```
**********************************************************
   75621 Thu Aug 15 17:43:48 2013
new/usr/src/uts/common/fs/zfs/dbuf.c
4047 panic from dbuf_free_range() from dmu_free_object() while doing zfs receive
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

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

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

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

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

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

/* ARGSUSED */
static int
```



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

        mutex_init(&db->db_mtx, NULL, MUTEX_DEFAULT, NULL);
        cv_init(&db->db_changed, NULL, CV_DEFAULT, NULL);
        refcount_create(&db->db_holds);
        return (0);
}
```

*unchanged_portion_omitted_*

```
/*
 * 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 across any level-1 dbufs in the
 * range that have not already been marked dirty, mark them dirty so
 * they stay in memory.
 *
 * This is a no-op if the dataset is in the middle of an incremental
 * receive; see comment below for details.
 */
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;

        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);
        if (start >= dn->dn_unlisted_l0_blkid * dn->dn_datablksz) {
                /* There can't be any dbufs in this range; no need to search. */
                mutex_exit(&dn->dn_dbufs_mtx);
                return;
        } else if (dmu_objset_is_receiving(dn->dn_objset)) {
        if (dmu_objset_is_receiving(dn->dn_objset)) {
                /*
                 * If we are receiving, we expect there to be no dbufs in
                 * the range to be freed, because receive modifies each
                 * block at most once, and in offset order.  If this is
                 * not the case, it can lead to performance problems,
                 * so note that we unexpectedly took the slow path.
                 * When processing a free record from a zfs receive,
                 * there should have been no previous modifications to the
                 * data in this range.  Therefore there should be no dbufs
                 * in the range.  Searching dn_dbufs for these non-existent
                 * dbufs can be very expensive, so simply ignore this.
                 */
                atomic_inc_64(&zfs_free_range_recv_miss);
                VERIFY3P(dbuf_find(dn, 0, start), ==, NULL);
                VERIFY3P(dbuf_find(dn, 0, end), ==, NULL);
                return;
        }

        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);
```

```
848                 if (db->db_level == 1 &&
849                     db->db_blkid >= first_l1 && db->db_blkid <= last_l1) {
850                         mutex_enter(&db->db_mtx);
851                         if (db->db_last_dirty &&
852                             db->db_last_dirty->dr_txg < txg) {
853                                 dbuf_add_ref(db, FTAG);
854                                 mutex_exit(&db->db_mtx);
855                                 dbuf_will_dirty(db, tx);
856                                 dbuf_rele(db, FTAG);
857                         } else {
858                                 mutex_exit(&db->db_mtx);
859                         }
860                 }

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

868                 /* found a level 0 buffer in the range */
869                 mutex_enter(&db->db_mtx);
870                 if (dbuf_undirty(db, tx)) {
871                         /* mutex has been dropped and dbuf destroyed */
872                         continue;
873                 }

875                 if (db->db_state == DB_UNCACHED ||
876                     db->db_state == DB_NOFILL ||
877                     db->db_state == DB_EVICTING) {
878                         ASSERT(db->db.db_data == NULL);
879                         mutex_exit(&db->db_mtx);
880                         continue;
881                 }
882                 if (db->db_state == DB_READ || db->db_state == DB_FILL) {
883                         /* will be handled in dbuf_read_done or dbuf_rele */
884                         db->db_freed_in_flight = TRUE;
885                         mutex_exit(&db->db_mtx);
886                         continue;
887                 }
888                 if (refcount_count(&db->db_holds) == 0) {
889                         ASSERT(db->db_buf);
890                         dbuf_clear(db);
891                         continue;
892                 }
893                 /* The dbuf is referenced */

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

898                         if (dr->dr_txg == txg) {
899                                 /*
900                                  * This buffer is "in-use", re-adjust the file
901                                  * size to reflect that this buffer may
902                                  * contain new data when we sync.
903                                  */
904                                 if (db->db_blkid != DMU_SPILL_BLKID &&
905                                     db->db_blkid > dn->dn_maxblkid)
906                                         dn->dn_maxblkid = db->db_blkid;
907                                 dbuf_unoverride(dr);
908                         } else {
909                                 /*
910                                  * This dbuf is not dirty in the open context.
911                                  * Either uncache it (if its not referenced in
912                                  * the open context) or reset its contents to
```



```
913                                  * empty.
914                                  */
915                                 dbuf_fix_old_data(db, txg);
916                         }
917                 }
918                 /* clear the contents if its cached */
919                 if (db->db_state == DB_CACHED) {
920                         ASSERT(db->db.db_data != NULL);
921                         arc_release(db->db_buf, db);
922                         bzero(db->db.db_data, db->db.db_size);
923                         arc_buf_freeze(db->db_buf);
924                 }

926                 mutex_exit(&db->db_mtx);
927         }
928         mutex_exit(&dn->dn_dbufs_mtx);
929 }
```

_____***unchanged_portion_omitted_***

```
1666 static dmu_buf_impl_t *
1667 dbuf_create(dnode_t *dn, uint8_t level, uint64_t blkid,
1668     dmu_buf_impl_t *parent, blkptr_t *blkptr)
1669 {
1670         objset_t *os = dn->dn_objset;
1671         dmu_buf_impl_t *db, *odb;

1673         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1674         ASSERT(dn->dn_type != DMU_OT_NONE);

1676         db = kmem_cache_alloc(dbuf_cache, KM_SLEEP);

1678         db->db_objset = os;
1679         db->db.db_object = dn->dn_object;
1680         db->db_level = level;
1681         db->db_blkid = blkid;
1682         db->db_last_dirty = NULL;
1683         db->db_dirtycnt = 0;
1684         db->db_dnode_handle = dn->dn_handle;
1685         db->db_parent = parent;
1686         db->db_blkptr = blkptr;

1688         db->db_user_ptr = NULL;
1689         db->db_user_data_ptr_ptr = NULL;
1690         db->db_evict_func = NULL;
1691         db->db_immediate_evict = 0;
1692         db->db_freed_in_flight = 0;

1694         if (blkid == DMU_BONUS_BLKID) {
1695                 ASSERT3P(parent, ==, dn->dn_dbuf);
1696                 db->db.db_size = DN_MAX_BONUSLEN -
1697                     (dn->dn_nblkptr-1) * sizeof (blkptr_t);
1698                 ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
1699                 db->db.db_offset = DMU_BONUS_BLKID;
1700                 db->db_state = DB_UNCACHED;
1701                 /* the bonus dbuf is not placed in the hash table */
1702                 arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1703                 return (db);
1704         } else if (blkid == DMU_SPILL_BLKID) {
1705                 db->db.db_size = (blkptr != NULL) ?
1706                     BP_GET_LSIZE(blkptr) : SPA_MINBLOCKSIZE;
1707                 db->db.db_offset = 0;
1708         } else {
1709                 int blocksize =
1710                     db->db_level ? 1<<dn->dn_indblkshift :  dn->dn_datablksz;
1711                 db->db.db_size = blocksize;
1712                 db->db.db_offset = db->db_blkid * blocksize;
```

```
1713         }

1715         /*
1716          * Hold the dn_dbufs_mtx while we get the new dbuf
1717          * in the hash table *and* added to the dbufs list.
1718          * This prevents a possible deadlock with someone
1719          * trying to look up this dbuf before its added to the
1720          * dn_dbufs list.
1721          */
1722         mutex_enter(&dn->dn_dbufs_mtx);
1723         db->db_state = DB_EVICTING;
1724         if ((odb = dbuf_hash_insert(db)) != NULL) {
1725                 /* someone else inserted it first */
1726                 kmem_cache_free(dbuf_cache, db);
1727                 mutex_exit(&dn->dn_dbufs_mtx);
1728                 return (odb);
1729         }
1730         list_insert_head(&dn->dn_dbufs, db);
1731         if (db->db_level == 0 && db->db_blkid >=
1732             dn->dn_unlisted_l0_blkid)
1733                 dn->dn_unlisted_l0_blkid = db->db_blkid + 1;
1734         db->db_state = DB_UNCACHED;
1735         mutex_exit(&dn->dn_dbufs_mtx);
1736         arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);

1738         if (parent && parent != dn->dn_dbuf)
1739                 dbuf_add_ref(parent, db);

1741         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1742             refcount_count(&dn->dn_holds) > 0);
1743         (void) refcount_add(&dn->dn_holds, db);
1744         (void) atomic_inc_32_nv(&dn->dn_dbufs_count);

