```
**********************************************************
  111486 Fri May 17 22:54:36 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.c
3699 zfs hold or release of a non-existent snapshot does not output error
3739 cannot set zfs quota or reservation on pool version < 22
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Eric Shrock <eric.schrock@delphix.com>
**********************************************************
```

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

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2012 DEY Storage Systems, Inc.  All rights reserved.
 * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2013 Martin Matuska. All rights reserved.
#endif /* ! codereview */
 */

#include <ctype.h>
#include <errno.h>
#include <libintl.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <stddef.h>
#include <zone.h>
#include <fcntl.h>
#include <sys/mntent.h>
#include <sys/mount.h>
#include <priv.h>
#include <pwd.h>
#include <grp.h>
#include <stddef.h>
#include <ucred.h>
#include <idmap.h>
#include <aclutils.h>
#include <directory.h>

#include <sys/dnode.h>
#include <sys/spa.h>
#include <sys/zap.h>
#include <libzfs.h>

#include "zfs_namecheck.h"
```



```
#include "zfs_prop.h"
#include "libzfs_impl.h"
#include "zfs_deleg.h"

static int userquota_propname_decode(const char *propname, boolean_t zoned,
    zfs_userquota_prop_t *typep, char *domain, int domainlen, uint64_t *ridp);

/*
 * Given a single type (not a mask of types), return the type in a human
 * readable form.
 */
const char *
zfs_type_to_name(zfs_type_t type)
{
        switch (type) {
        case ZFS_TYPE_FILESYSTEM:
                return (dgettext(TEXT_DOMAIN, "filesystem"));
        case ZFS_TYPE_SNAPSHOT:
                return (dgettext(TEXT_DOMAIN, "snapshot"));
        case ZFS_TYPE_VOLUME:
                return (dgettext(TEXT_DOMAIN, "volume"));
        }

        return (NULL);
}

/*
 * Given a path and mask of ZFS types, return a string describing this dataset.
 * This is used when we fail to open a dataset and we cannot get an exact type.
 * We guess what the type would have been based on the path and the mask of
 * acceptable types.
 */
static const char *
path_to_str(const char *path, int types)
{
        /*
         * When given a single type, always report the exact type.
         */
        if (types == ZFS_TYPE_SNAPSHOT)
                return (dgettext(TEXT_DOMAIN, "snapshot"));
        if (types == ZFS_TYPE_FILESYSTEM)
                return (dgettext(TEXT_DOMAIN, "filesystem"));
        if (types == ZFS_TYPE_VOLUME)
                return (dgettext(TEXT_DOMAIN, "volume"));

        /*
         * The user is requesting more than one type of dataset.  If this is the
         * case, consult the path itself.  If we're looking for a snapshot, and
         * a '@' is found, then report it as "snapshot".  Otherwise, remove the
         * snapshot attribute and try again.
         */
        if (types & ZFS_TYPE_SNAPSHOT) {
                if (strchr(path, '@') != NULL)
                        return (dgettext(TEXT_DOMAIN, "snapshot"));
                return (path_to_str(path, types & ~ZFS_TYPE_SNAPSHOT));
        }

        /*
         * The user has requested either filesystems or volumes.
         * We have no way of knowing a priori what type this would be, so always
         * report it as "filesystem" or "volume", our two primitive types.
         */
        if (types & ZFS_TYPE_FILESYSTEM)
                return (dgettext(TEXT_DOMAIN, "filesystem"));

        assert(types & ZFS_TYPE_VOLUME);
```

```
125         return (dgettext(TEXT_DOMAIN, "volume"));
126 }

128 /*
129  * Validate a ZFS path.  This is used even before trying to open the dataset, to
130  * provide a more meaningful error message.  We call zfs_error_aux() to
131  * explain exactly why the name was not valid.
132  */
133 int
134 zfs_validate_name(libzfs_handle_t *hdl, const char *path, int type,
135     boolean_t modifying)
136 {
137         namecheck_err_t why;
138         char what;

140         (void) zfs_prop_get_table();
141         if (dataset_namecheck(path, &why, &what) != 0) {
142                 if (hdl != NULL) {
143                         switch (why) {
144                         case NAME_ERR_TOOLONG:
145                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
146                                     "name is too long"));
147                                 break;

149                         case NAME_ERR_LEADING_SLASH:
150                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
151                                     "leading slash in name"));
152                                 break;

154                         case NAME_ERR_EMPTY_COMPONENT:
155                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
156                                     "empty component in name"));
157                                 break;

159                         case NAME_ERR_TRAILING_SLASH:
160                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
161                                     "trailing slash in name"));
162                                 break;

164                         case NAME_ERR_INVALCHAR:
165                                 zfs_error_aux(hdl,
166                                     dgettext(TEXT_DOMAIN, "invalid character "
167                                     "'%c' in name"), what);
168                                 break;

170                         case NAME_ERR_MULTIPLE_AT:
171                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
172                                     "multiple '@' delimiters in name"));
173                                 break;

175                         case NAME_ERR_NOLETTER:
176                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
177                                     "pool doesn't begin with a letter"));
178                                 break;

180                         case NAME_ERR_RESERVED:
181                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
182                                     "name is reserved"));
183                                 break;

185                         case NAME_ERR_DISKLIKE:
186                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
187                                     "reserved disk name"));
188                                 break;
189                         }
190                 }
```



```
192                 return (0);
193         }

195         if (!(type & ZFS_TYPE_SNAPSHOT) && strchr(path, '@') != NULL) {
196                 if (hdl != NULL)
197                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
198                             "snapshot delimiter '@' in filesystem name"));
199                 return (0);
200         }

202         if (type == ZFS_TYPE_SNAPSHOT && strchr(path, '@') == NULL) {
203                 if (hdl != NULL)
204                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
205                             "missing '@' delimiter in snapshot name"));
206                 return (0);
207         }

209         if (modifying && strchr(path, '%') != NULL) {
210                 if (hdl != NULL)
211                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
212                             "invalid character %c in name"), '%');
213                 return (0);
214         }

216         return (-1);
217 }

219 int
220 zfs_name_valid(const char *name, zfs_type_t type)
221 {
222         if (type == ZFS_TYPE_POOL)
223                 return (zpool_name_valid(NULL, B_FALSE, name));
224         return (zfs_validate_name(NULL, name, type, B_FALSE));
225 }

227 /*
228  * This function takes the raw DSL properties, and filters out the user-defined
229  * properties into a separate nvlist.
230  */
231 static nvlist_t *
232 process_user_props(zfs_handle_t *zhp, nvlist_t *props)
233 {
234         libzfs_handle_t *hdl = zhp->zfs_hdl;
235         nvpair_t *elem;
236         nvlist_t *propval;
237         nvlist_t *nvl;

239         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
240                 (void) no_memory(hdl);
241                 return (NULL);
242         }

244         elem = NULL;
245         while ((elem = nvlist_next_nvpair(props, elem)) != NULL) {
246                 if (!zfs_prop_user(nvpair_name(elem)))
247                         continue;

249                 verify(nvpair_value_nvlist(elem, &propval) == 0);
250                 if (nvlist_add_nvlist(nvl, nvpair_name(elem), propval) != 0) {
251                         nvlist_free(nvl);
252                         (void) no_memory(hdl);
253                         return (NULL);
254                 }
255         }
```

```
257         return (nvl);
258 }

260 static zpool_handle_t *
261 zpool_add_handle(zfs_handle_t *zhp, const char *pool_name)
262 {
263         libzfs_handle_t *hdl = zhp->zfs_hdl;
264         zpool_handle_t *zph;

266         if ((zph = zpool_open_canfail(hdl, pool_name)) != NULL) {
267                 if (hdl->libzfs_pool_handles != NULL)
268                         zph->zpool_next = hdl->libzfs_pool_handles;
269                 hdl->libzfs_pool_handles = zph;
270         }
271         return (zph);
272 }

274 static zpool_handle_t *
275 zpool_find_handle(zfs_handle_t *zhp, const char *pool_name, int len)
276 {
277         libzfs_handle_t *hdl = zhp->zfs_hdl;
278         zpool_handle_t *zph = hdl->libzfs_pool_handles;

280         while ((zph != NULL) &&
281             (strncmp(pool_name, zpool_get_name(zph), len) != 0))
282                 zph = zph->zpool_next;
283         return (zph);
284 }

286 /*
287  * Returns a handle to the pool that contains the provided dataset.
288  * If a handle to that pool already exists then that handle is returned.
289  * Otherwise, a new handle is created and added to the list of handles.
290  */
291 static zpool_handle_t *
292 zpool_handle(zfs_handle_t *zhp)
293 {
294         char *pool_name;
295         int len;
296         zpool_handle_t *zph;

298         len = strcspn(zhp->zfs_name, "/@") + 1;
299         pool_name = zfs_alloc(zhp->zfs_hdl, len);
300         (void) strlcpy(pool_name, zhp->zfs_name, len);

302         zph = zpool_find_handle(zhp, pool_name, len);
303         if (zph == NULL)
304                 zph = zpool_add_handle(zhp, pool_name);

306         free(pool_name);
307         return (zph);
308 }

310 void
311 zpool_free_handles(libzfs_handle_t *hdl)
312 {
313         zpool_handle_t *next, *zph = hdl->libzfs_pool_handles;

315         while (zph != NULL) {
316                 next = zph->zpool_next;
317                 zpool_close(zph);
318                 zph = next;
319         }
320         hdl->libzfs_pool_handles = NULL;
321 }
```



```
323 /*
324  * Utility function to gather stats (objset and zpl) for the given object.
325  */
326 static int
327 get_stats_ioctl(zfs_handle_t *zhp, zfs_cmd_t *zc)
328 {
329         libzfs_handle_t *hdl = zhp->zfs_hdl;

331         (void) strlcpy(zc->zc_name, zhp->zfs_name, sizeof (zc->zc_name));

333         while (ioctl(hdl->libzfs_fd, ZFS_IOC_OBJSET_STATS, zc) != 0) {
334                 if (errno == ENOMEM) {
335                         if (zcmd_expand_dst_nvlist(hdl, zc) != 0) {
336                                 return (-1);
337                         }
338                 } else {
339                         return (-1);
340                 }
341         }
342         return (0);
343 }

345 /*
346  * Utility function to get the received properties of the given object.
347  */
348 static int
349 get_recvd_props_ioctl(zfs_handle_t *zhp)
350 {
351         libzfs_handle_t *hdl = zhp->zfs_hdl;
352         nvlist_t *recvdprops;
353         zfs_cmd_t zc = { 0 };
354         int err;

356         if (zcmd_alloc_dst_nvlist(hdl, &zc, 0) != 0)
357                 return (-1);

359         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

361         while (ioctl(hdl->libzfs_fd, ZFS_IOC_OBJSET_RECVD_PROPS, &zc) != 0) {
362                 if (errno == ENOMEM) {
363                         if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
364                                 return (-1);
365                         }
366                 } else {
367                         zcmd_free_nvlists(&zc);
368                         return (-1);
369                 }
370         }

372         err = zcmd_read_dst_nvlist(zhp->zfs_hdl, &zc, &recvdprops);
373         zcmd_free_nvlists(&zc);
374         if (err != 0)
375                 return (-1);

377         nvlist_free(zhp->zfs_recvd_props);
378         zhp->zfs_recvd_props = recvdprops;

380         return (0);
381 }

383 static int
384 put_stats_zhdl(zfs_handle_t *zhp, zfs_cmd_t *zc)
385 {
386         nvlist_t *allprops, *userprops;

388         zhp->zfs_dmustats = zc->zc_objset_stats; /* structure assignment */
```

```
390         if (zcmd_read_dst_nvlist(zhp->zfs_hdl, zc, &allprops) != 0) {
391                 return (-1);
392         }
393
394         /*
395          * XXX Why do we store the user props separately, in addition to
396          * storing them in zfs_props?
397          */
398         if ((userprops = process_user_props(zhp, allprops)) == NULL) {
399                 nvlist_free(allprops);
400                 return (-1);
401         }
402
403         nvlist_free(zhp->zfs_props);
404         nvlist_free(zhp->zfs_user_props);
405
406         zhp->zfs_props = allprops;
407         zhp->zfs_user_props = userprops;
408
409         return (0);
410 }
411
412 static int
413 get_stats(zfs_handle_t *zhp)
414 {
415         int rc = 0;
416         zfs_cmd_t zc = { 0 };
417
418         if (zcmd_alloc_dst_nvlist(zhp->zfs_hdl, &zc, 0) != 0)
419                 return (-1);
420         if (get_stats_ioctl(zhp, &zc) != 0)
421                 rc = -1;
422         else if (put_stats_zhdl(zhp, &zc) != 0)
423                 rc = -1;
424         zcmd_free_nvlists(&zc);
425         return (rc);
426 }
427
428 /*
429  * Refresh the properties currently stored in the handle.
430  */
431 void
432 zfs_refresh_properties(zfs_handle_t *zhp)
433 {
434         (void) get_stats(zhp);
435 }
436
437 /*
438  * Makes a handle from the given dataset name.  Used by zfs_open() and
439  * zfs_iter_* to create child handles on the fly.
440  */
441 static int
442 make_dataset_handle_common(zfs_handle_t *zhp, zfs_cmd_t *zc)
443 {
444         if (put_stats_zhdl(zhp, zc) != 0)
445                 return (-1);
446
447         /*
448          * We've managed to open the dataset and gather statistics.  Determine
449          * the high-level type.
450          */
451         if (zhp->zfs_dmustats.dds_type == DMU_OST_ZVOL)
452                 zhp->zfs_head_type = ZFS_TYPE_VOLUME;
453         else if (zhp->zfs_dmustats.dds_type == DMU_OST_ZFS)
454                 zhp->zfs_head_type = ZFS_TYPE_FILESYSTEM;
```



```
455         else
456                 abort();
457
458         if (zhp->zfs_dmustats.dds_is_snapshot)
459                 zhp->zfs_type = ZFS_TYPE_SNAPSHOT;
460         else if (zhp->zfs_dmustats.dds_type == DMU_OST_ZVOL)
461                 zhp->zfs_type = ZFS_TYPE_VOLUME;
462         else if (zhp->zfs_dmustats.dds_type == DMU_OST_ZFS)
463                 zhp->zfs_type = ZFS_TYPE_FILESYSTEM;
464         else
465                 abort();        /* we should never see any other types */
466
467         if ((zhp->zpool_hdl = zpool_handle(zhp)) == NULL)
468                 return (-1);
469
470         return (0);
471 }
472
473 zfs_handle_t *
474 make_dataset_handle(libzfs_handle_t *hdl, const char *path)
475 {
476         zfs_cmd_t zc = { 0 };
477
478         zfs_handle_t *zhp = calloc(sizeof (zfs_handle_t), 1);
479
480         if (zhp == NULL)
481                 return (NULL);
482
483         zhp->zfs_hdl = hdl;
484         (void) strlcpy(zhp->zfs_name, path, sizeof (zhp->zfs_name));
485         if (zcmd_alloc_dst_nvlist(hdl, &zc, 0) != 0) {
486                 free(zhp);
487                 return (NULL);
488         }
489         if (get_stats_ioctl(zhp, &zc) == -1) {
490                 zcmd_free_nvlists(&zc);
491                 free(zhp);
492                 return (NULL);
493         }
494         if (make_dataset_handle_common(zhp, &zc) == -1) {
495                 free(zhp);
496                 zhp = NULL;
497         }
498         zcmd_free_nvlists(&zc);
499         return (zhp);
500 }
501
502 zfs_handle_t *
503 make_dataset_handle_zc(libzfs_handle_t *hdl, zfs_cmd_t *zc)
504 {
505         zfs_handle_t *zhp = calloc(sizeof (zfs_handle_t), 1);
506
507         if (zhp == NULL)
508                 return (NULL);
509
510         zhp->zfs_hdl = hdl;
511         (void) strlcpy(zhp->zfs_name, zc->zc_name, sizeof (zhp->zfs_name));
512         if (make_dataset_handle_common(zhp, zc) == -1) {
513                 free(zhp);
514                 return (NULL);
515         }
516         return (zhp);
517 }
518
519 zfs_handle_t *
520 zfs_handle_dup(zfs_handle_t *zhp_orig)
```

```
{
        zfs_handle_t *zhp = calloc(sizeof (zfs_handle_t), 1);

        if (zhp == NULL)
                return (NULL);

        zhp->zfs_hdl = zhp_orig->zfs_hdl;
        zhp->zpool_hdl = zhp_orig->zpool_hdl;
        (void) strlcpy(zhp->zfs_name, zhp_orig->zfs_name,
            sizeof (zhp->zfs_name));
        zhp->zfs_type = zhp_orig->zfs_type;
        zhp->zfs_head_type = zhp_orig->zfs_head_type;
        zhp->zfs_dmustats = zhp_orig->zfs_dmustats;
        if (zhp_orig->zfs_props != NULL) {
                if (nvlist_dup(zhp_orig->zfs_props, &zhp->zfs_props, 0) != 0) {
                        (void) no_memory(zhp->zfs_hdl);
                        zfs_close(zhp);
                        return (NULL);
                }
        }
        if (zhp_orig->zfs_user_props != NULL) {
                if (nvlist_dup(zhp_orig->zfs_user_props,
                    &zhp->zfs_user_props, 0) != 0) {
                        (void) no_memory(zhp->zfs_hdl);
                        zfs_close(zhp);
                        return (NULL);
                }
        }
        if (zhp_orig->zfs_recvd_props != NULL) {
                if (nvlist_dup(zhp_orig->zfs_recvd_props,
                    &zhp->zfs_recvd_props, 0)) {
                        (void) no_memory(zhp->zfs_hdl);
                        zfs_close(zhp);
                        return (NULL);
                }
        }
        zhp->zfs_mntcheck = zhp_orig->zfs_mntcheck;
        if (zhp_orig->zfs_mntopts != NULL) {
                zhp->zfs_mntopts = zfs_strdup(zhp_orig->zfs_hdl,
                    zhp_orig->zfs_mntopts);
        }
        zhp->zfs_props_table = zhp_orig->zfs_props_table;
        return (zhp);
}

/*
 * Opens the given snapshot, filesystem, or volume.   The 'types'
 * argument is a mask of acceptable types.  The function will print an
 * appropriate error message and return NULL if it can't be opened.
 */
zfs_handle_t *
zfs_open(libzfs_handle_t *hdl, const char *path, int types)
{
        zfs_handle_t *zhp;
        char errbuf[1024];

        (void) snprintf(errbuf, sizeof (errbuf),
            dgettext(TEXT_DOMAIN, "cannot open '%s'"), path);

        /*
         * Validate the name before we even try to open it.
         */
        if (!zfs_validate_name(hdl, path, ZFS_TYPE_DATASET, B_FALSE)) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                    "invalid dataset name"));
                (void) zfs_error(hdl, EZFS_INVALIDNAME, errbuf);
```



