CSE221 Lecture 13

Aronya Baksy

November 2024

1 Soft Updates: A Solution to the Metadata Update Problem

1.1 Goals

- low-cost sequencing of fine-grained updates to write-back cache blocks
- system to track and enforce file metadata updates that allows all writes (including metadata writes) to be async
- Improve file system availability and recovery time

1.2 File Metadata

- Metadata consists of pointers and descriptions for linking multiple disk sectors into files and identifying those files.
- This includes directories, inodes, free block lists etc.
- Primary invariant for metadata: must always be in a consistent state and not have dangling pointers to uninitialized space
- High performance file systems make use of write-back caching to update metadata
- **soft update**: tracks dependencies among updates to in-memory cached copies of metadata and enforces these dependencies, via **update sequencing**, as the dirty metadata blocks are written back to disk
- **Metadata Update Problem**: How to enforce rules:
  1. Never point to a structure before it is initialized
  2. Never reuse a resource before all previous pointers to it have been NULLed
  3. Never reset the last pointer to a live resource before a new pointer is set (e.g. renaming)
- Previous solutions:
  - **Synchronous writes**: force metadata update at disk speed instead of processor speed
  - **NVRAM**: requires actual hardware support for NVRAM, extra overhead for moving data to and from NVRAM, recovery is harder
  - Atomicity using **write-ahead logging** requires changes to on-disk structures, results in perf. degradation
  - **Scheduler-Enforced Ordering**: does not handle delayed writes safely when dependencies exist in metadata updates,
  - **Interbuffer Dependencies**: order enforced by cache write back code, frequent circular updates cause frequent disk synchronization

1.3 Soft Update Implementation

- Initial approach a **DAG** of dirty blocks which are written out when all dependent writes are complete, but doesn't handle circular dependencies
- dependency information is maintained at a very fine granularity: per field or pointer.
- A dirty block can be written to disk at any time, as long as any updates within the in-memory block that have pending dependencies are first temporarily rolled back - solves circular dependency problem
- Main sources of metadata updates that cause dependencies
  - **block allocation** (block ptr, free block list)
  - **block deallocation**
  - **link addition** (first update free inode list for new inode, then write inodes to disk before creating pointers to them),
  - **link removal**.

1.4 Implementation Issues

- **fsync** causes slowdowns and requires changes: soft updates code waits for all metadata updates in the disk queue to complete
- **unmount** requires finding and flushing dirty blocks: more activity than anticipated because all background tasks that are writing dirty blocks have to be identified
- **Memory usage**: mostly within existing implementations, some cases (like deleting large dir) can cause increase in mem usage which is handled by soft update code (monitors the mem usage, allows new str to be created at the same rate as old ones deleted if mem usage crosses some limits)
- **Useless writebacks** syncer daemon found to cause worst-case ordering of disk writes, modifying it to prioritize writes based on dependency info reduces writes by around 50%, also change LRU file cache replacement with dependency-based logic
- **fsck** must be modified to check that a file system is running with soft updates and clear out rather than saving unreferenced inodes in a lost/found directory

1.5 Evaluation

- **Microbenchmarks**: soft updates beat conventional approaches at creating and deleting files (reduced cost of async metadata updates is visible for small file ops, soft updates **crushes** conventional in deletes)
- Beats conventional FS in macrobenchmarks like **Postmark** (creates a pool of random text files, and then measures the time for a fixed no. of transactions) and **mail server** benchmarks.

2 SplitFS: Reducing Software Overhead in File Systems for Persistent Memory

2.1 Goals

- Split responsibility between user-space library on disk and kernel FS on **persistent memory** (PM)
- Beat existing micro and macro benchmarks for SOTA PM file systems
- Low s/w overhead
- Transparency to applications, apps need not be modified
- Minimal data copying and write I/O
- Low implementation cost, reuse existing techniques as much as possible
- Flexible choice of crash safety guarantees per app

2.2 Persistent memory

- PM will be placed on the memory bus like DRAM, accessed via processor loads and stores, has 2-4x read latency and 1/3rd (read) or 1/6th(write) b/w compared to DRAM but much higher capacity (upto 6TB).
- **split architecture**: a user-space library file system handles data operations while a kernel PM file system (ext4 DAX) handles file metadata operations.
- user-space library intercepts POSIX calls, mmaps the underlying file, and serves reads and overwrites via processor loads and stores.
- **relink** operation optimizes file appends, reduces overhead of kernel traps too: logically move extents of data from a *staging* temp file to the actual destination without actual data movements
- Optimized **logging** operations, all file updates are synchronous and atomic in **strict mode**.
- FS Modes: **POSIX** (all metadata updates are sync, appends are atomic but not sync), **sync** (all operations are sync but not atomic), **sync** mode (everything sync and atomic).

2.3 Implementation

- U-Split (user space lib linked directly with apps) and K-Split (kernel file system), read and write served directly in user spce (similar to Exokernel), metadata updates only in kernel space
- All files are mmaped (2MB surr. region), reads served via memcpy, writes via non-temporal instructions like `movnt`
- SplitFS uses temporary files called staging files for both appends and atomic data operations (appends re-linked using fsync)
- **Re-linking**:
  - logically moves PM blocks from the staging file to the target file without incurring any copies
  - ensures consistency using ext4 journalling
- Optmized Logging using checksums to check invalid log entries, compare-and-swap for atomic log updates, zero initialization, log size never grows beyond 128B (if log fills up then re-link all open files and reuse log)

2.4 Evaluation

- Syscall overhead: file data ops are faster than ext4, metadata updates slower because of extra data structures, stricter guarantees mean higher overheads
- Serving writes from user space and using staging files/relink operations for append makes these ops much faster than ext4 (2-5x)
- Max 100MB memory consumption, one physical CPU thread for execution