CS202 (003): Operating Systems
File System II

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

Last Time

Directories

"Spend all day generating data, come back the next morning, want to use it.”

— F. Corbató, on why files and directories are invented.

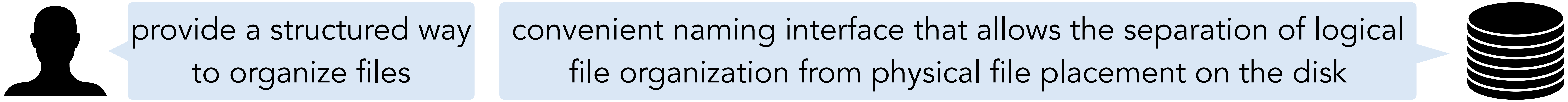
How to achieve this?

Users remember where on disk their files are (disk sector no.)?…

Use human-friendly names to represent files

That’s why directories exist

map names to file blocks on the disk



provide a structured way to organize files

convenient naming interface that allows the separation of logical file organization from physical file placement on the disk

Short history of directories

Single directory for the entire system

Put directory at a known disk location

Directory contains pairs of <name, inumber>

If one user uses a name, no one else can

Ancient computer’s style

Single directory for each user

Still clumsy, and ls on 10,000 files is a real pain

Hierarchical name spaces

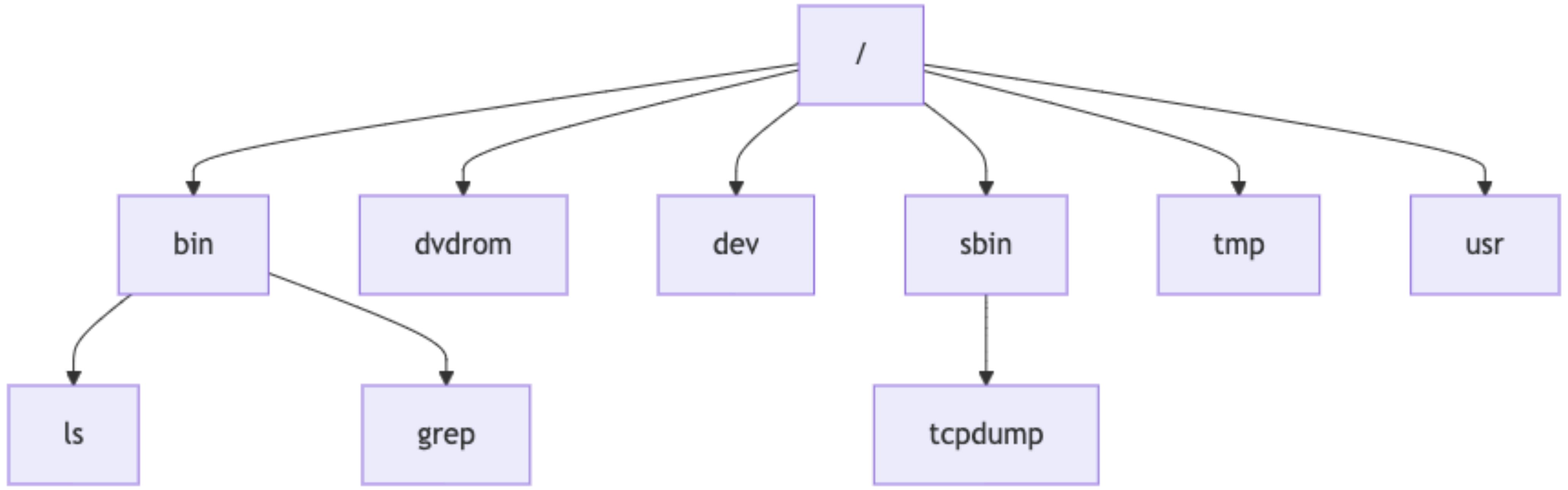
Allow directory to map names to files or other directories

FS forms a tree (or graph, if links allowed)

Large name spaces tend to be hierarchical

Hierarchical Unix

Used since CTSS (1960s), unix picked it up and used quite nicely



Directories stored on the disk just like regular files

A directory is a list of entries (tuple, location is typically the inode #)

i-node for directory contains a special flag bit, only special users can write directory files

