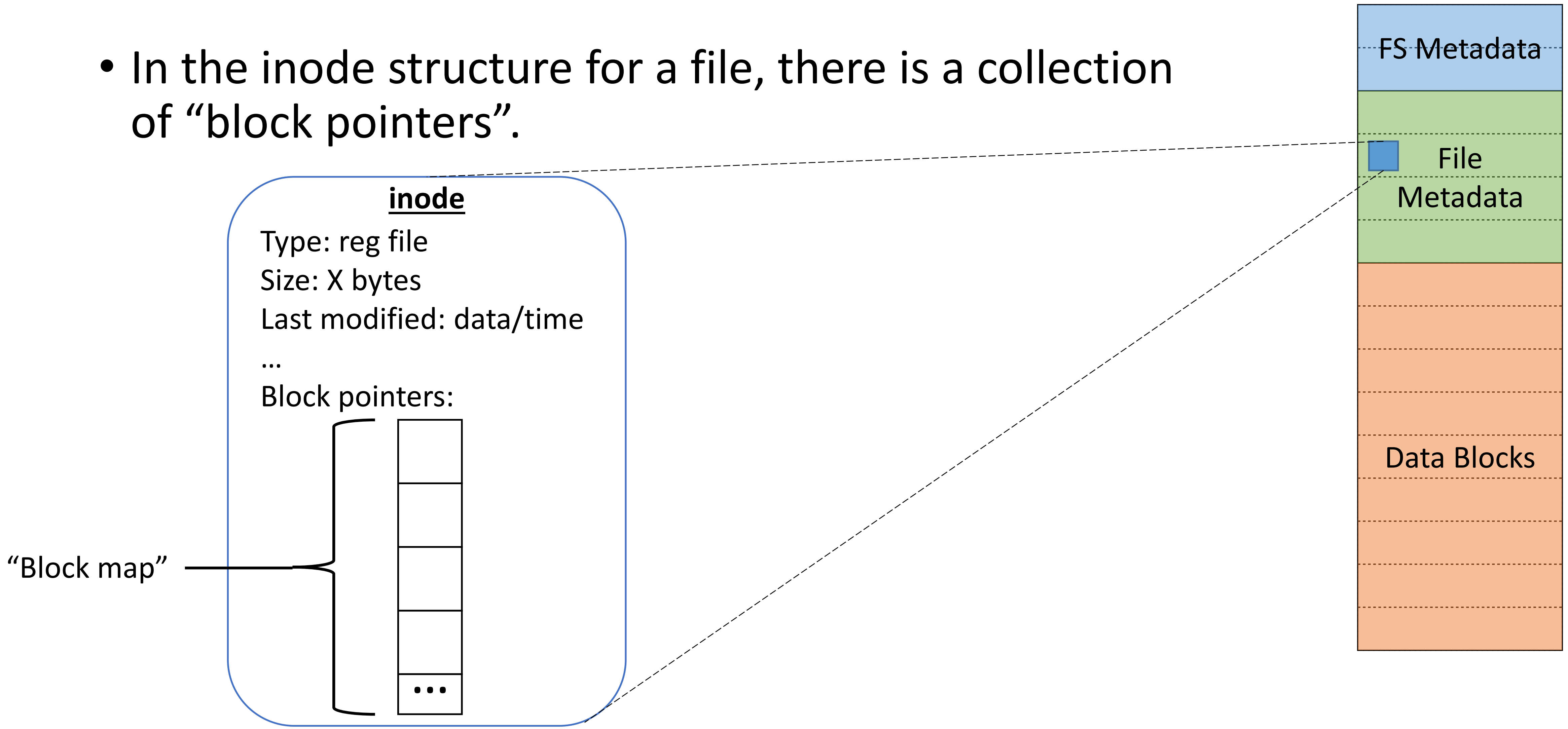
- Rest of disk: data blocks
  - Stores file contents.

- How do we find the data on disk?
  - In file metadata (inode): Block(s) file is stored in on disk.
  - Once we've found the metadata for a file, it will tell us which blocks the data is in.



