```
**********************************************************
   79874 Tue Jan  6 10:37:46 2015
new/usr/src/uts/common/fs/zfs/dbuf.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2012, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/dmu.h>
#include <sys/dmu_send.h>
#include <sys/dmu_impl.h>
#include <sys/dbuf.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dmu_tx.h>
#include <sys/spa.h>
#include <sys/zio.h>
#include <sys/dmu_zfetch.h>
#include <sys/sa.h>
#include <sys/sa_impl.h>
#include <sys/zfeature.h>
#include <sys/blkptr.h>
#include <sys/range_tree.h>

/*
 * Number of times that zfs_free_range() took the slow path while doing
 * a zfs receive.  A nonzero value indicates a potential performance problem.
 */
uint64_t zfs_free_range_recv_miss;

static void dbuf_destroy(dmu_buf_impl_t *db);
static boolean_t dbuf_undirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
static void dbuf_write(dbuf_dirty_record_t *dr, arc_buf_t *data, dmu_tx_t *tx);
```



```
#ifndef __lint
extern inline void dmu_buf_init_user(dmu_buf_user_t *dbu,
    dmu_buf_evict_func_t *evict_func, dmu_buf_t **clear_on_evict_dbufp);
#endif /* ! __lint */

/*
 * Global data structures and functions for the dbuf cache.
 */
static kmem_cache_t *dbuf_cache;
static taskq_t *dbu_evict_taskq;

/* ARGSUSED */
static int
dbuf_cons(void *vdb, void *unused, int kmflag)
{
        dmu_buf_impl_t *db = vdb;
        bzero(db, sizeof (dmu_buf_impl_t));

        mutex_init(&db->db_mtx, NULL, MUTEX_DEFAULT, NULL);
        cv_init(&db->db_changed, NULL, CV_DEFAULT, NULL);
        refcount_create(&db->db_holds);

        return (0);
}
```

*unchanged_portion_omitted_*

```
static arc_evict_func_t dbuf_do_evict;

typedef enum {
        DBVU_EVICTING,
        DBVU_NOT_EVICTING
} dbvu_verify_type_t;

static void
dbuf_verify_user(dmu_buf_impl_t *db, dbvu_verify_type_t verify_type)
{
#ifdef ZFS_DEBUG
        int64_t holds;

        if (db->db_user == NULL)
                return;

        /* Only data blocks support the attachment of user data. */
        ASSERT(db->db_level == 0);

        /* Clients must resolve a dbuf before attaching user data. */
        ASSERT(db->db.db_data != NULL);
        ASSERT3U(db->db_state, ==, DB_CACHED);

        holds = refcount_count(&db->db_holds);
        if (verify_type == DBVU_EVICTING) {
                /*
                 * Immediate eviction occurs when holds == dirtycnt.
                 * For normal eviction buffers, holds is zero on
                 * eviction, except when dbuf_fix_old_data() calls
                 * dbuf_clear_data().  However, the hold count can grow
                 * during eviction even though db_mtx is held (see
                 * dmu_bonus_hold() for an example), so we can only
                 * test the generic invariant that holds >= dirtycnt.
                 */
                ASSERT3U(holds, >=, db->db_dirtycnt);
        } else {
                if (db->db_immediate_evict == TRUE)
                        ASSERT3U(holds, >=, db->db_dirtycnt);
                else
                        ASSERT3U(holds, >, 0);
```

```
263         }
264 #endif
265 }

267 static void
268 dbuf_evict_user(dmu_buf_impl_t *db)
269 {
270         dmu_buf_user_t *dbu = db->db_user;

272         ASSERT(MUTEX_HELD(&db->db_mtx));

274         if (dbu == NULL)
223         if (db->db_level != 0 || db->db_evict_func == NULL)
275                 return;

277         dbuf_verify_user(db, DBVU_EVICTING);
278         db->db_user = NULL;

280 #ifdef ZFS_DEBUG
281         if (dbu->dbu_clear_on_evict_dbufp != NULL)
282                 *dbu->dbu_clear_on_evict_dbufp = NULL;
283 #endif

285         /*
286          * Invoke the callback from a taskq to avoid lock order reversals
287          * and limit stack depth.
288          */
289         taskq_dispatch_ent(dbu_evict_taskq, dbu->dbu_evict_func, dbu, 0,
290             &dbu->dbu_tqent);
226         db->db_evict_func(&db->db, db->db_user_ptr);
227         db->db_user_ptr = NULL;
228         db->db_evict_func = NULL;
291 }
```

*_____unchanged_portion_omitted_*

```
320 void
321 dbuf_init(void)
322 {
323         uint64_t hsize = 1ULL << 16;
324         dbuf_hash_table_t *h = &dbuf_hash_table;
325         int i;

327         /*
328          * The hash table is big enough to fill all of physical memory
329          * with an average 4K block size.  The table will take up
330          * totalmem*sizeof(void*)/4K (i.e. 2MB/GB with 8-byte pointers).
331          */
332         while (hsize * 4096 < physmem * PAGESIZE)
333                 hsize <<= 1;

335 retry:
336         h->hash_table_mask = hsize - 1;
337         h->hash_table = kmem_zalloc(hsize * sizeof (void *), KM_NOSLEEP);
338         if (h->hash_table == NULL) {
339                 /* XXX - we should really return an error instead of assert */
340                 ASSERT(hsize > (1ULL << 10));
341                 hsize >>= 1;
342                 goto retry;
343         }

345         dbuf_cache = kmem_cache_create("dmu_buf_impl_t",
346             sizeof (dmu_buf_impl_t),
347             0, dbuf_cons, dbuf_dest, NULL, NULL, NULL, 0);

349         for (i = 0; i < DBUF_MUTEXES; i++)
350                 mutex_init(&h->hash_mutexes[i], NULL, MUTEX_DEFAULT, NULL);
```



```
352         /*
353          * All entries are queued via taskq_dispatch_ent(), so min/maxalloc
354          * configuration is not required.
355          */
356         dbu_evict_taskq = taskq_create("dbu_evict", 1, minclsyspri, 0, 0, 0);
357 }

359 void
360 dbuf_fini(void)
361 {
362         dbuf_hash_table_t *h = &dbuf_hash_table;
363         int i;

365         for (i = 0; i < DBUF_MUTEXES; i++)
366                 mutex_destroy(&h->hash_mutexes[i]);
367         kmem_free(h->hash_table, (h->hash_table_mask + 1) * sizeof (void *));
368         kmem_cache_destroy(dbuf_cache);
369         taskq_destroy(dbu_evict_taskq);
370 }
```

*_____unchanged_portion_omitted_*

```
484 #endif

486 static void
487 dbuf_clear_data(dmu_buf_impl_t *db)
488 {
489         ASSERT(MUTEX_HELD(&db->db_mtx));
490         dbuf_evict_user(db);
491         db->db_buf = NULL;
492         db->db.db_data = NULL;
493         if (db->db_state != DB_NOFILL)
494                 db->db_state = DB_UNCACHED;
495 }

497 static void
498 dbuf_set_data(dmu_buf_impl_t *db, arc_buf_t *buf)
499 {
500         ASSERT(MUTEX_HELD(&db->db_mtx));
501         ASSERT(buf != NULL);

503         db->db_buf = buf;
422         if (buf != NULL) {
504         ASSERT(buf->b_data != NULL);
505         db->db.db_data = buf->b_data;
506         if (!arc_released(buf))
507                 arc_set_callback(buf, dbuf_do_evict, db);
427         } else {
428                 dbuf_evict_user(db);
429                 db->db.db_data = NULL;
430                 if (db->db_state != DB_NOFILL)
431                         db->db_state = DB_UNCACHED;
432         }
508 }

510 /*
511  * Loan out an arc_buf for read.  Return the loaned arc_buf.
512  */
513 arc_buf_t *
514 dbuf_loan_arcbuf(dmu_buf_impl_t *db)
515 {
516         arc_buf_t *abuf;

518         mutex_enter(&db->db_mtx);
519         if (arc_released(db->db_buf) || refcount_count(&db->db_holds) > 1) {
520                 int blksz = db->db.db_size;
521                 spa_t *spa = db->db_objset->os_spa;
```

```
523                 mutex_exit(&db->db_mtx);
524                 abuf = arc_loan_buf(spa, blksz);
525                 bcopy(db->db.db_data, abuf->b_data, blksz);
526         } else {
527                 abuf = db->db_buf;
528                 arc_loan_inuse_buf(abuf, db);
529                 dbuf_clear_data(db);
454                 dbuf_set_data(db, NULL);
530                 mutex_exit(&db->db_mtx);
531         }
532         return (abuf);
533 }
_____unchanged_portion_omitted_

748 static void
749 dbuf_noread(dmu_buf_impl_t *db)
750 {
751         ASSERT(!refcount_is_zero(&db->db_holds));
752         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
753         mutex_enter(&db->db_mtx);
754         while (db->db_state == DB_READ || db->db_state == DB_FILL)
755                 cv_wait(&db->db_changed, &db->db_mtx);
756         if (db->db_state == DB_UNCACHED) {
757                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
758                 spa_t *spa = db->db_objset->os_spa;

760                 ASSERT(db->db_buf == NULL);
761                 ASSERT(db->db.db_data == NULL);
762                 dbuf_set_data(db, arc_buf_alloc(spa, db->db.db_size, db, type));
763                 db->db_state = DB_FILL;
764         } else if (db->db_state == DB_NOFILL) {
765                 dbuf_clear_data(db);
690                 dbuf_set_data(db, NULL);
766         } else {
767                 ASSERT3U(db->db_state, ==, DB_CACHED);
768         }
769         mutex_exit(&db->db_mtx);
770 }

772 /*
773  * This is our just-in-time copy function.  It makes a copy of
774  * buffers, that have been modified in a previous transaction
775  * group, before we modify them in the current active group.
776  *
777  * This function is used in two places: when we are dirtying a
778  * buffer for the first time in a txg, and when we are freeing
779  * a range in a dnode that includes this buffer.
780  *
781  * Note that when we are called from dbuf_free_range() we do
782  * not put a hold on the buffer, we just traverse the active
783  * dbuf list for the dnode.
784  */
785 static void
786 dbuf_fix_old_data(dmu_buf_impl_t *db, uint64_t txg)
787 {
788         dbuf_dirty_record_t *dr = db->db_last_dirty;

790         ASSERT(MUTEX_HELD(&db->db_mtx));
791         ASSERT(db->db.db_data != NULL);
792         ASSERT(db->db_level == 0);
793         ASSERT(db->db.db_object != DMU_META_DNODE_OBJECT);

795         if (dr == NULL ||
796             (dr->dt.dl.dr_data !=
797             ((db->db_blkid  == DMU_BONUS_BLKID) ? db->db.db_data : db->db_buf)))
```



```
798                 return;

800         /*
801          * If the last dirty record for this dbuf has not yet synced
802          * and its referencing the dbuf data, either:
803          *      reset the reference to point to a new copy,
804          * or (if there a no active holders)
805          *      just null out the current db_data pointer.
806          */
807         ASSERT(dr->dr_txg >= txg - 2);
808         if (db->db_blkid == DMU_BONUS_BLKID) {
809                 /* Note that the data bufs here are zio_bufs */
810                 dr->dt.dl.dr_data = zio_buf_alloc(DN_MAX_BONUSLEN);
811                 arc_space_consume(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
812                 bcopy(db->db.db_data, dr->dt.dl.dr_data, DN_MAX_BONUSLEN);
813         } else if (refcount_count(&db->db_holds) > db->db_dirtycnt) {
814                 int size = db->db.db_size;
815                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
816                 spa_t *spa = db->db_objset->os_spa;

818                 dr->dt.dl.dr_data = arc_buf_alloc(spa, size, db, type);
819                 bcopy(db->db.db_data, dr->dt.dl.dr_data->b_data, size);
820         } else {
821                 dbuf_clear_data(db);
746                 dbuf_set_data(db, NULL);
822         }
823 }
_____unchanged_portion_omitted_

860 /*
861  * Evict (if its unreferenced) or clear (if its referenced) any level-0
862  * data blocks in the free range, so that any future readers will find
863  * empty blocks.
864  *
865  * This is a no-op if the dataset is in the middle of an incremental
866  * receive; see comment below for details.
867  */
868 void
869 dbuf_free_range(dnode_t *dn, uint64_t start_blkid, uint64_t end_blkid,
870     dmu_tx_t *tx)
871 {
872         dmu_buf_impl_t db_search;
873         dmu_buf_impl_t *db, *db_next;
797         dmu_buf_impl_t *db, *db_next, db_search;
874         uint64_t txg = tx->tx_txg;
875         avl_index_t where;

877         if (end_blkid > dn->dn_maxblkid && (end_blkid != DMU_SPILL_BLKID))
878                 end_blkid = dn->dn_maxblkid;
879         dprintf_dnode(dn, "start=%llu end=%llu\n", start_blkid, end_blkid);

881         db_search.db_level = 0;
882         db_search.db_blkid = start_blkid;
883         db_search.db_state = DB_SEARCH;

885         mutex_enter(&dn->dn_dbufs_mtx);
886         if (start_blkid >= dn->dn_unlisted_l0_blkid) {
887                 /* There can't be any dbufs in this range; no need to search. */
888 #ifdef DEBUG
889                 db = avl_find(&dn->dn_dbufs, &db_search, &where);
890                 ASSERT3P(db, ==, NULL);
891                 db = avl_nearest(&dn->dn_dbufs, where, AVL_AFTER);
892                 ASSERT(db == NULL || db->db_level > 0);
893 #endif
894                 mutex_exit(&dn->dn_dbufs_mtx);
895                 return;
```

```
896         } else if (dmu_objset_is_receiving(dn->dn_objset)) {
897                 /*
898                  * If we are receiving, we expect there to be no dbufs in
899                  * the range to be freed, because receive modifies each
900                  * block at most once, and in offset order.  If this is
901                  * not the case, it can lead to performance problems,
902                  * so note that we unexpectedly took the slow path.
903                  */
904                 atomic_inc_64(&zfs_free_range_recv_miss);
905         }

907         db = avl_find(&dn->dn_dbufs, &db_search, &where);
908         ASSERT3P(db, ==, NULL);
909         db = avl_nearest(&dn->dn_dbufs, where, AVL_AFTER);

911         for (; db != NULL; db = db_next) {
912                 db_next = AVL_NEXT(&dn->dn_dbufs, db);
913                 ASSERT(db->db_blkid != DMU_BONUS_BLKID);

915                 if (db->db_level != 0 || db->db_blkid > end_blkid) {
916                         break;
917                 }
918                 ASSERT3U(db->db_blkid, >=, start_blkid);

920                 /* found a level 0 buffer in the range */
921                 mutex_enter(&db->db_mtx);
922                 if (dbuf_undirty(db, tx)) {
923                         /* mutex has been dropped and dbuf destroyed */
924                         continue;
925                 }

927                 if (db->db_state == DB_UNCACHED ||
928                     db->db_state == DB_NOFILL ||
929                     db->db_state == DB_EVICTING) {
930                         ASSERT(db->db.db_data == NULL);
931                         mutex_exit(&db->db_mtx);
932                         continue;
933                 }
934                 if (db->db_state == DB_READ || db->db_state == DB_FILL) {
935                         /* will be handled in dbuf_read_done or dbuf_rele */
936                         db->db_freed_in_flight = TRUE;
937                         mutex_exit(&db->db_mtx);
938                         continue;
939                 }
940                 if (refcount_count(&db->db_holds) == 0) {
941                         ASSERT(db->db_buf);
942                         dbuf_clear(db);
943                         continue;
944                 }
945                 /* The dbuf is referenced */

947                 if (db->db_last_dirty != NULL) {
948                         dbuf_dirty_record_t *dr = db->db_last_dirty;

950                         if (dr->dr_txg == txg) {
951                                 /*
952                                  * This buffer is "in-use", re-adjust the file
953                                  * size to reflect that this buffer may
954                                  * contain new data when we sync.
955                                  */
956                                 if (db->db_blkid != DMU_SPILL_BLKID &&
957                                     db->db_blkid > dn->dn_maxblkid)
958                                         dn->dn_maxblkid = db->db_blkid;
959                                 dbuf_unoverride(dr);
960                         } else {
961                                 /*
```



```
962                                  * This dbuf is not dirty in the open context.
963                                  * Either uncache it (if its not referenced in
964                                  * the open context) or reset its contents to
965                                  * empty.
966                                  */
967                                 dbuf_fix_old_data(db, txg);
968                         }
969                 }
970                 /* clear the contents if its cached */
971                 if (db->db_state == DB_CACHED) {
972                         ASSERT(db->db.db_data != NULL);
973                         arc_release(db->db_buf, db);
974                         bzero(db->db.db_data, db->db.db_size);
975                         arc_buf_freeze(db->db_buf);
976                 }

978                 mutex_exit(&db->db_mtx);
979         }
980         mutex_exit(&dn->dn_dbufs_mtx);
981 }
```

unchanged_portion_omitted_

```
1365 /*
1366  * Undirty a buffer in the transaction group referenced by the given
1367  * transaction.  Return whether this evicted the dbuf.
1368  */
1369 static boolean_t
1370 dbuf_undirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1371 {
1372         dnode_t *dn;
1373         uint64_t txg = tx->tx_txg;
1374         dbuf_dirty_record_t *dr, **drp;

1376         ASSERT(txg != 0);
1377         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1378         ASSERT0(db->db_level);
1379         ASSERT(MUTEX_HELD(&db->db_mtx));

1381         /*
1382          * If this buffer is not dirty, we're done.
1383          */
1384         for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
1385                 if (dr->dr_txg <= txg)
1386                         break;
1387         if (dr == NULL || dr->dr_txg < txg)
1388                 return (B_FALSE);
1389         ASSERT(dr->dr_txg == txg);
1390         ASSERT(dr->dr_dbuf == db);

1392         DB_DNODE_ENTER(db);
1393         dn = DB_DNODE(db);

1395         dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);

1397         ASSERT(db->db.db_size != 0);

1399         /*
1400          * Any space we accounted for in dp_dirty_* will be cleaned up by
1401          * dsl_pool_sync().  This is relatively rare so the discrepancy
1402          * is not a big deal.
1403          */

1405         *drp = dr->dr_next;

1407         /*
1408          * Note that there are three places in dbuf_dirty()
```

```
1409          * where this dirty record may be put on a list.
1410          * Make sure to do a list_remove corresponding to
1411          * every one of those list_insert calls.
1412          */
1413         if (dr->dr_parent) {
1414                 mutex_enter(&dr->dr_parent->dt.di.dr_mtx);
1415                 list_remove(&dr->dr_parent->dt.di.dr_children, dr);
1416                 mutex_exit(&dr->dr_parent->dt.di.dr_mtx);
1417         } else if (db->db_blkid == DMU_SPILL_BLKID ||
1418             db->db_level+1 == dn->dn_nlevels) {
1419                 ASSERT(db->db_blkptr == NULL || db->db_parent == dn->dn_dbuf);
1420                 mutex_enter(&dn->dn_mtx);
1421                 list_remove(&dn->dn_dirty_records[txg & TXG_MASK], dr);
1422                 mutex_exit(&dn->dn_mtx);
1423         }
1424         DB_DNODE_EXIT(db);

1426         if (db->db_state != DB_NOFILL) {
1427                 dbuf_unoverride(dr);

1429                 ASSERT(db->db_buf != NULL);
1430                 ASSERT(dr->dt.dl.dr_data != NULL);
1431                 if (dr->dt.dl.dr_data != db->db_buf)
1432                         VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data, db));
1433         }

1435         if (db->db_level != 0) {
1436                 mutex_destroy(&dr->dt.di.dr_mtx);
1437                 list_destroy(&dr->dt.di.dr_children);
1438         }

1440         kmem_free(dr, sizeof (dbuf_dirty_record_t));

1442         ASSERT(db->db_dirtycnt > 0);
1443         db->db_dirtycnt -= 1;

1445         if (refcount_remove(&db->db_holds, (void *)(uintptr_t)txg) == 0) {
1446                 arc_buf_t *buf = db->db_buf;

1448                 ASSERT(db->db_state == DB_NOFILL || arc_released(buf));
1449                 dbuf_clear_data(db);
1373                 dbuf_set_data(db, NULL);
1450                 VERIFY(arc_buf_remove_ref(buf, db));
1451                 dbuf_evict(db);
1452                 return (B_TRUE);
1453         }

1455         return (B_FALSE);
1456 }
______unchanged_portion_omitted_

1767 static dmu_buf_impl_t *
1768 dbuf_create(dnode_t *dn, uint8_t level, uint64_t blkid,
1769     dmu_buf_impl_t *parent, blkptr_t *blkptr)
1770 {
1771         objset_t *os = dn->dn_objset;
1772         dmu_buf_impl_t *db, *odb;

1774         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1775         ASSERT(dn->dn_type != DMU_OT_NONE);

1777         db = kmem_cache_alloc(dbuf_cache, KM_SLEEP);

1779         db->db_objset = os;
1780         db->db.db_object = dn->dn_object;
1781         db->db_level = level;
```



```
1782         db->db_blkid = blkid;
1783         db->db_last_dirty = NULL;
1784         db->db_dirtycnt = 0;
1785         db->db_dnode_handle = dn->dn_handle;
1786         db->db_parent = parent;
1787         db->db_blkptr = blkptr;

1789         db->db_user = NULL;
1713         db->db_user_ptr = NULL;
1714         db->db_evict_func = NULL;
1790         db->db_immediate_evict = 0;
1791         db->db_freed_in_flight = 0;

1793         if (blkid == DMU_BONUS_BLKID) {
1794                 ASSERT3P(parent, ==, dn->dn_dbuf);
1795                 db->db.db_size = DN_MAX_BONUSLEN -
1796                     (dn->dn_nblkptr-1) * sizeof (blkptr_t);
1797                 ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
1798                 db->db.db_offset = DMU_BONUS_BLKID;
1799                 db->db_state = DB_UNCACHED;
1800                 /* the bonus dbuf is not placed in the hash table */
1801                 arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1802                 return (db);
1803         } else if (blkid == DMU_SPILL_BLKID) {
1804                 db->db.db_size = (blkptr != NULL) ?
1805                     BP_GET_LSIZE(blkptr) : SPA_MINBLOCKSIZE;
1806                 db->db.db_offset = 0;
1807         } else {
1808                 int blocksize =
1809                     db->db_level ? 1 << dn->dn_indblkshift : dn->dn_datablksz;
1810                 db->db.db_size = blocksize;
1811                 db->db.db_offset = db->db_blkid * blocksize;
1812         }

1814         /*
1815          * Hold the dn_dbufs_mtx while we get the new dbuf
1816          * in the hash table *and* added to the dbufs list.
1817          * This prevents a possible deadlock with someone
1818          * trying to look up this dbuf before its added to the
1819          * dn_dbufs list.
1820          */
1821         mutex_enter(&dn->dn_dbufs_mtx);
1822         db->db_state = DB_EVICTING;
1823         if ((odb = dbuf_hash_insert(db)) != NULL) {
1824                 /* someone else inserted it first */
1825                 kmem_cache_free(dbuf_cache, db);
1826                 mutex_exit(&dn->dn_dbufs_mtx);
1827                 return (odb);
1828         }
1829         avl_add(&dn->dn_dbufs, db);
1830         if (db->db_level == 0 && db->db_blkid >=
1831             dn->dn_unlisted_l0_blkid)
1832                 dn->dn_unlisted_l0_blkid = db->db_blkid + 1;
1833         db->db_state = DB_UNCACHED;
1834         mutex_exit(&dn->dn_dbufs_mtx);
1835         arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);

1837         if (parent && parent != dn->dn_dbuf)
1838                 dbuf_add_ref(parent, db);

1840         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1841             refcount_count(&dn->dn_holds) > 0);
1842         (void) refcount_add(&dn->dn_holds, db);
1843         atomic_inc_32(&dn->dn_dbufs_count);

1845         dprintf_dbuf(db, "db=%p\n", db);
```

```
1847         return (db);
1848 }
_____unchanged_portion_omitted_

2131 /*
2132  * dbuf_rele() for an already-locked dbuf.  This is necessary to allow
2133  * db_dirtycnt and db_holds to be updated atomically.
2134  */
2135 void
2136 dbuf_rele_and_unlock(dmu_buf_impl_t *db, void *tag)
2137 {
2138         int64_t holds;

2140         ASSERT(MUTEX_HELD(&db->db_mtx));
2141         DBUF_VERIFY(db);

2143         /*
2144          * Remove the reference to the dbuf before removing its hold on the
2145          * dnode so we can guarantee in dnode_move() that a referenced bonus
2146          * buffer has a corresponding dnode hold.
2147          */
2148         holds = refcount_remove(&db->db_holds, tag);
2149         ASSERT(holds >= 0);

2151         /*
2152          * We can't freeze indirects if there is a possibility that they
2153          * may be modified in the current syncing context.
2154          */
2155         if (db->db_buf && holds == (db->db_level == 0 ? db->db_dirtycnt : 0))
2156                 arc_buf_freeze(db->db_buf);

2158         if (holds == db->db_dirtycnt &&
2159             db->db_level == 0 && db->db_immediate_evict)
2160                 dbuf_evict_user(db);

2162         if (holds == 0) {
2163                 if (db->db_blkid == DMU_BONUS_BLKID) {
2164                         mutex_exit(&db->db_mtx);

2166                         /*
2167                          * If the dnode moves here, we cannot cross this barrier
2168                          * until the move completes.
2169                          */
2170                         DB_DNODE_ENTER(db);
2171                         atomic_dec_32(&DB_DNODE(db)->dn_dbufs_count);
2172                         DB_DNODE_EXIT(db);
2173                         /*
2174                          * The bonus buffer's dnode hold is no longer discounted
2175                          * in dnode_move(). The dnode cannot move until after
2176                          * the dnode_rele().
2177                          */
2178                         dnode_rele(DB_DNODE(db), db);
2179                 } else if (db->db_buf == NULL) {
2180                         /*
2181                          * This is a special case: we never associated this
2182                          * dbuf with any data allocated from the ARC.
2183                          */
2184                         ASSERT(db->db_state == DB_UNCACHED ||
2185                             db->db_state == DB_NOFILL);
2186                         dbuf_evict(db);
2187                 } else if (arc_released(db->db_buf)) {
2188                         arc_buf_t *buf = db->db_buf;
2189                         /*
2190                          * This dbuf has anonymous data associated with it.
2191                          */
```



```
2192                         dbuf_clear_data(db);
2117                         dbuf_set_data(db, NULL);
2193                         VERIFY(arc_buf_remove_ref(buf, db));
2194                         dbuf_evict(db);
2195                 } else {
2196                         VERIFY(!arc_buf_remove_ref(db->db_buf, db));

2198                         /*
2199                          * A dbuf will be eligible for eviction if either the
2200                          * 'primarycache' property is set or a duplicate
2201                          * copy of this buffer is already cached in the arc.
2202                          *
2203                          * In the case of the 'primarycache' a buffer
2204                          * is considered for eviction if it matches the
2205                          * criteria set in the property.
2206                          *
2207                          * To decide if our buffer is considered a
2208                          * duplicate, we must call into the arc to determine
2209                          * if multiple buffers are referencing the same
2210                          * block on-disk. If so, then we simply evict
2211                          * ourselves.
2212                          */
2213                         if (!DBUF_IS_CACHEABLE(db)) {
2214                                 if (db->db_blkptr != NULL &&
2215                                     !BP_IS_HOLE(db->db_blkptr) &&
2216                                     !BP_IS_EMBEDDED(db->db_blkptr)) {
2217                                         spa_t *spa =
2218                                             dmu_objset_spa(db->db_objset);
2219                                         blkptr_t bp = *db->db_blkptr;
2220                                         dbuf_clear(db);
2221                                         arc_freed(spa, &bp);
2222                                 } else {
2223                                         dbuf_clear(db);
2224                                 }
2225                         } else if (db->db_objset->os_evicting ||
2226                             arc_buf_eviction_needed(db->db_buf)) {
2150                         } else if (arc_buf_eviction_needed(db->db_buf)) {
2227                                 dbuf_clear(db);
2228                         } else {
2229                                 mutex_exit(&db->db_mtx);
2230                         }
2231                 }
2232         } else {
2233                 mutex_exit(&db->db_mtx);
2234         }
2235 }
_____unchanged_portion_omitted_

2244 void *
2245 dmu_buf_replace_user(dmu_buf_t *db_fake, dmu_buf_user_t *old_user,
2246     dmu_buf_user_t *new_user)
2169 dmu_buf_set_user(dmu_buf_t *db_fake, void *user_ptr,
2170     dmu_buf_evict_func_t *evict_func)
2247 {
2248         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

2250         mutex_enter(&db->db_mtx);
2251         dbuf_verify_user(db, DBVU_NOT_EVICTING);
2252         if (db->db_user == old_user)
2253                 db->db_user = new_user;
2254         else
2255                 old_user = db->db_user;
2256         dbuf_verify_user(db, DBVU_NOT_EVICTING);
2257         mutex_exit(&db->db_mtx);

2259         return (old_user);
```

```
2172         return (dmu_buf_update_user(db_fake, NULL, user_ptr, evict_func));
2260 }

2262 void *
2263 dmu_buf_set_user(dmu_buf_t *db_fake, dmu_buf_user_t *user)
2176 dmu_buf_set_user_ie(dmu_buf_t *db_fake, void *user_ptr,
2177     dmu_buf_evict_func_t *evict_func)
2264 {
2265         return (dmu_buf_replace_user(db_fake, NULL, user));
2266 }

2268 void *
2269 dmu_buf_set_user_ie(dmu_buf_t *db_fake, dmu_buf_user_t *user)
2270 {
2271         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

2273         db->db_immediate_evict = TRUE;
2274         return (dmu_buf_set_user(db_fake, user));
2182         return (dmu_buf_update_user(db_fake, NULL, user_ptr, evict_func));
2275 }

2277 void *
2278 dmu_buf_remove_user(dmu_buf_t *db_fake, dmu_buf_user_t *user)
2186 dmu_buf_update_user(dmu_buf_t *db_fake, void *old_user_ptr, void *user_ptr,
2187     dmu_buf_evict_func_t *evict_func)
2279 {
2280         return (dmu_buf_replace_user(db_fake, user, NULL));
2189         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
2190         ASSERT(db->db_level == 0);

2192         ASSERT((user_ptr == NULL) == (evict_func == NULL));

2194         mutex_enter(&db->db_mtx);

2196         if (db->db_user_ptr == old_user_ptr) {
2197                 db->db_user_ptr = user_ptr;
2198                 db->db_evict_func = evict_func;
2199         } else {
2200                 old_user_ptr = db->db_user_ptr;
2201         }

2203         mutex_exit(&db->db_mtx);
2204         return (old_user_ptr);
2281 }

2283 void *
2284 dmu_buf_get_user(dmu_buf_t *db_fake)
2285 {
2286         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
2211         ASSERT(!refcount_is_zero(&db->db_holds));

2288         dbuf_verify_user(db, DBVU_NOT_EVICTING);
2289         return (db->db_user);
2213         return (db->db_user_ptr);
2290 }

2292 void
2293 dmu_buf_user_evict_wait()
2294 {
2295         taskq_wait(dbu_evict_taskq);
2296 }

2298 boolean_t
2299 dmu_buf_freeable(dmu_buf_t *dbuf)
2300 {
2301         boolean_t res = B_FALSE;
```



```
2302         dmu_buf_impl_t *db = (dmu_buf_impl_t *)dbuf;

2304         if (db->db_blkptr)
2305                 res = dsl_dataset_block_freeable(db->db_objset->os_dsl_dataset,
2306                     db->db_blkptr, db->db_blkptr->blk_birth);

2308         return (res);
2309 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   46534 Tue Jan  6 10:37:47 2015
new/usr/src/uts/common/fs/zfs/dmu_objset.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

/* Portions Copyright 2010 Robert Milkowski */

#include <sys/cred.h>
#include <sys/zfs_context.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_synctask.h>
#include <sys/dsl_deleg.h>
#include <sys/dnode.h>
#include <sys/dbuf.h>
#include <sys/zvol.h>
#include <sys/dmu_tx.h>
#include <sys/zap.h>
#include <sys/zil.h>
#include <sys/dmu_impl.h>
#include <sys/zfs_ioctl.h>
#include <sys/sa.h>
#include <sys/zfs_onexit.h>
#include <sys/dsl_destroy.h>

/*
 * Needed to close a window in dnode_move() that allows the objset to be freed
 * before it can be safely accessed.
 */
krwlock_t os_lock;
```



```
void
dmu_objset_init(void)
{
        rw_init(&os_lock, NULL, RW_DEFAULT, NULL);
}
```

*unchanged_portion_omitted_*

```
int
dmu_objset_open_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
    objset_t **osp)
{
        objset_t *os;
        int i, err;

        ASSERT(ds == NULL || MUTEX_HELD(&ds->ds_opening_lock));

        os = kmem_zalloc(sizeof (objset_t), KM_SLEEP);
        os->os_dsl_dataset = ds;
        os->os_spa = spa;
        os->os_rootbp = bp;
        if (!BP_IS_HOLE(os->os_rootbp)) {
                arc_flags_t aflags = ARC_FLAG_WAIT;
                zbookmark_phys_t zb;
                SET_BOOKMARK(&zb, ds ? ds->ds_object : DMU_META_OBJSET,
                    ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);

                if (DMU_OS_IS_L2CACHEABLE(os))
                        aflags |= ARC_FLAG_L2CACHE;
                if (DMU_OS_IS_L2COMPRESSIBLE(os))
                        aflags |= ARC_FLAG_L2COMPRESS;

                dprintf_bp(os->os_rootbp, "reading %s", "");
                err = arc_read(NULL, spa, os->os_rootbp,
                    arc_getbuf_func, &os->os_phys_buf,
                    ZIO_PRIORITY_SYNC_READ, ZIO_FLAG_CANFAIL, &aflags, &zb);
                if (err != 0) {
                        kmem_free(os, sizeof (objset_t));
                        /* convert checksum errors into IO errors */
                        if (err == ECKSUM)
                                err = SET_ERROR(EIO);
                        return (err);
                }

                /* Increase the blocksize if we are permitted. */
                if (spa_version(spa) >= SPA_VERSION_USERSPACE &&
                    arc_buf_size(os->os_phys_buf) < sizeof (objset_phys_t)) {
                        arc_buf_t *buf = arc_buf_alloc(spa,
                            sizeof (objset_phys_t), &os->os_phys_buf,
                            ARC_BUFC_METADATA);
                        bzero(buf->b_data, sizeof (objset_phys_t));
                        bcopy(os->os_phys_buf->b_data, buf->b_data,
                            arc_buf_size(os->os_phys_buf));
                        (void) arc_buf_remove_ref(os->os_phys_buf,
                            &os->os_phys_buf);
                        os->os_phys_buf = buf;
                }

                os->os_phys = os->os_phys_buf->b_data;
                os->os_flags = os->os_phys->os_flags;
        } else {
                int size = spa_version(spa) >= SPA_VERSION_USERSPACE ?
                    sizeof (objset_phys_t) : OBJSET_OLD_PHYS_SIZE;
                os->os_phys_buf = arc_buf_alloc(spa, size,
                    &os->os_phys_buf, ARC_BUFC_METADATA);
                os->os_phys = os->os_phys_buf->b_data;
                bzero(os->os_phys, size);
```

```
342         }
344         /*
345          * Note: the changed_cb will be called once before the register
346          * func returns, thus changing the checksum/compression from the
347          * default (fletcher2/off).  Snapshots don't need to know about
348          * checksum/compression/copies.
349          */
350         if (ds != NULL) {
351                 err = dsl_prop_register(ds,
352                     zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
353                     primary_cache_changed_cb, os);
354                 if (err == 0) {
355                         err = dsl_prop_register(ds,
356                             zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
357                             secondary_cache_changed_cb, os);
358                 }
359                 if (!ds->ds_is_snapshot) {
358                 if (!dsl_dataset_is_snapshot(ds)) {
360                         if (err == 0) {
361                                 err = dsl_prop_register(ds,
362                                     zfs_prop_to_name(ZFS_PROP_CHECKSUM),
363                                     checksum_changed_cb, os);
364                         }
365                         if (err == 0) {
366                                 err = dsl_prop_register(ds,
367                                     zfs_prop_to_name(ZFS_PROP_COMPRESSION),
368                                     compression_changed_cb, os);
369                         }
370                         if (err == 0) {
371                                 err = dsl_prop_register(ds,
372                                     zfs_prop_to_name(ZFS_PROP_COPIES),
373                                     copies_changed_cb, os);
374                         }
375                         if (err == 0) {
376                                 err = dsl_prop_register(ds,
377                                     zfs_prop_to_name(ZFS_PROP_DEDUP),
378                                     dedup_changed_cb, os);
379                         }
380                         if (err == 0) {
381                                 err = dsl_prop_register(ds,
382                                     zfs_prop_to_name(ZFS_PROP_LOGBIAS),
383                                     logbias_changed_cb, os);
384                         }
385                         if (err == 0) {
386                                 err = dsl_prop_register(ds,
387                                     zfs_prop_to_name(ZFS_PROP_SYNC),
388                                     sync_changed_cb, os);
389                         }
390                         if (err == 0) {
391                                 err = dsl_prop_register(ds,
392                                     zfs_prop_to_name(
393                                     ZFS_PROP_REDUNDANT_METADATA),
394                                     redundant_metadata_changed_cb, os);
395                         }
396                         if (err == 0) {
397                                 err = dsl_prop_register(ds,
398                                     zfs_prop_to_name(ZFS_PROP_RECORDSIZE),
399                                     recordsize_changed_cb, os);
400                         }
401                 }
402                 if (err != 0) {
403                         VERIFY(arc_buf_remove_ref(os->os_phys_buf,
404                             &os->os_phys_buf));
405                         kmem_free(os, sizeof (objset_t));
406                         return (err);
```



```
407                 }
408         } else {
409                 /* It's the meta-objset. */
410                 os->os_checksum = ZIO_CHECKSUM_FLETCHER_4;
411                 os->os_compress = ZIO_COMPRESS_LZJB;
412                 os->os_copies = spa_max_replication(spa);
413                 os->os_dedup_checksum = ZIO_CHECKSUM_OFF;
414                 os->os_dedup_verify = B_FALSE;
415                 os->os_logbias = ZFS_LOGBIAS_LATENCY;
416                 os->os_sync = ZFS_SYNC_STANDARD;
417                 os->os_primary_cache = ZFS_CACHE_ALL;
418                 os->os_secondary_cache = ZFS_CACHE_ALL;
419         }
421         if (ds == NULL || !ds->ds_is_snapshot)
420         if (ds == NULL || !dsl_dataset_is_snapshot(ds))
422                 os->os_zil_header = os->os_phys->os_zil_header;
423         os->os_zil = zil_alloc(os, &os->os_zil_header);
425         for (i = 0; i < TXG_SIZE; i++) {
426                 list_create(&os->os_dirty_dnodes[i], sizeof (dnode_t),
427                     offsetof(dnode_t, dn_dirty_link[i]));
428                 list_create(&os->os_free_dnodes[i], sizeof (dnode_t),
429                     offsetof(dnode_t, dn_dirty_link[i]));
430         }
431         list_create(&os->os_dnodes, sizeof (dnode_t),
432             offsetof(dnode_t, dn_link));
433         list_create(&os->os_downgraded_dbufs, sizeof (dmu_buf_impl_t),
434             offsetof(dmu_buf_impl_t, db_link));
436         mutex_init(&os->os_lock, NULL, MUTEX_DEFAULT, NULL);
437         mutex_init(&os->os_obj_lock, NULL, MUTEX_DEFAULT, NULL);
438         mutex_init(&os->os_user_ptr_lock, NULL, MUTEX_DEFAULT, NULL);
440         dnode_special_open(os, &os->os_phys->os_meta_dnode,
441             DMU_META_DNODE_OBJECT, &os->os_meta_dnode);
439         DMU_META_DNODE(os) = dnode_special_open(os,
440             &os->os_phys->os_meta_dnode, DMU_META_DNODE_OBJECT,
441             &os->os_meta_dnode);
442         if (arc_buf_size(os->os_phys_buf) >= sizeof (objset_phys_t)) {
443                 dnode_special_open(os, &os->os_phys->os_userused_dnode,
444                     DMU_USERUSED_OBJECT, &os->os_userused_dnode);
445                 dnode_special_open(os, &os->os_phys->os_groupused_dnode,
446                     DMU_GROUPUSED_OBJECT, &os->os_groupused_dnode);
443                 DMU_USERUSED_DNODE(os) = dnode_special_open(os,
444                     &os->os_phys->os_userused_dnode, DMU_USERUSED_OBJECT,
445                     &os->os_userused_dnode);
446                 DMU_GROUPUSED_DNODE(os) = dnode_special_open(os,
447                     &os->os_phys->os_groupused_dnode, DMU_GROUPUSED_OBJECT,
448                     &os->os_groupused_dnode);
447         }
449         *osp = os;
450         return (0);
451 }
```

*unchanged_portion_omitted_*

```
505 /*
506  * dsl_pool must not be held when this is called.
507  * Upon successful return, there will be a longhold on the dataset,
508  * and the dsl_pool will not be held.
509  */
510 int
511 dmu_objset_own(const char *name, dmu_objset_type_t type,
512     boolean_t readonly, void *tag, objset_t **osp)
513 {
```

```
514         dsl_pool_t *dp;
515         dsl_dataset_t *ds;
516         int err;

518         err = dsl_pool_hold(name, FTAG, &dp);
519         if (err != 0)
520                 return (err);
521         err = dsl_dataset_own(dp, name, tag, &ds);
522         if (err != 0) {
523                 dsl_pool_rele(dp, FTAG);
524                 return (err);
525         }

527         err = dmu_objset_from_ds(ds, osp);
528         dsl_pool_rele(dp, FTAG);
529         if (err != 0) {
530                 dsl_dataset_disown(ds, tag);
531         } else if (type != DMU_OST_ANY && type != (*osp)->os_phys->os_type) {
532                 dsl_dataset_disown(ds, tag);
533                 return (SET_ERROR(EINVAL));
534         } else if (!readonly && ds->ds_is_snapshot) {
536         } else if (!readonly && dsl_dataset_is_snapshot(ds)) {
535                 dsl_dataset_disown(ds, tag);
536                 return (SET_ERROR(EROFS));
537         }
538         return (err);
539 }
      unchanged_portion_omitted_

587 void
588 dmu_objset_evict_dbufs(objset_t *os)
589 {
590         dnode_t dn_marker;
591         dnode_t *dn;

593         mutex_enter(&os->os_lock);
594         dn = list_head(&os->os_dnodes);
595         while (dn != NULL) {

596         /* process the mdn last, since the other dnodes have holds on it */
597         list_remove(&os->os_dnodes, DMU_META_DNODE(os));
598         list_insert_tail(&os->os_dnodes, DMU_META_DNODE(os));

596                 /*
597                  * Skip dnodes without holds.  We have to do this dance
598                  * because dnode_add_ref() only works if there is already a
599                  * hold.  If the dnode has no holds, then it has no dbufs.
601          * Find the first dnode with holds.  We have to do this dance
602          * because dnode_add_ref() only works if you already have a
603          * hold.  If there are no holds then it has no dbufs so OK to
604          * skip.
600                  */
601                 if (dnode_add_ref(dn, FTAG)) {
602                         list_insert_after(&os->os_dnodes, dn, &dn_marker);
603                         mutex_exit(&os->os_lock);
606         for (dn = list_head(&os->os_dnodes);
607             dn && !dnode_add_ref(dn, FTAG);
608             dn = list_next(&os->os_dnodes, dn))
609                 continue;

611         while (dn) {
612                 dnode_t *next_dn = dn;

614                 do {
615                         next_dn = list_next(&os->os_dnodes, next_dn);
616                 } while (next_dn && !dnode_add_ref(next_dn, FTAG));
```



```
618                 mutex_exit(&os->os_lock);
605                         dnode_evict_dbufs(dn);
606                         dnode_rele(dn, FTAG);

608                         mutex_enter(&os->os_lock);
609                         dn = list_next(&os->os_dnodes, &dn_marker);
610                         list_remove(&os->os_dnodes, &dn_marker);
611                 } else {
612                         dn = list_next(&os->os_dnodes, dn);
622                 dn = next_dn;
613                 }
614         }
615         mutex_exit(&os->os_lock);

617         if (DMU_USERUSED_DNODE(os) != NULL) {
618                 dnode_evict_dbufs(DMU_GROUPUSED_DNODE(os));
619                 dnode_evict_dbufs(DMU_USERUSED_DNODE(os));
620         }
621         dnode_evict_dbufs(DMU_META_DNODE(os));
622 }

624 /*
625  * Objset eviction processing is split into into two pieces.
626  * The first marks the objset as evicting, evicts any dbufs that
627  * have a refcount of zero, and then queues up the objset for the
628  * second phase of eviction.  Once os->os_dnodes has been cleared by
629  * dnode_buf_pageout()->dnode_destroy(), the second phase is executed.
630  * The second phase closes the special dnodes, dequeues the objset from
631  * the list of those undergoing eviction, and finally frees the objset.
632  *
633  * NOTE: Due to asynchronous eviction processing (invocation of
634  *       dnode_buf_pageout()), it is possible for the meta dnode for the
635  *       objset to have no holds even though os->os_dnodes is not empty.
636  */
637 void
638 dmu_objset_evict(objset_t *os)
639 {
640         dsl_dataset_t *ds = os->os_dsl_dataset;

642         for (int t = 0; t < TXG_SIZE; t++)
643                 ASSERT(!dmu_objset_is_dirty(os, t));

645         if (ds) {
646                 if (!ds->ds_is_snapshot) {
636                 if (!dsl_dataset_is_snapshot(ds)) {
647                         VERIFY0(dsl_prop_unregister(ds,
648                             zfs_prop_to_name(ZFS_PROP_CHECKSUM),
649                             checksum_changed_cb, os));
650                         VERIFY0(dsl_prop_unregister(ds,
651                             zfs_prop_to_name(ZFS_PROP_COMPRESSION),
652                             compression_changed_cb, os));
653                         VERIFY0(dsl_prop_unregister(ds,
654                             zfs_prop_to_name(ZFS_PROP_COPIES),
655                             copies_changed_cb, os));
656                         VERIFY0(dsl_prop_unregister(ds,
657                             zfs_prop_to_name(ZFS_PROP_DEDUP),
658                             dedup_changed_cb, os));
659                         VERIFY0(dsl_prop_unregister(ds,
660                             zfs_prop_to_name(ZFS_PROP_LOGBIAS),
661                             logbias_changed_cb, os));
662                         VERIFY0(dsl_prop_unregister(ds,
663                             zfs_prop_to_name(ZFS_PROP_SYNC),
664                             sync_changed_cb, os));
665                         VERIFY0(dsl_prop_unregister(ds,
666                             zfs_prop_to_name(ZFS_PROP_REDUNDANT_METADATA),
```

```
                            redundant_metadata_changed_cb, os));
                        VERIFY0(dsl_prop_unregister(ds,
                            zfs_prop_to_name(ZFS_PROP_RECORDSIZE),
                            recordsize_changed_cb, os));
                }
                VERIFY0(dsl_prop_unregister(ds,
                    zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
                    primary_cache_changed_cb, os));
                VERIFY0(dsl_prop_unregister(ds,
                    zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
                    secondary_cache_changed_cb, os));
        }

        if (os->os_sa)
                sa_tear_down(os);

        os->os_evicting = B_TRUE;
        dmu_objset_evict_dbufs(os);

        mutex_enter(&os->os_lock);
        spa_evicting_os_register(os->os_spa, os);
        if (list_is_empty(&os->os_dnodes)) {
                mutex_exit(&os->os_lock);
                dmu_objset_evict_done(os);
        } else {
                mutex_exit(&os->os_lock);
        }
}

void
dmu_objset_evict_done(objset_t *os)
{
        ASSERT3P(list_head(&os->os_dnodes), ==, NULL);

        dnode_special_close(&os->os_meta_dnode);
        if (DMU_USERUSED_DNODE(os)) {
                dnode_special_close(&os->os_userused_dnode);
                dnode_special_close(&os->os_groupused_dnode);
        }
        zil_free(os->os_zil);

        ASSERT3P(list_head(&os->os_dnodes), ==, NULL);

        VERIFY(arc_buf_remove_ref(os->os_phys_buf, &os->os_phys_buf));

        /*
         * This is a barrier to prevent the objset from going away in
         * dnode_move() until we can safely ensure that the objset is still in
         * use. We consider the objset valid before the barrier and invalid
         * after the barrier.
         */
        rw_enter(&os_lock, RW_READER);
        rw_exit(&os_lock);

        mutex_destroy(&os->os_lock);
        mutex_destroy(&os->os_obj_lock);
        mutex_destroy(&os->os_user_ptr_lock);
        spa_evicting_os_deregister(os->os_spa, os);
        kmem_free(os, sizeof (objset_t));
}
```

