File System Implementation

Questions answered in this lecture:

What on-disk structures to represent files and directories?

- Contiguous, Extents, Linked, FAT, Indexed, Multi-level indexed
- Which are good for different metrics?

What disk operations are needed for:

- make directory
- open file
- write/read file
- close file

Review: File Names

- **Different types of names work better in different contexts**

- **inode**
  - unique name for file system to use
  - records meta-data about file: file size, permissions, etc

- **path**
  - easy for people to remember
  - organizes files in hierarchical manner; encode locality information

- **file descriptor**
  - avoid frequent traversal of paths
  - remember multiple offsets for next read or write

Review: File API

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)

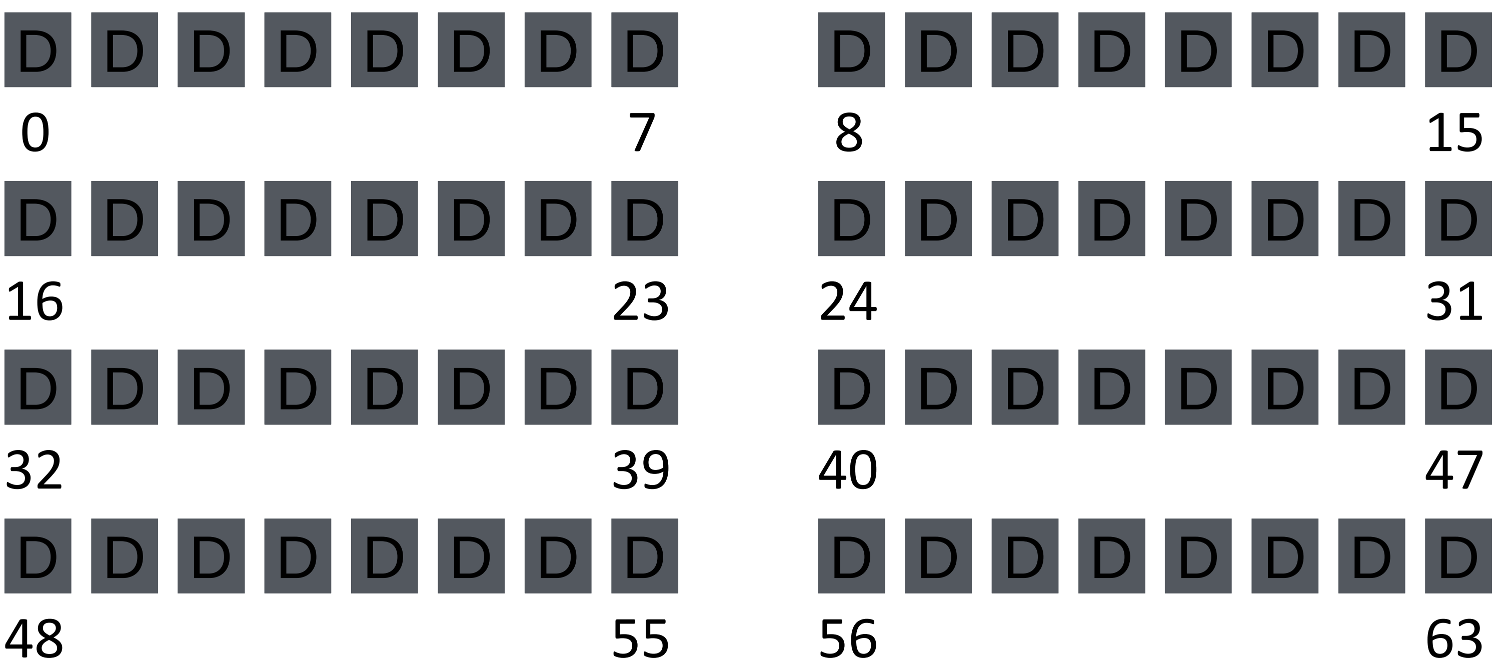
Implementation

- **1. On-disk structures**
  - how does file system represent files, directories?
- **2. Access methods**
  - what steps must reads/writes take?

Part 1: Disk Structures

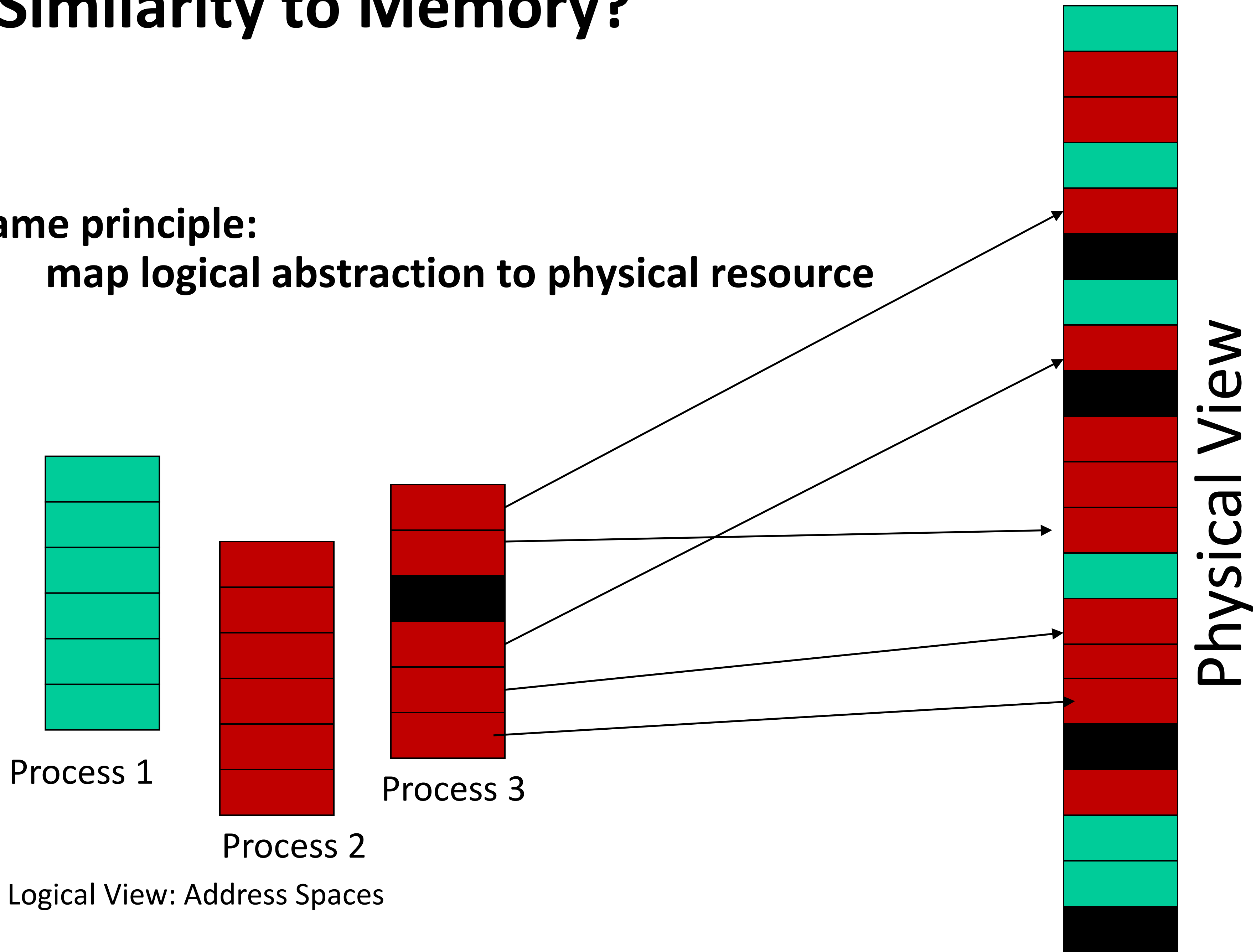
Persistent Store

- **Given: large array of blocks on disk**
- **Want: some structure to map files to disk blocks**



Similarity to Memory?

