

Crash Consistency

CS333

Williams College

This Video

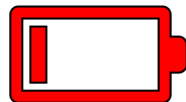
- Challenges of Maintaining FS Consistency
 - Redundancy & Invariants
- Two techniques for Maintaining Consistency
 - File system checkers (fsck)
 - Logging (journaling)

Redundancy is Everywhere




- Use explicit redundancy to safeguard important data
 - E.g., RAID (mirroring and parity), duplicating superblock
- Also have implicit redundancy in FS data structures
 - E.g., Block allocation information:
 - Inodes & indirect blocks store pointers to valid data blocks, implicitly representing allocation status
 - Bitmaps store allocated/free status explicitly
 - E.g., Link counts:
 - Inode stores link count
 - Can traverse directories and count entries
 - ...

Downsides to Redundancy

- Wasted space
 - Although we're often happy to trade space for performance or safety, there are limits
- Difficulty of maintaining invariants
 - Disk guarantees that we can write individual sectors atomically
 - If redundant/dependent information spans more than one sector, a crash in the middle of some logical update can leave our system in an inconsistent state



Implications of Crashes

- Think about the data structures that are updated when we append data to a file
 1. **Data bitmap** to allocate a new data block
 2. The file's **inode** to update block pointer/size/etc.
 3. The new **data block** itself
- Suppose we crash after any step above
 - After **data bitmap**: unreachable “allocated block” 
 - After writing **inode**: pointer to uninitialized data 
 - After writing **data**: Nothing. It's ok to crash here. 

Strategies to Avoid Inconsistency

- Ignore the issue until a crash occurs, then fix any issues that might have arisen
 - This is the approach that **file system checkers** like **fsck** (commonly pronounced “eff-sick”, “eff-suck”, “eff-ess-see-kay”) take
- Do some extra bookkeeping before making changes to our system so that if we crash in the middle, we can refer to our notes and complete the operation
 - This is the approach that **logging (journaling)** takes

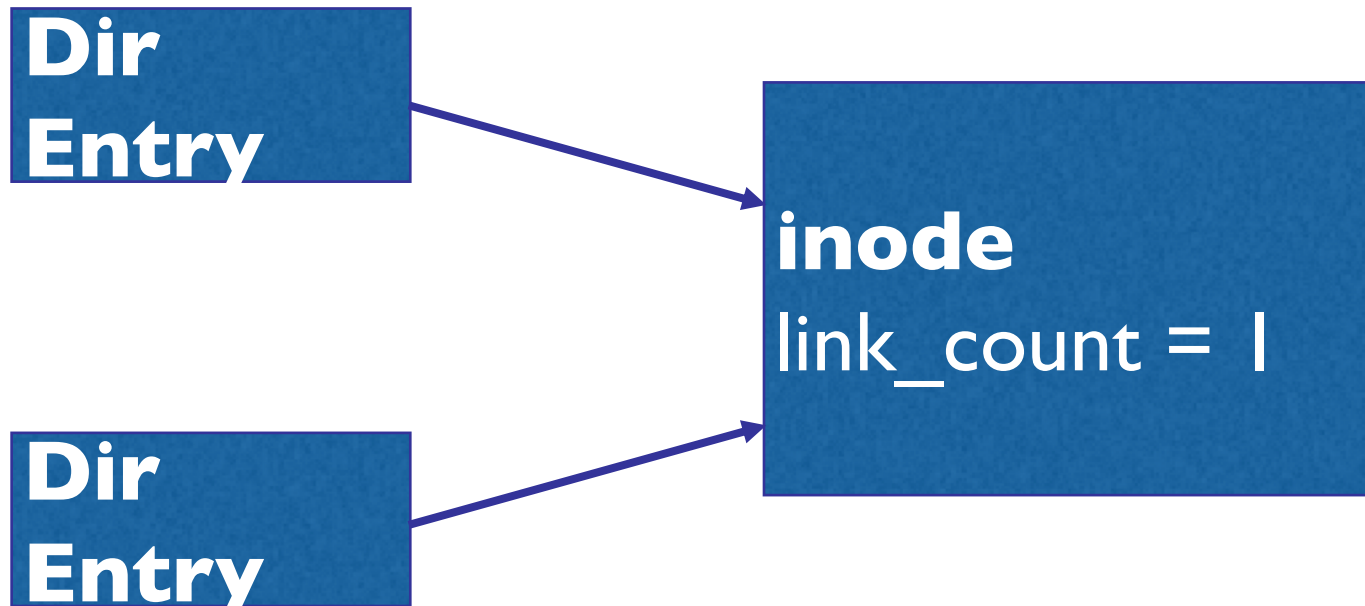
Consistency Strategy I:

FILE SYSTEM CHECKERS

File System Checkers

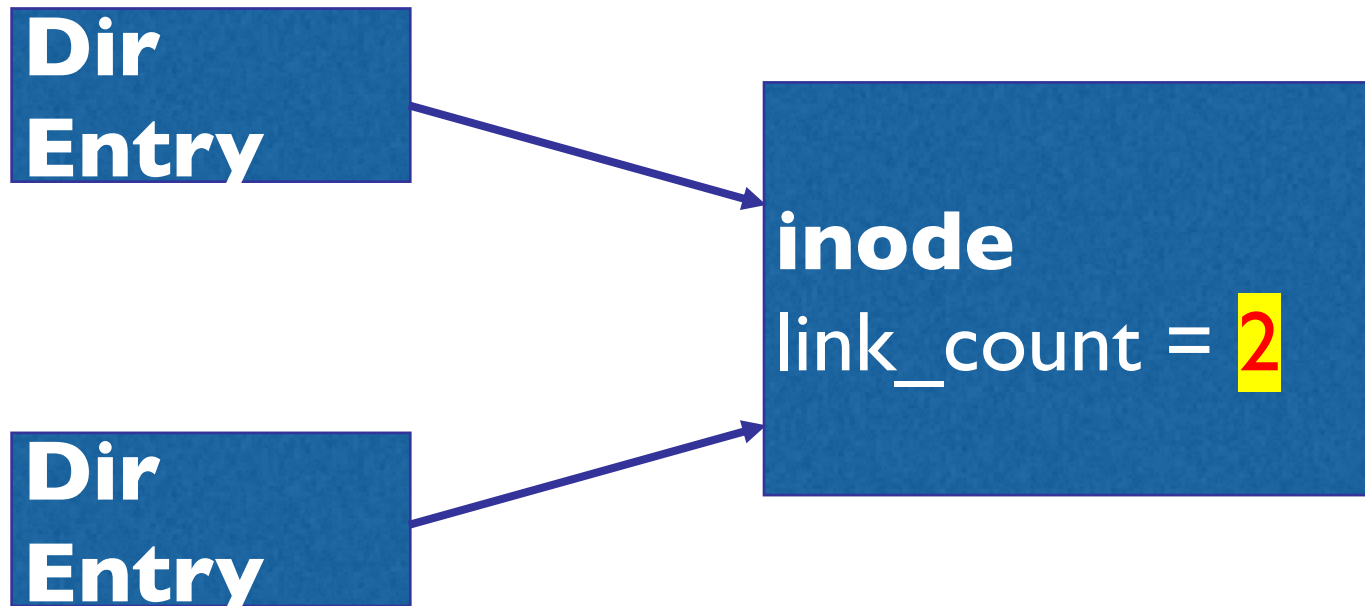
- Scan the file system metadata and check that all invariants are held
 - Pause the system so no-one makes any changes during our scan
 - If we detect any inconsistencies, fix them (as best as we can)

Example Invariant Check: Incorrect Link Count



How do we fix this to make our file system consistent?

Example Invariant Check: Incorrect Link Count

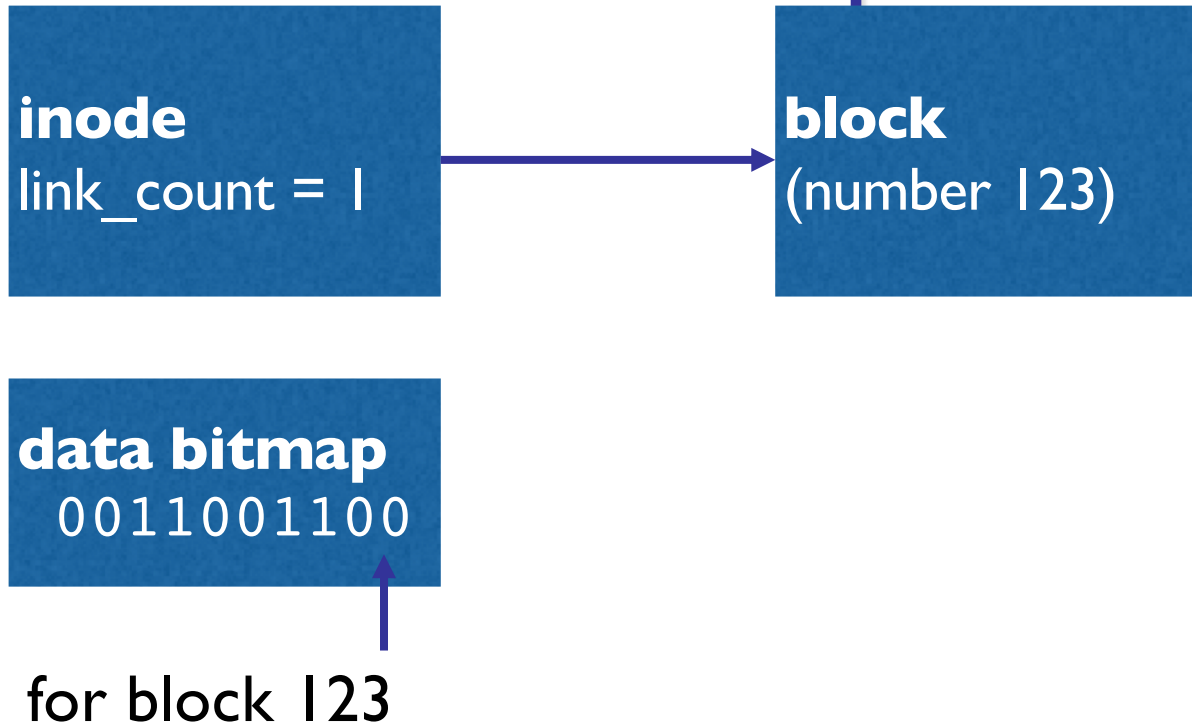


How do we fix this to make our file system consistent?

We count all directories that refer to each file and update the link count.

