```
**********************************************************
   35734 Tue Apr 23 10:00:49 2013
new/usr/src/uts/common/fs/zfs/dsl_dir.c
3739 cannot set zfs quota or reservation on pool version < 22
**********************************************************
```

```
/*
 * 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) 2013 Martin Matuska. All rights reserved.
#endif /* ! codereview */
 */

#include <sys/dmu.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/dsl_deleg.h>
#include <sys/spa.h>
#include <sys/metaslab.h>
#include <sys/zap.h>
#include <sys/zio.h>
#include <sys/arc.h>
#include <sys/sunddi.h>
#include "zfs_namecheck.h"

static uint64_t dsl_dir_space_towrite(dsl_dir_t *dd);

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

        for (t = 0; t < TXG_SIZE; t++) {
                ASSERT(!txg_list_member(&dp->dp_dirty_dirs, dd, t));
                ASSERT(dd->dd_tempreserved[t] == 0);
                ASSERT(dd->dd_space_towrite[t] == 0);
        }

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



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

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

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

        ASSERT(dsl_pool_config_held(dp));

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

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

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

                dsl_dir_snap_cmtime_update(dd);

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

                                err = zap_lookup(dp->dp_meta_objset,
                                    dd->dd_parent->dd_phys->dd_child_dir_zapobj,
                                    tail, sizeof (foundobj), 1, &foundobj);
                                ASSERT(err || foundobj == ddobj);
#endif
                                (void) strcpy(dd->dd_myname, tail);
                        } else {
                                err = zap_value_search(dp->dp_meta_objset,
```

```
128                                     dd->dd_parent->dd_phys->dd_child_dir_zapobj,
129                                     ddobj, 0, dd->dd_myname);
130                             }
131                             if (err != 0)
132                                     goto errout;
133                     } else {
134                             (void) strcpy(dd->dd_myname, spa_name(dp->dp_spa));
135                     }

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

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

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

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

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



```
194 void
195 dsl_dir_rele(dsl_dir_t *dd, void *tag)
196 {
197         dprintf_dd(dd, "%s\n", "");
198         spa_close(dd->dd_pool->dp_spa, tag);
199         dmu_buf_rele(dd->dd_dbuf, tag);
200 }

202 /* buf must be long enough (MAXNAMELEN + strlen(MOS_DIR_NAME) + 1 should do) */
203 void
204 dsl_dir_name(dsl_dir_t *dd, char *buf)
205 {
206         if (dd->dd_parent) {
207                 dsl_dir_name(dd->dd_parent, buf);
208                 (void) strcat(buf, "/");
209         } else {
210                 buf[0] = '\0';
211         }
212         if (!MUTEX_HELD(&dd->dd_lock)) {
213                 /*
214                  * recursive mutex so that we can use
215                  * dprintf_dd() with dd_lock held
216                  */
217                 mutex_enter(&dd->dd_lock);
218                 (void) strcat(buf, dd->dd_myname);
219                 mutex_exit(&dd->dd_lock);
220         } else {
221                 (void) strcat(buf, dd->dd_myname);
222         }
223 }

225 /* Calculate name length, avoiding all the strcat calls of dsl_dir_name */
226 int
227 dsl_dir_namelen(dsl_dir_t *dd)
228 {
229         int result = 0;

231         if (dd->dd_parent) {
232                 /* parent's name + 1 for the "/" */
233                 result = dsl_dir_namelen(dd->dd_parent) + 1;
234         }

236         if (!MUTEX_HELD(&dd->dd_lock)) {
237                 /* see dsl_dir_name */
238                 mutex_enter(&dd->dd_lock);
239                 result += strlen(dd->dd_myname);
240                 mutex_exit(&dd->dd_lock);
241         } else {
242                 result += strlen(dd->dd_myname);
243         }

245         return (result);
246 }

248 static int
249 getcomponent(const char *path, char *component, const char **nextp)
250 {
251         char *p;

253         if ((path == NULL) || (path[0] == '\0'))
254                 return (SET_ERROR(ENOENT));
255         /* This would be a good place to reserve some namespace... */
256         p = strpbrk(path, "/@");
257         if (p && (p[1] == '/' || p[1] == '@')) {
258                 /* two separators in a row */
259                 return (SET_ERROR(EINVAL));
```

```
260         }
261         if (p == NULL || p == path) {
262                 /*
263                  * if the first thing is an @ or /, it had better be an
264                  * @ and it had better not have any more ats or slashes,
265                  * and it had better have something after the @.
266                  */
267                 if (p != NULL &&
268                     (p[0] != '@' || strpbrk(path+1, "/@") || p[1] == '\0'))
269                         return (SET_ERROR(EINVAL));
270                 if (strlen(path) >= MAXNAMELEN)
271                         return (SET_ERROR(ENAMETOOLONG));
272                 (void) strcpy(component, path);
273                 p = NULL;
274         } else if (p[0] == '/') {
275                 if (p - path >= MAXNAMELEN)
276                         return (SET_ERROR(ENAMETOOLONG));
277                 (void) strncpy(component, path, p - path);
278                 component[p - path] = '\0';
279                 p++;
280         } else if (p[0] == '@') {
281                 /*
282                  * if the next separator is an @, there better not be
283                  * any more slashes.
284                  */
285                 if (strchr(path, '/'))
286                         return (SET_ERROR(EINVAL));
287                 if (p - path >= MAXNAMELEN)
288                         return (SET_ERROR(ENAMETOOLONG));
289                 (void) strncpy(component, path, p - path);
290                 component[p - path] = '\0';
291         } else {
292                 panic("invalid p=%p", (void *)p);
293         }
294         *nextp = p;
295         return (0);
296 }

298 /*
299  * Return the dsl_dir_t, and possibly the last component which couldn't
300  * be found in *tail.  The name must be in the specified dsl_pool_t.  This
301  * thread must hold the dp_config_rwlock for the pool.  Returns NULL if the
302  * path is bogus, or if tail==NULL and we couldn't parse the whole name.
303  * (*tail)[0] == '@' means that the last component is a snapshot.
304  */
305 int
306 dsl_dir_hold(dsl_pool_t *dp, const char *name, void *tag,
307     dsl_dir_t **ddp, const char **tailp)
308 {
309         char buf[MAXNAMELEN];
310         const char *spaname, *next, *nextnext = NULL;
311         int err;
312         dsl_dir_t *dd;
313         uint64_t ddobj;

315         err = getcomponent(name, buf, &next);
316         if (err != 0)
317                 return (err);

319         /* Make sure the name is in the specified pool. */
320         spaname = spa_name(dp->dp_spa);
321         if (strcmp(buf, spaname) != 0)
322                 return (SET_ERROR(EINVAL));

324         ASSERT(dsl_pool_config_held(dp));
```



```
326         err = dsl_dir_hold_obj(dp, dp->dp_root_dir_obj, NULL, tag, &dd);
327         if (err != 0) {
328                 return (err);
329         }

331         while (next != NULL) {
332                 dsl_dir_t *child_ds;
333                 err = getcomponent(next, buf, &nextnext);
334                 if (err != 0)
335                         break;
336                 ASSERT(next[0] != '\0');
337                 if (next[0] == '@')
338                         break;
339                 dprintf("looking up %s in obj%lld\n",
340                     buf, dd->dd_phys->dd_child_dir_zapobj);

342                 err = zap_lookup(dp->dp_meta_objset,
343                     dd->dd_phys->dd_child_dir_zapobj,
344                     buf, sizeof (ddobj), 1, &ddobj);
345                 if (err != 0) {
346                         if (err == ENOENT)
347                                 err = 0;
348                         break;
349                 }

351                 err = dsl_dir_hold_obj(dp, ddobj, buf, tag, &child_ds);
352                 if (err != 0)
353                         break;
354                 dsl_dir_rele(dd, tag);
355                 dd = child_ds;
356                 next = nextnext;
357         }

359         if (err != 0) {
360                 dsl_dir_rele(dd, tag);
361                 return (err);
362         }

364         /*
365          * It's an error if there's more than one component left, or
366          * tailp==NULL and there's any component left.
367          */
368         if (next != NULL &&
369             (tailp == NULL || (nextnext && nextnext[0] != '\0'))) {
370                 /* bad path name */
371                 dsl_dir_rele(dd, tag);
372                 dprintf("next=%p (%s) tail=%p\n", next, next?next:"", tailp);
373                 err = SET_ERROR(ENOENT);
374         }
375         if (tailp != NULL)
376                 *tailp = next;
377         *ddp = dd;
378         return (err);
379 }