```
                return (NULL);
        }

        /*
         * Try to get stats for the dataset, which will tell us if it exists.
         */
        errno = 0;
        if ((zhp = make_dataset_handle(hdl, path)) == NULL) {
                (void) zfs_standard_error(hdl, errno, errbuf);
                return (NULL);
        }

        if (!(types & zhp->zfs_type)) {
                (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
                zfs_close(zhp);
                return (NULL);
        }

        return (zhp);
}

/*
 * Release a ZFS handle.  Nothing to do but free the associated memory.
 */
void
zfs_close(zfs_handle_t *zhp)
{
        if (zhp->zfs_mntopts)
                free(zhp->zfs_mntopts);
        nvlist_free(zhp->zfs_props);
        nvlist_free(zhp->zfs_user_props);
        nvlist_free(zhp->zfs_recvd_props);
        free(zhp);
}

typedef struct mnttab_node {
        struct mnttab mtn_mt;
        avl_node_t mtn_node;
} mnttab_node_t;

static int
libzfs_mnttab_cache_compare(const void *arg1, const void *arg2)
{
        const mnttab_node_t *mtn1 = arg1;
        const mnttab_node_t *mtn2 = arg2;
        int rv;

        rv = strcmp(mtn1->mtn_mt.mnt_special, mtn2->mtn_mt.mnt_special);

        if (rv == 0)
                return (0);
        return (rv > 0 ? 1 : -1);
}

void
libzfs_mnttab_init(libzfs_handle_t *hdl)
{
        assert(avl_numnodes(&hdl->libzfs_mnttab_cache) == 0);
        avl_create(&hdl->libzfs_mnttab_cache, libzfs_mnttab_cache_compare,
            sizeof (mnttab_node_t), offsetof(mnttab_node_t, mtn_node));
}

void
libzfs_mnttab_update(libzfs_handle_t *hdl)
{
        struct mnttab entry;
```

```
654         rewind(hdl->libzfs_mnttab);
655         while (getmntent(hdl->libzfs_mnttab, &entry) == 0) {
656                 mnttab_node_t *mtn;
657
658                 if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0)
659                         continue;
660                 mtn = zfs_alloc(hdl, sizeof (mnttab_node_t));
661                 mtn->mtn_mt.mnt_special = zfs_strdup(hdl, entry.mnt_special);
662                 mtn->mtn_mt.mnt_mountp = zfs_strdup(hdl, entry.mnt_mountp);
663                 mtn->mtn_mt.mnt_fstype = zfs_strdup(hdl, entry.mnt_fstype);
664                 mtn->mtn_mt.mnt_mntopts = zfs_strdup(hdl, entry.mnt_mntopts);
665                 avl_add(&hdl->libzfs_mnttab_cache, mtn);
666         }
667 }
668
669 void
670 libzfs_mnttab_fini(libzfs_handle_t *hdl)
671 {
672         void *cookie = NULL;
673         mnttab_node_t *mtn;
674
675         while (mtn = avl_destroy_nodes(&hdl->libzfs_mnttab_cache, &cookie)) {
676                 free(mtn->mtn_mt.mnt_special);
677                 free(mtn->mtn_mt.mnt_mountp);
678                 free(mtn->mtn_mt.mnt_fstype);
679                 free(mtn->mtn_mt.mnt_mntopts);
680                 free(mtn);
681         }
682         avl_destroy(&hdl->libzfs_mnttab_cache);
683 }
684
685 void
686 libzfs_mnttab_cache(libzfs_handle_t *hdl, boolean_t enable)
687 {
688         hdl->libzfs_mnttab_enable = enable;
689 }
690
691 int
692 libzfs_mnttab_find(libzfs_handle_t *hdl, const char *fsname,
693     struct mnttab *entry)
694 {
695         mnttab_node_t find;
696         mnttab_node_t *mtn;
697
698         if (!hdl->libzfs_mnttab_enable) {
699                 struct mnttab srch = { 0 };
700
701                 if (avl_numnodes(&hdl->libzfs_mnttab_cache))
702                         libzfs_mnttab_fini(hdl);
703                 rewind(hdl->libzfs_mnttab);
704                 srch.mnt_special = (char *)fsname;
705                 srch.mnt_fstype = MNTTYPE_ZFS;
706                 if (getmntany(hdl->libzfs_mnttab, entry, &srch) == 0)
707                         return (0);
708                 else
709                         return (ENOENT);
710         }
711
712         if (avl_numnodes(&hdl->libzfs_mnttab_cache) == 0)
713                 libzfs_mnttab_update(hdl);
714
715         find.mtn_mt.mnt_special = (char *)fsname;
716         mtn = avl_find(&hdl->libzfs_mnttab_cache, &find, NULL);
717         if (mtn) {
718                 *entry = mtn->mtn_mt;
```



```
719                 return (0);
720         }
721         return (ENOENT);
722 }
723
724 void
725 libzfs_mnttab_add(libzfs_handle_t *hdl, const char *special,
726     const char *mountp, const char *mntopts)
727 {
728         mnttab_node_t *mtn;
729
730         if (avl_numnodes(&hdl->libzfs_mnttab_cache) == 0)
731                 return;
732         mtn = zfs_alloc(hdl, sizeof (mnttab_node_t));
733         mtn->mtn_mt.mnt_special = zfs_strdup(hdl, special);
734         mtn->mtn_mt.mnt_mountp = zfs_strdup(hdl, mountp);
735         mtn->mtn_mt.mnt_fstype = zfs_strdup(hdl, MNTTYPE_ZFS);
736         mtn->mtn_mt.mnt_mntopts = zfs_strdup(hdl, mntopts);
737         avl_add(&hdl->libzfs_mnttab_cache, mtn);
738 }
739
740 void
741 libzfs_mnttab_remove(libzfs_handle_t *hdl, const char *fsname)
742 {
743         mnttab_node_t find;
744         mnttab_node_t *ret;
745
746         find.mtn_mt.mnt_special = (char *)fsname;
747         if (ret = avl_find(&hdl->libzfs_mnttab_cache, (void *)&find, NULL)) {
748                 avl_remove(&hdl->libzfs_mnttab_cache, ret);
749                 free(ret->mtn_mt.mnt_special);
750                 free(ret->mtn_mt.mnt_mountp);
751                 free(ret->mtn_mt.mnt_fstype);
752                 free(ret->mtn_mt.mnt_mntopts);
753                 free(ret);
754         }
755 }
756
757 int
758 zfs_spa_version(zfs_handle_t *zhp, int *spa_version)
759 {
760         zpool_handle_t *zpool_handle = zhp->zpool_hdl;
761
762         if (zpool_handle == NULL)
763                 return (-1);
764
765         *spa_version = zpool_get_prop_int(zpool_handle,
766             ZPOOL_PROP_VERSION, NULL);
767         return (0);
768 }
769
770 /*
771  * The choice of reservation property depends on the SPA version.
772  */
773 static int
774 zfs_which_resv_prop(zfs_handle_t *zhp, zfs_prop_t *resv_prop)
775 {
776         int spa_version;
777
778         if (zfs_spa_version(zhp, &spa_version) < 0)
779                 return (-1);
780
781         if (spa_version >= SPA_VERSION_REFRESERVATION)
782                 *resv_prop = ZFS_PROP_REFRESERVATION;
783         else
784                 *resv_prop = ZFS_PROP_RESERVATION;
```

```
786         return (0);
787 }

789 /*
790  * Given an nvlist of properties to set, validates that they are correct, and
791  * parses any numeric properties (index, boolean, etc) if they are specified as
792  * strings.
793  */
794 nvlist_t *
795 zfs_valid_proplist(libzfs_handle_t *hdl, zfs_type_t type, nvlist_t *nvl,
796     uint64_t zoned, zfs_handle_t *zhp, const char *errbuf)
797 {
798         nvpair_t *elem;
799         uint64_t intval;
800         char *strval;
801         zfs_prop_t prop;
802         nvlist_t *ret;
803         int chosen_normal = -1;
804         int chosen_utf = -1;

806         if (nvlist_alloc(&ret, NV_UNIQUE_NAME, 0) != 0) {
807                 (void) no_memory(hdl);
808                 return (NULL);
809         }

811         /*
812          * Make sure this property is valid and applies to this type.
813          */

815         elem = NULL;
816         while ((elem = nvlist_next_nvpair(nvl, elem)) != NULL) {
817                 const char *propname = nvpair_name(elem);

819                 prop = zfs_name_to_prop(propname);
820                 if (prop == ZPROP_INVAL && zfs_prop_user(propname)) {
821                         /*
822                          * This is a user property: make sure it's a
823                          * string, and that it's less than ZAP_MAXNAMELEN.
824                          */
825                         if (nvpair_type(elem) != DATA_TYPE_STRING) {
826                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
827                                     "'%s' must be a string"), propname);
828                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
829                                 goto error;
830                         }

832                         if (strlen(nvpair_name(elem)) >= ZAP_MAXNAMELEN) {
833                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
834                                     "property name '%s' is too long"),
835                                     propname);
836                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
837                                 goto error;
838                         }

840                         (void) nvpair_value_string(elem, &strval);
841                         if (nvlist_add_string(ret, propname, strval) != 0) {
842                                 (void) no_memory(hdl);
843                                 goto error;
844                         }
845                         continue;
846                 }

848                 /*
849                  * Currently, only user properties can be modified on
850                  * snapshots.
```



```
851                  */
852                 if (type == ZFS_TYPE_SNAPSHOT) {
853                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
854                             "this property can not be modified for snapshots"));
855                         (void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
856                         goto error;
857                 }

859                 if (prop == ZPROP_INVAL && zfs_prop_userquota(propname)) {
860                         zfs_userquota_prop_t uqtype;
861                         char newpropname[128];
862                         char domain[128];
863                         uint64_t rid;
864                         uint64_t valary[3];

866                         if (userquota_propname_decode(propname, zoned,
867                             &uqtype, domain, sizeof (domain), &rid) != 0) {
868                                 zfs_error_aux(hdl,
869                                     dgettext(TEXT_DOMAIN,
870                                     "'%s' has an invalid user/group name"),
871                                     propname);
872                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
873                                 goto error;
874                         }

876                         if (uqtype != ZFS_PROP_USERQUOTA &&
877                             uqtype != ZFS_PROP_GROUPQUOTA) {
878                                 zfs_error_aux(hdl,
879                                     dgettext(TEXT_DOMAIN, "'%s' is readonly"),
880                                     propname);
881                                 (void) zfs_error(hdl, EZFS_PROPREADONLY,
882                                     errbuf);
883                                 goto error;
884                         }

886                         if (nvpair_type(elem) == DATA_TYPE_STRING) {
887                                 (void) nvpair_value_string(elem, &strval);
888                                 if (strcmp(strval, "none") == 0) {
889                                         intval = 0;
890                                 } else if (zfs_nicestrtonum(hdl,
891                                     strval, &intval) != 0) {
892                                         (void) zfs_error(hdl,
893                                             EZFS_BADPROP, errbuf);
894                                         goto error;
895                                 }
896                         } else if (nvpair_type(elem) ==
897                             DATA_TYPE_UINT64) {
898                                 (void) nvpair_value_uint64(elem, &intval);
899                                 if (intval == 0) {
900                                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
901                                             "use 'none' to disable "
902                                             "userquota/groupquota"));
903                                         goto error;
904                                 }
905                         } else {
906                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
907                                     "'%s' must be a number"), propname);
908                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
909                                 goto error;
910                         }

912                         /*
913                          * Encode the prop name as
914                          * userquota@<hex-rid>-domain, to make it easy
915                          * for the kernel to decode.
916                          */
```

```
917                     (void) snprintf(newpropname, sizeof (newpropname),
918                         "%s%llx-%s", zfs_userquota_prop_prefixes[uqtype],
919                         (longlong_t)rid, domain);
920                     valary[0] = uqtype;
921                     valary[1] = rid;
922                     valary[2] = intval;
923                     if (nvlist_add_uint64_array(ret, newpropname,
924                         valary, 3) != 0) {
925                             (void) no_memory(hdl);
926                             goto error;
927                     }
928                     continue;
929             } else if (prop == ZPROP_INVAL && zfs_prop_written(propname)) {
930                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
931                         "'%s' is readonly"),
932                         propname);
933                     (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
934                     goto error;
935             }
936
937             if (prop == ZPROP_INVAL) {
938                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
939                         "invalid property '%s'"), propname);
940                     (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
941                     goto error;
942             }
943
944             if (!zfs_prop_valid_for_type(prop, type)) {
945                     zfs_error_aux(hdl,
946                         dgettext(TEXT_DOMAIN, "'%s' does not "
947                         "apply to datasets of this type"), propname);
948                     (void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
949                     goto error;
950             }
951
952             if (zfs_prop_readonly(prop) &&
953                 (!zfs_prop_setonce(prop) || zhp != NULL)) {
954                     zfs_error_aux(hdl,
955                         dgettext(TEXT_DOMAIN, "'%s' is readonly"),
956                         propname);
957                     (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
958                     goto error;
959             }
960
961             if (zprop_parse_value(hdl, elem, prop, type, ret,
962                 &strval, &intval, errbuf) != 0)
963                     goto error;
964
965             /*
966              * Perform some additional checks for specific properties.
967              */
968             switch (prop) {
969             case ZFS_PROP_VERSION:
970             {
971                     int version;
972
973                     if (zhp == NULL)
974                             break;
975                     version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
976                     if (intval < version) {
977                             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
978                                 "Can not downgrade; already at version %u"),
979                                 version);
980                             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
981                             goto error;
982                     }
```



```
983                     break;
984             }
985
986             case ZFS_PROP_RECORDSIZE:
987             case ZFS_PROP_VOLBLOCKSIZE:
988                     /* must be power of two within SPA_{MIN,MAX}BLOCKSIZE */
989                     if (intval < SPA_MINBLOCKSIZE ||
990                         intval > SPA_MAXBLOCKSIZE || !ISP2(intval)) {
991                             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
992                                 "'%s' must be power of 2 from %u "
993                                 "to %uk"), propname,
994                                 (uint_t)SPA_MINBLOCKSIZE,
995                                 (uint_t)SPA_MAXBLOCKSIZE >> 10);
996                             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
997                             goto error;
998                     }
999                     break;
1000
1001            case ZFS_PROP_MLSLABEL:
1002            {
1003                    /*
1004                     * Verify the mlslabel string and convert to
1005                     * internal hex label string.
1006                     */
1007
1008                    m_label_t *new_sl;
1009                    char *hex = NULL;       /* internal label string */
1010
1011                    /* Default value is already OK. */
1012                    if (strcasecmp(strval, ZFS_MLSLABEL_DEFAULT) == 0)
1013                            break;
1014
1015                    /* Verify the label can be converted to binary form */
1016                    if (((new_sl = m_label_alloc(MAC_LABEL)) == NULL) ||
1017                        (str_to_label(strval, &new_sl, MAC_LABEL,
1018                        L_NO_CORRECTION, NULL) == -1)) {
1019                            goto badlabel;
1020                    }
1021
1022                    /* Now translate to hex internal label string */
1023                    if (label_to_str(new_sl, &hex, M_INTERNAL,
1024                        DEF_NAMES) != 0) {
1025                            if (hex)
1026                                    free(hex);
1027                            goto badlabel;
1028                    }
1029                    m_label_free(new_sl);
1030
1031                    /* If string is already in internal form, we're done. */
1032                    if (strcmp(strval, hex) == 0) {
1033                            free(hex);
1034                            break;
1035                    }
1036
1037                    /* Replace the label string with the internal form. */
1038                    (void) nvlist_remove(ret, zfs_prop_to_name(prop),
1039                        DATA_TYPE_STRING);
1040                    verify(nvlist_add_string(ret, zfs_prop_to_name(prop),
1041                        hex) == 0);
1042                    free(hex);
1043
1044                    break;
1045
1046    badlabel:
1047                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1048                        "invalid mlslabel '%s'"), strval);
```

```
1049                    (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1050                    m_label_free(new_sl);   /* OK if null */
1051                    goto error;

1053            }

1055            case ZFS_PROP_MOUNTPOINT:
1056            {
1057                    namecheck_err_t why;

1059                    if (strcmp(strval, ZFS_MOUNTPOINT_NONE) == 0 ||
1060                        strcmp(strval, ZFS_MOUNTPOINT_LEGACY) == 0)
1061                            break;

1063                    if (mountpoint_namecheck(strval, &why)) {
1064                            switch (why) {
1065                            case NAME_ERR_LEADING_SLASH:
1066                                    zfs_error_aux(hdl,
1067                                        dgettext(TEXT_DOMAIN,
1068                                        "'%s' must be an absolute path, "
1069                                        "'none', or 'legacy'"), propname);
1070                                    break;
1071                            case NAME_ERR_TOOLONG:
1072                                    zfs_error_aux(hdl,
1073                                        dgettext(TEXT_DOMAIN,
1074                                        "component of '%s' is too long"),
1075                                        propname);
1076                                    break;
1077                            }
1078                            (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1079                            goto error;
1080                    }
1081            }

1083                    /*FALLTHRU*/

1085            case ZFS_PROP_SHARESMB:
1086            case ZFS_PROP_SHARENFS:
1087                    /*
1088                     * For the mountpoint and sharenfs or sharesmb
1089                     * properties, check if it can be set in a
1090                     * global/non-global zone based on
1091                     * the zoned property value:
1092                     *
1093                     *              global zone         non-global zone
1094                     * ------------------------------------------------------
1095                     * zoned=on     mountpoint (no)     mountpoint (yes)
1096                     *              sharenfs (no)       sharenfs (no)
1097                     *              sharesmb (no)       sharesmb (no)
1098                     *
1099                     * zoned=off    mountpoint (yes)        N/A
1100                     *              sharenfs (yes)
1101                     *              sharesmb (yes)
1102                     */
1103                    if (zoned) {
1104                            if (getzoneid() == GLOBAL_ZONEID) {
1105                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1106                                        "'%s' cannot be set on "
1107                                        "dataset in a non-global zone"),
1108                                        propname);
1109                                    (void) zfs_error(hdl, EZFS_ZONED,
1110                                        errbuf);
1111                                    goto error;
1112                            } else if (prop == ZFS_PROP_SHARENFS ||
1113                                prop == ZFS_PROP_SHARESMB) {
1114                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
```



```
1115                                        "'%s' cannot be set in "
1116                                        "a non-global zone"), propname);
1117                                    (void) zfs_error(hdl, EZFS_ZONED,
1118                                        errbuf);
1119                                    goto error;
1120                            }
1121                    } else if (getzoneid() != GLOBAL_ZONEID) {
1122                            /*
1123                             * If zoned property is 'off', this must be in
1124                             * a global zone. If not, something is wrong.
1125                             */
1126                            zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1127                                "'%s' cannot be set while dataset "
1128                                "'zoned' property is set"), propname);
1129                            (void) zfs_error(hdl, EZFS_ZONED, errbuf);
1130                            goto error;
1131                    }

1133                    /*
1134                     * At this point, it is legitimate to set the
1135                     * property. Now we want to make sure that the
1136                     * property value is valid if it is sharenfs.
1137                     */
1138                    if ((prop == ZFS_PROP_SHARENFS ||
1139                         prop == ZFS_PROP_SHARESMB) &&
1140                        strcmp(strval, "on") != 0 &&
1141                        strcmp(strval, "off") != 0) {
1142                            zfs_share_proto_t proto;

1144                            if (prop == ZFS_PROP_SHARESMB)
1145                                    proto = PROTO_SMB;
1146                            else
1147                                    proto = PROTO_NFS;

1149                            /*
1150                             * Must be an valid sharing protocol
1151                             * option string so init the libshare
1152                             * in order to enable the parser and
1153                             * then parse the options. We use the
1154                             * control API since we don't care about
1155                             * the current configuration and don't
1156                             * want the overhead of loading it
1157                             * until we actually do something.
1158                             */

1160                            if (zfs_init_libshare(hdl,
1161                                SA_INIT_CONTROL_API) != SA_OK) {
1162                                    /*
1163                                     * An error occurred so we can't do
1164                                     * anything
1165                                     */
1166                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1167                                        "'%s' cannot be set: problem "
1168                                        "in share initialization"),
1169                                        propname);
1170                                    (void) zfs_error(hdl, EZFS_BADPROP,
1171                                        errbuf);
1172                                    goto error;
1173                            }

