Slides: Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau

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)

TODAY: IMPLEMENTATION

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

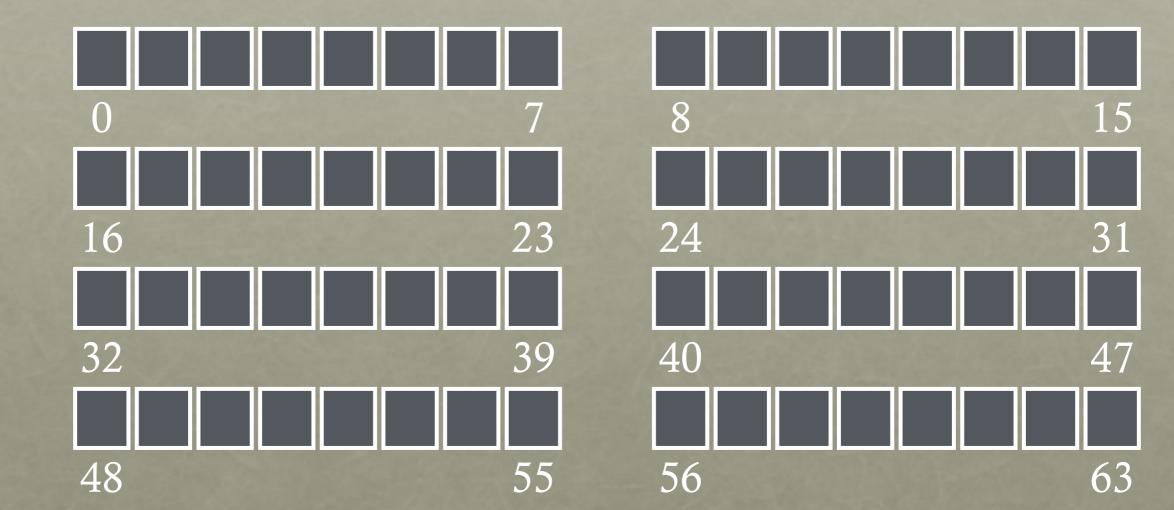
- 2. Access methods
 - what steps must reads/writes take?

PART 1: DISK STRUCTURES

PERSISTENT STORE

Given: large array of blocks on disk

Want: some structure to map files to disk blocks



SIMLARITY 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

AABBBCC

Fragmentation (internal and external)?Ability to grow file over time?Seek cost for sequential accesses?Speed to calculate random accesses?Wasted space for meta-data?

- Horrible external frag (needs periodic compaction)
- May not be able to without moving
- + Excellent performance
- + Simple calculation
- + Little overhead for meta-data

SMALL # OF EXTENTS

Allocate multiple contiguous regions (extents) per file

Meta-data: Small arr

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

A A A B B B B C <thC</th> <thC</th> <thC</th>

Fragmentation (internal and external)?Ability to grow file over time?Seek cost for sequential accesses?Speed to calculate random accesses?Wasted space for meta-data?

- Helps external fragmentation
- Can grow (until run out of extents)
- + Still good performance
- + Still simple calculation
- + 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

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

Fragmentation (internal and external)?Ability to grow file over time?Seek cost for sequential accesses?Speed to calculate random accesses?Wasted space for meta-data?

- + No external frag (use any block); internal?
- + Can grow easily
- +/- Depends on data layout
- Ridiculously poor
- 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

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

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

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

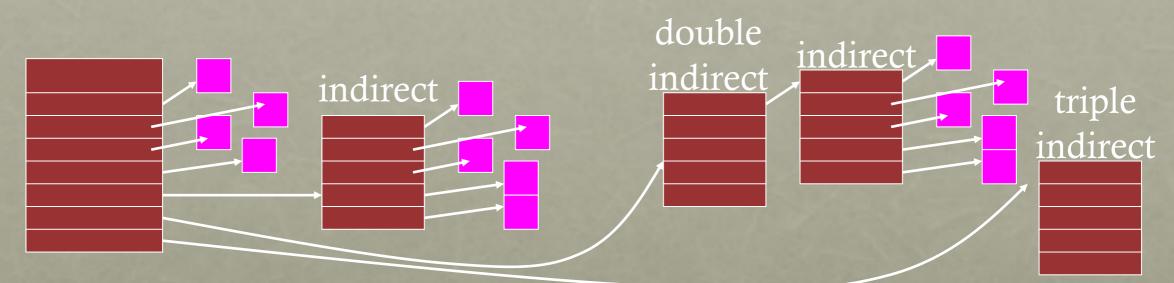
Advantages

- 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!)

