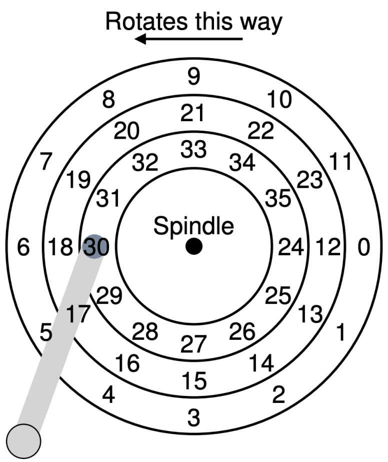
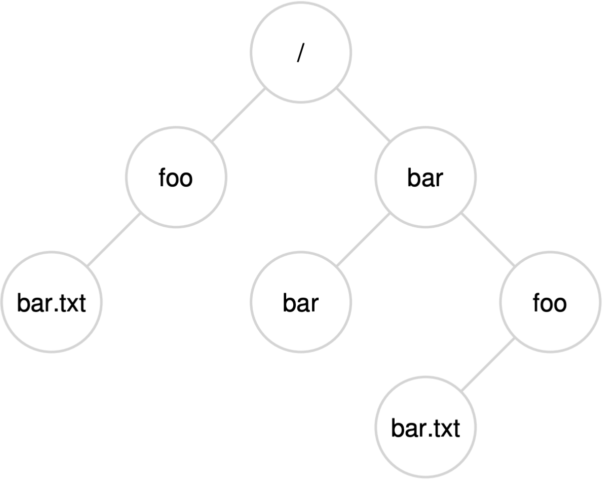
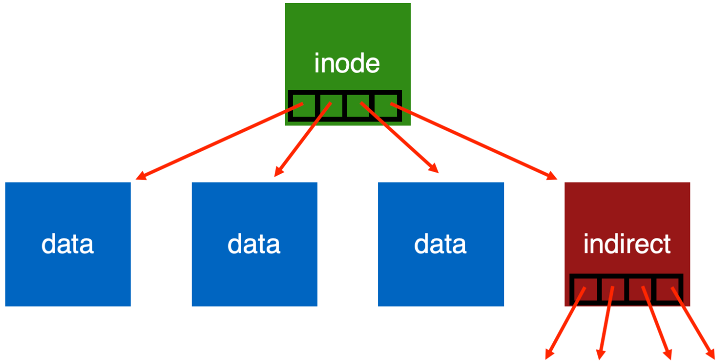
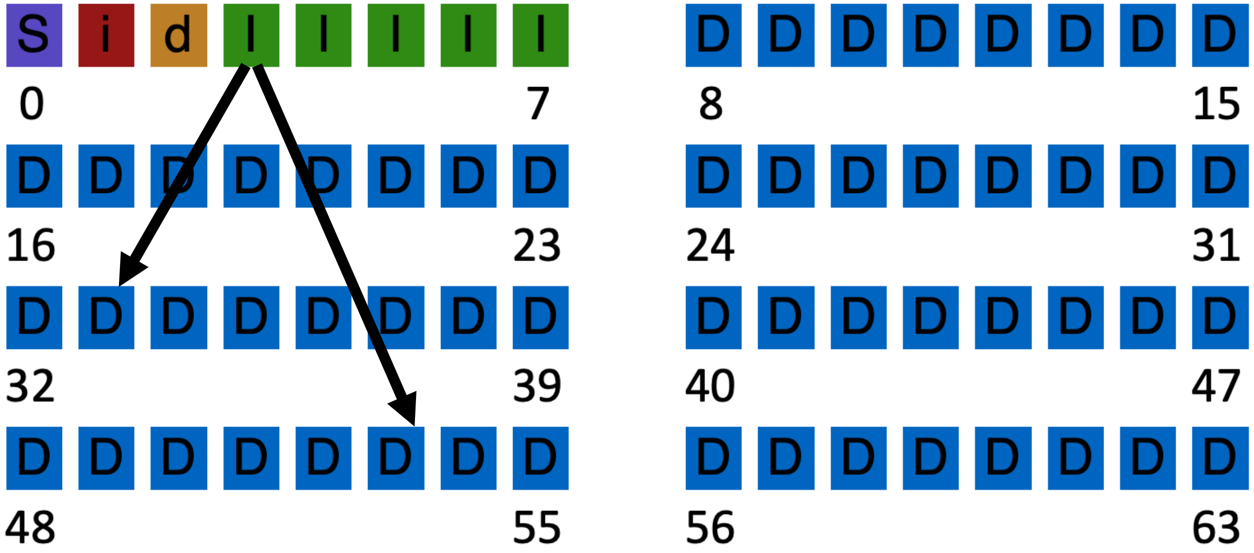
Persistence





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





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			

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…

- tradeoff durability vs. performance

How to allocate file data to disk blocks?

