

```

*****
111451 Sat Apr 13 10:17:36 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.c
3699 zfs hold or release of a non-existent snapshot does not output error
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 2012 by Delphix. All rights reserved.
25  * Copyright (c) 2012 DEY Storage Systems, Inc. All rights reserved.
26  * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
27  * Copyright (c) 2013 Martin Matuska. All rights reserved.
28 #endif /* ! codereview */
29 */

31 #include <ctype.h>
32 #include <errno.h>
33 #include <libintl.h>
34 #include <math.h>
35 #include <stdio.h>
36 #include <stdlib.h>
37 #include <strings.h>
38 #include <unistd.h>
39 #include <stddef.h>
40 #include <zone.h>
41 #include <fcntl.h>
42 #include <sys/mntent.h>
43 #include <sys/mount.h>
44 #include <priv.h>
45 #include <pwd.h>
46 #include <grp.h>
47 #include <stddef.h>
48 #include <ucred.h>
49 #include <idmap.h>
50 #include <aclutils.h>
51 #include <directory.h>

53 #include <sys/dnode.h>
54 #include <sys/spa.h>
55 #include <sys/zap.h>
56 #include <libzfs.h>

58 #include "zfs_namecheck.h"
59 #include "zfs_prop.h"
60 #include "libzfs_impl.h"
61 #include "zfs_deleg.h"

```

```

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

66 /*
67  * Given a single type (not a mask of types), return the type in a human
68  * readable form.
69  */
70 const char *
71 zfs_type_to_name(zfs_type_t type)
72 {
73     switch (type) {
74     case ZFS_TYPE_FILESYSTEM:
75         return (dgettext(TEXT_DOMAIN, "filesystem"));
76     case ZFS_TYPE_SNAPSHOT:
77         return (dgettext(TEXT_DOMAIN, "snapshot"));
78     case ZFS_TYPE_VOLUME:
79         return (dgettext(TEXT_DOMAIN, "volume"));
80     }

82     return (NULL);
83 }

85 /*
86  * Given a path and mask of ZFS types, return a string describing this dataset.
87  * This is used when we fail to open a dataset and we cannot get an exact type.
88  * We guess what the type would have been based on the path and the mask of
89  * acceptable types.
90  */
91 static const char *
92 path_to_str(const char *path, int types)
93 {
94     /*
95      * When given a single type, always report the exact type.
96      */
97     if (types == ZFS_TYPE_SNAPSHOT)
98         return (dgettext(TEXT_DOMAIN, "snapshot"));
99     if (types == ZFS_TYPE_FILESYSTEM)
100        return (dgettext(TEXT_DOMAIN, "filesystem"));
101     if (types == ZFS_TYPE_VOLUME)
102        return (dgettext(TEXT_DOMAIN, "volume"));

104     /*
105      * The user is requesting more than one type of dataset. If this is the
106      * case, consult the path itself. If we're looking for a snapshot, and
107      * a '@' is found, then report it as "snapshot". Otherwise, remove the
108      * snapshot attribute and try again.
109      */
110     if (types & ZFS_TYPE_SNAPSHOT) {
111         if (strchr(path, '@') != NULL)
112             return (dgettext(TEXT_DOMAIN, "snapshot"));
113         return (path_to_str(path, types & ~ZFS_TYPE_SNAPSHOT));
114     }

116     /*
117      * The user has requested either filesystems or volumes.
118      * We have no way of knowing a priori what type this would be, so always
119      * report it as "filesystem" or "volume", our two primitive types.
120      */
121     if (types & ZFS_TYPE_FILESYSTEM)
122        return (dgettext(TEXT_DOMAIN, "filesystem"));

124     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;
139
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;
148
149                 case NAME_ERR_LEADING_SLASH:
150                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
151                     "leading slash in name"));
152                     break;
153
154                 case NAME_ERR_EMPTY_COMPONENT:
155                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
156                     "empty component in name"));
157                     break;
158
159                 case NAME_ERR_TRAILING_SLASH:
160                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
161                     "trailing slash in name"));
162                     break;
163
164                 case NAME_ERR_INVALIDCHAR:
165                     zfs_error_aux(hdl,
166                     dgettext(TEXT_DOMAIN, "invalid character "
167                     "'%c' in name"), what);
168                     break;
169
170                 case NAME_ERR_MULTIPLE_AT:
171                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
172                     "multiple '@' delimiters in name"));
173                     break;
174
175                 case NAME_ERR_NOLETTER:
176                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
177                     "pool doesn't begin with a letter"));
178                     break;
179
180                 case NAME_ERR_RESERVED:
181                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
182                     "name is reserved"));
183                     break;
184
185                 case NAME_ERR_DISKLIKE:
186                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
187                     "reserved disk name"));
188                     break;
189             }
190         }
191
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     }
201
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     }
208
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     }
215
216     return (-1);
217 }
218
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 }
226
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;
238
239     if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
240         (void) no_memory(hdl);
241         return (NULL);
242     }
243
244     elem = NULL;
245     while ((elem = nvlist_next_nvpair(props, elem)) != NULL) {
246         if (!zfs_prop_user(nvpair_name(elem)))
247             continue;
248
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     }
256
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;
265
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 }
273
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;
279
280     while ((zph != NULL) &&
281            (strcmp(pool_name, zpool_get_name(zph), len) != 0))
282            zph = zph->zpool_next;
283     return (zph);
284 }
285
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;
297
298     len = strlen(zhp->zfs_name) + 1;
299     pool_name = zfs_alloc(zhp->zfs_hdl, len);
300     (void) strcpy(pool_name, zhp->zfs_name, len);
301
302     zph = zpool_find_handle(zhp, pool_name, len);
303     if (zph == NULL)
304         zph = zpool_add_handle(zhp, pool_name);
305
306     free(pool_name);
307     return (zph);
308 }
309
310 void
311 zpool_free_handles(libzfs_handle_t *hdl)
312 {
313     zpool_handle_t *next, *zph = hdl->libzfs_pool_handles;
314
315     while (zph != NULL) {
316         next = zph->zpool_next;
317         zpool_close(zph);
318         zph = next;
319     }
320     hdl->libzfs_pool_handles = NULL;
321 }
322
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;
330
331     (void) strcpy(zc->zc_name, zhp->zfs_name, sizeof (zc->zc_name));
332
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 }
344
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;
355
356     if (zcmd_alloc_dst_nvlist(hdl, &zc, 0) != 0)
357         return (-1);
358
359     (void) strcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
360
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     }
371
372     err = zcmd_read_dst_nvlist(zhp->zfs_hdl, &zc, &recvdprops);
373     zcmd_free_nvlists(&zc);
374     if (err != 0)
375         return (-1);
376
377     nvlist_free(zhp->zfs_recvd_props);
378     zhp->zfs_recvd_props = recvdprops;
379
380     return (0);
381 }
382
383 static int
384 put_stats_zhdl(zfs_handle_t *zhp, zfs_cmd_t *zc)
385 {
386     nvlist_t *allprops, *userprops;
387
388     zhp->zfs_dmustats = zc->zc_objset_stats; /* structure assignment */
389
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();

