Announcements

P3b graded; P3a needs to be entered into Learn@UW

P4: Threads (Part a and b) available
- Still need partner?
- Due Friday 11/18 at 9pm – really Sunday 11/27 9pm

Exam 2 Graded: 80-83% average
- Solutions posted, midterm2.pdf file in your handin

Read as we go along!
- Chapter 40

UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

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?

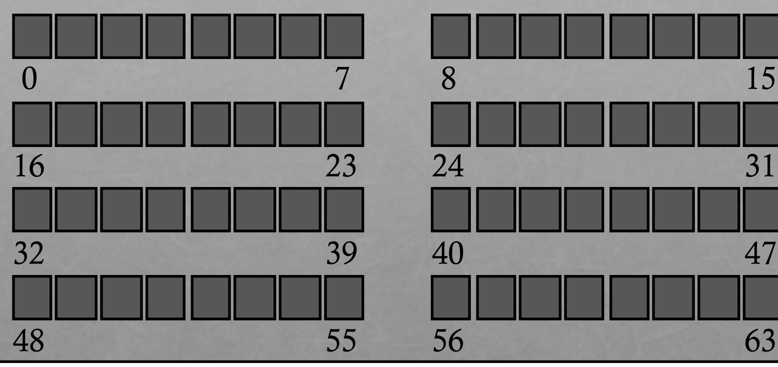
2. Access methods
   - what steps must reads/writes take?

Part 1: Disk Structures

Persistent Store

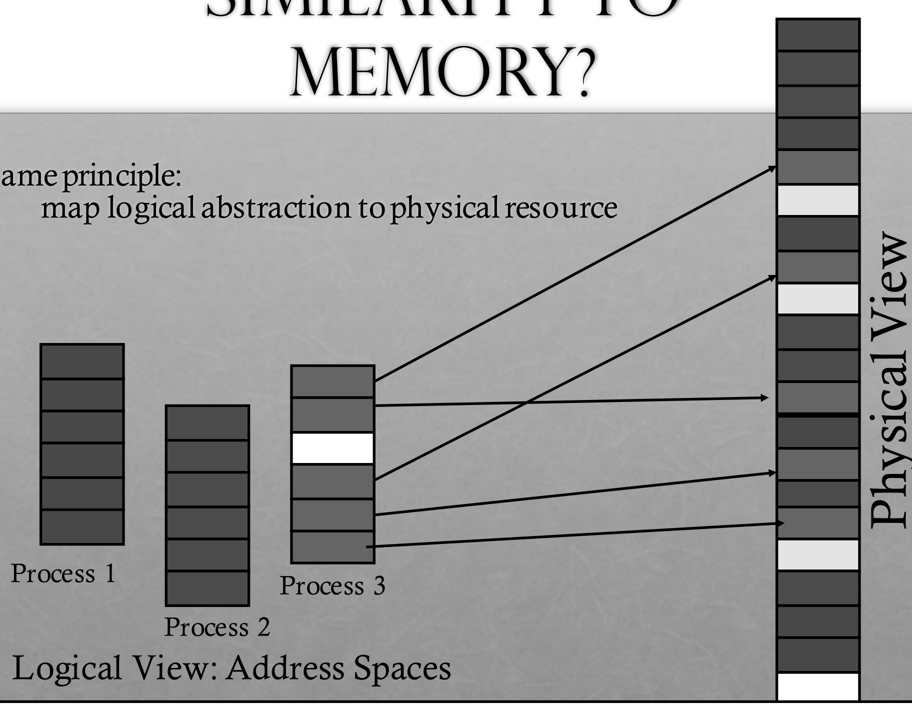
Given: large array of blocks on disk

Want: some structure to map files to disk blocks



Similarity to Memory?

