CS111, Lecture 5
Crash Recovery

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** | **Today/Lecture 6** | **Lecture 6-7**

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

Additional Filesystem Info

Q: Why do spinning disks only support reading/writing in units of sectors?
A: one reason is the disk does error-correction per-sector on disk, so imposes restriction on reading/writing whole sectors

Q: what about flash storage?

A: flash storage hardware is different and thus it does use different operations other than reading/writing single sectors (we'll see more later in the quarter!), but for compatibility they often use the same read/write sector interface as spinning hard drives.

Q: why does a larger block size mean more efficient I/O?

A: reading larger chunks, fewer seeks

Learning Goals

- Learn about the role of the free map and block cache in filesystems
- Understand the goals of crash recovery and potential tradeoffs
- Compare and contrast different approaches to crash recovery

Plan For Today

- Free space management
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes

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)
- We'll learn about how free blocks on disk are tracked. This factors into crash recovery (e.g. free blocks not in a consistent state).
- We'll learn about the **block cache** in memory that stores frequently-used blocks accessed from disk.

Plan For Today

- **Free space management**
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes

Free Space Management

Early Unix systems (like Unix v6) used a linked list of free blocks

- Initially sorted, so files allocated contiguously, but over time list becomes scrambled

More common: use a **bitmap**

- Array of bits, one per block: 1 means block is free, 0 means in use
- Takes up some space – e.g. 1TB capacity -> 2^{28} 4KB blocks -> 32 MB bitmap
- 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)

Problem: slow if disk is nearly full, and files become very scattered

Free Space Management

More common: use a **bitmap** – an array of bits, one per block, where 1 means block is free, 0 means in use.

- During allocation, search bit map for block close to previous block in file

Problem: slow if disk is nearly full, and blocks very scattered

- Expensive operation to find a free block on a mostly full disk
- Poor *locality* – data likely to be used next is not close by

Solution (used by BSD): don't let disk fill up!

- E.g. Linux pretends disk has less capacity than it really has (try **df** on myth!)
- Increase disk cost, but for better performance

Plan For Today

- Free space management
- **Block Cache**
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes

Block Cache

Problem: Accessing disk blocks is expensive, especially if we do it repeatedly for the same blocks.

Idea: use part of main memory to retain recently-accessed disk blocks. (Many OSes do this).

- A *cache* is a space to store and quickly access recently- / frequently-used data.
- Frequently-referenced blocks (e.g. indirect blocks for large files) usually in block cache. (not necessarily whole files, just individual blocks).
- Invisible to programs – programs don't need to know how this works

Challenge: how do we utilize it? What if it gets full?

Block Cache

Challenge: how do we utilize it? What if it gets full?

One approach - least-recently-used “LRU” replacement – If we need something not in the cache, we read it from disk and then add it to the cache. If there’s no room in the cache, we remove the least-recently-used element.

Block Cache

Key Question: When a block in the block cache is modified, do we stop and wait and immediately write it to disk? Or do we delay it slightly until later?

"Synchronous Writes" *Write immediately to disk*	**"Delayed Writes"** *Don't write immediately to disk*
	• Wait (e.g. Unix used 30sec) in case of more writes to a block, or it is deleted

Block Cache

Key Question: When a block in the block cache is modified, do we stop and wait and immediately write it to disk? Or do we delay it slightly until later?

"Synchronous Writes" *Write immediately to disk*	**"Delayed Writes"** *Don't write immediately to disk*
• **Slow**: program must wait to proceed until disk I/O completes • **Safer**: less risk (but not zero risk!) of data loss because it's written as soon as possible.	• Wait (e.g. Unix used 30sec) in case of more writes to a block, or it is deleted

Block Cache

Key Question: When a block in the block cache is modified, do we stop and wait and immediately write it to disk? Or do we delay it slightly until later?

"Synchronous Writes" *Write immediately to disk*	**"Delayed Writes"** *Don't write immediately to disk*
• **Slow**: program must wait to proceed until disk I/O completes • **Safer**: less risk (but not zero risk!) of data loss because it's written as soon as possible.	• Wait (e.g. Unix used 30sec) in case of more writes to a block, or it is deleted • **Fast + Efficient**: writes return immediately, eliminates disk I/Os in many cases (e.g. many small writes to the same block) • **Less safe**: may lose more data after a system crash! "Are you willing to lose your last 30sec of work in exchange for a performance bump?" (if e.g. ~2-10x faster) • (Aside– program can call **fsync** function to force disk write)

Plan For Today

- Free space management
- Block Cache
- **Crash Recovery Overview**
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes

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?

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

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. if a modification affects multiple blocks, a crash could occur when some of the blocks have been written to disk but not the others.
  - 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)

Ideally, filesystem operations would be **atomic,** meaning they happen either entirely or not at all. But this isn't fully possible.

Crash Recovery

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

Crash Recovery

We will discuss 3 approaches to crash recovery, building up to the most common one – **logging:**

1. Consistency Check on reboot (**fsck**)
2. Ordered Writes
3. Write-Ahead Logging (“Journaling”)

Plan For Today

- Free space management
- Block Cache
- Crash Recovery Overview
- **Approach #1: Consistency check on reboot (fsck)**
- Approach #2: Ordered Writes

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")

- 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 (!!) *How?*

Say we are doing two filesystem operations, and we have a block cache:

- *Delete file A:* delete inode A, update freemap to mark blocks free
- *Create file B, uses same blocks as A:* create inode B, update freemap to mark same blocks in use

<u>*Possibility:*</u> block cache reorders operations, only creates inode B, then crash!

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 (!!) *How?*

Action: Make a copy for each? (works, though potential security issues if block is migrated to unintended file) Remove from both? (probably not, don't want to lose potentially-useful data)

Example 3: inode *reference count* (# times referenced by a directory entry) = 1, but not referenced in any directory.

Action: create link in special lost+found directory.

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? Can we avoid having to scan the whole disk on reboot?

Plan For Today

- Free space management
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)
- **Approach #2: Ordered Writes**

Ordered Writes

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)

Key insight: *we are performing 2 operations – removing block from free list, plus adding block number to inode. If we want to ensure that a block is never both in the free list and in an inode simultaneously, which operation should we do first? Would this resolve all problems?*

Respond on PollEv: pollev.com/cs111
or text CS111 to 22333 once to join.

Which operation should we perform first? Would this ordering resolve all problems?

Update free list first - then no risk of filesystem corruption

0%

Update free list first - but it's possible we end up with a block that is marked used but not actually used

0%

Update inode first - then no risk of filesystem corruption

0%

Update inode first - but it's still possible the filesystem gets corrupted

0%

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?)**

Recap

- Free space management
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes

Next time: more about crash recovery

Lecture 5 takeaways: The free list tracks free blocks on disk and is commonly implemented using a bitmap. The block cache caches recently-accessed disk blocks. Crash recovery challenges include both data loss and inconsistency. **Fsck** and ordered writes are 2 approaches to crash recovery.