```
**********************************************************
  135192 Thu May 16 17:36:16 2013
new/usr/src/uts/common/fs/zfs/arc.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

/*
 * DVA-based Adjustable Replacement Cache
 *
 * While much of the theory of operation used here is
 * based on the self-tuning, low overhead replacement cache
 * presented by Megiddo and Modha at FAST 2003, there are some
 * significant differences:
 *
 * 1. The Megiddo and Modha model assumes any page is evictable.
 * Pages in its cache cannot be "locked" into memory.  This makes
 * the eviction algorithm simple: evict the last page in the list.
 * This also make the performance characteristics easy to reason
 * about.  Our cache is not so simple.  At any given moment, some
 * subset of the blocks in the cache are un-evictable because we
 * have handed out a reference to them.  Blocks are only evictable
 * when there are no external references active.  This makes
 * eviction far more problematic:  we choose to evict the evictable
 * blocks that are the "lowest" in the list.
 *
 * There are times when it is not possible to evict the requested
 * space.  In these circumstances we are unable to adjust the cache
 * size.  To prevent the cache growing unbounded at these times we
 * implement a "cache throttle" that slows the flow of new data
 * into the cache until we can make space available.
 *
 * 2. The Megiddo and Modha model assumes a fixed cache size.
 * Pages are evicted when the cache is full and there is a cache
 * miss.  Our model has a variable sized cache.  It grows with
 * high use, but also tries to react to memory pressure from the
 * operating system: decreasing its size when system memory is
```



```
 * tight.
 *
 * 3. The Megiddo and Modha model assumes a fixed page size. All
 * elements of the cache are therefore exactly the same size.  So
 * when adjusting the cache size following a cache miss, its simply
 * a matter of choosing a single page to evict.  In our model, we
 * have variable sized cache blocks (rangeing from 512 bytes to
 * 128K bytes).  We therefore choose a set of blocks to evict to make
 * space for a cache miss that approximates as closely as possible
 * the space used by the new block.
 *
 * See also:  "ARC: A Self-Tuning, Low Overhead Replacement Cache"
 * by N. Megiddo & D. Modha, FAST 2003
 */

/*
 * The locking model:
 *
 * A new reference to a cache buffer can be obtained in two
 * ways: 1) via a hash table lookup using the DVA as a key,
 * or 2) via one of the ARC lists.  The arc_read() interface
 * uses method 1, while the internal arc algorithms for
 * adjusting the cache use method 2.  We therefore provide two
 * types of locks: 1) the hash table lock array, and 2) the
 * arc list locks.
 *
 * Buffers do not have their own mutexes, rather they rely on the
 * hash table mutexes for the bulk of their protection (i.e. most
 * fields in the arc_buf_hdr_t are protected by these mutexes).
 *
 * buf_hash_find() returns the appropriate mutex (held) when it
 * locates the requested buffer in the hash table.  It returns
 * NULL for the mutex if the buffer was not in the table.
 *
 * buf_hash_remove() expects the appropriate hash mutex to be
 * already held before it is invoked.
 *
 * Each arc state also has a mutex which is used to protect the
 * buffer list associated with the state.  When attempting to
 * obtain a hash table lock while holding an arc list lock you
 * must use: mutex_tryenter() to avoid deadlock.  Also note that
 * the active state mutex must be held before the ghost state mutex.
 *
 * Arc buffers may have an associated eviction callback function.
 * This function will be invoked prior to removing the buffer (e.g.
 * in arc_do_user_evicts()).  Note however that the data associated
 * with the buffer may be evicted prior to the callback.  The callback
 * must be made with *no locks held* (to prevent deadlock).  Additionally,
 * the users of callbacks must ensure that their private data is
 * protected from simultaneous callbacks from arc_buf_evict()
 * and arc_do_user_evicts().
 *
 * Note that the majority of the performance stats are manipulated
 * with atomic operations.
 *
 * The L2ARC uses the l2arc_buflist_mtx global mutex for the following:
 *
 *      - L2ARC buflist creation
 *      - L2ARC buflist eviction
 *      - L2ARC write completion, which walks L2ARC buflists
 *      - ARC header destruction, as it removes from L2ARC buflists
 *      - ARC header release, as it removes from L2ARC buflists
 */
```

```
121 #include <sys/spa.h>
122 #include <sys/zio.h>
123 #include <sys/zfs_context.h>
124 #include <sys/arc.h>
125 #include <sys/refcount.h>
126 #include <sys/vdev.h>
127 #include <sys/vdev_impl.h>
128 #ifdef _KERNEL
129 #include <sys/vmsystm.h>
130 #include <vm/anon.h>
131 #include <sys/fs/swapnode.h>
132 #include <sys/dnlc.h>
133 #endif
134 #include <sys/callb.h>
135 #include <sys/kstat.h>
136 #include <zfs_fletcher.h>

138 #ifndef _KERNEL
139 /* set with ZFS_DEBUG=watch, to enable watchpoints on frozen buffers */
140 boolean_t arc_watch = B_FALSE;
141 int arc_procfd;
142 #endif

144 static kmutex_t         arc_reclaim_thr_lock;
145 static kcondvar_t       arc_reclaim_thr_cv;     /* used to signal reclaim thr */
146 static uint8_t          arc_thread_exit;

148 extern int zfs_write_limit_shift;
149 extern uint64_t zfs_write_limit_max;
150 extern kmutex_t zfs_write_limit_lock;

152 #define ARC_REDUCE_DNLC_PERCENT 3
153 uint_t arc_reduce_dnlc_percent = ARC_REDUCE_DNLC_PERCENT;

155 typedef enum arc_reclaim_strategy {
156         ARC_RECLAIM_AGGR,               /* Aggressive reclaim strategy */
157         ARC_RECLAIM_CONS                /* Conservative reclaim strategy */
158 } arc_reclaim_strategy_t;
______unchanged_portion_omitted_

375 #define ARCSTAT(stat)   (arc_stats.stat.value.ui64)

377 #define ARCSTAT_INCR(stat, val) \
378         atomic_add_64(&arc_stats.stat.value.ui64, (val))
378         atomic_add_64(&arc_stats.stat.value.ui64, (val));

380 #define ARCSTAT_BUMP(stat)      ARCSTAT_INCR(stat, 1)
381 #define ARCSTAT_BUMPDOWN(stat)  ARCSTAT_INCR(stat, -1)

383 #define ARCSTAT_MAX(stat, val) {                                        \
384         uint64_t m;                                                    \
385         while ((val) > (m = arc_stats.stat.value.ui64) &&              \
386             (m != atomic_cas_64(&arc_stats.stat.value.ui64, m, (val)))) \
387                 continue;                                              \
388 }
______unchanged_portion_omitted_

575 static buf_hash_table_t buf_hash_table;

577 #define BUF_HASH_INDEX(spa, dva, birth) \
578         (buf_hash(spa, dva, birth) & buf_hash_table.ht_mask)
579 #define BUF_HASH_LOCK_NTRY(idx) (buf_hash_table.ht_locks[idx & (BUF_LOCKS-1)])
580 #define BUF_HASH_LOCK(idx)      (&(BUF_HASH_LOCK_NTRY(idx).ht_lock))
581 #define HDR_LOCK(hdr) \
582         (BUF_HASH_LOCK(BUF_HASH_INDEX(hdr->b_spa, &hdr->b_dva, hdr->b_birth)))
```



```
584 uint64_t zfs_crc64_table[256];

586 /*
587  * Level 2 ARC
588  */

590 #define L2ARC_WRITE_SIZE        (8 * 1024 * 1024)       /* initial write max */
591 #define L2ARC_HEADROOM          2               /* num of writes */
592 #define L2ARC_FEED_SECS         1               /* caching interval secs */
593 #define L2ARC_FEED_MIN_MS       200             /* min caching interval ms */

595 #define l2arc_writes_sent       ARCSTAT(arcstat_l2_writes_sent)
596 #define l2arc_writes_done       ARCSTAT(arcstat_l2_writes_done)

598 /* L2ARC Performance Tunables */
598 /*
599  * L2ARC Performance Tunables
600  */
599 uint64_t l2arc_write_max = L2ARC_WRITE_SIZE;    /* default max write size */
600 uint64_t l2arc_write_boost = L2ARC_WRITE_SIZE;  /* extra write during warmup */
601 uint64_t l2arc_headroom = L2ARC_HEADROOM;       /* number of dev writes */
602 uint64_t l2arc_feed_secs = L2ARC_FEED_SECS;     /* interval seconds */
603 uint64_t l2arc_feed_min_ms = L2ARC_FEED_MIN_MS; /* min interval milliseconds */
604 boolean_t l2arc_noprefetch = B_TRUE;            /* don't cache prefetch bufs */
605 boolean_t l2arc_feed_again = B_TRUE;            /* turbo warmup */
606 boolean_t l2arc_norw = B_TRUE;                  /* no reads during writes */

608 /*
609  * L2ARC Internals
610  */
611 typedef struct l2arc_dev {
612         vdev_t                  *l2ad_vdev;     /* vdev */
613         spa_t                   *l2ad_spa;      /* spa */
614         uint64_t                l2ad_hand;      /* next write location */
615         uint64_t                l2ad_write;     /* desired write size, bytes */
616         uint64_t                l2ad_boost;     /* warmup write boost, bytes */
617         uint64_t                l2ad_start;     /* first addr on device */
618         uint64_t                l2ad_end;       /* last addr on device */
619         uint64_t                l2ad_evict;     /* last addr eviction reached */
620         boolean_t               l2ad_first;     /* first sweep through */
621         boolean_t               l2ad_writing;   /* currently writing */
622         list_t                  *l2ad_buflist;  /* buffer list */
623         list_node_t             l2ad_node;      /* device list node */
624 } l2arc_dev_t;
______unchanged_portion_omitted_

3517 int
3518 arc_tempreserve_space(uint64_t reserve, uint64_t txg)
3519 {
3520         int error;
3521         uint64_t anon_size;

3523 #ifdef ZFS_DEBUG
3524         /*
3525          * Once in a while, fail for no reason.  Everything should cope.
3526          */
3527         if (spa_get_random(10000) == 0) {
3528                 dprintf("forcing random failure\n");
3529                 return (SET_ERROR(ERESTART));
3530         }
3531 #endif
3532         if (reserve > arc_c/4 && !arc_no_grow)
3533                 arc_c = MIN(arc_c_max, reserve * 4);
3534         if (reserve > arc_c)
3535                 return (SET_ERROR(ENOMEM));
```

```
3537         /*
3538          * Don't count loaned bufs as in flight dirty data to prevent long
3539          * network delays from blocking transactions that are ready to be
3540          * assigned to a txg.
3541          */
3542         anon_size = MAX((int64_t)(arc_anon->arcs_size - arc_loaned_bytes), 0);

3544         /*
3545          * Writes will, almost always, require additional memory allocations
3546          * in order to compress/encrypt/etc the data.  We therefore need to
3548          * in order to compress/encrypt/etc the data.  We therefor need to
3547          * make sure that there is sufficient available memory for this.
3548          */
3549         if (error = arc_memory_throttle(reserve, anon_size, txg))
3550                 return (error);

3552         /*
3553          * Throttle writes when the amount of dirty data in the cache
3554          * gets too large.  We try to keep the cache less than half full
3555          * of dirty blocks so that our sync times don't grow too large.
3556          * Note: if two requests come in concurrently, we might let them
3557          * both succeed, when one of them should fail.  Not a huge deal.
3558          */

3560         if (reserve + arc_tempreserve + anon_size > arc_c / 2 &&
3561             anon_size > arc_c / 4) {
3562                 dprintf("failing, arc_tempreserve=%lluK anon_meta=%lluK "
3563                     "anon_data=%lluK tempreserve=%lluK arc_c=%lluK\n",
3564                     arc_tempreserve>>10,
3565                     arc_anon->arcs_lsize[ARC_BUFC_METADATA]>>10,
3566                     arc_anon->arcs_lsize[ARC_BUFC_DATA]>>10,
3567                     reserve>>10, arc_c>>10);
3568                 return (SET_ERROR(ERESTART));
3569         }
3570         atomic_add_64(&arc_tempreserve, reserve);
3571         return (0);
3572 }
```

_____unchanged_portion_omitted_

```
**********************************************************
    5987 Thu May 16 17:36:16 2013
new/usr/src/uts/common/fs/zfs/bptree.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#include <sys/arc.h>
#include <sys/bptree.h>
#include <sys/dmu.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_tx.h>
#include <sys/dmu_traverse.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_pool.h>
#include <sys/dnode.h>
#include <sys/refcount.h>
#include <sys/spa.h>

/*
 * A bptree is a queue of root block pointers from destroyed datasets. When a
 * dataset is destroyed its root block pointer is put on the end of the pool's
 * bptree queue so the dataset's blocks can be freed asynchronously by
 * dsl_scan_sync. This allows the delete operation to finish without traversing
 * all the dataset's blocks.
 *
 * Note that while bt_begin and bt_end are only ever incremented in this code,
 * Note that while bt_begin and bt_end are only ever incremented in this code
 * they are effectively reset to 0 every time the entire bptree is freed because
 * the bptree's object is destroyed and re-created.
 */

struct bptree_args {
        bptree_phys_t *ba_phys; /* data in bonus buffer, dirtied if freeing */
        boolean_t ba_free;      /* true if freeing during traversal */

        bptree_itor_t *ba_func; /* function to call for each blockpointer */
```



```
        void *ba_arg;           /* caller supplied argument to ba_func */
        dmu_tx_t *ba_tx;        /* caller supplied tx, NULL if not freeing */
} bptree_args_t;
```

*unchanged_portion_omitted_*

```
**********************************************************
   56489 Thu May 16 17:36:16 2013
new/usr/src/uts/common/fs/zfs/dnode.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1805 /*
1806  * Scans a block at the indicated "level" looking for a hole or data,
1807  * depending on 'flags'.
1808  *
1809  * If level > 0, then we are scanning an indirect block looking at its
1810  * pointers.  If level == 0, then we are looking at a block of dnodes.
1811  *
1812  * If we don't find what we are looking for in the block, we return ESRCH.
1813  * Otherwise, return with *offset pointing to the beginning (if searching
1814  * forwards) or end (if searching backwards) of the range covered by the
1815  * block pointer we matched on (or dnode).
1806  * This function scans a block at the indicated "level" looking for
1807  * a hole or data (depending on 'flags').  If level > 0, then we are
1808  * scanning an indirect block looking at its pointers.  If level == 0,
1809  * then we are looking at a block of dnodes.  If we don't find what we
1810  * are looking for in the block, we return ESRCH.  Otherwise, return
1811  * with *offset pointing to the beginning (if searching forwards) or
1812  * end (if searching backwards) of the range covered by the block
1813  * pointer we matched on (or dnode).
1816  *
1817  * The basic search algorithm used below by dnode_next_offset() is to
1818  * use this function to search up the block tree (widen the search) until
1819  * we find something (i.e., we don't return ESRCH) and then search back
1820  * down the tree (narrow the search) until we reach our original search
1821  * level.
1822  */
1823 static int
1824 dnode_next_offset_level(dnode_t *dn, int flags, uint64_t *offset,
1825         int lvl, uint64_t blkfill, uint64_t txg)
1826 {
1827         dmu_buf_impl_t *db = NULL;
1828         void *data = NULL;
1829         uint64_t epbs = dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;
1830         uint64_t epb = 1ULL << epbs;
1831         uint64_t minfill, maxfill;
1832         boolean_t hole;
1833         int i, inc, error, span;

1835         dprintf("probing object %llu offset %llx level %d of %u\n",
1836             dn->dn_object, *offset, lvl, dn->dn_phys->dn_nlevels);

1838         hole = ((flags & DNODE_FIND_HOLE) != 0);
1839         inc = (flags & DNODE_FIND_BACKWARDS) ? -1 : 1;
1840         ASSERT(txg == 0 || !hole);

1842         if (lvl == dn->dn_phys->dn_nlevels) {
1843                 error = 0;
1844                 epb = dn->dn_phys->dn_nblkptr;
1845                 data = dn->dn_phys->dn_blkptr;
1846         } else {
1847                 uint64_t blkid = dbuf_whichblock(dn, *offset) >> (epbs * lvl);
1848                 error = dbuf_hold_impl(dn, lvl, blkid, TRUE, FTAG, &db);
1849                 if (error) {
1850                         if (error != ENOENT)
```



```
1851                                 return (error);
1852                         if (hole)
1853                                 return (0);
1854                         /*
1855                          * This can only happen when we are searching up
1856                          * the block tree for data.  We don't really need to
1857                          * adjust the offset, as we will just end up looking
1858                          * at the pointer to this block in its parent, and its
1859                          * going to be unallocated, so we will skip over it.
1860                          */
1861                         return (SET_ERROR(ESRCH));
1862                 }
1863                 error = dbuf_read(db, NULL, DB_RF_CANFAIL | DB_RF_HAVESTRUCT);
1864                 if (error) {
1865                         dbuf_rele(db, FTAG);
1866                         return (error);
1867                 }
1868                 data = db->db.db_data;
1869         }

1871         if (db && txg &&
1872             (db->db_blkptr == NULL || db->db_blkptr->blk_birth <= txg)) {
1873                 /*
1874                  * This can only happen when we are searching up the tree
1875                  * and these conditions mean that we need to keep climbing.
1876                  */
1877                 error = SET_ERROR(ESRCH);
1878         } else if (lvl == 0) {
1879                 dnode_phys_t *dnp = data;
1880                 span = DNODE_SHIFT;
1881                 ASSERT(dn->dn_type == DMU_OT_DNODE);

1883                 for (i = (*offset >> span) & (blkfill - 1);
1884                     i >= 0 && i < blkfill; i += inc) {
1885                         if ((dnp[i].dn_type == DMU_OT_NONE) == hole)
1886                                 break;
1887                         *offset += (1ULL << span) * inc;
1888                 }
1889                 if (i < 0 || i == blkfill)
1890                         error = SET_ERROR(ESRCH);
1891         } else {
1892                 blkptr_t *bp = data;
1893                 uint64_t start = *offset;
1894                 span = (lvl - 1) * epbs + dn->dn_datablkshift;
1895                 minfill = 0;
1896                 maxfill = blkfill << ((lvl - 1) * epbs);

1898                 if (hole)
1899                         maxfill--;
1900                 else
1901                         minfill++;

1903                 *offset = *offset >> span;
1904                 for (i = BF64_GET(*offset, 0, epbs);
1905                     i >= 0 && i < epb; i += inc) {
1906                         if (bp[i].blk_fill >= minfill &&
1907                             bp[i].blk_fill <= maxfill &&
1908                             (hole || bp[i].blk_birth > txg))
1909                                 break;
1910                         if (inc > 0 || *offset > 0)
1911                                 *offset += inc;
1912                 }
1913                 *offset = *offset << span;
1914                 if (inc < 0) {
1915                         /* traversing backwards; position offset at the end */
1916                         ASSERT3U(*offset, <=, start);
```

```
1917                        *offset = MIN(*offset + (1ULL << span) - 1, start);
1918                } else if (*offset < start) {
1919                        *offset = start;
1920                }
1921                if (i < 0 || i >= epb)
1922                        error = SET_ERROR(ESRCH);
1923        }

1925        if (db)
1926                dbuf_rele(db, FTAG);

1928        return (error);
1929 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   19318 Thu May 16 17:36:17 2013
new/usr/src/uts/common/fs/zfs/dnode_sync.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 304 /*
 305  * Traverse the indicated range of the provided file
 305  * free_range: Traverse the indicated range of the provided file
 306  * and "free" all the blocks contained there.
 307  */
 308 static void
 309 dnode_sync_free_range(dnode_t *dn, uint64_t blkid, uint64_t nblks, dmu_tx_t *tx)
 310 {
 311         blkptr_t *bp = dn->dn_phys->dn_blkptr;
 312         dmu_buf_impl_t *db;
 313         int trunc, start, end, shift, i, err;
 314         int dnlevel = dn->dn_phys->dn_nlevels;

 316         if (blkid > dn->dn_phys->dn_maxblkid)
 317                 return;

 319         ASSERT(dn->dn_phys->dn_maxblkid < UINT64_MAX);
 320         trunc = blkid + nblks > dn->dn_phys->dn_maxblkid;
 321         if (trunc)
 322                 nblks = dn->dn_phys->dn_maxblkid - blkid + 1;

 324         /* There are no indirect blocks in the object */
 325         if (dnlevel == 1) {
 326                 if (blkid >= dn->dn_phys->dn_nblkptr) {
 327                         /* this range was never made persistent */
 328                         return;
 329                 }
 330                 ASSERT3U(blkid + nblks, <=, dn->dn_phys->dn_nblkptr);
 331                 (void) free_blocks(dn, bp + blkid, nblks, tx);
 332                 if (trunc) {
 333                         uint64_t off = (dn->dn_phys->dn_maxblkid + 1) *
 334                             (dn->dn_phys->dn_datablkszsec << SPA_MINBLOCKSHIFT);
 335                         dn->dn_phys->dn_maxblkid = (blkid ? blkid - 1 : 0);
 336                         ASSERT(off < dn->dn_phys->dn_maxblkid ||
 337                             dn->dn_phys->dn_maxblkid == 0 ||
 338                             dnode_next_offset(dn, 0, &off, 1, 1, 0) != 0);
 339                 }
 340                 return;
 341         }

 343         shift = (dnlevel - 1) * (dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT);
 344         start = blkid >> shift;
 345         ASSERT(start < dn->dn_phys->dn_nblkptr);
 346         end = (blkid + nblks - 1) >> shift;
 347         bp += start;
 348         for (i = start; i <= end; i++, bp++) {
 349                 if (BP_IS_HOLE(bp))
 350                         continue;
 351                 rw_enter(&dn->dn_struct_rwlock, RW_READER);
 352                 err = dbuf_hold_impl(dn, dnlevel-1, i, TRUE, FTAG, &db);
 353                 ASSERT0(err);
 354                 rw_exit(&dn->dn_struct_rwlock);

 356                 if (free_children(db, blkid, nblks, trunc, tx) == ALL) {
```



```
 357                         ASSERT3P(db->db_blkptr, ==, bp);
 358                         (void) free_blocks(dn, bp, 1, tx);
 359                 }
 360                 dbuf_rele(db, FTAG);
 361         }
 362         if (trunc) {
 363                 uint64_t off = (dn->dn_phys->dn_maxblkid + 1) *
 364                     (dn->dn_phys->dn_datablkszsec << SPA_MINBLOCKSHIFT);
 365                 dn->dn_phys->dn_maxblkid = (blkid ? blkid - 1 : 0);
 366                 ASSERT(off < dn->dn_phys->dn_maxblkid ||
 367                     dn->dn_phys->dn_maxblkid == 0 ||
 368                     dnode_next_offset(dn, 0, &off, 1, 1, 0) != 0);
 369         }
 370 }

 372 /*
 373  * Try to kick all the dnode's dbufs out of the cache...
 373  * Try to kick all the dnodes dbufs out of the cache...
 374  */
 375 void
 376 dnode_evict_dbufs(dnode_t *dn)
 377 {
 378         int progress;
 379         int pass = 0;

 381         do {
 382                 dmu_buf_impl_t *db, marker;
 383                 int evicting = FALSE;

 385                 progress = FALSE;
 386                 mutex_enter(&dn->dn_dbufs_mtx);
 387                 list_insert_tail(&dn->dn_dbufs, &marker);
 388                 db = list_head(&dn->dn_dbufs);
 389                 for (; db != &marker; db = list_head(&dn->dn_dbufs)) {
 390                         list_remove(&dn->dn_dbufs, db);
 391                         list_insert_tail(&dn->dn_dbufs, db);
 392 #ifdef  DEBUG
 393                         DB_DNODE_ENTER(db);
 394                         ASSERT3P(DB_DNODE(db), ==, dn);
 395                         DB_DNODE_EXIT(db);
 396 #endif  /* DEBUG */

 398                         mutex_enter(&db->db_mtx);
 399                         if (db->db_state == DB_EVICTING) {
 400                                 progress = TRUE;
 401                                 evicting = TRUE;
 402                                 mutex_exit(&db->db_mtx);
 403                         } else if (refcount_is_zero(&db->db_holds)) {
 404                                 progress = TRUE;
 405                                 dbuf_clear(db); /* exits db_mtx for us */
 406                         } else {
 407                                 mutex_exit(&db->db_mtx);
 408                         }

 410                 }
 411                 list_remove(&dn->dn_dbufs, &marker);
 412                 /*
 413                  * NB: we need to drop dn_dbufs_mtx between passes so
 414                  * that any DB_EVICTING dbufs can make progress.
 415                  * Ideally, we would have some cv we could wait on, but
 416                  * since we don't, just wait a bit to give the other
 417                  * thread a chance to run.
 418                  */
 419                 mutex_exit(&dn->dn_dbufs_mtx);
 420                 if (evicting)
 421                         delay(1);
```

```
                pass++;
                ASSERT(pass < 100); /* sanity check */
        } while (progress);

        rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
        if (dn->dn_bonus && refcount_is_zero(&dn->dn_bonus->db_holds)) {
                mutex_enter(&dn->dn_bonus->db_mtx);
                dbuf_evict(dn->dn_bonus);
                dn->dn_bonus = NULL;
        }
        rw_exit(&dn->dn_struct_rwlock);
}
```

unchanged_portion_omitted_

```
**********************************************************
   29200 Thu May 16 17:36:17 2013
new/usr/src/uts/common/fs/zfs/dsl_prop.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 380 /*
 381  * Unregister this callback.  Return 0 on success, ENOENT if ddname is
 382  * invalid, or ENOMSG if no matching callback registered.
 382  * invalid, ENOMSG if no matching callback registered.
 383  */
 384 int
 385 dsl_prop_unregister(dsl_dataset_t *ds, const char *propname,
 386     dsl_prop_changed_cb_t *callback, void *cbarg)
 387 {
 388         dsl_dir_t *dd = ds->ds_dir;
 389         dsl_prop_cb_record_t *cbr;

 391         mutex_enter(&dd->dd_lock);
 392         for (cbr = list_head(&dd->dd_prop_cbs);
 393             cbr; cbr = list_next(&dd->dd_prop_cbs, cbr)) {
 394                 if (cbr->cbr_ds == ds &&
 395                     cbr->cbr_func == callback &&
 396                     cbr->cbr_arg == cbarg &&
 397                     strcmp(cbr->cbr_propname, propname) == 0)
 398                         break;
 399         }

 401         if (cbr == NULL) {
 402                 mutex_exit(&dd->dd_lock);
 403                 return (SET_ERROR(ENOMSG));
 404         }

 406         list_remove(&dd->dd_prop_cbs, cbr);
 407         mutex_exit(&dd->dd_lock);
 408         kmem_free((void*)cbr->cbr_propname, strlen(cbr->cbr_propname)+1);
 409         kmem_free(cbr, sizeof (dsl_prop_cb_record_t));

 411         return (0);
 412 }
______unchanged_portion_omitted_
```

```
**********************************************************
   51577 Thu May 16 17:36:17 2013
new/usr/src/uts/common/fs/zfs/sa.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 * Portions Copyright 2011 iXsystems, Inc
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/dmu.h>
#include <sys/dmu_impl.h>
#include <sys/dmu_objset.h>
#include <sys/dbuf.h>
#include <sys/dnode.h>
#include <sys/zap.h>
#include <sys/sa.h>
#include <sys/sunddi.h>
#include <sys/sa_impl.h>
#include <sys/dnode.h>
#include <sys/errno.h>
#include <sys/zfs_context.h>

/*
 * ZFS System attributes:
 *
 * A generic mechanism to allow for arbitrary attributes
 * to be stored in a dnode.  The data will be stored in the bonus buffer of
 * the dnode and if necessary a special "spill" block will be used to handle
 * overflow situations.  The spill block will be sized to fit the data
 * from 512 - 128K.  When a spill block is used the BP (blkptr_t) for the
 * spill block is stored at the end of the current bonus buffer.  Any
 * attributes that would be in the way of the blkptr_t will be relocated
 * into the spill block.
```



```
 *
 * Attribute registration:
 *
 * Stored persistently on a per dataset basis
 * a mapping between attribute "string" names and their actual attribute
 * numeric values, length, and byteswap function.  The names are only used
 * during registration.  All  attributes are known by their unique attribute
 * id value.  If an attribute can have a variable size then the value
 * 0 will be used to indicate this.
 *
 * Attribute Layout:
 *
 * Attribute layouts are a way to compactly store multiple attributes, but
 * without taking the overhead associated with managing each attribute
 * individually.  Since you will typically have the same set of attributes
 * stored in the same order a single table will be used to represent that
 * layout.  The ZPL for example will usually have only about 10 different
 * layouts (regular files, device files, symlinks,
 * regular files + scanstamp, files/dir with extended attributes, and then
 * you have the possibility of all of those minus ACL, because it would
 * be kicked out into the spill block)
 *
 * Layouts are simply an array of the attributes and their
 * ordering i.e. [0, 1, 4, 5, 2]
 *
 * Each distinct layout is given a unique layout number and that is whats
 * stored in the header at the beginning of the SA data buffer.
 *
 * A layout only covers a single dbuf (bonus or spill).  If a set of
 * attributes is split up between the bonus buffer and a spill buffer then
 * two different layouts will be used.  This allows us to byteswap the
 * spill without looking at the bonus buffer and keeps the on disk format of
 * the bonus and spill buffer the same.
 *
 * Adding a single attribute will cause the entire set of attributes to
 * be rewritten and could result in a new layout number being constructed
 * as part of the rewrite if no such layout exists for the new set of
 * attribues.  The new attribute will be appended to the end of the already
 * existing attributes.
 *
 * Both the attribute registration and attribute layout information are
 * stored in normal ZAP attributes.  Their should be a small number of
 * known layouts and the set of attributes is assumed to typically be quite
 * small.
 *
 * The registered attributes and layout "table" information is maintained
 * in core and a special "sa_os_t" is attached to the objset_t.
 *
 * A special interface is provided to allow for quickly applying
 * a large set of attributes at once.  sa_replace_all_by_template() is
 * used to set an array of attributes.  This is used by the ZPL when
 * creating a brand new file.  The template that is passed into the function
 * specifies the attribute, size for variable length attributes, location of
 * data and special "data locator" function if the data isn't in a contiguous
 * location.
 *
 * Byteswap implications:
 *
#endif /* ! codereview */
 * Since the SA attributes are not entirely self describing we can't do
 * the normal byteswap processing.  The special ZAP layout attribute and
 * attribute registration attributes define the byteswap function and the
 * size of the attributes, unless it is variable sized.
 * The normal ZFS byteswapping infrastructure assumes you don't need
 * to read any objects in order to do the necessary byteswapping.  Whereas
 * SA attributes can only be properly byteswapped if the dataset is opened
```

```
123  * and the layout/attribute ZAP attributes are available.  Because of this
124  * the SA attributes will be byteswapped when they are first accessed by
125  * the SA code that will read the SA data.
126  */
127
128 typedef void (sa_iterfunc_t)(void *hdr, void *addr, sa_attr_type_t,
129     uint16_t length, int length_idx, boolean_t, void *userp);
130
131 static int sa_build_index(sa_handle_t *hdl, sa_buf_type_t buftype);
132 static void sa_idx_tab_hold(objset_t *os, sa_idx_tab_t *idx_tab);
133 static void *sa_find_idx_tab(objset_t *os, dmu_object_type_t bonustype,
134     void *data);
135 static void sa_idx_tab_rele(objset_t *os, void *arg);
136 static void sa_copy_data(sa_data_locator_t *func, void *start, void *target,
137     int buflen);
138 static int sa_modify_attrs(sa_handle_t *hdl, sa_attr_type_t newattr,
139     sa_data_op_t action, sa_data_locator_t *locator, void *datastart,
140     uint16_t buflen, dmu_tx_t *tx);
141
142 arc_byteswap_func_t *sa_bswap_table[] = {
143         byteswap_uint64_array,
144         byteswap_uint32_array,
145         byteswap_uint16_array,
146         byteswap_uint8_array,
147         zfs_acl_byteswap,
148 };
149
150 #define SA_COPY_DATA(f, s, t, l) \
151         { \
152                 if (f == NULL) { \
153                         if (l == 8) { \
154                                 *(uint64_t *)t = *(uint64_t *)s; \
155                         } else if (l == 16) { \
156                                 *(uint64_t *)t = *(uint64_t *)s; \
157                                 *(uint64_t *)((uintptr_t)t + 8) = \
158                                     *(uint64_t *)((uintptr_t)s + 8); \
159                         } else { \
160                                 bcopy(s, t, l); \
161                         } \
162                 } else \
163                         sa_copy_data(f, s, t, l); \
164         }
165
166 /*
167  * This table is fixed and cannot be changed.  Its purpose is to
168  * allow the SA code to work with both old/new ZPL file systems.
169  * It contains the list of legacy attributes.  These attributes aren't
170  * stored in the "attribute" registry zap objects, since older ZPL file systems
171  * won't have the registry.  Only objsets of type ZFS_TYPE_FILESYSTEM will
172  * use this static table.
173  */
174 sa_attr_reg_t sa_legacy_attrs[] = {
175         {"ZPL_ATIME", sizeof (uint64_t) * 2, SA_UINT64_ARRAY, 0},
176         {"ZPL_MTIME", sizeof (uint64_t) * 2, SA_UINT64_ARRAY, 1},
177         {"ZPL_CTIME", sizeof (uint64_t) * 2, SA_UINT64_ARRAY, 2},
178         {"ZPL_CRTIME", sizeof (uint64_t) * 2, SA_UINT64_ARRAY, 3},
179         {"ZPL_GEN", sizeof (uint64_t), SA_UINT64_ARRAY, 4},
180         {"ZPL_MODE", sizeof (uint64_t), SA_UINT64_ARRAY, 5},
181         {"ZPL_SIZE", sizeof (uint64_t), SA_UINT64_ARRAY, 6},
182         {"ZPL_PARENT", sizeof (uint64_t), SA_UINT64_ARRAY, 7},
183         {"ZPL_LINKS", sizeof (uint64_t), SA_UINT64_ARRAY, 8},
184         {"ZPL_XATTR", sizeof (uint64_t), SA_UINT64_ARRAY, 9},
185         {"ZPL_RDEV", sizeof (uint64_t), SA_UINT64_ARRAY, 10},
186         {"ZPL_FLAGS", sizeof (uint64_t), SA_UINT64_ARRAY, 11},
187         {"ZPL_UID", sizeof (uint64_t), SA_UINT64_ARRAY, 12},
188         {"ZPL_GID", sizeof (uint64_t), SA_UINT64_ARRAY, 13},
```



```
189         {"ZPL_PAD", sizeof (uint64_t) * 4, SA_UINT64_ARRAY, 14},
190         {"ZPL_ZNODE_ACL", 88, SA_UINT8_ARRAY, 15},
191 };
192
193 /*
194  * ZPL legacy layout
195  * This is only used for objects of type DMU_OT_ZNODE
196  */
197 sa_attr_type_t sa_legacy_zpl_layout[] = {
198     0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
199 };
200
201 /*
202  * Special dummy layout used for buffers with no attributes.
203  */
204 sa_attr_type_t sa_dummy_zpl_layout[] = { 0 };
205
206 static int sa_legacy_attr_count = 16;
207 static kmem_cache_t *sa_cache = NULL;
208
209 /*ARGSUSED*/
210 static int
211 sa_cache_constructor(void *buf, void *unused, int kmflag)
212 {
213         sa_handle_t *hdl = buf;
214
215         hdl->sa_bonus_tab = NULL;
216         hdl->sa_spill_tab = NULL;
217         hdl->sa_os = NULL;
218         hdl->sa_userp = NULL;
219         hdl->sa_bonus = NULL;
220         hdl->sa_spill = NULL;
221         mutex_init(&hdl->sa_lock, NULL, MUTEX_DEFAULT, NULL);
222         return (0);
223 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  174704 Thu May 16 17:36:18 2013
new/usr/src/uts/common/fs/zfs/spa.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

4517 /*
4518  * Detach a device from a mirror or replacing vdev.
4519  *
4520 #endif /* ! codereview */
4521  * If 'replace_done' is specified, only detach if the parent
4522  * is a replacing vdev.
4523  */
4524 int
4525 spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid, int replace_done)
4526 {
4527         uint64_t txg;
4528         int error;
4529         vdev_t *rvd = spa->spa_root_vdev;
4530         vdev_t *vd, *pvd, *cvd, *tvd;
4531         boolean_t unspare = B_FALSE;
4532         uint64_t unspare_guid = 0;
4533         char *vdpath;

4535         ASSERT(spa_writeable(spa));

4537         txg = spa_vdev_enter(spa);

4539         vd = spa_lookup_by_guid(spa, guid, B_FALSE);

4541         if (vd == NULL)
4542                 return (spa_vdev_exit(spa, NULL, txg, ENODEV));

4544         if (!vd->vdev_ops->vdev_op_leaf)
4545                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4547         pvd = vd->vdev_parent;

4549         /*
4550          * If the parent/child relationship is not as expected, don't do it.
4551          * Consider M(A,R(B,C)) -- that is, a mirror of A with a replacing
4552          * vdev that's replacing B with C.  The user's intent in replacing
4553          * is to go from M(A,B) to M(A,C).  If the user decides to cancel
4554          * the replace by detaching C, the expected behavior is to end up
4555          * M(A,B).  But suppose that right after deciding to detach C,
4556          * the replacement of B completes.  We would have M(A,C), and then
4557          * ask to detach C, which would leave us with just A -- not what
4558          * the user wanted.  To prevent this, we make sure that the
4559          * parent/child relationship hasn't changed -- in this example,
4560          * that C's parent is still the replacing vdev R.
4561          */
4562         if (pvd->vdev_guid != pguid && pguid != 0)
4563                 return (spa_vdev_exit(spa, NULL, txg, EBUSY));

4565         /*
4566          * Only 'replacing' or 'spare' vdevs can be replaced.
4567          */
4568         if (replace_done && pvd->vdev_ops != &vdev_replacing_ops &&
4569             pvd->vdev_ops != &vdev_spare_ops)
4570                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));
```



```
4572         ASSERT(pvd->vdev_ops != &vdev_spare_ops ||
4573             spa_version(spa) >= SPA_VERSION_SPARES);

4575         /*
4576          * Only mirror, replacing, and spare vdevs support detach.
4577          */
4578         if (pvd->vdev_ops != &vdev_replacing_ops &&
4579             pvd->vdev_ops != &vdev_mirror_ops &&
4580             pvd->vdev_ops != &vdev_spare_ops)
4581                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4583         /*
4584          * If this device has the only valid copy of some data,
4585          * we cannot safely detach it.
4586          */
4587         if (vdev_dtl_required(vd))
4588                 return (spa_vdev_exit(spa, NULL, txg, EBUSY));

4590         ASSERT(pvd->vdev_children >= 2);

4592         /*
4593          * If we are detaching the second disk from a replacing vdev, then
4594          * check to see if we changed the original vdev's path to have "/old"
4595          * at the end in spa_vdev_attach().  If so, undo that change now.
4596          */
4597         if (pvd->vdev_ops == &vdev_replacing_ops && vd->vdev_id > 0 &&
4598             vd->vdev_path != NULL) {
4599                 size_t len = strlen(vd->vdev_path);

4601                 for (int c = 0; c < pvd->vdev_children; c++) {
4602                         cvd = pvd->vdev_child[c];

4604                         if (cvd == vd || cvd->vdev_path == NULL)
4605                                 continue;

4607                         if (strncmp(cvd->vdev_path, vd->vdev_path, len) == 0 &&
4608                             strcmp(cvd->vdev_path + len, "/old") == 0) {
4609                                 spa_strfree(cvd->vdev_path);
4610                                 cvd->vdev_path = spa_strdup(vd->vdev_path);
4611                                 break;
4612                         }
4613                 }
4614         }

4616         /*
4617          * If we are detaching the original disk from a spare, then it implies
4618          * that the spare should become a real disk, and be removed from the
4619          * active spare list for the pool.
4620          */
4621         if (pvd->vdev_ops == &vdev_spare_ops &&
4622             vd->vdev_id == 0 &&
4623             pvd->vdev_child[pvd->vdev_children - 1]->vdev_isspare)
4624                 unspare = B_TRUE;

4626         /*
4627          * Erase the disk labels so the disk can be used for other things.
4628          * This must be done after all other error cases are handled,
4629          * but before we disembowel vd (so we can still do I/O to it).
4630          * But if we can't do it, don't treat the error as fatal --
4631          * it may be that the unwritability of the disk is the reason
4632          * it's being detached!
4633          */
4634         error = vdev_label_init(vd, 0, VDEV_LABEL_REMOVE);

4636         /*
```

```
4637          * Remove vd from its parent and compact the parent's children.
4638          */
4639         vdev_remove_child(pvd, vd);
4640         vdev_compact_children(pvd);

4642         /*
4643          * Remember one of the remaining children so we can get tvd below.
4644          */
4645         cvd = pvd->vdev_child[pvd->vdev_children - 1];

4647         /*
4648          * If we need to remove the remaining child from the list of hot spares,
4649          * do it now, marking the vdev as no longer a spare in the process.
4650          * We must do this before vdev_remove_parent(), because that can
4651          * change the GUID if it creates a new toplevel GUID.  For a similar
4652          * reason, we must remove the spare now, in the same txg as the detach;
4653          * otherwise someone could attach a new sibling, change the GUID, and
4654          * the subsequent attempt to spa_vdev_remove(unspare_guid) would fail.
4655          */
4656         if (unspare) {
4657                 ASSERT(cvd->vdev_isspare);
4658                 spa_spare_remove(cvd);
4659                 unspare_guid = cvd->vdev_guid;
4660                 (void) spa_vdev_remove(spa, unspare_guid, B_TRUE);
4661                 cvd->vdev_unspare = B_TRUE;
4662         }

4664         /*
4665          * If the parent mirror/replacing vdev only has one child,
4666          * the parent is no longer needed.  Remove it from the tree.
4667          */
4668         if (pvd->vdev_children == 1) {
4669                 if (pvd->vdev_ops == &vdev_spare_ops)
4670                         cvd->vdev_unspare = B_FALSE;
4671                 vdev_remove_parent(cvd);
4672                 cvd->vdev_resilvering = B_FALSE;
4673         }


4676         /*
4677          * We don't set tvd until now because the parent we just removed
4678          * may have been the previous top-level vdev.
4679          */
4680         tvd = cvd->vdev_top;
4681         ASSERT(tvd->vdev_parent == rvd);

4683         /*
4684          * Reevaluate the parent vdev state.
4685          */
4686         vdev_propagate_state(cvd);

4688         /*
4689          * If the 'autoexpand' property is set on the pool then automatically
4690          * try to expand the size of the pool. For example if the device we
4691          * just detached was smaller than the others, it may be possible to
4692          * add metaslabs (i.e. grow the pool). We need to reopen the vdev
4693          * first so that we can obtain the updated sizes of the leaf vdevs.
4694          */
4695         if (spa->spa_autoexpand) {
4696                 vdev_reopen(tvd);
4697                 vdev_expand(tvd, txg);
4698         }

4700         vdev_config_dirty(tvd);

4702         /*
```



```
4703          * Mark vd's DTL as dirty in this txg.  vdev_dtl_sync() will see that
4704          * vd->vdev_detached is set and free vd's DTL object in syncing context.
4705          * But first make sure we're not on any *other* txg's DTL list, to
4706          * prevent vd from being accessed after it's freed.
4707          */
4708         vdpath = spa_strdup(vd->vdev_path);
4709         for (int t = 0; t < TXG_SIZE; t++)
4710                 (void) txg_list_remove_this(&tvd->vdev_dtl_list, vd, t);
4711         vd->vdev_detached = B_TRUE;
4712         vdev_dirty(tvd, VDD_DTL, vd, txg);

4714         spa_event_notify(spa, vd, ESC_ZFS_VDEV_REMOVE);

4716         /* hang on to the spa before we release the lock */
4717         spa_open_ref(spa, FTAG);

4719         error = spa_vdev_exit(spa, vd, txg, 0);

4721         spa_history_log_internal(spa, "detach", NULL,
4722             "vdev=%s", vdpath);
4723         spa_strfree(vdpath);

4725         /*
4726          * If this was the removal of the original device in a hot spare vdev,
4727          * then we want to go through and remove the device from the hot spare
4728          * list of every other pool.
4729          */
4730         if (unspare) {
4731                 spa_t *altspa = NULL;

4733                 mutex_enter(&spa_namespace_lock);
4734                 while ((altspa = spa_next(altspa)) != NULL) {
4735                         if (altspa->spa_state != POOL_STATE_ACTIVE ||
4736                             altspa == spa)
4737                                 continue;

4739                         spa_open_ref(altspa, FTAG);
4740                         mutex_exit(&spa_namespace_lock);
4741                         (void) spa_vdev_remove(altspa, unspare_guid, B_TRUE);
4742                         mutex_enter(&spa_namespace_lock);
4743                         spa_close(altspa, FTAG);
4744                 }
4745                 mutex_exit(&spa_namespace_lock);

4747                 /* search the rest of the vdevs for spares to remove */
4748                 spa_vdev_resilver_done(spa);
4749         }

4751         /* all done with the spa; OK to release */
4752         mutex_enter(&spa_namespace_lock);
4753         spa_close(spa, FTAG);
4754         mutex_exit(&spa_namespace_lock);

4756         return (error);
4757 }

4759 /*
4760  * Split a set of devices from their mirrors, and create a new pool from them.
4761  */
4762 int
4763 spa_vdev_split_mirror(spa_t *spa, char *newname, nvlist_t *config,
4764     nvlist_t *props, boolean_t exp)
4765 {
4766         int error = 0;
4767         uint64_t txg, *glist;
4768         spa_t *newspa;
```

```
4769         uint_t c, children, lastlog;
4770         nvlist_t **child, *nvl, *tmp;
4771         dmu_tx_t *tx;
4772         char *altroot = NULL;
4773         vdev_t *rvd, **vml = NULL;                      /* vdev modify list */
4774         boolean_t activate_slog;

4776         ASSERT(spa_writeable(spa));

4778         txg = spa_vdev_enter(spa);

4780         /* clear the log and flush everything up to now */
4781         activate_slog = spa_passivate_log(spa);
4782         (void) spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
4783         error = spa_offline_log(spa);
4784         txg = spa_vdev_config_enter(spa);

4786         if (activate_slog)
4787                 spa_activate_log(spa);

4789         if (error != 0)
4790                 return (spa_vdev_exit(spa, NULL, txg, error));

4792         /* check new spa name before going any further */
4793         if (spa_lookup(newname) != NULL)
4794                 return (spa_vdev_exit(spa, NULL, txg, EEXIST));

4796         /*
4797          * scan through all the children to ensure they're all mirrors
4798          */
4799         if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvl) != 0 ||
4800             nvlist_lookup_nvlist_array(nvl, ZPOOL_CONFIG_CHILDREN, &child,
4801             &children) != 0)
4802                 return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4804         /* first, check to ensure we've got the right child count */
4805         rvd = spa->spa_root_vdev;
4806         lastlog = 0;
4807         for (c = 0; c < rvd->vdev_children; c++) {
4808                 vdev_t *vd = rvd->vdev_child[c];

4810                 /* don't count the holes & logs as children */
4811                 if (vd->vdev_islog || vd->vdev_ishole) {
4812                         if (lastlog == 0)
4813                                 lastlog = c;
4814                         continue;
4815                 }

4817                 lastlog = 0;
4818         }
4819         if (children != (lastlog != 0 ? lastlog : rvd->vdev_children))
4820                 return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4822         /* next, ensure no spare or cache devices are part of the split */
4823         if (nvlist_lookup_nvlist(nvl, ZPOOL_CONFIG_SPARES, &tmp) == 0 ||
4824             nvlist_lookup_nvlist(nvl, ZPOOL_CONFIG_L2CACHE, &tmp) == 0)
4825                 return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4827         vml = kmem_zalloc(children * sizeof (vdev_t *), KM_SLEEP);
4828         glist = kmem_zalloc(children * sizeof (uint64_t), KM_SLEEP);

4830         /* then, loop over each vdev and validate it */
4831         for (c = 0; c < children; c++) {
4832                 uint64_t is_hole = 0;

4834                 (void) nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_IS_HOLE,
```



```
4835                     &is_hole);

4837                 if (is_hole != 0) {
4838                         if (spa->spa_root_vdev->vdev_child[c]->vdev_ishole ||
4839                             spa->spa_root_vdev->vdev_child[c]->vdev_islog) {
4840                                 continue;
4841                         } else {
4842                                 error = SET_ERROR(EINVAL);
4843                                 break;
4844                         }
4845                 }

4847                 /* which disk is going to be split? */
4848                 if (nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_GUID,
4849                     &glist[c]) != 0) {
4850                         error = SET_ERROR(EINVAL);
4851                         break;
4852                 }

4854                 /* look it up in the spa */
4855                 vml[c] = spa_lookup_by_guid(spa, glist[c], B_FALSE);
4856                 if (vml[c] == NULL) {
4857                         error = SET_ERROR(ENODEV);
4858                         break;
4859                 }

4861                 /* make sure there's nothing stopping the split */
4862                 if (vml[c]->vdev_parent->vdev_ops != &vdev_mirror_ops ||
4863                     vml[c]->vdev_islog ||
4864                     vml[c]->vdev_ishole ||
4865                     vml[c]->vdev_isspare ||
4866                     vml[c]->vdev_isl2cache ||
4867                     !vdev_writeable(vml[c]) ||
4868                     vml[c]->vdev_children != 0 ||
4869                     vml[c]->vdev_state != VDEV_STATE_HEALTHY ||
4870                     c != spa->spa_root_vdev->vdev_child[c]->vdev_id) {
4871                         error = SET_ERROR(EINVAL);
4872                         break;
4873                 }

4875                 if (vdev_dtl_required(vml[c])) {
4876                         error = SET_ERROR(EBUSY);
4877                         break;
4878                 }

4880                 /* we need certain info from the top level */
4881                 VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_METASLAB_ARRAY,
4882                     vml[c]->vdev_top->vdev_ms_array) == 0);
4883                 VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_METASLAB_SHIFT,
4884                     vml[c]->vdev_top->vdev_ms_shift) == 0);
4885                 VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_ASIZE,
4886                     vml[c]->vdev_top->vdev_asize) == 0);
4887                 VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_ASHIFT,
4888                     vml[c]->vdev_top->vdev_ashift) == 0);
4889         }

4891         if (error != 0) {
4892                 kmem_free(vml, children * sizeof (vdev_t *));
4893                 kmem_free(glist, children * sizeof (uint64_t));
4894                 return (spa_vdev_exit(spa, NULL, txg, error));
4895         }

4897         /* stop writers from using the disks */
4898         for (c = 0; c < children; c++) {
4899                 if (vml[c] != NULL)
4900                         vml[c]->vdev_offline = B_TRUE;
```

```
4901         }
4902         vdev_reopen(spa->spa_root_vdev);
4903
4904         /*
4905          * Temporarily record the splitting vdevs in the spa config.  This
4906          * will disappear once the config is regenerated.
4907          */
4908         VERIFY(nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP) == 0);
4909         VERIFY(nvlist_add_uint64_array(nvl, ZPOOL_CONFIG_SPLIT_LIST,
4910             glist, children) == 0);
4911         kmem_free(glist, children * sizeof (uint64_t));
4912
4913         mutex_enter(&spa->spa_props_lock);
4914         VERIFY(nvlist_add_nvlist(spa->spa_config, ZPOOL_CONFIG_SPLIT,
4915             nvl) == 0);
4916         mutex_exit(&spa->spa_props_lock);
4917         spa->spa_config_splitting = nvl;
4918         vdev_config_dirty(spa->spa_root_vdev);
4919
4920         /* configure and create the new pool */
4921         VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_POOL_NAME, newname) == 0);
4922         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_STATE,
4923             exp ? POOL_STATE_EXPORTED : POOL_STATE_ACTIVE) == 0);
4924         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_VERSION,
4925             spa_version(spa)) == 0);
4926         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_TXG,
4927             spa->spa_config_txg) == 0);
4928         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_GUID,
4929             spa_generate_guid(NULL)) == 0);
4930         (void) nvlist_lookup_string(props,
4931             zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
4932
4933         /* add the new pool to the namespace */
4934         newspa = spa_add(newname, config, altroot);
4935         newspa->spa_config_txg = spa->spa_config_txg;
4936         spa_set_log_state(newspa, SPA_LOG_CLEAR);
4937
4938         /* release the spa config lock, retaining the namespace lock */
4939         spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
4940
4941         if (zio_injection_enabled)
4942                 zio_handle_panic_injection(spa, FTAG, 1);
4943
4944         spa_activate(newspa, spa_mode_global);
4945         spa_async_suspend(newspa);
4946
4947         /* create the new pool from the disks of the original pool */
4948         error = spa_load(newspa, SPA_LOAD_IMPORT, SPA_IMPORT_ASSEMBLE, B_TRUE);
4949         if (error)
4950                 goto out;
4951
4952         /* if that worked, generate a real config for the new pool */
4953         if (newspa->spa_root_vdev != NULL) {
4954                 VERIFY(nvlist_alloc(&newspa->spa_config_splitting,
4955                     NV_UNIQUE_NAME, KM_SLEEP) == 0);
4956                 VERIFY(nvlist_add_uint64(newspa->spa_config_splitting,
4957                     ZPOOL_CONFIG_SPLIT_GUID, spa_guid(spa)) == 0);
4958                 spa_config_set(newspa, spa_config_generate(newspa, NULL, -1ULL,
4959                     B_TRUE));
4960         }
4961
4962         /* set the props */
4963         if (props != NULL) {
4964                 spa_configfile_set(newspa, props, B_FALSE);
4965                 error = spa_prop_set(newspa, props);
4966                 if (error)
```



```
4967                         goto out;
4968         }
4969
4970         /* flush everything */
4971         txg = spa_vdev_config_enter(newspa);
4972         vdev_config_dirty(newspa->spa_root_vdev);
4973         (void) spa_vdev_config_exit(newspa, NULL, txg, 0, FTAG);
4974
4975         if (zio_injection_enabled)
4976                 zio_handle_panic_injection(spa, FTAG, 2);
4977
4978         spa_async_resume(newspa);
4979
4980         /* finally, update the original pool's config */
4981         txg = spa_vdev_config_enter(spa);
4982         tx = dmu_tx_create_dd(spa_get_dsl(spa)->dp_mos_dir);
4983         error = dmu_tx_assign(tx, TXG_WAIT);
4984         if (error != 0)
4985                 dmu_tx_abort(tx);
4986         for (c = 0; c < children; c++) {
4987                 if (vml[c] != NULL) {
4988                         vdev_split(vml[c]);
4989                         if (error == 0)
4990                                 spa_history_log_internal(spa, "detach", tx,
4991                                     "vdev=%s", vml[c]->vdev_path);
4992                         vdev_free(vml[c]);
4993                 }
4994         }
4995         vdev_config_dirty(spa->spa_root_vdev);
4996         spa->spa_config_splitting = NULL;
4997         nvlist_free(nvl);
4998         if (error == 0)
4999                 dmu_tx_commit(tx);
5000         (void) spa_vdev_exit(spa, NULL, txg, 0);
5001
5002         if (zio_injection_enabled)
5003                 zio_handle_panic_injection(spa, FTAG, 3);
5004
5005         /* split is complete; log a history record */
5006         spa_history_log_internal(newspa, "split", NULL,
5007             "from pool %s", spa_name(spa));
5008
5009         kmem_free(vml, children * sizeof (vdev_t *));
5010
5011         /* if we're not going to mount the filesystems in userland, export */
5012         if (exp)
5013                 error = spa_export_common(newname, POOL_STATE_EXPORTED, NULL,
5014                     B_FALSE, B_FALSE);
5015
5016         return (error);
5017
5018 out:
5019         spa_unload(newspa);
5020         spa_deactivate(newspa);
5021         spa_remove(newspa);
5022
5023         txg = spa_vdev_config_enter(spa);
5024
5025         /* re-online all offlined disks */
5026         for (c = 0; c < children; c++) {
5027                 if (vml[c] != NULL)
5028                         vml[c]->vdev_offline = B_FALSE;
5029         }
5030         vdev_reopen(spa->spa_root_vdev);
5031
5032         nvlist_free(spa->spa_config_splitting);
```

```
5033         spa->spa_config_splitting = NULL;
5034         (void) spa_vdev_exit(spa, NULL, txg, error);
5035
5036         kmem_free(vml, children * sizeof (vdev_t *));
5037         return (error);
5038 }
5039
5040 static nvlist_t *
5041 spa_nvlist_lookup_by_guid(nvlist_t **nvpp, int count, uint64_t target_guid)
5042 {
5043         for (int i = 0; i < count; i++) {
5044                 uint64_t guid;
5045
5046                 VERIFY(nvlist_lookup_uint64(nvpp[i], ZPOOL_CONFIG_GUID,
5047                     &guid) == 0);
5048
5049                 if (guid == target_guid)
5050                         return (nvpp[i]);
5051         }
5052
5053         return (NULL);
5054 }
5055
5056 static void
5057 spa_vdev_remove_aux(nvlist_t *config, char *name, nvlist_t **dev, int count,
5058         nvlist_t *dev_to_remove)
5059 {
5060         nvlist_t **newdev = NULL;
5061
5062         if (count > 1)
5063                 newdev = kmem_alloc((count - 1) * sizeof (void *), KM_SLEEP);
5064
5065         for (int i = 0, j = 0; i < count; i++) {
5066                 if (dev[i] == dev_to_remove)
5067                         continue;
5068                 VERIFY(nvlist_dup(dev[i], &newdev[j++], KM_SLEEP) == 0);
5069         }
5070
5071         VERIFY(nvlist_remove(config, name, DATA_TYPE_NVLIST_ARRAY) == 0);
5072         VERIFY(nvlist_add_nvlist_array(config, name, newdev, count - 1) == 0);
5073
5074         for (int i = 0; i < count - 1; i++)
5075                 nvlist_free(newdev[i]);
5076
5077         if (count > 1)
5078                 kmem_free(newdev, (count - 1) * sizeof (void *));
5079 }
5080
5081 /*
5082  * Evacuate the device.
5083  */
5084 static int
5085 spa_vdev_remove_evacuate(spa_t *spa, vdev_t *vd)
5086 {
5087         uint64_t txg;
5088         int error = 0;
5089
5090         ASSERT(MUTEX_HELD(&spa_namespace_lock));
5091         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == 0);
5092         ASSERT(vd == vd->vdev_top);
5093
5094         /*
5095          * Evacuate the device.  We don't hold the config lock as writer
5096          * since we need to do I/O but we do keep the
5097          * spa_namespace_lock held.  Once this completes the device
5098          * should no longer have any blocks allocated on it.
```



```
5099          */
5100         if (vd->vdev_islog) {
5101                 if (vd->vdev_stat.vs_alloc != 0)
5102                         error = spa_offline_log(spa);
5103         } else {
5104                 error = SET_ERROR(ENOTSUP);
5105         }
5106
5107         if (error)
5108                 return (error);
5109
5110         /*
5111          * The evacuation succeeded.  Remove any remaining MOS metadata
5112          * associated with this vdev, and wait for these changes to sync.
5113          */
5114         ASSERT0(vd->vdev_stat.vs_alloc);
5115         txg = spa_vdev_config_enter(spa);
5116         vd->vdev_removing = B_TRUE;
5117         vdev_dirty(vd, 0, NULL, txg);
5118         vdev_config_dirty(vd);
5119         spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
5120
5121         return (0);
5122 }
5123
5124 /*
5125  * Complete the removal by cleaning up the namespace.
5126  */
5127 static void
5128 spa_vdev_remove_from_namespace(spa_t *spa, vdev_t *vd)
5129 {
5130         vdev_t *rvd = spa->spa_root_vdev;
5131         uint64_t id = vd->vdev_id;
5132         boolean_t last_vdev = (id == (rvd->vdev_children - 1));
5133
5134         ASSERT(MUTEX_HELD(&spa_namespace_lock));
5135         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
5136         ASSERT(vd == vd->vdev_top);
5137
5138         /*
5139          * Only remove any devices which are empty.
5140          */
5141         if (vd->vdev_stat.vs_alloc != 0)
5142                 return;
5143
5144         (void) vdev_label_init(vd, 0, VDEV_LABEL_REMOVE);
5145
5146         if (list_link_active(&vd->vdev_state_dirty_node))
5147                 vdev_state_clean(vd);
5148         if (list_link_active(&vd->vdev_config_dirty_node))
5149                 vdev_config_clean(vd);
5150
5151         vdev_free(vd);
5152
5153         if (last_vdev) {
5154                 vdev_compact_children(rvd);
5155         } else {
5156                 vd = vdev_alloc_common(spa, id, 0, &vdev_hole_ops);
5157                 vdev_add_child(rvd, vd);
5158         }
5159         vdev_config_dirty(rvd);
5160
5161         /*
5162          * Reassess the health of our root vdev.
5163          */
5164         vdev_reopen(rvd);
```

```
5165 }

5167 /*
5168  * Remove a device from the pool -
5169  *
5170  * Removing a device from the vdev namespace requires several steps
5171  * and can take a significant amount of time.  As a result we use
5172  * the spa_vdev_config_[enter/exit] functions which allow us to
5173  * grab and release the spa_config_lock while still holding the namespace
5174  * lock.  During each step the configuration is synced out.
5175  *
5176  * Currently, this supports removing only hot spares, slogs, and level 2 ARC
5177  * devices.
4519  */

4521 /*
4522  * Remove a device from the pool.  Currently, this supports removing only hot
4523  * spares, slogs, and level 2 ARC devices.
5178  */
5179 int
5180 spa_vdev_remove(spa_t *spa, uint64_t guid, boolean_t unspare)
5181 {
5182         vdev_t *vd;
5183         metaslab_group_t *mg;
5184         nvlist_t **spares, **l2cache, *nv;
5185         uint64_t txg = 0;
5186         uint_t nspares, nl2cache;
5187         int error = 0;
5188         boolean_t locked = MUTEX_HELD(&spa_namespace_lock);

5190         ASSERT(spa_writeable(spa));

5192         if (!locked)
5193                 txg = spa_vdev_enter(spa);

5195         vd = spa_lookup_by_guid(spa, guid, B_FALSE);

5197         if (spa->spa_spares.sav_vdevs != NULL &&
5198             nvlist_lookup_nvlist_array(spa->spa_spares.sav_config,
5199             ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0 &&
5200             (nv = spa_nvlist_lookup_by_guid(spares, nspares, guid)) != NULL) {
5201                 /*
5202                  * Only remove the hot spare if it's not currently in use
5203                  * in this pool.
5204                  */
5205                 if (vd == NULL || unspare) {
5206                         spa_vdev_remove_aux(spa->spa_spares.sav_config,
5207                             ZPOOL_CONFIG_SPARES, spares, nspares, nv);
5208                         spa_load_spares(spa);
5209                         spa->spa_spares.sav_sync = B_TRUE;
5210                 } else {
5211                         error = SET_ERROR(EBUSY);
5212                 }
5213         } else if (spa->spa_l2cache.sav_vdevs != NULL &&
5214             nvlist_lookup_nvlist_array(spa->spa_l2cache.sav_config,
5215             ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0 &&
5216             (nv = spa_nvlist_lookup_by_guid(l2cache, nl2cache, guid)) != NULL) {
5217                 /*
5218                  * Cache devices can always be removed.
5219                  */
5220                 spa_vdev_remove_aux(spa->spa_l2cache.sav_config,
5221                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache, nv);
5222                 spa_load_l2cache(spa);
5223                 spa->spa_l2cache.sav_sync = B_TRUE;
5224         } else if (vd != NULL && vd->vdev_islog) {
5225                 ASSERT(!locked);
```



```
5226                 ASSERT(vd == vd->vdev_top);

5228                 /*
5229                  * XXX - Once we have bp-rewrite this should
5230                  * become the common case.
5231                  */

5233                 mg = vd->vdev_mg;

5235                 /*
5236                  * Stop allocating from this vdev.
5237                  */
5238                 metaslab_group_passivate(mg);

5240                 /*
5241                  * Wait for the youngest allocations and frees to sync,
5242                  * and then wait for the deferral of those frees to finish.
5243                  */
5244                 spa_vdev_config_exit(spa, NULL,
5245                     txg + TXG_CONCURRENT_STATES + TXG_DEFER_SIZE, 0, FTAG);

5247                 /*
5248                  * Attempt to evacuate the vdev.
5249                  */
5250                 error = spa_vdev_remove_evacuate(spa, vd);

5252                 txg = spa_vdev_config_enter(spa);

5254                 /*
5255                  * If we couldn't evacuate the vdev, unwind.
5256                  */
5257                 if (error) {
5258                         metaslab_group_activate(mg);
5259                         return (spa_vdev_exit(spa, NULL, txg, error));
5260                 }

5262                 /*
5263                  * Clean up the vdev namespace.
5264                  */
5265                 spa_vdev_remove_from_namespace(spa, vd);

5267         } else if (vd != NULL) {
5268                 /*
5269                  * Normal vdevs cannot be removed (yet).
5270                  */
5271                 error = SET_ERROR(ENOTSUP);
5272         } else {
5273                 /*
5274                  * There is no vdev of any kind with the specified guid.
5275                  */
5276                 error = SET_ERROR(ENOENT);
5277         }

5279         if (!locked)
5280                 return (spa_vdev_exit(spa, NULL, txg, error));

5282         return (error);
5283 }

5285 /*
5286  * Find any device that's done replacing, or a vdev marked 'unspare' that's
5287  * currently spared, so we can detach it.
4633  * current spared, so we can detach it.
5288  */
5289 static vdev_t *
5290 spa_vdev_resilver_done_hunt(vdev_t *vd)
```

```
5291 {
5292         vdev_t *newvd, *oldvd;

5294         for (int c = 0; c < vd->vdev_children; c++) {
5295                 oldvd = spa_vdev_resilver_done_hunt(vd->vdev_child[c]);
5296                 if (oldvd != NULL)
5297                         return (oldvd);
5298         }

5300         /*
5301          * Check for a completed replacement.  We always consider the first
5302          * vdev in the list to be the oldest vdev, and the last one to be
5303          * the newest (see spa_vdev_attach() for how that works).  In
5304          * the case where the newest vdev is faulted, we will not automatically
5305          * remove it after a resilver completes.  This is OK as it will require
5306          * user intervention to determine which disk the admin wishes to keep.
5307          */
5308         if (vd->vdev_ops == &vdev_replacing_ops) {
5309                 ASSERT(vd->vdev_children > 1);

5311                 newvd = vd->vdev_child[vd->vdev_children - 1];
5312                 oldvd = vd->vdev_child[0];

5314                 if (vdev_dtl_empty(newvd, DTL_MISSING) &&
5315                     vdev_dtl_empty(newvd, DTL_OUTAGE) &&
5316                     !vdev_dtl_required(oldvd))
5317                         return (oldvd);
5318         }

5320         /*
5321          * Check for a completed resilver with the 'unspare' flag set.
5322          */
5323         if (vd->vdev_ops == &vdev_spare_ops) {
5324                 vdev_t *first = vd->vdev_child[0];
5325                 vdev_t *last = vd->vdev_child[vd->vdev_children - 1];

5327                 if (last->vdev_unspare) {
5328                         oldvd = first;
5329                         newvd = last;
5330                 } else if (first->vdev_unspare) {
5331                         oldvd = last;
5332                         newvd = first;
5333                 } else {
5334                         oldvd = NULL;
5335                 }

5337                 if (oldvd != NULL &&
5338                     vdev_dtl_empty(newvd, DTL_MISSING) &&
5339                     vdev_dtl_empty(newvd, DTL_OUTAGE) &&
5340                     !vdev_dtl_required(oldvd))
5341                         return (oldvd);

5343                 /*
5344                  * If there are more than two spares attached to a disk,
5345                  * and those spares are not required, then we want to
5346                  * attempt to free them up now so that they can be used
5347                  * by other pools.  Once we're back down to a single
5348                  * disk+spare, we stop removing them.
5349                  */
5350                 if (vd->vdev_children > 2) {
5351                         newvd = vd->vdev_child[1];

5353                         if (newvd->vdev_isspare && last->vdev_isspare &&
5354                             vdev_dtl_empty(last, DTL_MISSING) &&
5355                             vdev_dtl_empty(last, DTL_OUTAGE) &&
5356                             !vdev_dtl_required(newvd))
```



```
5357                                 return (newvd);
5358                 }
5359         }

5361         return (NULL);
5362 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   14351 Thu May 16 17:36:18 2013
new/usr/src/uts/common/fs/zfs/spa_config.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 318 /*
 319  * Generate the pool's configuration based on the current in-core state.
 320  *
 321 #endif /* ! codereview */
 322  * We infer whether to generate a complete config or just one top-level config
 323  * based on whether vd is the root vdev.
 324  */
 325 nvlist_t *
 326 spa_config_generate(spa_t *spa, vdev_t *vd, uint64_t txg, int getstats)
 327 {
 328         nvlist_t *config, *nvroot;
 329         vdev_t *rvd = spa->spa_root_vdev;
 330         unsigned long hostid = 0;
 331         boolean_t locked = B_FALSE;
 332         uint64_t split_guid;

 334         if (vd == NULL) {
 335                 vd = rvd;
 336                 locked = B_TRUE;
 337                 spa_config_enter(spa, SCL_CONFIG | SCL_STATE, FTAG, RW_READER);
 338         }

 340         ASSERT(spa_config_held(spa, SCL_CONFIG | SCL_STATE, RW_READER) ==
 341             (SCL_CONFIG | SCL_STATE));

 343         /*
 344          * If txg is -1, report the current value of spa->spa_config_txg.
 345          */
 346         if (txg == -1ULL)
 347                 txg = spa->spa_config_txg;

 349         VERIFY(nvlist_alloc(&config, NV_UNIQUE_NAME, KM_SLEEP) == 0);

 351         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_VERSION,
 352             spa_version(spa)) == 0);
 353         VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_POOL_NAME,
 354             spa_name(spa)) == 0);
 355         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_STATE,
 356             spa_state(spa)) == 0);
 357         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_TXG,
 358             txg) == 0);
 359         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_GUID,
 360             spa_guid(spa)) == 0);
 361         VERIFY(spa->spa_comment == NULL || nvlist_add_string(config,
 362             ZPOOL_CONFIG_COMMENT, spa->spa_comment) == 0);


 365 #ifdef  _KERNEL
 366         hostid = zone_get_hostid(NULL);
 367 #else   /* _KERNEL */
 368         /*
 369          * We're emulating the system's hostid in userland, so we can't use
 370          * zone_get_hostid().
 371          */
```



```
 372         (void) ddi_strtoul(hw_serial, NULL, 10, &hostid);
 373 #endif  /* _KERNEL */
 374         if (hostid != 0) {
 375                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_HOSTID,
 376                     hostid) == 0);
 377         }
 378         VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_HOSTNAME,
 379             utsname.nodename) == 0);

 381         if (vd != rvd) {
 382                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_TOP_GUID,
 383                     vd->vdev_top->vdev_guid) == 0);
 384                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_GUID,
 385                     vd->vdev_guid) == 0);
 386                 if (vd->vdev_isspare)
 387                         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_IS_SPARE,
 388                             1ULL) == 0);
 389                 if (vd->vdev_islog)
 390                         VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_IS_LOG,
 391                             1ULL) == 0);
 392                 vd = vd->vdev_top;              /* label contains top config */
 393         } else {
 394                 /*
 395                  * Only add the (potentially large) split information
 396                  * in the mos config, and not in the vdev labels
 397                  */
 398                 if (spa->spa_config_splitting != NULL)
 399                         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_SPLIT,
 400                             spa->spa_config_splitting) == 0);
 401         }

 403         /*
 404          * Add the top-level config.  We even add this on pools which
 405          * don't support holes in the namespace.
 406          */
 407         vdev_top_config_generate(spa, config);

 409         /*
 410          * If we're splitting, record the original pool's guid.
 411          */
 412         if (spa->spa_config_splitting != NULL &&
 413             nvlist_lookup_uint64(spa->spa_config_splitting,
 414             ZPOOL_CONFIG_SPLIT_GUID, &split_guid) == 0) {
 415                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_SPLIT_GUID,
 416                     split_guid) == 0);
 417         }

 419         nvroot = vdev_config_generate(spa, vd, getstats, 0);
 420         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, nvroot) == 0);
 421         nvlist_free(nvroot);

 423         /*
 424          * Store what's necessary for reading the MOS in the label.
 425          */
 426         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_FEATURES_FOR_READ,
 427             spa->spa_label_features) == 0);

 429         if (getstats && spa_load_state(spa) == SPA_LOAD_NONE) {
 430                 ddt_histogram_t *ddh;
 431                 ddt_stat_t *dds;
 432                 ddt_object_t *ddo;

 434                 ddh = kmem_zalloc(sizeof (ddt_histogram_t), KM_SLEEP);
 435                 ddt_get_dedup_histogram(spa, ddh);
 436                 VERIFY(nvlist_add_uint64_array(config,
 437                     ZPOOL_CONFIG_DDT_HISTOGRAM,
```

```
438                     (uint64_t *)ddh, sizeof (*ddh) / sizeof (uint64_t)) == 0);
439                 kmem_free(ddh, sizeof (ddt_histogram_t));

441                 ddo = kmem_zalloc(sizeof (ddt_object_t), KM_SLEEP);
442                 ddt_get_dedup_object_stats(spa, ddo);
443                 VERIFY(nvlist_add_uint64_array(config,
444                     ZPOOL_CONFIG_DDT_OBJ_STATS,
445                     (uint64_t *)ddo, sizeof (*ddo) / sizeof (uint64_t)) == 0);
446                 kmem_free(ddo, sizeof (ddt_object_t));

448                 dds = kmem_zalloc(sizeof (ddt_stat_t), KM_SLEEP);
449                 ddt_get_dedup_stats(spa, dds);
450                 VERIFY(nvlist_add_uint64_array(config,
451                     ZPOOL_CONFIG_DDT_STATS,
452                     (uint64_t *)dds, sizeof (*dds) / sizeof (uint64_t)) == 0);
453                 kmem_free(dds, sizeof (ddt_stat_t));
454         }

456         if (locked)
457                 spa_config_exit(spa, SCL_CONFIG | SCL_STATE, FTAG);

459         return (config);
460 }

462 /*
463  * Update all disk labels, generate a fresh config based on the current
464  * in-core state, and sync the global config cache (do not sync the config
465  * cache if this is a booting rootpool).
466  */
467 void
468 spa_config_update(spa_t *spa, int what)
469 {
470         vdev_t *rvd = spa->spa_root_vdev;
471         uint64_t txg;
472         int c;

474         ASSERT(MUTEX_HELD(&spa_namespace_lock));

476         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
477         txg = spa_last_synced_txg(spa) + 1;
478         if (what == SPA_CONFIG_UPDATE_POOL) {
479                 vdev_config_dirty(rvd);
480         } else {
481                 /*
482                  * If we have top-level vdevs that were added but have
483                  * not yet been prepared for allocation, do that now.
484                  * (It's safe now because the config cache is up to date,
485                  * so it will be able to translate the new DVAs.)
486                  * See comments in spa_vdev_add() for full details.
487                  */
488                 for (c = 0; c < rvd->vdev_children; c++) {
489                         vdev_t *tvd = rvd->vdev_child[c];
490                         if (tvd->vdev_ms_array == 0)
491                                 vdev_metaslab_set_size(tvd);
492                         vdev_expand(tvd, txg);
493                 }
494         }
495         spa_config_exit(spa, SCL_ALL, FTAG);

497         /*
498          * Wait for the mosconfig to be regenerated and synced.
499          */
500         txg_wait_synced(spa->spa_dsl_pool, txg);

502         /*
503          * Update the global config cache to reflect the new mosconfig.
```



```
504          */
505         if (!spa->spa_is_root)
506                 spa_config_sync(spa, B_FALSE, what != SPA_CONFIG_UPDATE_POOL);

508         if (what == SPA_CONFIG_UPDATE_POOL)
509                 spa_config_update(spa, SPA_CONFIG_UPDATE_VDEVS);
510 }
```

```
**********************************************************
   45913 Thu May 16 17:36:18 2013
new/usr/src/uts/common/fs/zfs/spa_misc.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1335 /*
1336  * This is a stripped-down version of strtoull, suitable only for converting
1337  * lowercase hexadecimal numbers that don't overflow.
1337  * lowercase hexidecimal numbers that don't overflow.
1338  */
1339 uint64_t
1340 strtonum(const char *str, char **nptr)
1341 {
1342         uint64_t val = 0;
1343         char c;
1344         int digit;

1346         while ((c = *str) != '\0') {
1347                 if (c >= '0' && c <= '9')
1348                         digit = c - '0';
1349                 else if (c >= 'a' && c <= 'f')
1350                         digit = 10 + c - 'a';
1351                 else
1352                         break;

1354                 val *= 16;
1355                 val += digit;

1357                 str++;
1358         }

1360         if (nptr)
1361                 *nptr = (char *)str;

1363         return (val);
1364 }
______unchanged_portion_omitted_
```

```
**********************************************************
    7760 Thu May 16 17:36:19 2013
new/usr/src/uts/common/fs/zfs/sys/ddt.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

  56 #define DDT_TYPE_CURRENT                0

  58 #define DDT_COMPRESS_BYTEORDER_MASK     0x80
  59 #define DDT_COMPRESS_FUNCTION_MASK      0x7f

  61 /*
  62  * On-disk ddt entry:  key (name) and physical storage (value).
  63  */
  64 typedef struct ddt_key {
  65         zio_cksum_t     ddk_cksum;      /* 256-bit block checksum */
  66         /*
  67          * Encoded with logical & physical size, and compression, as follows:
  66         uint64_t        ddk_prop;       /* LSIZE, PSIZE, compression */
  67 } ddt_key_t;

  69 /*
  70  * ddk_prop layout:
  71  *
  68          *   +-------+-------+-------+-------+-------+-------+-------+-------+
  69          *   |   0   |   0   |   0   | comp  |     PSIZE     |     LSIZE     |
  70          *   +-------+-------+-------+-------+-------+-------+-------+-------+
  71          */
  72         uint64_t        ddk_prop;
  73 } ddt_key_t;

  75 #endif /* ! codereview */
  76 #define DDK_GET_LSIZE(ddk)      \
  77         BF64_GET_SB((ddk)->ddk_prop, 0, 16, SPA_MINBLOCKSHIFT, 1)
  78 #define DDK_SET_LSIZE(ddk, x)   \
  79         BF64_SET_SB((ddk)->ddk_prop, 0, 16, SPA_MINBLOCKSHIFT, 1, x)

  81 #define DDK_GET_PSIZE(ddk)      \
  82         BF64_GET_SB((ddk)->ddk_prop, 16, 16, SPA_MINBLOCKSHIFT, 1)
  83 #define DDK_SET_PSIZE(ddk, x)   \
  84         BF64_SET_SB((ddk)->ddk_prop, 16, 16, SPA_MINBLOCKSHIFT, 1, x)

  86 #define DDK_GET_COMPRESS(ddk)           BF64_GET((ddk)->ddk_prop, 32, 8)
  87 #define DDK_SET_COMPRESS(ddk, x)        BF64_SET((ddk)->ddk_prop, 32, 8, x)

  89 #define DDT_KEY_WORDS   (sizeof (ddt_key_t) / sizeof (uint64_t))

  91 typedef struct ddt_phys {
  92         dva_t           ddp_dva[SPA_DVAS_PER_BP];
  93         uint64_t        ddp_refcnt;
  94         uint64_t        ddp_phys_birth;
  95 } ddt_phys_t;

  97 enum ddt_phys_type {
  98         DDT_PHYS_DITTO = 0,
  99         DDT_PHYS_SINGLE = 1,
 100         DDT_PHYS_DOUBLE = 2,
 101         DDT_PHYS_TRIPLE = 3,
 102         DDT_PHYS_TYPES
 103 };
```



```
 105 /*
 106  * In-core ddt entry
 107  */
 108 struct ddt_entry {
 109         ddt_key_t       dde_key;
 110         ddt_phys_t      dde_phys[DDT_PHYS_TYPES];
 111         zio_t           *dde_lead_zio[DDT_PHYS_TYPES];
 112         void            *dde_repair_data;
 113         enum ddt_type   dde_type;
 114         enum ddt_class  dde_class;
 115         uint8_t         dde_loading;
 116         uint8_t         dde_loaded;
 117         kcondvar_t      dde_cv;
 118         avl_node_t      dde_node;
 119 };

 121 /*
 122  * In-core ddt
 123  */
 124 struct ddt {
 125         kmutex_t        ddt_lock;
 126         avl_tree_t      ddt_tree;
 127         avl_tree_t      ddt_repair_tree;
 128         enum zio_checksum ddt_checksum;
 129         spa_t           *ddt_spa;
 130         objset_t        *ddt_os;
 131         uint64_t        ddt_stat_object;
 132         uint64_t        ddt_object[DDT_TYPES][DDT_CLASSES];
 133         ddt_histogram_t ddt_histogram[DDT_TYPES][DDT_CLASSES];
 134         ddt_histogram_t ddt_histogram_cache[DDT_TYPES][DDT_CLASSES];
 135         ddt_object_t    ddt_object_stats[DDT_TYPES][DDT_CLASSES];
 136         avl_node_t      ddt_node;
 137 };

 139 /*
 140  * In-core and on-disk bookmark for DDT walks
 141  */
 142 typedef struct ddt_bookmark {
 143         uint64_t        ddb_class;
 144         uint64_t        ddb_type;
 145         uint64_t        ddb_checksum;
 146         uint64_t        ddb_cursor;
 147 } ddt_bookmark_t;

 149 /*
 150  * Ops vector to access a specific DDT object type.
 151  */
 152 typedef struct ddt_ops {
 153         char ddt_op_name[32];
 154         int (*ddt_op_create)(objset_t *os, uint64_t *object, dmu_tx_t *tx,
 155             boolean_t prehash);
 156         int (*ddt_op_destroy)(objset_t *os, uint64_t object, dmu_tx_t *tx);
 157         int (*ddt_op_lookup)(objset_t *os, uint64_t object, ddt_entry_t *dde);
 158         void (*ddt_op_prefetch)(objset_t *os, uint64_t object,
 159             ddt_entry_t *dde);
 160         int (*ddt_op_update)(objset_t *os, uint64_t object, ddt_entry_t *dde,
 161             dmu_tx_t *tx);
 162         int (*ddt_op_remove)(objset_t *os, uint64_t object, ddt_entry_t *dde,
 163             dmu_tx_t *tx);
 164         int (*ddt_op_walk)(objset_t *os, uint64_t object, ddt_entry_t *dde,
 165             uint64_t *walk);
 166         uint64_t (*ddt_op_count)(objset_t *os, uint64_t object);
 167 } ddt_ops_t;

 169 #define DDT_NAMELEN     80
```

```
171 extern void ddt_object_name(ddt_t *ddt, enum ddt_type type,
172     enum ddt_class class, char *name);
173 extern int ddt_object_walk(ddt_t *ddt, enum ddt_type type,
174     enum ddt_class class, uint64_t *walk, ddt_entry_t *dde);
175 extern uint64_t ddt_object_count(ddt_t *ddt, enum ddt_type type,
176     enum ddt_class class);
177 extern int ddt_object_info(ddt_t *ddt, enum ddt_type type,
178     enum ddt_class class, dmu_object_info_t *);
179 extern boolean_t ddt_object_exists(ddt_t *ddt, enum ddt_type type,
180     enum ddt_class class);

182 extern void ddt_bp_fill(const ddt_phys_t *ddp, blkptr_t *bp,
183     uint64_t txg);
184 extern void ddt_bp_create(enum zio_checksum checksum, const ddt_key_t *ddk,
185     const ddt_phys_t *ddp, blkptr_t *bp);

187 extern void ddt_key_fill(ddt_key_t *ddk, const blkptr_t *bp);

189 extern void ddt_phys_fill(ddt_phys_t *ddp, const blkptr_t *bp);
190 extern void ddt_phys_clear(ddt_phys_t *ddp);
191 extern void ddt_phys_addref(ddt_phys_t *ddp);
192 extern void ddt_phys_decref(ddt_phys_t *ddp);
193 extern void ddt_phys_free(ddt_t *ddt, ddt_key_t *ddk, ddt_phys_t *ddp,
194     uint64_t txg);
195 extern ddt_phys_t *ddt_phys_select(const ddt_entry_t *dde, const blkptr_t *bp);
196 extern uint64_t ddt_phys_total_refcnt(const ddt_entry_t *dde);

198 extern void ddt_stat_add(ddt_stat_t *dst, const ddt_stat_t *src, uint64_t neg);

200 extern void ddt_histogram_add(ddt_histogram_t *dst, const ddt_histogram_t *src);
201 extern void ddt_histogram_stat(ddt_stat_t *dds, const ddt_histogram_t *ddh);
202 extern boolean_t ddt_histogram_empty(const ddt_histogram_t *ddh);
203 extern void ddt_get_dedup_object_stats(spa_t *spa, ddt_object_t *ddo);
204 extern void ddt_get_dedup_histogram(spa_t *spa, ddt_histogram_t *ddh);
205 extern void ddt_get_dedup_stats(spa_t *spa, ddt_stat_t *dds_total);

207 extern uint64_t ddt_get_dedup_dspace(spa_t *spa);
208 extern uint64_t ddt_get_pool_dedup_ratio(spa_t *spa);

210 extern int ddt_ditto_copies_needed(ddt_t *ddt, ddt_entry_t *dde,
211     ddt_phys_t *ddp_willref);
212 extern int ddt_ditto_copies_present(ddt_entry_t *dde);

214 extern size_t ddt_compress(void *src, uchar_t *dst, size_t s_len, size_t d_len);
215 extern void ddt_decompress(uchar_t *src, void *dst, size_t s_len, size_t d_len);

217 extern ddt_t *ddt_select(spa_t *spa, const blkptr_t *bp);
218 extern void ddt_enter(ddt_t *ddt);
219 extern void ddt_exit(ddt_t *ddt);
220 extern ddt_entry_t *ddt_lookup(ddt_t *ddt, const blkptr_t *bp, boolean_t add);
221 extern void ddt_prefetch(spa_t *spa, const blkptr_t *bp);
222 extern void ddt_remove(ddt_t *ddt, ddt_entry_t *dde);

224 extern boolean_t ddt_class_contains(spa_t *spa, enum ddt_class max_class,
225     const blkptr_t *bp);

227 extern ddt_entry_t *ddt_repair_start(ddt_t *ddt, const blkptr_t *bp);
228 extern void ddt_repair_done(ddt_t *ddt, ddt_entry_t *dde);

230 extern int ddt_entry_compare(const void *x1, const void *x2);

232 extern void ddt_create(spa_t *spa);
233 extern int ddt_load(spa_t *spa);
234 extern void ddt_unload(spa_t *spa);
235 extern void ddt_sync(spa_t *spa, uint64_t txg);
```



```
236 extern int ddt_walk(spa_t *spa, ddt_bookmark_t *ddb, ddt_entry_t *dde);
237 extern int ddt_object_update(ddt_t *ddt, enum ddt_type type,
238     enum ddt_class class, ddt_entry_t *dde, dmu_tx_t *tx);

240 extern const ddt_ops_t ddt_zap_ops;

242 #ifdef  __cplusplus
243 }
244 #endif

246 #endif  /* _SYS_DDT_H */
```

```
**********************************************************
   10546 Thu May 16 17:36:19 2013
new/usr/src/uts/common/fs/zfs/sys/dnode.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 146 typedef struct dnode {
 147         /*
 148          * Protects the structure of the dnode, including the number of levels
 149          * of indirection (dn_nlevels), dn_maxblkid, and dn_next_*
 148          * dn_struct_rwlock protects the structure of the dnode,
 149          * including the number of levels of indirection (dn_nlevels),
 150          * dn_maxblkid, and dn_next_*
 150          */
 151         krwlock_t dn_struct_rwlock;

 153         /* Our link on dn_objset->os_dnodes list; protected by os_lock.  */
 154         list_node_t dn_link;

 156         /* immutable: */
 157         struct objset *dn_objset;
 158         uint64_t dn_object;
 159         struct dmu_buf_impl *dn_dbuf;
 160         struct dnode_handle *dn_handle;
 161         dnode_phys_t *dn_phys; /* pointer into dn->dn_dbuf->db.db_data */

 163         /*
 164          * Copies of stuff in dn_phys.  They're valid in the open
 165          * context (eg. even before the dnode is first synced).
 166          * Where necessary, these are protected by dn_struct_rwlock.
 167          */
 168         dmu_object_type_t dn_type;      /* object type */
 169         uint16_t dn_bonuslen;           /* bonus length */
 170         uint8_t dn_bonustype;           /* bonus type */
 171         uint8_t dn_nblkptr;             /* number of blkptrs (immutable) */
 172         uint8_t dn_checksum;            /* ZIO_CHECKSUM type */
 173         uint8_t dn_compress;            /* ZIO_COMPRESS type */
 174         uint8_t dn_nlevels;
 175         uint8_t dn_indblkshift;
 176         uint8_t dn_datablkshift;        /* zero if blksz not power of 2! */
 177         uint8_t dn_moved;               /* Has this dnode been moved? */
 178         uint16_t dn_datablkszsec;       /* in 512b sectors */
 179         uint32_t dn_datablksz;          /* in bytes */
 180         uint64_t dn_maxblkid;
 181         uint8_t dn_next_nblkptr[TXG_SIZE];
 182         uint8_t dn_next_nlevels[TXG_SIZE];
 183         uint8_t dn_next_indblkshift[TXG_SIZE];
 184         uint8_t dn_next_bonustype[TXG_SIZE];
 185         uint8_t dn_rm_spillblk[TXG_SIZE];       /* for removing spill blk */
 186         uint16_t dn_next_bonuslen[TXG_SIZE];
 187         uint32_t dn_next_blksz[TXG_SIZE];       /* next block size in bytes */

 189         /* protected by dn_dbufs_mtx; declared here to fill 32-bit hole */
 190         uint32_t dn_dbufs_count;        /* count of dn_dbufs */

 192         /* protected by os_lock: */
 193         list_node_t dn_dirty_link[TXG_SIZE];    /* next on dataset's dirty */

 195         /* protected by dn_mtx: */
 196         kmutex_t dn_mtx;
```



```
 197         list_t dn_dirty_records[TXG_SIZE];
 198         avl_tree_t dn_ranges[TXG_SIZE];
 199         uint64_t dn_allocated_txg;
 200         uint64_t dn_free_txg;
 201         uint64_t dn_assigned_txg;
 202         kcondvar_t dn_notxholds;
 203         enum dnode_dirtycontext dn_dirtyctx;
 204         uint8_t *dn_dirtyctx_firstset;          /* dbg: contents meaningless */

 206         /* protected by own devices */
 207         refcount_t dn_tx_holds;
 208         refcount_t dn_holds;

 210         kmutex_t dn_dbufs_mtx;
 211         list_t dn_dbufs;                /* descendent dbufs */

 213         /* protected by dn_struct_rwlock */
 214         struct dmu_buf_impl *dn_bonus;  /* bonus buffer dbuf */

 216         boolean_t dn_have_spill;        /* have spill or are spilling */

 218         /* parent IO for current sync write */
 219         zio_t *dn_zio;

 221         /* used in syncing context */
 222         uint64_t dn_oldused;    /* old phys used bytes */
 223         uint64_t dn_oldflags;   /* old phys dn_flags */
 224         uint64_t dn_olduid, dn_oldgid;
 225         uint64_t dn_newuid, dn_newgid;
 226         int dn_id_flags;

 228         /* holds prefetch structure */
 229         struct zfetch   dn_zfetch;
 230 } dnode_t;
______unchanged_portion_omitted_
```

```
**********************************************************
    5201 Thu May 16 17:36:19 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_pool.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_


typedef struct dsl_pool {
        /* Immutable */
        spa_t *dp_spa;
        struct objset *dp_meta_objset;
        struct dsl_dir *dp_root_dir;
        struct dsl_dir *dp_mos_dir;
        struct dsl_dir *dp_free_dir;
        struct dsl_dataset *dp_origin_snap;
        uint64_t dp_root_dir_obj;
        struct taskq *dp_vnrele_taskq;

        /* No lock needed - sync context only */
        blkptr_t dp_meta_rootbp;
        hrtime_t dp_read_overhead;
        uint64_t dp_throughput; /* bytes per millisec */
        uint64_t dp_write_limit;
        uint64_t dp_tmp_userrefs_obj;
        bpobj_t dp_free_bpobj;
        uint64_t dp_bptree_obj;
        uint64_t dp_empty_bpobj;

        struct dsl_scan *dp_scan;

        /* Uses dp_lock */
        kmutex_t dp_lock;
        uint64_t dp_space_towrite[TXG_SIZE];
        uint64_t dp_tempreserved[TXG_SIZE];
        uint64_t dp_mos_used_delta;
        uint64_t dp_mos_compressed_delta;
        uint64_t dp_mos_uncompressed_delta;

        /* Has its own locking */
        tx_state_t dp_tx;
        txg_list_t dp_dirty_datasets;
        txg_list_t dp_dirty_zilogs;
        txg_list_t dp_dirty_dirs;
        txg_list_t dp_sync_tasks;

        /*
         * Protects administrative changes (properties, namespace)
         *
#endif /* ! codereview */
         * It is only held for write in syncing context.  Therefore
         * syncing context does not need to ever have it for read, since
         * nobody else could possibly have it for write.
         */
        rrwlock_t dp_config_rwlock;

        zfs_all_blkstats_t *dp_blkstats;
} dsl_pool_t;

int dsl_pool_init(spa_t *spa, uint64_t txg, dsl_pool_t **dpp);
int dsl_pool_open(dsl_pool_t *dp);
```



```
void dsl_pool_close(dsl_pool_t *dp);
dsl_pool_t *dsl_pool_create(spa_t *spa, nvlist_t *zplprops, uint64_t txg);
void dsl_pool_sync(dsl_pool_t *dp, uint64_t txg);
void dsl_pool_sync_done(dsl_pool_t *dp, uint64_t txg);
int dsl_pool_sync_context(dsl_pool_t *dp);
uint64_t dsl_pool_adjustedsize(dsl_pool_t *dp, boolean_t netfree);
uint64_t dsl_pool_adjustedfree(dsl_pool_t *dp, boolean_t netfree);
int dsl_pool_tempreserve_space(dsl_pool_t *dp, uint64_t space, dmu_tx_t *tx);
void dsl_pool_tempreserve_clear(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx);
void dsl_pool_memory_pressure(dsl_pool_t *dp);
void dsl_pool_willuse_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx);
void dsl_free(dsl_pool_t *dp, uint64_t txg, const blkptr_t *bpp);
void dsl_free_sync(zio_t *pio, dsl_pool_t *dp, uint64_t txg,
    const blkptr_t *bpp);
void dsl_pool_create_origin(dsl_pool_t *dp, dmu_tx_t *tx);
void dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx);
void dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx);
void dsl_pool_mos_diduse_space(dsl_pool_t *dp,
    int64_t used, int64_t comp, int64_t uncomp);
void dsl_pool_config_enter(dsl_pool_t *dp, void *tag);
void dsl_pool_config_exit(dsl_pool_t *dp, void *tag);
boolean_t dsl_pool_config_held(dsl_pool_t *dp);

taskq_t *dsl_pool_vnrele_taskq(dsl_pool_t *dp);

int dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj,
    const char *tag, uint64_t now, dmu_tx_t *tx);
int dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj,
    const char *tag, dmu_tx_t *tx);
void dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp);
int dsl_pool_open_special_dir(dsl_pool_t *dp, const char *name, dsl_dir_t **);
int dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp);
void dsl_pool_rele(dsl_pool_t *dp, void *tag);

#ifdef  __cplusplus
}
#endif

#endif /* _SYS_DSL_POOL_H */
```

```
**********************************************************
    8379 Thu May 16 17:36:20 2013
new/usr/src/uts/common/fs/zfs/sys/sa_impl.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

151 /*
152  * header for all bonus and spill buffers.
153  *
154 #endif /* ! codereview */
155  * The header has a fixed portion with a variable number
156  * of "lengths" depending on the number of variable sized
157  * attribues which are determined by the "layout number"
158  */

160 #define SA_MAGIC        0x2F505A  /* ZFS SA */
161 typedef struct sa_hdr_phys {
162         uint32_t sa_magic;
163         /*
164          * Encoded with hdrsize and layout number as follows:
153         uint16_t sa_layout_info;  /* Encoded with hdrsize and layout number */
154         uint16_t sa_lengths[1]; /* optional sizes for variable length attrs */
155         /* ... Data follows the lengths.  */
156 } sa_hdr_phys_t;

158 /*
159  * sa_hdr_phys -> sa_layout_info
160  *
165          * 16      10       0
166          * +--------+-------+
167          * | hdrsz  |layout |
168          * +--------+-------+
169          *
170          * Bits 0-10 are the layout number
171          * Bits 11-16 are the size of the header.
172          * The hdrsize is the number * 8
173          *
174          * For example.
175          * hdrsz of 1 ==> 8 byte header
176          *          2 ==> 16 byte header
177          *
178          */
179         uint16_t sa_layout_info;
180         uint16_t sa_lengths[1]; /* optional sizes for variable length attrs */
181         /* ... Data follows the lengths.  */
182 } sa_hdr_phys_t;
183 #endif /* ! codereview */

185 #define SA_HDR_LAYOUT_NUM(hdr) BF32_GET(hdr->sa_layout_info, 0, 10)
186 #define SA_HDR_SIZE(hdr) BF32_GET_SB(hdr->sa_layout_info, 10, 6, 3, 0)
187 #define SA_HDR_LAYOUT_INFO_ENCODE(x, num, size) \
188 { \
189         BF32_SET_SB(x, 10, 6, 3, 0, size); \
190         BF32_SET(x, 0, 10, num); \
191 }

193 typedef enum sa_buf_type {
194         SA_BONUS = 1,
195         SA_SPILL = 2
196 } sa_buf_type_t;
```



```
198 typedef enum sa_data_op {
199         SA_LOOKUP,
200         SA_UPDATE,
201         SA_ADD,
202         SA_REPLACE,
203         SA_REMOVE
204 } sa_data_op_t;

206 /*
207  * Opaque handle used for most sa functions
208  *
209  * This needs to be kept as small as possible.
210  */

212 struct sa_handle {
213         kmutex_t        sa_lock;
214         dmu_buf_t       *sa_bonus;
215         dmu_buf_t       *sa_spill;
216         objset_t        *sa_os;
217         void            *sa_userp;
218         sa_idx_tab_t    *sa_bonus_tab;   /* idx of bonus */
219         sa_idx_tab_t    *sa_spill_tab; /* only present if spill activated */
220 };

222 #define SA_GET_DB(hdl, type)    \
223         (dmu_buf_impl_t *)((type == SA_BONUS) ? hdl->sa_bonus : hdl->sa_spill)

225 #define SA_GET_HDR(hdl, type) \
226         ((sa_hdr_phys_t *)((dmu_buf_impl_t *)(SA_GET_DB(hdl, \
227         type))->db.db_data))

229 #define SA_IDX_TAB_GET(hdl, type) \
230         (type == SA_BONUS ? hdl->sa_bonus_tab : hdl->sa_spill_tab)

232 #define IS_SA_BONUSTYPE(a)      \
233         ((a == DMU_OT_SA) ? B_TRUE : B_FALSE)

235 #define SA_BONUSTYPE_FROM_DB(db) \
236         (dmu_get_bonustype((dmu_buf_t *)db))

238 #define SA_BLKPTR_SPACE (DN_MAX_BONUSLEN - sizeof (blkptr_t))

240 #define SA_LAYOUT_NUM(x, type) \
241         ((!IS_SA_BONUSTYPE(type) ? 0 : (((IS_SA_BONUSTYPE(type)) && \
242         ((SA_HDR_LAYOUT_NUM(x)) == 0)) ? 1 : SA_HDR_LAYOUT_NUM(x))))


245 #define SA_REGISTERED_LEN(sa, attr) sa->sa_attr_table[attr].sa_length

247 #define SA_ATTR_LEN(sa, idx, attr, hdr) ((SA_REGISTERED_LEN(sa, attr) == 0) ?\
248         hdr->sa_lengths[TOC_LEN_IDX(idx->sa_idx_tab[attr])] : \
249         SA_REGISTERED_LEN(sa, attr))

251 #define SA_SET_HDR(hdr, num, size) \
252         { \
253                 hdr->sa_magic = SA_MAGIC; \
254                 SA_HDR_LAYOUT_INFO_ENCODE(hdr->sa_layout_info, num, size); \
255         }

257 #define SA_ATTR_INFO(sa, idx, hdr, attr, bulk, type, hdl) \
258         { \
259                 bulk.sa_size = SA_ATTR_LEN(sa, idx, attr, hdr); \
260                 bulk.sa_buftype = type; \
261                 bulk.sa_addr = \
262                     (void *)((uintptr_t)TOC_OFF(idx->sa_idx_tab[attr]) + \
```

```
263                     (uintptr_t)hdr); \
264 }

266 #define SA_HDR_SIZE_MATCH_LAYOUT(hdr, tb) \
267         (SA_HDR_SIZE(hdr) == (sizeof (sa_hdr_phys_t) + \
268         (tb->lot_var_sizes > 1 ? P2ROUNDUP((tb->lot_var_sizes - 1) * \
269         sizeof (uint16_t), 8) : 0)))

271 int sa_add_impl(sa_handle_t *, sa_attr_type_t,
272     uint32_t, sa_data_locator_t, void *, dmu_tx_t *);

274 void sa_register_update_callback_locked(objset_t *, sa_update_cb_t *);
275 int sa_size_locked(sa_handle_t *, sa_attr_type_t, int *);

277 void sa_default_locator(void **, uint32_t *, uint32_t, boolean_t, void *);
278 int sa_attr_size(sa_os_t *, sa_idx_tab_t *, sa_attr_type_t,
279     uint16_t *, sa_hdr_phys_t *);

281 #ifdef  __cplusplus
282 extern "C" {
283 #endif

285 #ifdef  __cplusplus
286 }
287 #endif

289 #endif  /* _SYS_SA_IMPL_H */
```

```
**********************************************************
   10869 Thu May 16 17:36:20 2013
new/usr/src/uts/common/fs/zfs/sys/spa_impl.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 115 struct spa {
 116         /*
 117          * Fields protected by spa_namespace_lock.
 118          */
 119         char            spa_name[MAXNAMELEN];   /* pool name */
 120         char            *spa_comment;           /* comment */
 121         avl_node_t      spa_avl;                /* node in spa_namespace_avl */
 122         nvlist_t        *spa_config;            /* last synced config */
 123         nvlist_t        *spa_config_syncing;    /* currently syncing config */
 124         nvlist_t        *spa_config_splitting;  /* config for splitting */
 125         nvlist_t        *spa_load_info;         /* info and errors from load */
 126         uint64_t        spa_config_txg;         /* txg of last config change */
 127         int             spa_sync_pass;          /* iterate-to-convergence */
 128         pool_state_t    spa_state;              /* pool state */
 129         int             spa_inject_ref;         /* injection references */
 130         uint8_t         spa_sync_on;            /* sync threads are running */
 131         spa_load_state_t spa_load_state;        /* current load operation */
 132         uint64_t        spa_import_flags;       /* import specific flags */
 133         spa_taskqs_t    spa_zio_taskq[ZIO_TYPES][ZIO_TASKQ_TYPES];
 134         dsl_pool_t      *spa_dsl_pool;
 135         boolean_t       spa_is_initializing;    /* true while opening pool */
 136         metaslab_class_t *spa_normal_class;     /* normal data class */
 137         metaslab_class_t *spa_log_class;        /* intent log data class */
 138         uint64_t        spa_first_txg;          /* first txg after spa_open() */
 139         uint64_t        spa_final_txg;          /* txg of export/destroy */
 140         uint64_t        spa_freeze_txg;         /* freeze pool at this txg */
 141         uint64_t        spa_load_max_txg;       /* best initial ub_txg */
 142         uint64_t        spa_claim_max_txg;      /* highest claimed birth txg */
 143         timespec_t      spa_loaded_ts;          /* 1st successful open time */
 144         objset_t        *spa_meta_objset;       /* copy of dp->dp_meta_objset */
 145         txg_list_t      spa_vdev_txg_list;      /* per-txg dirty vdev list */
 146         vdev_t          *spa_root_vdev;         /* top-level vdev container */
 147         uint64_t        spa_config_guid;        /* config pool guid */
 148         uint64_t        spa_load_guid;          /* spa_load initialized guid */
 149         uint64_t        spa_last_synced_guid;   /* last synced guid */
 150         list_t          spa_config_dirty_list;  /* vdevs with dirty config */
 151         list_t          spa_state_dirty_list;   /* vdevs with dirty state */
 152         spa_aux_vdev_t  spa_spares;             /* hot spares */
 153         spa_aux_vdev_t  spa_l2cache;            /* L2ARC cache devices */
 154         nvlist_t        *spa_label_features;    /* Features for reading MOS */
 155         uint64_t        spa_config_object;      /* MOS object for pool config */
 156         uint64_t        spa_config_generation;  /* config generation number */
 157         uint64_t        spa_syncing_txg;        /* txg currently syncing */
 158         bpobj_t         spa_deferred_bpobj;     /* deferred-free bplist */
 159         bplist_t        spa_free_bplist[TXG_SIZE]; /* bplist of stuff to free */
 160         uberblock_t     spa_ubsync;             /* last synced uberblock */
 161         uberblock_t     spa_uberblock;          /* current uberblock */
 162         boolean_t       spa_extreme_rewind;     /* rewind past deferred frees */
 163         uint64_t        spa_last_io;            /* lbolt of last non-scan I/O */
 164         kmutex_t        spa_scrub_lock;         /* resilver/scrub lock */
 165         uint64_t        spa_scrub_inflight;     /* in-flight scrub I/Os */
 166         kcondvar_t      spa_scrub_io_cv;        /* scrub I/O completion */
 167         uint8_t         spa_scrub_active;       /* active or suspended? */
 168         uint8_t         spa_scrub_type;         /* type of scrub we're doing */
```



```
 169         uint8_t         spa_scrub_finished;     /* indicator to rotate logs */
 170         uint8_t         spa_scrub_started;      /* started since last boot */
 171         uint8_t         spa_scrub_reopen;       /* scrub doing vdev_reopen */
 172         uint64_t        spa_scan_pass_start;    /* start time per pass/reboot */
 173         uint64_t        spa_scan_pass_exam;     /* examined bytes per pass */
 174         kmutex_t        spa_async_lock;         /* protect async state */
 175         kthread_t       *spa_async_thread;      /* thread doing async task */
 176         int             spa_async_suspended;    /* async tasks suspended */
 177         kcondvar_t      spa_async_cv;           /* wait for thread_exit() */
 178         uint16_t        spa_async_tasks;        /* async task mask */
 179         char            *spa_root;              /* alternate root directory */
 180         uint64_t        spa_ena;                /* spa-wide ereport ENA */
 181         int             spa_last_open_failed;   /* error if last open failed */
 182         uint64_t        spa_last_ubsync_txg;    /* "best" uberblock txg */
 183         uint64_t        spa_last_ubsync_txg_ts; /* timestamp from that ub */
 184         uint64_t        spa_load_txg;           /* ub txg that loaded */
 185         uint64_t        spa_load_txg_ts;        /* timestamp from that ub */
 186         uint64_t        spa_load_meta_errors;   /* verify metadata err count */
 187         uint64_t        spa_load_data_errors;   /* verify data err count */
 188         uint64_t        spa_verify_min_txg;     /* start txg of verify scrub */
 189         kmutex_t        spa_errlog_lock;        /* error log lock */
 190         uint64_t        spa_errlog_last;        /* last error log object */
 191         uint64_t        spa_errlog_scrub;       /* scrub error log object */
 192         kmutex_t        spa_errlist_lock;       /* error list/ereport lock */
 193         avl_tree_t      spa_errlist_last;       /* last error list */
 194         avl_tree_t      spa_errlist_scrub;      /* scrub error list */
 195         uint64_t        spa_deflate;            /* should we deflate? */
 196         uint64_t        spa_history;            /* history object */
 197         kmutex_t        spa_history_lock;       /* history lock */
 198         vdev_t          *spa_pending_vdev;      /* pending vdev additions */
 199         kmutex_t        spa_props_lock;         /* property lock */
 200         uint64_t        spa_pool_props_object;  /* object for properties */
 201         uint64_t        spa_bootfs;             /* default boot filesystem */
 202         uint64_t        spa_failmode;           /* failure mode for the pool */
 203         uint64_t        spa_delegation;         /* delegation on/off */
 204         list_t          spa_config_list;        /* previous cache file(s) */
 205         zio_t           *spa_async_zio_root;    /* root of all async I/O */
 206         zio_t           *spa_suspend_zio_root;  /* root of all suspended I/O */
 207         kmutex_t        spa_suspend_lock;       /* protects suspend_zio_root */
 208         kcondvar_t      spa_suspend_cv;         /* notification of resume */
 209         uint8_t         spa_suspended;          /* pool is suspended */
 210         uint8_t         spa_claiming;           /* pool is doing zil_claim() */
 211         boolean_t       spa_debug;              /* debug enabled? */
 212         boolean_t       spa_is_root;            /* pool is root */
 213         int             spa_minref;             /* num refs when first opened */
 214         int             spa_mode;               /* FREAD | FWRITE */
 215         spa_log_state_t spa_log_state;          /* log state */
 216         uint64_t        spa_autoexpand;         /* lun expansion on/off */
 217         ddt_t           *spa_ddt[ZIO_CHECKSUM_FUNCTIONS]; /* in-core DDTs */
 218         uint64_t        spa_ddt_stat_object;    /* DDT statistics */
 219         uint64_t        spa_dedup_ditto;        /* dedup ditto threshold */
 220         uint64_t        spa_dedup_checksum;     /* default dedup checksum */
 221         uint64_t        spa_dspace;             /* dspace in normal class */
 222         kmutex_t        spa_vdev_top_lock;      /* dueling offline/remove */
 223         kmutex_t        spa_proc_lock;          /* protects spa_proc* */
 224         kcondvar_t      spa_proc_cv;            /* spa_proc_state transitions */
 225         spa_proc_state_t spa_proc_state;        /* see definition */
 226         struct proc     *spa_proc;              /* "zpool-poolname" process */
 227         uint64_t        spa_did;                /* if procp != p0, did of t1 */
 228         boolean_t       spa_autoreplace;        /* autoreplace set in open */
 229         int             spa_vdev_locks;         /* locks grabbed */
 230         uint64_t        spa_creation_version;   /* version at pool creation */
 231         uint64_t        spa_prev_software_version; /* See ub_software_version */
 232         uint64_t        spa_feat_for_write_obj; /* required to write to pool */
 233         uint64_t        spa_feat_for_read_obj;  /* required to read from pool */
 234         uint64_t        spa_feat_desc_obj;      /* Feature descriptions */
```

```
235         cyclic_id_t     spa_deadman_cycid;      /* cyclic id */
236         uint64_t        spa_deadman_calls;      /* number of deadman calls */
237         uint64_t        spa_sync_starttime;     /* starting time fo spa_sync */
238         uint64_t        spa_deadman_synctime;   /* deadman expiration timer */
239         kmutex_t        spa_iokstat_lock;       /* protects spa_iokstat_* */
240         struct kstat    *spa_iokstat;           /* kstat of io to this pool */
241         /*
242          * spa_refcount & spa_config_lock must be the last elements
242          * spa_refcnt & spa_config_lock must be the last elements
243          * because refcount_t changes size based on compilation options.
244          * In order for the MDB module to function correctly, the other
245          * fields must remain in the same location.
246          */
247         spa_config_lock_t spa_config_lock[SCL_LOCKS]; /* config changes */
248         refcount_t      spa_refcount;           /* number of opens */
249 };
```

