

```
*****
14226 Tue Apr 23 15:47:36 2013
new/usr/src/uts/common/fs/zfs/dsl_userhold.c
3744 zfs shouldn't ignore errors unmounting snapshots
Submitted by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
*****
_unchanged_portion_omitted_
```

```
416 /*
417  * Called at spa_load time to release a stale temporary user hold.
418  * Also called by the onexit code.
419  */
420 void
421 dsl_dataset_user_release_tmp(dsl_pool_t *dp, uint64_t dsobj, const char *htag)
422 {
423     dsl_dataset_user_release_tmp_arg_t ddurta;
424     dsl_dataset_t *ds;
425     int error;
426
427 #ifdef _KERNEL
428     /* Make sure it is not mounted. */
429     dsl_pool_config_enter(dp, FTAG);
430     error = dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds);
431     if (error == 0) {
432         char name[MAXNAMELEN];
433         dsl_dataset_name(ds, name);
434         dsl_dataset_rele(ds, FTAG);
435         dsl_pool_config_exit(dp, FTAG);
436         (void) zfs_unmount_snap(name);
437     } else {
438         dsl_pool_config_exit(dp, FTAG);
439     }
440 #endif
441
442     ddurta.ddurta_dsobj = dsobj;
443     ddurta.ddurta_holds = fnvlist_alloc();
444     fnvlist_add_boolean(ddurta.ddurta_holds, htag);
445
446     (void) dsl_sync_task(spa_name(dp->dp_spa),
447         dsl_dataset_user_release_tmp_check,
448         dsl_dataset_user_release_tmp_sync, &ddurta, 1);
449     fnvlist_free(ddurta.ddurta_holds);
450 }
_unchanged_portion_omitted_
```

new/usr/src/uts/common/fs/zfs/sys/zfs_ioctl.h

1

10076 Tue Apr 23 15:47:36 2013

new/usr/src/uts/common/fs/zfs/sys/zfs_ioctl.h

3744 zfs shouldn't ignore errors unmounting snapshots

Submitted by: Will Andrews <willa@spectralogic.com>

Reviewed by: Matthew Ahrens <mahrens@delphix.com>

unchanged portion omitted

340 extern dev_info_t *zfs_dip;

342 extern int zfs_secpolicy_snapshot_perms(const char *name, cred_t *cr);

343 extern int zfs_secpolicy_rename_perms(const char *from,

344 const char *to, cred_t *cr);

345 extern int zfs_secpolicy_destroy_perms(const char *name, cred_t *cr);

346 extern int zfs_busy(void);

347 extern int zfs_unmount_snap(const char *);

348 extern void zfs_unmount_snap(const char *);

348 extern void zfs_destroy_unmount_origin(const char *);

350 /*

351 * ZFS minor numbers can refer to either a control device instance or

352 * a zvol. Depending on the value of zss_type, zss_data points to either

353 * a zvol_state_t or a zfs_onexit_t.

354 */

355 enum zfs_soft_state_type {

356 ZSST_ZVOL,

357 ZSST_CTLDEV

358 };

unchanged portion omitted

```

*****
144766 Tue Apr 23 15:47:36 2013
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
3744 zfs shouldn't ignore errors unmounting snapshots
Submitted by: Will Andrews <willa@spectralogic.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
*****
_____unchanged_portion_omitted_____

3365 /*
3366  * The dp_config_rwlock must not be held when calling this, because the
3367  * unmount may need to write out data.
3368  *
3369  * This function is best-effort. Callers must deal gracefully if it
3370  * remains mounted (or is remounted after this call).
3371  *
3372  * Returns 0 if the argument is not a snapshot, or it is not currently a
3373  * filesystem, or we were able to unmount it. Returns error code otherwise.
3374  #endif /* !codereview */
3375  */
3376  int
3377  zfs_unmount_snap(const char *snapname)
3378  {
3379      vfs_t *vfsp;
3380      zfsvfs_t *zfsvfs;
3381      int err;
3382  #endif /* !codereview */

3384      if (strchr(snapname, '@') == NULL)
3385          return (0);
3386      return;

3387      vfsp = zfs_get_vfs(snapname);
3388      if (vfsp == NULL)
3389          return (0);
3390      return;

3391      zfsvfs = vfsp->vfs_data;
3392      ASSERT(!dsl_pool_config_held(dmu_objset_pool(zfsvfs->z_os)));

3394      err = vn_vfswlock(vfsp->vfs_vnodecovered);
3385      if (vn_vfswlock(vfsp->vfs_vnodecovered) != 0) {
3386          VFS_RELE(vfsp);
3387          return;
3388      }
3395      VFS_RELE(vfsp);
3396      if (err != 0)
3397          return (err);
3398  #endif /* !codereview */

3400      /*
3401      * Always force the unmount for snapshots.
3402      */
3403      (void) dounmount(vfsp, MS_FORCE, kcred);
3404      return (0);
3405  #endif /* !codereview */
3406  }

3408 /* ARGSUSED */
3409 static int
3410 zfs_unmount_snap_cb(const char *snapname, void *arg)
3411 {
3412     return (zfs_unmount_snap(snapname));
3413 }
3414 zfs_unmount_snap(snapname);
3415 return (0);

