OSTEP
Persistence:
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-System API

File Names

Three types of names:

- inode number
- path
- file descriptor

Why?

File Names

inode
- unique name
- remember file size, permissions, etc

path
- easy to remember
- hierarchical

file descriptor
- avoid frequent traversal
- remember multiple offsets

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)
```

Special Calls

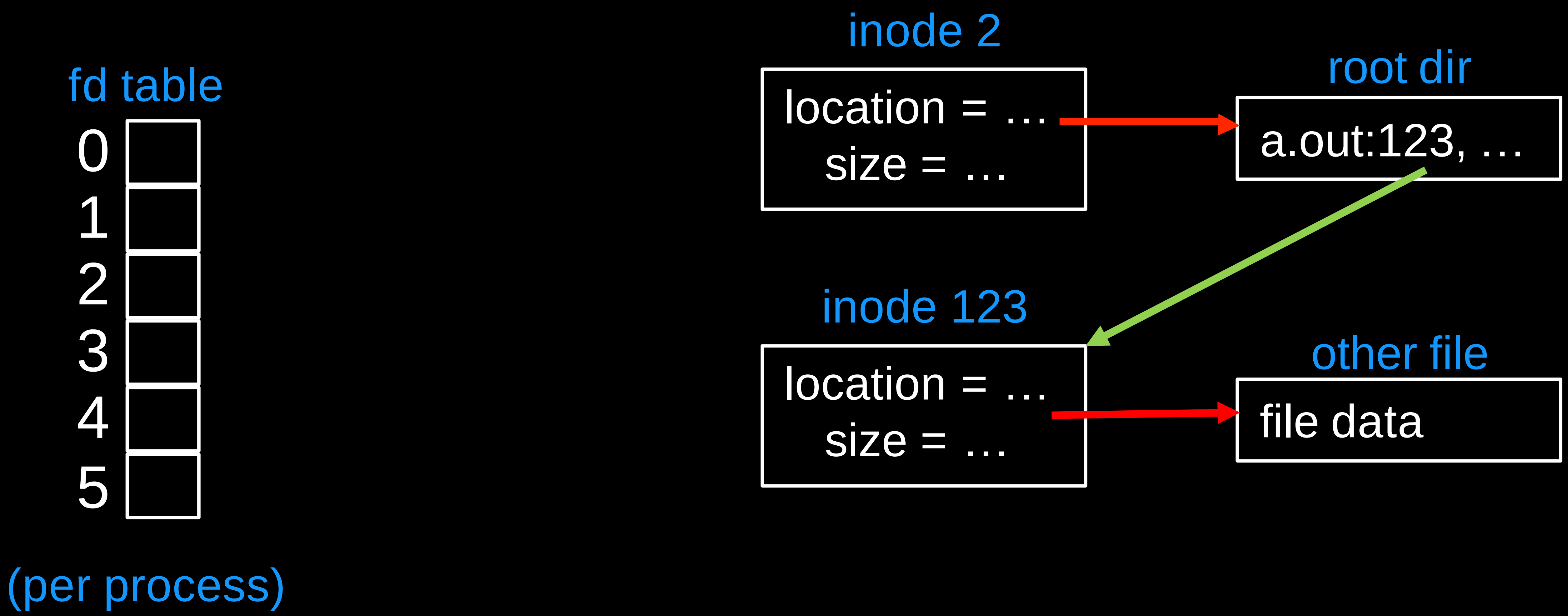
```
fsync(int fd)
rename(char *oldpath, char *newpath)
flock(int fd, int operation)
```

How do you delete a file?

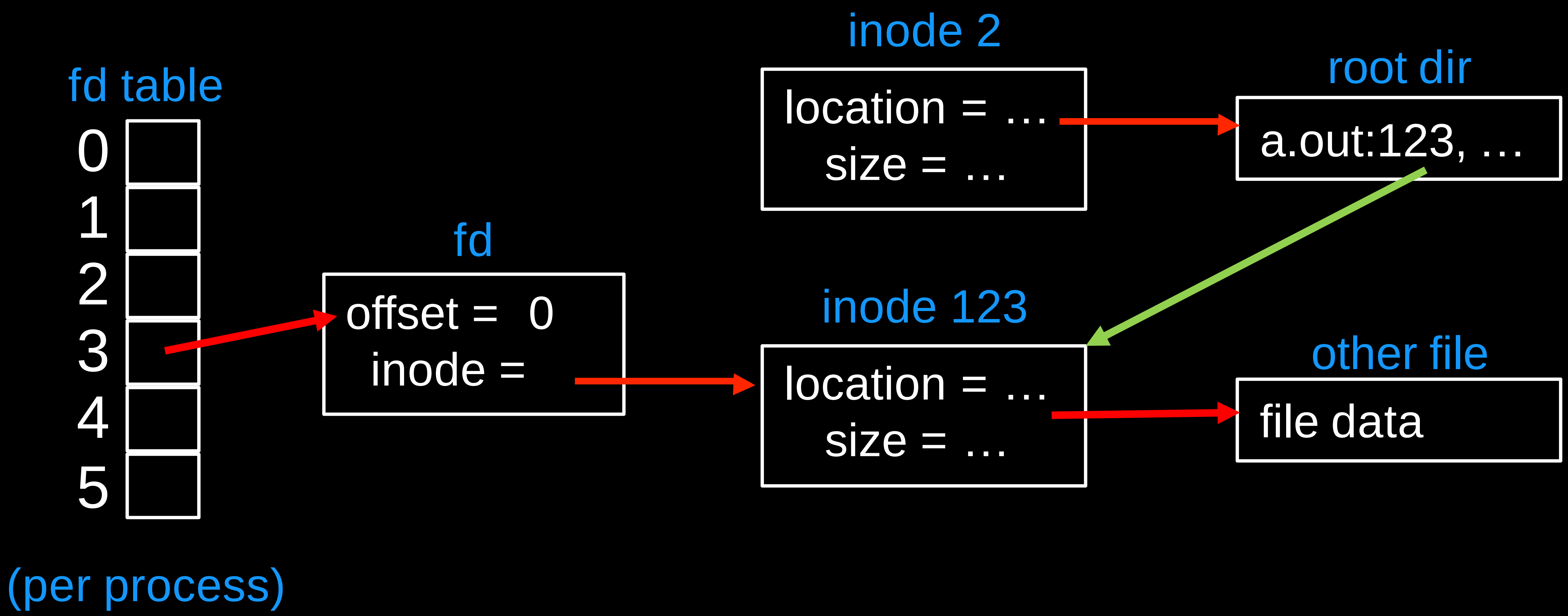
How do you delete a file?

You don’t! It’s garbage collected when there are no more names (fds or paths)

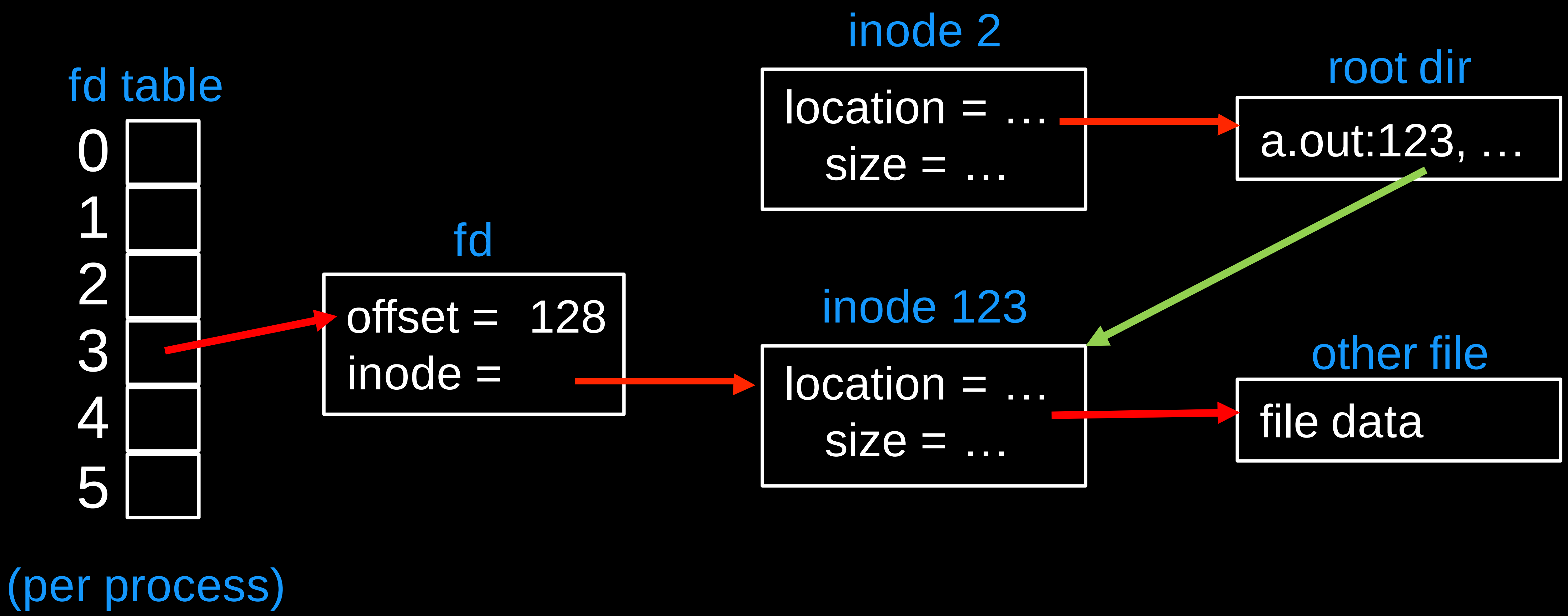
Inodes, Paths, FDs



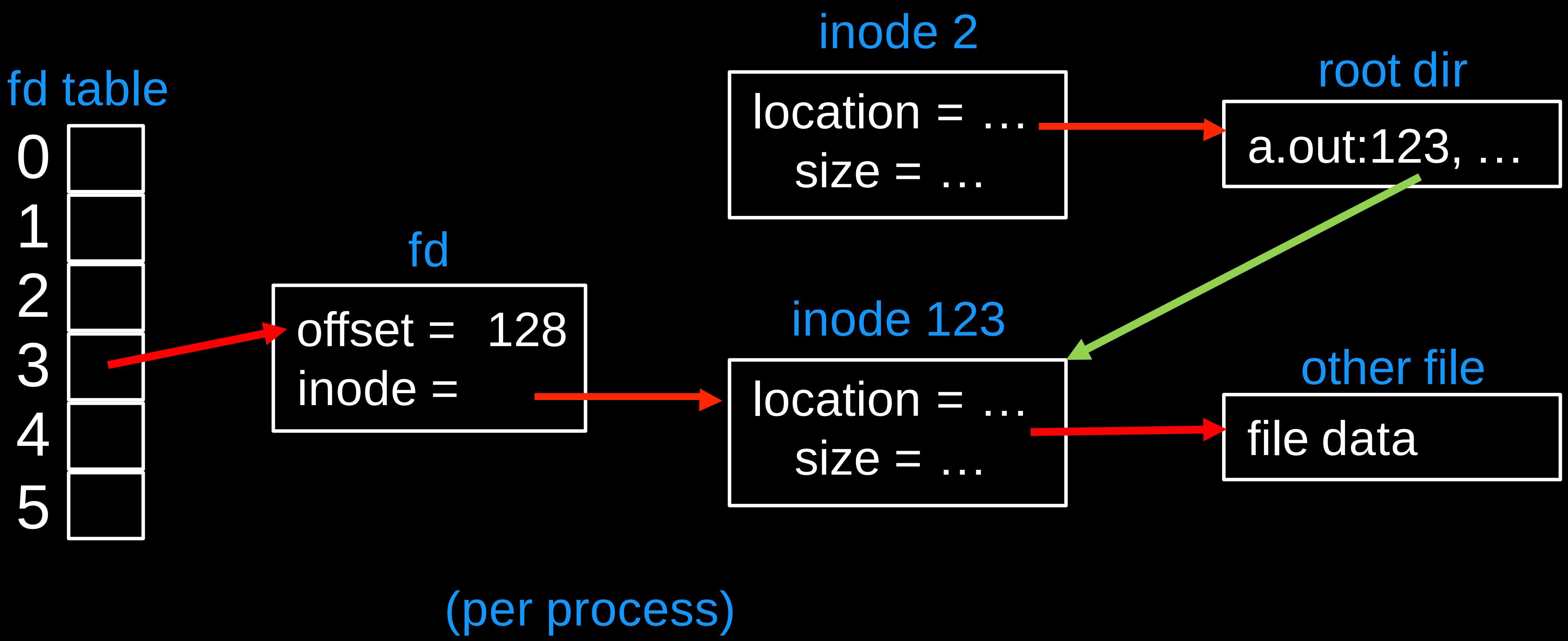
Inodes, Paths, FDs



Inodes, Paths, FDs



Inodes, Paths, FDs



opened /a.out, read 128 bytes

Today: Implementation

1. On-disk structures
   - how does file system represent files, directories?

