



Silicon Graphics, Inc.

XFS Overview & Internals

09 - Internals

© Copyright 2006 Silicon Graphics Inc. All rights reserved.
Permission is granted to copy, distribute, and/or modify this document
under the terms of the Creative Commons Attribution-Share Alike, Version
3.0 or any later version published by the Creative Commons Corp. A copy of
the license is available at <http://creativecommons.org/licenses/by-sa/3.0/us/>.

November 2006

XFS Slides 09 - Internals



XFS Internals

- Understand some of the unique features in XFS
- Why XFS differs from other Linux filesystems
- How these differences are implemented

xfs_vnodeops

- VFS interfaces are mapped to an IRIX-like vnode operations table
 - fs/xfs/xfs_vnodeops.c

```
xfs_vnodeops {  
    • open, close, fid, read, write, sendfile, splice, fsync  
        – file descriptors  
    • getattr, setattr  
        – inode attributes - stat(2)  
    • attr_get, attr_set, attr_list, attr_remove  
        – extended attributes  
    • access, lookup  
        – inode permissions/existence  
    • create, remove, symlink, readlink  
        – regular files, special files  
    • readdir, mkdir, rmdir, link, rename  
        – directories  
    • reclaim, release, inactive, iflush, bmap, flush_pages, flush_inval_pages, toss_pages  
        – inode / page cache state and/or lifecycle  
};
```

xfs_ioctl

- XFS specific system calls (`xfsctl()`) are dispatched by `xfs_ioctl()`
 - `fs/xfs/xfs_fs.h`
 - `fs/xfs/linux-2.6/xfs_ioctl.c`
- Can be exercised with `xfs_io`
- geometry, fscounts, [get|set]resblks, shutdown, freeze/thaw
 - filesystem level manipulation
- grow[fs|fslog|fsrt]
 - filesystem size (and maximum inode count) expansion
- [get|set]xflags, fs[get|set]xattr, fs[get|set]xattra, dioinfo
 - inode attribute information
 - direct I/O parameters (min/max/align)
- allocsp, freesp, resvsp, unresvsp
 - space allocation and/or preallocation
- bulkstat
 - many (sequential) inode's attributes – `stat(2)`
- xfsdump, quotacheck, dmapi
 - by-handle (open, fd-to-, path-to-, readlink, attrlist, attrmulti, ...)
 - manipulating inodes by “handles” (inum/igen/fsid)
- getbmap, getbmapa, swapext
 - inode data/attr fork extent information

xfs_ioctl – Miscellaneous

- geometry
 - Data displayed by xfs_info
- fscounts
 - XFS specific stat information
- resblks
 - allows dmapi to set aside some disk space
 - threads can be marked to say they can use it if about to run out of space
- grow
 - upward only
 - can't grow the log, not implemented in the kernel
 - also can be used to change the maximum space for inodes
- dioinfo changes direct I/O parameters
 - max direct I/O size is huge, no real limit

xfs_ioctl - Attribute Flags

- xflags
 - inode flags, see definition of `struct fsxattr`
- xattr
 - original IRIX version for inode flags
 - project id
 - extent size hint
 - how many extents are allocated on the data fork
- xattr passes out the same structure as xattr, but applies to the attribute rather than data fork.

xfs_ioctl - Space Allocation

- allocsp/freesp
 - allocate space
 - zeros
 - updates the inode size
- resvsp/unresvsp is for allocating but not zeroing
 - efficient way for applications to take advantage of unwritten extents
- posix_fallocate has no generic interface to call into the kernel
 - current implementation just writes zeros from user space

xfs_ioctl - Bulkstat

- bulkstat returns multiple inodes
 - scans entire filesystem, no way to start at a particular directory
 - hence xfsdump cannot dump a directory
 - flags to pass in to use incore or ondisk inode clusters
- byhandle interfaces
 - handles usually obtained from bulkstat
 - a handle is the combination of inum, igen and fsid
 - see DMAPI for more information on byhandle interfaces

XFS sysctls - Daemons

- fs.xfs.xfssyncd_centisecs (Min: 100 Default: 3000 Max: 720000)
 - The interval at which the xfssyncd thread flushes metadata out to disk. This thread will flush log activity out, and do some processing on unlinked inodes.
- fs.xfs.xfsbufd_centisecs (Min: 50 Default: 100 Max: 3000)
 - The interval at which xfsbufd scans the dirty metadata buffers list.
- fs.xfs.age_buffer_centisecs (Min: 100 Default: 1500 Max: 720000)
 - The age at which xfsbufd flushes dirty metadata buffers to disk.