```

```

3413 }

3415 /*
3416  * When a clone is destroyed, its origin may also need to be destroyed,
3417  * in which case it must be unmounted. This routine will do that unmount
3418  * if necessary.
3419  */
3420 void
3421 zfs_destroy_unmount_origin(const char *fsname)
3422 {
3423     int error;
3424     objset_t *os;
3425     dsl_dataset_t *ds;

3427     error = dmu_objset_hold(fsname, FTAG, &os);
3428     if (error != 0)
3429         return;
3430     ds = dmu_objset_ds(os);
3431     if (dsl_dir_is_clone(ds->ds_dir) && DS_IS_DEFER_DESTROY(ds->ds_prev)) {
3432         char originname[MAXNAMELEN];
3433         dsl_dataset_name(ds->ds_prev, originname);
3434         dmu_objset_rele(os, FTAG);
3435         (void) zfs_unmount_snap(originname);
3436         zfs_unmount_snap(originname);
3437     } else {
3438         dmu_objset_rele(os, FTAG);
3439     }

3441 /*
3442  * innvl: {
3443  *     "snaps" -> { snapshot1, snapshot2 }
3444  *     (optional boolean) "defer"
3445  * }
3446  *
3447  * outnvl: snapshot -> error code (int32)
3448  *
3449  */
3450 static int
3451 zfs_ioc_destroy_snaps(const char *poolname, nvlist_t *innvl, nvlist_t *outnvl)
3452 {
3453     int error, poolen;
3454     int poolen;
3455     nvlist_t *snaps;
3456     nvpair_t *pair;
3457     boolean_t defer;

3458     if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
3459         return (SET_ERROR(EINVAL));
3460     defer = nvlist_exists(innvl, "defer");

3462     poolen = strlen(poolname);
3463     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
3464          pair = nvlist_next_nvpair(snaps, pair)) {
3465         const char *name = nvpair_name(pair);

3467         /*
3468          * The snap must be in the specified pool.
3469          */
3470         if (strcmp(name, poolname, poolen) != 0 ||
3471             (name[poolen] != '/' && name[poolen] != '@'))
3472             return (SET_ERROR(EXDEV));

3474         error = zfs_unmount_snap(name);
3475         if (error)
3476             return (SET_ERROR(error));

```

```

3453         zfs_unmount_snap(name);
3477     }

3479     return (dsl_destroy_snapshots_nvl(snaps, defer, outnvl));
3480 }

3482 /*
3483  * inputs:
3484  * zc_name           name of dataset to destroy
3485  * zc_objset_type   type of objset
3486  * zc_defer_destroy mark for deferred destroy
3487  * outputs:         none
3488  */
3490 static int
3491 zfs_ioc_destroy(zfs_cmd_t *zc)
3492 {
3493     int err;

3495     if (zc->zc_objset_type == DMU_OST_ZFS) {
3496         err = zfs_unmount_snap(zc->zc_name);
3497         if (err)
3498             return (err);
3499     }
3500     if (strchr(zc->zc_name, '@') && zc->zc_objset_type == DMU_OST_ZFS)
3501         zfs_unmount_snap(zc->zc_name);

3501     if (strchr(zc->zc_name, '@'))
3502         err = dsl_destroy_snapshot(zc->zc_name, zc->zc_defer_destroy);
3503     else
3504         err = dsl_destroy_head(zc->zc_name);
3505     if (zc->zc_objset_type == DMU_OST_ZVOL && err == 0)
3506         (void) zvol_remove_minor(zc->zc_name);
3507     return (err);
3508 }
unchanged_portion_omitted

3538 static int
3539 recursive_unmount(const char *fsname, void *arg)
3540 {
3541     const char *snapname = arg;
3542     char fullname[MAXNAMELEN];

3544     (void) snprintf(fullname, sizeof (fullname), "%s@%s", fsname, snapname);
3545     return (zfs_unmount_snap(fullname));
3546     zfs_unmount_snap(fullname);
3547     return (0);
3548 }
unchanged_portion_omitted

4988 /*
4989  * innvl: {
4990  *     snapname -> { holdname, ... }
4991  *     ...
4992  * }
4993  *
4994  * outnvl: {
4995  *     snapname -> error value (int32)
4996  *     ...
4997  * }
4998  */
4999 /* ARGSUSED */
5000 static int
5001 zfs_ioc_release(const char *pool, nvlist_t *holds, nvlist_t *errlist)
5002 {
5003     nvpair_t *pair;

```

```

5004     int err;
5005 #endif /* ! codereview */

5007     /*
5008      * The release may cause the snapshot to be destroyed; make sure it
5009      * is not mounted.
5010      */
5011     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
5012          pair = nvlist_next_nvpair(holds, pair)) {
5013         err = zfs_unmount_snap(nvpair_name(pair));
5014         if (err)
5015             return (err);
5016     }
5017     pair = nvlist_next_nvpair(holds, pair);
5018     zfs_unmount_snap(nvpair_name(pair));

5018     return (dsl_dataset_user_release(holds, errlist));
5019 }
unchanged_portion_omitted

```