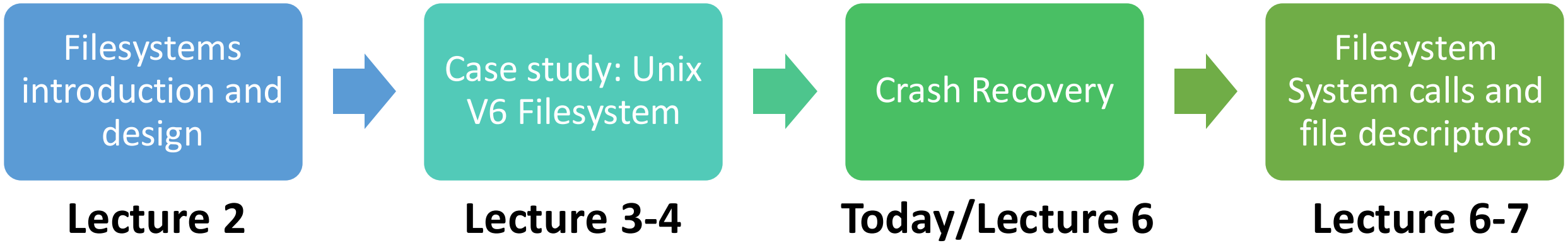
CS111, Lecture 5
Crash Recovery

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.

Other Filesystem Design Ideas

Larger block size? Improves efficiency of I/O and inodes but worsens internal fragmentation. Generally: challenges with both large and small files coexisting.

One idea: multiple block sizes

- Large blocks are 4KB, *fragments* are 512 bytes (8 fragments fit in a block)
- The last block in a file can be 0-7 fragments
- One large block can hold fragments from multiple files
- Get the time efficiency benefit of larger blocks, but the internal fragmentation benefit of smaller blocks (small files can use fragments)

Filesystem Techniques Today

- Filesystem design is a hard problem! Tradeoffs, challenges with large and small files.
- Even larger block sizes (16KB large blocks, 2KB fragments) – disk space cheap, internal fragmentation doesn't matter as much
- Reallocate files as blocks grow – initially allocate blocks one at a time, but when a file reaches a certain size, reallocate blocks looking for large contiguous clusters
- ext4 is a popular current Linux filesystem – you may notice similarities!
- NTFS (replacement for FAT) is the current Windows filesystem
- APFS ("Apple Filesystem") is the filesystem for Apple devices

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

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 – but all operations go through the block cache
- **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)

Block Cache

The block cache could also end up reordering operations!

- E.g. a bunch of operations performed, written to block cache with delayed writes
- After e.g. 30s, we go through and flush blocks to disk, could flush them in some order different from original operation order.

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.

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

Example Filesystem Operations

Main steps to create a file with some data in it:

- **Initialize a new inode**
- **Add directory entry** to refer to that inode
- **Update free list** to mark newly-used payload data blocks as used
- **Write data** to new payload blocks

Main steps to add another block of data to an existing file:

- **Update free list** to mark new block as used
- **Update inode** to store new block number (and other fields like size, etc.)
- **Write data** to new block

Operations may not be written to disk in these specific orders!

Crash Recovery

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

What if:

1. Update free list to mark new block as used
2. Update inode to store new block number

3. Write data to new block

Problem: on reboot, file has garbage block!

What if:

1. Update inode to store new block number
2. Write data to new block

3. Update free list to mark new block as used

Problem: our block could be given out later to someone else!

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?

What if:

1. Update inode to store new block number
2. Write data to new block

3. Update free list to mark new block as used

Action: remove block from free list

Possible fsck Scenarios

Example 2: block a part of two different files (how is this possible??)

Let's say we are deleting file A and also creating file B, which coincidentally uses the same old payload blocks that A used.

To delete file A, we need to…	To create file B, we need to…
• Delete inode A • Mark A's payload blocks as free in free list	• Create inode B • Add dirent for B • Mark same payload blocks as used in free list • Write data to new blocks

Key Idea: *after all these operations, their changes are in the block cache. But the cache could write the blocks to disk in some order, and we could crash before all blocks are written!*

Possible fsck Scenarios

Example 2: block a part of two different files (how is this possible??)

Let's say we are deleting file A and also creating file B, which coincidentally uses the same old payload blocks that A used.

What if: all operations performed in block cache, then block cache writes inode B's block to disk.

THEN CRASH!
No other blocks written to disk ☹

To delete file A, we need to…	**To create file B, we need to…**
• Delete inode A • Mark A's payload blocks as free in free list	• Create inode B • Add dirent for B • Mark same payload blocks as used in free list • Write data to new blocks

Possible fsck Scenarios

Example 2: block a part of two different files (how is this possible??)

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

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.