

```
new/usr/src/lib/libzfs/Makefile.com
```

```
*****
2278 Wed May 14 12:03:03 2014
new/usr/src/lib/libzfs/Makefile.com
zpool import is braindead
*****
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 2014 RackTop Systems.
25 #endif /* ! codereview */
26 #
28 LIBRARY= libzfs.a
29 VERS= .1
31 OBJS_SHARED=
32     zfeature_common.o \
33     zfs_comutil.o \
34     zfs_deleg.o \
35     zfs_fletcher.o \
36     zfs_namecheck.o \
37     zfs_prop.o \
38     zpool_prop.o \
39     zprop_common.o
41 OBJS_COMMON=
42     libzfs_changelist.o \
43     libzfs_config.o \
44     libzfs_dataset.o \
45     libzfs_diff.o \
46     libzfs_fru.o \
47     libzfs_import.o \
48     libzfs_iter.o \
49     libzfs_mount.o \
50     libzfs_pool.o \
51     libzfs_sendrecv.o \
52     libzfs_status.o \
53     libzfs_util.o
55 OBJECTS= $(OBJS_COMMON) $(OBJS_SHARED)
57 include ../../Makefile.lib
59 # libzfs must be installed in the root filesystem for mount(1M)
60 include ../../Makefile.rootfs
```

```
1
```

```
new/usr/src/lib/libzfs/Makefile.com
```

```
62 LIBS= $(DYNLIB) $(LINTLIB)
64 SRCDIR = .../common
66 INCS += -I$(SRCDIR)
67 INCS += -I../../uts/common/fs/zfs
68 INCS += -I../../common/zfs
69 INCS += -I../../libc/include
71 C99MODE= -xc99=%all
72 C99LMODE= -Xc99=%all
73 LDLIBS += -lc -lm -ldevid -lgen -lnvpair -luutil -lavl -lefi \
74     -ladm -lidmap -ltsol -lmd -lumem -lzfs_core
75 CPPFLAGS += $(INCS) -D_LARGEFILE64_SOURCE=1 -D_REENTRANT
77 CERRWARN += -Wno-switch
78 CERRWARN += -Wno-parentheses
24 CERRWARN += -Wno-uninitialized
25 CERRWARN += -Wno-unused-function
80 SRCS= $(OBJS_COMMON:%.o=$(SRCDIR)/%.c) \
81     $(OBJS_SHARED:%.o=$(SRC)/common/zfs/%.c)
82 $(LINTLIB) := SRCS= $(SRCDIR)/$(LINTSRC)
84 .KEEP_STATE:
86 all: $(LIBS)
88 lint: lintcheck
90 pics/%.o: ../../common/zfs/%.c
91     $(COMPILE.c) -o $@ $<
92         $(POST_PROCESS_O)
94 include ../../Makefile.targ
```

```
2
```

```
new/usr/src/lib/libzfs/common/libzfs_dataset.c
```

```
*****
114386 Wed May 14 12:03:03 2014
new/usr/src/lib/libzfs/common/libzfs_dataset.c
zpool import is braindead
*****
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) 2013, Joyent, Inc. All rights reserved.
25 * Copyright (c) 2013 by Delphix. All rights reserved.
26 * Copyright (c) 2012 DEY Storage Systems, Inc. All rights reserved.
27 * Copyright (c) 2013 Martin Matuska. All rights reserved.
28 * Copyright (c) 2013 Steven Hartland. All rights reserved.
29 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
30 * Copyright 2014 RackTop Systems.
31 #endif /* ! codereview */
32 */
34 #include <ctype.h>
35 #include <errno.h>
36 #include <libintl.h>
37 #include <math.h>
38 #include <stdio.h>
39 #include <stdlib.h>
40 #include <strings.h>
41 #include <unistd.h>
42 #include <stddef.h>
43 #include <zone.h>
44 #include <fcntl.h>
45 #include <sys/mntent.h>
46 #include <sys/mount.h>
47 #include <priv.h>
48 #include <pwd.h>
49 #include <grp.h>
50 #include <stddef.h>
51 #include <ucred.h>
52 #include <idmap.h>
53 #include <aclutils.h>
54 #include <directory.h>
56 #include <sys/dnode.h>
57 #include <sys/spa.h>
58 #include <sys/zap.h>
59 #include <libzfs.h>
61 #include "zfs_namecheck.h"
```

```
1
```

```
new/usr/src/lib/libzfs/common/libzfs_dataset.c
*****
62 #include "zfs_prop.h"
63 #include "libzfs_impl.h"
64 #include "zfs_deleg.h"
66 static int userquota_propname_decode(const char *propname, boolean_t zoned,
67         zfs_userquota_prop_t *typep, char *domain, int domainlen, uint64_t *ridp);
69 /*
70 * Given a single type (not a mask of types), return the type in a human
71 * readable form.
72 */
73 const char *
74 zfs_type_to_name(zfs_type_t type)
75 {
76     switch (type) {
77     case ZFS_TYPE_FILESYSTEM:
78         return (dgettext(TEXT_DOMAIN, "filesystem"));
79     case ZFS_TYPE_SNAPSHOT:
80         return (dgettext(TEXT_DOMAIN, "snapshot"));
81     case ZFS_TYPE_VOLUME:
82         return (dgettext(TEXT_DOMAIN, "volume"));
83     }
85     return (NULL);
86 }
88 /*
89 * Given a path and mask of ZFS types, return a string describing this dataset.
90 * This is used when we fail to open a dataset and we cannot get an exact type.
91 * We guess what the type would have been based on the path and the mask of
92 * acceptable types.
93 */
94 static const char *
95 path_to_str(const char *path, int types)
96 {
97     /*
98     * When given a single type, always report the exact type.
99     */
100    if (types == ZFS_TYPE_SNAPSHOT)
101        return (dgettext(TEXT_DOMAIN, "snapshot"));
102    if (types == ZFS_TYPE_FILESYSTEM)
103        return (dgettext(TEXT_DOMAIN, "filesystem"));
104    if (types == ZFS_TYPE_VOLUME)
105        return (dgettext(TEXT_DOMAIN, "volume"));
107    /*
108     * The user is requesting more than one type of dataset. If this is the
109     * case, consult the path itself. If we're looking for a snapshot, and
110     * a '@' is found, then report it as "snapshot". Otherwise, remove the
111     * snapshot attribute and try again.
112     */
113    if (types & ZFS_TYPE_SNAPSHOT) {
114        if (strchr(path, '@') != NULL)
115            return (dgettext(TEXT_DOMAIN, "snapshot"));
116        return (path_to_str(path, types & ~ZFS_TYPE_SNAPSHOT));
117    }
119    /*
120     * The user has requested either filesystems or volumes.
121     * We have no way of knowing a priori what type this would be, so always
122     * report it as "filesystem" or "volume", our two primitive types.
123     */
124    if (types & ZFS_TYPE_FILESYSTEM)
125        return (dgettext(TEXT_DOMAIN, "filesystem"));
127    assert(types & ZFS_TYPE_VOLUME);
```

```
2
```

```

128         return (dgettext(TEXT_DOMAIN, "volume")));
129     }
130
131     /*
132      * Validate a ZFS path. This is used even before trying to open the dataset, to
133      * provide a more meaningful error message. We call zfs_error_aux() to
134      * explain exactly why the name was not valid.
135     */
136    int
137    zfs_validate_name(libzfs_handle_t *hdl, const char *path, int type,
138                      boolean_t modifying)
139 {
140     namecheck_err_t why;
141     char what;
142
143     (void) zfs_prop_get_table();
144     if (dataset_namecheck(path, &why, &what) != 0) {
145         if (hdl != NULL) {
146             switch (why) {
147                 case NAME_ERR_TOOLONG:
148                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
149                                     "name is too long"));
150                     break;
151
152                 case NAME_ERR.LEADING_SLASH:
153                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
154                                     "leading slash in name"));
155                     break;
156
157                 case NAME_ERR.EMPTY_COMPONENT:
158                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
159                                     "empty component in name"));
160                     break;
161
162                 case NAME_ERR.TRAILING_SLASH:
163                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
164                                     "trailing slash in name"));
165                     break;
166
167                 case NAME_ERRINVALCHAR:
168                     zfs_error_aux(hdl,
169                         dgettext(TEXT_DOMAIN, "invalid character "
170                         "'%c' in name"), what);
171                     break;
172
173                 case NAME_ERRMULTIPLE_AT:
174                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
175                                     "multiple '@' delimiters in name"));
176                     break;
177
178                 case NAME_ERRNOLETTER:
179                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
180                                     "pool doesn't begin with a letter"));
181                     break;
182
183                 case NAME_ERR_RESERVED:
184                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
185                                     "name is reserved"));
186                     break;
187
188                 case NAME_ERR_DISKLIKE:
189                     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
190                                     "reserved disk name"));
191                     break;
192             }
193         }
194     }