```

```

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) strncpy(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) strncpy(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)
521 {
522     zfs_handle_t *zhp = calloc(sizeof (zfs_handle_t), 1);

```

```

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

527     zhp->zfs_hdl = zhp_orig->zfs_hdl;
528     zhp->zpool_hdl = zhp_orig->zpool_hdl;
529     (void) strcpy(zhp->zfs_name, zhp_orig->zfs_name,
530         sizeof (zhp->zfs_name));
531     zhp->zfs_type = zhp_orig->zfs_type;
532     zhp->zfs_head_type = zhp_orig->zfs_head_type;
533     zhp->zfs_dmstats = zhp_orig->zfs_dmstats;
534     if (zhp_orig->zfs_props != NULL) {
535         if (nvlst_dup(zhp_orig->zfs_props, &zhp->zfs_props, 0) != 0) {
536             (void) no_memory(zhp->zfs_hdl);
537             zfs_close(zhp);
538             return (NULL);
539         }
540     }
541     if (zhp_orig->zfs_user_props != NULL) {
542         if (nvlst_dup(zhp_orig->zfs_user_props,
543             &zhp->zfs_user_props, 0) != 0) {
544             (void) no_memory(zhp->zfs_hdl);
545             zfs_close(zhp);
546             return (NULL);
547         }
548     }
549     if (zhp_orig->zfs_recvd_props != NULL) {
550         if (nvlst_dup(zhp_orig->zfs_recvd_props,
551             &zhp->zfs_recvd_props, 0) != 0) {
552             (void) no_memory(zhp->zfs_hdl);
553             zfs_close(zhp);
554             return (NULL);
555         }
556     }
557     zhp->zfs_mntcheck = zhp_orig->zfs_mntcheck;
558     if (zhp_orig->zfs_mntopts != NULL) {
559         zhp->zfs_mntopts = zfs_strdup(zhp_orig->zfs_hdl,
560             zhp_orig->zfs_mntopts);
561     }
562     zhp->zfs_props_table = zhp_orig->zfs_props_table;
563     return (zhp);
564 }

566 /*
567  * Opens the given snapshot, filesystem, or volume.  The 'types'
568  * argument is a mask of acceptable types.  The function will print an
569  * appropriate error message and return NULL if it can't be opened.
570  */
571 zfs_handle_t *
572 zfs_open(libzfs_handle_t *hdl, const char *path, int types)
573 {
574     zfs_handle_t *zhp;
575     char errbuf[1024];

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

580     /*
581      * Validate the name before we even try to open it.
582      */
583     if (!zfs_validate_name(hdl, path, ZFS_TYPE_DATASET, B_FALSE)) {
584         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
585             "invalid dataset name"));
586         (void) zfs_error(hdl, EZFS_INVALIDNAME, errbuf);
587         return (NULL);
588     }

```

```

590     /*
591      * Try to get stats for the dataset, which will tell us if it exists.
592      */
593     errno = 0;
594     if ((zhp = make_dataset_handle(hdl, path)) == NULL) {
595         (void) zfs_standard_error(hdl, errno, errbuf);
596         return (NULL);
597     }

599     if (!(types & zhp->zfs_type)) {
600         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
601         zfs_close(zhp);
602         return (NULL);
603     }

605     return (zhp);
606 }

608 /*
609  * Release a ZFS handle.  Nothing to do but free the associated memory.
610  */
611 void
612 zfs_close(zfs_handle_t *zhp)
613 {
614     if (zhp->zfs_mntopts)
615         free(zhp->zfs_mntopts);
616     nvlst_free(zhp->zfs_props);
617     nvlst_free(zhp->zfs_user_props);
618     nvlst_free(zhp->zfs_recvd_props);
619     free(zhp);
620 }

622 typedef struct mnttab_node {
623     struct mnttab mtn_mt;
624     avl_node_t mtn_node;
625 } mnttab_node_t;

627 static int
628 libzfs_mnttab_cache_compare(const void *arg1, const void *arg2)
629 {
630     const mnttab_node_t *mtn1 = arg1;
631     const mnttab_node_t *mtn2 = arg2;
632     int rv;

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

636     if (rv == 0)
637         return (0);
638     return (rv > 0 ? 1 : -1);
639 }

641 void
642 libzfs_mnttab_init(libzfs_handle_t *hdl)
643 {
644     assert(avl_numnodes(&hdl->libzfs_mnttab_cache) == 0);
645     avl_create(&hdl->libzfs_mnttab_cache, libzfs_mnttab_cache_compare,
646         sizeof (mnttab_node_t), offsetof(mnttab_node_t, mtn_node));
647 }

649 void
650 libzfs_mnttab_update(libzfs_handle_t *hdl)
651 {
652     struct mnttab entry;

654     rewind(hdl->libzfs_mnttab);
655     while (getmntent(hdl->libzfs_mnttab, &entry) == 0) {