Disk layout of data matters!

- Why?
- Positioning latency: disk rotation; seek
- Sequential reads are faster than random reads

Allocation Strategies

Many different approaches

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

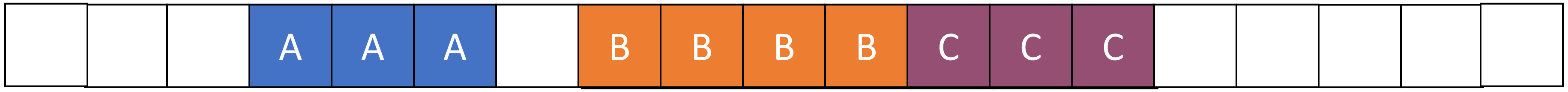
Questions

- Amount of fragmentation (internal and external)
  – free space 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

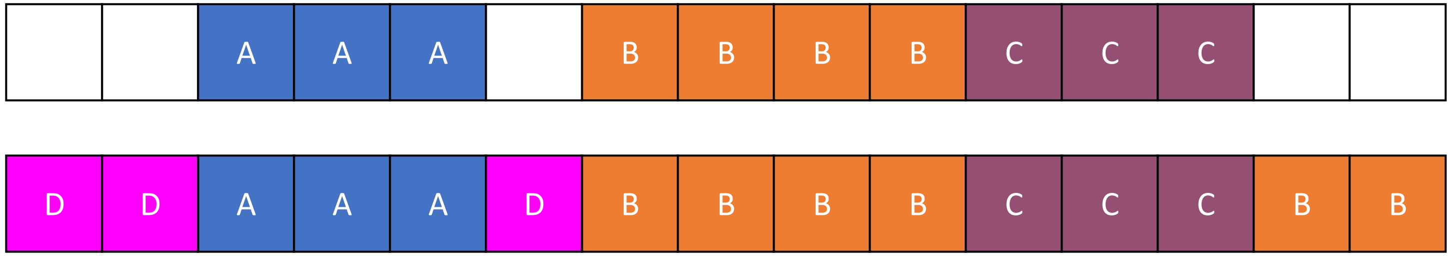


Fragmentation (internal and external)?	- Horrible external frag (needs periodic compaction)
Ability to grow file over time?	- May not be able to without moving
Seek cost for sequential accesses?	+ Excellent performance
Speed to calculate random accesses?	+ Simple calculation
Wasted space for meta-data?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

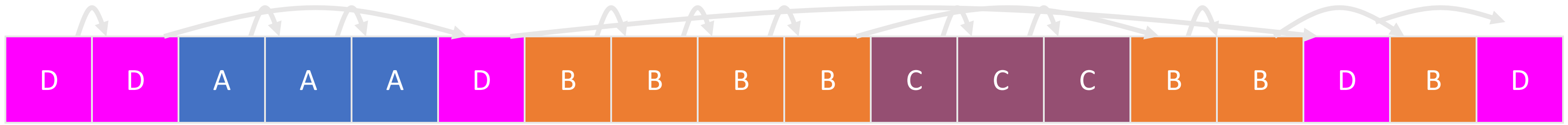
Speed to calculate random accesses? + Still simple calculation

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);

Ability to grow file over time? + Can grow easily

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

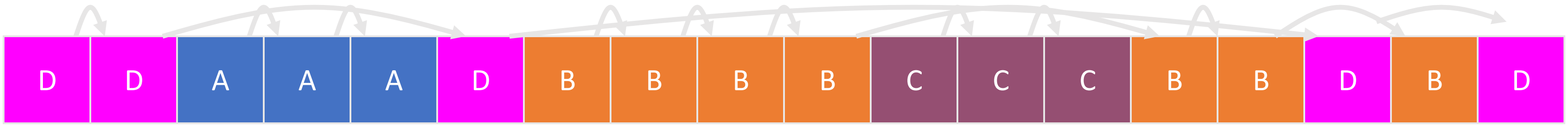
Speed to calculate random accesses? - Ridiculously poor