381 uint64_t
382 dsl_dir_create_sync(dsl_pool_t *dp, dsl_dir_t *pds, const char *name,
383     dmu_tx_t *tx)
384 {
385         objset_t *mos = dp->dp_meta_objset;
386         uint64_t ddobj;
387         dsl_dir_phys_t *ddphys;
388         dmu_buf_t *dbuf;

390         ddobj = dmu_object_alloc(mos, DMU_OT_DSL_DIR, 0,
391             DMU_OT_DSL_DIR, sizeof (dsl_dir_phys_t), tx);
```

```
392         if (pds) {
393                 VERIFY(0 == zap_add(mos, pds->dd_phys->dd_child_dir_zapobj,
394                     name, sizeof (uint64_t), 1, &ddobj, tx));
395         } else {
396                 /* it's the root dir */
397                 VERIFY(0 == zap_add(mos, DMU_POOL_DIRECTORY_OBJECT,
398                     DMU_POOL_ROOT_DATASET, sizeof (uint64_t), 1, &ddobj, tx));
399         }
400         VERIFY(0 == dmu_bonus_hold(mos, ddobj, FTAG, &dbuf));
401         dmu_buf_will_dirty(dbuf, tx);
402         ddphys = dbuf->db_data;

404         ddphys->dd_creation_time = gethrestime_sec();
405         if (pds)
406                 ddphys->dd_parent_obj = pds->dd_object;
407         ddphys->dd_props_zapobj = zap_create(mos,
408             DMU_OT_DSL_PROPS, DMU_OT_NONE, 0, tx);
409         ddphys->dd_child_dir_zapobj = zap_create(mos,
410             DMU_OT_DSL_DIR_CHILD_MAP, DMU_OT_NONE, 0, tx);
411         if (spa_version(dp->dp_spa) >= SPA_VERSION_USED_BREAKDOWN)
412                 ddphys->dd_flags |= DD_FLAG_USED_BREAKDOWN;
413         dmu_buf_rele(dbuf, FTAG);

415         return (ddobj);
416 }

418 boolean_t
419 dsl_dir_is_clone(dsl_dir_t *dd)
420 {
421         return (dd->dd_phys->dd_origin_obj &&
422             (dd->dd_pool->dp_origin_snap == NULL ||
423             dd->dd_phys->dd_origin_obj !=
424             dd->dd_pool->dp_origin_snap->ds_object));
425 }

427 void
428 dsl_dir_stats(dsl_dir_t *dd, nvlist_t *nv)
429 {
430         mutex_enter(&dd->dd_lock);
431         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USED,
432             dd->dd_phys->dd_used_bytes);
433         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_QUOTA, dd->dd_phys->dd_quota);
434         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_RESERVATION,
435             dd->dd_phys->dd_reserved);
436         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_COMPRESSRATIO,
437             dd->dd_phys->dd_compressed_bytes == 0 ? 100 :
438             (dd->dd_phys->dd_uncompressed_bytes * 100 /
439             dd->dd_phys->dd_compressed_bytes));
440         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_LOGICALUSED,
441             dd->dd_phys->dd_uncompressed_bytes);
442         if (dd->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
443                 dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USEDSNAP,
444                     dd->dd_phys->dd_used_breakdown[DD_USED_SNAP]);
445                 dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USEDDS,
446                     dd->dd_phys->dd_used_breakdown[DD_USED_HEAD]);
447                 dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USEDREFRESERV,
448                     dd->dd_phys->dd_used_breakdown[DD_USED_REFRSRV]);
449                 dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USEDCHILD,
450                     dd->dd_phys->dd_used_breakdown[DD_USED_CHILD] +
451                     dd->dd_phys->dd_used_breakdown[DD_USED_CHILD_RSRV]);
452         }
453         mutex_exit(&dd->dd_lock);

455         if (dsl_dir_is_clone(dd)) {
456                 dsl_dataset_t *ds;
457                 char buf[MAXNAMELEN];
```



```
459                 VERIFY0(dsl_dataset_hold_obj(dd->dd_pool,
460                     dd->dd_phys->dd_origin_obj, FTAG, &ds));
461                 dsl_dataset_name(ds, buf);
462                 dsl_dataset_rele(ds, FTAG);
463                 dsl_prop_nvlist_add_string(nv, ZFS_PROP_ORIGIN, buf);
464         }
465 }

467 void
468 dsl_dir_dirty(dsl_dir_t *dd, dmu_tx_t *tx)
469 {
470         dsl_pool_t *dp = dd->dd_pool;

472         ASSERT(dd->dd_phys);

474         if (txg_list_add(&dp->dp_dirty_dirs, dd, tx->tx_txg)) {
475                 /* up the hold count until we can be written out */
476                 dmu_buf_add_ref(dd->dd_dbuf, dd);
477         }
478 }

480 static int64_t
481 parent_delta(dsl_dir_t *dd, uint64_t used, int64_t delta)
482 {
483         uint64_t old_accounted = MAX(used, dd->dd_phys->dd_reserved);
484         uint64_t new_accounted = MAX(used + delta, dd->dd_phys->dd_reserved);
485         return (new_accounted - old_accounted);
486 }

488 void
489 dsl_dir_sync(dsl_dir_t *dd, dmu_tx_t *tx)
490 {
491         ASSERT(dmu_tx_is_syncing(tx));

493         mutex_enter(&dd->dd_lock);
494         ASSERT0(dd->dd_tempreserved[tx->tx_txg&TXG_MASK]);
495         dprintf_dd(dd, "txg=%llu towrite=%lluK\n", tx->tx_txg,
496             dd->dd_space_towrite[tx->tx_txg&TXG_MASK] / 1024);
497         dd->dd_space_towrite[tx->tx_txg&TXG_MASK] = 0;
498         mutex_exit(&dd->dd_lock);

500         /* release the hold from dsl_dir_dirty */
501         dmu_buf_rele(dd->dd_dbuf, dd);
502 }

504 static uint64_t
505 dsl_dir_space_towrite(dsl_dir_t *dd)
506 {
507         uint64_t space = 0;
508         int i;

510         ASSERT(MUTEX_HELD(&dd->dd_lock));

512         for (i = 0; i < TXG_SIZE; i++) {
513                 space += dd->dd_space_towrite[i&TXG_MASK];
514                 ASSERT3U(dd->dd_space_towrite[i&TXG_MASK], >=, 0);
515         }
516         return (space);
517 }

519 /*
520  * How much space would dd have available if ancestor had delta applied
521  * to it?  If ondiskonly is set, we're only interested in what's
522  * on-disk, not estimated pending changes.
523  */
```

```
524 uint64_t
525 dsl_dir_space_available(dsl_dir_t *dd,
526     dsl_dir_t *ancestor, int64_t delta, int ondiskonly)
527 {
528         uint64_t parentspace, myspace, quota, used;
529
530         /*
531          * If there are no restrictions otherwise, assume we have
532          * unlimited space available.
533          */
534         quota = UINT64_MAX;
535         parentspace = UINT64_MAX;
536
537         if (dd->dd_parent != NULL) {
538                 parentspace = dsl_dir_space_available(dd->dd_parent,
539                     ancestor, delta, ondiskonly);
540         }
541
542         mutex_enter(&dd->dd_lock);
543         if (dd->dd_phys->dd_quota != 0)
544                 quota = dd->dd_phys->dd_quota;
545         used = dd->dd_phys->dd_used_bytes;
546         if (!ondiskonly)
547                 used += dsl_dir_space_towrite(dd);
548
549         if (dd->dd_parent == NULL) {
550                 uint64_t poolsize = dsl_pool_adjustedsize(dd->dd_pool, FALSE);
551                 quota = MIN(quota, poolsize);
552         }
553
554         if (dd->dd_phys->dd_reserved > used && parentspace != UINT64_MAX) {
555                 /*
556                  * We have some space reserved, in addition to what our
557                  * parent gave us.
558                  */
559                 parentspace += dd->dd_phys->dd_reserved - used;
560         }
561
562         if (dd == ancestor) {
563                 ASSERT(delta <= 0);
564                 ASSERT(used >= -delta);
565                 used += delta;
566                 if (parentspace != UINT64_MAX)
567                         parentspace -= delta;
568         }
569
570         if (used > quota) {
571                 /* over quota */
572                 myspace = 0;
573         } else {
574                 /*
575                  * the lesser of the space provided by our parent and
576                  * the space left in our quota
577                  */
578                 myspace = MIN(parentspace, quota - used);
579         }
580
581         mutex_exit(&dd->dd_lock);
582
583         return (myspace);
584 }
585
586 struct tempreserve {
587         list_node_t tr_node;
588         dsl_pool_t *tr_dp;
589         dsl_dir_t *tr_ds;
```