```

```

656     mnttab_node_t *mtn;

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 }

669 void
670 libzfs_mnttab_fini(libzfs_handle_t *hdl)
671 {
672     void *cookie = NULL;
673     mnttab_node_t *mtn;

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 }

685 void
686 libzfs_mnttab_cache(libzfs_handle_t *hdl, boolean_t enable)
687 {
688     hdl->libzfs_mnttab_enable = enable;
689 }

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;

698     if (!hdl->libzfs_mnttab_enable) {
699         struct mnttab srch = { 0 };

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     }

712     if (avl_numnodes(&hdl->libzfs_mnttab_cache) == 0)
713         libzfs_mnttab_update(hdl);

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 }

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;

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 }

740 void
741 libzfs_mnttab_remove(libzfs_handle_t *hdl, const char *fsname)
742 {
743     mnttab_node_t find;
744     mnttab_node_t *ret;

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 }

757 int
758 zfs_spa_version(zfs_handle_t *zhp, int *spa_version)
759 {
760     zpool_handle_t *zpool_handle = zhp->zpool_hdl;

762     if (zpool_handle == NULL)
763         return (-1);

765     *spa_version = zpool_get_prop_int(zpool_handle,
766         ZPOOL_PROP_VERSION, NULL);
767     return (0);
768 }

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;

778     if (zfs_spa_version(zhp, &spa_version) < 0)
779         return (-1);

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;
805
806     if (nvlist_alloc(&ret, NV_UNIQUE_NAME, 0) != 0) {
807         (void) no_memory(hdl);
808         return (NULL);
809     }
810
811     /*
812     * Make sure this property is valid and applies to this type.
813     */
814
815     elem = NULL;
816     while ((elem = nvlist_next_nvpair(nvl, elem)) != NULL) {
817         const char *propname = nvpair_name(elem);
818
819         prop = zfs_name_to_prop(propname);
820         if (prop == ZPROP_INVALID && 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             }
831
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             }
839
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         }
847
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     }
858
859     if (prop == ZPROP_INVALID && 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];
865
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         }
875
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         }
885
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         }
911
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_INVALID && 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_INVALID) {
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 (strncasecmp(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
1047     badlabel:
1048     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1049         "invalid mlslabel '%s'"), strval);
1050     (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1051     m_label_free(new_sl); /* OK if null */
1052     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 sharesnfs 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      * -----
1094      * global zone                non-global zone
1095      * -----
1096      * zoned=on      mountpoint (no)    mountpoint (yes)
1097      *                sharesnfs (no)     sharesnfs (no)
1098      *                sharesmb (no)      sharesmb (no)
1099      *
1100      * zoned=off    mountpoint (yes)     N/A
1101      *                sharesnfs (yes)
1102      *                sharesmb (yes)
1103      */
1104     if (zoned) {
1105         if (getzoneid() == GLOBAL_ZONEID) {
1106             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1107                 "%s' cannot be set on "
1108                 "dataset in a non-global zone"),
1109                 propname);
1110             (void) zfs_error(hdl, EZFS_ZONED,
1111                 errbuf);
1112             goto error;
1113         } else if (prop == ZFS_PROP_SHARENFS ||
1114             prop == ZFS_PROP_SHARESMB) {
1115             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1116                 "%s' cannot be set in "
1117                 "a non-global zone"), propname);
1118             (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 sharesnfs.
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 a 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              * uninitialized 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 }
1193
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 }
1202
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];
1213
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;
1226
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         }
1239
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 }
1252
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);
1274
1275 error:
1276     nvlist_free(ret);
1277     return (NULL);
1278 }
1279
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;
1289
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);
1298
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));
1302
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     }

```

```

1449     if ((cl = changelist_gather(zhp, prop, 0, 0)) == NULL)
1450         goto error;
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     }
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     }
1473     if (do_prefix && (ret = changelist_prefix(cl)) != 0)
1474         goto error;
1476     /*
1477     * Execute the corresponding ioctl() to set this property.
1478     */
1479     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
1481     if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1482         goto error;
1484     ret = zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
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);
1508         /*
1509         * Refresh the statistics so the new property value
1510         * is reflected.
1511         */
1512         if (ret == 0)
1513             (void) get_stats(zhp);