*unchanged_portion_omitted_*

```
**********************************************************
    6907 Thu May 16 17:36:20 2013
new/usr/src/uts/common/fs/zfs/sys/space_map.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 87 /*
 88  * debug entry
 89  *
 90  *    1      3         10                    50
 91  *  ,---+--------+------------+---------------------------------.
 92  *  | 1 | action |  syncpass  |        txg (lower bits)         |
 93  *  `---+--------+------------+---------------------------------'
 94  *   63  62    60 59        50 49                               0
 95  *
 96  *
 97  *
 97  * non-debug entry
 98  *
 99  *    1               47                   1           15
100  *  ,-----------------------------------------------------------.
101  *  | 0 |   offset (sm_shift units)    | type |       run       |
102  *  `-----------------------------------------------------------'
103  *   63  62                          17   16   15               0
104  */

106 /* All this stuff takes and returns bytes */
107 #define SM_RUN_DECODE(x)        (BF64_DECODE(x, 0, 15) + 1)
108 #define SM_RUN_ENCODE(x)        BF64_ENCODE((x) - 1, 0, 15)
109 #define SM_TYPE_DECODE(x)       BF64_DECODE(x, 15, 1)
110 #define SM_TYPE_ENCODE(x)       BF64_ENCODE(x, 15, 1)
111 #define SM_OFFSET_DECODE(x)     BF64_DECODE(x, 16, 47)
112 #define SM_OFFSET_ENCODE(x)     BF64_ENCODE(x, 16, 47)
113 #define SM_DEBUG_DECODE(x)      BF64_DECODE(x, 63, 1)
114 #define SM_DEBUG_ENCODE(x)      BF64_ENCODE(x, 63, 1)

116 #define SM_DEBUG_ACTION_DECODE(x)       BF64_DECODE(x, 60, 3)
117 #define SM_DEBUG_ACTION_ENCODE(x)       BF64_ENCODE(x, 60, 3)

119 #define SM_DEBUG_SYNCPASS_DECODE(x)     BF64_DECODE(x, 50, 10)
120 #define SM_DEBUG_SYNCPASS_ENCODE(x)     BF64_ENCODE(x, 50, 10)

122 #define SM_DEBUG_TXG_DECODE(x)          BF64_DECODE(x, 0, 50)
123 #define SM_DEBUG_TXG_ENCODE(x)          BF64_ENCODE(x, 0, 50)

125 #define SM_RUN_MAX                      SM_RUN_DECODE(~0ULL)

127 #define SM_ALLOC        0x0
128 #define SM_FREE         0x1

130 /*
131  * The data for a given space map can be kept on blocks of any size.
132  * Larger blocks entail fewer i/o operations, but they also cause the
133  * DMU to keep more data in-core, and also to waste more i/o bandwidth
134  * when only a few blocks have changed since the last transaction group.
135  * This could use a lot more research, but for now, set the freelist
136  * block size to 4k (2^12).
137  */
138 #define SPACE_MAP_BLOCKSHIFT    12
```



```
140 typedef void space_map_func_t(space_map_t *sm, uint64_t start, uint64_t size);

142 extern void space_map_init(void);
143 extern void space_map_fini(void);
144 extern void space_map_create(space_map_t *sm, uint64_t start, uint64_t size,
145     uint8_t shift, kmutex_t *lp);
146 extern void space_map_destroy(space_map_t *sm);
147 extern void space_map_add(space_map_t *sm, uint64_t start, uint64_t size);
148 extern void space_map_remove(space_map_t *sm, uint64_t start, uint64_t size);
149 extern boolean_t space_map_contains(space_map_t *sm,
150     uint64_t start, uint64_t size);
151 extern space_seg_t *space_map_find(space_map_t *sm, uint64_t start,
152     uint64_t size, avl_index_t *wherep);
153 extern void space_map_swap(space_map_t **msrc, space_map_t **mdest);
154 extern void space_map_vacate(space_map_t *sm,
155     space_map_func_t *func, space_map_t *mdest);
156 extern void space_map_walk(space_map_t *sm,
157     space_map_func_t *func, space_map_t *mdest);

159 extern void space_map_load_wait(space_map_t *sm);
160 extern int space_map_load(space_map_t *sm, space_map_ops_t *ops,
161     uint8_t maptype, space_map_obj_t *smo, objset_t *os);
162 extern void space_map_unload(space_map_t *sm);

164 extern uint64_t space_map_alloc(space_map_t *sm, uint64_t size);
165 extern void space_map_claim(space_map_t *sm, uint64_t start, uint64_t size);
166 extern void space_map_free(space_map_t *sm, uint64_t start, uint64_t size);
167 extern uint64_t space_map_maxsize(space_map_t *sm);

169 extern void space_map_sync(space_map_t *sm, uint8_t maptype,
170     space_map_obj_t *smo, objset_t *os, dmu_tx_t *tx);
171 extern void space_map_truncate(space_map_obj_t *smo,
172     objset_t *os, dmu_tx_t *tx);

174 extern void space_map_ref_create(avl_tree_t *t);
175 extern void space_map_ref_destroy(avl_tree_t *t);
176 extern void space_map_ref_add_seg(avl_tree_t *t,
177     uint64_t start, uint64_t end, int64_t refcnt);
178 extern void space_map_ref_add_map(avl_tree_t *t,
179     space_map_t *sm, int64_t refcnt);
180 extern void space_map_ref_generate_map(avl_tree_t *t,
181     space_map_t *sm, int64_t minref);

183 #ifdef  __cplusplus
184 }
______unchanged_portion_omitted_
```

```
**********************************************************
    1631 Thu May 16 17:36:20 2013
new/usr/src/uts/common/fs/zfs/sys/unique.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_UNIQUE_H
#define _SYS_UNIQUE_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/zfs_context.h>

#ifdef  __cplusplus
extern "C" {
#endif

/* The number of significant bits in each unique value. */
#define UNIQUE_BITS     56

void unique_init(void);
void unique_fini(void);

/*
 * Return a new unique value (which will not be uniquified against until
 * it is unique_insert()-ed).
 * it is unique_insert()-ed.
 */
uint64_t unique_create(void);

/* Return a unique value, which equals the one passed in if possible. */
uint64_t unique_insert(uint64_t value);

/* Indicate that this value no longer needs to be uniquified against. */
void unique_remove(uint64_t value);

#ifdef  __cplusplus
```



```
}
______unchanged_portion_omitted_
```

```
**********************************************************
   11454 Thu May 16 17:36:21 2013
new/usr/src/uts/common/fs/zfs/sys/vdev_impl.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 242 /*
 243  * vdev_dirty() flags
 244  */
 245 #define VDD_METASLAB    0x01
 246 #define VDD_DTL         0x02

 248 /* Offset of embedded boot loader region on each label */
 249 #define VDEV_BOOT_OFFSET        (2 * sizeof (vdev_label_t))
 250 #endif /* ! codereview */
 251 /*
 252  * Size of embedded boot loader region on each label.
 248  * Size and offset of embedded boot loader region on each label.
 253  * The total size of the first two labels plus the boot area is 4MB.
 254  */
 251 #define VDEV_BOOT_OFFSET        (2 * sizeof (vdev_label_t))
 255 #define VDEV_BOOT_SIZE          (7ULL << 19)                    /* 3.5M */

 257 /*
 258  * Size of label regions at the start and end of each leaf device.
 259  */
 260 #define VDEV_LABEL_START_SIZE   (2 * sizeof (vdev_label_t) + VDEV_BOOT_SIZE)
 261 #define VDEV_LABEL_END_SIZE     (2 * sizeof (vdev_label_t))
 262 #define VDEV_LABELS             4
 263 #define VDEV_BEST_LABEL         VDEV_LABELS

 265 #define VDEV_ALLOC_LOAD         0
 266 #define VDEV_ALLOC_ADD          1
 267 #define VDEV_ALLOC_SPARE        2
 268 #define VDEV_ALLOC_L2CACHE      3
 269 #define VDEV_ALLOC_ROOTPOOL     4
 270 #define VDEV_ALLOC_SPLIT        5
 271 #define VDEV_ALLOC_ATTACH       6

 273 /*
 274  * Allocate or free a vdev
 275  */
 276 extern vdev_t *vdev_alloc_common(spa_t *spa, uint_t id, uint64_t guid,
 277     vdev_ops_t *ops);
 278 extern int vdev_alloc(spa_t *spa, vdev_t **vdp, nvlist_t *config,
 279     vdev_t *parent, uint_t id, int alloctype);
 280 extern void vdev_free(vdev_t *vd);

 282 /*
 283  * Add or remove children and parents
 284  */
 285 extern void vdev_add_child(vdev_t *pvd, vdev_t *cvd);
 286 extern void vdev_remove_child(vdev_t *pvd, vdev_t *cvd);
 287 extern void vdev_compact_children(vdev_t *pvd);
 288 extern vdev_t *vdev_add_parent(vdev_t *cvd, vdev_ops_t *ops);
 289 extern void vdev_remove_parent(vdev_t *cvd);

 291 /*
 292  * vdev sync load and sync
 293  */
```



```
 294 extern void vdev_load_log_state(vdev_t *nvd, vdev_t *ovd);
 295 extern boolean_t vdev_log_state_valid(vdev_t *vd);
 296 extern void vdev_load(vdev_t *vd);
 297 extern void vdev_sync(vdev_t *vd, uint64_t txg);
 298 extern void vdev_sync_done(vdev_t *vd, uint64_t txg);
 299 extern void vdev_dirty(vdev_t *vd, int flags, void *arg, uint64_t txg);

 301 /*
 302  * Available vdev types.
 303  */
 304 extern vdev_ops_t vdev_root_ops;
 305 extern vdev_ops_t vdev_mirror_ops;
 306 extern vdev_ops_t vdev_replacing_ops;
 307 extern vdev_ops_t vdev_raidz_ops;
 308 extern vdev_ops_t vdev_disk_ops;
 309 extern vdev_ops_t vdev_file_ops;
 310 extern vdev_ops_t vdev_missing_ops;
 311 extern vdev_ops_t vdev_hole_ops;
 312 extern vdev_ops_t vdev_spare_ops;

 314 /*
 315  * Common size functions
 316  */
 317 extern uint64_t vdev_default_asize(vdev_t *vd, uint64_t psize);
 318 extern uint64_t vdev_get_min_asize(vdev_t *vd);
 319 extern void vdev_set_min_asize(vdev_t *vd);

 321 /*
 322  * Global variables
 319  * zdb uses this tunable, so it must be declared here to make lint happy.
 323  */
 324 /* zdb uses this tunable, so it must be declared here to make lint happy. */
 325 #endif /* ! codereview */
 326 extern int zfs_vdev_cache_size;

 328 /*
 329  * The vdev_buf_t is used to translate between zio_t and buf_t, and back again.
 330  */
 331 typedef struct vdev_buf {
 332         buf_t   vb_buf;         /* buffer that describes the io */
 333         zio_t   *vb_io;         /* pointer back to the original zio_t */
 334 } vdev_buf_t;

 336 #ifdef  __cplusplus
 337 }
 338 #endif

 340 #endif  /* _SYS_VDEV_IMPL_H */
```

```
**********************************************************
   17681 Thu May 16 17:36:21 2013
new/usr/src/uts/common/fs/zfs/sys/zap.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _SYS_ZAP_H
#define _SYS_ZAP_H

/*
 * ZAP - ZFS Attribute Processor
 *
 * The ZAP is a module which sits on top of the DMU (Data Management
 * Unit) and implements a higher-level storage primitive using DMU
 * objects.  Its primary consumer is the ZPL (ZFS Posix Layer).
 *
 * A "zapobj" is a DMU object which the ZAP uses to stores attributes.
 * Users should use only zap routines to access a zapobj - they should
 * not access the DMU object directly using DMU routines.
 *
 * The attributes stored in a zapobj are name-value pairs.  The name is
 * a zero-terminated string of up to ZAP_MAXNAMELEN bytes (including
 * terminating NULL).  The value is an array of integers, which may be
 * 1, 2, 4, or 8 bytes long.  The total space used by the array (number
 * of integers * integer length) can be up to ZAP_MAXVALUELEN bytes.
 * Note that an 8-byte integer value can be used to store the location
 * (object number) of another dmu object (which may be itself a zapobj).
 * Note that you can use a zero-length attribute to store a single bit
 * of information - the attribute is present or not.
 *
 * The ZAP routines are thread-safe.  However, you must observe the
 * DMU's restriction that a transaction may not be operated on
 * concurrently.
 *
 * Any of the routines that return an int may return an I/O error (EIO
 * or ECHECKSUM).
 *
```



```
 *
 * Implementation / Performance Notes:
 *
 * The ZAP is intended to operate most efficiently on attributes with
 * short (49 bytes or less) names and single 8-byte values, for which
 * the microzap will be used.  The ZAP should be efficient enough so
 * that the user does not need to cache these attributes.
 *
 * The ZAP's locking scheme makes its routines thread-safe.  Operations
 * on different zapobjs will be processed concurrently.  Operations on
 * the same zapobj which only read data will be processed concurrently.
 * Operations on the same zapobj which modify data will be processed
 * concurrently when there are many attributes in the zapobj (because
 * the ZAP uses per-block locking - more than 128 * (number of cpus)
 * small attributes will suffice).
 */

/*
 * We're using zero-terminated byte strings (ie. ASCII or UTF-8 C
 * strings) for the names of attributes, rather than a byte string
 * bounded by an explicit length.  If some day we want to support names
 * in character sets which have embedded zeros (eg. UTF-16, UTF-32),
 * we'll have to add routines for using length-bounded strings.
 */

#include <sys/dmu.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Specifies matching criteria for ZAP lookups.
 * The matchtype specifies which entry will be accessed.
 * MT_EXACT: only find an exact match (non-normalized)
 * MT_FIRST: find the "first" normalized (case and Unicode
 *     form) match; the designated "first" match will not change as long
 *     as the set of entries with this normalization doesn't change
 * MT_BEST: if there is an exact match, find that, otherwise find the
 *     first normalized match
 */
typedef enum matchtype
{
        /* Only find an exact match (non-normalized) */
#endif /* ! codereview */
        MT_EXACT,
        /*
         * If there is an exact match, find that, otherwise find the
         * first normalized match.
         */
#endif /* ! codereview */
        MT_BEST,
        /*
         * Find the "first" normalized (case and Unicode form) match;
         * the designated "first" match will not change as long as the
         * set of entries with this normalization doesn't change.
         */
#endif /* ! codereview */
        MT_FIRST
} matchtype_t;

typedef enum zap_flags {
        /* Use 64-bit hash value (serialized cursors will always use 64-bits) */
        ZAP_FLAG_HASH64 = 1 << 0,
        /* Key is binary, not string (zap_add_uint64() can be used) */
        ZAP_FLAG_UINT64_KEY = 1 << 1,
```

```
116         /*
117          * First word of key (which must be an array of uint64) is
118          * already randomly distributed.
119          */
120         ZAP_FLAG_PRE_HASHED_KEY = 1 << 2,
121 } zap_flags_t;

123 /*
124  * Create a new zapobj with no attributes and return its object number.
125  * MT_EXACT will cause the zap object to only support MT_EXACT lookups,
126  * otherwise any matchtype can be used for lookups.
127  *
128  * normflags specifies what normalization will be done.  values are:
129  * 0: no normalization (legacy on-disk format, supports MT_EXACT matching
130  *     only)
131  * U8_TEXTPREP_TOLOWER: case normalization will be performed.
132  *     MT_FIRST/MT_BEST matching will find entries that match without
133  *     regard to case (eg. looking for "foo" can find an entry "Foo").
134  * Eventually, other flags will permit unicode normalization as well.
135  */
136 uint64_t zap_create(objset_t *ds, dmu_object_type_t ot,
137     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
138 uint64_t zap_create_norm(objset_t *ds, int normflags, dmu_object_type_t ot,
139     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
140 uint64_t zap_create_flags(objset_t *os, int normflags, zap_flags_t flags,
141     dmu_object_type_t ot, int leaf_blockshift, int indirect_blockshift,
142     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
143 uint64_t zap_create_link(objset_t *os, dmu_object_type_t ot,
144     uint64_t parent_obj, const char *name, dmu_tx_t *tx);

146 /*
147  * Create a new zapobj with no attributes from the given (unallocated)
148  * object number.
149  */
150 int zap_create_claim(objset_t *ds, uint64_t obj, dmu_object_type_t ot,
151     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
152 int zap_create_claim_norm(objset_t *ds, uint64_t obj,
153     int normflags, dmu_object_type_t ot,
154     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);

156 /*
157  * The zapobj passed in must be a valid ZAP object for all of the
158  * following routines.
159  */

161 /*
162  * Destroy this zapobj and all its attributes.
163  *
164  * Frees the object number using dmu_object_free.
165  */
166 int zap_destroy(objset_t *ds, uint64_t zapobj, dmu_tx_t *tx);

168 /*
169  * Manipulate attributes.
170  *
171  * 'integer_size' is in bytes, and must be 1, 2, 4, or 8.
172  */

174 /*
175  * Retrieve the contents of the attribute with the given name.
176  *
177  * If the requested attribute does not exist, the call will fail and
178  * return ENOENT.
179  *
180  * If 'integer_size' is smaller than the attribute's integer size, the
181  * call will fail and return EINVAL.
```



```
182  *
183  * If 'integer_size' is equal to or larger than the attribute's integer
184  * size, the call will succeed and return 0.
185  *
186  * When converting to a larger integer size, the integers will be treated as
187  * unsigned (ie. no sign-extension will be performed).
 99  * size, the call will succeed and return 0.  * When converting to a
100  * larger integer size, the integers will be treated as unsigned (ie. no
101  * sign-extension will be performed).
188  *
189  * 'num_integers' is the length (in integers) of 'buf'.
190  *
191  * If the attribute is longer than the buffer, as many integers as will
192  * fit will be transferred to 'buf'.  If the entire attribute was not
193  * transferred, the call will return EOVERFLOW.
194  */
195 int zap_lookup(objset_t *ds, uint64_t zapobj, const char *name,
196     uint64_t integer_size, uint64_t num_integers, void *buf);

198 /*
108  *
199  * If rn_len is nonzero, realname will be set to the name of the found
200  * entry (which may be different from the requested name if matchtype is
201  * not MT_EXACT).
202  *
203  * If normalization_conflictp is not NULL, it will be set if there is
204  * another name with the same case/unicode normalized form.
205  */
116 int zap_lookup(objset_t *ds, uint64_t zapobj, const char *name,
117     uint64_t integer_size, uint64_t num_integers, void *buf);
206 int zap_lookup_norm(objset_t *ds, uint64_t zapobj, const char *name,
207     uint64_t integer_size, uint64_t num_integers, void *buf,
208     matchtype_t mt, char *realname, int rn_len,
209     boolean_t *normalization_conflictp);
210 int zap_lookup_uint64(objset_t *os, uint64_t zapobj, const uint64_t *key,
211     int key_numints, uint64_t integer_size, uint64_t num_integers, void *buf);
212 int zap_contains(objset_t *ds, uint64_t zapobj, const char *name);
213 int zap_prefetch_uint64(objset_t *os, uint64_t zapobj, const uint64_t *key,
214     int key_numints);

216 int zap_count_write(objset_t *os, uint64_t zapobj, const char *name,
217     int add, uint64_t *towrite, uint64_t *tooverwrite);

219 /*
220  * Create an attribute with the given name and value.
221  *
222  * If an attribute with the given name already exists, the call will
223  * fail and return EEXIST.
224  */
225 int zap_add(objset_t *ds, uint64_t zapobj, const char *key,
226     int integer_size, uint64_t num_integers,
227     const void *val, dmu_tx_t *tx);
228 int zap_add_uint64(objset_t *ds, uint64_t zapobj, const uint64_t *key,
229     int key_numints, int integer_size, uint64_t num_integers,
230     const void *val, dmu_tx_t *tx);

232 /*
233  * Set the attribute with the given name to the given value.  If an
234  * attribute with the given name does not exist, it will be created.  If
235  * an attribute with the given name already exists, the previous value
236  * will be overwritten.  The integer_size may be different from the
237  * existing attribute's integer size, in which case the attribute's
238  * integer size will be updated to the new value.
239  */
240 int zap_update(objset_t *ds, uint64_t zapobj, const char *name,
241     int integer_size, uint64_t num_integers, const void *val, dmu_tx_t *tx);
```

```
242 int zap_update_uint64(objset_t *os, uint64_t zapobj, const uint64_t *key,
243     int key_numints,
244     int integer_size, uint64_t num_integers, const void *val, dmu_tx_t *tx);

246 /*
247  * Get the length (in integers) and the integer size of the specified
248  * attribute.
249  *
250  * If the requested attribute does not exist, the call will fail and
251  * return ENOENT.
252  */
253 int zap_length(objset_t *ds, uint64_t zapobj, const char *name,
254     uint64_t *integer_size, uint64_t *num_integers);
255 int zap_length_uint64(objset_t *os, uint64_t zapobj, const uint64_t *key,
256     int key_numints, uint64_t *integer_size, uint64_t *num_integers);

258 /*
259  * Remove the specified attribute.
260  *
261  * If the specified attribute does not exist, the call will fail and
262  * return ENOENT.
263  */
264 int zap_remove(objset_t *ds, uint64_t zapobj, const char *name, dmu_tx_t *tx);
265 int zap_remove_norm(objset_t *ds, uint64_t zapobj, const char *name,
266     matchtype_t mt, dmu_tx_t *tx);
267 int zap_remove_uint64(objset_t *os, uint64_t zapobj, const uint64_t *key,
268     int key_numints, dmu_tx_t *tx);

270 /*
271  * Returns (in *count) the number of attributes in the specified zap
272  * object.
273  */
274 int zap_count(objset_t *ds, uint64_t zapobj, uint64_t *count);

276 /*
277  * Returns (in name) the name of the entry whose (value & mask)
278  * (za_first_integer) is value, or ENOENT if not found.  The string
279  * pointed to by name must be at least 256 bytes long.  If mask==0, the
280  * match must be exact (ie, same as mask=-1ULL).
281  */
282 int zap_value_search(objset_t *os, uint64_t zapobj,
283     uint64_t value, uint64_t mask, char *name);

285 /*
286  * Transfer all the entries from fromobj into intoobj.  Only works on
287  * int_size=8 num_integers=1 values.  Fails if there are any duplicated
288  * entries.
289  */
290 int zap_join(objset_t *os, uint64_t fromobj, uint64_t intoobj, dmu_tx_t *tx);

292 /* Same as zap_join, but set the values to 'value'. */
293 int zap_join_key(objset_t *os, uint64_t fromobj, uint64_t intoobj,
294     uint64_t value, dmu_tx_t *tx);

296 /* Same as zap_join, but add together any duplicated entries. */
297 int zap_join_increment(objset_t *os, uint64_t fromobj, uint64_t intoobj,
298     dmu_tx_t *tx);

300 /*
301  * Manipulate entries where the name + value are the "same" (the name is
302  * a stringified version of the value).
303  */
304 int zap_add_int(objset_t *os, uint64_t obj, uint64_t value, dmu_tx_t *tx);
305 int zap_remove_int(objset_t *os, uint64_t obj, uint64_t value, dmu_tx_t *tx);
306 int zap_lookup_int(objset_t *os, uint64_t obj, uint64_t value);
307 int zap_increment_int(objset_t *os, uint64_t obj, uint64_t key, int64_t delta,
```



```
308     dmu_tx_t *tx);

310 /* Here the key is an int and the value is a different int. */
311 int zap_add_int_key(objset_t *os, uint64_t obj,
312     uint64_t key, uint64_t value, dmu_tx_t *tx);
313 int zap_update_int_key(objset_t *os, uint64_t obj,
314     uint64_t key, uint64_t value, dmu_tx_t *tx);
315 int zap_lookup_int_key(objset_t *os, uint64_t obj,
316     uint64_t key, uint64_t *valuep);

318 int zap_increment(objset_t *os, uint64_t obj, const char *name, int64_t delta,
319     dmu_tx_t *tx);

321 struct zap;
322 struct zap_leaf;
323 typedef struct zap_cursor {
324         /* This structure is opaque! */
325         objset_t *zc_objset;
326         struct zap *zc_zap;
327         struct zap_leaf *zc_leaf;
328         uint64_t zc_zapobj;
329         uint64_t zc_serialized;
330         uint64_t zc_hash;
331         uint32_t zc_cd;
332 } zap_cursor_t;
```

unchanged_portion_omitted_

```
**********************************************************
    7553 Thu May 16 17:36:21 2013
new/usr/src/uts/common/fs/zfs/sys/zap_leaf.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

  96 #define ZLF_ENTRIES_CDSORTED (1<<0)

  98 /*
  99  * TAKE NOTE:
 100  * If zap_leaf_phys_t is modified, zap_leaf_byteswap() must be modified.
 101  */
 102 typedef struct zap_leaf_phys {
 103         struct zap_leaf_header {
 104                 /* Public to ZAP */
 105 #endif /* ! codereview */
 106                 uint64_t lh_block_type;         /* ZBT_LEAF */
 107                 uint64_t lh_pad1;
 108                 uint64_t lh_prefix;             /* hash prefix of this leaf */
 109                 uint32_t lh_magic;              /* ZAP_LEAF_MAGIC */
 110                 uint16_t lh_nfree;              /* number free chunks */
 111                 uint16_t lh_nentries;           /* number of entries */
 112                 uint16_t lh_prefix_len;         /* num bits used to id this */

 114                 /* Private to zap_leaf */
 104 /* above is accessable to zap, below is zap_leaf private */

 115                 uint16_t lh_freelist;           /* chunk head of free list */
 116                 uint8_t lh_flags;               /* ZLF_* flags */
 117                 uint8_t lh_pad2[11];
 118         } l_hdr; /* 2 24-byte chunks */

 120         /*
 121          * The header is followed by a hash table with
 122          * ZAP_LEAF_HASH_NUMENTRIES(zap) entries.  The hash table is
 123          * followed by an array of ZAP_LEAF_NUMCHUNKS(zap)
 124          * zap_leaf_chunk structures.  These structures are accessed
 125          * with the ZAP_LEAF_CHUNK() macro.
 126          */

 128         uint16_t l_hash[1];
 129 } zap_leaf_phys_t;
______unchanged_portion_omitted_


 164 typedef struct zap_entry_handle {
 165         /* Set by zap_leaf and public to ZAP */
 156         /* below is set by zap_leaf.c and is public to zap.c */
 166         uint64_t zeh_num_integers;
 167         uint64_t zeh_hash;
 168         uint32_t zeh_cd;
 169         uint8_t zeh_integer_size;

 171         /* Private to zap_leaf */
 162         /* below is private to zap_leaf.c */
 172         uint16_t zeh_fakechunk;
 173         uint16_t *zeh_chunkp;
 174         zap_leaf_t *zeh_leaf;
 175 } zap_entry_handle_t;
```



```
 177 /*
 178  * Return a handle to the named entry, or ENOENT if not found.  The hash
 179  * value must equal zap_hash(name).
 180  */
 181 extern int zap_leaf_lookup(zap_leaf_t *l,
 182     struct zap_name *zn, zap_entry_handle_t *zeh);

 184 /*
 185  * Return a handle to the entry with this hash+cd, or the entry with the
 186  * next closest hash+cd.
 187  */
 188 extern int zap_leaf_lookup_closest(zap_leaf_t *l,
 189     uint64_t hash, uint32_t cd, zap_entry_handle_t *zeh);

 191 /*
 192  * Read the first num_integers in the attribute.  Integer size
 193  * conversion will be done without sign extension.  Return EINVAL if
 194  * integer_size is too small.  Return EOVERFLOW if there are more than
 195  * num_integers in the attribute.
 196  */
 197 extern int zap_entry_read(const zap_entry_handle_t *zeh,
 198     uint8_t integer_size, uint64_t num_integers, void *buf);

 200 extern int zap_entry_read_name(struct zap *zap, const zap_entry_handle_t *zeh,
 201     uint16_t buflen, char *buf);

 203 /*
 204  * Replace the value of an existing entry.
 205  *
 206  * May fail if it runs out of space (ENOSPC).
 197  * zap_entry_update may fail if it runs out of space (ENOSPC).
 207  */
 208 extern int zap_entry_update(zap_entry_handle_t *zeh,
 209     uint8_t integer_size, uint64_t num_integers, const void *buf);

 211 /*
 212  * Remove an entry.
 213  */
 214 extern void zap_entry_remove(zap_entry_handle_t *zeh);

 216 /*
 217  * Create an entry. An equal entry must not exist, and this entry must
 218  * belong in this leaf (according to its hash value).  Fills in the
 219  * entry handle on success.  Returns 0 on success or ENOSPC on failure.
 220  */
 221 extern int zap_entry_create(zap_leaf_t *l, struct zap_name *zn, uint32_t cd,
 222     uint8_t integer_size, uint64_t num_integers, const void *buf,
 223     zap_entry_handle_t *zeh);

 225 /* Determine whether there is another entry with the same normalized form. */
 216 /*
 217  * Return true if there are additional entries with the same normalized
 218  * form.
 219  */
 226 extern boolean_t zap_entry_normalization_conflict(zap_entry_handle_t *zeh,
 227     struct zap_name *zn, const char *name, struct zap *zap);

 229 /*
 230  * Other stuff.
 231  */

 233 extern void zap_leaf_init(zap_leaf_t *l, boolean_t sort);
 234 extern void zap_leaf_byteswap(zap_leaf_phys_t *buf, int len);
 235 extern void zap_leaf_split(zap_leaf_t *l, zap_leaf_t *nl, boolean_t sort);
 236 extern void zap_leaf_stats(struct zap *zap, zap_leaf_t *l,
 237     struct zap_stats *zs);
```

```
 239 #ifdef  __cplusplus
 240 }
______unchanged_portion_omitted_
```

```
**********************************************************
    8160 Thu May 16 17:36:21 2013
new/usr/src/uts/common/fs/zfs/sys/zfs_acl.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#ifndef _SYS_FS_ZFS_ACL_H
#define _SYS_FS_ZFS_ACL_H

#ifdef _KERNEL
#include <sys/isa_defs.h>
#include <sys/types32.h>
#endif
#include <sys/acl.h>
#include <sys/dmu.h>
#include <sys/zfs_fuid.h>
#include <sys/sa.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct znode_phys;

#define ACE_SLOT_CNT    6
#define ZFS_ACL_VERSION_INITIAL 0ULL
#define ZFS_ACL_VERSION_FUID    1ULL
#define ZFS_ACL_VERSION         ZFS_ACL_VERSION_FUID

/*
 * ZFS ACLs (Access Control Lists) are stored in various forms.
 *
 * ZFS ACLs are store in various forms.
 * Files created with ACL version ZFS_ACL_VERSION_INITIAL
 * will all be created with fixed length ACEs of type
 * zfs_oldace_t.
 *
 * Files with ACL version ZFS_ACL_VERSION_FUID will be created
```



```
 * with various sized ACEs.  The abstraction entries will utilize
 * zfs_ace_hdr_t, normal user/group entries will use zfs_ace_t
 * and some specialized CIFS ACEs will use zfs_object_ace_t.
 */

/*
 * All ACEs have a common hdr.  For
 * owner@, group@, and everyone@ this is all
 * thats needed.
 */
typedef struct zfs_ace_hdr {
        uint16_t z_type;
        uint16_t z_flags;
        uint32_t z_access_mask;
} zfs_ace_hdr_t;
______unchanged_portion_omitted_

typedef struct acl_ops {
        uint32_t        (*ace_mask_get) (void *acep); /* get  access mask */
        void            (*ace_mask_set) (void *acep,
                            uint32_t mask); /* set access mask */
        uint16_t        (*ace_flags_get) (void *acep);  /* get flags */
        void            (*ace_flags_set) (void *acep,
                            uint16_t flags); /* set flags */
        uint16_t        (*ace_type_get)(void *acep); /* get type */
        void            (*ace_type_set)(void *acep,
                            uint16_t type); /* set type */
        uint64_t        (*ace_who_get)(void *acep); /* get who/fuid */
        void            (*ace_who_set)(void *acep,
                            uint64_t who); /* set who/fuid */
        size_t          (*ace_size)(void *acep); /* how big is this ace */
        size_t          (*ace_abstract_size)(void); /* sizeof abstract entry */
        int             (*ace_mask_off)(void); /* off of access mask in ace */
        /* ptr to data if any */
#endif /* ! codereview */
        int             (*ace_data)(void *acep, void **datap);
                            /* ptr to data if any */
} acl_ops_t;
______unchanged_portion_omitted_
```