Same principle:
map logical abstraction to physical resource

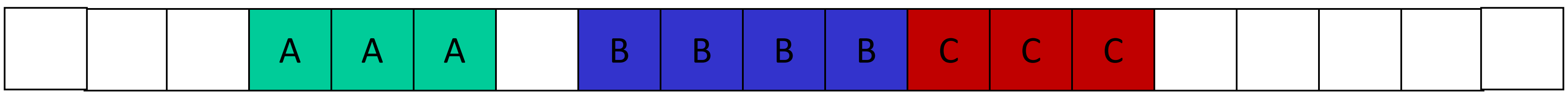


Allocation Strategies

- **Many different approaches**
  - Contiguous
  - Extent-based
  - Linked
  - File-allocation Tables
  - Indexed
  - Multi-level Indexed
- **Questions**
  - Amount of fragmentation (internal and external)
    – freespace that can't be used
  - 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 fragmentation (needs periodic compaction)

Ability to grow file over time? - May not be able to without moving

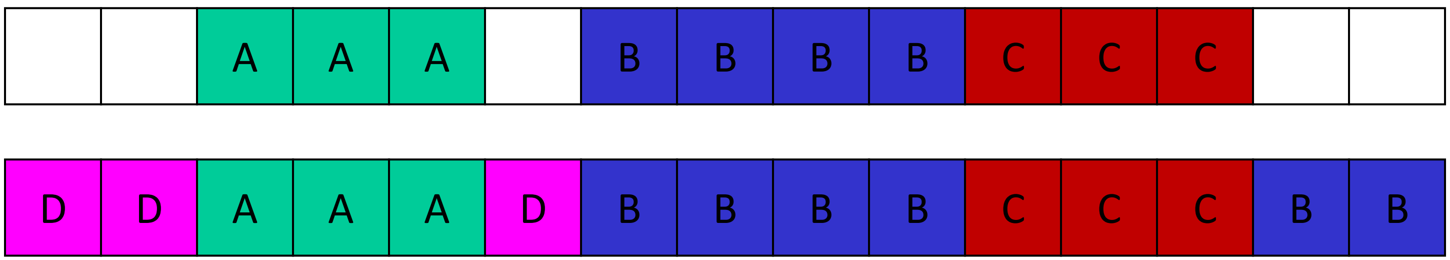
Seek cost for sequential accesses? + Excellent performance

Speed to calculate random accesses? + Simple calculation

Wasted space for meta-data? + Little overhead for meta-data

Small Fixed Number 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

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

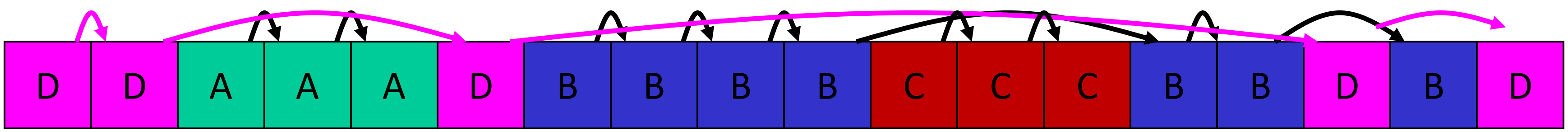
Seek cost for sequential accesses? + Still good performance

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: pointer and space in last block

Ability to grow file over time? + Can grow easily

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

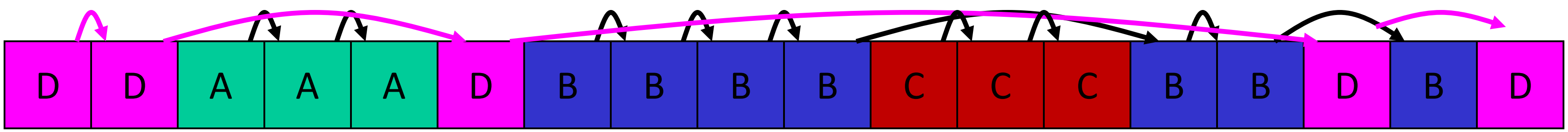
Speed to calculate random accesses? - Ridiculously poor

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

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

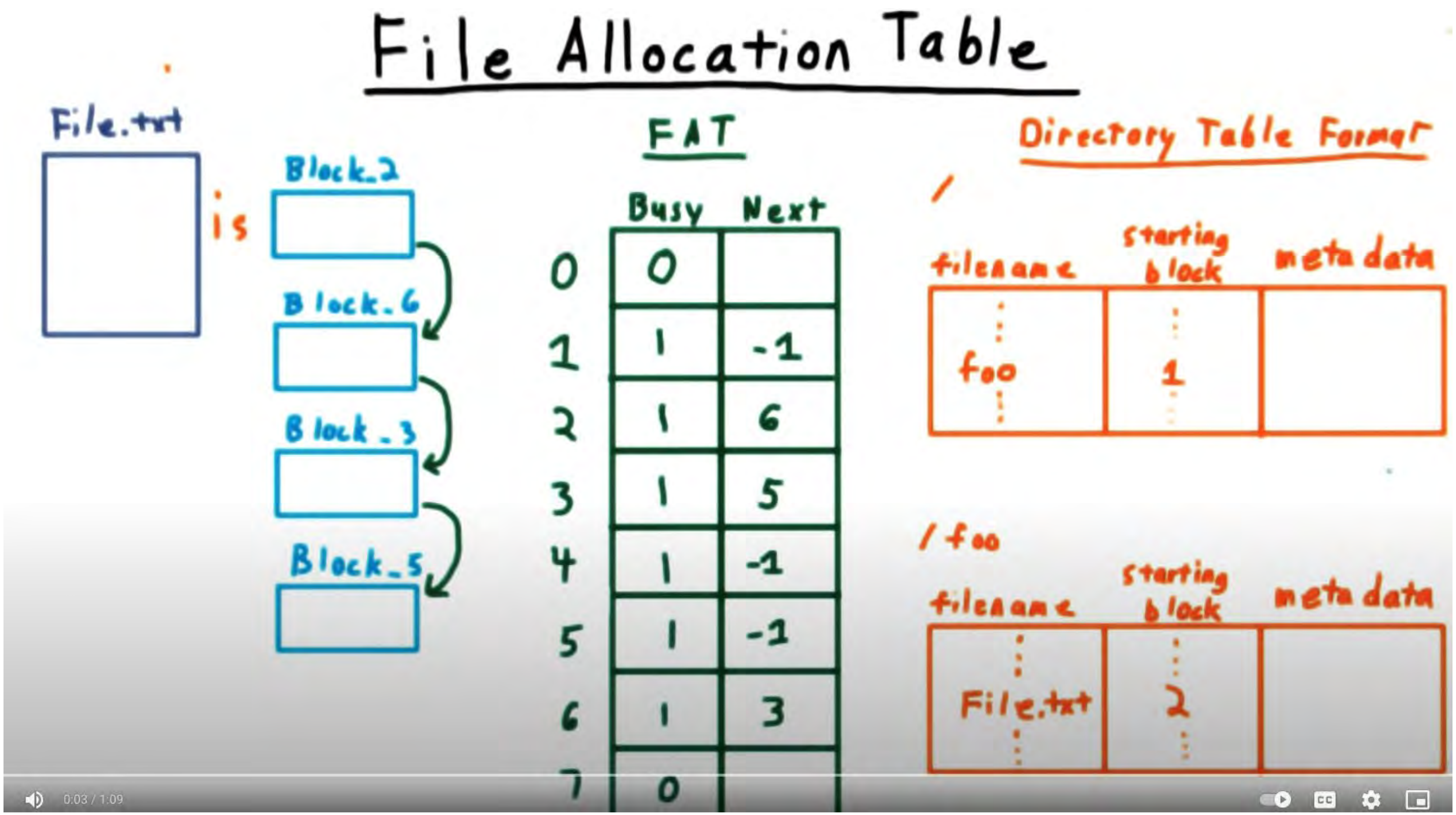
File-Allocation Table (FAT)

- **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?