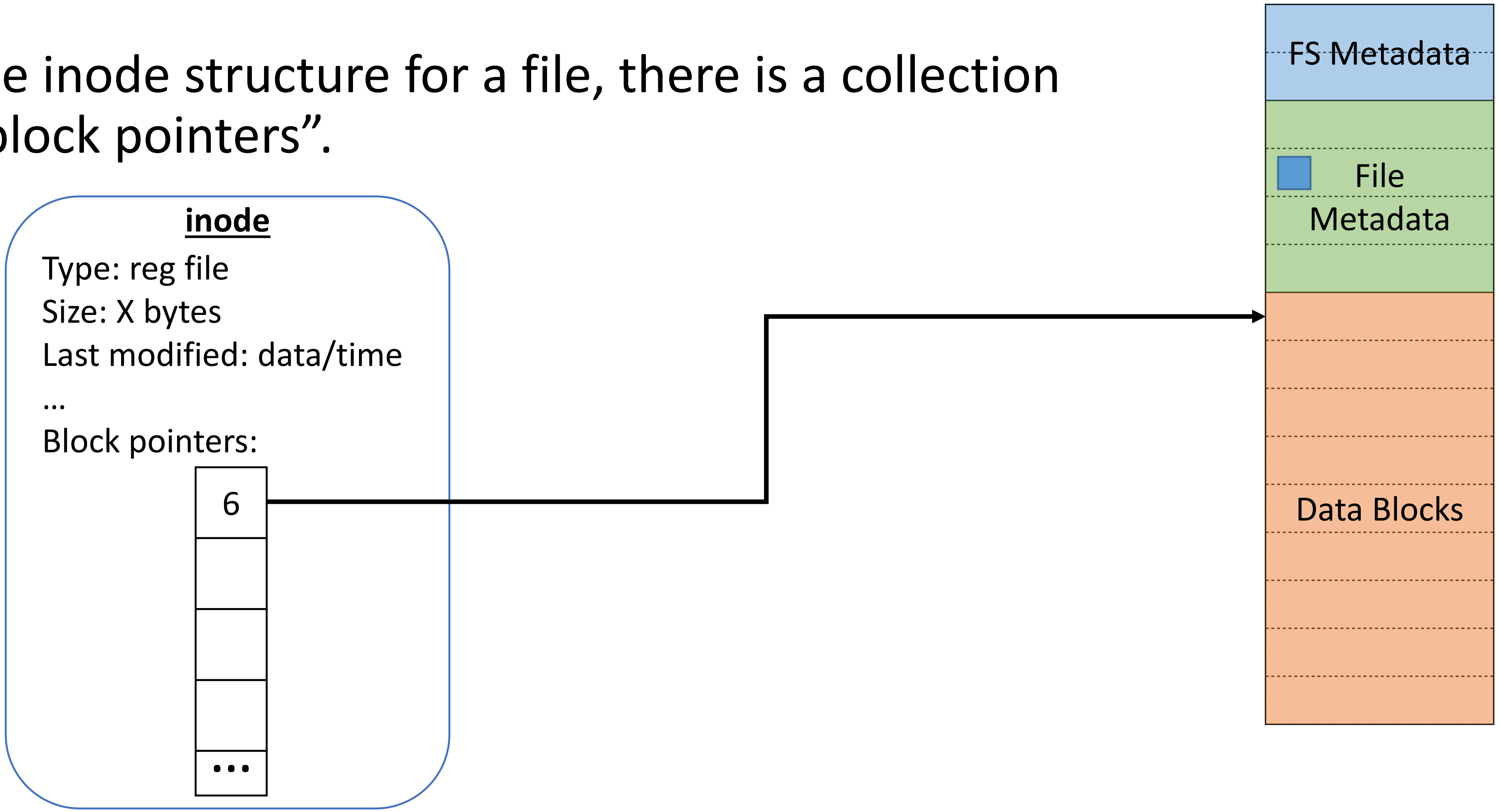
Data Blocks

- In the inode structure for a file, there is a collection of “block pointers”.