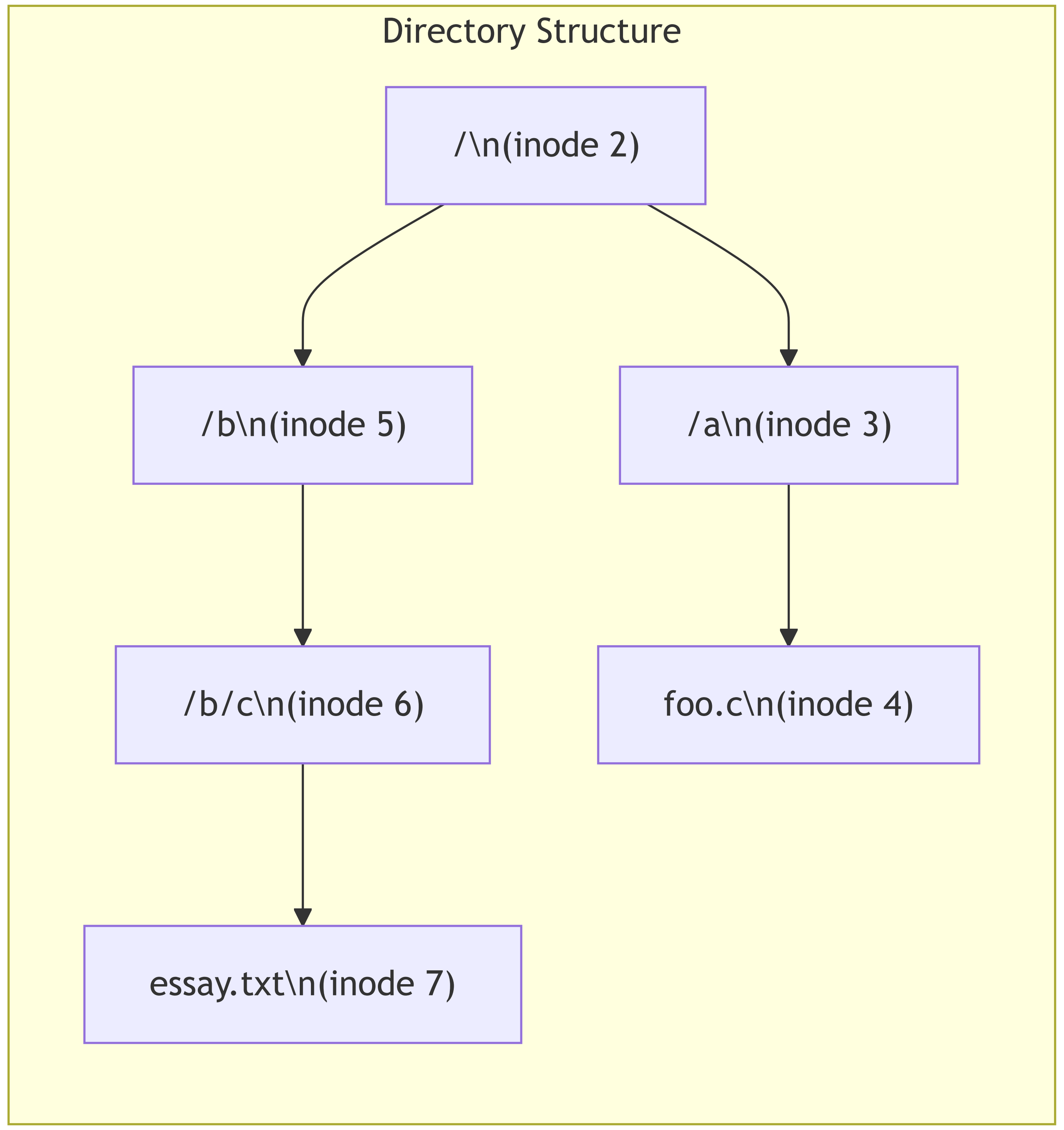
Key point: inode # might reference another directory
=> neatly turn the FS to a hierarchical tree