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

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)?Ability to grow file over time?Seek cost for sequential accesses?Speed to calculate random accesses?Wasted space for meta-data?

- + Both reasonable
- + Can grow
- + Still good performance
- +/- Some calculations depending on size
- + Relatively small overhead

ASSUME MULTI-LEVEL INDEXING

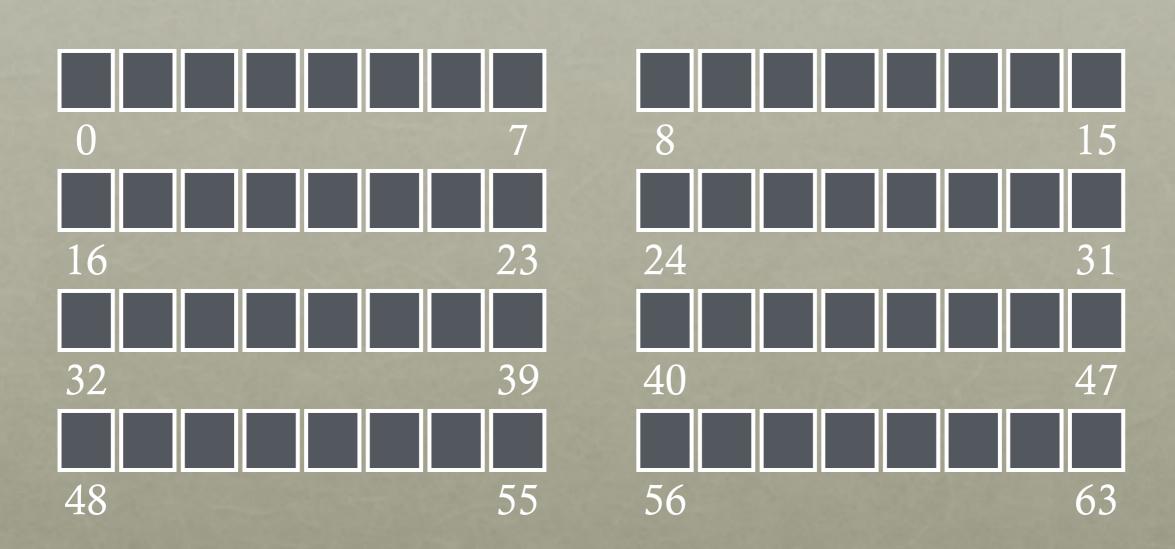
Simple approach

More complex file systems build from these basic data structures

ON-DISK STRUCTURES

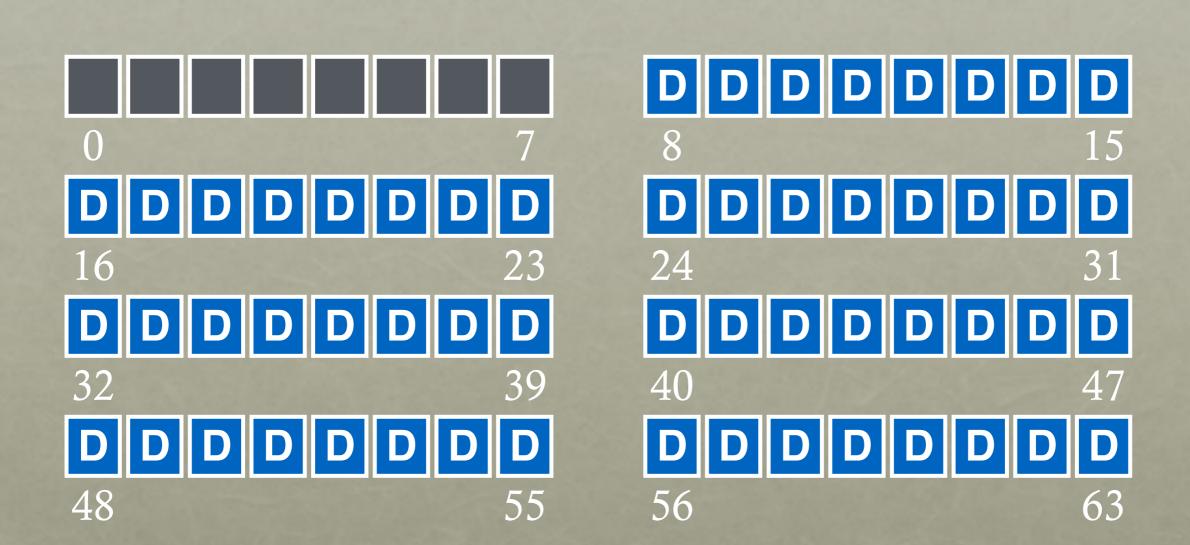
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

