```
**********************************************************
   77503 Thu Apr 25 16:14:55 2013
new/usr/src/cmd/mdb/common/modules/zfs/zfs.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_

 511 #define CHAIN_END 0xffff
 512 /*
 513  * ::zap_leaf [-v]
 514  *
 515  * Print a zap_leaf_phys_t, assumed to be 16k
 516  */
 517 /* ARGSUSED */
 518 static int
 519 zap_leaf(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
 520 {
 521         char buf[16*1024];
 522         int verbose = B_FALSE;
 523         int four = B_FALSE;
 524         zap_leaf_t l = { 0 };
 524         zap_leaf_t l;
 525         zap_leaf_phys_t *zlp = (void *)buf;
 526         int i;

 528         if (mdb_getopts(argc, argv,
 529             'v', MDB_OPT_SETBITS, TRUE, &verbose,
 530             '4', MDB_OPT_SETBITS, TRUE, &four,
 531             NULL) != argc)
 532                 return (DCMD_USAGE);

 534         l.l_phys = zlp;
 535         l.l_bs = 14; /* assume 16k blocks */
 536         if (four)
 537                 l.l_bs = 12;

 539         if (!(flags & DCMD_ADDRSPEC)) {
 540                 return (DCMD_USAGE);
 541         }

 543         if (mdb_vread(buf, sizeof (buf), addr) == -1) {
 544                 mdb_warn("failed to read zap_leaf_phys_t at %p", addr);
 545                 return (DCMD_ERR);
 546         }

 548         if (zlp->l_hdr.lh_block_type != ZBT_LEAF ||
 549             zlp->l_hdr.lh_magic != ZAP_LEAF_MAGIC) {
 550                 mdb_warn("This does not appear to be a zap_leaf_phys_t");
 551                 return (DCMD_ERR);
 552         }

 554         mdb_printf("zap_leaf_phys_t at %p:\n", addr);
 555         mdb_printf("    lh_prefix_len = %u\n", zlp->l_hdr.lh_prefix_len);
 556         mdb_printf("    lh_prefix = %llx\n", zlp->l_hdr.lh_prefix);
 557         mdb_printf("    lh_nentries = %u\n", zlp->l_hdr.lh_nentries);
 558         mdb_printf("    lh_nfree = %u\n", zlp->l_hdr.lh_nfree,
 559             zlp->l_hdr.lh_nfree * 100 / (ZAP_LEAF_NUMCHUNKS(&l)));
 560         mdb_printf("    lh_freelist = %u\n", zlp->l_hdr.lh_freelist);
 561         mdb_printf("    lh_flags = %x (%s)\n", zlp->l_hdr.lh_flags,
 562             zlp->l_hdr.lh_flags & ZLF_ENTRIES_CDSORTED ?
 563             "ENTRIES_CDSORTED" : "");

 565         if (verbose) {
 566                 mdb_printf(" hash table:\n");
```



```
 567                 for (i = 0; i < ZAP_LEAF_HASH_NUMENTRIES(&l); i++) {
 568                         if (zlp->l_hash[i] != CHAIN_END)
 569                                 mdb_printf("    %u: %u\n", i, zlp->l_hash[i]);
 570                 }
 571         }

 573         mdb_printf(" chunks:\n");
 574         for (i = 0; i < ZAP_LEAF_NUMCHUNKS(&l); i++) {
 575                 /* LINTED: alignment */
 576                 zap_leaf_chunk_t *zlc = &ZAP_LEAF_CHUNK(&l, i);
 577                 switch (zlc->l_entry.le_type) {
 578                 case ZAP_CHUNK_FREE:
 579                         if (verbose) {
 580                                 mdb_printf("    %u: free; lf_next = %u\n",
 581                                     i, zlc->l_free.lf_next);
 582                         }
 583                         break;
 584                 case ZAP_CHUNK_ENTRY:
 585                         mdb_printf("    %u: entry\n", i);
 586                         if (verbose) {
 587                                 mdb_printf("        le_next = %u\n",
 588                                     zlc->l_entry.le_next);
 589                         }
 590                         mdb_printf("        le_name_chunk = %u\n",
 591                             zlc->l_entry.le_name_chunk);
 592                         mdb_printf("        le_name_numints = %u\n",
 593                             zlc->l_entry.le_name_numints);
 594                         mdb_printf("        le_value_chunk = %u\n",
 595                             zlc->l_entry.le_value_chunk);
 596                         mdb_printf("        le_value_intlen = %u\n",
 597                             zlc->l_entry.le_value_intlen);
 598                         mdb_printf("        le_value_numints = %u\n",
 599                             zlc->l_entry.le_value_numints);
 600                         mdb_printf("        le_cd = %u\n",
 601                             zlc->l_entry.le_cd);
 602                         mdb_printf("        le_hash = %llx\n",
 603                             zlc->l_entry.le_hash);
 604                         break;
 605                 case ZAP_CHUNK_ARRAY:
 606                         mdb_printf("    %u: array", i);
 607                         if (strisprint((char *)zlc->l_array.la_array))
 608                                 mdb_printf(" \"%s\"", zlc->l_array.la_array);
 609                         mdb_printf("\n");
 610                         if (verbose) {
 611                                 int j;
 612                                 mdb_printf("        ");
 613                                 for (j = 0; j < ZAP_LEAF_ARRAY_BYTES; j++) {
 614                                         mdb_printf("%02x ",
 615                                             zlc->l_array.la_array[j]);
 616                                 }
 617                                 mdb_printf("\n");
 618                         }
 619                         if (zlc->l_array.la_next != CHAIN_END) {
 620                                 mdb_printf("        lf_next = %u\n",
 621                                     zlc->l_array.la_next);
 622                         }
 623                         break;
 624                 default:
 625                         mdb_printf("    %u: undefined type %u\n",
 626                             zlc->l_entry.le_type);
 627                 }
 628         }

 630         return (DCMD_OK);
 631 }
______unchanged_portion_omitted_
```

```
1428 struct mdb_dsl_dir_phys;
1429 typedef struct mdb_dsl_dir_dbuf {
1430         uint8_t dddb_pad[offsetof(dmu_buf_t, db_data)];
1431         uintptr_t dddb_data;
1432 } mdb_dsl_dir_dbuf_t;

1434 #endif /* ! codereview */
1435 typedef struct mdb_dsl_dir {
1436         union {
1437                 dmu_buf_t *dd_dmu_db;
1438                 mdb_dsl_dir_dbuf_t *dd_db;
1439         } dd_db_u;
1428         uintptr_t dd_phys;
1440         int64_t dd_space_towrite[TXG_SIZE];
1441 } mdb_dsl_dir_t;
```

*unchanged_portion_omitted_*

****
6419 Thu Apr 25 16:14:56 2013
new/usr/src/lib/libzfs/common/libzfs_impl.h
3752 want more verifiable dbuf user eviction
Submitted by: Justin Gibbs <justing@spectralogic.com>
Submitted by: Will Andrews <willa@spectralogic.com>
****

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

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _LIBZFS_IMPL_H
#define _LIBZFS_IMPL_H

#include <sys/dmu.h>
#include <sys/fs/zfs.h>
#include <sys/zfs_ioctl.h>
#include <sys/spa.h>
#include <sys/nvpair.h>

#include <libuutil.h>
#include <libzfs.h>
#include <libshare.h>
#include <libzfs_core.h>

#include <fm/libtopo.h>

#ifdef  __cplusplus
extern "C" {
#endif

#ifdef  VERIFY
#undef  VERIFY
#endif
#define VERIFY  verify

typedef struct libzfs_fru {
        char *zf_device;
        char *zf_fru;
        struct libzfs_fru *zf_chain;
        struct libzfs_fru *zf_next;
} libzfs_fru_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   75656 Thu Apr 25 16:14:56 2013
new/usr/src/uts/common/fs/zfs/dbuf.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_

 204 static arc_evict_func_t dbuf_do_evict;

 206 static void
 207 dbuf_verify_user(dmu_buf_impl_t *db, boolean_t evicting)
 208 {
 209 #ifdef ZFS_DEBUG

 211         if (db->db_level != 0)
 212                 ASSERT(db->db_user == NULL);

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

 217         /* Clients must resolve a dbuf before attaching user data. */
 218         ASSERT(db->db.db_data != NULL && db->db_state == DB_CACHED);
 219         /*
 220          * We can't check the hold count here, because they are modified
 221          * independently of the dbuf mutex.  But it would be nice to ensure
 222          * that the user has the appropriate number.
 223          */
 224 #endif
 225 }

 227 /*
 228  * Evict the dbuf's user, either immediately, or use a provided queue.
 229  *
 230  * Call dmu_buf_process_user_evicts or dmu_buf_destroy_user_evict_list
 231  * on the list when finished generating it.
 232  *
 233  * NOTE: If db->db_immediate_evict is FALSE, evict_list_p must be provided.
 234  * NOTE: See dmu_buf_user_t about how this process works.
 235  */
 236 static void
 237 dbuf_evict_user(dmu_buf_impl_t *db, list_t *evict_list_p)
 207 dbuf_evict_user(dmu_buf_impl_t *db)
 238 {
 239         ASSERT(MUTEX_HELD(&db->db_mtx));
 240         ASSERT(evict_list_p != NULL);
 241         dbuf_verify_user(db, /*evicting*/B_TRUE);
 242 #endif /* ! codereview */

 244         if (db->db_user == NULL)
 210         if (db->db_level != 0 || db->db_evict_func == NULL)
 245                 return;

 247         ASSERT(!list_link_active(&db->db_user->evict_queue_link));
 248         list_insert_head(evict_list_p, db->db_user);
 249         db->db_user = NULL;
 250 }

 252 /*
 253  * Replace the current user of the dbuf.  Requires that the caller knows who
 254  * the old user is.  Returns the old user, which may not necessarily be
 255  * the same old_user provided by the caller.
 256  */
 257 dmu_buf_user_t *
 258 dmu_buf_replace_user(dmu_buf_t *db_fake, dmu_buf_user_t *old_user,
```



```
 259     dmu_buf_user_t *new_user)
 260 {
 261         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

 263         mutex_enter(&db->db_mtx);
 264         dbuf_verify_user(db, /*evicting*/B_FALSE);
 265         if (db->db_user == old_user)
 266                 db->db_user = new_user;
 267         else
 268                 old_user = db->db_user;
 269         dbuf_verify_user(db, /*evicting*/B_FALSE);
 270         mutex_exit(&db->db_mtx);

 272         return (old_user);
 273 }

 275 /*
 276  * Set the user eviction data for the DMU beturns NULL on success,
 277  * or the existing user if another user currently owns the buffer.
 278  */
 279 dmu_buf_user_t *
 280 dmu_buf_set_user(dmu_buf_t *db_fake, dmu_buf_user_t *user)
 281 {
 282         return (dmu_buf_replace_user(db_fake, NULL, user));
 283 }

 285 dmu_buf_user_t *
 286 dmu_buf_set_user_ie(dmu_buf_t *db_fake, dmu_buf_user_t *user)
 287 {
 288         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

 290         db->db_immediate_evict = TRUE;
 291         return (dmu_buf_set_user(db_fake, user));
 292 }

 294 /*
 295  * Remove the user eviction data for the DMU buffer.
 296  */
 297 dmu_buf_user_t *
 298 dmu_buf_remove_user(dmu_buf_t *db_fake, dmu_buf_user_t *user)
 299 {
 300         return (dmu_buf_replace_user(db_fake, user, NULL));
 301 }

 303 /*
 304  * Returns the db_user set with dmu_buf_update_user(), or NULL if not set.
 305  */
 306 dmu_buf_user_t *
 307 dmu_buf_get_user(dmu_buf_t *db_fake)
 308 {
 309         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

 311         dbuf_verify_user(db, /*evicting*/B_FALSE);
 312         return (db->db_user);
 313 }

 315 static void
 316 dbuf_clear_data(dmu_buf_impl_t *db, list_t *evict_list_p)
 317 {
 318         ASSERT(MUTEX_HELD(&db->db_mtx));
 319         ASSERT(db->db_buf == NULL || !arc_has_callback(db->db_buf));
 320         dbuf_evict_user(db, evict_list_p);
 321         db->db_buf = NULL;
 322         db->db.db_data = NULL;
 323         if (db->db_state != DB_NOFILL)
 324                 db->db_state = DB_UNCACHED;
```

```
325 }

327 static void
328 dbuf_set_data(dmu_buf_impl_t *db, arc_buf_t *buf)
329 {
330         ASSERT(MUTEX_HELD(&db->db_mtx));
331         ASSERT(db->db_buf == NULL || !arc_has_callback(db->db_buf));
332         ASSERT(buf != NULL);

334         db->db_buf = buf;
335         ASSERT(buf->b_data != NULL);
336         db->db.db_data = buf->b_data;
337         if (!arc_released(buf))
338                 arc_set_callback(buf, dbuf_do_evict, db);
213         if (db->db_user_data_ptr_ptr)
214                 *db->db_user_data_ptr_ptr = db->db.db_data;
215         db->db_evict_func(&db->db, db->db_user_ptr);
216         db->db_user_ptr = NULL;
217         db->db_user_data_ptr_ptr = NULL;
218         db->db_evict_func = NULL;
339 }
```

*unchanged_portion_omitted_*

```
357 void
358 dbuf_evict(dmu_buf_impl_t *db, list_t *evict_list_p)
238 dbuf_evict(dmu_buf_impl_t *db)
359 {
360         ASSERT(MUTEX_HELD(&db->db_mtx));
361         ASSERT(db->db_buf == NULL);
362         ASSERT(db->db_data_pending == NULL);

364         dbuf_clear(db, evict_list_p);
244         dbuf_clear(db);
365         dbuf_destroy(db);
366 }
```

*unchanged_portion_omitted_*

```
525 #endif

407 static void
408 dbuf_update_data(dmu_buf_impl_t *db)
409 {
410         ASSERT(MUTEX_HELD(&db->db_mtx));
411         if (db->db_level == 0 && db->db_user_data_ptr_ptr) {
412                 ASSERT(!refcount_is_zero(&db->db_holds));
413                 *db->db_user_data_ptr_ptr = db->db.db_data;
414         }
415 }

417 static void
418 dbuf_set_data(dmu_buf_impl_t *db, arc_buf_t *buf)
419 {
420         ASSERT(MUTEX_HELD(&db->db_mtx));
421         ASSERT(db->db_buf == NULL || !arc_has_callback(db->db_buf));
422         db->db_buf = buf;
423         if (buf != NULL) {
424                 ASSERT(buf->b_data != NULL);
425                 db->db.db_data = buf->b_data;
426                 if (!arc_released(buf))
427                         arc_set_callback(buf, dbuf_do_evict, db);
428                 dbuf_update_data(db);
429         } else {
430                 dbuf_evict_user(db);
431                 db->db.db_data = NULL;
432                 if (db->db_state != DB_NOFILL)
433                         db->db_state = DB_UNCACHED;
434         }
```



```
435 }

527 /*
528  * Loan out an arc_buf for read.  Return the loaned arc_buf.
529  */
530 arc_buf_t *
531 dbuf_loan_arcbuf(dmu_buf_impl_t *db)
532 {
533         arc_buf_t *abuf;
534         list_t evict_list;

536         dmu_buf_create_user_evict_list(&evict_list);
537 #endif /* ! codereview */

539         mutex_enter(&db->db_mtx);
540         if (arc_released(db->db_buf) || refcount_count(&db->db_holds) > 1) {
541                 int blksz = db->db.db_size;
542                 spa_t *spa;

544                 mutex_exit(&db->db_mtx);
545                 DB_GET_SPA(&spa, db);
546                 abuf = arc_loan_buf(spa, blksz);
547                 bcopy(db->db.db_data, abuf->b_data, blksz);
548         } else {
549                 abuf = db->db_buf;
550                 arc_loan_inuse_buf(abuf, db);
551                 dbuf_clear_data(db, &evict_list);
444                 dbuf_set_data(db, NULL);
552                 mutex_exit(&db->db_mtx);
553         }
554         dmu_buf_destroy_user_evict_list(&evict_list);
555 #endif /* ! codereview */
556         return (abuf);
557 }

559 uint64_t
560 dbuf_whichblock(dnode_t *dn, uint64_t offset)
561 {
562         if (dn->dn_datablkshift) {
563                 return (offset >> dn->dn_datablkshift);
564         } else {
565                 ASSERT3U(offset, <, dn->dn_datablksz);
566                 return (0);
567         }
568 }

570 static void
571 dbuf_read_done(zio_t *zio, arc_buf_t *buf, void *vdb)
572 {
573         dmu_buf_impl_t *db = vdb;

575         mutex_enter(&db->db_mtx);
576         ASSERT3U(db->db_state, ==, DB_READ);
577         /*
578          * All reads are synchronous, so we must have a hold on the dbuf
579          */
580         ASSERT(refcount_count(&db->db_holds) > 0);
581         ASSERT(db->db_buf == NULL);
582         ASSERT(db->db.db_data == NULL);
583         if (db->db_level == 0 && db->db_freed_in_flight) {
584                 /* we were freed in flight; disregard any error */
585                 arc_release(buf, db);
586                 bzero(buf->b_data, db->db.db_size);
587                 arc_buf_freeze(buf);
588                 db->db_freed_in_flight = FALSE;
589                 dbuf_set_data(db, buf);
```

```
590                 db->db_state = DB_CACHED;
591         } else if (zio == NULL || zio->io_error == 0) {
592                 dbuf_set_data(db, buf);
593                 db->db_state = DB_CACHED;
594         } else {
595                 ASSERT(db->db_blkid != DMU_BONUS_BLKID);
596                 ASSERT3P(db->db_buf, ==, NULL);
597                 VERIFY(arc_buf_remove_ref(buf, db));
598                 db->db_state = DB_UNCACHED;
599         }
600         cv_broadcast(&db->db_changed);
601         dbuf_rele_and_unlock(db, NULL);
602 }

604 static void
605 dbuf_read_impl(dmu_buf_impl_t *db, zio_t *zio, uint32_t *flags)
606 {
607         dnode_t *dn;
608         spa_t *spa;
609         zbookmark_t zb;
610         uint32_t aflags = ARC_NOWAIT;

612         DB_DNODE_ENTER(db);
613         dn = DB_DNODE(db);
614         ASSERT(!refcount_is_zero(&db->db_holds));
615         /* We need the struct_rwlock to prevent db_blkptr from changing. */
616         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
617         ASSERT(MUTEX_HELD(&db->db_mtx));
618         ASSERT(db->db_state == DB_UNCACHED);
619         ASSERT(db->db_buf == NULL);

621         if (db->db_blkid == DMU_BONUS_BLKID) {
622                 int bonuslen = MIN(dn->dn_bonuslen, dn->dn_phys->dn_bonuslen);

624                 ASSERT3U(bonuslen, <=, db->db.db_size);
625                 db->db.db_data = zio_buf_alloc(DN_MAX_BONUSLEN);
626                 arc_space_consume(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
627                 if (bonuslen < DN_MAX_BONUSLEN)
628                         bzero(db->db.db_data, DN_MAX_BONUSLEN);
629                 if (bonuslen)
630                         bcopy(DN_BONUS(dn->dn_phys), db->db.db_data, bonuslen);
631                 DB_DNODE_EXIT(db);
447                 dbuf_update_data(db);
632                 db->db_state = DB_CACHED;
633                 mutex_exit(&db->db_mtx);
634                 return;
635         }

637         /*
638          * Recheck BP_IS_HOLE() after dnode_block_freed() in case dnode_sync()
639          * processes the delete record and clears the bp while we are waiting
640          * for the dn_mtx (resulting in a "no" from block_freed).
641          */
642         if (db->db_blkptr == NULL || BP_IS_HOLE(db->db_blkptr) ||
643             (db->db_level == 0 && (dnode_block_freed(dn, db->db_blkid) ||
644             BP_IS_HOLE(db->db_blkptr)))) {
645                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);

647                 dbuf_set_data(db, arc_buf_alloc(dn->dn_objset->os_spa,
648                     db->db.db_size, db, type));
649                 DB_DNODE_EXIT(db);
650                 bzero(db->db.db_data, db->db.db_size);
651                 db->db_state = DB_CACHED;
652                 *flags |= DB_RF_CACHED;
653                 mutex_exit(&db->db_mtx);
654                 return;
```



```
655         }

657         spa = dn->dn_objset->os_spa;
658         DB_DNODE_EXIT(db);

660         db->db_state = DB_READ;
661         mutex_exit(&db->db_mtx);

663         if (DBUF_IS_L2CACHEABLE(db))
664                 aflags |= ARC_L2CACHE;

666         SET_BOOKMARK(&zb, db->db_objset->os_dsl_dataset ?
667             db->db_objset->os_dsl_dataset->ds_object : DMU_META_OBJSET,
668             db->db.db_object, db->db_level, db->db_blkid);

670         dbuf_add_ref(db, NULL);

672         (void) arc_read(zio, spa, db->db_blkptr,
673             dbuf_read_done, db, ZIO_PRIORITY_SYNC_READ,
674             (*flags & DB_RF_CANFAIL) ? ZIO_FLAG_CANFAIL : ZIO_FLAG_MUSTSUCCEED,
675             &aflags, &zb);
676         if (aflags & ARC_CACHED)
677                 *flags |= DB_RF_CACHED;
678 }
```

*unchanged_portion_omitted_*

```
761 static void
762 dbuf_noread(dmu_buf_impl_t *db)
763 {
764         list_t evict_list;

766 #endif /* ! codereview */
767         ASSERT(!refcount_is_zero(&db->db_holds));
768         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
769         dmu_buf_create_user_evict_list(&evict_list);

771 #endif /* ! codereview */
772         mutex_enter(&db->db_mtx);
773         while (db->db_state == DB_READ || db->db_state == DB_FILL)
774                 cv_wait(&db->db_changed, &db->db_mtx);
775         if (db->db_state == DB_UNCACHED) {
776                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
777                 spa_t *spa;

779                 ASSERT(db->db_buf == NULL);
780                 ASSERT(db->db.db_data == NULL);
781                 DB_GET_SPA(&spa, db);
782                 dbuf_set_data(db, arc_buf_alloc(spa, db->db.db_size, db, type));
783                 db->db_state = DB_FILL;
784         } else if (db->db_state == DB_NOFILL) {
785                 dbuf_clear_data(db, &evict_list);
580                 dbuf_set_data(db, NULL);
786         } else {
787                 ASSERT3U(db->db_state, ==, DB_CACHED);
788         }
789         mutex_exit(&db->db_mtx);
790         dmu_buf_destroy_user_evict_list(&evict_list);
791 #endif /* ! codereview */
792 }

794 /*
795  * This is our just-in-time copy function.  It makes a copy of
796  * buffers, that have been modified in a previous transaction
797  * group, before we modify them in the current active group.
798  *
799  * This function is used in two places: when we are dirtying a
```

```
 * buffer for the first time in a txg, and when we are freeing
 * a range in a dnode that includes this buffer.
 *
 * Note that when we are called from dbuf_free_range() we do
 * not put a hold on the buffer, we just traverse the active
 * dbuf list for the dnode.
 */
static void
dbuf_fix_old_data(dmu_buf_impl_t *db, uint64_t txg, list_t *evict_list_p)
dbuf_fix_old_data(dmu_buf_impl_t *db, uint64_t txg)
{
        dbuf_dirty_record_t *dr = db->db_last_dirty;

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

        if (dr == NULL ||
            (dr->dt.dl.dr_data !=
            ((db->db_blkid  == DMU_BONUS_BLKID) ? db->db.db_data : db->db_buf)))
                return;

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

                DB_GET_SPA(&spa, db);
                dr->dt.dl.dr_data = arc_buf_alloc(spa, size, db, type);
                bcopy(db->db.db_data, dr->dt.dl.dr_data->b_data, size);
        } else {
                dbuf_clear_data(db, evict_list_p);
                dbuf_set_data(db, NULL);
        }
}
_____unchanged_portion_omitted_

/*
 * Evict (if its unreferenced) or clear (if its referenced) any level-0
 * data blocks in the free range, so that any future readers will find
 * empty blocks.  Also, if we happen accross any level-1 dbufs in the
 * range that have not already been marked dirty, mark them dirty so
 * they stay in memory.
 */
void
dbuf_free_range(dnode_t *dn, uint64_t start, uint64_t end, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db, *db_next;
        uint64_t txg = tx->tx_txg;
        int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
        uint64_t first_l1 = start >> epbs;
        uint64_t last_l1 = end >> epbs;
```



```
        list_t evict_list;

        dmu_buf_create_user_evict_list(&evict_list);
#endif /* ! codereview */

        if (end > dn->dn_maxblkid && (end != DMU_SPILL_BLKID)) {
                end = dn->dn_maxblkid;
                last_l1 = end >> epbs;
        }
        dprintf_dnode(dn, "start=%llu end=%llu\n", start, end);
        mutex_enter(&dn->dn_dbufs_mtx);
        for (db = list_head(&dn->dn_dbufs); db; db = db_next) {
                db_next = list_next(&dn->dn_dbufs, db);
                ASSERT(db->db_blkid != DMU_BONUS_BLKID);

                if (db->db_level == 1 &&
                    db->db_blkid >= first_l1 && db->db_blkid <= last_l1) {
                        mutex_enter(&db->db_mtx);
                        if (db->db_last_dirty &&
                            db->db_last_dirty->dr_txg < txg) {
                                dbuf_add_ref(db, FTAG);
                                mutex_exit(&db->db_mtx);
                                dbuf_will_dirty(db, tx);
                                dbuf_rele(db, FTAG);
                        } else {
                                mutex_exit(&db->db_mtx);
                        }
                }

                if (db->db_level != 0)
                        continue;
                dprintf_dbuf(db, "found buf %s\n", "");
                if (db->db_blkid < start || db->db_blkid > end)
                        continue;

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

                if (db->db_state == DB_UNCACHED ||
                    db->db_state == DB_NOFILL ||
                    db->db_state == DB_EVICTING) {
                        ASSERT(db->db.db_data == NULL);
                        mutex_exit(&db->db_mtx);
                        continue;
                }
                if (db->db_state == DB_READ || db->db_state == DB_FILL) {
                        /* will be handled in dbuf_read_done or dbuf_rele */
                        db->db_freed_in_flight = TRUE;
                        mutex_exit(&db->db_mtx);
                        continue;
                }
                if (refcount_count(&db->db_holds) == 0) {
                        ASSERT(db->db_buf);
                        dbuf_clear(db, &evict_list);
                        dbuf_clear(db);
                        continue;
                }
                /* The dbuf is referenced */

