```
**********************************************************
   35483 Fri Aug 17 09:23:31 2012
new/usr/src/uts/common/fs/zfs/dmu_tx.c
1862 incremental zfs receive fails for sparse file > 8PB
dmu_tx_count_free is doing a horrible over-estimation of used memory. It
assumes that the file is fully non-sparse and calculates a worst-case estimate
of how much memory is needed to hold all metadata for the file. If a large
hole needs to be freed, the estimation goes into the TB-range, which obviously
fails later on.
This patch tries to calculate a more realistic estimate by counting the l1
blocks (the loop for this is already present) and assumes a worst-case
distribution of those blocks over the full length given.
Reviewed by: Matt Ahrens <matthew.ahrens@delphix.com>
Reviewed by: Simon Klinkert <klinkert@webgods.de>
**********************************************************
______unchanged_portion_omitted_

static void
dmu_tx_count_free(dmu_tx_hold_t *txh, uint64_t off, uint64_t len)
{
        uint64_t blkid, nblks, lastblk;
        uint64_t space = 0, unref = 0, skipped = 0;
        dnode_t *dn = txh->txh_dnode;
        dsl_dataset_t *ds = dn->dn_objset->os_dsl_dataset;
        spa_t *spa = txh->txh_tx->tx_pool->dp_spa;
        int epbs;
        uint64_t l0span = 0, nl1blks = 0;
#endif /* ! codereview */

        if (dn->dn_nlevels == 0)
                return;

        /*
         * The struct_rwlock protects us against dn_nlevels
         * changing, in case (against all odds) we manage to dirty &
         * sync out the changes after we check for being dirty.
         * Also, dbuf_hold_impl() wants us to have the struct_rwlock.
         */
        rw_enter(&dn->dn_struct_rwlock, RW_READER);
        epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
        if (dn->dn_maxblkid == 0) {
                if (off == 0 && len >= dn->dn_datablksz) {
                        blkid = 0;
                        nblks = 1;
                } else {
                        rw_exit(&dn->dn_struct_rwlock);
                        return;
                }
        } else {
                blkid = off >> dn->dn_datablkshift;
                nblks = (len + dn->dn_datablksz - 1) >> dn->dn_datablkshift;

                if (blkid >= dn->dn_maxblkid) {
                        rw_exit(&dn->dn_struct_rwlock);
                        return;
                }
                if (blkid + nblks > dn->dn_maxblkid)
                        nblks = dn->dn_maxblkid - blkid;

        }
        l0span = nblks;    /* save for later use to calc level > 1 overhead */
#endif /* ! codereview */
        if (dn->dn_nlevels == 1) {
                int i;
                for (i = 0; i < nblks; i++) {
                        blkptr_t *bp = dn->dn_phys->dn_blkptr;
```

```
                        ASSERT3U(blkid + i, <, dn->dn_nblkptr);
                        bp += blkid + i;
                        if (dsl_dataset_block_freeable(ds, bp, bp->blk_birth)) {
                                dprintf_bp(bp, "can free old%s", "");
                                space += bp_get_dsize(spa, bp);
                        }
                        unref += BP_GET_ASIZE(bp);
                }
                nl1blks = 1;
#endif /* ! codereview */
                nblks = 0;
        }

        /*
         * Add in memory requirements of higher-level indirects.
         * This assumes a worst-possible scenario for dn_nlevels.