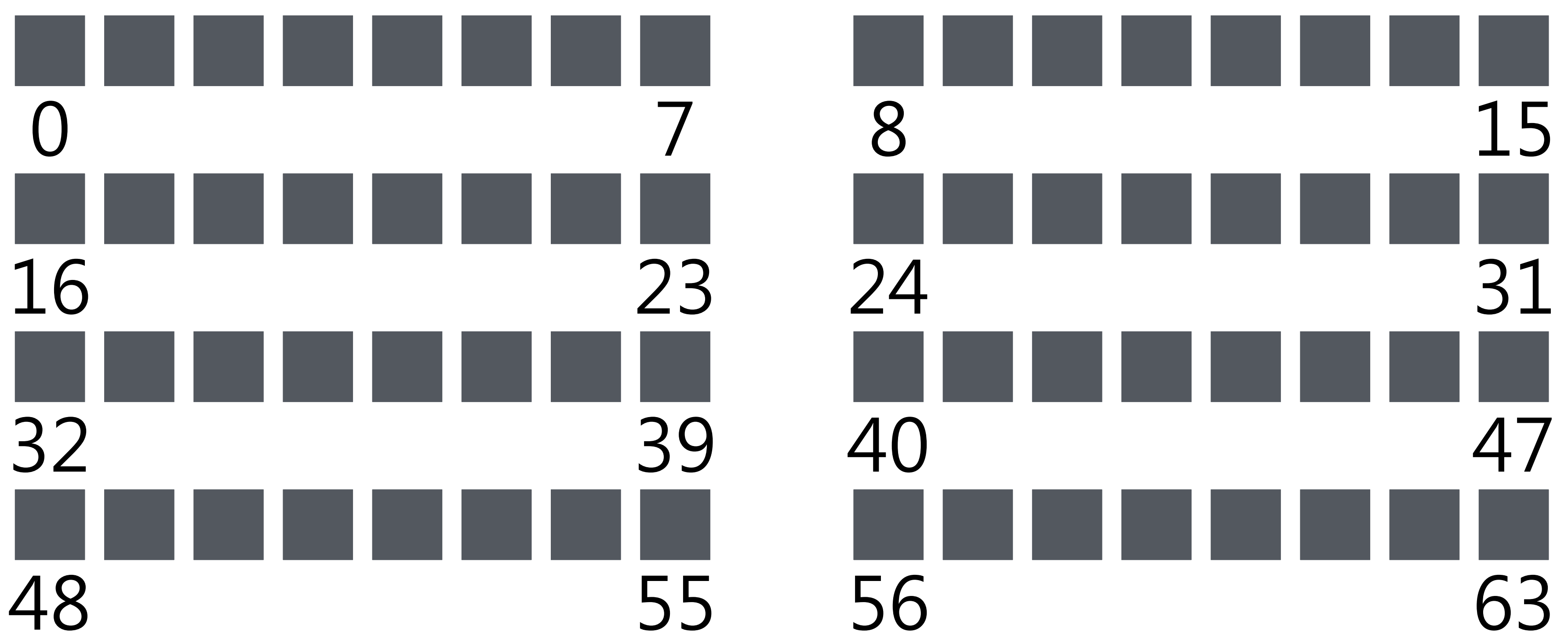
2. Access methods
   - what steps must reads/writes take?

Part 1:
Disk Structures

Persistent Store

Given: large array of blocks on disk

Want: some structure to map files to disk blocks



You could build a persistent malloc that saves to disk (instead of to memory)!

- use offsets instead of ptrs, writes instead of stores

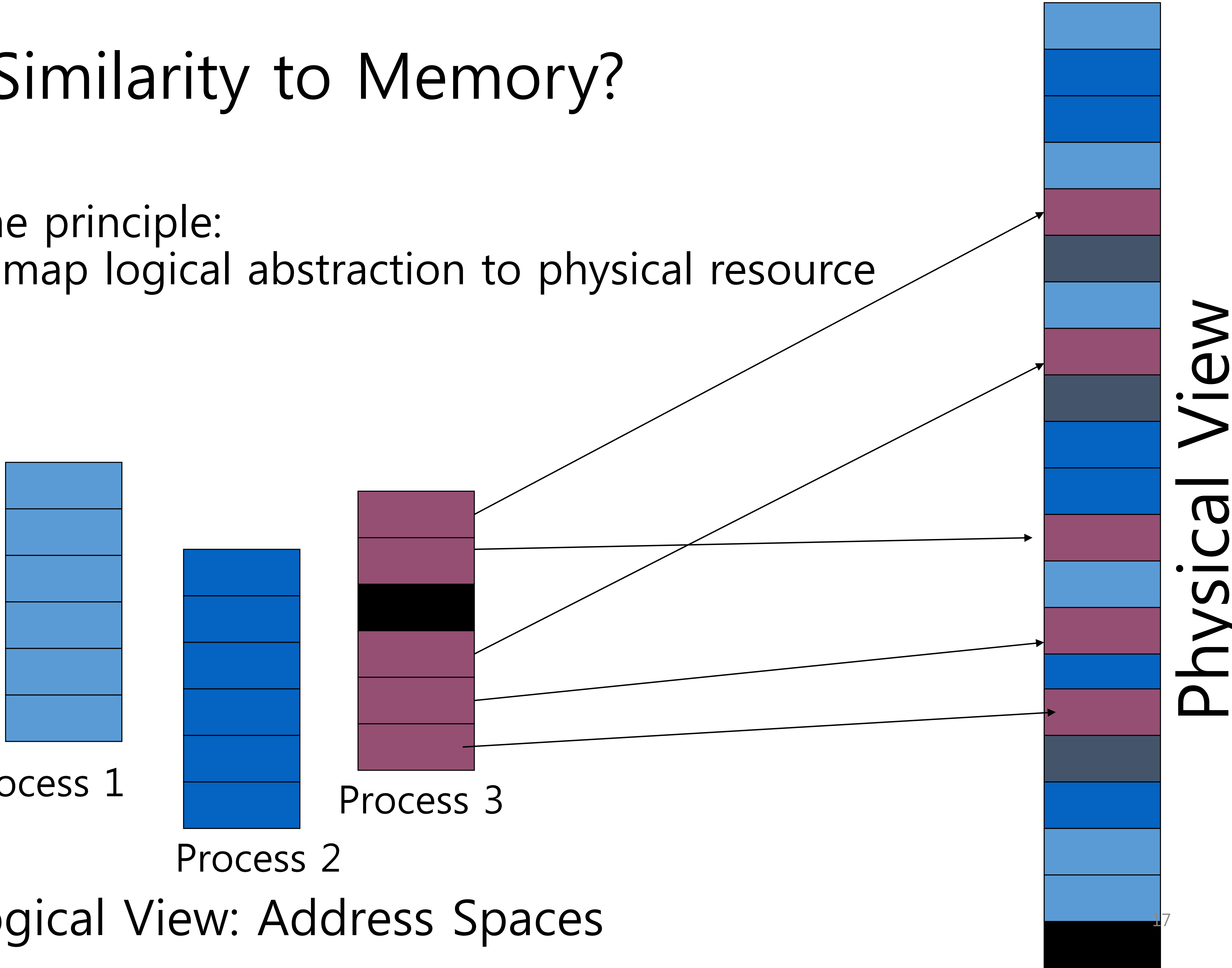
Persistent Malloc vs. FS

What features does a file system provide beyond what a persistent malloc would provide?

String names
Hierarchy (names within names)
Changeable file sizes
Sharing across processes
…

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

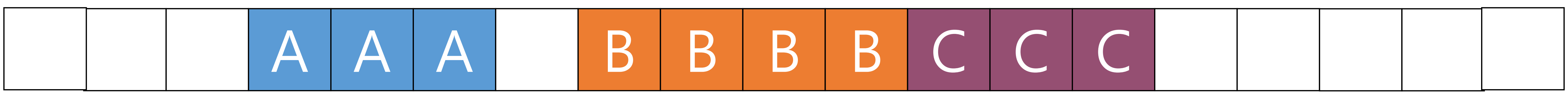
Questions

- 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 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

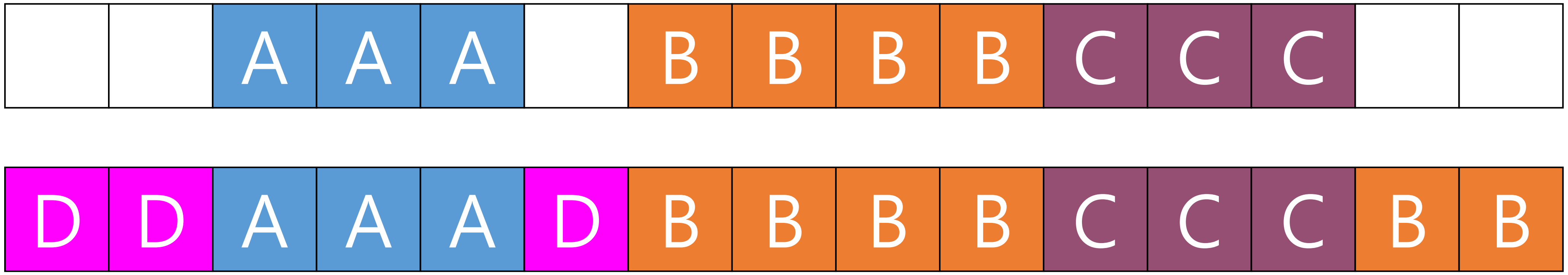
Wasted space for meta-data?