XFS sysctls - Debug

- `fs.xfs.error_level` (Min: 0 Default: 3 Max: 11)
 - A volume knob for error reporting when internal errors occur.
 - This will generate detailed messages & backtraces for filesystem shutdowns, for example.
 - Current threshold values are:
 - `XFS_ERRLEVEL_OFF`: 0
 - `XFS_ERRLEVEL_LOW`: 1
 - `XFS_ERRLEVEL_HIGH`: 5
- `fs.xfs.panic_mask` (Min: 0 Default: 0 Max: 127)
 - Causes certain error conditions to call `BUG()`. Value is a bitmask;
 - AND together the tags which represent errors which should cause panics:

• <code>XFS_NO_PTAG</code>	0
• <code>XFS_PTAG_IFLUSH</code>	0x00000001
• <code>XFS_PTAG_LOGRES</code>	0x00000002
• <code>XFS_PTAG_AILDELETE</code>	0x00000004
• <code>XFS_PTAG_ERROR_REPORT</code>	0x00000008
• <code>XFS_PTAG_SHUTDOWN_CORRUPT</code>	0x00000010
• <code>XFS_PTAG_SHUTDOWN_IOERROR</code>	0x00000020
• <code>XFS_PTAG_SHUTDOWN_LOGERROR</code>	0x00000040
- This option is intended for debugging only.

XFS sysctls - Compatibility

- `fs.xfs.irix_symlink_mode` (Min: 0 Default: 0 Max: 1)
 - Controls whether symlinks are created with mode 0777 (default) or whether their mode is affected by the umask (irix mode).
- `fs.xfs.irix_sgid_inherit` (Min: 0 Default: 0 Max: 1)
 - Controls files created in SGID directories
 - If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs of the parent dir, the ISGID bit is cleared if the `irix_sgid_inherit` compatibility sysctl is set.

XFS sysctls – Attribute Inheritance

- `fs.xfs.inherit_sync` (Min: 0 Default: 1 Max: 1)
 - Setting this to "1" will cause the "sync" flag set by the `xfs_io(8)` chattr command on a directory to be inherited by files in that directory.
- `fs.xfs.inherit_nodump` (Min: 0 Default: 1 Max: 1)
 - Setting this to "1" will cause the "nodump" flag set by the `xfs_io(8)` chattr command on a directory to be inherited by files in that directory.
- `fs.xfs.inherit_noatime` (Min: 0 Default: 1 Max: 1)
 - Setting this to "1" will cause the "noatime" flag set by the `xfs_io(8)` chattr command on a directory to be inherited by files in that directory.
- `fs.xfs.inherit_nosymlinks` (Min: 0 Default: 1 Max: 1)
 - Setting this to "1" will cause the "nosymlinks" flag set by the `xfs_io(8)` chattr command on a directory to be inherited by files in that directory.

XFS sysctls - Misc

- `fs.xfs.stats_clear` (Min: 0 Default: 0 Max: 1)
 - Setting this to "1" clears accumulated XFS statistics in `/proc/fs/xfs/stat`. It then immediately resets to "0".
- `fs.xfs.restrict_chown` (Min: 0 Default: 1 Max: 1)
 - Controls whether unprivileged users can use `chown` to "give away" a file to another user.
- `fs.xfs.rotorstep` (Min: 1 Default: 1 Max: 256)
 - In "inode32" allocation mode, this option determines how many files the allocator attempts to allocate in the same allocation group before moving to the next allocation group. The intent is to control the rate at which the allocator moves between allocation groups when allocating extents for new files.

