```
**********************************************************
   47473 Sun Jul 28 21:30:11 2013
new/usr/src/uts/common/fs/zfs/dmu_send.c
3888 zfs recv -F should destroy any snapshots created since the incremental sour
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Peng Dai <peng.dai@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 653 typedef struct dmu_recv_begin_arg {
 654         const char *drba_origin;
 655         dmu_recv_cookie_t *drba_cookie;
 656         cred_t *drba_cred;
 657         uint64_t drba_snapobj;
 658 } dmu_recv_begin_arg_t;

 660 static int
 661 recv_begin_check_existing_impl(dmu_recv_begin_arg_t *drba, dsl_dataset_t *ds,
 662     uint64_t fromguid)
 663 {
 664         uint64_t val;
 665         int error;
 666         dsl_pool_t *dp = ds->ds_dir->dd_pool;

 667         /* must not have any changes since most recent snapshot */
 668         if (!drba->drba_cookie->drc_force &&
 669             dsl_dataset_modified_since_lastsnap(ds))
 670                 return (SET_ERROR(ETXTBSY));

 668         /* temporary clone name must not exist */
 669         error = zap_lookup(dp->dp_meta_objset,
 670             ds->ds_dir->dd_phys->dd_child_dir_zapobj, recv_clone_name,
 671             8, 1, &val);
 672         if (error != ENOENT)
 673                 return (error == 0 ? EBUSY : error);

 675         /* new snapshot name must not exist */
 676         error = zap_lookup(dp->dp_meta_objset,
 677             ds->ds_phys->ds_snapnames_zapobj, drba->drba_cookie->drc_tosnap,
 678             8, 1, &val);
 679         if (error != ENOENT)
 680                 return (error == 0 ? EEXIST : error);

 682         if (fromguid != 0) {
 683                 dsl_dataset_t *snap;
 684                 uint64_t obj = ds->ds_phys->ds_prev_snap_obj;
 687                 /* if incremental, most recent snapshot must match fromguid */
 688                 if (ds->ds_prev == NULL)
 689                         return (SET_ERROR(ENODEV));

 686                 /* Find snapshot in this dir that matches fromguid. */
 691                 /*
 692                  * most recent snapshot must match fromguid, or there are no
 693                  * changes since the fromguid one
 694                  */
 695                 if (ds->ds_prev->ds_phys->ds_guid != fromguid) {
 696                         uint64_t birth = ds->ds_prev->ds_phys->ds_bp.blk_birth;
 697                         uint64_t obj = ds->ds_prev->ds_phys->ds_prev_snap_obj;
 687                 while (obj != 0) {
 699                                 dsl_dataset_t *snap;
 688                         error = dsl_dataset_hold_obj(dp, obj, FTAG,
 689                             &snap);
 690                         if (error != 0)
 691                                 return (SET_ERROR(ENODEV));
 692                         if (snap->ds_dir != ds->ds_dir) {
```



```
 704                                 if (snap->ds_phys->ds_creation_txg < birth) {
 693                                 dsl_dataset_rele(snap, FTAG);
 694                                 return (SET_ERROR(ENODEV));
 695                         }
 696                         if (snap->ds_phys->ds_guid == fromguid)
 697                                 break;
 708                         if (snap->ds_phys->ds_guid == fromguid) {
 709                                 dsl_dataset_rele(snap, FTAG);
 710                                 break; /* it's ok */
 711                         }
 698                         obj = snap->ds_phys->ds_prev_snap_obj;
 699                         dsl_dataset_rele(snap, FTAG);
 700                 }
 701                 if (obj == 0)
 702                         return (SET_ERROR(ENODEV));

 704                 if (drba->drba_cookie->drc_force) {
 705                         drba->drba_snapobj = obj;
 706                 } else {
 707                         /*
 708                          * If we are not forcing, there must be no
 709                          * changes since fromsnap.
 710                          */
 711                         if (dsl_dataset_modified_since_snap(ds, snap)) {
 712                                 dsl_dataset_rele(snap, FTAG);
 713                                 return (SET_ERROR(ETXTBSY));
 714                         }
 715                         drba->drba_snapobj = ds->ds_prev->ds_object;
 716                 }

 718                 dsl_dataset_rele(snap, FTAG);
 719         } else {
 720                 /* if full, most recent snapshot must be $ORIGIN */
 721                 if (ds->ds_phys->ds_prev_snap_txg >= TXG_INITIAL)
 722                         return (SET_ERROR(ENODEV));
 723                 drba->drba_snapobj = ds->ds_phys->ds_prev_snap_obj;
 724         }

 726         return (0);

 728 }
______unchanged_portion_omitted_

 814 static void
 815 dmu_recv_begin_sync(void *arg, dmu_tx_t *tx)
 816 {
 817         dmu_recv_begin_arg_t *drba = arg;
 818         dsl_pool_t *dp = dmu_tx_pool(tx);
 819         struct drr_begin *drrb = drba->drba_cookie->drc_drrb;
 820         const char *tofs = drba->drba_cookie->drc_tofs;
 821         dsl_dataset_t *ds, *newds;
 822         uint64_t dsobj;
 823         int error;
 824         uint64_t crflags;

 826         crflags = (drrb->drr_flags & DRR_FLAG_CI_DATA) ?
