PERSISTENCE: FILE SYSTEM STRUCTURES

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

- Midterm 2 Grades in Canvas
- Make-up Points for Projects 2 + 3 (more later) – Canvas Quizzes
  - Specification and Concepts/Implementation
    - Can take only one time
    - Up to 35 more points (scaled);
      - if > 70 points, no benefit
      - if received < 50 points, expect to take
  - Due 1 week from when made available (Fri, Mon)
- Project 6
  - Due next Friday
  - Specification Quiz – Due by tomorrow 6pm (idea: before Discussion Section)
- Discussion this week:
  - Project 6 - MapReduce

AGENDA / LEARNING OUTCOMES

What **on-disk structures** represent files and directories?

Contiguous, Extents, Linked, FAT, Indexed, **Multi-level indexed**

What disk **operations** are needed for:

make directory

open file

write/read file

close file

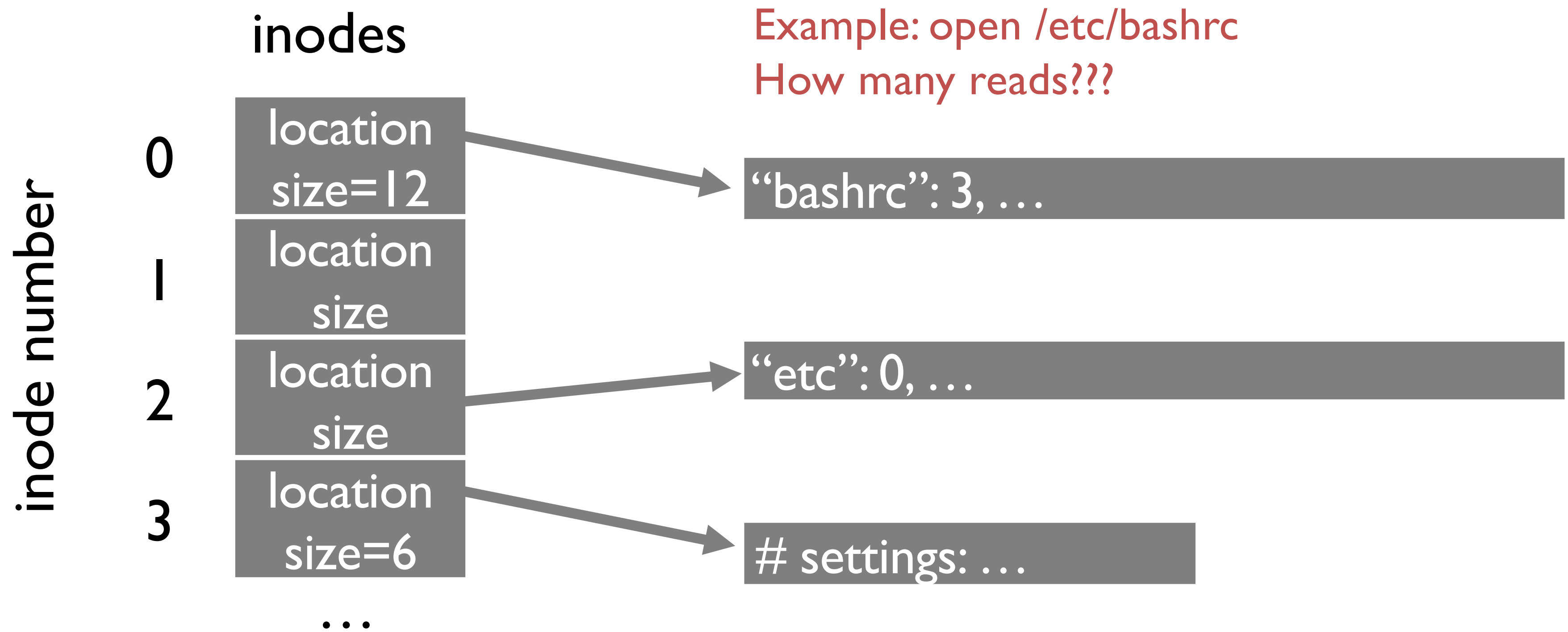
RECAP

FILE API WITH FILE DESCRIPTORS

```
int fd = open(char *path, int flag, mode_t mode)
read(int fd, void *buf, size_t nbyte)
write(int fd, void *buf, size_t nbyte)
close(int fd)
```

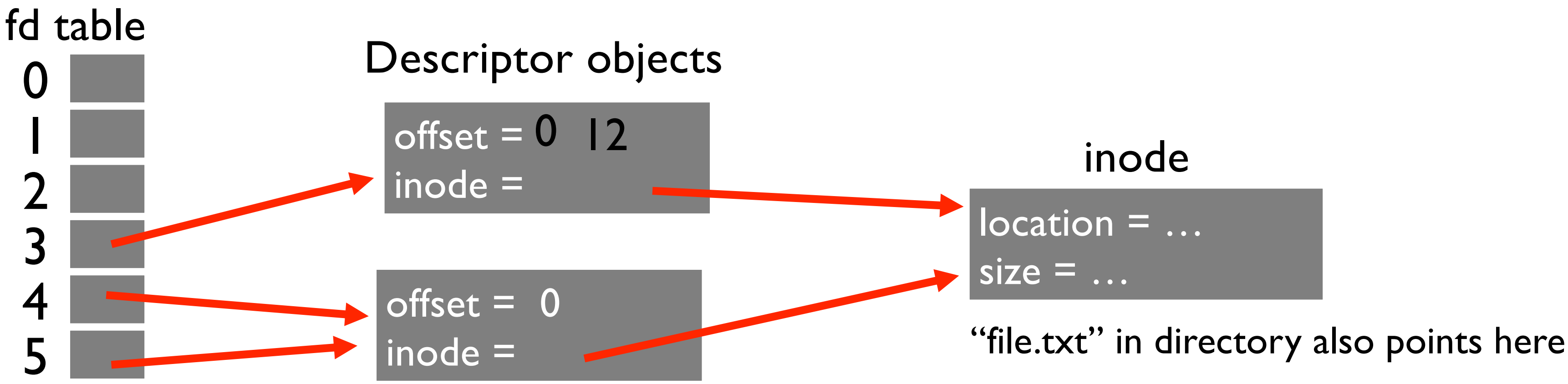
advantages:
- string names
- hierarchical
- traverse once
- offsets precisely defined

Example: open /etc/bashrc
How many reads???



1. Inode #2 → Get location of root directory
2. Read root directory data; see “etc” maps to inode 0
3. Inode #0 → Get location of etc directory
4. Read /etc directory; see “bashrc” is at inode 3
5. Inode #3 → Get location of /etc/bashrc file data
6. Read /etc/bashrc file data

CODE SNIPPET: OPEN VS. DUP




```
int fd1 = open("file.txt"); // returns 3
read(fd1, buf, 12);
int fd2 = open("file.txt"); // returns 4
int fd3 = dup(fd2);         // returns 5
```

DELETING FILES

There is no system call for deleting files!

