

```
*****
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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2013 by Delphix. All rights reserved.
24 * Copyright (c) 2013 Martin Matuska. All rights reserved.
25 #endif /* ! codereview */
26 */

28 #include <sys/dmu.h>
29 #include <sys/dmu_objset.h>
30 #include <sys/dmu_tx.h>
31 #include <sys/dsl_dataset.h>
32 #include <sys/dsl_dir.h>
33 #include <sys/dsl_prop.h>
34 #include <sys/dsl_syntask.h>
35 #include <sys/dsl_deleg.h>
36 #include <sys/spa.h>
37 #include <sys/metastab.h>
38 #include <sys/zap.h>
39 #include <sys/zio.h>
40 #include <sys/arc.h>
41 #include <sys/sunddi.h>
42 #include "zfs_namecheck.h"

44 static uint64_t dsl_dir_space_towrite(dsl_dir_t *dd);

46 /* ARGSUSED */
47 static void
48 dsl_dir_evict(dmu_buf_t *db, void *arg)
49 {
50     dsl_dir_t *dd = arg;
51     dsl_pool_t *dp = dd->dd_pool;
52     int t;

54     for (t = 0; t < TXG_SIZE; t++) {
55         ASSERT(!txg_list_member(&dp->dp_dirty_dirs, dd, t));
56         ASSERT(dd->dd_tempreserved[t] == 0);
57         ASSERT(dd->dd_space_towrite[t] == 0);
58     }
59     if (dd->dd_parent)
60         dsl_dir_rele(dd->dd_parent, dd);
61 }
```

```
63     spa_close(dd->dd_pool->dp_spa, dd);
64
65     /*
66      * The props callback list should have been cleaned up by
67      * objset_evict().
68      */
69     list_destroy(&dd->dd_prop_cbs);
70     mutex_destroy(&dd->dd_lock);
71     kmem_free(dd, sizeof (dsl_dir_t));
72 }

74 int
75 dsl_dir_hold_obj(dsl_pool_t *dp, uint64_t ddobj,
76                   const char *tail, void *tag, dsl_dir_t **ddp)
77 {
78     dmu_buf_t *dbuf;
79     dsl_dir_t *dd;
80     int err;

82     ASSERT(dsl_pool_config_held(dp));

84     err = dmu_bonus_hold(dp->dp_meta_objset, ddobj, tag, &dbuf);
85     if (err != 0)
86         return (err);
87     dd = dmu_buf_get_user(dbuf);
88 #ifdef ZFS_DEBUG
89     {
90         dmu_object_info_t doi;
91         dmu_object_info_from_db(dbuf, &doi);
92         ASSERT3U(doi.doi_type, ==, DMU_OT_DSL_DIR);
93         ASSERT3U(doi.doi_bonus_size, >=, sizeof (dsl_dir_phys_t));
94     }
95 #endif
96     if (dd == NULL) {
97         dsl_dir_t *winner;

99     dd = kmalloc(sizeof (dsl_dir_t), KM_SLEEP);
100    dd->dd_object = ddobj;
101    dd->dd_dbuf = dbuf;
102    dd->dd_pool = dp;
103    dd->dd_phys = dbuf->db_data;
104    mutex_init(&dd->dd_lock, NULL, MUTEX_DEFAULT, NULL);
105
106    list_create(&dd->dd_prop_cbs, sizeof (dsl_prop_cb_record_t),
107               offsetof(dsl_prop_cb_record_t, cbr_node));
108
109    dsl_dir_snap_cmtime_update(dd);

111    if (dd->dd_phys->dd_parent_obj) {
112        err = dsl_dir_hold_obj(dp, dd->dd_phys->dd_parent_obj,
113                               NULL, dd, &dd->dd_parent);
114        if (err != 0)
115            goto errout;
116        if (tail) {
117 #ifdef ZFS_DEBUG
118            uint64_t foundobj;
119
120            err = zap_lookup(dp->dp_meta_objset,
121                             dd->dd_parent->dd_phys->dd_child_dir_zapobj,
122                             tail, sizeof (foundobj), 1, &foundobj);
123            ASSERT(err || foundobj == ddobj);
124 #endif
125            (void) strcpy(dd->dd_myname, tail);
126        } else {
127            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         }
136
137         if (dsl_dir_is_clone(dd)) {
138             dmu_buf_t *origin_bonus;
139             dsl_dataset_phys_t *origin_phys;
140
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         }
155