 827             DS_FLAG_CI_DATASET : 0;

 829         error = dsl_dataset_hold(dp, tofs, FTAG, &ds);
 830         if (error == 0) {
 831                 /* create temporary clone */
 832                 dsl_dataset_t *snap = NULL;
 833                 if (drba->drba_snapobj != 0) {
 834                         VERIFY0(dsl_dataset_hold_obj(dp,
 835                             drba->drba_snapobj, FTAG, &snap));
 836                 }
```

```
 837                 dsobj = dsl_dataset_create_sync(ds->ds_dir, recv_clone_name,
 838                     snap, crflags, drba->drba_cred, tx);
 839                 dsl_dataset_rele(snap, FTAG);
 831                     ds->ds_prev, crflags, drba->drba_cred, tx);
 840                 dsl_dataset_rele(ds, FTAG);
 841         } else {
 842                 dsl_dir_t *dd;
 843                 const char *tail;
 844                 dsl_dataset_t *origin = NULL;

 846                 VERIFY0(dsl_dir_hold(dp, tofs, FTAG, &dd, &tail));

 848                 if (drba->drba_origin != NULL) {
 849                         VERIFY0(dsl_dataset_hold(dp, drba->drba_origin,
 850                             FTAG, &origin));
 851                 }

 853                 /* Create new dataset. */
 854                 dsobj = dsl_dataset_create_sync(dd,
 855                     strrchr(tofs, '/') + 1,
 856                     origin, crflags, drba->drba_cred, tx);
 857                 if (origin != NULL)
 858                         dsl_dataset_rele(origin, FTAG);
 859                 dsl_dir_rele(dd, FTAG);
 860                 drba->drba_cookie->drc_newfs = B_TRUE;
 861         }
 862         VERIFY0(dsl_dataset_own_obj(dp, dsobj, dmu_recv_tag, &newds));

 864         dmu_buf_will_dirty(newds->ds_dbuf, tx);
 865         newds->ds_phys->ds_flags |= DS_FLAG_INCONSISTENT;

 867         /*
 868          * If we actually created a non-clone, we need to create the
 869          * objset in our new dataset.
 870          */
 871         if (BP_IS_HOLE(dsl_dataset_get_blkptr(newds))) {
 872                 (void) dmu_objset_create_impl(dp->dp_spa,
 873                     newds, dsl_dataset_get_blkptr(newds), drrb->drr_type, tx);
 874         }

 876         drba->drba_cookie->drc_ds = newds;

 878         spa_history_log_internal_ds(newds, "receive", tx, "");
 879 }
```

unchanged_portion_omitted_

```
1542 static int
1543 dmu_recv_end_check(void *arg, dmu_tx_t *tx)
1544 {
1545         dmu_recv_cookie_t *drc = arg;
1546         dsl_pool_t *dp = dmu_tx_pool(tx);
1547         int error;

1549         ASSERT3P(drc->drc_ds->ds_owner, ==, dmu_recv_tag);

1551         if (!drc->drc_newfs) {
1552                 dsl_dataset_t *origin_head;

1554                 error = dsl_dataset_hold(dp, drc->drc_tofs, FTAG, &origin_head);
1555                 if (error != 0)
1556                         return (error);
1557                 if (drc->drc_force) {
1558                         /*
1559                          * We will destroy any snapshots in tofs (i.e. before
1560                          * origin_head) that are after the origin (which is
1561                          * the snap before drc_ds, because drc_ds can not
```



```
1562                          * have any snaps of its own).