Naming Magic

- **Bootstrapping: Where do you start looking?**
  - Root directory always inode #2 (0 and 1 historically reserved)
- **Special names:**
  - Root directory: “`/`”
  - Current directory: “`.`”
  - Parent directory: “`..`”
- **Some special names are provided by shell, not FS:**
  - User’s home directory: “∼”
  - Globbing: “`foo.*`” expands to all files starting “`foo.`”
- **Using the given names, only need two operations to navigate the entire name space:**
  - `cd` *name*: move into (change context to) directory *name*
  - `ls`: enumerate all names in current directory (context)

https://www.scs.stanford.edu/19wi-cs140/notes/file_systems.pdf

Example

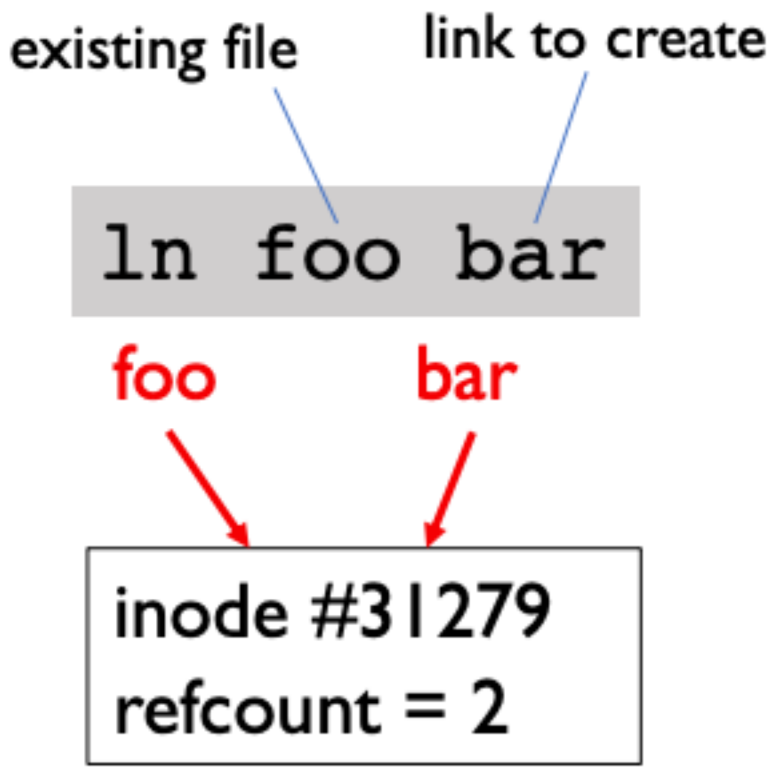

Directory Structure
/\n(inode 2)
/b\n(inode 5)
/a\n(inode 3)
/b/c\n(inode 6)
foo.c\n(inode 4)
essay.txt\n(inode 7)

Hard and soft links

Hard link

Multiple directory entries point to the same node.

Unix stores count of pointers to inode



What happens if one link is removed? the data are still accessible through any other link that remains

What happens if all links are removed? the space occupied by the data is freed

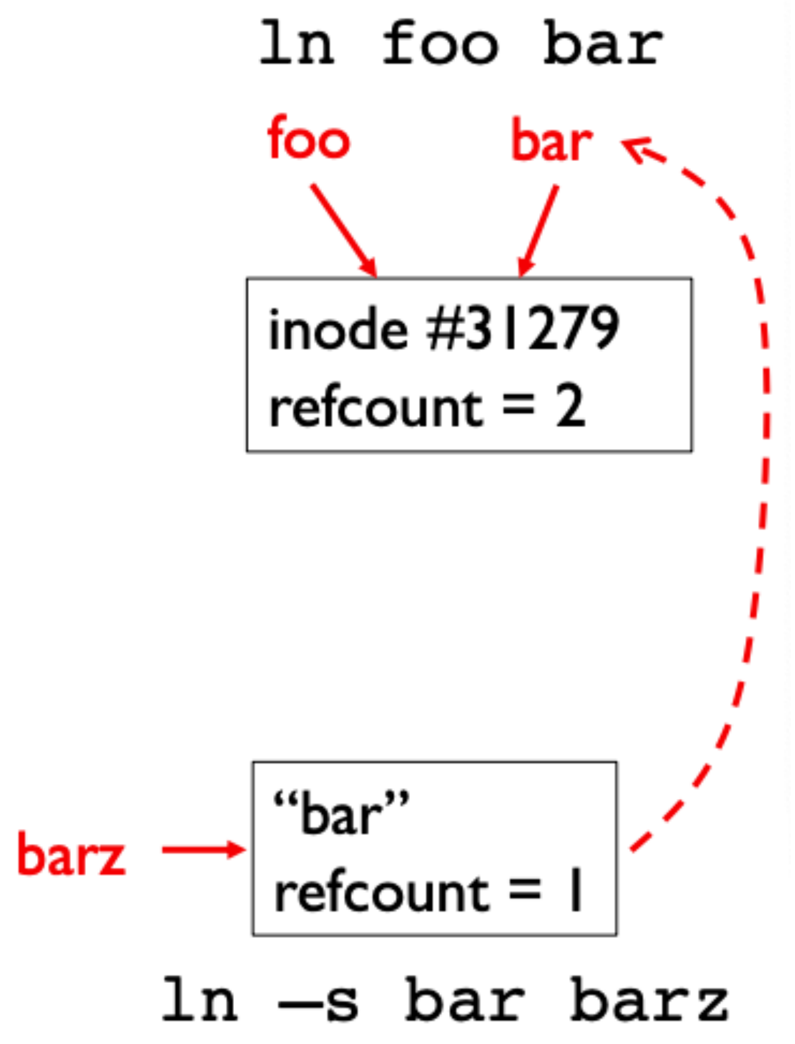
Hmmm, what happen if there are cycles? can't create hard link to directories