         */
        {
                uint64_t blkcnt = 1 + ((nblks >> epbs) >> epbs);
                int level = (dn->dn_nlevels > 1) ? 2 : 1;

                while (level++ < DN_MAX_LEVELS) {
                        txh->txh_memory_tohold += blkcnt << dn->dn_indblkshift;
                        blkcnt = 1 + (blkcnt >> epbs);
                }
                ASSERT(blkcnt <= dn->dn_nblkptr);
        }

        lastblk = blkid + nblks - 1;
        while (nblks) {
                dmu_buf_impl_t *dbuf;
                uint64_t ibyte, new_blkid;
                int epb = 1 << epbs;
                int err, i, blkoff, tochk;
                blkptr_t *bp;

                ibyte = blkid << dn->dn_datablkshift;
                err = dnode_next_offset(dn,
                    DNODE_FIND_HAVELOCK, &ibyte, 2, 1, 0);
                new_blkid = ibyte >> dn->dn_datablkshift;
                if (err == ESRCH) {
                        skipped += (lastblk >> epbs) - (blkid >> epbs) + 1;
                        break;
                }
                if (err) {
                        txh->txh_tx->tx_err = err;
                        break;
                }
                if (new_blkid > lastblk) {
                        skipped += (lastblk >> epbs) - (blkid >> epbs) + 1;
                        break;
                }

                if (new_blkid > blkid) {
                        ASSERT((new_blkid >> epbs) > (blkid >> epbs));
                        skipped += (new_blkid >> epbs) - (blkid >> epbs) - 1;
                        nblks -= new_blkid - blkid;
                        blkid = new_blkid;
                }
                blkoff = P2PHASE(blkid, epb);
                tochk = MIN(epb - blkoff, nblks);

                err = dbuf_hold_impl(dn, 1, blkid >> epbs, FALSE, FTAG, &dbuf);
                if (err) {
                        txh->txh_tx->tx_err = err;
                        break;
                }

                txh->txh_memory_tohold += dbuf->db.db_size;

                /*
                 * We don’t check memory_tohold against DMU_MAX_ACCESS because
                 * memory_tohold is an over-estimation (especially the >L1
                 * indirect blocks), so it could fail.  Callers should have
                 * already verified that they will not be holding too much
                 * memory.
                 */

                err = dbuf_read(dbuf, NULL, DB_RF_HAVESTRUCT | DB_RF_CANFAIL);
                if (err != 0) {
                        txh->txh_tx->tx_err = err;
                        dbuf_rele(dbuf, FTAG);
                        break;
                }

                bp = dbuf->db.db_data;
                bp += blkoff;

                for (i = 0; i < tochk; i++) {
                        if (dsl_dataset_block_freeable(ds, &bp[i],
                            bp[i].blk_birth)) {
                                dprintf_bp(&bp[i], "can free old%s", "");
                                space += bp_get_dsize(spa, &bp[i]);
                        }
                        unref += BP_GET_ASIZE(bp);
                }
                dbuf_rele(dbuf, FTAG);

                ++nl1blks;
#endif /* ! codereview */
                blkid += tochk;
                nblks -= tochk;
        }
        rw_exit(&dn->dn_struct_rwlock);

        /*
         * Add in memory requirements of higher-level indirects.
         * This assumes a worst-possible scenario for dn_nlevels and a
         * worst-possible distribution of l1-blocks over the region to free.
         */
        {
                uint64_t blkcnt = 1 + ((l0span >> epbs) >> epbs);
                int level = 2;
                /*
                 * Here we don’t use DN_MAX_LEVEL, but calculate it with the
                 * given datablkshift and indblkshift. This makes the
                 * difference between 19 and 8 on large files.
                 */
                int maxlevel = 2 + (DN_MAX_OFFSET_SHIFT - dn->dn_datablkshift) /
                    (dn->dn_indblkshift - SPA_BLKPTRSHIFT);

                while (level++ < maxlevel) {
                        txh->txh_memory_tohold += MIN(blkcnt, (nl1blks >> epbs))
                            << dn->dn_indblkshift;
                        blkcnt = 1 + (blkcnt >> epbs);
                }
        }

#endif /* ! codereview */
        /* account for new level 1 indirect blocks that might show up */
        if (skipped > 0) {
                txh->txh_fudge += skipped << dn->dn_indblkshift;
```

```
                skipped = MIN(skipped, DMU_MAX_DELETEBLKCNT >> epbs);
                txh->txh_memory_tohold += skipped << dn->dn_indblkshift;
        }
        txh->txh_space_tofree += space;
        txh->txh_space_tounref += unref;
}

void
dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off, uint64_t len)
{
        dmu_tx_hold_t *txh;
        dnode_t *dn;
        uint64_t start, end, i;
        int err, shift;
        zio_t *zio;

        ASSERT(tx->tx_txg == 0);

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

        /* first block */
        if (off != 0)
                dmu_tx_count_write(txh, off, 1);
        /* last block */
        if (len != DMU_OBJECT_END)
                dmu_tx_count_write(txh, off+len, 1);

        dmu_tx_count_dnode(txh);

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

        /*
         * For i/o error checking, read the first and last level-0
         * blocks, and all the level-1 blocks.  The above count_write’s
         * have already taken care of the level-0 blocks.