****
2620 Thu May 16 17:36:22 2013
new/usr/src/uts/common/fs/zfs/sys/zfs_rlock.h
3742 zfs comments need cleaner, more consistent style
Submitted by: Will Andrews <willa@spectralogic.com>
Submitted by: Alan Somers <alans@spectralogic.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Eric Schrock <eric.schrock@delphix.com>
****
_____***unchanged_portion_omitted_***

```
 59 /*
 60  * Lock a range (offset, length) as either shared (RL_READER)
 61  * or exclusive (RL_WRITER or RL_APPEND).  RL_APPEND is a special type that
 62  * is converted to RL_WRITER that specified to lock from the start of the
 63  * end of file.  Returns the range lock structure.
 60  * Lock a range (offset, length) as either shared (READER)
 61  * or exclusive (WRITER or APPEND). APPEND is a special type that
 62  * is converted to WRITER that specified to lock from the start of the
 63  * end of file.  zfs_range_lock() returns the range lock structure.
 64  */
 65 rl_t *zfs_range_lock(znode_t *zp, uint64_t off, uint64_t len, rl_type_t type);

 67 /* Unlock range and destroy range lock structure. */
 67 /*
 68  * Unlock range and destroy range lock structure.
 69  */
 68 void zfs_range_unlock(rl_t *rl);

 70 /*
 71  * Reduce range locked as RW_WRITER from whole file to specified range.
 72  * Asserts the whole file was previously locked.
 73  */
 74 void zfs_range_reduce(rl_t *rl, uint64_t off, uint64_t len);

 76 /*
 77  * AVL comparison function used to order range locks
 78  * Locks are ordered on the start offset of the range.
 79  * AVL comparison function used to compare range locks
 79  */
 80 int zfs_range_compare(const void *arg1, const void *arg2);

 82 #endif /* _KERNEL */

 84 #ifdef  __cplusplus
 85 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   13128 Thu May 16 17:36:22 2013
new/usr/src/uts/common/fs/zfs/sys/zfs_znode.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

  77 /*
  78  * Define special zfs pflags
  79  */
  80 #define ZFS_XATTR               0x1             /* is an extended attribute */
  81 #define ZFS_INHERIT_ACE         0x2             /* ace has inheritable ACEs */
  82 #define ZFS_ACL_TRIVIAL         0x4             /* files ACL is trivial */
  83 #define ZFS_ACL_OBJ_ACE         0x8             /* ACL has CMPLX Object ACE */
  84 #define ZFS_ACL_PROTECTED       0x10            /* ACL protected */
  85 #define ZFS_ACL_DEFAULTED       0x20            /* ACL should be defaulted */
  86 #define ZFS_ACL_AUTO_INHERIT    0x40            /* ACL should be inherited */
  87 #define ZFS_BONUS_SCANSTAMP     0x80            /* Scanstamp in bonus area */
  88 #define ZFS_NO_EXECS_DENIED     0x100           /* exec was given to everyone */

  90 #define SA_ZPL_ATIME(z)         z->z_attr_table[ZPL_ATIME]
  91 #define SA_ZPL_MTIME(z)         z->z_attr_table[ZPL_MTIME]
  92 #define SA_ZPL_CTIME(z)         z->z_attr_table[ZPL_CTIME]
  93 #define SA_ZPL_CRTIME(z)        z->z_attr_table[ZPL_CRTIME]
  94 #define SA_ZPL_GEN(z)           z->z_attr_table[ZPL_GEN]
  95 #define SA_ZPL_DACL_ACES(z)     z->z_attr_table[ZPL_DACL_ACES]
  96 #define SA_ZPL_XATTR(z)         z->z_attr_table[ZPL_XATTR]
  97 #define SA_ZPL_SYMLINK(z)       z->z_attr_table[ZPL_SYMLINK]
  98 #define SA_ZPL_RDEV(z)          z->z_attr_table[ZPL_RDEV]
  99 #define SA_ZPL_SCANSTAMP(z)     z->z_attr_table[ZPL_SCANSTAMP]
 100 #define SA_ZPL_UID(z)           z->z_attr_table[ZPL_UID]
 101 #define SA_ZPL_GID(z)           z->z_attr_table[ZPL_GID]
 102 #define SA_ZPL_PARENT(z)        z->z_attr_table[ZPL_PARENT]
 103 #define SA_ZPL_LINKS(z)         z->z_attr_table[ZPL_LINKS]
 104 #define SA_ZPL_MODE(z)          z->z_attr_table[ZPL_MODE]
 105 #define SA_ZPL_DACL_COUNT(z)    z->z_attr_table[ZPL_DACL_COUNT]
 106 #define SA_ZPL_FLAGS(z)         z->z_attr_table[ZPL_FLAGS]
 107 #define SA_ZPL_SIZE(z)          z->z_attr_table[ZPL_SIZE]
 108 #define SA_ZPL_ZNODE_ACL(z)     z->z_attr_table[ZPL_ZNODE_ACL]
 109 #define SA_ZPL_PAD(z)           z->z_attr_table[ZPL_PAD]

 111 /*
 112  * Is ID ephemeral?
 113  */
 114 #define IS_EPHEMERAL(x)         (x > MAXUID)

 116 /*
 117  * Should we use FUIDs?
 118  */
 119 #define USE_FUIDS(version, os)  (version >= ZPL_VERSION_FUID && \
 120     spa_version(dmu_objset_spa(os)) >= SPA_VERSION_FUID)
 121 #define USE_SA(version, os) (version >= ZPL_VERSION_SA && \
 122     spa_version(dmu_objset_spa(os)) >= SPA_VERSION_SA)

 124 #define MASTER_NODE_OBJ 1

 126 /*
 127  * Special attributes for master node.
 128  * "userquota@" and "groupquota@" are also valid (from
 129  * zfs_userquota_prop_prefixes[]).
 130  */
```



```
 131 #define ZFS_FSID                "FSID"
 132 #define ZFS_UNLINKED_SET        "DELETE_QUEUE"
 133 #define ZFS_ROOT_OBJ            "ROOT"
 134 #define ZPL_VERSION_STR         "VERSION"
 135 #define ZFS_FUID_TABLES         "FUID"
 136 #define ZFS_SHARES_DIR          "SHARES"
 137 #define ZFS_SA_ATTRS            "SA_ATTRS"

 139 #define ZFS_MAX_BLOCKSIZE       (SPA_MAXBLOCKSIZE)

 141 /* Path component length */
 141 /*
 142  * Path component length
 143  *
 144 #endif /* ! codereview */
 145  * The generic fs code uses MAXNAMELEN to represent
 146  * what the largest component length is.  Unfortunately,
 147  * this length includes the terminating NULL.  ZFS needs
 148  * to tell the users via pathconf() and statvfs() what the
 149  * true maximum length of a component is, excluding the NULL.
 150  */
 151 #define ZFS_MAXNAMELEN  (MAXNAMELEN - 1)

 153 /*
 154  * Convert mode bits (zp_mode) to BSD-style DT_* values for storing in
 155  * the directory entries.
 156  */
 157 #define IFTODT(mode) (((mode) & S_IFMT) >> 12)

 159 /*
 160  * The directory entry has the type (currently unused on Solaris) in the
 161  * top 4 bits, and the object number in the low 48 bits.  The "middle"
 162  * 12 bits are unused.
 163  */
 164 #define ZFS_DIRENT_TYPE(de) BF64_GET(de, 60, 4)
 165 #define ZFS_DIRENT_OBJ(de) BF64_GET(de, 0, 48)

 167 /*
 168  * Directory entry locks control access to directory entries.
 169  * They are used to protect creates, deletes, and renames.
 170  * Each directory znode has a mutex and a list of locked names.
 171  */
 172 #ifdef _KERNEL
 173 typedef struct zfs_dirlock {
 174         char            *dl_name;       /* directory entry being locked */
 175         uint32_t        dl_sharecnt;    /* 0 if exclusive, > 0 if shared */
 176         uint8_t         dl_namelock;    /* 1 if z_name_lock is NOT held */
 177         uint16_t        dl_namesize;    /* set if dl_name was allocated */
 178         kcondvar_t      dl_cv;          /* wait for entry to be unlocked */
 179         struct znode    *dl_dzp;        /* directory znode */
 180         struct zfs_dirlock *dl_next;    /* next in z_dirlocks list */
 181 } zfs_dirlock_t;

 183 typedef struct znode {
 184         struct zfsvfs   *z_zfsvfs;
 185         vnode_t         *z_vnode;
 186         uint64_t        z_id;           /* object ID for this znode */
 187         kmutex_t        z_lock;         /* znode modification lock */
 188         krwlock_t       z_parent_lock;  /* parent lock for directories */
 189         krwlock_t       z_name_lock;    /* "master" lock for dirent locks */
 190         zfs_dirlock_t   *z_dirlocks;    /* directory entry lock list */
 191         kmutex_t        z_range_lock;   /* protects changes to z_range_avl */
 192         avl_tree_t      z_range_avl;    /* avl tree of file range locks */
 193         uint8_t         z_unlinked;     /* file has been unlinked */
 194         uint8_t         z_atime_dirty;  /* atime needs to be synced */
 195         uint8_t         z_zn_prefetch;  /* Prefetch znodes? */
```

```
196         uint8_t         z_moved;        /* Has this znode been moved? */
197         uint_t          z_blksz;        /* block size in bytes */
198         uint_t          z_seq;          /* modification sequence number */
199         uint64_t        z_mapcnt;       /* number of pages mapped to file */
200         uint64_t        z_gen;          /* generation (cached) */
201         uint64_t        z_size;         /* file size (cached) */
202         uint64_t        z_atime[2];     /* atime (cached) */
203         uint64_t        z_links;        /* file links (cached) */
204         uint64_t        z_pflags;       /* pflags (cached) */
205         uint64_t        z_uid;          /* uid fuid (cached) */
206         uint64_t        z_gid;          /* gid fuid (cached) */
207         mode_t          z_mode;         /* mode (cached) */
208         uint32_t        z_sync_cnt;     /* synchronous open count */
209         kmutex_t        z_acl_lock;     /* acl data lock */
210         zfs_acl_t       *z_acl_cached;  /* cached acl */
211         list_node_t     z_link_node;    /* all znodes in fs link */
212         sa_handle_t     *z_sa_hdl;      /* handle to sa data */
213         boolean_t       z_is_sa;        /* are we native sa? */
214 } znode_t;


217 /*
218  * Range locking rules
219  * --------------------
220  * 1. When truncating a file (zfs_create, zfs_setattr, zfs_space) the whole
221  *    file range needs to be locked as RL_WRITER. Only then can the pages be
222  *    freed etc and zp_size reset. zp_size must be set within range lock.
223  * 2. For writes and punching holes (zfs_write & zfs_space) just the range
224  *    being written or freed needs to be locked as RL_WRITER.
225  *    Multiple writes at the end of the file must coordinate zp_size updates
226  *    to ensure data isn't lost. A compare and swap loop is currently used
227  *    to ensure the file size is at least the offset last written.
228  * 3. For reads (zfs_read, zfs_get_data & zfs_putapage) just the range being
229  *    read needs to be locked as RL_READER. A check against zp_size can then
230  *    be made for reading beyond end of file.
231  */

233 /*
234  * Convert between znode pointers and vnode pointers
235  */
236 #define ZTOV(ZP)        ((ZP)->z_vnode)
237 #define VTOZ(VP)        ((znode_t *)(VP)->v_data)

239 /* Called on entry to each ZFS vnode and vfs operation  */
143 /*
144  * ZFS_ENTER() is called on entry to each ZFS vnode and vfs operation.
145  * ZFS_EXIT() must be called before exitting the vop.
146  * ZFS_VERIFY_ZP() verifies the znode is valid.
147  */
240 #define ZFS_ENTER(zfsvfs) \
241         { \
242                 rrw_enter_read(&(zfsvfs)->z_teardown_lock, FTAG); \
243                 if ((zfsvfs)->z_unmounted) { \
244                         ZFS_EXIT(zfsvfs); \
245                         return (EIO); \
246                 } \
247         }

249 /* Must be called before exiting the vop */
250 #endif /* ! codereview */
251 #define ZFS_EXIT(zfsvfs) rrw_exit(&(zfsvfs)->z_teardown_lock, FTAG)

253 /* Verifies the znode is valid */
254 #endif /* ! codereview */
255 #define ZFS_VERIFY_ZP(zp) \
256         if ((zp)->z_sa_hdl == NULL) { \
```



```
257                 ZFS_EXIT((zp)->z_zfsvfs); \
258                 return (EIO); \
259         } \

261 /*
262  * Macros for dealing with dmu_buf_hold
263  */
264 #define ZFS_OBJ_HASH(obj_num)   ((obj_num) & (ZFS_OBJ_MTX_SZ - 1))
265 #define ZFS_OBJ_MUTEX(zfsvfs, obj_num)  \
266         (&(zfsvfs)->z_hold_mtx[ZFS_OBJ_HASH(obj_num)])
267 #define ZFS_OBJ_HOLD_ENTER(zfsvfs, obj_num) \
268         mutex_enter(ZFS_OBJ_MUTEX((zfsvfs), (obj_num)))
269 #define ZFS_OBJ_HOLD_TRYENTER(zfsvfs, obj_num) \
270         mutex_tryenter(ZFS_OBJ_MUTEX((zfsvfs), (obj_num)))
271 #define ZFS_OBJ_HOLD_EXIT(zfsvfs, obj_num) \
272         mutex_exit(ZFS_OBJ_MUTEX((zfsvfs), (obj_num)))

274 /* Encode ZFS stored time values from a struct timespec */
157 /*
158  * Macros to encode/decode ZFS stored time values from/to struct timespec
159  */
275 #define ZFS_TIME_ENCODE(tp, stmp)               \
276 {                                               \
277         (stmp)[0] = (uint64_t)(tp)->tv_sec;     \
278         (stmp)[1] = (uint64_t)(tp)->tv_nsec;    \
279 }

281 /* Decode ZFS stored time values to a struct timespec */
282 #endif /* ! codereview */
283 #define ZFS_TIME_DECODE(tp, stmp)               \
284 {                                               \
285         (tp)->tv_sec = (time_t)(stmp)[0];               \
286         (tp)->tv_nsec = (long)(stmp)[1];                \
287 }

289 /*
290  * Timestamp defines
291  */
292 #define ACCESSED                (AT_ATIME)
293 #define STATE_CHANGED           (AT_CTIME)
294 #define CONTENT_MODIFIED        (AT_MTIME | AT_CTIME)

296 #define ZFS_ACCESSTIME_STAMP(zfsvfs, zp) \
297         if ((zfsvfs)->z_atime && !((zfsvfs)->z_vfs->vfs_flag & VFS_RDONLY)) \
298                 zfs_tstamp_update_setup(zp, ACCESSED, NULL, NULL, B_FALSE);

300 extern int      zfs_init_fs(zfsvfs_t *, znode_t **);
301 extern void     zfs_set_dataprop(objset_t *);
302 extern void     zfs_create_fs(objset_t *os, cred_t *cr, nvlist_t *,
303     dmu_tx_t *tx);
304 extern void     zfs_tstamp_update_setup(znode_t *, uint_t, uint64_t [2],
305     uint64_t [2], boolean_t);
306 extern void     zfs_grow_blocksize(znode_t *, uint64_t, dmu_tx_t *);
307 extern int      zfs_freesp(znode_t *, uint64_t, uint64_t, int, boolean_t);
308 extern void     zfs_znode_init(void);
309 extern void     zfs_znode_fini(void);
310 extern int      zfs_zget(zfsvfs_t *, uint64_t, znode_t **);
311 extern int      zfs_rezget(znode_t *);
312 extern void     zfs_zinactive(znode_t *);
313 extern void     zfs_znode_delete(znode_t *, dmu_tx_t *);
314 extern void     zfs_znode_free(znode_t *);
315 extern void     zfs_remove_op_tables();
316 extern int      zfs_create_op_tables();
317 extern int      zfs_sync(vfs_t *vfsp, short flag, cred_t *cr);
318 extern dev_t    zfs_cmpldev(uint64_t);
319 extern int      zfs_get_zplprop(objset_t *os, zfs_prop_t prop, uint64_t *value);
```

```
320 extern int      zfs_get_stats(objset_t *os, nvlist_t *nv);
321 extern void     zfs_znode_dmu_fini(znode_t *);

323 extern void zfs_log_create(zilog_t *zilog, dmu_tx_t *tx, uint64_t txtype,
324     znode_t *dzp, znode_t *zp, char *name, vsecattr_t *, zfs_fuid_info_t *,
325     vattr_t *vap);
326 extern int zfs_log_create_txtype(zil_create_t, vsecattr_t *vsecp,
327     vattr_t *vap);
328 extern void zfs_log_remove(zilog_t *zilog, dmu_tx_t *tx, uint64_t txtype,
329     znode_t *dzp, char *name, uint64_t foid);
330 #define ZFS_NO_OBJECT   0       /* no object id */
331 extern void zfs_log_link(zilog_t *zilog, dmu_tx_t *tx, uint64_t txtype,
332     znode_t *dzp, znode_t *zp, char *name);
333 extern void zfs_log_symlink(zilog_t *zilog, dmu_tx_t *tx, uint64_t txtype,
334     znode_t *dzp, znode_t *zp, char *name, char *link);
335 extern void zfs_log_rename(zilog_t *zilog, dmu_tx_t *tx, uint64_t txtype,
336     znode_t *sdzp, char *sname, znode_t *tdzp, char *dname, znode_t *szp);
337 extern void zfs_log_write(zilog_t *zilog, dmu_tx_t *tx, int txtype,
338     znode_t *zp, offset_t off, ssize_t len, int ioflag);
339 extern void zfs_log_truncate(zilog_t *zilog, dmu_tx_t *tx, int txtype,
340     znode_t *zp, uint64_t off, uint64_t len);
341 extern void zfs_log_setattr(zilog_t *zilog, dmu_tx_t *tx, int txtype,
342     znode_t *zp, vattr_t *vap, uint_t mask_applied, zfs_fuid_info_t *fuidp);
343 extern void zfs_log_acl(zilog_t *zilog, dmu_tx_t *tx, znode_t *zp,
344     vsecattr_t *vsecp, zfs_fuid_info_t *fuidp);
345 extern void zfs_xvattr_set(znode_t *zp, xvattr_t *xvap, dmu_tx_t *tx);
346 extern void zfs_upgrade(zfsvfs_t *zfsvfs, dmu_tx_t *tx);
347 extern int zfs_create_share_dir(zfsvfs_t *zfsvfs, dmu_tx_t *tx);

349 extern caddr_t zfs_map_page(page_t *, enum seg_rw);
350 extern void zfs_unmap_page(page_t *, caddr_t);

352 extern zil_get_data_t zfs_get_data;
353 extern zil_replay_func_t *zfs_replay_vector[TX_MAX_TYPE];
354 extern int zfsfstype;

356 #endif /* _KERNEL */

358 extern int zfs_obj_to_path(objset_t *osp, uint64_t obj, char *buf, int len);

360 #ifdef  __cplusplus
361 }
362 #endif

364 #endif  /* _SYS_FS_ZFS_ZNODE_H */
```

```
**********************************************************
   15273 Thu May 16 17:36:22 2013
new/usr/src/uts/common/fs/zfs/sys/zil.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 229 /*
 230  * FUID ACL record will be an array of ACEs from the original ACL.
 231  * If this array includes ephemeral IDs, the record will also include
 232  * an array of log-specific FUIDs to replace the ephemeral IDs.
 233  * Only one copy of each unique domain will be present, so the log-specific
 234  * FUIDs will use an index into a compressed domain table.  On replay this
 235  * information will be used to construct real FUIDs (and bypass idmap,
 236  * since it may not be available).
 237  */

 239 /*
 240  * Log record for creates with optional ACL
 241  * This log record is also used for recording any FUID
 242  * information needed for replaying the create.  If the
 243  * file doesn't have any actual ACEs then the lr_aclcnt
 244  * would be zero.
 245  *
 246  * After lr_acl_flags, there are a lr_acl_bytes number of variable sized ace's.
 247  * If create is also setting xvattr's, then acl data follows xvattr.
 248  * If ACE FUIDs are needed then they will follow the xvattr_t.  Following
 249  * the FUIDs will be the domain table information.  The FUIDs for the owner
 250  * and group will be in lr_create.  Name follows ACL data.
 251 #endif /* ! codereview */
 252  */
 253 typedef struct {
 254         lr_create_t     lr_create;      /* common create portion */
 255         uint64_t        lr_aclcnt;      /* number of ACEs in ACL */
 256         uint64_t        lr_domcnt;      /* number of unique domains */
 257         uint64_t        lr_fuidcnt;     /* number of real fuids */
 258         uint64_t        lr_acl_bytes;   /* number of bytes in ACL */
 259         uint64_t        lr_acl_flags;   /* ACL flags */
 245         /* lr_acl_bytes number of variable sized ace's follows */
 246         /* if create is also setting xvattr's, then acl data follows xvattr */
 247         /* if ACE FUIDs are needed then they will follow the xvattr_t */
 248         /* Following the FUIDs will be the domain table information. */
 249         /* The FUIDs for the owner and group will be in the lr_create */
 250         /* portion of the record. */
 251         /* name follows ACL data */
 260 } lr_acl_create_t;
______unchanged_portion_omitted_
```

```
**********************************************************
    2905 Thu May 16 17:36:23 2013
new/usr/src/uts/common/fs/zfs/sys/zio_compress.h
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 */

#ifndef _SYS_ZIO_COMPRESS_H
#define _SYS_ZIO_COMPRESS_H

#include <sys/zio.h>

#ifdef  __cplusplus
extern "C" {
#endif

/* Common signature for all zio compress functions. */
/*
 * Common signature for all zio compress/decompress functions.
 */
typedef size_t zio_compress_func_t(void *src, void *dst,
    size_t s_len, size_t d_len, int);
/* Common signature for all zio decompress functions. */
#endif /* ! codereview */
typedef int zio_decompress_func_t(void *src, void *dst,
    size_t s_len, size_t d_len, int);

/*
 * Information about each compression function.
 */
typedef struct zio_compress_info {
        zio_compress_func_t     *ci_compress;   /* compression function */
        zio_decompress_func_t   *ci_decompress; /* decompression function */
        int                     ci_level;       /* level parameter */
```



```
        char                    *ci_name;       /* algorithm name */
} zio_compress_info_t;

extern zio_compress_info_t zio_compress_table[ZIO_COMPRESS_FUNCTIONS];

/*
 * Compression routines.
 */
extern size_t lzjb_compress(void *src, void *dst, size_t s_len, size_t d_len,
    int level);
extern int lzjb_decompress(void *src, void *dst, size_t s_len, size_t d_len,
    int level);
extern size_t gzip_compress(void *src, void *dst, size_t s_len, size_t d_len,
    int level);
extern int gzip_decompress(void *src, void *dst, size_t s_len, size_t d_len,
    int level);
extern size_t zle_compress(void *src, void *dst, size_t s_len, size_t d_len,
    int level);
extern int zle_decompress(void *src, void *dst, size_t s_len, size_t d_len,
    int level);
extern size_t lz4_compress(void *src, void *dst, size_t s_len, size_t d_len,
    int level);
extern int lz4_decompress(void *src, void *dst, size_t s_len, size_t d_len,
    int level);

/*
 * Compress and decompress data if necessary.
 */
extern size_t zio_compress_data(enum zio_compress c, void *src, void *dst,
    size_t s_len);
extern int zio_decompress_data(enum zio_compress c, void *src, void *dst,
    size_t s_len, size_t d_len);

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_ZIO_COMPRESS_H */
```

```
**********************************************************
   21899 Thu May 16 17:36:23 2013
new/usr/src/uts/common/fs/zfs/txg.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 562 /*
 563  * Delay this thread by delay nanoseconds if we are still in the open
 564  * transaction group and there is already a waiting txg quiescing or quiesced.
 565  * Abort the delay if this txg stalls or enters the quiescing state.
 564  * transaction group and there is already a waiting txg quiesing or quiesced.
 565  * Abort the delay if this txg stalls or enters the quiesing state.
 566  */
 567 void
 568 txg_delay(dsl_pool_t *dp, uint64_t txg, hrtime_t delay, hrtime_t resolution)
 569 {
 570         tx_state_t *tx = &dp->dp_tx;
 571         hrtime_t start = gethrtime();

 573         /* don't delay if this txg could transition to quiescing immediately */
 573         /* don't delay if this txg could transition to quiesing immediately */
 574         if (tx->tx_open_txg > txg ||
 575             tx->tx_syncing_txg == txg-1 || tx->tx_synced_txg == txg-1)
 576                 return;

 578         mutex_enter(&tx->tx_sync_lock);
 579         if (tx->tx_open_txg > txg || tx->tx_synced_txg == txg-1) {
 580                 mutex_exit(&tx->tx_sync_lock);
 581                 return;
 582         }

 584         while (gethrtime() - start < delay &&
 585             tx->tx_syncing_txg < txg-1 && !txg_stalled(dp)) {
 586                 (void) cv_timedwait_hires(&tx->tx_quiesce_more_cv,
 587                     &tx->tx_sync_lock, delay, resolution, 0);
 588         }

 590         mutex_exit(&tx->tx_sync_lock);
 591 }
______unchanged_portion_omitted_
```

```
**********************************************************
   86358 Thu May 16 17:36:23 2013
new/usr/src/uts/common/fs/zfs/vdev.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 958 /*
 959  * Determine whether this device is accessible.
 960  *
 961  * Read and write to several known locations: the pad regions of each
 962  * vdev label but the first, which we leave alone in case it contains
 963  * a VTOC.
 959  * Determine whether this device is accessible by reading and writing
 960  * to several known locations: the pad regions of each vdev label
 961  * but the first (which we leave alone in case it contains a VTOC).
 964  */
 965 zio_t *
 966 vdev_probe(vdev_t *vd, zio_t *zio)
 967 {
 968         spa_t *spa = vd->vdev_spa;
 969         vdev_probe_stats_t *vps = NULL;
 970         zio_t *pio;

 972         ASSERT(vd->vdev_ops->vdev_op_leaf);

 974         /*
 975          * Don't probe the probe.
 976          */
 977         if (zio && (zio->io_flags & ZIO_FLAG_PROBE))
 978                 return (NULL);

 980         /*
 981          * To prevent 'probe storms' when a device fails, we create
 982          * just one probe i/o at a time.  All zios that want to probe
 983          * this vdev will become parents of the probe io.
 984          */
 985         mutex_enter(&vd->vdev_probe_lock);

 987         if ((pio = vd->vdev_probe_zio) == NULL) {
 988                 vps = kmem_zalloc(sizeof (*vps), KM_SLEEP);

 990                 vps->vps_flags = ZIO_FLAG_CANFAIL | ZIO_FLAG_PROBE |
 991                     ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_AGGREGATE |
 992                     ZIO_FLAG_TRYHARD;

 994                 if (spa_config_held(spa, SCL_ZIO, RW_WRITER)) {
 995                         /*
 996                          * vdev_cant_read and vdev_cant_write can only
 997                          * transition from TRUE to FALSE when we have the
 998                          * SCL_ZIO lock as writer; otherwise they can only
 999                          * transition from FALSE to TRUE.  This ensures that
1000                          * any zio looking at these values can assume that
1001                          * failures persist for the life of the I/O.  That's
1002                          * important because when a device has intermittent
1003                          * connectivity problems, we want to ensure that
1004                          * they're ascribed to the device (ENXIO) and not
1005                          * the zio (EIO).
1006                          *
1007                          * Since we hold SCL_ZIO as writer here, clear both
1008                          * values so the probe can reevaluate from first
```



```
1009                          * principles.
1010                          */
1011                         vps->vps_flags |= ZIO_FLAG_CONFIG_WRITER;
1012                         vd->vdev_cant_read = B_FALSE;
1013                         vd->vdev_cant_write = B_FALSE;
1014                 }

1016                 vd->vdev_probe_zio = pio = zio_null(NULL, spa, vd,
1017                     vdev_probe_done, vps,
1018                     vps->vps_flags | ZIO_FLAG_DONT_PROPAGATE);

1020                 /*
1021                  * We can't change the vdev state in this context, so we
1022                  * kick off an async task to do it on our behalf.
1023                  */
1024                 if (zio != NULL) {
1025                         vd->vdev_probe_wanted = B_TRUE;
1026                         spa_async_request(spa, SPA_ASYNC_PROBE);
1027                 }
1028         }

1030         if (zio != NULL)
1031                 zio_add_child(zio, pio);

1033         mutex_exit(&vd->vdev_probe_lock);

1035         if (vps == NULL) {
1036                 ASSERT(zio != NULL);
1037                 return (NULL);
1038         }

1040         for (int l = 1; l < VDEV_LABELS; l++) {
1041                 zio_nowait(zio_read_phys(pio, vd,
1042                     vdev_label_offset(vd->vdev_psize, l,
1043                     offsetof(vdev_label_t, vl_pad2)),
1044                     VDEV_PAD_SIZE, zio_buf_alloc(VDEV_PAD_SIZE),
1045                     ZIO_CHECKSUM_OFF, vdev_probe_done, vps,
1046                     ZIO_PRIORITY_SYNC_READ, vps->vps_flags, B_TRUE));
1047         }

1049         if (zio == NULL)
1050                 return (pio);

1052         zio_nowait(pio);
1053         return (NULL);
1054 }
______unchanged_portion_omitted_

2183 /*
2184  * Online the given vdev.
2185  *
2186  * If 'ZFS_ONLINE_UNSPARE' is set, it implies two things.  First, any attached
2187  * spare device should be detached when the device finishes resilvering.
2188  * Second, the online should be treated like a 'test' online case, so no FMA
2189  * events are generated if the device fails to open.
2182  * Online the given vdev.  If 'unspare' is set, it implies two things.  First,
2183  * any attached spare device should be detached when the device finishes
2184  * resilvering.  Second, the online should be treated like a 'test' online case,
2185  * so no FMA events are generated if the device fails to open.
2190  */
2191 int
2192 vdev_online(spa_t *spa, uint64_t guid, uint64_t flags, vdev_state_t *newstate)
2193 {
2194         vdev_t *vd, *tvd, *pvd, *rvd = spa->spa_root_vdev;

2196         spa_vdev_state_enter(spa, SCL_NONE);
```

```
2198         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2199                 return (spa_vdev_state_exit(spa, NULL, ENODEV));

2201         if (!vd->vdev_ops->vdev_op_leaf)
2202                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2204         tvd = vd->vdev_top;
2205         vd->vdev_offline = B_FALSE;
2206         vd->vdev_tmpoffline = B_FALSE;
2207         vd->vdev_checkremove = !!(flags & ZFS_ONLINE_CHECKREMOVE);
2208         vd->vdev_forcefault = !!(flags & ZFS_ONLINE_FORCEFAULT);

2210         /* XXX - L2ARC 1.0 does not support expansion */
2211         if (!vd->vdev_aux) {
2212                 for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2213                         pvd->vdev_expanding = !!(flags & ZFS_ONLINE_EXPAND);
2214         }

2216         vdev_reopen(tvd);
2217         vd->vdev_checkremove = vd->vdev_forcefault = B_FALSE;

2219         if (!vd->vdev_aux) {
2220                 for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2221                         pvd->vdev_expanding = B_FALSE;
2222         }

2224         if (newstate)
2225                 *newstate = vd->vdev_state;
2226         if ((flags & ZFS_ONLINE_UNSPARE) &&
2227             !vdev_is_dead(vd) && vd->vdev_parent &&
2228             vd->vdev_parent->vdev_ops == &vdev_spare_ops &&
2229             vd->vdev_parent->vdev_child[0] == vd)
2230                 vd->vdev_unspare = B_TRUE;

2232         if ((flags & ZFS_ONLINE_EXPAND) || spa->spa_autoexpand) {

2234                 /* XXX - L2ARC 1.0 does not support expansion */
2235                 if (vd->vdev_aux)
2236                         return (spa_vdev_state_exit(spa, vd, ENOTSUP));
2237                 spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);
2238         }
2239         return (spa_vdev_state_exit(spa, vd, 0));
2240 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   12090 Thu May 16 17:36:24 2013
new/usr/src/uts/common/fs/zfs/vdev_queue.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/vdev_impl.h>
#include <sys/spa_impl.h>
#include <sys/zio.h>
#include <sys/avl.h>

/*
 * These tunables are for performance analysis.
 */
/* The maximum number of I/Os concurrently pending to each device. */
int zfs_vdev_max_pending = 10;

#endif /* ! codereview */
/*
 * The initial number of I/Os pending to each device, before it starts ramping
 * up to zfs_vdev_max_pending.
 * zfs_vdev_max_pending is the maximum number of i/os concurrently
 * pending to each device.  zfs_vdev_min_pending is the initial number
 * of i/os pending to each device (before it starts ramping up to
 * max_pending).
 */
int zfs_vdev_max_pending = 10;
int zfs_vdev_min_pending = 4;

/*
 * The deadlines are grouped into buckets based on zfs_vdev_time_shift:
```



```
 * deadline = pri + gethrtime() >> time_shift)
 */
int zfs_vdev_time_shift = 29; /* each bucket is 0.537 seconds */

/* exponential I/O issue ramp-up rate */
int zfs_vdev_ramp_rate = 2;

/*
 * To reduce IOPs, we aggregate small adjacent I/Os into one large I/O.
 * For read I/Os, we also aggregate across small adjacency gaps; for writes
 * we include spans of optional I/Os to aid aggregation at the disk even when
 * they aren't able to help us aggregate at this level.
 */
int zfs_vdev_aggregation_limit = SPA_MAXBLOCKSIZE;
int zfs_vdev_read_gap_limit = 32 << 10;
int zfs_vdev_write_gap_limit = 4 << 10;

/*
 * Virtual device vector for disk I/O scheduling.
 */
int
vdev_queue_deadline_compare(const void *x1, const void *x2)
{
        const zio_t *z1 = x1;
        const zio_t *z2 = x2;

        if (z1->io_deadline < z2->io_deadline)
                return (-1);
        if (z1->io_deadline > z2->io_deadline)
                return (1);

        if (z1->io_offset < z2->io_offset)
                return (-1);
        if (z1->io_offset > z2->io_offset)
                return (1);

        if (z1 < z2)
                return (-1);
        if (z1 > z2)
                return (1);

        return (0);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   64391 Thu May 16 17:36:24 2013
new/usr/src/uts/common/fs/zfs/vdev_raidz.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/spa.h>
#include <sys/vdev_impl.h>
#include <sys/zio.h>
#include <sys/zio_checksum.h>
#include <sys/fs/zfs.h>
#include <sys/fm/fs/zfs.h>

/*
 * Virtual device vector for RAID-Z.
 *
 * This vdev supports single, double, and triple parity. For single parity,
 * we use a simple XOR of all the data columns. For double or triple parity,
 * we use a special case of Reed-Solomon coding. This extends the
 * technique described in "The mathematics of RAID-6" by H. Peter Anvin by
 * drawing on the system described in "A Tutorial on Reed-Solomon Coding for
 * Fault-Tolerance in RAID-like Systems" by James S. Plank on which the
 * former is also based. The latter is designed to provide higher performance
 * for writes.
 *
 * Note that the Plank paper claimed to support arbitrary N+M, but was then
 * amended six years later identifying a critical flaw that invalidates its
 * claims. Nevertheless, the technique can be adapted to work for up to
 * triple parity. For additional parity, the amendment "Note: Correction to
 * the 1997 Tutorial on Reed-Solomon Coding" by James S. Plank and Ying Ding
 * is viable, but the additional complexity means that write performance will
 * suffer.
 *
 * All of the methods above operate on a Galois field, defined over the
 * integers mod 2^N. In our case we choose N=8 for GF(8) so that all elements
```



```
 * can be expressed with a single byte. Briefly, the operations on the
 * field are defined as follows:
 *
 *   o addition (+) is represented by a bitwise XOR
 *   o subtraction (-) is therefore identical to addition: A + B = A - B
 *   o multiplication of A by 2 is defined by the following bitwise expression:
 *
#endif /* ! codereview */
 *      (A * 2)_7 = A_6
 *      (A * 2)_6 = A_5
 *      (A * 2)_5 = A_4
 *      (A * 2)_4 = A_3 + A_7
 *      (A * 2)_3 = A_2 + A_7
 *      (A * 2)_2 = A_1 + A_7
 *      (A * 2)_1 = A_0
 *      (A * 2)_0 = A_7
 *
 * In C, multiplying by 2 is therefore ((a << 1) ^ ((a & 0x80) ? 0x1d : 0)).
 * As an aside, this multiplication is derived from the error correcting
 * primitive polynomial x^8 + x^4 + x^3 + x^2 + 1.
 *
 * Observe that any number in the field (except for 0) can be expressed as a
 * power of 2 -- a generator for the field. We store a table of the powers of
 * 2 and logs base 2 for quick look ups, and exploit the fact that A * B can
 * be rewritten as 2^(log_2(A) + log_2(B)) (where '+' is normal addition rather
 * than field addition). The inverse of a field element A (A^-1) is therefore
 * A ^ (255 - 1) = A^254.
 *
 * The up-to-three parity columns, P, Q, R over several data columns,
 * D_0, ... D_n-1, can be expressed by field operations:
 *
 *      P = D_0 + D_1 + ... + D_n-2 + D_n-1
 *      Q = 2^n-1 * D_0 + 2^n-2 * D_1 + ... + 2^1 * D_n-2 + 2^0 * D_n-1
 *        = ((...((D_0) * 2 + D_1) * 2 + ...) * 2 + D_n-2) * 2 + D_n-1
 *      R = 4^n-1 * D_0 + 4^n-2 * D_1 + ... + 4^1 * D_n-2 + 4^0 * D_n-1
 *        = ((...((D_0) * 4 + D_1) * 4 + ...) * 4 + D_n-2) * 4 + D_n-1
 *
 * We chose 1, 2, and 4 as our generators because 1 corresponds to the trival
 * XOR operation, and 2 and 4 can be computed quickly and generate linearly-
 * independent coefficients. (There are no additional coefficients that have
 * this property which is why the uncorrected Plank method breaks down.)
 *
 * See the reconstruction code below for how P, Q and R can used individually
 * or in concert to recover missing data columns.
 */

typedef struct raidz_col {
        uint64_t rc_devidx;             /* child device index for I/O */
        uint64_t rc_offset;             /* device offset */
        uint64_t rc_size;               /* I/O size */
        void *rc_data;                  /* I/O data */
        void *rc_gdata;                 /* used to store the "good" version */
        int rc_error;                   /* I/O error for this device */
        uint8_t rc_tried;               /* Did we attempt this I/O column? */
        uint8_t rc_skipped;             /* Did we skip this I/O column? */
} raidz_col_t;

typedef struct raidz_map {
        uint64_t rm_cols;               /* Regular column count */
        uint64_t rm_scols;              /* Count including skipped columns */
        uint64_t rm_bigcols;            /* Number of oversized columns */
        uint64_t rm_asize;              /* Actual total I/O size */
        uint64_t rm_missingdata;        /* Count of missing data devices */
        uint64_t rm_missingparity;      /* Count of missing parity devices */
        uint64_t rm_firstdatacol;       /* First data column/parity count */
        uint64_t rm_nskip;              /* Skipped sectors for padding */
```

```
123         uint64_t rm_skipstart;          /* Column index of padding start */
124         void *rm_datacopy;              /* rm_asize-buffer of copied data */
125         uintptr_t rm_reports;           /* # of referencing checksum reports */
126         uint8_t rm_freed;               /* map no longer has referencing ZIO */
127         uint8_t rm_ecksuminjected;      /* checksum error was injected */
128         raidz_col_t rm_col[1];          /* Flexible array of I/O columns */
129 } raidz_map_t;

131 #define VDEV_RAIDZ_P            0
132 #define VDEV_RAIDZ_Q            1
133 #define VDEV_RAIDZ_R            2

135 #define VDEV_RAIDZ_MUL_2(x)     (((x) << 1) ^ (((x) & 0x80) ? 0x1d : 0))
136 #define VDEV_RAIDZ_MUL_4(x)     (VDEV_RAIDZ_MUL_2(VDEV_RAIDZ_MUL_2(x)))

138 /*
139  * We provide a mechanism to perform the field multiplication operation on a
140  * 64-bit value all at once rather than a byte at a time. This works by
141  * creating a mask from the top bit in each byte and using that to
142  * conditionally apply the XOR of 0x1d.
143  */
144 #define VDEV_RAIDZ_64MUL_2(x, mask) \
145 { \
146         (mask) = (x) & 0x8080808080808080ULL; \
147         (mask) = ((mask) << 1) - ((mask) >> 7); \
148         (x) = (((x) << 1) & 0xfefefefefefefefeULL) ^ \
149             ((mask) & 0x1d1d1d1d1d1d1d1d); \
150 }

152 #define VDEV_RAIDZ_64MUL_4(x, mask) \
153 { \
154         VDEV_RAIDZ_64MUL_2((x), mask); \
155         VDEV_RAIDZ_64MUL_2((x), mask); \
156 }

158 /*
159  * Force reconstruction to use the general purpose method.
160  */
161 int vdev_raidz_default_to_general;

163 /* Powers of 2 in the Galois field defined above. */
 63 /*
 64  * These two tables represent powers and logs of 2 in the Galois field defined
 65  * above. These values were computed by repeatedly multiplying by 2 as above.
 66  */
164 static const uint8_t vdev_raidz_pow2[256] = {
165         0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
166         0x1d, 0x3a, 0x74, 0xe8, 0xcd, 0x87, 0x13, 0x26,
167         0x4c, 0x98, 0x2d, 0x5a, 0xb4, 0x75, 0xea, 0xc9,
168         0x8f, 0x03, 0x06, 0x0c, 0x18, 0x30, 0x60, 0xc0,
169         0x9d, 0x27, 0x4e, 0x9c, 0x25, 0x4a, 0x94, 0x35,
170         0x6a, 0xd4, 0xb5, 0x77, 0xee, 0xc1, 0x9f, 0x23,
171         0x46, 0x8c, 0x05, 0x0a, 0x14, 0x28, 0x50, 0xa0,
172         0x5d, 0xba, 0x69, 0xd2, 0xb9, 0x6f, 0xde, 0xa1,
173         0x5f, 0xbe, 0x61, 0xc2, 0x99, 0x2f, 0x5e, 0xbc,
174         0x65, 0xca, 0x89, 0x0f, 0x1e, 0x3c, 0x78, 0xf0,
175         0xfd, 0xe7, 0xd3, 0xbb, 0x6b, 0xd6, 0xb1, 0x7f,
176         0xfe, 0xe1, 0xdf, 0xa3, 0x5b, 0xb6, 0x71, 0xe2,
177         0xd9, 0xaf, 0x43, 0x86, 0x11, 0x22, 0x44, 0x88,
178         0x0d, 0x1a, 0x34, 0x68, 0xd0, 0xbd, 0x67, 0xce,
179         0x81, 0x1f, 0x3e, 0x7c, 0xf8, 0xed, 0xc7, 0x93,
180         0x3b, 0x76, 0xec, 0xc5, 0x97, 0x33, 0x66, 0xcc,
181         0x85, 0x17, 0x2e, 0x5c, 0xb8, 0x6d, 0xda, 0xa9,
182         0x4f, 0x9e, 0x21, 0x42, 0x84, 0x15, 0x2a, 0x54,
183         0xa8, 0x4d, 0x9a, 0x29, 0x52, 0xa4, 0x55, 0xaa,
184         0x49, 0x92, 0x39, 0x72, 0xe4, 0xd5, 0xb7, 0x73,
```



```
185         0xe6, 0xd1, 0xbf, 0x63, 0xc6, 0x91, 0x3f, 0x7e,
186         0xfc, 0xe5, 0xd7, 0xb3, 0x7b, 0xf6, 0xf1, 0xff,
187         0xe3, 0xdb, 0xab, 0x4b, 0x96, 0x31, 0x62, 0xc4,
188         0x95, 0x37, 0x6e, 0xdc, 0xa5, 0x57, 0xae, 0x41,
189         0x82, 0x19, 0x32, 0x64, 0xc8, 0x8d, 0x07, 0x0e,
190         0x1c, 0x38, 0x70, 0xe0, 0xdd, 0xa7, 0x53, 0xa6,
191         0x51, 0xa2, 0x59, 0xb2, 0x79, 0xf2, 0xf9, 0xef,
192         0xc3, 0x9b, 0x2b, 0x56, 0xac, 0x45, 0x8a, 0x09,
193         0x12, 0x24, 0x48, 0x90, 0x3d, 0x7a, 0xf4, 0xf5,
194         0xf7, 0xf3, 0xfb, 0xeb, 0xcb, 0x8b, 0x0b, 0x16,
195         0x2c, 0x58, 0xb0, 0x7d, 0xfa, 0xe9, 0xcf, 0x83,
196         0x1b, 0x36, 0x6c, 0xd8, 0xad, 0x47, 0x8e, 0x01
197 };
198 /* Logs of 2 in the Galois field defined above. */
199 #endif /* ! codereview */
200 static const uint8_t vdev_raidz_log2[256] = {
201         0x00, 0x00, 0x01, 0x19, 0x02, 0x32, 0x1a, 0xc6,
202         0x03, 0xdf, 0x33, 0xee, 0x1b, 0x68, 0xc7, 0x4b,
203         0x04, 0x64, 0xe0, 0x0e, 0x34, 0x8d, 0xef, 0x81,
204         0x1c, 0xc1, 0x69, 0xf8, 0xc8, 0x08, 0x4c, 0x71,
205         0x05, 0x8a, 0x65, 0x2f, 0xe1, 0x24, 0x0f, 0x21,
206         0x35, 0x93, 0x8e, 0xda, 0xf0, 0x12, 0x82, 0x45,
207         0x1d, 0xb5, 0xc2, 0x7d, 0x6a, 0x27, 0xf9, 0xb9,
208         0xc9, 0x9a, 0x09, 0x78, 0x4d, 0xe4, 0x72, 0xa6,
209         0x06, 0xbf, 0x8b, 0x62, 0x66, 0xdd, 0x30, 0xfd,
210         0xe2, 0x98, 0x25, 0xb3, 0x10, 0x91, 0x22, 0x88,
211         0x36, 0xd0, 0x94, 0xce, 0x8f, 0x96, 0xdb, 0xbd,
212         0xf1, 0xd2, 0x13, 0x5c, 0x83, 0x38, 0x46, 0x40,
213         0x1e, 0x42, 0xb6, 0xa3, 0xc3, 0x48, 0x7e, 0x6e,
214         0x6b, 0x3a, 0x28, 0x54, 0xfa, 0x85, 0xba, 0x3d,
215         0xca, 0x5e, 0x9b, 0x9f, 0x0a, 0x15, 0x79, 0x2b,
216         0x4e, 0xd4, 0xe5, 0xac, 0x73, 0xf3, 0xa7, 0x57,
217         0x07, 0x70, 0xc0, 0xf7, 0x8c, 0x80, 0x63, 0x0d,
218         0x67, 0x4a, 0xde, 0xed, 0x31, 0xc5, 0xfe, 0x18,
219         0xe3, 0xa5, 0x99, 0x77, 0x26, 0xb8, 0xb4, 0x7c,
220         0x11, 0x44, 0x92, 0xd9, 0x23, 0x20, 0x89, 0x2e,
221         0x37, 0x3f, 0xd1, 0x5b, 0x95, 0xbc, 0xcf, 0xcd,
222         0x90, 0x87, 0x97, 0xb2, 0xdc, 0xfc, 0xbe, 0x61,
223         0xf2, 0x56, 0xd3, 0xab, 0x14, 0x2a, 0x5d, 0x9e,
224         0x84, 0x3c, 0x39, 0x53, 0x47, 0x6d, 0x41, 0xa2,
225         0x1f, 0x2d, 0x43, 0xd8, 0xb7, 0x7b, 0xa4, 0x76,
226         0xc4, 0x17, 0x49, 0xec, 0x7f, 0x0c, 0x6f, 0xf6,
227         0x6c, 0xa1, 0x3b, 0x52, 0x29, 0x9d, 0x55, 0xaa,
228         0xfb, 0x60, 0x86, 0xb1, 0xbb, 0xcc, 0x3e, 0x5a,
229         0xcb, 0x59, 0x5f, 0xb0, 0x9c, 0xa9, 0xa0, 0x51,
230         0x0b, 0xf5, 0x16, 0xeb, 0x7a, 0x75, 0x2c, 0xd7,
231         0x4f, 0xae, 0xd5, 0xe9, 0xe6, 0xe7, 0xad, 0xe8,
232         0x74, 0xd6, 0xf4, 0xea, 0xa8, 0x50, 0x58, 0xaf,
233 };

235 static void vdev_raidz_generate_parity(raidz_map_t *rm);

237 /*
238  * Multiply a given number by 2 raised to the given power.
239  */
240 static uint8_t
241 vdev_raidz_exp2(uint_t a, int exp)
242 {
243         if (a == 0)
244                 return (0);

246         ASSERT(exp >= 0);
247         ASSERT(vdev_raidz_log2[a] > 0 || a == 1);

249         exp += vdev_raidz_log2[a];
250         if (exp > 255)
```

```
251                 exp -= 255;
253         return (vdev_raidz_pow2[exp]);
254 }
256 static void
257 vdev_raidz_map_free(raidz_map_t *rm)
258 {
259         int c;
260         size_t size;
262         for (c = 0; c < rm->rm_firstdatacol; c++) {
263                 zio_buf_free(rm->rm_col[c].rc_data, rm->rm_col[c].rc_size);
265                 if (rm->rm_col[c].rc_gdata != NULL)
266                         zio_buf_free(rm->rm_col[c].rc_gdata,
267                             rm->rm_col[c].rc_size);
268         }
270         size = 0;
271         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++)
272                 size += rm->rm_col[c].rc_size;
274         if (rm->rm_datacopy != NULL)
275                 zio_buf_free(rm->rm_datacopy, size);
277         kmem_free(rm, offsetof(raidz_map_t, rm_col[rm->rm_scols]));
278 }
280 static void
281 vdev_raidz_map_free_vsd(zio_t *zio)
282 {
283         raidz_map_t *rm = zio->io_vsd;
285         ASSERT0(rm->rm_freed);
286         rm->rm_freed = 1;
288         if (rm->rm_reports == 0)
289                 vdev_raidz_map_free(rm);
290 }
292 /*ARGSUSED*/
293 static void
294 vdev_raidz_cksum_free(void *arg, size_t ignored)
295 {
296         raidz_map_t *rm = arg;
298         ASSERT3U(rm->rm_reports, >, 0);
300         if (--rm->rm_reports == 0 && rm->rm_freed != 0)
301                 vdev_raidz_map_free(rm);
302 }
304 static void
305 vdev_raidz_cksum_finish(zio_cksum_report_t *zcr, const void *good_data)
306 {
307         raidz_map_t *rm = zcr->zcr_cbdata;
308         size_t c = zcr->zcr_cbinfo;
309         size_t x;
311         const char *good = NULL;
312         const char *bad = rm->rm_col[c].rc_data;
314         if (good_data == NULL) {
315                 zfs_ereport_finish_checksum(zcr, NULL, NULL, B_FALSE);
316                 return;
```



```
317         }
319         if (c < rm->rm_firstdatacol) {
320                 /*
321                  * The first time through, calculate the parity blocks for
322                  * the good data (this relies on the fact that the good
323                  * data never changes for a given logical ZIO)
324                  */
325                 if (rm->rm_col[0].rc_gdata == NULL) {
326                         char *bad_parity[VDEV_RAIDZ_MAXPARITY];
327                         char *buf;
329                         /*
330                          * Set up the rm_col[]s to generate the parity for
331                          * good_data, first saving the parity bufs and
332                          * replacing them with buffers to hold the result.
333                          */
334                         for (x = 0; x < rm->rm_firstdatacol; x++) {
335                                 bad_parity[x] = rm->rm_col[x].rc_data;
336                                 rm->rm_col[x].rc_data = rm->rm_col[x].rc_gdata =
337                                     zio_buf_alloc(rm->rm_col[x].rc_size);
338                         }
340                         /* fill in the data columns from good_data */
341                         buf = (char *)good_data;
342                         for (; x < rm->rm_cols; x++) {
343                                 rm->rm_col[x].rc_data = buf;
344                                 buf += rm->rm_col[x].rc_size;
345                         }
347                         /*
348                          * Construct the parity from the good data.
349                          */
350                         vdev_raidz_generate_parity(rm);
352                         /* restore everything back to its original state */
353                         for (x = 0; x < rm->rm_firstdatacol; x++)
354                                 rm->rm_col[x].rc_data = bad_parity[x];
356                         buf = rm->rm_datacopy;
357                         for (x = rm->rm_firstdatacol; x < rm->rm_cols; x++) {
358                                 rm->rm_col[x].rc_data = buf;
359                                 buf += rm->rm_col[x].rc_size;
360                         }
361                 }
363                 ASSERT3P(rm->rm_col[c].rc_gdata, !=, NULL);
364                 good = rm->rm_col[c].rc_gdata;
365         } else {
366                 /* adjust good_data to point at the start of our column */
367                 good = good_data;
369                 for (x = rm->rm_firstdatacol; x < c; x++)
370                         good += rm->rm_col[x].rc_size;
371         }
373         /* we drop the ereport if it ends up that the data was good */
374         zfs_ereport_finish_checksum(zcr, good, bad, B_TRUE);
375 }
377 /*
378  * Invoked indirectly by zfs_ereport_start_checksum(), called
379  * below when our read operation fails completely.  The main point
380  * is to keep a copy of everything we read from disk, so that at
381  * vdev_raidz_cksum_finish() time we can compare it with the good data.
382  */
```

```
383 static void
384 vdev_raidz_cksum_report(zio_t *zio, zio_cksum_report_t *zcr, void *arg)
385 {
386         size_t c = (size_t)(uintptr_t)arg;
387         caddr_t buf;
388
389         raidz_map_t *rm = zio->io_vsd;
390         size_t size;
391
392         /* set up the report and bump the refcount  */
393         zcr->zcr_cbdata = rm;
394         zcr->zcr_cbinfo = c;
395         zcr->zcr_finish = vdev_raidz_cksum_finish;
396         zcr->zcr_free = vdev_raidz_cksum_free;
397
398         rm->rm_reports++;
399         ASSERT3U(rm->rm_reports, >, 0);
400
401         if (rm->rm_datacopy != NULL)
402                 return;
403
404         /*
405          * It's the first time we're called for this raidz_map_t, so we need
406          * to copy the data aside; there's no guarantee that our zio's buffer
407          * won't be re-used for something else.
408          *
409          * Our parity data is already in separate buffers, so there's no need
410          * to copy them.
411          */
412
413         size = 0;
414         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++)
415                 size += rm->rm_col[c].rc_size;
416
417         buf = rm->rm_datacopy = zio_buf_alloc(size);
418
419         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
420                 raidz_col_t *col = &rm->rm_col[c];
421
422                 bcopy(col->rc_data, buf, col->rc_size);
423                 col->rc_data = buf;
424
425                 buf += col->rc_size;
426         }
427         ASSERT3P(buf - (caddr_t)rm->rm_datacopy, ==, size);
428 }
429
430 static const zio_vsd_ops_t vdev_raidz_vsd_ops = {
431         vdev_raidz_map_free_vsd,
432         vdev_raidz_cksum_report
433 };
434
435 /*
436  * Divides the IO evenly across all child vdevs; usually, dcols is
437  * the number of children in the target vdev.
438  */
439 static raidz_map_t *
440 vdev_raidz_map_alloc(zio_t *zio, uint64_t unit_shift, uint64_t dcols,
441     uint64_t nparity)
442 {
443         raidz_map_t *rm;
444         /* The starting RAIDZ (parent) vdev sector of the block. */
445         uint64_t b = zio->io_offset >> unit_shift;
446         /* The zio's size in units of the vdev's minimum sector size. */
447         uint64_t s = zio->io_size >> unit_shift;
448         /* The first column for this stripe. */
```



```
449         uint64_t f = b % dcols;
450         /* The starting byte offset on each child vdev. */
451         uint64_t o = (b / dcols) << unit_shift;
452         uint64_t q, r, c, bc, col, acols, scols, coff, devidx, asize, tot;
453
454         /*
455          * "Quotient": The number of data sectors for this stripe on all but
456          * the "big column" child vdevs that also contain "remainder" data.
457          */
458         q = s / (dcols - nparity);
459
460         /*
461          * "Remainder": The number of partial stripe data sectors in this I/O.
462          * This will add a sector to some, but not all, child vdevs.
463          */
464         r = s - q * (dcols - nparity);
465
466         /* The number of "big columns" - those which contain remainder data. */
467         bc = (r == 0 ? 0 : r + nparity);
468
469         /*
470          * The total number of data and parity sectors associated with
471          * this I/O.
472          */
473         tot = s + nparity * (q + (r == 0 ? 0 : 1));
474
475         /* acols: The columns that will be accessed. */
476         /* scols: The columns that will be accessed or skipped. */
477         if (q == 0) {
478                 /* Our I/O request doesn't span all child vdevs. */
479                 acols = bc;
480                 scols = MIN(dcols, roundup(bc, nparity + 1));
481         } else {
482                 acols = dcols;
483                 scols = dcols;
484         }
485
486         ASSERT3U(acols, <=, scols);
487
488         rm = kmem_alloc(offsetof(raidz_map_t, rm_col[scols]), KM_SLEEP);
489
490         rm->rm_cols = acols;
491         rm->rm_scols = scols;
492         rm->rm_bigcols = bc;
493         rm->rm_skipstart = bc;
494         rm->rm_missingdata = 0;
495         rm->rm_missingparity = 0;
496         rm->rm_firstdatacol = nparity;
497         rm->rm_datacopy = NULL;
498         rm->rm_reports = 0;
499         rm->rm_freed = 0;
500         rm->rm_ecksuminjected = 0;
501
502         asize = 0;
503
504         for (c = 0; c < scols; c++) {
505                 col = f + c;
506                 coff = o;
507                 if (col >= dcols) {
508                         col -= dcols;
509                         coff += 1ULL << unit_shift;
510                 }
511                 rm->rm_col[c].rc_devidx = col;
512                 rm->rm_col[c].rc_offset = coff;
513                 rm->rm_col[c].rc_data = NULL;
514                 rm->rm_col[c].rc_gdata = NULL;
```

```
515                 rm->rm_col[c].rc_error = 0;
516                 rm->rm_col[c].rc_tried = 0;
517                 rm->rm_col[c].rc_skipped = 0;

519                 if (c >= acols)
520                         rm->rm_col[c].rc_size = 0;
521                 else if (c < bc)
522                         rm->rm_col[c].rc_size = (q + 1) << unit_shift;
523                 else
524                         rm->rm_col[c].rc_size = q << unit_shift;

526                 asize += rm->rm_col[c].rc_size;
527         }

529         ASSERT3U(asize, ==, tot << unit_shift);
530         rm->rm_asize = roundup(asize, (nparity + 1) << unit_shift);
531         rm->rm_nskip = roundup(tot, nparity + 1) - tot;
532         ASSERT3U(rm->rm_asize - asize, ==, rm->rm_nskip << unit_shift);
533         ASSERT3U(rm->rm_nskip, <=, nparity);

535         for (c = 0; c < rm->rm_firstdatacol; c++)
536                 rm->rm_col[c].rc_data = zio_buf_alloc(rm->rm_col[c].rc_size);

538         rm->rm_col[c].rc_data = zio->io_data;

540         for (c = c + 1; c < acols; c++)
541                 rm->rm_col[c].rc_data = (char *)rm->rm_col[c - 1].rc_data +
542                     rm->rm_col[c - 1].rc_size;

544         /*
545          * If all data stored spans all columns, there's a danger that parity
546          * will always be on the same device and, since parity isn't read
547          * during normal operation, that that device's I/O bandwidth won't be
548          * used effectively. We therefore switch the parity every 1MB.
549          *
550          * ... at least that was, ostensibly, the theory. As a practical
551          * matter unless we juggle the parity between all devices evenly, we
552          * won't see any benefit. Further, occasional writes that aren't a
553          * multiple of the LCM of the number of children and the minimum
554          * stripe width are sufficient to avoid pessimal behavior.
555          * Unfortunately, this decision created an implicit on-disk format
556          * requirement that we need to support for all eternity, but only
557          * for single-parity RAID-Z.
558          *
559          * If we intend to skip a sector in the zeroth column for padding
560          * we must make sure to note this swap. We will never intend to
561          * skip the first column since at least one data and one parity
562          * column must appear in each row.
563          */
564         ASSERT(rm->rm_cols >= 2);
565         ASSERT(rm->rm_col[0].rc_size == rm->rm_col[1].rc_size);

567         if (rm->rm_firstdatacol == 1 && (zio->io_offset & (1ULL << 20))) {
568                 devidx = rm->rm_col[0].rc_devidx;
569                 o = rm->rm_col[0].rc_offset;
570                 rm->rm_col[0].rc_devidx = rm->rm_col[1].rc_devidx;
571                 rm->rm_col[0].rc_offset = rm->rm_col[1].rc_offset;
572                 rm->rm_col[1].rc_devidx = devidx;
573                 rm->rm_col[1].rc_offset = o;

575                 if (rm->rm_skipstart == 0)
576                         rm->rm_skipstart = 1;
577         }

579         zio->io_vsd = rm;
580         zio->io_vsd_ops = &vdev_raidz_vsd_ops;
```



```
581         return (rm);
582 }

584 static void
585 vdev_raidz_generate_parity_p(raidz_map_t *rm)
586 {
587         uint64_t *p, *src, pcount, ccount, i;
588         int c;

590         pcount = rm->rm_col[VDEV_RAIDZ_P].rc_size / sizeof (src[0]);

592         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
593                 src = rm->rm_col[c].rc_data;
594                 p = rm->rm_col[VDEV_RAIDZ_P].rc_data;
595                 ccount = rm->rm_col[c].rc_size / sizeof (src[0]);

597                 if (c == rm->rm_firstdatacol) {
598                         ASSERT(ccount == pcount);
599                         for (i = 0; i < ccount; i++, src++, p++) {
600                                 *p = *src;
601                         }
602                 } else {
603                         ASSERT(ccount <= pcount);
604                         for (i = 0; i < ccount; i++, src++, p++) {
605                                 *p ^= *src;
606                         }
607                 }
608         }
609 }

611 static void
612 vdev_raidz_generate_parity_pq(raidz_map_t *rm)
613 {
614         uint64_t *p, *q, *src, pcnt, ccnt, mask, i;
615         int c;

617         pcnt = rm->rm_col[VDEV_RAIDZ_P].rc_size / sizeof (src[0]);
618         ASSERT(rm->rm_col[VDEV_RAIDZ_P].rc_size ==
619             rm->rm_col[VDEV_RAIDZ_Q].rc_size);

621         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
622                 src = rm->rm_col[c].rc_data;
623                 p = rm->rm_col[VDEV_RAIDZ_P].rc_data;
624                 q = rm->rm_col[VDEV_RAIDZ_Q].rc_data;

626                 ccnt = rm->rm_col[c].rc_size / sizeof (src[0]);

628                 if (c == rm->rm_firstdatacol) {
629                         ASSERT(ccnt == pcnt || ccnt == 0);
630                         for (i = 0; i < ccnt; i++, src++, p++, q++) {
631                                 *p = *src;
632                                 *q = *src;
633                         }
634                         for (; i < pcnt; i++, src++, p++, q++) {
635                                 *p = 0;
636                                 *q = 0;
637                         }
638                 } else {
639                         ASSERT(ccnt <= pcnt);

641                         /*
642                          * Apply the algorithm described above by multiplying
643                          * the previous result and adding in the new value.
644                          */
645                         for (i = 0; i < ccnt; i++, src++, p++, q++) {
646                                 *p ^= *src;
```

```
                                VDEV_RAIDZ_64MUL_2(*q, mask);
                                *q ^= *src;
                        }

                        /*
                         * Treat short columns as though they are full of 0s.
                         * Note that there's therefore nothing needed for P.
                         */
                        for (; i < pcnt; i++, q++) {
                                VDEV_RAIDZ_64MUL_2(*q, mask);
                        }
                }
        }
}

static void
vdev_raidz_generate_parity_pqr(raidz_map_t *rm)
{
        uint64_t *p, *q, *r, *src, pcnt, ccnt, mask, i;
        int c;

        pcnt = rm->rm_col[VDEV_RAIDZ_P].rc_size / sizeof (src[0]);
        ASSERT(rm->rm_col[VDEV_RAIDZ_P].rc_size ==
            rm->rm_col[VDEV_RAIDZ_Q].rc_size);
        ASSERT(rm->rm_col[VDEV_RAIDZ_P].rc_size ==
            rm->rm_col[VDEV_RAIDZ_R].rc_size);

        for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
                src = rm->rm_col[c].rc_data;
                p = rm->rm_col[VDEV_RAIDZ_P].rc_data;
                q = rm->rm_col[VDEV_RAIDZ_Q].rc_data;
                r = rm->rm_col[VDEV_RAIDZ_R].rc_data;

                ccnt = rm->rm_col[c].rc_size / sizeof (src[0]);

                if (c == rm->rm_firstdatacol) {
                        ASSERT(ccnt == pcnt || ccnt == 0);
                        for (i = 0; i < ccnt; i++, src++, p++, q++, r++) {
                                *p = *src;
                                *q = *src;
                                *r = *src;
                        }
                        for (; i < pcnt; i++, src++, p++, q++, r++) {
                                *p = 0;
                                *q = 0;
                                *r = 0;
                        }
                } else {
                        ASSERT(ccnt <= pcnt);

                        /*
                         * Apply the algorithm described above by multiplying
                         * the previous result and adding in the new value.
                         */
                        for (i = 0; i < ccnt; i++, src++, p++, q++, r++) {
                                *p ^= *src;

                                VDEV_RAIDZ_64MUL_2(*q, mask);
                                *q ^= *src;

                                VDEV_RAIDZ_64MUL_4(*r, mask);
                                *r ^= *src;
                        }

                        /*
```



```
                         * Treat short columns as though they are full of 0s.
                         * Note that there's therefore nothing needed for P.
                         */
                        for (; i < pcnt; i++, q++, r++) {
                                VDEV_RAIDZ_64MUL_2(*q, mask);
                                VDEV_RAIDZ_64MUL_4(*r, mask);
                        }
                }
        }
}

/*
 * Generate RAID parity in the first virtual columns according to the number of
 * parity columns available.
 */
static void
vdev_raidz_generate_parity(raidz_map_t *rm)
{
        switch (rm->rm_firstdatacol) {
        case 1:
                vdev_raidz_generate_parity_p(rm);
                break;
        case 2:
                vdev_raidz_generate_parity_pq(rm);
                break;
        case 3:
                vdev_raidz_generate_parity_pqr(rm);
                break;
        default:
                cmn_err(CE_PANIC, "invalid RAID-Z configuration");
        }
}

static int
vdev_raidz_reconstruct_p(raidz_map_t *rm, int *tgts, int ntgts)
{
        uint64_t *dst, *src, xcount, ccount, count, i;
        int x = tgts[0];
        int c;

        ASSERT(ntgts == 1);
        ASSERT(x >= rm->rm_firstdatacol);
        ASSERT(x < rm->rm_cols);

        xcount = rm->rm_col[x].rc_size / sizeof (src[0]);
        ASSERT(xcount <= rm->rm_col[VDEV_RAIDZ_P].rc_size / sizeof (src[0]));
        ASSERT(xcount > 0);

        src = rm->rm_col[VDEV_RAIDZ_P].rc_data;
        dst = rm->rm_col[x].rc_data;
        for (i = 0; i < xcount; i++, dst++, src++) {
                *dst = *src;
        }

        for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
                src = rm->rm_col[c].rc_data;
                dst = rm->rm_col[x].rc_data;

                if (c == x)
                        continue;

                ccount = rm->rm_col[c].rc_size / sizeof (src[0]);
                count = MIN(ccount, xcount);

                for (i = 0; i < count; i++, dst++, src++) {
                        *dst ^= *src;
```

```
779                 }
780         }

782         return (1 << VDEV_RAIDZ_P);
783 }

785 static int
786 vdev_raidz_reconstruct_q(raidz_map_t *rm, int *tgts, int ntgts)
787 {
788         uint64_t *dst, *src, xcount, ccount, count, mask, i;
789         uint8_t *b;
790         int x = tgts[0];
791         int c, j, exp;

793         ASSERT(ntgts == 1);

795         xcount = rm->rm_col[x].rc_size / sizeof (src[0]);
796         ASSERT(xcount <= rm->rm_col[VDEV_RAIDZ_Q].rc_size / sizeof (src[0]));

798         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
799                 src = rm->rm_col[c].rc_data;
800                 dst = rm->rm_col[x].rc_data;

802                 if (c == x)
803                         ccount = 0;
804                 else
805                         ccount = rm->rm_col[c].rc_size / sizeof (src[0]);

807                 count = MIN(ccount, xcount);

809                 if (c == rm->rm_firstdatacol) {
810                         for (i = 0; i < count; i++, dst++, src++) {
811                                 *dst = *src;
812                         }
813                         for (; i < xcount; i++, dst++) {
814                                 *dst = 0;
815                         }

817                 } else {
818                         for (i = 0; i < count; i++, dst++, src++) {
819                                 VDEV_RAIDZ_64MUL_2(*dst, mask);
820                                 *dst ^= *src;
821                         }

823                         for (; i < xcount; i++, dst++) {
824                                 VDEV_RAIDZ_64MUL_2(*dst, mask);
825                         }
826                 }
827         }

829         src = rm->rm_col[VDEV_RAIDZ_Q].rc_data;
830         dst = rm->rm_col[x].rc_data;
831         exp = 255 - (rm->rm_cols - 1 - x);

833         for (i = 0; i < xcount; i++, dst++, src++) {
834                 *dst ^= *src;
835                 for (j = 0, b = (uint8_t *)dst; j < 8; j++, b++) {
836                         *b = vdev_raidz_exp2(*b, exp);
837                 }
838         }

840         return (1 << VDEV_RAIDZ_Q);
841 }

843 static int
844 vdev_raidz_reconstruct_pq(raidz_map_t *rm, int *tgts, int ntgts)
```



```
845 {
846         uint8_t *p, *q, *pxy, *qxy, *xd, *yd, tmp, a, b, aexp, bexp;
847         void *pdata, *qdata;
848         uint64_t xsize, ysize, i;
849         int x = tgts[0];
850         int y = tgts[1];

852         ASSERT(ntgts == 2);
853         ASSERT(x < y);
854         ASSERT(x >= rm->rm_firstdatacol);
855         ASSERT(y < rm->rm_cols);

857         ASSERT(rm->rm_col[x].rc_size >= rm->rm_col[y].rc_size);

859         /*
860          * Move the parity data aside -- we're going to compute parity as
861          * though columns x and y were full of zeros -- Pxy and Qxy. We want to
862          * reuse the parity generation mechanism without trashing the actual
863          * parity so we make those columns appear to be full of zeros by
864          * setting their lengths to zero.
865          */
866         pdata = rm->rm_col[VDEV_RAIDZ_P].rc_data;
867         qdata = rm->rm_col[VDEV_RAIDZ_Q].rc_data;
868         xsize = rm->rm_col[x].rc_size;
869         ysize = rm->rm_col[y].rc_size;

871         rm->rm_col[VDEV_RAIDZ_P].rc_data =
872             zio_buf_alloc(rm->rm_col[VDEV_RAIDZ_P].rc_size);
873         rm->rm_col[VDEV_RAIDZ_Q].rc_data =
874             zio_buf_alloc(rm->rm_col[VDEV_RAIDZ_Q].rc_size);
875         rm->rm_col[x].rc_size = 0;
876         rm->rm_col[y].rc_size = 0;

878         vdev_raidz_generate_parity_pq(rm);

880         rm->rm_col[x].rc_size = xsize;
881         rm->rm_col[y].rc_size = ysize;

883         p = pdata;
884         q = qdata;
885         pxy = rm->rm_col[VDEV_RAIDZ_P].rc_data;
886         qxy = rm->rm_col[VDEV_RAIDZ_Q].rc_data;
887         xd = rm->rm_col[x].rc_data;
888         yd = rm->rm_col[y].rc_data;

890         /*
891          * We now have:
892          *      Pxy = P + D_x + D_y
893          *      Qxy = Q + 2^(ndevs - 1 - x) * D_x + 2^(ndevs - 1 - y) * D_y
894          *
895          * We can then solve for D_x:
896          *      D_x = A * (P + Pxy) + B * (Q + Qxy)
897          * where
898          *      A = 2^(x - y) * (2^(x - y) + 1)^-1
899          *      B = 2^(ndevs - 1 - x) * (2^(x - y) + 1)^-1
900          *
901          * With D_x in hand, we can easily solve for D_y:
902          *      D_y = P + Pxy + D_x
903          */

905         a = vdev_raidz_pow2[255 + x - y];
906         b = vdev_raidz_pow2[255 - (rm->rm_cols - 1 - x)];
907         tmp = 255 - vdev_raidz_log2[a ^ 1];

909         aexp = vdev_raidz_log2[vdev_raidz_exp2(a, tmp)];
910         bexp = vdev_raidz_log2[vdev_raidz_exp2(b, tmp)];
```

```
912         for (i = 0; i < xsize; i++, p++, q++, pxy++, qxy++, xd++, yd++) {
913                 *xd = vdev_raidz_exp2(*p ^ *pxy, aexp) ^
914                     vdev_raidz_exp2(*q ^ *qxy, bexp);
915
916                 if (i < ysize)
917                         *yd = *p ^ *pxy ^ *xd;
918         }
919
920         zio_buf_free(rm->rm_col[VDEV_RAIDZ_P].rc_data,
921             rm->rm_col[VDEV_RAIDZ_P].rc_size);
922         zio_buf_free(rm->rm_col[VDEV_RAIDZ_Q].rc_data,
923             rm->rm_col[VDEV_RAIDZ_Q].rc_size);
924
925         /*
926          * Restore the saved parity data.
927          */
928         rm->rm_col[VDEV_RAIDZ_P].rc_data = pdata;
929         rm->rm_col[VDEV_RAIDZ_Q].rc_data = qdata;
930
931         return ((1 << VDEV_RAIDZ_P) | (1 << VDEV_RAIDZ_Q));
932 }
933
934 /* BEGIN CSTYLED */
935 /*
936  * In the general case of reconstruction, we must solve the system of linear
937  * equations defined by the coeffecients used to generate parity as well as
938  * the contents of the data and parity disks. This can be expressed with
939  * vectors for the original data (D) and the actual data (d) and parity (p)
940  * and a matrix composed of the identity matrix (I) and a dispersal matrix (V):
941  *
942  *             __   __                     __     __
943  *             |     |         __     __   |  p_0  |
944  *             |  V  |         |  D_0  |   | p_m-1 |
945  *             |     |    x    |   :   | = |  d_0  |
946  *             |  I  |         | D_n-1 |   |   :   |
947  *             |     |         ~~     ~~   | d_n-1 |
948  *             ~~   ~~                     ~~     ~~
949  *
950  * I is simply a square identity matrix of size n, and V is a vandermonde
951  * matrix defined by the coeffecients we chose for the various parity columns
952  * (1, 2, 4). Note that these values were chosen both for simplicity, speedy
953  * computation as well as linear separability.
954  *
955  *       __                __               __     __
956  *       |   1   ..  1 1 1 |             |  p_0  |
957  *       | 2^n-1 ..  4 2 1 |   __     __   |   :   |
958  *       | 4^n-1 .. 16 4 1 |   |  D_0  |   | p_m-1 |
959  *       |   1   ..  0 0 0 |   |  D_1  |   |  d_0  |
960  *       |   0   ..  0 0 0 | x |  D_2  | = |  d_1  |
961  *       |   :       : : : |   |   :   |   |  d_2  |
962  *       |   0   ..  1 0 0 |   | D_n-1 |   |   :   |
963  *       |   0   ..  0 1 0 |   ~~     ~~   |   :   |
964  *       |   0   ..  0 0 1 |               | d_n-1 |
965  *       ~~               ~~               ~~     ~~
966  *
967  * Note that I, V, d, and p are known. To compute D, we must invert the
968  * matrix and use the known data and parity values to reconstruct the unknown
969  * data values. We begin by removing the rows in V|I and d|p that correspond
970  * to failed or missing columns; we then make V|I square (n x n) and d|p
971  * sized n by removing rows corresponding to unused parity from the bottom up
972  * to generate (V|I)' and (d|p)'. We can then generate the inverse of (V|I)'
973  * using Gauss-Jordan elimination. In the example below we use m=3 parity
974  * columns, n=8 data columns, with errors in d_1, d_2, and p_1:
975  *           __                                                 __
976  *           |  1   1   1   1   1   1   1   1  |
```



```
977  *           | 128  64  32  16  8   4   2   1  | <-----+-+-- missing disks
978  *           |  19 205 116  29  64  16  4   1  |       / /
979  *           |  1   0   0   0   0   0   0   0  |      / /
980  *           |  0   1   0   0   0   0   0   0  | <--' /
981  *  (V|I)  = |  0   0   1   0   0   0   0   0  | <---'
982  *           |  0   0   0   1   0   0   0   0  |
983  *           |  0   0   0   0   1   0   0   0  |
984  *           |  0   0   0   0   0   1   0   0  |
985  *           |  0   0   0   0   0   0   1   0  |
986  *           |  0   0   0   0   0   0   0   1  |
987  *           ~~                             ~~
988  *           __                             __
989  *           |  1   1   1   1   1   1   1   1  |
990  *           | 128  64  32  16  8   4   2   1  |
991  *           |  19 205 116  29  64  16  4   1  |
992  *           |  1   0   0   0   0   0   0   0  |
993  *           |  0   1   0   0   0   0   0   0  |
994  *  (V|I)' = |  0   0   1   0   0   0   0   0  |
995  *           |  0   0   0   1   0   0   0   0  |
996  *           |  0   0   0   0   1   0   0   0  |
997  *           |  0   0   0   0   0   1   0   0  |
998  *           |  0   0   0   0   0   0   1   0  |
999  *           |  0   0   0   0   0   0   0   1  |
1000 *           ~~                             ~~
1001 *
1002 * Here we employ Gauss-Jordan elimination to find the inverse of (V|I)'. We
1003 * have carefully chosen the seed values 1, 2, and 4 to ensure that this
1004 * matrix is not singular.
1005 * __                                                             __
1006 * |  1   1   1   1   1   1   1   1     1   0   0   0   0   0   0   0  |
1007 * |  19 205 116  29  64  16  4   1     0   1   0   0   0   0   0   0  |
1008 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0  |
1009 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0  |
1010 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0  |
1011 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0  |
1012 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0  |
1013 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1  |
1014 * ~~                                                             ~~
1015 * __                                                             __
1016 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0  |
1017 * |  1   1   1   1   1   1   1   1     1   0   0   0   0   0   0   0  |
1018 * |  19 205 116  29  64  16  4   1     0   1   0   0   0   0   0   0  |
1019 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0  |
1020 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0  |
1021 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0  |
1022 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0  |
1023 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1  |
1024 * ~~                                                             ~~
1025 * __                                                             __
1026 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0  |
1027 * |  0   1   1   0   0   0   0   0     1   0   1   1   1   1   1   1  |
1028 * |  0  205 116  0   0   0   0   0     0   1   19  29  64  16  4   1  |
1029 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0  |
1030 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0  |
1031 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0  |
1032 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0  |
1033 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1  |
1034 * ~~                                                             ~~
1035 * __                                                             __
1036 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0  |
1037 * |  0   1   1   0   0   0   0   0     1   0   1   1   1   1   1   1  |
1038 * |  0   0  185  0   0   0   0   0    205  1  222 208 141 221 201 204 |
1039 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0  |
1040 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0  |
1041 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0  |
1042 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0  |
```

```
 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1  |
 * ~~                                                                 ~~
 *                                                                    
 * __                                                                 __
 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0  |
 * |  0   1   1   0   0   0   0   0     1   0   1   1   1   1   1   1  |
 * |  0   0   1   0   0   0   0   0   166 100   4   40 158 168 216 209  |
 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0  |
 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0  |
 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0  |
 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0  |
 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1  |
 * ~~                                                                 ~~
 *                                                                    
 * __                                                                 __
 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0  |
 * |  0   1   0   0   0   0   0   0   167 100   5   41 159 169 217 208  |
 * |  0   0   1   0   0   0   0   0   166 100   4   40 158 168 216 209  |
 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0  |
 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0  |
 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0  |
 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0  |
 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1  |
 * ~~                                                                 ~~
 *                                                                    
 *                    __                                         __   
 *                    |  0   0   1   0   0   0   0   0  |
 *                    | 167 100  5   41 159 169 217 208 |
 *                    | 166 100  4   40 158 168 216 209 |
 *       (V|I)'^-1 =  |  0   0   0   1   0   0   0   0  |
 *                    |  0   0   0   0   1   0   0   0  |
 *                    |  0   0   0   0   0   1   0   0  |
 *                    |  0   0   0   0   0   0   1   0  |
 *                    |  0   0   0   0   0   0   0   1  |
 *                    ~~                                 ~~
 *
 * We can then simply compute D = (V|I)'^-1 x (d|p)' to discover the values
 * of the missing data.
 *
 * As is apparent from the example above, the only non-trivial rows in the
 * inverse matrix correspond to the data disks that we're trying to
 * reconstruct. Indeed, those are the only rows we need as the others would
 * only be useful for reconstructing data known or assumed to be valid. For
 * that reason, we only build the coefficients in the rows that correspond to
 * targeted columns.
 */
/* END CSTYLED */

static void
vdev_raidz_matrix_init(raidz_map_t *rm, int n, int nmap, int *map,
    uint8_t **rows)
{
        int i, j;
        int pow;

        ASSERT(n == rm->rm_cols - rm->rm_firstdatacol);

        /*
         * Fill in the missing rows of interest.
         */
        for (i = 0; i < nmap; i++) {
                ASSERT3S(0, <=, map[i]);
                ASSERT3S(map[i], <=, 2);

                pow = map[i] * n;
                if (pow > 255)
                        pow -= 255;
                ASSERT(pow <= 255);
```



```
                for (j = 0; j < n; j++) {
                        pow -= map[i];
                        if (pow < 0)
                                pow += 255;
                        rows[i][j] = vdev_raidz_pow2[pow];
                }
        }
}

static void
vdev_raidz_matrix_invert(raidz_map_t *rm, int n, int nmissing, int *missing,
    uint8_t **rows, uint8_t **invrows, const uint8_t *used)
{
        int i, j, ii, jj;
        uint8_t log;

        /*
         * Assert that the first nmissing entries from the array of used
         * columns correspond to parity columns and that subsequent entries
         * correspond to data columns.
         */
        for (i = 0; i < nmissing; i++) {
                ASSERT3S(used[i], <, rm->rm_firstdatacol);
        }
        for (; i < n; i++) {
                ASSERT3S(used[i], >=, rm->rm_firstdatacol);
        }

        /*
         * First initialize the storage where we'll compute the inverse rows.
         */
        for (i = 0; i < nmissing; i++) {
                for (j = 0; j < n; j++) {
                        invrows[i][j] = (i == j) ? 1 : 0;
                }
        }

        /*
         * Subtract all trivial rows from the rows of consequence.
         */
        for (i = 0; i < nmissing; i++) {
                for (j = nmissing; j < n; j++) {
                        ASSERT3U(used[j], >=, rm->rm_firstdatacol);
                        jj = used[j] - rm->rm_firstdatacol;
                        ASSERT3S(jj, <, n);
                        invrows[i][j] = rows[i][jj];
                        rows[i][jj] = 0;
                }
        }

        /*
         * For each of the rows of interest, we must normalize it and subtract
         * a multiple of it from the other rows.
         */
        for (i = 0; i < nmissing; i++) {
                for (j = 0; j < missing[i]; j++) {
                        ASSERT0(rows[i][j]);
                }
                ASSERT3U(rows[i][missing[i]], !=, 0);

                /*
                 * Compute the inverse of the first element and multiply each
                 * element in the row by that value.
                 */
                log = 255 - vdev_raidz_log2[rows[i][missing[i]]];
```

```
1175                 for (j = 0; j < n; j++) {
1176                         rows[i][j] = vdev_raidz_exp2(rows[i][j], log);
1177                         invrows[i][j] = vdev_raidz_exp2(invrows[i][j], log);
1178                 }
1179
1180                 for (ii = 0; ii < nmissing; ii++) {
1181                         if (i == ii)
1182                                 continue;
1183
1184                         ASSERT3U(rows[ii][missing[i]], !=, 0);
1185
1186                         log = vdev_raidz_log2[rows[ii][missing[i]]];
1187
1188                         for (j = 0; j < n; j++) {
1189                                 rows[ii][j] ^=
1190                                     vdev_raidz_exp2(rows[i][j], log);
1191                                 invrows[ii][j] ^=
1192                                     vdev_raidz_exp2(invrows[i][j], log);
1193                         }
1194                 }
1195         }
1196
1197         /*
1198          * Verify that the data that is left in the rows are properly part of
1199          * an identity matrix.
1200          */
1201         for (i = 0; i < nmissing; i++) {
1202                 for (j = 0; j < n; j++) {
1203                         if (j == missing[i]) {
1204                                 ASSERT3U(rows[i][j], ==, 1);
1205                         } else {
1206                                 ASSERT0(rows[i][j]);
1207                         }
1208                 }
1209         }
1210 }
1211
1212 static void
1213 vdev_raidz_matrix_reconstruct(raidz_map_t *rm, int n, int nmissing,
1214     int *missing, uint8_t **invrows, const uint8_t *used)
1215 {
1216         int i, j, x, cc, c;
1217         uint8_t *src;
1218         uint64_t ccount;
1219         uint8_t *dst[VDEV_RAIDZ_MAXPARITY];
1220         uint64_t dcount[VDEV_RAIDZ_MAXPARITY];
1221         uint8_t log = 0;
1222         uint8_t val;
1223         int ll;
1224         uint8_t *invlog[VDEV_RAIDZ_MAXPARITY];
1225         uint8_t *p, *pp;
1226         size_t psize;
1227
1228         psize = sizeof (invlog[0][0]) * n * nmissing;
1229         p = kmem_alloc(psize, KM_SLEEP);
1230
1231         for (pp = p, i = 0; i < nmissing; i++) {
1232                 invlog[i] = pp;
1233                 pp += n;
1234         }
1235
1236         for (i = 0; i < nmissing; i++) {
1237                 for (j = 0; j < n; j++) {
1238                         ASSERT3U(invrows[i][j], !=, 0);
1239                         invlog[i][j] = vdev_raidz_log2[invrows[i][j]];
1240                 }
```



```
1241         }
1242
1243         for (i = 0; i < n; i++) {
1244                 c = used[i];
1245                 ASSERT3U(c, <, rm->rm_cols);
1246
1247                 src = rm->rm_col[c].rc_data;
1248                 ccount = rm->rm_col[c].rc_size;
1249                 for (j = 0; j < nmissing; j++) {
1250                         cc = missing[j] + rm->rm_firstdatacol;
1251                         ASSERT3U(cc, >=, rm->rm_firstdatacol);
1252                         ASSERT3U(cc, <, rm->rm_cols);
1253                         ASSERT3U(cc, !=, c);
1254
1255                         dst[j] = rm->rm_col[cc].rc_data;
1256                         dcount[j] = rm->rm_col[cc].rc_size;
1257                 }
1258
1259                 ASSERT(ccount >= rm->rm_col[missing[0]].rc_size || i > 0);
1260
1261                 for (x = 0; x < ccount; x++, src++) {
1262                         if (*src != 0)
1263                                 log = vdev_raidz_log2[*src];
1264
1265                         for (cc = 0; cc < nmissing; cc++) {
1266                                 if (x >= dcount[cc])
1267                                         continue;
1268
1269                                 if (*src == 0) {
1270                                         val = 0;
1271                                 } else {
1272                                         if ((ll = log + invlog[cc][i]) >= 255)
1273                                                 ll -= 255;
1274                                         val = vdev_raidz_pow2[ll];
1275                                 }
1276
1277                                 if (i == 0)
1278                                         dst[cc][x] = val;
1279                                 else
1280                                         dst[cc][x] ^= val;
1281                         }
1282                 }
1283         }
1284
1285         kmem_free(p, psize);
1286 }
1287
1288 static int
1289 vdev_raidz_reconstruct_general(raidz_map_t *rm, int *tgts, int ntgts)
1290 {
1291         int n, i, c, t, tt;
1292         int nmissing_rows;
1293         int missing_rows[VDEV_RAIDZ_MAXPARITY];
1294         int parity_map[VDEV_RAIDZ_MAXPARITY];
1295
1296         uint8_t *p, *pp;
1297         size_t psize;
1298
1299         uint8_t *rows[VDEV_RAIDZ_MAXPARITY];
1300         uint8_t *invrows[VDEV_RAIDZ_MAXPARITY];
1301         uint8_t *used;
1302
1303         int code = 0;
1304
1305
1306         n = rm->rm_cols - rm->rm_firstdatacol;
```

```
1308         /*
1309          * Figure out which data columns are missing.
1310          */
1311         nmissing_rows = 0;
1312         for (t = 0; t < ntgts; t++) {
1313                 if (tgts[t] >= rm->rm_firstdatacol) {
1314                         missing_rows[nmissing_rows++] =
1315                             tgts[t] - rm->rm_firstdatacol;
1316                 }
1317         }

1319         /*
1320          * Figure out which parity columns to use to help generate the missing
1321          * data columns.
1322          */
1323         for (tt = 0, c = 0, i = 0; i < nmissing_rows; c++) {
1324                 ASSERT(tt < ntgts);
1325                 ASSERT(c < rm->rm_firstdatacol);

1327                 /*
1328                  * Skip any targeted parity columns.
1329                  */
1330                 if (c == tgts[tt]) {
1331                         tt++;
1332                         continue;
1333                 }

1335                 code |= 1 << c;

1337                 parity_map[i] = c;
1338                 i++;
1339         }

1341         ASSERT(code != 0);
1342         ASSERT3U(code, <, 1 << VDEV_RAIDZ_MAXPARITY);

1344         psize = (sizeof (rows[0][0]) + sizeof (invrows[0][0])) *
1345             nmissing_rows * n + sizeof (used[0]) * n;
1346         p = kmem_alloc(psize, KM_SLEEP);

1348         for (pp = p, i = 0; i < nmissing_rows; i++) {
1349                 rows[i] = pp;
1350                 pp += n;
1351                 invrows[i] = pp;
1352                 pp += n;
1353         }
1354         used = pp;

1356         for (i = 0; i < nmissing_rows; i++) {
1357                 used[i] = parity_map[i];
1358         }

1360         for (tt = 0, c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
1361                 if (tt < nmissing_rows &&
1362                     c == missing_rows[tt] + rm->rm_firstdatacol) {
1363                         tt++;
1364                         continue;
1365                 }

1367                 ASSERT3S(i, <, n);
1368                 used[i] = c;
1369                 i++;
1370         }

1372         /*
```



```
1373          * Initialize the interesting rows of the matrix.
1374          */
1375         vdev_raidz_matrix_init(rm, n, nmissing_rows, parity_map, rows);

1377         /*
1378          * Invert the matrix.
1379          */
1380         vdev_raidz_matrix_invert(rm, n, nmissing_rows, missing_rows, rows,
1381             invrows, used);

1383         /*
1384          * Reconstruct the missing data using the generated matrix.
1385          */
1386         vdev_raidz_matrix_reconstruct(rm, n, nmissing_rows, missing_rows,
1387             invrows, used);

1389         kmem_free(p, psize);

1391         return (code);
1392 }

1394 static int
1395 vdev_raidz_reconstruct(raidz_map_t *rm, int *t, int nt)
1396 {
1397         int tgts[VDEV_RAIDZ_MAXPARITY], *dt;
1398         int ntgts;
1399         int i, c;
1400         int code;
1401         int nbadparity, nbaddata;
1402         int parity_valid[VDEV_RAIDZ_MAXPARITY];

1404         /*
1405          * The tgts list must already be sorted.
1406          */
1407         for (i = 1; i < nt; i++) {
1408                 ASSERT(t[i] > t[i - 1]);
1409         }

1411         nbadparity = rm->rm_firstdatacol;
1412         nbaddata = rm->rm_cols - nbadparity;
1413         ntgts = 0;
1414         for (i = 0, c = 0; c < rm->rm_cols; c++) {
1415                 if (c < rm->rm_firstdatacol)
1416                         parity_valid[c] = B_FALSE;

1418                 if (i < nt && c == t[i]) {
1419                         tgts[ntgts++] = c;
1420                         i++;
1421                 } else if (rm->rm_col[c].rc_error != 0) {
1422                         tgts[ntgts++] = c;
1423                 } else if (c >= rm->rm_firstdatacol) {
1424                         nbaddata--;
1425                 } else {
1426                         parity_valid[c] = B_TRUE;
1427                         nbadparity--;
1428                 }
1429         }

1431         ASSERT(ntgts >= nt);
1432         ASSERT(nbaddata >= 0);
1433         ASSERT(nbaddata + nbadparity == ntgts);

1435         dt = &tgts[nbadparity];

1437         /*
1438          * See if we can use any of our optimized reconstruction routines.
```

```
1439          */
1440         if (!vdev_raidz_default_to_general) {
1441                 switch (nbaddata) {
1442                 case 1:
1443                         if (parity_valid[VDEV_RAIDZ_P])
1444                                 return (vdev_raidz_reconstruct_p(rm, dt, 1));

1446                         ASSERT(rm->rm_firstdatacol > 1);

1448                         if (parity_valid[VDEV_RAIDZ_Q])
1449                                 return (vdev_raidz_reconstruct_q(rm, dt, 1));

1451                         ASSERT(rm->rm_firstdatacol > 2);
1452                         break;

1454                 case 2:
1455                         ASSERT(rm->rm_firstdatacol > 1);

1457                         if (parity_valid[VDEV_RAIDZ_P] &&
1458                             parity_valid[VDEV_RAIDZ_Q])
1459                                 return (vdev_raidz_reconstruct_pq(rm, dt, 2));

1461                         ASSERT(rm->rm_firstdatacol > 2);

1463                         break;
1464                 }
1465         }

1467         code = vdev_raidz_reconstruct_general(rm, tgts, ntgts);
1468         ASSERT(code < (1 << VDEV_RAIDZ_MAXPARITY));
1469         ASSERT(code > 0);
1470         return (code);
1471 }

1473 static int
1474 vdev_raidz_open(vdev_t *vd, uint64_t *asize, uint64_t *max_asize,
1475     uint64_t *ashift)
1476 {
1477         vdev_t *cvd;
1478         uint64_t nparity = vd->vdev_nparity;
1479         int c;
1480         int lasterror = 0;
1481         int numerrors = 0;

1483         ASSERT(nparity > 0);

1485         if (nparity > VDEV_RAIDZ_MAXPARITY ||
1486             vd->vdev_children < nparity + 1) {
1487                 vd->vdev_stat.vs_aux = VDEV_AUX_BAD_LABEL;
1488                 return (SET_ERROR(EINVAL));
1489         }

1491         vdev_open_children(vd);

1493         for (c = 0; c < vd->vdev_children; c++) {
1494                 cvd = vd->vdev_child[c];

1496                 if (cvd->vdev_open_error != 0) {
1497                         lasterror = cvd->vdev_open_error;
1498                         numerrors++;
1499                         continue;
1500                 }

1502                 *asize = MIN(*asize - 1, cvd->vdev_asize - 1) + 1;
1503                 *max_asize = MIN(*max_asize - 1, cvd->vdev_max_asize - 1) + 1;
1504                 *ashift = MAX(*ashift, cvd->vdev_ashift);
```



```
1505         }

1507         *asize *= vd->vdev_children;
1508         *max_asize *= vd->vdev_children;

1510         if (numerrors > nparity) {
1511                 vd->vdev_stat.vs_aux = VDEV_AUX_NO_REPLICAS;
1512                 return (lasterror);
1513         }

1515         return (0);
1516 }

1518 static void
1519 vdev_raidz_close(vdev_t *vd)
1520 {
1521         int c;

1523         for (c = 0; c < vd->vdev_children; c++)
1524                 vdev_close(vd->vdev_child[c]);
1525 }

1527 static uint64_t
1528 vdev_raidz_asize(vdev_t *vd, uint64_t psize)
1529 {
1530         uint64_t asize;
1531         uint64_t ashift = vd->vdev_top->vdev_ashift;
1532         uint64_t cols = vd->vdev_children;
1533         uint64_t nparity = vd->vdev_nparity;

1535         asize = ((psize - 1) >> ashift) + 1;
1536         asize += nparity * ((asize + cols - nparity - 1) / (cols - nparity));
1537         asize = roundup(asize, nparity + 1) << ashift;

1539         return (asize);
1540 }

1542 static void
1543 vdev_raidz_child_done(zio_t *zio)
1544 {
1545         raidz_col_t *rc = zio->io_private;

1547         rc->rc_error = zio->io_error;
1548         rc->rc_tried = 1;
1549         rc->rc_skipped = 0;
1550 }

1552 /*
1553  * Start an IO operation on a RAIDZ VDev
1554  *
1555  * Outline:
1556  * - For write operations:
1557  *   1. Generate the parity data
1558  *   2. Create child zio write operations to each column's vdev, for both
1559  *      data and parity.
1560  *   3. If the column skips any sectors for padding, create optional dummy
1561  *      write zio children for those areas to improve aggregation continuity.
1562  * - For read operations:
1563  *   1. Create child zio read operations to each data column's vdev to read
1564  *      the range of data required for zio.
1565  *   2. If this is a scrub or resilver operation, or if any of the data
1566  *      vdevs have had errors, then create zio read operations to the parity
1567  *      columns' VDevs as well.
1568  */
1569 static int
1570 vdev_raidz_io_start(zio_t *zio)
```

```
1571 {
1572         vdev_t *vd = zio->io_vd;
1573         vdev_t *tvd = vd->vdev_top;
1574         vdev_t *cvd;
1575         raidz_map_t *rm;
1576         raidz_col_t *rc;
1577         int c, i;

1579         rm = vdev_raidz_map_alloc(zio, tvd->vdev_ashift, vd->vdev_children,
1580             vd->vdev_nparity);

1582         ASSERT3U(rm->rm_asize, ==, vdev_psize_to_asize(vd, zio->io_size));

1584         if (zio->io_type == ZIO_TYPE_WRITE) {
1585                 vdev_raidz_generate_parity(rm);

1587                 for (c = 0; c < rm->rm_cols; c++) {
1588                         rc = &rm->rm_col[c];
1589                         cvd = vd->vdev_child[rc->rc_devidx];
1590                         zio_nowait(zio_vdev_child_io(zio, NULL, cvd,
1591                             rc->rc_offset, rc->rc_data, rc->rc_size,
1592                             zio->io_type, zio->io_priority, 0,
1593                             vdev_raidz_child_done, rc));
1594                 }

1596                 /*
1597                  * Generate optional I/Os for any skipped sectors to improve
1598                  * aggregation contiguity.
1599                  */
1600                 for (c = rm->rm_skipstart, i = 0; i < rm->rm_nskip; c++, i++) {
1601                         ASSERT(c <= rm->rm_scols);
1602                         if (c == rm->rm_scols)
1603                                 c = 0;
1604                         rc = &rm->rm_col[c];
1605                         cvd = vd->vdev_child[rc->rc_devidx];
1606                         zio_nowait(zio_vdev_child_io(zio, NULL, cvd,
1607                             rc->rc_offset + rc->rc_size, NULL,
1608                             1 << tvd->vdev_ashift,
1609                             zio->io_type, zio->io_priority,
1610                             ZIO_FLAG_NODATA | ZIO_FLAG_OPTIONAL, NULL, NULL));
1611                 }

1613                 return (ZIO_PIPELINE_CONTINUE);
1614         }

1616         ASSERT(zio->io_type == ZIO_TYPE_READ);

1618         /*
1619          * Iterate over the columns in reverse order so that we hit the parity
1620          * last -- any errors along the way will force us to read the parity.
1621          */
1622         for (c = rm->rm_cols - 1; c >= 0; c--) {
1623                 rc = &rm->rm_col[c];
1624                 cvd = vd->vdev_child[rc->rc_devidx];
1625                 if (!vdev_readable(cvd)) {
1626                         if (c >= rm->rm_firstdatacol)
1627                                 rm->rm_missingdata++;
1628                         else
1629                                 rm->rm_missingparity++;
1630                         rc->rc_error = SET_ERROR(ENXIO);
1631                         rc->rc_tried = 1;       /* don't even try */
1632                         rc->rc_skipped = 1;
1633                         continue;
1634                 }
1635                 if (vdev_dtl_contains(cvd, DTL_MISSING, zio->io_txg, 1)) {
1636                         if (c >= rm->rm_firstdatacol)
```



```
1637                                 rm->rm_missingdata++;
1638                         else
1639                                 rm->rm_missingparity++;
1640                         rc->rc_error = SET_ERROR(ESTALE);
1641                         rc->rc_skipped = 1;
1642                         continue;
1643                 }
1644                 if (c >= rm->rm_firstdatacol || rm->rm_missingdata > 0 ||
1645                     (zio->io_flags & (ZIO_FLAG_SCRUB | ZIO_FLAG_RESILVER))) {
1646                         zio_nowait(zio_vdev_child_io(zio, NULL, cvd,
1647                             rc->rc_offset, rc->rc_data, rc->rc_size,
1648                             zio->io_type, zio->io_priority, 0,
1649                             vdev_raidz_child_done, rc));
1650                 }
1651         }

1653         return (ZIO_PIPELINE_CONTINUE);
1654 }


1657 /*
1658  * Report a checksum error for a child of a RAID-Z device.
1659  */
1660 static void
1661 raidz_checksum_error(zio_t *zio, raidz_col_t *rc, void *bad_data)
1662 {
1663         vdev_t *vd = zio->io_vd->vdev_child[rc->rc_devidx];

1665         if (!(zio->io_flags & ZIO_FLAG_SPECULATIVE)) {
1666                 zio_bad_cksum_t zbc;
1667                 raidz_map_t *rm = zio->io_vsd;

1669                 mutex_enter(&vd->vdev_stat_lock);
1670                 vd->vdev_stat.vs_checksum_errors++;
1671                 mutex_exit(&vd->vdev_stat_lock);

1673                 zbc.zbc_has_cksum = 0;
1674                 zbc.zbc_injected = rm->rm_ecksuminjected;

1676                 zfs_ereport_post_checksum(zio->io_spa, vd, zio,
1677                     rc->rc_offset, rc->rc_size, rc->rc_data, bad_data,
1678                     &zbc);
1679         }
1680 }

1682 /*
1683  * We keep track of whether or not there were any injected errors, so that
1684  * any ereports we generate can note it.
1685  */
1686 static int
1687 raidz_checksum_verify(zio_t *zio)
1688 {
1689         zio_bad_cksum_t zbc;
1690         raidz_map_t *rm = zio->io_vsd;

1692         int ret = zio_checksum_error(zio, &zbc);
1693         if (ret != 0 && zbc.zbc_injected != 0)
1694                 rm->rm_ecksuminjected = 1;

1696         return (ret);
1697 }

1699 /*
1700  * Generate the parity from the data columns. If we tried and were able to
1701  * read the parity without error, verify that the generated parity matches the
1702  * data we read. If it doesn't, we fire off a checksum error. Return the
```

```
1703  * number such failures.
1704  */
1705 static int
1706 raidz_parity_verify(zio_t *zio, raidz_map_t *rm)
1707 {
1708         void *orig[VDEV_RAIDZ_MAXPARITY];
1709         int c, ret = 0;
1710         raidz_col_t *rc;

1712         for (c = 0; c < rm->rm_firstdatacol; c++) {
1713                 rc = &rm->rm_col[c];
1714                 if (!rc->rc_tried || rc->rc_error != 0)
1715                         continue;
1716                 orig[c] = zio_buf_alloc(rc->rc_size);
1717                 bcopy(rc->rc_data, orig[c], rc->rc_size);
1718         }

1720         vdev_raidz_generate_parity(rm);

1722         for (c = 0; c < rm->rm_firstdatacol; c++) {
1723                 rc = &rm->rm_col[c];
1724                 if (!rc->rc_tried || rc->rc_error != 0)
1725                         continue;
1726                 if (bcmp(orig[c], rc->rc_data, rc->rc_size) != 0) {
1727                         raidz_checksum_error(zio, rc, orig[c]);
1728                         rc->rc_error = SET_ERROR(ECKSUM);
1729                         ret++;
1730                 }
1731                 zio_buf_free(orig[c], rc->rc_size);
1732         }

1734         return (ret);
1735 }

1737 /*
1738  * Keep statistics on all the ways that we used parity to correct data.
1739  */
1740 static uint64_t raidz_corrected[1 << VDEV_RAIDZ_MAXPARITY];

1742 static int
1743 vdev_raidz_worst_error(raidz_map_t *rm)
1744 {
1745         int error = 0;

1747         for (int c = 0; c < rm->rm_cols; c++)
1748                 error = zio_worst_error(error, rm->rm_col[c].rc_error);

1750         return (error);
1751 }

1753 /*
1754  * Iterate over all combinations of bad data and attempt a reconstruction.
1755  * Note that the algorithm below is non-optimal because it doesn't take into
1756  * account how reconstruction is actually performed. For example, with
1757  * triple-parity RAID-Z the reconstruction procedure is the same if column 4
1758  * is targeted as invalid as if columns 1 and 4 are targeted since in both
1759  * cases we'd only use parity information in column 0.
1760  */
1761 static int
1762 vdev_raidz_combrec(zio_t *zio, int total_errors, int data_errors)
1763 {
1764         raidz_map_t *rm = zio->io_vsd;
1765         raidz_col_t *rc;
1766         void *orig[VDEV_RAIDZ_MAXPARITY];
1767         int tstore[VDEV_RAIDZ_MAXPARITY + 2];
1768         int *tgts = &tstore[1];
```



```
1769         int current, next, i, c, n;
1770         int code, ret = 0;

1772         ASSERT(total_errors < rm->rm_firstdatacol);

1774         /*
1775          * This simplifies one edge condition.
1776          */
1777         tgts[-1] = -1;

1779         for (n = 1; n <= rm->rm_firstdatacol - total_errors; n++) {
1780                 /*
1781                  * Initialize the targets array by finding the first n columns
1782                  * that contain no error.
1783                  *
1784                  * If there were no data errors, we need to ensure that we're
1785                  * always explicitly attempting to reconstruct at least one
1786                  * data column. To do this, we simply push the highest target
1787                  * up into the data columns.
1788                  */
1789                 for (c = 0, i = 0; i < n; i++) {
1790                         if (i == n - 1 && data_errors == 0 &&
1791                             c < rm->rm_firstdatacol) {
1792                                 c = rm->rm_firstdatacol;
1793                         }

1795                         while (rm->rm_col[c].rc_error != 0) {
1796                                 c++;
1797                                 ASSERT3S(c, <, rm->rm_cols);
1798                         }

1800                         tgts[i] = c++;
1801                 }

1803                 /*
1804                  * Setting tgts[n] simplifies the other edge condition.
1805                  */
1806                 tgts[n] = rm->rm_cols;

1808                 /*
1809                  * These buffers were allocated in previous iterations.
1810                  */
1811                 for (i = 0; i < n - 1; i++) {
1812                         ASSERT(orig[i] != NULL);
1813                 }

1815                 orig[n - 1] = zio_buf_alloc(rm->rm_col[0].rc_size);

1817                 current = 0;
1818                 next = tgts[current];

1820                 while (current != n) {
1821                         tgts[current] = next;
1822                         current = 0;

1824                         /*
1825                          * Save off the original data that we're going to
1826                          * attempt to reconstruct.
1827                          */
1828                         for (i = 0; i < n; i++) {
1829                                 ASSERT(orig[i] != NULL);
1830                                 c = tgts[i];
1831                                 ASSERT3S(c, >=, 0);
1832                                 ASSERT3S(c, <, rm->rm_cols);
1833                                 rc = &rm->rm_col[c];
1834                                 bcopy(rc->rc_data, orig[i], rc->rc_size);
```

```
1835                        }
1836
1837                        /*
1838                         * Attempt a reconstruction and exit the outer loop on
1839                         * success.
1840                         */
1841                        code = vdev_raidz_reconstruct(rm, tgts, n);
1842                        if (raidz_checksum_verify(zio) == 0) {
1843                                atomic_inc_64(&raidz_corrected[code]);
1844
1845                                for (i = 0; i < n; i++) {
1846                                        c = tgts[i];
1847                                        rc = &rm->rm_col[c];
1848                                        ASSERT(rc->rc_error == 0);
1849                                        if (rc->rc_tried)
1850                                                raidz_checksum_error(zio, rc,
1851                                                    orig[i]);
1852                                        rc->rc_error = SET_ERROR(ECKSUM);
1853                                }
1854
1855                                ret = code;
1856                                goto done;
1857                        }
1858
1859                        /*
1860                         * Restore the original data.
1861                         */
1862                        for (i = 0; i < n; i++) {
1863                                c = tgts[i];
1864                                rc = &rm->rm_col[c];
1865                                bcopy(orig[i], rc->rc_data, rc->rc_size);
1866                        }
1867
1868                        do {
1869                                /*
1870                                 * Find the next valid column after the current
1871                                 * position..
1872                                 */
1873                                for (next = tgts[current] + 1;
1874                                    next < rm->rm_cols &&
1875                                    rm->rm_col[next].rc_error != 0; next++)
1876                                        continue;
1877
1878                                ASSERT(next <= tgts[current + 1]);
1879
1880                                /*
1881                                 * If that spot is available, we're done here.
1882                                 */
1883                                if (next != tgts[current + 1])
1884                                        break;
1885
1886                                /*
1887                                 * Otherwise, find the next valid column after
1888                                 * the previous position.
1889                                 */
1890                                for (c = tgts[current - 1] + 1;
1891                                    rm->rm_col[c].rc_error != 0; c++)
1892                                        continue;
1893
1894                                tgts[current] = c;
1895                                current++;
1896
1897                        } while (current != n);
1898                }
1899        }
1900        n--;
```



```
1901 done:
1902         for (i = 0; i < n; i++) {
1903                 zio_buf_free(orig[i], rm->rm_col[0].rc_size);
1904         }
1905
1906         return (ret);
1907 }
1908
1909 /*
1910  * Complete an IO operation on a RAIDZ VDev
1911  *
1912  * Outline:
1913  * - For write operations:
1914  *   1. Check for errors on the child IOs.
1915  *   2. Return, setting an error code if too few child VDevs were written
1916  *      to reconstruct the data later.  Note that partial writes are
1917  *      considered successful if they can be reconstructed at all.
1918  * - For read operations:
1919  *   1. Check for errors on the child IOs.
1920  *   2. If data errors occurred:
1921  *      a. Try to reassemble the data from the parity available.
1922  *      b. If we haven't yet read the parity drives, read them now.
1923  *      c. If all parity drives have been read but the data still doesn't
1924  *         reassemble with a correct checksum, then try combinatorial
1925  *         reconstruction.
1926  *      d. If that doesn't work, return an error.
1927  *   3. If there were unexpected errors or this is a resilver operation,
1928  *      rewrite the vdevs that had errors.
1929  */
1930 static void
1931 vdev_raidz_io_done(zio_t *zio)
1932 {
1933         vdev_t *vd = zio->io_vd;
1934         vdev_t *cvd;
1935         raidz_map_t *rm = zio->io_vsd;
1936         raidz_col_t *rc;
1937         int unexpected_errors = 0;
1938         int parity_errors = 0;
1939         int parity_untried = 0;
1940         int data_errors = 0;
1941         int total_errors = 0;
1942         int n, c;
1943         int tgts[VDEV_RAIDZ_MAXPARITY];
1944         int code;
1945
1946         ASSERT(zio->io_bp != NULL);  /* XXX need to add code to enforce this */
1947
1948         ASSERT(rm->rm_missingparity <= rm->rm_firstdatacol);
1949         ASSERT(rm->rm_missingdata <= rm->rm_cols - rm->rm_firstdatacol);
1950
1951         for (c = 0; c < rm->rm_cols; c++) {
1952                 rc = &rm->rm_col[c];
1953
1954                 if (rc->rc_error) {
1955                         ASSERT(rc->rc_error != ECKSUM); /* child has no bp */
1956
1957                         if (c < rm->rm_firstdatacol)
1958                                 parity_errors++;
1959                         else
1960                                 data_errors++;
1961
1962                         if (!rc->rc_skipped)
1963                                 unexpected_errors++;
1964
1965                         total_errors++;
1966                 } else if (c < rm->rm_firstdatacol && !rc->rc_tried) {
```

```
1967                        parity_untried++;
1968                }
1969        }

1971        if (zio->io_type == ZIO_TYPE_WRITE) {
1972                /*
1973                 * XXX -- for now, treat partial writes as a success.
1974                 * (If we couldn't write enough columns to reconstruct
1975                 * the data, the I/O failed.  Otherwise, good enough.)
1976                 *
1977                 * Now that we support write reallocation, it would be better
1978                 * to treat partial failure as real failure unless there are
1979                 * no non-degraded top-level vdevs left, and not update DTLs
1980                 * if we intend to reallocate.
1981                 */
1982                /* XXPOLICY */
1983                if (total_errors > rm->rm_firstdatacol)
1984                        zio->io_error = vdev_raidz_worst_error(rm);

1986                return;
1987        }

1989        ASSERT(zio->io_type == ZIO_TYPE_READ);
1990        /*
1991         * There are three potential phases for a read:
1992         *      1. produce valid data from the columns read
1993         *      2. read all disks and try again
1994         *      3. perform combinatorial reconstruction
1995         *
1996         * Each phase is progressively both more expensive and less likely to
1997         * occur. If we encounter more errors than we can repair or all phases
1998         * fail, we have no choice but to return an error.
1999         */

2001        /*
2002         * If the number of errors we saw was correctable -- less than or equal
2003         * to the number of parity disks read -- attempt to produce data that
2004         * has a valid checksum. Naturally, this case applies in the absence of
2005         * any errors.
2006         */
2007        if (total_errors <= rm->rm_firstdatacol - parity_untried) {
2008                if (data_errors == 0) {
2009                        if (raidz_checksum_verify(zio) == 0) {
2010                                /*
2011                                 * If we read parity information (unnecessarily
2012                                 * as it happens since no reconstruction was
2013                                 * needed) regenerate and verify the parity.
2014                                 * We also regenerate parity when resilvering
2015                                 * so we can write it out to the failed device
2016                                 * later.
2017                                 */
2018                                if (parity_errors + parity_untried <
2019                                    rm->rm_firstdatacol ||
2020                                    (zio->io_flags & ZIO_FLAG_RESILVER)) {
2021                                        n = raidz_parity_verify(zio, rm);
2022                                        unexpected_errors += n;
2023                                        ASSERT(parity_errors + n <=
2024                                            rm->rm_firstdatacol);
2025                                }
2026                                goto done;
2027                        }
2028                } else {
2029                        /*
2030                         * We either attempt to read all the parity columns or
2031                         * none of them. If we didn't try to read parity, we
2032                         * wouldn't be here in the correctable case. There must
```



```
2033                         * also have been fewer parity errors than parity
2034                         * columns or, again, we wouldn't be in this code path.
2035                         */
2036                        ASSERT(parity_untried == 0);
2037                        ASSERT(parity_errors < rm->rm_firstdatacol);

2039                        /*
2040                         * Identify the data columns that reported an error.
2041                         */
2042                        n = 0;
2043                        for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
2044                                rc = &rm->rm_col[c];
2045                                if (rc->rc_error != 0) {
2046                                        ASSERT(n < VDEV_RAIDZ_MAXPARITY);
2047                                        tgts[n++] = c;
2048                                }
2049                        }

2051                        ASSERT(rm->rm_firstdatacol >= n);

2053                        code = vdev_raidz_reconstruct(rm, tgts, n);

2055                        if (raidz_checksum_verify(zio) == 0) {
2056                                atomic_inc_64(&raidz_corrected[code]);

2058                                /*
2059                                 * If we read more parity disks than were used
2060                                 * for reconstruction, confirm that the other
2061                                 * parity disks produced correct data. This
2062                                 * routine is suboptimal in that it regenerates
2063                                 * the parity that we already used in addition
2064                                 * to the parity that we're attempting to
2065                                 * verify, but this should be a relatively
2066                                 * uncommon case, and can be optimized if it
2067                                 * becomes a problem. Note that we regenerate
2068                                 * parity when resilvering so we can write it
2069                                 * out to failed devices later.
2070                                 */
2071                                if (parity_errors < rm->rm_firstdatacol - n ||
2072                                    (zio->io_flags & ZIO_FLAG_RESILVER)) {
2073                                        n = raidz_parity_verify(zio, rm);
2074                                        unexpected_errors += n;
2075                                        ASSERT(parity_errors + n <=
2076                                            rm->rm_firstdatacol);
2077                                }

2079                                goto done;
2080                        }
2081                }
2082        }

2084        /*
2085         * This isn't a typical situation -- either we got a read error or
2086         * a child silently returned bad data. Read every block so we can
2087         * try again with as much data and parity as we can track down. If
2088         * we've already been through once before, all children will be marked
2089         * as tried so we'll proceed to combinatorial reconstruction.
2090         */
2091        unexpected_errors = 1;
2092        rm->rm_missingdata = 0;
2093        rm->rm_missingparity = 0;

2095        for (c = 0; c < rm->rm_cols; c++) {
2096                if (rm->rm_col[c].rc_tried)
2097                        continue;
```

```
2099                zio_vdev_io_redone(zio);
2100                do {
2101                        rc = &rm->rm_col[c];
2102                        if (rc->rc_tried)
2103                                continue;
2104                        zio_nowait(zio_vdev_child_io(zio, NULL,
2105                            vd->vdev_child[rc->rc_devidx],
2106                            rc->rc_offset, rc->rc_data, rc->rc_size,
2107                            zio->io_type, zio->io_priority, 0,
2108                            vdev_raidz_child_done, rc));
2109                } while (++c < rm->rm_cols);
2110
2111                return;
2112        }
2113
2114        /*
2115         * At this point we've attempted to reconstruct the data given the
2116         * errors we detected, and we've attempted to read all columns. There
2117         * must, therefore, be one or more additional problems -- silent errors
2118         * resulting in invalid data rather than explicit I/O errors resulting
2119         * in absent data. We check if there is enough additional data to
2120         * possibly reconstruct the data and then perform combinatorial
2121         * reconstruction over all possible combinations. If that fails,
2122         * we're cooked.
2123         */
2124        if (total_errors > rm->rm_firstdatacol) {
2125                zio->io_error = vdev_raidz_worst_error(rm);
2126
2127        } else if (total_errors < rm->rm_firstdatacol &&
2128            (code = vdev_raidz_combrec(zio, total_errors, data_errors)) != 0) {
2129                /*
2130                 * If we didn't use all the available parity for the
2131                 * combinatorial reconstruction, verify that the remaining
2132                 * parity is correct.
2133                 */
2134                if (code != (1 << rm->rm_firstdatacol) - 1)
2135                        (void) raidz_parity_verify(zio, rm);
2136        } else {
2137                /*
2138                 * We're here because either:
2139                 *
2140                 *      total_errors == rm_first_datacol, or
2141                 *      vdev_raidz_combrec() failed
2142                 *
2143                 * In either case, there is enough bad data to prevent
2144                 * reconstruction.
2145                 *
2146                 * Start checksum ereports for all children which haven't
2147                 * failed, and the IO wasn't speculative.
2148                 */
2149                zio->io_error = SET_ERROR(ECKSUM);
2150
2151                if (!(zio->io_flags & ZIO_FLAG_SPECULATIVE)) {
2152                        for (c = 0; c < rm->rm_cols; c++) {
2153                                rc = &rm->rm_col[c];
2154                                if (rc->rc_error == 0) {
2155                                        zio_bad_cksum_t zbc;
2156                                        zbc.zbc_has_cksum = 0;
2157                                        zbc.zbc_injected =
2158                                            rm->rm_ecksuminjected;
2159
2160                                        zfs_ereport_start_checksum(
2161                                            zio->io_spa,
2162                                            vd->vdev_child[rc->rc_devidx],
2163                                            zio, rc->rc_offset, rc->rc_size,
2164                                            (void *)(uintptr_t)c, &zbc);
```



```
2165                                }
2166                        }
2167                }
2168        }
2169
2170 done:
2171        zio_checksum_verified(zio);
2172
2173        if (zio->io_error == 0 && spa_writeable(zio->io_spa) &&
2174            (unexpected_errors || (zio->io_flags & ZIO_FLAG_RESILVER))) {
2175                /*
2176                 * Use the good data we have in hand to repair damaged children.
2177                 */
2178                for (c = 0; c < rm->rm_cols; c++) {
2179                        rc = &rm->rm_col[c];
2180                        cvd = vd->vdev_child[rc->rc_devidx];
2181
2182                        if (rc->rc_error == 0)
2183                                continue;
2184
2185                        zio_nowait(zio_vdev_child_io(zio, NULL, cvd,
2186                            rc->rc_offset, rc->rc_data, rc->rc_size,
2187                            ZIO_TYPE_WRITE, zio->io_priority,
2188                            ZIO_FLAG_IO_REPAIR | (unexpected_errors ?
2189                            ZIO_FLAG_SELF_HEAL : 0), NULL, NULL));
2190                }
2191        }
2192 }
2193
2194 static void
2195 vdev_raidz_state_change(vdev_t *vd, int faulted, int degraded)
2196 {
2197        if (faulted > vd->vdev_nparity)
2198                vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
2199                    VDEV_AUX_NO_REPLICAS);
2200        else if (degraded + faulted != 0)
2201                vdev_set_state(vd, B_FALSE, VDEV_STATE_DEGRADED, VDEV_AUX_NONE);
2202        else
2203                vdev_set_state(vd, B_FALSE, VDEV_STATE_HEALTHY, VDEV_AUX_NONE);
2204 }
2205
2206 vdev_ops_t vdev_raidz_ops = {
2207        vdev_raidz_open,
2208        vdev_raidz_close,
2209        vdev_raidz_asize,
2210        vdev_raidz_io_start,
2211        vdev_raidz_io_done,
2212        vdev_raidz_state_change,
2213        NULL,
2214        NULL,
2215        VDEV_TYPE_RAIDZ,        /* name of this vdev type */
2216        B_FALSE                 /* not a leaf vdev */
2217 };
```

```
**********************************************************
   67493 Thu May 16 17:36:24 2013
new/usr/src/uts/common/fs/zfs/zfs_acl.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1335 static void
1336 zfs_acl_chmod(vtype_t vtype, uint64_t mode, boolean_t trim, zfs_acl_t *aclp)
1337 {
1338         void            *acep = NULL;
1339         uint64_t        who;
1340         int             new_count, new_bytes;
1341         int             ace_size;
1342         int             entry_type;
1343         uint16_t        iflags, type;
1344         uint32_t        access_mask;
1345         zfs_acl_node_t  *newnode;
1346         size_t          abstract_size = aclp->z_ops.ace_abstract_size();
1347         void            *zacep;
1348         boolean_t       isdir;
1349         trivial_acl_t   masks;

1351         new_count = new_bytes = 0;

1353         isdir = (vtype == VDIR);

1355         acl_trivial_access_masks((mode_t)mode, isdir, &masks);

1357         newnode = zfs_acl_node_alloc((abstract_size * 6) + aclp->z_acl_bytes);

1359         zacep = newnode->z_acldata;
1360         if (masks.allow0) {
1361                 zfs_set_ace(aclp, zacep, masks.allow0, ALLOW, -1, ACE_OWNER);
1362                 zacep = (void *)((uintptr_t)zacep + abstract_size);
1363                 new_count++;
1364                 new_bytes += abstract_size;
1365         }
1366         if (masks.deny1) {
1365         } if (masks.deny1) {
1367                 zfs_set_ace(aclp, zacep, masks.deny1, DENY, -1, ACE_OWNER);
1368                 zacep = (void *)((uintptr_t)zacep + abstract_size);
1369                 new_count++;
1370                 new_bytes += abstract_size;
1371         }
1372         if (masks.deny2) {
1373                 zfs_set_ace(aclp, zacep, masks.deny2, DENY, -1, OWNING_GROUP);
1374                 zacep = (void *)((uintptr_t)zacep + abstract_size);
1375                 new_count++;
1376                 new_bytes += abstract_size;
1377         }

1379         while (acep = zfs_acl_next_ace(aclp, acep, &who, &access_mask,
1380             &iflags, &type)) {
1381                 uint16_t inherit_flags;

1383                 entry_type = (iflags & ACE_TYPE_FLAGS);
1384                 inherit_flags = (iflags & ALL_INHERIT);

1386                 if ((entry_type == ACE_OWNER || entry_type == ACE_EVERYONE ||
1387                     (entry_type == OWNING_GROUP)) &&
```



```
1388                     ((inherit_flags & ACE_INHERIT_ONLY_ACE) == 0)) {
1389                         continue;
1390                 }

1392                 /*
1393                  * If this ACL has any inheritable ACEs, mark that in
1394                  * the hints (which are later masked into the pflags)
1395                  * so create knows to do inheritance.
1396                  */
1397                 if (isdir && (inherit_flags &
1398                     (ACE_FILE_INHERIT_ACE|ACE_DIRECTORY_INHERIT_ACE)))
1399                         aclp->z_hints |= ZFS_INHERIT_ACE;

1401                 if ((type != ALLOW && type != DENY) ||
1402                     (inherit_flags & ACE_INHERIT_ONLY_ACE)) {
1403                         switch (type) {
1404                         case ACE_ACCESS_ALLOWED_OBJECT_ACE_TYPE:
1405                         case ACE_ACCESS_DENIED_OBJECT_ACE_TYPE:
1406                         case ACE_SYSTEM_AUDIT_OBJECT_ACE_TYPE:
1407                         case ACE_SYSTEM_ALARM_OBJECT_ACE_TYPE:
1408                                 aclp->z_hints |= ZFS_ACL_OBJ_ACE;
1409                                 break;
1410                         }
1411                 } else {

1413                         /*
1414                          * Limit permissions to be no greater than
1415                          * group permissions.
1416                          * The "aclinherit" and "aclmode" properties
1417                          * affect policy for create and chmod(2),
1418                          * respectively.
1419                          */
1420                         if ((type == ALLOW) && trim)
1421                                 access_mask &= masks.group;
1422                 }
1423                 zfs_set_ace(aclp, zacep, access_mask, type, who, iflags);
1424                 ace_size = aclp->z_ops.ace_size(acep);
1425                 zacep = (void *)((uintptr_t)zacep + ace_size);
1426                 new_count++;
1427                 new_bytes += ace_size;
1428         }
1429         zfs_set_ace(aclp, zacep, masks.owner, 0, -1, ACE_OWNER);
1430         zacep = (void *)((uintptr_t)zacep + abstract_size);
1431         zfs_set_ace(aclp, zacep, masks.group, 0, -1, OWNING_GROUP);
1432         zacep = (void *)((uintptr_t)zacep + abstract_size);
1433         zfs_set_ace(aclp, zacep, masks.everyone, 0, -1, ACE_EVERYONE);

1435         new_count += 3;
1436         new_bytes += abstract_size * 3;
1437         zfs_acl_release_nodes(aclp);
1438         aclp->z_acl_count = new_count;
1439         aclp->z_acl_bytes = new_bytes;
1440         newnode->z_ace_count = new_count;
1441         newnode->z_size = new_bytes;
1442         list_insert_tail(&aclp->z_acl, newnode);
1443 }
______unchanged_portion_omitted_

1769 /*
1770  * Retrieve a file's ACL
1769  * Retrieve a files ACL
1771  */
1772 int
1773 zfs_getacl(znode_t *zp, vsecattr_t *vsecp, boolean_t skipaclchk, cred_t *cr)
1774 {
1775         zfs_acl_t       *aclp;
```

```
1776         ulong_t         mask;
1777         int             error;
1778         int             count = 0;
1779         int             largeace = 0;

1781         mask = vsecp->vsa_mask & (VSA_ACE | VSA_ACECNT |
1782             VSA_ACE_ACLFLAGS | VSA_ACE_ALLTYPES);

1784         if (mask == 0)
1785                 return (SET_ERROR(ENOSYS));

1787         if (error = zfs_zaccess(zp, ACE_READ_ACL, 0, skipaclchk, cr))
1788                 return (error);

1790         mutex_enter(&zp->z_acl_lock);

1792         error = zfs_acl_node_read(zp, B_FALSE, &aclp, B_FALSE);
1793         if (error != 0) {
1794                 mutex_exit(&zp->z_acl_lock);
1795                 return (error);
1796         }

1798         /*
1799          * Scan ACL to determine number of ACEs
1800          */
1801         if ((zp->z_pflags & ZFS_ACL_OBJ_ACE) && !(mask & VSA_ACE_ALLTYPES)) {
1802                 void *zacep = NULL;
1803                 uint64_t who;
1804                 uint32_t access_mask;
1805                 uint16_t type, iflags;

1807                 while (zacep = zfs_acl_next_ace(aclp, zacep,
1808                     &who, &access_mask, &iflags, &type)) {
1809                         switch (type) {
1810                         case ACE_ACCESS_ALLOWED_OBJECT_ACE_TYPE:
1811                         case ACE_ACCESS_DENIED_OBJECT_ACE_TYPE:
1812                         case ACE_SYSTEM_AUDIT_OBJECT_ACE_TYPE:
1813                         case ACE_SYSTEM_ALARM_OBJECT_ACE_TYPE:
1814                                 largeace++;
1815                                 continue;
1816                         default:
1817                                 count++;
1818                         }
1819                 }
1820                 vsecp->vsa_aclcnt = count;
1821         } else
1822                 count = (int)aclp->z_acl_count;

1824         if (mask & VSA_ACECNT) {
1825                 vsecp->vsa_aclcnt = count;
1826         }

1828         if (mask & VSA_ACE) {
1829                 size_t aclsz;

1831                 aclsz = count * sizeof (ace_t) +
1832                     sizeof (ace_object_t) * largeace;

1834                 vsecp->vsa_aclentp = kmem_alloc(aclsz, KM_SLEEP);
1835                 vsecp->vsa_aclentsz = aclsz;

1837                 if (aclp->z_version == ZFS_ACL_VERSION_FUID)
1838                         zfs_copy_fuid_2_ace(zp->z_zfsvfs, aclp, cr,
1839                             vsecp->vsa_aclentp, !(mask & VSA_ACE_ALLTYPES));
1840                 else {
1841                         zfs_acl_node_t *aclnode;
```



```
1842                         void *start = vsecp->vsa_aclentp;

1844                         for (aclnode = list_head(&aclp->z_acl); aclnode;
1845                             aclnode = list_next(&aclp->z_acl, aclnode)) {
1846                                 bcopy(aclnode->z_acldata, start,
1847                                     aclnode->z_size);
1848                                 start = (caddr_t)start + aclnode->z_size;
1849                         }
1850                         ASSERT((caddr_t)start - (caddr_t)vsecp->vsa_aclentp ==
1851                             aclp->z_acl_bytes);
1852                 }
1853         }
1854         if (mask & VSA_ACE_ACLFLAGS) {
1855                 vsecp->vsa_aclflags = 0;
1856                 if (zp->z_pflags & ZFS_ACL_DEFAULTED)
1857                         vsecp->vsa_aclflags |= ACL_DEFAULTED;
1858                 if (zp->z_pflags & ZFS_ACL_PROTECTED)
1859                         vsecp->vsa_aclflags |= ACL_PROTECTED;
1860                 if (zp->z_pflags & ZFS_ACL_AUTO_INHERIT)
1861                         vsecp->vsa_aclflags |= ACL_AUTO_INHERIT;
1862         }

1864         mutex_exit(&zp->z_acl_lock);

1866         return (0);
1867 }
```