1746         dprintf_dbuf(db, "db=%p\n", db);

1748         return (db);
1749 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   44317 Thu Aug 15 17:43:51 2013
new/usr/src/uts/common/fs/zfs/dmu.c
4047 panic from dbuf_free_range() from dmu_free_object() while doing zfs receive
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 567 /*
 568  * Get the next "chunk" of file data to free.  We traverse the file from
 569  * the end so that the file gets shorter over time (if we crashes in the
 570  * middle, this will leave us in a better state).  We find allocated file
 571  * data by simply searching the allocated level 1 indirects.
 572  *
 573  * On input, *start should be the first offset that does not need to be
 574  * freed (e.g. "offset + length").  On return, *start will be the first
 575  * offset that should be freed.
 576  */
 577 static int
 578 get_next_chunk(dnode_t *dn, uint64_t *start, uint64_t minimum)
 574 get_next_chunk(dnode_t *dn, uint64_t *start, uint64_t limit)
 579 {
 580         uint64_t maxblks = DMU_MAX_ACCESS >> (dn->dn_indblkshift + 1);
 581         /* bytes of data covered by a level-1 indirect block */
 576         uint64_t len = *start - limit;
 577         uint64_t blkcnt = 0;
 578         uint64_t maxblks = DMU_MAX_ACCESS / (1ULL << (dn->dn_indblkshift + 1));
 582         uint64_t iblkrange =
 583             dn->dn_datablksz * EPB(dn->dn_indblkshift, SPA_BLKPTRSHIFT);

 585         ASSERT3U(minimum, <=, *start);
 582         ASSERT(limit <= *start);

 587         if (*start - minimum <= iblkrange * maxblks) {
 588                 *start = minimum;
 584         if (len <= iblkrange * maxblks) {
 585                 *start = limit;
 589                 return (0);
 590         }
 591         ASSERT(ISP2(iblkrange));

 593         for (uint64_t blks = 0; *start > minimum && blks < maxblks; blks++) {
 590         while (*start > limit && blkcnt < maxblks) {
 594                 int err;

 596                 /*
 597                  * dnode_next_offset(BACKWARDS) will find an allocated L1
 598                  * indirect block at or before the input offset.  We must
 599                  * decrement *start so that it is at the end of the region
 600                  * to search.
 601                  */
 602                 (*start)--;
 593                 /* find next allocated L1 indirect */
 603                 err = dnode_next_offset(dn,
 604                     DNODE_FIND_BACKWARDS, start, 2, 1, 0);

 606                 /* if there are no indirect blocks before start, we are done */
 597                 /* if there are no more, then we are done */
 607                 if (err == ESRCH) {
 608                         *start = minimum;
 609                         break;
 610                 } else if (err != 0) {
 599                         *start = limit;
 600                         return (0);
 601                 } else if (err) {
```



```
 611                         return (err);
 612                 }
 604                 blkcnt += 1;

 614                 /* set start to the beginning of this L1 indirect */
 606                 /* reset offset to end of "next" block back */
 615                 *start = P2ALIGN(*start, iblkrange);
 608                 if (*start <= limit)
 609                         *start = limit;
 610                 else
 611                         *start -= 1;
 616         }
 617         if (*start < minimum)
 618                 *start = minimum;
 619         return (0);
 620 }

 622 static int
 623 dmu_free_long_range_impl(objset_t *os, dnode_t *dn, uint64_t offset,
 624     uint64_t length)
 618     uint64_t length, boolean_t free_dnode)
 625 {
 626         uint64_t object_size = (dn->dn_maxblkid + 1) * dn->dn_datablksz;
 627         int err;
 620         dmu_tx_t *tx;
 621         uint64_t object_size, start, end, len;
 622         boolean_t trunc = (length == DMU_OBJECT_END);
 623         int align, err;

 629         if (offset >= object_size)
 625         align = 1 << dn->dn_datablkshift;
 626         ASSERT(align > 0);
 627         object_size = align == 1 ? dn->dn_datablksz :
 628             (dn->dn_maxblkid + 1) << dn->dn_datablkshift;

 630         end = offset + length;
 631         if (trunc || end > object_size)
 632                 end = object_size;
 633         if (end <= offset)
 630                 return (0);
 635         length = end - offset;

 632         if (length == DMU_OBJECT_END || offset + length > object_size)
 633                 length = object_size - offset;

 635         while (length != 0) {
 636                 uint64_t chunk_end, chunk_begin;

 638                 chunk_end = chunk_begin = offset + length;

 640                 /* move chunk_begin backwards to the beginning of this chunk */
 641                 err = get_next_chunk(dn, &chunk_begin, offset);
 637         while (length) {
 638                 start = end;
 639                 /* assert(offset <= start) */
 640                 err = get_next_chunk(dn, &start, offset);
 642                 if (err)
 643                         return (err);
 644                 ASSERT3U(chunk_begin, >=, offset);
 645                 ASSERT3U(chunk_begin, <=, chunk_end);
 643                 len = trunc ? DMU_OBJECT_END : end - start;

 647                 dmu_tx_t *tx = dmu_tx_create(os);
 648                 dmu_tx_hold_free(tx, dn->dn_object,
 649                     chunk_begin, chunk_end - chunk_begin);
 645                 tx = dmu_tx_create(os);
```

```
646                 dmu_tx_hold_free(tx, dn->dn_object, start, len);
650                 err = dmu_tx_assign(tx, TXG_WAIT);
651                 if (err) {
652                         dmu_tx_abort(tx);
653                         return (err);
654                 }
655                 dnode_free_range(dn, chunk_begin, chunk_end - chunk_begin, tx);
656                 dmu_tx_commit(tx);

658                 length -= chunk_end - chunk_begin;
653                 dnode_free_range(dn, start, trunc ? -1 : len, tx);

655                 if (start == 0 && free_dnode) {
656                         ASSERT(trunc);
657                         dnode_free(dn, tx);
659         }

660                 length -= end - start;

662                 dmu_tx_commit(tx);
663                 end = start;
664         }
660         return (0);
661 }

663 int
664 dmu_free_long_range(objset_t *os, uint64_t object,
665     uint64_t offset, uint64_t length)
666 {
667         dnode_t *dn;
668         int err;

670         err = dnode_hold(os, object, FTAG, &dn);
671         if (err != 0)
672                 return (err);
673         err = dmu_free_long_range_impl(os, dn, offset, length);
678         err = dmu_free_long_range_impl(os, dn, offset, length, FALSE);
674         dnode_rele(dn, FTAG);
675         return (err);
676 }

678 int
679 dmu_free_long_object(objset_t *os, uint64_t object)
684 dmu_free_object(objset_t *os, uint64_t object)
680 {
686         dnode_t *dn;
681         dmu_tx_t *tx;
682         int err;

684         err = dmu_free_long_range(os, object, 0, DMU_OBJECT_END);
690         err = dnode_hold_impl(os, object, DNODE_MUST_BE_ALLOCATED,
691             FTAG, &dn);
685         if (err != 0)
686                 return (err);

694         if (dn->dn_nlevels == 1) {
688         tx = dmu_tx_create(os);
689         dmu_tx_hold_bonus(tx, object);
690         dmu_tx_hold_free(tx, object, 0, DMU_OBJECT_END);
697                 dmu_tx_hold_free(tx, dn->dn_object, 0, DMU_OBJECT_END);
691         err = dmu_tx_assign(tx, TXG_WAIT);
692         if (err == 0) {
693                 err = dmu_object_free(os, object, tx);
700                         dnode_free_range(dn, 0, DMU_OBJECT_END, tx);
701                         dnode_free(dn, tx);
694                 dmu_tx_commit(tx);
```



```
695         } else {
696                 dmu_tx_abort(tx);
697         }

706         } else {
707                 err = dmu_free_long_range_impl(os, dn, 0, DMU_OBJECT_END, TRUE);
708         }
709         dnode_rele(dn, FTAG);
699         return (err);
700 }
```

_____***unchanged_portion_omitted_***

****
49105 Thu Aug 15 17:43:54 2013
new/usr/src/uts/common/fs/zfs/dmu_send.c
4047 panic from dbuf_free_range() from dmu_free_object() while doing zfs receive
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
****
_____***unchanged_portion_omitted_***