         */
        if (dn->dn_nlevels > 1) {
                shift = dn->dn_datablkshift + dn->dn_indblkshift -
                    SPA_BLKPTRSHIFT;
                start = off >> shift;
                end = dn->dn_datablkshift ? ((off+len) >> shift) : 0;

                zio = zio_root(tx->tx_pool->dp_spa,
                    NULL, NULL, ZIO_FLAG_CANFAIL);
                for (i = start; i <= end; i++) {
                        uint64_t ibyte = i << shift;
                        err = dnode_next_offset(dn, 0, &ibyte, 2, 1, 0);
                        i = ibyte >> shift;
                        if (err == ESRCH)
                                break;
                        if (err) {
                                tx->tx_err = err;
                                return;
                        }

                        err = dmu_tx_check_ioerr(zio, dn, 1, i);
                        if (err) {
                                tx->tx_err = err;
                                return;
                        }
                }
                err = zio_wait(zio);
                if (err) {
                        tx->tx_err = err;
                        return;
                }
        }

        dmu_tx_count_free(txh, off, len);
}

void
dmu_tx_hold_zap(dmu_tx_t *tx, uint64_t object, int add, const char *name)
{
        dmu_tx_hold_t *txh;
        dnode_t *dn;
        uint64_t nblocks;
        int epbs, err;

        ASSERT(tx->tx_txg == 0);

        txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
            object, THT_ZAP, add, (uintptr_t)name);
        if (txh == NULL)
                return;
        dn = txh->txh_dnode;

        dmu_tx_count_dnode(txh);

        if (dn == NULL) {
                /*
                 * We will be able to fit a new object's entries into one leaf
                 * block.  So there will be at most 2 blocks total,
                 * including the header block.
                 */
                dmu_tx_count_write(txh, 0, 2 << fzap_default_block_shift);
                return;
        }

        ASSERT3P(DMU_OT_BYTESWAP(dn->dn_type), ==, DMU_BSWAP_ZAP);

        if (dn->dn_maxblkid == 0 && !add) {
                blkptr_t *bp;

                /*
                 * If there is only one block  (i.e. this is a micro-zap)
                 * and we are not adding anything, the accounting is simple.
                 */
                err = dmu_tx_check_ioerr(NULL, dn, 0, 0);
                if (err) {
                        tx->tx_err = err;
                        return;
                }

                /*
                 * Use max block size here, since we don't know how much
                 * the size will change between now and the dbuf dirty call.
                 */
                bp = &dn->dn_phys->dn_blkptr[0];
                if (dsl_dataset_block_freeable(dn->dn_objset->os_dsl_dataset,
                    bp, bp->blk_birth))
                        txh->txh_space_tooverwrite += SPA_MAXBLOCKSIZE;
                else
                        txh->txh_space_towrite += SPA_MAXBLOCKSIZE;
                if (!BP_IS_HOLE(bp))
```

```
                        txh->txh_space_tounref += SPA_MAXBLOCKSIZE;
                return;
        }

        if (dn->dn_maxblkid > 0 && name) {
                /*
                 * access the name in this fat-zap so that we'll check
                 * for i/o errors to the leaf blocks, etc.
                 */
                err = zap_lookup(dn->dn_objset, dn->dn_object, name,
                    8, 0, NULL);
                if (err == EIO) {
                        tx->tx_err = err;
                        return;
                }
        }

        err = zap_count_write(dn->dn_objset, dn->dn_object, name, add,
            &txh->txh_space_towrite, &txh->txh_space_tooverwrite);

        /*
         * If the modified blocks are scattered to the four winds,
         * we'll have to modify an indirect twig for each.
         */
        epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
        for (nblocks = dn->dn_maxblkid >> epbs; nblocks != 0; nblocks >>= epbs)
                if (dn->dn_objset->os_dsl_dataset->ds_phys->ds_prev_snap_obj)
                        txh->txh_space_towrite += 3 << dn->dn_indblkshift;
                else
                        txh->txh_space_tooverwrite += 3 << dn->dn_indblkshift;
}

void
dmu_tx_hold_bonus(dmu_tx_t *tx, uint64_t object)
{
        dmu_tx_hold_t *txh;

        ASSERT(tx->tx_txg == 0);

        txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
            object, THT_BONUS, 0, 0);
        if (txh)
                dmu_tx_count_dnode(txh);
}

void
dmu_tx_hold_space(dmu_tx_t *tx, uint64_t space)
{
        dmu_tx_hold_t *txh;
        ASSERT(tx->tx_txg == 0);

        txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
            DMU_NEW_OBJECT, THT_SPACE, space, 0);

        txh->txh_space_towrite += space;
}

int
dmu_tx_holds(dmu_tx_t *tx, uint64_t object)
{
        dmu_tx_hold_t *txh;
        int holds = 0;

        /*
         * By asserting that the tx is assigned, we're counting the
         * number of dn_tx_holds, which is the same as the number of
         * dn_holds.  Otherwise, we'd be counting dn_holds, but
         * dn_tx_holds could be 0.