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

```
                        if (dr->dr_txg == txg) {
                                /*
                                 * This buffer is "in-use", re-adjust the file
                                 * size to reflect that this buffer may
                                 * contain new data when we sync.
                                 */
                                if (db->db_blkid != DMU_SPILL_BLKID &&
                                    db->db_blkid > dn->dn_maxblkid)
                                        dn->dn_maxblkid = db->db_blkid;
                                dbuf_unoverride(dr);
                        } else {
                                /*
                                 * This dbuf is not dirty in the open context.
                                 * Either uncache it (if its not referenced in
                                 * the open context) or reset its contents to
                                 * empty.
                                 */
                                dbuf_fix_old_data(db, txg, &evict_list);
                                dbuf_fix_old_data(db, txg);
                        }
                }
                /* clear the contents if its cached */
                if (db->db_state == DB_CACHED) {
                        ASSERT(db->db.db_data != NULL);
                        arc_release(db->db_buf, db);
                        bzero(db->db.db_data, db->db.db_size);
                        arc_buf_freeze(db->db_buf);
                }

                mutex_exit(&db->db_mtx);
                dmu_buf_process_user_evicts(&evict_list);
#endif /* ! codereview */
        }
        mutex_exit(&dn->dn_dbufs_mtx);
        dmu_buf_destroy_user_evict_list(&evict_list);
#endif /* ! codereview */
}

static int
dbuf_block_freeable(dmu_buf_impl_t *db)
{
        dsl_dataset_t *ds = db->db_objset->os_dsl_dataset;
        uint64_t birth_txg = 0;

        /*
         * We don't need any locking to protect db_blkptr:
         * If it's syncing, then db_last_dirty will be set
         * so we'll ignore db_blkptr.
         */
        ASSERT(MUTEX_HELD(&db->db_mtx));
        if (db->db_last_dirty)
                birth_txg = db->db_last_dirty->dr_txg;
        else if (db->db_blkptr)
                birth_txg = db->db_blkptr->blk_birth;

        /*
         * If we don't exist or are in a snapshot, we can't be freed.
         * Don't pass the bp to dsl_dataset_block_freeable() since we
         * are holding the db_mtx lock and might deadlock if we are
         * prefetching a dedup-ed block.
         */
        if (birth_txg)
                return (ds == NULL ||
                    dsl_dataset_block_freeable(ds, NULL, birth_txg));
        else
                return (FALSE);
```



```
}

void
dbuf_new_size(dmu_buf_impl_t *db, int size, dmu_tx_t *tx)
{
        arc_buf_t *buf, *obuf;
        int osize = db->db.db_size;
        arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
        dnode_t *dn;

        ASSERT(db->db_blkid != DMU_BONUS_BLKID);

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

        /* XXX does *this* func really need the lock? */
        ASSERT(RW_WRITE_HELD(&dn->dn_struct_rwlock));

        /*
         * This call to dbuf_will_dirty() with the dn_struct_rwlock held
         * is OK, because there can be no other references to the db
         * when we are changing its size, so no concurrent DB_FILL can
         * be happening.
         */
        /*
         * XXX we should be doing a dbuf_read, checking the return
         * value and returning that up to our callers
         */
        dbuf_will_dirty(db, tx);

        /* create the data buffer for the new block */
        buf = arc_buf_alloc(dn->dn_objset->os_spa, size, db, type);

        /* copy old block data to the new block */
        obuf = db->db_buf;
        bcopy(obuf->b_data, buf->b_data, MIN(osize, size));
        /* zero the remainder */
        if (size > osize)
                bzero((uint8_t *)buf->b_data + osize, size - osize);

        mutex_enter(&db->db_mtx);
        dbuf_set_data(db, buf);
        VERIFY(arc_buf_remove_ref(obuf, db));
        db->db.db_size = size;

        if (db->db_level == 0) {
                ASSERT3U(db->db_last_dirty->dr_txg, ==, tx->tx_txg);
                db->db_last_dirty->dt.dl.dr_data = buf;
        }
        mutex_exit(&db->db_mtx);

        dnode_willuse_space(dn, size-osize, tx);
        DB_DNODE_EXIT(db);
}

void
dbuf_release_bp(dmu_buf_impl_t *db)
{
        objset_t *os;

        DB_GET_OBJSET(&os, db);
        ASSERT(dsl_pool_sync_context(dmu_objset_pool(os)));
        ASSERT(arc_released(os->os_phys_buf) ||
            list_link_active(&os->os_dsl_dataset->ds_synced_link));
        ASSERT(db->db_parent == NULL || arc_released(db->db_parent->db_buf));
```

```
1097         (void) arc_release(db->db_buf, db);
1098 }

1100 dbuf_dirty_record_t *
1101 dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1102 {
1103         dnode_t *dn;
1104         objset_t *os;
1105         dbuf_dirty_record_t **drp, *dr;
1106         int drop_struct_lock = FALSE;
1107         boolean_t do_free_accounting = B_FALSE;
1108         int txgoff = tx->tx_txg & TXG_MASK;
1109         list_t evict_list;

1111         dmu_buf_create_user_evict_list(&evict_list);
1112 #endif /* ! codereview */

1114         ASSERT(tx->tx_txg != 0);
1115         ASSERT(!refcount_is_zero(&db->db_holds));
1116         DMU_TX_DIRTY_BUF(tx, db);

1118         DB_DNODE_ENTER(db);
1119         dn = DB_DNODE(db);
1120         /*
1121          * Shouldn't dirty a regular buffer in syncing context.  Private
1122          * objects may be dirtied in syncing context, but only if they
1123          * were already pre-dirtied in open context.
1124          */
1125         ASSERT(!dmu_tx_is_syncing(tx) ||
1126             BP_IS_HOLE(dn->dn_objset->os_rootbp) ||
1127             DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1128             dn->dn_objset->os_dsl_dataset == NULL);
1129         /*
1130          * We make this assert for private objects as well, but after we
1131          * check if we're already dirty.  They are allowed to re-dirty
1132          * in syncing context.
1133          */
1134         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1135             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1136             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));

1138         mutex_enter(&db->db_mtx);
1139         /*
1140          * XXX make this true for indirects too?  The problem is that
1141          * transactions created with dmu_tx_create_assigned() from
1142          * syncing context don't bother holding ahead.
1143          */
1144         ASSERT(db->db_level != 0 ||
1145             db->db_state == DB_CACHED || db->db_state == DB_FILL ||
1146             db->db_state == DB_NOFILL);

1148         mutex_enter(&dn->dn_mtx);
1149         /*
1150          * Don't set dirtyctx to SYNC if we're just modifying this as we
1151          * initialize the objset.
1152          */
1153         if (dn->dn_dirtyctx == DN_UNDIRTIED &&
1154             !BP_IS_HOLE(dn->dn_objset->os_rootbp)) {
1155                 dn->dn_dirtyctx =
1156                     (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN);
1157                 ASSERT(dn->dn_dirtyctx_firstset == NULL);
1158                 dn->dn_dirtyctx_firstset = kmem_alloc(1, KM_SLEEP);
1159         }
1160         mutex_exit(&dn->dn_mtx);

1162         if (db->db_blkid == DMU_SPILL_BLKID)
```



```
1163                 dn->dn_have_spill = B_TRUE;

1165         /*
1166          * If this buffer is already dirty, we're done.
1167          */
1168         drp = &db->db_last_dirty;
1169         ASSERT(*drp == NULL || (*drp)->dr_txg <= tx->tx_txg ||
1170             db->db.db_object == DMU_META_DNODE_OBJECT);
1171         while ((dr = *drp) != NULL && dr->dr_txg > tx->tx_txg)
1172                 drp = &dr->dr_next;
1173         if (dr && dr->dr_txg == tx->tx_txg) {
1174                 DB_DNODE_EXIT(db);

1176                 if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID) {
1177                         /*
1178                          * If this buffer has already been written out,
1179                          * we now need to reset its state.
1180                          */
1181                         dbuf_unoverride(dr);
1182                         if (db->db.db_object != DMU_META_DNODE_OBJECT &&
1183                             db->db_state != DB_NOFILL)
1184                                 arc_buf_thaw(db->db_buf);
1185                 }
1186                 mutex_exit(&db->db_mtx);
1187                 dmu_buf_destroy_user_evict_list(&evict_list);
1188 #endif /* ! codereview */
1189                 return (dr);
1190         }

1192         /*
1193          * Only valid if not already dirty.
1194          */
1195         ASSERT(dn->dn_object == 0 ||
1196             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1197             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));

1199         ASSERT3U(dn->dn_nlevels, >, db->db_level);
1200         ASSERT((dn->dn_phys->dn_nlevels == 0 && db->db_level == 0) ||
1201             dn->dn_phys->dn_nlevels > db->db_level ||
1202             dn->dn_next_nlevels[txgoff] > db->db_level ||
1203             dn->dn_next_nlevels[(tx->tx_txg-1) & TXG_MASK] > db->db_level ||
1204             dn->dn_next_nlevels[(tx->tx_txg-2) & TXG_MASK] > db->db_level);

1206         /*
1207          * We should only be dirtying in syncing context if it's the
1208          * mos or we're initializing the os or it's a special object.
1209          * However, we are allowed to dirty in syncing context provided
1210          * we already dirtied it in open context.  Hence we must make
1211          * this assertion only if we're not already dirty.
1212          */
1213         os = dn->dn_objset;
1214         ASSERT(!dmu_tx_is_syncing(tx) || DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1215             os->os_dsl_dataset == NULL || BP_IS_HOLE(os->os_rootbp));
1216         ASSERT(db->db.db_size != 0);

1218         dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);

1220         if (db->db_blkid != DMU_BONUS_BLKID) {
1221                 /*
1222                  * Update the accounting.
1223                  * Note: we delay "free accounting" until after we drop
1224                  * the db_mtx.  This keeps us from grabbing other locks
1225                  * (and possibly deadlocking) in bp_get_dsize() while
1226                  * also holding the db_mtx.
1227                  */
1228                 dnode_willuse_space(dn, db->db.db_size, tx);
```

```
1229                do_free_accounting = dbuf_block_freeable(db);
1230        }
1231
1232        /*
1233         * If this buffer is dirty in an old transaction group we need
1234         * to make a copy of it so that the changes we make in this
1235         * transaction group won't leak out when we sync the older txg.
1236         */
1237        dr = kmem_zalloc(sizeof (dbuf_dirty_record_t), KM_SLEEP);
1238        if (db->db_level == 0) {
1239                void *data_old = db->db_buf;
1240
1241                if (db->db_state != DB_NOFILL) {
1242                        if (db->db_blkid == DMU_BONUS_BLKID) {
1243                                dbuf_fix_old_data(db, tx->tx_txg, &evict_list);
 715                            dbuf_fix_old_data(db, tx->tx_txg);
1244                                data_old = db->db.db_data;
1245                        } else if (db->db.db_object != DMU_META_DNODE_OBJECT) {
1246                                /*
1247                                 * Release the data buffer from the cache so
1248                                 * that we can modify it without impacting
1249                                 * possible other users of this cached data
1250                                 * block.  Note that indirect blocks and
1251                                 * private objects are not released until the
1252                                 * syncing state (since they are only modified
1253                                 * then).
1254                                 */
1255                                arc_release(db->db_buf, db);
1256                                dbuf_fix_old_data(db, tx->tx_txg, &evict_list);
 728                            dbuf_fix_old_data(db, tx->tx_txg);
1257                                data_old = db->db_buf;
1258                        }
1259                        ASSERT(data_old != NULL);
1260                }
1261                dr->dt.dl.dr_data = data_old;
1262        } else {
1263                mutex_init(&dr->dt.di.dr_mtx, NULL, MUTEX_DEFAULT, NULL);
1264                list_create(&dr->dt.di.dr_children,
1265                    sizeof (dbuf_dirty_record_t),
1266                    offsetof(dbuf_dirty_record_t, dr_dirty_node));
1267        }
1268        dr->dr_dbuf = db;
1269        dr->dr_txg = tx->tx_txg;
1270        dr->dr_next = *drp;
1271        *drp = dr;
1272
1273        /*
1274         * We could have been freed_in_flight between the dbuf_noread
1275         * and dbuf_dirty.  We win, as though the dbuf_noread() had
1276         * happened after the free.
1277         */
1278        if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
1279            db->db_blkid != DMU_SPILL_BLKID) {
1280                mutex_enter(&dn->dn_mtx);
1281                dnode_clear_range(dn, db->db_blkid, 1, tx);
1282                mutex_exit(&dn->dn_mtx);
1283                db->db_freed_in_flight = FALSE;
1284        }
1285
1286        /*
1287         * This buffer is now part of this txg
1288         */
1289        dbuf_add_ref(db, (void *)(uintptr_t)tx->tx_txg);
1290        db->db_dirtycnt += 1;
1291        ASSERT3U(db->db_dirtycnt, <=, 3);
```



```
1293        mutex_exit(&db->db_mtx);
1294        dmu_buf_destroy_user_evict_list(&evict_list);
1295 #endif /* ! codereview */
1296
1297        if (db->db_blkid == DMU_BONUS_BLKID ||
1298            db->db_blkid == DMU_SPILL_BLKID) {
1299                mutex_enter(&dn->dn_mtx);
1300                ASSERT(!list_link_active(&dr->dr_dirty_node));
1301                list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1302                mutex_exit(&dn->dn_mtx);
1303                dnode_setdirty(dn, tx);
1304                DB_DNODE_EXIT(db);
1305                return (dr);
1306        } else if (do_free_accounting) {
1307                blkptr_t *bp = db->db_blkptr;
1308                int64_t willfree = (bp && !BP_IS_HOLE(bp)) ?
1309                    bp_get_dsize(os->os_spa, bp) : db->db.db_size;
1310                /*
1311                 * This is only a guess -- if the dbuf is dirty
1312                 * in a previous txg, we don't know how much
1313                 * space it will use on disk yet.  We should
1314                 * really have the struct_rwlock to access
1315                 * db_blkptr, but since this is just a guess,
1316                 * it's OK if we get an odd answer.
1317                 */
1318                ddt_prefetch(os->os_spa, bp);
1319                dnode_willuse_space(dn, -willfree, tx);
1320        }
1321
1322        if (!RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
1323                rw_enter(&dn->dn_struct_rwlock, RW_READER);
1324                drop_struct_lock = TRUE;
1325        }
1326
1327        if (db->db_level == 0) {
1328                dnode_new_blkid(dn, db->db_blkid, tx, drop_struct_lock);
1329                ASSERT(dn->dn_maxblkid >= db->db_blkid);
1330        }
1331
1332        if (db->db_level+1 < dn->dn_nlevels) {
1333                dmu_buf_impl_t *parent = db->db_parent;
1334                dbuf_dirty_record_t *di;
1335                int parent_held = FALSE;
1336
1337                if (db->db_parent == NULL || db->db_parent == dn->dn_dbuf) {
1338                        int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
1339
1340                        parent = dbuf_hold_level(dn, db->db_level+1,
1341                            db->db_blkid >> epbs, FTAG);
1342                        ASSERT(parent != NULL);
1343                        parent_held = TRUE;
1344                }
1345                if (drop_struct_lock)
1346                        rw_exit(&dn->dn_struct_rwlock);
1347                ASSERT3U(db->db_level+1, ==, parent->db_level);
1348                di = dbuf_dirty(parent, tx);
1349                if (parent_held)
1350                        dbuf_rele(parent, FTAG);
1351
1352                mutex_enter(&db->db_mtx);
1353                /*  possible race with dbuf_undirty() */
1354                if (db->db_last_dirty == dr ||
1355                    dn->dn_object == DMU_META_DNODE_OBJECT) {
1356                        mutex_enter(&di->dt.di.dr_mtx);
1357                        ASSERT3U(di->dr_txg, ==, tx->tx_txg);
1358                        ASSERT(!list_link_active(&dr->dr_dirty_node));
```

```
1359                        list_insert_tail(&di->dt.di.dr_children, dr);
1360                        mutex_exit(&di->dt.di.dr_mtx);
1361                        dr->dr_parent = di;
1362                }
1363                mutex_exit(&db->db_mtx);
1364        } else {
1365                ASSERT(db->db_level+1 == dn->dn_nlevels);
1366                ASSERT(db->db_blkid < dn->dn_nblkptr);
1367                ASSERT(db->db_parent == NULL || db->db_parent == dn->dn_dbuf);
1368                mutex_enter(&dn->dn_mtx);
1369                ASSERT(!list_link_active(&dr->dr_dirty_node));
1370                list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1371                mutex_exit(&dn->dn_mtx);
1372                if (drop_struct_lock)
1373                        rw_exit(&dn->dn_struct_rwlock);
1374        }

1376        dnode_setdirty(dn, tx);
1377        DB_DNODE_EXIT(db);
1378        return (dr);
1379 }

1381 /*
1382  * Return TRUE if this evicted the dbuf.
1383  */
1384 static boolean_t
1385 dbuf_undirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1386 {
1387        dnode_t *dn;
1388        uint64_t txg = tx->tx_txg;
1389        dbuf_dirty_record_t *dr, **drp;
1390        list_t evict_list;
1391 #endif /* ! codereview */

1393        ASSERT(txg != 0);
1394        ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1395        ASSERT0(db->db_level);
1396        ASSERT(MUTEX_HELD(&db->db_mtx));

1398        /*
1399         * If this buffer is not dirty, we're done.
1400         */
1401        for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
1402                if (dr->dr_txg <= txg)
1403                        break;
1404        if (dr == NULL || dr->dr_txg < txg)
1405                return (B_FALSE);
1406        ASSERT(dr->dr_txg == txg);
1407        ASSERT(dr->dr_dbuf == db);

1409        dmu_buf_create_user_evict_list(&evict_list);

1411 #endif /* ! codereview */
1412        DB_DNODE_ENTER(db);
1413        dn = DB_DNODE(db);

1415        /*
1416         * Note:  This code will probably work even if there are concurrent
1417         * holders, but it is untested in that scenerio, as the ZPL and
1418         * ztest have additional locking (the range locks) that prevents
1419         * that type of concurrent access.
1420         */
1421        ASSERT3U(refcount_count(&db->db_holds), ==, db->db_dirtycnt);

1423        dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);
```



```
1425        ASSERT(db->db.db_size != 0);

1427        /* XXX would be nice to fix up dn_towrite_space[] */

1429        *drp = dr->dr_next;

1431        /*
1432         * Note that there are three places in dbuf_dirty()
1433         * where this dirty record may be put on a list.
1434         * Make sure to do a list_remove corresponding to
1435         * every one of those list_insert calls.
1436         */
1437        if (dr->dr_parent) {
1438                mutex_enter(&dr->dr_parent->dt.di.dr_mtx);
1439                list_remove(&dr->dr_parent->dt.di.dr_children, dr);
1440                mutex_exit(&dr->dr_parent->dt.di.dr_mtx);
1441        } else if (db->db_blkid == DMU_SPILL_BLKID ||
1442            db->db_level+1 == dn->dn_nlevels) {
1443                ASSERT(db->db_blkptr == NULL || db->db_parent == dn->dn_dbuf);
1444                mutex_enter(&dn->dn_mtx);
1445                list_remove(&dn->dn_dirty_records[txg & TXG_MASK], dr);
1446                mutex_exit(&dn->dn_mtx);
1447        }
1448        DB_DNODE_EXIT(db);

1450        if (db->db_state != DB_NOFILL) {
1451                dbuf_unoverride(dr);

1453                ASSERT(db->db_buf != NULL);
1454                ASSERT(dr->dt.dl.dr_data != NULL);
1455                if (dr->dt.dl.dr_data != db->db_buf)
1456                        VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data, db));
1457        }
1458        kmem_free(dr, sizeof (dbuf_dirty_record_t));

1460        ASSERT(db->db_dirtycnt > 0);
1461        db->db_dirtycnt -= 1;

1463        if (refcount_remove(&db->db_holds, (void *)(uintptr_t)txg) == 0) {
1464                arc_buf_t *buf = db->db_buf;

1466                ASSERT(db->db_state == DB_NOFILL || arc_released(buf));
1467                dbuf_clear_data(db, &evict_list);
 766                dbuf_set_data(db, NULL);
1468                VERIFY(arc_buf_remove_ref(buf, db));
1469                dbuf_evict(db, &evict_list);
1470                dmu_buf_destroy_user_evict_list(&evict_list);
 768                dbuf_evict(db);
1471                return (B_TRUE);
1472        }

1474        dmu_buf_destroy_user_evict_list(&evict_list);
1475 #endif /* ! codereview */
1476        return (B_FALSE);
1477 }

1479 #pragma weak dmu_buf_will_dirty = dbuf_will_dirty
1480 void
1481 dbuf_will_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1482 {
1483        int rf = DB_RF_MUST_SUCCEED | DB_RF_NOPREFETCH;

1485        ASSERT(tx->tx_txg != 0);
1486        ASSERT(!refcount_is_zero(&db->db_holds));

1488        DB_DNODE_ENTER(db);
```

```
1489         if (RW_WRITE_HELD(&DB_DNODE(db)->dn_struct_rwlock))
1490                 rf |= DB_RF_HAVESTRUCT;
1491         DB_DNODE_EXIT(db);
1492         (void) dbuf_read(db, NULL, rf);
1493         (void) dbuf_dirty(db, tx);
1494 }

1496 void
1497 dmu_buf_will_not_fill(dmu_buf_t *db_fake, dmu_tx_t *tx)
1498 {
1499         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

1501         db->db_state = DB_NOFILL;

1503         dmu_buf_will_fill(db_fake, tx);
1504 }

1506 void
1507 dmu_buf_will_fill(dmu_buf_t *db_fake, dmu_tx_t *tx)
1508 {
1509         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

1511         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1512         ASSERT(tx->tx_txg != 0);
1513         ASSERT(db->db_level == 0);
1514         ASSERT(!refcount_is_zero(&db->db_holds));

1516         ASSERT(db->db.db_object != DMU_META_DNODE_OBJECT ||
1517             dmu_tx_private_ok(tx));

1519         dbuf_noread(db);
1520         (void) dbuf_dirty(db, tx);
1521 }

1523 #pragma weak dmu_buf_fill_done = dbuf_fill_done
1524 /* ARGSUSED */
1525 void
1526 dbuf_fill_done(dmu_buf_impl_t *db, dmu_tx_t *tx)
1527 {
1528         mutex_enter(&db->db_mtx);
1529         DBUF_VERIFY(db);

1531         if (db->db_state == DB_FILL) {
1532                 if (db->db_level == 0 && db->db_freed_in_flight) {
1533                         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1534                         /* we were freed while filling */
1535                         /* XXX dbuf_undirty? */
1536                         bzero(db->db.db_data, db->db.db_size);
1537                         db->db_freed_in_flight = FALSE;
1538                 }
1539                 db->db_state = DB_CACHED;
1540                 cv_broadcast(&db->db_changed);
1541         }
1542         mutex_exit(&db->db_mtx);
1543 }

1545 /*
1546  * Directly assign a provided arc buf to a given dbuf if it's not referenced
1547  * by anybody except our caller. Otherwise copy arcbuf's contents to dbuf.
1548  */
1549 void
1550 dbuf_assign_arcbuf(dmu_buf_impl_t *db, arc_buf_t *buf, dmu_tx_t *tx)
1551 {
1552         ASSERT(!refcount_is_zero(&db->db_holds));
1553         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1554         ASSERT(db->db_level == 0);
```



```
1555         ASSERT(DBUF_GET_BUFC_TYPE(db) == ARC_BUFC_DATA);
1556         ASSERT(buf != NULL);
1557         ASSERT(arc_buf_size(buf) == db->db.db_size);
1558         ASSERT(tx->tx_txg != 0);

1560         arc_return_buf(buf, db);
1561         ASSERT(arc_released(buf));

1563         mutex_enter(&db->db_mtx);

1565         while (db->db_state == DB_READ || db->db_state == DB_FILL)
1566                 cv_wait(&db->db_changed, &db->db_mtx);

1568         ASSERT(db->db_state == DB_CACHED || db->db_state == DB_UNCACHED);

1570         if (db->db_state == DB_CACHED &&
1571             refcount_count(&db->db_holds) - 1 > db->db_dirtycnt) {
1572                 mutex_exit(&db->db_mtx);
1573                 (void) dbuf_dirty(db, tx);
1574                 bcopy(buf->b_data, db->db.db_data, db->db.db_size);
1575                 VERIFY(arc_buf_remove_ref(buf, db));
1576                 xuio_stat_wbuf_copied();
1577                 return;
1578         }

1580         xuio_stat_wbuf_nocopy();
1581         if (db->db_state == DB_CACHED) {
1582                 dbuf_dirty_record_t *dr = db->db_last_dirty;

1584                 ASSERT(db->db_buf != NULL);
1585                 if (dr != NULL && dr->dr_txg == tx->tx_txg) {
1586                         ASSERT(dr->dt.dl.dr_data == db->db_buf);
1587                         if (!arc_released(db->db_buf)) {
1588                                 ASSERT(dr->dt.dl.dr_override_state ==
1589                                     DR_OVERRIDDEN);
1590                                 arc_release(db->db_buf, db);
1591                         }
1592                         dr->dt.dl.dr_data = buf;
1593                         VERIFY(arc_buf_remove_ref(db->db_buf, db));
1594                 } else if (dr == NULL || dr->dt.dl.dr_data != db->db_buf) {
1595                         arc_release(db->db_buf, db);
1596                         VERIFY(arc_buf_remove_ref(db->db_buf, db));
1597                 }
1598                 db->db_buf = NULL;
1599         }
1600         ASSERT(db->db_buf == NULL);
1601         dbuf_set_data(db, buf);
1602         db->db_state = DB_FILL;
1603         mutex_exit(&db->db_mtx);
1604         (void) dbuf_dirty(db, tx);
1605         dbuf_fill_done(db, tx);
1606 }

1608 /*
1609  * "Clear" the contents of this dbuf.  This will mark the dbuf
1610  * EVICTING and clear *most* of its references.  Unfortunetely,
1611  * when we are not holding the dn_dbufs_mtx, we can't clear the
1612  * entry in the dn_dbufs list.  We have to wait until dbuf_destroy()
1613  * in this case.  For callers from the DMU we will usually see:
1614  *      dbuf_clear()->arc_buf_evict()->dbuf_do_evict()->dbuf_destroy()
1615  * For the arc callback, we will usually see:
1616  *      dbuf_do_evict()->dbuf_clear();dbuf_destroy()
1617  * Sometimes, though, we will get a mix of these two:
1618  *      DMU: dbuf_clear()->arc_buf_evict()
1619  *      ARC: dbuf_do_evict()->dbuf_destroy()
1620  */
```

```
1621 void
1622 dbuf_clear(dmu_buf_impl_t *db, list_t *evict_list_p)
 772 dbuf_clear(dmu_buf_impl_t *db)
1623 {
1624         dnode_t *dn;
1625         dmu_buf_impl_t *parent = db->db_parent;
1626         dmu_buf_impl_t *dndb;
1627         int dbuf_gone = FALSE;

1629         ASSERT(MUTEX_HELD(&db->db_mtx));
1630         ASSERT(refcount_is_zero(&db->db_holds));

1632         dbuf_evict_user(db, evict_list_p);
 782         dbuf_evict_user(db);

1634         if (db->db_state == DB_CACHED) {
1635                 ASSERT(db->db.db_data != NULL);
1636                 if (db->db_blkid == DMU_BONUS_BLKID) {
1637                         zio_buf_free(db->db.db_data, DN_MAX_BONUSLEN);
1638                         arc_space_return(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
1639                 }
1640                 db->db.db_data = NULL;
1641                 db->db_state = DB_UNCACHED;
1642         }

1644         ASSERT(db->db_state == DB_UNCACHED || db->db_state == DB_NOFILL);
1645         ASSERT(db->db_data_pending == NULL);

1647         db->db_state = DB_EVICTING;
1648         db->db_blkptr = NULL;

1650         DB_DNODE_ENTER(db);
1651         dn = DB_DNODE(db);
1652         dndb = dn->dn_dbuf;
1653         if (db->db_blkid != DMU_BONUS_BLKID && MUTEX_HELD(&dn->dn_dbufs_mtx)) {
1654                 list_remove(&dn->dn_dbufs, db);
1655                 (void) atomic_dec_32_nv(&dn->dn_dbufs_count);
1656                 membar_producer();
1657                 DB_DNODE_EXIT(db);
1658                 /*
1659                  * Decrementing the dbuf count means that the hold corresponding
1660                  * to the removed dbuf is no longer discounted in dnode_move(),
1661                  * so the dnode cannot be moved until after we release the hold.
1662                  * The membar_producer() ensures visibility of the decremented
1663                  * value in dnode_move(), since DB_DNODE_EXIT doesn't actually
1664                  * release any lock.
1665                  */
1666                 dnode_rele(dn, db);
1667                 db->db_dnode_handle = NULL;
1668         } else {
1669                 DB_DNODE_EXIT(db);
1670         }

1672         if (db->db_buf)
1673                 dbuf_gone = arc_buf_evict(db->db_buf);

1675         if (!dbuf_gone)
1676                 mutex_exit(&db->db_mtx);

1678         /*
1679          * If this dbuf is referenced from an indirect dbuf,
1680          * decrement the ref count on the indirect dbuf.
1681          */
1682         if (parent && parent != dndb)
1683                 dbuf_rele(parent, db);
1684 }
```