```
590         uint64_t tr_size;
591 };
592
593 static int
594 dsl_dir_tempreserve_impl(dsl_dir_t *dd, uint64_t asize, boolean_t netfree,
595     boolean_t ignorequota, boolean_t checkrefquota, list_t *tr_list,
596     dmu_tx_t *tx, boolean_t first)
597 {
598         uint64_t txg = tx->tx_txg;
599         uint64_t est_inflight, used_on_disk, quota, parent_rsrv;
600         uint64_t deferred = 0;
601         struct tempreserve *tr;
602         int retval = EDQUOT;
603         int txgidx = txg & TXG_MASK;
604         int i;
605         uint64_t ref_rsrv = 0;
606
607         ASSERT3U(txg, !=, 0);
608         ASSERT3S(asize, >, 0);
609
610         mutex_enter(&dd->dd_lock);
611
612         /*
613          * Check against the dsl_dir's quota.  We don't add in the delta
614          * when checking for over-quota because they get one free hit.
615          */
616         est_inflight = dsl_dir_space_towrite(dd);
617         for (i = 0; i < TXG_SIZE; i++)
618                 est_inflight += dd->dd_tempreserved[i];
619         used_on_disk = dd->dd_phys->dd_used_bytes;
620
621         /*
622          * On the first iteration, fetch the dataset's used-on-disk and
623          * refreservation values. Also, if checkrefquota is set, test if
624          * allocating this space would exceed the dataset's refquota.
625          */
626         if (first && tx->tx_objset) {
627                 int error;
628                 dsl_dataset_t *ds = tx->tx_objset->os_dsl_dataset;
629
630                 error = dsl_dataset_check_quota(ds, checkrefquota,
631                     asize, est_inflight, &used_on_disk, &ref_rsrv);
632                 if (error) {
633                         mutex_exit(&dd->dd_lock);
634                         return (error);
635                 }
636         }
637
638         /*
639          * If this transaction will result in a net free of space,
640          * we want to let it through.
641          */
642         if (ignorequota || netfree || dd->dd_phys->dd_quota == 0)
643                 quota = UINT64_MAX;
644         else
645                 quota = dd->dd_phys->dd_quota;
646
647         /*
648          * Adjust the quota against the actual pool size at the root
649          * minus any outstanding deferred frees.
650          * To ensure that it's possible to remove files from a full
651          * pool without inducing transient overcommits, we throttle
652          * netfree transactions against a quota that is slightly larger,
653          * but still within the pool's allocation slop.  In cases where
654          * we're very close to full, this will allow a steady trickle of
655          * removes to get through.
```

```
656          */
657         if (dd->dd_parent == NULL) {
658                 spa_t *spa = dd->dd_pool->dp_spa;
659                 uint64_t poolsize = dsl_pool_adjustedsize(dd->dd_pool, netfree);
660                 deferred = metaslab_class_get_deferred(spa_normal_class(spa));
661                 if (poolsize - deferred < quota) {
662                         quota = poolsize - deferred;
663                         retval = ENOSPC;
664                 }
665         }
666
667         /*
668          * If they are requesting more space, and our current estimate
669          * is over quota, they get to try again unless the actual
670          * on-disk is over quota and there are no pending changes (which
671          * may free up space for us).
672          */
673         if (used_on_disk + est_inflight >= quota) {
674                 if (est_inflight > 0 || used_on_disk < quota ||
675                     (retval == ENOSPC && used_on_disk < quota + deferred))
676                         retval = ERESTART;
677                 dprintf_dd(dd, "failing: used=%lluK inflight = %lluK "
678                     "quota=%lluK tr=%lluK err=%d\n",
679                     used_on_disk>>10, est_inflight>>10,
680                     quota>>10, asize>>10, retval);
681                 mutex_exit(&dd->dd_lock);
682                 return (SET_ERROR(retval));
683         }
684
685         /* We need to up our estimated delta before dropping dd_lock */
686         dd->dd_tempreserved[txgidx] += asize;
687
688         parent_rsrv = parent_delta(dd, used_on_disk + est_inflight,
689             asize - ref_rsrv);
690         mutex_exit(&dd->dd_lock);
691
692         tr = kmem_zalloc(sizeof (struct tempreserve), KM_SLEEP);
693         tr->tr_ds = dd;
694         tr->tr_size = asize;
695         list_insert_tail(tr_list, tr);
696
697         /* see if it's OK with our parent */
698         if (dd->dd_parent && parent_rsrv) {
699                 boolean_t ismos = (dd->dd_phys->dd_head_dataset_obj == 0);
700
701                 return (dsl_dir_tempreserve_impl(dd->dd_parent,
702                     parent_rsrv, netfree, ismos, TRUE, tr_list, tx, FALSE));
703         } else {
704                 return (0);
705         }
706 }
707
708 /*
709  * Reserve space in this dsl_dir, to be used in this tx's txg.
710  * After the space has been dirtied (and dsl_dir_willuse_space()
711  * has been called), the reservation should be canceled, using
712  * dsl_dir_tempreserve_clear().
713  */
714 int
715 dsl_dir_tempreserve_space(dsl_dir_t *dd, uint64_t lsize, uint64_t asize,
716     uint64_t fsize, uint64_t usize, void **tr_cookiep, dmu_tx_t *tx)
717 {
718         int err;
719         list_t *tr_list;
720
721         if (asize == 0) {
```



```
722                 *tr_cookiep = NULL;
723                 return (0);
724         }
725
726         tr_list = kmem_alloc(sizeof (list_t), KM_SLEEP);
727         list_create(tr_list, sizeof (struct tempreserve),
728             offsetof(struct tempreserve, tr_node));
729         ASSERT3S(asize, >, 0);
730         ASSERT3S(fsize, >=, 0);
731
732         err = arc_tempreserve_space(lsize, tx->tx_txg);
733         if (err == 0) {
734                 struct tempreserve *tr;
735
736                 tr = kmem_zalloc(sizeof (struct tempreserve), KM_SLEEP);
737                 tr->tr_size = lsize;
738                 list_insert_tail(tr_list, tr);
739
740                 err = dsl_pool_tempreserve_space(dd->dd_pool, asize, tx);
741         } else {
742                 if (err == EAGAIN) {
743                         txg_delay(dd->dd_pool, tx->tx_txg,
744                             MSEC2NSEC(10), MSEC2NSEC(10));
745                         err = SET_ERROR(ERESTART);
746                 }
747                 dsl_pool_memory_pressure(dd->dd_pool);
748         }
749
750         if (err == 0) {
751                 struct tempreserve *tr;
752
753                 tr = kmem_zalloc(sizeof (struct tempreserve), KM_SLEEP);
754                 tr->tr_dp = dd->dd_pool;
755                 tr->tr_size = asize;
756                 list_insert_tail(tr_list, tr);
757
758                 err = dsl_dir_tempreserve_impl(dd, asize, fsize >= asize,
759                     FALSE, asize > usize, tr_list, tx, TRUE);
760         }
761
762         if (err != 0)
763                 dsl_dir_tempreserve_clear(tr_list, tx);
764         else
765                 *tr_cookiep = tr_list;
766
767         return (err);
768 }
769
770 /*
771  * Clear a temporary reservation that we previously made with
772  * dsl_dir_tempreserve_space().
773  */
774 void
775 dsl_dir_tempreserve_clear(void *tr_cookie, dmu_tx_t *tx)
776 {
777         int txgidx = tx->tx_txg & TXG_MASK;
778         list_t *tr_list = tr_cookie;
779         struct tempreserve *tr;
780
781         ASSERT3U(tx->tx_txg, !=, 0);
782
783         if (tr_cookie == NULL)
784                 return;
785
786         while (tr = list_head(tr_list)) {
787                 if (tr->tr_dp) {
```