1175                            if (zfs_parse_options(strval, proto) != SA_OK) {
1176                                    /*
1177                                     * There was an error in parsing so
1178                                     * deal with it by issuing an error
1179                                     * message and leaving after
1180                                     * uninitializing the the libshare
```

```
1181                                     * interface.
1182                                     */
1183                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1184                                        "'%s' cannot be set to invalid "
1185                                        "options"), propname);
1186                                    (void) zfs_error(hdl, EZFS_BADPROP,
1187                                        errbuf);
1188                                    zfs_uninit_libshare(hdl);
1189                                    goto error;
1190                            }
1191                            zfs_uninit_libshare(hdl);
1192                    }

1194                    break;
1195            case ZFS_PROP_UTF8ONLY:
1196                    chosen_utf = (int)intval;
1197                    break;
1198            case ZFS_PROP_NORMALIZE:
1199                    chosen_normal = (int)intval;
1200                    break;
1201            }

1203            /*
1204             * For changes to existing volumes, we have some additional
1205             * checks to enforce.
1206             */
1207            if (type == ZFS_TYPE_VOLUME && zhp != NULL) {
1208                    uint64_t volsize = zfs_prop_get_int(zhp,
1209                        ZFS_PROP_VOLSIZE);
1210                    uint64_t blocksize = zfs_prop_get_int(zhp,
1211                        ZFS_PROP_VOLBLOCKSIZE);
1212                    char buf[64];

1214                    switch (prop) {
1215                    case ZFS_PROP_RESERVATION:
1216                    case ZFS_PROP_REFRESERVATION:
1217                            if (intval > volsize) {
1218                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1219                                        "'%s' is greater than current "
1220                                        "volume size"), propname);
1221                                    (void) zfs_error(hdl, EZFS_BADPROP,
1222                                        errbuf);
1223                                    goto error;
1224                            }
1225                            break;

1227                    case ZFS_PROP_VOLSIZE:
1228                            if (intval % blocksize != 0) {
1229                                    zfs_nicenum(blocksize, buf,
1230                                        sizeof (buf));
1231                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1232                                        "'%s' must be a multiple of "
1233                                        "volume block size (%s)"),
1234                                        propname, buf);
1235                                    (void) zfs_error(hdl, EZFS_BADPROP,
1236                                        errbuf);
1237                                    goto error;
1238                            }

1240                            if (intval == 0) {
1241                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1242                                        "'%s' cannot be zero"),
1243                                        propname);
1244                                    (void) zfs_error(hdl, EZFS_BADPROP,
1245                                        errbuf);
1246                                    goto error;
```



```
1247                            }
1248                            break;
1249                    }
1250            }
1251    }

1253    /*
1254     * If normalization was chosen, but no UTF8 choice was made,
1255     * enforce rejection of non-UTF8 names.
1256     *
1257     * If normalization was chosen, but rejecting non-UTF8 names
1258     * was explicitly not chosen, it is an error.
1259     */
1260    if (chosen_normal > 0 && chosen_utf < 0) {
1261            if (nvlist_add_uint64(ret,
1262                zfs_prop_to_name(ZFS_PROP_UTF8ONLY), 1) != 0) {
1263                    (void) no_memory(hdl);
1264                    goto error;
1265            }
1266    } else if (chosen_normal > 0 && chosen_utf == 0) {
1267            zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1268                "'%s' must be set 'on' if normalization chosen"),
1269                zfs_prop_to_name(ZFS_PROP_UTF8ONLY));
1270            (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1271            goto error;
1272    }
1273    return (ret);

1275 error:
1276    nvlist_free(ret);
1277    return (NULL);
1278 }

1280 int
1281 zfs_add_synthetic_resv(zfs_handle_t *zhp, nvlist_t *nvl)
1282 {
1283    uint64_t old_volsize;
1284    uint64_t new_volsize;
1285    uint64_t old_reservation;
1286    uint64_t new_reservation;
1287    zfs_prop_t resv_prop;
1288    nvlist_t *props;

1290    /*
1291     * If this is an existing volume, and someone is setting the volsize,
1292     * make sure that it matches the reservation, or add it if necessary.
1293     */
1294    old_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
1295    if (zfs_which_resv_prop(zhp, &resv_prop) < 0)
1296            return (-1);
1297    old_reservation = zfs_prop_get_int(zhp, resv_prop);

1299    props = fnvlist_alloc();
1300    fnvlist_add_uint64(props, zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
1301        zfs_prop_get_int(zhp, ZFS_PROP_VOLBLOCKSIZE));

1303    if ((zvol_volsize_to_reservation(old_volsize, props) !=
1304        old_reservation) || nvlist_exists(nvl,
1305        zfs_prop_to_name(resv_prop))) {
1306            fnvlist_free(props);
1307            return (0);
1308    }
1309    if (nvlist_lookup_uint64(nvl, zfs_prop_to_name(ZFS_PROP_VOLSIZE),
1310        &new_volsize) != 0) {
1311            fnvlist_free(props);
1312            return (-1);
```

```
1313         }
1314         new_reservation = zvol_volsize_to_reservation(new_volsize, props);
1315         fnvlist_free(props);

1317         if (nvlist_add_uint64(nvl, zfs_prop_to_name(resv_prop),
1318             new_reservation) != 0) {
1319                 (void) no_memory(zhp->zfs_hdl);
1320                 return (-1);
1321         }
1322         return (1);
1323 }

1325 void
1326 zfs_setprop_error(libzfs_handle_t *hdl, zfs_prop_t prop, int err,
1327     char *errbuf)
1328 {
1329         switch (err) {

1331         case ENOSPC:
1332                 /*
1333                  * For quotas and reservations, ENOSPC indicates
1334                  * something different; setting a quota or reservation
1335                  * doesn't use any disk space.
1336                  */
1337                 switch (prop) {
1338                 case ZFS_PROP_QUOTA:
1339                 case ZFS_PROP_REFQUOTA:
1340                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1341                             "size is less than current used or "
1342                             "reserved space"));
1343                         (void) zfs_error(hdl, EZFS_PROPSPACE, errbuf);
1344                         break;

1346                 case ZFS_PROP_RESERVATION:
1347                 case ZFS_PROP_REFRESERVATION:
1348                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1349                             "size is greater than available space"));
1350                         (void) zfs_error(hdl, EZFS_PROPSPACE, errbuf);
1351                         break;

1353                 default:
1354                         (void) zfs_standard_error(hdl, err, errbuf);
1355                         break;
1356                 }
1357                 break;

1359         case EBUSY:
1360                 (void) zfs_standard_error(hdl, EBUSY, errbuf);
1361                 break;

1363         case EROFS:
1364                 (void) zfs_error(hdl, EZFS_DSREADONLY, errbuf);
1365                 break;

1367         case ENOTSUP:
1368                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1369                     "pool and or dataset must be upgraded to set this "
1370                     "property or value"));
1371                 (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
1372                 break;

1374         case ERANGE:
1375                 if (prop == ZFS_PROP_COMPRESSION) {
1376                         (void) zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1377                             "property setting is not allowed on "
1378                             "bootable datasets"));
```



```
1379                         (void) zfs_error(hdl, EZFS_NOTSUP, errbuf);
1380                 } else {
1381                         (void) zfs_standard_error(hdl, err, errbuf);
1382                 }
1383                 break;

1385         case EINVAL:
1386                 if (prop == ZPROP_INVAL) {
1387                         (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1388                 } else {
1389                         (void) zfs_standard_error(hdl, err, errbuf);
1390                 }
1391                 break;

1393         case EOVERFLOW:
1394                 /*
1395                  * This platform can't address a volume this big.
1396                  */
1397 #ifdef _ILP32
1398                 if (prop == ZFS_PROP_VOLSIZE) {
1399                         (void) zfs_error(hdl, EZFS_VOLTOOBIG, errbuf);
1400                         break;
1401                 }
1402 #endif
1403                 /* FALLTHROUGH */
1404         default:
1405                 (void) zfs_standard_error(hdl, err, errbuf);
1406         }
1407 }

1409 /*
1410  * Given a property name and value, set the property for the given dataset.
1411  */
1412 int
1413 zfs_prop_set(zfs_handle_t *zhp, const char *propname, const char *propval)
1414 {
1415         zfs_cmd_t zc = { 0 };
1416         int ret = -1;
1417         prop_changelist_t *cl = NULL;
1418         char errbuf[1024];
1419         libzfs_handle_t *hdl = zhp->zfs_hdl;
1420         nvlist_t *nvl = NULL, *realprops;
1421         zfs_prop_t prop;
1422         boolean_t do_prefix = B_TRUE;
1423         int added_resv;

1425         (void) snprintf(errbuf, sizeof (errbuf),
1426             dgettext(TEXT_DOMAIN, "cannot set property for '%s'"),
1427             zhp->zfs_name);

1429         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0 ||
1430             nvlist_add_string(nvl, propname, propval) != 0) {
1431                 (void) no_memory(hdl);
1432                 goto error;
1433         }

1435         if ((realprops = zfs_valid_proplist(hdl, zhp->zfs_type, nvl,
1436             zfs_prop_get_int(zhp, ZFS_PROP_ZONED), zhp, errbuf)) == NULL)
1437                 goto error;

1439         nvlist_free(nvl);
1440         nvl = realprops;

1442         prop = zfs_name_to_prop(propname);

1444         if (prop == ZFS_PROP_VOLSIZE) {
```

```
1445                 if ((added_resv = zfs_add_synthetic_resv(zhp, nvl)) == -1)
1446                         goto error;
1447         }
1448
1449         if ((cl = changelist_gather(zhp, prop, 0, 0)) == NULL)
1450                 goto error;
1451
1452         if (prop == ZFS_PROP_MOUNTPOINT && changelist_haszonedchild(cl)) {
1453                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1454                     "child dataset with inherited mountpoint is used "
1455                     "in a non-global zone"));
1456                 ret = zfs_error(hdl, EZFS_ZONED, errbuf);
1457                 goto error;
1458         }
1459
1460         /*
1461          * We don't want to unmount & remount the dataset when changing
1462          * its canmount property to 'on' or 'noauto'.  We only use
1463          * the changelist logic to unmount when setting canmount=off.
1464          */
1465         if (prop == ZFS_PROP_CANMOUNT) {
1466                 uint64_t idx;
1467                 int err = zprop_string_to_index(prop, propval, &idx,
1468                     ZFS_TYPE_DATASET);
1469                 if (err == 0 && idx != ZFS_CANMOUNT_OFF)
1470                         do_prefix = B_FALSE;
1471         }
1472
1473         if (do_prefix && (ret = changelist_prefix(cl)) != 0)
1474                 goto error;
1475
1476         /*
1477          * Execute the corresponding ioctl() to set this property.
1478          */
1479         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
1480
1481         if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1482                 goto error;
1483
1484         ret = zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
1485
1486         if (ret != 0) {
1487                 zfs_setprop_error(hdl, prop, errno, errbuf);
1488                 if (added_resv && errno == ENOSPC) {
1489                         /* clean up the volsize property we tried to set */
1490                         uint64_t old_volsize = zfs_prop_get_int(zhp,
1491                             ZFS_PROP_VOLSIZE);
1492                         nvlist_free(nvl);
1493                         zcmd_free_nvlists(&zc);
1494                         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
1495                                 goto error;
1496                         if (nvlist_add_uint64(nvl,
1497                             zfs_prop_to_name(ZFS_PROP_VOLSIZE),
1498                             old_volsize) != 0)
1499                                 goto error;
1500                         if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1501                                 goto error;
1502                         (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
1503                 }
1504         } else {
1505                 if (do_prefix)
1506                         ret = changelist_postfix(cl);
1507
1508                 /*
1509                  * Refresh the statistics so the new property value
1510                  * is reflected.
```



```
1511                  */
1512                 if (ret == 0)
1513                         (void) get_stats(zhp);
1514         }
1515
1516 error:
1517         nvlist_free(nvl);
1518         zcmd_free_nvlists(&zc);
1519         if (cl)
1520                 changelist_free(cl);
1521         return (ret);
1522 }
1523
1524 /*
1525  * Given a property, inherit the value from the parent dataset, or if received
1526  * is TRUE, revert to the received value, if any.
1527  */
1528 int
1529 zfs_prop_inherit(zfs_handle_t *zhp, const char *propname, boolean_t received)
1530 {
1531         zfs_cmd_t zc = { 0 };
1532         int ret;
1533         prop_changelist_t *cl;
1534         libzfs_handle_t *hdl = zhp->zfs_hdl;
1535         char errbuf[1024];
1536         zfs_prop_t prop;
1537
1538         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
1539             "cannot inherit %s for '%s'"), propname, zhp->zfs_name);
1540
1541         zc.zc_cookie = received;
1542         if ((prop = zfs_name_to_prop(propname)) == ZPROP_INVAL) {
1543                 /*
1544                  * For user properties, the amount of work we have to do is very
1545                  * small, so just do it here.
1546                  */
1547                 if (!zfs_prop_user(propname)) {
1548                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1549                             "invalid property"));
1550                         return (zfs_error(hdl, EZFS_BADPROP, errbuf));
1551                 }
1552
1553                 (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
1554                 (void) strlcpy(zc.zc_value, propname, sizeof (zc.zc_value));
1555
1556                 if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_INHERIT_PROP, &zc) != 0)
1557                         return (zfs_standard_error(hdl, errno, errbuf));
1558
1559                 return (0);
1560         }
1561
1562         /*
1563          * Verify that this property is inheritable.
1564          */
1565         if (zfs_prop_readonly(prop))
1566                 return (zfs_error(hdl, EZFS_PROPREADONLY, errbuf));
1567
1568         if (!zfs_prop_inheritable(prop) && !received)
1569                 return (zfs_error(hdl, EZFS_PROPNONINHERIT, errbuf));
1570
1571         /*
1572          * Check to see if the value applies to this type
1573          */
1574         if (!zfs_prop_valid_for_type(prop, zhp->zfs_type))
1575                 return (zfs_error(hdl, EZFS_PROPTYPE, errbuf));
```

```
1577         /*
1578          * Normalize the name, to get rid of shorthand abbreviations.
1579          */
1580         propname = zfs_prop_to_name(prop);
1581         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
1582         (void) strlcpy(zc.zc_value, propname, sizeof (zc.zc_value));

1584         if (prop == ZFS_PROP_MOUNTPOINT && getzoneid() == GLOBAL_ZONEID &&
1585             zfs_prop_get_int(zhp, ZFS_PROP_ZONED)) {
1586                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1587                     "dataset is used in a non-global zone"));
1588                 return (zfs_error(hdl, EZFS_ZONED, errbuf));
1589         }

1591         /*
1592          * Determine datasets which will be affected by this change, if any.
1593          */
1594         if ((cl = changelist_gather(zhp, prop, 0, 0)) == NULL)
1595                 return (-1);

1597         if (prop == ZFS_PROP_MOUNTPOINT && changelist_haszonedchild(cl)) {
1598                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1599                     "child dataset with inherited mountpoint is used "
1600                     "in a non-global zone"));
1601                 ret = zfs_error(hdl, EZFS_ZONED, errbuf);
1602                 goto error;
1603         }

1605         if ((ret = changelist_prefix(cl)) != 0)
1606                 goto error;

1608         if ((ret = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_INHERIT_PROP, &zc)) != 0) {
1609                 return (zfs_standard_error(hdl, errno, errbuf));
1610         } else {

1612                 if ((ret = changelist_postfix(cl)) != 0)
1613                         goto error;

1615                 /*
1616                  * Refresh the statistics so the new property is reflected.
1617                  */
1618                 (void) get_stats(zhp);
1619         }

1621 error:
1622         changelist_free(cl);
1623         return (ret);
1624 }

1626 /*
1627  * True DSL properties are stored in an nvlist.  The following two functions
1628  * extract them appropriately.
1629  */
1630 static uint64_t
1631 getprop_uint64(zfs_handle_t *zhp, zfs_prop_t prop, char **source)
1632 {
1633         nvlist_t *nv;
1634         uint64_t value;

1636         *source = NULL;
1637         if (nvlist_lookup_nvlist(zhp->zfs_props,
1638             zfs_prop_to_name(prop), &nv) == 0) {
1639                 verify(nvlist_lookup_uint64(nv, ZPROP_VALUE, &value) == 0);
1640                 (void) nvlist_lookup_string(nv, ZPROP_SOURCE, source);
1641         } else {
1642                 verify(!zhp->zfs_props_table ||
```



```
1643                     zhp->zfs_props_table[prop] == B_TRUE);
1644                 value = zfs_prop_default_numeric(prop);
1645                 *source = "";
1646         }

1648         return (value);
1649 }

1651 static char *
1652 getprop_string(zfs_handle_t *zhp, zfs_prop_t prop, char **source)
1653 {
1654         nvlist_t *nv;
1655         char *value;

1657         *source = NULL;
1658         if (nvlist_lookup_nvlist(zhp->zfs_props,
1659             zfs_prop_to_name(prop), &nv) == 0) {
1660                 verify(nvlist_lookup_string(nv, ZPROP_VALUE, &value) == 0);
1661                 (void) nvlist_lookup_string(nv, ZPROP_SOURCE, source);
1662         } else {
1663                 verify(!zhp->zfs_props_table ||
1664                     zhp->zfs_props_table[prop] == B_TRUE);
1665                 if ((value = (char *)zfs_prop_default_string(prop)) == NULL)
1666                         value = "";
1667                 *source = "";
1668         }

1670         return (value);
1671 }

1673 static boolean_t
1674 zfs_is_recvd_props_mode(zfs_handle_t *zhp)
1675 {
1676         return (zhp->zfs_props == zhp->zfs_recvd_props);
1677 }

1679 static void
1680 zfs_set_recvd_props_mode(zfs_handle_t *zhp, uint64_t *cookie)
1681 {
1682         *cookie = (uint64_t)(uintptr_t)zhp->zfs_props;
1683         zhp->zfs_props = zhp->zfs_recvd_props;
1684 }

1686 static void
1687 zfs_unset_recvd_props_mode(zfs_handle_t *zhp, uint64_t *cookie)
1688 {
1689         zhp->zfs_props = (nvlist_t *)(uintptr_t)*cookie;
1690         *cookie = 0;
1691 }

1693 /*
1694  * Internal function for getting a numeric property.  Both zfs_prop_get() and
1695  * zfs_prop_get_int() are built using this interface.
1696  *
1697  * Certain properties can be overridden using 'mount -o'.  In this case, scan
1698  * the contents of the /etc/mnttab entry, searching for the appropriate options.
1699  * If they differ from the on-disk values, report the current values and mark
1700  * the source "temporary".
1701  */
1702 static int
1703 get_numeric_property(zfs_handle_t *zhp, zfs_prop_t prop, zprop_source_t *src,
1704     char **source, uint64_t *val)
1705 {
1706         zfs_cmd_t zc = { 0 };
1707         nvlist_t *zplprops = NULL;
1708         struct mnttab mnt;
```

```
1709         char *mntopt_on = NULL;
1710         char *mntopt_off = NULL;
1711         boolean_t received = zfs_is_recvd_props_mode(zhp);

1713         *source = NULL;

1715         switch (prop) {
1716         case ZFS_PROP_ATIME:
1717                 mntopt_on = MNTOPT_ATIME;
1718                 mntopt_off = MNTOPT_NOATIME;
1719                 break;