File-Allocation Table (FAT)



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

D	D	A	A	A	D	B	B	B	B	C	C	C	B	B	D	B	D

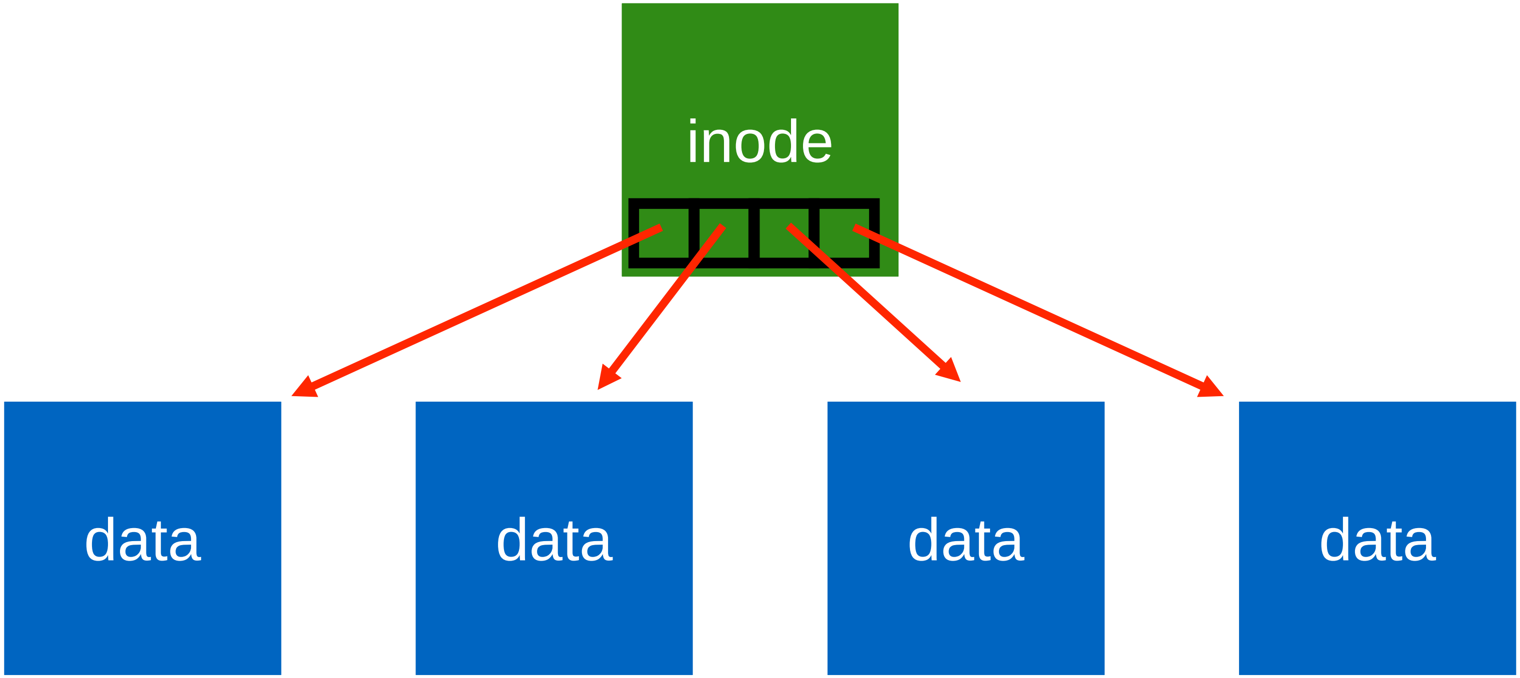
- Advantages
  - No external fragmentation
  - Files can be easily grown up to max file size
  - Supports random access
- Disadvantages
  - Large overhead for meta-data:
    - Wastes space for unneeded pointers (most files are small!)

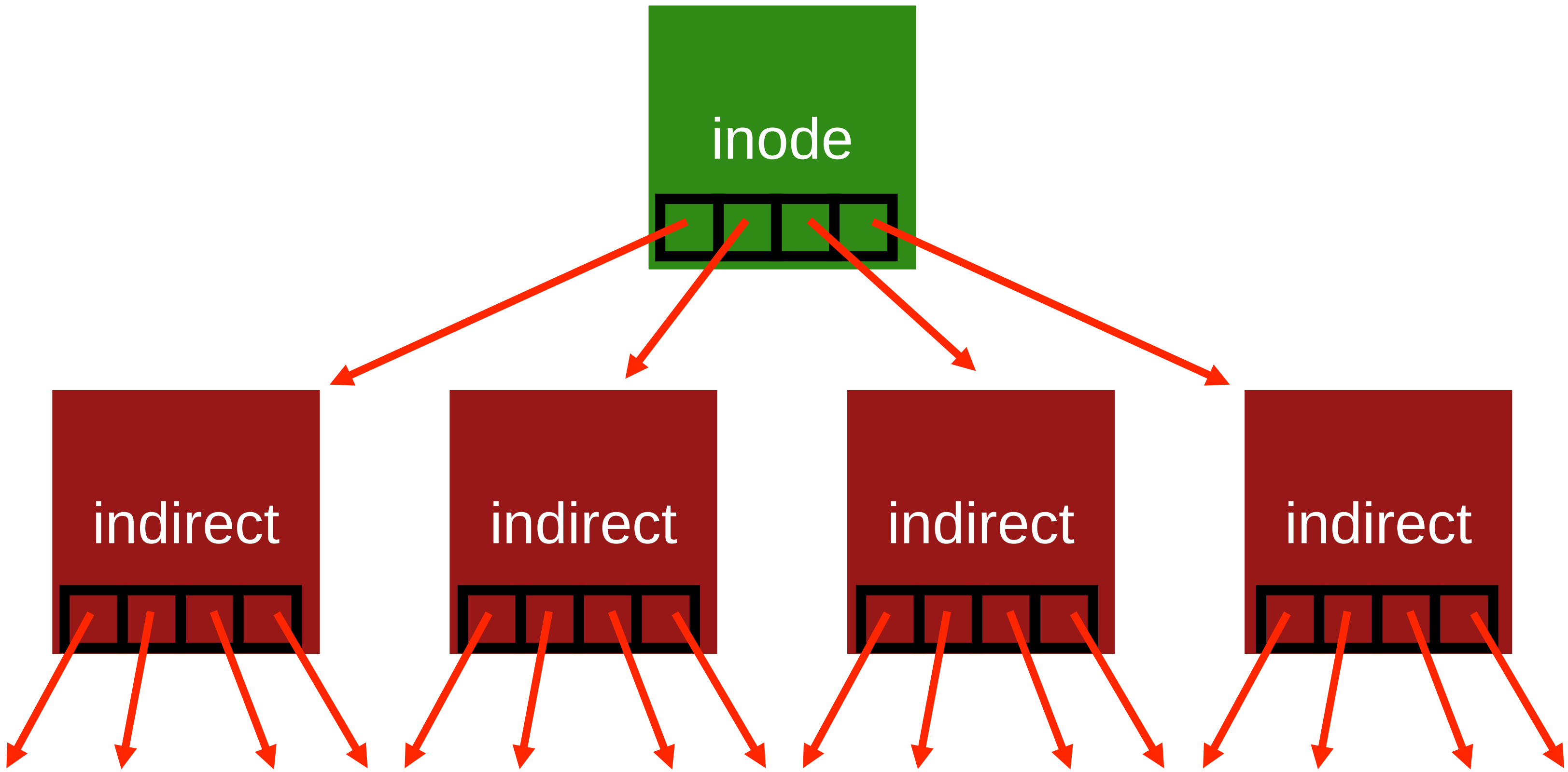
Inode

type
uid
rwx
size
blocks
time
ctime
links_count
addrs[N]

- Assume single level (just pointers to data blocks)
- What is max file size?
  - Assume 256-byte inodes (all can be used for pointers)
  - Assume 4-byte addrs
- How to get larger files?