```

```

195         return (0);
196     }
197
198     if (!(type & ZFS_TYPE_SNAPSHOT) && strchr(path, '@') != NULL) {
199         if (hdl != NULL)
200             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
201                             "snapshot delimiter '@' in filesystem name"));
202         return (0);
203     }
204
205     if (type == ZFS_TYPE_SNAPSHOT && strchr(path, '@') == NULL) {
206         if (hdl != NULL)
207             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
208                             "missing '@' delimiter in snapshot name"));
209         return (0);
210     }
211
212     if (modifying && strchr(path, '%') != NULL) {
213         if (hdl != NULL)
214             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
215                             "invalid character %c in name"), '%');
216         return (0);
217     }
218
219     return (-1);
220 }
221
222 int
223 zfs_name_valid(const char *name, zfs_type_t type)
224 {
225     if (type == ZFS_TYPE_POOL)
226         return (zpool_name_valid(NULL, B_FALSE, name));
227     return (zfs_validate_name(NULL, name, type, B_FALSE));
228 }
229
230 /*
231  * This function takes the raw DSL properties, and filters out the user-defined
232  * properties into a separate nvlist.
233  */
234 static nvlist_t *
235 process_user_props(zfs_handle_t *zhp, nvlist_t *props)
236 {
237     libzfs_handle_t *hdl = zhp->zfs_hdl;
238     nvpair_t *elem;
239     nvlist_t *propval;
240     nvlist_t *nvl;
241
242     if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
243         (void) no_memory(hdl);
244         return (NULL);
245     }
246
247     elem = NULL;
248     while ((elem = nvlist_next_nvpair(props, elem)) != NULL) {
249         if (!zfs_prop_user(nvpair_name(elem)))
250             continue;
251
252         verify(nvpair_value_nvlist(elem, &propval) == 0);
253         if (nvlist_add_nvlist(nvl, nvpair_name(elem), propval) != 0) {
254             nvlist_free(nvl);
255             (void) no_memory(hdl);
256             return (NULL);
257         }
258     }

```

```

260         return (nvl);
261     }

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

269     if ((zph = zpool_open_canfail(hdl, pool_name)) != NULL) {
270         if (hdl->libzfs_pool_handles != NULL)
271             zph->zpool_next = hdl->libzfs_pool_handles;
272         hdl->libzfs_pool_handles = zph;
273     }
274     return (zph);
275 }

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

283     while ((zph != NULL) &&
284            (strcmp(pool_name, zpool_get_name(zph), len) != 0))
285         zph = zph->zpool_next;
286     return (zph);
287 }

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

301     len = strcspn(zhp->zfs_name, "/@#") + 1;
302     pool_name = zfs_alloc(zhp->zfs_hdl, len);
303     (void) strcpy(pool_name, zhp->zfs_name, len);

305     zph = zpool_find_handle(zhp, pool_name, len);
306     if (zph == NULL)
307         zph = zpool_add_handle(zhp, pool_name);

309     free(pool_name);
310     return (zph);
311 }

313 void
314 zpool_free_handles(libzfs_handle_t *hdl)
315 {
316     zpool_handle_t *next, *zph = hdl->libzfs_pool_handles;

318     while (zph != NULL) {
319         next = zph->zpool_next;
320         zpool_close(zph);
321         zph = next;
322     }
323     hdl->libzfs_pool_handles = NULL;
324 }

```

```

326 /*
327  * Utility function to gather stats (objset and zpl) for the given object.
328 */
329 static int
330 get_stats_ioctl(zfs_handle_t *zhp, zfs_cmd_t *zc)
331 {
332     libzfs_handle_t *hdl = zhp->zfs_hdl;

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

336     while (ioctl(hdl->libzfs_fd, ZFS_IOC_OBJSET_STATS, zc) != 0) {
337         if (errno == ENOMEM) {
338             if (zcmd_expand_dst_nvlist(hdl, zc) != 0)
339                 return (-1);
340         } else {
341             return (-1);
342         }
343     }
344     return (0);
345 }

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

359     if (zcmd_alloc_dst_nvlist(hdl, &zc, 0) != 0)
360         return (-1);

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

364     while (ioctl(hdl->libzfs_fd, ZFS_IOC_OBJSET_RECVD_PROPS, &zc) != 0) {
365         if (errno == ENOMEM) {
366             if (zcmd_expand_dst_nvlist(hdl, &zc) != 0)
367                 return (-1);
368         } else {
369             zcmd_free_nvlists(&zc);
370             return (-1);
371         }
372     }
373 }

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

380     nvlist_free(zhp->zfs_recdv_props);
381     zhp->zfs_recdv_props = recvdprops;

383     return (0);
384 }

386 static int
387 put_stats_zhdl(zfs_handle_t *zhp, zfs_cmd_t *zc)
388 {
389     nvlist_t *allprops, *userprops;

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

```

```

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

```

```

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

```

```

524 {
525     zfs_handle_t *zhp = calloc(sizeof (zfs_handle_t), 1);
526
527     if (zhp == NULL)
528         return (NULL);
529
530     zhp->zfs_hdl = zhp_orig->zfs_hdl;
531     zhp->zpool_hdl = zhp_orig->zpool_hdl;
532     (void) strlcpy(zhp->zfs_name, zhp_orig->zfs_name,
533                     sizeof (zhp->zfs_name));
534     zhp->zfs_type = zhp_orig->zfs_type;
535     zhp->zfs_head_type = zhp_orig->zfs_head_type;
536     zhp->zfs_dmustats = zhp_orig->zfs_dmustats;
537     if (zhp_orig->zfs_props != NULL) {
538         if (nvlist_dup(zhp_orig->zfs_props, &zhp->zfs_props, 0) != 0) {
539             (void) no_memory(zhp->zfs_hdl);
540             zfs_close(zhp);
541             return (NULL);
542         }
543     }
544     if (zhp_orig->zfs_user_props != NULL) {
545         if (nvlist_dup(zhp_orig->zfs_user_props,
546                         &zhp->zfs_user_props, 0) != 0) {
547             (void) no_memory(zhp->zfs_hdl);
548             zfs_close(zhp);
549             return (NULL);
550         }
551     }
552     if (zhp_orig->zfs_recv_props != NULL) {
553         if (nvlist_dup(zhp_orig->zfs_recv_props,
554                         &zhp->zfs_recv_props, 0)) {
555             (void) no_memory(zhp->zfs_hdl);
556             zfs_close(zhp);
557             return (NULL);
558         }
559     }
560     zhp->zfs_mntcheck = zhp_orig->zfs_mntcheck;
561     if (zhp_orig->zfs_mntopts != NULL) {
562         zhp->zfs_mntopts = zfs_strdup(zhp_orig->zfs_hdl,
563                                         zhp_orig->zfs_mntopts);
564     }
565     zhp->zfs_props_table = zhp_orig->zfs_props_table;
566     return (zhp);
567 }

569 boolean_t
570 zfs_bookmark_exists(const char *path)
571 {
572     nvlist_t *bmarks;
573     nvlist_t *props;
574     char fname[ZFS_MAXNAMELEN];
575     char *bmark_name;
576     char *pound;
577     int err;
578     boolean_t rv;

581     (void) strlcpy(fname, path, sizeof (fname));
582     pound = strchr(fname, '#');
583     if (pound == NULL)
584         return (B_FALSE);
585
586     *pound = '\0';
587     bmark_name = pound + 1;
588     props = fnvlist_alloc();
589     err = lzc_get_bookmarks(fname, props, &bmarks);