```
1217 /* ARGSUSED */
1218 static int
1219 restore_freeobjects(struct restorearg *ra, objset_t *os,
1220     struct drr_freeobjects *drrfo)
1221 {
1222         uint64_t obj;

1224         if (drrfo->drr_firstobj + drrfo->drr_numobjs < drrfo->drr_firstobj)
1225                 return (SET_ERROR(EINVAL));

1227         for (obj = drrfo->drr_firstobj;
1228             obj < drrfo->drr_firstobj + drrfo->drr_numobjs;
1229             (void) dmu_object_next(os, &obj, FALSE, 0)) {
1230                 int err;

1232                 if (dmu_object_info(os, obj, NULL) != 0)
1233                         continue;

1235                 err = dmu_free_long_object(os, obj);
1235                 err = dmu_free_object(os, obj);
1236                 if (err != 0)
1237                         return (err);
1238         }
1239         return (0);
1240 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   36042 Thu Aug 15 17:43:56 2013
new/usr/src/uts/common/fs/zfs/dmu_tx.c
4047 panic from dbuf_free_range() from dmu_free_object() while doing zfs receive
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 585 void
 586 dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off, uint64_t len)
 587 {
 588         dmu_tx_hold_t *txh;
 589         dnode_t *dn;
 590         int err;
 591         zio_t *zio;

 593         ASSERT(tx->tx_txg == 0);

 595         txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
 596             object, THT_FREE, off, len);
 597         if (txh == NULL)
 598                 return;
 599         dn = txh->txh_dnode;

 601         if (off >= (dn->dn_maxblkid+1) * dn->dn_datablksz)
 602                 return;
 603         if (len == DMU_OBJECT_END)
 604                 len = (dn->dn_maxblkid+1) * dn->dn_datablksz - off;

 606         dmu_tx_count_dnode(txh);

 608         /*
 609          * For i/o error checking, we read the first and last level-0
 610          * blocks if they are not aligned, and all the level-1 blocks.
 611          *
 612          * Note:  dbuf_free_range() assumes that we have not instantiated
 613          * any level-0 dbufs that will be completely freed.  Therefore we must
 614          * exercise care to not read or count the first and last blocks
 615          * if they are blocksize-aligned.
 616          */
 617         if (dn->dn_datablkshift == 0) {
 618                 if (off != 0 || len < dn->dn_datablksz)
 619                         dmu_tx_count_write(txh, off, len);
 620         } else {
 621                 /* first block will be modified if it is not aligned */
 622                 if (!IS_P2ALIGNED(off, 1 << dn->dn_datablkshift))
 623                         dmu_tx_count_write(txh, off, 1);
 624                 /* last block will be modified if it is not aligned */
 625                 if (!IS_P2ALIGNED(off + len, 1 << dn->dn_datablkshift))
 626                         dmu_tx_count_write(txh, off+len, 1);
 627         }

 629         /*
 630          * Check level-1 blocks.
 631          */
 632         if (dn->dn_nlevels > 1) {
 633                 int shift = dn->dn_datablkshift + dn->dn_indblkshift -
 634                     SPA_BLKPTRSHIFT;
 635                 uint64_t start = off >> shift;
 636                 uint64_t end = (off + len) >> shift;

 638                 ASSERT(dn->dn_datablkshift != 0);
 639                 ASSERT(dn->dn_indblkshift != 0);

 641                 zio = zio_root(tx->tx_pool->dp_spa,
```



```
 642                     NULL, NULL, ZIO_FLAG_CANFAIL);
 643                 for (uint64_t i = start; i <= end; i++) {
 644                         uint64_t ibyte = i << shift;
 645                         err = dnode_next_offset(dn, 0, &ibyte, 2, 1, 0);
 646                         i = ibyte >> shift;
 647                         if (err == ESRCH)
 648                                 break;
 649                         if (err) {
 650                                 tx->tx_err = err;
 651                                 return;
 652                         }

 654                         err = dmu_tx_check_ioerr(zio, dn, 1, i);
 655                         if (err) {
 656                                 tx->tx_err = err;
 657                                 return;
 658                         }
 659                 }
 660                 err = zio_wait(zio);
 661                 if (err) {
 662                         tx->tx_err = err;
 663                         return;
 664                 }
 665         }

 667         dmu_tx_count_free(txh, off, len);
 668 }
______unchanged_portion_omitted_
```

```
**********************************************************
   56684 Thu Aug 15 17:43:59 2013
new/usr/src/uts/common/fs/zfs/dnode.c
4047 panic from dbuf_free_range() from dmu_free_object() while doing zfs receive
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright (c) 2013 by Delphix. All rights reserved.
  24  */

  26 #include <sys/zfs_context.h>
  27 #include <sys/dbuf.h>
  28 #include <sys/dnode.h>
  29 #include <sys/dmu.h>
  30 #include <sys/dmu_impl.h>
  31 #include <sys/dmu_tx.h>
  32 #include <sys/dmu_objset.h>
  33 #include <sys/dsl_dir.h>
  34 #include <sys/dsl_dataset.h>
  35 #include <sys/spa.h>
  36 #include <sys/zio.h>
  37 #include <sys/dmu_zfetch.h>

  39 static int free_range_compar(const void *node1, const void *node2);

  41 static kmem_cache_t *dnode_cache;
  42 /*
  43  * Define DNODE_STATS to turn on statistic gathering. By default, it is only
  44  * turned on when DEBUG is also defined.
  45  */
  46 #ifdef  DEBUG
  47 #define DNODE_STATS
  48 #endif  /* DEBUG */

  50 #ifdef  DNODE_STATS
  51 #define DNODE_STAT_ADD(stat)                    ((stat)++)
  52 #else
  53 #define DNODE_STAT_ADD(stat)                    /* nothing */
  54 #endif  /* DNODE_STATS */

  56 static dnode_phys_t dnode_phys_zero;

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



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

  63 /* ARGSUSED */
  64 static int
  65 dnode_cons(void *arg, void *unused, int kmflag)
  66 {
  67         dnode_t *dn = arg;
  68         int i;

  70         rw_init(&dn->dn_struct_rwlock, NULL, RW_DEFAULT, NULL);
  71         mutex_init(&dn->dn_mtx, NULL, MUTEX_DEFAULT, NULL);
  72         mutex_init(&dn->dn_dbufs_mtx, NULL, MUTEX_DEFAULT, NULL);
  73         cv_init(&dn->dn_notxholds, NULL, CV_DEFAULT, NULL);

  75         /*
  76          * Every dbuf has a reference, and dropping a tracked reference is
  77          * O(number of references), so don't track dn_holds.
  78          */
  79         refcount_create_untracked(&dn->dn_holds);
  80         refcount_create(&dn->dn_tx_holds);
  81         list_link_init(&dn->dn_link);

  83         bzero(&dn->dn_next_nblkptr[0], sizeof (dn->dn_next_nblkptr));
  84         bzero(&dn->dn_next_nlevels[0], sizeof (dn->dn_next_nlevels));
  85         bzero(&dn->dn_next_indblkshift[0], sizeof (dn->dn_next_indblkshift));
  86         bzero(&dn->dn_next_bonustype[0], sizeof (dn->dn_next_bonustype));
  87         bzero(&dn->dn_rm_spillblk[0], sizeof (dn->dn_rm_spillblk));
  88         bzero(&dn->dn_next_bonuslen[0], sizeof (dn->dn_next_bonuslen));
  89         bzero(&dn->dn_next_blksz[0], sizeof (dn->dn_next_blksz));

  91         for (i = 0; i < TXG_SIZE; i++) {
  92                 list_link_init(&dn->dn_dirty_link[i]);
  93                 avl_create(&dn->dn_ranges[i], free_range_compar,
  94                     sizeof (free_range_t),
  95                     offsetof(struct free_range, fr_node));
  96                 list_create(&dn->dn_dirty_records[i],
  97                     sizeof (dbuf_dirty_record_t),
  98                     offsetof(dbuf_dirty_record_t, dr_dirty_node));
  99         }

 101         dn->dn_allocated_txg = 0;
 102         dn->dn_free_txg = 0;
 103         dn->dn_assigned_txg = 0;
 104         dn->dn_dirtyctx = 0;
 105         dn->dn_dirtyctx_firstset = NULL;
 106         dn->dn_bonus = NULL;
 107         dn->dn_have_spill = B_FALSE;
 108         dn->dn_zio = NULL;
 109         dn->dn_oldused = 0;
 110         dn->dn_oldflags = 0;
 111         dn->dn_olduid = 0;
 112         dn->dn_oldgid = 0;
 113         dn->dn_newuid = 0;
 114         dn->dn_newgid = 0;
 115         dn->dn_id_flags = 0;