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


inode
data
data
data
data



Indirect blocks are stored in regular data blocks.

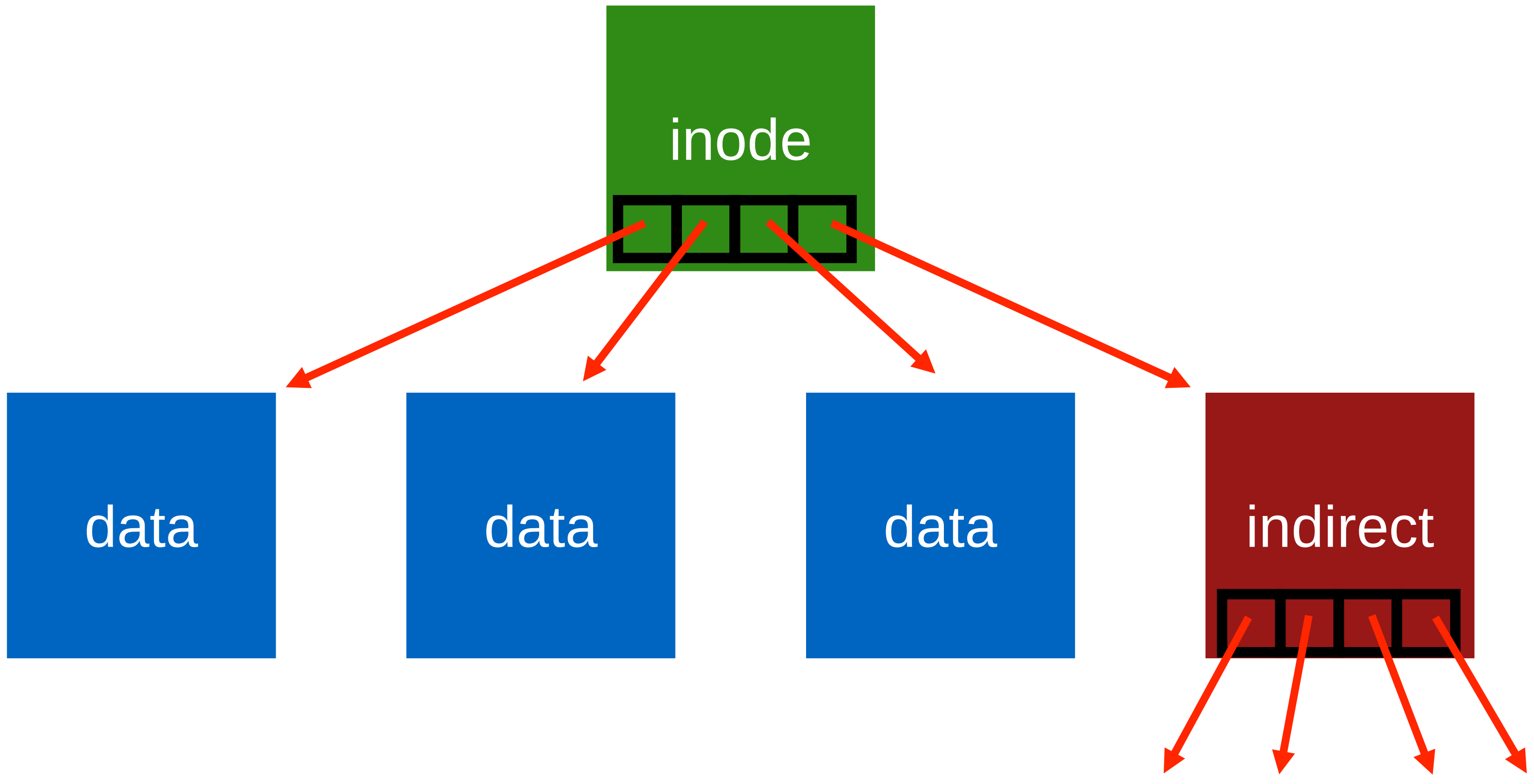
what if we want to optimize for small files?

Multi-Level Indexing

- **Optimize for small files**
  - Most files are small
  - Based on workloads - an important design principle

Most files are small	˜2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ˜50% full
Directories are typically small	Many have few entries; most have 20 or fewer

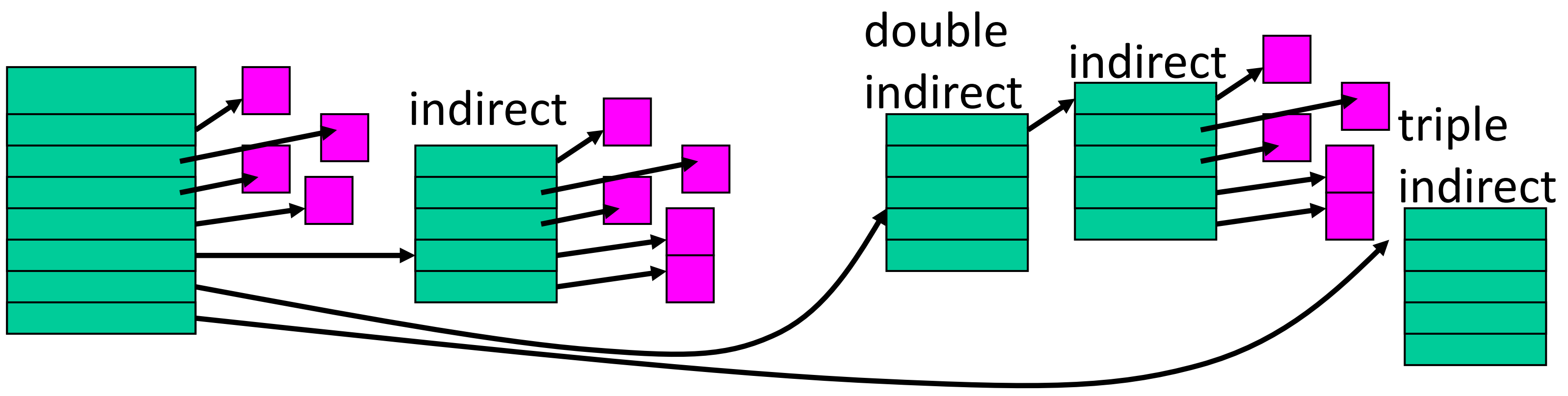
Figure 40.2: **File System Measurement Summary**