Soft link

(i.e. "name")

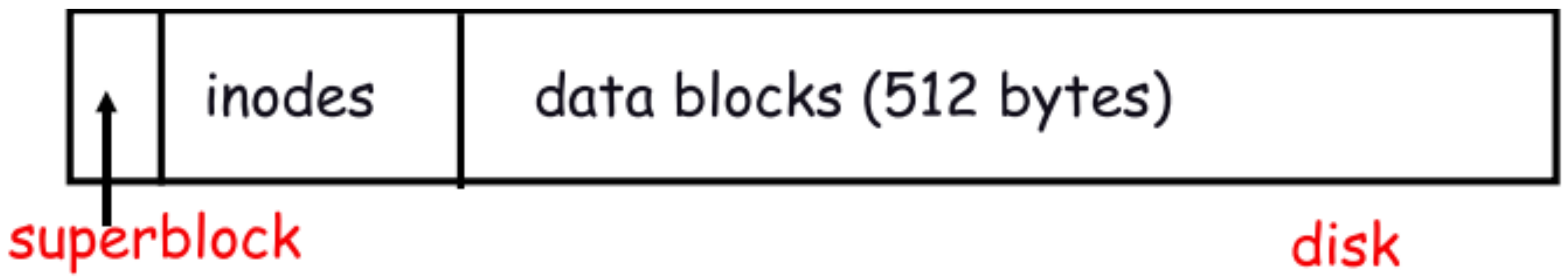
Point to a file/dir name, but the "point-to" file/dir might not exist

Create a new node with a special "symlink" bit set and contains name (path) of the linked target



Performance

Unix FS is simple and element, but… also slow



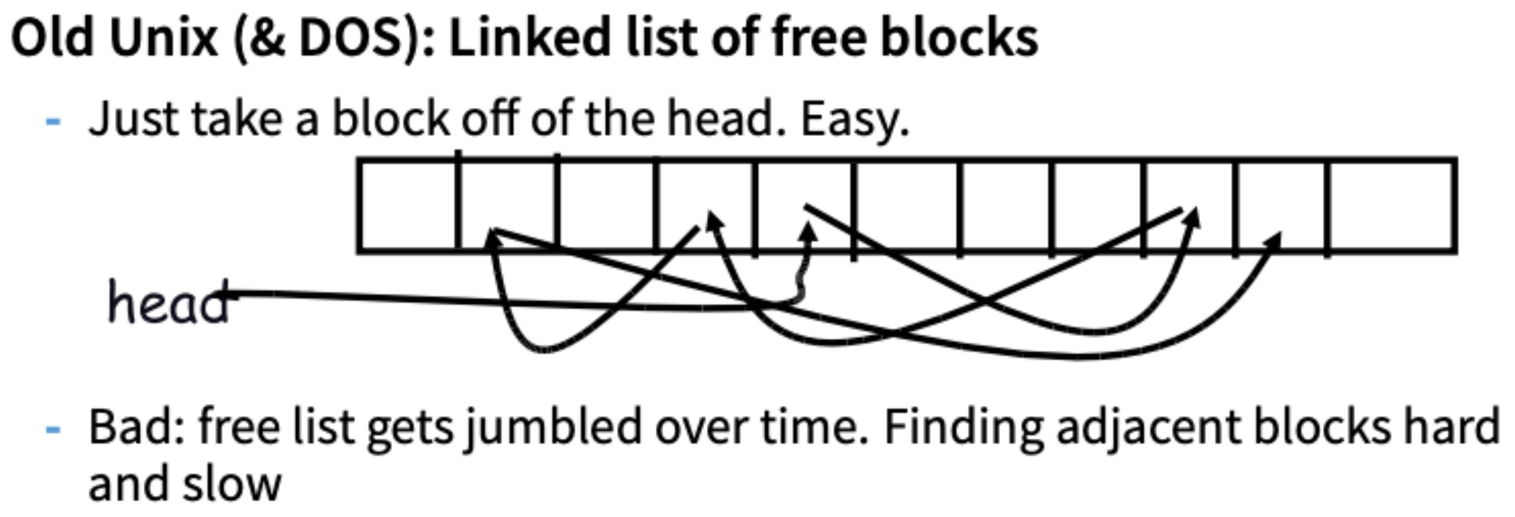
- Blocks too small (512 bytes)
- Inode has:
  - Too many layers of mapping indirection
  - Transfer rate low (get one block at time)

