(Not so) recent development in filesystems

Tomáš Hrubý

University of Otago and World45 Ltd.

March 19, 2008

Ext2 - de facto standard

- Based on MinixFS and BSD FFS
- Linux native FS
- Disk split in blocks and block-groups
- Block-groups reduce external fragmentation, contain super-block, free blocks bitmap, inodes and data blocks
- All inodes are allocated during Ext2 creation (usually 5% of volume size)
- Direct blocks and 2 levels of indirect blocks (file size limit)

Ext2 contd

- Spread inodes of unrelated directories among different block-groups
- Subdirs of root are spread over all groups
- Keep subdirectories in the same group as their parent if the group has space
- Files are kept *as close as possible* to the directory (heuristic)
- Large hashed directories (HTrees) to speedup dir ops
- Backward compatible

VFS

Extended filesystem introduced *VFS* interface in Linux

- Ext2 tailored (e.g. expects inodes)
- Super operations
  `sync_fs() alloc_inode() put_inode() ...`
- Directory operations
  `create() link() unlink() rename() ...`
- File operations `open() close() trunc() ...`
- `vinode`, `dentry` and other objects to make the VFS abstraction
- Address space operations (`prepare_write`, `commit_write`, ...)

Other operations to speed up work with files implemented separately
`splice() sendfile() ...`

Ext3 - journaled Ext2

- Much nicer code that uses `page` objects instead of deprecated buffer heads
- Ext3 mountable as Ext2 and adding journal to Ext2 is trivial too
- 3 journaling strategies
- No need to check the whole fs after crash

Ext3

Generic journaling layer *journaling block device* (JBD) keeps journal in *.journal* file

Journaling strategies :

- *Journal* - all data and metadata (duplication of data)
- *Ordered* - Only metadata logged, data written always before metadata
- *Writeback* - fastest, only metadata logged

Ext4 - Ext3 fork

- Addresses scalability
- Filesystem for large drives (overcoming 16TiB limit of Ext3)
- 48-bit => JDB2 supports larger then 32-bit values
- Metablock groups (clusters of block-groups - present in Ext3) to disperse block descriptors

Extents

- substitute for indirect blocks
- break forward compatibility between Ext3 and Ext4
- each covers up to 128MiB of contiguous space
- minimize fragmentation and truncate times
- constant depth tree of extents (like HTree)
- good for seq access, bad for random

Ext4 - contd

- Metadata checksumming - easier corruption detection
- Persistent preallocation for contiguous writes
- Delayed allocation postponed till page flush time
  Blocks are allocated in batches and none are allocated for short-lived files
- Online defragmentation
- It's still a work in progress and it's not ready for production systems

XFS - journaling by SGI

Journal

- Circular buffer
- Internal log in XFS data section or on a separate device
- Log of *logical* operations performed (only metadata)
- Unwritten data-blocks prior crash are zeroed after recovery
- Similar to Ext3 *writeback* mode

Blocks management

- Allocation groups similar to Ext2 block-groups, B-tree of inodes
- Variable size extents (*large writes*) managed by 2 B+trees index by length and first free block
- Allow parallel access to data structures
- Delayed allocation
- Rotorstep - when to move allocation within a file to next AG

LFS - Beyond journaling

Originally proposed by Rosenblum and Ousterhout in 1991

- In-memory disk cache is huge $\Rightarrow$ most updates in memory
- Random writes are slow $\Rightarrow$ large sequential writes
- All data are placed sequentially in an *infinite* log
- Disk is not *infinite* $\Rightarrow$ a garbage collector is required
- Segments and partial segments
- Variable number of inodes in `.ifile` in root directory
- *Everything* is journaled, which results in fast crash recovery
- Implementing read-only snapshots is trivial
- Journal within the log to inform the roll-forward utility about directory operations

Filesystems for NAND flash memories

Another use of Log-structured file systems

- NAND page must be written at once, writing to a clean page is simpler than read-clean-modify-writeback
- Writes are damaging pages $\Rightarrow$ wear levelling
- JFFS & JFFS2
  - To make wear levelling fair, garbage collector can occasionally move clean blocks as well
  - The whole fs must be scanned at mount time
- YAFFS & YAFFS2
  - Also keeps a tree of blocks in RAM
  - Version 2 uses checkpointing to avoid scanning at mount time
- LogFS
  - A new project that is motivated by rising size of SSDs
  - Wasn't originally designed as log, but . . .