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

File-Allocation Table (FAT)

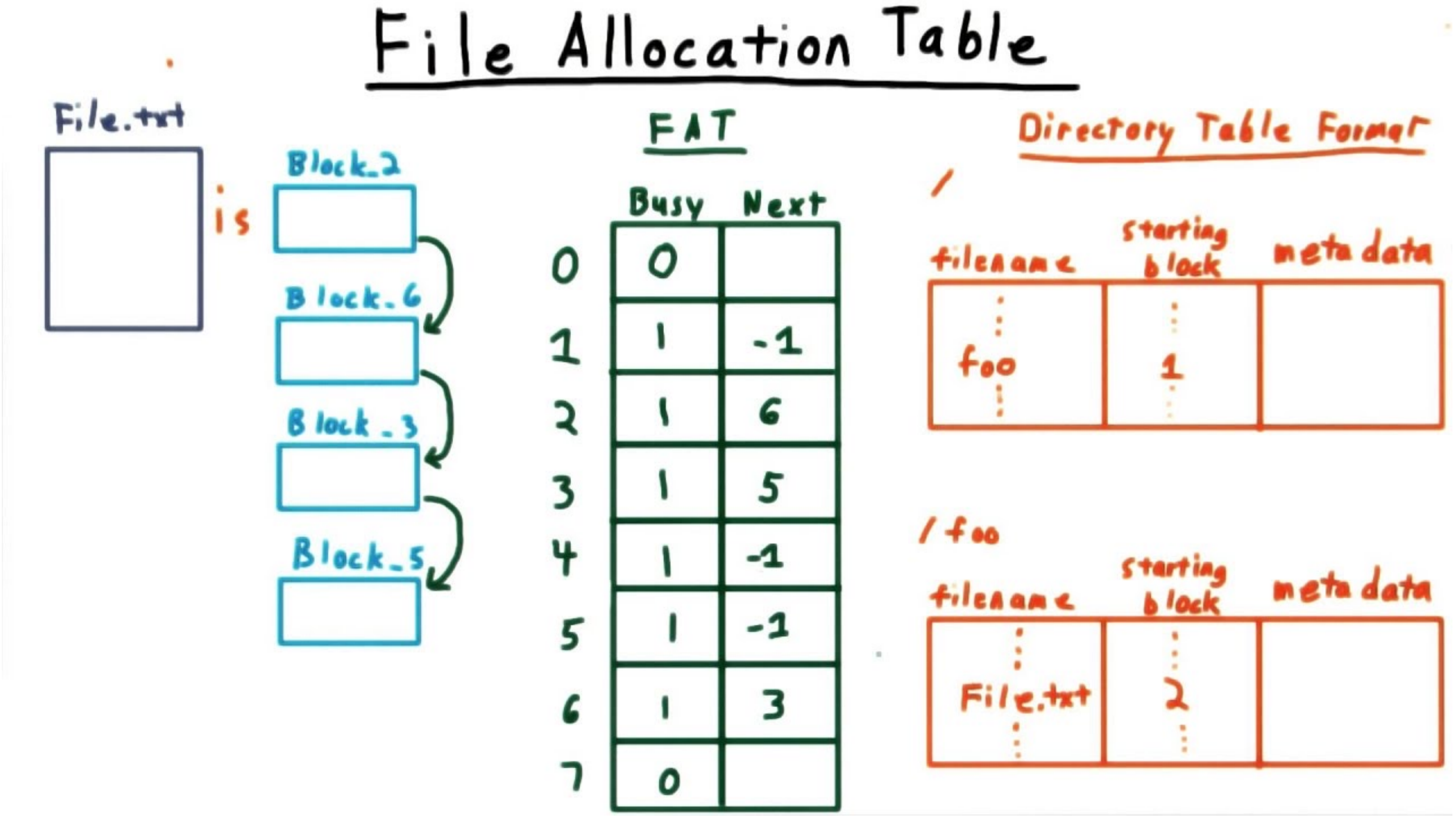
Variation of Linked allocation

- 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?