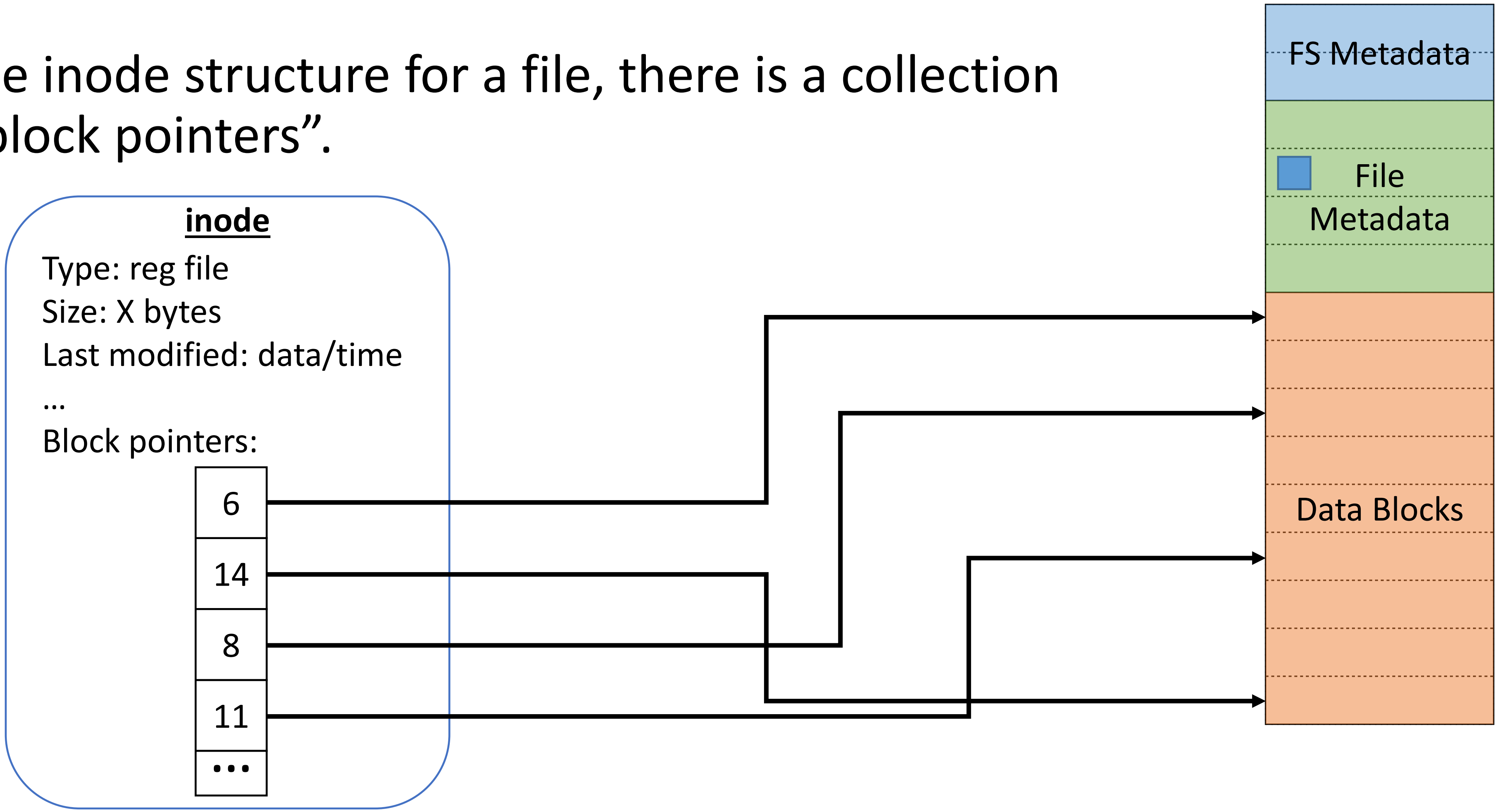
Data Blocks

- In the inode structure for a file, there is a collection of "block pointers".



Data Blocks

- In the inode structure for a file, there is a collection of “block pointers”.



How many block pointers do we need?

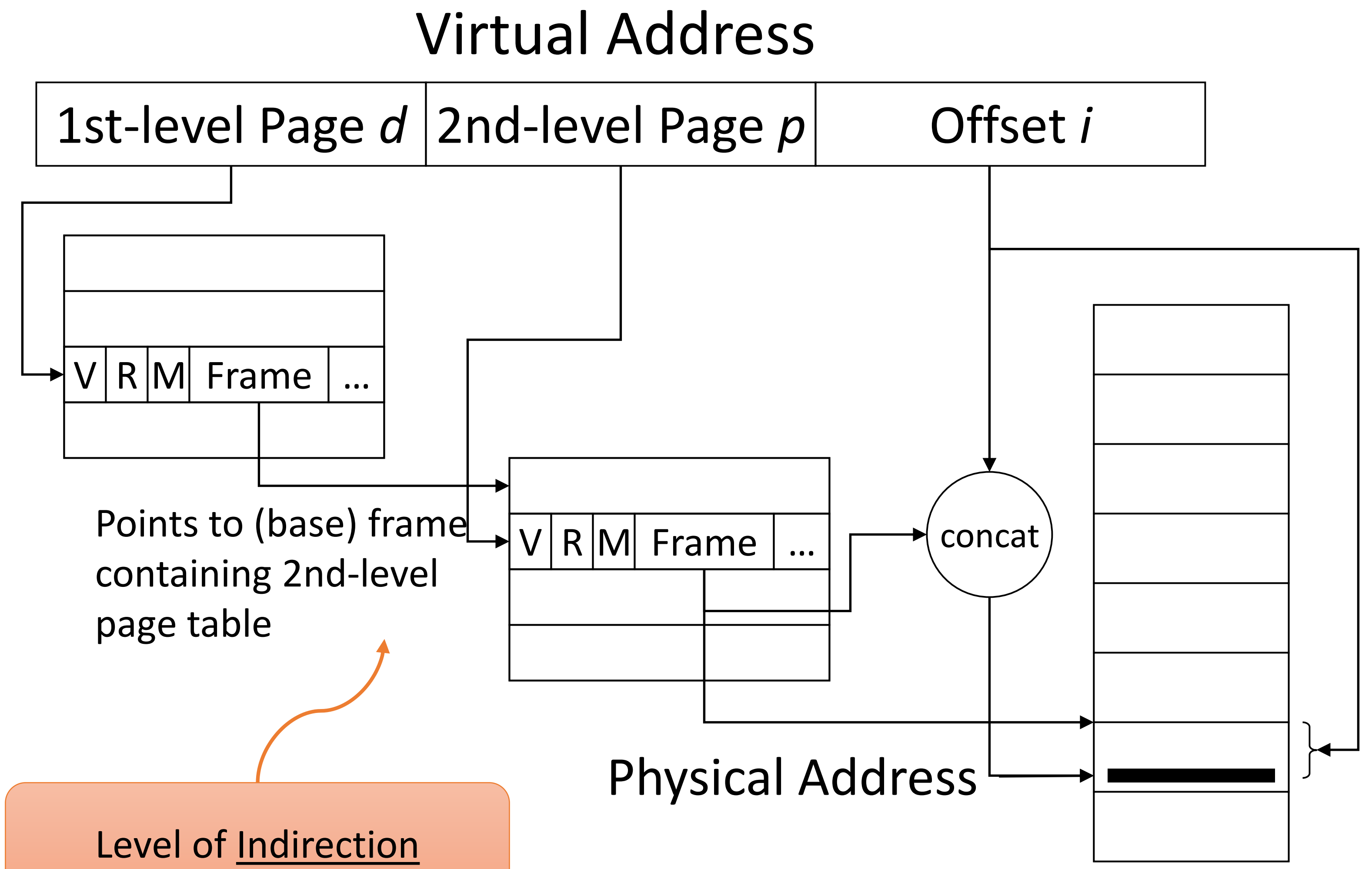
- Suppose one block stores 4 KB (2^{12} bytes) of data.
- Suppose we want an FS to support files up to 64 GB (2^{36} bytes).
- How many block pointers does an inode need?
  - $2^{36} / 2^{12} = 2^{24}$ block pointers
  - If each pointer is 64 bits (8 bytes)…
  - 2^{24} pointers -> 128 MB of block pointers for *every file*.