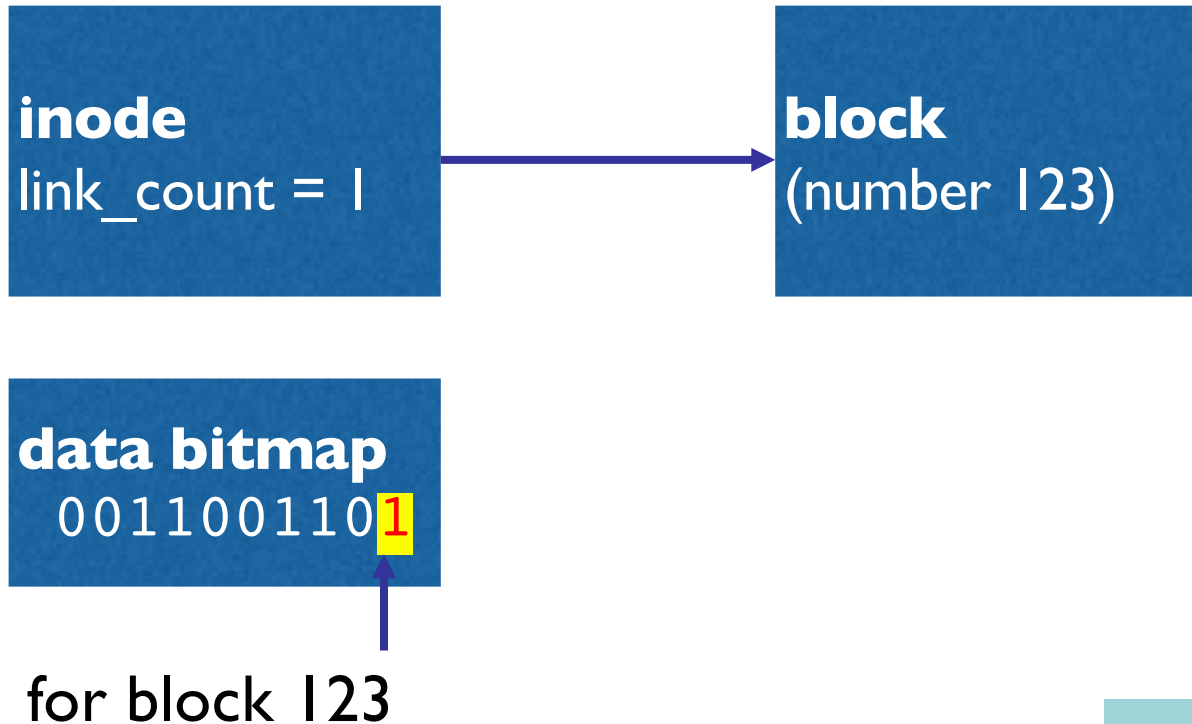
Is this correct?

Example Invariant Check: Data Bitmap



How do we fix this to make our file system consistent?

Example Invariant Check: Data Bitmap

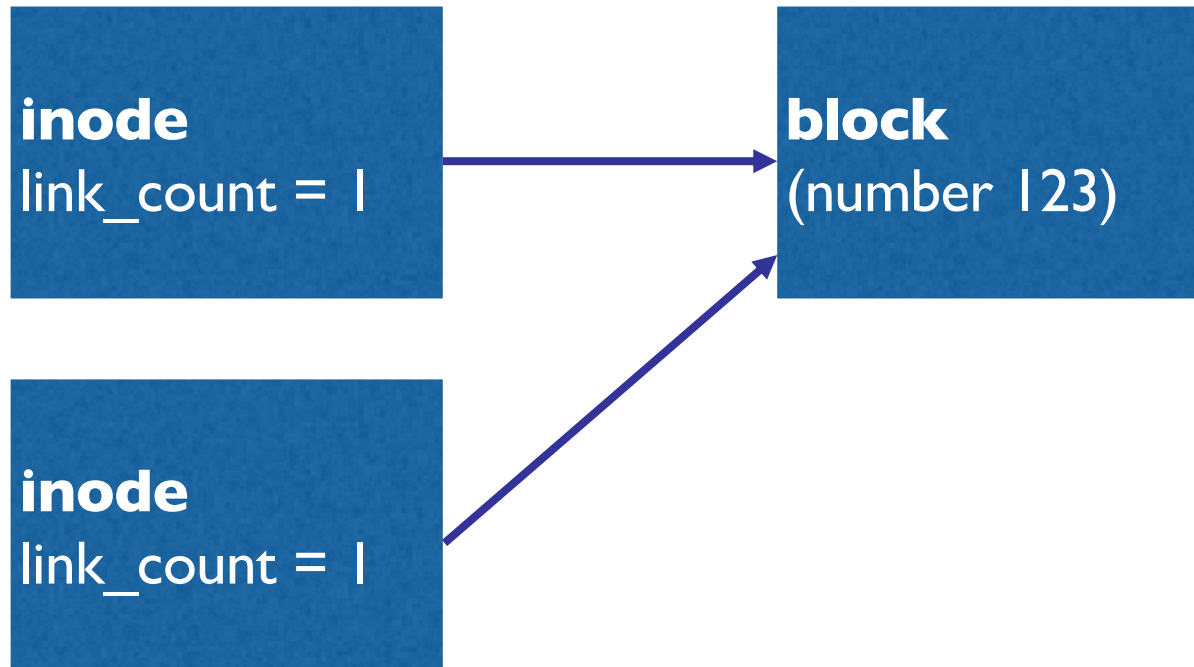


Is this correct?

How do we fix this to make our file system consistent?