+ 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

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

Seek cost for sequential accesses? + Still good performance

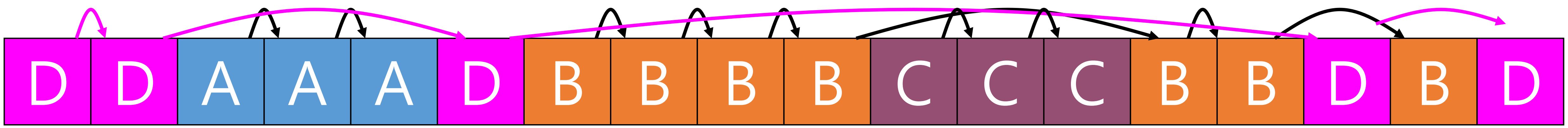
Speed to calculate random accesses? + Still simple calculation

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



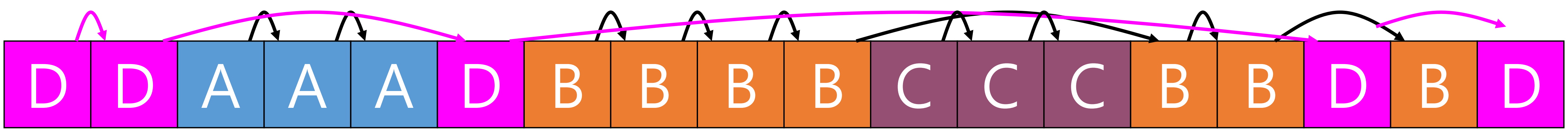
Fragmentation (internal and external)?	+ No external frag (use any block); internal?
Ability to grow file over time?	+ Can grow easily
Seek cost for sequential accesses?	+/- Depends on data layout
Speed to calculate random accesses?	- Ridiculously poor
Wasted space for meta-data?	- Waste pointer per 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?

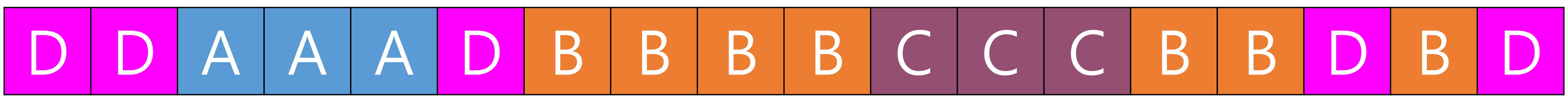
Comparison to Linked Allocation

- 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
- Supports random access

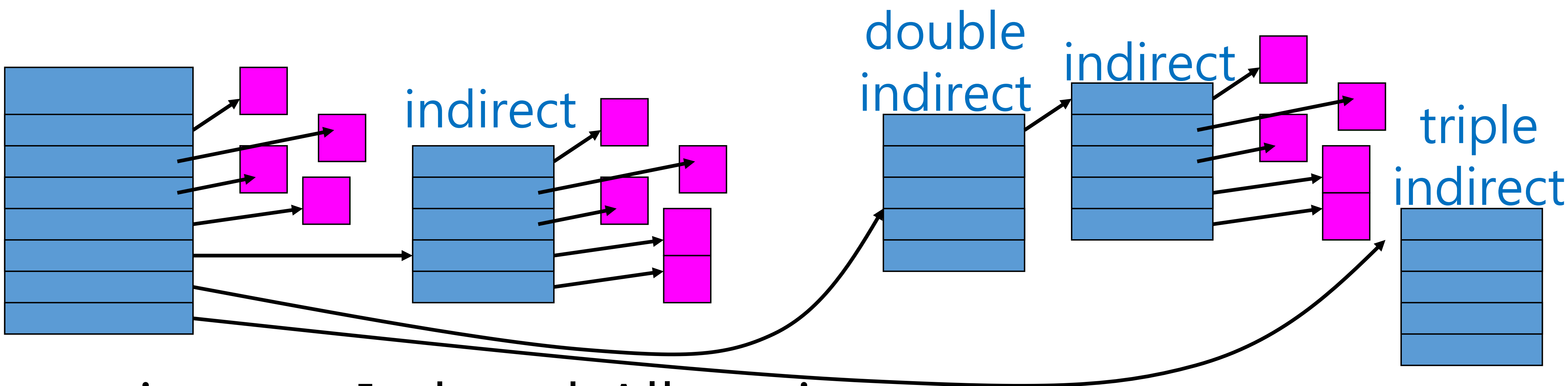
Disadvantages

- 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 what size??
- Disadvantage: Need to read indirect blocks of pointers to calculate addresses (extra disk read)
  - Keep indirect blocks cached in main memory

The Multi-Level Index

- To support bigger files, we use multi-level index.
- **Indirect pointer** points to a block that contains more pointers.
  - inode have fixed number of direct pointers (12) and a single indirect pointer.
  - If a file grows large enough, an indirect block is allocated, inode's slot for an indirect pointer is set to point to it.
    - $(12 + 1024)$ x 4 K or 4144 KB

The Multi-Level Index (Cont.)

- Double indirect pointer points to a block that contains indirect blocks.
  - Allow file to grow with an additional 1024×1024 or 1 million 4KB blocks.
- Triple indirect pointer points to a block that contains double indirect blocks.
- Multi-Level Index approach to pointing to file blocks.
  - Ex) twelve direct pointers, a single and a double indirect block.
    - over 4GB in size $(12+1024+1024^2) \times 4\text{KB}$