```

```

590     nvlist_free(props);
591     if (err != 0) {
592         nvlist_free(bmarks);
593         return (B_FALSE);
594     }
595
596     rv = nvlist_exists(bmarks, bmark_name);
597     nvlist_free(bmarks);
598     return (rv);
599 }

600 zfs_handle_t *
601 make_bookmark_handle(zfs_handle_t *parent, const char *path,
602                      nvlist_t *bmark_props)
603 {
604     zfs_handle_t *zhp = calloc(sizeof (zfs_handle_t), 1);
605
606     if (zhp == NULL)
607         return (NULL);
608
609     /* Fill in the name. */
610     zhp->zfs_hdl = parent->zfs_hdl;
611     (void) strlcpy(zhp->zfs_name, path, sizeof (zhp->zfs_name));
612
613     /* Set the property lists. */
614     if (nvlist_dup(bmark_props, &zhp->zfs_props, 0) != 0) {
615         free(zhp);
616         return (NULL);
617     }
618
619     /* Set the types. */
620     zhp->zfs_head_type = parent->zfs_head_type;
621     zhp->zfs_type = ZFS_TYPE_BOOKMARK;
622
623     if ((zhp->zpool_hdl = zpool_handle(zhp)) == NULL) {
624         nvlist_free(zhp->zfs_props);
625         free(zhp);
626         return (NULL);
627     }
628
629     return (zhp);
630 }

631 */

632 /* Opens the given snapshot, filesystem, or volume. The 'types' argument is a mask of acceptable types. The function will print an appropriate error message and return NULL if it can't be opened.
633 */
634 zfs_handle_t *
635 zfs_open(libzfs_handle_t *hdl, const char *path, int types)
636 {
637     zfs_handle_t *zhp;
638     char errbuf[1024];
639
640     (void) sprintf(errbuf, sizeof (errbuf),
641                   dgettext(TEXT_DOMAIN, "cannot open '%s'"), path);
642
643     /*
644      * Validate the name before we even try to open it.
645      */
646     if (!zfs_validate_name(hdl, path, ZFS_TYPE_DATASET, B_FALSE)) {
647         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
648                                     "invalid dataset name"));
649         (void) zfs_error(hdl, EZFS_INVALIDNAME, errbuf);
650         return (NULL);
651     }
652
653     (void) zfs_error(hdl, EZFS_INVALIDNAME, errbuf);
654
655 }

```

```

657     /*
658      * Try to get stats for the dataset, which will tell us if it exists.
659      */
660     errno = 0;
661     if ((zhp = make_dataset_handle(hdl, path)) == NULL) {
662         (void) zfs_standard_error(hdl, errno, errbuf);
663         return (NULL);
664     }
665
666     if (!(types & zhp->zfs_type)) {
667         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
668         zfs_close(zhp);
669         return (NULL);
670     }
671
672     return (zhp);
673 }
674
675 /* Release a ZFS handle. Nothing to do but free the associated memory.
676 */
677 void
678 zfs_close(zfs_handle_t *zhp)
679 {
680     if (zhp->zfs_mntopts)
681         free(zhp->zfs_mntopts);
682     nvlist_free(zhp->zfs_props);
683     nvlist_free(zhp->zfs_user_props);
684     nvlist_free(zhp->zfs_recvd_props);
685     free(zhp);
686 }
687
688 typedef struct mnnttab_node {
689     struct mnnttab mtn_mt;
690     avl_node_t mtn_node;
691 } mnnttab_node_t;
692
693 static int
694 libzfs_mnnttab_cache_compare(const void *arg1, const void *arg2)
695 {
696     const mnnttab_node_t *mtn1 = arg1;
697     const mnnttab_node_t *mtn2 = arg2;
698     int rv;
699
700     rv = strcmp(mtn1->mtn_mt.mnt_special, mtn2->mtn_mt.mnt_special);
701
702     if (rv == 0)
703         return (0);
704     return (rv > 0 ? 1 : -1);
705 }
706
707 void
708 libzfs_mnnttab_init(libzfs_handle_t *hdl)
709 {
710     assert(avl_numnodes(&hdl->libzfs_mnnttab_cache) == 0);
711     avl_create(&hdl->libzfs_mnnttab_cache, libzfs_mnnttab_cache_compare,
712               sizeof (mnnttab_node_t), offsetof(mnnttab_node_t, mtn_node));
713 }
714
715 void
716 libzfs_mnnttab_update(libzfs_handle_t *hdl)
717 {
718     struct mnnttab entry;
719
720     rewind(hdl->libzfs_mnnttab);

```

```

722     while (getmntent(hdl->libzfs_mnnttab, &entry) == 0) {
723         mnnttab_node_t *mtn;
724
725         if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0)
726             continue;
727         mtn = zfs_alloc(hdl, sizeof (mnnttab_node_t));
728         mtn->mtn_mt.mnt_special = zfs_strdup(hdl, entry.mnt_special);
729         mtn->mtn_mt.mnt_mountp = zfs_strdup(hdl, entry.mnt_mountp);
730         mtn->mtn_mt.mnt_fstype = zfs_strdup(hdl, entry.mnt_fstype);
731         mtn->mtn_mt.mnt_mntopts = zfs_strdup(hdl, entry.mnt_mntopts);
732         avl_add(&hdl->libzfs_mnnttab_cache, mtn);
733     }
734 }
735
736 void
737 libzfs_mnnttab_fini(libzfs_handle_t *hdl)
738 {
739     void *cookie = NULL;
740     mnnttab_node_t *mtn;
741
742     while (mtn = avl_destroy_nodes(&hdl->libzfs_mnnttab_cache, &cookie)) {
743         free(mtn->mtn_mt.mnt_special);
744         free(mtn->mtn_mt.mnt_mountp);
745         free(mtn->mtn_mt.mnt_fstype);
746         free(mtn->mtn_mt.mnt_mntopts);
747         free(mtn);
748     }
749     avl_destroy(&hdl->libzfs_mnnttab_cache);
750 }
751
752 void
753 libzfs_mnnttab_cache(libzfs_handle_t *hdl, boolean_t enable)
754 {
755     hdl->libzfs_mnnttab_enable = enable;
756 }
757
758 int
759 libzfs_mnnttab_find(libzfs_handle_t *hdl, const char *fsname,
760                      struct mnnttab *entry)
761 {
762     mnnttab_node_t find;
763     mnnttab_node_t *mtn;
764
765     if (!hdl->libzfs_mnnttab_enable) {
766         struct mnnttab srch = { 0 };
767
768         if (avl_numnodes(&hdl->libzfs_mnnttab_cache))
769             libzfs_mnnttab_fini(hdl);
770         rewind(hdl->libzfs_mnnttab);
771         srch.mnt_special = (char *)fsname;
772         srch.mnt_fstype = MNTTYPE_ZFS;
773         if (getmntany(hdl->libzfs_mnnttab, entry, &srch) == 0)
774             return (0);
775     }
776     else
777         return (ENOENT);
778 }
779
780 if (avl_numnodes(&hdl->libzfs_mnnttab_cache) == 0)
781     libzfs_mnnttab_update(hdl);
782
783 find.mtn_mt.mnt_special = (char *)fsname;
784 mtn = avl_find(&hdl->libzfs_mnnttab_cache, &find, NULL);
785 if (mtn) {
786     *entry = mtn->mtn_mt;
787     return (0);
788 }