*unchanged_portion_omitted_*

```
/*ARGSUSED*/
static int
dmu_objset_clone_check(void *arg, dmu_tx_t *tx)
{
```



```
        dmu_objset_clone_arg_t *doca = arg;
        dsl_dir_t *pdd;
        const char *tail;
        int error;
        dsl_dataset_t *origin;
        dsl_pool_t *dp = dmu_tx_pool(tx);

        if (strchr(doca->doca_clone, '@') != NULL)
                return (SET_ERROR(EINVAL));

        error = dsl_dir_hold(dp, doca->doca_clone, FTAG, &pdd, &tail);
        if (error != 0)
                return (error);
        if (tail == NULL) {
                dsl_dir_rele(pdd, FTAG);
                return (SET_ERROR(EEXIST));
        }
        /* You can't clone across pools. */
        if (pdd->dd_pool != dp) {
                dsl_dir_rele(pdd, FTAG);
                return (SET_ERROR(EXDEV));
        }
        error = dsl_fs_ss_limit_check(pdd, 1, ZFS_PROP_FILESYSTEM_LIMIT, NULL,
            doca->doca_cred);
        if (error != 0) {
                dsl_dir_rele(pdd, FTAG);
                return (SET_ERROR(EDQUOT));
        }
        dsl_dir_rele(pdd, FTAG);

        error = dsl_dataset_hold(dp, doca->doca_origin, FTAG, &origin);
        if (error != 0)
                return (error);

        /* You can't clone across pools. */
        if (origin->ds_dir->dd_pool != dp) {
                dsl_dataset_rele(origin, FTAG);
                return (SET_ERROR(EXDEV));
        }

        /* You can only clone snapshots, not the head datasets. */
        if (!origin->ds_is_snapshot) {
        if (!dsl_dataset_is_snapshot(origin)) {
                dsl_dataset_rele(origin, FTAG);
                return (SET_ERROR(EINVAL));
        }
        dsl_dataset_rele(origin, FTAG);

        return (0);
}
```

*unchanged_portion_omitted_*

```
int
dmu_objset_is_snapshot(objset_t *os)
{
        if (os->os_dsl_dataset != NULL)
                return (os->os_dsl_dataset->ds_is_snapshot);
                return (dsl_dataset_is_snapshot(os->os_dsl_dataset));
        else
                return (B_FALSE);
}
```

*unchanged_portion_omitted_*

```
**********************************************************
   59887 Tue Jan  6 10:37:47 2015
new/usr/src/uts/common/fs/zfs/dmu_send.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.com>
**********************************************************
______unchanged_portion_omitted_

 545 /*
 546  * Releases dp using the specified tag.
 547  */
 548 static int
 549 dmu_send_impl(void *tag, dsl_pool_t *dp, dsl_dataset_t *ds,
 550     zfs_bookmark_phys_t *fromzb, boolean_t is_clone, boolean_t embedok,
 551     boolean_t large_block_ok, int outfd, vnode_t *vp, offset_t *off)
 552 {
 553         objset_t *os;
 554         dmu_replay_record_t *drr;
 555         dmu_sendarg_t *dsp;
 556         int err;
 557         uint64_t fromtxg = 0;
 558         uint64_t featureflags = 0;

 560         err = dmu_objset_from_ds(ds, &os);
 561         if (err != 0) {
 562                 dsl_pool_rele(dp, tag);
 563                 return (err);
 564         }

 566         drr = kmem_zalloc(sizeof (dmu_replay_record_t), KM_SLEEP);
 567         drr->drr_type = DRR_BEGIN;
 568         drr->drr_u.drr_begin.drr_magic = DMU_BACKUP_MAGIC;
 569         DMU_SET_STREAM_HDRTYPE(drr->drr_u.drr_begin.drr_versioninfo,
 570             DMU_SUBSTREAM);

 572 #ifdef _KERNEL
 573         if (dmu_objset_type(os) == DMU_OST_ZFS) {
 574                 uint64_t version;
 575                 if (zfs_get_zplprop(os, ZFS_PROP_VERSION, &version) != 0) {
 576                         kmem_free(drr, sizeof (dmu_replay_record_t));
 577                         dsl_pool_rele(dp, tag);
 578                         return (SET_ERROR(EINVAL));
 579                 }
 580                 if (version >= ZPL_VERSION_SA) {
 581                         featureflags |= DMU_BACKUP_FEATURE_SA_SPILL;
 582                 }
 583         }
 584 #endif

 586         if (large_block_ok && ds->ds_large_blocks)
 587                 featureflags |= DMU_BACKUP_FEATURE_LARGE_BLOCKS;
 588         if (embedok &&
 589             spa_feature_is_active(dp->dp_spa, SPA_FEATURE_EMBEDDED_DATA)) {
 590                 featureflags |= DMU_BACKUP_FEATURE_EMBED_DATA;
 591                 if (spa_feature_is_active(dp->dp_spa, SPA_FEATURE_LZ4_COMPRESS))
 592                         featureflags |= DMU_BACKUP_FEATURE_EMBED_DATA_LZ4;
 593         } else {
 594                 embedok = B_FALSE;
 595         }

 597         DMU_SET_FEATUREFLAGS(drr->drr_u.drr_begin.drr_versioninfo,
 598             featureflags);
```



```
 600         drr->drr_u.drr_begin.drr_creation_time =
 601             dsl_dataset_phys(ds)->ds_creation_time;
 602         drr->drr_u.drr_begin.drr_type = dmu_objset_type(os);
 603         if (is_clone)
 604                 drr->drr_u.drr_begin.drr_flags |= DRR_FLAG_CLONE;
 605         drr->drr_u.drr_begin.drr_toguid = dsl_dataset_phys(ds)->ds_guid;
 606         if (dsl_dataset_phys(ds)->ds_flags & DS_FLAG_CI_DATASET)
 607                 drr->drr_u.drr_begin.drr_flags |= DRR_FLAG_CI_DATA;

 609         if (fromzb != NULL) {
 610                 drr->drr_u.drr_begin.drr_fromguid = fromzb->zbm_guid;
 611                 fromtxg = fromzb->zbm_creation_txg;
 612         }
 613         dsl_dataset_name(ds, drr->drr_u.drr_begin.drr_toname);
 614         if (!ds->ds_is_snapshot) {
 614         if (!dsl_dataset_is_snapshot(ds)) {
 615                 (void) strlcat(drr->drr_u.drr_begin.drr_toname, "@--head--",
 616                     sizeof (drr->drr_u.drr_begin.drr_toname));
 617         }

 619         dsp = kmem_zalloc(sizeof (dmu_sendarg_t), KM_SLEEP);

 621         dsp->dsa_drr = drr;
 622         dsp->dsa_vp = vp;
 623         dsp->dsa_outfd = outfd;
 624         dsp->dsa_proc = curproc;
 625         dsp->dsa_os = os;
 626         dsp->dsa_off = off;
 627         dsp->dsa_toguid = dsl_dataset_phys(ds)->ds_guid;
 628         ZIO_SET_CHECKSUM(&dsp->dsa_zc, 0, 0, 0, 0);
 629         dsp->dsa_pending_op = PENDING_NONE;
 630         dsp->dsa_incremental = (fromzb != NULL);
 631         dsp->dsa_featureflags = featureflags;

 633         mutex_enter(&ds->ds_sendstream_lock);
 634         list_insert_head(&ds->ds_sendstreams, dsp);
 635         mutex_exit(&ds->ds_sendstream_lock);

 637         dsl_dataset_long_hold(ds, FTAG);
 638         dsl_pool_rele(dp, tag);

 640         if (dump_bytes(dsp, drr, sizeof (dmu_replay_record_t)) != 0) {
 641                 err = dsp->dsa_err;
 642                 goto out;
 643         }

 645         err = traverse_dataset(ds, fromtxg, TRAVERSE_PRE | TRAVERSE_PREFETCH,
 646             backup_cb, dsp);

 648         if (dsp->dsa_pending_op != PENDING_NONE)
 649                 if (dump_bytes(dsp, drr, sizeof (dmu_replay_record_t)) != 0)
 650                         err = SET_ERROR(EINTR);

 652         if (err != 0) {
 653                 if (err == EINTR && dsp->dsa_err != 0)
 654                         err = dsp->dsa_err;
 655                 goto out;
 656         }

 658         bzero(drr, sizeof (dmu_replay_record_t));
 659         drr->drr_type = DRR_END;
 660         drr->drr_u.drr_end.drr_checksum = dsp->dsa_zc;
 661         drr->drr_u.drr_end.drr_toguid = dsp->dsa_toguid;

 663         if (dump_bytes(dsp, drr, sizeof (dmu_replay_record_t)) != 0) {
 664                 err = dsp->dsa_err;
```

```
665                 goto out;
666         }

668 out:
669         mutex_enter(&ds->ds_sendstream_lock);
670         list_remove(&ds->ds_sendstreams, dsp);
671         mutex_exit(&ds->ds_sendstream_lock);

673         kmem_free(drr, sizeof (dmu_replay_record_t));
674         kmem_free(dsp, sizeof (dmu_sendarg_t));

676         dsl_dataset_long_rele(ds, FTAG);

678         return (err);
679 }
```

unchanged_portion_omitted_

```
811 int
812 dmu_send_estimate(dsl_dataset_t *ds, dsl_dataset_t *fromds, uint64_t *sizep)
813 {
814         dsl_pool_t *dp = ds->ds_dir->dd_pool;
815         int err;
816         uint64_t size;

818         ASSERT(dsl_pool_config_held(dp));

820         /* tosnap must be a snapshot */
821         if (!ds->ds_is_snapshot)
822         if (!dsl_dataset_is_snapshot(ds))
823                 return (SET_ERROR(EINVAL));

825         /*
826          * fromsnap must be an earlier snapshot from the same fs as tosnap,
827          * or the origin's fs.
828          */
829         if (fromds != NULL && !dsl_dataset_is_before(ds, fromds, 0))
830                 return (SET_ERROR(EXDEV));

832         /* Get uncompressed size estimate of changed data. */
833         if (fromds == NULL) {
834                 size = dsl_dataset_phys(ds)->ds_uncompressed_bytes;
835         } else {
836                 uint64_t used, comp;
837                 err = dsl_dataset_space_written(fromds, ds,
838                     &used, &comp, &size);
839                 if (err != 0)
840                         return (err);
841         }

843         /*
844          * Assume that space (both on-disk and in-stream) is dominated by
845          * data.  We will adjust for indirect blocks and the copies property,
846          * but ignore per-object space used (eg, dnodes and DRR_OBJECT records).
847          */

849         /*
850          * Subtract out approximate space used by indirect blocks.
851          * Assume most space is used by data blocks (non-indirect, non-dnode).
852          * Assume all blocks are recordsize.  Assume ditto blocks and
853          * internal fragmentation counter out compression.
854          *
855          * Therefore, space used by indirect blocks is sizeof(blkptr_t) per
856          * block, which we observe in practice.
857          */
858         uint64_t recordsize;
859         err = dsl_prop_get_int_ds(ds, "recordsize", &recordsize);
```



```
860         if (err != 0)
861                 return (err);
862         size -= size / recordsize * sizeof (blkptr_t);

864         /* Add in the space for the record associated with each block. */
865         size += size / recordsize * sizeof (dmu_replay_record_t);

867         *sizep = size;

869         return (0);
870 }
```

unchanged_portion_omitted_

```
963 static int
964 dmu_recv_begin_check(void *arg, dmu_tx_t *tx)
965 {
966         dmu_recv_begin_arg_t *drba = arg;
967         dsl_pool_t *dp = dmu_tx_pool(tx);
968         struct drr_begin *drrb = drba->drba_cookie->drc_drrb;
969         uint64_t fromguid = drrb->drr_fromguid;
970         int flags = drrb->drr_flags;
971         int error;
972         uint64_t featureflags = DMU_GET_FEATUREFLAGS(drrb->drr_versioninfo);
973         dsl_dataset_t *ds;
974         const char *tofs = drba->drba_cookie->drc_tofs;

976         /* already checked */
977         ASSERT3U(drrb->drr_magic, ==, DMU_BACKUP_MAGIC);

979         if (DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) ==
980             DMU_COMPOUNDSTREAM ||
981             drrb->drr_type >= DMU_OST_NUMTYPES ||
982             ((flags & DRR_FLAG_CLONE) && drba->drba_origin == NULL))
983                 return (SET_ERROR(EINVAL));

985         /* Verify pool version supports SA if SA_SPILL feature set */
986         if ((featureflags & DMU_BACKUP_FEATURE_SA_SPILL) &&
987             spa_version(dp->dp_spa) < SPA_VERSION_SA)
988                 return (SET_ERROR(ENOTSUP));

990         /*
991          * The receiving code doesn't know how to translate a WRITE_EMBEDDED
992          * record to a plan WRITE record, so the pool must have the
993          * EMBEDDED_DATA feature enabled if the stream has WRITE_EMBEDDED
994          * records.  Same with WRITE_EMBEDDED records that use LZ4 compression.
995          */
996         if ((featureflags & DMU_BACKUP_FEATURE_EMBED_DATA) &&
997             !spa_feature_is_enabled(dp->dp_spa, SPA_FEATURE_EMBEDDED_DATA))
998                 return (SET_ERROR(ENOTSUP));
999         if ((featureflags & DMU_BACKUP_FEATURE_EMBED_DATA_LZ4) &&
1000            !spa_feature_is_enabled(dp->dp_spa, SPA_FEATURE_LZ4_COMPRESS))
1001                return (SET_ERROR(ENOTSUP));

1003        /*
1004         * The receiving code doesn't know how to translate large blocks
1005         * to smaller ones, so the pool must have the LARGE_BLOCKS
1006         * feature enabled if the stream has LARGE_BLOCKS.
1007         */
1008        if ((featureflags & DMU_BACKUP_FEATURE_LARGE_BLOCKS) &&
1009            !spa_feature_is_enabled(dp->dp_spa, SPA_FEATURE_LARGE_BLOCKS))
1010                return (SET_ERROR(ENOTSUP));

1012        error = dsl_dataset_hold(dp, tofs, FTAG, &ds);
1013        if (error == 0) {
1014                /* target fs already exists; recv into temp clone */
```

```
1015                /* Can't recv a clone into an existing fs */
1016                if (flags & DRR_FLAG_CLONE) {
1017                        dsl_dataset_rele(ds, FTAG);
1018                        return (SET_ERROR(EINVAL));
1019                }

1021                error = recv_begin_check_existing_impl(drba, ds, fromguid);
1022                dsl_dataset_rele(ds, FTAG);
1023        } else if (error == ENOENT) {
1024                /* target fs does not exist; must be a full backup or clone */
1025                char buf[MAXNAMELEN];

1027                /*
1028                 * If it's a non-clone incremental, we are missing the
1029                 * target fs, so fail the recv.
1030                 */
1031                if (fromguid != 0 && !(flags & DRR_FLAG_CLONE))
1032                        return (SET_ERROR(ENOENT));

1034                /* Open the parent of tofs */
1035                ASSERT3U(strlen(tofs), <, MAXNAMELEN);
1036                (void) strlcpy(buf, tofs, strrchr(tofs, '/') - tofs + 1);
1037                error = dsl_dataset_hold(dp, buf, FTAG, &ds);
1038                if (error != 0)
1039                        return (error);

1041                /*
1042                 * Check filesystem and snapshot limits before receiving. We'll
1043                 * recheck snapshot limits again at the end (we create the
1044                 * filesystems and increment those counts during begin_sync).
1045                 */
1046                error = dsl_fs_ss_limit_check(ds->ds_dir, 1,
1047                    ZFS_PROP_FILESYSTEM_LIMIT, NULL, drba->drba_cred);
1048                if (error != 0) {
1049                        dsl_dataset_rele(ds, FTAG);
1050                        return (error);
1051                }

1053                error = dsl_fs_ss_limit_check(ds->ds_dir, 1,
1054                    ZFS_PROP_SNAPSHOT_LIMIT, NULL, drba->drba_cred);
1055                if (error != 0) {
1056                        dsl_dataset_rele(ds, FTAG);
1057                        return (error);
1058                }

1060                if (drba->drba_origin != NULL) {
1061                        dsl_dataset_t *origin;
1062                        error = dsl_dataset_hold(dp, drba->drba_origin,
1063                            FTAG, &origin);
1064                        if (error != 0) {
1065                                dsl_dataset_rele(ds, FTAG);
1066                                return (error);
1067                        }
1068                        if (!origin->ds_is_snapshot) {
1068                        if (!dsl_dataset_is_snapshot(origin)) {
1069                                dsl_dataset_rele(origin, FTAG);
1070                                dsl_dataset_rele(ds, FTAG);
1071                                return (SET_ERROR(EINVAL));
1072                        }
1073                        if (dsl_dataset_phys(origin)->ds_guid != fromguid) {
1074                                dsl_dataset_rele(origin, FTAG);
1075                                dsl_dataset_rele(ds, FTAG);
1076                                return (SET_ERROR(ENODEV));
1077                        }
1078                        dsl_dataset_rele(origin, FTAG);
1079                }
```



```
1080                dsl_dataset_rele(ds, FTAG);
1081                error = 0;
1082        }
1083        return (error);
1084 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   17549 Tue Jan  6 10:37:47 2015
new/usr/src/uts/common/fs/zfs/dmu_traverse.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.com>
**********************************************************
______unchanged_portion_omitted_

 488 /*
 489  * NB: dataset must not be changing on-disk (eg, is a snapshot or we are
 490  * in syncing context).
 491  */
 492 static int
 493 traverse_impl(spa_t *spa, dsl_dataset_t *ds, uint64_t objset, blkptr_t *rootbp,
 494     uint64_t txg_start, zbookmark_phys_t *resume, int flags,
 495     blkptr_cb_t func, void *arg)
 496 {
 497         traverse_data_t td;
 498         prefetch_data_t pd = { 0 };
 499         zbookmark_phys_t czb;
 500         int err;

 502         ASSERT(ds == NULL || objset == ds->ds_object);
 503         ASSERT(!(flags & TRAVERSE_PRE) || !(flags & TRAVERSE_POST));

 505         /*
 506          * The data prefetching mechanism (the prefetch thread) is incompatible
 507          * with resuming from a bookmark.
 508          */
 509         ASSERT(resume == NULL || !(flags & TRAVERSE_PREFETCH_DATA));

 511         td.td_spa = spa;
 512         td.td_objset = objset;
 513         td.td_rootbp = rootbp;
 514         td.td_min_txg = txg_start;
 515         td.td_resume = resume;
 516         td.td_func = func;
 517         td.td_arg = arg;
 518         td.td_pfd = &pd;
 519         td.td_flags = flags;
 520         td.td_paused = B_FALSE;

 522         if (spa_feature_is_active(spa, SPA_FEATURE_HOLE_BIRTH)) {
 523                 VERIFY(spa_feature_enabled_txg(spa,
 524                     SPA_FEATURE_HOLE_BIRTH, &td.td_hole_birth_enabled_txg));
 525         } else {
 526                 td.td_hole_birth_enabled_txg = 0;
 527         }

 529         pd.pd_blks_max = zfs_pd_blks_max;
 530         pd.pd_flags = flags;
 531         mutex_init(&pd.pd_mtx, NULL, MUTEX_DEFAULT, NULL);
 532         cv_init(&pd.pd_cv, NULL, CV_DEFAULT, NULL);

 534         /* See comment on ZIL traversal in dsl_scan_visitds. */
 535         if (ds != NULL && !ds->ds_is_snapshot && !BP_IS_HOLE(rootbp)) {
 535         if (ds != NULL && !dsl_dataset_is_snapshot(ds) && !BP_IS_HOLE(rootbp)) {
 536                 arc_flags_t flags = ARC_FLAG_WAIT;
 537                 objset_phys_t *osp;
 538                 arc_buf_t *buf;

 540                 err = arc_read(NULL, td.td_spa, rootbp,
 541                     arc_getbuf_func, &buf,
```



```
 542                     ZIO_PRIORITY_ASYNC_READ, ZIO_FLAG_CANFAIL, &flags, NULL);
 543                 if (err != 0)
 544                         return (err);

 546                 osp = buf->b_data;
 547                 traverse_zil(&td, &osp->os_zil_header);
 548                 (void) arc_buf_remove_ref(buf, &buf);
 549         }

 551         if (!(flags & TRAVERSE_PREFETCH_DATA) ||
 552             0 == taskq_dispatch(system_taskq, traverse_prefetch_thread,
 553             &td, TQ_NOQUEUE))
 554                 pd.pd_exited = B_TRUE;

 556         SET_BOOKMARK(&czb, td.td_objset,
 557             ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
 558         err = traverse_visitbp(&td, NULL, rootbp, &czb);

 560         mutex_enter(&pd.pd_mtx);
 561         pd.pd_cancel = B_TRUE;
 562         cv_broadcast(&pd.pd_cv);
 563         while (!pd.pd_exited)
 564                 cv_wait(&pd.pd_cv, &pd.pd_mtx);
 565         mutex_exit(&pd.pd_mtx);

 567         mutex_destroy(&pd.pd_mtx);
 568         cv_destroy(&pd.pd_cv);

 570         return (err);
 571 }
______unchanged_portion_omitted_
```

```
**********************************************************
   55884 Tue Jan  6 10:37:47 2015
new/usr/src/uts/common/fs/zfs/dnode.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/dbuf.h>
#include <sys/dnode.h>
#include <sys/dmu.h>
#include <sys/dmu_impl.h>
#include <sys/dmu_tx.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_dataset.h>
#include <sys/spa.h>
#include <sys/zio.h>
#include <sys/dmu_zfetch.h>
#include <sys/range_tree.h>

static kmem_cache_t *dnode_cache;
/*
 * Define DNODE_STATS to turn on statistic gathering. By default, it is only
 * turned on when DEBUG is also defined.
 */
#ifdef  DEBUG
#define DNODE_STATS
#endif  /* DEBUG */

#ifdef  DNODE_STATS
#define DNODE_STAT_ADD(stat)                    ((stat)++)
#else
#define DNODE_STAT_ADD(stat)                    /* nothing */
#endif  /* DNODE_STATS */

static dnode_phys_t dnode_phys_zero;
```



```
int zfs_default_bs = SPA_MINBLOCKSHIFT;
int zfs_default_ibs = DN_MAX_INDBLKSHIFT;

static kmem_cbrc_t dnode_move(void *, void *, size_t, void *);

static int
dbuf_compare(const void *x1, const void *x2)
{
        const dmu_buf_impl_t *d1 = x1;
        const dmu_buf_impl_t *d2 = x2;

        if (d1->db_level < d2->db_level) {
                return (-1);
        }
        if (d1->db_level > d2->db_level) {
                return (1);
        }

        if (d1->db_blkid < d2->db_blkid) {
                return (-1);
        }
        if (d1->db_blkid > d2->db_blkid) {
                return (1);
        }

        if (d1->db_state == DB_SEARCH) {
                ASSERT3S(d2->db_state, !=, DB_SEARCH);
                return (-1);
        } else if (d2->db_state == DB_SEARCH) {
                ASSERT3S(d1->db_state, !=, DB_SEARCH);
                return (1);
        }

        if ((uintptr_t)d1 < (uintptr_t)d2) {
                return (-1);
        }
        if ((uintptr_t)d1 > (uintptr_t)d2) {
                return (1);
        }
        return (0);
}
```

*unchanged_portion_omitted_*

```
static dnode_t *
dnode_create(objset_t *os, dnode_phys_t *dnp, dmu_buf_impl_t *db,
    uint64_t object, dnode_handle_t *dnh)
{
        dnode_t *dn;
        dnode_t *dn = kmem_cache_alloc(dnode_cache, KM_SLEEP);

        dn = kmem_cache_alloc(dnode_cache, KM_SLEEP);
        ASSERT(!POINTER_IS_VALID(dn->dn_objset));
        dn->dn_moved = 0;

        /*
         * Defer setting dn_objset until the dnode is ready to be a candidate
         * for the dnode_move() callback.
         */
        dn->dn_object = object;
        dn->dn_dbuf = db;
        dn->dn_handle = dnh;
        dn->dn_phys = dnp;

        if (dnp->dn_datablkszsec) {
                dnode_setdblksz(dn, dnp->dn_datablkszsec << SPA_MINBLOCKSHIFT);
        } else {
```

```
424                 dn->dn_datablksz = 0;
425                 dn->dn_datablkszsec = 0;
426                 dn->dn_datablkshift = 0;
427         }
428         dn->dn_indblkshift = dnp->dn_indblkshift;
429         dn->dn_nlevels = dnp->dn_nlevels;
430         dn->dn_type = dnp->dn_type;
431         dn->dn_nblkptr = dnp->dn_nblkptr;
432         dn->dn_checksum = dnp->dn_checksum;
433         dn->dn_compress = dnp->dn_compress;
434         dn->dn_bonustype = dnp->dn_bonustype;
435         dn->dn_bonuslen = dnp->dn_bonuslen;
436         dn->dn_maxblkid = dnp->dn_maxblkid;
437         dn->dn_have_spill = ((dnp->dn_flags & DNODE_FLAG_SPILL_BLKPTR) != 0);
438         dn->dn_id_flags = 0;

440         dmu_zfetch_init(&dn->dn_zfetch, dn);

442         ASSERT(DMU_OT_IS_VALID(dn->dn_phys->dn_type));

444         mutex_enter(&os->os_lock);
445         if (dnh->dnh_dnode != NULL) {
446                 /* Lost the allocation race. */
447                 mutex_exit(&os->os_lock);
448                 kmem_cache_free(dnode_cache, dn);
449                 return (dnh->dnh_dnode);
450         }

452         /*
453          * Exclude special dnodes from os_dnodes so an empty os_dnodes
454          * signifies that the special dnodes have no references from
455          * their children (the entries in os_dnodes).  This allows
456          * dnode_destroy() to easily determine if the last child has
457          * been removed and then complete eviction of the objset.
458          */
459         if (!DMU_OBJECT_IS_SPECIAL(object))
460                 list_insert_head(&os->os_dnodes, dn);
461         membar_producer();

463         /*
464          * Everything else must be valid before assigning dn_objset
465          * makes the dnode eligible for dnode_move().
466          * Everything else must be valid before assigning dn_objset makes the
467          * dnode eligible for dnode_move().
466          */
467         dn->dn_objset = os;

469         dnh->dnh_dnode = dn;
470         mutex_exit(&os->os_lock);

472         arc_space_consume(sizeof (dnode_t), ARC_SPACE_OTHER);
473         return (dn);
474 }

476 /*
477  * Caller must be holding the dnode handle, which is released upon return.
478  */
479 static void
480 dnode_destroy(dnode_t *dn)
481 {
482         objset_t *os = dn->dn_objset;
483         boolean_t complete_os_eviction = B_FALSE;

485         ASSERT((dn->dn_id_flags & DN_ID_NEW_EXIST) == 0);

487         mutex_enter(&os->os_lock);
```



```
488         POINTER_INVALIDATE(&dn->dn_objset);
489         if (!DMU_OBJECT_IS_SPECIAL(dn->dn_object)) {
490                 list_remove(&os->os_dnodes, dn);
491                 complete_os_eviction =
492                     list_is_empty(&os->os_dnodes) &&
493                     list_link_active(&os->os_evicting_node);
494         }
495         mutex_exit(&os->os_lock);

497         /* the dnode can no longer move, so we can release the handle */
498         zrl_remove(&dn->dn_handle->dnh_zrlock);

500         dn->dn_allocated_txg = 0;
501         dn->dn_free_txg = 0;
502         dn->dn_assigned_txg = 0;

504         dn->dn_dirtyctx = 0;
505         if (dn->dn_dirtyctx_firstset != NULL) {
506                 kmem_free(dn->dn_dirtyctx_firstset, 1);
507                 dn->dn_dirtyctx_firstset = NULL;
508         }
509         if (dn->dn_bonus != NULL) {
510                 mutex_enter(&dn->dn_bonus->db_mtx);
511                 dbuf_evict(dn->dn_bonus);
512                 dn->dn_bonus = NULL;
513         }
514         dn->dn_zio = NULL;

516         dn->dn_have_spill = B_FALSE;
517         dn->dn_oldused = 0;
518         dn->dn_oldflags = 0;
519         dn->dn_olduid = 0;
520         dn->dn_oldgid = 0;
521         dn->dn_newuid = 0;
522         dn->dn_newgid = 0;
523         dn->dn_id_flags = 0;
524         dn->dn_unlisted_l0_blkid = 0;

526         dmu_zfetch_rele(&dn->dn_zfetch);
527         kmem_cache_free(dnode_cache, dn);
528         arc_space_return(sizeof (dnode_t), ARC_SPACE_OTHER);

530         if (complete_os_eviction)
531                 dmu_objset_evict_done(os);
532 }
_____unchanged_portion_omitted_
983 #endif  /* _KERNEL */

985 void
986 dnode_special_close(dnode_handle_t *dnh)
987 {
988         dnode_t *dn = dnh->dnh_dnode;

990         /*
991          * Wait for final references to the dnode to clear.  This can
992          * only happen if the arc is asyncronously evicting state that
993          * has a hold on this dnode while we are trying to evict this
994          * dnode.
995          */
996         while (refcount_count(&dn->dn_holds) > 0)
997                 delay(1);
998         ASSERT(dn->dn_dbuf == NULL ||
999             dmu_buf_get_user(&dn->dn_dbuf->db) == NULL);
1000        zrl_add(&dnh->dnh_zrlock);
1001        dnode_destroy(dn); /* implicit zrl_remove() */
1002        zrl_destroy(&dnh->dnh_zrlock);
```

```
1003         dnh->dnh_dnode = NULL;
1004 }

1006 void
 975 dnode_t *
1007 dnode_special_open(objset_t *os, dnode_phys_t *dnp, uint64_t object,
1008     dnode_handle_t *dnh)
1009 {
1010         dnode_t *dn;

1012         dn = dnode_create(os, dnp, NULL, object, dnh);
 979         dnode_t *dn = dnode_create(os, dnp, NULL, object, dnh);
 980         dnh->dnh_dnode = dn;
1013         zrl_init(&dnh->dnh_zrlock);
1014         DNODE_VERIFY(dn);
 983         return (dn);
1015 }

1017 static void
1018 dnode_buf_pageout(void *dbu)
 987 dnode_buf_pageout(dmu_buf_t *db, void *arg)
1019 {
1020         dnode_children_t *children_dnodes = dbu;
 989         dnode_children_t *children_dnodes = arg;
1021         int i;
 991         int epb = db->db_size >> DNODE_SHIFT;

1023         for (i = 0; i < children_dnodes->dnc_count; i++) {
 993         ASSERT(epb == children_dnodes->dnc_count);

 995         for (i = 0; i < epb; i++) {
1024                 dnode_handle_t *dnh = &children_dnodes->dnc_children[i];
1025                 dnode_t *dn;

1027                 /*
1028                  * The dnode handle lock guards against the dnode moving to
1029                  * another valid address, so there is no need here to guard
1030                  * against changes to or from NULL.
1031                  */
1032                 if (dnh->dnh_dnode == NULL) {
1033                         zrl_destroy(&dnh->dnh_zrlock);
1034                         continue;
1035                 }

1037                 zrl_add(&dnh->dnh_zrlock);
1038                 dn = dnh->dnh_dnode;
1039                 /*
1040                  * If there are holds on this dnode, then there should
1041                  * be holds on the dnode's containing dbuf as well; thus
1042                  * it wouldn't be eligible for eviction and this function
1043                  * would not have been called.
1044                  */
1045                 ASSERT(refcount_is_zero(&dn->dn_holds));
1046                 ASSERT(refcount_is_zero(&dn->dn_tx_holds));

1048                 dnode_destroy(dn); /* implicit zrl_remove() */
1049                 zrl_destroy(&dnh->dnh_zrlock);
1050                 dnh->dnh_dnode = NULL;
1051         }
1052         kmem_free(children_dnodes, sizeof (dnode_children_t) +
1053             children_dnodes->dnc_count * sizeof (dnode_handle_t));
1025             epb * sizeof (dnode_handle_t));
1054 }

1056 /*
1057  * errors:
```



```
1058  * EINVAL - invalid object number.
1059  * EIO - i/o error.
1060  * succeeds even for free dnodes.
1061  */
1062 int
1063 dnode_hold_impl(objset_t *os, uint64_t object, int flag,
1064     void *tag, dnode_t **dnp)
1065 {
1066         int epb, idx, err;
1067         int drop_struct_lock = FALSE;
1068         int type;
1069         uint64_t blk;
1070         dnode_t *mdn, *dn;
1071         dmu_buf_impl_t *db;
1072         dnode_children_t *children_dnodes;
1073         dnode_handle_t *dnh;

1075         /*
1076          * If you are holding the spa config lock as writer, you shouldn't
1077          * be asking the DMU to do *anything* unless it's the root pool
1078          * which may require us to read from the root filesystem while
1079          * holding some (not all) of the locks as writer.
1080          */
1081         ASSERT(spa_config_held(os->os_spa, SCL_ALL, RW_WRITER) == 0 ||
1082             (spa_is_root(os->os_spa) &&
1083             spa_config_held(os->os_spa, SCL_STATE, RW_WRITER)));

1085         if (object == DMU_USERUSED_OBJECT || object == DMU_GROUPUSED_OBJECT) {
1086                 dn = (object == DMU_USERUSED_OBJECT) ?
1087                     DMU_USERUSED_DNODE(os) : DMU_GROUPUSED_DNODE(os);
1088                 if (dn == NULL)
1089                         return (SET_ERROR(ENOENT));
1090                 type = dn->dn_type;
1091                 if ((flag & DNODE_MUST_BE_ALLOCATED) && type == DMU_OT_NONE)
1092                         return (SET_ERROR(ENOENT));
1093                 if ((flag & DNODE_MUST_BE_FREE) && type != DMU_OT_NONE)
1094                         return (SET_ERROR(EEXIST));
1095                 DNODE_VERIFY(dn);
1096                 (void) refcount_add(&dn->dn_holds, tag);
1097                 *dnp = dn;
1098                 return (0);
1099         }

1101         if (object == 0 || object >= DN_MAX_OBJECT)
1102                 return (SET_ERROR(EINVAL));

1104         mdn = DMU_META_DNODE(os);
1105         ASSERT(mdn->dn_object == DMU_META_DNODE_OBJECT);

1107         DNODE_VERIFY(mdn);

1109         if (!RW_WRITE_HELD(&mdn->dn_struct_rwlock)) {
1110                 rw_enter(&mdn->dn_struct_rwlock, RW_READER);
1111                 drop_struct_lock = TRUE;
1112         }

1114         blk = dbuf_whichblock(mdn, object * sizeof (dnode_phys_t));

1116         db = dbuf_hold(mdn, blk, FTAG);
1117         if (drop_struct_lock)
1118                 rw_exit(&mdn->dn_struct_rwlock);
1119         if (db == NULL)
1120                 return (SET_ERROR(EIO));
1121         err = dbuf_read(db, NULL, DB_RF_CANFAIL);
1122         if (err) {
1123                 dbuf_rele(db, FTAG);
```

```
1124                return (err);
1125        }

1127        ASSERT3U(db->db.db_size, >=, 1<<DNODE_SHIFT);
1128        epb = db->db.db_size >> DNODE_SHIFT;

1130        idx = object & (epb-1);

1132        ASSERT(DB_DNODE(db)->dn_type == DMU_OT_DNODE);
1133        children_dnodes = dmu_buf_get_user(&db->db);
1134        if (children_dnodes == NULL) {
1135                int i;
1136                dnode_children_t *winner;
1137                children_dnodes = kmem_zalloc(sizeof (dnode_children_t) +
1109                children_dnodes = kmem_alloc(sizeof (dnode_children_t) +
1138                    epb * sizeof (dnode_handle_t), KM_SLEEP);
1139                children_dnodes->dnc_count = epb;
1140                dnh = &children_dnodes->dnc_children[0];
1141                for (i = 0; i < epb; i++) {
1142                        zrl_init(&dnh[i].dnh_zrlock);
1115                        dnh[i].dnh_dnode = NULL;
1143                }
1144                dmu_buf_init_user(&children_dnodes->dnc_dbu,
1145                    dnode_buf_pageout, NULL);
1146                winner = dmu_buf_set_user(&db->db, &children_dnodes->dnc_dbu);
1147                if (winner != NULL) {
1117                if (winner = dmu_buf_set_user(&db->db, children_dnodes,
1118                    dnode_buf_pageout)) {

1149                        for (i = 0; i < epb; i++) {
1150                                zrl_destroy(&dnh[i].dnh_zrlock);
1151                        }

1153                        kmem_free(children_dnodes, sizeof (dnode_children_t) +
1154                            epb * sizeof (dnode_handle_t));
1155                        children_dnodes = winner;
1156                }
1157        }
1158        ASSERT(children_dnodes->dnc_count == epb);

1160        dnh = &children_dnodes->dnc_children[idx];
1161        zrl_add(&dnh->dnh_zrlock);
1162        dn = dnh->dnh_dnode;
1163        if (dn == NULL) {
1133        if ((dn = dnh->dnh_dnode) == NULL) {
1164                dnode_phys_t *phys = (dnode_phys_t *)db->db.db_data+idx;
1135                dnode_t *winner;

1166                dn = dnode_create(os, phys, db, object, dnh);
1138                winner = atomic_cas_ptr(&dnh->dnh_dnode, NULL, dn);
1139                if (winner != NULL) {
1140                        zrl_add(&dnh->dnh_zrlock);
1141                        dnode_destroy(dn); /* implicit zrl_remove() */
1142                        dn = winner;
1167        }
1144        }

1169        mutex_enter(&dn->dn_mtx);
1170        type = dn->dn_type;
1171        if (dn->dn_free_txg ||
1172            ((flag & DNODE_MUST_BE_ALLOCATED) && type == DMU_OT_NONE) ||
1173            ((flag & DNODE_MUST_BE_FREE) &&
1174            (type != DMU_OT_NONE || !refcount_is_zero(&dn->dn_holds)))) {
1175                mutex_exit(&dn->dn_mtx);
1176                zrl_remove(&dnh->dnh_zrlock);
1177                dbuf_rele(db, FTAG);
```



```
1178                return (type == DMU_OT_NONE ? ENOENT : EEXIST);
1179        }
1180        if (refcount_add(&dn->dn_holds, tag) == 1)
1181                dbuf_add_ref(db, dnh);
1182        mutex_exit(&dn->dn_mtx);

1159        if (refcount_add(&dn->dn_holds, tag) == 1)
1160                dbuf_add_ref(db, dnh);
1184        /* Now we can rely on the hold to prevent the dnode from moving. */
1185        zrl_remove(&dnh->dnh_zrlock);

1187        DNODE_VERIFY(dn);
1188        ASSERT3P(dn->dn_dbuf, ==, db);
1189        ASSERT3U(dn->dn_object, ==, object);
1190        dbuf_rele(db, FTAG);

1192        *dnp = dn;
1193        return (0);
1194 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   19607 Tue Jan  6 10:37:47 2015
new/usr/src/uts/common/fs/zfs/dnode_sync.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/dbuf.h>
#include <sys/dnode.h>
#include <sys/dmu.h>
#include <sys/dmu_tx.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_dataset.h>
#include <sys/spa.h>
#include <sys/range_tree.h>
#include <sys/zfeature.h>

static void
dnode_increase_indirection(dnode_t *dn, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db;
        int txgoff = tx->tx_txg & TXG_MASK;
        int nblkptr = dn->dn_phys->dn_nblkptr;
        int old_toplvl = dn->dn_phys->dn_nlevels - 1;
        int new_level = dn->dn_next_nlevels[txgoff];
        int i;

        rw_enter(&dn->dn_struct_rwlock, RW_WRITER);

        /* this dnode can't be paged out because it's dirty */
        ASSERT(dn->dn_phys->dn_type != DMU_OT_NONE);
        ASSERT(RW_WRITE_HELD(&dn->dn_struct_rwlock));
        ASSERT(new_level > 1 && dn->dn_phys->dn_nlevels > 0);

        db = dbuf_hold_level(dn, dn->dn_phys->dn_nlevels, 0, FTAG);
        ASSERT(db != NULL);
```



```
        dn->dn_phys->dn_nlevels = new_level;
        dprintf("os=%p obj=%llu, increase to %d\n", dn->dn_objset,
            dn->dn_object, dn->dn_phys->dn_nlevels);

        /* check for existing blkptrs in the dnode */
        for (i = 0; i < nblkptr; i++)
                if (!BP_IS_HOLE(&dn->dn_phys->dn_blkptr[i]))
                        break;
        if (i != nblkptr) {
                /* transfer dnode's block pointers to new indirect block */
                (void) dbuf_read(db, NULL, DB_RF_MUST_SUCCEED|DB_RF_HAVESTRUCT);
                ASSERT(db->db.db_data);
                ASSERT(arc_released(db->db_buf));
                ASSERT3U(sizeof (blkptr_t) * nblkptr, <=, db->db.db_size);
                bcopy(dn->dn_phys->dn_blkptr, db->db.db_data,
                    sizeof (blkptr_t) * nblkptr);
                arc_buf_freeze(db->db_buf);
        }

        /* set dbuf's parent pointers to new indirect buf */
        for (i = 0; i < nblkptr; i++) {
                dmu_buf_impl_t *child = dbuf_find(dn, old_toplvl, i);

                if (child == NULL)
                        continue;
#ifdef  DEBUG
                DB_DNODE_ENTER(child);
                ASSERT3P(DB_DNODE(child), ==, dn);
                DB_DNODE_EXIT(child);
#endif  /* DEBUG */
                if (child->db_parent && child->db_parent != dn->dn_dbuf) {
                        ASSERT(child->db_parent->db_level == db->db_level);
                        ASSERT(child->db_blkptr !=
                            &dn->dn_phys->dn_blkptr[child->db_blkid]);
                        mutex_exit(&child->db_mtx);
                        continue;
                }
                ASSERT(child->db_parent == NULL ||
                    child->db_parent == dn->dn_dbuf);

                child->db_parent = db;
                dbuf_add_ref(db, child);
                if (db->db.db_data)
                        child->db_blkptr = (blkptr_t *)db->db.db_data + i;
                else
                        child->db_blkptr = NULL;
                dprintf_dbuf_bp(child, child->db_blkptr,
                    "changed db_blkptr to new indirect %s", "");

                mutex_exit(&child->db_mtx);
        }

        bzero(dn->dn_phys->dn_blkptr, sizeof (blkptr_t) * nblkptr);

        dbuf_rele(db, FTAG);

        rw_exit(&dn->dn_struct_rwlock);
}
```

*_____unchanged_portion_omitted_*

```
/*
 * Try to kick all the dnode's dbufs out of the cache...
 */
void
dnode_evict_dbufs(dnode_t *dn)
```

```
399 {
400         dmu_buf_impl_t db_marker;
399         int progress;
400         int pass = 0;

402         do {
401         dmu_buf_impl_t *db, *db_next;
404                 int evicting = FALSE;

406                 progress = FALSE;
403         mutex_enter(&dn->dn_dbufs_mtx);
404         for (db = avl_first(&dn->dn_dbufs); db != NULL; db = db_next) {

409                         db_next = AVL_NEXT(&dn->dn_dbufs, db);
406 #ifdef  DEBUG
407                 DB_DNODE_ENTER(db);
408                 ASSERT3P(DB_DNODE(db), ==, dn);
409                 DB_DNODE_EXIT(db);
410 #endif  /* DEBUG */

412                 mutex_enter(&db->db_mtx);
413                 if (db->db_state != DB_EVICTING &&
414                     refcount_is_zero(&db->db_holds)) {
415                         db_marker.db_level = db->db_level;
416                         db_marker.db_blkid = db->db_blkid;
417                         db_marker.db_state = DB_SEARCH;
418                         avl_insert_here(&dn->dn_dbufs, &db_marker, db,
419                             AVL_BEFORE);

421                         dbuf_clear(db);

423                         db_next = AVL_NEXT(&dn->dn_dbufs, &db_marker);
424                         avl_remove(&dn->dn_dbufs, &db_marker);
417                         if (db->db_state == DB_EVICTING) {
418                                 progress = TRUE;
419                                 evicting = TRUE;
420                                 mutex_exit(&db->db_mtx);
421                         } else if (refcount_is_zero(&db->db_holds)) {
422                                 progress = TRUE;
423                                 dbuf_clear(db); /* exits db_mtx for us */
425                 } else {
426                         mutex_exit(&db->db_mtx);
427                         db_next = AVL_NEXT(&dn->dn_dbufs, db);
428                 }

429         }
429                 /*
430                  * NB: we need to drop dn_dbufs_mtx between passes so
431                  * that any DB_EVICTING dbufs can make progress.
432                  * Ideally, we would have some cv we could wait on, but
433                  * since we don't, just wait a bit to give the other
434                  * thread a chance to run.
435                  */
430         mutex_exit(&dn->dn_dbufs_mtx);
437                 if (evicting)
438                         delay(1);
439                 pass++;
440                 ASSERT(pass < 100); /* sanity check */
441         } while (progress);

432         rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
433         if (dn->dn_bonus && refcount_is_zero(&dn->dn_bonus->db_holds)) {
434                 mutex_enter(&dn->dn_bonus->db_mtx);
435                 dbuf_evict(dn->dn_bonus);
436                 dn->dn_bonus = NULL;
437         }
```



```
438         rw_exit(&dn->dn_struct_rwlock);
439 }
______unchanged_portion_omitted_

472 static void
473 dnode_sync_free(dnode_t *dn, dmu_tx_t *tx)
474 {
475         int txgoff = tx->tx_txg & TXG_MASK;

477         ASSERT(dmu_tx_is_syncing(tx));

479         /*
480          * Our contents should have been freed in dnode_sync() by the
481          * free range record inserted by the caller of dnode_free().
482          */
483         ASSERT0(DN_USED_BYTES(dn->dn_phys));
484         ASSERT(BP_IS_HOLE(dn->dn_phys->dn_blkptr));

486         dnode_undirty_dbufs(&dn->dn_dirty_records[txgoff]);
487         dnode_evict_dbufs(dn);
488         ASSERT(avl_is_empty(&dn->dn_dbufs));
500         ASSERT3P(dn->dn_bonus, ==, NULL);

490         /*
491          * XXX - It would be nice to assert this, but we may still
492          * have residual holds from async evictions from the arc...
493          *
494          * zfs_obj_to_path() also depends on this being
495          * commented out.
496          *
497          * ASSERT3U(refcount_count(&dn->dn_holds), ==, 1);
498          */

500         /* Undirty next bits */
501         dn->dn_next_nlevels[txgoff] = 0;
502         dn->dn_next_indblkshift[txgoff] = 0;
503         dn->dn_next_blksz[txgoff] = 0;

505         /* ASSERT(blkptrs are zero); */
506         ASSERT(dn->dn_phys->dn_type != DMU_OT_NONE);
507         ASSERT(dn->dn_type != DMU_OT_NONE);

509         ASSERT(dn->dn_free_txg > 0);
510         if (dn->dn_allocated_txg != dn->dn_free_txg)
511                 dmu_buf_will_dirty(&dn->dn_dbuf->db, tx);
512         bzero(dn->dn_phys, sizeof (dnode_phys_t));

514         mutex_enter(&dn->dn_mtx);
515         dn->dn_type = DMU_OT_NONE;
516         dn->dn_maxblkid = 0;
517         dn->dn_allocated_txg = 0;
518         dn->dn_free_txg = 0;
519         dn->dn_have_spill = B_FALSE;
520         mutex_exit(&dn->dn_mtx);

522         ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);

524         dnode_rele(dn, (void *)(uintptr_t)tx->tx_txg);
525         /*
526          * Now that we've released our hold, the dnode may
527          * be evicted, so we musn't access it.
528          */
529 }
______unchanged_portion_omitted_
```

****
12029 Tue Jan 6 10:37:47 2015
new/usr/src/uts/common/fs/zfs/dsl_bookmark.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.com>
****