```
788                         dsl_pool_tempreserve_clear(tr->tr_dp, tr->tr_size, tx);
789                 } else if (tr->tr_ds) {
790                         mutex_enter(&tr->tr_ds->dd_lock);
791                         ASSERT3U(tr->tr_ds->dd_tempreserved[txgidx], >=,
792                             tr->tr_size);
793                         tr->tr_ds->dd_tempreserved[txgidx] -= tr->tr_size;
794                         mutex_exit(&tr->tr_ds->dd_lock);
795                 } else {
796                         arc_tempreserve_clear(tr->tr_size);
797                 }
798                 list_remove(tr_list, tr);
799                 kmem_free(tr, sizeof (struct tempreserve));
800         }
801
802         kmem_free(tr_list, sizeof (list_t));
803 }
804
805 static void
806 dsl_dir_willuse_space_impl(dsl_dir_t *dd, int64_t space, dmu_tx_t *tx)
807 {
808         int64_t parent_space;
809         uint64_t est_used;
810
811         mutex_enter(&dd->dd_lock);
812         if (space > 0)
813                 dd->dd_space_towrite[tx->tx_txg & TXG_MASK] += space;
814
815         est_used = dsl_dir_space_towrite(dd) + dd->dd_phys->dd_used_bytes;
816         parent_space = parent_delta(dd, est_used, space);
817         mutex_exit(&dd->dd_lock);
818
819         /* Make sure that we clean up dd_space_to* */
820         dsl_dir_dirty(dd, tx);
821
822         /* XXX this is potentially expensive and unnecessary... */
823         if (parent_space && dd->dd_parent)
824                 dsl_dir_willuse_space_impl(dd->dd_parent, parent_space, tx);
825 }
826
827 /*
828  * Call in open context when we think we're going to write/free space,
829  * eg. when dirtying data.  Be conservative (ie. OK to write less than
830  * this or free more than this, but don't write more or free less).
831  */
832 void
833 dsl_dir_willuse_space(dsl_dir_t *dd, int64_t space, dmu_tx_t *tx)
834 {
835         dsl_pool_willuse_space(dd->dd_pool, space, tx);
836         dsl_dir_willuse_space_impl(dd, space, tx);
837 }
838
839 /* call from syncing context when we actually write/free space for this dd */
840 void
841 dsl_dir_diduse_space(dsl_dir_t *dd, dd_used_t type,
842     int64_t used, int64_t compressed, int64_t uncompressed, dmu_tx_t *tx)
843 {
844         int64_t accounted_delta;
845         boolean_t needlock = !MUTEX_HELD(&dd->dd_lock);
846
847         ASSERT(dmu_tx_is_syncing(tx));
848         ASSERT(type < DD_USED_NUM);
849
850         if (needlock)
851                 mutex_enter(&dd->dd_lock);
852         accounted_delta = parent_delta(dd, dd->dd_phys->dd_used_bytes, used);
853         ASSERT(used >= 0 || dd->dd_phys->dd_used_bytes >= -used);
```



```
854         ASSERT(compressed >= 0 ||
855             dd->dd_phys->dd_compressed_bytes >= -compressed);
856         ASSERT(uncompressed >= 0 ||
857             dd->dd_phys->dd_uncompressed_bytes >= -uncompressed);
858         dmu_buf_will_dirty(dd->dd_dbuf, tx);
859         dd->dd_phys->dd_used_bytes += used;
860         dd->dd_phys->dd_uncompressed_bytes += uncompressed;
861         dd->dd_phys->dd_compressed_bytes += compressed;
862
863         if (dd->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
864                 ASSERT(used > 0 ||
865                     dd->dd_phys->dd_used_breakdown[type] >= -used);
866                 dd->dd_phys->dd_used_breakdown[type] += used;
867 #ifdef DEBUG
868                 dd_used_t t;
869                 uint64_t u = 0;
870                 for (t = 0; t < DD_USED_NUM; t++)
871                         u += dd->dd_phys->dd_used_breakdown[t];
872                 ASSERT3U(u, ==, dd->dd_phys->dd_used_bytes);
873 #endif
874         }
875         if (needlock)
876                 mutex_exit(&dd->dd_lock);
877
878         if (dd->dd_parent != NULL) {
879                 dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD,
880                     accounted_delta, compressed, uncompressed, tx);
881                 dsl_dir_transfer_space(dd->dd_parent,
882                     used - accounted_delta,
883                     DD_USED_CHILD_RSRV, DD_USED_CHILD, tx);
884         }
885 }
886
887 void
888 dsl_dir_transfer_space(dsl_dir_t *dd, int64_t delta,
889     dd_used_t oldtype, dd_used_t newtype, dmu_tx_t *tx)
890 {
891         boolean_t needlock = !MUTEX_HELD(&dd->dd_lock);
892
893         ASSERT(dmu_tx_is_syncing(tx));
894         ASSERT(oldtype < DD_USED_NUM);
895         ASSERT(newtype < DD_USED_NUM);
896
897         if (delta == 0 || !(dd->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN))
898                 return;
899
900         if (needlock)
901                 mutex_enter(&dd->dd_lock);
902         ASSERT(delta > 0 ?
903             dd->dd_phys->dd_used_breakdown[oldtype] >= delta :
904             dd->dd_phys->dd_used_breakdown[newtype] >= -delta);
905         ASSERT(dd->dd_phys->dd_used_bytes >= ABS(delta));
906         dmu_buf_will_dirty(dd->dd_dbuf, tx);
907         dd->dd_phys->dd_used_breakdown[oldtype] -= delta;
908         dd->dd_phys->dd_used_breakdown[newtype] += delta;
909         if (needlock)
910                 mutex_exit(&dd->dd_lock);
911 }
912
913 typedef struct dsl_dir_set_qr_arg {
914         const char *ddsqra_name;
915         zprop_source_t ddsqra_source;
916         uint64_t ddsqra_value;
917 } dsl_dir_set_qr_arg_t;
918
919 static int
```

```
dsl_dir_set_quota_check(void *arg, dmu_tx_t *tx)
{
        dsl_dir_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        int error;
        uint64_t towrite, newval;

        error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
        if (error != 0)
                return (error);

        error = dsl_prop_predict(ds->ds_dir, "quota",
            ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
        if (error != 0) {
                dsl_dataset_rele(ds, FTAG);
                return (error);
        }

        if (newval == 0) {
                dsl_dataset_rele(ds, FTAG);
                return (0);
        }

        mutex_enter(&ds->ds_dir->dd_lock);
        /*
         * If we are doing the preliminary check in open context, and
         * there are pending changes, then don't fail it, since the
         * pending changes could under-estimate the amount of space to be
         * freed up.
         */
        towrite = dsl_dir_space_towrite(ds->ds_dir);
        if ((dmu_tx_is_syncing(tx) || towrite == 0) &&
            (newval < ds->ds_dir->dd_phys->dd_reserved ||
            newval < ds->ds_dir->dd_phys->dd_used_bytes + towrite)) {
                error = SET_ERROR(ENOSPC);
        }
        mutex_exit(&ds->ds_dir->dd_lock);
        dsl_dataset_rele(ds, FTAG);
        return (error);
}

static void
dsl_dir_set_quota_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dir_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        uint64_t newval;

        VERIFY0(dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds));

        if (spa_version(dp->dp_spa) >= SPA_VERSION_RECVD_PROPS) {
#endif /* ! codereview */
                dsl_prop_set_sync_impl(ds, zfs_prop_to_name(ZFS_PROP_QUOTA),
                    ddsqra->ddsqra_source, sizeof (ddsqra->ddsqra_value), 1,
                    &ddsqra->ddsqra_value, tx);

                VERIFY0(dsl_prop_get_int_ds(ds,
                    zfs_prop_to_name(ZFS_PROP_QUOTA), &newval));
        } else {
                newval = ddsqra->ddsqra_value;
                spa_history_log_internal_ds(ds, "set", tx, "%s=%lld",
                    zfs_prop_to_name(ZFS_PROP_QUOTA), (longlong_t)newval);
        }
#endif /* ! codereview */
```



```
        dmu_buf_will_dirty(ds->ds_dir->dd_dbuf, tx);
        mutex_enter(&ds->ds_dir->dd_lock);
        ds->ds_dir->dd_phys->dd_quota = newval;
        mutex_exit(&ds->ds_dir->dd_lock);
        dsl_dataset_rele(ds, FTAG);
}

int
dsl_dir_set_quota(const char *ddname, zprop_source_t source, uint64_t quota)
{
        dsl_dir_set_qr_arg_t ddsqra;

        ddsqra.ddsqra_name = ddname;
        ddsqra.ddsqra_source = source;
        ddsqra.ddsqra_value = quota;

        return (dsl_sync_task(ddname, dsl_dir_set_quota_check,
            dsl_dir_set_quota_sync, &ddsqra, 0));
}

int
dsl_dir_set_reservation_check(void *arg, dmu_tx_t *tx)
{
        dsl_dir_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        dsl_dir_t *dd;
        uint64_t newval, used, avail;
        int error;

        error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
        if (error != 0)
                return (error);
        dd = ds->ds_dir;

