

CS111, Lecture 6

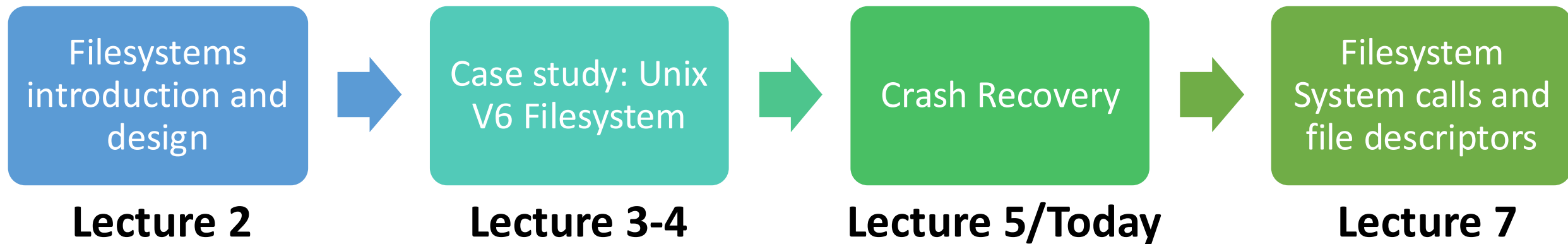
Crash Recovery, Continued

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?*



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
- can store any kind of block – inode block, singly-indirect block, dirents, etc.

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)

Crashes could happen at any time – can't fully guarantee all data is preserved but can lessen chance. (crash could even happen mid-block-write).

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?)**

Ordered Writes

Rule 1: Always initialize target (here: inode) before initializing new reference (here: dirent) to it



What if (creating a file):
1. Add dirent pointing to new inode



2. Initialize new inode

Problem: dirent pointing to uninitialized inode!



What if (creating a file):
1. Initialize new inode



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.) (e.g. adding block to free list), nullify all existing references to it (e.g. i_addr)

Ordered writes approach:

What if (file no longer needs last block):



1. Mark block as free in free list



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.



2. Mark block as free in free list

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

Ordered Writes Summary

We can prevent certain kinds of inconsistencies by making updates in a particular order.

In general:

1. Always initialize target before initializing new reference (e.g. initialize inode before adding directory entry to it)
2. 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!

Plan For Today

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

Ordered Writes

Problem: if we want to use delayed writes and thus have the block cache delay writes until later, the block cache needs a way to **remember** the order in which to write blocks to disk to adhere to ordered writes rules.

We could instead write them synchronously (why would that work?) - but that reduces performance.

Ordered Writes

Improvement: 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**).

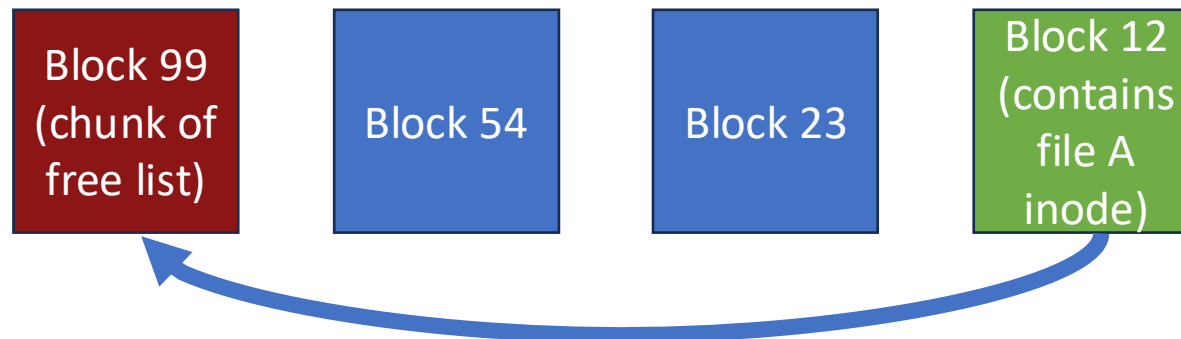
Ordered Writes

Improvement: 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**). We must write **free list** first.