1563                          */
1564                         uint64_t obj = origin_head->ds_phys->ds_prev_snap_obj;
1565                         while (obj != drc->drc_ds->ds_phys->ds_prev_snap_obj) {
1566                                 dsl_dataset_t *snap;
1567                                 error = dsl_dataset_hold_obj(dp, obj, FTAG,
1568                                     &snap);
1569                                 if (error != 0)
1570                                         return (error);
1571                                 if (snap->ds_dir != origin_head->ds_dir)
1572                                         error = SET_ERROR(EINVAL);
1573                                 if (error == 0)  {
1574                                         error = dsl_destroy_snapshot_check_impl(
1575                                             snap, B_FALSE);
1576                                 }
1577                                 obj = snap->ds_phys->ds_prev_snap_obj;
1578                                 dsl_dataset_rele(snap, FTAG);
1579                                 if (error != 0)
1580                                         return (error);
1581                         }
1582                 }
1583                 error = dsl_dataset_clone_swap_check_impl(drc->drc_ds,
1584                     origin_head, drc->drc_force, drc->drc_owner, tx);
1585                 if (error != 0) {
1586                         dsl_dataset_rele(origin_head, FTAG);
1587                         return (error);
1588                 }
1589                 error = dsl_dataset_snapshot_check_impl(origin_head,
1590                     drc->drc_tosnap, tx, B_TRUE);
1591                 dsl_dataset_rele(origin_head, FTAG);
1592                 if (error != 0)
1593                         return (error);

1595                 error = dsl_destroy_head_check_impl(drc->drc_ds, 1);
1596         } else {
1597                 error = dsl_dataset_snapshot_check_impl(drc->drc_ds,
1598                     drc->drc_tosnap, tx, B_TRUE);
1599         }
1600         return (error);
1601 }

1603 static void
1604 dmu_recv_end_sync(void *arg, dmu_tx_t *tx)
1605 {
1606         dmu_recv_cookie_t *drc = arg;
1607         dsl_pool_t *dp = dmu_tx_pool(tx);

1609         spa_history_log_internal_ds(drc->drc_ds, "finish receiving",
1610             tx, "snap=%s", drc->drc_tosnap);

1612         if (!drc->drc_newfs) {
1613                 dsl_dataset_t *origin_head;

1615                 VERIFY0(dsl_dataset_hold(dp, drc->drc_tofs, FTAG,
1616                     &origin_head));

1618                 if (drc->drc_force) {
1619                         /*
1620                          * Destroy any snapshots of drc_tofs (origin_head)
1621                          * after the origin (the snap before drc_ds).
1622                          */
1623                         uint64_t obj = origin_head->ds_phys->ds_prev_snap_obj;
1624                         while (obj != drc->drc_ds->ds_phys->ds_prev_snap_obj) {
1625                                 dsl_dataset_t *snap;
1626                                 VERIFY0(dsl_dataset_hold_obj(dp, obj, FTAG,
1627                                     &snap));
```

```
1628                                ASSERT3P(snap->ds_dir, ==, origin_head->ds_dir);
1629                                obj = snap->ds_phys->ds_prev_snap_obj;
1630                                dsl_destroy_snapshot_sync_impl(snap,
1631                                    B_FALSE, tx);
1632                                dsl_dataset_rele(snap, FTAG);
1633                        }
1634                }
1635                VERIFY3P(drc->drc_ds->ds_prev, ==,
1636                    origin_head->ds_prev);

1638                dsl_dataset_clone_swap_sync_impl(drc->drc_ds,
1639                    origin_head, tx);
1640                dsl_dataset_snapshot_sync_impl(origin_head,
1641                    drc->drc_tosnap, tx);

1643                /* set snapshot's creation time and guid */
1644                dmu_buf_will_dirty(origin_head->ds_prev->ds_dbuf, tx);
1645                origin_head->ds_prev->ds_phys->ds_creation_time =
1646                    drc->drc_drrb->drr_creation_time;
1647                origin_head->ds_prev->ds_phys->ds_guid =
1648                    drc->drc_drrb->drr_toguid;
1649                origin_head->ds_prev->ds_phys->ds_flags &=
1650                    ~DS_FLAG_INCONSISTENT;

1652                dmu_buf_will_dirty(origin_head->ds_dbuf, tx);
1653                origin_head->ds_phys->ds_flags &= ~DS_FLAG_INCONSISTENT;

1655                dsl_dataset_rele(origin_head, FTAG);
1656                dsl_destroy_head_sync_impl(drc->drc_ds, tx);

1658                if (drc->drc_owner != NULL)
1659                        VERIFY3P(origin_head->ds_owner, ==, drc->drc_owner);
1660        } else {
1661                dsl_dataset_t *ds = drc->drc_ds;

1663                dsl_dataset_snapshot_sync_impl(ds, drc->drc_tosnap, tx);

1665                /* set snapshot's creation time and guid */
1666                dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
1667                ds->ds_prev->ds_phys->ds_creation_time =
1668                    drc->drc_drrb->drr_creation_time;
1669                ds->ds_prev->ds_phys->ds_guid = drc->drc_drrb->drr_toguid;
1670                ds->ds_prev->ds_phys->ds_flags &= ~DS_FLAG_INCONSISTENT;

1672                dmu_buf_will_dirty(ds->ds_dbuf, tx);
1673                ds->ds_phys->ds_flags &= ~DS_FLAG_INCONSISTENT;
1674        }
1675        drc->drc_newsnapobj = drc->drc_ds->ds_phys->ds_prev_snap_obj;
1676        /*
1677         * Release the hold from dmu_recv_begin.  This must be done before
1678         * we return to open context, so that when we free the dataset's dnode,
1679         * we can evict its bonus buffer.