Poor clustering of related objects:

- Consecutive file blocks not close together
- Inodes far from data blocks (all at the beginning of the disk)
- Inodes for directory not close together
- Poor enumeration performance: e.g., “ls”, “grep foo *.c”

Usability problems:

- 14-character file names a pain
- Can’t atomically update file in crash-proof way



Old Unix (& DOS): Linked list of free blocks

- Just take a block off of the head. Easy.

- Bad: free list gets jumbled over time. Finding adjacent blocks hard and slow

FFS fixes these problems to a certain degree

larshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1984. A fast file system for UNIX. ACM Trans. Comput. Syst. 2, 3 (Aug. 1984), 181–197. https://doi.org/10.1145/989.990

Fast File System

What can we do to improve?

make block size bigger (4 KB, 8KB, or 16 KB)

cluster files in the same directory

make data blocks and inodes closer to each other

bitmaps to track free blocks (store separately)

reserve 10% space (user don’t know about it)

improving consistency (atomic rename, symbolic links, …)

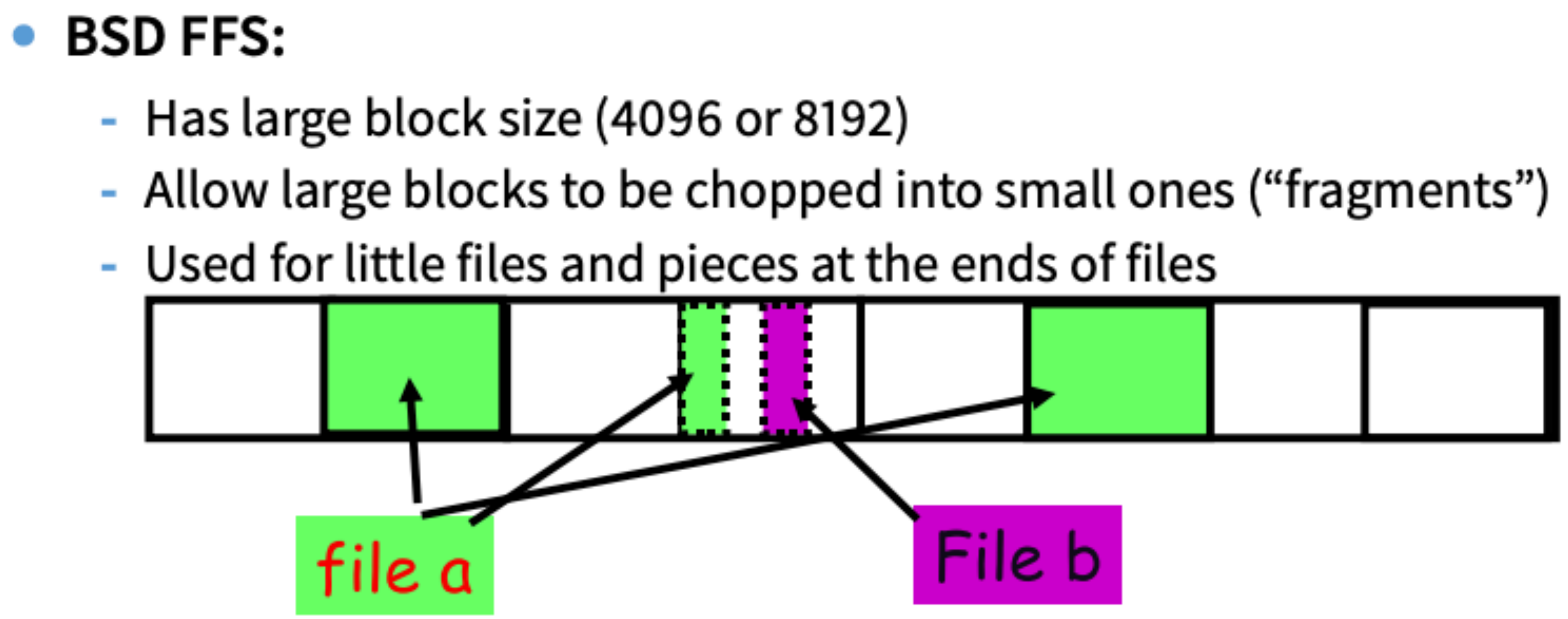
https://www.scs.stanford.edu/19wi-cs140/notes/file_systems.pdf