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     }
168
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);
184
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 }
201
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 }
224
225 /* Calculate name length, avoiding all the strcat calls of dsl_dir_name */
226 int
227 dsl_dir_namerlen(dsl_dir_t *dd)
228 {
229     int result = 0;
230
231     if (dd->dd_parent) {
232         /* parent's name + 1 for the "/" */
233         result = dsl_dir_namerlen(dd->dd_parent) + 1;
234     }
235
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     }
244
245     return (result);
246 }
247
248 static int
249 getcomponent(const char *path, char *component, const char **nextp)
250 {
251     char *p;
252
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 }
```

```

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;
403
404     ddphys->dd_creation_time = getrestime_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);
414
415     return (ddobj);
416 }
417
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 }
426
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_USED_SNAP,
444                                    dd->dd_phys->dd_used_breakdown[DD_USED_SNAP]);
445         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USED_DS,
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);
454
455     if (dsl_dir_is_clone(dd)) {
456         dsl_dataset_t *ds;
457         char buf[MAXNAMELEN];

```

```

458         VERIFY0(dsl_dataset_hold_obj(dd->dd_pool,
459                                     dd->dd_phys->dd_origin_obj, FTAG, &ds));
460         dsl_dataset_name(ds, buf);
461         dsl_dataset_rele(ds, FTAG);
462         dsl_prop_nvlist_add_string(nv, ZFS_PROP_ORIGIN, buf);
463     }
464 }
465
466 void
467 dsl_dir_dirty(dsl_dir_t *dd, dmu_tx_t *tx)
468 {
469     dsl_pool_t *dp = dd->dd_pool;
470
471     ASSERT(dd->dd_phys);
472
473     if (txg_list_add(&dp->dp_dirty_dirs, dd, tx->tx_txg)) {
474         /* up the hold count until we can be written out */
475         dmu_buf_add_ref(dd->dd_dbuf, dd);
476     }
477 }
478
479 static int64_t
480 parent_delta(dsl_dir_t *dd, uint64_t used, int64_t delta)
481 {
482     uint64_t old_accounted = MAX(used, dd->dd_phys->dd_reserved);
483     uint64_t new_accounted = MAX(used + delta, dd->dd_phys->dd_reserved);
484     return (new_accounted - old_accounted);
485 }
486
487 void
488 dsl_dir_sync(dsl_dir_t *dd, dmu_tx_t *tx)
489 {
490     ASSERT(dmu_tx_is_syncing(tx));
491
492     mutex_enter(&dd->dd_lock);
493     ASSERT0(dd->dd_tempreserved[tx->tx_txg&TXG_MASK]);
494     dprintf_dd(dd, "txg=%llu towrite=%lluK\n", tx->tx_txg,
495                dd->dd_space_towrite[tx->tx_txg&TXG_MASK] / 1024);
496     dd->dd_space_towrite[tx->tx_txg&TXG_MASK] = 0;
497     mutex_exit(&dd->dd_lock);
498
499     /* release the hold from dsl_dir_dirty */
500     dmu_buf_rele(dd->dd_dbuf, dd);
501 }
502
503 static uint64_t
504 dsl_dir_space_towrite(dsl_dir_t *dd)
505 {
506     uint64_t space = 0;
507     int i;
508
509     ASSERT(MUTEX_HELD(&dd->dd_lock));
510
511     for (i = 0; i < TXG_SIZE; i++) {
512         space += dd->dd_space_towrite[i&TXG_MASK];
513         ASSERT3U(dd->dd_space_towrite[i&TXG_MASK], >=, 0);
514     }
515     return (space);
516 }
517
518 /*
519  * How much space would dd have available if ancestor had delta applied
520  * to it? If ondiskonly is set, we're only interested in what's
521  * on-disk, not estimated pending changes.