```

```

788     return (ENOENT);
789 }

790 void
791 libzfs_mnttab_add(libzfs_handle_t *hdl, const char *special,
792     const char *mountp, const char *mntopts)
793 {
794     mnttab_node_t *mtn;
795
796     if (avl_numnodes(&hdl->libzfs_mnttab_cache) == 0)
797         return;
798     mtn = zfs_alloc(hdl, sizeof (mnttab_node_t));
799     mtn->mtn_mt.mnt_special = zfs_strdup(hdl, special);
800     mtn->mtn_mt.mnt_mountp = zfs_strdup(hdl, mountp);
801     mtn->mtn_mt.mnt_fstype = zfs_strdup(hdl, MNTTYPE_ZFS);
802     mtn->mtn_mt.mnt_mntopts = zfs_strdup(hdl, mntopts);
803     avl_add(&hdl->libzfs_mnttab_cache, mtn);
804 }
805 }

806 void
807 libzfs_mnttab_remove(libzfs_handle_t *hdl, const char *fsname)
808 {
809     mnttab_node_t find;
810     mnttab_node_t *ret;
811
812     find.mtn_mt.mnt_special = (char *)fsname;
813     if (ret = avl_find(&hdl->libzfs_mnttab_cache, (void *)&find, NULL)) {
814         avl_remove(&hdl->libzfs_mnttab_cache, ret);
815         free(ret->mtn_mt.mnt_special);
816         free(ret->mtn_mt.mnt_mountp);
817         free(ret->mtn_mt.mnt_fstype);
818         free(ret->mtn_mt.mnt_mntopts);
819         free(ret);
820     }
821 }
822 }

823 int
824 zfs_spa_version(zfs_handle_t *zhp, int *spa_version)
825 {
826     zpool_handle_t *zpool_handle = zhp->zpool_hdl;
827
828     if (zpool_handle == NULL)
829         return (-1);
830
831     *spa_version = zpool_get_prop_int(zpool_handle,
832         ZPOOL_PROP_VERSION, NULL);
833     return (0);
834 }
835 }

836 */
837 /* The choice of reservation property depends on the SPA version.
838 */
839 static int
840 zfs_which_resv_prop(zfs_handle_t *zhp, zfs_prop_t *resv_prop)
841 {
842     int spa_version;
843
844     if (zfs_spa_version(zhp, &spa_version) < 0)
845         return (-1);
846
847     if (spa_version >= SPA_VERSION_REFRESERVATION)
848         *resv_prop = ZFS_PROP_REFRESERVATION;
849     else
850         *resv_prop = ZFS_PROP_RESERVATION;
851
852     return (0);
853 }
```

```

854 }

855 /*
856  * Given an nvlist of properties to set, validates that they are correct, and
857  * parses any numeric properties (index, boolean, etc) if they are specified as
858  * strings.
859 */
860 nvlist_t *
861 zfs_valid_proplist(libzfs_handle_t *hdl, zfs_type_t type, nvlist_t *nvl,
862     uint64_t zoned, zfs_handle_t *zhp, const char *errbuf)
863 {
864     nvpair_t *elem;
865     uint64_t intval;
866     char *strval;
867     zfs_prop_t prop;
868     nvlist_t *ret;
869     int chosen_normal = -1;
870     int chosen_utf = -1;

871     if (nvlist_alloc(&ret, NV_UNIQUE_NAME, 0) != 0) {
872         (void) no_memory(hdl);
873         return (NULL);
874     }
875
876     /*
877      * Make sure this property is valid and applies to this type.
878      */
879     elem = NULL;
880     while ((elem = nvlist_next_nvpair(nvl, elem)) != NULL) {
881         const char *propname = nvpair_name(elem);

882         prop = zfs_name_to_prop(propname);
883         if (prop == ZPROP_INVAL && zfs_prop_user(propname)) {
884             /*
885              * This is a user property: make sure it's a
886              * string, and that it's less than ZAP_MAXNAMELEN.
887              */
888             if (nvpair_type(elem) != DATA_TYPE_STRING) {
889                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
890                     "'%s' must be a string", propname));
891                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
892                 goto error;
893             }
894
895             if (strlen(nvpair_name(elem)) >= ZAP_MAXNAMELEN) {
896                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
897                     "property name '%s' is too long",
898                     propname));
899                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
900                 goto error;
901             }
902
903             (void) nvpair_value_string(elem, &strval);
904             if (nvlist_add_string(ret, propname, strval) != 0) {
905                 (void) no_memory(hdl);
906                 goto error;
907             }
908             continue;
909         }
910
911         /*
912          * Currently, only user properties can be modified on
913          * snapshots.
914          */
915         if (type == ZFS_TYPE_SNAPSHOT) {
```

```

920         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
921                         "'this property can not be modified for snapshots'"));
922         (void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
923         goto error;
924     }
925
926     if (prop == ZPROP_INVAL && zfs_prop_userquota(propname)) {
927         zfs_userquota_prop_t uqtype;
928         char newpropname[128];
929         char domain[128];
930         uint64_t rid;
931         uint64_t valary[3];
932
933         if (userquota_propname_decode(propname, zoned,
934             &uqtype, domain, sizeof(domain), &rid) != 0) {
935             zfs_error_aux(hdl,
936                         dgettext(TEXT_DOMAIN,
937                             "'$s' has an invalid user/group name",
938                             propname));
939             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
940             goto error;
941         }
942
943         if (uqtype != ZFS_PROP_USERQUOTA &&
944             uqtype != ZFS_PROP_GROUPQUOTA) {
945             zfs_error_aux(hdl,
946                         dgettext(TEXT_DOMAIN, "'%s' is readonly",
947                             propname));
948             (void) zfs_error(hdl, EZFS_PROP_READONLY,
949                             errbuf);
950             goto error;
951         }
952
953         if (nvpair_type(elem) == DATA_TYPE_STRING) {
954             (void) nvpair_value_string(elem, &strval);
955             if (strcmp(strval, "none") == 0) {
956                 intval = 0;
957             } else if (zfs_nicestrtonum(hdl,
958                 strval, &intval) != 0) {
959                 (void) zfs_error(hdl,
960                               EZFS_BADPROP, errbuf);
961                 goto error;
962             }
963         } else if (nvpair_type(elem) ==
964             DATA_TYPE_UINT64) {
965             (void) nvpair_value_uint64(elem, &intval);
966             if (intval == 0) {
967                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
968                     "use 'none' to disable "
969                     "userquota/groupquota"));
970                 goto error;
971             }
972         } else {
973             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
974                         "'%s' must be a number", propname));
975             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
976             goto error;
977         }
978
979         /*
980          * Encode the prop name as
981          * userquota@<hex-rid>-domain, to make it easy
982          * for the kernel to decode.
983          */
984         (void) sprintf(newpropname, sizeof(newpropname),
985                       "%s%llx-%s", zfs_userquota_prop_prefixes[uqtype],

```

```

986             (longlong_t)rid, domain);
987             valary[0] = uqtype;
988             valary[1] = rid;
989             valary[2] = intval;
990             if (nvlist_add_uint64_array(ret, newpropname,
991                 valary, 3) != 0) {
992                 (void) no_memory(hdl);
993                 goto error;
994             }
995             continue;
996         } else if (prop == ZPROP_INVAL && zfs_prop_written(propname)) {
997             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
998                         "'%s' is readonly",
999                         propname));
1000            (void) zfs_error(hdl, EZFS_PROP_READONLY, errbuf);
1001            goto error;
1002        }
1003
1004        if (prop == ZPROP_INVAL) {
1005            zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1006                            "invalid property '%s'", propname));
1007            (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1008            goto error;
1009        }
1010
1011        if (!zfs_prop_valid_for_type(prop, type)) {
1012            zfs_error_aux(hdl,
1013                         dgettext(TEXT_DOMAIN, "'%s' does not "
1014                             "apply to datasets of this type"),
1015                         propname);
1016            (void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
1017            goto error;
1018        }
1019
1020        if (zfs_prop_READONLY(prop) &&
1021             (!zfs_prop_setonce(prop) || zhp != NULL)) {
1022            zfs_error_aux(hdl,
1023                         dgettext(TEXT_DOMAIN, "'%s' is readonly"),
1024                         propname);
1025            (void) zfs_error(hdl, EZFS_PROP_READONLY, errbuf);
1026            goto error;
1027        }
1028
1029        if (zprop_parse_value(hdl, elem, prop, type, ret,
1030             &strval, &intval, errbuf) != 0)
1031            goto error;
1032
1033        /*
1034         * Perform some additional checks for specific properties.
1035         */
1036        switch (prop) {
1037        case ZFS_PROP_VERSION:
1038        {
1039            int version;
1040
1041            if (zhp == NULL)
1042                break;
1043            version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1044            if (intval < version) {
1045                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1046                                "Can not downgrade; already at version %u"),
1047                                version);
1048                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1049                goto error;
1050            }
1051            break;
1052        }

```

```

1053     case ZFS_PROP_RECORDSIZE:
1054     case ZFS_PROP_VOLBLOCKSIZE:
1055         /* must be power of two within SPA_{MIN,MAX}BLOCKSIZE */
1056         if (intval < SPA_MINBLOCKSIZE || intval > SPA_MAXBLOCKSIZE || !ISP2(intval)) {
1057             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1058                                         "'%s' must be power of 2 from %u "
1059                                         "to %uk", propname,
1060                                         (uint_t)SPA_MINBLOCKSIZE,
1061                                         (uint_t)SPA_MAXBLOCKSIZE >> 10));
1062             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1063             goto error;
1064         }
1065         break;
1066
1067     case ZFS_PROP_MSLABEL:
1068     {
1069         /*
1070          * Verify the mslabel string and convert to
1071          * internal hex label string.
1072         */
1073
1074         m_label_t *new_sl;
1075         char *hex = NULL;           /* internal label string */
1076
1077         /* Default value is already OK. */
1078         if (strcasecmp(strval, ZFS_MSLABEL_DEFAULT) == 0)
1079             break;
1080
1081         /* Verify the label can be converted to binary form */
1082         if (((new_sl = m_label_alloc(MAC_LABEL)) == NULL) ||
1083             (str_to_label(strval, &new_sl, MAC_LABEL,
1084                           L_NO_CORRECTION, NULL) == -1)) {
1085             goto badlabel;
1086         }
1087
1088         /* Now translate to hex internal label string */
1089         if (label_to_str(new_sl, &hex, M_INTERNAL,
1090                          DEF_NAMES) != 0) {
1091             if (hex)
1092                 free(hex);
1093             goto badlabel;
1094         }
1095         m_label_free(new_sl);
1096
1097         /* If string is already in internal form, we're done. */
1098         if (strcmp(strval, hex) == 0) {
1099             free(hex);
1100             break;
1101         }
1102
1103         /* Replace the label string with the internal form. */
1104         (void) nvlist_remove(ret, zfs_prop_to_name(prop),
1105                             DATA_TYPE_STRING);
1106         verify(nvlist_add_string(ret, zfs_prop_to_name(prop),
1107                                 hex) == 0);
1108         free(hex);
1109
1110         break;
1111
1112     badlabel:
1113         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1114                                     "invalid mslabel '%s'", strval);
1115         (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1116         m_label_free(new_sl); /* OK if null */
1117