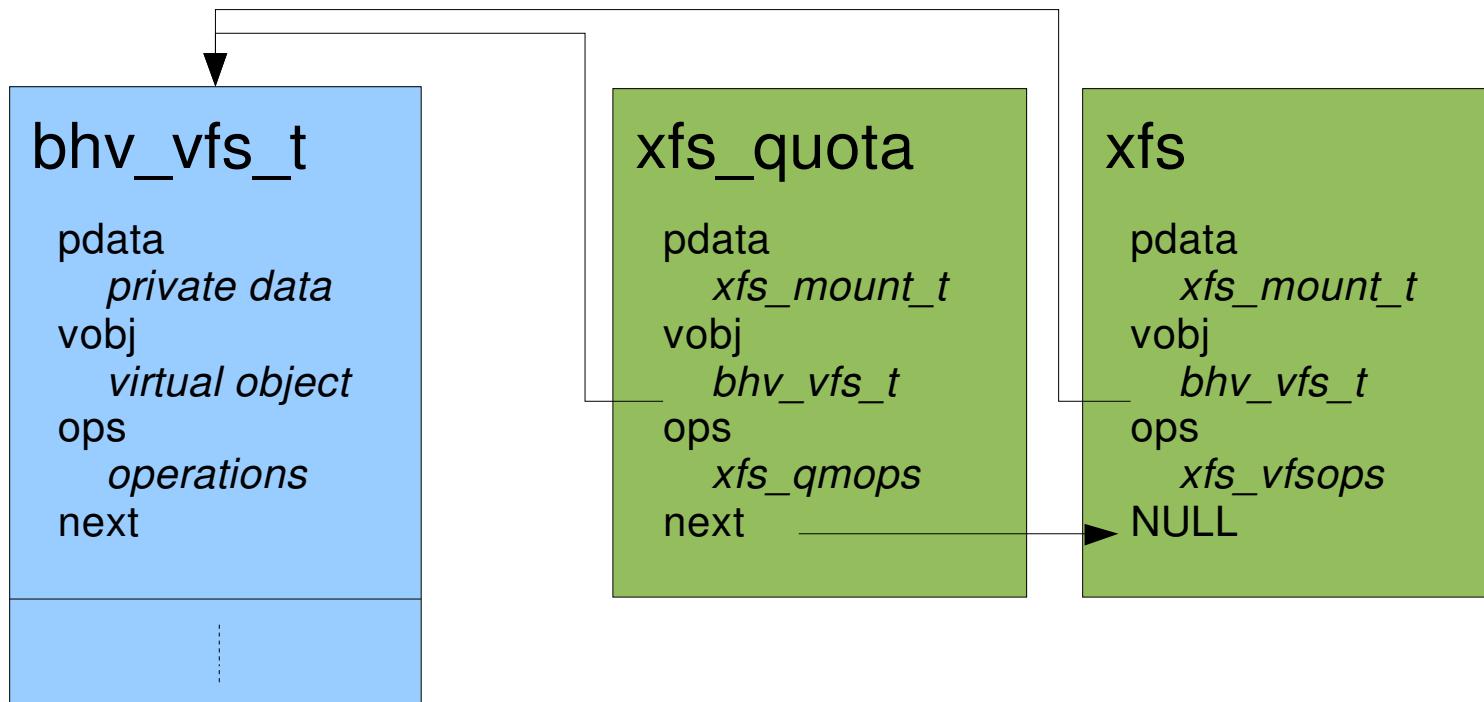
Generic sysctls

- dentry-state
 - Number of directory entries
 - Number of unused entries
 - Reclaim >secs when short on memory
- file-max
 - Maximum number of files system wide
- file-nr
 - # files allocated
 - Number of files in use
 - Max number of files system wide
- inode-state
 - Number of active inodes
 - Number of free inode entries

Behaviours

- A behaviour chain is a way to chain different operations.
- The filesystem call stack typically
 - enter the linux vfs level
 - calls the xfs equivalents which do little work
 - they call the behaviour VOP
- Each operation can choose to call the next operation in the chain or return
 - it is not strictly a stack
- Each behaviour has an index which is where it wants to sit in the chain.
- CXFS, dmapi and quotas use this to "intercept" requests.
- New XFS features may add additional behaviours

Behaviours



Mount Path

- Mounting an XFS filesystem has several steps
 - `xfs_fs_fill_super` in `xfs_super.c`
1. Allocate a `bhv_vfs` struct (`vfs_allocate`)
 2. Setup initial behaviour module chain (`bhv_insert_all_vfsops`)
 3. Parse mount options (`bhv_vfs_parseargs`)
 4. Perform mount (`bhv_vfs_mount`)

Mount – Setup Behaviour Chain

- See `bhv_insert_all_vfsops` in `xfs_vfs.c`
- CXFS, DMAPI and Quotas can register with XFS so that their vfsops are loaded into the behaviour chain
- This does module loading and reference counting
- Each behaviour can also have custom operations that need to be loaded
 - Specialised callouts in XFS to specific behaviour modules, if loaded
 - These are loaded in `xfs_mount` once we know module will be used

Mount – Parse Mount Options

- Each behaviour can grab options that are used by that behaviour module
 - XFS does not need to know or understand additional mount options
- Behaviours may then remove themselves from the chain if they are not needed
 - no dmi option so no need for dmapi
 - no quota options so no need for quotas
 - see `bhv_remove_vfsops`