1721         case ZFS_PROP_DEVICES:
1722                 mntopt_on = MNTOPT_DEVICES;
1723                 mntopt_off = MNTOPT_NODEVICES;
1724                 break;

1726         case ZFS_PROP_EXEC:
1727                 mntopt_on = MNTOPT_EXEC;
1728                 mntopt_off = MNTOPT_NOEXEC;
1729                 break;

1731         case ZFS_PROP_READONLY:
1732                 mntopt_on = MNTOPT_RO;
1733                 mntopt_off = MNTOPT_RW;
1734                 break;

1736         case ZFS_PROP_SETUID:
1737                 mntopt_on = MNTOPT_SETUID;
1738                 mntopt_off = MNTOPT_NOSETUID;
1739                 break;

1741         case ZFS_PROP_XATTR:
1742                 mntopt_on = MNTOPT_XATTR;
1743                 mntopt_off = MNTOPT_NOXATTR;
1744                 break;

1746         case ZFS_PROP_NBMAND:
1747                 mntopt_on = MNTOPT_NBMAND;
1748                 mntopt_off = MNTOPT_NONBMAND;
1749                 break;
1750         }

1752         /*
1753          * Because looking up the mount options is potentially expensive
1754          * (iterating over all of /etc/mnttab), we defer its calculation until
1755          * we're looking up a property which requires its presence.
1756          */
1757         if (!zhp->zfs_mntcheck &&
1758             (mntopt_on != NULL || prop == ZFS_PROP_MOUNTED)) {
1759                 libzfs_handle_t *hdl = zhp->zfs_hdl;
1760                 struct mnttab entry;

1762                 if (libzfs_mnttab_find(hdl, zhp->zfs_name, &entry) == 0) {
1763                         zhp->zfs_mntopts = zfs_strdup(hdl,
1764                             entry.mnt_mntopts);
1765                         if (zhp->zfs_mntopts == NULL)
1766                                 return (-1);
1767                 }

1769                 zhp->zfs_mntcheck = B_TRUE;
1770         }

1772         if (zhp->zfs_mntopts == NULL)
1773                 mnt.mnt_mntopts = "";
1774         else
```



```
1775                 mnt.mnt_mntopts = zhp->zfs_mntopts;

1777         switch (prop) {
1778         case ZFS_PROP_ATIME:
1779         case ZFS_PROP_DEVICES:
1780         case ZFS_PROP_EXEC:
1781         case ZFS_PROP_READONLY:
1782         case ZFS_PROP_SETUID:
1783         case ZFS_PROP_XATTR:
1784         case ZFS_PROP_NBMAND:
1785                 *val = getprop_uint64(zhp, prop, source);

1787                 if (received)
1788                         break;

1790                 if (hasmntopt(&mnt, mntopt_on) && !*val) {
1791                         *val = B_TRUE;
1792                         if (src)
1793                                 *src = ZPROP_SRC_TEMPORARY;
1794                 } else if (hasmntopt(&mnt, mntopt_off) && *val) {
1795                         *val = B_FALSE;
1796                         if (src)
1797                                 *src = ZPROP_SRC_TEMPORARY;
1798                 }
1799                 break;

1801         case ZFS_PROP_CANMOUNT:
1802         case ZFS_PROP_VOLSIZE:
1803         case ZFS_PROP_QUOTA:
1804         case ZFS_PROP_REFQUOTA:
1805         case ZFS_PROP_RESERVATION:
1806         case ZFS_PROP_REFRESERVATION:
1807                 *val = getprop_uint64(zhp, prop, source);

1809                 if (*source == NULL) {
1810                         /* not default, must be local */
1811                         *source = zhp->zfs_name;
1812                 }
1813                 break;

1815         case ZFS_PROP_MOUNTED:
1816                 *val = (zhp->zfs_mntopts != NULL);
1817                 break;

1819         case ZFS_PROP_NUMCLONES:
1820                 *val = zhp->zfs_dmustats.dds_num_clones;
1821                 break;

1823         case ZFS_PROP_VERSION:
1824         case ZFS_PROP_NORMALIZE:
1825         case ZFS_PROP_UTF8ONLY:
1826         case ZFS_PROP_CASE:
1827                 if (!zfs_prop_valid_for_type(prop, zhp->zfs_head_type) ||
1828                     zcmd_alloc_dst_nvlist(zhp->zfs_hdl, &zc, 0) != 0)
1829                         return (-1);
1830                 (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
1831                 if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_OBJSET_ZPLPROPS, &zc)) {
1832                         zcmd_free_nvlists(&zc);
1833                         return (-1);
1834                 }
1835                 if (zcmd_read_dst_nvlist(zhp->zfs_hdl, &zc, &zplprops) != 0 ||
1836                     nvlist_lookup_uint64(zplprops, zfs_prop_to_name(prop),
1837                     val) != 0) {
1838                         zcmd_free_nvlists(&zc);
1839                         return (-1);
1840                 }
```

```
1841                 if (zplprops)
1842                         nvlist_free(zplprops);
1843                 zcmd_free_nvlists(&zc);
1844                 break;

1846         default:
1847                 switch (zfs_prop_get_type(prop)) {
1848                 case PROP_TYPE_NUMBER:
1849                 case PROP_TYPE_INDEX:
1850                         *val = getprop_uint64(zhp, prop, source);
1851                         /*
1852                          * If we tried to use a default value for a
1853                          * readonly property, it means that it was not
1854                          * present.
1855                          */
1856                         if (zfs_prop_readonly(prop) &&
1857                             *source != NULL && (*source)[0] == '\0') {
1858                                 *source = NULL;
1859                         }
1860                         break;

1862                 case PROP_TYPE_STRING:
1863                 default:
1864                         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
1865                             "cannot get non-numeric property"));
1866                         return (zfs_error(zhp->zfs_hdl, EZFS_BADPROP,
1867                             dgettext(TEXT_DOMAIN, "internal error")));
1868                 }
1869         }

1871         return (0);
1872 }

1874 /*
1875  * Calculate the source type, given the raw source string.
1876  */
1877 static void
1878 get_source(zfs_handle_t *zhp, zprop_source_t *srctype, char *source,
1879     char *statbuf, size_t statlen)
1880 {
1881         if (statbuf == NULL || *srctype == ZPROP_SRC_TEMPORARY)
1882                 return;

1884         if (source == NULL) {
1885                 *srctype = ZPROP_SRC_NONE;
1886         } else if (source[0] == '\0') {
1887                 *srctype = ZPROP_SRC_DEFAULT;
1888         } else if (strstr(source, ZPROP_SOURCE_VAL_RECVD) != NULL) {
1889                 *srctype = ZPROP_SRC_RECEIVED;
1890         } else {
1891                 if (strcmp(source, zhp->zfs_name) == 0) {
1892                         *srctype = ZPROP_SRC_LOCAL;
1893                 } else {
1894                         (void) strlcpy(statbuf, source, statlen);
1895                         *srctype = ZPROP_SRC_INHERITED;
1896                 }
1897         }

1899 }

1901 int
1902 zfs_prop_get_recvd(zfs_handle_t *zhp, const char *propname, char *propbuf,
1903     size_t proplen, boolean_t literal)
1904 {
1905         zfs_prop_t prop;
1906         int err = 0;
```



```
1908         if (zhp->zfs_recvd_props == NULL)
1909                 if (get_recvd_props_ioctl(zhp) != 0)
1910                         return (-1);

1912         prop = zfs_name_to_prop(propname);

1914         if (prop != ZPROP_INVAL) {
1915                 uint64_t cookie;
1916                 if (!nvlist_exists(zhp->zfs_recvd_props, propname))
1917                         return (-1);
1918                 zfs_set_recvd_props_mode(zhp, &cookie);
1919                 err = zfs_prop_get(zhp, prop, propbuf, proplen,
1920                     NULL, NULL, 0, literal);
1921                 zfs_unset_recvd_props_mode(zhp, &cookie);
1922         } else {
1923                 nvlist_t *propval;
1924                 char *recvdval;
1925                 if (nvlist_lookup_nvlist(zhp->zfs_recvd_props,
1926                     propname, &propval) != 0)
1927                         return (-1);
1928                 verify(nvlist_lookup_string(propval, ZPROP_VALUE,
1929                     &recvdval) == 0);
1930                 (void) strlcpy(propbuf, recvdval, proplen);
1931         }

1933         return (err == 0 ? 0 : -1);
1934 }

1936 static int
1937 get_clones_string(zfs_handle_t *zhp, char *propbuf, size_t proplen)
1938 {
1939         nvlist_t *value;
1940         nvpair_t *pair;

1942         value = zfs_get_clones_nvl(zhp);
1943         if (value == NULL)
1944                 return (-1);

1946         propbuf[0] = '\0';
1947         for (pair = nvlist_next_nvpair(value, NULL); pair != NULL;
1948             pair = nvlist_next_nvpair(value, pair)) {
1949                 if (propbuf[0] != '\0')
1950                         (void) strlcat(propbuf, ",", proplen);
1951                 (void) strlcat(propbuf, nvpair_name(pair), proplen);
1952         }

1954         return (0);
1955 }

1957 struct get_clones_arg {
1958         uint64_t numclones;
1959         nvlist_t *value;
1960         const char *origin;
1961         char buf[ZFS_MAXNAMELEN];
1962 };

1964 int
1965 get_clones_cb(zfs_handle_t *zhp, void *arg)
1966 {
1967         struct get_clones_arg *gca = arg;

1969         if (gca->numclones == 0) {
1970                 zfs_close(zhp);
1971                 return (0);
1972         }
```

```
1974         if (zfs_prop_get(zhp, ZFS_PROP_ORIGIN, gca->buf, sizeof (gca->buf),
1975             NULL, NULL, 0, B_TRUE) != 0)
1976                 goto out;
1977         if (strcmp(gca->buf, gca->origin) == 0) {
1978                 fnvlist_add_boolean(gca->value, zfs_get_name(zhp));
1979                 gca->numclones--;
1980         }

1982 out:
1983         (void) zfs_iter_children(zhp, get_clones_cb, gca);
1984         zfs_close(zhp);
1985         return (0);
1986 }

1988 nvlist_t *
1989 zfs_get_clones_nvl(zfs_handle_t *zhp)
1990 {
1991         nvlist_t *nv, *value;

1993         if (nvlist_lookup_nvlist(zhp->zfs_props,
1994             zfs_prop_to_name(ZFS_PROP_CLONES), &nv) != 0) {
1995                 struct get_clones_arg gca;

1997                 /*
1998                  * if this is a snapshot, then the kernel wasn't able
1999                  * to get the clones.  Do it by slowly iterating.
2000                  */
2001                 if (zhp->zfs_type != ZFS_TYPE_SNAPSHOT)
2002                         return (NULL);
2003                 if (nvlist_alloc(&nv, NV_UNIQUE_NAME, 0) != 0)
2004                         return (NULL);
2005                 if (nvlist_alloc(&value, NV_UNIQUE_NAME, 0) != 0) {
2006                         nvlist_free(nv);
2007                         return (NULL);
2008                 }

2010                 gca.numclones = zfs_prop_get_int(zhp, ZFS_PROP_NUMCLONES);
2011                 gca.value = value;
2012                 gca.origin = zhp->zfs_name;

2014                 if (gca.numclones != 0) {
2015                         zfs_handle_t *root;
2016                         char pool[ZFS_MAXNAMELEN];
2017                         char *cp = pool;

2019                         /* get the pool name */
2020                         (void) strlcpy(pool, zhp->zfs_name, sizeof (pool));
2021                         (void) strsep(&cp, "/@");
2022                         root = zfs_open(zhp->zfs_hdl, pool,
2023                             ZFS_TYPE_FILESYSTEM);

2025                         (void) get_clones_cb(root, &gca);
2026                 }

2028                 if (gca.numclones != 0 ||
2029                     nvlist_add_nvlist(nv, ZPROP_VALUE, value) != 0 ||
2030                     nvlist_add_nvlist(zhp->zfs_props,
2031                     zfs_prop_to_name(ZFS_PROP_CLONES), nv) != 0) {
2032                         nvlist_free(nv);
2033                         nvlist_free(value);
2034                         return (NULL);
2035                 }
2036                 nvlist_free(nv);
2037                 nvlist_free(value);
2038                 verify(0 == nvlist_lookup_nvlist(zhp->zfs_props,
```



```
2039                     zfs_prop_to_name(ZFS_PROP_CLONES), &nv));
2040         }

2042         verify(nvlist_lookup_nvlist(nv, ZPROP_VALUE, &value) == 0);

2044         return (value);
2045 }

2047 /*
2048  * Retrieve a property from the given object.  If 'literal' is specified, then
2049  * numbers are left as exact values.  Otherwise, numbers are converted to a
2050  * human-readable form.
2051  *
2052  * Returns 0 on success, or -1 on error.
2053  */
2054 int
2055 zfs_prop_get(zfs_handle_t *zhp, zfs_prop_t prop, char *propbuf, size_t proplen,
2056     zprop_source_t *src, char *statbuf, size_t statlen, boolean_t literal)
2057 {
2058         char *source = NULL;
2059         uint64_t val;
2060         char *str;
2061         const char *strval;
2062         boolean_t received = zfs_is_recvd_props_mode(zhp);

2064         /*
2065          * Check to see if this property applies to our object
2066          */
2067         if (!zfs_prop_valid_for_type(prop, zhp->zfs_type))
2068                 return (-1);

2070         if (received && zfs_prop_readonly(prop))
2071                 return (-1);

2073         if (src)
2074                 *src = ZPROP_SRC_NONE;

2076         switch (prop) {
2077         case ZFS_PROP_CREATION:
2078                 /*
2079                  * 'creation' is a time_t stored in the statistics.  We convert
2080                  * this into a string unless 'literal' is specified.
2081                  */
2082                 {
2083                         val = getprop_uint64(zhp, prop, &source);
2084                         time_t time = (time_t)val;
2085                         struct tm t;

2087                         if (literal ||
2088                             localtime_r(&time, &t) == NULL ||
2089                             strftime(propbuf, proplen, "%a %b %e %k:%M %Y",
2090                             &t) == 0)
2091                                 (void) snprintf(propbuf, proplen, "%llu", val);
2092                 }
2093                 break;

2095         case ZFS_PROP_MOUNTPOINT:
2096                 /*
2097                  * Getting the precise mountpoint can be tricky.
2098                  *
2099                  *  - for 'none' or 'legacy', return those values.
2100                  *  - for inherited mountpoints, we want to take everything
2101                  *    after our ancestor and append it to the inherited value.
2102                  *
2103                  * If the pool has an alternate root, we want to prepend that
2104                  * root to any values we return.
```

```
2105                 */
2107                str = getprop_string(zhp, prop, &source);
2109                if (str[0] == '/') {
2110                        char buf[MAXPATHLEN];
2111                        char *root = buf;
2112                        const char *relpath;
2114                        /*
2115                         * If we inherit the mountpoint, even from a dataset
2116                         * with a received value, the source will be the path of
2117                         * the dataset we inherit from. If source is
2118                         * ZPROP_SOURCE_VAL_RECVD, the received value is not
2119                         * inherited.
2120                         */
2121                        if (strcmp(source, ZPROP_SOURCE_VAL_RECVD) == 0) {
2122                                relpath = "";
2123                        } else {
2124                                relpath = zhp->zfs_name + strlen(source);
2125                                if (relpath[0] == '/')
2126                                        relpath++;
2127                        }
2129                        if ((zpool_get_prop(zhp->zpool_hdl,
2130                            ZPOOL_PROP_ALTROOT, buf, MAXPATHLEN, NULL)) ||
2131                            (strcmp(root, "-") == 0))
2132                                root[0] = '\0';
2133                        /*
2134                         * Special case an alternate root of '/'. This will
2135                         * avoid having multiple leading slashes in the
2136                         * mountpoint path.
2137                         */
2138                        if (strcmp(root, "/") == 0)
2139                                root++;
2141                        /*
2142                         * If the mountpoint is '/' then skip over this
2143                         * if we are obtaining either an alternate root or
2144                         * an inherited mountpoint.
2145                         */
2146                        if (str[1] == '\0' && (root[0] != '\0' ||
2147                            relpath[0] != '\0'))
2148                                str++;
2150                        if (relpath[0] == '\0')
2151                                (void) snprintf(propbuf, proplen, "%s%s",
2152                                    root, str);
2153                        else
2154                                (void) snprintf(propbuf, proplen, "%s%s%s%s",
2155                                    root, str, relpath[0] == '@' ? "" : "/",
2156                                    relpath);
2157                } else {
2158                        /* 'legacy' or 'none' */
2159                        (void) strlcpy(propbuf, str, proplen);
2160                }
2162                break;
2164        case ZFS_PROP_ORIGIN:
2165                (void) strlcpy(propbuf, getprop_string(zhp, prop, &source),
2166                    proplen);
2167                /*
2168                 * If there is no parent at all, return failure to indicate that
2169                 * it doesn't apply to this dataset.
2170                 */
```



```
2171                if (propbuf[0] == '\0')
2172                        return (-1);
2173                break;
2175        case ZFS_PROP_CLONES:
2176                if (get_clones_string(zhp, propbuf, proplen) != 0)
2177                        return (-1);
2178                break;
2180        case ZFS_PROP_QUOTA:
2181        case ZFS_PROP_REFQUOTA:
2182        case ZFS_PROP_RESERVATION:
2183        case ZFS_PROP_REFRESERVATION:
2185                if (get_numeric_property(zhp, prop, src, &source, &val) != 0)
2186                        return (-1);
2188                /*
2189                 * If quota or reservation is 0, we translate this into 'none'
2190                 * (unless literal is set), and indicate that it's the default
2191                 * value.  Otherwise, we print the number nicely and indicate
2192                 * that its set locally.
2193                 */
2194                if (val == 0) {
2195                        if (literal)
2196                                (void) strlcpy(propbuf, "0", proplen);
2197                        else
2198                                (void) strlcpy(propbuf, "none", proplen);
2199                } else {
2200                        if (literal)
2201                                (void) snprintf(propbuf, proplen, "%llu",
2202                                    (u_longlong_t)val);
2203                        else
2204                                zfs_nicenum(val, propbuf, proplen);
2205                }
2206                break;
2208        case ZFS_PROP_REFRATIO:
2209        case ZFS_PROP_COMPRESSRATIO:
2210                if (get_numeric_property(zhp, prop, src, &source, &val) != 0)
2211                        return (-1);
2212                (void) snprintf(propbuf, proplen, "%llu.%02llux",
2213                    (u_longlong_t)(val / 100),
2214                    (u_longlong_t)(val % 100));
2215                break;
2217        case ZFS_PROP_TYPE:
2218                switch (zhp->zfs_type) {
2219                case ZFS_TYPE_FILESYSTEM:
2220                        str = "filesystem";
2221                        break;
2222                case ZFS_TYPE_VOLUME:
2223                        str = "volume";
2224                        break;
2225                case ZFS_TYPE_SNAPSHOT:
2226                        str = "snapshot";
2227                        break;
2228                default:
2229                        abort();
2230                }
2231                (void) snprintf(propbuf, proplen, "%s", str);
2232                break;
2234        case ZFS_PROP_MOUNTED:
2235                /*
2236                 * The 'mounted' property is a pseudo-property that described
```

```
2237                 * whether the filesystem is currently mounted.  Even though
2238                 * it's a boolean value, the typical values of "on" and "off"
2239                 * don't make sense, so we translate to "yes" and "no".
2240                 */
2241                if (get_numeric_property(zhp, ZFS_PROP_MOUNTED,
2242                    src, &source, &val) != 0)
2243                        return (-1);
2244                if (val)
2245                        (void) strlcpy(propbuf, "yes", proplen);
2246                else
2247                        (void) strlcpy(propbuf, "no", proplen);
2248                break;