*_____unchanged_portion_omitted_*


```
1753 static dmu_buf_impl_t *
1754 dbuf_create(dnode_t *dn, uint8_t level, uint64_t blkid,
1755     dmu_buf_impl_t *parent, blkptr_t *blkptr)
1756 {
1757         objset_t *os = dn->dn_objset;
1758         dmu_buf_impl_t *db, *odb;

1760         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1761         ASSERT(dn->dn_type != DMU_OT_NONE);

1763         db = kmem_cache_alloc(dbuf_cache, KM_SLEEP);

1765         db->db_objset = os;
1766         db->db.db_object = dn->dn_object;
1767         db->db_level = level;
1768         db->db_blkid = blkid;
1769         db->db_last_dirty = NULL;
1770         db->db_dirtycnt = 0;
1771         db->db_dnode_handle = dn->dn_handle;
1772         db->db_parent = parent;
1773         db->db_blkptr = blkptr;

1775         db->db_user = NULL;
 925         db->db_user_ptr = NULL;
 926         db->db_user_data_ptr_ptr = NULL;
 927         db->db_evict_func = NULL;
1776         db->db_immediate_evict = 0;
1777         db->db_freed_in_flight = 0;

1779         if (blkid == DMU_BONUS_BLKID) {
1780                 ASSERT3P(parent, ==, dn->dn_dbuf);
1781                 db->db.db_size = DN_MAX_BONUSLEN -
1782                     (dn->dn_nblkptr-1) * sizeof (blkptr_t);
1783                 ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
1784                 db->db.db_offset = DMU_BONUS_BLKID;
1785                 db->db_state = DB_UNCACHED;
1786                 /* the bonus dbuf is not placed in the hash table */
1787                 arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1788                 return (db);
1789         } else if (blkid == DMU_SPILL_BLKID) {
1790                 db->db.db_size = (blkptr != NULL) ?
1791                     BP_GET_LSIZE(blkptr) : SPA_MINBLOCKSIZE;
1792                 db->db.db_offset = 0;
1793         } else {
1794                 int blocksize =
1795                     db->db_level ? 1<<dn->dn_indblkshift :  dn->dn_datablksz;
1796                 db->db.db_size = blocksize;
1797                 db->db.db_offset = db->db_blkid * blocksize;
1798         }

1800         /*
1801          * Hold the dn_dbufs_mtx while we get the new dbuf
1802          * in the hash table *and* added to the dbufs list.
1803          * This prevents a possible deadlock with someone
1804          * trying to look up this dbuf before its added to the
1805          * dn_dbufs list.
1806          */
1807         mutex_enter(&dn->dn_dbufs_mtx);
1808         db->db_state = DB_EVICTING;
1809         if ((odb = dbuf_hash_insert(db)) != NULL) {
1810                 /* someone else inserted it first */
1811                 kmem_cache_free(dbuf_cache, db);
1812                 mutex_exit(&dn->dn_dbufs_mtx);
1813                 return (odb);
1814         }
```

```
1815         list_insert_head(&dn->dn_dbufs, db);
1816         db->db_state = DB_UNCACHED;
1817         mutex_exit(&dn->dn_dbufs_mtx);
1818         arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);

1820         if (parent && parent != dn->dn_dbuf)
1821                 dbuf_add_ref(parent, db);

1823         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1824             refcount_count(&dn->dn_holds) > 0);
1825         (void) refcount_add(&dn->dn_holds, db);
1826         (void) atomic_inc_32_nv(&dn->dn_dbufs_count);

1828         dprintf_dbuf(db, "db=%p\n", db);

1830         return (db);
1831 }

1833 static int
1834 dbuf_do_evict(void *private)
1835 {
1836         arc_buf_t *buf = private;
1837         dmu_buf_impl_t *db = buf->b_private;
1838         list_t evict_list;

1840         dmu_buf_create_user_evict_list(&evict_list);
1841 #endif /* ! codereview */

1843         if (!MUTEX_HELD(&db->db_mtx))
1844                 mutex_enter(&db->db_mtx);

1846         ASSERT(refcount_is_zero(&db->db_holds));

1848         if (db->db_state != DB_EVICTING) {
1849                 ASSERT(db->db_state == DB_CACHED);
1850                 DBUF_VERIFY(db);
1851                 db->db_buf = NULL;
1852                 dbuf_evict(db, &evict_list);
 990                 dbuf_evict(db);
1853         } else {
1854                 mutex_exit(&db->db_mtx);
1855                 dbuf_destroy(db);
1856         }
1857         dmu_buf_destroy_user_evict_list(&evict_list);
1858 #endif /* ! codereview */
1859         return (0);
1860 }

1862 static void
1863 dbuf_destroy(dmu_buf_impl_t *db)
1864 {
1865         ASSERT(refcount_is_zero(&db->db_holds));

1867         if (db->db_blkid != DMU_BONUS_BLKID) {
1868                 /*
1869                  * If this dbuf is still on the dn_dbufs list,
1870                  * remove it from that list.
1871                  */
1872                 if (db->db_dnode_handle != NULL) {
1873                         dnode_t *dn;

1875                         DB_DNODE_ENTER(db);
1876                         dn = DB_DNODE(db);
1877                         mutex_enter(&dn->dn_dbufs_mtx);
1878                         list_remove(&dn->dn_dbufs, db);
1879                         (void) atomic_dec_32_nv(&dn->dn_dbufs_count);
```



```
1880                         mutex_exit(&dn->dn_dbufs_mtx);
1881                         DB_DNODE_EXIT(db);
1882                         /*
1883                          * Decrementing the dbuf count means that the hold
1884                          * corresponding to the removed dbuf is no longer
1885                          * discounted in dnode_move(), so the dnode cannot be
1886                          * moved until after we release the hold.
1887                          */
1888                         dnode_rele(dn, db);
1889                         db->db_dnode_handle = NULL;
1890                 }
1891                 dbuf_hash_remove(db);
1892         }
1893         db->db_parent = NULL;
1894         db->db_buf = NULL;

1896         ASSERT(!list_link_active(&db->db_link));
1897         ASSERT(db->db.db_data == NULL);
1898         ASSERT(db->db_hash_next == NULL);
1899         ASSERT(db->db_blkptr == NULL);
1900         ASSERT(db->db_data_pending == NULL);

1902         kmem_cache_free(dbuf_cache, db);
1903         arc_space_return(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1904 }

1906 void
1907 dbuf_prefetch(dnode_t *dn, uint64_t blkid)
1908 {
1909         dmu_buf_impl_t *db = NULL;
1910         blkptr_t *bp = NULL;

1912         ASSERT(blkid != DMU_BONUS_BLKID);
1913         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));

1915         if (dnode_block_freed(dn, blkid))
1916                 return;

1918         /* dbuf_find() returns with db_mtx held */
1919         if (db = dbuf_find(dn, 0, blkid)) {
1920                 /*
1921                  * This dbuf is already in the cache.  We assume that
1922                  * it is already CACHED, or else about to be either
1923                  * read or filled.
1924                  */
1925                 mutex_exit(&db->db_mtx);
1926                 return;
1927         }

1929         if (dbuf_findbp(dn, 0, blkid, TRUE, &db, &bp) == 0) {
1930                 if (bp && !BP_IS_HOLE(bp)) {
1931                         int priority = dn->dn_type == DMU_OT_DDT_ZAP ?
1932                             ZIO_PRIORITY_DDT_PREFETCH : ZIO_PRIORITY_ASYNC_READ;
1933                         dsl_dataset_t *ds = dn->dn_objset->os_dsl_dataset;
1934                         uint32_t aflags = ARC_NOWAIT | ARC_PREFETCH;
1935                         zbookmark_t zb;

1937                         SET_BOOKMARK(&zb, ds ? ds->ds_object : DMU_META_OBJSET,
1938                             dn->dn_object, 0, blkid);

1940                         (void) arc_read(NULL, dn->dn_objset->os_spa,
1941                             bp, NULL, NULL, priority,
1942                             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE,
1943                             &aflags, &zb);
1944                 }
1945                 if (db)
```

```
1946                        dbuf_rele(db, NULL);
1947        }
1948 }

1950 /*
1951  * Returns with db_holds incremented, and db_mtx not held.
1952  * Note: dn_struct_rwlock must be held.
1953  */
1954 int
1955 dbuf_hold_impl(dnode_t *dn, uint8_t level, uint64_t blkid, int fail_sparse,
1956     void *tag, dmu_buf_impl_t **dbp)
1957 {
1958        dmu_buf_impl_t *db, *parent = NULL;
1959        list_t evict_list;
1960 #endif /* ! codereview */

1962        ASSERT(blkid != DMU_BONUS_BLKID);
1963        ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1964        ASSERT3U(dn->dn_nlevels, >, level);

1966        dmu_buf_create_user_evict_list(&evict_list);

1968 #endif /* ! codereview */
1969        *dbp = NULL;
1970 top:
1971        /* dbuf_find() returns with db_mtx held */
1972        db = dbuf_find(dn, level, blkid);

1974        if (db == NULL) {
1975                blkptr_t *bp = NULL;
1976                int err;

1978                ASSERT3P(parent, ==, NULL);
1979                err = dbuf_findbp(dn, level, blkid, fail_sparse, &parent, &bp);
1980                if (fail_sparse) {
1981                        if (err == 0 && bp && BP_IS_HOLE(bp))
1982                                err = SET_ERROR(ENOENT);
1983                        if (err) {
1984                                if (parent)
1985                                        dbuf_rele(parent, NULL);
1986                                return (err);
1987                        }
1988                }
1989                if (err && err != ENOENT)
1990                        return (err);
1991                db = dbuf_create(dn, level, blkid, parent, bp);
1992        }

1994        if (db->db_buf && refcount_is_zero(&db->db_holds)) {
1995                arc_buf_add_ref(db->db_buf, db);
1996                if (db->db_buf->b_data == NULL) {
1997                        dbuf_clear(db, &evict_list);
 995                        dbuf_clear(db);
1998                        if (parent) {
1999                                dbuf_rele(parent, NULL);
2000                                parent = NULL;
2001                        }
2002                        goto top;
2003                }
2004                ASSERT3P(db->db.db_data, ==, db->db_buf->b_data);
2005        }

2007        ASSERT(db->db_buf == NULL || arc_referenced(db->db_buf));

2009        /*
2010         * If this buffer is currently syncing out, and we are are
```



```
2011         * still referencing it from db_data, we need to make a copy
2012         * of it in case we decide we want to dirty it again in this txg.
2013         */
2014        if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
2015            dn->dn_object != DMU_META_DNODE_OBJECT &&
2016            db->db_state == DB_CACHED && db->db_data_pending) {
2017                dbuf_dirty_record_t *dr = db->db_data_pending;

2019                if (dr->dt.dl.dr_data == db->db_buf) {
2020                        arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);

2022                        dbuf_set_data(db,
2023                            arc_buf_alloc(dn->dn_objset->os_spa,
2024                            db->db.db_size, db, type));
2025                        bcopy(dr->dt.dl.dr_data->b_data, db->db.db_data,
2026                            db->db.db_size);
2027                }
2028        }

2030        (void) refcount_add(&db->db_holds, tag);
1029        dbuf_update_data(db);
2031        DBUF_VERIFY(db);
2032        mutex_exit(&db->db_mtx);

2034        dmu_buf_destroy_user_evict_list(&evict_list);

2036 #endif /* ! codereview */
2037        /* NOTE: we can't rele the parent until after we drop the db_mtx */
2038        if (parent)
2039                dbuf_rele(parent, NULL);

2041        ASSERT3P(DB_DNODE(db), ==, dn);
2042        ASSERT3U(db->db_blkid, ==, blkid);
2043        ASSERT3U(db->db_level, ==, level);
2044        *dbp = db;

2046        return (0);
2047 }

2049 dmu_buf_impl_t *
2050 dbuf_hold(dnode_t *dn, uint64_t blkid, void *tag)
2051 {
2052        dmu_buf_impl_t *db;
2053        int err = dbuf_hold_impl(dn, 0, blkid, FALSE, tag, &db);
2054        return (err ? NULL : db);
2055 }

2057 dmu_buf_impl_t *
2058 dbuf_hold_level(dnode_t *dn, int level, uint64_t blkid, void *tag)
2059 {
2060        dmu_buf_impl_t *db;
2061        int err = dbuf_hold_impl(dn, level, blkid, FALSE, tag, &db);
2062        return (err ? NULL : db);
2063 }

2065 void
2066 dbuf_create_bonus(dnode_t *dn)
2067 {
2068        ASSERT(RW_WRITE_HELD(&dn->dn_struct_rwlock));

2070        ASSERT(dn->dn_bonus == NULL);
2071        dn->dn_bonus = dbuf_create(dn, 0, DMU_BONUS_BLKID, dn->dn_dbuf, NULL);
2072 }

2074 int
2075 dbuf_spill_set_blksz(dmu_buf_t *db_fake, uint64_t blksz, dmu_tx_t *tx)
```

```
2076 {
2077         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
2078         dnode_t *dn;
2079
2080         if (db->db_blkid != DMU_SPILL_BLKID)
2081                 return (SET_ERROR(ENOTSUP));
2082         if (blksz == 0)
2083                 blksz = SPA_MINBLOCKSIZE;
2084         if (blksz > SPA_MAXBLOCKSIZE)
2085                 blksz = SPA_MAXBLOCKSIZE;
2086         else
2087                 blksz = P2ROUNDUP(blksz, SPA_MINBLOCKSIZE);
2088
2089         DB_DNODE_ENTER(db);
2090         dn = DB_DNODE(db);
2091         rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
2092         dbuf_new_size(db, blksz, tx);
2093         rw_exit(&dn->dn_struct_rwlock);
2094         DB_DNODE_EXIT(db);
2095
2096         return (0);
2097 }
2098
2099 void
2100 dbuf_rm_spill(dnode_t *dn, dmu_tx_t *tx)
2101 {
2102         dbuf_free_range(dn, DMU_SPILL_BLKID, DMU_SPILL_BLKID, tx);
2103 }
2104
2105 #pragma weak dmu_buf_add_ref = dbuf_add_ref
2106 void
2107 dbuf_add_ref(dmu_buf_impl_t *db, void *tag)
2108 {
2109         int64_t holds = refcount_add(&db->db_holds, tag);
2110         ASSERT(holds > 1);
2111 }
2112
2113 /*
2114  * If you call dbuf_rele() you had better not be referencing the dnode handle
2115  * unless you have some other direct or indirect hold on the dnode. (An indirect
2116  * hold is a hold on one of the dnode's dbufs, including the bonus buffer.)
2117  * Without that, the dbuf_rele() could lead to a dnode_rele() followed by the
2118  * dnode's parent dbuf evicting its dnode handles.
2119  */
2120 #pragma weak dmu_buf_rele = dbuf_rele
2121 void
2122 dbuf_rele(dmu_buf_impl_t *db, void *tag)
2123 {
2124         mutex_enter(&db->db_mtx);
2125         dbuf_rele_and_unlock(db, tag);
2126 }
2127
2128 /*
2129  * dbuf_rele() for an already-locked dbuf.  This is necessary to allow
2130  * db_dirtycnt and db_holds to be updated atomically.
2131  */
2132 void
2133 dbuf_rele_and_unlock(dmu_buf_impl_t *db, void *tag)
2134 {
2135         int64_t holds;
2136         list_t evict_list;
2137 #endif /* ! codereview */
2138
2139         ASSERT(MUTEX_HELD(&db->db_mtx));
2140         DBUF_VERIFY(db);
```



```
2142         dmu_buf_create_user_evict_list(&evict_list);
2143
2144 #endif /* ! codereview */
2145         /*
2146          * Remove the reference to the dbuf before removing its hold on the
2147          * dnode so we can guarantee in dnode_move() that a referenced bonus
2148          * buffer has a corresponding dnode hold.
2149          */
2150         holds = refcount_remove(&db->db_holds, tag);
2151         ASSERT(holds >= 0);
2152
2153         /*
2154          * We can't freeze indirects if there is a possibility that they
2155          * may be modified in the current syncing context.
2156          */
2157         if (db->db_buf && holds == (db->db_level == 0 ? db->db_dirtycnt : 0))
2158                 arc_buf_freeze(db->db_buf);
2159
2160         if (holds == db->db_dirtycnt &&
2161             db->db_level == 0 && db->db_immediate_evict)
2162                 dbuf_evict_user(db, &evict_list);
1033                 dbuf_evict_user(db);
2163
2164         if (holds == 0) {
2165                 if (db->db_blkid == DMU_BONUS_BLKID) {
2166                         mutex_exit(&db->db_mtx);
2167
2168                         /*
2169                          * If the dnode moves here, we cannot cross this barrier
2170                          * until the move completes.
2171                          */
2172                         DB_DNODE_ENTER(db);
2173                         (void) atomic_dec_32_nv(&DB_DNODE(db)->dn_dbufs_count);
2174                         DB_DNODE_EXIT(db);
2175                         /*
2176                          * The bonus buffer's dnode hold is no longer discounted
2177                          * in dnode_move(). The dnode cannot move until after
2178                          * the dnode_rele().
2179                          */
2180                         dnode_rele(DB_DNODE(db), db);
2181                 } else if (db->db_buf == NULL) {
2182                         /*
2183                          * This is a special case: we never associated this
2184                          * dbuf with any data allocated from the ARC.
2185                          */
2186                         ASSERT(db->db_state == DB_UNCACHED ||
2187                             db->db_state == DB_NOFILL);
2188                         dbuf_evict(db, &evict_list);
1059                         dbuf_evict(db);
2189                 } else if (arc_released(db->db_buf)) {
2190                         arc_buf_t *buf = db->db_buf;
2191                         /*
2192                          * This dbuf has anonymous data associated with it.
2193                          */
2194                         dbuf_clear_data(db, &evict_list);
1065                         dbuf_set_data(db, NULL);
2195                         VERIFY(arc_buf_remove_ref(buf, db));
2196                         dbuf_evict(db, &evict_list);
1067                         dbuf_evict(db);
2197                 } else {
2198                         VERIFY(!arc_buf_remove_ref(db->db_buf, db));
2199
2200                         /*
2201                          * A dbuf will be eligible for eviction if either the
2202                          * 'primarycache' property is set or a duplicate
2203                          * copy of this buffer is already cached in the arc.
```

```
2204                          *
2205                          * In the case of the 'primarycache' a buffer
2206                          * is considered for eviction if it matches the
2207                          * criteria set in the property.
2208                          *
2209                          * To decide if our buffer is considered a
2210                          * duplicate, we must call into the arc to determine
2211                          * if multiple buffers are referencing the same
2212                          * block on-disk. If so, then we simply evict
2213                          * ourselves.
2214                          */
2215                         if (!DBUF_IS_CACHEABLE(db) ||
2216                             arc_buf_eviction_needed(db->db_buf))
2217                                 dbuf_clear(db, &evict_list);
1088                                  dbuf_clear(db);
2218                         else
2219                                 mutex_exit(&db->db_mtx);
2220                 }
2221         } else {
2222                 mutex_exit(&db->db_mtx);
2223         }
2224         dmu_buf_destroy_user_evict_list(&evict_list);
2225 #endif /* ! codereview */
2226 }

2228 #pragma weak dmu_buf_refcount = dbuf_refcount
2229 uint64_t
2230 dbuf_refcount(dmu_buf_impl_t *db)
2231 {
2232         return (refcount_count(&db->db_holds));
2233 }

1095 void *
1096 dmu_buf_set_user(dmu_buf_t *db_fake, void *user_ptr, void *user_data_ptr_ptr,
1097     dmu_buf_evict_func_t *evict_func)
1098 {
1099         return (dmu_buf_update_user(db_fake, NULL, user_ptr,
1100             user_data_ptr_ptr, evict_func));
1101 }

1103 void *
1104 dmu_buf_set_user_ie(dmu_buf_t *db_fake, void *user_ptr, void *user_data_ptr_ptr,
1105     dmu_buf_evict_func_t *evict_func)
1106 {
1107         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

1109         db->db_immediate_evict = TRUE;
1110         return (dmu_buf_update_user(db_fake, NULL, user_ptr,
1111             user_data_ptr_ptr, evict_func));
1112 }

1114 void *
1115 dmu_buf_update_user(dmu_buf_t *db_fake, void *old_user_ptr, void *user_ptr,
1116     void *user_data_ptr_ptr, dmu_buf_evict_func_t *evict_func)
1117 {
1118         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
1119         ASSERT(db->db_level == 0);

1121         ASSERT((user_ptr == NULL) == (evict_func == NULL));

1123         mutex_enter(&db->db_mtx);

1125         if (db->db_user_ptr == old_user_ptr) {
1126                 db->db_user_ptr = user_ptr;
1127                 db->db_user_data_ptr_ptr = user_data_ptr_ptr;
1128                 db->db_evict_func = evict_func;
```



```
1130                 dbuf_update_data(db);
1131         } else {
1132                 old_user_ptr = db->db_user_ptr;
1133         }

1135         mutex_exit(&db->db_mtx);
1136         return (old_user_ptr);
1137 }

1139 void *
1140 dmu_buf_get_user(dmu_buf_t *db_fake)
1141 {
1142         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
1143         ASSERT(!refcount_is_zero(&db->db_holds));

1145         return (db->db_user_ptr);
1146 }

2235 boolean_t
2236 dmu_buf_freeable(dmu_buf_t *dbuf)
2237 {
2238         boolean_t res = B_FALSE;
2239         dmu_buf_impl_t *db = (dmu_buf_impl_t *)dbuf;

2241         if (db->db_blkptr)
2242                 res = dsl_dataset_block_freeable(db->db_objset->os_dsl_dataset,
2243                     db->db_blkptr, db->db_blkptr->blk_birth);

2245         return (res);
2246 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   56719 Thu Apr 25 16:14:57 2013
new/usr/src/uts/common/fs/zfs/dnode.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_

 433 /*
 434  * Caller must be holding the dnode handle, which is released upon return.
 435  */
 436 static void
 437 dnode_destroy(dnode_t *dn)
 438 {
 439         objset_t *os = dn->dn_objset;

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

 443         mutex_enter(&os->os_lock);
 444         POINTER_INVALIDATE(&dn->dn_objset);
 445         list_remove(&os->os_dnodes, dn);
 446         mutex_exit(&os->os_lock);

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

 451         dn->dn_allocated_txg = 0;
 452         dn->dn_free_txg = 0;
 453         dn->dn_assigned_txg = 0;

 455         dn->dn_dirtyctx = 0;
 456         if (dn->dn_dirtyctx_firstset != NULL) {
 457                 kmem_free(dn->dn_dirtyctx_firstset, 1);
 458                 dn->dn_dirtyctx_firstset = NULL;
 459         }
 460         if (dn->dn_bonus != NULL) {
 461                 list_t evict_list;

 463                 dmu_buf_create_user_evict_list(&evict_list);
 464 #endif /* ! codereview */
 465                 mutex_enter(&dn->dn_bonus->db_mtx);
 466                 dbuf_evict(dn->dn_bonus, &evict_list);
 467                 dmu_buf_destroy_user_evict_list(&evict_list);
 461                 dbuf_evict(dn->dn_bonus);
 468                 dn->dn_bonus = NULL;
 469         }
 470         dn->dn_zio = NULL;

 472         dn->dn_have_spill = B_FALSE;
 473         dn->dn_oldused = 0;
 474         dn->dn_oldflags = 0;
 475         dn->dn_olduid = 0;
 476         dn->dn_oldgid = 0;
 477         dn->dn_newuid = 0;
 478         dn->dn_newgid = 0;
 479         dn->dn_id_flags = 0;

 481         dmu_zfetch_rele(&dn->dn_zfetch);
 482         kmem_cache_free(dnode_cache, dn);
 483         arc_space_return(sizeof (dnode_t), ARC_SPACE_OTHER);
 484 }
______unchanged_portion_omitted_

 964 static void
 965 dnode_buf_pageout(dmu_buf_user_t *dbu)
```



```
 959 dnode_buf_pageout(dmu_buf_t *db, void *arg)
 966 {
 967         dnode_children_t *children_dnodes = (dnode_children_t *)dbu;
 961         dnode_children_t *children_dnodes = arg;
 968         int i;
 963         int epb = db->db_size >> DNODE_SHIFT;

 965         ASSERT(epb == children_dnodes->dnc_count);

 970         for (i = 0; i < children_dnodes->dnc_count; i++) {
 967         for (i = 0; i < epb; i++) {
 971                 dnode_handle_t *dnh = &children_dnodes->dnc_children[i];
 972                 dnode_t *dn;

 974                 /*
 975                  * The dnode handle lock guards against the dnode moving to
 976                  * another valid address, so there is no need here to guard
 977                  * against changes to or from NULL.
 978                  */
 979                 if (dnh->dnh_dnode == NULL) {
 980                         zrl_destroy(&dnh->dnh_zrlock);
 981                         continue;
 982                 }

 984                 zrl_add(&dnh->dnh_zrlock);
 985                 dn = dnh->dnh_dnode;
 986                 /*
 987                  * If there are holds on this dnode, then there should
 988                  * be holds on the dnode's containing dbuf as well; thus
 989                  * it wouldn't be eligible for eviction and this function
 990                  * would not have been called.
 991                  */
 992                 ASSERT(refcount_is_zero(&dn->dn_holds));
 993                 ASSERT(refcount_is_zero(&dn->dn_tx_holds));

 995                 dnode_destroy(dn); /* implicit zrl_remove() */
 996                 zrl_destroy(&dnh->dnh_zrlock);
 997                 dnh->dnh_dnode = NULL;
 998         }
 999         kmem_free(children_dnodes, sizeof (dnode_children_t) +
1000             (children_dnodes->dnc_count - 1) * sizeof (dnode_handle_t));
 997             (epb - 1) * sizeof (dnode_handle_t));
1001 }

1003 /*
1004  * errors:
1005  * EINVAL - invalid object number.
1006  * EIO - i/o error.
1007  * succeeds even for free dnodes.
1008  */
1009 int
1010 dnode_hold_impl(objset_t *os, uint64_t object, int flag,
1011     void *tag, dnode_t **dnp)
1012 {
1013         int epb, idx, err;
1014         int drop_struct_lock = FALSE;
1015         int type;
1016         uint64_t blk;
1017         dnode_t *mdn, *dn;
1018         dmu_buf_impl_t *db;
1019         dnode_children_t *children_dnodes;
1020         dnode_handle_t *dnh;

1022         /*
1023          * If you are holding the spa config lock as writer, you shouldn't
1024          * be asking the DMU to do *anything* unless it's the root pool
```

```
1025          * which may require us to read from the root filesystem while
1026          * holding some (not all) of the locks as writer.
1027          */
1028         ASSERT(spa_config_held(os->os_spa, SCL_ALL, RW_WRITER) == 0 ||
1029             (spa_is_root(os->os_spa) &&
1030             spa_config_held(os->os_spa, SCL_STATE, RW_WRITER)));

1032         if (object == DMU_USERUSED_OBJECT || object == DMU_GROUPUSED_OBJECT) {
1033                 dn = (object == DMU_USERUSED_OBJECT) ?
1034                     DMU_USERUSED_DNODE(os) : DMU_GROUPUSED_DNODE(os);
1035                 if (dn == NULL)
1036                         return (SET_ERROR(ENOENT));
1037                 type = dn->dn_type;
1038                 if ((flag & DNODE_MUST_BE_ALLOCATED) && type == DMU_OT_NONE)
1039                         return (SET_ERROR(ENOENT));
1040                 if ((flag & DNODE_MUST_BE_FREE) && type != DMU_OT_NONE)
1041                         return (SET_ERROR(EEXIST));
1042                 DNODE_VERIFY(dn);
1043                 (void) refcount_add(&dn->dn_holds, tag);
1044                 *dnp = dn;
1045                 return (0);
1046         }

1048         if (object == 0 || object >= DN_MAX_OBJECT)
1049                 return (SET_ERROR(EINVAL));

1051         mdn = DMU_META_DNODE(os);
1052         ASSERT(mdn->dn_object == DMU_META_DNODE_OBJECT);

1054         DNODE_VERIFY(mdn);

1056         if (!RW_WRITE_HELD(&mdn->dn_struct_rwlock)) {
1057                 rw_enter(&mdn->dn_struct_rwlock, RW_READER);
1058                 drop_struct_lock = TRUE;
1059         }

1061         blk = dbuf_whichblock(mdn, object * sizeof (dnode_phys_t));

1063         db = dbuf_hold(mdn, blk, FTAG);
1064         if (drop_struct_lock)
1065                 rw_exit(&mdn->dn_struct_rwlock);
1066         if (db == NULL)
1067                 return (SET_ERROR(EIO));
1068         err = dbuf_read(db, NULL, DB_RF_CANFAIL);
1069         if (err) {
1070                 dbuf_rele(db, FTAG);
1071                 return (err);
1072         }

1074         ASSERT3U(db->db.db_size, >=, 1<<DNODE_SHIFT);
1075         epb = db->db.db_size >> DNODE_SHIFT;

1077         idx = object & (epb-1);

1079         ASSERT(DB_DNODE(db)->dn_type == DMU_OT_DNODE);
1080         children_dnodes = (dnode_children_t *)dmu_buf_get_user(&db->db);
1077         children_dnodes = dmu_buf_get_user(&db->db);
1081         if (children_dnodes == NULL) {
1082                 int i;
1083                 dnode_children_t *winner;
1084                 children_dnodes = kmem_alloc(sizeof (dnode_children_t) +
1085                     (epb - 1) * sizeof (dnode_handle_t), KM_SLEEP);
1086                 children_dnodes->dnc_count = epb;
1087                 dnh = &children_dnodes->dnc_children[0];
1088                 for (i = 0; i < epb; i++) {
1089                         zrl_init(&dnh[i].dnh_zrlock);
```



```
1090                         dnh[i].dnh_dnode = NULL;
1091                 }
1092                 dmu_buf_init_user(&children_dnodes->db_evict,
1093                     dnode_buf_pageout);
1094                 winner = (dnode_children_t *)
1095                     dmu_buf_set_user(&db->db, &children_dnodes->db_evict);
1096                 if (winner) {
1089                 if (winner = dmu_buf_set_user(&db->db, children_dnodes, NULL,
1090                     dnode_buf_pageout)) {
1097                         kmem_free(children_dnodes, sizeof (dnode_children_t) +
1098                             (epb - 1) * sizeof (dnode_handle_t));
1099                         children_dnodes = winner;
1100                 }
1101         }
1102         ASSERT(children_dnodes->dnc_count == epb);

1104         dnh = &children_dnodes->dnc_children[idx];
1105         zrl_add(&dnh->dnh_zrlock);
1106         if ((dn = dnh->dnh_dnode) == NULL) {
1107                 dnode_phys_t *phys = (dnode_phys_t *)db->db.db_data+idx;
1108                 dnode_t *winner;

1110                 dn = dnode_create(os, phys, db, object, dnh);
1111                 winner = atomic_cas_ptr(&dnh->dnh_dnode, NULL, dn);
1112                 if (winner != NULL) {
1113                         zrl_add(&dnh->dnh_zrlock);
1114                         dnode_destroy(dn); /* implicit zrl_remove() */
1115                         dn = winner;
1116                 }
1117         }

1119         mutex_enter(&dn->dn_mtx);
1120         type = dn->dn_type;
1121         if (dn->dn_free_txg ||
1122             ((flag & DNODE_MUST_BE_ALLOCATED) && type == DMU_OT_NONE) ||
1123             ((flag & DNODE_MUST_BE_FREE) &&
1124             (type != DMU_OT_NONE || !refcount_is_zero(&dn->dn_holds)))) {
1125                 mutex_exit(&dn->dn_mtx);
1126                 zrl_remove(&dnh->dnh_zrlock);
1127                 dbuf_rele(db, FTAG);
1128                 return (type == DMU_OT_NONE ? ENOENT : EEXIST);
1129         }
1130         mutex_exit(&dn->dn_mtx);

1132         if (refcount_add(&dn->dn_holds, tag) == 1)
1133                 dbuf_add_ref(db, dnh);
1134         /* Now we can rely on the hold to prevent the dnode from moving. */
1135         zrl_remove(&dnh->dnh_zrlock);

1137         DNODE_VERIFY(dn);
1138         ASSERT3P(dn->dn_dbuf, ==, db);
1139         ASSERT3U(dn->dn_object, ==, object);
1140         dbuf_rele(db, FTAG);

1142         *dnp = dn;
1143         return (0);
1144 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   19546 Thu Apr 25 16:14:57 2013
new/usr/src/uts/common/fs/zfs/dnode_sync.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_

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

 382         dmu_buf_create_user_evict_list(&evict_list);
 383 #endif /* ! codereview */

 385         do {
 386                 dmu_buf_impl_t *db, marker;
 387                 int evicting = FALSE;

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

 402                         mutex_enter(&db->db_mtx);
 403                         if (db->db_state == DB_EVICTING) {
 404                                 progress = TRUE;
 405                                 evicting = TRUE;
 406                                 mutex_exit(&db->db_mtx);
 407                         } else if (refcount_is_zero(&db->db_holds)) {
 408                                 progress = TRUE;
 409                                 dbuf_clear(db, &evict_list); /* exits db_mtx */
 380                                 dbuf_clear(db); /* exits db_mtx for us */
 410                         } else {
 411                                 mutex_exit(&db->db_mtx);
 412                         }
 413                         ASSERT(MUTEX_NOT_HELD(&db->db_mtx));
 414                         dmu_buf_process_user_evicts(&evict_list);

 415                 }
 416                 list_remove(&dn->dn_dbufs, &marker);
 417                 /*
 418                  * NB: we need to drop dn_dbufs_mtx between passes so
 419                  * that any DB_EVICTING dbufs can make progress.
 420                  * Ideally, we would have some cv we could wait on, but
 421                  * since we don't, just wait a bit to give the other
 422                  * thread a chance to run.
 423                  */
 424                 mutex_exit(&dn->dn_dbufs_mtx);
 425                 if (evicting)
 426                         delay(1);
```



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

 431         rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
 432         if (dn->dn_bonus && refcount_is_zero(&dn->dn_bonus->db_holds)) {
 433                 mutex_enter(&dn->dn_bonus->db_mtx);
 434                 dbuf_evict(dn->dn_bonus, &evict_list);
 404                 dbuf_evict(dn->dn_bonus);
 435                 dn->dn_bonus = NULL;
 436         }
 437         rw_exit(&dn->dn_struct_rwlock);
 438         dmu_buf_destroy_user_evict_list(&evict_list);
 439 #endif /* ! codereview */
 440 }

 442 static void
 443 dnode_undirty_dbufs(list_t *list)
 444 {
 445         dbuf_dirty_record_t *dr;

 447         while (dr = list_head(list)) {
 448                 dmu_buf_impl_t *db = dr->dr_dbuf;
 449                 uint64_t txg = dr->dr_txg;

 451                 if (db->db_level != 0)
 452                         dnode_undirty_dbufs(&dr->dt.di.dr_children);

 454                 mutex_enter(&db->db_mtx);
 455                 /* XXX - use dbuf_undirty()? */
 456                 list_remove(list, dr);
 457                 ASSERT(db->db_last_dirty == dr);
 458                 db->db_last_dirty = NULL;
 459                 db->db_dirtycnt -= 1;
 460                 if (db->db_level == 0) {
 461                         ASSERT(db->db_blkid == DMU_BONUS_BLKID ||
 462                             dr->dt.dl.dr_data == db->db_buf);
 463                         dbuf_unoverride(dr);
 464                 }
 465                 kmem_free(dr, sizeof (dbuf_dirty_record_t));
 466                 dbuf_rele_and_unlock(db, (void *)(uintptr_t)txg);
 467         }
 468 }

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

 475         ASSERT(dmu_tx_is_syncing(tx));

 477         /*
 478          * Our contents should have been freed in dnode_sync() by the
 479          * free range record inserted by the caller of dnode_free().
 480          */
 481         ASSERT0(DN_USED_BYTES(dn->dn_phys));
 482         ASSERT(BP_IS_HOLE(dn->dn_phys->dn_blkptr));

 484         dnode_undirty_dbufs(&dn->dn_dirty_records[txgoff]);
 485         dnode_evict_dbufs(dn);
 486         ASSERT3P(list_head(&dn->dn_dbufs), ==, NULL);
 487         ASSERT3P(dn->dn_bonus, ==, NULL);

 489         /*
 490          * XXX - It would be nice to assert this, but we may still
 491          * have residual holds from async evictions from the arc...
```

```
492          *
493          * zfs_obj_to_path() also depends on this being
494          * commented out.
495          *
496          * ASSERT3U(refcount_count(&dn->dn_holds), ==, 1);
497          */

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

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

508         ASSERT(dn->dn_free_txg > 0);
509         if (dn->dn_allocated_txg != dn->dn_free_txg)
510                 dbuf_will_dirty(dn->dn_dbuf, tx);
511         bzero(dn->dn_phys, sizeof (dnode_phys_t));

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

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

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

530 /*
531  * Write out the dnode's dirty buffers.
532  */
533 void
534 dnode_sync(dnode_t *dn, dmu_tx_t *tx)
535 {
536         free_range_t *rp;
537         dnode_phys_t *dnp = dn->dn_phys;
538         int txgoff = tx->tx_txg & TXG_MASK;
539         list_t *list = &dn->dn_dirty_records[txgoff];
540         static const dnode_phys_t zerodn = { 0 };
541         boolean_t kill_spill = B_FALSE;

543         ASSERT(dmu_tx_is_syncing(tx));
544         ASSERT(dnp->dn_type != DMU_OT_NONE || dn->dn_allocated_txg);
545         ASSERT(dnp->dn_type != DMU_OT_NONE ||
546             bcmp(dnp, &zerodn, DNODE_SIZE) == 0);
547         DNODE_VERIFY(dn);

549         ASSERT(dn->dn_dbuf == NULL || arc_released(dn->dn_dbuf->db_buf));

551         if (dmu_objset_userused_enabled(dn->dn_objset) &&
552             !DMU_OBJECT_IS_SPECIAL(dn->dn_object)) {
553                 mutex_enter(&dn->dn_mtx);
554                 dn->dn_oldused = DN_USED_BYTES(dn->dn_phys);
555                 dn->dn_oldflags = dn->dn_phys->dn_flags;
556                 dn->dn_phys->dn_flags |= DNODE_FLAG_USERUSED_ACCOUNTED;
557                 mutex_exit(&dn->dn_mtx);
```



```
558                 dmu_objset_userquota_get_ids(dn, B_FALSE, tx);
559         } else {
560                 /* Once we account for it, we should always account for it. */
561                 ASSERT(!(dn->dn_phys->dn_flags &
562                     DNODE_FLAG_USERUSED_ACCOUNTED));
563         }

565         mutex_enter(&dn->dn_mtx);
566         if (dn->dn_allocated_txg == tx->tx_txg) {
567                 /* The dnode is newly allocated or reallocated */
568                 if (dnp->dn_type == DMU_OT_NONE) {
569                         /* this is a first alloc, not a realloc */
570                         dnp->dn_nlevels = 1;
571                         dnp->dn_nblkptr = dn->dn_nblkptr;
572                 }

574                 dnp->dn_type = dn->dn_type;
575                 dnp->dn_bonustype = dn->dn_bonustype;
576                 dnp->dn_bonuslen = dn->dn_bonuslen;
577         }

579         ASSERT(dnp->dn_nlevels > 1 ||
580             BP_IS_HOLE(&dnp->dn_blkptr[0]) ||
581             BP_GET_LSIZE(&dnp->dn_blkptr[0]) ==
582             dnp->dn_datablkszsec << SPA_MINBLOCKSHIFT);

584         if (dn->dn_next_blksz[txgoff]) {
585                 ASSERT(P2PHASE(dn->dn_next_blksz[txgoff],
586                     SPA_MINBLOCKSIZE) == 0);
587                 ASSERT(BP_IS_HOLE(&dnp->dn_blkptr[0]) ||
588                     dn->dn_maxblkid == 0 || list_head(list) != NULL ||
589                     avl_last(&dn->dn_ranges[txgoff]) ||
590                     dn->dn_next_blksz[txgoff] >> SPA_MINBLOCKSHIFT ==
591                     dnp->dn_datablkszsec);
592                 dnp->dn_datablkszsec =
593                     dn->dn_next_blksz[txgoff] >> SPA_MINBLOCKSHIFT;
594                 dn->dn_next_blksz[txgoff] = 0;
595         }

597         if (dn->dn_next_bonuslen[txgoff]) {
598                 if (dn->dn_next_bonuslen[txgoff] == DN_ZERO_BONUSLEN)
599                         dnp->dn_bonuslen = 0;
600                 else
601                         dnp->dn_bonuslen = dn->dn_next_bonuslen[txgoff];
602                 ASSERT(dnp->dn_bonuslen <= DN_MAX_BONUSLEN);
603                 dn->dn_next_bonuslen[txgoff] = 0;
604         }

606         if (dn->dn_next_bonustype[txgoff]) {
607                 ASSERT(DMU_OT_IS_VALID(dn->dn_next_bonustype[txgoff]));
608                 dnp->dn_bonustype = dn->dn_next_bonustype[txgoff];
609                 dn->dn_next_bonustype[txgoff] = 0;
610         }

612         /*
613          * We will either remove a spill block when a file is being removed
614          * or we have been asked to remove it.
615          */
616         if (dn->dn_rm_spillblk[txgoff] ||
617             ((dnp->dn_flags & DNODE_FLAG_SPILL_BLKPTR) &&
618             dn->dn_free_txg > 0 && dn->dn_free_txg <= tx->tx_txg)) {
619                 if ((dnp->dn_flags & DNODE_FLAG_SPILL_BLKPTR))
620                         kill_spill = B_TRUE;
621                 dn->dn_rm_spillblk[txgoff] = 0;
622         }
```

```
624         if (dn->dn_next_indblkshift[txgoff]) {
625                 ASSERT(dnp->dn_nlevels == 1);
626                 dnp->dn_indblkshift = dn->dn_next_indblkshift[txgoff];
627                 dn->dn_next_indblkshift[txgoff] = 0;
628         }

630         /*
631          * Just take the live (open-context) values for checksum and compress.
632          * Strictly speaking it's a future leak, but nothing bad happens if we
633          * start using the new checksum or compress algorithm a little early.
634          */
635         dnp->dn_checksum = dn->dn_checksum;
636         dnp->dn_compress = dn->dn_compress;

638         mutex_exit(&dn->dn_mtx);

640         if (kill_spill) {
641                 (void) free_blocks(dn, &dn->dn_phys->dn_spill, 1, tx);
642                 mutex_enter(&dn->dn_mtx);
643                 dnp->dn_flags &= ~DNODE_FLAG_SPILL_BLKPTR;
644                 mutex_exit(&dn->dn_mtx);
645         }

647         /* process all the "freed" ranges in the file */
648         while (rp = avl_last(&dn->dn_ranges[txgoff])) {
649                 dnode_sync_free_range(dn, rp->fr_blkid, rp->fr_nblks, tx);
650                 /* grab the mutex so we don't race with dnode_block_freed() */
651                 mutex_enter(&dn->dn_mtx);
652                 avl_remove(&dn->dn_ranges[txgoff], rp);
653                 mutex_exit(&dn->dn_mtx);
654                 kmem_free(rp, sizeof (free_range_t));
655         }

657         if (dn->dn_free_txg > 0 && dn->dn_free_txg <= tx->tx_txg) {
658                 dnode_sync_free(dn, tx);
659                 return;
660         }

662         if (dn->dn_next_nblkptr[txgoff]) {
663                 /* this should only happen on a realloc */
664                 ASSERT(dn->dn_allocated_txg == tx->tx_txg);
665                 if (dn->dn_next_nblkptr[txgoff] > dnp->dn_nblkptr) {
666                         /* zero the new blkptrs we are gaining */
667                         bzero(dnp->dn_blkptr + dnp->dn_nblkptr,
668                             sizeof (blkptr_t) *
669                             (dn->dn_next_nblkptr[txgoff] - dnp->dn_nblkptr));
670 #ifdef ZFS_DEBUG
671                 } else {
672                         int i;
673                         ASSERT(dn->dn_next_nblkptr[txgoff] < dnp->dn_nblkptr);
674                         /* the blkptrs we are losing better be unallocated */
675                         for (i = dn->dn_next_nblkptr[txgoff];
676                             i < dnp->dn_nblkptr; i++)
677                                 ASSERT(BP_IS_HOLE(&dnp->dn_blkptr[i]));
678 #endif
679                 }
680                 mutex_enter(&dn->dn_mtx);
681                 dnp->dn_nblkptr = dn->dn_next_nblkptr[txgoff];
682                 dn->dn_next_nblkptr[txgoff] = 0;
683                 mutex_exit(&dn->dn_mtx);
684         }

686         if (dn->dn_next_nlevels[txgoff]) {
687                 dnode_increase_indirection(dn, tx);
688                 dn->dn_next_nlevels[txgoff] = 0;
689         }
```



```
691         dbuf_sync_list(list, tx);

693         if (!DMU_OBJECT_IS_SPECIAL(dn->dn_object)) {
694                 ASSERT3P(list_head(list), ==, NULL);
695                 dnode_rele(dn, (void *)(uintptr_t)tx->tx_txg);
696         }

698         /*
699          * Although we have dropped our reference to the dnode, it
700          * can't be evicted until its written, and we haven't yet
701          * initiated the IO for the dnode's dbuf.
702          */
703 }
```

```
**********************************************************
   80781 Thu Apr 25 16:14:57 2013
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_

 242 /* ARGSUSED */
 243 static void
 244 dsl_dataset_evict_impl(dsl_dataset_t *ds, boolean_t evict_deadlist)
 244 dsl_dataset_evict(dmu_buf_t *db, void *dsv)
 245 {
 246         dsl_dataset_t *ds = dsv;

 247         ASSERT(ds->ds_owner == NULL);

 249         ds->ds_dbuf = NULL;

 251 #endif /* ! codereview */
 252         unique_remove(ds->ds_fsid_guid);

 254         if (ds->ds_objset != NULL)
 255                 dmu_objset_evict(ds->ds_objset);

 257         if (ds->ds_prev) {
 258                 dsl_dataset_rele(ds->ds_prev, ds);
 259                 ds->ds_prev = NULL;
 260         }

 262         bplist_destroy(&ds->ds_pending_deadlist);
 263         if (evict_deadlist)
 250         if (ds->ds_phys->ds_deadlist_obj != 0)
 264                 dsl_deadlist_close(&ds->ds_deadlist);
 265         if (ds->ds_dir)
 266                 dsl_dir_rele(ds->ds_dir, ds);

 268         ASSERT(!list_link_active(&ds->ds_synced_link));

 270         mutex_destroy(&ds->ds_lock);
 271         mutex_destroy(&ds->ds_opening_lock);
 272         refcount_destroy(&ds->ds_longholds);

 274         kmem_free(ds, sizeof (dsl_dataset_t));
 275 }

 277 /* ARGSUSED */
 278 static void
 279 dsl_dataset_evict(dmu_buf_user_t *dbu)
 280 {
 281         dsl_dataset_evict_impl((dsl_dataset_t *)dbu, B_TRUE);
 282 }

 284 #endif /* ! codereview */
 285 int
 286 dsl_dataset_get_snapname(dsl_dataset_t *ds)
 287 {
 288         dsl_dataset_phys_t *headphys;
 289         int err;
 290         dmu_buf_t *headdbuf;
 291         dsl_pool_t *dp = ds->ds_dir->dd_pool;
 292         objset_t *mos = dp->dp_meta_objset;

 294         if (ds->ds_snapname[0])
 295                 return (0);
```



```
 296         if (ds->ds_phys->ds_next_snap_obj == 0)
 297                 return (0);

 299         err = dmu_bonus_hold(mos, ds->ds_dir->dd_phys->dd_head_dataset_obj,
 300             FTAG, &headdbuf);
 301         if (err != 0)
 302                 return (err);
 303         headphys = headdbuf->db_data;
 304         err = zap_value_search(dp->dp_meta_objset,
 305             headphys->ds_snapnames_zapobj, ds->ds_object, 0, ds->ds_snapname);
 306         dmu_buf_rele(headdbuf, FTAG);
 307         return (err);
 308 }

 310 int
 311 dsl_dataset_snap_lookup(dsl_dataset_t *ds, const char *name, uint64_t *value)
 312 {
 313         objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
 314         uint64_t snapobj = ds->ds_phys->ds_snapnames_zapobj;
 315         matchtype_t mt;
 316         int err;

 318         if (ds->ds_phys->ds_flags & DS_FLAG_CI_DATASET)
 319                 mt = MT_FIRST;
 320         else
 321                 mt = MT_EXACT;

 323         err = zap_lookup_norm(mos, snapobj, name, 8, 1,
 324             value, mt, NULL, 0, NULL);
 325         if (err == ENOTSUP && mt == MT_FIRST)
 326                 err = zap_lookup(mos, snapobj, name, 8, 1, value);
 327         return (err);
 328 }

 330 int
 331 dsl_dataset_snap_remove(dsl_dataset_t *ds, const char *name, dmu_tx_t *tx)
 332 {
 333         objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
 334         uint64_t snapobj = ds->ds_phys->ds_snapnames_zapobj;
 335         matchtype_t mt;
 336         int err;

 338         dsl_dir_snap_cmtime_update(ds->ds_dir);

 340         if (ds->ds_phys->ds_flags & DS_FLAG_CI_DATASET)
 341                 mt = MT_FIRST;
 342         else
 343                 mt = MT_EXACT;

 345         err = zap_remove_norm(mos, snapobj, name, mt, tx);
 346         if (err == ENOTSUP && mt == MT_FIRST)
 347                 err = zap_remove(mos, snapobj, name, tx);
 348         return (err);
 349 }

 351 int
 352 dsl_dataset_hold_obj(dsl_pool_t *dp, uint64_t dsobj, void *tag,
 353     dsl_dataset_t **dsp)
 354 {
 355         objset_t *mos = dp->dp_meta_objset;
 356         dmu_buf_t *dbuf;
 357         dsl_dataset_t *ds;
 358         int err;
 359         dmu_object_info_t doi;