FS STRUCTS: EMPTY DISK



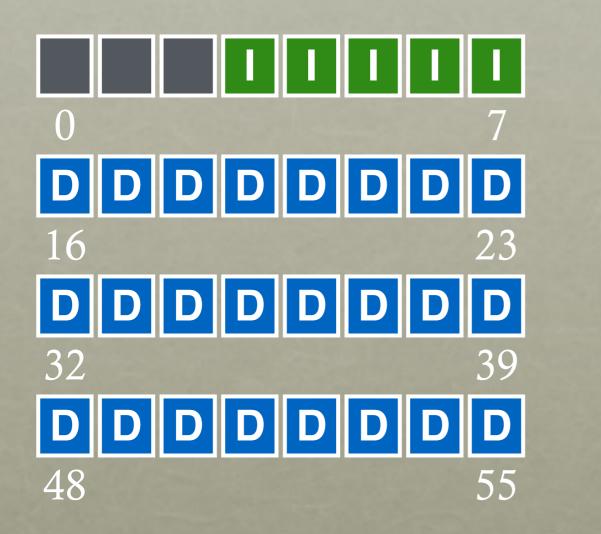
Assume each block is 4KB

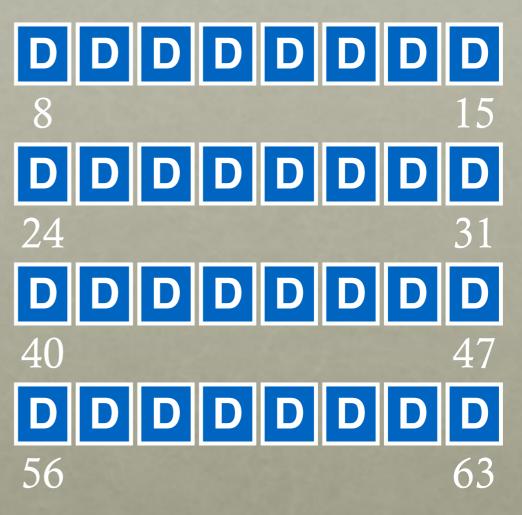
DATA BLOCKS



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

INODES





ONE INODE BLOCK

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

4KB disk block

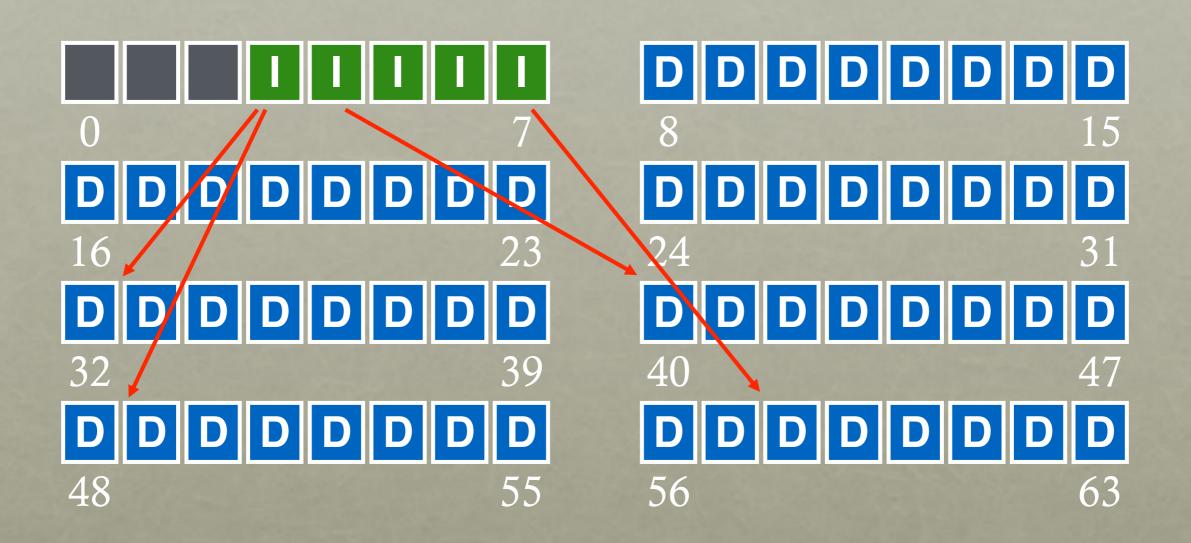
16 inodes per inode block.