         */
        ASSERT(tx->tx_txg != 0);

        /* if (tx->tx_anyobj == TRUE) */
                /* return (0); */

        for (txh = list_head(&tx->tx_holds); txh;
            txh = list_next(&tx->tx_holds, txh)) {
                if (txh->txh_dnode && txh->txh_dnode->dn_object == object)
                        holds++;
        }

        return (holds);
}

#ifdef ZFS_DEBUG
void
dmu_tx_dirty_buf(dmu_tx_t *tx, dmu_buf_impl_t *db)
{
        dmu_tx_hold_t *txh;
        int match_object = FALSE, match_offset = FALSE;
        dnode_t *dn;

        DB_DNODE_ENTER(db);
        dn = DB_DNODE(db);
        ASSERT(tx->tx_txg != 0);
        ASSERT(tx->tx_objset == NULL || dn->dn_objset == tx->tx_objset);
        ASSERT3U(dn->dn_object, ==, db->db.db_object);

        if (tx->tx_anyobj) {
                DB_DNODE_EXIT(db);
                return;
        }

        /* XXX No checking on the meta dnode for now */
        if (db->db.db_object == DMU_META_DNODE_OBJECT) {
                DB_DNODE_EXIT(db);
                return;
        }

        for (txh = list_head(&tx->tx_holds); txh;
            txh = list_next(&tx->tx_holds, txh)) {
                ASSERT(dn == NULL || dn->dn_assigned_txg == tx->tx_txg);
                if (txh->txh_dnode == dn && txh->txh_type != THT_NEWOBJECT)
                        match_object = TRUE;
                if (txh->txh_dnode == NULL || txh->txh_dnode == dn) {
                        int datablkshift = dn->dn_datablkshift ?
                            dn->dn_datablkshift : SPA_MAXBLOCKSHIFT;
                        int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
                        int shift = datablkshift + epbs * db->db_level;
                        uint64_t beginblk = shift >= 64 ? 0 :
                            (txh->txh_arg1 >> shift);
                        uint64_t endblk = shift >= 64 ? 0 :
                            ((txh->txh_arg1 + txh->txh_arg2 - 1) >> shift);
                        uint64_t blkid = db->db_blkid;

                        /* XXX txh_arg2 better not be zero... */

                        dprintf("found txh type %x beginblk=%llx endblk=%llx\n",
                            txh->txh_type, beginblk, endblk);

                        switch (txh->txh_type) {
                        case THT_WRITE:
                                if (blkid >= beginblk && blkid <= endblk)
```

```
                                        match_offset = TRUE;
                                /*
                                 * We will let this hold work for the bonus
                                 * or spill buffer so that we don't need to
                                 * hold it when creating a new object.
                                 */
                                if (blkid == DMU_BONUS_BLKID ||
                                    blkid == DMU_SPILL_BLKID)
                                        match_offset = TRUE;
                                /*
                                 * They might have to increase nlevels,
                                 * thus dirtying the new TLIBs.  Or the
                                 * might have to change the block size,
                                 * thus dirying the new lvl=0 blk=0.
                                 */
                                if (blkid == 0)
                                        match_offset = TRUE;
                                break;
                        case THT_FREE:
                                /*
                                 * We will dirty all the level 1 blocks in
                                 * the free range and perhaps the first and
                                 * last level 0 block.