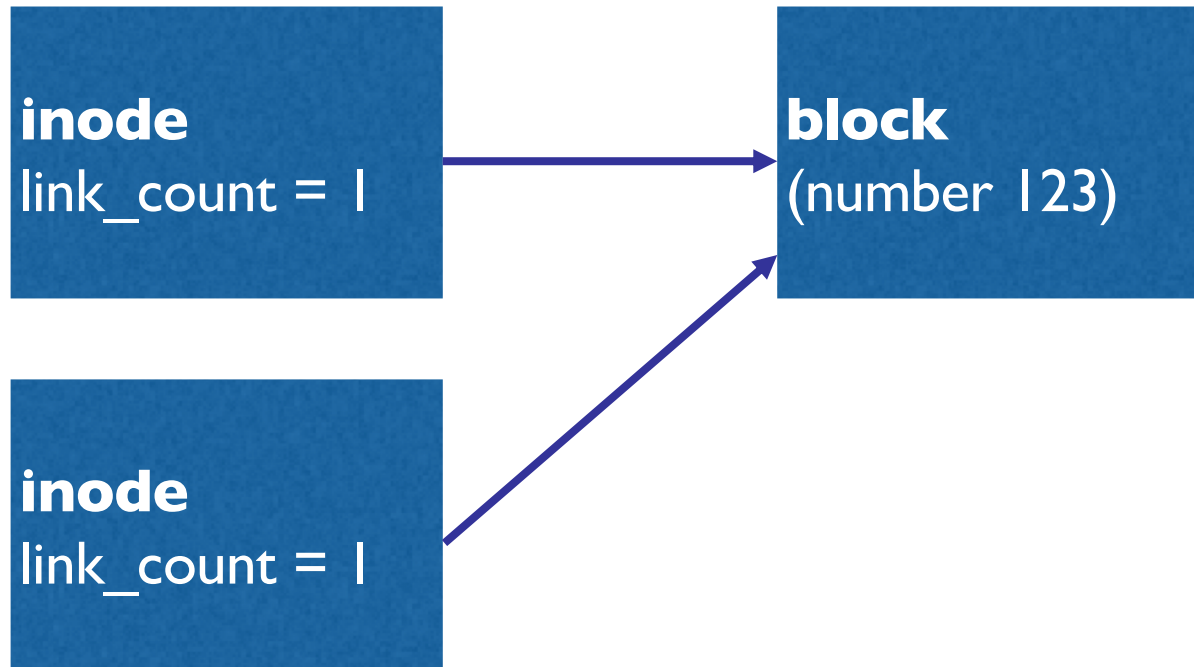
We scan all inode direct and indirect pointers and keep track of which blocks are allocated. Then update our bitmaps.

Example Invariant Check: Duplicate Pointers



How do we fix this to make our file system consistent?

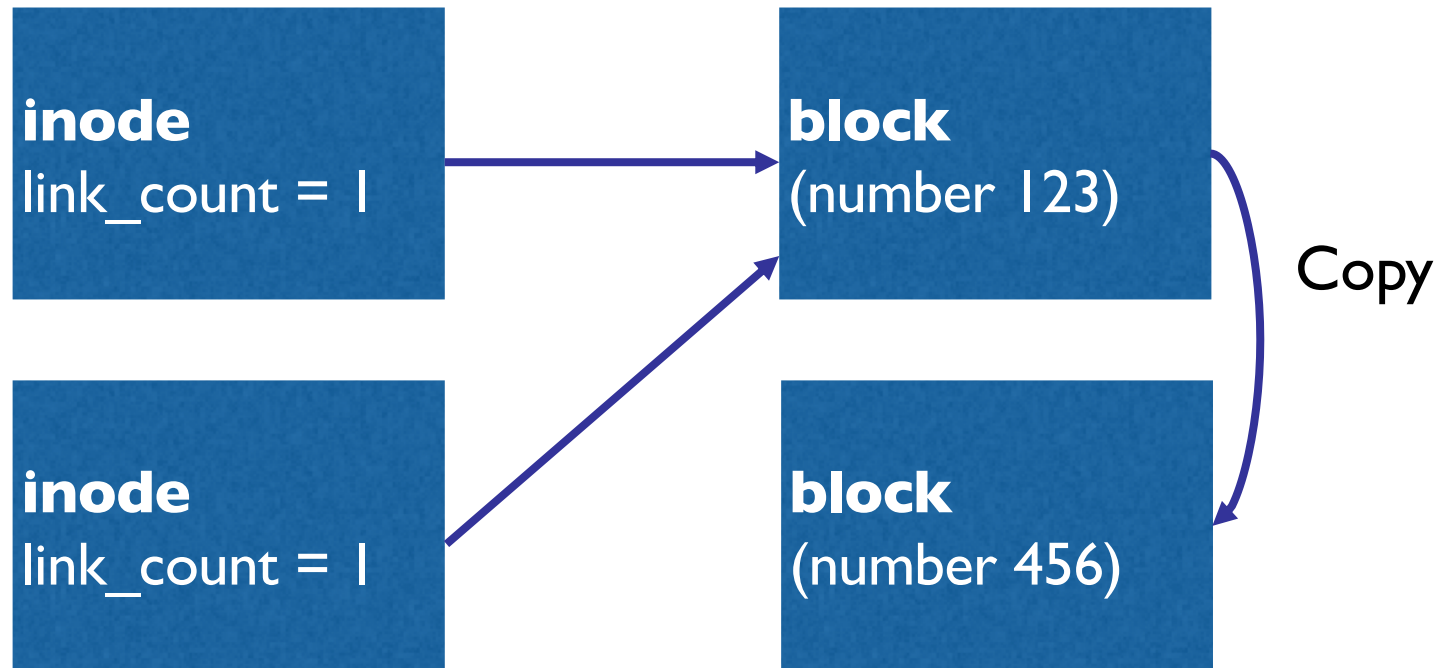
Example Invariant Check: Duplicate Pointers



How do we fix this to make our file system consistent?

Two files can't share the same block, so we copy the block and have each inode point to a copy.

