

```

*****
8538 Wed May 29 20:27:07 2013
new/usr/src/cmd/ndmpd/ndmp/ndmpd_chkpnt.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
1 /*
2  * Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
3  * Copyright (c) 2013 by Delphix. All rights reserved.
4  * Copyright (c) 2013 Steven Hartland. All rights reserved.
5  #endif /* !codereview */
6  */

8 /*
9  * BSD 3 Clause License
10 *
11 * Copyright (c) 2007, The Storage Networking Industry Association.
12 *
13 * Redistribution and use in source and binary forms, with or without
14 * modification, are permitted provided that the following conditions
15 * are met:
16 *   - Redistributions of source code must retain the above copyright
17 *     notice, this list of conditions and the following disclaimer.
18 *
19 *   - Redistributions in binary form must reproduce the above copyright
20 *     notice, this list of conditions and the following disclaimer in
21 *     the documentation and/or other materials provided with the
22 *     distribution.
23 *
24 *   - Neither the name of The Storage Networking Industry Association (SNIA)
25 *     nor the names of its contributors may be used to endorse or promote
26 *     products derived from this software without specific prior written
27 *     permission.
28 *
29 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
30 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
31 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
32 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
33 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
34 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
35 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
36 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
37 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
38 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
39 * POSSIBILITY OF SUCH DAMAGE.
40 */

42 #include <stdio.h>
43 #include <string.h>
44 #include "ndmpd.h"
45 #include <libzfs.h>

47 typedef struct snap_param {
48     char *snp_name;
49     boolean_t snp_found;
50 } snap_param_t;

52 static int cleanup_fd = -1;

54 /*
55  * ndmp_has_backup
56  *
57  * Call backup function which looks for backup snapshot.
58  * This is a callback function used with zfs_iter_snapshots.
59  *
60  * Parameters:

```

```

61  * zhp (input) - ZFS handle pointer
62  * data (output) - 0 - no backup snapshot
63  *                 1 - has backup snapshot
64  *
65  * Returns:
66  *   0: on success
67  *  -1: otherwise
68  */
69 static int
70 ndmp_has_backup(zfs_handle_t *zhp, void *data)
71 {
72     const char *name;
73     snap_param_t *chp = (snap_param_t *)data;

75     name = zfs_get_name(zhp);
76     if (name == NULL ||
77         strstr(name, chp->snp_name) == NULL) {
78         zfs_close(zhp);
79         return (-1);
80     }

82     chp->snp_found = 1;
83     zfs_close(zhp);

85     return (0);
86 }

88 /*
89  * ndmp_has_backup_snapshot
90  *
91  * Returns TRUE if the volume has an active backup snapshot, otherwise,
92  * returns FALSE.
93  *
94  * Parameters:
95  *   volname (input) - name of the volume
96  *
97  * Returns:
98  *   0: on success
99  *  -1: otherwise
100 */
101 static int
102 ndmp_has_backup_snapshot(char *volname, char *jobname)
103 {
104     zfs_handle_t *zhp;
105     snap_param_t snp;
106     char cname[ZFS_MAXNAMELEN];

108     (void) mutex_lock(&zlib_mtx);
109     if ((zhp = zfs_open(zlib, volname, ZFS_TYPE_DATASET)) == 0) {
110         NDMP_LOG(LOG_ERR, "Cannot open snapshot %s.", volname);
111         (void) mutex_unlock(&zlib_mtx);
112         return (-1);
113     }

115     snp.snp_found = 0;
116     (void) snprintf(cname, ZFS_MAXNAMELEN, "@%s", jobname);
117     snp.snp_name = cname;

119     (void) zfs_iter_snapshots(zhp, ndmp_has_backup, &snp);
120     zfs_close(zhp);
121     (void) mutex_unlock(&zlib_mtx);

123     return (snp.snp_found);
124 }

126 /*

```

```

127 * ndmp_create_snapshot
128 *
129 * This function will parse the path to get the real volume name.
130 * It will then create a snapshot based on volume and job name.
131 * This function should be called before the NDMP backup is started.
132 *
133 * Parameters:
134 *   vol_name (input) - name of the volume
135 *
136 * Returns:
137 *   0: on success
138 *  -1: otherwise
139 */
140 int
141 ndmp_create_snapshot(char *vol_name, char *jname)
142 {
143     char vol[ZFS_MAXNAMELEN];
144
145     if (vol_name == 0 ||
146         get_zfsvolname(vol, sizeof (vol), vol_name) == -1)
147         return (0);
148
149     /*
150      * If there is an old snapshot left from the previous
151      * backup it could be stale one and it must be
152      * removed before using it.
153      */
154     if (ndmp_has_backup_snapshot(vol, jname))
155         (void) snapshot_destroy(vol, jname, B_FALSE, B_TRUE, NULL);
156
157     return (snapshot_create(vol, jname, B_FALSE, B_TRUE));
158 }
159
160 /*
161 * ndmp_remove_snapshot
162 *
163 * This function will parse the path to get the real volume name.
164 * It will then remove the snapshot for that volume and job name.
165 * This function should be called after NDMP backup is finished.
166 *
167 * Parameters:
168 *   vol_name (input) - name of the volume
169 *
170 * Returns:
171 *   0: on success
172 *  -1: otherwise
173 */
174 int
175 ndmp_remove_snapshot(char *vol_name, char *jname)
176 {
177     char vol[ZFS_MAXNAMELEN];
178
179     if (vol_name == 0 ||
180         get_zfsvolname(vol, sizeof (vol), vol_name) == -1)
181         return (0);
182
183     return (snapshot_destroy(vol, jname, B_FALSE, B_TRUE, NULL));
184 }
185
186 /*
187 * Put a hold on snapshot
188 */
189 int
190 snapshot_hold(char *volname, char *snapname, char *jname, boolean_t recursive)
191 {
192     zfs_handle_t *zhp;

```

```

193     char *p;
194
195     if ((zhp = zfs_open(zlibh, volname, ZFS_TYPE_DATASET)) == 0) {
196         NDMP_LOG(LOG_ERR, "Cannot open volume %s.", volname);
197         return (-1);
198     }
199
200     if (cleanup_fd == -1 && (cleanup_fd = open(ZFS_DEV,
201         O_RDWR|O_EXCL)) < 0) {
202         NDMP_LOG(LOG_ERR, "Cannot open dev %d", errno);
203         zfs_close(zhp);
204         return (-1);
205     }
206
207     p = strchr(snapname, '@') + 1;
208     if (zfs_hold(zhp, p, jname, recursive, cleanup_fd) != 0) {
209         if (zfs_hold(zhp, p, jname, recursive, B_FALSE, cleanup_fd) != 0) {
210             NDMP_LOG(LOG_ERR, "Cannot hold snapshot %s", p);
211             zfs_close(zhp);
212             return (-1);
213         }
214         zfs_close(zhp);
215     }
216 }
217
218 unchanged_portion_omitted

```

```

*****
161560 Wed May 29 20:27:07 2013
new/usr/src/cmd/zfs/zfs_main.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
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 2012 Nexenta Systems, Inc. All rights reserved.
25 * Copyright (c) 2012 by Delphix. All rights reserved.
26 * Copyright 2012 Milan Jurik. All rights reserved.
27 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
28 * Copyright (c) 2013 Steven Hartland. All rights reserved.
29 #endif /* !codereview */
30 */

32 #include <assert.h>
33 #include <ctype.h>
34 #include <errno.h>
35 #include <libgen.h>
36 #include <libintl.h>
37 #include <libutil.h>
38 #include <libnvpair.h>
39 #include <locale.h>
40 #include <stddef.h>
41 #include <stdio.h>
42 #include <stdlib.h>
43 #include <strings.h>
44 #include <unistd.h>
45 #include <fcntl.h>
46 #include <zone.h>
47 #include <grp.h>
48 #include <pwd.h>
49 #include <signal.h>
50 #include <sys/list.h>
51 #include <sys/mkdev.h>
52 #include <sys/mntent.h>
53 #include <sys/mnttab.h>
54 #include <sys/mount.h>
55 #include <sys/stat.h>
56 #include <sys/fs/zfs.h>
57 #include <sys/types.h>
58 #include <time.h>

60 #include <libzfs.h>

```

```

61 #include <libzfs_core.h>
62 #include <zfs_prop.h>
63 #include <zfs_deleg.h>
64 #include <libutil.h>
65 #include <aclutils.h>
66 #include <directory.h>

68 #include "zfs_iter.h"
69 #include "zfs_util.h"
70 #include "zfs_comutil.h"

72 libzfs_handle_t *g_zfs;

74 static FILE *mnttab_file;
75 static char history_str[HIS_MAX_RECORD_LEN];
76 static boolean_t log_history = B_TRUE;

78 static int zfs_do_clone(int argc, char **argv);
79 static int zfs_do_create(int argc, char **argv);
80 static int zfs_do_destroy(int argc, char **argv);
81 static int zfs_do_get(int argc, char **argv);
82 static int zfs_do_inherit(int argc, char **argv);
83 static int zfs_do_list(int argc, char **argv);
84 static int zfs_do_mount(int argc, char **argv);
85 static int zfs_do_rename(int argc, char **argv);
86 static int zfs_do_rollback(int argc, char **argv);
87 static int zfs_do_set(int argc, char **argv);
88 static int zfs_do_upgrade(int argc, char **argv);
89 static int zfs_do_snapshot(int argc, char **argv);
90 static int zfs_do_unmount(int argc, char **argv);
91 static int zfs_do_share(int argc, char **argv);
92 static int zfs_do_unshare(int argc, char **argv);
93 static int zfs_do_send(int argc, char **argv);
94 static int zfs_do_receive(int argc, char **argv);
95 static int zfs_do_promote(int argc, char **argv);
96 static int zfs_do_userspace(int argc, char **argv);
97 static int zfs_do_allow(int argc, char **argv);
98 static int zfs_do_unallow(int argc, char **argv);
99 static int zfs_do_hold(int argc, char **argv);
100 static int zfs_do_holds(int argc, char **argv);
101 static int zfs_do_release(int argc, char **argv);
102 static int zfs_do_diff(int argc, char **argv);

104 /*
105  * Enable a reasonable set of defaults for libumem debugging on DEBUG builds.
106  */

108 #ifdef DEBUG
109 const char *
110 _umem_debug_init(void)
111 {
112     return ("default,verbose"); /* $UMEM_DEBUG setting */
113 }

115 const char *
116 _umem_logging_init(void)
117 {
118     return ("fail,contents"); /* $UMEM_LOGGING setting */
119 }
120 #endif

122 typedef enum {
123     HELP_CLONE,
124     HELP_CREATE,
125     HELP_DESTROY,
126     HELP_GET,

```

```

127     HELP_INHERIT,
128     HELP_UPGRADE,
129     HELP_LIST,
130     HELP_MOUNT,
131     HELP_PROMOTE,
132     HELP_RECEIVE,
133     HELP_RENAME,
134     HELP_ROLLBACK,
135     HELP_SEND,
136     HELP_SET,
137     HELP_SHARE,
138     HELP_SNAPSHOT,
139     HELP_UNMOUNT,
140     HELP_UNSHARE,
141     HELP_ALLOW,
142     HELP_UNALLOW,
143     HELP_USERSPACE,
144     HELP_GROUPSPACE,
145     HELP_HOLD,
146     HELP_HOLDS,
147     HELP_RELEASE,
148     HELP_DIFF,
149 } zfs_help_t;

151 typedef struct zfs_command {
152     const char    *name;
153     int           (*func)(int argc, char **argv);
154     zfs_help_t    usage;
155 } zfs_command_t;

157 /*
158  * Master command table. Each ZFS command has a name, associated function, and
159  * usage message. The usage messages need to be internationalized, so we have
160  * to have a function to return the usage message based on a command index.
161  *
162  * These commands are organized according to how they are displayed in the usage
163  * message. An empty command (one with a NULL name) indicates an empty line in
164  * the generic usage message.
165  */
166 static zfs_command_t command_table[] = {
167     {"create",      zfs_do_create,      HELP_CREATE},
168     {"destroy",    zfs_do_destroy,     HELP_DESTROY},
169     {NULL},
170     {"snapshot",  zfs_do_snapshot,     HELP_SNAPSHOT},
171     {"rollback",  zfs_do_rollback,     HELP_ROLLBACK},
172     {"clone",     zfs_do_clone,        HELP_CLONE},
173     {"promote",   zfs_do_promote,      HELP_PROMOTE},
174     {"rename",    zfs_do_rename,       HELP_RENAME},
175     {NULL},
176     {"list",      zfs_do_list,         HELP_LIST},
177     {NULL},
178     {"set",       zfs_do_set,          HELP_SET},
179     {"get",       zfs_do_get,          HELP_GET},
180     {"inherit",   zfs_do_inherit,      HELP_INHERIT},
181     {"upgrade",   zfs_do_upgrade,      HELP_UPGRADE},
182     {"userspace", zfs_do_userspace,    HELP_USERSPACE},
183     {"groupspace", zfs_do_groupspace,  HELP_GROUPSPACE},
184     {NULL},
185     {"mount",     zfs_do_mount,        HELP_MOUNT},
186     {"unmount",   zfs_do_unmount,      HELP_UNMOUNT},
187     {"share",     zfs_do_share,        HELP_SHARE},
188     {"unshare",   zfs_do_unshare,     HELP_UNSHARE},
189     {NULL},
190     {"send",      zfs_do_send,         HELP_SEND},
191     {"receive",   zfs_do_receive,      HELP_RECEIVE},
192     {NULL},

```

```

193     {"allow",     zfs_do_allow,        HELP_ALLOW},
194     {NULL},
195     {"unallow",   zfs_do_unallow,     HELP_UNALLOW},
196     {NULL},
197     {"hold",      zfs_do_hold,        HELP_HOLD},
198     {"holds",     zfs_do_holds,       HELP_HOLDS},
199     {"release",   zfs_do_release,     HELP_RELEASE},
200     {"diff",      zfs_do_diff,        HELP_DIFF},
201 };

203 #define NCOMMAND      (sizeof (command_table) / sizeof (command_table[0]))

205 zfs_command_t *current_command;

207 static const char *
208 get_usage(zfs_help_t idx)
209 {
210     switch (idx) {
211     case HELP_CLONE:
212         return (gettext("\tclone [-p] [-o property=value] ... "
213             "<snapshot> <filesystem|volume>\n"));
214     case HELP_CREATE:
215         return (gettext("\tcreate [-p] [-o property=value] ... "
216             "<filesystem>\n"
217             "\tcreate [-ps] [-b blocksize] [-o property=value] ... "
218             "-V <size> <volume>\n"));
219     case HELP_DESTROY:
220         return (gettext("\tdestroy [-fnpRrv] <filesystem|volume>\n"
221             "\tdestroy [-dnpRrv] "
222             "<filesystem|volume>@<snap>[%<snap>][,...]\n"));
223     case HELP_GET:
224         return (gettext("\tget [-rHp] [-d max] "
225             "[-o \"all\" | field,...] [-t type,...] "
226             "[-s source,...]\n"
227             "\t <\"all\" | property[,...]> "
228             "[filesystem|volume|snapshot] ... \n"));
229     case HELP_INHERIT:
230         return (gettext("\tinherit [-rS] <property> "
231             "<filesystem|volume|snapshot> ... \n"));
232     case HELP_UPGRADE:
233         return (gettext("\tupgrade [-v]\n"
234             "\tupgrade [-r] [-V version] <-a | filesystem ...>\n"));
235     case HELP_LIST:
236         return (gettext("\tlist [-rH] [-d max] "
237             "[-o property,...] [-t type,...] [-s property] ... \n"
238             "\t [-S property] ... "
239             "[filesystem|volume|snapshot] ... \n"));
240     case HELP_MOUNT:
241         return (gettext("\tmount\n"
242             "\tmount [-vO] [-o opts] <-a | filesystem>\n"));
243     case HELP_PROMOTE:
244         return (gettext("\tpromote <clone-filesystem>\n"));
245     case HELP_RECEIVE:
246         return (gettext("\treceive [-vnFu] <filesystem|volume| "
247             "snapshot>\n"
248             "\treceive [-vnFu] [-d | -e] <filesystem>\n"));
249     case HELP_RENAME:
250         return (gettext("\trename [-f] <filesystem|volume|snapshot> "
251             "<filesystem|volume|snapshot>\n"
252             "\trename [-f] -p <filesystem|volume> <filesystem|volume>\n"
253             "\trename -r <snapshot> <snapshot>));
254     case HELP_ROLLBACK:
255         return (gettext("\trollback [-rRf] <snapshot>\n"));
256     case HELP_SEND:
257         return (gettext("\tsend [-DnppRv] [-[iI] snapshot] "
258             "<snapshot>\n"));

```

```

259     case HELP_SET:
260         return (gettext("\tset <property=value> "
261             "<filesystem|volume|snapshot> ... \n"));
262     case HELP_SHARE:
263         return (gettext("\tshare <-a | filesystem>\n"));
264     case HELP_SNAPSHOT:
265         return (gettext("\tsnapshot [-r] [-o property=value] ... "
266             "<filesystem@snapname|volume@snapname> ... \n"));
267     case HELP_UNMOUNT:
268         return (gettext("\tunmount [-f] "
269             "<-a | filesystem|mountpoint>\n"));
270     case HELP_UNSHARE:
271         return (gettext("\tunshare "
272             "<-a | filesystem|mountpoint>\n"));
273     case HELP_ALLOW:
274         return (gettext("\tallow <filesystem|volume>\n"
275             "\tallow [-ldug] "
276             "<\\"everyone\"|user|group>[,...] <perm|setname>[,...] \n"
277             "\t    <filesystem|volume>\n"
278             "\tallow [-ld] -e <perm|setname>[,...] "
279             "<filesystem|volume>\n"
280             "\tallow -c <perm|setname>[,...] <filesystem|volume>\n"
281             "\tallow -s @setname <perm|setname>[,...] "
282             "<filesystem|volume>\n"));
283     case HELP_UNALLOW:
284         return (gettext("\tunallow [-rldug] "
285             "<\\"everyone\"|user|group>[,...] \n"
286             "\t    [<perm|setname>[,...]] <filesystem|volume>\n"
287             "\tunallow [-rld] -e [<perm|setname>[,...]] "
288             "<filesystem|volume>\n"
289             "\tunallow [-r] -c [<perm|setname>[,...]] "
290             "<filesystem|volume>\n"
291             "\tunallow [-r] -s @setname [<perm|setname>[,...]] "
292             "<filesystem|volume>\n"));
293     case HELP_USERSPACE:
294         return (gettext("\tuserspace [-Hinp] [-o field[,...]] "
295             "[-s field] ... \n\t[-S field] ... "
296             "[-t type[,...]] <filesystem|snapshot>\n"));
297     case HELP_GROUPSSPACE:
298         return (gettext("\tgroupspace [-Hinp] [-o field[,...]] "
299             "[-s field] ... \n\t[-S field] ... "
300             "[-t type[,...]] <filesystem|snapshot>\n"));
301     case HELP_HOLD:
302         return (gettext("\thold [-r] <tag> <snapshot> ... \n"));
303     case HELP_HOLDS:
304         return (gettext("\tholds [-r] <snapshot> ... \n"));
305     case HELP_RELEASE:
306         return (gettext("\trelease [-r] <tag> <snapshot> ... \n"));
307     case HELP_DIFF:
308         return (gettext("\tdiff [-FHT] <snapshot> "
309             "[<snapshot|filesystem>\n"));
310     }
311
312     abort();
313     /* NOTREACHED */
314 }
315
316 void
317 nomem(void)
318 {
319     (void) fprintf(stderr, gettext("internal error: out of memory\n"));
320     exit(1);
321 }
322
323 /*
324  * Utility function to guarantee malloc() success.

```

```

325 */
326
327 void *
328 safe_malloc(size_t size)
329 {
330     void *data;
331
332     if ((data = calloc(1, size)) == NULL)
333         nomem();
334
335     return (data);
336 }
337
338 static char *
339 safe_strdup(char *str)
340 {
341     char *dupstr = strdup(str);
342
343     if (dupstr == NULL)
344         nomem();
345
346     return (dupstr);
347 }
348
349 /*
350  * Callback routine that will print out information for each of
351  * the properties.
352  */
353 static int
354 usage_prop_cb(int prop, void *cb)
355 {
356     FILE *fp = cb;
357
358     (void) fprintf(fp, "\t%-15s ", zfs_prop_to_name(prop));
359
360     if (zfs_prop_readonly(prop))
361         (void) fprintf(fp, " NO ");
362     else
363         (void) fprintf(fp, " YES ");
364
365     if (zfs_prop_inheritable(prop))
366         (void) fprintf(fp, " YES ");
367     else
368         (void) fprintf(fp, " NO ");
369
370     if (zfs_prop_values(prop) == NULL)
371         (void) fprintf(fp, "-\n");
372     else
373         (void) fprintf(fp, "%s\n", zfs_prop_values(prop));
374
375     return (ZPROP_CONT);
376 }
377
378 /*
379  * Display usage message.  If we're inside a command, display only the usage for
380  * that command.  Otherwise, iterate over the entire command table and display
381  * a complete usage message.
382  */
383 static void
384 usage(boolean_t requested)
385 {
386     int i;
387     boolean_t show_properties = B_FALSE;
388     FILE *fp = requested ? stdout : stderr;
389
390     if (current_command == NULL) {

```

```

392     (void) fprintf(fp, gettext("usage: zfs command args ...\n"));
393     (void) fprintf(fp,
394         gettext("where 'command' is one of the following:\n\n"));
396     for (i = 0; i < NCOMMAND; i++) {
397         if (command_table[i].name == NULL)
398             (void) fprintf(fp, "\n");
399         else
400             (void) fprintf(fp, "%s",
401                 get_usage(command_table[i].usage));
402     }
404     (void) fprintf(fp, gettext("\nEach dataset is of the form: "
405         "pool/[dataset/]dataset[@name]\n"));
406 } else {
407     (void) fprintf(fp, gettext("usage:\n"));
408     (void) fprintf(fp, "%s", get_usage(current_command->usage));
409 }
411 if (current_command != NULL &&
412     (strcmp(current_command->name, "set") == 0 ||
413      strcmp(current_command->name, "get") == 0 ||
414      strcmp(current_command->name, "inherit") == 0 ||
415      strcmp(current_command->name, "list") == 0))
416     show_properties = B_TRUE;
418 if (show_properties) {
419     (void) fprintf(fp,
420         gettext("\nThe following properties are supported:\n"));
422     (void) fprintf(fp, "\n\t%-14s %s %s %s\n\n",
423         "PROPERTY", "EDIT", "INHERIT", "VALUES");
425     /* Iterate over all properties */
426     (void) zprop_iter(usage_prop_cb, fp, B_FALSE, B_TRUE,
427         ZFS_TYPE_DATASET);
429     (void) fprintf(fp, "\t%-15s ", "userused@...");
430     (void) fprintf(fp, " NO NO <size>\n");
431     (void) fprintf(fp, "\t%-15s ", "groupused@...");
432     (void) fprintf(fp, " NO NO <size>\n");
433     (void) fprintf(fp, "\t%-15s ", "userquota@...");
434     (void) fprintf(fp, "YES NO <size> | none\n");
435     (void) fprintf(fp, "\t%-15s ", "groupquota@...");
436     (void) fprintf(fp, "YES NO <size> | none\n");
437     (void) fprintf(fp, "\t%-15s ", "written<snap>");
438     (void) fprintf(fp, " NO NO <size>\n");
440     (void) fprintf(fp, gettext("\nSizes are specified in bytes "
441         "with standard units such as K, M, G, etc.\n"));
442     (void) fprintf(fp, gettext("\nUser-defined properties can "
443         "be specified by using a name containing a colon (:).\n"));
444     (void) fprintf(fp, gettext("\nThe {user|group}{used|quota}@ "
445         "properties must be appended with\n"
446         "a user or group specifier of one of these forms:\n"
447         "  POSIX name      (eg: \"matt\")\n"
448         "  POSIX id        (eg: \"126829\")\n"
449         "  SMB name@domain (eg: \"matt@sun\")\n"
450         "  SMB SID         (eg: \"S-1-234-567-89\")\n"));
451 } else {
452     (void) fprintf(fp,
453         gettext("\nFor the property list, run: %s\n"),
454         "zfs set|get");
455     (void) fprintf(fp,
456         gettext("\nFor the delegated permission list, run: %s\n"),

```

```

457         "zfs allow|unallow");
458     }
460     /*
461     * See comments at end of main().
462     */
463     if (getenv("ZFS_ABORT") != NULL) {
464         (void) printf("dumping core by request\n");
465         abort();
466     }
468     exit(requested ? 0 : 2);
469 }
471 static int
472 parseprop(nvlist_t *props)
473 {
474     char *propname = optarg;
475     char *propval, *strval;
477     if ((propval = strchr(propname, '=') == NULL) {
478         (void) fprintf(stderr, gettext("missing "
479             "'=' for -o option\n"));
480         return (-1);
481     }
482     *propval = '\0';
483     propval++;
484     if (nvlist_lookup_string(props, propname, &strval) == 0) {
485         (void) fprintf(stderr, gettext("property '%s' "
486             "specified multiple times\n"), propname);
487         return (-1);
488     }
489     if (nvlist_add_string(props, propname, propval) != 0)
490         nomem();
491     return (0);
492 }
494 static int
495 parse_depth(char *opt, int *flags)
496 {
497     char *tmp;
498     int depth;
500     depth = (int)strtol(opt, &tmp, 0);
501     if (*tmp) {
502         (void) fprintf(stderr,
503             gettext("%s is not an integer\n"), optarg);
504         usage(B_FALSE);
505     }
506     if (depth < 0) {
507         (void) fprintf(stderr,
508             gettext("Depth can not be negative.\n"));
509         usage(B_FALSE);
510     }
511     *flags |= (ZFS_ITER_DEPTH_LIMIT|ZFS_ITER_RECURSE);
512     return (depth);
513 }
515 #define PROGRESS_DELAY 2 /* seconds */
517 static char *pt_reverse = "\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b";
518 static time_t pt_begin;
519 static char *pt_header = NULL;
520 static boolean_t pt_shown;
522 static void

```

```

523 start_progress_timer(void)
524 {
525     pt_begin = time(NULL) + PROGRESS_DELAY;
526     pt_shown = B_FALSE;
527 }

529 static void
530 set_progress_header(char *header)
531 {
532     assert(pt_header == NULL);
533     pt_header = safe_strdup(header);
534     if (pt_shown) {
535         (void) printf("%s: ", header);
536         (void) fflush(stdout);
537     }
538 }

540 static void
541 update_progress(char *update)
542 {
543     if (!pt_shown && time(NULL) > pt_begin) {
544         int len = strlen(update);

546         (void) printf("%s: %s*.*s", pt_header, update, len, len,
547                     pt_reverse);
548         (void) fflush(stdout);
549         pt_shown = B_TRUE;
550     } else if (pt_shown) {
551         int len = strlen(update);

553         (void) printf("%s*.*s", update, len, len, pt_reverse);
554         (void) fflush(stdout);
555     }
556 }

558 static void
559 finish_progress(char *done)
560 {
561     if (pt_shown) {
562         (void) printf("%s\n", done);
563         (void) fflush(stdout);
564     }
565     free(pt_header);
566     pt_header = NULL;
567 }
568 /*
569  * zfs clone [-p] [-o prop=value] ... <snap> <fs | vol>
570  *
571  * Given an existing dataset, create a writable copy whose initial contents
572  * are the same as the source. The newly created dataset maintains a
573  * dependency on the original; the original cannot be destroyed so long as
574  * the clone exists.
575  *
576  * The '-p' flag creates all the non-existing ancestors of the target first.
577  */
578 static int
579 zfs_do_clone(int argc, char **argv)
580 {
581     zfs_handle_t *zhp = NULL;
582     boolean_t parents = B_FALSE;
583     nvlist_t *props;
584     int ret = 0;
585     int c;

587     if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0)
588         nomem();

```

```

590     /* check options */
591     while ((c = getopt(argc, argv, "o:p")) != -1) {
592         switch (c) {
593             case 'o':
594                 if (parseprop(props))
595                     return (1);
596                 break;
597             case 'p':
598                 parents = B_TRUE;
599                 break;
600             case '?':
601                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
602                               optopt);
603                 goto usage;
604         }
605     }

607     argc -= optind;
608     argv += optind;

610     /* check number of arguments */
611     if (argc < 1) {
612         (void) fprintf(stderr, gettext("missing source dataset "
613                                       "argument\n"));
614         goto usage;
615     }
616     if (argc < 2) {
617         (void) fprintf(stderr, gettext("missing target dataset "
618                                       "argument\n"));
619         goto usage;
620     }
621     if (argc > 2) {
622         (void) fprintf(stderr, gettext("too many arguments\n"));
623         goto usage;
624     }

626     /* open the source dataset */
627     if ((zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_SNAPSHOT)) == NULL)
628         return (1);

630     if (parents && zfs_name_valid(argv[1], ZFS_TYPE_FILESYSTEM |
631                                 ZFS_TYPE_VOLUME)) {
632         /*
633          * Now create the ancestors of the target dataset. If the
634          * target already exists and '-p' option was used we should not
635          * complain.
636          */
637         if (zfs_dataset_exists(g_zfs, argv[1], ZFS_TYPE_FILESYSTEM |
638                               ZFS_TYPE_VOLUME))
639             return (0);
640         if (zfs_create_ancestors(g_zfs, argv[1]) != 0)
641             return (1);
642     }

644     /* pass to libzfs */
645     ret = zfs_clone(zhp, argv[1], props);

647     /* create the mountpoint if necessary */
648     if (ret == 0) {
649         zfs_handle_t *clone;

651         clone = zfs_open(g_zfs, argv[1], ZFS_TYPE_DATASET);
652         if (clone != NULL) {
653             if (zfs_get_type(clone) != ZFS_TYPE_VOLUME)
654                 if ((ret = zfs_mount(clone, NULL, 0)) == 0)

```

```

655         ret = zfs_share(clone);
656         zfs_close(clone);
657     }
658 }

660     zfs_close(zhp);
661     nvlist_free(props);

663     return (!ret);

665 usage:
666     if (zhp)
667         zfs_close(zhp);
668     nvlist_free(props);
669     usage(B_FALSE);
670     return (-1);
671 }

673 /*
674  * zfs create [-p] [-o prop=value] ... fs
675  * zfs create [-ps] [-b blocksize] [-o prop=value] ... -V vol size
676  *
677  * Create a new dataset. This command can be used to create filesystems
678  * and volumes. Snapshot creation is handled by 'zfs snapshot'.
679  * For volumes, the user must specify a size to be used.
680  *
681  * The '-s' flag applies only to volumes, and indicates that we should not try
682  * to set the reservation for this volume. By default we set a reservation
683  * equal to the size for any volume. For pools with SPA_VERSION >=
684  * SPA_VERSION_REFRESERVATION, we set a reservation instead.
685  *
686  * The '-p' flag creates all the non-existing ancestors of the target first.
687  */
688 static int
689 zfs_do_create(int argc, char **argv)
690 {
691     zfs_type_t type = ZFS_TYPE_FILESYSTEM;
692     zfs_handle_t *zhp = NULL;
693     uint64_t volsize;
694     int c;
695     boolean_t noreserve = B_FALSE;
696     boolean_t bflag = B_FALSE;
697     boolean_t parents = B_FALSE;
698     int ret = 1;
699     nvlist_t *props;
700     uint64_t intval;
701     int canmount = ZFS_CANMOUNT_OFF;

703     if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0)
704         nomem();

706     /* check options */
707     while ((c = getopt(argc, argv, "V:b:so:p")) != -1) {
708         switch (c) {
709             case 'V':
710                 type = ZFS_TYPE_VOLUME;
711                 if (zfs_nicestrtonum(g_zfs, optarg, &intval) != 0) {
712                     (void) fprintf(stderr, gettext("bad volume "
713 "size '%s': %s\n"), optarg,
714 libzfs_error_description(g_zfs));
715                     goto error;
716                 }
717             case 's':
718                 if (nvlist_add_uint64(props,
719 zfs_prop_to_name(ZFS_PROP_VOLSIZE), intval) != 0)
720                     nomem();

```

```

721         volsize = intval;
722         break;
723     case 'p':
724         parents = B_TRUE;
725         break;
726     case 'b':
727         bflag = B_TRUE;
728         if (zfs_nicestrtonum(g_zfs, optarg, &intval) != 0) {
729             (void) fprintf(stderr, gettext("bad volume "
730 "block size '%s': %s\n"), optarg,
731 libzfs_error_description(g_zfs));
732             goto error;
733         }
734     case 'o':
735         if (nvlist_add_uint64(props,
736 zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
737 intval) != 0)
738             nomem();
739         break;
740     case 'o':
741         if (parseprop(props))
742             goto error;
743         break;
744     case 's':
745         noreserve = B_TRUE;
746         break;
747     case ':':
748         (void) fprintf(stderr, gettext("missing size "
749 "argument\n"));
750         goto badusage;
751     case '?':
752         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
753 optopt);
754         goto badusage;
755     }
756 }

758 if ((bflag || noreserve) && type != ZFS_TYPE_VOLUME) {
759     (void) fprintf(stderr, gettext("'s' and '-b' can only be "
760 "used when creating a volume\n"));
761     goto badusage;
762 }

764     argc -= optind;
765     argv += optind;

767     /* check number of arguments */
768     if (argc == 0) {
769         (void) fprintf(stderr, gettext("missing %s argument\n"),
770 zfs_type_to_name(type));
771         goto badusage;
772     }
773     if (argc > 1) {
774         (void) fprintf(stderr, gettext("too many arguments\n"));
775         goto badusage;
776     }

778     if (type == ZFS_TYPE_VOLUME && !noreserve) {
779         zpool_handle_t *zpool_handle;
780         nvlist_t *real_props;
781         uint64_t spa_version;
782         char *p;
783         zfs_prop_t resv_prop;
784         char *strval;
785         char msg[1024];

```



```

787     if (p = strchr(argv[0], '/'))
788         *p = '\0';
789     zpool_handle = zpool_open(g_zfs, argv[0]);
790     if (p != NULL)
791         *p = '/';
792     if (zpool_handle == NULL)
793         goto error;
794     spa_version = zpool_get_prop_int(zpool_handle,
795         ZPOOL_PROP_VERSION, NULL);
796     zpool_close(zpool_handle);
797     if (spa_version >= SPA_VERSION_REFRESERVATION)
798         resv_prop = ZFS_PROP_REFRESERVATION;
799     else
800         resv_prop = ZFS_PROP_RESERVATION;
801
802     (void) snprintf(msg, sizeof (msg),
803         gettext("cannot create '%s'"), argv[0]);
804     if (props && (real_props = zfs_valid_proplist(g_zfs, type,
805         props, 0, NULL, msg)) == NULL)
806         goto error;
807
808     volsize = zvol_volsize_to_reservation(volsize, real_props);
809     nvlist_free(real_props);
810
811     if (nvlist_lookup_string(props, zfs_prop_to_name(resv_prop),
812         &strval) != 0) {
813         if (nvlist_add_uint64(props,
814             zfs_prop_to_name(resv_prop), volsize) != 0) {
815             nvlist_free(props);
816             nomem();
817         }
818     }
819 }
820
821 if (parents && zfs_name_valid(argv[0], type)) {
822     /*
823      * Now create the ancestors of target dataset.  If the target
824      * already exists and '-p' option was used we should not
825      * complain.
826      */
827     if (zfs_dataset_exists(g_zfs, argv[0], type)) {
828         ret = 0;
829         goto error;
830     }
831     if (zfs_create_ancestors(g_zfs, argv[0]) != 0)
832         goto error;
833 }
834
835 /* pass to libzfs */
836 if (zfs_create(g_zfs, argv[0], type, props) != 0)
837     goto error;
838
839 if ((zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_DATASET)) == NULL)
840     goto error;
841
842 ret = 0;
843 /*
844  * if the user doesn't want the dataset automatically mounted,
845  * then skip the mount/share step
846  */
847 if (zfs_prop_valid_for_type(ZFS_PROP_CANMOUNT, type))
848     canmount = zfs_prop_get_int(zhp, ZFS_PROP_CANMOUNT);
849
850 /*
851  * Mount and/or share the new filesystem as appropriate.  We provide a
852  * verbose error message to let the user know that their filesystem was

```

```

853     * in fact created, even if we failed to mount or share it.
854     */
855     if (canmount == ZFS_CANMOUNT_ON) {
856         if (zfs_mount(zhp, NULL, 0) != 0) {
857             (void) fprintf(stderr, gettext("filesystem "
858                 "successfully created, but not mounted\n"));
859             ret = 1;
860         } else if (zfs_share(zhp) != 0) {
861             (void) fprintf(stderr, gettext("filesystem "
862                 "successfully created, but not shared\n"));
863             ret = 1;
864         }
865     }
866 error:
867     if (zhp)
868         zfs_close(zhp);
869     nvlist_free(props);
870     return (ret);
871 badusage:
872     nvlist_free(props);
873     usage(B_FALSE);
874     return (2);
875 }
876 }
877
878 /*
879  * zfs destroy [-rRf] <fs, vol>
880  * zfs destroy [-rRd] <snap>
881  *
882  * -r      Recursively destroy all children
883  * -R      Recursively destroy all dependents, including clones
884  * -f      Force unmounting of any dependents
885  * -d      If we can't destroy now, mark for deferred destruction
886  *
887  * Destroys the given dataset.  By default, it will unmount any filesystems,
888  * and refuse to destroy a dataset that has any dependents.  A dependent can
889  * either be a child, or a clone of a child.
890  */
891 typedef struct destroy_cbdata {
892     boolean_t    cb_first;
893     boolean_t    cb_force;
894     boolean_t    cb_recurse;
895     boolean_t    cb_error;
896     boolean_t    cb_dclones;
897     zfs_handle_t *cb_target;
898     boolean_t    cb_defer_destroy;
899     boolean_t    cb_verbose;
900     boolean_t    cb_parsable;
901     boolean_t    cb_dryrun;
902     nvlist_t     *cb_nvlist;
903     nvlist_t     *cb_batchedsnaps;
904
905     /* first snap in contiguous run */
906     char         *cb_firstsnap;
907     /* previous snap in contiguous run */
908     char         *cb_prevsnap;
909     int64_t      cb_snapused;
910     char         *cb_snapspec;
911 } destroy_cbdata_t;
912
913 /*
914  * Check for any dependents based on the '-r' or '-R' flags.
915  */
916 static int
917 destroy_check_dependent(zfs_handle_t *zhp, void *data)
918 {

```

```

919  destroy_cbdata_t *cbp = data;
920  const char *tname = zfs_get_name(cbp->cb_target);
921  const char *name = zfs_get_name(zhp);

923  if (strncmp(tname, name, strlen(tname)) == 0 &&
924      (name[strlen(tname)] == '/' || name[strlen(tname)] == '@')) {
925      /*
926       * This is a direct descendant, not a clone somewhere else in
927       * the hierarchy.
928       */
929      if (cbp->cb_recurse)
930          goto out;

932      if (cbp->cb_first) {
933          (void) fprintf(stderr, gettext("cannot destroy '%s': "
934              "%s has children\n"),
935              zfs_get_name(cbp->cb_target),
936              zfs_type_to_name(zfs_get_type(cbp->cb_target)));
937          (void) fprintf(stderr, gettext("use '-r' to destroy "
938              "the following datasets:\n"));
939          cbp->cb_first = B_FALSE;
940          cbp->cb_error = B_TRUE;
941      }

943      (void) fprintf(stderr, "%s\n", zfs_get_name(zhp));
944  } else {
945      /*
946       * This is a clone. We only want to report this if the '-r'
947       * wasn't specified, or the target is a snapshot.
948       */
949      if (!cbp->cb_recurse &&
950          zfs_get_type(cbp->cb_target) != ZFS_TYPE_SNAPSHOT)
951          goto out;

953      if (cbp->cb_first) {
954          (void) fprintf(stderr, gettext("cannot destroy '%s': "
955              "%s has dependent clones\n"),
956              zfs_get_name(cbp->cb_target),
957              zfs_type_to_name(zfs_get_type(cbp->cb_target)));
958          (void) fprintf(stderr, gettext("use '-R' to destroy "
959              "the following datasets:\n"));
960          cbp->cb_first = B_FALSE;
961          cbp->cb_error = B_TRUE;
962          cbp->cb_dryrun = B_TRUE;
963      }

965      (void) fprintf(stderr, "%s\n", zfs_get_name(zhp));
966  }

968  out:
969      zfs_close(zhp);
970      return (0);
971  }

973  static int
974  destroy_callback(zfs_handle_t *zhp, void *data)
975  {
976      destroy_cbdata_t *cb = data;
977      const char *name = zfs_get_name(zhp);

979      if (cb->cb_verbose) {
980          if (cb->cb_parsable) {
981              (void) printf("destroy\t%s\n", name);
982          } else if (cb->cb_dryrun) {
983              (void) printf(gettext("would destroy %s\n"),
984                  name);

```

```

985          } else {
986              (void) printf(gettext("will destroy %s\n"),
987                  name);
988          }
989      }

991      /*
992       * Ignore pools (which we've already flagged as an error before getting
993       * here).
994       */
995      if (strchr(zfs_get_name(zhp), '/') == NULL &&
996          zfs_get_type(zhp) == ZFS_TYPE_FILESYSTEM) {
997          zfs_close(zhp);
998          return (0);
999      }
1000      if (cb->cb_dryrun) {
1001          zfs_close(zhp);
1002          return (0);
1003      }

1005      /*
1006       * We batch up all contiguous snapshots (even of different
1007       * filesystems) and destroy them with one ioctl. We can't
1008       * simply do all snap deletions and then all fs deletions,
1009       * because we must delete a clone before its origin.
1010       */
1011      if (zfs_get_type(zhp) == ZFS_TYPE_SNAPSHOT) {
1012          fnvlist_add_boolean(cb->cb_batchedsnaps, name);
1013      } else {
1014          int error = zfs_destroy_snaps_nvlist(g_zfs,
1015              cb->cb_batchedsnaps, B_FALSE);
1016          fnvlist_free(cb->cb_batchedsnaps);
1017          cb->cb_batchedsnaps = fnvlist_alloc();

1019          if (error != 0 ||
1020              zfs_unmount(zhp, NULL, cb->cb_force ? MS_FORCE : 0) != 0 ||
1021              zfs_destroy(zhp, cb->cb_defer_destroy) != 0) {
1022              zfs_close(zhp);
1023              return (-1);
1024          }
1025      }

1027      zfs_close(zhp);
1028      return (0);
1029  }

1031  static int
1032  destroy_print_cb(zfs_handle_t *zhp, void *arg)
1033  {
1034      destroy_cbdata_t *cb = arg;
1035      const char *name = zfs_get_name(zhp);
1036      int err = 0;

1038      if (nvlist_exists(cb->cb_nvlist, name)) {
1039          if (cb->cb_firstsnap == NULL)
1040              cb->cb_firstsnap = strdup(name);
1041          if (cb->cb_prevsnap != NULL)
1042              free(cb->cb_prevsnap);
1043          /* this snap continues the current range */
1044          cb->cb_prevsnap = strdup(name);
1045          if (cb->cb_firstsnap == NULL || cb->cb_prevsnap == NULL)
1046              nomem();
1047          if (cb->cb_verbose) {
1048              if (cb->cb_parsable) {
1049                  (void) printf("destroy\t%s\n", name);
1050              } else if (cb->cb_dryrun) {

```

```

1051         (void) printf(gettext("would destroy %s\n"),
1052             name);
1053     } else {
1054         (void) printf(gettext("will destroy %s\n"),
1055             name);
1056     }
1057 }
1058 } else if (cb->cb_firstsnap != NULL) {
1059     /* end of this range */
1060     uint64_t used = 0;
1061     err = lzcnaprangespace(cb->cb_firstsnap,
1062         cb->cb_prevsnap, &used);
1063     cb->cb_snapused += used;
1064     free(cb->cb_firstsnap);
1065     cb->cb_firstsnap = NULL;
1066     free(cb->cb_prevsnap);
1067     cb->cb_prevsnap = NULL;
1068 }
1069 zfs_close(zhp);
1070 return (err);
1071 }

1073 static int
1074 destroy_print_snapshots(zfs_handle_t *zhp, destroy_cbdata_t *cb)
1075 {
1076     int err = 0;
1077     assert(cb->cb_firstsnap == NULL);
1078     assert(cb->cb_prevsnap == NULL);
1079     err = zfs_iter_snapshots_sorted(zhp, destroy_print_cb, cb);
1080     if (cb->cb_firstsnap != NULL) {
1081         uint64_t used = 0;
1082         if (err == 0) {
1083             err = lzcnaprangespace(cb->cb_firstsnap,
1084                 cb->cb_prevsnap, &used);
1085         }
1086         cb->cb_snapused += used;
1087         free(cb->cb_firstsnap);
1088         cb->cb_firstsnap = NULL;
1089         free(cb->cb_prevsnap);
1090         cb->cb_prevsnap = NULL;
1091     }
1092     return (err);
1093 }

1095 static int
1096 snapshot_to_nvlist(zfs_handle_t *zhp, void *arg)
1097 {
1098     destroy_cbdata_t *cb = arg;
1099     int err = 0;

1101     /* Check for clones. */
1102     if (!cb->cb_dclones && !cb->cb_defer_destroy) {
1103         cb->cb_target = zhp;
1104         cb->cb_first = B_TRUE;
1105         err = zfs_iter_dependents(zhp, B_TRUE,
1106             destroy_check_dependent, cb);
1107     }

1109     if (err == 0) {
1110         if (nvlist_add_boolean(cb->cb_nvlist, zfs_get_name(zhp)))
1111             nomem();
1112     }
1113     zfs_close(zhp);
1114     return (err);
1115 }

```

```

1117 static int
1118 gather_snapshots(zfs_handle_t *zhp, void *arg)
1119 {
1120     destroy_cbdata_t *cb = arg;
1121     int err = 0;

1123     err = zfs_iter_snapspec(zhp, cb->cb_snapspec, snapshot_to_nvlist_cb, cb);
1124     if (err == ENOENT)
1125         err = 0;
1126     if (err != 0)
1127         goto out;

1129     if (cb->cb_verbose) {
1130         err = destroy_print_snapshots(zhp, cb);
1131         if (err != 0)
1132             goto out;
1133     }

1135     if (cb->cb_recurse)
1136         err = zfs_iter_filesystems(zhp, gather_snapshots, cb);

1138 out:
1139     zfs_close(zhp);
1140     return (err);
1141 }

1143 static int
1144 destroy_clones(destroy_cbdata_t *cb)
1145 {
1146     nvpair_t *pair;
1147     for (pair = nvlist_next_nvpair(cb->cb_nvlist, NULL);
1148         pair != NULL;
1149         pair = nvlist_next_nvpair(cb->cb_nvlist, pair)) {
1150         zfs_handle_t *zhp = zfs_open(g_zfs, nvpair_name(pair),
1151             ZFS_TYPE_SNAPSHOT);
1152         if (zhp != NULL) {
1153             boolean_t defer = cb->cb_defer_destroy;
1154             int err = 0;

1156             /*
1157              * We can't defer destroy non-snapshots, so set it to
1158              * false while destroying the clones.
1159              */
1160             cb->cb_defer_destroy = B_FALSE;
1161             err = zfs_iter_dependents(zhp, B_FALSE,
1162                 destroy_callback, cb);
1163             cb->cb_defer_destroy = defer;
1164             zfs_close(zhp);
1165             if (err != 0)
1166                 return (err);
1167         }
1168     }
1169     return (0);
1170 }

1172 static int
1173 zfs_do_destroy(int argc, char **argv)
1174 {
1175     destroy_cbdata_t cb = { 0 };
1176     int rv = 0;
1177     int err = 0;
1178     int c;
1179     zfs_handle_t *zhp = NULL;
1180     char *at;
1181     zfs_type_t type = ZFS_TYPE_DATASET;

```

```

1183  /* check options */
1184  while ((c = getopt(argc, argv, "vpndfrR")) != -1) {
1185      switch (c) {
1186          case 'v':
1187              cb.cb_verbose = B_TRUE;
1188              break;
1189          case 'p':
1190              cb.cb_verbose = B_TRUE;
1191              cb.cb_parsable = B_TRUE;
1192              break;
1193          case 'n':
1194              cb.cb_dryrun = B_TRUE;
1195              break;
1196          case 'd':
1197              cb.cb_defer_destroy = B_TRUE;
1198              type = ZFS_TYPE_SNAPSHOT;
1199              break;
1200          case 'f':
1201              cb.cb_force = B_TRUE;
1202              break;
1203          case 'r':
1204              cb.cb_recurse = B_TRUE;
1205              break;
1206          case 'R':
1207              cb.cb_recurse = B_TRUE;
1208              cb.cb_doclones = B_TRUE;
1209              break;
1210          case '?':
1211          default:
1212              (void) fprintf(stderr, gettext("invalid option '%c'\n"),
1213                  optopt);
1214              usage(B_FALSE);
1215          }
1216      }

1218  argc -= optind;
1219  argv += optind;

1221  /* check number of arguments */
1222  if (argc == 0) {
1223      (void) fprintf(stderr, gettext("missing dataset argument\n"));
1224      usage(B_FALSE);
1225  }
1226  if (argc > 1) {
1227      (void) fprintf(stderr, gettext("too many arguments\n"));
1228      usage(B_FALSE);
1229  }

1231  at = strchr(argv[0], '@');
1232  if (at != NULL) {

1234      /* Build the list of snaps to destroy in cb_nvl. */
1235      cb.cb_nvl = fnvlist_alloc();

1237      *at = '\0';
1238      zhp = zfs_open(g_zfs, argv[0],
1239          ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
1240      if (zhp == NULL)
1241          return (1);

1243      cb.cb_snapspec = at + 1;
1244      if (gather_snapshots(zfs_handle_dup(zhp), &cb) != 0 ||
1245          cb.cb_error) {
1246          rv = 1;
1247          goto out;
1248      }

```

```

1250      if (nvlst_empty(cb.cb_nvl)) {
1251          (void) fprintf(stderr, gettext("could not find any "
1252              "snapshots to destroy; check snapshot names.\n"));
1253          rv = 1;
1254          goto out;
1255      }

1257      if (cb.cb_verbose) {
1258          char buf[16];
1259          zfs_nicenum(cb.cb_snapused, buf, sizeof (buf));
1260          if (cb.cb_parsable) {
1261              (void) printf("reclaim\t%llu\n",
1262                  cb.cb_snapused);
1263          } else if (cb.cb_dryrun) {
1264              (void) printf(gettext("would reclaim %s\n"),
1265                  buf);
1266          } else {
1267              (void) printf(gettext("will reclaim %s\n"),
1268                  buf);
1269          }
1270      }

1272      if (!cb.cb_dryrun) {
1273          if (cb.cb_doclones) {
1274              cb.cb_batchedsnaps = fnvlist_alloc();
1275              err = destroy_clones(&cb);
1276              if (err == 0) {
1277                  err = zfs_destroy_snaps_nvl(g_zfs,
1278                      cb.cb_batchedsnaps, B_FALSE);
1279              }
1280              if (err != 0) {
1281                  rv = 1;
1282                  goto out;
1283              }
1284          }
1285          if (err == 0) {
1286              err = zfs_destroy_snaps_nvl(g_zfs, cb.cb_nvl,
1287                  cb.cb_defer_destroy);
1288          }
1289      }

1291      if (err != 0)
1292          rv = 1;
1293  } else {
1294      /* Open the given dataset */
1295      if ((zhp = zfs_open(g_zfs, argv[0], type)) == NULL)
1296          return (1);

1298      cb.cb_target = zhp;

1300      /*
1301       * Perform an explicit check for pools before going any further.
1302       */
1303      if (!cb.cb_recurse && strchr(zfs_get_name(zhp), '/') == NULL &&
1304          zfs_get_type(zhp) == ZFS_TYPE_FILESYSTEM) {
1305          (void) fprintf(stderr, gettext("cannot destroy '%s': "
1306              "operation does not apply to pools\n"),
1307              zfs_get_name(zhp));
1308          (void) fprintf(stderr, gettext("use 'zfs destroy -r "
1309              "%s' to destroy all datasets in the pool\n"),
1310              zfs_get_name(zhp));
1311          (void) fprintf(stderr, gettext("use 'zpool destroy %s' "
1312              "to destroy the pool itself\n"), zfs_get_name(zhp));
1313          rv = 1;
1314          goto out;

```

```

1315     }
1316
1317     /*
1318     * Check for any dependents and/or clones.
1319     */
1320     cb.cb_first = B_TRUE;
1321     if (!cb.cb_doclones &&
1322         zfs_iter_dependents(zhp, B_TRUE, destroy_check_dependent,
1323             &cb) != 0) {
1324         rv = 1;
1325         goto out;
1326     }
1327
1328     if (cb.cb_error) {
1329         rv = 1;
1330         goto out;
1331     }
1332
1333     cb.cb_batchedsnaps = fnvlist_alloc();
1334     if (zfs_iter_dependents(zhp, B_FALSE, destroy_callback,
1335         &cb) != 0) {
1336         rv = 1;
1337         goto out;
1338     }
1339
1340     /*
1341     * Do the real thing. The callback will close the
1342     * handle regardless of whether it succeeds or not.
1343     */
1344     err = destroy_callback(zhp, &cb);
1345     zhp = NULL;
1346     if (err == 0) {
1347         err = zfs_destroy_snaps_nvlist(g_zfs,
1348             cb.cb_batchedsnaps, cb.cb_defer_destroy);
1349     }
1350     if (err != 0)
1351         rv = 1;
1352 }
1353
1354 out:
1355     fnvlist_free(cb.cb_batchedsnaps);
1356     fnvlist_free(cb.cb_nvlist);
1357     if (zhp != NULL)
1358         zfs_close(zhp);
1359     return (rv);
1360 }
1361
1362 static boolean_t
1363 is_recvd_column(zprop_get_cbdata_t *cbp)
1364 {
1365     int i;
1366     zfs_get_column_t col;
1367
1368     for (i = 0; i < ZFS_GET_NCOLS &&
1369         (col = cbp->cb_columns[i]) != GET_COL_NONE; i++)
1370         if (col == GET_COL_RECVD)
1371             return (B_TRUE);
1372     return (B_FALSE);
1373 }
1374
1375 /*
1376 * zfs get [-rHp] [-o all | field[,field]...] [-s source[,source]...]
1377 * < all | property[,property]... > < fs | snap | vol > ...
1378 *
1379 * -r    recurse over any child datasets
1380 * -H    scripted mode. Headers are stripped, and fields are separated

```

```

1381 *      by tabs instead of spaces.
1382 *      -o      Set of fields to display. One of "name,property,value,
1383 *             received,source". Default is "name,property,value,source".
1384 *             "all" is an alias for all five.
1385 *      -s      Set of sources to allow. One of
1386 *             "local,default,inherited,received,temporary,none". Default is
1387 *             all six.
1388 *      -p      Display values in parsable (literal) format.
1389 *
1390 * Prints properties for the given datasets. The user can control which
1391 * columns to display as well as which property types to allow.
1392 */
1393
1394 /*
1395 * Invoked to display the properties for a single dataset.
1396 */
1397 static int
1398 get_callback(zfs_handle_t *zhp, void *data)
1399 {
1400     char buf[ZFS_MAXPROPLEN];
1401     char rbuf[ZFS_MAXPROPLEN];
1402     zprop_source_t sourcetype;
1403     char source[ZFS_MAXNAMELEN];
1404     zprop_get_cbdata_t *cbp = data;
1405     nvlist_t *user_props = zfs_get_user_props(zhp);
1406     zprop_list_t *pl = cbp->cb_proplist;
1407     nvlist_t *propval;
1408     char *strval;
1409     char *sourceval;
1410     boolean_t received = is_recvd_column(cbp);
1411
1412     for (; pl != NULL; pl = pl->pl_next) {
1413         char *recvdval = NULL;
1414         /*
1415          * Skip the special fake placeholder. This will also skip over
1416          * the name property when 'all' is specified.
1417          */
1418         if (pl->pl_prop == ZFS_PROP_NAME &&
1419             pl == cbp->cb_proplist)
1420             continue;
1421
1422         if (pl->pl_prop != ZPROP_INVAL) {
1423             if (zfs_prop_get(zhp, pl->pl_prop, buf,
1424                 sizeof (buf), &sourcetype, source,
1425                 sizeof (source),
1426                 cbp->cb_literal) != 0) {
1427                 if (pl->pl_all)
1428                     continue;
1429                 if (!zfs_prop_valid_for_type(pl->pl_prop,
1430                     ZFS_TYPE_DATASET)) {
1431                     (void) fprintf(stderr,
1432                         gettext("No such property '%s'\n"),
1433                         zfs_prop_to_name(pl->pl_prop));
1434                     continue;
1435                 }
1436                 sourcetype = ZPROP_SRC_NONE;
1437                 (void) strlcpy(buf, "-", sizeof (buf));
1438             }
1439
1440             if (received && (zfs_prop_get_recvd(zhp,
1441                 zfs_prop_to_name(pl->pl_prop), rbuf, sizeof (rbuf),
1442                 cbp->cb_literal) == 0))
1443                 recvdval = rbuf;
1444
1445             zprop_print_one_property(zfs_get_name(zhp), cbp,
1446                 zfs_prop_to_name(pl->pl_prop),

```

```

1447     buf, sourcetype, source, recvdval);
1448 } else if (zfs_prop_userquota(pl->pl_user_prop)) {
1449     sourcetype = ZPROP_SRC_LOCAL;
1450
1451     if (zfs_prop_get_userquota(zhp, pl->pl_user_prop,
1452         buf, sizeof (buf), cbp->cb_literal) != 0) {
1453         sourcetype = ZPROP_SRC_NONE;
1454         (void) strlcpy(buf, "-", sizeof (buf));
1455     }
1456
1457     zprop_print_one_property(zfs_get_name(zhp), cbp,
1458         pl->pl_user_prop, buf, sourcetype, source, NULL);
1459 } else if (zfs_prop_written(pl->pl_user_prop)) {
1460     sourcetype = ZPROP_SRC_LOCAL;
1461
1462     if (zfs_prop_get_written(zhp, pl->pl_user_prop,
1463         buf, sizeof (buf), cbp->cb_literal) != 0) {
1464         sourcetype = ZPROP_SRC_NONE;
1465         (void) strlcpy(buf, "-", sizeof (buf));
1466     }
1467
1468     zprop_print_one_property(zfs_get_name(zhp), cbp,
1469         pl->pl_user_prop, buf, sourcetype, source, NULL);
1470 } else {
1471     if (nvlist_lookup_nvlist(user_props,
1472         pl->pl_user_prop, &propval) != 0) {
1473         if (pl->pl_all)
1474             continue;
1475         sourcetype = ZPROP_SRC_NONE;
1476         strval = "-";
1477     } else {
1478         verify(nvlist_lookup_string(propval,
1479             ZPROP_VALUE, &strval) == 0);
1480         verify(nvlist_lookup_string(propval,
1481             ZPROP_SOURCE, &sourceval) == 0);
1482
1483         if (strcmp(sourceval,
1484             zfs_get_name(zhp)) == 0) {
1485             sourcetype = ZPROP_SRC_LOCAL;
1486         } else if (strcmp(sourceval,
1487             ZPROP_SOURCE_VAL_RECVD) == 0) {
1488             sourcetype = ZPROP_SRC_RECEIVED;
1489         } else {
1490             sourcetype = ZPROP_SRC_INHERITED;
1491             (void) strlcpy(source,
1492                 sourceval, sizeof (source));
1493         }
1494     }
1495
1496     if (received && (zfs_prop_get_recvd(zhp,
1497         pl->pl_user_prop, rbuf, sizeof (rbuf),
1498         cbp->cb_literal) == 0))
1499         recvdval = rbuf;
1500
1501     zprop_print_one_property(zfs_get_name(zhp), cbp,
1502         pl->pl_user_prop, strval, sourcetype,
1503         source, recvdval);
1504 }
1505 }
1506
1507     return (0);
1508 }
1509
1510 static int
1511 zfs_do_get(int argc, char **argv)
1512 {

```

```

1513     zprop_get_cbdata_t cb = { 0 };
1514     int i, c, flags = ZFS_ITER_ARGS_CAN_BE_PATHS;
1515     int types = ZFS_TYPE_DATASET;
1516     char *value, *fields;
1517     int ret = 0;
1518     int limit = 0;
1519     zprop_list_t fake_name = { 0 };
1520
1521     /*
1522      * Set up default columns and sources.
1523      */
1524     cb.cb_sources = ZPROP_SRC_ALL;
1525     cb.cb_columns[0] = GET_COL_NAME;
1526     cb.cb_columns[1] = GET_COL_PROPERTY;
1527     cb.cb_columns[2] = GET_COL_VALUE;
1528     cb.cb_columns[3] = GET_COL_SOURCE;
1529     cb.cb_type = ZFS_TYPE_DATASET;
1530
1531     /* check options */
1532     while ((c = getopt(argc, argv, "d:o:s:rt:Hp")) != -1) {
1533         switch (c) {
1534             case 'p':
1535                 cb.cb_literal = B_TRUE;
1536                 break;
1537             case 'd':
1538                 limit = parse_depth(optarg, &flags);
1539                 break;
1540             case 'r':
1541                 flags |= ZFS_ITER_RECURSE;
1542                 break;
1543             case 'H':
1544                 cb.cb_scripted = B_TRUE;
1545                 break;
1546             case ':':
1547                 (void) fprintf(stderr, gettext("missing argument for "
1548                     "%c" option\n"), optopt);
1549                 usage(B_FALSE);
1550                 break;
1551             case 'o':
1552                 /*
1553                  * Process the set of columns to display. We zero out
1554                  * the structure to give us a blank slate.
1555                  */
1556                 bzero(&cb.cb_columns, sizeof (cb.cb_columns));
1557                 i = 0;
1558                 while (*optarg != '\0') {
1559                     static char *col_subopts[] =
1560                         { "name", "property", "value", "received",
1561                             "source", "all", NULL };
1562
1563                     if (i == ZFS_GET_NCOLS) {
1564                         (void) fprintf(stderr, gettext("too "
1565                             "many fields given to -o "
1566                             "option\n"));
1567                         usage(B_FALSE);
1568                     }
1569
1570                     switch (getsubopt(&optarg, col_subopts,
1571                         &value)) {
1572                         case 0:
1573                             cb.cb_columns[i++] = GET_COL_NAME;
1574                             break;
1575                         case 1:
1576                             cb.cb_columns[i++] = GET_COL_PROPERTY;
1577                             break;
1578                         case 2:

```

```

1579         cb.cb_columns[i++] = GET_COL_VALUE;
1580         break;
1581     case 3:
1582         cb.cb_columns[i++] = GET_COL_RECVD;
1583         flags |= ZFS_ITER_RECVD_PROPS;
1584         break;
1585     case 4:
1586         cb.cb_columns[i++] = GET_COL_SOURCE;
1587         break;
1588     case 5:
1589         if (i > 0) {
1590             (void) fprintf(stderr,
1591                 gettext("\nall\ conflicts "
1592                     "with specific fields "
1593                     "given to -o option\n"));
1594             usage(B_FALSE);
1595         }
1596         cb.cb_columns[0] = GET_COL_NAME;
1597         cb.cb_columns[1] = GET_COL_PROPERTY;
1598         cb.cb_columns[2] = GET_COL_VALUE;
1599         cb.cb_columns[3] = GET_COL_RECVD;
1600         cb.cb_columns[4] = GET_COL_SOURCE;
1601         flags |= ZFS_ITER_RECVD_PROPS;
1602         i = ZFS_GET_NCOLS;
1603         break;
1604     default:
1605         (void) fprintf(stderr,
1606             gettext("invalid column name "
1607                 "%s\n"), value);
1608         usage(B_FALSE);
1609     }
1610 }
1611 break;
1612
1613 case 's':
1614     cb.cb_sources = 0;
1615     while (*optarg != '\0') {
1616         static char *source_subopts[] = {
1617             "local", "default", "inherited",
1618             "received", "temporary", "none",
1619             NULL };
1620
1621         switch (getsubopt(&optarg, source_subopts,
1622             &value)) {
1623         case 0:
1624             cb.cb_sources |= ZPROP_SRC_LOCAL;
1625             break;
1626         case 1:
1627             cb.cb_sources |= ZPROP_SRC_DEFAULT;
1628             break;
1629         case 2:
1630             cb.cb_sources |= ZPROP_SRC_INHERITED;
1631             break;
1632         case 3:
1633             cb.cb_sources |= ZPROP_SRC_RECEIVED;
1634             break;
1635         case 4:
1636             cb.cb_sources |= ZPROP_SRC_TEMPORARY;
1637             break;
1638         case 5:
1639             cb.cb_sources |= ZPROP_SRC_NONE;
1640             break;
1641         default:
1642             (void) fprintf(stderr,
1643                 gettext("invalid source "
1644                     "%s\n"), value);

```

```

1645         usage(B_FALSE);
1646     }
1647     }
1648     break;
1649
1650     case 't':
1651         types = 0;
1652         flags &= ~ZFS_ITER_PROP_LISTSNAPS;
1653         while (*optarg != '\0') {
1654             static char *type_subopts[] = { "filesystem",
1655                 "volume", "snapshot", "all", NULL };
1656
1657             switch (getsubopt(&optarg, type_subopts,
1658                 &value)) {
1659             case 0:
1660                 types |= ZFS_TYPE_FILESYSTEM;
1661                 break;
1662             case 1:
1663                 types |= ZFS_TYPE_VOLUME;
1664                 break;
1665             case 2:
1666                 types |= ZFS_TYPE_SNAPSHOT;
1667                 break;
1668             case 3:
1669                 types = ZFS_TYPE_DATASET;
1670                 break;
1671             default:
1672                 (void) fprintf(stderr,
1673                     gettext("invalid type '%s'\n"),
1674                     value);
1675                 usage(B_FALSE);
1676             }
1677         }
1678     }
1679     break;
1680
1681     case '?':
1682         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
1683             optopt);
1684         usage(B_FALSE);
1685     }
1686 }
1687
1688     argc -= optind;
1689     argv += optind;
1690
1691     if (argc < 1) {
1692         (void) fprintf(stderr, gettext("missing property "
1693             "argument\n"));
1694         usage(B_FALSE);
1695     }
1696
1697     fields = argv[0];
1698
1699     if (zprop_get_list(g_zfs, fields, &cb.cb_proplist, ZFS_TYPE_DATASET)
1700         != 0)
1701         usage(B_FALSE);
1702
1703     argc--;
1704     argv++;
1705
1706     /*
1707     * As part of zfs_expand_proplist(), we keep track of the maximum column
1708     * width for each property. For the 'NAME' (and 'SOURCE') columns, we
1709     * need to know the maximum name length. However, the user likely did
1710     * not specify 'name' as one of the properties to fetch, so we need to

```

```

1711     * make sure we always include at least this property for
1712     * print_get_headers() to work properly.
1713     */
1714     if (cb.cb_proplist != NULL) {
1715         fake_name.pl_prop = ZFS_PROP_NAME;
1716         fake_name.pl_width = strlen(gettext("NAME"));
1717         fake_name.pl_next = cb.cb_proplist;
1718         cb.cb_proplist = &fake_name;
1719     }
1721     cb.cb_first = B_TRUE;
1723     /* run for each object */
1724     ret = zfs_for_each(argc, argv, flags, types, NULL,
1725         &cb.cb_proplist, limit, get_callback, &cb);
1727     if (cb.cb_proplist == &fake_name)
1728         zprop_free_list(fake_name.pl_next);
1729     else
1730         zprop_free_list(cb.cb_proplist);
1732     return (ret);
1733 }
1735 /*
1736  * inherit [-rS] <property> <fs|vol> ...
1737  *
1738  *   -r   Recurse over all children
1739  *   -S   Revert to received value, if any
1740  *
1741  * For each dataset specified on the command line, inherit the given property
1742  * from its parent. Inheriting a property at the pool level will cause it to
1743  * use the default value. The '-r' flag will recurse over all children, and is
1744  * useful for setting a property on a hierarchy-wide basis, regardless of any
1745  * local modifications for each dataset.
1746  */
1748 typedef struct inherit_cbdata {
1749     const char *cb_propname;
1750     boolean_t cb_received;
1751 } inherit_cbdata_t;
1753 static int
1754 inherit_recurse_cb(zfs_handle_t *zhp, void *data)
1755 {
1756     inherit_cbdata_t *cb = data;
1757     zfs_prop_t prop = zfs_name_to_prop(cb->cb_propname);
1759     /*
1760      * If we're doing it recursively, then ignore properties that
1761      * are not valid for this type of dataset.
1762      */
1763     if (prop != ZPROP_INVALID &&
1764         !zfs_prop_valid_for_type(prop, zfs_get_type(zhp)))
1765         return (0);
1767     return (zfs_prop_inherit(zhp, cb->cb_propname, cb->cb_received) != 0);
1768 }
1770 static int
1771 inherit_cb(zfs_handle_t *zhp, void *data)
1772 {
1773     inherit_cbdata_t *cb = data;
1775     return (zfs_prop_inherit(zhp, cb->cb_propname, cb->cb_received) != 0);
1776 }

```

```

1778 static int
1779 zfs_do_inherit(int argc, char **argv)
1780 {
1781     int c;
1782     zfs_prop_t prop;
1783     inherit_cbdata_t cb = { 0 };
1784     char *propname;
1785     int ret = 0;
1786     int flags = 0;
1787     boolean_t received = B_FALSE;
1789     /* check options */
1790     while ((c = getopt(argc, argv, "rS")) != -1) {
1791         switch (c) {
1792             case 'r':
1793                 flags |= ZFS_ITER_RECURSE;
1794                 break;
1795             case 'S':
1796                 received = B_TRUE;
1797                 break;
1798             case '?':
1799             default:
1800                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
1801                     optopt);
1802                 usage(B_FALSE);
1803         }
1804     }
1806     argc -= optind;
1807     argv += optind;
1809     /* check number of arguments */
1810     if (argc < 1) {
1811         (void) fprintf(stderr, gettext("missing property argument\n"));
1812         usage(B_FALSE);
1813     }
1814     if (argc < 2) {
1815         (void) fprintf(stderr, gettext("missing dataset argument\n"));
1816         usage(B_FALSE);
1817     }
1819     propname = argv[0];
1820     argc--;
1821     argv++;
1823     if ((prop = zfs_name_to_prop(propname)) != ZPROP_INVALID) {
1824         if (zfs_prop_readonly(prop)) {
1825             (void) fprintf(stderr, gettext(
1826                 "%s property is read-only\n",
1827                 propname));
1828             return (1);
1829         }
1830         if (!zfs_prop_inheritable(prop) && !received) {
1831             (void) fprintf(stderr, gettext("'%s' property cannot
1832                 "be inherited\n"), propname);
1833             if (prop == ZFS_PROP_QUOTA ||
1834                 prop == ZFS_PROP_RESERVATION ||
1835                 prop == ZFS_PROP_REFQUOTA ||
1836                 prop == ZFS_PROP_REFRESERVATION)
1837                 (void) fprintf(stderr, gettext("use 'zfs set "
1838                     "%s=none' to clear\n"), propname);
1839             return (1);
1840         }
1841         if (received && (prop == ZFS_PROP_VOLSIZE ||
1842             prop == ZFS_PROP_VERSION)) {

```



```

1843         (void) fprintf(stderr, gettext("%s' property cannot "
1844         "be reverted to a received value\n"), propname);
1845         return (1);
1846     }
1847 } else if (!zfs_prop_user(propname)) {
1848     (void) fprintf(stderr, gettext("invalid property '%s'\n"),
1849     propname);
1850     usage(B_FALSE);
1851 }
1852
1853 cb.cb_propname = propname;
1854 cb.cb_received = received;
1855
1856 if (flags & ZFS_ITER_RECURSE) {
1857     ret = zfs_for_each(argc, argv, flags, ZFS_TYPE_DATASET,
1858     NULL, NULL, 0, inherit_recurse_cb, &cb);
1859 } else {
1860     ret = zfs_for_each(argc, argv, flags, ZFS_TYPE_DATASET,
1861     NULL, NULL, 0, inherit_cb, &cb);
1862 }
1863
1864 return (ret);
1865 }
1866
1867 typedef struct upgrade_cbdata {
1868     uint64_t cb_numupgraded;
1869     uint64_t cb_numsamegraded;
1870     uint64_t cb_numfailed;
1871     uint64_t cb_version;
1872     boolean_t cb_newer;
1873     boolean_t cb_foundedone;
1874     char cb_lastfs[ZFS_MAXNAMELEN];
1875 } upgrade_cbdata_t;
1876
1877 static int
1878 same_pool(zfs_handle_t *zhp, const char *name)
1879 {
1880     int len1 = strcspn(name, "@/");
1881     const char *zhname = zfs_get_name(zhp);
1882     int len2 = strcspn(zhname, "@/");
1883
1884     if (len1 != len2)
1885         return (B_FALSE);
1886     return (strcmp(name, zhname, len1) == 0);
1887 }
1888
1889 static int
1890 upgrade_list_callback(zfs_handle_t *zhp, void *data)
1891 {
1892     upgrade_cbdata_t *cb = data;
1893     int version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1894
1895     /* list if it's old/new */
1896     if ((!cb->cb_newer && version < ZPL_VERSION) ||
1897         (cb->cb_newer && version > ZPL_VERSION)) {
1898         char *str;
1899         if (cb->cb_newer) {
1900             str = gettext("The following filesystems are "
1901             "formatted using a newer software version and\n"
1902             "cannot be accessed on the current system.\n\n");
1903         } else {
1904             str = gettext("The following filesystems are "
1905             "out of date, and can be upgraded. After being\n"
1906             "upgraded, these filesystems (and any 'zfs send' "
1907             "streams generated from\n"
1908             "subsequent snapshots) will no longer be "

```

```

1909         "accessible by older software versions.\n\n");
1910     }
1911
1912     if (!cb->cb_foundedone) {
1913         (void) puts(str);
1914         (void) printf(gettext("VER  FILESYSTEM\n"));
1915         (void) printf(gettext("---  -----\n"));
1916         cb->cb_foundedone = B_TRUE;
1917     }
1918
1919     (void) printf("%2u  %s\n", version, zfs_get_name(zhp));
1920 }
1921
1922 return (0);
1923 }
1924
1925 static int
1926 upgrade_set_callback(zfs_handle_t *zhp, void *data)
1927 {
1928     upgrade_cbdata_t *cb = data;
1929     int version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1930     int needed_spa_version;
1931     int spa_version;
1932
1933     if (zfs_spa_version(zhp, &spa_version) < 0)
1934         return (-1);
1935
1936     needed_spa_version = zfs_spa_version_map(cb->cb_version);
1937
1938     if (needed_spa_version < 0)
1939         return (-1);
1940
1941     if (spa_version < needed_spa_version) {
1942         /* can't upgrade */
1943         (void) printf(gettext("%s: can not be "
1944         "upgraded; the pool version needs to first "
1945         "be upgraded\nto version %d\n\n"),
1946         zfs_get_name(zhp), needed_spa_version);
1947         cb->cb_numfailed++;
1948         return (0);
1949     }
1950
1951     /* upgrade */
1952     if (version < cb->cb_version) {
1953         char verstr[16];
1954         (void) snprintf(verstr, sizeof(verstr),
1955         "%llu", cb->cb_version);
1956         if (cb->cb_lastfs[0] && !same_pool(zhp, cb->cb_lastfs)) {
1957             /*
1958              * If they did "zfs upgrade -a", then we could
1959              * be doing ioctl's to different pools. We need
1960              * to log this history once to each pool, and bypass
1961              * the normal history logging that happens in main().
1962              */
1963             (void) zpool_log_history(g_zfs, history_str);
1964             log_history = B_FALSE;
1965         }
1966         if (zfs_prop_set(zhp, "version", verstr) == 0)
1967             cb->cb_numupgraded++;
1968     } else
1969         cb->cb_numfailed++;
1970     (void) strcpy(cb->cb_lastfs, zfs_get_name(zhp));
1971 } else if (version > cb->cb_version) {
1972     /* can't downgrade */
1973     (void) printf(gettext("%s: can not be downgraded; "
1974     "it is already at version %u\n"),

```

```

1975     zfs_get_name(zhp, version);
1976     cb->cb_numfailed++;
1977 } else {
1978     cb->cb_numsamegraded++;
1979 }
1980     return (0);
1981 }

1983 /*
1984 * zfs upgrade
1985 * zfs upgrade -v
1986 * zfs upgrade [-r] [-V <version>] <-a | filesystem>
1987 */
1988 static int
1989 zfs_do_upgrade(int argc, char **argv)
1990 {
1991     boolean_t all = B_FALSE;
1992     boolean_t showversions = B_FALSE;
1993     int ret = 0;
1994     upgrade_cbdata_t cb = { 0 };
1995     char c;
1996     int flags = ZFS_ITER_ARGS_CAN_BE_PATHS;

1998     /* check options */
1999     while ((c = getopt(argc, argv, "rvV:a")) != -1) {
2000         switch (c) {
2001             case 'r':
2002                 flags |= ZFS_ITER_RECURSE;
2003                 break;
2004             case 'v':
2005                 showversions = B_TRUE;
2006                 break;
2007             case 'V':
2008                 if (zfs_prop_string_to_index(ZFS_PROP_VERSION,
2009                     optarg, &cb.cb_version) != 0) {
2010                     (void) fprintf(stderr,
2011                         gettext("invalid version %s\n"), optarg);
2012                     usage(B_FALSE);
2013                 }
2014                 break;
2015             case 'a':
2016                 all = B_TRUE;
2017                 break;
2018             case '?':
2019             default:
2020                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
2021                     optarg);
2022                 usage(B_FALSE);
2023         }
2024     }

2026     argc -= optind;
2027     argv += optind;

2029     if ((!all && !argc) && ((flags & ZFS_ITER_RECURSE) | cb.cb_version))
2030         usage(B_FALSE);
2031     if (showversions && (flags & ZFS_ITER_RECURSE || all ||
2032         cb.cb_version || argc))
2033         usage(B_FALSE);
2034     if ((all || !argc) && (showversions))
2035         usage(B_FALSE);
2036     if (all && argc)
2037         usage(B_FALSE);

2039     if (showversions) {
2040         /* Show info on available versions. */

```

```

2041         (void) printf(gettext("The following filesystem versions are "
2042             "supported:\n\n"));
2043         (void) printf(gettext("VER  DESCRIPTION\n"));
2044         (void) printf("-----\n");
2045         "-----\n");
2046         (void) printf(gettext(" 1  Initial ZFS filesystem version\n"));
2047         (void) printf(gettext(" 2  Enhanced directory entries\n"));
2048         (void) printf(gettext(" 3  Case insensitive and filesystem "
2049             "user identifier (FUID)\n"));
2050         (void) printf(gettext(" 4  userquota, groupquota "
2051             "properties\n"));
2052         (void) printf(gettext(" 5  System attributes\n"));
2053         (void) printf(gettext("\nFor more information on a particular "
2054             "version, including supported releases,\n"));
2055         (void) printf("see the ZFS Administration Guide.\n\n");
2056         ret = 0;
2057     } else if (argc || all) {
2058         /* Upgrade filesystems */
2059         if (cb.cb_version == 0)
2060             cb.cb_version = ZPL_VERSION;
2061         ret = zfs_for_each(argc, argv, flags, ZFS_TYPE_FILESYSTEM,
2062             NULL, NULL, 0, upgrade_set_callback, &cb);
2063         (void) printf(gettext("%llu filesystems upgraded\n"),
2064             cb.cb_numupgraded);
2065         if (cb.cb_numsamegraded) {
2066             (void) printf(gettext("%llu filesystems already at "
2067                 "this version\n"),
2068                 cb.cb_numsamegraded);
2069         }
2070         if (cb.cb_numfailed != 0)
2071             ret = 1;
2072     } else {
2073         /* List old-version filesystems */
2074         boolean_t found;
2075         (void) printf(gettext("This system is currently running "
2076             "ZFS filesystem version %llu.\n\n"), ZPL_VERSION);

2078         flags |= ZFS_ITER_RECURSE;
2079         ret = zfs_for_each(0, NULL, flags, ZFS_TYPE_FILESYSTEM,
2080             NULL, NULL, 0, upgrade_list_callback, &cb);

2082         found = cb.cb_foundone;
2083         cb.cb_foundone = B_FALSE;
2084         cb.cb_newer = B_TRUE;

2086         ret = zfs_for_each(0, NULL, flags, ZFS_TYPE_FILESYSTEM,
2087             NULL, NULL, 0, upgrade_list_callback, &cb);

2089         if (!cb.cb_foundone && !found) {
2090             (void) printf(gettext("All filesystems are "
2091                 "formatted with the current version.\n"));
2092         }
2093     }

2095     return (ret);
2096 }

2098 /*
2099 * zfs userspace [-Hinp] [-o field[,...]] [-s field [-s field]...]
2100 * [-S field [-S field]...] [-t type[,...]] filesystem | snapshot
2101 * zfs groupspace [-Hinp] [-o field[,...]] [-s field [-s field]...]
2102 * [-S field [-S field]...] [-t type[,...]] filesystem | snapshot
2103 *
2104 * -H Scripted mode; elide headers and separate columns by tabs.
2105 * -i Translate SID to POSIX ID.
2106 * -n Print numeric ID instead of user/group name.

