CS202: File Systems III: Crash Recovery

Logistics Reminder:
- No class on Thu (11/23) or next Tuesday (11/28)
- Lab 5 has new due date: 12/8
  → Review session: 11/27 (Monday)
    7–8 PM

- Last Class
  - Directories
  - FFS

- This Class
  - Dealing with crashes

Crash Recovery

Core Problem
- Disk writes persist
  → By design
- Disk operation only atomic at sector or block granularity
- Why Bad?

`ls > /tmp/out`

1. Allocate inode for out
   - Read inode bitmap
   - Update bitmap

2. Modify /tmp to point to allocated inode
   - Read /tmp inode to find datablock
   - Read datablock to find empty index
   - Add link to out

3. Allocate datablock for /tmp/out

4. Write to datablock
**Observation:** Ill-timed crashes can lead to
- Resource leaks
- Inconsistent file metadata
- Data loss

**Goal:** Come up with mechanisms to ensure disk consistency after crash

**But First, Things Get A Bit Worse**

**Diagram:**
- **Memory**
- **Buffer cache**
- **Program**
- **Kernel**

Modifications are not immediately written to disk.
Can force them (fsync (2)) but expensive.

Why is this a problem for crash recovery?

Recovery mechanisms:

1. Ad hoc /fsck
2. Copy on write
3. Journaling

Ad hoc /fsck

Assumption: Recovery logic will execute when a computer restarts after a crash.

Usually implemented in a program called fsck.

Goal: Ensure metadata remains consistent → no leaked data blocks or inodes.

Non-goal: Data consistency → a file's contents might
Approach: Design FS logic so that fsck can repair metadata after crash.

> ls > /tmp/out

1. **Write Data to Data Block**

2. **Write Data Block Ref To Inode**
   - For /tmp/out

3. **Update Inode Bitmap**

4. **Update Data Block Bitmap**

5. **Update /tmp To Link To**
   - /tmp/out

Please, How the Approach?
Copy-on-Write

- Core Idea: Do not modify blocks, copy instead
  (one exception: superblock/ublock)

- Can achieve both metadata & data consistency
- How? [Switch to Handout]

- Why This Works
  - UberBlock Updated Atomically
    (One Block)
  - Operation
    - Copies
  - Uber Block Update

**Commit Point**

Other Questions

a) Problems with this approach?

b) Benefits beyond crash recovery
**Journaling**

- Builds on ideas that are similar to copy on write
  - Do not modify the only copy of the filesystem
    - Need a commit point after which operation is guaranteed to appear

- Difference: reduce overheads

---

**Operation** (e.g., add a data block to file at inode 20)

1. **Describe Operation**
2. **Apply Changes**

---

**Log Requirements**
Need to be able to distinguish b/w ops that have been completely logged & those who have not.

Why might log for an op be incomplete?

How? [switch to]

Generally, a completely logged op is committed.

Log contents

Depends on type of log.

0) Redo logging

Log records how op changes

Disk

Allocate inode for out
Recovery logic applies changes from any completely logged op.

Q: When is it safe to apply changes to disk (other than during recovery)?

Undologging

Log records how to undo each step in an op.

Allocate inode for out

\[ \Rightarrow \text{value of bitmap before operation} \]
Recovery Logic: undo steps from any op that is not completely logged.

Q: When is it safe to mark that an op is completely logged?

Q: When is it safe to apply changes to the disk?

Redo vs undo logging

<table>
<thead>
<tr>
<th>Redo</th>
<th>Undo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pro</td>
<td></td>
</tr>
</tbody>
</table>
Redo + UNDO Logging
- See Notes

What To Log?

- Metadata: Inodes, Bitmaps, Etc.

- Data:

...
Figure 1: Copy-on-write filesystem: modifying a data block

(a) Initial State

(b) System allocates and creates new versions of all modified blocks.

(c) System updates Uberblock to point to new version of blocks.
Figure 2: Copy-on-write filesystem: adding a data block
(a) Initial State  

(b) System allocates and creates new versions of all modified blocks.

(c) System updates Uberblock to point to new version of blocks.

Figure 3: Copy-on-write filesystem: creating a file
Figure 4: Redo logging in a filesystem