Example of a FAT

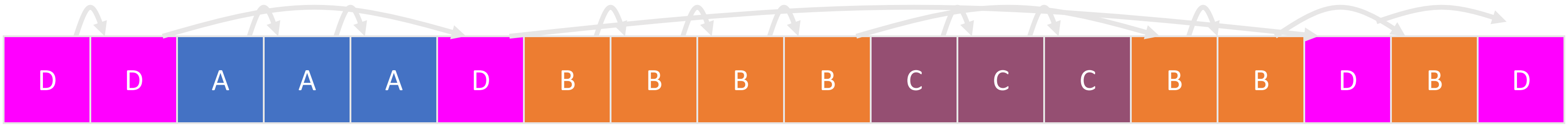


https://www.youtube.com/watch?v=mgQtlXBxH0c

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?

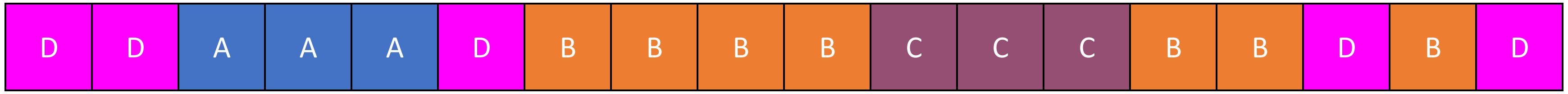
Comparison to Linked Allocation

- 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 pointers at file creation time



Advantages

- No external fragmentation
- Files can be easily grown up to max file size
- Supports random access

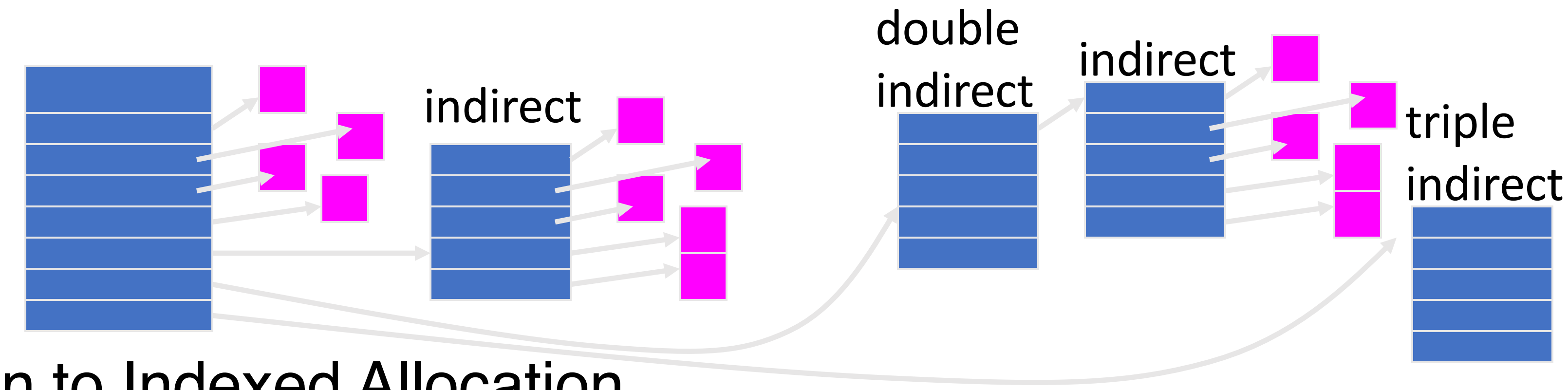
Disadvantages

- 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

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

Summary/Future

We’ve described a very simple FS.

- basic on-disk structures
- the basic ops

Future questions:

- how to handle **crashes**?

Crash Consistency

Questions answered:

What benefits and complexities exist because of data **redundancy?**

What can go wrong if disk blocks are not updated consistently?

How can file system be **checked and fixed** after crash?

How can **journaling** be used to obtain **atomic updates**?

How can the **performance** of journaling be improved?

Data Redundancy

Definition:

if *A* and *B* are two pieces of data,
and knowing *A* eliminates some or all values *B* could be,
there is redundancy between *A* and *B*

File system examples:

- **Superblock**: field contains total blocks in FS
- **Inodes**: field contains pointer to data block
- Is there redundancy between these two types of fields? Why or why not?

File System Redundancy Example

Superblock: field contains total number of blocks in FS

DATA = N

Inode: field contains pointer to data block; possible DATA?

DATA in {0, 1, 2, …, N - 1}

Pointers to block N or after are invalid!