- Many file system use a multi-level index.
  - Linux EXT2, EXT3, NetApp's WAFL, Unix file system.
  - Linux EXT4 use extents instead of simple pointers.

The Multi-Level Index (Cont.)

Most files are small	Roughly 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 the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file system remain -50% full
Directories are typically small	Many have few entries; most have 20 or fewer

File System Measurement Summary

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
  - 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

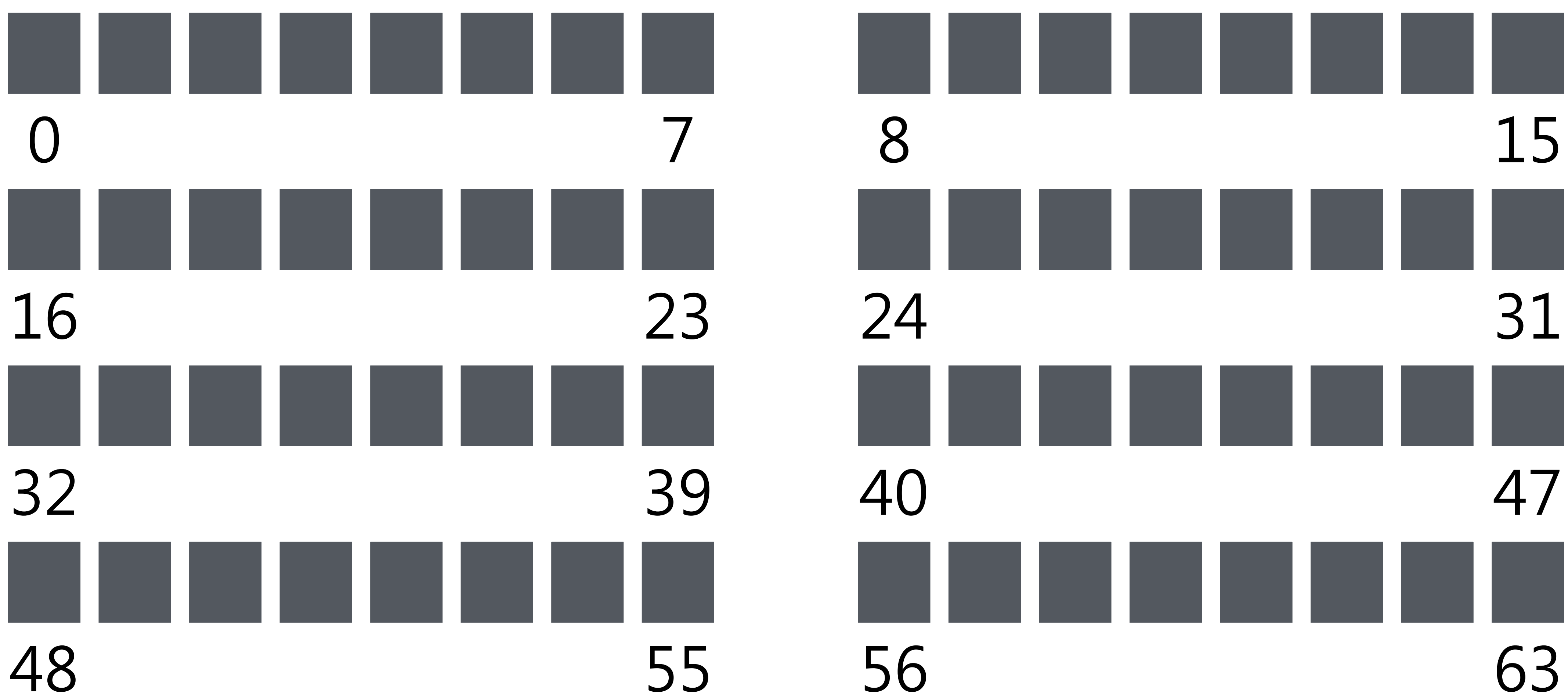
Simple approach

More complex file systems build from these basic data structures

On-Disk Structures

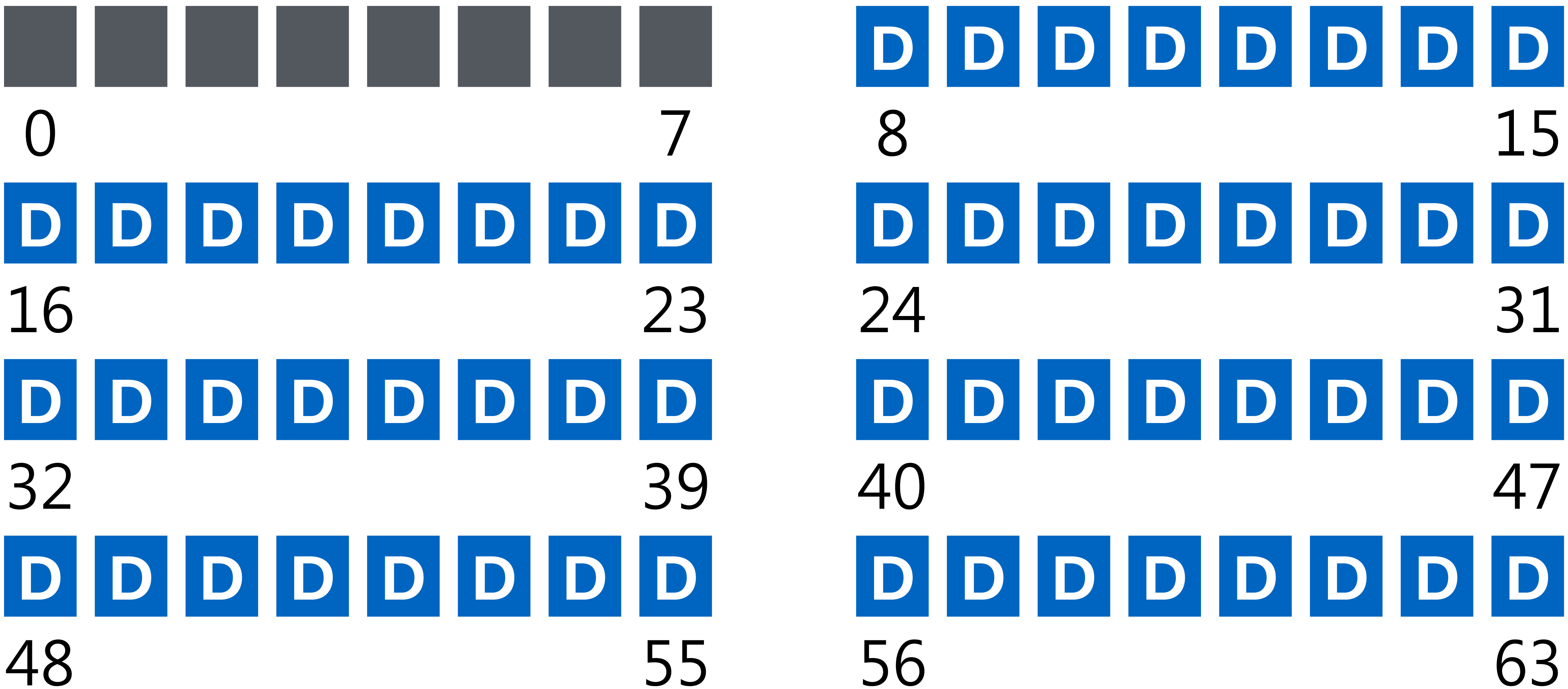
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

FS Structs: Empty Disk



Assume each block is 4KB

Data Blocks

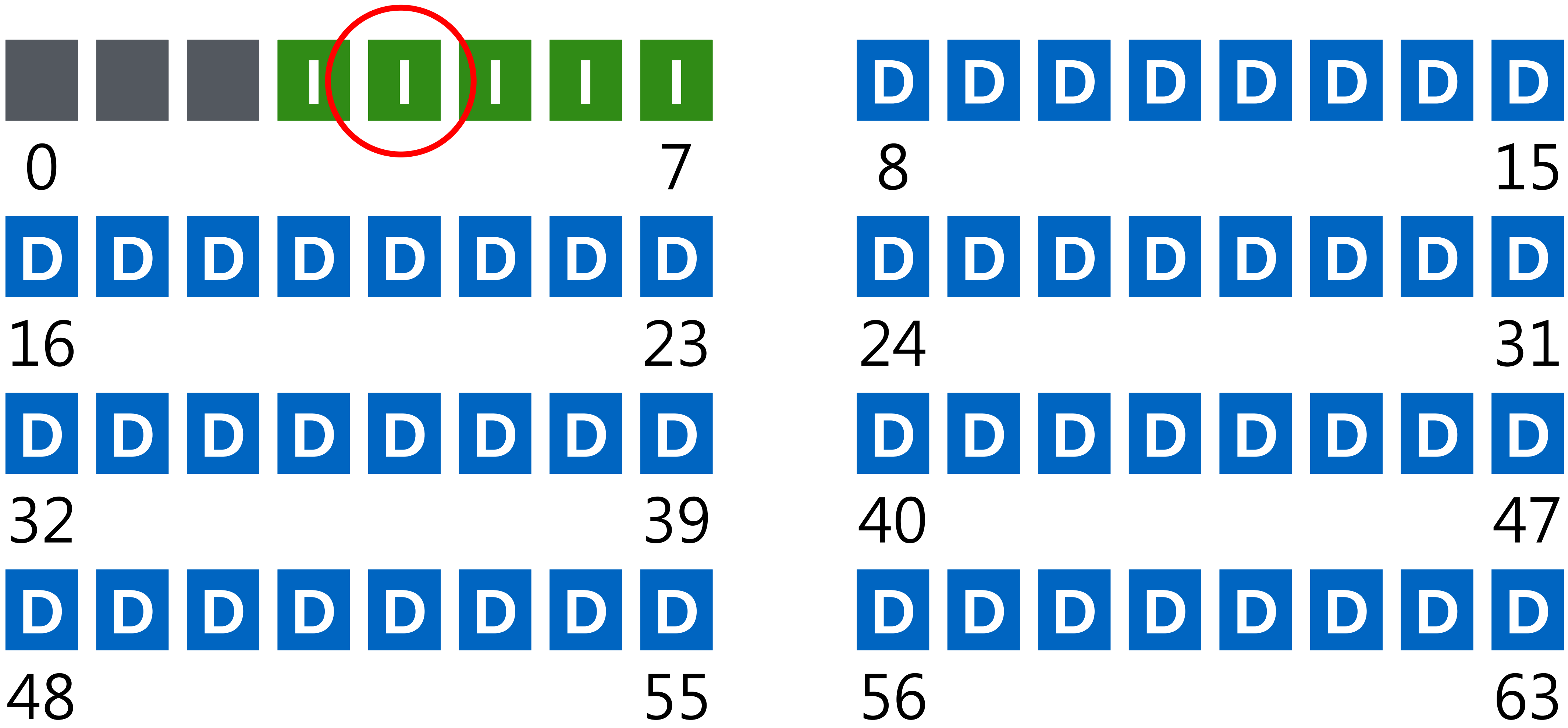


Not actual layout : Examine better layout in next lecture
Purpose: Relative number of each time of block

On-Disk Structures

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

Inodes



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)

Inode

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

file or directory?

Inode

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

user and permissions

Inode

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

size in bytes and blocks

Inode

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

access time, create time

Inode

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

how many paths

Inode

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

N data blocks

Inode

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

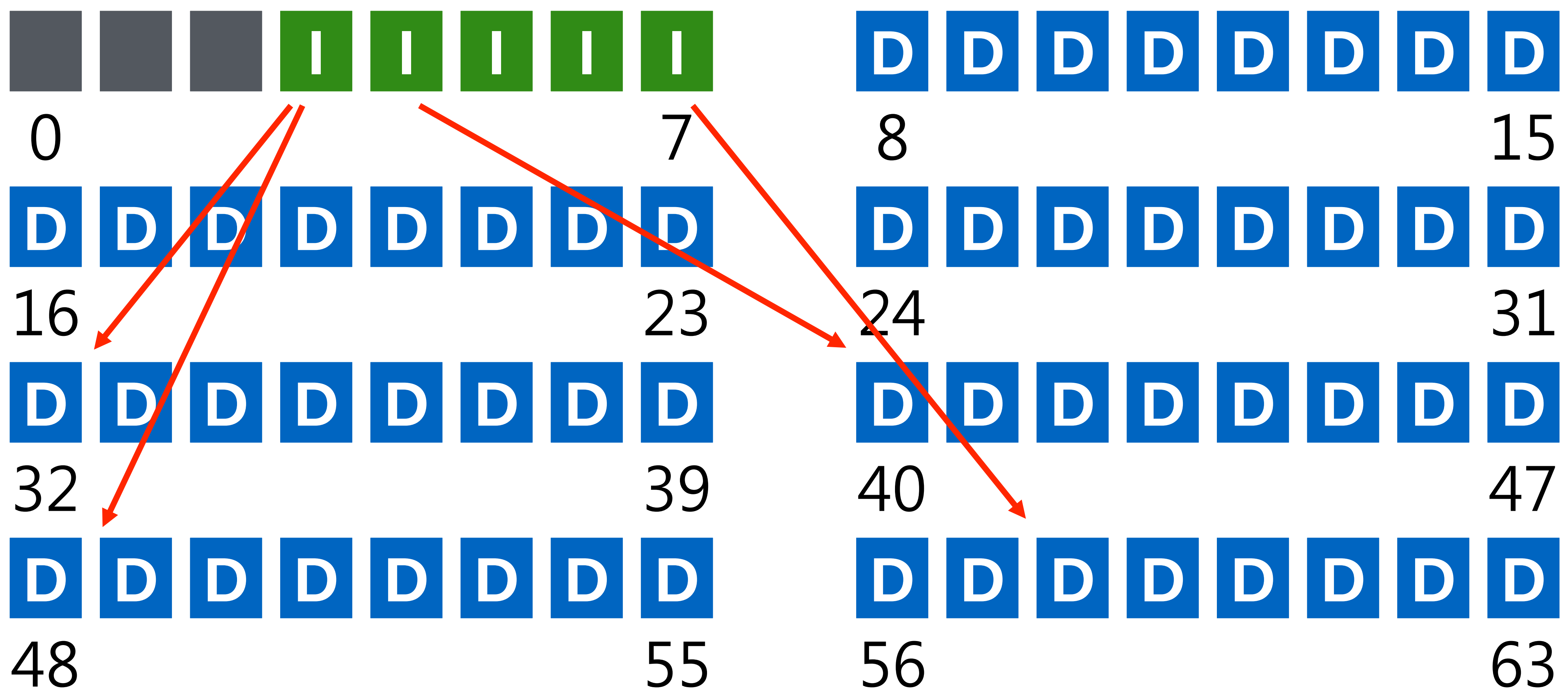
Assume single level (just pointers to data blocks)

What is max file size?
Assume 256-byte inodes (all can be used for pointers)
Assume 4-byte addrs

How to get larger files?

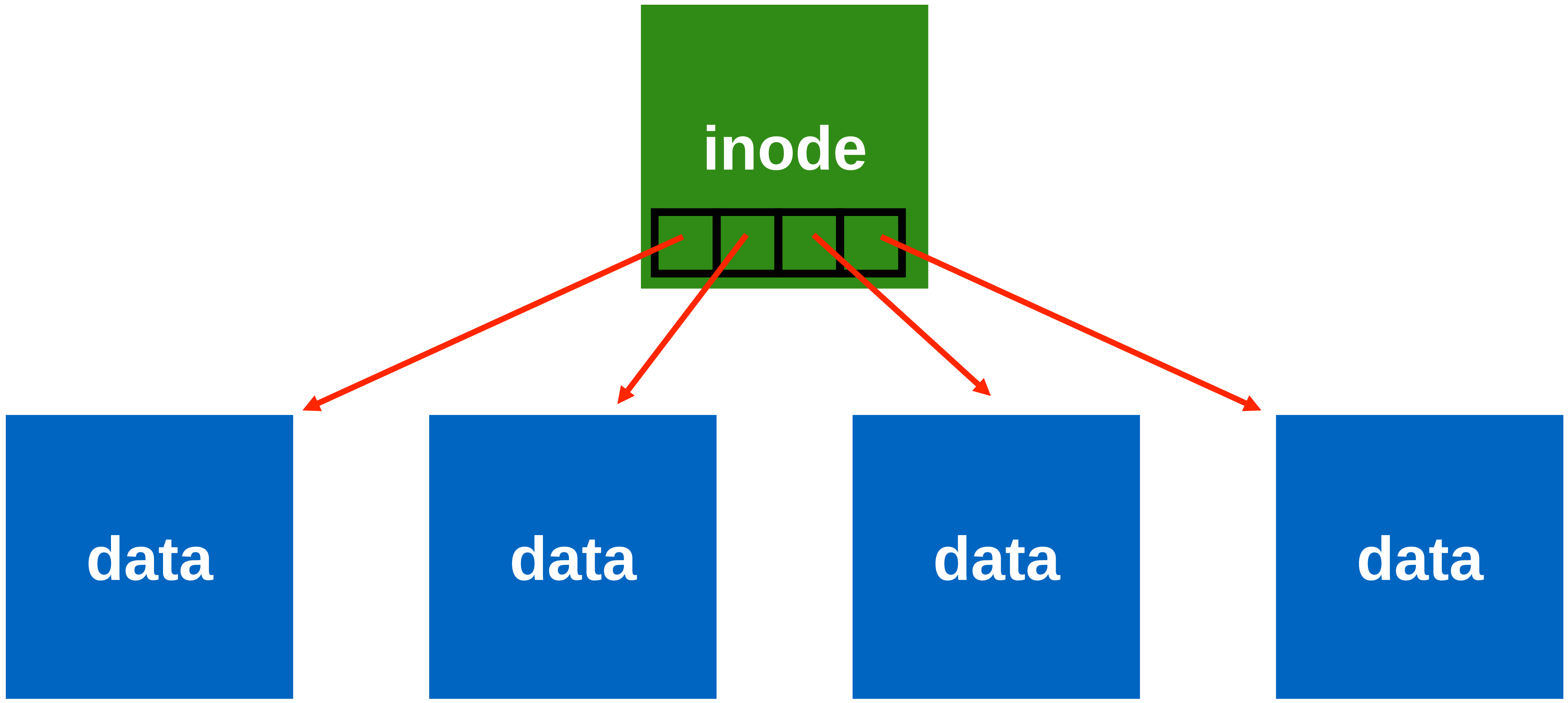
256 / 4 = 64
64 * 4K = 256 KB!