 117         dn->dn_dbufs_count = 0;
 118         dn->dn_unlisted_l0_blkid = 0;
 119         list_create(&dn->dn_dbufs, sizeof (dmu_buf_impl_t),
 120             offsetof(dmu_buf_impl_t, db_link));

 122         dn->dn_moved = 0;
 123         return (0);
 124 }
```

```
126 /* ARGSUSED */
127 static void
128 dnode_dest(void *arg, void *unused)
129 {
130         int i;
131         dnode_t *dn = arg;

133         rw_destroy(&dn->dn_struct_rwlock);
134         mutex_destroy(&dn->dn_mtx);
135         mutex_destroy(&dn->dn_dbufs_mtx);
136         cv_destroy(&dn->dn_notxholds);
137         refcount_destroy(&dn->dn_holds);
138         refcount_destroy(&dn->dn_tx_holds);
139         ASSERT(!list_link_active(&dn->dn_link));

141         for (i = 0; i < TXG_SIZE; i++) {
142                 ASSERT(!list_link_active(&dn->dn_dirty_link[i]));
143                 avl_destroy(&dn->dn_ranges[i]);
144                 list_destroy(&dn->dn_dirty_records[i]);
145                 ASSERT0(dn->dn_next_nblkptr[i]);
146                 ASSERT0(dn->dn_next_nlevels[i]);
147                 ASSERT0(dn->dn_next_indblkshift[i]);
148                 ASSERT0(dn->dn_next_bonustype[i]);
149                 ASSERT0(dn->dn_rm_spillblk[i]);
150                 ASSERT0(dn->dn_next_bonuslen[i]);
151                 ASSERT0(dn->dn_next_blksz[i]);
152         }

154         ASSERT0(dn->dn_allocated_txg);
155         ASSERT0(dn->dn_free_txg);
156         ASSERT0(dn->dn_assigned_txg);
157         ASSERT0(dn->dn_dirtyctx);
158         ASSERT3P(dn->dn_dirtyctx_firstset, ==, NULL);
159         ASSERT3P(dn->dn_bonus, ==, NULL);
160         ASSERT(!dn->dn_have_spill);
161         ASSERT3P(dn->dn_zio, ==, NULL);
162         ASSERT0(dn->dn_oldused);
163         ASSERT0(dn->dn_oldflags);
164         ASSERT0(dn->dn_olduid);
165         ASSERT0(dn->dn_oldgid);
166         ASSERT0(dn->dn_newuid);
167         ASSERT0(dn->dn_newgid);
168         ASSERT0(dn->dn_id_flags);

170         ASSERT0(dn->dn_dbufs_count);
171         ASSERT0(dn->dn_unlisted_l0_blkid);
172         list_destroy(&dn->dn_dbufs);
173 }
```

*unchanged_portion_omitted_*

```
435 /*
436  * Caller must be holding the dnode handle, which is released upon return.
437  */
438 static void
439 dnode_destroy(dnode_t *dn)
440 {
441         objset_t *os = dn->dn_objset;

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

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

450         /* the dnode can no longer move, so we can release the handle */
```



```
451         zrl_remove(&dn->dn_handle->dnh_zrlock);

453         dn->dn_allocated_txg = 0;
454         dn->dn_free_txg = 0;
455         dn->dn_assigned_txg = 0;

457         dn->dn_dirtyctx = 0;
458         if (dn->dn_dirtyctx_firstset != NULL) {
459                 kmem_free(dn->dn_dirtyctx_firstset, 1);
460                 dn->dn_dirtyctx_firstset = NULL;
461         }
462         if (dn->dn_bonus != NULL) {
463                 mutex_enter(&dn->dn_bonus->db_mtx);
464                 dbuf_evict(dn->dn_bonus);
465                 dn->dn_bonus = NULL;
466         }
467         dn->dn_zio = NULL;

469         dn->dn_have_spill = B_FALSE;
470         dn->dn_oldused = 0;
471         dn->dn_oldflags = 0;
472         dn->dn_olduid = 0;
473         dn->dn_oldgid = 0;
474         dn->dn_newuid = 0;
475         dn->dn_newgid = 0;
476         dn->dn_id_flags = 0;
477         dn->dn_unlisted_l0_blkid = 0;

479         dmu_zfetch_rele(&dn->dn_zfetch);
480         kmem_cache_free(dnode_cache, dn);
481         arc_space_return(sizeof (dnode_t), ARC_SPACE_OTHER);
482 }
```

*unchanged_portion_omitted_*

```
649 #endif  /* DNODE_STATS */

651 static void
652 dnode_move_impl(dnode_t *odn, dnode_t *ndn)
653 {
654         int i;

656         ASSERT(!RW_LOCK_HELD(&odn->dn_struct_rwlock));
657         ASSERT(MUTEX_NOT_HELD(&odn->dn_mtx));
658         ASSERT(MUTEX_NOT_HELD(&odn->dn_dbufs_mtx));
659         ASSERT(!RW_LOCK_HELD(&odn->dn_zfetch.zf_rwlock));

661         /* Copy fields. */
662         ndn->dn_objset = odn->dn_objset;
663         ndn->dn_object = odn->dn_object;
664         ndn->dn_dbuf = odn->dn_dbuf;
665         ndn->dn_handle = odn->dn_handle;
666         ndn->dn_phys = odn->dn_phys;
667         ndn->dn_type = odn->dn_type;
668         ndn->dn_bonuslen = odn->dn_bonuslen;
669         ndn->dn_bonustype = odn->dn_bonustype;
670         ndn->dn_nblkptr = odn->dn_nblkptr;
671         ndn->dn_checksum = odn->dn_checksum;
672         ndn->dn_compress = odn->dn_compress;
673         ndn->dn_nlevels = odn->dn_nlevels;
674         ndn->dn_indblkshift = odn->dn_indblkshift;
675         ndn->dn_datablkshift = odn->dn_datablkshift;
676         ndn->dn_datablkszsec = odn->dn_datablkszsec;
677         ndn->dn_datablksz = odn->dn_datablksz;
678         ndn->dn_maxblkid = odn->dn_maxblkid;
679         bcopy(&odn->dn_next_nblkptr[0], &ndn->dn_next_nblkptr[0],
680             sizeof (odn->dn_next_nblkptr));
681         bcopy(&odn->dn_next_nlevels[0], &ndn->dn_next_nlevels[0],
```

```
682             sizeof (odn->dn_next_nlevels));
683         bcopy(&odn->dn_next_indblkshift[0], &ndn->dn_next_indblkshift[0],
684             sizeof (odn->dn_next_indblkshift));
685         bcopy(&odn->dn_next_bonustype[0], &ndn->dn_next_bonustype[0],
686             sizeof (odn->dn_next_bonustype));
687         bcopy(&odn->dn_rm_spillblk[0], &ndn->dn_rm_spillblk[0],
688             sizeof (odn->dn_rm_spillblk));
689         bcopy(&odn->dn_next_bonuslen[0], &ndn->dn_next_bonuslen[0],
690             sizeof (odn->dn_next_bonuslen));
691         bcopy(&odn->dn_next_blksz[0], &ndn->dn_next_blksz[0],
692             sizeof (odn->dn_next_blksz));
693         for (i = 0; i < TXG_SIZE; i++) {
694                 list_move_tail(&ndn->dn_dirty_records[i],
695                     &odn->dn_dirty_records[i]);
696         }
697         bcopy(&odn->dn_ranges[0], &ndn->dn_ranges[0], sizeof (odn->dn_ranges));
698         ndn->dn_allocated_txg = odn->dn_allocated_txg;
699         ndn->dn_free_txg = odn->dn_free_txg;
700         ndn->dn_assigned_txg = odn->dn_assigned_txg;
701         ndn->dn_dirtyctx = odn->dn_dirtyctx;
702         ndn->dn_dirtyctx_firstset = odn->dn_dirtyctx_firstset;
703         ASSERT(refcount_count(&odn->dn_tx_holds) == 0);
704         refcount_transfer(&ndn->dn_holds, &odn->dn_holds);
705         ASSERT(list_is_empty(&ndn->dn_dbufs));
706         list_move_tail(&ndn->dn_dbufs, &odn->dn_dbufs);
707         ndn->dn_dbufs_count = odn->dn_dbufs_count;
708         ndn->dn_unlisted_l0_blkid = odn->dn_unlisted_l0_blkid;
709         ndn->dn_bonus = odn->dn_bonus;
710         ndn->dn_have_spill = odn->dn_have_spill;
711         ndn->dn_zio = odn->dn_zio;
712         ndn->dn_oldused = odn->dn_oldused;
713         ndn->dn_oldflags = odn->dn_oldflags;
714         ndn->dn_olduid = odn->dn_olduid;
715         ndn->dn_oldgid = odn->dn_oldgid;
716         ndn->dn_newuid = odn->dn_newuid;
717         ndn->dn_newgid = odn->dn_newgid;
718         ndn->dn_id_flags = odn->dn_id_flags;
719         dmu_zfetch_init(&ndn->dn_zfetch, NULL);
720         list_move_tail(&ndn->dn_zfetch.zf_stream, &odn->dn_zfetch.zf_stream);
721         ndn->dn_zfetch.zf_dnode = odn->dn_zfetch.zf_dnode;
722         ndn->dn_zfetch.zf_stream_cnt = odn->dn_zfetch.zf_stream_cnt;
723         ndn->dn_zfetch.zf_alloc_fail = odn->dn_zfetch.zf_alloc_fail;