        /*
         * If we are doing the preliminary check in open context, the
         * space estimates may be inaccurate.
         */
        if (!dmu_tx_is_syncing(tx)) {
                dsl_dataset_rele(ds, FTAG);
                return (0);
        }

        error = dsl_prop_predict(ds->ds_dir,
            zfs_prop_to_name(ZFS_PROP_RESERVATION),
            ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
        if (error != 0) {
                dsl_dataset_rele(ds, FTAG);
                return (error);
        }

        mutex_enter(&dd->dd_lock);
        used = dd->dd_phys->dd_used_bytes;
        mutex_exit(&dd->dd_lock);

        if (dd->dd_parent) {
                avail = dsl_dir_space_available(dd->dd_parent,
                    NULL, 0, FALSE);
        } else {
                avail = dsl_pool_adjustedsize(dd->dd_pool, B_FALSE) - used;
        }

        if (MAX(used, newval) > MAX(used, dd->dd_phys->dd_reserved)) {
                uint64_t delta = MAX(used, newval) -
```

```
1052                     MAX(used, dd->dd_phys->dd_reserved);

1054                 if (delta > avail ||
1055                     (dd->dd_phys->dd_quota > 0 &&
1056                     newval > dd->dd_phys->dd_quota))
1057                         error = SET_ERROR(ENOSPC);
1058         }

1060         dsl_dataset_rele(ds, FTAG);
1061         return (error);
1062 }

1064 void
1065 dsl_dir_set_reservation_sync_impl(dsl_dir_t *dd, uint64_t value, dmu_tx_t *tx)
1066 {
1067         uint64_t used;
1068         int64_t delta;

1070         dmu_buf_will_dirty(dd->dd_dbuf, tx);

1072         mutex_enter(&dd->dd_lock);
1073         used = dd->dd_phys->dd_used_bytes;
1074         delta = MAX(used, value) - MAX(used, dd->dd_phys->dd_reserved);
1075         dd->dd_phys->dd_reserved = value;

1077         if (dd->dd_parent != NULL) {
1078                 /* Roll up this additional usage into our ancestors */
1079                 dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD_RSRV,
1080                     delta, 0, 0, tx);
1081         }
1082         mutex_exit(&dd->dd_lock);
1083 }


1086 static void
1087 dsl_dir_set_reservation_sync(void *arg, dmu_tx_t *tx)
1088 {
1089         dsl_dir_set_qr_arg_t *ddsqra = arg;
1090         dsl_pool_t *dp = dmu_tx_pool(tx);
1091         dsl_dataset_t *ds;
1092         uint64_t newval;

1094         VERIFY0(dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds));

1096         if (spa_version(dp->dp_spa) >= SPA_VERSION_RECVD_PROPS) {
1097                 dsl_prop_set_sync_impl(ds,
1098                     zfs_prop_to_name(ZFS_PROP_RESERVATION),
  24         dsl_prop_set_sync_impl(ds, zfs_prop_to_name(ZFS_PROP_RESERVATION),
1099                     ddsqra->ddsqra_source, sizeof (ddsqra->ddsqra_value), 1,
1100                     &ddsqra->ddsqra_value, tx);

1102                 VERIFY0(dsl_prop_get_int_ds(ds,
1103                     zfs_prop_to_name(ZFS_PROP_RESERVATION), &newval));
1104         } else {
1105                 newval = ddsqra->ddsqra_value;
1106                 spa_history_log_internal_ds(ds, "set", tx, "%s=%lld",
1107                     zfs_prop_to_name(ZFS_PROP_RESERVATION),
1108                     (longlong_t)newval);
1109         }
1110 #endif /* ! codereview */

1112         dsl_dir_set_reservation_sync_impl(ds->ds_dir, newval, tx);
1113         dsl_dataset_rele(ds, FTAG);
1114 }

1116 int
```



```
1117 dsl_dir_set_reservation(const char *ddname, zprop_source_t source,
1118     uint64_t reservation)
1119 {
1120         dsl_dir_set_qr_arg_t ddsqra;

1122         ddsqra.ddsqra_name = ddname;
1123         ddsqra.ddsqra_source = source;
1124         ddsqra.ddsqra_value = reservation;

1126         return (dsl_sync_task(ddname, dsl_dir_set_reservation_check,
1127             dsl_dir_set_reservation_sync, &ddsqra, 0));
1128 }

1130 static dsl_dir_t *
1131 closest_common_ancestor(dsl_dir_t *ds1, dsl_dir_t *ds2)
1132 {
1133         for (; ds1; ds1 = ds1->dd_parent) {
1134                 dsl_dir_t *dd;
1135                 for (dd = ds2; dd; dd = dd->dd_parent) {
1136                         if (ds1 == dd)
1137                                 return (dd);
1138                 }
1139         }
1140         return (NULL);
1141 }

1143 /*
1144  * If delta is applied to dd, how much of that delta would be applied to
1145  * ancestor?  Syncing context only.
1146  */
1147 static int64_t
1148 would_change(dsl_dir_t *dd, int64_t delta, dsl_dir_t *ancestor)
1149 {
1150         if (dd == ancestor)
1151                 return (delta);

1153         mutex_enter(&dd->dd_lock);
1154         delta = parent_delta(dd, dd->dd_phys->dd_used_bytes, delta);
1155         mutex_exit(&dd->dd_lock);
1156         return (would_change(dd->dd_parent, delta, ancestor));
1157 }

1159 typedef struct dsl_dir_rename_arg {
1160         const char *ddra_oldname;
1161         const char *ddra_newname;
1162 } dsl_dir_rename_arg_t;

1164 /* ARGSUSED */
1165 static int
1166 dsl_valid_rename(dsl_pool_t *dp, dsl_dataset_t *ds, void *arg)
1167 {
1168         int *deltap = arg;
1169         char namebuf[MAXNAMELEN];

1171         dsl_dataset_name(ds, namebuf);

1173         if (strlen(namebuf) + *deltap >= MAXNAMELEN)
1174                 return (SET_ERROR(ENAMETOOLONG));
1175         return (0);
1176 }

1178 static int
1179 dsl_dir_rename_check(void *arg, dmu_tx_t *tx)
1180 {
1181         dsl_dir_rename_arg_t *ddra = arg;
1182         dsl_pool_t *dp = dmu_tx_pool(tx);
```

```
1183         dsl_dir_t *dd, *newparent;
1184         const char *mynewname;
1185         int error;
1186         int delta = strlen(ddra->ddra_newname) - strlen(ddra->ddra_oldname);

1188         /* target dir should exist */
1189         error = dsl_dir_hold(dp, ddra->ddra_oldname, FTAG, &dd, NULL);
1190         if (error != 0)
1191                 return (error);

1193         /* new parent should exist */
1194         error = dsl_dir_hold(dp, ddra->ddra_newname, FTAG,
1195             &newparent, &mynewname);
1196         if (error != 0) {
1197                 dsl_dir_rele(dd, FTAG);
1198                 return (error);
1199         }

1201         /* can't rename to different pool */
1202         if (dd->dd_pool != newparent->dd_pool) {
1203                 dsl_dir_rele(newparent, FTAG);
1204                 dsl_dir_rele(dd, FTAG);
1205                 return (SET_ERROR(ENXIO));
1206         }

1208         /* new name should not already exist */
1209         if (mynewname == NULL) {
1210                 dsl_dir_rele(newparent, FTAG);
1211                 dsl_dir_rele(dd, FTAG);
1212                 return (SET_ERROR(EEXIST));
1213         }

1215         /* if the name length is growing, validate child name lengths */
1216         if (delta > 0) {
1217                 error = dmu_objset_find_dp(dp, dd->dd_object, dsl_valid_rename,
1218                     &delta, DS_FIND_CHILDREN | DS_FIND_SNAPSHOTS);
1219                 if (error != 0) {
1220                         dsl_dir_rele(newparent, FTAG);
1221                         dsl_dir_rele(dd, FTAG);
1222                         return (error);
1223                 }
1224         }

1226         if (newparent != dd->dd_parent) {
1227                 /* is there enough space? */
1228                 uint64_t myspace =
1229                     MAX(dd->dd_phys->dd_used_bytes, dd->dd_phys->dd_reserved);

1231                 /* no rename into our descendant */
1232                 if (closest_common_ancestor(dd, newparent) == dd) {
1233                         dsl_dir_rele(newparent, FTAG);
1234                         dsl_dir_rele(dd, FTAG);
1235                         return (SET_ERROR(EINVAL));
1236                 }

1238                 error = dsl_dir_transfer_possible(dd->dd_parent,
1239                     newparent, myspace);
1240                 if (error != 0) {
1241                         dsl_dir_rele(newparent, FTAG);
1242                         dsl_dir_rele(dd, FTAG);
1243                         return (error);
1244                 }
1245         }

