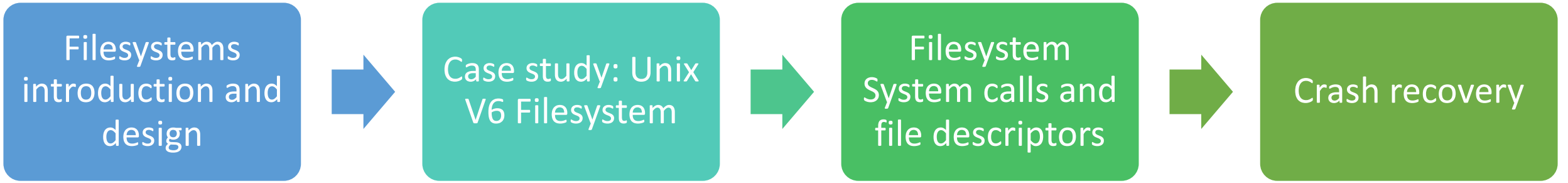
CS111, Lecture 7
Crash Recovery, Continued

Topic 1: Filesystems - How can we design filesystems to manage files on disk, and what are the tradeoffs inherent in designing them? How can we interact with the filesystem in our programs?

CS111 Topic 1: Filesystems



Lecture 1/ lecture 2 | **Lectures 2-4** | **Lecture 5** | **Lecture 6 / Today**

assign2: implement a program that can repair a filesystem after a crash!

Learning Goals

- Gain exposure to 3 approaches to crash recovery: consistency checks on reboot, ordered writes and write-ahead logging
- Compare and contrast different approaches to crash recovery
- Understand the limitations and tradeoffs of crash recovery

Plan For Today

- **Recap**: Crash Recovery So Far
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging (“Journaling”)
- assign2

Plan For Today

- **Recap: Crash Recovery So Far**
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging (“Journaling”)
- assign2

Crash Recovery

Sometimes, computers crash or shut down unexpectedly. In those situations, we want to avoid filesystem data loss or corruption as much as possible.

How can we recover from crashes without losing file data or corrupting the disk?

Crash Recovery

Key challenge: tradeoffs between *crash recovery abilities* and *filesystem performance.*

Crash Recovery

To understand crash recovery, we need to understand all places where filesystem data is stored and maintained.

- We know about most of the disk itself (e.g. Unix V6 layout)
- Now we know that **free blocks can be tracked using a bitmap**. This factors into crash recovery (e.g. free blocks not in a consistent state).
- There is also the **block cache** in memory that stores frequently-used blocks accessed from disk. **Now we know that updates may not always be written to disk immediately, and the cache may reorder writes** (e.g. if we write file A and then B, B may be written before A)**.** This factors into crash recovery (e.g. losing last 30sec of work, and trying to maintain disk consistency).

Plan For Today

- **Recap**: Crash Recovery So Far
- **Approach #1: Consistency check on reboot (fsck)**
- Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging (“Journaling”)
- assign2

Crash Recovery

Idea #1: don't make any design changes to the filesystem structure to implement crash recovery. Instead, let's write a program that runs on bootup to check the filesystem for consistency and repair any problems it can.

Example: Unix **fsck** ("file system check")

- Must check whether there was a clean shutdown (if so, no work to do). How do we know? **Set flag on disk on clean shutdown, clear flag on reboot**.
- If there wasn't, then scan disk contents, identify inconsistencies, repair them.
- Scans metadata (inodes, indirect blocks, free list, directories)
- Goals: restore consistency, minimize info loss

Possible fsck Scenarios

Example #1: block in file and also in free list?

Action: remove block from free list

Example 2: block a part of two different files (!!)

(Maybe deleting a file, then making a new file, but with the block cache the new file updates were written but not the old file updates)

Action: randomly pick which file should get it? Make a copy for each? Remove from both? (probably not, don't want to lose potentially-useful data)

Example 3: inode *reference count* > 0, but not referenced in any directory.

Action: create link in special lost+found directory.

Crash Recovery

Idea #1: don't make any design changes to the filesystem structure to implement crash recovery. Instead, let's write a program that runs on bootup to check the filesystem for consistency and repair any problems it can.

Example: Unix **fsck** ("file system check")

- If there wasn't a clean shutdown, then scan disk contents, identify inconsistencies, repair them.
- Scans metadata (inodes, indirect blocks, free list, directories)
- Goals: restore consistency, minimize info loss

What are the downsides/limitations of fsck? Respond on pollev.com/cs111 or text CS111 to 22333 once to join.

What are the downsides/limitations of fsck?

Limitations of fsck

What are the downsides/limitations of **fsck**?