inode	inode	inode	inode	
16	17	18	19	
inode	inode	inode	inode	
20	21	22	23	
inode	inode	inode	inode	
24	25	26	27	
inode	inode	inode	inode	
28	29	30	31	

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



INODE

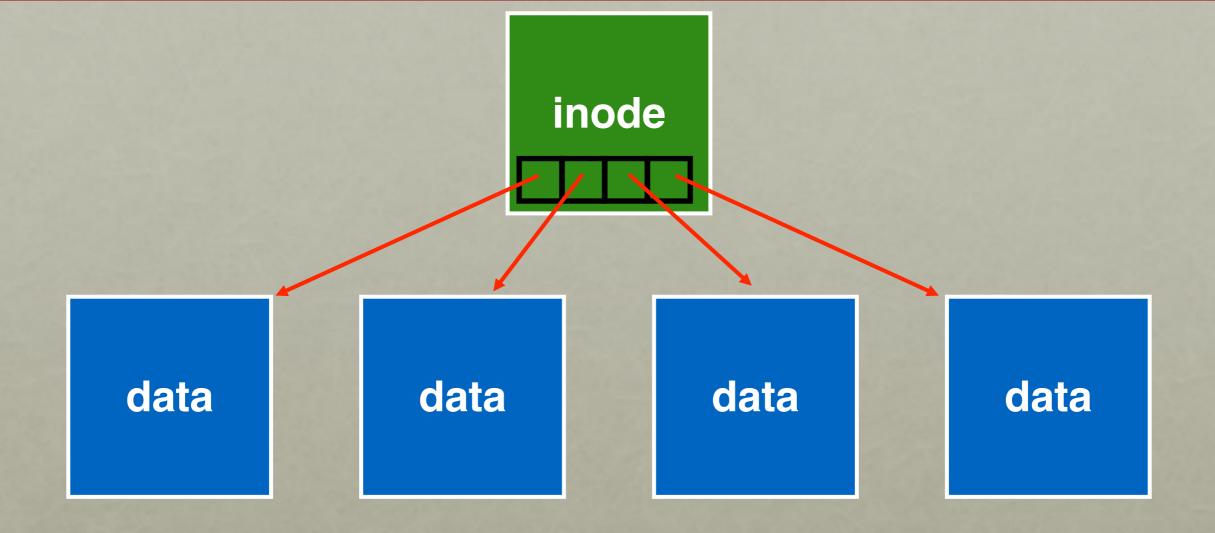
type uid rwx size blocks time ctime links_count addrs[N]

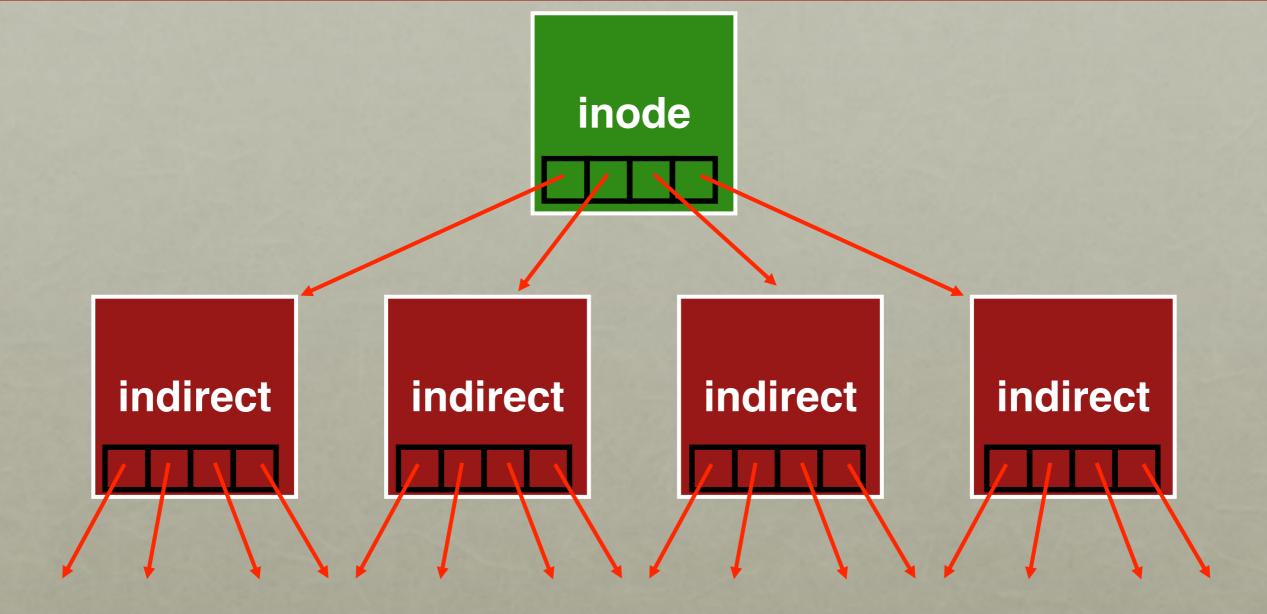
Assume single level (just pointers to data blocks)