```

```

2107 *      -o      Control which fields to display.
2108 *      -p      Use exact (parseable) numeric output.
2109 *      -s      Specify sort columns, descending order.
2110 *      -S      Specify sort columns, ascending order.
2111 *      -t      Control which object types to display.
2112 *
2113 *      Displays space consumed by, and quotas on, each user in the specified
2114 *      filesystem or snapshot.
2115 */

2117 /* us_field_types, us_field_hdr and us_field_names should be kept in sync */
2118 enum us_field_types {
2119     USFIELD_TYPE,
2120     USFIELD_NAME,
2121     USFIELD_USED,
2122     USFIELD_QUOTA
2123 };
2124 static char *us_field_hdr[] = { "TYPE", "NAME", "USED", "QUOTA" };
2125 static char *us_field_names[] = { "type", "name", "used", "quota" };
2126 #define USFIELD_LAST      (sizeof (us_field_names) / sizeof (char *))

2128 #define USTYPE_PSX_GRP    (1 << 0)
2129 #define USTYPE_PSX_USR    (1 << 1)
2130 #define USTYPE_SMB_GRP    (1 << 2)
2131 #define USTYPE_SMB_USR    (1 << 3)
2132 #define USTYPE_ALL        \
2133     (USTYPE_PSX_GRP | USTYPE_PSX_USR | USTYPE_SMB_GRP | USTYPE_SMB_USR)

2135 static int us_type_bits[] = {
2136     USTYPE_PSX_GRP,
2137     USTYPE_PSX_USR,
2138     USTYPE_SMB_GRP,
2139     USTYPE_SMB_USR,
2140     USTYPE_ALL
2141 };
2142 static char *us_type_names[] = { "posixgroup", "posxiuser", "smbgroup",
2143     "smbuser", "all" };

2145 typedef struct us_node {
2146     nvlist_t      *usn_nvl;
2147     uu_avl_node_t  usn_avlnode;
2148     uu_list_node_t usn_listnode;
2149 } us_node_t;

2151 typedef struct us_cbdata {
2152     nvlist_t      **cb_nvlp;
2153     uu_avl_pool_t *cb_avl_pool;
2154     uu_avl_t      *cb_avl;
2155     boolean_t     cb_numname;
2156     boolean_t     cb_nicenum;
2157     boolean_t     cb_sid2posix;
2158     zfs_userquota_prop_t cb_prop;
2159     zfs_sort_column_t *cb_sortcol;
2160     size_t        cb_width[USFIELD_LAST];
2161 } us_cbdata_t;

2163 static boolean_t us_populated = B_FALSE;

2165 typedef struct {
2166     zfs_sort_column_t *si_sortcol;
2167     boolean_t        si_numname;
2168 } us_sort_info_t;

2170 static int
2171 us_field_index(char *field)
2172 {

```

```

2173     int i;

2175     for (i = 0; i < USFIELD_LAST; i++) {
2176         if (strcmp(field, us_field_names[i]) == 0)
2177             return (i);
2178     }

2180     return (-1);
2181 }

2183 static int
2184 us_compare(const void *larg, const void *rarg, void *unused)
2185 {
2186     const us_node_t *l = larg;
2187     const us_node_t *r = rarg;
2188     us_sort_info_t *si = (us_sort_info_t *)unused;
2189     zfs_sort_column_t *sortcol = si->si_sortcol;
2190     boolean_t numname = si->si_numname;
2191     nvlist_t *lnvl = l->usn_nvl;
2192     nvlist_t *rnvl = r->usn_nvl;
2193     int rc = 0;
2194     boolean_t lvb, rvb;

2196     for (; sortcol != NULL; sortcol = sortcol->sc_next) {
2197         char *lvstr = "";
2198         char *rvstr = "";
2199         uint32_t lv32 = 0;
2200         uint32_t rv32 = 0;
2201         uint64_t lv64 = 0;
2202         uint64_t rv64 = 0;
2203         zfs_prop_t prop = sortcol->sc_prop;
2204         const char *propname = NULL;
2205         boolean_t reverse = sortcol->sc_reverse;

2207         switch (prop) {
2208             case ZFS_PROP_TYPE:
2209                 propname = "type";
2210                 (void) nvlist_lookup_uint32(lnvl, propname, &lv32);
2211                 (void) nvlist_lookup_uint32(rnvl, propname, &rv32);
2212                 if (rv32 != lv32)
2213                     rc = (rv32 < lv32) ? 1 : -1;
2214                 break;
2215             case ZFS_PROP_NAME:
2216                 propname = "name";
2217                 if (numname) {
2218                     (void) nvlist_lookup_uint64(lnvl, propname,
2219                         &lv64);
2220                     (void) nvlist_lookup_uint64(rnvl, propname,
2221                         &rv64);
2222                     if (rv64 != lv64)
2223                         rc = (rv64 < lv64) ? 1 : -1;
2224                 } else {
2225                     (void) nvlist_lookup_string(lnvl, propname,
2226                         &lvstr);
2227                     (void) nvlist_lookup_string(rnvl, propname,
2228                         &rvstr);
2229                     rc = strcmp(lvstr, rvstr);
2230                 }
2231                 break;
2232             case ZFS_PROP_USED:
2233             case ZFS_PROP_QUOTA:
2234                 if (!us_populated)
2235                     break;
2236                 if (prop == ZFS_PROP_USED)
2237                     propname = "used";
2238                 else

```

```

2239         propname = "quota";
2240         (void) nvlist_lookup_uint64(lnvl, propname, &lv64);
2241         (void) nvlist_lookup_uint64(rnvl, propname, &rv64);
2242         if (rv64 != lv64)
2243             rc = (rv64 < lv64) ? 1 : -1;
2244         break;
2245     }
2247     if (rc != 0) {
2248         if (rc < 0)
2249             return (reverse ? 1 : -1);
2250         else
2251             return (reverse ? -1 : 1);
2252     }
2253 }
2255 /*
2256  * If entries still seem to be the same, check if they are of the same
2257  * type (smbentity is added only if we are doing SID to POSIX ID
2258  * translation where we can have duplicate type/name combinations).
2259  */
2260 if (nvlist_lookup_boolean_value(lnvl, "smbentity", &lrv) == 0 &&
2261     nvlist_lookup_boolean_value(rnvl, "smbentity", &rvb) == 0 &&
2262     lrv != rvb)
2263     return (lrv < rvb ? -1 : 1);
2265 return (0);
2266 }
2268 static inline const char *
2269 us_type2str(unsigned field_type)
2270 {
2271     switch (field_type) {
2272     case USTYPE_PXK_USR:
2273         return ("POSIX User");
2274     case USTYPE_PXK_GRP:
2275         return ("POSIX Group");
2276     case USTYPE_SMB_USR:
2277         return ("SMB User");
2278     case USTYPE_SMB_GRP:
2279         return ("SMB Group");
2280     default:
2281         return ("Undefined");
2282     }
2283 }
2285 static int
2286 userspace_cb(void *arg, const char *domain, uid_t rid, uint64_t space)
2287 {
2288     us_cbdata_t *cb = (us_cbdata_t *)arg;
2289     zfs_userquota_prop_t prop = cb->cb_prop;
2290     char *name = NULL;
2291     char *propname;
2292     char sizebuf[32];
2293     us_node_t *node;
2294     uu_avl_pool_t *avl_pool = cb->cb_avl_pool;
2295     uu_avl_t *avl = cb->cb_avl;
2296     uu_avl_index_t idx;
2297     nvlist_t *props;
2298     us_node_t *n;
2299     zfs_sort_column_t *sortcol = cb->cb_sortcol;
2300     unsigned type;
2301     const char *typestr;
2302     size_t namelen;
2303     size_t typelen;
2304     size_t sizelen;

```

```

2305     int typeidx, nameidx, sizeidx;
2306     us_sort_info_t sortinfo = { sortcol, cb->cb_numname };
2307     boolean_t smbentity = B_FALSE;
2309     if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0)
2310         nomem();
2311     node = safe_malloc(sizeof (us_node_t));
2312     uu_avl_node_init(node, &node->usn_avlnode, avl_pool);
2313     node->usn_nvl = props;
2315     if (domain != NULL && domain[0] != '\0') {
2316         /* SMB */
2317         char sid[ZFS_MAXNAMELEN + 32];
2318         uid_t id;
2319         uint64_t classes;
2320         int err;
2321         directory_error_t e;
2323         smbentity = B_TRUE;
2325         (void) snprintf(sid, sizeof (sid), "%s-%u", domain, rid);
2327         if (prop == ZFS_PROP_GROUPUSED || prop == ZFS_PROP_GROUPQUOTA) {
2328             type = USTYPE_SMB_GRP;
2329             err = sid_to_id(sid, B_FALSE, &id);
2330         } else {
2331             type = USTYPE_SMB_USR;
2332             err = sid_to_id(sid, B_TRUE, &id);
2333         }
2335         if (err == 0) {
2336             rid = id;
2337             if (!cb->cb_sid2posix) {
2338                 e = directory_name_from_sid(NULL, sid, &name,
2339                     &classes);
2340                 if (e != NULL)
2341                     directory_error_free(e);
2342                 if (name == NULL)
2343                     name = sid;
2344             }
2345         }
2346     }
2348     if (cb->cb_sid2posix || domain == NULL || domain[0] == '\0') {
2349         /* POSIX or -i */
2350         if (prop == ZFS_PROP_GROUPUSED || prop == ZFS_PROP_GROUPQUOTA) {
2351             type = USTYPE_PXK_GRP;
2352             if (!cb->cb_numname) {
2353                 struct group *g;
2355                 if ((g = getgrgid(rid)) != NULL)
2356                     name = g->gr_name;
2357             }
2358         } else {
2359             type = USTYPE_PXK_USR;
2360             if (!cb->cb_numname) {
2361                 struct passwd *p;
2363                 if ((p = getpwuid(rid)) != NULL)
2364                     name = p->pw_name;
2365             }
2366         }
2367     }
2369     /*
2370      * Make sure that the type/name combination is unique when doing

```

```

2371  * SID to POSIX ID translation (hence changing the type from SMB to
2372  * POSIX).
2373  */
2374  if (cb->cb_sid2posix &&
2375      nvlist_add_boolean_value(props, "smbentity", smbentity) != 0)
2376      nomem();

2378  /* Calculate/update width of TYPE field */
2379  typestr = us_type2str(type);
2380  typelen = strlen(gettext(typestr));
2381  typeidx = us_field_index("type");
2382  if (typelen > cb->cb_width[typeidx])
2383      cb->cb_width[typeidx] = typelen;
2384  if (nvlist_add_uint32(props, "type", type) != 0)
2385      nomem();

2387  /* Calculate/update width of NAME field */
2388  if ((cb->cb_numname && cb->cb_sid2posix) || name == NULL) {
2389      if (nvlist_add_uint64(props, "name", rid) != 0)
2390          nomem();
2391      namelen = snprintf(NULL, 0, "%u", rid);
2392  } else {
2393      if (nvlist_add_string(props, "name", name) != 0)
2394          nomem();
2395      namelen = strlen(name);
2396  }
2397  nameidx = us_field_index("name");
2398  if (namelen > cb->cb_width[nameidx])
2399      cb->cb_width[nameidx] = namelen;

2401  /*
2402   * Check if this type/name combination is in the list and update it;
2403   * otherwise add new node to the list.
2404   */
2405  if ((n = uu_avl_find(avl, node, &sortinfo, &idx)) == NULL) {
2406      uu_avl_insert(avl, node, idx);
2407  } else {
2408      nvlist_free(props);
2409      free(node);
2410      node = n;
2411      props = node->usn_nvl;
2412  }

2414  /* Calculate/update width of USED/QUOTA fields */
2415  if (cb->cb_nicenum)
2416      zfs_nicenum(space, sizebuf, sizeof (sizebuf));
2417  else
2418      (void) snprintf(sizebuf, sizeof (sizebuf), "%llu", space);
2419  sizelen = strlen(sizebuf);
2420  if (prop == ZFS_PROP_USERUSED || prop == ZFS_PROP_GROUPUSED) {
2421      propname = "used";
2422      if (!nvlist_exists(props, "quota"))
2423          (void) nvlist_add_uint64(props, "quota", 0);
2424  } else {
2425      propname = "quota";
2426      if (!nvlist_exists(props, "used"))
2427          (void) nvlist_add_uint64(props, "used", 0);
2428  }
2429  sizeidx = us_field_index(propname);
2430  if (sizelen > cb->cb_width[sizeidx])
2431      cb->cb_width[sizeidx] = sizelen;

2433  if (nvlist_add_uint64(props, propname, space) != 0)
2434      nomem();

2436  return (0);

```

```

2437  }

2439  static void
2440  print_us_node(boolean_t scripted, boolean_t parsable, int *fields, int types,
2441               size_t *width, us_node_t *node)
2442  {
2443      nvlist_t *nvl = node->usn_nvl;
2444      char valstr[ZFS_MAXNAMELEN];
2445      boolean_t first = B_TRUE;
2446      int cfield = 0;
2447      int field;
2448      uint32_t ustype;

2450      /* Check type */
2451      (void) nvlist_lookup_uint32(nvl, "type", &ustype);
2452      if (!(ustype & types))
2453          return;

2455      while ((field = fields[cfield]) != USFIELD_LAST) {
2456          nvpair_t *nvp = NULL;
2457          data_type_t type;
2458          uint32_t val32;
2459          uint64_t val64;
2460          char *strval = NULL;

2462          while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
2463              if (strcmp(nvpair_name(nvp),
2464                          us_field_names[field]) == 0)
2465                  break;
2466          }

2468          type = nvpair_type(nvp);
2469          switch (type) {
2470              case DATA_TYPE_UINT32:
2471                  (void) nvpair_value_uint32(nvp, &val32);
2472                  break;
2473              case DATA_TYPE_UINT64:
2474                  (void) nvpair_value_uint64(nvp, &val64);
2475                  break;
2476              case DATA_TYPE_STRING:
2477                  (void) nvpair_value_string(nvp, &strval);
2478                  break;
2479              default:
2480                  (void) fprintf(stderr, "invalid data type\n");
2481          }

2483          switch (field) {
2484              case USFIELD_TYPE:
2485                  strval = (char *)us_type2str(val32);
2486                  break;
2487              case USFIELD_NAME:
2488                  if (type == DATA_TYPE_UINT64) {
2489                      (void) sprintf(valstr, "%llu", val64);
2490                      strval = valstr;
2491                  }
2492                  break;
2493              case USFIELD_USED:
2494              case USFIELD_QUOTA:
2495                  if (type == DATA_TYPE_UINT64) {
2496                      if (parsable) {
2497                          (void) sprintf(valstr, "%llu", val64);
2498                      } else {
2499                          zfs_nicenum(val64, valstr,
2500                                      sizeof (valstr));
2501                      }
2502                  }
2503                  if (field == USFIELD_QUOTA &&

```

```

2503         strcmp(valstr, "0") == 0)
2504             strval = "none";
2505     else
2506         strval = valstr;
2507     }
2508     break;
2509 }

2511     if (!first) {
2512         if (scripted)
2513             (void) printf("\t");
2514         else
2515             (void) printf(" ");
2516     }
2517     if (scripted)
2518         (void) printf("%s", strval);
2519     else if (field == USFIELD_TYPE || field == USFIELD_NAME)
2520         (void) printf("%-*s", width[field], strval);
2521     else
2522         (void) printf("%*s", width[field], strval);

2524     first = B_FALSE;
2525     cfield++;
2526 }

2528     (void) printf("\n");
2529 }

2531 static void
2532 print_us(boolean_t scripted, boolean_t parsable, int *fields, int types,
2533         size_t *width, boolean_t rmnode, uu_avl_t *avl)
2534 {
2535     us_node_t *node;
2536     const char *col;
2537     int cfield = 0;
2538     int field;

2540     if (!scripted) {
2541         boolean_t first = B_TRUE;

2543         while ((field = fields[cfield]) != USFIELD_LAST) {
2544             col = gettext(us_field_hdr[field]);
2545             if (field == USFIELD_TYPE || field == USFIELD_NAME) {
2546                 (void) printf(first ? "%-*s" : " %-*s",
2547                     width[field], col);
2548             } else {
2549                 (void) printf(first ? "%*s" : " %*s",
2550                     width[field], col);
2551             }
2552             first = B_FALSE;
2553             cfield++;
2554         }
2555         (void) printf("\n");
2556     }

2558     for (node = uu_avl_first(avl); node; node = uu_avl_next(avl, node)) {
2559         print_us_node(scripted, parsable, fields, types, width, node);
2560         if (rmnode)
2561             nvlist_free(node->usn_nvlist);
2562     }
2563 }

2565 static int
2566 zfs_do_userspace(int argc, char **argv)
2567 {
2568     zfs_handle_t *zhp;

```

```

2569     zfs_userquota_prop_t p;
2570     uu_avl_pool_t *avl_pool;
2571     uu_avl_t *avl_tree;
2572     uu_avl_walk_t *walk;
2573     char *delim;
2574     char deffields[] = "type,name,used,quota";
2575     char *ofield = NULL;
2576     char *tfield = NULL;
2577     int cfield = 0;
2578     int fields[256];
2579     int i;
2580     boolean_t scripted = B_FALSE;
2581     boolean_t prtnum = B_FALSE;
2582     boolean_t parsable = B_FALSE;
2583     boolean_t sid2posix = B_FALSE;
2584     int ret = 0;
2585     int c;
2586     zfs_sort_column_t *sortcol = NULL;
2587     int types = USTYPE_PXS_USR | USTYPE_SMB_USR;
2588     us_cbdata_t cb;
2589     us_node_t *node;
2590     us_node_t *rmnode;
2591     uu_list_pool_t *listpool;
2592     uu_list_t *list;
2593     uu_avl_index_t idx = 0;
2594     uu_list_index_t idx2 = 0;

2596     if (argc < 2)
2597         usage(B_FALSE);

2599     if (strcmp(argv[0], "groupspace") == 0)
2600         /* Toggle default group types */
2601         types = USTYPE_PXS_GRP | USTYPE_SMB_GRP;

2603     while ((c = getopt(argc, argv, "nHpo:s:t:i")) != -1) {
2604         switch (c) {
2605             case 'n':
2606                 prtnum = B_TRUE;
2607                 break;
2608             case 'H':
2609                 scripted = B_TRUE;
2610                 break;
2611             case 'p':
2612                 parsable = B_TRUE;
2613                 break;
2614             case 'o':
2615                 ofield = optarg;
2616                 break;
2617             case 's':
2618             case 'S':
2619                 if (zfs_add_sort_column(&sortcol, optarg,
2620                     c == 's' ? B_FALSE : B_TRUE) != 0) {
2621                     (void) fprintf(stderr,
2622                         gettext("invalid field '%s'\n"), optarg);
2623                     usage(B_FALSE);
2624                 }
2625                 break;
2626             case 't':
2627                 tfield = optarg;
2628                 break;
2629             case 'i':
2630                 sid2posix = B_TRUE;
2631                 break;
2632             case ':':
2633                 (void) fprintf(stderr, gettext("missing argument for "
2634                     "'%c' option\n"), optopt);

```

```

2635         usage(B_FALSE);
2636         break;
2637     case '?:
2638         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
2639             optopt);
2640         usage(B_FALSE);
2641     }
2642 }
2643
2644 argc -= optind;
2645 argv += optind;
2646
2647 if (argc < 1) {
2648     (void) fprintf(stderr, gettext("missing dataset name\n"));
2649     usage(B_FALSE);
2650 }
2651 if (argc > 1) {
2652     (void) fprintf(stderr, gettext("too many arguments\n"));
2653     usage(B_FALSE);
2654 }
2655
2656 /* Use default output fields if not specified using -o */
2657 if (ofield == NULL)
2658     ofield = deffields;
2659 do {
2660     if ((delim = strchr(ofield, ',')) != NULL)
2661         *delim = '\0';
2662     if ((fields[cfield++] = us_field_index(ofield)) == -1) {
2663         (void) fprintf(stderr, gettext("invalid type '%s' "
2664             "for -o option\n"), ofield);
2665         return (-1);
2666     }
2667     if (delim != NULL)
2668         ofield = delim + 1;
2669 } while (delim != NULL);
2670 fields[cfield] = USFIELD_LAST;
2671
2672 /* Override output types (-t option) */
2673 if (tfield != NULL) {
2674     types = 0;
2675     do {
2676         boolean_t found = B_FALSE;
2677
2678         if ((delim = strchr(tfield, ',')) != NULL)
2679             *delim = '\0';
2680         for (i = 0; i < sizeof(us_type_bits) / sizeof(int);
2681             i++) {
2682             if (strcmp(tfield, us_type_names[i]) == 0) {
2683                 found = B_TRUE;
2684                 types |= us_type_bits[i];
2685                 break;
2686             }
2687         }
2688         if (!found) {
2689             (void) fprintf(stderr, gettext("invalid type "
2690                 "'%s' for -t option\n"), tfield);
2691             return (-1);
2692         }
2693         if (delim != NULL)
2694             tfield = delim + 1;
2695     } while (delim != NULL);
2696 }
2697
2698 if ((zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_DATASET)) == NULL)
2699     return (1);

```

```

2702     if ((avl_pool = uu_avl_pool_create("us_avl_pool", sizeof(us_node_t),
2703         offsetof(us_node_t, usn_avlnode), us_compare, UU_DEFAULT)) == NULL)
2704         nomem();
2705     if ((avl_tree = uu_avl_create(avl_pool, NULL, UU_DEFAULT)) == NULL)
2706         nomem();
2707
2708     /* Always add default sorting columns */
2709     (void) zfs_add_sort_column(&sortcol, "type", B_FALSE);
2710     (void) zfs_add_sort_column(&sortcol, "name", B_FALSE);
2711
2712     cb.cb_sortcol = sortcol;
2713     cb.cb_numname = prtnum;
2714     cb.cb_nicenum = !parsable;
2715     cb.cb_avl_pool = avl_pool;
2716     cb.cb_avl = avl_tree;
2717     cb.cb_sid2posix = sid2posix;
2718
2719     for (i = 0; i < USFIELD_LAST; i++)
2720         cb.cb_width[i] = strlen(gettext(us_field_hdr[i]));
2721
2722     for (p = 0; p < ZFS_NUM_USERQUOTA_PROPS; p++) {
2723         if (((p == ZFS_PROP_USERUSED | p == ZFS_PROP_USERQUOTA) &&
2724             !(types & (USTYPE_PXS_USR | USTYPE_SMB_USR))) |
2725             ((p == ZFS_PROP_GROUPUSED | p == ZFS_PROP_GROUPQUOTA) &&
2726             !(types & (USTYPE_PXS_GRP | USTYPE_SMB_GRP))))
2727             continue;
2728         cb.cb_prop = p;
2729         if ((ret = zfs_userspace(zhp, p, userspace_cb, &cb)) != 0)
2730             return (ret);
2731     }
2732
2733     /* Sort the list */
2734     if ((node = uu_avl_first(avl_tree)) == NULL)
2735         return (0);
2736
2737     us_populated = B_TRUE;
2738
2739     listpool = uu_list_pool_create("tmplist", sizeof(us_node_t),
2740         offsetof(us_node_t, usn_listnode), NULL, UU_DEFAULT);
2741     list = uu_list_create(listpool, NULL, UU_DEFAULT);
2742     uu_list_node_init(node, &node->usn_listnode, listpool);
2743
2744     while (node != NULL) {
2745         rmnode = node;
2746         node = uu_avl_next(avl_tree, node);
2747         uu_avl_remove(avl_tree, rmnode);
2748         if (uu_list_find(list, rmnode, NULL, &idx2) == NULL)
2749             uu_list_insert(list, rmnode, idx2);
2750     }
2751
2752     for (node = uu_list_first(list); node != NULL;
2753         node = uu_list_next(list, node)) {
2754         us_sort_info_t sortinfo = { sortcol, cb.cb_numname };
2755
2756         if (uu_avl_find(avl_tree, node, &sortinfo, &idx) == NULL)
2757             uu_avl_insert(avl_tree, node, idx);
2758     }
2759
2760     uu_list_destroy(list);
2761     uu_list_pool_destroy(listpool);
2762
2763     /* Print and free node nvlist memory */
2764     print_us(scripted, parsable, fields, types, cb.cb_width, B_TRUE,
2765         cb.cb_avl);

```

```

2767     zfs_free_sort_columns(sortcol);

2769     /* Clean up the AVL tree */
2770     if ((walk = uu_avl_walk_start(cb.cb_avl, UU_WALK_ROBUST)) == NULL)
2771         nomem();

2773     while ((node = uu_avl_walk_next(walk)) != NULL) {
2774         uu_avl_remove(cb.cb_avl, node);
2775         free(node);
2776     }

2778     uu_avl_walk_end(walk);
2779     uu_avl_destroy(avl_tree);
2780     uu_avl_pool_destroy(avl_pool);

2782     return (ret);
2783 }

2785 /*
2786  * list [-r][-d max] [-H] [-o property[,property]...] [-t type[,type]...]
2787  *      [-s property [-s property]...] [-S property [-S property]...]
2788  *      <dataset> ...
2789  *
2790  *      -r      Recurse over all children
2791  *      -d      Limit recursion by depth.
2792  *      -H      Scripted mode; elide headers and separate columns by tabs
2793  *      -o      Control which fields to display.
2794  *      -t      Control which object types to display.
2795  *      -s      Specify sort columns, descending order.
2796  *      -S      Specify sort columns, ascending order.
2797  *
2798  *      When given no arguments, lists all filesystems in the system.
2799  *      Otherwise, list the specified datasets, optionally recursing down them if
2800  *      '-r' is specified.
2801  */
2802 typedef struct list_cbdata {
2803     boolean_t      cb_first;
2804     boolean_t      cb_scripted;
2805     zprop_list_t   *cb_proplist;
2806 } list_cbdata_t;

2808 /*
2809  * Given a list of columns to display, output appropriate headers for each one.
2810  */
2811 static void
2812 print_header(zprop_list_t *pl)
2813 {
2814     char headerbuf[ZFS_MAXPROPLEN];
2815     const char *header;
2816     int i;
2817     boolean_t first = B_TRUE;
2818     boolean_t right_justify;

2820     for (; pl != NULL; pl = pl->pl_next) {
2821         if (!first) {
2822             (void) printf(" ");
2823         } else {
2824             first = B_FALSE;
2825         }

2827         right_justify = B_FALSE;
2828         if (pl->pl_prop != ZPROP_INVAL) {
2829             header = zfs_prop_column_name(pl->pl_prop);
2830             right_justify = zfs_prop_align_right(pl->pl_prop);
2831         } else {
2832             for (i = 0; pl->pl_user_prop[i] != '\0'; i++)

```

```

2833         headerbuf[i] = toupper(pl->pl_user_prop[i]);
2834         headerbuf[i] = '\0';
2835         header = headerbuf;
2836     }

2838     if (pl->pl_next == NULL && !right_justify)
2839         (void) printf("%s", header);
2840     else if (right_justify)
2841         (void) printf("%*s", pl->pl_width, header);
2842     else
2843         (void) printf("%-*s", pl->pl_width, header);
2844     }

2846     (void) printf("\n");
2847 }

2849 /*
2850  * Given a dataset and a list of fields, print out all the properties according
2851  * to the described layout.
2852  */
2853 static void
2854 print_dataset(zfs_handle_t *zhp, zprop_list_t *pl, boolean_t scripted)
2855 {
2856     boolean_t first = B_TRUE;
2857     char property[ZFS_MAXPROPLEN];
2858     nvlist_t *userprops = zfs_get_user_props(zhp);
2859     nvlist_t *propval;
2860     char *propstr;
2861     boolean_t right_justify;
2862     int width;

2864     for (; pl != NULL; pl = pl->pl_next) {
2865         if (!first) {
2866             if (scripted)
2867                 (void) printf("\t");
2868             else
2869                 (void) printf(" ");
2870         } else {
2871             first = B_FALSE;
2872         }

2874         if (pl->pl_prop != ZPROP_INVAL) {
2875             if (zfs_prop_get(zhp, pl->pl_prop, property,
2876                 sizeof (property), NULL, NULL, 0, B_FALSE) != 0)
2877                 propstr = "-";
2878             else
2879                 propstr = property;

2881             right_justify = zfs_prop_align_right(pl->pl_prop);
2882         } else if (zfs_prop_userquota(pl->pl_user_prop)) {
2883             if (zfs_prop_get_userquota(zhp, pl->pl_user_prop,
2884                 property, sizeof (property), B_FALSE) != 0)
2885                 propstr = "-";
2886             else
2887                 propstr = property;
2888             right_justify = B_TRUE;
2889         } else if (zfs_prop_written(pl->pl_user_prop)) {
2890             if (zfs_prop_get_written(zhp, pl->pl_user_prop,
2891                 property, sizeof (property), B_FALSE) != 0)
2892                 propstr = "-";
2893             else
2894                 propstr = property;
2895             right_justify = B_TRUE;
2896         } else {
2897             if (nvlist_lookup_nvlist(userprops,
2898                 pl->pl_user_prop, &propval) != 0)

```



```

2899         propstr = "-";
2900     else
2901         verify(nvlist_lookup_string(propval,
2902             ZPROP_VALUE, &propstr) == 0);
2903         right_justify = B_FALSE;
2904     }
2905
2906     width = pl->pl_width;
2907
2908     /*
2909     * If this is being called in scripted mode, or if this is the
2910     * last column and it is left-justified, don't include a width
2911     * format specifier.
2912     */
2913     if (scripted || (pl->pl_next == NULL && !right_justify))
2914         (void) printf("%s", propstr);
2915     else if (right_justify)
2916         (void) printf("%*s", width, propstr);
2917     else
2918         (void) printf("%-*s", width, propstr);
2919 }
2920
2921 (void) printf("\n");
2922 }
2923
2924 /*
2925 * Generic callback function to list a dataset or snapshot.
2926 */
2927 static int
2928 list_callback(zfs_handle_t *zhp, void *data)
2929 {
2930     list_cbdata_t *cbp = data;
2931
2932     if (cbp->cb_first) {
2933         if (!cbp->cb_scripted)
2934             print_header(cbp->cb_proplist);
2935         cbp->cb_first = B_FALSE;
2936     }
2937
2938     print_dataset(zhp, cbp->cb_proplist, cbp->cb_scripted);
2939
2940     return (0);
2941 }
2942
2943 static int
2944 zfs_do_list(int argc, char **argv)
2945 {
2946     int c;
2947     boolean_t scripted = B_FALSE;
2948     static char default_fields[] =
2949         "name,used,available,referenced,mountpoint";
2950     int types = ZFS_TYPE_DATASET;
2951     boolean_t types_specified = B_FALSE;
2952     char *fields = NULL;
2953     list_cbdata_t cb = { 0 };
2954     char *value;
2955     int limit = 0;
2956     int ret = 0;
2957     zfs_sort_column_t *sortcol = NULL;
2958     int flags = ZFS_ITER_PROP_LISTSNAPS | ZFS_ITER_ARGS_CAN_BE_PATHS;
2959
2960     /* check options */
2961     while ((c = getopt(argc, argv, ":d:ort:Hs:S:")) != -1) {
2962         switch (c) {
2963             case 'o':
2964                 fields = optarg;

```

```

2965         break;
2966     case 'd':
2967         limit = parse_depth(optarg, &flags);
2968         break;
2969     case 'r':
2970         flags |= ZFS_ITER_RECURSE;
2971         break;
2972     case 'H':
2973         scripted = B_TRUE;
2974         break;
2975     case 's':
2976         if (zfs_add_sort_column(&sortcol, optarg,
2977             B_FALSE) != 0) {
2978             (void) fprintf(stderr,
2979                 gettext("invalid property '%s'\n"), optarg);
2980             usage(B_FALSE);
2981         }
2982         break;
2983     case 'S':
2984         if (zfs_add_sort_column(&sortcol, optarg,
2985             B_TRUE) != 0) {
2986             (void) fprintf(stderr,
2987                 gettext("invalid property '%s'\n"), optarg);
2988             usage(B_FALSE);
2989         }
2990         break;
2991     case 't':
2992         types = 0;
2993         types_specified = B_TRUE;
2994         flags &= ~ZFS_ITER_PROP_LISTSNAPS;
2995         while (*optarg != '\0') {
2996             static char *type_subopts[] = { "filesystem",
2997                 "volume", "snapshot", "all", NULL };
2998
2999             switch (getsubopt(&optarg, type_subopts,
3000                 &value)) {
3001                 case 0:
3002                     types |= ZFS_TYPE_FILESYSTEM;
3003                     break;
3004                 case 1:
3005                     types |= ZFS_TYPE_VOLUME;
3006                     break;
3007                 case 2:
3008                     types |= ZFS_TYPE_SNAPSHOT;
3009                     break;
3010                 case 3:
3011                     types = ZFS_TYPE_DATASET;
3012                     break;
3013
3014                 default:
3015                     (void) fprintf(stderr,
3016                         gettext("invalid type '%s'\n"),
3017                         value);
3018                     usage(B_FALSE);
3019             }
3020         }
3021         break;
3022     case ':':
3023         (void) fprintf(stderr, gettext("missing argument for "
3024             "'%c' option\n"), optopt);
3025         usage(B_FALSE);
3026         break;
3027     case '?':
3028         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3029             optopt);
3030         usage(B_FALSE);

```

```

3031     }
3032 }
3034 argc -= optind;
3035 argv += optind;
3037 if (fields == NULL)
3038     fields = default_fields;
3040 /*
3041  * If "-o space" and no types were specified, don't display snapshots.
3042  */
3043 if (strcmp(fields, "space") == 0 && types_specified == B_FALSE)
3044     types &= ~ZFS_TYPE_SNAPSHOT;
3046 /*
3047  * If the user specifies '-o all', the zprop_get_list() doesn't
3048  * normally include the name of the dataset. For 'zfs list', we always
3049  * want this property to be first.
3050  */
3051 if (zprop_get_list(g_zfs, fields, &cb.cb_proplist, ZFS_TYPE_DATASET)
3052     != 0)
3053     usage(B_FALSE);
3055 cb.cb_scripted = scripted;
3056 cb.cb_first = B_TRUE;
3058 ret = zfs_for_each(argc, argv, flags, types, sortcol, &cb.cb_proplist,
3059     limit, list_callback, &cb);
3061 zprop_free_list(cb.cb_proplist);
3062 zfs_free_sort_columns(sortcol);
3064 if (ret == 0 && cb.cb_first && !cb.cb_scripted)
3065     (void) printf(gettext("no datasets available\n"));
3067 return (ret);
3068 }
3070 /*
3071  * zfs rename [-f] <fs | snap | vol> <fs | snap | vol>
3072  * zfs rename [-f] -p <fs | vol> <fs | vol>
3073  * zfs rename -r <snap> <snap>
3074  *
3075  * Renames the given dataset to another of the same type.
3076  *
3077  * The '-p' flag creates all the non-existing ancestors of the target first.
3078  */
3079 /* ARGSUSED */
3080 static int
3081 zfs_do_rename(int argc, char **argv)
3082 {
3083     zfs_handle_t *zhp;
3084     int c;
3085     int ret = 0;
3086     boolean_t recurse = B_FALSE;
3087     boolean_t parents = B_FALSE;
3088     boolean_t force_unmount = B_FALSE;
3090     /* check options */
3091     while ((c = getopt(argc, argv, "prf")) != -1) {
3092         switch (c) {
3093             case 'p':
3094                 parents = B_TRUE;
3095                 break;
3096             case 'r':

```

```

3097         recurse = B_TRUE;
3098         break;
3099     case 'f':
3100         force_unmount = B_TRUE;
3101         break;
3102     case '?':
3103     default:
3104         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3105             optopt);
3106         usage(B_FALSE);
3107     }
3108 }
3110 argc -= optind;
3111 argv += optind;
3113 /* check number of arguments */
3114 if (argc < 1) {
3115     (void) fprintf(stderr, gettext("missing source dataset "
3116         "argument\n"));
3117     usage(B_FALSE);
3118 }
3119 if (argc < 2) {
3120     (void) fprintf(stderr, gettext("missing target dataset "
3121         "argument\n"));
3122     usage(B_FALSE);
3123 }
3124 if (argc > 2) {
3125     (void) fprintf(stderr, gettext("too many arguments\n"));
3126     usage(B_FALSE);
3127 }
3129 if (recurse && parents) {
3130     (void) fprintf(stderr, gettext("-p and -r options are mutually "
3131         "exclusive\n"));
3132     usage(B_FALSE);
3133 }
3135 if (recurse && strchr(argv[0], '@') == 0) {
3136     (void) fprintf(stderr, gettext("source dataset for recursive "
3137         "rename must be a snapshot\n"));
3138     usage(B_FALSE);
3139 }
3141 if ((zhp = zfs_open(g_zfs, argv[0], parents ? ZFS_TYPE_FILESYSTEM |
3142     ZFS_TYPE_VOLUME : ZFS_TYPE_DATASET)) == NULL)
3143     return (1);
3145 /* If we were asked and the name looks good, try to create ancestors. */
3146 if (parents && zfs_name_valid(argv[1], zfs_get_type(zhp)) &&
3147     zfs_create_ancestors(g_zfs, argv[1]) != 0) {
3148     zfs_close(zhp);
3149     return (1);
3150 }
3152 ret = (zfs_rename(zhp, argv[1], recurse, force_unmount) != 0);
3154 zfs_close(zhp);
3155 return (ret);
3156 }
3158 /*
3159  * zfs promote <fs>
3160  *
3161  * Promotes the given clone fs to be the parent
3162  */

```

```

3163 /* ARGSUSED */
3164 static int
3165 zfs_do_promote(int argc, char **argv)
3166 {
3167     zfs_handle_t *zhp;
3168     int ret = 0;
3169
3170     /* check options */
3171     if (argc > 1 && argv[1][0] == '-') {
3172         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3173             argv[1][1]);
3174         usage(B_FALSE);
3175     }
3176
3177     /* check number of arguments */
3178     if (argc < 2) {
3179         (void) fprintf(stderr, gettext("missing clone filesystem"
3180             " argument\n"));
3181         usage(B_FALSE);
3182     }
3183     if (argc > 2) {
3184         (void) fprintf(stderr, gettext("too many arguments\n"));
3185         usage(B_FALSE);
3186     }
3187
3188     zhp = zfs_open(g_zfs, argv[1], ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3189     if (zhp == NULL)
3190         return (1);
3191
3192     ret = (zfs_promote(zhp) != 0);
3193
3194     zfs_close(zhp);
3195     return (ret);
3196 }
3197
3198 /*
3199  * zfs rollback [-rRf] <snapshot>
3200  *
3201  * -r Delete any intervening snapshots before doing rollback
3202  * -R Delete any snapshots and their clones
3203  * -f ignored for backwards compatibility
3204  *
3205  * Given a filesystem, rollback to a specific snapshot, discarding any changes
3206  * since then and making it the active dataset. If more recent snapshots exist,
3207  * the command will complain unless the '-r' flag is given.
3208  */
3209 typedef struct rollback_cbdata {
3210     uint64_t cb_create;
3211     boolean_t cb_first;
3212     int cb_doclones;
3213     char *cb_target;
3214     int cb_error;
3215     boolean_t cb_recurse;
3216     boolean_t cb_dependent;
3217 } rollback_cbdata_t;
3218
3219
3220 /*
3221  * Report any snapshots more recent than the one specified. Used when '-r' is
3222  * not specified. We reuse this same callback for the snapshot dependents - if
3223  * 'cb_dependent' is set, then this is a dependent and we should report it
3224  * without checking the transaction group.
3225  */
3226 static int
3227 rollback_check(zfs_handle_t *zhp, void *data)
3228 {

```

```

3229     rollback_cbdata_t *cbp = data;
3230
3231     if (cbp->cb_doclones) {
3232         zfs_close(zhp);
3233         return (0);
3234     }
3235
3236     if (!cbp->cb_dependent) {
3237         if (strcmp(zfs_get_name(zhp), cbp->cb_target) != 0 &&
3238             zfs_get_type(zhp) == ZFS_TYPE_SNAPSHOT &&
3239             zfs_prop_get_int(zhp, ZFS_PROP_CREATETXG) >
3240                 cbp->cb_create) {
3241
3242             if (cbp->cb_first && !cbp->cb_recurse) {
3243                 (void) fprintf(stderr, gettext("cannot "
3244                     "rollback to '%s': more recent snapshots "
3245                     "exist\n"),
3246                     cbp->cb_target);
3247                 (void) fprintf(stderr, gettext("use '-r' to "
3248                     "force deletion of the following "
3249                     "snapshots:\n"));
3250                 cbp->cb_first = 0;
3251                 cbp->cb_error = 1;
3252             }
3253
3254             if (cbp->cb_recurse) {
3255                 cbp->cb_dependent = B_TRUE;
3256                 if (zfs_iter_dependents(zhp, B_TRUE,
3257                     rollback_check, cbp) != 0) {
3258                     zfs_close(zhp);
3259                     return (-1);
3260                 }
3261                 cbp->cb_dependent = B_FALSE;
3262             } else {
3263                 (void) fprintf(stderr, "%s\n",
3264                     zfs_get_name(zhp));
3265             }
3266         }
3267     } else {
3268         if (cbp->cb_first && cbp->cb_recurse) {
3269             (void) fprintf(stderr, gettext("cannot rollback to "
3270                 "'%s': clones of previous snapshots exist\n"),
3271                 cbp->cb_target);
3272             (void) fprintf(stderr, gettext("use '-R' to "
3273                 "force deletion of the following clones and "
3274                 "dependents:\n"));
3275             cbp->cb_first = 0;
3276             cbp->cb_error = 1;
3277         }
3278
3279         (void) fprintf(stderr, "%s\n", zfs_get_name(zhp));
3280     }
3281
3282     zfs_close(zhp);
3283     return (0);
3284 }
3285
3286 static int
3287 zfs_do_rollback(int argc, char **argv)
3288 {
3289     int ret = 0;
3290     int c;
3291     boolean_t force = B_FALSE;
3292     rollback_cbdata_t cb = { 0 };
3293     zfs_handle_t *zhp, *snap;
3294     char parentname[ZFS_MAXNAMELEN];

```

```

3295     char *delim;
3297     /* check options */
3298     while ((c = getopt(argc, argv, "rRf")) != -1) {
3299         switch (c) {
3300             case 'r':
3301                 cb.cb_recurse = 1;
3302                 break;
3303             case 'R':
3304                 cb.cb_recurse = 1;
3305                 cb.cb_doclones = 1;
3306                 break;
3307             case 'f':
3308                 force = B_TRUE;
3309                 break;
3310             case '?':
3311                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3312                     optopt);
3313                 usage(B_FALSE);
3314             }
3315     }
3317     argc -= optind;
3318     argv += optind;
3320     /* check number of arguments */
3321     if (argc < 1) {
3322         (void) fprintf(stderr, gettext("missing dataset argument\n"));
3323         usage(B_FALSE);
3324     }
3325     if (argc > 1) {
3326         (void) fprintf(stderr, gettext("too many arguments\n"));
3327         usage(B_FALSE);
3328     }
3330     /* open the snapshot */
3331     if ((snap = zfs_open(g_zfs, argv[0], ZFS_TYPE_SNAPSHOT)) == NULL)
3332         return (1);
3334     /* open the parent dataset */
3335     (void) strncpy(parentname, argv[0], sizeof (parentname));
3336     verify((delim = strrchr(parentname, '@')) != NULL);
3337     *delim = '\0';
3338     if ((zhp = zfs_open(g_zfs, parentname, ZFS_TYPE_DATASET)) == NULL) {
3339         zfs_close(snap);
3340         return (1);
3341     }
3343     /*
3344      * Check for more recent snapshots and/or clones based on the presence
3345      * of '-r' and '-R'.
3346      */
3347     cb.cb_target = argv[0];
3348     cb.cb_create = zfs_prop_get_int(snap, ZFS_PROP_CREATETXG);
3349     cb.cb_first = B_TRUE;
3350     cb.cb_error = 0;
3351     if ((ret = zfs_iter_children(zhp, rollback_check, &cb)) != 0)
3352         goto out;
3354     if ((ret = cb.cb_error) != 0)
3355         goto out;
3357     /*
3358      * Rollback parent to the given snapshot.
3359      */
3360     ret = zfs_rollback(zhp, snap, force);

```

```

3362 out:
3363     zfs_close(snap);
3364     zfs_close(zhp);
3366     if (ret == 0)
3367         return (0);
3368     else
3369         return (1);
3370 }
3372 /*
3373  * zfs set property=value { fs | snap | vol } ...
3374  *
3375  * Sets the given property for all datasets specified on the command line.
3376  */
3377 typedef struct set_cbdata {
3378     char      *cb_propname;
3379     char      *cb_value;
3380 } set_cbdata_t;
3382 static int
3383 set_callback(zfs_handle_t *zhp, void *data)
3384 {
3385     set_cbdata_t *cbp = data;
3387     if (zfs_prop_set(zhp, cbp->cb_propname, cbp->cb_value) != 0) {
3388         switch (libzfs_errno(g_zfs)) {
3389             case EZFS_MOUNTFAILED:
3390                 (void) fprintf(stderr, gettext("property may be set "
3391                     "but unable to remount filesystem\n"));
3392                 break;
3393             case EZFS_SHARENFSFAILED:
3394                 (void) fprintf(stderr, gettext("property may be set "
3395                     "but unable to reshare filesystem\n"));
3396                 break;
3397             }
3398         return (1);
3399     }
3400     return (0);
3401 }
3403 static int
3404 zfs_do_set(int argc, char **argv)
3405 {
3406     set_cbdata_t cb;
3407     int ret = 0;
3409     /* check for options */
3410     if (argc > 1 && argv[1][0] == '-') {
3411         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3412             argv[1][1]);
3413         usage(B_FALSE);
3414     }
3416     /* check number of arguments */
3417     if (argc < 2) {
3418         (void) fprintf(stderr, gettext("missing property=value "
3419             "argument\n"));
3420         usage(B_FALSE);
3421     }
3422     if (argc < 3) {
3423         (void) fprintf(stderr, gettext("missing dataset name\n"));
3424         usage(B_FALSE);
3425     }

```

```

3427 /* validate property=value argument */
3428 cb.cb_propname = argv[1];
3429 if (((cb.cb_value = strchr(cb.cb_propname, '=') == NULL) ||
3430     (cb.cb_value[1] == '\\0')) {
3431     (void) fprintf(stderr, gettext("missing value in "
3432     "property=value argument\n"));
3433     usage(B_FALSE);
3434 }
3436 *cb.cb_value = '\\0';
3437 cb.cb_value++;
3439 if (*cb.cb_propname == '\\0') {
3440     (void) fprintf(stderr,
3441     gettext("missing property in property=value argument\n"));
3442     usage(B_FALSE);
3443 }
3445 ret = zfs_for_each(argc - 2, argv + 2, NULL,
3446     ZFS_TYPE_DATASET, NULL, NULL, 0, set_callback, &cb);
3448 return (ret);
3449 }
3451 typedef struct snap_cbdata {
3452     nvlist_t *sd_nvlist;
3453     boolean_t sd_recursive;
3454     const char *sd_snapname;
3455 } snap_cbdata_t;
3457 static int
3458 zfs_snapshot_cb(zfs_handle_t *zhp, void *arg)
3459 {
3460     snap_cbdata_t *sd = arg;
3461     char *name;
3462     int rv = 0;
3463     int error;
3465     error = asprintf(&name, "%s%s", zfs_get_name(zhp), sd->sd_snapname);
3466     if (error == -1)
3467         nomem();
3468     fnvlist_add_boolean(sd->sd_nvlist, name);
3469     free(name);
3471     if (sd->sd_recursive)
3472         rv = zfs_iter_filesystems(zhp, zfs_snapshot_cb, sd);
3473     zfs_close(zhp);
3474     return (rv);
3475 }
3477 /*
3478 * zfs snapshot [-r] [-o prop=value] ... <fs>[snap]
3479 * Creates a snapshot with the given name. While functionally equivalent to
3480 * 'zfs create', it is a separate command to differentiate intent.
3481 */
3483 static int
3484 zfs_do_snapshot(int argc, char **argv)
3485 {
3486     int ret = 0;
3487     char c;
3488     nvlist_t *props;
3489     snap_cbdata_t sd = { 0 };
3490     boolean_t multiple_snaps = B_FALSE;
3492     if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0)

```

```

3493     nomem();
3494     if (nvlist_alloc(&sd.sd_nvlist, NV_UNIQUE_NAME, 0) != 0)
3495         nomem();
3497 /* check options */
3498 while ((c = getopt(argc, argv, "ro:")) != -1) {
3499     switch (c) {
3500     case 'o':
3501         if (parseprop(props))
3502             return (1);
3503         break;
3504     case 'r':
3505         sd.sd_recursive = B_TRUE;
3506         multiple_snaps = B_TRUE;
3507         break;
3508     case '?':
3509         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3510             optopt);
3511         goto usage;
3512     }
3513 }
3515 argc -= optind;
3516 argv += optind;
3518 /* check number of arguments */
3519 if (argc < 1) {
3520     (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3521     goto usage;
3522 }
3524 if (argc > 1)
3525     multiple_snaps = B_TRUE;
3526 for (; argc > 0; argc--, argv++) {
3527     char *atp;
3528     zfs_handle_t *zhp;
3530     atp = strchr(argv[0], '@');
3531     if (atp == NULL)
3532         goto usage;
3533     *atp = '\\0';
3534     sd.sd_snapname = atp + 1;
3535     zhp = zfs_open(g_zfs, argv[0],
3536         ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3537     if (zhp == NULL)
3538         goto usage;
3539     if (zfs_snapshot_cb(zhp, &sd) != 0)
3540         goto usage;
3541 }
3543 ret = zfs_snapshot_nvlist(g_zfs, sd.sd_nvlist, props);
3544 nvlist_free(sd.sd_nvlist);
3545 nvlist_free(props);
3546 if (ret != 0 && multiple_snaps)
3547     (void) fprintf(stderr, gettext("no snapshots were created\n"));
3548 return (ret != 0);
3550 usage:
3551     nvlist_free(sd.sd_nvlist);
3552     nvlist_free(props);
3553     usage(B_FALSE);
3554     return (-1);
3555 }
3557 /*
3558 * Send a backup stream to stdout.

```

```

3559 */
3560 static int
3561 zfs_do_send(int argc, char **argv)
3562 {
3563     char *fromname = NULL;
3564     char *toname = NULL;
3565     char *cp;
3566     zfs_handle_t *zhp;
3567     sendflags_t flags = { 0 };
3568     int c, err;
3569     nvlist_t *dbgnv = NULL;
3570     boolean_t extraverbose = B_FALSE;

3572     /* check options */
3573     while ((c = getopt(argc, argv, ":i:IRpPvDn?:?") != -1) {
3574         switch (c) {
3575             case 'i':
3576                 if (fromname)
3577                     usage(B_FALSE);
3578                 fromname = optarg;
3579                 break;
3580             case 'I':
3581                 if (fromname)
3582                     usage(B_FALSE);
3583                 fromname = optarg;
3584                 flags.doall = B_TRUE;
3585                 break;
3586             case 'R':
3587                 flags.replicate = B_TRUE;
3588                 break;
3589             case 'p':
3590                 flags.props = B_TRUE;
3591                 break;
3592             case 'P':
3593                 flags.parsable = B_TRUE;
3594                 flags.verbose = B_TRUE;
3595                 break;
3596             case 'v':
3597                 if (flags.verbose)
3598                     extraverbose = B_TRUE;
3599                 flags.verbose = B_TRUE;
3600                 flags.progress = B_TRUE;
3601                 break;
3602             case 'D':
3603                 flags.dedup = B_TRUE;
3604                 break;
3605             case 'n':
3606                 flags.dryrun = B_TRUE;
3607                 break;
3608             case ':':
3609                 (void) fprintf(stderr, gettext("missing argument for "
3610                 "'%c' option\n"), optopt);
3611                 usage(B_FALSE);
3612                 break;
3613             case '?':
3614                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3615                 optopt);
3616                 usage(B_FALSE);
3617             }
3618     }

3620     argc -= optind;
3621     argv += optind;

3623     /* check number of arguments */
3624     if (argc < 1) {

```

```

3625         (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3626         usage(B_FALSE);
3627     }
3628     if (argc > 1) {
3629         (void) fprintf(stderr, gettext("too many arguments\n"));
3630         usage(B_FALSE);
3631     }

3633     if (!flags.dryrun && isatty(STDOUT_FILENO)) {
3634         (void) fprintf(stderr,
3635         gettext("Error: Stream can not be written to a terminal.\n"
3636         "You must redirect standard output.\n"));
3637         return (1);
3638     }

3640     cp = strchr(argv[0], '@');
3641     if (cp == NULL) {
3642         (void) fprintf(stderr,
3643         gettext("argument must be a snapshot\n"));
3644         usage(B_FALSE);
3645     }
3646     *cp = '\0';
3647     toname = cp + 1;
3648     zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3649     if (zhp == NULL)
3650         return (1);

3652     /*
3653      * If they specified the full path to the snapshot, chop off
3654      * everything except the short name of the snapshot, but special
3655      * case if they specify the origin.
3656      */
3657     if (fromname && (cp = strchr(fromname, '@')) != NULL) {
3658         char origin[ZFS_MAXNAMELEN];
3659         zprop_source_t src;

3661         (void) zfs_prop_get(zhp, ZFS_PROP_ORIGIN,
3662         origin, sizeof (origin), &src, NULL, 0, B_FALSE);

3664         if (strcmp(origin, fromname) == 0) {
3665             fromname = NULL;
3666             flags.fromorigin = B_TRUE;
3667         } else {
3668             *cp = '\0';
3669             if (cp != fromname && strcmp(argv[0], fromname)) {
3670                 (void) fprintf(stderr,
3671                 gettext("incremental source must be "
3672                 "in same filesystem\n"));
3673                 usage(B_FALSE);
3674             }
3675             fromname = cp + 1;
3676             if (strchr(fromname, '@') || strchr(fromname, '/')) {
3677                 (void) fprintf(stderr,
3678                 gettext("invalid incremental source\n"));
3679                 usage(B_FALSE);
3680             }
3681         }
3682     }

3684     if (flags.replicate && fromname == NULL)
3685         flags.doall = B_TRUE;

3687     err = zfs_send(zhp, fromname, toname, &flags, STDOUT_FILENO, NULL, 0,
3688     extraverbose ? &dbgnv : NULL);

3690     if (extraverbose && dbgnv != NULL) {

```

```

3691     /*
3692     * dump_nvlist prints to stdout, but that's been
3693     * redirected to a file. Make it print to stderr
3694     * instead.
3695     */
3696     (void) dup2(STDERR_FILENO, STDOUT_FILENO);
3697     dump_nvlist(dbgnv, 0);
3698     nvlist_free(dbgnv);
3699 }
3700 zfs_close(zhp);
3702 return (err != 0);
3703 }
3705 /*
3706 * zfs receive [-vnFu] [-d | -e] <fs@snap>
3707 *
3708 * Restore a backup stream from stdin.
3709 */
3710 static int
3711 zfs_do_receive(int argc, char **argv)
3712 {
3713     int c, err;
3714     recvflags_t flags = { 0 };
3716     /* check options */
3717     while ((c = getopt(argc, argv, ":denuvF")) != -1) {
3718         switch (c) {
3719             case 'd':
3720                 flags.isprefix = B_TRUE;
3721                 break;
3722             case 'e':
3723                 flags.isprefix = B_TRUE;
3724                 flags.istail = B_TRUE;
3725                 break;
3726             case 'n':
3727                 flags.dryrun = B_TRUE;
3728                 break;
3729             case 'u':
3730                 flags.nomount = B_TRUE;
3731                 break;
3732             case 'v':
3733                 flags.verbose = B_TRUE;
3734                 break;
3735             case 'F':
3736                 flags.force = B_TRUE;
3737                 break;
3738             case ':':
3739                 (void) fprintf(stderr, gettext("missing argument for "
3740                 "%c' option\n"), optopt);
3741                 usage(B_FALSE);
3742                 break;
3743             case '?':
3744                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3745                 optopt);
3746                 usage(B_FALSE);
3747             }
3748         }
3750     argc -= optind;
3751     argv += optind;
3753     /* check number of arguments */
3754     if (argc < 1) {
3755         (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3756         usage(B_FALSE);

```

```

3757     }
3758     if (argc > 1) {
3759         (void) fprintf(stderr, gettext("too many arguments\n"));
3760         usage(B_FALSE);
3761     }
3763     if (isatty(STDIN_FILENO)) {
3764         (void) fprintf(stderr,
3765         gettext("Error: Backup stream can not be read "
3766         "from a terminal.\n"
3767         "You must redirect standard input.\n"));
3768         return (1);
3769     }
3771     err = zfs_receive(g_zfs, argv[0], &flags, STDIN_FILENO, NULL);
3773     return (err != 0);
3774 }
3776 /*
3777 * allow/unallow stuff
3778 */
3779 /* copied from zfs/sys/dsl_deleg.h */
3780 #define ZFS_DELEG_PERM_CREATE "create"
3781 #define ZFS_DELEG_PERM_DESTROY "destroy"
3782 #define ZFS_DELEG_PERM_SNAPSHOT "snapshot"
3783 #define ZFS_DELEG_PERM_ROLLBACK "rollback"
3784 #define ZFS_DELEG_PERM_CLONE "clone"
3785 #define ZFS_DELEG_PERM_PROMOTE "promote"
3786 #define ZFS_DELEG_PERM_RENAME "rename"
3787 #define ZFS_DELEG_PERM_MOUNT "mount"
3788 #define ZFS_DELEG_PERM_SHARE "share"
3789 #define ZFS_DELEG_PERM_SEND "send"
3790 #define ZFS_DELEG_PERM_RECEIVE "receive"
3791 #define ZFS_DELEG_PERM_ALLOW "allow"
3792 #define ZFS_DELEG_PERM_USERPROP "userprop"
3793 #define ZFS_DELEG_PERM_VSCAN "vscan" /* ??? */
3794 #define ZFS_DELEG_PERM_USERQUOTA "userquota"
3795 #define ZFS_DELEG_PERM_GROUPQUOTA "groupquota"
3796 #define ZFS_DELEG_PERM_USERUSED "userused"
3797 #define ZFS_DELEG_PERM_GROUPUSED "groupused"
3798 #define ZFS_DELEG_PERM_HOLD "hold"
3799 #define ZFS_DELEG_PERM_RELEASE "release"
3800 #define ZFS_DELEG_PERM_DIFF "diff"
3802 #define ZFS_NUM_DELEG_NOTES ZFS_DELEG_NOTE_NONE
3804 static zfs_deleg_perm_tab_t zfs_deleg_perm_tbl[] = {
3805     { ZFS_DELEG_PERM_ALLOW, ZFS_DELEG_NOTE_ALLOW },
3806     { ZFS_DELEG_PERM_CLONE, ZFS_DELEG_NOTE_CLONE },
3807     { ZFS_DELEG_PERM_CREATE, ZFS_DELEG_NOTE_CREATE },
3808     { ZFS_DELEG_PERM_DESTROY, ZFS_DELEG_NOTE_DESTROY },
3809     { ZFS_DELEG_PERM_DIFF, ZFS_DELEG_NOTE_DIFF },
3810     { ZFS_DELEG_PERM_HOLD, ZFS_DELEG_NOTE_HOLD },
3811     { ZFS_DELEG_PERM_MOUNT, ZFS_DELEG_NOTE_MOUNT },
3812     { ZFS_DELEG_PERM_PROMOTE, ZFS_DELEG_NOTE_PROMOTE },
3813     { ZFS_DELEG_PERM_RECEIVE, ZFS_DELEG_NOTE_RECEIVE },
3814     { ZFS_DELEG_PERM_RELEASE, ZFS_DELEG_NOTE_RELEASE },
3815     { ZFS_DELEG_PERM_RENAME, ZFS_DELEG_NOTE_RENAME },
3816     { ZFS_DELEG_PERM_ROLLBACK, ZFS_DELEG_NOTE_ROLLBACK },
3817     { ZFS_DELEG_PERM_SEND, ZFS_DELEG_NOTE_SEND },
3818     { ZFS_DELEG_PERM_SHARE, ZFS_DELEG_NOTE_SHARE },
3819     { ZFS_DELEG_PERM_SNAPSHOT, ZFS_DELEG_NOTE_SNAPSHOT },
3821     { ZFS_DELEG_PERM_GROUPQUOTA, ZFS_DELEG_NOTE_GROUPQUOTA },
3822     { ZFS_DELEG_PERM_GROUPUSED, ZFS_DELEG_NOTE_GROUPUSED },

```

```

3823     { ZFS_DELEG_PERM_USERPROP, ZFS_DELEG_NOTE_USERPROP },
3824     { ZFS_DELEG_PERM_USERQUOTA, ZFS_DELEG_NOTE_USERQUOTA },
3825     { ZFS_DELEG_PERM_USERUSED, ZFS_DELEG_NOTE_USERUSED },
3826     { NULL, ZFS_DELEG_NOTE_NONE }
3827 };

3829 /* permission structure */
3830 typedef struct deleg_perm {
3831     zfs_deleg_who_type_t    dp_who_type;
3832     const char              *dp_name;
3833     boolean_t               dp_local;
3834     boolean_t               dp_descend;
3835 } deleg_perm_t;

3837 /* */
3838 typedef struct deleg_perm_node {
3839     deleg_perm_t            dpn_perm;
3841     uu_avl_node_t          dpn_avl_node;
3842 } deleg_perm_node_t;

3844 typedef struct fs_perm fs_perm_t;

3846 /* permissions set */
3847 typedef struct who_perm {
3848     zfs_deleg_who_type_t    who_type;
3849     const char              *who_name;           /* id */
3850     char                    who_ug_name[256];    /* user/group name */
3851     fs_perm_t               *who_fsperm;        /* uplink */

3853     uu_avl_t                *who_deleg_perm_avl; /* permissions */
3854 } who_perm_t;

3856 /* */
3857 typedef struct who_perm_node {
3858     who_perm_t              who_perm;
3859     uu_avl_node_t          who_avl_node;
3860 } who_perm_node_t;

3862 typedef struct fs_perm_set fs_perm_set_t;
3863 /* fs permissions */
3864 struct fs_perm {
3865     const char              *fsp_name;

3867     uu_avl_t                *fsp_sc_avl;        /* sets,create */
3868     uu_avl_t                *fsp_uge_avl;      /* user,group,everyone */

3870     fs_perm_set_t          *fsp_set;          /* uplink */
3871 };

3873 /* */
3874 typedef struct fs_perm_node {
3875     fs_perm_t               fspn_fsperm;
3876     uu_avl_t                *fspn_avl;

3878     uu_list_node_t         fspn_list_node;
3879 } fs_perm_node_t;

3881 /* top level structure */
3882 struct fs_perm_set {
3883     uu_list_pool_t         *fsps_list_pool;
3884     uu_list_t              *fsps_list; /* list of fs_perms */

3886     uu_avl_pool_t         *fsps_named_set_avl_pool;
3887     uu_avl_pool_t         *fsps_who_perm_avl_pool;
3888     uu_avl_pool_t         *fsps_deleg_perm_avl_pool;

```

```

3889 };

3891 static inline const char *
3892 deleg_perm_type(zfs_deleg_note_t note)
3893 {
3894     /* subcommands */
3895     switch (note) {
3896         /* SUBCOMMANDS */
3897         /* OTHER */
3898     case ZFS_DELEG_NOTE_GROUPQUOTA:
3899     case ZFS_DELEG_NOTE_GROUPUSED:
3900     case ZFS_DELEG_NOTE_USERPROP:
3901     case ZFS_DELEG_NOTE_USERQUOTA:
3902     case ZFS_DELEG_NOTE_USERUSED:
3903         /* other */
3904         return (gettext("other"));
3905     default:
3906         return (gettext("subcommand"));
3907     }
3908 }

3910 static int inline
3911 who_type2weight(zfs_deleg_who_type_t who_type)
3912 {
3913     int res;
3914     switch (who_type) {
3915     case ZFS_DELEG_NAMED_SET_SETS:
3916     case ZFS_DELEG_NAMED_SET:
3917         res = 0;
3918         break;
3919     case ZFS_DELEG_CREATE_SETS:
3920     case ZFS_DELEG_CREATE:
3921         res = 1;
3922         break;
3923     case ZFS_DELEG_USER_SETS:
3924     case ZFS_DELEG_USER:
3925         res = 2;
3926         break;
3927     case ZFS_DELEG_GROUP_SETS:
3928     case ZFS_DELEG_GROUP:
3929         res = 3;
3930         break;
3931     case ZFS_DELEG_EVERYONE_SETS:
3932     case ZFS_DELEG_EVERYONE:
3933         res = 4;
3934         break;
3935     default:
3936         res = -1;
3937     }

3939     return (res);
3940 }

3942 /* ARGSUSED */
3943 static int
3944 who_perm_compare(const void *larg, const void *rarg, void *unused)
3945 {
3946     const who_perm_node_t *l = larg;
3947     const who_perm_node_t *r = rarg;
3948     zfs_deleg_who_type_t ltype = l->who_perm.who_type;
3949     zfs_deleg_who_type_t rtype = r->who_perm.who_type;
3950     int lweight = who_type2weight(ltype);
3951     int rweight = who_type2weight(rtype);
3952     int res = lweight - rweight;
3953     if (res == 0)
3954         res = strcmp(l->who_perm.who_name, r->who_perm.who_name,

```



```

3955         ZFS_MAX_DELEG_NAME-1);
3957     if (res == 0)
3958         return (0);
3959     if (res > 0)
3960         return (1);
3961     else
3962         return (-1);
3963 }
3965 /* ARGSUSED */
3966 static int
3967 deleg_perm_compare(const void *larg, const void *rarg, void *unused)
3968 {
3969     const deleg_perm_node_t *l = larg;
3970     const deleg_perm_node_t *r = rarg;
3971     int res = strcmp(l->dpn_perm.dp_name, r->dpn_perm.dp_name,
3972         ZFS_MAX_DELEG_NAME-1);
3974     if (res == 0)
3975         return (0);
3977     if (res > 0)
3978         return (1);
3979     else
3980         return (-1);
3981 }
3983 static inline void
3984 fs_perm_set_init(fs_perm_set_t *fspset)
3985 {
3986     bzero(fspset, sizeof (fs_perm_set_t));
3988     if ((fspset->fsps_list_pool = uu_list_pool_create("fsps_list_pool",
3989         sizeof (fs_perm_node_t), offsetof(fs_perm_node_t, fspn_list_node),
3990         NULL, UU_DEFAULT)) == NULL)
3991         nomem();
3992     if ((fspset->fsps_list = uu_list_create(fspset->fsps_list_pool, NULL,
3993         UU_DEFAULT)) == NULL)
3994         nomem();
3996     if ((fspset->fsps_named_set_avl_pool = uu_avl_pool_create(
3997         "named_set_avl_pool", sizeof (who_perm_node_t), offsetof(
3998         who_perm_node_t, who_avl_node), who_perm_compare,
3999         UU_DEFAULT)) == NULL)
4000         nomem();
4002     if ((fspset->fsps_who_perm_avl_pool = uu_avl_pool_create(
4003         "who_perm_avl_pool", sizeof (who_perm_node_t), offsetof(
4004         who_perm_node_t, who_avl_node), who_perm_compare,
4005         UU_DEFAULT)) == NULL)
4006         nomem();
4008     if ((fspset->fsps_deleg_perm_avl_pool = uu_avl_pool_create(
4009         "deleg_perm_avl_pool", sizeof (deleg_perm_node_t), offsetof(
4010         deleg_perm_node_t, dpn_avl_node), deleg_perm_compare, UU_DEFAULT))
4011         == NULL)
4012         nomem();
4013 }
4015 static inline void fs_perm_fini(fs_perm_t *);
4016 static inline void who_perm_fini(who_perm_t *);
4018 static inline void
4019 fs_perm_set_fini(fs_perm_set_t *fspset)
4020 {

```

```

4021     fs_perm_node_t *node = uu_list_first(fspset->fsps_list);
4023     while (node != NULL) {
4024         fs_perm_node_t *next_node =
4025             uu_list_next(fspset->fsps_list, node);
4026         fs_perm_t *fsperm = &node->fspn_fsperm;
4027         fs_perm_fini(fsperm);
4028         uu_list_remove(fspset->fsps_list, node);
4029         free(node);
4030         node = next_node;
4031     }
4033     uu_avl_pool_destroy(fspset->fsps_named_set_avl_pool);
4034     uu_avl_pool_destroy(fspset->fsps_who_perm_avl_pool);
4035     uu_avl_pool_destroy(fspset->fsps_deleg_perm_avl_pool);
4036 }
4038 static inline void
4039 deleg_perm_init(deleg_perm_t *deleg_perm, zfs_deleg_who_type_t type,
4040     const char *name)
4041 {
4042     deleg_perm->dp_who_type = type;
4043     deleg_perm->dp_name = name;
4044 }
4046 static inline void
4047 who_perm_init(who_perm_t *who_perm, fs_perm_t *fsperm,
4048     zfs_deleg_who_type_t type, const char *name)
4049 {
4050     uu_avl_pool_t *pool;
4051     pool = fsperm->fsp_set->fsps_deleg_perm_avl_pool;
4053     bzero(who_perm, sizeof (who_perm_t));
4055     if ((who_perm->who_deleg_perm_avl = uu_avl_create(pool, NULL,
4056         UU_DEFAULT)) == NULL)
4057         nomem();
4059     who_perm->who_type = type;
4060     who_perm->who_name = name;
4061     who_perm->who_fsperm = fsperm;
4062 }
4064 static inline void
4065 who_perm_fini(who_perm_t *who_perm)
4066 {
4067     deleg_perm_node_t *node = uu_avl_first(who_perm->who_deleg_perm_avl);
4069     while (node != NULL) {
4070         deleg_perm_node_t *next_node =
4071             uu_avl_next(who_perm->who_deleg_perm_avl, node);
4073         uu_avl_remove(who_perm->who_deleg_perm_avl, node);
4074         free(node);
4075         node = next_node;
4076     }
4078     uu_avl_destroy(who_perm->who_deleg_perm_avl);
4079 }
4081 static inline void
4082 fs_perm_init(fs_perm_t *fsperm, fs_perm_set_t *fspset, const char *fname)
4083 {
4084     uu_avl_pool_t *nset_pool = fspset->fsps_named_set_avl_pool;
4085     uu_avl_pool_t *who_pool = fspset->fsps_who_perm_avl_pool;

```

```

4087     bzero(fsperm, sizeof (fs_perm_t));
4089     if ((fsperm->fsp_sc_avl = uu_avl_create(nset_pool, NULL, UU_DEFAULT))
4090         == NULL)
4091         nomem();
4093     if ((fsperm->fsp_uge_avl = uu_avl_create(who_pool, NULL, UU_DEFAULT))
4094         == NULL)
4095         nomem();
4097     fsperm->fsp_set = fspset;
4098     fsperm->fsp_name = fsname;
4099 }

4101 static inline void
4102 fs_perm_fini(fs_perm_t *fsperm)
4103 {
4104     who_perm_node_t *node = uu_avl_first(fsperm->fsp_sc_avl);
4105     while (node != NULL) {
4106         who_perm_node_t *next_node = uu_avl_next(fsperm->fsp_sc_avl,
4107             node);
4108         who_perm_t *who_perm = &node->who_perm;
4109         who_perm_fini(who_perm);
4110         uu_avl_remove(fsperm->fsp_sc_avl, node);
4111         free(node);
4112         node = next_node;
4113     }
4115     node = uu_avl_first(fsperm->fsp_uge_avl);
4116     while (node != NULL) {
4117         who_perm_node_t *next_node = uu_avl_next(fsperm->fsp_uge_avl,
4118             node);
4119         who_perm_t *who_perm = &node->who_perm;
4120         who_perm_fini(who_perm);
4121         uu_avl_remove(fsperm->fsp_uge_avl, node);
4122         free(node);
4123         node = next_node;
4124     }
4126     uu_avl_destroy(fsperm->fsp_sc_avl);
4127     uu_avl_destroy(fsperm->fsp_uge_avl);
4128 }

4130 static void inline
4131 set_deleg_perm_node(uu_avl_t *avl, deleg_perm_node_t *node,
4132     zfs_deleg_who_type_t who_type, const char *name, char locality)
4133 {
4134     uu_avl_index_t idx = 0;
4136     deleg_perm_node_t *found_node = NULL;
4137     deleg_perm_t *deleg_perm = &node->dpn_perm;
4139     deleg_perm_init(deleg_perm, who_type, name);
4141     if ((found_node = uu_avl_find(avl, node, NULL, &idx))
4142         == NULL)
4143         uu_avl_insert(avl, node, idx);
4144     else {
4145         node = found_node;
4146         deleg_perm = &node->dpn_perm;
4147     }
4150     switch (locality) {
4151     case ZFS_DELEG_LOCAL:
4152         deleg_perm->dp_local = B_TRUE;

```

```

4153         break;
4154     case ZFS_DELEG_DESCENDENT:
4155         deleg_perm->dp_descend = B_TRUE;
4156         break;
4157     case ZFS_DELEG_NA:
4158         break;
4159     default:
4160         assert(B_FALSE); /* invalid locality */
4161     }
4162 }

4164 static inline int
4165 parse_who_perm(who_perm_t *who_perm, nvlist_t *nvl, char locality)
4166 {
4167     nvpair_t *nvp = NULL;
4168     fs_perm_set_t *fspset = who_perm->who_fspset->fsp_set;
4169     uu_avl_t *avl = who_perm->who_deleg_perm_avl;
4170     zfs_deleg_who_type_t who_type = who_perm->who_type;
4172     while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
4173         const char *name = nvpair_name(nvp);
4174         data_type_t type = nvpair_type(nvp);
4175         uu_avl_pool_t *avl_pool = fspset->fsps_deleg_perm_avl_pool;
4176         deleg_perm_node_t *node =
4177             safe_malloc(sizeof (deleg_perm_node_t));
4179         assert(type == DATA_TYPE_BOOLEAN);
4181         uu_avl_node_init(node, &node->dpn_avl_node, avl_pool);
4182         set_deleg_perm_node(avl, node, who_type, name, locality);
4183     }
4185     return (0);
4186 }

4188 static inline int
4189 parse_fs_perm(fs_perm_t *fsperm, nvlist_t *nvl)
4190 {
4191     nvpair_t *nvp = NULL;
4192     fs_perm_set_t *fspset = fsperm->fsp_set;
4194     while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
4195         nvlist_t *nvl2 = NULL;
4196         const char *name = nvpair_name(nvp);
4197         uu_avl_t *avl = NULL;
4198         uu_avl_pool_t *avl_pool;
4199         zfs_deleg_who_type_t perm_type = name[0];
4200         char perm_locality = name[1];
4201         const char *perm_name = name + 3;
4202         boolean_t is_set = B_TRUE;
4203         who_perm_t *who_perm = NULL;
4205         assert('$' == name[2]);
4207         if (nvpair_value_nvlist(nvp, &nvl2) != 0)
4208             return (-1);
4210         switch (perm_type) {
4211         case ZFS_DELEG_CREATE:
4212         case ZFS_DELEG_CREATE_SETS:
4213         case ZFS_DELEG_NAMED_SET:
4214         case ZFS_DELEG_NAMED_SET_SETS:
4215             avl_pool = fspset->fsps_named_set_avl_pool;
4216             avl = fsperm->fsp_sc_avl;
4217             break;
4218         case ZFS_DELEG_USER:

```



```

4351         break;
4352     case ZFS_DELEG_NOTE_PROMOTE:
4353         str = gettext("Must also have the 'mount'\n\t\t\t\t\tand"
4354             " 'promote' ability in the origin file system");
4355         break;
4356     case ZFS_DELEG_NOTE_RECEIVE:
4357         str = gettext("Must also have the 'mount' and 'create'"
4358             " ability");
4359         break;
4360     case ZFS_DELEG_NOTE_RELEASE:
4361         str = gettext("Allows releasing a user hold which\n\t\t\t\t\t"
4362             "might destroy the snapshot");
4363         break;
4364     case ZFS_DELEG_NOTE_RENAME:
4365         str = gettext("Must also have the 'mount' and 'create'"
4366             "\n\t\t\t\t\tability in the new parent");
4367         break;
4368     case ZFS_DELEG_NOTE_ROLLBACK:
4369         str = gettext("");
4370         break;
4371     case ZFS_DELEG_NOTE_SEND:
4372         str = gettext("");
4373         break;
4374     case ZFS_DELEG_NOTE_SHARE:
4375         str = gettext("Allows sharing file systems over NFS or SMB"
4376             "\n\t\t\t\t\tprotocols");
4377         break;
4378     case ZFS_DELEG_NOTE_SNAPSHOT:
4379         str = gettext("");
4380         break;
4381     /*
4382     * case ZFS_DELEG_NOTE_VSCAN:
4383     *     str = gettext("");
4384     *     break;
4385     */
4386     /* OTHER */
4387     case ZFS_DELEG_NOTE_GROUPQUOTA:
4388         str = gettext("Allows accessing any groupquota@... property");
4389         break;
4390     case ZFS_DELEG_NOTE_GROUPUSED:
4391         str = gettext("Allows reading any groupused@... property");
4392         break;
4393     case ZFS_DELEG_NOTE_USERPROP:
4394         str = gettext("Allows changing any user property");
4395         break;
4396     case ZFS_DELEG_NOTE_USERQUOTA:
4397         str = gettext("Allows accessing any userquota@... property");
4398         break;
4399     case ZFS_DELEG_NOTE_USERUSED:
4400         str = gettext("Allows reading any userused@... property");
4401         break;
4402         /* other */
4403     default:
4404         str = "";
4405     }
4407     return (str);
4408 }

4410 struct allow_opts {
4411     boolean_t local;
4412     boolean_t descend;
4413     boolean_t user;
4414     boolean_t group;
4415     boolean_t everyone;
4416     boolean_t create;

```

```

4417     boolean_t set;
4418     boolean_t recursive; /* unallow only */
4419     boolean_t prt_usage;

4421     boolean_t prt_perms;
4422     char *who;
4423     char *perms;
4424     const char *dataset;
4425 };

4427 static inline int
4428 prop_cmp(const void *a, const void *b)
4429 {
4430     const char *str1 = *(const char **)a;
4431     const char *str2 = *(const char **)b;
4432     return (strcmp(str1, str2));
4433 }

4435 static void
4436 allow_usage(boolean_t un, boolean_t requested, const char *msg)
4437 {
4438     const char *opt_desc[] = {
4439         "-h", gettext("show this help message and exit"),
4440         "-l", gettext("set permission locally"),
4441         "-d", gettext("set permission for descents"),
4442         "-u", gettext("set permission for user"),
4443         "-g", gettext("set permission for group"),
4444         "-e", gettext("set permission for everyone"),
4445         "-c", gettext("set create time permission"),
4446         "-s", gettext("define permission set"),
4447         /* unallow only */
4448         "-r", gettext("remove permissions recursively"),
4449     };
4450     size_t unallow_size = sizeof (opt_desc) / sizeof (char *);
4451     size_t allow_size = unallow_size - 2;
4452     const char *props[ZFS_NUM_PROPS];
4453     int i;
4454     size_t count = 0;
4455     FILE *fp = requested ? stdout : stderr;
4456     zprop_desc_t *pdtbl = zfs_prop_get_table();
4457     const char *fmt = gettext("%-16s %-14s\t%s\n");

4459     (void) fprintf(fp, gettext("Usage: %s\n"), get_usage(un ? HELP_UNALLOW :
4460         HELP_ALLOW));
4461     (void) fprintf(fp, gettext("Options:\n"));
4462     for (int i = 0; i < (un ? unallow_size : allow_size); i++) {
4463         const char *opt = opt_desc[i++];
4464         const char *optdsc = opt_desc[i];
4465         (void) fprintf(fp, gettext(" %-10s %s\n"), opt, optdsc);
4466     }

4468     (void) fprintf(fp, gettext("\nThe following permissions are "
4469         "supported:\n\n"));
4470     (void) fprintf(fp, fmt, gettext("NAME"), gettext("TYPE"),
4471         gettext("NOTES"));
4472     for (i = 0; i < ZFS_NUM_DELEG_NOTES; i++) {
4473         const char *perm_name = zfs_deleg_perm_tbl[i].z_perm;
4474         zfs_deleg_note_t perm_note = zfs_deleg_perm_tbl[i].z_note;
4475         const char *perm_type = deleg_perm_type(perm_note);
4476         const char *perm_comment = deleg_perm_comment(perm_note);
4477         (void) fprintf(fp, fmt, perm_name, perm_type, perm_comment);
4478     }

4480     for (i = 0; i < ZFS_NUM_PROPS; i++) {
4481         zprop_desc_t *pd = &pdtbl[i];
4482         if (pd->pd_visible != B_TRUE)

```

```

4483         continue;
4485         if (pd->pd_attr == PROP_READONLY)
4486             continue;
4488         props[count++] = pd->pd_name;
4489     }
4490     props[count] = NULL;
4492     qsort(props, count, sizeof (char *), prop_cmp);
4494     for (i = 0; i < count; i++)
4495         (void) fprintf(fp, fmt, props[i], gettext("property"), "");
4497     if (msg != NULL)
4498         (void) fprintf(fp, gettext("\nzfs: error: %s"), msg);
4500     exit(requested ? 0 : 2);
4501 }
4503 static inline const char *
4504 munge_args(int argc, char **argv, boolean_t un, size_t expected_argc,
4505           char **permssp)
4506 {
4507     if (un && argc == expected_argc - 1)
4508         *permssp = NULL;
4509     else if (argc == expected_argc)
4510         *permssp = argv[argc - 2];
4511     else
4512         allow_usage(un, B_FALSE,
4513                   gettext("wrong number of parameters\n"));
4515     return (argv[argc - 1]);
4516 }
4518 static void
4519 parse_allow_args(int argc, char **argv, boolean_t un, struct allow_opts *opts)
4520 {
4521     int uge_sum = opts->user + opts->group + opts->everyone;
4522     int csuge_sum = opts->create + opts->set + uge_sum;
4523     int ldcsuge_sum = csuge_sum + opts->local + opts->descend;
4524     int all_sum = un ? ldcsuge_sum + opts->recursive : ldcsuge_sum;
4526     if (uge_sum > 1)
4527         allow_usage(un, B_FALSE,
4528                   gettext("-u, -g, and -e are mutually exclusive\n"));
4530     if (opts->prt_usage)
4531         if (argc == 0 && all_sum == 0)
4532             allow_usage(un, B_TRUE, NULL);
4533         else
4534             usage(B_FALSE);
4536     if (opts->set) {
4537         if (csuge_sum > 1)
4538             allow_usage(un, B_FALSE,
4539                       gettext("invalid options combined with -s\n"));
4541         opts->dataset = munge_args(argc, argv, un, 3, &opts->perms);
4542         if (argv[0][0] != '@')
4543             allow_usage(un, B_FALSE,
4544                       gettext("invalid set name: missing '@' prefix\n"));
4545         opts->who = argv[0];
4546     } else if (opts->create) {
4547         if (ldcsuge_sum > 1)
4548             allow_usage(un, B_FALSE,

```

```

4549         gettext("invalid options combined with -c\n"));
4550         opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4551     } else if (opts->everyone) {
4552         if (csuge_sum > 1)
4553             allow_usage(un, B_FALSE,
4554                       gettext("invalid options combined with -e\n"));
4555         opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4556     } else if (uge_sum == 0 && argc > 0 && strcmp(argv[0], "everyone")
4557              == 0) {
4558         opts->everyone = B_TRUE;
4559         argc--;
4560         argv++;
4561         opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4562     } else if (argc == 1 && !un) {
4563         opts->prt_perms = B_TRUE;
4564         opts->dataset = argv[argc-1];
4565     } else {
4566         opts->dataset = munge_args(argc, argv, un, 3, &opts->perms);
4567         opts->who = argv[0];
4568     }
4570     if (!opts->local && !opts->descend) {
4571         opts->local = B_TRUE;
4572         opts->descend = B_TRUE;
4573     }
4574 }
4576 static void
4577 store_allow_perm(zfs_deleg_who_type_t type, boolean_t local, boolean_t descend,
4578                const char *who, char *perms, nvlist_t *top_nvlist)
4579 {
4580     int i;
4581     char ld[2] = { '\0', '\0' };
4582     char who_buf[ZFS_MAXNAMELEN+32];
4583     char base_type;
4584     char set_type;
4585     nvlist_t *base_nvlist = NULL;
4586     nvlist_t *set_nvlist = NULL;
4587     nvlist_t *nvl;
4589     if (nvlist_alloc(&base_nvlist, NV_UNIQUE_NAME, 0) != 0)
4590         nomem();
4591     if (nvlist_alloc(&set_nvlist, NV_UNIQUE_NAME, 0) != 0)
4592         nomem();
4594     switch (type) {
4595     case ZFS_DELEG_NAMED_SET_SETS:
4596     case ZFS_DELEG_NAMED_SET:
4597         set_type = ZFS_DELEG_NAMED_SET_SETS;
4598         base_type = ZFS_DELEG_NAMED_SET;
4599         ld[0] = ZFS_DELEG_NA;
4600         break;
4601     case ZFS_DELEG_CREATE_SETS:
4602     case ZFS_DELEG_CREATE:
4603         set_type = ZFS_DELEG_CREATE_SETS;
4604         base_type = ZFS_DELEG_CREATE;
4605         ld[0] = ZFS_DELEG_NA;
4606         break;
4607     case ZFS_DELEG_USER_SETS:
4608     case ZFS_DELEG_USER:
4609         set_type = ZFS_DELEG_USER_SETS;
4610         base_type = ZFS_DELEG_USER;
4611         if (local)
4612             ld[0] = ZFS_DELEG_LOCAL;
4613         if (descend)
4614             ld[1] = ZFS_DELEG_DESCENDENT;

```

```

4615         break;
4616     case ZFS_DELEG_GROUP_SETS:
4617     case ZFS_DELEG_GROUP:
4618         set_type = ZFS_DELEG_GROUP_SETS;
4619         base_type = ZFS_DELEG_GROUP;
4620         if (local)
4621             ld[0] = ZFS_DELEG_LOCAL;
4622         if (descend)
4623             ld[1] = ZFS_DELEG_DESCENDENT;
4624         break;
4625     case ZFS_DELEG_EVERYONE_SETS:
4626     case ZFS_DELEG_EVERYONE:
4627         set_type = ZFS_DELEG_EVERYONE_SETS;
4628         base_type = ZFS_DELEG_EVERYONE;
4629         if (local)
4630             ld[0] = ZFS_DELEG_LOCAL;
4631         if (descend)
4632             ld[1] = ZFS_DELEG_DESCENDENT;
4633     }

4635     if (perms != NULL) {
4636         char *curr = perms;
4637         char *end = curr + strlen(perms);

4639         while (curr < end) {
4640             char *delim = strchr(curr, ',');
4641             if (delim == NULL)
4642                 delim = end;
4643             else
4644                 *delim = '\\0';

4646             if (curr[0] == '@')
4647                 nvl = set_nvl;
4648             else
4649                 nvl = base_nvl;

4651             (void) nvlist_add_boolean(nvl, curr);
4652             if (delim != end)
4653                 *delim = ',';
4654             curr = delim + 1;
4655         }

4657         for (i = 0; i < 2; i++) {
4658             char locality = ld[i];
4659             if (locality == 0)
4660                 continue;

4662             if (!nvlist_empty(base_nvl)) {
4663                 if (who != NULL)
4664                     (void) snprintf(who_buf,
4665                                     sizeof(who_buf), "%c%c%s",
4666                                     base_type, locality, who);
4667                 else
4668                     (void) snprintf(who_buf,
4669                                     sizeof(who_buf), "%c%c$",
4670                                     base_type, locality);

4672                 (void) nvlist_add_nvlist(top_nvl, who_buf,
4673                                           base_nvl);
4674             }

4677             if (!nvlist_empty(set_nvl)) {
4678                 if (who != NULL)
4679                     (void) snprintf(who_buf,
4680                                     sizeof(who_buf), "%c%c%s",

```

```

4681         set_type, locality, who);
4682     else
4683         (void) snprintf(who_buf,
4684                         sizeof(who_buf), "%c%c$",
4685                         set_type, locality);

4687         (void) nvlist_add_nvlist(top_nvl, who_buf,
4688                                   set_nvl);
4689     }
4690 }
4691 } else {
4692     for (i = 0; i < 2; i++) {
4693         char locality = ld[i];
4694         if (locality == 0)
4695             continue;

4697         if (who != NULL)
4698             (void) snprintf(who_buf, sizeof(who_buf),
4699                             "%c%c%s", base_type, locality, who);
4700         else
4701             (void) snprintf(who_buf, sizeof(who_buf),
4702                             "%c%c$", base_type, locality);
4703         (void) nvlist_add_boolean(top_nvl, who_buf);

4705         if (who != NULL)
4706             (void) snprintf(who_buf, sizeof(who_buf),
4707                             "%c%c%s", set_type, locality, who);
4708         else
4709             (void) snprintf(who_buf, sizeof(who_buf),
4710                             "%c%c$", set_type, locality);
4711         (void) nvlist_add_boolean(top_nvl, who_buf);
4712     }
4713 }
4714 }

4716 static int
4717 construct_fsacl_list(boolean_t un, struct allow_opts *opts, nvlist_t **nvlp)
4718 {
4719     if (nvlist_alloc(nvlp, NV_UNIQUE_NAME, 0) != 0)
4720         nomem();

4722     if (opts->set) {
4723         store_allow_perm(ZFS_DELEG_NAMED_SET, opts->local,
4724                         opts->descend, opts->who, opts->perms, *nvlp);
4725     } else if (opts->create) {
4726         store_allow_perm(ZFS_DELEG_CREATE, opts->local,
4727                         opts->descend, NULL, opts->perms, *nvlp);
4728     } else if (opts->everyone) {
4729         store_allow_perm(ZFS_DELEG_EVERYONE, opts->local,
4730                         opts->descend, NULL, opts->perms, *nvlp);
4731     } else {
4732         char *curr = opts->who;
4733         char *end = curr + strlen(curr);

4735         while (curr < end) {
4736             const char *who;
4737             zfs_deleg_who_type_t who_type;
4738             char *endch;
4739             char *delim = strchr(curr, ',');
4740             char errbuf[256];
4741             char id[64];
4742             struct passwd *p = NULL;
4743             struct group *g = NULL;

4745             uid_t rid;
4746             if (delim == NULL)

```

```

4747     delim = end;
4748     else
4749         *delim = '\0';

4751     rid = (uid_t)strtoul(curr, &endch, 0);
4752     if (opts->user) {
4753         who_type = ZFS_DELEG_USER;
4754         if (*endch != '\0')
4755             p = getpwnam(curr);
4756         else
4757             p = getpwuid(rid);

4759         if (p != NULL)
4760             rid = p->pw_uid;
4761         else {
4762             (void) snprintf(errbuf, 256, gettext(
4763                 "invalid user %s"), curr);
4764             allow_usage(un, B_TRUE, errbuf);
4765         }
4766     } else if (opts->group) {
4767         who_type = ZFS_DELEG_GROUP;
4768         if (*endch != '\0')
4769             g = getgrnam(curr);
4770         else
4771             g = getgrgid(rid);

4773         if (g != NULL)
4774             rid = g->gr_gid;
4775         else {
4776             (void) snprintf(errbuf, 256, gettext(
4777                 "invalid group %s"), curr);
4778             allow_usage(un, B_TRUE, errbuf);
4779         }
4780     } else {
4781         if (*endch != '\0') {
4782             p = getpwnam(curr);
4783         } else {
4784             p = getpwuid(rid);
4785         }

4787         if (p == NULL)
4788             if (*endch != '\0') {
4789                 g = getgrnam(curr);
4790             } else {
4791                 g = getgrgid(rid);
4792             }

4794         if (p != NULL) {
4795             who_type = ZFS_DELEG_USER;
4796             rid = p->pw_uid;
4797         } else if (g != NULL) {
4798             who_type = ZFS_DELEG_GROUP;
4799             rid = g->gr_gid;
4800         } else {
4801             (void) snprintf(errbuf, 256, gettext(
4802                 "invalid user/group %s"), curr);
4803             allow_usage(un, B_TRUE, errbuf);
4804         }
4805     }

4807     (void) sprintf(id, "%u", rid);
4808     who = id;

4810     store_allow_perm(who_type, opts->local,
4811         opts->descend, who, opts->perms, *nvp);
4812     curr = delim + 1;

```

```

4813     }
4814 }

4816     return (0);
4817 }

4819 static void
4820 print_set_creat_perms(uu_avl_t *who_avl)
4821 {
4822     const char *sc_title[] = {
4823         gettext("Permission sets:\n"),
4824         gettext("Create time permissions:\n"),
4825         NULL
4826     };
4827     const char **title_ptr = sc_title;
4828     who_perm_node_t *who_node = NULL;
4829     int prev_weight = -1;

4831     for (who_node = uu_avl_first(who_avl); who_node != NULL;
4832         who_node = uu_avl_next(who_avl, who_node)) {
4833         uu_avl_t *avl = who_node->who_perm.who_deleg_perm_avl;
4834         zfs_deleg_who_type_t who_type = who_node->who_perm.who_type;
4835         const char *who_name = who_node->who_perm.who_name;
4836         int weight = who_type2weight(who_type);
4837         boolean_t first = B_TRUE;
4838         deleg_perm_node_t *deleg_node;

4840         if (prev_weight != weight) {
4841             (void) printf(*title_ptr++);
4842             prev_weight = weight;
4843         }

4845         if (who_name == NULL || strlen(who_name, 1) == 0)
4846             (void) printf("\t");
4847         else
4848             (void) printf("\t%s ", who_name);

4850         for (deleg_node = uu_avl_first(avl); deleg_node != NULL;
4851             deleg_node = uu_avl_next(avl, deleg_node)) {
4852             if (first) {
4853                 (void) printf("%s",
4854                     deleg_node->dpn_perm.dp_name);
4855                 first = B_FALSE;
4856             } else
4857                 (void) printf(",%s",
4858                     deleg_node->dpn_perm.dp_name);
4859         }

4861         (void) printf("\n");
4862     }
4863 }

4865 static void inline
4866 print_uge_deleg_perms(uu_avl_t *who_avl, boolean_t local, boolean_t descend,
4867     const char *title)
4868 {
4869     who_perm_node_t *who_node = NULL;
4870     boolean_t prt_title = B_TRUE;
4871     uu_avl_walk_t *walk;

4873     if ((walk = uu_avl_walk_start(who_avl, UU_WALK_ROBUST)) == NULL)
4874         nomem();

4876     while ((who_node = uu_avl_walk_next(walk)) != NULL) {
4877         const char *who_name = who_node->who_perm.who_name;
4878         const char *nice_who_name = who_node->who_perm.who_ug_name;

```

```

4879     uu_avl_t *avl = who_node->who_perm.who_deleg_perm_avl;
4880     zfs_deleg_who_type_t who_type = who_node->who_perm.who_type;
4881     char delim = ' ';
4882     deleg_perm_node_t *deleg_node;
4883     boolean_t prt_who = B_TRUE;

4885     for (deleg_node = uu_avl_first(avl);
4886          deleg_node != NULL;
4887          deleg_node = uu_avl_next(avl, deleg_node)) {
4888         if (local != deleg_node->dpn_perm.dp_local ||
4889             descend != deleg_node->dpn_perm.dp_descend)
4890             continue;

4892         if (prt_who) {
4893             const char *who = NULL;
4894             if (prt_title) {
4895                 prt_title = B_FALSE;
4896                 (void) printf(title);
4897             }

4899             switch (who_type) {
4900             case ZFS_DELEG_USER_SETS:
4901             case ZFS_DELEG_USER:
4902                 who = gettext("user");
4903                 if (nice_who_name)
4904                     who_name = nice_who_name;
4905                 break;
4906             case ZFS_DELEG_GROUP_SETS:
4907             case ZFS_DELEG_GROUP:
4908                 who = gettext("group");
4909                 if (nice_who_name)
4910                     who_name = nice_who_name;
4911                 break;
4912             case ZFS_DELEG_EVERYONE_SETS:
4913             case ZFS_DELEG_EVERYONE:
4914                 who = gettext("everyone");
4915                 who_name = NULL;
4916             }

4918             prt_who = B_FALSE;
4919             if (who_name == NULL)
4920                 (void) printf("\t%s", who);
4921             else
4922                 (void) printf("\t%s %s", who, who_name);
4923         }

4925         (void) printf("%c%s", delim,
4926                      deleg_node->dpn_perm.dp_name);
4927         delim = ' ';
4928     }

4930     if (!prt_who)
4931         (void) printf("\n");
4932 }

4934 uu_avl_walk_end(walk);
4935 }

4937 static void
4938 print_fs_perms(fs_perm_set_t *fspset)
4939 {
4940     fs_perm_node_t *node = NULL;
4941     char buf[ZFS_MAXNAMELEN+32];
4942     const char *dsname = buf;

4944     for (node = uu_list_first(fspset->fsps_list); node != NULL;

```

```

4945     node = uu_list_next(fspset->fsps_list, node)) {
4946         uu_avl_t *sc_avl = node->fspn_fsperm.fsp_sc_avl;
4947         uu_avl_t *uge_avl = node->fspn_fsperm.fsp_uge_avl;
4948         int left = 0;

4950         (void) snprintf(buf, ZFS_MAXNAMELEN+32,
4951                        gettext("---- Permissions on %s "),
4952                        node->fspn_fsperm.fsp_name);
4953         (void) printf(dsname);
4954         left = 70 - strlen(buf);
4955         while (left-- > 0)
4956             (void) printf("-");
4957         (void) printf("\n");

4959         print_set_creat_perms(sc_avl);
4960         print_uge_deleg_perms(uge_avl, B_TRUE, B_FALSE,
4961                              gettext("Local permissions:\n"));
4962         print_uge_deleg_perms(uge_avl, B_FALSE, B_TRUE,
4963                              gettext("Descendent permissions:\n"));
4964         print_uge_deleg_perms(uge_avl, B_TRUE, B_TRUE,
4965                              gettext("Local+Descendent permissions:\n"));
4966     }
4967 }

4969 static fs_perm_set_t fs_perm_set = { NULL, NULL, NULL, NULL };

4971 struct deleg_perms {
4972     boolean_t un;
4973     nvlist_t *nvl;
4974 };

4976 static int
4977 set_deleg_perms(zfs_handle_t *zhp, void *data)
4978 {
4979     struct deleg_perms *perms = (struct deleg_perms *)data;
4980     zfs_type_t zfs_type = zfs_get_type(zhp);

4982     if (zfs_type != ZFS_TYPE_FILESYSTEM && zfs_type != ZFS_TYPE_VOLUME)
4983         return (0);

4985     return (zfs_set_fsacl(zhp, perms->un, perms->nvl));
4986 }

4988 static int
4989 zfs_do_allow_unallow_impl(int argc, char **argv, boolean_t un)
4990 {
4991     zfs_handle_t *zhp;
4992     nvlist_t *perm_nvl = NULL;
4993     nvlist_t *update_perm_nvl = NULL;
4994     int error = 1;
4995     int c;
4996     struct allow_opts opts = { 0 };

4998     const char *optstr = un ? "ldugecsh" : "ldugecsh";

5000     /* check opts */
5001     while ((c = getopt(argc, argv, optstr)) != -1) {
5002         switch (c) {
5003             case 'l':
5004                 opts.local = B_TRUE;
5005                 break;
5006             case 'd':
5007                 opts.descend = B_TRUE;
5008                 break;
5009             case 'u':
5010                 opts.user = B_TRUE;