We add a dependency from block storing its inode to the free list block storing the bit for the new block.

Block cache:

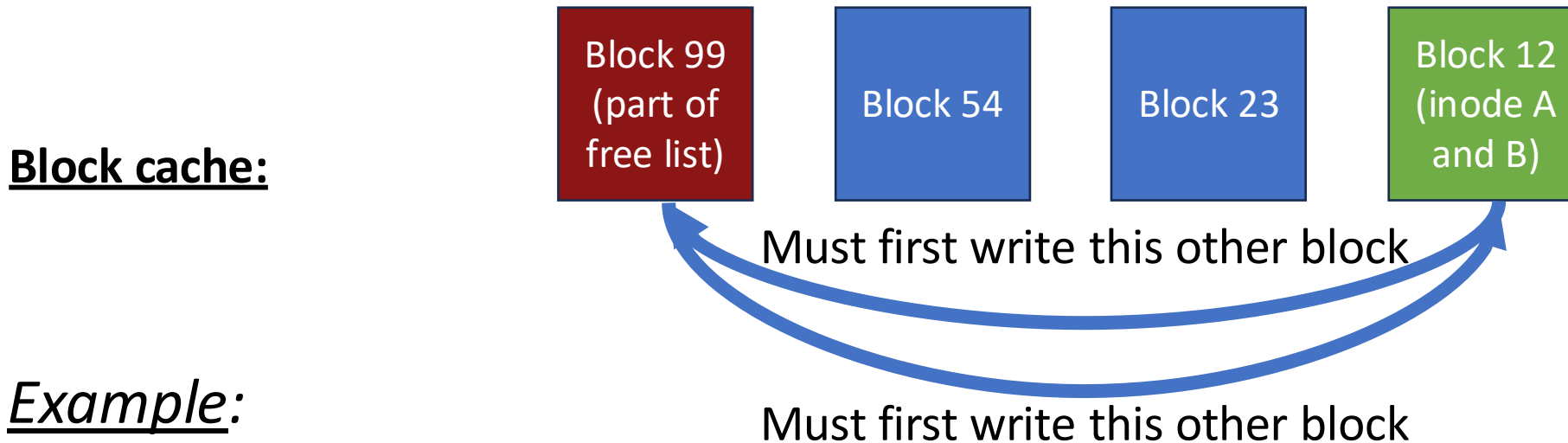


Must first write this other block

Ordered Writes

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

Tricky to get right– circular dependencies possible, must avoid!



Example:

*First, operation to delete file A (update **inode block 12**, then **free list block 99**)*

*Later, operation to create file B (update **free list block 99**, then **inode block 12**)*

If new operation will create circular dependency, stop+write existing ops to disk

Ordered Writes

Problem: 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”
- If a crash occurs, “replay” (go through log in order and execute logged operations) 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”)

- 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

There are **four key issues** we need to work through to fully understand logging.

Write-Ahead Logging (“Journaling”)

Issue #1: log can get long!

Solution: occasional “checkpoints” – truncate the log occasionally once we confirm that portion of the log is no longer needed.

Write-Ahead Logging (“Journaling”)

Issue #2: what about multi-step operations? (like creating a file).

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

- **Initialize a new inode** (eg. i_addr)
- **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

log:

...	Initialize inode 2 (eg. i_addr)	Add dirent for inode 2	Mark block 55 used
-----	---------------------------------------	---------------------------	-----------------------

Write-Ahead Logging (“Journaling”)

Issue #2: what about multi-step operations? (like creating a file).

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

- **Initialize a new inode** (e.g. i_addr)
- **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

Wait - what if we crash before logging all steps, and then replay only the steps that were logged?

Respond at pollev.com/cs111

log:

...	Initialize inode 2 (eg. i_addr)	Add dirent for inode 2	Mark block 55 used
-----	---------------------------------------	---------------------------	-----------------------

L6. What is the problem if we crash before logging all steps, and then replay only the steps that were logged?