522  */
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      * reservation 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 = kmalloc(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 = kmalloc(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 = kmalloc(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 = kmalloc(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     }
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 kmem_free(tr_list, sizeof (list_t));
802 }
803 }

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;

811     mutex_enter(&dd->dd_lock);
812     if (space > 0)
813         dd->dd_space_towrite[tx->tx_txg & TXG_MASK] += space;

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);

819     /* Make sure that we clean up dd_space_to */
820     dsl_dir_dirty(dd, tx);

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 }

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 }

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);

847     ASSERT(dmu_tx_is_syncing(tx));
848     ASSERT(type < DD_USED_NUM);

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 || dd->dd_phys->dd_compressed_bytes >= -compressed);
855     ASSERT(uncompressed >= 0 || dd->dd_phys->dd_uncompressed_bytes >= -uncompressed);
856     dmu_buf_will_dirty(dd->dddbuf, tx);
857     dd->dd_phys->dd_used_bytes += used;
858     dd->dd_phys->dd_uncompressed_bytes += uncompressed;
859     dd->dd_phys->dd_compressed_bytes += compressed;

863     if (dd->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
864         ASSERT(used > 0 || dd->dd_phys->dd_used_breakdown[type] >= -used);
865         dd->dd_phys->dd_used_breakdown[type] += used;
866     }
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);

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 }

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);

893     ASSERT(dmu_tx_is_syncing(tx));
894     ASSERT(oldtype < DD_USED_NUM);
895     ASSERT(newtype < DD_USED_NUM);

897     if (delta == 0 || !(dd->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN))
898         return;

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->dddbuf, 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 }

913 typedef struct dsl_dir_set_gr_arg {
914     const char *ddsgra_name;
915     zprop_source_t ddsgra_source;
916     uint64_t ddsgra_value;
917 } dsl_dir_set_gr_arg_t;

919 static int

```

```

920 dsl_dir_set_quota_check(void *arg, dmu_tx_t *tx)
921 {
922     dsl_dir_set_gr_arg_t *ddsgra = arg;
923     dsl_pool_t *dp = dmu_tx_pool(tx);
924     dsl_dataset_t *ds;
925     int error;
926     uint64_t towrite, newval;

928     error = dsl_dataset_hold(dp, ddsgra->ddsgra_name, FTAG, &ds);
929     if (error != 0)
930         return (error);

932     error = dsl_prop_predict(ds->ds_dir, "quota",
933                             ddsgra->ddsgra_source, ddsgra->ddsgra_value, &newval);
934     if (error != 0) {
935         dsl_dataset_rele(ds, FTAG);
936         return (error);
937     }

939     if (newval == 0) {
940         dsl_dataset_rele(ds, FTAG);
941         return (0);
942     }

944     mutex_enter(&ds->ds_dir->dd_lock);
945     /*
946      * If we are doing the preliminary check in open context, and
947      * there are pending changes, then don't fail it, since the
948      * pending changes could under-estimate the amount of space to be
949      * freed up.
950     */
951     towrite = dsl_dir_space_towrite(ds->ds_dir);
952     if ((dmu_tx_is_syncing(tx) || towrite == 0) &&
953         (newval < ds->ds_dir->dd_phys->dd_reserved ||
954          newval < ds->ds_dir->dd_phys->dd_used_bytes + towrite)) {
955         error = SET_ERROR(ENOSPC);
956     }
957     mutex_exit(&ds->ds_dir->dd_lock);
958     dsl_dataset_rele(ds, FTAG);
959     return (error);
960 }

962 static void
963 dsl_dir_set_quota_sync(void *arg, dmu_tx_t *tx)
964 {
965     dsl_dir_set_gr_arg_t *ddsgra = arg;
966     dsl_pool_t *dp = dmu_tx_pool(tx);
967     dsl_dataset_t *ds;
968     uint64_t newval;

970     VERIFY0(dsl_dataset_hold(dp, ddsgra->ddsgra_name, FTAG, &ds));

972     if (spa_version(dp->dp_spa) >= SPA_VERSION_RECVD_PROPS) {
973 #endif /* ! codereview */
974         dsl_prop_set_sync_impl(ds, zfs_prop_to_name(ZFS_PROP_QUOTA),
975                               ddsgra->ddsgra_source, sizeof (ddsgra->ddsgra_value), 1,