Same principle:
map logical abstraction to physical resource



Logical View: Address Spaces

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

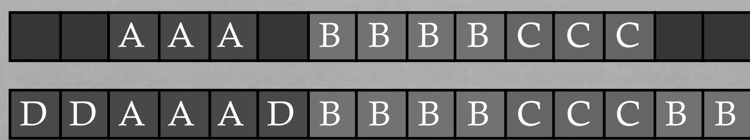
Speed to calculate random accesses? + Simple calculation

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 (generally)

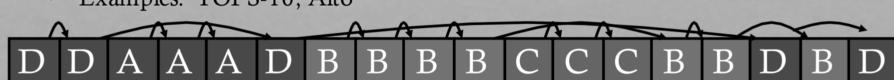
Speed to calculate random accesses? + Still simple calculation

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

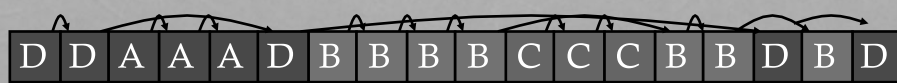
Wasted space for meta-data? - Waste pointer per block

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
- Supports random access

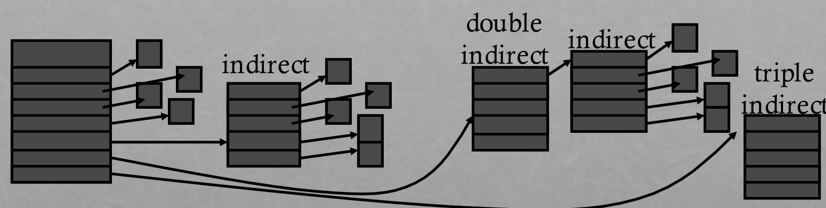
Disadvantages

- 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 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
  - 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

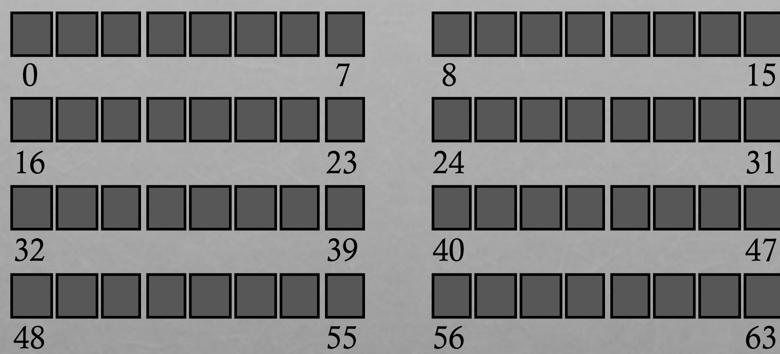
Simple approach

More complex file systems build from these basic data structures

On-Disk Structures

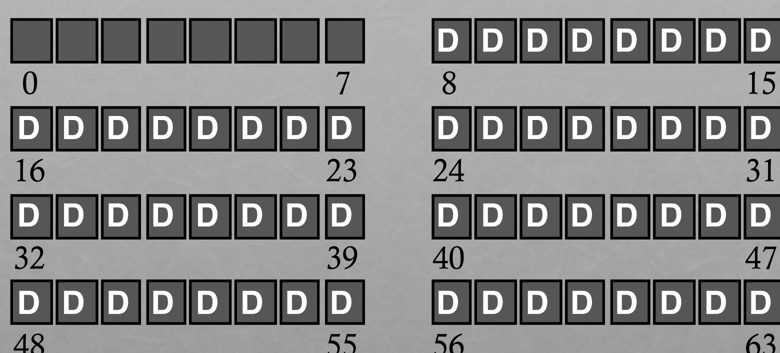
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