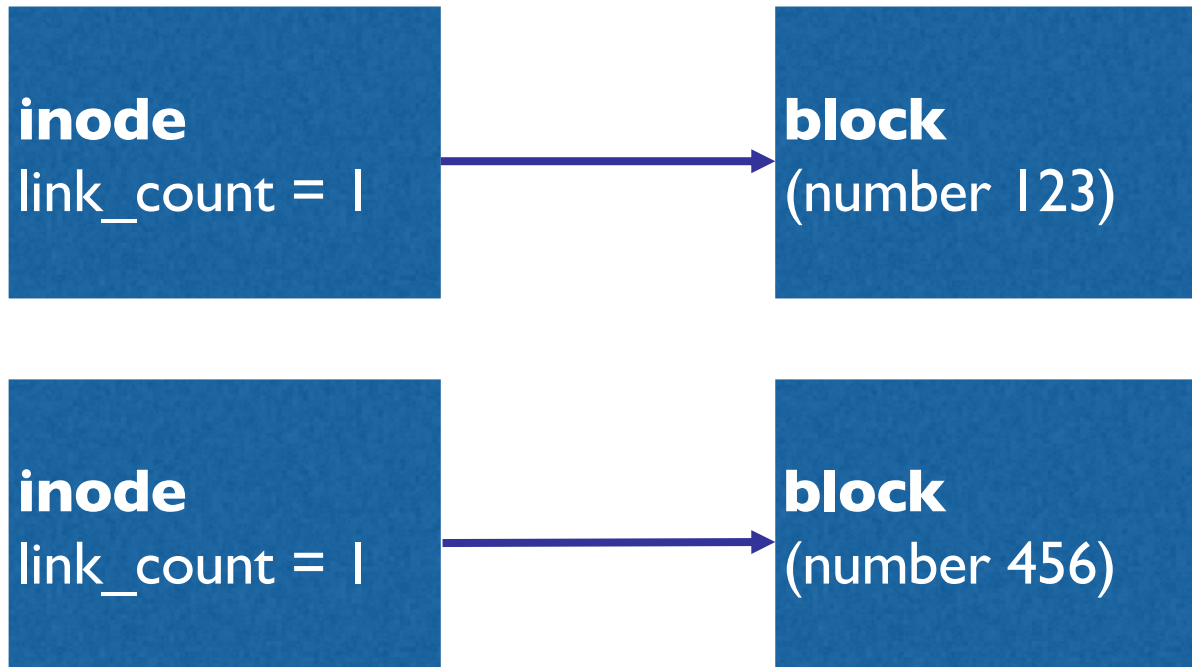
Example Invariant Check: Duplicate Pointers



How do we fix this to make our file system consistent?

Two files can't share the same block, so we copy the block and have each inode point to a copy.

Example Invariant Check: Duplicate Pointers

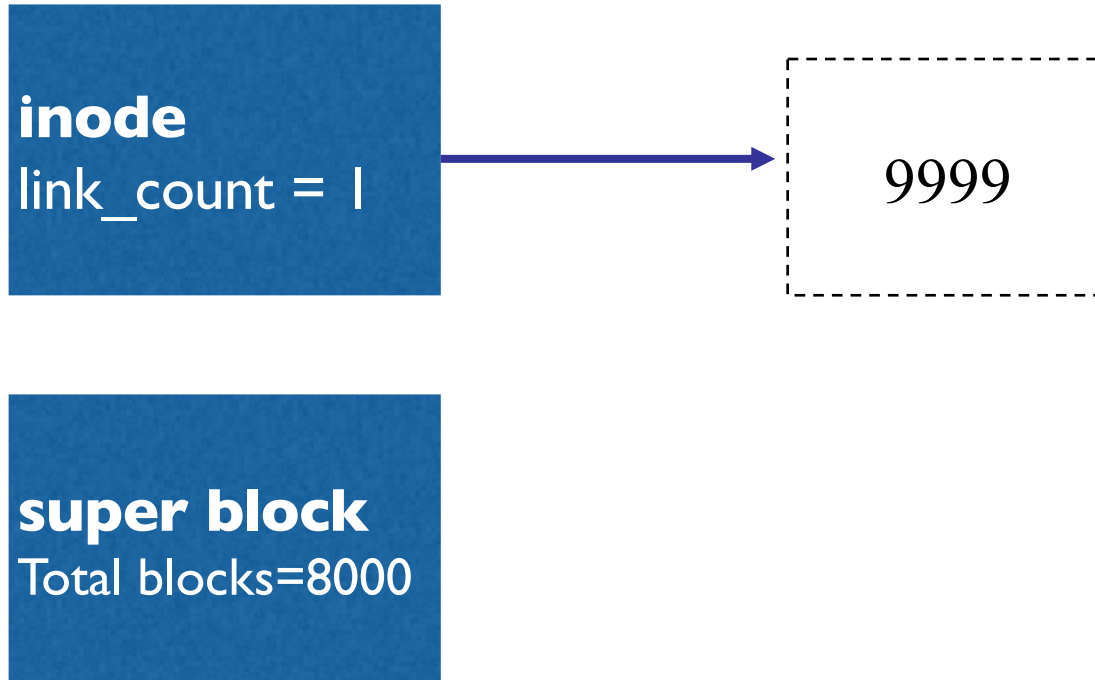


How do we fix this to make our file system consistent?

Two files can't share the same block, so we copy the block and have each inode point to a copy.

Is this correct?

Example Invariant Check: Bad Pointer



How do we fix this to make our file system consistent?

Example Invariant Check: Bad Pointer

inode
link_count = 1

super block
Total blocks=8000

How do we fix this to make our file system consistent?

We update the inode to remove the pointer.

Is this correct?

FSCK problems

- It's not always obvious how to fix a problem
 - We may become consistent, but we may lose data
 - Strawman fix: erase our FS
- FSCK is slow
 - Traversing our data structures scales with our file system size
 - We can't make progress while FSCK is running

Consistency Strategy 2:

LOGGING (JOURNALING)

Logging Strategy

When the system receives an update request

1. “Make notes” about the required changes
 2. Perform the update
 3. Delete the (now redundant) notes
- If the system crashes between steps 1 and 2, we can consult our notes and recover
 - We are always either left in a consistent state or left with the information we need to restore our system to a consistent state.