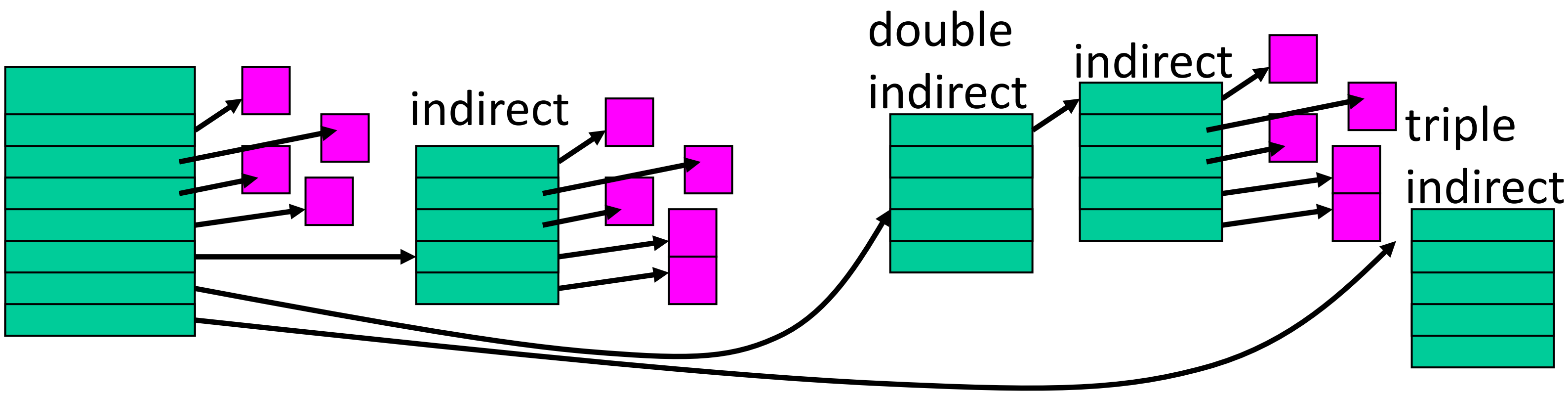
Better for small files

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



Multi-Level Indexing



- Comparison to Indexed Allocation
  - Advantage:
    - Does not waste space for unneeded pointers
    - Still fast access for small files
    - Can grow to what size??
  - Disadvantage:
    - Need to read indirect blocks of pointers to calculate addresses (extra disk read)
    - Keep indirect blocks cached in main memory

Flexible # of Extents

- **Dynamic multiple contiguous regions (extents) per file**
  - Organize extents into multi-level tree structure
    - Each leaf node: starting block and contiguous size
    - 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

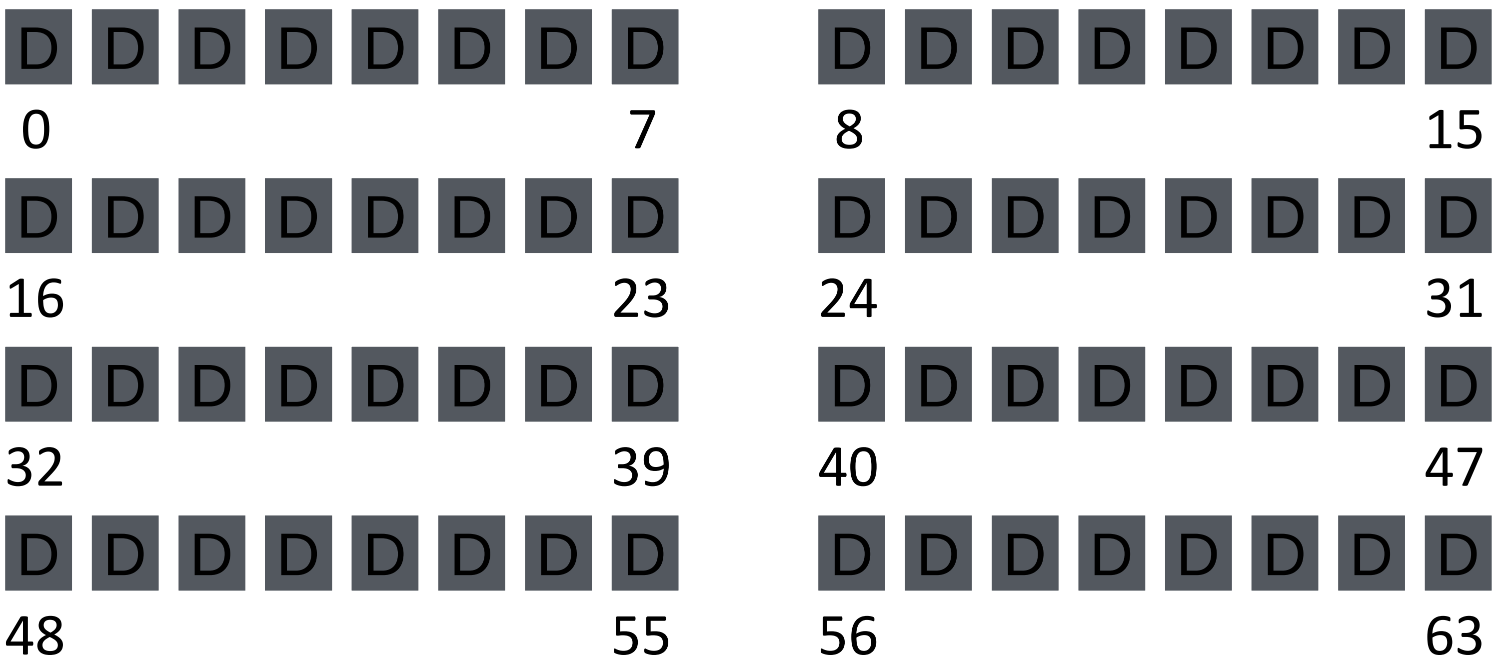
Assume Multi-Level Indexing

- Simple approach
- More complex file systems build from these basic data structures

On-Disk Structures

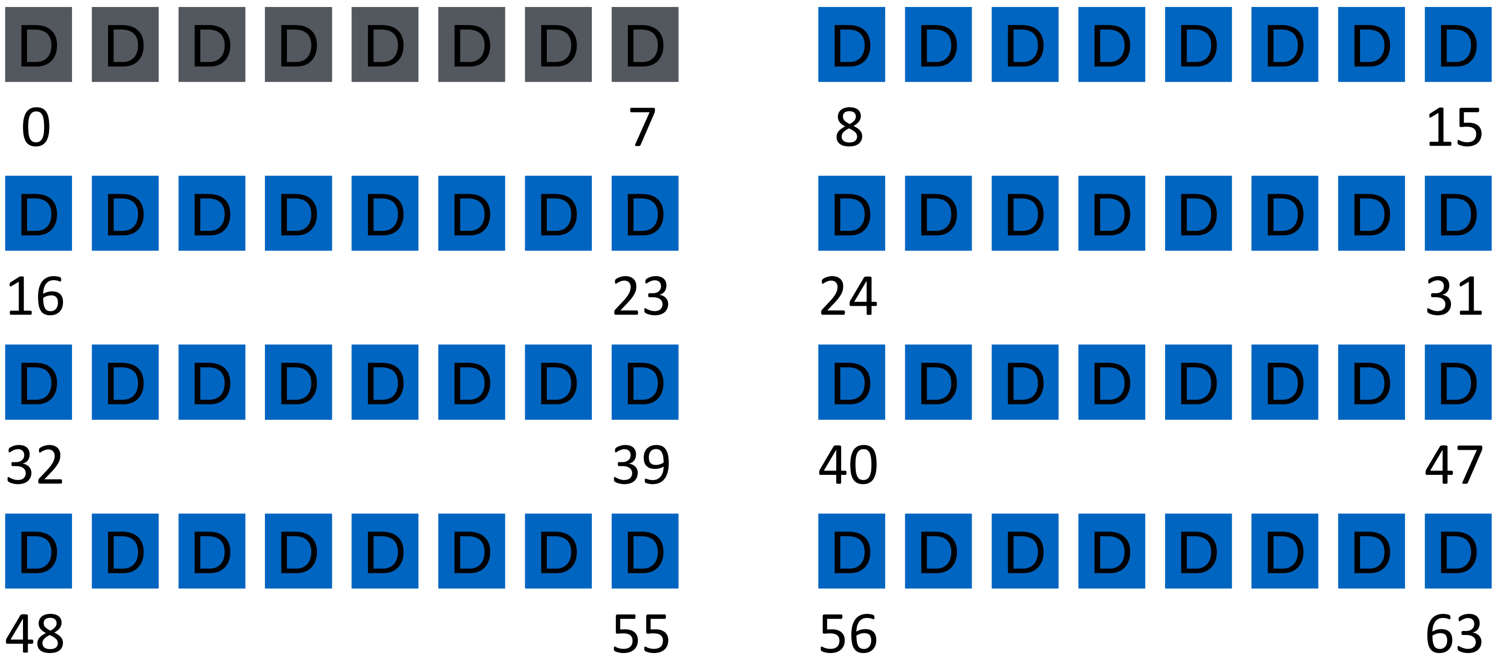
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