```

```

1514     }
1516 error:
1517     nvlist_free(nvl);
1518     zcmd_free_nvlists(&zc);
1519     if (cl)
1520         changelist_free(cl);
1521     return (ret);
1522 }
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;
1538     (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
1539         "cannot inherit %s for '%s'", propname, zhp->zfs_name));
1541     zc.zc_cookie = received;
1542     if ((prop = zfs_name_to_prop(propname)) == ZPROP_INVALID) {
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         }
1553         (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
1554         (void) strncpy(zc.zc_value, propname, sizeof(zc.zc_value));
1556         if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_INHERIT_PROP, &zc) != 0)
1557             return (zfs_standard_error(hdl, errno, errbuf));
1559         return (0);
1560     }
1562     /*
1563     * Verify that this property is inheritable.
1564     */
1565     if (zfs_prop_readonly(prop))
1566         return (zfs_error(hdl, EZFS_PROPREADONLY, errbuf));
1568     if (!zfs_prop_inheritable(prop) && !received)
1569         return (zfs_error(hdl, EZFS_PROPNONINHERIT, errbuf));
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) strncpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
1582     (void) strncpy(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) strcpy(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;
1845
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;
1861
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     }
1870
1871     return (0);
1872 }
1873
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;
1883
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) strncpy(statbuf, source, statlen);
1895             *srctype = ZPROP_SRC_INHERITED;
1896         }
1897     }
1898 }
1899
1900 int
1901 zfs_prop_get_recvd(zfs_handle_t *zhp, const char *propname, char *propbuf,
1902                  size_t proplen, boolean_t literal)
1903 {
1904     zfs_prop_t prop;
1905     int err = 0;
1906
1907     if (zhp->zfs_recvd_props == NULL)
1908         if (get_recvd_props_ioctl(zhp) != 0)

```

```

1910         return (-1);
1911
1912     prop = zfs_name_to_prop(propname);
1913
1914     if (prop != ZPROP_INVALID) {
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) strncpy(propbuf, recvdval, proplen);
1931     }
1932
1933     return (err == 0 ? 0 : -1);
1934 }
1935
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;
1941
1942     value = zfs_get_clones_nvlist(zhp);
1943     if (value == NULL)
1944         return (-1);
1945
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     }
1953
1954     return (0);
1955 }
1956
1957 struct get_clones_arg {
1958     uint64_t numclones;
1959     nvlist_t *value;
1960     const char *origin;
1961     char buf[ZFS_MAXNAMELEN];
1962 };
1963
1964 int
1965 get_clones_cb(zfs_handle_t *zhp, void *arg)
1966 {
1967     struct get_clones_arg *gca = arg;
1968
1969     if (gca->numclones == 0) {
1970         zfs_close(zhp);
1971         return (0);
1972     }
1973
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         nvlist_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_nvlist(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) strncpy(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;
2113
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         }
2128
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++;
2140
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++;
2149
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     }
2161
2162     break;
2163
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;
2179
2180 case ZFS_PROP_QUOTA:
2181 case ZFS_PROP_REFQUOTA:
2182 case ZFS_PROP_RESERVATION:
2183 case ZFS_PROP_REFRESERVATION:
2184
2185     if (get_numeric_property(zhp, prop, src, &source, &val) != 0)
2186         return (-1);
2187
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;
2207
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.%02lux",
2213         (u_longlong_t)(val / 100),
2214         (u_longlong_t)(val % 100));
2215     break;
2216
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;
2233
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;
2249
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;
2258
2259 case ZFS_PROP_MLSLABEL:
2260     {
2261         m_label_t *new_sl = NULL;
2262         char *ascii = NULL; /* human readable label */
2263
2264         (void) strlcpy(propbuf,
2265             getprop_string(zhp, prop, &source), proplen);
2266
2267         if (literal || (strcasecmp(propbuf,
2268             ZFS_MLSLABEL_DEFAULT) == 0))
2269             break;
2270
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          */
2276
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         }
2282
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);
2291
2292         (void) strlcpy(propbuf, ascii, proplen);
2293         free(ascii);
2294     }
2295     break;
2296
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;
2307
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;
2320
2321             case PROP_TYPE_STRING:
2322                 (void) strlcpy(propbuf,
2323                     getprop_string(zhp, prop, &source), proplen);
2324                 break;
2325
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;
2334
2335             default:
2336                 abort();
2337         }
2338     }
2339
2340     get_source(zhp, src, source, statbuf, statlen);
2341
2342     return (0);
2343 }
2344
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;
2355
2356     (void) get_numeric_property(zhp, prop, NULL, &source, &val);
2357
2358     return (val);
2359 }
2360
2361 int
2362 zfs_prop_set_int(zfs_handle_t *zhp, zfs_prop_t prop, uint64_t val)
2363 {
2364     char buf[64];
2365
2366     (void) snprintf(buf, sizeof (buf), "%llu", (longlong_t)val);
2367     return (zfs_prop_set(zhp, zfs_prop_to_name(prop), buf));
2368 }
2369
2370 /*
2371 * Similar to zfs_prop_get(), but returns the value as an integer.