_____***unchanged_portion_omitted_***

```
 113 static int
 114 dsl_bookmark_create_check_impl(dsl_dataset_t *snapds, const char *bookmark_name,
 115     dmu_tx_t *tx)
 116 {
 117         dsl_pool_t *dp = dmu_tx_pool(tx);
 118         dsl_dataset_t *bmark_fs;
 119         char *shortname;
 120         int error;
 121         zfs_bookmark_phys_t bmark_phys;

 123         if (!snapds->ds_is_snapshot)
 123         if (!dsl_dataset_is_snapshot(snapds))
 124                 return (SET_ERROR(EINVAL));

 126         error = dsl_bookmark_hold_ds(dp, bookmark_name,
 127             &bmark_fs, FTAG, &shortname);
 128         if (error != 0)
 129                 return (error);

 131         if (!dsl_dataset_is_before(bmark_fs, snapds, 0)) {
 132                 dsl_dataset_rele(bmark_fs, FTAG);
 133                 return (SET_ERROR(EINVAL));
 134         }

 136         error = dsl_dataset_bmark_lookup(bmark_fs, shortname,
 137             &bmark_phys);
 138         dsl_dataset_rele(bmark_fs, FTAG);
 139         if (error == 0)
 140                 return (SET_ERROR(EEXIST));
 141         if (error == ESRCH)
 142                 return (0);
 143         return (error);
 144 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   94190 Tue Jan  6 10:37:47 2015
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2011, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2014, Joyent, Inc. All rights reserved.
 * Copyright (c) 2014 RackTop Systems.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#include <sys/dmu_objset.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/dmu_traverse.h>
#include <sys/dmu_impl.h>
#include <sys/dmu_tx.h>
#include <sys/arc.h>
#include <sys/zio.h>
#include <sys/zap.h>
#include <sys/zfeature.h>
#include <sys/unique.h>
#include <sys/zfs_context.h>
#include <sys/zfs_ioctl.h>
#include <sys/spa.h>
#include <sys/zfs_znode.h>
#include <sys/zfs_onexit.h>
#include <sys/zvol.h>
#include <sys/dsl_scan.h>
#include <sys/dsl_deadlist.h>
#include <sys/dsl_destroy.h>
#include <sys/dsl_userhold.h>
#include <sys/dsl_bookmark.h>

/*
 * The SPA supports block sizes up to 16MB.  However, very large blocks
 * can have an impact on i/o latency (e.g. tying up a spinning disk for
 * ~300ms), and also potentially on the memory allocator.  Therefore,
```



```
 * we do not allow the recordsize to be set larger than zfs_max_recordsize
 * (default 1MB).  Larger blocks can be created by changing this tunable,
 * and pools with larger blocks can always be imported and used, regardless
 * of this setting.
 */
int zfs_max_recordsize = 1 * 1024 * 1024;

#define SWITCH64(x, y) \
        { \
                uint64_t __tmp = (x); \
                (x) = (y); \
                (y) = __tmp; \
        }

#define DS_REF_MAX      (1ULL << 62)

extern inline dsl_dataset_phys_t *dsl_dataset_phys(dsl_dataset_t *ds);
extern inline boolean_t dsl_dataset_is_snapshot(dsl_dataset_t *ds);

/*
 * Figure out how much of this delta should be propogated to the dsl_dir
 * layer.  If there's a refreservation, that space has already been
 * partially accounted for in our ancestors.
 */
static int64_t
parent_delta(dsl_dataset_t *ds, int64_t delta)
{
        dsl_dataset_phys_t *ds_phys;
        uint64_t old_bytes, new_bytes;

        if (ds->ds_reserved == 0)
                return (delta);

        ds_phys = dsl_dataset_phys(ds);
        old_bytes = MAX(ds_phys->ds_unique_bytes, ds->ds_reserved);
        new_bytes = MAX(ds_phys->ds_unique_bytes + delta, ds->ds_reserved);

        ASSERT3U(ABS((int64_t)(new_bytes - old_bytes)), <=, ABS(delta));
        return (new_bytes - old_bytes);
}
_____unchanged_portion_omitted_

int
dsl_dataset_block_kill(dsl_dataset_t *ds, const blkptr_t *bp, dmu_tx_t *tx,
    boolean_t async)
{
        int used = bp_get_dsize_sync(tx->tx_pool->dp_spa, bp);
        int compressed = BP_GET_PSIZE(bp);
        int uncompressed = BP_GET_UCSIZE(bp);

        if (BP_IS_HOLE(bp))
                return (0);

        ASSERT(dmu_tx_is_syncing(tx));
        ASSERT(bp->blk_birth <= tx->tx_txg);

        if (ds == NULL) {
                dsl_free(tx->tx_pool, tx->tx_txg, bp);
                dsl_pool_mos_diduse_space(tx->tx_pool,
                    -used, -compressed, -uncompressed);
                return (used);
        }
        ASSERT3P(tx->tx_pool, ==, ds->ds_dir->dd_pool);

        ASSERT(!ds->ds_is_snapshot);
        ASSERT(!dsl_dataset_is_snapshot(ds));
```

```
159         dmu_buf_will_dirty(ds->ds_dbuf, tx);
160
161         if (bp->blk_birth > dsl_dataset_phys(ds)->ds_prev_snap_txg) {
162                 int64_t delta;
163
164                 dprintf_bp(bp, "freeing ds=%llu", ds->ds_object);
165                 dsl_free(tx->tx_pool, tx->tx_txg, bp);
166
167                 mutex_enter(&ds->ds_lock);
168                 ASSERT(dsl_dataset_phys(ds)->ds_unique_bytes >= used ||
169                     !DS_UNIQUE_IS_ACCURATE(ds));
170                 delta = parent_delta(ds, -used);
171                 dsl_dataset_phys(ds)->ds_unique_bytes -= used;
172                 mutex_exit(&ds->ds_lock);
173                 dsl_dir_diduse_space(ds->ds_dir, DD_USED_HEAD,
174                     delta, -compressed, -uncompressed, tx);
175                 dsl_dir_transfer_space(ds->ds_dir, -used - delta,
176                     DD_USED_REFRSRV, DD_USED_HEAD, tx);
177         } else {
178                 dprintf_bp(bp, "putting on dead list: %s", "");
179                 if (async) {
180                         /*
181                          * We are here as part of zio's write done callback,
182                          * which means we're a zio interrupt thread.  We can't
183                          * call dsl_deadlist_insert() now because it may block
184                          * waiting for I/O.  Instead, put bp on the deferred
185                          * queue and let dsl_pool_sync() finish the job.
186                          */
187                         bplist_append(&ds->ds_pending_deadlist, bp);
188                 } else {
189                         dsl_deadlist_insert(&ds->ds_deadlist, bp, tx);
190                 }
191                 ASSERT3U(ds->ds_prev->ds_object, ==,
192                     dsl_dataset_phys(ds)->ds_prev_snap_obj);
193                 ASSERT(dsl_dataset_phys(ds->ds_prev)->ds_num_children > 0);
194                 /* if (bp->blk_birth > prev prev snap txg) prev unique += bs */
195                 if (dsl_dataset_phys(ds->ds_prev)->ds_next_snap_obj ==
196                     ds->ds_object && bp->blk_birth >
197                     dsl_dataset_phys(ds->ds_prev)->ds_prev_snap_txg) {
198                         dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
199                         mutex_enter(&ds->ds_prev->ds_lock);
200                         dsl_dataset_phys(ds->ds_prev)->ds_unique_bytes += used;
201                         mutex_exit(&ds->ds_prev->ds_lock);
202                 }
203                 if (bp->blk_birth > ds->ds_dir->dd_origin_txg) {
204                         dsl_dir_transfer_space(ds->ds_dir, used,
205                             DD_USED_HEAD, DD_USED_SNAP, tx);
206                 }
207         }
208         mutex_enter(&ds->ds_lock);
209         ASSERT3U(dsl_dataset_phys(ds)->ds_referenced_bytes, >=, used);
210         dsl_dataset_phys(ds)->ds_referenced_bytes -= used;
211         ASSERT3U(dsl_dataset_phys(ds)->ds_compressed_bytes, >=, compressed);
212         dsl_dataset_phys(ds)->ds_compressed_bytes -= compressed;
213         ASSERT3U(dsl_dataset_phys(ds)->ds_uncompressed_bytes, >=, uncompressed);
214         dsl_dataset_phys(ds)->ds_uncompressed_bytes -= uncompressed;
215         mutex_exit(&ds->ds_lock);
216
217         return (used);
218 }
```

*unchanged_portion_omitted_*

```
256 /* ARGSUSED */
256 static void
257 dsl_dataset_evict(void *dbu)
258 dsl_dataset_evict(dmu_buf_t *db, void *dsv)
```



```
258 {
259         dsl_dataset_t *ds = dbu;
260         dsl_dataset_t *ds = dsv;
261         ASSERT(ds->ds_owner == NULL);
263         ds->ds_dbuf = NULL;
265         unique_remove(ds->ds_fsid_guid);
267         if (ds->ds_objset != NULL)
268                 dmu_objset_evict(ds->ds_objset);
270         if (ds->ds_prev) {
271                 dsl_dataset_rele(ds->ds_prev, ds);
272                 ds->ds_prev = NULL;
273         }
275         bplist_destroy(&ds->ds_pending_deadlist);
276         if (ds->ds_deadlist.dl_os != NULL)
275         if (dsl_dataset_phys(ds)->ds_deadlist_obj != 0)
277                 dsl_deadlist_close(&ds->ds_deadlist);
278         if (ds->ds_dir)
279                 dsl_dir_async_rele(ds->ds_dir, ds);
278                 dsl_dir_rele(ds->ds_dir, ds);
281         ASSERT(!list_link_active(&ds->ds_synced_link));
283         mutex_destroy(&ds->ds_lock);
284         mutex_destroy(&ds->ds_opening_lock);
285         mutex_destroy(&ds->ds_sendstream_lock);
286         refcount_destroy(&ds->ds_longholds);
288         kmem_free(ds, sizeof (dsl_dataset_t));
289 }
```

*unchanged_portion_omitted_*

```
363 int
364 dsl_dataset_hold_obj(dsl_pool_t *dp, uint64_t dsobj, void *tag,
365     dsl_dataset_t **dsp)
366 {
367         objset_t *mos = dp->dp_meta_objset;
368         dmu_buf_t *dbuf;
369         dsl_dataset_t *ds;
370         int err;
371         dmu_object_info_t doi;
373         ASSERT(dsl_pool_config_held(dp));
375         err = dmu_bonus_hold(mos, dsobj, tag, &dbuf);
376         if (err != 0)
377                 return (err);
379         /* Make sure dsobj has the correct object type. */
380         dmu_object_info_from_db(dbuf, &doi);
381         if (doi.doi_bonus_type != DMU_OT_DSL_DATASET) {
382                 dmu_buf_rele(dbuf, tag);
383                 return (SET_ERROR(EINVAL));
384         }
386         ds = dmu_buf_get_user(dbuf);
387         if (ds == NULL) {
388                 dsl_dataset_t *winner = NULL;
390                 ds = kmem_zalloc(sizeof (dsl_dataset_t), KM_SLEEP);
391                 ds->ds_dbuf = dbuf;
```

```
392                 ds->ds_object = dsobj;
393                 ds->ds_is_snapshot = dsl_dataset_phys(ds)->ds_num_children != 0;

395                 mutex_init(&ds->ds_lock, NULL, MUTEX_DEFAULT, NULL);
396                 mutex_init(&ds->ds_opening_lock, NULL, MUTEX_DEFAULT, NULL);
397                 mutex_init(&ds->ds_sendstream_lock, NULL, MUTEX_DEFAULT, NULL);
398                 refcount_create(&ds->ds_longholds);

400                 bplist_create(&ds->ds_pending_deadlist);
401                 dsl_deadlist_open(&ds->ds_deadlist,
402                     mos, dsl_dataset_phys(ds)->ds_deadlist_obj);

404                 list_create(&ds->ds_sendstreams, sizeof (dmu_sendarg_t),
405                     offsetof(dmu_sendarg_t, dsa_link));

407                 if (doi.doi_type == DMU_OTN_ZAP_METADATA) {
408                         err = zap_contains(mos, dsobj, DS_FIELD_LARGE_BLOCKS);
409                         if (err == 0)
410                                 ds->ds_large_blocks = B_TRUE;
411                         else
412                                 ASSERT3U(err, ==, ENOENT);
413                 }

415                 if (err == 0) {
416                         err = dsl_dir_hold_obj(dp,
417                             dsl_dataset_phys(ds)->ds_dir_obj, NULL, ds,
418                             &ds->ds_dir);
419                 }
420                 if (err != 0) {
421                         mutex_destroy(&ds->ds_lock);
422                         mutex_destroy(&ds->ds_opening_lock);
423                         mutex_destroy(&ds->ds_sendstream_lock);
424                         refcount_destroy(&ds->ds_longholds);
425                         bplist_destroy(&ds->ds_pending_deadlist);
426                         dsl_deadlist_close(&ds->ds_deadlist);
427                         kmem_free(ds, sizeof (dsl_dataset_t));
428                         dmu_buf_rele(dbuf, tag);
429                         return (err);
430                 }

432                 if (!ds->ds_is_snapshot) {
430                 if (!dsl_dataset_is_snapshot(ds)) {
433                         ds->ds_snapname[0] = '\0';
434                         if (dsl_dataset_phys(ds)->ds_prev_snap_obj != 0) {
435                                 err = dsl_dataset_hold_obj(dp,
436                                     dsl_dataset_phys(ds)->ds_prev_snap_obj,
437                                     ds, &ds->ds_prev);
438                         }
439                         if (doi.doi_type == DMU_OTN_ZAP_METADATA) {
440                                 int zaperr = zap_lookup(mos, ds->ds_object,
441                                     DS_FIELD_BOOKMARK_NAMES,
442                                     sizeof (ds->ds_bookmarks), 1,
443                                     &ds->ds_bookmarks);
444                                 if (zaperr != ENOENT)
445                                         VERIFY0(zaperr);
446                         }
447                 } else {
448                         if (zfs_flags & ZFS_DEBUG_SNAPNAMES)
449                                 err = dsl_dataset_get_snapname(ds);
450                         if (err == 0 &&
451                             dsl_dataset_phys(ds)->ds_userrefs_obj != 0) {
452                                 err = zap_count(
453                                     ds->ds_dir->dd_pool->dp_meta_objset,
454                                     dsl_dataset_phys(ds)->ds_userrefs_obj,
455                                     &ds->ds_userrefs);
456                         }
```



```
457                 }

459                 if (err == 0 && !ds->ds_is_snapshot) {
457                 if (err == 0 && !dsl_dataset_is_snapshot(ds)) {
460                         err = dsl_prop_get_int_ds(ds,
461                             zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
462                             &ds->ds_reserved);
463                         if (err == 0) {
464                                 err = dsl_prop_get_int_ds(ds,
465                                     zfs_prop_to_name(ZFS_PROP_REFQUOTA),
466                                     &ds->ds_quota);
467                         }
468                 } else {
469                         ds->ds_reserved = ds->ds_quota = 0;
470                 }

472                 dmu_buf_init_user(&ds->ds_dbu, dsl_dataset_evict, &ds->ds_dbuf);
473                 if (err == 0)
474                         winner = dmu_buf_set_user_ie(dbuf, &ds->ds_dbu);

476                 if (err != 0 || winner != NULL) {
470                 if (err != 0 || (winner = dmu_buf_set_user_ie(dbuf, ds,
471                     dsl_dataset_evict)) != NULL) {
477                         bplist_destroy(&ds->ds_pending_deadlist);
478                         dsl_deadlist_close(&ds->ds_deadlist);
479                         if (ds->ds_prev)
480                                 dsl_dataset_rele(ds->ds_prev, ds);
481                         dsl_dir_rele(ds->ds_dir, ds);
482                         mutex_destroy(&ds->ds_lock);
483                         mutex_destroy(&ds->ds_opening_lock);
484                         mutex_destroy(&ds->ds_sendstream_lock);
485                         refcount_destroy(&ds->ds_longholds);
486                         kmem_free(ds, sizeof (dsl_dataset_t));
487                         if (err != 0) {
488                                 dmu_buf_rele(dbuf, tag);
489                                 return (err);
490                         }
491                         ds = winner;
492                 } else {
493                         ds->ds_fsid_guid =
494                             unique_insert(dsl_dataset_phys(ds)->ds_fsid_guid);
495                 }
496         }
497         ASSERT3P(ds->ds_dbuf, ==, dbuf);
498         ASSERT3P(dsl_dataset_phys(ds), ==, dbuf->db_data);
499         ASSERT(dsl_dataset_phys(ds)->ds_prev_snap_obj != 0 ||
500             spa_version(dp->dp_spa) < SPA_VERSION_ORIGIN ||
501             dp->dp_origin_snap == NULL || ds == dp->dp_origin_snap);
502         *dsp = ds;
503         return (0);
504 }
```

*unchanged_portion_omitted_*

```
842 /*
843  * The unique space in the head dataset can be calculated by subtracting
844  * the space used in the most recent snapshot, that is still being used
845  * in this file system, from the space currently in use.  To figure out
846  * the space in the most recent snapshot still in use, we need to take
847  * the total space used in the snapshot and subtract out the space that
848  * has been freed up since the snapshot was taken.
849  */
850 void
851 dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds)
852 {
853         uint64_t mrs_used;
854         uint64_t dlused, dlcomp, dluncomp;
```

```
 856         ASSERT(!ds->ds_is_snapshot);
 851         ASSERT(!dsl_dataset_is_snapshot(ds));

 858         if (dsl_dataset_phys(ds)->ds_prev_snap_obj != 0)
 859                 mrs_used = dsl_dataset_phys(ds->ds_prev)->ds_referenced_bytes;
 860         else
 861                 mrs_used = 0;

 863         dsl_deadlist_space(&ds->ds_deadlist, &dlused, &dlcomp, &dluncomp);

 865         ASSERT3U(dlused, <=, mrs_used);
 866         dsl_dataset_phys(ds)->ds_unique_bytes =
 867             dsl_dataset_phys(ds)->ds_referenced_bytes - (mrs_used - dlused);

 869         if (spa_version(ds->ds_dir->dd_pool->dp_spa) >=
 870             SPA_VERSION_UNIQUE_ACCURATE)
 871                 dsl_dataset_phys(ds)->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;
 872 }
_____unchanged_portion_omitted_

1583 void
1584 dsl_dataset_stats(dsl_dataset_t *ds, nvlist_t *nv)
1585 {
1586         dsl_pool_t *dp = ds->ds_dir->dd_pool;
1587         uint64_t refd, avail, uobjs, aobjs, ratio;

1589         ASSERT(dsl_pool_config_held(dp));

1591         ratio = dsl_dataset_phys(ds)->ds_compressed_bytes == 0 ? 100 :
1592             (dsl_dataset_phys(ds)->ds_uncompressed_bytes * 100 /
1593             dsl_dataset_phys(ds)->ds_compressed_bytes);

1595         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFRATIO, ratio);
1596         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_LOGICALREFERENCED,
1597             dsl_dataset_phys(ds)->ds_uncompressed_bytes);

1599         if (ds->ds_is_snapshot) {
1594         if (dsl_dataset_is_snapshot(ds)) {
1600                 dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_COMPRESSRATIO, ratio);
1601                 dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USED,
1602                     dsl_dataset_phys(ds)->ds_unique_bytes);
1603                 get_clones_stat(ds, nv);
1604         } else {
1605                 if (ds->ds_prev != NULL && ds->ds_prev != dp->dp_origin_snap) {
1606                         char buf[MAXNAMELEN];
1607                         dsl_dataset_name(ds->ds_prev, buf);
1608                         dsl_prop_nvlist_add_string(nv, ZFS_PROP_PREV_SNAP, buf);
1609                 }

1611                 dsl_dir_stats(ds->ds_dir, nv);
1612         }

1614         dsl_dataset_space(ds, &refd, &avail, &uobjs, &aobjs);
1615         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_AVAILABLE, avail);
1616         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFERENCED, refd);

1618         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_CREATION,
1619             dsl_dataset_phys(ds)->ds_creation_time);
1620         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_CREATETXG,
1621             dsl_dataset_phys(ds)->ds_creation_txg);
1622         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFQUOTA,
1623             ds->ds_quota);
1624         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_REFRESERVATION,
1625             ds->ds_reserved);
1626         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_GUID,
```



```
1627             dsl_dataset_phys(ds)->ds_guid);
1628         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_UNIQUE,
1629             dsl_dataset_phys(ds)->ds_unique_bytes);
1630         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_OBJSETID,
1631             ds->ds_object);
1632         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USERREFS,
1633             ds->ds_userrefs);
1634         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_DEFER_DESTROY,
1635             DS_IS_DEFER_DESTROY(ds) ? 1 : 0);

1637         if (dsl_dataset_phys(ds)->ds_prev_snap_obj != 0) {
1638                 uint64_t written, comp, uncomp;
1639                 dsl_pool_t *dp = ds->ds_dir->dd_pool;
1640                 dsl_dataset_t *prev;

1642                 int err = dsl_dataset_hold_obj(dp,
1643                     dsl_dataset_phys(ds)->ds_prev_snap_obj, FTAG, &prev);
1644                 if (err == 0) {
1645                         err = dsl_dataset_space_written(prev, ds, &written,
1646                             &comp, &uncomp);
1647                         dsl_dataset_rele(prev, FTAG);
1648                         if (err == 0) {
1649                                 dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_WRITTEN,
1650                                     written);
1651                         }
1652                 }
1653         }
1654 }

1656 void
1657 dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat)
1658 {
1659         dsl_pool_t *dp = ds->ds_dir->dd_pool;
1660         ASSERT(dsl_pool_config_held(dp));

1662         stat->dds_creation_txg = dsl_dataset_phys(ds)->ds_creation_txg;
1663         stat->dds_inconsistent =
1664             dsl_dataset_phys(ds)->ds_flags & DS_FLAG_INCONSISTENT;
1665         stat->dds_guid = dsl_dataset_phys(ds)->ds_guid;
1666         stat->dds_origin[0] = '\0';
1667         if (ds->ds_is_snapshot) {
1662         if (dsl_dataset_is_snapshot(ds)) {
1668                 stat->dds_is_snapshot = B_TRUE;
1669                 stat->dds_num_clones =
1670                     dsl_dataset_phys(ds)->ds_num_children - 1;
1671         } else {
1672                 stat->dds_is_snapshot = B_FALSE;
1673                 stat->dds_num_clones = 0;

1675                 if (dsl_dir_is_clone(ds->ds_dir)) {
1676                         dsl_dataset_t *ods;

1678                         VERIFY0(dsl_dataset_hold_obj(dp,
1679                             dsl_dir_phys(ds->ds_dir)->dd_origin_obj,
1680                             FTAG, &ods));
1681                         dsl_dataset_name(ods, stat->dds_origin);
1682                         dsl_dataset_rele(ods, FTAG);
1683                 }
1684         }
1685 }
_____unchanged_portion_omitted_

1913 static int
1914 dsl_dataset_rollback_check(void *arg, dmu_tx_t *tx)
1915 {
1916         dsl_dataset_rollback_arg_t *ddra = arg;
```

```
1917         dsl_pool_t *dp = dmu_tx_pool(tx);
1918         dsl_dataset_t *ds;
1919         int64_t unused_refres_delta;
1920         int error;

1922         error = dsl_dataset_hold(dp, ddra->ddra_fsname, FTAG, &ds);
1923         if (error != 0)
1924                 return (error);

1926         /* must not be a snapshot */
1927         if (ds->ds_is_snapshot) {
1922         if (dsl_dataset_is_snapshot(ds)) {
1928                 dsl_dataset_rele(ds, FTAG);
1929                 return (SET_ERROR(EINVAL));
1930         }

1932         /* must have a most recent snapshot */
1933         if (dsl_dataset_phys(ds)->ds_prev_snap_txg < TXG_INITIAL) {
1934                 dsl_dataset_rele(ds, FTAG);
1935                 return (SET_ERROR(EINVAL));
1936         }

1938         /* must not have any bookmarks after the most recent snapshot */
1939         nvlist_t *proprequest = fnvlist_alloc();
1940         fnvlist_add_boolean(proprequest, zfs_prop_to_name(ZFS_PROP_CREATETXG));
1941         nvlist_t *bookmarks = fnvlist_alloc();
1942         error = dsl_get_bookmarks_impl(ds, proprequest, bookmarks);
1943         fnvlist_free(proprequest);
1944         if (error != 0)
1945                 return (error);
1946         for (nvpair_t *pair = nvlist_next_nvpair(bookmarks, NULL);
1947             pair != NULL; pair = nvlist_next_nvpair(bookmarks, pair)) {
1948                 nvlist_t *valuenv =
1949                     fnvlist_lookup_nvlist(fnvpair_value_nvlist(pair),
1950                     zfs_prop_to_name(ZFS_PROP_CREATETXG));
1951                 uint64_t createtxg = fnvlist_lookup_uint64(valuenv, "value");
1952                 if (createtxg > dsl_dataset_phys(ds)->ds_prev_snap_txg) {
1953                         fnvlist_free(bookmarks);
1954                         dsl_dataset_rele(ds, FTAG);
1955                         return (SET_ERROR(EEXIST));
1956                 }
1957         }
1958         fnvlist_free(bookmarks);

1960         error = dsl_dataset_handoff_check(ds, ddra->ddra_owner, tx);
1961         if (error != 0) {
1962                 dsl_dataset_rele(ds, FTAG);
1963                 return (error);
1964         }

1966         /*
1967          * Check if the snap we are rolling back to uses more than
1968          * the refquota.
1969          */
1970         if (ds->ds_quota != 0 &&
1971             dsl_dataset_phys(ds->ds_prev)->ds_referenced_bytes > ds->ds_quota) {
1972                 dsl_dataset_rele(ds, FTAG);
1973                 return (SET_ERROR(EDQUOT));
1974         }

1976         /*
1977          * When we do the clone swap, we will temporarily use more space
1978          * due to the refreservation (the head will no longer have any
1979          * unique space, so the entire amount of the refreservation will need
1980          * to be free).  We will immediately destroy the clone, freeing
1981          * this space, but the freeing happens over many txg's.
```



```
1982          */
1983         unused_refres_delta = (int64_t)MIN(ds->ds_reserved,
1984             dsl_dataset_phys(ds)->ds_unique_bytes);

1986         if (unused_refres_delta > 0 &&
1987             unused_refres_delta >
1988             dsl_dir_space_available(ds->ds_dir, NULL, 0, TRUE)) {
1989                 dsl_dataset_rele(ds, FTAG);
1990                 return (SET_ERROR(ENOSPC));
1991         }

1993         dsl_dataset_rele(ds, FTAG);
1994         return (0);
1995 }
_____unchanged_portion_omitted_

2486 static int
2487 promote_hold(dsl_dataset_promote_arg_t *ddpa, dsl_pool_t *dp, void *tag)
2488 {
2489         int error;
2490         dsl_dir_t *dd;
2491         struct promotenode *snap;

2493         error = dsl_dataset_hold(dp, ddpa->ddpa_clonename, tag,
2494             &ddpa->ddpa_clone);
2495         if (error != 0)
2496                 return (error);
2497         dd = ddpa->ddpa_clone->ds_dir;

2499         if (ddpa->ddpa_clone->ds_is_snapshot ||
2494         if (dsl_dataset_is_snapshot(ddpa->ddpa_clone) ||
2500             !dsl_dir_is_clone(dd)) {
2501                 dsl_dataset_rele(ddpa->ddpa_clone, tag);
2502                 return (SET_ERROR(EINVAL));
2503         }

2505         error = snaplist_make(dp, 0, dsl_dir_phys(dd)->dd_origin_obj,
2506             &ddpa->shared_snaps, tag);
2507         if (error != 0)
2508                 goto out;

2510         error = snaplist_make(dp, 0, ddpa->ddpa_clone->ds_object,
2511             &ddpa->clone_snaps, tag);
2512         if (error != 0)
2513                 goto out;

2515         snap = list_head(&ddpa->shared_snaps);
2516         ASSERT3U(snap->ds->ds_object, ==, dsl_dir_phys(dd)->dd_origin_obj);
2517         error = snaplist_make(dp, dsl_dir_phys(dd)->dd_origin_obj,
2518             dsl_dir_phys(snap->ds->ds_dir)->dd_head_dataset_obj,
2519             &ddpa->origin_snaps, tag);
2520         if (error != 0)
2521                 goto out;

2523         if (dsl_dir_phys(snap->ds->ds_dir)->dd_origin_obj != 0) {
2524                 error = dsl_dataset_hold_obj(dp,
2525                     dsl_dir_phys(snap->ds->ds_dir)->dd_origin_obj,
2526                     tag, &ddpa->origin_origin);
2527                 if (error != 0)
2528                         goto out;
2529         }
2530 out:
2531         if (error != 0)
2532                 promote_rele(ddpa, tag);
2533         return (error);
2534 }
_____unchanged_portion_omitted_
```

```
2584 int
2585 dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
2586     dsl_dataset_t *origin_head, boolean_t force, void *owner, dmu_tx_t *tx)
2587 {
2588         int64_t unused_refres_delta;

2590         /* they should both be heads */
2591         if (clone->ds_is_snapshot ||
2592             origin_head->ds_is_snapshot)
2586         if (dsl_dataset_is_snapshot(clone) ||
2587             dsl_dataset_is_snapshot(origin_head))
2593                 return (SET_ERROR(EINVAL));

2595         /* if we are not forcing, the branch point should be just before them */
2596         if (!force && clone->ds_prev != origin_head->ds_prev)
2597                 return (SET_ERROR(EINVAL));

2599         /* clone should be the clone (unless they are unrelated) */
2600         if (clone->ds_prev != NULL &&
2601             clone->ds_prev != clone->ds_dir->dd_pool->dp_origin_snap &&
2602             origin_head->ds_dir != clone->ds_prev->ds_dir)
2603                 return (SET_ERROR(EINVAL));

2605         /* the clone should be a child of the origin */
2606         if (clone->ds_dir->dd_parent != origin_head->ds_dir)
2607                 return (SET_ERROR(EINVAL));

2609         /* origin_head shouldn't be modified unless 'force' */
2610         if (!force &&
2611             dsl_dataset_modified_since_snap(origin_head, origin_head->ds_prev))
2612                 return (SET_ERROR(ETXTBSY));

2614         /* origin_head should have no long holds (e.g. is not mounted) */
2615         if (dsl_dataset_handoff_check(origin_head, owner, tx))
2616                 return (SET_ERROR(EBUSY));

2618         /* check amount of any unconsumed refreservation */
2619         unused_refres_delta =
2620             (int64_t)MIN(origin_head->ds_reserved,
2621             dsl_dataset_phys(origin_head)->ds_unique_bytes) -
2622             (int64_t)MIN(origin_head->ds_reserved,
2623             dsl_dataset_phys(clone)->ds_unique_bytes);

2625         if (unused_refres_delta > 0 &&
2626             unused_refres_delta >
2627             dsl_dir_space_available(origin_head->ds_dir, NULL, 0, TRUE))
2628                 return (SET_ERROR(ENOSPC));

2630         /* clone can't be over the head's refquota */
2631         if (origin_head->ds_quota != 0 &&
2632             dsl_dataset_phys(clone)->ds_referenced_bytes >
2633             origin_head->ds_quota)
2634                 return (SET_ERROR(EDQUOT));

2636         return (0);
2637 }
```

_____***unchanged_portion_omitted_***

```
2854 /* ARGSUSED */
2855 static int
2856 dsl_dataset_set_refquota_check(void *arg, dmu_tx_t *tx)
2857 {
2858         dsl_dataset_set_qr_arg_t *ddsqra = arg;
2859         dsl_pool_t *dp = dmu_tx_pool(tx);
```



```
2860         dsl_dataset_t *ds;
2861         int error;
2862         uint64_t newval;

2864         if (spa_version(dp->dp_spa) < SPA_VERSION_REFQUOTA)
2865                 return (SET_ERROR(ENOTSUP));

2867         error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
2868         if (error != 0)
2869                 return (error);

2871         if (ds->ds_is_snapshot) {
2866         if (dsl_dataset_is_snapshot(ds)) {
2872                 dsl_dataset_rele(ds, FTAG);
2873                 return (SET_ERROR(EINVAL));
2874         }

2876         error = dsl_prop_predict(ds->ds_dir,
2877             zfs_prop_to_name(ZFS_PROP_REFQUOTA),
2878             ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
2879         if (error != 0) {
2880                 dsl_dataset_rele(ds, FTAG);
2881                 return (error);
2882         }

2884         if (newval == 0) {
2885                 dsl_dataset_rele(ds, FTAG);
2886                 return (0);
2887         }

2889         if (newval < dsl_dataset_phys(ds)->ds_referenced_bytes ||
2890             newval < ds->ds_reserved) {
2891                 dsl_dataset_rele(ds, FTAG);
2892                 return (SET_ERROR(ENOSPC));
2893         }

2895         dsl_dataset_rele(ds, FTAG);
2896         return (0);
2897 }
```

_____***unchanged_portion_omitted_***

```
2938 static int
2939 dsl_dataset_set_refreservation_check(void *arg, dmu_tx_t *tx)
2940 {
2941         dsl_dataset_set_qr_arg_t *ddsqra = arg;
2942         dsl_pool_t *dp = dmu_tx_pool(tx);
2943         dsl_dataset_t *ds;
2944         int error;
2945         uint64_t newval, unique;

2947         if (spa_version(dp->dp_spa) < SPA_VERSION_REFRESERVATION)
2948                 return (SET_ERROR(ENOTSUP));

2950         error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
2951         if (error != 0)
2952                 return (error);

2954         if (ds->ds_is_snapshot) {
2949         if (dsl_dataset_is_snapshot(ds)) {
2955                 dsl_dataset_rele(ds, FTAG);
2956                 return (SET_ERROR(EINVAL));
2957         }

2959         error = dsl_prop_predict(ds->ds_dir,
2960             zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
2961             ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
```

```
2962         if (error != 0) {
2963                 dsl_dataset_rele(ds, FTAG);
2964                 return (error);
2965         }

2967         /*
2968          * If we are doing the preliminary check in open context, the
2969          * space estimates may be inaccurate.
2970          */
2971         if (!dmu_tx_is_syncing(tx)) {
2972                 dsl_dataset_rele(ds, FTAG);
2973                 return (0);
2974         }

2976         mutex_enter(&ds->ds_lock);
2977         if (!DS_UNIQUE_IS_ACCURATE(ds))
2978                 dsl_dataset_recalc_head_uniq(ds);
2979         unique = dsl_dataset_phys(ds)->ds_unique_bytes;
2980         mutex_exit(&ds->ds_lock);

2982         if (MAX(unique, newval) > MAX(unique, ds->ds_reserved)) {
2983                 uint64_t delta = MAX(unique, newval) -
2984                     MAX(unique, ds->ds_reserved);

2986                 if (delta >
2987                     dsl_dir_space_available(ds->ds_dir, NULL, 0, B_TRUE) ||
2988                     (ds->ds_quota > 0 && newval > ds->ds_quota)) {
2989                         dsl_dataset_rele(ds, FTAG);
2990                         return (SET_ERROR(ENOSPC));
2991                 }
2992         }

2994         dsl_dataset_rele(ds, FTAG);
2995         return (0);
2996 }
```

*unchanged_portion_omitted_*

```
3144 /*
3145  * Return (in *usedp) the amount of space that will be reclaimed if firstsnap,
3146  * lastsnap, and all snapshots in between are deleted.
3147  *
3148  * blocks that would be freed            [---------------------------]
3149  * snapshots                       ---O-------O--------O-------O--------O
3150  *                                          firstsnap        lastsnap
3151  *
3152  * This is the set of blocks that were born after the snap before firstsnap,
3153  * (birth > firstsnap->prev_snap_txg) and died before the snap after the
3154  * last snap (ie, is on lastsnap->ds_next->ds_deadlist or an earlier deadlist).
3155  * We calculate this by iterating over the relevant deadlists (from the snap
3156  * after lastsnap, backward to the snap after firstsnap), summing up the
3157  * space on the deadlist that was born after the snap before firstsnap.
3158  */
3159 int
3160 dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap,
3161     dsl_dataset_t *lastsnap,
3162     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
3163 {
3164         int err = 0;
3165         uint64_t snapobj;
3166         dsl_pool_t *dp = firstsnap->ds_dir->dd_pool;

3168         ASSERT(firstsnap->ds_is_snapshot);
3169         ASSERT(lastsnap->ds_is_snapshot);
3163         ASSERT(dsl_dataset_is_snapshot(firstsnap));
3164         ASSERT(dsl_dataset_is_snapshot(lastsnap));
```



```
3171         /*
3172          * Check that the snapshots are in the same dsl_dir, and firstsnap
3173          * is before lastsnap.
3174          */
3175         if (firstsnap->ds_dir != lastsnap->ds_dir ||
3176             dsl_dataset_phys(firstsnap)->ds_creation_txg >
3177             dsl_dataset_phys(lastsnap)->ds_creation_txg)
3178                 return (SET_ERROR(EINVAL));

3180         *usedp = *compp = *uncompp = 0;

3182         snapobj = dsl_dataset_phys(lastsnap)->ds_next_snap_obj;
3183         while (snapobj != firstsnap->ds_object) {
3184                 dsl_dataset_t *ds;
3185                 uint64_t used, comp, uncomp;

3187                 err = dsl_dataset_hold_obj(dp, snapobj, FTAG, &ds);
3188                 if (err != 0)
3189                         break;

3191                 dsl_deadlist_space_range(&ds->ds_deadlist,
3192                     dsl_dataset_phys(firstsnap)->ds_prev_snap_txg, UINT64_MAX,
3193                     &used, &comp, &uncomp);
3194                 *usedp += used;
3195                 *compp += comp;
3196                 *uncompp += uncomp;

3198                 snapobj = dsl_dataset_phys(ds)->ds_prev_snap_obj;
3199                 ASSERT3U(snapobj, !=, 0);
3200                 dsl_dataset_rele(ds, FTAG);
3201         }
3202         return (err);
3203 }
```