976                               &ddsgra->ddsgra_value, tx);

978         VERIFY0(dsl_prop_get_int(ds,
979                                 zfs_prop_to_name(ZFS_PROP_QUOTA), &newval));
980     } else {
981         newval = ddsgra->ddsgra_value;
982         spa_history_log_internal_ds(ds, "set", tx, "%s=%lld",
983                                     zfs_prop_to_name(ZFS_PROP_QUOTA), (longlong_t)newval);
984     }
985 #endif /* ! codereview */

```

```

987     dmu_buf_will_dirty(ds->ds_dir->dddbuf, tx);
988     mutex_enter(&ds->ds_dir->dd_lock);
989     ds->ds_dir->dd_phys->dd_quota = newval;
990     mutex_exit(&ds->ds_dir->dd_lock);
991     dsl_dataset_rele(ds, FTAG);
992 }

994 int
995 dsl_dir_set_quota(const char *ddname, zprop_source_t source, uint64_t quota)
996 {
997     dsl_dir_set_gr_arg_t ddsgra;

999     ddsgra.ddsgra_name = ddname;
1000     ddsgra.ddsgra_source = source;
1001     ddsgra.ddsgra_value = quota;

1003     return (dsl_sync_task(ddname, dsl_dir_set_quota_check,
1004                           dsl_dir_set_quota_sync, &ddsgra, 0));
1005 }

1007 int
1008 dsl_dir_set_reservation_check(void *arg, dmu_tx_t *tx)
1009 {
1010     dsl_dir_set_gr_arg_t *ddsgra = arg;
1011     dsl_pool_t *dp = dmu_tx_pool(tx);
1012     dsl_dataset_t *ds;
1013     dsl_dir_t *dd;
1014     uint64_t newval, used, avail;
1015     int error;

1017     error = dsl_dataset_hold(dp, ddsgra->ddsgra_name, FTAG, &ds);
1018     if (error != 0)
1019         return (error);
1020     dd = ds->ds_dir;

1022     /*
1023      * If we are doing the preliminary check in open context, the
1024      * space estimates may be inaccurate.
1025     */
1026     if (!dmu_tx_is_syncing(tx)) {
1027         dsl_dataset_rele(ds, FTAG);
1028         return (0);
1029     }

1031     error = dsl_prop_predict(ds->ds_dir,
1032                             zfs_prop_to_name(ZFS_PROP_RESERVATION),
1033                             ddsgra->ddsgra_source, ddsgra->ddsgra_value, &newval);
1034     if (error != 0) {
1035         dsl_dataset_rele(ds, FTAG);
1036         return (error);
1037     }

1039     mutex_enter(&dd->dd_lock);
1040     used = dd->dd_phys->dd_used_bytes;
1041     mutex_exit(&dd->dd_lock);

1043     if (dd->dd_parent) {
1044         avail = dsl_dir_space_available(dd->dd_parent,
1045                                         NULL, 0, FALSE);
1046     } else {
1047         avail = dsl_pool_adjustedsize(dd->dd_pool, B_FALSE) - used;
1048     }

1050     if (MAX(used, newval) > MAX(used, dd->dd_phys->dd_reserved)) {
1051         uint64_t delta = MAX(used, newval) -

```

```

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

1053         if (delta > avail ||
1054             (dd->dd_phys->dd_quota > 0 &&
1055              newval > dd->dd_phys->dd_quota))
1056             error = SET_ERROR(ENOSPC);
1057     }

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;
1069
1070     dmu_buf_will_dirty(dd->dd_dbuf, tx);
1071
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;
1076
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_gr_arg_t *ddsgra = 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, ddsgra->ddsgra_name, FTAG, &ds));
1095
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),
1099             dsl_prop_set_sync_impl(ds, zfs_prop_to_name(ZFS_PROP_RESERVATION),
1100                         ddsgra->ddsgra_source, sizeof (ddsgra->ddsgra_value), 1,
1101                         &ddsgra->ddsgra_value, tx);

1102         VERIFY0(dsl_prop_get_int_ds(ds,
1103             zfs_prop_to_name(ZFS_PROP_RESERVATION), &newval));
1104     } else {
1105         newval = ddsgra->ddsgra_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_gr_arg_t ddsgra;
1121
1122     ddsgra.ddsgra_name = ddname;
1123     ddsgra.ddsgra_source = source;
1124     ddsgra.ddsgra_value = reservation;
1125
1126     return (dsl_sync_task(ddname, dsl_dir_set_reservation_check,
1127                           dsl_dir_set_reservation_sync, &ddsgra, 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);
1152
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 }

1158 typedef struct dsl_dir_rename_arg {
1159     const char *ddra_oldname;
1160     const char *ddra_newname;
1161 } 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];
1170