1680         */
1681        dsl_dataset_disown(drc->drc_ds, dmu_recv_tag);
1682        drc->drc_ds = NULL;
1683 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   82692 Sun Jul 28 21:30:18 2013
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
3888 zfs recv -F should destroy any snapshots created since the incremental sour
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Peng Dai <peng.dai@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1535 boolean_t
1536 dsl_dataset_modified_since_snap(dsl_dataset_t *ds, dsl_dataset_t *snap)
1536 dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds)
1537 {
1538         dsl_pool_t *dp = ds->ds_dir->dd_pool;

1540         ASSERT(dsl_pool_config_held(dp));
1541         if (snap == NULL)
1541         if (ds->ds_prev == NULL)
1542                 return (B_FALSE);
1543         if (ds->ds_phys->ds_bp.blk_birth >
1544             snap->ds_phys->ds_creation_txg) {
1545                 objset_t *os, *os_snap;
1544             ds->ds_prev->ds_phys->ds_creation_txg) {
1545                 objset_t *os, *os_prev;
1546                 /*
1547                  * It may be that only the ZIL differs, because it was
1548                  * reset in the head.  Don't count that as being
1549                  * modified.
1550                  */
1551                 if (dmu_objset_from_ds(ds, &os) != 0)
1552                         return (B_TRUE);
1553                 if (dmu_objset_from_ds(snap, &os_snap) != 0)
1553                 if (dmu_objset_from_ds(ds->ds_prev, &os_prev) != 0)
1554                         return (B_TRUE);
1555                 return (bcmp(&os->os_phys->os_meta_dnode,
1556                     &os_snap->os_phys->os_meta_dnode,
1556                     &os_prev->os_phys->os_meta_dnode,
1557                     sizeof (os->os_phys->os_meta_dnode)) != 0);
1558         }
1559         return (B_FALSE);
1560 }
______unchanged_portion_omitted_

2348 int
2349 dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
2350     dsl_dataset_t *origin_head, boolean_t force, void *owner, dmu_tx_t *tx)
2351 {
2352         int64_t unused_refres_delta;

2354         /* they should both be heads */
2355         if (dsl_dataset_is_snapshot(clone) ||
2356             dsl_dataset_is_snapshot(origin_head))
2357                 return (SET_ERROR(EINVAL));

2359         /* if we are not forcing, the branch point should be just before them */
2360         if (!force && clone->ds_prev != origin_head->ds_prev)
2359         /* the branch point should be just before them */
2360         if (clone->ds_prev != origin_head->ds_prev)
2361                 return (SET_ERROR(EINVAL));

2363         /* clone should be the clone (unless they are unrelated) */
2364         if (clone->ds_prev != NULL &&
2365             clone->ds_prev != clone->ds_dir->dd_pool->dp_origin_snap &&
2366             origin_head->ds_dir != clone->ds_prev->ds_dir)
2366             origin_head->ds_object !=
```



```
2367             clone->ds_prev->ds_phys->ds_next_snap_obj)
2367                 return (SET_ERROR(EINVAL));

2369         /* the clone should be a child of the origin */
2370         if (clone->ds_dir->dd_parent != origin_head->ds_dir)
2371                 return (SET_ERROR(EINVAL));

2373         /* origin_head shouldn't be modified unless 'force' */
2374         if (!force &&
2375             dsl_dataset_modified_since_snap(origin_head, origin_head->ds_prev))
2375         if (!force && dsl_dataset_modified_since_lastsnap(origin_head))
2376                 return (SET_ERROR(ETXTBSY));

2378         /* origin_head should have no long holds (e.g. is not mounted) */
2379         if (dsl_dataset_handoff_check(origin_head, owner, tx))
2380                 return (SET_ERROR(EBUSY));

2382         /* check amount of any unconsumed refreservation */
2383         unused_refres_delta =
2384             (int64_t)MIN(origin_head->ds_reserved,
2385             origin_head->ds_phys->ds_unique_bytes) -
2386             (int64_t)MIN(origin_head->ds_reserved,
2387             clone->ds_phys->ds_unique_bytes);

2389         if (unused_refres_delta > 0 &&