                                 */
                                if (blkid >= beginblk && (blkid <= endblk ||
                                    txh->txh_arg2 == DMU_OBJECT_END))
                                        match_offset = TRUE;
                                break;
                        case THT_SPILL:
                                if (blkid == DMU_SPILL_BLKID)
                                        match_offset = TRUE;
                                break;
                        case THT_BONUS:
                                if (blkid == DMU_BONUS_BLKID)
                                        match_offset = TRUE;
                                break;
                        case THT_ZAP:
                                match_offset = TRUE;
                                break;
                        case THT_NEWOBJECT:
                                match_object = TRUE;
                                break;
                        default:
                                ASSERT(!"bad txh_type");
                        }
                }
                if (match_object && match_offset) {
                        DB_DNODE_EXIT(db);
                        return;
                }
        }
        DB_DNODE_EXIT(db);
        panic("dirtying dbuf obj=%llx lvl=%u blkid=%llx but not tx_held\n",
            (u_longlong_t)db->db.db_object, db->db_level,
            (u_longlong_t)db->db_blkid);
}
#endif

static int
dmu_tx_try_assign(dmu_tx_t *tx, uint64_t txg_how)
{
        dmu_tx_hold_t *txh;
        spa_t *spa = tx->tx_pool->dp_spa;
        uint64_t memory, asize, fsize, usize;
        uint64_t towrite, tofree, tooverwrite, tounref, tohold, fudge;
        ASSERT3U(tx->tx_txg, ==, 0);

        if (tx->tx_err)
                return (tx->tx_err);

        if (spa_suspended(spa)) {
                /*
                 * If the user has indicated a blocking failure mode
                 * then return ERESTART which will block in dmu_tx_wait().
                 * Otherwise, return EIO so that an error can get
                 * propagated back to the VOP calls.
                 *
                 * Note that we always honor the txg_how flag regardless
                 * of the failuremode setting.
                 */
                if (spa_get_failmode(spa) == ZIO_FAILURE_MODE_CONTINUE &&
                    txg_how != TXG_WAIT)
                        return (EIO);

                return (ERESTART);
        }

        tx->tx_txg = txg_hold_open(tx->tx_pool, &tx->tx_txgh);
        tx->tx_needassign_txh = NULL;

        /*
         * NB: No error returns are allowed after txg_hold_open, but
         * before processing the dnode holds, due to the
         * dmu_tx_unassign() logic.
         */

        towrite = tofree = tooverwrite = tounref = tohold = fudge = 0;
        for (txh = list_head(&tx->tx_holds); txh;
            txh = list_next(&tx->tx_holds, txh)) {
                dnode_t *dn = txh->txh_dnode;
                if (dn != NULL) {
                        mutex_enter(&dn->dn_mtx);
                        if (dn->dn_assigned_txg == tx->tx_txg - 1) {
                                mutex_exit(&dn->dn_mtx);
                                tx->tx_needassign_txh = txh;
                                return (ERESTART);
                        }
                        if (dn->dn_assigned_txg == 0)
                                dn->dn_assigned_txg = tx->tx_txg;
                        ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);
                        (void) refcount_add(&dn->dn_tx_holds, tx);
                        mutex_exit(&dn->dn_mtx);
                }
                towrite += txh->txh_space_towrite;
                tofree += txh->txh_space_tofree;
                tooverwrite += txh->txh_space_tooverwrite;
                tounref += txh->txh_space_tounref;
                tohold += txh->txh_memory_tohold;
                fudge += txh->txh_fudge;
        }

        /*
         * NB: This check must be after we've held the dnodes, so that
         * the dmu_tx_unassign() logic will work properly
         */
        if (txg_how >= TXG_INITIAL && txg_how != tx->tx_txg)
                return (ERESTART);

        /*
         * If a snapshot has been taken since we made our estimates,
         * assume that we won't be able to free or overwrite anything.
```

```
         */
        if (tx->tx_objset &&
            dsl_dataset_prev_snap_txg(tx->tx_objset->os_dsl_dataset) >
            tx->tx_lastsnap_txg) {
                towrite += tooverwrite;
                tooverwrite = tofree = 0;
        }

        /* needed allocation: worst-case estimate of write space */
        asize = spa_get_asize(tx->tx_pool->dp_spa, towrite + tooverwrite);
        /* freed space estimate: worst-case overwrite + free estimate */
        fsize = spa_get_asize(tx->tx_pool->dp_spa, tooverwrite) + tofree;
        /* convert unrefd space to worst-case estimate */
        usize = spa_get_asize(tx->tx_pool->dp_spa, tounref);
        /* calculate memory footprint estimate */
        memory = towrite + tooverwrite + tohold;

#ifdef ZFS_DEBUG
        /*
         * Add in 'tohold' to account for our dirty holds on this memory
         * XXX - the "fudge" factor is to account for skipped blocks that
         * we missed because dnode_next_offset() misses in-core-only blocks.