2250        case ZFS_PROP_NAME:
2251                /*
2252                 * The 'name' property is a pseudo-property derived from the
2253                 * dataset name.  It is presented as a real property to simplify
2254                 * consumers.
2255                 */
2256                (void) strlcpy(propbuf, zhp->zfs_name, proplen);
2257                break;

2259        case ZFS_PROP_MLSLABEL:
2260                {
2261                        m_label_t *new_sl = NULL;
2262                        char *ascii = NULL;     /* human readable label */

2264                        (void) strlcpy(propbuf,
2265                            getprop_string(zhp, prop, &source), proplen);

2267                        if (literal || (strcasecmp(propbuf,
2268                            ZFS_MLSLABEL_DEFAULT) == 0))
2269                                break;

2271                        /*
2272                         * Try to translate the internal hex string to
2273                         * human-readable output.  If there are any
2274                         * problems just use the hex string.
2275                         */

2277                        if (str_to_label(propbuf, &new_sl, MAC_LABEL,
2278                            L_NO_CORRECTION, NULL) == -1) {
2279                                m_label_free(new_sl);
2280                                break;
2281                        }

2283                        if (label_to_str(new_sl, &ascii, M_LABEL,
2284                            DEF_NAMES) != 0) {
2285                                if (ascii)
2286                                        free(ascii);
2287                                m_label_free(new_sl);
2288                                break;
2289                        }
2290                        m_label_free(new_sl);

2292                        (void) strlcpy(propbuf, ascii, proplen);
2293                        free(ascii);
2294                }
2295                break;

2297        case ZFS_PROP_GUID:
2298                /*
2299                 * GUIDs are stored as numbers, but they are identifiers.
2300                 * We don't want them to be pretty printed, because pretty
2301                 * printing mangles the ID into a truncated and useless value.
2302                 */
```



```
2303                if (get_numeric_property(zhp, prop, src, &source, &val) != 0)
2304                        return (-1);
2305                (void) snprintf(propbuf, proplen, "%llu", (u_longlong_t)val);
2306                break;

2308        default:
2309                switch (zfs_prop_get_type(prop)) {
2310                case PROP_TYPE_NUMBER:
2311                        if (get_numeric_property(zhp, prop, src,
2312                            &source, &val) != 0)
2313                                return (-1);
2314                        if (literal)
2315                                (void) snprintf(propbuf, proplen, "%llu",
2316                                    (u_longlong_t)val);
2317                        else
2318                                zfs_nicenum(val, propbuf, proplen);
2319                        break;

2321                case PROP_TYPE_STRING:
2322                        (void) strlcpy(propbuf,
2323                            getprop_string(zhp, prop, &source), proplen);
2324                        break;

2326                case PROP_TYPE_INDEX:
2327                        if (get_numeric_property(zhp, prop, src,
2328                            &source, &val) != 0)
2329                                return (-1);
2330                        if (zfs_prop_index_to_string(prop, val, &strval) != 0)
2331                                return (-1);
2332                        (void) strlcpy(propbuf, strval, proplen);
2333                        break;

2335                default:
2336                        abort();
2337                }
2338        }

2340        get_source(zhp, src, source, statbuf, statlen);

2342        return (0);
2343 }

2345 /*
2346  * Utility function to get the given numeric property.  Does no validation that
2347  * the given property is the appropriate type; should only be used with
2348  * hard-coded property types.
2349  */
2350 uint64_t
2351 zfs_prop_get_int(zfs_handle_t *zhp, zfs_prop_t prop)
2352 {
2353        char *source;
2354        uint64_t val;

2356        (void) get_numeric_property(zhp, prop, NULL, &source, &val);

2358        return (val);
2359 }

2361 int
2362 zfs_prop_set_int(zfs_handle_t *zhp, zfs_prop_t prop, uint64_t val)
2363 {
2364        char buf[64];

2366        (void) snprintf(buf, sizeof (buf), "%llu", (longlong_t)val);
2367        return (zfs_prop_set(zhp, zfs_prop_to_name(prop), buf));
2368 }
```

```
/*
 * Similar to zfs_prop_get(), but returns the value as an integer.
 */
int
zfs_prop_get_numeric(zfs_handle_t *zhp, zfs_prop_t prop, uint64_t *value,
    zprop_source_t *src, char *statbuf, size_t statlen)
{
        char *source;

        /*
         * Check to see if this property applies to our object
         */
        if (!zfs_prop_valid_for_type(prop, zhp->zfs_type)) {
                return (zfs_error_fmt(zhp->zfs_hdl, EZFS_PROPTYPE,
                    dgettext(TEXT_DOMAIN, "cannot get property '%s'"),
                    zfs_prop_to_name(prop)));
        }

        if (src)
                *src = ZPROP_SRC_NONE;

        if (get_numeric_property(zhp, prop, src, &source, value) != 0)
                return (-1);

        get_source(zhp, src, source, statbuf, statlen);

        return (0);
}

static int
idmap_id_to_numeric_domain_rid(uid_t id, boolean_t isuser,
    char **domainp, idmap_rid_t *ridp)
{
        idmap_get_handle_t *get_hdl = NULL;
        idmap_stat status;
        int err = EINVAL;

        if (idmap_get_create(&get_hdl) != IDMAP_SUCCESS)
                goto out;

        if (isuser) {
                err = idmap_get_sidbyuid(get_hdl, id,
                    IDMAP_REQ_FLG_USE_CACHE, domainp, ridp, &status);
        } else {
                err = idmap_get_sidbygid(get_hdl, id,
                    IDMAP_REQ_FLG_USE_CACHE, domainp, ridp, &status);
        }
        if (err == IDMAP_SUCCESS &&
            idmap_get_mappings(get_hdl) == IDMAP_SUCCESS &&
            status == IDMAP_SUCCESS)
                err = 0;
        else
                err = EINVAL;
out:
        if (get_hdl)
                idmap_get_destroy(get_hdl);
        return (err);
}

/*
 * convert the propname into parameters needed by kernel
 * Eg: userquota@ahrens -> ZFS_PROP_USERQUOTA, "", 126829
 * Eg: userused@matt@domain -> ZFS_PROP_USERUSED, "S-1-123-456", 789
 */
static int
```



```
userquota_propname_decode(const char *propname, boolean_t zoned,
    zfs_userquota_prop_t *typep, char *domain, int domainlen, uint64_t *ridp)
{
        zfs_userquota_prop_t type;
        char *cp, *end;
        char *numericsid = NULL;
        boolean_t isuser;

        domain[0] = '\0';

        /* Figure out the property type ({user|group}{quota|space}) */
        for (type = 0; type < ZFS_NUM_USERQUOTA_PROPS; type++) {
                if (strncmp(propname, zfs_userquota_prop_prefixes[type],
                    strlen(zfs_userquota_prop_prefixes[type])) == 0)
                        break;
        }
        if (type == ZFS_NUM_USERQUOTA_PROPS)
                return (EINVAL);
        *typep = type;

        isuser = (type == ZFS_PROP_USERQUOTA ||
            type == ZFS_PROP_USERUSED);

        cp = strchr(propname, '@') + 1;

        if (strchr(cp, '@')) {
                /*
                 * It's a SID name (eg "user@domain") that needs to be
                 * turned into S-1-domainID-RID.
                 */
                directory_error_t e;
                if (zoned && getzoneid() == GLOBAL_ZONEID)
                        return (ENOENT);
                if (isuser) {
                        e = directory_sid_from_user_name(NULL,
                            cp, &numericsid);
                } else {
                        e = directory_sid_from_group_name(NULL,
                            cp, &numericsid);
                }
                if (e != NULL) {
                        directory_error_free(e);
                        return (ENOENT);
                }
                if (numericsid == NULL)
                        return (ENOENT);
                cp = numericsid;
                /* will be further decoded below */
        }

        if (strncmp(cp, "S-1-", 4) == 0) {
                /* It's a numeric SID (eg "S-1-234-567-89") */
                (void) strlcpy(domain, cp, domainlen);
                cp = strrchr(domain, '-');
                *cp = '\0';
                cp++;

                errno = 0;
                *ridp = strtoull(cp, &end, 10);
                if (numericsid) {
                        free(numericsid);
                        numericsid = NULL;
                }
                if (errno != 0 || *end != '\0')
                        return (EINVAL);
        } else if (!isdigit(*cp)) {
```

```
2501                 /*
2502                  * It's a user/group name (eg "user") that needs to be
2503                  * turned into a uid/gid
2504                  */
2505                 if (zoned && getzoneid() == GLOBAL_ZONEID)
2506                         return (ENOENT);
2507                 if (isuser) {
2508                         struct passwd *pw;
2509                         pw = getpwnam(cp);
2510                         if (pw == NULL)
2511                                 return (ENOENT);
2512                         *ridp = pw->pw_uid;
2513                 } else {
2514                         struct group *gr;
2515                         gr = getgrnam(cp);
2516                         if (gr == NULL)
2517                                 return (ENOENT);
2518                         *ridp = gr->gr_gid;
2519                 }
2520         } else {
2521                 /* It's a user/group ID (eg "12345"). */
2522                 uid_t id = strtoul(cp, &end, 10);
2523                 idmap_rid_t rid;
2524                 char *mapdomain;

2526                 if (*end != '\0')
2527                         return (EINVAL);
2528                 if (id > MAXUID) {
2529                         /* It's an ephemeral ID. */
2530                         if (idmap_id_to_numeric_domain_rid(id, isuser,
2531                             &mapdomain, &rid) != 0)
2532                                 return (ENOENT);
2533                         (void) strlcpy(domain, mapdomain, domainlen);
2534                         *ridp = rid;
2535                 } else {
2536                         *ridp = id;
2537                 }
2538         }

2540         ASSERT3P(numericsid, ==, NULL);
2541         return (0);
2542 }

2544 static int
2545 zfs_prop_get_userquota_common(zfs_handle_t *zhp, const char *propname,
2546     uint64_t *propvalue, zfs_userquota_prop_t *typep)
2547 {
2548         int err;
2549         zfs_cmd_t zc = { 0 };

2551         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

2553         err = userquota_propname_decode(propname,
2554             zfs_prop_get_int(zhp, ZFS_PROP_ZONED),
2555             typep, zc.zc_value, sizeof (zc.zc_value), &zc.zc_guid);
2556         zc.zc_objset_type = *typep;
2557         if (err)
2558                 return (err);

2560         err = ioctl(zhp->zfs_hdl->libzfs_fd, ZFS_IOC_USERSPACE_ONE, &zc);
2561         if (err)
2562                 return (err);

2564         *propvalue = zc.zc_cookie;
2565         return (0);
2566 }
```



```
2568 int
2569 zfs_prop_get_userquota_int(zfs_handle_t *zhp, const char *propname,
2570     uint64_t *propvalue)
2571 {
2572         zfs_userquota_prop_t type;

2574         return (zfs_prop_get_userquota_common(zhp, propname, propvalue,
2575             &type));
2576 }

2578 int
2579 zfs_prop_get_userquota(zfs_handle_t *zhp, const char *propname,
2580     char *propbuf, int proplen, boolean_t literal)
2581 {
2582         int err;
2583         uint64_t propvalue;
2584         zfs_userquota_prop_t type;

2586         err = zfs_prop_get_userquota_common(zhp, propname, &propvalue,
2587             &type);

2589         if (err)
2590                 return (err);

2592         if (literal) {
2593                 (void) snprintf(propbuf, proplen, "%llu", propvalue);
2594         } else if (propvalue == 0 &&
2595             (type == ZFS_PROP_USERQUOTA || type == ZFS_PROP_GROUPQUOTA)) {
2596                 (void) strlcpy(propbuf, "none", proplen);
2597         } else {
2598                 zfs_nicenum(propvalue, propbuf, proplen);
2599         }
2600         return (0);
2601 }

2603 int
2604 zfs_prop_get_written_int(zfs_handle_t *zhp, const char *propname,
2605     uint64_t *propvalue)
2606 {
2607         int err;
2608         zfs_cmd_t zc = { 0 };
2609         const char *snapname;

2611         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

2613         snapname = strchr(propname, '@') + 1;
2614         if (strchr(snapname, '@')) {
2615                 (void) strlcpy(zc.zc_value, snapname, sizeof (zc.zc_value));
2616         } else {
2617                 /* snapname is the short name, append it to zhp's fsname */
2618                 char *cp;

2620                 (void) strlcpy(zc.zc_value, zhp->zfs_name,
2621                     sizeof (zc.zc_value));
2622                 cp = strchr(zc.zc_value, '@');
2623                 if (cp != NULL)
2624                         *cp = '\0';
2625                 (void) strlcat(zc.zc_value, "@", sizeof (zc.zc_value));
2626                 (void) strlcat(zc.zc_value, snapname, sizeof (zc.zc_value));
2627         }

2629         err = ioctl(zhp->zfs_hdl->libzfs_fd, ZFS_IOC_SPACE_WRITTEN, &zc);
2630         if (err)
2631                 return (err);
```

```
2633         *propvalue = zc.zc_cookie;
2634         return (0);
2635 }

2637 int
2638 zfs_prop_get_written(zfs_handle_t *zhp, const char *propname,
2639     char *propbuf, int proplen, boolean_t literal)
2640 {
2641         int err;
2642         uint64_t propvalue;

2644         err = zfs_prop_get_written_int(zhp, propname, &propvalue);

2646         if (err)
2647                 return (err);

2649         if (literal) {
2650                 (void) snprintf(propbuf, proplen, "%llu", propvalue);
2651         } else {
2652                 zfs_nicenum(propvalue, propbuf, proplen);
2653         }
2654         return (0);
2655 }

2657 /*
2658  * Returns the name of the given zfs handle.
2659  */
2660 const char *
2661 zfs_get_name(const zfs_handle_t *zhp)
2662 {
2663         return (zhp->zfs_name);
2664 }

2666 /*
2667  * Returns the type of the given zfs handle.
2668  */
2669 zfs_type_t
2670 zfs_get_type(const zfs_handle_t *zhp)
2671 {
2672         return (zhp->zfs_type);
2673 }

2675 /*
2676  * Is one dataset name a child dataset of another?
2677  *
2678  * Needs to handle these cases:
2679  * Dataset 1    "a/foo"         "a/foo"         "a/foo"         "a/foo"
2680  * Dataset 2    "a/fo"          "a/foobar"      "a/bar/baz"     "a/foo/bar"
2681  * Descendant?  No.             No.             No.             Yes.
2682  */
2683 static boolean_t
2684 is_descendant(const char *ds1, const char *ds2)
2685 {
2686         size_t d1len = strlen(ds1);

2688         /* ds2 can't be a descendant if it's smaller */
2689         if (strlen(ds2) < d1len)
2690                 return (B_FALSE);

2692         /* otherwise, compare strings and verify that there's a '/' char */
2693         return (ds2[d1len] == '/' && (strncmp(ds1, ds2, d1len) == 0));
2694 }

2696 /*
2697  * Given a complete name, return just the portion that refers to the parent.
2698  * Will return -1 if there is no parent (path is just the name of the
```



```
2699  * pool).
2700  */
2701 static int
2702 parent_name(const char *path, char *buf, size_t buflen)
2703 {
2704         char *slashp;

2706         (void) strlcpy(buf, path, buflen);

2708         if ((slashp = strrchr(buf, '/')) == NULL)
2709                 return (-1);
2710         *slashp = '\0';

2712         return (0);
2713 }

2715 /*
2716  * If accept_ancestor is false, then check to make sure that the given path has
2717  * a parent, and that it exists.  If accept_ancestor is true, then find the
2718  * closest existing ancestor for the given path.  In prefixlen return the
2719  * length of already existing prefix of the given path.  We also fetch the
2720  * 'zoned' property, which is used to validate property settings when creating
2721  * new datasets.
2722  */
2723 static int
2724 check_parents(libzfs_handle_t *hdl, const char *path, uint64_t *zoned,
2725     boolean_t accept_ancestor, int *prefixlen)
2726 {
2727         zfs_cmd_t zc = { 0 };
2728         char parent[ZFS_MAXNAMELEN];
2729         char *slash;
2730         zfs_handle_t *zhp;
2731         char errbuf[1024];
2732         uint64_t is_zoned;

2734         (void) snprintf(errbuf, sizeof (errbuf),
2735             dgettext(TEXT_DOMAIN, "cannot create '%s'"), path);

2737         /* get parent, and check to see if this is just a pool */
2738         if (parent_name(path, parent, sizeof (parent)) != 0) {
2739                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2740                     "missing dataset name"));
2741                 return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2742         }

2744         /* check to see if the pool exists */
2745         if ((slash = strchr(parent, '/')) == NULL)
2746                 slash = parent + strlen(parent);
2747         (void) strncpy(zc.zc_name, parent, slash - parent);
2748         zc.zc_name[slash - parent] = '\0';
2749         if (ioctl(hdl->libzfs_fd, ZFS_IOC_OBJSET_STATS, &zc) != 0 &&
2750             errno == ENOENT) {
2751                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2752                     "no such pool '%s'"), zc.zc_name);
2753                 return (zfs_error(hdl, EZFS_NOENT, errbuf));
2754         }

2756         /* check to see if the parent dataset exists */
2757         while ((zhp = make_dataset_handle(hdl, parent)) == NULL) {
2758                 if (errno == ENOENT && accept_ancestor) {
2759                         /*
2760                          * Go deeper to find an ancestor, give up on top level.
2761                          */
2762                         if (parent_name(parent, parent, sizeof (parent)) != 0) {
2763                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2764                                     "no such pool '%s'"), zc.zc_name);
```

```
2765                                return (zfs_error(hdl, EZFS_NOENT, errbuf));
2766                        }
2767                } else if (errno == ENOENT) {
2768                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2769                            "parent does not exist"));
2770                        return (zfs_error(hdl, EZFS_NOENT, errbuf));
2771                } else
2772                        return (zfs_standard_error(hdl, errno, errbuf));
2773        }

2775        is_zoned = zfs_prop_get_int(zhp, ZFS_PROP_ZONED);
2776        if (zoned != NULL)
2777                *zoned = is_zoned;

2779        /* we are in a non-global zone, but parent is in the global zone */
2780        if (getzoneid() != GLOBAL_ZONEID && !is_zoned) {
2781                (void) zfs_standard_error(hdl, EPERM, errbuf);
2782                zfs_close(zhp);
2783                return (-1);
2784        }

2786        /* make sure parent is a filesystem */
2787        if (zfs_get_type(zhp) != ZFS_TYPE_FILESYSTEM) {
2788                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2789                    "parent is not a filesystem"));
2790                (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
2791                zfs_close(zhp);
2792                return (-1);
2793        }

2795        zfs_close(zhp);
2796        if (prefixlen != NULL)
2797                *prefixlen = strlen(parent);
2798        return (0);
2799 }

2801 /*
2802  * Finds whether the dataset of the given type(s) exists.
2803  */
2804 boolean_t
2805 zfs_dataset_exists(libzfs_handle_t *hdl, const char *path, zfs_type_t types)
2806 {
2807        zfs_handle_t *zhp;

2809        if (!zfs_validate_name(hdl, path, types, B_FALSE))
2810                return (B_FALSE);

2812        /*
2813         * Try to get stats for the dataset, which will tell us if it exists.
2814         */
2815        if ((zhp = make_dataset_handle(hdl, path)) != NULL) {
2816                int ds_type = zhp->zfs_type;