Hold up…

- Every inode needs 128 MB of pointers?!?
- What if I want to store a small 1 KB file…
- I have to store 128 MB of metadata?

- Problem: FS doesn’t know in advance how big a file is going to be, so we need a lot of block pointers in case it’s big.
- Having lots of pointers makes our inodes large (e.g., 128 MB).
- Result: If we want to store a small file (e.g., 1 KB) it ends up using a lot of disk space.
- <u>What can we do about this?</u>

Recall: Multi-Level Page Tables



Points to (base) frame containing 2nd-level page table

Level of Indirection

Insight: VAS is typically sparsely populated.

Idea: every process gets a page directory (1st-level table)

Only allocate 2nd-level tables when the process is using that VAS region!

Block Map (Traditional FS Design)

- inode contains total 13 pointers (104 bytes per inode!)
  - 10 direct: references 10 data blocks
  - 1 singly-indirect: references n data blocks
  - 1 doubly-indirect: references n^2 data blocks
  - 1 triply-indirect: references n^3 data blocks
- For a data block of 4096 bytes
  - Assuming pointer requires 8 bytes, n = 512
  - Max file size: $(10 + 512 + 512^2 + 512^3) * 4096 \approx 512$ GB

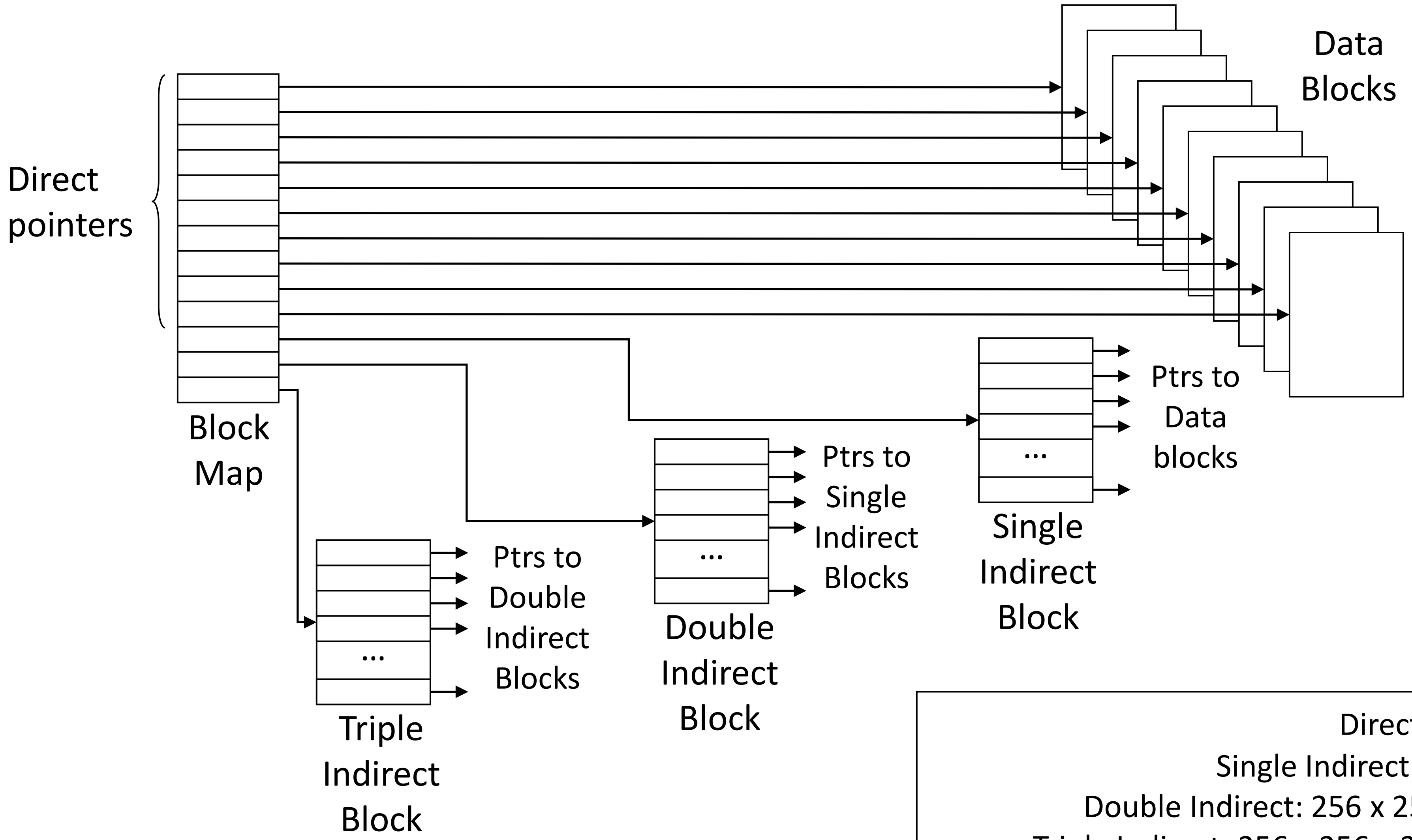
MUCH larger max file size.