Inode (and associated file) is **garbage collected** when there are no references

Paths are deleted when: `unlink()` is called

→ File not deleted when other hard links point to it

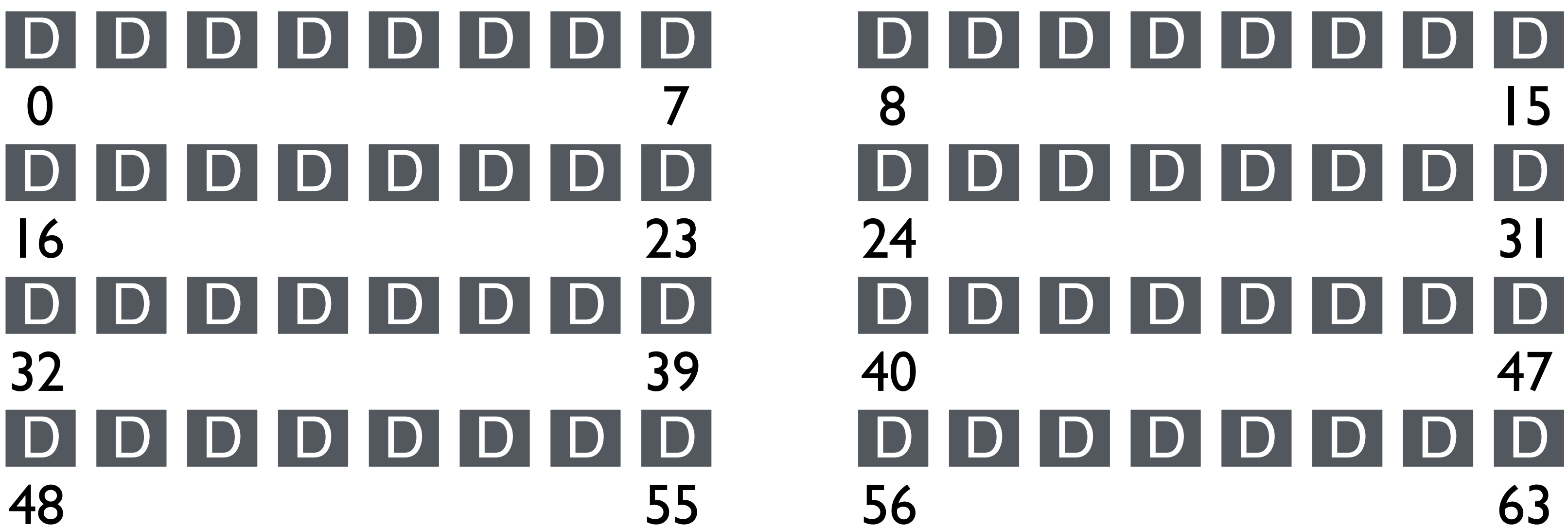
→ File could be deleted when only a symbolic link points to it

FDs are deleted when: `close()` or process quits

→ File not deleted if any process has it open

FILESYSTEM DISK STRUCTURES

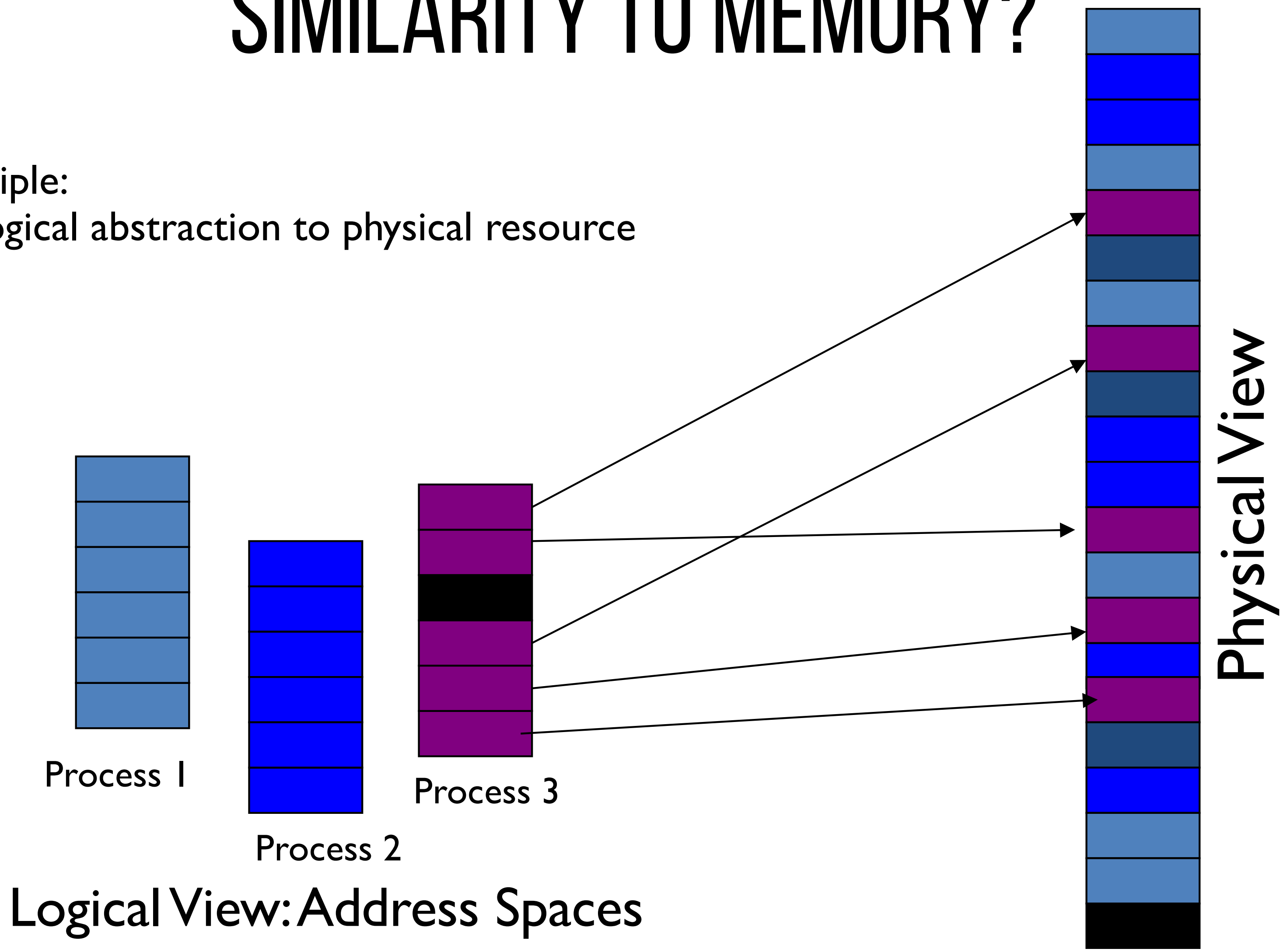
FS STRUCTS: HOW TO USE DISK BLOCKS?