2818                zfs_close(zhp);
2819                if (types & ds_type)
2820                        return (B_TRUE);
2821        }
2822        return (B_FALSE);
2823 }

2825 /*
2826  * Given a path to 'target', create all the ancestors between
2827  * the prefixlen portion of the path, and the target itself.
2828  * Fail if the initial prefixlen-ancestor does not already exist.
2829  */
2830 int
```



```
2831 create_parents(libzfs_handle_t *hdl, char *target, int prefixlen)
2832 {
2833        zfs_handle_t *h;
2834        char *cp;
2835        const char *opname;

2837        /* make sure prefix exists */
2838        cp = target + prefixlen;
2839        if (*cp != '/') {
2840                assert(strchr(cp, '/') == NULL);
2841                h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
2842        } else {
2843                *cp = '\0';
2844                h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
2845                *cp = '/';
2846        }
2847        if (h == NULL)
2848                return (-1);
2849        zfs_close(h);

2851        /*
2852         * Attempt to create, mount, and share any ancestor filesystems,
2853         * up to the prefixlen-long one.
2854         */
2855        for (cp = target + prefixlen + 1;
2856            cp = strchr(cp, '/'); *cp = '/', cp++) {

2858                *cp = '\0';

2860                h = make_dataset_handle(hdl, target);
2861                if (h) {
2862                        /* it already exists, nothing to do here */
2863                        zfs_close(h);
2864                        continue;
2865                }

2867                if (zfs_create(hdl, target, ZFS_TYPE_FILESYSTEM,
2868                    NULL) != 0) {
2869                        opname = dgettext(TEXT_DOMAIN, "create");
2870                        goto ancestorerr;
2871                }

2873                h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
2874                if (h == NULL) {
2875                        opname = dgettext(TEXT_DOMAIN, "open");
2876                        goto ancestorerr;
2877                }

2879                if (zfs_mount(h, NULL, 0) != 0) {
2880                        opname = dgettext(TEXT_DOMAIN, "mount");
2881                        goto ancestorerr;
2882                }

2884                if (zfs_share(h) != 0) {
2885                        opname = dgettext(TEXT_DOMAIN, "share");
2886                        goto ancestorerr;
2887                }

2889                zfs_close(h);
2890        }

2892        return (0);

2894 ancestorerr:
2895        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2896            "failed to %s ancestor '%s'"), opname, target);
```

```
2897         return (-1);
2898 }

2900 /*
2901  * Creates non-existing ancestors of the given path.
2902  */
2903 int
2904 zfs_create_ancestors(libzfs_handle_t *hdl, const char *path)
2905 {
2906         int prefix;
2907         char *path_copy;
2908         int rc;

2910         if (check_parents(hdl, path, NULL, B_TRUE, &prefix) != 0)
2911                 return (-1);

2913         if ((path_copy = strdup(path)) != NULL) {
2914                 rc = create_parents(hdl, path_copy, prefix);
2915                 free(path_copy);
2916         }
2917         if (path_copy == NULL || rc != 0)
2918                 return (-1);

2920         return (0);
2921 }

2923 /*
2924  * Create a new filesystem or volume.
2925  */
2926 int
2927 zfs_create(libzfs_handle_t *hdl, const char *path, zfs_type_t type,
2928     nvlist_t *props)
2929 {
2930         int ret;
2931         uint64_t size = 0;
2932         uint64_t blocksize = zfs_prop_default_numeric(ZFS_PROP_VOLBLOCKSIZE);
2933         char errbuf[1024];
2934         uint64_t zoned;
2935         dmu_objset_type_t ost;

2937         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2938             "cannot create '%s'"), path);

2940         /* validate the path, taking care to note the extended error message */
2941         if (!zfs_validate_name(hdl, path, type, B_TRUE))
2942                 return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));

2944         /* validate parents exist */
2945         if (check_parents(hdl, path, &zoned, B_FALSE, NULL) != 0)
2946                 return (-1);

2948         /*
2949          * The failure modes when creating a dataset of a different type over
2950          * one that already exists is a little strange.  In particular, if you
2951          * try to create a dataset on top of an existing dataset, the ioctl()
2952          * will return ENOENT, not EEXIST.  To prevent this from happening, we
2953          * first try to see if the dataset exists.
2954          */
2955         if (zfs_dataset_exists(hdl, path, ZFS_TYPE_DATASET)) {
2956                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2957                     "dataset already exists"));
2958                 return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2959         }

2961         if (type == ZFS_TYPE_VOLUME)
2962                 ost = DMU_OST_ZVOL;
```



```
2963         else
2964                 ost = DMU_OST_ZFS;

2966         if (props && (props = zfs_valid_proplist(hdl, type, props,
2967             zoned, NULL, errbuf)) == 0)
2968                 return (-1);

2970         if (type == ZFS_TYPE_VOLUME) {
2971                 /*
2972                  * If we are creating a volume, the size and block size must
2973                  * satisfy a few restraints.  First, the blocksize must be a
2974                  * valid block size between SPA_{MIN,MAX}BLOCKSIZE.  Second, the
2975                  * volsize must be a multiple of the block size, and cannot be
2976                  * zero.
2977                  */
2978                 if (props == NULL || nvlist_lookup_uint64(props,
2979                     zfs_prop_to_name(ZFS_PROP_VOLSIZE), &size) != 0) {
2980                         nvlist_free(props);
2981                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2982                             "missing volume size"));
2983                         return (zfs_error(hdl, EZFS_BADPROP, errbuf));
2984                 }

2986                 if ((ret = nvlist_lookup_uint64(props,
2987                     zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
2988                     &blocksize)) != 0) {
2989                         if (ret == ENOENT) {
2990                                 blocksize = zfs_prop_default_numeric(
2991                                     ZFS_PROP_VOLBLOCKSIZE);
2992                         } else {
2993                                 nvlist_free(props);
2994                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2995                                     "missing volume block size"));
2996                                 return (zfs_error(hdl, EZFS_BADPROP, errbuf));
2997                         }
2998                 }

3000                 if (size == 0) {
3001                         nvlist_free(props);
3002                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3003                             "volume size cannot be zero"));
3004                         return (zfs_error(hdl, EZFS_BADPROP, errbuf));
3005                 }

3007                 if (size % blocksize != 0) {
3008                         nvlist_free(props);
3009                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3010                             "volume size must be a multiple of volume block "
3011                             "size"));
3012                         return (zfs_error(hdl, EZFS_BADPROP, errbuf));
3013                 }
3014         }

3016         /* create the dataset */
3017         ret = lzc_create(path, ost, props);
3018         nvlist_free(props);

3020         /* check for failure */
3021         if (ret != 0) {
3022                 char parent[ZFS_MAXNAMELEN];
3023                 (void) parent_name(path, parent, sizeof (parent));

3025                 switch (errno) {
3026                 case ENOENT:
3027                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3028                             "no such parent '%s'"), parent);
```

```
3029                        return (zfs_error(hdl, EZFS_NOENT, errbuf));

3031                case EINVAL:
3032                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3033                            "parent '%s' is not a filesystem"), parent);
3034                        return (zfs_error(hdl, EZFS_BADTYPE, errbuf));

3036                case EDOM:
3037                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3038                            "volume block size must be power of 2 from "
3039                            "%u to %uk"),
3040                            (uint_t)SPA_MINBLOCKSIZE,
3041                            (uint_t)SPA_MAXBLOCKSIZE >> 10);

3043                        return (zfs_error(hdl, EZFS_BADPROP, errbuf));

3045                case ENOTSUP:
3046                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3047                            "pool must be upgraded to set this "
3048                            "property or value"));
3049                        return (zfs_error(hdl, EZFS_BADVERSION, errbuf));
3050 #ifdef _ILP32
3051                case EOVERFLOW:
3052                        /*
3053                         * This platform can't address a volume this big.
3054                         */
3055                        if (type == ZFS_TYPE_VOLUME)
3056                                return (zfs_error(hdl, EZFS_VOLTOOBIG,
3057                                    errbuf));
3058 #endif
3059                        /* FALLTHROUGH */
3060                default:
3061                        return (zfs_standard_error(hdl, errno, errbuf));
3062                }
3063        }

3065        return (0);
3066 }

3068 /*
3069  * Destroys the given dataset.  The caller must make sure that the filesystem
3070  * isn't mounted, and that there are no active dependents. If the file system
3071  * does not exist this function does nothing.
3072  */
3073 int
3074 zfs_destroy(zfs_handle_t *zhp, boolean_t defer)
3075 {
3076        zfs_cmd_t zc = { 0 };

3078        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

3080        if (ZFS_IS_VOLUME(zhp)) {
3081                zc.zc_objset_type = DMU_OST_ZVOL;
3082        } else {
3083                zc.zc_objset_type = DMU_OST_ZFS;
3084        }

3086        zc.zc_defer_destroy = defer;
3087        if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_DESTROY, &zc) != 0 &&
3088            errno != ENOENT) {
3089                return (zfs_standard_error_fmt(zhp->zfs_hdl, errno,
3090                    dgettext(TEXT_DOMAIN, "cannot destroy '%s'"),
3091                    zhp->zfs_name));
3092        }

3094        remove_mountpoint(zhp);
```



```
3096        return (0);
3097 }

3099 struct destroydata {
3100        nvlist_t *nvl;
3101        const char *snapname;
3102 };

3104 static int
3105 zfs_check_snap_cb(zfs_handle_t *zhp, void *arg)
3106 {
3107        struct destroydata *dd = arg;
3108        zfs_handle_t *szhp;
3109        char name[ZFS_MAXNAMELEN];
3110        int rv = 0;

3112        (void) snprintf(name, sizeof (name),
3113            "%s@%s", zhp->zfs_name, dd->snapname);

3115        szhp = make_dataset_handle(zhp->zfs_hdl, name);
3116        if (szhp) {
3117                verify(nvlist_add_boolean(dd->nvl, name) == 0);
3118                zfs_close(szhp);
3119        }

3121        rv = zfs_iter_filesystems(zhp, zfs_check_snap_cb, dd);
3122        zfs_close(zhp);
3123        return (rv);
3124 }

3126 /*
3127  * Destroys all snapshots with the given name in zhp & descendants.
3128  */
3129 int
3130 zfs_destroy_snaps(zfs_handle_t *zhp, char *snapname, boolean_t defer)
3131 {
3132        int ret;
3133        struct destroydata dd = { 0 };

3135        dd.snapname = snapname;
3136        verify(nvlist_alloc(&dd.nvl, NV_UNIQUE_NAME, 0) == 0);
3137        (void) zfs_check_snap_cb(zfs_handle_dup(zhp), &dd);

3139        if (nvlist_next_nvpair(dd.nvl, NULL) == NULL) {
3140                ret = zfs_standard_error_fmt(zhp->zfs_hdl, ENOENT,
3141                    dgettext(TEXT_DOMAIN, "cannot destroy '%s@%s'"),
3142                    zhp->zfs_name, snapname);
3143        } else {
3144                ret = zfs_destroy_snaps_nvl(zhp->zfs_hdl, dd.nvl, defer);
3145        }
3146        nvlist_free(dd.nvl);
3147        return (ret);
3148 }

3150 /*
3151  * Destroys all the snapshots named in the nvlist.
3152  */
3153 int
3154 zfs_destroy_snaps_nvl(libzfs_handle_t *hdl, nvlist_t *snaps, boolean_t defer)
3155 {
3156        int ret;
3157        nvlist_t *errlist;

3159        ret = lzc_destroy_snaps(snaps, defer, &errlist);
```

```
3161         if (ret == 0)
3162                 return (0);

3164         if (nvlist_next_nvpair(errlist, NULL) == NULL) {
3165                 char errbuf[1024];
3166                 (void) snprintf(errbuf, sizeof (errbuf),
3167                     dgettext(TEXT_DOMAIN, "cannot destroy snapshots"));

3169                 ret = zfs_standard_error(hdl, ret, errbuf);
3170         }
3171         for (nvpair_t *pair = nvlist_next_nvpair(errlist, NULL);
3172             pair != NULL; pair = nvlist_next_nvpair(errlist, pair)) {
3173                 char errbuf[1024];
3174                 (void) snprintf(errbuf, sizeof (errbuf),
3175                     dgettext(TEXT_DOMAIN, "cannot destroy snapshot %s"),
3176                     nvpair_name(pair));

3178                 switch (fnvpair_value_int32(pair)) {
3179                 case EEXIST:
3180                         zfs_error_aux(hdl,
3181                             dgettext(TEXT_DOMAIN, "snapshot is cloned"));
3182                         ret = zfs_error(hdl, EZFS_EXISTS, errbuf);
3183                         break;
3184                 default:
3185                         ret = zfs_standard_error(hdl, errno, errbuf);
3186                         break;
3187                 }
3188         }

3190         return (ret);
3191 }

3193 /*
3194  * Clones the given dataset.  The target must be of the same type as the source.
3195  */
3196 int
3197 zfs_clone(zfs_handle_t *zhp, const char *target, nvlist_t *props)
3198 {
3199         char parent[ZFS_MAXNAMELEN];
3200         int ret;
3201         char errbuf[1024];
3202         libzfs_handle_t *hdl = zhp->zfs_hdl;
3203         uint64_t zoned;

3205         assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);

3207         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3208             "cannot create '%s'"), target);

3210         /* validate the target/clone name */
3211         if (!zfs_validate_name(hdl, target, ZFS_TYPE_FILESYSTEM, B_TRUE))
3212                 return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));

3214         /* validate parents exist */
3215         if (check_parents(hdl, target, &zoned, B_FALSE, NULL) != 0)
3216                 return (-1);

3218         (void) parent_name(target, parent, sizeof (parent));

3220         /* do the clone */

3222         if (props) {
3223                 zfs_type_t type;
3224                 if (ZFS_IS_VOLUME(zhp)) {
3225                         type = ZFS_TYPE_VOLUME;
3226                 } else {
```



```
3227                         type = ZFS_TYPE_FILESYSTEM;
3228                 }
3229                 if ((props = zfs_valid_proplist(hdl, type, props, zoned,
3230                     zhp, errbuf)) == NULL)
3231                         return (-1);
3232         }

3234         ret = lzc_clone(target, zhp->zfs_name, props);
3235         nvlist_free(props);

3237         if (ret != 0) {
3238                 switch (errno) {

3240                 case ENOENT:
3241                         /*
3242                          * The parent doesn't exist.  We should have caught this
3243                          * above, but there may a race condition that has since
3244                          * destroyed the parent.
3245                          *
3246                          * At this point, we don't know whether it's the source
3247                          * that doesn't exist anymore, or whether the target
3248                          * dataset doesn't exist.
3249                          */
3250                         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
3251                             "no such parent '%s'"), parent);
3252                         return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));

3254                 case EXDEV:
3255                         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
3256                             "source and target pools differ"));
3257                         return (zfs_error(zhp->zfs_hdl, EZFS_CROSSTARGET,
3258                             errbuf));

3260                 default:
3261                         return (zfs_standard_error(zhp->zfs_hdl, errno,
3262                             errbuf));
3263                 }
3264         }

3266         return (ret);
3267 }

3269 /*
3270  * Promotes the given clone fs to be the clone parent.
3271  */
3272 int
3273 zfs_promote(zfs_handle_t *zhp)
3274 {
3275         libzfs_handle_t *hdl = zhp->zfs_hdl;
3276         zfs_cmd_t zc = { 0 };
3277         char parent[MAXPATHLEN];
3278         int ret;
3279         char errbuf[1024];

3281         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3282             "cannot promote '%s'"), zhp->zfs_name);

3284         if (zhp->zfs_type == ZFS_TYPE_SNAPSHOT) {
3285                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3286                     "snapshots can not be promoted"));
3287                 return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
3288         }

3290         (void) strlcpy(parent, zhp->zfs_dmustats.dds_origin, sizeof (parent));
3291         if (parent[0] == '\0') {
3292                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
```

```
3293                     "not a cloned filesystem"));
3294                 return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
3295         }

3297         (void) strlcpy(zc.zc_value, zhp->zfs_dmustats.dds_origin,
3298             sizeof (zc.zc_value));
3299         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
3300         ret = zfs_ioctl(hdl, ZFS_IOC_PROMOTE, &zc);

3302         if (ret != 0) {
3303                 int save_errno = errno;

3305                 switch (save_errno) {
3306                 case EEXIST:
3307                         /* There is a conflicting snapshot name. */
3308                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3309                             "conflicting snapshot '%s' from parent '%s'"),
3310                             zc.zc_string, parent);
3311                         return (zfs_error(hdl, EZFS_EXISTS, errbuf));

3313                 default:
3314                         return (zfs_standard_error(hdl, save_errno, errbuf));
3315                 }
3316         }
3317         return (ret);
3318 }

3320 typedef struct snapdata {
3321         nvlist_t *sd_nvl;
3322         const char *sd_snapname;
3323 } snapdata_t;

3325 static int
3326 zfs_snapshot_cb(zfs_handle_t *zhp, void *arg)
3327 {
3328         snapdata_t *sd = arg;
3329         char name[ZFS_MAXNAMELEN];
3330         int rv = 0;

3332         (void) snprintf(name, sizeof (name),
3333             "%s@%s", zfs_get_name(zhp), sd->sd_snapname);

3335         fnvlist_add_boolean(sd->sd_nvl, name);

3337         rv = zfs_iter_filesystems(zhp, zfs_snapshot_cb, sd);
3338         zfs_close(zhp);
3339         return (rv);
3340 }

3342 /*
3343  * Creates snapshots.  The keys in the snaps nvlist are the snapshots to be
3344  * created.
3345  */
3346 int
3347 zfs_snapshot_nvl(libzfs_handle_t *hdl, nvlist_t *snaps, nvlist_t *props)
3348 {
3349         int ret;
3350         char errbuf[1024];
3351         nvpair_t *elem;
3352         nvlist_t *errors;

3354         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3355             "cannot create snapshots "));

3357         elem = NULL;
3358         while ((elem = nvlist_next_nvpair(snaps, elem)) != NULL) {
```



```
3359                 const char *snapname = nvpair_name(elem);

3361                 /* validate the target name */
3362                 if (!zfs_validate_name(hdl, snapname, ZFS_TYPE_SNAPSHOT,
3363                     B_TRUE)) {
3364                         (void) snprintf(errbuf, sizeof (errbuf),
3365                             dgettext(TEXT_DOMAIN,
3366                             "cannot create snapshot '%s'"), snapname);
3367                         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3368                 }
3369         }

3371         if (props != NULL &&
3372             (props = zfs_valid_proplist(hdl, ZFS_TYPE_SNAPSHOT,
3373             props, B_FALSE, NULL, errbuf)) == NULL) {
3374                 return (-1);
3375         }

3377         ret = lzc_snapshot(snaps, props, &errors);

3379         if (ret != 0) {
3380                 boolean_t printed = B_FALSE;
3381                 for (elem = nvlist_next_nvpair(errors, NULL);
3382                     elem != NULL;
3383                     elem = nvlist_next_nvpair(errors, elem)) {
3384                         (void) snprintf(errbuf, sizeof (errbuf),
3385                             dgettext(TEXT_DOMAIN,
3386                             "cannot create snapshot '%s'"), nvpair_name(elem));
3387                         (void) zfs_standard_error(hdl,
3388                             fnvpair_value_int32(elem), errbuf);
3389                         printed = B_TRUE;
3390                 }
3391                 if (!printed) {
3392                         switch (ret) {
3393                         case EXDEV:
3394                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3395                                     "multiple snapshots of same "
3396                                     "fs not allowed"));
3397                                 (void) zfs_error(hdl, EZFS_EXISTS, errbuf);