725         /*
726          * Update back pointers. Updating the handle fixes the back pointer of
727          * every descendant dbuf as well as the bonus dbuf.
728          */
729         ASSERT(ndn->dn_handle->dnh_dnode == odn);
730         ndn->dn_handle->dnh_dnode = ndn;
731         if (ndn->dn_zfetch.zf_dnode == odn) {
732                 ndn->dn_zfetch.zf_dnode = ndn;
733         }

735         /*
736          * Invalidate the original dnode by clearing all of its back pointers.
737          */
738         odn->dn_dbuf = NULL;
739         odn->dn_handle = NULL;
740         list_create(&odn->dn_dbufs, sizeof (dmu_buf_impl_t),
741             offsetof(dmu_buf_impl_t, db_link));
742         odn->dn_dbufs_count = 0;
743         odn->dn_unlisted_l0_blkid = 0;
744         odn->dn_bonus = NULL;
745         odn->dn_zfetch.zf_dnode = NULL;

747         /*
```



```
748          * Set the low bit of the objset pointer to ensure that dnode_move()
749          * recognizes the dnode as invalid in any subsequent callback.
750          */
751         POINTER_INVALIDATE(&odn->dn_objset);

753         /*
754          * Satisfy the destructor.
755          */
756         for (i = 0; i < TXG_SIZE; i++) {
757                 list_create(&odn->dn_dirty_records[i],
758                     sizeof (dbuf_dirty_record_t),
759                     offsetof(dbuf_dirty_record_t, dr_dirty_node));
760                 odn->dn_ranges[i].avl_root = NULL;
761                 odn->dn_ranges[i].avl_numnodes = 0;
762                 odn->dn_next_nlevels[i] = 0;
763                 odn->dn_next_indblkshift[i] = 0;
764                 odn->dn_next_bonustype[i] = 0;
765                 odn->dn_rm_spillblk[i] = 0;
766                 odn->dn_next_bonuslen[i] = 0;
767                 odn->dn_next_blksz[i] = 0;
768         }
769         odn->dn_allocated_txg = 0;
770         odn->dn_free_txg = 0;
771         odn->dn_assigned_txg = 0;
772         odn->dn_dirtyctx = 0;
773         odn->dn_dirtyctx_firstset = NULL;
774         odn->dn_have_spill = B_FALSE;
775         odn->dn_zio = NULL;
776         odn->dn_oldused = 0;
777         odn->dn_oldflags = 0;
778         odn->dn_olduid = 0;
779         odn->dn_oldgid = 0;
780         odn->dn_newuid = 0;
781         odn->dn_newgid = 0;
782         odn->dn_id_flags = 0;

784         /*
785          * Mark the dnode.
786          */
787         ndn->dn_moved = 1;
788         odn->dn_moved = (uint8_t)-1;
789 }
______unchanged_portion_omitted_

1517 void
1518 dnode_free_range(dnode_t *dn, uint64_t off, uint64_t len, dmu_tx_t *tx)
1519 {
1520         dmu_buf_impl_t *db;
1521         uint64_t blkoff, blkid, nblks;
1522         int blksz, blkshift, head, tail;
1523         int trunc = FALSE;
1524         int epbs;

1526         rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
1527         blksz = dn->dn_datablksz;
1528         blkshift = dn->dn_datablkshift;
1529         epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;

1531         if (len == DMU_OBJECT_END) {
1526         if (len == -1ULL) {
1532                 len = UINT64_MAX - off;
1533                 trunc = TRUE;
1534         }

1536         /*
1537          * First, block align the region to free:
```

```
1538          */
1539         if (ISP2(blksz)) {
1540                 head = P2NPHASE(off, blksz);
1541                 blkoff = P2PHASE(off, blksz);
1542                 if ((off >> blkshift) > dn->dn_maxblkid)
1543                         goto out;
1544         } else {
1545                 ASSERT(dn->dn_maxblkid == 0);
1546                 if (off == 0 && len >= blksz) {
1547                         /* Freeing the whole block; fast-track this request */
1548                         blkid = 0;
1549                         nblks = 1;
1550                         goto done;
1551                 } else if (off >= blksz) {
1552                         /* Freeing past end-of-data */
1553                         goto out;
1554                 } else {
1555                         /* Freeing part of the block. */
1556                         head = blksz - off;
1557                         ASSERT3U(head, >, 0);
1558                 }
1559                 blkoff = off;
1560         }
1561         /* zero out any partial block data at the start of the range */
1562         if (head) {
1563                 ASSERT3U(blkoff + head, ==, blksz);
1564                 if (len < head)
1565                         head = len;
1566                 if (dbuf_hold_impl(dn, 0, dbuf_whichblock(dn, off), TRUE,
1567                     FTAG, &db) == 0) {
1568                         caddr_t data;

1570                         /* don't dirty if it isn't on disk and isn't dirty */
1571                         if (db->db_last_dirty ||
1572                             (db->db_blkptr && !BP_IS_HOLE(db->db_blkptr))) {
1573                                 rw_exit(&dn->dn_struct_rwlock);
1574                                 dbuf_will_dirty(db, tx);
1575                                 rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
1576                                 data = db->db.db_data;
1577                                 bzero(data + blkoff, head);
1578                         }
1579                         dbuf_rele(db, FTAG);
1580                 }
1581                 off += head;
1582                 len -= head;
1583         }

1585         /* If the range was less than one block, we're done */
1586         if (len == 0)
1587                 goto out;

1589         /* If the remaining range is past end of file, we're done */
1590         if ((off >> blkshift) > dn->dn_maxblkid)
1591                 goto out;

1593         ASSERT(ISP2(blksz));
1594         if (trunc)
1595                 tail = 0;
1596         else
1597                 tail = P2PHASE(len, blksz);

1599         ASSERT0(P2PHASE(off, blksz));
1600         /* zero out any partial block data at the end of the range */
1601         if (tail) {
1602                 if (len < tail)
1603                         tail = len;
```



```
1604                 if (dbuf_hold_impl(dn, 0, dbuf_whichblock(dn, off+len),
1605                     TRUE, FTAG, &db) == 0) {
1606                         /* don't dirty if not on disk and not dirty */
1607                         if (db->db_last_dirty ||
1608                             (db->db_blkptr && !BP_IS_HOLE(db->db_blkptr))) {
1609                                 rw_exit(&dn->dn_struct_rwlock);
1610                                 dbuf_will_dirty(db, tx);
1611                                 rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
1612                                 bzero(db->db.db_data, tail);
1613                         }
1614                         dbuf_rele(db, FTAG);
1615                 }
1616                 len -= tail;
1617         }

1619         /* If the range did not include a full block, we are done */
1620         if (len == 0)
1621                 goto out;

1623         ASSERT(IS_P2ALIGNED(off, blksz));
1624         ASSERT(trunc || IS_P2ALIGNED(len, blksz));
1625         blkid = off >> blkshift;
1626         nblks = len >> blkshift;
1627         if (trunc)
1628                 nblks += 1;