1247         dsl_dir_rele(newparent, FTAG);
1248         dsl_dir_rele(dd, FTAG);
```



```
1249         return (0);
1250 }

1252 static void
1253 dsl_dir_rename_sync(void *arg, dmu_tx_t *tx)
1254 {
1255         dsl_dir_rename_arg_t *ddra = arg;
1256         dsl_pool_t *dp = dmu_tx_pool(tx);
1257         dsl_dir_t *dd, *newparent;
1258         const char *mynewname;
1259         int error;
1260         objset_t *mos = dp->dp_meta_objset;

1262         VERIFY0(dsl_dir_hold(dp, ddra->ddra_oldname, FTAG, &dd, NULL));
1263         VERIFY0(dsl_dir_hold(dp, ddra->ddra_newname, FTAG, &newparent,
1264             &mynewname));

1266         /* Log this before we change the name. */
1267         spa_history_log_internal_dd(dd, "rename", tx,
1268             "-> %s", ddra->ddra_newname);

1270         if (newparent != dd->dd_parent) {
1271                 dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD,
1272                     -dd->dd_phys->dd_used_bytes,
1273                     -dd->dd_phys->dd_compressed_bytes,
1274                     -dd->dd_phys->dd_uncompressed_bytes, tx);
1275                 dsl_dir_diduse_space(newparent, DD_USED_CHILD,
1276                     dd->dd_phys->dd_used_bytes,
1277                     dd->dd_phys->dd_compressed_bytes,
1278                     dd->dd_phys->dd_uncompressed_bytes, tx);

1280                 if (dd->dd_phys->dd_reserved > dd->dd_phys->dd_used_bytes) {
1281                         uint64_t unused_rsrv = dd->dd_phys->dd_reserved -
1282                             dd->dd_phys->dd_used_bytes;

1284                         dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD_RSRV,
1285                             -unused_rsrv, 0, 0, tx);
1286                         dsl_dir_diduse_space(newparent, DD_USED_CHILD_RSRV,
1287                             unused_rsrv, 0, 0, tx);
1288                 }
1289         }

1291         dmu_buf_will_dirty(dd->dd_dbuf, tx);

1293         /* remove from old parent zapobj */
1294         error = zap_remove(mos, dd->dd_parent->dd_phys->dd_child_dir_zapobj,
1295             dd->dd_myname, tx);
1296         ASSERT0(error);

1298         (void) strcpy(dd->dd_myname, mynewname);
1299         dsl_dir_rele(dd->dd_parent, dd);
1300         dd->dd_phys->dd_parent_obj = newparent->dd_object;
1301         VERIFY0(dsl_dir_hold_obj(dp,
1302             newparent->dd_object, NULL, dd, &dd->dd_parent));

1304         /* add to new parent zapobj */
1305         VERIFY0(zap_add(mos, newparent->dd_phys->dd_child_dir_zapobj,
1306             dd->dd_myname, 8, 1, &dd->dd_object, tx));

1308         dsl_prop_notify_all(dd);

1310         dsl_dir_rele(newparent, FTAG);
1311         dsl_dir_rele(dd, FTAG);
1312 }

1314 int
```

```
1315 dsl_dir_rename(const char *oldname, const char *newname)
1316 {
1317         dsl_dir_rename_arg_t ddra;

1319         ddra.ddra_oldname = oldname;
1320         ddra.ddra_newname = newname;

1322         return (dsl_sync_task(oldname,
1323             dsl_dir_rename_check, dsl_dir_rename_sync, &ddra, 3));
1324 }

1326 int
1327 dsl_dir_transfer_possible(dsl_dir_t *sdd, dsl_dir_t *tdd, uint64_t space)
1328 {
1329         dsl_dir_t *ancestor;
1330         int64_t adelta;
1331         uint64_t avail;

1333         ancestor = closest_common_ancestor(sdd, tdd);
1334         adelta = would_change(sdd, -space, ancestor);
1335         avail = dsl_dir_space_available(tdd, ancestor, adelta, FALSE);
1336         if (avail < space)
1337                 return (SET_ERROR(ENOSPC));

1339         return (0);
1340 }

1342 timestruc_t
1343 dsl_dir_snap_cmtime(dsl_dir_t *dd)
1344 {
1345         timestruc_t t;

1347         mutex_enter(&dd->dd_lock);
1348         t = dd->dd_snap_cmtime;
1349         mutex_exit(&dd->dd_lock);

1351         return (t);
1352 }

1354 void
1355 dsl_dir_snap_cmtime_update(dsl_dir_t *dd)
1356 {
1357         timestruc_t t;

1359         gethrestime(&t);
1360         mutex_enter(&dd->dd_lock);
1361         dd->dd_snap_cmtime = t;
1362         mutex_exit(&dd->dd_lock);
1363 }
```

```
**********************************************************
   29134 Tue Apr 23 10:00:50 2013
new/usr/src/uts/common/fs/zfs/dsl_prop.c
3739 cannot set zfs quota or reservation on pool version < 22
**********************************************************
```

```
/*
 * 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) 2013 Martin Matuska. All rights reserved.
#endif /* ! codereview */
 */

#include <sys/zfs_context.h>
#include <sys/dmu.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/spa.h>
#include <sys/zap.h>
#include <sys/fs/zfs.h>

#include "zfs_prop.h"

#define ZPROP_INHERIT_SUFFIX "$inherit"
#define ZPROP_RECVD_SUFFIX "$recvd"

static int
dodefault(const char *propname, int intsz, int numints, void *buf)
{
        zfs_prop_t prop;

        /*
         * The setonce properties are read-only, BUT they still
         * have a default value that can be used as the initial
         * value.
         */
        if ((prop = zfs_name_to_prop(propname)) == ZPROP_INVAL ||
            (zfs_prop_readonly(prop) && !zfs_prop_setonce(prop)))
                return (SET_ERROR(ENOENT));

        if (zfs_prop_get_type(prop) == PROP_TYPE_STRING) {
                if (intsz != 1)
                        return (SET_ERROR(EOVERFLOW));
```



```
                (void) strncpy(buf, zfs_prop_default_string(prop),
                    numints);
        } else {
                if (intsz != 8 || numints < 1)
                        return (SET_ERROR(EOVERFLOW));

                *(uint64_t *)buf = zfs_prop_default_numeric(prop);
        }

        return (0);
}

int
dsl_prop_get_dd(dsl_dir_t *dd, const char *propname,
    int intsz, int numints, void *buf, char *setpoint, boolean_t snapshot)
{
        int err = ENOENT;
        dsl_dir_t *target = dd;
        objset_t *mos = dd->dd_pool->dp_meta_objset;
        zfs_prop_t prop;
        boolean_t inheritable;
        boolean_t inheriting = B_FALSE;
        char *inheritstr;
        char *recvdstr;

        ASSERT(dsl_pool_config_held(dd->dd_pool));

        if (setpoint)
                setpoint[0] = '\0';

        prop = zfs_name_to_prop(propname);
        inheritable = (prop == ZPROP_INVAL || zfs_prop_inheritable(prop));
        inheritstr = kmem_asprintf("%s%s", propname, ZPROP_INHERIT_SUFFIX);
        recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

        /*
         * Note: dd may become NULL, therefore we shouldn't dereference it
         * after this loop.
         */
        for (; dd != NULL; dd = dd->dd_parent) {
                if (dd != target || snapshot) {
                        if (!inheritable)
                                break;
                        inheriting = B_TRUE;
                }

                /* Check for a local value. */
                err = zap_lookup(mos, dd->dd_phys->dd_props_zapobj, propname,
                    intsz, numints, buf);
                if (err != ENOENT) {
                        if (setpoint != NULL && err == 0)
                                dsl_dir_name(dd, setpoint);
                        break;
                }

                /*
                 * Skip the check for a received value if there is an explicit
                 * inheritance entry.
                 */
                err = zap_contains(mos, dd->dd_phys->dd_props_zapobj,
                    inheritstr);
                if (err != 0 && err != ENOENT)
                        break;

                if (err == ENOENT) {
                        /* Check for a received value. */
```

