CS111, Lecture 6
Crash Recovery, Continued

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 14 through 14.1

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 → Crash Recovery → Filesystem System calls and file descriptors

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

- Understand the write-ahead logging approach to crash recovery
- Compare and contrast different crash recovery approaches

Plan For Today

- **Recap**: Crash Recovery so far
- More about Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging (“Journaling”)

Plan For Today

- **Recap: Crash Recovery so far**
- More about Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging (“Journaling”)

Free List and Block Cache

Filesystems commonly use a **bitmap** to track free blocks (1 bit / block , 1 = free, 0 = used).

- During allocation, search bit map for block close to previous block in file
  - Want *locality* – data likely used next is close by (linked list not as good)
- To ensure a minimum number of free blocks, we can pretend the disk is fuller than it actually is (!)

Many OSes have a block cache that stores recently-accessed disk blocks.

- Helps us avoid always having to read a block from disk – check cache first
- “least recently used” one way to manage cache – if full, kick out least recently used block

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)

- Note: garbage data in a file is an example of data loss rather than inconsistency

Approach #1: fsck

Idea #1: 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”)

- Runs when we reboot after a crash - scan metadata, identify inconsistencies, repair them, with goal of restoring consistency, minimizing info loss.
- Downsides:
  - Time – can’t restart system until **fsck** completes
  - Restores consistency, but other limitations – e.g. can’t prevent info loss, system may be consistent but unusable (e.g. system files moved to lost+found), security issues migrating a block to another file

Ordered Writes

Idea #2: 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?)**

Plan For Today

- **Recap**: Crash Recovery so far
- **More about Approach #2: Ordered Writes**
- Approach #3: Write-Ahead Logging (“Journaling”)

Ordered Writes

Idea #2: We could prevent certain kinds of inconsistencies by making updates in a particular order.

In general:

- Always initialize target before initializing new reference (e.g. initialize inode before adding directory entry to it)
- Before reusing a resource (inode, disk block, etc.), nullify all existing references to it (e.g. adding block to free list)

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

Ordered Writes

Rule 1: Always initialize target before initializing new reference (e.g. initialize inode before adding directory entry to it)

Ordered writes approach:

What if (creating a file):

✓ 1. Add dirent pointing to new inode

Crash!

2. Initialize new inode

Problem: dirent pointing to uninitialized inode!

What if (creating a file):

✓ 1. Initialize new inode

Crash!

2. Add dirent pointing to new inode

Less-severe issue: file not referred to by any dirent (can stick in lost+found).

Ordered Writes

Rule 2: Before reusing a resource (inode, disk block, etc.), nullify all existing references to it (e.g. adding block to free list)

Ordered writes approach:

What if (file no longer needs last block):

✓ 1. Mark block as free in free list

Crash!

2. Update inode to no longer refer to this block number.

Problem: block in file and marked free in free list!

What if (file no longer needs last block):

✓ 1. Update inode to no longer refer to this block number.

Crash!

2. Mark block as free in free list

Less-severe issue: leaked block (can reclaim later)

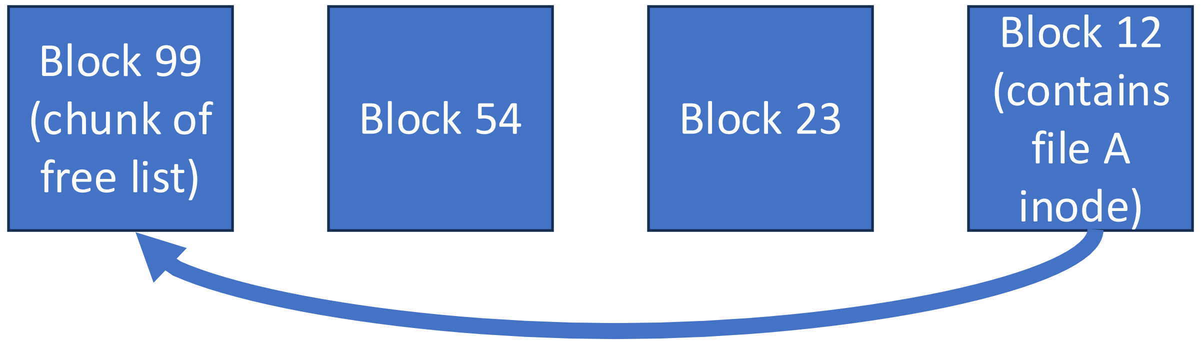
Ordered Writes

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

<u>Improvement:</u> block cache tracks dependencies between blocks for later – can do writes asynchronously, since we remember the required order of operations.

- *Example:* adding a block to file A (requires updating inode and free list). <u>We must write free list first.</u> So add dependency from block storing its inode (say block 12) to the free list block (say block 99) storing the bit for the new block.