FS Structs: Empty Disk



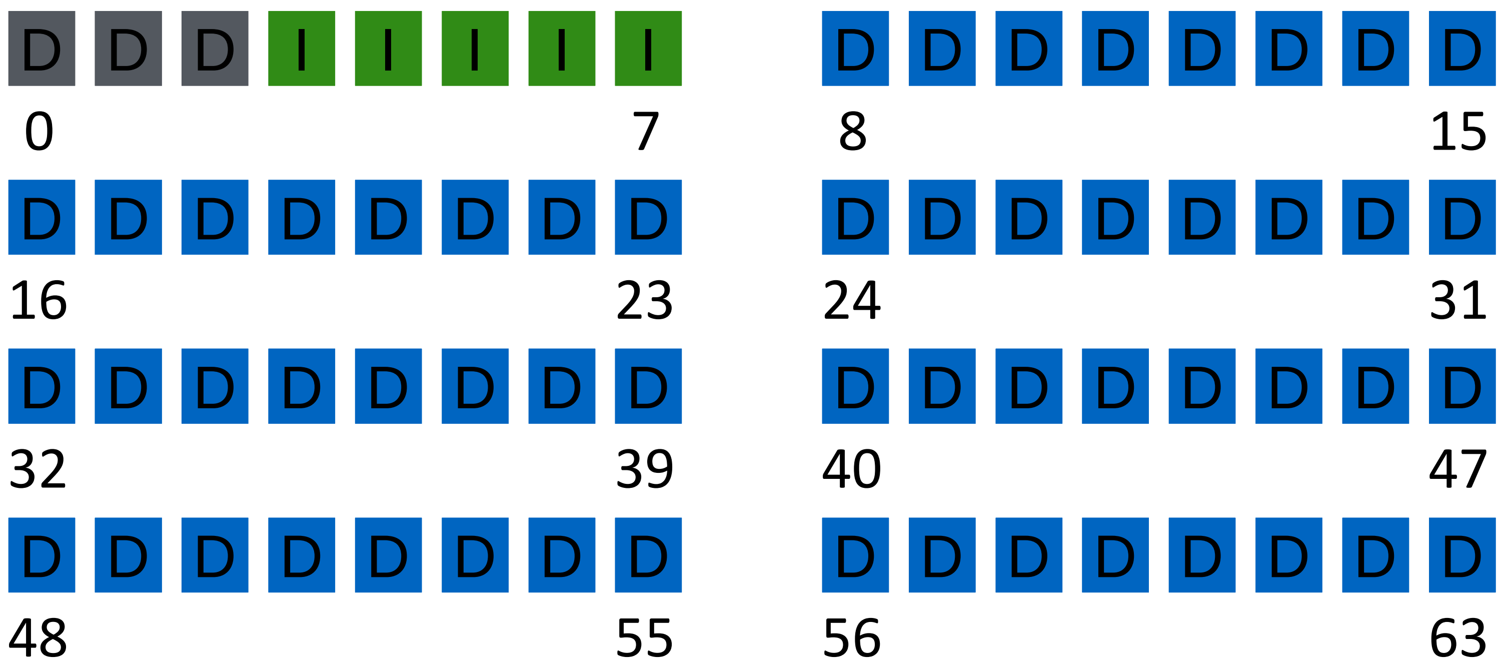
Assume each block is 4KB

Data Blocks



Not actual layout : Examine better layout in next lecture
Purpose: Relative number of each time of block

Inodes



D D D 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

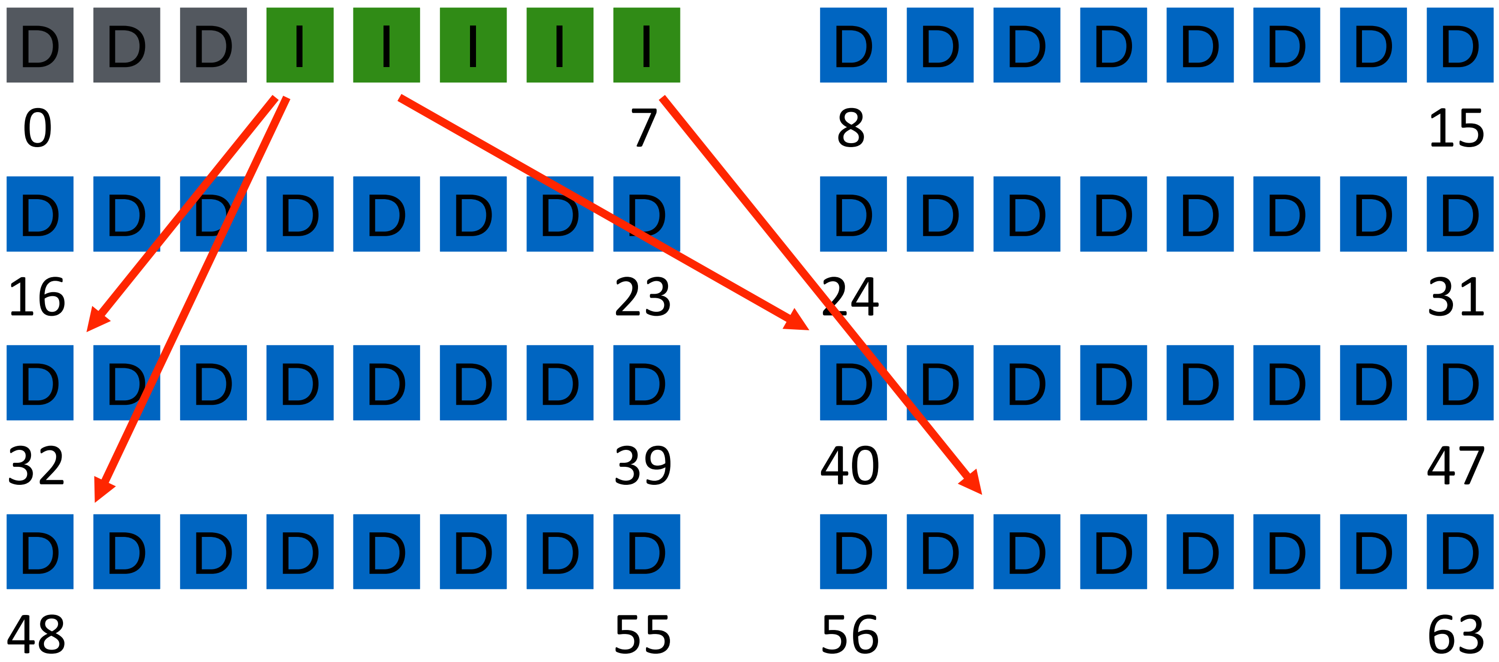
One Inode Block

- Each inode is typically 256 bytes (depends on the FS, maybe 128 bytes)
- 4KB disk block
- 16 inodes per inode block.