```

```

2372 */
2373 int
2374 zfs_prop_get_numeric(zfs_handle_t *zhp, zfs_prop_t prop, uint64_t *value,
2375 zprop_source_t *src, char *statbuf, size_t statlen)
2376 {
2377     char *source;
2378
2379     /*
2380      * Check to see if this property applies to our object
2381      */
2382     if (!zfs_prop_valid_for_type(prop, zhp->zfs_type)) {
2383         return (zfs_error_fmt(zhp->zfs_hdl, EZFS_PROPTYPE,
2384             dgettext(TEXT_DOMAIN, "cannot get property '%s'"),
2385             zfs_prop_to_name(prop)));
2386     }
2387
2388     if (src)
2389         *src = ZPROP_SRC_NONE;
2390
2391     if (get_numeric_property(zhp, prop, src, &source, value) != 0)
2392         return (-1);
2393
2394     get_source(zhp, src, source, statbuf, statlen);
2395
2396     return (0);
2397 }
2398
2399 static int
2400 idmap_id_to_numeric_domain_rid(uid_t id, boolean_t isuser,
2401 char **domainp, idmap_rid_t *ridp)
2402 {
2403     idmap_get_handle_t *get_hdl = NULL;
2404     idmap_stat status;
2405     int err = EINVAL;
2406
2407     if (idmap_get_create(&get_hdl) != IDMAP_SUCCESS)
2408         goto out;
2409
2410     if (isuser) {
2411         err = idmap_get_sidbyuid(get_hdl, id,
2412             IDMAP_REQ_FLG_USE_CACHE, domainp, ridp, &status);
2413     } else {
2414         err = idmap_get_sidbygid(get_hdl, id,
2415             IDMAP_REQ_FLG_USE_CACHE, domainp, ridp, &status);
2416     }
2417     if (err == IDMAP_SUCCESS &&
2418         idmap_get_mappings(get_hdl) == IDMAP_SUCCESS &&
2419         status == IDMAP_SUCCESS)
2420         err = 0;
2421     else
2422         err = EINVAL;
2423 out:
2424     if (get_hdl)
2425         idmap_get_destroy(get_hdl);
2426     return (err);
2427 }
2428
2429 /*
2430  * convert the propname into parameters needed by kernel
2431  * Eg: userquota@ahrens -> ZFS_PROP_USERQUOTA, "", 126829
2432  * Eg: userused@matt@domain -> ZFS_PROP_USERUSED, "S-1-123-456", 789
2433  */
2434 static int
2435 userquota_propname_decode(const char *propname, boolean_t zoned,
2436 zfs_userquota_prop_t *typep, char *domain, int domainlen, uint64_t *ridp)
2437 {

```

```

2438     zfs_userquota_prop_t type;
2439     char *cp, *end;
2440     char *numeric_sid = NULL;
2441     boolean_t isuser;
2442
2443     domain[0] = '\0';
2444
2445     /* Figure out the property type ({user|group}{quota|space}) */
2446     for (type = 0; type < ZFS_NUM_USERQUOTA_PROPS; type++) {
2447         if (strncmp(propname, zfs_userquota_prop_prefixes[type],
2448             strlen(zfs_userquota_prop_prefixes[type])) == 0)
2449             break;
2450     }
2451     if (type == ZFS_NUM_USERQUOTA_PROPS)
2452         return (EINVAL);
2453     *typep = type;
2454
2455     isuser = (type == ZFS_PROP_USERQUOTA ||
2456         type == ZFS_PROP_USERUSED);
2457
2458     cp = strchr(propname, '@') + 1;
2459
2460     if (strchr(cp, '@')) {
2461         /*
2462          * It's a SID name (eg "user@domain") that needs to be
2463          * turned into S-1-domainID-RID.
2464          */
2465         directory_error_t e;
2466         if (zoned && getzoneid() == GLOBAL_ZONEID)
2467             return (ENOENT);
2468         if (isuser) {
2469             e = directory_sid_from_user_name(NULL,
2470                 cp, &numeric_sid);
2471         } else {
2472             e = directory_sid_from_group_name(NULL,
2473                 cp, &numeric_sid);
2474         }
2475         if (e != NULL) {
2476             directory_error_free(e);
2477             return (ENOENT);
2478         }
2479         if (numeric_sid == NULL)
2480             return (ENOENT);
2481         cp = numeric_sid;
2482         /* will be further decoded below */
2483     }
2484
2485     if (strncmp(cp, "S-1-", 4) == 0) {
2486         /* It's a numeric SID (eg "S-1-234-567-89") */
2487         (void) strncpy(domain, cp, domainlen);
2488         cp = strchr(domain, '-');
2489         *cp = '\0';
2490         cp++;
2491
2492         errno = 0;
2493         *ridp = strtoull(cp, &end, 10);
2494         if (numeric_sid) {
2495             free(numeric_sid);
2496             numeric_sid = NULL;
2497         }
2498         if (errno != 0 || *end != '\0')
2499             return (EINVAL);
2500     } 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;
2525
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 }
2539
2540 ASSERT3P(numericid, ==, NULL);
2541 return (0);
2542 }
2543
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 };
2550
2551     (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
2552
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);
2559
2560     err = ioctl(zhp->zfs_hdl->libzfs_fd, ZFS_IOC_USERSPACE_ONE, &zc);
2561     if (err)
2562         return (err);
2563
2564     *propvalue = zc.zc_cookie;
2565     return (0);
2566 }
2567
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;
2573
2574     return (zfs_prop_get_userquota_common(zhp, propname, propvalue,
2575         &type));
2576 }
2577
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;
2585
2586     err = zfs_prop_get_userquota_common(zhp, propname, &propvalue,
2587         &type);
2588
2589     if (err)
2590         return (err);
2591
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 }
2602
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;
2610
2611     (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
2612
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;
2619
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     }
2628
2629     err = ioctl(zhp->zfs_hdl->libzfs_fd, ZFS_IOC_SPACE_WRITTEN, &zc);
2630     if (err)
2631         return (err);
2632
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) strncpy(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;
2909
2910     if (check_parents(hdl, path, NULL, B_TRUE, &prefix) != 0)
2911         return (-1);
2912
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);
2919
2920     return (0);
2921 }
2922
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     dmuf_objset_type_t ost;
2936
2937     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2938 "cannot create '%s'"), path);
2939
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));
2943
2944     /* validate parents exist */
2945     if (check_parents(hdl, path, &zoned, B_FALSE, NULL) != 0)
2946         return (-1);
2947
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     }
2960
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);
2969
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         }
2985
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         }
2999
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         }
3006
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     }
3015
3016     /* create the dataset */
3017     ret = lz_create(path, ost, props);
3018     nvlist_free(props);
3019
3020     /* check for failure */
3021     if (ret != 0) {
3022         char parent[ZFS_MAXNAMELEN];
3023         (void) parent_name(path, parent, sizeof (parent));
3024
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));
3030
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) strcpy(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(zhp->zfs_hdl, &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 = lzcd_destroy_snaps(snaps, defer, &errlist);

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