Mount – Actual Filesystem Mount

- Implemented in `xfs_mount` in `xfs_vfsops.c`
- Loads specialised behaviour operations for those modules in use
- Opens up the log and realtime device
- `xfs_alloc_buftarg` starts a kernel thread for delayed write buffers for each device
- XFS then call `xfs_start_flags` that starts setting up the `xfs_mount_t` structure, everything we can do before doing I/O from the mount options
- Then call `xfs_readsb` to read the super block and then `xfs_finish_flags` to compare the second part of the mount options to what was in the super block

Mount – Actual Filesystem Mount

- Tell the buffers what the sector size is going to be with `xfs_setsize_bufarg`
- Check to see if this device supports write barriers
- Finally we call `xfs_ioinit`, which is a behaviour custom operation
 - this one will call into CXFS if loaded for this filesystem
- `xfs_ioinit` calls `xfs_mountfs` to complete the mount

Mount – xfs_mountfs

- Calls xfs_mount_common sets up additional mount_t fields from superblock
- Check the end of the filesystem really does exist for each device
- Initialise various data structures
 - Number of read ahead buffers based on physical memory
 - inode and stripe alignment
 - allocate and initialise inode hash for this filesystem
- Validate root inode and link into super block
- Set up the transactions and log, and do log recovery
- Call out to quota custom ops to do the quota check

Transactions

- Transactions are used to record metadata changes to the filesystem
- Each transaction is an atomic change to the filesystem
- Only in the core of XFS, not in the higher layers of XFS
- After reserving space for a transaction, it is very difficult to cancel the transaction
 - Calling routines will try to pull inodes incore, setup `xfs_inode_t` and dquot to avoid problems from here on
 - They must have references on any objects to be attached to the transaction

Creating Transactions

- A transaction is typically coded as

- Create the incore structure for the transaction

```
tp = xfs_trans_alloc(type);
```

- Reserve space for the transaction, quick lookup in `mount_t` structure

- can return ENOSPACE

- reserved space can be large to cater for large ondisk structure changes

- exceeding the reservation will cause XFS to dump a lot of diagnostics before shutting down

```
error = xfs_trans_reserve(tp, data, log, rt, ...);
```

- Now make changes, allocate space, free space, etc.

- Attach superblock/inode(s)/buffers etc, log ranges within these objects, typically via

```
xfs_trans_log_inode(tp, ip, XFS_ILOG_CORE);
```