Assume each block is 4KB

SIMILARITY TO MEMORY?

Same principle:
map logical abstraction to physical resource



ALLOCATION STRATEGIES

Many different approaches

- Contiguous
- Extent-based
- Linked
- File-allocation Tables
- Indexed
- Multi-level Indexed

WHAT METRICS MATTER?

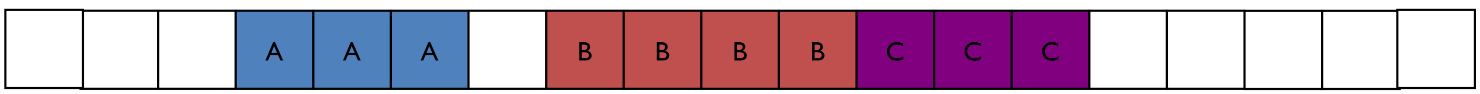
Questions

- Amount of fragmentation (internal and external)
- Ability to grow file over time?
- Performance of sequential accesses (contiguous layout)?
- Speed to find data blocks for random accesses?
- Wasted space for meta-data overhead (everything that isn't data)?
  - Meta-data must be stored persistently too!

CONTIGUOUS ALLOCATION

Allocate each file to contiguous sectors on disk

- Meta-data: Starting block and size of file
- OS allocates by finding sufficient free space
    - Must predict future size of file; Should space be reserved?
- Example: IBM OS/360

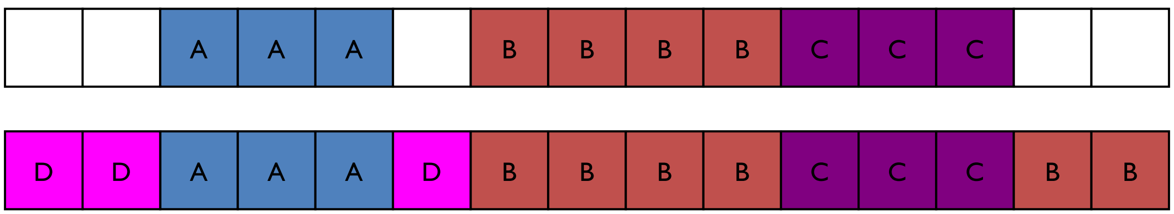


Fragmentation (internal and external)?	- Horrible external frag (needs periodic compaction)
Ability to grow file over time?	- May not be able to without moving
Seek cost for sequential accesses?	+ Excellent performance
Speed to calculate random accesses?	+ Simple calculation
Wasted space for meta-data?	+ Little overhead for meta-data

SMALL # OF EXTENTS

Allocate multiple contiguous regions (extents) per file

– Meta-data: Small array (2-6) designating each extent
Each entry: starting block and size



Fragmentation (internal and external)? +/- Helps external fragmentation (until out of extents)

Ability to grow file over time? +/- Can grow (until run out of extents)

Seek cost for sequential accesses? + Still good performance (generally)

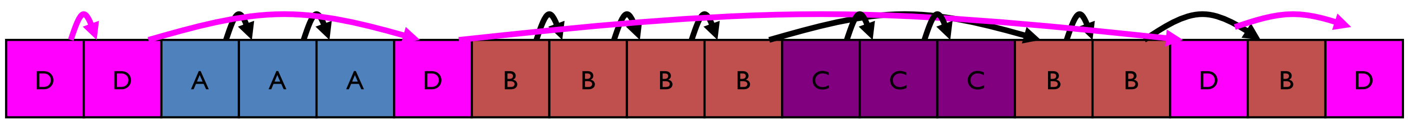
Speed to calculate random accesses? + Still simple calculation

Wasted space for meta-data? + Still small overhead for meta-data

LINKED ALLOCATION

Allocate linked-list of **fixed-sized** blocks (multiple sectors)

- Meta-data: Location of first block of file
  Each block also contains pointer to next block
- Examples: TOPS-10, Alto



Fragmentation (internal and external)? + No external frag (use any block); internal?

Ability to grow file over time? + Can grow easily

Seek cost for sequential accesses? +/- Depends on data layout

Speed to calculate random accesses? - Ridiculously poor

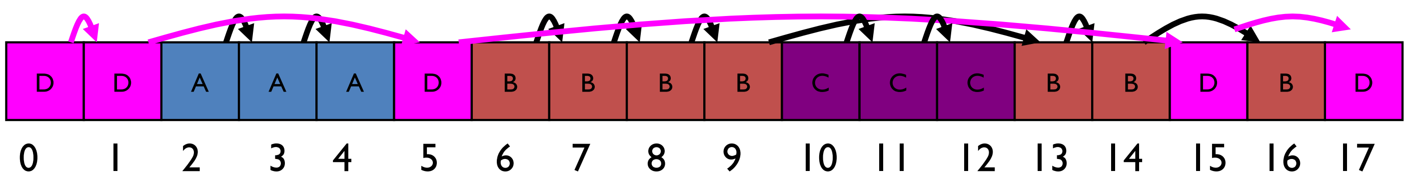
Wasted space for meta-data? - Waste pointer per block

Trade-off: Block size (does not need to equal sector size)

FILE-ALLOCATION TABLE (FAT)

Variation of Linked allocation

- Keep linked-list information for all files in on-disk FAT table
- Meta-data: Location of first block of file
  - And, FAT table itself



Draw corresponding FAT Table?

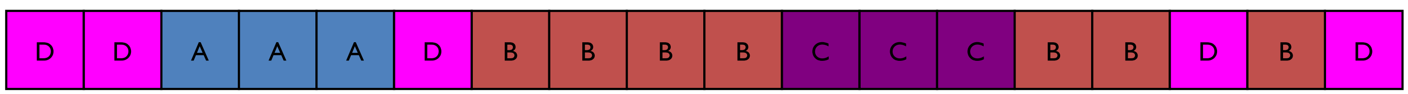
Comparison to Linked Allocation

- Same basic advantages and disadvantages
- Disadvantage: Read from two disk locations for every data read
- Optimization: Cache FAT in main memory
  - Advantage: Greatly improves random accesses
  - What portions should be cached? Scale with larger file systems?