```



```

3164     if (nvlist_next_nvpair(errrlist, NULL) == NULL) {
3165         char errbuf[1024];
3166         (void) snprintf(errbuf, sizeof (errbuf),
3167             dgettext(TEXT_DOMAIN, "cannot destroy snapshots"));
3168
3169         ret = zfs_standard_error(hdl, ret, errbuf);
3170     }
3171     for (nvpair_t *pair = nvlist_next_nvpair(errrlist, NULL);
3172          pair != NULL; pair = nvlist_next_nvpair(errrlist, pair)) {
3173         char errbuf[1024];
3174         (void) snprintf(errbuf, sizeof (errbuf),
3175             dgettext(TEXT_DOMAIN, "cannot destroy snapshot %s"),
3176             nvpair_name(pair));
3177
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     }
3189
3190     return (ret);
3191 }
3192
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;
3204
3205     assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
3206
3207     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3208         "cannot create '%s'"), target);
3209
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));
3213
3214     /* validate parents exist */
3215     if (check_parents(hdl, target, &zoned, B_FALSE, NULL) != 0)
3216         return (-1);
3217
3218     (void) parent_name(target, parent, sizeof (parent));
3219
3220     /* do the clone */
3221
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     }
3233
3234     ret = lzc_clone(target, zhp->zfs_name, props);
3235     nvlist_free(props);
3236
3237     if (ret != 0) {
3238         switch (errno) {
3239
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));
3253
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));
3259
3260             default:
3261                 return (zfs_standard_error(zhp->zfs_hdl, errno,
3262                     errbuf));
3263         }
3264     }
3265
3266     return (ret);
3267 }
3268
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];
3280
3281     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3282         "cannot promote '%s'", zhp->zfs_name);
3283
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     }
3289
3290     (void) strncpy(parent, zhp->zfs_dmstats.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) strncpy(zc.zc_value, zhp->zfs_dmustats.dds_origin,
3298                 sizeof (zc.zc_value));
3299     (void) strncpy(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_nvlist;
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_nvlist, 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_nvlist(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) strncpy(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_nvlist, NV_UNIQUE_NAME, 0) == 0);
3439     if (recursive) {
3440         (void) zfs_snapshot_cb(zfs_handle_dup(zhp), &sd);
3441     } else {
3442         nvlist_add_boolean(sd.sd_nvlist, path);
3443     }

3445     ret = zfs_snapshot_nvlist(hdl, sd.sd_nvlist, props);
3446     nvlist_free(sd.sd_nvlist);
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) strncpy(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 *zhdp = 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) strncpy(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) strcat(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 (strcmp(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 (strcmp(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 }

```

```

3692     if (recursive) {
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         }
3707     } else {
3708         if ((cl = changelist_gather(zhp, ZFS_PROP_NAME, 0,
3709             force_unmount ? MS_FORCE : 0)) == NULL)
3710             return (-1);
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         }
3720         if ((ret = changelist_prefix(cl)) != 0)
3721             goto error;
3722     }
3724     if (ZFS_IS_VOLUME(zhp))
3725         zc.zc_objset_type = DMU_OST_ZVOL;
3726     else
3727         zc.zc_objset_type = DMU_OST_ZFS;
3729     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
3730     (void) strncpy(zc.zc_value, target, sizeof(zc.zc_value));
3732     zc.zc_cookie = recursive;
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);
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         }
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     }
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 }
3777 nvlist_t *
3778 zfs_get_user_props(zfs_handle_t *zhp)
3779 {
3780     return (zhp->zfs_user_props);
3781 }
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 }
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];
3816     if (zprop_expand_list(hdl, plp, ZFS_TYPE_DATASET) != 0)
3817         return (-1);
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     }
3834
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         }
3846
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             }
3855
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 }
3863
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;
3870
3871     if (entry->pl_prop != ZPROP_INVAL) {
3872         if (zfs_prop_get(zhp, entry->pl_prop,
3873                          buf, sizeof (buf), 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     }

```

```

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 }
3897
3898     return (0);
3899 }
3900
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;
3908
3909     (void) strncpy(zc.zc_name, dataset, sizeof (zc.zc_name));
3910     (void) strncpy(zc.zc_value, path, sizeof (zc.zc_value));
3911     if (resource)
3912         (void) strncpy(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 }
3920
3921 void
3922 zfs_prune_proplist(zfs_handle_t *zhp, uint8_t *props)
3923 {
3924     nvpair_t *curr;
3925
3926     /*
3927      * Keep a reference to the props-table against which we prune the
3928      * properties.
3929      */
3930     zhp->zfs_props_table = props;
3931
3932     curr = nvlist_next_nvpair(zhp->zfs_props, NULL);
3933
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);
3937
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 }
3951
3952 static int
3953 zfs_smb_acl_mgmt(libzfs_handle_t *hdl, char *dataset, char *path,
3954                  zfs_smb_acl_op_t cmd, char *resource1, char *resource2)
3955 {