<u>Block cache:</u>

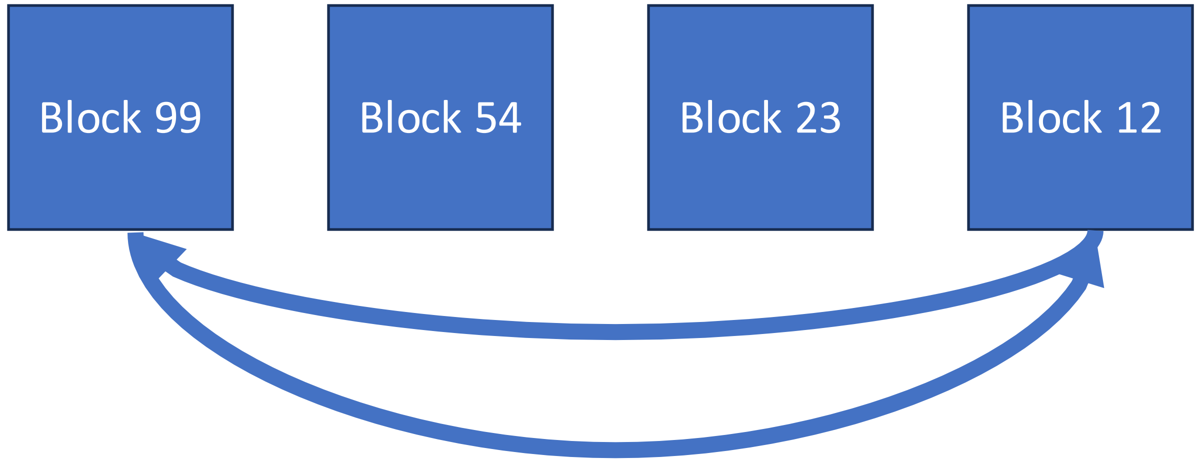

Block 99 (chunk of free list)
Block 54
Block 23
Block 12 (contains file A inode)

Must first write

Ordered Writes

Improvement: block cache tracks dependencies between blocks for later – can do writes asynchronously, since we can remember the required order of operations. **Tricky to get right– circular dependencies possible, must avoid!**

Block cache:



Example: delete file A (update inode block 12, then free list block 99), also create file B (update free list block 99, then inode block 12)

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 so far
- More about Approach #2: Ordered Writes
- **Approach #3: Write-Ahead Logging (“Journaling”)**

Write-Ahead Logging (Journaling)

Keep a "paper trail" of disk operations that we can revisit if a crash occurs.

- append-only log file 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.
- Log is "source of truth"

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) – many allow choice 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")

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)

Write-Ahead Logging (“Journaling”)

Problem: what if we crash, replay the log and crash *again while replaying*? (Assume no more crashes after that). (Hint: every time we crash, we reboot and replay the log). (log goes left to right, old to new).

set inode 5’s size to 1024	append block 24 to inode 5	…

Respond on PollEv:
pollev.com/cs111

What would happen if we crash, replay, and then crash again (just once) while replaying?

Nobody has responded yet.

Hang tight! Responses are coming in.

Write-Ahead Logging ("Journaling")

Problem: what if we crash, replay the log and crash *again while replaying*? (Assume no more crashes after that). (Hint: every time we crash, we reboot and replay the log). (log goes left to right, old to new).

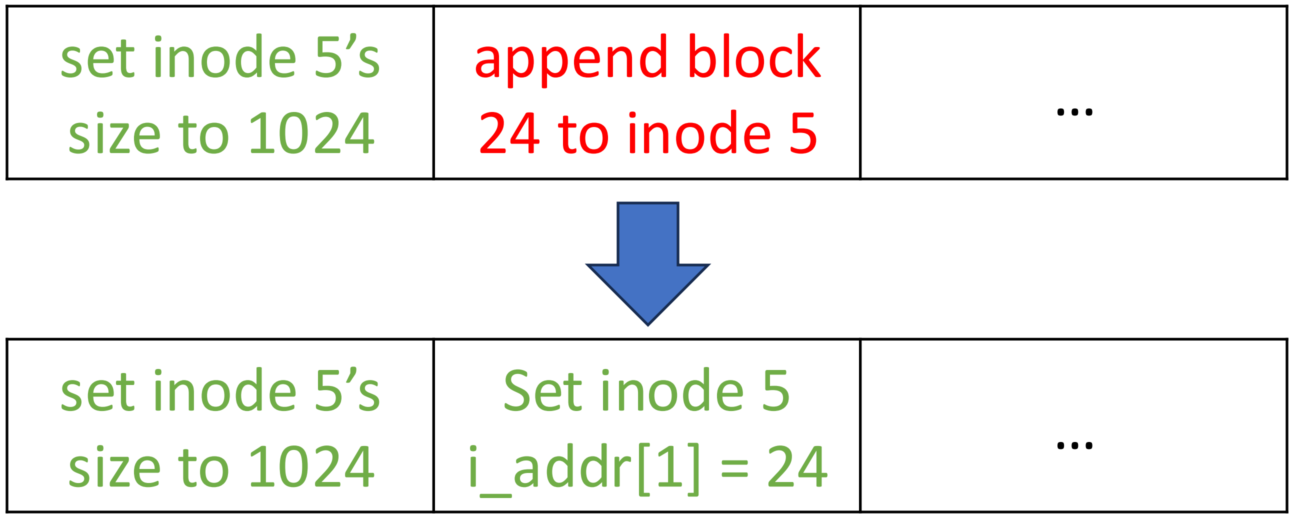
set inode 5's size to 1024	append block 24 to inode 5	…

Inode 5 would have block 24 appended twice! (uh oh)

Write-Ahead Logging ("Journaling")

Problem: it's possible for us to replay a log operation that has already happened. (ie. By crashing during replay, or by crashing after operation is done but before log is truncated).

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"


set inode 5's size to 1024
append block 24 to inode 5
…
set inode 5's size to 1024
Set inode 5 i_addr[1] = 24
…

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 all 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)** – add-on program (no changes to how filesystem performs operations when running) that runs 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.

assign2

- 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

Recap

- **Recap**: Crash Recovery so far
- More about Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging (“Journaling”)

Lecture 6 takeaways: There are various ways to implement crash recovery, such as logging, each with tradeoffs between durability, consistency and performance.

Next time: using the filesystem in our own programs