*unchanged_portion_omitted_*

```
3276 /*
3277  * Return TRUE if 'earlier' is an earlier snapshot in 'later's timeline.
3278  * For example, they could both be snapshots of the same filesystem, and
3279  * 'earlier' is before 'later'.  Or 'earlier' could be the origin of
3280  * 'later's filesystem.  Or 'earlier' could be an older snapshot in the origin's
3281  * filesystem.  Or 'earlier' could be the origin's origin.
3282  *
3283  * If non-zero, earlier_txg is used instead of earlier's ds_creation_txg.
3284  */
3285 boolean_t
3286 dsl_dataset_is_before(dsl_dataset_t *later, dsl_dataset_t *earlier,
3287         uint64_t earlier_txg)
3288 {
3289         dsl_pool_t *dp = later->ds_dir->dd_pool;
3290         int error;
3291         boolean_t ret;

3293         ASSERT(dsl_pool_config_held(dp));
3294         ASSERT(earlier->ds_is_snapshot || earlier_txg != 0);
3289         ASSERT(dsl_dataset_is_snapshot(earlier) || earlier_txg != 0);

3296         if (earlier_txg == 0)
3297                 earlier_txg = dsl_dataset_phys(earlier)->ds_creation_txg;

3299         if (later->ds_is_snapshot &&
3294             (dsl_dataset_is_snapshot(later) &&
3300             earlier_txg >= dsl_dataset_phys(later)->ds_creation_txg)
3301                 return (B_FALSE);

3303         if (later->ds_dir == earlier->ds_dir)
3304                 return (B_TRUE);
```

```
3305         if (!dsl_dir_is_clone(later->ds_dir))
3306                 return (B_FALSE);

3308         if (dsl_dir_phys(later->ds_dir)->dd_origin_obj == earlier->ds_object)
3309                 return (B_TRUE);
3310         dsl_dataset_t *origin;
3311         error = dsl_dataset_hold_obj(dp,
3312             dsl_dir_phys(later->ds_dir)->dd_origin_obj, FTAG, &origin);
3313         if (error != 0)
3314                 return (B_FALSE);
3315         ret = dsl_dataset_is_before(origin, earlier, earlier_txg);
3316         dsl_dataset_rele(origin, FTAG);
3317         return (ret);
3318 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   14105 Tue Jan  6 10:37:48 2015
new/usr/src/uts/common/fs/zfs/dsl_deadlist.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#include <sys/dsl_dataset.h>
#include <sys/dmu.h>
#include <sys/refcount.h>
#include <sys/zap.h>
#include <sys/zfs_context.h>
#include <sys/dsl_pool.h>

/*
 * Deadlist concurrency:
 *
 * Deadlists can only be modified from the syncing thread.
 *
 * Except for dsl_deadlist_insert(), it can only be modified with the
 * dp_config_rwlock held with RW_WRITER.
 *
 * The accessors (dsl_deadlist_space() and dsl_deadlist_space_range()) can
 * be called concurrently, from open context, with the dl_config_rwlock held
 * with RW_READER.
 *
 * Therefore, we only need to provide locking between dsl_deadlist_insert() and
 * the accessors, protecting:
 *     dl_phys->dl_used,comp,uncomp
 *     and protecting the dl_tree from being loaded.
 * The locking is provided by dl_lock.  Note that locking on the bpobj_t
 * provides its own locking, and dl_oldfmt is immutable.
 */

static int
dsl_deadlist_compare(const void *arg1, const void *arg2)
{
        const dsl_deadlist_entry_t *dle1 = arg1;
```



```
        const dsl_deadlist_entry_t *dle2 = arg2;

        if (dle1->dle_mintxg < dle2->dle_mintxg)
                return (-1);
        else if (dle1->dle_mintxg > dle2->dle_mintxg)
                return (+1);
        else
                return (0);
}
_____unchanged_portion_omitted_

void
dsl_deadlist_close(dsl_deadlist_t *dl)
{
        void *cookie = NULL;
        dsl_deadlist_entry_t *dle;

        dl->dl_os = NULL;

        if (dl->dl_oldfmt) {
                dl->dl_oldfmt = B_FALSE;
                bpobj_close(&dl->dl_bpobj);
                return;
        }

        if (dl->dl_havetree) {
                while ((dle = avl_destroy_nodes(&dl->dl_tree, &cookie))
                    != NULL) {
                        bpobj_close(&dle->dle_bpobj);
                        kmem_free(dle, sizeof (*dle));
                }
                avl_destroy(&dl->dl_tree);
        }
        dmu_buf_rele(dl->dl_dbuf, dl);
        mutex_destroy(&dl->dl_lock);
        dl->dl_dbuf = NULL;
        dl->dl_phys = NULL;
}
_____unchanged_portion_omitted_
```

```
**********************************************************
   19840 Tue Jan  6 10:37:48 2015
new/usr/src/uts/common/fs/zfs/dsl_deleg.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.com>
**********************************************************
______unchanged_portion_omitted_

 539 /*
 540  * Check if user has requested permission.
 541  */
 542 int
 543 dsl_deleg_access_impl(dsl_dataset_t *ds, const char *perm, cred_t *cr)
 544 {
 545         dsl_dir_t *dd;
 546         dsl_pool_t *dp;
 547         void *cookie;
 548         int     error;
 549         char    checkflag;
 550         objset_t *mos;
 551         avl_tree_t permsets;
 552         perm_set_t *setnode;

 554         dp = ds->ds_dir->dd_pool;
 555         mos = dp->dp_meta_objset;

 557         if (dsl_delegation_on(mos) == B_FALSE)
 558                 return (SET_ERROR(ECANCELED));

 560         if (spa_version(dmu_objset_spa(dp->dp_meta_objset)) <
 561             SPA_VERSION_DELEGATED_PERMS)
 562                 return (SET_ERROR(EPERM));

 564         if (ds->ds_is_snapshot) {
 564         if (dsl_dataset_is_snapshot(ds)) {
 565                 /*
 566                  * Snapshots are treated as descendents only,
 567                  * local permissions do not apply.
 568                  */
 569                 checkflag = ZFS_DELEG_DESCENDENT;
 570         } else {
 571                 checkflag = ZFS_DELEG_LOCAL;
 572         }

 574         avl_create(&permsets, perm_set_compare, sizeof (perm_set_t),
 575             offsetof(perm_set_t, p_node));

 577         ASSERT(dsl_pool_config_held(dp));
 578         for (dd = ds->ds_dir; dd != NULL; dd = dd->dd_parent,
 579             checkflag = ZFS_DELEG_DESCENDENT) {
 580                 uint64_t zapobj;
 581                 boolean_t expanded;

 583                 /*
 584                  * If not in global zone then make sure
 585                  * the zoned property is set
 586                  */
 587                 if (!INGLOBALZONE(curproc)) {
 588                         uint64_t zoned;

 590                         if (dsl_prop_get_dd(dd,
 591                             zfs_prop_to_name(ZFS_PROP_ZONED),
 592                             8, 1, &zoned, NULL, B_FALSE) != 0)
```



```
 593                                 break;
 594                         if (!zoned)
 595                                 break;
 596                 }
 597                 zapobj = dsl_dir_phys(dd)->dd_deleg_zapobj;

 599                 if (zapobj == 0)
 600                         continue;

 602                 dsl_load_user_sets(mos, zapobj, &permsets, checkflag, cr);
 603 again:
 604                 expanded = B_FALSE;
 605                 for (setnode = avl_first(&permsets); setnode;
 606                     setnode = AVL_NEXT(&permsets, setnode)) {
 607                         if (setnode->p_matched == B_TRUE)
 608                                 continue;

 610                         /* See if this set directly grants this permission */
 611                         error = dsl_check_access(mos, zapobj,
 612                             ZFS_DELEG_NAMED_SET, 0, setnode->p_setname, perm);
 613                         if (error == 0)
 614                                 goto success;
 615                         if (error == EPERM)
 616                                 setnode->p_matched = B_TRUE;

 618                         /* See if this set includes other sets */
 619                         error = dsl_load_sets(mos, zapobj,
 620                             ZFS_DELEG_NAMED_SET_SETS, 0,
 621                             setnode->p_setname, &permsets);
 622                         if (error == 0)
 623                                 setnode->p_matched = expanded = B_TRUE;
 624                 }
 625                 /*
 626                  * If we expanded any sets, that will define more sets,
 627                  * which we need to check.
 628                  */
 629                 if (expanded)
 630                         goto again;

 632                 error = dsl_check_user_access(mos, zapobj, perm, checkflag, cr);
 633                 if (error == 0)
 634                         goto success;
 635         }
 636         error = SET_ERROR(EPERM);
 637 success:

 639         cookie = NULL;
 640         while ((setnode = avl_destroy_nodes(&permsets, &cookie)) != NULL)
 641                 kmem_free(setnode, sizeof (perm_set_t));

 643         return (error);
 644 }
______unchanged_portion_omitted_
```

```
**********************************************************
   27596 Tue Jan  6 10:37:48 2015
new/usr/src/uts/common/fs/zfs/dsl_destroy.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.com>
**********************************************************
______unchanged_portion_omitted_

  51 int
  52 dsl_destroy_snapshot_check_impl(dsl_dataset_t *ds, boolean_t defer)
  53 {
  54         if (!ds->ds_is_snapshot)
  54         if (!dsl_dataset_is_snapshot(ds))
  55                 return (SET_ERROR(EINVAL));

  57         if (dsl_dataset_long_held(ds))
  58                 return (SET_ERROR(EBUSY));

  60         /*
  61          * Only allow deferred destroy on pools that support it.
  62          * NOTE: deferred destroy is only supported on snapshots.
  63          */
  64         if (defer) {
  65                 if (spa_version(ds->ds_dir->dd_pool->dp_spa) <
  66                     SPA_VERSION_USERREFS)
  67                         return (SET_ERROR(ENOTSUP));
  68                 return (0);
  69         }

  71         /*
  72          * If this snapshot has an elevated user reference count,
  73          * we can't destroy it yet.
  74          */
  75         if (ds->ds_userrefs > 0)
  76                 return (SET_ERROR(EBUSY));

  78         /*
  79          * Can't delete a branch point.
  80          */
  81         if (dsl_dataset_phys(ds)->ds_num_children > 1)
  82                 return (SET_ERROR(EEXIST));

  84         return (0);
  85 }
______unchanged_portion_omitted_

 237 void
 238 dsl_destroy_snapshot_sync_impl(dsl_dataset_t *ds, boolean_t defer, dmu_tx_t *tx)
 239 {
 240         int err;
 241         int after_branch_point = FALSE;
 242         dsl_pool_t *dp = ds->ds_dir->dd_pool;
 243         objset_t *mos = dp->dp_meta_objset;
 244         dsl_dataset_t *ds_prev = NULL;
 245         uint64_t obj;

 247         ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));
 248         ASSERT3U(dsl_dataset_phys(ds)->ds_bp.blk_birth, <=, tx->tx_txg);
 249         ASSERT(refcount_is_zero(&ds->ds_longholds));

 251         if (defer &&
 252             (ds->ds_userrefs > 0 ||
 253             dsl_dataset_phys(ds)->ds_num_children > 1)) {
```



```
 254                 ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);
 255                 dmu_buf_will_dirty(ds->ds_dbuf, tx);
 256                 dsl_dataset_phys(ds)->ds_flags |= DS_FLAG_DEFER_DESTROY;
 257                 spa_history_log_internal_ds(ds, "defer_destroy", tx, "");
 258                 return;
 259         }

 261         ASSERT3U(dsl_dataset_phys(ds)->ds_num_children, <=, 1);

 263         /* We need to log before removing it from the namespace. */
 264         spa_history_log_internal_ds(ds, "destroy", tx, "");

 266         dsl_scan_ds_destroyed(ds, tx);

 268         obj = ds->ds_object;

 270         if (ds->ds_large_blocks) {
 271                 ASSERT0(zap_contains(mos, obj, DS_FIELD_LARGE_BLOCKS));
 272                 spa_feature_decr(dp->dp_spa, SPA_FEATURE_LARGE_BLOCKS, tx);
 273         }
 274         if (dsl_dataset_phys(ds)->ds_prev_snap_obj != 0) {
 275                 ASSERT3P(ds->ds_prev, ==, NULL);
 276                 VERIFY0(dsl_dataset_hold_obj(dp,
 277                     dsl_dataset_phys(ds)->ds_prev_snap_obj, FTAG, &ds_prev));
 278                 after_branch_point =
 279                     (dsl_dataset_phys(ds_prev)->ds_next_snap_obj != obj);

 281                 dmu_buf_will_dirty(ds_prev->ds_dbuf, tx);
 282                 if (after_branch_point &&
 283                     dsl_dataset_phys(ds_prev)->ds_next_clones_obj != 0) {
 284                         dsl_dataset_remove_from_next_clones(ds_prev, obj, tx);
 285                         if (dsl_dataset_phys(ds)->ds_next_snap_obj != 0) {
 286                                 VERIFY0(zap_add_int(mos,
 287                                     dsl_dataset_phys(ds_prev)->
 288                                     ds_next_clones_obj,
 289                                     dsl_dataset_phys(ds)->ds_next_snap_obj,
 290                                     tx));
 291                         }
 292                 }
 293                 if (!after_branch_point) {
 294                         dsl_dataset_phys(ds_prev)->ds_next_snap_obj =
 295                             dsl_dataset_phys(ds)->ds_next_snap_obj;
 296                 }
 297         }

 299         dsl_dataset_t *ds_next;
 300         uint64_t old_unique;
 301         uint64_t used = 0, comp = 0, uncomp = 0;

 303         VERIFY0(dsl_dataset_hold_obj(dp,
 304             dsl_dataset_phys(ds)->ds_next_snap_obj, FTAG, &ds_next));
 305         ASSERT3U(dsl_dataset_phys(ds_next)->ds_prev_snap_obj, ==, obj);

 307         old_unique = dsl_dataset_phys(ds_next)->ds_unique_bytes;

 309         dmu_buf_will_dirty(ds_next->ds_dbuf, tx);
 310         dsl_dataset_phys(ds_next)->ds_prev_snap_obj =
 311             dsl_dataset_phys(ds)->ds_prev_snap_obj;
 312         dsl_dataset_phys(ds_next)->ds_prev_snap_txg =
 313             dsl_dataset_phys(ds)->ds_prev_snap_txg;
 314         ASSERT3U(dsl_dataset_phys(ds)->ds_prev_snap_txg, ==,
 315             ds_prev ? dsl_dataset_phys(ds_prev)->ds_creation_txg : 0);

 317         if (ds_next->ds_deadlist.dl_oldfmt) {
 318                 process_old_deadlist(ds, ds_prev, ds_next,
 319                     after_branch_point, tx);
```

```
320         } else {
321                 /* Adjust prev's unique space. */
322                 if (ds_prev && !after_branch_point) {
323                         dsl_deadlist_space_range(&ds_next->ds_deadlist,
324                             dsl_dataset_phys(ds_prev)->ds_prev_snap_txg,
325                             dsl_dataset_phys(ds)->ds_prev_snap_txg,
326                             &used, &comp, &uncomp);
327                         dsl_dataset_phys(ds_prev)->ds_unique_bytes += used;
328                 }
329
330                 /* Adjust snapused. */
331                 dsl_deadlist_space_range(&ds_next->ds_deadlist,
332                     dsl_dataset_phys(ds)->ds_prev_snap_txg, UINT64_MAX,
333                     &used, &comp, &uncomp);
334                 dsl_dir_diduse_space(ds->ds_dir, DD_USED_SNAP,
335                     -used, -comp, -uncomp, tx);
336
337                 /* Move blocks to be freed to pool's free list. */
338                 dsl_deadlist_move_bpobj(&ds_next->ds_deadlist,
339                     &dp->dp_free_bpobj, dsl_dataset_phys(ds)->ds_prev_snap_txg,
340                     tx);
341                 dsl_dir_diduse_space(tx->tx_pool->dp_free_dir,
342                     DD_USED_HEAD, used, comp, uncomp, tx);
343
344                 /* Merge our deadlist into next's and free it. */
345                 dsl_deadlist_merge(&ds_next->ds_deadlist,
346                     dsl_dataset_phys(ds)->ds_deadlist_obj, tx);
347         }
348         dsl_deadlist_close(&ds->ds_deadlist);
349         dsl_deadlist_free(mos, dsl_dataset_phys(ds)->ds_deadlist_obj, tx);
350         dmu_buf_will_dirty(ds->ds_dbuf, tx);
351         dsl_dataset_phys(ds)->ds_deadlist_obj = 0;
352
353         /* Collapse range in clone heads */
354         dsl_dataset_remove_clones_key(ds,
355             dsl_dataset_phys(ds)->ds_creation_txg, tx);
356
357         if (ds_next->ds_is_snapshot) {
357         if (dsl_dataset_is_snapshot(ds_next)) {
358                 dsl_dataset_t *ds_nextnext;
359
360                 /*
361                  * Update next's unique to include blocks which
362                  * were previously shared by only this snapshot
363                  * and it.  Those blocks will be born after the
364                  * prev snap and before this snap, and will have
365                  * died after the next snap and before the one
366                  * after that (ie. be on the snap after next's
367                  * deadlist).
368                  */
369                 VERIFY0(dsl_dataset_hold_obj(dp,
370                     dsl_dataset_phys(ds_next)->ds_next_snap_obj,
371                     FTAG, &ds_nextnext));
372                 dsl_deadlist_space_range(&ds_nextnext->ds_deadlist,
373                     dsl_dataset_phys(ds)->ds_prev_snap_txg,
374                     dsl_dataset_phys(ds)->ds_creation_txg,
375                     &used, &comp, &uncomp);
376                 dsl_dataset_phys(ds_next)->ds_unique_bytes += used;
377                 dsl_dataset_rele(ds_nextnext, FTAG);
378                 ASSERT3P(ds_next->ds_prev, ==, NULL);
379
380                 /* Collapse range in this head. */
381                 dsl_dataset_t *hds;
382                 VERIFY0(dsl_dataset_hold_obj(dp,
383                     dsl_dir_phys(ds->ds_dir)->dd_head_dataset_obj, FTAG, &hds));
384                 dsl_deadlist_remove_key(&hds->ds_deadlist,
```



```
385                     dsl_dataset_phys(ds)->ds_creation_txg, tx);
386                 dsl_dataset_rele(hds, FTAG);
387
388         } else {
389                 ASSERT3P(ds_next->ds_prev, ==, ds);
390                 dsl_dataset_rele(ds_next->ds_prev, ds_next);
391                 ds_next->ds_prev = NULL;
392                 if (ds_prev) {
393                         VERIFY0(dsl_dataset_hold_obj(dp,
394                             dsl_dataset_phys(ds)->ds_prev_snap_obj,
395                             ds_next, &ds_next->ds_prev));
396                 }
397
398                 dsl_dataset_recalc_head_uniq(ds_next);
399
400                 /*
401                  * Reduce the amount of our unconsumed refreservation
402                  * being charged to our parent by the amount of
403                  * new unique data we have gained.
404                  */
405                 if (old_unique < ds_next->ds_reserved) {
406                         int64_t mrsdelta;
407                         uint64_t new_unique =
408                             dsl_dataset_phys(ds_next)->ds_unique_bytes;
409
410                         ASSERT(old_unique <= new_unique);
411                         mrsdelta = MIN(new_unique - old_unique,
412                             ds_next->ds_reserved - old_unique);
413                         dsl_dir_diduse_space(ds->ds_dir,
414                             DD_USED_REFRSRV, -mrsdelta, 0, 0, tx);
415                 }
416         }
417         dsl_dataset_rele(ds_next, FTAG);
418
419         /*
420          * This must be done after the dsl_traverse(), because it will
421          * re-open the objset.
422          */
423         if (ds->ds_objset) {
424                 dmu_objset_evict(ds->ds_objset);
425                 ds->ds_objset = NULL;
426         }
427
428         /* remove from snapshot namespace */
429         dsl_dataset_t *ds_head;
430         ASSERT(dsl_dataset_phys(ds)->ds_snapnames_zapobj == 0);
431         VERIFY0(dsl_dataset_hold_obj(dp,
432             dsl_dir_phys(ds->ds_dir)->dd_head_dataset_obj, FTAG, &ds_head));
433         VERIFY0(dsl_dataset_get_snapname(ds));
434 #ifdef ZFS_DEBUG
435         {
436                 uint64_t val;
437
438                 err = dsl_dataset_snap_lookup(ds_head,
439                     ds->ds_snapname, &val);
440                 ASSERT0(err);
441                 ASSERT3U(val, ==, obj);
442         }
443 #endif
444         VERIFY0(dsl_dataset_snap_remove(ds_head, ds->ds_snapname, tx, B_TRUE));
445         dsl_dataset_rele(ds_head, FTAG);
446
447         if (ds_prev != NULL)
448                 dsl_dataset_rele(ds_prev, FTAG);
449
450         spa_prop_clear_bootfs(dp->dp_spa, ds->ds_object, tx);
```

```
452         if (dsl_dataset_phys(ds)->ds_next_clones_obj != 0) {
453                 uint64_t count;
454                 ASSERT0(zap_count(mos,
455                     dsl_dataset_phys(ds)->ds_next_clones_obj, &count) &&
456                     count == 0);
457                 VERIFY0(dmu_object_free(mos,
458                     dsl_dataset_phys(ds)->ds_next_clones_obj, tx));
459         }
460         if (dsl_dataset_phys(ds)->ds_props_obj != 0)
461                 VERIFY0(zap_destroy(mos, dsl_dataset_phys(ds)->ds_props_obj,
462                     tx));
463         if (dsl_dataset_phys(ds)->ds_userrefs_obj != 0)
464                 VERIFY0(zap_destroy(mos, dsl_dataset_phys(ds)->ds_userrefs_obj,
465                     tx));
466         dsl_dir_rele(ds->ds_dir, ds);
467         ds->ds_dir = NULL;
468         dmu_object_free_zapified(mos, obj, tx);
469 }
```

*unchanged_portion_omitted_*

```
600 int
601 dsl_destroy_head_check_impl(dsl_dataset_t *ds, int expected_holds)
602 {
603         int error;
604         uint64_t count;
605         objset_t *mos;

607         ASSERT(!ds->ds_is_snapshot);
608         if (ds->ds_is_snapshot)
607         ASSERT(!dsl_dataset_is_snapshot(ds));
608         if (dsl_dataset_is_snapshot(ds))
609                 return (SET_ERROR(EINVAL));

611         if (refcount_count(&ds->ds_longholds) != expected_holds)
612                 return (SET_ERROR(EBUSY));

614         mos = ds->ds_dir->dd_pool->dp_meta_objset;

616         /*
617          * Can't delete a head dataset if there are snapshots of it.
618          * (Except if the only snapshots are from the branch we cloned
619          * from.)
620          */
621         if (ds->ds_prev != NULL &&
622             dsl_dataset_phys(ds->ds_prev)->ds_next_snap_obj == ds->ds_object)
623                 return (SET_ERROR(EBUSY));

625         /*
626          * Can't delete if there are children of this fs.
627          */
628         error = zap_count(mos,
629             dsl_dir_phys(ds->ds_dir)->dd_child_dir_zapobj, &count);
630         if (error != 0)
631                 return (error);
632         if (count != 0)
633                 return (SET_ERROR(EEXIST));

635         if (dsl_dir_is_clone(ds->ds_dir) && DS_IS_DEFER_DESTROY(ds->ds_prev) &&
636             dsl_dataset_phys(ds->ds_prev)->ds_num_children == 2 &&
637             ds->ds_prev->ds_userrefs == 0) {
638                 /* We need to remove the origin snapshot as well. */
639                 if (!refcount_is_zero(&ds->ds_prev->ds_longholds))
640                         return (SET_ERROR(EBUSY));
641         }
642         return (0);
```



```
643 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   56638 Tue Jan  6 10:37:48 2015
new/usr/src/uts/common/fs/zfs/dsl_dir.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 Martin Matuska. All rights reserved.
 * Copyright (c) 2014 Joyent, Inc. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#include <sys/dmu.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/dsl_deleg.h>
#include <sys/dmu_impl.h>
#include <sys/spa.h>
#include <sys/metaslab.h>
#include <sys/zap.h>
#include <sys/zio.h>
#include <sys/arc.h>
#include <sys/sunddi.h>
#include <sys/zfeature.h>
#include <sys/policy.h>
#include <sys/zfs_znode.h>
#include "zfs_namecheck.h"
#include "zfs_prop.h"

/*
 * Filesystem and Snapshot Limits
 * ------------------------------
 *
 * These limits are used to restrict the number of filesystems and/or snapshots
 * that can be created at a given level in the tree or below. A typical
 * use-case is with a delegated dataset where the administrator wants to ensure
 * that a user within the zone is not creating too many additional filesystems
```



```
 * or snapshots, even though they're not exceeding their space quota.
 *
 * The filesystem and snapshot counts are stored as extensible properties. This
 * capability is controlled by a feature flag and must be enabled to be used.
 * Once enabled, the feature is not active until the first limit is set. At
 * that point, future operations to create/destroy filesystems or snapshots
 * will validate and update the counts.
 *
 * Because the count properties will not exist before the feature is active,
 * the counts are updated when a limit is first set on an uninitialized
 * dsl_dir node in the tree (The filesystem/snapshot count on a node includes
 * all of the nested filesystems/snapshots. Thus, a new leaf node has a
 * filesystem count of 0 and a snapshot count of 0. Non-existent filesystem and
 * snapshot count properties on a node indicate uninitialized counts on that
 * node.) When first setting a limit on an uninitialized node, the code starts
 * at the filesystem with the new limit and descends into all sub-filesystems
 * to add the count properties.
 *
 * In practice this is lightweight since a limit is typically set when the
 * filesystem is created and thus has no children. Once valid, changing the
 * limit value won't require a re-traversal since the counts are already valid.
 * When recursively fixing the counts, if a node with a limit is encountered
 * during the descent, the counts are known to be valid and there is no need to
 * descend into that filesystem's children. The counts on filesystems above the
 * one with the new limit will still be uninitialized, unless a limit is
 * eventually set on one of those filesystems. The counts are always recursively
 * updated when a limit is set on a dataset, unless there is already a limit.
 * When a new limit value is set on a filesystem with an existing limit, it is
 * possible for the new limit to be less than the current count at that level
 * since a user who can change the limit is also allowed to exceed the limit.
 *
 * Once the feature is active, then whenever a filesystem or snapshot is
 * created, the code recurses up the tree, validating the new count against the
 * limit at each initialized level. In practice, most levels will not have a
 * limit set. If there is a limit at any initialized level up the tree, the
 * check must pass or the creation will fail. Likewise, when a filesystem or
 * snapshot is destroyed, the counts are recursively adjusted all the way up
 * the initizized nodes in the tree. Renaming a filesystem into different point
 * in the tree will first validate, then update the counts on each branch up to
 * the common ancestor. A receive will also validate the counts and then update
 * them.
 *
 * An exception to the above behavior is that the limit is not enforced if the
 * user has permission to modify the limit. This is primarily so that
 * recursive snapshots in the global zone always work. We want to prevent a
 * denial-of-service in which a lower level delegated dataset could max out its
 * limit and thus block recursive snapshots from being taken in the global zone.
 * Because of this, it is possible for the snapshot count to be over the limit
 * and snapshots taken in the global zone could cause a lower level dataset to
 * hit or exceed its limit. The administrator taking the global zone recursive
 * snapshot should be aware of this side-effect and behave accordingly.
 * For consistency, the filesystem limit is also not enforced if the user can
 * modify the limit.
 *
 * The filesystem and snapshot limits are validated by dsl_fs_ss_limit_check()
 * and updated by dsl_fs_ss_count_adjust(). A new limit value is setup in
 * dsl_dir_activate_fs_ss_limit() and the counts are adjusted, if necessary, by
 * dsl_dir_init_fs_ss_count().
 *
 * There is a special case when we receive a filesystem that already exists. In
 * this case a temporary clone name of %X is created (see dmu_recv_begin). We
 * never update the filesystem counts for temporary clones.
 *
 * Likewise, we do not update the snapshot counts for temporary snapshots,
 * such as those created by zfs diff.
 */
```

```
125 extern inline dsl_dir_phys_t *dsl_dir_phys(dsl_dir_t *dd);

127 static uint64_t dsl_dir_space_towrite(dsl_dir_t *dd);

128 /* ARGSUSED */
129 static void
130 dsl_dir_evict(void *dbu)
130 dsl_dir_evict(dmu_buf_t *db, void *arg)
131 {
132         dsl_dir_t *dd = dbu;
132         dsl_dir_t *dd = arg;
133         dsl_pool_t *dp = dd->dd_pool;
134         int t;

136         dd->dd_dbuf = NULL;

138         for (t = 0; t < TXG_SIZE; t++) {
139                 ASSERT(!txg_list_member(&dp->dp_dirty_dirs, dd, t));
140                 ASSERT(dd->dd_tempreserved[t] == 0);
141                 ASSERT(dd->dd_space_towrite[t] == 0);
142         }

144         if (dd->dd_parent)
145                 dsl_dir_async_rele(dd->dd_parent, dd);
143                 dsl_dir_rele(dd->dd_parent, dd);

147         spa_async_close(dd->dd_pool->dp_spa, dd);
145         spa_close(dd->dd_pool->dp_spa, dd);

149         /*
150          * The props callback list should have been cleaned up by
151          * objset_evict().
152          */
153         list_destroy(&dd->dd_prop_cbs);
154         mutex_destroy(&dd->dd_lock);
155         kmem_free(dd, sizeof (dsl_dir_t));
156 }

158 int
159 dsl_dir_hold_obj(dsl_pool_t *dp, uint64_t ddobj,
160     const char *tail, void *tag, dsl_dir_t **ddp)
161 {
162         dmu_buf_t *dbuf;
163         dsl_dir_t *dd;
164         int err;

166         ASSERT(dsl_pool_config_held(dp));

168         err = dmu_bonus_hold(dp->dp_meta_objset, ddobj, tag, &dbuf);
169         if (err != 0)
170                 return (err);
171         dd = dmu_buf_get_user(dbuf);
172 #ifdef ZFS_DEBUG
173         {
174                 dmu_object_info_t doi;
175                 dmu_object_info_from_db(dbuf, &doi);
176                 ASSERT3U(doi.doi_bonus_type, ==, DMU_OT_DSL_DIR);
177                 ASSERT3U(doi.doi_bonus_size, >=, sizeof (dsl_dir_phys_t));
178         }
179 #endif
180         if (dd == NULL) {
181                 dsl_dir_t *winner;

183                 dd = kmem_zalloc(sizeof (dsl_dir_t), KM_SLEEP);
184                 dd->dd_object = ddobj;
```



```
185                 dd->dd_dbuf = dbuf;
186                 dd->dd_pool = dp;
187                 mutex_init(&dd->dd_lock, NULL, MUTEX_DEFAULT, NULL);

189                 list_create(&dd->dd_prop_cbs, sizeof (dsl_prop_cb_record_t),
190                     offsetof(dsl_prop_cb_record_t, cbr_node));

192                 dsl_dir_snap_cmtime_update(dd);

194                 if (dsl_dir_phys(dd)->dd_parent_obj) {
195                         err = dsl_dir_hold_obj(dp,
196                             dsl_dir_phys(dd)->dd_parent_obj, NULL, dd,
197                             &dd->dd_parent);
198                         if (err != 0)
199                                 goto errout;
200                         if (tail) {
201 #ifdef ZFS_DEBUG
202                                 uint64_t foundobj;

204                                 err = zap_lookup(dp->dp_meta_objset,
205                                     dsl_dir_phys(dd->dd_parent)->
206                                     dd_child_dir_zapobj, tail,
207                                     sizeof (foundobj), 1, &foundobj);
208                                 ASSERT(err || foundobj == ddobj);
209 #endif
210                                 (void) strcpy(dd->dd_myname, tail);
211                         } else {
212                                 err = zap_value_search(dp->dp_meta_objset,
213                                     dsl_dir_phys(dd->dd_parent)->
214                                     dd_child_dir_zapobj,
215                                     ddobj, 0, dd->dd_myname);
216                         }
217                         if (err != 0)
218                                 goto errout;
219                 } else {
220                         (void) strcpy(dd->dd_myname, spa_name(dp->dp_spa));
221                 }

223                 if (dsl_dir_is_clone(dd)) {
224                         dmu_buf_t *origin_bonus;
225                         dsl_dataset_phys_t *origin_phys;

227                         /*
228                          * We can't open the origin dataset, because
229                          * that would require opening this dsl_dir.
230                          * Just look at its phys directly instead.
231                          */
232                         err = dmu_bonus_hold(dp->dp_meta_objset,
233                             dsl_dir_phys(dd)->dd_origin_obj, FTAG,
234                             &origin_bonus);
235                         if (err != 0)
236                                 goto errout;
237                         origin_phys = origin_bonus->db_data;
238                         dd->dd_origin_txg =
239                             origin_phys->ds_creation_txg;
240                         dmu_buf_rele(origin_bonus, FTAG);
241                 }

243                 dmu_buf_init_user(&dd->dd_dbu, dsl_dir_evict, &dd->dd_dbuf);
244                 winner = dmu_buf_set_user_ie(dbuf, &dd->dd_dbu);
245                 if (winner != NULL) {
241                 winner = dmu_buf_set_user_ie(dbuf, dd, dsl_dir_evict);
242                 if (winner) {
246                         if (dd->dd_parent)
247                                 dsl_dir_rele(dd->dd_parent, dd);
248                         mutex_destroy(&dd->dd_lock);
```

```
249                         kmem_free(dd, sizeof (dsl_dir_t));
250                         dd = winner;
251                 } else {
252                         spa_open_ref(dp->dp_spa, dd);
253                 }
254         }

256         /*
257          * The dsl_dir_t has both open-to-close and instantiate-to-evict
258          * holds on the spa.  We need the open-to-close holds because
259          * otherwise the spa_refcnt wouldn't change when we open a
260          * dir which the spa also has open, so we could incorrectly
261          * think it was OK to unload/export/destroy the pool.  We need
262          * the instantiate-to-evict hold because the dsl_dir_t has a
263          * pointer to the dd_pool, which has a pointer to the spa_t.
264          */
265         spa_open_ref(dp->dp_spa, tag);
266         ASSERT3P(dd->dd_pool, ==, dp);
267         ASSERT3U(dd->dd_object, ==, ddobj);
268         ASSERT3P(dd->dd_dbuf, ==, dbuf);
269         *ddp = dd;
270         return (0);

272 errout:
273         if (dd->dd_parent)
274                 dsl_dir_rele(dd->dd_parent, dd);
275         mutex_destroy(&dd->dd_lock);
276         kmem_free(dd, sizeof (dsl_dir_t));
277         dmu_buf_rele(dbuf, tag);
278         return (err);
279 }
_____unchanged_portion_omitted_

289 /*
290  * Remove a reference to the given dsl dir that is being asynchronously
291  * released.  Async releases occur from a taskq performing eviction of
292  * dsl datasets and dirs.  This process is identical to a normal release
293  * with the exception of using the async API for releasing the reference on
294  * the spa.
295  */
296 void
297 dsl_dir_async_rele(dsl_dir_t *dd, void *tag)
298 {
299         dprintf_dd(dd, "%s\n", "");
300         spa_async_close(dd->dd_pool->dp_spa, tag);
301         dmu_buf_rele(dd->dd_dbuf, tag);
302 }

304 /* buf must be long enough (MAXNAMELEN + strlen(MOS_DIR_NAME) + 1 should do) */
305 void
306 dsl_dir_name(dsl_dir_t *dd, char *buf)
307 {
308         if (dd->dd_parent) {
309                 dsl_dir_name(dd->dd_parent, buf);
310                 (void) strcat(buf, "/");
311         } else {
312                 buf[0] = '\0';
313         }
314         if (!MUTEX_HELD(&dd->dd_lock)) {
315                 /*
316                  * recursive mutex so that we can use
317                  * dprintf_dd() with dd_lock held
318                  */
319                 mutex_enter(&dd->dd_lock);
320                 (void) strcat(buf, dd->dd_myname);
321                 mutex_exit(&dd->dd_lock);
```



```
322         } else {
323                 (void) strcat(buf, dd->dd_myname);
324         }
325 }
_____unchanged_portion_omitted_

400 /*
401  * Return the dsl_dir_t, and possibly the last component which couldn't
402  * be found in *tail.  The name must be in the specified dsl_pool_t.  This
403  * thread must hold the dp_config_rwlock for the pool.  Returns NULL if the
404  * path is bogus, or if tail==NULL and we couldn't parse the whole name.
405  * (*tail)[0] == '@' means that the last component is a snapshot.
406  */
407 int
408 dsl_dir_hold(dsl_pool_t *dp, const char *name, void *tag,
409     dsl_dir_t **ddp, const char **tailp)
410 {
411         char buf[MAXNAMELEN];
412         const char *spaname, *next, *nextnext = NULL;
413         int err;
414         dsl_dir_t *dd;
415         uint64_t ddobj;

417         err = getcomponent(name, buf, &next);
418         if (err != 0)
419                 return (err);

421         /* Make sure the name is in the specified pool. */
422         spaname = spa_name(dp->dp_spa);
423         if (strcmp(buf, spaname) != 0)
424                 return (SET_ERROR(EINVAL));

426         ASSERT(dsl_pool_config_held(dp));

428         err = dsl_dir_hold_obj(dp, dp->dp_root_dir_obj, NULL, tag, &dd);
429         if (err != 0) {
430                 return (err);
431         }

433         while (next != NULL) {
434                 dsl_dir_t *child_dd;
416                 dsl_dir_t *child_ds;
435                 err = getcomponent(next, buf, &nextnext);
436                 if (err != 0)
437                         break;
438                 ASSERT(next[0] != '\0');
439                 if (next[0] == '@')
440                         break;
441                 dprintf("looking up %s in obj%lld\n",
442                     buf, dsl_dir_phys(dd)->dd_child_dir_zapobj);

444                 err = zap_lookup(dp->dp_meta_objset,
445                     dsl_dir_phys(dd)->dd_child_dir_zapobj,
446                     buf, sizeof (ddobj), 1, &ddobj);
447                 if (err != 0) {
448                         if (err == ENOENT)
449                                 err = 0;
450                         break;
451                 }

453                 err = dsl_dir_hold_obj(dp, ddobj, buf, tag, &child_dd);
435                 err = dsl_dir_hold_obj(dp, ddobj, buf, tag, &child_ds);
454                 if (err != 0)
455                         break;
456                 dsl_dir_rele(dd, tag);
457                 dd = child_dd;
```

```
439                 dd = child_ds;
458                 next = nextnext;
459         }

461         if (err != 0) {
462                 dsl_dir_rele(dd, tag);
463                 return (err);
464         }

466         /*
467          * It's an error if there's more than one component left, or
468          * tailp==NULL and there's any component left.
469          */
470         if (next != NULL &&
471             (tailp == NULL || (nextnext && nextnext[0] != '\0'))) {
472                 /* bad path name */
473                 dsl_dir_rele(dd, tag);
474                 dprintf("next=%p (%s) tail=%p\n", next, next?next:"", tailp);
475                 err = SET_ERROR(ENOENT);
476         }
477         if (tailp != NULL)
478                 *tailp = next;
479         *ddp = dd;
480         return (err);
481 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   30886 Tue Jan  6 10:37:49 2015
new/usr/src/uts/common/fs/zfs/dsl_pool.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2011, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 Steven Hartland. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#include <sys/dsl_pool.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_synctask.h>
#include <sys/dsl_scan.h>
#include <sys/dnode.h>
#include <sys/dmu_tx.h>
#include <sys/dmu_objset.h>
#include <sys/arc.h>
#include <sys/zap.h>
#include <sys/zio.h>
#include <sys/zfs_context.h>
#include <sys/fs/zfs.h>
#include <sys/zfs_znode.h>
#include <sys/spa_impl.h>
#include <sys/dsl_deadlist.h>
#include <sys/bptree.h>
#include <sys/zfeature.h>
#include <sys/zil_impl.h>
#include <sys/dsl_userhold.h>

/*
 * ZFS Write Throttle
 * ------------------
 *
 * ZFS must limit the rate of incoming writes to the rate at which it is able
 * to sync data modifications to the backend storage. Throttling by too much
 * creates an artificial limit; throttling by too little can only be sustained
 * for short periods and would lead to highly lumpy performance. On a per-pool
```



```
 * basis, ZFS tracks the amount of modified (dirty) data. As operations change
 * data, the amount of dirty data increases; as ZFS syncs out data, the amount
 * of dirty data decreases. When the amount of dirty data exceeds a
 * predetermined threshold further modifications are blocked until the amount
 * of dirty data decreases (as data is synced out).
 *
 * The limit on dirty data is tunable, and should be adjusted according to
 * both the IO capacity and available memory of the system. The larger the
 * window, the more ZFS is able to aggregate and amortize metadata (and data)
 * changes. However, memory is a limited resource, and allowing for more dirty
 * data comes at the cost of keeping other useful data in memory (for example
 * ZFS data cached by the ARC).
 *
 * Implementation
 *
 * As buffers are modified dsl_pool_willuse_space() increments both the per-
 * txg (dp_dirty_pertxg[]) and poolwide (dp_dirty_total) accounting of
 * dirty space used; dsl_pool_dirty_space() decrements those values as data
 * is synced out from dsl_pool_sync(). While only the poolwide value is
 * relevant, the per-txg value is useful for debugging. The tunable
 * zfs_dirty_data_max determines the dirty space limit. Once that value is
 * exceeded, new writes are halted until space frees up.
 *
 * The zfs_dirty_data_sync tunable dictates the threshold at which we
 * ensure that there is a txg syncing (see the comment in txg.c for a full
 * description of transaction group stages).
 *
 * The IO scheduler uses both the dirty space limit and current amount of
 * dirty data as inputs. Those values affect the number of concurrent IOs ZFS
 * issues. See the comment in vdev_queue.c for details of the IO scheduler.
 *
 * The delay is also calculated based on the amount of dirty data.  See the
 * comment above dmu_tx_delay() for details.
 */

/*
 * zfs_dirty_data_max will be set to zfs_dirty_data_max_percent% of all memory,
 * capped at zfs_dirty_data_max_max.  It can also be overridden in /etc/system.
 */
uint64_t zfs_dirty_data_max;
uint64_t zfs_dirty_data_max_max = 4ULL * 1024 * 1024 * 1024;
int zfs_dirty_data_max_percent = 10;

/*
 * If there is at least this much dirty data, push out a txg.
 */
uint64_t zfs_dirty_data_sync = 64 * 1024 * 1024;

/*
 * Once there is this amount of dirty data, the dmu_tx_delay() will kick in
 * and delay each transaction.
 * This value should be >= zfs_vdev_async_write_active_max_dirty_percent.
 */
int zfs_delay_min_dirty_percent = 60;

/*
 * This controls how quickly the delay approaches infinity.
 * Larger values cause it to delay more for a given amount of dirty data.
 * Therefore larger values will cause there to be less dirty data for a
 * given throughput.
 *
 * For the smoothest delay, this value should be about 1 billion divided
 * by the maximum number of operations per second.  This will smoothly
 * handle between 10x and 1/10th this number.
 *
 * Note: zfs_delay_scale * zfs_dirty_data_max must be < 2^64, due to the
```

```
 124  * multiply in dmu_tx_delay().
 125  */
 126 uint64_t zfs_delay_scale = 1000 * 1000 * 1000 / 2000;


 129 hrtime_t zfs_throttle_delay = MSEC2NSEC(10);
 130 hrtime_t zfs_throttle_resolution = MSEC2NSEC(10);

 132 int
 133 dsl_pool_open_special_dir(dsl_pool_t *dp, const char *name, dsl_dir_t **ddp)
 134 {
 135         uint64_t obj;
 136         int err;

 138         err = zap_lookup(dp->dp_meta_objset,
 139             dsl_dir_phys(dp->dp_root_dir)->dd_child_dir_zapobj,
 140             name, sizeof (obj), 1, &obj);
 141         if (err)
 142                 return (err);

 144         return (dsl_dir_hold_obj(dp, obj, name, dp, ddp));
 145 }
_____unchanged_portion_omitted_

 286 void
 287 dsl_pool_close(dsl_pool_t *dp)
 288 {
 289         /*
 290          * Drop our references from dsl_pool_open().
 291          *
 292          * Since we held the origin_snap from "syncing" context (which
 293          * includes pool-opening context), it actually only got a "ref"
 294          * and not a hold, so just drop that here.
 295          */
 296         if (dp->dp_origin_snap)
 297                 dsl_dataset_rele(dp->dp_origin_snap, dp);
 298         if (dp->dp_mos_dir)
 299                 dsl_dir_rele(dp->dp_mos_dir, dp);
 300         if (dp->dp_free_dir)
 301                 dsl_dir_rele(dp->dp_free_dir, dp);
 302         if (dp->dp_leak_dir)
 303                 dsl_dir_rele(dp->dp_leak_dir, dp);
 304         if (dp->dp_root_dir)
 305                 dsl_dir_rele(dp->dp_root_dir, dp);

 307         bpobj_close(&dp->dp_free_bpobj);

 309         /* undo the dmu_objset_open_impl(mos) from dsl_pool_open() */
 310         if (dp->dp_meta_objset)
 311                 dmu_objset_evict(dp->dp_meta_objset);

 313         txg_list_destroy(&dp->dp_dirty_datasets);
 314         txg_list_destroy(&dp->dp_dirty_zilogs);
 315         txg_list_destroy(&dp->dp_sync_tasks);
 316         txg_list_destroy(&dp->dp_dirty_dirs);

 318         arc_flush(dp->dp_spa);
 319         txg_fini(dp);
 320         dsl_scan_fini(dp);
 321         dmu_buf_user_evict_wait();

 323         rrw_destroy(&dp->dp_config_rwlock);
 324         mutex_destroy(&dp->dp_lock);
 325         taskq_destroy(dp->dp_vnrele_taskq);
 326         if (dp->dp_blkstats)
 327                 kmem_free(dp->dp_blkstats, sizeof (zfs_all_blkstats_t));
```



```
 328         kmem_free(dp, sizeof (dsl_pool_t));
 329 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   29101 Tue Jan  6 10:37:49 2015
new/usr/src/uts/common/fs/zfs/dsl_prop.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.com>
**********************************************************
______unchanged_portion_omitted_

 160 int
 161 dsl_prop_get_ds(dsl_dataset_t *ds, const char *propname,
 162     int intsz, int numints, void *buf, char *setpoint)
 163 {
 164         zfs_prop_t prop = zfs_name_to_prop(propname);
 165         boolean_t inheritable;
 166         boolean_t snapshot;
 166         uint64_t zapobj;

 168         ASSERT(dsl_pool_config_held(ds->ds_dir->dd_pool));
 169         inheritable = (prop == ZPROP_INVAL || zfs_prop_inheritable(prop));
 171         snapshot = dsl_dataset_is_snapshot(ds);
 170         zapobj = dsl_dataset_phys(ds)->ds_props_obj;

 172         if (zapobj != 0) {
 173                 objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
 174                 int err;

 176                 ASSERT(ds->ds_is_snapshot);
 178                 ASSERT(snapshot);

 178                 /* Check for a local value. */
 179                 err = zap_lookup(mos, zapobj, propname, intsz, numints, buf);
 180                 if (err != ENOENT) {
 181                         if (setpoint != NULL && err == 0)
 182                                 dsl_dataset_name(ds, setpoint);
 183                         return (err);
 184                 }

 186                 /*
 187                  * Skip the check for a received value if there is an explicit
 188                  * inheritance entry.
 189                  */
 190                 if (inheritable) {
 191                         char *inheritstr = kmem_asprintf("%s%s", propname,
 192                             ZPROP_INHERIT_SUFFIX);
 193                         err = zap_contains(mos, zapobj, inheritstr);
 194                         strfree(inheritstr);
 195                         if (err != 0 && err != ENOENT)
 196                                 return (err);
 197                 }

 199                 if (err == ENOENT) {
 200                         /* Check for a received value. */
 201                         char *recvdstr = kmem_asprintf("%s%s", propname,
 202                             ZPROP_RECVD_SUFFIX);
 203                         err = zap_lookup(mos, zapobj, recvdstr,
 204                             intsz, numints, buf);
 205                         strfree(recvdstr);
 206                         if (err != ENOENT) {
 207                                 if (setpoint != NULL && err == 0)
 208                                         (void) strcpy(setpoint,
 209                                             ZPROP_SOURCE_VAL_RECVD);
 210                                 return (err);
 211                         }
```



```
 212                 }
 213         }

 215         return (dsl_prop_get_dd(ds->ds_dir, propname,
 216             intsz, numints, buf, setpoint, ds->ds_is_snapshot));
 218             intsz, numints, buf, setpoint, snapshot));
 217 }
______unchanged_portion_omitted_

 528 void
 529 dsl_prop_set_sync_impl(dsl_dataset_t *ds, const char *propname,
 530     zprop_source_t source, int intsz, int numints, const void *value,
 531     dmu_tx_t *tx)
 532 {
 533         objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
 534         uint64_t zapobj, intval, dummy;
 535         int isint;
 536         char valbuf[32];
 537         const char *valstr = NULL;
 538         char *inheritstr;
 539         char *recvdstr;
 540         char *tbuf = NULL;
 541         int err;
 542         uint64_t version = spa_version(ds->ds_dir->dd_pool->dp_spa);

 544         isint = (dodefault(propname, 8, 1, &intval) == 0);

 546         if (ds->ds_is_snapshot) {
 548         if (dsl_dataset_is_snapshot(ds)) {
 547                 ASSERT(version >= SPA_VERSION_SNAP_PROPS);
 548                 if (dsl_dataset_phys(ds)->ds_props_obj == 0) {
 549                         dmu_buf_will_dirty(ds->ds_dbuf, tx);
 550                         dsl_dataset_phys(ds)->ds_props_obj =
 551                             zap_create(mos,
 552                             DMU_OT_DSL_PROPS, DMU_OT_NONE, 0, tx);
 553                 }
 554                 zapobj = dsl_dataset_phys(ds)->ds_props_obj;
 555         } else {
 556                 zapobj = dsl_dir_phys(ds->ds_dir)->dd_props_zapobj;
 557         }

 559         if (version < SPA_VERSION_RECVD_PROPS) {
 560                 if (source & ZPROP_SRC_NONE)
 561                         source = ZPROP_SRC_NONE;
 562                 else if (source & ZPROP_SRC_RECEIVED)
 563                         source = ZPROP_SRC_LOCAL;
 564         }

 566         inheritstr = kmem_asprintf("%s%s", propname, ZPROP_INHERIT_SUFFIX);
 567         recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

 569         switch (source) {
 570         case ZPROP_SRC_NONE:
 571                 /*
 572                  * revert to received value, if any (inherit -S)
 573                  * - remove propname
 574                  * - remove propname$inherit
 575                  */
 576                 err = zap_remove(mos, zapobj, propname, tx);
 577                 ASSERT(err == 0 || err == ENOENT);
 578                 err = zap_remove(mos, zapobj, inheritstr, tx);
 579                 ASSERT(err == 0 || err == ENOENT);
 580                 break;
 581         case ZPROP_SRC_LOCAL:
 582                 /*
 583                  * remove propname$inherit
```

```
584                  * set propname -> value
585                  */
586                 err = zap_remove(mos, zapobj, inheritstr, tx);
587                 ASSERT(err == 0 || err == ENOENT);
588                 VERIFY0(zap_update(mos, zapobj, propname,
589                     intsz, numints, value, tx));
590                 break;
591         case ZPROP_SRC_INHERITED:
592                 /*
593                  * explicitly inherit
594                  * - remove propname
595                  * - set propname$inherit
596                  */
597                 err = zap_remove(mos, zapobj, propname, tx);
598                 ASSERT(err == 0 || err == ENOENT);
599                 if (version >= SPA_VERSION_RECVD_PROPS &&
600                     dsl_prop_get_int_ds(ds, ZPROP_HAS_RECVD, &dummy) == 0) {
601                         dummy = 0;
602                         VERIFY0(zap_update(mos, zapobj, inheritstr,
603                             8, 1, &dummy, tx));
604                 }
605                 break;
606         case ZPROP_SRC_RECEIVED:
607                 /*
608                  * set propname$recvd -> value
609                  */
610                 err = zap_update(mos, zapobj, recvdstr,
611                     intsz, numints, value, tx);
612                 ASSERT(err == 0);
613                 break;
614         case (ZPROP_SRC_NONE | ZPROP_SRC_LOCAL | ZPROP_SRC_RECEIVED):
615                 /*
616                  * clear local and received settings
617                  * - remove propname
618                  * - remove propname$inherit
619                  * - remove propname$recvd
620                  */
621                 err = zap_remove(mos, zapobj, propname, tx);
622                 ASSERT(err == 0 || err == ENOENT);
623                 err = zap_remove(mos, zapobj, inheritstr, tx);
624                 ASSERT(err == 0 || err == ENOENT);
625                 /* FALLTHRU */
626         case (ZPROP_SRC_NONE | ZPROP_SRC_RECEIVED):
627                 /*
628                  * remove propname$recvd
629                  */
630                 err = zap_remove(mos, zapobj, recvdstr, tx);
631                 ASSERT(err == 0 || err == ENOENT);
632                 break;
633         default:
634                 cmn_err(CE_PANIC, "unexpected property source: %d", source);
635         }

637         strfree(inheritstr);
638         strfree(recvdstr);

640         if (isint) {
641                 VERIFY0(dsl_prop_get_int_ds(ds, propname, &intval));

643                 if (ds->ds_is_snapshot) {
645                 if (dsl_dataset_is_snapshot(ds)) {
644                         dsl_prop_cb_record_t *cbr;
645                         /*
646                          * It's a snapshot; nothing can inherit this
647                          * property, so just look for callbacks on this
648                          * ds here.
```



```
649                          */
650                         mutex_enter(&ds->ds_dir->dd_lock);
651                         for (cbr = list_head(&ds->ds_dir->dd_prop_cbs); cbr;
652                             cbr = list_next(&ds->ds_dir->dd_prop_cbs, cbr)) {
653                                 if (cbr->cbr_ds == ds &&
654                                     strcmp(cbr->cbr_propname, propname) == 0)
655                                         cbr->cbr_func(cbr->cbr_arg, intval);
656                         }
657                         mutex_exit(&ds->ds_dir->dd_lock);
658                 } else {
659                         dsl_prop_changed_notify(ds->ds_dir->dd_pool,
660                             ds->ds_dir->dd_object, propname, intval, TRUE);
661                 }

663                 (void) snprintf(valbuf, sizeof (valbuf),
664                     "%lld", (longlong_t)intval);
665                 valstr = valbuf;
666         } else {
667                 if (source == ZPROP_SRC_LOCAL) {
668                         valstr = value;
669                 } else {
670                         tbuf = kmem_alloc(ZAP_MAXVALUELEN, KM_SLEEP);
671                         if (dsl_prop_get_ds(ds, propname, 1,
672                             ZAP_MAXVALUELEN, tbuf, NULL) == 0)
673                                 valstr = tbuf;
674                 }
675         }

677         spa_history_log_internal_ds(ds, (source == ZPROP_SRC_NONE ||
678             source == ZPROP_SRC_INHERITED) ? "inherit" : "set", tx,
679             "%s=%s", propname, (valstr == NULL ? "" : valstr));

681         if (tbuf != NULL)
682                 kmem_free(tbuf, ZAP_MAXVALUELEN);
683 }
_____unchanged_portion_omitted_

730 static int
731 dsl_props_set_check(void *arg, dmu_tx_t *tx)
732 {
733         dsl_props_set_arg_t *dpsa = arg;
734         dsl_pool_t *dp = dmu_tx_pool(tx);
735         dsl_dataset_t *ds;
736         uint64_t version;
737         nvpair_t *elem = NULL;
738         int err;

740         err = dsl_dataset_hold(dp, dpsa->dpsa_dsname, FTAG, &ds);
741         if (err != 0)
742                 return (err);

744         version = spa_version(ds->ds_dir->dd_pool->dp_spa);
745         while ((elem = nvlist_next_nvpair(dpsa->dpsa_props, elem)) != NULL) {
746                 if (strlen(nvpair_name(elem)) >= ZAP_MAXNAMELEN) {
747                         dsl_dataset_rele(ds, FTAG);
748                         return (SET_ERROR(ENAMETOOLONG));
749                 }
750                 if (nvpair_type(elem) == DATA_TYPE_STRING) {
751                         char *valstr = fnvpair_value_string(elem);
752                         if (strlen(valstr) >= (version <
753                             SPA_VERSION_STMF_PROP ?
754                             ZAP_OLDMAXVALUELEN : ZAP_MAXVALUELEN)) {
755                                 dsl_dataset_rele(ds, FTAG);
756                                 return (E2BIG);
757                         }
758                 }
```

```
 759         }

 761         if (ds->ds_is_snapshot && version < SPA_VERSION_SNAP_PROPS) {
 763         if (dsl_dataset_is_snapshot(ds) && version < SPA_VERSION_SNAP_PROPS) {
 762                 dsl_dataset_rele(ds, FTAG);
 763                 return (SET_ERROR(ENOTSUP));
 764         }
 765         dsl_dataset_rele(ds, FTAG);
 766         return (0);
 767 }
______unchanged_portion_omitted_

 969 /*
 970  * Iterate over all properties for this dataset and return them in an nvlist.
 971  */
 972 static int
 973 dsl_prop_get_all_ds(dsl_dataset_t *ds, nvlist_t **nvp,
 974     dsl_prop_getflags_t flags)
 975 {
 976         dsl_dir_t *dd = ds->ds_dir;
 977         dsl_pool_t *dp = dd->dd_pool;
 978         objset_t *mos = dp->dp_meta_objset;
 979         int err = 0;
 980         char setpoint[MAXNAMELEN];

 982         VERIFY(nvlist_alloc(nvp, NV_UNIQUE_NAME, KM_SLEEP) == 0);

 984         if (ds->ds_is_snapshot)
 986         if (dsl_dataset_is_snapshot(ds))
 985                 flags |= DSL_PROP_GET_SNAPSHOT;

 987         ASSERT(dsl_pool_config_held(dp));

 989         if (dsl_dataset_phys(ds)->ds_props_obj != 0) {
 990                 ASSERT(flags & DSL_PROP_GET_SNAPSHOT);
 991                 dsl_dataset_name(ds, setpoint);
 992                 err = dsl_prop_get_all_impl(mos,
 993                     dsl_dataset_phys(ds)->ds_props_obj, setpoint, flags, *nvp);
 994                 if (err)
 995                         goto out;
 996         }

 998         for (; dd != NULL; dd = dd->dd_parent) {
 999                 if (dd != ds->ds_dir || (flags & DSL_PROP_GET_SNAPSHOT)) {
1000                         if (flags & (DSL_PROP_GET_LOCAL |
1001                             DSL_PROP_GET_RECEIVED))
1002                                 break;
1003                         flags |= DSL_PROP_GET_INHERITING;
1004                 }
1005                 dsl_dir_name(dd, setpoint);
1006                 err = dsl_prop_get_all_impl(mos,
1007                     dsl_dir_phys(dd)->dd_props_zapobj, setpoint, flags, *nvp);
1008                 if (err)
1009                         break;
1010         }
1011 out:
1012         return (err);
1013 }
______unchanged_portion_omitted_
```

```
**********************************************************
   53801 Tue Jan  6 10:37:49 2015
new/usr/src/uts/common/fs/zfs/dsl_scan.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.com>
**********************************************************
______unchanged_portion_omitted_

 374 static uint64_t
 375 dsl_scan_ds_maxtxg(dsl_dataset_t *ds)
 376 {
 377         uint64_t smt = ds->ds_dir->dd_pool->dp_scan->scn_phys.scn_max_txg;
 378         if (ds->ds_is_snapshot)
 378         if (dsl_dataset_is_snapshot(ds))
 379                 return (MIN(smt, dsl_dataset_phys(ds)->ds_creation_txg));
 380         return (smt);
 381 }
______unchanged_portion_omitted_

 803 void
 804 dsl_scan_ds_destroyed(dsl_dataset_t *ds, dmu_tx_t *tx)
 805 {
 806         dsl_pool_t *dp = ds->ds_dir->dd_pool;
 807         dsl_scan_t *scn = dp->dp_scan;
 808         uint64_t mintxg;

 810         if (scn->scn_phys.scn_state != DSS_SCANNING)
 811                 return;

 813         if (scn->scn_phys.scn_bookmark.zb_objset == ds->ds_object) {
 814                 if (ds->ds_is_snapshot) {
 814                 if (dsl_dataset_is_snapshot(ds)) {
 815                         /* Note, scn_cur_{min,max}_txg stays the same. */
 816                         scn->scn_phys.scn_bookmark.zb_objset =
 817                             dsl_dataset_phys(ds)->ds_next_snap_obj;
 818                         zfs_dbgmsg("destroying ds %llu; currently traversing; "
 819                             "reset zb_objset to %llu",
 820                             (u_longlong_t)ds->ds_object,
 821                             (u_longlong_t)dsl_dataset_phys(ds)->
 822                             ds_next_snap_obj);
 823                         scn->scn_phys.scn_flags |= DSF_VISIT_DS_AGAIN;
 824                 } else {
 825                         SET_BOOKMARK(&scn->scn_phys.scn_bookmark,
 826                             ZB_DESTROYED_OBJSET, 0, 0, 0);
 827                         zfs_dbgmsg("destroying ds %llu; currently traversing; "
 828                             "reset bookmark to -1,0,0,0",
 829                             (u_longlong_t)ds->ds_object);
 830                 }
 831         } else if (zap_lookup_int_key(dp->dp_meta_objset,
 832             scn->scn_phys.scn_queue_obj, ds->ds_object, &mintxg) == 0) {
 833                 ASSERT3U(dsl_dataset_phys(ds)->ds_num_children, <=, 1);
 834                 VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
 835                     scn->scn_phys.scn_queue_obj, ds->ds_object, tx));
 836                 if (ds->ds_is_snapshot) {
 836                 if (dsl_dataset_is_snapshot(ds)) {
 837                         /*
 838                          * We keep the same mintxg; it could be >
 839                          * ds_creation_txg if the previous snapshot was
 840                          * deleted too.
 841                          */
 842                         VERIFY(zap_add_int_key(dp->dp_meta_objset,
 843                             scn->scn_phys.scn_queue_obj,
 844                             dsl_dataset_phys(ds)->ds_next_snap_obj,
```



```
 845                             mintxg, tx) == 0);
 846                         zfs_dbgmsg("destroying ds %llu; in queue; "
 847                             "replacing with %llu",
 848                             (u_longlong_t)ds->ds_object,
 849                             (u_longlong_t)dsl_dataset_phys(ds)->
 850                             ds_next_snap_obj);
 851                 } else {
 852                         zfs_dbgmsg("destroying ds %llu; in queue; removing",
 853                             (u_longlong_t)ds->ds_object);
 854                 }
 855         } else {
 856                 zfs_dbgmsg("destroying ds %llu; ignoring",
 857                     (u_longlong_t)ds->ds_object);
 858         }

 860         /*
 861          * dsl_scan_sync() should be called after this, and should sync
 862          * out our changed state, but just to be safe, do it here.
 863          */
 864         dsl_scan_sync_state(scn, tx);
 865 }
______unchanged_portion_omitted_

1001 static void
1002 dsl_scan_visitds(dsl_scan_t *scn, uint64_t dsobj, dmu_tx_t *tx)
1003 {
1004         dsl_pool_t *dp = scn->scn_dp;
1005         dsl_dataset_t *ds;
1006         objset_t *os;

1008         VERIFY3U(0, ==, dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));

1010         if (dmu_objset_from_ds(ds, &os))
1011                 goto out;

1013         /*
1014          * Only the ZIL in the head (non-snapshot) is valid.  Even though
1015          * snapshots can have ZIL block pointers (which may be the same
1016          * BP as in the head), they must be ignored.  So we traverse the
1017          * ZIL here, rather than in scan_recurse(), because the regular
1018          * snapshot block-sharing rules don't apply to it.
1019          */
1020         if (DSL_SCAN_IS_SCRUB_RESILVER(scn) && !ds->ds_is_snapshot)
1020         if (DSL_SCAN_IS_SCRUB_RESILVER(scn) && !dsl_dataset_is_snapshot(ds))
1021                 dsl_scan_zil(dp, &os->os_zil_header);

1023         /*
1024          * Iterate over the bps in this ds.
1025          */
1026         dmu_buf_will_dirty(ds->ds_dbuf, tx);
1027         dsl_scan_visit_rootbp(scn, ds, &dsl_dataset_phys(ds)->ds_bp, tx);

1029         char *dsname = kmem_alloc(ZFS_MAXNAMELEN, KM_SLEEP);
1030         dsl_dataset_name(ds, dsname);
1031         zfs_dbgmsg("scanned dataset %llu (%s) with min=%llu max=%llu; "
1032             "pausing=%u",
1033             (longlong_t)dsobj, dsname,
1034             (longlong_t)scn->scn_phys.scn_cur_min_txg,
1035             (longlong_t)scn->scn_phys.scn_cur_max_txg,
1036             (int)scn->scn_pausing);
1037         kmem_free(dsname, ZFS_MAXNAMELEN);

1039         if (scn->scn_pausing)
1040                 goto out;

1042         /*
```

```
1043          * We've finished this pass over this dataset.
1044          */

1046         /*
1047          * If we did not completely visit this dataset, do another pass.
1048          */
1049         if (scn->scn_phys.scn_flags & DSF_VISIT_DS_AGAIN) {
1050                 zfs_dbgmsg("incomplete pass; visiting again");
1051                 scn->scn_phys.scn_flags &= ~DSF_VISIT_DS_AGAIN;
1052                 VERIFY(zap_add_int_key(dp->dp_meta_objset,
1053                     scn->scn_phys.scn_queue_obj, ds->ds_object,
1054                     scn->scn_phys.scn_cur_max_txg, tx) == 0);
1055                 goto out;
1056         }

1058         /*
1059          * Add descendent datasets to work queue.
1060          */
1061         if (dsl_dataset_phys(ds)->ds_next_snap_obj != 0) {
1062                 VERIFY(zap_add_int_key(dp->dp_meta_objset,
1063                     scn->scn_phys.scn_queue_obj,
1064                     dsl_dataset_phys(ds)->ds_next_snap_obj,
1065                     dsl_dataset_phys(ds)->ds_creation_txg, tx) == 0);
1066         }
1067         if (dsl_dataset_phys(ds)->ds_num_children > 1) {
1068                 boolean_t usenext = B_FALSE;
1069                 if (dsl_dataset_phys(ds)->ds_next_clones_obj != 0) {
1070                         uint64_t count;
1071                         /*
1072                          * A bug in a previous version of the code could
1073                          * cause upgrade_clones_cb() to not set
1074                          * ds_next_snap_obj when it should, leading to a
1075                          * missing entry.  Therefore we can only use the
1076                          * next_clones_obj when its count is correct.
1077                          */
1078                         int err = zap_count(dp->dp_meta_objset,
1079                             dsl_dataset_phys(ds)->ds_next_clones_obj, &count);
1080                         if (err == 0 &&
1081                             count == dsl_dataset_phys(ds)->ds_num_children - 1)
1082                                 usenext = B_TRUE;
1083                 }

1085                 if (usenext) {
1086                         VERIFY0(zap_join_key(dp->dp_meta_objset,
1087                             dsl_dataset_phys(ds)->ds_next_clones_obj,
1088                             scn->scn_phys.scn_queue_obj,
1089                             dsl_dataset_phys(ds)->ds_creation_txg, tx));
1090                 } else {
1091                         struct enqueue_clones_arg eca;
1092                         eca.tx = tx;
1093                         eca.originobj = ds->ds_object;

1095                         VERIFY0(dmu_objset_find_dp(dp, dp->dp_root_dir_obj,
1096                             enqueue_clones_cb, &eca, DS_FIND_CHILDREN));
1097                 }
1098         }

1100 out:
1101         dsl_dataset_rele(ds, FTAG);
1102 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   17584 Tue Jan  6 10:37:49 2015
new/usr/src/uts/common/fs/zfs/dsl_userhold.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.com>
**********************************************************
______unchanged_portion_omitted_

 346 static int
 347 dsl_dataset_user_release_check_one(dsl_dataset_user_release_arg_t *ddura,
 348     dsl_dataset_t *ds, nvlist_t *holds, const char *snapname)
 349 {
 350         uint64_t zapobj;
 351         nvlist_t *holds_found;
 352         objset_t *mos;
 353         int numholds;

 355         if (!ds->ds_is_snapshot)
 355         if (!dsl_dataset_is_snapshot(ds))
 356                 return (SET_ERROR(EINVAL));

 358         if (nvlist_empty(holds))
 359                 return (0);

 361         numholds = 0;
 362         mos = ds->ds_dir->dd_pool->dp_meta_objset;
 363         zapobj = dsl_dataset_phys(ds)->ds_userrefs_obj;
 364         holds_found = fnvlist_alloc();

 366         for (nvpair_t *pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
 367             pair = nvlist_next_nvpair(holds, pair)) {
 368                 uint64_t tmp;
 369                 int error;
 370                 const char *holdname = nvpair_name(pair);

 372                 if (zapobj != 0)
 373                         error = zap_lookup(mos, zapobj, holdname, 8, 1, &tmp);
 374                 else
 375                         error = SET_ERROR(ENOENT);

 377                 /*
 378                  * Non-existent holds are put on the errlist, but don't
 379                  * cause an overall failure.
 380                  */
 381                 if (error == ENOENT) {
 382                         if (ddura->ddura_errlist != NULL) {
 383                                 char *errtag = kmem_asprintf("%s#%s",
 384                                     snapname, holdname);
 385                                 fnvlist_add_int32(ddura->ddura_errlist, errtag,
 386                                     ENOENT);
 387                                 strfree(errtag);
 388                         }
 389                         continue;
 390                 }

 392                 if (error != 0) {
 393                         fnvlist_free(holds_found);
 394                         return (error);
 395                 }

 397                 fnvlist_add_boolean(holds_found, holdname);
 398                 numholds++;
 399         }
```



```
 401         if (DS_IS_DEFER_DESTROY(ds) &&
 402             dsl_dataset_phys(ds)->ds_num_children == 1 &&
 403             ds->ds_userrefs == numholds) {
 404                 /* we need to destroy the snapshot as well */
 405                 if (dsl_dataset_long_held(ds)) {
 406                         fnvlist_free(holds_found);
 407                         return (SET_ERROR(EBUSY));
 408                 }
 409                 fnvlist_add_boolean(ddura->ddura_todelete, snapname);
 410         }

 412         if (numholds != 0) {
 413                 fnvlist_add_nvlist(ddura->ddura_chkholds, snapname,
 414                     holds_found);
 415         }
 416         fnvlist_free(holds_found);

 418         return (0);
 419 }
______unchanged_portion_omitted_
```

```
**********************************************************
   51692 Tue Jan  6 10:37:49 2015
new/usr/src/uts/common/fs/zfs/sa.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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.
 * Copyright (c) 2014 Spectra Logic Corporation, 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:
 *
 * 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
 * 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  */

 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);

 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);

 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 };
_____unchanged_portion_omitted_

 200 /*
 201  * Special dummy layout used for buffers with no attributes.
 202  */
 203 sa_attr_type_t sa_dummy_zpl_layout[] = { 0 };

 205 static int sa_legacy_attr_count = 16;
 206 static kmem_cache_t *sa_cache = NULL;

 208 /*ARGSUSED*/
 209 static int
 210 sa_cache_constructor(void *buf, void *unused, int kmflag)
 211 {
 212         sa_handle_t *hdl = buf;

 214         hdl->sa_dbu.dbu_evict_func = NULL;
 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 }

 225 /*ARGSUSED*/
 226 static void
 227 sa_cache_destructor(void *buf, void *unused)
 228 {
 229         sa_handle_t *hdl = buf;
 230         hdl->sa_dbu.dbu_evict_func = NULL;
 231         mutex_destroy(&hdl->sa_lock);
 232 }
_____unchanged_portion_omitted_

1306 /*ARGSUSED*/
1307 static void
1308 sa_evict(void *dbu)
1304 void
```



```
1305 sa_evict(dmu_buf_t *db, void *sap)
1309 {
1310         panic("evicting sa dbuf\n");
1307         panic("evicting sa dbuf %p\n", (void *)db);
1311 }
_____unchanged_portion_omitted_

1346 void
1347 sa_handle_destroy(sa_handle_t *hdl)
1348 {
1349         dmu_buf_t *db = hdl->sa_bonus;

1351         mutex_enter(&hdl->sa_lock);
1352         (void) dmu_buf_remove_user(db, &hdl->sa_dbu);
1347         (void) dmu_buf_update_user((dmu_buf_t *)hdl->sa_bonus, hdl,
1348             NULL, NULL);

1354         if (hdl->sa_bonus_tab) {
1355                 sa_idx_tab_rele(hdl->sa_os, hdl->sa_bonus_tab);
1356                 hdl->sa_bonus_tab = NULL;
1357         }
1358         if (hdl->sa_spill_tab) {
1359                 sa_idx_tab_rele(hdl->sa_os, hdl->sa_spill_tab);
1360                 hdl->sa_spill_tab = NULL;
1361         }

1363         dmu_buf_rele(hdl->sa_bonus, NULL);

1365         if (hdl->sa_spill)
1366                 dmu_buf_rele((dmu_buf_t *)hdl->sa_spill, NULL);
1367         mutex_exit(&hdl->sa_lock);

1369         kmem_cache_free(sa_cache, hdl);
1370 }

1372 int
1373 sa_handle_get_from_db(objset_t *os, dmu_buf_t *db, void *userp,
1374     sa_handle_type_t hdl_type, sa_handle_t **handlepp)
1375 {
1376         int error = 0;
1377         dmu_object_info_t doi;
1378         sa_handle_t *handle = NULL;
1374         sa_handle_t *handle;

1380 #ifdef ZFS_DEBUG
1381         dmu_object_info_from_db(db, &doi);
1382         ASSERT(doi.doi_bonus_type == DMU_OT_SA ||
1383             doi.doi_bonus_type == DMU_OT_ZNODE);
1384 #endif
1385         /* find handle, if it exists */
1386         /* if one doesn't exist then create a new one, and initialize it */

1388         if (hdl_type == SA_HDL_SHARED)
1389                 handle = dmu_buf_get_user(db);

1384         handle = (hdl_type == SA_HDL_SHARED) ? dmu_buf_get_user(db) : NULL;
1391         if (handle == NULL) {
1392                 sa_handle_t *winner = NULL;

1386                 sa_handle_t *newhandle;
1394                 handle = kmem_cache_alloc(sa_cache, KM_SLEEP);
1395                 handle->sa_userp = userp;
1396                 handle->sa_bonus = db;
1397                 handle->sa_os = os;
1398                 handle->sa_spill = NULL;
```

```
1400                 error = sa_build_index(handle, SA_BONUS);
1394                 newhandle = (hdl_type == SA_HDL_SHARED) ?
1395                     dmu_buf_set_user_ie(db, handle, sa_evict) : NULL;

1402                 if (hdl_type == SA_HDL_SHARED) {
1403                         dmu_buf_init_user(&handle->sa_dbu, sa_evict, NULL);
1404                         winner = dmu_buf_set_user_ie(db, &handle->sa_dbu);
1405                 }

1407                 if (winner != NULL) {
1397                 if (newhandle != NULL) {
1408                         kmem_cache_free(sa_cache, handle);
1409                         handle = winner;
1399                         handle = newhandle;
1410                 }
1411         }
1412         *handlepp = handle;

1414         return (error);
1415 }
______unchanged_portion_omitted_

1909 void
1910 sa_update_user(sa_handle_t *newhdl, sa_handle_t *oldhdl)
1911 {
1912         (void) dmu_buf_update_user((dmu_buf_t *)newhdl->sa_bonus,
1913             oldhdl, newhdl, sa_evict);
1914         oldhdl->sa_bonus = NULL;
1915 }

1919 void
1920 sa_set_userp(sa_handle_t *hdl, void *ptr)
1921 {
1922         hdl->sa_userp = ptr;
1923 }
______unchanged_portion_omitted_
```

```
**********************************************************
  180708 Tue Jan  6 10:37:50 2015
new/usr/src/uts/common/fs/zfs/spa.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2011, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013, 2014, Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

/*
 * SPA: Storage Pool Allocator
 *
 * This file contains all the routines used when modifying on-disk SPA state.
 * This includes opening, importing, destroying, exporting a pool, and syncing a
 * pool.
 */

#include <sys/zfs_context.h>
#include <sys/fm/fs/zfs.h>
#include <sys/spa_impl.h>
#include <sys/zio.h>
#include <sys/zio_checksum.h>
#include <sys/dmu.h>
#include <sys/dmu_tx.h>
#include <sys/zap.h>
#include <sys/zil.h>
#include <sys/ddt.h>
#include <sys/vdev_impl.h>
#include <sys/metaslab.h>
#include <sys/metaslab_impl.h>
#include <sys/uberblock_impl.h>
#include <sys/txg.h>
#include <sys/avl.h>
#include <sys/dmu_traverse.h>
#include <sys/dmu_objset.h>
#include <sys/unique.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_dataset.h>
```



```
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/fs/zfs.h>
#include <sys/arc.h>
#include <sys/callb.h>
#include <sys/systeminfo.h>
#include <sys/spa_boot.h>
#include <sys/zfs_ioctl.h>
#include <sys/dsl_scan.h>
#include <sys/zfeature.h>
#include <sys/dsl_destroy.h>

#ifdef  _KERNEL
#include <sys/bootprops.h>
#include <sys/callb.h>
#include <sys/cpupart.h>
#include <sys/pool.h>
#include <sys/sysdc.h>
#include <sys/zone.h>
#endif  /* _KERNEL */

#include "zfs_prop.h"
#include "zfs_comutil.h"

/*
 * The interval, in seconds, at which failed configuration cache file writes
 * should be retried.
 */
static int zfs_ccw_retry_interval = 300;

typedef enum zti_modes {
        ZTI_MODE_FIXED,                 /* value is # of threads (min 1) */
        ZTI_MODE_BATCH,                 /* cpu-intensive; value is ignored */
        ZTI_MODE_NULL,                  /* don't create a taskq */
        ZTI_NMODES
} zti_modes_t;
_____unchanged_portion_omitted_
#endif

/*
 * Activate an uninitialized pool.
 */
static void
spa_activate(spa_t *spa, int mode)
{
        ASSERT(spa->spa_state == POOL_STATE_UNINITIALIZED);

        spa->spa_state = POOL_STATE_ACTIVE;
        spa->spa_mode = mode;

        spa->spa_normal_class = metaslab_class_create(spa, zfs_metaslab_ops);
        spa->spa_log_class = metaslab_class_create(spa, zfs_metaslab_ops);

        /* Try to create a covering process */
        mutex_enter(&spa->spa_proc_lock);
        ASSERT(spa->spa_proc_state == SPA_PROC_NONE);
        ASSERT(spa->spa_proc == &p0);
        spa->spa_did = 0;

        /* Only create a process if we're going to be around a while. */
        if (spa_create_process && strcmp(spa->spa_name, TRYIMPORT_NAME) != 0) {
                if (newproc(spa_thread, (caddr_t)spa, syscid, maxclsyspri,
                    NULL, 0) == 0) {
                        spa->spa_proc_state = SPA_PROC_CREATED;
                        while (spa->spa_proc_state == SPA_PROC_CREATED) {
```

```
1069                                cv_wait(&spa->spa_proc_cv,
1070                                    &spa->spa_proc_lock);
1071                        }
1072                        ASSERT(spa->spa_proc_state == SPA_PROC_ACTIVE);
1073                        ASSERT(spa->spa_proc != &p0);
1074                        ASSERT(spa->spa_did != 0);
1075                } else {
1076 #ifdef _KERNEL
1077                        cmn_err(CE_WARN,
1078                            "Couldn't create process for zfs pool \"%s\"\n",
1079                            spa->spa_name);
1080 #endif
1081                }
1082        }
1083        mutex_exit(&spa->spa_proc_lock);

1085        /* If we didn't create a process, we need to create our taskqs. */
1086        if (spa->spa_proc == &p0) {
1087                spa_create_zio_taskqs(spa);
1088        }

1090        list_create(&spa->spa_config_dirty_list, sizeof (vdev_t),
1091            offsetof(vdev_t, vdev_config_dirty_node));
1092        list_create(&spa->spa_evicting_os_list, sizeof (objset_t),
1093            offsetof(objset_t, os_evicting_node));
1094        list_create(&spa->spa_state_dirty_list, sizeof (vdev_t),
1095            offsetof(vdev_t, vdev_state_dirty_node));

1097        txg_list_create(&spa->spa_vdev_txg_list,
1098            offsetof(struct vdev, vdev_txg_node));

1100        avl_create(&spa->spa_errlist_scrub,
1101            spa_error_entry_compare, sizeof (spa_error_entry_t),
1102            offsetof(spa_error_entry_t, se_avl));
1103        avl_create(&spa->spa_errlist_last,
1104            spa_error_entry_compare, sizeof (spa_error_entry_t),
1105            offsetof(spa_error_entry_t, se_avl));
1106 }

1108 /*
1109  * Opposite of spa_activate().
1110  */
1111 static void
1112 spa_deactivate(spa_t *spa)
1113 {
1114        ASSERT(spa->spa_sync_on == B_FALSE);
1115        ASSERT(spa->spa_dsl_pool == NULL);
1116        ASSERT(spa->spa_root_vdev == NULL);
1117        ASSERT(spa->spa_async_zio_root == NULL);
1118        ASSERT(spa->spa_state != POOL_STATE_UNINITIALIZED);

1120        spa_evicting_os_wait(spa);

1122        txg_list_destroy(&spa->spa_vdev_txg_list);

1124        list_destroy(&spa->spa_config_dirty_list);
1125        list_destroy(&spa->spa_evicting_os_list);
1126        list_destroy(&spa->spa_state_dirty_list);

1128        for (int t = 0; t < ZIO_TYPES; t++) {
1129                for (int q = 0; q < ZIO_TASKQ_TYPES; q++) {
1130                        spa_taskqs_fini(spa, t, q);
1131                }
1132        }

1134        metaslab_class_destroy(spa->spa_normal_class);
```



```
1135        spa->spa_normal_class = NULL;

1137        metaslab_class_destroy(spa->spa_log_class);
1138        spa->spa_log_class = NULL;

1140        /*
1141         * If this was part of an import or the open otherwise failed, we may
1142         * still have errors left in the queues.  Empty them just in case.
1143         */
1144        spa_errlog_drain(spa);

1146        avl_destroy(&spa->spa_errlist_scrub);
1147        avl_destroy(&spa->spa_errlist_last);

1149        spa->spa_state = POOL_STATE_UNINITIALIZED;

1151        mutex_enter(&spa->spa_proc_lock);
1152        if (spa->spa_proc_state != SPA_PROC_NONE) {
1153                ASSERT(spa->spa_proc_state == SPA_PROC_ACTIVE);
1154                spa->spa_proc_state = SPA_PROC_DEACTIVATE;
1155                cv_broadcast(&spa->spa_proc_cv);
1156                while (spa->spa_proc_state == SPA_PROC_DEACTIVATE) {
1157                        ASSERT(spa->spa_proc != &p0);
1158                        cv_wait(&spa->spa_proc_cv, &spa->spa_proc_lock);
1159                }
1160                ASSERT(spa->spa_proc_state == SPA_PROC_GONE);
1161                spa->spa_proc_state = SPA_PROC_NONE;
1162        }
1163        ASSERT(spa->spa_proc == &p0);
1164        mutex_exit(&spa->spa_proc_lock);

1166        /*
1167         * We want to make sure spa_thread() has actually exited the ZFS
1168         * module, so that the module can't be unloaded out from underneath
1169         * it.
1170         */
1171        if (spa->spa_did != 0) {
1172                thread_join(spa->spa_did);
1173                spa->spa_did = 0;
1174        }
1175 }
_____unchanged_portion_omitted_

2067 static int
2068 spa_load(spa_t *spa, spa_load_state_t state, spa_import_type_t type,
2069     boolean_t mosconfig)
2070 {
2071        nvlist_t *config = spa->spa_config;
2072        char *ereport = FM_EREPORT_ZFS_POOL;
2073        char *comment;
2074        int error;
2075        uint64_t pool_guid;
2076        nvlist_t *nvl;

2078        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID, &pool_guid))
2079                return (SET_ERROR(EINVAL));

2081        ASSERT(spa->spa_comment == NULL);
2082        if (nvlist_lookup_string(config, ZPOOL_CONFIG_COMMENT, &comment) == 0)
2083                spa->spa_comment = spa_strdup(comment);

2085        /*
2086         * Versioning wasn't explicitly added to the label until later, so if
2087         * it's not present treat it as the initial version.
2088         */
2089        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VERSION,
```

```
2090             &spa->spa_ubsync.ub_version) != 0)
2091                 spa->spa_ubsync.ub_version = SPA_VERSION_INITIAL;
2092
2093         (void) nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_TXG,
2094             &spa->spa_config_txg);
2095
2096         if ((state == SPA_LOAD_IMPORT || state == SPA_LOAD_TRYIMPORT) &&
2097             spa_guid_exists(pool_guid, 0)) {
2098                 error = SET_ERROR(EEXIST);
2099         } else {
2100                 spa->spa_config_guid = pool_guid;
2101
2102                 if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_SPLIT,
2103                     &nvl) == 0) {
2104                         VERIFY(nvlist_dup(nvl, &spa->spa_config_splitting,
2105                             KM_SLEEP) == 0);
2106                 }
2107
2108                 nvlist_free(spa->spa_load_info);
2109                 spa->spa_load_info = fnvlist_alloc();
2110
2111                 gethrestime(&spa->spa_loaded_ts);
2112                 error = spa_load_impl(spa, pool_guid, config, state, type,
2113                     mosconfig, &ereport);
2114         }
2115
2116         /*
2117          * Don't count references from objsets that are already closed
2118          * and are making their way through the eviction process.
2119          */
2120         spa_evicting_os_wait(spa);
2121         spa->spa_minref = refcount_count(&spa->spa_refcount);
2122         if (error) {
2123                 if (error != EEXIST) {
2124                         spa->spa_loaded_ts.tv_sec = 0;
2125                         spa->spa_loaded_ts.tv_nsec = 0;
2126                 }
2127                 if (error != EBADF) {
2128                         zfs_ereport_post(ereport, spa, NULL, NULL, 0, 0);
2129                 }
2130         }
2131         spa->spa_load_state = error ? SPA_LOAD_ERROR : SPA_LOAD_NONE;
2132         spa->spa_ena = 0;
2133
2134         return (error);
2135 }
_____unchanged_portion_omitted_

3458 /*
3459  * Pool Creation
3460  */
3461 int
3462 spa_create(const char *pool, nvlist_t *nvroot, nvlist_t *props,
3463     nvlist_t *zplprops)
3464 {
3465         spa_t *spa;
3466         char *altroot = NULL;
3467         vdev_t *rvd;
3468         dsl_pool_t *dp;
3469         dmu_tx_t *tx;
3470         int error = 0;
3471         uint64_t txg = TXG_INITIAL;
3472         nvlist_t **spares, **l2cache;
3473         uint_t nspares, nl2cache;
3474         uint64_t version, obj;
3475         boolean_t has_features;
```



```
3477         /*
3478          * If this pool already exists, return failure.
3479          */
3480         mutex_enter(&spa_namespace_lock);
3481         if (spa_lookup(pool) != NULL) {
3482                 mutex_exit(&spa_namespace_lock);
3483                 return (SET_ERROR(EEXIST));
3484         }
3485
3486         /*
3487          * Allocate a new spa_t structure.
3488          */
3489         (void) nvlist_lookup_string(props,
3490             zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
3491         spa = spa_add(pool, NULL, altroot);
3492         spa_activate(spa, spa_mode_global);
3493
3494         if (props && (error = spa_prop_validate(spa, props))) {
3495                 spa_deactivate(spa);
3496                 spa_remove(spa);
3497                 mutex_exit(&spa_namespace_lock);
3498                 return (error);
3499         }
3500
3501         has_features = B_FALSE;
3502         for (nvpair_t *elem = nvlist_next_nvpair(props, NULL);
3503             elem != NULL; elem = nvlist_next_nvpair(props, elem)) {
3504                 if (zpool_prop_feature(nvpair_name(elem)))
3505                         has_features = B_TRUE;
3506         }
3507
3508         if (has_features || nvlist_lookup_uint64(props,
3509             zpool_prop_to_name(ZPOOL_PROP_VERSION), &version) != 0) {
3510                 version = SPA_VERSION;
3511         }
3512         ASSERT(SPA_VERSION_IS_SUPPORTED(version));
3513
3514         spa->spa_first_txg = txg;
3515         spa->spa_uberblock.ub_txg = txg - 1;
3516         spa->spa_uberblock.ub_version = version;
3517         spa->spa_ubsync = spa->spa_uberblock;
3518
3519         /*
3520          * Create "The Godfather" zio to hold all async IOs
3521          */
3522         spa->spa_async_zio_root = kmem_alloc(max_ncpus * sizeof (void *),
3523             KM_SLEEP);
3524         for (int i = 0; i < max_ncpus; i++) {
3525                 spa->spa_async_zio_root[i] = zio_root(spa, NULL, NULL,
3526                     ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE |
3527                     ZIO_FLAG_GODFATHER);
3528         }
3529
3530         /*
3531          * Create the root vdev.
3532          */
3533         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3534
3535         error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, VDEV_ALLOC_ADD);
3536
3537         ASSERT(error != 0 || rvd != NULL);
3538         ASSERT(error != 0 || spa->spa_root_vdev == rvd);
3539
3540         if (error == 0 && !zfs_allocatable_devs(nvroot))
3541                 error = SET_ERROR(EINVAL);
```

```
3543         if (error == 0 &&
3544             (error = vdev_create(rvd, txg, B_FALSE)) == 0 &&
3545             (error = spa_validate_aux(spa, nvroot, txg,
3546             VDEV_ALLOC_ADD)) == 0) {
3547                 for (int c = 0; c < rvd->vdev_children; c++) {
3548                         vdev_metaslab_set_size(rvd->vdev_child[c]);
3549                         vdev_expand(rvd->vdev_child[c], txg);
3550                 }
3551         }

3553         spa_config_exit(spa, SCL_ALL, FTAG);

3555         if (error != 0) {
3556                 spa_unload(spa);
3557                 spa_deactivate(spa);
3558                 spa_remove(spa);
3559                 mutex_exit(&spa_namespace_lock);
3560                 return (error);
3561         }

3563         /*
3564          * Get the list of spares, if specified.
3565          */
3566         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
3567             &spares, &nspares) == 0) {
3568                 VERIFY(nvlist_alloc(&spa->spa_spares.sav_config, NV_UNIQUE_NAME,
3569                     KM_SLEEP) == 0);
3570                 VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
3571                     ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
3572                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3573                 spa_load_spares(spa);
3574                 spa_config_exit(spa, SCL_ALL, FTAG);
3575                 spa->spa_spares.sav_sync = B_TRUE;
3576         }

3578         /*
3579          * Get the list of level 2 cache devices, if specified.
3580          */
3581         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
3582             &l2cache, &nl2cache) == 0) {
3583                 VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config,
3584                     NV_UNIQUE_NAME, KM_SLEEP) == 0);
3585                 VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
3586                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3587                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3588                 spa_load_l2cache(spa);
3589                 spa_config_exit(spa, SCL_ALL, FTAG);
3590                 spa->spa_l2cache.sav_sync = B_TRUE;
3591         }

3593         spa->spa_is_initializing = B_TRUE;
3594         spa->spa_dsl_pool = dp = dsl_pool_create(spa, zplprops, txg);
3595         spa->spa_meta_objset = dp->dp_meta_objset;
3596         spa->spa_is_initializing = B_FALSE;

3598         /*
3599          * Create DDTs (dedup tables).
3600          */
3601         ddt_create(spa);

3603         spa_update_dspace(spa);

3605         tx = dmu_tx_create_assigned(dp, txg);

3607         /*
```



```
3608          * Create the pool config object.
3609          */
3610         spa->spa_config_object = dmu_object_alloc(spa->spa_meta_objset,
3611             DMU_OT_PACKED_NVLIST, SPA_CONFIG_BLOCKSIZE,
3612             DMU_OT_PACKED_NVLIST_SIZE, sizeof (uint64_t), tx);

3614         if (zap_add(spa->spa_meta_objset,
3615             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_CONFIG,
3616             sizeof (uint64_t), 1, &spa->spa_config_object, tx) != 0) {
3617                 cmn_err(CE_PANIC, "failed to add pool config");
3618         }

3620         if (spa_version(spa) >= SPA_VERSION_FEATURES)
3621                 spa_feature_create_zap_objects(spa, tx);

3623         if (zap_add(spa->spa_meta_objset,
3624             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_CREATION_VERSION,
3625             sizeof (uint64_t), 1, &version, tx) != 0) {
3626                 cmn_err(CE_PANIC, "failed to add pool version");
3627         }

3629         /* Newly created pools with the right version are always deflated. */
3630         if (version >= SPA_VERSION_RAIDZ_DEFLATE) {
3631                 spa->spa_deflate = TRUE;
3632                 if (zap_add(spa->spa_meta_objset,
3633                     DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_DEFLATE,
3634                     sizeof (uint64_t), 1, &spa->spa_deflate, tx) != 0) {
3635                         cmn_err(CE_PANIC, "failed to add deflate");
3636                 }
3637         }

3639         /*
3640          * Create the deferred-free bpobj.  Turn off compression
3641          * because sync-to-convergence takes longer if the blocksize
3642          * keeps changing.
3643          */
3644         obj = bpobj_alloc(spa->spa_meta_objset, 1 << 14, tx);
3645         dmu_object_set_compress(spa->spa_meta_objset, obj,
3646             ZIO_COMPRESS_OFF, tx);
3647         if (zap_add(spa->spa_meta_objset,
3648             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_SYNC_BPOBJ,
3649             sizeof (uint64_t), 1, &obj, tx) != 0) {
3650                 cmn_err(CE_PANIC, "failed to add bpobj");
3651         }
3652         VERIFY3U(0, ==, bpobj_open(&spa->spa_deferred_bpobj,
3653             spa->spa_meta_objset, obj));

3655         /*
3656          * Create the pool's history object.
3657          */
3658         if (version >= SPA_VERSION_ZPOOL_HISTORY)
3659                 spa_history_create_obj(spa, tx);

3661         /*
3662          * Set pool properties.
3663          */
3664         spa->spa_bootfs = zpool_prop_default_numeric(ZPOOL_PROP_BOOTFS);
3665         spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);
3666         spa->spa_failmode = zpool_prop_default_numeric(ZPOOL_PROP_FAILUREMODE);
3667         spa->spa_autoexpand = zpool_prop_default_numeric(ZPOOL_PROP_AUTOEXPAND);

3669         if (props != NULL) {
3670                 spa_configfile_set(spa, props, B_FALSE);
3671                 spa_sync_props(props, tx);
3672         }
```

```
3674         dmu_tx_commit(tx);

3676         spa->spa_sync_on = B_TRUE;
3677         txg_sync_start(spa->spa_dsl_pool);

3679         /*
3680          * We explicitly wait for the first transaction to complete so that our
3681          * bean counters are appropriately updated.
3682          */
3683         txg_wait_synced(spa->spa_dsl_pool, txg);

3685         spa_config_sync(spa, B_FALSE, B_TRUE);

3687         spa_history_log_version(spa, "create");

3689         /*
3690          * Don't count references from objsets that are already closed
3691          * and are making their way through the eviction process.
3692          */
3693         spa_evicting_os_wait(spa);
3694         spa->spa_minref = refcount_count(&spa->spa_refcount);

3696         mutex_exit(&spa_namespace_lock);

3698         return (0);
3699 }
```