inode is now WAY smaller:
104 byte block map vs. 128 MB

Storage space for file metadata (inode's block map) now scales with file size.

This all sounds great!
Is it free?

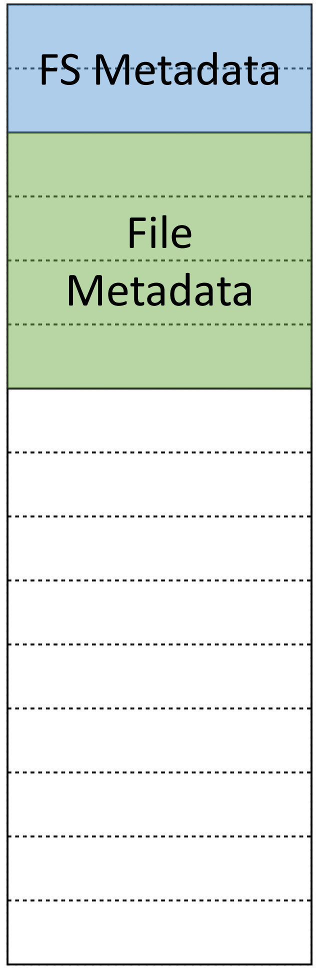
Block Pointers



Direct: 10 x 4 KB = 40 KB
Single Indirect: 256 x 4 KB = 1 MB
Double Indirect: 256 x 256 x 4 KB = 256 MB
Triple Indirect: 256 x 256 x 256 x 4 KB = 512 GB