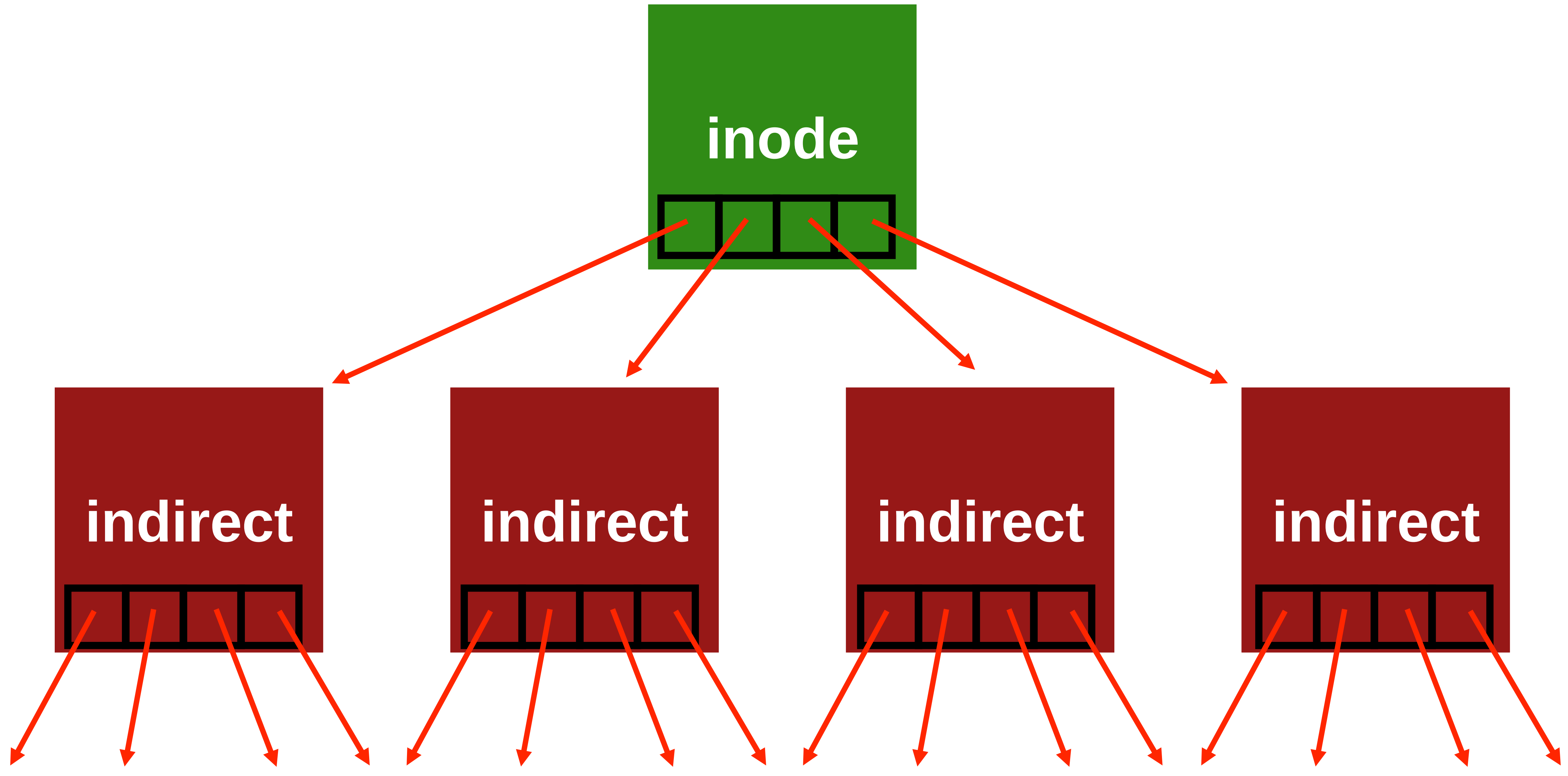
Inodes



On-Disk Structures

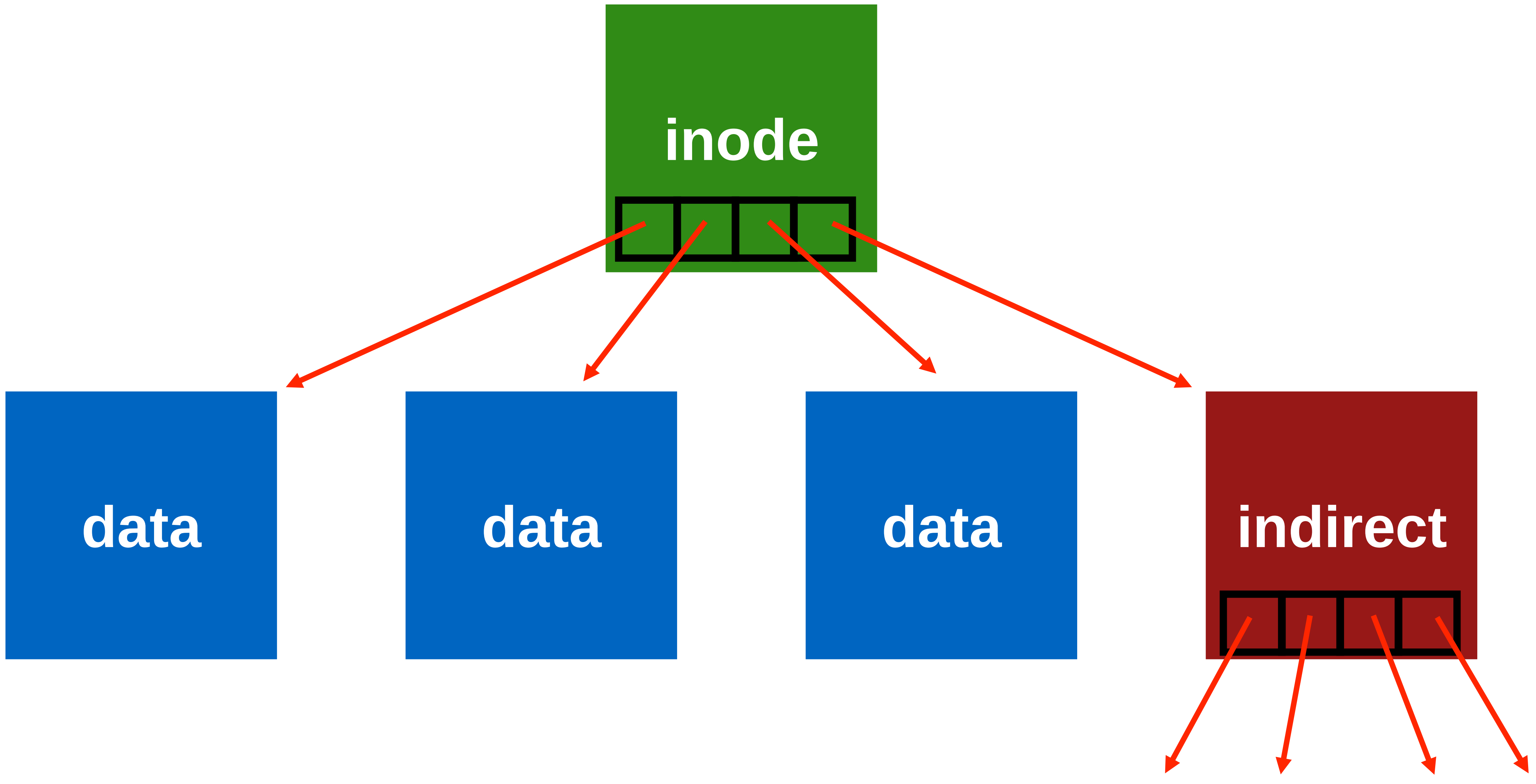
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock


inode
data
data
data
data



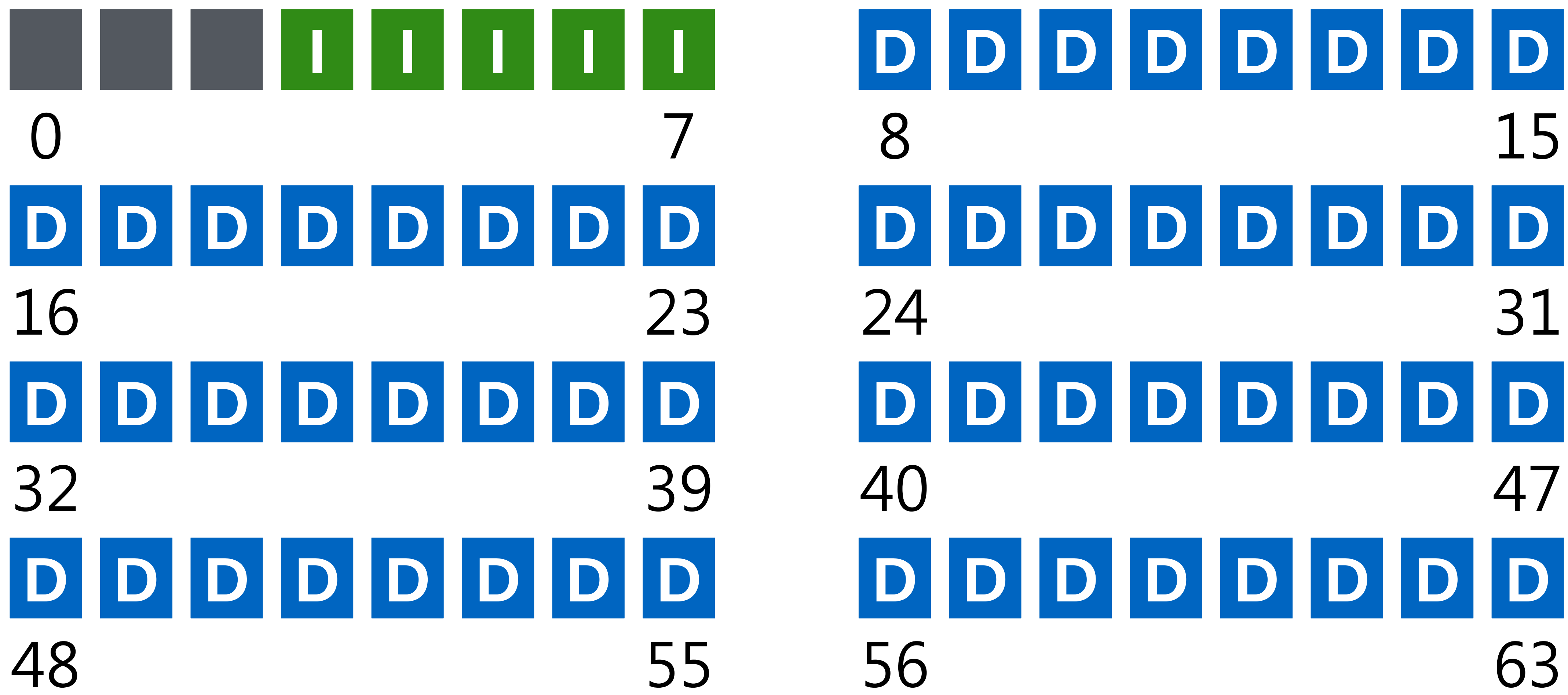
Indirect blocks are stored in regular data blocks.

what if we want to optimize for small files?