         */
        tx->tx_space_towrite = asize +
            spa_get_asize(tx->tx_pool->dp_spa, tohold + fudge);
        tx->tx_space_tofree = tofree;
        tx->tx_space_tooverwrite = tooverwrite;
        tx->tx_space_tounref = tounref;
#endif

        if (tx->tx_dir && asize != 0) {
                int err = dsl_dir_tempreserve_space(tx->tx_dir, memory,
                    asize, fsize, usize, &tx->tx_tempreserve_cookie, tx);
                if (err)
                        return (err);
        }

        return (0);
}

static void
dmu_tx_unassign(dmu_tx_t *tx)
{
        dmu_tx_hold_t *txh;

        if (tx->tx_txg == 0)
                return;

        txg_rele_to_quiesce(&tx->tx_txgh);

        for (txh = list_head(&tx->tx_holds); txh != tx->tx_needassign_txh;
            txh = list_next(&tx->tx_holds, txh)) {
                dnode_t *dn = txh->txh_dnode;

                if (dn == NULL)
                        continue;
                mutex_enter(&dn->dn_mtx);
                ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);

                if (refcount_remove(&dn->dn_tx_holds, tx) == 0) {
                        dn->dn_assigned_txg = 0;
                        cv_broadcast(&dn->dn_notxholds);
                }
                mutex_exit(&dn->dn_mtx);
        }
        txg_rele_to_sync(&tx->tx_txgh);

        tx->tx_lasttried_txg = tx->tx_txg;
        tx->tx_txg = 0;
}

/*
 * Assign tx to a transaction group.  txg_how can be one of:
 *
 * (1)  TXG_WAIT.  If the current open txg is full, waits until there’s
 *      a new one.  This should be used when you’re not holding locks.
 *      If will only fail if we’re truly out of space (or over quota).
 *
 * (2)  TXG_NOWAIT.  If we can’t assign into the current open txg without
 *      blocking, returns immediately with ERESTART.  This should be used
 *      whenever you’re holding locks.  On an ERESTART error, the caller
 *      should drop locks, do a dmu_tx_wait(tx), and try again.
 *
 * (3)  A specific txg.  Use this if you need to ensure that multiple
 *      transactions all sync in the same txg.  Like TXG_NOWAIT, it
 *      returns ERESTART if it can’t assign you into the requested txg.
 */
int
dmu_tx_assign(dmu_tx_t *tx, uint64_t txg_how)
{
        int err;

        ASSERT(tx->tx_txg == 0);
        ASSERT(txg_how != 0);
        ASSERT(!dsl_pool_sync_context(tx->tx_pool));

        while ((err = dmu_tx_try_assign(tx, txg_how)) != 0) {
                dmu_tx_unassign(tx);

                if (err != ERESTART || txg_how != TXG_WAIT)
                        return (err);

                dmu_tx_wait(tx);
        }

        txg_rele_to_quiesce(&tx->tx_txgh);

        return (0);
}

void
dmu_tx_wait(dmu_tx_t *tx)
{
        spa_t *spa = tx->tx_pool->dp_spa;

        ASSERT(tx->tx_txg == 0);

        /*
         * It’s possible that the pool has become active after this thread
         * has tried to obtain a tx. If that’s the case then his
         * tx_lasttried_txg would not have been assigned.
         */
        if (spa_suspended(spa) || tx->tx_lasttried_txg == 0) {
                txg_wait_synced(tx->tx_pool, spa_last_synced_txg(spa) + 1);
        } else if (tx->tx_needassign_txh) {
                dnode_t *dn = tx->tx_needassign_txh->txh_dnode;

                mutex_enter(&dn->dn_mtx);
                while (dn->dn_assigned_txg == tx->tx_lasttried_txg - 1)
                        cv_wait(&dn->dn_notxholds, &dn->dn_mtx);
                mutex_exit(&dn->dn_mtx);
```

```
                tx->tx_needassign_txh = NULL;
        } else {
                txg_wait_open(tx->tx_pool, tx->tx_lasttried_txg + 1);
        }
}

void
dmu_tx_willuse_space(dmu_tx_t *tx, int64_t delta)
{
#ifdef ZFS_DEBUG
        if (tx->tx_dir == NULL || delta == 0)
                return;

        if (delta > 0) {
                ASSERT3U(refcount_count(&tx->tx_space_written) + delta, <=,
                    tx->tx_space_towrite);