1171     dsl_dataset_name(ds, namebuf);
1172
1173     if (strlen(namebuf) + *deltap >= MAXNAMELEN)
1174         return (SET_ERROR(ENAMETOOLONG));
1175     return (0);
1176 }

1177 static int
1178 dsl_dir_rename_check(void *arg, dmu_tx_t *tx)
1179 {
1180     dsl_dir_rename_arg_t *ddra = arg;
1181     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);
1187
1188     /* target dir should exist */
1189     error = dsl_dir_hold(dp, ddra->ddra_oldname, FTAG, &dd, NULL);
1190     if (error != 0)
1191         return (error);
1192
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     }
1200
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     }
1207
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     }
1214
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     }
1225
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);
1230
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         }
1237
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     }
1246
1247     dsl_dir_rele(newparent, FTAG);
1248     dsl_dir_rele(dd, FTAG);

```

```

1249         return (0);
1250     }
1251
1252     static void
1253     dsl_dir_rename_sync(void *arg, dmux_tx_t *tx)
1254     {
1255         dsl_dir_rename_arg_t *ddra = arg;
1256         dsl_pool_t *dp = dmux_tx_pool(tx);
1257         dsl_dir_t *dd, *newparent;
1258         const char *mynewname;
1259         int error;
1260         objset_t *mos = dp->dp_meta_objset;
1261
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));
1265
1266         /* Log this before we change the name. */
1267         spa_history_log_internal_dd(dd, "rename", tx,
1268                                     "-> %s", ddra->ddra_newname);
1269
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);
1279
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;
1283
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         }
1290
1291         dmux_buf_will_dirty(dd->dddbuf, tx);
1292
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);
1297
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));
1303
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));
1307
1308         dsl_prop_notify_all(dd);
1309
1310         dsl_dir_rele(newparent, FTAG);
1311         dsl_dir_rele(dd, FTAG);
1312     }
1313
1314     int

```

```
1315 dsl_dir_rename(const char *oldname, const char *newname)
1316 {
1317     dsl_dir_rename_arg_t ddra;
1318
1319     ddra.ddra_oldname = oldname;
1320     ddra.ddra_newname = newname;
1321
1322     return (dsl_sync_task(oldname,
1323         dsl_dir_rename_check, dsl_dir_rename_sync, &ddra, 3));
1324 }
1325
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;
1332
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));
1338
1339     return (0);
1340 }
1341
1342 timestruc_t
1343 dsl_dir_snap_cmtime(dsl_dir_t *dd)
1344 {
1345     timestruc_t t;
1346
1347     mutex_enter(&dd->dd_lock);
1348     t = dd->dd_snap_cmtime;
1349     mutex_exit(&dd->dd_lock);
1350
1351     return (t);
1352 }
1353
1354 void
1355 dsl_dir_snap_cmtime_update(dsl_dir_t *dd)
1356 {
1357     timestruc_t t;
1358
1359     gethrestime(&t);
1360     mutex_enter(&dd->dd_lock);
1361     dd->dd_snap_cmtime = t;
1362     mutex_exit(&dd->dd_lock);
1363 }
```

new/usr/src/uts/common/fs/zfs/dsl_prop.c

```
*****
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
*****
```

```
1 /*
2  * CDDL HEADER START
3 *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7 *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2013 by Delphix. All rights reserved.
24 * Copyright (c) 2013 Martin Matuska. All rights reserved.
25 #endif /* ! codereview */
26 */

28 #include <sys/zfs_context.h>
29 #include <sys/dmu.h>
30 #include <sys/dmu_objset.h>
31 #include <sys/dmu_tx.h>
32 #include <sys/dsl_dataset.h>
33 #include <sys/dsl_dir.h>
34 #include <sys/dsl_prop.h>
35 #include <sys/dsl_syntask.h>
36 #include <sys/spa.h>
37 #include <sys/zap.h>
38 #include <sys/fs/zfs.h>

40 #include "zfs_prop.h"

42 #define ZPROP_INHERIT_SUFFIX "$inherit"
43 #define ZPROP_RECVD_SUFFIX "$recv"

45 static int
46 dodefault(const char *propname, int intsz, int numints, void *buf)
47 {
48     zfs_prop_t prop;

50     /*
51      * The setonce properties are read-only, BUT they still
52      * have a default value that can be used as the initial
53      * value.