INDEXED ALLOCATION

Allocate fixed-sized blocks for each file

- Meta-data: Fixed-sized array of block pointers
- Allocate space for ptrs at file creation time



Advantages

- No external fragmentation
- Files can be easily grown up to max file size (determined by number of block pointers)
- Supports random access

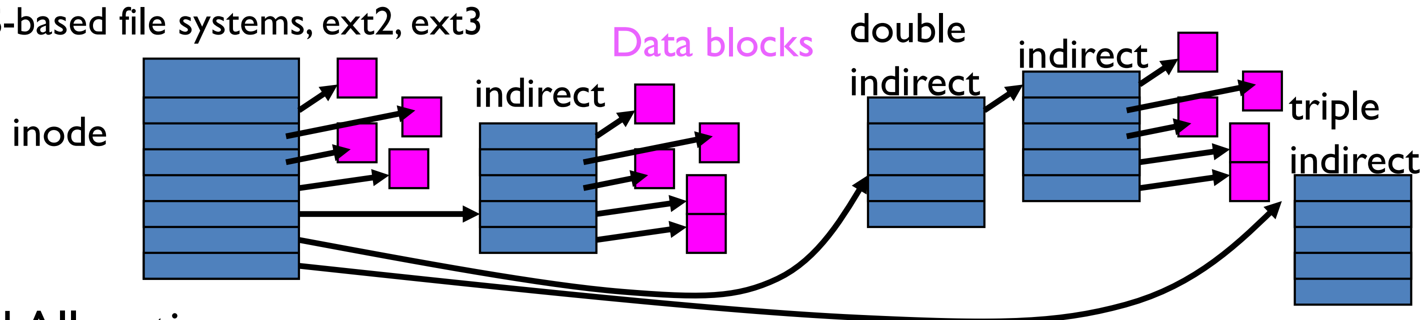
Disadvantages

- Large overhead for meta-data:
  - Wastes space for unneeded pointers (most files are small!)

MULTI-LEVEL INDEXING

Variation of Indexed Allocation

- Dynamically allocate hierarchy of pointers to blocks as needed
- Meta-data: Small number of pointers allocated statically
  - Additional pointers to blocks of pointers
- Examples: UNIX FFS-based file systems, ext2, ext3



Comparison to Indexed Allocation

- Advantage: Does not waste space for unneeded pointers
  - + Still fast access for small files
  - + Can grow to very large size
- Disadvantage: Need to read indirect blocks of pointers to find addresses (extra disk read)
  - Keep indirect blocks cached in main memory (esp for sequential)

FLEXIBLE # OF EXTENTS

Modern file systems:

Dynamic multiple contiguous regions (extents) per file

- Organize extents into multi-level tree structure
  - Each leaf node: starting block and contiguous size for data
  - Minimizes meta-data overhead when have few extents
  - Allows growth beyond fixed number of extents

Fragmentation (internal and external)?	+ Both reasonable
Ability to grow file over time?	+ Can grow
Seek cost for sequential accesses?	+ Still good performance
Speed to calculate random accesses?	+/- Some calculations depending on size
Wasted space for meta-data?	+ Relatively small overhead