```



```

5011         break;
5012     case 'g':
5013         opts.group = B_TRUE;
5014         break;
5015     case 'e':
5016         opts.everyone = B_TRUE;
5017         break;
5018     case 's':
5019         opts.set = B_TRUE;
5020         break;
5021     case 'c':
5022         opts.create = B_TRUE;
5023         break;
5024     case 'r':
5025         opts.recursive = B_TRUE;
5026         break;
5027     case ':':
5028         (void) fprintf(stderr, gettext("missing argument for "
5029             "%c' option\n"), optopt);
5030         usage(B_FALSE);
5031         break;
5032     case 'h':
5033         opts.prt_usage = B_TRUE;
5034         break;
5035     case '?':
5036         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5037             optopt);
5038         usage(B_FALSE);
5039     }
5040 }

5042 argc -= optind;
5043 argv += optind;

5045 /* check arguments */
5046 parse_allow_args(argc, argv, un, &opts);

5048 /* try to open the dataset */
5049 if ((zhp = zfs_open(g_zfs, opts.dataset, ZFS_TYPE_FILESYSTEM |
5050     ZFS_TYPE_VOLUME)) == NULL) {
5051     (void) fprintf(stderr, "Failed to open dataset: %s\n",
5052         opts.dataset);
5053     return (-1);
5054 }

5056 if (zfs_get_fsacl(zhp, &perm_nvlist) != 0)
5057     goto cleanup2;

5059 fs_perm_set_init(&fs_perm_set);
5060 if (parse_fs_perm_set(&fs_perm_set, perm_nvlist) != 0) {
5061     (void) fprintf(stderr, "Failed to parse fsacl permissions\n");
5062     goto cleanup1;
5063 }

5065 if (opts.prt_perms)
5066     print_fs_perms(&fs_perm_set);
5067 else {
5068     (void) construct_fsacl_list(un, &opts, &update_perm_nvlist);
5069     if (zfs_set_fsacl(zhp, un, update_perm_nvlist) != 0)
5070         goto cleanup0;

5072     if (un && opts.recursive) {
5073         struct deleg_perms data = { un, update_perm_nvlist };
5074         if (zfs_iter_filesystems(zhp, set_deleg_perms,
5075             &data) != 0)
5076             goto cleanup0;

```

```

5077     }
5078 }

5080     error = 0;

5082 cleanup0:
5083     nvlist_free(perm_nvlist);
5084     if (update_perm_nvlist != NULL)
5085         nvlist_free(update_perm_nvlist);
5086 cleanup1:
5087     fs_perm_set_fini(&fs_perm_set);
5088 cleanup2:
5089     zfs_close(zhp);

5091     return (error);
5092 }

5094 static int
5095 zfs_do_allow(int argc, char **argv)
5096 {
5097     return (zfs_do_allow_unallow_impl(argc, argv, B_FALSE));
5098 }

5100 static int
5101 zfs_do_unallow(int argc, char **argv)
5102 {
5103     return (zfs_do_allow_unallow_impl(argc, argv, B_TRUE));
5104 }

5106 static int
5107 zfs_do_hold_rele_impl(int argc, char **argv, boolean_t holding)
5108 {
5109     int errors = 0;
5110     int i;
5111     const char *tag;
5112     boolean_t recursive = B_FALSE;
5113     const char *opts = holding ? "rt" : "r";
5114     int c;

5116     /* check options */
5117     while ((c = getopt(argc, argv, opts)) != -1) {
5118         switch (c) {
5119             case 'r':
5120                 recursive = B_TRUE;
5121                 break;
5122             case '?':
5123                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5124                     optopt);
5125                 usage(B_FALSE);
5126             }
5127     }

5129     argc -= optind;
5130     argv += optind;

5132     /* check number of arguments */
5133     if (argc < 2)
5134         usage(B_FALSE);

5136     tag = argv[0];
5137     --argv;
5138     ++argv;

5140     if (holding && tag[0] == '.') {
5141         /* tags starting with '.' are reserved for libzfs */
5142         (void) fprintf(stderr, gettext("tag may not start with '.'\n"));

```

```
5143         usage(B_FALSE);
5144     }
5145
5146     for (i = 0; i < argc; ++i) {
5147         zfs_handle_t *zhp;
5148         char parent[ZFS_MAXNAMELEN];
5149         const char *delim;
5150         char *path = argv[i];
5151
5152         delim = strchr(path, '@');
5153         if (delim == NULL) {
5154             (void) fprintf(stderr,
5155                 gettext("%s' is not a snapshot\n"), path);
5156             ++errors;
5157             continue;
5158         }
5159         (void) strncpy(parent, path, delim - path);
5160         parent[delim - path] = '\0';
5161
5162         zhp = zfs_open(g_zfs, parent,
5163             ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
5164         if (zhp == NULL) {
5165             ++errors;
5166             continue;
5167         }
5168         if (holding) {
5169             if (zfs_hold(zhp, delim+1, tag, recursive, -1) != 0)
5170                 if (zfs_hold(zhp, delim+1, tag, recursive,
5171                     B_FALSE, -1) != 0)
5172                     ++errors;
5173             } else {
5174                 if (zfs_release(zhp, delim+1, tag, recursive) != 0)
5175                     ++errors;
5176             }
5177         zfs_close(zhp);
5178     }
5179     return (errors != 0);
5180 }
5181 _____unchanged_portion_omitted_____
```

```

*****
26978 Wed May 29 20:27:07 2013
new/usr/src/lib/libzfs/common/libzfs.h
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
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 2011 Nexenta Systems, Inc. All rights reserved.
25 * Copyright (c) 2012 by Delphix. All rights reserved.
26 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
27 * Copyright (c) 2013 Steven Hartland. All rights reserved.
28 #endif /* ! codereview */
29 */

31 #ifndef _LIBZFS_H
32 #define _LIBZFS_H

34 #include <assert.h>
35 #include <libnvpair.h>
36 #include <sys/mnttab.h>
37 #include <sys/param.h>
38 #include <sys/types.h>
39 #include <sys/varargs.h>
40 #include <sys/fs/zfs.h>
41 #include <sys/avl.h>
42 #include <ucred.h>

44 #ifdef __cplusplus
45 extern "C" {
46 #endif

48 /*
49  * Miscellaneous ZFS constants
50  */
51 #define ZFS_MAXNAMELEN      MAXNAMELEN
52 #define ZPOOL_MAXNAMELEN   MAXNAMELEN
53 #define ZFS_MAXPROPLEN     MAXPATHLEN
54 #define ZPOOL_MAXPROPLEN   MAXPATHLEN

56 /*
57  * libzfs errors
58  */
59 typedef enum zfs_error {
60     EZFS_SUCCESS = 0,      /* no error -- success */

```

```

61     EZFS_NOMEM = 2000,    /* out of memory */
62     EZFS_BADPROP,        /* invalid property value */
63     EZFS_PROPREADONLY,   /* cannot set readonly property */
64     EZFS_PROPTYPE,       /* property does not apply to dataset type */
65     EZFS_PROPNONINHERIT, /* property is not inheritable */
66     EZFS_PROPSPACE,     /* bad quota or reservation */
67     EZFS_BADTYPE,       /* dataset is not of appropriate type */
68     EZFS_BUSY,          /* pool or dataset is busy */
69     EZFS_EXISTS,        /* pool or dataset already exists */
70     EZFS_NOENT,         /* no such pool or dataset */
71     EZFS_BADSTREAM,     /* bad backup stream */
72     EZFS_DSREADONLY,    /* dataset is readonly */
73     EZFS_VOLTOOBIG,     /* volume is too large for 32-bit system */
74     EZFS_INVALIDNAME,   /* invalid dataset name */
75     EZFS_BADRESTORE,    /* unable to restore to destination */
76     EZFS_BADBACKUP,     /* backup failed */
77     EZFS_BADTARGET,     /* bad attach/detach/replace target */
78     EZFS_NODEVICE,     /* no such device in pool */
79     EZFS_BADDEV,        /* invalid device to add */
80     EZFS_NOREPLICAS,    /* no valid replicas */
81     EZFS_RESILVERING,   /* currently resilvering */
82     EZFS_BADVERSION,    /* unsupported version */
83     EZFS_POOLUNAVAIL,   /* pool is currently unavailable */
84     EZFS_DEVOVERFLOW,   /* too many devices in one vdev */
85     EZFS_BADPATH,       /* must be an absolute path */
86     EZFS_CROSSTARGET,   /* rename or clone across pool or dataset */
87     EZFS_ZONED,         /* used improperly in local zone */
88     EZFS_MOUNTFAILED,   /* failed to mount dataset */
89     EZFS_UMOUNTFAILED,  /* failed to unmount dataset */
90     EZFS_UNSHARENFSFAILED, /* unshare(1M) failed */
91     EZFS_SHARENFSFAILED, /* share(1M) failed */
92     EZFS_PERM,          /* permission denied */
93     EZFS_NOSPC,         /* out of space */
94     EZFS_FAULT,        /* bad address */
95     EZFS_IO,           /* I/O error */
96     EZFS_INTR,         /* signal received */
97     EZFS_ISSPARE,      /* device is a hot spare */
98     EZFS_INVALCONFIG, /* invalid vdev configuration */
99     EZFS_RECURSIVE,   /* recursive dependency */
100    EZFS_NOHISTORY,    /* no history object */
101    EZFS_POOLPROPS,    /* couldn't retrieve pool props */
102    EZFS_POOL_NOTSUP,  /* ops not supported for this type of pool */
103    EZFS_POOL_INVALARG, /* invalid argument for this pool operation */
104    EZFS_NAMETOOLONG,  /* dataset name is too long */
105    EZFS_OPENFAILED,   /* open of device failed */
106    EZFS_NOCAP,        /* couldn't get capacity */
107    EZFS_LABELFAILED,  /* write of label failed */
108    EZFS_BADWHO,       /* invalid permission who */
109    EZFS_BADPERM,      /* invalid permission */
110    EZFS_BADPERMSET,   /* invalid permission set name */
111    EZFS_NODELEGATION, /* delegated administration is disabled */
112    EZFS_UNSHARESMBFAILED, /* failed to unshare over smb */
113    EZFS_SHARESMBFAILED, /* failed to share over smb */
114    EZFS_BADCACHE,     /* bad cache file */
115    EZFS_ISL2CACHE,    /* device is for the level 2 ARC */
116    EZFS_VDEVNOTSUP,   /* unsupported vdev type */
117    EZFS_NOTSUP,       /* ops not supported on this dataset */
118    EZFS_ACTIVE_SPARE, /* pool has active shared spare devices */
119    EZFS_UNPLAYED_LOGS, /* log device has unplayed logs */
120    EZFS_REFTAG_RELE,  /* snapshot release: tag not found */
121    EZFS_REFTAG_HOLD,  /* snapshot hold: tag already exists */
122    EZFS_TAGTOOLONG,   /* snapshot hold/rele: tag too long */
123    EZFS_PIPEFAILED,   /* pipe create failed */
124    EZFS_THREADCREATEFAILED, /* thread create failed */
125    EZFS_POSTSPLIT_ONLINE, /* onlining a disk after splitting it */
126    EZFS_SCRUBBING,    /* currently scrubbing */

```

```

127     EZFS_NO_SCRUB,          /* no active scrub */
128     EZFS_DIFF,             /* general failure of zfs diff */
129     EZFS_DIFFDATA,        /* bad zfs diff data */
130     EZFS_POOLREADONLY,    /* pool is in read-only mode */
131     EZFS_UNKNOWN
132 } zfs_error_t;

134 /*
135  * The following data structures are all part
136  * of the zfs_allow_t data structure which is
137  * used for printing 'allow' permissions.
138  * It is a linked list of zfs_allow_t's which
139  * then contain avl tree's for user/group/sets/...
140  * and each one of the entries in those trees have
141  * avl tree's for the permissions they belong to and
142  * whether they are local,descendent or local+descendent
143  * permissions. The AVL trees are used primarily for
144  * sorting purposes, but also so that we can quickly find
145  * a given user and or permission.
146  */
147 typedef struct zfs_perm_node {
148     avl_node_t z_node;
149     char z_pname[MAXPATHLEN];
150 } zfs_perm_node_t;

152 typedef struct zfs_allow_node {
153     avl_node_t z_node;
154     char z_key[MAXPATHLEN];          /* name, such as joe */
155     avl_tree_t z_localedescend;     /* local+descendent perms */
156     avl_tree_t z_local;             /* local permissions */
157     avl_tree_t z_descend;           /* descendent permissions */
158 } zfs_allow_node_t;

160 typedef struct zfs_allow {
161     struct zfs_allow *z_next;
162     char z_setpoint[MAXPATHLEN];
163     avl_tree_t z_sets;
164     avl_tree_t z_crperms;
165     avl_tree_t z_user;
166     avl_tree_t z_group;
167     avl_tree_t z_everyone;
168 } zfs_allow_t;

170 /*
171  * Basic handle types
172  */
173 typedef struct zfs_handle zfs_handle_t;
174 typedef struct zpool_handle zpool_handle_t;
175 typedef struct libzfs_handle libzfs_handle_t;

177 /*
178  * Library initialization
179  */
180 extern libzfs_handle_t *libzfs_init(void);
181 extern void libzfs_fini(libzfs_handle_t *);

183 extern libzfs_handle_t *zpool_get_handle(zpool_handle_t *);
184 extern libzfs_handle_t *zfs_get_handle(zfs_handle_t *);

186 extern void libzfs_print_on_error(libzfs_handle_t *, boolean_t);

188 extern void zfs_save_arguments(int argc, char **, char *, int);
189 extern int zpool_log_history(libzfs_handle_t *, const char *);

191 extern int libzfs_errno(libzfs_handle_t *);
192 extern const char *libzfs_error_action(libzfs_handle_t *);

```

```

193 extern const char *libzfs_error_description(libzfs_handle_t *);
194 extern void libzfs_mnttab_init(libzfs_handle_t *);
195 extern void libzfs_mnttab_fini(libzfs_handle_t *);
196 extern void libzfs_mnttab_cache(libzfs_handle_t *, boolean_t);
197 extern int libzfs_mnttab_find(libzfs_handle_t *, const char *,
198     struct mnttab *);
199 extern void libzfs_mnttab_add(libzfs_handle_t *, const char *,
200     const char *, const char *);
201 extern void libzfs_mnttab_remove(libzfs_handle_t *, const char *);

203 /*
204  * Basic handle functions
205  */
206 extern zpool_handle_t *zpool_open(libzfs_handle_t *, const char *);
207 extern zpool_handle_t *zpool_open_canfail(libzfs_handle_t *, const char *);
208 extern void zpool_close(zpool_handle_t *);
209 extern const char *zpool_get_name(zpool_handle_t *);
210 extern int zpool_get_state(zpool_handle_t *);
211 extern char *zpool_state_to_name(vdev_state_t, vdev_aux_t);
212 extern void zpool_free_handles(libzfs_handle_t *);

214 /*
215  * Iterate over all active pools in the system.
216  */
217 typedef int (*zpool_iter_f)(zpool_handle_t *, void *);
218 extern int zpool_iter(libzfs_handle_t *, zpool_iter_f, void *);

220 /*
221  * Functions to create and destroy pools
222  */
223 extern int zpool_create(libzfs_handle_t *, const char *, nvlist_t *,
224     nvlist_t *, nvlist_t *);
225 extern int zpool_destroy(zpool_handle_t *, const char *);
226 extern int zpool_add(zpool_handle_t *, nvlist_t *);

228 typedef struct splitflags {
229     /* do not split, but return the config that would be split off */
230     int dryrun : 1;

232     /* after splitting, import the pool */
233     int import : 1;
234 } splitflags_t;

236 /*
237  * Functions to manipulate pool and vdev state
238  */
239 extern int zpool_scan(zpool_handle_t *, pool_scan_func_t);
240 extern int zpool_clear(zpool_handle_t *, const char *, nvlist_t *);
241 extern int zpool_reguid(zpool_handle_t *);
242 extern int zpool_reopen(zpool_handle_t *);

244 extern int zpool_vdev_online(zpool_handle_t *, const char *, int,
245     vdev_state_t *);
246 extern int zpool_vdev_offline(zpool_handle_t *, const char *, boolean_t);
247 extern int zpool_vdev_attach(zpool_handle_t *, const char *,
248     const char *, nvlist_t *, int);
249 extern int zpool_vdev_detach(zpool_handle_t *, const char *);
250 extern int zpool_vdev_remove(zpool_handle_t *, const char *);
251 extern int zpool_vdev_split(zpool_handle_t *, char *, nvlist_t **, nvlist_t *,
252     splitflags_t);

254 extern int zpool_vdev_fault(zpool_handle_t *, uint64_t, vdev_aux_t);
255 extern int zpool_vdev_degrade(zpool_handle_t *, uint64_t, vdev_aux_t);
256 extern int zpool_vdev_clear(zpool_handle_t *, uint64_t);

258 extern nvlist_t *zpool_find_vdev(zpool_handle_t *, const char *, boolean_t *,

```

```

259     boolean_t *, boolean_t *);
260 extern nvlist_t *zpool_find_vdev_by_physpath(zpool_handle_t *, const char *,
261     boolean_t *, boolean_t *, boolean_t *);
262 extern int zpool_label_disk(libzfs_handle_t *, zpool_handle_t *, char *);

264 /*
265  * Functions to manage pool properties
266  */
267 extern int zpool_set_prop(zpool_handle_t *, const char *, const char *);
268 extern int zpool_get_prop(zpool_handle_t *, zpool_prop_t, char *,
269     size_t proplen, zprop_source_t *);
270 extern uint64_t zpool_get_prop_int(zpool_handle_t *, zpool_prop_t,
271     zprop_source_t *);

273 extern const char *zpool_prop_to_name(zpool_prop_t);
274 extern const char *zpool_prop_values(zpool_prop_t);

276 /*
277  * Pool health statistics.
278  */
279 typedef enum {
280     /*
281     * The following correspond to faults as defined in the (fault.fs.zfs.*)
282     * event namespace. Each is associated with a corresponding message ID.
283     */
284     ZPOOL_STATUS_CORRUPT_CACHE, /* corrupt /kernel/drv/zpool.cache */
285     ZPOOL_STATUS_MISSING_DEV_R, /* missing device with replicas */
286     ZPOOL_STATUS_MISSING_DEV_NR, /* missing device with no replicas */
287     ZPOOL_STATUS_CORRUPT_LABEL_R, /* bad device label with replicas */
288     ZPOOL_STATUS_CORRUPT_LABEL_NR, /* bad device label with no replicas */
289     ZPOOL_STATUS_BAD_GUID_SUM, /* sum of device guids didn't match */
290     ZPOOL_STATUS_CORRUPT_POOL, /* pool metadata is corrupted */
291     ZPOOL_STATUS_CORRUPT_DATA, /* data errors in user (meta)data */
292     ZPOOL_STATUS_FAILING_DEV, /* device experiencing errors */
293     ZPOOL_STATUS_VERSION_NEWER, /* newer on-disk version */
294     ZPOOL_STATUS_HOSTID_MISMATCH, /* last accessed by another system */
295     ZPOOL_STATUS_IO_FAILURE_WAIT, /* failed I/O, failmode 'wait' */
296     ZPOOL_STATUS_IO_FAILURE_CONTINUE, /* failed I/O, failmode 'continue' */
297     ZPOOL_STATUS_BAD_LOG, /* cannot read log chain(s) */

299     /*
300     * If the pool has unsupported features but can still be opened in
301     * read-only mode, its status is ZPOOL_STATUS_UNSUP_FEAT_WRITE. If the
302     * pool has unsupported features but cannot be opened at all, its
303     * status is ZPOOL_STATUS_UNSUP_FEAT_READ.
304     */
305     ZPOOL_STATUS_UNSUP_FEAT_READ, /* unsupported features for read */
306     ZPOOL_STATUS_UNSUP_FEAT_WRITE, /* unsupported features for write */

308     /*
309     * These faults have no corresponding message ID. At the time we are
310     * checking the status, the original reason for the FMA fault (I/O or
311     * checksum errors) has been lost.
312     */
313     ZPOOL_STATUS_FAULTED_DEV_R, /* faulted device with replicas */
314     ZPOOL_STATUS_FAULTED_DEV_NR, /* faulted device with no replicas */

316     /*
317     * The following are not faults per se, but still an error possibly
318     * requiring administrative attention. There is no corresponding
319     * message ID.
320     */
321     ZPOOL_STATUS_VERSION_OLDER, /* older legacy on-disk version */
322     ZPOOL_STATUS_FEAT_DISABLED, /* supported features are disabled */
323     ZPOOL_STATUS_RESILVERING, /* device being resilvered */
324     ZPOOL_STATUS_OFFLINE_DEV, /* device online */

```

```

325     ZPOOL_STATUS_REMOVED_DEV, /* removed device */

327     /*
328     * Finally, the following indicates a healthy pool.
329     */
330     ZPOOL_STATUS_OK
331 } zpool_status_t;

333 extern zpool_status_t zpool_get_status(zpool_handle_t *, char **);
334 extern zpool_status_t zpool_import_status(nvlist_t *, char **);
335 extern void zpool_dump_ddt(const ddt_stat_t *dds, const ddt_histogram_t *ddh);

337 /*
338  * Statistics and configuration functions.
339  */
340 extern nvlist_t *zpool_get_config(zpool_handle_t *, nvlist_t **);
341 extern nvlist_t *zpool_get_features(zpool_handle_t *);
342 extern int zpool_refresh_stats(zpool_handle_t *, boolean_t *);
343 extern int zpool_get_errlog(zpool_handle_t *, nvlist_t **);

345 /*
346  * Import and export functions
347  */
348 extern int zpool_export(zpool_handle_t *, boolean_t, const char *);
349 extern int zpool_export_force(zpool_handle_t *, const char *);
350 extern int zpool_import(libzfs_handle_t *, nvlist_t *, const char *,
351     char *altroot);
352 extern int zpool_import_props(libzfs_handle_t *, nvlist_t *, const char *,
353     nvlist_t *, int);
354 extern void zpool_print_unsup_feat(nvlist_t *config);

356 /*
357  * Search for pools to import
358  */

360 typedef struct importargs {
361     char **path; /* a list of paths to search */
362     int paths; /* number of paths to search */
363     char *poolname; /* name of a pool to find */
364     uint64_t guid; /* guid of a pool to find */
365     char *cachefile; /* cachefile to use for import */
366     int can_be_active : 1; /* can the pool be active? */
367     int unique : 1; /* does 'poolname' already exist? */
368     int exists : 1; /* set on return if pool already exists */
369 } importargs_t;

371 extern nvlist_t *zpool_search_import(libzfs_handle_t *, importargs_t *);

373 /* legacy pool search routines */
374 extern nvlist_t *zpool_find_import(libzfs_handle_t *, int, char **);
375 extern nvlist_t *zpool_find_import_cached(libzfs_handle_t *, const char *,
376     char *, uint64_t);

378 /*
379  * Miscellaneous pool functions
380  */
381 struct zfs_cmd;

383 extern const char *zfs_history_event_names[];

385 extern char *zpool_vdev_name(libzfs_handle_t *, zpool_handle_t *, nvlist_t *,
386     boolean_t verbose);
387 extern int zpool_upgrade(zpool_handle_t *, uint64_t);
388 extern int zpool_get_history(zpool_handle_t *, nvlist_t **);
389 extern int zpool_history_unpack(char *, uint64_t, uint64_t *,
390     nvlist_t ***, uint_t *);

```

```

391 extern void zpool_obj_to_path(zpool_handle_t *, uint64_t, uint64_t, char *,
392     size_t len);
393 extern int zfs_ioctl(libzfs_handle_t *, int, struct zfs_cmd *);
394 extern int zpool_get_physpath(zpool_handle_t *, char *, size_t);
395 extern void zpool_explain_recover(libzfs_handle_t *, const char *, int,
396     nvlist_t *);
397
398 /*
399  * Basic handle manipulations.  These functions do not create or destroy the
400  * underlying datasets, only the references to them.
401  */
402 extern zfs_handle_t *zfs_open(libzfs_handle_t *, const char *, int);
403 extern zfs_handle_t *zfs_handle_dup(zfs_handle_t *);
404 extern void zfs_close(zfs_handle_t *);
405 extern zfs_type_t zfs_get_type(const zfs_handle_t *);
406 extern const char *zfs_get_name(const zfs_handle_t *);
407 extern zpool_handle_t *zfs_get_pool_handle(const zfs_handle_t *);
408
409 /*
410  * Property management functions.  Some functions are shared with the kernel,
411  * and are found in sys/fs/zfs.h.
412  */
413
414 /*
415  * zfs dataset property management
416  */
417 extern const char *zfs_prop_default_string(zfs_prop_t);
418 extern uint64_t zfs_prop_default_numeric(zfs_prop_t);
419 extern const char *zfs_prop_column_name(zfs_prop_t);
420 extern boolean_t zfs_prop_align_right(zfs_prop_t);
421
422 extern nvlist_t *zfs_valid_proplist(libzfs_handle_t *, zfs_type_t,
423     nvlist_t *, uint64_t, zfs_handle_t *, const char *);
424
425 extern const char *zfs_prop_to_name(zfs_prop_t);
426 extern int zfs_prop_set(zfs_handle_t *, const char *, const char *);
427 extern int zfs_prop_get(zfs_handle_t *, zfs_prop_t, char *, size_t,
428     zprop_source_t *, char *, size_t, boolean_t);
429 extern int zfs_prop_get_recvd(zfs_handle_t *, const char *, char *, size_t,
430     boolean_t);
431 extern int zfs_prop_get_numeric(zfs_handle_t *, zfs_prop_t, uint64_t *,
432     zprop_source_t *, char *, size_t);
433 extern int zfs_prop_get_userquota_int(zfs_handle_t *zhp, const char *propname,
434     uint64_t *propvalue);
435 extern int zfs_prop_get_userquota(zfs_handle_t *zhp, const char *propname,
436     char *propbuf, int proplen, boolean_t literal);
437 extern int zfs_prop_get_written_int(zfs_handle_t *zhp, const char *propname,
438     uint64_t *propvalue);
439 extern int zfs_prop_get_written(zfs_handle_t *zhp, const char *propname,
440     char *propbuf, int proplen, boolean_t literal);
441 extern int zfs_prop_get_feature(zfs_handle_t *zhp, const char *propname,
442     char *buf, size_t len);
443 extern uint64_t zfs_prop_get_int(zfs_handle_t *, zfs_prop_t);
444 extern int zfs_prop_inherit(zfs_handle_t *, const char *, boolean_t);
445 extern const char *zfs_prop_values(zfs_prop_t);
446 extern int zfs_prop_is_string(zfs_prop_t prop);
447 extern nvlist_t *zfs_get_user_props(zfs_handle_t *);
448 extern nvlist_t *zfs_get_recvd_props(zfs_handle_t *);
449 extern nvlist_t *zfs_get_clones_nvlist(zfs_handle_t *);
450
451
452 typedef struct zprop_list {
453     int         pl_prop;
454     char        *pl_user_prop;
455     struct zprop_list *pl_next;
456     boolean_t   pl_all;

```

```

457     size_t      pl_width;
458     size_t      pl_recvd_width;
459     boolean_t   pl_fixed;
460 } zprop_list_t;
461
462 extern int zfs_expand_proplist(zfs_handle_t *, zprop_list_t **, boolean_t);
463 extern void zfs_prune_proplist(zfs_handle_t *, uint8_t *);
464
465 #define ZFS_MOUNTPOINT_NONE    "none"
466 #define ZFS_MOUNTPOINT_LEGACY "legacy"
467
468 #define ZFS_FEATURE_DISABLED   "disabled"
469 #define ZFS_FEATURE_ENABLED    "enabled"
470 #define ZFS_FEATURE_ACTIVE     "active"
471
472 #define ZFS_UNSUPPORTED_INACTIVE    "inactive"
473 #define ZFS_UNSUPPORTED_READONLY    "readonly"
474
475 /*
476  * zpool property management
477  */
478 extern int zpool_expand_proplist(zpool_handle_t *, zprop_list_t **);
479 extern int zpool_prop_get_feature(zpool_handle_t *, const char *, char *,
480     size_t);
481 extern const char *zpool_prop_default_string(zpool_prop_t);
482 extern uint64_t zpool_prop_default_numeric(zpool_prop_t);
483 extern const char *zpool_prop_column_name(zpool_prop_t);
484 extern boolean_t zpool_prop_align_right(zpool_prop_t);
485
486 /*
487  * Functions shared by zfs and zpool property management.
488  */
489 extern int zprop_iter(zprop_func func, void *cb, boolean_t show_all,
490     boolean_t ordered, zfs_type_t type);
491 extern int zprop_get_list(libzfs_handle_t *, char *, zprop_list_t **,
492     zfs_type_t);
493 extern void zprop_free_list(zprop_list_t *);
494
495 #define ZFS_GET_NCOLS    5
496
497 typedef enum {
498     GET_COL_NONE,
499     GET_COL_NAME,
500     GET_COL_PROPERTY,
501     GET_COL_VALUE,
502     GET_COL_RECVD,
503     GET_COL_SOURCE
504 } zfs_get_column_t;
505
506 /*
507  * Functions for printing zfs or zpool properties
508  */
509 typedef struct zprop_get_cbdata {
510     int cb_sources;
511     zfs_get_column_t cb_columns[ZFS_GET_NCOLS];
512     int cb_colwidths[ZFS_GET_NCOLS + 1];
513     boolean_t cb_scripted;
514     boolean_t cb_literal;
515     boolean_t cb_first;
516     zprop_list_t *cb_proplist;
517     zfs_type_t cb_type;
518 } zprop_get_cbdata_t;
519
520 void zprop_print_one_property(const char *, zprop_get_cbdata_t *,
521     const char *, const char *, zprop_source_t, const char *,
522     const char *);

```

```

524 /*
525  * Iterator functions.
526  */
527 typedef int (*zfs_iter_f)(zfs_handle_t *, void *);
528 extern int zfs_iter_root(libzfs_handle_t *, zfs_iter_f, void *);
529 extern int zfs_iter_children(zfs_handle_t *, zfs_iter_f, void *);
530 extern int zfs_iter_dependents(zfs_handle_t *, boolean_t, zfs_iter_f, void *);
531 extern int zfs_iter_filesystems(zfs_handle_t *, zfs_iter_f, void *);
532 extern int zfs_iter_snapshots(zfs_handle_t *, zfs_iter_f, void *);
533 extern int zfs_iter_snapshots_sorted(zfs_handle_t *, zfs_iter_f, void *);
534 extern int zfs_iter_snapspec(zfs_handle_t *, const char *, zfs_iter_f, void *);

536 typedef struct get_all_cb {
537     zfs_handle_t    **cb_handles;
538     size_t          cb_alloc;
539     size_t          cb_used;
540     boolean_t       cb_verbose;
541     int             (*cb_getone)(zfs_handle_t *, void *);
542 } get_all_cb_t;

544 void libzfs_add_handle(get_all_cb_t *, zfs_handle_t *);
545 int libzfs_dataset_cmp(const void *, const void *);

547 /*
548  * Functions to create and destroy datasets.
549  */
550 extern int zfs_create(libzfs_handle_t *, const char *, zfs_type_t,
551     nvlist_t *);
552 extern int zfs_create_ancestors(libzfs_handle_t *, const char *);
553 extern int zfs_destroy(zfs_handle_t *, boolean_t);
554 extern int zfs_destroy_snaps(zfs_handle_t *, char *, boolean_t);
555 extern int zfs_destroy_snaps_nvl(libzfs_handle_t *, nvlist_t *, boolean_t);
556 extern int zfs_clone(zfs_handle_t *, const char *, nvlist_t *);
557 extern int zfs_snapshot(libzfs_handle_t *, const char *, boolean_t, nvlist_t *);
558 extern int zfs_snapshot_nvl(libzfs_handle_t *hdl, nvlist_t *snaps,
559     nvlist_t *props);
560 extern int zfs_rollback(zfs_handle_t *, zfs_handle_t *, boolean_t);
561 extern int zfs_rename(zfs_handle_t *, const char *, boolean_t, boolean_t);

563 typedef struct sendflags {
564     /* print informational messages (ie, -v was specified) */
565     boolean_t verbose;

567     /* recursive send (ie, -R) */
568     boolean_t replicate;

570     /* for incrementals, do all intermediate snapshots */
571     boolean_t doall;

573     /* if dataset is a clone, do incremental from its origin */
574     boolean_t fromorigin;

576     /* do deduplication */
577     boolean_t dedup;

579     /* send properties (ie, -p) */
580     boolean_t props;

582     /* do not send (no-op, ie. -n) */
583     boolean_t dryrun;

585     /* parsable verbose output (ie. -P) */
586     boolean_t parsable;

588     /* show progress (ie. -v) */

```

```

589     boolean_t progress;
590 } sendflags_t;

592 typedef boolean_t (snapfilter_cb_t)(zfs_handle_t *, void *);

594 extern int zfs_send(zfs_handle_t *, const char *, const char *,
595     sendflags_t *, int, snapfilter_cb_t, void *, nvlist_t **);

597 extern int zfs_promote(zfs_handle_t *);
598 extern int zfs_hold(zfs_handle_t *, const char *, const char *,
599     boolean_t, int);
600 extern int zfs_hold_nvl(zfs_handle_t *, int, nvlist_t *);
601 extern int zfs_release(zfs_handle_t *, const char *, const char *, boolean_t);
602 extern int zfs_get_holds(zfs_handle_t *, nvlist_t **);
603 extern uint64_t zvol_volsize_to_reservation(uint64_t, nvlist_t *);

605 typedef int (*zfs_userspace_cb_t)(void *arg, const char *domain,
606     uid_t rid, uint64_t space);

608 extern int zfs_userspace(zfs_handle_t *, zfs_userquota_prop_t,
609     zfs_userspace_cb_t, void *);

611 extern int zfs_get_fsacl(zfs_handle_t *, nvlist_t **);
612 extern int zfs_set_fsacl(zfs_handle_t *, boolean_t, nvlist_t *);

614 typedef struct recvflags {
615     /* print informational messages (ie, -v was specified) */
616     boolean_t verbose;

618     /* the destination is a prefix, not the exact fs (ie, -d) */
619     boolean_t isprefix;

621     /*
622      * Only the tail of the sent snapshot path is appended to the
623      * destination to determine the received snapshot name (ie, -e).
624      */
625     boolean_t istail;

627     /* do not actually do the recv, just check if it would work (ie, -n) */
628     boolean_t dryrun;

630     /* rollback/destroy filesystems as necessary (eg, -F) */
631     boolean_t force;

633     /* set "canmount=off" on all modified filesystems */
634     boolean_t canmountoff;

636     /* byteswap flag is used internally; callers need not specify */
637     boolean_t byteswap;

639     /* do not mount file systems as they are extracted (private) */
640     boolean_t nomount;
641 } recvflags_t;

```

unchanged\_portion\_omitted

```

*****
111557 Wed May 29 20:27:08 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
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 * Copyright (c) 2013 Steven Hartland. All rights reserved.
29 #endif /* ! codereview */
30 */

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

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

59 #include "zfs_namecheck.h"
60 #include "zfs_prop.h"

```

```

61 #include "libzfs_impl.h"
62 #include "zfs_deleg.h"

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

67 /*
68  * Given a single type (not a mask of types), return the type in a human
69  * readable form.
70  */
71 const char *
72 zfs_type_to_name(zfs_type_t type)
73 {
74     switch (type) {
75     case ZFS_TYPE_FILESYSTEM:
76         return (dgettext(TEXT_DOMAIN, "filesystem"));
77     case ZFS_TYPE_SNAPSHOT:
78         return (dgettext(TEXT_DOMAIN, "snapshot"));
79     case ZFS_TYPE_VOLUME:
80         return (dgettext(TEXT_DOMAIN, "volume"));
81     }

83     return (NULL);
84 }

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

105    /*
106     * The user is requesting more than one type of dataset. If this is the
107     * case, consult the path itself. If we're looking for a snapshot, and
108     * a '@' is found, then report it as "snapshot". Otherwise, remove the
109     * snapshot attribute and try again.
110     */
111    if (types & ZFS_TYPE_SNAPSHOT) {
112        if (strchr(path, '@') != NULL)
113            return (dgettext(TEXT_DOMAIN, "snapshot"));
114        return (path_to_str(path, types & ~ZFS_TYPE_SNAPSHOT));
115    }

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

125    assert(types & ZFS_TYPE_VOLUME);
126    return (dgettext(TEXT_DOMAIN, "volume"));

```



```

127 }
129 /*
130 * Validate a ZFS path. This is used even before trying to open the dataset, to
131 * provide a more meaningful error message. We call zfs_error_aux() to
132 * explain exactly why the name was not valid.
133 */
134 int
135 zfs_validate_name(libzfs_handle_t *hdl, const char *path, int type,
136                 boolean_t modifying)
137 {
138     namecheck_err_t why;
139     char what;
141
142     (void) zfs_prop_get_table();
143     if (dataset_namecheck(path, &why, &what) != 0) {
144         if (hdl != NULL) {
145             switch (why) {
146                 case NAME_ERR_TOOLONG:
147                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
148                                     "name is too long"));
149                     break;
150
151                 case NAME_ERR_LEADING_SLASH:
152                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
153                                     "leading slash in name"));
154                     break;
155
156                 case NAME_ERR_EMPTY_COMPONENT:
157                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
158                                     "empty component in name"));
159                     break;
160
161                 case NAME_ERR_TRAILING_SLASH:
162                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
163                                     "trailing slash in name"));
164                     break;
165
166                 case NAME_ERR_INVALIDCHAR:
167                     zfs_error_aux(hdl,
168                                     dgettext(TEXT_DOMAIN, "invalid character "
169                                     "'%c' in name"), what);
170                     break;
171
172                 case NAME_ERR_MULTIPLE_AT:
173                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
174                                     "multiple '@' delimiters in name"));
175                     break;
176
177                 case NAME_ERR_NOLETTER:
178                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
179                                     "pool doesn't begin with a letter"));
180                     break;
181
182                 case NAME_ERR_RESERVED:
183                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
184                                     "name is reserved"));
185                     break;
186
187                 case NAME_ERR_DISKLIKE:
188                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
189                                     "reserved disk name"));
190                     break;
191             }
192         }
193     }

```

```

193         return (0);
194     }
196     if (!(type & ZFS_TYPE_SNAPSHOT) && strchr(path, '@') != NULL) {
197         if (hdl != NULL)
198             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
199                                     "snapshot delimiter '@' in filesystem name"));
200         return (0);
201     }
203     if (type == ZFS_TYPE_SNAPSHOT && strchr(path, '@') == NULL) {
204         if (hdl != NULL)
205             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
206                                     "missing '@' delimiter in snapshot name"));
207         return (0);
208     }
210     if (modifying && strchr(path, '%') != NULL) {
211         if (hdl != NULL)
212             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
213                                     "invalid character %c in name"), '%');
214         return (0);
215     }
217     return (-1);
218 }
220 int
221 zfs_name_valid(const char *name, zfs_type_t type)
222 {
223     if (type == ZFS_TYPE_POOL)
224         return (zpool_name_valid(NULL, B_FALSE, name));
225     return (zfs_validate_name(NULL, name, type, B_FALSE));
226 }
228 /*
229 * This function takes the raw DSL properties, and filters out the user-defined
230 * properties into a separate nvlist.
231 */
232 static nvlist_t *
233 process_user_props(zfs_handle_t *zhp, nvlist_t *props)
234 {
235     libzfs_handle_t *hdl = zhp->zfs_hdl;
236     nvpair_t *elem;
237     nvlist_t *propval;
238     nvlist_t *nvl;
240     if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
241         (void) no_memory(hdl);
242         return (NULL);
243     }
245     elem = NULL;
246     while ((elem = nvlist_next_nvpair(props, elem)) != NULL) {
247         if (!zfs_prop_user(nvpair_name(elem)))
248             continue;
250         verify(nvpair_value_nvlist(elem, &propval) == 0);
251         if (nvlist_add_nvlist(nvl, nvpair_name(elem), propval) != 0) {
252             nvlist_free(nvl);
253             (void) no_memory(hdl);
254             return (NULL);
255         }
256     }
258     return (nvl);

```

```

259 }

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

267     if ((zph = zpool_open_canfail(hdl, pool_name)) != NULL) {
268         if (hdl->libzfs_pool_handles != NULL)
269             zph->zpool_next = hdl->libzfs_pool_handles;
270         hdl->libzfs_pool_handles = zph;
271     }
272     return (zph);
273 }

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

281     while ((zph != NULL) &&
282           (strcmp(pool_name, zpool_get_name(zph), len) != 0))
283           zph = zph->zpool_next;
284     return (zph);
285 }

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

299     len = strchrspn(zhp->zfs_name, "/@") + 1;
300     pool_name = zfs_alloc(zhp->zfs_hdl, len);
301     (void) strcpy(pool_name, zhp->zfs_name, len);

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

307     free(pool_name);
308     return (zph);
309 }

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

316     while (zph != NULL) {
317         next = zph->zpool_next;
318         zpool_close(zph);
319         zph = next;
320     }
321     hdl->libzfs_pool_handles = NULL;
322 }

324 /*

```

```

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

332     (void) strcpy(zc->zc_name, zhp->zfs_name, sizeof (zc->zc_name));

334     while (ioctl(hdl->libzfs_fd, ZFS_IOC_OBJSET_STATS, zc) != 0) {
335         if (errno == ENOMEM) {
336             if (zcmd_expand_dst_nvlist(hdl, zc) != 0) {
337                 return (-1);
338             }
339         } else {
340             return (-1);
341         }
342     }
343     return (0);
344 }

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

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

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

362     while (ioctl(hdl->libzfs_fd, ZFS_IOC_OBJSET_RECVD_PROPS, &zc) != 0) {
363         if (errno == ENOMEM) {
364             if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
365                 return (-1);
366             }
367         } else {
368             zcmd_free_nvlists(&zc);
369             return (-1);
370         }
371     }

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

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

381     return (0);
382 }

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

389     zhp->zfs_dmstats = zc->zc_objset_stats; /* structure assignment */

```

```

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

```

```

457         abort();
458
459     if (zhp->zfs_dmustats.dds_is_snapshot)
460         zhp->zfs_type = ZFS_TYPE_SNAPSHOT;
461     else if (zhp->zfs_dmustats.dds_type == DMU_OST_ZVOL)
462         zhp->zfs_type = ZFS_TYPE_VOLUME;
463     else if (zhp->zfs_dmustats.dds_type == DMU_OST_ZFS)
464         zhp->zfs_type = ZFS_TYPE_FILESYSTEM;
465     else
466         abort(); /* we should never see any other types */
467
468     if ((zhp->zpool_hdl = zpool_handle(zhp)) == NULL)
469         return (-1);
470
471     return (0);
472 }
473
474 zfs_handle_t *
475 make_dataset_handle(libzfs_handle_t *hdl, const char *path)
476 {
477     zfs_cmd_t zc = { 0 };
478
479     zfs_handle_t *zhp = calloc(sizeof (zfs_handle_t), 1);
480
481     if (zhp == NULL)
482         return (NULL);
483
484     zhp->zfs_hdl = hdl;
485     (void) strncpy(zhp->zfs_name, path, sizeof (zhp->zfs_name));
486     if (zcmd_alloc_dst_nvlist(hdl, &zc, 0) != 0) {
487         free(zhp);
488         return (NULL);
489     }
490     if (get_stats_ioctl(zhp, &zc) == -1) {
491         zcmd_free_nvlists(&zc);
492         free(zhp);
493         return (NULL);
494     }
495     if (make_dataset_handle_common(zhp, &zc) == -1) {
496         free(zhp);
497         zhp = NULL;
498     }
499     zcmd_free_nvlists(&zc);
500     return (zhp);
501 }
502
503 zfs_handle_t *
504 make_dataset_handle_zc(libzfs_handle_t *hdl, zfs_cmd_t *zc)
505 {
506     zfs_handle_t *zhp = calloc(sizeof (zfs_handle_t), 1);
507
508     if (zhp == NULL)
509         return (NULL);
510
511     zhp->zfs_hdl = hdl;
512     (void) strncpy(zhp->zfs_name, zc->zname, sizeof (zhp->zfs_name));
513     if (make_dataset_handle_common(zhp, zc) == -1) {
514         free(zhp);
515         return (NULL);
516     }
517     return (zhp);
518 }
519
520 zfs_handle_t *
521 zfs_handle_dup(zfs_handle_t *zhp_orig)
522 {

```

```

523     zfs_handle_t *zhp = calloc(sizeof(zfs_handle_t), 1);
524
525     if (zhp == NULL)
526         return (NULL);
527
528     zhp->zfs_hdl = zhp_orig->zfs_hdl;
529     zhp->zpool_hdl = zhp_orig->zpool_hdl;
530     (void) strcpy(zhp->zfs_name, zhp_orig->zfs_name,
531                 sizeof(zhp->zfs_name));
532     zhp->zfs_type = zhp_orig->zfs_type;
533     zhp->zfs_head_type = zhp_orig->zfs_head_type;
534     zhp->zfs_dmstats = zhp_orig->zfs_dmstats;
535     if (zhp_orig->zfs_props != NULL) {
536         if (nvlist_dup(zhp_orig->zfs_props, &zhp->zfs_props, 0) != 0) {
537             (void) no_memory(zhp->zfs_hdl);
538             zfs_close(zhp);
539             return (NULL);
540         }
541     }
542     if (zhp_orig->zfs_user_props != NULL) {
543         if (nvlist_dup(zhp_orig->zfs_user_props,
544                     &zhp->zfs_user_props, 0) != 0) {
545             (void) no_memory(zhp->zfs_hdl);
546             zfs_close(zhp);
547             return (NULL);
548         }
549     }
550     if (zhp_orig->zfs_recvd_props != NULL) {
551         if (nvlist_dup(zhp_orig->zfs_recvd_props,
552                     &zhp->zfs_recvd_props, 0) != 0) {
553             (void) no_memory(zhp->zfs_hdl);
554             zfs_close(zhp);
555             return (NULL);
556         }
557     }
558     zhp->zfs_mntcheck = zhp_orig->zfs_mntcheck;
559     if (zhp_orig->zfs_mntopts != NULL) {
560         zhp->zfs_mntopts = zfs_strdup(zhp_orig->zfs_hdl,
561                                     zhp_orig->zfs_mntopts);
562     }
563     zhp->zfs_props_table = zhp_orig->zfs_props_table;
564     return (zhp);
565 }
566
567 /*
568 * Opens the given snapshot, filesystem, or volume. The 'types'
569 * argument is a mask of acceptable types. The function will print an
570 * appropriate error message and return NULL if it can't be opened.
571 */
572 zfs_handle_t *
573 zfs_open(libzfs_handle_t *hdl, const char *path, int types)
574 {
575     zfs_handle_t *zhp;
576     char errbuf[1024];
577
578     (void) snprintf(errbuf, sizeof(errbuf),
579                    dgettext(TEXT_DOMAIN, "cannot open '%s'"), path);
580
581     /*
582      * Validate the name before we even try to open it.
583      */
584     if (!zfs_validate_name(hdl, path, ZFS_TYPE_DATASET, B_FALSE)) {
585         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
586                                   "invalid dataset name"));
587         (void) zfs_error(hdl, EZFS_INVALIDNAME, errbuf);
588         return (NULL);

```

```

589     }
590
591     /*
592      * Try to get stats for the dataset, which will tell us if it exists.
593      */
594     errno = 0;
595     if ((zhp = make_dataset_handle(hdl, path)) == NULL) {
596         (void) zfs_standard_error(hdl, errno, errbuf);
597         return (NULL);
598     }
599
600     if (!(types & zhp->zfs_type)) {
601         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
602         zfs_close(zhp);
603         return (NULL);
604     }
605
606     return (zhp);
607 }
608
609 /*
610 * Release a ZFS handle. Nothing to do but free the associated memory.
611 */
612 void
613 zfs_close(zfs_handle_t *zhp)
614 {
615     if (zhp->zfs_mntopts)
616         free(zhp->zfs_mntopts);
617     nvlist_free(zhp->zfs_props);
618     nvlist_free(zhp->zfs_user_props);
619     nvlist_free(zhp->zfs_recvd_props);
620     free(zhp);
621 }
622
623 typedef struct mnttab_node {
624     struct mnttab mtn_mt;
625     avl_node_t mtn_node;
626 } mnttab_node_t;
627
628 static int
629 libzfs_mnttab_cache_compare(const void *arg1, const void *arg2)
630 {
631     const mnttab_node_t *mtn1 = arg1;
632     const mnttab_node_t *mtn2 = arg2;
633     int rv;
634
635     rv = strcmp(mtn1->mtn_mt.mnt_special, mtn2->mtn_mt.mnt_special);
636
637     if (rv == 0)
638         return (0);
639     return (rv > 0 ? 1 : -1);
640 }
641
642 void
643 libzfs_mnttab_init(libzfs_handle_t *hdl)
644 {
645     assert(avl_numnodes(&hdl->libzfs_mnttab_cache) == 0);
646     avl_create(&hdl->libzfs_mnttab_cache, libzfs_mnttab_cache_compare,
647              sizeof(mnttab_node_t), offsetof(mnttab_node_t, mtn_node));
648 }
649
650 void
651 libzfs_mnttab_update(libzfs_handle_t *hdl)
652 {
653     struct mnttab entry;

```

```

655     rewind(hdl->libzfs_mnttab);
656     while (getmntent(hdl->libzfs_mnttab, &entry) == 0) {
657         mnttab_node_t *mtn;

659         if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0)
660             continue;
661         mtn = zfs_alloc(hdl, sizeof (mnttab_node_t));
662         mtn->mtn_mt.mnt_special = zfs_strdup(hdl, entry.mnt_special);
663         mtn->mtn_mt.mnt_mountp = zfs_strdup(hdl, entry.mnt_mountp);
664         mtn->mtn_mt.mnt_fstype = zfs_strdup(hdl, entry.mnt_fstype);
665         mtn->mtn_mt.mnt_mntopts = zfs_strdup(hdl, entry.mnt_mntopts);
666         avl_add(&hdl->libzfs_mnttab_cache, mtn);
667     }
668 }

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

676     while (mtn = avl_destroy_nodes(&hdl->libzfs_mnttab_cache, &cookie)) {
677         free(mtn->mtn_mt.mnt_special);
678         free(mtn->mtn_mt.mnt_mountp);
679         free(mtn->mtn_mt.mnt_fstype);
680         free(mtn->mtn_mt.mnt_mntopts);
681         free(mtn);
682     }
683     avl_destroy(&hdl->libzfs_mnttab_cache);
684 }

686 void
687 libzfs_mnttab_cache(libzfs_handle_t *hdl, boolean_t enable)
688 {
689     hdl->libzfs_mnttab_enable = enable;
690 }

692 int
693 libzfs_mnttab_find(libzfs_handle_t *hdl, const char *fsname,
694                   struct mnttab *entry)
695 {
696     mnttab_node_t find;
697     mnttab_node_t *mtn;

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

702         if (avl_numnodes(&hdl->libzfs_mnttab_cache))
703             libzfs_mnttab_fini(hdl);
704         rewind(hdl->libzfs_mnttab);
705         srch.mnt_special = (char *)fsname;
706         srch.mnt_fstype = MNTTYPE_ZFS;
707         if (getmntany(hdl->libzfs_mnttab, entry, &srch) == 0)
708             return (0);
709         else
710             return (ENOENT);
711     }

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

716     find.mtn_mt.mnt_special = (char *)fsname;
717     mtn = avl_find(&hdl->libzfs_mnttab_cache, &find, NULL);
718     if (mtn) {
719         *entry = mtn->mtn_mt;
720         return (0);

```

```

721     }
722     return (ENOENT);
723 }

725 void
726 libzfs_mnttab_add(libzfs_handle_t *hdl, const char *special,
727                  const char *mountp, const char *mntopts)
728 {
729     mnttab_node_t *mtn;

731     if (avl_numnodes(&hdl->libzfs_mnttab_cache) == 0)
732         return;
733     mtn = zfs_alloc(hdl, sizeof (mnttab_node_t));
734     mtn->mtn_mt.mnt_special = zfs_strdup(hdl, special);
735     mtn->mtn_mt.mnt_mountp = zfs_strdup(hdl, mountp);
736     mtn->mtn_mt.mnt_fstype = zfs_strdup(hdl, MNTTYPE_ZFS);
737     mtn->mtn_mt.mnt_mntopts = zfs_strdup(hdl, mntopts);
738     avl_add(&hdl->libzfs_mnttab_cache, mtn);
739 }

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

747     find.mtn_mt.mnt_special = (char *)fsname;
748     if (ret = avl_find(&hdl->libzfs_mnttab_cache, (void *)&find, NULL)) {
749         avl_remove(&hdl->libzfs_mnttab_cache, ret);
750         free(ret->mtn_mt.mnt_special);
751         free(ret->mtn_mt.mnt_mountp);
752         free(ret->mtn_mt.mnt_fstype);
753         free(ret->mtn_mt.mnt_mntopts);
754         free(ret);
755     }
756 }

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

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

766     *spa_version = zpool_get_prop_int(zpool_handle,
767                                       ZPOOL_PROP_VERSION, NULL);
768     return (0);
769 }

771 /*
772  * The choice of reservation property depends on the SPA version.
773  */
774 static int
775 zfs_which_resv_prop(zfs_handle_t *zhp, zfs_prop_t *resv_prop)
776 {
777     int spa_version;

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

782     if (spa_version >= SPA_VERSION_REFRESERVATION)
783         *resv_prop = ZFS_PROP_REFRESERVATION;
784     else
785         *resv_prop = ZFS_PROP_RESERVATION;

```

```

787     return (0);
788 }

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

807     if (nvlist_alloc(&ret, NV_UNIQUE_NAME, 0) != 0) {
808         (void) no_memory(hdl);
809         return (NULL);
810     }

812     /*
813      * Make sure this property is valid and applies to this type.
814      */

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

820         prop = zfs_name_to_prop(propname);
821         if (prop == ZPROP_INVALID && zfs_prop_user(propname)) {
822             /*
823              * This is a user property: make sure it's a
824              * string, and that it's less than ZAP_MAXNAMELEN.
825              */
826             if (nvpair_type(elem) != DATA_TYPE_STRING) {
827                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
828                     "'%s' must be a string"), propname);
829                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
830                 goto error;
831             }

833             if (strlen(nvpair_name(elem)) >= ZAP_MAXNAMELEN) {
834                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
835                     "property name '%s' is too long"),
836                     propname);
837                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
838                 goto error;
839             }

841             (void) nvpair_value_string(elem, &strval);
842             if (nvlist_add_string(ret, propname, strval) != 0) {
843                 (void) no_memory(hdl);
844                 goto error;
845             }
846             continue;
847         }

849         /*
850          * Currently, only user properties can be modified on
851          * snapshots.
852          */

```

```

853     if (type == ZFS_TYPE_SNAPSHOT) {
854         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
855             "this property can not be modified for snapshots"));
856         (void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
857         goto error;
858     }

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

867         if (userquota_decode(propname, zoned,
868             &uqtype, domain, sizeof(domain), &rid) != 0) {
869             zfs_error_aux(hdl,
870                 dgettext(TEXT_DOMAIN,
871                     "'%s' has an invalid user/group name"),
872                     propname);
873             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
874             goto error;
875         }

877         if (uqtype != ZFS_PROP_USERQUOTA &&
878             uqtype != ZFS_PROP_GROUPQUOTA) {
879             zfs_error_aux(hdl,
880                 dgettext(TEXT_DOMAIN, "'%s' is readonly"),
881                 propname);
882             (void) zfs_error(hdl, EZFS_PROPREADONLY,
883                 errbuf);
884             goto error;
885         }

887         if (nvpair_type(elem) == DATA_TYPE_STRING) {
888             (void) nvpair_value_string(elem, &strval);
889             if (strcmp(strval, "none") == 0) {
890                 intval = 0;
891             } else if (zfs_nicestrtonum(hdl,
892                 strval, &intval) != 0) {
893                 (void) zfs_error(hdl,
894                     EZFS_BADPROP, errbuf);
895                 goto error;
896             }
897         } else if (nvpair_type(elem) ==
898             DATA_TYPE_UINT64) {
899             (void) nvpair_value_uint64(elem, &intval);
900             if (intval == 0) {
901                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
902                     "use 'none' to disable "
903                     "userquota/groupquota"));
904                 goto error;
905             }
906         } else {
907             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
908                 "'%s' must be a number"), propname);
909             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
910             goto error;
911         }

913         /*
914          * Encode the prop name as
915          * userquota@chex-rid>-domain, to make it easy
916          * for the kernel to decode.
917          */
918         (void) snprintf(newpropname, sizeof(newpropname),

```

```

919         "%s%llx-%s", zfs_userquota_prop_prefixes[uqtype],
920         (longlong_t)rid, domain);
921         valary[0] = uqtype;
922         valary[1] = rid;
923         valary[2] = intval;
924         if (nvlist_add_uint64_array(ret, newpropname,
925         valary, 3) != 0) {
926             (void) no_memory(hdl);
927             goto error;
928         }
929         continue;
930     } else if (prop == ZPROP_INVALID && zfs_prop_written(propname)) {
931         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
932         "'%s' is readonly"),
933         propname);
934         (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
935         goto error;
936     }
937
938     if (prop == ZPROP_INVALID) {
939         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
940         "invalid property '%s'"), propname);
941         (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
942         goto error;
943     }
944
945     if (!zfs_prop_valid_for_type(prop, type)) {
946         zfs_error_aux(hdl,
947         dgettext(TEXT_DOMAIN, "'%s' does not "
948         "apply to datasets of this type"), propname);
949         (void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
950         goto error;
951     }
952
953     if (zfs_prop_readonly(prop) &&
954         (!zfs_prop_setonce(prop) || zhp != NULL)) {
955         zfs_error_aux(hdl,
956         dgettext(TEXT_DOMAIN, "'%s' is readonly"),
957         propname);
958         (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
959         goto error;
960     }
961
962     if (zprop_parse_value(hdl, elem, prop, type, ret,
963         &strval, &intval, errbuf) != 0)
964         goto error;
965
966     /*
967     * Perform some additional checks for specific properties.
968     */
969     switch (prop) {
970     case ZFS_PROP_VERSION:
971     {
972         int version;
973
974         if (zhp == NULL)
975             break;
976         version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
977         if (intval < version) {
978             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
979             "Can not downgrade; already at version %u"),
980             version);
981             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
982             goto error;
983         }
984         break;

```

```

985     }
986
987     case ZFS_PROP_RECORDSIZE:
988     case ZFS_PROP_VOLBLOCKSIZE:
989         /* must be power of two within SPA_{MIN,MAX}BLOCKSIZE */
990         if (intval < SPA_MINBLOCKSIZE ||
991             intval > SPA_MAXBLOCKSIZE || !ISP2(intval)) {
992             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
993             "'%s' must be power of 2 from %u "
994             "to %uk"), propname,
995             (uint_t)SPA_MINBLOCKSIZE,
996             (uint_t)SPA_MAXBLOCKSIZE >> 10);
997             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
998             goto error;
999         }
1000         break;
1001
1002     case ZFS_PROP_MLSLABEL:
1003     {
1004         /*
1005         * Verify the mlslabel string and convert to
1006         * internal hex label string.
1007         */
1008
1009         m_label_t *new_sl;
1010         char *hex = NULL; /* internal label string */
1011
1012         /* Default value is already OK. */
1013         if (strcasecmp(strval, ZFS_MLSLABEL_DEFAULT) == 0)
1014             break;
1015
1016         /* Verify the label can be converted to binary form */
1017         if (((new_sl = m_label_alloc(MAC_LABEL)) == NULL) ||
1018             (str_to_label(strval, &new_sl, MAC_LABEL,
1019             L_NO_CORRECTION, NULL) == -1)) {
1020             goto badlabel;
1021         }
1022
1023         /* Now translate to hex internal label string */
1024         if (label_to_str(new_sl, &hex, M_INTERNAL,
1025             DEF_NAMES) != 0) {
1026             if (hex)
1027                 free(hex);
1028             goto badlabel;
1029         }
1030         m_label_free(new_sl);
1031
1032         /* If string is already in internal form, we're done. */
1033         if (strcmp(strval, hex) == 0) {
1034             free(hex);
1035             break;
1036         }
1037
1038         /* Replace the label string with the internal form. */
1039         (void) nvlist_remove(ret, zfs_prop_to_name(prop),
1040             DATA_TYPE_STRING);
1041         verify(nvlist_add_string(ret, zfs_prop_to_name(prop),
1042             hex) == 0);
1043         free(hex);
1044
1045         break;
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;
1054     }
1056     case ZFS_PROP_MOUNTPOINT:
1057     {
1058         namecheck_err_t why;
1060         if (strcmp(strval, ZFS_MOUNTPOINT_NONE) == 0 ||
1061             strcmp(strval, ZFS_MOUNTPOINT_LEGACY) == 0)
1062             break;
1064         if (mountpoint_namecheck(strval, &why) {
1065             switch (why) {
1066                 case NAME_ERR_LEADING_SLASH:
1067                     zfs_error_aux(hdl,
1068                         dgettext(TEXT_DOMAIN,
1069                             "'%s' must be an absolute path, "
1070                             "'none', or 'legacy'"), propname);
1071                     break;
1072                 case NAME_ERR_TOOLONG:
1073                     zfs_error_aux(hdl,
1074                         dgettext(TEXT_DOMAIN,
1075                             "component of '%s' is too long"),
1076                             propname);
1077                     break;
1078             }
1079             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1080             goto error;
1081         }
1082     }
1084     /*FALLTHRU*/
1086     case ZFS_PROP_SHARESMB:
1087     case ZFS_PROP_SHARENFS:
1088     /*
1089      * For the mountpoint and sharenfs or sharesmb
1090      * properties, check if it can be set in a
1091      * global/non-global zone based on
1092      * the zoned property value:
1093      *
1094      * -----
1095      * global zone          non-global zone
1096      * -----
1097      * zoned=on      mountpoint (no)    mountpoint (yes)
1098      *                sharenfs (no)     sharenfs (no)
1099      *                sharesmb (no)     sharesmb (no)
1100      *
1101      * zoned=off    mountpoint (yes)    N/A
1102      *                sharenfs (yes)
1103      *                sharesmb (yes)
1104      */
1105     if (zoned) {
1106         if (getzoneid() == GLOBAL_ZONEID) {
1107             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1108                 "'%s' cannot be set on "
1109                 "dataset in a non-global zone"),
1110                 propname);
1111             (void) zfs_error(hdl, EZFS_ZONED,
1112                 errbuf);
1113             goto error;
1114         } else if (prop == ZFS_PROP_SHARENFS ||
1115             prop == ZFS_PROP_SHARESMB) {
1116             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1117                 "'%s' cannot be set in "

```

```

1117         "a non-global zone"), propname);
1118         (void) zfs_error(hdl, EZFS_ZONED,
1119             errbuf);
1120         goto error;
1121     }
1122     } else if (getzoneid() != GLOBAL_ZONEID) {
1123     /*
1124      * If zoned property is 'off', this must be in
1125      * a global zone. If not, something is wrong.
1126      */
1127     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1128         "'%s' cannot be set while dataset "
1129         "'zoned' property is set"), propname);
1130     (void) zfs_error(hdl, EZFS_ZONED, errbuf);
1131     goto error;
1132     }
1134     /*
1135      * At this point, it is legitimate to set the
1136      * property. Now we want to make sure that the
1137      * property value is valid if it is sharenfs.
1138      */
1139     if ((prop == ZFS_PROP_SHARENFS ||
1140         prop == ZFS_PROP_SHARESMB) &&
1141         strcmp(strval, "on") != 0 &&
1142         strcmp(strval, "off") != 0) {
1143         zfs_share_proto_t proto;
1145         if (prop == ZFS_PROP_SHARESMB)
1146             proto = PROTO_SMB;
1147         else
1148             proto = PROTO_NFS;
1150         /*
1151          * Must be a valid sharing protocol
1152          * option string so init the libshare
1153          * in order to enable the parser and
1154          * then parse the options. We use the
1155          * control API since we don't care about
1156          * the current configuration and don't
1157          * want the overhead of loading it
1158          * until we actually do something.
1159          */
1161         if (zfs_init_libshare(hdl,
1162             SA_INIT_CONTROL_API) != SA_OK) {
1163             /*
1164              * An error occurred so we can't do
1165              * anything
1166              */
1167             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1168                 "'%s' cannot be set: problem "
1169                 "in share initialization"),
1170                 propname);
1171             (void) zfs_error(hdl, EZFS_BADPROP,
1172                 errbuf);
1173             goto error;
1174         }
1176         if (zfs_parse_options(strval, proto) != SA_OK) {
1177             /*
1178              * There was an error in parsing so
1179              * deal with it by issuing an error
1180              * message and leaving after
1181              * uninitialized the the libshare
1182              * interface.

```



```

1183     */
1184     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1185     "'%s' cannot be set to invalid "
1186     "options"), propname);
1187     (void) zfs_error(hdl, EZFS_BADPROP,
1188     errbuf);
1189     zfs_uninit_libshare(hdl);
1190     goto error;
1191 }
1192     zfs_uninit_libshare(hdl);
1193 }
1194
1195     break;
1196 case ZFS_PROP_UTF8ONLY:
1197     chosen_utf = (int)intval;
1198     break;
1199 case ZFS_PROP_NORMALIZE:
1200     chosen_normal = (int)intval;
1201     break;
1202 }
1203
1204 /*
1205  * For changes to existing volumes, we have some additional
1206  * checks to enforce.
1207  */
1208 if (type == ZFS_TYPE_VOLUME && zhp != NULL) {
1209     uint64_t volsize = zfs_prop_get_int(zhp,
1210     ZFS_PROP_VOLSIZE);
1211     uint64_t blocksize = zfs_prop_get_int(zhp,
1212     ZFS_PROP_VOLBLOCKSIZE);
1213     char buf[64];
1214
1215     switch (prop) {
1216     case ZFS_PROP_RESERVATION:
1217     case ZFS_PROP_REFRESERVATION:
1218         if (intval > volsize) {
1219             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1220             "'%s' is greater than current "
1221             "volume size"), propname);
1222             (void) zfs_error(hdl, EZFS_BADPROP,
1223             errbuf);
1224             goto error;
1225         }
1226         break;
1227
1228     case ZFS_PROP_VOLSIZE:
1229         if (intval % blocksize != 0) {
1230             zfs_nicenum(blocksize, buf,
1231             sizeof(buf));
1232             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1233             "'%s' must be a multiple of "
1234             "volume block size (%s)",
1235             propname, buf));
1236             (void) zfs_error(hdl, EZFS_BADPROP,
1237             errbuf);
1238             goto error;
1239         }
1240
1241         if (intval == 0) {
1242             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1243             "'%s' cannot be zero"),
1244             propname);
1245             (void) zfs_error(hdl, EZFS_BADPROP,
1246             errbuf);
1247             goto error;
1248         }

```

```

1249         break;
1250     }
1251 }
1252
1253 }
1254
1255 /*
1256  * If normalization was chosen, but no UTF8 choice was made,
1257  * enforce rejection of non-UTF8 names.
1258  */
1259 /*
1260  * If normalization was chosen, but rejecting non-UTF8 names
1261  * was explicitly not chosen, it is an error.
1262  */
1263 if (chosen_normal > 0 && chosen_utf < 0) {
1264     if (nvlst_add_uint64(ret,
1265     zfs_prop_to_name(ZFS_PROP_UTF8ONLY), 1) != 0) {
1266         (void) no_memory(hdl);
1267         goto error;
1268     }
1269 } else if (chosen_normal > 0 && chosen_utf == 0) {
1270     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1271     "'%s' must be set 'on' if normalization chosen"),
1272     zfs_prop_to_name(ZFS_PROP_UTF8ONLY));
1273     (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1274     goto error;
1275 }
1276 return (ret);
1277
1278 error:
1279     nvlst_free(ret);
1280     return (NULL);
1281 }
1282
1283 int
1284 zfs_add_synthetic_resv(zfs_handle_t *zhp, nvlst_t *nvl)
1285 {
1286     uint64_t old_volsize;
1287     uint64_t new_volsize;
1288     uint64_t old_reservation;
1289     uint64_t new_reservation;
1290     zfs_prop_t resv_prop;
1291     nvlst_t *props;
1292
1293     /*
1294      * If this is an existing volume, and someone is setting the volsize,
1295      * make sure that it matches the reservation, or add it if necessary.
1296      */
1297     old_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
1298     if (zfs_which_resv_prop(zhp, &resv_prop) < 0)
1299         return (-1);
1300     old_reservation = zfs_prop_get_int(zhp, resv_prop);
1301
1302     props = fnvlst_alloc();
1303     fnvlst_add_uint64(props, zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
1304     zfs_prop_get_int(zhp, ZFS_PROP_VOLBLOCKSIZE));
1305
1306     if ((zvol_volsize_to_reservation(old_volsize, props) !=
1307     old_reservation) || nvlst_exists(nvl,
1308     zfs_prop_to_name(resv_prop))) {
1309         fnvlst_free(props);
1310         return (0);
1311     }
1312     if (nvlst_lookup_uint64(nvl, zfs_prop_to_name(ZFS_PROP_VOLSIZE),
1313     &new_volsize) != 0) {
1314         fnvlst_free(props);
1315         return (-1);
1316     }

```

```

1315     new_reservation = zvol_volsize_to_reservation(new_volsize, props);
1316     fnvlist_free(props);

1318     if (nvlist_add_uint64(nvl, zfs_prop_to_name(resv_prop),
1319         new_reservation) != 0) {
1320         (void) no_memory(zhp->zfs_hdl);
1321         return (-1);
1322     }
1323     return (1);
1324 }

1326 void
1327 zfs_setprop_error(libzfs_handle_t *hdl, zfs_prop_t prop, int err,
1328     char *errbuf)
1329 {
1330     switch (err) {

1332     case ENOSPC:
1333         /*
1334          * For quotas and reservations, ENOSPC indicates
1335          * something different; setting a quota or reservation
1336          * doesn't use any disk space.
1337          */
1338         switch (prop) {
1339             case ZFS_PROP_QUOTA:
1340             case ZFS_PROP_REFQUOTA:
1341                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1342                     "size is less than current used or "
1343                     "reserved space"));
1344                 (void) zfs_error(hdl, EZFS_PROPSPACE, errbuf);
1345                 break;

1347             case ZFS_PROP_RESERVATION:
1348             case ZFS_PROP_REFRESERVATION:
1349                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1350                     "size is greater than available space"));
1351                 (void) zfs_error(hdl, EZFS_PROPSPACE, errbuf);
1352                 break;

1354             default:
1355                 (void) zfs_standard_error(hdl, err, errbuf);
1356                 break;
1357         }
1358         break;

1360     case EBUSY:
1361         (void) zfs_standard_error(hdl, EBUSY, errbuf);
1362         break;

1364     case EROFS:
1365         (void) zfs_error(hdl, EZFS_DSREADONLY, errbuf);
1366         break;

1368     case ENOTSUP:
1369         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1370             "pool and or dataset must be upgraded to set this "
1371             "property or value"));
1372         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
1373         break;

1375     case ERANGE:
1376         if (prop == ZFS_PROP_COMPRESSION) {
1377             (void) zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1378                 "property setting is not allowed on "
1379                 "bootable datasets"));
1380             (void) zfs_error(hdl, EZFS_NOTSUP, errbuf);

```

```

1381     } else {
1382         (void) zfs_standard_error(hdl, err, errbuf);
1383     }
1384     break;

1386     case EINVAL:
1387         if (prop == ZPROP_INVAL) {
1388             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1389         } else {
1390             (void) zfs_standard_error(hdl, err, errbuf);
1391         }
1392         break;

1394     case EOVERFLOW:
1395         /*
1396          * This platform can't address a volume this big.
1397          */
1398     #ifdef _ILP32
1399         if (prop == ZFS_PROP_VOLSIZE) {
1400             (void) zfs_error(hdl, EZFS_VOLTOOBIG, errbuf);
1401             break;
1402         }
1403     #endif
1404         /* FALLTHROUGH */
1405     default:
1406         (void) zfs_standard_error(hdl, err, errbuf);
1407     }
1408 }

1410 /*
1411  * Given a property name and value, set the property for the given dataset.
1412  */
1413 int
1414 zfs_prop_set(zfs_handle_t *zhp, const char *propname, const char *propval)
1415 {
1416     zfs_cmd_t zc = { 0 };
1417     int ret = -1;
1418     prop_changelist_t *cl = NULL;
1419     char errbuf[1024];
1420     libzfs_handle_t *hdl = zhp->zfs_hdl;
1421     nvlist_t *nvl = NULL, *realprops;
1422     zfs_prop_t prop;
1423     boolean_t do_prefix = B_TRUE;
1424     int added_resv;

1426     (void) snprintf(errbuf, sizeof (errbuf),
1427         dgettext(TEXT_DOMAIN, "cannot set property for '%s'"),
1428         zhp->zfs_name);

1430     if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0 ||
1431         nvlist_add_string(nvl, propname, propval) != 0) {
1432         (void) no_memory(hdl);
1433         goto error;
1434     }

1436     if ((realprops = zfs_valid_proplist(hdl, zhp->zfs_type, nvl,
1437         zfs_prop_get_int(zhp, ZFS_PROP_ZONED), zhp, errbuf)) == NULL)
1438         goto error;

1440     nvlist_free(nvl);
1441     nvl = realprops;

1443     prop = zfs_name_to_prop(propname);

1445     if (prop == ZFS_PROP_VOLSIZE) {
1446         if ((added_resv = zfs_add_synthetic_resv(zhp, nvl)) == -1)

```

```

1447         goto error;
1448     }
1450     if ((cl = changelist_gather(zhp, prop, 0, 0)) == NULL)
1451         goto error;
1453     if (prop == ZFS_PROP_MOUNTPOINT && changelist_haszonedchild(cl)) {
1454         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1455             "child dataset with inherited mountpoint is used "
1456             "in a non-global zone"));
1457         ret = zfs_error(hdl, EZFS_ZONED, errbuf);
1458         goto error;
1459     }
1461     /*
1462     * We don't want to unmount & remount the dataset when changing
1463     * its canmount property to 'on' or 'noauto'. We only use
1464     * the changelist logic to unmount when setting canmount=off.
1465     */
1466     if (prop == ZFS_PROP_CANMOUNT) {
1467         uint64_t idx;
1468         int err = zprop_string_to_index(prop, propval, &idx,
1469             ZFS_TYPE_DATASET);
1470         if (err == 0 && idx != ZFS_CANMOUNT_OFF)
1471             do_prefix = B_FALSE;
1472     }
1474     if (do_prefix && (ret = changelist_prefix(cl)) != 0)
1475         goto error;
1477     /*
1478     * Execute the corresponding ioctl() to set this property.
1479     */
1480     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
1482     if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1483         goto error;
1485     ret = zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
1487     if (ret != 0) {
1488         zfs_setprop_error(hdl, prop, errno, errbuf);
1489         if (added_resv && errno == ENOSPC) {
1490             /* clean up the volsize property we tried to set */
1491             uint64_t old_volsize = zfs_prop_get_int(zhp,
1492                 ZFS_PROP_VOLSIZE);
1493             nvlist_free(nvl);
1494             zcmd_free_nvlists(&zc);
1495             if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
1496                 goto error;
1497             if (nvlist_add_uint64(nvl,
1498                 zfs_prop_to_name(ZFS_PROP_VOLSIZE),
1499                 old_volsize) != 0)
1500                 goto error;
1501             if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1502                 goto error;
1503             (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
1504         }
1505     } else {
1506         if (do_prefix)
1507             ret = changelist_postfix(cl);
1509         /*
1510         * Refresh the statistics so the new property value
1511         * is reflected.
1512         */

```

```

1513         if (ret == 0)
1514             (void) get_stats(zhp);
1515     }
1517 error:
1518     nvlist_free(nvl);
1519     zcmd_free_nvlists(&zc);
1520     if (cl)
1521         changelist_free(cl);
1522     return (ret);
1523 }
1525 /*
1526 * Given a property, inherit the value from the parent dataset, or if received
1527 * is TRUE, revert to the received value, if any.
1528 */
1529 int
1530 zfs_prop_inherit(zfs_handle_t *zhp, const char *propname, boolean_t received)
1531 {
1532     zfs_cmd_t zc = { 0 };
1533     int ret;
1534     prop_changelist_t *cl;
1535     libzfs_handle_t *hdl = zhp->zfs_hdl;
1536     char errbuf[1024];
1537     zfs_prop_t prop;
1539     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
1540         "cannot inherit %s for '%s'", propname, zhp->zfs_name));
1542     zc.zc_cookie = received;
1543     if ((prop = zfs_name_to_prop(propname)) == ZPROP_INVALID) {
1544         /*
1545         * For user properties, the amount of work we have to do is very
1546         * small, so just do it here.
1547         */
1548         if (!zfs_prop_user(propname)) {
1549             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1550                 "invalid property"));
1551             return (zfs_error(hdl, EZFS_BADPROP, errbuf));
1552         }
1554         (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
1555         (void) strncpy(zc.zc_value, propname, sizeof (zc.zc_value));
1557         if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_INHERIT_PROP, &zc) != 0)
1558             return (zfs_standard_error(hdl, errno, errbuf));
1560         return (0);
1561     }
1563     /*
1564     * Verify that this property is inheritable.
1565     */
1566     if (zfs_prop_readonly(prop))
1567         return (zfs_error(hdl, EZFS_PROPREADONLY, errbuf));
1569     if (!zfs_prop_inheritable(prop) && !received)
1570         return (zfs_error(hdl, EZFS_PROPNONINHERIT, errbuf));
1572     /*
1573     * Check to see if the value applies to this type
1574     */
1575     if (!zfs_prop_valid_for_type(prop, zhp->zfs_type))
1576         return (zfs_error(hdl, EZFS_PROPTYPE, errbuf));
1578     /*

```

```

1579     * Normalize the name, to get rid of shorthand abbreviations.
1580     */
1581     propname = zfs_prop_to_name(prop);
1582     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
1583     (void) strncpy(zc.zc_value, propname, sizeof(zc.zc_value));

1585     if (prop == ZFS_PROP_MOUNTPOINT && getzoneid() == GLOBAL_ZONEID &&
1586         zfs_prop_get_int(zhp, ZFS_PROP_ZONED)) {
1587         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1588             "dataset is used in a non-global zone"));
1589         return (zfs_error(hdl, EZFS_ZONED, errbuf));
1590     }

1592     /*
1593     * Determine datasets which will be affected by this change, if any.
1594     */
1595     if ((cl = changelist_gather(zhp, prop, 0, 0)) == NULL)
1596         return (-1);

1598     if (prop == ZFS_PROP_MOUNTPOINT && changelist_haszonedchild(cl)) {
1599         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1600             "child dataset with inherited mountpoint is used "
1601             "in a non-global zone"));
1602         ret = zfs_error(hdl, EZFS_ZONED, errbuf);
1603         goto error;
1604     }

1606     if ((ret = changelist_prefix(cl)) != 0)
1607         goto error;

1609     if ((ret = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_INHERIT_PROP, &zc)) != 0) {
1610         return (zfs_standard_error(hdl, errno, errbuf));
1611     } else {
1613         if ((ret = changelist_postfix(cl)) != 0)
1614             goto error;

1616         /*
1617         * Refresh the statistics so the new property is reflected.
1618         */
1619         (void) get_stats(zhp);
1620     }

1622 error:
1623     changelist_free(cl);
1624     return (ret);
1625 }

1627 /*
1628 * True DSL properties are stored in an nvlist. The following two functions
1629 * extract them appropriately.
1630 */
1631 static uint64_t
1632 getprop_uint64(zfs_handle_t *zhp, zfs_prop_t prop, char **source)
1633 {
1634     nvlist_t *nv;
1635     uint64_t value;

1637     *source = NULL;
1638     if (nvlist_lookup_nvlist(zhp->zfs_props,
1639         zfs_prop_to_name(prop), &nv) == 0) {
1640         verify(nvlist_lookup_uint64(nv, ZPROP_VALUE, &value) == 0);
1641         (void) nvlist_lookup_string(nv, ZPROP_SOURCE, source);
1642     } else {
1643         verify(!zhp->zfs_props_table ||
1644             zhp->zfs_props_table[prop] == B_TRUE);

```

```

1645         value = zfs_prop_default_numeric(prop);
1646         *source = "";
1647     }

1649     return (value);
1650 }

1652 static char *
1653 getprop_string(zfs_handle_t *zhp, zfs_prop_t prop, char **source)
1654 {
1655     nvlist_t *nv;
1656     char *value;

1658     *source = NULL;
1659     if (nvlist_lookup_nvlist(zhp->zfs_props,
1660         zfs_prop_to_name(prop), &nv) == 0) {
1661         verify(nvlist_lookup_string(nv, ZPROP_VALUE, &value) == 0);
1662         (void) nvlist_lookup_string(nv, ZPROP_SOURCE, source);
1663     } else {
1664         verify(!zhp->zfs_props_table ||
1665             zhp->zfs_props_table[prop] == B_TRUE);
1666         if ((value = (char *)zfs_prop_default_string(prop)) == NULL)
1667             value = "";
1668         *source = "";
1669     }

1671     return (value);
1672 }

1674 static boolean_t
1675 zfs_is_recvd_props_mode(zfs_handle_t *zhp)
1676 {
1677     return (zhp->zfs_props == zhp->zfs_recvd_props);
1678 }

1680 static void
1681 zfs_set_recvd_props_mode(zfs_handle_t *zhp, uint64_t *cookie)
1682 {
1683     *cookie = (uint64_t)(uintptr_t)zhp->zfs_props;
1684     zhp->zfs_props = zhp->zfs_recvd_props;
1685 }

1687 static void
1688 zfs_unset_recvd_props_mode(zfs_handle_t *zhp, uint64_t *cookie)
1689 {
1690     zhp->zfs_props = (nvlist_t *) (uintptr_t) *cookie;
1691     *cookie = 0;
1692 }

1694 /*
1695 * Internal function for getting a numeric property. Both zfs_prop_get() and
1696 * zfs_prop_get_int() are built using this interface.
1697 *
1698 * Certain properties can be overridden using 'mount -o'. In this case, scan
1699 * the contents of the /etc/mnttab entry, searching for the appropriate options.
1700 * If they differ from the on-disk values, report the current values and mark
1701 * the source "temporary".
1702 */
1703 static int
1704 get_numeric_property(zfs_handle_t *zhp, zfs_prop_t prop, zprop_source_t *src,
1705     char **source, uint64_t *val)
1706 {
1707     zfs_cmd_t zc = { 0 };
1708     nvlist_t *zplprops = NULL;
1709     struct mnttab mnt;
1710     char *mntopt_on = NULL;

```

```

1711 char *mntopt_off = NULL;
1712 boolean_t received = zfs_is_recvd_props_mode(zhp);

1714 *source = NULL;

1716 switch (prop) {
1717 case ZFS_PROP_ATIME:
1718     mntopt_on = MNTOPT_ATIME;
1719     mntopt_off = MNTOPT_NOATIME;
1720     break;

1722 case ZFS_PROP_DEVICES:
1723     mntopt_on = MNTOPT_DEVICES;
1724     mntopt_off = MNTOPT_NODEVICES;
1725     break;

1727 case ZFS_PROP_EXEC:
1728     mntopt_on = MNTOPT_EXEC;
1729     mntopt_off = MNTOPT_NOEXEC;
1730     break;

1732 case ZFS_PROP_READONLY:
1733     mntopt_on = MNTOPT_RO;
1734     mntopt_off = MNTOPT_RW;
1735     break;

1737 case ZFS_PROP_SETUID:
1738     mntopt_on = MNTOPT_SETUID;
1739     mntopt_off = MNTOPT_NOSETUID;
1740     break;

1742 case ZFS_PROP_XATTR:
1743     mntopt_on = MNTOPT_XATTR;
1744     mntopt_off = MNTOPT_NOXATTR;
1745     break;

1747 case ZFS_PROP_NEMAND:
1748     mntopt_on = MNTOPT_NEMAND;
1749     mntopt_off = MNTOPT_NONEMAND;
1750     break;
1751 }

1753 /*
1754  * Because looking up the mount options is potentially expensive
1755  * (iterating over all of /etc/mnttab), we defer its calculation until
1756  * we're looking up a property which requires its presence.
1757  */
1758 if (!zhp->zfs_mntcheck &&
1759     (mntopt_on != NULL || prop == ZFS_PROP_MOUNTED)) {
1760     libzfs_handle_t *hdl = zhp->zfs_hdl;
1761     struct mnttab entry;

1763     if (libzfs_mnttab_find(hdl, zhp->zfs_name, &entry) == 0) {
1764         zhp->zfs_mntopts = zfs_strdup(hdl,
1765             entry.mnt_mntopts);
1766         if (zhp->zfs_mntopts == NULL)
1767             return (-1);
1768     }

1770     zhp->zfs_mntcheck = B_TRUE;
1771 }

1773 if (zhp->zfs_mntopts == NULL)
1774     mnt.mnt_mntopts = "";
1775 else
1776     mnt.mnt_mntopts = zhp->zfs_mntopts;

```

```

1778 switch (prop) {
1779 case ZFS_PROP_ATIME:
1780 case ZFS_PROP_DEVICES:
1781 case ZFS_PROP_EXEC:
1782 case ZFS_PROP_READONLY:
1783 case ZFS_PROP_SETUID:
1784 case ZFS_PROP_XATTR:
1785 case ZFS_PROP_NEMAND:
1786     *val = getprop_uint64(zhp, prop, source);

1788     if (received)
1789         break;

1791     if (hasmntopt(&mnt, mntopt_on) && !*val) {
1792         *val = B_TRUE;
1793         if (src)
1794             *src = ZPROP_SRC_TEMPORARY;
1795     } else if (hasmntopt(&mnt, mntopt_off) && *val) {
1796         *val = B_FALSE;
1797         if (src)
1798             *src = ZPROP_SRC_TEMPORARY;
1799     }
1800     break;

1802 case ZFS_PROP_CANMOUNT:
1803 case ZFS_PROP_VOLSIZE:
1804 case ZFS_PROP_QUOTA:
1805 case ZFS_PROP_REFQUOTA:
1806 case ZFS_PROP_RESERVATION:
1807 case ZFS_PROP_REFRESERVATION:
1808     *val = getprop_uint64(zhp, prop, source);

1810     if (*source == NULL) {
1811         /* not default, must be local */
1812         *source = zhp->zfs_name;
1813     }
1814     break;

1816 case ZFS_PROP_MOUNTED:
1817     *val = (zhp->zfs_mntopts != NULL);
1818     break;

1820 case ZFS_PROP_NUMCLONES:
1821     *val = zhp->zfs_dmustats.dds_num_clones;
1822     break;

1824 case ZFS_PROP_VERSION:
1825 case ZFS_PROP_NORMALIZE:
1826 case ZFS_PROP_UTF8ONLY:
1827 case ZFS_PROP_CASE:
1828     if (!zfs_prop_valid_for_type(prop, zhp->zfs_head_type) ||
1829         zcmd_alloc_dst_nvlist(zhp->zfs_hdl, &zc, 0) != 0)
1830         return (-1);
1831     (void) strcpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
1832     if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_OBJSET_ZPLPROPS, &zc) {
1833         zcmd_free_nvlists(&zc);
1834         return (-1);
1835     }
1836     if (zcmd_read_dst_nvlist(zhp->zfs_hdl, &zc, &zplprops) != 0 ||
1837         nvlist_lookup_uint64(zplprops, zfs_prop_to_name(prop),
1838             val) != 0) {
1839         zcmd_free_nvlists(&zc);
1840         return (-1);
1841     }
1842     if (zplprops)

```

```

1843         nvlist_free(zplprops);
1844         zcmd_free_nvlists(&zcmd);
1845         break;

1847     default:
1848         switch (zfs_prop_get_type(prop)) {
1849             case PROP_TYPE_NUMBER:
1850             case PROP_TYPE_INDEX:
1851                 *val = getprop_uint64(zhp, prop, source);
1852                 /*
1853                  * If we tried to use a default value for a
1854                  * readonly property, it means that it was not
1855                  * present.
1856                  */
1857                 if (zfs_prop_readonly(prop) &&
1858                     *source != NULL && (*source)[0] == '\0') {
1859                     *source = NULL;
1860                 }
1861                 break;

1863             case PROP_TYPE_STRING:
1864             default:
1865                 zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
1866                     "cannot get non-numeric property"));
1867                 return (zfs_error(zhp->zfs_hdl, EZFS_BADPROP,
1868                     dgettext(TEXT_DOMAIN, "internal error")));
1869         }
1870     }

1872     return (0);
1873 }

1875 /*
1876  * Calculate the source type, given the raw source string.
1877  */
1878 static void
1879 get_source(zfs_handle_t *zhp, zprop_source_t *srctype, char *source,
1880           char *statbuf, size_t statlen)
1881 {
1882     if (statbuf == NULL || *srctype == ZPROP_SRC_TEMPORARY)
1883         return;

1885     if (source == NULL) {
1886         *srctype = ZPROP_SRC_NONE;
1887     } else if (source[0] == '\0') {
1888         *srctype = ZPROP_SRC_DEFAULT;
1889     } else if (strstr(source, ZPROP_SOURCE_VAL_RECVD) != NULL) {
1890         *srctype = ZPROP_SRC_RECEIVED;
1891     } else {
1892         if (strcmp(source, zhp->zfs_name) == 0) {
1893             *srctype = ZPROP_SRC_LOCAL;
1894         } else {
1895             (void) strncpy(statbuf, source, statlen);
1896             *srctype = ZPROP_SRC_INHERITED;
1897         }
1898     }

1900 }

1902 int
1903 zfs_prop_get_recvd(zfs_handle_t *zhp, const char *propname, char *propbuf,
1904                  size_t proplen, boolean_t literal)
1905 {
1906     zfs_prop_t prop;
1907     int err = 0;