File Metadata (inode) Summary

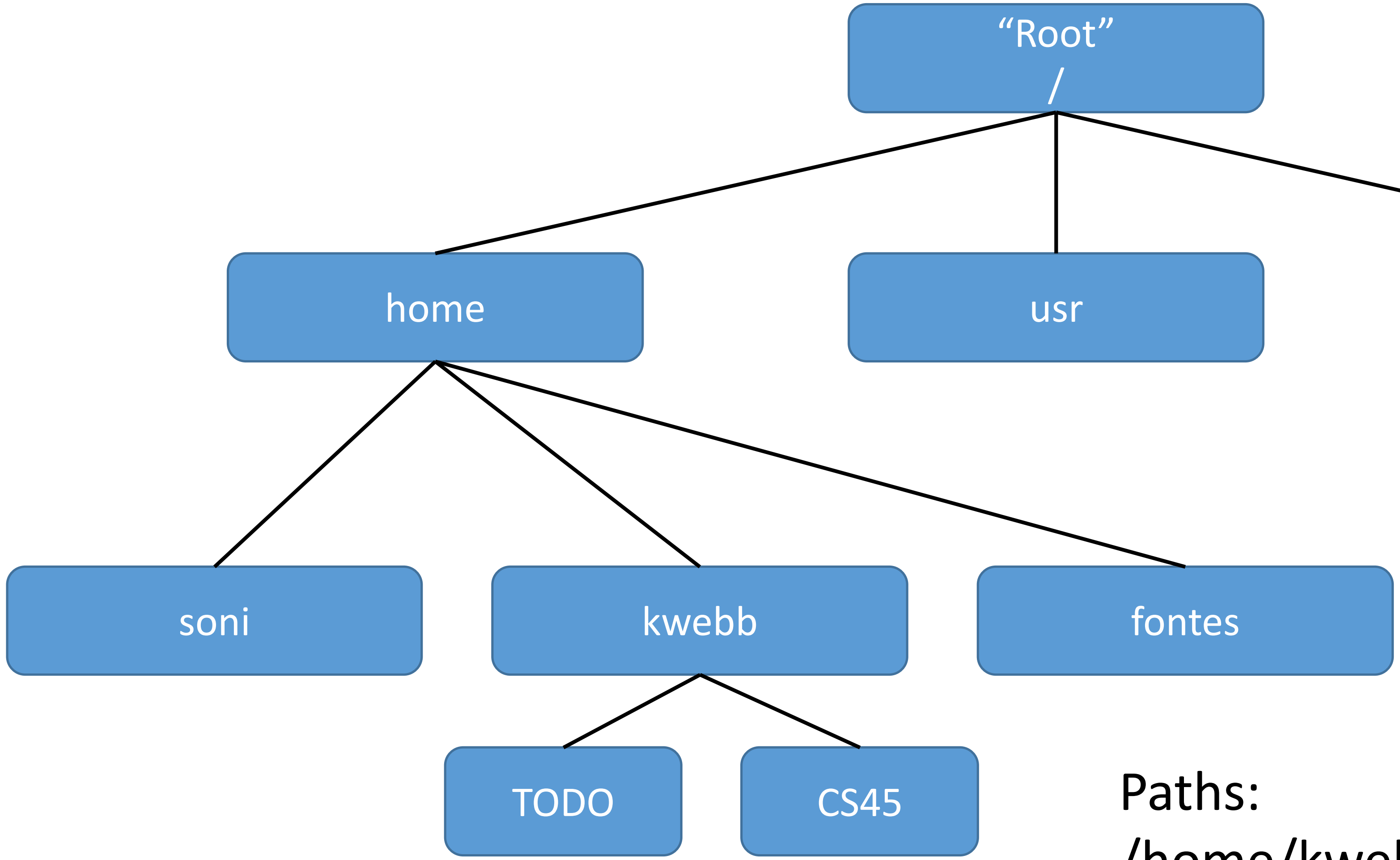
- Information about files.
  - Everything we know about a file encapsulated in **inode** structure.
  - Each file has an inode.
- Typically about 2-5% of blocks reserved for inodes.
- Every file needs an inode. It includes:
  - Type of file (to the OS).
  - File size.
  - Most recent modification time.
  - Number of times file is hard linked.
  - Block(s) it's stored in on disk. (**block map**)
  - Many more…
  - NOT the name of the file. (it might be linked from multiple directories)



What's in a name?

So far, missing:

file path (name) -> inode



Query path

⬇

Retrieve file info/data

Paths:
/home/kwebb/TODO
/home/soni

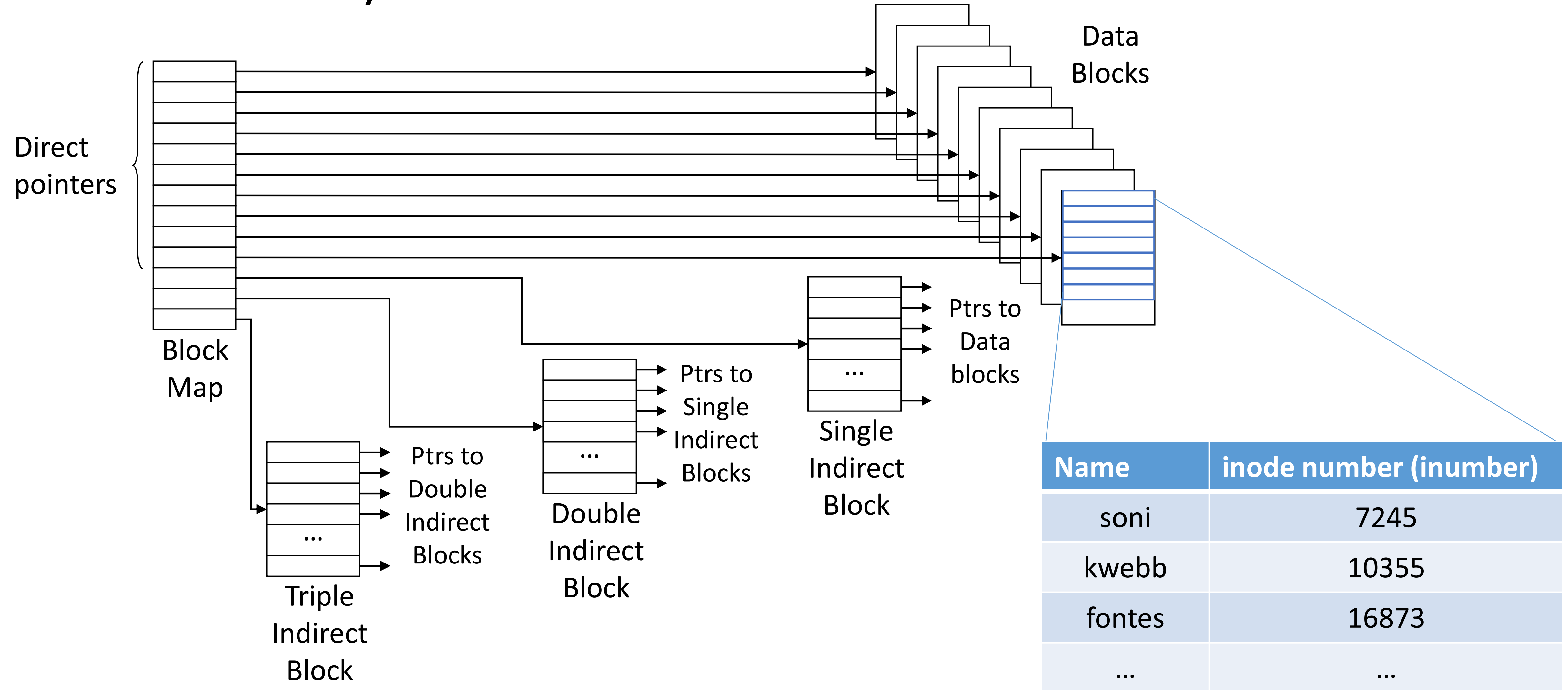
Directories

- A directory is a file.
  - That is, just like a regular file, it has an inode, with attributes and a block map.

- What's different/special about a directory?

- The OS interprets the data of a directory differently.
  - Rather than ignoring the data and just handing to users (regular files)…
  - Directory contains a collection of mappings: name -> inode number

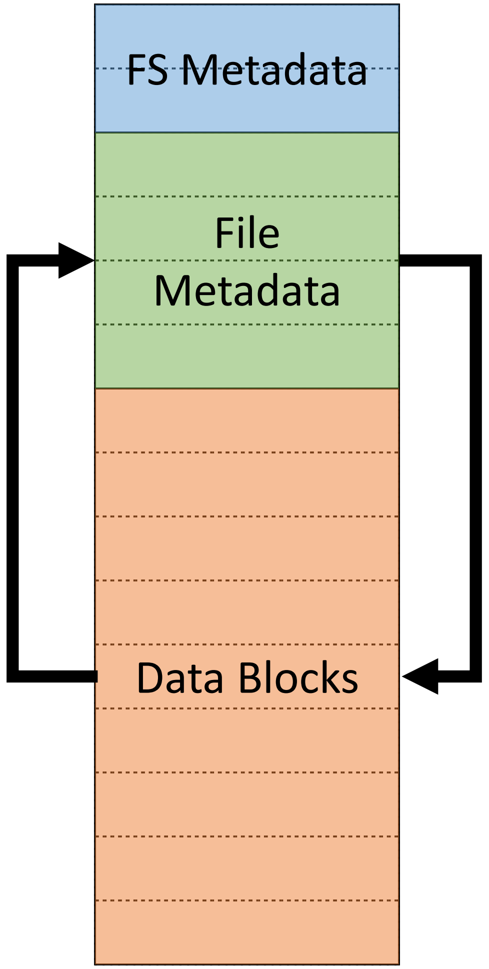
Directory



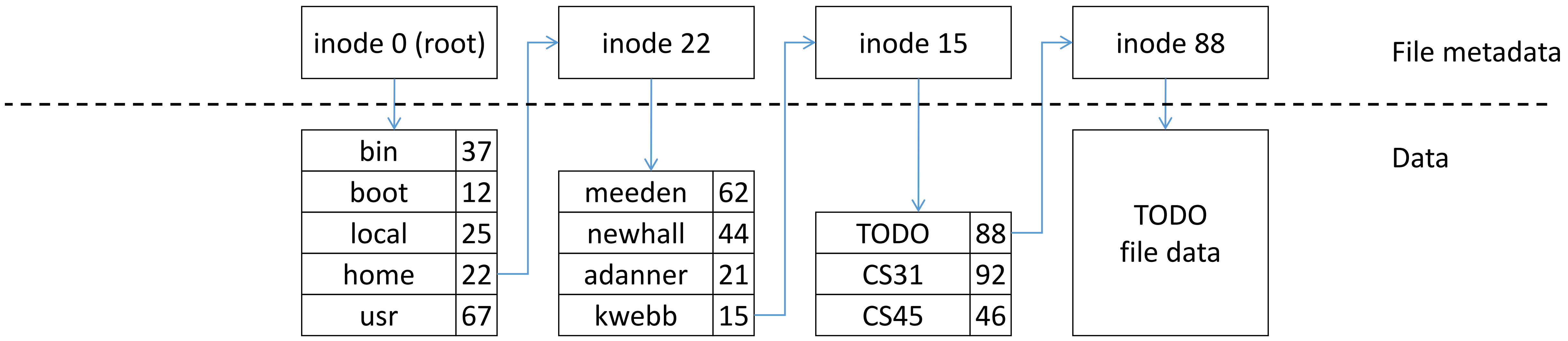
Name	inode number (inumber)
soni	7245
kwebb	10355
fontes	16873
…	…

Data Blocks

- Rest of disk: data blocks
- Stores user file and directory content.
  - Once we’ve found the metadata for a file, it will tell us which blocks the data is in.
  - If the data is a directory, it will refer to other file metadata.