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

How to get larger files?

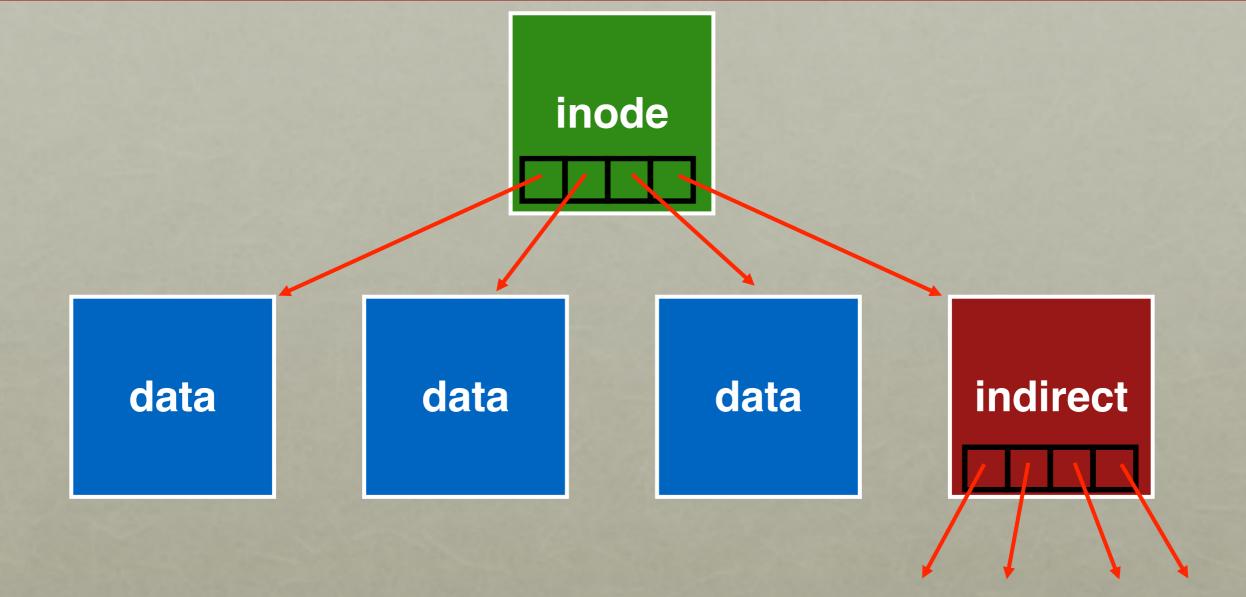
256 / 4 = 64 64 * 4K = 256 KB!





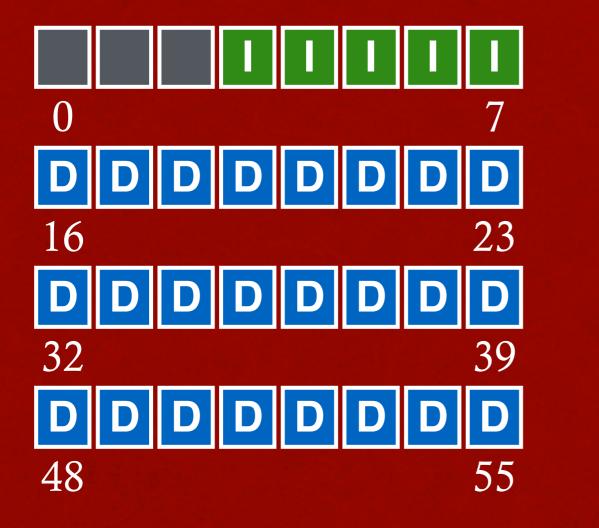
Indirect blocks are stored in regular data blocks.