1630         /*
1631          * Read in and mark all the level-1 indirects dirty,
1632          * so that they will stay in memory until syncing phase.
1633          * Always dirty the first and last indirect to make sure
1634          * we dirty all the partial indirects.
1635          */
1636         if (dn->dn_nlevels > 1) {
1637                 uint64_t i, first, last;
1638                 int shift = epbs + dn->dn_datablkshift;

1640                 first = blkid >> epbs;
1641                 if (db = dbuf_hold_level(dn, 1, first, FTAG)) {
1642                         dbuf_will_dirty(db, tx);
1643                         dbuf_rele(db, FTAG);
1644                 }
1645                 if (trunc)
1646                         last = dn->dn_maxblkid >> epbs;
1647                 else
1648                         last = (blkid + nblks - 1) >> epbs;
1649                 if (last > first && (db = dbuf_hold_level(dn, 1, last, FTAG))) {
1650                         dbuf_will_dirty(db, tx);
1651                         dbuf_rele(db, FTAG);
1652                 }
1653                 for (i = first + 1; i < last; i++) {
1654                         uint64_t ibyte = i << shift;
1655                         int err;

1657                         err = dnode_next_offset(dn,
1658                             DNODE_FIND_HAVELOCK, &ibyte, 1, 1, 0);
1659                         i = ibyte >> shift;
1660                         if (err == ESRCH || i >= last)
1661                                 break;
1662                         ASSERT(err == 0);
1663                         db = dbuf_hold_level(dn, 1, i, FTAG);
1664                         if (db) {
1665                                 dbuf_will_dirty(db, tx);
1666                                 dbuf_rele(db, FTAG);
1667                         }
1668                 }
1669         }
```

```
1670 done:
1671         /*
1672          * Add this range to the dnode range list.
1673          * We will finish up this free operation in the syncing phase.
1674          */
1675         mutex_enter(&dn->dn_mtx);
1676         dnode_clear_range(dn, blkid, nblks, tx);
1677         {
1678                 free_range_t *rp, *found;
1679                 avl_index_t where;
1680                 avl_tree_t *tree = &dn->dn_ranges[tx->tx_txg&TXG_MASK];

1682                 /* Add new range to dn_ranges */
1683                 rp = kmem_alloc(sizeof (free_range_t), KM_SLEEP);
1684                 rp->fr_blkid = blkid;
1685                 rp->fr_nblks = nblks;
1686                 found = avl_find(tree, rp, &where);
1687                 ASSERT(found == NULL);
1688                 avl_insert(tree, rp, where);
1689                 dprintf_dnode(dn, "blkid=%llu nblks=%llu txg=%llu\n",
1690                     blkid, nblks, tx->tx_txg);
1691         }
1692         mutex_exit(&dn->dn_mtx);

1694         dbuf_free_range(dn, blkid, blkid + nblks - 1, tx);
1695         dnode_setdirty(dn, tx);
1696 out:
1697         if (trunc && dn->dn_maxblkid >= (off >> blkshift))
1698                 dn->dn_maxblkid = (off >> blkshift ? (off >> blkshift) - 1 : 0);

1700         rw_exit(&dn->dn_struct_rwlock);
1701 }
```

_____***unchanged_portion_omitted_***

****
25699 Thu Aug 15 17:44:02 2013
new/usr/src/uts/common/fs/zfs/dsl_destroy.c
4047 panic from dbuf_free_range() from dmu_free_object() while doing zfs receive
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
****
_____***unchanged_portion_omitted_***

```
 858 int
 859 dsl_destroy_head(const char *name)
 860 {
 861         dsl_destroy_head_arg_t ddha;
 862         int error;
 863         spa_t *spa;
 864         boolean_t isenabled;

 866 #ifdef _KERNEL
 867         zfs_destroy_unmount_origin(name);
 868 #endif

 870         error = spa_open(name, &spa, FTAG);
 871         if (error != 0)
 872                 return (error);
 873         isenabled = spa_feature_is_enabled(spa,
 874             &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY]);
 875         spa_close(spa, FTAG);

 877         ddha.ddha_name = name;

 879         if (!isenabled) {
 880                 objset_t *os;

 882                 error = dsl_sync_task(name, dsl_destroy_head_check,
 883                     dsl_destroy_head_begin_sync, &ddha, 0);
 884                 if (error != 0)
 885                         return (error);

 887                 /*
 888                  * Head deletion is processed in one txg on old pools;
 889                  * remove the objects from open context so that the txg sync
 890                  * is not too long.
 891                  */
 892                 error = dmu_objset_own(name, DMU_OST_ANY, B_FALSE, FTAG, &os);
 893                 if (error == 0) {
 894                         uint64_t prev_snap_txg =
 895                             dmu_objset_ds(os)->ds_phys->ds_prev_snap_txg;
 896                         for (uint64_t obj = 0; error == 0;
 897                             error = dmu_object_next(os, &obj, FALSE,
 898                             prev_snap_txg))
 899                                 (void) dmu_free_long_object(os, obj);
 899                                 (void) dmu_free_object(os, obj);
 900                         /* sync out all frees */
 901                         txg_wait_synced(dmu_objset_pool(os), 0);
 902                         dmu_objset_disown(os, FTAG);
 903                 }
 904         }

 906         return (dsl_sync_task(name, dsl_destroy_head_check,
 907             dsl_destroy_head_sync, &ddha, 0));
 908 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   28897 Thu Aug 15 17:44:05 2013
new/usr/src/uts/common/fs/zfs/sys/dmu.h
4047 panic from dbuf_free_range() from dmu_free_object() while doing zfs receive
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  24  * Copyright (c) 2013 by Delphix. All rights reserved.
  24  * Copyright (c) 2012 by Delphix. All rights reserved.
  25  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
  26  * Copyright (c) 2012, Joyent, Inc. All rights reserved.
  27  */

  29 /* Portions Copyright 2010 Robert Milkowski */

  31 #ifndef _SYS_DMU_H
  32 #define _SYS_DMU_H

  34 /*
  35  * This file describes the interface that the DMU provides for its
  36  * consumers.
  37  *
  38  * The DMU also interacts with the SPA.  That interface is described in
  39  * dmu_spa.h.
  40  */

  42 #include <sys/inttypes.h>
  43 #include <sys/types.h>
  44 #include <sys/param.h>
  45 #include <sys/cred.h>
  46 #include <sys/time.h>
  47 #include <sys/fs/zfs.h>

  49 #ifdef  __cplusplus
  50 extern "C" {
  51 #endif

  53 struct uio;
  54 struct xuio;
  55 struct page;
  56 struct vnode;
  57 struct spa;
  58 struct zilog;
```



```
  59 struct zio;
  60 struct blkptr;
  61 struct zap_cursor;
  62 struct dsl_dataset;
  63 struct dsl_pool;
  64 struct dnode;
  65 struct drr_begin;
  66 struct drr_end;
  67 struct zbookmark;
  68 struct spa;
  69 struct nvlist;
  70 struct arc_buf;
  71 struct zio_prop;
  72 struct sa_handle;

  74 typedef struct objset objset_t;
  75 typedef struct dmu_tx dmu_tx_t;
  76 typedef struct dsl_dir dsl_dir_t;

  78 typedef enum dmu_object_byteswap {
  79         DMU_BSWAP_UINT8,
  80         DMU_BSWAP_UINT16,
  81         DMU_BSWAP_UINT32,
  82         DMU_BSWAP_UINT64,
  83         DMU_BSWAP_ZAP,
  84         DMU_BSWAP_DNODE,
  85         DMU_BSWAP_OBJSET,
  86         DMU_BSWAP_ZNODE,
  87         DMU_BSWAP_OLDACL,
  88         DMU_BSWAP_ACL,
  89         /*
  90          * Allocating a new byteswap type number makes the on-disk format
  91          * incompatible with any other format that uses the same number.
  92          *
  93          * Data can usually be structured to work with one of the
  94          * DMU_BSWAP_UINT* or DMU_BSWAP_ZAP types.
  95          */
  96         DMU_BSWAP_NUMFUNCS
  97 } dmu_object_byteswap_t;
_____unchanged_portion_omitted_