2390             unused_refres_delta >
2391             dsl_dir_space_available(origin_head->ds_dir, NULL, 0, TRUE))
2392                 return (SET_ERROR(ENOSPC));

2394         /* clone can't be over the head's refquota */
2395         if (origin_head->ds_quota != 0 &&
2396             clone->ds_phys->ds_referenced_bytes > origin_head->ds_quota)
2397                 return (SET_ERROR(EDQUOT));

2399         return (0);
2400 }

2402 void
2403 dsl_dataset_clone_swap_sync_impl(dsl_dataset_t *clone,
2404     dsl_dataset_t *origin_head, dmu_tx_t *tx)
2405 {
2406         dsl_pool_t *dp = dmu_tx_pool(tx);
2407         int64_t unused_refres_delta;

2409         ASSERT(clone->ds_reserved == 0);
2410         ASSERT(origin_head->ds_quota == 0 ||
2411             clone->ds_phys->ds_unique_bytes <= origin_head->ds_quota);
2412         ASSERT3P(clone->ds_prev, ==, origin_head->ds_prev);

2414         dmu_buf_will_dirty(clone->ds_dbuf, tx);
2415         dmu_buf_will_dirty(origin_head->ds_dbuf, tx);

2417         if (clone->ds_objset != NULL) {
2418                 dmu_objset_evict(clone->ds_objset);
2419                 clone->ds_objset = NULL;
2420         }

2422         if (origin_head->ds_objset != NULL) {
2423                 dmu_objset_evict(origin_head->ds_objset);
2424                 origin_head->ds_objset = NULL;
2425         }

2427         unused_refres_delta =
2428             (int64_t)MIN(origin_head->ds_reserved,
2429             origin_head->ds_phys->ds_unique_bytes) -
2430             (int64_t)MIN(origin_head->ds_reserved,
```

```
2431             clone->ds_phys->ds_unique_bytes);

2433         /*
2434          * Reset origin's unique bytes, if it exists.
2435          */
2436         if (clone->ds_prev) {
2437                 dsl_dataset_t *origin = clone->ds_prev;
2438                 uint64_t comp, uncomp;

2440                 dmu_buf_will_dirty(origin->ds_dbuf, tx);
2441                 dsl_deadlist_space_range(&clone->ds_deadlist,
2442                     origin->ds_phys->ds_prev_snap_txg, UINT64_MAX,
2443                     &origin->ds_phys->ds_unique_bytes, &comp, &uncomp);
2444         }

2446         /* swap blkptrs */
2447         {
2448                 blkptr_t tmp;
2449                 tmp = origin_head->ds_phys->ds_bp;
2450                 origin_head->ds_phys->ds_bp = clone->ds_phys->ds_bp;
2451                 clone->ds_phys->ds_bp = tmp;
2452         }

2454         /* set dd_*_bytes */
2455         {
2456                 int64_t dused, dcomp, duncomp;
2457                 uint64_t cdl_used, cdl_comp, cdl_uncomp;
2458                 uint64_t odl_used, odl_comp, odl_uncomp;

2460                 ASSERT3U(clone->ds_dir->dd_phys->
2461                     dd_used_breakdown[DD_USED_SNAP], ==, 0);

2463                 dsl_deadlist_space(&clone->ds_deadlist,
2464                     &cdl_used, &cdl_comp, &cdl_uncomp);
2465                 dsl_deadlist_space(&origin_head->ds_deadlist,
2466                     &odl_used, &odl_comp, &odl_uncomp);

2468                 dused = clone->ds_phys->ds_referenced_bytes + cdl_used -
2469                     (origin_head->ds_phys->ds_referenced_bytes + odl_used);
2470                 dcomp = clone->ds_phys->ds_compressed_bytes + cdl_comp -
2471                     (origin_head->ds_phys->ds_compressed_bytes + odl_comp);
2472                 duncomp = clone->ds_phys->ds_uncompressed_bytes +
2473                     cdl_uncomp -
2474                     (origin_head->ds_phys->ds_uncompressed_bytes + odl_uncomp);

2476                 dsl_dir_diduse_space(origin_head->ds_dir, DD_USED_HEAD,
2477                     dused, dcomp, duncomp, tx);
2478                 dsl_dir_diduse_space(clone->ds_dir, DD_USED_HEAD,
2479                     -dused, -dcomp, -duncomp, tx);

2481                 /*
2482                  * The difference in the space used by snapshots is the
2483                  * difference in snapshot space due to the head's
2484                  * deadlist (since that's the only thing that's
2485                  * changing that affects the snapused).