ASSUME MULTI-LEVEL INDEXING

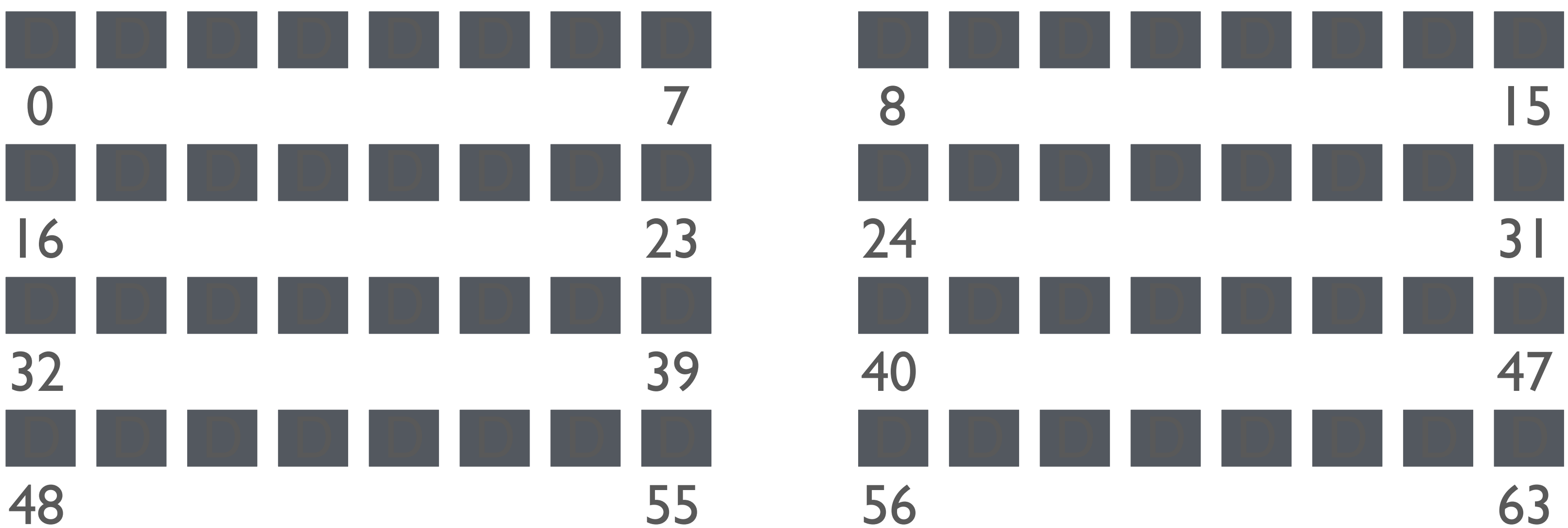
Simple approach

More complex file systems build from these basic data structures

ON-DISK STRUCTURES

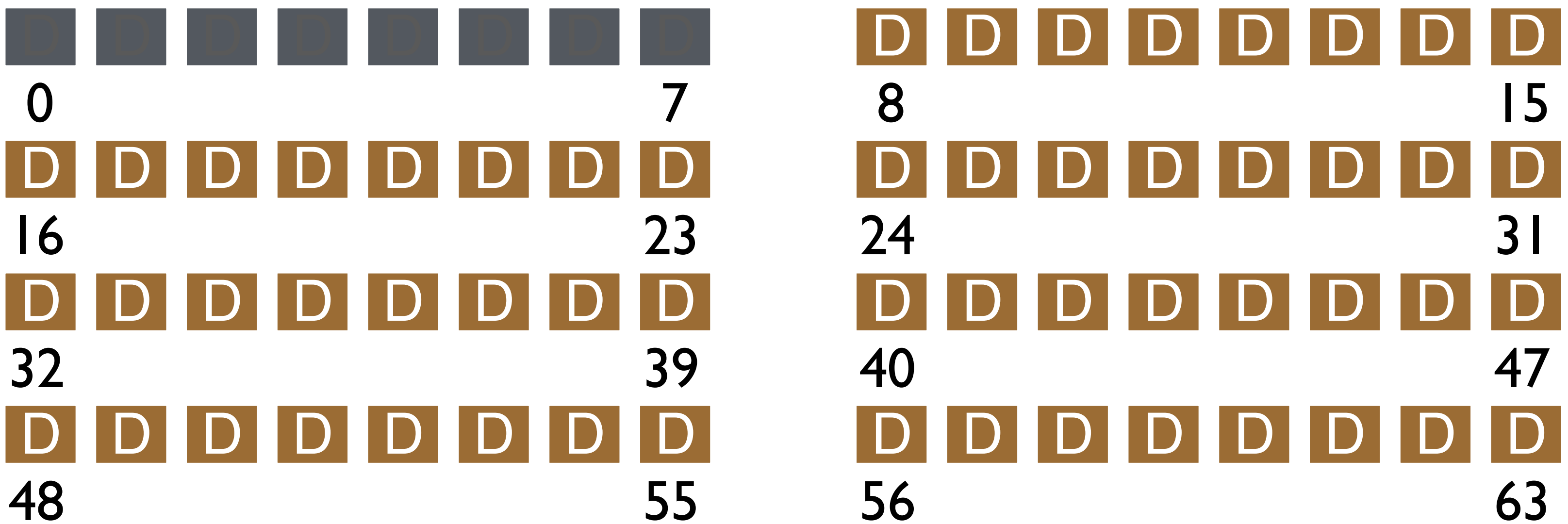
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

FS STRUCTS: EMPTY DISK



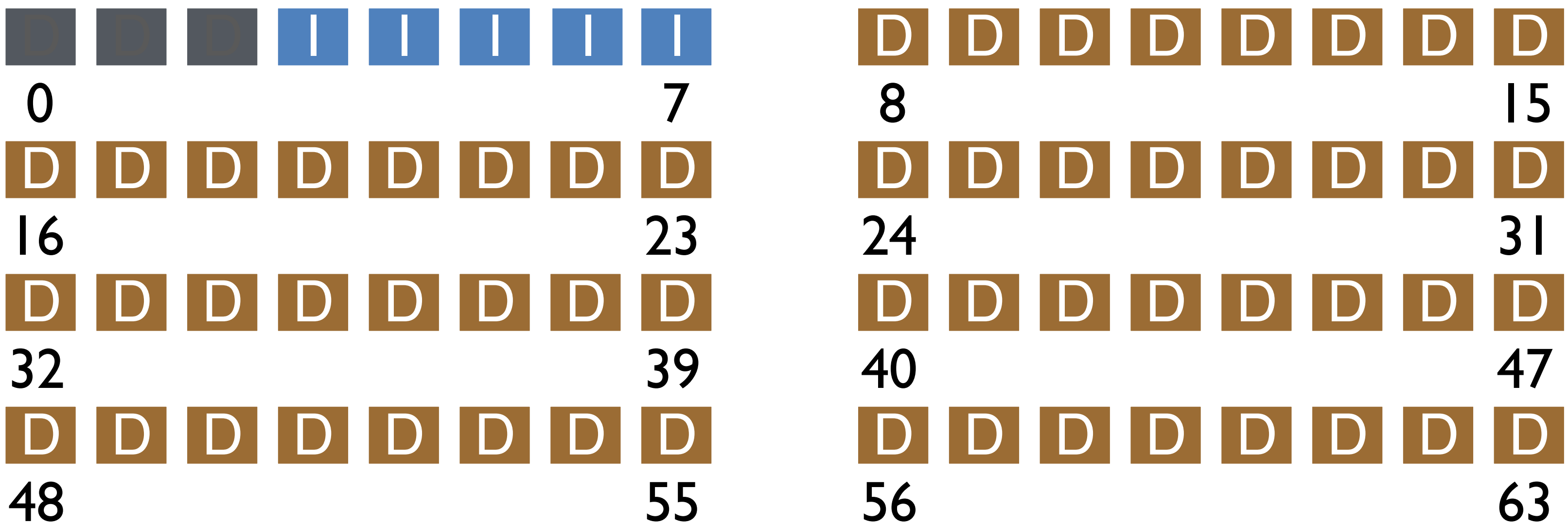
Assume each block is 4KB

FS STRUCTS: DATA BLOCKS



Not actual layout : Examine better layout in next lecture
Relative number of blocks is important

INODES



I I I I I
0 7
D D D D D D D D
8 15
D D D D D D D D
16 23
D D D D D D D D
24 31
D D D D D D D D
32 39
D D D D D D D D
40 47
D D D D D D D D
48 55
D D D D D D D D
56 63

ONE INODE BLOCK

Each inode is typically 256 bytes
(depends on the FS, maybe 128 bytes)

- 4KB disk block
- 16 inodes per inode block

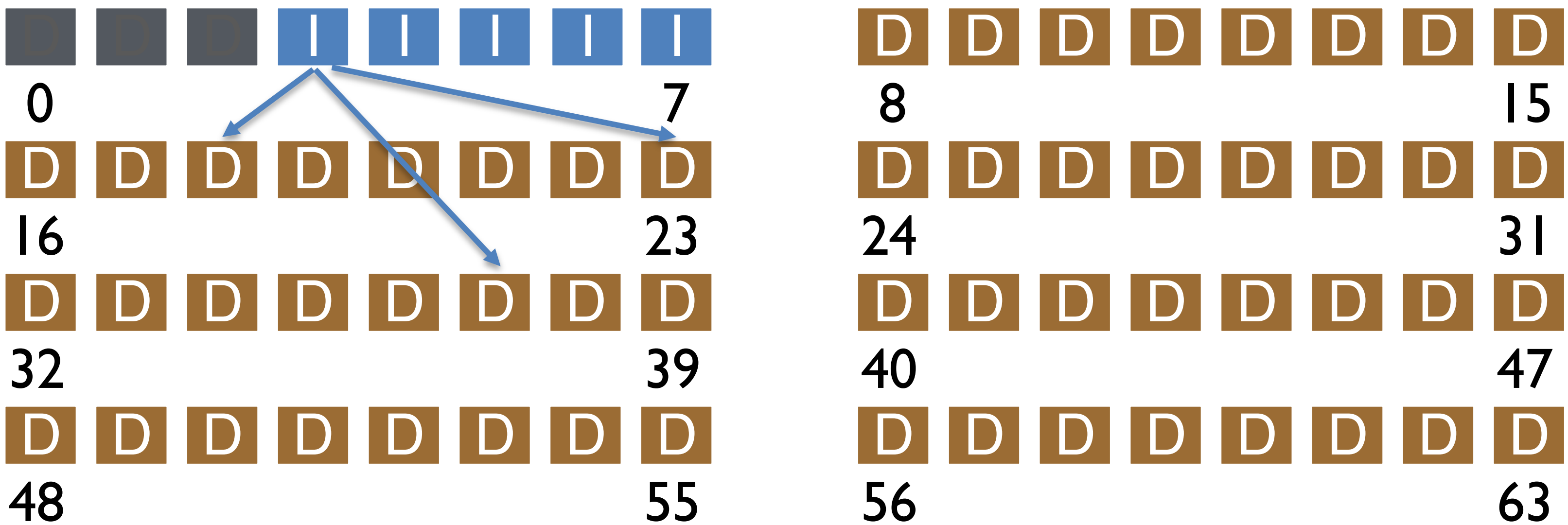
How to modify 1 inode?