FS Structs: Empty Disk



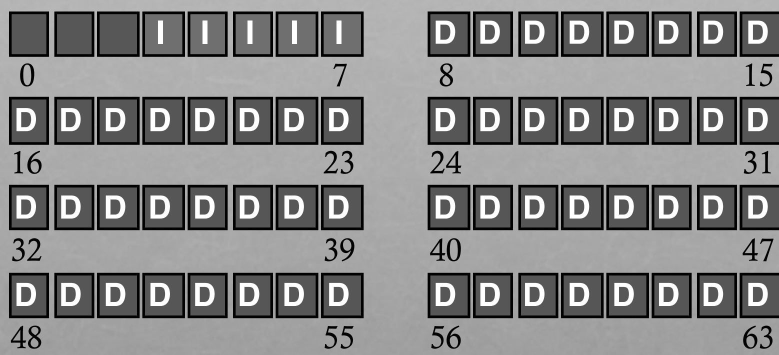
Assume each block is 4KB

Data Blocks



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

Inodes



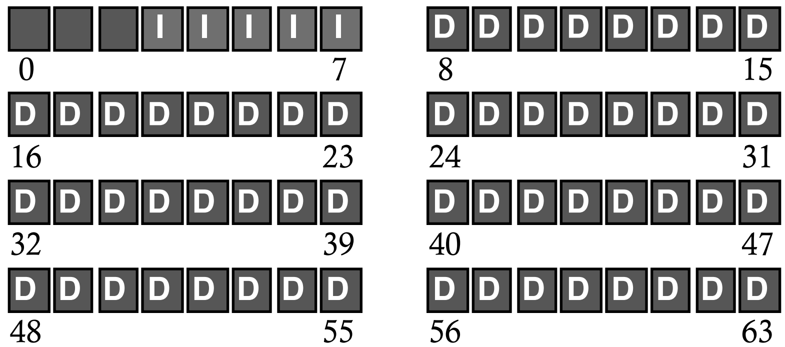
One Inode Block

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?

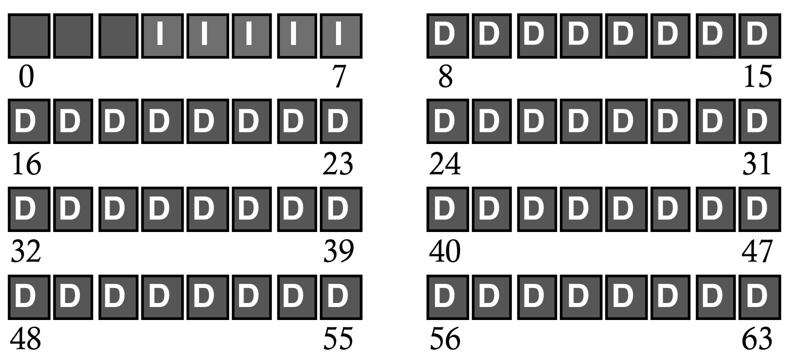
Static calculation to determine where particular inode resides on disk

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



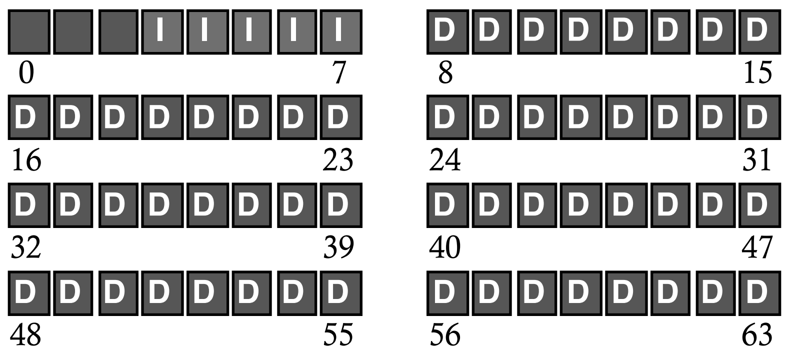
Block: inode start + 0/16 = 3 + 0 = 3
Offset within block: 0 % 16 * 256 = 0

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



Block: inode start + 4/16 = 3 + 0
Offset within block: 4 % 16 * 256 = 4 * 256

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

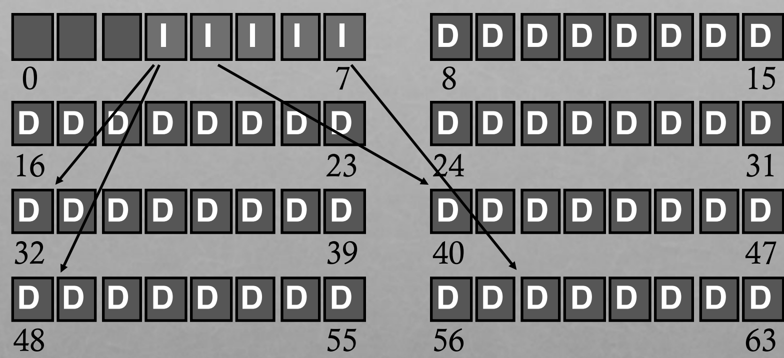


Block: inode start + 40/16 = 3 + 2 = 5
Offset within block: 40 % 16 * 256 = 8 * 256