_____***unchanged_portion_omitted_***

```
1924 /*
1925  * Set a file's ACL
1924  * Set a files ACL
1926  */
1927 int
1928 zfs_setacl(znode_t *zp, vsecattr_t *vsecp, boolean_t skipaclchk, cred_t *cr)
1929 {
1930         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
1931         zilog_t         *zilog = zfsvfs->z_log;
1932         ulong_t         mask = vsecp->vsa_mask & (VSA_ACE | VSA_ACECNT);
1933         dmu_tx_t        *tx;
1934         int             error;
1935         zfs_acl_t       *aclp;
1936         zfs_fuid_info_t *fuidp = NULL;
1937         boolean_t       fuid_dirtied;
1938         uint64_t        acl_obj;

1940         if (mask == 0)
1941                 return (SET_ERROR(ENOSYS));

1943         if (zp->z_pflags & ZFS_IMMUTABLE)
1944                 return (SET_ERROR(EPERM));

1946         if (error = zfs_zaccess(zp, ACE_WRITE_ACL, 0, skipaclchk, cr))
1947                 return (error);

1949         error = zfs_vsec_2_aclp(zfsvfs, ZTOV(zp)->v_type, vsecp, cr, &fuidp,
1950             &aclp);
1951         if (error)
1952                 return (error);

1954         /*
1955          * If ACL wide flags aren't being set then preserve any
1956          * existing flags.
1957          */
1958         if (!(vsecp->vsa_mask & VSA_ACE_ACLFLAGS)) {
1959                 aclp->z_hints |=
1960                     (zp->z_pflags & V4_ACL_WIDE_FLAGS);
```

```
1961         }
1962 top:
1963         mutex_enter(&zp->z_acl_lock);
1964         mutex_enter(&zp->z_lock);

1966         tx = dmu_tx_create(zfsvfs->z_os);

1968         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);

1970         fuid_dirtied = zfsvfs->z_fuid_dirty;
1971         if (fuid_dirtied)
1972                 zfs_fuid_txhold(zfsvfs, tx);

1974         /*
1975          * If old version and ACL won't fit in bonus and we aren't
1976          * upgrading then take out necessary DMU holds
1977          */

1979         if ((acl_obj = zfs_external_acl(zp)) != 0) {
1980                 if (zfsvfs->z_version >= ZPL_VERSION_FUID &&
1981                     zfs_znode_acl_version(zp) <= ZFS_ACL_VERSION_INITIAL) {
1982                         dmu_tx_hold_free(tx, acl_obj, 0,
1983                             DMU_OBJECT_END);
1984                         dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0,
1985                             aclp->z_acl_bytes);
1986                 } else {
1987                         dmu_tx_hold_write(tx, acl_obj, 0, aclp->z_acl_bytes);
1988                 }
1989         } else if (!zp->z_is_sa && aclp->z_acl_bytes > ZFS_ACE_SPACE) {
1990                 dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0, aclp->z_acl_bytes);
1991         }

1993         zfs_sa_upgrade_txholds(tx, zp);
1994         error = dmu_tx_assign(tx, TXG_NOWAIT);
1995         if (error) {
1996                 mutex_exit(&zp->z_acl_lock);
1997                 mutex_exit(&zp->z_lock);

1999                 if (error == ERESTART) {
2000                         dmu_tx_wait(tx);
2001                         dmu_tx_abort(tx);
2002                         goto top;
2003                 }
2004                 dmu_tx_abort(tx);
2005                 zfs_acl_free(aclp);
2006                 return (error);
2007         }

2009         error = zfs_aclset_common(zp, aclp, cr, tx);
2010         ASSERT(error == 0);
2011         ASSERT(zp->z_acl_cached == NULL);
2012         zp->z_acl_cached = aclp;

2014         if (fuid_dirtied)
2015                 zfs_fuid_sync(zfsvfs, tx);

2017         zfs_log_acl(zilog, tx, zp, vsecp, fuidp);

2019         if (fuidp)
2020                 zfs_fuid_info_free(fuidp);
2021         dmu_tx_commit(tx);
2022 done:
2023         mutex_exit(&zp->z_lock);
2024         mutex_exit(&zp->z_acl_lock);

2026         return (error);
```



```
2027 }
```