```

```

1909     if (zhp->zfs_recvd_props == NULL)
1910         if (get_recvd_props_ioctl(zhp) != 0)
1911             return (-1);

1913     prop = zfs_name_to_prop(propname);

1915     if (prop != ZPROP_INVALID) {
1916         uint64_t cookie;
1917         if (!nvlist_exists(zhp->zfs_recvd_props, propname))
1918             return (-1);
1919         zfs_set_recvd_props_mode(zhp, &cookie);
1920         err = zfs_prop_get(zhp, prop, propbuf, proplen,
1921             NULL, NULL, 0, literal);
1922         zfs_unset_recvd_props_mode(zhp, &cookie);
1923     } else {
1924         nvlist_t *propval;
1925         char *recvdval;
1926         if (nvlist_lookup_nvlist(zhp->zfs_recvd_props,
1927             propname, &propval) != 0)
1928             return (-1);
1929         verify(nvlist_lookup_string(propval, ZPROP_VALUE,
1930             &recvdval) == 0);
1931         (void) strncpy(propbuf, recvdval, proplen);
1932     }

1934     return (err == 0 ? 0 : -1);
1935 }

1937 static int
1938 get_clones_string(zfs_handle_t *zhp, char *propbuf, size_t proplen)
1939 {
1940     nvlist_t *value;
1941     nvpair_t *pair;

1943     value = zfs_get_clones_nvlist(zhp);
1944     if (value == NULL)
1945         return (-1);

1947     propbuf[0] = '\0';
1948     for (pair = nvlist_next_nvpair(value, NULL); pair != NULL;
1949         pair = nvlist_next_nvpair(value, pair)) {
1950         if (propbuf[0] != '\0')
1951             (void) strcat(propbuf, ",", proplen);
1952         (void) strcat(propbuf, nvpair_name(pair), proplen);
1953     }

1955     return (0);
1956 }

1958 struct get_clones_arg {
1959     uint64_t numclones;
1960     nvlist_t *value;
1961     const char *origin;
1962     char buf[ZFS_MAXNAMELEN];
1963 };

1965 int
1966 get_clones_cb(zfs_handle_t *zhp, void *arg)
1967 {
1968     struct get_clones_arg *gca = arg;

1970     if (gca->numclones == 0) {
1971         zfs_close(zhp);
1972         return (0);
1973     }

```

```

1975     if (zfs_prop_get(zhp, ZFS_PROP_ORIGIN, gca->buf, sizeof (gca->buf),
1976         NULL, NULL, 0, B_TRUE) != 0)
1977         goto out;
1978     if (strcmp(gca->buf, gca->origin) == 0) {
1979         fnvlist_add_boolean(gca->value, zfs_get_name(zhp));
1980         gca->numclones--;
1981     }
1982
1983 out:
1984     (void) zfs_iter_children(zhp, get_clones_cb, gca);
1985     zfs_close(zhp);
1986     return (0);
1987 }
1988
1989 nvlist_t *
1990 zfs_get_clones_nvlist(zfs_handle_t *zhp)
1991 {
1992     nvlist_t *nv, *value;
1993
1994     if (nvlist_lookup_nvlist(zhp->zfs_props,
1995         zfs_prop_to_name(ZFS_PROP_CLONES), &nv) != 0) {
1996         struct get_clones_arg gca;
1997
1998         /*
1999          * if this is a snapshot, then the kernel wasn't able
2000          * to get the clones. Do it by slowly iterating.
2001          */
2002         if (zhp->zfs_type != ZFS_TYPE_SNAPSHOT)
2003             return (NULL);
2004         if (nvlist_alloc(&nv, NV_UNIQUE_NAME, 0) != 0)
2005             return (NULL);
2006         if (nvlist_alloc(&value, NV_UNIQUE_NAME, 0) != 0) {
2007             nvlist_free(nv);
2008             return (NULL);
2009         }
2010
2011         gca.numclones = zfs_prop_get_int(zhp, ZFS_PROP_NUMCLONES);
2012         gca.value = value;
2013         gca.origin = zhp->zfs_name;
2014
2015         if (gca.numclones != 0) {
2016             zfs_handle_t *root;
2017             char pool[ZFS_MAXNAMELEN];
2018             char *cp = pool;
2019
2020             /* get the pool name */
2021             (void) strcpy(pool, zhp->zfs_name, sizeof (pool));
2022             (void) strsep(&cp, "@");
2023             root = zfs_open(zhp->zfs_hdl, pool,
2024                 ZFS_TYPE_FILESYSTEM);
2025
2026             (void) get_clones_cb(root, &gca);
2027         }
2028
2029         if (gca.numclones != 0 ||
2030             nvlist_add_nvlist(nv, ZPROP_VALUE, value) != 0 ||
2031             nvlist_add_nvlist(zhp->zfs_props,
2032                 zfs_prop_to_name(ZFS_PROP_CLONES), nv) != 0) {
2033             nvlist_free(nv);
2034             nvlist_free(value);
2035             return (NULL);
2036         }
2037         nvlist_free(nv);
2038         nvlist_free(value);
2039         verify(0 == nvlist_lookup_nvlist(zhp->zfs_props,
2040             zfs_prop_to_name(ZFS_PROP_CLONES), &nv));

```

```

2041     }
2042
2043     verify(nvlist_lookup_nvlist(nv, ZPROP_VALUE, &value) == 0);
2044
2045     return (value);
2046 }
2047
2048 /*
2049  * Retrieve a property from the given object. If 'literal' is specified, then
2050  * numbers are left as exact values. Otherwise, numbers are converted to a
2051  * human-readable form.
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);
2063
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);
2069
2070     if (received && zfs_prop_readonly(prop))
2071         return (-1);
2072
2073     if (src)
2074         *src = ZPROP_SRC_NONE;
2075
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;
2086
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;
2094
2095     case ZFS_PROP_MOUNTPOINT:
2096         /*
2097          * Getting the precise mountpoint can be tricky.
2098          * - for 'none' or 'legacy', return those values.
2099          * - for inherited mountpoints, we want to take everything
2100          *   after our ancestor and append it to the inherited value.
2101          * - If the pool has an alternate root, we want to prepend that
2102          *   root to any values we return.
2103          */

```

```

2108     str = getprop_string(zhp, prop, &source);
2110     if (str[0] == '/') {
2111         char buf[MAXPATHLEN];
2112         char *root = buf;
2113         const char *relpath;
2115         /*
2116          * If we inherit the mountpoint, even from a dataset
2117          * with a received value, the source will be the path of
2118          * the dataset we inherit from. If source is
2119          * ZPROP_SOURCE_VAL_RECVD, the received value is not
2120          * inherited.
2121          */
2122         if (strcmp(source, ZPROP_SOURCE_VAL_RECVD) == 0) {
2123             relpath = "";
2124         } else {
2125             relpath = zhp->zfs_name + strlen(source);
2126             if (relpath[0] == '/')
2127                 relpath++;
2128         }
2130         if ((zpool_get_prop(zhp->zpool_hdl,
2131             ZPOOL_PROP_ALTRoot, buf, MAXPATHLEN, NULL)) ||
2132             (strcmp(root, "-") == 0))
2133             root[0] = '\0';
2134         /*
2135          * Special case an alternate root of '/'. This will
2136          * avoid having multiple leading slashes in the
2137          * mountpoint path.
2138          */
2139         if (strcmp(root, "/") == 0)
2140             root++;
2142         /*
2143          * If the mountpoint is '/' then skip over this
2144          * if we are obtaining either an alternate root or
2145          * an inherited mountpoint.
2146          */
2147         if (str[1] == '\0' && (root[0] != '\0' ||
2148             relpath[0] != '\0'))
2149             str++;
2151         if (relpath[0] == '\0')
2152             (void) snprintf(propbuf, proplen, "%s%s",
2153                 root, str);
2154         else
2155             (void) snprintf(propbuf, proplen, "%s%s%s%s",
2156                 root, str, relpath[0] == '@' ? " " : "/",
2157                 relpath);
2158     } else {
2159         /* 'legacy' or 'none' */
2160         (void) strlcpy(propbuf, str, proplen);
2161     }
2163     break;
2165 case ZFS_PROP_ORIGIN:
2166     (void) strlcpy(propbuf, getprop_string(zhp, prop, &source),
2167         proplen);
2168     /*
2169      * If there is no parent at all, return failure to indicate that
2170      * it doesn't apply to this dataset.
2171      */
2172     if (propbuf[0] == '\0')

```

```

2173         return (-1);
2174     break;
2176 case ZFS_PROP_CLONES:
2177     if (get_clones_string(zhp, propbuf, proplen) != 0)
2178         return (-1);
2179     break;
2181 case ZFS_PROP_QUOTA:
2182 case ZFS_PROP_REFQUOTA:
2183 case ZFS_PROP_RESERVATION:
2184 case ZFS_PROP_REFRESERVATION:
2186     if (get_numeric_property(zhp, prop, src, &source, &val) != 0)
2187         return (-1);
2189     /*
2190      * If quota or reservation is 0, we translate this into 'none'
2191      * (unless literal is set), and indicate that it's the default
2192      * value. Otherwise, we print the number nicely and indicate
2193      * that its set locally.
2194      */
2195     if (val == 0) {
2196         if (literal)
2197             (void) strlcpy(propbuf, "0", proplen);
2198         else
2199             (void) strlcpy(propbuf, "none", proplen);
2200     } else {
2201         if (literal)
2202             (void) snprintf(propbuf, proplen, "%llu",
2203                 (u_longlong_t)val);
2204         else
2205             zfs_nicenum(val, propbuf, proplen);
2206     }
2207     break;
2209 case ZFS_PROP_REFRATIO:
2210 case ZFS_PROP_COMPRESSRATIO:
2211     if (get_numeric_property(zhp, prop, src, &source, &val) != 0)
2212         return (-1);
2213     (void) snprintf(propbuf, proplen, "%llu.%02lux",
2214         (u_longlong_t)(val / 100),
2215         (u_longlong_t)(val % 100));
2216     break;
2218 case ZFS_PROP_TYPE:
2219     switch (zhp->zfs_type) {
2220     case ZFS_TYPE_FILESYSTEM:
2221         str = "filesystem";
2222         break;
2223     case ZFS_TYPE_VOLUME:
2224         str = "volume";
2225         break;
2226     case ZFS_TYPE_SNAPSHOT:
2227         str = "snapshot";
2228         break;
2229     default:
2230         abort();
2231     }
2232     (void) snprintf(propbuf, proplen, "%s", str);
2233     break;
2235 case ZFS_PROP_MOUNTED:
2236     /*
2237      * The 'mounted' property is a pseudo-property that described
2238      * whether the filesystem is currently mounted. Even though

```



```

2239     * it's a boolean value, the typical values of "on" and "off"
2240     * don't make sense, so we translate to "yes" and "no".
2241     */
2242     if (get_numeric_property(zhp, ZFS_PROP_MOUNTED,
2243         src, &source, &val) != 0)
2244         return (-1);
2245     if (val)
2246         (void) strlcpy(propbuf, "yes", proplen);
2247     else
2248         (void) strlcpy(propbuf, "no", proplen);
2249     break;

2251 case ZFS_PROP_NAME:
2252     /*
2253     * The 'name' property is a pseudo-property derived from the
2254     * dataset name. It is presented as a real property to simplify
2255     * consumers.
2256     */
2257     (void) strlcpy(propbuf, zhp->zfs_name, proplen);
2258     break;

2260 case ZFS_PROP_MLSLABEL:
2261     {
2262         m_label_t *new_sl = NULL;
2263         char *ascii = NULL; /* human readable label */

2265         (void) strlcpy(propbuf,
2266             getprop_string(zhp, prop, &source), proplen);

2268         if (literal || (strcascmp(propbuf,
2269             ZFS_MLSLABEL_DEFAULT) == 0))
2270             break;

2272         /*
2273         * Try to translate the internal hex string to
2274         * human-readable output. If there are any
2275         * problems just use the hex string.
2276         */

2278         if (str_to_label(propbuf, &new_sl, MAC_LABEL,
2279             L_NO_CORRECTION, NULL) == -1) {
2280             m_label_free(new_sl);
2281             break;
2282         }

2284         if (label_to_str(new_sl, &ascii, M_LABEL,
2285             DEF_NAMES) != 0) {
2286             if (ascii)
2287                 free(ascii);
2288             m_label_free(new_sl);
2289             break;
2290         }
2291         m_label_free(new_sl);

2293         (void) strlcpy(propbuf, ascii, proplen);
2294         free(ascii);
2295     }
2296     break;

2298 case ZFS_PROP_GUID:
2299     /*
2300     * GUIDs are stored as numbers, but they are identifiers.
2301     * We don't want them to be pretty printed, because pretty
2302     * printing mangles the ID into a truncated and useless value.
2303     */
2304     if (get_numeric_property(zhp, prop, src, &source, &val) != 0)

```

```

2305         return (-1);
2306         (void) snprintf(propbuf, proplen, "%llu", (u_longlong_t)val);
2307         break;

2309     default:
2310         switch (zfs_prop_get_type(prop)) {
2311             case PROP_TYPE_NUMBER:
2312                 if (get_numeric_property(zhp, prop, src,
2313                     &source, &val) != 0)
2314                     return (-1);
2315                 if (literal)
2316                     (void) snprintf(propbuf, proplen, "%llu",
2317                         (u_longlong_t)val);
2318                 else
2319                     zfs_nicenum(val, propbuf, proplen);
2320                 break;

2322             case PROP_TYPE_STRING:
2323                 (void) strlcpy(propbuf,
2324                     getprop_string(zhp, prop, &source), proplen);
2325                 break;

2327             case PROP_TYPE_INDEX:
2328                 if (get_numeric_property(zhp, prop, src,
2329                     &source, &val) != 0)
2330                     return (-1);
2331                 if (zfs_prop_index_to_string(prop, val, &strval) != 0)
2332                     return (-1);
2333                 (void) strlcpy(propbuf, strval, proplen);
2334                 break;

2336             default:
2337                 abort();
2338         }
2339     }

2341     get_source(zhp, src, source, statbuf, statlen);

2343     return (0);
2344 }

2346 /*
2347  * Utility function to get the given numeric property. Does no validation that
2348  * the given property is the appropriate type; should only be used with
2349  * hard-coded property types.
2350  */
2351 uint64_t
2352 zfs_prop_get_int(zfs_handle_t *zhp, zfs_prop_t prop)
2353 {
2354     char *source;
2355     uint64_t val;

2357     (void) get_numeric_property(zhp, prop, NULL, &source, &val);

2359     return (val);
2360 }

2362 int
2363 zfs_prop_set_int(zfs_handle_t *zhp, zfs_prop_t prop, uint64_t val)
2364 {
2365     char buf[64];

2367     (void) snprintf(buf, sizeof (buf), "%llu", (longlong_t)val);
2368     return (zfs_prop_set(zhp, zfs_prop_to_name(prop), buf));
2369 }

```

```

2371 /*
2372  * Similar to zfs_prop_get(), but returns the value as an integer.
2373  */
2374 int
2375 zfs_prop_get_numeric(zfs_handle_t *zhp, zfs_prop_t prop, uint64_t *value,
2376     zprop_source_t *src, char *statbuf, size_t statlen)
2377 {
2378     char *source;
2379
2380     /*
2381      * Check to see if this property applies to our object
2382      */
2383     if (!zfs_prop_valid_for_type(prop, zhp->zfs_type)) {
2384         return (zfs_error_fmt(zhp->zfs_hdl, EZFS_PROPTYPE,
2385             dgettext(TEXT_DOMAIN, "cannot get property '%s'",
2386                 zfs_prop_to_name(prop))));
2387     }
2388
2389     if (src)
2390         *src = ZPROP_SRC_NONE;
2391
2392     if (get_numeric_property(zhp, prop, src, &source, value) != 0)
2393         return (-1);
2394
2395     get_source(zhp, src, source, statbuf, statlen);
2396
2397     return (0);
2398 }
2399
2400 static int
2401 idmap_id_to_numeric_domain_rid(uid_t id, boolean_t isuser,
2402     char **domainp, idmap_rid_t *ridp)
2403 {
2404     idmap_get_handle_t *get_hdl = NULL;
2405     idmap_stat status;
2406     int err = EINVAL;
2407
2408     if (idmap_get_create(&get_hdl) != IDMAP_SUCCESS)
2409         goto out;
2410
2411     if (isuser) {
2412         err = idmap_get_sidbyuid(get_hdl, id,
2413             IDMAP_REQ_FLG_USE_CACHE, domainp, ridp, &status);
2414     } else {
2415         err = idmap_get_sidbygid(get_hdl, id,
2416             IDMAP_REQ_FLG_USE_CACHE, domainp, ridp, &status);
2417     }
2418     if (err == IDMAP_SUCCESS &&
2419         idmap_get_mappings(get_hdl) == IDMAP_SUCCESS &&
2420         status == IDMAP_SUCCESS)
2421         err = 0;
2422     else
2423         err = EINVAL;
2424 out:
2425     if (get_hdl)
2426         idmap_get_destroy(get_hdl);
2427     return (err);
2428 }
2429
2430 /*
2431  * convert the propname into parameters needed by kernel
2432  * Eg: userquota@ahrens -> ZFS_PROP_USERQUOTA, "", 126829
2433  * Eg: userused@matt@domain -> ZFS_PROP_USERUSED, "S-1-123-456", 789
2434  */
2435 static int
2436 userquota_propname_decode(const char *propname, boolean_t zoned,

```

```

2437     zfs_userquota_prop_t *typep, char *domain, int domainlen, uint64_t *ridp)
2438 {
2439     zfs_userquota_prop_t type;
2440     char *cp, *end;
2441     char *numeric_sid = NULL;
2442     boolean_t isuser;
2443
2444     domain[0] = '\0';
2445
2446     /* Figure out the property type ({user|group}{quota|space}) */
2447     for (type = 0; type < ZFS_NUM_USERQUOTA_PROPS; type++) {
2448         if (strncmp(propname, zfs_userquota_prop_prefixes[type],
2449             strlen(zfs_userquota_prop_prefixes[type])) == 0)
2450             break;
2451     }
2452     if (type == ZFS_NUM_USERQUOTA_PROPS)
2453         return (EINVAL);
2454     *typep = type;
2455
2456     isuser = (type == ZFS_PROP_USERQUOTA ||
2457         type == ZFS_PROP_USERUSED);
2458
2459     cp = strchr(propname, '@') + 1;
2460
2461     if (strchr(cp, '@')) {
2462         /*
2463          * It's a SID name (eg "user@domain") that needs to be
2464          * turned into S-1-domainID-RID.
2465          */
2466         directory_error_t e;
2467         if (zoned && getzoneid() == GLOBAL_ZONEID)
2468             return (ENOENT);
2469         if (isuser) {
2470             e = directory_sid_from_user_name(NULL,
2471                 cp, &numeric_sid);
2472         } else {
2473             e = directory_sid_from_group_name(NULL,
2474                 cp, &numeric_sid);
2475         }
2476         if (e != NULL) {
2477             directory_error_free(e);
2478             return (ENOENT);
2479         }
2480         if (numeric_sid == NULL)
2481             return (ENOENT);
2482         cp = numeric_sid;
2483         /* will be further decoded below */
2484     }
2485
2486     if (strncmp(cp, "S-1-", 4) == 0) {
2487         /* It's a numeric SID (eg "S-1-234-567-89") */
2488         (void) strncpy(domain, cp, domainlen);
2489         cp = strchr(domain, '-');
2490         *cp = '\0';
2491         cp++;
2492
2493         errno = 0;
2494         *ridp = strtoull(cp, &end, 10);
2495         if (numeric_sid) {
2496             free(numeric_sid);
2497             numeric_sid = NULL;
2498         }
2499         if (errno != 0 || *end != '\0')
2500             return (EINVAL);
2501     } else if (!isdigit(*cp)) {
2502         /*

```

```

2503     * It's a user/group name (eg "user") that needs to be
2504     * turned into a uid/gid
2505     */
2506     if (zoned && getzoneid() == GLOBAL_ZONEID)
2507         return (ENOENT);
2508     if (isuser) {
2509         struct passwd *pw;
2510         pw = getpwnam(cp);
2511         if (pw == NULL)
2512             return (ENOENT);
2513         *ridp = pw->pw_uid;
2514     } else {
2515         struct group *gr;
2516         gr = getgrnam(cp);
2517         if (gr == NULL)
2518             return (ENOENT);
2519         *ridp = gr->gr_gid;
2520     }
2521 } else {
2522     /* It's a user/group ID (eg "12345"). */
2523     uid_t id = strtoul(cp, &end, 10);
2524     idmap_rid_t rid;
2525     char *mapdomain;
2527     if (*end != '\0')
2528         return (EINVAL);
2529     if (id > MAXUID) {
2530         /* It's an ephemeral ID. */
2531         if (idmap_id_to_numeric_domain_rid(id, isuser,
2532             &mapdomain, &rid) != 0)
2533             return (ENOENT);
2534         (void) strlcpy(domain, mapdomain, domainlen);
2535         *ridp = rid;
2536     } else {
2537         *ridp = id;
2538     }
2539 }
2541     ASSERT3P(numericid, ==, NULL);
2542     return (0);
2543 }
2545 static int
2546 zfs_prop_get_userquota_common(zfs_handle_t *zhp, const char *propname,
2547     uint64_t *propvalue, zfs_userquota_prop_t *typep)
2548 {
2549     int err;
2550     zfs_cmd_t zc = { 0 };
2552     (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
2554     err = userquota_propname_decode(propname,
2555         zfs_prop_get_int(zhp, ZFS_PROP_ZONED),
2556         typep, zc.zc_value, sizeof (zc.zc_value), &zc.zc_guid);
2557     zc.zc_objset_type = *typep;
2558     if (err)
2559         return (err);
2561     err = ioctl(zhp->zfs_hdl->libzfs_fd, ZFS_IOC_USERSPACE_ONE, &zc);
2562     if (err)
2563         return (err);
2565     *propvalue = zc.zc_cookie;
2566     return (0);
2567 }

```

```

2569 int
2570 zfs_prop_get_userquota_int(zfs_handle_t *zhp, const char *propname,
2571     uint64_t *propvalue)
2572 {
2573     zfs_userquota_prop_t type;
2575     return (zfs_prop_get_userquota_common(zhp, propname, propvalue,
2576         &type));
2577 }
2579 int
2580 zfs_prop_get_userquota(zfs_handle_t *zhp, const char *propname,
2581     char *propbuf, int proplen, boolean_t literal)
2582 {
2583     int err;
2584     uint64_t propvalue;
2585     zfs_userquota_prop_t type;
2587     err = zfs_prop_get_userquota_common(zhp, propname, &propvalue,
2588         &type);
2590     if (err)
2591         return (err);
2593     if (literal) {
2594         (void) snprintf(propbuf, proplen, "%llu", propvalue);
2595     } else if (propvalue == 0 &&
2596         (type == ZFS_PROP_USERQUOTA || type == ZFS_PROP_GROUPQUOTA)) {
2597         (void) strlcpy(propbuf, "none", proplen);
2598     } else {
2599         zfs_nicenum(propvalue, propbuf, proplen);
2600     }
2601     return (0);
2602 }
2604 int
2605 zfs_prop_get_written_int(zfs_handle_t *zhp, const char *propname,
2606     uint64_t *propvalue)
2607 {
2608     int err;
2609     zfs_cmd_t zc = { 0 };
2610     const char *snapname;
2612     (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
2614     snapname = strchr(propname, '@') + 1;
2615     if (strchr(snapname, '@')) {
2616         (void) strlcpy(zc.zc_value, snapname, sizeof (zc.zc_value));
2617     } else {
2618         /* snapname is the short name, append it to zhp's fsname */
2619         char *cp;
2621         (void) strlcpy(zc.zc_value, zhp->zfs_name,
2622             sizeof (zc.zc_value));
2623         cp = strchr(zc.zc_value, '@');
2624         if (cp != NULL)
2625             *cp = '\0';
2626         (void) strlcat(zc.zc_value, "@", sizeof (zc.zc_value));
2627         (void) strlcat(zc.zc_value, snapname, sizeof (zc.zc_value));
2628     }
2630     err = ioctl(zhp->zfs_hdl->libzfs_fd, ZFS_IOC_SPACE_WRITTEN, &zc);
2631     if (err)
2632         return (err);
2634     *propvalue = zc.zc_cookie;

```

```

2635     return (0);
2636 }

2638 int
2639 zfs_prop_get_written(zfs_handle_t *zhp, const char *propname,
2640     char *propbuf, int proplen, boolean_t literal)
2641 {
2642     int err;
2643     uint64_t propvalue;

2645     err = zfs_prop_get_written_int(zhp, propname, &propvalue);

2647     if (err)
2648         return (err);

2650     if (literal) {
2651         (void) snprintf(propbuf, proplen, "%llu", propvalue);
2652     } else {
2653         zfs_nicenum(propvalue, propbuf, proplen);
2654     }
2655     return (0);
2656 }

2658 /*
2659  * Returns the name of the given zfs handle.
2660  */
2661 const char *
2662 zfs_get_name(const zfs_handle_t *zhp)
2663 {
2664     return (zhp->zfs_name);
2665 }

2667 /*
2668  * Returns the type of the given zfs handle.
2669  */
2670 zfs_type_t
2671 zfs_get_type(const zfs_handle_t *zhp)
2672 {
2673     return (zhp->zfs_type);
2674 }

2676 /*
2677  * Is one dataset name a child dataset of another?
2678  */
2679 * Needs to handle these cases:
2680 * Dataset 1   "a/foo"           "a/foo"           "a/foo"           "a/foo"
2681 * Dataset 2   "a/fo"            "a/foobar"        "a/bar/baz"       "a/foo/bar"
2682 * Descendant? No.               No.               No.               Yes.
2683 */
2684 static boolean_t
2685 is_descendant(const char *ds1, const char *ds2)
2686 {
2687     size_t d1len = strlen(ds1);

2689     /* ds2 can't be a descendant if it's smaller */
2690     if (strlen(ds2) < d1len)
2691         return (B_FALSE);

2693     /* otherwise, compare strings and verify that there's a '/' char */
2694     return (ds2[d1len] == '/' && (strncmp(ds1, ds2, d1len) == 0));
2695 }

2697 /*
2698  * Given a complete name, return just the portion that refers to the parent.
2699  * Will return -1 if there is no parent (path is just the name of the
2700  * pool).

```

```

2701  */
2702 static int
2703 parent_name(const char *path, char *buf, size_t buflen)
2704 {
2705     char *slashp;

2707     (void) strncpy(buf, path, buflen);

2709     if ((slashp = strrchr(buf, '/')) == NULL)
2710         return (-1);
2711     *slashp = '\0';

2713     return (0);
2714 }

2716 /*
2717  * If accept_ancestor is false, then check to make sure that the given path has
2718  * a parent, and that it exists. If accept_ancestor is true, then find the
2719  * closest existing ancestor for the given path. In prefixlen return the
2720  * length of already existing prefix of the given path. We also fetch the
2721  * 'zoned' property, which is used to validate property settings when creating
2722  * new datasets.
2723  */
2724 static int
2725 check_parents(libzfs_handle_t *hdl, const char *path, uint64_t *zoned,
2726     boolean_t accept_ancestor, int *prefixlen)
2727 {
2728     zfs_cmd_t zc = { 0 };
2729     char parent[ZFS_MAXNAMELEN];
2730     char *slash;
2731     zfs_handle_t *zhp;
2732     char errbuf[1024];
2733     uint64_t is_zoned;

2735     (void) snprintf(errbuf, sizeof (errbuf),
2736         dgettext(TEXT_DOMAIN, "cannot create '%s'"), path);

2738     /* get parent, and check to see if this is just a pool */
2739     if (parent_name(path, parent, sizeof (parent)) != 0) {
2740         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2741             "missing dataset name"));
2742         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2743     }

2745     /* check to see if the pool exists */
2746     if ((slash = strchr(parent, '/')) == NULL)
2747         slash = parent + strlen(parent);
2748     (void) strncpy(zc.zc_name, parent, slash - parent);
2749     zc.zc_name[slash - parent] = '\0';
2750     if (ioctl(hdl->libzfs_fd, ZFS_IOC_OBJSET_STATS, &zc) != 0 &&
2751         errno == ENOENT) {
2752         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2753             "no such pool '%s'"), zc.zc_name);
2754         return (zfs_error(hdl, EZFS_NOENT, errbuf));
2755     }

2757     /* check to see if the parent dataset exists */
2758     while ((zhp = make_dataset_handle(hdl, parent)) == NULL) {
2759         if (errno == ENOENT && accept_ancestor) {
2760             /*
2761              * Go deeper to find an ancestor, give up on top level.
2762              */
2763             if (parent_name(parent, parent, sizeof (parent)) != 0) {
2764                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2765                     "no such pool '%s'"), zc.zc_name);
2766                 return (zfs_error(hdl, EZFS_NOENT, errbuf));

```

```

2767     }
2768     } else if (errno == ENOENT) {
2769         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2770             "parent does not exist"));
2771         return (zfs_error(hdl, EZFS_NOENT, errbuf));
2772     } else
2773         return (zfs_standard_error(hdl, errno, errbuf));
2774 }
2775
2776 is_zoned = zfs_prop_get_int(zhp, ZFS_PROP_ZONED);
2777 if (zoned != NULL)
2778     *zoned = is_zoned;
2779
2780 /* we are in a non-global zone, but parent is in the global zone */
2781 if (getzoneid() != GLOBAL_ZONEID && !is_zoned) {
2782     (void) zfs_standard_error(hdl, EPERM, errbuf);
2783     zfs_close(zhp);
2784     return (-1);
2785 }
2786
2787 /* make sure parent is a filesystem */
2788 if (zfs_get_type(zhp) != ZFS_TYPE_FILESYSTEM) {
2789     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2790         "parent is not a filesystem"));
2791     (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
2792     zfs_close(zhp);
2793     return (-1);
2794 }
2795
2796 zfs_close(zhp);
2797 if (prefixlen != NULL)
2798     *prefixlen = strlen(parent);
2799 return (0);
2800 }
2801
2802 /*
2803  * Finds whether the dataset of the given type(s) exists.
2804  */
2805 boolean_t
2806 zfs_dataset_exists(libzfs_handle_t *hdl, const char *path, zfs_type_t types)
2807 {
2808     zfs_handle_t *zhp;
2809
2810     if (!zfs_validate_name(hdl, path, types, B_FALSE))
2811         return (B_FALSE);
2812
2813     /*
2814      * Try to get stats for the dataset, which will tell us if it exists.
2815      */
2816     if ((zhp = make_dataset_handle(hdl, path)) != NULL) {
2817         int ds_type = zhp->zfs_type;
2818
2819         zfs_close(zhp);
2820         if (types & ds_type)
2821             return (B_TRUE);
2822     }
2823     return (B_FALSE);
2824 }
2825
2826 /*
2827  * Given a path to 'target', create all the ancestors between
2828  * the prefixlen portion of the path, and the target itself.
2829  * Fail if the initial prefixlen-ancestor does not already exist.
2830  */
2831 int
2832 create_parents(libzfs_handle_t *hdl, char *target, int prefixlen)

```

```

2833 {
2834     zfs_handle_t *h;
2835     char *cp;
2836     const char *opname;
2837
2838     /* make sure prefix exists */
2839     cp = target + prefixlen;
2840     if (*cp != '/') {
2841         assert(strchr(cp, '/') == NULL);
2842         h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
2843     } else {
2844         *cp = '\0';
2845         h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
2846         *cp = '/';
2847     }
2848     if (h == NULL)
2849         return (-1);
2850     zfs_close(h);
2851
2852     /*
2853      * Attempt to create, mount, and share any ancestor filesystems,
2854      * up to the prefixlen-long one.
2855      */
2856     for (cp = target + prefixlen + 1;
2857          cp = strchr(cp, '/'); *cp = '/', cp++) {
2858
2859         *cp = '\0';
2860
2861         h = make_dataset_handle(hdl, target);
2862         if (h) {
2863             /* it already exists, nothing to do here */
2864             zfs_close(h);
2865             continue;
2866         }
2867
2868         if (zfs_create(hdl, target, ZFS_TYPE_FILESYSTEM,
2869             NULL) != 0) {
2870             opname = dgettext(TEXT_DOMAIN, "create");
2871             goto ancestorerr;
2872         }
2873
2874         h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
2875         if (h == NULL) {
2876             opname = dgettext(TEXT_DOMAIN, "open");
2877             goto ancestorerr;
2878         }
2879
2880         if (zfs_mount(h, NULL, 0) != 0) {
2881             opname = dgettext(TEXT_DOMAIN, "mount");
2882             goto ancestorerr;
2883         }
2884
2885         if (zfs_share(h) != 0) {
2886             opname = dgettext(TEXT_DOMAIN, "share");
2887             goto ancestorerr;
2888         }
2889
2890         zfs_close(h);
2891     }
2892
2893     return (0);
2894
2895 ancestorerr:
2896     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2897         "failed to %s ancestor '%s'", opname, target));
2898     return (-1);

```

```

2899 }

2901 /*
2902  * Creates non-existing ancestors of the given path.
2903  */
2904 int
2905 zfs_create_ancestors(libzfs_handle_t *hdl, const char *path)
2906 {
2907     int prefix;
2908     char *path_copy;
2909     int rc;

2911     if (check_parents(hdl, path, NULL, B_TRUE, &prefix) != 0)
2912         return (-1);

2914     if ((path_copy = strdup(path)) != NULL) {
2915         rc = create_parents(hdl, path_copy, prefix);
2916         free(path_copy);
2917     }
2918     if (path_copy == NULL || rc != 0)
2919         return (-1);

2921     return (0);
2922 }

2924 /*
2925  * Create a new filesystem or volume.
2926  */
2927 int
2928 zfs_create(libzfs_handle_t *hdl, const char *path, zfs_type_t type,
2929           nvlist_t *props)
2930 {
2931     int ret;
2932     uint64_t size = 0;
2933     uint64_t blocksize = zfs_prop_default_numeric(ZFS_PROP_VOLBLOCKSIZE);
2934     char errbuf[1024];
2935     uint64_t zoned;
2936     dmu_objset_type_t ost;

2938     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2939 "cannot create '%s'", path);

2941     /* validate the path, taking care to note the extended error message */
2942     if (!zfs_validate_name(hdl, path, type, B_TRUE))
2943         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));

2945     /* validate parents exist */
2946     if (check_parents(hdl, path, &zoned, B_FALSE, NULL) != 0)
2947         return (-1);

2949     /*
2950      * The failure modes when creating a dataset of a different type over
2951      * one that already exists is a little strange. In particular, if you
2952      * try to create a dataset on top of an existing dataset, the ioctl()
2953      * will return ENOENT, not EEXIST. To prevent this from happening, we
2954      * first try to see if the dataset exists.
2955      */
2956     if (zfs_dataset_exists(hdl, path, ZFS_TYPE_DATASET)) {
2957         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2958 "dataset already exists"));
2959         return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2960     }

2962     if (type == ZFS_TYPE_VOLUME)
2963         ost = DMU_OST_ZVOL;
2964     else

```

```

2965         ost = DMU_OST_ZFS;

2967     if (props && (props = zfs_valid_proplist(hdl, type, props,
2968 zoned, NULL, errbuf)) == 0)
2969         return (-1);

2971     if (type == ZFS_TYPE_VOLUME) {
2972         /*
2973          * If we are creating a volume, the size and block size must
2974          * satisfy a few restraints. First, the blocksize must be a
2975          * valid block size between SPA_{MIN,MAX}BLOCKSIZE. Second, the
2976          * volsize must be a multiple of the block size, and cannot be
2977          * zero.
2978          */
2979         if (props == NULL || nvlist_lookup_uint64(props,
2980 zfs_prop_to_name(ZFS_PROP_VOLSIZE), &size) != 0) {
2981             nvlist_free(props);
2982             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2983 "missing volume size"));
2984             return (zfs_error(hdl, EZFS_BADPROP, errbuf));
2985         }

2987         if ((ret = nvlist_lookup_uint64(props,
2988 zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
2989 &blocksize) != 0) {
2990             if (ret == ENOENT) {
2991                 blocksize = zfs_prop_default_numeric(
2992 ZFS_PROP_VOLBLOCKSIZE);
2993             } else {
2994                 nvlist_free(props);
2995                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2996 "missing volume block size"));
2997                 return (zfs_error(hdl, EZFS_BADPROP, errbuf));
2998             }
2999         }

3001         if (size == 0) {
3002             nvlist_free(props);
3003             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3004 "volume size cannot be zero"));
3005             return (zfs_error(hdl, EZFS_BADPROP, errbuf));
3006         }

3008         if (size % blocksize != 0) {
3009             nvlist_free(props);
3010             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3011 "volume size must be a multiple of volume block "
3012 "size"));
3013             return (zfs_error(hdl, EZFS_BADPROP, errbuf));
3014         }
3015     }

3017     /* create the dataset */
3018     ret = lz_create(path, ost, props);
3019     nvlist_free(props);

3021     /* check for failure */
3022     if (ret != 0) {
3023         char parent[ZFS_MAXNAMELEN];
3024         (void) parent_name(path, parent, sizeof (parent));

3026         switch (errno) {
3027             case ENOENT:
3028                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3029 "no such parent '%s'", parent);
3030                 return (zfs_error(hdl, EZFS_NOENT, errbuf));

```

```

3032     case EINVAL:
3033         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3034             "parent '%s' is not a filesystem"), parent);
3035         return (zfs_error(hdl, EZFS_BADTYPE, errbuf));

3037     case EDOM:
3038         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3039             "volume block size must be power of 2 from "
3040             "%u to %uk"),
3041             (uint_t)SPA_MINBLOCKSIZE,
3042             (uint_t)SPA_MAXBLOCKSIZE >> 10);

3044         return (zfs_error(hdl, EZFS_BADPROP, errbuf));

3046     case ENOTSUP:
3047         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3048             "pool must be upgraded to set this "
3049             "property or value"));
3050         return (zfs_error(hdl, EZFS_BADVERSION, errbuf));
3051 #ifdef _ILP32
3052     case EOVERFLOW:
3053         /*
3054          * This platform can't address a volume this big.
3055          */
3056         if (type == ZFS_TYPE_VOLUME)
3057             return (zfs_error(hdl, EZFS_VOLTOOBIG,
3058                 errbuf));
3059 #endif
3060         /* FALLTHROUGH */
3061     default:
3062         return (zfs_standard_error(hdl, errno, errbuf));
3063 }
3064 }

3066 return (0);
3067 }

3069 /*
3070  * Destroys the given dataset. The caller must make sure that the filesystem
3071  * isn't mounted, and that there are no active dependents. If the file system
3072  * does not exist this function does nothing.
3073  */
3074 int
3075 zfs_destroy(zfs_handle_t *zhp, boolean_t defer)
3076 {
3077     zfs_cmd_t zc = { 0 };

3079     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

3081     if (ZFS_IS_VOLUME(zhp)) {
3082         zc.zc_objset_type = DMU_OST_ZVOL;
3083     } else {
3084         zc.zc_objset_type = DMU_OST_ZFS;
3085     }

3087     zc.zc_defer_destroy = defer;
3088     if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_DESTROY, &zc) != 0 &&
3089         errno != ENOENT) {
3090         return (zfs_standard_error_fmt(zhp->zfs_hdl, errno,
3091             dgettext(TEXT_DOMAIN, "cannot destroy '%s'"),
3092             zhp->zfs_name));
3093     }

3095     remove_mountpoint(zhp);

```

```

3097         return (0);
3098     }

3100 struct destroydata {
3101     nvlist_t *nvl;
3102     const char *snapname;
3103 };

3105 static int
3106 zfs_check_snap_cb(zfs_handle_t *zhp, void *arg)
3107 {
3108     struct destroydata *dd = arg;
3109     zfs_handle_t *szhp;
3110     char name[ZFS_MAXNAMELEN];
3111     int rv = 0;

3113     (void) snprintf(name, sizeof (name),
3114         "%s@%s", zhp->zfs_name, dd->snapname);

3116     szhp = make_dataset_handle(zhp->zfs_hdl, name);
3117     if (szhp) {
3118         verify(nvlist_add_boolean(dd->nvl, name) == 0);
3119         zfs_close(szhp);
3120     }

3122     rv = zfs_iter_filesystems(zhp, zfs_check_snap_cb, dd);
3123     zfs_close(zhp);
3124     return (rv);
3125 }

3127 /*
3128  * Destroys all snapshots with the given name in zhp & descendants.
3129  */
3130 int
3131 zfs_destroy_snaps(zfs_handle_t *zhp, char *snapname, boolean_t defer)
3132 {
3133     int ret;
3134     struct destroydata dd = { 0 };

3136     dd.snapname = snapname;
3137     verify(nvlist_alloc(&dd.nvl, NV_UNIQUE_NAME, 0) == 0);
3138     (void) zfs_check_snap_cb(zhp->zfs_hdl, &dd);

3140     if (nvlist_next_nvpair(dd.nvl, NULL) == NULL) {
3141         ret = zfs_standard_error_fmt(zhp->zfs_hdl, ENOENT,
3142             dgettext(TEXT_DOMAIN, "cannot destroy '%s@%s'"),
3143             zhp->zfs_name, snapname);
3144     } else {
3145         ret = zfs_destroy_snaps_nvl(zhp->zfs_hdl, dd.nvl, defer);
3146     }
3147     nvlist_free(dd.nvl);
3148     return (ret);
3149 }

3151 /*
3152  * Destroys all the snapshots named in the nvlist.
3153  */
3154 int
3155 zfs_destroy_snaps_nvl(libzfs_handle_t *hdl, nvlist_t *snaps, boolean_t defer)
3156 {
3157     int ret;
3158     nvlist_t *errlist;

3160     ret = lzc_destroy_snaps(snaps, defer, &errlist);
3162     if (ret == 0)

```

```

3163         return (0);
3165     if (nvlist_next_nvpair(errlist, NULL) == NULL) {
3166         char errbuf[1024];
3167         (void) snprintf(errbuf, sizeof(errbuf),
3168             dgettext(TEXT_DOMAIN, "cannot destroy snapshots"));
3170         ret = zfs_standard_error(hdl, ret, errbuf);
3171     }
3172     for (nvpair_t *pair = nvlist_next_nvpair(errlist, NULL);
3173          pair != NULL; pair = nvlist_next_nvpair(errlist, pair)) {
3174         char errbuf[1024];
3175         (void) snprintf(errbuf, sizeof(errbuf),
3176             dgettext(TEXT_DOMAIN, "cannot destroy snapshot %s"),
3177             nvpair_name(pair));
3179         switch (fnvpair_value_int32(pair)) {
3180         case EEXIST:
3181             zfs_error_aux(hdl,
3182                 dgettext(TEXT_DOMAIN, "snapshot is cloned"));
3183             ret = zfs_error(hdl, EZFS_EXISTS, errbuf);
3184             break;
3185         default:
3186             ret = zfs_standard_error(hdl, errno, errbuf);
3187             break;
3188         }
3189     }
3191     return (ret);
3192 }
3194 /*
3195  * Clones the given dataset.  The target must be of the same type as the source.
3196  */
3197 int
3198 zfs_clone(zfs_handle_t *zhp, const char *target, nvlist_t *props)
3199 {
3200     char parent[ZFS_MAXNAMELEN];
3201     int ret;
3202     char errbuf[1024];
3203     libzfs_handle_t *hdl = zhp->zfs_hdl;
3204     uint64_t zoned;
3206     assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
3208     (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
3209         "cannot create '%s'"), target);
3211     /* validate the target/clone name */
3212     if (!zfs_validate_name(hdl, target, ZFS_TYPE_FILESYSTEM, B_TRUE))
3213         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3215     /* validate parents exist */
3216     if (check_parents(hdl, target, &zoned, B_FALSE, NULL) != 0)
3217         return (-1);
3219     (void) parent_name(target, parent, sizeof(parent));
3221     /* do the clone */
3223     if (props) {
3224         zfs_type_t type;
3225         if (ZFS_IS_VOLUME(zhp)) {
3226             type = ZFS_TYPE_VOLUME;
3227         } else {
3228             type = ZFS_TYPE_FILESYSTEM;

```

```

3229     }
3230     if ((props = zfs_valid_proplist(hdl, type, props, zoned,
3231         zhp, errbuf)) == NULL)
3232         return (-1);
3233     }
3235     ret = lzc_clone(target, zhp->zfs_name, props);
3236     nvlist_free(props);
3238     if (ret != 0) {
3239         switch (errno) {
3241         case ENOENT:
3242             /*
3243              * The parent doesn't exist.  We should have caught this
3244              * above, but there may a race condition that has since
3245              * destroyed the parent.
3246              *
3247              * At this point, we don't know whether it's the source
3248              * that doesn't exist anymore, or whether the target
3249              * dataset doesn't exist.
3250              */
3251             zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
3252                 "no such parent '%s'"), parent);
3253             return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
3255         case EXDEV:
3256             zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
3257                 "source and target pools differ"));
3258             return (zfs_error(zhp->zfs_hdl, EZFS_CROSTARGET,
3259                 errbuf));
3261         default:
3262             return (zfs_standard_error(zhp->zfs_hdl, errno,
3263                 errbuf));
3264         }
3265     }
3267     return (ret);
3268 }
3270 /*
3271  * Promotes the given clone fs to be the clone parent.
3272  */
3273 int
3274 zfs_promote(zfs_handle_t *zhp)
3275 {
3276     libzfs_handle_t *hdl = zhp->zfs_hdl;
3277     zfs_cmd_t zc = { 0 };
3278     char parent[MAXPATHLEN];
3279     int ret;
3280     char errbuf[1024];
3282     (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
3283         "cannot promote '%s'"), zhp->zfs_name);
3285     if (zhp->zfs_type == ZFS_TYPE_SNAPSHOT) {
3286         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3287             "snapshots can not be promoted"));
3288         return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
3289     }
3291     (void) strncpy(parent, zhp->zfs_dmstats.dds_origin, sizeof(parent));
3292     if (parent[0] == '\0') {
3293         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3294             "not a cloned filesystem"));

```



```

3295         return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
3296     }
3298     (void) strncpy(zc.zc_value, zhp->zfs_dmstats.dds_origin,
3299                 sizeof (zc.zc_value));
3300     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
3301     ret = zfs_ioctl(hdl, ZFS_IOC_PROMOTE, &zc);
3303     if (ret != 0) {
3304         int save_errno = errno;
3306         switch (save_errno) {
3307             case EEXIST:
3308                 /* There is a conflicting snapshot name. */
3309                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3310                     "conflicting snapshot '%s' from parent '%s'",
3311                     zc.zc_string, parent);
3312                 return (zfs_error(hdl, EZFS_EXISTS, errbuf));
3314             default:
3315                 return (zfs_standard_error(hdl, save_errno, errbuf));
3316         }
3317     }
3318     return (ret);
3319 }
3321 typedef struct snapdata {
3322     nvlist_t *sd_nvl;
3323     const char *sd_snapname;
3324 } snapdata_t;
3326 static int
3327 zfs_snapshot_cb(zfs_handle_t *zhp, void *arg)
3328 {
3329     snapdata_t *sd = arg;
3330     char name[ZFS_MAXNAMELEN];
3331     int rv = 0;
3333     (void) snprintf(name, sizeof (name),
3334                    "%s%s", zfs_get_name(zhp), sd->sd_snapname);
3336     fnvlist_add_boolean(sd->sd_nvl, name);
3338     rv = zfs_iter_filesystems(zhp, zfs_snapshot_cb, sd);
3339     zfs_close(zhp);
3340     return (rv);
3341 }
3343 /*
3344  * Creates snapshots. The keys in the snaps nvlist are the snapshots to be
3345  * created.
3346  */
3347 int
3348 zfs_snapshot_nvl(libzfs_handle_t *hdl, nvlist_t *snaps, nvlist_t *props)
3349 {
3350     int ret;
3351     char errbuf[1024];
3352     nvpair_t *elem;
3353     nvlist_t *errors;
3355     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3356                 "cannot create snapshots "));
3358     elem = NULL;
3359     while ((elem = nvlist_next_nvpair(snaps, elem)) != NULL) {
3360         const char *snapname = nvpair_name(elem);

```

```

3362         /* validate the target name */
3363         if (!zfs_validate_name(hdl, snapname, ZFS_TYPE_SNAPSHOT,
3364             B_TRUE)) {
3365             (void) snprintf(errbuf, sizeof (errbuf),
3366                 dgettext(TEXT_DOMAIN,
3367                     "cannot create snapshot '%s'", snapname);
3368             return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3369         }
3370     }
3372     if (props != NULL &&
3373         (props = zfs_valid_proplist(hdl, ZFS_TYPE_SNAPSHOT,
3374             props, B_FALSE, NULL, errbuf)) == NULL) {
3375         return (-1);
3376     }
3378     ret = lzfs_snapshot(snaps, props, &errors);
3380     if (ret != 0) {
3381         boolean_t printed = B_FALSE;
3382         for (elem = nvlist_next_nvpair(errors, NULL);
3383             elem != NULL;
3384             elem = nvlist_next_nvpair(errors, elem)) {
3385             (void) snprintf(errbuf, sizeof (errbuf),
3386                 dgettext(TEXT_DOMAIN,
3387                     "cannot create snapshot '%s'", nvpair_name(elem));
3388             (void) zfs_standard_error(hdl,
3389                 fnvpair_value_int32(elem), errbuf);
3390             printed = B_TRUE;
3391         }
3392         if (!printed) {
3393             switch (ret) {
3394                 case EXDEV:
3395                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3396                         "multiple snapshots of same "
3397                         "fs not allowed"));
3398                     (void) zfs_error(hdl, EZFS_EXISTS, errbuf);
3400                 break;
3401                 default:
3402                     (void) zfs_standard_error(hdl, ret, errbuf);
3403             }
3404         }
3405     }
3407     nvlist_free(props);
3408     nvlist_free(errors);
3409     return (ret);
3410 }
3412 int
3413 zfs_snapshot(libzfs_handle_t *hdl, const char *path, boolean_t recursive,
3414             nvlist_t *props)
3415 {
3416     int ret;
3417     snapdata_t sd = { 0 };
3418     char fsname[ZFS_MAXNAMELEN];
3419     char *cp;
3420     zfs_handle_t *zhp;
3421     char errbuf[1024];
3423     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3424                 "cannot snapshot %s"), path);
3426     if (!zfs_validate_name(hdl, path, ZFS_TYPE_SNAPSHOT, B_TRUE))

```

```

3427         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3429     (void) strncpy(fsname, path, sizeof (fsname));
3430     cp = strchr(fsname, '@');
3431     *cp = '\0';
3432     sd.sd_snapname = cp + 1;
3434     if ((zhp = zfs_open(hdl, fsname, ZFS_TYPE_FILESYSTEM |
3435         ZFS_TYPE_VOLUME)) == NULL) {
3436         return (-1);
3437     }
3439     verify(nvlist_alloc(&sd.sd_nvlist, NV_UNIQUE_NAME, 0) == 0);
3440     if (recursive) {
3441         (void) zfs_snapshot_cb(zfs_handle_dup(zhp), &sd);
3442     } else {
3443         fnvlist_add_boolean(sd.sd_nvlist, path);
3444     }
3446     ret = zfs_snapshot_nvlist(hdl, sd.sd_nvlist, props);
3447     nvlist_free(sd.sd_nvlist);
3448     zfs_close(zhp);
3449     return (ret);
3450 }
3452 /*
3453  * Destroy any more recent snapshots. We invoke this callback on any dependents
3454  * of the snapshot first. If the 'cb_dependent' member is non-zero, then this
3455  * is a dependent and we should just destroy it without checking the transaction
3456  * group.
3457  */
3458 typedef struct rollback_data {
3459     const char    *cb_target;          /* the snapshot */
3460     uint64_t      cb_create;          /* creation time reference */
3461     boolean_t     cb_error;
3462     boolean_t     cb_dependent;
3463     boolean_t     cb_force;
3464 } rollback_data_t;
3466 static int
3467 rollback_destroy(zfs_handle_t *zhp, void *data)
3468 {
3469     rollback_data_t *cbp = data;
3471     if (!cbp->cb_dependent) {
3472         if (strcmp(zhp->zfs_name, cbp->cb_target) != 0 &&
3473             zfs_get_type(zhp) == ZFS_TYPE_SNAPSHOT &&
3474             zfs_prop_get_int(zhp, ZFS_PROP_CREATETXG) >
3475             cbp->cb_create) {
3477                 cbp->cb_dependent = B_TRUE;
3478                 cbp->cb_error |= zfs_iter_dependents(zhp, B_FALSE,
3479                     rollback_destroy, cbp);
3480                 cbp->cb_dependent = B_FALSE;
3482                 cbp->cb_error |= zfs_destroy(zhp, B_FALSE);
3483             }
3484     } else {
3485         /* We must destroy this clone; first unmount it */
3486         prop_changelist_t *clp;
3488         clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
3489             cbp->cb_force ? MS_FORCE : 0);
3490         if (clp == NULL || changelist_prefix(clp) != 0) {
3491             cbp->cb_error = B_TRUE;
3492             zfs_close(zhp);

```

```

3493         return (0);
3494     }
3495     if (zfs_destroy(zhp, B_FALSE) != 0)
3496         cbp->cb_error = B_TRUE;
3497     else
3498         changelist_remove(clp, zhp->zfs_name);
3499     (void) changelist_postfix(clp);
3500     changelist_free(clp);
3501 }
3503     zfs_close(zhp);
3504     return (0);
3505 }
3507 /*
3508  * Given a dataset, rollback to a specific snapshot, discarding any
3509  * data changes since then and making it the active dataset.
3510  *
3511  * Any snapshots more recent than the target are destroyed, along with
3512  * their dependents.
3513  */
3514 int
3515 zfs_rollback(zfs_handle_t *zhp, zfs_handle_t *snap, boolean_t force)
3516 {
3517     rollback_data_t cb = { 0 };
3518     int err;
3519     zfs_cmd_t zc = { 0 };
3520     boolean_t restore_resv = 0;
3521     uint64_t old_volsize, new_volsize;
3522     zfs_prop_t resv_prop;
3524     assert(zhp->zfs_type == ZFS_TYPE_FILESYSTEM ||
3525         zhp->zfs_type == ZFS_TYPE_VOLUME);
3527     /*
3528      * Destroy all recent snapshots and their dependents.
3529      */
3530     cb.cb_force = force;
3531     cb.cb_target = snap->zfs_name;
3532     cb.cb_create = zfs_prop_get_int(snap, ZFS_PROP_CREATETXG);
3533     (void) zfs_iter_children(zhp, rollback_destroy, &cb);
3535     if (cb.cb_error)
3536         return (-1);
3538     /*
3539      * Now that we have verified that the snapshot is the latest,
3540      * rollback to the given snapshot.
3541      */
3543     if (zhp->zfs_type == ZFS_TYPE_VOLUME) {
3544         if (zfs_which_resv_prop(zhp, &resv_prop) < 0)
3545             return (-1);
3546         old_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
3547         restore_resv =
3548             (old_volsize == zfs_prop_get_int(zhp, resv_prop));
3549     }
3551     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
3553     if (ZFS_IS_VOLUME(zhp))
3554         zc.zc_objset_type = DMU_OST_ZVOL;
3555     else
3556         zc.zc_objset_type = DMU_OST_ZFS;
3558     /*

```

```

3559  * We rely on zfs_iter_children() to verify that there are no
3560  * newer snapshots for the given dataset. Therefore, we can
3561  * simply pass the name on to the ioctl() call. There is still
3562  * an unlikely race condition where the user has taken a
3563  * snapshot since we verified that this was the most recent.
3564  *
3565  */
3566  if ((err = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_ROLLBACK, &zrc)) != 0) {
3567      (void) zfs_standard_error_fmt(zhp->zfs_hdl, errno,
3568          dgettext(TEXT_DOMAIN, "cannot rollback '%s'",
3569              zhp->zfs_name);
3570      return (err);
3571  }
3572
3573  /*
3574  * For volumes, if the pre-rollback volsize matched the pre-
3575  * rollback reservation and the volsize has changed then set
3576  * the reservation property to the post-rollback volsize.
3577  * Make a new handle since the rollback closed the dataset.
3578  */
3579  if ((zhp->zfs_type == ZFS_TYPE_VOLUME) &&
3580      (zhp = make_dataset_handle(zhp->zfs_hdl, zhp->zfs_name))) {
3581      if (restore_resv) {
3582          new_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
3583          if (old_volsize != new_volsize)
3584              err = zfs_prop_set_int(zhp, resv_prop,
3585                  new_volsize);
3586      }
3587      zfs_close(zhp);
3588  }
3589  return (err);
3590 }
3591
3592 /*
3593  * Renames the given dataset.
3594  */
3595 int
3596 zfs_rename(zfs_handle_t *zhp, const char *target, boolean_t recursive,
3597     boolean_t force_unmount)
3598 {
3599     int ret;
3600     zfs_cmd_t zc = { 0 };
3601     char *delim;
3602     prop_changelist_t *cl = NULL;
3603     zfs_handle_t *zhdp = NULL;
3604     char *parentname = NULL;
3605     char parent[ZFS_MAXNAMELEN];
3606     libzfs_handle_t *hdl = zhp->zfs_hdl;
3607     char errbuf[1024];
3608
3609     /* if we have the same exact name, just return success */
3610     if (strcmp(zhp->zfs_name, target) == 0)
3611         return (0);
3612
3613     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3614         "cannot rename to '%s'", target);
3615
3616     /*
3617      * Make sure the target name is valid
3618      */
3619     if (zhp->zfs_type == ZFS_TYPE_SNAPSHOT) {
3620         if ((strchr(target, '@') == NULL) ||
3621             *target == '@') {
3622             /*
3623              * Snapshot target name is abbreviated,
3624              * reconstruct full dataset name

```

```

3625     */
3626     (void) strncpy(parent, zhp->zfs_name,
3627         sizeof (parent));
3628     delim = strchr(parent, '@');
3629     if (strchr(target, '@') == NULL)
3630         *(++delim) = '\0';
3631     else
3632         *delim = '\0';
3633     (void) strlcat(parent, target, sizeof (parent));
3634     target = parent;
3635 } else {
3636     /*
3637      * Make sure we're renaming within the same dataset.
3638      */
3639     delim = strchr(target, '@');
3640     if (strcmp(zhp->zfs_name, target, delim - target)
3641         != 0 || zhp->zfs_name[delim - target] != '@') {
3642         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3643             "snapshots must be part of same "
3644             "dataset"));
3645         return (zfs_error(hdl, EZFS_CROSSTARGET,
3646             errbuf));
3647     }
3648 }
3649 if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
3650     return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3651 } else {
3652     if (recursive) {
3653         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3654             "recursive rename must be a snapshot"));
3655         return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
3656     }
3657
3658     if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
3659         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3660
3661     /* validate parents */
3662     if (check_parents(hdl, target, NULL, B_FALSE, NULL) != 0)
3663         return (-1);
3664
3665     /* make sure we're in the same pool */
3666     verify((delim = strchr(target, '/')) != NULL);
3667     if (strcmp(zhp->zfs_name, target, delim - target) != 0 ||
3668         zhp->zfs_name[delim - target] != '/') {
3669         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3670             "datasets must be within same pool"));
3671         return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));
3672     }
3673
3674     /* new name cannot be a child of the current dataset name */
3675     if (is_descendant(zhp->zfs_name, target)) {
3676         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3677             "New dataset name cannot be a descendant of "
3678             "current dataset name"));
3679         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3680     }
3681 }
3682
3683 (void) snprintf(errbuf, sizeof (errbuf),
3684     dgettext(TEXT_DOMAIN, "cannot rename '%s'", zhp->zfs_name);
3685
3686 if (getzoneid() == GLOBAL_ZONEID &&
3687     zfs_prop_get_int(zhp, ZFS_PROP_ZONED)) {
3688     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3689         "dataset is used in a non-global zone"));
3690     return (zfs_error(hdl, EZFS_ZONED, errbuf));

```

```

3691     }
3693     if (recursive) {
3695         parentname = zfs_strdup(zhp->zfs_hdl, zhp->zfs_name);
3696         if (parentname == NULL) {
3697             ret = -1;
3698             goto error;
3699         }
3700         delim = strchr(parentname, '@');
3701         *delim = '\0';
3702         zhrp = zfs_open(zhp->zfs_hdl, parentname, ZFS_TYPE_DATASET);
3703         if (zhrp == NULL) {
3704             ret = -1;
3705             goto error;
3706         }
3708     } else {
3709         if ((cl = changelist_gather(zhp, ZFS_PROP_NAME, 0,
3710             force_unmount ? MS_FORCE : 0)) == NULL)
3711             return (-1);
3713         if (changelist_haszonedchild(cl)) {
3714             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3715                 "child dataset with inherited mountpoint is used "
3716                 "in a non-global zone"));
3717             (void) zfs_error(hdl, EZFS_ZONED, errbuf);
3718             goto error;
3719         }
3721         if ((ret = changelist_prefix(cl)) != 0)
3722             goto error;
3723     }
3725     if (ZFS_IS_VOLUME(zhp))
3726         zc.zc_objset_type = DMU_OST_ZVOL;
3727     else
3728         zc.zc_objset_type = DMU_OST_ZFS;
3730     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
3731     (void) strncpy(zc.zc_value, target, sizeof(zc.zc_value));
3733     zc.zc_cookie = recursive;
3735     if ((ret = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_RENAME, &zc)) != 0) {
3736         /*
3737          * if it was recursive, the one that actually failed will
3738          * be in zc.zc_name
3739          */
3740         (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
3741             "cannot rename '%s'", zc.zc_name);
3743         if (recursive && errno == EEXIST) {
3744             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3745                 "a child dataset already has a snapshot "
3746                 "with the new name"));
3747             (void) zfs_error(hdl, EZFS_EXISTS, errbuf);
3748         } else {
3749             (void) zfs_standard_error(zhp->zfs_hdl, errno, errbuf);
3750         }
3752         /*
3753          * On failure, we still want to remount any filesystems that
3754          * were previously mounted, so we don't alter the system state.
3755          */
3756         if (!recursive)

```

```

3757         (void) changelist_postfix(cl);
3758     } else {
3759         if (!recursive) {
3760             changelist_rename(cl, zfs_get_name(zhp), target);
3761             ret = changelist_postfix(cl);
3762         }
3763     }
3765 error:
3766     if (parentname) {
3767         free(parentname);
3768     }
3769     if (zhrp) {
3770         zfs_close(zhrp);
3771     }
3772     if (cl) {
3773         changelist_free(cl);
3774     }
3775     return (ret);
3776 }
3778 nvlist_t *
3779 zfs_get_user_props(zfs_handle_t *zhp)
3780 {
3781     return (zhp->zfs_user_props);
3782 }
3784 nvlist_t *
3785 zfs_get_recvd_props(zfs_handle_t *zhp)
3786 {
3787     if (zhp->zfs_recvd_props == NULL)
3788         if (get_recvd_props_ioctl(zhp) != 0)
3789             return (NULL);
3790     return (zhp->zfs_recvd_props);
3791 }
3793 /*
3794  * This function is used by 'zfs list' to determine the exact set of columns to
3795  * display, and their maximum widths. This does two main things:
3796  *
3797  * - If this is a list of all properties, then expand the list to include
3798  *   all native properties, and set a flag so that for each dataset we look
3799  *   for new unique user properties and add them to the list.
3800  *
3801  * - For non fixed-width properties, keep track of the maximum width seen
3802  *   so that we can size the column appropriately. If the user has
3803  *   requested received property values, we also need to compute the width
3804  *   of the RECEIVED column.
3805  */
3806 int
3807 zfs_expand_proplist(zfs_handle_t *zhp, zprop_list_t **plp, boolean_t received)
3808 {
3809     libzfs_handle_t *hdl = zhp->zfs_hdl;
3810     zprop_list_t *entry;
3811     zprop_list_t **last, **start;
3812     nvlist_t *userprops, *propval;
3813     nvpair_t *elem;
3814     char *strval;
3815     char buf[ZFS_MAXPROPLEN];
3817     if (zprop_expand_list(hdl, plp, ZFS_TYPE_DATASET) != 0)
3818         return (-1);
3820     userprops = zfs_get_user_props(zhp);
3822     entry = *plp;

```

```

3823     if (entry->pl_all && nvlist_next_nvpair(userprops, NULL) != NULL) {
3824         /*
3825          * Go through and add any user properties as necessary.  We
3826          * start by incrementing our list pointer to the first
3827          * non-native property.
3828          */
3829         start = plp;
3830         while (*start != NULL) {
3831             if ((*start)->pl_prop == ZPROP_INVAL)
3832                 break;
3833             start = &(*start)->pl_next;
3834         }
3835
3836         elem = NULL;
3837         while ((elem = nvlist_next_nvpair(userprops, elem)) != NULL) {
3838             /*
3839              * See if we've already found this property in our list.
3840              */
3841             for (last = start; *last != NULL;
3842                  last = &(*last)->pl_next) {
3843                 if (strcmp((*last)->pl_user_prop,
3844                             nvpair_name(elem)) == 0)
3845                     break;
3846             }
3847
3848             if (*last == NULL) {
3849                 if ((entry = zfs_alloc(hdl,
3850                                     sizeof (zprop_list_t))) == NULL ||
3851                     ((entry->pl_user_prop = zfs_strdup(hdl,
3852                                                         nvpair_name(elem))) == NULL) {
3853                     free(entry);
3854                     return (-1);
3855                 }
3856
3857                 entry->pl_prop = ZPROP_INVAL;
3858                 entry->pl_width = strlen(nvpair_name(elem));
3859                 entry->pl_all = B_TRUE;
3860                 *last = entry;
3861             }
3862         }
3863     }
3864
3865     /*
3866      * Now go through and check the width of any non-fixed columns
3867      */
3868     for (entry = *plp; entry != NULL; entry = entry->pl_next) {
3869         if (entry->pl_fixed)
3870             continue;
3871
3872         if (entry->pl_prop != ZPROP_INVAL) {
3873             if (zfs_prop_get(zhp, entry->pl_prop,
3874                             buf, sizeof (buf), NULL, 0, B_FALSE) == 0) {
3875                 if (strlen(buf) > entry->pl_width)
3876                     entry->pl_width = strlen(buf);
3877             }
3878             if (received && zfs_prop_get_recvd(zhp,
3879                                                 zfs_prop_to_name(entry->pl_prop),
3880                                                 buf, sizeof (buf), B_FALSE) == 0)
3881                 if (strlen(buf) > entry->pl_recvd_width)
3882                     entry->pl_recvd_width = strlen(buf);
3883         } else {
3884             if (nvlist_lookup_nvlist(userprops, entry->pl_user_prop,
3885                                     &propval) == 0) {
3886                 verify(nvlist_lookup_string(propval,
3887                                             ZPROP_VALUE, &strval) == 0);
3888                 if (strlen(strval) > entry->pl_width)

```

```

3889         entry->pl_width = strlen(strval);
3890     }
3891     if (received && zfs_prop_get_recvd(zhp,
3892         entry->pl_user_prop,
3893         buf, sizeof (buf), B_FALSE) == 0)
3894         if (strlen(buf) > entry->pl_recvd_width)
3895             entry->pl_recvd_width = strlen(buf);
3896     }
3897 }
3898
3899     return (0);
3900 }
3901
3902 int
3903 zfs_deleg_share_nfs(libzfs_handle_t *hdl, char *dataset, char *path,
3904 char *resource, void *export, void *sharetab,
3905 int sharemax, zfs_share_op_t operation)
3906 {
3907     zfs_cmd_t zc = { 0 };
3908     int error;
3909
3910     (void) strcpy(zc.zc_name, dataset, sizeof (zc.zc_name));
3911     (void) strcpy(zc.zc_value, path, sizeof (zc.zc_value));
3912     if (resource)
3913         (void) strcpy(zc.zc_string, resource, sizeof (zc.zc_string));
3914     zc.zc_share.z_sharedata = (uint64_t)(uintptr_t)sharetab;
3915     zc.zc_share.z_exportdata = (uint64_t)(uintptr_t)export;
3916     zc.zc_share.z_sharetype = operation;
3917     zc.zc_share.z_sharemax = sharemax;
3918     error = ioctl(hdl->libzfs_fd, ZFS_IOC_SHARE, &zc);
3919     return (error);
3920 }
3921
3922 void
3923 zfs_prune_proplist(zfs_handle_t *zhp, uint8_t *props)
3924 {
3925     nvpair_t *curr;
3926
3927     /*
3928      * Keep a reference to the props-table against which we prune the
3929      * properties.
3930      */
3931     zhp->zfs_props_table = props;
3932
3933     curr = nvlist_next_nvpair(zhp->zfs_props, NULL);
3934
3935     while (curr) {
3936         zfs_prop_t zfs_prop = zfs_name_to_prop(nvpair_name(curr));
3937         nvpair_t *next = nvlist_next_nvpair(zhp->zfs_props, curr);
3938
3939         /*
3940          * User properties will result in ZPROP_INVAL, and since we
3941          * only know how to prune standard ZFS properties, we always
3942          * leave these in the list.  This can also happen if we
3943          * encounter an unknown DSL property (when running older
3944          * software, for example).
3945          */
3946         if (zfs_prop != ZPROP_INVAL && props[zfs_prop] == B_FALSE)
3947             (void) nvlist_remove(zhp->zfs_props,
3948                                 nvpair_name(curr), nvpair_type(curr));
3949         curr = next;
3950     }
3951 }
3952
3953 static int
3954 zfs_smb_acl_mgmt(libzfs_handle_t *hdl, char *dataset, char *path,

```

```

3955     zfs_smb_acl_op_t cmd, char *resource1, char *resource2)
3956 {
3957     zfs_cmd_t zc = { 0 };
3958     nvlist_t *nvlist = NULL;
3959     int error;

3961     (void) strcpy(zc.zc_name, dataset, sizeof (zc.zc_name));
3962     (void) strcpy(zc.zc_value, path, sizeof (zc.zc_value));
3963     zc.zc_cookie = (uint64_t)cmd;

3965     if (cmd == ZFS_SMB_ACL_RENAME) {
3966         if (nvlist_alloc(&nvlist, NV_UNIQUE_NAME, 0) != 0) {
3967             (void) no_memory(hdl);
3968             return (NULL);
3969         }
3970     }

3972     switch (cmd) {
3973     case ZFS_SMB_ACL_ADD:
3974     case ZFS_SMB_ACL_REMOVE:
3975         (void) strcpy(zc.zc_string, resource1, sizeof (zc.zc_string));
3976         break;
3977     case ZFS_SMB_ACL_RENAME:
3978         if (nvlist_add_string(nvlist, ZFS_SMB_ACL_SRC,
3979             resource1) != 0) {
3980             (void) no_memory(hdl);
3981             return (-1);
3982         }
3983         if (nvlist_add_string(nvlist, ZFS_SMB_ACL_TARGET,
3984             resource2) != 0) {
3985             (void) no_memory(hdl);
3986             return (-1);
3987         }
3988         if (zcmd_write_src_nvlist(hdl, &zc, nvlist) != 0) {
3989             nvlist_free(nvlist);
3990             return (-1);
3991         }
3992         break;
3993     case ZFS_SMB_ACL_PURGE:
3994         break;
3995     default:
3996         return (-1);
3997     }
3998     error = ioctl(hdl->libzfs_fd, ZFS_IOC_SMB_ACL, &zc);
3999     if (nvlist)
4000         nvlist_free(nvlist);
4001     return (error);
4002 }

4004 int
4005 zfs_smb_acl_add(libzfs_handle_t *hdl, char *dataset,
4006     char *path, char *resource)
4007 {
4008     return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_ADD,
4009     resource, NULL));
4010 }

4012 int
4013 zfs_smb_acl_remove(libzfs_handle_t *hdl, char *dataset,
4014     char *path, char *resource)
4015 {
4016     return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_REMOVE,
4017     resource, NULL));
4018 }

4020 int

```

```

4021 zfs_smb_acl_purge(libzfs_handle_t *hdl, char *dataset, char *path)
4022 {
4023     return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_PURGE,
4024     NULL, NULL));
4025 }

4027 int
4028 zfs_smb_acl_rename(libzfs_handle_t *hdl, char *dataset, char *path,
4029     char *oldname, char *newname)
4030 {
4031     return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_RENAME,
4032     oldname, newname));
4033 }

4035 int
4036 zfs_userspace(zfs_handle_t *zhp, zfs_userquota_prop_t type,
4037     zfs_userspace_cb_t func, void *arg)
4038 {
4039     zfs_cmd_t zc = { 0 };
4040     zfs_useracct_t buf[100];
4041     libzfs_handle_t *hdl = zhp->zfs_hdl;
4042     int ret;

4044     (void) strcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

4046     zc.zc_objset_type = type;
4047     zc.zc_nvlist_dst = (uintptr_t)buf;

4049     for (;;) {
4050         zfs_useracct_t *zua = buf;

4052         zc.zc_nvlist_dst_size = sizeof (buf);
4053         if (zfs_ioctl(hdl, ZFS_IOC_USERSPACE_MANY, &zc) != 0) {
4054             char errbuf[1024];

4056             (void) snprintf(errbuf, sizeof (errbuf),
4057                 dgettext(TEXT_DOMAIN,
4058                     "cannot get used/quota for %s"), zc.zc_name);
4059             return (zfs_standard_error_fmt(hdl, errno, errbuf));
4060         }
4061         if (zc.zc_nvlist_dst_size == 0)
4062             break;

4064         while (zc.zc_nvlist_dst_size > 0) {
4065             if ((ret = func(arg, zua->zu_domain, zua->zu_rid,
4066                 zua->zu_space)) != 0)
4067                 return (ret);
4068             zua++;
4069             zc.zc_nvlist_dst_size -= sizeof (zfs_useracct_t);
4070         }
4071     }

4073     return (0);
4074 }

4076 struct holdarg {
4077     nvlist_t *nvl;
4078     const char *snapname;
4079     const char *tag;
4080     boolean_t recursive;
4081 };

4083 static int
4084 zfs_hold_one(zfs_handle_t *zhp, void *arg)
4085 {
4086     struct holdarg *ha = arg;

```

```

28     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     if (lzc_exists(name))
35     szhp = make_dataset_handle(zhp->zfs_hdl, name);
36     if (szhp) {
4094         fnvlist_add_string(ha->nvl, name, ha->tag);
38         zfs_close(szhp);
39     }

4096     if (ha->recursive)
4097         rv = zfs_iter_filesystems(zhp, zfs_hold_one, ha);
4098     zfs_close(zhp);
4099     return (rv);
4100 }

4102 int
4103 zfs_hold(zfs_handle_t *zhp, const char *snapname, const char *tag,
4104          boolean_t recursive, int cleanup_fd)
49     boolean_t recursive, boolean_t enoent_ok, int cleanup_fd)
4105 {
4106     int ret;
4107     struct holdarg ha;
4108     nvlist_t *errors;
53     libzfs_handle_t *hdl = zhp->zfs_hdl;
54     char errbuf[1024];
55     nvpair_t *elem;
56

4109     ha.nvl = fnvlist_alloc();
4110     ha.snapname = snapname;
4111     ha.tag = tag;
4112     ha.recursive = recursive;
4113     (void) zfs_hold_one(zfs_handle_dup(zhp), &ha);

4115     if (nvlist_next_nvpair(ha.nvl, NULL) == NULL) {
4116         char errbuf[1024];

4118 #endif /* ! codereview */
4119         fnvlist_free(ha.nvl);
4120         ret = ENOENT;
4121         if (!enoent_ok) {
4122             (void) snprintf(errbuf, sizeof (errbuf),
4123                            dgettext(TEXT_DOMAIN,
4124                                     "cannot hold snapshot '%s@%s'"),
4125                            zhp->zfs_name, snapname);
4126             (void) zfs_standard_error(zhp->zfs_hdl, ret, errbuf);
4127             (void) zfs_standard_error(hdl, ret, errbuf);
4128         }
4129         return (ret);
4130     }

4132     ret = zfs_hold_nvl(zhp, cleanup_fd, ha.nvl);
4133     ret = lzc_hold(ha.nvl, cleanup_fd, &errors);
4134     fnvlist_free(ha.nvl);

4136     return (ret);
4137 }

4138 int
4139 zfs_hold_nvl(zfs_handle_t *zhp, int cleanup_fd, nvlist_t *holds)
4140 {
4141     int ret;

```

```

4139     nvlist_t *errors;
4140     libzfs_handle_t *hdl = zhp->zfs_hdl;
4141     char errbuf[1024];
4142     nvpair_t *elem;

4144     errors = NULL;
4145     ret = lzc_hold(holds, cleanup_fd, &errors);

4147     if (ret == 0) {
4148         /* There may be errors even in the success case. */
4149         fnvlist_free(errors);
4150         if (ret == 0)
4151             return (0);
4152 #endif /* ! codereview */

4154     if (nvlist_next_nvpair(errors, NULL) == NULL) {
4155         /* no hold-specific errors */
4156         (void) snprintf(errbuf, sizeof (errbuf),
4157                        dgettext(TEXT_DOMAIN, "cannot hold"));
4158         switch (ret) {
4159             case ENOTSUP:
4160                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4161                                             "pool must be upgraded"));
4162                 (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4163                 break;
4164             case EINVAL:
4165                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4166                 break;
4167             default:
4168                 (void) zfs_standard_error(hdl, ret, errbuf);
4169         }
4170     }

4172     for (elem = nvlist_next_nvpair(errors, NULL);
4173          elem != NULL;
4174          elem = nvlist_next_nvpair(errors, elem)) {
4175         (void) snprintf(errbuf, sizeof (errbuf),
4176                        dgettext(TEXT_DOMAIN,
4177                                 "cannot hold snapshot '%s'", nvpair_name(elem));
4178         switch (fnvpair_value_int32(elem)) {
4179             case E2BIG:
4180                 /*
4181                  * Temporary tags wind up having the ds object id
4182                  * prepended. So even if we passed the length check
4183                  * above, it's still possible for the tag to wind
4184                  * up being slightly too long.
4185                  */
4186                 (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
4187                 break;
4188             case EINVAL:
4189                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4190                 break;
4191             case EEXIST:
4192                 (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
4193                 break;
4194             case ENOENT:
4195                 if (enoent_ok)
4196                     return (ENOENT);
4197                 /* FALLTHROUGH */
4198             default:
4199                 (void) zfs_standard_error(hdl,
4200                                           fnvpair_value_int32(elem), errbuf);
4201         }
4202     }

```

```

4200     fnvlist_free(errors);
4201     return (ret);
4202 }

94 struct releasearg {
95     nvlist_t *nvl;
96     const char *snapname;
97     const char *tag;
98     boolean_t recursive;
99 };

4204 static int
4205 zfs_release_one(zfs_handle_t *zhp, void *arg)
4206 {
4207     struct holdarg *ha = arg;
4208     zfs_handle_t *szhp;
4209     char name[ZFS_MAXNAMELEN];
4210     int rv = 0;

4211     (void) snprintf(name, sizeof (name),
4212                    "%s@%s", zhp->zfs_name, ha->snapname);

4214     if (lzc_exists(name)) {
4215         szhp = make_dataset_handle(zhp->zfs_hdl, name);
4216         if (szhp) {
4217             nvlist_t *holds = fnvlist_alloc();
4218             fnvlist_add_boolean(holds, ha->tag);
4219             fnvlist_add_nvlist(ha->nvl, name, holds);
4220             fnvlist_free(holds);
4221             zfs_close(szhp);
4222         }

4223     if (ha->recursive)
4224         rv = zfs_iter_filesystems(zhp, zfs_release_one, ha);
4225     zfs_close(zhp);
4226     return (rv);
4227 }

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

4238     ha.nvl = fnvlist_alloc();
4239     ha.snapname = snapname;
4240     ha.tag = tag;
4241     ha.recursive = recursive;
4242     (void) zfs_release_one(zfs_handle_dup(zhp), &ha);

4244     if (nvlist_next_nvpair(ha.nvl, NULL) == NULL) {
4245         fnvlist_free(ha.nvl);
4246         ret = ENOENT;
4247         (void) snprintf(errbuf, sizeof (errbuf),
4248                        dgettext(TEXT_DOMAIN,
4249                                "cannot release hold from snapshot '%s@%s'"),
4250                        zhp->zfs_name, snapname);
4251         (void) zfs_standard_error(hdl, ret, errbuf);
4252         return (ret);
4253     }

```

```

4255     errors = NULL;
4256 #endif /* ! codereview */
4257     ret = lzc_release(ha.nvl, &errors);
4258     fnvlist_free(ha.nvl);

4260     if (ret == 0) {
4261         /* There may be errors even in the success case. */
4262         fnvlist_free(errors);
4263         if (ret == 0)
4264             return (0);
4265 #endif /* ! codereview */

4267     if (nvlist_next_nvpair(errors, NULL) == NULL) {
4268         /* no hold-specific errors */
4269         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
4270                                                         "cannot release"));
4271         switch (errno) {
4272             case ENOTSUP:
4273                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4274                                             "pool must be upgraded"));
4275                 (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4276                 break;
4277             default:
4278                 (void) zfs_standard_error_fmt(hdl, errno, errbuf);
4279         }
4280     }

4282     for (elem = nvlist_next_nvpair(errors, NULL);
4283          elem != NULL;
4284          elem = nvlist_next_nvpair(errors, elem)) {
4285         (void) snprintf(errbuf, sizeof (errbuf),
4286                        dgettext(TEXT_DOMAIN,
4287                                "cannot release hold from snapshot '%s'"),
4288                                nvpair_name(elem));
4289         switch (fnvpair_value_int32(elem)) {
4290             case ESRCH:
4291                 (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
4292                 break;
4293             case EINVAL:
4294                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4295                 break;
4296             default:
4297                 (void) zfs_standard_error_fmt(hdl,
4298                                                fnvpair_value_int32(elem), errbuf);
4299         }
4300     }

4302     fnvlist_free(errors);
4303     return (ret);
4304 }

4306 int
4307 zfs_get_fsacl(zfs_handle_t *zhp, nvlist_t **nvl)
4308 {
4309     zfs_cmd_t zc = { 0 };
4310     libzfs_handle_t *hdl = zhp->zfs_hdl;
4311     int nvsz = 2048;
4312     void *nvbuf;
4313     int err = 0;
4314     char errbuf[1024];

4316     assert(zhp->zfs_type == ZFS_TYPE_VOLUME ||
4317           zhp->zfs_type == ZFS_TYPE_FILESYSTEM);

4319     tryagain:

```



```

4321     nvbuf = malloc(nvsz);
4322     if (nvbuf == NULL) {
4323         err = (zfs_error(hdl, EZFS_NOMEM, strerror(errno)));
4324         goto out;
4325     }
4327     zc.zc_nvlist_dst_size = nvsz;
4328     zc.zc_nvlist_dst = (uintptr_t)nvbuf;
4330
4330     (void) strncpy(zc.zc_name, zhp->zfs_name, ZFS_MAXNAMELEN);
4332
4332     if (ioctl(hdl->libzfs_fd, ZFS_IOC_GET_FSACL, &zc) != 0) {
4333         (void) snprintf(errbuf, sizeof(errbuf),
4334             dgettext(TEXT_DOMAIN, "cannot get permissions on '%s'",
4335                 zc.zc_name);
4336         switch (errno) {
4337             case ENOMEM:
4338                 free(nvbuf);
4339                 nvsz = zc.zc_nvlist_dst_size;
4340                 goto tryagain;
4342
4342             case ENOTSUP:
4343                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4344                     "pool must be upgraded"));
4345                 err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4346                 break;
4347             case EINVAL:
4348                 err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4349                 break;
4350             case ENOENT:
4351                 err = zfs_error(hdl, EZFS_NOENT, errbuf);
4352                 break;
4353             default:
4354                 err = zfs_standard_error_fmt(hdl, errno, errbuf);
4355                 break;
4356         }
4357     } else {
4358         /* success */
4359         int rc = nvlist_unpack(nvbuf, zc.zc_nvlist_dst_size, nvl, 0);
4360         if (rc) {
4361             (void) snprintf(errbuf, sizeof(errbuf), dgettext(
4362                 TEXT_DOMAIN, "cannot get permissions on '%s'",
4363                 zc.zc_name);
4364             err = zfs_standard_error_fmt(hdl, rc, errbuf);
4365         }
4366     }
4368     free(nvbuf);
4369 out:
4370     return (err);
4371 }
4373 int
4374 zfs_set_fs_acl(zfs_handle_t *zhp, boolean_t un, nvlist_t *nvl)
4375 {
4376     zfs_cmd_t zc = { 0 };
4377     libzfs_handle_t *hdl = zhp->zfs_hdl;
4378     char *nvbuf;
4379     char errbuf[1024];
4380     size_t nvsz;
4381     int err;
4383
4383     assert(zhp->zfs_type == ZFS_TYPE_VOLUME ||
4384         zhp->zfs_type == ZFS_TYPE_FILESYSTEM);

```

```

4386     err = nvlist_size(nvl, &nvsz, NV_ENCODE_NATIVE);
4387     assert(err == 0);
4389     nvbuf = malloc(nvsz);
4391
4391     err = nvlist_pack(nvl, &nvbuf, &nvsz, NV_ENCODE_NATIVE, 0);
4392     assert(err == 0);
4394
4394     zc.zc_nvlist_src_size = nvsz;
4395     zc.zc_nvlist_src = (uintptr_t)nvbuf;
4396     zc.zc_perm_action = un;
4398
4398     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
4400
4400     if (zfs_ioctl(hdl, ZFS_IOC_SET_FSACL, &zc) != 0) {
4401         (void) snprintf(errbuf, sizeof(errbuf),
4402             dgettext(TEXT_DOMAIN, "cannot set permissions on '%s'",
4403                 zc.zc_name);
4404         switch (errno) {
4405             case ENOTSUP:
4406                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4407                     "pool must be upgraded"));
4408                 err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4409                 break;
4410             case EINVAL:
4411                 err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4412                 break;
4413             case ENOENT:
4414                 err = zfs_error(hdl, EZFS_NOENT, errbuf);
4415                 break;
4416             default:
4417                 err = zfs_standard_error_fmt(hdl, errno, errbuf);
4418                 break;
4419         }
4420     }
4422     free(nvbuf);
4424
4424     return (err);
4425 }
4427 int
4428 zfs_get_holds(zfs_handle_t *zhp, nvlist_t **nvl)
4429 {
4430     int err;
4431     char errbuf[1024];
4433
4433     err = lzc_get_holds(zhp->zfs_name, nvl);
4435
4435     if (err != 0) {
4436         libzfs_handle_t *hdl = zhp->zfs_hdl;
4438
4438         (void) snprintf(errbuf, sizeof(errbuf),
4439             dgettext(TEXT_DOMAIN, "cannot get holds for '%s'",
4440                 zhp->zfs_name);
4441         switch (err) {
4442             case ENOTSUP:
4443                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4444                     "pool must be upgraded"));
4445                 err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4446                 break;
4447             case EINVAL:
4448                 err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4449                 break;
4450             case ENOENT:
4451                 err = zfs_error(hdl, EZFS_NOENT, errbuf);

```

```
4452             break;
4453         default:
4454             err = zfs_standard_error_fmt(hdl, errno, errbuf);
4455             break;
4456     }
4457 }
4459     return (err);
4460 }

4462 uint64_t
4463 zvol_volsize_to_reservation(uint64_t volsize, nvlist_t *props)
4464 {
4465     uint64_t numdb;
4466     uint64_t nblocks, volblocksize;
4467     int ncopies;
4468     char *strval;

4470     if (nvlist_lookup_string(props,
4471         zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)
4472         ncopies = atoi(strval);
4473     else
4474         ncopies = 1;
4475     if (nvlist_lookup_uint64(props,
4476         zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
4477         &volblocksize) != 0)
4478         volblocksize = ZVOL_DEFAULT_BLOCKSIZE;
4479     nblocks = volsize/volblocksize;
4480     /* start with metadnode L0-L6 */
4481     numdb = 7;
4482     /* calculate number of indirects */
4483     while (nblocks > 1) {
4484         nblocks += DNODES_PER_LEVEL - 1;
4485         nblocks /= DNODES_PER_LEVEL;
4486         numdb += nblocks;
4487     }
4488     numdb *= MIN(SPA_DVAS_PER_BP, ncopies + 1);
4489     volsize *= ncopies;
4490     /*
4491      * this is exactly DN_MAX_INDBLKSHIFT when metadata isn't
4492      * compressed, but in practice they compress down to about
4493      * 1100 bytes
4494      */
4495     numdb *= 1ULL << DN_MAX_INDBLKSHIFT;
4496     volsize += numdb;
4497     return (volsize);
4498 }
```

```

*****
84531 Wed May 29 20:27:08 2013
new/usr/src/lib/libzfs/common/libzfs_sendrecv.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
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, Joyent, Inc. All rights reserved.
26 * Copyright (c) 2013 Steven Hartland. All rights reserved.
27 #endif /* ! codereview */
28 */

30 #include <assert.h>
31 #include <ctype.h>
32 #include <errno.h>
33 #include <libintl.h>
34 #include <stdio.h>
35 #include <stdlib.h>
36 #include <strings.h>
37 #include <unistd.h>
38 #include <stddef.h>
39 #include <fcntl.h>
40 #include <sys/mount.h>
41 #include <pthread.h>
42 #include <umem.h>
43 #include <time.h>

45 #include <libzfs.h>

47 #include "zfs_namecheck.h"
48 #include "zfs_prop.h"
49 #include "zfs_fletcher.h"
50 #include "libzfs_impl.h"
51 #include <sha2.h>
52 #include <sys/zio_checksum.h>
53 #include <sys/ddt.h>

55 /* in libzfs_dataset.c */
56 extern void zfs_setprop_error(libzfs_handle_t *, zfs_prop_t, int, char *);

58 static int zfs_receive_impl(libzfs_handle_t *, const char *, rcvflags_t *,
59     int, const char *, nvlist_t *, avl_tree_t *, char **, int, uint64_t *);