```

```

3956     zfs_cmd_t zc = { 0 };
3957     nvlist_t *nvlist = NULL;
3958     int error;

3960     (void) strlcpy(zc.zc_name, dataset, sizeof (zc.zc_name));
3961     (void) strlcpy(zc.zc_value, path, sizeof (zc.zc_value));
3962     zc.zc_cookie = (uint64_t)cmd;

3964     if (cmd == ZFS_SMB_ACL_RENAME) {
3965         if (nvlist_alloc(&nvlist, NV_UNIQUE_NAME, 0) != 0) {
3966             (void) no_memory(hdl);
3967             return (NULL);
3968         }
3969     }

3971     switch (cmd) {
3972     case ZFS_SMB_ACL_ADD:
3973     case ZFS_SMB_ACL_REMOVE:
3974         (void) strlcpy(zc.zc_string, resource1, sizeof (zc.zc_string));
3975         break;
3976     case ZFS_SMB_ACL_RENAME:
3977         if (nvlist_add_string(nvlist, ZFS_SMB_ACL_SRC,
3978             resource1) != 0) {
3979             (void) no_memory(hdl);
3980             return (-1);
3981         }
3982         if (nvlist_add_string(nvlist, ZFS_SMB_ACL_TARGET,
3983             resource2) != 0) {
3984             (void) no_memory(hdl);
3985             return (-1);
3986         }
3987         if (zcmd_write_src_nvlist(hdl, &zc, nvlist) != 0) {
3988             nvlist_free(nvlist);
3989             return (-1);
3990         }
3991         break;
3992     case ZFS_SMB_ACL_PURGE:
3993         break;
3994     default:
3995         return (-1);
3996     }
3997     error = ioctl(hdl->libzfs_fd, ZFS_IOC_SMB_ACL, &zc);
3998     if (nvlist)
3999         nvlist_free(nvlist);
4000     return (error);
4001 }

4003 int
4004 zfs_smb_acl_add(libzfs_handle_t *hdl, char *dataset,
4005     char *path, char *resource)
4006 {
4007     return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_ADD,
4008     resource, NULL));
4009 }

4011 int
4012 zfs_smb_acl_remove(libzfs_handle_t *hdl, char *dataset,
4013     char *path, char *resource)
4014 {
4015     return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_REMOVE,
4016     resource, NULL));
4017 }

4019 int
4020 zfs_smb_acl_purge(libzfs_handle_t *hdl, char *dataset, char *path)
4021 {

```

```

4022     return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_PURGE,
4023     NULL, NULL));
4024 }

4026 int
4027 zfs_smb_acl_rename(libzfs_handle_t *hdl, char *dataset, char *path,
4028     char *oldname, char *newname)
4029 {
4030     return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_RENAME,
4031     oldname, newname));
4032 }

4034 int
4035 zfs_userspace(zfs_handle_t *zhp, zfs_userquota_prop_t type,
4036     zfs_userspace_cb_t func, void *arg)
4037 {
4038     zfs_cmd_t zc = { 0 };
4039     zfs_useracct_t buf[100];
4040     libzfs_handle_t *hdl = zhp->zfs_hdl;
4041     int ret;

4043     (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

4045     zc.zc_objset_type = type;
4046     zc.zc_nvlist_dst = (uintptr_t)buf;

4048     for (;;) {
4049         zfs_useracct_t *zua = buf;

4051         zc.zc_nvlist_dst_size = sizeof (buf);
4052         if (zfs_ioctl(hdl, ZFS_IOC_USERSPACE_MANY, &zc) != 0) {
4053             char errbuf[1024];

4055             (void) snprintf(errbuf, sizeof (errbuf),
4056                 dgettext(TEXT_DOMAIN,
4057                     "cannot get used/quota for %s"), zc.zc_name);
4058             return (zfs_standard_error_fmt(hdl, errno, errbuf));
4059         }
4060         if (zc.zc_nvlist_dst_size == 0)
4061             break;

4063         while (zc.zc_nvlist_dst_size > 0) {
4064             if ((ret = func(arg, zua->zu_domain, zua->zu_rid,
4065                 zua->zu_space)) != 0)
4066                 return (ret);
4067             zua++;
4068             zc.zc_nvlist_dst_size -= sizeof (zfs_useracct_t);
4069         }
4070     }

4072     return (0);
4073 }

4075 struct holdarg {
4076     nvlist_t *nvl;
4077     const char *snapname;
4078     const char *tag;
4079     boolean_t recursive;
4080 };

4082 static int
4083 zfs_hold_one(zfs_handle_t *zhp, void *arg)
4084 {
4085     struct holdarg *ha = arg;
4086     zfs_handle_t *szhp;
4087     char name[ZFS_MAXNAMELEN];

```

```

4088     int rv = 0;
4090     (void) snprintf(name, sizeof (name),
4091         "%s%s", zhp->zfs_name, ha->snapname);
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     }
4099     if (ha->recursive)
4100         rv = zfs_iter_filesystems(zhp, zfs_hold_one, ha);
4101     zfs_close(zhp);
4102     return (rv);
4103 }
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;
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);
4122     if (nvlist_next_nvpair(ha.nvl, NULL) == NULL) {
4123         fnvlist_free(ha.nvl);
4124         ret = ENOENT;
4125         (void) snprintf(errbuf, sizeof (errbuf),
4126             dgettext(TEXT_DOMAIN, "cannot hold snapshot '%s%s'"),
4127             zhp->zfs_name, snapname);
4128         (void) zfs_standard_error(hdl, ret, errbuf);
4129         return (ret);
4130     }
4132 #endif /* ! codereview */
4133     ret = lzc_hold(ha.nvl, cleanup_fd, &errors);
4134     fnvlist_free(ha.nvl);
4136     if (ret == 0)
4137         return (0);
4139     if (nvlist_next_nvpair(errors, NULL) == NULL) {
4140         /* no hold-specific errors */
4141         (void) snprintf(errbuf, sizeof (errbuf),
4142             dgettext(TEXT_DOMAIN, "cannot hold"));
4143         switch (ret) {
4144             case ENOTSUP:
4145                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4146                     "pool must be upgraded"));
4147                 (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4148                 break;
4149             case EINVAL:
4150                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4151                 break;
4152             default:
4153                 (void) zfs_standard_error(hdl, ret, errbuf);