 284 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  * Returns ENOENT, EIO, or 0.
414  */
415 int dmu_bonus_hold(objset_t *os, uint64_t object, void *tag, dmu_buf_t **);
416 int dmu_bonus_max(void);
417 int dmu_set_bonus(dmu_buf_t *, int, dmu_tx_t *);
418 int dmu_set_bonustype(dmu_buf_t *, dmu_object_type_t, dmu_tx_t *);
419 dmu_object_type_t dmu_get_bonustype(dmu_buf_t *);
420 int dmu_rm_spill(objset_t *, uint64_t, dmu_tx_t *);

422 /*
423  * Special spill buffer support used by "SA" framework
424  */

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

431 /*
432  * Obtain the DMU buffer from the specified object which contains the
433  * specified offset.  dmu_buf_hold() puts a "hold" on the buffer, so
434  * that it will remain in memory.  You must release the hold with
435  * dmu_buf_rele().  You musn't access the dmu_buf_t after releasing your
436  * hold.  You must have a hold on any dmu_buf_t* you pass to the DMU.
437  *
438  * You must call dmu_buf_read, dmu_buf_will_dirty, or dmu_buf_will_fill
439  * on the returned buffer before reading or writing the buffer's
440  * db_data.  The comments for those routines describe what particular
```

```
 441  * operations are valid after calling them.
 442  *
 443  * The object number must be a valid, allocated object number.
 444  */
 445 int dmu_buf_hold(objset_t *os, uint64_t object, uint64_t offset,
 446     void *tag, dmu_buf_t **, int flags);
 447 void dmu_buf_add_ref(dmu_buf_t *db, void* tag);
 448 void dmu_buf_rele(dmu_buf_t *db, void *tag);
 449 uint64_t dmu_buf_refcount(dmu_buf_t *db);

 451 /*
 452  * dmu_buf_hold_array holds the DMU buffers which contain all bytes in a
 453  * range of an object.  A pointer to an array of dmu_buf_t*'s is
 454  * returned (in *dbpp).
 455  *
 456  * dmu_buf_rele_array releases the hold on an array of dmu_buf_t*'s, and
 457  * frees the array.  The hold on the array of buffers MUST be released
 458  * with dmu_buf_rele_array.  You can NOT release the hold on each buffer
 459  * individually with dmu_buf_rele.
 460  */
 461 int dmu_buf_hold_array_by_bonus(dmu_buf_t *db, uint64_t offset,
 462     uint64_t length, int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp);
 463 void dmu_buf_rele_array(dmu_buf_t **, int numbufs, void *tag);

 465 /*
 466  * Returns NULL on success, or the existing user ptr if it's already
 467  * been set.
 468  *
 469  * user_ptr is for use by the user and can be obtained via dmu_buf_get_user().
 470  *
 471  * user_data_ptr_ptr should be NULL, or a pointer to a pointer which
 472  * will be set to db->db_data when you are allowed to access it.  Note
 473  * that db->db_data (the pointer) can change when you do dmu_buf_read(),
 474  * dmu_buf_tryupgrade(), dmu_buf_will_dirty(), or dmu_buf_will_fill().
 475  * *user_data_ptr_ptr will be set to the new value when it changes.
 476  *
 477  * If non-NULL, pageout func will be called when this buffer is being
 478  * excised from the cache, so that you can clean up the data structure
 479  * pointed to by user_ptr.
 480  *
 481  * dmu_evict_user() will call the pageout func for all buffers in a
 482  * objset with a given pageout func.
 483  */
 484 void *dmu_buf_set_user(dmu_buf_t *db, void *user_ptr, void *user_data_ptr_ptr,
 485     dmu_buf_evict_func_t *pageout_func);
 486 /*
 487  * set_user_ie is the same as set_user, but request immediate eviction
 488  * when hold count goes to zero.
 489  */
 490 void *dmu_buf_set_user_ie(dmu_buf_t *db, void *user_ptr,
 491     void *user_data_ptr_ptr, dmu_buf_evict_func_t *pageout_func);
 492 void *dmu_buf_update_user(dmu_buf_t *db_fake, void *old_user_ptr,
 493     void *user_ptr, void *user_data_ptr_ptr,
 494     dmu_buf_evict_func_t *pageout_func);
 495 void dmu_evict_user(objset_t *os, dmu_buf_evict_func_t *func);

 497 /*
 498  * Returns the user_ptr set with dmu_buf_set_user(), or NULL if not set.
 499  */
 500 void *dmu_buf_get_user(dmu_buf_t *db);

 502 /*
 503  * Returns the blkptr associated with this dbuf, or NULL if not set.
 504  */
 505 struct blkptr *dmu_buf_get_blkptr(dmu_buf_t *db);
```



```
 507 /*
 508  * Indicate that you are going to modify the buffer's data (db_data).
 509  *
 510  * The transaction (tx) must be assigned to a txg (ie. you've called
 511  * dmu_tx_assign()).  The buffer's object must be held in the tx
 512  * (ie. you've called dmu_tx_hold_object(tx, db->db_object)).
 513  */
 514 void dmu_buf_will_dirty(dmu_buf_t *db, dmu_tx_t *tx);

 516 /*
 517  * Tells if the given dbuf is freeable.
 518  */
 519 boolean_t dmu_buf_freeable(dmu_buf_t *);

 521 /*
 522  * You must create a transaction, then hold the objects which you will
 523  * (or might) modify as part of this transaction.  Then you must assign
 524  * the transaction to a transaction group.  Once the transaction has
 525  * been assigned, you can modify buffers which belong to held objects as
 526  * part of this transaction.  You can't modify buffers before the
 527  * transaction has been assigned; you can't modify buffers which don't
 528  * belong to objects which this transaction holds; you can't hold
 529  * objects once the transaction has been assigned.  You may hold an
 530  * object which you are going to free (with dmu_object_free()), but you
 531  * don't have to.
 532  *
 533  * You can abort the transaction before it has been assigned.
 534  *
 535  * Note that you may hold buffers (with dmu_buf_hold) at any time,
 536  * regardless of transaction state.
 537  */

 539 #define DMU_NEW_OBJECT  (-1ULL)
 540 #define DMU_OBJECT_END  (-1ULL)

 542 dmu_tx_t *dmu_tx_create(objset_t *os);
 543 void dmu_tx_hold_write(dmu_tx_t *tx, uint64_t object, uint64_t off, int len);
 544 void dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off,
 545     uint64_t len);
 546 void dmu_tx_hold_zap(dmu_tx_t *tx, uint64_t object, int add, const char *name);
 547 void dmu_tx_hold_bonus(dmu_tx_t *tx, uint64_t object);
 548 void dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object);
 549 void dmu_tx_hold_sa(dmu_tx_t *tx, struct sa_handle *hdl, boolean_t may_grow);
 550 void dmu_tx_hold_sa_create(dmu_tx_t *tx, int total_size);
 551 void dmu_tx_abort(dmu_tx_t *tx);
 552 int dmu_tx_assign(dmu_tx_t *tx, enum txg_how txg_how);
 553 void dmu_tx_wait(dmu_tx_t *tx);
 554 void dmu_tx_commit(dmu_tx_t *tx);

 556 /*
 557  * To register a commit callback, dmu_tx_callback_register() must be called.
 558  *
 559  * dcb_data is a pointer to caller private data that is passed on as a
 560  * callback parameter. The caller is responsible for properly allocating and
 561  * freeing it.
 562  *
 563  * When registering a callback, the transaction must be already created, but
 564  * it cannot be committed or aborted. It can be assigned to a txg or not.
 565  *
 566  * The callback will be called after the transaction has been safely written
 567  * to stable storage and will also be called if the dmu_tx is aborted.
 568  * If there is any error which prevents the transaction from being committed to
 569  * disk, the callback will be called with a value of error != 0.
 570  */
 571 typedef void dmu_tx_callback_func_t(void *dcb_data, int error);
```

```
573 void dmu_tx_callback_register(dmu_tx_t *tx, dmu_tx_callback_func_t *dcb_func,
574     void *dcb_data);

576 /*
577  * Free up the data blocks for a defined range of a file.  If size is
578  * -1, the range from offset to end-of-file is freed.
579  */
580 int dmu_free_range(objset_t *os, uint64_t object, uint64_t offset,
581         uint64_t size, dmu_tx_t *tx);
582 int dmu_free_long_range(objset_t *os, uint64_t object, uint64_t offset,
583         uint64_t size);
584 int dmu_free_long_object(objset_t *os, uint64_t object);
584 int dmu_free_object(objset_t *os, uint64_t object);