*unchanged_portion_omitted_*

```
2344 /*
2345  * Determine whether Access should be granted/denied.
2346  *
2347 #endif /* ! codereview */
2348  * The least priv subsytem is always consulted as a basic privilege
2349  * can define any form of access.
2350  */
2351 int
2352 zfs_zaccess(znode_t *zp, int mode, int flags, boolean_t skipaclchk, cred_t *cr)
2353 {
2354         uint32_t        working_mode;
2355         int             error;
2356         int             is_attr;
2357         boolean_t       check_privs;
2358         znode_t         *xzp;
2359         znode_t         *check_zp = zp;
2360         mode_t          needed_bits;
2361         uid_t           owner;

2363         is_attr = ((zp->z_pflags & ZFS_XATTR) && (ZTOV(zp)->v_type == VDIR));

2365         /*
2366          * If attribute then validate against base file
2367          */
2368         if (is_attr) {
2369                 uint64_t        parent;

2371                 if ((error = sa_lookup(zp->z_sa_hdl,
2372                     SA_ZPL_PARENT(zp->z_zfsvfs), &parent,
2373                     sizeof (parent))) != 0)
2374                         return (error);

2376                 if ((error = zfs_zget(zp->z_zfsvfs,
2377                     parent, &xzp)) != 0)        {
2378                         return (error);
2379                 }

2381                 check_zp = xzp;

2383                 /*
2384                  * fixup mode to map to xattr perms
2385                  */

2387                 if (mode & (ACE_WRITE_DATA|ACE_APPEND_DATA)) {
2388                         mode &= ~(ACE_WRITE_DATA|ACE_APPEND_DATA);
2389                         mode |= ACE_WRITE_NAMED_ATTRS;
2390                 }

2392                 if (mode & (ACE_READ_DATA|ACE_EXECUTE)) {
2393                         mode &= ~(ACE_READ_DATA|ACE_EXECUTE);
2394                         mode |= ACE_READ_NAMED_ATTRS;
2395                 }
2396         }

2398         owner = zfs_fuid_map_id(zp->z_zfsvfs, zp->z_uid, cr, ZFS_OWNER);
2399         /*
2400          * Map the bits required to the standard vnode flags VREAD|VWRITE|VEXEC
2401          * in needed_bits.  Map the bits mapped by working_mode (currently
2402          * missing) in missing_bits.
2403          * Call secpolicy_vnode_access2() with (needed_bits & ~checkmode),
2404          * needed_bits.
2405          */
2406         needed_bits = 0;
```

```
2408         working_mode = mode;
2409         if ((working_mode & (ACE_READ_ACL|ACE_READ_ATTRIBUTES)) &&
2410             owner == crgetuid(cr))
2411                 working_mode &= ~(ACE_READ_ACL|ACE_READ_ATTRIBUTES);

2413         if (working_mode & (ACE_READ_DATA|ACE_READ_NAMED_ATTRS|
2414             ACE_READ_ACL|ACE_READ_ATTRIBUTES|ACE_SYNCHRONIZE))
2415                 needed_bits |= VREAD;
2416         if (working_mode & (ACE_WRITE_DATA|ACE_WRITE_NAMED_ATTRS|
2417             ACE_APPEND_DATA|ACE_WRITE_ATTRIBUTES|ACE_SYNCHRONIZE))
2418                 needed_bits |= VWRITE;
2419         if (working_mode & ACE_EXECUTE)
2420                 needed_bits |= VEXEC;

2422         if ((error = zfs_zaccess_common(check_zp, mode, &working_mode,
2423             &check_privs, skipaclchk, cr)) == 0) {
2424                 if (is_attr)
2425                         VN_RELE(ZTOV(xzp));
2426                 return (secpolicy_vnode_access2(cr, ZTOV(zp), owner,
2427                     needed_bits, needed_bits));
2428         }

2430         if (error && !check_privs) {
2431                 if (is_attr)
2432                         VN_RELE(ZTOV(xzp));
2433                 return (error);
2434         }

2436         if (error && (flags & V_APPEND)) {
2437                 error = zfs_zaccess_append(zp, &working_mode, &check_privs, cr);
2438         }

2440         if (error && check_privs) {
2441                 mode_t          checkmode = 0;

2443                 /*
2444                  * First check for implicit owner permission on
2445                  * read_acl/read_attributes
2446                  */

2448                 error = 0;
2449                 ASSERT(working_mode != 0);

2451                 if ((working_mode & (ACE_READ_ACL|ACE_READ_ATTRIBUTES) &&
2452                     owner == crgetuid(cr)))
2453                         working_mode &= ~(ACE_READ_ACL|ACE_READ_ATTRIBUTES);

2455                 if (working_mode & (ACE_READ_DATA|ACE_READ_NAMED_ATTRS|
2456                     ACE_READ_ACL|ACE_READ_ATTRIBUTES|ACE_SYNCHRONIZE))
2457                         checkmode |= VREAD;
2458                 if (working_mode & (ACE_WRITE_DATA|ACE_WRITE_NAMED_ATTRS|
2459                     ACE_APPEND_DATA|ACE_WRITE_ATTRIBUTES|ACE_SYNCHRONIZE))
2460                         checkmode |= VWRITE;
2461                 if (working_mode & ACE_EXECUTE)
2462                         checkmode |= VEXEC;

2464                 error = secpolicy_vnode_access2(cr, ZTOV(check_zp), owner,
2465                     needed_bits & ~checkmode, needed_bits);

2467                 if (error == 0 && (working_mode & ACE_WRITE_OWNER))
2468                         error = secpolicy_vnode_chown(cr, owner);
2469                 if (error == 0 && (working_mode & ACE_WRITE_ACL))
2470                         error = secpolicy_vnode_setdac(cr, owner);

2472                 if (error == 0 && (working_mode &
```



```
2473                     (ACE_DELETE|ACE_DELETE_CHILD)))
2474                         error = secpolicy_vnode_remove(cr);

2476                 if (error == 0 && (working_mode & ACE_SYNCHRONIZE)) {
2477                         error = secpolicy_vnode_chown(cr, owner);
2478                 }
2479                 if (error == 0) {
2480                         /*
2481                          * See if any bits other than those already checked
2482                          * for are still present.  If so then return EACCES
2483                          */
2484                         if (working_mode & ~(ZFS_CHECKED_MASKS)) {
2485                                 error = SET_ERROR(EACCES);
2486                         }
2487                 }
2488         } else if (error == 0) {
2489                 error = secpolicy_vnode_access2(cr, ZTOV(zp), owner,
2490                     needed_bits, needed_bits);
2491         }


2494         if (is_attr)
2495                 VN_RELE(ZTOV(xzp));

2497         return (error);
2498 }

2500 /*
2501  * Translate traditional unix VREAD/VWRITE/VEXEC mode into
2502  * native ACL format and call zfs_zaccess()
2503  */
2504 int
2505 zfs_zaccess_rwx(znode_t *zp, mode_t mode, int flags, cred_t *cr)
2506 {
2507         return (zfs_zaccess(zp, zfs_unix_to_v4(mode >> 6), flags, B_FALSE, cr));
2508 }

2510 /*
2511  * Access function for secpolicy_vnode_setattr
2512  */
2513 int
2514 zfs_zaccess_unix(znode_t *zp, mode_t mode, cred_t *cr)
2515 {
2516         int v4_mode = zfs_unix_to_v4(mode >> 6);

2518         return (zfs_zaccess(zp, v4_mode, 0, B_FALSE, cr));
2519 }

2521 static int
2522 zfs_delete_final_check(znode_t *zp, znode_t *dzp,
2523     mode_t available_perms, cred_t *cr)
2524 {
2525         int error;
2526         uid_t downer;

2528         downer = zfs_fuid_map_id(dzp->z_zfsvfs, dzp->z_uid, cr, ZFS_OWNER);

2530         error = secpolicy_vnode_access2(cr, ZTOV(dzp),
2531             downer, available_perms, VWRITE|VEXEC);

2533         if (error == 0)
2534                 error = zfs_sticky_remove_access(dzp, zp, cr);

2536         return (error);
2537 }
```

```
2539 /*
2540  * Determine whether Access should be granted/deny, without
2541  * consulting least priv subsystem.
2542  *
2345  *
2543  * The following chart is the recommended NFSv4 enforcement for
2544  * ability to delete an object.
2545  *
2546  *      -------------------------------------------------------
2547  *      |   Parent Dir  |           Target Object Permissions |
2548  *      |  permissions  |                                     |
2549  *      -------------------------------------------------------
2550  *      |               | ACL Allows | ACL Denies| Delete     |
2551  *      |               |  Delete    |  Delete   | unspecified|
2552  *      -------------------------------------------------------
2553  *      |  ACL Allows   | Permit     | Permit    | Permit     |
2554  *      |  DELETE_CHILD |                                     |
2555  *      -------------------------------------------------------
2556  *      |  ACL Denies   | Permit     | Deny      | Deny       |
2557  *      |  DELETE_CHILD |            |           |            |
2558  *      -------------------------------------------------------
2559  *      | ACL specifies |            |           |            |
2560  *      | only allow    | Permit     | Permit    | Permit     |
2561  *      | write and     |            |           |            |
2562  *      | execute       |            |           |            |
2563  *      -------------------------------------------------------
2564  *      | ACL denies    |            |           |            |
2565  *      | write and     | Permit     | Deny      | Deny       |
2566  *      | execute       |            |           |            |
2567  *      -------------------------------------------------------
2568  *         ^
2569  *         |
2570  *         No search privilege, can't even look up file?
2571  *
2572  */
2573 int
2574 zfs_zaccess_delete(znode_t *dzp, znode_t *zp, cred_t *cr)
2575 {
2576         uint32_t dzp_working_mode = 0;
2577         uint32_t zp_working_mode = 0;
2578         int dzp_error, zp_error;
2579         mode_t available_perms;
2580         boolean_t dzpcheck_privs = B_TRUE;
2581         boolean_t zpcheck_privs = B_TRUE;

2583         /*
2584          * We want specific DELETE permissions to
2585          * take precedence over WRITE/EXECUTE.  We don't
2586          * want an ACL such as this to mess us up.
2587          * user:joe:write_data:deny,user:joe:delete:allow
2588          *
2589          * However, deny permissions may ultimately be overridden
2590          * by secpolicy_vnode_access().
2591          *
2592          * We will ask for all of the necessary permissions and then
2593          * look at the working modes from the directory and target object
2594          * to determine what was found.
2595          */

2597         if (zp->z_pflags & (ZFS_IMMUTABLE | ZFS_NOUNLINK))
2598                 return (SET_ERROR(EPERM));

2600         /*
2601          * First row
2602          * If the directory permissions allow the delete, we are done.
2603          */
```



```
2604         if ((dzp_error = zfs_zaccess_common(dzp, ACE_DELETE_CHILD,
2605             &dzp_working_mode, &dzpcheck_privs, B_FALSE, cr)) == 0)
2606                 return (0);

2608         /*
2609          * If target object has delete permission then we are done
2610          */
2611         if ((zp_error = zfs_zaccess_common(zp, ACE_DELETE, &zp_working_mode,
2612             &zpcheck_privs, B_FALSE, cr)) == 0)
2613                 return (0);

2615         ASSERT(dzp_error && zp_error);

2617         if (!dzpcheck_privs)
2618                 return (dzp_error);
2619         if (!zpcheck_privs)
2620                 return (zp_error);

2622         /*
2623          * Second row
2624          *
2625          * If directory returns EACCES then delete_child was denied
2626          * due to deny delete_child.  In this case send the request through
2627          * secpolicy_vnode_remove().  We don't use zfs_delete_final_check()
2628          * since that *could* allow the delete based on write/execute permission
2629          * and we want delete permissions to override write/execute.
2630          */

2632         if (dzp_error == EACCES)
2633                 return (secpolicy_vnode_remove(cr));

2635         /*
2636          * Third Row
2637          * only need to see if we have write/execute on directory.
2638          */

2640         dzp_error = zfs_zaccess_common(dzp, ACE_EXECUTE|ACE_WRITE_DATA,
2641             &dzp_working_mode, &dzpcheck_privs, B_FALSE, cr);

2643         if (dzp_error != 0 && !dzpcheck_privs)
2644                 return (dzp_error);

2646         /*
2647          * Fourth row
2648          */

2650         available_perms = (dzp_working_mode & ACE_WRITE_DATA) ? 0 : VWRITE;
2651         available_perms |= (dzp_working_mode & ACE_EXECUTE) ? 0 : VEXEC;

2653         return (zfs_delete_final_check(zp, dzp, available_perms, cr));

2655 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   34539 Thu May 16 17:36:25 2013
new/usr/src/uts/common/fs/zfs/zfs_ctldir.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1052 /*
1053  * pvp is the '.zfs' directory (zfsctl_node_t).
1054  *
1055 #endif /* ! codereview */
1056  * Creates vp, which is '.zfs/snapshot' (zfsctl_snapdir_t).
1057  *
1058  * This function is the callback to create a GFS vnode for '.zfs/snapshot'
1059  * when a lookup is performed on .zfs for "snapshot".
1060  */
1061 vnode_t *
1062 zfsctl_mknode_snapdir(vnode_t *pvp)
1063 {
1064         vnode_t *vp;
1065         zfsctl_snapdir_t *sdp;

1067         vp = gfs_dir_create(sizeof (zfsctl_snapdir_t), pvp,
1068             zfsctl_ops_snapdir, NULL, NULL, MAXNAMELEN,
1069             zfsctl_snapdir_readdir_cb, NULL);
1070         sdp = vp->v_data;
1071         sdp->sd_node.zc_id = ZFSCTL_INO_SNAPDIR;
1072         sdp->sd_node.zc_cmtime = ((zfsctl_node_t *)pvp->v_data)->zc_cmtime;
1073         mutex_init(&sdp->sd_lock, NULL, MUTEX_DEFAULT, NULL);
1074         avl_create(&sdp->sd_snaps, snapentry_compare,
1075             sizeof (zfs_snapentry_t), offsetof(zfs_snapentry_t, se_node));
1076         return (vp);
1077 }

1079 vnode_t *
1080 zfsctl_mknode_shares(vnode_t *pvp)
1081 {
1082         vnode_t *vp;
1083         zfsctl_node_t *sdp;

1085         vp = gfs_dir_create(sizeof (zfsctl_node_t), pvp,
1086             zfsctl_ops_shares, NULL, NULL, MAXNAMELEN,
1087             NULL, NULL);
1088         sdp = vp->v_data;
1089         sdp->zc_cmtime = ((zfsctl_node_t *)pvp->v_data)->zc_cmtime;
1090         return (vp);

1092 }

1094 /* ARGSUSED */
1095 static int
1096 zfsctl_shares_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
1097     caller_context_t *ct)
1098 {
1099         zfsvfs_t *zfsvfs = vp->v_vfsp->vfs_data;
1100         znode_t *dzp;
1101         int error;

1103         ZFS_ENTER(zfsvfs);
1104         if (zfsvfs->z_shares_dir == 0) {
1105                 ZFS_EXIT(zfsvfs);
```



```
1106                 return (SET_ERROR(ENOTSUP));
1107         }
1108         if ((error = zfs_zget(zfsvfs, zfsvfs->z_shares_dir, &dzp)) == 0) {
1109                 error = VOP_GETATTR(ZTOV(dzp), vap, flags, cr, ct);
1110                 VN_RELE(ZTOV(dzp));
1111         }
1112         ZFS_EXIT(zfsvfs);
1113         return (error);


1116 }

1118 /* ARGSUSED */
1119 static int
1120 zfsctl_snapdir_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
1121     caller_context_t *ct)
1122 {
1123         zfsvfs_t *zfsvfs = vp->v_vfsp->vfs_data;
1124         zfsctl_snapdir_t *sdp = vp->v_data;

1126         ZFS_ENTER(zfsvfs);
1127         zfsctl_common_getattr(vp, vap);
1128         vap->va_nodeid = gfs_file_inode(vp);
1129         vap->va_nlink = vap->va_size = avl_numnodes(&sdp->sd_snaps) + 2;
1130         vap->va_ctime = vap->va_mtime = dmu_objset_snap_cmtime(zfsvfs->z_os);
1131         ZFS_EXIT(zfsvfs);

1133         return (0);
1134 }

1136 /* ARGSUSED */
1137 static void
1138 zfsctl_snapdir_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
1139 {
1140         zfsctl_snapdir_t *sdp = vp->v_data;
1141         void *private;

1143         private = gfs_dir_inactive(vp);
1144         if (private != NULL) {
1145                 ASSERT(avl_numnodes(&sdp->sd_snaps) == 0);
1146                 mutex_destroy(&sdp->sd_lock);
1147                 avl_destroy(&sdp->sd_snaps);
1148                 kmem_free(private, sizeof (zfsctl_snapdir_t));
1149         }
1150 }

1152 static const fs_operation_def_t zfsctl_tops_snapdir[] = {
1153         { VOPNAME_OPEN,         { .vop_open = zfsctl_common_open }      },
1154         { VOPNAME_CLOSE,        { .vop_close = zfsctl_common_close }    },
1155         { VOPNAME_IOCTL,        { .error = fs_inval }                   },
1156         { VOPNAME_GETATTR,      { .vop_getattr = zfsctl_snapdir_getattr } },
1157         { VOPNAME_ACCESS,       { .vop_access = zfsctl_common_access }  },
1158         { VOPNAME_RENAME,       { .vop_rename = zfsctl_snapdir_rename } },
1159         { VOPNAME_RMDIR,        { .vop_rmdir = zfsctl_snapdir_remove }  },
1160         { VOPNAME_MKDIR,        { .vop_mkdir = zfsctl_snapdir_mkdir }   },
1161         { VOPNAME_READDIR,      { .vop_readdir = gfs_vop_readdir }      },
1162         { VOPNAME_LOOKUP,       { .vop_lookup = zfsctl_snapdir_lookup } },
1163         { VOPNAME_SEEK,         { .vop_seek = fs_seek }                 },
1164         { VOPNAME_INACTIVE,     { .vop_inactive = zfsctl_snapdir_inactive } },
1165         { VOPNAME_FID,          { .vop_fid = zfsctl_common_fid }        },
1166         { NULL }
1167 };

1169 static const fs_operation_def_t zfsctl_tops_shares[] = {
1170         { VOPNAME_OPEN,         { .vop_open = zfsctl_common_open }      },
1171         { VOPNAME_CLOSE,        { .vop_close = zfsctl_common_close }    },
```

```
1172         { VOPNAME_IOCTL,         { .error = fs_inval }                  },
1173         { VOPNAME_GETATTR,       { .vop_getattr = zfsctl_shares_getattr } },
1174         { VOPNAME_ACCESS,        { .vop_access = zfsctl_common_access }  },
1175         { VOPNAME_READDIR,       { .vop_readdir = zfsctl_shares_readdir } },
1176         { VOPNAME_LOOKUP,        { .vop_lookup = zfsctl_shares_lookup }  },
1177         { VOPNAME_SEEK,          { .vop_seek = fs_seek }                 },
1178         { VOPNAME_INACTIVE,      { .vop_inactive = gfs_vop_inactive } },
1179         { VOPNAME_FID,           { .vop_fid = zfsctl_shares_fid } },
1180         { NULL }
1181 };

1183 /*
1184  * pvp is the GFS vnode '.zfs/snapshot'.
1185  *
1186  * This creates a GFS node under '.zfs/snapshot' representing each
1187  * snapshot.  This newly created GFS node is what we mount snapshot
1188  * vfs_t's ontop of.
1189  */
1190 static vnode_t *
1191 zfsctl_snapshot_mknode(vnode_t *pvp, uint64_t objset)
1192 {
1193         vnode_t *vp;
1194         zfsctl_node_t *zcp;

1196         vp = gfs_dir_create(sizeof (zfsctl_node_t), pvp,
1197             zfsctl_ops_snapshot, NULL, NULL, MAXNAMELEN, NULL, NULL);
1198         zcp = vp->v_data;
1199         zcp->zc_id = objset;

1201         return (vp);
1202 }

1204 static void
1205 zfsctl_snapshot_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
1206 {
1207         zfsctl_snapdir_t *sdp;
1208         zfs_snapentry_t *sep, *next;
1209         vnode_t *dvp;

1211         VERIFY(gfs_dir_lookup(vp, "..", &dvp, cr, 0, NULL, NULL) == 0);
1212         sdp = dvp->v_data;

1214         mutex_enter(&sdp->sd_lock);

1216         if (vp->v_count > 1) {
1217                 mutex_exit(&sdp->sd_lock);
1218                 return;
1219         }
1220         ASSERT(!vn_ismntpt(vp));

1222         sep = avl_first(&sdp->sd_snaps);
1223         while (sep != NULL) {
1224                 next = AVL_NEXT(&sdp->sd_snaps, sep);

1226                 if (sep->se_root == vp) {
1227                         avl_remove(&sdp->sd_snaps, sep);
1228                         kmem_free(sep->se_name, strlen(sep->se_name) + 1);
1229                         kmem_free(sep, sizeof (zfs_snapentry_t));
1230                         break;
1231                 }
1232                 sep = next;
1233         }
1234         ASSERT(sep != NULL);

1236         mutex_exit(&sdp->sd_lock);
1237         VN_RELE(dvp);
```



```
1239         /*
1240          * Dispose of the vnode for the snapshot mount point.
1241          * This is safe to do because once this entry has been removed
1242          * from the AVL tree, it can't be found again, so cannot become
1243          * "active".  If we lookup the same name again we will end up
1244          * creating a new vnode.
1245          */
1246         gfs_vop_inactive(vp, cr, ct);
1247 }


1250 /*
1251  * These VP's should never see the light of day.  They should always
1252  * be covered.
1253  */
1254 static const fs_operation_def_t zfsctl_tops_snapshot[] = {
1255         VOPNAME_INACTIVE, { .vop_inactive =  zfsctl_snapshot_inactive },
1256         NULL, NULL
1257 };

1259 int
1260 zfsctl_lookup_objset(vfs_t *vfsp, uint64_t objsetid, zfsvfs_t **zfsvfsp)
1261 {
1262         zfsvfs_t *zfsvfs = vfsp->vfs_data;
1263         vnode_t *dvp, *vp;
1264         zfsctl_snapdir_t *sdp;
1265         zfsctl_node_t *zcp;
1266         zfs_snapentry_t *sep;
1267         int error;

1269         ASSERT(zfsvfs->z_ctldir != NULL);
1270         error = zfsctl_root_lookup(zfsvfs->z_ctldir, "snapshot", &dvp,
1271             NULL, 0, NULL, kcred, NULL, NULL, NULL);
1272         if (error != 0)
1273                 return (error);
1274         sdp = dvp->v_data;

1276         mutex_enter(&sdp->sd_lock);
1277         sep = avl_first(&sdp->sd_snaps);
1278         while (sep != NULL) {
1279                 vp = sep->se_root;
1280                 zcp = vp->v_data;
1281                 if (zcp->zc_id == objsetid)
1282                         break;

1284                 sep = AVL_NEXT(&sdp->sd_snaps, sep);
1285         }

1287         if (sep != NULL) {
1288                 VN_HOLD(vp);
1289                 /*
1290                  * Return the mounted root rather than the covered mount point.
1291                  * Takes the GFS vnode at .zfs/snapshot/<snapshot objsetid>
1292                  * and returns the ZFS vnode mounted on top of the GFS node.
1293                  * This ZFS vnode is the root of the vfs for objset 'objsetid'.
1294                  */
1295                 error = traverse(&vp);
1296                 if (error == 0) {
1297                         if (vp == sep->se_root)
1298                                 error = SET_ERROR(EINVAL);
1299                         else
1300                                 *zfsvfsp = VTOZ(vp)->z_zfsvfs;
1301                 }
1302                 mutex_exit(&sdp->sd_lock);
1303                 VN_RELE(vp);
```

```
1304         } else {
1305                 error = SET_ERROR(EINVAL);
1306                 mutex_exit(&sdp->sd_lock);
1307         }

1309         VN_RELE(dvp);

1311         return (error);
1312 }

1314 /*
1315  * Unmount any snapshots for the given filesystem.  This is called from
1316  * zfs_umount() - if we have a ctldir, then go through and unmount all the
1317  * snapshots.
1318  */
1319 int
1320 zfsctl_umount_snapshots(vfs_t *vfsp, int fflags, cred_t *cr)
1321 {
1322         zfsvfs_t *zfsvfs = vfsp->vfs_data;
1323         vnode_t *dvp;
1324         zfsctl_snapdir_t *sdp;
1325         zfs_snapentry_t *sep, *next;
1326         int error;

1328         ASSERT(zfsvfs->z_ctldir != NULL);
1329         error = zfsctl_root_lookup(zfsvfs->z_ctldir, "snapshot", &dvp,
1330             NULL, 0, NULL, cr, NULL, NULL, NULL);
1331         if (error != 0)
1332                 return (error);
1333         sdp = dvp->v_data;

1335         mutex_enter(&sdp->sd_lock);

1337         sep = avl_first(&sdp->sd_snaps);
1338         while (sep != NULL) {
1339                 next = AVL_NEXT(&sdp->sd_snaps, sep);

1341                 /*
1342                  * If this snapshot is not mounted, then it must
1343                  * have just been unmounted by somebody else, and
1344                  * will be cleaned up by zfsctl_snapdir_inactive().
1345                  */
1346                 if (vn_ismntpt(sep->se_root)) {
1347                         avl_remove(&sdp->sd_snaps, sep);
1348                         error = zfsctl_unmount_snap(sep, fflags, cr);
1349                         if (error) {
1350                                 avl_add(&sdp->sd_snaps, sep);
1351                                 break;
1352                         }
1353                 }
1354                 sep = next;
1355         }

1357         mutex_exit(&sdp->sd_lock);
1358         VN_RELE(dvp);

1360         return (error);
1361 }
```

```
**********************************************************
  144076 Thu May 16 17:36:25 2013
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 325 /*
 326  * zfs_earlier_version
 327  *
 326  * Return non-zero if the spa version is less than requested version.
 327  */
 328 static int
 329 zfs_earlier_version(const char *name, int version)
 330 {
 331         spa_t *spa;

 333         if (spa_open(name, &spa, FTAG) == 0) {
 334                 if (spa_version(spa) < version) {
 335                         spa_close(spa, FTAG);
 336                         return (1);
 337                 }
 338                 spa_close(spa, FTAG);
 339         }
 340         return (0);
 341 }

 343 /*
 346  * zpl_earlier_version
 347  *
 344  * Return TRUE if the ZPL version is less than requested version.
 345  */
 346 static boolean_t
 347 zpl_earlier_version(const char *name, int version)
 348 {
 349         objset_t *os;
 350         boolean_t rc = B_TRUE;

 352         if (dmu_objset_hold(name, FTAG, &os) == 0) {
 353                 uint64_t zplversion;

 355                 if (dmu_objset_type(os) != DMU_OST_ZFS) {
 356                         dmu_objset_rele(os, FTAG);
 357                         return (B_TRUE);
 358                 }
 359                 /* XXX reading from non-owned objset */
 360                 if (zfs_get_zplprop(os, ZFS_PROP_VERSION, &zplversion) == 0)
 361                         rc = zplversion < version;
 362                 dmu_objset_rele(os, FTAG);
 363         }
 364         return (rc);
 365 }
______unchanged_portion_omitted_

2950 #define ZFS_PROP_UNDEFINED      ((uint64_t)-1)

2952 /*
2953  * inputs:
2954  * os                   parent objset pointer (NULL if root fs)
2955  * fuids_ok             fuids allowed in this version of the spa?
2956  * sa_ok                SAs allowed in this version of the spa?
```



```
2957 #endif /* ! codereview */
2958  * createprops          list of properties requested by creator
2958  * default_zplver       zpl version to use if unspecified in createprops
2959  * fuids_ok             fuids allowed in this version of the spa?
2960  * os                   parent objset pointer (NULL if root fs)
2959  *
2960  * outputs:
2961  * zplprops     values for the zplprops we attach to the master node object
2962  * is_ci        true if requested file system will be purely case-insensitive
2963  *
2964  * Determine the settings for utf8only, normalization and
2965  * casesensitivity.  Specific values may have been requested by the
2966  * creator and/or we can inherit values from the parent dataset.  If
2967  * the file system is of too early a vintage, a creator can not
2968  * request settings for these properties, even if the requested
2969  * setting is the default value.  We don't actually want to create dsl
2970  * properties for these, so remove them from the source nvlist after
2971  * processing.
2972  */
2973 static int
2974 zfs_fill_zplprops_impl(objset_t *os, uint64_t zplver,
2975     boolean_t fuids_ok, boolean_t sa_ok, nvlist_t *createprops,
2976     nvlist_t *zplprops, boolean_t *is_ci)
2977 {
2978         uint64_t sense = ZFS_PROP_UNDEFINED;
2979         uint64_t norm = ZFS_PROP_UNDEFINED;
2980         uint64_t u8 = ZFS_PROP_UNDEFINED;

2982         ASSERT(zplprops != NULL);

2984         /*
2985          * Pull out creator prop choices, if any.
2986          */
2987         if (createprops) {
2988                 (void) nvlist_lookup_uint64(createprops,
2989                     zfs_prop_to_name(ZFS_PROP_VERSION), &zplver);
2990                 (void) nvlist_lookup_uint64(createprops,
2991                     zfs_prop_to_name(ZFS_PROP_NORMALIZE), &norm);
2992                 (void) nvlist_remove_all(createprops,
2993                     zfs_prop_to_name(ZFS_PROP_NORMALIZE));
2994                 (void) nvlist_lookup_uint64(createprops,
2995                     zfs_prop_to_name(ZFS_PROP_UTF8ONLY), &u8);
2996                 (void) nvlist_remove_all(createprops,
2997                     zfs_prop_to_name(ZFS_PROP_UTF8ONLY));
2998                 (void) nvlist_lookup_uint64(createprops,
2999                     zfs_prop_to_name(ZFS_PROP_CASE), &sense);
3000                 (void) nvlist_remove_all(createprops,
3001                     zfs_prop_to_name(ZFS_PROP_CASE));
3002         }

3004         /*
3005          * If the zpl version requested is whacky or the file system
3006          * or pool is version is too "young" to support normalization
3007          * and the creator tried to set a value for one of the props,
3008          * error out.
3009          */
3010         if ((zplver < ZPL_VERSION_INITIAL || zplver > ZPL_VERSION) ||
3011             (zplver >= ZPL_VERSION_FUID && !fuids_ok) ||
3012             (zplver >= ZPL_VERSION_SA && !sa_ok) ||
3013             (zplver < ZPL_VERSION_NORMALIZATION &&
3014             (norm != ZFS_PROP_UNDEFINED || u8 != ZFS_PROP_UNDEFINED ||
3015             sense != ZFS_PROP_UNDEFINED)))
3016                 return (SET_ERROR(ENOTSUP));

3018         /*
3019          * Put the version in the zplprops
```

```
3020          */
3021         VERIFY(nvlist_add_uint64(zplprops,
3022             zfs_prop_to_name(ZFS_PROP_VERSION), zplver) == 0);

3024         if (norm == ZFS_PROP_UNDEFINED)
3025                 VERIFY(zfs_get_zplprop(os, ZFS_PROP_NORMALIZE, &norm) == 0);
3026         VERIFY(nvlist_add_uint64(zplprops,
3027             zfs_prop_to_name(ZFS_PROP_NORMALIZE), norm) == 0);

3029         /*
3030          * If we're normalizing, names must always be valid UTF-8 strings.
3031          */
3032         if (norm)
3033                 u8 = 1;
3034         if (u8 == ZFS_PROP_UNDEFINED)
3035                 VERIFY(zfs_get_zplprop(os, ZFS_PROP_UTF8ONLY, &u8) == 0);
3036         VERIFY(nvlist_add_uint64(zplprops,
3037             zfs_prop_to_name(ZFS_PROP_UTF8ONLY), u8) == 0);

3039         if (sense == ZFS_PROP_UNDEFINED)
3040                 VERIFY(zfs_get_zplprop(os, ZFS_PROP_CASE, &sense) == 0);
3041         VERIFY(nvlist_add_uint64(zplprops,
3042             zfs_prop_to_name(ZFS_PROP_CASE), sense) == 0);

3044         if (is_ci)
3045                 *is_ci = (sense == ZFS_CASE_INSENSITIVE);

3047         return (0);
3048 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   17962 Thu May 16 17:36:25 2013
new/usr/src/uts/common/fs/zfs/zfs_log.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 213 /*
 214  * Handles TX_CREATE, TX_CREATE_ATTR, TX_MKDIR, TX_MKDIR_ATTR and
 215  * TK_MKXATTR transactions.
 214  * zfs_log_create() is used to handle TX_CREATE, TX_CREATE_ATTR, TX_MKDIR,
 215  * TX_MKDIR_ATTR and TX_MKXATTR
 216  * transactions.
 216  *
 217  * TX_CREATE and TX_MKDIR are standard creates, but they may have FUID
 218  * domain information appended prior to the name.  In this case the
 219  * uid/gid in the log record will be a log centric FUID.
 220  *
 221  * TX_CREATE_ACL_ATTR and TX_MKDIR_ACL_ATTR handle special creates that
 222  * may contain attributes, ACL and optional fuid information.
 223  *
 224  * TX_CREATE_ACL and TX_MKDIR_ACL handle special creates that specify
 225  * and ACL and normal users/groups in the ACEs.
 226  *
 227  * There may be an optional xvattr attribute information similar
 228  * to zfs_log_setattr.
 229  *
 230  * Also, after the file name "domain" strings may be appended.
 231  */
 232 void
 233 zfs_log_create(zilog_t *zilog, dmu_tx_t *tx, uint64_t txtype,
 234     znode_t *dzp, znode_t *zp, char *name, vsecattr_t *vsecp,
 235     zfs_fuid_info_t *fuidp, vattr_t *vap)
 236 {
 237         itx_t *itx;
 238         lr_create_t *lr;
 239         lr_acl_create_t *lracl;
 240         size_t aclsize = (vsecp != NULL) ? vsecp->vsa_aclentsz : 0;
 241         size_t xvatsize = 0;
 242         size_t txsize;
 243         xvattr_t *xvap = (xvattr_t *)vap;
 244         void *end;
 245         size_t lrsize;
 246         size_t namesize = strlen(name) + 1;
 247         size_t fuidsz = 0;

 249         if (zil_replaying(zilog, tx))
 250                 return;

 252         /*
 253          * If we have FUIDs present then add in space for
 254          * domains and ACE fuid's if any.
 255          */
 256         if (fuidp) {
 257                 fuidsz += fuidp->z_domain_str_sz;
 258                 fuidsz += fuidp->z_fuid_cnt * sizeof (uint64_t);
 259         }

 261         if (vap->va_mask & AT_XVATTR)
 262                 xvatsize = ZIL_XVAT_SIZE(xvap->xva_mapsize);
```



```
 264         if ((int)txtype == TX_CREATE_ATTR || (int)txtype == TX_MKDIR_ATTR ||
 265             (int)txtype == TX_CREATE || (int)txtype == TX_MKDIR ||
 266             (int)txtype == TX_MKXATTR) {
 267                 txsize = sizeof (*lr) + namesize + fuidsz + xvatsize;
 268                 lrsize = sizeof (*lr);
 269         } else {
 270                 txsize =
 271                     sizeof (lr_acl_create_t) + namesize + fuidsz +
 272                     ZIL_ACE_LENGTH(aclsize) + xvatsize;
 273                 lrsize = sizeof (lr_acl_create_t);
 274         }

 276         itx = zil_itx_create(txtype, txsize);

 278         lr = (lr_create_t *)&itx->itx_lr;
 279         lr->lr_doid = dzp->z_id;
 280         lr->lr_foid = zp->z_id;
 281         lr->lr_mode = zp->z_mode;
 282         if (!IS_EPHEMERAL(zp->z_uid)) {
 283                 lr->lr_uid = (uint64_t)zp->z_uid;
 284         } else {
 285                 lr->lr_uid = fuidp->z_fuid_owner;
 286         }
 287         if (!IS_EPHEMERAL(zp->z_gid)) {
 288                 lr->lr_gid = (uint64_t)zp->z_gid;
 289         } else {
 290                 lr->lr_gid = fuidp->z_fuid_group;
 291         }
 292         (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_GEN(zp->z_zfsvfs), &lr->lr_gen,
 293             sizeof (uint64_t));
 294         (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_CRTIME(zp->z_zfsvfs),
 295             lr->lr_crtime, sizeof (uint64_t) * 2);

 297         if (sa_lookup(zp->z_sa_hdl, SA_ZPL_RDEV(zp->z_zfsvfs), &lr->lr_rdev,
 298             sizeof (lr->lr_rdev)) != 0)
 299                 lr->lr_rdev = 0;

 301         /*
 302          * Fill in xvattr info if any
 303          */
 304         if (vap->va_mask & AT_XVATTR) {
 305                 zfs_log_xvattr((lr_attr_t *)((caddr_t)lr + lrsize), xvap);
 306                 end = (caddr_t)lr + lrsize + xvatsize;
 307         } else {
 308                 end = (caddr_t)lr + lrsize;
 309         }

 311         /* Now fill in any ACL info */

 313         if (vsecp) {
 314                 lracl = (lr_acl_create_t *)&itx->itx_lr;
 315                 lracl->lr_aclcnt = vsecp->vsa_aclcnt;
 316                 lracl->lr_acl_bytes = aclsize;
 317                 lracl->lr_domcnt = fuidp ? fuidp->z_domain_cnt : 0;
 318                 lracl->lr_fuidcnt  = fuidp ? fuidp->z_fuid_cnt : 0;
 319                 if (vsecp->vsa_aclflags & VSA_ACE_ACLFLAGS)
 320                         lracl->lr_acl_flags = (uint64_t)vsecp->vsa_aclflags;
 321                 else
 322                         lracl->lr_acl_flags = 0;

 324                 bcopy(vsecp->vsa_aclentp, end, aclsize);
 325                 end = (caddr_t)end + ZIL_ACE_LENGTH(aclsize);
 326         }

 328         /* drop in FUID info */
 329         if (fuidp) {
```

```
330                 end = zfs_log_fuid_ids(fuidp, end);
331                 end = zfs_log_fuid_domains(fuidp, end);
332         }
333         /*
334          * Now place file name in log record
335          */
336         bcopy(name, end, namesize);
337
338         zil_itx_assign(zilog, itx, tx);
339 }
340
341 /*
342  * Handles both TX_REMOVE and TX_RMDIR transactions.
343  * zfs_log_remove() handles both TX_REMOVE and TX_RMDIR transactions.
343  */
344 void
345 zfs_log_remove(zilog_t *zilog, dmu_tx_t *tx, uint64_t txtype,
346         znode_t *dzp, char *name, uint64_t foid)
347 {
348         itx_t *itx;
349         lr_remove_t *lr;
350         size_t namesize = strlen(name) + 1;
351
352         if (zil_replaying(zilog, tx))
353                 return;
354
355         itx = zil_itx_create(txtype, sizeof (*lr) + namesize);
356         lr = (lr_remove_t *)&itx->itx_lr;
357         lr->lr_doid = dzp->z_id;
358         bcopy(name, (char *)(lr + 1), namesize);
359
360         itx->itx_oid = foid;
361
362         zil_itx_assign(zilog, itx, tx);
363 }
364
365 /*
366  * Handles TX_LINK transactions.
367  * zfs_log_link() handles TX_LINK transactions.
367  */
368 void
369 zfs_log_link(zilog_t *zilog, dmu_tx_t *tx, uint64_t txtype,
370         znode_t *dzp, znode_t *zp, char *name)
371 {
372         itx_t *itx;
373         lr_link_t *lr;
374         size_t namesize = strlen(name) + 1;
375
376         if (zil_replaying(zilog, tx))
377                 return;
378
379         itx = zil_itx_create(txtype, sizeof (*lr) + namesize);
380         lr = (lr_link_t *)&itx->itx_lr;
381         lr->lr_doid = dzp->z_id;
382         lr->lr_link_obj = zp->z_id;
383         bcopy(name, (char *)(lr + 1), namesize);
384
385         zil_itx_assign(zilog, itx, tx);
386 }
387
388 /*
389  * Handles TX_SYMLINK transactions.
390  * zfs_log_symlink() handles TX_SYMLINK transactions.
390  */
391 void
392 zfs_log_symlink(zilog_t *zilog, dmu_tx_t *tx, uint64_t txtype,
```



```
393     znode_t *dzp, znode_t *zp, char *name, char *link)
394 {
395         itx_t *itx;
396         lr_create_t *lr;
397         size_t namesize = strlen(name) + 1;
398         size_t linksize = strlen(link) + 1;
399
400         if (zil_replaying(zilog, tx))
401                 return;
402
403         itx = zil_itx_create(txtype, sizeof (*lr) + namesize + linksize);
404         lr = (lr_create_t *)&itx->itx_lr;
405         lr->lr_doid = dzp->z_id;
406         lr->lr_foid = zp->z_id;
407         lr->lr_uid = zp->z_uid;
408         lr->lr_gid = zp->z_gid;
409         lr->lr_mode = zp->z_mode;
410         (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_GEN(zp->z_zfsvfs), &lr->lr_gen,
411             sizeof (uint64_t));
412         (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_CRTIME(zp->z_zfsvfs),
413             lr->lr_crtime, sizeof (uint64_t) * 2);
414         bcopy(name, (char *)(lr + 1), namesize);
415         bcopy(link, (char *)(lr + 1) + namesize, linksize);
416
417         zil_itx_assign(zilog, itx, tx);
418 }
419
420 /*
421  * Handles TX_RENAME transactions.
422  * zfs_log_rename() handles TX_RENAME transactions.
422  */
423 void
424 zfs_log_rename(zilog_t *zilog, dmu_tx_t *tx, uint64_t txtype,
425         znode_t *sdzp, char *sname, znode_t *tdzp, char *dname, znode_t *szp)
426 {
427         itx_t *itx;
428         lr_rename_t *lr;
429         size_t snamesize = strlen(sname) + 1;
430         size_t dnamesize = strlen(dname) + 1;
431
432         if (zil_replaying(zilog, tx))
433                 return;
434
435         itx = zil_itx_create(txtype, sizeof (*lr) + snamesize + dnamesize);
436         lr = (lr_rename_t *)&itx->itx_lr;
437         lr->lr_sdoid = sdzp->z_id;
438         lr->lr_tdoid = tdzp->z_id;
439         bcopy(sname, (char *)(lr + 1), snamesize);
440         bcopy(dname, (char *)(lr + 1) + snamesize, dnamesize);
441         itx->itx_oid = szp->z_id;
442
443         zil_itx_assign(zilog, itx, tx);
444 }
445
446 /*
447  * Handles TX_WRITE transactions.
448  * zfs_log_write() handles TX_WRITE transactions.
448  */
449 ssize_t zfs_immediate_write_sz = 32768;
450
451 void
452 zfs_log_write(zilog_t *zilog, dmu_tx_t *tx, int txtype,
453         znode_t *zp, offset_t off, ssize_t resid, int ioflag)
454 {
455         itx_wr_state_t write_state;
456         boolean_t slogging;
```

```
457         uintptr_t fsync_cnt;
458         ssize_t immediate_write_sz;
459
460         if (zil_replaying(zilog, tx) || zp->z_unlinked)
461                 return;
462
463         immediate_write_sz = (zilog->zl_logbias == ZFS_LOGBIAS_THROUGHPUT)
464             ? 0 : zfs_immediate_write_sz;
465
466         slogging = spa_has_slogs(zilog->zl_spa) &&
467             (zilog->zl_logbias == ZFS_LOGBIAS_LATENCY);
468         if (resid > immediate_write_sz && !slogging && resid <= zp->z_blksz)
469                 write_state = WR_INDIRECT;
470         else if (ioflag & (FSYNC | FDSYNC))
471                 write_state = WR_COPIED;
472         else
473                 write_state = WR_NEED_COPY;
474
475         if ((fsync_cnt = (uintptr_t)tsd_get(zfs_fsyncer_key)) != 0) {
476                 (void) tsd_set(zfs_fsyncer_key, (void *)(fsync_cnt - 1));
477         }
478
479         while (resid) {
480                 itx_t *itx;
481                 lr_write_t *lr;
482                 ssize_t len;
483
484                 /*
485                  * If the write would overflow the largest block then split it.
486                  */
487                 if (write_state != WR_INDIRECT && resid > ZIL_MAX_LOG_DATA)
488                         len = SPA_MAXBLOCKSIZE >> 1;
489                 else
490                         len = resid;
491
492                 itx = zil_itx_create(txtype, sizeof (*lr) +
493                     (write_state == WR_COPIED ? len : 0));
494                 lr = (lr_write_t *)&itx->itx_lr;
495                 if (write_state == WR_COPIED && dmu_read(zp->z_zfsvfs->z_os,
496                     zp->z_id, off, len, lr + 1, DMU_READ_NO_PREFETCH) != 0) {
497                         zil_itx_destroy(itx);
498                         itx = zil_itx_create(txtype, sizeof (*lr));
499                         lr = (lr_write_t *)&itx->itx_lr;
500                         write_state = WR_NEED_COPY;
501                 }
502
503                 itx->itx_wr_state = write_state;
504                 if (write_state == WR_NEED_COPY)
505                         itx->itx_sod += len;
506                 lr->lr_foid = zp->z_id;
507                 lr->lr_offset = off;
508                 lr->lr_length = len;
509                 lr->lr_blkoff = 0;
510                 BP_ZERO(&lr->lr_blkptr);
511
512                 itx->itx_private = zp->z_zfsvfs;
513
514                 if (!(ioflag & (FSYNC | FDSYNC)) && (zp->z_sync_cnt == 0) &&
515                     (fsync_cnt == 0))
516                         itx->itx_sync = B_FALSE;
517
518                 zil_itx_assign(zilog, itx, tx);
519
520                 off += len;
521                 resid -= len;
522         }
```



```
523 }
524
525 /*
526  * Handles TX_TRUNCATE transactions.
527  * zfs_log_truncate() handles TX_TRUNCATE transactions.
528  */
529 void
530 zfs_log_truncate(zilog_t *zilog, dmu_tx_t *tx, int txtype,
531         znode_t *zp, uint64_t off, uint64_t len)
532 {
533         itx_t *itx;
534         lr_truncate_t *lr;
535
536         if (zil_replaying(zilog, tx) || zp->z_unlinked)
537                 return;
538
539         itx = zil_itx_create(txtype, sizeof (*lr));
540         lr = (lr_truncate_t *)&itx->itx_lr;
541         lr->lr_foid = zp->z_id;
542         lr->lr_offset = off;
543         lr->lr_length = len;
544
545         itx->itx_sync = (zp->z_sync_cnt != 0);
546         zil_itx_assign(zilog, itx, tx);
547 }
548
549 /*
550  * Handles TX_SETATTR transactions.
551  * zfs_log_setattr() handles TX_SETATTR transactions.
552  */
553 void
554 zfs_log_setattr(zilog_t *zilog, dmu_tx_t *tx, int txtype,
555         znode_t *zp, vattr_t *vap, uint_t mask_applied, zfs_fuid_info_t *fuidp)
556 {
557         itx_t           *itx;
558         lr_setattr_t    *lr;
559         xvattr_t        *xvap = (xvattr_t *)vap;
560         size_t          recsize = sizeof (lr_setattr_t);
561         void            *start;
562
563         if (zil_replaying(zilog, tx) || zp->z_unlinked)
564                 return;
565
566         /*
567          * If XVATTR set, then log record size needs to allow
568          * for lr_attr_t + xvattr mask, mapsize and create time
569          * plus actual attribute values
570          */
571         if (vap->va_mask & AT_XVATTR)
572                 recsize = sizeof (*lr) + ZIL_XVAT_SIZE(xvap->xva_mapsize);
573
574         if (fuidp)
575                 recsize += fuidp->z_domain_str_sz;
576
577         itx = zil_itx_create(txtype, recsize);
578         lr = (lr_setattr_t *)&itx->itx_lr;
579         lr->lr_foid = zp->z_id;
580         lr->lr_mask = (uint64_t)mask_applied;
581         lr->lr_mode = (uint64_t)vap->va_mode;
582         if ((mask_applied & AT_UID) && IS_EPHEMERAL(vap->va_uid))
583                 lr->lr_uid = fuidp->z_fuid_owner;
584         else
585                 lr->lr_uid = (uint64_t)vap->va_uid;
586
587         if ((mask_applied & AT_GID) && IS_EPHEMERAL(vap->va_gid))
588                 lr->lr_gid = fuidp->z_fuid_group;
```

```
587         else
588                 lr->lr_gid = (uint64_t)vap->va_gid;
589
590         lr->lr_size = (uint64_t)vap->va_size;
591         ZFS_TIME_ENCODE(&vap->va_atime, lr->lr_atime);
592         ZFS_TIME_ENCODE(&vap->va_mtime, lr->lr_mtime);
593         start = (lr_setattr_t *)(lr + 1);
594         if (vap->va_mask & AT_XVATTR) {
595                 zfs_log_xvattr((lr_attr_t *)start, xvap);
596                 start = (caddr_t)start + ZIL_XVAT_SIZE(xvap->xva_mapsize);
597         }
598
599         /*
600          * Now stick on domain information if any on end
601          */
602
603         if (fuidp)
604                 (void) zfs_log_fuid_domains(fuidp, start);
605
606         itx->itx_sync = (zp->z_sync_cnt != 0);
607         zil_itx_assign(zilog, itx, tx);
608 }
609
610 /*
611  * Handles TX_ACL transactions.
612  * zfs_log_acl() handles TX_ACL transactions.
612  */
613 void
614 zfs_log_acl(zilog_t *zilog, dmu_tx_t *tx, znode_t *zp,
615     vsecattr_t *vsecp, zfs_fuid_info_t *fuidp)
616 {
617         itx_t *itx;
618         lr_acl_v0_t *lrv0;
619         lr_acl_t *lr;
620         int txtype;
621         int lrsize;
622         size_t txsize;
623         size_t aclbytes = vsecp->vsa_aclentsz;
624
625         if (zil_replaying(zilog, tx) || zp->z_unlinked)
626                 return;
627
628         txtype = (zp->z_zfsvfs->z_version < ZPL_VERSION_FUID) ?
629             TX_ACL_V0 : TX_ACL;
630
631         if (txtype == TX_ACL)
632                 lrsize = sizeof (*lr);
633         else
634                 lrsize = sizeof (*lrv0);
635
636         txsize = lrsize +
637             ((txtype == TX_ACL) ? ZIL_ACE_LENGTH(aclbytes) : aclbytes) +
638             (fuidp ? fuidp->z_domain_str_sz : 0) +
639             sizeof (uint64_t) * (fuidp ? fuidp->z_fuid_cnt : 0);
640
641         itx = zil_itx_create(txtype, txsize);
642
643         lr = (lr_acl_t *)&itx->itx_lr;
644         lr->lr_foid = zp->z_id;
645         if (txtype == TX_ACL) {
646                 lr->lr_acl_bytes = aclbytes;
647                 lr->lr_domcnt = fuidp ? fuidp->z_domain_cnt : 0;
648                 lr->lr_fuidcnt = fuidp ? fuidp->z_fuid_cnt : 0;
649                 if (vsecp->vsa_mask & VSA_ACE_ACLFLAGS)
650                         lr->lr_acl_flags = (uint64_t)vsecp->vsa_aclflags;
651                 else
```



```
652                         lr->lr_acl_flags = 0;
653         }
654         lr->lr_aclcnt = (uint64_t)vsecp->vsa_aclcnt;
655
656         if (txtype == TX_ACL_V0) {
657                 lrv0 = (lr_acl_v0_t *)lr;
658                 bcopy(vsecp->vsa_aclentp, (ace_t *)(lrv0 + 1), aclbytes);
659         } else {
660                 void *start = (ace_t *)(lr + 1);
661
662                 bcopy(vsecp->vsa_aclentp, start, aclbytes);
663
664                 start = (caddr_t)start + ZIL_ACE_LENGTH(aclbytes);
665
666                 if (fuidp) {
667                         start = zfs_log_fuid_ids(fuidp, start);
668                         (void) zfs_log_fuid_domains(fuidp, start);
669                 }
670         }
671
672         itx->itx_sync = (zp->z_sync_cnt != 0);
673         zil_itx_assign(zilog, itx, tx);
674 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   17062 Thu May 16 17:36:26 2013
new/usr/src/uts/common/fs/zfs/zfs_rlock.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

/*
 * This file contains the code to implement file range locking in
 * ZFS, although there isn't much specific to ZFS (all that comes to mind is
 * ZFS, although there isn't much specific to ZFS (all that comes to mind
 * support for growing the blocksize).
 *
 * Interface
 * ---------
 * Defined in zfs_rlock.h but essentially:
 *      rl = zfs_range_lock(zp, off, len, lock_type);
 *      zfs_range_unlock(rl);
 *      zfs_range_reduce(rl, off, len);
 *
 * AVL tree
 * --------
 * An AVL tree is used to maintain the state of the existing ranges
 * that are locked for exclusive (writer) or shared (reader) use.
 * The starting range offset is used for searching and sorting the tree.
 *
 * Common case
 * -----------
 * The (hopefully) usual case is of no overlaps or contention for
 * locks. On entry to zfs_lock_range() a rl_t is allocated; the tree
 * searched that finds no overlap, and *this* rl_t is placed in the tree.
 *
 * Overlaps/Reference counting/Proxy locks
 * ---------------------------------------
 * The avl code only allows one node at a particular offset. Also it's very
```



```
 * inefficient to search through all previous entries looking for overlaps
 * (because the very 1st in the ordered list might be at offset 0 but
 * cover the whole file).
 * So this implementation uses reference counts and proxy range locks.
 * Firstly, only reader locks use reference counts and proxy locks,
 * because writer locks are exclusive.
 * When a reader lock overlaps with another then a proxy lock is created
 * for that range and replaces the original lock. If the overlap
 * is exact then the reference count of the proxy is simply incremented.
 * Otherwise, the proxy lock is split into smaller lock ranges and
 * new proxy locks created for non overlapping ranges.
 * The reference counts are adjusted accordingly.
 * Meanwhile, the orginal lock is kept around (this is the callers handle)
 * and its offset and length are used when releasing the lock.
 *
 * Thread coordination
 * -------------------
 * In order to make wakeups efficient and to ensure multiple continuous
 * readers on a range don't starve a writer for the same range lock,
 * two condition variables are allocated in each rl_t.
 * If a writer (or reader) can't get a range it initialises the writer
 * (or reader) cv; sets a flag saying there's a writer (or reader) waiting;
 * and waits on that cv. When a thread unlocks that range it wakes up all
 * writers then all readers before destroying the lock.
 *
 * Append mode writes
 * ------------------
 * Append mode writes need to lock a range at the end of a file.
 * The offset of the end of the file is determined under the
 * range locking mutex, and the lock type converted from RL_APPEND to
 * RL_WRITER and the range locked.
 *
 * Grow block handling
 * -------------------
 * ZFS supports multiple block sizes currently upto 128K. The smallest
 * block size is used for the file which is grown as needed. During this
 * growth all other writers and readers must be excluded.
 * So if the block size needs to be grown then the whole file is
 * exclusively locked, then later the caller will reduce the lock
 * range to just the range to be written using zfs_reduce_range.
 */

#include <sys/zfs_rlock.h>

/*
 * Check if a write lock can be grabbed, or wait and recheck until available.
 */
static void
zfs_range_lock_writer(znode_t *zp, rl_t *new)
{
        avl_tree_t *tree = &zp->z_range_avl;
        rl_t *rl;
        avl_index_t where;
        uint64_t end_size;
        uint64_t off = new->r_off;
        uint64_t len = new->r_len;

        for (;;) {
                /*
                 * Range locking is also used by zvol and uses a
                 * dummied up znode. However, for zvol, we don't need to
                 * append or grow blocksize, and besides we don't have
                 * a "sa" data or z_zfsvfs - so skip that processing.
                 *
                 * Yes, this is ugly, and would be solved by not handling
                 * grow or append in range lock code. If that was done then
```

```
122                  * we could make the range locking code generically available
123                  * to other non-zfs consumers.
124                  */
125                 if (zp->z_vnode) { /* caller is ZPL */
126                         /*
127                          * If in append mode pick up the current end of file.
128                          * This is done under z_range_lock to avoid races.
129                          */
130                         if (new->r_type == RL_APPEND)
131                                 new->r_off = zp->z_size;
132
133                         /*
134                          * If we need to grow the block size then grab the whole
135                          * file range. This is also done under z_range_lock to
136                          * avoid races.
137                          */
138                         end_size = MAX(zp->z_size, new->r_off + len);
139                         if (end_size > zp->z_blksz && (!ISP2(zp->z_blksz) ||
140                             zp->z_blksz < zp->z_zfsvfs->z_max_blksz)) {
141                                 new->r_off = 0;
142                                 new->r_len = UINT64_MAX;
143                         }
144                 }
145
146                 /*
147                  * First check for the usual case of no locks
148                  */
149                 if (avl_numnodes(tree) == 0) {
150                         new->r_type = RL_WRITER; /* convert to writer */
151                         avl_add(tree, new);
152                         return;
153                 }
154
155                 /*
156                  * Look for any locks in the range.
157                  */
158                 rl = avl_find(tree, new, &where);
159                 if (rl)
160                         goto wait; /* already locked at same offset */
161
162                 rl = (rl_t *)avl_nearest(tree, where, AVL_AFTER);
163                 if (rl && (rl->r_off < new->r_off + new->r_len))
164                         goto wait;
165
166                 rl = (rl_t *)avl_nearest(tree, where, AVL_BEFORE);
167                 if (rl && rl->r_off + rl->r_len > new->r_off)
168                         goto wait;
169
170                 new->r_type = RL_WRITER; /* convert possible RL_APPEND */
171                 avl_insert(tree, new, where);
172                 return;
173 wait:
174                 if (!rl->r_write_wanted) {
175                         cv_init(&rl->r_wr_cv, NULL, CV_DEFAULT, NULL);
176                         rl->r_write_wanted = B_TRUE;
177                 }
178                 cv_wait(&rl->r_wr_cv, &zp->z_range_lock);
179
180                 /* reset to original */
181                 new->r_off = off;
182                 new->r_len = len;
183         }
184 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   10532 Thu May 16 17:36:26 2013
new/usr/src/uts/common/fs/zfs/zfs_sa.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 187 /*
 188  * I'm not convinced we should do any of this upgrade.
 189  * since the SA code can read both old/new znode formats
 190  * with probably little to no performance difference.
 190  * with probably little to know performance difference.
 191  *
 192  * All new files will be created with the new format.
 193  */

 195 void
 196 zfs_sa_upgrade(sa_handle_t *hdl, dmu_tx_t *tx)
 197 {
 198         dmu_buf_t *db = sa_get_db(hdl);
 199         znode_t *zp = sa_get_userdata(hdl);
 200         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
 201         sa_bulk_attr_t bulk[20];
 202         int count = 0;
 203         sa_bulk_attr_t sa_attrs[20] = { 0 };
 204         zfs_acl_locator_cb_t locate = { 0 };
 205         uint64_t uid, gid, mode, rdev, xattr, parent;
 206         uint64_t crtime[2], mtime[2], ctime[2];
 207         zfs_acl_phys_t znode_acl;
 208         char scanstamp[AV_SCANSTAMP_SZ];
 209         boolean_t drop_lock = B_FALSE;

 211         /*
 212          * No upgrade if ACL isn't cached
 213          * since we won't know which locks are held
 214          * and ready the ACL would require special "locked"
 215          * interfaces that would be messy
 216          */
 217         if (zp->z_acl_cached == NULL || ZTOV(zp)->v_type == VLNK)
 218                 return;

 220         /*
 221          * If the z_lock is held and we aren't the owner
 222          * the just return since we don't want to deadlock
 223          * trying to update the status of z_is_sa.  This
 224          * file can then be upgraded at a later time.
 225          *
 226          * Otherwise, we know we are doing the
 227          * sa_update() that caused us to enter this function.
 228          */
 229         if (mutex_owner(&zp->z_lock) != curthread) {
 230                 if (mutex_tryenter(&zp->z_lock) == 0)
 231                         return;
 232                 else
 233                         drop_lock = B_TRUE;
 234         }

 236         /* First do a bulk query of the attributes that aren't cached */
 237         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, &mtime, 16);
 238         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, &ctime, 16);
 239         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CRTIME(zfsvfs), NULL, &crtime, 16);
```



```
 240         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs), NULL, &mode, 8);
 241         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_PARENT(zfsvfs), NULL, &parent, 8);
 242         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_XATTR(zfsvfs), NULL, &xattr, 8);
 243         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_RDEV(zfsvfs), NULL, &rdev, 8);
 244         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_UID(zfsvfs), NULL, &uid, 8);
 245         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_GID(zfsvfs), NULL, &gid, 8);
 246         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_ZNODE_ACL(zfsvfs), NULL,
 247             &znode_acl, 88);

 249         if (sa_bulk_lookup_locked(hdl, bulk, count) != 0)
 250                 goto done;


 253         /*
 254          * While the order here doesn't matter its best to try and organize
 255          * it is such a way to pick up an already existing layout number
 256          */
 257         count = 0;
 258         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_MODE(zfsvfs), NULL, &mode, 8);
 259         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_SIZE(zfsvfs), NULL,
 260             &zp->z_size, 8);
 261         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_GEN(zfsvfs),
 262             NULL, &zp->z_gen, 8);
 263         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_UID(zfsvfs), NULL, &uid, 8);
 264         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_GID(zfsvfs), NULL, &gid, 8);
 265         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_PARENT(zfsvfs),
 266             NULL, &parent, 8);
 267         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_FLAGS(zfsvfs), NULL,
 268             &zp->z_pflags, 8);
 269         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_ATIME(zfsvfs), NULL,
 270             zp->z_atime, 16);
 271         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_MTIME(zfsvfs), NULL,
 272             &mtime, 16);
 273         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_CTIME(zfsvfs), NULL,
 274             &ctime, 16);
 275         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_CRTIME(zfsvfs), NULL,
 276             &crtime, 16);
 277         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_LINKS(zfsvfs), NULL,
 278             &zp->z_links, 8);
 279         if (zp->z_vnode->v_type == VBLK || zp->z_vnode->v_type == VCHR)
 280                 SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_RDEV(zfsvfs), NULL,
 281                     &rdev, 8);
 282         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_DACL_COUNT(zfsvfs), NULL,
 283             &zp->z_acl_cached->z_acl_count, 8);

 285         if (zp->z_acl_cached->z_version < ZFS_ACL_VERSION_FUID)
 286                 zfs_acl_xform(zp, zp->z_acl_cached, CRED());

 288         locate.cb_aclp = zp->z_acl_cached;
 289         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_DACL_ACES(zfsvfs),
 290             zfs_acl_data_locator, &locate, zp->z_acl_cached->z_acl_bytes);

 292         if (xattr)
 293                 SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_XATTR(zfsvfs),
 294                     NULL, &xattr, 8);

 296         /* if scanstamp then add scanstamp */

 298         if (zp->z_pflags & ZFS_BONUS_SCANSTAMP) {
 299                 bcopy((caddr_t)db->db_data + ZFS_OLD_ZNODE_PHYS_SIZE,
 300                     scanstamp, AV_SCANSTAMP_SZ);
 301                 SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_SCANSTAMP(zfsvfs),
 302                     NULL, scanstamp, AV_SCANSTAMP_SZ);
 303                 zp->z_pflags &= ~ZFS_BONUS_SCANSTAMP;
 304         }
```

```
 306         VERIFY(dmu_set_bonustype(db, DMU_OT_SA, tx) == 0);
 307         VERIFY(sa_replace_all_by_template_locked(hdl, sa_attrs,
 308             count, tx) == 0);
 309         if (znode_acl.z_acl_extern_obj)
 310                 VERIFY(0 == dmu_object_free(zfsvfs->z_os,
 311                     znode_acl.z_acl_extern_obj, tx));

 313         zp->z_is_sa = B_TRUE;
 314 done:
 315         if (drop_lock)
 316                 mutex_exit(&zp->z_lock);
 317 }
```