- Time: can't restart system until **fsck** completes. Larger disks mean larger recovery time (Used to be manageable, but now to read every block sequentially in a 5TB disk -> 8 hours!)
- Restores consistency but doesn't prevent loss of information.
- Restores consistency but filesystem may still be unusable (e.g. a bunch of core system files moved to lost+found)
- Security issues: a block could migrate from a password file to some other random file.

Can we do better? What if we made design changes to the filesystem structure to implement crash recovery?

Plan For Today

- **Recap**: Crash Recovery So Far
- Approach #1: Consistency check on reboot (**fsck**)
- **Approach #2: Ordered Writes**
- Approach #3: Write-Ahead Logging (“Journaling”)
- assign2

Ordered Writes

Idea #2: what if we could make any design changes to the filesystem structure to implement crash recovery? What could we implement?

Let's revisit a corruption example: block in file and also in free list. (e.g. file growing, claims block from free list, but crash before free list updates)

What could we require about the order of operations here to ensure that a block is never both in the free list and in an inode?

We could require that writes happen in a particular order. E.g. always write updates to free list before updates to inode in this example.

Ordered Writes

Idea #2: what if we could make any design changes to the filesystem structure to implement crash recovery? What could we implement?

We could prevent certain kinds of inconsistencies by making updates in a particular order.

Example: adding block to file: first write back the free list, then write the inode. Thus we could never have a block in both the free list and an inode. **However, we could leak disk blocks (how?)**

Ordered Writes

Idea #2: We could prevent certain kinds of inconsistencies by making updates in a particular order. In some situations, force synchronous writes to ensure a particular order.

In general:

- Always initialize target before initializing new reference (e.g. initialize inode before adding directory entry to it)
- Never reuse a resource (inode, disk block, etc.) before nullifying all existing references to it (e.g. adding block to free list)
- Never clear last reference to a live resource before setting new reference, preserving data so you don't lose it (e.g. moving a file)

Result: eliminate the need to wait for **fsck** on reboot!

Ordered Writes

Downside #1: *performance*. This approach forces synchronous metadata writes in the middle of operations, partially defeating the point of the block cache.

Improvement: don't actually do synchronous writes, just keep track of dependencies in the block cache to remember what order we must do operations when we actually do them.

Example: after adding block to file, add dependency between inode block and free list block. When it's time to write inode to disk, make sure free list block has been written first.

Tricky to get right– circular dependencies possible! (A -> B -> C -> A)

Ordered Writes

Downside #2: can leak resources (e.g. free block removed from free list but never used)

Improvement: run **fsck** in the background to reclaim leaked resources (**fsck** can run in background because filesystem is repaired, but resources have leaked)

Can we do better? What if we left a paper trail of disk operations?

Plan For Today

- **Recap**: Crash Recovery So Far
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes
- **Approach #3: Write-Ahead Logging ("Journaling")**
- assign2

Write-Ahead Logging (Journaling)

- Have an append-only log on disk that stores information about disk operations
- Before performing an operation, record its info in the log, and write that to disk *before* doing the operation itself ("write-ahead")
  - E.g. "I am adding block 4267 to inode 27, index 5"
- Then, the actual block updates can be carried out later, in any order
- If a crash occurs, replay the log to make sure all updates are completed on disk. Thus, we can detect/fix inconsistencies without a full disk scan.

Write-Ahead Logging (Journaling)

- Typically we only log *metadata operations*, not actual file data operations (data is much more expensive, since much more written to log). Tradeoff!
- Most modern filesystems do some sort of logging (e.g. Windows NTFS) – may allow you to choose whether you want data logging or not.
- Logs one of the most important data structures used in systems today

Write-Ahead Logging (Journaling)

Problem: log can get long!

Solution: occasional "checkpoints" – truncate the log occasionally once we confirm that portion of the log is no longer needed.

Problem: could be multiple log entries for a single "operation" that should happen atomically.

Solution: have a log mechanism to track "transactions" (atomic operations) and only replay those if the entire transaction is fully entered into the log.

Problem: we could replay a log operation that has already happened.

Solution: make all log entries *idempotent* (doing multiple times has same effect as doing once). E.g. "append block X to file" (bad) vs. "set block number X to Y"

Plan For Today

- **Recap**: Crash Recovery So Far
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging (“Journaling”)
- **assign2**

assign2

- Implement a program that replays a log after a crash
- Mix of filesystem exploration (playing around with simulated filesystems, viewing logs and filesystem state) and coding (about ~10-15 lines total)
- Released tomorrow morning (Tues.)

Recap

- **Recap**: Crash Recovery So Far
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging ("Journaling")
- assign2

Next time: introduction to multiprocessing

Lecture 7 takeaway: There are various ways to implement crash recovery, each with tradeoffs between durability, consistency and performance. Many filesystems today implement logging to recover metadata operations after a crash.