```

```

1118                     goto error;
1119
1120     }
1121
1122     case ZFS_PROP_MOUNTPOINT:
1123     {
1124         namecheck_err_t why;
1125
1126         if (strcmp(strval, ZFS_MOUNTPOINT_NONE) == 0 ||
1127             strcmp(strval, ZFS_MOUNTPOINT_LEGACY) == 0)
1128             break;
1129
1130         if (mountpoint_namecheck(strval, &why)) {
1131             switch (why) {
1132                 case NAME_ERR_LEADING_SLASH:
1133                     zfs_error_aux(hdl,
1134                                   dgettext(TEXT_DOMAIN,
1135                                   "'%s' must be an absolute path, "
1136                                   "'none', or 'legacy'", propname);
1137                     break;
1138                 case NAME_ERR_TOOLONG:
1139                     zfs_error_aux(hdl,
1140                                   dgettext(TEXT_DOMAIN,
1141                                   "component of '%s' is too long",
1142                                   propname));
1143                     break;
1144             }
1145             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1146             goto error;
1147         }
1148     }
1149
1150     /*FALLTHRU*/
1151
1152     case ZFS_PROP_SHARESMB:
1153     case ZFS_PROP_SHARENFS:
1154     /*
1155      * For the mountpoint and sharenfs or sharesmb
1156      * properties, check if it can be set in a
1157      * global/non-global zone based on
1158      * the zoned property value:
1159      *
1160      * -----
1161      *   global zone      non-global zone
1162      *   zoned=on        mountpoint (no)    mountpoint (yes)
1163      *   sharenfs (no)   sharesmb (no)   sharenfs (no)
1164      *   sharesmb (no)   mountpoint (yes)  N/A
1165      *   zoned=off        sharenfs (yes)   sharesmb (yes)
1166      *   sharesmb (yes)   mountpoint (yes)
1167      *   zoned=on        sharenfs (yes)   sharesmb (yes)
1168      *   sharesmb (yes)   mountpoint (yes)
1169      */
1170     if (zoned) {
1171         if (getzoneid() == GLOBAL_ZONEID) {
1172             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1173                                         "'%s' cannot be set on "
1174                                         "dataset in a non-global zone",
1175                                         propname));
1176             (void) zfs_error(hdl, EZFS_ZONED,
1177                               errbuf);
1178             goto error;
1179         } else if (prop == ZFS_PROP_SHARENFS ||
1180                    prop == ZFS_PROP_SHARESMB) {
1181             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1182                                         "'%s' cannot be set in "
1183                                         "a non-global zone"), propname);
1184

```

new/usr/src/lib/libzfs/common/libzfs\_dataset.c

19

```

1184                                     (void) zfs_error(hdl, EZFS_ZONED,
1185                                         errbuf);
1186                                     goto error;
1187                                 }
1188 } else if (getzoneid() != GLOBAL_ZONEID) {
1189 /* 
1190 * If zoned property is 'off', this must be in
1191 * a global zone. If not, something is wrong.
1192 */
1193 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1194     "'%s' cannot be set while dataset "
1195     "'zoned' property is set"), propname);
1196 (void) zfs_error(hdl, EZFS_ZONED, errbuf);
1197 goto error;
1198 }

1199 /*
1200 * At this point, it is legitimate to set the
1201 * property. Now we want to make sure that the
1202 * property value is valid if it is sharenfs.
1203 */
1204 if ((prop == ZFS_PROP_SHARENFS ||
1205     prop == ZFS_PROP_SHARESMB) &&
1206     strcmp(strval, "on") != 0 &&
1207     strcmp(strval, "off") != 0) {
1208     zfs_share_proto_t proto;

1209     if (prop == ZFS_PROP_SHARESMB)
1210         proto = PROTO_SMB;
1211     else
1212         proto = PROTO_NFS;

1213 /*
1214 * Must be an valid sharing protocol
1215 * option string so init the libshare
1216 * in order to enable the parser and
1217 * then parse the options. We use the
1218 * control API since we don't care about
1219 * the current configuration and don't
1220 * want the overhead of loading it
1221 * until we actually do something.
1222 */
1223
1224 if (zfs_init_libshare(hdl,
1225     SA_INIT_CONTROL_API) != SA_OK) {
1226 /* 
1227 * An error occurred so we can't do
1228 * anything
1229 */
1230 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1231     "'%s' cannot be set: problem "
1232     "in share initialization"),
1233     propname);
1234 (void) zfs_error(hdl, EZFS_BADPROP,
1235     errbuf);
1236 goto error;
1237 }

1238 if (zfs_parse_options(strval, proto) != SA_OK) {
1239 /* 
1240 * There was an error in parsing so
1241 * deal with it by issuing an error
1242 * message and leaving after
1243 * uninitialized the the libshare
1244 * interface.
1245 */
1246
1247
1248
1249

```

new/usr/src/lib/libzfs/common/libzfs\_dataset.c

```

zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
    "'%s' cannot be set to invalid "
    "options"), propname);
(void) zfs_error(hdl, EZFS_BADPROP,
    errbuf);
zfs_uninit_libshare(hdl);
goto error;

init_libshare(hdl);

int)intval;
E:
= (int)intval;

ing volumes, we have some additional

OLUME && zhp != NULL) {
ze = zfs_prop_get_int(zhp,
OLSIZE);
size = zfs_prop_get_int(zhp,
OLBLOCKSIZE);

[
RESERVATION:
REFRESERVATION:
tval > volsize) {
    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
        "'%s' is greater than current "
        "volume size"), propname);
(void) zfs_error(hdl, EZFS_BADPROP,
    errbuf);
goto error;

VOLSIZE:
tval % blocksize != 0) {
    zfs_nicenum(blocksize, buf,
        sizeof (buf));
zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
    "'%s' must be a multiple of "
    "volume block size (%s"),
    propname, buf);
(void) zfs_error(hdl, EZFS_BADPROP,
    errbuf);
goto error;

tval == 0) {
    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
        "'%s' cannot be zero"),
        propname);
(void) zfs_error(hdl, EZFS_BADPROP,
    errbuf);
goto error;

```

new/usr/src/lib/libzfs/common/libzfs\_dataset.c

21

```

1316     }
1317 }
1318 }

1320 /*
1321 * If normalization was chosen, but no UTF8 choice was made,
1322 * enforce rejection of non-UTF8 names.
1323 *
1324 * If normalization was chosen, but rejecting non-UTF8 names
1325 * was explicitly not chosen, it is an error.
1326 */
1327 if (chosen_normal > 0 && chosen_utf < 0) {
1328     if (nvlist_add_uint64(ret,
1329         zfs_prop_to_name(ZFS_PROP_UTF8ONLY), 1) != 0) {
1330         (void) no_memory(hdl);
1331         goto error;
1332     }
1333 } else if (chosen_normal > 0 && chosen_utf == 0) {
1334     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1335         "'%s' must be set 'on' if normalization chosen"),
1336         zfs_prop_to_name(ZFS_PROP_UTF8ONLY));
1337     (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1338     goto error;
1339 }
1340 return (ret);

1342 error:
1343 nvlist_free(ret);
1344 return (NULL);
1345 }

1347 int
1348 zfs_add_synthetic_resv(zfs_handle_t *zhp, nvlist_t *nvl)
1349 {
1350     uint64_t old_volsize;
1351     uint64_t new_volsize;
1352     uint64_t old_reservation;
1353     uint64_t new_reservation;
1354     zfs_prop_t resv_prop;
1355     nvlist_t *props;

1357 /*
1358 * If this is an existing volume, and someone is setting the volsize,
1359 * make sure that it matches the reservation, or add it if necessary.
1360 */
1361 old_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
1362 if (zfs_which_resv_prop(zhp, &resv_prop) < 0)
1363     return (-1);
1364 old_reservation = zfs_prop_get_int(zhp, resv_prop);

1366 props = fnvlist_alloc();
1367 fnvlist_add_uint64(props, zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
1368                     zfs_prop_get_int(zhp, ZFS_PROP_VOLBLOCKSIZE));

1370 if ((zvol_volsize_to_reservation(old_volsize, props) !=
1371     old_reservation) || nvlist_exists(nvl,
1372         zfs_prop_to_name(resv_prop))) {
1373     fnvlist_free(props);
1374     return (0);
1375 }
1376 if (nvlist_lookup_uint64(nvl, zfs_prop_to_name(ZFS_PROP_VOLSIZE),
1377     &new_volsize) != 0) {
1378     fnvlist_free(props);
1379     return (-1);
1380 }
1381 new_reservation = zvol_volsize_to_reservation(new_volsize, props);