                (void) refcount_add_many(&tx->tx_space_written, delta, NULL);
        } else {
                (void) refcount_add_many(&tx->tx_space_freed, -delta, NULL);
        }
#endif
}

void
dmu_tx_commit(dmu_tx_t *tx)
{
        dmu_tx_hold_t *txh;

        ASSERT(tx->tx_txg != 0);

        while (txh = list_head(&tx->tx_holds)) {
                dnode_t *dn = txh->txh_dnode;

                list_remove(&tx->tx_holds, txh);
                kmem_free(txh, sizeof (dmu_tx_hold_t));
                if (dn == NULL)
                        continue;
                mutex_enter(&dn->dn_mtx);
                ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);

                if (refcount_remove(&dn->dn_tx_holds, tx) == 0) {
                        dn->dn_assigned_txg = 0;
                        cv_broadcast(&dn->dn_notxholds);
                }
                mutex_exit(&dn->dn_mtx);
                dnode_rele(dn, tx);
        }

        if (tx->tx_tempreserve_cookie)
                dsl_dir_tempreserve_clear(tx->tx_tempreserve_cookie, tx);

        if (!list_is_empty(&tx->tx_callbacks))
                txg_register_callbacks(&tx->tx_txgh, &tx->tx_callbacks);

        if (tx->tx_anyobj == FALSE)
                txg_rele_to_sync(&tx->tx_txgh);

        list_destroy(&tx->tx_callbacks);
        list_destroy(&tx->tx_holds);
#ifdef ZFS_DEBUG
        dprintf("towrite=%llu written=%llu tofree=%llu freed=%llu\n",
            tx->tx_space_towrite, refcount_count(&tx->tx_space_written),
            tx->tx_space_tofree, refcount_count(&tx->tx_space_freed));
        refcount_destroy_many(&tx->tx_space_written,
            refcount_count(&tx->tx_space_written));
        refcount_destroy_many(&tx->tx_space_freed,
            refcount_count(&tx->tx_space_freed));
#endif
        kmem_free(tx, sizeof (dmu_tx_t));
}

void
dmu_tx_abort(dmu_tx_t *tx)
{
        dmu_tx_hold_t *txh;

        ASSERT(tx->tx_txg == 0);

        while (txh = list_head(&tx->tx_holds)) {
                dnode_t *dn = txh->txh_dnode;

                list_remove(&tx->tx_holds, txh);
                kmem_free(txh, sizeof (dmu_tx_hold_t));
                if (dn != NULL)
                        dnode_rele(dn, tx);
        }

        /*
         * Call any registered callbacks with an error code.
         */
        if (!list_is_empty(&tx->tx_callbacks))
                dmu_tx_do_callbacks(&tx->tx_callbacks, ECANCELED);

        list_destroy(&tx->tx_callbacks);
        list_destroy(&tx->tx_holds);
#ifdef ZFS_DEBUG
        refcount_destroy_many(&tx->tx_space_written,
            refcount_count(&tx->tx_space_written));
        refcount_destroy_many(&tx->tx_space_freed,
            refcount_count(&tx->tx_space_freed));
#endif
        kmem_free(tx, sizeof (dmu_tx_t));
}

uint64_t
dmu_tx_get_txg(dmu_tx_t *tx)
{
        ASSERT(tx->tx_txg != 0);
        return (tx->tx_txg);
}

void
dmu_tx_callback_register(dmu_tx_t *tx, dmu_tx_callback_func_t *func, void *data)
{
        dmu_tx_callback_t *dcb;

        dcb = kmem_alloc(sizeof (dmu_tx_callback_t), KM_SLEEP);

        dcb->dcb_func = func;
        dcb->dcb_data = data;

        list_insert_tail(&tx->tx_callbacks, dcb);
}

/*
 * Call all the commit callbacks on a list, with a given error code.
 */
void
dmu_tx_do_callbacks(list_t *cb_list, int error)
{
        dmu_tx_callback_t *dcb;
```

```
        while (dcb = list_head(cb_list)) {
                list_remove(cb_list, dcb);
                dcb->dcb_func(dcb->dcb_data, error);
                kmem_free(dcb, sizeof (dmu_tx_callback_t));
        }
}

/*
 * Interface to hold a bunch of attributes.
 * used for creating new files.
 * attrsize is the total size of all attributes
 * to be added during object creation
 *
 * For updating/adding a single attribute dmu_tx_hold_sa() should be used.
 */

/*
 * hold necessary attribute name for attribute registration.
 * should be a very rare case where this is needed.  If it does
 * happen it would only happen on the first write to the file system.