Total-blocks field has redundancy with inode pointers

Pros and CONs of Redundancy

Redundancy may improve:

- reliability
  - Superblocks in FFS
- performance
  - bitmaps

But Redundancy could hurt!

- capacity
- consistency
  - Redundancy implies certain combinations of values are (possibly) illegal
  - Illegal combinations: inconsistency

Consistency Examples

Assumptions:

Superblock: field contains total blocks in FS.
DATA = 1024
Inode: field contains pointer to data block.
DATA in {0, 1, 2, …, 1023}

Scenario 1: Consistent or not?

Superblock: field contains total blocks in FS.
DATA = 1024
Inode: field contains pointer to data block.
DATA = 241

Scenario 2: Consistent or not?

Superblock: field contains total blocks in FS.
DATA = 1024
node: field contains pointer to data block.
DATA = 2345

Why is consistency challenging?

File system may perform several disk writes to redundant blocks

If file system is interrupted between writes, may leave data in inconsistent state

What can interrupt write operations?

- power loss
- kernel panic
- reboot

Bad things that can happen: inconsistency, garbage data, data loss,

Question for You…

File system is appending to a file and must update:

- inode
- data bitmap
- data block

What happens if crash after only updating some blocks?

a) **bitmap**: lost block & data

b) **data**: Data loss, but otherwise OK

c) **inode**: point to garbage (what?), **another file may use**

d) **bitmap** and **data**: lost block & data (nothing can reach it)

e) **bitmap** and **inode**: point to garbage

f) **data** and **inode**: **another file may use (from bitmap)**

How can file system fix Inconsistencies?

Solution #1:

- FSCK = file system checker

Strategy:

- After crash, scan whole disk for contradictions and “fix” if needed
- Keep file system off-line until FSCK completes

For example, how to tell if data bitmap block is consistent?

Read every valid inode+indirect block
If pointer to data block, the corresponding bit should be 1; else bit is 0

Fsck Checks

Hundreds of types of checks over different fields…

Do superblocks match?

Do directories contain “.” and “..”?

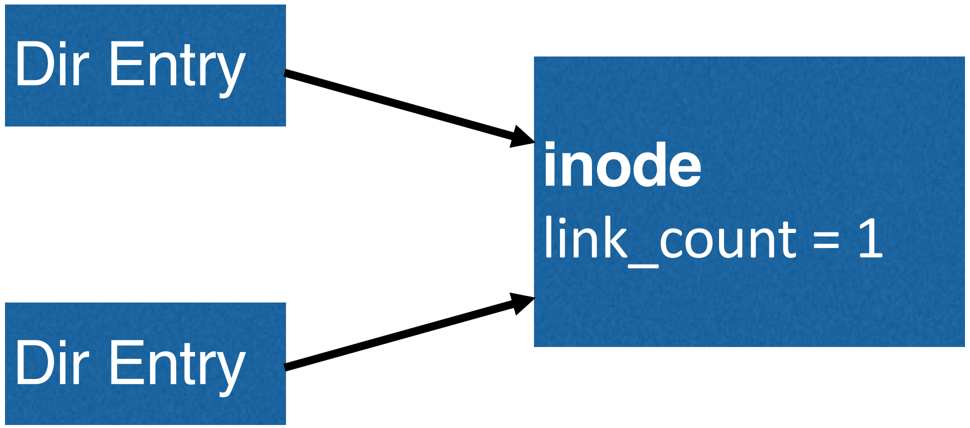
Do number of dir entries equal **inode link counts**?

Do different inodes ever point to **same block**?

…

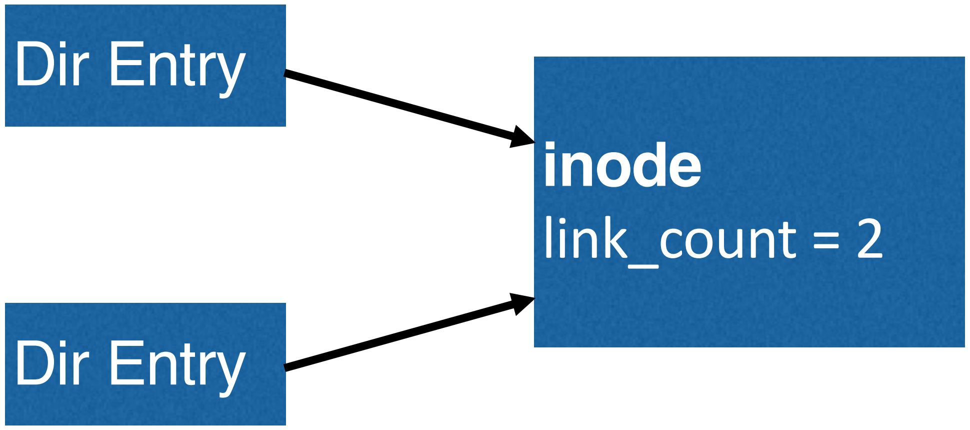
How to solve problems?