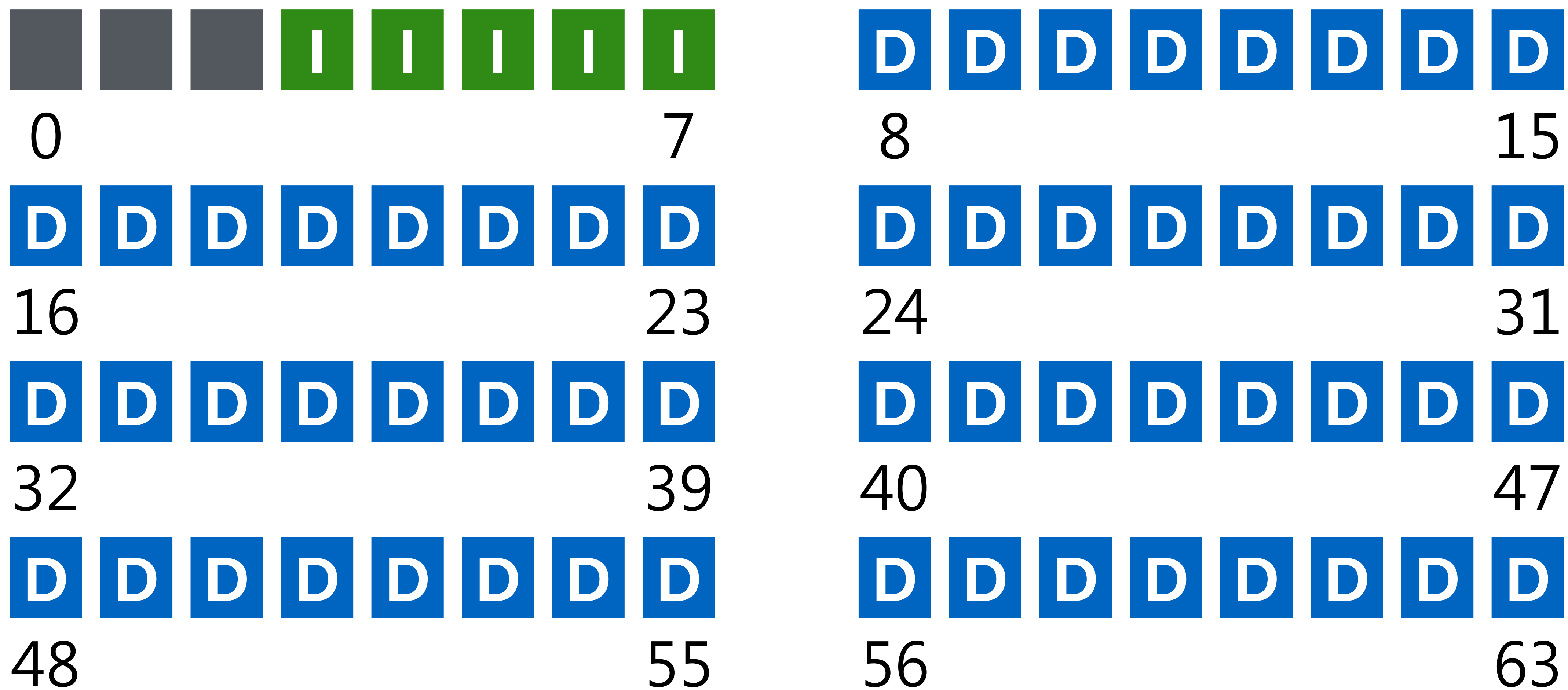


Better for small files

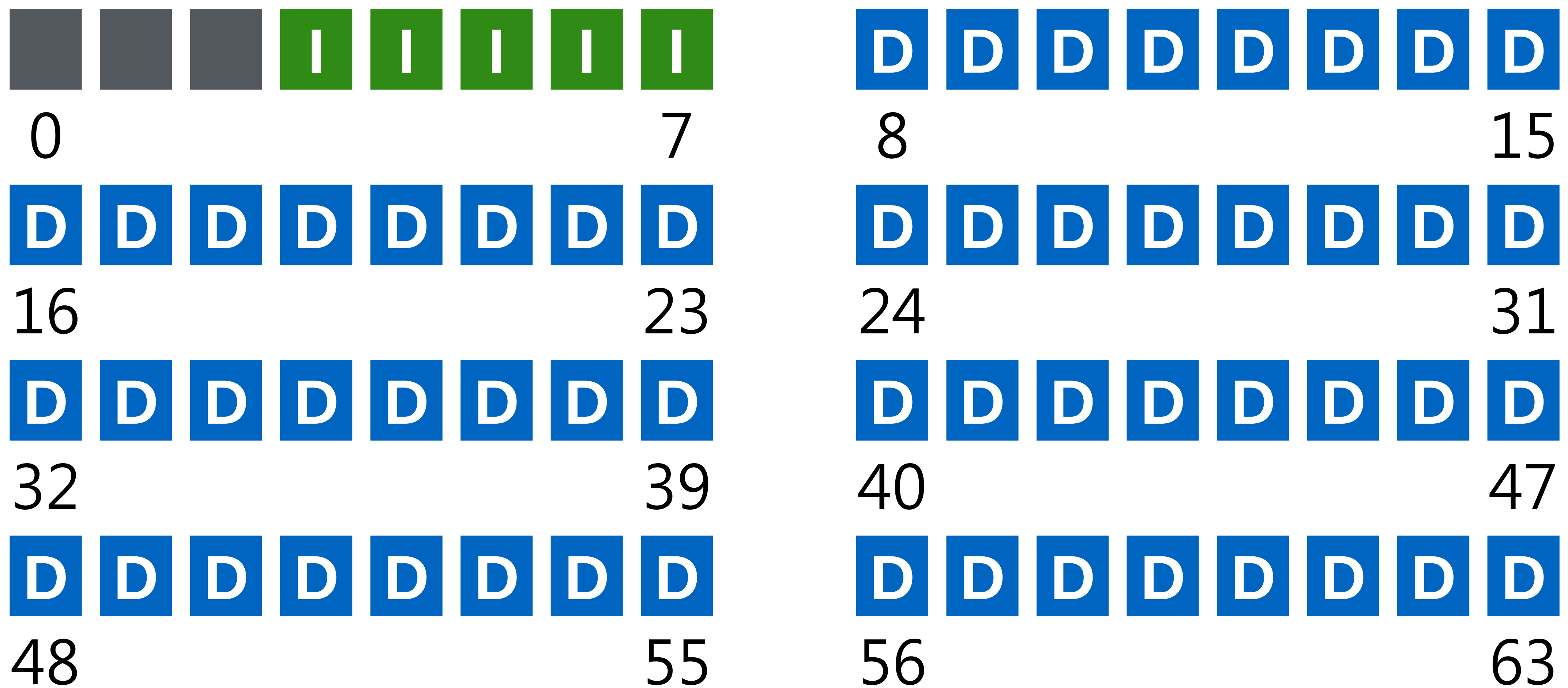
Assume 256 byte inodes (16 inodes/block).
What is offset for inode with number 0?



Assume 256 byte inodes (16 inodes/block).
What is offset for inode with number 4?



Assume 256 byte inodes (16 inodes/block).
What is offset for inode with number 40?



Various Link Structures

Tree (usually unbalanced)
- with indirect blocks
- e.g., ext3

Extents
- store offset+size pairs
- e.g., ext4

Linked list
- each data block points to the next
- e.g., FAT

On-Disk Structures

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

Directory Organization

- Directory contains a list of (entry name, inode number) pairs.
- Each directory has two extra files ."dot" for current directory and .."dot-dot" for parent directory
  - For example, `dir` has three files (`foo, bar, foobar`)

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

on-disk for dir

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

Simple Directory List Example

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

unlink("foo")

On-Disk Structures

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

Free Space Management

- File system track which inode and data block are free or not.
- In order to manage free space, we have two simple bitmaps.
  - When file is newly created, it allocated inode by searching the inode bitmap and update on-disk bitmap.
  - Pre-allocation policy is commonly used for allocate contiguous blocks.

Allocation

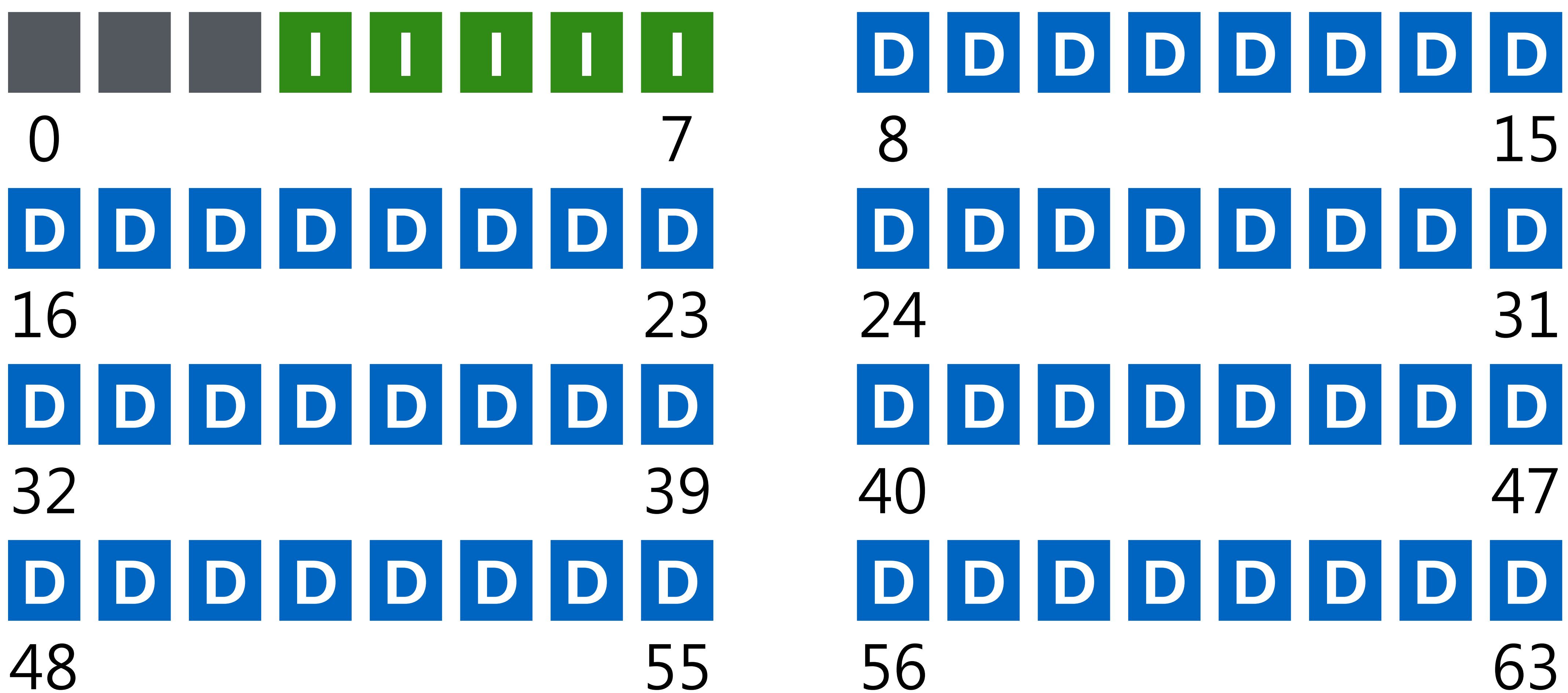
How do we find free data blocks or free inodes?