 */
static void
dmu_tx_sa_registration_hold(sa_os_t *sa, dmu_tx_t *tx)
{
        int i;

        if (!sa->sa_need_attr_registration)
                return;

        for (i = 0; i != sa->sa_num_attrs; i++) {
                if (!sa->sa_attr_table[i].sa_registered) {
                        if (sa->sa_reg_attr_obj)
                                dmu_tx_hold_zap(tx, sa->sa_reg_attr_obj,
                                    B_TRUE, sa->sa_attr_table[i].sa_name);
                        else
                                dmu_tx_hold_zap(tx, DMU_NEW_OBJECT,
                                    B_TRUE, sa->sa_attr_table[i].sa_name);
                }
        }
}


void
dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object)
{
        dnode_t *dn;
        dmu_tx_hold_t *txh;

        txh = dmu_tx_hold_object_impl(tx, tx->tx_objset, object,
            THT_SPILL, 0, 0);

        dn = txh->txh_dnode;

        if (dn == NULL)
                return;

        /* If blkptr doesn’t exist then add space to towrite */
        if (!(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR)) {
                txh->txh_space_towrite += SPA_MAXBLOCKSIZE;
        } else {
                blkptr_t *bp;

                bp = &dn->dn_phys->dn_spill;
                if (dsl_dataset_block_freeable(dn->dn_objset->os_dsl_dataset,
                    bp, bp->blk_birth))
                        txh->txh_space_tooverwrite += SPA_MAXBLOCKSIZE;
                else
                        txh->txh_space_towrite += SPA_MAXBLOCKSIZE;
                if (!BP_IS_HOLE(bp))
                        txh->txh_space_tounref += SPA_MAXBLOCKSIZE;
        }
}

void
dmu_tx_hold_sa_create(dmu_tx_t *tx, int attrsize)
{
        sa_os_t *sa = tx->tx_objset->os_sa;

        dmu_tx_hold_bonus(tx, DMU_NEW_OBJECT);

        if (tx->tx_objset->os_sa->sa_master_obj == 0)
                return;

        if (tx->tx_objset->os_sa->sa_layout_attr_obj)
                dmu_tx_hold_zap(tx, sa->sa_layout_attr_obj, B_TRUE, NULL);
        else {
                dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_LAYOUTS);
                dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_REGISTRY);
                dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
                dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
        }

        dmu_tx_sa_registration_hold(sa, tx);

        if (attrsize <= DN_MAX_BONUSLEN && !sa->sa_force_spill)
                return;

        (void) dmu_tx_hold_object_impl(tx, tx->tx_objset, DMU_NEW_OBJECT,
            THT_SPILL, 0, 0);
}

/*
 * Hold SA attribute
 *
 * dmu_tx_hold_sa(dmu_tx_t *tx, sa_handle_t *, attribute, add, size)
 *
 * variable_size is the total size of all variable sized attributes
 * passed to this function.  It is not the total size of all
 * variable size attributes that *may* exist on this object.
 */
void
dmu_tx_hold_sa(dmu_tx_t *tx, sa_handle_t *hdl, boolean_t may_grow)
{
        uint64_t object;
        sa_os_t *sa = tx->tx_objset->os_sa;

        ASSERT(hdl != NULL);

        object = sa_handle_object(hdl);

        dmu_tx_hold_bonus(tx, object);

        if (tx->tx_objset->os_sa->sa_master_obj == 0)
                return;

        if (tx->tx_objset->os_sa->sa_reg_attr_obj == 0 ||
            tx->tx_objset->os_sa->sa_layout_attr_obj == 0) {
                dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_LAYOUTS);
                dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_REGISTRY);
                dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
                dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
        }
```

```

        dmu_tx_sa_registration_hold(sa, tx);

        if (may_grow && tx->tx_objset->os_sa->sa_layout_attr_obj)
                dmu_tx_hold_zap(tx, sa->sa_layout_attr_obj, B_TRUE, NULL);

        if (sa->sa_force_spill || may_grow || hdl->sa_spill) {
                ASSERT(tx->tx_txg == 0);
                dmu_tx_hold_spill(tx, object);
        } else {
                dmu_buf_impl_t *db = (dmu_buf_impl_t *)hdl->sa_bonus;
                dnode_t *dn;

                DB_DNODE_ENTER(db);
                dn = DB_DNODE(db);
                if (dn->dn_have_spill) {
                        ASSERT(tx->tx_txg == 0);
                        dmu_tx_hold_spill(tx, object);
                }
                DB_DNODE_EXIT(db);
        }
}
```