54      */
55     if ((prop = zfs_name_to_prop(propname)) == ZPROP_INVAL ||
56         (zfs_prop_READONLY(prop) && !zfs_prop_setonce(prop)))
57         return (SET_ERROR(ENOENT));

59     if (zfs_prop_get_type(prop) == PROP_TYPE_STRING) {
60         if (intsz != 1)
61             return (SET_ERROR(EOVERFLOW));
62     }
```

1

```
new/usr/src/uts/common/fs/zfs/dsl_prop.c
```

```
62             (void) strncpy(buf, zfs_prop_default_string(prop),
63                          numints);
64         } else {
65             if (intsz != 8 || numints < 1)
66                 return (SET_ERROR(EOVERFLOW));
67             *(uint64_t *)buf = zfs_prop_default_numeric(prop);
68         }
69     }
70     return (0);
71 }

74 int
75 dsl_prop_get_dd(dsl_dir_t *dd, const char *propname,
76                  int intsz, int numints, void *buf, char *setpoint, boolean_t snapshot)
77 {
78     int err = ENOENT;
79     dsl_dir_t *target = dd;
80     objset_t *mos = dd->dd_pool->dp_meta_objset;
81     zfs_prop_t prop;
82     boolean_t inheritable;
83     boolean_t inheriting = B_FALSE;
84     char *inheritstr;
85     char *recvstr;

87     ASSERT(dsl_pool_config_held(dd->dd_pool));

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

92     prop = zfs_name_to_prop(propname);
93     inheritable = (prop == ZPROP_INVAL || zfs_prop_inheritable(prop));
94     inheritstr = kmem_asprintf("%s%s", propname, ZPROP_INHERIT_SUFFIX);
95     recvstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

97     /*
98      * Note: dd may become NULL, therefore we shouldn't dereference it
99      * after this loop.
100     */
101    for (; dd != NULL; dd = dd->dd_parent) {
102        if (dd != target || snapshot) {
103            if (!inheritable)
104                break;
105            inheriting = B_TRUE;
106        }

108        /* Check for a local value. */
109        err = zap_lookup(mos, dd->dd_phys->dd_props_zapobj, propname,
110                        intsz, numints, buf);
111        if (err != ENOENT) {
112            if (setpoint != NULL && err == 0)
113                dsl_dir_name(dd, setpoint);
114            break;
115        }

117        /*
118         * Skip the check for a received value if there is an explicit
119         * inheritance entry.
120         */
121        err = zap_contains(mos, dd->dd_phys->dd_props_zapobj,
122                           inheritstr);
123        if (err != 0 && err != ENOENT)
124            break;

126        if (err == ENOENT) {
127            /* Check for a received value. */
128        }
129    }
130 }
```

2

```

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(revdstr);
157 
158 return (err);
159 }

160 int
161 dsl_prop_get_ds(dsl_dataset_t *ds, const char *propname,
162                  int intsz, int numints, void *buf, char *setpoint)
163 {
164     zfs_prop_t prop = zfs_name_to_prop(propname);
165     boolean_t inheritable;
166     boolean_t snapshot;
167     uint64_t zapobj;
168 
169     ASSERT(dsl_pool_config_held(ds->ds_dir->dd_pool));
170     inheritable = (prop == ZPROP_INVAL || zfs_prop_inheritable(prop));
171     snapshot = (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds));
172     zapobj = (ds->ds_phys == NULL ? 0 : ds->ds_phys->ds_props_obj);
173 
174     if (zapobj != 0) {
175         objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
176         int err;
177 
178         ASSERT(snapshot);
179 
180         /* Check for a local value. */
181         err = zap_lookup(mos, zapobj, propname, intsz, numints, buf);
182         if (err != ENOENT) {
183             if (setpoint != NULL && err == 0)
184                 dsl_dataset_name(ds, setpoint);
185             return (err);
186         }
187 
188         /*
189          * Skip the check for a received value if there is an explicit
190          * inheritance entry.
191          */
192         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 *recvstr = kmem_asprintf("%s%s", propname,
205                                       ZPROP_RECV_SUFFIX);
206         err = zap_lookup(mos, zapobj, recvstr,
207                         intsz, numints, buf);
208         strfree(recvstr);
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     return (dsl_prop_get_dd(ds->ds_dir, propname,
218                             intsz, numints, buf, setpoint, snapshot));
219 }
220 
221 /*
222  * Register interest in the named property. We'll call the callback
223  * once to notify it of the current property value, and again each time
224  * the property changes, until this callback is unregistered.