INODE

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

INODES



INODE: ATTEMPT 1 (SINGLE LEVEL)

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

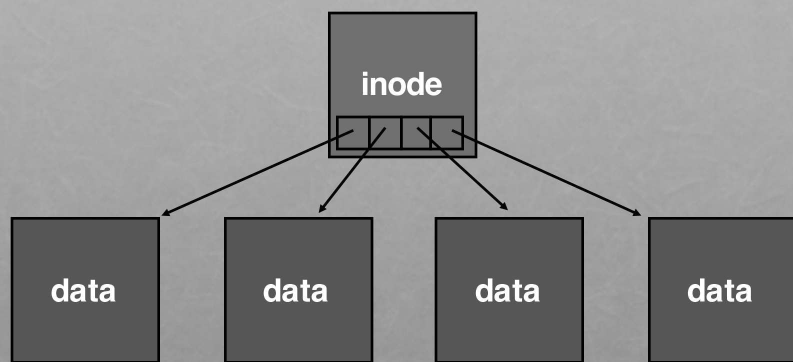
Assume single level (just pointers to data blocks)

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

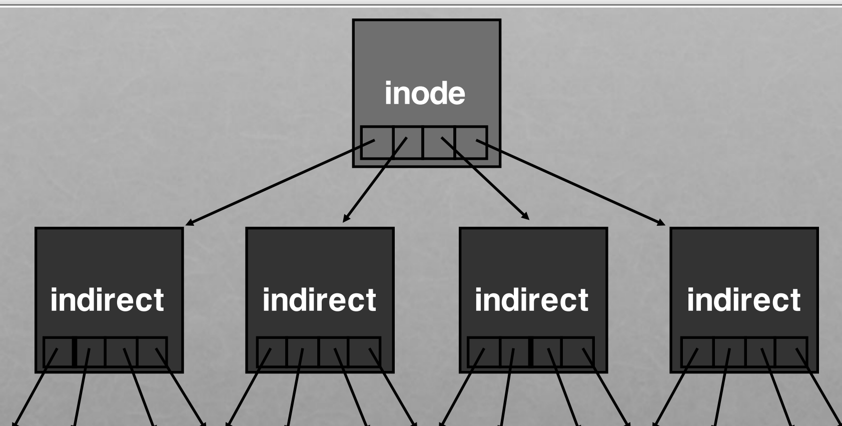
256 / 4 = 64 pointers
64 * 4K = 256 KB!

How to get larger files?

SINGLE LEVEL



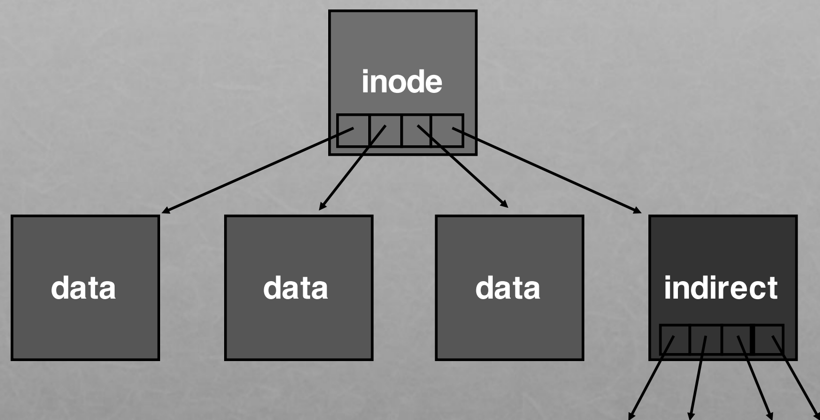
BALANCED TREE: ATTEMPT 2



Indirect blocks are stored in regular data blocks

what if we want to optimize for small files?