```

new/usr/src/lib/libzfs/common/libzfs\_dataset.c

```

1382     fnvlist_free(props);
1384
1385     if (nvlist_add_uint64(nvl, zfs_prop_to_name(resv_prop),
1386                           new_reservation) != 0) {
1387         (void) no_memory(zhp->zfs_hdl);
1388         return (-1);
1389     }
1390     return (1);
1391 }
1392 void
1393 zfs_setprop_error(libzfs_handle_t *hdl, zfs_prop_t prop, int err,
1394                     char *errbuf)
1395 {
1396     switch (err) {
1398
1399     case ENOSPC:
1400         /*
1401          * For quotas and reservations, ENOSPC indicates
1402          * something different; setting a quota or reservation
1403          * doesn't use any disk space.
1404         */
1405         switch (prop) {
1406             case ZFS_PROP_QUOTA:
1407                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1408                                     "size is less than current used or "
1409                                     "reserved space"));
1410                 (void) zfs_error(hdl, EZFS_PROPSPACE, errbuf);
1411                 break;
1412
1413             case ZFS_PROP_RESERVATION:
1414             case ZFS_PROP_REFRESERVATION:
1415                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1416                                     "size is greater than available space"));
1417                 (void) zfs_error(hdl, EZFS_PROPSPACE, errbuf);
1418                 break;
1419
1420             default:
1421                 (void) zfs_standard_error(hdl, err, errbuf);
1422                 break;
1423         }
1424         break;
1425
1426     case EBUSY:
1427         (void) zfs_standard_error(hdl, EBUSY, errbuf);
1428         break;
1429
1430     case EROFS:
1431         (void) zfs_error(hdl, EZFS_DSREADONLY, errbuf);
1432         break;
1433
1434     case ENOTSUP:
1435         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1436                                     "pool and/or dataset must be upgraded to set this "
1437                                     "property or value"));
1438         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
1439         break;
1440
1441     case ERANGE:
1442         if (prop == ZFS_PROP_COMPRESSION) {
1443             (void) zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1444                                         "property setting is not allowed on "
1445                                         "bootable datasets"));
1446             (void) zfs_error(hdl, EZFS_NOTSUP, errbuf);
1447         } else {

```

new/usr/src/lib/libzfs/common/libzfs\_dataset.c

23

```

1448         }
1449     }
1450 }

1452     case EINVAL:
1453         if (prop == ZPROP_INVAL) {
1454             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1455         } else {
1456             (void) zfs_standard_error(hdl, err, errbuf);
1457         }
1458     break;

1460     case EOVERRLOW:
1461         /*
1462          * This platform can't address a volume this big.
1463         */
1464 #ifdef _ILP32
1465         if (prop == ZFS_PROP_VOLSIZE) {
1466             (void) zfs_error(hdl, EZFS_VOLTOOBIG, errbuf);
1467             break;
1468         }
1469 #endif
1470         /* FALLTHROUGH */
1471     default:
1472         (void) zfs_standard_error(hdl, err, errbuf);
1473     }
1474 }

1476 /*
1477  * Given a property name and value, set the property for the given dataset.
1478 */
1479 int
1480 zfs_prop_set(zfs_handle_t *zhp, const char *propname, const char *propval)
1481 {
1482     zfs_cmd_t zc = { 0 };
1483     int ret = -1;
1484     prop_changelist_t *cl = NULL;
1485     char errbuf[1024];
1486     libzfs_handle_t *hdl = zhp->zfs_hdl;
1487     nvlist_t *nvl = NULL, *realprops;
1488     zfs_prop_t prop;
1489     boolean_t do_prefix = B_TRUE;
1490     int added_resv = 0;
1491     int added_resv;

1492     (void) snprintf(errbuf, sizeof (errbuf),
1493                     dgettext(TEXT_DOMAIN, "cannot set property for '%s'"),
1494                     zhp->zfs_name);

1496     if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0 ||
1497         nvlist_add_string(nvl, propname, propval) != 0) {
1498         (void) no_memory(hdl);
1499         goto error;
1500     }

1502     if ((realprops = zfs_valid_proplist(hdl, zhp->zfs_type, nvl,
1503                                         ZFS_PROP_ZONED, zhp, errbuf)) == NULL)
1504         goto error;

1506     nvlist_free(nvl);
1507     nvl = realprops;

1509     prop = zfs_name_to_prop(propname);

1511     if (prop == ZFS_PROP_VOLSIZE) {
1512         if ((added_resv = zfs_add_synthetic_resv(zhp, nvl)) == -1)
1513             goto error;
1514     }

```

new/usr/src/lib/libzfs/common/libzfs\_dataset.c

```

1513                         goto error;
1514
1515         }
1516
1517         if ((cl = changelist_gather(zhp, prop, 0, 0)) == NULL)
1518             goto error;
1519
1520         if (prop == ZFS_PROP_MOUNTPOINT && changelist_haszonedchild(cl)) {
1521             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1522                                     "child dataset with inherited mountpoint is used "
1523                                     "in a non-global zone"));
1524             ret = zfs_error(hdl, EZFS_ZONED, errbuf);
1525             goto error;
1526         }
1527
1528         /*
1529          * We don't want to unmount & remount the dataset when changing
1530          * its canmount property to 'on' or 'noauto'.  We only use
1531          * the changelist logic to unmount when setting canmount=off.
1532         */
1533         if (prop == ZFS_PROP_CANMOUNT) {
1534             uint64_t idx;
1535             int err = zprop_string_to_index(prop, propval, &idx,
1536                                             ZFS_TYPE_DATASET);
1537             if (err == 0 && idx != ZFS_CANMOUNT_OFF)
1538                 do_prefix = B_FALSE;
1539         }
1540
1541         if (do_prefix && (ret = changelist_prefix(cl)) != 0)
1542             goto error;
1543
1544         /*
1545          * Execute the corresponding ioctl() to set this property.
1546         */
1547         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
1548
1549         if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1550             goto error;
1551
1552         ret = zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
1553
1554         if (ret != 0) {
1555             zfs_setprop_error(hdl, prop, errno, errbuf);
1556             if (added_resv && errno == ENOSPC) {
1557                 /* clean up the volsize property we tried to set */
1558                 uint64_t old_volsize = zfs_prop_get_int(zhp,
1559                                              ZFS_PROP_VOLSIZE);
1560                 nvlist_free(nvl);
1561                 zcmd_free_nvlists(&zc);
1562                 if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
1563                     goto error;
1564                 if (nvlist_add_uint64(nvl,
1565                                       zfs_prop_to_name(ZFS_PROP_VOLSIZE),
1566                                       old_volsize) != 0)
1567                     goto error;
1568                 if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1569                     goto error;
1570                 (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
1571             }
1572         } else {
1573             if (do_prefix)
1574                 ret = changelist_postfix(cl);
1575
1576             /*
1577              * Refresh the statistics so the new property value
1578              * is reflected.
1579             */

```

```

1579         if (ret == 0)
1580             (void) get_stats(zhp);
1581     }
1583 error:
1584     nvlist_free(nvl);
1585     zcmd_free_nvlists(&zc);
1586     if (cl)
1587         changelist_free(cl);
1588     return (ret);
1589 }
_____unchanged_portion_omitted_
3002 /*
3003  * Creates non-existing ancestors of the given path.
3004  */
3005 int
3006 zfs_create_ancestors(libzfs_handle_t *hdl, const char *path)
3007 {
3008     int prefix;
3009     char *path_copy;
3010     int rc = 0;
3011     int rc;
3012
3013     if (check_parents(hdl, path, NULL, B_TRUE, &prefix) != 0)
3014         return (-1);
3015
3016     if ((path_copy = strdup(path)) != NULL) {
3017         rc = create_parents(hdl, path_copy, prefix);
3018         free(path_copy);
3019     }
3020     if (path_copy == NULL || rc != 0)
3021         return (-1);
3022
3023 }
_____unchanged_portion_omitted_
3619 /*
3620  * Given a dataset, rollback to a specific snapshot, discarding any
3621  * data changes since then and making it the active dataset.
3622  *
3623  * Any snapshots and bookmarks more recent than the target are
3624  * destroyed, along with their dependents (i.e. clones).
3625  */
3626 int
3627 zfs_rollback(zfs_handle_t *zhp, zfs_handle_t *snap, boolean_t force)
3628 {
3629     rollback_data_t cb = { 0 };
3630     int err;
3631     boolean_t restore_resv = 0;
3632     uint64_t old_volsize = 0, new_volsize;
3633     uint64_t old_volsize, new_volsize;
3634     zfs_prop_t resv_prop;
3635
3636     assert(zhp->zfs_type == ZFS_TYPE_FILESYSTEM ||
3637           zhp->zfs_type == ZFS_TYPE_VOLUME);
3638
3639     /*
3640      * Destroy all recent snapshots and their dependents.
3641      */
3642     cb.cb_force = force;
3643     cb.cb_target = snap->zfs_name;
3644     cb.cb_create = zfs_prop_get_int(snap, ZFS_PROP_CREATETXG);
3645     (void) zfs_iter_snapshots(zhp, rollback_destroy, &cb);
3646     (void) zfs_iter_bookmarks(zhp, rollback_destroy, &cb);