```

```

61 static const zio_cksum_t zero_cksum = { 0 };

63 typedef struct dedup_arg {
64     int inputfd;
65     int outputfd;
66     libzfs_handle_t *dedup_hdl;
67 } dedup_arg_t;

69 typedef struct progress_arg {
70     zfs_handle_t *pa_zhp;
71     int pa_fd;
72     boolean_t pa_parsable;
73 } progress_arg_t;

75 typedef struct dataref {
76     uint64_t ref_guid;
77     uint64_t ref_object;
78     uint64_t ref_offset;
79 } dataref_t;

81 typedef struct dedup_entry {
82     struct dedup_entry *dde_next;
83     zio_cksum_t dde_cksum;
84     uint64_t dde_prop;
85     dataref_t dde_ref;
86 } dedup_entry_t;

88 #define MAX_DDT_PHYSMEM_PERCENT 20
89 #define SMALLEST_POSSIBLE_MAX_DDT_MB 128

91 typedef struct dedup_table {
92     dedup_entry_t **dedup_hash_array;
93     umem_cache_t *ddecache;
94     uint64_t max_ddt_size; /* max dedup table size in bytes */
95     uint64_t cur_ddt_size; /* current dedup table size in bytes */
96     uint64_t ddt_count;
97     int numhashbits;
98     boolean_t ddt_full;
99 } dedup_table_t;

101 static int
102 high_order_bit(uint64_t n)
103 {
104     int count;

106     for (count = 0; n != 0; count++)
107         n >>= 1;
108     return (count);
109 }

111 static size_t
112 sread(void *buf, size_t len, FILE *stream)
113 {
114     size_t outlen;

116     if ((outlen = fread(buf, len, 1, stream)) == 0)
117         return (0);

119     return (outlen);
120 }

122 static void
123 ddt_hash_append(libzfs_handle_t *hdl, dedup_table_t *ddt, dedup_entry_t **ddepp,
124     zio_cksum_t *cs, uint64_t prop, dataref_t *dr)
125 {
126     dedup_entry_t *dde;

```

```

128     if (ddt->cur_ddt_size >= ddt->max_ddt_size) {
129         if (ddt->ddt_full == B_FALSE) {
130             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
131                 "Dedup table full. Deduplication will continue "
132                 "with existing table entries"));
133             ddt->ddt_full = B_TRUE;
134         }
135         return;
136     }

138     if ((dde = umem_cache_alloc(ddt->ddecache, UMEM_DEFAULT))
139         != NULL) {
140         assert(*ddepp == NULL);
141         dde->dde_next = NULL;
142         dde->dde_chksum = *cs;
143         dde->dde_prop = prop;
144         dde->dde_ref = *dr;
145         *ddepp = dde;
146         ddt->cur_ddt_size += sizeof (dedup_entry_t);
147         ddt->ddt_count++;
148     }
149 }

151 /*
152  * Using the specified dedup table, do a lookup for an entry with
153  * the checksum cs. If found, return the block's reference info
154  * in *dr. Otherwise, insert a new entry in the dedup table, using
155  * the reference information specified by *dr.
156  *
157  * return value: true - entry was found
158  *               false - entry was not found
159  */
160 static boolean_t
161 ddt_update(libzfs_handle_t *hdl, dedup_table_t *ddt, zio_cksum_t *cs,
162     uint64_t prop, dataref_t *dr)
163 {
164     uint32_t hashcode;
165     dedup_entry_t **ddepp;

167     hashcode = BF64_GET(cs->zc_word[0], 0, ddt->numhashbits);

169     for (ddepp = &(ddt->dedup_hash_array[hashcode]); *ddepp != NULL;
170         ddepp = &((*ddepp)->dde_next)) {
171         if (ZIO_CHECKSUM_EQUAL((*ddepp)->dde_chksum), *cs) &&
172             (*ddepp)->dde_prop == prop) {
173             *dr = (*ddepp)->dde_ref;
174             return (B_TRUE);
175         }
176     }
177     ddt_hash_append(hdl, ddt, ddepp, cs, prop, dr);
178     return (B_FALSE);
179 }

181 static int
182 cksum_and_write(const void *buf, uint64_t len, zio_cksum_t *zc, int outfd)
183 {
184     fletcher_4_incremental_native(buf, len, zc);
185     return (write(outfd, buf, len));
186 }

188 /*
189  * This function is started in a separate thread when the dedup option
190  * has been requested. The main send thread determines the list of
191  * snapshots to be included in the send stream and makes the ioctl calls
192  * for each one. But instead of having the ioctl send the output to the

```

```

193  * the output fd specified by the caller of zfs_send(), the
194  * ioctl is told to direct the output to a pipe, which is read by the
195  * alternate thread running THIS function. This function does the
196  * dedup'ing by:
197  * 1. building a dedup table (the DDT)
198  * 2. doing checksums on each data block and inserting a record in the DDT
199  * 3. looking for matching checksums, and
200  * 4. sending a DRR_WRITE_BYREF record instead of a write record whenever
201  *    a duplicate block is found.
202  * The output of this function then goes to the output fd requested
203  * by the caller of zfs_send().
204  */
205 static void *
206 cksummer(void *arg)
207 {
208     dedup_arg_t *dda = arg;
209     char *buf = malloc(1<<20);
210     dmu_replay_record_t thedrr;
211     dmu_replay_record_t *drr = &thedrr;
212     struct drr_begin *drrb = &thedrr.drr_u.drr_begin;
213     struct drr_end *drr_e = &thedrr.drr_u.drr_end;
214     struct drr_object *drr_o = &thedrr.drr_u.drr_object;
215     struct drr_write *drr_w = &thedrr.drr_u.drr_write;
216     struct drr_spill *drr_s = &thedrr.drr_u.drr_spill;
217     FILE *ofp;
218     int outfd;
219     dmu_replay_record_t wbr_drr = {0};
220     struct drr_write_byref *wbr_drrr = &wbr_drr.drr_u.drr_write_byref;
221     dedup_table_t ddt;
222     zio_cksum_t stream_cksum;
223     uint64_t physmem = sysconf(_SC_PHYS_PAGES) * sysconf(_SC_PAGESIZE);
224     uint64_t numbuckets;

226     ddt.max_ddt_size =
227         MAX((physmem * MAX_DDT_PHYSMEM_PERCENT)/100,
228             SMALLEST_POSSIBLE_MAX_DDT_MB<<20);

230     numbuckets = ddt.max_ddt_size/(sizeof (dedup_entry_t));

232     /*
233      * numbuckets must be a power of 2. Increase number to
234      * a power of 2 if necessary.
235      */
236     if (!ISP2(numbuckets))
237         numbuckets = 1 << high_order_bit(numbuckets);

239     ddt.dedup_hash_array = calloc(numbuckets, sizeof (dedup_entry_t *));
240     ddt.ddecache = umem_cache_create("dde", sizeof (dedup_entry_t), 0,
241         NULL, NULL, NULL, NULL, NULL, 0);
242     ddt.cur_ddt_size = numbuckets * sizeof (dedup_entry_t *);
243     ddt.numhashbits = high_order_bit(numbuckets) - 1;
244     ddt.ddt_full = B_FALSE;

246     /* Initialize the write-by-reference block. */
247     wbr_drr.drr_type = DRR_WRITE_BYREF;
248     wbr_drr.drr_payloadlen = 0;

250     outfd = dda->outputfd;
251     ofp = fdopen(dda->inputfd, "r");
252     while (ssread(drr, sizeof (dmu_replay_record_t), ofp) != 0) {

254         switch (drr->drr_type) {
255             case DRR_BEGIN:
256             {
257                 int fflags;
258                 ZIO_SET_CHECKSUM(&stream_cksum, 0, 0, 0, 0);

```

```

260     /* set the DEDUP feature flag for this stream */
261     fflags = DMU_GET_FEATUREFLAGS(drrb->drr_versioninfo);
262     fflags |= (DMU_BACKUP_FEATURE_DEDUP |
263              DMU_BACKUP_FEATURE_DEDUPPROPS);
264     DMU_SET_FEATUREFLAGS(drrb->drr_versioninfo, fflags);

266     if (cksum_and_write(drr, sizeof (dmu_replay_record_t),
267                        &stream_cksum, outfd) == -1)
268         goto out;
269     if (DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) ==
270         DMU_COMPOUNDSTREAM && drr->drr_payloadlen != 0) {
271         int sz = drr->drr_payloadlen;

273         if (sz > 1<<20) {
274             free(buf);
275             buf = malloc(sz);
276         }
277         (void) ssread(buf, sz, ofp);
278         if (ferror(stdin))
279             perror("fread");
280         if (cksum_and_write(buf, sz, &stream_cksum,
281                           outfd) == -1)
282             goto out;
283     }
284     break;
285 }

287 case DRR_END:
288 {
289     /* use the recalculated checksum */
290     ZIO_SET_CHECKSUM(&drr->drr_checksum,
291                    stream_cksum.zc_word[0], stream_cksum.zc_word[1],
292                    stream_cksum.zc_word[2], stream_cksum.zc_word[3]);
293     if ((write(outfd, drr,
294              sizeof (dmu_replay_record_t))) == -1)
295         goto out;
296     break;
297 }

299 case DRR_OBJECT:
300 {
301     if (cksum_and_write(drr, sizeof (dmu_replay_record_t),
302                        &stream_cksum, outfd) == -1)
303         goto out;
304     if (drr->drr_bonuslen > 0) {
305         (void) ssread(buf,
306                     P2ROUNDUP((uint64_t)drr->drr_bonuslen, 8),
307                     ofp);
308         if (cksum_and_write(buf,
309                             P2ROUNDUP((uint64_t)drr->drr_bonuslen, 8),
310                             &stream_cksum, outfd) == -1)
311             goto out;
312     }
313     break;
314 }

316 case DRR_SPILL:
317 {
318     if (cksum_and_write(drr, sizeof (dmu_replay_record_t),
319                        &stream_cksum, outfd) == -1)
320         goto out;
321     (void) ssread(buf, drr->drr_length, ofp);
322     if (cksum_and_write(buf, drr->drr_length,
323                        &stream_cksum, outfd) == -1)
324         goto out;

```

```

325         break;
326     }

328 case DRR_FREEOBJECTS:
329 {
330     if (cksum_and_write(drr, sizeof (dmu_replay_record_t),
331                        &stream_cksum, outfd) == -1)
332         goto out;
333     break;
334 }

336 case DRR_WRITE:
337 {
338     dataref_t     dataref;

340     (void) ssread(buf, drr->drr_length, ofp);

342     /*
343      * Use the existing checksum if it's dedup-capable,
344      * else calculate a SHA256 checksum for it.
345      */

347     if (ZIO_CHECKSUM_EQUAL(drr->drr_key.ddk_cksum,
348                           zero_cksum) ||
349         !DRR_IS_DEDUP_CAPABLE(drr->drr_checksumflags)) {
350         SHA256_CTX     ctx;
351         zio_cksum_t     tmpsha256;

353         SHA256Init(&ctx);
354         SHA256Update(&ctx, buf, drr->drr_length);
355         SHA256Final(&tmpsha256, &ctx);
356         drr->drr_key.ddk_cksum.zc_word[0] =
357             BE_64(tmpsha256.zc_word[0]);
358         drr->drr_key.ddk_cksum.zc_word[1] =
359             BE_64(tmpsha256.zc_word[1]);
360         drr->drr_key.ddk_cksum.zc_word[2] =
361             BE_64(tmpsha256.zc_word[2]);
362         drr->drr_key.ddk_cksum.zc_word[3] =
363             BE_64(tmpsha256.zc_word[3]);
364         drr->drr_checksumtype = ZIO_CHECKSUM_SHA256;
365         drr->drr_checksumflags = DRR_CHECKSUM_DEDUP;
366     }

368     dataref.ref_guid = drr->drr_toguid;
369     dataref.ref_object = drr->drr_object;
370     dataref.ref_offset = drr->drr_offset;

372     if (drr->drr_dedup_hdl != NULL) {
373         drr->drr_key.ddk_cksum, drr->drr_key.ddk_prop,
374         &dataref) {
375             /* block already present in stream */
376             wbr_drr->drr_object = drr->drr_object;
377             wbr_drr->drr_offset = drr->drr_offset;
378             wbr_drr->drr_length = drr->drr_length;
379             wbr_drr->drr_toguid = drr->drr_toguid;
380             wbr_drr->drr_refguid = dataref.ref_guid;
381             wbr_drr->drr_refobject =
382                 dataref.ref_object;
383             wbr_drr->drr_refoffset =
384                 dataref.ref_offset;

386             wbr_drr->drr_checksumtype =
387                 drr->drr_checksumtype;
388             wbr_drr->drr_checksumflags =
389                 drr->drr_checksumflags;
390             wbr_drr->drr_key.ddk_cksum =

```

```

391         drrw->drr_key.ddk_cksum;
392         wbr_drrr->drr_key.ddk_prop =
393         drrw->drr_key.ddk_prop;

395         if (cksum_and_write(&wbr_drr,
396         sizeof (dmu_replay_record_t), &stream_cksum,
397         outfd) == -1)
398             goto out;
399     } else {
400         /* block not previously seen */
401         if (cksum_and_write(drr,
402         sizeof (dmu_replay_record_t), &stream_cksum,
403         outfd) == -1)
404             goto out;
405         if (cksum_and_write(buf,
406         drrw->drr_length,
407         &stream_cksum, outfd) == -1)
408             goto out;
409     }
410     break;
411 }

413 case DRR_FREE:
414 {
415     if (cksum_and_write(drr, sizeof (dmu_replay_record_t),
416     &stream_cksum, outfd) == -1)
417         goto out;
418     break;
419 }

421 default:
422     (void) printf("INVALID record type 0x%x\n",
423     drr->drr_type);
424     /* should never happen, so assert */
425     assert(B_FALSE);
426 }
427 }
428 out:
429     umem_cache_destroy(ddt.ddecache);
430     free(ddt.dedup_hash_array);
431     free(buf);
432     (void) fclose(ofp);

434     return (NULL);
435 }

437 /*
438  * Routines for dealing with the AVL tree of fs-nvlists
439  */
440 typedef struct fsavl_node {
441     avl_node_t fn_node;
442     nvlist_t *fn_nvfs;
443     char *fn_snapname;
444     uint64_t fn_guid;
445 } fsavl_node_t;

447 static int
448 fsavl_compare(const void *arg1, const void *arg2)
449 {
450     const fsavl_node_t *fn1 = arg1;
451     const fsavl_node_t *fn2 = arg2;

453     if (fn1->fn_guid > fn2->fn_guid)
454         return (+1);
455     else if (fn1->fn_guid < fn2->fn_guid)
456         return (-1);

```

```

457     else
458         return (0);
459 }

461 /*
462  * Given the GUID of a snapshot, find its containing filesystem and
463  * (optionally) name.
464  */
465 static nvlist_t *
466 fsavl_find(avl_tree_t *avl, uint64_t snapguid, char **snapname)
467 {
468     fsavl_node_t fn_find;
469     fsavl_node_t *fn;

471     fn_find.fn_guid = snapguid;

473     fn = avl_find(avl, &fn_find, NULL);
474     if (fn) {
475         if (snapname)
476             *snapname = fn->fn_snapname;
477         return (fn->fn_nvfs);
478     }
479     return (NULL);
480 }

482 static void
483 fsavl_destroy(avl_tree_t *avl)
484 {
485     fsavl_node_t *fn;
486     void *cookie;

488     if (avl == NULL)
489         return;

491     cookie = NULL;
492     while ((fn = avl_destroy_nodes(avl, &cookie)) != NULL)
493         free(fn);
494     avl_destroy(avl);
495     free(avl);
496 }

498 /*
499  * Given an nvlist, produce an avl tree of snapshots, ordered by guid
500  */
501 static avl_tree_t *
502 fsavl_create(nvlist_t *fss)
503 {
504     avl_tree_t *fsavl;
505     nvpair_t *fselem = NULL;

507     if ((fsavl = malloc(sizeof (avl_tree_t))) == NULL)
508         return (NULL);

510     avl_create(fsavl, fsavl_compare, sizeof (fsavl_node_t),
511     offsetof(fsavl_node_t, fn_node));

513     while ((fselem = nvlist_next_nvpair(fss, fselem)) != NULL) {
514         nvlist_t *nvfs, *snaps;
515         nvpair_t *snapelem = NULL;

517         VERIFY(0 == nvpair_value_nvlist(fselem, &nvfs));
518         VERIFY(0 == nvlist_lookup_nvlist(nvfs, "snaps", &snaps));

520         while ((snapelem =
521         nvlist_next_nvpair(snaps, snapelem)) != NULL) {
522             fsavl_node_t *fn;

```

```

523         uint64_t guid;

525         VERIFY(0 == nvpair_value_uint64(snapelem, &guid));
526         if ((fn = malloc(sizeof (fsavl_node_t))) == NULL) {
527             fsavl_destroy(fsavl);
528             return (NULL);
529         }
530         fn->fn_nvfs = nvfs;
531         fn->fn_snapname = nvpair_name(snapelem);
532         fn->fn_guid = guid;

534         /*
535          * Note: if there are multiple snaps with the
536          * same GUID, we ignore all but one.
537          */
538         if (avl_find(fsavl, fn, NULL) == NULL)
539             avl_add(fsavl, fn);
540         else
541             free(fn);
542     }
543 }

545     return (fsavl);
546 }

548 /*
549  * Routines for dealing with the giant nvlist of fs-nvlists, etc.
550  */
551 typedef struct send_data {
552     uint64_t parent_fromsnap_guid;
553     nvlist_t *parent_snaps;
554     nvlist_t *fss;
555     nvlist_t *snapprops;
556     const char *fromsnap;
557     const char *tosnap;
558     boolean_t recursive;

560     /*
561      * The header nvlist is of the following format:
562      * {
563      *   "tosnap" -> string
564      *   "fromsnap" -> string (if incremental)
565      *   "fss" -> {
566      *     id -> {
567      *
568      *       "name" -> string (full name; for debugging)
569      *       "parentfromsnap" -> number (guid of fromsnap in parent)
570      *
571      *       "props" -> { name -> value (only if set here) }
572      *       "snaps" -> { name (lastname) -> number (guid) }
573      *       "snapprops" -> { name (lastname) -> { name -> value } }
574      *
575      *       "origin" -> number (guid) (if clone)
576      *       "sent" -> boolean (not on-disk)
577      *     }
578      *   }
579     */
580 } send_data_t;

584 static void send_iterate_prop(zfs_handle_t *zhp, nvlist_t *nv);

586 static int
587 send_iterate_snap(zfs_handle_t *zhp, void *arg)
588 {

```

```

589     send_data_t *sd = arg;
590     uint64_t guid = zhp->zfs_dmstats.dds_guid;
591     char *snapname;
592     nvlist_t *nv;

594     snapname = strrchr(zhp->zfs_name, '@')+1;

596     VERIFY(0 == nvlist_add_uint64(sd->parent_snaps, snapname, guid));
597     /*
598      * NB: if there is no fromsnap here (it's a newly created fs in
599      * an incremental replication), we will substitute the tosnap.
600      */
601     if ((sd->fromsnap && strcmp(snapname, sd->fromsnap) == 0) ||
602         (sd->parent_fromsnap_guid == 0 && sd->tosnap &&
603          strcmp(snapname, sd->tosnap) == 0)) {
604         sd->parent_fromsnap_guid = guid;
605     }

607     VERIFY(0 == nvlist_alloc(&nv, NV_UNIQUE_NAME, 0));
608     send_iterate_prop(zhp, nv);
609     VERIFY(0 == nvlist_add_nvlist(sd->snapprops, snapname, nv));
610     nvlist_free(nv);

612     zfs_close(zhp);
613     return (0);
614 }

616 static void
617 send_iterate_prop(zfs_handle_t *zhp, nvlist_t *nv)
618 {
619     nvpair_t *elem = NULL;

621     while ((elem = nvlist_next_nvpair(zhp->zfs_props, elem)) != NULL) {
622         char *propname = nvpair_name(elem);
623         zfs_prop_t prop = zfs_name_to_prop(propname);
624         nvlist_t *propnv;

626         if (!zfs_prop_user(propname)) {
627             /*
628              * Realistically, this should never happen. However,
629              * we want the ability to add DSL properties without
630              * needing to make incompatible version changes. We
631              * need to ignore unknown properties to allow older
632              * software to still send datasets containing these
633              * properties, with the unknown properties elided.
634              */
635             if (prop == ZPROP_INVALID)
636                 continue;

638             if (zfs_prop_readonly(prop))
639                 continue;
640         }

642         verify(nvpair_value_nvlist(elem, &propnv) == 0);
643         if (prop == ZFS_PROP_QUOTA || prop == ZFS_PROP_RESERVATION ||
644             prop == ZFS_PROP_REFQUOTA ||
645             prop == ZFS_PROP_REFRESERVATION) {
646             char *source;
647             uint64_t value;
648             verify(nvlist_lookup_uint64(propnv,
649                 ZPROP_VALUE, &value) == 0);
650             if (zhp->zfs_type == ZFS_TYPE_SNAPSHOT)
651                 continue;
652             /*
653              * May have no source before SPA_VERSION_RECVD_PROPS,
654              * but is still modifiable.

```

```

655     */
656     if (nvlist_lookup_string(propnv,
657         ZPROP_SOURCE, &source) == 0) {
658         if ((strcmp(source, zhp->zfs_name) != 0) &&
659             (strcmp(source,
660                 ZPROP_SOURCE_VAL_RECVD) != 0))
661             continue;
662     }
663     } else {
664         char *source;
665         if (nvlist_lookup_string(propnv,
666             ZPROP_SOURCE, &source) != 0)
667             continue;
668         if ((strcmp(source, zhp->zfs_name) != 0) &&
669             (strcmp(source, ZPROP_SOURCE_VAL_RECVD) != 0))
670             continue;
671     }
672
673     if (zfs_prop_user(propname) ||
674         zfs_prop_get_type(prop) == PROP_TYPE_STRING) {
675         char *value;
676         verify(nvlist_lookup_string(propnv,
677             ZPROP_VALUE, &value) == 0);
678         VERIFY(0 == nvlist_add_string(nv, propname, value));
679     } else {
680         uint64_t value;
681         verify(nvlist_lookup_uint64(propnv,
682             ZPROP_VALUE, &value) == 0);
683         VERIFY(0 == nvlist_add_uint64(nv, propname, value));
684     }
685 }
686
687 /*
688  * recursively generate nvlists describing datasets. See comment
689  * for the data structure send_data_t above for description of contents
690  * of the nvlist.
691  */
692 static int
693 send_iterate_fs(zfs_handle_t *zhp, void *arg)
694 {
695     send_data_t *sd = arg;
696     nvlist_t *nvfs, *nv;
697     int rv = 0;
698     uint64_t parent_fromsnap_guid_save = sd->parent_fromsnap_guid;
699     uint64_t guid = zhp->zfs_dmustats.dds_guid;
700     char guidstring[64];
701
702     VERIFY(0 == nvlist_alloc(&nvfs, NV_UNIQUE_NAME, 0));
703     VERIFY(0 == nvlist_add_string(nvfs, "name", zhp->zfs_name));
704     VERIFY(0 == nvlist_add_uint64(nvfs, "parentfromsnap",
705         sd->parent_fromsnap_guid));
706
707     if (zhp->zfs_dmustats.dds_origin[0]) {
708         zfs_handle_t *origin = zfs_open(zhp->zfs_hdl,
709             zhp->zfs_dmustats.dds_origin, ZFS_TYPE_SNAPSHOT);
710         if (origin == NULL)
711             return (-1);
712         VERIFY(0 == nvlist_add_uint64(nvfs, "origin",
713             origin->zfs_dmustats.dds_guid));
714     }
715
716     /* iterate over props */
717     VERIFY(0 == nvlist_alloc(&nv, NV_UNIQUE_NAME, 0));
718     send_iterate_prop(zhp, nv);
719     VERIFY(0 == nvlist_add_nvlist(nvfs, "props", nv));

```

```

721     nvlist_free(nv);
722
723     /* iterate over snaps, and set sd->parent_fromsnap_guid */
724     sd->parent_fromsnap_guid = 0;
725     VERIFY(0 == nvlist_alloc(&sd->parent_snaps, NV_UNIQUE_NAME, 0));
726     VERIFY(0 == nvlist_alloc(&sd->snapprops, NV_UNIQUE_NAME, 0));
727     (void) zfs_iter_snapshots(zhp, send_iterate_snap, sd);
728     VERIFY(0 == nvlist_add_nvlist(nvfs, "snaps", sd->parent_snaps));
729     VERIFY(0 == nvlist_add_nvlist(nvfs, "snapprops", sd->snapprops));
730     nvlist_free(sd->parent_snaps);
731     nvlist_free(sd->snapprops);
732
733     /* add this fs to nvlist */
734     (void) snprintf(guidstring, sizeof(guidstring),
735         "0x%llx", (longlong_t)guid);
736     VERIFY(0 == nvlist_add_nvlist(sd->fss, guidstring, nvfs));
737     nvlist_free(nvfs);
738
739     /* iterate over children */
740     if (sd->recursive)
741         rv = zfs_iter_filesystems(zhp, send_iterate_fs, sd);
742
743     sd->parent_fromsnap_guid = parent_fromsnap_guid_save;
744
745     zfs_close(zhp);
746     return (rv);
747 }
748
749 static int
750 gather_nvlist(libzfs_handle_t *hdl, const char *fsname, const char *fromsnap,
751     const char *tosnap, boolean_t recursive, nvlist_t **nvlp, avl_tree_t **avlp)
752 {
753     zfs_handle_t *zhp;
754     send_data_t sd = { 0 };
755     int error;
756
757     zhp = zfs_open(hdl, fsname, ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
758     if (zhp == NULL)
759         return (EZFS_BADTYPE);
760
761     VERIFY(0 == nvlist_alloc(&sd.fss, NV_UNIQUE_NAME, 0));
762     sd.fromsnap = fromsnap;
763     sd.tosnap = tosnap;
764     sd.recursive = recursive;
765
766     if ((error = send_iterate_fs(zhp, &sd)) != 0) {
767         nvlist_free(sd.fss);
768         if (avlp != NULL)
769             *avlp = NULL;
770         *nvlp = NULL;
771         return (error);
772     }
773
774     if (avlp != NULL && (*avlp = fsavl_create(sd.fss)) == NULL) {
775         nvlist_free(sd.fss);
776         *nvlp = NULL;
777         return (EZFS_NOMEM);
778     }
779
780     *nvlp = sd.fss;
781     return (0);
782 }
783
784 /*
785  * Routines specific to "zfs send"
786  */

```



```

787 typedef struct send_dump_data {
788     /* these are all just the short snapname (the part after the @) */
789     const char *fromsnap;
790     const char *tosnap;
791     char prevsnap[ZFS_MAXNAMELEN];
792     uint64_t prevsnap_obj;
793     boolean_t seenfrom, seento, replicate, doall, fromorigin;
794     boolean_t verbose, dryrun, parsable, progress;
795     int outfd;
796     boolean_t err;
797     nvlist_t *fss;
798     nvlist_t *snapholds;
799 #endif /* ! codereview */
800     avl_tree_t *fsavl;
801     snapfilter_cb_t *filter_cb;
802     void *filter_cb_arg;
803     nvlist_t *debugnv;
804     char holdtag[ZFS_MAXNAMELEN];
805     int cleanup_fd;
806     uint64_t size;
807 } send_dump_data_t;

809 static int
810 estimate_ioctl(zfs_handle_t *zhp, uint64_t fromsnap_obj,
811               boolean_t fromorigin, uint64_t *sizep)
812 {
813     zfs_cmd_t zc = { 0 };
814     libzfs_handle_t *hdl = zhp->zfs_hdl;

816     assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
817     assert(fromsnap_obj == 0 || !fromorigin);

819     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
820     zc.zc_obj = fromorigin;
821     zc.zc_sendobj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
822     zc.zc_fromobj = fromsnap_obj;
823     zc.zc_guid = 1; /* estimate flag */

825     if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_SEND, &zc) != 0) {
826         char errbuf[1024];
827         (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
828             "warning: cannot estimate space for '%s'", zhp->zfs_name));

830         switch (errno) {
831             case EXDEV:
832                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
833                     "not an earlier snapshot from the same fs"));
834                 return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));

836             case ENOENT:
837                 if (zfs_dataset_exists(hdl, zc.zc_name,
838                     ZFS_TYPE_SNAPSHOT)) {
839                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
840                         "incremental source (@%s) does not exist"),
841                         zc.zc_value);
842                 }
843                 return (zfs_error(hdl, EZFS_NOENT, errbuf));

845             case EDQUOT:
846             case EFBIG:
847             case EIO:
848             case ENOLINK:
849             case ENOSPC:
850             case ENOSTR:
851             case ENXIO:
852             case EPIPE:

```

```

853             case ERANGE:
854             case EFAULT:
855             case EROFS:
856                 zfs_error_aux(hdl, strerror(errno));
857                 return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));

859             default:
860                 return (zfs_standard_error(hdl, errno, errbuf));
861         }
862     }

864     *sizep = zc.zc_objset_type;

866     return (0);
867 }

869 /*
870  * Dumps a backup of the given snapshot (incremental from fromsnap if it's not
871  * NULL) to the file descriptor specified by outfd.
872  */
873 static int
874 dump_ioctl(zfs_handle_t *zhp, const char *fromsnap, uint64_t fromsnap_obj,
875            boolean_t fromorigin, int outfd, nvlist_t *debugnv)
876 {
877     zfs_cmd_t zc = { 0 };
878     libzfs_handle_t *hdl = zhp->zfs_hdl;
879     nvlist_t *thisdbg;

881     assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
882     assert(fromsnap_obj == 0 || !fromorigin);

884     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof(zc.zc_name));
885     zc.zc_cookie = outfd;
886     zc.zc_obj = fromorigin;
887     zc.zc_sendobj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
888     zc.zc_fromobj = fromsnap_obj;

890     VERIFY(0 == nvlist_alloc(&thisdbg, NV_UNIQUE_NAME, 0));
891     if (fromsnap && fromsnap[0] != '\0') {
892         VERIFY(0 == nvlist_add_string(thisdbg,
893             "fromsnap", fromsnap));
894     }

896     if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_SEND, &zc) != 0) {
897         char errbuf[1024];
898         (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
899             "warning: cannot send '%s'", zhp->zfs_name));

901         VERIFY(0 == nvlist_add_uint64(thisdbg, "error", errno));
902         if (debugnv) {
903             VERIFY(0 == nvlist_add_nvlist(debugnv,
904                 zhp->zfs_name, thisdbg));
905         }
906         nvlist_free(thisdbg);

908         switch (errno) {
909             case EXDEV:
910                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
911                     "not an earlier snapshot from the same fs"));
912                 return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));

914             case ENOENT:
915                 if (zfs_dataset_exists(hdl, zc.zc_name,
916                     ZFS_TYPE_SNAPSHOT)) {
917                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
918                         "incremental source (@%s) does not exist"),

```

```

919         zc.zc_value);
920     }
921     return (zfs_error(hdl, EZFS_NOENT, errbuf));

923     case EDQUOT:
924     case EFBIG:
925     case EIO:
926     case ENOLINK:
927     case ENOSPC:
928     case ENOSTR:
929     case ENXIO:
930     case EPIPE:
931     case ERANGE:
932     case EFAULT:
933     case EROFS:
934         zfs_error_aux(hdl, strerror(errno));
935         return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));

937     default:
938         return (zfs_standard_error(hdl, errno, errbuf));
939     }
940 }

942 if (debugnv)
943     VERIFY(0 == nvlist_add_nvlist(debugnv, zhp->zfs_name, thisdbg));
944 nvlist_free(thisdbg);

946     return (0);
947 }

949 static void
950 gather_holds(zfs_handle_t *zhp, send_dump_data_t *sdd)
951 {
952     static int
953     hold_for_send(zfs_handle_t *zhp, send_dump_data_t *sdd)
954     {
955         zfs_handle_t *pzhp;
956         int error = 0;
957         char *thissnap;

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

959         if (sdd->dryrun)
960             return (0);

961         /*
962          * zfs_send() only sets snapholds for sends that need them,
963          * zfs_send() only opens a cleanup_fd for sends that need it,
964          * e.g. replication and doall.
965          */
966         if (sdd->snapholds == NULL)
967             return;
968         if (sdd->cleanup_fd == -1)
969             return (0);

970         fnvlist_add_string(sdd->snapholds, zhp->zfs_name, sdd->holdtag);
971         thissnap = strchr(zhp->zfs_name, '@') + 1;
972         *(thissnap - 1) = '\0';
973         pzhp = zfs_open(zhp->zfs_hdl, zhp->zfs_name, ZFS_TYPE_DATASET);
974         *(thissnap - 1) = '@';

975         /*
976          * It's OK if the parent no longer exists. The send code will
977          * handle that error.
978          */
979         if (pzhp) {
980             error = zfs_hold(pzhp, thissnap, sdd->holdtag,

```

```

981         B_FALSE, B_TRUE, sdd->cleanup_fd);
982         zfs_close(pzhp);
983     }

984     return (error);
985 }

986     unchanged_portion_omitted

1011 static int
1012 dump_snapshot(zfs_handle_t *zhp, void *arg)
1013 {
1014     send_dump_data_t *sdd = arg;
1015     progress_arg_t pa = { 0 };
1016     pthread_t tid;

1017     char *thissnap;
1018     int err;
1019     boolean_t isfromsnap, istosnap, fromorigin;
1020     boolean_t exclude = B_FALSE;

1021     err = 0;
1022 #endif /* !codereview */
1023     thissnap = strchr(zhp->zfs_name, '@') + 1;
1024     isfromsnap = (sdd->fromsnap != NULL &&
1025                 strcmp(sdd->fromsnap, thissnap) == 0);

1026     if (!sdd->seenfrom && isfromsnap) {
1027         gather_holds(zhp, sdd);
1028         err = hold_for_send(zhp, sdd);
1029         if (err == 0) {
1030             sdd->seenfrom = B_TRUE;
1031             (void) strcpy(sdd->prevsnap, thissnap);
1032             sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
1033             sdd->prevsnap_obj = zfs_prop_get_int(zhp,
1034                                                 ZFS_PROP_OBJSETID);
1035         } else if (err == ENOENT) {
1036             err = 0;
1037         }
1038         zfs_close(zhp);
1039         return (0);
1040         return (err);
1041     }

1042     if (sdd->seento || !sdd->seenfrom) {
1043         zfs_close(zhp);
1044         return (0);
1045     }

1046     istosnap = (strcmp(sdd->tosnap, thissnap) == 0);
1047     if (istosnap)
1048         sdd->seento = B_TRUE;

1049     if (!sdd->doall && !isfromsnap && !istosnap) {
1050         if (sdd->replicate) {
1051             char *snapname;
1052             nvlist_t *snapprops;
1053             /*
1054              * Filter out all intermediate snapshots except origin
1055              * snapshots needed to replicate clones.
1056              */
1057             nvlist_t *nvfs = fsavl_find(sdd->fsavl,
1058                                         zhp->zfs_dmustats.dds_guid, &snapname);

1059             VERIFY(0 == nvlist_lookup_nvlist(nvfs,
1060                                             "snapprops", &snapprops));
1061             VERIFY(0 == nvlist_lookup_nvlist(snapprops,

```

```

1060         thissnap, &snapprops));
1061         exclude = !nvlist_exists(snapprops, "is_clone_origin");
1062     } else {
1063         exclude = B_TRUE;
1064     }
1065 }
1066
1067 /*
1068  * If a filter function exists, call it to determine whether
1069  * this snapshot will be sent.
1070  */
1071 if (exclude || (sdd->filter_cb != NULL &&
1072     sdd->filter_cb(zhp, sdd->filter_cb_arg) == B_FALSE)) {
1073     /*
1074      * This snapshot is filtered out. Don't send it, and don't
1075      * set prevsnap_obj, so it will be as if this snapshot didn't
1076      * exist, and the next accepted snapshot will be sent as
1077      * an incremental from the last accepted one, or as the
1078      * first (and full) snapshot in the case of a replication,
1079      * non-incremental send.
1080      */
1081     zfs_close(zhp);
1082     return (0);
1083 }
1084
1085 gather_holds(zhp, sdd);
1086 err = hold_for_send(zhp, sdd);
1087 if (err) {
1088     if (err == ENOENT)
1089         err = 0;
1090     zfs_close(zhp);
1091     return (err);
1092 }
1093
1094 fromorigin = sdd->prevsnap[0] == '\0' &&
1095     (sdd->fromorigin || sdd->replicate);
1096
1097 if (sdd->verbose) {
1098     uint64_t size;
1099     err = estimate_ioctl(zhp, sdd->prevsnap_obj,
1100         fromorigin, &size);
1101 }
1102
1103 if (sdd->parsable) {
1104     if (sdd->prevsnap[0] != '\0') {
1105         (void) fprintf(stderr, "incremental\t%s\t%s",
1106             sdd->prevsnap, zhp->zfs_name);
1107     } else {
1108         (void) fprintf(stderr, "full\t%s",
1109             zhp->zfs_name);
1110     }
1111 } else {
1112     (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1113         "send from %s to %s"),
1114         sdd->prevsnap, zhp->zfs_name);
1115 }
1116 if (err == 0) {
1117     if (sdd->parsable) {
1118         (void) fprintf(stderr, "\t%llu\n",
1119             (longlong_t)size);
1120     } else {
1121         char buf[16];
1122         zfs_nicenum(size, buf, sizeof(buf));
1123         (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1124             "estimated size is %s\n"), buf);
1125     }
1126     sdd->size += size;

```

```

1118     } else {
1119         (void) fprintf(stderr, "\n");
1120     }
1121 }
1122
1123 if (!sdd->dryrun) {
1124     /*
1125      * If progress reporting is requested, spawn a new thread to
1126      * poll ZFS_IOC_SEND_PROGRESS at a regular interval.
1127      */
1128     if (sdd->progress) {
1129         pa.pa_zhp = zhp;
1130         pa.pa_fd = sdd->outfd;
1131         pa.pa_parsable = sdd->parsable;
1132
1133         if (err = pthread_create(&tid, NULL,
1134             send_progress_thread, &pa)) {
1135             zfs_close(zhp);
1136             return (err);
1137         }
1138     }
1139
1140     err = dump_ioctl(zhp, sdd->prevsnap, sdd->prevsnap_obj,
1141         fromorigin, sdd->outfd, sdd->debugnv);
1142
1143     if (sdd->progress) {
1144         (void) pthread_cancel(tid);
1145         (void) pthread_join(tid, NULL);
1146     }
1147 }
1148
1149 (void) strcpy(sdd->prevsnap, thissnap);
1150 sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
1151 zfs_close(zhp);
1152 return (err);
1153 }
1154
1155 unchanged_portion_omitted
1156
1157 1325 /*
1158 1326  * Generate a send stream for the dataset identified by the argument zhp.
1159 1327  * The content of the send stream is the snapshot identified by
1160 1328  * 'tosnap'. Incremental streams are requested in two ways:
1161 1329  * - from the snapshot identified by "fromsnap" (if non-null) or
1162 1330  * - from the origin of the dataset identified by zhp, which must
1163 1331  * be a clone. In this case, "fromsnap" is null and "fromorigin"
1164 1332  * is TRUE.
1165 1333  *
1166 1334  * The send stream is recursive (i.e. dumps a hierarchy of snapshots) and
1167 1335  * uses a special header (with a hdrtype field of DMU_COMPOUNDSTREAM)
1168 1336  * if "replicate" is set. If "doall" is set, dump all the intermediate
1169 1337  * snapshots. The DMU_COMPOUNDSTREAM header is used in the "doall"
1170 1338  * case too. If "props" is set, send properties.
1171 1339  */
1172 1340 int
1173 1341 zfs_send(zfs_handle_t *zhp, const char *fromsnap, const char *tosnap,
1174 1342     sendflags_t *flags, int outfd, snapfilter_cb_t filter_func,
1175 1343     void *cb_arg, nvlist_t **debugnvp)
1176 1344 {
1177 1345     char errbuf[1024];
1178 1346     send_dump_data_t sdd = { 0 };
1179 1347     int err = 0;
1180 1348     nvlist_t *fss = NULL;
1181 1349     avl_tree_t *fsavl = NULL;
1182 1350     static uint64_t holdseq;
1183 1351     int spa_version;

```

```

1353 pthread_t tid = 0;
1354 pthread_t tid;
1355 int pipefd[2];
1356 dedup_arg_t dda = { 0 };
1357 int featureflags = 0;

1358 (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
1359 "cannot send '%s'", zhp->zfs_name);

1361 if (fromsnap && fromsnap[0] == '\0') {
1362     zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
1363 "zero-length incremental source"));
1364     return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
1365 }

1367 if (zhp->zfs_type == ZFS_TYPE_FILESYSTEM) {
1368     uint64_t version;
1369     version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1370     if (version >= ZPL_VERSION_SA) {
1371         featureflags |= DMU_BACKUP_FEATURE_SA_SPILL;
1372     }
1373 }

1375 if (flags->dedup && !flags->dryrun) {
1376     featureflags |= (DMU_BACKUP_FEATURE_DEDUP |
1377 DMU_BACKUP_FEATURE_DEDUPPROPS);
1378     if (err = pipe(pipefd)) {
1379         zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1380         return (zfs_error(zhp->zfs_hdl, EZFS_PIPEFAILED,
1381 errbuf));
1382     }
1383     dda.outputfd = outfd;
1384     dda.inputfd = pipefd[1];
1385     dda.dedup_hdl = zhp->zfs_hdl;
1386     if (err = pthread_create(&tid, NULL, cksummer, &dda)) {
1387         (void) close(pipefd[0]);
1388         (void) close(pipefd[1]);
1389         zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1390         return (zfs_error(zhp->zfs_hdl,
1391 EZFS_THREADCREATEFAILED, errbuf));
1392     }
1393 }

1395 if (flags->replicate || flags->doall || flags->props) {
1396     dmu_replay_record_t drr = { 0 };
1397     char *packbuf = NULL;
1398     size_t buflen = 0;
1399     zio_cksum_t zc = { 0 };

1401     if (flags->replicate || flags->props) {
1402         nvlist_t *hdrnv;

1404         VERIFY(0 == nvlist_alloc(&hdrnv, NV_UNIQUE_NAME, 0));
1405         if (fromsnap) {
1406             VERIFY(0 == nvlist_add_string(hdrnv,
1407 "fromsnap", fromsnap));
1408         }
1409         VERIFY(0 == nvlist_add_string(hdrnv, "tosnap", tosnap));
1410         if (!flags->replicate) {
1411             VERIFY(0 == nvlist_add_boolean(hdrnv,
1412 "not_recursive"));
1413         }

1415         err = gather_nvlist(zhp->zfs_hdl, zhp->zfs_name,
1416 fromsnap, tosnap, flags->replicate, &fss, &fsavl);
1417         if (err)

```

```

1418         goto err_out;
1419         VERIFY(0 == nvlist_add_nvlist(hdrnv, "fss", fss));
1420         err = nvlist_pack(hdrnv, &packbuf, &buflen,
1421 NV_ENCODE_XDR, 0);
1422         if (debugnvp)
1423             *debugnvp = hdrnv;
1424         else
1425             nvlist_free(hdrnv);
1426         if (err)
1427             if (err) {
1428                 fsavl_destroy(fsavl);
1429                 nvlist_free(fss);
1430                 goto stderr_out;
1431             }
1432     }

1433     if (!flags->dryrun) {
1434         /* write first begin record */
1435         drr.drr_type = DRR_BEGIN;
1436         drr.drr_u.drr_begin.drr_magic = DMU_BACKUP_MAGIC;
1437         DMU_SET_STREAM_HDRTYPE(drr.drr_u.drr_begin.
1438 drr_versioninfo, DMU_COMPOUNDSTREAM);
1439         DMU_SET_FEATUREFLAGS(drr.drr_u.drr_begin.
1440 drr_versioninfo, featureflags);
1441         (void) snprintf(drr.drr_u.drr_begin.drr_toname,
1442 sizeof (drr.drr_u.drr_begin.drr_toname),
1443 "%s@%s", zhp->zfs_name, tosnap);
1444         drr.drr_payloadlen = buflen;
1445         err = cksum_and_write(&drr, sizeof (drr), &zc, outfd);

1447         /* write header nvlist */
1448         if (err != -1 && packbuf != NULL) {
1449             err = cksum_and_write(packbuf, buflen, &zc,
1450 outfd);
1451         }
1452         free(packbuf);
1453         if (err == -1) {
1454             fsavl_destroy(fsavl);
1455             nvlist_free(fss);
1456             err = errno;
1457             goto stderr_out;
1458         }

1459         /* write end record */
1460         bzero(&drr, sizeof (drr));
1461         drr.drr_type = DRR_END;
1462         drr.drr_u.drr_end.drr_checksum = zc;
1463         err = write(outfd, &drr, sizeof (drr));
1464         if (err == -1) {
1465             fsavl_destroy(fsavl);
1466             nvlist_free(fss);
1467             err = errno;
1468             goto stderr_out;
1469         }

1470         err = 0;
1471     }

1472     /* dump each stream */
1473     sdd.fromsnap = fromsnap;
1474     sdd.tosnap = tosnap;
1475     if (tid != 0)
1476         if (flags->dedup)
1477             sdd.outputfd = pipefd[0];
1478     else

```

```

1475         sdd.outfd = outfd;
1476     sdd.replicate = flags->replicate;
1477     sdd.doall = flags->doall;
1478     sdd.fromorigin = flags->fromorigin;
1479     sdd.fss = fss;
1480     sdd.fsavl = fsavl;
1481     sdd.verbose = flags->verbose;
1482     sdd.parsable = flags->parsable;
1483     sdd.progress = flags->progress;
1484     sdd.dryrun = flags->dryrun;
1485     sdd.filter_cb = filter_func;
1486     sdd.filter_cb_arg = cb_arg;
1487     if (debugnvp)
1488         sdd.debugnv = *debugnvp;

1490 /*
1491  * Some flags require that we place user holds on the datasets that are
1492  * being sent so they don't get destroyed during the send. We can skip
1493  * this step if the pool is imported read-only since the datasets cannot
1494  * be destroyed.
1495  */
1496 if (!flags->dryrun && !zpool_get_prop_int(zfs_get_pool_handle(zhp),
1497     ZPOOL_PROP_READONLY, NULL) &&
1498     zfs_spa_version(zhp, &spa_version) == 0 &&
1499     spa_version >= SPA_VERSION_USERREFS &&
1500     (flags->doall || flags->replicate)) {
1501     ++holdseq;
1502     (void) snprintf(sdd.holdtag, sizeof (sdd.holdtag),
1503         ".send-%d-%llu", getpid(), (u_longlong_t)holdseq);
1504     sdd.cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
1505     if (sdd.cleanup_fd < 0) {
1506         err = errno;
1507         goto stderr_out;
1508     }
1509     sdd.snapholds = fnvlist_alloc();
1510 #endif /* ! codereview */
1511     } else {
1512         sdd.cleanup_fd = -1;
1513         sdd.snapholds = NULL;
1514 #endif /* ! codereview */
1515     }
1516     if (flags->verbose || sdd.snapholds != NULL) {
1517         if (flags->verbose) {
1518             /*
1519              * Do a verbose no-op dry run to get all the verbose output
1520              * or to gather snapshot hold's before generating any data,
1521              * then do a non-verbose real run to generate the streams.
1522              * before generating any data. Then do a non-verbose real
1523              * run to generate the streams.
1524              */
1525             sdd.dryrun = B_TRUE;
1526             err = dump_filesystems(zhp, &sdd);

1528             if (err != 0)
1529                 goto stderr_out;

1532             if (flags->verbose) {
1533                 sdd.dryrun = flags->dryrun;
1534                 sdd.verbose = B_FALSE;
1535                 if (flags->parsable) {
1536                     (void) fprintf(stderr, "size\t%llu\n",
1537                         (longlong_t)sdd.size);
1538                 } else {
1539                     char buf[16];
1540                     zfs_nicenum(sdd.size, buf, sizeof (buf));
1541                     (void) fprintf(stderr, dgettext(TEXT_DOMAIN,

```

```

1536         "total estimated size is %s\n"), buf);
1537     }
1538 }

1540 /* Ensure no snaps found is treated as an error. */
1541 if (!sdd.seento) {
1542     err = ENOENT;
1543     goto err_out;
1544 }

1546 /* Skip the second run if dryrun was requested. */
1547 if (flags->dryrun)
1548     goto err_out;

1550 if (sdd.snapholds != NULL) {
1551     err = zfs_hold_nvlist(zhp, sdd.cleanup_fd, sdd.snapholds);
1552     if (err != 0)
1553         goto stderr_out;

1555     fnvlist_free(sdd.snapholds);
1556     sdd.snapholds = NULL;
1557 }

1559 sdd.dryrun = B_FALSE;
1560 sdd.verbose = B_FALSE;
1561 }

1563 #endif /* ! codereview */
1564 err = dump_filesystems(zhp, &sdd);
1565 fsavl_destroy(fsavl);
1566 nvlist_free(fss);

1568 /* Ensure no snaps found is treated as an error. */
1569 if (err == 0 && !sdd.seento)
1570     err = ENOENT;

1572 if (tid != 0) {
1573     if (err != 0)
1574         (void) pthread_cancel(tid);
1575     (void) pthread_join(tid, NULL);
1576     if (flags->dedup) {
1577         (void) close(pipefd[0]);
1578         (void) pthread_join(tid, NULL);
1579     }

1581 if (sdd.cleanup_fd != -1) {
1582     VERIFY(0 == close(sdd.cleanup_fd));
1583     sdd.cleanup_fd = -1;
1584 }

1586 if (!flags->dryrun && (flags->replicate || flags->doall ||
1587     flags->props)) {
1588     /*
1589      * write final end record. NB: want to do this even if
1590      * there was some error, because it might not be totally
1591      * failed.
1592      */
1593     dmu_replay_record_t drr = { 0 };
1594     drr.drr_type = DRR_END;
1595     if (write(outfd, &drr, sizeof (drr)) == -1) {
1596         return (zfs_standard_error(zhp->zfs_hdl,
1597             errno, errbuf));
1598     }
1599 }

1601 return (err || sdd.err);

```

```
1601 stderr_out:
1602     err = zfs_standard_error(zhp->zfs_hdl, err, errbuf);
1603 err_out:
1604     fsavl_destroy(fsavl);
1605     nvlist_free(fss);
1606     fnvlist_free(sdd.snapholds);

1608 #endif /* ! codereview */
1609     if (sdd.cleanup_fd != -1)
1610         VERIFY(0 == close(sdd.cleanup_fd));
1611     if (tid != 0) {
1612         if (flags->dedup) {
1613             (void) pthread_cancel(tid);
1614             (void) pthread_join(tid, NULL);
1615             (void) close(pipefd[0]);
1616         }
1617     }
    return (err);
}
__unchanged_portion_omitted__
```

```

*****
17104 Wed May 29 20:27:08 2013
new/usr/src/lib/libzfs_core/common/libzfs_core.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
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) 2012 by Delphix. All rights reserved.
24  * Copyright (c) 2013 Steven Hartland. All rights reserved.
25 #endif /* !codereview */
26 */

28 /*
29  * LibZFS_Core (lzc) is intended to replace most functionality in libzfs.
30  * It has the following characteristics:
31  *
32  * - Thread Safe. libzfs_core is accessible concurrently from multiple
33  * threads. This is accomplished primarily by avoiding global data
34  * (e.g. caching). Since it's thread-safe, there is no reason for a
35  * process to have multiple libzfs "instances". Therefore, we store
36  * our few pieces of data (e.g. the file descriptor) in global
37  * variables. The fd is reference-counted so that the libzfs_core
38  * library can be "initialized" multiple times (e.g. by different
39  * consumers within the same process).
40  *
41  * - Committed Interface. The libzfs_core interface will be committed,
42  * therefore consumers can compile against it and be confident that
43  * their code will continue to work on future releases of this code.
44  * Currently, the interface is Evolving (not Committed), but we intend
45  * to commit to it once it is more complete and we determine that it
46  * meets the needs of all consumers.
47  *
48  * - Programatic Error Handling. libzfs_core communicates errors with
49  * defined error numbers, and doesn't print anything to stdout/stderr.
50  *
51  * - Thin Layer. libzfs_core is a thin layer, marshaling arguments
52  * to/from the kernel ioctls. There is generally a 1:1 correspondence
53  * between libzfs_core functions and ioctls to /dev/zfs.
54  *
55  * - Clear Atomicity. Because libzfs_core functions are generally 1:1
56  * with kernel ioctls, and kernel ioctls are general atomic, each
57  * libzfs_core function is atomic. For example, creating multiple
58  * snapshots with a single call to lzc_snapshot() is atomic -- it
59  * can't fail with only some of the requested snapshots created, even
60  * in the event of power loss or system crash.

```

```

61 *
62 * - Continued libzfs Support. Some higher-level operations (e.g.
63 * support for "zfs send -R") are too complicated to fit the scope of
64 * libzfs_core. This functionality will continue to live in libzfs.
65 * Where appropriate, libzfs will use the underlying atomic operations
66 * of libzfs_core. For example, libzfs may implement "zfs send -R |
67 * zfs receive" by using individual "send one snapshot", rename,
68 * destroy, and "receive one snapshot" operations in libzfs_core.
69 * /sbin/zfs and /zbin/zpool will link with both libzfs and
70 * libzfs_core. Other consumers should aim to use only libzfs_core,
71 * since that will be the supported, stable interface going forwards.
72 */

74 #include <libzfs_core.h>
75 #include <ctype.h>
76 #include <unistd.h>
77 #include <stdlib.h>
78 #include <string.h>
79 #include <errno.h>
80 #include <fcntl.h>
81 #include <pthread.h>
82 #include <sys/nvpair.h>
83 #include <sys/param.h>
84 #include <sys/types.h>
85 #include <sys/stat.h>
86 #include <sys/zfs_ioctl.h>

88 static int g_fd;
89 static pthread_mutex_t g_lock = PTHREAD_MUTEX_INITIALIZER;
90 static int g_refcount;

92 int
93 libzfs_core_init(void)
94 {
95     (void) pthread_mutex_lock(&g_lock);
96     if (g_refcount == 0) {
97         g_fd = open("/dev/zfs", O_RDWR);
98         if (g_fd < 0) {
99             (void) pthread_mutex_unlock(&g_lock);
100             return (errno);
101         }
102     }
103     g_refcount++;
104     (void) pthread_mutex_unlock(&g_lock);
105     return (0);
106 }

108 void
109 libzfs_core_fini(void)
110 {
111     (void) pthread_mutex_lock(&g_lock);
112     ASSERT3S(g_refcount, >, 0);
113     g_refcount--;
114     if (g_refcount == 0)
115         (void) close(g_fd);
116     (void) pthread_mutex_unlock(&g_lock);
117 }

119 static int
120 lzc_ioctl(zfs_ioc_t ioc, const char *name,
121          nvlist_t *source, nvlist_t **resultp)
122 {
123     zfs_cmd_t zc = { 0 };
124     int error = 0;
125     char *packed;
126     size_t size;

```

```

128     ASSERT3S(g_refcount, >, 0);
130     (void) strncpy(zc.zc_name, name, sizeof (zc.zc_name));

132     packed = fnvlist_pack(source, &size);
133     zc.zc_nvlist_src = (uint64_t)(uintptr_t)packed;
134     zc.zc_nvlist_src_size = size;

136     if (resultp != NULL) {
137         *resultp = NULL;
138         zc.zc_nvlist_dst_size = MAX(size * 2, 128 * 1024);
139         zc.zc_nvlist_dst = (uint64_t)(uintptr_t)
140             malloc(zc.zc_nvlist_dst_size);
141         if (zc.zc_nvlist_dst == NULL) {
142             error = ENOMEM;
143             goto out;
144         }
145     }

147     while (ioctl(g_fd, ioc, &zc) != 0) {
148         if (errno == ENOMEM && resultp != NULL) {
149             free((void *) (uintptr_t)zc.zc_nvlist_dst);
150             zc.zc_nvlist_dst_size *= 2;
151             zc.zc_nvlist_dst = (uint64_t)(uintptr_t)
152                 malloc(zc.zc_nvlist_dst_size);
153             if (zc.zc_nvlist_dst == NULL) {
154                 error = ENOMEM;
155                 goto out;
156             }
157         } else {
158             error = errno;
159             break;
160         }
161     }
162     if (zc.zc_nvlist_dst_filled) {
163         *resultp = fnvlist_unpack((void *) (uintptr_t)zc.zc_nvlist_dst,
164             zc.zc_nvlist_dst_size);
165     }

167 out:
168     fnvlist_pack_free(packed, size);
169     free((void *) (uintptr_t)zc.zc_nvlist_dst);
170     return (error);
171 }

173 int
174 lzc_create(const char *fsname, dmub_objset_type_t type, nvlist_t *props)
175 {
176     int error;
177     nvlist_t *args = fnvlist_alloc();
178     fnvlist_add_int32(args, "type", type);
179     if (props != NULL)
180         fnvlist_add_nvlist(args, "props", props);
181     error = lzc_ioctl(ZFS_IOC_CREATE, fsname, args, NULL);
182     nvlist_free(args);
183     return (error);
184 }

186 int
187 lzc_clone(const char *fsname, const char *origin,
188     nvlist_t *props)
189 {
190     int error;
191     nvlist_t *args = fnvlist_alloc();
192     fnvlist_add_string(args, "origin", origin);

```

```

193     if (props != NULL)
194         fnvlist_add_nvlist(args, "props", props);
195     error = lzc_ioctl(ZFS_IOC_CLONE, fsname, args, NULL);
196     nvlist_free(args);
197     return (error);
198 }

200 /*
201  * Creates snapshots.
202  *
203  * The keys in the snaps nvlist are the snapshots to be created.
204  * They must all be in the same pool.
205  *
206  * The props nvlist is properties to set. Currently only user properties
207  * are supported. { user:prop_name -> string value }
208  *
209  * The returned results nvlist will have an entry for each snapshot that failed.
210  * The value will be the (int32) error code.
211  *
212  * The return value will be 0 if all snapshots were created, otherwise it will
213  * be the errno of a (unspecified) snapshot that failed.
214  */
215 int
216 lzc_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t **errlist)
217 {
218     nvpair_t *elem;
219     nvlist_t *args;
220     int error;
221     char pool[MAXNAMELEN];

223     *errlist = NULL;

225     /* determine the pool name */
226     elem = nvlist_next_nvpair(snaps, NULL);
227     if (elem == NULL)
228         return (0);
229     (void) strncpy(pool, nvpair_name(elem), sizeof (pool));
230     pool[strcspn(pool, "@")] = '\0';

232     args = fnvlist_alloc();
233     fnvlist_add_nvlist(args, "snaps", snaps);
234     if (props != NULL)
235         fnvlist_add_nvlist(args, "props", props);

237     error = lzc_ioctl(ZFS_IOC_SNAPSHOT, pool, args, errlist);
238     nvlist_free(args);

240     return (error);
241 }

243 /*
244  * Destroys snapshots.
245  *
246  * The keys in the snaps nvlist are the snapshots to be destroyed.
247  * They must all be in the same pool.
248  *
249  * Snapshots that do not exist will be silently ignored.
250  *
251  * If 'defer' is not set, and a snapshot has user holds or clones, the
252  * destroy operation will fail and none of the snapshots will be
253  * destroyed.
254  *
255  * If 'defer' is set, and a snapshot has user holds or clones, it will be
256  * marked for deferred destruction, and will be destroyed when the last hold
257  * or clone is removed/destroyed.
258  */

```



```

259 * The return value will be ENOENT if none of the snapshots existed.
260 *
261 #endif /* ! codereview */
262 * The return value will be 0 if all snapshots were destroyed (or marked for
263 * later destruction if 'defer' is set) or didn't exist to begin with and
264 * at least one snapshot was destroyed.
265 * later destruction if 'defer' is set) or didn't exist to begin with.
266 *
267 * Otherwise the return value will be the errno of a (unspecified) snapshot
268 * that failed, no snapshots will be destroyed, and the errlist will have an
269 * entry for each snapshot that failed. The value in the errlist will be
270 * the (int32) error code.
271 */
272 int
273 lzc_destroy_snaps(nvlist_t *snaps, boolean_t defer, nvlist_t **errlist)
274 {
275     nvpair_t *elem;
276     nvlist_t *args;
277     int error;
278     char pool[MAXNAMELEN];
279
280     /* determine the pool name */
281     elem = nvlist_next_nvpair(snaps, NULL);
282     if (elem == NULL)
283         return (0);
284     (void) strncpy(pool, nvpair_name(elem), sizeof (pool));
285     pool[strcspn(pool, "/@")] = '\0';
286
287     args = fnvlist_alloc();
288     fnvlist_add_nvlist(args, "snaps", snaps);
289     if (defer)
290         fnvlist_add_boolean(args, "defer");
291
292     error = lzc_ioctl(ZFS_IOC_DESTROY_SNAPS, pool, args, errlist);
293     nvlist_free(args);
294
295     return (error);
296 }

```

unchanged portion omitted

```

339 /*
340 * Create "user holds" on snapshots. If there is a hold on a snapshot,
341 * the snapshot can not be destroyed. (However, it can be marked for deletion
342 * by lzc_destroy_snaps(defer=B_TRUE).)
343 *
344 * The keys in the nvlist are snapshot names.
345 * The snapshots must all be in the same pool.
346 * The value is the name of the hold (string type).
347 *
348 * If cleanup_fd is not -1, it must be the result of open("/dev/zfs", O_EXCL).
349 * In this case, when the cleanup_fd is closed (including on process
350 * termination), the holds will be released. If the system is shut down
351 * uncleanly, the holds will be released when the pool is next opened
352 * or imported.
353 *
354 * Holds for snapshots which don't exist will be skipped and have an entry
355 * added to errlist, but will not cause an overall failure, except in the
356 * case that all holds were skipped.
357 *
358 * The return value will be ENOENT if none of the snapshots for the requested
359 * holds existed.
360 *
361 * The return value will be 0 if the nvl holds was empty or all holds, for
362 * snapshots that existed, were successfully created and at least one hold
363 * was created.

```

```

364 *
365 * Otherwise the return value will be the errno of a (unspecified) hold that
366 * failed and no holds will be created.
367 *
368 * In all cases the errlist will have an entry for each hold that failed
369 * (name = snapshot), with its value being the error code (int32).
370 * The return value will be 0 if all holds were created. Otherwise the return
371 * value will be the errno of a (unspecified) hold that failed, no holds will
372 * be created, and the errlist will have an entry for each hold that
373 * failed (name = snapshot). The value in the errlist will be the error
374 * code (int32).
375 */
376 int
377 lzc_hold(nvlist_t *holds, int cleanup_fd, nvlist_t **errlist)
378 {
379     char pool[MAXNAMELEN];
380     nvlist_t *args;
381     nvpair_t *elem;
382     int error;
383
384     /* determine the pool name */
385     elem = nvlist_next_nvpair(holds, NULL);
386     if (elem == NULL)
387         return (0);
388     (void) strncpy(pool, nvpair_name(elem), sizeof (pool));
389     pool[strcspn(pool, "/@")] = '\0';
390
391     args = fnvlist_alloc();
392     fnvlist_add_nvlist(args, "holds", holds);
393     if (cleanup_fd != -1)
394         fnvlist_add_int32(args, "cleanup_fd", cleanup_fd);
395
396     error = lzc_ioctl(ZFS_IOC_HOLD, pool, args, errlist);
397     nvlist_free(args);
398     return (error);
399 }

```

unchanged portion omitted

```

396 /*
397 * Release "user holds" on snapshots. If the snapshot has been marked for
398 * deferred destroy (by lzc_destroy_snaps(defer=B_TRUE)), it does not have
399 * any clones, and all the user holds are removed, then the snapshot will be
400 * destroyed.
401 *
402 * The keys in the nvlist are snapshot names.
403 * The snapshots must all be in the same pool.
404 * The value is a nvlist whose keys are the holds to remove.
405 *
406 * Holds which failed to release because they didn't exist will have an entry
407 * added to errlist, but will not cause an overall failure, except in the
408 * case that all releases were skipped.
409 *
410 * The return value will be ENOENT if none of the specified holds existed.
411 *
412 * The return value will be 0 if the nvl holds was empty or all holds, that
413 * existed, were successfully removed and at least one hold was removed.
414 *
415 * Otherwise the return value will be the errno of a (unspecified) hold that
416 * failed to release and no holds will be released.
417 *
418 * In all cases the errlist will have an entry for each hold that failed to
419 * release.
420 * The return value will be 0 if all holds were removed.
421 * Otherwise the return value will be the errno of a (unspecified) release
422 * that failed, no holds will be released, and the errlist will have an
423 * entry for each snapshot that has failed releases (name = snapshot).
424 * The value in the errlist will be the error code (int32) of a failed release.

```

```
420 */
421 int
422 lzc_release(nvlist_t *holds, nvlist_t **errlist)
423 {
424     char pool[MAXNAMELEN];
425     nvpair_t *elem;
426
427     /* determine the pool name */
428     elem = nvlist_next_nvpair(holds, NULL);
429     if (elem == NULL)
430         return (0);
431     (void) strncpy(pool, nvpair_name(elem), sizeof (pool));
432     pool[strcspn(pool, "@/")] = '\0';
433
434     return (lzc_ioctl(ZFS_IOC_RELEASE, pool, holds, errlist));
435 }
436
437 _____unchanged_portion_omitted_____
```

```

*****
25817 Wed May 29 20:27:08 2013
new/usr/src/uts/common/fs/zfs/dsl_destroy.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2013 by Delphix. All rights reserved.
24 * Copyright (c) 2013 Steven Hartland. All rights reserved.
25 #endif /* !codereview */
26 */

28 #include <sys/zfs_context.h>
29 #include <sys/dsl_userhold.h>
30 #include <sys/dsl_dataset.h>
31 #include <sys/dsl_synctask.h>
32 #include <sys/dmu_tx.h>
33 #include <sys/dsl_pool.h>
34 #include <sys/dsl_dir.h>
35 #include <sys/dmu_traverse.h>
36 #include <sys/dsl_scan.h>
37 #include <sys/dmu_objset.h>
38 #include <sys/zap.h>
39 #include <sys/zfeature.h>
40 #include <sys/zfs_ioctl.h>
41 #include <sys/dsl_deleg.h>

43 typedef struct dmu_snapshots_destroy_arg {
44     nvlist_t *dsda_snaps;
45     nvlist_t *dsda_successful_snaps;
46     boolean_t dsda_defer;
47     nvlist_t *dsda_errlist;
48 } dmu_snapshots_destroy_arg_t;

50 /*
51  * ds must be owned.
52  */
53 static int
54 dsl_destroy_snapshot_check_impl(dsl_dataset_t *ds, boolean_t defer)
55 {
56     if (!dsl_dataset_is_snapshot(ds))
57         return (SET_ERROR(EINVAL));

59     if (dsl_dataset_long_held(ds))
60         return (SET_ERROR(EBUSY));

```

```

62     /*
63      * Only allow deferred destroy on pools that support it.
64      * NOTE: deferred destroy is only supported on snapshots.
65      */
66     if (defer) {
67         if (spa_version(ds->ds_dir->dd_pool->dp_spa) <
68             SPA_VERSION_USERREFS)
69             return (SET_ERROR(ENOTSUP));
70         return (0);
71     }

73     /*
74      * If this snapshot has an elevated user reference count,
75      * we can't destroy it yet.
76      */
77     if (ds->ds_userrefs > 0)
78         return (SET_ERROR(EBUSY));

80     /*
81      * Can't delete a branch point.
82      */
83     if (ds->ds_phys->ds_num_children > 1)
84         return (SET_ERROR(EEXIST));

86     return (0);
87 }

89 static int
90 dsl_destroy_snapshot_check(void *arg, dmu_tx_t *tx)
91 {
92     dmu_snapshots_destroy_arg_t *dsda = arg;
93     dsl_pool_t *dp = dmu_tx_pool(tx);
94     nvpair_t *pair;
95     int error = 0;

97     if (!dmu_tx_is_syncing(tx))
98         return (0);

100     for (pair = nvlist_next_nvpair(dsda->dsda_snaps, NULL);
101          pair != NULL; pair = nvlist_next_nvpair(dsda->dsda_snaps, pair)) {
102         dsl_dataset_t *ds;

104         error = dsl_dataset_hold(dp, nvpair_name(pair),
105             FTAG, &ds);

107         /*
108          * If the snapshot does not exist, silently ignore it
109          * (it's "already destroyed").
110          */
111         if (error == ENOENT)
112             continue;

114         if (error == 0) {
115             error = dsl_destroy_snapshot_check_impl(ds,
116                 dsda->dsda_defer);
117             dsl_dataset_rele(ds, FTAG);
118         }

120         if (error == 0) {
121             fnvlist_add_boolean(dsda->dsda_successful_snaps,
122                 nvpair_name(pair));
123         } else {
124             fnvlist_add_int32(dsda->dsda_errlist,
125                 nvpair_name(pair), error);
126         }

```

```

127     }
129     pair = nvlist_next_nvpair(dsda->dsda_errlist, NULL);
130     if (pair != NULL)
131         return (fnvpair_value_int32(pair));
133     if (nvlist_next_nvpair(dsda->dsda_successful_snaps, NULL) == NULL)
134         return (ENOENT);
136 #endif /* ! codereview */
137     return (0);
138 }
140 struct process_old_arg {
141     dsl_dataset_t *ds;
142     dsl_dataset_t *ds_prev;
143     boolean_t after_branch_point;
144     zio_t *pio;
145     uint64_t used, comp, uncomp;
146 };
148 static int
149 process_old_cb(void *arg, const blkptr_t *bp, dmu_tx_t *tx)
150 {
151     struct process_old_arg *poa = arg;
152     dsl_pool_t *dp = poa->ds->ds_dir->dd_pool;
154     if (bp->blk_birth <= poa->ds->ds_phys->ds_prev_snap_txg) {
155         dsl_deadlist_insert(&poa->ds->ds_deadlist, bp, tx);
156         if (poa->ds_prev && !poa->after_branch_point &&
157             bp->blk_birth >
158             poa->ds_prev->ds_phys->ds_prev_snap_txg) {
159             poa->ds_prev->ds_phys->ds_unique_bytes +=
160                 bp_get_dsize_sync(dp->dp_spa, bp);
161         }
162     } else {
163         poa->used += bp_get_dsize_sync(dp->dp_spa, bp);
164         poa->comp += BP_GET_PSIZE(bp);
165         poa->uncomp += BP_GET_UCSIZE(bp);
166         dsl_free_sync(poa->pio, dp, tx->tx_txg, bp);
167     }
168     return (0);
169 }
171 static void
172 process_old_deadlist(dsl_dataset_t *ds, dsl_dataset_t *ds_prev,
173     dsl_dataset_t *ds_next, boolean_t after_branch_point, dmu_tx_t *tx)
174 {
175     struct process_old_arg poa = { 0 };
176     dsl_pool_t *dp = ds->ds_dir->dd_pool;
177     objset_t *mos = dp->dp_meta_objset;
178     uint64_t deadlist_obj;
180     ASSERT(ds->ds_deadlist.dl_oldfmt);
181     ASSERT(ds_next->ds_deadlist.dl_oldfmt);
183     poa.ds = ds;
184     poa.ds_prev = ds_prev;
185     poa.after_branch_point = after_branch_point;
186     poa.pio = zio_root(dp->dp_spa, NULL, NULL, ZIO_FLAG_MUSTSUCCEED);
187     VERIFY0(bpobj_iterate(&ds_next->ds_deadlist.dl_bpobj,
188         process_old_cb, &poa, tx));
189     VERIFY0(zio_wait(poa.pio));
190     ASSERT3U(poa.used, ==, ds->ds_phys->ds_unique_bytes);
192     /* change snapused */

```

```

193     dsl_dir_diduse_space(ds->ds_dir, DD_USED_SNAP,
194         -poa.used, -poa.comp, -poa.uncomp, tx);
196     /* swap next's deadlist to our deadlist */
197     dsl_deadlist_close(&ds->ds_deadlist);
198     dsl_deadlist_close(&ds_next->ds_deadlist);
199     deadlist_obj = ds->ds_phys->ds_deadlist_obj;
200     ds->ds_phys->ds_deadlist_obj = ds_next->ds_phys->ds_deadlist_obj;
201     ds_next->ds_phys->ds_deadlist_obj = deadlist_obj;
202     dsl_deadlist_open(&ds->ds_deadlist, mos, ds->ds_phys->ds_deadlist_obj);
203     dsl_deadlist_open(&ds_next->ds_deadlist, mos,
204         ds_next->ds_phys->ds_deadlist_obj);
205 }
207 static void
208 dsl_dataset_remove_clones_key(dsl_dataset_t *ds, uint64_t mintxg, dmu_tx_t *tx)
209 {
210     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
211     zap_cursor_t zc;
212     zap_attribute_t za;
214     /*
215      * If it is the old version, dd_clones doesn't exist so we can't
216      * find the clones, but dsl_deadlist_remove_key() is a no-op so it
217      * doesn't matter.
218      */
219     if (ds->ds_dir->dd_phys->dd_clones == 0)
220         return;
222     for (zap_cursor_init(&zc, mos, ds->ds_dir->dd_phys->dd_clones);
223         zap_cursor_retrieve(&zc, &za) == 0;
224         zap_cursor_advance(&zc)) {
225         dsl_dataset_t *clone;
227         VERIFY0(dsl_dataset_hold_obj(ds->ds_dir->dd_pool,
228             za.za_first_integer, FTAG, &clone));
229         if (clone->ds_dir->dd_origin_txg > mintxg) {
230             dsl_deadlist_remove_key(&clone->ds_deadlist,
231                 mintxg, tx);
232             dsl_dataset_remove_clones_key(clone, mintxg, tx);
233         }
234         dsl_dataset_rele(clone, FTAG);
235     }
236     zap_cursor_fini(&zc);
237 }
239 void
240 dsl_destroy_snapshot_sync_impl(dsl_dataset_t *ds, boolean_t defer, dmu_tx_t *tx)
241 {
242     int err;
243     int after_branch_point = FALSE;
244     dsl_pool_t *dp = ds->ds_dir->dd_pool;
245     objset_t *mos = dp->dp_meta_objset;
246     dsl_dataset_t *ds_prev = NULL;
247     uint64_t obj;
249     ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));
250     ASSERT3U(ds->ds_phys->ds_bp.blk_birth, <=, tx->tx_txg);
251     ASSERT(refcount_is_zero(&ds->ds_longholds));
253     if (defer &&
254         (ds->ds_userrefs > 0 || ds->ds_phys->ds_num_children > 1)) {
255         ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);
256         dmu_buf_will_dirty(ds->ds_dbuf, tx);
257         ds->ds_phys->ds_flags |= DS_FLAG_DEFER_DESTROY;
258         spa_history_log_internal_ds(ds, "defer_destroy", tx, "");

```

```

259         return;
260     }

262     ASSERT3U(ds->ds_phys->ds_num_children, <=, 1);

264     /* We need to log before removing it from the namespace. */
265     spa_history_log_internal_ds(ds, "destroy", tx, "");

267     dsl_scan_ds_destroyed(ds, tx);

269     obj = ds->ds_object;

271     if (ds->ds_phys->ds_prev_snap_obj != 0) {
272         ASSERT3P(ds->ds_prev, ==, NULL);
273         VERIFY0(dsl_dataset_hold_obj(dp,
274             ds->ds_phys->ds_prev_snap_obj, FTAG, &ds_prev));
275         after_branch_point =
276             (ds_prev->ds_phys->ds_next_snap_obj != obj);

278         dmu_buf_will_dirty(ds_prev->ds_dbuf, tx);
279         if (after_branch_point &&
280             ds_prev->ds_phys->ds_next_clones_obj != 0) {
281             dsl_dataset_remove_from_next_clones(ds_prev, obj, tx);
282             if (ds->ds_phys->ds_next_snap_obj != 0) {
283                 VERIFY0(zap_add_int(mos,
284                     ds_prev->ds_phys->ds_next_clones_obj,
285                     ds->ds_phys->ds_next_snap_obj, tx));
286             }
287         }
288         if (!after_branch_point) {
289             ds_prev->ds_phys->ds_next_snap_obj =
290                 ds->ds_phys->ds_next_snap_obj;
291         }
292     }

294     dsl_dataset_t *ds_next;
295     uint64_t old_unique;
296     uint64_t used = 0, comp = 0, uncomp = 0;

298     VERIFY0(dsl_dataset_hold_obj(dp,
299         ds->ds_phys->ds_next_snap_obj, FTAG, &ds_next));
300     ASSERT3U(ds_next->ds_phys->ds_prev_snap_obj, ==, obj);

302     old_unique = ds_next->ds_phys->ds_unique_bytes;

304     dmu_buf_will_dirty(ds_next->ds_dbuf, tx);
305     ds_next->ds_phys->ds_prev_snap_obj =
306         ds->ds_phys->ds_prev_snap_obj;
307     ds_next->ds_phys->ds_prev_snap_txg =
308         ds->ds_phys->ds_prev_snap_txg;
309     ASSERT3U(ds->ds_phys->ds_prev_snap_txg, ==,
310         ds_prev ? ds_prev->ds_phys->ds_creation_txg : 0);

312     if (ds_next->ds_deadlist.dl_oldfmt) {
313         process_old_deadlist(ds, ds_prev, ds_next,
314             after_branch_point, tx);
315     } else {
316         /* Adjust prev's unique space. */
317         if (ds_prev && !after_branch_point) {
318             dsl_deadlist_space_range(&ds_next->ds_deadlist,
319                 ds_prev->ds_phys->ds_prev_snap_txg,
320                 ds->ds_phys->ds_prev_snap_txg,
321                 &used, &comp, &uncomp);
322             ds_prev->ds_phys->ds_unique_bytes += used;
323         }

```

```

325         /* Adjust snapused. */
326         dsl_deadlist_space_range(&ds_next->ds_deadlist,
327             ds->ds_phys->ds_prev_snap_txg, UINT64_MAX,
328             &used, &comp, &uncomp);
329         dsl_dir_diduse_space(ds->ds_dir, DD_USED_SNAP,
330             -used, -comp, -uncomp, tx);

332         /* Move blocks to be freed to pool's free list. */
333         dsl_deadlist_move_bpobj(&ds_next->ds_deadlist,
334             &dp->dp_free_bpobj, ds->ds_phys->ds_prev_snap_txg,
335             tx);
336         dsl_dir_diduse_space(tx->tx_pool->dp_free_dir,
337             DD_USED_HEAD, used, comp, uncomp, tx);

339         /* Merge our deadlist into next's and free it. */
340         dsl_deadlist_merge(&ds_next->ds_deadlist,
341             ds->ds_phys->ds_deadlist_obj, tx);
342     }
343     dsl_deadlist_close(&ds->ds_deadlist);
344     dsl_deadlist_free(mos, ds->ds_phys->ds_deadlist_obj, tx);
345     dmu_buf_will_dirty(ds->ds_dbuf, tx);
346     ds->ds_phys->ds_deadlist_obj = 0;

348     /* Collapse range in clone heads */
349     dsl_dataset_remove_clones_key(ds,
350         ds->ds_phys->ds_creation_txg, tx);

352     if (dsl_dataset_is_snapshot(ds_next)) {
353         dsl_dataset_t *ds_nextnext;

355         /*
356          * Update next's unique to include blocks which
357          * were previously shared by only this snapshot
358          * and it. Those blocks will be born after the
359          * prev snap and before this snap, and will have
360          * died after the next snap and before the one
361          * after that (ie. be on the snap after next's
362          * deadlist).
363          */
364         VERIFY0(dsl_dataset_hold_obj(dp,
365             ds_next->ds_phys->ds_next_snap_obj, FTAG, &ds_nextnext));
366         dsl_deadlist_space_range(&ds_nextnext->ds_deadlist,
367             ds->ds_phys->ds_prev_snap_txg,
368             ds->ds_phys->ds_creation_txg,
369             &used, &comp, &uncomp);
370         ds_next->ds_phys->ds_unique_bytes += used;
371         dsl_dataset_rele(ds_nextnext, FTAG);
372         ASSERT3P(ds_next->ds_prev, ==, NULL);

374         /* Collapse range in this head. */
375         dsl_dataset_t *hds;
376         VERIFY0(dsl_dataset_hold_obj(dp,
377             ds->ds_dir->dd_phys->dd_head_dataset_obj, FTAG, &hds));
378         dsl_deadlist_remove_key(&hds->ds_deadlist,
379             ds->ds_phys->ds_creation_txg, tx);
380         dsl_dataset_rele(hds, FTAG);

382     } else {
383         ASSERT3P(ds_next->ds_prev, ==, ds);
384         dsl_dataset_rele(ds_next->ds_prev, ds_next);
385         ds_next->ds_prev = NULL;
386         if (ds_prev) {
387             VERIFY0(dsl_dataset_hold_obj(dp,
388                 ds->ds_phys->ds_prev_snap_obj,
389                 ds_next, &ds_next->ds_prev));
390         }

```

```

392     dsl_dataset_recalc_head_uniq(ds_next);
393
394     /*
395      * Reduce the amount of our unconsumed reservation
396      * being charged to our parent by the amount of
397      * new unique data we have gained.
398      */
399     if (old_unique < ds_next->ds_reserved) {
400         int64_t mrsdelta;
401         uint64_t new_unique =
402             ds_next->ds_phys->ds_unique_bytes;
403
404         ASSERT(old_unique <= new_unique);
405         mrsdelta = MIN(new_unique - old_unique,
406             ds_next->ds_reserved - old_unique);
407         dsl_dir_diduse_space(ds->ds_dir,
408             DD_USED_REFRSRV, -mrsdelta, 0, 0, tx);
409     }
410     dsl_dataset_rele(ds_next, FTAG);
411
412     /*
413      * This must be done after the dsl_traverse(), because it will
414      * re-open the objset.
415      */
416     if (ds->ds_objset) {
417         dmu_objset_evict(ds->ds_objset);
418         ds->ds_objset = NULL;
419     }
420
421     /* remove from snapshot namespace */
422     dsl_dataset_t *ds_head;
423     ASSERT(ds->ds_phys->ds_snapnames_zapobj == 0);
424     VERIFY0(dsl_dataset_hold_obj(dp,
425         ds->ds_dir->dd_phys->dd_head_dataset_obj, FTAG, &ds_head));
426     VERIFY0(dsl_dataset_get_snapname(ds));
427 #ifdef ZFS_DEBUG
428     {
429         uint64_t val;
430
431         err = dsl_dataset_snap_lookup(ds_head,
432             ds->ds_snapname, &val);
433         ASSERT0(err);
434         ASSERT3U(val, ==, obj);
435     }
436 #endif
437     VERIFY0(dsl_dataset_snap_remove(ds_head, ds->ds_snapname, tx));
438     dsl_dataset_rele(ds_head, FTAG);
439
440     if (ds_prev != NULL)
441         dsl_dataset_rele(ds_prev, FTAG);
442
443     spa_prop_clear_bootfs(dp->dp_spa, ds->ds_object, tx);
444
445     if (ds->ds_phys->ds_next_clones_obj != 0) {
446         uint64_t count;
447         ASSERT0(zap_count(mos,
448             ds->ds_phys->ds_next_clones_obj, &count) && count == 0);
449         VERIFY0(dmu_object_free(mos,
450             ds->ds_phys->ds_next_clones_obj, tx));
451     }
452     if (ds->ds_phys->ds_props_obj != 0)
453         VERIFY0(zap_destroy(mos, ds->ds_phys->ds_props_obj, tx));
454     if (ds->ds_phys->ds_userrefs_obj != 0)
455         VERIFY0(zap_destroy(mos, ds->ds_phys->ds_userrefs_obj, tx));
456

```

```

457     dsl_dir_rele(ds->ds_dir, ds);
458     ds->ds_dir = NULL;
459     VERIFY0(dmu_object_free(mos, obj, tx));
460 }
461
462 static void
463 dsl_destroy_snapshot_sync(void *arg, dmu_tx_t *tx)
464 {
465     dmu_snapshots_destroy_arg_t *dsda = arg;
466     dsl_pool_t *dp = dmu_tx_pool(tx);
467     nvpair_t *pair;
468
469     for (pair = nvlist_next_nvpair(dsda->dsda_successful_snaps, NULL);
470          pair != NULL;
471          pair = nvlist_next_nvpair(dsda->dsda_successful_snaps, pair)) {
472         dsl_dataset_t *ds;
473
474         VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
475
476         dsl_destroy_snapshot_sync_impl(ds, dsda->dsda_defer, tx);
477         dsl_dataset_rele(ds, FTAG);
478     }
479 }
480
481 /*
482 * The semantics of this function are described in the comment above
483 * lz_c_destroy_snaps(). To summarize:
484 *
485 * The snapshots must all be in the same pool.
486 *
487 * Snapshots that don't exist will be silently ignored (considered to be
488 * "already deleted").
489 *
490 * On success, all snaps will be destroyed and this will return 0.
491 * On failure, no snaps will be destroyed, the errlist will be filled in,
492 * and this will return an errno.
493 */
494 int
495 dsl_destroy_snapshots_nvlist(nvlist_t *snaps, boolean_t defer,
496     nvlist_t *errlist)
497 {
498     dmu_snapshots_destroy_arg_t dsda;
499     int error;
500     nvpair_t *pair;
501
502     pair = nvlist_next_nvpair(snaps, NULL);
503     if (pair == NULL)
504         return (0);
505
506     dsda.dsda_snaps = snaps;
507     dsda.dsda_successful_snaps = fnvlist_alloc();
508     dsda.dsda_defer = defer;
509     dsda.dsda_errlist = errlist;
510
511     error = dsl_sync_task(nvpair_name(pair),
512         dsl_destroy_snapshot_check, dsl_destroy_snapshot_sync,
513         &dsda, 0);
514     fnvlist_free(dsda.dsda_successful_snaps);
515
516     return (error);
517 }
518
519 int
520 dsl_destroy_snapshot(const char *name, boolean_t defer)
521 {
522     int error;

```

```

523     nvlist_t *nvl = fnvlist_alloc();
524     nvlist_t *errlist = fnvlist_alloc();

526     fnvlist_add_boolean(nvl, name);
527     error = dsl_destroy_snapshots_nvl(nvl, defer, errlist);
528     fnvlist_free(errlist);
529     fnvlist_free(nvl);
530     return (error);
531 }

533 struct killarg {
534     dsl_dataset_t *ds;
535     dmu_tx_t *tx;
536 };

538 /* ARGSUSED */
539 static int
540 kill_blkptr(spa_t *spa, zillog_t *zillog, const blkptr_t *bp,
541     const zbookmark_t *zb, const dnode_phys_t *dnp, void *arg)
542 {
543     struct killarg *ka = arg;
544     dmu_tx_t *tx = ka->tx;

546     if (bp == NULL)
547         return (0);

549     if (zb->zb_level == ZB_ZIL_LEVEL) {
550         ASSERT(zillog != NULL);
551         /*
552          * It's a block in the intent log. It has no
553          * accounting, so just free it.
554          */
555         dsl_free(ka->tx->tx_pool, ka->tx->tx_tkg, bp);
556     } else {
557         ASSERT(zillog == NULL);
558         ASSERT3U(bp->blk_birth, >, ka->ds->ds_phys->ds_prev_snap_tkg);
559         (void) dsl_dataset_block_kill(ka->ds, bp, tx, B_FALSE);
560     }

562     return (0);
563 }

565 static void
566 old_synchronous_dataset_destroy(dsl_dataset_t *ds, dmu_tx_t *tx)
567 {
568     struct killarg ka;

570     /*
571      * Free everything that we point to (that's born after
572      * the previous snapshot, if we are a clone)
573      *
574      * NB: this should be very quick, because we already
575      * freed all the objects in open context.
576      */
577     ka.ds = ds;
578     ka.tx = tx;
579     VERIFY0(traverse_dataset(ds,
580         ds->ds_phys->ds_prev_snap_tkg, TRAVERSE_POST,
581         kill_blkptr, &ka));
582     ASSERT(!DS_UNIQUE_IS_ACCURATE(ds) || ds->ds_phys->ds_unique_bytes == 0);
583 }

585 typedef struct dsl_destroy_head_arg {
586     const char *ddha_name;
587 } dsl_destroy_head_arg_t;