 361         ASSERT(dsl_pool_config_held(dp));
```

```
363         err = dmu_bonus_hold(mos, dsobj, tag, &dbuf);
364         if (err != 0)
365                 return (err);
366
367         /* Make sure dsobj has the correct object type. */
368         dmu_object_info_from_db(dbuf, &doi);
369         if (doi.doi_type != DMU_OT_DSL_DATASET)
370                 return (SET_ERROR(EINVAL));
371
372         ds = (dsl_dataset_t *)dmu_buf_get_user(dbuf);
264         ds = dmu_buf_get_user(dbuf);
373         if (ds == NULL) {
374                 dsl_dataset_t *winner = NULL;
375
376                 ds = kmem_zalloc(sizeof (dsl_dataset_t), KM_SLEEP);
377                 ds->ds_dbuf = dbuf;
378                 ds->ds_object = dsobj;
271                 ds->ds_phys = dbuf->db_data;
379
380                 mutex_init(&ds->ds_lock, NULL, MUTEX_DEFAULT, NULL);
381                 mutex_init(&ds->ds_opening_lock, NULL, MUTEX_DEFAULT, NULL);
382                 mutex_init(&ds->ds_sendstream_lock, NULL, MUTEX_DEFAULT, NULL);
383                 refcount_create(&ds->ds_longholds);
384
385                 bplist_create(&ds->ds_pending_deadlist);
386                 dsl_deadlist_open(&ds->ds_deadlist,
387                     mos, ds->ds_phys->ds_deadlist_obj);
388
389                 list_create(&ds->ds_sendstreams, sizeof (dmu_sendarg_t),
390                     offsetof(dmu_sendarg_t, dsa_link));
391
392                 if (err == 0) {
393                         err = dsl_dir_hold_obj(dp,
394                             ds->ds_phys->ds_dir_obj, NULL, ds, &ds->ds_dir);
395                 }
396                 if (err != 0) {
397                         mutex_destroy(&ds->ds_lock);
398                         mutex_destroy(&ds->ds_opening_lock);
399                         refcount_destroy(&ds->ds_longholds);
400                         bplist_destroy(&ds->ds_pending_deadlist);
401                         dsl_deadlist_close(&ds->ds_deadlist);
402                         kmem_free(ds, sizeof (dsl_dataset_t));
403                         dmu_buf_rele(dbuf, tag);
404                         return (err);
405                 }
406
407                 if (!dsl_dataset_is_snapshot(ds)) {
408                         ds->ds_snapname[0] = '\0';
409                         if (ds->ds_phys->ds_prev_snap_obj != 0) {
410                                 err = dsl_dataset_hold_obj(dp,
411                                     ds->ds_phys->ds_prev_snap_obj,
412                                     ds, &ds->ds_prev);
413                         }
414                 } else {
415                         if (zfs_flags & ZFS_DEBUG_SNAPNAMES)
416                                 err = dsl_dataset_get_snapname(ds);
417                         if (err == 0 && ds->ds_phys->ds_userrefs_obj != 0) {
418                                 err = zap_count(
419                                     ds->ds_dir->dd_pool->dp_meta_objset,
420                                     ds->ds_phys->ds_userrefs_obj,
421                                     &ds->ds_userrefs);
422                         }
423                 }
424
425                 if (err == 0 && !dsl_dataset_is_snapshot(ds)) {
```



```
426                         err = dsl_prop_get_int_ds(ds,
427                             zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
428                             &ds->ds_reserved);
429                         if (err == 0) {
430                                 err = dsl_prop_get_int_ds(ds,
431                                     zfs_prop_to_name(ZFS_PROP_REFQUOTA),
432                                     &ds->ds_quota);
433                         }
434                 } else {
435                         ds->ds_reserved = ds->ds_quota = 0;
436                 }
437
438                 dmu_buf_init_user(&ds->db_evict, dsl_dataset_evict);
439                 if (err == 0)
440                         winner = (dsl_dataset_t *)
441                             dmu_buf_set_user_ie(dbuf, &ds->db_evict);
442
443                 if (err || winner) {
331                 if (err != 0 || (winner = dmu_buf_set_user_ie(dbuf, ds,
332                     &ds->ds_phys, dsl_dataset_evict)) != NULL) {
444                         bplist_destroy(&ds->ds_pending_deadlist);
445                         dsl_deadlist_close(&ds->ds_deadlist);
446                         if (ds->ds_prev)
447                                 dsl_dataset_rele(ds->ds_prev, ds);
448                         dsl_dir_rele(ds->ds_dir, ds);
449                         mutex_destroy(&ds->ds_lock);
450                         mutex_destroy(&ds->ds_opening_lock);
451                         refcount_destroy(&ds->ds_longholds);
452                         kmem_free(ds, sizeof (dsl_dataset_t));
453                         if (err != 0) {
454                                 dmu_buf_rele(dbuf, tag);
455                                 return (err);
456                         }
457                         ds = winner;
458                 } else {
459                         ds->ds_fsid_guid =
460                             unique_insert(ds->ds_phys->ds_fsid_guid);
461                 }
462         }
463         ASSERT3P(ds->ds_dbuf, ==, dbuf);
464         ASSERT3P(ds->ds_phys, ==, dbuf->db_data);
465         ASSERT(ds->ds_phys->ds_prev_snap_obj != 0 ||
466             spa_version(dp->dp_spa) < SPA_VERSION_ORIGIN ||
467             dp->dp_origin_snap == NULL || ds == dp->dp_origin_snap);
468         *dsp = ds;
469         return (0);
470 }
_____unchanged_portion_omitted_

636 void
637 dsl_dataset_disown(dsl_dataset_t *ds, void *tag)
638 {
639         ASSERT(ds->ds_owner == tag && ds->ds_dbuf != NULL);
640
641         mutex_enter(&ds->ds_lock);
642         ds->ds_owner = NULL;
643         mutex_exit(&ds->ds_lock);
644         dsl_dataset_long_rele(ds, tag);
645         if (ds->ds_dbuf != NULL)
646                 dsl_dataset_rele(ds, tag);
647         else
648                 dsl_dataset_evict_impl(ds, B_FALSE);
537                 dsl_dataset_evict(NULL, ds);
649 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   35255 Thu Apr 25 16:14:58 2013
new/usr/src/uts/common/fs/zfs/dsl_dir.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
```

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

#include <sys/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/spa.h>
#include <sys/metaslab.h>
#include <sys/zap.h>
#include <sys/zio.h>
#include <sys/arc.h>
#include <sys/sunddi.h>
#include "zfs_namecheck.h"

static uint64_t dsl_dir_space_towrite(dsl_dir_t *dd);

/* ARGSUSED */
static void
dsl_dir_evict(dmu_buf_user_t *dbu)
dsl_dir_evict(dmu_buf_t *db, void *arg)
{
        dsl_dir_t *dd = (dsl_dir_t *)dbu;
        dsl_dir_t *dd = arg;
        dsl_pool_t *dp = dd->dd_pool;
        int t;

        dd->dd_dbuf = NULL;

#endif /* ! codereview */
        for (t = 0; t < TXG_SIZE; t++) {
                ASSERT(!txg_list_member(&dp->dp_dirty_dirs, dd, t));
                ASSERT(dd->dd_tempreserved[t] == 0);
```



```
                ASSERT(dd->dd_space_towrite[t] == 0);
        }

        if (dd->dd_parent)
                dsl_dir_rele(dd->dd_parent, dd);

        spa_close(dd->dd_pool->dp_spa, dd);

        /*
         * The props callback list should have been cleaned up by
         * objset_evict().
         */
        list_destroy(&dd->dd_prop_cbs);
        mutex_destroy(&dd->dd_lock);
        kmem_free(dd, sizeof (dsl_dir_t));
}

int
dsl_dir_hold_obj(dsl_pool_t *dp, uint64_t ddobj,
    const char *tail, void *tag, dsl_dir_t **ddp)
{
        dmu_buf_t *dbuf;
        dsl_dir_t *dd;
        int err;

        ASSERT(dsl_pool_config_held(dp));

        err = dmu_bonus_hold(dp->dp_meta_objset, ddobj, tag, &dbuf);
        if (err != 0)
                return (err);
        dd = (dsl_dir_t *)dmu_buf_get_user(dbuf);
        dd = dmu_buf_get_user(dbuf);
#ifdef ZFS_DEBUG
        {
                dmu_object_info_t doi;
                dmu_object_info_from_db(dbuf, &doi);
                ASSERT3U(doi.doi_type, ==, DMU_OT_DSL_DIR);
                ASSERT3U(doi.doi_bonus_size, >=, sizeof (dsl_dir_phys_t));
        }
#endif
        if (dd == NULL) {
                dsl_dir_t *winner;

                dd = kmem_zalloc(sizeof (dsl_dir_t), KM_SLEEP);
                dd->dd_object = ddobj;
                dd->dd_dbuf = dbuf;
                dd->dd_pool = dp;
                dd->dd_phys = dbuf->db_data;
                mutex_init(&dd->dd_lock, NULL, MUTEX_DEFAULT, NULL);

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

                dsl_dir_snap_cmtime_update(dd);

                if (dd->dd_phys->dd_parent_obj) {
                        err = dsl_dir_hold_obj(dp, dd->dd_phys->dd_parent_obj,
                            NULL, dd, &dd->dd_parent);
                        if (err != 0)
                                goto errout;
                        if (tail) {
#ifdef ZFS_DEBUG
                                uint64_t foundobj;

                                err = zap_lookup(dp->dp_meta_objset,
                                    dd->dd_parent->dd_phys->dd_child_dir_zapobj,
```

```
122                                     tail, sizeof (foundobj), 1, &foundobj);
123                                 ASSERT(err || foundobj == ddobj);
124 #endif
125                                 (void) strcpy(dd->dd_myname, tail);
126                         } else {
127                                 err = zap_value_search(dp->dp_meta_objset,
128                                     dd->dd_parent->dd_phys->dd_child_dir_zapobj,
129                                     ddobj, 0, dd->dd_myname);
130                         }
131                         if (err != 0)
132                                 goto errout;
133                 } else {
134                         (void) strcpy(dd->dd_myname, spa_name(dp->dp_spa));
135                 }

137                 if (dsl_dir_is_clone(dd)) {
138                         dmu_buf_t *origin_bonus;
139                         dsl_dataset_phys_t *origin_phys;

141                         /*
142                          * We can't open the origin dataset, because
143                          * that would require opening this dsl_dir.
144                          * Just look at its phys directly instead.
145                          */
146                         err = dmu_bonus_hold(dp->dp_meta_objset,
147                             dd->dd_phys->dd_origin_obj, FTAG, &origin_bonus);
148                         if (err != 0)
149                                 goto errout;
150                         origin_phys = origin_bonus->db_data;
151                         dd->dd_origin_txg =
152                             origin_phys->ds_creation_txg;
153                         dmu_buf_rele(origin_bonus, FTAG);
154                 }

156                 dmu_buf_init_user(&dd->db_evict, dsl_dir_evict);
157                 winner = (dsl_dir_t *)dmu_buf_set_user_ie(dbuf, &dd->db_evict);
121                 winner = dmu_buf_set_user_ie(dbuf, dd, &dd->dd_phys,
122                     dsl_dir_evict);
158                 if (winner) {
159                         if (dd->dd_parent)
160                                 dsl_dir_rele(dd->dd_parent, dd);
161                         mutex_destroy(&dd->dd_lock);
162                         kmem_free(dd, sizeof (dsl_dir_t));
163                         dd = winner;
164                 } else {
165                         spa_open_ref(dp->dp_spa, dd);
166                 }
167         }

169         /*
170          * The dsl_dir_t has both open-to-close and instantiate-to-evict
171          * holds on the spa.  We need the open-to-close holds because
172          * otherwise the spa_refcnt wouldn't change when we open a
173          * dir which the spa also has open, so we could incorrectly
174          * think it was OK to unload/export/destroy the pool.  We need
175          * the instantiate-to-evict hold because the dsl_dir_t has a
176          * pointer to the dd_pool, which has a pointer to the spa_t.
177          */
178         spa_open_ref(dp->dp_spa, tag);
179         ASSERT3P(dd->dd_pool, ==, dp);
180         ASSERT3U(dd->dd_object, ==, ddobj);
181         ASSERT3P(dd->dd_dbuf, ==, dbuf);
182         *ddp = dd;
183         return (0);

185 errout:
```



```
186         if (dd->dd_parent)
187                 dsl_dir_rele(dd->dd_parent, dd);
188         mutex_destroy(&dd->dd_lock);
189         kmem_free(dd, sizeof (dsl_dir_t));
190         dmu_buf_rele(dbuf, tag);
191         return (err);
192 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   29181 Thu Apr 25 16:14:58 2013
new/usr/src/uts/common/fs/zfs/dsl_prop.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_

 159 int
 160 dsl_prop_get_ds(dsl_dataset_t *ds, const char *propname,
 161     int intsz, int numints, void *buf, char *setpoint)
 162 {
 163         zfs_prop_t prop = zfs_name_to_prop(propname);
 164         boolean_t inheritable;
 165         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));
 170         snapshot = (DS_HAS_PHYS(ds) && dsl_dataset_is_snapshot(ds));
 171         zapobj = (DS_HAS_PHYS(ds) ? ds->ds_phys->ds_props_obj : 0);
 170         snapshot = (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds));
 171         zapobj = (ds->ds_phys == NULL ? 0 : ds->ds_phys->ds_props_obj);

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

 177                 ASSERT(snapshot);

 179                 /* Check for a local value. */
 180                 err = zap_lookup(mos, zapobj, propname, intsz, numints, buf);
 181                 if (err != ENOENT) {
 182                         if (setpoint != NULL && err == 0)
 183                                 dsl_dataset_name(ds, setpoint);
 184                         return (err);
 185                 }

 187                 /*
 188                  * Skip the check for a received value if there is an explicit
 189                  * inheritance entry.
 190                  */
 191                 if (inheritable) {
 192                         char *inheritstr = kmem_asprintf("%s%s", propname,
 193                             ZPROP_INHERIT_SUFFIX);
 194                         err = zap_contains(mos, zapobj, inheritstr);
 195                         strfree(inheritstr);
 196                         if (err != 0 && err != ENOENT)
 197                                 return (err);
 198                 }

 200                 if (err == ENOENT) {
 201                         /* Check for a received value. */
 202                         char *recvdstr = kmem_asprintf("%s%s", propname,
 203                             ZPROP_RECVD_SUFFIX);
 204                         err = zap_lookup(mos, zapobj, recvdstr,
 205                             intsz, numints, buf);
 206                         strfree(recvdstr);
 207                         if (err != ENOENT) {
 208                                 if (setpoint != NULL && err == 0)
 209                                         (void) strcpy(setpoint,
 210                                             ZPROP_SOURCE_VAL_RECVD);
 211                                 return (err);
 212                         }
 213                 }
```



```
 214         }

 216         return (dsl_prop_get_dd(ds->ds_dir, propname,
 217             intsz, numints, buf, setpoint, snapshot));
 218 }
______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_HAS_PHYS(ds) && dsl_dataset_is_snapshot(ds)) {
 546         if (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds)) {
 547                 ASSERT(version >= SPA_VERSION_SNAP_PROPS);
 548                 if (ds->ds_phys->ds_props_obj == 0) {
 549                         dmu_buf_will_dirty(ds->ds_dbuf, tx);
 550                         ds->ds_phys->ds_props_obj =
 551                             zap_create(mos,
 552                             DMU_OT_DSL_PROPS, DMU_OT_NONE, 0, tx);
 553                 }
 554                 zapobj = ds->ds_phys->ds_props_obj;
 555         } else {
 556                 zapobj = ds->ds_dir->dd_phys->dd_props_zapobj;
 557         }

 559         if (version < SPA_VERSION_RECVD_PROPS) {
 560                 zfs_prop_t prop = zfs_name_to_prop(propname);
 561                 if (prop == ZFS_PROP_QUOTA || prop == ZFS_PROP_RESERVATION)
 562                         return;

 564                 if (source & ZPROP_SRC_NONE)
 565                         source = ZPROP_SRC_NONE;
 566                 else if (source & ZPROP_SRC_RECEIVED)
 567                         source = ZPROP_SRC_LOCAL;
 568         }

 570         inheritstr = kmem_asprintf("%s%s", propname, ZPROP_INHERIT_SUFFIX);
 571         recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

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

```
586                 /*
587                  * remove propname$inherit
588                  * set propname -> value
589                  */
590                 err = zap_remove(mos, zapobj, inheritstr, tx);
591                 ASSERT(err == 0 || err == ENOENT);
592                 VERIFY0(zap_update(mos, zapobj, propname,
593                     intsz, numints, value, tx));
594                 break;
595         case ZPROP_SRC_INHERITED:
596                 /*
597                  * explicitly inherit
598                  * - remove propname
599                  * - set propname$inherit
600                  */
601                 err = zap_remove(mos, zapobj, propname, tx);
602                 ASSERT(err == 0 || err == ENOENT);
603                 if (version >= SPA_VERSION_RECVD_PROPS &&
604                     dsl_prop_get_int_ds(ds, ZPROP_HAS_RECVD, &dummy) == 0) {
605                         dummy = 0;
606                         VERIFY0(zap_update(mos, zapobj, inheritstr,
607                             8, 1, &dummy, tx));
608                 }
609                 break;
610         case ZPROP_SRC_RECEIVED:
611                 /*
612                  * set propname$recvd -> value
613                  */
614                 err = zap_update(mos, zapobj, recvdstr,
615                     intsz, numints, value, tx);
616                 ASSERT(err == 0);
617                 break;
618         case (ZPROP_SRC_NONE | ZPROP_SRC_LOCAL | ZPROP_SRC_RECEIVED):
619                 /*
620                  * clear local and received settings
621                  * - remove propname
622                  * - remove propname$inherit
623                  * - remove propname$recvd
624                  */
625                 err = zap_remove(mos, zapobj, propname, tx);
626                 ASSERT(err == 0 || err == ENOENT);
627                 err = zap_remove(mos, zapobj, inheritstr, tx);
628                 ASSERT(err == 0 || err == ENOENT);
629                 /* FALLTHRU */
630         case (ZPROP_SRC_NONE | ZPROP_SRC_RECEIVED):
631                 /*
632                  * remove propname$recvd
633                  */
634                 err = zap_remove(mos, zapobj, recvdstr, tx);
635                 ASSERT(err == 0 || err == ENOENT);
636                 break;
637         default:
638                 cmn_err(CE_PANIC, "unexpected property source: %d", source);
639         }

641         strfree(inheritstr);
642         strfree(recvdstr);

644         if (isint) {
645                 VERIFY0(dsl_prop_get_int_ds(ds, propname, &intval));

647                 if (DS_HAS_PHYS(ds) && dsl_dataset_is_snapshot(ds)) {
647                 if (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds)) {
648                         dsl_prop_cb_record_t *cbr;
649                         /*
650                          * It's a snapshot; nothing can inherit this
```



```
651                          * property, so just look for callbacks on this
652                          * ds here.
653                          */
654                         mutex_enter(&ds->ds_dir->dd_lock);
655                         for (cbr = list_head(&ds->ds_dir->dd_prop_cbs); cbr;
656                             cbr = list_next(&ds->ds_dir->dd_prop_cbs, cbr)) {
657                                 if (cbr->cbr_ds == ds &&
658                                     strcmp(cbr->cbr_propname, propname) == 0)
659                                         cbr->cbr_func(cbr->cbr_arg, intval);
660                         }
661                         mutex_exit(&ds->ds_dir->dd_lock);
662                 } else {
663                         dsl_prop_changed_notify(ds->ds_dir->dd_pool,
664                             ds->ds_dir->dd_object, propname, intval, TRUE);
665                 }

667                 (void) snprintf(valbuf, sizeof (valbuf),
668                     "%lld", (longlong_t)intval);
669                 valstr = valbuf;
670         } else {
671                 if (source == ZPROP_SRC_LOCAL) {
672                         valstr = value;
673                 } else {
674                         tbuf = kmem_alloc(ZAP_MAXVALUELEN, KM_SLEEP);
675                         if (dsl_prop_get_ds(ds, propname, 1,
676                             ZAP_MAXVALUELEN, tbuf, NULL) == 0)
677                                 valstr = tbuf;
678                 }
679         }

681         spa_history_log_internal_ds(ds, (source == ZPROP_SRC_NONE ||
682             source == ZPROP_SRC_INHERITED) ? "inherit" : "set", tx,
683             "%s=%s", propname, (valstr == NULL ? "" : valstr));

685         if (tbuf != NULL)
686                 kmem_free(tbuf, ZAP_MAXVALUELEN);
687 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   51823 Thu Apr 25 16:14:58 2013
new/usr/src/uts/common/fs/zfs/sa.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
```

_____unchanged_portion_omitted_

```
1300 /*ARGSUSED*/
1301 void
1302 sa_evict(dmu_buf_user_t *dbu)
1302 sa_evict(dmu_buf_t *db, void *sap)
1303 {
1304         panic("evicting sa dbuf\n");
1304         panic("evicting sa dbuf %p\n", (void *)db);
1305 }
```

_____unchanged_portion_omitted_

```
1340 void
1341 sa_handle_destroy(sa_handle_t *hdl)
1342 {
1343         dmu_buf_t *db = hdl->sa_bonus;

1345 #endif /* ! codereview */
1346         mutex_enter(&hdl->sa_lock);
1347         (void) dmu_buf_remove_user(db, &hdl->db_evict);
1343         (void) dmu_buf_update_user((dmu_buf_t *)hdl->sa_bonus, hdl,
1344             NULL, NULL, NULL);

1349         if (hdl->sa_bonus_tab) {
1350                 sa_idx_tab_rele(hdl->sa_os, hdl->sa_bonus_tab);
1351                 hdl->sa_bonus_tab = NULL;
1352         }
1353         if (hdl->sa_spill_tab) {
1354                 sa_idx_tab_rele(hdl->sa_os, hdl->sa_spill_tab);
1355                 hdl->sa_spill_tab = NULL;
1356         }

1358         dmu_buf_rele(hdl->sa_bonus, NULL);

1360         if (hdl->sa_spill)
1361                 dmu_buf_rele((dmu_buf_t *)hdl->sa_spill, NULL);
1362         mutex_exit(&hdl->sa_lock);

1364         kmem_cache_free(sa_cache, hdl);
1365 }

1367 int
1368 sa_handle_get_from_db(objset_t *os, dmu_buf_t *db, void *userp,
1369     sa_handle_type_t hdl_type, sa_handle_t **handlepp)
1370 {
1371         int error = 0;
1372         dmu_object_info_t doi;
1373         sa_handle_t *handle = NULL;
1370         sa_handle_t *handle;

1375 #ifdef ZFS_DEBUG
1376         dmu_object_info_from_db(db, &doi);
1377         ASSERT(doi.doi_bonus_type == DMU_OT_SA ||
1378             doi.doi_bonus_type == DMU_OT_ZNODE);
1379 #endif
1380         /* find handle, if it exists */
1381         /* if one doesn't exist then create a new one, and initialize it */

1383         if (hdl_type == SA_HDL_SHARED)
```



```
1384                 handle = (sa_handle_t *)dmu_buf_get_user(db);

1380         handle = (hdl_type == SA_HDL_SHARED) ? dmu_buf_get_user(db) : NULL;
1386         if (handle == NULL) {
1387                 sa_handle_t *winner = NULL;

1389                 bzero(&handle->db_evict, sizeof(dmu_buf_user_t));
1382                 sa_handle_t *newhandle;
1390                 handle = kmem_cache_alloc(sa_cache, KM_SLEEP);
1391                 handle->sa_userp = userp;
1392                 handle->sa_bonus = db;
1393                 handle->sa_os = os;
1394                 handle->sa_spill = NULL;

1396                 error = sa_build_index(handle, SA_BONUS);
1397                 if (hdl_type == SA_HDL_SHARED) {
1398                         dmu_buf_init_user(&handle->db_evict, sa_evict);
1399                         winner = (sa_handle_t *)
1400                             dmu_buf_set_user_ie(db, &handle->db_evict);
1401                 }
1402                 if (winner != NULL) {
1390                 newhandle = (hdl_type == SA_HDL_SHARED) ?
1391                     dmu_buf_set_user_ie(db, handle,
1392                     NULL, sa_evict) : NULL;

1394                 if (newhandle != NULL) {
1403                         kmem_cache_free(sa_cache, handle);
1404                         handle = winner;
1396                         handle = newhandle;
1405                 }
1406         }
1407         *handlepp = handle;

1409         return (error);
1410 }
```

_____unchanged_portion_omitted_

```
1914 void
1915 sa_update_user(sa_handle_t *newhdl, sa_handle_t *oldhdl)
1916 {
1917         dmu_buf_user_t *new_user = &newhdl->db_evict;
1918         dmu_buf_user_t *old_user = &oldhdl->db_evict;

1920         dmu_buf_init_user(new_user, sa_evict);
1921         VERIFY(dmu_buf_replace_user(newhdl->sa_bonus, old_user,
1922             new_user) == old_user);
1909         (void) dmu_buf_update_user((dmu_buf_t *)newhdl->sa_bonus,
1910             oldhdl, newhdl, NULL, sa_evict);
1923         oldhdl->sa_bonus = NULL;
1924 }
```

_____unchanged_portion_omitted_

```
**********************************************************
    9996 Thu Apr 25 16:14:58 2013
new/usr/src/uts/common/fs/zfs/sys/dbuf.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______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)
         */
        blkptr_t *db_blkptr;

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

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

        /*
         * Current state of the buffer
         */
        dbuf_states_t db_state;
```



```
        /*
         * Refcount accessed by dmu_buf_{hold,rele}.
         * If nonzero, the buffer can't be destroyed.
         * Protected by db_mtx.
         */
        refcount_t db_holds;

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

        kcondvar_t db_changed;
        dbuf_dirty_record_t *db_data_pending;

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

        /*
         * Our link on the owner dnodes's dn_dbufs list.
         * Protected by its dn_dbufs_mtx.
         */
        list_node_t db_link;

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

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

        uint8_t db_immediate_evict;
        uint8_t db_freed_in_flight;

        uint8_t db_dirtycnt;
} dmu_buf_impl_t;
______unchanged_portion_omitted_


uint64_t dbuf_whichblock(struct dnode *di, uint64_t offset);

dmu_buf_impl_t *dbuf_create_tlib(struct dnode *dn, char *data);
void dbuf_create_bonus(struct dnode *dn);
int dbuf_spill_set_blksz(dmu_buf_t *db, uint64_t blksz, dmu_tx_t *tx);
void dbuf_spill_hold(struct dnode *dn, dmu_buf_impl_t **dbp, void *tag);

void dbuf_rm_spill(struct dnode *dn, dmu_tx_t *tx);

dmu_buf_impl_t *dbuf_hold(struct dnode *dn, uint64_t blkid, void *tag);
dmu_buf_impl_t *dbuf_hold_level(struct dnode *dn, int level, uint64_t blkid,
    void *tag);
int dbuf_hold_impl(struct dnode *dn, uint8_t level, uint64_t blkid, int create,
    void *tag, dmu_buf_impl_t **dbp);

void dbuf_prefetch(struct dnode *dn, uint64_t blkid);

void dbuf_add_ref(dmu_buf_impl_t *db, void *tag);
uint64_t dbuf_refcount(dmu_buf_impl_t *db);

void dbuf_rele(dmu_buf_impl_t *db, void *tag);
void dbuf_rele_and_unlock(dmu_buf_impl_t *db, void *tag);

dmu_buf_impl_t *dbuf_find(struct dnode *dn, uint8_t level, uint64_t blkid);

int dbuf_read(dmu_buf_impl_t *db, zio_t *zio, uint32_t flags);
void dbuf_will_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
```

```
266 void dbuf_fill_done(dmu_buf_impl_t *db, dmu_tx_t *tx);
267 void dmu_buf_will_not_fill(dmu_buf_t *db, dmu_tx_t *tx);
268 void dmu_buf_will_fill(dmu_buf_t *db, dmu_tx_t *tx);
269 void dmu_buf_fill_done(dmu_buf_t *db, dmu_tx_t *tx);
270 void dbuf_assign_arcbuf(dmu_buf_impl_t *db, arc_buf_t *buf, dmu_tx_t *tx);
271 dbuf_dirty_record_t *dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
272 arc_buf_t *dbuf_loan_arcbuf(dmu_buf_impl_t *db);

274 void dbuf_clear(dmu_buf_impl_t *db, list_t *evict_list_p);
275 void dbuf_evict(dmu_buf_impl_t *db, list_t *evict_list_p);
276 void dbuf_clear(dmu_buf_impl_t *db);
277 void dbuf_evict(dmu_buf_impl_t *db);

277 void dbuf_setdirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
278 void dbuf_unoverride(dbuf_dirty_record_t *dr);
279 void dbuf_sync_list(list_t *list, dmu_tx_t *tx);
280 void dbuf_release_bp(dmu_buf_impl_t *db);

282 void dbuf_free_range(struct dnode *dn, uint64_t start, uint64_t end,
283     struct dmu_tx *);

285 void dbuf_new_size(dmu_buf_impl_t *db, int size, dmu_tx_t *tx);