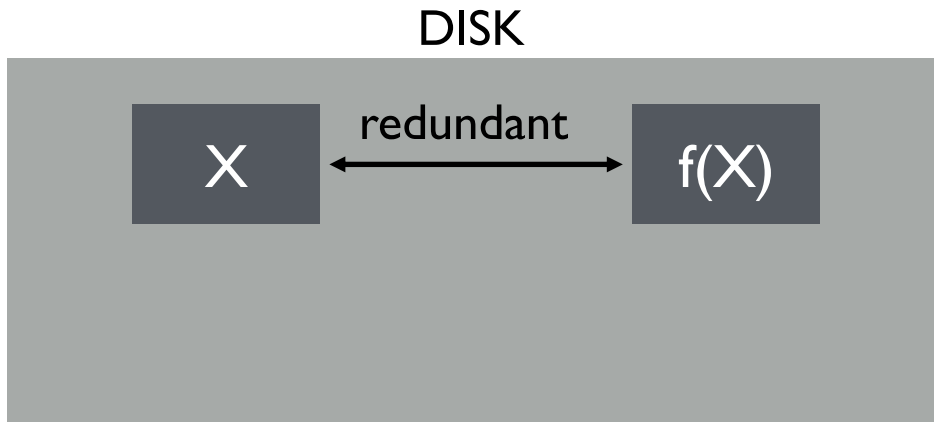
Atomicity

- Logging gives us a way to ensure **atomicity**
 - If an operation should take our system from some **state A** to another **state B**, then our system is always either in **state A** or in **state B**.
 - We are never exposed to an intermediate state, or to a state that is neither A nor B.
- These guarantees are much stronger than what we get with FSCK
 - FSCK gives consistency.
 - Atomicity gives correctness.

LOGGING EXAMPLE

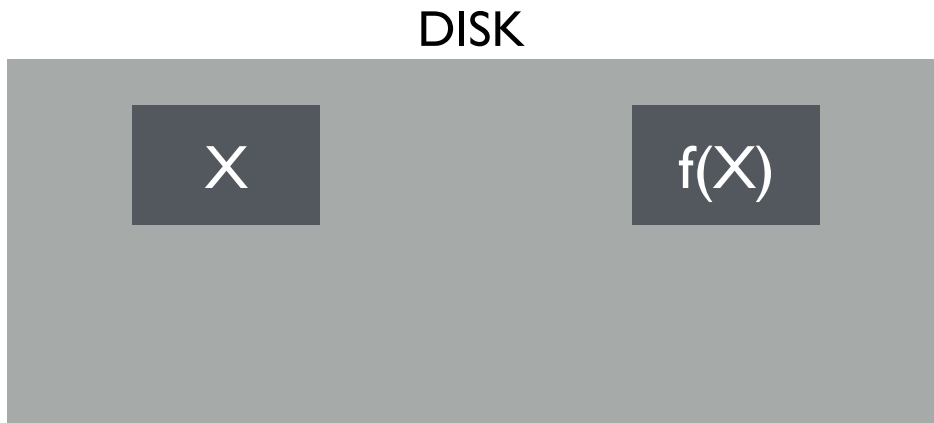
Updating Dependent Data Without Logging

Want to replace X with Y .



Updating Dependent Data Without Logging

Want to replace X with Y .

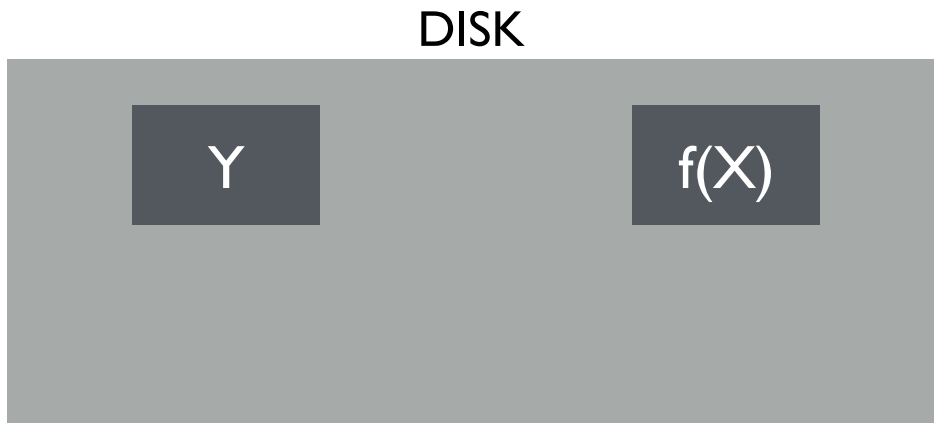


Could we survive
a crash at this point?

Yes!

Updating Dependent Data Without Logging

Want to replace X with Y .

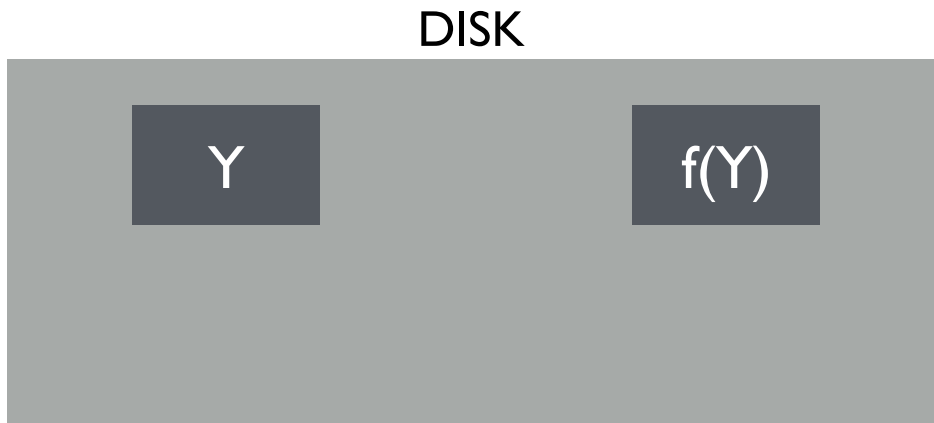


Could we survive
a crash at this point?

No!

Updating Dependent Data Without Logging

Want to replace X with Y .