```

```

4154     }
4155 }
4157     for (elem = nvlist_next_nvpair(errors, NULL);
4158         elem != NULL;
4159         elem = nvlist_next_nvpair(errors, elem)) {
4160         (void) snprintf(errbuf, sizeof (errbuf),
4161             dgettext(TEXT_DOMAIN,
4162                 "cannot hold snapshot '%s'", nvpair_name(elem));
4163         switch (fnvpair_value_int32(elem)) {
4164             case E2BIG:
4165                 /*
4166                  * Temporary tags wind up having the ds object id
4167                  * prepended. So even if we passed the length check
4168                  * above, it's still possible for the tag to wind
4169                  * up being slightly too long.
4170                  */
4171                 (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
4172                 break;
4173             case EINVAL:
4174                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4175                 break;
4176             case EEXIST:
4177                 (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
4178                 break;
4179             case ENOENT:
4180                 if (enoent_ok)
4181                     return (ENOENT);
4182                 /* FALLTHROUGH */
4183             default:
4184                 (void) zfs_standard_error(hdl,
4185                     fnvpair_value_int32(elem), errbuf);
4186         }
4187     }
4189     fnvlist_free(errors);
4190     return (ret);
4191 }
4193 struct releasearg {
4194     nvlist_t *nvl;
4195     const char *snapname;
4196     const char *tag;
4197     boolean_t recursive;
4198 };
4200 static int
4201 zfs_release_one(zfs_handle_t *zhp, void *arg)
4202 {
4203     struct holdarg *ha = arg;
4204     zfs_handle_t *szhp;
4205     char name[ZFS_MAXNAMELEN];
4206     int rv = 0;
4208     (void) snprintf(name, sizeof (name),
4209         "%s%s", zhp->zfs_name, ha->snapname);
4211     szhp = make_dataset_handle(zhp->zfs_hdl, name);
4212     if (szhp) {
4213         nvlist_t *holds = fnvlist_alloc();
4214         fnvlist_add_boolean(holds, ha->tag);
4215         fnvlist_add_nvlist(ha->nvl, name, holds);
4216         zfs_close(szhp);
4217     }
4219     if (ha->recursive)

```



```

4220         rv = zfs_iter_filesystems(zhp, zfs_release_one, ha);
4221         zfs_close(zhp);
4222         return (rv);
4223     }

4225 int
4226 zfs_release(zfs_handle_t *zhp, const char *snapname, const char *tag,
4227             boolean_t recursive)
4228 {
4229     int ret;
4230     struct holdarg ha;
4231     nvlist_t *errors;
4232     nvpair_t *elem;
4233     libzfs_handle_t *hdl = zhp->zfs_hdl;
4234     char errbuf[1024];
4235 #endif /* ! codereview */

4237     ha.nvl = fnvlist_alloc();
4238     ha.snapname = snapname;
4239     ha.tag = tag;
4240     ha.recursive = recursive;
4241     (void) zfs_release_one(zfs_handle_dup(zhp), &ha);

4243     if (nvlist_next_nvpair(ha.nvl, NULL) == NULL) {
4244         fnvlist_free(ha.nvl);
4245         ret = ENOENT;
4246         (void) snprintf(errbuf, sizeof (errbuf),
4247             dgettext(TEXT_DOMAIN,
4248                 "cannot release hold from snapshot '%s%s'",
4249                 zhp->zfs_name, snapname);
4250         (void) zfs_standard_error(hdl, ret, errbuf);
4251         return (ret);
4252     }

4254 #endif /* ! codereview */
4255     ret = lzc_release(ha.nvl, &errors);
4256     fnvlist_free(ha.nvl);

4258     if (ret == 0)
4259         return (0);

4261     if (nvlist_next_nvpair(errors, NULL) == NULL) {
4262         /* no hold-specific errors */
4263         char errbuf[1024];
4264         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
4265             "cannot release"));
4266         switch (errno) {
4267             case ENOTSUP:
4268                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4269                     "pool must be upgraded"));
4270                 (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4271                 break;
4272             default:
4273                 (void) zfs_standard_error_fmt(hdl, errno, errbuf);
4274         }

4276     for (elem = nvlist_next_nvpair(errors, NULL);
4277          elem != NULL;
4278          elem = nvlist_next_nvpair(errors, elem)) {
4279         char errbuf[1024];
4280         (void) snprintf(errbuf, sizeof (errbuf),
4281             dgettext(TEXT_DOMAIN,

```

```

4282         nvpair_name(elem));
4283         switch (fnvpair_value_int32(elem)) {
4284             case ESRCH:
4285                 (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
4286                 break;
4287             case EINVAL:
4288                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4289                 break;
4290             default:
4291                 (void) zfs_standard_error_fmt(hdl,
4292                     fnvpair_value_int32(elem), errbuf);
4293         }
4294     }

4296     fnvlist_free(errors);
4297     return (ret);
4298 }

```

unchanged\_portion\_omitted