_____*unchanged_portion_omitted_*

```
4179 /*
4180  * Pool export/destroy
4181  *
4182  * The act of destroying or exporting a pool is very simple.  We make sure there
4183  * is no more pending I/O and any references to the pool are gone.  Then, we
4184  * update the pool state and sync all the labels to disk, removing the
4185  * configuration from the cache afterwards. If the 'hardforce' flag is set, then
4186  * we don't sync the labels or remove the configuration cache.
4187  */
4188 static int
4189 spa_export_common(char *pool, int new_state, nvlist_t **oldconfig,
4190     boolean_t force, boolean_t hardforce)
4191 {
4192         spa_t *spa;

4194         if (oldconfig)
4195                 *oldconfig = NULL;

4197         if (!(spa_mode_global & FWRITE))
4198                 return (SET_ERROR(EROFS));

4200         mutex_enter(&spa_namespace_lock);
4201         if ((spa = spa_lookup(pool)) == NULL) {
4202                 mutex_exit(&spa_namespace_lock);
4203                 return (SET_ERROR(ENOENT));
4204         }

4206         /*
4207          * Put a hold on the pool, drop the namespace lock, stop async tasks,
4208          * reacquire the namespace lock, and see if we can export.
4209          */
4210         spa_open_ref(spa, FTAG);
4211         mutex_exit(&spa_namespace_lock);
4212         spa_async_suspend(spa);
4213         mutex_enter(&spa_namespace_lock);
4214         spa_close(spa, FTAG);

4216         /*
```



```
4217          * The pool will be in core if it's openable,
4218          * in which case we can modify its state.
4219          */
4220         if (spa->spa_state != POOL_STATE_UNINITIALIZED && spa->spa_sync_on) {
4221                 /*
4222                  * Objsets may be open only because they're dirty, so we
4223                  * have to force it to sync before checking spa_refcnt.
4224                  */
4225                 txg_wait_synced(spa->spa_dsl_pool, 0);
4226                 spa_evicting_os_wait(spa);

4228                 /*
4229                  * A pool cannot be exported or destroyed if there are active
4230                  * references.  If we are resetting a pool, allow references by
4231                  * fault injection handlers.
4232                  */
4233                 if (!spa_refcount_zero(spa) ||
4234                     (spa->spa_inject_ref != 0 &&
4235                     new_state != POOL_STATE_UNINITIALIZED)) {
4236                         spa_async_resume(spa);
4237                         mutex_exit(&spa_namespace_lock);
4238                         return (SET_ERROR(EBUSY));
4239                 }

4241                 /*
4242                  * A pool cannot be exported if it has an active shared spare.
4243                  * This is to prevent other pools stealing the active spare
4244                  * from an exported pool. At user's own will, such pool can
4245                  * be forcedly exported.
4246                  */
4247                 if (!force && new_state == POOL_STATE_EXPORTED &&
4248                     spa_has_active_shared_spare(spa)) {
4249                         spa_async_resume(spa);
4250                         mutex_exit(&spa_namespace_lock);
4251                         return (SET_ERROR(EXDEV));
4252                 }

4254                 /*
4255                  * We want this to be reflected on every label,
4256                  * so mark them all dirty.  spa_unload() will do the
4257                  * final sync that pushes these changes out.
4258                  */
4259                 if (new_state != POOL_STATE_UNINITIALIZED && !hardforce) {
4260                         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
4261                         spa->spa_state = new_state;
4262                         spa->spa_final_txg = spa_last_synced_txg(spa) +
4263                             TXG_DEFER_SIZE + 1;
4264                         vdev_config_dirty(spa->spa_root_vdev);
4265                         spa_config_exit(spa, SCL_ALL, FTAG);
4266                 }
4267         }

4269         spa_event_notify(spa, NULL, ESC_ZFS_POOL_DESTROY);

4271         if (spa->spa_state != POOL_STATE_UNINITIALIZED) {
4272                 spa_unload(spa);
4273                 spa_deactivate(spa);
4274         }

4276         if (oldconfig && spa->spa_config)
4277                 VERIFY(nvlist_dup(spa->spa_config, oldconfig, 0) == 0);

4279         if (new_state != POOL_STATE_UNINITIALIZED) {
4280                 if (!hardforce)
4281                         spa_config_sync(spa, B_TRUE, B_TRUE);
4282                 spa_remove(spa);
```

```
4283         }
4284         mutex_exit(&spa_namespace_lock);

4286         return (0);
4287 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   52352 Tue Jan  6 10:37:50 2015
new/usr/src/uts/common/fs/zfs/spa_misc.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2011, 2014 by Delphix. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/spa_impl.h>
#include <sys/spa_boot.h>
#include <sys/zio.h>
#include <sys/zio_checksum.h>
#include <sys/zio_compress.h>
#include <sys/dmu.h>
#include <sys/dmu_tx.h>
#include <sys/zap.h>
#include <sys/zil.h>
#include <sys/vdev_impl.h>
#include <sys/metaslab.h>
#include <sys/uberblock_impl.h>
#include <sys/txg.h>
#include <sys/avl.h>
#include <sys/unique.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_scan.h>
#include <sys/fs/zfs.h>
#include <sys/metaslab_impl.h>
#include <sys/arc.h>
#include <sys/ddt.h>
#include "zfs_prop.h"
#include "zfeature_common.h"

/*
 * SPA locking
 *
```



```
 * There are four basic locks for managing spa_t structures:
 *
 * spa_namespace_lock (global mutex)
 *
 *      This lock must be acquired to do any of the following:
 *
 *              - Lookup a spa_t by name
 *              - Add or remove a spa_t from the namespace
 *              - Increase spa_refcount from non-zero
 *              - Check if spa_refcount is zero
 *              - Rename a spa_t
 *              - add/remove/attach/detach devices
 *              - Held for the duration of create/destroy/import/export
 *
 *      It does not need to handle recursion.  A create or destroy may
 *      reference objects (files or zvols) in other pools, but by
 *      definition they must have an existing reference, and will never need
 *      to lookup a spa_t by name.
 *
 * spa_refcount (per-spa refcount_t protected by mutex)
 *
 *      This reference count keep track of any active users of the spa_t.  The
 *      spa_t cannot be destroyed or freed while this is non-zero.  Internally,
 *      the refcount is never really 'zero' - opening a pool implicitly keeps
 *      some references in the DMU.  Internally we check against spa_minref, but
 *      present the image of a zero/non-zero value to consumers.
 *
 * spa_config_lock[] (per-spa array of rwlocks)
 *
 *      This protects the spa_t from config changes, and must be held in
 *      the following circumstances:
 *
 *              - RW_READER to perform I/O to the spa
 *              - RW_WRITER to change the vdev config
 *
 * The locking order is fairly straightforward:
 *
 *              spa_namespace_lock      ->      spa_refcount
 *
 *      The namespace lock must be acquired to increase the refcount from 0
 *      or to check if it is zero.
 *
 *              spa_refcount            ->      spa_config_lock[]
 *
 *      There must be at least one valid reference on the spa_t to acquire
 *      the config lock.
 *
 *              spa_namespace_lock      ->      spa_config_lock[]
 *
 *      The namespace lock must always be taken before the config lock.
 *
 *
 * The spa_namespace_lock can be acquired directly and is globally visible.
 *
 * The namespace is manipulated using the following functions, all of which
 * require the spa_namespace_lock to be held.
 *
 *      spa_lookup()            Lookup a spa_t by name.
 *
 *      spa_add()               Create a new spa_t in the namespace.
 *
 *      spa_remove()            Remove a spa_t from the namespace.  This also
 *                              frees up any memory associated with the spa_t.
 *
 *      spa_next()              Returns the next spa_t in the system, or the
 *                              first if NULL is passed.
```

```
 124  *
 125  *      spa_evict_all()         Shutdown and remove all spa_t structures in
 126  *                              the system.
 127  *
 128  *      spa_guid_exists()       Determine whether a pool/device guid exists.
 129  *
 130  * The spa_refcount is manipulated using the following functions:
 131  *
 132  *      spa_open_ref()          Adds a reference to the given spa_t.  Must be
 133  *                              called with spa_namespace_lock held if the
 134  *                              refcount is currently zero.
 135  *
 136  *      spa_close()             Remove a reference from the spa_t.  This will
 137  *                              not free the spa_t or remove it from the
 138  *                              namespace.  No locking is required.
 139  *
 140  *      spa_refcount_zero()     Returns true if the refcount is currently
 141  *                              zero.  Must be called with spa_namespace_lock
 142  *                              held.
 143  *
 144  * The spa_config_lock[] is an array of rwlocks, ordered as follows:
 145  * SCL_CONFIG > SCL_STATE > SCL_ALLOC > SCL_ZIO > SCL_FREE > SCL_VDEV.
 146  * spa_config_lock[] is manipulated with spa_config_{enter,exit,held}().
 147  *
 148  * To read the configuration, it suffices to hold one of these locks as reader.
 149  * To modify the configuration, you must hold all locks as writer.  To modify
 150  * vdev state without altering the vdev tree's topology (e.g. online/offline),
 151  * you must hold SCL_STATE and SCL_ZIO as writer.
 152  *
 153  * We use these distinct config locks to avoid recursive lock entry.
 154  * For example, spa_sync() (which holds SCL_CONFIG as reader) induces
 155  * block allocations (SCL_ALLOC), which may require reading space maps
 156  * from disk (dmu_read() -> zio_read() -> SCL_ZIO).
 157  *
 158  * The spa config locks cannot be normal rwlocks because we need the
 159  * ability to hand off ownership.  For example, SCL_ZIO is acquired
 160  * by the issuing thread and later released by an interrupt thread.
 161  * They do, however, obey the usual write-wanted semantics to prevent
 162  * writer (i.e. system administrator) starvation.
 163  *
 164  * The lock acquisition rules are as follows:
 165  *
 166  * SCL_CONFIG
 167  *      Protects changes to the vdev tree topology, such as vdev
 168  *      add/remove/attach/detach.  Protects the dirty config list
 169  *      (spa_config_dirty_list) and the set of spares and l2arc devices.
 170  *
 171  * SCL_STATE
 172  *      Protects changes to pool state and vdev state, such as vdev
 173  *      online/offline/fault/degrade/clear.  Protects the dirty state list
 174  *      (spa_state_dirty_list) and global pool state (spa_state).
 175  *
 176  * SCL_ALLOC
 177  *      Protects changes to metaslab groups and classes.
 178  *      Held as reader by metaslab_alloc() and metaslab_claim().
 179  *
 180  * SCL_ZIO
 181  *      Held by bp-level zios (those which have no io_vd upon entry)
 182  *      to prevent changes to the vdev tree.  The bp-level zio implicitly
 183  *      protects all of its vdev child zios, which do not hold SCL_ZIO.
 184  *
 185  * SCL_FREE
 186  *      Protects changes to metaslab groups and classes.
 187  *      Held as reader by metaslab_free().  SCL_FREE is distinct from
 188  *      SCL_ALLOC, and lower than SCL_ZIO, so that we can safely free
 189  *      blocks in zio_done() while another i/o that holds either
```



```
 190  *      SCL_ALLOC or SCL_ZIO is waiting for this i/o to complete.
 191  *
 192  * SCL_VDEV
 193  *      Held as reader to prevent changes to the vdev tree during trivial
 194  *      inquiries such as bp_get_dsize().  SCL_VDEV is distinct from the
 195  *      other locks, and lower than all of them, to ensure that it's safe
 196  *      to acquire regardless of caller context.
 197  *
 198  * In addition, the following rules apply:
 199  *
 200  * (a)  spa_props_lock protects pool properties, spa_config and spa_config_list.
 201  *      The lock ordering is SCL_CONFIG > spa_props_lock.
 202  *
 203  * (b)  I/O operations on leaf vdevs.  For any zio operation that takes
 204  *      an explicit vdev_t argument -- such as zio_ioctl(), zio_read_phys(),
 205  *      or zio_write_phys() -- the caller must ensure that the config cannot
 206  *      cannot change in the interim, and that the vdev cannot be reopened.
 207  *      SCL_STATE as reader suffices for both.
 208  *
 209  * The vdev configuration is protected by spa_vdev_enter() / spa_vdev_exit().
 210  *
 211  *      spa_vdev_enter()        Acquire the namespace lock and the config lock
 212  *                              for writing.
 213  *
 214  *      spa_vdev_exit()         Release the config lock, wait for all I/O
 215  *                              to complete, sync the updated configs to the
 216  *                              cache, and release the namespace lock.
 217  *
 218  * vdev state is protected by spa_vdev_state_enter() / spa_vdev_state_exit().
 219  * Like spa_vdev_enter/exit, these are convenience wrappers -- the actual
 220  * locking is, always, based on spa_namespace_lock and spa_config_lock[].
 221  *
 222  * spa_rename() is also implemented within this file since it requires
 223  * manipulation of the namespace.
 224  */

 226 static avl_tree_t spa_namespace_avl;
 227 kmutex_t spa_namespace_lock;
 228 static kcondvar_t spa_namespace_cv;
 229 static int spa_active_count;
 230 int spa_max_replication_override = SPA_DVAS_PER_BP;

 232 static kmutex_t spa_spare_lock;
 233 static avl_tree_t spa_spare_avl;
 234 static kmutex_t spa_l2cache_lock;
 235 static avl_tree_t spa_l2cache_avl;

 237 kmem_cache_t *spa_buffer_pool;
 238 int spa_mode_global;

 240 #ifdef ZFS_DEBUG
 241 /* Everything except dprintf and spa is on by default in debug builds */
 242 int zfs_flags = ~(ZFS_DEBUG_DPRINTF | ZFS_DEBUG_SPA);
 243 #else
 244 int zfs_flags = 0;
 245 #endif

 247 /*
 248  * zfs_recover can be set to nonzero to attempt to recover from
 249  * otherwise-fatal errors, typically caused by on-disk corruption.  When
 250  * set, calls to zfs_panic_recover() will turn into warning messages.
 251  * This should only be used as a last resort, as it typically results
 252  * in leaked space, or worse.
 253  */
 254 boolean_t zfs_recover = B_FALSE;
```

```
256 /*
257  * If destroy encounters an EIO while reading metadata (e.g. indirect
258  * blocks), space referenced by the missing metadata can not be freed.
259  * Normally this causes the background destroy to become "stalled", as
260  * it is unable to make forward progress.  While in this stalled state,
261  * all remaining space to free from the error-encountering filesystem is
262  * "temporarily leaked".  Set this flag to cause it to ignore the EIO,
263  * permanently leak the space from indirect blocks that can not be read,
264  * and continue to free everything else that it can.
265  *
266  * The default, "stalling" behavior is useful if the storage partially
267  * fails (i.e. some but not all i/os fail), and then later recovers.  In
268  * this case, we will be able to continue pool operations while it is
269  * partially failed, and when it recovers, we can continue to free the
270  * space, with no leaks.  However, note that this case is actually
271  * fairly rare.
272  *
273  * Typically pools either (a) fail completely (but perhaps temporarily,
274  * e.g. a top-level vdev going offline), or (b) have localized,
275  * permanent errors (e.g. disk returns the wrong data due to bit flip or
276  * firmware bug).  In case (a), this setting does not matter because the
277  * pool will be suspended and the sync thread will not be able to make
278  * forward progress regardless.  In case (b), because the error is
279  * permanent, the best we can do is leak the minimum amount of space,
280  * which is what setting this flag will do.  Therefore, it is reasonable
281  * for this flag to normally be set, but we chose the more conservative
282  * approach of not setting it, so that there is no possibility of
283  * leaking space in the "partial temporary" failure case.
284  */
285 boolean_t zfs_free_leak_on_eio = B_FALSE;

287 /*
288  * Expiration time in milliseconds. This value has two meanings. First it is
289  * used to determine when the spa_deadman() logic should fire. By default the
290  * spa_deadman() will fire if spa_sync() has not completed in 1000 seconds.
291  * Secondly, the value determines if an I/O is considered "hung". Any I/O that
292  * has not completed in zfs_deadman_synctime_ms is considered "hung" resulting
293  * in a system panic.
294  */
295 uint64_t zfs_deadman_synctime_ms = 1000000ULL;

297 /*
298  * Check time in milliseconds. This defines the frequency at which we check
299  * for hung I/O.
300  */
301 uint64_t zfs_deadman_checktime_ms = 5000ULL;

303 /*
304  * Override the zfs deadman behavior via /etc/system. By default the
305  * deadman is enabled except on VMware and sparc deployments.
306  */
307 int zfs_deadman_enabled = -1;

309 /*
310  * The worst case is single-sector max-parity RAID-Z blocks, in which
311  * case the space requirement is exactly (VDEV_RAIDZ_MAXPARITY + 1)
312  * times the size; so just assume that.  Add to this the fact that
313  * we can have up to 3 DVAs per bp, and one more factor of 2 because
314  * the block may be dittoed with up to 3 DVAs by ddt_sync().  All together,
315  * the worst case is:
316  *     (VDEV_RAIDZ_MAXPARITY + 1) * SPA_DVAS_PER_BP * 2 == 24
317  */
318 int spa_asize_inflation = 24;

320 /*
321  * Normally, we don't allow the last 3.2% (1/(2^spa_slop_shift)) of space in
```



```
322  * the pool to be consumed.  This ensures that we don't run the pool
323  * completely out of space, due to unaccounted changes (e.g. to the MOS).
324  * It also limits the worst-case time to allocate space.  If we have
325  * less than this amount of free space, most ZPL operations (e.g. write,
326  * create) will return ENOSPC.
327  *
328  * Certain operations (e.g. file removal, most administrative actions) can
329  * use half the slop space.  They will only return ENOSPC if less than half
330  * the slop space is free.  Typically, once the pool has less than the slop
331  * space free, the user will use these operations to free up space in the pool.
332  * These are the operations that call dsl_pool_adjustedsize() with the netfree
333  * argument set to TRUE.
334  *
335  * A very restricted set of operations are always permitted, regardless of
336  * the amount of free space.  These are the operations that call
337  * dsl_sync_task(ZFS_SPACE_CHECK_NONE), e.g. "zfs destroy".  If these
338  * operations result in a net increase in the amount of space used,
339  * it is possible to run the pool completely out of space, causing it to
340  * be permanently read-only.
341  *
342  * See also the comments in zfs_space_check_t.
343  */
344 int spa_slop_shift = 5;

346 /*
347  * ==========================================================================
348  * SPA config locking
349  * ==========================================================================
350  */
351 static void
352 spa_config_lock_init(spa_t *spa)
353 {
354         for (int i = 0; i < SCL_LOCKS; i++) {
355                 spa_config_lock_t *scl = &spa->spa_config_lock[i];
356                 mutex_init(&scl->scl_lock, NULL, MUTEX_DEFAULT, NULL);
357                 cv_init(&scl->scl_cv, NULL, CV_DEFAULT, NULL);
358                 refcount_create_untracked(&scl->scl_count);
359                 scl->scl_writer = NULL;
360                 scl->scl_write_wanted = 0;
361         }
362 }
_____unchanged_portion_omitted_

535 /*
536  * Create an uninitialized spa_t with the given name.  Requires
537  * spa_namespace_lock.  The caller must ensure that the spa_t doesn't already
538  * exist by calling spa_lookup() first.
539  */
540 spa_t *
541 spa_add(const char *name, nvlist_t *config, const char *altroot)
542 {
543         spa_t *spa;
544         spa_config_dirent_t *dp;
545         cyc_handler_t hdlr;
546         cyc_time_t when;

548         ASSERT(MUTEX_HELD(&spa_namespace_lock));

550         spa = kmem_zalloc(sizeof (spa_t), KM_SLEEP);

552         mutex_init(&spa->spa_async_lock, NULL, MUTEX_DEFAULT, NULL);
553         mutex_init(&spa->spa_errlist_lock, NULL, MUTEX_DEFAULT, NULL);
554         mutex_init(&spa->spa_errlog_lock, NULL, MUTEX_DEFAULT, NULL);
555         mutex_init(&spa->spa_evicting_os_lock, NULL, MUTEX_DEFAULT, NULL);
556         mutex_init(&spa->spa_history_lock, NULL, MUTEX_DEFAULT, NULL);
557         mutex_init(&spa->spa_proc_lock, NULL, MUTEX_DEFAULT, NULL);
```

```
558         mutex_init(&spa->spa_props_lock, NULL, MUTEX_DEFAULT, NULL);
559         mutex_init(&spa->spa_scrub_lock, NULL, MUTEX_DEFAULT, NULL);
560         mutex_init(&spa->spa_suspend_lock, NULL, MUTEX_DEFAULT, NULL);
561         mutex_init(&spa->spa_vdev_top_lock, NULL, MUTEX_DEFAULT, NULL);
562         mutex_init(&spa->spa_iokstat_lock, NULL, MUTEX_DEFAULT, NULL);

564         cv_init(&spa->spa_async_cv, NULL, CV_DEFAULT, NULL);
565         cv_init(&spa->spa_evicting_os_cv, NULL, CV_DEFAULT, NULL);
566         cv_init(&spa->spa_proc_cv, NULL, CV_DEFAULT, NULL);
567         cv_init(&spa->spa_scrub_io_cv, NULL, CV_DEFAULT, NULL);
568         cv_init(&spa->spa_suspend_cv, NULL, CV_DEFAULT, NULL);

570         for (int t = 0; t < TXG_SIZE; t++)
571                 bplist_create(&spa->spa_free_bplist[t]);

573         (void) strlcpy(spa->spa_name, name, sizeof (spa->spa_name));
574         spa->spa_state = POOL_STATE_UNINITIALIZED;
575         spa->spa_freeze_txg = UINT64_MAX;
576         spa->spa_final_txg = UINT64_MAX;
577         spa->spa_load_max_txg = UINT64_MAX;
578         spa->spa_proc = &p0;
579         spa->spa_proc_state = SPA_PROC_NONE;

581         hdlr.cyh_func = spa_deadman;
582         hdlr.cyh_arg = spa;
583         hdlr.cyh_level = CY_LOW_LEVEL;

585         spa->spa_deadman_synctime = MSEC2NSEC(zfs_deadman_synctime_ms);

587         /*
588          * This determines how often we need to check for hung I/Os after
589          * the cyclic has already fired. Since checking for hung I/Os is
590          * an expensive operation we don't want to check too frequently.
591          * Instead wait for 5 seconds before checking again.
592          */
593         when.cyt_interval = MSEC2NSEC(zfs_deadman_checktime_ms);
594         when.cyt_when = CY_INFINITY;
595         mutex_enter(&cpu_lock);
596         spa->spa_deadman_cycid = cyclic_add(&hdlr, &when);
597         mutex_exit(&cpu_lock);

599         refcount_create(&spa->spa_refcount);
600         spa_config_lock_init(spa);

602         avl_add(&spa_namespace_avl, spa);

604         /*
605          * Set the alternate root, if there is one.
606          */
607         if (altroot) {
608                 spa->spa_root = spa_strdup(altroot);
609                 spa_active_count++;
610         }

612         /*
613          * Every pool starts with the default cachefile
614          */
615         list_create(&spa->spa_config_list, sizeof (spa_config_dirent_t),
616             offsetof(spa_config_dirent_t, scd_link));

618         dp = kmem_zalloc(sizeof (spa_config_dirent_t), KM_SLEEP);
619         dp->scd_path = altroot ? NULL : spa_strdup(spa_config_path);
620         list_insert_head(&spa->spa_config_list, dp);

622         VERIFY(nvlist_alloc(&spa->spa_load_info, NV_UNIQUE_NAME,
623             KM_SLEEP) == 0);
```



```
625         if (config != NULL) {
626                 nvlist_t *features;

628                 if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_FEATURES_FOR_READ,
629                     &features) == 0) {
630                         VERIFY(nvlist_dup(features, &spa->spa_label_features,
631                             0) == 0);
632                 }

634                 VERIFY(nvlist_dup(config, &spa->spa_config, 0) == 0);
635         }

637         if (spa->spa_label_features == NULL) {
638                 VERIFY(nvlist_alloc(&spa->spa_label_features, NV_UNIQUE_NAME,
639                     KM_SLEEP) == 0);
640         }

642         spa->spa_iokstat = kstat_create("zfs", 0, name,
643             "disk", KSTAT_TYPE_IO, 1, 0);
644         if (spa->spa_iokstat) {
645                 spa->spa_iokstat->ks_lock = &spa->spa_iokstat_lock;
646                 kstat_install(spa->spa_iokstat);
647         }

649         spa->spa_debug = ((zfs_flags & ZFS_DEBUG_SPA) != 0);

651         /*
652          * As a pool is being created, treat all features as disabled by
653          * setting SPA_FEATURE_DISABLED for all entries in the feature
654          * refcount cache.
655          */
656         for (int i = 0; i < SPA_FEATURES; i++) {
657                 spa->spa_feat_refcount_cache[i] = SPA_FEATURE_DISABLED;
658         }

660         return (spa);
661 }

663 /*
664  * Removes a spa_t from the namespace, freeing up any memory used.  Requires
665  * spa_namespace_lock.  This is called only after the spa_t has been closed and
666  * deactivated.
667  */
668 void
669 spa_remove(spa_t *spa)
670 {
671         spa_config_dirent_t *dp;

673         ASSERT(MUTEX_HELD(&spa_namespace_lock));
674         ASSERT(spa->spa_state == POOL_STATE_UNINITIALIZED);
675         ASSERT3U(refcount_count(&spa->spa_refcount), ==, 0);

677         nvlist_free(spa->spa_config_splitting);

679         avl_remove(&spa_namespace_avl, spa);
680         cv_broadcast(&spa_namespace_cv);

682         if (spa->spa_root) {
683                 spa_strfree(spa->spa_root);
684                 spa_active_count--;
685         }

687         while ((dp = list_head(&spa->spa_config_list)) != NULL) {
688                 list_remove(&spa->spa_config_list, dp);
689                 if (dp->scd_path != NULL)
```

```
690                         spa_strfree(dp->scd_path);
691                 kmem_free(dp, sizeof (spa_config_dirent_t));
692         }

694         list_destroy(&spa->spa_config_list);

696         nvlist_free(spa->spa_label_features);
697         nvlist_free(spa->spa_load_info);
698         spa_config_set(spa, NULL);

700         mutex_enter(&cpu_lock);
701         if (spa->spa_deadman_cycid != CYCLIC_NONE)
702                 cyclic_remove(spa->spa_deadman_cycid);
703         mutex_exit(&cpu_lock);
704         spa->spa_deadman_cycid = CYCLIC_NONE;

706         refcount_destroy(&spa->spa_refcount);

708         spa_config_lock_destroy(spa);

710         kstat_delete(spa->spa_iokstat);
711         spa->spa_iokstat = NULL;

713         for (int t = 0; t < TXG_SIZE; t++)
714                 bplist_destroy(&spa->spa_free_bplist[t]);

716         cv_destroy(&spa->spa_async_cv);
717         cv_destroy(&spa->spa_evicting_os_cv);
718         cv_destroy(&spa->spa_proc_cv);
719         cv_destroy(&spa->spa_scrub_io_cv);
720         cv_destroy(&spa->spa_suspend_cv);

722         mutex_destroy(&spa->spa_async_lock);
723         mutex_destroy(&spa->spa_errlist_lock);
724         mutex_destroy(&spa->spa_errlog_lock);
725         mutex_destroy(&spa->spa_evicting_os_lock);
726         mutex_destroy(&spa->spa_history_lock);
727         mutex_destroy(&spa->spa_proc_lock);
728         mutex_destroy(&spa->spa_props_lock);
729         mutex_destroy(&spa->spa_scrub_lock);
730         mutex_destroy(&spa->spa_suspend_lock);
731         mutex_destroy(&spa->spa_vdev_top_lock);
732         mutex_destroy(&spa->spa_iokstat_lock);

734         kmem_free(spa, sizeof (spa_t));
735 }
```