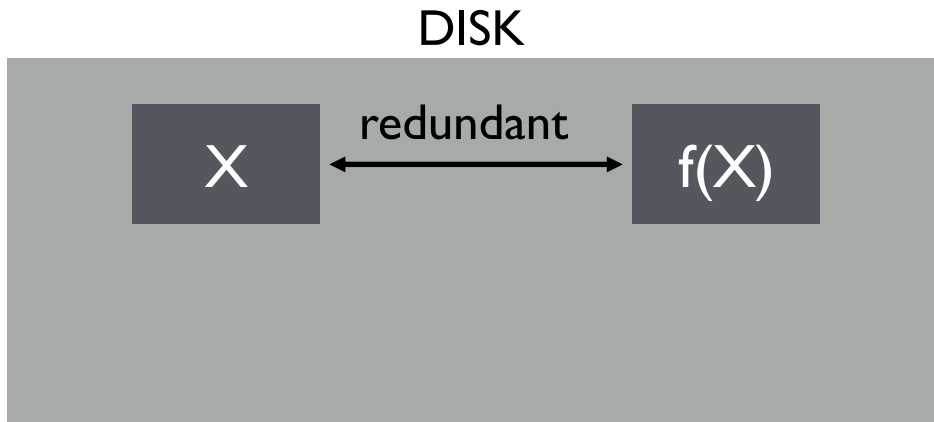


Could we survive
a crash at this point?

Yes!

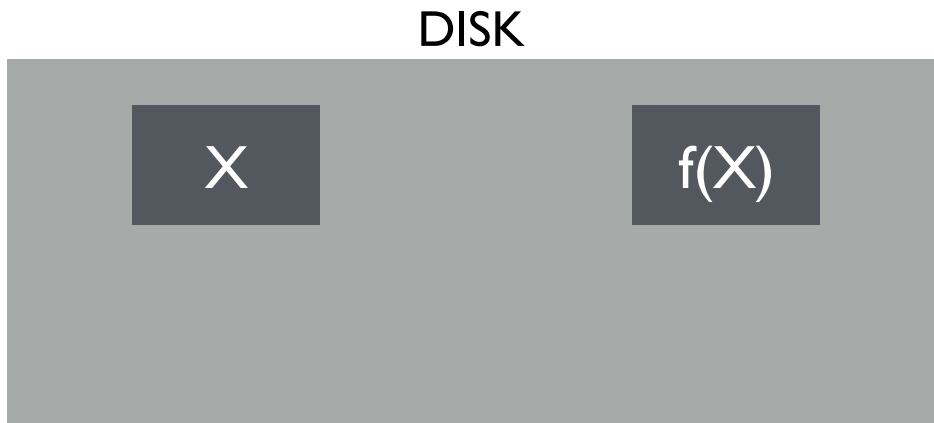
Preserving Invariants with Logging

Want to replace X with Y . **Logging:**



Preserving Invariants with Logging

Want to replace X with Y . **Logging:**

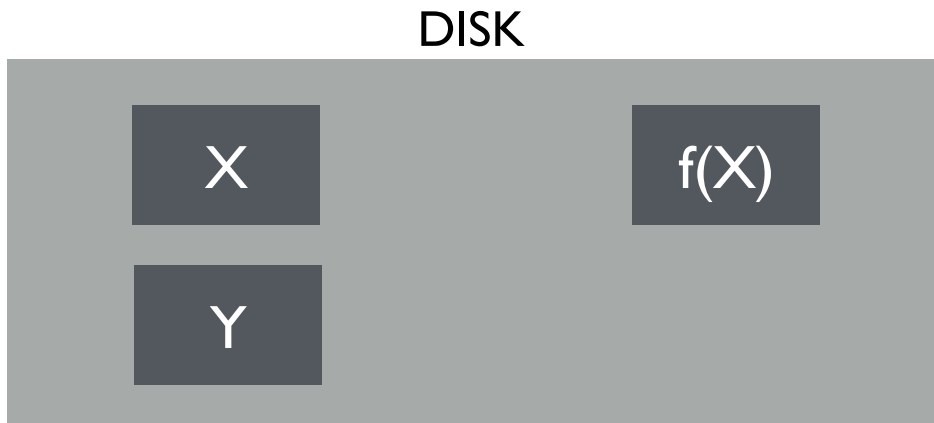


Could we survive a crash at this point?

Yes!

Preserving Invariants with Logging

Want to replace X with Y . **Logging:**

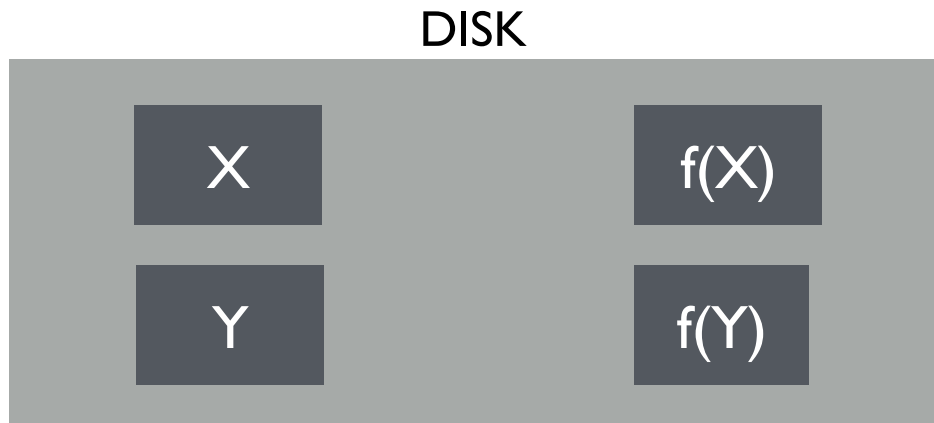


Could we survive a crash at this point?

Yes!