```
128                         err = zap_lookup(mos, dd->dd_phys->dd_props_zapobj,
129                             recvdstr, intsz, numints, buf);
130                         if (err != ENOENT) {
131                                 if (setpoint != NULL && err == 0) {
132                                         if (inheriting) {
133                                                 dsl_dir_name(dd, setpoint);
134                                         } else {
135                                                 (void) strcpy(setpoint,
136                                                     ZPROP_SOURCE_VAL_RECVD);
137                                         }
138                                 }
139                                 break;
140                         }
141                 }
142
143                 /*
144                  * If we found an explicit inheritance entry, err is zero even
145                  * though we haven't yet found the value, so reinitializing err
146                  * at the end of the loop (instead of at the beginning) ensures
147                  * that err has a valid post-loop value.
148                  */
149                 err = SET_ERROR(ENOENT);
150         }
151
152         if (err == ENOENT)
153                 err = dodefault(propname, intsz, numints, buf);
154
155         strfree(inheritstr);
156         strfree(recvdstr);
157
158         return (err);
159 }
160
161 int
162 dsl_prop_get_ds(dsl_dataset_t *ds, const char *propname,
163     int intsz, int numints, void *buf, char *setpoint)
164 {
165         zfs_prop_t prop = zfs_name_to_prop(propname);
166         boolean_t inheritable;
167         boolean_t snapshot;
168         uint64_t zapobj;
169
170         ASSERT(dsl_pool_config_held(ds->ds_dir->dd_pool));
171         inheritable = (prop == ZPROP_INVAL || zfs_prop_inheritable(prop));
172         snapshot = (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds));
173         zapobj = (ds->ds_phys == NULL ? 0 : ds->ds_phys->ds_props_obj);
174
175         if (zapobj != 0) {
176                 objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
177                 int err;
178
179                 ASSERT(snapshot);
180
181                 /* Check for a local value. */
182                 err = zap_lookup(mos, zapobj, propname, intsz, numints, buf);
183                 if (err != ENOENT) {
184                         if (setpoint != NULL && err == 0)
185                                 dsl_dataset_name(ds, setpoint);
186                         return (err);
187                 }
188
189                 /*
190                  * Skip the check for a received value if there is an explicit
191                  * inheritance entry.
192                  */
193                 if (inheritable) {
```



```
194                         char *inheritstr = kmem_asprintf("%s%s", propname,
195                             ZPROP_INHERIT_SUFFIX);
196                         err = zap_contains(mos, zapobj, inheritstr);
197                         strfree(inheritstr);
198                         if (err != 0 && err != ENOENT)
199                                 return (err);
200                 }
201
202                 if (err == ENOENT) {
203                         /* Check for a received value. */
204                         char *recvdstr = kmem_asprintf("%s%s", propname,
205                             ZPROP_RECVD_SUFFIX);
206                         err = zap_lookup(mos, zapobj, recvdstr,
207                             intsz, numints, buf);
208                         strfree(recvdstr);
209                         if (err != ENOENT) {
210                                 if (setpoint != NULL && err == 0)
211                                         (void) strcpy(setpoint,
212                                             ZPROP_SOURCE_VAL_RECVD);
213                                 return (err);
214                         }
215                 }
216         }
217
218         return (dsl_prop_get_dd(ds->ds_dir, propname,
219             intsz, numints, buf, setpoint, snapshot));
220 }
221
222 /*
223  * Register interest in the named property.  We'll call the callback
224  * once to notify it of the current property value, and again each time
225  * the property changes, until this callback is unregistered.
226  *
227  * Return 0 on success, errno if the prop is not an integer value.
228  */
229 int
230 dsl_prop_register(dsl_dataset_t *ds, const char *propname,
231     dsl_prop_changed_cb_t *callback, void *cbarg)
232 {
233         dsl_dir_t *dd = ds->ds_dir;
234         dsl_pool_t *dp = dd->dd_pool;
235         uint64_t value;
236         dsl_prop_cb_record_t *cbr;
237         int err;
238
239         ASSERT(dsl_pool_config_held(dp));
240
241         err = dsl_prop_get_int_ds(ds, propname, &value);
242         if (err != 0)
243                 return (err);
244
245         cbr = kmem_alloc(sizeof (dsl_prop_cb_record_t), KM_SLEEP);
246         cbr->cbr_ds = ds;
247         cbr->cbr_propname = kmem_alloc(strlen(propname)+1, KM_SLEEP);
248         (void) strcpy((char *)cbr->cbr_propname, propname);
249         cbr->cbr_func = callback;
250         cbr->cbr_arg = cbarg;
251         mutex_enter(&dd->dd_lock);
252         list_insert_head(&dd->dd_prop_cbs, cbr);
253         mutex_exit(&dd->dd_lock);
254
255         cbr->cbr_func(cbr->cbr_arg, value);
256         return (0);
257 }
258
259 int
```

```
260 dsl_prop_get(const char *dsname, const char *propname,
261     int intsz, int numints, void *buf, char *setpoint)
262 {
263         objset_t *os;
264         int error;
265
266         error = dmu_objset_hold(dsname, FTAG, &os);
267         if (error != 0)
268                 return (error);
269
270         error = dsl_prop_get_ds(dmu_objset_ds(os), propname,
271             intsz, numints, buf, setpoint);
272
273         dmu_objset_rele(os, FTAG);
274         return (error);
275 }
276
277 /*
278  * Get the current property value.  It may have changed by the time this
279  * function returns, so it is NOT safe to follow up with
280  * dsl_prop_register() and assume that the value has not changed in
281  * between.
282  *
283  * Return 0 on success, ENOENT if ddname is invalid.
284  */
285 int
286 dsl_prop_get_integer(const char *ddname, const char *propname,
287     uint64_t *valuep, char *setpoint)
288 {
289         return (dsl_prop_get(ddname, propname, 8, 1, valuep, setpoint));
290 }
291
292 int
293 dsl_prop_get_int_ds(dsl_dataset_t *ds, const char *propname,
294     uint64_t *valuep)
295 {
296         return (dsl_prop_get_ds(ds, propname, 8, 1, valuep, NULL));
297 }
298
299 /*
300  * Predict the effective value of the given special property if it were set with
301  * the given value and source. This is not a general purpose function. It exists
302  * only to handle the special requirements of the quota and reservation
303  * properties. The fact that these properties are non-inheritable greatly
304  * simplifies the prediction logic.
305  *
306  * Returns 0 on success, a positive error code on failure, or -1 if called with
307  * a property not handled by this function.
308  */
309 int
310 dsl_prop_predict(dsl_dir_t *dd, const char *propname,
311     zprop_source_t source, uint64_t value, uint64_t *newvalp)
312 {
313         zfs_prop_t prop = zfs_name_to_prop(propname);
314         objset_t *mos;
315         uint64_t zapobj;
316         uint64_t version;
317         char *recvdstr;
318         int err = 0;
319
320         switch (prop) {
321         case ZFS_PROP_QUOTA:
322         case ZFS_PROP_RESERVATION:
323         case ZFS_PROP_REFQUOTA:
324         case ZFS_PROP_REFRESERVATION:
325                 break;
```



```
326         default:
327                 return (-1);
328         }
329
330         mos = dd->dd_pool->dp_meta_objset;
331         zapobj = dd->dd_phys->dd_props_zapobj;
332         recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);
333
334         version = spa_version(dd->dd_pool->dp_spa);
335         if (version < SPA_VERSION_RECVD_PROPS) {
336                 if (source & ZPROP_SRC_NONE)
337                         source = ZPROP_SRC_NONE;
338                 else if (source & ZPROP_SRC_RECEIVED)
339                         source = ZPROP_SRC_LOCAL;
340         }
341
342         switch (source) {
343         case ZPROP_SRC_NONE:
344                 /* Revert to the received value, if any. */
345                 err = zap_lookup(mos, zapobj, recvdstr, 8, 1, newvalp);
346                 if (err == ENOENT)
347                         *newvalp = 0;
348                 break;
349         case ZPROP_SRC_LOCAL:
350                 *newvalp = value;
351                 break;
352         case ZPROP_SRC_RECEIVED:
353                 /*
354                  * If there's no local setting, then the new received value will
355                  * be the effective value.
356                  */
357                 err = zap_lookup(mos, zapobj, propname, 8, 1, newvalp);
358                 if (err == ENOENT)
359                         *newvalp = value;
360                 break;
361         case (ZPROP_SRC_NONE | ZPROP_SRC_RECEIVED):
362                 /*
363                  * We're clearing the received value, so the local setting (if
364                  * it exists) remains the effective value.
365                  */
366                 err = zap_lookup(mos, zapobj, propname, 8, 1, newvalp);
367                 if (err == ENOENT)
368                         *newvalp = 0;
369                 break;
370         default:
371                 panic("unexpected property source: %d", source);
372         }
373
374         strfree(recvdstr);
375
376         if (err == ENOENT)
377                 return (0);
378
379         return (err);
380 }
381
382 /*
383  * Unregister this callback.  Return 0 on success, ENOENT if ddname is
384  * invalid, ENOMSG if no matching callback registered.
385  */
386 int
387 dsl_prop_unregister(dsl_dataset_t *ds, const char *propname,
388     dsl_prop_changed_cb_t *callback, void *cbarg)
389 {
390         dsl_dir_t *dd = ds->ds_dir;
391         dsl_prop_cb_record_t *cbr;
```