2486                  */
2487                 dsl_deadlist_space_range(&clone->ds_deadlist,
2488                     origin_head->ds_dir->dd_origin_txg, UINT64_MAX,
2489                     &cdl_used, &cdl_comp, &cdl_uncomp);
2490                 dsl_deadlist_space_range(&origin_head->ds_deadlist,
2491                     origin_head->ds_dir->dd_origin_txg, UINT64_MAX,
2492                     &odl_used, &odl_comp, &odl_uncomp);
2493                 dsl_dir_transfer_space(origin_head->ds_dir, cdl_used - odl_used,
2494                     DD_USED_HEAD, DD_USED_SNAP, tx);
2495         }
```



```
2497         /* swap ds_*_bytes */
2498         SWITCH64(origin_head->ds_phys->ds_referenced_bytes,
2499             clone->ds_phys->ds_referenced_bytes);
2500         SWITCH64(origin_head->ds_phys->ds_compressed_bytes,
2501             clone->ds_phys->ds_compressed_bytes);
2502         SWITCH64(origin_head->ds_phys->ds_uncompressed_bytes,
2503             clone->ds_phys->ds_uncompressed_bytes);
2504         SWITCH64(origin_head->ds_phys->ds_unique_bytes,
2505             clone->ds_phys->ds_unique_bytes);

2507         /* apply any parent delta for change in unconsumed refreservation */
2508         dsl_dir_diduse_space(origin_head->ds_dir, DD_USED_REFRSRV,
2509             unused_refres_delta, 0, 0, tx);

2511         /*
2512          * Swap deadlists.
2513          */
2514         dsl_deadlist_close(&clone->ds_deadlist);
2515         dsl_deadlist_close(&origin_head->ds_deadlist);
2516         SWITCH64(origin_head->ds_phys->ds_deadlist_obj,
2517             clone->ds_phys->ds_deadlist_obj);
2518         dsl_deadlist_open(&clone->ds_deadlist, dp->dp_meta_objset,
2519             clone->ds_phys->ds_deadlist_obj);
2520         dsl_deadlist_open(&origin_head->ds_deadlist, dp->dp_meta_objset,
2521             origin_head->ds_phys->ds_deadlist_obj);

2523         dsl_scan_ds_clone_swapped(origin_head, clone, tx);

2525         spa_history_log_internal_ds(clone, "clone swap", tx,
2526             "parent=%s", origin_head->ds_dir->dd_myname);
2527 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   25694 Sun Jul 28 21:30:21 2013
new/usr/src/uts/common/fs/zfs/dsl_destroy.c
3888 zfs recv -F should destroy any snapshots created since the incremental sour
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Peng Dai <peng.dai@delphix.com>
**********************************************************
_____unchanged_portion_omitted_

  49 int
  49 /*
  50  * ds must be owned.
  51  */
  52 static int
  50 dsl_destroy_snapshot_check_impl(dsl_dataset_t *ds, boolean_t defer)
  51 {
  52         if (!dsl_dataset_is_snapshot(ds))
  53                 return (SET_ERROR(EINVAL));

  55         if (dsl_dataset_long_held(ds))
  56                 return (SET_ERROR(EBUSY));

  58         /*
  59          * Only allow deferred destroy on pools that support it.
  60          * NOTE: deferred destroy is only supported on snapshots.
  61          */
  62         if (defer) {
  63                 if (spa_version(ds->ds_dir->dd_pool->dp_spa) <
  64                     SPA_VERSION_USERREFS)
  65                         return (SET_ERROR(ENOTSUP));
  66                 return (0);
  67         }

  69         /*
  70          * If this snapshot has an elevated user reference count,
  71          * we can't destroy it yet.
  72          */
  73         if (ds->ds_userrefs > 0)
  74                 return (SET_ERROR(EBUSY));

  76         /*
  77          * Can't delete a branch point.
  78          */
  79         if (ds->ds_phys->ds_num_children > 1)
  80                 return (SET_ERROR(EEXIST));

  82         return (0);
  83 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   10284 Sun Jul 28 21:30:24 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
3888 zfs recv -F should destroy any snapshots created since the incremental sour
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Peng Dai <peng.dai@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  * Copyright (c) 2012, Joyent, Inc. All rights reserved.
  25  * Copyright (c) 2013 Steven Hartland. All rights reserved.