```

```

3647     if (cb.cb_error)
3648         return (-1);
3649
3650     /*
3651      * Now that we have verified that the snapshot is the latest,
3652      * rollback to the given snapshot.
3653      */
3654
3655     if (zhp->zfs_type == ZFS_TYPE_VOLUME) {
3656         if (zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE) < 0)
3657             return (-1);
3658         old_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
3659         restore_resv =
3660             (old_volsize == zfs_prop_get_int(zhp, resv_prop));
3661     }
3662
3663     /*
3664      * We rely on zfs_iter_children() to verify that there are no
3665      * newer snapshots for the given dataset. Therefore, we can
3666      * simply pass the name on to the ioctl() call. There is still
3667      * an unlikely race condition where the user has taken a
3668      * snapshot since we verified that this was the most recent.
3669      */
3670     err = lzc_rollback(zhp->zfs_name, NULL, 0);
3671     if (err != 0) {
3672         (void) zfs_standard_error_fmt(zhp->zfs_hdl, errno,
3673                                       dgettext(TEXT_DOMAIN, "cannot rollback '%s'"),
3674                                       zhp->zfs_name);
3675         return (err);
3676     }
3677
3678     /*
3679      * For volumes, if the pre-rollback volsize matched the pre-
3680      * rollback reservation and the volsize has changed then set
3681      * the reservation property to the post-rollback volsize.
3682      * Make a new handle since the rollback closed the dataset.
3683      */
3684     if ((zhp->zfs_type == ZFS_TYPE_VOLUME) &&
3685         (zhp = make_dataset_handle(zhp->zfs_hdl, zhp->zfs_name))) {
3686         if (restore_resv) {
3687             new_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
3688             if (old_volsize != new_volsize)
3689                 err = zfs_prop_set_int(zhp, resv_prop,
3690                                       new_volsize);
3691         }
3692         zfs_close(zhp);
3693     }
3694     return (err);
3695 }
3696
3697 /*
3698  * Renames the given dataset.
3699  */
3700 int
3701 zfs_rename(zfs_handle_t *zhp, const char *target, boolean_t recursive,
3702             boolean_t force_unmount)
3703 {
3704     int ret = -1;
3705     int ret;
3706     zfs_cmd_t zc = { 0 };
3707     char *delim;
3708     prop_changelist_t *cl = NULL;
3709     zfs_handle_t *zhrp = NULL;
3710     char *parentname = NULL;
3711     char parent[ZFS_MAXNAMELEN];

```

```

3711     libzfs_handle_t *hdl = zhp->zfs_hdl;
3712     char errbuf[1024];
3713
3714     /* if we have the same exact name, just return success */
3715     if (strcmp(zhp->zfs_name, target) == 0)
3716         return (0);
3717
3718     (void) sprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3719             "cannot rename to '%s'"), target);
3720
3721     /*
3722      * Make sure the target name is valid
3723
3724      if (zhp->zfs_type == ZFS_TYPE_SNAPSHOT) {
3725          if ((strchr(target, '@') == NULL) ||
3726              *target == '@') {
3727              /*
3728                  * Snapshot target name is abbreviated,
3729                  * reconstruct full dataset name
3730                  */
3731              (void) strlcpy(parent, zhp->zfs_name,
3732                             sizeof (parent));
3733              delim = strchr(parent, '@');
3734              if (strchr(target, '@') == NULL)
3735                  *(++delim) = '\0';
3736              else
3737                  *delim = '\0';
3738              (void) strlcat(parent, target, sizeof (parent));
3739              target = parent;
3740      } else {
3741          /*
3742              * Make sure we're renaming within the same dataset.
3743              */
3744          delim = strchr(target, '@');
3745          if (strncmp(zhp->zfs_name, target, delim - target)
3746              != 0 || zhp->zfs_name[delim - target] != '@') {
3747              zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3748                  "snapshots must be part of same "
3749                  "dataset"));
3750              return (zfs_error(hdl, EZFS_CROSSTARGET,
3751                               errbuf));
3752          }
3753          if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
3754              return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3755      } else {
3756          if (recursive) {
3757              zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3758                  "recursive rename must be a snapshot"));
3759              return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
3760      }
3761
3762      if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
3763          return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3764
3765      /* validate parents */
3766      if (check_parents(hdl, target, NULL, B_FALSE, NULL) != 0)
3767          return (-1);
3768
3769      /* make sure we're in the same pool */
3770      verify((delim = strchr(target, '/')) != NULL);
3771      if (strncmp(zhp->zfs_name, target, delim - target) != 0 ||
3772          zhp->zfs_name[delim - target] != '/') {
3773          zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3774                  "datasets must be within same pool"));
3775          return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));
3776

```

```

3777
3778
3779     /*
3780      * new name cannot be a child of the current dataset name */
3781     if (is_descendant(zhp->zfs_name, target)) {
3782         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3783             "New dataset name cannot be a descendant of "
3784             "current dataset name"));
3785         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3786     }
3787
3788     (void) sprintf(errbuf, sizeof (errbuf),
3789                 dgettext(TEXT_DOMAIN, "cannot rename '%s'", zhp->zfs_name));
3790
3791     if (getzoneid() == GLOBAL_ZONEID &&
3792         zfs_prop_get_int(zhp, ZFS_PROP_ZONED)) {
3793         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3794             "dataset is used in a non-global zone"));
3795         return (zfs_error(hdl, EZFS_ZONED, errbuf));
3796     }
3797
3798     if (recursive) {
3799
3800         parentname = zfs_strdup(zhp->zfs_hdl, zhp->zfs_name);
3801         if (parentname == NULL) {
3802             ret = -1;
3803             goto error;
3804         }
3805         delim = strchr(parentname, '@');
3806         *delim = '\0';
3807         zhrp = zfs_open(zhp->zfs_hdl, parentname, ZFS_TYPE_DATASET);
3808         if (zhrp == NULL) {
3809             ret = -1;
3810             goto error;
3811         }
3812
3813     } else {
3814         if ((cl = changelist_gather(zhp, ZFS_PROP_NAME, 0,
3815             force_unmount ? MS_FORCE : 0)) == NULL)
3816             return (-1);
3817
3818         if (changelist_haszonedchild(cl)) {
3819             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3820                 "child dataset with inherited mountpoint is used "
3821                 "in a non-global zone"));
3822             (void) zfs_error(hdl, EZFS_ZONED, errbuf);
3823             goto error;
3824         }
3825
3826         if ((ret = changelist_prefix(cl)) != 0)
3827             goto error;
3828     }
3829
3830     if (ZFS_IS_VOLUME(zhp))
3831         zc.zc_objset_type = DMU_OST_ZVOL;
3832     else
3833         zc.zc_objset_type = DMU_OST_ZFS;
3834
3835     (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
3836     (void) strlcpy(zc.zc_value, target, sizeof (zc.zc_value));
3837
3838     zc.zc_cookie = recursive;
3839
3840     if ((ret = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_RENAME, &zc)) != 0) {
3841         /*
3842             * if it was recursive, the one that actually failed will