```
393         mutex_enter(&dd->dd_lock);
394         for (cbr = list_head(&dd->dd_prop_cbs);
395             cbr; cbr = list_next(&dd->dd_prop_cbs, cbr)) {
396                 if (cbr->cbr_ds == ds &&
397                     cbr->cbr_func == callback &&
398                     cbr->cbr_arg == cbarg &&
399                     strcmp(cbr->cbr_propname, propname) == 0)
400                         break;
401         }

403         if (cbr == NULL) {
404                 mutex_exit(&dd->dd_lock);
405                 return (SET_ERROR(ENOMSG));
406         }

408         list_remove(&dd->dd_prop_cbs, cbr);
409         mutex_exit(&dd->dd_lock);
410         kmem_free((void*)cbr->cbr_propname, strlen(cbr->cbr_propname)+1);
411         kmem_free(cbr, sizeof (dsl_prop_cb_record_t));

413         return (0);
414 }

416 boolean_t
417 dsl_prop_hascb(dsl_dataset_t *ds)
418 {
419         dsl_dir_t *dd = ds->ds_dir;
420         boolean_t rv = B_FALSE;
421         dsl_prop_cb_record_t *cbr;

423         mutex_enter(&dd->dd_lock);
424         for (cbr = list_head(&dd->dd_prop_cbs); cbr;
425             cbr = list_next(&dd->dd_prop_cbs, cbr)) {
426                 if (cbr->cbr_ds == ds) {
427                         rv = B_TRUE;
428                         break;
429                 }
430         }
431         mutex_exit(&dd->dd_lock);
432         return (rv);
433 }

435 /* ARGSUSED */
436 static int
437 dsl_prop_notify_all_cb(dsl_pool_t *dp, dsl_dataset_t *ds, void *arg)
438 {
439         dsl_dir_t *dd = ds->ds_dir;
440         dsl_prop_cb_record_t *cbr;

442         mutex_enter(&dd->dd_lock);
443         for (cbr = list_head(&dd->dd_prop_cbs); cbr;
444             cbr = list_next(&dd->dd_prop_cbs, cbr)) {
445                 uint64_t value;

447                 if (dsl_prop_get_ds(cbr->cbr_ds, cbr->cbr_propname,
448                     sizeof (value), 1, &value, NULL) == 0)
449                         cbr->cbr_func(cbr->cbr_arg, value);
450         }
451         mutex_exit(&dd->dd_lock);

453         return (0);
454 }

456 /*
457  * Update all property values for ddobj & its descendants.  This is used
```



```
458  * when renaming the dir.
459  */
460 void
461 dsl_prop_notify_all(dsl_dir_t *dd)
462 {
463         dsl_pool_t *dp = dd->dd_pool;
464         ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));
465         (void) dmu_objset_find_dp(dp, dd->dd_object, dsl_prop_notify_all_cb,
466             NULL, DS_FIND_CHILDREN);
467 }

469 static void
470 dsl_prop_changed_notify(dsl_pool_t *dp, uint64_t ddobj,
471     const char *propname, uint64_t value, int first)
472 {
473         dsl_dir_t *dd;
474         dsl_prop_cb_record_t *cbr;
475         objset_t *mos = dp->dp_meta_objset;
476         zap_cursor_t zc;
477         zap_attribute_t *za;
478         int err;

480         ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));
481         err = dsl_dir_hold_obj(dp, ddobj, NULL, FTAG, &dd);
482         if (err)
483                 return;

485         if (!first) {
486                 /*
487                  * If the prop is set here, then this change is not
488                  * being inherited here or below; stop the recursion.
489                  */
490                 err = zap_contains(mos, dd->dd_phys->dd_props_zapobj, propname);
491                 if (err == 0) {
492                         dsl_dir_rele(dd, FTAG);
493                         return;
494                 }
495                 ASSERT3U(err, ==, ENOENT);
496         }

498         mutex_enter(&dd->dd_lock);
499         for (cbr = list_head(&dd->dd_prop_cbs); cbr;
500             cbr = list_next(&dd->dd_prop_cbs, cbr)) {
501                 uint64_t propobj = cbr->cbr_ds->ds_phys->ds_props_obj;

503                 if (strcmp(cbr->cbr_propname, propname) != 0)
504                         continue;

506                 /*
507                  * If the property is set on this ds, then it is not
508                  * inherited here; don't call the callback.
509                  */
510                 if (propobj && 0 == zap_contains(mos, propobj, propname))
511                         continue;

513                 cbr->cbr_func(cbr->cbr_arg, value);
514         }
515         mutex_exit(&dd->dd_lock);

517         za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
518         for (zap_cursor_init(&zc, mos,
519             dd->dd_phys->dd_child_dir_zapobj);
520             zap_cursor_retrieve(&zc, za) == 0;
521             zap_cursor_advance(&zc)) {
522                 dsl_prop_changed_notify(dp, za->za_first_integer,
523                     propname, value, FALSE);
```

```
524         }
525         kmem_free(za, sizeof (zap_attribute_t));
526         zap_cursor_fini(&zc);
527         dsl_dir_rele(dd, FTAG);
528 }

530 void
531 dsl_prop_set_sync_impl(dsl_dataset_t *ds, const char *propname,
532     zprop_source_t source, int intsz, int numints, const void *value,
533     dmu_tx_t *tx)
534 {
535         objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
536         uint64_t zapobj, intval, dummy;
537         int isint;
538         char valbuf[32];
539         const char *valstr = NULL;
540         char *inheritstr;
541         char *recvdstr;
542         char *tbuf = NULL;
543         int err;
544         uint64_t version = spa_version(ds->ds_dir->dd_pool->dp_spa);

546         isint = (dodefault(propname, 8, 1, &intval) == 0);

548         if (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds)) {
549                 ASSERT(version >= SPA_VERSION_SNAP_PROPS);
550                 if (ds->ds_phys->ds_props_obj == 0) {
551                         dmu_buf_will_dirty(ds->ds_dbuf, tx);
552                         ds->ds_phys->ds_props_obj =
553                             zap_create(mos,
554                             DMU_OT_DSL_PROPS, DMU_OT_NONE, 0, tx);
555                 }
556                 zapobj = ds->ds_phys->ds_props_obj;
557         } else {
558                 zapobj = ds->ds_dir->dd_phys->dd_props_zapobj;
559         }

561         if (version < SPA_VERSION_RECVD_PROPS) {
 24                 zfs_prop_t prop = zfs_name_to_prop(propname);
 25                 if (prop == ZFS_PROP_QUOTA || prop == ZFS_PROP_RESERVATION)
 26                         return;

562                 if (source & ZPROP_SRC_NONE)
563                         source = ZPROP_SRC_NONE;
564                 else if (source & ZPROP_SRC_RECEIVED)
565                         source = ZPROP_SRC_LOCAL;
566         }

568         inheritstr = kmem_asprintf("%s%s", propname, ZPROP_INHERIT_SUFFIX);
569         recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

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



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

639         strfree(inheritstr);
640         strfree(recvdstr);

642         if (isint) {
643                 VERIFY0(dsl_prop_get_int_ds(ds, propname, &intval));

645                 if (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds)) {
646                         dsl_prop_cb_record_t *cbr;
647                         /*
648                          * It's a snapshot; nothing can inherit this
649                          * property, so just look for callbacks on this
650                          * ds here.
651                          */
```

```
652                         mutex_enter(&ds->ds_dir->dd_lock);
653                         for (cbr = list_head(&ds->ds_dir->dd_prop_cbs); cbr;
654                             cbr = list_next(&ds->ds_dir->dd_prop_cbs, cbr)) {
655                                 if (cbr->cbr_ds == ds &&
656                                     strcmp(cbr->cbr_propname, propname) == 0)
657                                         cbr->cbr_func(cbr->cbr_arg, intval);
658                         }
659                         mutex_exit(&ds->ds_dir->dd_lock);
660                 } else {
661                         dsl_prop_changed_notify(ds->ds_dir->dd_pool,
662                             ds->ds_dir->dd_object, propname, intval, TRUE);
663                 }

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

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

683         if (tbuf != NULL)
684                 kmem_free(tbuf, ZAP_MAXVALUELEN);
685 }
```

_____***unchanged_portion_omitted_***