Imbalanced Tree: FFS Solution



Better for small files

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

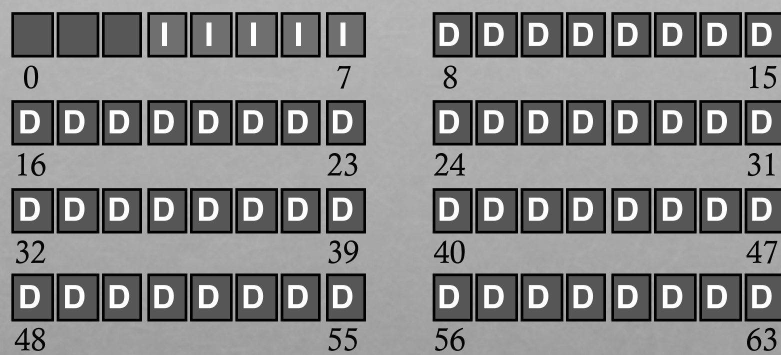
How do we find free data blocks or free inodes?

Free list

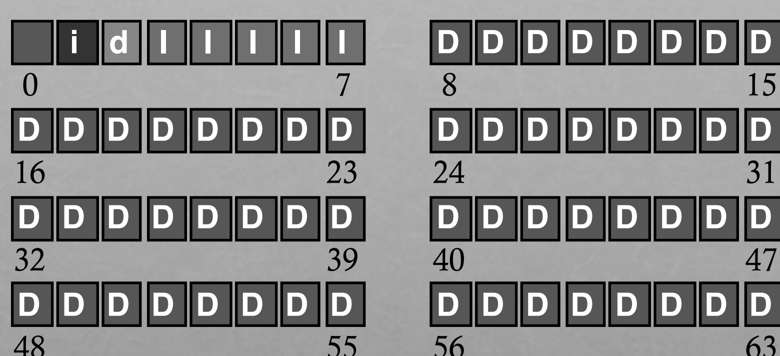
Bitmaps

Tradeoffs in next lecture…

Bitmaps?



Opportunity for Inconsistency (fsck)



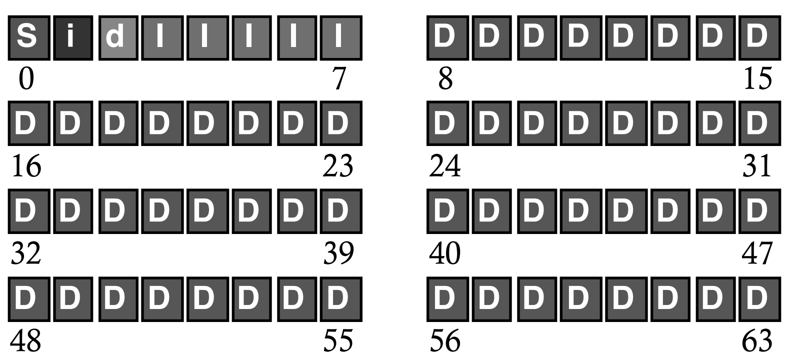
Superblock

Need to know basic FS configuration metadata, like:

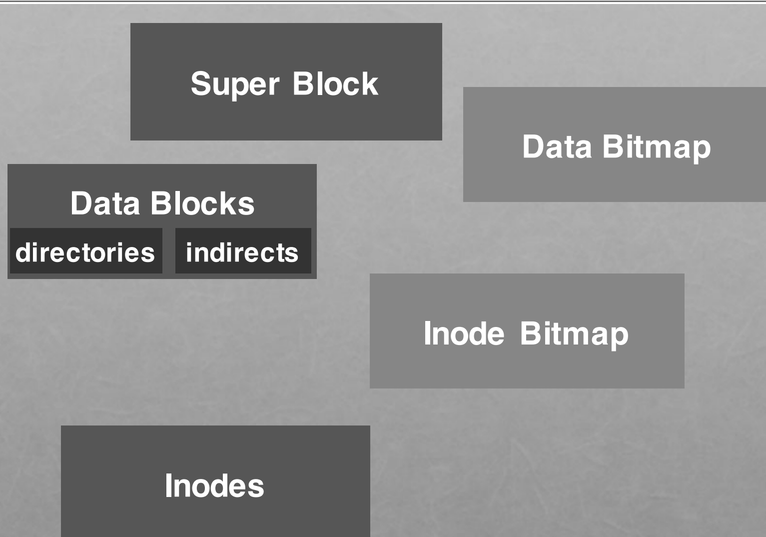
- block size
- # of inodes

Store this in superblock

Super Block



ON-DISK STRUCTURES



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?