- Commit the transaction, copying data from that attached objects

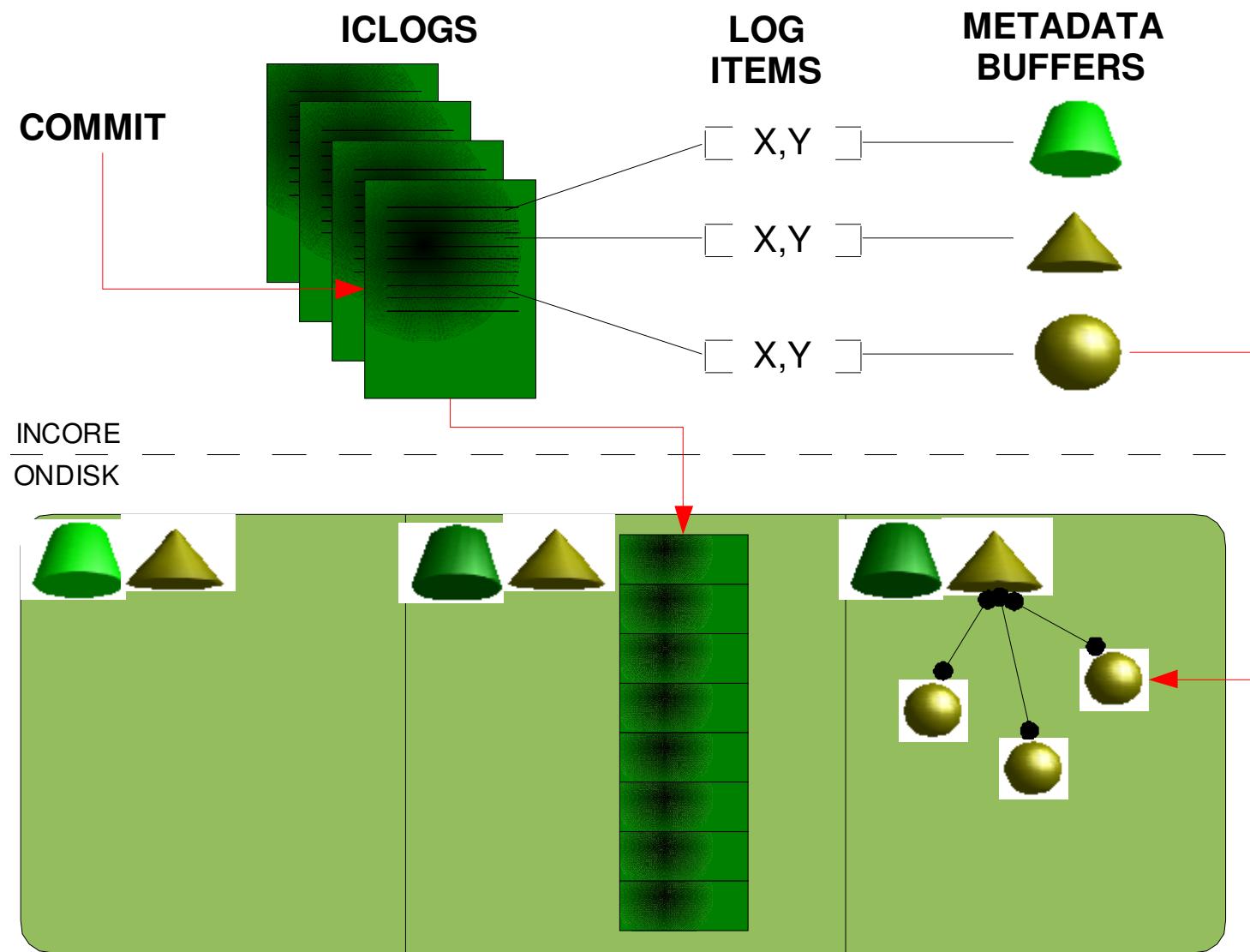
```
error = xfs_trans_commit(tp);
```

In Core Logs

- There are normally 8 in-core log buffers (`iclogs`)
 - depends on memory of system
 - can be set by a mount option.
- An in-core log is written where
 - it fills up
 - you get a synchronous op, more transactions are asynchronous
 - a sync the incore buffer is written
- When XFS receives an I/O completion XFS can unpin the first metadata buffers
 - Once unpinned they can be written to disk
- The active item list (AIL) is used to prevent the metadata buffers from being written multiple times if it is in multiple transactions.
- Therefore, transactions and metadata buffers have a lifecycle.

Log Sequence Numbers

- An LSN is the log sequence number
 - 64bit with two 32bit values
 - the first is the cycle number
 - the second is the block number
- The block number is assigned when it is committed
- The cycle number is incremented each time we have cycled through the log
- The metadata in the log is only the metadata that has changed, in theory, but the whole inode tends to be logged.
- One of the flaws of the xfs logging, is that everything that changed is logged even if sometime later we changed that field again, we still log the original change.
 - XFS holy grail is to avoid duplicates in the log



xfs_bmapi

- Many routines within XFS have to allocate ondisk space
 - metadata
 - inodes
 - extents
- This is all done through xfs_bmapi
 - access extent map for reading
 - setup delayed allocation
 - perform actual allocation
 - convert unwritten extents to written extents
- If you trace calls through XFS xfs_bmapi is the centre of XFS
- In write mode you are usually in a transaction
 - except when doing delayed allocation, transaction pointer will be null

xfs_bmap_alloc

- xfs_bmap_alloc is the switch between the different allocators
 - normal (xfs_bmap_btalloc)
 - realtime (xfs_bmap_rtalloc)
 - filestreams
- Realtime uses two bitmaps
 - one for free space
 - one for larger clusters of freespace
- Quota accounting is also done in bmap routines
- inodes have two pointers to dquot's
 - one for the user
 - one for the group project

Memory Allocation

- IRIX was very good at ensuring memory allocations succeeded
 - XFS was written on IRIX
- Linux is not as good with memory allocations, so this has been the source of many XFS problems

```
# head -2 /proc/slabinfo; grep -i xfs /proc/slabinfo  
# slabtop
```

Memory allocations for transactions

- If it is in a transaction, the memory allocation code will behave differently to try to ensure this critical thread does not sleep
 - This flag is a generic linux change
- In some cases you could ask for many MBs of contiguous space
 - Linux does not like this.
- You could have a whole lot of dirty page cache pages but in order to flush those out you have to call back into XFS which can allocate memory.
- Recent changes to XFS means we are much better at managing memory allocation and helps us to avoid these issues
 - 2.6.17 and should be included in SLES10SP1