*unchanged_portion_omitted_*

```
782 /*
783  * Remove a reference to the given spa_t held by a dsl dir that is
784  * being asynchronously released.  Async releases occur from a taskq
785  * performing eviction of dsl datasets and dirs.  The namespace lock
786  * isn't held and the hold by the object being evicted may contribute to
787  * spa_minref (e.g. dataset or directory released during pool export),
788  * so the asserts in spa_close() do not apply.
789  */
790 void
791 spa_async_close(spa_t *spa, void *tag)
792 {
793         (void) refcount_remove(&spa->spa_refcount, tag);
794 }

796 /*
797  * Check to see if the spa refcount is zero.  Must be called with
798  * spa_namespace_lock held.  We really compare against spa_minref, which is the
799  * number of references acquired when opening a pool
```



```
800  */
801 boolean_t
802 spa_refcount_zero(spa_t *spa)
803 {
804         ASSERT(MUTEX_HELD(&spa_namespace_lock));

806         return (refcount_count(&spa->spa_refcount) == spa->spa_minref);
807 }
```

*unchanged_portion_omitted_*

```
1692 void
1693 spa_evicting_os_register(spa_t *spa, objset_t *os)
1694 {
1695         mutex_enter(&spa->spa_evicting_os_lock);
1696         list_insert_head(&spa->spa_evicting_os_list, os);
1697         mutex_exit(&spa->spa_evicting_os_lock);
1698 }

1700 void
1701 spa_evicting_os_deregister(spa_t *spa, objset_t *os)
1702 {
1703         mutex_enter(&spa->spa_evicting_os_lock);
1704         list_remove(&spa->spa_evicting_os_list, os);
1705         cv_broadcast(&spa->spa_evicting_os_cv);
1706         mutex_exit(&spa->spa_evicting_os_lock);
1707 }

1709 void
1710 spa_evicting_os_wait(spa_t *spa)
1711 {
1712         mutex_enter(&spa->spa_evicting_os_lock);
1713         while (!list_is_empty(&spa->spa_evicting_os_list))
1714                 cv_wait(&spa->spa_evicting_os_cv, &spa->spa_evicting_os_lock);
1715         mutex_exit(&spa->spa_evicting_os_lock);

1717         dmu_buf_user_evict_wait();
1718 }

1720 int
1721 spa_max_replication(spa_t *spa)
1722 {
1723         /*
1724          * As of SPA_VERSION == SPA_VERSION_DITTO_BLOCKS, we are able to
1725          * handle BPs with more than one DVA allocated.  Set our max
1726          * replication level accordingly.
1727          */
1728         if (spa_version(spa) < SPA_VERSION_DITTO_BLOCKS)
1729                 return (1);
1730         return (MIN(SPA_DVAS_PER_BP, spa_max_replication_override));
1731 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   10311 Tue Jan  6 10:37:50 2015
new/usr/src/uts/common/fs/zfs/sys/dbuf.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#ifndef _SYS_DBUF_H
#define _SYS_DBUF_H

#include <sys/dmu.h>
#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/zio.h>
#include <sys/arc.h>
#include <sys/zfs_context.h>
#include <sys/refcount.h>
#include <sys/zrlock.h>

#ifdef  __cplusplus
extern "C" {
#endif

#define IN_DMU_SYNC 2

/*
 * define flags for dbuf_read
 */

#define DB_RF_MUST_SUCCEED      (1 << 0)
#define DB_RF_CANFAIL           (1 << 1)
#define DB_RF_HAVESTRUCT        (1 << 2)
#define DB_RF_NOPREFETCH        (1 << 3)
#define DB_RF_NEVERWAIT         (1 << 4)
#define DB_RF_CACHED            (1 << 5)

/*
```



```
 * The simplified state transition diagram for dbufs looks like:
 *
 *              +----> READ ----+
 *              |               |
 *              |               V
 *  (alloc)-->UNCACHED       CACHED-->EVICTING-->(free)
 *              |               ^        ^
 *              |               |        |
 *              +----> FILL ----+        |
 *              |                        |
 *              |                        |
 *              +--------> NOFILL -------+
 *
 * DB_SEARCH is an invalid state for a dbuf. It is used by dbuf_free_range
 * to find all dbufs in a range of a dnode and must be less than any other
 * dbuf_states_t (see comment on dn_dbufs in dnode.h).
 */
typedef enum dbuf_states {
        DB_SEARCH = -1,
        DB_UNCACHED,
        DB_FILL,
        DB_NOFILL,
        DB_READ,
        DB_CACHED,
        DB_EVICTING
} dbuf_states_t;
_____unchanged_portion_omitted_

typedef struct dmu_buf_impl {
        /*
         * The following members are immutable, with the exception of
         * db.db_data, which is protected by db_mtx.
         */

        /* the publicly visible structure */
        dmu_buf_t db;

        /* the objset we belong to */
        struct objset *db_objset;

        /*
         * handle to safely access the dnode we belong to (NULL when evicted)
         */
        struct dnode_handle *db_dnode_handle;

        /*
         * our parent buffer; if the dnode points to us directly,
         * db_parent == db_dnode_handle->dnh_dnode->dn_dbuf
         * only accessed by sync thread ???
         * (NULL when evicted)
         * May change from NULL to non-NULL under the protection of db_mtx
         * (see dbuf_check_blkptr())
         */
        struct dmu_buf_impl *db_parent;

        /*
         * link for hash table of all dmu_buf_impl_t's
         */
        struct dmu_buf_impl *db_hash_next;

        /* our block number */
        uint64_t db_blkid;

        /*
         * Pointer to the blkptr_t which points to us. May be NULL if we
         * don't have one yet. (NULL when evicted)
```

```
187          */
188         blkptr_t *db_blkptr;

190         /*
191          * Our indirection level.  Data buffers have db_level==0.
192          * Indirect buffers which point to data buffers have
193          * db_level==1. etc.  Buffers which contain dnodes have
194          * db_level==0, since the dnodes are stored in a file.
195          */
196         uint8_t db_level;

198         /* db_mtx protects the members below */
199         kmutex_t db_mtx;

201         /*
202          * Current state of the buffer
203          */
204         dbuf_states_t db_state;

206         /*
207          * Refcount accessed by dmu_buf_{hold,rele}.
208          * If nonzero, the buffer can't be destroyed.
209          * Protected by db_mtx.
210          */
211         refcount_t db_holds;

213         /* buffer holding our data */
214         arc_buf_t *db_buf;

216         kcondvar_t db_changed;
217         dbuf_dirty_record_t *db_data_pending;

219         /* pointer to most recent dirty record for this buffer */
220         dbuf_dirty_record_t *db_last_dirty;

222         /*
223          * Our link on the owner dnodes's dn_dbufs list.
224          * Protected by its dn_dbufs_mtx.
225          */
226         avl_node_t db_link;

228         /* Data which is unique to data (leaf) blocks: */

230         /* User callback information. */
231         dmu_buf_user_t *db_user;
229         /* stuff we store for the user (see dmu_buf_set_user) */
230         void *db_user_ptr;
231         dmu_buf_evict_func_t *db_evict_func;

233         uint8_t db_immediate_evict;
234         uint8_t db_freed_in_flight;

236         uint8_t db_dirtycnt;
237 } dmu_buf_impl_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   32618 Tue Jan  6 10:37:51 2015
new/usr/src/uts/common/fs/zfs/sys/dmu.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2011, 2014 by Delphix. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright 2013 DEY Storage Systems, Inc.
 * Copyright 2014 HybridCluster. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

/* Portions Copyright 2010 Robert Milkowski */

#ifndef _SYS_DMU_H
#define _SYS_DMU_H

/*
 * This file describes the interface that the DMU provides for its
 * consumers.
 *
 * The DMU also interacts with the SPA.  That interface is described in
 * dmu_spa.h.
 */

#include <sys/zfs_context.h>
#include <sys/inttypes.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/cred.h>
#include <sys/time.h>
#include <sys/fs/zfs.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct uio;
```



```
struct xuio;
struct page;
struct vnode;
struct spa;
struct zilog;
struct zio;
struct blkptr;
struct zap_cursor;
struct dsl_dataset;
struct dsl_pool;
struct dnode;
struct drr_begin;
struct drr_end;
struct zbookmark_phys;
struct spa;
struct nvlist;
struct arc_buf;
struct zio_prop;
struct sa_handle;

typedef struct objset objset_t;
typedef struct dmu_tx dmu_tx_t;
typedef struct dsl_dir dsl_dir_t;

typedef enum dmu_object_byteswap {
        DMU_BSWAP_UINT8,
        DMU_BSWAP_UINT16,
        DMU_BSWAP_UINT32,
        DMU_BSWAP_UINT64,
        DMU_BSWAP_ZAP,
        DMU_BSWAP_DNODE,
        DMU_BSWAP_OBJSET,
        DMU_BSWAP_ZNODE,
        DMU_BSWAP_OLDACL,
        DMU_BSWAP_ACL,
        /*
         * Allocating a new byteswap type number makes the on-disk format
         * incompatible with any other format that uses the same number.
         *
         * Data can usually be structured to work with one of the
         * DMU_BSWAP_UINT* or DMU_BSWAP_ZAP types.
         */
        DMU_BSWAP_NUMFUNCS
} dmu_object_byteswap_t;
```

*_____unchanged_portion_omitted_*

```
typedef void dmu_buf_evict_func_t(struct dmu_buf *db, void *user_ptr);

/*
 * The names of zap entries in the DIRECTORY_OBJECT of the MOS.
 */
#define DMU_POOL_DIRECTORY_OBJECT       1
#define DMU_POOL_CONFIG                 "config"
#define DMU_POOL_FEATURES_FOR_WRITE     "features_for_write"
#define DMU_POOL_FEATURES_FOR_READ      "features_for_read"
#define DMU_POOL_FEATURE_DESCRIPTIONS   "feature_descriptions"
#define DMU_POOL_FEATURE_ENABLED_TXG    "feature_enabled_txg"
#define DMU_POOL_ROOT_DATASET           "root_dataset"
#define DMU_POOL_SYNC_BPOBJ             "sync_bplist"
#define DMU_POOL_ERRLOG_SCRUB           "errlog_scrub"
#define DMU_POOL_ERRLOG_LAST            "errlog_last"
#define DMU_POOL_SPARES                 "spares"
#define DMU_POOL_DEFLATE                "deflate"
#define DMU_POOL_HISTORY                "history"
#define DMU_POOL_PROPS                  "pool_props"
#define DMU_POOL_L2CACHE                "l2cache"
```

```
311 #define DMU_POOL_TMP_USERREFS           "tmp_userrefs"
312 #define DMU_POOL_DDT                    "DDT-%s-%s-%s"
313 #define DMU_POOL_DDT_STATS              "DDT-statistics"
314 #define DMU_POOL_CREATION_VERSION       "creation_version"
315 #define DMU_POOL_SCAN                   "scan"
316 #define DMU_POOL_FREE_BPOBJ             "free_bpobj"
317 #define DMU_POOL_BPTREE_OBJ             "bptree_obj"
318 #define DMU_POOL_EMPTY_BPOBJ            "empty_bpobj"

320 /*
321  * Allocate an object from this objset.  The range of object numbers
322  * available is (0, DN_MAX_OBJECT).  Object 0 is the meta-dnode.
323  *
324  * The transaction must be assigned to a txg.  The newly allocated
325  * object will be "held" in the transaction (ie. you can modify the
326  * newly allocated object in this transaction).
327  *
328  * dmu_object_alloc() chooses an object and returns it in *objectp.
329  *
330  * dmu_object_claim() allocates a specific object number.  If that
331  * number is already allocated, it fails and returns EEXIST.
332  *
333  * Return 0 on success, or ENOSPC or EEXIST as specified above.
334  */
335 uint64_t dmu_object_alloc(objset_t *os, dmu_object_type_t ot,
336     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
337 int dmu_object_claim(objset_t *os, uint64_t object, dmu_object_type_t ot,
338     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
339 int dmu_object_reclaim(objset_t *os, uint64_t object, dmu_object_type_t ot,
340     int blocksize, dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *txp);

342 /*
343  * Free an object from this objset.
344  *
345  * The object's data will be freed as well (ie. you don't need to call
346  * dmu_free(object, 0, -1, tx)).
347  *
348  * The object need not be held in the transaction.
349  *
350  * If there are any holds on this object's buffers (via dmu_buf_hold()),
351  * or tx holds on the object (via dmu_tx_hold_object()), you can not
352  * free it; it fails and returns EBUSY.
353  *
354  * If the object is not allocated, it fails and returns ENOENT.
355  *
356  * Return 0 on success, or EBUSY or ENOENT as specified above.
357  */
358 int dmu_object_free(objset_t *os, uint64_t object, dmu_tx_t *tx);

360 /*
361  * Find the next allocated or free object.
362  *
363  * The objectp parameter is in-out.  It will be updated to be the next
364  * object which is allocated.  Ignore objects which have not been
365  * modified since txg.
366  *
367  * XXX Can only be called on a objset with no dirty data.
368  *
369  * Returns 0 on success, or ENOENT if there are no more objects.
370  */
371 int dmu_object_next(objset_t *os, uint64_t *objectp,
372     boolean_t hole, uint64_t txg);

374 /*
375  * Set the data blocksize for an object.
376  *
```



```
377  * The object cannot have any blocks allcated beyond the first.  If
378  * the first block is allocated already, the new size must be greater
379  * than the current block size.  If these conditions are not met,
380  * ENOTSUP will be returned.
381  *
382  * Returns 0 on success, or EBUSY if there are any holds on the object
383  * contents, or ENOTSUP as described above.
384  */
385 int dmu_object_set_blocksize(objset_t *os, uint64_t object, uint64_t size,
386     int ibs, dmu_tx_t *tx);

388 /*
389  * Set the checksum property on a dnode.  The new checksum algorithm will
390  * apply to all newly written blocks; existing blocks will not be affected.
391  */
392 void dmu_object_set_checksum(objset_t *os, uint64_t object, uint8_t checksum,
393     dmu_tx_t *tx);

395 /*
396  * Set the compress property on a dnode.  The new compression algorithm will
397  * apply to all newly written blocks; existing blocks will not be affected.
398  */
399 void dmu_object_set_compress(objset_t *os, uint64_t object, uint8_t compress,
400     dmu_tx_t *tx);

402 void
403 dmu_write_embedded(objset_t *os, uint64_t object, uint64_t offset,
404     void *data, uint8_t etype, uint8_t comp, int uncompressed_size,
405     int compressed_size, int byteorder, dmu_tx_t *tx);

407 /*
408  * Decide how to write a block: checksum, compression, number of copies, etc.
409  */
410 #define WP_NOFILL       0x1
411 #define WP_DMU_SYNC     0x2
412 #define WP_SPILL        0x4

414 void dmu_write_policy(objset_t *os, struct dnode *dn, int level, int wp,
415     struct zio_prop *zp);
416 /*
417  * The bonus data is accessed more or less like a regular buffer.
418  * You must dmu_bonus_hold() to get the buffer, which will give you a
419  * dmu_buf_t with db_offset==-1ULL, and db_size = the size of the bonus
420  * data.  As with any normal buffer, you must call dmu_buf_read() to
421  * read db_data, dmu_buf_will_dirty() before modifying it, and the
422  * object must be held in an assigned transaction before calling
423  * dmu_buf_will_dirty.  You may use dmu_buf_set_user() on the bonus
424  * buffer as well.  You must release your hold with dmu_buf_rele().
425  *
426  * Returns ENOENT, EIO, or 0.
427  */
428 int dmu_bonus_hold(objset_t *os, uint64_t object, void *tag, dmu_buf_t **);
429 int dmu_bonus_max(void);
430 int dmu_set_bonus(dmu_buf_t *, int, dmu_tx_t *);
431 int dmu_set_bonustype(dmu_buf_t *, dmu_object_type_t, dmu_tx_t *);
432 dmu_object_type_t dmu_get_bonustype(dmu_buf_t *);
433 int dmu_rm_spill(objset_t *, uint64_t, dmu_tx_t *);

435 /*
436  * Special spill buffer support used by "SA" framework
437  */

439 int dmu_spill_hold_by_bonus(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp);
440 int dmu_spill_hold_by_dnode(struct dnode *dn, uint32_t flags,
441     void *tag, dmu_buf_t **dbp);
442 int dmu_spill_hold_existing(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp);
```

```
 444 /*
 445  * Obtain the DMU buffer from the specified object which contains the
 446  * specified offset.  dmu_buf_hold() puts a "hold" on the buffer, so
 447  * that it will remain in memory.  You must release the hold with
 448  * dmu_buf_rele().  You musn't access the dmu_buf_t after releasing your
 449  * hold.  You must have a hold on any dmu_buf_t* you pass to the DMU.
 450  *
 451  * You must call dmu_buf_read, dmu_buf_will_dirty, or dmu_buf_will_fill
 452  * on the returned buffer before reading or writing the buffer's
 453  * db_data.  The comments for those routines describe what particular
 454  * operations are valid after calling them.
 455  *
 456  * The object number must be a valid, allocated object number.
 457  */
 458 int dmu_buf_hold(objset_t *os, uint64_t object, uint64_t offset,
 459     void *tag, dmu_buf_t **, int flags);
 460 void dmu_buf_add_ref(dmu_buf_t *db, void* tag);
 461 void dmu_buf_rele(dmu_buf_t *db, void *tag);
 462 uint64_t dmu_buf_refcount(dmu_buf_t *db);

 464 /*
 465  * dmu_buf_hold_array holds the DMU buffers which contain all bytes in a
 466  * range of an object.  A pointer to an array of dmu_buf_t*'s is
 467  * returned (in *dbpp).
 468  *
 469  * dmu_buf_rele_array releases the hold on an array of dmu_buf_t*'s, and
 470  * frees the array.  The hold on the array of buffers MUST be released
 471  * with dmu_buf_rele_array.  You can NOT release the hold on each buffer
 472  * individually with dmu_buf_rele.
 473  */
 474 int dmu_buf_hold_array_by_bonus(dmu_buf_t *db, uint64_t offset,
 475     uint64_t length, int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp);
 476 void dmu_buf_rele_array(dmu_buf_t **, int numbufs, void *tag);

 478 typedef void dmu_buf_evict_func_t(void *user_ptr);

 480 /*
 481  * A DMU buffer user object may be associated with a dbuf for the
 482  * duration of its lifetime.  This allows the user of a dbuf (client)
 483  * to attach private data to a dbuf (e.g. in-core only data such as a
 484  * dnode_children_t, zap_t, or zap_leaf_t) and be optionally notified
 485  * when that dbuf has been evicted.  Clients typically respond to the
 486  * eviction notification by freeing their private data, thus ensuring
 487  * the same lifetime for both dbuf and private data.
 482  * Returns NULL on success, or the existing user ptr if it's already
 483  * been set.
 488  *
 489  * The mapping from a dmu_buf_user_t to any client private data is the
 490  * client's responsibility.  All current consumers of the API with private
 491  * data embed a dmu_buf_user_t as the first member of the structure for
 492  * their private data.  This allows conversions between the two types
 493  * with a simple cast.  Since the DMU buf user API never needs access
 494  * to the private data, other strategies can be employed if necessary
 495  * or convenient for the client (e.g. using container_of() to do the
 496  * conversion for private data that cannot have the dmu_buf_user_t as
 497  * its first member).
 485  * user_ptr is for use by the user and can be obtained via dmu_buf_get_user().
 498  *
 499  * Eviction callbacks are executed without the dbuf mutex held or any
 500  * other type of mechanism to guarantee that the dbuf is still available.
 501  * For this reason, users must assume the dbuf has already been freed
 502  * and not reference the dbuf from the callback context.
 487  * If non-NULL, pageout func will be called when this buffer is being
 488  * excised from the cache, so that you can clean up the data structure
 489  * pointed to by user_ptr.
```



```
 503  *
 504  * Users requesting "immediate eviction" are notified as soon as the dbuf
 505  * is only referenced by dirty records (dirties == holds).  Otherwise the
 506  * notification occurs after eviction processing for the dbuf begins.
 491  * dmu_evict_user() will call the pageout func for all buffers in a
 492  * objset with a given pageout func.
 507  */
 508 typedef struct dmu_buf_user {
 509         /*
 510          * Asynchronous user eviction callback state.
 511          */
 512         taskq_ent_t     dbu_tqent;

 514         /* This instance's eviction function pointer. */
 515         dmu_buf_evict_func_t *dbu_evict_func;
 516 #ifdef ZFS_DEBUG
 517         /*
 518          * Pointer to user's dbuf pointer.  NULL for clients that do
 519          * not associate a dbuf with their user data.
 520          *
 521          * The dbuf pointer is cleared upon eviction so as to catch
 522          * use-after-evict bugs in clients.
 523          */
 524         dmu_buf_t **dbu_clear_on_evict_dbufp;
 525 #endif
 526 } dmu_buf_user_t;

 494 void *dmu_buf_set_user(dmu_buf_t *db, void *user_ptr,
 495     dmu_buf_evict_func_t *pageout_func);
 528 /*
 529  * Initialize the given dmu_buf_user_t instance with the eviction function
 530  * evict_func, to be called when the user is evicted.
 531  *
 532  * NOTE: This function should only be called once on a given dmu_buf_user_t.
 533  *       To allow enforcement of this, dbu must already be zeroed on entry.
 497  * set_user_ie is the same as set_user, but request immediate eviction
 498  * when hold count goes to zero.
 534  */
 535 #ifdef __lint
 536 /* Very ugly, but it beats issuing suppression directives in many Makefiles. */
 537 extern void
 538 dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func,
 539     dmu_buf_t **clear_on_evict_dbufp);
 540 #else /* __lint */
 541 inline void
 542 dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func,
 543     dmu_buf_t **clear_on_evict_dbufp)
 544 {
 545         ASSERT(dbu->dbu_evict_func == NULL);
 546         ASSERT(evict_func != NULL);
 547         dbu->dbu_evict_func = evict_func;
 548 #ifdef ZFS_DEBUG
 549         dbu->dbu_clear_on_evict_dbufp = clear_on_evict_dbufp;
 550 #endif
 551 }
 552 #endif /* __lint */
 500 void *dmu_buf_set_user_ie(dmu_buf_t *db, void *user_ptr,
 501     dmu_buf_evict_func_t *pageout_func);
 502 void *dmu_buf_update_user(dmu_buf_t *db_fake, void *old_user_ptr,
 503     void *user_ptr, dmu_buf_evict_func_t *pageout_func);
 504 void dmu_evict_user(objset_t *os, dmu_buf_evict_func_t *func);

 554 /*
 555  * Attach user data to a dbuf and mark it for normal (when the dbuf's
 556  * data is cleared or its reference count goes to zero) eviction processing.
 557  *
```

```
 * Returns NULL on success, or the existing user if another user currently
 * owns the buffer.
 * Returns the user_ptr set with dmu_buf_set_user(), or NULL if not set.
 */
void *dmu_buf_set_user(dmu_buf_t *db, dmu_buf_user_t *user);

/*
 * Attach user data to a dbuf and mark it for immediate (its dirty and
 * reference counts are equal) eviction processing.
 *
 * Returns NULL on success, or the existing user if another user currently
 * owns the buffer.
 */
void *dmu_buf_set_user_ie(dmu_buf_t *db, dmu_buf_user_t *user);

/*
 * Replace the current user of a dbuf.
 *
 * If given the current user of a dbuf, replaces the dbuf's user with
 * "new_user" and returns the user data pointer that was replaced.
 * Otherwise returns the current, and unmodified, dbuf user pointer.
 */
void *dmu_buf_replace_user(dmu_buf_t *db,
    dmu_buf_user_t *old_user, dmu_buf_user_t *new_user);

/*
 * Remove the specified user data for a DMU buffer.
 *
 * Returns the user that was removed on success, or the current user if
 * another user currently owns the buffer.
 */
void *dmu_buf_remove_user(dmu_buf_t *db, dmu_buf_user_t *user);

/*
 * Returns the user data (dmu_buf_user_t *) associated with this dbuf.
 */
void *dmu_buf_get_user(dmu_buf_t *db);

/* Block until any in-progress dmu buf user evictions complete. */
void dmu_buf_user_evict_wait(void);

/*
 * Returns the blkptr associated with this dbuf, or NULL if not set.
 */
struct blkptr *dmu_buf_get_blkptr(dmu_buf_t *db);

/*
 * Indicate that you are going to modify the buffer's data (db_data).
 *
 * The transaction (tx) must be assigned to a txg (ie. you've called
 * dmu_tx_assign()).  The buffer's object must be held in the tx
 * (ie. you've called dmu_tx_hold_object(tx, db->db_object)).
 */
void dmu_buf_will_dirty(dmu_buf_t *db, dmu_tx_t *tx);

/*
 * Tells if the given dbuf is freeable.
 */
boolean_t dmu_buf_freeable(dmu_buf_t *);

/*
 * You must create a transaction, then hold the objects which you will
 * (or might) modify as part of this transaction.  Then you must assign
 * the transaction to a transaction group.  Once the transaction has
 * been assigned, you can modify buffers which belong to held objects as
 * part of this transaction.  You can't modify buffers before the
```



```
 * transaction has been assigned; you can't modify buffers which don't
 * belong to objects which this transaction holds; you can't hold
 * objects once the transaction has been assigned.  You may hold an
 * object which you are going to free (with dmu_object_free()), but you
 * don't have to.
 *
 * You can abort the transaction before it has been assigned.
 *
 * Note that you may hold buffers (with dmu_buf_hold) at any time,
 * regardless of transaction state.
 */

#define DMU_NEW_OBJECT  (-1ULL)
#define DMU_OBJECT_END  (-1ULL)

dmu_tx_t *dmu_tx_create(objset_t *os);
void dmu_tx_hold_write(dmu_tx_t *tx, uint64_t object, uint64_t off, int len);
void dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off,
    uint64_t len);
void dmu_tx_hold_zap(dmu_tx_t *tx, uint64_t object, int add, const char *name);
void dmu_tx_hold_bonus(dmu_tx_t *tx, uint64_t object);
void dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object);
void dmu_tx_hold_sa(dmu_tx_t *tx, struct sa_handle *hdl, boolean_t may_grow);
void dmu_tx_hold_sa_create(dmu_tx_t *tx, int total_size);
void dmu_tx_abort(dmu_tx_t *tx);
int dmu_tx_assign(dmu_tx_t *tx, enum txg_how txg_how);
void dmu_tx_wait(dmu_tx_t *tx);
void dmu_tx_commit(dmu_tx_t *tx);
void dmu_tx_mark_netfree(dmu_tx_t *tx);

/*
 * To register a commit callback, dmu_tx_callback_register() must be called.
 *
 * dcb_data is a pointer to caller private data that is passed on as a
 * callback parameter. The caller is responsible for properly allocating and
 * freeing it.
 *
 * When registering a callback, the transaction must be already created, but
 * it cannot be committed or aborted. It can be assigned to a txg or not.
 *
 * The callback will be called after the transaction has been safely written
 * to stable storage and will also be called if the dmu_tx is aborted.
 * If there is any error which prevents the transaction from being committed to
 * disk, the callback will be called with a value of error != 0.
 */
typedef void dmu_tx_callback_func_t(void *dcb_data, int error);

void dmu_tx_callback_register(dmu_tx_t *tx, dmu_tx_callback_func_t *dcb_func,
    void *dcb_data);

/*
 * Free up the data blocks for a defined range of a file.  If size is
 * -1, the range from offset to end-of-file is freed.
 */
int dmu_free_range(objset_t *os, uint64_t object, uint64_t offset,
        uint64_t size, dmu_tx_t *tx);
int dmu_free_long_range(objset_t *os, uint64_t object, uint64_t offset,
        uint64_t size);
int dmu_free_long_object(objset_t *os, uint64_t object);

/*
 * Convenience functions.
 *
 * Canfail routines will return 0 on success, or an errno if there is a
 * nonrecoverable I/O error.
 */
```

```
689 #define DMU_READ_PREFETCH       0 /* prefetch */
690 #define DMU_READ_NO_PREFETCH    1 /* don't prefetch */
691 int dmu_read(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
692         void *buf, uint32_t flags);
693 void dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
694         const void *buf, dmu_tx_t *tx);
695 void dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
696         dmu_tx_t *tx);
697 int dmu_read_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size);
698 int dmu_read_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size);
699 int dmu_write_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size,
700     dmu_tx_t *tx);
701 int dmu_write_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size,
702     dmu_tx_t *tx);
703 int dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset,
704     uint64_t size, struct page *pp, dmu_tx_t *tx);
705 struct arc_buf *dmu_request_arcbuf(dmu_buf_t *handle, int size);
706 void dmu_return_arcbuf(struct arc_buf *buf);
707 void dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, struct arc_buf *buf,
708     dmu_tx_t *tx);
709 int dmu_xuio_init(struct xuio *uio, int niov);
710 void dmu_xuio_fini(struct xuio *uio);
711 int dmu_xuio_add(struct xuio *uio, struct arc_buf *abuf, offset_t off,
712     size_t n);
713 int dmu_xuio_cnt(struct xuio *uio);
714 struct arc_buf *dmu_xuio_arcbuf(struct xuio *uio, int i);
715 void dmu_xuio_clear(struct xuio *uio, int i);
716 void xuio_stat_wbuf_copied();
717 void xuio_stat_wbuf_nocopy();

719 extern int zfs_prefetch_disable;
720 extern int zfs_max_recordsize;

722 /*
723  * Asynchronously try to read in the data.
724  */
725 void dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset,
726     uint64_t len);

728 typedef struct dmu_object_info {
729         /* All sizes are in bytes unless otherwise indicated. */
730         uint32_t doi_data_block_size;
731         uint32_t doi_metadata_block_size;
732         dmu_object_type_t doi_type;
733         dmu_object_type_t doi_bonus_type;
734         uint64_t doi_bonus_size;
735         uint8_t doi_indirection;                /* 2 = dnode->indirect->data */
736         uint8_t doi_checksum;
737         uint8_t doi_compress;
738         uint8_t doi_nblkptr;
739         uint8_t doi_pad[4];
740         uint64_t doi_physical_blocks_512;       /* data + metadata, 512b blks */
741         uint64_t doi_max_offset;
742         uint64_t doi_fill_count;                /* number of non-empty blocks */
743 } dmu_object_info_t;
```