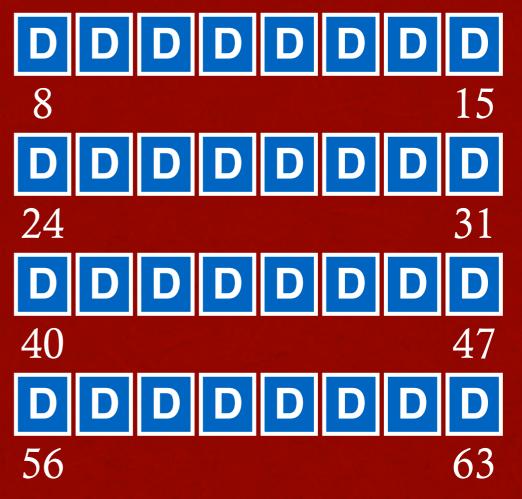
what if we want to optimize for small files?



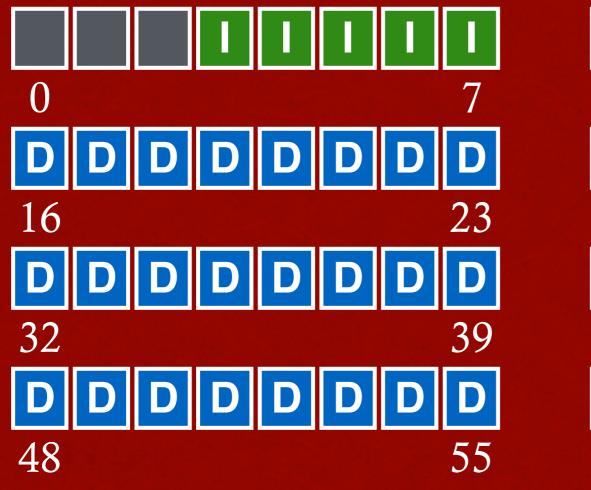
Better for small files

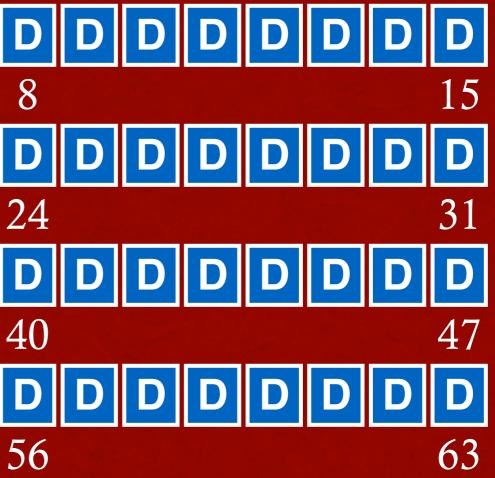
Assume 256 byte inodes (16 inodes/block). What is offset for inode with number 0?