Metadata Buffering

- `xfs_buf.{h,c}` implements `xfs_buf_t` the XFS metadata buffer cache
 - Multi-page buffers
 - Buffer “pinning”
 - Prevent them from being written even if “dirty”
 - Transaction log must be written first
 - Several “private” buffer pointers
 - Locking, iodone semaphore for I/O waiters
 - Callbacks for: iodone, relse, pre-write
- In-core log buffers also implemented via `xfs_buf_t`
- This causes some oddities since they use
 - sub-buffer-sized I/Os
 - non-page-cache buffers
- Separate address space from bdev
 - Prevents user space buffering from impacting metadata buffering

Metadata I/O Completion

- xfslogd/N
 - per-CPU daemon
 - Threads that handle I/O completion work for iclog buffers
 - xlog_state_do_callbacks
 - multiple completions to unpin metadata buffers waiting for this transaction
 - and also metadata
 - xfs_buf_do_callbacks
 - typically, removing from AIL and freeing up buffer_item memory
- xfsdatad/N
 - similar idea for data path
 - very different between 2.4 and 2.6

Delayed write buffers

- xfsbufd
 - kernel thread, one per filesystem device
 - buffers are time stamped when queued
 - xfsbufd walks the *xfs_buftarg_t* (“buffer target”) hash table finding delayed write buffers
 - default is every 5 seconds look for buffers more than 30 seconds old
 - metadata and log data specific, file data is handled differently
- Tweak the age at which unpinned and dirty metadata buffers will be considered for flushing
 - `/proc/sys/fs/xfs/age_buffer_centisecs`
- Tunable daemon wakeup interval
 - `/proc/sys/fs/xfs/xfsbufd_centisecs`

I/O Path

- Has two parts
 - code that handles read and write calls
 - both buffered and direct I/O
 - `xfs_lrw.c`
 - code that actually writes something out
- Uses Linux `get_block_t` interfaces and `struct buffer_head`
- XFS code is different to other filesystems because of
 - different locking
 - delayed allocation
 - dmapi

I/O Path - Locking

- There are two locks within the linux inode
 - the most interesting is the i_mutex
 - the second and inner most lock XFS does not use
- XFS will hold it for the entire buffered write call
- This is not the case for direct I/O
 - ext3/reiserfs will hold the i_mutex for the whole direct I/O which will serialise their direct I/O path.
- i_mutex is often taken outside of xfs as well
- XFS also has the iolock and ilock mrlocks on the xfs inode
- Must be taken in this order
 - i_mutex
 - iolock
 - ilock

I/O Path - DMAP Integration

- DMAP requires extra hooks for DMAP events on read and writes to a file
 - This code includes lots of branches and is not coded for to be enabled at compile time
- DMAP also needs to perform *invisible* I/O to remove and replace the file data without changing the inode access/change/modification times

I/O Path – Delayed Allocation

- Initial write reserves space only
 - Must ensure that when write to disk occurs there is space available
- Allocation of real extents occurs when data is actually being written
 - Data can be coalesced into much larger I/Os to disk
 - Allows allocator to allocate much larger extents than just individual I/O request sizes from the application

sync(2)

- XFS implements an optimization to sync(2) of metadata:
 - XFS will only force the log out, such that any dirty metadata that is incore is written to the log only, the metadata itself is not necessarily written
 - This is safe, since all change is ondisk
 - File data is guaranteed too (even barriers)
- Log and metadata are written to disk for
 - freeze/thaw
 - remount ro
 - unmount
- Applications like **grub** have been bitten in the past, but fixed nowadays

Data writeout

- Triggered by the VM subsystem calling into XFS
 - `xfs_aops.c`
 - `xfs_vm_writepage(s)`
 - `xfs_page_state_convert`
- Page cache pages attached to inodes via a radix-tree (2.6)
 - `inode->i_mapping`
 - `page->mapping`
- XFS does its own writeout, sort of
 - due to delayed allocation and unwritten extents
 - extent conversion requires a transaction
- Unlike write page, read path is mostly generic linux implementation

XFS writepage

- Rewritten in 2.6
- XFS tries very hard to cluster pages
 - `xfs_add_to_ioend`
 - `xfs_cluster_write`
- This allows XFS to do a much smaller number of extent conversions
 - rather than converting for every buffer head
- Uses `struct bio` for writing more than 1 page at a time