Static calculation to determine where particular inode resides on disk

INODE

type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
Num Blocks
time (access)
ctime (create)
mtime (modify)
links_count (# paths)
addrs[N] (N data blocks)

FS STRUCTS: INODE DATA POINTERS



INODE

type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
Blocks
time (access)
ctime (create)
links_count (# paths)
addrs[N] (N data blocks)

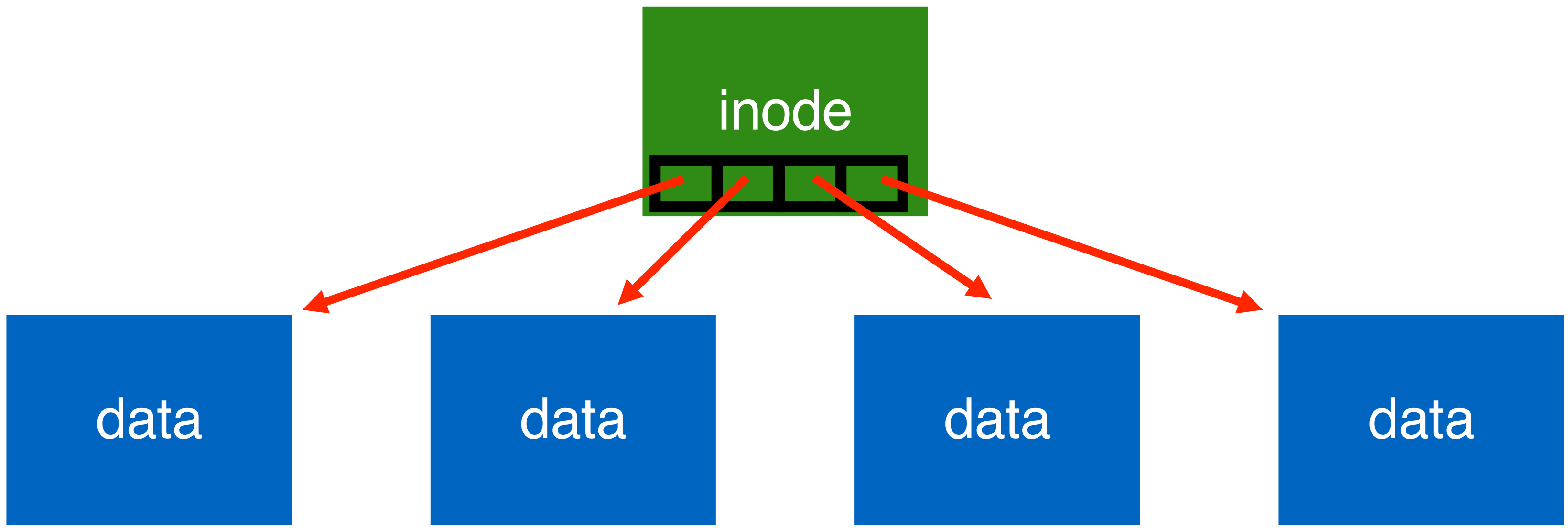
Assume single level
(just pointers to data blocks)

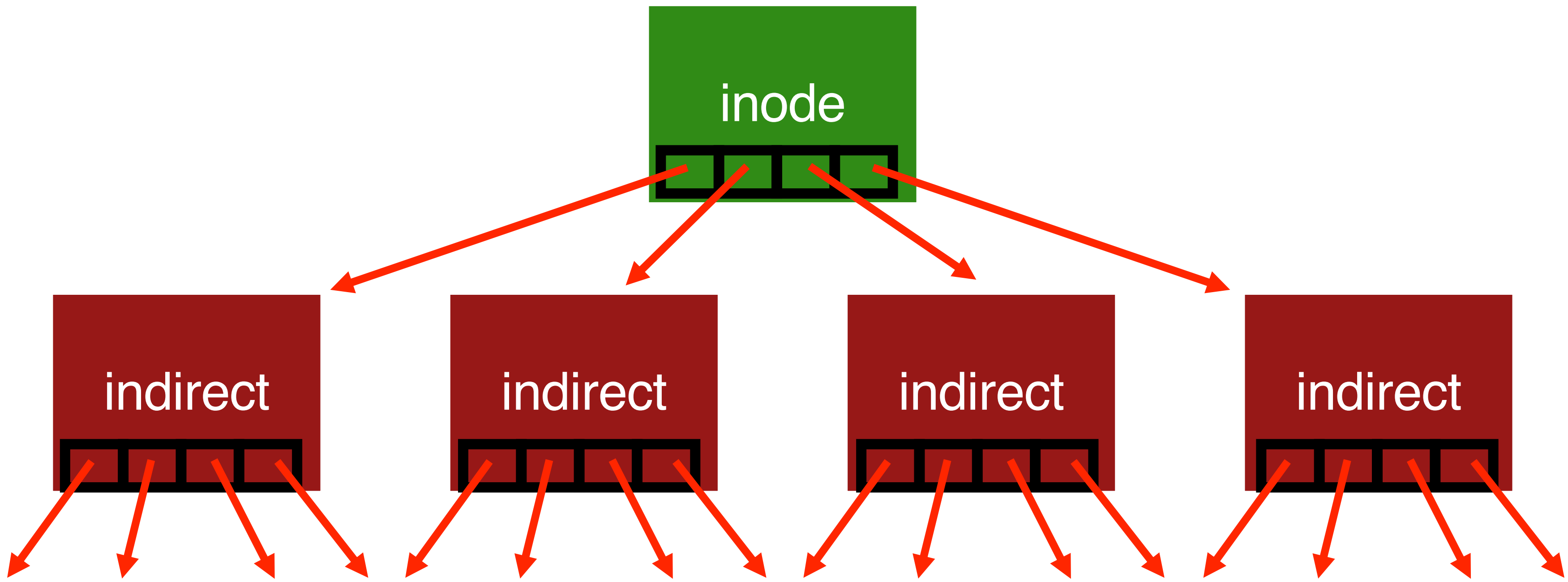
What is max file size?