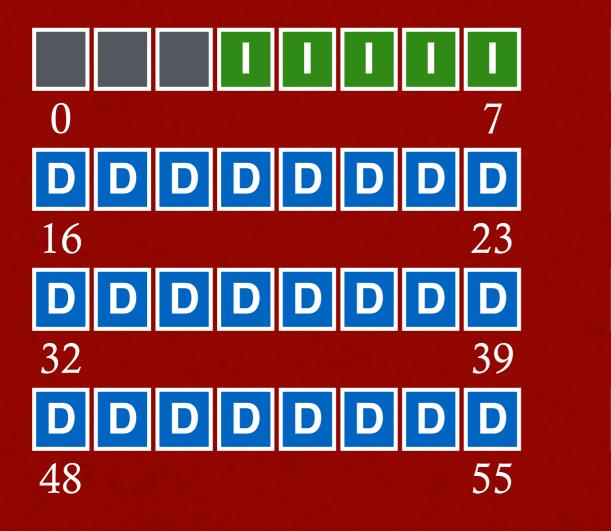


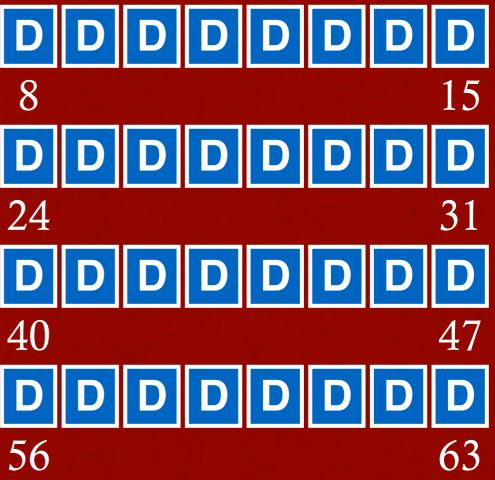
Assume 256 byte inodes (16 inodes/block). What is offset for inode with number 4?





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





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

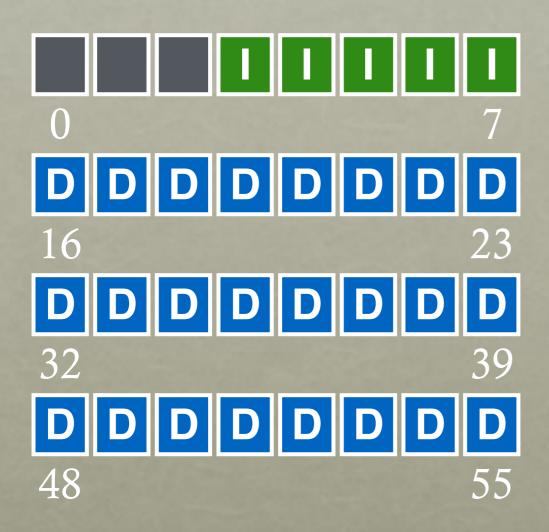
How do we find free data blocks or free inodes?

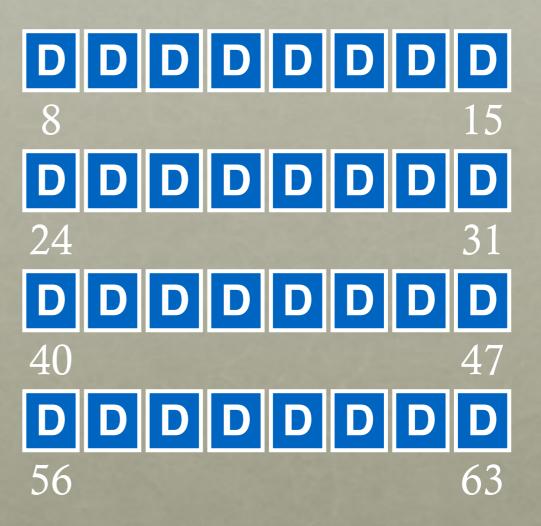
Free list

Bitmaps

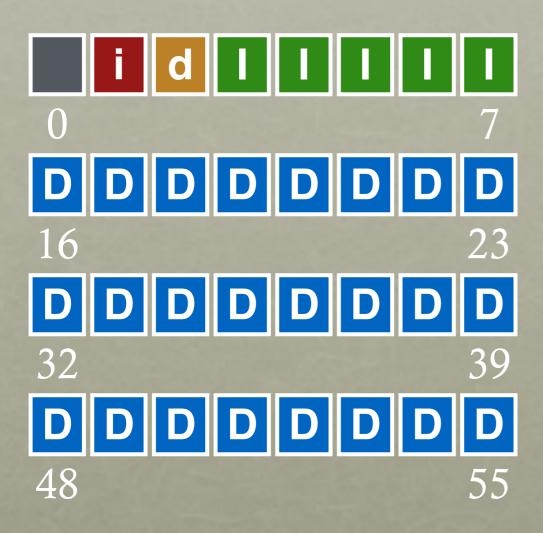
Tradeoffs in next lecture...