3399                                 break;
3400                         default:
3401                                 (void) zfs_standard_error(hdl, ret, errbuf);
3402                         }
3403                 }
3404         }

3406         nvlist_free(props);
3407         nvlist_free(errors);
3408         return (ret);
3409 }

3411 int
3412 zfs_snapshot(libzfs_handle_t *hdl, const char *path, boolean_t recursive,
3413     nvlist_t *props)
3414 {
3415         int ret;
3416         snapdata_t sd = { 0 };
3417         char fsname[ZFS_MAXNAMELEN];
3418         char *cp;
3419         zfs_handle_t *zhp;
3420         char errbuf[1024];

3422         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3423             "cannot snapshot %s"), path);
```

```
3425         if (!zfs_validate_name(hdl, path, ZFS_TYPE_SNAPSHOT, B_TRUE))
3426                 return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));

3428         (void) strlcpy(fsname, path, sizeof (fsname));
3429         cp = strchr(fsname, '@');
3430         *cp = '\0';
3431         sd.sd_snapname = cp + 1;

3433         if ((zhp = zfs_open(hdl, fsname, ZFS_TYPE_FILESYSTEM |
3434             ZFS_TYPE_VOLUME)) == NULL) {
3435                 return (-1);
3436         }

3438         verify(nvlist_alloc(&sd.sd_nvl, NV_UNIQUE_NAME, 0) == 0);
3439         if (recursive) {
3440                 (void) zfs_snapshot_cb(zfs_handle_dup(zhp), &sd);
3441         } else {
3442                 fnvlist_add_boolean(sd.sd_nvl, path);
3443         }

3445         ret = zfs_snapshot_nvl(hdl, sd.sd_nvl, props);
3446         nvlist_free(sd.sd_nvl);
3447         zfs_close(zhp);
3448         return (ret);
3449 }

3451 /*
3452  * Destroy any more recent snapshots.  We invoke this callback on any dependents
3453  * of the snapshot first.  If the 'cb_dependent' member is non-zero, then this
3454  * is a dependent and we should just destroy it without checking the transaction
3455  * group.
3456  */
3457 typedef struct rollback_data {
3458         const char      *cb_target;             /* the snapshot */
3459         uint64_t        cb_create;              /* creation time reference */
3460         boolean_t       cb_error;
3461         boolean_t       cb_dependent;
3462         boolean_t       cb_force;
3463 } rollback_data_t;

3465 static int
3466 rollback_destroy(zfs_handle_t *zhp, void *data)
3467 {
3468         rollback_data_t *cbp = data;

3470         if (!cbp->cb_dependent) {
3471                 if (strcmp(zhp->zfs_name, cbp->cb_target) != 0 &&
3472                     zfs_get_type(zhp) == ZFS_TYPE_SNAPSHOT &&
3473                     zfs_prop_get_int(zhp, ZFS_PROP_CREATETXG) >
3474                     cbp->cb_create) {

3476                         cbp->cb_dependent = B_TRUE;
3477                         cbp->cb_error |= zfs_iter_dependents(zhp, B_FALSE,
3478                             rollback_destroy, cbp);
3479                         cbp->cb_dependent = B_FALSE;

3481                         cbp->cb_error |= zfs_destroy(zhp, B_FALSE);
3482                 }
3483         } else {
3484                 /* We must destroy this clone; first unmount it */
3485                 prop_changelist_t *clp;

3487                 clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
3488                     cbp->cb_force ? MS_FORCE: 0);
3489                 if (clp == NULL || changelist_prefix(clp) != 0) {
3490                         cbp->cb_error = B_TRUE;
```



```
3491                         zfs_close(zhp);
3492                         return (0);
3493                 }
3494                 if (zfs_destroy(zhp, B_FALSE) != 0)
3495                         cbp->cb_error = B_TRUE;
3496                 else
3497                         changelist_remove(clp, zhp->zfs_name);
3498                 (void) changelist_postfix(clp);
3499                 changelist_free(clp);
3500         }

3502         zfs_close(zhp);
3503         return (0);
3504 }

3506 /*
3507  * Given a dataset, rollback to a specific snapshot, discarding any
3508  * data changes since then and making it the active dataset.
3509  *
3510  * Any snapshots more recent than the target are destroyed, along with
3511  * their dependents.
3512  */
3513 int
3514 zfs_rollback(zfs_handle_t *zhp, zfs_handle_t *snap, boolean_t force)
3515 {
3516         rollback_data_t cb = { 0 };
3517         int err;
3518         zfs_cmd_t zc = { 0 };
3519         boolean_t restore_resv = 0;
3520         uint64_t old_volsize, new_volsize;
3521         zfs_prop_t resv_prop;

3523         assert(zhp->zfs_type == ZFS_TYPE_FILESYSTEM ||
3524             zhp->zfs_type == ZFS_TYPE_VOLUME);

3526         /*
3527          * Destroy all recent snapshots and their dependents.
3528          */
3529         cb.cb_force = force;
3530         cb.cb_target = snap->zfs_name;
3531         cb.cb_create = zfs_prop_get_int(snap, ZFS_PROP_CREATETXG);
3532         (void) zfs_iter_children(zhp, rollback_destroy, &cb);

3534         if (cb.cb_error)
3535                 return (-1);

3537         /*
3538          * Now that we have verified that the snapshot is the latest,
3539          * rollback to the given snapshot.
3540          */

3542         if (zhp->zfs_type == ZFS_TYPE_VOLUME) {
3543                 if (zfs_which_resv_prop(zhp, &resv_prop) < 0)
3544                         return (-1);
3545                 old_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
3546                 restore_resv =
3547                     (old_volsize == zfs_prop_get_int(zhp, resv_prop));
3548         }

3550         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

3552         if (ZFS_IS_VOLUME(zhp))
3553                 zc.zc_objset_type = DMU_OST_ZVOL;
3554         else
3555                 zc.zc_objset_type = DMU_OST_ZFS;
```

```
3557         /*
3558          * We rely on zfs_iter_children() to verify that there are no
3559          * newer snapshots for the given dataset.  Therefore, we can
3560          * simply pass the name on to the ioctl() call.  There is still
3561          * an unlikely race condition where the user has taken a
3562          * snapshot since we verified that this was the most recent.
3563          *
3564          */
3565         if ((err = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_ROLLBACK, &zc)) != 0) {
3566                 (void) zfs_standard_error_fmt(zhp->zfs_hdl, errno,
3567                     dgettext(TEXT_DOMAIN, "cannot rollback '%s'"),
3568                     zhp->zfs_name);
3569                 return (err);
3570         }
3571
3572         /*
3573          * For volumes, if the pre-rollback volsize matched the pre-
3574          * rollback reservation and the volsize has changed then set
3575          * the reservation property to the post-rollback volsize.
3576          * Make a new handle since the rollback closed the dataset.
3577          */
3578         if ((zhp->zfs_type == ZFS_TYPE_VOLUME) &&
3579             (zhp = make_dataset_handle(zhp->zfs_hdl, zhp->zfs_name))) {
3580                 if (restore_resv) {
3581                         new_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
3582                         if (old_volsize != new_volsize)
3583                                 err = zfs_prop_set_int(zhp, resv_prop,
3584                                     new_volsize);
3585                 }
3586                 zfs_close(zhp);
3587         }
3588         return (err);
3589 }
3590
3591 /*
3592  * Renames the given dataset.
3593  */
3594 int
3595 zfs_rename(zfs_handle_t *zhp, const char *target, boolean_t recursive,
3596     boolean_t force_unmount)
3597 {
3598         int ret;
3599         zfs_cmd_t zc = { 0 };
3600         char *delim;
3601         prop_changelist_t *cl = NULL;
3602         zfs_handle_t *zhrp = NULL;
3603         char *parentname = NULL;
3604         char parent[ZFS_MAXNAMELEN];
3605         libzfs_handle_t *hdl = zhp->zfs_hdl;
3606         char errbuf[1024];
3607
3608         /* if we have the same exact name, just return success */
3609         if (strcmp(zhp->zfs_name, target) == 0)
3610                 return (0);
3611
3612         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3613             "cannot rename to '%s'"), target);
3614
3615         /*
3616          * Make sure the target name is valid
3617          */
3618         if (zhp->zfs_type == ZFS_TYPE_SNAPSHOT) {
3619                 if ((strchr(target, '@') == NULL) ||
3620                     *target == '@') {
3621                         /*
3622                          * Snapshot target name is abbreviated,
```



```
3623                          * reconstruct full dataset name
3624                          */
3625                         (void) strlcpy(parent, zhp->zfs_name,
3626                             sizeof (parent));
3627                         delim = strchr(parent, '@');
3628                         if (strchr(target, '@') == NULL)
3629                                 *(++delim) = '\0';
3630                         else
3631                                 *delim = '\0';
3632                         (void) strlcat(parent, target, sizeof (parent));
3633                         target = parent;
3634                 } else {
3635                         /*
3636                          * Make sure we're renaming within the same dataset.
3637                          */
3638                         delim = strchr(target, '@');
3639                         if (strncmp(zhp->zfs_name, target, delim - target)
3640                             != 0 || zhp->zfs_name[delim - target] != '@') {
3641                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3642                                     "snapshots must be part of same "
3643                                     "dataset"));
3644                                 return (zfs_error(hdl, EZFS_CROSSTARGET,
3645                                     errbuf));
3646                         }
3647                 }
3648                 if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
3649                         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3650         } else {
3651                 if (recursive) {
3652                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3653                             "recursive rename must be a snapshot"));
3654                         return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
3655                 }
3656
3657                 if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
3658                         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3659
3660                 /* validate parents */
3661                 if (check_parents(hdl, target, NULL, B_FALSE, NULL) != 0)
3662                         return (-1);
3663
3664                 /* make sure we're in the same pool */
3665                 verify((delim = strchr(target, '/')) != NULL);
3666                 if (strncmp(zhp->zfs_name, target, delim - target) != 0 ||
3667                     zhp->zfs_name[delim - target] != '/') {
3668                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3669                             "datasets must be within same pool"));
3670                         return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));
3671                 }
3672
3673                 /* new name cannot be a child of the current dataset name */
3674                 if (is_descendant(zhp->zfs_name, target)) {
3675                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3676                             "New dataset name cannot be a descendant of "
3677                             "current dataset name"));
3678                         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3679                 }
3680         }
3681
3682         (void) snprintf(errbuf, sizeof (errbuf),
3683             dgettext(TEXT_DOMAIN, "cannot rename '%s'"), zhp->zfs_name);
3684
3685         if (getzoneid() == GLOBAL_ZONEID &&
3686             zfs_prop_get_int(zhp, ZFS_PROP_ZONED)) {
3687                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3688                     "dataset is used in a non-global zone"));
```

```
3689                return (zfs_error(hdl, EZFS_ZONED, errbuf));
3690        }
3691
3692        if (recursive) {
3693
3694                parentname = zfs_strdup(zhp->zfs_hdl, zhp->zfs_name);
3695                if (parentname == NULL) {
3696                        ret = -1;
3697                        goto error;
3698                }
3699                delim = strchr(parentname, '@');
3700                *delim = '\0';
3701                zhrp = zfs_open(zhp->zfs_hdl, parentname, ZFS_TYPE_DATASET);
3702                if (zhrp == NULL) {
3703                        ret = -1;
3704                        goto error;
3705                }
3706
3707        } else {
3708                if ((cl = changelist_gather(zhp, ZFS_PROP_NAME, 0,
3709                    force_unmount ? MS_FORCE : 0)) == NULL)
3710                        return (-1);
3711
3712                if (changelist_haszonedchild(cl)) {
3713                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3714                            "child dataset with inherited mountpoint is used "
3715                            "in a non-global zone"));
3716                        (void) zfs_error(hdl, EZFS_ZONED, errbuf);
3717                        goto error;
3718                }
3719
3720                if ((ret = changelist_prefix(cl)) != 0)
3721                        goto error;
3722        }
3723
3724        if (ZFS_IS_VOLUME(zhp))
3725                zc.zc_objset_type = DMU_OST_ZVOL;
3726        else
3727                zc.zc_objset_type = DMU_OST_ZFS;
3728
3729        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
3730        (void) strlcpy(zc.zc_value, target, sizeof (zc.zc_value));
3731
3732        zc.zc_cookie = recursive;
3733
3734        if ((ret = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_RENAME, &zc)) != 0) {
3735                /*
3736                 * if it was recursive, the one that actually failed will
3737                 * be in zc.zc_name
3738                 */
3739                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3740                    "cannot rename '%s'"), zc.zc_name);
3741
3742                if (recursive && errno == EEXIST) {
3743                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3744                            "a child dataset already has a snapshot "
3745                            "with the new name"));
3746                        (void) zfs_error(hdl, EZFS_EXISTS, errbuf);
3747                } else {
3748                        (void) zfs_standard_error(zhp->zfs_hdl, errno, errbuf);
3749                }
3750
3751                /*
3752                 * On failure, we still want to remount any filesystems that
3753                 * were previously mounted, so we don't alter the system state.
3754                 */
```



```
3755                if (!recursive)
3756                        (void) changelist_postfix(cl);
3757        } else {
3758                if (!recursive) {
3759                        changelist_rename(cl, zfs_get_name(zhp), target);
3760                        ret = changelist_postfix(cl);
3761                }
3762        }
3763
3764 error:
3765        if (parentname) {
3766                free(parentname);
3767        }
3768        if (zhrp) {
3769                zfs_close(zhrp);
3770        }
3771        if (cl) {
3772                changelist_free(cl);
3773        }
3774        return (ret);
3775 }
3776
3777 nvlist_t *
3778 zfs_get_user_props(zfs_handle_t *zhp)
3779 {
3780        return (zhp->zfs_user_props);
3781 }
3782
3783 nvlist_t *
3784 zfs_get_recvd_props(zfs_handle_t *zhp)
3785 {
3786        if (zhp->zfs_recvd_props == NULL)
3787                if (get_recvd_props_ioctl(zhp) != 0)
3788                        return (NULL);
3789        return (zhp->zfs_recvd_props);
3790 }
3791
3792 /*
3793  * This function is used by 'zfs list' to determine the exact set of columns to
3794  * display, and their maximum widths.  This does two main things:
3795  *
3796  *      - If this is a list of all properties, then expand the list to include
3797  *        all native properties, and set a flag so that for each dataset we look
3798  *        for new unique user properties and add them to the list.
3799  *
3800  *      - For non fixed-width properties, keep track of the maximum width seen
3801  *        so that we can size the column appropriately. If the user has
3802  *        requested received property values, we also need to compute the width
3803  *        of the RECEIVED column.
3804  */
3805 int
3806 zfs_expand_proplist(zfs_handle_t *zhp, zprop_list_t **plp, boolean_t received)
3807 {
3808        libzfs_handle_t *hdl = zhp->zfs_hdl;
3809        zprop_list_t *entry;
3810        zprop_list_t **last, **start;
3811        nvlist_t *userprops, *propval;
3812        nvpair_t *elem;
3813        char *strval;
3814        char buf[ZFS_MAXPROPLEN];
3815
3816        if (zprop_expand_list(hdl, plp, ZFS_TYPE_DATASET) != 0)
3817                return (-1);
3818
3819        userprops = zfs_get_user_props(zhp);
```

```
3821         entry = *plp;
3822         if (entry->pl_all && nvlist_next_nvpair(userprops, NULL) != NULL) {
3823                 /*
3824                  * Go through and add any user properties as necessary.  We
3825                  * start by incrementing our list pointer to the first
3826                  * non-native property.
3827                  */
3828                 start = plp;
3829                 while (*start != NULL) {
3830                         if ((*start)->pl_prop == ZPROP_INVAL)
3831                                 break;
3832                         start = &(*start)->pl_next;
3833                 }

3835                 elem = NULL;
3836                 while ((elem = nvlist_next_nvpair(userprops, elem)) != NULL) {
3837                         /*
3838                          * See if we've already found this property in our list.
3839                          */
3840                         for (last = start; *last != NULL;
3841                             last = &(*last)->pl_next) {
3842                                 if (strcmp((*last)->pl_user_prop,
3843                                     nvpair_name(elem)) == 0)
3844                                         break;
3845                         }

3847                         if (*last == NULL) {
3848                                 if ((entry = zfs_alloc(hdl,
3849                                     sizeof (zprop_list_t))) == NULL ||
3850                                     ((entry->pl_user_prop = zfs_strdup(hdl,
3851                                     nvpair_name(elem)))) == NULL) {
3852                                         free(entry);
3853                                         return (-1);
3854                                 }

3856                                 entry->pl_prop = ZPROP_INVAL;
3857                                 entry->pl_width = strlen(nvpair_name(elem));
3858                                 entry->pl_all = B_TRUE;
3859                                 *last = entry;
3860                         }
3861                 }
3862         }

3864         /*
3865          * Now go through and check the width of any non-fixed columns
3866          */
3867         for (entry = *plp; entry != NULL; entry = entry->pl_next) {
3868                 if (entry->pl_fixed)
3869                         continue;

3871                 if (entry->pl_prop != ZPROP_INVAL) {
3872                         if (zfs_prop_get(zhp, entry->pl_prop,
3873                             buf, sizeof (buf), NULL, NULL, 0, B_FALSE) == 0) {
3874                                 if (strlen(buf) > entry->pl_width)
3875                                         entry->pl_width = strlen(buf);
3876                         }
3877                         if (received && zfs_prop_get_recvd(zhp,
3878                             zfs_prop_to_name(entry->pl_prop),
3879                             buf, sizeof (buf), B_FALSE) == 0)
3880                                 if (strlen(buf) > entry->pl_recvd_width)
3881                                         entry->pl_recvd_width = strlen(buf);
3882                 } else {
3883                         if (nvlist_lookup_nvlist(userprops, entry->pl_user_prop,
3884                             &propval) == 0) {
3885                                 verify(nvlist_lookup_string(propval,
3886                                     ZPROP_VALUE, &strval) == 0);
```



```
3887                                 if (strlen(strval) > entry->pl_width)
3888                                         entry->pl_width = strlen(strval);
3889                         }
3890                         if (received && zfs_prop_get_recvd(zhp,
3891                             entry->pl_user_prop,
3892                             buf, sizeof (buf), B_FALSE) == 0)
3893                                 if (strlen(buf) > entry->pl_recvd_width)
3894                                         entry->pl_recvd_width = strlen(buf);
3895                 }
3896         }

3898         return (0);
3899 }

3901 int
3902 zfs_deleg_share_nfs(libzfs_handle_t *hdl, char *dataset, char *path,
3903     char *resource, void *export, void *sharetab,
3904     int sharemax, zfs_share_op_t operation)
3905 {
3906         zfs_cmd_t zc = { 0 };
3907         int error;

3909         (void) strlcpy(zc.zc_name, dataset, sizeof (zc.zc_name));
3910         (void) strlcpy(zc.zc_value, path, sizeof (zc.zc_value));
3911         if (resource)
3912                 (void) strlcpy(zc.zc_string, resource, sizeof (zc.zc_string));
3913         zc.zc_share.z_sharedata = (uint64_t)(uintptr_t)sharetab;
3914         zc.zc_share.z_exportdata = (uint64_t)(uintptr_t)export;
3915         zc.zc_share.z_sharetype = operation;
3916         zc.zc_share.z_sharemax = sharemax;
3917         error = ioctl(hdl->libzfs_fd, ZFS_IOC_SHARE, &zc);
3918         return (error);
3919 }

3921 void
3922 zfs_prune_proplist(zfs_handle_t *zhp, uint8_t *props)
3923 {
3924         nvpair_t *curr;