- Assume 256-byte inodes
- (assume all can be used for pointers)
- Assume 4-byte addrs

256 / 4 = 64 pointers
64 * 4K = 256 KB!

How to get larger files?


inode
data
data
data
data



Indirect blocks are stored in regular data blocks

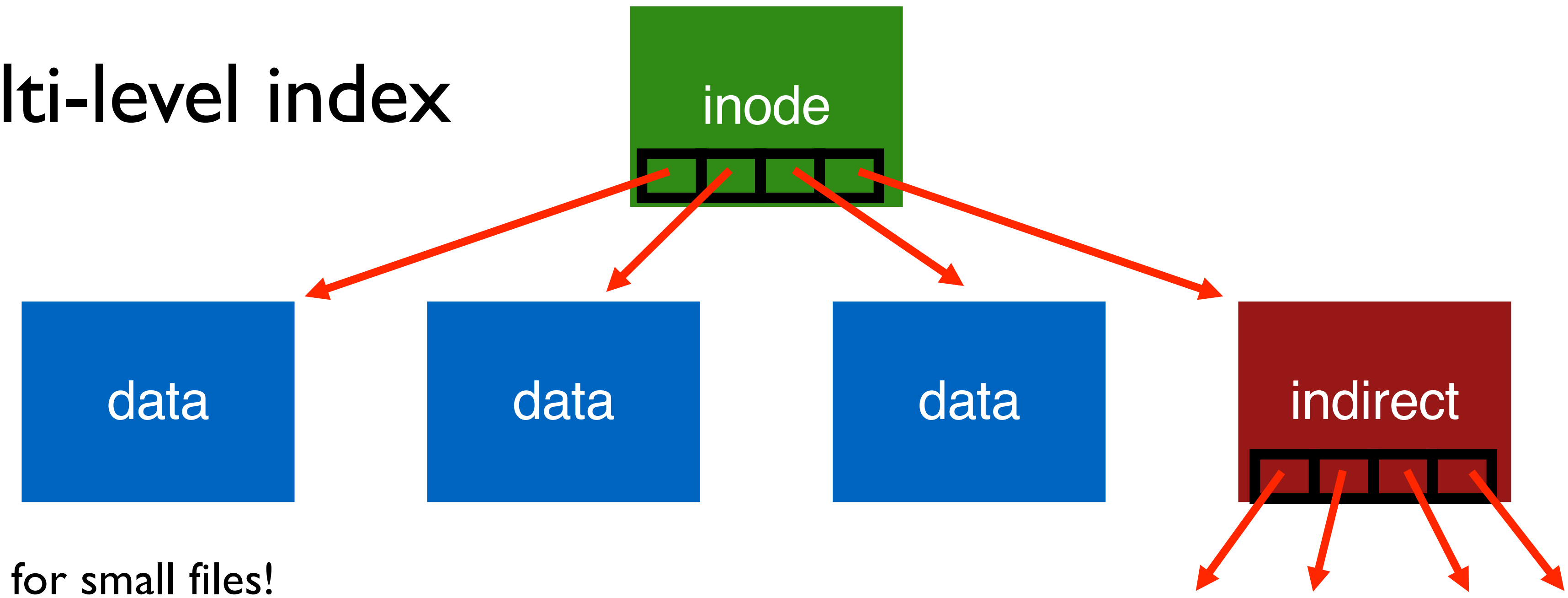
Largest file size with 64 indirect blocks?

64 * 1024 * 4KB = 64 MB

Any Cons?

what if we want to optimize for small files?

Multi-level index



Better for small files!
How to handle even larger files?

Double indirect blocks
Triple indirect blocks

Example:
12 direct pointers + single + double indirect block.
Block size of 4 KB and 4-byte pointers

$(12 + 1024 + 1024*1024) \times 4\ \text{KB} = 4\ \text{GB}$

DIRECTORIES

File systems vary

Common design:

- Store directory entries in data blocks
- Large directories just use multiple data blocks
- Use bit in inode to distinguish directories from files

Various formats could be used

- lists
- b-trees

SIMPLE DIRECTORY LIST EXAMPLE

valid	name	inode
1	.	134
1	..	35
1	foo	80
1	bar	23

unlink("foo")

Remove entry from directory
What do we do to the inode?

ALLOCATION

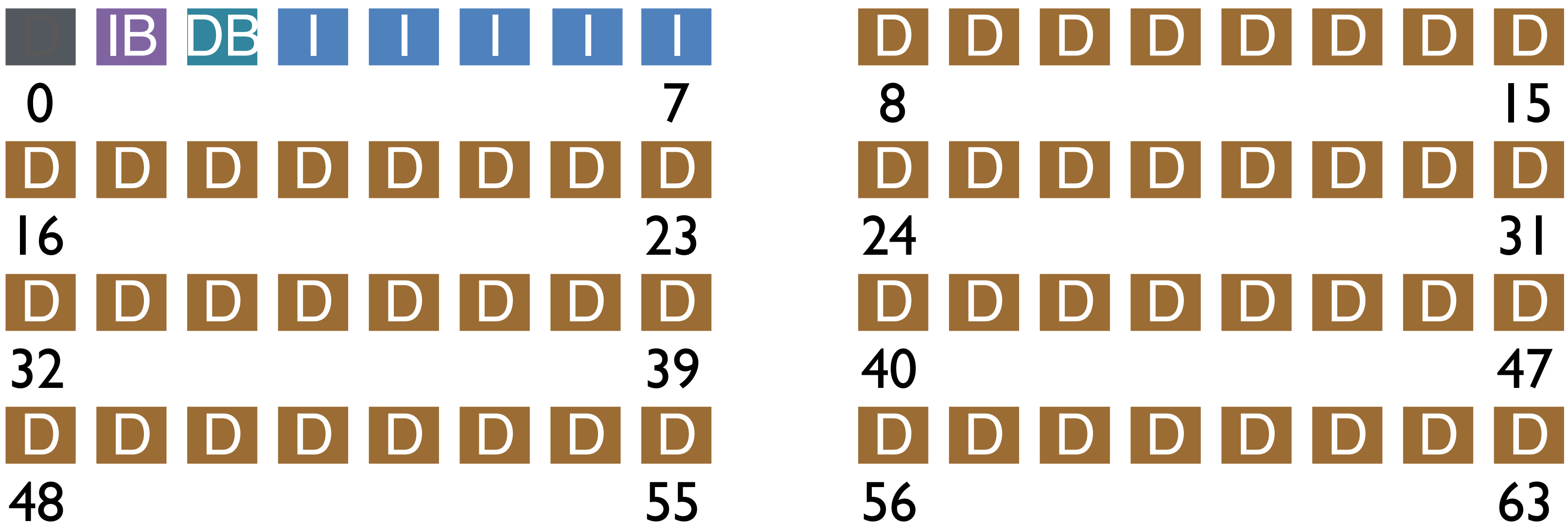
How do we find free data blocks or free inodes?