Nobody has responded yet.

Hang tight! Responses are coming in.

Write-Ahead Logging (“Journaling”)

Issue #2: what about multi-step operations? (like creating a file).

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

- **Initialize a new inode** (e.g. `i_addr`)
- **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

Problem: inconsistency! Inode 2 `i_addr` includes block 55, but block 55 isn’t marked free in the free list.

log:

...	Initialize inode 2 (eg. <code>i_addr</code>)	Append dirent for inode 2	Mark block 55 used
-----	---	---------------------------------	-----------------------



Write-Ahead Logging (“Journaling”)

Issue #2: what about multi-step operations? (like creating a file).

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

- **Initialize a new inode** (e.g. i_addr)
- **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

What we want: if all the operation’s steps aren’t logged, *don’t replay any of them.*

log:

...	Initialize inode 2 (eg. i_addr)	Append dirent for inode 2	Mark block 5 as used
-----	---------------------------------------	---------------------------------	-------------------------

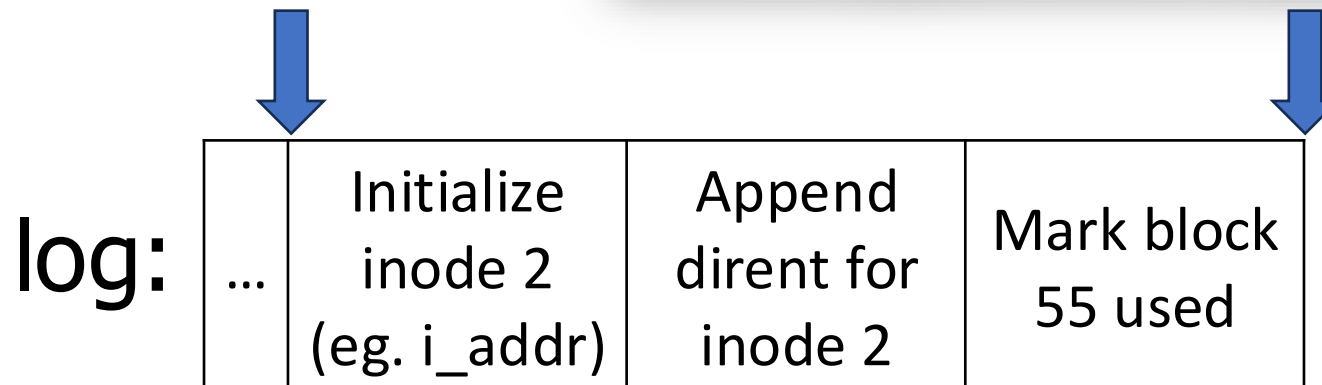
Write-Ahead Logging (“Journaling”)

Issue #2: what about multi-step operations? (like creating a file).

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

- **Initialize a new inode** (e.g. `i_addr`)
- **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

What we want: if all the operation’s steps aren’t logged, *don’t replay any of them.*



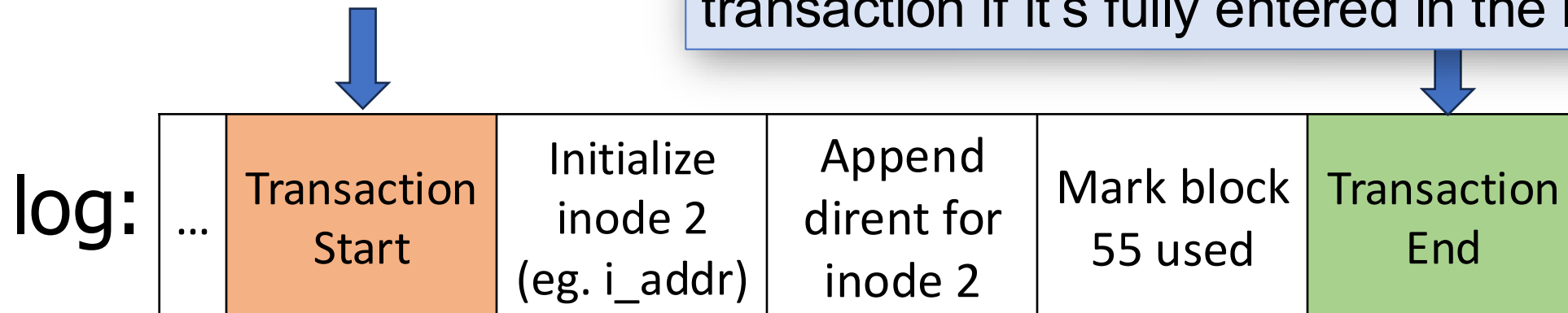
Write-Ahead Logging (“Journaling”)