3926         /*
3927          * Keep a reference to the props-table against which we prune the
3928          * properties.
3929          */
3930         zhp->zfs_props_table = props;

3932         curr = nvlist_next_nvpair(zhp->zfs_props, NULL);

3934         while (curr) {
3935                 zfs_prop_t zfs_prop = zfs_name_to_prop(nvpair_name(curr));
3936                 nvpair_t *next = nvlist_next_nvpair(zhp->zfs_props, curr);

3938                 /*
3939                  * User properties will result in ZPROP_INVAL, and since we
3940                  * only know how to prune standard ZFS properties, we always
3941                  * leave these in the list.  This can also happen if we
3942                  * encounter an unknown DSL property (when running older
3943                  * software, for example).
3944                  */
3945                 if (zfs_prop != ZPROP_INVAL && props[zfs_prop] == B_FALSE)
3946                         (void) nvlist_remove(zhp->zfs_props,
3947                             nvpair_name(curr), nvpair_type(curr));
3948                 curr = next;
3949         }
3950 }

3952 static int
```

```
zfs_smb_acl_mgmt(libzfs_handle_t *hdl, char *dataset, char *path,
    zfs_smb_acl_op_t cmd, char *resource1, char *resource2)
{
        zfs_cmd_t zc = { 0 };
        nvlist_t *nvlist = NULL;
        int error;

        (void) strlcpy(zc.zc_name, dataset, sizeof (zc.zc_name));
        (void) strlcpy(zc.zc_value, path, sizeof (zc.zc_value));
        zc.zc_cookie = (uint64_t)cmd;

        if (cmd == ZFS_SMB_ACL_RENAME) {
                if (nvlist_alloc(&nvlist, NV_UNIQUE_NAME, 0) != 0) {
                        (void) no_memory(hdl);
                        return (NULL);
                }
        }

        switch (cmd) {
        case ZFS_SMB_ACL_ADD:
        case ZFS_SMB_ACL_REMOVE:
                (void) strlcpy(zc.zc_string, resource1, sizeof (zc.zc_string));
                break;
        case ZFS_SMB_ACL_RENAME:
                if (nvlist_add_string(nvlist, ZFS_SMB_ACL_SRC,
                    resource1) != 0) {
                                (void) no_memory(hdl);
                                return (-1);
                }
                if (nvlist_add_string(nvlist, ZFS_SMB_ACL_TARGET,
                    resource2) != 0) {
                                (void) no_memory(hdl);
                                return (-1);
                }
                if (zcmd_write_src_nvlist(hdl, &zc, nvlist) != 0) {
                        nvlist_free(nvlist);
                        return (-1);
                }
                break;
        case ZFS_SMB_ACL_PURGE:
                break;
        default:
                return (-1);
        }
        error = ioctl(hdl->libzfs_fd, ZFS_IOC_SMB_ACL, &zc);
        if (nvlist)
                nvlist_free(nvlist);
        return (error);
}

int
zfs_smb_acl_add(libzfs_handle_t *hdl, char *dataset,
    char *path, char *resource)
{
        return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_ADD,
            resource, NULL));
}

int
zfs_smb_acl_remove(libzfs_handle_t *hdl, char *dataset,
    char *path, char *resource)
{
        return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_REMOVE,
            resource, NULL));
}
```



```
int
zfs_smb_acl_purge(libzfs_handle_t *hdl, char *dataset, char *path)
{
        return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_PURGE,
            NULL, NULL));
}

int
zfs_smb_acl_rename(libzfs_handle_t *hdl, char *dataset, char *path,
    char *oldname, char *newname)
{
        return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_RENAME,
            oldname, newname));
}

int
zfs_userspace(zfs_handle_t *zhp, zfs_userquota_prop_t type,
    zfs_userspace_cb_t func, void *arg)
{
        zfs_cmd_t zc = { 0 };
        zfs_useracct_t buf[100];
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        int ret;

        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

        zc.zc_objset_type = type;
        zc.zc_nvlist_dst = (uintptr_t)buf;

        for (;;) {
                zfs_useracct_t *zua = buf;

                zc.zc_nvlist_dst_size = sizeof (buf);
                if (zfs_ioctl(hdl, ZFS_IOC_USERSPACE_MANY, &zc) != 0) {
                        char errbuf[1024];

                        (void) snprintf(errbuf, sizeof (errbuf),
                            dgettext(TEXT_DOMAIN,
                            "cannot get used/quota for %s"), zc.zc_name);
                        return (zfs_standard_error_fmt(hdl, errno, errbuf));
                }
                if (zc.zc_nvlist_dst_size == 0)
                        break;

                while (zc.zc_nvlist_dst_size > 0) {
                        if ((ret = func(arg, zua->zu_domain, zua->zu_rid,
                            zua->zu_space)) != 0)
                                return (ret);
                        zua++;
                        zc.zc_nvlist_dst_size -= sizeof (zfs_useracct_t);
                }
        }

        return (0);
}

struct holdarg {
        nvlist_t *nvl;
        const char *snapname;
        const char *tag;
        boolean_t recursive;
};

static int
zfs_hold_one(zfs_handle_t *zhp, void *arg)
{
```

```
4085         struct holdarg *ha = arg;
4086         zfs_handle_t *szhp;
4087         char name[ZFS_MAXNAMELEN];
4088         int rv = 0;
4089
4090         (void) snprintf(name, sizeof (name),
4091             "%s@%s", zhp->zfs_name, ha->snapname);
4092
4093         szhp = make_dataset_handle(zhp->zfs_hdl, name);
4094         if (szhp) {
4095                 fnvlist_add_string(ha->nvl, name, ha->tag);
4096                 zfs_close(szhp);
4097         }
4098
4099         if (ha->recursive)
4100                 rv = zfs_iter_filesystems(zhp, zfs_hold_one, ha);
4101         zfs_close(zhp);
4102         return (rv);
4103 }
4104
4105 int
4106 zfs_hold(zfs_handle_t *zhp, const char *snapname, const char *tag,
4107     boolean_t recursive, boolean_t enoent_ok, int cleanup_fd)
4108 {
4109         int ret;
4110         struct holdarg ha;
4111         nvlist_t *errors;
4112         libzfs_handle_t *hdl = zhp->zfs_hdl;
4113         char errbuf[1024];
4114         nvpair_t *elem;
4115
4116         ha.nvl = fnvlist_alloc();
4117         ha.snapname = snapname;
4118         ha.tag = tag;
4119         ha.recursive = recursive;
4120         (void) zfs_hold_one(zfs_handle_dup(zhp), &ha);
4121
4122         if (nvlist_next_nvpair(ha.nvl, NULL) == NULL) {
4123                 fnvlist_free(ha.nvl);
4124                 ret = ENOENT;
4125                 if (!enoent_ok) {
4126                         (void) snprintf(errbuf, sizeof (errbuf),
4127                             dgettext(TEXT_DOMAIN,
4128                             "cannot hold snapshot '%s@%s'"),
4129                             zhp->zfs_name, snapname);
4130                         (void) zfs_standard_error(hdl, ret, errbuf);
4131                 }
4132                 return (ret);
4133         }
4134
4135 #endif /* ! codereview */
4136         ret = lzc_hold(ha.nvl, cleanup_fd, &errors);
4137         fnvlist_free(ha.nvl);
4138
4139         if (ret == 0)
4140                 return (0);
4141
4142         if (nvlist_next_nvpair(errors, NULL) == NULL) {
4143                 /* no hold-specific errors */
4144                 (void) snprintf(errbuf, sizeof (errbuf),
4145                     dgettext(TEXT_DOMAIN, "cannot hold"));
4146                 switch (ret) {
4147                 case ENOTSUP:
4148                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4149                             "pool must be upgraded"));
4150                         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
```



```
4151                         break;
4152                 case EINVAL:
4153                         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4154                         break;
4155                 default:
4156                         (void) zfs_standard_error(hdl, ret, errbuf);
4157                 }
4158         }
4159
4160         for (elem = nvlist_next_nvpair(errors, NULL);
4161             elem != NULL;
4162             elem = nvlist_next_nvpair(errors, elem)) {
4163                 (void) snprintf(errbuf, sizeof (errbuf),
4164                     dgettext(TEXT_DOMAIN,
4165                     "cannot hold snapshot '%s'"), nvpair_name(elem));
4166                 switch (fnvpair_value_int32(elem)) {
4167                 case E2BIG:
4168                         /*
4169                          * Temporary tags wind up having the ds object id
4170                          * prepended. So even if we passed the length check
4171                          * above, it's still possible for the tag to wind
4172                          * up being slightly too long.
4173                          */
4174                         (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
4175                         break;
4176                 case EINVAL:
4177                         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4178                         break;
4179                 case EEXIST:
4180                         (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
4181                         break;
4182                 case ENOENT:
4183                         if (enoent_ok)
4184                                 return (ENOENT);
4185                         /* FALLTHROUGH */
4186                 default:
4187                         (void) zfs_standard_error(hdl,
4188                             fnvpair_value_int32(elem), errbuf);
4189                 }
4190         }
4191
4192         fnvlist_free(errors);
4193         return (ret);
4194 }
4195
4196 struct releasearg {
4197         nvlist_t *nvl;
4198         const char *snapname;
4199         const char *tag;
4200         boolean_t recursive;
4201 };
4202
4203 static int
4204 zfs_release_one(zfs_handle_t *zhp, void *arg)
4205 {
4206         struct holdarg *ha = arg;
4207         zfs_handle_t *szhp;
4208         char name[ZFS_MAXNAMELEN];
4209         int rv = 0;
4210
4211         (void) snprintf(name, sizeof (name),
4212             "%s@%s", zhp->zfs_name, ha->snapname);
4213
4214         szhp = make_dataset_handle(zhp->zfs_hdl, name);
4215         if (szhp) {
4216                 nvlist_t *holds = fnvlist_alloc();
```

```
4217                 fnvlist_add_boolean(holds, ha->tag);
4218                 fnvlist_add_nvlist(ha->nvl, name, holds);
4219                 zfs_close(szhp);
4220         }

4222         if (ha->recursive)
4223                 rv = zfs_iter_filesystems(zhp, zfs_release_one, ha);
4224         zfs_close(zhp);
4225         return (rv);
4226 }

4228 int
4229 zfs_release(zfs_handle_t *zhp, const char *snapname, const char *tag,
4230     boolean_t recursive)
4231 {
4232         int ret;
4233         struct holdarg ha;
4234         nvlist_t *errors;
4235         nvpair_t *elem;
4236         libzfs_handle_t *hdl = zhp->zfs_hdl;
4237         char errbuf[1024];
4238 #endif /* ! codereview */

4240         ha.nvl = fnvlist_alloc();
4241         ha.snapname = snapname;
4242         ha.tag = tag;
4243         ha.recursive = recursive;
4244         (void) zfs_release_one(zfs_handle_dup(zhp), &ha);

4246         if (nvlist_next_nvpair(ha.nvl, NULL) == NULL) {
4247                 fnvlist_free(ha.nvl);
4248                 ret = ENOENT;
4249                 (void) snprintf(errbuf, sizeof (errbuf),
4250                     dgettext(TEXT_DOMAIN,
4251                     "cannot release hold from snapshot '%s@%s'"),
4252                     zhp->zfs_name, snapname);
4253                 (void) zfs_standard_error(hdl, ret, errbuf);
4254                 return (ret);
4255         }

4257 #endif /* ! codereview */
4258         ret = lzc_release(ha.nvl, &errors);
4259         fnvlist_free(ha.nvl);

4261         if (ret == 0)
4262                 return (0);

4264         if (nvlist_next_nvpair(errors, NULL) == NULL) {
4265                 /* no hold-specific errors */
  27                 char errbuf[1024];

4266                 (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
4267                     "cannot release"));
4268                 switch (errno) {
4269                 case ENOTSUP:
4270                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4271                             "pool must be upgraded"));
4272                         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4273                         break;
4274                 default:
4275                         (void) zfs_standard_error_fmt(hdl, errno, errbuf);
4276                 }
4277         }

4279         for (elem = nvlist_next_nvpair(errors, NULL);
4280             elem != NULL;
```



```
4281             elem = nvlist_next_nvpair(errors, elem)) {
  45                 char errbuf[1024];

4282                 (void) snprintf(errbuf, sizeof (errbuf),
4283                     dgettext(TEXT_DOMAIN,
4284                     "cannot release hold from snapshot '%s'"),
4285                     nvpair_name(elem));
4286                 switch (fnvpair_value_int32(elem)) {
4287                 case ESRCH:
4288                         (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
4289                         break;
4290                 case EINVAL:
4291                         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4292                         break;
4293                 default:
4294                         (void) zfs_standard_error_fmt(hdl,
4295                             fnvpair_value_int32(elem), errbuf);
4296                 }
4297         }

4299         fnvlist_free(errors);
4300         return (ret);
4301 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   35734 Fri May 17 22:54:37 2013
new/usr/src/uts/common/fs/zfs/dsl_dir.c
3699 zfs hold or release of a non-existent snapshot does not output error
3739 cannot set zfs quota or reservation on pool version < 22
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Eric Shrock <eric.schrock@delphix.com>
**********************************************************
```

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

```
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                 }

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
```

```
 * to it?  If ondiskonly is set, we're only interested in what's
 * on-disk, not estimated pending changes.
 */
uint64_t
dsl_dir_space_available(dsl_dir_t *dd,
    dsl_dir_t *ancestor, int64_t delta, int ondiskonly)
{
        uint64_t parentspace, myspace, quota, used;

        /*
         * If there are no restrictions otherwise, assume we have
         * unlimited space available.
         */
        quota = UINT64_MAX;
        parentspace = UINT64_MAX;

        if (dd->dd_parent != NULL) {
                parentspace = dsl_dir_space_available(dd->dd_parent,
                    ancestor, delta, ondiskonly);
        }

        mutex_enter(&dd->dd_lock);
        if (dd->dd_phys->dd_quota != 0)
                quota = dd->dd_phys->dd_quota;
        used = dd->dd_phys->dd_used_bytes;
        if (!ondiskonly)
                used += dsl_dir_space_towrite(dd);

        if (dd->dd_parent == NULL) {
                uint64_t poolsize = dsl_pool_adjustedsize(dd->dd_pool, FALSE);
                quota = MIN(quota, poolsize);
        }

        if (dd->dd_phys->dd_reserved > used && parentspace != UINT64_MAX) {
                /*
                 * We have some space reserved, in addition to what our
                 * parent gave us.
                 */
                parentspace += dd->dd_phys->dd_reserved - used;
        }

        if (dd == ancestor) {
                ASSERT(delta <= 0);
                ASSERT(used >= -delta);
                used += delta;
                if (parentspace != UINT64_MAX)
                        parentspace -= delta;
        }

        if (used > quota) {
                /* over quota */
                myspace = 0;
        } else {
                /*
                 * the lesser of the space provided by our parent and
                 * the space left in our quota
                 */
                myspace = MIN(parentspace, quota - used);
        }

        mutex_exit(&dd->dd_lock);

        return (myspace);
}

struct tempreserve {
```



```
        list_node_t tr_node;
        dsl_pool_t *tr_dp;
        dsl_dir_t *tr_ds;
        uint64_t tr_size;
};

static int
dsl_dir_tempreserve_impl(dsl_dir_t *dd, uint64_t asize, boolean_t netfree,
    boolean_t ignorequota, boolean_t checkrefquota, list_t *tr_list,
    dmu_tx_t *tx, boolean_t first)
{
        uint64_t txg = tx->tx_txg;
        uint64_t est_inflight, used_on_disk, quota, parent_rsrv;
        uint64_t deferred = 0;
        struct tempreserve *tr;
        int retval = EDQUOT;
        int txgidx = txg & TXG_MASK;
        int i;
        uint64_t ref_rsrv = 0;

        ASSERT3U(txg, !=, 0);
        ASSERT3S(asize, >, 0);

        mutex_enter(&dd->dd_lock);

        /*
         * Check against the dsl_dir's quota.  We don't add in the delta
         * when checking for over-quota because they get one free hit.
         */
        est_inflight = dsl_dir_space_towrite(dd);
        for (i = 0; i < TXG_SIZE; i++)
                est_inflight += dd->dd_tempreserved[i];
        used_on_disk = dd->dd_phys->dd_used_bytes;

        /*
         * On the first iteration, fetch the dataset's used-on-disk and
         * refreservation values. Also, if checkrefquota is set, test if
         * allocating this space would exceed the dataset's refquota.
         */
        if (first && tx->tx_objset) {
                int error;
                dsl_dataset_t *ds = tx->tx_objset->os_dsl_dataset;

                error = dsl_dataset_check_quota(ds, checkrefquota,
                    asize, est_inflight, &used_on_disk, &ref_rsrv);
                if (error) {
                        mutex_exit(&dd->dd_lock);
                        return (error);
                }
        }

        /*
         * If this transaction will result in a net free of space,
         * we want to let it through.
         */
        if (ignorequota || netfree || dd->dd_phys->dd_quota == 0)
                quota = UINT64_MAX;
        else
                quota = dd->dd_phys->dd_quota;

        /*
         * Adjust the quota against the actual pool size at the root
         * minus any outstanding deferred frees.
         * To ensure that it's possible to remove files from a full
         * pool without inducing transient overcommits, we throttle
         * 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;
```

```
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;
```

```
} dsl_dir_set_qr_arg_t;

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;
        }
```

```
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
1054                 if (delta > avail ||
1055                     (dd->dd_phys->dd_quota > 0 &&
1056                     newval > dd->dd_phys->dd_quota))
1057                         error = SET_ERROR(ENOSPC);
1058         }
1059
1060         dsl_dataset_rele(ds, FTAG);
1061         return (error);
1062 }
1063
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 }
1084
1085
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;
1093
1094         VERIFY0(dsl_dataset_hold(dp, ddsqra->ddsqra_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),
  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);
1101
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 */
1111
1112         dsl_dir_set_reservation_sync_impl(ds->ds_dir, newval, tx);
1113         dsl_dataset_rele(ds, FTAG);
```



```
1114 }
1115
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;
1121
1122         ddsqra.ddsqra_name = ddname;
1123         ddsqra.ddsqra_source = source;
1124         ddsqra.ddsqra_value = reservation;
1125
1126         return (dsl_sync_task(ddname, dsl_dir_set_reservation_check,
1127             dsl_dir_set_reservation_sync, &ddsqra, 0));
1128 }
1129
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 }
1142
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
1159 typedef struct dsl_dir_rename_arg {
1160         const char *ddra_oldname;
1161         const char *ddra_newname;
1162 } dsl_dir_rename_arg_t;
1163
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
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 Fri May 17 22:54:38 2013
new/usr/src/uts/common/fs/zfs/dsl_prop.c
3699 zfs hold or release of a non-existent snapshot does not output error
3739 cannot set zfs quota or reservation on pool version < 22
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Eric Shrock <eric.schrock@delphix.com>
**********************************************************
```

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

```
126                 if (err == ENOENT) {
127                         /* 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 }
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;
266         error = dmu_objset_hold(dsname, FTAG, &os);
267         if (error != 0)
268                 return (error);
270         error = dsl_prop_get_ds(dmu_objset_ds(os), propname,
271             intsz, numints, buf, setpoint);
273         dmu_objset_rele(os, FTAG);
274         return (error);
275 }
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 }
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 }
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;
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                 }
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);
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                 }
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                 }
374                 strfree(recvdstr);
376                 if (err == ENOENT)
377                         return (0);
379                 return (err);
380 }
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_*