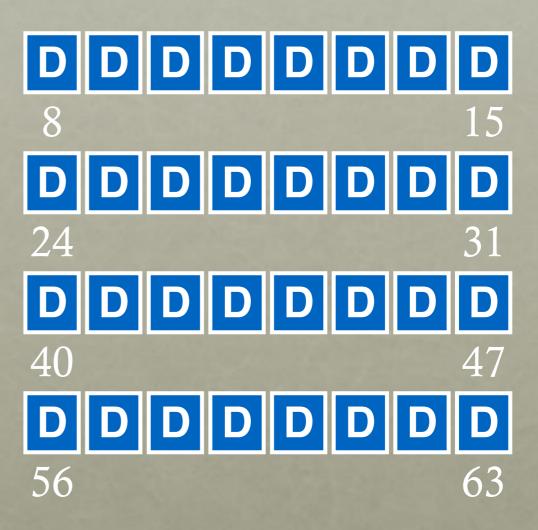
BITMAPS?





OPPORTUNITY FOR INCONSISTENCY (FSCK)





SUPERBLOCK

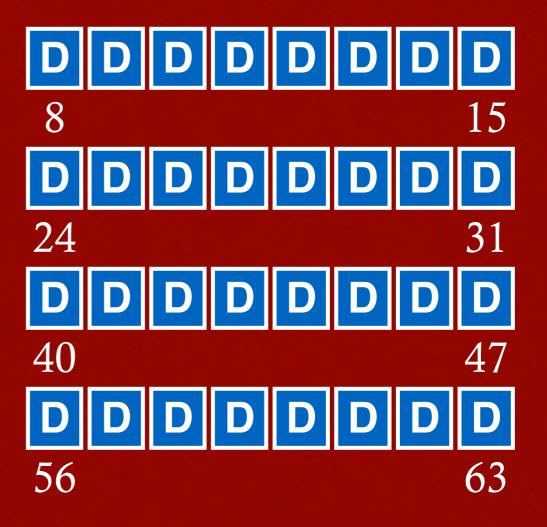
Need to know basic FS configuration metadata, like:

- block size
- # of inodes

Store this in superblock

SUPER BLOCK





ON-DISK STRUCTURES



directories indirects

Inode Bitmap

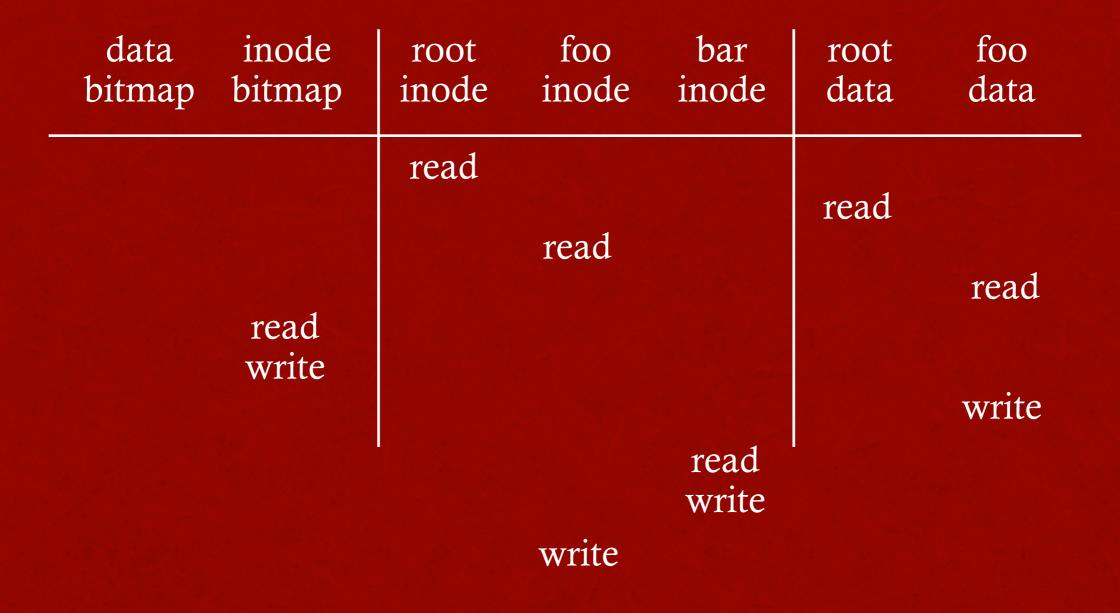
Data Bitmap

Inode Table

PART 2 : OPERATIONS

- create file
- write
- open
- read
- close

create /foo/bar



What needs to be read and written?

open /foo/bar

inode oitmap	root inode	foo inode	bar inode	root data	foo data	bar data
	read			read		
		read		ICau	read	
			read		Teau	

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 write				read			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			

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

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?