unchanged_portion_omitted_

```
**********************************************************
   60200 Thu May 16 17:36:26 2013
new/usr/src/uts/common/fs/zfs/zfs_vfsops.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1350 /*
1351  * Check that the hex label string is appropriate for the dataset being
1352  * mounted into the global_zone proper.
1351  * zfs_check_global_label:
1352  *      Check that the hex label string is appropriate for the dataset
1353  *      being mounted into the global_zone proper.
1353  *
1354  * Return an error if the hex label string is not default or
1355  * admin_low/admin_high.  For admin_low labels, the corresponding
1356  * dataset must be readonly.
1357  */
1358 int
1359 zfs_check_global_label(const char *dsname, const char *hexsl)
1360 {
1361         if (strcasecmp(hexsl, ZFS_MLSLABEL_DEFAULT) == 0)
1362                 return (0);
1363         if (strcasecmp(hexsl, ADMIN_HIGH) == 0)
1364                 return (0);
1365         if (strcasecmp(hexsl, ADMIN_LOW) == 0) {
1366                 /* must be readonly */
1367                 uint64_t rdonly;

1369                 if (dsl_prop_get_integer(dsname,
1370                     zfs_prop_to_name(ZFS_PROP_READONLY), &rdonly, NULL))
1371                         return (SET_ERROR(EACCES));
1372                 return (rdonly ? 0 : EACCES);
1373         }
1374         return (SET_ERROR(EACCES));
1375 }

1377 /*
1379  * zfs_mount_label_policy:
1378  * Determine whether the mount is allowed according to MAC check.
1379  * by comparing (where appropriate) label of the dataset against
1380  * the label of the zone being mounted into.  If the dataset has
1381  * no label, create one.
1382  *
1383  * Returns 0 if access allowed, error otherwise (e.g. EACCES)
1385  *      Returns:
1386  *               0 :    access allowed
1387  *              >0 :    error code, such as EACCES
1384  */
1385 static int
1386 zfs_mount_label_policy(vfs_t *vfsp, char *osname)
1387 {
1388         int             error, retv;
1389         zone_t          *mntzone = NULL;
1390         ts_label_t      *mnt_tsl;
1391         bslabel_t       *mnt_sl;
1392         bslabel_t       ds_sl;
1393         char            ds_hexsl[MAXNAMELEN];

1395         retv = EACCES;                          /* assume the worst */
```



```
1397         /*
1398          * Start by getting the dataset label if it exists.
1399          */
1400         error = dsl_prop_get(osname, zfs_prop_to_name(ZFS_PROP_MLSLABEL),
1401             1, sizeof (ds_hexsl), &ds_hexsl, NULL);
1402         if (error)
1403                 return (SET_ERROR(EACCES));

1405         /*
1406          * If labeling is NOT enabled, then disallow the mount of datasets
1407          * which have a non-default label already.  No other label checks
1408          * are needed.
1409          */
1410         if (!is_system_labeled()) {
1411                 if (strcasecmp(ds_hexsl, ZFS_MLSLABEL_DEFAULT) == 0)
1412                         return (0);
1413                 return (SET_ERROR(EACCES));
1414         }

1416         /*
1417          * Get the label of the mountpoint.  If mounting into the global
1418          * zone (i.e. mountpoint is not within an active zone and the
1419          * zoned property is off), the label must be default or
1420          * admin_low/admin_high only; no other checks are needed.
1421          */
1422         mntzone = zone_find_by_any_path(refstr_value(vfsp->vfs_mntpt), B_FALSE);
1423         if (mntzone->zone_id == GLOBAL_ZONEID) {
1424                 uint64_t zoned;

1426                 zone_rele(mntzone);

1428                 if (dsl_prop_get_integer(osname,
1429                     zfs_prop_to_name(ZFS_PROP_ZONED), &zoned, NULL))
1430                         return (SET_ERROR(EACCES));
1431                 if (!zoned)
1432                         return (zfs_check_global_label(osname, ds_hexsl));
1433                 else
1434                         /*
1435                          * This is the case of a zone dataset being mounted
1436                          * initially, before the zone has been fully created;
1437                          * allow this mount into global zone.
1438                          */
1439                         return (0);
1440         }

1442         mnt_tsl = mntzone->zone_slabel;
1443         ASSERT(mnt_tsl != NULL);
1444         label_hold(mnt_tsl);
1445         mnt_sl = label2bslabel(mnt_tsl);

1447         if (strcasecmp(ds_hexsl, ZFS_MLSLABEL_DEFAULT) == 0) {
1448                 /*
1449                  * The dataset doesn't have a real label, so fabricate one.
1450                  */
1451                 char *str = NULL;

1453                 if (l_to_str_internal(mnt_sl, &str) == 0 &&
1454                     dsl_prop_set_string(osname,
1455                     zfs_prop_to_name(ZFS_PROP_MLSLABEL),
1456                     ZPROP_SRC_LOCAL, str) == 0)
1457                         retv = 0;
1458                 if (str != NULL)
1459                         kmem_free(str, strlen(str) + 1);
1460         } else if (hexstr_to_label(ds_hexsl, &ds_sl) == 0) {
1461                 /*
1462                  * Now compare labels to complete the MAC check.  If the
```

```
1463                  * labels are equal then allow access.  If the mountpoint
1464                  * label dominates the dataset label, allow readonly access.
1465                  * Otherwise, access is denied.
1466                  */
1467                 if (blequal(mnt_sl, &ds_sl))
1468                         retv = 0;
1469                 else if (bldominates(mnt_sl, &ds_sl)) {
1470                         vfs_setmntopt(vfsp, MNTOPT_RO, NULL, 0);
1471                         retv = 0;
1472                 }
1473         }

1475         label_rele(mnt_tsl);
1476         zone_rele(mntzone);
1477         return (retv);
1478 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  130563 Thu May 16 17:36:27 2013
new/usr/src/uts/common/fs/zfs/zfs_vnops.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

offset_t zfs_read_chunk_size = 1024 * 1024; /* Tunable */

/*
 * Read bytes from specified file into supplied buffer.
 *
 *      IN:     vp      - vnode of file to be read from.
 *              uio     - structure supplying read location, range info,
 *                        and return buffer.
 *              ioflag  - SYNC flags; used to provide FRSYNC semantics.
 *              cr      - credentials of caller.
 *              ct      - caller context
 *
 *      OUT:    uio     - updated offset and range, buffer filled.
 *
 *      RETURN: 0 on success, error code on failure.
 *      RETURN: 0 if success
 *              error code if failure
 *
 * Side Effects:
 *      vp - atime updated if byte count > 0
 */
/* ARGSUSED */
static int
zfs_read(vnode_t *vp, uio_t *uio, int ioflag, cred_t *cr, caller_context_t *ct)
{
        znode_t         *zp = VTOZ(vp);
        zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
        objset_t        *os;
        ssize_t         n, nbytes;
        int             error = 0;
        rl_t            *rl;
        xuio_t          *xuio = NULL;

        ZFS_ENTER(zfsvfs);
        ZFS_VERIFY_ZP(zp);
        os = zfsvfs->z_os;

        if (zp->z_pflags & ZFS_AV_QUARANTINED) {
                ZFS_EXIT(zfsvfs);
                return (SET_ERROR(EACCES));
        }

        /*
         * Validate file offset
         */
        if (uio->uio_loffset < (offset_t)0) {
                ZFS_EXIT(zfsvfs);
                return (SET_ERROR(EINVAL));
        }

        /*
         * Fasttrack empty reads
         */
        if (uio->uio_resid == 0) {
```



```
                ZFS_EXIT(zfsvfs);
                return (0);
        }

        /*
         * Check for mandatory locks
         */
        if (MANDMODE(zp->z_mode)) {
                if (error = chklock(vp, FREAD,
                    uio->uio_loffset, uio->uio_resid, uio->uio_fmode, ct)) {
                        ZFS_EXIT(zfsvfs);
                        return (error);
                }
        }

        /*
         * If we're in FRSYNC mode, sync out this znode before reading it.
         */
        if (ioflag & FRSYNC || zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
                zil_commit(zfsvfs->z_log, zp->z_id);

        /*
         * Lock the range against changes.
         */
        rl = zfs_range_lock(zp, uio->uio_loffset, uio->uio_resid, RL_READER);

        /*
         * If we are reading past end-of-file we can skip
         * to the end; but we might still need to set atime.
         */
        if (uio->uio_loffset >= zp->z_size) {
                error = 0;
                goto out;
        }

        ASSERT(uio->uio_loffset < zp->z_size);
        n = MIN(uio->uio_resid, zp->z_size - uio->uio_loffset);

        if ((uio->uio_extflg == UIO_XUIO) &&
            (((xuio_t *)uio)->xu_type == UIOTYPE_ZEROCOPY)) {
                int nblk;
                int blksz = zp->z_blksz;
                uint64_t offset = uio->uio_loffset;

                xuio = (xuio_t *)uio;
                if ((ISP2(blksz))) {
                        nblk = (P2ROUNDUP(offset + n, blksz) - P2ALIGN(offset,
                            blksz)) / blksz;
                } else {
                        ASSERT(offset + n <= blksz);
                        nblk = 1;
                }
                (void) dmu_xuio_init(xuio, nblk);

                if (vn_has_cached_data(vp)) {
                        /*
                         * For simplicity, we always allocate a full buffer
                         * even if we only expect to read a portion of a block.
                         */
                        while (--nblk >= 0) {
                                (void) dmu_xuio_add(xuio,
                                    dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
                                    blksz), 0, blksz);
                        }
                }
        }
```

```
541         while (n > 0) {
542                 nbytes = MIN(n, zfs_read_chunk_size -
543                     P2PHASE(uio->uio_loffset, zfs_read_chunk_size));
545                 if (vn_has_cached_data(vp))
546                         error = mappedread(vp, nbytes, uio);
547                 else
548                         error = dmu_read_uio(os, zp->z_id, uio, nbytes);
549                 if (error) {
550                         /* convert checksum errors into IO errors */
551                         if (error == ECKSUM)
552                                 error = SET_ERROR(EIO);
553                         break;
554                 }
556                 n -= nbytes;
557         }
558 out:
559         zfs_range_unlock(rl);
561         ZFS_ACCESSTIME_STAMP(zfsvfs, zp);
562         ZFS_EXIT(zfsvfs);
563         return (error);
564 }
566 /*
567  * Write the bytes to a file.
568  *
569  *      IN:     vp      - vnode of file to be written to.
570  *              uio     - structure supplying write location, range info,
571  *                        and data buffer.
572  *              ioflag  - FAPPEND, FSYNC, and/or FDSYNC.  FAPPEND is
573  *                        set if in append mode.
573  *              ioflag  - FAPPEND flag set if in append mode.
574  *              cr      - credentials of caller.
575  *              ct      - caller context (NFS/CIFS fem monitor only)
576  *
577  *      OUT:    uio     - updated offset and range.
578  *
579  *      RETURN: 0 on success, error code on failure.
579  *      RETURN: 0 if success
580  *              error code if failure
580  *
581  * Timestamps:
582  *      vp - ctime|mtime updated if byte count > 0
583  */
585 /* ARGSUSED */
586 static int
587 zfs_write(vnode_t *vp, uio_t *uio, int ioflag, cred_t *cr, caller_context_t *ct)
588 {
589         znode_t         *zp = VTOZ(vp);
590         rlim64_t        limit = uio->uio_llimit;
591         ssize_t         start_resid = uio->uio_resid;
592         ssize_t         tx_bytes;
593         uint64_t        end_size;
594         dmu_tx_t        *tx;
595         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
596         zilog_t         *zilog;
597         offset_t        woff;
598         ssize_t         n, nbytes;
599         rl_t            *rl;
600         int             max_blksz = zfsvfs->z_max_blksz;
601         int             error = 0;
602         arc_buf_t       *abuf;
```



```
603         iovec_t         *aiov = NULL;
604         xuio_t          *xuio = NULL;
605         int             i_iov = 0;
606         int             iovcnt = uio->uio_iovcnt;
607         iovec_t         *iovp = uio->uio_iov;
608         int             write_eof;
609         int             count = 0;
610         sa_bulk_attr_t  bulk[4];
611         uint64_t        mtime[2], ctime[2];
613         /*
614          * Fasttrack empty write
615          */
616         n = start_resid;
617         if (n == 0)
618                 return (0);
620         if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
621                 limit = MAXOFFSET_T;
623         ZFS_ENTER(zfsvfs);
624         ZFS_VERIFY_ZP(zp);
626         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, &mtime, 16);
627         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, &ctime, 16);
628         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_SIZE(zfsvfs), NULL,
629             &zp->z_size, 8);
630         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
631             &zp->z_pflags, 8);
633         /*
634          * If immutable or not appending then return EPERM
635          */
636         if ((zp->z_pflags & (ZFS_IMMUTABLE | ZFS_READONLY)) ||
637             ((zp->z_pflags & ZFS_APPENDONLY) && !(ioflag & FAPPEND) &&
638             (uio->uio_loffset < zp->z_size))) {
639                 ZFS_EXIT(zfsvfs);
640                 return (SET_ERROR(EPERM));
641         }
643         zilog = zfsvfs->z_log;
645         /*
646          * Validate file offset
647          */
648         woff = ioflag & FAPPEND ? zp->z_size : uio->uio_loffset;
649         if (woff < 0) {
650                 ZFS_EXIT(zfsvfs);
651                 return (SET_ERROR(EINVAL));
652         }
654         /*
655          * Check for mandatory locks before calling zfs_range_lock()
656          * in order to prevent a deadlock with locks set via fcntl().
657          */
658         if (MANDMODE((mode_t)zp->z_mode) &&
659             (error = chklock(vp, FWRITE, woff, n, uio->uio_fmode, ct)) != 0) {
660                 ZFS_EXIT(zfsvfs);
661                 return (error);
662         }
664         /*
665          * Pre-fault the pages to ensure slow (eg NFS) pages
666          * don't hold up txg.
667          * Skip this if uio contains loaned arc_buf.
668          */
```

```
669         if ((uio->uio_extflg == UIO_XUIO) &&
670             (((xuio_t *)uio)->xu_type == UIOTYPE_ZEROCOPY))
671                 xuio = (xuio_t *)uio;
672         else
673                 uio_prefaultpages(MIN(n, max_blksz), uio);
674
675         /*
676          * If in append mode, set the io offset pointer to eof.
677          */
678         if (ioflag & FAPPEND) {
679                 /*
680                  * Obtain an appending range lock to guarantee file append
681                  * semantics.  We reset the write offset once we have the lock.
682                  */
683                 rl = zfs_range_lock(zp, 0, n, RL_APPEND);
684                 woff = rl->r_off;
685                 if (rl->r_len == UINT64_MAX) {
686                         /*
687                          * We overlocked the file because this write will cause
688                          * the file block size to increase.
689                          * Note that zp_size cannot change with this lock held.
690                          */
691                         woff = zp->z_size;
692                 }
693                 uio->uio_loffset = woff;
694         } else {
695                 /*
696                  * Note that if the file block size will change as a result of
697                  * this write, then this range lock will lock the entire file
698                  * so that we can re-write the block safely.
699                  */
700                 rl = zfs_range_lock(zp, woff, n, RL_WRITER);
701         }
702
703         if (woff >= limit) {
704                 zfs_range_unlock(rl);
705                 ZFS_EXIT(zfsvfs);
706                 return (SET_ERROR(EFBIG));
707         }
708
709         if ((woff + n) > limit || woff > (limit - n))
710                 n = limit - woff;
711
712         /* Will this write extend the file length? */
713         write_eof = (woff + n > zp->z_size);
714
715         end_size = MAX(zp->z_size, woff + n);
716
717         /*
718          * Write the file in reasonable size chunks.  Each chunk is written
719          * in a separate transaction; this keeps the intent log records small
720          * and allows us to do more fine-grained space accounting.
721          */
722         while (n > 0) {
723                 abuf = NULL;
724                 woff = uio->uio_loffset;
725 again:
726                 if (zfs_owner_overquota(zfsvfs, zp, B_FALSE) ||
727                     zfs_owner_overquota(zfsvfs, zp, B_TRUE)) {
728                         if (abuf != NULL)
729                                 dmu_return_arcbuf(abuf);
730                         error = SET_ERROR(EDQUOT);
731                         break;
732                 }
733
734                 if (xuio && abuf == NULL) {
```



```
735                         ASSERT(i_iov < iovcnt);
736                         aiov = &iovp[i_iov];
737                         abuf = dmu_xuio_arcbuf(xuio, i_iov);
738                         dmu_xuio_clear(xuio, i_iov);
739                         DTRACE_PROBE3(zfs_cp_write, int, i_iov,
740                             iovec_t *, aiov, arc_buf_t *, abuf);
741                         ASSERT((aiov->iov_base == abuf->b_data) ||
742                             ((char *)aiov->iov_base - (char *)abuf->b_data +
743                             aiov->iov_len == arc_buf_size(abuf)));
744                         i_iov++;
745                 } else if (abuf == NULL && n >= max_blksz &&
746                     woff >= zp->z_size &&
747                     P2PHASE(woff, max_blksz) == 0 &&
748                     zp->z_blksz == max_blksz) {
749                         /*
750                          * This write covers a full block.  "Borrow" a buffer
751                          * from the dmu so that we can fill it before we enter
752                          * a transaction.  This avoids the possibility of
753                          * holding up the transaction if the data copy hangs
754                          * up on a pagefault (e.g., from an NFS server mapping).
755                          */
756                         size_t cbytes;
757
758                         abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
759                             max_blksz);
760                         ASSERT(abuf != NULL);
761                         ASSERT(arc_buf_size(abuf) == max_blksz);
762                         if (error = uiocopy(abuf->b_data, max_blksz,
763                             UIO_WRITE, uio, &cbytes)) {
764                                 dmu_return_arcbuf(abuf);
765                                 break;
766                         }
767                         ASSERT(cbytes == max_blksz);
768                 }
769
770                 /*
771                  * Start a transaction.
772                  */
773                 tx = dmu_tx_create(zfsvfs->z_os);
774                 dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
775                 dmu_tx_hold_write(tx, zp->z_id, woff, MIN(n, max_blksz));
776                 zfs_sa_upgrade_txholds(tx, zp);
777                 error = dmu_tx_assign(tx, TXG_NOWAIT);
778                 if (error) {
779                         if (error == ERESTART) {
780                                 dmu_tx_wait(tx);
781                                 dmu_tx_abort(tx);
782                                 goto again;
783                         }
784                         dmu_tx_abort(tx);
785                         if (abuf != NULL)
786                                 dmu_return_arcbuf(abuf);
787                         break;
788                 }
789
790                 /*
791                  * If zfs_range_lock() over-locked we grow the blocksize
792                  * and then reduce the lock range.  This will only happen
793                  * on the first iteration since zfs_range_reduce() will
794                  * shrink down r_len to the appropriate size.
795                  */
796                 if (rl->r_len == UINT64_MAX) {
797                         uint64_t new_blksz;
798
799                         if (zp->z_blksz > max_blksz) {
800                                 ASSERT(!ISP2(zp->z_blksz));
```

```
801                             new_blksz = MIN(end_size, SPA_MAXBLOCKSIZE);
802                     } else {
803                             new_blksz = MIN(end_size, max_blksz);
804                     }
805                     zfs_grow_blocksize(zp, new_blksz, tx);
806                     zfs_range_reduce(rl, woff, n);
807             }

809             /*
810              * XXX - should we really limit each write to z_max_blksz?
811              * Perhaps we should use SPA_MAXBLOCKSIZE chunks?
812              */
813             nbytes = MIN(n, max_blksz - P2PHASE(woff, max_blksz));

815             if (abuf == NULL) {
816                     tx_bytes = uio->uio_resid;
817                     error = dmu_write_uio_dbuf(sa_get_db(zp->z_sa_hdl),
818                         uio, nbytes, tx);
819                     tx_bytes -= uio->uio_resid;
820             } else {
821                     tx_bytes = nbytes;
822                     ASSERT(xuio == NULL || tx_bytes == aiov->iov_len);
823                     /*
824                      * If this is not a full block write, but we are
825                      * extending the file past EOF and this data starts
826                      * block-aligned, use assign_arcbuf().  Otherwise,
827                      * write via dmu_write().
828                      */
829                     if (tx_bytes < max_blksz && (!write_eof ||
830                         aiov->iov_base != abuf->b_data)) {
831                             ASSERT(xuio);
832                             dmu_write(zfsvfs->z_os, zp->z_id, woff,
833                                 aiov->iov_len, aiov->iov_base, tx);
834                             dmu_return_arcbuf(abuf);
835                             xuio_stat_wbuf_copied();
836                     } else {
837                             ASSERT(xuio || tx_bytes == max_blksz);
838                             dmu_assign_arcbuf(sa_get_db(zp->z_sa_hdl),
839                                 woff, abuf, tx);
840                     }
841                     ASSERT(tx_bytes <= uio->uio_resid);
842                     uioskip(uio, tx_bytes);
843             }
844             if (tx_bytes && vn_has_cached_data(vp)) {
845                     update_pages(vp, woff,
846                         tx_bytes, zfsvfs->z_os, zp->z_id);
847             }

849             /*
850              * If we made no progress, we're done.  If we made even
851              * partial progress, update the znode and ZIL accordingly.
852              */
853             if (tx_bytes == 0) {
854                     (void) sa_update(zp->z_sa_hdl, SA_ZPL_SIZE(zfsvfs),
855                         (void *)&zp->z_size, sizeof (uint64_t), tx);
856                     dmu_tx_commit(tx);
857                     ASSERT(error != 0);
858                     break;
859             }

861             /*
862              * Clear Set-UID/Set-GID bits on successful write if not
863              * privileged and at least one of the excute bits is set.
864              *
865              * It would be nice to to this after all writes have
866              * been done, but that would still expose the ISUID/ISGID
```



```
867              * to another app after the partial write is committed.
868              *
869              * Note: we don't call zfs_fuid_map_id() here because
870              * user 0 is not an ephemeral uid.
871              */
872             mutex_enter(&zp->z_acl_lock);
873             if ((zp->z_mode & (S_IXUSR | (S_IXUSR >> 3) |
874                 (S_IXUSR >> 6))) != 0 &&
875                 (zp->z_mode & (S_ISUID | S_ISGID)) != 0 &&
876                 secpolicy_vnode_setid_retain(cr,
877                 (zp->z_mode & S_ISUID) != 0 && zp->z_uid == 0) != 0) {
878                     uint64_t newmode;
879                     zp->z_mode &= ~(S_ISUID | S_ISGID);
880                     newmode = zp->z_mode;
881                     (void) sa_update(zp->z_sa_hdl, SA_ZPL_MODE(zfsvfs),
882                         (void *)&newmode, sizeof (uint64_t), tx);
883             }
884             mutex_exit(&zp->z_acl_lock);

886             zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
887                 B_TRUE);

889             /*
890              * Update the file size (zp_size) if it has changed;
891              * account for possible concurrent updates.
892              */
893             while ((end_size = zp->z_size) < uio->uio_loffset) {
894                     (void) atomic_cas_64(&zp->z_size, end_size,
895                         uio->uio_loffset);
896                     ASSERT(error == 0);
897             }
898             /*
899              * If we are replaying and eof is non zero then force
900              * the file size to the specified eof. Note, there's no
901              * concurrency during replay.
902              */
903             if (zfsvfs->z_replay && zfsvfs->z_replay_eof != 0)
904                     zp->z_size = zfsvfs->z_replay_eof;

906             error = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);

908             zfs_log_write(zilog, tx, TX_WRITE, zp, woff, tx_bytes, ioflag);
909             dmu_tx_commit(tx);

911             if (error != 0)
912                     break;
913             ASSERT(tx_bytes == nbytes);
914             n -= nbytes;

916             if (!xuio && n > 0)
917                     uio_prefaultpages(MIN(n, max_blksz), uio);
918     }

920     zfs_range_unlock(rl);

922     /*
923      * If we're in replay mode, or we made no progress, return error.
924      * Otherwise, it's at least a partial write, so it's successful.
925      */
926     if (zfsvfs->z_replay || uio->uio_resid == start_resid) {
927             ZFS_EXIT(zfsvfs);
928             return (error);
929     }

931     if (ioflag & (FSYNC | FDSYNC) ||
932         zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
```

```
 933                 zil_commit(zilog, zp->z_id);

 935         ZFS_EXIT(zfsvfs);
 936         return (0);
 937 }
_____unchanged_portion_omitted_


1134 /*
1135  * Lookup an entry in a directory, or an extended attribute directory.
1136  * If it exists, return a held vnode reference for it.
1137  *
1138  *      IN:     dvp     - vnode of directory to search.
1139  *              nm      - name of entry to lookup.
1140  *              pnp     - full pathname to lookup [UNUSED].
1141  *              flags   - LOOKUP_XATTR set if looking for an attribute.
1142  *              rdir    - root directory vnode [UNUSED].
1143  *              cr      - credentials of caller.
1144  *              ct      - caller context
1145  *              direntflags - directory lookup flags
1146  *              realpnp - returned pathname.
1147  *
1148  *      OUT:    vpp     - vnode of located entry, NULL if not found.
1149  *
1150  *      RETURN: 0 on success, error code on failure.
1151  *      RETURN: 0 if success
1152  *              error code if failure
1151  *
1152  * Timestamps:
1153  *      NA
1154  */
1155 /* ARGSUSED */
1156 static int
1157 zfs_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, struct pathname *pnp,
1158     int flags, vnode_t *rdir, cred_t *cr,  caller_context_t *ct,
1159     int *direntflags, pathname_t *realpnp)
1160 {
1161         znode_t *zdp = VTOZ(dvp);
1162         zfsvfs_t *zfsvfs = zdp->z_zfsvfs;
1163         int     error = 0;

1165         /* fast path */
1166         if (!(flags & (LOOKUP_XATTR | FIGNORECASE))) {

1168                 if (dvp->v_type != VDIR) {
1169                         return (SET_ERROR(ENOTDIR));
1170                 } else if (zdp->z_sa_hdl == NULL) {
1171                         return (SET_ERROR(EIO));
1172                 }

1174                 if (nm[0] == 0 || (nm[0] == '.' && nm[1] == '\0')) {
1175                         error = zfs_fastaccesschk_execute(zdp, cr);
1176                         if (!error) {
1177                                 *vpp = dvp;
1178                                 VN_HOLD(*vpp);
1179                                 return (0);
1180                         }
1181                         return (error);
1182                 } else {
1183                         vnode_t *tvp = dnlc_lookup(dvp, nm);

1185                         if (tvp) {
1186                                 error = zfs_fastaccesschk_execute(zdp, cr);
1187                                 if (error) {
1188                                         VN_RELE(tvp);
1189                                         return (error);
```



```
1190                                 }
1191                                 if (tvp == DNLC_NO_VNODE) {
1192                                         VN_RELE(tvp);
1193                                         return (SET_ERROR(ENOENT));
1194                                 } else {
1195                                         *vpp = tvp;
1196                                         return (specvp_check(vpp, cr));
1197                                 }
1198                         }
1199                 }
1200         }

1202         DTRACE_PROBE2(zfs__fastpath__lookup__miss, vnode_t *, dvp, char *, nm);

1204         ZFS_ENTER(zfsvfs);
1205         ZFS_VERIFY_ZP(zdp);

1207         *vpp = NULL;

1209         if (flags & LOOKUP_XATTR) {
1210                 /*
1211                  * If the xattr property is off, refuse the lookup request.
1212                  */
1213                 if (!(zfsvfs->z_vfs->vfs_flag & VFS_XATTR)) {
1214                         ZFS_EXIT(zfsvfs);
1215                         return (SET_ERROR(EINVAL));
1216                 }

1218                 /*
1219                  * We don't allow recursive attributes..
1220                  * Maybe someday we will.
1221                  */
1222                 if (zdp->z_pflags & ZFS_XATTR) {
1223                         ZFS_EXIT(zfsvfs);
1224                         return (SET_ERROR(EINVAL));
1225                 }

1227                 if (error = zfs_get_xattrdir(VTOZ(dvp), vpp, cr, flags)) {
1228                         ZFS_EXIT(zfsvfs);
1229                         return (error);
1230                 }

1232                 /*
1233                  * Do we have permission to get into attribute directory?
1234                  */

1236                 if (error = zfs_zaccess(VTOZ(*vpp), ACE_EXECUTE, 0,
1237                     B_FALSE, cr)) {
1238                         VN_RELE(*vpp);
1239                         *vpp = NULL;
1240                 }

1242                 ZFS_EXIT(zfsvfs);
1243                 return (error);
1244         }

1246         if (dvp->v_type != VDIR) {
1247                 ZFS_EXIT(zfsvfs);
1248                 return (SET_ERROR(ENOTDIR));
1249         }

1251         /*
1252          * Check accessibility of directory.
1253          */

1255         if (error = zfs_zaccess(zdp, ACE_EXECUTE, 0, B_FALSE, cr)) {
```

```
1256                 ZFS_EXIT(zfsvfs);
1257                 return (error);
1258         }

1260         if (zfsvfs->z_utf8 && u8_validate(nm, strlen(nm),
1261             NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
1262                 ZFS_EXIT(zfsvfs);
1263                 return (SET_ERROR(EILSEQ));
1264         }

1266         error = zfs_dirlook(zdp, nm, vpp, flags, direntflags, realpnp);
1267         if (error == 0)
1268                 error = specvp_check(vpp, cr);

1270         ZFS_EXIT(zfsvfs);
1271         return (error);
1272 }

1274 /*
1275  * Attempt to create a new entry in a directory.  If the entry
1276  * already exists, truncate the file if permissible, else return
1277  * an error.  Return the vp of the created or trunc'd file.
1278  *
1279  *      IN:     dvp     - vnode of directory to put new file entry in.
1280  *              name    - name of new file entry.
1281  *              vap     - attributes of new file.
1282  *              excl    - flag indicating exclusive or non-exclusive mode.
1283  *              mode    - mode to open file with.
1284  *              cr      - credentials of caller.
1285  *              flag    - large file flag [UNUSED].
1286  *              ct      - caller context
1287  *              vsecp   - ACL to be set
1288  *
1289  *      OUT:    vpp     - vnode of created or trunc'd entry.
1290  *
1291  *      RETURN: 0 on success, error code on failure.
1293  *      RETURN: 0 if success
1294  *              error code if failure
1292  *
1293  * Timestamps:
1294  *      dvp - ctime|mtime updated if new entry created
1295  *       vp - ctime|mtime always, atime if new
1296  */

1298 /* ARGSUSED */
1299 static int
1300 zfs_create(vnode_t *dvp, char *name, vattr_t *vap, vcexcl_t excl,
1301     int mode, vnode_t **vpp, cred_t *cr, int flag, caller_context_t *ct,
1302     vsecattr_t *vsecp)
1303 {
1304         znode_t         *zp, *dzp = VTOZ(dvp);
1305         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1306         zilog_t         *zilog;
1307         objset_t        *os;
1308         zfs_dirlock_t   *dl;
1309         dmu_tx_t        *tx;
1310         int             error;
1311         ksid_t          *ksid;
1312         uid_t           uid;
1313         gid_t           gid = crgetgid(cr);
1314         zfs_acl_ids_t   acl_ids;
1315         boolean_t       fuid_dirtied;
1316         boolean_t       have_acl = B_FALSE;

1318         /*
1319          * If we have an ephemeral id, ACL, or XVATTR then
```



```
1320          * make sure file system is at proper version
1321          */

1323         ksid = crgetsid(cr, KSID_OWNER);
1324         if (ksid)
1325                 uid = ksid_getid(ksid);
1326         else
1327                 uid = crgetuid(cr);

1329         if (zfsvfs->z_use_fuids == B_FALSE &&
1330             (vsecp || (vap->va_mask & AT_XVATTR) ||
1331             IS_EPHEMERAL(uid) || IS_EPHEMERAL(gid)))
1332                 return (SET_ERROR(EINVAL));

1334         ZFS_ENTER(zfsvfs);
1335         ZFS_VERIFY_ZP(dzp);
1336         os = zfsvfs->z_os;
1337         zilog = zfsvfs->z_log;

1339         if (zfsvfs->z_utf8 && u8_validate(name, strlen(name),
1340             NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
1341                 ZFS_EXIT(zfsvfs);
1342                 return (SET_ERROR(EILSEQ));
1343         }

1345         if (vap->va_mask & AT_XVATTR) {
1346                 if ((error = secpolicy_xvattr((xvattr_t *)vap,
1347                     crgetuid(cr), cr, vap->va_type)) != 0) {
1348                         ZFS_EXIT(zfsvfs);
1349                         return (error);
1350                 }
1351         }
1352 top:
1353         *vpp = NULL;

1355         if ((vap->va_mode & VSVTX) && secpolicy_vnode_stky_modify(cr))
1356                 vap->va_mode &= ~VSVTX;

1358         if (*name == '\0') {
1359                 /*
1360                  * Null component name refers to the directory itself.
1361                  */
1362                 VN_HOLD(dvp);
1363                 zp = dzp;
1364                 dl = NULL;
1365                 error = 0;
1366         } else {
1367                 /* possible VN_HOLD(zp) */
1368                 int zflg = 0;

1370                 if (flag & FIGNORECASE)
1371                         zflg |= ZCILOOK;

1373                 error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
1374                     NULL, NULL);
1375                 if (error) {
1376                         if (have_acl)
1377                                 zfs_acl_ids_free(&acl_ids);
1378                         if (strcmp(name, "..") == 0)
1379                                 error = SET_ERROR(EISDIR);
1380                         ZFS_EXIT(zfsvfs);
1381                         return (error);
1382                 }
1383         }

1385         if (zp == NULL) {
```

```
1386                uint64_t txtype;

1388                /*
1389                 * Create a new file object and update the directory
1390                 * to reference it.
1391                 */
1392                if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
1393                        if (have_acl)
1394                                zfs_acl_ids_free(&acl_ids);
1395                        goto out;
1396                }

1398                /*
1399                 * We only support the creation of regular files in
1400                 * extended attribute directories.
1401                 */

1403                if ((dzp->z_pflags & ZFS_XATTR) &&
1404                    (vap->va_type != VREG)) {
1405                        if (have_acl)
1406                                zfs_acl_ids_free(&acl_ids);
1407                        error = SET_ERROR(EINVAL);
1408                        goto out;
1409                }

1411                if (!have_acl && (error = zfs_acl_ids_create(dzp, 0, vap,
1412                    cr, vsecp, &acl_ids)) != 0)
1413                        goto out;
1414                have_acl = B_TRUE;

1416                if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
1417                        zfs_acl_ids_free(&acl_ids);
1418                        error = SET_ERROR(EDQUOT);
1419                        goto out;
1420                }

1422                tx = dmu_tx_create(os);

1424                dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
1425                    ZFS_SA_BASE_ATTR_SIZE);

1427                fuid_dirtied = zfsvfs->z_fuid_dirty;
1428                if (fuid_dirtied)
1429                        zfs_fuid_txhold(zfsvfs, tx);
1430                dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
1431                dmu_tx_hold_sa(tx, dzp->z_sa_hdl, B_FALSE);
1432                if (!zfsvfs->z_use_sa &&
1433                    acl_ids.z_aclp->z_acl_bytes > ZFS_ACE_SPACE) {
1434                        dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
1435                            0, acl_ids.z_aclp->z_acl_bytes);
1436                }
1437                error = dmu_tx_assign(tx, TXG_NOWAIT);
1438                if (error) {
1439                        zfs_dirent_unlock(dl);
1440                        if (error == ERESTART) {
1441                                dmu_tx_wait(tx);
1442                                dmu_tx_abort(tx);
1443                                goto top;
1444                        }
1445                        zfs_acl_ids_free(&acl_ids);
1446                        dmu_tx_abort(tx);
1447                        ZFS_EXIT(zfsvfs);
1448                        return (error);
1449                }
1450                zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids);
```



```
1452                if (fuid_dirtied)
1453                        zfs_fuid_sync(zfsvfs, tx);

1455                (void) zfs_link_create(dl, zp, tx, ZNEW);
1456                txtype = zfs_log_create_txtype(Z_FILE, vsecp, vap);
1457                if (flag & FIGNORECASE)
1458                        txtype |= TX_CI;
1459                zfs_log_create(zilog, tx, txtype, dzp, zp, name,
1460                    vsecp, acl_ids.z_fuidp, vap);
1461                zfs_acl_ids_free(&acl_ids);
1462                dmu_tx_commit(tx);
1463        } else {
1464                int aflags = (flag & FAPPEND) ? V_APPEND : 0;

1466                if (have_acl)
1467                        zfs_acl_ids_free(&acl_ids);
1468                have_acl = B_FALSE;

1470                /*
1471                 * A directory entry already exists for this name.
1472                 */
1473                /*
1474                 * Can't truncate an existing file if in exclusive mode.
1475                 */
1476                if (excl == EXCL) {
1477                        error = SET_ERROR(EEXIST);
1478                        goto out;
1479                }
1480                /*
1481                 * Can't open a directory for writing.
1482                 */
1483                if ((ZTOV(zp)->v_type == VDIR) && (mode & S_IWRITE)) {
1484                        error = SET_ERROR(EISDIR);
1485                        goto out;
1486                }
1487                /*
1488                 * Verify requested access to file.
1489                 */
1490                if (mode && (error = zfs_zaccess_rwx(zp, mode, aflags, cr))) {
1491                        goto out;
1492                }

1494                mutex_enter(&dzp->z_lock);
1495                dzp->z_seq++;
1496                mutex_exit(&dzp->z_lock);

1498                /*
1499                 * Truncate regular files if requested.
1500                 */
1501                if ((ZTOV(zp)->v_type == VREG) &&
1502                    (vap->va_mask & AT_SIZE) && (vap->va_size == 0)) {
1503                        /* we can't hold any locks when calling zfs_freesp() */
1504                        zfs_dirent_unlock(dl);
1505                        dl = NULL;
1506                        error = zfs_freesp(zp, 0, 0, mode, TRUE);
1507                        if (error == 0) {
1508                                vnevent_create(ZTOV(zp), ct);
1509                        }
1510                }
1511        }
1512 out:

1514        if (dl)
1515                zfs_dirent_unlock(dl);

1517        if (error) {
```

```
1518                 if (zp)
1519                         VN_RELE(ZTOV(zp));
1520         } else {
1521                 *vpp = ZTOV(zp);
1522                 error = specvp_check(vpp, cr);
1523         }

1525         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
1526                 zil_commit(zilog, 0);

1528         ZFS_EXIT(zfsvfs);
1529         return (error);
1530 }

1532 /*
1533  * Remove an entry from a directory.
1534  *
1535  *      IN:     dvp     - vnode of directory to remove entry from.
1536  *              name    - name of entry to remove.
1537  *              cr      - credentials of caller.
1538  *              ct      - caller context
1539  *              flags   - case flags
1540  *
1541  *      RETURN: 0 on success, error code on failure.
1544  *      RETURN: 0 if success
1545  *              error code if failure
1542  *
1543  * Timestamps:
1544  *      dvp - ctime|mtime
1545  *       vp - ctime (if nlink > 0)
1546  */

1548 uint64_t null_xattr = 0;

1550 /*ARGSUSED*/
1551 static int
1552 zfs_remove(vnode_t *dvp, char *name, cred_t *cr, caller_context_t *ct,
1553     int flags)
1554 {
1555         znode_t         *zp, *dzp = VTOZ(dvp);
1556         znode_t         *xzp;
1557         vnode_t         *vp;
1558         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1559         zilog_t         *zilog;
1560         uint64_t        acl_obj, xattr_obj;
1561         uint64_t        xattr_obj_unlinked = 0;
1562         uint64_t        obj = 0;
1563         zfs_dirlock_t   *dl;
1564         dmu_tx_t        *tx;
1565         boolean_t       may_delete_now, delete_now = FALSE;
1566         boolean_t       unlinked, toobig = FALSE;
1567         uint64_t        txtype;
1568         pathname_t      *realnmp = NULL;
1569         pathname_t      realnm;
1570         int             error;
1571         int             zflg = ZEXISTS;

1573         ZFS_ENTER(zfsvfs);
1574         ZFS_VERIFY_ZP(dzp);
1575         zilog = zfsvfs->z_log;

1577         if (flags & FIGNORECASE) {
1578                 zflg |= ZCILOOK;
1579                 pn_alloc(&realnm);
1580                 realnmp = &realnm;
1581         }
```



```
1583 top:
1584         xattr_obj = 0;
1585         xzp = NULL;
1586         /*
1587          * Attempt to lock directory; fail if entry doesn't exist.
1588          */
1589         if (error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
1590             NULL, realnmp)) {
1591                 if (realnmp)
1592                         pn_free(realnmp);
1593                 ZFS_EXIT(zfsvfs);
1594                 return (error);
1595         }

1597         vp = ZTOV(zp);

1599         if (error = zfs_zaccess_delete(dzp, zp, cr)) {
1600                 goto out;
1601         }

1603         /*
1604          * Need to use rmdir for removing directories.
1605          */
1606         if (vp->v_type == VDIR) {
1607                 error = SET_ERROR(EPERM);
1608                 goto out;
1609         }

1611         vnevent_remove(vp, dvp, name, ct);

1613         if (realnmp)
1614                 dnlc_remove(dvp, realnmp->pn_buf);
1615         else
1616                 dnlc_remove(dvp, name);

1618         mutex_enter(&vp->v_lock);
1619         may_delete_now = vp->v_count == 1 && !vn_has_cached_data(vp);
1620         mutex_exit(&vp->v_lock);

1622         /*
1623          * We may delete the znode now, or we may put it in the unlinked set;
1624          * it depends on whether we're the last link, and on whether there are
1625          * other holds on the vnode.  So we dmu_tx_hold() the right things to
1626          * allow for either case.
1627          */
1628         obj = zp->z_id;
1629         tx = dmu_tx_create(zfsvfs->z_os);
1630         dmu_tx_hold_zap(tx, dzp->z_id, FALSE, name);
1631         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1632         zfs_sa_upgrade_txholds(tx, zp);
1633         zfs_sa_upgrade_txholds(tx, dzp);
1634         if (may_delete_now) {
1635                 toobig =
1636                     zp->z_size > zp->z_blksz * DMU_MAX_DELETEBLKCNT;
1637                 /* if the file is too big, only hold_free a token amount */
1638                 dmu_tx_hold_free(tx, zp->z_id, 0,
1639                     (toobig ? DMU_MAX_ACCESS : DMU_OBJECT_END));
1640         }

1642         /* are there any extended attributes? */
1643         error = sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
1644             &xattr_obj, sizeof (xattr_obj));
1645         if (error == 0 && xattr_obj) {
1646                 error = zfs_zget(zfsvfs, xattr_obj, &xzp);
1647                 ASSERT0(error);
```

```
1648                 dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
1649                 dmu_tx_hold_sa(tx, xzp->z_sa_hdl, B_FALSE);
1650         }

1652         mutex_enter(&zp->z_lock);
1653         if ((acl_obj = zfs_external_acl(zp)) != 0 && may_delete_now)
1654                 dmu_tx_hold_free(tx, acl_obj, 0, DMU_OBJECT_END);
1655         mutex_exit(&zp->z_lock);

1657         /* charge as an update -- would be nice not to charge at all */
1658         dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);

1660         error = dmu_tx_assign(tx, TXG_NOWAIT);
1661         if (error) {
1662                 zfs_dirent_unlock(dl);
1663                 VN_RELE(vp);
1664                 if (xzp)
1665                         VN_RELE(ZTOV(xzp));
1666                 if (error == ERESTART) {
1667                         dmu_tx_wait(tx);
1668                         dmu_tx_abort(tx);
1669                         goto top;
1670                 }
1671                 if (realnmp)
1672                         pn_free(realnmp);
1673                 dmu_tx_abort(tx);
1674                 ZFS_EXIT(zfsvfs);
1675                 return (error);
1676         }

1678         /*
1679          * Remove the directory entry.
1680          */
1681         error = zfs_link_destroy(dl, zp, tx, zflg, &unlinked);

1683         if (error) {
1684                 dmu_tx_commit(tx);
1685                 goto out;
1686         }

1688         if (unlinked) {

1690                 /*
1691                  * Hold z_lock so that we can make sure that the ACL obj
1692                  * hasn't changed.  Could have been deleted due to
1693                  * zfs_sa_upgrade().
1694                  */
1695                 mutex_enter(&zp->z_lock);
1696                 mutex_enter(&vp->v_lock);
1697                 (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
1698                     &xattr_obj_unlinked, sizeof (xattr_obj_unlinked));
1699                 delete_now = may_delete_now && !toobig &&
1700                     vp->v_count == 1 && !vn_has_cached_data(vp) &&
1701                     xattr_obj == xattr_obj_unlinked && zfs_external_acl(zp) ==
1702                     acl_obj;
1703                 mutex_exit(&vp->v_lock);
1704         }

1706         if (delete_now) {
1707                 if (xattr_obj_unlinked) {
1708                         ASSERT3U(xzp->z_links, ==, 2);
1709                         mutex_enter(&xzp->z_lock);
1710                         xzp->z_unlinked = 1;
1711                         xzp->z_links = 0;
1712                         error = sa_update(xzp->z_sa_hdl, SA_ZPL_LINKS(zfsvfs),
1713                             &xzp->z_links, sizeof (xzp->z_links), tx);
```



```
1714                         ASSERT3U(error,  ==,  0);
1715                         mutex_exit(&xzp->z_lock);
1716                         zfs_unlinked_add(xzp, tx);

1718                         if (zp->z_is_sa)
1719                                 error = sa_remove(zp->z_sa_hdl,
1720                                     SA_ZPL_XATTR(zfsvfs), tx);
1721                         else
1722                                 error = sa_update(zp->z_sa_hdl,
1723                                     SA_ZPL_XATTR(zfsvfs), &null_xattr,
1724                                     sizeof (uint64_t), tx);
1725                         ASSERT0(error);
1726                 }
1727                 mutex_enter(&vp->v_lock);
1728                 vp->v_count--;
1729                 ASSERT0(vp->v_count);
1730                 mutex_exit(&vp->v_lock);
1731                 mutex_exit(&zp->z_lock);
1732                 zfs_znode_delete(zp, tx);
1733         } else if (unlinked) {
1734                 mutex_exit(&zp->z_lock);
1735                 zfs_unlinked_add(zp, tx);
1736         }

1738         txtype = TX_REMOVE;
1739         if (flags & FIGNORECASE)
1740                 txtype |= TX_CI;
1741         zfs_log_remove(zilog, tx, txtype, dzp, name, obj);

1743         dmu_tx_commit(tx);
1744 out:
1745         if (realnmp)
1746                 pn_free(realnmp);

1748         zfs_dirent_unlock(dl);

1750         if (!delete_now)
1751                 VN_RELE(vp);
1752         if (xzp)
1753                 VN_RELE(ZTOV(xzp));

1755         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
1756                 zil_commit(zilog, 0);

1758         ZFS_EXIT(zfsvfs);
1759         return (error);
1760 }

1762 /*
1763  * Create a new directory and insert it into dvp using the name
1764  * provided.  Return a pointer to the inserted directory.
1765  *
1766  *      IN:     dvp     - vnode of directory to add subdir to.
1767  *              dirname - name of new directory.
1768  *              vap     - attributes of new directory.
1769  *              cr      - credentials of caller.
1770  *              ct      - caller context
1771  *              flags   - case flags
1772 #endif /* ! codereview */
1773  *              vsecp   - ACL to be set
1774  *
1775  *      OUT:    vpp     - vnode of created directory.
1776  *
1777  *      RETURN: 0 on success, error code on failure.
1775  *      RETURN: 0 if success
1776  *              error code if failure
```

```
1778  *
1779  * Timestamps:
1780  *      dvp - ctime|mtime updated
1781  *       vp - ctime|mtime|atime updated
1782  */
1783 /*ARGSUSED*/
1784 static int
1785 zfs_mkdir(vnode_t *dvp, char *dirname, vattr_t *vap, vnode_t **vpp, cred_t *cr,
1786     caller_context_t *ct, int flags, vsecattr_t *vsecp)
1787 {
1788         znode_t         *zp, *dzp = VTOZ(dvp);
1789         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1790         zilog_t         *zilog;
1791         zfs_dirlock_t   *dl;
1792         uint64_t        txtype;
1793         dmu_tx_t        *tx;
1794         int             error;
1795         int             zf = ZNEW;
1796         ksid_t          *ksid;
1797         uid_t           uid;
1798         gid_t           gid = crgetgid(cr);
1799         zfs_acl_ids_t   acl_ids;
1800         boolean_t       fuid_dirtied;

1802         ASSERT(vap->va_type == VDIR);

1804         /*
1805          * If we have an ephemeral id, ACL, or XVATTR then
1806          * make sure file system is at proper version
1807          */

1809         ksid = crgetsid(cr, KSID_OWNER);
1810         if (ksid)
1811                 uid = ksid_getid(ksid);
1812         else
1813                 uid = crgetuid(cr);
1814         if (zfsvfs->z_use_fuids == B_FALSE &&
1815             (vsecp || (vap->va_mask & AT_XVATTR) ||
1816             IS_EPHEMERAL(uid) || IS_EPHEMERAL(gid)))
1817                 return (SET_ERROR(EINVAL));

1819         ZFS_ENTER(zfsvfs);
1820         ZFS_VERIFY_ZP(dzp);
1821         zilog = zfsvfs->z_log;

1823         if (dzp->z_pflags & ZFS_XATTR) {
1824                 ZFS_EXIT(zfsvfs);
1825                 return (SET_ERROR(EINVAL));
1826         }

1828         if (zfsvfs->z_utf8 && u8_validate(dirname,
1829             strlen(dirname), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
1830                 ZFS_EXIT(zfsvfs);
1831                 return (SET_ERROR(EILSEQ));
1832         }
1833         if (flags & FIGNORECASE)
1834                 zf |= ZCILOOK;

1836         if (vap->va_mask & AT_XVATTR) {
1837                 if ((error = secpolicy_xvattr((xvattr_t *)vap,
1838                     crgetuid(cr), cr, vap->va_type)) != 0) {
1839                         ZFS_EXIT(zfsvfs);
1840                         return (error);
1841                 }
1842         }
```



```
1844         if ((error = zfs_acl_ids_create(dzp, 0, vap, cr,
1845             vsecp, &acl_ids)) != 0) {
1846                 ZFS_EXIT(zfsvfs);
1847                 return (error);
1848         }
1849         /*
1850          * First make sure the new directory doesn't exist.
1851          *
1852          * Existence is checked first to make sure we don't return
1853          * EACCES instead of EEXIST which can cause some applications
1854          * to fail.
1855          */
1856 top:
1857         *vpp = NULL;

1859         if (error = zfs_dirent_lock(&dl, dzp, dirname, &zp, zf,
1860             NULL, NULL)) {
1861                 zfs_acl_ids_free(&acl_ids);
1862                 ZFS_EXIT(zfsvfs);
1863                 return (error);
1864         }

1866         if (error = zfs_zaccess(dzp, ACE_ADD_SUBDIRECTORY, 0, B_FALSE, cr)) {
1867                 zfs_acl_ids_free(&acl_ids);
1868                 zfs_dirent_unlock(dl);
1869                 ZFS_EXIT(zfsvfs);
1870                 return (error);
1871         }

1873         if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
1874                 zfs_acl_ids_free(&acl_ids);
1875                 zfs_dirent_unlock(dl);
1876                 ZFS_EXIT(zfsvfs);
1877                 return (SET_ERROR(EDQUOT));
1878         }

1880         /*
1881          * Add a new entry to the directory.
1882          */
1883         tx = dmu_tx_create(zfsvfs->z_os);
1884         dmu_tx_hold_zap(tx, dzp->z_id, TRUE, dirname);
1885         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, FALSE, NULL);
1886         fuid_dirtied = zfsvfs->z_fuid_dirty;
1887         if (fuid_dirtied)
1888                 zfs_fuid_txhold(zfsvfs, tx);
1889         if (!zfsvfs->z_use_sa && acl_ids.z_aclp->z_acl_bytes > ZFS_ACE_SPACE) {
1890                 dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0,
1891                     acl_ids.z_aclp->z_acl_bytes);
1892         }

1894         dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
1895             ZFS_SA_BASE_ATTR_SIZE);

1897         error = dmu_tx_assign(tx, TXG_NOWAIT);
1898         if (error) {
1899                 zfs_dirent_unlock(dl);
1900                 if (error == ERESTART) {
1901                         dmu_tx_wait(tx);
1902                         dmu_tx_abort(tx);
1903                         goto top;
1904                 }
1905                 zfs_acl_ids_free(&acl_ids);
1906                 dmu_tx_abort(tx);
1907                 ZFS_EXIT(zfsvfs);
1908                 return (error);
1909         }
```

```
1911         /*
1912          * Create new node.
1913          */
1914         zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids);

1916         if (fuid_dirtied)
1917                 zfs_fuid_sync(zfsvfs, tx);

1919         /*
1920          * Now put new name in parent dir.
1921          */
1922         (void) zfs_link_create(dl, zp, tx, ZNEW);

1924         *vpp = ZTOV(zp);

1926         txtype = zfs_log_create_txtype(Z_DIR, vsecp, vap);
1927         if (flags & FIGNORECASE)
1928                 txtype |= TX_CI;
1929         zfs_log_create(zilog, tx, txtype, dzp, zp, dirname, vsecp,
1930             acl_ids.z_fuidp, vap);

1932         zfs_acl_ids_free(&acl_ids);

1934         dmu_tx_commit(tx);

1936         zfs_dirent_unlock(dl);

1938         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
1939                 zil_commit(zilog, 0);

1941         ZFS_EXIT(zfsvfs);
1942         return (0);
1943 }

1945 /*
1946  * Remove a directory subdir entry.  If the current working
1947  * directory is the same as the subdir to be removed, the
1948  * remove will fail.
1949  *
1950  *      IN:     dvp     - vnode of directory to remove from.
1951  *              name    - name of directory to be removed.
1952  *              cwd     - vnode of current working directory.
1953  *              cr      - credentials of caller.
1954  *              ct      - caller context
1955  *              flags   - case flags
1956  *
1957  *      RETURN: 0 on success, error code on failure.
1956  *      RETURN: 0 if success
1957  *              error code if failure
1958  *
1959  * Timestamps:
1960  *      dvp - ctime|mtime updated
1961  */
1962 /*ARGSUSED*/
1963 static int
1964 zfs_rmdir(vnode_t *dvp, char *name, vnode_t *cwd, cred_t *cr,
1965     caller_context_t *ct, int flags)
1966 {
1967         znode_t         *dzp = VTOZ(dvp);
1968         znode_t         *zp;
1969         vnode_t         *vp;
1970         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1971         zilog_t         *zilog;
1972         zfs_dirlock_t   *dl;
1973         dmu_tx_t        *tx;
```



```
1974         int             error;
1975         int             zflg = ZEXISTS;

1977         ZFS_ENTER(zfsvfs);
1978         ZFS_VERIFY_ZP(dzp);
1979         zilog = zfsvfs->z_log;

1981         if (flags & FIGNORECASE)
1982                 zflg |= ZCILOOK;
1983 top:
1984         zp = NULL;

1986         /*
1987          * Attempt to lock directory; fail if entry doesn't exist.
1988          */
1989         if (error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
1990             NULL, NULL)) {
1991                 ZFS_EXIT(zfsvfs);
1992                 return (error);
1993         }

1995         vp = ZTOV(zp);

1997         if (error = zfs_zaccess_delete(dzp, zp, cr)) {
1998                 goto out;
1999         }

2001         if (vp->v_type != VDIR) {
2002                 error = SET_ERROR(ENOTDIR);
2003                 goto out;
2004         }

2006         if (vp == cwd) {
2007                 error = SET_ERROR(EINVAL);
2008                 goto out;
2009         }

2011         vnevent_rmdir(vp, dvp, name, ct);

2013         /*
2014          * Grab a lock on the directory to make sure that noone is
2015          * trying to add (or lookup) entries while we are removing it.
2016          */
2017         rw_enter(&zp->z_name_lock, RW_WRITER);

2019         /*
2020          * Grab a lock on the parent pointer to make sure we play well
2021          * with the treewalk and directory rename code.
2022          */
2023         rw_enter(&zp->z_parent_lock, RW_WRITER);

2025         tx = dmu_tx_create(zfsvfs->z_os);
2026         dmu_tx_hold_zap(tx, dzp->z_id, FALSE, name);
2027         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
2028         dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
2029         zfs_sa_upgrade_txholds(tx, zp);
2030         zfs_sa_upgrade_txholds(tx, dzp);
2031         error = dmu_tx_assign(tx, TXG_NOWAIT);
2032         if (error) {
2033                 rw_exit(&zp->z_parent_lock);
2034                 rw_exit(&zp->z_name_lock);
2035                 zfs_dirent_unlock(dl);
2036                 VN_RELE(vp);
2037                 if (error == ERESTART) {
2038                         dmu_tx_wait(tx);
2039                         dmu_tx_abort(tx);
```

```
2040                        goto top;
2041                }
2042                dmu_tx_abort(tx);
2043                ZFS_EXIT(zfsvfs);
2044                return (error);
2045        }

2047        error = zfs_link_destroy(dl, zp, tx, zflg, NULL);

2049        if (error == 0) {
2050                uint64_t txtype = TX_RMDIR;
2051                if (flags & FIGNORECASE)
2052                        txtype |= TX_CI;
2053                zfs_log_remove(zilog, tx, txtype, dzp, name, ZFS_NO_OBJECT);
2054        }

2056        dmu_tx_commit(tx);

2058        rw_exit(&zp->z_parent_lock);
2059        rw_exit(&zp->z_name_lock);
2060 out:
2061        zfs_dirent_unlock(dl);

2063        VN_RELE(vp);

2065        if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
2066                zil_commit(zilog, 0);

2068        ZFS_EXIT(zfsvfs);
2069        return (error);
2070 }

2072 /*
2073  * Read as many directory entries as will fit into the provided
2074  * buffer from the given directory cursor position (specified in
2075  * the uio structure).
2075  * the uio structure.
2076  *
2077  *      IN:     vp      - vnode of directory to read.
2078  *              uio     - structure supplying read location, range info,
2079  *                        and return buffer.
2080  *              cr      - credentials of caller.
2081  *              ct      - caller context
2082  *              flags   - case flags
2083  *
2084  *      OUT:    uio     - updated offset and range, buffer filled.
2085  *              eofp    - set to true if end-of-file detected.
2086  *
2087  *      RETURN: 0 on success, error code on failure.
2087  *      RETURN: 0 if success
2088  *              error code if failure
2088  *
2089  * Timestamps:
2090  *      vp - atime updated
2091  *
2092  * Note that the low 4 bits of the cookie returned by zap is always zero.
2093  * This allows us to use the low range for "special" directory entries:
2094  * We use 0 for '.', and 1 for '..'.  If this is the root of the filesystem,
2095  * we use the offset 2 for the '.zfs' directory.
2096  */
2097 /* ARGSUSED */
2098 static int
2099 zfs_readdir(vnode_t *vp, uio_t *uio, cred_t *cr, int *eofp,
2100     caller_context_t *ct, int flags)
2101 {
2102         znode_t         *zp = VTOZ(vp);
```



```
2103         iovec_t         *iovp;
2104         edirent_t       *eodp;
2105         dirent64_t      *odp;
2106         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
2107         objset_t        *os;
2108         caddr_t         outbuf;
2109         size_t          bufsize;
2110         zap_cursor_t    zc;
2111         zap_attribute_t zap;
2112         uint_t          bytes_wanted;
2113         uint64_t        offset; /* must be unsigned; checks for < 1 */
2114         uint64_t        parent;
2115         int             local_eof;
2116         int             outcount;
2117         int             error;
2118         uint8_t         prefetch;
2119         boolean_t       check_sysattrs;

2121         ZFS_ENTER(zfsvfs);
2122         ZFS_VERIFY_ZP(zp);

2124         if ((error = sa_lookup(zp->z_sa_hdl, SA_ZPL_PARENT(zfsvfs),
2125             &parent, sizeof (parent))) != 0) {
2126                 ZFS_EXIT(zfsvfs);
2127                 return (error);
2128         }

2130         /*
2131          * If we are not given an eof variable,
2132          * use a local one.
2133          */
2134         if (eofp == NULL)
2135                 eofp = &local_eof;

2137         /*
2138          * Check for valid iov_len.
2139          */
2140         if (uio->uio_iov->iov_len <= 0) {
2141                 ZFS_EXIT(zfsvfs);
2142                 return (SET_ERROR(EINVAL));
2143         }

2145         /*
2146          * Quit if directory has been removed (posix)
2147          */
2148         if ((*eofp = zp->z_unlinked) != 0) {
2149                 ZFS_EXIT(zfsvfs);
2150                 return (0);
2151         }

2153         error = 0;
2154         os = zfsvfs->z_os;
2155         offset = uio->uio_loffset;
2156         prefetch = zp->z_zn_prefetch;

2158         /*
2159          * Initialize the iterator cursor.
2160          */
2161         if (offset <= 3) {
2162                 /*
2163                  * Start iteration from the beginning of the directory.
2164                  */
2165                 zap_cursor_init(&zc, os, zp->z_id);
2166         } else {
2167                 /*
2168                  * The offset is a serialized cursor.
```

```
2169                     */
2170                    zap_cursor_init_serialized(&zc, os, zp->z_id, offset);
2171            }

2173            /*
2174             * Get space to change directory entries into fs independent format.
2175             */
2176            iovp = uio->uio_iov;
2177            bytes_wanted = iovp->iov_len;
2178            if (uio->uio_segflg != UIO_SYSSPACE || uio->uio_iovcnt != 1) {
2179                    bufsize = bytes_wanted;
2180                    outbuf = kmem_alloc(bufsize, KM_SLEEP);
2181                    odp = (struct dirent64 *)outbuf;
2182            } else {
2183                    bufsize = bytes_wanted;
2184                    outbuf = NULL;
2185                    odp = (struct dirent64 *)iovp->iov_base;
2186            }
2187            eodp = (struct edirent *)odp;

2189            /*
2190             * If this VFS supports the system attribute view interface; and
2191             * we're looking at an extended attribute directory; and we care
2192             * about normalization conflicts on this vfs; then we must check
2193             * for normalization conflicts with the sysattr name space.
2194             */
2195            check_sysattrs = vfs_has_feature(vp->v_vfsp, VFSFT_SYSATTR_VIEWS) &&
2196                (vp->v_flag & V_XATTRDIR) && zfsvfs->z_norm &&
2197                (flags & V_RDDIR_ENTFLAGS);

2199            /*
2200             * Transform to file-system independent format
2201             */
2202            outcount = 0;
2203            while (outcount < bytes_wanted) {
2204                    ino64_t objnum;
2205                    ushort_t reclen;
2206                    off64_t *next = NULL;

2208                    /*
2209                     * Special case '.', '..', and '.zfs'.
2210                     */
2211                    if (offset == 0) {
2212                            (void) strcpy(zap.za_name, ".");
2213                            zap.za_normalization_conflict = 0;
2214                            objnum = zp->z_id;
2215                    } else if (offset == 1) {
2216                            (void) strcpy(zap.za_name, "..");
2217                            zap.za_normalization_conflict = 0;
2218                            objnum = parent;
2219                    } else if (offset == 2 && zfs_show_ctldir(zp)) {
2220                            (void) strcpy(zap.za_name, ZFS_CTLDIR_NAME);
2221                            zap.za_normalization_conflict = 0;
2222                            objnum = ZFSCTL_INO_ROOT;
2223                    } else {
2224                            /*
2225                             * Grab next entry.
2226                             */
2227                            if (error = zap_cursor_retrieve(&zc, &zap)) {
2228                                    if ((*eofp = (error == ENOENT)) != 0)
2229                                            break;
2230                                    else
2231                                            goto update;
2232                            }

2234                            if (zap.za_integer_length != 8 ||
```



```
2235                                zap.za_num_integers != 1) {
2236                                    cmn_err(CE_WARN, "zap_readdir: bad directory "
2237                                        "entry, obj = %lld, offset = %lld\n",
2238                                        (u_longlong_t)zp->z_id,
2239                                        (u_longlong_t)offset);
2240                                    error = SET_ERROR(ENXIO);
2241                                    goto update;
2242                            }

2244                            objnum = ZFS_DIRENT_OBJ(zap.za_first_integer);
2245                            /*
2246                             * MacOS X can extract the object type here such as:
2247                             * uint8_t type = ZFS_DIRENT_TYPE(zap.za_first_integer);
2248                             */

2250                            if (check_sysattrs && !zap.za_normalization_conflict) {
2251                                    zap.za_normalization_conflict =
2252                                        xattr_sysattr_casechk(zap.za_name);
2253                            }
2254                    }

2256                    if (flags & V_RDDIR_ACCFILTER) {
2257                            /*
2258                             * If we have no access at all, don't include
2259                             * this entry in the returned information
2260                             */
2261                            znode_t *ezp;
2262                            if (zfs_zget(zp->z_zfsvfs, objnum, &ezp) != 0)
2263                                    goto skip_entry;
2264                            if (!zfs_has_access(ezp, cr)) {
2265                                    VN_RELE(ZTOV(ezp));
2266                                    goto skip_entry;
2267                            }
2268                            VN_RELE(ZTOV(ezp));
2269                    }

2271                    if (flags & V_RDDIR_ENTFLAGS)
2272                            reclen = EDIRENT_RECLEN(strlen(zap.za_name));
2273                    else
2274                            reclen = DIRENT64_RECLEN(strlen(zap.za_name));

2276                    /*
2277                     * Will this entry fit in the buffer?
2278                     */
2279                    if (outcount + reclen > bufsize) {
2280                            /*
2281                             * Did we manage to fit anything in the buffer?
2282                             */
2283                            if (!outcount) {
2284                                    error = SET_ERROR(EINVAL);
2285                                    goto update;
2286                            }
2287                            break;
2288                    }
2289                    if (flags & V_RDDIR_ENTFLAGS) {
2290                            /*
2291                             * Add extended flag entry:
2292                             */
2293                            eodp->ed_ino = objnum;
2294                            eodp->ed_reclen = reclen;
2295                            /* NOTE: ed_off is the offset for the *next* entry */
2296                            next = &(eodp->ed_off);
2297                            eodp->ed_eflags = zap.za_normalization_conflict ?
2298                                ED_CASE_CONFLICT : 0;
2299                            (void) strncpy(eodp->ed_name, zap.za_name,
2300                                EDIRENT_NAMELEN(reclen));
```

```
2301                        eodp = (edirent_t *)((intptr_t)eodp + reclen);
2302                } else {
2303                        /*
2304                         * Add normal entry:
2305                         */
2306                        odp->d_ino = objnum;
2307                        odp->d_reclen = reclen;
2308                        /* NOTE: d_off is the offset for the *next* entry */
2309                        next = &(odp->d_off);
2310                        (void) strncpy(odp->d_name, zap.za_name,
2311                            DIRENT64_NAMELEN(reclen));
2312                        odp = (dirent64_t *)((intptr_t)odp + reclen);
2313                }
2314                outcount += reclen;

2316                ASSERT(outcount <= bufsize);

2318                /* Prefetch znode */
2319                if (prefetch)
2320                        dmu_prefetch(os, objnum, 0, 0);

2322        skip_entry:
2323                /*
2324                 * Move to the next entry, fill in the previous offset.
2325                 */
2326                if (offset > 2 || (offset == 2 && !zfs_show_ctldir(zp))) {
2327                        zap_cursor_advance(&zc);
2328                        offset = zap_cursor_serialize(&zc);
2329                } else {
2330                        offset += 1;
2331                }
2332                if (next)
2333                        *next = offset;
2334        }
2335        zp->z_zn_prefetch = B_FALSE; /* a lookup will re-enable pre-fetching */

2337        if (uio->uio_segflg == UIO_SYSSPACE && uio->uio_iovcnt == 1) {
2338                iovp->iov_base += outcount;
2339                iovp->iov_len -= outcount;
2340                uio->uio_resid -= outcount;
2341        } else if (error = uiomove(outbuf, (long)outcount, UIO_READ, uio)) {
2342                /*
2343                 * Reset the pointer.
2344                 */
2345                offset = uio->uio_loffset;
2346        }

2348 update:
2349        zap_cursor_fini(&zc);
2350        if (uio->uio_segflg != UIO_SYSSPACE || uio->uio_iovcnt != 1)
2351                kmem_free(outbuf, bufsize);

2353        if (error == ENOENT)
2354                error = 0;

2356        ZFS_ACCESSTIME_STAMP(zfsvfs, zp);

2358        uio->uio_loffset = offset;
2359        ZFS_EXIT(zfsvfs);
2360        return (error);
2361 }
```