Link Count (example 1)



How to fix to have consistent file system?

Link Count (example 1)



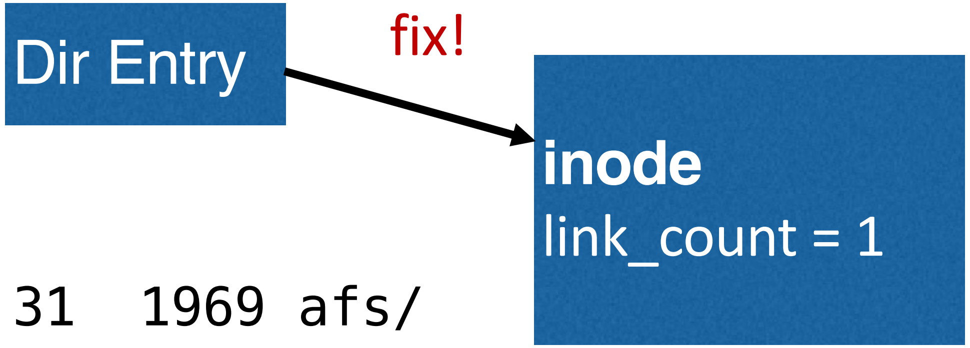
Simple fix!

Link Count (example 2)

inode
link_count = 1

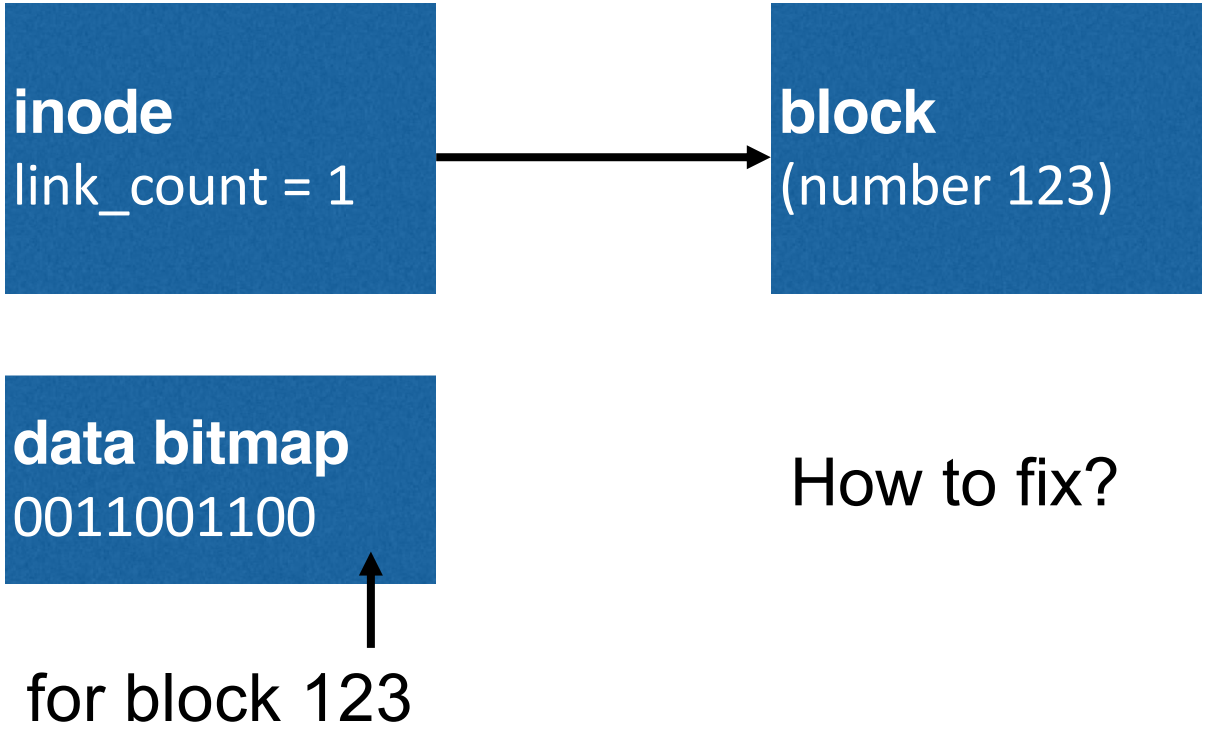
How to fix???