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			

Update data bitmap to show allocated data blocks

Update bar inode with new data pointers and file size

read /foo/bar – assume opened

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

Update timestamps in bar inode

close /foo/bar

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

nothing to do on disk!

ATOMIC FILE UPDATE

Say application wants to update `file.txt` atomically

- If crash, should see only old contents or only new contents

1. write new data to `file.txt.tmp` file
2. `fsync file.txt.tmp`
3. `rename file.txt.tmp` over `file.txt`, replacing it

Rename

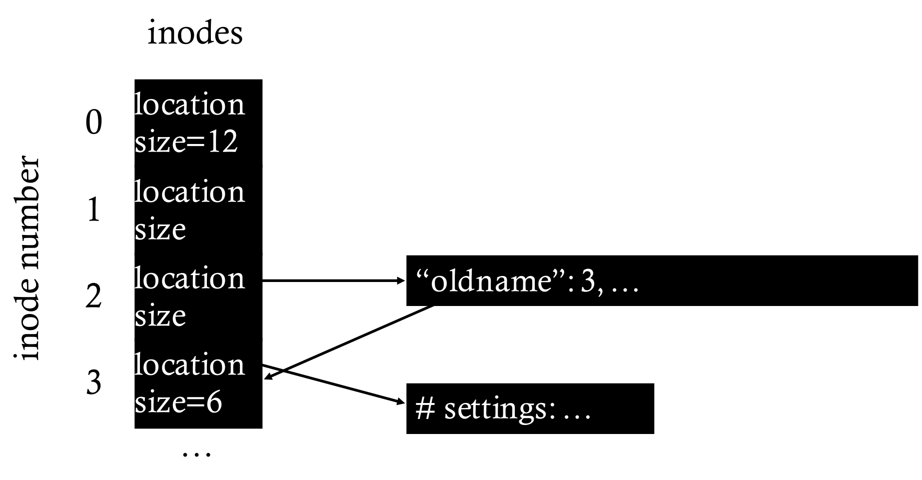
rename (char *old, char *new):

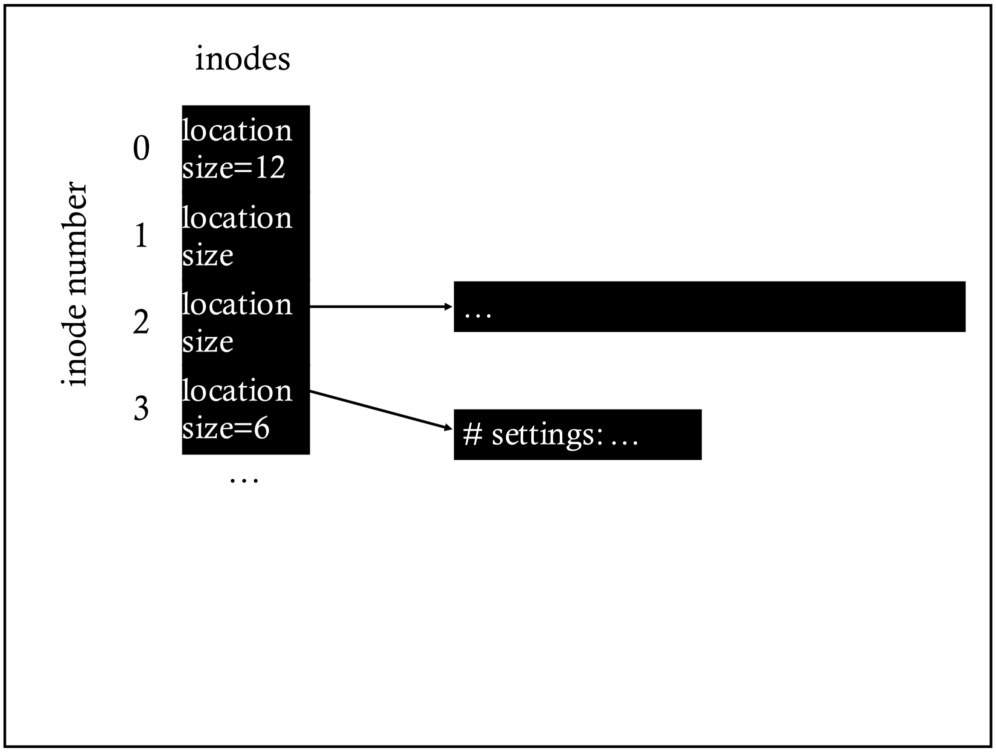
- deletes an old link to a file
- creates a new link to a file