*unchanged_portion_omitted_*

```
2393 /*
2394  * Get the requested file attributes and place them in the provided
```



```
2395  * vattr structure.
2396  *
2397  *      IN:     vp      - vnode of file.
2398  *              vap     - va_mask identifies requested attributes.
2399  *                        If AT_XVATTR set, then optional attrs are requested
2400  *              flags   - ATTR_NOACLCHECK (CIFS server context)
2401  *              cr      - credentials of caller.
2402  *              ct      - caller context
2403  *
2404  *      OUT:    vap     - attribute values.
2405  *
2406  *      RETURN: 0 (always succeeds).
2407  *      RETURN: 0 (always succeeds)
2407  */
2408 /* ARGSUSED */
2409 static int
2410 zfs_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2411     caller_context_t *ct)
2412 {
2413        znode_t *zp = VTOZ(vp);
2414        zfsvfs_t *zfsvfs = zp->z_zfsvfs;
2415        int     error = 0;
2416        uint64_t links;
2417        uint64_t mtime[2], ctime[2];
2418        xvattr_t *xvap = (xvattr_t *)vap;       /* vap may be an xvattr_t * */
2419        xoptattr_t *xoap = NULL;
2420        boolean_t skipaclchk = (flags & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;
2421        sa_bulk_attr_t bulk[2];
2422        int count = 0;

2424        ZFS_ENTER(zfsvfs);
2425        ZFS_VERIFY_ZP(zp);

2427        zfs_fuid_map_ids(zp, cr, &vap->va_uid, &vap->va_gid);

2429        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, &mtime, 16);
2430        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, &ctime, 16);

2432        if ((error = sa_bulk_lookup(zp->z_sa_hdl, bulk, count)) != 0) {
2433                ZFS_EXIT(zfsvfs);
2434                return (error);
2435        }

2437        /*
2438         * If ACL is trivial don't bother looking for ACE_READ_ATTRIBUTES.
2439         * Also, if we are the owner don't bother, since owner should
2440         * always be allowed to read basic attributes of file.
2441         */
2442        if (!(zp->z_pflags & ZFS_ACL_TRIVIAL) &&
2443            (vap->va_uid != crgetuid(cr))) {
2444                if (error = zfs_zaccess(zp, ACE_READ_ATTRIBUTES, 0,
2445                    skipaclchk, cr)) {
2446                        ZFS_EXIT(zfsvfs);
2447                        return (error);
2448                }
2449        }

2451        /*
2452         * Return all attributes.  It's cheaper to provide the answer
2453         * than to determine whether we were asked the question.
2454         */

2456        mutex_enter(&zp->z_lock);
2457        vap->va_type = vp->v_type;
2458        vap->va_mode = zp->z_mode & MODEMASK;
2459        vap->va_fsid = zp->z_zfsvfs->z_vfs->vfs_dev;
```

```
2460         vap->va_nodeid = zp->z_id;
2461         if ((vp->v_flag & VROOT) && zfs_show_ctldir(zp))
2462                 links = zp->z_links + 1;
2463         else
2464                 links = zp->z_links;
2465         vap->va_nlink = MIN(links, UINT32_MAX); /* nlink_t limit! */
2466         vap->va_size = zp->z_size;
2467         vap->va_rdev = vp->v_rdev;
2468         vap->va_seq = zp->z_seq;

2470         /*
2471          * Add in any requested optional attributes and the create time.
2472          * Also set the corresponding bits in the returned attribute bitmap.
2473          */
2474         if ((xoap = xva_getxoptattr(xvap)) != NULL && zfsvfs->z_use_fuids) {
2475                 if (XVA_ISSET_REQ(xvap, XAT_ARCHIVE)) {
2476                         xoap->xoa_archive =
2477                             ((zp->z_pflags & ZFS_ARCHIVE) != 0);
2478                         XVA_SET_RTN(xvap, XAT_ARCHIVE);
2479                 }

2481                 if (XVA_ISSET_REQ(xvap, XAT_READONLY)) {
2482                         xoap->xoa_readonly =
2483                             ((zp->z_pflags & ZFS_READONLY) != 0);
2484                         XVA_SET_RTN(xvap, XAT_READONLY);
2485                 }

2487                 if (XVA_ISSET_REQ(xvap, XAT_SYSTEM)) {
2488                         xoap->xoa_system =
2489                             ((zp->z_pflags & ZFS_SYSTEM) != 0);
2490                         XVA_SET_RTN(xvap, XAT_SYSTEM);
2491                 }

2493                 if (XVA_ISSET_REQ(xvap, XAT_HIDDEN)) {
2494                         xoap->xoa_hidden =
2495                             ((zp->z_pflags & ZFS_HIDDEN) != 0);
2496                         XVA_SET_RTN(xvap, XAT_HIDDEN);
2497                 }

2499                 if (XVA_ISSET_REQ(xvap, XAT_NOUNLINK)) {
2500                         xoap->xoa_nounlink =
2501                             ((zp->z_pflags & ZFS_NOUNLINK) != 0);
2502                         XVA_SET_RTN(xvap, XAT_NOUNLINK);
2503                 }

2505                 if (XVA_ISSET_REQ(xvap, XAT_IMMUTABLE)) {
2506                         xoap->xoa_immutable =
2507                             ((zp->z_pflags & ZFS_IMMUTABLE) != 0);
2508                         XVA_SET_RTN(xvap, XAT_IMMUTABLE);
2509                 }

2511                 if (XVA_ISSET_REQ(xvap, XAT_APPENDONLY)) {
2512                         xoap->xoa_appendonly =
2513                             ((zp->z_pflags & ZFS_APPENDONLY) != 0);
2514                         XVA_SET_RTN(xvap, XAT_APPENDONLY);
2515                 }

2517                 if (XVA_ISSET_REQ(xvap, XAT_NODUMP)) {
2518                         xoap->xoa_nodump =
2519                             ((zp->z_pflags & ZFS_NODUMP) != 0);
2520                         XVA_SET_RTN(xvap, XAT_NODUMP);
2521                 }

2523                 if (XVA_ISSET_REQ(xvap, XAT_OPAQUE)) {
2524                         xoap->xoa_opaque =
2525                             ((zp->z_pflags & ZFS_OPAQUE) != 0);
```



```
2526                         XVA_SET_RTN(xvap, XAT_OPAQUE);
2527                 }

2529                 if (XVA_ISSET_REQ(xvap, XAT_AV_QUARANTINED)) {
2530                         xoap->xoa_av_quarantined =
2531                             ((zp->z_pflags & ZFS_AV_QUARANTINED) != 0);
2532                         XVA_SET_RTN(xvap, XAT_AV_QUARANTINED);
2533                 }

2535                 if (XVA_ISSET_REQ(xvap, XAT_AV_MODIFIED)) {
2536                         xoap->xoa_av_modified =
2537                             ((zp->z_pflags & ZFS_AV_MODIFIED) != 0);
2538                         XVA_SET_RTN(xvap, XAT_AV_MODIFIED);
2539                 }

2541                 if (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP) &&
2542                     vp->v_type == VREG) {
2543                         zfs_sa_get_scanstamp(zp, xvap);
2544                 }

2546                 if (XVA_ISSET_REQ(xvap, XAT_CREATETIME)) {
2547                         uint64_t times[2];

2549                         (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_CRTIME(zfsvfs),
2550                             times, sizeof (times));
2551                         ZFS_TIME_DECODE(&xoap->xoa_createtime, times);
2552                         XVA_SET_RTN(xvap, XAT_CREATETIME);
2553                 }

2555                 if (XVA_ISSET_REQ(xvap, XAT_REPARSE)) {
2556                         xoap->xoa_reparse = ((zp->z_pflags & ZFS_REPARSE) != 0);
2557                         XVA_SET_RTN(xvap, XAT_REPARSE);
2558                 }
2559                 if (XVA_ISSET_REQ(xvap, XAT_GEN)) {
2560                         xoap->xoa_generation = zp->z_gen;
2561                         XVA_SET_RTN(xvap, XAT_GEN);
2562                 }

2564                 if (XVA_ISSET_REQ(xvap, XAT_OFFLINE)) {
2565                         xoap->xoa_offline =
2566                             ((zp->z_pflags & ZFS_OFFLINE) != 0);
2567                         XVA_SET_RTN(xvap, XAT_OFFLINE);
2568                 }

2570                 if (XVA_ISSET_REQ(xvap, XAT_SPARSE)) {
2571                         xoap->xoa_sparse =
2572                             ((zp->z_pflags & ZFS_SPARSE) != 0);
2573                         XVA_SET_RTN(xvap, XAT_SPARSE);
2574                 }
2575         }

2577         ZFS_TIME_DECODE(&vap->va_atime, zp->z_atime);
2578         ZFS_TIME_DECODE(&vap->va_mtime, mtime);
2579         ZFS_TIME_DECODE(&vap->va_ctime, ctime);

2581         mutex_exit(&zp->z_lock);

2583         sa_object_size(zp->z_sa_hdl, &vap->va_blksize, &vap->va_nblocks);

2585         if (zp->z_blksz == 0) {
2586                 /*
2587                  * Block size hasn't been set; suggest maximal I/O transfers.
2588                  */
2589                 vap->va_blksize = zfsvfs->z_max_blksz;
2590         }
```

```
2592         ZFS_EXIT(zfsvfs);
2593         return (0);
2594 }

2596 /*
2597  * Set the file attributes to the values contained in the
2598  * vattr structure.
2599  *
2600  *      IN:     vp      - vnode of file to be modified.
2601  *              vap     - new attribute values.
2602  *                        If AT_XVATTR set, then optional attrs are being set
2603  *              flags   - ATTR_UTIME set if non-default time values provided.
2604  *                      - ATTR_NOACLCHECK (CIFS context only).
2605  *              cr      - credentials of caller.
2606  *              ct      - caller context
2607  *
2608  *      RETURN: 0 on success, error code on failure.
2609  *      RETURN: 0 if success
2610  *              error code if failure
2609  *
2610  * Timestamps:
2611  *      vp - ctime updated, mtime updated if size changed.
2612  */
2613 /* ARGSUSED */
2614 static int
2615 zfs_setattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2616     caller_context_t *ct)
2617 {
2618         znode_t         *zp = VTOZ(vp);
2619         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
2620         zilog_t         *zilog;
2621         dmu_tx_t        *tx;
2622         vattr_t         oldva;
2623         xvattr_t        tmpxvattr;
2624         uint_t          mask = vap->va_mask;
2625         uint_t          saved_mask = 0;
2626         int             trim_mask = 0;
2627         uint64_t        new_mode;
2628         uint64_t        new_uid, new_gid;
2629         uint64_t        xattr_obj;
2630         uint64_t        mtime[2], ctime[2];
2631         znode_t         *attrzp;
2632         int             need_policy = FALSE;
2633         int             err, err2;
2634         zfs_fuid_info_t *fuidp = NULL;
2635         xvattr_t *xvap = (xvattr_t *)vap;       /* vap may be an xvattr_t * */
2636         xoptattr_t      *xoap;
2637         zfs_acl_t       *aclp;
2638         boolean_t skipaclchk = (flags & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;
2639         boolean_t       fuid_dirtied = B_FALSE;
2640         sa_bulk_attr_t  bulk[7], xattr_bulk[7];
2641         int             count = 0, xattr_count = 0;

2643         if (mask == 0)
2644                 return (0);

2646         if (mask & AT_NOSET)
2647                 return (SET_ERROR(EINVAL));

2649         ZFS_ENTER(zfsvfs);
2650         ZFS_VERIFY_ZP(zp);

2652         zilog = zfsvfs->z_log;

2654         /*
2655          * Make sure that if we have ephemeral uid/gid or xvattr specified
```



```
2656          * that file system is at proper version level
2657          */

2659         if (zfsvfs->z_use_fuids == B_FALSE &&
2660             (((mask & AT_UID) && IS_EPHEMERAL(vap->va_uid)) ||
2661             ((mask & AT_GID) && IS_EPHEMERAL(vap->va_gid)) ||
2662             (mask & AT_XVATTR))) {
2663                 ZFS_EXIT(zfsvfs);
2664                 return (SET_ERROR(EINVAL));
2665         }

2667         if (mask & AT_SIZE && vp->v_type == VDIR) {
2668                 ZFS_EXIT(zfsvfs);
2669                 return (SET_ERROR(EISDIR));
2670         }

2672         if (mask & AT_SIZE && vp->v_type != VREG && vp->v_type != VFIFO) {
2673                 ZFS_EXIT(zfsvfs);
2674                 return (SET_ERROR(EINVAL));
2675         }

2677         /*
2678          * If this is an xvattr_t, then get a pointer to the structure of
2679          * optional attributes.  If this is NULL, then we have a vattr_t.
2680          */
2681         xoap = xva_getxoptattr(xvap);

2683         xva_init(&tmpxvattr);

2685         /*
2686          * Immutable files can only alter immutable bit and atime
2687          */
2688         if ((zp->z_pflags & ZFS_IMMUTABLE) &&
2689             ((mask & (AT_SIZE|AT_UID|AT_GID|AT_MTIME|AT_MODE)) ||
2690             ((mask & AT_XVATTR) && XVA_ISSET_REQ(xvap, XAT_CREATETIME)))) {
2691                 ZFS_EXIT(zfsvfs);
2692                 return (SET_ERROR(EPERM));
2693         }

2695         if ((mask & AT_SIZE) && (zp->z_pflags & ZFS_READONLY)) {
2696                 ZFS_EXIT(zfsvfs);
2697                 return (SET_ERROR(EPERM));
2698         }

2700         /*
2701          * Verify timestamps doesn't overflow 32 bits.
2702          * ZFS can handle large timestamps, but 32bit syscalls can't
2703          * handle times greater than 2039.  This check should be removed
2704          * once large timestamps are fully supported.
2705          */
2706         if (mask & (AT_ATIME | AT_MTIME)) {
2707                 if (((mask & AT_ATIME) && TIMESPEC_OVERFLOW(&vap->va_atime)) ||
2708                     ((mask & AT_MTIME) && TIMESPEC_OVERFLOW(&vap->va_mtime))) {
2709                         ZFS_EXIT(zfsvfs);
2710                         return (SET_ERROR(EOVERFLOW));
2711                 }
2712         }

2714 top:
2715         attrzp = NULL;
2716         aclp = NULL;

2718         /* Can this be moved to before the top label? */
2719         if (zfsvfs->z_vfs->vfs_flag & VFS_RDONLY) {
2720                 ZFS_EXIT(zfsvfs);
2721                 return (SET_ERROR(EROFS));
```

```
2722         }

2724         /*
2725          * First validate permissions
2726          */

2728         if (mask & AT_SIZE) {
2729                 err = zfs_zaccess(zp, ACE_WRITE_DATA, 0, skipaclchk, cr);
2730                 if (err) {
2731                         ZFS_EXIT(zfsvfs);
2732                         return (err);
2733                 }
2734                 /*
2735                  * XXX - Note, we are not providing any open
2736                  * mode flags here (like FNDELAY), so we may
2737                  * block if there are locks present... this
2738                  * should be addressed in openat().
2739                  */
2740                 /* XXX - would it be OK to generate a log record here? */
2741                 err = zfs_freesp(zp, vap->va_size, 0, 0, FALSE);
2742                 if (err) {
2743                         ZFS_EXIT(zfsvfs);
2744                         return (err);
2745                 }
2746         }

2748         if (mask & (AT_ATIME|AT_MTIME) ||
2749             ((mask & AT_XVATTR) && (XVA_ISSET_REQ(xvap, XAT_HIDDEN) ||
2750             XVA_ISSET_REQ(xvap, XAT_READONLY) ||
2751             XVA_ISSET_REQ(xvap, XAT_ARCHIVE) ||
2752             XVA_ISSET_REQ(xvap, XAT_OFFLINE) ||
2753             XVA_ISSET_REQ(xvap, XAT_SPARSE) ||
2754             XVA_ISSET_REQ(xvap, XAT_CREATETIME) ||
2755             XVA_ISSET_REQ(xvap, XAT_SYSTEM)))) {
2756                 need_policy = zfs_zaccess(zp, ACE_WRITE_ATTRIBUTES, 0,
2757                     skipaclchk, cr);
2758         }

2760         if (mask & (AT_UID|AT_GID)) {
2761                 int     idmask = (mask & (AT_UID|AT_GID));
2762                 int     take_owner;
2763                 int     take_group;

2765                 /*
2766                  * NOTE: even if a new mode is being set,
2767                  * we may clear S_ISUID/S_ISGID bits.
2768                  */

2770                 if (!(mask & AT_MODE))
2771                         vap->va_mode = zp->z_mode;

2773                 /*
2774                  * Take ownership or chgrp to group we are a member of
2775                  */

2777                 take_owner = (mask & AT_UID) && (vap->va_uid == crgetuid(cr));
2778                 take_group = (mask & AT_GID) &&
2779                     zfs_groupmember(zfsvfs, vap->va_gid, cr);

2781                 /*
2782                  * If both AT_UID and AT_GID are set then take_owner and
2783                  * take_group must both be set in order to allow taking
2784                  * ownership.
2785                  *
2786                  * Otherwise, send the check through secpolicy_vnode_setattr()
2787                  *
```



```
2788                  */

2790                 if (((idmask == (AT_UID|AT_GID)) && take_owner && take_group) ||
2791                     ((idmask == AT_UID) && take_owner) ||
2792                     ((idmask == AT_GID) && take_group)) {
2793                         if (zfs_zaccess(zp, ACE_WRITE_OWNER, 0,
2794                             skipaclchk, cr) == 0) {
2795                                 /*
2796                                  * Remove setuid/setgid for non-privileged users
2797                                  */
2798                                 secpolicy_setid_clear(vap, cr);
2799                                 trim_mask = (mask & (AT_UID|AT_GID));
2800                         } else {
2801                                 need_policy =  TRUE;
2802                         }
2803                 } else {
2804                         need_policy =  TRUE;
2805                 }
2806         }

2808         mutex_enter(&zp->z_lock);
2809         oldva.va_mode = zp->z_mode;
2810         zfs_fuid_map_ids(zp, cr, &oldva.va_uid, &oldva.va_gid);
2811         if (mask & AT_XVATTR) {
2812                 /*
2813                  * Update xvattr mask to include only those attributes
2814                  * that are actually changing.
2815                  *
2816                  * the bits will be restored prior to actually setting
2817                  * the attributes so the caller thinks they were set.
2818                  */
2819                 if (XVA_ISSET_REQ(xvap, XAT_APPENDONLY)) {
2820                         if (xoap->xoa_appendonly !=
2821                             ((zp->z_pflags & ZFS_APPENDONLY) != 0)) {
2822                                 need_policy = TRUE;
2823                         } else {
2824                                 XVA_CLR_REQ(xvap, XAT_APPENDONLY);
2825                                 XVA_SET_REQ(&tmpxvattr, XAT_APPENDONLY);
2826                         }
2827                 }

2829                 if (XVA_ISSET_REQ(xvap, XAT_NOUNLINK)) {
2830                         if (xoap->xoa_nounlink !=
2831                             ((zp->z_pflags & ZFS_NOUNLINK) != 0)) {
2832                                 need_policy = TRUE;
2833                         } else {
2834                                 XVA_CLR_REQ(xvap, XAT_NOUNLINK);
2835                                 XVA_SET_REQ(&tmpxvattr, XAT_NOUNLINK);
2836                         }
2837                 }

2839                 if (XVA_ISSET_REQ(xvap, XAT_IMMUTABLE)) {
2840                         if (xoap->xoa_immutable !=
2841                             ((zp->z_pflags & ZFS_IMMUTABLE) != 0)) {
2842                                 need_policy = TRUE;
2843                         } else {
2844                                 XVA_CLR_REQ(xvap, XAT_IMMUTABLE);
2845                                 XVA_SET_REQ(&tmpxvattr, XAT_IMMUTABLE);
2846                         }
2847                 }

2849                 if (XVA_ISSET_REQ(xvap, XAT_NODUMP)) {
2850                         if (xoap->xoa_nodump !=
2851                             ((zp->z_pflags & ZFS_NODUMP) != 0)) {
2852                                 need_policy = TRUE;
2853                         } else {
```

```
2854                                XVA_CLR_REQ(xvap, XAT_NODUMP);
2855                                XVA_SET_REQ(&tmpxvattr, XAT_NODUMP);
2856                        }
2857                }

2859                if (XVA_ISSET_REQ(xvap, XAT_AV_MODIFIED)) {
2860                        if (xoap->xoa_av_modified !=
2861                            ((zp->z_pflags & ZFS_AV_MODIFIED) != 0)) {
2862                                need_policy = TRUE;
2863                        } else {
2864                                XVA_CLR_REQ(xvap, XAT_AV_MODIFIED);
2865                                XVA_SET_REQ(&tmpxvattr, XAT_AV_MODIFIED);
2866                        }
2867                }

2869                if (XVA_ISSET_REQ(xvap, XAT_AV_QUARANTINED)) {
2870                        if ((vp->v_type != VREG &&
2871                            xoap->xoa_av_quarantined) ||
2872                            xoap->xoa_av_quarantined !=
2873                            ((zp->z_pflags & ZFS_AV_QUARANTINED) != 0)) {
2874                                need_policy = TRUE;
2875                        } else {
2876                                XVA_CLR_REQ(xvap, XAT_AV_QUARANTINED);
2877                                XVA_SET_REQ(&tmpxvattr, XAT_AV_QUARANTINED);
2878                        }
2879                }

2881                if (XVA_ISSET_REQ(xvap, XAT_REPARSE)) {
2882                        mutex_exit(&zp->z_lock);
2883                        ZFS_EXIT(zfsvfs);
2884                        return (SET_ERROR(EPERM));
2885                }

2887                if (need_policy == FALSE &&
2888                    (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP) ||
2889                    XVA_ISSET_REQ(xvap, XAT_OPAQUE))) {
2890                        need_policy = TRUE;
2891                }
2892        }

2894        mutex_exit(&zp->z_lock);

2896        if (mask & AT_MODE) {
2897                if (zfs_zaccess(zp, ACE_WRITE_ACL, 0, skipaclchk, cr) == 0) {
2898                        err = secpolicy_setid_setsticky_clear(vp, vap,
2899                            &oldva, cr);
2900                        if (err) {
2901                                ZFS_EXIT(zfsvfs);
2902                                return (err);
2903                        }
2904                        trim_mask |= AT_MODE;
2905                } else {
2906                        need_policy = TRUE;
2907                }
2908        }

2910        if (need_policy) {
2911                /*
2912                 * If trim_mask is set then take ownership
2913                 * has been granted or write_acl is present and user
2914                 * has the ability to modify mode.  In that case remove
2915                 * UID|GID and or MODE from mask so that
2916                 * secpolicy_vnode_setattr() doesn't revoke it.
2917                 */

2919                if (trim_mask) {
```



```
2920                        saved_mask = vap->va_mask;
2921                        vap->va_mask &= ~trim_mask;
2922                }
2923                err = secpolicy_vnode_setattr(cr, vp, vap, &oldva, flags,
2924                    (int (*)(void *, int, cred_t *))zfs_zaccess_unix, zp);
2925                if (err) {
2926                        ZFS_EXIT(zfsvfs);
2927                        return (err);
2928                }

2930                if (trim_mask)
2931                        vap->va_mask |= saved_mask;
2932        }

2934        /*
2935         * secpolicy_vnode_setattr, or take ownership may have
2936         * changed va_mask
2937         */
2938        mask = vap->va_mask;

2940        if ((mask & (AT_UID | AT_GID))) {
2941                err = sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
2942                    &xattr_obj, sizeof (xattr_obj));

2944                if (err == 0 && xattr_obj) {
2945                        err = zfs_zget(zp->z_zfsvfs, xattr_obj, &attrzp);
2946                        if (err)
2947                                goto out2;
2948                }
2949                if (mask & AT_UID) {
2950                        new_uid = zfs_fuid_create(zfsvfs,
2951                            (uint64_t)vap->va_uid, cr, ZFS_OWNER, &fuidp);
2952                        if (new_uid != zp->z_uid &&
2953                            zfs_fuid_overquota(zfsvfs, B_FALSE, new_uid)) {
2954                                if (attrzp)
2955                                        VN_RELE(ZTOV(attrzp));
2956                                err = SET_ERROR(EDQUOT);
2957                                goto out2;
2958                        }
2959                }

2961                if (mask & AT_GID) {
2962                        new_gid = zfs_fuid_create(zfsvfs, (uint64_t)vap->va_gid,
2963                            cr, ZFS_GROUP, &fuidp);
2964                        if (new_gid != zp->z_gid &&
2965                            zfs_fuid_overquota(zfsvfs, B_TRUE, new_gid)) {
2966                                if (attrzp)
2967                                        VN_RELE(ZTOV(attrzp));
2968                                err = SET_ERROR(EDQUOT);
2969                                goto out2;
2970                        }
2971                }
2972        }
2973        tx = dmu_tx_create(zfsvfs->z_os);

2975        if (mask & AT_MODE) {
2976                uint64_t pmode = zp->z_mode;
2977                uint64_t acl_obj;
2978                new_mode = (pmode & S_IFMT) | (vap->va_mode & ~S_IFMT);

2980                if (zp->z_zfsvfs->z_acl_mode == ZFS_ACL_RESTRICTED &&
2981                    !(zp->z_pflags & ZFS_ACL_TRIVIAL)) {
2982                        err = SET_ERROR(EPERM);
2983                        goto out;
2984                }
```

```
2986                if (err = zfs_acl_chmod_setattr(zp, &aclp, new_mode))
2987                        goto out;

2989                mutex_enter(&zp->z_lock);
2990                if (!zp->z_is_sa && ((acl_obj = zfs_external_acl(zp)) != 0)) {
2991                        /*
2992                         * Are we upgrading ACL from old V0 format
2993                         * to V1 format?
2994                         */
2995                        if (zfsvfs->z_version >= ZPL_VERSION_FUID &&
2996                            zfs_znode_acl_version(zp) ==
2997                            ZFS_ACL_VERSION_INITIAL) {
2998                                dmu_tx_hold_free(tx, acl_obj, 0,
2999                                    DMU_OBJECT_END);
3000                                dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
3001                                    0, aclp->z_acl_bytes);
3002                        } else {
3003                                dmu_tx_hold_write(tx, acl_obj, 0,
3004                                    aclp->z_acl_bytes);
3005                        }
3006                } else if (!zp->z_is_sa && aclp->z_acl_bytes > ZFS_ACE_SPACE) {
3007                        dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
3008                            0, aclp->z_acl_bytes);
3009                }
3010                mutex_exit(&zp->z_lock);
3011                dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
3012        } else {
3013                if ((mask & AT_XVATTR) &&
3014                    XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP))
3015                        dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
3016                else
3017                        dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
3018        }

3020        if (attrzp) {
3021                dmu_tx_hold_sa(tx, attrzp->z_sa_hdl, B_FALSE);
3022        }

3024        fuid_dirtied = zfsvfs->z_fuid_dirty;
3025        if (fuid_dirtied)
3026                zfs_fuid_txhold(zfsvfs, tx);

3028        zfs_sa_upgrade_txholds(tx, zp);

3030        err = dmu_tx_assign(tx, TXG_NOWAIT);
3031        if (err) {
3032                if (err == ERESTART)
3033                        dmu_tx_wait(tx);
3034                goto out;
3035        }

3037        count = 0;
3038        /*
3039         * Set each attribute requested.
3040         * We group settings according to the locks they need to acquire.
3041         *
3042         * Note: you cannot set ctime directly, although it will be
3043         * updated as a side-effect of calling this function.
3044         */


3047        if (mask & (AT_UID|AT_GID|AT_MODE))
3048                mutex_enter(&zp->z_acl_lock);
3049        mutex_enter(&zp->z_lock);

3051        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
```



```
3052            &zp->z_pflags, sizeof (zp->z_pflags));

3054        if (attrzp) {
3055                if (mask & (AT_UID|AT_GID|AT_MODE))
3056                        mutex_enter(&attrzp->z_acl_lock);
3057                mutex_enter(&attrzp->z_lock);
3058                SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3059                    SA_ZPL_FLAGS(zfsvfs), NULL, &attrzp->z_pflags,
3060                    sizeof (attrzp->z_pflags));
3061        }

3063        if (mask & (AT_UID|AT_GID)) {

3065                if (mask & AT_UID) {
3066                        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_UID(zfsvfs), NULL,
3067                            &new_uid, sizeof (new_uid));
3068                        zp->z_uid = new_uid;
3069                        if (attrzp) {
3070                                SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3071                                    SA_ZPL_UID(zfsvfs), NULL, &new_uid,
3072                                    sizeof (new_uid));
3073                                attrzp->z_uid = new_uid;
3074                        }
3075                }

3077                if (mask & AT_GID) {
3078                        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_GID(zfsvfs),
3079                            NULL, &new_gid, sizeof (new_gid));
3080                        zp->z_gid = new_gid;
3081                        if (attrzp) {
3082                                SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3083                                    SA_ZPL_GID(zfsvfs), NULL, &new_gid,
3084                                    sizeof (new_gid));
3085                                attrzp->z_gid = new_gid;
3086                        }
3087                }
3088                if (!(mask & AT_MODE)) {
3089                        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs),
3090                            NULL, &new_mode, sizeof (new_mode));
3091                        new_mode = zp->z_mode;
3092                }
3093                err = zfs_acl_chown_setattr(zp);
3094                ASSERT(err == 0);
3095                if (attrzp) {
3096                        err = zfs_acl_chown_setattr(attrzp);
3097                        ASSERT(err == 0);
3098                }
3099        }

3101        if (mask & AT_MODE) {
3102                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs), NULL,
3103                    &new_mode, sizeof (new_mode));
3104                zp->z_mode = new_mode;
3105                ASSERT3U((uintptr_t)aclp, !=, NULL);
3106                err = zfs_aclset_common(zp, aclp, cr, tx);
3107                ASSERT0(err);
3108                if (zp->z_acl_cached)
3109                        zfs_acl_free(zp->z_acl_cached);
3110                zp->z_acl_cached = aclp;
3111                aclp = NULL;
3112        }


3115        if (mask & AT_ATIME) {
3116                ZFS_TIME_ENCODE(&vap->va_atime, zp->z_atime);
3117                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_ATIME(zfsvfs), NULL,
```

```
3118                    &zp->z_atime, sizeof (zp->z_atime));
3119            }
3120
3121            if (mask & AT_MTIME) {
3122                    ZFS_TIME_ENCODE(&vap->va_mtime, mtime);
3123                    SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL,
3124                        mtime, sizeof (mtime));
3125            }
3126
3127            /* XXX - shouldn't this be done *before* the ATIME/MTIME checks? */
3128            if (mask & AT_SIZE && !(mask & AT_MTIME)) {
3129                    SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs),
3130                        NULL, mtime, sizeof (mtime));
3131                    SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
3132                        &ctime, sizeof (ctime));
3133                    zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
3134                        B_TRUE);
3135            } else if (mask != 0) {
3136                    SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
3137                        &ctime, sizeof (ctime));
3138                    zfs_tstamp_update_setup(zp, STATE_CHANGED, mtime, ctime,
3139                        B_TRUE);
3140                    if (attrzp) {
3141                            SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3142                                SA_ZPL_CTIME(zfsvfs), NULL,
3143                                &ctime, sizeof (ctime));
3144                            zfs_tstamp_update_setup(attrzp, STATE_CHANGED,
3145                                mtime, ctime, B_TRUE);
3146                    }
3147            }
3148            /*
3149             * Do this after setting timestamps to prevent timestamp
3150             * update from toggling bit
3151             */
3152
3153            if (xoap && (mask & AT_XVATTR)) {
3154
3155                    /*
3156                     * restore trimmed off masks
3157                     * so that return masks can be set for caller.
3158                     */
3159
3160                    if (XVA_ISSET_REQ(&tmpxvattr, XAT_APPENDONLY)) {
3161                            XVA_SET_REQ(xvap, XAT_APPENDONLY);
3162                    }
3163                    if (XVA_ISSET_REQ(&tmpxvattr, XAT_NOUNLINK)) {
3164                            XVA_SET_REQ(xvap, XAT_NOUNLINK);
3165                    }
3166                    if (XVA_ISSET_REQ(&tmpxvattr, XAT_IMMUTABLE)) {
3167                            XVA_SET_REQ(xvap, XAT_IMMUTABLE);
3168                    }
3169                    if (XVA_ISSET_REQ(&tmpxvattr, XAT_NODUMP)) {
3170                            XVA_SET_REQ(xvap, XAT_NODUMP);
3171                    }
3172                    if (XVA_ISSET_REQ(&tmpxvattr, XAT_AV_MODIFIED)) {
3173                            XVA_SET_REQ(xvap, XAT_AV_MODIFIED);
3174                    }
3175                    if (XVA_ISSET_REQ(&tmpxvattr, XAT_AV_QUARANTINED)) {
3176                            XVA_SET_REQ(xvap, XAT_AV_QUARANTINED);
3177                    }
3178
3179                    if (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP))
3180                            ASSERT(vp->v_type == VREG);
3181
3182                    zfs_xvattr_set(zp, xvap, tx);
3183            }
```



```
3185            if (fuid_dirtied)
3186                    zfs_fuid_sync(zfsvfs, tx);
3187
3188            if (mask != 0)
3189                    zfs_log_setattr(zilog, tx, TX_SETATTR, zp, vap, mask, fuidp);
3190
3191            mutex_exit(&zp->z_lock);
3192            if (mask & (AT_UID|AT_GID|AT_MODE))
3193                    mutex_exit(&zp->z_acl_lock);
3194
3195            if (attrzp) {
3196                    if (mask & (AT_UID|AT_GID|AT_MODE))
3197                            mutex_exit(&attrzp->z_acl_lock);
3198                    mutex_exit(&attrzp->z_lock);
3199            }
3200    out:
3201            if (err == 0 && attrzp) {
3202                    err2 = sa_bulk_update(attrzp->z_sa_hdl, xattr_bulk,
3203                        xattr_count, tx);
3204                    ASSERT(err2 == 0);
3205            }
3206
3207            if (attrzp)
3208                    VN_RELE(ZTOV(attrzp));
3209
3210    #endif /* ! codereview */
3211            if (aclp)
3212                    zfs_acl_free(aclp);
3213
3214            if (fuidp) {
3215                    zfs_fuid_info_free(fuidp);
3216                    fuidp = NULL;
3217            }
3218
3219            if (err) {
3220                    dmu_tx_abort(tx);
3221                    if (err == ERESTART)
3222                            goto top;
3223            } else {
3224                    err2 = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);
3225                    dmu_tx_commit(tx);
3226            }
3227
3228    out2:
3229            if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3230                    zil_commit(zilog, 0);
3231
3232            ZFS_EXIT(zfsvfs);
3233            return (err);
3234    }
3235
3236    typedef struct zfs_zlock {
3237            krwlock_t       *zl_rwlock;     /* lock we acquired */
3238            znode_t         *zl_znode;      /* znode we held */
3239            struct zfs_zlock *zl_next;      /* next in list */
3240    } zfs_zlock_t;
3241
3242    /*
3243     * Drop locks and release vnodes that were held by zfs_rename_lock().
3244     */
3245    static void
3246    zfs_rename_unlock(zfs_zlock_t **zlpp)
3247    {
3248            zfs_zlock_t *zl;
```

```
3250         while ((zl = *zlpp) != NULL) {
3251                 if (zl->zl_znode != NULL)
3252                         VN_RELE(ZTOV(zl->zl_znode));
3253                 rw_exit(zl->zl_rwlock);
3254                 *zlpp = zl->zl_next;
3255                 kmem_free(zl, sizeof (*zl));
3256         }
3257 }

3259 /*
3260  * Search back through the directory tree, using the ".." entries.
3261  * Lock each directory in the chain to prevent concurrent renames.
3262  * Fail any attempt to move a directory into one of its own descendants.
3263  * XXX - z_parent_lock can overlap with map or grow locks
3264  */
3265 static int
3266 zfs_rename_lock(znode_t *szp, znode_t *tdzp, znode_t *sdzp, zfs_zlock_t **zlpp)
3267 {
3268         zfs_zlock_t     *zl;
3269         znode_t         *zp = tdzp;
3270         uint64_t        rootid = zp->z_zfsvfs->z_root;
3271         uint64_t        oidp = zp->z_id;
3272         krwlock_t       *rwlp = &szp->z_parent_lock;
3273         krw_t           rw = RW_WRITER;

3275         /*
3276          * First pass write-locks szp and compares to zp->z_id.
3277          * Later passes read-lock zp and compare to zp->z_parent.
3278          */
3279         do {
3280                 if (!rw_tryenter(rwlp, rw)) {
3281                         /*
3282                          * Another thread is renaming in this path.
3283                          * Note that if we are a WRITER, we don't have any
3284                          * parent_locks held yet.
3285                          */
3286                         if (rw == RW_READER && zp->z_id > szp->z_id) {
3287                                 /*
3288                                  * Drop our locks and restart
3289                                  */
3290                                 zfs_rename_unlock(&zl);
3291                                 *zlpp = NULL;
3292                                 zp = tdzp;
3293                                 oidp = zp->z_id;
3294                                 rwlp = &szp->z_parent_lock;
3295                                 rw = RW_WRITER;
3296                                 continue;
3297                         } else {
3298                                 /*
3299                                  * Wait for other thread to drop its locks
3300                                  */
3301                                 rw_enter(rwlp, rw);
3302                         }
3303                 }

3305                 zl = kmem_alloc(sizeof (*zl), KM_SLEEP);
3306                 zl->zl_rwlock = rwlp;
3307                 zl->zl_znode = NULL;
3308                 zl->zl_next = *zlpp;
3309                 *zlpp = zl;

3311                 if (oidp == szp->z_id)          /* We're a descendant of szp */
3312                         return (SET_ERROR(EINVAL));

3314                 if (oidp == rootid)             /* We've hit the top */
3315                         return (0);
```



```
3317                 if (rw == RW_READER) {          /* i.e. not the first pass */
3318                         int error = zfs_zget(zp->z_zfsvfs, oidp, &zp);
3319                         if (error)
3320                                 return (error);
3321                         zl->zl_znode = zp;
3322                 }
3323                 (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_PARENT(zp->z_zfsvfs),
3324                     &oidp, sizeof (oidp));
3325                 rwlp = &zp->z_parent_lock;
3326                 rw = RW_READER;

3328         } while (zp->z_id != sdzp->z_id);

3330         return (0);
3331 }

3333 /*
3334  * Move an entry from the provided source directory to the target
3335  * directory.  Change the entry name as indicated.
3336  *
3337  *      IN:     sdvp    - Source directory containing the "old entry".
3338  *              snm     - Old entry name.
3339  *              tdvp    - Target directory to contain the "new entry".
3340  *              tnm     - New entry name.
3341  *              cr      - credentials of caller.
3342  *              ct      - caller context
3343  *              flags   - case flags
3344  *
3345  *      RETURN: 0 on success, error code on failure.
3211  *      RETURN: 0 if success
3212  *              error code if failure
3346  *
3347  * Timestamps:
3348  *      sdvp,tdvp - ctime|mtime updated
3349  */
3350 /*ARGSUSED*/
3351 static int
3352 zfs_rename(vnode_t *sdvp, char *snm, vnode_t *tdvp, char *tnm, cred_t *cr,
3353     caller_context_t *ct, int flags)
3354 {
3355         znode_t         *tdzp, *szp, *tzp;
3356         znode_t         *sdzp = VTOZ(sdvp);
3357         zfsvfs_t        *zfsvfs = sdzp->z_zfsvfs;
3358         zilog_t         *zilog;
3359         vnode_t         *realvp;
3360         zfs_dirlock_t   *sdl, *tdl;
3361         dmu_tx_t        *tx;
3362         zfs_zlock_t     *zl;
3363         int             cmp, serr, terr;
3364         int             error = 0;
3365         int             zflg = 0;

3367         ZFS_ENTER(zfsvfs);
3368         ZFS_VERIFY_ZP(sdzp);
3369         zilog = zfsvfs->z_log;

3371         /*
3372          * Make sure we have the real vp for the target directory.
3373          */
3374         if (VOP_REALVP(tdvp, &realvp, ct) == 0)
3375                 tdvp = realvp;

3377         if (tdvp->v_vfsp != sdvp->v_vfsp || zfsctl_is_node(tdvp)) {
3378                 ZFS_EXIT(zfsvfs);
3379                 return (SET_ERROR(EXDEV));
```

```
3380         }

3382         tdzp = VTOZ(tdvp);
3383         ZFS_VERIFY_ZP(tdzp);
3384         if (zfsvfs->z_utf8 && u8_validate(tnm,
3385             strlen(tnm), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3386                 ZFS_EXIT(zfsvfs);
3387                 return (SET_ERROR(EILSEQ));
3388         }

3390         if (flags & FIGNORECASE)
3391                 zflg |= ZCILOOK;

3393 top:
3394         szp = NULL;
3395         tzp = NULL;
3396         zl = NULL;

3398         /*
3399          * This is to prevent the creation of links into attribute space
3400          * by renaming a linked file into/outof an attribute directory.
3401          * See the comment in zfs_link() for why this is considered bad.
3402          */
3403         if ((tdzp->z_pflags & ZFS_XATTR) != (sdzp->z_pflags & ZFS_XATTR)) {
3404                 ZFS_EXIT(zfsvfs);
3405                 return (SET_ERROR(EINVAL));
3406         }

3408         /*
3409          * Lock source and target directory entries.  To prevent deadlock,
3410          * a lock ordering must be defined.  We lock the directory with
3411          * the smallest object id first, or if it's a tie, the one with
3412          * the lexically first name.
3413          */
3414         if (sdzp->z_id < tdzp->z_id) {
3415                 cmp = -1;
3416         } else if (sdzp->z_id > tdzp->z_id) {
3417                 cmp = 1;
3418         } else {
3419                 /*
3420                  * First compare the two name arguments without
3421                  * considering any case folding.
3422                  */
3423                 int nofold = (zfsvfs->z_norm & ~U8_TEXTPREP_TOUPPER);

3425                 cmp = u8_strcmp(snm, tnm, 0, nofold, U8_UNICODE_LATEST, &error);
3426                 ASSERT(error == 0 || !zfsvfs->z_utf8);
3427                 if (cmp == 0) {
3428                         /*
3429                          * POSIX: "If the old argument and the new argument
3430                          * both refer to links to the same existing file,
3431                          * the rename() function shall return successfully
3432                          * and perform no other action."
3433                          */
3434                         ZFS_EXIT(zfsvfs);
3435                         return (0);
3436                 }
3437                 /*
3438                  * If the file system is case-folding, then we may
3439                  * have some more checking to do.  A case-folding file
3440                  * system is either supporting mixed case sensitivity
3441                  * access or is completely case-insensitive.  Note
3442                  * that the file system is always case preserving.
3443                  *
3444                  * In mixed sensitivity mode case sensitive behavior
3445                  * is the default.  FIGNORECASE must be used to
```



```
3446                  * explicitly request case insensitive behavior.
3447                  *
3448                  * If the source and target names provided differ only
3449                  * by case (e.g., a request to rename 'tim' to 'Tim'),
3450                  * we will treat this as a special case in the
3451                  * case-insensitive mode: as long as the source name
3452                  * is an exact match, we will allow this to proceed as
3453                  * a name-change request.
3454                  */
3455                 if ((zfsvfs->z_case == ZFS_CASE_INSENSITIVE ||
3456                     (zfsvfs->z_case == ZFS_CASE_MIXED &&
3457                     flags & FIGNORECASE)) &&
3458                     u8_strcmp(snm, tnm, 0, zfsvfs->z_norm, U8_UNICODE_LATEST,
3459                     &error) == 0) {
3460                         /*
3461                          * case preserving rename request, require exact
3462                          * name matches
3463                          */
3464                         zflg |= ZCIEXACT;
3465                         zflg &= ~ZCILOOK;
3466                 }
3467         }

3469         /*
3470          * If the source and destination directories are the same, we should
3471          * grab the z_name_lock of that directory only once.
3472          */
3473         if (sdzp == tdzp) {
3474                 zflg |= ZHAVELOCK;
3475                 rw_enter(&sdzp->z_name_lock, RW_READER);
3476         }

3478         if (cmp < 0) {
3479                 serr = zfs_dirent_lock(&sdl, sdzp, snm, &szp,
3480                     ZEXISTS | zflg, NULL, NULL);
3481                 terr = zfs_dirent_lock(&tdl,
3482                     tdzp, tnm, &tzp, ZRENAMING | zflg, NULL, NULL);
3483         } else {
3484                 terr = zfs_dirent_lock(&tdl,
3485                     tdzp, tnm, &tzp, zflg, NULL, NULL);
3486                 serr = zfs_dirent_lock(&sdl,
3487                     sdzp, snm, &szp, ZEXISTS | ZRENAMING | zflg,
3488                     NULL, NULL);
3489         }

3491         if (serr) {
3492                 /*
3493                  * Source entry invalid or not there.
3494                  */
3495                 if (!terr) {
3496                         zfs_dirent_unlock(tdl);
3497                         if (tzp)
3498                                 VN_RELE(ZTOV(tzp));
3499                 }

3501                 if (sdzp == tdzp)
3502                         rw_exit(&sdzp->z_name_lock);

3504                 if (strcmp(snm, "..") == 0)
3505                         serr = SET_ERROR(EINVAL);
3506                 ZFS_EXIT(zfsvfs);
3507                 return (serr);
3508         }
3509         if (terr) {
3510                 zfs_dirent_unlock(sdl);
3511                 VN_RELE(ZTOV(szp));
```

```
3513                if (sdzp == tdzp)
3514                        rw_exit(&sdzp->z_name_lock);
3515
3516                if (strcmp(tnm, "..") == 0)
3517                        terr = SET_ERROR(EINVAL);
3518                ZFS_EXIT(zfsvfs);
3519                return (terr);
3520        }
3521
3522        /*
3523         * Must have write access at the source to remove the old entry
3524         * and write access at the target to create the new entry.
3525         * Note that if target and source are the same, this can be
3526         * done in a single check.
3527         */
3528
3529        if (error = zfs_zaccess_rename(sdzp, szp, tdzp, tzp, cr))
3530                goto out;
3531
3532        if (ZTOV(szp)->v_type == VDIR) {
3533                /*
3534                 * Check to make sure rename is valid.
3535                 * Can’t do a move like this: /usr/a/b to /usr/a/b/c/d
3536                 */
3537                if (error = zfs_rename_lock(szp, tdzp, sdzp, &zl))
3538                        goto out;
3539        }
3540
3541        /*
3542         * Does target exist?
3543         */
3544        if (tzp) {
3545                /*
3546                 * Source and target must be the same type.
3547                 */
3548                if (ZTOV(szp)->v_type == VDIR) {
3549                        if (ZTOV(tzp)->v_type != VDIR) {
3550                                error = SET_ERROR(ENOTDIR);
3551                                goto out;
3552                        }
3553                } else {
3554                        if (ZTOV(tzp)->v_type == VDIR) {
3555                                error = SET_ERROR(EISDIR);
3556                                goto out;
3557                        }
3558                }
3559                /*
3560                 * POSIX dictates that when the source and target
3561                 * entries refer to the same file object, rename
3562                 * must do nothing and exit without error.
3563                 */
3564                if (szp->z_id == tzp->z_id) {
3565                        error = 0;
3566                        goto out;
3567                }
3568        }
3569
3570        vnevent_rename_src(ZTOV(szp), sdvp, snm, ct);
3571        if (tzp)
3572                vnevent_rename_dest(ZTOV(tzp), tdvp, tnm, ct);
3573
3574        /*
3575         * notify the target directory if it is not the same
3576         * as source directory.
3577         */
```



```
3578        if (tdvp != sdvp) {
3579                vnevent_rename_dest_dir(tdvp, ct);
3580        }
3581
3582        tx = dmu_tx_create(zfsvfs->z_os);
3583        dmu_tx_hold_sa(tx, szp->z_sa_hdl, B_FALSE);
3584        dmu_tx_hold_sa(tx, sdzp->z_sa_hdl, B_FALSE);
3585        dmu_tx_hold_zap(tx, sdzp->z_id, FALSE, snm);
3586        dmu_tx_hold_zap(tx, tdzp->z_id, TRUE, tnm);
3587        if (sdzp != tdzp) {
3588                dmu_tx_hold_sa(tx, tdzp->z_sa_hdl, B_FALSE);
3589                zfs_sa_upgrade_txholds(tx, tdzp);
3590        }
3591        if (tzp) {
3592                dmu_tx_hold_sa(tx, tzp->z_sa_hdl, B_FALSE);
3593                zfs_sa_upgrade_txholds(tx, tzp);
3594        }
3595
3596        zfs_sa_upgrade_txholds(tx, szp);
3597        dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
3598        error = dmu_tx_assign(tx, TXG_NOWAIT);
3599        if (error) {
3600                if (zl != NULL)
3601                        zfs_rename_unlock(&zl);
3602                zfs_dirent_unlock(sdl);
3603                zfs_dirent_unlock(tdl);
3604
3605                if (sdzp == tdzp)
3606                        rw_exit(&sdzp->z_name_lock);
3607
3608                VN_RELE(ZTOV(szp));
3609                if (tzp)
3610                        VN_RELE(ZTOV(tzp));
3611                if (error == ERESTART) {
3612                        dmu_tx_wait(tx);
3613                        dmu_tx_abort(tx);
3614                        goto top;
3615                }
3616                dmu_tx_abort(tx);
3617                ZFS_EXIT(zfsvfs);
3618                return (error);
3619        }
3620
3621        if (tzp)        /* Attempt to remove the existing target */
3622                error = zfs_link_destroy(tdl, tzp, tx, zflg, NULL);
3623
3624        if (error == 0) {
3625                error = zfs_link_create(tdl, szp, tx, ZRENAMING);
3626                if (error == 0) {
3627                        szp->z_pflags |= ZFS_AV_MODIFIED;
3628
3629                        error = sa_update(szp->z_sa_hdl, SA_ZPL_FLAGS(zfsvfs),
3630                            (void *)&szp->z_pflags, sizeof (uint64_t), tx);
3631                        ASSERT0(error);
3632
3633                        error = zfs_link_destroy(sdl, szp, tx, ZRENAMING, NULL);
3634                        if (error == 0) {
3635                                zfs_log_rename(zilog, tx, TX_RENAME |
3636                                    (flags & FIGNORECASE ? TX_CI : 0), sdzp,
3637                                    sdl->dl_name, tdzp, tdl->dl_name, szp);
3638
3639                                /*
3640                                 * Update path information for the target vnode
3641                                 */
3642                                vn_renamepath(tdvp, ZTOV(szp), tnm,
3643                                    strlen(tnm));
```

```
3644                        } else {
3645                                /*
3646                                 * At this point, we have successfully created
3647                                 * the target name, but have failed to remove
3648                                 * the source name.  Since the create was done
3649                                 * with the ZRENAMING flag, there are
3650                                 * complications; for one, the link count is
3651                                 * wrong.  The easiest way to deal with this
3652                                 * is to remove the newly created target, and
3653                                 * return the original error.  This must
3654                                 * succeed; fortunately, it is very unlikely to
3655                                 * fail, since we just created it.
3656                                 */
3657                                VERIFY3U(zfs_link_destroy(tdl, szp, tx,
3658                                    ZRENAMING, NULL), ==, 0);
3659                        }
3660                }
3661        }

3663        dmu_tx_commit(tx);
3664 out:
3665        if (zl != NULL)
3666                zfs_rename_unlock(&zl);

3668        zfs_dirent_unlock(sdl);
3669        zfs_dirent_unlock(tdl);

3671        if (sdzp == tdzp)
3672                rw_exit(&sdzp->z_name_lock);


3675        VN_RELE(ZTOV(szp));
3676        if (tzp)
3677                VN_RELE(ZTOV(tzp));

3679        if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3680                zil_commit(zilog, 0);

3682        ZFS_EXIT(zfsvfs);
3683        return (error);
3684 }

3686 /*
3687  * Insert the indicated symbolic reference entry into the directory.
3688  *
3689  *      IN:     dvp     - Directory to contain new symbolic link.
3690  *              link    - Name for new symlink entry.
3691  *              vap     - Attributes of new entry.
3559  *              target  - Target path of new symlink.
3692  *              cr      - credentials of caller.
3693  *              ct      - caller context
3694  *              flags   - case flags
3695  *
3696  *      RETURN: 0 on success, error code on failure.
3564  *      RETURN: 0 if success
3565  *              error code if failure
3697  *
3698  * Timestamps:
3699  *      dvp - ctime|mtime updated
3700  */
3701 /*ARGSUSED*/
3702 static int
3703 zfs_symlink(vnode_t *dvp, char *name, vattr_t *vap, char *link, cred_t *cr,
3704     caller_context_t *ct, int flags)
3705 {
3706        znode_t         *zp, *dzp = VTOZ(dvp);
```



```
3707        zfs_dirlock_t   *dl;
3708        dmu_tx_t        *tx;
3709        zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
3710        zilog_t         *zilog;
3711        uint64_t        len = strlen(link);
3712        int             error;
3713        int             zflg = ZNEW;
3714        zfs_acl_ids_t   acl_ids;
3715        boolean_t       fuid_dirtied;
3716        uint64_t        txtype = TX_SYMLINK;

3718        ASSERT(vap->va_type == VLNK);

3720        ZFS_ENTER(zfsvfs);
3721        ZFS_VERIFY_ZP(dzp);
3722        zilog = zfsvfs->z_log;

3724        if (zfsvfs->z_utf8 && u8_validate(name, strlen(name),
3725            NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3726                ZFS_EXIT(zfsvfs);
3727                return (SET_ERROR(EILSEQ));
3728        }
3729        if (flags & FIGNORECASE)
3730                zflg |= ZCILOOK;

3732        if (len > MAXPATHLEN) {
3733                ZFS_EXIT(zfsvfs);
3734                return (SET_ERROR(ENAMETOOLONG));
3735        }

3737        if ((error = zfs_acl_ids_create(dzp, 0,
3738            vap, cr, NULL, &acl_ids)) != 0) {
3739                ZFS_EXIT(zfsvfs);
3740                return (error);
3741        }
3742 top:
3743        /*
3744         * Attempt to lock directory; fail if entry already exists.
3745         */
3746        error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg, NULL, NULL);
3747        if (error) {
3748                zfs_acl_ids_free(&acl_ids);
3749                ZFS_EXIT(zfsvfs);
3750                return (error);
3751        }

3753        if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
3754                zfs_acl_ids_free(&acl_ids);
3755                zfs_dirent_unlock(dl);
3756                ZFS_EXIT(zfsvfs);
3757                return (error);
3758        }

3760        if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
3761                zfs_acl_ids_free(&acl_ids);
3762                zfs_dirent_unlock(dl);
3763                ZFS_EXIT(zfsvfs);
3764                return (SET_ERROR(EDQUOT));
3765        }
3766        tx = dmu_tx_create(zfsvfs->z_os);
3767        fuid_dirtied = zfsvfs->z_fuid_dirty;
3768        dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0, MAX(1, len));
3769        dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
3770        dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
3771            ZFS_SA_BASE_ATTR_SIZE + len);
3772        dmu_tx_hold_sa(tx, dzp->z_sa_hdl, B_FALSE);
```

```
3773         if (!zfsvfs->z_use_sa && acl_ids.z_aclp->z_acl_bytes > ZFS_ACE_SPACE) {
3774                 dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0,
3775                     acl_ids.z_aclp->z_acl_bytes);
3776         }
3777         if (fuid_dirtied)
3778                 zfs_fuid_txhold(zfsvfs, tx);
3779         error = dmu_tx_assign(tx, TXG_NOWAIT);
3780         if (error) {
3781                 zfs_dirent_unlock(dl);
3782                 if (error == ERESTART) {
3783                         dmu_tx_wait(tx);
3784                         dmu_tx_abort(tx);
3785                         goto top;
3786                 }
3787                 zfs_acl_ids_free(&acl_ids);
3788                 dmu_tx_abort(tx);
3789                 ZFS_EXIT(zfsvfs);
3790                 return (error);
3791         }

3793         /*
3794          * Create a new object for the symlink.
3795          * for version 4 ZPL datsets the symlink will be an SA attribute
3796          */
3797         zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids);

3799         if (fuid_dirtied)
3800                 zfs_fuid_sync(zfsvfs, tx);

3802         mutex_enter(&zp->z_lock);
3803         if (zp->z_is_sa)
3804                 error = sa_update(zp->z_sa_hdl, SA_ZPL_SYMLINK(zfsvfs),
3805                     link, len, tx);
3806         else
3807                 zfs_sa_symlink(zp, link, len, tx);
3808         mutex_exit(&zp->z_lock);

3810         zp->z_size = len;
3811         (void) sa_update(zp->z_sa_hdl, SA_ZPL_SIZE(zfsvfs),
3812             &zp->z_size, sizeof (zp->z_size), tx);
3813         /*
3814          * Insert the new object into the directory.
3815          */
3816         (void) zfs_link_create(dl, zp, tx, ZNEW);

3818         if (flags & FIGNORECASE)
3819                 txtype |= TX_CI;
3820         zfs_log_symlink(zilog, tx, txtype, dzp, zp, name, link);

3822         zfs_acl_ids_free(&acl_ids);

3824         dmu_tx_commit(tx);

3826         zfs_dirent_unlock(dl);

3828         VN_RELE(ZTOV(zp));

3830         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3831                 zil_commit(zilog, 0);

3833         ZFS_EXIT(zfsvfs);
3834         return (error);
3835 }

3837 /*
3838  * Return, in the buffer contained in the provided uio structure,
```



```
3839  * the symbolic path referred to by vp.
3840  *
3841  *      IN:     vp      - vnode of symbolic link.
3842  *              uio     - structure to contain the link path.
3711  *              uoip    - structure to contain the link path.
3843  *              cr      - credentials of caller.
3844  *              ct      - caller context
3845  *
3846  *      OUT:    uio     - structure containing the link path.
3715  *      OUT:    uio     - structure to contain the link path.
3847  *
3848  *      RETURN: 0 on success, error code on failure.
3717  *      RETURN: 0 if success
3718  *              error code if failure
3849  *
3850  * Timestamps:
3851  *      vp - atime updated
3852  */
3853 /* ARGSUSED */
3854 static int
3855 zfs_readlink(vnode_t *vp, uio_t *uio, cred_t *cr, caller_context_t *ct)
3856 {
3857         znode_t         *zp = VTOZ(vp);
3858         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
3859         int             error;

3861         ZFS_ENTER(zfsvfs);
3862         ZFS_VERIFY_ZP(zp);

3864         mutex_enter(&zp->z_lock);
3865         if (zp->z_is_sa)
3866                 error = sa_lookup_uio(zp->z_sa_hdl,
3867                     SA_ZPL_SYMLINK(zfsvfs), uio);
3868         else
3869                 error = zfs_sa_readlink(zp, uio);
3870         mutex_exit(&zp->z_lock);

3872         ZFS_ACCESSTIME_STAMP(zfsvfs, zp);

3874         ZFS_EXIT(zfsvfs);
3875         return (error);
3876 }

3878 /*
3879  * Insert a new entry into directory tdvp referencing svp.
3880  *
3881  *      IN:     tdvp    - Directory to contain new entry.
3882  *              svp     - vnode of new entry.
3883  *              name    - name of new entry.
3884  *              cr      - credentials of caller.
3885  *              ct      - caller context
3886  *
3887  *      RETURN: 0 on success, error code on failure.
3757  *      RETURN: 0 if success
3758  *              error code if failure
3888  *
3889  * Timestamps:
3890  *      tdvp - ctime|mtime updated
3891  *       svp - ctime updated
3892  */
3893 /* ARGSUSED */
3894 static int
3895 zfs_link(vnode_t *tdvp, vnode_t *svp, char *name, cred_t *cr,
3896     caller_context_t *ct, int flags)
3897 {
3898         znode_t         *dzp = VTOZ(tdvp);
```

```
3899         znode_t         *tzp, *szp;
3900         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
3901         zilog_t         *zilog;
3902         zfs_dirlock_t   *dl;
3903         dmu_tx_t        *tx;
3904         vnode_t         *realvp;
3905         int             error;
3906         int             zf = ZNEW;
3907         uint64_t        parent;
3908         uid_t           owner;

3910         ASSERT(tdvp->v_type == VDIR);

3912         ZFS_ENTER(zfsvfs);
3913         ZFS_VERIFY_ZP(dzp);
3914         zilog = zfsvfs->z_log;

3916         if (VOP_REALVP(svp, &realvp, ct) == 0)
3917                 svp = realvp;

3919         /*
3920          * POSIX dictates that we return EPERM here.
3921          * Better choices include ENOTSUP or EISDIR.
3922          */
3923         if (svp->v_type == VDIR) {
3924                 ZFS_EXIT(zfsvfs);
3925                 return (SET_ERROR(EPERM));
3926         }

3928         if (svp->v_vfsp != tdvp->v_vfsp || zfsctl_is_node(svp)) {
3929                 ZFS_EXIT(zfsvfs);
3930                 return (SET_ERROR(EXDEV));
3931         }

3933         szp = VTOZ(svp);
3934         ZFS_VERIFY_ZP(szp);

3936         /* Prevent links to .zfs/shares files */

3938         if ((error = sa_lookup(szp->z_sa_hdl, SA_ZPL_PARENT(zfsvfs),
3939             &parent, sizeof (uint64_t))) != 0) {
3940                 ZFS_EXIT(zfsvfs);
3941                 return (error);
3942         }
3943         if (parent == zfsvfs->z_shares_dir) {
3944                 ZFS_EXIT(zfsvfs);
3945                 return (SET_ERROR(EPERM));
3946         }

3948         if (zfsvfs->z_utf8 && u8_validate(name,
3949             strlen(name), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3950                 ZFS_EXIT(zfsvfs);
3951                 return (SET_ERROR(EILSEQ));
3952         }
3953         if (flags & FIGNORECASE)
3954                 zf |= ZCILOOK;

3956         /*
3957          * We do not support links between attributes and non-attributes
3958          * because of the potential security risk of creating links
3959          * into "normal" file space in order to circumvent restrictions
3960          * imposed in attribute space.
3961          */
3962         if ((szp->z_pflags & ZFS_XATTR) != (dzp->z_pflags & ZFS_XATTR)) {
3963                 ZFS_EXIT(zfsvfs);
3964                 return (SET_ERROR(EINVAL));
```



```
3965         }


3968         owner = zfs_fuid_map_id(zfsvfs, szp->z_uid, cr, ZFS_OWNER);
3969         if (owner != crgetuid(cr) && secpolicy_basic_link(cr) != 0) {
3970                 ZFS_EXIT(zfsvfs);
3971                 return (SET_ERROR(EPERM));
3972         }

3974         if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
3975                 ZFS_EXIT(zfsvfs);
3976                 return (error);
3977         }

3979 top:
3980         /*
3981          * Attempt to lock directory; fail if entry already exists.
3982          */
3983         error = zfs_dirent_lock(&dl, dzp, name, &tzp, zf, NULL, NULL);
3984         if (error) {
3985                 ZFS_EXIT(zfsvfs);
3986                 return (error);
3987         }

3989         tx = dmu_tx_create(zfsvfs->z_os);
3990         dmu_tx_hold_sa(tx, szp->z_sa_hdl, B_FALSE);
3991         dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
3992         zfs_sa_upgrade_txholds(tx, szp);
3993         zfs_sa_upgrade_txholds(tx, dzp);
3994         error = dmu_tx_assign(tx, TXG_NOWAIT);
3995         if (error) {
3996                 zfs_dirent_unlock(dl);
3997                 if (error == ERESTART) {
3998                         dmu_tx_wait(tx);
3999                         dmu_tx_abort(tx);
4000                         goto top;
4001                 }
4002                 dmu_tx_abort(tx);
4003                 ZFS_EXIT(zfsvfs);
4004                 return (error);
4005         }

4007         error = zfs_link_create(dl, szp, tx, 0);

4009         if (error == 0) {
4010                 uint64_t txtype = TX_LINK;
4011                 if (flags & FIGNORECASE)
4012                         txtype |= TX_CI;
4013                 zfs_log_link(zilog, tx, txtype, dzp, szp, name);
4014         }

4016         dmu_tx_commit(tx);

4018         zfs_dirent_unlock(dl);

4020         if (error == 0) {
4021                 vnevent_link(svp, ct);
4022         }

4024         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
4025                 zil_commit(zilog, 0);

4027         ZFS_EXIT(zfsvfs);
4028         return (error);
4029 }
```

_____***unchanged_portion_omitted_***

```
4044 /*
4045  * Push a page out to disk, klustering if possible.
4046  *
4047  *      IN:     vp      - file to push page to.
4048  *              pp      - page to push.
4049  *              flags   - additional flags.
4050  *              cr      - credentials of caller.
4051  *
4052  *      OUT:    offp    - start of range pushed.
4053  *              lenp    - len of range pushed.
4054  *
4055  *      RETURN: 0 on success, error code on failure.
3926  *      RETURN: 0 if success
3927  *              error code if failure
4056  *
4057  * NOTE: callers must have locked the page to be pushed.  On
4058  * exit, the page (and all other pages in the kluster) must be
4059  * unlocked.
4060  */
4061 /* ARGSUSED */
4062 static int
4063 zfs_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp,
4064                 size_t *lenp, int flags, cred_t *cr)
4065 {
4066         znode_t         *zp = VTOZ(vp);
4067         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4068         dmu_tx_t        *tx;
4069         u_offset_t      off, koff;
4070         size_t          len, klen;
4071         int             err;

4073         off = pp->p_offset;
4074         len = PAGESIZE;
4075         /*
4076          * If our blocksize is bigger than the page size, try to kluster
4077          * multiple pages so that we write a full block (thus avoiding
4078          * a read-modify-write).
4079          */
4080         if (off < zp->z_size && zp->z_blksz > PAGESIZE) {
4081                 klen = P2ROUNDUP((ulong_t)zp->z_blksz, PAGESIZE);
4082                 koff = ISP2(klen) ? P2ALIGN(off, (u_offset_t)klen) : 0;
4083                 ASSERT(koff <= zp->z_size);
4084                 if (koff + klen > zp->z_size)
4085                         klen = P2ROUNDUP(zp->z_size - koff, (uint64_t)PAGESIZE);
4086                 pp = pvn_write_kluster(vp, pp, &off, &len, koff, klen, flags);
4087         }
4088         ASSERT3U(btop(len), ==, btopr(len));

4090         /*
4091          * Can't push pages past end-of-file.
4092          */
4093         if (off >= zp->z_size) {
4094                 /* ignore all pages */
4095                 err = 0;
4096                 goto out;
4097         } else if (off + len > zp->z_size) {
4098                 int npages = btopr(zp->z_size - off);
4099                 page_t *trunc;

4101                 page_list_break(&pp, &trunc, npages);
4102                 /* ignore pages past end of file */
4103                 if (trunc)
4104                         pvn_write_done(trunc, flags);
4105                 len = zp->z_size - off;
4106         }
```



```
4108         if (zfs_owner_overquota(zfsvfs, zp, B_FALSE) ||
4109             zfs_owner_overquota(zfsvfs, zp, B_TRUE)) {
4110                 err = SET_ERROR(EDQUOT);
4111                 goto out;
4112         }
4113 top:
4114         tx = dmu_tx_create(zfsvfs->z_os);
4115         dmu_tx_hold_write(tx, zp->z_id, off, len);

4117         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
4118         zfs_sa_upgrade_txholds(tx, zp);
4119         err = dmu_tx_assign(tx, TXG_NOWAIT);
4120         if (err != 0) {
4121                 if (err == ERESTART) {
4122                         dmu_tx_wait(tx);
4123                         dmu_tx_abort(tx);
4124                         goto top;
4125                 }
4126                 dmu_tx_abort(tx);
4127                 goto out;
4128         }

4130         if (zp->z_blksz <= PAGESIZE) {
4131                 caddr_t va = zfs_map_page(pp, S_READ);
4132                 ASSERT3U(len, <=, PAGESIZE);
4133                 dmu_write(zfsvfs->z_os, zp->z_id, off, len, va, tx);
4134                 zfs_unmap_page(pp, va);
4135         } else {
4136                 err = dmu_write_pages(zfsvfs->z_os, zp->z_id, off, len, pp, tx);
4137         }

4139         if (err == 0) {
4140                 uint64_t mtime[2], ctime[2];
4141                 sa_bulk_attr_t bulk[3];
4142                 int count = 0;

4144                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL,
4145                     &mtime, 16);
4146                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
4147                     &ctime, 16);
4148                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
4149                     &zp->z_pflags, 8);
4150                 zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
4151                     B_TRUE);
4152                 zfs_log_write(zfsvfs->z_log, tx, TX_WRITE, zp, off, len, 0);
4153         }
4154         dmu_tx_commit(tx);

4156 out:
4157         pvn_write_done(pp, (err ? B_ERROR : 0) | flags);
4158         if (offp)
4159                 *offp = off;
4160         if (lenp)
4161                 *lenp = len;

4163         return (err);
4164 }

4166 /*
4167  * Copy the portion of the file indicated from pages into the file.
4168  * The pages are stored in a page list attached to the files vnode.
4169  *
4170  *      IN:     vp      - vnode of file to push page data to.
4171  *              off     - position in file to put data.
4172  *              len     - amount of data to write.
```

```
 *              flags   - flags to control the operation.
 *              cr      - credentials of caller.
 *              ct      - caller context.
 *
 *      RETURN: 0 on success, error code on failure.
 *      RETURN: 0 if success
 *              error code if failure
 *
 * Timestamps:
 *      vp - ctime|mtime updated
 */
/*ARGSUSED*/
static int
zfs_putpage(vnode_t *vp, offset_t off, size_t len, int flags, cred_t *cr,
    caller_context_t *ct)
{
        znode_t         *zp = VTOZ(vp);
        zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
        page_t          *pp;
        size_t          io_len;
        u_offset_t      io_off;
        uint_t          blksz;
        rl_t            *rl;
        int             error = 0;

        ZFS_ENTER(zfsvfs);
        ZFS_VERIFY_ZP(zp);

        /*
         * There's nothing to do if no data is cached.
         */
        if (!vn_has_cached_data(vp)) {
                ZFS_EXIT(zfsvfs);
                return (0);
        }

        /*
         * Align this request to the file block size in case we kluster.
         * XXX - this can result in pretty aggresive locking, which can
         * impact simultanious read/write access.  One option might be
         * to break up long requests (len == 0) into block-by-block
         * operations to get narrower locking.
         */
        blksz = zp->z_blksz;
        if (ISP2(blksz))
                io_off = P2ALIGN_TYPED(off, blksz, u_offset_t);
        else
                io_off = 0;
        if (len > 0 && ISP2(blksz))
                io_len = P2ROUNDUP_TYPED(len + (off - io_off), blksz, size_t);
        else
                io_len = 0;

        if (io_len == 0) {
                /*
                 * Search the entire vp list for pages >= io_off.
                 */
                rl = zfs_range_lock(zp, io_off, UINT64_MAX, RL_WRITER);
                error = pvn_vplist_dirty(vp, io_off, zfs_putapage, flags, cr);
                goto out;
        }
        rl = zfs_range_lock(zp, io_off, io_len, RL_WRITER);

        if (off > zp->z_size) {
                /* past end of file */
                zfs_range_unlock(rl);
```



```
                ZFS_EXIT(zfsvfs);
                return (0);
        }

        len = MIN(io_len, P2ROUNDUP(zp->z_size, PAGESIZE) - io_off);

        for (off = io_off; io_off < off + len; io_off += io_len) {
                if ((flags & B_INVAL) || ((flags & B_ASYNC) == 0)) {
                        pp = page_lookup(vp, io_off,
                            (flags & (B_INVAL | B_FREE)) ? SE_EXCL : SE_SHARED);
                } else {
                        pp = page_lookup_nowait(vp, io_off,
                            (flags & B_FREE) ? SE_EXCL : SE_SHARED);
                }

                if (pp != NULL && pvn_getdirty(pp, flags)) {
                        int err;

                        /*
                         * Found a dirty page to push
                         */
                        err = zfs_putapage(vp, pp, &io_off, &io_len, flags, cr);
                        if (err)
                                error = err;
                } else {
                        io_len = PAGESIZE;
                }
        }
out:
        zfs_range_unlock(rl);
        if ((flags & B_ASYNC) == 0 || zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
                zil_commit(zfsvfs->z_log, zp->z_id);
        ZFS_EXIT(zfsvfs);
        return (error);
}
```

*unchanged_portion_omitted_*

```
/*
 * Bounds-check the seek operation.
 *
 *      IN:     vp      - vnode seeking within
 *              ooff    - old file offset
 *              noffp   - pointer to new file offset
 *              ct      - caller context
 *
 *      RETURN: 0 on success, EINVAL if new offset invalid.
 *      RETURN: 0 if success
 *              EINVAL if new offset invalid
 */
/* ARGSUSED */
static int
zfs_seek(vnode_t *vp, offset_t ooff, offset_t *noffp,
    caller_context_t *ct)
{
        if (vp->v_type == VDIR)
                return (0);
        return ((*noffp < 0 || *noffp > MAXOFFSET_T) ? EINVAL : 0);
}
```

*unchanged_portion_omitted_*

```
/*
 * Return pointers to the pages for the file region [off, off + len]
 * in the pl array.  If plsz is greater than len, this function may
 * also return page pointers from after the specified region
 * (i.e. the region [off, off + plsz]).  These additional pages are
 * only returned if they are already in the cache, or were created as
```

```
4463  * part of a klustered read.
4464  *
4465  *      IN:     vp      - vnode of file to get data from.
4466  *              off     - position in file to get data from.
4467  *              len     - amount of data to retrieve.
4468  *              plsz    - length of provided page list.
4469  *              seg     - segment to obtain pages for.
4470  *              addr    - virtual address of fault.
4471  *              rw      - mode of created pages.
4472  *              cr      - credentials of caller.
4473  *              ct      - caller context.
4474  *
4475  *      OUT:    protp   - protection mode of created pages.
4476  *              pl      - list of pages created.
4477  *
4478  *      RETURN: 0 on success, error code on failure.
4352  *      RETURN: 0 if success
4353  *              error code if failure
4479  *
4480  * Timestamps:
4481  *      vp - atime updated
4482  */
4483 /* ARGSUSED */
4484 static int
4485 zfs_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
4486     page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
4487     enum seg_rw rw, cred_t *cr, caller_context_t *ct)
4488 {
4489         znode_t         *zp = VTOZ(vp);
4490         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4491         page_t          **pl0 = pl;
4492         int             err = 0;

4494         /* we do our own caching, faultahead is unnecessary */
4495         if (pl == NULL)
4496                 return (0);
4497         else if (len > plsz)
4498                 len = plsz;
4499         else
4500                 len = P2ROUNDUP(len, PAGESIZE);
4501         ASSERT(plsz >= len);

4503         ZFS_ENTER(zfsvfs);
4504         ZFS_VERIFY_ZP(zp);

4506         if (protp)
4507                 *protp = PROT_ALL;

4509         /*
4510          * Loop through the requested range [off, off + len) looking
4511          * for pages.  If we don't find a page, we will need to create
4512          * a new page and fill it with data from the file.
4513          */
4514         while (len > 0) {
4515                 if (*pl = page_lookup(vp, off, SE_SHARED))
4516                         *(pl+1) = NULL;
4517                 else if (err = zfs_fillpage(vp, off, seg, addr, pl, plsz, rw))
4518                         goto out;
4519                 while (*pl) {
4520                         ASSERT3U((*pl)->p_offset, ==, off);
4521                         off += PAGESIZE;
4522                         addr += PAGESIZE;
4523                         if (len > 0) {
4524                                 ASSERT3U(len, >=, PAGESIZE);
4525                                 len -= PAGESIZE;
4526                         }
```



```
4527                         ASSERT3U(plsz, >=, PAGESIZE);
4528                         plsz -= PAGESIZE;
4529                         pl++;
4530                 }
4531         }

4533         /*
4534          * Fill out the page array with any pages already in the cache.
4535          */
4536         while (plsz > 0 &&
4537             (*pl++ = page_lookup_nowait(vp, off, SE_SHARED))) {
4538                         off += PAGESIZE;
4539                         plsz -= PAGESIZE;
4540         }
4541 out:
4542         if (err) {
4543                 /*
4544                  * Release any pages we have previously locked.
4545                  */
4546                 while (pl > pl0)
4547                         page_unlock(*--pl);
4548         } else {
4549                 ZFS_ACCESSTIME_STAMP(zfsvfs, zp);
4550         }

4552         *pl = NULL;

4554         ZFS_EXIT(zfsvfs);
4555         return (err);
4556 }

4558 /*
4559  * Request a memory map for a section of a file.  This code interacts
4560  * with common code and the VM system as follows:
4561  *
4562  * - common code calls mmap(), which ends up in smmap_common()
4563  * - this calls VOP_MAP(), which takes you into (say) zfs
4564  * - zfs_map() calls as_map(), passing segvn_create() as the callback
4565  * - segvn_create() creates the new segment and calls VOP_ADDMAP()
4566  * - zfs_addmap() updates z_mapcnt
4437  *      common code calls mmap(), which ends up in smmap_common()
4438  *
4439  *      this calls VOP_MAP(), which takes you into (say) zfs
4440  *
4441  *      zfs_map() calls as_map(), passing segvn_create() as the callback
4442  *
4443  *      segvn_create() creates the new segment and calls VOP_ADDMAP()
4444  *
4445  *      zfs_addmap() updates z_mapcnt
4567  */
4568 /*ARGSUSED*/
4569 static int
4570 zfs_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
4571     size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
4572     caller_context_t *ct)
4573 {
4574         znode_t *zp = VTOZ(vp);
4575         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
4576         segvn_crargs_t  vn_a;
4577         int             error;

4579         ZFS_ENTER(zfsvfs);
4580         ZFS_VERIFY_ZP(zp);

4582         if ((prot & PROT_WRITE) && (zp->z_pflags &
4583             (ZFS_IMMUTABLE | ZFS_READONLY | ZFS_APPENDONLY))) {
```

```
4584                 ZFS_EXIT(zfsvfs);
4585                 return (SET_ERROR(EPERM));
4586         }

4588         if ((prot & (PROT_READ | PROT_EXEC)) &&
4589             (zp->z_pflags & ZFS_AV_QUARANTINED)) {
4590                 ZFS_EXIT(zfsvfs);
4591                 return (SET_ERROR(EACCES));
4592         }

4594         if (vp->v_flag & VNOMAP) {
4595                 ZFS_EXIT(zfsvfs);
4596                 return (SET_ERROR(ENOSYS));
4597         }

4599         if (off < 0 || len > MAXOFFSET_T - off) {
4600                 ZFS_EXIT(zfsvfs);
4601                 return (SET_ERROR(ENXIO));
4602         }

4604         if (vp->v_type != VREG) {
4605                 ZFS_EXIT(zfsvfs);
4606                 return (SET_ERROR(ENODEV));
4607         }

4609         /*
4610          * If file is locked, disallow mapping.
4611          */
4612         if (MANDMODE(zp->z_mode) && vn_has_flocks(vp)) {
4613                 ZFS_EXIT(zfsvfs);
4614                 return (SET_ERROR(EAGAIN));
4615         }

4617         as_rangelock(as);
4618         error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);
4619         if (error != 0) {
4620                 as_rangeunlock(as);
4621                 ZFS_EXIT(zfsvfs);
4622                 return (error);
4623         }

4625         vn_a.vp = vp;
4626         vn_a.offset = (u_offset_t)off;
4627         vn_a.type = flags & MAP_TYPE;
4628         vn_a.prot = prot;
4629         vn_a.maxprot = maxprot;
4630         vn_a.cred = cr;
4631         vn_a.amp = NULL;
4632         vn_a.flags = flags & ~MAP_TYPE;
4633         vn_a.szc = 0;
4634         vn_a.lgrp_mem_policy_flags = 0;

4636         error = as_map(as, *addrp, len, segvn_create, &vn_a);

4638         as_rangeunlock(as);
4639         ZFS_EXIT(zfsvfs);
4640         return (error);
4641 }
```

_____***unchanged_portion_omitted_***

```
4694 /*
4695  * Free or allocate space in a file.  Currently, this function only
4696  * supports the `F_FREESP' command.  However, this command is somewhat
4697  * misnamed, as its functionality includes the ability to allocate as
4698  * well as free space.
4699  *
```



```
4700  *      IN:     vp      - vnode of file to free data in.
4701  *              cmd     - action to take (only F_FREESP supported).
4702  *              bfp     - section of file to free/alloc.
4703  *              flag    - current file open mode flags.
4704  *              offset  - current file offset.
4705  *              cr      - credentials of caller [UNUSED].
4706  *              ct      - caller context.
4707  *
4708  *      RETURN: 0 on success, error code on failure.
4587  *      RETURN: 0 if success
4588  *              error code if failure
4709  *
4710  * Timestamps:
4711  *      vp - ctime|mtime updated
4712  */
4713 /* ARGSUSED */
4714 static int
4715 zfs_space(vnode_t *vp, int cmd, flock64_t *bfp, int flag,
4716     offset_t offset, cred_t *cr, caller_context_t *ct)
4717 {
4718         znode_t         *zp = VTOZ(vp);
4719         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4720         uint64_t        off, len;
4721         int             error;

4723         ZFS_ENTER(zfsvfs);
4724         ZFS_VERIFY_ZP(zp);

4726         if (cmd != F_FREESP) {
4727                 ZFS_EXIT(zfsvfs);
4728                 return (SET_ERROR(EINVAL));
4729         }

4731         if (error = convoff(vp, bfp, 0, offset)) {
4732                 ZFS_EXIT(zfsvfs);
4733                 return (error);
4734         }

4736         if (bfp->l_len < 0) {
4737                 ZFS_EXIT(zfsvfs);
4738                 return (SET_ERROR(EINVAL));
4739         }

4741         off = bfp->l_start;
4742         len = bfp->l_len; /* 0 means from off to end of file */

4744         error = zfs_freesp(zp, off, len, flag, TRUE);

4746         ZFS_EXIT(zfsvfs);
4747         return (error);
4748 }
```