Issue #2: what about multi-step operations? (like creating a file).

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

- **Initialize a new inode** (e.g. `i_addr`)
- **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

A **transaction** is a group of operations that should happen atomically. Only replay a transaction if it's fully entered in the log.



Write-Ahead Logging (“Journaling”)

Issue #2: what about multi-step operations? (like creating a file).

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

- **Initialize a new inode** (e.g. i_addr)
- **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

A **transaction** is a group of operations that should happen atomically. Only replay a transaction if it's fully entered in the log.

log:

...	Transaction Start	Initialize inode 2 (eg. i_addr)	Append dirent for inode 2
-----	-------------------	---------------------------------	---------------------------

No transaction end?
Skip this transaction.

Write-Ahead Logging (“Journaling”)

Issue #2: what about multi-step operations? (like creating a file).

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

- **Initialize a new inode** (e.g. `i_addr`)
- **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

Additionally: when you want to do a multi-step operation, log all the steps *first*, and *then* perform them.

log:

...	Transaction Start	Initialize inode 2 (eg. <code>i_addr</code>)	Append dirent for inode 2
-----	-------------------	---	---------------------------

Write-Ahead Logging (“Journaling”)

Issue #3: can you think of a scenario where “append dirent for inode 2” is replayed *multiple times*?

Possibility: we could crash mid-replay! And then replay the same entries *again*. This means we’ll append two dirents!

log:

...	Transaction Start	Initialize inode 2 (eg. i_addr)	Append dirent for inode 2	Mark block 55 used	Transaction End
-----	-------------------	---------------------------------	---------------------------	--------------------	-----------------

Write-Ahead Logging (“Journaling”)

Issue #3: can you think of a scenario where “append dirent for inode 2” is replayed *multiple times*?

Possibility: we could crash mid-replay! And then replay the same entries *again*. This means we’ll append two dirents!

Solution: require all log entries to be *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”

log:

...	Transaction Start	Initialize inode 2 (eg. i_addr)	Append dirent for inode 2	Mark block 55 used	Transaction End
-----	-------------------	---------------------------------	---------------------------	--------------------	-----------------

Write-Ahead Logging (“Journaling”)

Issue #3: can you think of a scenario where “append dirent for inode 2” is replayed *multiple times*?

Possibility: we could crash mid-replay! And then replay the same entries *again*. This means we’ll append two dirents!

Solution: require all log entries to be *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”

log:

...	Transaction Start	Initialize inode 2 (eg. i_addr)	Set block 230, dirent 5 to XXXX	Mark block 55 used	Transaction End
-----	-------------------	---------------------------------	---------------------------------	--------------------	-----------------

Write-Ahead Logging (“Journaling”)

Issue #4: we must make sure that log entries are written to disk before their actual operations are written to disk. (“write-ahead logging”).

Insight: the log is stored in blocks, too. Let’s use the “ordered writes” idea of tracking dependencies! E.g. when a block cache block needs to be written, first flush all log entry blocks. (Though this risks losing some log entries! Why?)

We could instead synchronously write to the log every time a metadata operation is performed (why would that work?) - but that reduces performance.

Write-Ahead Logging (“Journaling”)

Cool thought: as long as the log is written first, the block cache can write metadata operations in any order!

If we crash midway, we’ll replay the operations in the right order from the log.

Write-Ahead Logging (“Journaling”)

Key Idea: 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