Link Count (example 2)



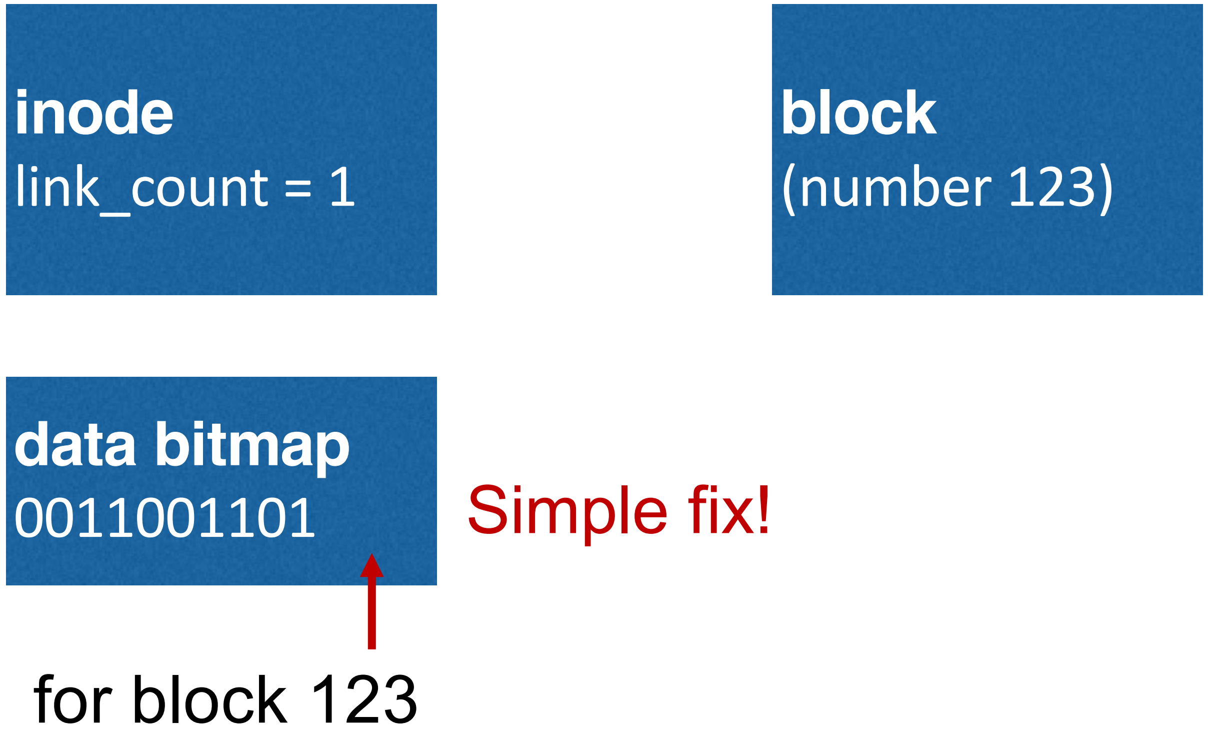

```
ls -l /
total 150
drwxr-xr-x  401 18432 Dec 31  1969 afs/
drwxr-xr-x.   2 4096  Nov  3 09:42 bin/
drwxr-xr-x.   5 4096  Aug  1 14:21 boot/
dr-xr-xr-x.  13 4096  Nov  3 09:41 lib/
dr-xr-xr-x.  10 12288 Nov  3 09:41 lib64/
drwx------.   2 16384 Aug  1 10:57 lost+found/
...
```

Data Bitmap



How to fix?

Data Bitmap


inode
link_count = 1
block
(number 123)
data bitmap
0011001101
Simple fix!
for block 123