287 #define DB_DNODE(_db)           ((_db)->db_dnode_handle->dnh_dnode)
288 #define DB_DNODE_LOCK(_db)      ((_db)->db_dnode_handle->dnh_zrlock)
289 #define DB_DNODE_ENTER(_db)     (zrl_add(&DB_DNODE_LOCK(_db)))
290 #define DB_DNODE_EXIT(_db)      (zrl_remove(&DB_DNODE_LOCK(_db)))
291 #define DB_DNODE_HELD(_db)      (!zrl_is_zero(&DB_DNODE_LOCK(_db)))
292 #define DB_GET_SPA(_spa_p, _db) {               \
293         dnode_t *__dn;                         \
294         DB_DNODE_ENTER(_db);                   \
295         __dn = DB_DNODE(_db);                  \
296         *(_spa_p) = __dn->dn_objset->os_spa;    \
297         DB_DNODE_EXIT(_db);                    \
298 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   30813 Thu Apr 25 16:14:59 2013
new/usr/src/uts/common/fs/zfs/sys/dmu.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
```

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

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. 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>
#endif /* ! codereview */
#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;
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;

#define DMU_OT_NEWTYPE 0x80
#define DMU_OT_METADATA 0x40
#define DMU_OT_BYTESWAP_MASK 0x3f

/*
 * Defines a uint8_t object type. Object types specify if the data
 * in the object is metadata (boolean) and how to byteswap the data
 * (dmu_object_byteswap_t).
 */
#define DMU_OT(byteswap, metadata) \
        (DMU_OT_NEWTYPE | \
        ((metadata) ? DMU_OT_METADATA : 0) | \
        ((byteswap) & DMU_OT_BYTESWAP_MASK))

#define DMU_OT_IS_VALID(ot) (((ot) & DMU_OT_NEWTYPE) ? \
        ((ot) & DMU_OT_BYTESWAP_MASK) < DMU_BSWAP_NUMFUNCS : \
        (ot) < DMU_OT_NUMTYPES)

#define DMU_OT_IS_METADATA(ot) (((ot) & DMU_OT_NEWTYPE) ? \
        ((ot) & DMU_OT_METADATA) : \
        dmu_ot[(ot)].ot_metadata)

#define DMU_OT_BYTESWAP(ot) (((ot) & DMU_OT_NEWTYPE) ? \
        ((ot) & DMU_OT_BYTESWAP_MASK) : \
        dmu_ot[(ot)].ot_byteswap)
```

```
127 typedef enum dmu_object_type {
128         DMU_OT_NONE,
129         /* general: */
130         DMU_OT_OBJECT_DIRECTORY,        /* ZAP */
131         DMU_OT_OBJECT_ARRAY,            /* UINT64 */
132         DMU_OT_PACKED_NVLIST,           /* UINT8 (XDR by nvlist_pack/unpack) */
133         DMU_OT_PACKED_NVLIST_SIZE,      /* UINT64 */
134         DMU_OT_BPOBJ,                   /* UINT64 */
135         DMU_OT_BPOBJ_HDR,               /* UINT64 */
136         /* spa: */
137         DMU_OT_SPACE_MAP_HEADER,        /* UINT64 */
138         DMU_OT_SPACE_MAP,               /* UINT64 */
139         /* zil: */
140         DMU_OT_INTENT_LOG,              /* UINT64 */
141         /* dmu: */
142         DMU_OT_DNODE,                   /* DNODE */
143         DMU_OT_OBJSET,                  /* OBJSET */
144         /* dsl: */
145         DMU_OT_DSL_DIR,                 /* UINT64 */
146         DMU_OT_DSL_DIR_CHILD_MAP,       /* ZAP */
147         DMU_OT_DSL_DS_SNAP_MAP,         /* ZAP */
148         DMU_OT_DSL_PROPS,               /* ZAP */
149         DMU_OT_DSL_DATASET,             /* UINT64 */
150         /* zpl: */
151         DMU_OT_ZNODE,                   /* ZNODE */
152         DMU_OT_OLDACL,                  /* Old ACL */
153         DMU_OT_PLAIN_FILE_CONTENTS,     /* UINT8 */
154         DMU_OT_DIRECTORY_CONTENTS,      /* ZAP */
155         DMU_OT_MASTER_NODE,             /* ZAP */
156         DMU_OT_UNLINKED_SET,            /* ZAP */
157         /* zvol: */
158         DMU_OT_ZVOL,                    /* UINT8 */
159         DMU_OT_ZVOL_PROP,               /* ZAP */
160         /* other; for testing only! */
161         DMU_OT_PLAIN_OTHER,             /* UINT8 */
162         DMU_OT_UINT64_OTHER,            /* UINT64 */
163         DMU_OT_ZAP_OTHER,               /* ZAP */
164         /* new object types: */
165         DMU_OT_ERROR_LOG,               /* ZAP */
166         DMU_OT_SPA_HISTORY,             /* UINT8 */
167         DMU_OT_SPA_HISTORY_OFFSETS,     /* spa_his_phys_t */
168         DMU_OT_POOL_PROPS,              /* ZAP */
169         DMU_OT_DSL_PERMS,               /* ZAP */
170         DMU_OT_ACL,                     /* ACL */
171         DMU_OT_SYSACL,                  /* SYSACL */
172         DMU_OT_FUID,                    /* FUID table (Packed NVLIST UINT8) */
173         DMU_OT_FUID_SIZE,               /* FUID table size UINT64 */
174         DMU_OT_NEXT_CLONES,             /* ZAP */
175         DMU_OT_SCAN_QUEUE,              /* ZAP */
176         DMU_OT_USERGROUP_USED,          /* ZAP */
177         DMU_OT_USERGROUP_QUOTA,         /* ZAP */
178         DMU_OT_USERREFS,                /* ZAP */
179         DMU_OT_DDT_ZAP,                 /* ZAP */
180         DMU_OT_DDT_STATS,               /* ZAP */
181         DMU_OT_SA,                      /* System attr */
182         DMU_OT_SA_MASTER_NODE,          /* ZAP */
183         DMU_OT_SA_ATTR_REGISTRATION,    /* ZAP */
184         DMU_OT_SA_ATTR_LAYOUTS,         /* ZAP */
185         DMU_OT_SCAN_XLATE,              /* ZAP */
186         DMU_OT_DEDUP,                   /* fake dedup BP from ddt_bp_create() */
187         DMU_OT_DEADLIST,                /* ZAP */
188         DMU_OT_DEADLIST_HDR,            /* UINT64 */
189         DMU_OT_DSL_CLONES,              /* ZAP */
190         DMU_OT_BPOBJ_SUBOBJ,            /* UINT64 */
191         /*
```



```
192          * Do not allocate new object types here. Doing so makes the on-disk
193          * format incompatible with any other format that uses the same object
194          * type number.
195          *
196          * When creating an object which does not have one of the above types
197          * use the DMU_OTN_* type with the correct byteswap and metadata
198          * values.
199          *
200          * The DMU_OTN_* types do not have entries in the dmu_ot table,
201          * use the DMU_OT_IS_METDATA() and DMU_OT_BYTESWAP() macros instead
202          * of indexing into dmu_ot directly (this works for both DMU_OT_* types
203          * and DMU_OTN_* types).
204          */
205         DMU_OT_NUMTYPES,
206
207         /*
208          * Names for valid types declared with DMU_OT().
209          */
210         DMU_OTN_UINT8_DATA = DMU_OT(DMU_BSWAP_UINT8, B_FALSE),
211         DMU_OTN_UINT8_METADATA = DMU_OT(DMU_BSWAP_UINT8, B_TRUE),
212         DMU_OTN_UINT16_DATA = DMU_OT(DMU_BSWAP_UINT16, B_FALSE),
213         DMU_OTN_UINT16_METADATA = DMU_OT(DMU_BSWAP_UINT16, B_TRUE),
214         DMU_OTN_UINT32_DATA = DMU_OT(DMU_BSWAP_UINT32, B_FALSE),
215         DMU_OTN_UINT32_METADATA = DMU_OT(DMU_BSWAP_UINT32, B_TRUE),
216         DMU_OTN_UINT64_DATA = DMU_OT(DMU_BSWAP_UINT64, B_FALSE),
217         DMU_OTN_UINT64_METADATA = DMU_OT(DMU_BSWAP_UINT64, B_TRUE),
218         DMU_OTN_ZAP_DATA = DMU_OT(DMU_BSWAP_ZAP, B_FALSE),
219         DMU_OTN_ZAP_METADATA = DMU_OT(DMU_BSWAP_ZAP, B_TRUE),
220 } dmu_object_type_t;

222 typedef enum txg_how {
223         TXG_WAIT = 1,
224         TXG_NOWAIT,
225 } txg_how_t;

227 void byteswap_uint64_array(void *buf, size_t size);
228 void byteswap_uint32_array(void *buf, size_t size);
229 void byteswap_uint16_array(void *buf, size_t size);
230 void byteswap_uint8_array(void *buf, size_t size);
231 void zap_byteswap(void *buf, size_t size);
232 void zfs_oldacl_byteswap(void *buf, size_t size);
233 void zfs_acl_byteswap(void *buf, size_t size);
234 void zfs_znode_byteswap(void *buf, size_t size);

236 #define DS_FIND_SNAPSHOTS       (1<<0)
237 #define DS_FIND_CHILDREN        (1<<1)

239 /*
240  * The maximum number of bytes that can be accessed as part of one
241  * operation, including metadata.
242  */
243 #define DMU_MAX_ACCESS (10<<20) /* 10MB */
244 #define DMU_MAX_DELETEBLKCNT (20480) /* ~5MB of indirect blocks */

246 #define DMU_USERUSED_OBJECT     (-1ULL)
247 #define DMU_GROUPUSED_OBJECT    (-2ULL)
248 #define DMU_DEADLIST_OBJECT     (-3ULL)

250 /*
251  * artificial blkids for bonus buffer and spill blocks
252  */
253 #define DMU_BONUS_BLKID         (-1ULL)
254 #define DMU_SPILL_BLKID         (-2ULL)
255 /*
256  * Public routines to create, destroy, open, and close objsets.
257  */
```

```
258 int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
259 int dmu_objset_own(const char *name, dmu_objset_type_t type,
260     boolean_t readonly, void *tag, objset_t **osp);
261 void dmu_objset_rele(objset_t *os, void *tag);
262 void dmu_objset_disown(objset_t *os, void *tag);
263 int dmu_objset_open_ds(struct dsl_dataset *ds, objset_t **osp);

265 void dmu_objset_evict_dbufs(objset_t *os);
266 int dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
267     void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg);
268 int dmu_objset_clone(const char *name, const char *origin);
269 int dsl_destroy_snapshots_nvl(struct nvlist *snaps, boolean_t defer,
270     struct nvlist *errlist);
271 int dmu_objset_snapshot_one(const char *fsname, const char *snapname);
272 int dmu_objset_snapshot_tmp(const char *, const char *, int);
273 int dmu_objset_find(char *name, int func(const char *, void *), void *arg,
274     int flags);
275 void dmu_objset_byteswap(void *buf, size_t size);
276 int dsl_dataset_rename_snapshot(const char *fsname,
277     const char *oldsnapname, const char *newsnapname, boolean_t recursive);

279 typedef struct dmu_buf {
280         uint64_t db_object;             /* object that this buffer is part of */
281         uint64_t db_offset;             /* byte offset in this object */
282         uint64_t db_size;               /* size of buffer in bytes */
283         void *db_data;                  /* data in buffer */
284 } dmu_buf_t;

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

286 /*
287  * The names of zap entries in the DIRECTORY_OBJECT of the MOS.
288  */
289 #define DMU_POOL_DIRECTORY_OBJECT       1
290 #define DMU_POOL_CONFIG                 "config"
291 #define DMU_POOL_FEATURES_FOR_WRITE     "features_for_write"
292 #define DMU_POOL_FEATURES_FOR_READ      "features_for_read"
293 #define DMU_POOL_FEATURE_DESCRIPTIONS   "feature_descriptions"
294 #define DMU_POOL_ROOT_DATASET           "root_dataset"
295 #define DMU_POOL_SYNC_BPOBJ             "sync_bplist"
296 #define DMU_POOL_ERRLOG_SCRUB           "errlog_scrub"
297 #define DMU_POOL_ERRLOG_LAST            "errlog_last"
298 #define DMU_POOL_SPARES                 "spares"
299 #define DMU_POOL_DEFLATE                "deflate"
300 #define DMU_POOL_HISTORY                "history"
301 #define DMU_POOL_PROPS                  "pool_props"
302 #define DMU_POOL_L2CACHE                "l2cache"
303 #define DMU_POOL_TMP_USERREFS           "tmp_userrefs"
304 #define DMU_POOL_DDT                    "DDT-%s-%s-%s"
305 #define DMU_POOL_DDT_STATS              "DDT-statistics"
306 #define DMU_POOL_CREATION_VERSION       "creation_version"
307 #define DMU_POOL_SCAN                   "scan"
308 #define DMU_POOL_FREE_BPOBJ             "free_bpobj"
309 #define DMU_POOL_BPTREE_OBJ             "bptree_obj"
310 #define DMU_POOL_EMPTY_BPOBJ            "empty_bpobj"

312 /*
313  * Allocate an object from this objset.  The range of object numbers
314  * available is (0, DN_MAX_OBJECT).  Object 0 is the meta-dnode.
315  *
316  * The transaction must be assigned to a txg.  The newly allocated
317  * object will be "held" in the transaction (ie. you can modify the
318  * newly allocated object in this transaction).
319  *
320  * dmu_object_alloc() chooses an object and returns it in *objectp.
321  *
```



```
322  * dmu_object_claim() allocates a specific object number.  If that
323  * number is already allocated, it fails and returns EEXIST.
324  *
325  * Return 0 on success, or ENOSPC or EEXIST as specified above.
326  */
327 uint64_t dmu_object_alloc(objset_t *os, dmu_object_type_t ot,
328     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
329 int dmu_object_claim(objset_t *os, uint64_t object, dmu_object_type_t ot,
330     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
331 int dmu_object_reclaim(objset_t *os, uint64_t object, dmu_object_type_t ot,
332     int blocksize, dmu_object_type_t bonustype, int bonuslen);

334 /*
335  * Free an object from this objset.
336  *
337  * The object's data will be freed as well (ie. you don't need to call
338  * dmu_free(object, 0, -1, tx)).
339  *
340  * The object need not be held in the transaction.
341  *
342  * If there are any holds on this object's buffers (via dmu_buf_hold()),
343  * or tx holds on the object (via dmu_tx_hold_object()), you can not
344  * free it; it fails and returns EBUSY.
345  *
346  * If the object is not allocated, it fails and returns ENOENT.
347  *
348  * Return 0 on success, or EBUSY or ENOENT as specified above.
349  */
350 int dmu_object_free(objset_t *os, uint64_t object, dmu_tx_t *tx);

352 /*
353  * Find the next allocated or free object.
354  *
355  * The objectp parameter is in-out.  It will be updated to be the next
356  * object which is allocated.  Ignore objects which have not been
357  * modified since txg.
358  *
359  * XXX Can only be called on a objset with no dirty data.
360  *
361  * Returns 0 on success, or ENOENT if there are no more objects.
362  */
363 int dmu_object_next(objset_t *os, uint64_t *objectp,
364     boolean_t hole, uint64_t txg);

366 /*
367  * Set the data blocksize for an object.
368  *
369  * The object cannot have any blocks allcated beyond the first.  If
370  * the first block is allocated already, the new size must be greater
371  * than the current block size.  If these conditions are not met,
372  * ENOTSUP will be returned.
373  *
374  * Returns 0 on success, or EBUSY if there are any holds on the object
375  * contents, or ENOTSUP as described above.
376  */
377 int dmu_object_set_blocksize(objset_t *os, uint64_t object, uint64_t size,
378     int ibs, dmu_tx_t *tx);

380 /*
381  * Set the checksum property on a dnode.  The new checksum algorithm will
382  * apply to all newly written blocks; existing blocks will not be affected.
383  */
384 void dmu_object_set_checksum(objset_t *os, uint64_t object, uint8_t checksum,
385     dmu_tx_t *tx);

387 /*
```

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

 394 /*
 395  * Decide how to write a block: checksum, compression, number of copies, etc.
 396  */
 397 #define WP_NOFILL       0x1
 398 #define WP_DMU_SYNC     0x2
 399 #define WP_SPILL        0x4

 401 void dmu_write_policy(objset_t *os, struct dnode *dn, int level, int wp,
 402     struct zio_prop *zp);
 403 /*
 404  * The bonus data is accessed more or less like a regular buffer.
 405  * You must dmu_bonus_hold() to get the buffer, which will give you a
 406  * dmu_buf_t with db_offset==-1ULL, and db_size = the size of the bonus
 407  * data.  As with any normal buffer, you must call dmu_buf_read() to
 408  * read db_data, dmu_buf_will_dirty() before modifying it, and the
 409  * object must be held in an assigned transaction before calling
 410  * dmu_buf_will_dirty.  You may use dmu_buf_set_user() on the bonus
 411  * buffer as well.  You must release your hold with dmu_buf_rele().
 412  */
 413 int dmu_bonus_hold(objset_t *os, uint64_t object, void *tag, dmu_buf_t **);
 414 int dmu_bonus_max(void);
 415 int dmu_set_bonus(dmu_buf_t *, int, dmu_tx_t *);
 416 int dmu_set_bonustype(dmu_buf_t *, dmu_object_type_t, dmu_tx_t *);
 417 dmu_object_type_t dmu_get_bonustype(dmu_buf_t *);
 418 int dmu_rm_spill(objset_t *, uint64_t, dmu_tx_t *);

 420 /*
 421  * Special spill buffer support used by "SA" framework
 422  */

 424 int dmu_spill_hold_by_bonus(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp);
 425 int dmu_spill_hold_by_dnode(struct dnode *dn, uint32_t flags,
 426     void *tag, dmu_buf_t **dbp);
 427 int dmu_spill_hold_existing(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp);

 429 /*
 430  * Obtain the DMU buffer from the specified object which contains the
 431  * specified offset.  dmu_buf_hold() puts a "hold" on the buffer, so
 432  * that it will remain in memory.  You must release the hold with
 433  * dmu_buf_rele().  You musn't access the dmu_buf_t after releasing your
 434  * hold.  You must have a hold on any dmu_buf_t* you pass to the DMU.
 435  *
 436  * You must call dmu_buf_read, dmu_buf_will_dirty, or dmu_buf_will_fill
 437  * on the returned buffer before reading or writing the buffer's
 438  * db_data.  The comments for those routines describe what particular
 439  * operations are valid after calling them.
 440  *
 441  * The object number must be a valid, allocated object number.
 442  */
 443 int dmu_buf_hold(objset_t *os, uint64_t object, uint64_t offset,
 444     void *tag, dmu_buf_t **, int flags);
 445 void dmu_buf_add_ref(dmu_buf_t *db, void* tag);
 446 void dmu_buf_rele(dmu_buf_t *db, void *tag);
 447 uint64_t dmu_buf_refcount(dmu_buf_t *db);

 449 /*
 450  * dmu_buf_hold_array holds the DMU buffers which contain all bytes in a
 451  * range of an object.  A pointer to an array of dmu_buf_t*'s is
 452  * returned (in *dbpp).
 453  *
```



```
 454  * dmu_buf_rele_array releases the hold on an array of dmu_buf_t*'s, and
 455  * frees the array.  The hold on the array of buffers MUST be released
 456  * with dmu_buf_rele_array.  You can NOT release the hold on each buffer
 457  * individually with dmu_buf_rele.
 458  */
 459 int dmu_buf_hold_array_by_bonus(dmu_buf_t *db, uint64_t offset,
 460     uint64_t length, int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp);
 461 void dmu_buf_rele_array(dmu_buf_t **, int numbufs, void *tag);

 463 struct dmu_buf_user;

 465 typedef void dmu_buf_evict_func_t(struct dmu_buf_user *);

 467 #endif /* ! codereview */
 468 /*
 469  * The DMU buffer user object is used to allow private data to be
 470  * associated with a dbuf for the duration of its lifetime.  This private
 471  * data must include a dmu_buf_user_t as its first object, which is passed
 472  * into the DMU user data API and can be attached to a dbuf.  Clients can
 473  * regain access to their private data structure with a cast.
 474  *
 475  * DMU buffer users can be notified via a callback when their associated
 476  * dbuf has been evicted.  This is typically used to free the user's
 477  * private data.  The eviction callback is executed without the dbuf
 478  * mutex held or any other type of mechanism to guarantee that the
 479  * dbuf is still available.  For this reason, users must assume the dbuf
 480  * has already been freed and not reference the dbuf from the callback
 481  * context.
 482  *
 483  * Users requestion "immediate eviction" are notified as soon as the dbuf
 484  * is only referenced by dirty records (dirties == holds).  Otherwise the
 485  * eviction callback occurs after the last reference to the dbuf is dropped.
 486  *
 487  * Eviction Callback Processing
 488  * ============================
 489  * In any context where a dbuf reference drop may trigger an eviction, an
 490  * eviction queue object must be provided.  This queue must then be
 491  * processed while not holding any dbuf locks.  In this way, the user can
 492  * perform any work needed in their eviction function without fear of
 493  * lock order reversals.
 494  *
 495  * Implementation Note
 496  * ===========================
 497  * Some users will occasionally want to map a structure directly onto the
 498  * backing dbuf.  Using an union with an name alias macro to access these
 499  * overlays reduces the ugliness of code that accesses them.  Initial work on
 500  * user objects involved using a macro that took the user object as an
 501  * argument to access the fields, which resulted in hundreds of lines of
 502  * needless diffs and wasn't any easier to read.
 221  * Returns NULL on success, or the existing user ptr if it's already
 222  * been set.
 223  *
 224  * user_ptr is for use by the user and can be obtained via dmu_buf_get_user().
 225  *
 226  * user_data_ptr_ptr should be NULL, or a pointer to a pointer which
 227  * will be set to db->db_data when you are allowed to access it.  Note
 228  * that db->db_data (the pointer) can change when you do dmu_buf_read(),
 229  * dmu_buf_tryupgrade(), dmu_buf_will_dirty(), or dmu_buf_will_fill().
 230  * *user_data_ptr_ptr will be set to the new value when it changes.
 231  *
 232  * If non-NULL, pageout func will be called when this buffer is being
 233  * excised from the cache, so that you can clean up the data structure
 234  * pointed to by user_ptr.
 235  *
 236  * dmu_evict_user() will call the pageout func for all buffers in a
 237  * objset with a given pageout func.
```

```
 */
typedef struct dmu_buf_user {
        /*
         * This instance's link in the eviction queue.  Set when the buffer
         * has evicted and the callback needs to be called.
void *dmu_buf_set_user(dmu_buf_t *db, void *user_ptr, void *user_data_ptr_ptr,
    dmu_buf_evict_func_t *pageout_func);
/*
 * set_user_ie is the same as set_user, but request immediate eviction
 * when hold count goes to zero.
         */
        list_node_t evict_queue_link;
        /** This instance's eviction function pointer. */
        dmu_buf_evict_func_t *evict_func;
} dmu_buf_user_t;

/*
 * Initialize the given dmu_buf_user_t instance with the eviction function
 * evict_func, to be called when the user is evicted.
 *
 * NOTE: This function should only be called once on a given object.  To
 *       help enforce this, dbu should already be zeroed on entry.
 */
static inline void
dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func)
{
        ASSERT(dbu->evict_func == NULL);
        ASSERT(!list_link_active(&dbu->evict_queue_link));
        dbu->evict_func = evict_func;
}

static inline void
dmu_buf_create_user_evict_list(list_t *evict_list_p)
{
        list_create(evict_list_p, sizeof(dmu_buf_user_t),
            offsetof(dmu_buf_user_t, evict_queue_link));
}

static inline void
dmu_buf_process_user_evicts(list_t *evict_list_p)
{
        dmu_buf_user_t *dbu, *next;

        for (dbu = (dmu_buf_user_t *)list_head(evict_list_p); dbu != NULL;
            dbu = next) {
                next = (dmu_buf_user_t *)list_next(evict_list_p, dbu);
                list_remove(evict_list_p, dbu);
                dbu->evict_func(dbu);
        }
}
void *dmu_buf_set_user_ie(dmu_buf_t *db, void *user_ptr,
    void *user_data_ptr_ptr, dmu_buf_evict_func_t *pageout_func);
void *dmu_buf_update_user(dmu_buf_t *db_fake, void *old_user_ptr,
    void *user_ptr, void *user_data_ptr_ptr,
    dmu_buf_evict_func_t *pageout_func);
void dmu_evict_user(objset_t *os, dmu_buf_evict_func_t *func);

static inline void
dmu_buf_destroy_user_evict_list(list_t *evict_list_p)
{
        dmu_buf_process_user_evicts(evict_list_p);
        list_destroy(evict_list_p);
}

dmu_buf_user_t *dmu_buf_set_user(dmu_buf_t *db, dmu_buf_user_t *user);
dmu_buf_user_t *dmu_buf_set_user_ie(dmu_buf_t *db, dmu_buf_user_t *user);
```



```
dmu_buf_user_t *dmu_buf_replace_user(dmu_buf_t *db,
    dmu_buf_user_t *old_user, dmu_buf_user_t *new_user);
dmu_buf_user_t *dmu_buf_remove_user(dmu_buf_t *db, dmu_buf_user_t *user);
dmu_buf_user_t *dmu_buf_get_user(dmu_buf_t *db);
/*
 * Returns the user_ptr set with dmu_buf_set_user(), or NULL if not set.
 */
void *dmu_buf_get_user(dmu_buf_t *db);

/*
 * 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);

/*
 * To register a commit callback, dmu_tx_callback_register() must be called.
 *
```

```
 620  * dcb_data is a pointer to caller private data that is passed on as a
 621  * callback parameter. The caller is responsible for properly allocating and
 622  * freeing it.
 623  *
 624  * When registering a callback, the transaction must be already created, but
 625  * it cannot be committed or aborted. It can be assigned to a txg or not.
 626  *
 627  * The callback will be called after the transaction has been safely written
 628  * to stable storage and will also be called if the dmu_tx is aborted.
 629  * If there is any error which prevents the transaction from being committed to
 630  * disk, the callback will be called with a value of error != 0.
 631  */
 632 typedef void dmu_tx_callback_func_t(void *dcb_data, int error);

 634 void dmu_tx_callback_register(dmu_tx_t *tx, dmu_tx_callback_func_t *dcb_func,
 635     void *dcb_data);

 637 /*
 638  * Free up the data blocks for a defined range of a file.  If size is
 639  * -1, the range from offset to end-of-file is freed.
 640  */
 641 int dmu_free_range(objset_t *os, uint64_t object, uint64_t offset,
 642         uint64_t size, dmu_tx_t *tx);
 643 int dmu_free_long_range(objset_t *os, uint64_t object, uint64_t offset,
 644         uint64_t size);
 645 int dmu_free_object(objset_t *os, uint64_t object);

 647 /*
 648  * Convenience functions.
 649  *
 650  * Canfail routines will return 0 on success, or an errno if there is a
 651  * nonrecoverable I/O error.
 652  */
 653 #define DMU_READ_PREFETCH       0 /* prefetch */
 654 #define DMU_READ_NO_PREFETCH    1 /* don't prefetch */
 655 int dmu_read(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
 656         void *buf, uint32_t flags);
 657 void dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
 658         const void *buf, dmu_tx_t *tx);
 659 void dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
 660         dmu_tx_t *tx);
 661 int dmu_read_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size);
 662 int dmu_write_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size,
 663     dmu_tx_t *tx);
 664 int dmu_write_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size,
 665     dmu_tx_t *tx);
 666 int dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset,
 667     uint64_t size, struct page *pp, dmu_tx_t *tx);
 668 struct arc_buf *dmu_request_arcbuf(dmu_buf_t *handle, int size);
 669 void dmu_return_arcbuf(struct arc_buf *buf);
 670 void dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, struct arc_buf *buf,
 671     dmu_tx_t *tx);
 672 int dmu_xuio_init(struct xuio *uio, int niov);
 673 void dmu_xuio_fini(struct xuio *uio);
 674 int dmu_xuio_add(struct xuio *uio, struct arc_buf *abuf, offset_t off,
 675     size_t n);
 676 int dmu_xuio_cnt(struct xuio *uio);
 677 struct arc_buf *dmu_xuio_arcbuf(struct xuio *uio, int i);
 678 void dmu_xuio_clear(struct xuio *uio, int i);
 679 void xuio_stat_wbuf_copied();
 680 void xuio_stat_wbuf_nocopy();

 682 extern int zfs_prefetch_disable;

 684 /*
 685  * Asynchronously try to read in the data.
```



```
 686  */
 687 void dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset,
 688     uint64_t len);

 690 typedef struct dmu_object_info {
 691         /* All sizes are in bytes unless otherwise indicated. */
 692         uint32_t doi_data_block_size;
 693         uint32_t doi_metadata_block_size;
 694         dmu_object_type_t doi_type;
 695         dmu_object_type_t doi_bonus_type;
 696         uint64_t doi_bonus_size;
 697         uint8_t doi_indirection;                /* 2 = dnode->indirect->data */
 698         uint8_t doi_checksum;
 699         uint8_t doi_compress;
 700         uint8_t doi_pad[5];
 701         uint64_t doi_physical_blocks_512;       /* data + metadata, 512b blks */
 702         uint64_t doi_max_offset;
 703         uint64_t doi_fill_count;                /* number of non-empty blocks */
 704 } dmu_object_info_t;
_____unchanged_portion_omitted_
```

```
**********************************************************
    8239 Thu Apr 25 16:14:59 2013
new/usr/src/uts/common/fs/zfs/sys/dmu_impl.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */

#ifndef _SYS_DMU_IMPL_H
#define _SYS_DMU_IMPL_H

#include <sys/txg_impl.h>
#include <sys/zio.h>
#include <sys/dnode.h>
#include <sys/zfs_context.h>
#include <sys/zfs_ioctl.h>
#include <sys/dmu.h>
#endif /* ! codereview */

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * This is the locking strategy for the DMU.  Numbers in parenthesis are
 * cases that use that lock order, referenced below:
 *
 * ARC is self-contained
 * bplist is self-contained
 * refcount is self-contained
 * txg is self-contained (hopefully!)
 * zst_lock
 * zf_rwlock
 *
 * XXX try to improve evicting path?
 *
 * dp_config_rwlock > os_obj_lock > dn_struct_rwlock >
 *      dn_dbufs_mtx > hash_mutexes > db_mtx > dd_lock > leafs
 *
 * dp_config_rwlock
 *    must be held before: everything
```



```
 *    protects dd namespace changes
 *    protects property changes globally
 *    held from:
 *      dsl_dir_open/r:
 *      dsl_dir_create_sync/w:
 *      dsl_dir_sync_destroy/w:
 *      dsl_dir_rename_sync/w:
 *      dsl_prop_changed_notify/r:
 *
 * os_obj_lock
 *   must be held before:
 *      everything except dp_config_rwlock
 *   protects os_obj_next
 *   held from:
 *      dmu_object_alloc: dn_dbufs_mtx, db_mtx, hash_mutexes, dn_struct_rwlock
 *
 * dn_struct_rwlock
 *   must be held before:
 *      everything except dp_config_rwlock and os_obj_lock
 *   protects structure of dnode (eg. nlevels)
 *      db_blkptr can change when syncing out change to nlevels
 *      dn_maxblkid
 *      dn_nlevels
 *      dn_*blksz*
 *      phys nlevels, maxblkid, physical blkptr_t's (?)
 *   held from:
 *      callers of dbuf_read_impl, dbuf_hold[_impl], dbuf_prefetch
 *      dmu_object_info_from_dnode: dn_dirty_mtx (dn_datablksz)
 *      dmu_tx_count_free:
 *      dbuf_read_impl: db_mtx, dmu_zfetch()
 *      dmu_zfetch: zf_rwlock/r, zst_lock, dbuf_prefetch()
 *      dbuf_new_size: db_mtx
 *      dbuf_dirty: db_mtx
 *      dbuf_findbp: (callers, phys? - the real need)
 *      dbuf_create: dn_dbufs_mtx, hash_mutexes, db_mtx (phys?)
 *      dbuf_prefetch: dn_dirty_mtx, hash_mutexes, db_mtx, dn_dbufs_mtx
 *      dbuf_hold_impl: hash_mutexes, db_mtx, dn_dbufs_mtx, dbuf_findbp()
 *      dnode_sync/w (increase_indirection): db_mtx (phys)
 *      dnode_set_blksz/w: dn_dbufs_mtx (dn_*blksz*)
 *      dnode_new_blkid/w: (dn_maxblkid)
 *      dnode_free_range/w: dn_dirty_mtx (dn_maxblkid)
 *      dnode_next_offset: (phys)
 *
 * dn_dbufs_mtx
 *    must be held before:
 *      db_mtx, hash_mutexes
 *    protects:
 *      dn_dbufs
 *      dn_evicted
 *    held from:
 *      dmu_evict_user: db_mtx (dn_dbufs)
 *      dbuf_free_range: db_mtx (dn_dbufs)
 *      dbuf_remove_ref: db_mtx, callees:
 *              dbuf_hash_remove: hash_mutexes, db_mtx
 *      dbuf_create: hash_mutexes, db_mtx (dn_dbufs)
 *      dnode_set_blksz: (dn_dbufs)
 *
 * hash_mutexes (global)
 *   must be held before:
 *      db_mtx
 *   protects dbuf_hash_table (global) and db_hash_next
 *   held from:
 *      dbuf_find: db_mtx
 *      dbuf_hash_insert: db_mtx
 *      dbuf_hash_remove: db_mtx
 *
```

```
 * db_mtx (meta-leaf)
 *   must be held before:
 *      dn_mtx, dn_dirty_mtx, dd_lock (leaf mutexes)
 *   protects:
 *      db_state
 *      db_holds
 *      db_buf
 *      db_changed
 *      db_data_pending
 *      db_dirtied
 *      db_link
 *      db_dirty_node (??)
 *      db_dirtycnt
 *      db_d.*
 *      db.*
 *   held from:
 *      dbuf_dirty: dn_mtx, dn_dirty_mtx
 *      dbuf_dirty->dsl_dir_willuse_space: dd_lock
 *      dbuf_dirty->dbuf_new_block->dsl_dataset_block_freeable: dd_lock
 *      dbuf_undirty: dn_dirty_mtx (db_d)
 *      dbuf_write_done: dn_dirty_mtx (db_state)
 *      dbuf_*
 *      dmu_buf_update_user: none (db_d)
 *      dmu_evict_user: none (db_d) (maybe can eliminate)
 *      dbuf_find: none (db_holds)
 *      dbuf_hash_insert: none (db_holds)
 *      dmu_buf_read_array_impl: none (db_state, db_changed)
 *      dmu_sync: none (db_dirty_node, db_d)
 *      dnode_reallocate: none (db)
 *
 * dn_mtx (leaf)
 *   protects:
 *      dn_dirty_dbufs
 *      dn_ranges
 *      phys accounting
 *      dn_allocated_txg
 *      dn_free_txg
 *      dn_assigned_txg
 *      dd_assigned_tx
 *      dn_notxholds
 *      dn_dirtyctx
 *      dn_dirtyctx_firstset
 *      (dn_phys copy fields?)
 *      (dn_phys contents?)
 *   held from:
 *      dnode_*
 *      dbuf_dirty: none
 *      dbuf_sync: none (phys accounting)
 *      dbuf_undirty: none (dn_ranges, dn_dirty_dbufs)
 *      dbuf_write_done: none (phys accounting)
 *      dmu_object_info_from_dnode: none (accounting)
 *      dmu_tx_commit: none
 *      dmu_tx_hold_object_impl: none
 *      dmu_tx_try_assign: dn_notxholds(cv)
 *      dmu_tx_unassign: none
 *
 * dd_lock
 *    must be held before:
 *      ds_lock
 *      ancestors' dd_lock
 *    protects:
 *      dd_prop_cbs
 *      dd_sync_*
 *      dd_used_bytes
 *      dd_tempreserved
 *      dd_space_towrite
```



```
 *      dd_myname
 *      dd_phys accounting?
 *    held from:
 *      dsl_dir_*
 *      dsl_prop_changed_notify: none (dd_prop_cbs)
 *      dsl_prop_register: none (dd_prop_cbs)
 *      dsl_prop_unregister: none (dd_prop_cbs)
 *      dsl_dataset_block_freeable: none (dd_sync_*)
 *
 * os_lock (leaf)
 *   protects:
 *      os_dirty_dnodes
 *      os_free_dnodes
 *      os_dnodes
 *      os_downgraded_dbufs
 *      dn_dirtyblksz
 *      dn_dirty_link
 *   held from:
 *      dnode_create: none (os_dnodes)
 *      dnode_destroy: none (os_dnodes)
 *      dnode_setdirty: none (dn_dirtyblksz, os_*_dnodes)
 *      dnode_free: none (dn_dirtyblksz, os_*_dnodes)
 *
 * ds_lock
 *    protects:
 *      ds_objset
 *      ds_open_refcount
 *      ds_snapname
 *      ds_phys accounting
 *      ds_phys userrefs zapobj
 *      ds_reserved
 *    held from:
 *      dsl_dataset_*
 *
 * dr_mtx (leaf)
 *    protects:
 *      dr_children
 *    held from:
 *      dbuf_dirty
 *      dbuf_undirty
 *      dbuf_sync_indirect
 *      dnode_new_blkid
 */

struct objset;
struct dmu_pool;

typedef struct dmu_xuio {
        int next;
        int cnt;
        struct arc_buf **bufs;
        iovec_t *iovp;
} dmu_xuio_t;

typedef struct xuio_stats {
        /* loaned yet not returned arc_buf */
        kstat_named_t xuiostat_onloan_rbuf;
        kstat_named_t xuiostat_onloan_wbuf;
        /* whether a copy is made when loaning out a read buffer */
        kstat_named_t xuiostat_rbuf_copied;
        kstat_named_t xuiostat_rbuf_nocopy;
        /* whether a copy is made when assigning a write buffer */
        kstat_named_t xuiostat_wbuf_copied;
        kstat_named_t xuiostat_wbuf_nocopy;
} xuio_stats_t;
```

```
258 static xuio_stats_t xuio_stats = {
259         { "onloan_read_buf",    KSTAT_DATA_UINT64 },
260         { "onloan_write_buf",   KSTAT_DATA_UINT64 },
261         { "read_buf_copied",    KSTAT_DATA_UINT64 },
262         { "read_buf_nocopy",    KSTAT_DATA_UINT64 },
263         { "write_buf_copied",   KSTAT_DATA_UINT64 },
264         { "write_buf_nocopy",   KSTAT_DATA_UINT64 }
265 };

267 #define XUIOSTAT_INCR(stat, val)        \
268         atomic_add_64(&xuio_stats.stat.value.ui64, (val))
269 #define XUIOSTAT_BUMP(stat)     XUIOSTAT_INCR(stat, 1)

271 /*
272  * The list of data whose inclusion in a send stream can be pending from
273  * one call to backup_cb to another.  Multiple calls to dump_free() and
274  * dump_freeobjects() can be aggregated into a single DRR_FREE or
275  * DRR_FREEOBJECTS replay record.
276  */
277 typedef enum {
278         PENDING_NONE,
279         PENDING_FREE,
280         PENDING_FREEOBJECTS
281 } dmu_pendop_t;

283 typedef struct dmu_sendarg {
284         list_node_t dsa_link;
285         dmu_replay_record_t *dsa_drr;
286         vnode_t *dsa_vp;
287         int dsa_outfd;
288         struct proc *dsa_proc;
289         offset_t *dsa_off;
290         objset_t *dsa_os;
291         zio_cksum_t dsa_zc;
292         uint64_t dsa_toguid;
293         int dsa_err;
294         dmu_pendop_t dsa_pending_op;
295 } dmu_sendarg_t;


298 #ifdef  __cplusplus
299 }
300 #endif

302 #endif  /* _SYS_DMU_IMPL_H */
```

```
**********************************************************
   10615 Thu Apr 25 16:14:59 2013
new/usr/src/uts/common/fs/zfs/sys/dnode.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_

typedef struct dnode_children {
        dmu_buf_user_t db_evict;        /* User evict data */
#endif /* ! codereview */
        size_t dnc_count;               /* number of children */
        dnode_handle_t dnc_children[1]; /* sized dynamically */
} dnode_children_t;

typedef struct free_range {
        avl_node_t fr_node;
        uint64_t fr_blkid;
        uint64_t fr_nblks;
} free_range_t;

dnode_t *dnode_special_open(struct objset *dd, dnode_phys_t *dnp,
    uint64_t object, dnode_handle_t *dnh);
void dnode_special_close(dnode_handle_t *dnh);

void dnode_setbonuslen(dnode_t *dn, int newsize, dmu_tx_t *tx);
void dnode_setbonus_type(dnode_t *dn, dmu_object_type_t, dmu_tx_t *tx);
void dnode_rm_spill(dnode_t *dn, dmu_tx_t *tx);

int dnode_hold(struct objset *dd, uint64_t object,
    void *ref, dnode_t **dnp);
int dnode_hold_impl(struct objset *dd, uint64_t object, int flag,
    void *ref, dnode_t **dnp);
boolean_t dnode_add_ref(dnode_t *dn, void *ref);
void dnode_rele(dnode_t *dn, void *ref);
void dnode_setdirty(dnode_t *dn, dmu_tx_t *tx);
void dnode_sync(dnode_t *dn, dmu_tx_t *tx);
void dnode_allocate(dnode_t *dn, dmu_object_type_t ot, int blocksize, int ibs,
    dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
void dnode_reallocate(dnode_t *dn, dmu_object_type_t ot, int blocksize,
    dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
void dnode_free(dnode_t *dn, dmu_tx_t *tx);
void dnode_byteswap(dnode_phys_t *dnp);
void dnode_buf_byteswap(void *buf, size_t size);
void dnode_verify(dnode_t *dn);
int dnode_set_blksz(dnode_t *dn, uint64_t size, int ibs, dmu_tx_t *tx);
void dnode_free_range(dnode_t *dn, uint64_t off, uint64_t len, dmu_tx_t *tx);
void dnode_clear_range(dnode_t *dn, uint64_t blkid,
    uint64_t nblks, dmu_tx_t *tx);
void dnode_diduse_space(dnode_t *dn, int64_t space);
void dnode_willuse_space(dnode_t *dn, int64_t space, dmu_tx_t *tx);
void dnode_new_blkid(dnode_t *dn, uint64_t blkid, dmu_tx_t *tx, boolean_t);
uint64_t dnode_block_freed(dnode_t *dn, uint64_t blkid);
void dnode_init(void);
void dnode_fini(void);
int dnode_next_offset(dnode_t *dn, int flags, uint64_t *off,
    int minlvl, uint64_t blkfill, uint64_t txg);
void dnode_evict_dbufs(dnode_t *dn);