unchanged_portion_omitted_

```
**********************************************************
    5967 Tue Jan  6 10:37:51 2015
new/usr/src/uts/common/fs/zfs/sys/dmu_objset.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

/* Portions Copyright 2010 Robert Milkowski */

#ifndef _SYS_DMU_OBJSET_H
#define _SYS_DMU_OBJSET_H

#include <sys/spa.h>
#include <sys/arc.h>
#include <sys/txg.h>
#include <sys/zfs_context.h>
#include <sys/dnode.h>
#include <sys/zio.h>
#include <sys/zil.h>
#include <sys/sa.h>

#ifdef  __cplusplus
extern "C" {
#endif

extern krwlock_t os_lock;

struct dsl_pool;
struct dsl_dataset;
struct dmu_tx;

#define OBJSET_PHYS_SIZE 2048
#define OBJSET_OLD_PHYS_SIZE 1024

#define OBJSET_BUF_HAS_USERUSED(buf) \
        (arc_buf_size(buf) > OBJSET_OLD_PHYS_SIZE)
```



```
#define OBJSET_FLAG_USERACCOUNTING_COMPLETE     (1ULL<<0)

typedef struct objset_phys {
        dnode_phys_t os_meta_dnode;
        zil_header_t os_zil_header;
        uint64_t os_type;
        uint64_t os_flags;
        char os_pad[OBJSET_PHYS_SIZE - sizeof (dnode_phys_t)*3 -
            sizeof (zil_header_t) - sizeof (uint64_t)*2];
        dnode_phys_t os_userused_dnode;
        dnode_phys_t os_groupused_dnode;
} objset_phys_t;

struct objset {
        /* Immutable: */
        struct dsl_dataset *os_dsl_dataset;
        spa_t *os_spa;
        arc_buf_t *os_phys_buf;
        objset_phys_t *os_phys;
        /*
         * The following "special" dnodes have no parent, are exempt
         * from dnode_move(), and are not recorded in os_dnodes, but they
         * root their descendents in this objset using handles anyway, so
         * that all access to dnodes from dbufs consistently uses handles.
         * The following "special" dnodes have no parent and are exempt from
         * dnode_move(), but they root their descendents in this objset using
         * handles anyway, so that all access to dnodes from dbufs consistently
         * uses handles.
         */
        dnode_handle_t os_meta_dnode;
        dnode_handle_t os_userused_dnode;
        dnode_handle_t os_groupused_dnode;
        zilog_t *os_zil;

        list_node_t os_evicting_node;

        /* can change, under dsl_dir's locks: */
        enum zio_checksum os_checksum;
        enum zio_compress os_compress;
        uint8_t os_copies;
        enum zio_checksum os_dedup_checksum;
        boolean_t os_dedup_verify;
        boolean_t os_evicting;
        zfs_logbias_op_t os_logbias;
        zfs_cache_type_t os_primary_cache;
        zfs_cache_type_t os_secondary_cache;
        zfs_sync_type_t os_sync;
        zfs_redundant_metadata_type_t os_redundant_metadata;
        int os_recordsize;

        /* no lock needed: */
        struct dmu_tx *os_synctx; /* XXX sketchy */
        blkptr_t *os_rootbp;
        zil_header_t os_zil_header;
        list_t os_synced_dnodes;
        uint64_t os_flags;

        /* Protected by os_obj_lock */
        kmutex_t os_obj_lock;
        uint64_t os_obj_next;

        /* Protected by os_lock */
        kmutex_t os_lock;
        list_t os_dirty_dnodes[TXG_SIZE];
        list_t os_free_dnodes[TXG_SIZE];
        list_t os_dnodes;
```

```
 120         list_t os_downgraded_dbufs;

 122         /* stuff we store for the user */
 123         kmutex_t os_user_ptr_lock;
 124         void *os_user_ptr;
 125         sa_os_t *os_sa;
 126 };

 128 #define DMU_META_OBJSET         0
 129 #define DMU_META_DNODE_OBJECT   0
 130 #define DMU_OBJECT_IS_SPECIAL(obj) ((int64_t)(obj) <= 0)
 131 #define DMU_META_DNODE(os)      ((os)->os_meta_dnode.dnh_dnode)
 132 #define DMU_USERUSED_DNODE(os)  ((os)->os_userused_dnode.dnh_dnode)
 133 #define DMU_GROUPUSED_DNODE(os) ((os)->os_groupused_dnode.dnh_dnode)

 135 #define DMU_OS_IS_L2CACHEABLE(os)                               \
 136         ((os)->os_secondary_cache == ZFS_CACHE_ALL ||           \
 137         (os)->os_secondary_cache == ZFS_CACHE_METADATA)

 139 #define DMU_OS_IS_L2COMPRESSIBLE(os)    (zfs_mdcomp_disable == B_FALSE)

 141 /* called from zpl */
 142 int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
 143 int dmu_objset_own(const char *name, dmu_objset_type_t type,
 144     boolean_t readonly, void *tag, objset_t **osp);
 145 void dmu_objset_refresh_ownership(objset_t *os, void *tag);
 146 void dmu_objset_rele(objset_t *os, void *tag);
 147 void dmu_objset_disown(objset_t *os, void *tag);
 148 int dmu_objset_from_ds(struct dsl_dataset *ds, objset_t **osp);

 150 void dmu_objset_stats(objset_t *os, nvlist_t *nv);
 151 void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);
 152 void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
 153     uint64_t *usedobjsp, uint64_t *availobjsp);
 154 uint64_t dmu_objset_fsid_guid(objset_t *os);
 155 int dmu_objset_find_dp(struct dsl_pool *dp, uint64_t ddobj,
 156     int func(struct dsl_pool *, struct dsl_dataset *, void *),
 157     void *arg, int flags);
 158 int dmu_objset_prefetch(const char *name, void *arg);
 159 void dmu_objset_evict_dbufs(objset_t *os);
 160 timestruc_t dmu_objset_snap_cmtime(objset_t *os);

 162 /* called from dsl */
 163 void dmu_objset_sync(objset_t *os, zio_t *zio, dmu_tx_t *tx);
 164 boolean_t dmu_objset_is_dirty(objset_t *os, uint64_t txg);
 165 objset_t *dmu_objset_create_impl(spa_t *spa, struct dsl_dataset *ds,
 166     blkptr_t *bp, dmu_objset_type_t type, dmu_tx_t *tx);
 167 int dmu_objset_open_impl(spa_t *spa, struct dsl_dataset *ds, blkptr_t *bp,
 168     objset_t **osp);
 169 void dmu_objset_evict(objset_t *os);
 170 void dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx);
 171 void dmu_objset_userquota_get_ids(dnode_t *dn, boolean_t before, dmu_tx_t *tx);
 172 boolean_t dmu_objset_userused_enabled(objset_t *os);
 173 int dmu_objset_userspace_upgrade(objset_t *os);
 174 boolean_t dmu_objset_userspace_present(objset_t *os);
 175 int dmu_fsname(const char *snapname, char *buf);

 177 void dmu_objset_evict_done(objset_t *os);

 179 void dmu_objset_init(void);
 180 void dmu_objset_fini(void);

 182 #ifdef  __cplusplus
 183 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   11306 Tue Jan  6 10:37:51 2015
new/usr/src/uts/common/fs/zfs/sys/dnode.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#ifndef _SYS_DNODE_H
#define _SYS_DNODE_H

#include <sys/zfs_context.h>
#include <sys/avl.h>
#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/zio.h>
#include <sys/refcount.h>
#include <sys/dmu_zfetch.h>
#include <sys/zrlock.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * dnode_hold() flags.
 */
#define DNODE_MUST_BE_ALLOCATED 1
#define DNODE_MUST_BE_FREE      2

/*
 * dnode_next_offset() flags.
 */
#define DNODE_FIND_HOLE         1
#define DNODE_FIND_BACKWARDS    2
#define DNODE_FIND_HAVELOCK     4

/*
 * Fixed constants.
```



```
 */
#define DNODE_SHIFT             9       /* 512 bytes */
#define DN_MIN_INDBLKSHIFT      12      /* 4k */
#define DN_MAX_INDBLKSHIFT      14      /* 16k */
#define DNODE_BLOCK_SHIFT       14      /* 16k */
#define DNODE_CORE_SIZE         64      /* 64 bytes for dnode sans blkptrs */
#define DN_MAX_OBJECT_SHIFT     48      /* 256 trillion (zfs_fid_t limit) */
#define DN_MAX_OFFSET_SHIFT     64      /* 2^64 bytes in a dnode */

/*
 * dnode id flags
 *
 * Note: a file will never ever have its
 * ids moved from bonus->spill
 * and only in a crypto environment would it be on spill
 */
#define DN_ID_CHKED_BONUS       0x1
#define DN_ID_CHKED_SPILL       0x2
#define DN_ID_OLD_EXIST         0x4
#define DN_ID_NEW_EXIST         0x8

/*
 * Derived constants.
 */
#define DNODE_SIZE      (1 << DNODE_SHIFT)
#define DN_MAX_NBLKPTR  ((DNODE_SIZE - DNODE_CORE_SIZE) >> SPA_BLKPTRSHIFT)
#define DN_MAX_BONUSLEN (DNODE_SIZE - DNODE_CORE_SIZE - (1 << SPA_BLKPTRSHIFT))
#define DN_MAX_OBJECT   (1ULL << DN_MAX_OBJECT_SHIFT)
#define DN_ZERO_BONUSLEN        (DN_MAX_BONUSLEN + 1)
#define DN_KILL_SPILLBLK (1)

#define DNODES_PER_BLOCK_SHIFT  (DNODE_BLOCK_SHIFT - DNODE_SHIFT)
#define DNODES_PER_BLOCK        (1ULL << DNODES_PER_BLOCK_SHIFT)
#define DNODES_PER_LEVEL_SHIFT  (DN_MAX_INDBLKSHIFT - SPA_BLKPTRSHIFT)
#define DNODES_PER_LEVEL        (1ULL << DNODES_PER_LEVEL_SHIFT)

/* The +2 here is a cheesy way to round up */
#define DN_MAX_LEVELS   (2 + ((DN_MAX_OFFSET_SHIFT - SPA_MINBLOCKSHIFT) / \
        (DN_MIN_INDBLKSHIFT - SPA_BLKPTRSHIFT)))

#define DN_BONUS(dnp)   ((void*)((dnp)->dn_bonus + \
        (((dnp)->dn_nblkptr - 1) * sizeof (blkptr_t))))

#define DN_USED_BYTES(dnp) (((dnp)->dn_flags & DNODE_FLAG_USED_BYTES) ? \
        (dnp)->dn_used : (dnp)->dn_used << SPA_MINBLOCKSHIFT)

#define EPB(blkshift, typeshift)        (1 << (blkshift - typeshift))

struct dmu_buf_impl;
struct objset;
struct zio;

enum dnode_dirtycontext {
        DN_UNDIRTIED,
        DN_DIRTY_OPEN,
        DN_DIRTY_SYNC
};
______unchanged_portion_omitted_

typedef struct dnode_children {
        dmu_buf_user_t dnc_dbu;         /* User evict data */
        size_t dnc_count;               /* number of children */
        dnode_handle_t dnc_children[];  /* sized dynamically */
} dnode_children_t;
______unchanged_portion_omitted_
```

```
270 void dnode_special_open(struct objset *dd, dnode_phys_t *dnp,
268 dnode_t *dnode_special_open(struct objset *dd, dnode_phys_t *dnp,
271     uint64_t object, dnode_handle_t *dnh);
272 void dnode_special_close(dnode_handle_t *dnh);

274 void dnode_setbonuslen(dnode_t *dn, int newsize, dmu_tx_t *tx);
275 void dnode_setbonus_type(dnode_t *dn, dmu_object_type_t, dmu_tx_t *tx);
276 void dnode_rm_spill(dnode_t *dn, dmu_tx_t *tx);

278 int dnode_hold(struct objset *dd, uint64_t object,
279     void *ref, dnode_t **dnp);
280 int dnode_hold_impl(struct objset *dd, uint64_t object, int flag,
281     void *ref, dnode_t **dnp);
282 boolean_t dnode_add_ref(dnode_t *dn, void *ref);
283 void dnode_rele(dnode_t *dn, void *ref);
284 void dnode_setdirty(dnode_t *dn, dmu_tx_t *tx);
285 void dnode_sync(dnode_t *dn, dmu_tx_t *tx);
286 void dnode_allocate(dnode_t *dn, dmu_object_type_t ot, int blocksize, int ibs,
287     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
288 void dnode_reallocate(dnode_t *dn, dmu_object_type_t ot, int blocksize,
289     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
290 void dnode_free(dnode_t *dn, dmu_tx_t *tx);
291 void dnode_byteswap(dnode_phys_t *dnp);
292 void dnode_buf_byteswap(void *buf, size_t size);
293 void dnode_verify(dnode_t *dn);
294 int dnode_set_blksz(dnode_t *dn, uint64_t size, int ibs, dmu_tx_t *tx);
295 void dnode_free_range(dnode_t *dn, uint64_t off, uint64_t len, dmu_tx_t *tx);
296 void dnode_diduse_space(dnode_t *dn, int64_t space);
297 void dnode_willuse_space(dnode_t *dn, int64_t space, dmu_tx_t *tx);
298 void dnode_new_blkid(dnode_t *dn, uint64_t blkid, dmu_tx_t *tx, boolean_t);
299 uint64_t dnode_block_freed(dnode_t *dn, uint64_t blkid);
300 void dnode_init(void);
301 void dnode_fini(void);
302 int dnode_next_offset(dnode_t *dn, int flags, uint64_t *off,
303     int minlvl, uint64_t blkfill, uint64_t txg);
304 void dnode_evict_dbufs(dnode_t *dn);

306 #ifdef ZFS_DEBUG

308 /*
309  * There should be a ## between the string literal and fmt, to make it
310  * clear that we're joining two strings together, but that piece of shit
311  * gcc doesn't support that preprocessor token.
312  */
313 #define dprintf_dnode(dn, fmt, ...) do { \
314         if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
315         char __db_buf[32]; \
316         uint64_t __db_obj = (dn)->dn_object; \
317         if (__db_obj == DMU_META_DNODE_OBJECT) \
318                 (void) strcpy(__db_buf, "mdn"); \
319         else \
320                 (void) snprintf(__db_buf, sizeof (__db_buf), "%lld", \
321                     (u_longlong_t)__db_obj);\
322         dprintf_ds((dn)->dn_objset->os_dsl_dataset, "obj=%s " fmt, \
323             __db_buf, __VA_ARGS__); \
324         } \
325 _NOTE(CONSTCOND) } while (0)

327 #define DNODE_VERIFY(dn)                dnode_verify(dn)
328 #define FREE_VERIFY(db, start, end, tx) free_verify(db, start, end, tx)

330 #else

332 #define dprintf_dnode(db, fmt, ...)
333 #define DNODE_VERIFY(dn)
334 #define FREE_VERIFY(db, start, end, tx)
```



```
336 #endif

338 #ifdef  __cplusplus
339 }
```

unchanged_portion_omitted_

```
**********************************************************
   11414 Tue Jan  6 10:37:51 2015
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright (c) 2013 Steven Hartland. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#ifndef _SYS_DSL_DATASET_H
#define _SYS_DSL_DATASET_H

#include <sys/dmu.h>
#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/zio.h>
#include <sys/bplist.h>
#include <sys/dsl_synctask.h>
#include <sys/zfs_context.h>
#include <sys/dsl_deadlist.h>
#include <sys/refcount.h>

#ifdef __cplusplus
extern "C" {
#endif

struct dsl_dataset;
struct dsl_dir;
struct dsl_pool;

#define DS_FLAG_INCONSISTENT    (1ULL<<0)
#define DS_IS_INCONSISTENT(ds)  \
        (dsl_dataset_phys(ds)->ds_flags & DS_FLAG_INCONSISTENT)

/*
 * Do not allow this dataset to be promoted.
 */
#define DS_FLAG_NOPROMOTE       (1ULL<<1)
```



```
/*
 * DS_FLAG_UNIQUE_ACCURATE is set if ds_unique_bytes has been correctly
 * calculated for head datasets (starting with SPA_VERSION_UNIQUE_ACCURATE,
 * refquota/refreservations).
 */
#define DS_FLAG_UNIQUE_ACCURATE (1ULL<<2)

/*
 * DS_FLAG_DEFER_DESTROY is set after 'zfs destroy -d' has been called
 * on a dataset. This allows the dataset to be destroyed using 'zfs release'.
 */
#define DS_FLAG_DEFER_DESTROY   (1ULL<<3)
#define DS_IS_DEFER_DESTROY(ds) \
        (dsl_dataset_phys(ds)->ds_flags & DS_FLAG_DEFER_DESTROY)

/*
 * DS_FIELD_* are strings that are used in the "extensified" dataset zap object.
 * They should be of the format <reverse-dns>:<field>.
 */

/*
 * This field's value is the object ID of a zap object which contains the
 * bookmarks of this dataset.  If it is present, then this dataset is counted
 * in the refcount of the SPA_FEATURES_BOOKMARKS feature.
 */
#define DS_FIELD_BOOKMARK_NAMES "com.delphix:bookmarks"

/*
 * This field is present (with value=0) if this dataset may contain large
 * blocks (>128KB).  If it is present, then this dataset
 * is counted in the refcount of the SPA_FEATURE_LARGE_BLOCKS feature.
 */
#define DS_FIELD_LARGE_BLOCKS "org.open-zfs:large_blocks"

/*
 * DS_FLAG_CI_DATASET is set if the dataset contains a file system whose
 * name lookups should be performed case-insensitively.
 */
#define DS_FLAG_CI_DATASET      (1ULL<<16)

#define DS_CREATE_FLAG_NODIRTY  (1ULL<<24)

typedef struct dsl_dataset_phys {
        uint64_t ds_dir_obj;            /* DMU_OT_DSL_DIR */
        uint64_t ds_prev_snap_obj;      /* DMU_OT_DSL_DATASET */
        uint64_t ds_prev_snap_txg;
        uint64_t ds_next_snap_obj;      /* DMU_OT_DSL_DATASET */
        uint64_t ds_snapnames_zapobj;   /* DMU_OT_DSL_DS_SNAP_MAP 0 for snaps */
        uint64_t ds_num_children;       /* clone/snap children; ==0 for head */
        uint64_t ds_creation_time;      /* seconds since 1970 */
        uint64_t ds_creation_txg;
        uint64_t ds_deadlist_obj;       /* DMU_OT_DEADLIST */
        /*
         * ds_referenced_bytes, ds_compressed_bytes, and ds_uncompressed_bytes
         * include all blocks referenced by this dataset, including those
         * shared with any other datasets.
         */
        uint64_t ds_referenced_bytes;
        uint64_t ds_compressed_bytes;
        uint64_t ds_uncompressed_bytes;
        uint64_t ds_unique_bytes;       /* only relevant to snapshots */
        /*
         * The ds_fsid_guid is a 56-bit ID that can change to avoid
         * collisions.  The ds_guid is a 64-bit ID that will never
         * change, so there is a small probability that it will collide.
```

```
124          */
125         uint64_t ds_fsid_guid;
126         uint64_t ds_guid;
127         uint64_t ds_flags;              /* DS_FLAG_* */
128         blkptr_t ds_bp;
129         uint64_t ds_next_clones_obj;    /* DMU_OT_DSL_CLONES */
130         uint64_t ds_props_obj;          /* DMU_OT_DSL_PROPS for snaps */
131         uint64_t ds_userrefs_obj;       /* DMU_OT_USERREFS */
132         uint64_t ds_pad[5]; /* pad out to 320 bytes for good measure */
133 } dsl_dataset_phys_t;

135 typedef struct dsl_dataset {
136         dmu_buf_user_t ds_dbu;

138         /* Immutable: */
139         struct dsl_dir *ds_dir;
140         dmu_buf_t *ds_dbuf;
141         uint64_t ds_object;
142         uint64_t ds_fsid_guid;
143         boolean_t ds_is_snapshot;

145         /* only used in syncing context, only valid for non-snapshots: */
146         struct dsl_dataset *ds_prev;
147         uint64_t ds_bookmarks;  /* DMU_OTN_ZAP_METADATA */
148         boolean_t ds_large_blocks;
149         boolean_t ds_need_large_blocks;

151         /* has internal locking: */
152         dsl_deadlist_t ds_deadlist;
153         bplist_t ds_pending_deadlist;

155         /* protected by lock on pool's dp_dirty_datasets list */
156         txg_node_t ds_dirty_link;
157         list_node_t ds_synced_link;

159         /*
160          * ds_phys->ds_<accounting> is also protected by ds_lock.
161          * Protected by ds_lock:
162          */
163         kmutex_t ds_lock;
164         objset_t *ds_objset;
165         uint64_t ds_userrefs;
166         void *ds_owner;

168         /*
169          * Long holds prevent the ds from being destroyed; they allow the
170          * ds to remain held even after dropping the dp_config_rwlock.
171          * Owning counts as a long hold.  See the comments above
172          * dsl_pool_hold() for details.
173          */
174         refcount_t ds_longholds;

176         /* no locking; only for making guesses */
177         uint64_t ds_trysnap_txg;

179         /* for objset_open() */
180         kmutex_t ds_opening_lock;

182         uint64_t ds_reserved;   /* cached refreservation */
183         uint64_t ds_quota;      /* cached refquota */

185         kmutex_t ds_sendstream_lock;
186         list_t ds_sendstreams;

188         /* Protected by ds_lock; keep at end of struct for better locality */
189         char ds_snapname[MAXNAMELEN];
```



```
190 } dsl_dataset_t;
_____unchanged_portion_omitted_

198 /*
199  * The max length of a temporary tag prefix is the number of hex digits
200  * required to express UINT64_MAX plus one for the hyphen.
201  */
202 #define MAX_TAG_PREFIX_LEN      17

200 inline boolean_t
201 dsl_dataset_is_snapshot(dsl_dataset_t *ds)
202 {
203         return (dsl_dataset_phys(ds)->ds_num_children != 0);
204 }

204 #define DS_UNIQUE_IS_ACCURATE(ds)       \
205         ((dsl_dataset_phys(ds)->ds_flags & DS_FLAG_UNIQUE_ACCURATE) != 0)

207 int dsl_dataset_hold(struct dsl_pool *dp, const char *name, void *tag,
208     dsl_dataset_t **dsp);
209 int dsl_dataset_hold_obj(struct dsl_pool *dp, uint64_t dsobj, void *tag,
210     dsl_dataset_t **);
211 void dsl_dataset_rele(dsl_dataset_t *ds, void *tag);
212 int dsl_dataset_own(struct dsl_pool *dp, const char *name,
213     void *tag, dsl_dataset_t **dsp);
214 int dsl_dataset_own_obj(struct dsl_pool *dp, uint64_t dsobj,
215     void *tag, dsl_dataset_t **dsp);
216 void dsl_dataset_disown(dsl_dataset_t *ds, void *tag);
217 void dsl_dataset_name(dsl_dataset_t *ds, char *name);
218 boolean_t dsl_dataset_tryown(dsl_dataset_t *ds, void *tag);
219 uint64_t dsl_dataset_create_sync(dsl_dir_t *pds, const char *lastname,
220     dsl_dataset_t *origin, uint64_t flags, cred_t *, dmu_tx_t *);
221 uint64_t dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
222     uint64_t flags, dmu_tx_t *tx);
223 int dsl_dataset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors);
224 int dsl_dataset_promote(const char *name, char *conflsnap);
225 int dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
226     boolean_t force);
227 int dsl_dataset_rename_snapshot(const char *fsname,
228     const char *oldsnapname, const char *newsnapname, boolean_t recursive);
229 int dsl_dataset_snapshot_tmp(const char *fsname, const char *snapname,
230     minor_t cleanup_minor, const char *htag);

232 blkptr_t *dsl_dataset_get_blkptr(dsl_dataset_t *ds);
233 void dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx);

235 spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);

237 boolean_t dsl_dataset_modified_since_snap(dsl_dataset_t *ds,
238     dsl_dataset_t *snap);

240 void dsl_dataset_sync(dsl_dataset_t *os, zio_t *zio, dmu_tx_t *tx);

242 void dsl_dataset_block_born(dsl_dataset_t *ds, const blkptr_t *bp,
243     dmu_tx_t *tx);
244 int dsl_dataset_block_kill(dsl_dataset_t *ds, const blkptr_t *bp,
245     dmu_tx_t *tx, boolean_t async);
246 boolean_t dsl_dataset_block_freeable(dsl_dataset_t *ds, const blkptr_t *bp,
247     uint64_t blk_birth);
248 uint64_t dsl_dataset_prev_snap_txg(dsl_dataset_t *ds);

250 void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
251 void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);
252 void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
253 void dsl_dataset_space(dsl_dataset_t *ds,
254     uint64_t *refdbytesp, uint64_t *availbytesp,
```

```
255     uint64_t *usedobjsp, uint64_t *availobjsp);
256 uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
257 int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
258     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
259 int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
260     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
261 boolean_t dsl_dataset_is_dirty(dsl_dataset_t *ds);
262 int dsl_dataset_activate_large_blocks(const char *dsname);
263 void dsl_dataset_activate_large_blocks_sync_impl(uint64_t dsobj, dmu_tx_t *tx);

265 int dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf);

267 int dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
268     uint64_t asize, uint64_t inflight, uint64_t *used,
269     uint64_t *ref_rsrv);
270 int dsl_dataset_set_refquota(const char *dsname, zprop_source_t source,
271     uint64_t quota);
272 int dsl_dataset_set_refreservation(const char *dsname, zprop_source_t source,
273     uint64_t reservation);

275 boolean_t dsl_dataset_is_before(dsl_dataset_t *later, dsl_dataset_t *earlier,
276     uint64_t earlier_txg);
277 void dsl_dataset_long_hold(dsl_dataset_t *ds, void *tag);
278 void dsl_dataset_long_rele(dsl_dataset_t *ds, void *tag);
279 boolean_t dsl_dataset_long_held(dsl_dataset_t *ds);

281 int dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
282     dsl_dataset_t *origin_head, boolean_t force, void *owner, dmu_tx_t *tx);
283 void dsl_dataset_clone_swap_sync_impl(dsl_dataset_t *clone,
284     dsl_dataset_t *origin_head, dmu_tx_t *tx);
285 int dsl_dataset_snapshot_check_impl(dsl_dataset_t *ds, const char *snapname,
286     dmu_tx_t *tx, boolean_t recv, uint64_t cnt, cred_t *cr);
287 void dsl_dataset_snapshot_sync_impl(dsl_dataset_t *ds, const char *snapname,
288     dmu_tx_t *tx);

290 void dsl_dataset_remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj,
291     dmu_tx_t *tx);
292 void dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds);
293 int dsl_dataset_get_snapname(dsl_dataset_t *ds);
294 int dsl_dataset_snap_lookup(dsl_dataset_t *ds, const char *name,
295     uint64_t *value);
296 int dsl_dataset_snap_remove(dsl_dataset_t *ds, const char *name, dmu_tx_t *tx,
297     boolean_t adj_cnt);
298 void dsl_dataset_set_refreservation_sync_impl(dsl_dataset_t *ds,
299     zprop_source_t source, uint64_t value, dmu_tx_t *tx);
300 void dsl_dataset_zapify(dsl_dataset_t *ds, dmu_tx_t *tx);
301 int dsl_dataset_rollback(const char *fsname, void *owner, nvlist_t *result);

303 #ifdef ZFS_DEBUG
304 #define dprintf_ds(ds, fmt, ...) do { \
305         if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
306         char *__ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
307         dsl_dataset_name(ds, __ds_name); \
308         dprintf("ds=%s " fmt, __ds_name, __VA_ARGS__); \
309         kmem_free(__ds_name, MAXNAMELEN); \
310         } \
311 _NOTE(CONSTCOND) } while (0)
312 #else
313 #define dprintf_ds(dd, fmt, ...)
314 #endif

316 #ifdef  __cplusplus
317 }
318 #endif

320 #endif /* _SYS_DSL_DATASET_H */
```

```
**********************************************************
    6475 Tue Jan  6 10:37:51 2015
new/usr/src/uts/common/fs/zfs/sys/dsl_dir.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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.
 * Copyright (c) 2014, Joyent, Inc. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#ifndef _SYS_DSL_DIR_H
#define _SYS_DSL_DIR_H

#include <sys/dmu.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_synctask.h>
#include <sys/refcount.h>
#include <sys/zfs_context.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct dsl_dataset;

/*
 * DD_FIELD_* are strings that are used in the "extensified" dsl_dir zap object.
 * They should be of the format <reverse-dns>:<field>.
 */

#define DD_FIELD_FILESYSTEM_COUNT       "com.joyent:filesystem_count"
#define DD_FIELD_SNAPSHOT_COUNT         "com.joyent:snapshot_count"

typedef enum dd_used {
        DD_USED_HEAD,
        DD_USED_SNAP,
        DD_USED_CHILD,
        DD_USED_CHILD_RSRV,
        DD_USED_REFRSRV,
        DD_USED_NUM
} dd_used_t;
```



```
} dd_used_t;
_____unchanged_portion_omitted_

struct dsl_dir {
        dmu_buf_user_t dd_dbu;

        /* These are immutable; no lock needed: */
        uint64_t dd_object;
        dsl_pool_t *dd_pool;

        /* Stable until user eviction; no lock needed: */
        dmu_buf_t *dd_dbuf;

        /* protected by lock on pool's dp_dirty_dirs list */
        txg_node_t dd_dirty_link;

        /* protected by dp_config_rwlock */
        dsl_dir_t *dd_parent;

        /* Protected by dd_lock */
        kmutex_t dd_lock;
        list_t dd_prop_cbs; /* list of dsl_prop_cb_record_t's */
        timestruc_t dd_snap_cmtime; /* last time snapshot namespace changed */
        uint64_t dd_origin_txg;

        /* gross estimate of space used by in-flight tx's */
        uint64_t dd_tempreserved[TXG_SIZE];
        /* amount of space we expect to write; == amount of dirty data */
        int64_t dd_space_towrite[TXG_SIZE];

        /* protected by dd_lock; keep at end of struct for better locality */
        char dd_myname[MAXNAMELEN];
};
_____unchanged_portion_omitted_

void dsl_dir_rele(dsl_dir_t *dd, void *tag);
void dsl_dir_async_rele(dsl_dir_t *dd, void *tag);
int dsl_dir_hold(dsl_pool_t *dp, const char *name, void *tag,
    dsl_dir_t **, const char **tail);
int dsl_dir_hold_obj(dsl_pool_t *dp, uint64_t ddobj,
    const char *tail, void *tag, dsl_dir_t **);
void dsl_dir_name(dsl_dir_t *dd, char *buf);
int dsl_dir_namelen(dsl_dir_t *dd);
uint64_t dsl_dir_create_sync(dsl_pool_t *dp, dsl_dir_t *pds,
    const char *name, dmu_tx_t *tx);
void dsl_dir_stats(dsl_dir_t *dd, nvlist_t *nv);
uint64_t dsl_dir_space_available(dsl_dir_t *dd,
    dsl_dir_t *ancestor, int64_t delta, int ondiskonly);
void dsl_dir_dirty(dsl_dir_t *dd, dmu_tx_t *tx);
void dsl_dir_sync(dsl_dir_t *dd, dmu_tx_t *tx);
int dsl_dir_tempreserve_space(dsl_dir_t *dd, uint64_t mem,
    uint64_t asize, uint64_t fsize, uint64_t usize, void **tr_cookiep,
    dmu_tx_t *tx);
void dsl_dir_tempreserve_clear(void *tr_cookie, dmu_tx_t *tx);
void dsl_dir_willuse_space(dsl_dir_t *dd, int64_t space, dmu_tx_t *tx);
void dsl_dir_diduse_space(dsl_dir_t *dd, dd_used_t type,
    int64_t used, int64_t compressed, int64_t uncompressed, dmu_tx_t *tx);
void dsl_dir_transfer_space(dsl_dir_t *dd, int64_t delta,
    dd_used_t oldtype, dd_used_t newtype, dmu_tx_t *tx);
int dsl_dir_set_quota(const char *ddname, zprop_source_t source,
    uint64_t quota);
int dsl_dir_set_reservation(const char *ddname, zprop_source_t source,
    uint64_t reservation);
int dsl_dir_activate_fs_ss_limit(const char *);
int dsl_fs_ss_limit_check(dsl_dir_t *, uint64_t, zfs_prop_t, dsl_dir_t *,
    cred_t *);
```

```
155 void dsl_fs_ss_count_adjust(dsl_dir_t *, int64_t, const char *, dmu_tx_t *);
156 int dsl_dir_rename(const char *oldname, const char *newname);
157 int dsl_dir_transfer_possible(dsl_dir_t *sdd, dsl_dir_t *tdd,
158     uint64_t fs_cnt, uint64_t ss_cnt, uint64_t space, cred_t *);
159 boolean_t dsl_dir_is_clone(dsl_dir_t *dd);
160 void dsl_dir_new_refreservation(dsl_dir_t *dd, struct dsl_dataset *ds,
161     uint64_t reservation, cred_t *cr, dmu_tx_t *tx);
162 void dsl_dir_snap_cmtime_update(dsl_dir_t *dd);
163 timestruc_t dsl_dir_snap_cmtime(dsl_dir_t *dd);
164 void dsl_dir_set_reservation_sync_impl(dsl_dir_t *dd, uint64_t value,
165     dmu_tx_t *tx);
166 void dsl_dir_zapify(dsl_dir_t *dd, dmu_tx_t *tx);
167 boolean_t dsl_dir_is_zapified(dsl_dir_t *dd);

169 /* internal reserved dir name */
170 #define MOS_DIR_NAME "$MOS"
171 #define ORIGIN_DIR_NAME "$ORIGIN"
172 #define XLATION_DIR_NAME "$XLATION"
173 #define FREE_DIR_NAME "$FREE"
174 #define LEAK_DIR_NAME "$LEAK"

176 #ifdef ZFS_DEBUG
177 #define dprintf_dd(dd, fmt, ...) do { \
178         if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
179         char *__ds_name = kmem_alloc(MAXNAMELEN + strlen(MOS_DIR_NAME) + 1, \
180             KM_SLEEP); \
181         dsl_dir_name(dd, __ds_name); \
182         dprintf("dd=%s " fmt, __ds_name, __VA_ARGS__); \
183         kmem_free(__ds_name, MAXNAMELEN + strlen(MOS_DIR_NAME) + 1); \
184         } \
185 _NOTE(CONSTCOND) } while (0)
186 #else
187 #define dprintf_dd(dd, fmt, ...)
188 #endif

190 #ifdef  __cplusplus
191 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    5135 Tue Jan  6 10:37:52 2015
new/usr/src/uts/common/fs/zfs/sys/sa.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.com>
**********************************************************
______unchanged_portion_omitted_

 111 struct sa_handle;
 112 typedef void *sa_lookup_tab_t;
 113 typedef struct sa_handle sa_handle_t;

 115 typedef void (sa_update_cb_t)(sa_handle_t *, dmu_tx_t *tx);

 117 int sa_handle_get(objset_t *, uint64_t, void *userp,
 118     sa_handle_type_t, sa_handle_t **);
 119 int sa_handle_get_from_db(objset_t *, dmu_buf_t *, void *userp,
 120     sa_handle_type_t, sa_handle_t **);
 121 void sa_handle_destroy(sa_handle_t *);
 122 int sa_buf_hold(objset_t *, uint64_t, void *, dmu_buf_t **);
 123 void sa_buf_rele(dmu_buf_t *, void *);
 124 int sa_lookup(sa_handle_t *, sa_attr_type_t, void *buf, uint32_t buflen);
 125 int sa_update(sa_handle_t *, sa_attr_type_t, void *buf,
 126     uint32_t buflen, dmu_tx_t *);
 127 int sa_remove(sa_handle_t *, sa_attr_type_t, dmu_tx_t *);
 128 int sa_bulk_lookup(sa_handle_t *, sa_bulk_attr_t *, int count);
 129 int sa_bulk_lookup_locked(sa_handle_t *, sa_bulk_attr_t *, int count);
 130 int sa_bulk_update(sa_handle_t *, sa_bulk_attr_t *, int count, dmu_tx_t *);
 131 int sa_size(sa_handle_t *, sa_attr_type_t, int *);
 132 int sa_update_from_cb(sa_handle_t *, sa_attr_type_t,
 133     uint32_t buflen, sa_data_locator_t *, void *userdata, dmu_tx_t *);
 134 void sa_object_info(sa_handle_t *, dmu_object_info_t *);
 135 void sa_object_size(sa_handle_t *, uint32_t *, u_longlong_t *);
 136 void sa_update_user(sa_handle_t *, sa_handle_t *);
 136 void *sa_get_userdata(sa_handle_t *);
 137 void sa_set_userp(sa_handle_t *, void *);
 138 dmu_buf_t *sa_get_db(sa_handle_t *);
 139 uint64_t sa_handle_object(sa_handle_t *);
 140 boolean_t sa_attr_would_spill(sa_handle_t *, sa_attr_type_t, int size);
 141 void sa_register_update_callback(objset_t *, sa_update_cb_t *);
 142 int sa_setup(objset_t *, uint64_t, sa_attr_reg_t *, int, sa_attr_type_t **);
 143 void sa_tear_down(objset_t *);
 144 int sa_replace_all_by_template(sa_handle_t *, sa_bulk_attr_t *,
 145     int, dmu_tx_t *);
 146 int sa_replace_all_by_template_locked(sa_handle_t *, sa_bulk_attr_t *,
 147     int, dmu_tx_t *);
 148 boolean_t sa_enabled(objset_t *);
 149 void sa_cache_init();
 150 void sa_cache_fini();
 151 int sa_set_sa_object(objset_t *, uint64_t);
 152 int sa_hdrsize(void *);
 153 void sa_handle_lock(sa_handle_t *);
 154 void sa_handle_unlock(sa_handle_t *);

 156 #ifdef _KERNEL
 157 int sa_lookup_uio(sa_handle_t *, sa_attr_type_t, uio_t *);
 158 #endif

 160 #ifdef  __cplusplus
 161 extern "C" {
 162 #endif
```



```
 165 #ifdef  __cplusplus
 166 }
______unchanged_portion_omitted_
```

****
8513 Tue Jan 6 10:37:52 2015
new/usr/src/uts/common/fs/zfs/sys/sa_impl.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#ifndef _SYS_SA_IMPL_H
#define _SYS_SA_IMPL_H

#include <sys/dmu.h>
#include <sys/refcount.h>
#include <sys/list.h>

/*
 * Array of known attributes and their
 * various characteristics.
 */
typedef struct sa_attr_table {
        sa_attr_type_t  sa_attr;
        uint8_t sa_registered;
        uint16_t sa_length;
        sa_bswap_type_t sa_byteswap;
        char *sa_name;
} sa_attr_table_t;
```

*unchanged_portion_omitted_*

```
/*
 * Opaque handle used for most sa functions
 *
 * This needs to be kept as small as possible.
 */

struct sa_handle {
        dmu_buf_user_t  sa_dbu;
        kmutex_t        sa_lock;
        dmu_buf_t       *sa_bonus;
        dmu_buf_t       *sa_spill;
```



```
        objset_t        *sa_os;
        void            *sa_userp;
        sa_idx_tab_t    *sa_bonus_tab;   /* idx of bonus */
        sa_idx_tab_t    *sa_spill_tab; /* only present if spill activated */
};
```

*unchanged_portion_omitted_*

```
**********************************************************
   33096 Tue Jan  6 10:37:52 2015
new/usr/src/uts/common/fs/zfs/sys/spa.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2011, 2014 by Delphix. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#ifndef _SYS_SPA_H
#define _SYS_SPA_H

#include <sys/avl.h>
#include <sys/zfs_context.h>
#include <sys/nvpair.h>
#include <sys/sysmacros.h>
#include <sys/types.h>
#include <sys/fs/zfs.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Forward references that lots of things need.
 */
typedef struct spa spa_t;
typedef struct vdev vdev_t;
typedef struct metaslab metaslab_t;
typedef struct metaslab_group metaslab_group_t;
typedef struct metaslab_class metaslab_class_t;
typedef struct zio zio_t;
typedef struct zilog zilog_t;
typedef struct spa_aux_vdev spa_aux_vdev_t;
typedef struct ddt ddt_t;
typedef struct ddt_entry ddt_entry_t;
struct dsl_pool;
struct dsl_dataset;
```



```
/*
 * General-purpose 32-bit and 64-bit bitfield encodings.
 */
#define BF32_DECODE(x, low, len)        P2PHASE((x) >> (low), 1U << (len))
#define BF64_DECODE(x, low, len)        P2PHASE((x) >> (low), 1ULL << (len))
#define BF32_ENCODE(x, low, len)        (P2PHASE((x), 1U << (len)) << (low))
#define BF64_ENCODE(x, low, len)        (P2PHASE((x), 1ULL << (len)) << (low))

#define BF32_GET(x, low, len)           BF32_DECODE(x, low, len)
#define BF64_GET(x, low, len)           BF64_DECODE(x, low, len)

#define BF32_SET(x, low, len, val) do { \
        ASSERT3U(val, <, 1U << (len)); \
        ASSERT3U(low + len, <=, 32); \
        (x) ^= BF32_ENCODE((x >> low) ^ (val), low, len); \
_NOTE(CONSTCOND) } while (0)

#define BF64_SET(x, low, len, val) do { \
        ASSERT3U(val, <, 1ULL << (len)); \
        ASSERT3U(low + len, <=, 64); \
        ((x) ^= BF64_ENCODE((x >> low) ^ (val), low, len)); \
_NOTE(CONSTCOND) } while (0)

#define BF32_GET_SB(x, low, len, shift, bias)   \
        ((BF32_GET(x, low, len) + (bias)) << (shift))
#define BF64_GET_SB(x, low, len, shift, bias)   \
        ((BF64_GET(x, low, len) + (bias)) << (shift))

#define BF32_SET_SB(x, low, len, shift, bias, val) do { \
        ASSERT(IS_P2ALIGNED(val, 1U << shift)); \
        ASSERT3S((val) >> (shift), >=, bias); \
        BF32_SET(x, low, len, ((val) >> (shift)) - (bias)); \
_NOTE(CONSTCOND) } while (0)
#define BF64_SET_SB(x, low, len, shift, bias, val) do { \
        ASSERT(IS_P2ALIGNED(val, 1ULL << shift)); \
        ASSERT3S((val) >> (shift), >=, bias); \
        BF64_SET(x, low, len, ((val) >> (shift)) - (bias)); \
_NOTE(CONSTCOND) } while (0)

/*
 * We currently support block sizes from 512 bytes to 16MB.
 * The benefits of larger blocks, and thus larger IO, need to be weighed
 * against the cost of COWing a giant block to modify one byte, and the
 * large latency of reading or writing a large block.
 *
 * Note that although blocks up to 16MB are supported, the recordsize
 * property can not be set larger than zfs_max_recordsize (default 1MB).
 * See the comment near zfs_max_recordsize in dsl_dataset.c for details.
 *
 * Note that although the LSIZE field of the blkptr_t can store sizes up
 * to 32MB, the dnode's dn_datablkszsec can only store sizes up to
 * 32MB - 512 bytes.  Therefore, we limit SPA_MAXBLOCKSIZE to 16MB.
 */
#define SPA_MINBLOCKSHIFT       9
#define SPA_OLD_MAXBLOCKSHIFT   17
#define SPA_MAXBLOCKSHIFT       24
#define SPA_MINBLOCKSIZE        (1ULL << SPA_MINBLOCKSHIFT)
#define SPA_OLD_MAXBLOCKSIZE    (1ULL << SPA_OLD_MAXBLOCKSHIFT)
#define SPA_MAXBLOCKSIZE        (1ULL << SPA_MAXBLOCKSHIFT)

/*
 * Size of block to hold the configuration data (a packed nvlist)
 */
#define SPA_CONFIG_BLOCKSIZE    (1ULL << 14)

/*
```

```
 124  * The DVA size encodings for LSIZE and PSIZE support blocks up to 32MB.
 125  * The ASIZE encoding should be at least 64 times larger (6 more bits)
 126  * to support up to 4-way RAID-Z mirror mode with worst-case gang block
 127  * overhead, three DVAs per bp, plus one more bit in case we do anything
 128  * else that expands the ASIZE.
 129  */
 130 #define SPA_LSIZEBITS           16      /* LSIZE up to 32M (2^16 * 512) */
 131 #define SPA_PSIZEBITS           16      /* PSIZE up to 32M (2^16 * 512) */
 132 #define SPA_ASIZEBITS           24      /* ASIZE up to 64 times larger  */

 134 /*
 135  * All SPA data is represented by 128-bit data virtual addresses (DVAs).
 136  * The members of the dva_t should be considered opaque outside the SPA.
 137  */
 138 typedef struct dva {
 139         uint64_t        dva_word[2];
 140 } dva_t;
_____unchanged_portion_omitted_

 581 /* state manipulation functions */
 582 extern int spa_open(const char *pool, spa_t **, void *tag);
 583 extern int spa_open_rewind(const char *pool, spa_t **, void *tag,
 584     nvlist_t *policy, nvlist_t **config);
 585 extern int spa_get_stats(const char *pool, nvlist_t **config, char *altroot,
 586     size_t buflen);
 587 extern int spa_create(const char *pool, nvlist_t *config, nvlist_t *props,
 588     nvlist_t *zplprops);
 589 extern int spa_import_rootpool(char *devpath, char *devid);
 590 extern int spa_import(const char *pool, nvlist_t *config, nvlist_t *props,
 591     uint64_t flags);
 592 extern nvlist_t *spa_tryimport(nvlist_t *tryconfig);
 593 extern int spa_destroy(char *pool);
 594 extern int spa_export(char *pool, nvlist_t **oldconfig, boolean_t force,
 595     boolean_t hardforce);
 596 extern int spa_reset(char *pool);
 597 extern void spa_async_request(spa_t *spa, int flag);
 598 extern void spa_async_unrequest(spa_t *spa, int flag);
 599 extern void spa_async_suspend(spa_t *spa);
 600 extern void spa_async_resume(spa_t *spa);
 601 extern spa_t *spa_inject_addref(char *pool);
 602 extern void spa_inject_delref(spa_t *spa);
 603 extern void spa_scan_stat_init(spa_t *spa);
 604 extern int spa_scan_get_stats(spa_t *spa, pool_scan_stat_t *ps);

 606 #define SPA_ASYNC_CONFIG_UPDATE 0x01
 607 #define SPA_ASYNC_REMOVE        0x02
 608 #define SPA_ASYNC_PROBE         0x04
 609 #define SPA_ASYNC_RESILVER_DONE 0x08
 610 #define SPA_ASYNC_RESILVER      0x10
 611 #define SPA_ASYNC_AUTOEXPAND    0x20
 612 #define SPA_ASYNC_REMOVE_DONE   0x40
 613 #define SPA_ASYNC_REMOVE_STOP   0x80

 615 /*
 616  * Controls the behavior of spa_vdev_remove().
 617  */
 618 #define SPA_REMOVE_UNSPARE      0x01
 619 #define SPA_REMOVE_DONE         0x02

 621 /* device manipulation */
 622 extern int spa_vdev_add(spa_t *spa, nvlist_t *nvroot);
 623 extern int spa_vdev_attach(spa_t *spa, uint64_t guid, nvlist_t *nvroot,
 624     int replacing);
 625 extern int spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid,
 626     int replace_done);
 627 extern int spa_vdev_remove(spa_t *spa, uint64_t guid, boolean_t unspare);
```



```
 628 extern boolean_t spa_vdev_remove_active(spa_t *spa);
 629 extern int spa_vdev_setpath(spa_t *spa, uint64_t guid, const char *newpath);
 630 extern int spa_vdev_setfru(spa_t *spa, uint64_t guid, const char *newfru);
 631 extern int spa_vdev_split_mirror(spa_t *spa, char *newname, nvlist_t *config,
 632     nvlist_t *props, boolean_t exp);

 634 /* spare state (which is global across all pools) */
 635 extern void spa_spare_add(vdev_t *vd);
 636 extern void spa_spare_remove(vdev_t *vd);
 637 extern boolean_t spa_spare_exists(uint64_t guid, uint64_t *pool, int *refcnt);
 638 extern void spa_spare_activate(vdev_t *vd);

 640 /* L2ARC state (which is global across all pools) */
 641 extern void spa_l2cache_add(vdev_t *vd);
 642 extern void spa_l2cache_remove(vdev_t *vd);
 643 extern boolean_t spa_l2cache_exists(uint64_t guid, uint64_t *pool);
 644 extern void spa_l2cache_activate(vdev_t *vd);
 645 extern void spa_l2cache_drop(spa_t *spa);

 647 /* scanning */
 648 extern int spa_scan(spa_t *spa, pool_scan_func_t func);
 649 extern int spa_scan_stop(spa_t *spa);

 651 /* spa syncing */
 652 extern void spa_sync(spa_t *spa, uint64_t txg); /* only for DMU use */
 653 extern void spa_sync_allpools(void);

 655 /* spa namespace global mutex */
 656 extern kmutex_t spa_namespace_lock;

 658 /*
 659  * SPA configuration functions in spa_config.c
 660  */

 662 #define SPA_CONFIG_UPDATE_POOL  0
 663 #define SPA_CONFIG_UPDATE_VDEVS 1

 665 extern void spa_config_sync(spa_t *, boolean_t, boolean_t);
 666 extern void spa_config_load(void);
 667 extern nvlist_t *spa_all_configs(uint64_t *);
 668 extern void spa_config_set(spa_t *spa, nvlist_t *config);
 669 extern nvlist_t *spa_config_generate(spa_t *spa, vdev_t *vd, uint64_t txg,
 670     int getstats);
 671 extern void spa_config_update(spa_t *spa, int what);

 673 /*
 674  * Miscellaneous SPA routines in spa_misc.c
 675  */

 677 /* Namespace manipulation */
 678 extern spa_t *spa_lookup(const char *name);
 679 extern spa_t *spa_add(const char *name, nvlist_t *config, const char *altroot);
 680 extern void spa_remove(spa_t *spa);
 681 extern spa_t *spa_next(spa_t *prev);

 683 /* Refcount functions */
 684 extern void spa_open_ref(spa_t *spa, void *tag);
 685 extern void spa_close(spa_t *spa, void *tag);
 686 extern void spa_async_close(spa_t *spa, void *tag);
 687 extern boolean_t spa_refcount_zero(spa_t *spa);

 689 #define SCL_NONE        0x00
 690 #define SCL_CONFIG      0x01
 691 #define SCL_STATE       0x02
 692 #define SCL_L2ARC       0x04            /* hack until L2ARC 2.0 */
 693 #define SCL_ALLOC       0x08
```