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)

Inodes


D D D 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

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

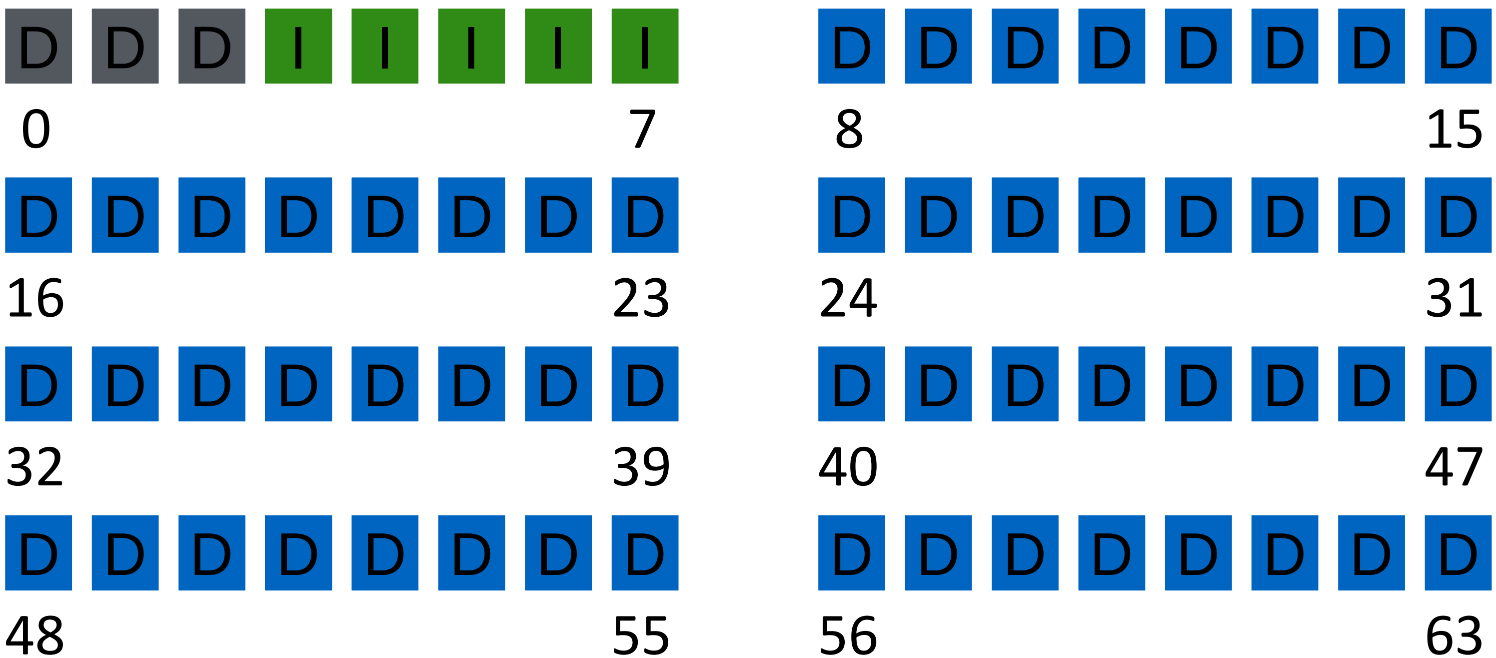
valid	name	inode
1	.	134
1	..	35
1	foo	80
1	bar	23

unlink("foo")

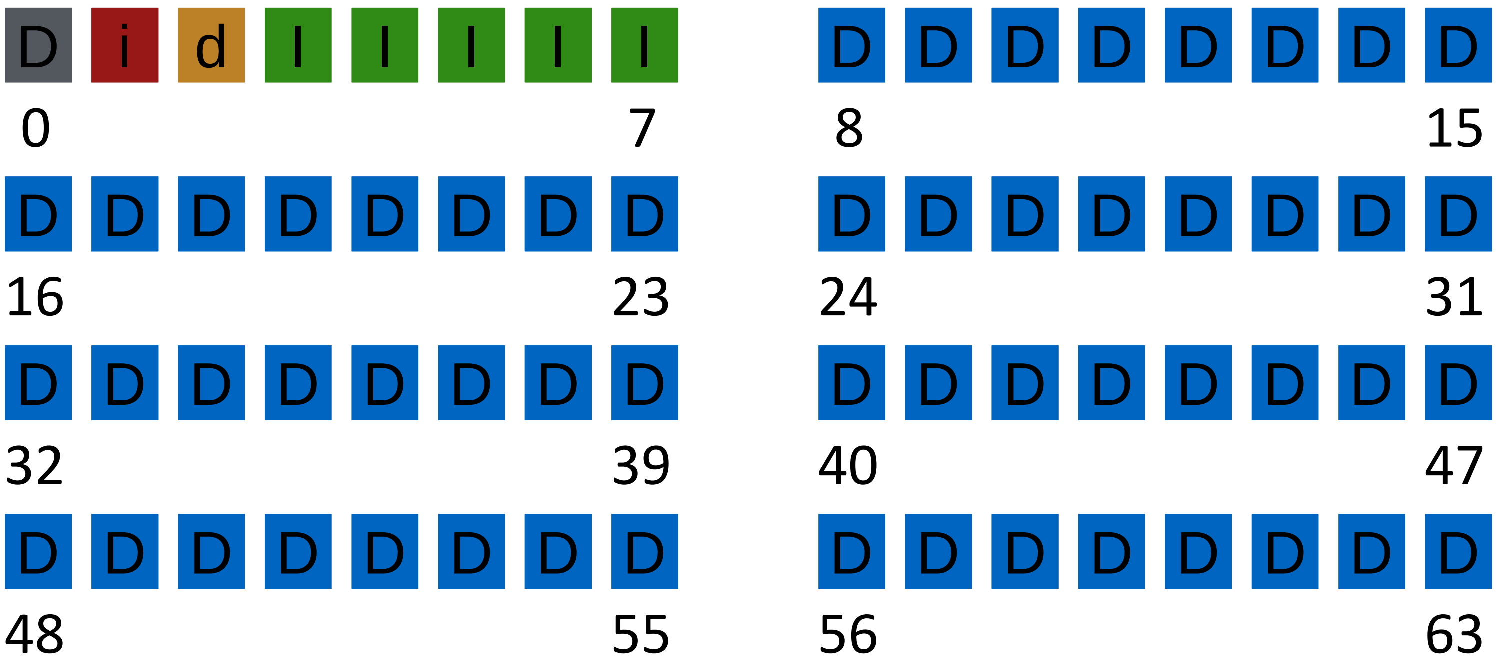
Allocation

- How do we find free data blocks or free inodes?
- Free list
- Bitmaps
- Tradeoffs between data structures

Bitmaps?