```

```

589 int
590 dsl_destroy_head_check_impl(dsl_dataset_t *ds, int expected_holds)
591 {
592     int error;
593     uint64_t count;
594     objset_t *mos;

596     if (dsl_dataset_is_snapshot(ds))
597         return (SET_ERROR(EINVAL));

599     if (refcount_count(&ds->ds_longholds) != expected_holds)
600         return (SET_ERROR(EBUSY));

602     mos = ds->ds_dir->dd_pool->dp_meta_objset;

604     /*
605      * Can't delete a head dataset if there are snapshots of it.
606      * (Except if the only snapshots are from the branch we cloned
607      * from.)
608      */
609     if (ds->ds_prev != NULL &&
610         ds->ds_prev->ds_phys->ds_next_snap_obj == ds->ds_object)
611         return (SET_ERROR(EBUSY));

613     /*
614      * Can't delete if there are children of this fs.
615      */
616     error = zap_count(mos,
617         ds->ds_dir->dd_phys->dd_child_dir_zapobj, &count);
618     if (error != 0)
619         return (error);
620     if (count != 0)
621         return (SET_ERROR(EEXIST));

623     if (dsl_dir_is_clone(ds->ds_dir) && DS_IS_DEFER_DESTROY(ds->ds_prev) &&
624         ds->ds_prev->ds_phys->ds_num_children == 2 &&
625         ds->ds_prev->ds_userrefs == 0) {
626         /* We need to remove the origin snapshot as well. */
627         if (!refcount_is_zero(&ds->ds_prev->ds_longholds))
628             return (SET_ERROR(EBUSY));
629     }
630     return (0);
631 }

633 static int
634 dsl_destroy_head_check(void *arg, dmu_tx_t *tx)
635 {
636     dsl_destroy_head_arg_t *ddha = arg;
637     dsl_pool_t *dp = dmu_tx_pool(tx);
638     dsl_dataset_t *ds;
639     int error;

641     error = dsl_dataset_hold(dp, ddha->ddha_name, FTAG, &ds);
642     if (error != 0)
643         return (error);

645     error = dsl_destroy_head_check_impl(ds, 0);
646     dsl_dataset_rele(ds, FTAG);
647     return (error);
648 }

650 static void
651 dsl_dir_destroy_sync(uint64_t ddoobj, dmu_tx_t *tx)
652 {
653     dsl_dir_t *dd;
654     dsl_pool_t *dp = dmu_tx_pool(tx);

```

```

655     objset_t *mos = dp->dp_meta_objset;
656     dd_used_t t;

658     ASSERT(RRW_WRITE_HELD(&dmu_tx_pool(tx)->dp_config_rwlock));

660     VERIFY0(dsl_dir_hold_obj(dp, ddoobj, NULL, FTAG, &dd));

662     ASSERT0(dd->dd_phys->dd_head_dataset_obj);

664     /*
665      * Remove our reservation. The impl() routine avoids setting the
666      * actual property, which would require the (already destroyed) ds.
667      */
668     dsl_dir_set_reservation_sync_impl(dd, 0, tx);

670     ASSERT0(dd->dd_phys->dd_used_bytes);
671     ASSERT0(dd->dd_phys->dd_reserved);
672     for (t = 0; t < DD_USED_NUM; t++)
673         ASSERT0(dd->dd_phys->dd_used_breakdown[t]);

675     VERIFY0(zap_destroy(mos, dd->dd_phys->dd_child_dir_zapobj, tx));
676     VERIFY0(zap_destroy(mos, dd->dd_phys->dd_props_zapobj, tx));
677     VERIFY0(dsl_deleg_destroy(mos, dd->dd_phys->dd_deleg_zapobj, tx));
678     VERIFY0(zap_remove(mos,
679         dd->dd_parent->dd_phys->dd_child_dir_zapobj, dd->dd_myname, tx));

681     dsl_dir_rele(dd, FTAG);
682     VERIFY0(dmu_object_free(mos, ddoobj, tx));
683 }

685 void
686 dsl_destroy_head_sync_impl(dsl_dataset_t *ds, dmu_tx_t *tx)
687 {
688     dsl_pool_t *dp = dmu_tx_pool(tx);
689     objset_t *mos = dp->dp_meta_objset;
690     uint64_t obj, ddoobj, prevobj = 0;
691     boolean_t rmorigin;

693     ASSERT3U(ds->ds_phys->ds_num_children, <=, 1);
694     ASSERT(ds->ds_prev == NULL ||
695         ds->ds_prev->ds_phys->ds_next_snap_obj != ds->ds_object);
696     ASSERT3U(ds->ds_phys->ds_bp.blk_birth, <=, tx->tx_txg);
697     ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

699     /* We need to log before removing it from the namespace. */
700     spa_history_log_internal(ds, "destroy", tx, "");

702     rmorigin = (dsl_dir_is_clone(ds->ds_dir) &&
703         DS_IS_DEFER_DESTROY(ds->ds_prev) &&
704         ds->ds_prev->ds_phys->ds_num_children == 2 &&
705         ds->ds_prev->ds_userrefs == 0);

707     /* Remove our reservation */
708     if (ds->ds_reserved != 0) {
709         dsl_dataset_set_refreservation_sync_impl(ds,
710             (ZPROP_SRC_NONE | ZPROP_SRC_LOCAL | ZPROP_SRC_RECEIVED),
711             0, tx);
712         ASSERT0(ds->ds_reserved);
713     }

715     dsl_scan_ds_destroyed(ds, tx);

717     obj = ds->ds_object;

719     if (ds->ds_phys->ds_prev_snap_obj != 0) {
720         /* This is a clone */

```

```

721         ASSERT(ds->ds_prev != NULL);
722         ASSERT3U(ds->ds_prev->ds_phys->ds_next_snap_obj, !=, obj);
723         ASSERT0(ds->ds_phys->ds_next_snap_obj);

725         dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
726         if (ds->ds_prev->ds_phys->ds_next_clones_obj != 0) {
727             dsl_dataset_remove_from_next_clones(ds->ds_prev,
728                 obj, tx);
729         }

731         ASSERT3U(ds->ds_prev->ds_phys->ds_num_children, >, 1);
732         ds->ds_prev->ds_phys->ds_num_children--;
733     }

735     zfeature_info_t *async_destroy =
736         &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY];
737     objset_t *os;

739     /*
740      * Destroy the deadlist. Unless it's a clone, the
741      * deadlist should be empty. (If it's a clone, it's
742      * safe to ignore the deadlist contents.)
743      */
744     dsl_deadlist_close(&ds->ds_deadlist);
745     dsl_deadlist_free(mos, ds->ds_phys->ds_deadlist_obj, tx);
746     dmu_buf_will_dirty(ds->ds_dbuf, tx);
747     ds->ds_phys->ds_deadlist_obj = 0;

749     VERIFY0(dmu_objset_from_ds(ds, &os));

751     if (!spa_feature_is_enabled(dp->dp_spa, async_destroy)) {
752         old_synchronous_dataset_destroy(ds, tx);
753     } else {
754         /*
755          * Move the bptree into the pool's list of trees to
756          * clean up and update space accounting information.
757          */
758         uint64_t used, comp, uncomp;

760         zil_destroy_sync(dmu_objset_zil(os), tx);

762         if (!spa_feature_is_active(dp->dp_spa, async_destroy)) {
763             dsl_scan_t *scn = dp->dp_scan;

765             spa_feature_incr(dp->dp_spa, async_destroy, tx);
766             dp->dp_bptree_obj = bptree_alloc(mos, tx);
767             VERIFY0(zap_add(mos,
768                 DMU_POOL_DIRECTORY_OBJECT,
769                 DMU_POOL_BPTREE_OBJ, sizeof(uint64_t), 1,
770                 &dp->dp_bptree_obj, tx));
771             ASSERT(!scn->scn_async_destroying);
772             scn->scn_async_destroying = B_TRUE;
773         }

775         used = ds->ds_dir->dd_phys->dd_used_bytes;
776         comp = ds->ds_dir->dd_phys->dd_compressed_bytes;
777         uncomp = ds->ds_dir->dd_phys->dd_uncompressed_bytes;

779         ASSERT(1DS_UNIQUE_IS_ACCURATE(ds) ||
780             ds->ds_phys->ds_unique_bytes == used);

782         bptree_add(mos, dp->dp_bptree_obj,
783             &ds->ds_phys->ds_bp, ds->ds_phys->ds_prev_snap_txg,
784             used, comp, uncomp, tx);
785         dsl_dir_diduse_space(ds->ds_dir, DD_USED_HEAD,
786             -used, -comp, -uncomp, tx);

```



```

787     dsl_dir_diduse_space(dp->dp_free_dir, DD_USED_HEAD,
788         used, comp, uncomp, tx);
789 }
791 if (ds->ds_prev != NULL) {
792     if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
793         VERIFY0(zap_remove_int(mos,
794             ds->ds_prev->ds_dir->dd_phys->dd_clones,
795             ds->ds_object, tx));
796     }
797     prevobj = ds->ds_prev->ds_object;
798     dsl_dataset_rele(ds->ds_prev, ds);
799     ds->ds_prev = NULL;
800 }
802 /*
803  * This must be done after the dsl_traverse(), because it will
804  * re-open the objset.
805  */
806 if (ds->ds_objset) {
807     dmu_objset_evict(ds->ds_objset);
808     ds->ds_objset = NULL;
809 }
811 /* Erase the link in the dir */
812 dmu_buf_will_dirty(ds->ds_dir->dd_dbuf, tx);
813 ds->ds_dir->dd_phys->dd_head_dataset_obj = 0;
814 ddobj = ds->ds_dir->dd_object;
815 ASSERT(ds->ds_phys->ds_snapnames_zapobj != 0);
816 VERIFY0(zap_destroy(mos, ds->ds_phys->ds_snapnames_zapobj, tx));
818 spa_prop_clear_bootfs(dp->dp_spa, ds->ds_object, tx);
820 ASSERT0(ds->ds_phys->ds_next_clones_obj);
821 ASSERT0(ds->ds_phys->ds_props_obj);
822 ASSERT0(ds->ds_phys->ds_userrefs_obj);
823 dsl_dir_rele(ds->ds_dir, ds);
824 ds->ds_dir = NULL;
825 VERIFY0(dmu_object_free(mos, obj, tx));
827 dsl_dir_destroy_sync(ddobj, tx);
829 if (rmorigin) {
830     dsl_dataset_t *prev;
831     VERIFY0(dsl_dataset_hold_obj(dp, prevobj, FTAG, &prev));
832     dsl_destroy_snapshot_sync_impl(prev, B_FALSE, tx);
833     dsl_dataset_rele(prev, FTAG);
834 }
835 }
837 static void
838 dsl_destroy_head_sync(void *arg, dmu_tx_t *tx)
839 {
840     dsl_destroy_head_arg_t *ddha = arg;
841     dsl_pool_t *dp = dmu_tx_pool(tx);
842     dsl_dataset_t *ds;
844     VERIFY0(dsl_dataset_hold(dp, ddha->ddha_name, FTAG, &ds));
845     dsl_destroy_head_sync_impl(ds, tx);
846     dsl_dataset_rele(ds, FTAG);
847 }
849 static void
850 dsl_destroy_head_begin_sync(void *arg, dmu_tx_t *tx)
851 {
852     dsl_destroy_head_arg_t *ddha = arg;

```

```

853     dsl_pool_t *dp = dmu_tx_pool(tx);
854     dsl_dataset_t *ds;
856     VERIFY0(dsl_dataset_hold(dp, ddha->ddha_name, FTAG, &ds));
858     /* Mark it as inconsistent on-disk, in case we crash */
859     dmu_buf_will_dirty(ds->ds_dbuf, tx);
860     ds->ds_phys->ds_flags |= DS_FLAG_INCONSISTENT;
862     spa_history_log_internal(ds, "destroy begin", tx, "");
863     dsl_dataset_rele(ds, FTAG);
864 }
866 int
867 dsl_destroy_head(const char *name)
868 {
869     dsl_destroy_head_arg_t ddha;
870     int error;
871     spa_t *spa;
872     boolean_t isenabled;
874 #ifdef _KERNEL
875     zfs_destroy_unmount_origin(name);
876 #endif
878     error = spa_open(name, &spa, FTAG);
879     if (error != 0)
880         return (error);
881     isenabled = spa_feature_is_enabled(spa,
882         &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY]);
883     spa_close(spa, FTAG);
885     ddha.ddha_name = name;
887     if (!isenabled) {
888         objset_t *os;
890         error = dsl_sync_task(name, dsl_destroy_head_check,
891             dsl_destroy_head_begin_sync, &ddha, 0);
892         if (error != 0)
893             return (error);
895         /*
896          * Head deletion is processed in one txg on old pools;
897          * remove the objects from open context so that the txg sync
898          * is not too long.
899          */
900         error = dmu_objset_own(name, DMU_OST_ANY, B_FALSE, FTAG, &os);
901         if (error == 0) {
902             uint64_t prev_snap_txg =
903                 dmu_objset_ds(os)->ds_phys->ds_prev_snap_txg;
904             for (uint64_t obj = 0; error == 0;
905                 error = dmu_object_next(os, &obj, FALSE,
906                     prev_snap_txg))
907                 (void) dmu_free_object(os, obj);
908             /* sync out all frees */
909             txg_wait_synced(dmu_objset_pool(os), 0);
910             dmu_objset_disown(os, FTAG);
911         }
912     }
914     return (dsl_sync_task(name, dsl_destroy_head_check,
915         dsl_destroy_head_sync, &ddha, 0));
916 }
918 /*

```

```
919 * Note, this function is used as the callback for dmu_objset_find(). We
920 * always return 0 so that we will continue to find and process
921 * inconsistent datasets, even if we encounter an error trying to
922 * process one of them.
923 */
924 /* ARGSUSED */
925 int
926 dsl_destroy_inconsistent(const char *dsname, void *arg)
927 {
928     objset_t *os;
929
930     if (dmu_objset_hold(dsname, FTAG, &os) == 0) {
931         boolean_t inconsistent = DS_IS_INCONSISTENT(dmu_objset_ds(os));
932         dmu_objset_rele(os, FTAG);
933         if (inconsistent)
934             (void) dsl_destroy_head(dsname);
935     }
936     return (0);
937 }
```

```

*****
29795 Wed May 29 20:27:09 2013
new/usr/src/uts/common/fs/zfs/dsl_pool.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23  * Copyright (c) 2013 by Delphix. All rights reserved.
24  * Copyright (c) 2013 Steven Hartland. All rights reserved.
25 #endif /* !codereview */
26 */

28 #include <sys/dsl_pool.h>
29 #include <sys/dsl_dataset.h>
30 #include <sys/dsl_prop.h>
31 #include <sys/dsl_dir.h>
32 #include <sys/dsl_synctask.h>
33 #include <sys/dsl_scan.h>
34 #include <sys/dnode.h>
35 #include <sys/dmu_tx.h>
36 #include <sys/dmu_objset.h>
37 #include <sys/arc.h>
38 #include <sys/zap.h>
39 #include <sys/zio.h>
40 #include <sys/zfs_context.h>
41 #include <sys/fs/zfs.h>
42 #include <sys/zfs_znode.h>
43 #include <sys/spa_impl.h>
44 #include <sys/dsl_deadlist.h>
45 #include <sys/bptree.h>
46 #include <sys/zfeature.h>
47 #include <sys/zil_impl.h>
48 #include <sys/dsl_userhold.h>

50 int zfs_no_write_throttle = 0;
51 int zfs_write_limit_shift = 3;
52 int zfs_txg_synctime_ms = 1000;
                    /* 1/8th of physical memory */
                    /* target milliseconds to sync a txg */

54 uint64_t zfs_write_limit_min = 32 << 20;
                    /* min write limit is 32MB */
55 uint64_t zfs_write_limit_max = 0;
                    /* max data payload per txg */
56 uint64_t zfs_write_limit_inflated = 0;
57 uint64_t zfs_write_limit_override = 0;

59 kmutex_t zfs_write_limit_lock;

```

```

61 static pgcnt_t old_physmem = 0;

63 hrtime_t zfs_throttle_delay = MSEC2NSEC(10);
64 hrtime_t zfs_throttle_resolution = MSEC2NSEC(10);

66 int
67 dsl_pool_open_special_dir(dsl_pool_t *dp, const char *name, dsl_dir_t **ddp)
68 {
69     uint64_t obj;
70     int err;

72     err = zap_lookup(dp->dp_meta_objset,
73                    dp->dp_root_dir->dd_phys->dd_child_dir_zapobj,
74                    name, sizeof(obj), 1, &obj);
75     if (err)
76         return (err);

78     return (dsl_dir_hold_obj(dp, obj, name, dp, ddp));
79 }

81 static dsl_pool_t *
82 dsl_pool_open_impl(spa_t *spa, uint64_t txg)
83 {
84     dsl_pool_t *dp;
85     blkptr_t *bp = spa_get_rootblkptr(spa);

87     dp = kmem_zalloc(sizeof(dsl_pool_t), KM_SLEEP);
88     dp->dp_spa = spa;
89     dp->dp_meta_rootbp = *bp;
90     rrw_init(&dp->dp_config_rwlock, B_TRUE);
91     dp->dp_write_limit = zfs_write_limit_min;
92     txg_init(dp, txg);

94     txg_list_create(&dp->dp_dirty_datasets,
95                   offsetof(dsl_dataset_t, ds_dirty_link));
96     txg_list_create(&dp->dp_dirty_zilogs,
97                   offsetof(zilog_t, zl_dirty_link));
98     txg_list_create(&dp->dp_dirty_dirs,
99                   offsetof(dsl_dir_t, dd_dirty_link));
100    txg_list_create(&dp->dp_sync_tasks,
101                  offsetof(dsl_sync_task_t, dst_node));

103    mutex_init(&dp->dp_lock, NULL, MUTEX_DEFAULT, NULL);

105    dp->dp_vnrele_taskq = taskq_create("zfs_vn_rele_taskq", 1, minclsyspri,
106                                     1, 4, 0);

108    return (dp);
109 }

111 int
112 dsl_pool_init(spa_t *spa, uint64_t txg, dsl_pool_t **dpp)
113 {
114     int err;
115     dsl_pool_t *dp = dsl_pool_open_impl(spa, txg);

117     err = dmu_objset_open_impl(spa, NULL, &dp->dp_meta_rootbp,
118                               &dp->dp_meta_objset);
119     if (err != 0)
120         dsl_pool_close(dp);
121     else
122         *dpp = dp;

124     return (err);
125 }

```

```

127 int
128 dsl_pool_open(dsl_pool_t *dp)
129 {
130     int err;
131     dsl_dir_t *dd;
132     dsl_dataset_t *ds;
133     uint64_t obj;

135     rrw_enter(&dp->dp_config_rwlock, RW_WRITER, FTAG);
136     err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
137         DMU_POOL_ROOT_DATASET, sizeof (uint64_t), 1,
138         &dp->dp_root_dir_obj);
139     if (err)
140         goto out;

142     err = dsl_dir_hold_obj(dp, dp->dp_root_dir_obj,
143         NULL, dp, &dp->dp_root_dir);
144     if (err)
145         goto out;

147     err = dsl_pool_open_special_dir(dp, MOS_DIR_NAME, &dp->dp_mos_dir);
148     if (err)
149         goto out;

151     if (spa_version(dp->dp_spa) >= SPA_VERSION_ORIGIN) {
152         err = dsl_pool_open_special_dir(dp, ORIGIN_DIR_NAME, &dd);
153         if (err)
154             goto out;
155         err = dsl_dataset_hold_obj(dp, dd->dd_phys->dd_head_dataset_obj,
156             FTAG, &ds);
157         if (err == 0) {
158             err = dsl_dataset_hold_obj(dp,
159                 ds->ds_phys->ds_prev_snap_obj, dp,
160                 &dp->dp_origin_snap);
161             dsl_dataset_rele(ds, FTAG);
162         }
163         dsl_dir_rele(dd, dp);
164         if (err)
165             goto out;
166     }

168     if (spa_version(dp->dp_spa) >= SPA_VERSION_DEADLISTS) {
169         err = dsl_pool_open_special_dir(dp, FREE_DIR_NAME,
170             &dp->dp_free_dir);
171         if (err)
172             goto out;

174         err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
175             DMU_POOL_FREE_BPOBJ, sizeof (uint64_t), 1, &obj);
176         if (err)
177             goto out;
178         VERIFY0(bpobj_open(&dp->dp_free_bpobj,
179             dp->dp_meta_objset, obj));
180     }

182     if (spa_feature_is_active(dp->dp_spa,
183         &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
184         err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
185             DMU_POOL_BPTREE_OBJ, sizeof (uint64_t), 1,
186             &dp->dp_bptree_obj);
187         if (err != 0)
188             goto out;
189     }

191     if (spa_feature_is_active(dp->dp_spa,
192         &spa_feature_table[SPA_FEATURE_EMPTY_BPOBJ])) {

```

```

193         err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
194             DMU_POOL_EMPTY_BPOBJ, sizeof (uint64_t), 1,
195             &dp->dp_empty_bpobj);
196         if (err != 0)
197             goto out;
198     }

200     err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
201         DMU_POOL_TMP_USERREFS, sizeof (uint64_t), 1,
202         &dp->dp_tmp_userrefs_obj);
203     if (err == ENOENT)
204         err = 0;
205     if (err)
206         goto out;

208     err = dsl_scan_init(dp, dp->dp_tx.tx_open_txg);

210 out:
211     rrw_exit(&dp->dp_config_rwlock, FTAG);
212     return (err);
213 }

215 void
216 dsl_pool_close(dsl_pool_t *dp)
217 {
218     /* drop our references from dsl_pool_open() */

220     /*
221      * Since we held the origin_snap from "syncing" context (which
222      * includes pool-opening context), it actually only got a "ref"
223      * and not a hold, so just drop that here.
224      */
225     if (dp->dp_origin_snap)
226         dsl_dataset_rele(dp->dp_origin_snap, dp);
227     if (dp->dp_mos_dir)
228         dsl_dir_rele(dp->dp_mos_dir, dp);
229     if (dp->dp_free_dir)
230         dsl_dir_rele(dp->dp_free_dir, dp);
231     if (dp->dp_root_dir)
232         dsl_dir_rele(dp->dp_root_dir, dp);

234     bpobj_close(&dp->dp_free_bpobj);

236     /* undo the dmub_objset_open_impl(mos) from dsl_pool_open() */
237     if (dp->dp_meta_objset)
238         dmub_objset_evict(dp->dp_meta_objset);

240     txg_list_destroy(&dp->dp_dirty_datasets);
241     txg_list_destroy(&dp->dp_dirty_zilogs);
242     txg_list_destroy(&dp->dp_sync_tasks);
243     txg_list_destroy(&dp->dp_dirty_dirs);

245     arc_flush(dp->dp_spa);
246     txg_fini(dp);
247     dsl_scan_fini(dp);
248     rrw_destroy(&dp->dp_config_rwlock);
249     mutex_destroy(&dp->dp_lock);
250     taskq_destroy(dp->dp_vnrele_taskq);
251     if (dp->dp_blkstats)
252         kmem_free(dp->dp_blkstats, sizeof (zfs_all_blkstats_t));
253     kmem_free(dp, sizeof (dsl_pool_t));
254 }

256 dsl_pool_t *
257 dsl_pool_create(spa_t *spa, nvlist_t *zplprops, uint64_t txg)
258 {

```

```

259     int err;
260     dsl_pool_t *dp = dsl_pool_open_impl(spa, txg);
261     dmu_tx_t *tx = dmu_tx_create_assigned(dp, txg);
262     objset_t *os;
263     dsl_dataset_t *ds;
264     uint64_t obj;

266     rrw_enter(&dp->dp_config_rwlock, RW_WRITER, FTAG);

268     /* create and open the MOS (meta-objset) */
269     dp->dp_meta_objset = dmu_objset_create_impl(spa,
270         NULL, &dp->dp_meta_rootbp, DMU_OST_META, tx);

272     /* create the pool directory */
273     err = zap_create_claim(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
274         DMU_OT_OBJECT_DIRECTORY, DMU_OT_NONE, 0, tx);
275     ASSERT0(err);

277     /* Initialize scan structures */
278     VERIFY0(dsl_scan_init(dp, txg));

280     /* create and open the root dir */
281     dp->dp_root_dir_obj = dsl_dir_create_sync(dp, NULL, NULL, tx);
282     VERIFY0(dsl_dir_hold_obj(dp, dp->dp_root_dir_obj,
283         NULL, dp, &dp->dp_root_dir));

285     /* create and open the meta-objset dir */
286     (void) dsl_dir_create_sync(dp, dp->dp_root_dir, MOS_DIR_NAME, tx);
287     VERIFY0(dsl_pool_open_special_dir(dp,
288         MOS_DIR_NAME, &dp->dp_mos_dir));

290     if (spa_version(spa) >= SPA_VERSION_DEADLISTS) {
291         /* create and open the free dir */
292         (void) dsl_dir_create_sync(dp, dp->dp_root_dir,
293             FREE_DIR_NAME, tx);
294         VERIFY0(dsl_pool_open_special_dir(dp,
295             FREE_DIR_NAME, &dp->dp_free_dir));

297         /* create and open the free bplist */
298         obj = bpobj_alloc(dp->dp_meta_objset, SPA_MAXBLOCKSIZE, tx);
299         VERIFY(zap_add(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
300             DMU_POOL_FREE_BPOBJ, sizeof(uint64_t), 1, &obj, tx) == 0);
301         VERIFY0(bpobj_open(&dp->dp_free_bpobj,
302             dp->dp_meta_objset, obj));
303     }

305     if (spa_version(spa) >= SPA_VERSION_DSL_SCRUB)
306         dsl_pool_create_origin(dp, tx);

308     /* create the root dataset */
309     obj = dsl_dataset_create_sync_dd(dp->dp_root_dir, NULL, 0, tx);

311     /* create the root objset */
312     VERIFY0(dsl_dataset_hold_obj(dp, obj, FTAG, &ds));
313     os = dmu_objset_create_impl(dp->dp_spa, ds,
314         dsl_dataset_get_blkptr(ds), DMU_OST_ZFS, tx);
315 #ifdef _KERNEL
316     zfs_create_fs(os, kored, zplprops, tx);
317 #endif
318     dsl_dataset_rele(ds, FTAG);

320     dmu_tx_commit(tx);

322     rrw_exit(&dp->dp_config_rwlock, FTAG);

324     return (dp);

```

```

325 }

327 /*
328  * Account for the meta-objset space in its placeholder dsl_dir.
329  */
330 void
331 dsl_pool_mos_diduse_space(dsl_pool_t *dp,
332     int64_t used, int64_t comp, int64_t uncomp)
333 {
334     ASSERT3U(comp, ==, uncomp); /* it's all metadata */
335     mutex_enter(&dp->dp_lock);
336     dp->dp_mos_used_delta += used;
337     dp->dp_mos_compressed_delta += comp;
338     dp->dp_mos_uncompressed_delta += uncomp;
339     mutex_exit(&dp->dp_lock);
340 }

342 static int
343 deadlist_enqueue_cb(void *arg, const blkptr_t *bp, dmu_tx_t *tx)
344 {
345     dsl_deadlist_t *dl = arg;
346     dsl_deadlist_insert(dl, bp, tx);
347     return (0);
348 }

350 void
351 dsl_pool_sync(dsl_pool_t *dp, uint64_t txg)
352 {
353     zio_t *zio;
354     dmu_tx_t *tx;
355     dsl_dir_t *dd;
356     dsl_dataset_t *ds;
357     objset_t *mos = dp->dp_meta_objset;
358     hrtime_t start, write_time;
359     uint64_t data_written;
360     int err;
361     list_t synced_datasets;

363     list_create(&synced_datasets, sizeof(dsl_dataset_t),
364         offsetof(dsl_dataset_t, ds_synced_link));

366     /*
367      * We need to copy dp_space_towrite() before doing
368      * dsl_sync_task_sync(), because
369      * dsl_dataset_snapshot_reserve_space() will increase
370      * dp_space_towrite but not actually write anything.
371      */
372     data_written = dp->dp_space_towrite[txg & TXG_MASK];

374     tx = dmu_tx_create_assigned(dp, txg);

376     dp->dp_read_overhead = 0;
377     start = gethrtime();

379     zio = zio_root(dp->dp_spa, NULL, NULL, ZIO_FLAG_MUSTSUCCEED);
380     while (ds = txg_list_remove(&dp->dp_dirty_datasets, txg)) {
381         /*
382          * We must not sync any non-MOS datasets twice, because
383          * we may have taken a snapshot of them. However, we
384          * may sync newly-created datasets on pass 2.
385          */
386         ASSERT(!list_link_active(&ds->ds_synced_link));
387         list_insert_tail(&synced_datasets, ds);
388         dsl_dataset_sync(ds, zio, tx);
389     }
390     DTRACE_PROBE(pool_sync_1setup);

```

```

391     err = zio_wait(zio);
393     write_time = gethrtime() - start;
394     ASSERT(err == 0);
395     DTRACE_PROBE(pool_sync_2rootzio);
397     /*
398      * After the data blocks have been written (ensured by the zio_wait()
399      * above), update the user/group space accounting.
400      */
401     for (ds = list_head(&synced_datasets); ds;
402          ds = list_next(&synced_datasets, ds))
403         dmu_objset_do_userquota_updates(ds->ds_objset, tx);
405     /*
406      * Sync the datasets again to push out the changes due to
407      * userspace updates. This must be done before we process the
408      * sync tasks, so that any snapshots will have the correct
409      * user accounting information (and we won't get confused
410      * about which blocks are part of the snapshot).
411      */
412     zio = zio_root(dp->dp_spa, NULL, NULL, ZIO_FLAG_MUSTSUCCEED);
413     while (ds = txg_list_remove(&dp->dp_dirty_datasets, txg)) {
414         ASSERT(list_link_active(&ds->ds_synced_link));
415         dmu_buf_rele(ds->ds_dbuf, ds);
416         dsl_dataset_sync(ds, zio, tx);
417     }
418     err = zio_wait(zio);
420     /*
421      * Now that the datasets have been completely synced, we can
422      * clean up our in-memory structures accumulated while syncing:
423      *
424      * - move dead blocks from the pending deadlist to the on-disk deadlist
425      * - release hold from dsl_dataset_dirty()
426      */
427     while (ds = list_remove_head(&synced_datasets)) {
428         objset_t *os = ds->ds_objset;
429         bplist_iterate(&ds->ds_pending_deadlist,
430                      deadlist_enqueue_cb, &ds->ds_deadlist, tx);
431         ASSERT(!dmu_objset_is_dirty(os, txg));
432         dmu_buf_rele(ds->ds_dbuf, ds);
433     }
435     start = gethrtime();
436     while (dd = txg_list_remove(&dp->dp_dirty_dirs, txg))
437         dsl_dir_sync(dd, tx);
438     write_time += gethrtime() - start;
440     /*
441      * The MOS's space is accounted for in the pool/$MOS
442      * (dp_mos_dir). We can't modify the mos while we're syncing
443      * it, so we remember the deltas and apply them here.
444      */
445     if (dp->dp_mos_used_delta != 0 || dp->dp_mos_compressed_delta != 0 ||
446         dp->dp_mos_uncompressed_delta != 0) {
447         dsl_dir_diduse_space(dp->dp_mos_dir, DD_USED_HEAD,
448                             dp->dp_mos_used_delta,
449                             dp->dp_mos_compressed_delta,
450                             dp->dp_mos_uncompressed_delta, tx);
451         dp->dp_mos_used_delta = 0;
452         dp->dp_mos_compressed_delta = 0;
453         dp->dp_mos_uncompressed_delta = 0;
454     }
456     start = gethrtime();

```

```

457     if (list_head(&mos->os_dirty_dnodes[txg & TXG_MASK]) != NULL ||
458         list_head(&mos->os_free_dnodes[txg & TXG_MASK]) != NULL) {
459         zio = zio_root(dp->dp_spa, NULL, NULL, ZIO_FLAG_MUSTSUCCEED);
460         dmu_objset_sync(mos, zio, tx);
461         err = zio_wait(zio);
462         ASSERT(err == 0);
463         dprintf_bp(&dp->dp_meta_rootbp, "meta objset rootbp is %s", "");
464         spa_set_rootblkptr(dp->dp_spa, &dp->dp_meta_rootbp);
465     }
466     write_time += gethrtime() - start;
467     DTRACE_PROBE2(pool_sync_4io, hrttime_t, write_time,
468                  hrttime_t, dp->dp_read_overhead);
469     write_time -= dp->dp_read_overhead;
471     /*
472      * If we modify a dataset in the same txg that we want to destroy it,
473      * its dsl_dir's dd_dbuf will be dirty, and thus have a hold on it.
474      * dsl_dir_destroy_check() will fail if there are unexpected holds.
475      * Therefore, we want to sync the MOS (thus syncing the dd_dbuf
476      * and clearing the hold on it) before we process the sync_tasks.
477      * The MOS data dirtied by the sync_tasks will be synced on the next
478      * pass.
479      */
480     DTRACE_PROBE(pool_sync_3task);
481     if (!txg_list_empty(&dp->dp_sync_tasks, txg)) {
482         dsl_sync_task_t *dst;
483         /*
484          * No more sync tasks should have been added while we
485          * were syncing.
486          */
487         ASSERT(spa_sync_pass(dp->dp_spa) == 1);
488         while (dst = txg_list_remove(&dp->dp_sync_tasks, txg))
489             dsl_sync_task_sync(dst, tx);
490     }
492     dmu_tx_commit(tx);
494     dp->dp_space_towrite[txg & TXG_MASK] = 0;
495     ASSERT(dp->dp_temppreserved[txg & TXG_MASK] == 0);
497     /*
498      * If the write limit max has not been explicitly set, set it
499      * to a fraction of available physical memory (default 1/8th).
500      * Note that we must inflate the limit because the spa
501      * inflates write sizes to account for data replication.
502      * Check this each sync phase to catch changing memory size.
503      */
504     if (physmem != old_physmem && zfs_write_limit_shift) {
505         mutex_enter(&zfs_write_limit_lock);
506         old_physmem = physmem;
507         zfs_write_limit_max = ptob(physmem) >> zfs_write_limit_shift;
508         zfs_write_limit_inflated = MAX(zfs_write_limit_min,
509                                       spa_get_asize(dp->dp_spa, zfs_write_limit_max));
510         mutex_exit(&zfs_write_limit_lock);
511     }
513     /*
514      * Attempt to keep the sync time consistent by adjusting the
515      * amount of write traffic allowed into each transaction group.
516      * Weight the throughput calculation towards the current value:
517      * thru = 3/4 old_thru + 1/4 new_thru
518      *
519      * Note: write_time is in nanosecs while dp_throughput is expressed in
520      * bytes per millisecond.
521      */
522     ASSERT(zfs_write_limit_min > 0);

```

```

523     if (data_written > zfs_write_limit_min / 8 &&
524         write_time > MSEC2NSEC(1)) {
525         uint64_t throughput = data_written / NSEC2MSEC(write_time);

527         if (dp->dp_throughput)
528             dp->dp_throughput = throughput / 4 +
529                 3 * dp->dp_throughput / 4;
530         else
531             dp->dp_throughput = throughput;
532         dp->dp_write_limit = MIN(zfs_write_limit_inflated,
533             MAX(zfs_write_limit_min,
534                 dp->dp_throughput * zfs_txg_synctime_ms));
535     }
536 }

538 void
539 dsl_pool_sync_done(dsl_pool_t *dp, uint64_t txg)
540 {
541     zillog_t *zillog;
542     dsl_dataset_t *ds;

544     while (zillog = txg_list_remove(&dp->dp_dirty_zilogs, txg)) {
545         ds = dmu_objset_ds(zilog->zl_os);
546         zil_clean(zilog, txg);
547         ASSERT(!dmu_objset_is_dirty(zilog->zl_os, txg));
548         dmu_buf_rele(ds->ds_dbuf, zillog);
549     }
550     ASSERT(!dmu_objset_is_dirty(dp->dp_meta_objset, txg));
551 }

553 /*
554  * TRUE if the current thread is the tx_sync_thread or if we
555  * are being called from SPA context during pool initialization.
556  */
557 int
558 dsl_pool_sync_context(dsl_pool_t *dp)
559 {
560     return (curthread == dp->dp_tx.tx_sync_thread ||
561         spa_is_initializing(dp->dp_spa));
562 }

564 uint64_t
565 dsl_pool_adjustedsize(dsl_pool_t *dp, boolean_t netfree)
566 {
567     uint64_t space, resv;

569     /*
570      * Reserve about 1.6% (1/64), or at least 32MB, for allocation
571      * efficiency.
572      * XXX The intent log is not accounted for, so it must fit
573      * within this slop.
574      *
575      * If we're trying to assess whether it's OK to do a free,
576      * cut the reservation in half to allow forward progress
577      * (e.g. make it possible to rm(1) files from a full pool).
578      */
579     space = spa_get_dspace(dp->dp_spa);
580     resv = MAX(space >> 6, SPA_MINDEVSZ >> 1);
581     if (netfree)
582         resv >>= 1;

584     return (space - resv);
585 }

587 int
588 dsl_pool_tempereserve_space(dsl_pool_t *dp, uint64_t space, dmu_tx_t *tx)

```

```

589 {
590     uint64_t reserved = 0;
591     uint64_t write_limit = (zfs_write_limit_override ?
592         zfs_write_limit_override : dp->dp_write_limit);

594     if (zfs_no_write_throttle) {
595         atomic_add_64(&dp->dp_tempereserved[tx->tx_txg & TXG_MASK],
596             space);
597         return (0);
598     }

600     /*
601      * Check to see if we have exceeded the maximum allowed IO for
602      * this transaction group. We can do this without locks since
603      * a little slop here is ok. Note that we do the reserved check
604      * with only half the requested reserve: this is because the
605      * reserve requests are worst-case, and we really don't want to
606      * throttle based off of worst-case estimates.
607      */
608     if (write_limit > 0) {
609         reserved = dp->dp_space_towrite[tx->tx_txg & TXG_MASK]
610             + dp->dp_tempereserved[tx->tx_txg & TXG_MASK] / 2;

612         if (reserved && reserved > write_limit)
613             return (SET_ERROR(ERESTART));
614     }

616     atomic_add_64(&dp->dp_tempereserved[tx->tx_txg & TXG_MASK], space);

618     /*
619      * If this transaction group is over 7/8ths capacity, delay
620      * the caller 1 clock tick. This will slow down the "fill"
621      * rate until the sync process can catch up with us.
622      */
623     if (reserved && reserved > (write_limit - (write_limit >> 3))) {
624         txg_delay(dp, tx->tx_txg, zfs_throttle_delay,
625             zfs_throttle_resolution);
626     }

628     return (0);
629 }

631 void
632 dsl_pool_tempereserve_clear(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx)
633 {
634     ASSERT(dp->dp_tempereserved[tx->tx_txg & TXG_MASK] >= space);
635     atomic_add_64(&dp->dp_tempereserved[tx->tx_txg & TXG_MASK], -space);
636 }

638 void
639 dsl_pool_memory_pressure(dsl_pool_t *dp)
640 {
641     uint64_t space_inuse = 0;
642     int i;

644     if (dp->dp_write_limit == zfs_write_limit_min)
645         return;

647     for (i = 0; i < TXG_SIZE; i++) {
648         space_inuse += dp->dp_space_towrite[i];
649         space_inuse += dp->dp_tempereserved[i];
650     }
651     dp->dp_write_limit = MAX(zfs_write_limit_min,
652         MIN(dp->dp_write_limit, space_inuse / 4));
653 }

```

```

655 void
656 dsl_pool_willuse_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx)
657 {
658     if (space > 0) {
659         mutex_enter(&dp->dp_lock);
660         dp->dp_space_towrite[tx->tx_txcg & TXG_MASK] += space;
661         mutex_exit(&dp->dp_lock);
662     }
663 }
664
665 /* ARGSUSED */
666 static int
667 upgrade_clones_cb(dsl_pool_t *dp, dsl_dataset_t *hds, void *arg)
668 {
669     dmu_tx_t *tx = arg;
670     dsl_dataset_t *ds, *prev = NULL;
671     int err;
672
673     err = dsl_dataset_hold_obj(dp, hds->ds_object, FTAG, &ds);
674     if (err)
675         return (err);
676
677     while (ds->ds_phys->ds_prev_snap_obj != 0) {
678         err = dsl_dataset_hold_obj(dp, ds->ds_phys->ds_prev_snap_obj,
679             FTAG, &prev);
680         if (err) {
681             dsl_dataset_rele(ds, FTAG);
682             return (err);
683         }
684
685         if (prev->ds_phys->ds_next_snap_obj != ds->ds_object)
686             break;
687         dsl_dataset_rele(ds, FTAG);
688         ds = prev;
689         prev = NULL;
690     }
691
692     if (prev == NULL) {
693         prev = dp->dp_origin_snap;
694
695         /*
696          * The $ORIGIN can't have any data, or the accounting
697          * will be wrong.
698          */
699         ASSERT0(prev->ds_phys->ds_bp.blk_birth);
700
701         /* The origin doesn't get attached to itself */
702         if (ds->ds_object == prev->ds_object) {
703             dsl_dataset_rele(ds, FTAG);
704             return (0);
705         }
706
707         dmu_buf_will_dirty(ds->ds_dbuf, tx);
708         ds->ds_phys->ds_prev_snap_obj = prev->ds_object;
709         ds->ds_phys->ds_prev_snap_txcg = prev->ds_phys->ds_creation_txcg;
710
711         dmu_buf_will_dirty(ds->ds_dir->dd_dbuf, tx);
712         ds->ds_dir->dd_phys->dd_origin_obj = prev->ds_object;
713
714         dmu_buf_will_dirty(prev->ds_dbuf, tx);
715         prev->ds_phys->ds_num_children++;
716
717         if (ds->ds_phys->ds_next_snap_obj == 0) {
718             ASSERT(ds->ds_prev == NULL);
719             VERIFY0(dsl_dataset_hold_obj(dp,
720                 ds->ds_phys->ds_prev_snap_obj, ds, &ds->ds_prev));

```

```

721     }
722 }
723
724 ASSERT3U(ds->ds_dir->dd_phys->dd_origin_obj, ==, prev->ds_object);
725 ASSERT3U(ds->ds_phys->ds_prev_snap_obj, ==, prev->ds_object);
726
727 if (prev->ds_phys->ds_next_clones_obj == 0) {
728     dmu_buf_will_dirty(prev->ds_dbuf, tx);
729     prev->ds_phys->ds_next_clones_obj =
730         zap_create(dp->dp_meta_objset,
731             DMU_OT_NEXT_CLONES, DMU_OT_NONE, 0, tx);
732 }
733 VERIFY0(zap_add_int(dp->dp_meta_objset,
734     prev->ds_phys->ds_next_clones_obj, ds->ds_object, tx));
735
736 dsl_dataset_rele(ds, FTAG);
737 if (prev != dp->dp_origin_snap)
738     dsl_dataset_rele(prev, FTAG);
739 return (0);
740 }
741
742 void
743 dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx)
744 {
745     ASSERT(dmu_tx_is_syncing(tx));
746     ASSERT(dp->dp_origin_snap != NULL);
747
748     VERIFY0(dmu_objset_find_dp(dp, dp->dp_root_dir_obj, upgrade_clones_cb,
749         tx, DS_FIND_CHILDREN));
750 }
751
752 /* ARGSUSED */
753 static int
754 upgrade_dir_clones_cb(dsl_pool_t *dp, dsl_dataset_t *ds, void *arg)
755 {
756     dmu_tx_t *tx = arg;
757     objset_t *mos = dp->dp_meta_objset;
758
759     if (ds->ds_dir->dd_phys->dd_origin_obj != 0) {
760         dsl_dataset_t *origin;
761
762         VERIFY0(dsl_dataset_hold_obj(dp,
763             ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &origin));
764
765         if (origin->ds_dir->dd_phys->dd_clones == 0) {
766             dmu_buf_will_dirty(origin->ds_dir->dd_dbuf, tx);
767             origin->ds_dir->dd_phys->dd_clones = zap_create(mos,
768                 DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
769         }
770
771         VERIFY0(zap_add_int(dp->dp_meta_objset,
772             origin->ds_dir->dd_phys->dd_clones, ds->ds_object, tx));
773
774         dsl_dataset_rele(origin, FTAG);
775     }
776     return (0);
777 }
778
779 void
780 dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx)
781 {
782     ASSERT(dmu_tx_is_syncing(tx));
783     uint64_t obj;
784
785     (void) dsl_dir_create_sync(dp, dp->dp_root_dir, FREE_DIR_NAME, tx);
786     VERIFY0(dsl_pool_open_special_dir(dp,

```



```

787     FREE_DIR_NAME, &dp->dp_free_dir));
788
789     /*
790     * We can't use bpobj_alloc(), because spa_version() still
791     * returns the old version, and we need a new-version bpobj with
792     * subobj support. So call dmu_object_alloc() directly.
793     */
794     obj = dmu_object_alloc(dp->dp_meta_objset, DMU_OT_BPOBJ,
795     SPA_MAXBLOCKSIZE, DMU_OT_BPOBJ_HDR, sizeof (bpobj_phys_t), tx);
796     VERIFY0(zap_add(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
797     DMU_POOL_FREE_BPOBJ, sizeof (uint64_t), 1, &obj, tx));
798     VERIFY0(bpobj_open(&dp->dp_free_bpobj, dp->dp_meta_objset, obj));
799
800     VERIFY0(dmu_objset_find_dp(dp, dp->dp_root_dir_obj,
801     upgrade_dir_clones_cb, tx, DS_FIND_CHILDREN));
802 }
803
804 void
805 dsl_pool_create_origin(dsl_pool_t *dp, dmu_tx_t *tx)
806 {
807     uint64_t dsobj;
808     dsl_dataset_t *ds;
809
810     ASSERT(dmu_tx_is_syncing(tx));
811     ASSERT(dp->dp_origin_snap == NULL);
812     ASSERT(rrw_held(&dp->dp_config_rwlock, RW_WRITER));
813
814     /* create the origin dir, ds, & snap-ds */
815     dsobj = dsl_dataset_create_sync(dp->dp_root_dir, ORIGIN_DIR_NAME,
816     NULL, 0, kcred, tx);
817     VERIFY0(dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
818     dsl_dataset_snapshot_sync_impl(ds, ORIGIN_DIR_NAME, tx);
819     VERIFY0(dsl_dataset_hold_obj(dp, ds->ds_phys->ds_prev_snap_obj,
820     dp, &dp->dp_origin_snap));
821     dsl_dataset_rele(ds, FTAG);
822 }
823
824 taskq_t *
825 dsl_pool_vnrele_taskq(dsl_pool_t *dp)
826 {
827     return (dp->dp_vnrele_taskq);
828 }
829
830 /*
831 * Walk through the pool-wide zap object of temporary snapshot user holds
832 * and release them.
833 */
834 void
835 dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp)
836 {
837     zap_attribute_t za;
838     zap_cursor_t zc;
839     objset_t *mos = dp->dp_meta_objset;
840     uint64_t zapobj = dp->dp_tmp_userrefs_obj;
841     nvlist_t *holds;
842 #endif /* ! codereview */
843
844     if (zapobj == 0)
845         return;
846     ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);
847
848     holds = fnvlist_alloc();
849
850 #endif /* ! codereview */
851     for (zap_cursor_init(&zc, mos, zapobj);
852         zap_cursor_retrieve(&zc, &za) == 0;

```

```

853         zap_cursor_advance(&zc)) {
854             char *htag;
855             uint64_t dsobj;
856             nvlist_t *tags;
857 #endif /* ! codereview */
858
859             htag = strchr(za.za_name, '-');
860             *htag = '\0';
861             ++htag;
862             if (nvlist_lookup_nvlist(holds, za.za_name, &tags) != 0) {
863                 tags = fnvlist_alloc();
864                 fnvlist_add_boolean(tags, htag);
865                 fnvlist_add_nvlist(holds, za.za_name, tags);
866                 fnvlist_free(tags);
867             } else {
868                 fnvlist_add_boolean(tags, htag);
869             }
870             dsobj = strtonum(za.za_name, NULL);
871             dsl_dataset_user_release_tmp(dp, dsobj, htag);
872             dsl_dataset_user_release_tmp(dp, holds);
873             fnvlist_free(holds);
874 #endif /* ! codereview */
875         zap_cursor_fini(&zc);
876     }
877
878     /*
879     * Create the pool-wide zap object for storing temporary snapshot holds.
880     */
881     void
882     dsl_pool_user_hold_create_obj(dsl_pool_t *dp, dmu_tx_t *tx)
883     {
884         objset_t *mos = dp->dp_meta_objset;
885
886         ASSERT(dp->dp_tmp_userrefs_obj == 0);
887         ASSERT(dmu_tx_is_syncing(tx));
888
889         dp->dp_tmp_userrefs_obj = zap_create_link(mos, DMU_OT_USERREFS,
890         DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_TMP_USERREFS, tx);
891     }
892
893     static int
894     dsl_pool_user_hold_rele_impl(dsl_pool_t *dp, uint64_t dsobj,
895     const char *tag, uint64_t now, dmu_tx_t *tx, boolean_t holding)
896     {
897         objset_t *mos = dp->dp_meta_objset;
898         uint64_t zapobj = dp->dp_tmp_userrefs_obj;
899         char *name;
900         int error;
901
902         ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);
903         ASSERT(dmu_tx_is_syncing(tx));
904
905         /*
906          * If the pool was created prior to SPA_VERSION_USERREFS, the
907          * zap object for temporary holds might not exist yet.
908          */
909         if (zapobj == 0) {
910             if (holding) {
911                 dsl_pool_user_hold_create_obj(dp, tx);
912                 zapobj = dp->dp_tmp_userrefs_obj;
913             } else {
914                 return (SET_ERROR(ENOENT));
915             }
916         }
917     }

```

```

917     name = kmem_asprintf("%llx-%s", (u_longlong_t)dsobj, tag);
918     if (holding)
919         error = zap_add(mos, zapobj, name, 8, 1, &now, tx);
920     else
921         error = zap_remove(mos, zapobj, name, tx);
922     strfree(name);
923
924     return (error);
925 }
926
927 /*
928 * Add a temporary hold for the given dataset object and tag.
929 */
930 int
931 dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj, const char *tag,
932                  uint64_t now, dmu_tx_t *tx)
933 {
934     return (dsl_pool_user_hold_rele_impl(dp, dsobj, tag, now, tx, B_TRUE));
935 }
936
937 /*
938 * Release a temporary hold for the given dataset object and tag.
939 */
940 int
941 dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj, const char *tag,
942                    dmu_tx_t *tx)
943 {
944     return (dsl_pool_user_hold_rele_impl(dp, dsobj, tag, NULL,
945                                       tx, B_FALSE));
946 }
947
948 /*
949 * DSL Pool Configuration Lock
950 *
951 * The dp_config_rwlock protects against changes to DSL state (e.g. dataset
952 * creation / destruction / rename / property setting). It must be held for
953 * read to hold a dataset or dsl_dir. I.e. you must call
954 * dsl_pool_config_enter() or dsl_pool_hold() before calling
955 * dsl_{dataset,dir}_hold_{obj}. In most circumstances, the dp_config_rwlock
956 * must be held continuously until all datasets and dsl_dirs are released.
957 *
958 * The only exception to this rule is that if a "long hold" is placed on
959 * a dataset, then the dp_config_rwlock may be dropped while the dataset
960 * is still held. The long hold will prevent the dataset from being
961 * destroyed -- the destroy will fail with EBUSY. A long hold can be
962 * obtained by calling dsl_dataset_long_hold(), or by "owning" a dataset
963 * (by calling dsl_{dataset,objset}_{try}own_{obj}).
964 *
965 * Legitimate long-holders (including owners) should be long-running, cancelable
966 * tasks that should cause "zfs destroy" to fail. This includes DMU
967 * consumers (i.e. a ZPL filesystem being mounted or ZVOL being open),
968 * "zfs send", and "zfs diff". There are several other long-holders whose
969 * uses are suboptimal (e.g. "zfs promote", and zil_suspend()).
970 *
971 * The usual formula for long-holding would be:
972 * dsl_pool_hold()
973 * dsl_dataset_hold()
974 * ... perform checks ...
975 * dsl_dataset_long_hold()
976 * dsl_pool_rele()
977 * ... perform long-running task ...
978 * dsl_dataset_long_rele()
979 * dsl_dataset_rele()
980 *
981 * Note that when the long hold is released, the dataset is still held but
982 * the pool is not held. The dataset may change arbitrarily during this time

```

```

983 * (e.g. it could be destroyed). Therefore you shouldn't do anything to the
984 * dataset except release it.
985 *
986 * User-initiated operations (e.g. ioctls, zfs_ioc_*) are either read-only
987 * or modifying operations.
988 *
989 * Modifying operations should generally use dsl_sync_task(). The sync task
990 * infrastructure enforces proper locking strategy with respect to the
991 * dp_config_rwlock. See the comment above dsl_sync_task() for details.
992 *
993 * Read-only operations will manually hold the pool, then the dataset, obtain
994 * information from the dataset, then release the pool and dataset.
995 * dmu_objset_{hold,rele}() are convenience routines that also do the pool
996 * hold/rele.
997 */
998
999 int
1000 dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp)
1001 {
1002     spa_t *spa;
1003     int error;
1004
1005     error = spa_open(name, &spa, tag);
1006     if (error == 0) {
1007         *dp = spa_get_dsl(spa);
1008         dsl_pool_config_enter(*dp, tag);
1009     }
1010     return (error);
1011 }
1012
1013 void
1014 dsl_pool_rele(dsl_pool_t *dp, void *tag)
1015 {
1016     dsl_pool_config_exit(dp, tag);
1017     spa_close(dp->dp_spa, tag);
1018 }
1019
1020 void
1021 dsl_pool_config_enter(dsl_pool_t *dp, void *tag)
1022 {
1023     /*
1024      * We use a "reentrant" reader-writer lock, but not reentrantly.
1025      *
1026      * The rrwlock can (with the track_all flag) track all reading threads,
1027      * which is very useful for debugging which code path failed to release
1028      * the lock, and for verifying that the *current* thread does hold
1029      * the lock.
1030      *
1031      * (Unlike a rwlock, which knows that N threads hold it for
1032      * read, but not *which* threads, so rw_held(RW_READER) returns TRUE
1033      * if any thread holds it for read, even if this thread doesn't).
1034      */
1035     ASSERT(!rrw_held(&dp->dp_config_rwlock, RW_READER));
1036     rrw_enter(&dp->dp_config_rwlock, RW_READER, tag);
1037 }
1038
1039 void
1040 dsl_pool_config_exit(dsl_pool_t *dp, void *tag)
1041 {
1042     rrw_exit(&dp->dp_config_rwlock, tag);
1043 }
1044
1045 boolean_t
1046 dsl_pool_config_held(dsl_pool_t *dp)
1047 {
1048     return (RRW_LOCK_HELD(&dp->dp_config_rwlock));

```

1049 }

```

*****
17479 Wed May 29 20:27:09 2013
new/usr/src/uts/common/fs/zfs/dsl_userhold.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2013 by Delphix. All rights reserved.
24 * Copyright (c) 2013 Steven Hartland. All rights reserved.
25 #endif /* ! codereview */
26 */

28 #include <sys/zfs_context.h>
29 #include <sys/dsl_userhold.h>
30 #include <sys/dsl_dataset.h>
31 #include <sys/dsl_destroy.h>
32 #include <sys/dsl_synctask.h>
33 #include <sys/dmu_tx.h>
34 #include <sys/zfs_onexit.h>
35 #include <sys/dsl_pool.h>
36 #include <sys/dsl_dir.h>
37 #include <sys/zfs_ioctl.h>
38 #include <sys/zap.h>

40 typedef struct dsl_dataset_user_hold_arg {
41     nvlist_t *dduha_holds;
42     nvlist_t *dduha_chkholds;
43 #endif /* ! codereview */
44     nvlist_t *dduha_errlist;
45     minor_t dduha_minor;
46 } dsl_dataset_user_hold_arg_t;

48 /*
49  * If you add new checks here, you may need to add additional checks to the
50  * "temporary" case in snapshot_check() in dmu_objset.c.
51  */
52 int
53 dsl_dataset_user_hold_check_one(dsl_dataset_t *ds, const char *htag,
54     boolean_t temphold, dmu_tx_t *tx)
55 {
56     dsl_pool_t *dp = dmu_tx_pool(tx);
57     objset_t *mos = dp->dp_meta_objset;
58     int error = 0;

60     ASSERT(RRW_READ_HELD(&dp->dp_config_rwlock));

```

```

62 #endif /* ! codereview */
63     if (strlen(htag) > MAXNAMELEN)
64         return (E2BIG);
65     /* Tempholds have a more restricted length */
66     if (temphold && strlen(htag) + MAX_TAG_PREFIX_LEN >= MAXNAMELEN)
67         return (E2BIG);

69     /* tags must be unique (if ds already exists) */
70     if (ds != NULL && ds->ds_phys->ds_userrefs_obj != 0) {
71         if (ds != NULL) {
72             mutex_enter(&ds->ds_lock);
73             if (ds->ds_phys->ds_userrefs_obj != 0) {
74                 uint64_t value;

73 #endif /* ! codereview */
74         error = zap_lookup(mos, ds->ds_phys->ds_userrefs_obj,
75             htag, 8, 1, &value);
76         if (error == 0)
77             error = SET_ERROR(EEXIST);
78         else if (error == ENOENT)
79             error = 0;
80         }
81     }
82     mutex_exit(&ds->ds_lock);

82     return (error);
83 }

85 static int
86 dsl_dataset_user_hold_check(void *arg, dmu_tx_t *tx)
87 {
88     dsl_dataset_user_hold_arg_t *dduha = arg;
89     dsl_pool_t *dp = dmu_tx_pool(tx);
90     nvpair_t *pair;
91     int rv = 0;

92     if (spa_version(dp->dp_spa) < SPA_VERSION_USERREFS)
93         return (SET_ERROR(ENOTSUP));

95     if (!dmu_tx_is_syncing(tx))
96         return (0);

98 #endif /* ! codereview */
99     for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;
100         pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {
101         dsl_dataset_t *ds;
102 #endif /* ! codereview */
103         int error = 0;
104         char *htag, *name;
105         dsl_dataset_t *ds;
106         char *htag;

106         /* must be a snapshot */
107         name = nvpair_name(pair);
108         if (strchr(name, '@') == NULL)
109             if (strchr(nvpair_name(pair), '@') == NULL)
110                 error = SET_ERROR(EINVAL);

111         if (error == 0)
112             error = nvpair_value_string(pair, &htag);

114         if (error == 0)
115             error = dsl_dataset_hold(dp, name, FTAG, &ds);

54         if (error == 0) {

```

```

55     error = dsl_dataset_hold(dp,
56         nvpair_name(pair), FTAG, &ds);
57     }
117     if (error == 0) {
118         error = dsl_dataset_user_hold_check_one(ds, htag,
119             dduha->dduha_minor != 0, tx);
120         dsl_dataset_rele(ds, FTAG);
121     }

123     if (error == 0) {
124         fnvlist_add_string(dduha->dduha_chkholds, name, htag);
125     } else {
126         /*
127          * We register ENOENT errors so they can be correctly
128          * reported if needed, such as when all holds fail.
129          */
130         fnvlist_add_int32(dduha->dduha_errlist, name, error);
131         if (error != ENOENT)
132             return (error);
133     }
134     if (error != 0) {
135         rv = error;
136         fnvlist_add_int32(dduha->dduha_errlist,
137             nvpair_name(pair), error);
138     }
139 }

140 return (0);
141 return (rv);
142 }

144 static void
145 dsl_dataset_user_hold_sync_one_impl(nvlist_t *tmpholds, dsl_dataset_t *ds,
146     const char *htag, minor_t minor, uint64_t now, dmu_tx_t *tx)
147 void
148 dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
149     minor_t minor, uint64_t now, dmu_tx_t *tx)
150 {
151     dsl_pool_t *dp = ds->ds_dir->dd_pool;
152     objset_t *mos = dp->dp_meta_objset;
153     uint64_t zapobj;

154     ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

155     mutex_enter(&ds->ds_lock);
156     if (ds->ds_phys->ds_userrefs_obj == 0) {
157         /*
158          * This is the first user hold for this dataset. Create
159          * the userrefs zap object.
160          */
161         dmu_buf_will_dirty(ds->ds_dbuf, tx);
162         zapobj = ds->ds_phys->ds_userrefs_obj =
163             zap_create(mos, DMU_OT_USERREFS, DMU_OT_NONE, 0, tx);
164     } else {
165         zapobj = ds->ds_phys->ds_userrefs_obj;
166     }
167     ds->ds_userrefs++;
168     mutex_exit(&ds->ds_lock);

169     VERIFY0(zap_add(mos, zapobj, htag, 8, 1, &now, tx));

170     if (minor != 0) {

```

```

170     char name[MAXNAMELEN];
171     nvlist_t *tags;

172 #endif /* ! codereview */
173 VERIFY0(dsl_pool_user_hold(dp, ds->ds_object,
174     htag, now, tx));
175     (void) snprintf(name, sizeof (name), "%llx",
176         (u_longlong_t)ds->ds_object);
177

178     if (nvlist_lookup_nvlist(tmpholds, name, &tags) != 0) {
179         tags = fnvlist_alloc();
180         fnvlist_add_boolean(tags, htag);
181         fnvlist_add_nvlist(tmpholds, name, tags);
182         fnvlist_free(tags);
183     } else {
184         fnvlist_add_boolean(tags, htag);
185     }
186     dsl_register_onexit_hold_cleanup(ds, htag, minor);
187 }

188 spa_history_log_internal_ds(ds, "hold", tx,
189     "tag=%s temp=%d refs=%llu",
190     htag, minor != 0, ds->ds_userrefs);
191 }

192 }

193 typedef struct zfs_hold_cleanup_arg {
194     char zhca_spaname[MAXNAMELEN];
195     uint64_t zhca_spa_load_guid;
196     nvlist_t *zhca_holds;
197 } zfs_hold_cleanup_arg_t;

200 static void
201 dsl_dataset_user_release_onexit(void *arg)
202 {
203     zfs_hold_cleanup_arg_t *ca = (zfs_hold_cleanup_arg_t *)arg;
204     spa_t *spa;
205     int error;

206     error = spa_open(ca->zhca_spaname, &spa, FTAG);
207     if (error != 0) {
208         zfs_dbgmsg("couldn't release holds on pool=%s "
209             "because pool is no longer loaded",
210             ca->zhca_spaname);
211         return;
212     }
213     if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
214         zfs_dbgmsg("couldn't release holds on pool=%s "
215             "because pool is no longer loaded (guid doesn't match)",
216             ca->zhca_spaname);
217         spa_close(spa, FTAG);
218         return;
219     }

220     (void) dsl_dataset_user_release_tmp(spa_get_dsl(spa), ca->zhca_holds);
221     fnvlist_free(ca->zhca_holds);
222     kmem_free(ca, sizeof (zfs_hold_cleanup_arg_t));
223     spa_close(spa, FTAG);
224 }

225 static void
226 dsl_onexit_hold_cleanup(spa_t *spa, nvlist_t *holds, minor_t minor)
227 {
228     zfs_hold_cleanup_arg_t *ca;

229     if (minor == 0 || nvlist_next_nvpair(holds, NULL) == NULL) {
230         fnvlist_free(holds);

```

```

235     return;
236 }
238 ASSERT(spa != NULL);
239 ca = kmem_alloc(sizeof (*ca), KM_SLEEP);
241 (void) strcpy(ca->zhca_spaname, spa_name(spa),
242             sizeof (ca->zhca_spaname));
243 ca->zhca_spa_load_guid = spa_load_guid(spa);
244 ca->zhca_holds = holds;
245 VERIFY0(zfs_onexit_add_cb(minor,
246                        dsl_dataset_user_release_onexit, ca, NULL));
247 }
249 void
250 dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
251                               minor_t minor, uint64_t now, dmu_tx_t *tx)
252 {
253     nvlist_t *tmpholds;
255     if (minor != 0)
256         tmpholds = fnvlist_alloc();
257     else
258         tmpholds = NULL;
259     dsl_dataset_user_hold_sync_one_impl(tmpholds, ds, htag, minor, now, tx);
260     dsl_onexit_hold_cleanup(dsl_dataset_get_spa(ds), tmpholds, minor);
261 }
263 #endif /* ! codereview */
264 static void
265 dsl_dataset_user_hold_sync(void *arg, dmu_tx_t *tx)
266 {
267     dsl_dataset_user_hold_arg_t *dduha = arg;
268     dsl_pool_t *dp = dmu_tx_pool(tx);
269     nvpair_t *pair;
270     nvlist_t *tmpholds;
271 #endif /* ! codereview */
272     uint64_t now = gethrstime_sec();
274     if (dduha->dduha_minor != 0)
275         tmpholds = fnvlist_alloc();
276     else
277         tmpholds = NULL;
278     for (pair = nvlist_next_nvpair(dduha->dduha_chkholds, NULL);
279          pair != NULL;
280          pair = nvlist_next_nvpair(dduha->dduha_chkholds, pair)) {
281         for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;
282              pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {
283             dsl_dataset_t *ds;
284             VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
285             dsl_dataset_user_hold_sync_one_impl(tmpholds, ds,
286                                                fnvpair_value_string(pair),
287                                                dduha->dduha_minor, now, tx);
288             dsl_dataset_user_hold_sync_one(ds, fnvpair_value_string(pair),
289                                           dduha->dduha_minor, now, tx);
287             dsl_dataset_rele(ds, FTAG);
288         }
289     }
290     dsl_onexit_hold_cleanup(dp->dp_spa, tmpholds, dduha->dduha_minor);
291 #endif /* ! codereview */
292 }
293 /*
294 * The full semantics of this function are described in the comment above
295 * lzc_hold().
296 */

```

```

297 * To summarize:
298 #endif /* ! codereview */
299 * holds is nvl of snapname -> holdname
300 * errlist will be filled in with snapname -> error
114 * if cleanup_minor is not 0, the holds will be temporary, cleaned up
115 * when the process exits.
301 *
302 * The snapholds must all be in the same pool.
303 *
304 * Holds for snapshots that don't exist will be skipped.
305 *
306 * If none of the snapshots for requested holds exist then ENOENT will be
307 * returned.
308 *
309 * If cleanup_minor is not 0, the holds will be temporary, which will be cleaned
310 * up when the process exits.
311 *
312 * On success all the holds, for snapshots that existed, will be created and 0
313 * will be returned.
314 *
315 * On failure no holds will be created, the errlist will be filled in,
316 * and an errno will returned.
317 *
318 * In all cases the errlist will contain entries for holds where the snapshot
319 * didn't exist.
117 * if any fails, all will fail.
320 */
321 int
322 dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor, nvlist_t *errlist)
323 {
324     dsl_dataset_user_hold_arg_t dduha;
325     nvpair_t *pair;
326     int ret;
327 #endif /* ! codereview */
329     pair = nvlist_next_nvpair(holds, NULL);
330     if (pair == NULL)
331         return (0);
333     dduha.dduha_holds = holds;
334     dduha.dduha_chkholds = fnvlist_alloc();
335 #endif /* ! codereview */
336     dduha.dduha_errlist = errlist;
337     dduha.dduha_minor = cleanup_minor;
339     ret = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
340                        dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds));
341     fnvlist_free(dduha.dduha_chkholds);
343     return (ret);
124     return (dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
125                          dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds)));
344 }
346 typedef int (dsl_holdfunc_t)(dsl_pool_t *dp, const char *name, void *tag,
347                             dsl_dataset_t **dsp);
349 #endif /* ! codereview */
350 typedef struct dsl_dataset_user_release_arg {
351     dsl_holdfunc_t *ddura_holdfunc;
352 #endif /* ! codereview */
353     nvlist_t *ddura_holds;
354     nvlist_t *ddura_todelete;
355     nvlist_t *ddura_errlist;
356     nvlist_t *ddura_chkholds;
357 #endif /* ! codereview */

```

```

358 } dsl_dataset_user_release_arg_t;

360 /* Place a dataset hold on the snapshot identified by passed dsobj string */
361 static int
362 dsl_dataset_hold_obj_string(dsl_pool_t *dp, const char *dsobj, void *tag,
363                             dsl_dataset_t **dsp)
364 {
365     return (dsl_dataset_hold_obj(dp, strtonum(dsobj, NULL), tag, dsp));
366 }

368 #endif /* ! codereview */
369 static int
370 dsl_dataset_user_release_check_one(dsl_dataset_user_release_arg_t *ddura,
371                                     dsl_dataset_t *ds, nvlist_t *holds, const char *name)
372 dsl_dataset_user_release_check_one(dsl_dataset_t *ds,
373                                     nvlist_t *holds, boolean_t *todelete)
374 {
375     uint64_t zapobj;
376     nvpair_t *pair;
377     nvlist_t *holds_found;
378 #endif /* ! codereview */
379     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
380     int ret, numholds;
381     int error;
382     int numholds = 0;

383     *todelete = B_FALSE;

384     if (!dsl_dataset_is_snapshot(ds))
385         return (SET_ERROR(EINVAL));

386     zapobj = ds->ds_phys->ds_userrefs_obj;
387     if (zapobj == 0)
388         return (SET_ERROR(ESRCH));

389     ret = 0;
390     numholds = 0;
391     holds_found = fnvlist_alloc();

392 #endif /* ! codereview */
393     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
394          pair = nvlist_next_nvpair(holds, pair)) {
395         /* Make sure the hold exists */
396         uint64_t tmp;
397         int error;
398         const char *name;

399         name = nvpair_name(pair);
400         error = zap_lookup(mos, zapobj, name, 8, 1, &tmp);

401         /* Non-existent holds aren't always an error. */
402         error = zap_lookup(mos, zapobj, nvpair_name(pair), 8, 1, &tmp);
403         if (error == ENOENT)
404             continue;

405         if (error != 0) {
406             fnvlist_free(holds_found);
407             error = SET_ERROR(ESRCH);
408             if (error != 0)
409                 return (error);
410         }

411         fnvlist_add_boolean(holds_found, name);
412 #endif /* ! codereview */
413         numholds++;
414     }

```

```

415     if (DS_IS_DEFER_DESTROY(ds) && ds->ds_phys->ds_num_children == 1 &&
416         ds->ds_userrefs == numholds) {
417         /* we need to destroy the snapshot as well */
418         if (dsl_dataset_long_held(ds)) {
419             fnvlist_free(holds_found);

420             if (dsl_dataset_long_held(ds))
421                 return (SET_ERROR(EBUSY));
422             *todelete = B_TRUE;
423             fnvlist_add_boolean(ddura->ddura_todelete, name);
424         }

425     if (numholds == 0)
426         ret = ENOENT;
427     else
428         fnvlist_add_nvlist(ddura->ddura_chkholds, name, holds_found);
429     fnvlist_free(holds_found);

430     return (ret);
431     return (0);
432 }

433 static int
434 dsl_dataset_user_release_check(void *arg, dmu_tx_t *tx)
435 {
436     dsl_dataset_user_release_arg_t *ddura;
437     dsl_holdfunc_t *holdfunc;
438     dsl_pool_t *dp;
439     dsl_dataset_user_release_arg_t *ddura = arg;
440     dsl_pool_t *dp = dmu_tx_pool(tx);
441     nvpair_t *pair;
442     int rv = 0;

443     if (!dmu_tx_is_syncing(tx))
444         return (0);

445     ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

446     dp = dmu_tx_pool(tx);
447     ddura = (dsl_dataset_user_release_arg_t *)arg;
448     holdfunc = ddura->ddura_holdfunc;

449 #endif /* ! codereview */
450     for (pair = nvlist_next_nvpair(ddura->ddura_holds, NULL); pair != NULL;
451          pair = nvlist_next_nvpair(ddura->ddura_holds, pair)) {
452         const char *name;
453         const char *name = nvpair_name(pair);
454         int error;
455         dsl_dataset_t *ds;
456         nvlist_t *holds;

457         name = nvpair_name(pair);
458 #endif /* ! codereview */
459         error = nvpair_value_nvlist(pair, &holds);
460         if (error != 0)
461             error = (SET_ERROR(EINVAL));
462         if (error == 0)
463             error = holdfunc(dp, name, FTAG, &ds);
464         return (SET_ERROR(EINVAL));

465     error = dsl_dataset_hold(dp, name, FTAG, &ds);
466     if (error == 0) {
467         error = dsl_dataset_user_release_check_one(ddura, ds,
468             holds, name);

```

```

180         boolean_t deleteme;
181         error = dsl_dataset_user_release_check_one(ds,
182             holds, &deleteme);
183         if (error == 0 && deleteme) {
184             fnvlist_add_boolean(ddura->ddura_todelete,
185                 name);
186         }
469         dsl_dataset_rele(ds, FTAG);
470     }
471     if (error != 0) {
472         if (ddura->ddura_errlist != NULL) {
473             fnvlist_add_int32(ddura->ddura_errlist, name,
474                 error);
475             fnvlist_add_int32(ddura->ddura_errlist,
476                 name, error);
477         }
478         /* Non-existent holds aren't always an error. */
479         if (error != ENOENT)
480             return (error);
481         rv = error;
482     }
483 }
484
485 /*
486  * Return ENOENT if none of the holds existed avoiding the overhead
487  * of a sync.
488  */
489 if (nvlist_next_nvpair(ddura->ddura_chkholds, NULL) == NULL)
490     return (ENOENT);
491
492 return (0);
493 return (rv);
494 }
495
496 static void
497 dsl_dataset_user_release_sync_one(dsl_dataset_user_release_arg_t *ddura,
498     dsl_dataset_t *ds, nvlist_t *holds, dmu_tx_t *tx)
499 {
500     dsl_dataset_user_release_sync_one(dsl_dataset_t *ds, nvlist_t *holds,
501     dmu_tx_t *tx)
502     {
503         dsl_pool_t *dp = ds->ds_dir->dd_pool;
504         objset_t *mos = dp->dp_meta_objset;
505         uint64_t zapobj;
506         int error;
507         nvpair_t *pair;
508
509         for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
510             pair = nvlist_next_nvpair(holds, pair)) {
511             int error;
512             const char *name;
513
514             name = nvpair_name(pair);
515
516             /* Remove temporary hold if one exists. */
517             error = dsl_pool_user_release(dp, ds->ds_object, name, tx);
518             VERIFY(error == 0 || error == ENOENT);
519
520             VERIFY0(zap_remove(mos, ds->ds_phys->ds_userrefs_obj, name,
521                 tx));
522             #endif /* ! codereview */
523             ds->ds_userrefs--;
524             error = dsl_pool_user_release(dp, ds->ds_object,
525                 nvpair_name(pair), tx);
526             VERIFY(error == 0 || error == ENOENT);
527             zapobj = ds->ds_phys->ds_userrefs_obj;
528             VERIFY0(zap_remove(mos, zapobj, nvpair_name(pair), tx));

```

```

516         spa_history_log_internal_ds(ds, "release", tx,
517             "tag=%s refs=%lld", name, (longlong_t)ds->ds_userrefs);
518         "tag=%s refs=%lld", nvpair_name(pair),
519             (longlong_t)ds->ds_userrefs);
520     }
521 }
522
523 static void
524 dsl_dataset_user_release_sync(void *arg, dmu_tx_t *tx)
525 {
526     dsl_dataset_user_release_arg_t *ddura = arg;
527     dsl_holdfunc_t *holdfunc = ddura->ddura_holdfunc;
528     #endif /* ! codereview */
529     dsl_pool_t *dp = dmu_tx_pool(tx);
530     nvpair_t *pair;
531
532     ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));
533
534     for (pair = nvlist_next_nvpair(ddura->ddura_chkholds, NULL);
535         pair != NULL; pair = nvlist_next_nvpair(ddura->ddura_chkholds,
536             pair)) {
537         for (pair = nvlist_next_nvpair(ddura->ddura_holds, NULL); pair != NULL;
538             pair = nvlist_next_nvpair(ddura->ddura_holds, pair)) {
539             dsl_dataset_t *ds;
540             const char *name;
541             #endif /* ! codereview */
542
543             name = nvpair_name(pair);
544             VERIFY0(holdfunc(dp, name, FTAG, &ds));
545
546             dsl_dataset_user_release_sync_one(ddura, ds,
547                 VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
548             dsl_dataset_user_release_sync_one(ds,
549                 nvpair_value_nvlist(pair), tx);
550             if (nvlist_exists(ddura->ddura_todelete, name)) {
551                 if (nvlist_exists(ddura->ddura_todelete,
552                     nvpair_name(pair))) {
553                     ASSERT(ds->ds_userrefs == 0 &&
554                         ds->ds_phys->ds_num_children == 1 &&
555                         DS_IS_DEFER_DESTROY(ds));
556                     dsl_destroy_snapshot_sync_impl(ds, B_FALSE, tx);
557                 }
558             }
559             dsl_dataset_rele(ds, FTAG);
560         }
561     }
562 }
563
564 /*
565  * The full semantics of this function are described in the comment above
566  * lzc_release().
567  *
568  * To summarize:
569  * Releases holds specified in the nvl holds.
570  *
571  * #endif /* ! codereview */
572  * holds is nvl of snapname -> { holdname, ... }
573  * errlist will be filled in with snapname -> error
574  *
575  * If tmpdp is not NULL the names for holds should be the dsobj's of snapshots,
576  * otherwise they should be the names of shapshots.
577  *
578  * As a release may cause snapshots to be destroyed this tries to ensure they
579  * aren't mounted.
580  *
581  * The release of non-existent holds are skipped.
582  */

```



```

573 * At least one hold must have been released for the this function to succeed
574 * and return 0.
246 * if any fails, all will fail.
575 */
576 static int
577 dsl_dataset_user_release_impl(nvlist_t *holds, nvlist_t *errlist,
578 dsl_pool_t *tmpdp)
248 int
249 dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist)
579 {
580     dsl_dataset_user_release_arg_t ddura;
581     nvpair_t *pair;
582     char *pool;
583 #endif /* ! codereview */
584     int error;

586     pair = nvlist_next_nvpair(holds, NULL);
587     if (pair == NULL)
588         return (0);

590 #ifndef _KERNEL
591     /*
592     * The release may cause snapshots to be destroyed; make sure they
593     * are not mounted.
594     */
595     if (tmpdp != NULL) {
596         /* Temporary holds are specified by dsobj string. */
597         ddura.ddura_holdfunc = dsl_dataset_hold_obj_string;
598         pool = spa_name(tmpdp->dp_spa);
599         ddura.ddura_holds = holds;
600         ddura.ddura_errlist = errlist;
601         ddura.ddura_todelete = fvnlist_alloc();

600         dsl_pool_config_enter(tmpdp, FTAG);
601         for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
602             pair = nvlist_next_nvpair(holds, pair)) {
257             error = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_release_check,
258                 dsl_dataset_user_release_sync, &ddura, fvnlist_num_pairs(holds));
259             fvnlist_free(ddura.ddura_todelete);
260             return (error);
261         }

263 typedef struct dsl_dataset_user_release_tmp_arg {
264     uint64_t ddurta_dsobj;
265     nvlist_t *ddurta_holds;
266     boolean_t ddurta_deleteme;
267 } dsl_dataset_user_release_tmp_arg_t;

269 static int
270 dsl_dataset_user_release_tmp_check(void *arg, dmu_tx_t *tx)
271 {
272     dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
273     dsl_pool_t *dp = dmu_tx_pool(tx);
274     dsl_dataset_t *ds;
275     int error;

605         error = dsl_dataset_hold_obj_string(tmpdp,
606             nvpair_name(pair), FTAG, &ds);
277     if (!dmu_tx_is_syncing(tx))
278         return (0);

280     error = dsl_dataset_hold_obj(dp, ddurta->ddurta_dsobj, FTAG, &ds);
281     if (error)
282         return (error);

284     error = dsl_dataset_user_release_check_one(ds,

```

```

285         ddurta->ddurta_holds, &ddurta->ddurta_deleteme);
286     dsl_dataset_rele(ds, FTAG);
287     return (error);
288 }

290 static void
291 dsl_dataset_user_release_tmp_sync(void *arg, dmu_tx_t *tx)
292 {
293     dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
294     dsl_pool_t *dp = dmu_tx_pool(tx);
295     dsl_dataset_t *ds;

297     VERIFY0(dsl_dataset_hold_obj(dp, ddurta->ddurta_dsobj, FTAG, &ds));
298     dsl_dataset_user_release_sync_one(ds, ddurta->ddurta_holds, tx);
299     if (ddurta->ddurta_deleteme) {
300         ASSERT(ds->ds_userrefs == 0 &&
301             ds->ds_phys->ds_num_children == 1 &&
302             DS_IS_DEFER_DESTROY(ds));
303         dsl_destroy_snapshot_sync_impl(ds, B_FALSE, tx);
304     }
305     dsl_dataset_rele(ds, FTAG);
306 }

308 /*
309 * Called at spa_load time to release a stale temporary user hold.
310 * Also called by the onexit code.
311 */
312 void
313 dsl_dataset_user_release_tmp(dsl_pool_t *dp, uint64_t dsobj, const char *htag)
314 {
315     dsl_dataset_user_release_tmp_arg_t ddurta;
316     dsl_dataset_t *ds;
317     int error;

319 #ifndef _KERNEL
320     /* Make sure it is not mounted. */
321     dsl_pool_config_enter(dp, FTAG);
322     error = dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds);
323     if (error == 0) {
324         char name[MAXNAMELEN];
325         dsl_dataset_name(ds, name);
326         dsl_dataset_rele(ds, FTAG);
327         dsl_pool_config_exit(dp, FTAG);
328         zfs_unmount_snap(name);
329     }
330     dsl_pool_config_exit(tmpdp, FTAG);
331 #endif /* ! codereview */
332 } else {
333     /* Non-temporary holds are specified by name. */
334     ddura.ddura_holdfunc = dsl_dataset_hold;
335     pool = nvpair_name(pair);

337     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
338         pair = nvlist_next_nvpair(holds, pair)) {
339         zfs_unmount_snap(nvpair_name(pair));
340     }
341     dsl_pool_config_exit(dp, FTAG);
342 }
343 #endif

345 ddura.ddura_holds = holds;
346 ddura.ddura_errlist = errlist;
347 ddura.ddura_todelete = fvnlist_alloc();
348 ddura.ddura_chkholds = fvnlist_alloc();

```

```

633     error = dsl_sync_task(pool, dsl_dataset_user_release_check,
634         dsl_dataset_user_release_sync, &ddura,
635         fnvlist_num_pairs(holds));
636     fnvlist_free(ddura.ddura_todelete);
637     fnvlist_free(ddura.ddura_chkholds);

639     return (error);
633     ddurta.ddurta_dsobj = dsobj;
634     ddurta.ddurta_holds = fnvlist_alloc();
635     fnvlist_add_boolean(ddurta.ddurta_holds, htag);

337     (void) dsl_sync_task(spa_name(dp->dp_spa),
338         dsl_dataset_user_release_tmp_check,
339         dsl_dataset_user_release_tmp_sync, &ddurta, 1);
340     fnvlist_free(ddurta.ddurta_holds);
640 }

642 /*
643  * holds is nvl of snapname -> { holdname, ... }
644  * errlist will be filled in with snapname -> error
645  */
646 int
647 dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist)
648 {
649     typedef struct zfs_hold_cleanup_arg {
650         char zhca_spaname[MAXNAMELEN];
651         uint64_t zhca_spa_load_guid;
652         uint64_t zhca_dsobj;
653         char zhca_htag[MAXNAMELEN];
654     } zfs_hold_cleanup_arg_t;

350 static void
351 dsl_dataset_user_release_onexit(void *arg)
648 {
649     return (dsl_dataset_user_release_impl(holds, errlist, NULL));
353     zfs_hold_cleanup_arg_t *ca = arg;
354     spa_t *spa;
355     int error;

357     error = spa_open(ca->zhca_spaname, &spa, FTAG);
358     if (error != 0) {
359         zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
360             "because pool is no longer loaded",
361             ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
362         return;
363     }
364     if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
365         zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
366             "because pool is no longer loaded (guid doesn't match)",
367             ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
368         spa_close(spa, FTAG);
369         return;
370     }

372     dsl_dataset_user_release_tmp(spa_get_dsl(spa),
373         ca->zhca_dsobj, ca->zhca_htag);
374     kmem_free(ca, sizeof (zfs_hold_cleanup_arg_t));
375     spa_close(spa, FTAG);
650 }

652 /*
653  * holds is nvl of snapsobj -> { holdname, ... }
654  */
655 #endif /* ! codereview */
656 void
657 dsl_dataset_user_release_tmp(struct dsl_pool *dp, nvlist_t *holds)
378 dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,

```

```

379     minor_t minor)
658 {
659     ASSERT(dp != NULL);
660     (void) dsl_dataset_user_release_impl(holds, NULL, dp);
381     zfs_hold_cleanup_arg_t *ca = kmem_alloc(sizeof (*ca), KM_SLEEP);
382     spa_t *spa = dsl_dataset_get_spa(ds);
383     (void) strncpy(ca->zhca_spaname, spa_name(spa),
384         sizeof (ca->zhca_spaname));
385     ca->zhca_spa_load_guid = spa_load_guid(spa);
386     ca->zhca_dsobj = ds->ds_object;
387     (void) strncpy(ca->zhca_htag, htag, sizeof (ca->zhca_htag));
388     VERIFY0(zfs_onexit_add_cb(minor,
389         dsl_dataset_user_release_onexit, ca, NULL));
661 }
_____unchanged_portion_omitted_

```

```

*****
10207 Wed May 29 20:27:09 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23  * Copyright (c) 2012 by Delphix. All rights reserved.
24  * Copyright (c) 2012, Joyent, Inc. All rights reserved.
25  * Copyright (c) 2013 Steven Hartland. All rights reserved.
26 #endif /* ! codereview */
27 */

29 #ifndef _SYS_DSL_DATASET_H
30 #define _SYS_DSL_DATASET_H

32 #include <sys/dmu.h>
33 #include <sys/spa.h>
34 #include <sys/txg.h>
35 #include <sys/zio.h>
36 #include <sys/bplist.h>
37 #include <sys/dsl_synctask.h>
38 #include <sys/zfs_context.h>
39 #include <sys/dsl_deadlist.h>
40 #include <sys/refcount.h>

42 #ifdef __cplusplus
43 extern "C" {
44 #endif

46 struct dsl_dataset;
47 struct dsl_dir;
48 struct dsl_pool;

50 #define DS_FLAG_INCONSISTENT (1ULL<<0)
51 #define DS_IS_INCONSISTENT(ds) \
52     ((ds)->ds_phys->ds_flags & DS_FLAG_INCONSISTENT)
53 /*
54  * Note: nopromote can not yet be set, but we want support for it in this
55  * on-disk version, so that we don't need to upgrade for it later.
56  */
57 #define DS_FLAG_NOPROMOTE (1ULL<<1)

59 /*
60  * DS_FLAG_UNIQUE_ACCURATE is set if ds_unique_bytes has been correctly

```

```

61  * calculated for head datasets (starting with SPA_VERSION_UNIQUE_ACCURATE,
62  * refquota/reservations).
63  */
64 #define DS_FLAG_UNIQUE_ACCURATE (1ULL<<2)

66 /*
67  * DS_FLAG_DEFER_DESTROY is set after 'zfs destroy -d' has been called
68  * on a dataset. This allows the dataset to be destroyed using 'zfs release'.
69  */
70 #define DS_FLAG_DEFER_DESTROY (1ULL<<3)
71 #define DS_IS_DEFER_DESTROY(ds) \
72     ((ds)->ds_phys->ds_flags & DS_FLAG_DEFER_DESTROY)

74 /*
75  * DS_FLAG_CI_DATASET is set if the dataset contains a file system whose
76  * name lookups should be performed case-insensitively.
77  */
78 #define DS_FLAG_CI_DATASET (1ULL<<16)

80 #define DS_CREATE_FLAG_NODIRTY (1ULL<<24)

82 typedef struct dsl_dataset_phys {
83     uint64_t ds_dir_obj; /* DMU_OT_DSL_DIR */
84     uint64_t ds_prev_snap_obj; /* DMU_OT_DSL_DATASET */
85     uint64_t ds_prev_snap_txg;
86     uint64_t ds_next_snap_obj; /* DMU_OT_DSL_DATASET */
87     uint64_t ds_snapnames_zapobj; /* DMU_OT_DSL_DS_SNAP_MAP 0 for snaps */
88     uint64_t ds_num_children; /* clone/snap children; ==0 for head */
89     uint64_t ds_creation_time; /* seconds since 1970 */
90     uint64_t ds_creation_txg;
91     uint64_t ds_deadlist_obj; /* DMU_OT_DEADLIST */
92     /*
93      * ds_referenced_bytes, ds_compressed_bytes, and ds_uncompressed_bytes
94      * include all blocks referenced by this dataset, including those
95      * shared with any other datasets.
96      */
97     uint64_t ds_referenced_bytes;
98     uint64_t ds_compressed_bytes;
99     uint64_t ds_uncompressed_bytes;
100    uint64_t ds_unique_bytes; /* only relevant to snapshots */
101    /*
102     * The ds_fsid_guid is a 56-bit ID that can change to avoid
103     * collisions. The ds_guid is a 64-bit ID that will never
104     * change, so there is a small probability that it will collide.
105     */
106    uint64_t ds_fsid_guid;
107    uint64_t ds_guid;
108    uint64_t ds_flags; /* DS_FLAG_* */
109    blkptr_t ds_bp;
110    uint64_t ds_next_clones_obj; /* DMU_OT_DSL_CLONES */
111    uint64_t ds_props_obj; /* DMU_OT_DSL_PROPS for snaps */
112    uint64_t ds_userrefs_obj; /* DMU_OT_USERREFS */
113    uint64_t ds_pad[5]; /* pad out to 320 bytes for good measure */
114 } dsl_dataset_phys_t;

116 typedef struct dsl_dataset {
117     /* Immutable: */
118     struct dsl_dir *ds_dir;
119     dsl_dataset_phys_t *ds_phys;
120     dmu_buf_t *ds_dbuf;
121     uint64_t ds_object;
122     uint64_t ds_fsid_guid;

124     /* only used in syncing context, only valid for non-snapshots: */
125     struct dsl_dataset *ds_prev;