Preserving Invariants with Logging

Want to replace X with Y . **Logging:**

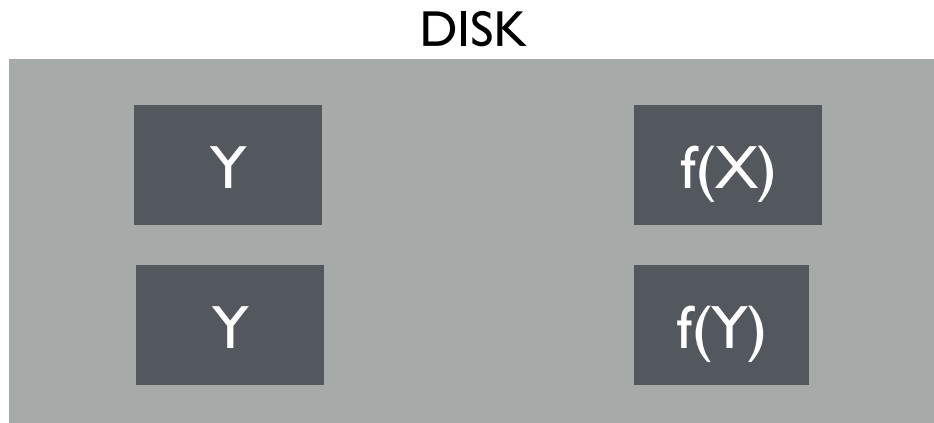


Could we survive a crash at this point?

Yes!

Preserving Invariants with Logging

Want to replace X with Y . **Logging:**

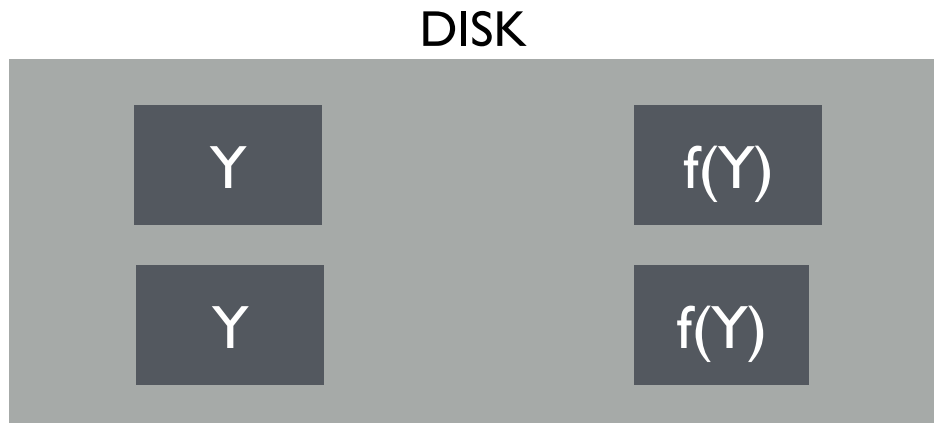


Could we survive a crash at this point?

Yes!

Preserving Invariants with Logging

Want to replace X with Y. **Logging:**

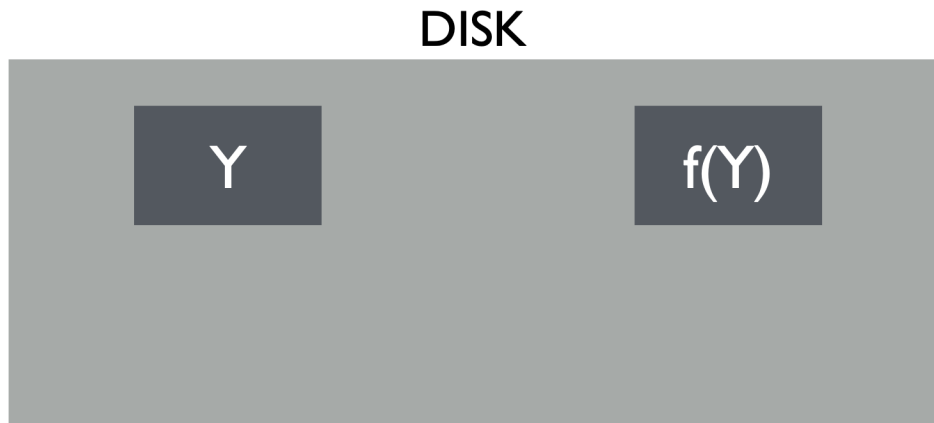


Could we survive
a crash at this point?

Yes!

Preserving Invariants with Logging

Want to replace X with Y . **Logging:**



Could we survive a crash at this point?

Yes!

We could survive a crash at any point in time because our log lets us recover.

Logging Considerations

- If we log everything, we write all data twice
 - This halves our performance!
 - (This is OK in algorithms, but not in systems)
 - Common solution: only log metadata
 - We may lose data, but we will always be consistent
 - Another approach is to use a **logical log**
 - Instead of storing final values in our log, we encode just enough information needed to describe the change
 - Can be more complicated, but may yield some gains

Logging Considerations

- If we crash in the middle of a logging operation, how do we know?
 - We add a **commit entry** at the end of our log transaction. Logging becomes:
 1. Write a **transaction begin** record
 2. Write log entries
 3. Write a **transaction end** record
 - If we don't see a corresponding pair of transaction begin/ends, we don't try to replay that log

Summary

- File system checkers let us confirm our system is consistent, and recover from *some* types of inconsistencies
 - **Problem:** The cost of a system-wide check is expensive, and the guarantees are not strong
- Logging lets us transition from consistent state to consistent state as a series of checkpoints
 - The overheads per operation can be high
 - Commonly mitigated by only logging metadata

Summary (continued)

- Many file systems use logging (journals)
 - Often in metadata-only mode
- Other techniques also exist
 - **Copy-on-write**
 - All data is written to a new location, and then a “pointer swap” transitions to new version
 - **Soft updates**
 - Carefully ordering updates to structures in a way that avoids inconsistencies
 - Difficult to get right, not ubiquitous in practice

