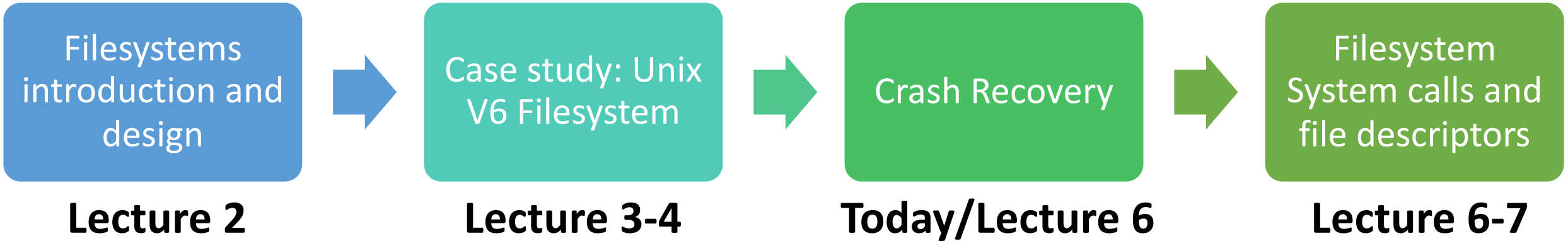
CS111, Lecture 5
Crash Recovery

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?*



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

- 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
- Approach #3: Write-Ahead Logging (“Journaling”)

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
- Approach #3: Write-Ahead Logging (“Journaling”)

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: 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
- Approach #3: Write-Ahead Logging (“Journaling”)

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.

The block cache allows us to *delay* operations until later (e.g. when modifying a single block many times, don't write to disk every time, but rather once later on). We'll see more about this in a bit!

Plan For Today

- Free space management
- Block Cache
- **Crash Recovery Overview**
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging (“Journaling”)

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 in their entirety without interruption – they are never left in an incomplete state. But this isn't fully possible.

Crash Recovery

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

Tradeoffs Example: the Block Cache

The block cache is a great example of how we may have to choose between crash recovery abilities and filesystem performance.

Key question: what happens when a block in the cache is modified? Do we stop and wait and immediately write it to disk? Or do we delay it slightly until later?

Block Cache

Key Question: what happens when a block in the cache is modified? Do we immediately write it to disk? **Pros/cons?**

If we immediately write to disk ("synchronous writes"):

- Safer: less risk (but not zero risk!) of data loss because it's written as soon as possible.
- Slow: program must wait to proceed until disk I/O completes

Block Cache

Key Question: what happens when a block in the cache is modified? Do we immediately write it to disk? **Pros/cons?**

If we don't immediately write to disk ("delayed writes"):

- Wait a while (Unix chose 30 seconds) in case there are more writes to that block, or it is deleted
- Fast and Efficient: writes return immediately, eliminates disk I/Os in many cases (e.g. many small writes to the same block)
- Dangerous: may lose more data after a system crash! "Are you willing to lose your last 30sec of work in exchange for performance bump?"
- (Side note – **fsync** system call lets a program force a write to disk)

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
- Approach #3: Write-Ahead Logging (“Journaling”)

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: give to newest file? randomly pick which gets it? Make a copy for each? 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? What if we made design changes to the filesystem structure to implement crash recovery?

Plan For Today

- Free space management
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)
- **Approach #2: Ordered Writes**
- Approach #3: Write-Ahead Logging (“Journaling”)

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)

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: 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

- Free space management
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes
- **Approach #3: Write-Ahead Logging (“Journaling”)**

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) – 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")

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

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.

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

- Free space management
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)
- Approach #2: Ordered Writes
- Approach #3: Write-Ahead Logging ("Journaling")

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**, ordered writes and logging are 3 approaches to crash recovery.