```

```

127  /* has internal locking: */
128  dsl_deadlist_t ds_deadlist;
129  bplist_t ds_pending_deadlist;

131  /* protected by lock on pool's dp_dirty_datasets list */
132  txg_node_t ds_dirty_link;
133  list_node_t ds_synced_link;

135  /*
136   * ds_phys->ds_accounting is also protected by ds_lock.
137   * Protected by ds_lock:
138   */
139  kmutex_t ds_lock;
140  objset_t *ds_objset;
141  uint64_t ds_userrefs;
142  void *ds_owner;

144  /*
145   * Long holds prevent the ds from being destroyed; they allow the
146   * ds to remain held even after dropping the dp_config_rwlock.
147   * Owning counts as a long hold. See the comments above
148   * dsl_pool_hold() for details.
149   */
150  refcount_t ds_longholds;

152  /* no locking; only for making guesses */
153  uint64_t ds_trysnap_txg;

155  /* for objset_open() */
156  kmutex_t ds_opening_lock;

158  uint64_t ds_reserved; /* cached reservation */
159  uint64_t ds_quota; /* cached refquota */

161  kmutex_t ds_sendstream_lock;
162  list_t ds_sendstreams;

164  /* Protected by ds_lock; keep at end of struct for better locality */
165  char ds_snapname[MAXNAMELEN];
166 } dsl_dataset_t;

168 /*
169  * The max length of a temporary tag prefix is the number of hex digits
170  * required to express UINTE64_MAX plus one for the hyphen.
171  */
172 #define MAX_TAG_PREFIX_LEN 17

174 #define dsl_dataset_is_snapshot(ds) \
175 ((ds)->ds_phys->ds_num_children != 0)

177 #define DS_UNIQUE_IS_ACCURATE(ds) \
178 (((ds)->ds_phys->ds_flags & DS_FLAG_UNIQUE_ACCURATE) != 0)

180 int dsl_dataset_hold(struct dsl_pool *dp, const char *name, void *tag,
181 dsl_dataset_t **dsp);
182 int dsl_dataset_hold_obj(struct dsl_pool *dp, uint64_t dsobj, void *tag,
183 dsl_dataset_t **);
184 void dsl_dataset_rele(dsl_dataset_t *ds, void *tag);
185 int dsl_dataset_own(struct dsl_pool *dp, const char *name,
186 void *tag, dsl_dataset_t **dsp);
187 int dsl_dataset_own_obj(struct dsl_pool *dp, uint64_t dsobj,
188 void *tag, dsl_dataset_t **dsp);
189 void dsl_dataset_disown(dsl_dataset_t *ds, void *tag);
190 void dsl_dataset_name(dsl_dataset_t *ds, char *name);
191 boolean_t dsl_dataset_tryown(dsl_dataset_t *ds, void *tag);
25 void dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,

```

```

26  minor_t minor);
192 uint64_t dsl_dataset_create_sync(dsl_dir_t *pds, const char *lastname,
193 dsl_dataset_t *origin, uint64_t flags, cred_t *, dmu_tx_t *);
194 uint64_t dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
195 uint64_t flags, dmu_tx_t *tx);
196 int dsl_dataset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors);
197 int dsl_dataset_promote(const char *name, char *conflsnap);
198 int dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
199 boolean_t force);
200 int dsl_dataset_rename_snapshot(const char *fsname,
201 const char *oldsnapname, const char *newsnapname, boolean_t recursive);
202 int dsl_dataset_snapshot_tmp(const char *fsname, const char *snapname,
203 minor_t cleanup_minor, const char *htag);

205 blkptr_t *dsl_dataset_get_blkptr(dsl_dataset_t *ds);
206 void dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx);

208 spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);

210 boolean_t dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds);

212 void dsl_dataset_sync(dsl_dataset_t *os, zio_t *zio, dmu_tx_t *tx);

214 void dsl_dataset_block_born(dsl_dataset_t *ds, const blkptr_t *bp,
215 dmu_tx_t *tx);
216 int dsl_dataset_block_kill(dsl_dataset_t *ds, const blkptr_t *bp,
217 dmu_tx_t *tx, boolean_t async);
218 boolean_t dsl_dataset_block_freeable(dsl_dataset_t *ds, const blkptr_t *bp,
219 uint64_t blk_birth);
220 uint64_t dsl_dataset_prev_snap_txg(dsl_dataset_t *ds);

222 void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
223 void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);
224 void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
225 void dsl_dataset_space(dsl_dataset_t *ds,
226 uint64_t *refdbytesp, uint64_t *availbytesp,
227 uint64_t *usedobjsp, uint64_t *availobjsp);
228 uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
229 int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
230 uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
231 int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
232 uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
233 boolean_t dsl_dataset_is_dirty(dsl_dataset_t *ds);

235 int dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf);

237 int dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
238 uint64_t asize, uint64_t inflight, uint64_t *used,
239 uint64_t *ref_rsrv);
240 int dsl_dataset_set_refquota(const char *dsname, zprop_source_t source,
241 uint64_t quota);
242 int dsl_dataset_set_reservation(const char *dsname, zprop_source_t source,
243 uint64_t reservation);

245 boolean_t dsl_dataset_is_before(dsl_dataset_t *later, dsl_dataset_t *earlier);
246 void dsl_dataset_long_hold(dsl_dataset_t *ds, void *tag);
247 void dsl_dataset_long_rele(dsl_dataset_t *ds, void *tag);
248 boolean_t dsl_dataset_long_held(dsl_dataset_t *ds);

250 int dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
251 dsl_dataset_t *origin_head, boolean_t force);
252 void dsl_dataset_clone_swap_sync_impl(dsl_dataset_t *clone,
253 dsl_dataset_t *origin_head, dmu_tx_t *tx);
254 int dsl_dataset_snapshot_check_impl(dsl_dataset_t *ds, const char *snapname,
255 dmu_tx_t *tx);
256 void dsl_dataset_snapshot_sync_impl(dsl_dataset_t *ds, const char *snapname,

```

```
257     dmu_tx_t *tx);
259 void dsl_dataset_remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj,
260     dmu_tx_t *tx);
261 void dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds);
262 int dsl_dataset_get_snapname(dsl_dataset_t *ds);
263 int dsl_dataset_snap_lookup(dsl_dataset_t *ds, const char *name,
264     uint64_t *value);
265 int dsl_dataset_snap_remove(dsl_dataset_t *ds, const char *name, dmu_tx_t *tx);
266 void dsl_dataset_set_refreservation_sync_impl(dsl_dataset_t *ds,
267     zprop_source_t source, uint64_t value, dmu_tx_t *tx);
268 int dsl_dataset_rollback(const char *fsname);
270 #ifdef ZFS_DEBUG
271 #define dprintf_ds(ds, fmt, ...) do { \
272     if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
273         char *__ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
274         dsl_dataset_name(ds, __ds_name); \
275         dprintf("ds=%s " fmt, __ds_name, __VA_ARGS__); \
276         kmem_free(__ds_name, MAXNAMELEN); \
277     } \
278     _NOTE(CONSTCOND) } while (0)
279 #else
280 #define dprintf_ds(dd, fmt, ...)
281 #endif
283 #ifdef __cplusplus
284 }
285 unchanged_portion_omitted
```

new/usr/src/uts/common/fs/zfs/sys/dsl\_userhold.h

1

```
*****
1886 Wed May 29 20:27:09 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_userhold.h
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****

2 /*
3  * CDDL HEADER START
4  *
5  * The contents of this file are subject to the terms of the
6  * Common Development and Distribution License (the "License").
7  * You may not use this file except in compliance with the License.
8  *
9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 * or http://www.opensolaris.org/os/licensing.
11 * See the License for the specific language governing permissions
12 * and limitations under the License.
13 *
14 * When distributing Covered Code, include this CDDL HEADER in each
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 * If applicable, add the following below this CDDL HEADER, with the
17 * fields enclosed by brackets "[]" replaced with your own identifying
18 * information: Portions Copyright [yyyy] [name of copyright owner]
19 *
20 * CDDL HEADER END
21 */
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, Joyent, Inc. All rights reserved.
26  * Copyright (c) 2013 Steven Hartland. All rights reserved.
27 #endif /* ! codereview */
28 */

30 #ifndef _SYS_DSL_USERHOLD_H
31 #define _SYS_DSL_USERHOLD_H

33 #include <sys/nvpair.h>
34 #include <sys/types.h>

36 #ifdef __cplusplus
37 extern "C" {
38 #endif

40 struct dsl_pool;
41 struct dsl_dataset;
42 struct dmu_tx;

44 int dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor,
45     nvlist_t *errlist);
46 int dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist);
47 int dsl_dataset_get_holds(const char *dsname, nvlist_t *nvl);
48 void dsl_dataset_user_release_tmp(struct dsl_pool *dp, nvlist_t *holds);
49 void dsl_dataset_user_release_tmp(struct dsl_pool *dp, uint64_t dsobj,
50     const char *htag);
49 int dsl_dataset_user_hold_check_one(struct dsl_dataset *ds, const char *htag,
50     boolean_t temphold, struct dmu_tx *tx);
51 void dsl_dataset_user_hold_sync_one(struct dsl_dataset *ds, const char *htag,
52     minor_t minor, uint64_t now, struct dmu_tx *tx);

54 #ifdef __cplusplus
55 }
56 #endif

58 #endif /* _SYS_DSL_USERHOLD_H */
```

```

*****
143944 Wed May 29 20:27:10 2013
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
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 * Portions Copyright 2011 Martin Matuska
25 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
26 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
27 * Copyright (c) 2013 by Delphix. All rights reserved.
28 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
29 * Copyright (c) 2013 Steven Hartland. All rights reserved.
30 #endif /* !codereview */
31 */

33 /*
34 * ZFS ioctls.
35 *
36 * This file handles the ioctls to /dev/zfs, used for configuring ZFS storage
37 * pools and filesystems, e.g. with /sbin/zfs and /sbin/zpool.
38 *
39 * There are two ways that we handle ioctls: the legacy way where almost
40 * all of the logic is in the ioctl callback, and the new way where most
41 * of the marshalling is handled in the common entry point, zfsdev_ioctl().
42 *
43 * Non-legacy ioctls should be registered by calling
44 * zfs_ioctl_register() from zfs_ioctl_init(). The ioctl is invoked
45 * from userland by lzc_ioctl().
46 *
47 * The registration arguments are as follows:
48 *
49 * const char *name
50 *   The name of the ioctl. This is used for history logging. If the
51 *   ioctl returns successfully (the callback returns 0), and allow_log
52 *   is true, then a history log entry will be recorded with the input &
53 *   output nvlists. The log entry can be printed with "zpool history -i".
54 *
55 * zfs_ioc_t ioc
56 *   The ioctl request number, which userland will pass to ioctl(2).
57 *   The ioctl numbers can change from release to release, because
58 *   the caller (libzfs) must be matched to the kernel.
59 *
60 * zfs_secpolicy_func_t *secpolicy

```

```

61 * This function will be called before the zfs_ioc_func_t, to
62 * determine if this operation is permitted. It should return EPERM
63 * on failure, and 0 on success. Checks include determining if the
64 * dataset is visible in this zone, and if the user has either all
65 * zfs privileges in the zone (SYS MOUNT), or has been granted permission
66 * to do this operation on this dataset with "zfs allow".
67 *
68 * zfs_ioc_namecheck_t namecheck
69 * This specifies what to expect in the zfs_cmd_t:zc_name -- a pool
70 * name, a dataset name, or nothing. If the name is not well-formed,
71 * the ioctl will fail and the callback will not be called.
72 * Therefore, the callback can assume that the name is well-formed
73 * (e.g. is null-terminated, doesn't have more than one '@' character,
74 * doesn't have invalid characters).
75 *
76 * zfs_ioc_poolcheck_t pool_check
77 * This specifies requirements on the pool state. If the pool does
78 * not meet them (is suspended or is readonly), the ioctl will fail
79 * and the callback will not be called. If any checks are specified
80 * (i.e. it is not POOL_CHECK_NONE), namecheck must not be NO_NAME.
81 * Multiple checks can be or-ed together (e.g. POOL_CHECK_SUSPENDED |
82 * POOL_CHECK_READONLY).
83 *
84 * boolean_t smush_outnvlst
85 * If smush_outnvlst is true, then the output is presumed to be a
86 * list of errors, and it will be "smushed" down to fit into the
87 * caller's buffer, by removing some entries and replacing them with a
88 * single "N_MORE_ERRORS" entry indicating how many were removed. See
89 * nvlst_smush() for details. If smush_outnvlst is false, and the
90 * outnvlst does not fit into the userland-provided buffer, then the
91 * ioctl will fail with ENOMEM.
92 *
93 * zfs_ioc_func_t *func
94 * The callback function that will perform the operation.
95 *
96 * The callback should return 0 on success, or an error number on
97 * failure. If the function fails, the userland ioctl will return -1,
98 * and errno will be set to the callback's return value. The callback
99 * will be called with the following arguments:
100 *
101 * const char *name
102 *   The name of the pool or dataset to operate on, from
103 *   zfs_cmd_t:zc_name. The 'namecheck' argument specifies the
104 *   expected type (pool, dataset, or none).
105 *
106 * nvlst_t *innvl
107 *   The input nvlst, deserialized from zfs_cmd_t:zc_nvlst_src. Or
108 *   NULL if no input nvlst was provided. Changes to this nvlst are
109 *   ignored. If the input nvlst could not be deserialized, the
110 *   ioctl will fail and the callback will not be called.
111 *
112 * nvlst_t *outnvl
113 *   The output nvlst, initially empty. The callback can fill it in,
114 *   and it will be returned to userland by serializing it into
115 *   zfs_cmd_t:zc_nvlst_dst. If it is non-empty, and serialization
116 *   fails (e.g. because the caller didn't supply a large enough
117 *   buffer), then the overall ioctl will fail. See the
118 *   'smush_nvlst' argument above for additional behaviors.
119 *
120 * There are two typical uses of the output nvlst:
121 * - To return state, e.g. property values. In this case,
122 *   smush_outnvlst should be false. If the buffer was not large
123 *   enough, the caller will reallocate a larger buffer and try
124 *   the ioctl again.
125 *
126 * - To return multiple errors from an ioctl which makes on-disk

```

```

127 *      changes.  In this case, smush_outnvlst should be true.
128 *      Ioctls which make on-disk modifications should generally not
129 *      use the outnvl if they succeed, because the caller can not
130 *      distinguish between the operation failing, and
131 *      deserialization failing.
132 */

134 #include <sys/types.h>
135 #include <sys/param.h>
136 #include <sys/errno.h>
137 #include <sys/uio.h>
138 #include <sys/buf.h>
139 #include <sys/modctl.h>
140 #include <sys/open.h>
141 #include <sys/file.h>
142 #include <sys/kmem.h>
143 #include <sys/conf.h>
144 #include <sys/cmn_err.h>
145 #include <sys/stat.h>
146 #include <sys/zfs_ioctl.h>
147 #include <sys/zfs_vfsops.h>
148 #include <sys/zfs_znode.h>
149 #include <sys/zap.h>
150 #include <sys/spa.h>
151 #include <sys/spa_impl.h>
152 #include <sys/vdev.h>
153 #include <sys/priv_impl.h>
154 #include <sys/dmu.h>
155 #include <sys/dsl_dir.h>
156 #include <sys/dsl_dataset.h>
157 #include <sys/dsl_prop.h>
158 #include <sys/dsl_deleg.h>
159 #include <sys/dmu_objset.h>
160 #include <sys/dmu_impl.h>
161 #include <sys/dmu_tx.h>
162 #include <sys/ddi.h>
163 #include <sys/sunddi.h>
164 #include <sys/sunldi.h>
165 #include <sys/policy.h>
166 #include <sys/zone.h>
167 #include <sys/nvpair.h>
168 #include <sys/pathname.h>
169 #include <sys/mount.h>
170 #include <sys/sdt.h>
171 #include <sys/fs/zfs.h>
172 #include <sys/zfs_ctldir.h>
173 #include <sys/zfs_dir.h>
174 #include <sys/zfs_onexit.h>
175 #include <sys/zvol.h>
176 #include <sys/dsl_scan.h>
177 #include <sharefs/share.h>
178 #include <sys/dmu_objset.h>
179 #include <sys/dmu_send.h>
180 #include <sys/dsl_destroy.h>
181 #include <sys/dsl_userhold.h>
182 #include <sys/zfeature.h>

184 #include "zfs_namecheck.h"
185 #include "zfs_prop.h"
186 #include "zfs_deleg.h"
187 #include "zfs_comutil.h"

189 extern struct modlfs zfs_modlfs;

191 extern void zfs_init(void);
192 extern void zfs_fini(void);

```

```

194 ldi_ident_t zfs_li = NULL;
195 dev_info_t *zfs_dip;

197 uint_t zfs_fsyncer_key;
198 extern uint_t rrw_tsd_key;
199 static uint_t zfs_allow_log_key;

201 typedef int zfs_ioc_legacy_func_t(zfs_cmd_t *);
202 typedef int zfs_ioc_func_t(const char *, nvlist_t *, nvlist_t *);
203 typedef int zfs_secpolicy_func_t(zfs_cmd_t *, nvlist_t *, cred_t *);

205 typedef enum {
206     NO_NAME,
207     POOL_NAME,
208     DATASET_NAME
209 } zfs_ioc_namecheck_t;

211 typedef enum {
212     POOL_CHECK_NONE           = 1 << 0,
213     POOL_CHECK_SUSPENDED     = 1 << 1,
214     POOL_CHECK_READONLY     = 1 << 2,
215 } zfs_ioc_poolcheck_t;

217 typedef struct zfs_ioc_vec {
218     zfs_ioc_legacy_func_t *zvec_legacy_func;
219     zfs_ioc_func_t *zvec_func;
220     zfs_secpolicy_func_t *zvec_secpolicy;
221     zfs_ioc_namecheck_t zvec_namecheck;
222     boolean_t zvec_allow_log;
223     zfs_ioc_poolcheck_t zvec_pool_check;
224     boolean_t zvec_smush_outnvlst;
225     const char *zvec_name;
226 } zfs_ioc_vec_t;

228 /* This array is indexed by zfs_userquota_prop_t */
229 static const char *userquota_perms[] = {
230     ZFS_DELEG_PERM_USERUSED,
231     ZFS_DELEG_PERM_USERQUOTA,
232     ZFS_DELEG_PERM_GROUPUSED,
233     ZFS_DELEG_PERM_GROUPQUOTA,
234 };

236 static int zfs_ioc_userspace_upgrade(zfs_cmd_t *zc);
237 static int zfs_check_settable(const char *name, nvpair_t *property,
238     cred_t *cr);
239 static int zfs_check_clearable(char *dataset, nvlist_t *props,
240     nvlist_t **errors);
241 static int zfs_fill_zplprops_root(uint64_t, nvlist_t *, nvlist_t *,
242     boolean_t *);
243 int zfs_set_prop_nvlist(const char *, zprop_source_t, nvlist_t *, nvlist_t *);
244 static int get_nvlist(uint64_t nvl, uint64_t size, int iflag, nvlist_t **nvp);

246 static int zfs_prop_activate_feature(spa_t *spa, zfeature_info_t *feature);

248 /* _NOTE(PRINTFLIKE(4)) - this is printf-like, but lint is too whiney */
249 void
250 _dprintf(const char *file, const char *func, int line, const char *fmt, ...)
251 {
252     const char *newfile;
253     char buf[512];
254     va_list adx;

256     /*
257      * Get rid of annoying "../common/" prefix to filename.
258      */

```



```

259     newfile = strrchr(file, '/');
260     if (newfile != NULL) {
261         newfile = newfile + 1; /* Get rid of leading / */
262     } else {
263         newfile = file;
264     }
265
266     va_start(adx, fmt);
267     (void) vsnprintf(buf, sizeof (buf), fmt, adx);
268     va_end(adx);
269
270     /*
271     * To get this data, use the zfs-dprintf probe as so:
272     * dtrace -q -n 'zfs-dprintf \
273     * /stringof(arg0) == "dbuf.c"/ \
274     * {printf("%s: %s", stringof(arg1), stringof(arg3))}'
275     * arg0 = file name
276     * arg1 = function name
277     * arg2 = line number
278     * arg3 = message
279     */
280     DTRACE_PROBE4(zfs_dprintf,
281                 char *, newfile, char *, func, int, line, char *, buf);
282 }
283
284 static void
285 history_str_free(char *buf)
286 {
287     kmem_free(buf, HIS_MAX_RECORD_LEN);
288 }
289
290 static char *
291 history_str_get(zfs_cmd_t *zc)
292 {
293     char *buf;
294
295     if (zc->zfc_history == NULL)
296         return (NULL);
297
298     buf = kmem_alloc(HIS_MAX_RECORD_LEN, KM_SLEEP);
299     if (copyinstr((void *) (uintptr_t) zc->zfc_history,
300                 buf, HIS_MAX_RECORD_LEN, NULL) != 0) {
301         history_str_free(buf);
302         return (NULL);
303     }
304
305     buf[HIS_MAX_RECORD_LEN - 1] = '\0';
306
307     return (buf);
308 }
309
310 /*
311 * Check to see if the named dataset is currently defined as bootable
312 */
313 static boolean_t
314 zfs_is_bootfs(const char *name)
315 {
316     objset_t *os;
317
318     if (dmu_objset_hold(name, FTAG, &os) == 0) {
319         boolean_t ret;
320         ret = (dmu_objset_id(os) == spa_bootfs(dmu_objset_spa(os)));
321         dmu_objset_rele(os, FTAG);
322         return (ret);
323     }
324     return (B_FALSE);

```

```

325 }
326
327 /*
328 * zfs_earlier_version
329 *
330 * Return non-zero if the spa version is less than requested version.
331 */
332 static int
333 zfs_earlier_version(const char *name, int version)
334 {
335     spa_t *spa;
336
337     if (spa_open(name, &spa, FTAG) == 0) {
338         if (spa_version(spa) < version) {
339             spa_close(spa, FTAG);
340             return (1);
341         }
342         spa_close(spa, FTAG);
343     }
344     return (0);
345 }
346
347 /*
348 * zpl_earlier_version
349 *
350 * Return TRUE if the ZPL version is less than requested version.
351 */
352 static boolean_t
353 zpl_earlier_version(const char *name, int version)
354 {
355     objset_t *os;
356     boolean_t rc = B_TRUE;
357
358     if (dmu_objset_hold(name, FTAG, &os) == 0) {
359         uint64_t zplversion;
360
361         if (dmu_objset_type(os) != DMU_OST_ZFS) {
362             dmu_objset_rele(os, FTAG);
363             return (B_TRUE);
364         }
365         /* XXX reading from non-owned objset */
366         if (zfs_get_zplprop(os, ZFS_PROP_VERSION, &zplversion) == 0)
367             rc = zplversion < version;
368         dmu_objset_rele(os, FTAG);
369     }
370     return (rc);
371 }
372
373 static void
374 zfs_log_history(zfs_cmd_t *zc)
375 {
376     spa_t *spa;
377     char *buf;
378
379     if ((buf = history_str_get(zc)) == NULL)
380         return;
381
382     if (spa_open(zc->zfc_name, &spa, FTAG) == 0) {
383         if (spa_version(spa) >= SPA_VERSION_ZPOOL_HISTORY)
384             (void) spa_history_log(spa, buf);
385         spa_close(spa, FTAG);
386     }
387     history_str_free(buf);
388 }
389
390 /*

```

```

391 * Policy for top-level read operations (list pools). Requires no privileges,
392 * and can be used in the local zone, as there is no associated dataset.
393 */
394 /* ARGSUSED */
395 static int
396 zfs_secpolicy_none(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
397 {
398     return (0);
399 }
401 /*
402 * Policy for dataset read operations (list children, get statistics). Requires
403 * no privileges, but must be visible in the local zone.
404 */
405 /* ARGSUSED */
406 static int
407 zfs_secpolicy_read(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
408 {
409     if (INGLOBALZONE(curproc) ||
410         zone_dataset_visible(zc->zc_name, NULL))
411         return (0);
413     return (SET_ERROR(ENOENT));
414 }
416 static int
417 zfs_dozonecheck_impl(const char *dataset, uint64_t zoned, cred_t *cr)
418 {
419     int writable = 1;
421     /*
422      * The dataset must be visible by this zone -- check this first
423      * so they don't see EPERM on something they shouldn't know about.
424      */
425     if (!INGLOBALZONE(curproc) &&
426         !zone_dataset_visible(dataset, &writable))
427         return (SET_ERROR(ENOENT));
429     if (INGLOBALZONE(curproc)) {
430         /*
431          * If the fs is zoned, only root can access it from the
432          * global zone.
433          */
434         if (secpolicy_zfs(cr) && zoned)
435             return (SET_ERROR(EPERM));
436     } else {
437         /*
438          * If we are in a local zone, the 'zoned' property must be set.
439          */
440         if (!zoned)
441             return (SET_ERROR(EPERM));
443         /* must be writable by this zone */
444         if (!writable)
445             return (SET_ERROR(EPERM));
446     }
447     return (0);
448 }
450 static int
451 zfs_dozonecheck(const char *dataset, cred_t *cr)
452 {
453     uint64_t zoned;
455     if (dsl_prop_get_integer(dataset, "zoned", &zoned, NULL))
456         return (SET_ERROR(ENOENT));

```

```

458     return (zfs_dozonecheck_impl(dataset, zoned, cr));
459 }
461 static int
462 zfs_dozonecheck_ds(const char *dataset, dsl_dataset_t *ds, cred_t *cr)
463 {
464     uint64_t zoned;
466     if (dsl_prop_get_int_ds(ds, "zoned", &zoned))
467         return (SET_ERROR(ENOENT));
469     return (zfs_dozonecheck_impl(dataset, zoned, cr));
470 }
472 static int
473 zfs_secpolicy_write_perms_ds(const char *name, dsl_dataset_t *ds,
474     const char *perm, cred_t *cr)
475 {
476     int error;
478     error = zfs_dozonecheck_ds(name, ds, cr);
479     if (error == 0) {
480         error = secpolicy_zfs(cr);
481         if (error != 0)
482             error = dsl_deleg_access_impl(ds, perm, cr);
483     }
484     return (error);
485 }
487 static int
488 zfs_secpolicy_write_perms(const char *name, const char *perm, cred_t *cr)
489 {
490     int error;
491     dsl_dataset_t *ds;
492     dsl_pool_t *dp;
494     error = dsl_pool_hold(name, FTAG, &dp);
495     if (error != 0)
496         return (error);
498     error = dsl_dataset_hold(dp, name, FTAG, &ds);
499     if (error != 0) {
500         dsl_pool_rele(dp, FTAG);
501         return (error);
502     }
504     error = zfs_secpolicy_write_perms_ds(name, ds, perm, cr);
506     dsl_dataset_rele(ds, FTAG);
507     dsl_pool_rele(dp, FTAG);
508     return (error);
509 }
511 /*
512 * Policy for setting the security label property.
513 *
514 * Returns 0 for success, non-zero for access and other errors.
515 */
516 static int
517 zfs_set_slabel_policy(const char *name, char *strval, cred_t *cr)
518 {
519     char          ds_hexsl[MAXNAMELEN];
520     bslabel_t    ds_sl, new_sl;
521     boolean_t    new_default = FALSE;
522     uint64_t     zoned;

```

```

523     int             needed_priv = -1;
524     int             error;

526     /* First get the existing dataset label. */
527     error = dsl_prop_get(name, zfs_prop_to_name(ZFS_PROP_MLSLABEL),
528     1, sizeof(ds_hexsl), &ds_hexsl, NULL);
529     if (error != 0)
530         return (SET_ERROR(EPERM));

532     if (strcasecmp(strval, ZFS_MLSLABEL_DEFAULT) == 0)
533         new_default = TRUE;

535     /* The label must be translatable */
536     if (!new_default && (hexstr_to_label(strval, &new_sl) != 0))
537         return (SET_ERROR(EINVAL));

539     /*
540     * In a non-global zone, disallow attempts to set a label that
541     * doesn't match that of the zone; otherwise no other checks
542     * are needed.
543     */
544     if (!INGLOBALZONE(curproc)) {
545         if (new_default || !blequal(&new_sl, CR_SL(CRED())))
546             return (SET_ERROR(EPERM));
547         return (0);
548     }

550     /*
551     * For global-zone datasets (i.e., those whose zoned property is
552     * "off", verify that the specified new label is valid for the
553     * global zone.
554     */
555     if (dsl_prop_get_integer(name,
556     zfs_prop_to_name(ZFS_PROP_ZONED), &zoned, NULL))
557         return (SET_ERROR(EPERM));
558     if (!zoned) {
559         if (zfs_check_global_label(name, strval) != 0)
560             return (SET_ERROR(EPERM));
561     }

563     /*
564     * If the existing dataset label is nondefault, check if the
565     * dataset is mounted (label cannot be changed while mounted).
566     * Get the zfsvfs; if there isn't one, then the dataset isn't
567     * mounted (or isn't a dataset, doesn't exist, ...).
568     */
569     if (strcasecmp(ds_hexsl, ZFS_MLSLABEL_DEFAULT) != 0) {
570         objset_t *os;
571         static char *setsl_tag = "setsl_tag";

573         /*
574         * Try to own the dataset; abort if there is any error,
575         * (e.g., already mounted, in use, or other error).
576         */
577         error = dmu_objset_own(name, DMU_OST_ZFS, B_TRUE,
578         setsl_tag, &os);
579         if (error != 0)
580             return (SET_ERROR(EPERM));

582         dmu_objset_disown(os, setsl_tag);

584         if (new_default) {
585             needed_priv = PRIV_FILE_DOWNGRADE_SL;
586             goto out_check;
587         }

```

```

589         if (hexstr_to_label(strval, &new_sl) != 0)
590             return (SET_ERROR(EPERM));

592         if (blstrictdom(&ds_sl, &new_sl))
593             needed_priv = PRIV_FILE_DOWNGRADE_SL;
594         else if (blstrictdom(&new_sl, &ds_sl))
595             needed_priv = PRIV_FILE_UPGRADE_SL;
596     } else {
597         /* dataset currently has a default label */
598         if (!new_default)
599             needed_priv = PRIV_FILE_UPGRADE_SL;
600     }

602 out_check:
603     if (needed_priv != -1)
604         return (PRIV_POLICY(cr, needed_priv, B_FALSE, EPERM, NULL));
605     return (0);
606 }

608 static int
609 zfs_secpolicy_setprop(const char *dsname, zfs_prop_t prop, nvpair_t *propval,
610     cred_t *cr)
611 {
612     char *strval;

614     /*
615     * Check permissions for special properties.
616     */
617     switch (prop) {
618     case ZFS_PROP_ZONED:
619         /*
620         * Disallow setting of 'zoned' from within a local zone.
621         */
622         if (!INGLOBALZONE(curproc))
623             return (SET_ERROR(EPERM));
624         break;

626     case ZFS_PROP_QUOTA:
627         if (!INGLOBALZONE(curproc)) {
628             uint64_t zoned;
629             char setpoint[MAXNAMELEN];
630             /*
631             * Unprivileged users are allowed to modify the
632             * quota on things *under* (ie. contained by)
633             * the thing they own.
634             */
635             if (dsl_prop_get_integer(dsname, "zoned", &zoned,
636             setpoint))
637                 return (SET_ERROR(EPERM));
638             if (!zoned || strlen(dsname) <= strlen(setpoint))
639                 return (SET_ERROR(EPERM));
640         }
641         break;

643     case ZFS_PROP_MLSLABEL:
644         if (!is_system_labeled())
645             return (SET_ERROR(EPERM));

647         if (nvpair_value_string(propval, &strval) == 0) {
648             int err;

650             err = zfs_set_slabel_policy(dsname, strval, CRED());
651             if (err != 0)
652                 return (err);
653         }
654         break;

```

```

655     }
656     return (zfs_secpolicy_write_perms(dsname, zfs_prop_to_name(prop), cr));
657 }
658 }

660 /* ARGSUSED */
661 static int
662 zfs_secpolicy_set_fsacl(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
663 {
664     int error;

666     error = zfs_dozonecheck(zc->zc_name, cr);
667     if (error != 0)
668         return (error);

670     /*
671      * permission to set permissions will be evaluated later in
672      * dsl_deleg_can_allow()
673      */
674     return (0);
675 }

677 /* ARGSUSED */
678 static int
679 zfs_secpolicy_rollback(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
680 {
681     return (zfs_secpolicy_write_perms(zc->zc_name,
682         ZFS_DELEG_PERM_ROLLBACK, cr));
683 }

685 /* ARGSUSED */
686 static int
687 zfs_secpolicy_send(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
688 {
689     dsl_pool_t *dp;
690     dsl_dataset_t *ds;
691     char *cp;
692     int error;

694     /*
695      * Generate the current snapshot name from the given objsetid, then
696      * use that name for the secpolicy/zone checks.
697      */
698     cp = strchr(zc->zc_name, '@');
699     if (cp == NULL)
700         return (SET_ERROR(EINVAL));
701     error = dsl_pool_hold(zc->zc_name, FTAG, &dp);
702     if (error != 0)
703         return (error);

705     error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &ds);
706     if (error != 0) {
707         dsl_pool_rele(dp, FTAG);
708         return (error);
709     }

711     dsl_dataset_name(ds, zc->zc_name);

713     error = zfs_secpolicy_write_perms_ds(zc->zc_name, ds,
714         ZFS_DELEG_PERM_SEND, cr);
715     dsl_dataset_rele(ds, FTAG);
716     dsl_pool_rele(dp, FTAG);

718     return (error);
719 }

```

```

721 /* ARGSUSED */
722 static int
723 zfs_secpolicy_send_new(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
724 {
725     return (zfs_secpolicy_write_perms(zc->zc_name,
726         ZFS_DELEG_PERM_SEND, cr));
727 }

729 /* ARGSUSED */
730 static int
731 zfs_secpolicy_deleg_share(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
732 {
733     vnode_t *vp;
734     int error;

736     if ((error = lookupname(zc->zc_value, UIO_SYSSPACE,
737         NO_FOLLOW, NULL, &vp)) != 0)
738         return (error);

740     /* Now make sure mntpnt and dataset are ZFS */

742     if (vp->v_vfsp->vfs_fstype != zfsfstype ||
743         (strcmp((char *)refstr_value(vp->v_vfsp->vfs_resource),
744             zc->zc_name) != 0)) {
745         VN_RELE(vp);
746         return (SET_ERROR(EPERM));
747     }

749     VN_RELE(vp);
750     return (dsl_deleg_access(zc->zc_name,
751         ZFS_DELEG_PERM_SHARE, cr));
752 }

754 int
755 zfs_secpolicy_share(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
756 {
757     if (!INGLOBALZONE(curproc))
758         return (SET_ERROR(EPERM));

760     if (secpolicy_nfs(cr) == 0) {
761         return (0);
762     } else {
763         return (zfs_secpolicy_deleg_share(zc, innvl, cr));
764     }
765 }

767 int
768 zfs_secpolicy_smb_acl(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
769 {
770     if (!INGLOBALZONE(curproc))
771         return (SET_ERROR(EPERM));

773     if (secpolicy_smb(cr) == 0) {
774         return (0);
775     } else {
776         return (zfs_secpolicy_deleg_share(zc, innvl, cr));
777     }
778 }

780 static int
781 zfs_get_parent(const char *datasetname, char *parent, int parentsz)
782 {
783     char *cp;

785     /*
786      * Remove the @bla or /bla from the end of the name to get the parent.

```

```

787  */
788  (void) strncpy(parent, datasetname, parentsize);
789  cp = strchr(parent, '@');
790  if (cp != NULL) {
791      cp[0] = '\0';
792  } else {
793      cp = strchr(parent, '/');
794      if (cp == NULL)
795          return (SET_ERROR(ENOENT));
796      cp[0] = '\0';
797  }
799  return (0);
800 }

802 int
803 zfs_secpolicy_destroy_perms(const char *name, cred_t *cr)
804 {
805     int error;

807     if ((error = zfs_secpolicy_write_perms(name,
808         ZFS_DELEG_PERM_MOUNT, cr)) != 0)
809         return (error);

811     return (zfs_secpolicy_write_perms(name, ZFS_DELEG_PERM_DESTROY, cr));
812 }

814 /* ARGSUSED */
815 static int
816 zfs_secpolicy_destroy(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
817 {
818     return (zfs_secpolicy_destroy_perms(zc->zc_name, cr));
819 }

821 /*
822  * Destroying snapshots with delegated permissions requires
823  * descendant mount and destroy permissions.
824  */
825 /* ARGSUSED */
826 static int
827 zfs_secpolicy_destroy_snaps(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
828 {
829     nvlist_t *snaps;
830     nvpair_t *pair, *nextpair;
831     int error = 0;

833     if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
834         return (SET_ERROR(EINVAL));
835     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
836         pair = nextpair) {
837         dsl_pool_t *dp;
838         dsl_dataset_t *ds;

840         error = dsl_pool_hold(nvpair_name(pair), FTAG, &dp);
841         if (error != 0)
842             break;
843         nextpair = nvlist_next_nvpair(snaps, pair);
844         error = dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds);
845         if (error == 0)
846             dsl_dataset_rele(ds, FTAG);
847         dsl_pool_rele(dp, FTAG);

849         if (error == 0) {
850             error = zfs_secpolicy_destroy_perms(nvpair_name(pair),
851                 cr);
852         } else if (error == ENOENT) {

```

```

853  */
854  * Ignore any snapshots that don't exist (we consider
855  * them "already destroyed"). Remove the name from the
856  * nvl here in case the snapshot is created between
857  * now and when we try to destroy it (in which case
858  * we don't want to destroy it since we haven't
859  * checked for permission).
860  */
861  fnvlist_remove_nvpair(snaps, pair);
862  error = 0;
863  }
864  if (error != 0)
865      break;
866  }

868  return (error);
869 }

871 int
872 zfs_secpolicy_rename_perms(const char *from, const char *to, cred_t *cr)
873 {
874     char    parentname[MAXNAMELEN];
875     int     error;

877     if ((error = zfs_secpolicy_write_perms(from,
878         ZFS_DELEG_PERM_RENAME, cr)) != 0)
879         return (error);

881     if ((error = zfs_secpolicy_write_perms(from,
882         ZFS_DELEG_PERM_MOUNT, cr)) != 0)
883         return (error);

885     if ((error = zfs_get_parent(to, parentname,
886         sizeof (parentname))) != 0)
887         return (error);

889     if ((error = zfs_secpolicy_write_perms(parentname,
890         ZFS_DELEG_PERM_CREATE, cr)) != 0)
891         return (error);

893     if ((error = zfs_secpolicy_write_perms(parentname,
894         ZFS_DELEG_PERM_MOUNT, cr)) != 0)
895         return (error);

897     return (error);
898 }

900 /* ARGSUSED */
901 static int
902 zfs_secpolicy_rename(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
903 {
904     return (zfs_secpolicy_rename_perms(zc->zc_name, zc->zc_value, cr));
905 }

907 /* ARGSUSED */
908 static int
909 zfs_secpolicy_promote(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
910 {
911     dsl_pool_t *dp;
912     dsl_dataset_t *clone;
913     int error;

915     error = zfs_secpolicy_write_perms(zc->zc_name,
916         ZFS_DELEG_PERM_PROMOTE, cr);
917     if (error != 0)
918         return (error);

```

```

920     error = dsl_pool_hold(zc->zc_name, FTAG, &dp);
921     if (error != 0)
922         return (error);
924
925     error = dsl_dataset_hold(dp, zc->zc_name, FTAG, &clone);
926
927     if (error == 0) {
928         char parentname[MAXNAMELEN];
929         dsl_dataset_t *origin = NULL;
930         dsl_dir_t *dd;
931         dd = clone->ds_dir;
932
933         error = dsl_dataset_hold_obj(dd->dd_pool,
934             dd->dd_phys->dd_origin_obj, FTAG, &origin);
935         if (error != 0) {
936             dsl_dataset_rele(clone, FTAG);
937             dsl_pool_rele(dp, FTAG);
938             return (error);
939         }
940
941         error = zfs_secpolicy_write_perms_ds(zc->zc_name, clone,
942             ZFS_DELEG_PERM_MOUNT, cr);
943
944         dsl_dataset_name(origin, parentname);
945         if (error == 0) {
946             error = zfs_secpolicy_write_perms_ds(parentname, origin,
947                 ZFS_DELEG_PERM_PROMOTE, cr);
948         }
949         dsl_dataset_rele(clone, FTAG);
950         dsl_dataset_rele(origin, FTAG);
951     }
952     dsl_pool_rele(dp, FTAG);
953     return (error);
954 }
955
956 /* ARGSUSED */
957 static int
958 zfs_secpolicy_recv(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
959 {
960     int error;
961
962     if ((error = zfs_secpolicy_write_perms(zc->zc_name,
963         ZFS_DELEG_PERM_RECEIVE, cr)) != 0)
964         return (error);
965
966     if ((error = zfs_secpolicy_write_perms(zc->zc_name,
967         ZFS_DELEG_PERM_MOUNT, cr)) != 0)
968         return (error);
969
970     return (zfs_secpolicy_write_perms(zc->zc_name,
971         ZFS_DELEG_PERM_CREATE, cr));
972 }
973
974 int
975 zfs_secpolicy_snapshot_perms(const char *name, cred_t *cr)
976 {
977     return (zfs_secpolicy_write_perms(name,
978         ZFS_DELEG_PERM_SNAPSHOT, cr));
979 }
980
981 /*
982  * Check for permission to create each snapshot in the nvlist.
983  */
984 /* ARGSUSED */
985 static int

```

```

986 zfs_secpolicy_snapshot(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
987 {
988     nvlist_t *snaps;
989     int error = 0;
990     nvpair_t *pair;
991
992     if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
993         return (SET_ERROR(EINVAL));
994     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
995         pair = nvlist_next_nvpair(snaps, pair)) {
996         char *name = nvpair_name(pair);
997         char *atp = strchr(name, '@');
998
999         if (atp == NULL) {
1000             error = SET_ERROR(EINVAL);
1001             break;
1002         }
1003         *atp = '\0';
1004         error = zfs_secpolicy_snapshot_perms(name, cr);
1005         *atp = '@';
1006         if (error != 0)
1007             break;
1008     }
1009     return (error);
1010 }
1011
1012 /* ARGSUSED */
1013 static int
1014 zfs_secpolicy_log_history(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1015 {
1016     /*
1017      * Even root must have a proper TSD so that we know what pool
1018      * to log to.
1019      */
1020     if (tsd_get(zfs_allow_log_key) == NULL)
1021         return (SET_ERROR(EPERM));
1022     return (0);
1023 }
1024
1025 static int
1026 zfs_secpolicy_create_clone(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1027 {
1028     char parentname[MAXNAMELEN];
1029     int error;
1030     char *origin;
1031
1032     if ((error = zfs_get_parent(zc->zc_name, parentname,
1033         sizeof (parentname))) != 0)
1034         return (error);
1035
1036     if (nvlist_lookup_string(innvl, "origin", &origin) == 0 &&
1037         (error = zfs_secpolicy_write_perms(origin,
1038             ZFS_DELEG_PERM_CLONE, cr)) != 0)
1039         return (error);
1040
1041     if ((error = zfs_secpolicy_write_perms(parentname,
1042         ZFS_DELEG_PERM_CREATE, cr)) != 0)
1043         return (error);
1044
1045     return (zfs_secpolicy_write_perms(parentname,
1046         ZFS_DELEG_PERM_MOUNT, cr));
1047 }
1048
1049 /*
1050  * Policy for pool operations - create/destroy pools, add vdevs, etc. Requires
1051  * SYS_CONFIG privilege, which is not available in a local zone.

```

```

1051 */
1052 /* ARGSUSED */
1053 static int
1054 zfs_secpolicy_config(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1055 {
1056     if (secpolicy_sys_config(cr, B_FALSE) != 0)
1057         return (SET_ERROR(EPERM));
1058
1059     return (0);
1060 }
1061
1062 /*
1063  * Policy for object to name lookups.
1064  */
1065 /* ARGSUSED */
1066 static int
1067 zfs_secpolicy_diff(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1068 {
1069     int error;
1070
1071     if ((error = secpolicy_sys_config(cr, B_FALSE)) == 0)
1072         return (0);
1073
1074     error = zfs_secpolicy_write_perms(zc->zc_name, ZFS_DELEG_PERM_DIFF, cr);
1075     return (error);
1076 }
1077
1078 /*
1079  * Policy for fault injection. Requires all privileges.
1080  */
1081 /* ARGSUSED */
1082 static int
1083 zfs_secpolicy_inject(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1084 {
1085     return (secpolicy_zinject(cr));
1086 }
1087
1088 /* ARGSUSED */
1089 static int
1090 zfs_secpolicy_inherit_prop(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1091 {
1092     zfs_prop_t prop = zfs_name_to_prop(zc->zc_value);
1093
1094     if (prop == ZPROP_INVALID) {
1095         if (!zfs_prop_user(zc->zc_value))
1096             return (SET_ERROR(EINVAL));
1097         return (zfs_secpolicy_write_perms(zc->zc_name,
1098             ZFS_DELEG_PERM_USERPROP, cr));
1099     } else {
1100         return (zfs_secpolicy_setprop(zc->zc_name, prop,
1101             NULL, cr));
1102     }
1103 }
1104
1105 static int
1106 zfs_secpolicy_userspace_one(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1107 {
1108     int err = zfs_secpolicy_read(zc, innvl, cr);
1109     if (err)
1110         return (err);
1111
1112     if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
1113         return (SET_ERROR(EINVAL));
1114
1115     if (zc->zc_value[0] == 0) {
1116         /*

```

```

1117         * They are asking about a posix uid/gid. If it's
1118         * themself, allow it.
1119         */
1120         if (zc->zc_objset_type == ZFS_PROP_USERUSED ||
1121             zc->zc_objset_type == ZFS_PROP_USERQUOTA) {
1122             if (zc->zc_guid == crgetuid(cr))
1123                 return (0);
1124         } else {
1125             if (groupmember(zc->zc_guid, cr))
1126                 return (0);
1127         }
1128     }
1129
1130     return (zfs_secpolicy_write_perms(zc->zc_name,
1131         userquota_perms[zc->zc_objset_type], cr));
1132 }
1133
1134 static int
1135 zfs_secpolicy_userspace_many(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1136 {
1137     int err = zfs_secpolicy_read(zc, innvl, cr);
1138     if (err)
1139         return (err);
1140
1141     if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
1142         return (SET_ERROR(EINVAL));
1143
1144     return (zfs_secpolicy_write_perms(zc->zc_name,
1145         userquota_perms[zc->zc_objset_type], cr));
1146 }
1147
1148 /* ARGSUSED */
1149 static int
1150 zfs_secpolicy_userspace_upgrade(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1151 {
1152     return (zfs_secpolicy_setprop(zc->zc_name, ZFS_PROP_VERSION,
1153         NULL, cr));
1154 }
1155
1156 /* ARGSUSED */
1157 static int
1158 zfs_secpolicy_hold(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1159 {
1160     nvpair_t *pair;
1161     nvlist_t *holds;
1162     int error;
1163
1164     error = nvlist_lookup_nvlist(innvl, "holds", &holds);
1165     if (error != 0)
1166         return (SET_ERROR(EINVAL));
1167
1168     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
1169         pair = nvlist_next_nvpair(holds, pair)) {
1170         char fsname[MAXNAMELEN];
1171         error = dmu_fsname(nvpair_name(pair), fsname);
1172         if (error != 0)
1173             return (error);
1174         error = zfs_secpolicy_write_perms(fsname,
1175             ZFS_DELEG_PERM_HOLD, cr);
1176         if (error != 0)
1177             return (error);
1178     }
1179     return (0);
1180 }
1181
1182 /* ARGSUSED */

```

```

1183 static int
1184 zfs_secpolicy_release(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1185 {
1186     nvpair_t *pair;
1187     int error;
1188
1189     for (pair = nvlist_next_nvpair(innvl, NULL); pair != NULL;
1190          pair = nvlist_next_nvpair(innvl, pair)) {
1191         char fsname[MAXNAMELEN];
1192         error = dmu_fsname(nvpair_name(pair), fsname);
1193         if (error != 0)
1194             return (error);
1195         error = zfs_secpolicy_write_perms(fsname,
1196             ZFS_DELEG_PERM_RELEASE, cr);
1197         if (error != 0)
1198             return (error);
1199     }
1200     return (0);
1201 }
1202
1203 /*
1204  * Policy for allowing temporary snapshots to be taken or released
1205  */
1206 static int
1207 zfs_secpolicy_tmp_snapshot(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1208 {
1209     /*
1210      * A temporary snapshot is the same as a snapshot,
1211      * hold, destroy and release all rolled into one.
1212      * Delegated diff alone is sufficient that we allow this.
1213      */
1214     int error;
1215
1216     if ((error = zfs_secpolicy_write_perms(zc->zc_name,
1217         ZFS_DELEG_PERM_DIFF, cr)) == 0)
1218         return (0);
1219
1220     error = zfs_secpolicy_snapshot_perms(zc->zc_name, cr);
1221     if (error == 0)
1222         error = zfs_secpolicy_hold(zc, innvl, cr);
1223     if (error == 0)
1224         error = zfs_secpolicy_release(zc, innvl, cr);
1225     if (error == 0)
1226         error = zfs_secpolicy_destroy(zc, innvl, cr);
1227     return (error);
1228 }
1229
1230 /*
1231  * Returns the nvlist as specified by the user in the zfs_cmd_t.
1232  */
1233 static int
1234 get_nvlist(uint64_t nvl, uint64_t size, int iflag, nvlist_t **nvp)
1235 {
1236     char *packed;
1237     int error;
1238     nvlist_t *list = NULL;
1239
1240     /*
1241      * Read in and unpack the user-supplied nvlist.
1242      */
1243     if (size == 0)
1244         return (SET_ERROR(EINVAL));
1245
1246     packed = kmem_alloc(size, KM_SLEEP);
1247
1248     if ((error = ddi_copyin((void *) (uintptr_t) nvl, packed, size,

```

```

1249         iflag)) != 0) {
1250             kmem_free(packed, size);
1251             return (error);
1252         }
1253
1254     if ((error = nvlist_unpack(packed, size, &list, 0)) != 0) {
1255         kmem_free(packed, size);
1256         return (error);
1257     }
1258
1259     kmem_free(packed, size);
1260
1261     *nvp = list;
1262     return (0);
1263 }
1264
1265 /*
1266  * Reduce the size of this nvlist until it can be serialized in 'max' bytes.
1267  * Entries will be removed from the end of the nvlist, and one int32 entry
1268  * named "N_MORE_ERRORS" will be added indicating how many entries were
1269  * removed.
1270  */
1271 static int
1272 nvlist_smush(nvlist_t *errors, size_t max)
1273 {
1274     size_t size;
1275
1276     size = fnvlist_size(errors);
1277
1278     if (size > max) {
1279         nvpair_t *more_errors;
1280         int n = 0;
1281
1282         if (max < 1024)
1283             return (SET_ERROR(ENOMEM));
1284
1285         fnvlist_add_int32(errors, ZPROP_N_MORE_ERRORS, 0);
1286         more_errors = nvlist_prev_nvpair(errors, NULL);
1287
1288         do {
1289             nvpair_t *pair = nvlist_prev_nvpair(errors,
1290                 more_errors);
1291             fnvlist_remove_nvpair(errors, pair);
1292             n++;
1293             size = fnvlist_size(errors);
1294         } while (size > max);
1295
1296         fnvlist_remove_nvpair(errors, more_errors);
1297         fnvlist_add_int32(errors, ZPROP_N_MORE_ERRORS, n);
1298         ASSERT3U(fnvlist_size(errors), <=, max);
1299     }
1300
1301     return (0);
1302 }
1303
1304 static int
1305 put_nvlist(zfs_cmd_t *zc, nvlist_t *nvl)
1306 {
1307     char *packed = NULL;
1308     int error = 0;
1309     size_t size;
1310
1311     size = fnvlist_size(nvl);
1312
1313     if (size > zc->zc_nvlist_dst_size) {
1314         error = SET_ERROR(ENOMEM);

```



```

1315     } else {
1316         packed = fnvlist_pack(nvl, &size);
1317         if (ddi_copyout(packed, (void *) (uintptr_t)zc->zc_nvlist_dst,
1318             size, zc->zc_iflags) != 0)
1319             error = SET_ERROR(EFAULT);
1320         fnvlist_pack_free(packed, size);
1321     }
1322
1323     zc->zc_nvlist_dst_size = size;
1324     zc->zc_nvlist_dst_filled = B_TRUE;
1325     return (error);
1326 }
1327
1328 static int
1329 getzfsvfs(const char *dsname, zfsvfs_t **zfvpt)
1330 {
1331     objset_t *os;
1332     int error;
1333
1334     error = dmu_objset_hold(dsname, FTAG, &os);
1335     if (error != 0)
1336         return (error);
1337     if (dmu_objset_type(os) != DMU_OST_ZFS) {
1338         dmu_objset_rele(os, FTAG);
1339         return (SET_ERROR(EINVAL));
1340     }
1341
1342     mutex_enter(&os->os_user_ptr_lock);
1343     *zfvpt = dmu_objset_get_user(os);
1344     if (*zfvpt) {
1345         VFS_HOLD((*zfvpt)->z_vfs);
1346     } else {
1347         error = SET_ERROR(ESRCH);
1348     }
1349     mutex_exit(&os->os_user_ptr_lock);
1350     dmu_objset_rele(os, FTAG);
1351     return (error);
1352 }
1353
1354 /*
1355  * Find a zfsvfs_t for a mounted filesystem, or create our own, in which
1356  * case its z_vfs will be NULL, and it will be opened as the owner.
1357  * If 'writer' is set, the z_teardown_lock will be held for RW_WRITER,
1358  * which prevents all vnode ops from running.
1359  */
1360 static int
1361 zfsvfs_hold(const char *name, void *tag, zfsvfs_t **zfvpt, boolean_t writer)
1362 {
1363     int error = 0;
1364
1365     if (getzfsvfs(name, zfvpt) != 0)
1366         error = zfsvfs_create(name, zfvpt);
1367     if (error == 0) {
1368         rw_enter(&(*zfvpt)->z_teardown_lock, (writer) ? RW_WRITER :
1369             RW_READER, tag);
1370         if ((*zfvpt)->z_unmounted) {
1371             /*
1372              * XXX we could probably try again, since the unmounting
1373              * thread should be just about to disassociate the
1374              * objset from the zfsvfs.
1375              */
1376             rw_exit(&(*zfvpt)->z_teardown_lock, tag);
1377             return (SET_ERROR(EBUSY));
1378         }
1379     }
1380     return (error);

```

```

1381 }
1382
1383 static void
1384 zfsvfs_rele(zfsvfs_t *zfsvfs, void *tag)
1385 {
1386     rw_exit(&zfsvfs->z_teardown_lock, tag);
1387
1388     if (zfsvfs->z_vfs) {
1389         VFS_RELE(zfsvfs->z_vfs);
1390     } else {
1391         dmu_objset_disown(zfsvfs->z_os, zfsvfs);
1392         zfsvfs_free(zfsvfs);
1393     }
1394 }
1395
1396 static int
1397 zfs_ioc_pool_create(zfs_cmd_t *zc)
1398 {
1399     int error;
1400     nvlist_t *config, *props = NULL;
1401     nvlist_t *rootprops = NULL;
1402     nvlist_t *zplprops = NULL;
1403
1404     if (error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1405         zc->zc_iflags, &config))
1406         return (error);
1407
1408     if (zc->zc_nvlist_src_size != 0 && (error =
1409         get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
1410             zc->zc_iflags, &props))) {
1411         nvlist_free(config);
1412         return (error);
1413     }
1414
1415     if (props) {
1416         nvlist_t *nvl = NULL;
1417         uint64_t version = SPA_VERSION;
1418
1419         (void) nvlist_lookup_uint64(props,
1420             zpool_prop_to_name(ZPOOL_PROP_VERSION), &version);
1421         if (!SPA_VERSION_IS_SUPPORTED(version)) {
1422             error = SET_ERROR(EINVAL);
1423             goto pool_props_bad;
1424         }
1425         (void) nvlist_lookup_nvlist(props, ZPOOL_ROOTFS_PROPS, &nvl);
1426         if (nvl) {
1427             error = nvlist_dup(nvl, &rootprops, KM_SLEEP);
1428             if (error != 0) {
1429                 nvlist_free(config);
1430                 nvlist_free(props);
1431                 return (error);
1432             }
1433             (void) nvlist_remove_all(props, ZPOOL_ROOTFS_PROPS);
1434         }
1435         VERIFY(nvlist_alloc(&zplprops, NV_UNIQUE_NAME, KM_SLEEP) == 0);
1436         error = zfs_fill_zplprops_root(version, rootprops,
1437             zplprops, NULL);
1438         if (error != 0)
1439             goto pool_props_bad;
1440     }
1441
1442     error = spa_create(zc->zc_name, config, props, zplprops);
1443
1444     /*
1445      * Set the remaining root properties
1446      */

```

```

1447     if (!error && (error = zfs_set_prop_nvlist(zc->zc_name,
1448         ZPROP_SRC_LOCAL, rootprops, NULL)) != 0)
1449         (void) spa_destroy(zc->zc_name);

1451 pool_props_bad:
1452     nvlist_free(rootprops);
1453     nvlist_free(zplprops);
1454     nvlist_free(config);
1455     nvlist_free(props);

1457     return (error);
1458 }

1460 static int
1461 zfs_ioc_pool_destroy(zfs_cmd_t *zc)
1462 {
1463     int error;
1464     zfs_log_history(zc);
1465     error = spa_destroy(zc->zc_name);
1466     if (error == 0)
1467         zvol_remove_minors(zc->zc_name);
1468     return (error);
1469 }

1471 static int
1472 zfs_ioc_pool_import(zfs_cmd_t *zc)
1473 {
1474     nvlist_t *config, *props = NULL;
1475     uint64_t guid;
1476     int error;

1478     if ((error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1479         zc->zc_iflags, &config)) != 0)
1480         return (error);

1482     if (zc->zc_nvlist_src_size != 0 && (error =
1483         get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
1484         zc->zc_iflags, &props))) {
1485         nvlist_free(config);
1486         return (error);
1487     }

1489     if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID, &guid) != 0 ||
1490         guid != zc->zc_guid)
1491         error = SET_ERROR(EINVAL);
1492     else
1493         error = spa_import(zc->zc_name, config, props, zc->zc_cookie);

1495     if (zc->zc_nvlist_dst != 0) {
1496         int err;

1498         if ((err = put_nvlist(zc, config)) != 0)
1499             error = err;
1500     }

1502     nvlist_free(config);

1504     if (props)
1505         nvlist_free(props);

1507     return (error);
1508 }

1510 static int
1511 zfs_ioc_pool_export(zfs_cmd_t *zc)
1512 {

```

```

1513     int error;
1514     boolean_t force = (boolean_t)zc->zc_cookie;
1515     boolean_t hardforce = (boolean_t)zc->zc_guid;

1517     zfs_log_history(zc);
1518     error = spa_export(zc->zc_name, NULL, force, hardforce);
1519     if (error == 0)
1520         zvol_remove_minors(zc->zc_name);
1521     return (error);
1522 }

1524 static int
1525 zfs_ioc_pool_configs(zfs_cmd_t *zc)
1526 {
1527     nvlist_t *configs;
1528     int error;

1530     if ((configs = spa_all_configs(&zc->zc_cookie)) == NULL)
1531         return (SET_ERROR(EXIST));

1533     error = put_nvlist(zc, configs);

1535     nvlist_free(configs);

1537     return (error);
1538 }

1540 /*
1541  * inputs:
1542  *   zc_name           name of the pool
1543  *
1544  * outputs:
1545  *   zc_cookie         real errno
1546  *   zc_nvlist_dst     config nvlist
1547  *   zc_nvlist_dst_size size of config nvlist
1548  */
1549 static int
1550 zfs_ioc_pool_stats(zfs_cmd_t *zc)
1551 {
1552     nvlist_t *config;
1553     int error;
1554     int ret = 0;

1556     error = spa_get_stats(zc->zc_name, &config, zc->zc_value,
1557         sizeof(zc->zc_value));

1559     if (config != NULL) {
1560         ret = put_nvlist(zc, config);
1561         nvlist_free(config);
1562     }

1563     /*
1564      * The config may be present even if 'error' is non-zero.
1565      * In this case we return success, and preserve the real errno
1566      * in 'zc_cookie'.
1567      */
1568     zc->zc_cookie = error;
1569     } else {
1570         ret = error;
1571     }

1573     return (ret);
1574 }

1576 /*
1577  * Try to import the given pool, returning pool stats as appropriate so that
1578  * user land knows which devices are available and overall pool health.

```

```

1579 */
1580 static int
1581 zfs_ioc_pool_tryimport(zfs_cmd_t *zc)
1582 {
1583     nvlist_t *tryconfig, *config;
1584     int error;
1585
1586     if ((error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1587         zc->zc_iflags, &tryconfig)) != 0)
1588         return (error);
1589
1590     config = spa_tryimport(tryconfig);
1591
1592     nvlist_free(tryconfig);
1593
1594     if (config == NULL)
1595         return (SET_ERROR(EINVAL));
1596
1597     error = put_nvlist(zc, config);
1598     nvlist_free(config);
1599
1600     return (error);
1601 }
1602
1603 /*
1604  * inputs:
1605  * zc_name      name of the pool
1606  * zc_cookie    scan func (pool_scan_func_t)
1607  */
1608 static int
1609 zfs_ioc_pool_scan(zfs_cmd_t *zc)
1610 {
1611     spa_t *spa;
1612     int error;
1613
1614     if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1615         return (error);
1616
1617     if (zc->zc_cookie == POOL_SCAN_NONE)
1618         error = spa_scan_stop(spa);
1619     else
1620         error = spa_scan(spa, zc->zc_cookie);
1621
1622     spa_close(spa, FTAG);
1623
1624     return (error);
1625 }
1626
1627 static int
1628 zfs_ioc_pool_freeze(zfs_cmd_t *zc)
1629 {
1630     spa_t *spa;
1631     int error;
1632
1633     error = spa_open(zc->zc_name, &spa, FTAG);
1634     if (error == 0) {
1635         spa_freeze(spa);
1636         spa_close(spa, FTAG);
1637     }
1638     return (error);
1639 }
1640
1641 static int
1642 zfs_ioc_pool_upgrade(zfs_cmd_t *zc)
1643 {
1644     spa_t *spa;

```

```

1645     int error;
1646
1647     if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1648         return (error);
1649
1650     if (zc->zc_cookie < spa_version(spa) ||
1651         !SPA_VERSION_IS_SUPPORTED(zc->zc_cookie)) {
1652         spa_close(spa, FTAG);
1653         return (SET_ERROR(EINVAL));
1654     }
1655
1656     spa_upgrade(spa, zc->zc_cookie);
1657     spa_close(spa, FTAG);
1658
1659     return (error);
1660 }
1661
1662 static int
1663 zfs_ioc_pool_get_history(zfs_cmd_t *zc)
1664 {
1665     spa_t *spa;
1666     char *hist_buf;
1667     uint64_t size;
1668     int error;
1669
1670     if ((size = zc->zc_history_len) == 0)
1671         return (SET_ERROR(EINVAL));
1672
1673     if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1674         return (error);
1675
1676     if (spa_version(spa) < SPA_VERSION_ZPOOL_HISTORY) {
1677         spa_close(spa, FTAG);
1678         return (SET_ERROR(ENOTSUP));
1679     }
1680
1681     hist_buf = kmem_alloc(size, KM_SLEEP);
1682     if ((error = spa_history_get(spa, &zc->zc_history_offset,
1683         &zc->zc_history_len, hist_buf)) == 0) {
1684         error = ddi_copyout(hist_buf,
1685             (void *) (uintptr_t) zc->zc_history,
1686             zc->zc_history_len, zc->zc_iflags);
1687     }
1688
1689     spa_close(spa, FTAG);
1690     kmem_free(hist_buf, size);
1691     return (error);
1692 }
1693
1694 static int
1695 zfs_ioc_pool_reguid(zfs_cmd_t *zc)
1696 {
1697     spa_t *spa;
1698     int error;
1699
1700     error = spa_open(zc->zc_name, &spa, FTAG);
1701     if (error == 0) {
1702         error = spa_change_guid(spa);
1703         spa_close(spa, FTAG);
1704     }
1705     return (error);
1706 }
1707
1708 static int
1709 zfs_ioc_dsobj_to_dsname(zfs_cmd_t *zc)
1710 {

```

```

1711     return (dsl_dsobj_to_dsname(zc->zc_name, zc->zc_obj, zc->zc_value));
1712 }

1714 /*
1715  * inputs:
1716  * zc_name      name of filesystem
1717  * zc_obj       object to find
1718  *
1719  * outputs:
1720  * zc_value     name of object
1721  */
1722 static int
1723 zfs_ioc_obj_to_path(zfs_cmd_t *zc)
1724 {
1725     objset_t *os;
1726     int error;

1728     /* XXX reading from objset not owned */
1729     if ((error = dmu_objset_hold(zc->zc_name, FTAG, &os)) != 0)
1730         return (error);
1731     if (dmu_objset_type(os) != DMU_OST_ZFS) {
1732         dmu_objset_rele(os, FTAG);
1733         return (SET_ERROR(EINVAL));
1734     }
1735     error = zfs_obj_to_path(os, zc->zc_obj, zc->zc_value,
1736         sizeof (zc->zc_value));
1737     dmu_objset_rele(os, FTAG);

1739     return (error);
1740 }

1742 /*
1743  * inputs:
1744  * zc_name      name of filesystem
1745  * zc_obj       object to find
1746  *
1747  * outputs:
1748  * zc_stat      stats on object
1749  * zc_value     path to object
1750  */
1751 static int
1752 zfs_ioc_obj_to_stats(zfs_cmd_t *zc)
1753 {
1754     objset_t *os;
1755     int error;

1757     /* XXX reading from objset not owned */
1758     if ((error = dmu_objset_hold(zc->zc_name, FTAG, &os)) != 0)
1759         return (error);
1760     if (dmu_objset_type(os) != DMU_OST_ZFS) {
1761         dmu_objset_rele(os, FTAG);
1762         return (SET_ERROR(EINVAL));
1763     }
1764     error = zfs_obj_to_stats(os, zc->zc_obj, &zc->zc_stat, zc->zc_value,
1765         sizeof (zc->zc_value));
1766     dmu_objset_rele(os, FTAG);

1768     return (error);
1769 }

1771 static int
1772 zfs_ioc_vdev_add(zfs_cmd_t *zc)
1773 {
1774     spa_t *spa;
1775     int error;
1776     nvlist_t *config, **l2cache, **spares;

```

```

1777     uint_t nl2cache = 0, nspares = 0;

1779     error = spa_open(zc->zc_name, &spa, FTAG);
1780     if (error != 0)
1781         return (error);

1783     error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1784         zc->zc_iflags, &config);
1785     (void) nvlist_lookup_nvlist_array(config, ZPOOL_CONFIG_L2CACHE,
1786         &l2cache, &nl2cache);

1788     (void) nvlist_lookup_nvlist_array(config, ZPOOL_CONFIG_SPARES,
1789         &spares, &nspares);

1791     /*
1792      * A root pool with concatenated devices is not supported.
1793      * Thus, can not add a device to a root pool.
1794      *
1795      * Intent log device can not be added to a rootpool because
1796      * during mountroot, zil is replayed, a seperated log device
1797      * can not be accessed during the mountroot time.
1798      *
1799      * l2cache and spare devices are ok to be added to a rootpool.
1800      */
1801     if (spa_bootfs(spa) != 0 && nl2cache == 0 && nspares == 0) {
1802         nvlist_free(config);
1803         spa_close(spa, FTAG);
1804         return (SET_ERROR(EDOM));
1805     }

1807     if (error == 0) {
1808         error = spa_vdev_add(spa, config);
1809         nvlist_free(config);
1810     }
1811     spa_close(spa, FTAG);
1812     return (error);
1813 }

1815 /*
1816  * inputs:
1817  * zc_name      name of the pool
1818  * zc_nvlist_conf nvlist of devices to remove
1819  * zc_cookie    to stop the remove?
1820  */
1821 static int
1822 zfs_ioc_vdev_remove(zfs_cmd_t *zc)
1823 {
1824     spa_t *spa;
1825     int error;

1827     error = spa_open(zc->zc_name, &spa, FTAG);
1828     if (error != 0)
1829         return (error);
1830     error = spa_vdev_remove(spa, zc->zc_guid, B_FALSE);
1831     spa_close(spa, FTAG);
1832     return (error);
1833 }

1835 static int
1836 zfs_ioc_vdev_set_state(zfs_cmd_t *zc)
1837 {
1838     spa_t *spa;
1839     int error;
1840     vdev_state_t newstate = VDEV_STATE_UNKNOWN;

1842     if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)

```

```

1843         return (error);
1844     switch (zc->zc_cookie) {
1845     case VDEV_STATE_ONLINE:
1846         error = vdev_online(spa, zc->zc_guid, zc->zc_obj, &newstate);
1847         break;
1848
1849     case VDEV_STATE_OFFLINE:
1850         error = vdev_offline(spa, zc->zc_guid, zc->zc_obj);
1851         break;
1852
1853     case VDEV_STATE_FAULTED:
1854         if (zc->zc_obj != VDEV_AUX_ERR_EXCEEDED &&
1855             zc->zc_obj != VDEV_AUX_EXTERNAL)
1856             zc->zc_obj = VDEV_AUX_ERR_EXCEEDED;
1857
1858         error = vdev_fault(spa, zc->zc_guid, zc->zc_obj);
1859         break;
1860
1861     case VDEV_STATE_DEGRADED:
1862         if (zc->zc_obj != VDEV_AUX_ERR_EXCEEDED &&
1863             zc->zc_obj != VDEV_AUX_EXTERNAL)
1864             zc->zc_obj = VDEV_AUX_ERR_EXCEEDED;
1865
1866         error = vdev_degrade(spa, zc->zc_guid, zc->zc_obj);
1867         break;
1868
1869     default:
1870         error = SET_ERROR(EINVAL);
1871     }
1872     zc->zc_cookie = newstate;
1873     spa_close(spa, FTAG);
1874     return (error);
1875 }
1876
1877 static int
1878 zfs_ioc_vdev_attach(zfs_cmd_t *zc)
1879 {
1880     spa_t *spa;
1881     int replacing = zc->zc_cookie;
1882     nvlist_t *config;
1883     int error;
1884
1885     if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1886         return (error);
1887
1888     if ((error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1889         zc->zc_iflags, &config)) == 0) {
1890         error = spa_vdev_attach(spa, zc->zc_guid, config, replacing);
1891         nvlist_free(config);
1892     }
1893
1894     spa_close(spa, FTAG);
1895     return (error);
1896 }
1897
1898 static int
1899 zfs_ioc_vdev_detach(zfs_cmd_t *zc)
1900 {
1901     spa_t *spa;
1902     int error;
1903
1904     if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1905         return (error);
1906
1907     error = spa_vdev_detach(spa, zc->zc_guid, 0, B_FALSE);

```

```

1909     spa_close(spa, FTAG);
1910     return (error);
1911 }
1912
1913 static int
1914 zfs_ioc_vdev_split(zfs_cmd_t *zc)
1915 {
1916     spa_t *spa;
1917     nvlist_t *config, *props = NULL;
1918     int error;
1919     boolean_t exp = !(zc->zc_cookie & ZPOOL_EXPORT_AFTER_SPLIT);
1920
1921     if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1922         return (error);
1923
1924     if (error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1925         zc->zc_iflags, &config)) {
1926         spa_close(spa, FTAG);
1927         return (error);
1928     }
1929
1930     if (zc->zc_nvlist_src_size != 0 && (error =
1931         get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
1932         zc->zc_iflags, &props))) {
1933         spa_close(spa, FTAG);
1934         nvlist_free(config);
1935         return (error);
1936     }
1937
1938     error = spa_vdev_split_mirror(spa, zc->zc_string, config, props, exp);
1939
1940     spa_close(spa, FTAG);
1941
1942     nvlist_free(config);
1943     nvlist_free(props);
1944
1945     return (error);
1946 }
1947
1948 static int
1949 zfs_ioc_vdev_setpath(zfs_cmd_t *zc)
1950 {
1951     spa_t *spa;
1952     char *path = zc->zc_value;
1953     uint64_t guid = zc->zc_guid;
1954     int error;
1955
1956     error = spa_open(zc->zc_name, &spa, FTAG);
1957     if (error != 0)
1958         return (error);
1959
1960     error = spa_vdev_setpath(spa, guid, path);
1961     spa_close(spa, FTAG);
1962     return (error);
1963 }
1964
1965 static int
1966 zfs_ioc_vdev_setfru(zfs_cmd_t *zc)
1967 {
1968     spa_t *spa;
1969     char *fru = zc->zc_value;
1970     uint64_t guid = zc->zc_guid;
1971     int error;
1972
1973     error = spa_open(zc->zc_name, &spa, FTAG);
1974     if (error != 0)

```

```

1975         return (error);

1977     error = spa_vdev_setfru(spa, guid, fru);
1978     spa_close(spa, FTAG);
1979     return (error);
1980 }

1982 static int
1983 zfs_ioc_objset_stats_impl(zfs_cmd_t *zc, objset_t *os)
1984 {
1985     int error = 0;
1986     nvlist_t *nv;

1988     dmu_objset_fast_stat(os, &zc->zc_objset_stats);

1990     if (zc->zc_nvlist_dst != 0 &&
1991         (error = dsl_prop_get_all(os, &nv) == 0) {
1992         dmu_objset_stats(os, nv);
1993         /*
1994          * NB: zvol_get_stats() will read the objset contents,
1995          * which we aren't supposed to do with a
1996          * DS_MODE_USER hold, because it could be
1997          * inconsistent. So this is a bit of a workaround...
1998          * XXX reading with out owning
1999          */
2000         if (!zc->zc_objset_stats.dds_inconsistent &&
2001             dmu_objset_type(os) == DMU_OST_ZVOL) {
2002             error = zvol_get_stats(os, nv);
2003             if (error == EIO)
2004                 return (error);
2005             VERIFY0(error);
2006         }
2007         error = put_nvlist(zc, nv);
2008         nvlist_free(nv);
2009     }

2011     return (error);
2012 }

2014 /*
2015  * inputs:
2016  *  zc_name          name of filesystem
2017  *  zc_nvlist_dst_size  size of buffer for property nvlist
2018  *
2019  * outputs:
2020  *  zc_objset_stats  stats
2021  *  zc_nvlist_dst    property nvlist
2022  *  zc_nvlist_dst_size  size of property nvlist
2023  */
2024 static int
2025 zfs_ioc_objset_stats(zfs_cmd_t *zc)
2026 {
2027     objset_t *os;
2028     int error;

2030     error = dmu_objset_hold(zc->zc_name, FTAG, &os);
2031     if (error == 0) {
2032         error = zfs_ioc_objset_stats_impl(zc, os);
2033         dmu_objset_rele(os, FTAG);
2034     }

2036     return (error);
2037 }

2039 /*
2040  * inputs:

```

```

2041  *  zc_name          name of filesystem
2042  *  zc_nvlist_dst_size  size of buffer for property nvlist
2043  *
2044  * outputs:
2045  *  zc_nvlist_dst    received property nvlist
2046  *  zc_nvlist_dst_size  size of received property nvlist
2047  *
2048  * Gets received properties (distinct from local properties on or after
2049  * SPA_VERSION_RECVD_PROPS) for callers who want to differentiate received from
2050  * local property values.
2051  */
2052 static int
2053 zfs_ioc_objset_recvd_props(zfs_cmd_t *zc)
2054 {
2055     int error = 0;
2056     nvlist_t *nv;

2058     /*
2059     * Without this check, we would return local property values if the
2060     * caller has not already received properties on or after
2061     * SPA_VERSION_RECVD_PROPS.
2062     */
2063     if (!dsl_prop_get_hasrecvd(zc->zc_name))
2064         return (SET_ERROR(ENOTSUP));

2066     if (zc->zc_nvlist_dst != 0 &&
2067         (error = dsl_prop_get_received(zc->zc_name, &nv) == 0) {
2068         error = put_nvlist(zc, nv);
2069         nvlist_free(nv);
2070     }

2072     return (error);
2073 }

2075 static int
2076 nvl_add_zplprop(objset_t *os, nvlist_t *props, zfs_prop_t prop)
2077 {
2078     uint64_t value;
2079     int error;

2081     /*
2082     * zfs_get_zplprop() will either find a value or give us
2083     * the default value (if there is one).
2084     */
2085     if ((error = zfs_get_zplprop(os, prop, &value)) != 0)
2086         return (error);
2087     VERIFY(nvl_add_uint64(props, zfs_prop_to_name(prop), value) == 0);
2088     return (0);
2089 }

2091 /*
2092  * inputs:
2093  *  zc_name          name of filesystem
2094  *  zc_nvlist_dst_size  size of buffer for zpl property nvlist
2095  *
2096  * outputs:
2097  *  zc_nvlist_dst    zpl property nvlist
2098  *  zc_nvlist_dst_size  size of zpl property nvlist
2099  */
2100 static int
2101 zfs_ioc_objset_zplprops(zfs_cmd_t *zc)
2102 {
2103     objset_t *os;
2104     int err;

2106     /* XXX reading without owning */

```

```

2107     if (err = dmu_objset_hold(zc->zc_name, FTAG, &os))
2108         return (err);
2110     dmu_objset_fast_stat(os, &zc->zc_objset_stats);
2112     /*
2113      * NB: nvl_add_zplprop() will read the objset contents,
2114      * which we aren't supposed to do with a DS_MODE_USER
2115      * hold, because it could be inconsistent.
2116      */
2117     if (zc->zc_nvlist_dst != NULL &&
2118         !zc->zc_objset_stats.dds_inconsistent &&
2119         dmu_objset_type(os) == DMU_OST_ZFS) {
2120         nvlist_t *nv;
2122         VERIFY(nvlist_alloc(&nv, NV_UNIQUE_NAME, KM_SLEEP) == 0);
2123         if ((err = nvl_add_zplprop(os, nv, ZFS_PROP_VERSION)) == 0 &&
2124             (err = nvl_add_zplprop(os, nv, ZFS_PROP_NORMALIZE)) == 0 &&
2125             (err = nvl_add_zplprop(os, nv, ZFS_PROP_UTF8ONLY)) == 0 &&
2126             (err = nvl_add_zplprop(os, nv, ZFS_PROP_CASE)) == 0)
2127             err = put_nvlist(zc, nv);
2128         nvlist_free(nv);
2129     } else {
2130         err = SET_ERROR(ENOENT);
2131     }
2132     dmu_objset_rele(os, FTAG);
2133     return (err);
2134 }
2136 static boolean_t
2137 dataset_name_hidden(const char *name)
2138 {
2139     /*
2140      * Skip over datasets that are not visible in this zone,
2141      * internal datasets (which have a $ in their name), and
2142      * temporary datasets (which have a % in their name).
2143      */
2144     if (strchr(name, '$') != NULL)
2145         return (B_TRUE);
2146     if (strchr(name, '%') != NULL)
2147         return (B_TRUE);
2148     if (!INGLOBALZONE(curproc) && !zone_dataset_visible(name, NULL))
2149         return (B_TRUE);
2150     return (B_FALSE);
2151 }
2153 /*
2154  * inputs:
2155  * zc_name          name of filesystem
2156  * zc_cookie        zap cursor
2157  * zc_nvlist_dst_size  size of buffer for property nvlist
2158  *
2159  * outputs:
2160  * zc_name          name of next filesystem
2161  * zc_cookie        zap cursor
2162  * zc_objset_stats  stats
2163  * zc_nvlist_dst    property nvlist
2164  * zc_nvlist_dst_size  size of property nvlist
2165  */
2166 static int
2167 zfs_ioc_dataset_list_next(zfs_cmd_t *zc)
2168 {
2169     objset_t *os;
2170     int error;
2171     char *p;
2172     size_t orig_len = strlen(zc->zc_name);

```

```

2174 top:
2175     if (error = dmu_objset_hold(zc->zc_name, FTAG, &os)) {
2176         if (error == ENOENT)
2177             error = SET_ERROR(ESRCH);
2178         return (error);
2179     }
2181     p = strrchr(zc->zc_name, '/');
2182     if (p == NULL || p[1] != '\0')
2183         (void) strcat(zc->zc_name, "/", sizeof (zc->zc_name));
2184     p = zc->zc_name + strlen(zc->zc_name);
2186     do {
2187         error = dmu_dir_list_next(os,
2188             sizeof (zc->zc_name) - (p - zc->zc_name), p,
2189             NULL, &zc->zc_cookie);
2190         if (error == ENOENT)
2191             error = SET_ERROR(ESRCH);
2192     } while (error == 0 && dataset_name_hidden(zc->zc_name));
2193     dmu_objset_rele(os, FTAG);
2195     /*
2196      * If it's an internal dataset (ie. with a '$' in its name),
2197      * don't try to get stats for it, otherwise we'll return ENOENT.
2198      */
2199     if (error == 0 && strchr(zc->zc_name, '$') == NULL) {
2200         error = zfs_ioc_objset_stats(zc); /* fill in the stats */
2201         if (error == ENOENT) {
2202             /* We lost a race with destroy, get the next one. */
2203             zc->zc_name[orig_len] = '\0';
2204             goto top;
2205         }
2206     }
2207     return (error);
2208 }
2210 /*
2211  * inputs:
2212  * zc_name          name of filesystem
2213  * zc_cookie        zap cursor
2214  * zc_nvlist_dst_size  size of buffer for property nvlist
2215  *
2216  * outputs:
2217  * zc_name          name of next snapshot
2218  * zc_objset_stats  stats
2219  * zc_nvlist_dst    property nvlist
2220  * zc_nvlist_dst_size  size of property nvlist
2221  */
2222 static int
2223 zfs_ioc_snapshot_list_next(zfs_cmd_t *zc)
2224 {
2225     objset_t *os;
2226     int error;
2228     error = dmu_objset_hold(zc->zc_name, FTAG, &os);
2229     if (error != 0) {
2230         return (error == ENOENT ? ESRCH : error);
2231     }
2233     /*
2234      * A dataset name of maximum length cannot have any snapshots,
2235      * so exit immediately.
2236      */
2237     if (strlcat(zc->zc_name, "@", sizeof (zc->zc_name)) >= MAXNAMELEN) {
2238         dmu_objset_rele(os, FTAG);

```

```

2239         return (SET_ERROR(ESRCH));
2240     }

2242     error = dmu_snapshot_list_next(os,
2243         sizeof(zc->zc_name) - strlen(zc->zc_name),
2244         zc->zc_name + strlen(zc->zc_name), &zc->zc_obj, &zc->zc_cookie,
2245         NULL);

2247     if (error == 0) {
2248         dsl_dataset_t *ds;
2249         dsl_pool_t *dp = os->os_dsl_dataset->ds_dir->dd_pool;

2251         error = dsl_dataset_hold_obj(dp, zc->zc_obj, FTAG, &ds);
2252         if (error == 0) {
2253             objset_t *ossnap;

2255             error = dmu_objset_from_ds(ds, &ossnap);
2256             if (error == 0)
2257                 error = zfs_ioc_objset_stats_impl(zc, ossnap);
2258             dsl_dataset_rele(ds, FTAG);
2259         }
2260     } else if (error == ENOENT) {
2261         error = SET_ERROR(ESRCH);
2262     }

2264     dmu_objset_rele(os, FTAG);
2265     /* if we failed, undo the @ that we tacked on to zc_name */
2266     if (error != 0)
2267         *strchr(zc->zc_name, '@') = '\0';
2268     return (error);
2269 }

2271 static int
2272 zfs_prop_set_userquota(const char *dsname, nvpair_t *pair)
2273 {
2274     const char *propname = nvpair_name(pair);
2275     uint64_t *valary;
2276     unsigned int vallen;
2277     const char *domain;
2278     char *dash;
2279     zfs_userquota_prop_t type;
2280     uint64_t rid;
2281     uint64_t quota;
2282     zfsvfs_t *zfsvfs;
2283     int err;

2285     if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2286         nvlist_t *attrs;
2287         VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
2288         if (nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2289             &pair) != 0)
2290             return (SET_ERROR(EINVAL));
2291     }

2293     /*
2294      * A correctly constructed propname is encoded as
2295      * userquota@<rid>-<domain>.
2296      */
2297     if ((dash = strchr(propname, '-')) == NULL ||
2298         nvpair_value_uint64_array(pair, &valary, &vallen) != 0 ||
2299         vallen != 3)
2300         return (SET_ERROR(EINVAL));

2302     domain = dash + 1;
2303     type = valary[0];
2304     rid = valary[1];

```

```

2305         quota = valary[2];

2307         err = zfsvfs_hold(dsname, FTAG, &zfsvfs, B_FALSE);
2308         if (err == 0) {
2309             err = zfs_set_userquota(zfsvfs, type, domain, rid, quota);
2310             zfsvfs_rele(zfsvfs, FTAG);
2311         }

2313         return (err);
2314     }

2316     /*
2317      * If the named property is one that has a special function to set its value,
2318      * return 0 on success and a positive error code on failure; otherwise if it is
2319      * not one of the special properties handled by this function, return -1.
2320      *
2321      * XXX: It would be better for callers of the property interface if we handled
2322      * these special cases in dsl_prop.c (in the dsl layer).
2323      */
2324     static int
2325     zfs_prop_set_special(const char *dsname, zprop_source_t source,
2326         nvpair_t *pair)
2327     {
2328         const char *propname = nvpair_name(pair);
2329         zfs_prop_t prop = zfs_name_to_prop(propname);
2330         uint64_t intval;
2331         int err;

2333         if (prop == ZPROP_INVALID) {
2334             if (zfs_prop_userquota(propname))
2335                 return (zfs_prop_set_userquota(dsname, pair));
2336             return (-1);
2337         }

2339         if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2340             nvlist_t *attrs;
2341             VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
2342             VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2343                 &pair) == 0);
2344         }

2346         if (zfs_prop_get_type(prop) == PROP_TYPE_STRING)
2347             return (-1);

2349         VERIFY(0 == nvpair_value_uint64(pair, &intval));

2351         switch (prop) {
2352         case ZFS_PROP_QUOTA:
2353             err = dsl_dir_set_quota(dsname, source, intval);
2354             break;
2355         case ZFS_PROP_REFQUOTA:
2356             err = dsl_dataset_set_refquota(dsname, source, intval);
2357             break;
2358         case ZFS_PROP_RESERVATION:
2359             err = dsl_dir_set_reservation(dsname, source, intval);
2360             break;
2361         case ZFS_PROP_REFRESERVATION:
2362             err = dsl_dataset_set_refreservation(dsname, source, intval);
2363             break;
2364         case ZFS_PROP_VOLSIZE:
2365             err = zvol_set_volsize(dsname, intval);
2366             break;
2367         case ZFS_PROP_VERSION:
2368             {
2369                 zfsvfs_t *zfsvfs;

```



```

2371     if ((err = zfsvfs_hold(dsname, FTAG, &zfsvfs, B_TRUE)) != 0)
2372         break;
2374     err = zfs_set_version(zfsvfs, intval);
2375     zfsvfs_rele(zfsvfs, FTAG);
2377     if (err == 0 && intval >= ZPL_VERSION_USERSPACE) {
2378         zfs_cmd_t *zc;
2380         zc = kmem_zalloc(sizeof(zfs_cmd_t), KM_SLEEP);
2381         (void) strcpy(zc->zc_name, dsname);
2382         (void) zfs_ioc_userspace_upgrade(zc);
2383         kmem_free(zc, sizeof(zfs_cmd_t));
2384     }
2385     break;
2386 }
2387 case ZFS_PROP_COMPRESSION:
2388 {
2389     if (intval == ZIO_COMPRESS_LZ4) {
2390         zfeature_info_t *feature =
2391             &spa_feature_table[SPA_FEATURE_LZ4_COMPRESS];
2392         spa_t *spa;
2394         if ((err = spa_open(dsname, &spa, FTAG)) != 0)
2395             return (err);
2397         /*
2398          * Setting the LZ4 compression algorithm activates
2399          * the feature.
2400          */
2401         if (!spa_feature_is_active(spa, feature)) {
2402             if ((err = zfs_prop_activate_feature(spa,
2403                 feature)) != 0) {
2404                 spa_close(spa, FTAG);
2405                 return (err);
2406             }
2407         }
2409         spa_close(spa, FTAG);
2410     }
2411     /*
2412      * We still want the default set action to be performed in the
2413      * caller, we only performed zfeature settings here.
2414      */
2415     err = -1;
2416     break;
2417 }
2419 default:
2420     err = -1;
2421 }
2423 return (err);
2424 }
2426 /*
2427  * This function is best effort. If it fails to set any of the given properties,
2428  * it continues to set as many as it can and returns the last error
2429  * encountered. If the caller provides a non-NULL errlist, it will be filled in
2430  * with the list of names of all the properties that failed along with the
2431  * corresponding error numbers.
2432  *
2433  * If every property is set successfully, zero is returned and errlist is not
2434  * modified.
2435  */
2436 int

```

```

2437 zfs_set_prop_nvlist(const char *dsname, zprop_source_t source, nvlist_t *nvl,
2438     nvlist_t *errlist)
2439 {
2440     nvpair_t *pair;
2441     nvpair_t *propval;
2442     int rv = 0;
2443     uint64_t intval;
2444     char *strval;
2445     nvlist_t *genericnvl = fnvlist_alloc();
2446     nvlist_t *retrynnvl = fnvlist_alloc();
2448 retry:
2449     pair = NULL;
2450     while ((pair = nvlist_next_nvpair(nvl, pair)) != NULL) {
2451         const char *propname = nvpair_name(pair);
2452         zfs_prop_t prop = zfs_name_to_prop(propname);
2453         int err = 0;
2455         /* decode the property value */
2456         propval = pair;
2457         if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2458             nvlist_t *attrs;
2459             attrs = fnvpair_value_nvlist(pair);
2460             if (nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2461                 &propval) != 0)
2462                 err = SET_ERROR(EINVAL);
2463         }
2465         /* Validate value type */
2466         if (err == 0 && prop == ZPROP_INVALID) {
2467             if (zfs_prop_user(propname)) {
2468                 if (nvpair_type(propval) != DATA_TYPE_STRING)
2469                     err = SET_ERROR(EINVAL);
2470             } else if (zfs_prop_userquota(propname)) {
2471                 if (nvpair_type(propval) !=
2472                     DATA_TYPE_UINT64_ARRAY)
2473                     err = SET_ERROR(EINVAL);
2474             } else {
2475                 err = SET_ERROR(EINVAL);
2476             }
2477         } else if (err == 0) {
2478             if (nvpair_type(propval) == DATA_TYPE_STRING) {
2479                 if (zfs_prop_get_type(prop) != PROP_TYPE_STRING)
2480                     err = SET_ERROR(EINVAL);
2481             } else if (nvpair_type(propval) == DATA_TYPE_UINT64) {
2482                 const char *unused;
2484                 intval = fnvpair_value_uint64(propval);
2486                 switch (zfs_prop_get_type(prop)) {
2487                     case PROP_TYPE_NUMBER:
2488                         break;
2489                     case PROP_TYPE_STRING:
2490                         err = SET_ERROR(EINVAL);
2491                         break;
2492                     case PROP_TYPE_INDEX:
2493                         if (zfs_prop_index_to_string(prop,
2494                             intval, &unused) != 0)
2495                             err = SET_ERROR(EINVAL);
2496                         break;
2497                     default:
2498                         cmn_err(CE_PANIC,
2499                             "unknown property type");
2500                 }
2501             } else {
2502                 err = SET_ERROR(EINVAL);