  26  */

  28 #ifndef _SYS_DSL_DATASET_H
  29 #define _SYS_DSL_DATASET_H

  31 #include <sys/dmu.h>
  32 #include <sys/spa.h>
  33 #include <sys/txg.h>
  34 #include <sys/zio.h>
  35 #include <sys/bplist.h>
  36 #include <sys/dsl_synctask.h>
  37 #include <sys/zfs_context.h>
  38 #include <sys/dsl_deadlist.h>
  39 #include <sys/refcount.h>

  41 #ifdef  __cplusplus
  42 extern "C" {
  43 #endif

  45 struct dsl_dataset;
  46 struct dsl_dir;
  47 struct dsl_pool;

  49 #define DS_FLAG_INCONSISTENT    (1ULL<<0)
  50 #define DS_IS_INCONSISTENT(ds)  \
  51         ((ds)->ds_phys->ds_flags & DS_FLAG_INCONSISTENT)
  52 /*
  53  * Note: nopromote can not yet be set, but we want support for it in this
  54  * on-disk version, so that we don't need to upgrade for it later.
  55  */
  56 #define DS_FLAG_NOPROMOTE       (1ULL<<1)
```



```
  58 /*
  59  * DS_FLAG_UNIQUE_ACCURATE is set if ds_unique_bytes has been correctly
  60  * calculated for head datasets (starting with SPA_VERSION_UNIQUE_ACCURATE,
  61  * refquota/refreservations).
  62  */
  63 #define DS_FLAG_UNIQUE_ACCURATE (1ULL<<2)

  65 /*
  66  * DS_FLAG_DEFER_DESTROY is set after 'zfs destroy -d' has been called
  67  * on a dataset. This allows the dataset to be destroyed using 'zfs release'.
  68  */
  69 #define DS_FLAG_DEFER_DESTROY   (1ULL<<3)
  70 #define DS_IS_DEFER_DESTROY(ds) \
  71         ((ds)->ds_phys->ds_flags & DS_FLAG_DEFER_DESTROY)

  73 /*
  74  * DS_FLAG_CI_DATASET is set if the dataset contains a file system whose
  75  * name lookups should be performed case-insensitively.
  76  */
  77 #define DS_FLAG_CI_DATASET      (1ULL<<16)

  79 #define DS_CREATE_FLAG_NODIRTY  (1ULL<<24)

  81 typedef struct dsl_dataset_phys {
  82         uint64_t ds_dir_obj;            /* DMU_OT_DSL_DIR */
  83         uint64_t ds_prev_snap_obj;      /* DMU_OT_DSL_DATASET */
  84         uint64_t ds_prev_snap_txg;
  85         uint64_t ds_next_snap_obj;      /* DMU_OT_DSL_DATASET */
  86         uint64_t ds_snapnames_zapobj;   /* DMU_OT_DSL_DS_SNAP_MAP 0 for snaps */
  87         uint64_t ds_num_children;       /* clone/snap children; ==0 for head */
  88         uint64_t ds_creation_time;      /* seconds since 1970 */
  89         uint64_t ds_creation_txg;
  90         uint64_t ds_deadlist_obj;       /* DMU_OT_DEADLIST */
  91         /*
  92          * ds_referenced_bytes, ds_compressed_bytes, and ds_uncompressed_bytes
  93          * include all blocks referenced by this dataset, including those
  94          * shared with any other datasets.
  95          */
  96         uint64_t ds_referenced_bytes;
  97         uint64_t ds_compressed_bytes;
  98         uint64_t ds_uncompressed_bytes;
  99         uint64_t ds_unique_bytes;       /* only relevant to snapshots */
 100         /*
 101          * The ds_fsid_guid is a 56-bit ID that can change to avoid
 102          * collisions.  The ds_guid is a 64-bit ID that will never
 103          * change, so there is a small probability that it will collide.
 104          */
 105         uint64_t ds_fsid_guid;
 106         uint64_t ds_guid;
 107         uint64_t ds_flags;              /* DS_FLAG_* */
 108         blkptr_t ds_bp;
 109         uint64_t ds_next_clones_obj;    /* DMU_OT_DSL_CLONES */
 110         uint64_t ds_props_obj;          /* DMU_OT_DSL_PROPS for snaps */
 111         uint64_t ds_userrefs_obj;       /* DMU_OT_USERREFS */
 112         uint64_t ds_pad[5]; /* pad out to 320 bytes for good measure */
 113 } dsl_dataset_phys_t;
_____unchanged_portion_omitted_

 167 /*
 168  * The max length of a temporary tag prefix is the number of hex digits
 169  * required to express UINT64_MAX plus one for the hyphen.