#ifdef ZFS_DEBUG

/*
 * There should be a ## between the string literal and fmt, to make it
 * clear that we're joining two strings together, but that piece of shit
 * gcc doesn't support that preprocessor token.
```



```
 */
#define dprintf_dnode(dn, fmt, ...) do { \
        if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
        char __db_buf[32]; \
        uint64_t __db_obj = (dn)->dn_object; \
        if (__db_obj == DMU_META_DNODE_OBJECT) \
                (void) strcpy(__db_buf, "mdn"); \
        else \
                (void) snprintf(__db_buf, sizeof (__db_buf), "%lld", \
                    (u_longlong_t)__db_obj);\
        dprintf_ds((dn)->dn_objset->os_dsl_dataset, "obj=%s " fmt, \
            __db_buf, __VA_ARGS__); \
        } \
_NOTE(CONSTCOND) } while (0)

#define DNODE_VERIFY(dn)                dnode_verify(dn)
#define FREE_VERIFY(db, start, end, tx) free_verify(db, start, end, tx)

#else

#define dprintf_dnode(db, fmt, ...)
#define DNODE_VERIFY(dn)
#define FREE_VERIFY(db, start, end, tx)

#endif

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_DNODE_H */
```

```
**********************************************************
   11005 Thu Apr 25 16:14:59 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
```

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

#endif /* ! codereview */
#define DS_FLAG_INCONSISTENT    (1ULL<<0)
#define DS_IS_INCONSISTENT(ds)  \
        ((ds)->ds_phys->ds_flags & DS_FLAG_INCONSISTENT)
/*
 * Note: nopromote can not yet be set, but we want support for it in this
 * on-disk version, so that we don't need to upgrade for it later.
 */
#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) \
        ((ds)->ds_phys->ds_flags & DS_FLAG_DEFER_DESTROY)

/*
 * 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.
         */
        uint64_t ds_fsid_guid;
        uint64_t ds_guid;
        uint64_t ds_flags;              /* DS_FLAG_* */
        blkptr_t ds_bp;
        uint64_t ds_next_clones_obj;    /* DMU_OT_DSL_CLONES */
        uint64_t ds_props_obj;          /* DMU_OT_DSL_PROPS for snaps */
        uint64_t ds_userrefs_obj;       /* DMU_OT_USERREFS */
        uint64_t ds_pad[5]; /* pad out to 320 bytes for good measure */
} dsl_dataset_phys_t;

typedef struct dsl_dataset_dbuf {
        uint8_t dsdb_pad[offsetof(dmu_buf_t, db_data)];
        dsl_dataset_phys_t *dsdb_data;
} dsl_dataset_dbuf_t;

#endif /* ! codereview */
typedef struct dsl_dataset {
        dmu_buf_user_t db_evict;
```

```
#endif /* ! codereview */
        /* Immutable: */
        struct dsl_dir *ds_dir;
        union {
                dmu_buf_t *ds_dmu_db;
                dsl_dataset_dbuf_t *ds_db;
        } ds_db_u;
        dsl_dataset_phys_t *ds_phys;
        dmu_buf_t *ds_dbuf;
        uint64_t ds_object;
        uint64_t ds_fsid_guid;

        /* only used in syncing context, only valid for non-snapshots: */
        struct dsl_dataset *ds_prev;

        /* has internal locking: */
        dsl_deadlist_t ds_deadlist;
        bplist_t ds_pending_deadlist;

        /* protected by lock on pool's dp_dirty_datasets list */
        txg_node_t ds_dirty_link;
        list_node_t ds_synced_link;

        /*
         * ds_phys->ds_<accounting> is also protected by ds_lock.
         * Protected by ds_lock:
         */
        kmutex_t ds_lock;
        objset_t *ds_objset;
        uint64_t ds_userrefs;
        void *ds_owner;

        /*
         * Long holds prevent the ds from being destroyed; they allow the
         * ds to remain held even after dropping the dp_config_rwlock.
         * Owning counts as a long hold.  See the comments above
         * dsl_pool_hold() for details.
         */
        refcount_t ds_longholds;

        /* no locking; only for making guesses */
        uint64_t ds_trysnap_txg;

        /* for objset_open() */
        kmutex_t ds_opening_lock;

        uint64_t ds_reserved;   /* cached refreservation */
        uint64_t ds_quota;      /* cached refquota */

        kmutex_t ds_sendstream_lock;
        list_t ds_sendstreams;

        /* Protected by ds_lock; keep at end of struct for better locality */
        char ds_snapname[MAXNAMELEN];
} dsl_dataset_t;

/* See sys/dmu.h:dmu_buf_user_t for why we have these. */
#define ds_dbuf ds_db_u.ds_dmu_db
#define ds_phys ds_db_u.ds_db->dsdb_data

struct dsl_ds_destroyarg {
        dsl_dataset_t *ds;              /* ds to destroy */
        dsl_dataset_t *rm_origin;       /* also remove our origin? */
        boolean_t is_origin_rm;         /* set if removing origin snap */
        boolean_t defer;                /* destroy -d requested? */
```



```
        boolean_t releasing;            /* destroying due to release? */
        boolean_t need_prep;            /* do we need to retry due to EBUSY? */
};

#endif /* ! codereview */
/*
 * The max length of a temporary tag prefix is the number of hex digits
 * required to express UINT64_MAX plus one for the hyphen.
 */
#define MAX_TAG_PREFIX_LEN      17

#define dsl_dataset_is_snapshot(ds) \
        ((ds)->ds_phys->ds_num_children != 0)

#define DS_UNIQUE_IS_ACCURATE(ds)       \
        (((ds)->ds_phys->ds_flags & DS_FLAG_UNIQUE_ACCURATE) != 0)

int dsl_dataset_hold(struct dsl_pool *dp, const char *name, void *tag,
    dsl_dataset_t **dsp);
int dsl_dataset_hold_obj(struct dsl_pool *dp, uint64_t dsobj, void *tag,
    dsl_dataset_t **);
void dsl_dataset_rele(dsl_dataset_t *ds, void *tag);
int dsl_dataset_own(struct dsl_pool *dp, const char *name,
    void *tag, dsl_dataset_t **dsp);
int dsl_dataset_own_obj(struct dsl_pool *dp, uint64_t dsobj,
    void *tag, dsl_dataset_t **dsp);
void dsl_dataset_disown(dsl_dataset_t *ds, void *tag);
void dsl_dataset_name(dsl_dataset_t *ds, char *name);
boolean_t dsl_dataset_tryown(dsl_dataset_t *ds, void *tag);
void dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
    minor_t minor);
uint64_t dsl_dataset_create_sync(dsl_dir_t *pds, const char *lastname,
    dsl_dataset_t *origin, uint64_t flags, cred_t *, dmu_tx_t *);
uint64_t dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
    uint64_t flags, dmu_tx_t *tx);
int dsl_dataset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors);
int dsl_dataset_promote(const char *name, char *conflsnap);
int dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
    boolean_t force);
int dsl_dataset_rename_snapshot(const char *fsname,
    const char *oldsnapname, const char *newsnapname, boolean_t recursive);
int dsl_dataset_snapshot_tmp(const char *fsname, const char *snapname,
    minor_t cleanup_minor, const char *htag);

blkptr_t *dsl_dataset_get_blkptr(dsl_dataset_t *ds);
void dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx);

spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);

boolean_t dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds);

void dsl_dataset_sync(dsl_dataset_t *os, zio_t *zio, dmu_tx_t *tx);

void dsl_dataset_block_born(dsl_dataset_t *ds, const blkptr_t *bp,
    dmu_tx_t *tx);
int dsl_dataset_block_kill(dsl_dataset_t *ds, const blkptr_t *bp,
    dmu_tx_t *tx, boolean_t async);
boolean_t dsl_dataset_block_freeable(dsl_dataset_t *ds, const blkptr_t *bp,
    uint64_t blk_birth);
uint64_t dsl_dataset_prev_snap_txg(dsl_dataset_t *ds);

void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);
void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
void dsl_dataset_space(dsl_dataset_t *ds,
    uint64_t *refdbytesp, uint64_t *availbytesp,
```

```
256     uint64_t *usedobjsp, uint64_t *availobjsp);
257 uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
258 int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
259     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
260 int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
261     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
262 boolean_t dsl_dataset_is_dirty(dsl_dataset_t *ds);

264 int dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf);

266 int dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
267     uint64_t asize, uint64_t inflight, uint64_t *used,
268     uint64_t *ref_rsrv);
269 int dsl_dataset_set_refquota(const char *dsname, zprop_source_t source,
270     uint64_t quota);
271 int dsl_dataset_set_refreservation(const char *dsname, zprop_source_t source,
272     uint64_t reservation);

274 boolean_t dsl_dataset_is_before(dsl_dataset_t *later, dsl_dataset_t *earlier);
275 void dsl_dataset_long_hold(dsl_dataset_t *ds, void *tag);
276 void dsl_dataset_long_rele(dsl_dataset_t *ds, void *tag);
277 boolean_t dsl_dataset_long_held(dsl_dataset_t *ds);

279 int dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
280     dsl_dataset_t *origin_head, boolean_t force);
281 void dsl_dataset_clone_swap_sync_impl(dsl_dataset_t *clone,
282     dsl_dataset_t *origin_head, dmu_tx_t *tx);
283 int dsl_dataset_snapshot_check_impl(dsl_dataset_t *ds, const char *snapname,
284     dmu_tx_t *tx);
285 void dsl_dataset_snapshot_sync_impl(dsl_dataset_t *ds, const char *snapname,
286     dmu_tx_t *tx);

288 void dsl_dataset_remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj,
289     dmu_tx_t *tx);
290 void dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds);
291 int dsl_dataset_get_snapname(dsl_dataset_t *ds);
292 int dsl_dataset_snap_lookup(dsl_dataset_t *ds, const char *name,
293     uint64_t *value);
294 int dsl_dataset_snap_remove(dsl_dataset_t *ds, const char *name, dmu_tx_t *tx);
295 void dsl_dataset_set_refreservation_sync_impl(dsl_dataset_t *ds,
296     zprop_source_t source, uint64_t value, dmu_tx_t *tx);
297 int dsl_dataset_rollback(const char *fsname);

299 #ifdef ZFS_DEBUG
300 #define dprintf_ds(ds, fmt, ...) do { \
301         if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
302         char *__ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
303         dsl_dataset_name(ds, __ds_name); \
304         dprintf("ds=%s " fmt, __ds_name, __VA_ARGS__); \
305         kmem_free(__ds_name, MAXNAMELEN); \
306         } \
307 _NOTE(CONSTCOND) } while (0)
308 #else
309 #define dprintf_ds(dd, fmt, ...)
310 #endif

312 #ifdef  __cplusplus
313 }
314 #endif

316 #endif /* _SYS_DSL_DATASET_H */
```

```
**********************************************************
    2790 Thu Apr 25 16:15:00 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_deleg.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.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) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _SYS_DSL_DELEG_H
#define _SYS_DSL_DELEG_H

#include <sys/zfs_context.h>
#endif /* ! codereview */
#include <sys/dmu.h>
#include <sys/dsl_pool.h>
#include <sys/zfs_context.h>

#ifdef  __cplusplus
extern "C" {
#endif

#define ZFS_DELEG_PERM_NONE             ""
#define ZFS_DELEG_PERM_CREATE           "create"
#define ZFS_DELEG_PERM_DESTROY          "destroy"
#define ZFS_DELEG_PERM_SNAPSHOT         "snapshot"
#define ZFS_DELEG_PERM_ROLLBACK         "rollback"
#define ZFS_DELEG_PERM_CLONE            "clone"
#define ZFS_DELEG_PERM_PROMOTE          "promote"
#define ZFS_DELEG_PERM_RENAME           "rename"
#define ZFS_DELEG_PERM_MOUNT            "mount"
#define ZFS_DELEG_PERM_SHARE            "share"
#define ZFS_DELEG_PERM_SEND             "send"
#define ZFS_DELEG_PERM_RECEIVE          "receive"
#define ZFS_DELEG_PERM_ALLOW            "allow"
#define ZFS_DELEG_PERM_USERPROP         "userprop"
#define ZFS_DELEG_PERM_VSCAN            "vscan"
#define ZFS_DELEG_PERM_USERQUOTA        "userquota"
#define ZFS_DELEG_PERM_GROUPQUOTA       "groupquota"
#define ZFS_DELEG_PERM_USERUSED         "userused"
#define ZFS_DELEG_PERM_GROUPUSED        "groupused"
#define ZFS_DELEG_PERM_HOLD             "hold"
#define ZFS_DELEG_PERM_RELEASE          "release"
```



```
#define ZFS_DELEG_PERM_DIFF             "diff"

/*
 * Note: the names of properties that are marked delegatable are also
 * valid delegated permissions
 */

int dsl_deleg_get(const char *ddname, nvlist_t **nvp);
int dsl_deleg_set(const char *ddname, nvlist_t *nvp, boolean_t unset);
int dsl_deleg_access(const char *ddname, const char *perm, cred_t *cr);
int dsl_deleg_access_impl(struct dsl_dataset *ds, const char *perm, cred_t *cr);
void dsl_deleg_set_create_perms(dsl_dir_t *dd, dmu_tx_t *tx, cred_t *cr);
int dsl_deleg_can_allow(char *ddname, nvlist_t *nvp, cred_t *cr);
int dsl_deleg_can_unallow(char *ddname, nvlist_t *nvp, cred_t *cr);
int dsl_deleg_destroy(objset_t *os, uint64_t zapobj, dmu_tx_t *tx);
boolean_t dsl_delegation_on(objset_t *os);

#ifdef  __cplusplus
}
```

_____*unchanged_portion_omitted_*

```
**********************************************************
    5809 Thu Apr 25 16:15:00 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_dir.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_
```

```
 77 typedef struct dsl_dir_dbuf {
 78         uint8_t dddb_pad[offsetof(dmu_buf_t, db_data)];
 79         dsl_dir_phys_t *dddb_data;
 80 } dsl_dir_dbuf_t;

 82 #endif /* ! codereview */
 83 struct dsl_dir {
 84         dmu_buf_user_t db_evict;

 86 #endif /* ! codereview */
 87         /* These are immutable; no lock needed: */
 88         uint64_t dd_object;
 89         union {
 90                 dmu_buf_t *dd_dmu_db;
 91                 dsl_dir_dbuf_t *dd_db;
 92         } dd_db_u;
 77         dsl_dir_phys_t *dd_phys;
 78         dmu_buf_t *dd_dbuf;
 93         dsl_pool_t *dd_pool;

 95         /* protected by lock on pool's dp_dirty_dirs list */
 96         txg_node_t dd_dirty_link;

 98         /* protected by dp_config_rwlock */
 99         dsl_dir_t *dd_parent;

101         /* Protected by dd_lock */
102         kmutex_t dd_lock;
103         list_t dd_prop_cbs; /* list of dsl_prop_cb_record_t's */
104         timestruc_t dd_snap_cmtime; /* last time snapshot namespace changed */
105         uint64_t dd_origin_txg;

107         /* gross estimate of space used by in-flight tx's */
108         uint64_t dd_tempreserved[TXG_SIZE];
109         /* amount of space we expect to write; == amount of dirty data */
110         int64_t dd_space_towrite[TXG_SIZE];

112         /* protected by dd_lock; keep at end of struct for better locality */
113         char dd_myname[MAXNAMELEN];
114 };

116 /* See sys/dmu.h:dmu_buf_user_t for why we have these. */
117 #define dd_dbuf dd_db_u.dd_dmu_db
118 #define dd_phys dd_db_u.dd_db->dddb_data

120 #endif /* ! codereview */
121 void dsl_dir_rele(dsl_dir_t *dd, void *tag);
122 int dsl_dir_hold(dsl_pool_t *dp, const char *name, void *tag,
123     dsl_dir_t **, const char **tail);
124 int dsl_dir_hold_obj(dsl_pool_t *dp, uint64_t ddobj,
125     const char *tail, void *tag, dsl_dir_t **);
126 void dsl_dir_name(dsl_dir_t *dd, char *buf);
127 int dsl_dir_namelen(dsl_dir_t *dd);
128 uint64_t dsl_dir_create_sync(dsl_pool_t *dp, dsl_dir_t *pds,
129     const char *name, dmu_tx_t *tx);
130 void dsl_dir_stats(dsl_dir_t *dd, nvlist_t *nv);
131 uint64_t dsl_dir_space_available(dsl_dir_t *dd,
```



```
132     dsl_dir_t *ancestor, int64_t delta, int ondiskonly);
133 void dsl_dir_dirty(dsl_dir_t *dd, dmu_tx_t *tx);
134 void dsl_dir_sync(dsl_dir_t *dd, dmu_tx_t *tx);
135 int dsl_dir_tempreserve_space(dsl_dir_t *dd, uint64_t mem,
136     uint64_t asize, uint64_t fsize, uint64_t usize, void **tr_cookiep,
137     dmu_tx_t *tx);
138 void dsl_dir_tempreserve_clear(void *tr_cookie, dmu_tx_t *tx);
139 void dsl_dir_willuse_space(dsl_dir_t *dd, int64_t space, dmu_tx_t *tx);
140 void dsl_dir_diduse_space(dsl_dir_t *dd, dd_used_t type,
141     int64_t used, int64_t compressed, int64_t uncompressed, dmu_tx_t *tx);
142 void dsl_dir_transfer_space(dsl_dir_t *dd, int64_t delta,
143     dd_used_t oldtype, dd_used_t newtype, dmu_tx_t *tx);
144 int dsl_dir_set_quota(const char *ddname, zprop_source_t source,
145     uint64_t quota);
146 int dsl_dir_set_reservation(const char *ddname, zprop_source_t source,
147     uint64_t reservation);
148 int dsl_dir_rename(const char *oldname, const char *newname);
149 int dsl_dir_transfer_possible(dsl_dir_t *sdd, dsl_dir_t *tdd, uint64_t space);
150 boolean_t dsl_dir_is_clone(dsl_dir_t *dd);
151 void dsl_dir_new_refreservation(dsl_dir_t *dd, struct dsl_dataset *ds,
152     uint64_t reservation, cred_t *cr, dmu_tx_t *tx);
153 void dsl_dir_snap_cmtime_update(dsl_dir_t *dd);
154 timestruc_t dsl_dir_snap_cmtime(dsl_dir_t *dd);
155 void dsl_dir_set_reservation_sync_impl(dsl_dir_t *dd, uint64_t value,
156     dmu_tx_t *tx);

158 /* internal reserved dir name */
159 #define MOS_DIR_NAME "$MOS"
160 #define ORIGIN_DIR_NAME "$ORIGIN"
161 #define XLATION_DIR_NAME "$XLATION"
162 #define FREE_DIR_NAME "$FREE"

164 #ifdef ZFS_DEBUG
165 #define dprintf_dd(dd, fmt, ...) do { \
166         if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
167         char *__ds_name = kmem_alloc(MAXNAMELEN + strlen(MOS_DIR_NAME) + 1, \
168             KM_SLEEP); \
169         dsl_dir_name(dd, __ds_name); \
170         dprintf("dd=%s " fmt, __ds_name, __VA_ARGS__); \
171         kmem_free(__ds_name, MAXNAMELEN + strlen(MOS_DIR_NAME) + 1); \
172         } \
173 _NOTE(CONSTCOND) } while (0)
174 #else
175 #define dprintf_dd(dd, fmt, ...)
176 #endif

178 #ifdef  __cplusplus
179 }
180 #endif

182 #endif /* _SYS_DSL_DIR_H */
```

```
**********************************************************
    8415 Thu Apr 25 16:15:00 2013
new/usr/src/uts/common/fs/zfs/sys/sa_impl.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_

 205 /*
 206  * Opaque handle used for most sa functions
 207  *
 208  * This needs to be kept as small as possible.
 209  */

 211 struct sa_handle {
 212         dmu_buf_user_t  db_evict;
 213 #endif /* ! codereview */
 214         kmutex_t        sa_lock;
 215         dmu_buf_t       *sa_bonus;
 216         dmu_buf_t       *sa_spill;
 217         objset_t        *sa_os;
 218         void            *sa_userp;
 219         sa_idx_tab_t    *sa_bonus_tab;   /* idx of bonus */
 220         sa_idx_tab_t    *sa_spill_tab; /* only present if spill activated */
 221 };

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

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

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

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

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

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

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


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

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

 252 #define SA_SET_HDR(hdr, num, size) \
 253         { \
 254                 hdr->sa_magic = SA_MAGIC; \
 255                 SA_HDR_LAYOUT_INFO_ENCODE(hdr->sa_layout_info, num, size); \
 256         }

 258 #define SA_ATTR_INFO(sa, idx, hdr, attr, bulk, type, hdl) \
 259         { \
 260                 bulk.sa_size = SA_ATTR_LEN(sa, idx, attr, hdr); \
 261                 bulk.sa_buftype = type; \
```



```
 262                 bulk.sa_addr = \
 263                     (void *)((uintptr_t)TOC_OFF(idx->sa_idx_tab[attr]) + \
 264                     (uintptr_t)hdr); \
 265 }

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

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

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

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

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

 286 #ifdef  __cplusplus
 287 }
 288 #endif

 290 #endif  /* _SYS_SA_IMPL_H */
```

```
**********************************************************
    7196 Thu Apr 25 16:15:01 2013
new/usr/src/uts/common/fs/zfs/sys/zap_impl.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_
```

```
 65 typedef struct mzap_dbuf {
 66         uint8_t mzdb_pad[offsetof(dmu_buf_t, db_data)];
 67         mzap_phys_t *mzdb_data;
 68 } mzap_dbuf_t;

 70 #endif /* ! codereview */
 71 typedef struct mzap_ent {
 72         avl_node_t mze_node;
 73         int mze_chunkid;
 74         uint64_t mze_hash;
 75         uint32_t mze_cd; /* copy from mze_phys->mze_cd */
 76 } mzap_ent_t;

 78 #define MZE_PHYS(zap, mze) \
 79         (&(zap)->zap_m_phys->mz_chunk[(mze)->mze_chunkid])
 65         (&(zap)->zap_m.zap_phys->mz_chunk[(mze)->mze_chunkid])

 81 /*
 82  * The (fat) zap is stored in one object. It is an array of
 83  * 1<<FZAP_BLOCK_SHIFT byte blocks. The layout looks like one of:
 84  *
 85  * ptrtbl fits in first block:
 86  *      [zap_phys_t zap_ptrtbl_shift < 6] [zap_leaf_t] ...
 87  *
 88  * ptrtbl too big for first block:
 89  *      [zap_phys_t zap_ptrtbl_shift >= 6] [zap_leaf_t] [ptrtbl] ...
 90  *
 91  */

 93 struct dmu_buf;
 94 struct zap_leaf;

 96 #define ZBT_LEAF                ((1ULL << 63) + 0)
 97 #define ZBT_HEADER              ((1ULL << 63) + 1)
 98 #define ZBT_MICRO               ((1ULL << 63) + 3)
 99 /* any other values are ptrtbl blocks */

101 /*
102  * the embedded pointer table takes up half a block:
103  * block size / entry size (2^3) / 2
104  */
105 #define ZAP_EMBEDDED_PTRTBL_SHIFT(zap) (FZAP_BLOCK_SHIFT(zap) - 3 - 1)

107 /*
108  * The embedded pointer table starts half-way through the block.  Since
109  * the pointer table itself is half the block, it starts at (64-bit)
110  * word number (1<<ZAP_EMBEDDED_PTRTBL_SHIFT(zap)).
111  */
112 #define ZAP_EMBEDDED_PTRTBL_ENT(zap, idx) \
113         ((uint64_t *)(zap)->zap_f_phys) \
 99         ((uint64_t *)(zap)->zap_f.zap_phys) \
114         [(idx) + (1<<ZAP_EMBEDDED_PTRTBL_SHIFT(zap))]

116 /*
117  * TAKE NOTE:
118  * If zap_phys_t is modified, zap_byteswap() must be modified.
119  */
```



```
120 typedef struct zap_phys {
121         uint64_t zap_block_type;        /* ZBT_HEADER */
122         uint64_t zap_magic;             /* ZAP_MAGIC */

124         struct zap_table_phys {
125                 uint64_t zt_blk;        /* starting block number */
126                 uint64_t zt_numblks;    /* number of blocks */
127                 uint64_t zt_shift;      /* bits to index it */
128                 uint64_t zt_nextblk;    /* next (larger) copy start block */
129                 uint64_t zt_blks_copied; /* number source blocks copied */
130         } zap_ptrtbl;

132         uint64_t zap_freeblk;           /* the next free block */
133         uint64_t zap_num_leafs;         /* number of leafs */
134         uint64_t zap_num_entries;       /* number of entries */
135         uint64_t zap_salt;              /* salt to stir into hash function */
136         uint64_t zap_normflags;         /* flags for u8_textprep_str() */
137         uint64_t zap_flags;             /* zap_flags_t */
138         /*
139          * This structure is followed by padding, and then the embedded
140          * pointer table.  The embedded pointer table takes up second
141          * half of the block.  It is accessed using the
142          * ZAP_EMBEDDED_PTRTBL_ENT() macro.
143          */
144 } zap_phys_t;

146 typedef struct zap_table_phys zap_table_phys_t;

148 typedef struct fzap_dbuf {
149         uint8_t fzdb_pad[offsetof(dmu_buf_t, db_data)];
150         zap_phys_t *fzdb_data;
151 } fzap_dbuf_t;

153 #endif /* ! codereview */
154 typedef struct zap {
155         dmu_buf_user_t db_evict;
156 #endif /* ! codereview */
157         objset_t *zap_objset;
158         uint64_t zap_object;
159         union {
160                 dmu_buf_t *zap_dmu_db;
161                 mzap_dbuf_t *mzap_db;
162                 fzap_dbuf_t *fzap_db;
163         } zap_db_u;
134         struct dmu_buf *zap_dbuf;
164         krwlock_t zap_rwlock;
165         boolean_t zap_ismicro;
166         int zap_normflags;
167         uint64_t zap_salt;
168         union {
169                 struct {
170                         /* protects zap_num_entries */
141                         zap_phys_t *zap_phys;

143                         /*
144                          * zap_num_entries_mtx protects
145                          * zap_num_entries
146                          */
171                         kmutex_t zap_num_entries_mtx;
172                         int zap_block_shift;
173                 } zap_fat;
174                 struct {
151                         mzap_phys_t *zap_phys;
175                         int16_t zap_num_entries;
176                         int16_t zap_num_chunks;
177                         int16_t zap_alloc_next;
```

```
178                     avl_tree_t zap_avl;
179             } zap_micro;
180     } zap_u;
181 } zap_t;

183 /* See sys/dmu.h:dmu_buf_user_t for why we have these. */
184 #define zap_dbuf        zap_db_u.zap_dmu_db
185 #define zap_f           zap_u.zap_fat
186 #define zap_m           zap_u.zap_micro
187 #define zap_f_phys      zap_db_u.fzap_db->fzdb_data
188 #define zap_m_phys      zap_db_u.mzap_db->mzdb_data

190 #endif /* ! codereview */
191 typedef struct zap_name {
192         zap_t *zn_zap;
193         int zn_key_intlen;
194         const void *zn_key_orig;
195         int zn_key_orig_numints;
196         const void *zn_key_norm;
197         int zn_key_norm_numints;
198         uint64_t zn_hash;
199         matchtype_t zn_matchtype;
200         char zn_normbuf[ZAP_MAXNAMELEN];
201 } zap_name_t;

160 #define zap_f   zap_u.zap_fat
161 #define zap_m   zap_u.zap_micro

203 boolean_t zap_match(zap_name_t *zn, const char *matchname);
204 int zap_lockdir(objset_t *os, uint64_t obj, dmu_tx_t *tx,
205     krw_t lti, boolean_t fatreader, boolean_t adding, zap_t **zapp);
206 void zap_unlockdir(zap_t *zap);
207 void zap_evict(dmu_buf_user_t *dbu);
167 void zap_evict(dmu_buf_t *db, void *vmzap);
208 zap_name_t *zap_name_alloc(zap_t *zap, const char *key, matchtype_t mt);
209 void zap_name_free(zap_name_t *zn);
210 int zap_hashbits(zap_t *zap);
211 uint32_t zap_maxcd(zap_t *zap);
212 uint64_t zap_getflags(zap_t *zap);

214 #define ZAP_HASH_IDX(hash, n) (((n) == 0) ? 0 : ((hash) >> (64 - (n))))

216 void fzap_byteswap(void *buf, size_t size);
217 int fzap_count(zap_t *zap, uint64_t *count);
218 int fzap_lookup(zap_name_t *zn,
219     uint64_t integer_size, uint64_t num_integers, void *buf,
220     char *realname, int rn_len, boolean_t *normalization_conflictp);
221 void fzap_prefetch(zap_name_t *zn);
222 int fzap_count_write(zap_name_t *zn, int add, uint64_t *towrite,
223     uint64_t *tooverwrite);
224 int fzap_add(zap_name_t *zn, uint64_t integer_size, uint64_t num_integers,
225     const void *val, dmu_tx_t *tx);
226 int fzap_update(zap_name_t *zn,
227     int integer_size, uint64_t num_integers, const void *val, dmu_tx_t *tx);
228 int fzap_length(zap_name_t *zn,
229     uint64_t *integer_size, uint64_t *num_integers);
230 int fzap_remove(zap_name_t *zn, dmu_tx_t *tx);
231 int fzap_cursor_retrieve(zap_t *zap, zap_cursor_t *zc, zap_attribute_t *za);
232 void fzap_get_stats(zap_t *zap, zap_stats_t *zs);
233 void zap_put_leaf(struct zap_leaf *l);

235 int fzap_add_cd(zap_name_t *zn,
236     uint64_t integer_size, uint64_t num_integers,
237     const void *val, uint32_t cd, dmu_tx_t *tx);
238 void fzap_upgrade(zap_t *zap, dmu_tx_t *tx, zap_flags_t flags);
239 int fzap_cursor_move_to_key(zap_cursor_t *zc, zap_name_t *zn);
```



```
241 #ifdef  __cplusplus
242 }
```

unchanged_portion_omitted_

```
**********************************************************
    7867 Thu Apr 25 16:15:01 2013
new/usr/src/uts/common/fs/zfs/sys/zap_leaf.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_
```

```
130 typedef struct zap_leaf_dbuf {
131         uint8_t zldb_pad[offsetof(dmu_buf_t, db_data)];
132         zap_leaf_phys_t *zldb_data;
133 } zap_leaf_dbuf_t;

135 #endif /* ! codereview */
136 typedef union zap_leaf_chunk {
137         struct zap_leaf_entry {
138                 uint8_t le_type;                /* always ZAP_CHUNK_ENTRY */
139                 uint8_t le_value_intlen;        /* size of value's ints */
140                 uint16_t le_next;               /* next entry in hash chain */
141                 uint16_t le_name_chunk;         /* first chunk of the name */
142                 uint16_t le_name_numints;       /* ints in name (incl null) */
143                 uint16_t le_value_chunk;        /* first chunk of the value */
144                 uint16_t le_value_numints;      /* value length in ints */
145                 uint32_t le_cd;                 /* collision differentiator */
146                 uint64_t le_hash;               /* hash value of the name */
147         } l_entry;
148         struct zap_leaf_array {
149                 uint8_t la_type;                /* always ZAP_CHUNK_ARRAY */
150                 uint8_t la_array[ZAP_LEAF_ARRAY_BYTES];
151                 uint16_t la_next;               /* next blk or CHAIN_END */
152         } l_array;
153         struct zap_leaf_free {
154                 uint8_t lf_type;                /* always ZAP_CHUNK_FREE */
155                 uint8_t lf_pad[ZAP_LEAF_ARRAY_BYTES];
156                 uint16_t lf_next;       /* next in free list, or CHAIN_END */
157         } l_free;
158 } zap_leaf_chunk_t;

160 typedef struct zap_leaf {
161         dmu_buf_user_t db_evict;
162 #endif /* ! codereview */
163         krwlock_t l_rwlock;
164         uint64_t l_blkid;               /* 1<<ZAP_BLOCK_SHIFT byte block off */
165         int l_bs;                       /* block size shift */
166         union {
167                 dmu_buf_t *l_dmu_db;
168                 zap_leaf_dbuf_t *l_db;
169         } zl_db_u;
130         dmu_buf_t *l_dbuf;
131         zap_leaf_phys_t *l_phys;
170 } zap_leaf_t;

172 #define l_dbuf zl_db_u.l_dmu_db
173 #define l_phys zl_db_u.l_db->zldb_data
174 #endif /* ! codereview */

176 typedef struct zap_entry_handle {
177         /* below is set by zap_leaf.c and is public to zap.c */
178         uint64_t zeh_num_integers;
179         uint64_t zeh_hash;
180         uint32_t zeh_cd;
181         uint8_t zeh_integer_size;

183         /* below is private to zap_leaf.c */
184         uint16_t zeh_fakechunk;
```



```
185         uint16_t *zeh_chunkp;
186         zap_leaf_t *zeh_leaf;
187 } zap_entry_handle_t;

189 /*
190  * Return a handle to the named entry, or ENOENT if not found.  The hash
191  * value must equal zap_hash(name).
192  */
193 extern int zap_leaf_lookup(zap_leaf_t *l,
194     struct zap_name *zn, zap_entry_handle_t *zeh);

196 /*
197  * Return a handle to the entry with this hash+cd, or the entry with the
198  * next closest hash+cd.
199  */
200 extern int zap_leaf_lookup_closest(zap_leaf_t *l,
201     uint64_t hash, uint32_t cd, zap_entry_handle_t *zeh);

203 /*
204  * Read the first num_integers in the attribute.  Integer size
205  * conversion will be done without sign extension.  Return EINVAL if
206  * integer_size is too small.  Return EOVERFLOW if there are more than
207  * num_integers in the attribute.
208  */
209 extern int zap_entry_read(const zap_entry_handle_t *zeh,
210     uint8_t integer_size, uint64_t num_integers, void *buf);

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

215 /*
216  * Replace the value of an existing entry.
217  *
218  * zap_entry_update may fail if it runs out of space (ENOSPC).
219  */
220 extern int zap_entry_update(zap_entry_handle_t *zeh,
221     uint8_t integer_size, uint64_t num_integers, const void *buf);

223 /*
224  * Remove an entry.
225  */
226 extern void zap_entry_remove(zap_entry_handle_t *zeh);

228 /*
229  * Create an entry. An equal entry must not exist, and this entry must
230  * belong in this leaf (according to its hash value).  Fills in the
231  * entry handle on success.  Returns 0 on success or ENOSPC on failure.
232  */
233 extern int zap_entry_create(zap_leaf_t *l, struct zap_name *zn, uint32_t cd,
234     uint8_t integer_size, uint64_t num_integers, const void *buf,
235     zap_entry_handle_t *zeh);

237 /*
238  * Return true if there are additional entries with the same normalized
239  * form.
240  */
241 extern boolean_t zap_entry_normalization_conflict(zap_entry_handle_t *zeh,
242     struct zap_name *zn, const char *name, struct zap *zap);

244 /*
245  * Other stuff.
246  */

248 extern void zap_leaf_init(zap_leaf_t *l, boolean_t sort);
249 extern void zap_leaf_byteswap(zap_leaf_phys_t *buf, int len);
250 extern void zap_leaf_split(zap_leaf_t *l, zap_leaf_t *nl, boolean_t sort);
```

```
251 extern void zap_leaf_stats(struct zap *zap, zap_leaf_t *l,
252     struct zap_stats *zs);

254 #ifdef  __cplusplus
255 }
256 #endif

258 #endif /* _SYS_ZAP_LEAF_H */
```

```
**********************************************************
    2133 Thu Apr 25 16:15:01 2013
new/usr/src/uts/common/fs/zfs/sys/zfs_context.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _SYS_ZFS_CONTEXT_H
#define _SYS_ZFS_CONTEXT_H

#ifdef  __cplusplus
extern "C" {
#endif

#ifndef _KERNEL
#include <stddef.h>
#endif
#endif /* ! codereview */
#include <sys/note.h>
#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/atomic.h>
#include <sys/sysmacros.h>
#include <sys/bitmap.h>
#include <sys/cmn_err.h>
#include <sys/kmem.h>
#include <sys/taskq.h>
#include <sys/taskq_impl.h>
#include <sys/buf.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/cpuvar.h>
#include <sys/kobj.h>
#include <sys/conf.h>
#include <sys/disp.h>
#include <sys/debug.h>
```



```
#include <sys/random.h>
#include <sys/byteorder.h>
#include <sys/systm.h>
#include <sys/list.h>
#include <sys/uio.h>
#include <sys/dirent.h>
#include <sys/time.h>
#include <vm/seg_kmem.h>
#include <sys/zone.h>
#include <sys/uio.h>
#include <sys/zfs_debug.h>
#include <sys/sysevent.h>
#include <sys/sysevent/eventdefs.h>
#include <sys/sysevent/dev.h>
#include <sys/fm/util.h>
#include <sys/sunddi.h>
#include <sys/cyclic.h>

#define CPU_SEQID       (CPU->cpu_seqid)

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_ZFS_CONTEXT_H */
```

```
**********************************************************
   10120 Thu Apr 25 16:15:01 2013
new/usr/src/uts/common/fs/zfs/sys/zfs_ioctl.h
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
```