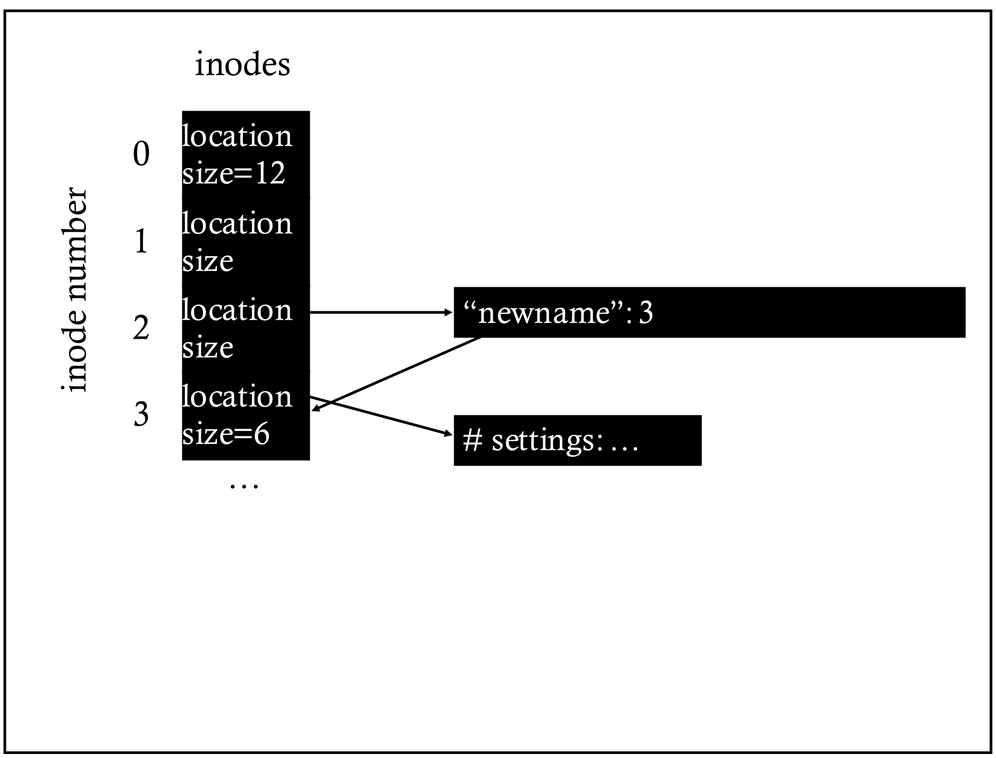
Just changes name of file, does not move data

Even when renaming to new directory (unless…?)

What can go wrong if system crashes at wrong time?




inodes
inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6
…
…
settings: …


inodes
inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6
…
“newname”: 3
settings: …

Rename

rename(char *old, char *new):

- deletes an old link to a file
- creates a new link to a file

What if we crash?

FS does extra work to guarantee atomicity; return to this issue later…

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?

Communicating Requirements: fsync

File system keeps newly written data in memory for awhile

Write buffering improves performance (why?)

But what if system **crashes** before buffers are flushed?

If application cares:

`fsync(int fd)` forces buffers to flush to disk, and (usually) tells disk to flush its write cache too

Makes data **durable**