Making block size bigger

What happen if the data is smaller than the block size?

Data transfer overhead increases

Block wastage (internal fragmentation) increases



- **BSD FFS:**
  - Has large block size (4096 or 8192)
  - Allow large blocks to be chopped into small ones (“fragments”)
  - Used for little files and pieces at the ends of files

https://www.scs.stanford.edu/19wi-cs140/notes/file_systems.pdf

FFS: Use of Bitmaps

Bitmaps to track free blocks (store separately)

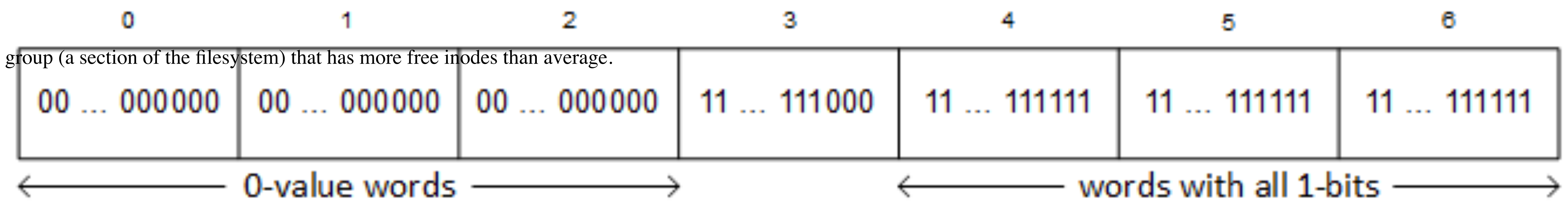
easier to find contiguous blocks
can keep the entire thing in memory

```
b = (number of bits per word) * (number of 0-value words) + offset to first 1 bit
```

(c)

- `bit[b] == 1` ⇒ `block[b]` is free
- `bit[b] == 0` ⇒ `block[b]` is allocated or in use

new directories are being placed in a cylinder group (a section of the filesystem) that has more free inodes than average.



0	1	2	3	4	5	6
00 ... 000000	00 ... 000000	00 ... 000000	11 ... 111000	11 ... 111111	11 ... 111111	11 ... 111111

(d)