```
694 #define SCL_ZIO         0x10
695 #define SCL_FREE        0x20
696 #define SCL_VDEV        0x40
697 #define SCL_LOCKS       7
698 #define SCL_ALL         ((1 << SCL_LOCKS) - 1)
699 #define SCL_STATE_ALL   (SCL_STATE | SCL_L2ARC | SCL_ZIO)

701 /* Pool configuration locks */
702 extern int spa_config_tryenter(spa_t *spa, int locks, void *tag, krw_t rw);
703 extern void spa_config_enter(spa_t *spa, int locks, void *tag, krw_t rw);
704 extern void spa_config_exit(spa_t *spa, int locks, void *tag);
705 extern int spa_config_held(spa_t *spa, int locks, krw_t rw);

707 /* Pool vdev add/remove lock */
708 extern uint64_t spa_vdev_enter(spa_t *spa);
709 extern uint64_t spa_vdev_config_enter(spa_t *spa);
710 extern void spa_vdev_config_exit(spa_t *spa, vdev_t *vd, uint64_t txg,
711     int error, char *tag);
712 extern int spa_vdev_exit(spa_t *spa, vdev_t *vd, uint64_t txg, int error);

714 /* Pool vdev state change lock */
715 extern void spa_vdev_state_enter(spa_t *spa, int oplock);
716 extern int spa_vdev_state_exit(spa_t *spa, vdev_t *vd, int error);

718 /* Log state */
719 typedef enum spa_log_state {
720         SPA_LOG_UNKNOWN = 0,    /* unknown log state */
721         SPA_LOG_MISSING,        /* missing log(s) */
722         SPA_LOG_CLEAR,          /* clear the log(s) */
723         SPA_LOG_GOOD,           /* log(s) are good */
724 } spa_log_state_t;

726 extern spa_log_state_t spa_get_log_state(spa_t *spa);
727 extern void spa_set_log_state(spa_t *spa, spa_log_state_t state);
728 extern int spa_offline_log(spa_t *spa);

730 /* Log claim callback */
731 extern void spa_claim_notify(zio_t *zio);

733 /* Accessor functions */
734 extern boolean_t spa_shutting_down(spa_t *spa);
735 extern struct dsl_pool *spa_get_dsl(spa_t *spa);
736 extern boolean_t spa_is_initializing(spa_t *spa);
737 extern blkptr_t *spa_get_rootblkptr(spa_t *spa);
738 extern void spa_set_rootblkptr(spa_t *spa, const blkptr_t *bp);
739 extern void spa_altroot(spa_t *, char *, size_t);
740 extern int spa_sync_pass(spa_t *spa);
741 extern char *spa_name(spa_t *spa);
742 extern uint64_t spa_guid(spa_t *spa);
743 extern uint64_t spa_load_guid(spa_t *spa);
744 extern uint64_t spa_last_synced_txg(spa_t *spa);
745 extern uint64_t spa_first_txg(spa_t *spa);
746 extern uint64_t spa_syncing_txg(spa_t *spa);
747 extern uint64_t spa_version(spa_t *spa);
748 extern pool_state_t spa_state(spa_t *spa);
749 extern spa_load_state_t spa_load_state(spa_t *spa);
750 extern uint64_t spa_freeze_txg(spa_t *spa);
751 extern uint64_t spa_get_asize(spa_t *spa, uint64_t lsize);
752 extern uint64_t spa_get_dspace(spa_t *spa);
753 extern uint64_t spa_get_slop_space(spa_t *spa);
754 extern void spa_update_dspace(spa_t *spa);
755 extern uint64_t spa_version(spa_t *spa);
756 extern boolean_t spa_deflate(spa_t *spa);
757 extern metaslab_class_t *spa_normal_class(spa_t *spa);
758 extern metaslab_class_t *spa_log_class(spa_t *spa);
759 extern void spa_evicting_os_register(spa_t *, objset_t *os);
```



```
760 extern void spa_evicting_os_deregister(spa_t *, objset_t *os);
761 extern void spa_evicting_os_wait(spa_t *spa);
762 extern int spa_max_replication(spa_t *spa);
763 extern int spa_prev_software_version(spa_t *spa);
764 extern int spa_busy(void);
765 extern uint8_t spa_get_failmode(spa_t *spa);
766 extern boolean_t spa_suspended(spa_t *spa);
767 extern uint64_t spa_bootfs(spa_t *spa);
768 extern uint64_t spa_delegation(spa_t *spa);
769 extern objset_t *spa_meta_objset(spa_t *spa);
770 extern uint64_t spa_deadman_synctime(spa_t *spa);

772 /* Miscellaneous support routines */
773 extern void spa_activate_mos_feature(spa_t *spa, const char *feature,
774     dmu_tx_t *tx);
775 extern void spa_deactivate_mos_feature(spa_t *spa, const char *feature);
776 extern int spa_rename(const char *oldname, const char *newname);
777 extern spa_t *spa_by_guid(uint64_t pool_guid, uint64_t device_guid);
778 extern boolean_t spa_guid_exists(uint64_t pool_guid, uint64_t device_guid);
779 extern char *spa_strdup(const char *);
780 extern void spa_strfree(char *);
781 extern uint64_t spa_get_random(uint64_t range);
782 extern uint64_t spa_generate_guid(spa_t *spa);
783 extern void snprintf_blkptr(char *buf, size_t buflen, const blkptr_t *bp);
784 extern void spa_freeze(spa_t *spa);
785 extern int spa_change_guid(spa_t *spa);
786 extern void spa_upgrade(spa_t *spa, uint64_t version);
787 extern void spa_evict_all(void);
788 extern vdev_t *spa_lookup_by_guid(spa_t *spa, uint64_t guid,
789     boolean_t l2cache);
790 extern boolean_t spa_has_spare(spa_t *, uint64_t guid);
791 extern uint64_t dva_get_dsize_sync(spa_t *spa, const dva_t *dva);
792 extern uint64_t bp_get_dsize_sync(spa_t *spa, const blkptr_t *bp);
793 extern uint64_t bp_get_dsize(spa_t *spa, const blkptr_t *bp);
794 extern boolean_t spa_has_slogs(spa_t *spa);
795 extern boolean_t spa_is_root(spa_t *spa);
796 extern boolean_t spa_writeable(spa_t *spa);
797 extern boolean_t spa_has_pending_synctask(spa_t *spa);
798 extern int spa_maxblocksize(spa_t *spa);
799 extern void zfs_blkptr_verify(spa_t *spa, const blkptr_t *bp);

801 extern int spa_mode(spa_t *spa);
802 extern uint64_t strtonum(const char *str, char **nptr);

804 extern char *spa_his_ievent_table[];

806 extern void spa_history_create_obj(spa_t *spa, dmu_tx_t *tx);
807 extern int spa_history_get(spa_t *spa, uint64_t *offset, uint64_t *len_read,
808     char *his_buf);
809 extern int spa_history_log(spa_t *spa, const char *his_buf);
810 extern int spa_history_log_nvl(spa_t *spa, nvlist_t *nvl);
811 extern void spa_history_log_version(spa_t *spa, const char *operation);
812 extern void spa_history_log_internal(spa_t *spa, const char *operation,
813     dmu_tx_t *tx, const char *fmt, ...);
814 extern void spa_history_log_internal_ds(struct dsl_dataset *ds, const char *op,
815     dmu_tx_t *tx, const char *fmt, ...);
816 extern void spa_history_log_internal_dd(dsl_dir_t *dd, const char *operation,
817     dmu_tx_t *tx, const char *fmt, ...);

819 /* error handling */
820 struct zbookmark_phys;
821 extern void spa_log_error(spa_t *spa, zio_t *zio);
822 extern void zfs_ereport_post(const char *class, spa_t *spa, vdev_t *vd,
823     zio_t *zio, uint64_t stateoroffset, uint64_t length);
824 extern void zfs_post_remove(spa_t *spa, vdev_t *vd);
825 extern void zfs_post_state_change(spa_t *spa, vdev_t *vd);
```

```
826 extern void zfs_post_autoreplace(spa_t *spa, vdev_t *vd);
827 extern uint64_t spa_get_errlog_size(spa_t *spa);
828 extern int spa_get_errlog(spa_t *spa, void *uaddr, size_t *count);
829 extern void spa_errlog_rotate(spa_t *spa);
830 extern void spa_errlog_drain(spa_t *spa);
831 extern void spa_errlog_sync(spa_t *spa, uint64_t txg);
832 extern void spa_get_errlists(spa_t *spa, avl_tree_t *last, avl_tree_t *scrub);

834 /* vdev cache */
835 extern void vdev_cache_stat_init(void);
836 extern void vdev_cache_stat_fini(void);

838 /* Initialization and termination */
839 extern void spa_init(int flags);
840 extern void spa_fini(void);
841 extern void spa_boot_init();

843 /* properties */
844 extern int spa_prop_set(spa_t *spa, nvlist_t *nvp);
845 extern int spa_prop_get(spa_t *spa, nvlist_t **nvp);
846 extern void spa_prop_clear_bootfs(spa_t *spa, uint64_t obj, dmu_tx_t *tx);
847 extern void spa_configfile_set(spa_t *, nvlist_t *, boolean_t);

849 /* asynchronous event notification */
850 extern void spa_event_notify(spa_t *spa, vdev_t *vdev, const char *name);

852 #ifdef ZFS_DEBUG
853 #define dprintf_bp(bp, fmt, ...) do {                           \
854         if (zfs_flags & ZFS_DEBUG_DPRINTF) {                    \
855         char *__blkbuf = kmem_alloc(BP_SPRINTF_LEN, KM_SLEEP);  \
856         snprintf_blkptr(__blkbuf, BP_SPRINTF_LEN, (bp));        \
857         dprintf(fmt " %s\n", __VA_ARGS__, __blkbuf);            \
858         kmem_free(__blkbuf, BP_SPRINTF_LEN);                    \
859         } \
860 _NOTE(CONSTCOND) } while (0)
861 #else
862 #define dprintf_bp(bp, fmt, ...)
863 #endif

865 extern boolean_t spa_debug_enabled(spa_t *spa);
866 #define spa_dbgmsg(spa, ...)                    \
867 {                                               \
868         if (spa_debug_enabled(spa))             \
869                 zfs_dbgmsg(__VA_ARGS__);        \
870 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   11562 Tue Jan  6 10:37:52 2015
new/usr/src/uts/common/fs/zfs/sys/spa_impl.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2011, 2014 by Delphix. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#ifndef _SYS_SPA_IMPL_H
#define _SYS_SPA_IMPL_H

#include <sys/spa.h>
#include <sys/vdev.h>
#include <sys/metaslab.h>
#include <sys/dmu.h>
#include <sys/dsl_pool.h>
#include <sys/uberblock_impl.h>
#include <sys/zfs_context.h>
#include <sys/avl.h>
#include <sys/refcount.h>
#include <sys/bplist.h>
#include <sys/bpobj.h>
#include <sys/zfeature.h>
#include <zfeature_common.h>

#ifdef  __cplusplus
extern "C" {
#endif

typedef struct spa_error_entry {
        zbookmark_phys_t        se_bookmark;
        char                    *se_name;
        avl_node_t              se_avl;
} spa_error_entry_t;
```

*unchanged_portion_omitted_*

```
struct spa {
        /*
```



```
         * Fields protected by spa_namespace_lock.
         */
        char            spa_name[MAXNAMELEN];   /* pool name */
        char            *spa_comment;           /* comment */
        avl_node_t      spa_avl;                /* node in spa_namespace_avl */
        nvlist_t        *spa_config;            /* last synced config */
        nvlist_t        *spa_config_syncing;    /* currently syncing config */
        nvlist_t        *spa_config_splitting;  /* config for splitting */
        nvlist_t        *spa_load_info;         /* info and errors from load */
        uint64_t        spa_config_txg;         /* txg of last config change */
        int             spa_sync_pass;          /* iterate-to-convergence */
        pool_state_t    spa_state;              /* pool state */
        int             spa_inject_ref;         /* injection references */
        uint8_t         spa_sync_on;            /* sync threads are running */
        spa_load_state_t spa_load_state;        /* current load operation */
        uint64_t        spa_import_flags;       /* import specific flags */
        spa_taskqs_t    spa_zio_taskq[ZIO_TYPES][ZIO_TASKQ_TYPES];
        dsl_pool_t      *spa_dsl_pool;
        boolean_t       spa_is_initializing;    /* true while opening pool */
        metaslab_class_t *spa_normal_class;     /* normal data class */
        metaslab_class_t *spa_log_class;        /* intent log data class */
        uint64_t        spa_first_txg;          /* first txg after spa_open() */
        uint64_t        spa_final_txg;          /* txg of export/destroy */
        uint64_t        spa_freeze_txg;         /* freeze pool at this txg */
        uint64_t        spa_load_max_txg;       /* best initial ub_txg */
        uint64_t        spa_claim_max_txg;      /* highest claimed birth txg */
        timespec_t      spa_loaded_ts;          /* 1st successful open time */
        objset_t        *spa_meta_objset;       /* copy of dp->dp_meta_objset */
        kmutex_t        spa_evicting_os_lock;   /* Evicting objset list lock */
        list_t          spa_evicting_os_list;   /* Objsets being evicted. */
        kcondvar_t      spa_evicting_os_cv;     /* Objset Eviction Completion */
        txg_list_t      spa_vdev_txg_list;      /* per-txg dirty vdev list */
        vdev_t          *spa_root_vdev;         /* top-level vdev container */
        uint64_t        spa_config_guid;        /* config pool guid */
        uint64_t        spa_load_guid;          /* spa_load initialized guid */
        uint64_t        spa_last_synced_guid;   /* last synced guid */
        list_t          spa_config_dirty_list;  /* vdevs with dirty config */
        list_t          spa_state_dirty_list;   /* vdevs with dirty state */
        spa_aux_vdev_t  spa_spares;             /* hot spares */
        spa_aux_vdev_t  spa_l2cache;            /* L2ARC cache devices */
        nvlist_t        *spa_label_features;    /* Features for reading MOS */
        uint64_t        spa_config_object;      /* MOS object for pool config */
        uint64_t        spa_config_generation;  /* config generation number */
        uint64_t        spa_syncing_txg;        /* txg currently syncing */
        bpobj_t         spa_deferred_bpobj;     /* deferred-free bplist */
        bplist_t        spa_free_bplist[TXG_SIZE]; /* bplist of stuff to free */
        uberblock_t     spa_ubsync;             /* last synced uberblock */
        uberblock_t     spa_uberblock;          /* current uberblock */
        boolean_t       spa_extreme_rewind;     /* rewind past deferred frees */
        uint64_t        spa_last_io;            /* lbolt of last non-scan I/O */
        kmutex_t        spa_scrub_lock;         /* resilver/scrub lock */
        uint64_t        spa_scrub_inflight;     /* in-flight scrub I/Os */
        kcondvar_t      spa_scrub_io_cv;        /* scrub I/O completion */
        uint8_t         spa_scrub_active;       /* active or suspended? */
        uint8_t         spa_scrub_type;         /* type of scrub we're doing */
        uint8_t         spa_scrub_finished;     /* indicator to rotate logs */
        uint8_t         spa_scrub_started;      /* started since last boot */
        uint8_t         spa_scrub_reopen;       /* scrub doing vdev_reopen */
        uint64_t        spa_scan_pass_start;    /* start time per pass/reboot */
        uint64_t        spa_scan_pass_exam;     /* examined bytes per pass */
        kmutex_t        spa_async_lock;         /* protect async state */
        kthread_t       *spa_async_thread;      /* thread doing async task */
        int             spa_async_suspended;    /* async tasks suspended */
        kcondvar_t      spa_async_cv;           /* wait for thread_exit() */
        uint16_t        spa_async_tasks;        /* async task mask */
        char            *spa_root;              /* alternate root directory */
```

```
186         uint64_t        spa_ena;                /* spa-wide ereport ENA */
187         int             spa_last_open_failed;   /* error if last open failed */
188         uint64_t        spa_last_ubsync_txg;    /* "best" uberblock txg */
189         uint64_t        spa_last_ubsync_txg_ts; /* timestamp from that ub */
190         uint64_t        spa_load_txg;           /* ub txg that loaded */
191         uint64_t        spa_load_txg_ts;        /* timestamp from that ub */
192         uint64_t        spa_load_meta_errors;   /* verify metadata err count */
193         uint64_t        spa_load_data_errors;   /* verify data err count */
194         uint64_t        spa_verify_min_txg;     /* start txg of verify scrub */
195         kmutex_t        spa_errlog_lock;        /* error log lock */
196         uint64_t        spa_errlog_last;        /* last error log object */
197         uint64_t        spa_errlog_scrub;       /* scrub error log object */
198         kmutex_t        spa_errlist_lock;       /* error list/ereport lock */
199         avl_tree_t      spa_errlist_last;       /* last error list */
200         avl_tree_t      spa_errlist_scrub;      /* scrub error list */
201         uint64_t        spa_deflate;            /* should we deflate? */
202         uint64_t        spa_history;            /* history object */
203         kmutex_t        spa_history_lock;       /* history lock */
204         vdev_t          *spa_pending_vdev;      /* pending vdev additions */
205         kmutex_t        spa_props_lock;         /* property lock */
206         uint64_t        spa_pool_props_object;  /* object for properties */
207         uint64_t        spa_bootfs;             /* default boot filesystem */
208         uint64_t        spa_failmode;           /* failure mode for the pool */
209         uint64_t        spa_delegation;         /* delegation on/off */
210         list_t          spa_config_list;        /* previous cache file(s) */
211         /* per-CPU array of root of async I/O: */
212         zio_t           **spa_async_zio_root;
213         zio_t           *spa_suspend_zio_root;  /* root of all suspended I/O */
214         kmutex_t        spa_suspend_lock;       /* protects suspend_zio_root */
215         kcondvar_t      spa_suspend_cv;         /* notification of resume */
216         uint8_t         spa_suspended;          /* pool is suspended */
217         uint8_t         spa_claiming;           /* pool is doing zil_claim() */
218         boolean_t       spa_debug;              /* debug enabled? */
219         boolean_t       spa_is_root;            /* pool is root */
220         int             spa_minref;             /* num refs when first opened */
221         int             spa_mode;               /* FREAD | FWRITE */
222         spa_log_state_t spa_log_state;          /* log state */
223         uint64_t        spa_autoexpand;         /* lun expansion on/off */
224         ddt_t           *spa_ddt[ZIO_CHECKSUM_FUNCTIONS]; /* in-core DDTs */
225         uint64_t        spa_ddt_stat_object;    /* DDT statistics */
226         uint64_t        spa_dedup_ditto;        /* dedup ditto threshold */
227         uint64_t        spa_dedup_checksum;     /* default dedup checksum */
228         uint64_t        spa_dspace;             /* dspace in normal class */
229         kmutex_t        spa_vdev_top_lock;      /* dueling offline/remove */
230         kmutex_t        spa_proc_lock;          /* protects spa_proc* */
231         kcondvar_t      spa_proc_cv;            /* spa_proc_state transitions */
232         spa_proc_state_t spa_proc_state;        /* see definition */
233         struct proc     *spa_proc;              /* "zpool-poolname" process */
234         uint64_t        spa_did;                /* if procp != p0, did of t1 */
235         boolean_t       spa_autoreplace;        /* autoreplace set in open */
236         int             spa_vdev_locks;         /* locks grabbed */
237         uint64_t        spa_creation_version;   /* version at pool creation */
238         uint64_t        spa_prev_software_version; /* See ub_software_version */
239         uint64_t        spa_feat_for_write_obj; /* required to write to pool */
240         uint64_t        spa_feat_for_read_obj;  /* required to read from pool */
241         uint64_t        spa_feat_desc_obj;      /* Feature descriptions */
242         uint64_t        spa_feat_enabled_txg_obj; /* Feature enabled txg */
243         /* cache feature refcounts */
244         uint64_t        spa_feat_refcount_cache[SPA_FEATURES];
245         cyclic_id_t     spa_deadman_cycid;      /* cyclic id */
246         uint64_t        spa_deadman_calls;      /* number of deadman calls */
247         hrtime_t        spa_sync_starttime;     /* starting time fo spa_sync */
248         uint64_t        spa_deadman_synctime;   /* deadman expiration timer */

250         /*
251          * spa_iokstat_lock protects spa_iokstat and
```



```
252          * spa_queue_stats[].
253          */
254         kmutex_t        spa_iokstat_lock;
255         struct kstat    *spa_iokstat;           /* kstat of io to this pool */
256         struct {
257                 int spa_active;
258                 int spa_queued;
259         } spa_queue_stats[ZIO_PRIORITY_NUM_QUEUEABLE];

261         hrtime_t        spa_ccw_fail_time;      /* Conf cache write fail time */

263         /*
264          * spa_refcount & spa_config_lock must be the last elements
265          * because refcount_t changes size based on compilation options.
266          * In order for the MDB module to function correctly, the other
267          * fields must remain in the same location.
268          */
269         spa_config_lock_t spa_config_lock[SCL_LOCKS]; /* config changes */
270         refcount_t      spa_refcount;           /* number of opens */
271 };
```

_____***unchanged_portion_omitted_***

****
6904 Tue Jan 6 10:37:52 2015
new/usr/src/uts/common/fs/zfs/sys/zap_impl.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#ifndef _SYS_ZAP_IMPL_H
#define _SYS_ZAP_IMPL_H

#include <sys/zap.h>
#include <sys/zfs_context.h>
#include <sys/avl.h>

#ifdef  __cplusplus
extern "C" {
#endif

extern int fzap_default_block_shift;

#define ZAP_MAGIC 0x2F52AB2ABULL

#define FZAP_BLOCK_SHIFT(zap)   ((zap)->zap_f.zap_block_shift)

#define MZAP_ENT_LEN            64
#define MZAP_NAME_LEN           (MZAP_ENT_LEN - 8 - 4 - 2)
#define MZAP_MAX_BLKSZ          SPA_OLD_MAXBLOCKSIZE

#define ZAP_NEED_CD             (-1U)

typedef struct mzap_ent_phys {
        uint64_t mze_value;
        uint32_t mze_cd;
        uint16_t mze_pad;       /* in case we want to chain them someday */
        char mze_name[MZAP_NAME_LEN];
} mzap_ent_phys_t;
```

*unchanged_portion_omitted_*


```
typedef struct zap_table_phys zap_table_phys_t;

typedef struct zap {
        dmu_buf_user_t zap_dbu;
        objset_t *zap_objset;
        uint64_t zap_object;
        struct dmu_buf *zap_dbuf;
        krwlock_t zap_rwlock;
        boolean_t zap_ismicro;
        int zap_normflags;
        uint64_t zap_salt;
        union {
                struct {
                        /*
                         * zap_num_entries_mtx protects
                         * zap_num_entries
                         */
                        kmutex_t zap_num_entries_mtx;
                        int zap_block_shift;
                } zap_fat;
                struct {
                        int16_t zap_num_entries;
                        int16_t zap_num_chunks;
                        int16_t zap_alloc_next;
                        avl_tree_t zap_avl;
                } zap_micro;
        } zap_u;
} zap_t;
```

*unchanged_portion_omitted_*

```
#define zap_f   zap_u.zap_fat
#define zap_m   zap_u.zap_micro

boolean_t zap_match(zap_name_t *zn, const char *matchname);
int zap_lockdir(objset_t *os, uint64_t obj, dmu_tx_t *tx,
    krw_t lti, boolean_t fatreader, boolean_t adding, zap_t **zapp);
void zap_unlockdir(zap_t *zap);
void zap_evict(void *dbu);
void zap_evict(dmu_buf_t *db, void *vmzap);
zap_name_t *zap_name_alloc(zap_t *zap, const char *key, matchtype_t mt);
void zap_name_free(zap_name_t *zn);
int zap_hashbits(zap_t *zap);
uint32_t zap_maxcd(zap_t *zap);
uint64_t zap_getflags(zap_t *zap);

#define ZAP_HASH_IDX(hash, n) (((n) == 0) ? 0 : ((hash) >> (64 - (n))))

void fzap_byteswap(void *buf, size_t size);
int fzap_count(zap_t *zap, uint64_t *count);
int fzap_lookup(zap_name_t *zn,
    uint64_t integer_size, uint64_t num_integers, void *buf,
    char *realname, int rn_len, boolean_t *normalization_conflictp);
void fzap_prefetch(zap_name_t *zn);
int fzap_count_write(zap_name_t *zn, int add, uint64_t *towrite,
    uint64_t *tooverwrite);
int fzap_add(zap_name_t *zn, uint64_t integer_size, uint64_t num_integers,
    const void *val, dmu_tx_t *tx);
int fzap_update(zap_name_t *zn,
    int integer_size, uint64_t num_integers, const void *val, dmu_tx_t *tx);
int fzap_length(zap_name_t *zn,
    uint64_t *integer_size, uint64_t *num_integers);
int fzap_remove(zap_name_t *zn, dmu_tx_t *tx);
int fzap_cursor_retrieve(zap_t *zap, zap_cursor_t *zc, zap_attribute_t *za);
void fzap_get_stats(zap_t *zap, zap_stats_t *zs);
void zap_put_leaf(struct zap_leaf *l);
```

```
 229 int fzap_add_cd(zap_name_t *zn,
 230     uint64_t integer_size, uint64_t num_integers,
 231     const void *val, uint32_t cd, dmu_tx_t *tx);
 232 void fzap_upgrade(zap_t *zap, dmu_tx_t *tx, zap_flags_t flags);

 234 #ifdef  __cplusplus
 235 }
```

unchanged_portion_omitted_

```
**********************************************************
    7713 Tue Jan  6 10:37:53 2015
new/usr/src/uts/common/fs/zfs/sys/zap_leaf.h
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2014 Spectra Logic Corporation, All rights reserved.
 */

#ifndef _SYS_ZAP_LEAF_H
#define _SYS_ZAP_LEAF_H

#include <sys/zap.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct zap;
struct zap_name;
struct zap_stats;

#define ZAP_LEAF_MAGIC 0x2AB1EAF

/* chunk size = 24 bytes */
#define ZAP_LEAF_CHUNKSIZE 24

/*
 * The amount of space available for chunks is:
 * block size (1<<l->l_bs) - hash entry size (2) * number of hash
 * entries - header space (2*chunksize)
 */
#define ZAP_LEAF_NUMCHUNKS(l) \
        (((1<<(l)->l_bs) - 2*ZAP_LEAF_HASH_NUMENTRIES(l)) / \
        ZAP_LEAF_CHUNKSIZE - 2)

/*
 * The amount of space within the chunk available for the array is:
 * chunk size - space for type (1) - space for next pointer (2)
 */
#define ZAP_LEAF_ARRAY_BYTES (ZAP_LEAF_CHUNKSIZE - 3)
```



```
#define ZAP_LEAF_ARRAY_NCHUNKS(bytes) \
        (((bytes)+ZAP_LEAF_ARRAY_BYTES-1)/ZAP_LEAF_ARRAY_BYTES)

/*
 * Low water mark:  when there are only this many chunks free, start
 * growing the ptrtbl.  Ideally, this should be larger than a
 * "reasonably-sized" entry.  20 chunks is more than enough for the
 * largest directory entry (MAXNAMELEN (256) byte name, 8-byte value),
 * while still being only around 3% for 16k blocks.
 */
#define ZAP_LEAF_LOW_WATER (20)

/*
 * The leaf hash table has block size / 2^5 (32) number of entries,
 * which should be more than enough for the maximum number of entries,
 * which is less than block size / CHUNKSIZE (24) / minimum number of
 * chunks per entry (3).
 */
#define ZAP_LEAF_HASH_SHIFT(l) ((l)->l_bs - 5)
#define ZAP_LEAF_HASH_NUMENTRIES(l) (1 << ZAP_LEAF_HASH_SHIFT(l))

/*
 * The chunks start immediately after the hash table.  The end of the
 * hash table is at l_hash + HASH_NUMENTRIES, which we simply cast to a
 * chunk_t.
 */
#define ZAP_LEAF_CHUNK(l, idx) \
        ((zap_leaf_chunk_t *) \
        (zap_leaf_phys(l)->l_hash + ZAP_LEAF_HASH_NUMENTRIES(l)))[idx]
#define ZAP_LEAF_ENTRY(l, idx) (&ZAP_LEAF_CHUNK(l, idx).l_entry)

typedef enum zap_chunk_type {
        ZAP_CHUNK_FREE = 253,
        ZAP_CHUNK_ENTRY = 252,
        ZAP_CHUNK_ARRAY = 251,
        ZAP_CHUNK_TYPE_MAX = 250
} zap_chunk_type_t;
```

*unchanged_portion_omitted_*

```
typedef struct zap_leaf {
        dmu_buf_user_t l_dbu;
        krwlock_t l_rwlock;
        uint64_t l_blkid;               /* 1<<ZAP_BLOCK_SHIFT byte block off */
        int l_bs;                       /* block size shift */
        dmu_buf_t *l_dbuf;
} zap_leaf_t;
```

*unchanged_portion_omitted_*

```
**********************************************************
   33413 Tue Jan  6 10:37:53 2015
new/usr/src/uts/common/fs/zfs/zap.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

/*
 * This file contains the top half of the zfs directory structure
 * implementation. The bottom half is in zap_leaf.c.
 *
 * The zdir is an extendable hash data structure. There is a table of
 * pointers to buckets (zap_t->zd_data->zd_leafs). The buckets are
 * each a constant size and hold a variable number of directory entries.
 * The buckets (aka "leaf nodes") are implemented in zap_leaf.c.
 *
 * The pointer table holds a power of 2 number of pointers.
 * (1<<zap_t->zd_data->zd_phys->zd_prefix_len).  The bucket pointed to
 * by the pointer at index i in the table holds entries whose hash value
 * has a zd_prefix_len - bit prefix
 */

#include <sys/spa.h>
#include <sys/dmu.h>
#include <sys/zfs_context.h>
#include <sys/zfs_znode.h>
#include <sys/fs/zfs.h>
#include <sys/zap.h>
#include <sys/refcount.h>
#include <sys/zap_impl.h>
#include <sys/zap_leaf.h>

int fzap_default_block_shift = 14; /* 16k blocksize */

extern inline zap_phys_t *zap_f_phys(zap_t *zap);

static void zap_leaf_pageout(dmu_buf_t *db, void *vl);
static uint64_t zap_allocate_blocks(zap_t *zap, int nblocks);
```



```
void
fzap_byteswap(void *vbuf, size_t size)
{
        uint64_t block_type;

        block_type = *(uint64_t *)vbuf;

        if (block_type == ZBT_LEAF || block_type == BSWAP_64(ZBT_LEAF))
                zap_leaf_byteswap(vbuf, size);
        else {
                /* it's a ptrtbl block */
                byteswap_uint64_array(vbuf, size);
        }
}

void
fzap_upgrade(zap_t *zap, dmu_tx_t *tx, zap_flags_t flags)
{
        dmu_buf_t *db;
        zap_leaf_t *l;
        int i;
        zap_phys_t *zp;

        ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
        zap->zap_ismicro = FALSE;

        zap->zap_dbu.dbu_evict_func = zap_evict;
        (void) dmu_buf_update_user(zap->zap_dbuf, zap, zap, zap_evict);

        mutex_init(&zap->zap_f.zap_num_entries_mtx, 0, 0, 0);
        zap->zap_f.zap_block_shift = highbit64(zap->zap_dbuf->db_size) - 1;

        zp = zap_f_phys(zap);
        /*
         * explicitly zero it since it might be coming from an
         * initialized microzap
         */
        bzero(zap->zap_dbuf->db_data, zap->zap_dbuf->db_size);
        zp->zap_block_type = ZBT_HEADER;
        zp->zap_magic = ZAP_MAGIC;

        zp->zap_ptrtbl.zt_shift = ZAP_EMBEDDED_PTRTBL_SHIFT(zap);

        zp->zap_freeblk = 2;            /* block 1 will be the first leaf */
        zp->zap_num_leafs = 1;
        zp->zap_num_entries = 0;
        zp->zap_salt = zap->zap_salt;
        zp->zap_normflags = zap->zap_normflags;
        zp->zap_flags = flags;

        /* block 1 will be the first leaf */
        for (i = 0; i < (1<<zp->zap_ptrtbl.zt_shift); i++)
                ZAP_EMBEDDED_PTRTBL_ENT(zap, i) = 1;

        /*
         * set up block 1 - the first leaf
         */
        VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
            1<<FZAP_BLOCK_SHIFT(zap), FTAG, &db, DMU_READ_NO_PREFETCH));
        dmu_buf_will_dirty(db, tx);

        l = kmem_zalloc(sizeof (zap_leaf_t), KM_SLEEP);
        l->l_dbuf = db;

        zap_leaf_init(l, zp->zap_normflags != 0);
```

```
123         kmem_free(l, sizeof (zap_leaf_t));
124         dmu_buf_rele(db, FTAG);
125 }
```

*unchanged_portion_omitted_*

```
390 static void
391 zap_leaf_pageout(void *dbu)
392 {
393         zap_leaf_t *l = dbu;

395         rw_destroy(&l->l_rwlock);
396         kmem_free(l, sizeof (zap_leaf_t));
397 }

399 static zap_leaf_t *
400 zap_create_leaf(zap_t *zap, dmu_tx_t *tx)
401 {
402         void *winner;
403         zap_leaf_t *l = kmem_zalloc(sizeof (zap_leaf_t), KM_SLEEP);
394         zap_leaf_t *l = kmem_alloc(sizeof (zap_leaf_t), KM_SLEEP);

405         ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));

407         rw_init(&l->l_rwlock, 0, 0, 0);
408         rw_enter(&l->l_rwlock, RW_WRITER);
409         l->l_blkid = zap_allocate_blocks(zap, 1);
410         l->l_dbuf = NULL;

412         VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
413             l->l_blkid << FZAP_BLOCK_SHIFT(zap), NULL, &l->l_dbuf,
414             DMU_READ_NO_PREFETCH));
415         dmu_buf_init_user(&l->l_dbu, zap_leaf_pageout, &l->l_dbuf);
416         winner = dmu_buf_set_user(l->l_dbuf, &l->l_dbu);
406         winner = dmu_buf_set_user(l->l_dbuf, l, zap_leaf_pageout);
417         ASSERT(winner == NULL);
418         dmu_buf_will_dirty(l->l_dbuf, tx);

420         zap_leaf_init(l, zap->zap_normflags != 0);

422         zap_f_phys(zap)->zap_num_leafs++;

424         return (l);
425 }
```

*unchanged_portion_omitted_*

```
438 _NOTE(ARGSUSED(0))
439 static void
440 zap_leaf_pageout(dmu_buf_t *db, void *vl)
441 {
442         zap_leaf_t *l = vl;

444         rw_destroy(&l->l_rwlock);
445         kmem_free(l, sizeof (zap_leaf_t));
446 }

448 static zap_leaf_t *
449 zap_open_leaf(uint64_t blkid, dmu_buf_t *db)
450 {
451         zap_leaf_t *l, *winner;

453         ASSERT(blkid != 0);

455         l = kmem_zalloc(sizeof (zap_leaf_t), KM_SLEEP);
455         l = kmem_alloc(sizeof (zap_leaf_t), KM_SLEEP);
456         rw_init(&l->l_rwlock, 0, 0, 0);
```



```
457         rw_enter(&l->l_rwlock, RW_WRITER);
458         l->l_blkid = blkid;
459         l->l_bs = highbit64(db->db_size) - 1;
460         l->l_dbuf = db;

462         dmu_buf_init_user(&l->l_dbu, zap_leaf_pageout, &l->l_dbuf);
463         winner = dmu_buf_set_user(db, &l->l_dbu);
462         winner = dmu_buf_set_user(db, l, zap_leaf_pageout);

465         rw_exit(&l->l_rwlock);
466         if (winner != NULL) {
467                 /* someone else set it first */
468                 zap_leaf_pageout(&l->l_dbu);
467                 zap_leaf_pageout(NULL, l);
469                 l = winner;
470         }

472         /*
473          * lhr_pad was previously used for the next leaf in the leaf
474          * chain.  There should be no chained leafs (as we have removed
475          * support for them).
476          */
477         ASSERT0(zap_leaf_phys(l)->l_hdr.lh_pad1);

479         /*
480          * There should be more hash entries than there can be
481          * chunks to put in the hash table
482          */
483         ASSERT3U(ZAP_LEAF_HASH_NUMENTRIES(l), >, ZAP_LEAF_NUMCHUNKS(l) / 3);

485         /* The chunks should begin at the end of the hash table */
486         ASSERT3P(&ZAP_LEAF_CHUNK(l, 0), ==,
487             &zap_leaf_phys(l)->l_hash[ZAP_LEAF_HASH_NUMENTRIES(l)]);

489         /* The chunks should end at the end of the block */
490         ASSERT3U((uintptr_t)&ZAP_LEAF_CHUNK(l, ZAP_LEAF_NUMCHUNKS(l)) -
491             (uintptr_t)zap_leaf_phys(l), ==, l->l_dbuf->db_size);

493         return (l);
494 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   34313 Tue Jan  6 10:37:53 2015
new/usr/src/uts/common/fs/zfs/zap_micro.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2011, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 */

#include <sys/zio.h>
#include <sys/spa.h>
#include <sys/dmu.h>
#include <sys/zfs_context.h>
#include <sys/zap.h>
#include <sys/refcount.h>
#include <sys/zap_impl.h>
#include <sys/zap_leaf.h>
#include <sys/avl.h>
#include <sys/arc.h>
#include <sys/dmu_objset.h>

#ifdef _KERNEL
#include <sys/sunddi.h>
#endif

extern inline mzap_phys_t *zap_m_phys(zap_t *zap);

static int mzap_upgrade(zap_t **zapp, dmu_tx_t *tx, zap_flags_t flags);

uint64_t
zap_getflags(zap_t *zap)
{
        if (zap->zap_ismicro)
                return (0);
        return (zap_f_phys(zap)->zap_flags);
}
```

*unchanged_portion_omitted_*

```
static zap_t *
mzap_open(objset_t *os, uint64_t obj, dmu_buf_t *db)
```



```
{
        zap_t *winner;
        zap_t *zap;
        int i;

        ASSERT3U(MZAP_ENT_LEN, ==, sizeof (mzap_ent_phys_t));

        zap = kmem_zalloc(sizeof (zap_t), KM_SLEEP);
        rw_init(&zap->zap_rwlock, 0, 0, 0);
        rw_enter(&zap->zap_rwlock, RW_WRITER);
        zap->zap_objset = os;
        zap->zap_object = obj;
        zap->zap_dbuf = db;

        if (*(uint64_t *)db->db_data != ZBT_MICRO) {
                mutex_init(&zap->zap_f.zap_num_entries_mtx, 0, 0, 0);
                zap->zap_f.zap_block_shift = highbit64(db->db_size) - 1;
        } else {
                zap->zap_ismicro = TRUE;
        }

        /*
         * Make sure that zap_ismicro is set before we let others see
         * it, because zap_lockdir() checks zap_ismicro without the lock
         * held.
         */
        dmu_buf_init_user(&zap->zap_dbu, zap_evict, &zap->zap_dbuf);
        winner = dmu_buf_set_user(db, &zap->zap_dbu);
        winner = dmu_buf_set_user(db, zap, zap_evict);

        if (winner != NULL) {
                rw_exit(&zap->zap_rwlock);
                rw_destroy(&zap->zap_rwlock);
                if (!zap->zap_ismicro)
                        mutex_destroy(&zap->zap_f.zap_num_entries_mtx);
                kmem_free(zap, sizeof (zap_t));
                return (winner);
        }

        if (zap->zap_ismicro) {
                zap->zap_salt = zap_m_phys(zap)->mz_salt;
                zap->zap_normflags = zap_m_phys(zap)->mz_normflags;
                zap->zap_m.zap_num_chunks = db->db_size / MZAP_ENT_LEN - 1;
                avl_create(&zap->zap_m.zap_avl, mze_compare,
                    sizeof (mzap_ent_t), offsetof(mzap_ent_t, mze_node));

                for (i = 0; i < zap->zap_m.zap_num_chunks; i++) {
                        mzap_ent_phys_t *mze =
                            &zap_m_phys(zap)->mz_chunk[i];
                        if (mze->mze_name[0]) {
                                zap_name_t *zn;

                                zap->zap_m.zap_num_entries++;
                                zn = zap_name_alloc(zap, mze->mze_name,
                                    MT_EXACT);
                                mze_insert(zap, i, zn->zn_hash);
                                zap_name_free(zn);
                        }
                }
        } else {
                zap->zap_salt = zap_f_phys(zap)->zap_salt;
                zap->zap_normflags = zap_f_phys(zap)->zap_normflags;

                ASSERT3U(sizeof (struct zap_leaf_header), ==,
                    2*ZAP_LEAF_CHUNKSIZE);
```

```
430                 /*
431                  * The embedded pointer table should not overlap the
432                  * other members.
433                  */
434                 ASSERT3P(&ZAP_EMBEDDED_PTRTBL_ENT(zap, 0), >,
435                     &zap_f_phys(zap)->zap_salt);

437                 /*
438                  * The embedded pointer table should end at the end of
439                  * the block
440                  */
441                 ASSERT3U((uintptr_t)&ZAP_EMBEDDED_PTRTBL_ENT(zap,
442                     1<<ZAP_EMBEDDED_PTRTBL_SHIFT(zap)) -
443                     (uintptr_t)zap_f_phys(zap), ==,
444                     zap->zap_dbuf->db_size);
445         }
446         rw_exit(&zap->zap_rwlock);
447         return (zap);
448 }
```

_____***unchanged_portion_omitted_***

```
682 _NOTE(ARGSUSED(0))
684 void
685 zap_evict(void *dbu)
684 zap_evict(dmu_buf_t *db, void *vzap)
686 {
687         zap_t *zap = dbu;
686         zap_t *zap = vzap;

689         rw_destroy(&zap->zap_rwlock);

691         if (zap->zap_ismicro)
692                 mze_destroy(zap);
693         else
694                 mutex_destroy(&zap->zap_f.zap_num_entries_mtx);

696         kmem_free(zap, sizeof (zap_t));
697 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10515 Tue Jan  6 10:37:53 2015
new/usr/src/uts/common/fs/zfs/zfs_sa.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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.
 */

#include <sys/zfs_context.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/vnode.h>
#include <sys/sa.h>
#include <sys/zfs_acl.h>
#include <sys/zfs_sa.h>

/*
 * ZPL attribute registration table.
 * Order of attributes doesn't matter
 * a unique value will be assigned for each
 * attribute that is file system specific
 *
 * This is just the set of ZPL attributes that this
 * version of ZFS deals with natively.  The file system
 * could have other attributes stored in files, but they will be
 * ignored.  The SA framework will preserve them, just that
 * this version of ZFS won't change or delete them.
 */

sa_attr_reg_t zfs_attr_table[ZPL_END+1] = {
        {"ZPL_ATIME", sizeof (uint64_t) * 2, SA_UINT64_ARRAY, 0},
        {"ZPL_MTIME", sizeof (uint64_t) * 2, SA_UINT64_ARRAY, 1},
        {"ZPL_CTIME", sizeof (uint64_t) * 2, SA_UINT64_ARRAY, 2},
        {"ZPL_CRTIME", sizeof (uint64_t) * 2, SA_UINT64_ARRAY, 3},
        {"ZPL_GEN", sizeof (uint64_t), SA_UINT64_ARRAY, 4},
        {"ZPL_MODE", sizeof (uint64_t), SA_UINT64_ARRAY, 5},
        {"ZPL_SIZE", sizeof (uint64_t), SA_UINT64_ARRAY, 6},
        {"ZPL_PARENT", sizeof (uint64_t), SA_UINT64_ARRAY, 7},
        {"ZPL_LINKS", sizeof (uint64_t), SA_UINT64_ARRAY, 8},
        {"ZPL_XATTR", sizeof (uint64_t), SA_UINT64_ARRAY, 9},
        {"ZPL_RDEV", sizeof (uint64_t), SA_UINT64_ARRAY, 10},
```



```
        {"ZPL_FLAGS", sizeof (uint64_t), SA_UINT64_ARRAY, 11},
        {"ZPL_UID", sizeof (uint64_t), SA_UINT64_ARRAY, 12},
        {"ZPL_GID", sizeof (uint64_t), SA_UINT64_ARRAY, 13},
        {"ZPL_PAD", sizeof (uint64_t) * 4, SA_UINT64_ARRAY, 14},
        {"ZPL_ZNODE_ACL", 88, SA_UINT8_ARRAY, 15},
        {"ZPL_DACL_COUNT", sizeof (uint64_t), SA_UINT64_ARRAY, 0},
        {"ZPL_SYMLINK", 0, SA_UINT8_ARRAY, 0},
        {"ZPL_SCANSTAMP", 32, SA_UINT8_ARRAY, 0},
        {"ZPL_DACL_ACES", 0, SA_ACL, 0},
        {NULL, 0, 0, 0}
};
______unchanged_portion_omitted_
```

****
58068 Tue Jan 6 10:37:53 2015
new/usr/src/uts/common/fs/zfs/zil.c
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.com>
****

_____***unchanged_portion_omitted_***

```
 467 /*
 468  * Called when we create in-memory log transactions so that we know
 469  * to cleanup the itxs at the end of spa_sync().
 470  */
 471 void
 472 zilog_dirty(zilog_t *zilog, uint64_t txg)
 473 {
 474         dsl_pool_t *dp = zilog->zl_dmu_pool;
 475         dsl_dataset_t *ds = dmu_objset_ds(zilog->zl_os);

 477         if (ds->ds_is_snapshot)
 477         if (dsl_dataset_is_snapshot(ds))
 478                 panic("dirtying snapshot!");

 480         if (txg_list_add(&dp->dp_dirty_zilogs, zilog, txg)) {
 481                 /* up the hold count until we can be written out */
 482                 dmu_buf_add_ref(ds->ds_dbuf, zilog);
 483         }
 484 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    2374 Tue Jan  6 10:37:54 2015
new/usr/src/uts/sparc/stmf_sbd/Makefile
5056 ZFS deadlock on db_mtx and dn_holds
Reviewed by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matt Ahrens <mahrens@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
Approved by: Dan McDonald <danmcd@omniti.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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
#
#       This makefile drives the production of the stmf_sbd driver for
#       COMSTAR.

#
#       Path to the base of the uts directory tree (usually /usr/src/uts).
#
UTSBASE = ../..

ARCHDIR:sh = cd ..; basename ‘pwd‘

#
#       Define the module and object file sets.
#
MODULE          = stmf_sbd
OBJECTS         = $(STMF_SBD_OBJS:%=$(OBJS_DIR)/%)
LINTS           = $(STMF_SBD_OBJS:%.o=$(LINTS_DIR)/%.ln)
ROOTMODULE      = $(ROOT_DRV_DIR)/$(MODULE)
CONF_SRCDIR     = $(UTSBASE)/common/io/comstar/lu/stmf_sbd

#
#       Include common rules.
#
include ../Makefile.$(ARCHDIR)

#
#       Define targets
#
ALL_TARGET      = $(BINARY) $(SRC_CONFILE)
LINT_TARGET     = $(MODULE).lint
INSTALL_TARGET  = $(BINARY) $(ROOTMODULE) $(ROOT_CONFFILE)

#
#       Overrides and depends_on
#
```



```
MODSTUBS_DIR     = $(OBJS_DIR)
LDFLAGS         += -dy -Ndrv/stmf -Nfs/zfs
LINTTAGS        += -L$(LINT_LIB_DIR) -lstmf -lzfs

INC_PATH        += -I$(UTSBASE)/common/fs/zfs

C99MODE=        -xc99=%all
C99LMODE=       -Xc99=%all

#
# For now, disable these lint checks; maintainers should endeavor
# to investigate and remove these for maximum lint coverage.
#
LINTTAGS        += -erroff=E_BAD_PTR_CAST_ALIGN

CERRWARN        += -_gcc=-Wno-switch
CERRWARN        += -_gcc=-Wno-parentheses
CERRWARN        += -_gcc=-Wno-unused-label
CERRWARN        += -_gcc=-Wno-uninitialized

#
#       Default build targets.
#
.KEEP_STATE:

def:            $(DEF_DEPS)

all:            $(ALL_DEPS)

clean:          $(CLEAN_DEPS)

clobber:        $(CLOBBER_DEPS)

lint:           $(LINT_DEPS)

modlintlib:     $(MODLINTLIB_DEPS)

clean.lint:     $(CLEAN_LINT_DEPS)

install:        $(INSTALL_DEPS)

#
#       Include common targets.
#
include ../Makefile.targ
```