586 /*
587  * Convenience functions.
588  *
589  * Canfail routines will return 0 on success, or an errno if there is a
590  * nonrecoverable I/O error.
591  */
592 #define DMU_READ_PREFETCH       0 /* prefetch */
593 #define DMU_READ_NO_PREFETCH    1 /* don't prefetch */
594 int dmu_read(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
595         void *buf, uint32_t flags);
596 void dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
597         const void *buf, dmu_tx_t *tx);
598 void dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
599         dmu_tx_t *tx);
600 int dmu_read_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size);
601 int dmu_write_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size,
602     dmu_tx_t *tx);
603 int dmu_write_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size,
604     dmu_tx_t *tx);
605 int dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset,
606     uint64_t size, struct page *pp, dmu_tx_t *tx);
607 struct arc_buf *dmu_request_arcbuf(dmu_buf_t *handle, int size);
608 void dmu_return_arcbuf(struct arc_buf *buf);
609 void dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, struct arc_buf *buf,
610     dmu_tx_t *tx);
611 int dmu_xuio_init(struct xuio *uio, int niov);
612 void dmu_xuio_fini(struct xuio *uio);
613 int dmu_xuio_add(struct xuio *uio, struct arc_buf *abuf, offset_t off,
614     size_t n);
615 int dmu_xuio_cnt(struct xuio *uio);
616 struct arc_buf *dmu_xuio_arcbuf(struct xuio *uio, int i);
617 void dmu_xuio_clear(struct xuio *uio, int i);
618 void xuio_stat_wbuf_copied();
619 void xuio_stat_wbuf_nocopy();

621 extern int zfs_prefetch_disable;

623 /*
624  * Asynchronously try to read in the data.
625  */
626 void dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset,
627     uint64_t len);

629 typedef struct dmu_object_info {
630         /* All sizes are in bytes unless otherwise indicated. */
631         uint32_t doi_data_block_size;
632         uint32_t doi_metadata_block_size;
633         dmu_object_type_t doi_type;
634         dmu_object_type_t doi_bonus_type;
635         uint64_t doi_bonus_size;
636         uint8_t doi_indirection;                /* 2 = dnode->indirect->data */
637         uint8_t doi_checksum;
```



```
638         uint8_t doi_compress;
639         uint8_t doi_pad[5];
640         uint64_t doi_physical_blocks_512;       /* data + metadata, 512b blks */
641         uint64_t doi_max_offset;
642         uint64_t doi_fill_count;                /* number of non-empty blocks */
643 } dmu_object_info_t;
```

unchanged_portion_omitted_

```
**********************************************************
   10649 Thu Aug 15 17:44:07 2013
new/usr/src/uts/common/fs/zfs/sys/dnode.h
4047 panic from dbuf_free_range() from dmu_free_object() while doing zfs receive
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright (c) 2013 by Delphix. All rights reserved.
  23  * Copyright (c) 2012 by Delphix. All rights reserved.
  24  */

  26 #ifndef _SYS_DNODE_H
  27 #define _SYS_DNODE_H

  29 #include <sys/zfs_context.h>
  30 #include <sys/avl.h>
  31 #include <sys/spa.h>
  32 #include <sys/txg.h>
  33 #include <sys/zio.h>
  34 #include <sys/refcount.h>
  35 #include <sys/dmu_zfetch.h>
  36 #include <sys/zrlock.h>

  38 #ifdef  __cplusplus
  39 extern "C" {
  40 #endif

  42 /*
  43  * dnode_hold() flags.
  44  */
  45 #define DNODE_MUST_BE_ALLOCATED 1
  46 #define DNODE_MUST_BE_FREE      2

  48 /*
  49  * dnode_next_offset() flags.
  50  */
  51 #define DNODE_FIND_HOLE         1
  52 #define DNODE_FIND_BACKWARDS    2
  53 #define DNODE_FIND_HAVELOCK     4

  55 /*
  56  * Fixed constants.
  57  */
  58 #define DNODE_SHIFT             9       /* 512 bytes */
```



```
  59 #define DN_MIN_INDBLKSHIFT      10      /* 1k */
  60 #define DN_MAX_INDBLKSHIFT      14      /* 16k */
  61 #define DNODE_BLOCK_SHIFT       14      /* 16k */
  62 #define DNODE_CORE_SIZE         64      /* 64 bytes for dnode sans blkptrs */
  63 #define DN_MAX_OBJECT_SHIFT     48      /* 256 trillion (zfs_fid_t limit) */
  64 #define DN_MAX_OFFSET_SHIFT     64      /* 2^64 bytes in a dnode */

  66 /*
  67  * dnode id flags
  68  *
  69  * Note: a file will never ever have its
  70  * ids moved from bonus->spill
  71  * and only in a crypto environment would it be on spill
  72  */
  73 #define DN_ID_CHKED_BONUS       0x1
  74 #define DN_ID_CHKED_SPILL       0x2
  75 #define DN_ID_OLD_EXIST         0x4
  76 #define DN_ID_NEW_EXIST         0x8

  78 /*
  79  * Derived constants.
  80  */
  81 #define DNODE_SIZE      (1 << DNODE_SHIFT)
  82 #define DN_MAX_NBLKPTR  ((DNODE_SIZE - DNODE_CORE_SIZE) >> SPA_BLKPTRSHIFT)
  83 #define DN_MAX_BONUSLEN (DNODE_SIZE - DNODE_CORE_SIZE - (1 << SPA_BLKPTRSHIFT))
  84 #define DN_MAX_OBJECT   (1ULL << DN_MAX_OBJECT_SHIFT)
  85 #define DN_ZERO_BONUSLEN        (DN_MAX_BONUSLEN + 1)
  86 #define DN_KILL_SPILLBLK (1)

  88 #define DNODES_PER_BLOCK_SHIFT  (DNODE_BLOCK_SHIFT - DNODE_SHIFT)
  89 #define DNODES_PER_BLOCK        (1ULL << DNODES_PER_BLOCK_SHIFT)
  90 #define DNODES_PER_LEVEL_SHIFT  (DN_MAX_INDBLKSHIFT - SPA_BLKPTRSHIFT)
  91 #define DNODES_PER_LEVEL        (1ULL << DNODES_PER_LEVEL_SHIFT)

  93 /* The +2 here is a cheesy way to round up */
  94 #define DN_MAX_LEVELS   (2 + ((DN_MAX_OFFSET_SHIFT - SPA_MINBLOCKSHIFT) / \
  95         (DN_MIN_INDBLKSHIFT - SPA_BLKPTRSHIFT)))

  97 #define DN_BONUS(dnp)   ((void*)((dnp)->dn_bonus + \
  98         (((dnp)->dn_nblkptr - 1) * sizeof (blkptr_t))))

 100 #define DN_USED_BYTES(dnp) (((dnp)->dn_flags & DNODE_FLAG_USED_BYTES) ? \
 101         (dnp)->dn_used : (dnp)->dn_used << SPA_MINBLOCKSHIFT)

 103 #define EPB(blkshift, typeshift)        (1 << (blkshift - typeshift))

 105 struct dmu_buf_impl;
 106 struct objset;
 107 struct zio;

 109 enum dnode_dirtycontext {
 110         DN_UNDIRTIED,
 111         DN_DIRTY_OPEN,
 112         DN_DIRTY_SYNC
 113 };
_____unchanged_portion_omitted_

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

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

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

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

189         /* protected by dn_dbufs_mtx; declared here to fill 32-bit hole */
190         uint32_t dn_dbufs_count;        /* count of dn_dbufs */
191         /* There are no level-0 blocks of this blkid or higher in dn_dbufs */
192         uint64_t dn_unlisted_l0_blkid;

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

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

208         /* protected by own devices */
209         refcount_t dn_tx_holds;
210         refcount_t dn_holds;

212         kmutex_t dn_dbufs_mtx;
213         list_t dn_dbufs;                /* descendent dbufs */

215         /* protected by dn_struct_rwlock */
216         struct dmu_buf_impl *dn_bonus;  /* bonus buffer dbuf */

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

220         /* parent IO for current sync write */
```



```
221         zio_t *dn_zio;

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

230         /* holds prefetch structure */
231         struct zfetch   dn_zfetch;
232 } dnode_t;
```

*unchanged_portion_omitted_*