D D D 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

Opportunity for Inconsistency (fsck)



i **free bitmap for inodes**

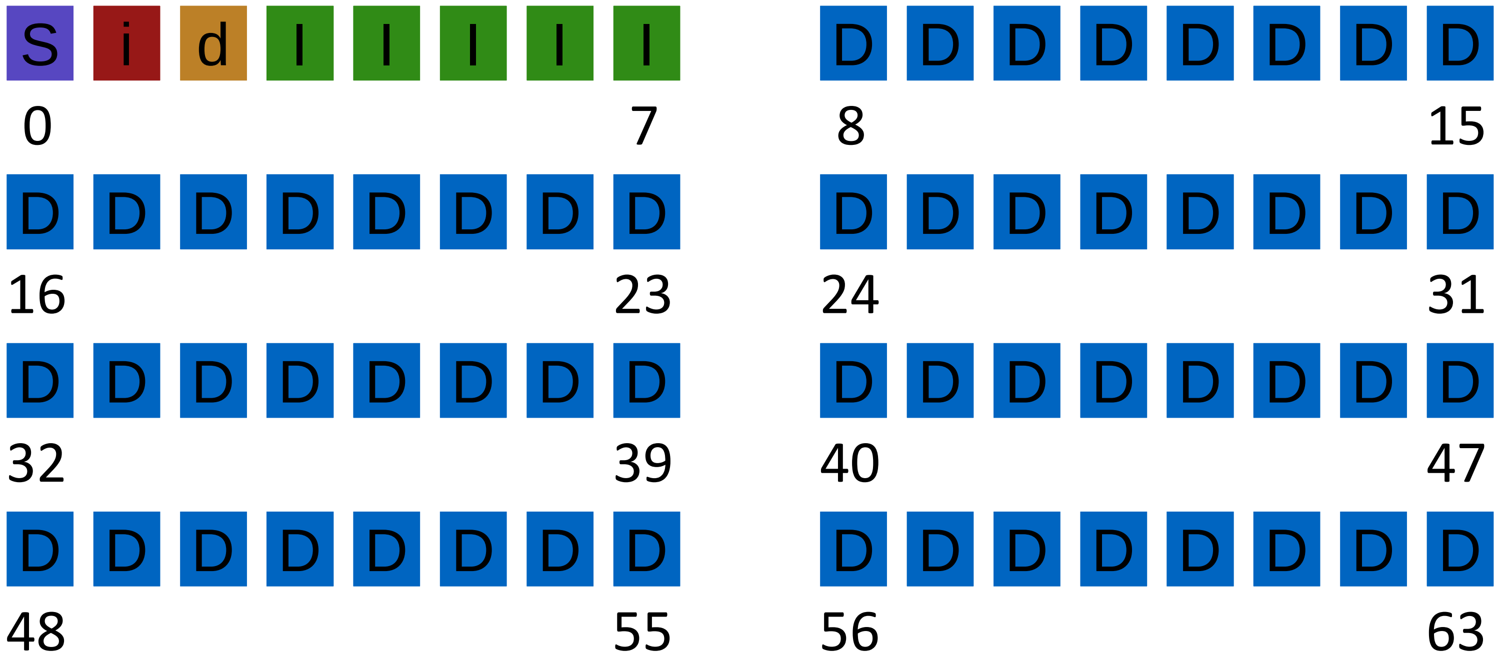
d **free bitmap for data blocks**

Superblock

- **Need to know basic FS configuration metadata, like:**
  - block size
  - # of inodes

- **Store this in superblock**

Super Block

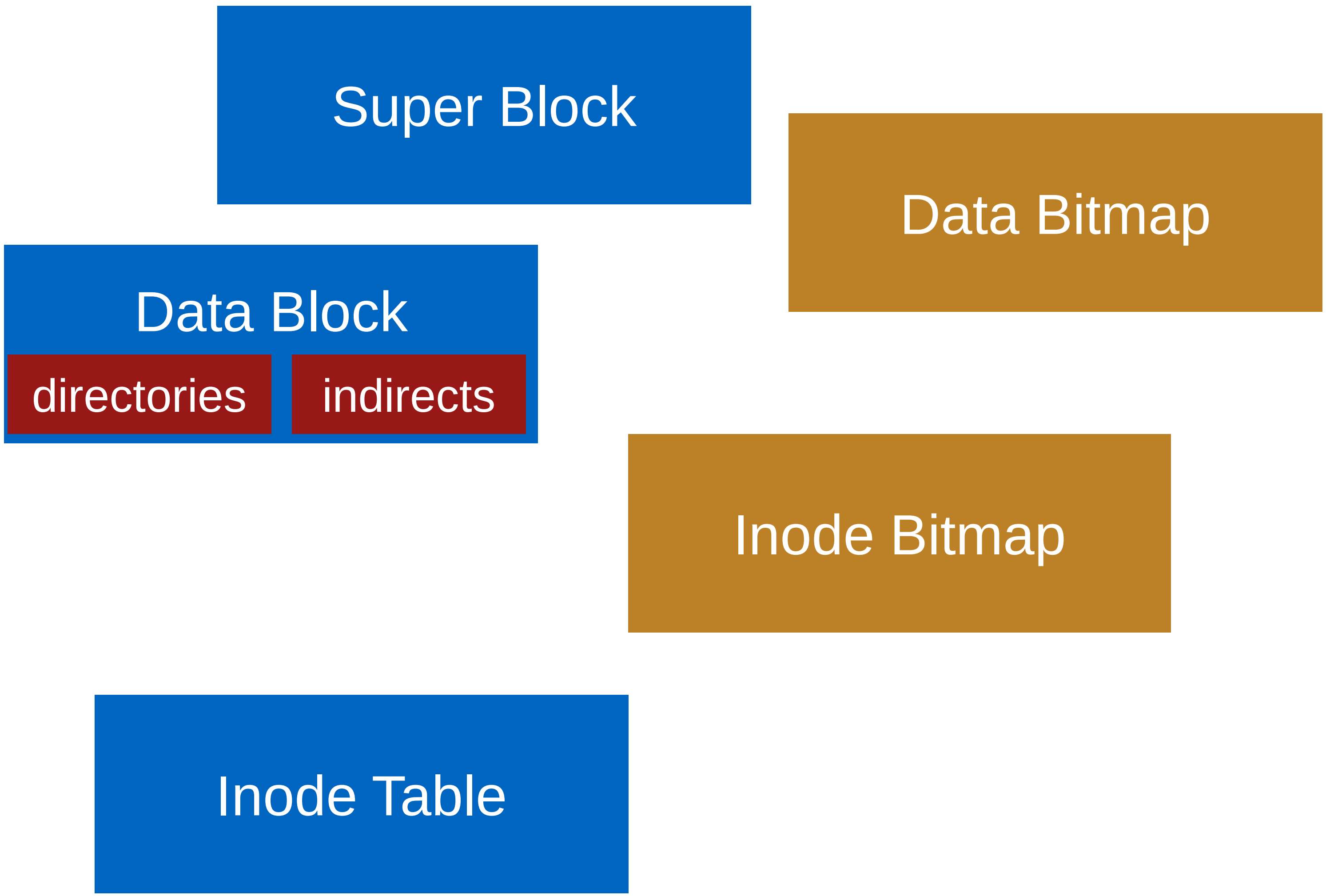


S super block

i free bitmap for inodes

d free bitmap for data blocks

On-Disk Structures



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
		read				
					read	
			read			
						read
	read					
	write					
						write
				read		
				write		
			write			

What needs to be read and written?

Why read bar inode?

When write partial of a block, it needs to be read from disk
When write an entire block, no read is needed.

open /foo/bar

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

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			
read							
write							
							write
				write			

read /foo/bar – assume opened

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
							read
				write			

Why write bar inode?

Update the access time

close /foo/bar

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

nothing to do on disk!

Each process minds its business in its file descriptor table

Optimization

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

- **Cache for:**
  - reads
  - write buffering

- **Virtual memory and disk cache**
  - static partitioning – like 10% of memory for disk cache
  - dynamic partitioning – page cache
  - page cache = virtual memory pages + file system pages

Write Buffering

- **Why does procrastination (拖延) help?**
  - Locality
  - Batching
  - e.g., an inode with create+update can be batched
- **Overwrites, deletes, scheduling**
  - Shared structs (e.g., bitmaps+dirs) often overwritten, reducing the number of writes
  - a temporary file does not need to be write to disk
  - batching leads to OS controlled scheduling
- **We decide: how much to buffer, how long to buffer…**
  - tradeoffs?
  - modern file systems buffer writes between 5-30 seconds

Summary/Future

- **We've described a very simple FS.**
  - basic on-disk structures
  - the basic ops
- **Future questions:**
  - how to allocate efficiently to obtain good performance from disk?
  - how to handle crashes?