The Tux3 File System

Daniel Phillips

Samsung Research America (Silicon Valley)

d.phillips@partner.samsung.com

The Tux3 File System

Why Tux3?

The Local filesystem is still important!

- Affects the performance of everything
- Affects the reliability of everything
- Affects the flexibility of everything

“Everything is a file”

The Tux3 File System

But Why Tux3?

- Back to basics:
  - Data Safety
  - Performance
  - Robustness
  - Simplicity
- Advance the state of the art

The Tux3 File System

- Zumastor - enterprise NAS project
- Ddsnap - simple versioning but better than LVM
- Second generation algorithm: Versioned Pointers

  “Hey, let's build a filesystem around this!”

- Tux3 makes progress
- Community lines up behind Btrfs
- Tux3 goes to sleep for three years
- Tux3 comes back to life
- Tux3 starts winning benchmarks

The Tux3 File System

The Past: Traditional Elements

- Inode table, Block bitmaps, Directory files

The Present: Modernized Elements

- Extents, Btrees, Write anywere

The Future: Original Contributions

- New atomic commit technology
- New indexing technology
- New versioning technology

The Tux3 File System

Tux3 traditional elements

- Uniform blocks
- Block Bitmaps
- Inode table
- Index tree for file data
- Exactly one pointer to each extent
- Directories are just files

The Tux3 File System

Tux3 modern elements

- Extents
- File index is a btree
- Inode table is a btree
- Variable sized inodes
- Variable number of inode attributes
- Metadata position is unrestricted

The Tux3 File System

Tux3 advances

- Delta updates, Page Forking
  - Strong ordering
- Async frontend/backend
  - Eliminate transaction stalls
- Log/unify commit
  - Eliminate recursive copy to root
  - Resolve bitmap recursion
- Shardmap scalable index
  - A billion files per directory
- Versioned Pointers

The Tux3 File System

Inode table

1) Look up inode number in directory
2) Look up inode details in inode table

Sounds like extra work!

But...

- Due to heavy caching, does not hurt in practice
- Simplifies hard link implementation
- Concentrate on optimizing separate algorithms

The Tux3 File System

Block Bitmaps

- Competing idea: Free Extent Tree
  - Single block hole needs one bit vs 16 bytes
- Setting bits is cheap compared to finding free blocks

Delete from fragmented fs:

- Removing one file could update many bitmap blocks
- But delete is in background so front end does not care
- If fragmented, bitmap updates are the least of your worries

The Tux3 File System

Allocation

- Linear allocation is optimal most of the time!
- Cheap test to determine when linear is best
  - Otherwise go to heuristic guided search
- Maintain group allocation counts similar to Ext2/3/4
  - Allocation count table is a file just like bitmap
  - Accelerates nonlocal searches
  - Additional update cost is worth it
- No in-place update – extra challenge
- Tie allocation goal to inode number

The Tux3 File System

Log and Unify

- Log metadata changes instead of flushing blocks
  - Extent allocations
  - Index pointer updates
- Avoids recursive copy-on-write to tree root
- Periodically “Unify” logged changes to filesystem tree
  - Particularly effective for bitmap updates
- Free entire log at unify and start new
- Faster than journalling – no double write
- Less read fragmentation than log structured fs

The Tux3 File System

Atomic Commit

- Batch updates together in deltas
  - Delta transition only at user transaction boundaries
  - Gives internal consistency without analysis
- Allocate update blocks in free space of last commit
- Full ACID for data and metadata
- Bitmap recursion resolved by logging to next delta
  - Result: consistent image always needs log replay
- Always replay log on mount

The Tux3 File System

Front/Back Separation

- User filesystem transactions run in front end
- All media update work is done in back end
- Front end normally does not stall on update
- Deleting a file just sets a flag in the inode
  - Actual truncation work is done in back end
  - Even outperforms tmpfs on some loads
- SMP friendly – back end runs on separate processor
- Lock friendly – only one task updates metadata

The Tux3 File System

Block Forking

- Writing a data block in previous delta forces a copy
  - Prevents corruption of delta in flight
  - Lets frontend transactions run asynchronously
  - Side effect: Prevents changes in middle of DMA
- Key enabler for front/back separation
- Forking works by changing cache pages
  - All mmap ptes must be updated – tricky!
- Multiple blocks per page complicates it considerably

The Tux3 File System

Inode Attributes

- Variable sized inodes
- Variable number of attributes
- Variable length attributes
- Typical inode size around 100 bytes
- Easy to add more attributes as needed
- Xattrs same form as other inode attributes
- All attributes carry version tags
- Atime stamps go into separate table

Shardmap Directory Index

- Successor to HTree (Ext3/4 directory index)
- Solves scalability problems above millions of files
- Scalable hash table broken into shards
- Each shard is:
  - A hash table in memory
  - A fifo on media
- Solves the write multiplication problem
  - Only append to fifo tail on commit
- Must “rehash” and “reshard” as directory expands

The Tux3 File System

Versioned Pointers

- All version info is in:
  - Data Extent pointers
  - Inode Attributes
  - Directory Entries
- No extra complexity for physical metadata
- Still exactly one pointer to any extent or block
  - Enables “traditional” design
- Less total versioning metadata vs shared subtrees
- Potential drawback: scan more metadata

The Tux3 File System

Roadmap

Before merge:

- Allocation – resist fragmentation
- ENOSPC – Robust volume full behavior

After merge:

- FSCK and repairing FSCK
- Shardmap directory index
- Data Compression
- Versioning - snapshots

Questions?

Daniel Phillips

Samsung Research America (Silicon Valley)

d.phillips@partner.samsung.com