_____***unchanged_portion_omitted_***

```
4922 /*
4923  * The smallest read we may consider to loan out an arcbuf.
4924  * This must be a power of 2.
4803  * Tunable, both must be a power of 2.
4804  *
4805  * zcr_blksz_min: the smallest read we may consider to loan out an arcbuf
4806  * zcr_blksz_max: if set to less than the file block size, allow loaning out of
4807  *                an arcbuf for a partial block read
4925  */
4926 int zcr_blksz_min = (1 << 10);  /* 1K */
4927 /*
4928  * If set to less than the file block size, allow loaning out of an
4929  * arcbuf for a partial block read.  This must be a power of 2.
```

```
4930  */
4931 #endif /* ! codereview */
4932 int zcr_blksz_max = (1 << 17);  /* 128K */
4933
4934 /*ARGSUSED*/
4935 static int
4936 zfs_reqzcbuf(vnode_t *vp, enum uio_rw ioflag, xuio_t *xuio, cred_t *cr,
4937     caller_context_t *ct)
4938 {
4939         znode_t *zp = VTOZ(vp);
4940         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
4941         int max_blksz = zfsvfs->z_max_blksz;
4942         uio_t *uio = &xuio->xu_uio;
4943         ssize_t size = uio->uio_resid;
4944         offset_t offset = uio->uio_loffset;
4945         int blksz;
4946         int fullblk, i;
4947         arc_buf_t *abuf;
4948         ssize_t maxsize;
4949         int preamble, postamble;
4950
4951         if (xuio->xu_type != UIOTYPE_ZEROCOPY)
4952                 return (SET_ERROR(EINVAL));
4953
4954         ZFS_ENTER(zfsvfs);
4955         ZFS_VERIFY_ZP(zp);
4956         switch (ioflag) {
4957         case UIO_WRITE:
4958                 /*
4959                  * Loan out an arc_buf for write if write size is bigger than
4960                  * max_blksz, and the file's block size is also max_blksz.
4961                  */
4962                 blksz = max_blksz;
4963                 if (size < blksz || zp->z_blksz != blksz) {
4964                         ZFS_EXIT(zfsvfs);
4965                         return (SET_ERROR(EINVAL));
4966                 }
4967                 /*
4968                  * Caller requests buffers for write before knowing where the
4969                  * write offset might be (e.g. NFS TCP write).
4970                  */
4971                 if (offset == -1) {
4972                         preamble = 0;
4973                 } else {
4974                         preamble = P2PHASE(offset, blksz);
4975                         if (preamble) {
4976                                 preamble = blksz - preamble;
4977                                 size -= preamble;
4978                         }
4979                 }
4980
4981                 postamble = P2PHASE(size, blksz);
4982                 size -= postamble;
4983
4984                 fullblk = size / blksz;
4985                 (void) dmu_xuio_init(xuio,
4986                     (preamble != 0) + fullblk + (postamble != 0));
4987                 DTRACE_PROBE3(zfs_reqzcbuf_align, int, preamble,
4988                     int, postamble, int,
4989                     (preamble != 0) + fullblk + (postamble != 0));
4990
4991                 /*
4992                  * Have to fix iov base/len for partial buffers.  They
4993                  * currently represent full arc_buf's.
4994                  */
4995                 if (preamble) {
```



```
4996                         /* data begins in the middle of the arc_buf */
4997                         abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
4998                             blksz);
4999                         ASSERT(abuf);
5000                         (void) dmu_xuio_add(xuio, abuf,
5001                             blksz - preamble, preamble);
5002                 }
5003
5004                 for (i = 0; i < fullblk; i++) {
5005                         abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
5006                             blksz);
5007                         ASSERT(abuf);
5008                         (void) dmu_xuio_add(xuio, abuf, 0, blksz);
5009                 }
5010
5011                 if (postamble) {
5012                         /* data ends in the middle of the arc_buf */
5013                         abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
5014                             blksz);
5015                         ASSERT(abuf);
5016                         (void) dmu_xuio_add(xuio, abuf, 0, postamble);
5017                 }
5018                 break;
5019         case UIO_READ:
5020                 /*
5021                  * Loan out an arc_buf for read if the read size is larger than
5022                  * the current file block size.  Block alignment is not
5023                  * considered.  Partial arc_buf will be loaned out for read.
5024                  */
5025                 blksz = zp->z_blksz;
5026                 if (blksz < zcr_blksz_min)
5027                         blksz = zcr_blksz_min;
5028                 if (blksz > zcr_blksz_max)
5029                         blksz = zcr_blksz_max;
5030                 /* avoid potential complexity of dealing with it */
5031                 if (blksz > max_blksz) {
5032                         ZFS_EXIT(zfsvfs);
5033                         return (SET_ERROR(EINVAL));
5034                 }
5035
5036                 maxsize = zp->z_size - uio->uio_loffset;
5037                 if (size > maxsize)
5038                         size = maxsize;
5039
5040                 if (size < blksz || vn_has_cached_data(vp)) {
5041                         ZFS_EXIT(zfsvfs);
5042                         return (SET_ERROR(EINVAL));
5043                 }
5044                 break;
5045         default:
5046                 ZFS_EXIT(zfsvfs);
5047                 return (SET_ERROR(EINVAL));
5048         }
5049
5050         uio->uio_extflg = UIO_XUIO;
5051         XUIO_XUZC_RW(xuio) = ioflag;
5052         ZFS_EXIT(zfsvfs);
5053         return (0);
5054 }
5055
5056 /*ARGSUSED*/
5057 static int
5058 zfs_retzcbuf(vnode_t *vp, xuio_t *xuio, cred_t *cr, caller_context_t *ct)
5059 {
5060         int i;
5061         arc_buf_t *abuf;
```

```
5062         int ioflag = XUIO_XUZC_RW(xuio);

5064         ASSERT(xuio->xu_type == UIOTYPE_ZEROCOPY);

5066         i = dmu_xuio_cnt(xuio);
5067         while (i-- > 0) {
5068                 abuf = dmu_xuio_arcbuf(xuio, i);
5069                 /*
5070                  * if abuf == NULL, it must be a write buffer
5071                  * that has been returned in zfs_write().
5072                  */
5073                 if (abuf)
5074                         dmu_return_arcbuf(abuf);
5075                 ASSERT(abuf || ioflag == UIO_WRITE);
5076         }

5078         dmu_xuio_fini(xuio);
5079         return (0);
5080 }

5082 /*
5083  * Predeclare these here so that the compiler assumes that
5084  * this is an "old style" function declaration that does
5085  * not include arguments => we won't get type mismatch errors
5086  * in the initializations that follow.
5087  */
5088 static int zfs_inval();
5089 static int zfs_isdir();

5091 static int
5092 zfs_inval()
5093 {
5094         return (SET_ERROR(EINVAL));
5095 }

5097 static int
5098 zfs_isdir()
5099 {
5100         return (SET_ERROR(EISDIR));
5101 }
5102 /*
5103  * Directory vnode operations template
5104  */
5105 vnodeops_t *zfs_dvnodeops;
5106 const fs_operation_def_t zfs_dvnodeops_template[] = {
5107         VOPNAME_OPEN,           { .vop_open = zfs_open },
5108         VOPNAME_CLOSE,          { .vop_close = zfs_close },
5109         VOPNAME_READ,           { .error = zfs_isdir },
5110         VOPNAME_WRITE,          { .error = zfs_isdir },
5111         VOPNAME_IOCTL,          { .vop_ioctl = zfs_ioctl },
5112         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5113         VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5114         VOPNAME_ACCESS,         { .vop_access = zfs_access },
5115         VOPNAME_LOOKUP,         { .vop_lookup = zfs_lookup },
5116         VOPNAME_CREATE,         { .vop_create = zfs_create },
5117         VOPNAME_REMOVE,         { .vop_remove = zfs_remove },
5118         VOPNAME_LINK,           { .vop_link = zfs_link },
5119         VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5120         VOPNAME_MKDIR,          { .vop_mkdir = zfs_mkdir },
5121         VOPNAME_RMDIR,          { .vop_rmdir = zfs_rmdir },
5122         VOPNAME_READDIR,        { .vop_readdir = zfs_readdir },
5123         VOPNAME_SYMLINK,        { .vop_symlink = zfs_symlink },
5124         VOPNAME_FSYNC,          { .vop_fsync = zfs_fsync },
5125         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5126         VOPNAME_FID,            { .vop_fid = zfs_fid },
5127         VOPNAME_SEEK,           { .vop_seek = zfs_seek },
```



```
5128         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5129         VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5130         VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5131         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5132         NULL,                   NULL
5133 };

5135 /*
5136  * Regular file vnode operations template
5137  */
5138 vnodeops_t *zfs_fvnodeops;
5139 const fs_operation_def_t zfs_fvnodeops_template[] = {
5140         VOPNAME_OPEN,           { .vop_open = zfs_open },
5141         VOPNAME_CLOSE,          { .vop_close = zfs_close },
5142         VOPNAME_READ,           { .vop_read = zfs_read },
5143         VOPNAME_WRITE,          { .vop_write = zfs_write },
5144         VOPNAME_IOCTL,          { .vop_ioctl = zfs_ioctl },
5145         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5146         VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5147         VOPNAME_ACCESS,         { .vop_access = zfs_access },
5148         VOPNAME_LOOKUP,         { .vop_lookup = zfs_lookup },
5149         VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5150         VOPNAME_FSYNC,          { .vop_fsync = zfs_fsync },
5151         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5152         VOPNAME_FID,            { .vop_fid = zfs_fid },
5153         VOPNAME_SEEK,           { .vop_seek = zfs_seek },
5154         VOPNAME_FRLOCK,         { .vop_frlock = zfs_frlock },
5155         VOPNAME_SPACE,          { .vop_space = zfs_space },
5156         VOPNAME_GETPAGE,        { .vop_getpage = zfs_getpage },
5157         VOPNAME_PUTPAGE,        { .vop_putpage = zfs_putpage },
5158         VOPNAME_MAP,            { .vop_map = zfs_map },
5159         VOPNAME_ADDMAP,         { .vop_addmap = zfs_addmap },
5160         VOPNAME_DELMAP,         { .vop_delmap = zfs_delmap },
5161         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5162         VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5163         VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5164         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5165         VOPNAME_REQZCBUF,       { .vop_reqzcbuf = zfs_reqzcbuf },
5166         VOPNAME_RETZCBUF,       { .vop_retzcbuf = zfs_retzcbuf },
5167         NULL,                   NULL
5168 };

5170 /*
5171  * Symbolic link vnode operations template
5172  */
5173 vnodeops_t *zfs_symvnodeops;
5174 const fs_operation_def_t zfs_symvnodeops_template[] = {
5175         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5176         VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5177         VOPNAME_ACCESS,         { .vop_access = zfs_access },
5178         VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5179         VOPNAME_READLINK,       { .vop_readlink = zfs_readlink },
5180         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5181         VOPNAME_FID,            { .vop_fid = zfs_fid },
5182         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5183         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5184         NULL,                   NULL
5185 };

5187 /*
5188  * special share hidden files vnode operations template
5189  */
5190 vnodeops_t *zfs_sharevnodeops;
5191 const fs_operation_def_t zfs_sharevnodeops_template[] = {
5192         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5193         VOPNAME_ACCESS,         { .vop_access = zfs_access },
```

```
5194         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5195         VOPNAME_FID,            { .vop_fid = zfs_fid },
5196         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5197         VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5198         VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5199         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5200         NULL,                   NULL
5201 };

5203 /*
5204  * Extended attribute directory vnode operations template
5205  *
5206 #endif /* ! codereview */
5207  * This template is identical to the directory vnodes
5208  * operation template except for restricted operations:
5209  *      VOP_MKDIR()
5210  *      VOP_SYMLINK()
5211  *
5212 #endif /* ! codereview */
5213  * Note that there are other restrictions embedded in:
5214  *      zfs_create()    - restrict type to VREG
5215  *      zfs_link()      - no links into/out of attribute space
5216  *      zfs_rename()    - no moves into/out of attribute space
5217  */
5218 vnodeops_t *zfs_xdvnodeops;
5219 const fs_operation_def_t zfs_xdvnodeops_template[] = {
5220         VOPNAME_OPEN,           { .vop_open = zfs_open },
5221         VOPNAME_CLOSE,          { .vop_close = zfs_close },
5222         VOPNAME_IOCTL,          { .vop_ioctl = zfs_ioctl },
5223         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5224         VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5225         VOPNAME_ACCESS,         { .vop_access = zfs_access },
5226         VOPNAME_LOOKUP,         { .vop_lookup = zfs_lookup },
5227         VOPNAME_CREATE,         { .vop_create = zfs_create },
5228         VOPNAME_REMOVE,         { .vop_remove = zfs_remove },
5229         VOPNAME_LINK,           { .vop_link = zfs_link },
5230         VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5231         VOPNAME_MKDIR,          { .error = zfs_inval },
5232         VOPNAME_RMDIR,          { .vop_rmdir = zfs_rmdir },
5233         VOPNAME_READDIR,        { .vop_readdir = zfs_readdir },
5234         VOPNAME_SYMLINK,        { .error = zfs_inval },
5235         VOPNAME_FSYNC,          { .vop_fsync = zfs_fsync },
5236         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5237         VOPNAME_FID,            { .vop_fid = zfs_fid },
5238         VOPNAME_SEEK,           { .vop_seek = zfs_seek },
5239         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5240         VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5241         VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5242         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5243         NULL,                   NULL
5244 };

5246 /*
5247  * Error vnode operations template
5248  */
5249 vnodeops_t *zfs_evnodeops;
5250 const fs_operation_def_t zfs_evnodeops_template[] = {
5251         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5252         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5253         NULL,                   NULL
5254 };
```

```
**********************************************************
   53484 Thu May 16 17:36:27 2013
new/usr/src/uts/common/fs/zfs/zfs_znode.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1008 /*
1009  * Update in-core attributes.  It is assumed the caller will be doing an
1010  * sa_bulk_update to push the changes out.
1009  * zfs_xvattr_set only updates the in-core attributes
1010  * it is assumed the caller will be doing an sa_bulk_update
1011  * to push the changes out
1011  */
1012 void
1013 zfs_xvattr_set(znode_t *zp, xvattr_t *xvap, dmu_tx_t *tx)
1014 {
1015         xoptattr_t *xoap;

1017         xoap = xva_getxoptattr(xvap);
1018         ASSERT(xoap);

1020         if (XVA_ISSET_REQ(xvap, XAT_CREATETIME)) {
1021                 uint64_t times[2];
1022                 ZFS_TIME_ENCODE(&xoap->xoa_createtime, times);
1023                 (void) sa_update(zp->z_sa_hdl, SA_ZPL_CRTIME(zp->z_zfsvfs),
1024                     &times, sizeof (times), tx);
1025                 XVA_SET_RTN(xvap, XAT_CREATETIME);
1026         }
1027         if (XVA_ISSET_REQ(xvap, XAT_READONLY)) {
1028                 ZFS_ATTR_SET(zp, ZFS_READONLY, xoap->xoa_readonly,
1029                     zp->z_pflags, tx);
1030                 XVA_SET_RTN(xvap, XAT_READONLY);
1031         }
1032         if (XVA_ISSET_REQ(xvap, XAT_HIDDEN)) {
1033                 ZFS_ATTR_SET(zp, ZFS_HIDDEN, xoap->xoa_hidden,
1034                     zp->z_pflags, tx);
1035                 XVA_SET_RTN(xvap, XAT_HIDDEN);
1036         }
1037         if (XVA_ISSET_REQ(xvap, XAT_SYSTEM)) {
1038                 ZFS_ATTR_SET(zp, ZFS_SYSTEM, xoap->xoa_system,
1039                     zp->z_pflags, tx);
1040                 XVA_SET_RTN(xvap, XAT_SYSTEM);
1041         }
1042         if (XVA_ISSET_REQ(xvap, XAT_ARCHIVE)) {
1043                 ZFS_ATTR_SET(zp, ZFS_ARCHIVE, xoap->xoa_archive,
1044                     zp->z_pflags, tx);
1045                 XVA_SET_RTN(xvap, XAT_ARCHIVE);
1046         }
1047         if (XVA_ISSET_REQ(xvap, XAT_IMMUTABLE)) {
1048                 ZFS_ATTR_SET(zp, ZFS_IMMUTABLE, xoap->xoa_immutable,
1049                     zp->z_pflags, tx);
1050                 XVA_SET_RTN(xvap, XAT_IMMUTABLE);
1051         }
1052         if (XVA_ISSET_REQ(xvap, XAT_NOUNLINK)) {
1053                 ZFS_ATTR_SET(zp, ZFS_NOUNLINK, xoap->xoa_nounlink,
1054                     zp->z_pflags, tx);
1055                 XVA_SET_RTN(xvap, XAT_NOUNLINK);
1056         }
1057         if (XVA_ISSET_REQ(xvap, XAT_APPENDONLY)) {
1058                 ZFS_ATTR_SET(zp, ZFS_APPENDONLY, xoap->xoa_appendonly,
```



```
1059                     zp->z_pflags, tx);
1060                 XVA_SET_RTN(xvap, XAT_APPENDONLY);
1061         }
1062         if (XVA_ISSET_REQ(xvap, XAT_NODUMP)) {
1063                 ZFS_ATTR_SET(zp, ZFS_NODUMP, xoap->xoa_nodump,
1064                     zp->z_pflags, tx);
1065                 XVA_SET_RTN(xvap, XAT_NODUMP);
1066         }
1067         if (XVA_ISSET_REQ(xvap, XAT_OPAQUE)) {
1068                 ZFS_ATTR_SET(zp, ZFS_OPAQUE, xoap->xoa_opaque,
1069                     zp->z_pflags, tx);
1070                 XVA_SET_RTN(xvap, XAT_OPAQUE);
1071         }
1072         if (XVA_ISSET_REQ(xvap, XAT_AV_QUARANTINED)) {
1073                 ZFS_ATTR_SET(zp, ZFS_AV_QUARANTINED,
1074                     xoap->xoa_av_quarantined, zp->z_pflags, tx);
1075                 XVA_SET_RTN(xvap, XAT_AV_QUARANTINED);
1076         }
1077         if (XVA_ISSET_REQ(xvap, XAT_AV_MODIFIED)) {
1078                 ZFS_ATTR_SET(zp, ZFS_AV_MODIFIED, xoap->xoa_av_modified,
1079                     zp->z_pflags, tx);
1080                 XVA_SET_RTN(xvap, XAT_AV_MODIFIED);
1081         }
1082         if (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP)) {
1083                 zfs_sa_set_scanstamp(zp, xvap, tx);
1084                 XVA_SET_RTN(xvap, XAT_AV_SCANSTAMP);
1085         }
1086         if (XVA_ISSET_REQ(xvap, XAT_REPARSE)) {
1087                 ZFS_ATTR_SET(zp, ZFS_REPARSE, xoap->xoa_reparse,
1088                     zp->z_pflags, tx);
1089                 XVA_SET_RTN(xvap, XAT_REPARSE);
1090         }
1091         if (XVA_ISSET_REQ(xvap, XAT_OFFLINE)) {
1092                 ZFS_ATTR_SET(zp, ZFS_OFFLINE, xoap->xoa_offline,
1093                     zp->z_pflags, tx);
1094                 XVA_SET_RTN(xvap, XAT_OFFLINE);
1095         }
1096         if (XVA_ISSET_REQ(xvap, XAT_SPARSE)) {
1097                 ZFS_ATTR_SET(zp, ZFS_SPARSE, xoap->xoa_sparse,
1098                     zp->z_pflags, tx);
1099                 XVA_SET_RTN(xvap, XAT_SPARSE);
1100         }
1101 }
______unchanged_portion_omitted_

1443 /*
1444  * Increase the file length
1445  *
1446  *      IN:     zp      - znode of file to free data in.
1447  *              end     - new end-of-file
1448  *
1449  *      RETURN: 0 on success, error code on failure
1450  *      RETURN: 0 if success
1451  *              error code if failure
1450  */
1451 static int
1452 zfs_extend(znode_t *zp, uint64_t end)
1453 {
1454         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
1455         dmu_tx_t *tx;
1456         rl_t *rl;
1457         uint64_t newblksz;
1458         int error;

1460         /*
1461          * We will change zp_size, lock the whole file.
```

```
1462          */
1463         rl = zfs_range_lock(zp, 0, UINT64_MAX, RL_WRITER);

1465         /*
1466          * Nothing to do if file already at desired length.
1467          */
1468         if (end <= zp->z_size) {
1469                 zfs_range_unlock(rl);
1470                 return (0);
1471         }
1472 top:
1473         tx = dmu_tx_create(zfsvfs->z_os);
1474         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1475         zfs_sa_upgrade_txholds(tx, zp);
1476         if (end > zp->z_blksz &&
1477             (!ISP2(zp->z_blksz) || zp->z_blksz < zfsvfs->z_max_blksz)) {
1478                 /*
1479                  * We are growing the file past the current block size.
1480                  */
1481                 if (zp->z_blksz > zp->z_zfsvfs->z_max_blksz) {
1482                         ASSERT(!ISP2(zp->z_blksz));
1483                         newblksz = MIN(end, SPA_MAXBLOCKSIZE);
1484                 } else {
1485                         newblksz = MIN(end, zp->z_zfsvfs->z_max_blksz);
1486                 }
1487                 dmu_tx_hold_write(tx, zp->z_id, 0, newblksz);
1488         } else {
1489                 newblksz = 0;
1490         }

1492         error = dmu_tx_assign(tx, TXG_NOWAIT);
1493         if (error) {
1494                 if (error == ERESTART) {
1495                         dmu_tx_wait(tx);
1496                         dmu_tx_abort(tx);
1497                         goto top;
1498                 }
1499                 dmu_tx_abort(tx);
1500                 zfs_range_unlock(rl);
1501                 return (error);
1502         }

1504         if (newblksz)
1505                 zfs_grow_blocksize(zp, newblksz, tx);

1507         zp->z_size = end;

1509         VERIFY(0 == sa_update(zp->z_sa_hdl, SA_ZPL_SIZE(zp->z_zfsvfs),
1510             &zp->z_size, sizeof (zp->z_size), tx));

1512         zfs_range_unlock(rl);

1514         dmu_tx_commit(tx);

1516         return (0);
1517 }

1519 /*
1520  * Free space in a file.
1521  *
1522  *      IN:     zp      - znode of file to free data in.
1523  *              off     - start of section to free.
1524  *              len     - length of section to free.
1525  *
1526  *      RETURN: 0 on success, error code on failure
1528  *      RETURN: 0 if success
```



```
1529  *              error code if failure
1527  */
1528 static int
1529 zfs_free_range(znode_t *zp, uint64_t off, uint64_t len)
1530 {
1531         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
1532         rl_t *rl;
1533         int error;

1535         /*
1536          * Lock the range being freed.
1537          */
1538         rl = zfs_range_lock(zp, off, len, RL_WRITER);

1540         /*
1541          * Nothing to do if file already at desired length.
1542          */
1543         if (off >= zp->z_size) {
1544                 zfs_range_unlock(rl);
1545                 return (0);
1546         }

1548         if (off + len > zp->z_size)
1549                 len = zp->z_size - off;

1551         error = dmu_free_long_range(zfsvfs->z_os, zp->z_id, off, len);

1553         zfs_range_unlock(rl);

1555         return (error);
1556 }

1558 /*
1559  * Truncate a file
1560  *
1561  *      IN:     zp      - znode of file to free data in.
1562  *              end     - new end-of-file.
1563  *
1564  *      RETURN: 0 on success, error code on failure
1567  *      RETURN: 0 if success
1568  *              error code if failure
1565  */
1566 static int
1567 zfs_trunc(znode_t *zp, uint64_t end)
1568 {
1569         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
1570         vnode_t *vp = ZTOV(zp);
1571         dmu_tx_t *tx;
1572         rl_t *rl;
1573         int error;
1574         sa_bulk_attr_t bulk[2];
1575         int count = 0;

1577         /*
1578          * We will change zp_size, lock the whole file.
1579          */
1580         rl = zfs_range_lock(zp, 0, UINT64_MAX, RL_WRITER);

1582         /*
1583          * Nothing to do if file already at desired length.
1584          */
1585         if (end >= zp->z_size) {
1586                 zfs_range_unlock(rl);
1587                 return (0);
1588         }
```

```
1590         error = dmu_free_long_range(zfsvfs->z_os, zp->z_id, end,  -1);
1591         if (error) {
1592                 zfs_range_unlock(rl);
1593                 return (error);
1594         }
1595 top:
1596         tx = dmu_tx_create(zfsvfs->z_os);
1597         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1598         zfs_sa_upgrade_txholds(tx, zp);
1599         error = dmu_tx_assign(tx, TXG_NOWAIT);
1600         if (error) {
1601                 if (error == ERESTART) {
1602                         dmu_tx_wait(tx);
1603                         dmu_tx_abort(tx);
1604                         goto top;
1605                 }
1606                 dmu_tx_abort(tx);
1607                 zfs_range_unlock(rl);
1608                 return (error);
1609         }

1611         zp->z_size = end;
1612         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_SIZE(zfsvfs),
1613             NULL, &zp->z_size, sizeof (zp->z_size));

1615         if (end == 0) {
1616                 zp->z_pflags &= ~ZFS_SPARSE;
1617                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs),
1618                     NULL, &zp->z_pflags, 8);
1619         }
1620         VERIFY(sa_bulk_update(zp->z_sa_hdl, bulk, count, tx) == 0);

1622         dmu_tx_commit(tx);

1624         /*
1625          * Clear any mapped pages in the truncated region.  This has to
1626          * happen outside of the transaction to avoid the possibility of
1627          * a deadlock with someone trying to push a page that we are
1628          * about to invalidate.
1629          */
1630         if (vn_has_cached_data(vp)) {
1631                 page_t *pp;
1632                 uint64_t start = end & PAGEMASK;
1633                 int poff = end & PAGEOFFSET;

1635                 if (poff != 0 && (pp = page_lookup(vp, start, SE_SHARED))) {
1636                         /*
1637                          * We need to zero a partial page.
1638                          */
1639                         pagezero(pp, poff, PAGESIZE - poff);
1640                         start += PAGESIZE;
1641                         page_unlock(pp);
1642                 }
1643                 error = pvn_vplist_dirty(vp, start, zfs_no_putpage,
1644                     B_INVAL | B_TRUNC, NULL);
1645                 ASSERT(error == 0);
1646         }

1648         zfs_range_unlock(rl);

1650         return (0);
1651 }

1653 /*
1654  * Free space in a file
1655  *
```



```
1656  *      IN:     zp      - znode of file to free data in.
1657  *              off     - start of range
1658  *              len     - end of range (0 => EOF)
1659  *              flag    - current file open mode flags.
1660  *              log     - TRUE if this action should be logged
1661  *
1662  *      RETURN: 0 on success, error code on failure
1666  *      RETURN: 0 if success
1667  *              error code if failure
1663  */
1664 int
1665 zfs_freesp(znode_t *zp, uint64_t off, uint64_t len, int flag, boolean_t log)
1666 {
1667         vnode_t *vp = ZTOV(zp);
1668         dmu_tx_t *tx;
1669         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
1670         zilog_t *zilog = zfsvfs->z_log;
1671         uint64_t mode;
1672         uint64_t mtime[2], ctime[2];
1673         sa_bulk_attr_t bulk[3];
1674         int count = 0;
1675         int error;

1677         if ((error = sa_lookup(zp->z_sa_hdl, SA_ZPL_MODE(zfsvfs), &mode,
1678             sizeof (mode))) != 0)
1679                 return (error);

1681         if (off > zp->z_size) {
1682                 error =  zfs_extend(zp, off+len);
1683                 if (error == 0 && log)
1684                         goto log;
1685                 else
1686                         return (error);
1687         }

1689         /*
1690          * Check for any locks in the region to be freed.
1691          */

1693         if (MANDLOCK(vp, (mode_t)mode)) {
1694                 uint64_t length = (len ? len : zp->z_size - off);
1695                 if (error = chklock(vp, FWRITE, off, length, flag, NULL))
1696                         return (error);
1697         }

1699         if (len == 0) {
1700                 error = zfs_trunc(zp, off);
1701         } else {
1702                 if ((error = zfs_free_range(zp, off, len)) == 0 &&
1703                     off + len > zp->z_size)
1704                         error = zfs_extend(zp, off+len);
1705         }
1706         if (error || !log)
1707                 return (error);
1708 log:
1709         tx = dmu_tx_create(zfsvfs->z_os);
1710         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1711         zfs_sa_upgrade_txholds(tx, zp);
1712         error = dmu_tx_assign(tx, TXG_NOWAIT);
1713         if (error) {
1714                 if (error == ERESTART) {
1715                         dmu_tx_wait(tx);
1716                         dmu_tx_abort(tx);
1717                         goto log;
1718                 }
1719                 dmu_tx_abort(tx);
```

```
1720                 return (error);
1721         }

1723         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, mtime, 16);
1724         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, ctime, 16);
1725         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs),
1726             NULL, &zp->z_pflags, 8);
1727         zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime, B_TRUE);
1728         error = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);
1729         ASSERT(error == 0);

1731         zfs_log_truncate(zilog, tx, TX_TRUNCATE, zp, off, len);

1733         dmu_tx_commit(tx);
1734         return (0);
1735 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   57657 Thu May 16 17:36:28 2013
new/usr/src/uts/common/fs/zfs/zil.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

/* Portions Copyright 2010 Robert Milkowski */

#include <sys/zfs_context.h>
#include <sys/spa.h>
#include <sys/dmu.h>
#include <sys/zap.h>
#include <sys/arc.h>
#include <sys/stat.h>
#include <sys/resource.h>
#include <sys/zil.h>
#include <sys/zil_impl.h>
#include <sys/dsl_dataset.h>
#include <sys/vdev_impl.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_pool.h>

/*
 * The zfs intent log (ZIL) saves transaction records of system calls
 * that change the file system in memory with enough information
 * to be able to replay them. These are stored in memory until
 * either the DMU transaction group (txg) commits them to the stable pool
 * and they can be discarded, or they are flushed to the stable log
 * (also in the pool) due to a fsync, O_DSYNC or other synchronous
 * requirement. In the event of a panic or power fail then those log
 * records (transactions) are replayed.
 *
 * There is one ZIL per file system. Its on-disk (pool) format consists
 * of 3 parts:
 *
 *      - ZIL header
 *      - ZIL blocks
```



```
 *      - ZIL records
 *
 * A log record holds a system call transaction. Log blocks can
 * hold many log records and the blocks are chained together.
 * Each ZIL block contains a block pointer (blkptr_t) to the next
 * ZIL block in the chain. The ZIL header points to the first
 * block in the chain. Note there is not a fixed place in the pool
 * to hold blocks. They are dynamically allocated and freed as
 * needed from the blocks available. Figure X shows the ZIL structure:
 */

/*
 * Disable intent logging replay.  This global ZIL switch affects all pools.
 * This global ZIL switch affects all pools
 */
int zil_replay_disable = 0;
int zil_replay_disable = 0;    /* disable intent logging replay */

/*
 * Tunable parameter for debugging or performance analysis.  Setting
 * zfs_nocacheflush will cause corruption on power loss if a volatile
 * out-of-order write cache is enabled.
 */
boolean_t zfs_nocacheflush = B_FALSE;

static kmem_cache_t *zil_lwb_cache;

static void zil_async_to_sync(zilog_t *zilog, uint64_t foid);

#define LWB_EMPTY(lwb) ((BP_GET_LSIZE(&lwb->lwb_blk) - \
    sizeof (zil_chain_t)) == (lwb->lwb_sz - lwb->lwb_nused))


/*
 * ziltest is by and large an ugly hack, but very useful in
 * checking replay without tedious work.
 * When running ziltest we want to keep all itx's and so maintain
 * a single list in the zl_itxg[] that uses a high txg: ZILTEST_TXG
 * We subtract TXG_CONCURRENT_STATES to allow for common code.
 */
#define ZILTEST_TXG (UINT64_MAX - TXG_CONCURRENT_STATES)

static int
zil_bp_compare(const void *x1, const void *x2)
{
        const dva_t *dva1 = &((zil_bp_node_t *)x1)->zn_dva;
        const dva_t *dva2 = &((zil_bp_node_t *)x2)->zn_dva;

        if (DVA_GET_VDEV(dva1) < DVA_GET_VDEV(dva2))
                return (-1);
        if (DVA_GET_VDEV(dva1) > DVA_GET_VDEV(dva2))
                return (1);

        if (DVA_GET_OFFSET(dva1) < DVA_GET_OFFSET(dva2))
                return (-1);
        if (DVA_GET_OFFSET(dva1) > DVA_GET_OFFSET(dva2))
                return (1);

        return (0);
}
_____unchanged_portion_omitted_

/*
 * Define a limited set of intent log block sizes.
 *
#endif /* ! codereview */
```

```
 * These must be a multiple of 4KB. Note only the amount used (again
 * aligned to 4KB) actually gets written. However, we can't always just
 * allocate SPA_MAXBLOCKSIZE as the slog space could be exhausted.
 */
uint64_t zil_block_buckets[] = {
    4096,               /* non TX_WRITE */
    8192+4096,          /* data base */
    32*1024 + 4096,     /* NFS writes */
    UINT64_MAX
};

/*
 * Use the slog as long as the logbias is 'latency' and the current commit size
 * is less than the limit or the total list size is less than 2X the limit.
 * Limit checking is disabled by setting zil_slog_limit to UINT64_MAX.
 */
uint64_t zil_slog_limit = 1024 * 1024;
#define USE_SLOG(zilog) (((zilog)->zl_logbias == ZFS_LOGBIAS_LATENCY) && \
        (((zilog)->zl_cur_used < zil_slog_limit) || \
        ((zilog)->zl_itx_list_sz < (zil_slog_limit << 1))))

/*
 * Start a log block write and advance to the next log block.
 * Calls are serialized.
 */
static lwb_t *
zil_lwb_write_start(zilog_t *zilog, lwb_t *lwb)
{
        lwb_t *nlwb = NULL;
        zil_chain_t *zilc;
        spa_t *spa = zilog->zl_spa;
        blkptr_t *bp;
        dmu_tx_t *tx;
        uint64_t txg;
        uint64_t zil_blksz, wsz;
        int i, error;

        if (BP_GET_CHECKSUM(&lwb->lwb_blk) == ZIO_CHECKSUM_ZILOG2) {
                zilc = (zil_chain_t *)lwb->lwb_buf;
                bp = &zilc->zc_next_blk;
        } else {
                zilc = (zil_chain_t *)(lwb->lwb_buf + lwb->lwb_sz);
                bp = &zilc->zc_next_blk;
        }

        ASSERT(lwb->lwb_nused <= lwb->lwb_sz);

        /*
         * Allocate the next block and save its address in this block
         * before writing it in order to establish the log chain.
         * Note that if the allocation of nlwb synced before we wrote
         * the block that points at it (lwb), we'd leak it if we crashed.
         * Therefore, we don't do dmu_tx_commit() until zil_lwb_write_done().
         * We dirty the dataset to ensure that zil_sync() will be called
         * to clean up in the event of allocation failure or I/O failure.
         */
        tx = dmu_tx_create(zilog->zl_os);
        VERIFY(dmu_tx_assign(tx, TXG_WAIT) == 0);
        dsl_dataset_dirty(dmu_objset_ds(zilog->zl_os), tx);
        txg = dmu_tx_get_txg(tx);

        lwb->lwb_tx = tx;

        /*
         * Log blocks are pre-allocated. Here we select the size of the next
         * block, based on size used in the last block.
```



```
         * - first find the smallest bucket that will fit the block from a
         *   limited set of block sizes. This is because it's faster to write
         *   blocks allocated from the same metaslab as they are adjacent or
         *   close.
         * - next find the maximum from the new suggested size and an array of
         *   previous sizes. This lessens a picket fence effect of wrongly
         *   guesssing the size if we have a stream of say 2k, 64k, 2k, 64k
         *   requests.
         *
         * Note we only write what is used, but we can't just allocate
         * the maximum block size because we can exhaust the available
         * pool log space.
         */
        zil_blksz = zilog->zl_cur_used + sizeof (zil_chain_t);
        for (i = 0; zil_blksz > zil_block_buckets[i]; i++)
                continue;
        zil_blksz = zil_block_buckets[i];
        if (zil_blksz == UINT64_MAX)
                zil_blksz = SPA_MAXBLOCKSIZE;
        zilog->zl_prev_blks[zilog->zl_prev_rotor] = zil_blksz;
        for (i = 0; i < ZIL_PREV_BLKS; i++)
                zil_blksz = MAX(zil_blksz, zilog->zl_prev_blks[i]);
        zilog->zl_prev_rotor = (zilog->zl_prev_rotor + 1) & (ZIL_PREV_BLKS - 1);

        BP_ZERO(bp);
        /* pass the old blkptr in order to spread log blocks across devs */
        error = zio_alloc_zil(spa, txg, bp, &lwb->lwb_blk, zil_blksz,
            USE_SLOG(zilog));
        if (error == 0) {
                ASSERT3U(bp->blk_birth, ==, txg);
                bp->blk_cksum = lwb->lwb_blk.blk_cksum;
                bp->blk_cksum.zc_word[ZIL_ZC_SEQ]++;

                /*
                 * Allocate a new log write buffer (lwb).
                 */
                nlwb = zil_alloc_lwb(zilog, bp, txg);

                /* Record the block for later vdev flushing */
                zil_add_block(zilog, &lwb->lwb_blk);
        }

        if (BP_GET_CHECKSUM(&lwb->lwb_blk) == ZIO_CHECKSUM_ZILOG2) {
                /* For Slim ZIL only write what is used. */
                wsz = P2ROUNDUP_TYPED(lwb->lwb_nused, ZIL_MIN_BLKSZ, uint64_t);
                ASSERT3U(wsz, <=, lwb->lwb_sz);
                zio_shrink(lwb->lwb_zio, wsz);

        } else {
                wsz = lwb->lwb_sz;
        }

        zilc->zc_pad = 0;
        zilc->zc_nused = lwb->lwb_nused;
        zilc->zc_eck.zec_cksum = lwb->lwb_blk.blk_cksum;

        /*
         * clear unused data for security
         */
        bzero(lwb->lwb_buf + lwb->lwb_nused, wsz - lwb->lwb_nused);

        zio_nowait(lwb->lwb_zio); /* Kick off the write for the old log block */

        /*
         * If there was an allocation failure then nlwb will be null which
         * forces a txg_wait_synced().
```

```
1016          */
1017         return (nlwb);
1018 }

1020 static lwb_t *
1021 zil_lwb_commit(zilog_t *zilog, itx_t *itx, lwb_t *lwb)
1022 {
1023         lr_t *lrc = &itx->itx_lr; /* common log record */
1024         lr_write_t *lrw = (lr_write_t *)lrc;
1025         char *lr_buf;
1026         uint64_t txg = lrc->lrc_txg;
1027         uint64_t reclen = lrc->lrc_reclen;
1028         uint64_t dlen = 0;

1030         if (lwb == NULL)
1031                 return (NULL);

1033         ASSERT(lwb->lwb_buf != NULL);
1034         ASSERT(zilog_is_dirty(zilog) ||
1035             spa_freeze_txg(zilog->zl_spa) != UINT64_MAX);

1037         if (lrc->lrc_txtype == TX_WRITE && itx->itx_wr_state == WR_NEED_COPY)
1038                 dlen = P2ROUNDUP_TYPED(
1039                     lrw->lr_length, sizeof (uint64_t), uint64_t);

1041         zilog->zl_cur_used += (reclen + dlen);

1043         zil_lwb_write_init(zilog, lwb);

1045         /*
1046          * If this record won't fit in the current log block, start a new one.
1047          */
1048         if (lwb->lwb_nused + reclen + dlen > lwb->lwb_sz) {
1049                 lwb = zil_lwb_write_start(zilog, lwb);
1050                 if (lwb == NULL)
1051                         return (NULL);
1052                 zil_lwb_write_init(zilog, lwb);
1053                 ASSERT(LWB_EMPTY(lwb));
1054                 if (lwb->lwb_nused + reclen + dlen > lwb->lwb_sz) {
1055                         txg_wait_synced(zilog->zl_dmu_pool, txg);
1056                         return (lwb);
1057                 }
1058         }

1060         lr_buf = lwb->lwb_buf + lwb->lwb_nused;
1061         bcopy(lrc, lr_buf, reclen);
1062         lrc = (lr_t *)lr_buf;
1063         lrw = (lr_write_t *)lrc;

1065         /*
1066          * If it's a write, fetch the data or get its blkptr as appropriate.
1067          */
1068         if (lrc->lrc_txtype == TX_WRITE) {
1069                 if (txg > spa_freeze_txg(zilog->zl_spa))
1070                         txg_wait_synced(zilog->zl_dmu_pool, txg);
1071                 if (itx->itx_wr_state != WR_COPIED) {
1072                         char *dbuf;
1073                         int error;

1075                         if (dlen) {
1076                                 ASSERT(itx->itx_wr_state == WR_NEED_COPY);
1077                                 dbuf = lr_buf + reclen;
1078                                 lrw->lr_common.lrc_reclen += dlen;
1079                         } else {
1080                                 ASSERT(itx->itx_wr_state == WR_INDIRECT);
1081                                 dbuf = NULL;
```



```
1082                         }
1083                         error = zilog->zl_get_data(
1084                             itx->itx_private, lrw, dbuf, lwb->lwb_zio);
1085                         if (error == EIO) {
1086                                 txg_wait_synced(zilog->zl_dmu_pool, txg);
1087                                 return (lwb);
1088                         }
1089                         if (error != 0) {
1090                                 ASSERT(error == ENOENT || error == EEXIST ||
1091                                     error == EALREADY);
1092                                 return (lwb);
1093                         }
1094                 }
1095         }

1097         /*
1098          * We're actually making an entry, so update lrc_seq to be the
1099          * log record sequence number.  Note that this is generally not
1100          * equal to the itx sequence number because not all transactions
1101          * are synchronous, and sometimes spa_sync() gets there first.
1102          */
1103         lrc->lrc_seq = ++zilog->zl_lr_seq; /* we are single threaded */
1104         lwb->lwb_nused += reclen + dlen;
1105         lwb->lwb_max_txg = MAX(lwb->lwb_max_txg, txg);
1106         ASSERT3U(lwb->lwb_nused, <=, lwb->lwb_sz);
1107         ASSERT0(P2PHASE(lwb->lwb_nused, sizeof (uint64_t)));

1109         return (lwb);
1110 }

1112 itx_t *
1113 zil_itx_create(uint64_t txtype, size_t lrsize)
1114 {
1115         itx_t *itx;

1117         lrsize = P2ROUNDUP_TYPED(lrsize, sizeof (uint64_t), size_t);

1119         itx = kmem_alloc(offsetof(itx_t, itx_lr) + lrsize, KM_SLEEP);
1120         itx->itx_lr.lrc_txtype = txtype;
1121         itx->itx_lr.lrc_reclen = lrsize;
1122         itx->itx_sod = lrsize; /* if write & WR_NEED_COPY will be increased */
1123         itx->itx_lr.lrc_seq = 0;        /* defensive */
1124         itx->itx_sync = B_TRUE;         /* default is synchronous */

1126         return (itx);
1127 }

1129 void
1130 zil_itx_destroy(itx_t *itx)
1131 {
1132         kmem_free(itx, offsetof(itx_t, itx_lr) + itx->itx_lr.lrc_reclen);
1133 }

1135 /*
1136  * Free up the sync and async itxs. The itxs_t has already been detached
1137  * so no locks are needed.
1138  */
1139 static void
1140 zil_itxg_clean(itxs_t *itxs)
1141 {
1142         itx_t *itx;
1143         list_t *list;
1144         avl_tree_t *t;
1145         void *cookie;
1146         itx_async_node_t *ian;
```

```
1148         list = &itxs->i_sync_list;
1149         while ((itx = list_head(list)) != NULL) {
1150                 list_remove(list, itx);
1151                 kmem_free(itx, offsetof(itx_t, itx_lr) +
1152                     itx->itx_lr.lrc_reclen);
1153         }
1155         cookie = NULL;
1156         t = &itxs->i_async_tree;
1157         while ((ian = avl_destroy_nodes(t, &cookie)) != NULL) {
1158                 list = &ian->ia_list;
1159                 while ((itx = list_head(list)) != NULL) {
1160                         list_remove(list, itx);
1161                         kmem_free(itx, offsetof(itx_t, itx_lr) +
1162                             itx->itx_lr.lrc_reclen);
1163                 }
1164                 list_destroy(list);
1165                 kmem_free(ian, sizeof (itx_async_node_t));
1166         }
1167         avl_destroy(t);
1169         kmem_free(itxs, sizeof (itxs_t));
1170 }
1172 static int
1173 zil_aitx_compare(const void *x1, const void *x2)
1174 {
1175         const uint64_t o1 = ((itx_async_node_t *)x1)->ia_foid;
1176         const uint64_t o2 = ((itx_async_node_t *)x2)->ia_foid;
1178         if (o1 < o2)
1179                 return (-1);
1180         if (o1 > o2)
1181                 return (1);
1183         return (0);
1184 }
1186 /*
1187  * Remove all async itx with the given oid.
1188  */
1189 static void
1190 zil_remove_async(zilog_t *zilog, uint64_t oid)
1191 {
1192         uint64_t otxg, txg;
1193         itx_async_node_t *ian;
1194         avl_tree_t *t;
1195         avl_index_t where;
1196         list_t clean_list;
1197         itx_t *itx;
1199         ASSERT(oid != 0);
1200         list_create(&clean_list, sizeof (itx_t), offsetof(itx_t, itx_node));
1202         if (spa_freeze_txg(zilog->zl_spa) != UINT64_MAX) /* ziltest support */
1203                 otxg = ZILTEST_TXG;
1204         else
1205                 otxg = spa_last_synced_txg(zilog->zl_spa) + 1;
1207         for (txg = otxg; txg < (otxg + TXG_CONCURRENT_STATES); txg++) {
1208                 itxg_t *itxg = &zilog->zl_itxg[txg & TXG_MASK];
1210                 mutex_enter(&itxg->itxg_lock);
1211                 if (itxg->itxg_txg != txg) {
1212                         mutex_exit(&itxg->itxg_lock);
1213                         continue;
```



```
1214                 }
1216                 /*
1217                  * Locate the object node and append its list.
1218                  */
1219                 t = &itxg->itxg_itxs->i_async_tree;
1220                 ian = avl_find(t, &oid, &where);
1221                 if (ian != NULL)
1222                         list_move_tail(&clean_list, &ian->ia_list);
1223                 mutex_exit(&itxg->itxg_lock);
1224         }
1225         while ((itx = list_head(&clean_list)) != NULL) {
1226                 list_remove(&clean_list, itx);
1227                 kmem_free(itx, offsetof(itx_t, itx_lr) +
1228                     itx->itx_lr.lrc_reclen);
1229         }
1230         list_destroy(&clean_list);
1231 }
1233 void
1234 zil_itx_assign(zilog_t *zilog, itx_t *itx, dmu_tx_t *tx)
1235 {
1236         uint64_t txg;
1237         itxg_t *itxg;
1238         itxs_t *itxs, *clean = NULL;
1240         /*
1241          * Object ids can be re-instantiated in the next txg so
1242          * remove any async transactions to avoid future leaks.
1243          * This can happen if a fsync occurs on the re-instantiated
1244          * object for a WR_INDIRECT or WR_NEED_COPY write, which gets
1245          * the new file data and flushes a write record for the old object.
1246          */
1247         if ((itx->itx_lr.lrc_txtype & ~TX_CI) == TX_REMOVE)
1248                 zil_remove_async(zilog, itx->itx_oid);
1250         /*
1251          * Ensure the data of a renamed file is committed before the rename.
1252          */
1253         if ((itx->itx_lr.lrc_txtype & ~TX_CI) == TX_RENAME)
1254                 zil_async_to_sync(zilog, itx->itx_oid);
1256         if (spa_freeze_txg(zilog->zl_spa) != UINT64_MAX)
1257                 txg = ZILTEST_TXG;
1258         else
1259                 txg = dmu_tx_get_txg(tx);
1261         itxg = &zilog->zl_itxg[txg & TXG_MASK];
1262         mutex_enter(&itxg->itxg_lock);
1263         itxs = itxg->itxg_itxs;
1264         if (itxg->itxg_txg != txg) {
1265                 if (itxs != NULL) {
1266                         /*
1267                          * The zil_clean callback hasn't got around to cleaning
1268                          * this itxg. Save the itxs for release below.
1269                          * This should be rare.
1270                          */
1271                         atomic_add_64(&zilog->zl_itx_list_sz, -itxg->itxg_sod);
1272                         itxg->itxg_sod = 0;
1273                         clean = itxg->itxg_itxs;
1274                 }
1275                 ASSERT(itxg->itxg_sod == 0);
1276                 itxg->itxg_txg = txg;
1277                 itxs = itxg->itxg_itxs = kmem_zalloc(sizeof (itxs_t), KM_SLEEP);
1279                 list_create(&itxs->i_sync_list, sizeof (itx_t),
```

```
1280                     offsetof(itx_t, itx_node));
1281                 avl_create(&itxs->i_async_tree, zil_aitx_compare,
1282                     sizeof (itx_async_node_t),
1283                     offsetof(itx_async_node_t, ia_node));
1284         }
1285         if (itx->itx_sync) {
1286                 list_insert_tail(&itxs->i_sync_list, itx);
1287                 atomic_add_64(&zilog->zl_itx_list_sz, itx->itx_sod);
1288                 itxg->itxg_sod += itx->itx_sod;
1289         } else {
1290                 avl_tree_t *t = &itxs->i_async_tree;
1291                 uint64_t foid = ((lr_ooo_t *)&itx->itx_lr)->lr_foid;
1292                 itx_async_node_t *ian;
1293                 avl_index_t where;

1295                 ian = avl_find(t, &foid, &where);
1296                 if (ian == NULL) {
1297                         ian = kmem_alloc(sizeof (itx_async_node_t), KM_SLEEP);
1298                         list_create(&ian->ia_list, sizeof (itx_t),
1299                             offsetof(itx_t, itx_node));
1300                         ian->ia_foid = foid;
1301                         avl_insert(t, ian, where);
1302                 }
1303                 list_insert_tail(&ian->ia_list, itx);
1304         }

1306         itx->itx_lr.lrc_txg = dmu_tx_get_txg(tx);
1307         zilog_dirty(zilog, txg);
1308         mutex_exit(&itxg->itxg_lock);

1310         /* Release the old itxs now we've dropped the lock */
1311         if (clean != NULL)
1312                 zil_itxg_clean(clean);
1313 }

1315 /*
1316  * If there are any in-memory intent log transactions which have now been
1317  * synced then start up a taskq to free them. We should only do this after we
1318  * have written out the uberblocks (i.e. txg has been comitted) so that
1319  * don't inadvertently clean out in-memory log records that would be required
1320  * by zil_commit().
1321  */
1322 void
1323 zil_clean(zilog_t *zilog, uint64_t synced_txg)
1324 {
1325         itxg_t *itxg = &zilog->zl_itxg[synced_txg & TXG_MASK];
1326         itxs_t *clean_me;

1328         mutex_enter(&itxg->itxg_lock);
1329         if (itxg->itxg_itxs == NULL || itxg->itxg_txg == ZILTEST_TXG) {
1330                 mutex_exit(&itxg->itxg_lock);
1331                 return;
1332         }
1333         ASSERT3U(itxg->itxg_txg, <=, synced_txg);
1334         ASSERT(itxg->itxg_txg != 0);
1335         ASSERT(zilog->zl_clean_taskq != NULL);
1336         atomic_add_64(&zilog->zl_itx_list_sz, -itxg->itxg_sod);
1337         itxg->itxg_sod = 0;
1338         clean_me = itxg->itxg_itxs;
1339         itxg->itxg_itxs = NULL;
1340         itxg->itxg_txg = 0;
1341         mutex_exit(&itxg->itxg_lock);
1342         /*
1343          * Preferably start a task queue to free up the old itxs but
1344          * if taskq_dispatch can't allocate resources to do that then
1345          * free it in-line. This should be rare. Note, using TQ_SLEEP
```



```
1346          * created a bad performance problem.
1347          */
1348         if (taskq_dispatch(zilog->zl_clean_taskq,
1349             (void (*)(void *))zil_itxg_clean, clean_me, TQ_NOSLEEP) == NULL)
1350                 zil_itxg_clean(clean_me);
1351 }

1353 /*
1354  * Get the list of itxs to commit into zl_itx_commit_list.
1355  */
1356 static void
1357 zil_get_commit_list(zilog_t *zilog)
1358 {
1359         uint64_t otxg, txg;
1360         list_t *commit_list = &zilog->zl_itx_commit_list;
1361         uint64_t push_sod = 0;

1363         if (spa_freeze_txg(zilog->zl_spa) != UINT64_MAX) /* ziltest support */
1364                 otxg = ZILTEST_TXG;
1365         else
1366                 otxg = spa_last_synced_txg(zilog->zl_spa) + 1;

1368         for (txg = otxg; txg < (otxg + TXG_CONCURRENT_STATES); txg++) {
1369                 itxg_t *itxg = &zilog->zl_itxg[txg & TXG_MASK];

1371                 mutex_enter(&itxg->itxg_lock);
1372                 if (itxg->itxg_txg != txg) {
1373                         mutex_exit(&itxg->itxg_lock);
1374                         continue;
1375                 }

1377                 list_move_tail(commit_list, &itxg->itxg_itxs->i_sync_list);
1378                 push_sod += itxg->itxg_sod;
1379                 itxg->itxg_sod = 0;

1381                 mutex_exit(&itxg->itxg_lock);
1382         }
1383         atomic_add_64(&zilog->zl_itx_list_sz, -push_sod);
1384 }

1386 /*
1387  * Move the async itxs for a specified object to commit into sync lists.
1388  */
1389 static void
1390 zil_async_to_sync(zilog_t *zilog, uint64_t foid)
1391 {
1392         uint64_t otxg, txg;
1393         itx_async_node_t *ian;
1394         avl_tree_t *t;
1395         avl_index_t where;

1397         if (spa_freeze_txg(zilog->zl_spa) != UINT64_MAX) /* ziltest support */
1398                 otxg = ZILTEST_TXG;
1399         else
1400                 otxg = spa_last_synced_txg(zilog->zl_spa) + 1;

1402         for (txg = otxg; txg < (otxg + TXG_CONCURRENT_STATES); txg++) {
1403                 itxg_t *itxg = &zilog->zl_itxg[txg & TXG_MASK];

1405                 mutex_enter(&itxg->itxg_lock);
1406                 if (itxg->itxg_txg != txg) {
1407                         mutex_exit(&itxg->itxg_lock);
1408                         continue;
1409                 }

1411                 /*
```

```
1412                 * If a foid is specified then find that node and append its
1413                 * list. Otherwise walk the tree appending all the lists
1414                 * to the sync list. We add to the end rather than the
1415                 * beginning to ensure the create has happened.
1416                 */
1417                t = &itxg->itxg_itxs->i_async_tree;
1418                if (foid != 0) {
1419                        ian = avl_find(t, &foid, &where);
1420                        if (ian != NULL) {
1421                                list_move_tail(&itxg->itxg_itxs->i_sync_list,
1422                                    &ian->ia_list);
1423                        }
1424                } else {
1425                        void *cookie = NULL;

1427                        while ((ian = avl_destroy_nodes(t, &cookie)) != NULL) {
1428                                list_move_tail(&itxg->itxg_itxs->i_sync_list,
1429                                    &ian->ia_list);
1430                                list_destroy(&ian->ia_list);
1431                                kmem_free(ian, sizeof (itx_async_node_t));
1432                        }
1433                }
1434                mutex_exit(&itxg->itxg_lock);
1435        }
1436 }

1438 static void
1439 zil_commit_writer(zilog_t *zilog)
1440 {
1441        uint64_t txg;
1442        itx_t *itx;
1443        lwb_t *lwb;
1444        spa_t *spa = zilog->zl_spa;
1445        int error = 0;

1447        ASSERT(zilog->zl_root_zio == NULL);

1449        mutex_exit(&zilog->zl_lock);

1451        zil_get_commit_list(zilog);

1453        /*
1454         * Return if there's nothing to commit before we dirty the fs by
1455         * calling zil_create().
1456         */
1457        if (list_head(&zilog->zl_itx_commit_list) == NULL) {
1458                mutex_enter(&zilog->zl_lock);
1459                return;
1460        }

1462        if (zilog->zl_suspend) {
1463                lwb = NULL;
1464        } else {
1465                lwb = list_tail(&zilog->zl_lwb_list);
1466                if (lwb == NULL)
1467                        lwb = zil_create(zilog);
1468        }

1470        DTRACE_PROBE1(zil__cw1, zilog_t *, zilog);
1471        while (itx = list_head(&zilog->zl_itx_commit_list)) {
1472                txg = itx->itx_lr.lrc_txg;
1473                ASSERT(txg);

1475                if (txg > spa_last_synced_txg(spa) || txg > spa_freeze_txg(spa))
1476                        lwb = zil_lwb_commit(zilog, itx, lwb);
1477                list_remove(&zilog->zl_itx_commit_list, itx);
```



```
1478                kmem_free(itx, offsetof(itx_t, itx_lr)
1479                    + itx->itx_lr.lrc_reclen);
1480        }
1481        DTRACE_PROBE1(zil__cw2, zilog_t *, zilog);

1483        /* write the last block out */
1484        if (lwb != NULL && lwb->lwb_zio != NULL)
1485                lwb = zil_lwb_write_start(zilog, lwb);

1487        zilog->zl_cur_used = 0;

1489        /*
1490         * Wait if necessary for the log blocks to be on stable storage.
1491         */
1492        if (zilog->zl_root_zio) {
1493                error = zio_wait(zilog->zl_root_zio);
1494                zilog->zl_root_zio = NULL;
1495                zil_flush_vdevs(zilog);
1496        }

1498        if (error || lwb == NULL)
1499                txg_wait_synced(zilog->zl_dmu_pool, 0);

1501        mutex_enter(&zilog->zl_lock);

1503        /*
1504         * Remember the highest committed log sequence number for ztest.
1505         * We only update this value when all the log writes succeeded,
1506         * because ztest wants to ASSERT that it got the whole log chain.
1507         */
1508        if (error == 0 && lwb != NULL)
1509                zilog->zl_commit_lr_seq = zilog->zl_lr_seq;
1510 }

1512 /*
1513  * Commit zfs transactions to stable storage.
1514  * If foid is 0 push out all transactions, otherwise push only those
1515  * for that object or might reference that object.
1516  *
1517  * itxs are committed in batches. In a heavily stressed zil there will be
1518  * a commit writer thread who is writing out a bunch of itxs to the log
1519  * for a set of committing threads (cthreads) in the same batch as the writer.
1520  * Those cthreads are all waiting on the same cv for that batch.
1521  *
1522  * There will also be a different and growing batch of threads that are
1523  * waiting to commit (qthreads). When the committing batch completes
1524  * a transition occurs such that the cthreads exit and the qthreads become
1525  * cthreads. One of the new cthreads becomes the writer thread for the
1526  * batch. Any new threads arriving become new qthreads.
1527  *
1528  * Only 2 condition variables are needed and there's no transition
1529  * between the two cvs needed. They just flip-flop between qthreads
1530  * and cthreads.
1531  *
1532  * Using this scheme we can efficiently wakeup up only those threads
1533  * that have been committed.
1534  */
1535 void
1536 zil_commit(zilog_t *zilog, uint64_t foid)
1537 {
1538        uint64_t mybatch;

1540        if (zilog->zl_sync == ZFS_SYNC_DISABLED)
1541                return;

1543        /* move the async itxs for the foid to the sync queues */
```

```
1544         zil_async_to_sync(zilog, foid);

1546         mutex_enter(&zilog->zl_lock);
1547         mybatch = zilog->zl_next_batch;
1548         while (zilog->zl_writer) {
1549                 cv_wait(&zilog->zl_cv_batch[mybatch & 1], &zilog->zl_lock);
1550                 if (mybatch <= zilog->zl_com_batch) {
1551                         mutex_exit(&zilog->zl_lock);
1552                         return;
1553                 }
1554         }

1556         zilog->zl_next_batch++;
1557         zilog->zl_writer = B_TRUE;
1558         zil_commit_writer(zilog);
1559         zilog->zl_com_batch = mybatch;
1560         zilog->zl_writer = B_FALSE;
1561         mutex_exit(&zilog->zl_lock);

1563         /* wake up one thread to become the next writer */
1564         cv_signal(&zilog->zl_cv_batch[(mybatch+1) & 1]);

1566         /* wake up all threads waiting for this batch to be committed */
1567         cv_broadcast(&zilog->zl_cv_batch[mybatch & 1]);
1568 }

1570 /*
1571  * Called in syncing context to free committed log blocks and update log header.
1572  */
1573 void
1574 zil_sync(zilog_t *zilog, dmu_tx_t *tx)
1575 {
1576         zil_header_t *zh = zil_header_in_syncing_context(zilog);
1577         uint64_t txg = dmu_tx_get_txg(tx);
1578         spa_t *spa = zilog->zl_spa;
1579         uint64_t *replayed_seq = &zilog->zl_replayed_seq[txg & TXG_MASK];
1580         lwb_t *lwb;

1582         /*
1583          * We don't zero out zl_destroy_txg, so make sure we don't try
1584          * to destroy it twice.
1585          */
1586         if (spa_sync_pass(spa) != 1)
1587                 return;

1589         mutex_enter(&zilog->zl_lock);

1591         ASSERT(zilog->zl_stop_sync == 0);

1593         if (*replayed_seq != 0) {
1594                 ASSERT(zh->zh_replay_seq < *replayed_seq);
1595                 zh->zh_replay_seq = *replayed_seq;
1596                 *replayed_seq = 0;
1597         }

1599         if (zilog->zl_destroy_txg == txg) {
1600                 blkptr_t blk = zh->zh_log;

1602                 ASSERT(list_head(&zilog->zl_lwb_list) == NULL);

1604                 bzero(zh, sizeof (zil_header_t));
1605                 bzero(zilog->zl_replayed_seq, sizeof (zilog->zl_replayed_seq));

1607                 if (zilog->zl_keep_first) {
1608                         /*
1609                          * If this block was part of log chain that couldn't
```



```
1610                          * be claimed because a device was missing during
1611                          * zil_claim(), but that device later returns,
1612                          * then this block could erroneously appear valid.
1613                          * To guard against this, assign a new GUID to the new
1614                          * log chain so it doesn't matter what blk points to.
1615                          */
1616                         zil_init_log_chain(zilog, &blk);
1617                         zh->zh_log = blk;
1618                 }
1619         }

1621         while ((lwb = list_head(&zilog->zl_lwb_list)) != NULL) {
1622                 zh->zh_log = lwb->lwb_blk;
1623                 if (lwb->lwb_buf != NULL || lwb->lwb_max_txg > txg)
1624                         break;
1625                 list_remove(&zilog->zl_lwb_list, lwb);
1626                 zio_free_zil(spa, txg, &lwb->lwb_blk);
1627                 kmem_cache_free(zil_lwb_cache, lwb);

1629                 /*
1630                  * If we don't have anything left in the lwb list then
1631                  * we've had an allocation failure and we need to zero
1632                  * out the zil_header blkptr so that we don't end
1633                  * up freeing the same block twice.
1634                  */
1635                 if (list_head(&zilog->zl_lwb_list) == NULL)
1636                         BP_ZERO(&zh->zh_log);
1637         }
1638         mutex_exit(&zilog->zl_lock);
1639 }

1641 void
1642 zil_init(void)
1643 {
1644         zil_lwb_cache = kmem_cache_create("zil_lwb_cache",
1645             sizeof (struct lwb), 0, NULL, NULL, NULL, NULL, NULL, 0);
1646 }

1648 void
1649 zil_fini(void)
1650 {
1651         kmem_cache_destroy(zil_lwb_cache);
1652 }

1654 void
1655 zil_set_sync(zilog_t *zilog, uint64_t sync)
1656 {
1657         zilog->zl_sync = sync;
1658 }

1660 void
1661 zil_set_logbias(zilog_t *zilog, uint64_t logbias)
1662 {
1663         zilog->zl_logbias = logbias;
1664 }

1666 zilog_t *
1667 zil_alloc(objset_t *os, zil_header_t *zh_phys)
1668 {
1669         zilog_t *zilog;

1671         zilog = kmem_zalloc(sizeof (zilog_t), KM_SLEEP);

1673         zilog->zl_header = zh_phys;
1674         zilog->zl_os = os;
1675         zilog->zl_spa = dmu_objset_spa(os);
```

```
1676         zilog->zl_dmu_pool = dmu_objset_pool(os);
1677         zilog->zl_destroy_txg = TXG_INITIAL - 1;
1678         zilog->zl_logbias = dmu_objset_logbias(os);
1679         zilog->zl_sync = dmu_objset_syncprop(os);
1680         zilog->zl_next_batch = 1;

1682         mutex_init(&zilog->zl_lock, NULL, MUTEX_DEFAULT, NULL);

1684         for (int i = 0; i < TXG_SIZE; i++) {
1685                 mutex_init(&zilog->zl_itxg[i].itxg_lock, NULL,
1686                     MUTEX_DEFAULT, NULL);
1687         }

1689         list_create(&zilog->zl_lwb_list, sizeof (lwb_t),
1690             offsetof(lwb_t, lwb_node));

1692         list_create(&zilog->zl_itx_commit_list, sizeof (itx_t),
1693             offsetof(itx_t, itx_node));

1695         mutex_init(&zilog->zl_vdev_lock, NULL, MUTEX_DEFAULT, NULL);

1697         avl_create(&zilog->zl_vdev_tree, zil_vdev_compare,
1698             sizeof (zil_vdev_node_t), offsetof(zil_vdev_node_t, zv_node));

1700         cv_init(&zilog->zl_cv_writer, NULL, CV_DEFAULT, NULL);
1701         cv_init(&zilog->zl_cv_suspend, NULL, CV_DEFAULT, NULL);
1702         cv_init(&zilog->zl_cv_batch[0], NULL, CV_DEFAULT, NULL);
1703         cv_init(&zilog->zl_cv_batch[1], NULL, CV_DEFAULT, NULL);

1705         return (zilog);
1706 }

1708 void
1709 zil_free(zilog_t *zilog)
1710 {
1711         zilog->zl_stop_sync = 1;

1713         ASSERT0(zilog->zl_suspend);
1714         ASSERT0(zilog->zl_suspending);

1716         ASSERT(list_is_empty(&zilog->zl_lwb_list));
1717         list_destroy(&zilog->zl_lwb_list);

1719         avl_destroy(&zilog->zl_vdev_tree);
1720         mutex_destroy(&zilog->zl_vdev_lock);

1722         ASSERT(list_is_empty(&zilog->zl_itx_commit_list));
1723         list_destroy(&zilog->zl_itx_commit_list);

1725         for (int i = 0; i < TXG_SIZE; i++) {
1726                 /*
1727                  * It's possible for an itx to be generated that doesn't dirty
1728                  * a txg (e.g. ztest TX_TRUNCATE). So there's no zil_clean()
1729                  * callback to remove the entry. We remove those here.
1730                  *
1731                  * Also free up the ziltest itxs.
1732                  */
1733                 if (zilog->zl_itxg[i].itxg_itxs)
1734                         zil_itxg_clean(zilog->zl_itxg[i].itxg_itxs);
1735                 mutex_destroy(&zilog->zl_itxg[i].itxg_lock);
1736         }

1738         mutex_destroy(&zilog->zl_lock);

1740         cv_destroy(&zilog->zl_cv_writer);
1741         cv_destroy(&zilog->zl_cv_suspend);
```



```
1742         cv_destroy(&zilog->zl_cv_batch[0]);
1743         cv_destroy(&zilog->zl_cv_batch[1]);

1745         kmem_free(zilog, sizeof (zilog_t));
1746 }

1748 /*
1749  * Open an intent log.
1750  */
1751 zilog_t *
1752 zil_open(objset_t *os, zil_get_data_t *get_data)
1753 {
1754         zilog_t *zilog = dmu_objset_zil(os);

1756         ASSERT(zilog->zl_clean_taskq == NULL);
1757         ASSERT(zilog->zl_get_data == NULL);
1758         ASSERT(list_is_empty(&zilog->zl_lwb_list));

1760         zilog->zl_get_data = get_data;
1761         zilog->zl_clean_taskq = taskq_create("zil_clean", 1, minclsyspri,
1762             2, 2, TASKQ_PREPOPULATE);

1764         return (zilog);
1765 }

1767 /*
1768  * Close an intent log.
1769  */
1770 void
1771 zil_close(zilog_t *zilog)
1772 {
1773         lwb_t *lwb;
1774         uint64_t txg = 0;

1776         zil_commit(zilog, 0); /* commit all itx */

1778         /*
1779          * The lwb_max_txg for the stubby lwb will reflect the last activity
1780          * for the zil.  After a txg_wait_synced() on the txg we know all the
1781          * callbacks have occurred that may clean the zil.  Only then can we
1782          * destroy the zl_clean_taskq.
1783          */
1784         mutex_enter(&zilog->zl_lock);
1785         lwb = list_tail(&zilog->zl_lwb_list);
1786         if (lwb != NULL)
1787                 txg = lwb->lwb_max_txg;
1788         mutex_exit(&zilog->zl_lock);
1789         if (txg)
1790                 txg_wait_synced(zilog->zl_dmu_pool, txg);
1791         ASSERT(!zilog_is_dirty(zilog));

1793         taskq_destroy(zilog->zl_clean_taskq);
1794         zilog->zl_clean_taskq = NULL;
1795         zilog->zl_get_data = NULL;

1797         /*
1798          * We should have only one LWB left on the list; remove it now.
1799          */
1800         mutex_enter(&zilog->zl_lock);
1801         lwb = list_head(&zilog->zl_lwb_list);
1802         if (lwb != NULL) {
1803                 ASSERT(lwb == list_tail(&zilog->zl_lwb_list));
1804                 list_remove(&zilog->zl_lwb_list, lwb);
1805                 zio_buf_free(lwb->lwb_buf, lwb->lwb_sz);
1806                 kmem_cache_free(zil_lwb_cache, lwb);
1807         }
```

```
1808         mutex_exit(&zilog->zl_lock);
1809 }

1811 static char *suspend_tag = "zil suspending";

1813 /*
1814  * Suspend an intent log.  While in suspended mode, we still honor
1815  * synchronous semantics, but we rely on txg_wait_synced() to do it.
1816  * On old version pools, we suspend the log briefly when taking a
1817  * snapshot so that it will have an empty intent log.
1818  *
1819  * Long holds are not really intended to be used the way we do here --
1820  * held for such a short time.  A concurrent caller of dsl_dataset_long_held()
1821  * could fail.  Therefore we take pains to only put a long hold if it is
1822  * actually necessary.  Fortunately, it will only be necessary if the
1823  * objset is currently mounted (or the ZVOL equivalent).  In that case it
1824  * will already have a long hold, so we are not really making things any worse.
1825  *
1826  * Ideally, we would locate the existing long-holder (i.e. the zfsvfs_t or
1827  * zvol_state_t), and use their mechanism to prevent their hold from being
1828  * dropped (e.g. VFS_HOLD()).  However, that would be even more pain for
1829  * very little gain.
1830  *
1831  * if cookiep == NULL, this does both the suspend & resume.
1832  * Otherwise, it returns with the dataset "long held", and the cookie
1833  * should be passed into zil_resume().
1834  */
1835 int
1836 zil_suspend(const char *osname, void **cookiep)
1837 {
1838         objset_t *os;
1839         zilog_t *zilog;
1840         const zil_header_t *zh;
1841         int error;

1843         error = dmu_objset_hold(osname, suspend_tag, &os);
1844         if (error != 0)
1845                 return (error);
1846         zilog = dmu_objset_zil(os);

1848         mutex_enter(&zilog->zl_lock);
1849         zh = zilog->zl_header;

1851         if (zh->zh_flags & ZIL_REPLAY_NEEDED) {         /* unplayed log */
1852                 mutex_exit(&zilog->zl_lock);
1853                 dmu_objset_rele(os, suspend_tag);
1854                 return (SET_ERROR(EBUSY));
1855         }

1857         /*
1858          * Don't put a long hold in the cases where we can avoid it.  This
1859          * is when there is no cookie so we are doing a suspend & resume
1860          * (i.e. called from zil_vdev_offline()), and there's nothing to do
1861          * for the suspend because it's already suspended, or there's no ZIL.
1862          */
1863         if (cookiep == NULL && !zilog->zl_suspending &&
1864             (zilog->zl_suspend > 0 || BP_IS_HOLE(&zh->zh_log))) {
1865                 mutex_exit(&zilog->zl_lock);
1866                 dmu_objset_rele(os, suspend_tag);
1867                 return (0);
1868         }

1870         dsl_dataset_long_hold(dmu_objset_ds(os), suspend_tag);
1871         dsl_pool_rele(dmu_objset_pool(os), suspend_tag);

1873         zilog->zl_suspend++;
```



```
1875         if (zilog->zl_suspend > 1) {
1876                 /*
1877                  * Someone else is already suspending it.
1878                  * Just wait for them to finish.
1879                  */

1881                 while (zilog->zl_suspending)
1882                         cv_wait(&zilog->zl_cv_suspend, &zilog->zl_lock);
1883                 mutex_exit(&zilog->zl_lock);

1885                 if (cookiep == NULL)
1886                         zil_resume(os);
1887                 else
1888                         *cookiep = os;
1889                 return (0);
1890         }

1892         /*
1893          * If there is no pointer to an on-disk block, this ZIL must not
1894          * be active (e.g. filesystem not mounted), so there's nothing
1895          * to clean up.
1896          */
1897         if (BP_IS_HOLE(&zh->zh_log)) {
1898                 ASSERT(cookiep != NULL); /* fast path already handled */

1900                 *cookiep = os;
1901                 mutex_exit(&zilog->zl_lock);
1902                 return (0);
1903         }

1905         zilog->zl_suspending = B_TRUE;
1906         mutex_exit(&zilog->zl_lock);

1908         zil_commit(zilog, 0);

1910         zil_destroy(zilog, B_FALSE);

1912         mutex_enter(&zilog->zl_lock);
1913         zilog->zl_suspending = B_FALSE;
1914         cv_broadcast(&zilog->zl_cv_suspend);
1915         mutex_exit(&zilog->zl_lock);

1917         if (cookiep == NULL)
1918                 zil_resume(os);
1919         else
1920                 *cookiep = os;
1921         return (0);
1922 }

1924 void
1925 zil_resume(void *cookie)
1926 {
1927         objset_t *os = cookie;
1928         zilog_t *zilog = dmu_objset_zil(os);

1930         mutex_enter(&zilog->zl_lock);
1931         ASSERT(zilog->zl_suspend != 0);
1932         zilog->zl_suspend--;
1933         mutex_exit(&zilog->zl_lock);
1934         dsl_dataset_long_rele(dmu_objset_ds(os), suspend_tag);
1935         dsl_dataset_rele(dmu_objset_ds(os), suspend_tag);
1936 }

1938 typedef struct zil_replay_arg {
1939         zil_replay_func_t **zr_replay;
```

```
1940         void            *zr_arg;
1941         boolean_t       zr_byteswap;
1942         char            *zr_lr;
1943 } zil_replay_arg_t;

1945 static int
1946 zil_replay_error(zilog_t *zilog, lr_t *lr, int error)
1947 {
1948         char name[MAXNAMELEN];

1950         zilog->zl_replaying_seq--;      /* didn't actually replay this one */

1952         dmu_objset_name(zilog->zl_os, name);

1954         cmn_err(CE_WARN, "ZFS replay transaction error %d, "
1955             "dataset %s, seq 0x%llx, txtype %llu %s\n", error, name,
1956             (u_longlong_t)lr->lrc_seq,
1957             (u_longlong_t)(lr->lrc_txtype & ~TX_CI),
1958             (lr->lrc_txtype & TX_CI) ? "CI" : "");

1960         return (error);
1961 }

1963 static int
1964 zil_replay_log_record(zilog_t *zilog, lr_t *lr, void *zra, uint64_t claim_txg)
1965 {
1966         zil_replay_arg_t *zr = zra;
1967         const zil_header_t *zh = zilog->zl_header;
1968         uint64_t reclen = lr->lrc_reclen;
1969         uint64_t txtype = lr->lrc_txtype;
1970         int error = 0;

1972         zilog->zl_replaying_seq = lr->lrc_seq;

1974         if (lr->lrc_seq <= zh->zh_replay_seq)   /* already replayed */
1975                 return (0);

1977         if (lr->lrc_txg < claim_txg)            /* already committed */
1978                 return (0);

1980         /* Strip case-insensitive bit, still present in log record */
1981         txtype &= ~TX_CI;

1983         if (txtype == 0 || txtype >= TX_MAX_TYPE)
1984                 return (zil_replay_error(zilog, lr, EINVAL));

1986         /*
1987          * If this record type can be logged out of order, the object
1988          * (lr_foid) may no longer exist.  That's legitimate, not an error.
1989          */
1990         if (TX_OOO(txtype)) {
1991                 error = dmu_object_info(zilog->zl_os,
1992                     ((lr_ooo_t *)lr)->lr_foid, NULL);
1993                 if (error == ENOENT || error == EEXIST)
1994                         return (0);
1995         }

1997         /*
1998          * Make a copy of the data so we can revise and extend it.
1999          */
2000         bcopy(lr, zr->zr_lr, reclen);

2002         /*
2003          * If this is a TX_WRITE with a blkptr, suck in the data.
2004          */
2005         if (txtype == TX_WRITE && reclen == sizeof (lr_write_t)) {
```



```
2006                 error = zil_read_log_data(zilog, (lr_write_t *)lr,
2007                     zr->zr_lr + reclen);
2008                 if (error != 0)
2009                         return (zil_replay_error(zilog, lr, error));
2010         }

2012         /*
2013          * The log block containing this lr may have been byteswapped
2014          * so that we can easily examine common fields like lrc_txtype.
2015          * However, the log is a mix of different record types, and only the
2016          * replay vectors know how to byteswap their records.  Therefore, if
2017          * the lr was byteswapped, undo it before invoking the replay vector.
2018          */
2019         if (zr->zr_byteswap)
2020                 byteswap_uint64_array(zr->zr_lr, reclen);

2022         /*
2023          * We must now do two things atomically: replay this log record,
2024          * and update the log header sequence number to reflect the fact that
2025          * we did so. At the end of each replay function the sequence number
2026          * is updated if we are in replay mode.
2027          */
2028         error = zr->zr_replay[txtype](zr->zr_arg, zr->zr_lr, zr->zr_byteswap);
2029         if (error != 0) {
2030                 /*
2031                  * The DMU's dnode layer doesn't see removes until the txg
2032                  * commits, so a subsequent claim can spuriously fail with
2033                  * EEXIST. So if we receive any error we try syncing out
2034                  * any removes then retry the transaction.  Note that we
2035                  * specify B_FALSE for byteswap now, so we don't do it twice.
2036                  */
2037                 txg_wait_synced(spa_get_dsl(zilog->zl_spa), 0);
2038                 error = zr->zr_replay[txtype](zr->zr_arg, zr->zr_lr, B_FALSE);
2039                 if (error != 0)
2040                         return (zil_replay_error(zilog, lr, error));
2041         }
2042         return (0);
2043 }

2045 /* ARGSUSED */
2046 static int
2047 zil_incr_blks(zilog_t *zilog, blkptr_t *bp, void *arg, uint64_t claim_txg)
2048 {
2049         zilog->zl_replay_blks++;

2051         return (0);
2052 }

2054 /*
2055  * If this dataset has a non-empty intent log, replay it and destroy it.
2056  */
2057 void
2058 zil_replay(objset_t *os, void *arg, zil_replay_func_t *replay_func[TX_MAX_TYPE])
2059 {
2060         zilog_t *zilog = dmu_objset_zil(os);
2061         const zil_header_t *zh = zilog->zl_header;
2062         zil_replay_arg_t zr;

2064         if ((zh->zh_flags & ZIL_REPLAY_NEEDED) == 0) {
2065                 zil_destroy(zilog, B_TRUE);
2066                 return;
2067         }

2069         zr.zr_replay = replay_func;
2070         zr.zr_arg = arg;
2071         zr.zr_byteswap = BP_SHOULD_BYTESWAP(&zh->zh_log);
```

```
2072         zr.zr_lr = kmem_alloc(2 * SPA_MAXBLOCKSIZE, KM_SLEEP);

2074         /*
2075          * Wait for in-progress removes to sync before starting replay.
2076          */
2077         txg_wait_synced(zilog->zl_dmu_pool, 0);

2079         zilog->zl_replay = B_TRUE;
2080         zilog->zl_replay_time = ddi_get_lbolt();
2081         ASSERT(zilog->zl_replay_blks == 0);
2082         (void) zil_parse(zilog, zil_incr_blks, zil_replay_log_record, &zr,
2083             zh->zh_claim_txg);
2084         kmem_free(zr.zr_lr, 2 * SPA_MAXBLOCKSIZE);

2086         zil_destroy(zilog, B_FALSE);
2087         txg_wait_synced(zilog->zl_dmu_pool, zilog->zl_destroy_txg);
2088         zilog->zl_replay = B_FALSE;
2089 }

2091 boolean_t
2092 zil_replaying(zilog_t *zilog, dmu_tx_t *tx)
2093 {
2094         if (zilog->zl_sync == ZFS_SYNC_DISABLED)
2095                 return (B_TRUE);

2097         if (zilog->zl_replay) {
2098                 dsl_dataset_dirty(dmu_objset_ds(zilog->zl_os), tx);
2099                 zilog->zl_replayed_seq[dmu_tx_get_txg(tx) & TXG_MASK] =
2100                     zilog->zl_replaying_seq;
2101                 return (B_TRUE);
2102         }

2104         return (B_FALSE);
2105 }

2107 /* ARGSUSED */
2108 int
2109 zil_vdev_offline(const char *osname, void *arg)
2110 {
2111         int error;

2113         error = zil_suspend(osname, NULL);
2114         if (error != 0)
2115                 return (SET_ERROR(EEXIST));
2116         return (0);
2117 }
```

```
**********************************************************
   89624 Thu May 16 17:36:28 2013
new/usr/src/uts/common/fs/zfs/zio.c
3742 zfs comments need cleaner, more consistent style
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
Reviewed by:    George Wilson <george.wilson@delphix.com>
Reviewed by:    Eric Schrock <eric.schrock@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1185 /*
1186  * Execute the I/O pipeline until one of the following occurs:
1187  *
1188  *      (1) the I/O completes
1189  *      (2) the pipeline stalls waiting for dependent child I/Os
1190  *      (3) the I/O issues, so we're waiting for an I/O completion interrupt
1191  *      (4) the I/O is delegated by vdev-level caching or aggregation
1192  *      (5) the I/O is deferred due to vdev-level queueing
1193  *      (6) the I/O is handed off to another thread.
1194  *
1195  * In all cases, the pipeline stops whenever there's no CPU work; it never
1196  * burns a thread in cv_wait().
1187  * (1) the I/O completes; (2) the pipeline stalls waiting for
1188  * dependent child I/Os; (3) the I/O issues, so we're waiting
1189  * for an I/O completion interrupt; (4) the I/O is delegated by
1190  * vdev-level caching or aggregation; (5) the I/O is deferred
1191  * due to vdev-level queueing; (6) the I/O is handed off to
1192  * another thread.  In all cases, the pipeline stops whenever
1193  * there's no CPU work; it never burns a thread in cv_wait().
1197  *
1198  * There's no locking on io_stage because there's no legitimate way
1199  * for multiple threads to be attempting to process the same I/O.
1200  */
1201 static zio_pipe_stage_t *zio_pipeline[];

1203 void
1204 zio_execute(zio_t *zio)
1205 {
1206         zio->io_executor = curthread;

1208         while (zio->io_stage < ZIO_STAGE_DONE) {
1209                 enum zio_stage pipeline = zio->io_pipeline;
1210                 enum zio_stage stage = zio->io_stage;
1211                 int rv;

1213                 ASSERT(!MUTEX_HELD(&zio->io_lock));
1214                 ASSERT(ISP2(stage));
1215                 ASSERT(zio->io_stall == NULL);

1217                 do {
1218                         stage <<= 1;
1219                 } while ((stage & pipeline) == 0);

1221                 ASSERT(stage <= ZIO_STAGE_DONE);

1223                 /*
1224                  * If we are in interrupt context and this pipeline stage
1225                  * will grab a config lock that is held across I/O,
1226                  * or may wait for an I/O that needs an interrupt thread
1227                  * to complete, issue async to avoid deadlock.
1228                  *
1229                  * For VDEV_IO_START, we cut in line so that the io will
1230                  * be sent to disk promptly.
1231                  */
```



```
1232                 if ((stage & ZIO_BLOCKING_STAGES) && zio->io_vd == NULL &&
1233                     zio_taskq_member(zio, ZIO_TASKQ_INTERRUPT)) {
1234                         boolean_t cut = (stage == ZIO_STAGE_VDEV_IO_START) ?
1235                             zio_requeue_io_start_cut_in_line : B_FALSE;
1236                         zio_taskq_dispatch(zio, ZIO_TASKQ_ISSUE, cut);
1237                         return;
1238                 }

1240                 zio->io_stage = stage;
1241                 rv = zio_pipeline[highbit(stage) - 1](zio);

1243                 if (rv == ZIO_PIPELINE_STOP)
1244                         return;

1246                 ASSERT(rv == ZIO_PIPELINE_CONTINUE);
1247         }
1248 }
______unchanged_portion_omitted_
```