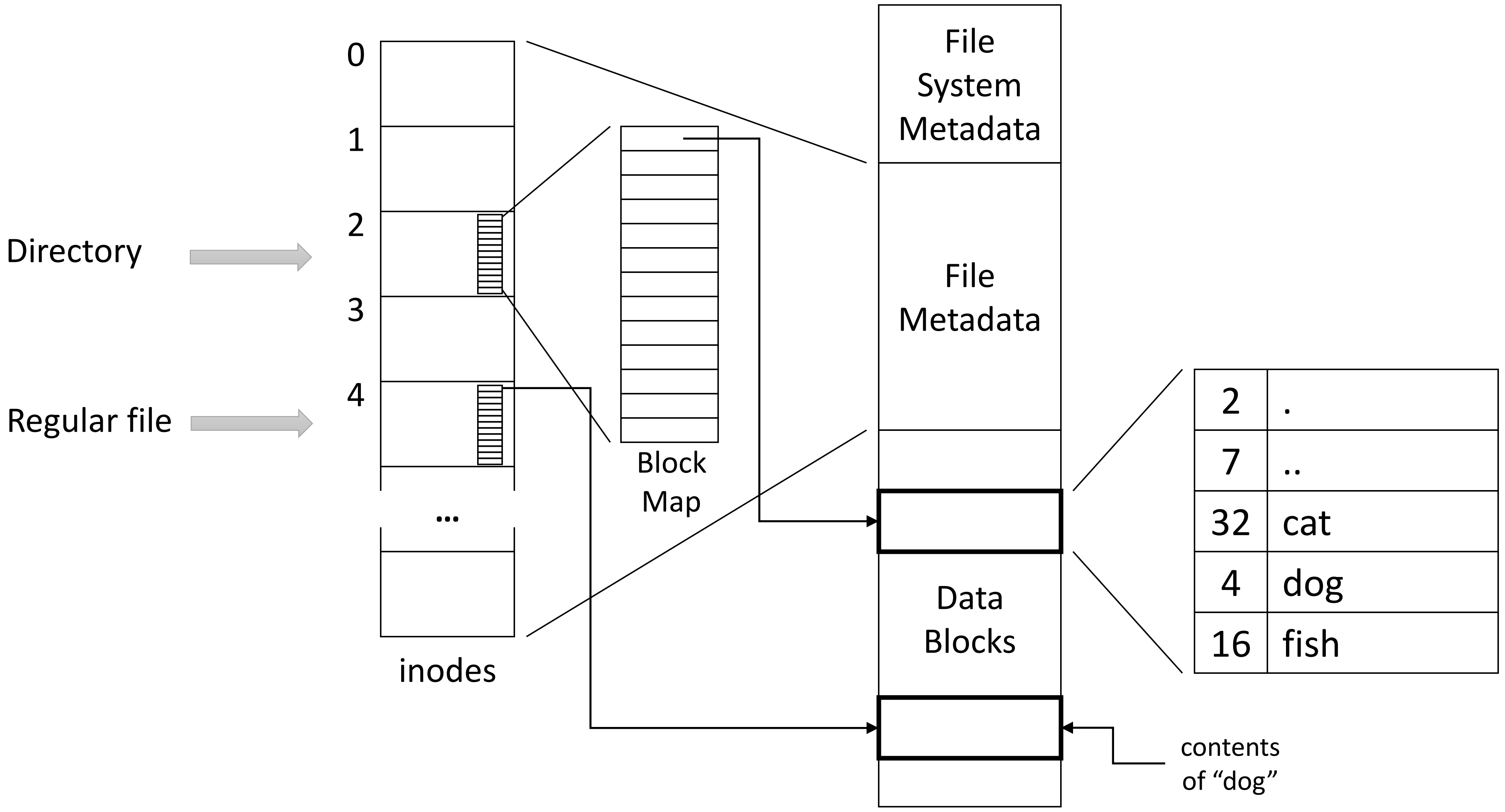
Directory Lookup Example



- Given pathname: /home/kwebb/TODO
  1. Inode 0 block map points to data block(s) of root directory
  2. Look up “home” in root directory to get inode 22
  3. Inode 22 block map points to data block(s) of home directory
  4. Look up “kwebb” in home directory to get inode 15

  …

The Big Picture



Summary

- Modern disk interface let’s OS read/write numbered blocks.
- File system’s goal is to add nice user abstractions on top of blocks.
- We focused on two main file types (to the OS):
  - Regular file: user data, OS doesn’t care what it contains
  - Directory: maps names to inodes in the file system