XFS transactions

Still not transactions in database-like sense

- Transaction per inode and operation
- Allocates required space beforehand
  - Allocating thread cannot sleep
  - Linux does not like allocation of many contiguous MiBs
  - In order to flush dirty pages allocation might be required!!!
- Makes changes
- Writes inode and other info (e.g., superblock) to the log
- Commits changes to data area

sync(2) optimization - writes metadata to log, not necessarily to fs

NTFS transactions - Vista

Beyond low level transactions (journaling)

- Atomic operations on a *single file* - preventing corrupted files when application crashes while updating a file
- Atomic operations spanning *multiple files* - if a collection of files must be updated and consistency is an issue

ZFS : the last word in file systems

Developed by SUN in 2004

- 128-bit (should be enough for some time ;-)
- Uses virtual storage pools to span more disks (no volume mgr)
- End-to-end checksumming - upper structures contain checksum of lower structures, checked on every access!
- Copy-on-write transactional model (do all changes or nothing)
- Mimics LogFSs without need of garbage collector
- Simple implementation of snapshots and writable clones
- Dynamic striping automatically expands to new devices
- Variable blocksizes

ZFS : the last word in file systems

Transactions

- All file-system level operations on virtual disks
- Operations are grouped in transactional objects
- All interactions occur through Data Management Unit (DMU)
- All transactions through DMU are atomic $\Rightarrow$ data always consistent
- ZFS internally keeps an intentions log (ZIL)
- In case of power outage, COW keeps old data till the end of the update operation

Interface

- ZPL - POSIX layer
- ZVOL - *raw* virtual device backed by ZFS

BTRFS - ZFS by Oracle

- Started in 2007, still in early stage
- Looks like reworking ZFS (similar set of features)
- Everything is a B+Tree
- Groups all items of an object in the same part of the B+Tree
- Different indexing of directories for `readdir()` and other ops
- Back references for easy validation and faster corruption recovery
- CRFS (consistent NFS) uses Btrfs as on-disk system

FUSE - writing FS in userspace

- Primarily for virtual file systems
  (e.g, GmailFS, WikipediaFS, iTuneFS, . . .)
- used by NTFS-G3 driver, considered for ZFS Linux port
- Very thin kernel layer relays fs syscalls to the userspace driver
- Driver is an executable linked with FUSE library
- Access via `/dev/fuse` file
- Multiple mounts with different file descriptors

UnionFS - why?

- Union mount of filesystems in Linux
- Transparent overlay of files and directories of separate file systems (branches)
- A single coherent file system
- Content of directories with the same path is merged
- Each branch has a priority (for lookups)
- Writes to read-only files are redirected to highest-priority writable branch
- Read-only root on Live-CDs can be changed in tmpfs
- Still work in progress, not in mainline kernel

UnionFS - how?

- A stackable FS (virtual - does not store data itself)
- Unlike BSD, Linux does not have any generic stackable layer
- Based on FiST template language (to ease porting, like eCryptFS)
- Implements functionality of VFS and acts as VFS in the same time
- Problems with coherency if the lower fs is used to change data
- Uses a separate partition or loopback device to store persistent information instead of in the branches (ODF)
- ODF is generic for stackable but used only by UnionFS, more or less like JBD is used only by Ext3
- As a result this allows stacking of UnionFS on top of itself

What if FS gets corrupted?

Problems

- We have fsck (or similar) ... which takes ages
  reported `xfs_repair` runtime of up to 8 days!
- Size of disks is growing mush faster than their speed
- Some FSs can heal themselves in run time (from mirrors)
- Repair utility may need to read all FS objects

What next?

1. Repair must get smarter
2. File systems must be redesigned for easier repair

XFS troubles

Problems

- IRIX with many slow CPUs with good I/O throughput
- Allocation groups can be processed in parallel
- Smart prefetching of objects and data blocks associated with processed inodes (fast until OOM happens)
- Both makes it even slower on Linux! (2-3x faster CPU bad I/O)
- Exploiting metadata patterns - contiguous, lots of single blocks
- Bad I/O patterns - backward seeks, long seeks

Solution

1. Prefetch threads use bandwidth instead of seeks, large I/O, throw away non-metadata
2. Unified cache for all phases $\Rightarrow$ no purge

ChunkFS : repair driven FS design

Problem

- `xfs_repair` and `e2fsck` improvement erased in 1-2 years

Solution

1. Incremental check, checksums, redundancy, metadata isolation
2. Divide FS into metadata isolation groups (chunks)
3. Continuation inodes when files outgrow chunks (dirs are files!)
4. Smart and sparse allocation limits the number of continuation inodes
5. Fixing only broken chunks and following backward links to update other chunks

Research still in progress, proof-of-concept based on Ext2 exists!

Thank you for your attention

Questions ...