Free list

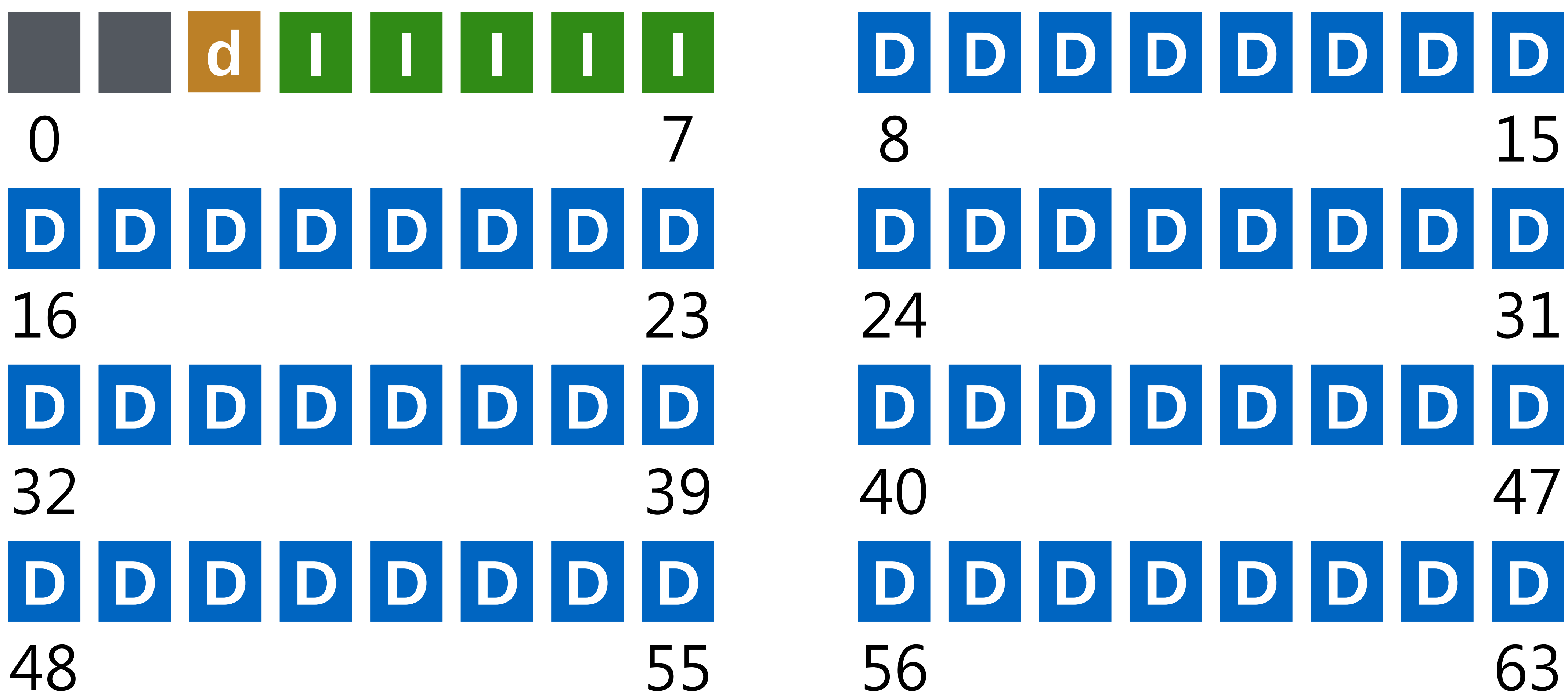
Bitmaps

Tradeoffs in next lecture…

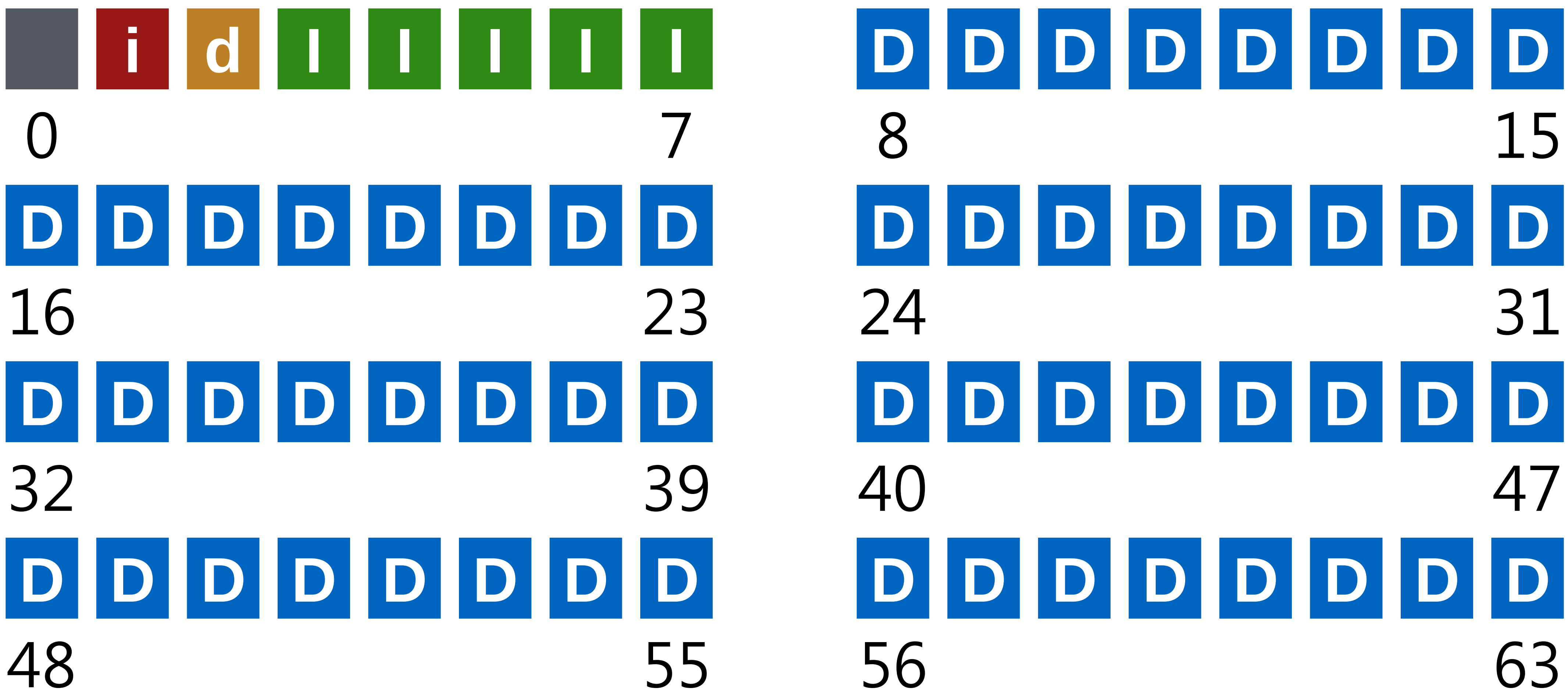
Bitmaps?


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

Data Bitmap


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

Inode bitmap
Opportunity for Inconsistency (fsck)



On-Disk Structures

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

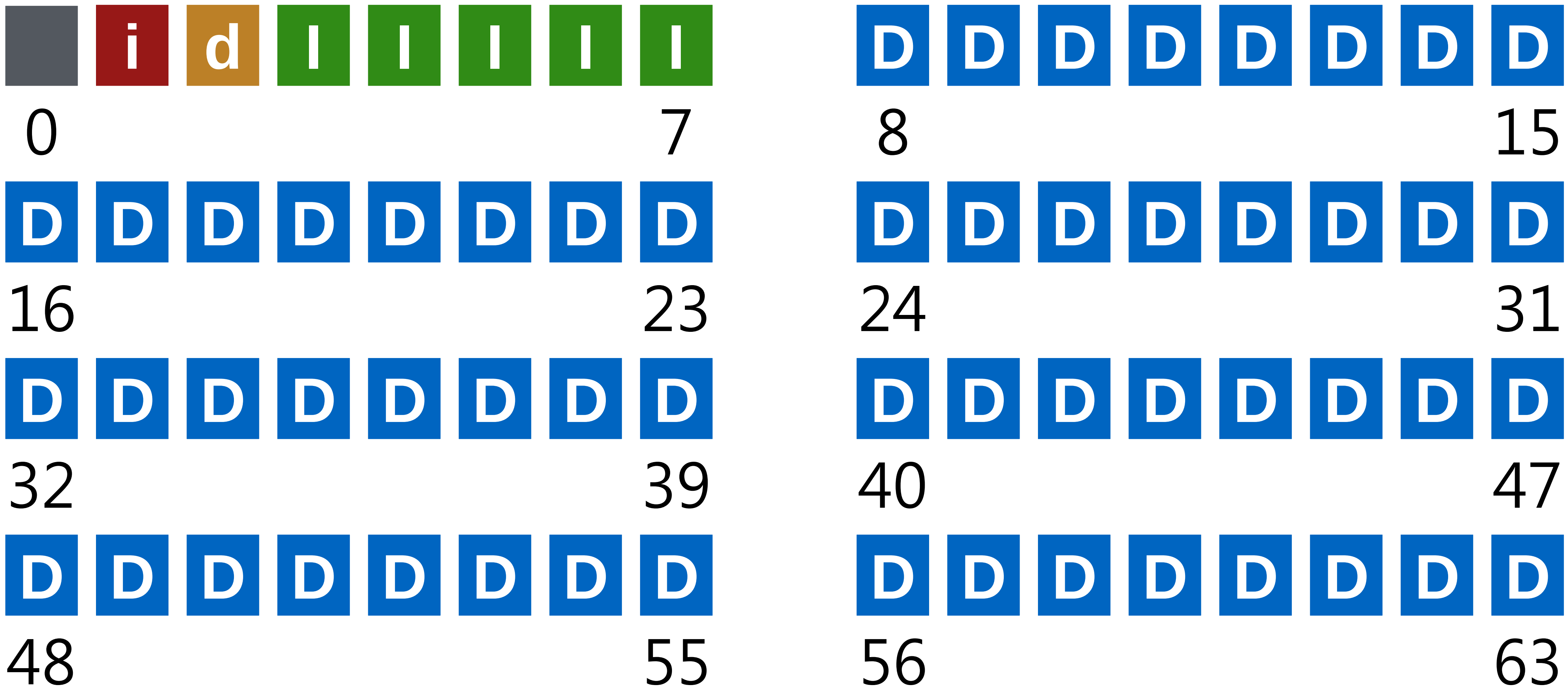
Superblock

Need to know basic FS configuration metadata, like:

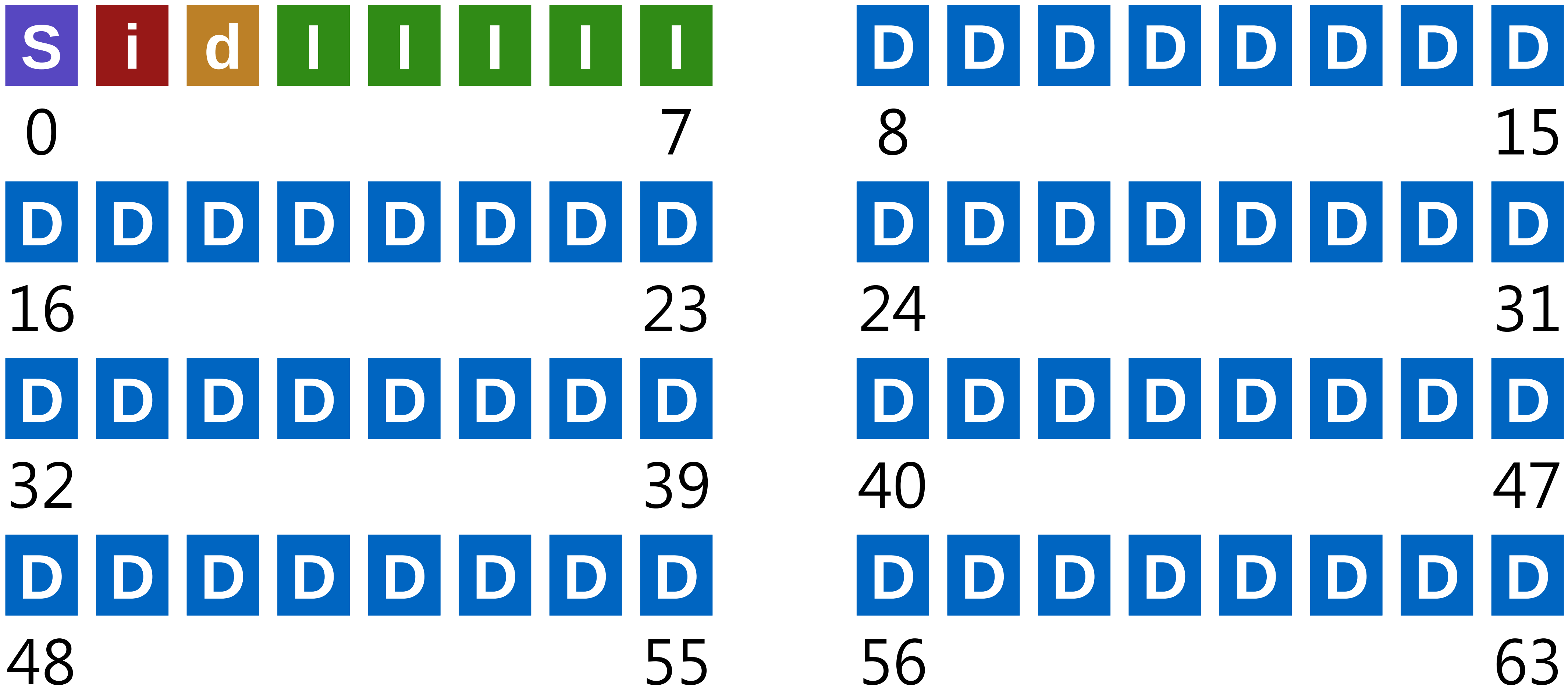
- block size
- how many inode are there
- how much free data

Store this in superblock

Super Block



Super Block



S i 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

On-Disk Structures

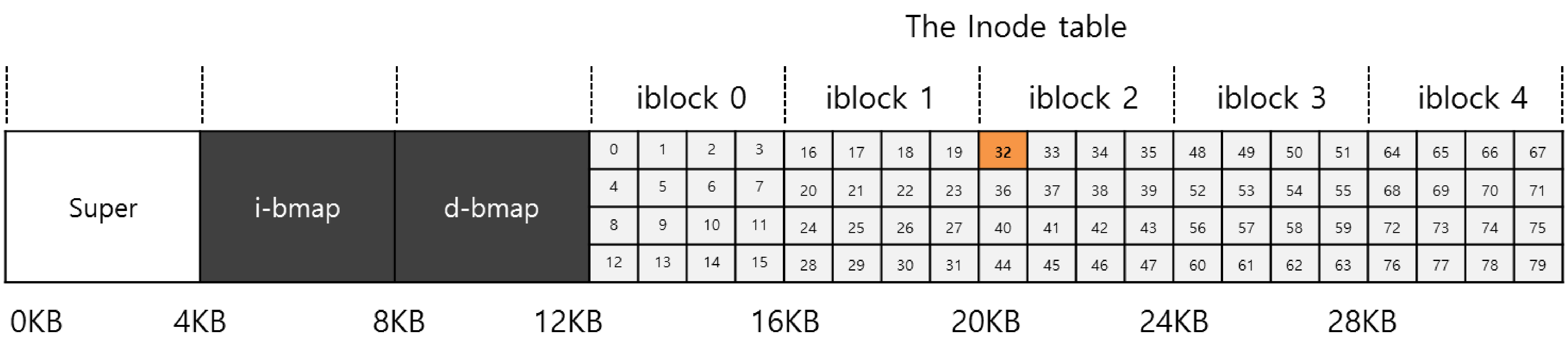
- superblock
- data block
- data bitmap
- inode table
- inode bitmap
- indirect block
- directories

On-Disk Structures

Core	Performance
Super Block	
Data Block (directories, indirects)	Data Bitmap
Inode Table	Inode Bitmap

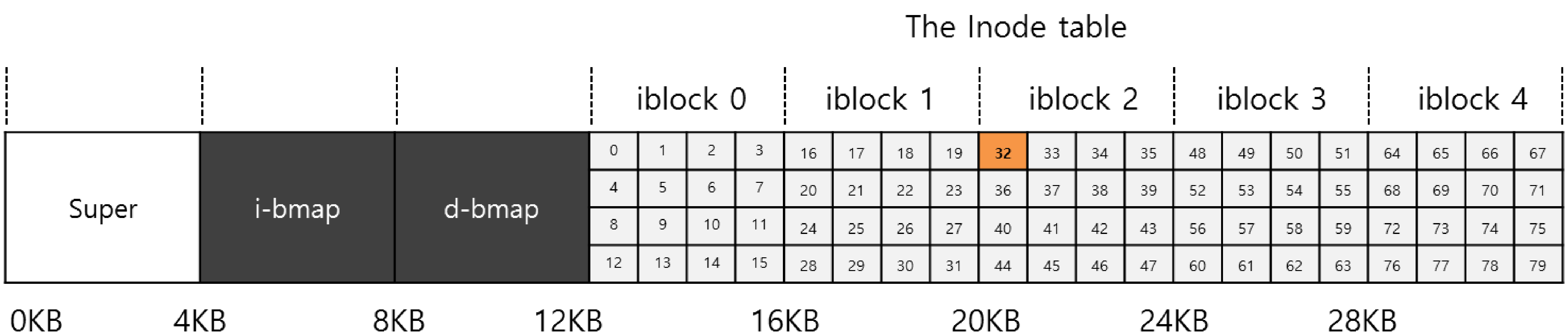
File Organization: The inode

- Each inode is referred to by inode number.
  - by inode number, File system calculate where the inode is on the disk.
  - Ex) inode number: 32
    - Calculate the offset into the inode region (32 x sizeof(inode) (256 bytes) = 8192
    - Add start address of the inode table(12 KB) + inode region(8 KB) = 20 KB



File Organization: The inode (Cont.)

- Disk are not byte addressable, sector addressable.
- Disk consist of a large number of addressable sectors, (512 bytes)
  - Ex) Fetch the block of inode (inode number: 32)
    - Sector address `iaddr` of the inode block:
    - `blk : (inumber * sizeof(inode)) / blocksize`
    - `sector : (blk * blocksize) + inodeStratAddr ) /sectorsize`



File Organization: The inode (Cont.)

- `inode` have all of the information about a file
  - File type (regular file, directory, etc.),
  - Size, the number of blocks allocated to it.
  - Protection information(who ones the file, who can access, etc).
  - Time information.
  - Etc.

File Organization: The inode (Cont.)

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
4	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
2	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

The EXT2 Inode

Part 2 : Operations

FS
- mkfs
- mount

File
- create file
- write
- open
- read
- close

mkfs

Different version for each file system (e.g., mkfs.ext4, mkfs.xfs, mkfs.btrfs, etc)

- Initialize metadata (bitmaps, inode table).
- Create empty root directory.

Part 2 : Operations

FS

- mkfs
- mount

File

- create file
- write
- open
- read
- close

mount

Add the file system to the FS tree.

Minimally requires reading superblock.

Part 2 : Operations

FS

- mkfs
- mount

File

- 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?

Part 2 : Operations

FS

- mkfs
- mount

File

- create file
- write
- open
- read
- close

Access Paths: Writing to Disk

- Issue `write()` to update the file with new contents.
- File may allocate a block (unless the block is being overwritten).
  - Need to update data block, data bitmap.
  - It generates five I/Os:
    - one to read the data bitmap
    - one to write the bitmap (to reflect its new state to disk)
    - two more to read and then write the inode
    - one to write the actual block itself.
  - To create file, it also allocate space for directory, causing high I/O traffic.

Access Paths: Writing to Disk (Cont.)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create			read							
(/foo/bar)						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write							write		
					write					
write()					read					
	read									
	write								write	
					write					
write()					read					
	read									write
	write									