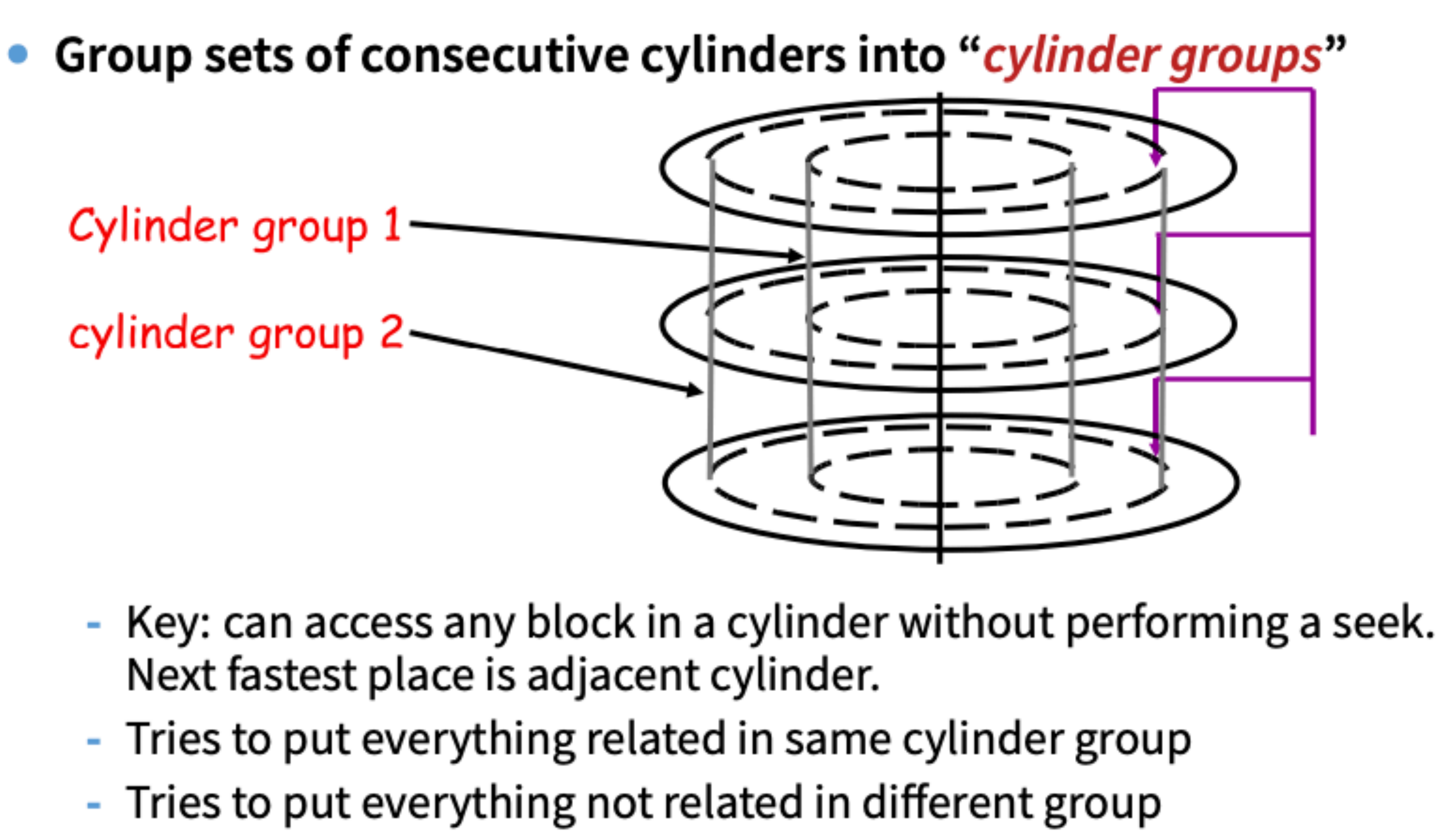
$$\frac{2^{41}\text{ bytes}}{\text{disk }\cancel{\text{file system}}} \times \frac{1\text{ block}}{2^{12}\text{ bytes}} \times \frac{1\text{ bit}}{\text{block}} \times \frac{1\text{ byte}}{8\text{ bits}}$$

