CS111, Lecture 7

Crash Recovery, Continued

😷 masks strongly recommended

CS198 Section Leading!

cs198@cs.stanford.edu

cs198.stanford.edu – application due 10/19

Announcements

- New helper hours times + tea hours shifted this week
- Assign0 grades released soon!

CS111 Topic 1: Filesystems

Key Question: *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?*

Filesystems introduction and design → Case study: Unix V6 Filesystem → Filesystem System calls and file descriptors → Crash recovery

Lecture 2 | **Lecture 3-4** | **Lecture 5** | **Today/Lecture 7**

assign2: implement a program that can repair a filesystem after a crash, and explore some of the security and ethical implications of OSes / filesystems.

Learning Goals

- Gain exposure to 2 further approaches to crash recovery: 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 and fsck
- Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging (“Journaling”)
- assign2

Plan For Today

- **Recap: Crash Recovery and fsck**
- Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging ("Journaling")
- assign2

Crash Recovery

Challenge #1 – data loss: crashes can happen at any time, and not all data might have been saved to disk.

- E.g. if you saved a file but it hadn’t actually been written to disk yet.

Challenge #2 - inconsistency: Crashes could happen even in the middle of operations, and this could leave the disk in an inconsistent state.

- E.g. adding block to file: inode was written to store block number, but block wasn’t marked in the filesystem as used (it’s still listed in the free list)

Approach #1: fsck

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

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 and 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?

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? E.g., can we avoid leaking data?

Plan For Today

- **Recap**: Crash Recovery and fsck
- Approach #2: Ordered Writes
- **Approach #3: Write-Ahead Logging ("Journaling")**
- assign2

Write-Ahead Logging (Journaling)

Let's keep a "paper trail" of disk operations that we can revisit in the event of a crash.

- 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

assign2 Log Example

```
[offset 33562846]
* LSN 1838326418
  LogBlockAlloc
    blockno: 1027
    zero_on_replay: 0

[offset 33562862]
* LSN 1838326419
  LogPatch
    blockno: 8
    offset_in_block: 136
    bytes: 0304
  inode #52 (i_addr[0] = block pointer 1035)
```

Write-Ahead Logging ("Journaling")

What are the downsides/limitations of our logging design so far?

Respond on PollEv: pollev.com/cs111fall23
or text CS111FALL23 to 22333 once to join.

What are the downsides/limitations of our logging design so far?

Nobody has responded yet.

Hang tight! Responses are coming in.

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. (assign2 wraps each transaction with LogBegin and LogCommit)

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"

Write-Ahead Logging ("Journaling")

Problem: log entries must be written synchronously before the operations

Solution: delay writes for log, too (i.e. build log, but don't write immediately; when a block cache block is written, write relevant log entries then). Though this risks losing some log entries.

Logging doesn't guarantee that everything is preserved, but it does guarantee that what's there is consistent (separates *durability* – data will be preserved – from *consistency* – state is consistent)

Crash Recovery

Ultimately, tradeoffs between *durability, consistency* and *performance*

- E.g. if you want durability, you're going to have to sacrifice performance
- E.g. if you want highest performance, you're going to have to give up some crash recovery capability
- What kinds of failures are most important to recover from, and how much are you willing to trade off other benefits (e.g. performance)?

Still lingering problems – e.g. disks themselves can fail

Crash Recovery

We've discussed 3 main approaches to crash recovery:

1. **Consistency check on reboot (fsck)** – no filesystem changes, run program on boot to repair whatever we can. But can't restore everything and may take a while.
2. **Ordered Writes** – modify the write operations to always happen in particular orders, eliminating various kinds of inconsistencies. But requires doing synchronous writes or tracking dependencies and can leak resources.
3. **Write-Ahead Logging** – log metadata (and optionally file data) operations before doing the operations to create a paper trail we can redo in case of a crash.

Demo – Filesystem Recovery

- Assign2 tools let you simulate real filesystems, make them crash, and experiment with recovery tools
- 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)
- Also kicks off embedded ethics discussions about OS trust and security
- You'll have a chance to play with these tools in the assignment and in section this week.

Plan For Today

- Recap: Crash Recovery and fsck
- Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging ("Journaling")
- Assign2

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.

Next time: introduction to multiprocessing