Free list

Bitmaps (one bit designates state of each block; set to 1 if allocated, 0 if free)

Tradeoffs in next lecture…

FS STRUCTS: BITMAPS



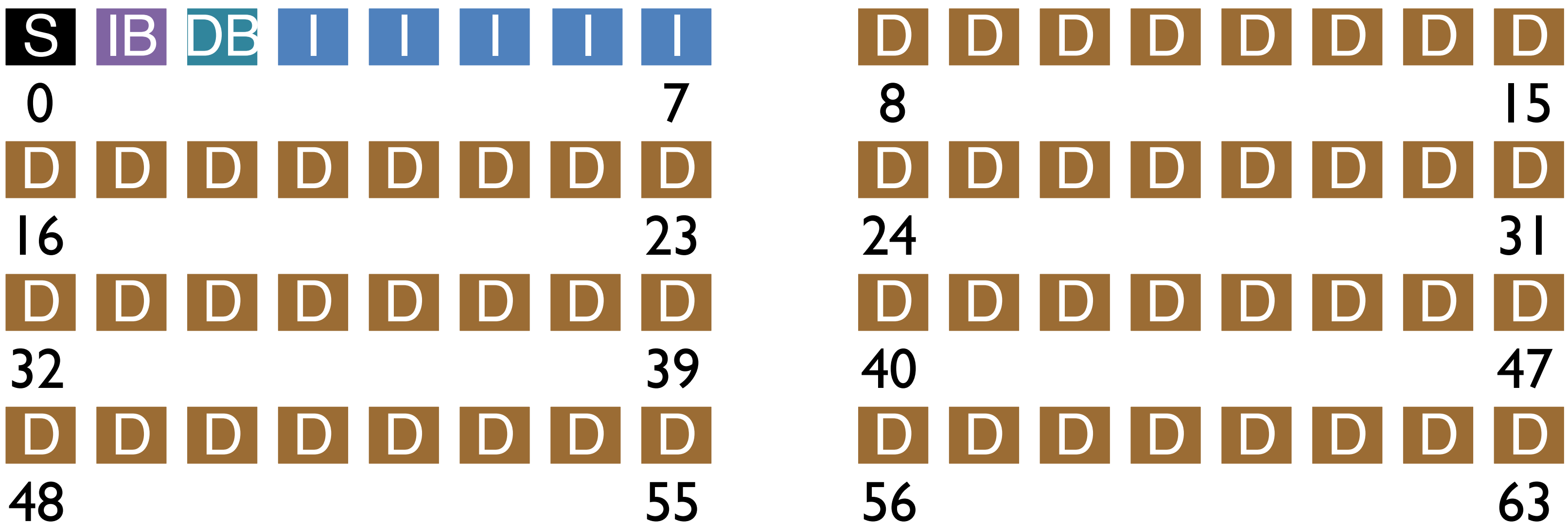
SUPERBLOCK

Need to know basic FS configuration metadata, like:

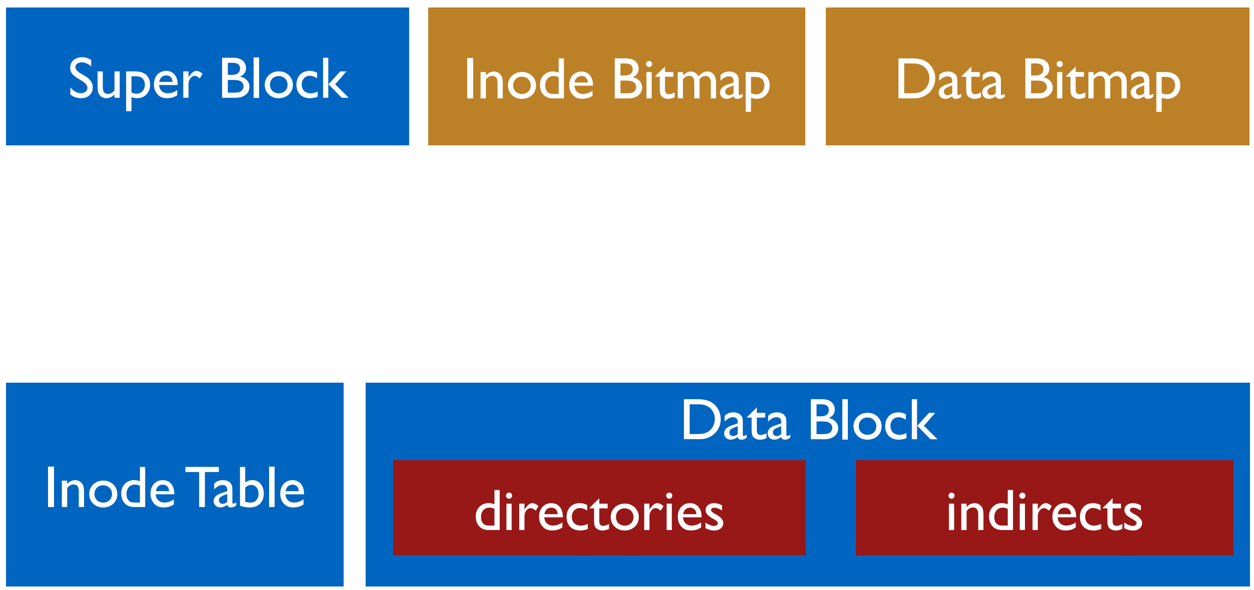
- block size
- # of inodes

Store this in superblock

FS STRUCTS: SUPERBLOCK



SUMMARY


Super Block
Inode Bitmap
Data Bitmap
Inode Table
Data Block
directories
indirects

PART 2 : OPERATIONS

- create file
- write
- open
- read
- close

create /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data

What needs to be read and written?

open /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

write to /foo/bar (assume file exists and has been opened)

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

read /foo/bar – assume opened

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

close /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

EFFICIENCY

How can we avoid this excessive I/O for basic ops?

Cache for:

- reads
- write buffering

WRITE BUFFERING

Why does procrastination help?

Overwrites, deletes, scheduling

Shared structs (e.g., bitmaps+dirs) often overwritten.

We decide: how much to buffer, how long to buffer…

- tradeoffs?

NEXT STEPS

Next class: UNIX Fast-File System