225  *
226  * Return 0 on success, errno if the prop is not an integer value.
227  */
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 * Get the current property value. It may have changed by the time this
278 * function returns, so it is NOT safe to follow up with
279 * dsl_prop_register() and assume that the value has not changed in
280 * between.
281 *
282 * Return 0 on success, ENOENT if ddname is invalid.
283 */
284 int
285 dsl_prop_get_integer(const char *ddname, const char *propname,
286     uint64_t *valuep, char *setpoint)
287 {
288     return (dsl_prop_get(ddname, propname, 8, 1, valuep, setpoint));
289 }
290
291 int
292 dsl_prop_get_int_ds(dsl_dataset_t *ds, const char *propname,
293     uint64_t *valuep)
294 {
295     return (dsl_prop_get_ds(ds, propname, 8, 1, valuep, NULL));
296 }
297
298 /*
299 * Predict the effective value of the given special property if it were set with
300 * the given value and source. This is not a general purpose function. It exists
301 * only to handle the special requirements of the quota and reservation
302 * properties. The fact that these properties are non-inheritable greatly
303 * simplifies the prediction logic.
304 *
305 * Returns 0 on success, a positive error code on failure, or -1 if called with
306 * a property not handled by this function.
307 */
308 int
309 dsl_prop_predict(dsl_dir_t *dd, const char *propname,
310     zprop_source_t source, uint64_t value, uint64_t *newvalp)
311 {
312     zfs_prop_t prop = zfs_name_to_prop(propname);
313     objset_t *mos;
314     uint64_t zapobj;
315     uint64_t version;
316     char *recvdst;
317     int err = 0;
318
319     switch (prop) {
320     case ZFS_PROP_QUOTA:
321     case ZFS_PROP_RESERVATION:
322     case ZFS_PROP_REFQUOTA:
323     case ZFS_PROP_REFRESERVATION:
324         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     recvdst = 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, recvdst, 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(recvdst);
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 = 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     }
402
403     if (cbr == NULL) {
404         mutex_exit(&dd->dd_lock);
405         return (SET_ERROR(ENOMSG));
406     }
407
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));
412
413     return (0);
414 }
415
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;
422
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 }
434
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;
441
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;
446
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);
452
453     return (0);
454 }
455
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     }
468
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;
479
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;
484
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         }
497
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;
502
503             if (strcmp(cbr->cbr_propname, propname) != 0)
504                 continue;
505
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;
512
513             cbr->cbr_func(cbr->cbr_arg, value);
514         }
515         mutex_exit(&dd->dd_lock);
516
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 *recvdst;
542     char *tbuf = NULL;
543     int err;
544     uint64_t version = spa_version(ds->ds_dir->dd_pool->dp_spa);

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

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

558     if (version < SPA_VERSION_RECVD_PROPS) {
559         zfs_prop_t prop = zfs_name_to_prop(propname);
560         if (prop == ZFS_PROP_QUOTA || prop == ZFS_PROP_RESERVATION)
561             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     }

567     inheritstr = kmem_asprintf("%s%s", propname, ZPROP_INHERIT_SUFFIX);
568     recvdst = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

569     switch (source) {
570     case ZPROP_SRC_NONE:
571         /*
572          * revert to received value, if any (inherit -S)
573          * - remove propname
574          * - remove propname$inherit
575          */
576         err = zap_remove(mos, zapobj, propname, tx);
577         ASSERT(err == 0 || err == ENOENT);
578         err = zap_remove(mos, zapobj, inheritstr, tx);
579         ASSERT(err == 0 || err == ENOENT);
580         break;
581     case ZPROP_SRC_LOCAL:
582         /*
583          * remove propname$inherit
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
266
```

```
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 }
664
665 (void) sprintf(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 = kmalloc(ZAP_MAXVALUELEN, KM_SLEEP);
673         if (dsl_prop_get_ds(ds, propname, 1,
674                             ZAP_MAXVALUELEN, tbuf, NULL) == 0)
675             valstr = tbuf;
676     }
677 }
678
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));
682
683 if (tbuf != NULL)
684     kmem_free(tbuf, ZAP_MAXVALUELEN);
685 }
```

unchanged portion omitted