					write					

File Creation Timeline (Time Increasing Downward)

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			

Part 2 : Operations

FS
- mkfs
- mount

File
- create file
- write
- open
- read
- close

open /foo/bar

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

Part 2 : Operations

FS

- mkfs
- mount

File

- create file
- write
- open
- read
- close

Access Paths: Reading a File From Disk

- Issue an `open("/foo/bar", O_RDONLY)`,
  - Traverse the pathname and thus locate the desired indoe.
  - Begin at the root of the file system `(/)`
    - In most Unix file systems, the root inode number is 2
  - Filesystem reads in the block that contains inode number 2.
  - Look inside of it to find pointer to data blocks (contents of the root).
  - By reading in one or more directory data blocks, It will find "foo" directory.
  - Traverse recursively the path name until the desired inode ("bar")
  - Check finale permissions, allocate a file descriptor for this process and returns file descriptor to user.

Access Paths: Reading a File From Disk (Cont.)

- Issue `read()` to read from the file.
  - Read in the first block of the file, consulting the inode to find the location of such a block.
    - Update the inode with a new last accessed time.
    - Update in-memory open file table for file descriptor, the file offset.

- When file is closed:
  - File descriptor should be deallocated, but for now, that is all the file system really needs to do. No dis I/Os take place.

Access Paths: Reading a File From Disk (Cont.)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

File Read Timeline (Time Increasing Downward)

read /foo/bar – assume opened

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

Part 2 : Operations

FS

- mkfs
- mount

File

- create file
- write
- open
- read
- close

close /foo/bar

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

nothing to do on disk!

Efficiency

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

Cache for:
- reads
- write buffering

Caching and Buffering

- Reading and writing files are expensive, incurring many I/Os.
  - For example, long pathname(/1/2/3/…./100/file.txt)
    - One to read the inode of the directory and at least one read its data.
    - Literally perform hundreds of reads just to open the file.

- In order to reduce I/O traffic, file systems aggressively use system memory(DRAM) to cache.
  - Early file system use fixed-size cache to hold popular blocks.
    - Static partitioning of memory can be wasteful;
  - Modem systems use dynamic partitioning approach, unified page cache.
- Read I/O can be avoided by large cache.

Caching and Buffering (Cont.)

- Write traffic has to go to disk for persistent, Thus, cache does not reduce write I/Os.
- File system use write buffering for write performance benefits.
  - delaying writes (file system batch some updates into a smaller set of I/Os).
  - By buffering a number of writes in memory, the file system can then schedule the subsequent I/Os.
  - By avoiding writes

- Some application force flush data to disk by calling `fsync()` or direct I/O.

Structures

What data is likely to be read frequently?
- superblock
- data block
- data bitmap
- inode table
- inode bitmap
- indirect block
- directories

Unified Page Cache

Instead of a dedicated file-system cache, draw pages from a common pool for FS and processes.

API change:
- read
- shrink_cache (Linux)

LRU Example

Ops	Hits	State
read 1	miss	1
read 2	miss	1,2
read 3	miss	1,2,3
read 4	miss	1,2,3,4
shrink	-	2,3,4
shrink	-	3,4
read 1	miss	1,3,4
read 2	miss	1,2,3,4
read 3	hit	1,2,3,4
read 4	hit	1,2,3,4

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?

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**?