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

#ifndef _SYS_ZFS_IOCTL_H
#define _SYS_ZFS_IOCTL_H

#ifndef _KERNEL
#include <stddef.h>
#endif
#endif /* ! codereview */
#include <sys/cred.h>
#include <sys/dmu.h>
#include <sys/zio.h>
#include <sys/dsl_deleg.h>
#include <sys/spa.h>
#include <sys/zfs_stat.h>

#ifdef _KERNEL
#include <sys/nvpair.h>
#endif  /* _KERNEL */

#ifdef __cplusplus
extern "C" {
#endif

/*
 * The structures in this file are passed between userland and the
 * kernel.  Userland may be running a 32-bit process, while the kernel
 * is 64-bit.  Therefore, these structures need to compile the same in
 * 32-bit and 64-bit.  This means not using type "long", and adding
 * explicit padding so that the 32-bit structure will not be packed more
 * tightly than the 64-bit structure (which requires 64-bit alignment).
 */

/*
 * Property values for snapdir
 */
```



```
#define ZFS_SNAPDIR_HIDDEN              0
#define ZFS_SNAPDIR_VISIBLE             1

/*
 * Field manipulation macros for the drr_versioninfo field of the
 * send stream header.
 */

/*
 * Header types for zfs send streams.
 */
typedef enum drr_headertype {
        DMU_SUBSTREAM = 0x1,
        DMU_COMPOUNDSTREAM = 0x2
} drr_headertype_t;

#define DMU_GET_STREAM_HDRTYPE(vi)      BF64_GET((vi), 0, 2)
#define DMU_SET_STREAM_HDRTYPE(vi, x)   BF64_SET((vi), 0, 2, x)

#define DMU_GET_FEATUREFLAGS(vi)        BF64_GET((vi), 2, 30)
#define DMU_SET_FEATUREFLAGS(vi, x)     BF64_SET((vi), 2, 30, x)

/*
 * Feature flags for zfs send streams (flags in drr_versioninfo)
 */

#define DMU_BACKUP_FEATURE_DEDUP        (0x1)
#define DMU_BACKUP_FEATURE_DEDUPPROPS   (0x2)
#define DMU_BACKUP_FEATURE_SA_SPILL     (0x4)

/*
 * Mask of all supported backup features
 */
#define DMU_BACKUP_FEATURE_MASK (DMU_BACKUP_FEATURE_DEDUP | \
                DMU_BACKUP_FEATURE_DEDUPPROPS | DMU_BACKUP_FEATURE_SA_SPILL)

/* Are all features in the given flag word currently supported? */
#define DMU_STREAM_SUPPORTED(x) (!((x) & ~DMU_BACKUP_FEATURE_MASK))

/*
 * The drr_versioninfo field of the dmu_replay_record has the
 * following layout:
 *
 *      64      56      48      40      32      24      16      8       0
 *      +-------+-------+-------+-------+-------+-------+-------+-------+
 *      |               reserved        |        feature-flags      |C|S|
 *      +-------+-------+-------+-------+-------+-------+-------+-------+
 *
 * The low order two bits indicate the header type: SUBSTREAM (0x1)
 * or COMPOUNDSTREAM (0x2).  Using two bits for this is historical:
 * this field used to be a version number, where the two version types
 * were 1 and 2.  Using two bits for this allows earlier versions of
 * the code to be able to recognize send streams that don't use any
 * of the features indicated by feature flags.
 */

#define DMU_BACKUP_MAGIC 0x2F5bacbacULL

#define DRR_FLAG_CLONE          (1<<0)
#define DRR_FLAG_CI_DATA        (1<<1)

/*
 * flags in the drr_checksumflags field in the DRR_WRITE and
 * DRR_WRITE_BYREF blocks
 */
#define DRR_CHECKSUM_DEDUP      (1<<0)
```

```
#define DRR_IS_DEDUP_CAPABLE(flags)     ((flags) & DRR_CHECKSUM_DEDUP)

/*
 * zfs ioctl command structure
 */
typedef struct dmu_replay_record {
        enum {
                DRR_BEGIN, DRR_OBJECT, DRR_FREEOBJECTS,
                DRR_WRITE, DRR_FREE, DRR_END, DRR_WRITE_BYREF,
                DRR_SPILL, DRR_NUMTYPES
        } drr_type;
        uint32_t drr_payloadlen;
        union {
                struct drr_begin {
                        uint64_t drr_magic;
                        uint64_t drr_versioninfo; /* was drr_version */
                        uint64_t drr_creation_time;
                        dmu_objset_type_t drr_type;
                        uint32_t drr_flags;
                        uint64_t drr_toguid;
                        uint64_t drr_fromguid;
                        char drr_toname[MAXNAMELEN];
                } drr_begin;
                struct drr_end {
                        zio_cksum_t drr_checksum;
                        uint64_t drr_toguid;
                } drr_end;
                struct drr_object {
                        uint64_t drr_object;
                        dmu_object_type_t drr_type;
                        dmu_object_type_t drr_bonustype;
                        uint32_t drr_blksz;
                        uint32_t drr_bonuslen;
                        uint8_t drr_checksumtype;
                        uint8_t drr_compress;
                        uint8_t drr_pad[6];
                        uint64_t drr_toguid;
                        /* bonus content follows */
                } drr_object;
                struct drr_freeobjects {
                        uint64_t drr_firstobj;
                        uint64_t drr_numobjs;
                        uint64_t drr_toguid;
                } drr_freeobjects;
                struct drr_write {
                        uint64_t drr_object;
                        dmu_object_type_t drr_type;
                        uint32_t drr_pad;
                        uint64_t drr_offset;
                        uint64_t drr_length;
                        uint64_t drr_toguid;
                        uint8_t drr_checksumtype;
                        uint8_t drr_checksumflags;
                        uint8_t drr_pad2[6];
                        ddt_key_t drr_key; /* deduplication key */
                        /* content follows */
                } drr_write;
                struct drr_free {
                        uint64_t drr_object;
                        uint64_t drr_offset;
                        uint64_t drr_length;
                        uint64_t drr_toguid;
                } drr_free;
                struct drr_write_byref {
                        /* where to put the data */
```



```
                        uint64_t drr_object;
                        uint64_t drr_offset;
                        uint64_t drr_length;
                        uint64_t drr_toguid;
                        /* where to find the prior copy of the data */
                        uint64_t drr_refguid;
                        uint64_t drr_refobject;
                        uint64_t drr_refoffset;
                        /* properties of the data */
                        uint8_t drr_checksumtype;
                        uint8_t drr_checksumflags;
                        uint8_t drr_pad2[6];
                        ddt_key_t drr_key; /* deduplication key */
                } drr_write_byref;
                struct drr_spill {
                        uint64_t drr_object;
                        uint64_t drr_length;
                        uint64_t drr_toguid;
                        uint64_t drr_pad[4]; /* needed for crypto */
                        /* spill data follows */
                } drr_spill;
        } drr_u;
} dmu_replay_record_t;

/* diff record range types */
typedef enum diff_type {
        DDR_NONE = 0x1,
        DDR_INUSE = 0x2,
        DDR_FREE = 0x4
} diff_type_t;

/*
 * The diff reports back ranges of free or in-use objects.
 */
typedef struct dmu_diff_record {
        uint64_t ddr_type;
        uint64_t ddr_first;
        uint64_t ddr_last;
} dmu_diff_record_t;

typedef struct zinject_record {
        uint64_t        zi_objset;
        uint64_t        zi_object;
        uint64_t        zi_start;
        uint64_t        zi_end;
        uint64_t        zi_guid;
        uint32_t        zi_level;
        uint32_t        zi_error;
        uint64_t        zi_type;
        uint32_t        zi_freq;
        uint32_t        zi_failfast;
        char            zi_func[MAXNAMELEN];
        uint32_t        zi_iotype;
        int32_t         zi_duration;
        uint64_t        zi_timer;
        uint32_t        zi_cmd;
        uint32_t        zi_pad;
} zinject_record_t;

#define ZINJECT_NULL            0x1
#define ZINJECT_FLUSH_ARC       0x2
#define ZINJECT_UNLOAD_SPA      0x4

typedef enum zinject_type {
        ZINJECT_UNINITIALIZED,
        ZINJECT_DATA_FAULT,
```

```
258         ZINJECT_DEVICE_FAULT,
259         ZINJECT_LABEL_FAULT,
260         ZINJECT_IGNORED_WRITES,
261         ZINJECT_PANIC,
262         ZINJECT_DELAY_IO,
263 } zinject_type_t;

265 typedef struct zfs_share {
266         uint64_t        z_exportdata;
267         uint64_t        z_sharedata;
268         uint64_t        z_sharetype;    /* 0 = share, 1 = unshare */
269         uint64_t        z_sharemax;  /* max length of share string */
270 } zfs_share_t;

272 /*
273  * ZFS file systems may behave the usual, POSIX-compliant way, where
274  * name lookups are case-sensitive.  They may also be set up so that
275  * all the name lookups are case-insensitive, or so that only some
276  * lookups, the ones that set an FIGNORECASE flag, are case-insensitive.
277  */
278 typedef enum zfs_case {
279         ZFS_CASE_SENSITIVE,
280         ZFS_CASE_INSENSITIVE,
281         ZFS_CASE_MIXED
282 } zfs_case_t;

284 typedef struct zfs_cmd {
285         char            zc_name[MAXPATHLEN];    /* name of pool or dataset */
286         uint64_t        zc_nvlist_src;          /* really (char *) */
287         uint64_t        zc_nvlist_src_size;
288         uint64_t        zc_nvlist_dst;          /* really (char *) */
289         uint64_t        zc_nvlist_dst_size;
290         boolean_t       zc_nvlist_dst_filled;   /* put an nvlist in dst? */
291         int             zc_pad2;

293         /*
294          * The following members are for legacy ioctls which haven't been
295          * converted to the new method.
296          */
297         uint64_t        zc_history;             /* really (char *) */
298         char            zc_value[MAXPATHLEN * 2];
299         char            zc_string[MAXNAMELEN];
300         uint64_t        zc_guid;
301         uint64_t        zc_nvlist_conf;         /* really (char *) */
302         uint64_t        zc_nvlist_conf_size;
303         uint64_t        zc_cookie;
304         uint64_t        zc_objset_type;
305         uint64_t        zc_perm_action;
306         uint64_t        zc_history_len;
307         uint64_t        zc_history_offset;
308         uint64_t        zc_obj;
309         uint64_t        zc_iflags;              /* internal to zfs(7fs) */
310         zfs_share_t     zc_share;
311         dmu_objset_stats_t zc_objset_stats;
312         struct drr_begin zc_begin_record;
313         zinject_record_t zc_inject_record;
314         boolean_t       zc_defer_destroy;
315         boolean_t       zc_temphold;
316         uint64_t        zc_action_handle;
317         int             zc_cleanup_fd;
318         uint8_t         zc_pad[4];              /* alignment */
319         uint64_t        zc_sendobj;
320         uint64_t        zc_fromobj;
321         uint64_t        zc_createtxg;
322         zfs_stat_t      zc_stat;
323 } zfs_cmd_t;
```



```
325 typedef struct zfs_useracct {
326         char zu_domain[256];
327         uid_t zu_rid;
328         uint32_t zu_pad;
329         uint64_t zu_space;
330 } zfs_useracct_t;

332 #define ZFSDEV_MAX_MINOR        (1 << 16)
333 #define ZFS_MIN_MINOR   (ZFSDEV_MAX_MINOR + 1)

335 #define ZPOOL_EXPORT_AFTER_SPLIT 0x1

337 #ifdef _KERNEL

339 typedef struct zfs_creat {
340         nvlist_t        *zct_zplprops;
341         nvlist_t        *zct_props;
342 } zfs_creat_t;

344 extern dev_info_t *zfs_dip;

346 extern int zfs_secpolicy_snapshot_perms(const char *name, cred_t *cr);
347 extern int zfs_secpolicy_rename_perms(const char *from,
348     const char *to, cred_t *cr);
349 extern int zfs_secpolicy_destroy_perms(const char *name, cred_t *cr);
350 extern int zfs_busy(void);
351 extern void zfs_unmount_snap(const char *);
352 extern void zfs_destroy_unmount_origin(const char *);

354 /*
355  * ZFS minor numbers can refer to either a control device instance or
356  * a zvol. Depending on the value of zss_type, zss_data points to either
357  * a zvol_state_t or a zfs_onexit_t.
358  */
359 enum zfs_soft_state_type {
360         ZSST_ZVOL,
361         ZSST_CTLDEV
362 };

364 typedef struct zfs_soft_state {
365         enum zfs_soft_state_type zss_type;
366         void *zss_data;
367 } zfs_soft_state_t;

369 extern void *zfsdev_get_soft_state(minor_t minor,
370     enum zfs_soft_state_type which);
371 extern minor_t zfsdev_minor_alloc(void);

373 extern void *zfsdev_state;
374 extern kmutex_t zfsdev_state_lock;

376 #endif  /* _KERNEL */

378 #ifdef  __cplusplus
379 }
380 #endif

382 #endif  /* _SYS_ZFS_IOCTL_H */
```

```
**********************************************************
   33655 Thu Apr 25 16:15:02 2013
new/usr/src/uts/common/fs/zfs/zap.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.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.
 */

/*
 * 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 */

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->db_evict.evict_func = zap_evict;
        (void) dmu_buf_update_user(zap->zap_dbuf, zap, zap,
            &zap->zap_f.zap_phys, zap_evict);

        mutex_init(&zap->zap_f.zap_num_entries_mtx, 0, 0, 0);
        zap->zap_f.zap_block_shift = highbit(zap->zap_dbuf->db_size) - 1;

        zp = zap->zap_f.zap_phys;
        zp = zap->zap_f.zap_phys;
        /*
         * 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;
        l->l_phys = db->db_data;

        zap_leaf_init(l, zp->zap_normflags != 0);
```

```
120         kmem_free(l, sizeof (zap_leaf_t));
121         dmu_buf_rele(db, FTAG);
122 }
_____unchanged_portion_omitted_

314 static int
315 zap_grow_ptrtbl(zap_t *zap, dmu_tx_t *tx)
316 {
317         /*
318          * The pointer table should never use more hash bits than we
319          * have (otherwise we'd be using useless zero bits to index it).
320          * If we are within 2 bits of running out, stop growing, since
321          * this is already an aberrant condition.
322          */
323         if (zap->zap_f_phys->zap_ptrtbl.zt_shift >= zap_hashbits(zap) - 2)
327         if (zap->zap_f.zap_phys->zap_ptrtbl.zt_shift >= zap_hashbits(zap) - 2)
324                 return (SET_ERROR(ENOSPC));

326         if (zap->zap_f_phys->zap_ptrtbl.zt_numblks == 0) {
330         if (zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks == 0) {
327                 /*
328                  * We are outgrowing the "embedded" ptrtbl (the one
329                  * stored in the header block).  Give it its own entire
330                  * block, which will double the size of the ptrtbl.
331                  */
332                 uint64_t newblk;
333                 dmu_buf_t *db_new;
334                 int err;

336                 ASSERT3U(zap->zap_f_phys->zap_ptrtbl.zt_shift, ==,
340                 ASSERT3U(zap->zap_f.zap_phys->zap_ptrtbl.zt_shift, ==,
337                     ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
338                 ASSERT0(zap->zap_f_phys->zap_ptrtbl.zt_blk);
342                 ASSERT0(zap->zap_f.zap_phys->zap_ptrtbl.zt_blk);

340                 newblk = zap_allocate_blocks(zap, 1);
341                 err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
342                     newblk << FZAP_BLOCK_SHIFT(zap), FTAG, &db_new,
343                     DMU_READ_NO_PREFETCH);
344                 if (err)
345                         return (err);
346                 dmu_buf_will_dirty(db_new, tx);
347                 zap_ptrtbl_transfer(&ZAP_EMBEDDED_PTRTBL_ENT(zap, 0),
348                     db_new->db_data, 1 << ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
349                 dmu_buf_rele(db_new, FTAG);

351                 zap->zap_f_phys->zap_ptrtbl.zt_blk = newblk;
352                 zap->zap_f_phys->zap_ptrtbl.zt_numblks = 1;
353                 zap->zap_f_phys->zap_ptrtbl.zt_shift++;
355                 zap->zap_f.zap_phys->zap_ptrtbl.zt_blk = newblk;
356                 zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks = 1;
357                 zap->zap_f.zap_phys->zap_ptrtbl.zt_shift++;

355                 ASSERT3U(1ULL << zap->zap_f_phys->zap_ptrtbl.zt_shift, ==,
356                     zap->zap_f_phys->zap_ptrtbl.zt_numblks <<
359                 ASSERT3U(1ULL << zap->zap_f.zap_phys->zap_ptrtbl.zt_shift, ==,
360                     zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks <<
357                     (FZAP_BLOCK_SHIFT(zap)-3));

359                 return (0);
360         } else {
361                 return (zap_table_grow(zap, &zap->zap_f_phys->zap_ptrtbl,
365                 return (zap_table_grow(zap, &zap->zap_f.zap_phys->zap_ptrtbl,
362                     zap_ptrtbl_transfer, tx));
363         }
364 }
```



```
366 static void
367 zap_increment_num_entries(zap_t *zap, int delta, dmu_tx_t *tx)
368 {
369         dmu_buf_will_dirty(zap->zap_dbuf, tx);
370         mutex_enter(&zap->zap_f.zap_num_entries_mtx);
371         ASSERT(delta > 0 || zap->zap_f_phys->zap_num_entries >= -delta);
372         zap->zap_f_phys->zap_num_entries += delta;
375         ASSERT(delta > 0 || zap->zap_f.zap_phys->zap_num_entries >= -delta);
376         zap->zap_f.zap_phys->zap_num_entries += delta;
373         mutex_exit(&zap->zap_f.zap_num_entries_mtx);
374 }

376 static uint64_t
377 zap_allocate_blocks(zap_t *zap, int nblocks)
378 {
379         uint64_t newblk;
380         ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
381         newblk = zap->zap_f_phys->zap_freeblk;
382         zap->zap_f_phys->zap_freeblk += nblocks;
385         newblk = zap->zap_f.zap_phys->zap_freeblk;
386         zap->zap_f.zap_phys->zap_freeblk += nblocks;
383         return (newblk);
384 }

386 static void
387 zap_leaf_pageout(dmu_buf_user_t *dbu)
388 {
389         zap_leaf_t *l = (zap_leaf_t *)dbu;

391         rw_destroy(&l->l_rwlock);
392         kmem_free(l, sizeof (zap_leaf_t));
393 }

395 #endif /* ! codereview */
396 static zap_leaf_t *
397 zap_create_leaf(zap_t *zap, dmu_tx_t *tx)
398 {
399         void *winner;
400         zap_leaf_t *l = kmem_alloc(sizeof (zap_leaf_t), KM_SLEEP);

402         ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));

404         rw_init(&l->l_rwlock, 0, 0, 0);
405         rw_enter(&l->l_rwlock, RW_WRITER);
406         l->l_blkid = zap_allocate_blocks(zap, 1);
407         l->l_dbuf = NULL;
390         l->l_phys = NULL;

409         VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
410             l->l_blkid << FZAP_BLOCK_SHIFT(zap), NULL, &l->l_dbuf,
411             DMU_READ_NO_PREFETCH));
412         dmu_buf_init_user(&l->db_evict, zap_leaf_pageout);
413         winner = (zap_leaf_t *)dmu_buf_set_user(l->l_dbuf, &l->db_evict);
395         winner = dmu_buf_set_user(l->l_dbuf, l, &l->l_phys, zap_leaf_pageout);
414         ASSERT(winner == NULL);
415         dmu_buf_will_dirty(l->l_dbuf, tx);

417         zap_leaf_init(l, zap->zap_normflags != 0);

419         zap->zap_f_phys->zap_num_leafs++;
401         zap->zap_f.zap_phys->zap_num_leafs++;

421         return (l);
422 }
```

```
424 int
425 fzap_count(zap_t *zap, uint64_t *count)
426 {
427         ASSERT(!zap->zap_ismicro);
428         mutex_enter(&zap->zap_f.zap_num_entries_mtx); /* unnecessary */
429         *count = zap->zap_f_phys->zap_num_entries;
411         *count = zap->zap_f.zap_phys->zap_num_entries;
430         mutex_exit(&zap->zap_f.zap_num_entries_mtx);
431         return (0);
432 }
```

unchanged_portion_omitted_

```
427 _NOTE(ARGSUSED(0))
428 static void
429 zap_leaf_pageout(dmu_buf_t *db, void *vl)
430 {
431         zap_leaf_t *l = vl;

433         rw_destroy(&l->l_rwlock);
434         kmem_free(l, sizeof (zap_leaf_t));
435 }

445 static zap_leaf_t *
446 zap_open_leaf(uint64_t blkid, dmu_buf_t *db)
447 {
448         zap_leaf_t *l, *winner;

450         ASSERT(blkid != 0);

452         l = kmem_alloc(sizeof (zap_leaf_t), KM_SLEEP);
453         rw_init(&l->l_rwlock, 0, 0, 0);
454         rw_enter(&l->l_rwlock, RW_WRITER);
455         l->l_blkid = blkid;
456         l->l_bs = highbit(db->db_size)-1;
457         l->l_dbuf = db;
450         l->l_phys = NULL;

459         dmu_buf_init_user(&l->db_evict, zap_leaf_pageout);
460         winner = (zap_leaf_t *)dmu_buf_set_user(db, &l->db_evict);
452         winner = dmu_buf_set_user(db, l, &l->l_phys, zap_leaf_pageout);

462         rw_exit(&l->l_rwlock);
463         if (winner != NULL) {
464                 /* someone else set it first */
465                 zap_leaf_pageout(&l->db_evict);
457                 zap_leaf_pageout(NULL, l);
466                 l = winner;
467         }

469         /*
470          * lhr_pad was previously used for the next leaf in the leaf
471          * chain.  There should be no chained leafs (as we have removed
472          * support for them).
473          */
474         ASSERT0(l->l_phys->l_hdr.lh_pad1);

476         /*
477          * There should be more hash entries than there can be
478          * chunks to put in the hash table
479          */
480         ASSERT3U(ZAP_LEAF_HASH_NUMENTRIES(l), >, ZAP_LEAF_NUMCHUNKS(l) / 3);

482         /* The chunks should begin at the end of the hash table */
483         ASSERT3P(&ZAP_LEAF_CHUNK(l, 0), ==,
484             &l->l_phys->l_hash[ZAP_LEAF_HASH_NUMENTRIES(l)]);
```



```
486         /* The chunks should end at the end of the block */
487         ASSERT3U((uintptr_t)&ZAP_LEAF_CHUNK(l, ZAP_LEAF_NUMCHUNKS(l)) -
488             (uintptr_t)l->l_phys, ==, l->l_dbuf->db_size);

490         return (l);
491 }

493 static int
494 zap_get_leaf_byblk(zap_t *zap, uint64_t blkid, dmu_tx_t *tx, krw_t lt,
495     zap_leaf_t **lp)
496 {
497         dmu_buf_t *db;
498         zap_leaf_t *l;
499         int bs = FZAP_BLOCK_SHIFT(zap);
500         int err;

502         ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));

504         err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
505             blkid << bs, NULL, &db, DMU_READ_NO_PREFETCH);
506         if (err)
507                 return (err);

509         ASSERT3U(db->db_object, ==, zap->zap_object);
510         ASSERT3U(db->db_offset, ==, blkid << bs);
511         ASSERT3U(db->db_size, ==, 1 << bs);
512         ASSERT(blkid != 0);

514         l = (zap_leaf_t *)dmu_buf_get_user(db);
506         l = dmu_buf_get_user(db);

516         if (l == NULL)
517                 l = zap_open_leaf(blkid, db);

519         rw_enter(&l->l_rwlock, lt);
520         /*
521          * Must lock before dirtying, otherwise l->l_phys could change,
522          * causing ASSERT below to fail.
523          */
524         if (lt == RW_WRITER)
525                 dmu_buf_will_dirty(db, tx);
526         ASSERT3U(l->l_blkid, ==, blkid);
527         ASSERT3P(l->l_dbuf, ==, db);
528         ASSERT3P(l->l_phys, ==, l->l_dbuf->db_data);
529         ASSERT3U(l->l_phys->l_hdr.lh_block_type, ==, ZBT_LEAF);
530         ASSERT3U(l->l_phys->l_hdr.lh_magic, ==, ZAP_LEAF_MAGIC);

532         *lp = l;
533         return (0);
534 }

536 static int
537 zap_idx_to_blk(zap_t *zap, uint64_t idx, uint64_t *valp)
538 {
539         ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));

541         if (zap->zap_f_phys->zap_ptrtbl.zt_numblks == 0) {
533         if (zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks == 0) {
542                 ASSERT3U(idx, <,
543                     (1ULL << zap->zap_f_phys->zap_ptrtbl.zt_shift));
535                     (1ULL << zap->zap_f.zap_phys->zap_ptrtbl.zt_shift));
544                 *valp = ZAP_EMBEDDED_PTRTBL_ENT(zap, idx);
545                 return (0);
546         } else {
547                 return (zap_table_load(zap, &zap->zap_f_phys->zap_ptrtbl,
539                 return (zap_table_load(zap, &zap->zap_f.zap_phys->zap_ptrtbl,
```

```
548                     idx, valp));
549         }
550 }

552 static int
553 zap_set_idx_to_blk(zap_t *zap, uint64_t idx, uint64_t blk, dmu_tx_t *tx)
554 {
555         ASSERT(tx != NULL);
556         ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));

558         if (zap->zap_f_phys->zap_ptrtbl.zt_blk == 0) {
550         if (zap->zap_f.zap_phys->zap_ptrtbl.zt_blk == 0) {
559                 ZAP_EMBEDDED_PTRTBL_ENT(zap, idx) = blk;
560                 return (0);
561         } else {
562                 return (zap_table_store(zap, &zap->zap_f_phys->zap_ptrtbl,
554                 return (zap_table_store(zap, &zap->zap_f.zap_phys->zap_ptrtbl,
563                     idx, blk, tx));
564         }
565 }

567 static int
568 zap_deref_leaf(zap_t *zap, uint64_t h, dmu_tx_t *tx, krw_t lt, zap_leaf_t **lp)
569 {
570         uint64_t idx, blk;
571         int err;

573         ASSERT(zap->zap_dbuf == NULL ||
574             zap->zap_f_phys == zap->zap_dbuf->db_data);
575         ASSERT3U(zap->zap_f_phys->zap_magic, ==, ZAP_MAGIC);
576         idx = ZAP_HASH_IDX(h, zap->zap_f_phys->zap_ptrtbl.zt_shift);
566             zap->zap_f.zap_phys == zap->zap_dbuf->db_data);
567         ASSERT3U(zap->zap_f.zap_phys->zap_magic, ==, ZAP_MAGIC);
568         idx = ZAP_HASH_IDX(h, zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
577         err = zap_idx_to_blk(zap, idx, &blk);
578         if (err != 0)
579                 return (err);
580         err = zap_get_leaf_byblk(zap, blk, tx, lt, lp);

582         ASSERT(err || ZAP_HASH_IDX(h, (*lp)->l_phys->l_hdr.lh_prefix_len) ==
583             (*lp)->l_phys->l_hdr.lh_prefix);
584         return (err);
585 }

587 static int
588 zap_expand_leaf(zap_name_t *zn, zap_leaf_t *l, dmu_tx_t *tx, zap_leaf_t **lp)
589 {
590         zap_t *zap = zn->zn_zap;
591         uint64_t hash = zn->zn_hash;
592         zap_leaf_t *nl;
593         int prefix_diff, i, err;
594         uint64_t sibling;
595         int old_prefix_len = l->l_phys->l_hdr.lh_prefix_len;

597         ASSERT3U(old_prefix_len, <=, zap->zap_f_phys->zap_ptrtbl.zt_shift);
589         ASSERT3U(old_prefix_len, <=, zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
598         ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));

600         ASSERT3U(ZAP_HASH_IDX(hash, old_prefix_len), ==,
601             l->l_phys->l_hdr.lh_prefix);

603         if (zap_tryupgradedir(zap, tx) == 0 ||
604             old_prefix_len == zap->zap_f_phys->zap_ptrtbl.zt_shift) {
596             old_prefix_len == zap->zap_f.zap_phys->zap_ptrtbl.zt_shift) {
605                 /* We failed to upgrade, or need to grow the pointer table */
606                 objset_t *os = zap->zap_objset;
```



```
607                 uint64_t object = zap->zap_object;

609                 zap_put_leaf(l);
610                 zap_unlockdir(zap);
611                 err = zap_lockdir(os, object, tx, RW_WRITER,
612                     FALSE, FALSE, &zn->zn_zap);
613                 zap = zn->zn_zap;
614                 if (err)
615                         return (err);
616                 ASSERT(!zap->zap_ismicro);

618                 while (old_prefix_len ==
619                     zap->zap_f_phys->zap_ptrtbl.zt_shift) {
611                     zap->zap_f.zap_phys->zap_ptrtbl.zt_shift) {
620                         err = zap_grow_ptrtbl(zap, tx);
621                         if (err)
622                                 return (err);
623                 }

625                 err = zap_deref_leaf(zap, hash, tx, RW_WRITER, &l);
626                 if (err)
627                         return (err);

629                 if (l->l_phys->l_hdr.lh_prefix_len != old_prefix_len) {
630                         /* it split while our locks were down */
631                         *lp = l;
632                         return (0);
633                 }
634         }
635         ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
636         ASSERT3U(old_prefix_len, <, zap->zap_f_phys->zap_ptrtbl.zt_shift);
628         ASSERT3U(old_prefix_len, <, zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
637         ASSERT3U(ZAP_HASH_IDX(hash, old_prefix_len), ==,
638             l->l_phys->l_hdr.lh_prefix);

640         prefix_diff = zap->zap_f_phys->zap_ptrtbl.zt_shift -
632         prefix_diff = zap->zap_f.zap_phys->zap_ptrtbl.zt_shift -
641             (old_prefix_len + 1);
642         sibling = (ZAP_HASH_IDX(hash, old_prefix_len + 1) | 1) << prefix_diff;

644         /* check for i/o errors before doing zap_leaf_split */
645         for (i = 0; i < (1ULL<<prefix_diff); i++) {
646                 uint64_t blk;
647                 err = zap_idx_to_blk(zap, sibling+i, &blk);
648                 if (err)
649                         return (err);
650                 ASSERT3U(blk, ==, l->l_blkid);
651         }

653         nl = zap_create_leaf(zap, tx);
654         zap_leaf_split(l, nl, zap->zap_normflags != 0);

656         /* set sibling pointers */
657         for (i = 0; i < (1ULL << prefix_diff); i++) {
658                 err = zap_set_idx_to_blk(zap, sibling+i, nl->l_blkid, tx);
659                 ASSERT0(err); /* we checked for i/o errors above */
660         }

662         if (hash & (1ULL << (64 - l->l_phys->l_hdr.lh_prefix_len))) {
663                 /* we want the sibling */
664                 zap_put_leaf(l);
665                 *lp = nl;
666         } else {
667                 zap_put_leaf(nl);
668                 *lp = l;
669         }
```

```
 671         return (0);
 672 }

 674 static void
 675 zap_put_leaf_maybe_grow_ptrtbl(zap_name_t *zn, zap_leaf_t *l, dmu_tx_t *tx)
 676 {
 677         zap_t *zap = zn->zn_zap;
 678         int shift = zap->zap_f_phys->zap_ptrtbl.zt_shift;
 670         int shift = zap->zap_f.zap_phys->zap_ptrtbl.zt_shift;
 679         int leaffull = (l->l_phys->l_hdr.lh_prefix_len == shift &&
 680             l->l_phys->l_hdr.lh_nfree < ZAP_LEAF_LOW_WATER);

 682         zap_put_leaf(l);

 684         if (leaffull || zap->zap_f_phys->zap_ptrtbl.zt_nextblk) {
 676         if (leaffull || zap->zap_f.zap_phys->zap_ptrtbl.zt_nextblk) {
 685                 int err;

 687                 /*
 688                  * We are in the middle of growing the pointer table, or
 689                  * this leaf will soon make us grow it.
 690                  */
 691                 if (zap_tryupgradedir(zap, tx) == 0) {
 692                         objset_t *os = zap->zap_objset;
 693                         uint64_t zapobj = zap->zap_object;

 695                         zap_unlockdir(zap);
 696                         err = zap_lockdir(os, zapobj, tx,
 697                             RW_WRITER, FALSE, FALSE, &zn->zn_zap);
 698                         zap = zn->zn_zap;
 699                         if (err)
 700                                 return;
 701                 }

 703                 /* could have finished growing while our locks were down */
 704                 if (zap->zap_f_phys->zap_ptrtbl.zt_shift == shift)
 696                 if (zap->zap_f.zap_phys->zap_ptrtbl.zt_shift == shift)
 705                         (void) zap_grow_ptrtbl(zap, tx);
 706         }
 707 }
```

*unchanged_portion_omitted_*

```
 927 void
 928 fzap_prefetch(zap_name_t *zn)
 929 {
 930         uint64_t idx, blk;
 931         zap_t *zap = zn->zn_zap;
 932         int bs;

 934         idx = ZAP_HASH_IDX(zn->zn_hash,
 935             zap->zap_f_phys->zap_ptrtbl.zt_shift);
 927             zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
 936         if (zap_idx_to_blk(zap, idx, &blk) != 0)
 937                 return;
 938         bs = FZAP_BLOCK_SHIFT(zap);
 939         dmu_prefetch(zap->zap_objset, zap->zap_object, blk << bs, 1 << bs);
 940 }
```

*unchanged_portion_omitted_*