```

```

2503     }
2504     }
2506     /* Validate permissions */
2507     if (err == 0)
2508         err = zfs_check_settable(dsname, pair, CRED());
2510     if (err == 0) {
2511         err = zfs_prop_set_special(dsname, source, pair);
2512         if (err == -1) {
2513             /*
2514              * For better performance we build up a list of
2515              * properties to set in a single transaction.
2516              */
2517             err = nvlist_add_nvpair(genericnvl, pair);
2518             } else if (err != 0 && nvl != retrynvl) {
2519                 /*
2520                  * This may be a spurious error caused by
2521                  * receiving quota and reservation out of order.
2522                  * Try again in a second pass.
2523                  */
2524                 err = nvlist_add_nvpair(retrynvl, pair);
2525             }
2526         }
2528     if (err != 0) {
2529         if (errlist != NULL)
2530             fnvlist_add_int32(errlist, propname, err);
2531         rv = err;
2532     }
2533 }
2535 if (nvl != retrynvl && !nvlist_empty(retrynvl)) {
2536     nvl = retrynvl;
2537     goto retry;
2538 }
2540 if (!nvlist_empty(genericnvl) &&
2541     dsl_props_set(dsname, source, genericnvl) != 0) {
2542     /*
2543      * If this fails, we still want to set as many properties as we
2544      * can, so try setting them individually.
2545      */
2546     pair = NULL;
2547     while ((pair = nvlist_next_nvpair(genericnvl, pair)) != NULL) {
2548         const char *propname = nvpair_name(pair);
2549         int err = 0;
2551         propval = pair;
2552         if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2553             nvlist_t *attrs;
2554             attrs = fnvpair_value_nvlist(pair);
2555             propval = fnvlist_lookup_nvpair(attrs,
2556                 ZPROP_VALUE);
2557         }
2559         if (nvpair_type(propval) == DATA_TYPE_STRING) {
2560             strval = fnvpair_value_string(propval);
2561             err = dsl_prop_set_string(dsname, propname,
2562                 source, strval);
2563         } else {
2564             intval = fnvpair_value_uint64(propval);
2565             err = dsl_prop_set_int(dsname, propname, source,
2566                 intval);
2567         }

```

```

2569         if (err != 0) {
2570             if (errlist != NULL) {
2571                 fnvlist_add_int32(errlist, propname,
2572                     err);
2573             }
2574             rv = err;
2575         }
2576     }
2577     }
2578     nvlist_free(genericnvl);
2579     nvlist_free(retrynvl);
2581     return (rv);
2582 }
2584 /*
2585  * Check that all the properties are valid user properties.
2586  */
2587 static int
2588 zfs_check_userprops(const char *fsname, nvlist_t *nvl)
2589 {
2590     nvpair_t *pair = NULL;
2591     int error = 0;
2593     while ((pair = nvlist_next_nvpair(nvl, pair)) != NULL) {
2594         const char *propname = nvpair_name(pair);
2595         char *valstr;
2597         if (!zfs_prop_user(propname) ||
2598             nvpair_type(pair) != DATA_TYPE_STRING)
2599             return (SET_ERROR(EINVAL));
2601         if (error = zfs_secpolicy_write_perms(fsname,
2602             ZFS_DELEG_PERM_USERPROP, CRED()))
2603             return (error);
2605         if (strlen(propname) >= ZAP_MAXNAMELEN)
2606             return (SET_ERROR(ENAMETOOLONG));
2608         VERIFY(nvpair_value_string(pair, &valstr) == 0);
2609         if (strlen(valstr) >= ZAP_MAXVALUELEN)
2610             return (E2BIG);
2611     }
2612     return (0);
2613 }
2615 static void
2616 props_skip(nvlist_t *props, nvlist_t *skipped, nvlist_t **newprops)
2617 {
2618     nvpair_t *pair;
2620     VERIFY(nvlist_alloc(newprops, NV_UNIQUE_NAME, KM_SLEEP) == 0);
2622     pair = NULL;
2623     while ((pair = nvlist_next_nvpair(props, pair)) != NULL) {
2624         if (nvlist_exists(skipped, nvpair_name(pair)))
2625             continue;
2627         VERIFY(nvlist_add_nvpair(*newprops, pair) == 0);
2628     }
2629 }
2631 static int
2632 clear_received_props(const char *dsname, nvlist_t *props,
2633     nvlist_t *skipped)
2634 {

```

```

2635     int err = 0;
2636     nvlist_t *cleared_props = NULL;
2637     props_skip(props, skipped, &cleared_props);
2638     if (!nvlist_empty(cleared_props)) {
2639         /*
2640          * Acts on local properties until the dataset has received
2641          * properties at least once on or after SPA_VERSION_RECVD_PROPS.
2642          */
2643         zprop_source_t flags = (ZPROP_SRC_NONE |
2644             (dsl_prop_get_hasrecvd(dsname) ? ZPROP_SRC_RECEIVED : 0));
2645         err = zfs_set_prop_nvlist(dsname, flags, cleared_props, NULL);
2646     }
2647     nvlist_free(cleared_props);
2648     return (err);
2649 }

2651 /*
2652 * inputs:
2653 * zc_name          name of filesystem
2654 * zc_value         name of property to set
2655 * zc_nvlist_src[_size] nvlist of properties to apply
2656 * zc_cookie       received properties flag
2657 *
2658 * outputs:
2659 * zc_nvlist_dst[_size] error for each unapplied received property
2660 */
2661 static int
2662 zfs_ioc_set_prop(zfs_cmd_t *zc)
2663 {
2664     nvlist_t *nvl;
2665     boolean_t received = zc->zc_cookie;
2666     zprop_source_t source = (received ? ZPROP_SRC_RECEIVED :
2667         ZPROP_SRC_LOCAL);
2668     nvlist_t *errors;
2669     int error;

2671     if ((error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
2672         zc->zc_iflags, &nvl)) != 0)
2673         return (error);

2675     if (received) {
2676         nvlist_t *origprops;

2678         if (dsl_prop_get_received(zc->zc_name, &origprops) == 0) {
2679             (void) clear_received_props(zc->zc_name,
2680                 origprops, nvl);
2681             nvlist_free(origprops);
2682         }

2684         error = dsl_prop_set_hasrecvd(zc->zc_name);
2685     }

2687     errors = fnvlist_alloc();
2688     if (error == 0)
2689         error = zfs_set_prop_nvlist(zc->zc_name, source, nvl, errors);

2691     if (zc->zc_nvlist_dst != NULL && errors != NULL) {
2692         (void) put_nvlist(zc, errors);
2693     }

2695     nvlist_free(errors);
2696     nvlist_free(nvl);
2697     return (error);
2698 }

2700 /*

```

```

2701 * inputs:
2702 * zc_name          name of filesystem
2703 * zc_value         name of property to inherit
2704 * zc_cookie       revert to received value if TRUE
2705 *
2706 * outputs:         none
2707 */
2708 static int
2709 zfs_ioc_inherit_prop(zfs_cmd_t *zc)
2710 {
2711     const char *propname = zc->zc_value;
2712     zfs_prop_t prop = zfs_name_to_prop(propname);
2713     boolean_t received = zc->zc_cookie;
2714     zprop_source_t source = (received
2715         ? ZPROP_SRC_NONE /* revert to received value, if any */
2716         : ZPROP_SRC_INHERITED); /* explicitly inherit */

2718     if (received) {
2719         nvlist_t *dummy;
2720         nvpair_t *pair;
2721         zprop_type_t type;
2722         int err;

2724         /*
2725          * zfs_prop_set_special() expects properties in the form of an
2726          * nvpair with type info.
2727          */
2728         if (prop == ZPROP_INVALID) {
2729             if (!zfs_prop_user(propname))
2730                 return (SET_ERROR(EINVAL));

2732             type = PROP_TYPE_STRING;
2733         } else if (prop == ZFS_PROP_VOLSIZE ||
2734             prop == ZFS_PROP_VERSION) {
2735             return (SET_ERROR(EINVAL));
2736         } else {
2737             type = zfs_prop_get_type(prop);
2738         }

2740         VERIFY(nvlist_alloc(&dummy, NV_UNIQUE_NAME, KM_SLEEP) == 0);

2742         switch (type) {
2743             case PROP_TYPE_STRING:
2744                 VERIFY(0 == nvlist_add_string(dummy, propname, ""));
2745                 break;
2746             case PROP_TYPE_NUMBER:
2747             case PROP_TYPE_INDEX:
2748                 VERIFY(0 == nvlist_add_uint64(dummy, propname, 0));
2749                 break;
2750             default:
2751                 nvlist_free(dummy);
2752                 return (SET_ERROR(EINVAL));
2753         }

2755         pair = nvlist_next_nvpair(dummy, NULL);
2756         err = zfs_prop_set_special(zc->zc_name, source, pair);
2757         nvlist_free(dummy);
2758         if (err != -1)
2759             return (err); /* special property already handled */
2760     } else {
2761         /*
2762          * Only check this in the non-received case. We want to allow
2763          * 'inherit -S' to revert non-inheritable properties like quota
2764          * and reservation to the received or default values even though
2765          * they are not considered inheritable.
2766          */

```

```

2767         if (prop != ZPROP_INVALID && !zfs_prop_inheritable(prop))
2768             return (SET_ERROR(EINVAL));
2769     }

2771     /* property name has been validated by zfs_secpolicy_inherit_prop() */
2772     return (dsl_prop_inherit(zc->zname, zc->zvalue, source));
2773 }

2775 static int
2776 zfs_ioc_pool_set_props(zfs_cmd_t *zc)
2777 {
2778     nvlist_t *props;
2779     spa_t *spa;
2780     int error;
2781     nvpair_t *pair;

2783     if (error = get_nvlist(zc->zname_src, zc->zname_src_size,
2784                          zc->ziflags, &props))
2785         return (error);

2787     /*
2788      * If the only property is the configfile, then just do a spa_lookup()
2789      * to handle the faulted case.
2790      */
2791     pair = nvlist_next_nvpair(props, NULL);
2792     if (pair != NULL && strcmp(nvpair_name(pair),
2793                             zpool_prop_to_name(ZPOOL_PROP_CACHEFILE)) == 0 &&
2794         nvlist_next_nvpair(props, pair) == NULL) {
2795         mutex_enter(&spa_namespace_lock);
2796         if ((spa = spa_lookup(zc->zname)) != NULL) {
2797             spa_configfile_set(spa, props, B_FALSE);
2798             spa_config_sync(spa, B_FALSE, B_TRUE);
2799         }
2800         mutex_exit(&spa_namespace_lock);
2801         if (spa != NULL) {
2802             nvlist_free(props);
2803             return (0);
2804         }
2805     }

2807     if ((error = spa_open(zc->zname, &spa, FTAG)) != 0) {
2808         nvlist_free(props);
2809         return (error);
2810     }

2812     error = spa_prop_set(spa, props);

2814     nvlist_free(props);
2815     spa_close(spa, FTAG);

2817     return (error);
2818 }

2820 static int
2821 zfs_ioc_pool_get_props(zfs_cmd_t *zc)
2822 {
2823     spa_t *spa;
2824     int error;
2825     nvlist_t *nvp = NULL;

2827     if ((error = spa_open(zc->zname, &spa, FTAG)) != 0) {
2828         /*
2829          * If the pool is faulted, there may be properties we can still
2830          * get (such as altroot and cachefile), so attempt to get them
2831          * anyway.
2832          */

```

```

2833         mutex_enter(&spa_namespace_lock);
2834         if ((spa = spa_lookup(zc->zname)) != NULL)
2835             error = spa_prop_get(spa, &nvp);
2836         mutex_exit(&spa_namespace_lock);
2837     } else {
2838         error = spa_prop_get(spa, &nvp);
2839         spa_close(spa, FTAG);
2840     }

2842     if (error == 0 && zc->zname_nvlist_dst != NULL)
2843         error = put_nvlist(zc, nvp);
2844     else
2845         error = SET_ERROR(EFAULT);

2847     nvlist_free(nvp);
2848     return (error);
2849 }

2851 /*
2852  * inputs:
2853  * zc_name           name of filesystem
2854  * zc_nvlist_src[_size] nvlist of delegated permissions
2855  * zc_perm_action    allow/unallow flag
2856  * outputs:
2857  *                   none
2858  */
2859 static int
2860 zfs_ioc_set_fsacl(zfs_cmd_t *zc)
2861 {
2862     int error;
2863     nvlist_t *fsaclnv = NULL;

2865     if ((error = get_nvlist(zc->zname_src, zc->zname_src_size,
2866                          zc->ziflags, &fsaclnv)) != 0)
2867         return (error);

2869     /*
2870      * Verify nvlist is constructed correctly
2871      */
2872     if ((error = zfs_deleg_verify_nvlist(fsaclnv)) != 0) {
2873         nvlist_free(fsaclnv);
2874         return (SET_ERROR(EINVAL));
2875     }

2877     /*
2878      * If we don't have PRIV_SYS_MOUNT, then validate
2879      * that user is allowed to hand out each permission in
2880      * the nvlist(s)
2881      */

2883     error = secpolicy_zfs(CRED());
2884     if (error != 0) {
2885         if (zc->zperm_action == B_FALSE) {
2886             error = dsl_deleg_can_allow(zc->zname,
2887                                       fsaclnv, CRED());
2888         } else {
2889             error = dsl_deleg_can_unallow(zc->zname,
2890                                         fsaclnv, CRED());
2891         }
2892     }

2894     if (error == 0)
2895         error = dsl_deleg_set(zc->zname, fsaclnv, zc->zperm_action);

2897     nvlist_free(fsaclnv);
2898     return (error);

```

```

2899 }

2901 /*
2902  * inputs:
2903  * zc_name          name of filesystem
2904  *
2905  * outputs:
2906  * zc_nvlist_src[_size] nvlist of delegated permissions
2907  */
2908 static int
2909 zfs_ioc_get_fsacl(zfs_cmd_t *zc)
2910 {
2911     nvlist_t *nvp;
2912     int error;

2914     if ((error = dsl_deleg_get(zc->zc_name, &nvp)) == 0) {
2915         error = put_nvlist(zc, nvp);
2916         nvlist_free(nvp);
2917     }

2919     return (error);
2920 }

2922 /*
2923  * Search the vfs list for a specified resource. Returns a pointer to it
2924  * or NULL if no suitable entry is found. The caller of this routine
2925  * is responsible for releasing the returned vfs pointer.
2926  */
2927 static vfs_t *
2928 zfs_get_vfs(const char *resource)
2929 {
2930     struct vfs *vfsp;
2931     struct vfs *vfs_found = NULL;

2933     vfs_list_read_lock();
2934     vfsp = rootvfs;
2935     do {
2936         if (strcmp(refstr_value(vfsp->vfs_resource), resource) == 0) {
2937             VFS_HOLD(vfsp);
2938             vfs_found = vfsp;
2939             break;
2940         }
2941         vfsp = vfsp->vfs_next;
2942     } while (vfsp != rootvfs);
2943     vfs_list_unlock();
2944     return (vfs_found);
2945 }

2947 /* ARGSUSED */
2948 static void
2949 zfs_create_cb(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx)
2950 {
2951     zfs_creat_t *zct = arg;

2953     zfs_create_fs(os, cr, zct->zct_zplprops, tx);
2954 }

2956 #define ZFS_PROP_UNDEFINED    ((uint64_t)-1)

2958 /*
2959  * inputs:
2960  * createprops      list of properties requested by creator
2961  * default_zplver   zpl version to use if unspecified in createprops
2962  * fuids_ok         fuids allowed in this version of the spa?
2963  * os               parent objset pointer (NULL if root fs)
2964  */

```

```

2965  * outputs:
2966  * zplprops         values for the zplprops we attach to the master node object
2967  * is_ci           true if requested file system will be purely case-insensitive
2968  *
2969  * Determine the settings for utf8only, normalization and
2970  * casesensitivity. Specific values may have been requested by the
2971  * creator and/or we can inherit values from the parent dataset. If
2972  * the file system is of too early a vintage, a creator can not
2973  * request settings for these properties, even if the requested
2974  * setting is the default value. We don't actually want to create dsl
2975  * properties for these, so remove them from the source nvlist after
2976  * processing.
2977  */
2978 static int
2979 zfs_fill_zplprops_impl(objset_t *os, uint64_t zplver,
2980     boolean_t fuids_ok, boolean_t sa_ok, nvlist_t *createprops,
2981     nvlist_t *zplprops, boolean_t *is_ci)
2982 {
2983     uint64_t sense = ZFS_PROP_UNDEFINED;
2984     uint64_t norm = ZFS_PROP_UNDEFINED;
2985     uint64_t u8 = ZFS_PROP_UNDEFINED;

2987     ASSERT(zplprops != NULL);

2989     /*
2990      * Pull out creator prop choices, if any.
2991      */
2992     if (createprops) {
2993         (void) nvlist_lookup_uint64(createprops,
2994             zfs_prop_to_name(ZFS_PROP_VERSION), &zplver);
2995         (void) nvlist_lookup_uint64(createprops,
2996             zfs_prop_to_name(ZFS_PROP_NORMALIZE), &norm);
2997         (void) nvlist_remove_all(createprops,
2998             zfs_prop_to_name(ZFS_PROP_NORMALIZE));
2999         (void) nvlist_lookup_uint64(createprops,
3000             zfs_prop_to_name(ZFS_PROP_UTF8ONLY), &u8);
3001         (void) nvlist_remove_all(createprops,
3002             zfs_prop_to_name(ZFS_PROP_UTF8ONLY));
3003         (void) nvlist_lookup_uint64(createprops,
3004             zfs_prop_to_name(ZFS_PROP_CASE), &sense);
3005         (void) nvlist_remove_all(createprops,
3006             zfs_prop_to_name(ZFS_PROP_CASE));
3007     }

3009     /*
3010      * If the zpl version requested is whacky or the file system
3011      * or pool is version is too "young" to support normalization
3012      * and the creator tried to set a value for one of the props,
3013      * error out.
3014      */
3015     if ((zplver < ZPL_VERSION_INITIAL || zplver > ZPL_VERSION) ||
3016         (zplver >= ZPL_VERSION_FUID && !fuids_ok) ||
3017         (zplver >= ZPL_VERSION_SA && !sa_ok) ||
3018         (zplver < ZPL_VERSION_NORMALIZATION &&
3019             (norm != ZFS_PROP_UNDEFINED || u8 != ZFS_PROP_UNDEFINED ||
3020             sense != ZFS_PROP_UNDEFINED)))
3021         return (SET_ERROR(ENOTSUP));

3023     /*
3024      * Put the version in the zplprops
3025      */
3026     VERIFY(nvlist_add_uint64(zplprops,
3027         zfs_prop_to_name(ZFS_PROP_VERSION), zplver) == 0);

3029     if (norm == ZFS_PROP_UNDEFINED)
3030         VERIFY(zfs_get_zplprop(os, ZFS_PROP_NORMALIZE, &norm) == 0);

```

```

3031     VERIFY(nvlist_add_uint64(zplprops,
3032         zfs_prop_to_name(ZFS_PROP_NORMALIZE), norm) == 0);
3033
3034     /*
3035      * If we're normalizing, names must always be valid UTF-8 strings.
3036      */
3037     if (norm)
3038         u8 = 1;
3039     if (u8 == ZFS_PROP_UNDEFINED)
3040         VERIFY(zfs_get_zplprop(os, ZFS_PROP_UTF8ONLY, &u8) == 0);
3041     VERIFY(nvlist_add_uint64(zplprops,
3042         zfs_prop_to_name(ZFS_PROP_UTF8ONLY), u8) == 0);
3043
3044     if (sense == ZFS_PROP_UNDEFINED)
3045         VERIFY(zfs_get_zplprop(os, ZFS_PROP_CASE, &sense) == 0);
3046     VERIFY(nvlist_add_uint64(zplprops,
3047         zfs_prop_to_name(ZFS_PROP_CASE), sense) == 0);
3048
3049     if (is_ci)
3050         *is_ci = (sense == ZFS_CASE_INSENSITIVE);
3051
3052     return (0);
3053 }
3054
3055 static int
3056 zfs_fill_zplprops(const char *dataset, nvlist_t *createprops,
3057     nvlist_t *zplprops, boolean_t *is_ci)
3058 {
3059     boolean_t fuids_ok, sa_ok;
3060     uint64_t zplver = ZPL_VERSION;
3061     objset_t *os = NULL;
3062     char parentname[MAXNAMELEN];
3063     char *cp;
3064     spa_t *spa;
3065     uint64_t spa_vers;
3066     int error;
3067
3068     (void) strncpy(parentname, dataset, sizeof (parentname));
3069     cp = strrchr(parentname, '/');
3070     ASSERT(cp != NULL);
3071     cp[0] = '\0';
3072
3073     if ((error = spa_open(dataset, &spa, FTAG)) != 0)
3074         return (error);
3075
3076     spa_vers = spa_version(spa);
3077     spa_close(spa, FTAG);
3078
3079     zplver = zfs_zpl_version_map(spa_vers);
3080     fuids_ok = (zplver >= ZPL_VERSION_FUID);
3081     sa_ok = (zplver >= ZPL_VERSION_SA);
3082
3083     /*
3084      * Open parent object set so we can inherit zplprop values.
3085      */
3086     if ((error = dmub_objset_hold(parentname, FTAG, &os)) != 0)
3087         return (error);
3088
3089     error = zfs_fill_zplprops_impl(os, zplver, fuids_ok, sa_ok, createprops,
3090         zplprops, is_ci);
3091     dmub_objset_rele(os, FTAG);
3092     return (error);
3093 }
3094
3095 static int
3096 zfs_fill_zplprops_root(uint64_t spa_vers, nvlist_t *createprops,

```

```

3097     nvlist_t *zplprops, boolean_t *is_ci)
3098 {
3099     boolean_t fuids_ok;
3100     boolean_t sa_ok;
3101     uint64_t zplver = ZPL_VERSION;
3102     int error;
3103
3104     zplver = zfs_zpl_version_map(spa_vers);
3105     fuids_ok = (zplver >= ZPL_VERSION_FUID);
3106     sa_ok = (zplver >= ZPL_VERSION_SA);
3107
3108     error = zfs_fill_zplprops_impl(NULL, zplver, fuids_ok, sa_ok,
3109         createprops, zplprops, is_ci);
3110     return (error);
3111 }
3112
3113 /*
3114 * innvl: {
3115 *     "type" -> dmub_objset_type_t (int32)
3116 *     (optional) "props" -> { prop -> value }
3117 * }
3118 *
3119 * outnvl: propname -> error code (int32)
3120 */
3121 static int
3122 zfs_ioc_create(const char *fsname, nvlist_t *innvl, nvlist_t *outnvl)
3123 {
3124     int error = 0;
3125     zfs_creat_t zct = { 0 };
3126     nvlist_t *nvprops = NULL;
3127     void (*cbfunc)(objset_t *os, void *arg, cred_t *cr, dmub_tx_t *tx);
3128     int32_t type32;
3129     dmub_objset_type_t type;
3130     boolean_t is_insensitive = B_FALSE;
3131
3132     if (nvlist_lookup_int32(innvl, "type", &type32) != 0)
3133         return (SET_ERROR(EINVAL));
3134     type = type32;
3135     (void) nvlist_lookup_nvlist(innvl, "props", &nvprops);
3136
3137     switch (type) {
3138     case DMU_OST_ZFS:
3139         cbfunc = zfs_create_cb;
3140         break;
3141
3142     case DMU_OST_ZVOL:
3143         cbfunc = zvol_create_cb;
3144         break;
3145
3146     default:
3147         cbfunc = NULL;
3148         break;
3149     }
3150     if (strchr(fsname, '@') ||
3151         strchr(fsname, '%'))
3152         return (SET_ERROR(EINVAL));
3153
3154     zct.zct_props = nvprops;
3155
3156     if (cbfunc == NULL)
3157         return (SET_ERROR(EINVAL));
3158
3159     if (type == DMU_OST_ZVOL) {
3160         uint64_t volsize, volblocksize;
3161
3162         if (nvprops == NULL)

```

```

3163         return (SET_ERROR(EINVAL));
3164     if (nvlist_lookup_uint64(nvprops,
3165         zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE), &volsize) != 0)
3166         return (SET_ERROR(EINVAL));

3168     if ((error = nvlist_lookup_uint64(nvprops,
3169         zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
3170         &volblocksize)) != 0 && error != ENOENT)
3171         return (SET_ERROR(EINVAL));

3173     if (error != 0)
3174         volblocksize = zfs_prop_default_numeric(
3175             ZFS_PROP_VOLBLOCKSIZE);

3177     if ((error = zvol_check_volblocksize(
3178         volblocksize)) != 0 ||
3179         (error = zvol_check_volsize(volsize,
3180         volblocksize)) != 0)
3181         return (error);
3182 } else if (type == DMU_OST_ZFS) {
3183     int error;

3185     /*
3186     * We have to have normalization and
3187     * case-folding flags correct when we do the
3188     * file system creation, so go figure them out
3189     * now.
3190     */
3191     VERIFY(nvlist_alloc(&zct.zct_zplprops,
3192         NV_UNIQUE_NAME, KM_SLEEP) == 0);
3193     error = zfs_fill_zplprops(fsname, nvprops,
3194         zct.zct_zplprops, &is_insensitive);
3195     if (error != 0) {
3196         nvlist_free(zct.zct_zplprops);
3197         return (error);
3198     }
3199 }

3201 error = dmu_objset_create(fsname, type,
3202     is_insensitive ? DS_FLAG_CI_DATASET : 0, cbfunc, &zct);
3203 nvlist_free(zct.zct_zplprops);

3205 /*
3206 * It would be nice to do this atomically.
3207 */
3208 if (error == 0) {
3209     error = zfs_set_prop_nvlist(fsname, ZPROP_SRC_LOCAL,
3210         nvprops, outnvl);
3211     if (error != 0)
3212         (void) dsl_destroy_head(fsname);
3213 }
3214 return (error);
3215 }

3217 /*
3218 * innvl: {
3219 *     "origin" -> name of origin snapshot
3220 *     (optional) "props" -> { prop -> value }
3221 * }
3222 *
3223 * outnvl: propname -> error code (int32)
3224 */
3225 static int
3226 zfs_ioc_clone(const char *fsname, nvlist_t *innvl, nvlist_t *outnvl)
3227 {
3228     int error = 0;

```

```

3229     nvlist_t *nvprops = NULL;
3230     char *origin_name;

3232     if (nvlist_lookup_string(innvl, "origin", &origin_name) != 0)
3233         return (SET_ERROR(EINVAL));
3234     (void) nvlist_lookup_nvlist(innvl, "props", &nvprops);

3236     if (strchr(fsname, '@') ||
3237         strchr(fsname, '%'))
3238         return (SET_ERROR(EINVAL));

3240     if (dataset_namecheck(origin_name, NULL, NULL) != 0)
3241         return (SET_ERROR(EINVAL));
3242     error = dmu_objset_clone(fsname, origin_name);
3243     if (error != 0)
3244         return (error);

3246     /*
3247     * It would be nice to do this atomically.
3248     */
3249     if (error == 0) {
3250         error = zfs_set_prop_nvlist(fsname, ZPROP_SRC_LOCAL,
3251             nvprops, outnvl);
3252         if (error != 0)
3253             (void) dsl_destroy_head(fsname);
3254     }
3255     return (error);
3256 }

3258 /*
3259 * innvl: {
3260 *     "snaps" -> { snapshot1, snapshot2 }
3261 *     (optional) "props" -> { prop -> value (string) }
3262 * }
3263 *
3264 * outnvl: snapshot -> error code (int32)
3265 */
3266 static int
3267 zfs_ioc_snapshot(const char *poolname, nvlist_t *innvl, nvlist_t *outnvl)
3268 {
3269     nvlist_t *snaps;
3270     nvlist_t *props = NULL;
3271     int error, poollen;
3272     nvpair_t *pair;

3274     (void) nvlist_lookup_nvlist(innvl, "props", &props);
3275     if ((error = zfs_check_userprops(poolname, props)) != 0)
3276         return (error);

3278     if (!nvlist_empty(props) &&
3279         zfs_earlier_version(poolname, SPA_VERSION_SNAP_PROPS))
3280         return (SET_ERROR(ENOTSUP));

3282     if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
3283         return (SET_ERROR(EINVAL));
3284     poollen = strlen(poolname);
3285     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
3286         pair = nvlist_next_nvpair(snaps, pair)) {
3287         const char *name = nvpair_name(pair);
3288         const char *cp = strchr(name, '@');

3290         /*
3291         * The snap name must contain an @, and the part after it must
3292         * contain only valid characters.
3293         */
3294         if (cp == NULL || snapshot_namecheck(cp + 1, NULL, NULL) != 0)

```

```

3295         return (SET_ERROR(EINVAL));
3297     /*
3298     * The snap must be in the specified pool.
3299     */
3300     if (strcmp(name, poolname, poolen) != 0 ||
3301         (name[poolen] != '/' && name[poolen] != '@'))
3302         return (SET_ERROR(EXDEV));
3304     /* This must be the only snap of this fs. */
3305     for (nvpair_t *pair2 = nvlist_next_nvpair(snaps, pair);
3306          pair2 != NULL; pair2 = nvlist_next_nvpair(snaps, pair2)) {
3307         if (strcmp(name, nvpair_name(pair2), cp - name + 1)
3308             == 0) {
3309             return (SET_ERROR(EXDEV));
3310         }
3311     }
3312 }
3314     error = dsl_dataset_snapshot(snaps, props, outnvl);
3315     return (error);
3316 }
3318 /*
3319  * innvl: "message" -> string
3320  */
3321 /* ARGSUSED */
3322 static int
3323 zfs_ioc_log_history(const char *unused, nvlist_t *innvl, nvlist_t *outnvl)
3324 {
3325     char *message;
3326     spa_t *spa;
3327     int error;
3328     char *poolname;
3330     /*
3331     * The poolname in the ioctl is not set, we get it from the TSD,
3332     * which was set at the end of the last successful ioctl that allows
3333     * logging. The secpolicy func already checked that it is set.
3334     * Only one log ioctl is allowed after each successful ioctl, so
3335     * we clear the TSD here.
3336     */
3337     poolname = tsd_get(zfs_allow_log_key);
3338     (void) tsd_set(zfs_allow_log_key, NULL);
3339     error = spa_open(poolname, &spa, FTAG);
3340     strfree(poolname);
3341     if (error != 0)
3342         return (error);
3344     if (nvlist_lookup_string(innvl, "message", &message) != 0) {
3345         spa_close(spa, FTAG);
3346         return (SET_ERROR(EINVAL));
3347     }
3349     if (spa_version(spa) < SPA_VERSION_ZPOOL_HISTORY) {
3350         spa_close(spa, FTAG);
3351         return (SET_ERROR(ENOTSUP));
3352     }
3354     error = spa_history_log(spa, message);
3355     spa_close(spa, FTAG);
3356     return (error);
3357 }
3359 /*
3360  * The dp_config_rwlock must not be held when calling this, because the

```

```

3361  * unmount may need to write out data.
3362  */
3363  * This function is best-effort. Callers must deal gracefully if it
3364  * remains mounted (or is remounted after this call).
3365  */
3366 void
3367 zfs_unmount_snap(const char *snapname)
3368 {
3369     vfs_t *vfsp;
3370     zfsvfs_t *zfsvfs;
3372     if ( strchr(snapname, '@') == NULL)
3373         return;
3375     vfsp = zfs_get_vfsp(snapname);
3376     if (vfsp == NULL)
3377         return;
3379     zfsvfs = vfsp->vfs_data;
3380     ASSERT(!dsl_pool_config_held(dmu_objset_pool(zfsvfs->z_os)));
3382     if (vn_vfswlock(vfsp->vfs_vnodecovered) != 0) {
3383         VFS_RELE(vfsp);
3384         return;
3385     }
3386     VFS_RELE(vfsp);
3388     /*
3389     * Always force the unmount for snapshots.
3390     */
3391     (void) dounmount(vfsp, MS_FORCE, kcred);
3392 }
3394 /* ARGSUSED */
3395 static int
3396 zfs_unmount_snap_cb(const char *snapname, void *arg)
3397 {
3398     zfs_unmount_snap(snapname);
3399     return (0);
3400 }
3402 /*
3403  * When a clone is destroyed, its origin may also need to be destroyed,
3404  * in which case it must be unmounted. This routine will do that unmount
3405  * if necessary.
3406  */
3407 void
3408 zfs_destroy_unmount_origin(const char *fsname)
3409 {
3410     int error;
3411     objset_t *os;
3412     dsl_dataset_t *ds;
3414     error = dmu_objset_hold(fsname, FTAG, &os);
3415     if (error != 0)
3416         return;
3417     ds = dmu_objset_ds(os);
3418     if (dsl_dir_is_clone(ds->ds_dir) && DS_IS_DEFER_DESTROY(ds->ds_prev)) {
3419         char originname[MAXNAMELEN];
3420         dsl_dataset_name(ds->ds_prev, originname);
3421         dmu_objset_rele(os, FTAG);
3422         zfs_unmount_snap(originname);
3423     } else {
3424         dmu_objset_rele(os, FTAG);
3425     }
3426 }

```



```

3428 /*
3429 * innvl: {
3430 *   "snaps" -> { snapshot1, snapshot2 }
3431 *   (optional boolean) "defer"
3432 * }
3433 *
3434 * outnvl: snapshot -> error code (int32)
3435 */
3436 static int
3437 zfs_ioc_destroy_snaps(const char *poolname, nvlist_t *innvl, nvlist_t *outnvl)
3438 {
3439     int poollen;
3440     nvlist_t *snaps;
3441     nvpair_t *pair;
3442     boolean_t defer;
3443
3444     if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
3445         return (SET_ERROR(EINVAL));
3446     defer = nvlist_exists(innvl, "defer");
3447
3448     poollen = strlen(poolname);
3449     for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
3450          pair = nvlist_next_nvpair(snaps, pair)) {
3451         const char *name = nvpair_name(pair);
3452
3453         /*
3454          * The snap must be in the specified pool.
3455          */
3456         if (strncmp(name, poolname, poollen) != 0 ||
3457             (name[poollen] != '/' && name[poollen] != '@'))
3458             return (SET_ERROR(EXDEV));
3459
3460         zfs_unmount_snap(name);
3461     }
3462
3463     return (dsl_destroy_snapshots_nvl(snaps, defer, outnvl));
3464 }
3465
3466 /*
3467 * inputs:
3468 *   zc_name          name of dataset to destroy
3469 *   zc_objset_type   type of objset
3470 *   zc_defer_destroy mark for deferred destroy
3471 *
3472 * outputs:          none
3473 */
3474 static int
3475 zfs_ioc_destroy(zfs_cmd_t *zc)
3476 {
3477     int err;
3478     if (strchr(zc->zc_name, '@') && zc->zc_objset_type == DMU_OST_ZFS)
3479         zfs_unmount_snap(zc->zc_name);
3480
3481     if (strchr(zc->zc_name, '@'))
3482         err = dsl_destroy_snapshot(zc->zc_name, zc->zc_defer_destroy);
3483     else
3484         err = dsl_destroy_head(zc->zc_name);
3485     if (zc->zc_objset_type == DMU_OST_ZVOL && err == 0)
3486         (void) zvol_remove_minior(zc->zc_name);
3487     return (err);
3488 }
3489
3490 /*
3491 * inputs:

```

```

3493 *   zc_name          name of dataset to rollback (to most recent snapshot)
3494 *
3495 * outputs:          none
3496 */
3497 static int
3498 zfs_ioc_rollback(zfs_cmd_t *zc)
3499 {
3500     zfsvfs_t *zfsvfs;
3501     int error;
3502
3503     if (getzfsvfs(zc->zc_name, &zfsvfs) == 0) {
3504         error = zfs_suspend_fs(zfsvfs);
3505         if (error == 0) {
3506             int resume_err;
3507
3508             error = dsl_dataset_rollback(zc->zc_name);
3509             resume_err = zfs_resume_fs(zfsvfs, zc->zc_name);
3510             error = error ? error : resume_err;
3511         }
3512         VFS_RELE(zfsvfs->z_vfs);
3513     } else {
3514         error = dsl_dataset_rollback(zc->zc_name);
3515     }
3516     return (error);
3517 }
3518
3519 static int
3520 recursive_unmount(const char *fsname, void *arg)
3521 {
3522     const char *snapname = arg;
3523     char fullname[MAXNAMELEN];
3524
3525     (void) snprintf(fullname, sizeof (fullname), "%s%s", fsname, snapname);
3526     zfs_unmount_snap(fullname);
3527     return (0);
3528 }
3529
3530 /*
3531 * inputs:
3532 *   zc_name          old name of dataset
3533 *   zc_value         new name of dataset
3534 *   zc_cookie        recursive flag (only valid for snapshots)
3535 *
3536 * outputs:          none
3537 */
3538 static int
3539 zfs_ioc_rename(zfs_cmd_t *zc)
3540 {
3541     boolean_t recursive = zc->zc_cookie & 1;
3542     char *at;
3543
3544     zc->zc_value[sizeof (zc->zc_value) - 1] = '\0';
3545     if (dataset_namecheck(zc->zc_value, NULL, NULL) != 0 ||
3546         strchr(zc->zc_value, '%'))
3547         return (SET_ERROR(EINVAL));
3548
3549     at = strchr(zc->zc_name, '@');
3550     if (at != NULL) {
3551         /* snaps must be in same fs */
3552         if (strncmp(zc->zc_name, zc->zc_value, at - zc->zc_name + 1))
3553             return (SET_ERROR(EXDEV));
3554         *at = '\0';
3555         if (zc->zc_objset_type == DMU_OST_ZFS) {
3556             int error = dm_uobjset_find(zc->zc_name,
3557                 recursive_unmount, at + 1,
3558                 recursive ? DS_FIND_CHILDREN : 0);

```

```

3559         if (error != 0)
3560             return (error);
3561     }
3562     return (dsl_dataset_rename_snapshot(zc->zc_name,
3563         at + 1, strchr(zc->zc_value, '@') + 1, recursive));
3564 } else {
3565     if (zc->zc_objset_type == DMU_OST_ZVOL)
3566         (void) zvol_remove_minor(zc->zc_name);
3567     return (dsl_dir_rename(zc->zc_name, zc->zc_value));
3568 }
3569 }

3571 static int
3572 zfs_check_settable(const char *dsname, nvpair_t *pair, cred_t *cr)
3573 {
3574     const char *propname = nvpair_name(pair);
3575     boolean_t issnap = (strchr(dsname, '@') != NULL);
3576     zfs_prop_t prop = zfs_name_to_prop(propname);
3577     uint64_t intval;
3578     int err;

3580     if (prop == ZPROP_INVALID) {
3581         if (zfs_prop_user(propname)) {
3582             if (err = zfs_secpolicy_write_perms(dsname,
3583                 ZFS_DELEG_PERM_USERPROP, cr))
3584                 return (err);
3585             return (0);
3586         }

3588         if (!issnap && zfs_prop_userquota(propname)) {
3589             const char *perm = NULL;
3590             const char *uq_prefix =
3591                 zfs_userquota_prop_prefixes[ZFS_PROP_USERQUOTA];
3592             const char *gq_prefix =
3593                 zfs_userquota_prop_prefixes[ZFS_PROP_GROUPQUOTA];

3595             if (strncmp(propname, uq_prefix,
3596                 strlen(uq_prefix)) == 0) {
3597                 perm = ZFS_DELEG_PERM_USERQUOTA;
3598             } else if (strncmp(propname, gq_prefix,
3599                 strlen(gq_prefix)) == 0) {
3600                 perm = ZFS_DELEG_PERM_GROUPQUOTA;
3601             } else {
3602                 /* USERUSED and GROUPUSED are read-only */
3603                 return (SET_ERROR(EINVAL));
3604             }

3606             if (err = zfs_secpolicy_write_perms(dsname, perm, cr))
3607                 return (err);
3608             return (0);
3609         }

3611         return (SET_ERROR(EINVAL));
3612     }

3614     if (issnap)
3615         return (SET_ERROR(EINVAL));

3617     if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
3618         /*
3619          * dsl_prop_get_all_impl() returns properties in this
3620          * format.
3621          */
3622         nvlist_t *attrs;
3623         VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
3624         VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,

```

```

3625         &pair) == 0);
3626     }

3628     /*
3629     * Check that this value is valid for this pool version
3630     */
3631     switch (prop) {
3632     case ZFS_PROP_COMPRESSION:
3633         /*
3634          * If the user specified gzip compression, make sure
3635          * the SPA supports it. We ignore any errors here since
3636          * we'll catch them later.
3637          */
3638         if (nvpair_type(pair) == DATA_TYPE_UINT64 &&
3639             nvpair_value_uint64(pair, &intval) == 0) {
3640             if (intval >= ZIO_COMPRESS_GZIP_1 &&
3641                 intval <= ZIO_COMPRESS_GZIP_9 &&
3642                 zfs_earlier_version(dsname,
3643                     SPA_VERSION_GZIP_COMPRESSION)) {
3644                 return (SET_ERROR(ENOTSUP));
3645             }

3647             if (intval == ZIO_COMPRESS_ZLE &&
3648                 zfs_earlier_version(dsname,
3649                     SPA_VERSION_ZLE_COMPRESSION))
3650                 return (SET_ERROR(ENOTSUP));

3652             if (intval == ZIO_COMPRESS_LZ4) {
3653                 zfeature_info_t *feature =
3654                     &spa_feature_table[
3655                         SPA_FEATURE_LZ4_COMPRESS];
3656                 spa_t *spa;

3658                 if ((err = spa_open(dsname, &spa, FTAG)) != 0)
3659                     return (err);

3661                 if (!spa_feature_is_enabled(spa, feature)) {
3662                     spa_close(spa, FTAG);
3663                     return (SET_ERROR(ENOTSUP));
3664                 }
3665                 spa_close(spa, FTAG);
3666             }

3668             /*
3669             * If this is a bootable dataset then
3670             * verify that the compression algorithm
3671             * is supported for booting. We must return
3672             * something other than ENOTSUP since it
3673             * implies a downrev pool version.
3674             */
3675             if (zfs_is_bootfs(dsname) &&
3676                 !BOOTFS_COMPRESS_VALID(intval)) {
3677                 return (SET_ERROR(ERANGE));
3678             }
3679         }
3680         break;

3682     case ZFS_PROP_COPIES:
3683         if (zfs_earlier_version(dsname, SPA_VERSION_DITTO_BLOCKS))
3684             return (SET_ERROR(ENOTSUP));
3685         break;

3687     case ZFS_PROP_DEDUP:
3688         if (zfs_earlier_version(dsname, SPA_VERSION_DEDUP))
3689             return (SET_ERROR(ENOTSUP));
3690         break;

```

```

3692     case ZFS_PROP_SHARESMB:
3693         if (zpl_earlier_version(dsname, ZPL_VERSION_FUID))
3694             return (SET_ERROR(ENOTSUP));
3695         break;
3697     case ZFS_PROP_ACLINHERIT:
3698         if (nvpair_type(pair) == DATA_TYPE_UINT64 &&
3699             nvpair_value_uint64(pair, &intval) == 0) {
3700             if (intval == ZFS_ACL_PASSTHROUGH_X &&
3701                 zfs_earlier_version(dsname,
3702                     SPA_VERSION_PASSTHROUGH_X))
3703                 return (SET_ERROR(ENOTSUP));
3704             }
3705         break;
3706     }
3708     return (zfs_secpolicy_setprop(dsname, prop, pair, CRED()));
3709 }
3711 /*
3712  * Checks for a race condition to make sure we don't increment a feature flag
3713  * multiple times.
3714  */
3715 static int
3716 zfs_prop_activate_feature_check(void *arg, dmu_tx_t *tx)
3717 {
3718     spa_t *spa = dmu_tx_pool(tx)->dp_spa;
3719     zfeature_info_t *feature = arg;
3721     if (!spa_feature_is_active(spa, feature))
3722         return (0);
3723     else
3724         return (SET_ERROR(EBUSY));
3725 }
3727 /*
3728  * The callback invoked on feature activation in the sync task caused by
3729  * zfs_prop_activate_feature.
3730  */
3731 static void
3732 zfs_prop_activate_feature_sync(void *arg, dmu_tx_t *tx)
3733 {
3734     spa_t *spa = dmu_tx_pool(tx)->dp_spa;
3735     zfeature_info_t *feature = arg;
3737     spa_feature_incr(spa, feature, tx);
3738 }
3740 /*
3741  * Activates a feature on a pool in response to a property setting. This
3742  * creates a new sync task which modifies the pool to reflect the feature
3743  * as being active.
3744  */
3745 static int
3746 zfs_prop_activate_feature(spa_t *spa, zfeature_info_t *feature)
3747 {
3748     int err;
3750     /* EBUSY here indicates that the feature is already active */
3751     err = dsl_sync_task(spa_name(spa),
3752         zfs_prop_activate_feature_check, zfs_prop_activate_feature_sync,
3753         feature, 2);
3755     if (err != 0 && err != EBUSY)
3756         return (err);

```

```

3757     else
3758         return (0);
3759 }
3761 /*
3762  * Removes properties from the given props list that fail permission checks
3763  * needed to clear them and to restore them in case of a receive error. For each
3764  * property, make sure we have both set and inherit permissions.
3765  *
3766  * Returns the first error encountered if any permission checks fail. If the
3767  * caller provides a non-NULL errlist, it also gives the complete list of names
3768  * of all the properties that failed a permission check along with the
3769  * corresponding error numbers. The caller is responsible for freeing the
3770  * returned errlist.
3771  *
3772  * If every property checks out successfully, zero is returned and the list
3773  * pointed at by errlist is NULL.
3774  */
3775 static int
3776 zfs_check_clearable(char *dataset, nvlist_t *props, nvlist_t **errlist)
3777 {
3778     zfs_cmd_t *zc;
3779     nvpair_t *pair, *next_pair;
3780     nvlist_t *errors;
3781     int err, rv = 0;
3783     if (props == NULL)
3784         return (0);
3786     VERIFY(nvlist_alloc(&errors, NV_UNIQUE_NAME, KM_SLEEP) == 0);
3788     zc = kmem_alloc(sizeof (zfs_cmd_t), KM_SLEEP);
3789     (void) strcpy(zc->zc_name, dataset);
3790     pair = nvlist_next_nvpair(props, NULL);
3791     while (pair != NULL) {
3792         next_pair = nvlist_next_nvpair(props, pair);
3794         (void) strcpy(zc->zc_value, nvpair_name(pair));
3795         if ((err = zfs_check_settable(dataset, pair, CRED())) != 0 ||
3796             (err = zfs_secpolicy_inherit_prop(zc, NULL, CRED())) != 0) {
3797             VERIFY(nvlist_remove_nvpair(props, pair) == 0);
3798             VERIFY(nvlist_add_int32(errors,
3799                 zc->zc_value, err) == 0);
3800         }
3801         pair = next_pair;
3802     }
3803     kmem_free(zc, sizeof (zfs_cmd_t));
3805     if ((pair = nvlist_next_nvpair(errors, NULL)) == NULL) {
3806         nvlist_free(errors);
3807         errors = NULL;
3808     } else {
3809         VERIFY(nvpair_value_int32(pair, &rv) == 0);
3810     }
3812     if (errlist == NULL)
3813         nvlist_free(errors);
3814     else
3815         *errlist = errors;
3817     return (rv);
3818 }
3820 static boolean_t
3821 propval_equals(nvpair_t *p1, nvpair_t *p2)
3822 {

```

```

3823     if (nvpair_type(p1) == DATA_TYPE_NVLIST) {
3824         /* dsl_prop_get_all_impl() format */
3825         nvlist_t *attrs;
3826         VERIFY(nvpair_value_nvlist(p1, &attrs) == 0);
3827         VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
3828             &p1) == 0);
3829     }

3831     if (nvpair_type(p2) == DATA_TYPE_NVLIST) {
3832         nvlist_t *attrs;
3833         VERIFY(nvpair_value_nvlist(p2, &attrs) == 0);
3834         VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
3835             &p2) == 0);
3836     }

3838     if (nvpair_type(p1) != nvpair_type(p2))
3839         return (B_FALSE);

3841     if (nvpair_type(p1) == DATA_TYPE_STRING) {
3842         char *valstr1, *valstr2;

3844         VERIFY(nvpair_value_string(p1, (char **)&valstr1) == 0);
3845         VERIFY(nvpair_value_string(p2, (char **)&valstr2) == 0);
3846         return (strcmp(valstr1, valstr2) == 0);
3847     } else {
3848         uint64_t intvall, intval2;

3850         VERIFY(nvpair_value_uint64(p1, &intvall) == 0);
3851         VERIFY(nvpair_value_uint64(p2, &intval2) == 0);
3852         return (intvall == intval2);
3853     }
3854 }

3856 /*
3857  * Remove properties from props if they are not going to change (as determined
3858  * by comparison with origprops). Remove them from origprops as well, since we
3859  * do not need to clear or restore properties that won't change.
3860  */
3861 static void
3862 props_reduce(nvlist_t *props, nvlist_t *origprops)
3863 {
3864     nvpair_t *pair, *next_pair;

3866     if (origprops == NULL)
3867         return; /* all props need to be received */

3869     pair = nvlist_next_nvpair(props, NULL);
3870     while (pair != NULL) {
3871         const char *propname = nvpair_name(pair);
3872         nvpair_t *match;

3874         next_pair = nvlist_next_nvpair(props, pair);

3876         if ((nvlist_lookup_nvpair(origprops, propname,
3877             &match) != 0) || !propval_equals(pair, match))
3878             goto next; /* need to set received value */

3880         /* don't clear the existing received value */
3881         (void) nvlist_remove_nvpair(origprops, match);
3882         /* don't bother receiving the property */
3883         (void) nvlist_remove_nvpair(props, pair);
3884     next:
3885         pair = next_pair;
3886     }
3887 }

```

```

3889 #ifdef DEBUG
3890 static boolean_t zfs_ioc_recv_inject_err;
3891 #endif

3893 /*
3894  * inputs:
3895  * zc_name                name of containing filesystem
3896  * zc_nvlist_src[_size]  nvlist of properties to apply
3897  * zc_value              name of snapshot to create
3898  * zc_string            name of clone origin (if DRR_FLAG_CLONE)
3899  * zc_cookie            file descriptor to recv from
3900  * zc_begin_record      the BEGIN record of the stream (not byteswapped)
3901  * zc_guid              force flag
3902  * zc_cleanup_fd        cleanup-on-exit file descriptor
3903  * zc_action_handle     handle for this guid/ds mapping (or zero on first call)
3904  *
3905  * outputs:
3906  * zc_cookie            number of bytes read
3907  * zc_nvlist_dst[_size] error for each unapplied received property
3908  * zc_obj              zprop_errflags_t
3909  * zc_action_handle     handle for this guid/ds mapping
3910  */
3911 static int
3912 zfs_ioc_recv(zfs_cmd_t *zc)
3913 {
3914     file_t *fp;
3915     dmu_recv_cookie_t drc;
3916     boolean_t force = (boolean_t)zc->zc_guid;
3917     int fd;
3918     int error = 0;
3919     int props_error = 0;
3920     nvlist_t *errors;
3921     offset_t off;
3922     nvlist_t *props = NULL; /* sent properties */
3923     nvlist_t *origprops = NULL; /* existing properties */
3924     char *origin = NULL;
3925     char *tosnap;
3926     char tofs[ZFS_MAXNAMELEN];
3927     boolean_t first_recvd_props = B_FALSE;

3929     if (dataset_namecheck(zc->zc_value, NULL, NULL) != 0 ||
3930         strchr(zc->zc_value, '@') == NULL ||
3931         strchr(zc->zc_value, '%'))
3932         return (SET_ERROR(EINVAL));

3934     (void) strcpy(tofs, zc->zc_value);
3935     tosnap = strchr(tofs, '@');
3936     *tosnap++ = '\0';

3938     if (zc->zc_nvlist_src != NULL &&
3939         (error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
3940             zc->zc_iflags, &props)) != 0)
3941         return (error);

3943     fd = zc->zc_cookie;
3944     fp = getf(fd);
3945     if (fp == NULL) {
3946         nvlist_free(props);
3947         return (SET_ERROR(EBADF));
3948     }

3950     VERIFY(nvlist_alloc(&errors, NV_UNIQUE_NAME, KM_SLEEP) == 0);

3952     if (zc->zc_string[0])
3953         origin = zc->zc_string;

```

```

3955     error = dmuf_recv_begin(tofs, tosnap,
3956         &zc->zc_begin_record, force, origin, &drc);
3957     if (error != 0)
3958         goto out;

3960     /*
3961     * Set properties before we receive the stream so that they are applied
3962     * to the new data. Note that we must call dmuf_recv_stream() if
3963     * dmuf_recv_begin() succeeds.
3964     */
3965     if (props != NULL && !drc.drc_newfs) {
3966         if (spa_version(dsl_dataset_get_spa(drc.drc_ds)) >=
3967             SPA_VERSION_RECVD_PROPS &&
3968             !dsl_prop_get_hasrecvd(tofs))
3969             first_recvd_props = B_TRUE;

3971     /*
3972     * If new received properties are supplied, they are to
3973     * completely replace the existing received properties, so stash
3974     * away the existing ones.
3975     */
3976     if (dsl_prop_get_received(tofs, &origprops) == 0) {
3977         nvlist_t *errlist = NULL;
3978         /*
3979         * Don't bother writing a property if its value won't
3980         * change (and avoid the unnecessary security checks).
3981         *
3982         * The first receive after SPA_VERSION_RECVD_PROPS is a
3983         * special case where we blow away all local properties
3984         * regardless.
3985         */
3986         if (!first_recvd_props)
3987             props_reduce(props, origprops);
3988         if (zfs_check_clearable(tofs, origprops, &errlist) != 0)
3989             (void) nvlist_merge(errors, errlist, 0);
3990         nvlist_free(errlist);

3992         if (clear_received_props(tofs, origprops,
3993             first_recvd_props ? NULL : props) != 0)
3994             zc->zc_obj |= ZPROP_ERR_NOCLEAR;
3995     } else {
3996         zc->zc_obj |= ZPROP_ERR_NOCLEAR;
3997     }
3998 }

4000 if (props != NULL) {
4001     props_error = dsl_prop_set_hasrecvd(tofs);

4003     if (props_error == 0) {
4004         (void) zfs_set_prop_nvlist(tofs, ZPROP_SRC_RECEIVED,
4005             props, errors);
4006     }
4007 }

4009 if (zc->zc_nvlist_dst_size != 0 &&
4010     (nvlist_smush(errors, zc->zc_nvlist_dst_size) != 0 ||
4011     put_nvlist(zc, errors) != 0)) {
4012     /*
4013     * Caller made zc->zc_nvlist_dst less than the minimum expected
4014     * size or supplied an invalid address.
4015     */
4016     props_error = SET_ERROR(EINVAL);
4017 }

4019 off = fp->f_offset;
4020 error = dmuf_recv_stream(&drc, fp->f_vnode, &off, zc->zc_cleanup_fd,

```

```

4021     &zc->zc_action_handle);

4023     if (error == 0) {
4024         zfsvfs_t *zfsvfs = NULL;

4026         if (getzfsvfs(tofs, &zfsvfs) == 0) {
4027             /* online recv */
4028             int end_err;

4030             error = zfs_suspend_fs(zfsvfs);
4031             /*
4032             * If the suspend fails, then the recv_end will
4033             * likely also fail, and clean up after itself.
4034             */
4035             end_err = dmuf_recv_end(&drc);
4036             if (error == 0)
4037                 error = zfs_resume_fs(zfsvfs, tofs);
4038             error = error ? error : end_err;
4039             VFS_RELE(zfsvfs->z_vfs);
4040         } else {
4041             error = dmuf_recv_end(&drc);
4042         }
4043     }

4045     zc->zc_cookie = off - fp->f_offset;
4046     if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
4047         fp->f_offset = off;

4049 #ifdef DEBUG
4050     if (zfs_ioc_recv_inject_err) {
4051         zfs_ioc_recv_inject_err = B_FALSE;
4052         error = 1;
4053     }
4054 #endif
4055     /*
4056     * On error, restore the original props.
4057     */
4058     if (error != 0 && props != NULL && !drc.drc_newfs) {
4059         if (clear_received_props(tofs, props, NULL) != 0) {
4060             /*
4061             * We failed to clear the received properties.
4062             * Since we may have left a $recvd value on the
4063             * system, we can't clear the $hasrecvd flag.
4064             */
4065             zc->zc_obj |= ZPROP_ERR_NORESTORE;
4066         } else if (first_recvd_props) {
4067             dsl_prop_unset_hasrecvd(tofs);
4068         }
4069     }

4070     if (origprops == NULL && !drc.drc_newfs) {
4071         /* We failed to stash the original properties. */
4072         zc->zc_obj |= ZPROP_ERR_NORESTORE;
4073     }

4075     /*
4076     * dsl_props_set() will not convert RECEIVED to LOCAL on or
4077     * after SPA_VERSION_RECVD_PROPS, so we need to specify LOCAL
4078     * explicitly if we're restoring local properties cleared in the
4079     * first new-style receive.
4080     */
4081     if (origprops != NULL &&
4082         zfs_set_prop_nvlist(tofs, (first_recvd_props ?
4083             ZPROP_SRC_LOCAL : ZPROP_SRC_RECEIVED),
4084         origprops, NULL) != 0) {
4085         /*
4086         * We stashed the original properties but failed to

```

```

4087         * restore them.
4088         */
4089         zc->zc_obj |= ZPROP_ERR_NORESTORE;
4090     }
4091 }
4092 out:
4093     nvlist_free(props);
4094     nvlist_free(origprops);
4095     nvlist_free(errors);
4096     releasef(fd);
4098     if (error == 0)
4099         error = props_error;
4101     return (error);
4102 }
4104 /*
4105  * inputs:
4106  * zc_name      name of snapshot to send
4107  * zc_cookie    file descriptor to send stream to
4108  * zc_obj       fromorigin flag (mutually exclusive with zc_fromobj)
4109  * zc_sendobj   objsetid of snapshot to send
4110  * zc_fromobj   objsetid of incremental fromsnap (may be zero)
4111  * zc_guid      if set, estimate size of stream only.  zc_cookie is ignored.
4112  *             output size in zc_objset_type.
4113  *
4114  * outputs: none
4115  */
4116 static int
4117 zfs_ioc_send(zfs_cmd_t *zc)
4118 {
4119     int error;
4120     offset_t off;
4121     boolean_t estimate = (zc->zc_guid != 0);
4123     if (zc->zc_obj != 0) {
4124         dsl_pool_t *dp;
4125         dsl_dataset_t *tosnap;
4127         error = dsl_pool_hold(zc->zc_name, FTAG, &dp);
4128         if (error != 0)
4129             return (error);
4131         error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &tosnap);
4132         if (error != 0) {
4133             dsl_pool_rele(dp, FTAG);
4134             return (error);
4135         }
4137         if (dsl_dir_is_clone(tosnap->ds_dir))
4138             zc->zc_fromobj = tosnap->ds_dir->dd_phys->dd_origin_obj;
4139         dsl_dataset_rele(tosnap, FTAG);
4140         dsl_pool_rele(dp, FTAG);
4141     }
4143     if (estimate) {
4144         dsl_pool_t *dp;
4145         dsl_dataset_t *tosnap;
4146         dsl_dataset_t *fromsnap = NULL;
4148         error = dsl_pool_hold(zc->zc_name, FTAG, &dp);
4149         if (error != 0)
4150             return (error);
4152         error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &tosnap);

```

```

4153         if (error != 0) {
4154             dsl_pool_rele(dp, FTAG);
4155             return (error);
4156         }
4158         if (zc->zc_fromobj != 0) {
4159             error = dsl_dataset_hold_obj(dp, zc->zc_fromobj,
4160                 FTAG, &fromsnap);
4161             if (error != 0) {
4162                 dsl_dataset_rele(tosnap, FTAG);
4163                 dsl_pool_rele(dp, FTAG);
4164                 return (error);
4165             }
4166         }
4168         error = dmu_send_estimate(tosnap, fromsnap,
4169             &zc->zc_objset_type);
4171         if (fromsnap != NULL)
4172             dsl_dataset_rele(fromsnap, FTAG);
4173         dsl_dataset_rele(tosnap, FTAG);
4174         dsl_pool_rele(dp, FTAG);
4175     } else {
4176         file_t *fp = getf(zc->zc_cookie);
4177         if (fp == NULL)
4178             return (SET_ERROR(EBADF));
4180         off = fp->f_offset;
4181         error = dmu_send_obj(zc->zc_name, zc->zc_sendobj,
4182             zc->zc_fromobj, zc->zc_cookie, fp->f_vnode, &off);
4184         if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
4185             fp->f_offset = off;
4186         releasef(zc->zc_cookie);
4187     }
4188     return (error);
4189 }
4191 /*
4192  * inputs:
4193  * zc_name      name of snapshot on which to report progress
4194  * zc_cookie    file descriptor of send stream
4195  *
4196  * outputs:
4197  * zc_cookie    number of bytes written in send stream thus far
4198  */
4199 static int
4200 zfs_ioc_send_progress(zfs_cmd_t *zc)
4201 {
4202     dsl_pool_t *dp;
4203     dsl_dataset_t *ds;
4204     dmu_sendarg_t *dsp = NULL;
4205     int error;
4207     error = dsl_pool_hold(zc->zc_name, FTAG, &dp);
4208     if (error != 0)
4209         return (error);
4211     error = dsl_dataset_hold(dp, zc->zc_name, FTAG, &ds);
4212     if (error != 0) {
4213         dsl_pool_rele(dp, FTAG);
4214         return (error);
4215     }
4217     mutex_enter(&ds->ds_sendstream_lock);

```

```

4219  /*
4220  * Iterate over all the send streams currently active on this dataset.
4221  * If there's one which matches the specified file descriptor _and_ the
4222  * stream was started by the current process, return the progress of
4223  * that stream.
4224  */
4225  for (dsp = list_head(&ds->ds_sendstreams); dsp != NULL;
4226       dsp = list_next(&ds->ds_sendstreams, dsp)) {
4227     if (dsp->dsa_outfd == zc->zc_cookie &&
4228         dsp->dsa_proc == curproc)
4229         break;
4230 }

4232 if (dsp != NULL)
4233     zc->zc_cookie = *(dsp->dsa_off);
4234 else
4235     error = SET_ERROR(ENOENT);

4237 mutex_exit(&ds->ds_sendstream_lock);
4238 dsl_dataset_rele(ds, FTAG);
4239 dsl_pool_rele(dp, FTAG);
4240 return (error);
4241 }

4243 static int
4244 zfs_ioc_inject_fault(zfs_cmd_t *zc)
4245 {
4246     int id, error;

4248     error = zio_inject_fault(zc->zc_name, (int)zc->zc_guid, &id,
4249                             &zc->zc_inject_record);

4251     if (error == 0)
4252         zc->zc_guid = (uint64_t)id;

4254     return (error);
4255 }

4257 static int
4258 zfs_ioc_clear_fault(zfs_cmd_t *zc)
4259 {
4260     return (zio_clear_fault((int)zc->zc_guid));
4261 }

4263 static int
4264 zfs_ioc_inject_list_next(zfs_cmd_t *zc)
4265 {
4266     int id = (int)zc->zc_guid;
4267     int error;

4269     error = zio_inject_list_next(&id, zc->zc_name, sizeof (zc->zc_name),
4270                                 &zc->zc_inject_record);

4272     zc->zc_guid = id;

4274     return (error);
4275 }

4277 static int
4278 zfs_ioc_error_log(zfs_cmd_t *zc)
4279 {
4280     spa_t *spa;
4281     int error;
4282     size_t count = (size_t)zc->zc_nvlist_dst_size;

4284     if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)

```

```

4285         return (error);

4287     error = spa_get_errlog(spa, (void *) (uintptr_t)zc->zc_nvlist_dst,
4288                             &count);
4289     if (error == 0)
4290         zc->zc_nvlist_dst_size = count;
4291     else
4292         zc->zc_nvlist_dst_size = spa_get_errlog_size(spa);

4294     spa_close(spa, FTAG);

4296     return (error);
4297 }

4299 static int
4300 zfs_ioc_clear(zfs_cmd_t *zc)
4301 {
4302     spa_t *spa;
4303     vdev_t *vd;
4304     int error;

4306     /*
4307     * On zpool clear we also fix up missing slogs
4308     */
4309     mutex_enter(&spa_namespace_lock);
4310     spa = spa_lookup(zc->zc_name);
4311     if (spa == NULL) {
4312         mutex_exit(&spa_namespace_lock);
4313         return (SET_ERROR(EIO));
4314     }
4315     if (spa_get_log_state(spa) == SPA_LOG_MISSING) {
4316         /* we need to let spa_open/spa_load clear the chains */
4317         spa_set_log_state(spa, SPA_LOG_CLEAR);
4318     }
4319     spa->spa_last_open_failed = 0;
4320     mutex_exit(&spa_namespace_lock);

4322     if (zc->zc_cookie & ZPOOL_NO_REWIND) {
4323         error = spa_open(zc->zc_name, &spa, FTAG);
4324     } else {
4325         nvlist_t *policy;
4326         nvlist_t *config = NULL;

4328         if (zc->zc_nvlist_src == NULL)
4329             return (SET_ERROR(EINVAL));

4331         if ((error = get_nvlist(zc->zc_nvlist_src,
4332                                 zc->zc_nvlist_src_size, zc->zc_iflags, &policy)) == 0) {
4333             error = spa_open_rewind(zc->zc_name, &spa, FTAG,
4334                                     policy, &config);
4335             if (config != NULL) {
4336                 int err;

4338                 if ((err = put_nvlist(zc, config)) != 0)
4339                     error = err;
4340                 nvlist_free(config);
4341             }
4342             nvlist_free(policy);
4343         }
4344     }

4346     if (error != 0)
4347         return (error);

4349     spa_vdev_state_enter(spa, SCL_NONE);

```

```

4351     if (zc->zc_guid == 0) {
4352         vd = NULL;
4353     } else {
4354         vd = spa_lookup_by_guid(spa, zc->zc_guid, B_TRUE);
4355         if (vd == NULL) {
4356             (void) spa_vdev_state_exit(spa, NULL, ENODEV);
4357             spa_close(spa, FTAG);
4358             return (SET_ERROR(ENODEV));
4359         }
4360     }
4362     vdev_clear(spa, vd);
4364     (void) spa_vdev_state_exit(spa, NULL, 0);
4366     /*
4367     * Resume any suspended I/Os.
4368     */
4369     if (zio_resume(spa) != 0)
4370         error = SET_ERROR(EIO);
4372     spa_close(spa, FTAG);
4374     return (error);
4375 }
4377 static int
4378 zfs_ioc_pool_reopen(zfs_cmd_t *zc)
4379 {
4380     spa_t *spa;
4381     int error;
4383     error = spa_open(zc->zc_name, &spa, FTAG);
4384     if (error != 0)
4385         return (error);
4387     spa_vdev_state_enter(spa, SCL_NONE);
4389     /*
4390     * If a resilver is already in progress then set the
4391     * spa_scrub_reopen flag to B_TRUE so that we don't restart
4392     * the scan as a side effect of the reopen. Otherwise, let
4393     * vdev_open() decide if a resilver is required.
4394     */
4395     spa->spa_scrub_reopen = dsl_scan_resilvering(spa->spa_dsl_pool);
4396     vdev_reopen(spa->spa_root_vdev);
4397     spa->spa_scrub_reopen = B_FALSE;
4399     (void) spa_vdev_state_exit(spa, NULL, 0);
4400     spa_close(spa, FTAG);
4401     return (0);
4402 }
4403 /*
4404 * inputs:
4405 * zc_name      name of filesystem
4406 * zc_value     name of origin snapshot
4407 *
4408 * outputs:
4409 * zc_string    name of conflicting snapshot, if there is one
4410 */
4411 static int
4412 zfs_ioc_promote(zfs_cmd_t *zc)
4413 {
4414     char *cp;
4416     /*

```

```

4417     * We don't need to unmount *all* the origin fs's snapshots, but
4418     * it's easier.
4419     */
4420     cp = strchr(zc->zc_value, '@');
4421     if (cp)
4422         *cp = '\0';
4423     (void) dmu_objset_find(zc->zc_value,
4424         zfs_unmount_snap_cb, NULL, DS_FIND_SNAPSHOTS);
4425     return (dsl_dataset_promote(zc->zc_name, zc->zc_string));
4426 }
4428 /*
4429 * Retrieve a single {user|group}{used|quota}@... property.
4430 *
4431 * inputs:
4432 * zc_name      name of filesystem
4433 * zc_objset_type zfs_userquota_prop_t
4434 * zc_value     domain name (eg. "S-1-234-567-89")
4435 * zc_guid      RID/UID/GID
4436 *
4437 * outputs:
4438 * zc_cookie    property value
4439 */
4440 static int
4441 zfs_ioc_userspace_one(zfs_cmd_t *zc)
4442 {
4443     zfsvfs_t *zfsvfs;
4444     int error;
4446     if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
4447         return (SET_ERROR(EINVAL));
4449     error = zfsvfs_hold(zc->zc_name, FTAG, &zfsvfs, B_FALSE);
4450     if (error != 0)
4451         return (error);
4453     error = zfs_userspace_one(zfsvfs,
4454         zc->zc_objset_type, zc->zc_value, zc->zc_guid, &zc->zc_cookie);
4455     zfsvfs_rele(zfsvfs, FTAG);
4457     return (error);
4458 }
4460 /*
4461 * inputs:
4462 * zc_name      name of filesystem
4463 * zc_cookie    zap cursor
4464 * zc_objset_type zfs_userquota_prop_t
4465 * zc_nvlist_dst[_size] buffer to fill (not really an nvlist)
4466 *
4467 * outputs:
4468 * zc_nvlist_dst[_size] data buffer (array of zfs_useracct_t)
4469 * zc_cookie    zap cursor
4470 */
4471 static int
4472 zfs_ioc_userspace_many(zfs_cmd_t *zc)
4473 {
4474     zfsvfs_t *zfsvfs;
4475     int bufsize = zc->zc_nvlist_dst_size;
4477     if (bufsize <= 0)
4478         return (SET_ERROR(ENOMEM));
4480     int error = zfsvfs_hold(zc->zc_name, FTAG, &zfsvfs, B_FALSE);
4481     if (error != 0)
4482         return (error);

```



```

4484 void *buf = kmem_alloc(bufsize, KM_SLEEP);
4486 error = zfs_userspace_many(zfsvfs, zc->zc_objset_type, &zc->zc_cookie,
4487     buf, &zc->zc_nvlist_dst_size);
4489 if (error == 0) {
4490     error = xcopypout(buf,
4491         (void *) (uintptr_t) zc->zc_nvlist_dst,
4492         zc->zc_nvlist_dst_size);
4493 }
4494 kmem_free(buf, bufsize);
4495 zfsvfs_rele(zfsvfs, FTAG);
4497 return (error);
4498 }
4500 /*
4501  * inputs:
4502  * zc_name      name of filesystem
4503  *
4504  * outputs:
4505  * none
4506  */
4507 static int
4508 zfs_ioc_userspace_upgrade(zfs_cmd_t *zc)
4509 {
4510     objset_t *os;
4511     int error = 0;
4512     zfsvfs_t *zfsvfs;
4514     if (getzfsvfs(zc->zc_name, &zfsvfs) == 0) {
4515         if (!dmu_objset_userused_enabled(zfsvfs->z_os)) {
4516             /*
4517              * If userused is not enabled, it may be because the
4518              * objset needs to be closed & reopened (to grow the
4519              * objset_phys_t). Suspend/resume the fs will do that.
4520              */
4521             error = zfs_suspend_fs(zfsvfs);
4522             if (error == 0)
4523                 error = zfs_resume_fs(zfsvfs, zc->zc_name);
4524         }
4525         if (error == 0)
4526             error = dmu_objset_userspace_upgrade(zfsvfs->z_os);
4527         VFS_RELE(zfsvfs->z_vfs);
4528     } else {
4529         /* XXX kind of reading contents without owning */
4530         error = dmu_objset_hold(zc->zc_name, FTAG, &os);
4531         if (error != 0)
4532             return (error);
4534         error = dmu_objset_userspace_upgrade(os);
4535         dmu_objset_rele(os, FTAG);
4536     }
4538     return (error);
4539 }
4541 /*
4542  * We don't want to have a hard dependency
4543  * against some special symbols in sharefs
4544  * nfs, and smbshr. Determine them if needed when
4545  * the first file system is shared.
4546  * Neither sharefs, nfs or smbshr are unloadable modules.
4547  */
4548 int (*zfnsexport_fs)(void *arg);

```

```

4549 int (*zshare_fs)(enum sharefs_sys_op, share_t *, uint32_t);
4550 int (*zsmbexport_fs)(void *arg, boolean_t add_share);
4552 int zfs_nfsshare_initd;
4553 int zfs_smbshare_initd;
4555 ddi_modhandle_t nfs_mod;
4556 ddi_modhandle_t sharefs_mod;
4557 ddi_modhandle_t smbshr_mod;
4558 kmutex_t zfs_share_lock;
4560 static int
4561 zfs_init_sharefs()
4562 {
4563     int error;
4565     ASSERT(MUTEX_HELD(&zfs_share_lock));
4566     /* Both NFS and SMB shares also require sharetab support. */
4567     if (sharefs_mod == NULL && ((sharefs_mod =
4568         ddi_modopen("fs/sharefs",
4569             KRTLD_MODE_FIRST, &error)) == NULL)) {
4570         return (SET_ERROR(ENOSYS));
4571     }
4572     if (zshare_fs == NULL && ((zshare_fs =
4573         (int (*)(enum sharefs_sys_op, share_t *, uint32_t))
4574         ddi_modsym(sharefs_mod, "sharefs_impl", &error)) == NULL)) {
4575         return (SET_ERROR(ENOSYS));
4576     }
4577     return (0);
4578 }
4580 static int
4581 zfs_ioc_share(zfs_cmd_t *zc)
4582 {
4583     int error;
4584     int opcode;
4586     switch (zc->zc_share.z_sharetype) {
4587     case ZFS_SHARE_NFS:
4588     case ZFS_UNSHARE_NFS:
4589         if (zfs_nfsshare_initd == 0) {
4590             mutex_enter(&zfs_share_lock);
4591             if (nfs_mod == NULL && ((nfs_mod = ddi_modopen("fs/nfs",
4592                 KRTLD_MODE_FIRST, &error)) == NULL)) {
4593                 mutex_exit(&zfs_share_lock);
4594                 return (SET_ERROR(ENOSYS));
4595             }
4596             if (zfnsexport_fs == NULL &&
4597                 ((zfnsexport_fs = (int (*)(void *))
4598                 ddi_modsym(nfs_mod,
4599                     "nfs_export", &error)) == NULL)) {
4600                 mutex_exit(&zfs_share_lock);
4601                 return (SET_ERROR(ENOSYS));
4602             }
4603             error = zfs_init_sharefs();
4604             if (error != 0) {
4605                 mutex_exit(&zfs_share_lock);
4606                 return (SET_ERROR(ENOSYS));
4607             }
4608             zfs_nfsshare_initd = 1;
4609             mutex_exit(&zfs_share_lock);
4610         }
4611         break;
4612     case ZFS_SHARE_SMB:
4613     case ZFS_UNSHARE_SMB:
4614         if (zfs_smbshare_initd == 0) {

```

```

4615     mutex_enter(&zfs_share_lock);
4616     if (smbsrv_mod == NULL && ((smbsrv_mod =
4617         ddi_modopen("drv/smbdrv",
4618             KRTLD_MODE_FIRST, &error)) == NULL)) {
4619         mutex_exit(&zfs_share_lock);
4620         return (SET_ERROR(ENOSYS));
4621     }
4622     if (zsmsexport_fs == NULL && ((zsmsexport_fs =
4623         (int (*)(void *, boolean_t))ddi_modsym(smbsrv_mod,
4624             "smb_server_share", &error)) == NULL)) {
4625         mutex_exit(&zfs_share_lock);
4626         return (SET_ERROR(ENOSYS));
4627     }
4628     error = zfs_init_sharefs();
4629     if (error != 0) {
4630         mutex_exit(&zfs_share_lock);
4631         return (SET_ERROR(ENOSYS));
4632     }
4633     zfs_smbshare_inited = 1;
4634     mutex_exit(&zfs_share_lock);
4635 }
4636 break;
4637 default:
4638     return (SET_ERROR(EINVAL));
4639 }

4641 switch (zc->zshare.z_sharetype) {
4642 case ZFS_SHARE_NFS:
4643 case ZFS_UNSHARE_NFS:
4644     if (error =
4645         zfs_export_fs((void *)
4646             (uintptr_t)zc->zshare.z_exportdata))
4647         return (error);
4648     break;
4649 case ZFS_SHARE_SMB:
4650 case ZFS_UNSHARE_SMB:
4651     if (error = zsmsexport_fs((void *)
4652         (uintptr_t)zc->zshare.z_exportdata,
4653         zc->zshare.z_sharetype == ZFS_SHARE_SMB ?
4654         B_TRUE: B_FALSE)) {
4655         return (error);
4656     }
4657     break;
4658 }

4660 opcode = (zc->zshare.z_sharetype == ZFS_SHARE_NFS ||
4661     zc->zshare.z_sharetype == ZFS_SHARE_SMB) ?
4662     SHAREFS_ADD : SHAREFS_REMOVE;

4664 /*
4665  * Add or remove share from sharetab
4666  */
4667 error = zshare_fs(opcode,
4668     (void *) (uintptr_t)zc->zshare.z_sharedata,
4669     zc->zshare.z_sharemax);

4671     return (error);

4673 }

4675 ace_t full_access[] = {
4676     {(uid_t)-1, ACE_ALL_PERMS, ACE_EVERYONE, 0}
4677 };

4679 /*
4680  * inputs:

```

```

4681  * zc_name           name of containing filesystem
4682  * zc_obj            object # beyond which we want next in-use object #
4683  *
4684  * outputs:
4685  * zc_obj            next in-use object #
4686  */
4687 static int
4688 zfs_ioc_next_obj(zfs_cmd_t *zc)
4689 {
4690     objset_t *os = NULL;
4691     int error;

4693     error = dmub_objset_hold(zc->zshare.zc_name, FTAG, &os);
4694     if (error != 0)
4695         return (error);

4697     error = dmub_object_next(os, &zc->zshare.zc_obj, B_FALSE,
4698         os->os_dsl_dataset->ds_phys->ds_prev_snap_txg);

4700     dmub_objset_rele(os, FTAG);
4701     return (error);
4702 }

4704 /*
4705  * inputs:
4706  * zc_name           name of filesystem
4707  * zc_value          prefix name for snapshot
4708  * zc_cleanup_fd    cleanup-on-exit file descriptor for calling process
4709  *
4710  * outputs:
4711  * zc_value          short name of new snapshot
4712  */
4713 static int
4714 zfs_ioc_tmp_snapshot(zfs_cmd_t *zc)
4715 {
4716     char *snap_name;
4717     char *hold_name;
4718     int error;
4719     minor_t minor;

4721     error = zfs_onexit_fd_hold(zc->zshare.zc_cleanup_fd, &minor);
4722     if (error != 0)
4723         return (error);

4725     snap_name = kmem_asprintf("%s-%016llx", zc->zshare.zc_value,
4726         (u_longlong_t)ddi_get_lbolt64());
4727     hold_name = kmem_asprintf("%s", zc->zshare.zc_value);

4729     error = dsl_dataset_snapshot_tmp(zc->zshare.zc_name, snap_name, minor,
4730         hold_name);
4731     if (error == 0)
4732         (void) strcpy(zc->zshare.zc_value, snap_name);
4733     strfree(snap_name);
4734     strfree(hold_name);
4735     zfs_onexit_fd_rele(zc->zshare.zc_cleanup_fd);
4736     return (error);
4737 }

4739 /*
4740  * inputs:
4741  * zc_name           name of "to" snapshot
4742  * zc_value          name of "from" snapshot
4743  * zc_cookie         file descriptor to write diff data on
4744  *
4745  * outputs:
4746  * dmub_diff_record_t's to the file descriptor

```

```

4747 */
4748 static int
4749 zfs_ioc_diff(zfs_cmd_t *zc)
4750 {
4751     file_t *fp;
4752     offset_t off;
4753     int error;
4754
4755     fp = getf(zc->zc_cookie);
4756     if (fp == NULL)
4757         return (SET_ERROR(EBADF));
4758
4759     off = fp->f_offset;
4760
4761     error = dmu_diff(zc->zc_name, zc->zc_value, fp->f_vnode, &off);
4762
4763     if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
4764         fp->f_offset = off;
4765     releasef(zc->zc_cookie);
4766
4767     return (error);
4768 }
4769
4770 /*
4771  * Remove all ACL files in shares dir
4772  */
4773 static int
4774 zfs_smb_acl_purge(znode_t *dzp)
4775 {
4776     zap_cursor_t zc;
4777     zap_attribute_t zap;
4778     zfsvfs_t *zfsvfs = dzp->z_zfsvfs;
4779     int error;
4780
4781     for (zap_cursor_init(&zc, zfsvfs->z_os, dzp->z_id);
4782          (error = zap_cursor_retrieve(&zc, &zap)) == 0;
4783          zap_cursor_advance(&zc)) {
4784         if ((error = VOP_REMOVE(ZTOV(dzp), zap.za_name, kcred,
4785                                NULL, 0)) != 0)
4786             break;
4787     }
4788     zap_cursor_fini(&zc);
4789     return (error);
4790 }
4791
4792 static int
4793 zfs_ioc_smb_acl(zfs_cmd_t *zc)
4794 {
4795     vnode_t *vp;
4796     znode_t *dzp;
4797     vnode_t *resourcevp = NULL;
4798     znode_t *sharedir;
4799     zfsvfs_t *zfsvfs;
4800     nvlist_t *nvlist;
4801     char *src, *target;
4802     vattr_t vattr;
4803     vsecattr_t vsec;
4804     int error = 0;
4805
4806     if ((error = lookupname(zc->zc_value, UIO_SYSSPACE,
4807                            NO_FOLLOW, NULL, &vp)) != 0)
4808         return (error);
4809
4810     /* Now make sure mntpnt and dataset are ZFS */
4811     if (vp->v_vfsp->vfs_fstype != zfsfstype ||

```

```

4813         (strcmp((char *)refstr_value(vp->v_vfsp->vfs_resource),
4814                zc->zc_name) != 0)) {
4815             VN_RELE(vp);
4816             return (SET_ERROR(EINVAL));
4817         }
4818
4819     dzp = VTOZ(vp);
4820     zfsvfs = dzp->z_zfsvfs;
4821     ZFS_ENTER(zfsvfs);
4822
4823     /*
4824      * Create share dir if its missing.
4825      */
4826     mutex_enter(&zfsvfs->z_lock);
4827     if (zfsvfs->z_shares_dir == 0) {
4828         dmu_tx_t *tx;
4829
4830         tx = dmu_tx_create(zfsvfs->z_os);
4831         dmu_tx_hold_zap(tx, MASTER_NODE_OBJ, TRUE,
4832                        ZFS_SHARES_DIR);
4833         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, FALSE, NULL);
4834         error = dmu_tx_assign(tx, TXG_WAIT);
4835         if (error != 0) {
4836             dmu_tx_abort(tx);
4837         } else {
4838             error = zfs_create_share_dir(zfsvfs, tx);
4839             dmu_tx_commit(tx);
4840         }
4841         if (error != 0) {
4842             mutex_exit(&zfsvfs->z_lock);
4843             VN_RELE(vp);
4844             ZFS_EXIT(zfsvfs);
4845             return (error);
4846         }
4847     }
4848     mutex_exit(&zfsvfs->z_lock);
4849
4850     ASSERT(zfsvfs->z_shares_dir);
4851     if ((error = zfs_zget(zfsvfs, zfsvfs->z_shares_dir, &sharedir)) != 0) {
4852         VN_RELE(vp);
4853         ZFS_EXIT(zfsvfs);
4854         return (error);
4855     }
4856
4857     switch (zc->zc_cookie) {
4858     case ZFS_SMB_ACL_ADD:
4859         vattr.va_mask = AT_MODE|AT_UID|AT_GID|AT_TYPE;
4860         vattr.va_type = VREG;
4861         vattr.va_mode = S_IFREG|0777;
4862         vattr.va_uid = 0;
4863         vattr.va_gid = 0;
4864
4865         vsec.vsa_mask = VSA_ACE;
4866         vsec.vsa_aclentp = &full_access;
4867         vsec.vsa_aclentsz = sizeof (full_access);
4868         vsec.vsa_aclcnt = 1;
4869
4870         error = VOP_CREATE(ZTOV(sharedir), zc->zc_string,
4871                           &vattr, EXCL, 0, &resourcevp, kcred, 0, NULL, &vsec);
4872         if (resourcevp)
4873             VN_RELE(resourcevp);
4874         break;
4875
4876     case ZFS_SMB_ACL_REMOVE:
4877         error = VOP_REMOVE(ZTOV(sharedir), zc->zc_string, kcred,
4878                           NULL, 0);

```

```

4879         break;
4881     case ZFS_SMB_ACL_RENAME:
4882         if ((error = get_nvlist(zc->zcvlist_src,
4883             zc->zcvlist_src_size, zc->zcviflags, &nvlist)) != 0) {
4884             VN_RELE(vp);
4885             ZFS_EXIT(zfsvfs);
4886             return (error);
4887         }
4888         if (nvlist_lookup_string(nvlist, ZFS_SMB_ACL_SRC, &src) ||
4889             nvlist_lookup_string(nvlist, ZFS_SMB_ACL_TARGET,
4890                 &target)) {
4891             VN_RELE(vp);
4892             VN_RELE(ZTOV(sharedir));
4893             ZFS_EXIT(zfsvfs);
4894             nvlist_free(nvlist);
4895             return (error);
4896         }
4897         error = VOP_RENAME(ZTOV(sharedir), src, ZTOV(sharedir), target,
4898             kcred, NULL, 0);
4899         nvlist_free(nvlist);
4900         break;
4902     case ZFS_SMB_ACL_PURGE:
4903         error = zfs_smb_acl_purge(sharedir);
4904         break;
4906     default:
4907         error = SET_ERROR(EINVAL);
4908         break;
4909 }
4911 VN_RELE(vp);
4912 VN_RELE(ZTOV(sharedir));
4914 ZFS_EXIT(zfsvfs);
4916 return (error);
4917 }
4919 /*
4920 * innvl: {
4921 *     "holds" -> { snapname -> holdname (string), ... }
4922 *     (optional) "cleanup_fd" -> fd (int32)
4923 * }
4924 *
4925 * outnvl: {
4926 *     snapname -> error value (int32)
4927 *     ...
4928 * }
4929 */
4930 /* ARGSUSED */
4931 static int
4932 zfs_ioc_hold(const char *pool, nvlist_t *args, nvlist_t *errlist)
4933 {
4934     nvlist_t *holds;
4935     int cleanup_fd = -1;
4936     int error;
4937     minor_t minor = 0;
4939     error = nvlist_lookup_nvlist(args, "holds", &holds);
4940     if (error != 0)
4941         return (SET_ERROR(EINVAL));
4943     if (nvlist_lookup_int32(args, "cleanup_fd", &cleanup_fd) == 0) {
4944         error = zfs_onexit_fd_hold(cleanup_fd, &minor);

```

```

4945         if (error != 0)
4946             return (error);
4947     }
4949     error = dsl_dataset_user_hold(holds, minor, errlist);
4950     if (minor != 0)
4951         zfs_onexit_fd_rele(cleanup_fd);
4952     return (error);
4953 }
4955 /*
4956 * innvl is not used.
4957 *
4958 * outnvl: {
4959 *     holdname -> time added (uint64 seconds since epoch)
4960 *     ...
4961 * }
4962 */
4963 /* ARGSUSED */
4964 static int
4965 zfs_ioc_get_holds(const char *snapname, nvlist_t *args, nvlist_t *outnvl)
4966 {
4967     return (dsl_dataset_get_holds(snapname, outnvl));
4968 }
4970 /*
4971 * innvl: {
4972 *     snapname -> { holdname, ... }
4973 *     ...
4974 * }
4975 *
4976 * outnvl: {
4977 *     snapname -> error value (int32)
4978 *     ...
4979 * }
4980 */
4981 /* ARGSUSED */
4982 static int
4983 zfs_ioc_release(const char *pool, nvlist_t *holds, nvlist_t *errlist)
4984 {
4985     29     nvpair_t *pair;
4986
4987     31     /*
4988     32      * The release may cause the snapshot to be destroyed; make sure it
4989     33      * is not mounted.
4990     34      */
4991     35     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
4992         36         pair = nvlist_next_nvpair(holds, pair))
4993         37         zfs_unmount_snap(nvpair_name(pair));
4985     return (dsl_dataset_user_release(holds, errlist));
4986 }

```

unchanged portion omitted