 170  */
 171 #define MAX_TAG_PREFIX_LEN      17

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

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

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

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

spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);

boolean_t dsl_dataset_modified_since_snap(dsl_dataset_t *ds,
    dsl_dataset_t *snap);
boolean_t dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds);

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

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

void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);
void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
void dsl_dataset_space(dsl_dataset_t *ds,
    uint64_t *refdbytesp, uint64_t *availbytesp,
    uint64_t *usedobjsp, uint64_t *availobjsp);
uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
    uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
    uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
boolean_t dsl_dataset_is_dirty(dsl_dataset_t *ds);

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

int dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
    uint64_t asize, uint64_t inflight, uint64_t *used,
    uint64_t *ref_rsrv);
```



```
int dsl_dataset_set_refquota(const char *dsname, zprop_source_t source,
    uint64_t quota);
int dsl_dataset_set_refreservation(const char *dsname, zprop_source_t source,
    uint64_t reservation);

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

int dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
    dsl_dataset_t *origin_head, boolean_t force, void *owner, dmu_tx_t *tx);
void dsl_dataset_clone_swap_sync_impl(dsl_dataset_t *clone,
    dsl_dataset_t *origin_head, dmu_tx_t *tx);
int dsl_dataset_snapshot_check_impl(dsl_dataset_t *ds, const char *snapname,
    dmu_tx_t *tx, boolean_t recv);
void dsl_dataset_snapshot_sync_impl(dsl_dataset_t *ds, const char *snapname,
    dmu_tx_t *tx);

void dsl_dataset_remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj,
    dmu_tx_t *tx);
void dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds);
int dsl_dataset_get_snapname(dsl_dataset_t *ds);
int dsl_dataset_snap_lookup(dsl_dataset_t *ds, const char *name,
    uint64_t *value);
int dsl_dataset_snap_remove(dsl_dataset_t *ds, const char *name, dmu_tx_t *tx);
void dsl_dataset_set_refreservation_sync_impl(dsl_dataset_t *ds,
    zprop_source_t source, uint64_t value, dmu_tx_t *tx);
int dsl_dataset_rollback(const char *fsname, void *owner);

#ifdef ZFS_DEBUG
#define dprintf_ds(ds, fmt, ...) do { \
        if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
        char *__ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
        dsl_dataset_name(ds, __ds_name); \
        dprintf("ds=%s " fmt, __ds_name, __VA_ARGS__); \
        kmem_free(__ds_name, MAXNAMELEN); \
        } \
_NOTE(CONSTCOND) } while (0)
#else
#define dprintf_ds(dd, fmt, ...)
#endif

#ifdef  __cplusplus
}
```

*unchanged_portion_omitted_*

****
1740 Sun Jul 28 21:30:28 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_destroy.h
3888 zfs recv -F should destroy any snapshots created since the incremental sour
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Peng Dai <peng.dai@delphix.com>
****

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */

#ifndef _SYS_DSL_DESTROY_H
#define _SYS_DSL_DESTROY_H

#ifdef  __cplusplus
extern "C" {
#endif

struct nvlist;
struct dsl_dataset;
struct dmu_tx;

int dsl_destroy_snapshots_nvl(struct nvlist *, boolean_t,
    struct nvlist *);
int dsl_destroy_snapshot(const char *, boolean_t);
int dsl_destroy_head(const char *);
int dsl_destroy_head_check_impl(struct dsl_dataset *, int);
void dsl_destroy_head_sync_impl(struct dsl_dataset *, struct dmu_tx *);
int dsl_destroy_inconsistent(const char *, void *);
int dsl_destroy_snapshot_check_impl(struct dsl_dataset *, boolean_t);
void dsl_destroy_snapshot_sync_impl(struct dsl_dataset *,
    boolean_t, struct dmu_tx *);
int dsl_destroy_snapshots_nvl(struct nvlist *snaps, boolean_t defer,
    struct nvlist *errlist);
int dsl_destroy_snapshot(const char *name, boolean_t defer);
int dsl_destroy_head(const char *name);
int dsl_destroy_head_check_impl(struct dsl_dataset *ds, int expected_holds);
void dsl_destroy_head_sync_impl(struct dsl_dataset *ds, struct dmu_tx *tx);
int dsl_destroy_inconsistent(const char *dsname, void *arg);
void dsl_destroy_snapshot_sync_impl(struct dsl_dataset *ds,
    boolean_t defer, struct dmu_tx *tx);
```



```
#ifdef  __cplusplus
}
```

_____unchanged_portion_omitted_