```
1266 void
1267 fzap_get_stats(zap_t *zap, zap_stats_t *zs)
1268 {
1269         int bs = FZAP_BLOCK_SHIFT(zap);
1270         zs->zs_blocksize = 1ULL << bs;
```



```
1272         /*
1273          * Set zap_phys_t fields
1274          */
1275         zs->zs_num_leafs = zap->zap_f_phys->zap_num_leafs;
1276         zs->zs_num_entries = zap->zap_f_phys->zap_num_entries;
1277         zs->zs_num_blocks = zap->zap_f_phys->zap_freeblk;
1278         zs->zs_block_type = zap->zap_f_phys->zap_block_type;
1279         zs->zs_magic = zap->zap_f_phys->zap_magic;
1280         zs->zs_salt = zap->zap_f_phys->zap_salt;
1267         zs->zs_num_leafs = zap->zap_f.zap_phys->zap_num_leafs;
1268         zs->zs_num_entries = zap->zap_f.zap_phys->zap_num_entries;
1269         zs->zs_num_blocks = zap->zap_f.zap_phys->zap_freeblk;
1270         zs->zs_block_type = zap->zap_f.zap_phys->zap_block_type;
1271         zs->zs_magic = zap->zap_f.zap_phys->zap_magic;
1272         zs->zs_salt = zap->zap_f.zap_phys->zap_salt;

1282         /*
1283          * Set zap_ptrtbl fields
1284          */
1285         zs->zs_ptrtbl_len = 1ULL << zap->zap_f_phys->zap_ptrtbl.zt_shift;
1286         zs->zs_ptrtbl_nextblk = zap->zap_f_phys->zap_ptrtbl.zt_nextblk;
1277         zs->zs_ptrtbl_len = 1ULL << zap->zap_f.zap_phys->zap_ptrtbl.zt_shift;
1278         zs->zs_ptrtbl_nextblk = zap->zap_f.zap_phys->zap_ptrtbl.zt_nextblk;
1287         zs->zs_ptrtbl_blks_copied =
1288             zap->zap_f_phys->zap_ptrtbl.zt_blks_copied;
1289         zs->zs_ptrtbl_zt_blk = zap->zap_f_phys->zap_ptrtbl.zt_blk;
1290         zs->zs_ptrtbl_zt_numblks = zap->zap_f_phys->zap_ptrtbl.zt_numblks;
1291         zs->zs_ptrtbl_zt_shift = zap->zap_f_phys->zap_ptrtbl.zt_shift;
1280             zap->zap_f.zap_phys->zap_ptrtbl.zt_blks_copied;
1281         zs->zs_ptrtbl_zt_blk = zap->zap_f.zap_phys->zap_ptrtbl.zt_blk;
1282         zs->zs_ptrtbl_zt_numblks = zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks;
1283         zs->zs_ptrtbl_zt_shift = zap->zap_f.zap_phys->zap_ptrtbl.zt_shift;

1293         if (zap->zap_f_phys->zap_ptrtbl.zt_numblks == 0) {
1285         if (zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks == 0) {
1294                 /* the ptrtbl is entirely in the header block. */
1295                 zap_stats_ptrtbl(zap, &ZAP_EMBEDDED_PTRTBL_ENT(zap, 0),
1296                     1 << ZAP_EMBEDDED_PTRTBL_SHIFT(zap), zs);
1297         } else {
1298                 int b;

1300                 dmu_prefetch(zap->zap_objset, zap->zap_object,
1301                     zap->zap_f_phys->zap_ptrtbl.zt_blk << bs,
1302                     zap->zap_f_phys->zap_ptrtbl.zt_numblks << bs);
1293                     zap->zap_f.zap_phys->zap_ptrtbl.zt_blk << bs,
1294                     zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks << bs);

1304                 for (b = 0; b < zap->zap_f_phys->zap_ptrtbl.zt_numblks;
1296                 for (b = 0; b < zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks;
1305                     b++) {
1306                         dmu_buf_t *db;
1307                         int err;

1309                         err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
1310                             (zap->zap_f_phys->zap_ptrtbl.zt_blk + b) << bs,
1302                             (zap->zap_f.zap_phys->zap_ptrtbl.zt_blk + b) << bs,
1311                             FTAG, &db, DMU_READ_NO_PREFETCH);
1312                         if (err == 0) {
1313                                 zap_stats_ptrtbl(zap, db->db_data,
1314                                     1<<(bs-3), zs);
1315                                 dmu_buf_rele(db, FTAG);
1316                         }
1317                 }
1318         }
1319 }
```

```
1321 int
1322 fzap_count_write(zap_name_t *zn, int add, uint64_t *towrite,
1323     uint64_t *tooverwrite)
1324 {
1325         zap_t *zap = zn->zn_zap;
1326         zap_leaf_t *l;
1327         int err;

1329         /*
1330          * Account for the header block of the fatzap.
1331          */
1332         if (!add && dmu_buf_freeable(zap->zap_dbuf)) {
1333                 *tooverwrite += zap->zap_dbuf->db_size;
1334         } else {
1335                 *towrite += zap->zap_dbuf->db_size;
1336         }

1338         /*
1339          * Account for the pointer table blocks.
1340          * If we are adding we need to account for the following cases :
1341          * - If the pointer table is embedded, this operation could force an
1342          *   external pointer table.
1343          * - If this already has an external pointer table this operation
1344          *   could extend the table.
1345          */
1346         if (add) {
1347                 if (zap->zap_f_phys->zap_ptrtbl.zt_blk == 0)
1339                 if (zap->zap_f.zap_phys->zap_ptrtbl.zt_blk == 0)
1348                         *towrite += zap->zap_dbuf->db_size;
1349                 else
1350                         *towrite += (zap->zap_dbuf->db_size * 3);
1351         }

1353         /*
1354          * Now, check if the block containing leaf is freeable
1355          * and account accordingly.
1356          */
1357         err = zap_deref_leaf(zap, zn->zn_hash, NULL, RW_READER, &l);
1358         if (err != 0) {
1359                 return (err);
1360         }

1362         if (!add && dmu_buf_freeable(l->l_dbuf)) {
1363                 *tooverwrite += l->l_dbuf->db_size;
1364         } else {
1365                 /*
1366                  * If this an add operation, the leaf block could split.
1367                  * Hence, we need to account for an additional leaf block.
1368                  */
1369                 *towrite += (add ? 2 : 1) * l->l_dbuf->db_size;
1370         }

1372         zap_put_leaf(l);
1373         return (0);
1374 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   23138 Thu Apr 25 16:15:02 2013
new/usr/src/uts/common/fs/zfs/zap_leaf.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_

 103 void
 104 zap_leaf_byteswap(zap_leaf_phys_t *buf, int size)
 105 {
 106         int i;
 107         zap_leaf_t l = { 0 };

 107         zap_leaf_t l;
 109         l.l_bs = highbit(size)-1;
 109         l.l_phys = buf;

 111         buf->l_hdr.lh_block_type =      BSWAP_64(buf->l_hdr.lh_block_type);
 112         buf->l_hdr.lh_prefix =          BSWAP_64(buf->l_hdr.lh_prefix);
 113         buf->l_hdr.lh_magic =           BSWAP_32(buf->l_hdr.lh_magic);
 114         buf->l_hdr.lh_nfree =           BSWAP_16(buf->l_hdr.lh_nfree);
 115         buf->l_hdr.lh_nentries =        BSWAP_16(buf->l_hdr.lh_nentries);
 116         buf->l_hdr.lh_prefix_len =      BSWAP_16(buf->l_hdr.lh_prefix_len);
 117         buf->l_hdr.lh_freelist =        BSWAP_16(buf->l_hdr.lh_freelist);

 119         for (i = 0; i < ZAP_LEAF_HASH_NUMENTRIES(&l); i++)
 120                 buf->l_hash[i] = BSWAP_16(buf->l_hash[i]);

 122         for (i = 0; i < ZAP_LEAF_NUMCHUNKS(&l); i++) {
 123                 zap_leaf_chunk_t *lc = &ZAP_LEAF_CHUNK(&l, i);
 124                 struct zap_leaf_entry *le;

 126                 switch (lc->l_free.lf_type) {
 127                 case ZAP_CHUNK_ENTRY:
 128                         le = &lc->l_entry;

 130                         le->le_type =           BSWAP_8(le->le_type);
 131                         le->le_value_intlen =   BSWAP_8(le->le_value_intlen);
 132                         le->le_next =           BSWAP_16(le->le_next);
 133                         le->le_name_chunk =     BSWAP_16(le->le_name_chunk);
 134                         le->le_name_numints =   BSWAP_16(le->le_name_numints);
 135                         le->le_value_chunk =    BSWAP_16(le->le_value_chunk);
 136                         le->le_value_numints =  BSWAP_16(le->le_value_numints);
 137                         le->le_cd =             BSWAP_32(le->le_cd);
 138                         le->le_hash =           BSWAP_64(le->le_hash);
 139                         break;
 140                 case ZAP_CHUNK_FREE:
 141                         lc->l_free.lf_type =    BSWAP_8(lc->l_free.lf_type);
 142                         lc->l_free.lf_next =    BSWAP_16(lc->l_free.lf_next);
 143                         break;
 144                 case ZAP_CHUNK_ARRAY:
 145                         lc->l_array.la_type =   BSWAP_8(lc->l_array.la_type);
 146                         lc->l_array.la_next =   BSWAP_16(lc->l_array.la_next);
 147                         /* la_array doesn't need swapping */
 148                         break;
 149                 default:
 150                         ASSERT(!"bad leaf type");
 151                 }
 152         }
 153 }
______unchanged_portion_omitted_

 830 void
 831 zap_leaf_stats(zap_t *zap, zap_leaf_t *l, zap_stats_t *zs)
```



```
 832 {
 833         int i, n;

 835         n = zap->zap_f_phys->zap_ptrtbl.zt_shift -
 835         n = zap->zap_f.zap_phys->zap_ptrtbl.zt_shift -
 836             l->l_phys->l_hdr.lh_prefix_len;
 837         n = MIN(n, ZAP_HISTOGRAM_SIZE-1);
 838         zs->zs_leafs_with_2n_pointers[n]++;


 841         n = l->l_phys->l_hdr.lh_nentries/5;
 842         n = MIN(n, ZAP_HISTOGRAM_SIZE-1);
 843         zs->zs_blocks_with_n5_entries[n]++;

 845         n = ((1<<FZAP_BLOCK_SHIFT(zap)) -
 846             l->l_phys->l_hdr.lh_nfree * (ZAP_LEAF_ARRAY_BYTES+1))*10 /
 847             (1<<FZAP_BLOCK_SHIFT(zap));
 848         n = MIN(n, ZAP_HISTOGRAM_SIZE-1);
 849         zs->zs_blocks_n_tenths_full[n]++;

 851         for (i = 0; i < ZAP_LEAF_HASH_NUMENTRIES(l); i++) {
 852                 int nentries = 0;
 853                 int chunk = l->l_phys->l_hash[i];

 855                 while (chunk != CHAIN_END) {
 856                         struct zap_leaf_entry *le =
 857                             ZAP_LEAF_ENTRY(l, chunk);

 859                         n = 1 + ZAP_LEAF_ARRAY_NCHUNKS(le->le_name_numints) +
 860                             ZAP_LEAF_ARRAY_NCHUNKS(le->le_value_numints *
 861                             le->le_value_intlen);
 862                         n = MIN(n, ZAP_HISTOGRAM_SIZE-1);
 863                         zs->zs_entries_using_n_chunks[n]++;

 865                         chunk = le->le_next;
 866                         nentries++;
 867                 }

 869                 n = nentries;
 870                 n = MIN(n, ZAP_HISTOGRAM_SIZE-1);
 871                 zs->zs_buckets_with_n_entries[n]++;
 872         }
 873 }
______unchanged_portion_omitted_
```

```
**********************************************************
   34972 Thu Apr 25 16:15:02 2013
new/usr/src/uts/common/fs/zfs/zap_micro.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
```

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

#include <sys/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>

#ifdef _KERNEL
#include <sys/sunddi.h>
#endif

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->zap_f_phys->zap_flags);
        return (zap->zap_u.zap_fat.zap_phys->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 = highbit(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->db_evict, zap_evict);
        winner = (zap_t *)dmu_buf_set_user(db, &zap->db_evict);
        winner = dmu_buf_set_user(db, zap, &zap->zap_m.zap_phys, 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->zap_m_phys->mz_salt;
                zap->zap_normflags = zap->zap_m_phys->mz_normflags;
                zap->zap_salt = zap->zap_m.zap_phys->mz_salt;
                zap->zap_normflags = zap->zap_m.zap_phys->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->zap_m_phys->mz_chunk[i];
                            &zap->zap_m.zap_phys->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->zap_f_phys->zap_salt;
                zap->zap_normflags = zap->zap_f_phys->zap_normflags;
                zap->zap_salt = zap->zap_f.zap_phys->zap_salt;
                zap->zap_normflags = zap->zap_f.zap_phys->zap_normflags;

                ASSERT3U(sizeof (struct zap_leaf_header), ==,
                    2*ZAP_LEAF_CHUNKSIZE);
```

```
426                 /*
427                  * The embedded pointer table should not overlap the
428                  * other members.
429                  */
430                 ASSERT3P(&ZAP_EMBEDDED_PTRTBL_ENT(zap, 0), >,
431                     &zap->zap_f_phys->zap_salt);
430                     &zap->zap_f.zap_phys->zap_salt);

433                 /*
434                  * The embedded pointer table should end at the end of
435                  * the block
436                  */
437                 ASSERT3U((uintptr_t)&ZAP_EMBEDDED_PTRTBL_ENT(zap,
438                     1<<ZAP_EMBEDDED_PTRTBL_SHIFT(zap)) -
439                     (uintptr_t)zap->zap_f_phys, ==,
438                     (uintptr_t)zap->zap_f.zap_phys, ==,
440                     zap->zap_dbuf->db_size);
441         }
442         rw_exit(&zap->zap_rwlock);
443         return (zap);
444 }

446 int
447 zap_lockdir(objset_t *os, uint64_t obj, dmu_tx_t *tx,
448     krw_t lti, boolean_t fatreader, boolean_t adding, zap_t **zapp)
449 {
450         zap_t *zap;
451         dmu_buf_t *db;
452         krw_t lt;
453         int err;

455         *zapp = NULL;

457         err = dmu_buf_hold(os, obj, 0, NULL, &db, DMU_READ_NO_PREFETCH);
458         if (err)
459                 return (err);

461 #ifdef ZFS_DEBUG
462         {
463                 dmu_object_info_t doi;
464                 dmu_object_info_from_db(db, &doi);
465                 ASSERT3U(DMU_OT_BYTESWAP(doi.doi_type), ==, DMU_BSWAP_ZAP);
466         }
467 #endif

469         zap = (zap_t *)dmu_buf_get_user(db);
468         zap = dmu_buf_get_user(db);
470         if (zap == NULL)
471                 zap = mzap_open(os, obj, db);

473         /*
474          * We're checking zap_ismicro without the lock held, in order to
475          * tell what type of lock we want.  Once we have some sort of
476          * lock, see if it really is the right type.  In practice this
477          * can only be different if it was upgraded from micro to fat,
478          * and micro wanted WRITER but fat only needs READER.
479          */
480         lt = (!zap->zap_ismicro && fatreader) ? RW_READER : lti;
481         rw_enter(&zap->zap_rwlock, lt);
482         if (lt != ((!zap->zap_ismicro && fatreader) ? RW_READER : lti)) {
483                 /* it was upgraded, now we only need reader */
484                 ASSERT(lt == RW_WRITER);
485                 ASSERT(RW_READER ==
486                     (!zap->zap_ismicro && fatreader) ? RW_READER : lti);
487                 rw_downgrade(&zap->zap_rwlock);
488                 lt = RW_READER;
```



```
489         }

491         zap->zap_objset = os;

493         if (lt == RW_WRITER)
494                 dmu_buf_will_dirty(db, tx);

496         ASSERT3P(zap->zap_dbuf, ==, db);

498         ASSERT(!zap->zap_ismicro ||
499             zap->zap_m.zap_num_entries <= zap->zap_m.zap_num_chunks);
500         if (zap->zap_ismicro && tx && adding &&
501             zap->zap_m.zap_num_entries == zap->zap_m.zap_num_chunks) {
502                 uint64_t newsz = db->db_size + SPA_MINBLOCKSIZE;
503                 if (newsz > MZAP_MAX_BLKSZ) {
504                         dprintf("upgrading obj %llu: num_entries=%u\n",
505                             obj, zap->zap_m.zap_num_entries);
506                         *zapp = zap;
507                         return (mzap_upgrade(zapp, tx, 0));
508                 }
509                 err = dmu_object_set_blocksize(os, obj, newsz, 0, tx);
510                 ASSERT0(err);
511                 zap->zap_m.zap_num_chunks =
512                     db->db_size / MZAP_ENT_LEN - 1;
513         }

515         *zapp = zap;
516         return (0);
517 }
```

_____unchanged_portion_omitted_

```
679 /* ARGSUSED */
680 void
681 zap_evict(dmu_buf_user_t *dbu)
681 zap_evict(dmu_buf_t *db, void *vzap)
682 {
683         zap_t *zap = (zap_t *)dbu;
683         zap_t *zap = vzap;

685         rw_destroy(&zap->zap_rwlock);

687         if (zap->zap_ismicro)
688                 mze_destroy(zap);
689         else
690                 mutex_destroy(&zap->zap_f.zap_num_entries_mtx);

692         kmem_free(zap, sizeof (zap_t));
693 }
```

_____unchanged_portion_omitted_

```
929 static void
930 mzap_addent(zap_name_t *zn, uint64_t value)
931 {
932         int i;
933         zap_t *zap = zn->zn_zap;
934         int start = zap->zap_m.zap_alloc_next;
935         uint32_t cd;

937         ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));

939 #ifdef ZFS_DEBUG
940         for (i = 0; i < zap->zap_m.zap_num_chunks; i++) {
941                 mzap_ent_phys_t *mze = &zap->zap_m_phys->mz_chunk[i];
941                 mzap_ent_phys_t *mze = &zap->zap_m.zap_phys->mz_chunk[i];
942                 ASSERT(strcmp(zn->zn_key_orig, mze->mze_name) != 0);
943         }
```

```
 944 #endif

 946         cd = mze_find_unused_cd(zap, zn->zn_hash);
 947         /* given the limited size of the microzap, this can't happen */
 948         ASSERT(cd < zap_maxcd(zap));

 950 again:
 951         for (i = start; i < zap->zap_m.zap_num_chunks; i++) {
 952                 mzap_ent_phys_t *mze = &zap->zap_m_phys->mz_chunk[i];
 952                 mzap_ent_phys_t *mze = &zap->zap_m.zap_phys->mz_chunk[i];
 953                 if (mze->mze_name[0] == 0) {
 954                         mze->mze_value = value;
 955                         mze->mze_cd = cd;
 956                         (void) strcpy(mze->mze_name, zn->zn_key_orig);
 957                         zap->zap_m.zap_num_entries++;
 958                         zap->zap_m.zap_alloc_next = i+1;
 959                         if (zap->zap_m.zap_alloc_next ==
 960                             zap->zap_m.zap_num_chunks)
 961                                 zap->zap_m.zap_alloc_next = 0;
 962                         mze_insert(zap, i, zn->zn_hash);
 963                         return;
 964                 }
 965         }
 966         if (start != 0) {
 967                 start = 0;
 968                 goto again;
 969         }
 970         ASSERT(!"out of entries!");
 971 }
_____unchanged_portion_omitted_

1128 int
1129 zap_remove_norm(objset_t *os, uint64_t zapobj, const char *name,
1130     matchtype_t mt, dmu_tx_t *tx)
1131 {
1132         zap_t *zap;
1133         int err;
1134         mzap_ent_t *mze;
1135         zap_name_t *zn;

1137         err = zap_lockdir(os, zapobj, tx, RW_WRITER, TRUE, FALSE, &zap);
1138         if (err)
1139                 return (err);
1140         zn = zap_name_alloc(zap, name, mt);
1141         if (zn == NULL) {
1142                 zap_unlockdir(zap);
1143                 return (SET_ERROR(ENOTSUP));
1144         }
1145         if (!zap->zap_ismicro) {
1146                 err = fzap_remove(zn, tx);
1147         } else {
1148                 mze = mze_find(zn);
1149                 if (mze == NULL) {
1150                         err = SET_ERROR(ENOENT);
1151                 } else {
1152                         zap->zap_m.zap_num_entries--;
1153                         bzero(&zap->zap_m_phys->mz_chunk[mze->mze_chunkid],
1153                         bzero(&zap->zap_m.zap_phys->mz_chunk[mze->mze_chunkid],
1154                             sizeof (mzap_ent_phys_t));
1155                         mze_remove(zap, mze);
1156                 }
1157         }
1158         zap_name_free(zn);
1159         zap_unlockdir(zap);
1160         return (err);
1161 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   10563 Thu Apr 25 16:15:02 2013
new/usr/src/uts/common/fs/zfs/zfs_sa.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.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>
#endif /* ! codereview */
#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}
};

#ifdef _KERNEL

int
zfs_sa_readlink(znode_t *zp, uio_t *uio)
{
        dmu_buf_t *db = sa_get_db(zp->z_sa_hdl);
        size_t bufsz;
        int error;

        bufsz = zp->z_size;
        if (bufsz + ZFS_OLD_ZNODE_PHYS_SIZE <= db->db_size) {
                error = uiomove((caddr_t)db->db_data +
                    ZFS_OLD_ZNODE_PHYS_SIZE,
                    MIN((size_t)bufsz, uio->uio_resid), UIO_READ, uio);
        } else {
                dmu_buf_t *dbp;
                if ((error = dmu_buf_hold(zp->z_zfsvfs->z_os, zp->z_id,
                    0, FTAG, &dbp, DMU_READ_NO_PREFETCH)) == 0) {
                        error = uiomove(dbp->db_data,
                            MIN((size_t)bufsz, uio->uio_resid), UIO_READ, uio);
                        dmu_buf_rele(dbp, FTAG);
                }
        }
        return (error);
}

void
zfs_sa_symlink(znode_t *zp, char *link, int len, dmu_tx_t *tx)
{
        dmu_buf_t *db = sa_get_db(zp->z_sa_hdl);

        if (ZFS_OLD_ZNODE_PHYS_SIZE + len <= dmu_bonus_max()) {
                VERIFY(dmu_set_bonus(db,
                    len + ZFS_OLD_ZNODE_PHYS_SIZE, tx) == 0);
                if (len) {
                        bcopy(link, (caddr_t)db->db_data +
                            ZFS_OLD_ZNODE_PHYS_SIZE, len);
                }
        } else {
                dmu_buf_t *dbp;

                zfs_grow_blocksize(zp, len, tx);
                VERIFY(0 == dmu_buf_hold(zp->z_zfsvfs->z_os,
                    zp->z_id, 0, FTAG, &dbp, DMU_READ_NO_PREFETCH));

                dmu_buf_will_dirty(dbp, tx);

                ASSERT3U(len, <=, dbp->db_size);
                bcopy(link, dbp->db_data, len);
                dmu_buf_rele(dbp, FTAG);
        }
}

void
zfs_sa_get_scanstamp(znode_t *zp, xvattr_t *xvap)
```

```
126 {
127         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
128         xoptattr_t *xoap;

130         ASSERT(MUTEX_HELD(&zp->z_lock));
131         VERIFY((xoap = xva_getxoptattr(xvap)) != NULL);
132         if (zp->z_is_sa) {
133                 if (sa_lookup(zp->z_sa_hdl, SA_ZPL_SCANSTAMP(zfsvfs),
134                     &xoap->xoa_av_scanstamp,
135                     sizeof (xoap->xoa_av_scanstamp)) != 0)
136                         return;
137         } else {
138                 dmu_object_info_t doi;
139                 dmu_buf_t *db = sa_get_db(zp->z_sa_hdl);
140                 int len;

142                 if (!(zp->z_pflags & ZFS_BONUS_SCANSTAMP))
143                         return;

145                 sa_object_info(zp->z_sa_hdl, &doi);
146                 len = sizeof (xoap->xoa_av_scanstamp) +
147                     ZFS_OLD_ZNODE_PHYS_SIZE;

149                 if (len <= doi.doi_bonus_size) {
150                         (void) memcpy(xoap->xoa_av_scanstamp,
151                             (caddr_t)db->db_data + ZFS_OLD_ZNODE_PHYS_SIZE,
152                             sizeof (xoap->xoa_av_scanstamp));
153                 }
154         }
155         XVA_SET_RTN(xvap, XAT_AV_SCANSTAMP);
156 }

158 void
159 zfs_sa_set_scanstamp(znode_t *zp, xvattr_t *xvap, dmu_tx_t *tx)
160 {
161         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
162         xoptattr_t *xoap;

164         ASSERT(MUTEX_HELD(&zp->z_lock));
165         VERIFY((xoap = xva_getxoptattr(xvap)) != NULL);
166         if (zp->z_is_sa)
167                 VERIFY(0 == sa_update(zp->z_sa_hdl, SA_ZPL_SCANSTAMP(zfsvfs),
168                     &xoap->xoa_av_scanstamp,
169                     sizeof (xoap->xoa_av_scanstamp), tx));
170         else {
171                 dmu_object_info_t doi;
172                 dmu_buf_t *db = sa_get_db(zp->z_sa_hdl);
173                 int len;

175                 sa_object_info(zp->z_sa_hdl, &doi);
176                 len = sizeof (xoap->xoa_av_scanstamp) +
177                     ZFS_OLD_ZNODE_PHYS_SIZE;
178                 if (len > doi.doi_bonus_size)
179                         VERIFY(dmu_set_bonus(db, len, tx) == 0);
180                 (void) memcpy((caddr_t)db->db_data + ZFS_OLD_ZNODE_PHYS_SIZE,
181                     xoap->xoa_av_scanstamp, sizeof (xoap->xoa_av_scanstamp));

183                 zp->z_pflags |= ZFS_BONUS_SCANSTAMP;
184                 VERIFY(0 == sa_update(zp->z_sa_hdl, SA_ZPL_FLAGS(zfsvfs),
185                     &zp->z_pflags, sizeof (uint64_t), tx));
186         }
187 }

189 /*
190  * I'm not convinced we should do any of this upgrade.
191  * since the SA code can read both old/new znode formats
```



```
192  * with probably little to know performance difference.
193  *
194  * All new files will be created with the new format.
195  */

197 void
198 zfs_sa_upgrade(sa_handle_t *hdl, dmu_tx_t *tx)
199 {
200         dmu_buf_t *db = sa_get_db(hdl);
201         znode_t *zp = sa_get_userdata(hdl);
202         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
203         sa_bulk_attr_t bulk[20];
204         int count = 0;
205         sa_bulk_attr_t sa_attrs[20] = { 0 };
206         zfs_acl_locator_cb_t locate = { 0 };
207         uint64_t uid, gid, mode, rdev, xattr, parent;
208         uint64_t crtime[2], mtime[2], ctime[2];
209         zfs_acl_phys_t znode_acl;
210         char scanstamp[AV_SCANSTAMP_SZ];
211         boolean_t drop_lock = B_FALSE;

213         /*
214          * No upgrade if ACL isn't cached
215          * since we won't know which locks are held
216          * and ready the ACL would require special "locked"
217          * interfaces that would be messy
218          */
219         if (zp->z_acl_cached == NULL || ZTOV(zp)->v_type == VLNK)
220                 return;

222         /*
223          * If the z_lock is held and we aren't the owner
224          * the just return since we don't want to deadlock
225          * trying to update the status of z_is_sa.  This
226          * file can then be upgraded at a later time.
227          *
228          * Otherwise, we know we are doing the
229          * sa_update() that caused us to enter this function.
230          */
231         if (mutex_owner(&zp->z_lock) != curthread) {
232                 if (mutex_tryenter(&zp->z_lock) == 0)
233                         return;
234                 else
235                         drop_lock = B_TRUE;
236         }

238         /* First do a bulk query of the attributes that aren't cached */
239         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, &mtime, 16);
240         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, &ctime, 16);
241         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CRTIME(zfsvfs), NULL, &crtime, 16);
242         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs), NULL, &mode, 8);
243         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_PARENT(zfsvfs), NULL, &parent, 8);
244         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_XATTR(zfsvfs), NULL, &xattr, 8);
245         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_RDEV(zfsvfs), NULL, &rdev, 8);
246         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_UID(zfsvfs), NULL, &uid, 8);
247         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_GID(zfsvfs), NULL, &gid, 8);
248         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_ZNODE_ACL(zfsvfs), NULL,
249             &znode_acl, 88);

251         if (sa_bulk_lookup_locked(hdl, bulk, count) != 0)
252                 goto done;


255         /*
256          * While the order here doesn't matter its best to try and organize
257          * it is such a way to pick up an already existing layout number
```

```
258          */
259         count = 0;
260         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_MODE(zfsvfs), NULL, &mode, 8);
261         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_SIZE(zfsvfs), NULL,
262             &zp->z_size, 8);
263         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_GEN(zfsvfs),
264             NULL, &zp->z_gen, 8);
265         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_UID(zfsvfs), NULL, &uid, 8);
266         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_GID(zfsvfs), NULL, &gid, 8);
267         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_PARENT(zfsvfs),
268             NULL, &parent, 8);
269         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_FLAGS(zfsvfs), NULL,
270             &zp->z_pflags, 8);
271         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_ATIME(zfsvfs), NULL,
272             zp->z_atime, 16);
273         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_MTIME(zfsvfs), NULL,
274             &mtime, 16);
275         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_CTIME(zfsvfs), NULL,
276             &ctime, 16);
277         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_CRTIME(zfsvfs), NULL,
278             &crtime, 16);
279         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_LINKS(zfsvfs), NULL,
280             &zp->z_links, 8);
281         if (zp->z_vnode->v_type == VBLK || zp->z_vnode->v_type == VCHR)
282                 SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_RDEV(zfsvfs), NULL,
283                     &rdev, 8);
284         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_DACL_COUNT(zfsvfs), NULL,
285             &zp->z_acl_cached->z_acl_count, 8);

287         if (zp->z_acl_cached->z_version < ZFS_ACL_VERSION_FUID)
288                 zfs_acl_xform(zp, zp->z_acl_cached, CRED());

290         locate.cb_aclp = zp->z_acl_cached;
291         SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_DACL_ACES(zfsvfs),
292             zfs_acl_data_locator, &locate, zp->z_acl_cached->z_acl_bytes);

294         if (xattr)
295                 SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_XATTR(zfsvfs),
296                     NULL, &xattr, 8);

298         /* if scanstamp then add scanstamp */

300         if (zp->z_pflags & ZFS_BONUS_SCANSTAMP) {
301                 bcopy((caddr_t)db->db_data + ZFS_OLD_ZNODE_PHYS_SIZE,
302                     scanstamp, AV_SCANSTAMP_SZ);
303                 SA_ADD_BULK_ATTR(sa_attrs, count, SA_ZPL_SCANSTAMP(zfsvfs),
304                     NULL, scanstamp, AV_SCANSTAMP_SZ);
305                 zp->z_pflags &= ~ZFS_BONUS_SCANSTAMP;
306         }

308         VERIFY(dmu_set_bonustype(db, DMU_OT_SA, tx) == 0);
309         VERIFY(sa_replace_all_by_template_locked(hdl, sa_attrs,
310             count, tx) == 0);
311         if (znode_acl.z_acl_extern_obj)
312                 VERIFY(0 == dmu_object_free(zfsvfs->z_os,
313                     znode_acl.z_acl_extern_obj, tx));

315         zp->z_is_sa = B_TRUE;
316 done:
317         if (drop_lock)
318                 mutex_exit(&zp->z_lock);
319 }

321 void
322 zfs_sa_upgrade_txholds(dmu_tx_t *tx, znode_t *zp)
323 {
```



```
324         if (!zp->z_zfsvfs->z_use_sa || zp->z_is_sa)
325                 return;


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

330         if (zfs_external_acl(zp)) {
331                 dmu_tx_hold_free(tx, zfs_external_acl(zp), 0,
332                     DMU_OBJECT_END);
333         }
334 }

336 #endif
```

```
**********************************************************
   53536 Thu Apr 25 16:15:03 2013
new/usr/src/uts/common/fs/zfs/zfs_znode.c
3752 want more verifiable dbuf user eviction
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Will Andrews <willa@spectralogic.com>
**********************************************************
______unchanged_portion_omitted_

1104 int
1105 zfs_zget(zfsvfs_t *zfsvfs, uint64_t obj_num, znode_t **zpp)
1106 {
1107         dmu_object_info_t doi;
1108         dmu_buf_t       *db;
1109         znode_t         *zp;
1110         int err;
1111         sa_handle_t     *hdl;

1113         *zpp = NULL;

1115         ZFS_OBJ_HOLD_ENTER(zfsvfs, obj_num);

1117         err = sa_buf_hold(zfsvfs->z_os, obj_num, NULL, &db);
1118         if (err) {
1119                 ZFS_OBJ_HOLD_EXIT(zfsvfs, obj_num);
1120                 return (err);
1121         }

1123         dmu_object_info_from_db(db, &doi);
1124         if (doi.doi_bonus_type != DMU_OT_SA &&
1125             (doi.doi_bonus_type != DMU_OT_ZNODE ||
1126             (doi.doi_bonus_type == DMU_OT_ZNODE &&
1127             doi.doi_bonus_size < sizeof (znode_phys_t)))) {
1128                 sa_buf_rele(db, NULL);
1129                 ZFS_OBJ_HOLD_EXIT(zfsvfs, obj_num);
1130                 return (SET_ERROR(EINVAL));
1131         }

1133         hdl = (sa_handle_t *)dmu_buf_get_user(db);
1133         hdl = dmu_buf_get_user(db);
1134         if (hdl != NULL) {
1135                 zp  = sa_get_userdata(hdl);


1138                 /*
1139                  * Since "SA" does immediate eviction we
1140                  * should never find a sa handle that doesn't
1141                  * know about the znode.
1142                  */

1144                 ASSERT3P(zp, !=, NULL);

1146                 mutex_enter(&zp->z_lock);
1147                 ASSERT3U(zp->z_id, ==, obj_num);
1148                 if (zp->z_unlinked) {
1149                         err = SET_ERROR(ENOENT);
1150                 } else {
1151                         VN_HOLD(ZTOV(zp));
1152                         *zpp = zp;
1153                         err = 0;
1154                 }
1155                 sa_buf_rele(db, NULL);
1156                 mutex_exit(&zp->z_lock);
1157                 ZFS_OBJ_HOLD_EXIT(zfsvfs, obj_num);
1158                 return (err);
1159         }
```



```
1161         /*
1162          * Not found create new znode/vnode
1163          * but only if file exists.
1164          *
1165          * There is a small window where zfs_vget() could
1166          * find this object while a file create is still in
1167          * progress.  This is checked for in zfs_znode_alloc()
1168          *
1169          * if zfs_znode_alloc() fails it will drop the hold on the
1170          * bonus buffer.
1171          */
1172         zp = zfs_znode_alloc(zfsvfs, db, doi.doi_data_block_size,
1173             doi.doi_bonus_type, NULL);
1174         if (zp == NULL) {
1175                 err = SET_ERROR(ENOENT);
1176         } else {
1177                 *zpp = zp;
1178         }
1179         ZFS_OBJ_HOLD_EXIT(zfsvfs, obj_num);
1180         return (err);
1181 }
______unchanged_portion_omitted_
```