$$2^{41}/2^{15} = 2^{26}\text{ bytes} = \boxed{64\text{ MB}}$$

2 TB disk / 4KB disk blocks
= 500,000,000 entries = 60MB.

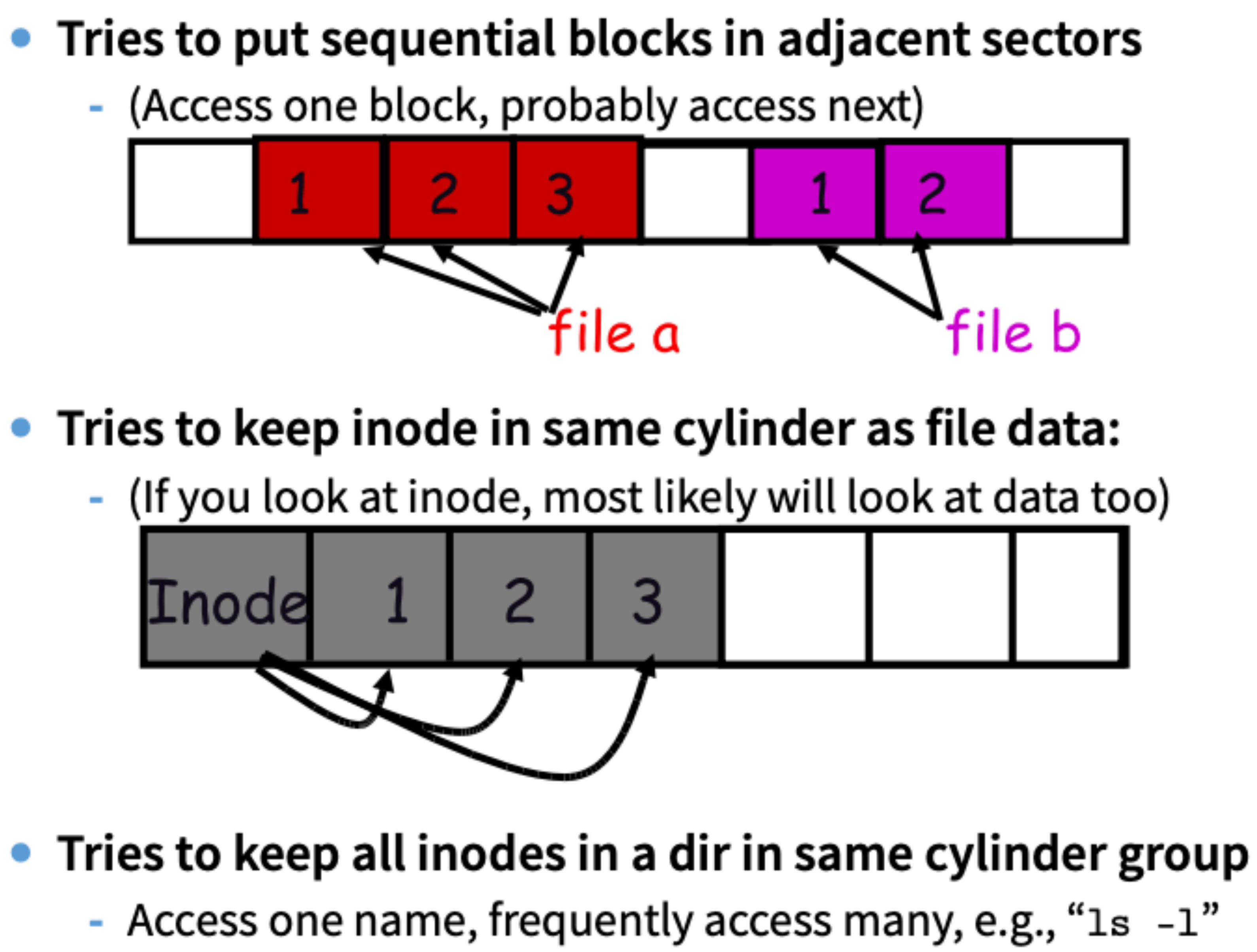
https://www.scs.stanford.edu/19wi-cs140/notes/file_systems.pdf

FFS: Clustering related objects



- Group sets of consecutive cylinders into "*cylinder groups*"

  - Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
  - Tries to put everything related in same cylinder group
  - Tries to put everything not related in different group



- Tries to put sequential blocks in adjacent sectors
  - (Access one block, probably access next)

- Tries to keep inode in same cylinder as file data:
  - (If you look at inode, most likely will look at data too)

- Tries to keep all inodes in a dir in same cylinder group
  - Access one name, frequently access many, e.g., "`ls -l`"

Each cylinder group basically a mini-Unix file system

```
[superblock | bookkeeping info | inodes | bitmap | data blocks (512 bytes each) ]
```

New directories are being placed in a cylinder group (a section of the filesystem) that has more free inodes than average.

When file grows too big send its remainder to dierent cylinder group.

https://www.scs.stanford.edu/19wi-cs140/notes/file_systems.pdf

FFS: Performance

20-40% of disk bandwidth for large files

10-20x of original Unix file system!

Still not the best we can do
(meta-data writes happen synchronously, which really hurts performance.
but making asynchronous requires a story for crash recovery.)

https://www.scs.stanford.edu/19wi-cs140/notes/file_systems.pdf

FFS: Other hacks

Big file cache

kernel maintains a **buffer cache** in memory

```
ReadDiskCache(blockNum, readbuf) {
   ptr = buffercache.get(blockNum);
   if (ptr) {
      copy BLKSIZE bytes from ptr to readbuf
   } else {
      newBuf = malloc(BLKSIZE);
      ReadDisk(blockNum, newBuf);
      buffercache.insert(blockNum, newBuf);
      copy BLKSIZE bytes from newBuf to readbuf
   }
```

Reduce rotation delay

No rotation delay if you're reading the whole track.

Work with big chunks

Read ahead in big chunks (64KB)
Write in big chunks

https://www.scs.stanford.edu/19wi-cs140/notes/file_systems.pdf

Lab 4 is due tomorrow!