```

```
3843     * be in zc.zc_name
3844     */
3845     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3846         "cannot rename '%s'), zc.zc_name);
3847
3848     if (recursive && errno == EEXIST) {
3849         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3850             "a child dataset already has a snapshot "
3851             "with the new name"));
3852     } else {
3853         (void) zfs_error(hdl, EZFS_EXISTS, errbuf);
3854     }
3855
3856     /*
3857     * On failure, we still want to remount any filesystems that
3858     * were previously mounted, so we don't alter the system state.
3859     */
3860     if (!recursive)
3861         (void) changelist_postfix(cl);
3862     } else {
3863         if (!recursive) {
3864             changelist_rename(cl, zfs_get_name(zhp), target);
3865             ret = changelist_postfix(cl);
3866         }
3867     }
3868 }
3869
3870 error:
3871     if (parentname) {
3872         free(parentname);
3873     }
3874     if (zhrp) {
3875         zfs_close(zhrp);
3876     }
3877     if (cl) {
3878         changelist_free(cl);
3879     }
3880     return (ret);
3881 }
```

unchanged\_portion\_omitted

new/usr/src/lib/libzfs/common/libzfs\_import.c

1

```

*****
43779 Wed May 14 12:03:03 2014
new/usr/src/lib/libzfs/common/libzfs_import.c
zpool import is braindead
*****



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 by 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 2014 Nexenta Systems, Inc. All rights reserved.
25 * Copyright 2014 RackTop Systems.
26 #endif /* ! codereview */
27 */

29 /*
30 * Pool import support functions.
31 *
32 * To import a pool, we rely on reading the configuration information from the
33 * ZFS label of each device. If we successfully read the label, then we
34 * organize the configuration information in the following hierarchy:
35 *
36 *      pool guid -> toplevel vdev guid -> label txg
37 *
38 * Duplicate entries matching this same tuple will be discarded. Once we have
39 * examined every device, we pick the best label txg config for each toplevel
40 * vdev. We then arrange these toplevel vdevs into a complete pool config, and
41 * update any paths that have changed. Finally, we attempt to import the pool
42 * using our derived config, and record the results.
43 */

45 #include <ctype.h>
46 #include <devid.h>
47 #include <dirent.h>
48 #include <errno.h>
49 #include <libintl.h>
50 #include <stddef.h>
51 #include <stdlib.h>
52 #include <string.h>
53 #include <sys/stat.h>
54 #include <unistd.h>
55 #include <fcntl.h>
56 #include <sys/vtoc.h>
57 #include <sys/dktp/fdisk.h>
58 #include <sys/efi_partition.h>
59 #include <thread_pool.h>

61 #include <sys/vdev_impl.h>

```

new/usr/src/lib/libzfs/common/libzfs\_import.c

```

63 #include "libzfs.h"
64 #include "libzfs_impl.h"

66 /*
67  * Intermediate structures used to gather configuration information.
68 */
69 typedef struct config_entry {
70     uint64_t                         ce_txg;
71     nvlist_t                          *ce_config;
72     struct config_entry              *ce_next;
73 } config_entry_t;

75 typedef struct vdev_entry {
76     uint64_t                         ve_guid;
77     config_entry_t                   *ve_configs;
78     struct vdev_entry              *ve_next;
79 } vdev_entry_t;

81 typedef struct pool_entry {
82     uint64_t                         pe_guid;
83     vdev_entry_t                     *pe_vdevs;
84     struct pool_entry              *pe_next;
85 } pool_entry_t;

87 typedef struct name_entry {
88     char                            *ne_name;
89     uint64_t                         ne_guid;
90     struct name_entry              *ne_next;
91 } name_entry_t;

93 typedef struct pool_list {
94     pool_entry_t                    *pools;
95     name_entry_t                   *names;
96 } pool_list_t;

98 static char *
99 get_devid(const char *path)
100 {
101     int fd;
102     ddi_devid_t devid;
103     char *minor, *ret;

105     if ((fd = open(path, O_RDONLY)) < 0)
106         return (NULL);

108     minor = NULL;
109     ret = NULL;
110     if (devid_get(fd, &devid) == 0) {
111         if (devid_get_minor_name(fd, &minor) == 0)
112             ret = devid_str_encode(devid, minor);
113         if (minor != NULL)
114             devid_str_free(minor);
115         devid_free(devid);
116     }
117     (void) close(fd);

119     return (ret);
120 }

123 /*
124  * Go through and fix up any path and/or devid information for the given vdev
125  * configuration.
126 */
127 static int

```

```

128 fix_paths(nvlist_t *nv, name_entry_t *names)
129 {
130     nvlist_t **child;
131     uint_t c, children;
132     uint64_t guid;
133     name_entry_t *ne, *best;
134     char *path, *devid;
135     int matched;
136
137     if (nvlist_lookup_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN,
138         &child, &children) == 0) {
139         for (c = 0; c < children; c++)
140             if (fix_paths(child[c], names) != 0)
141                 return (-1);
142     }
143
144     /*
145      * This is a leaf (file or disk) vdev. In either case, go through
146      * the name list and see if we find a matching guid. If so, replace
147      * the path and see if we can calculate a new devid.
148      *
149      * There may be multiple names associated with a particular guid, in
150      * which case we have overlapping slices or multiple paths to the same
151      * disk. If this is the case, then we want to pick the path that is
152      * the most similar to the original, where "most similar" is the number
153      * of matching characters starting from the end of the path. This will
154      * preserve slice numbers even if the disks have been reorganized, and
155      * will also catch preferred disk names if multiple paths exist.
156      */
157     verify(nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) == 0);
158     if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PATH, &path) != 0)
159         path = NULL;
160
161     matched = 0;
162     best = NULL;
163     for (ne = names; ne != NULL; ne = ne->ne_next) {
164         if (ne->ne_guid == guid) {
165             const char *src, *dst;
166             int count;
167
168             if (path == NULL) {
169                 best = ne;
170                 break;
171             }
172
173             src = ne->ne_name + strlen(ne->ne_name) - 1;
174             dst = path + strlen(path) - 1;
175             for (count = 0; src >= ne->ne_name && dst >= path;
176                  src--, dst--, count++)
177                 if (*src != *dst)
178                     break;
179
180             /*
181              * At this point, 'count' is the number of characters
182              * matched from the end.
183              */
184             if (count > matched || best == NULL) {
185                 best = ne;
186                 matched = count;
187             }
188         }
189     }
190
191     if (best == NULL)
192         return (0);

```

```

195     if (nvlist_add_string(nv, ZPOOL_CONFIG_PATH, best->ne_name) != 0)
196         return (-1);
197
198     if ((devid = get_devid(best->ne_name)) == NULL) {
199         (void) nvlist_remove_all(nv, ZPOOL_CONFIG_DEVID);
200     } else {
201         if (nvlist_add_string(nv, ZPOOL_CONFIG_DEVID, devid) != 0)
202             return (-1);
203         devid_str_free(devid);
204     }
205
206     return (0);
207 }
208
209 /*
210  * Add the given configuration to the list of known devices.
211  */
212 static int
213 add_config(libzfs_handle_t *hdl, pool_list_t *pl, const char *path,
214             nvlist_t *config)
215 {
216     uint64_t pool_guid, vdev_guid, top_guid, txg, state;
217     pool_entry_t *pe;
218     vdev_entry_t *ve;
219     config_entry_t *ce;
220     name_entry_t *ne;
221
222     /*
223      * If this is a hot spare not currently in use or level 2 cache
224      * device, add it to the list of names to translate, but don't do
225      * anything else.
226      */
227     if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_STATE,
228         &state) == 0 &&
229         (state == POOL_STATE_SPARE || state == POOL_STATE_L2CACHE) &&
230         nvlist_lookup_uint64(config, ZPOOL_CONFIG_GUID, &vdev_guid) == 0) {
231         if ((ne = zfs_alloc(hdl, sizeof (name_entry_t))) == NULL)
232             return (-1);
233
234         if ((ne->ne_name = zfs_strdup(hdl, path)) == NULL) {
235             free(ne);
236             return (-1);
237         }
238         ne->ne_guid = vdev_guid;
239         ne->ne_next = pl->names;
240         pl->names = ne;
241         return (0);
242     }
243
244     /*
245      * If we have a valid config but cannot read any of these fields, then
246      * it means we have a half-initialized label. In vdev_label_init()
247      * we write a label with txg == 0 so that we can identify the device
248      * in case the user refers to the same disk later on. If we fail to
249      * create the pool, we'll be left with a label in this state
250      * which should not be considered part of a valid pool.
251      */
252     if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID,
253         &pool_guid) != 0 ||
254         nvlist_lookup_uint64(config, ZPOOL_CONFIG_GUID,
255         &vdev_guid) != 0 ||
256         nvlist_lookup_uint64(config, ZPOOL_CONFIG_TOP_GUID,
257         &top_guid) != 0 ||
258         nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_TXG,
259         &txg) != 0 || txg == 0) {

```

```

260             nvlist_free(config);
261             return (0);
262     }
263
264     /*
265      * First, see if we know about this pool.  If not, then add it to the
266      * list of known pools.
267      */
268     for (pe = pl->pools; pe != NULL; pe = pe->pe_next) {
269         if (pe->pe_guid == pool_guid)
270             break;
271     }
272
273     if (pe == NULL) {
274         if ((pe = zfs_alloc(hdl, sizeof (pool_entry_t))) == NULL) {
275             nvlist_free(config);
276             return (-1);
277         }
278         pe->pe_guid = pool_guid;
279         pe->pe_next = pl->pools;
280         pl->pools = pe;
281     }
282
283     /*
284      * Second, see if we know about this toplevel vdev.  Add it if its
285      * missing.
286      */
287     for (ve = pe->pe_vdevs; ve != NULL; ve = ve->ve_next) {
288         if (ve->ve_guid == top_guid)
289             break;
290     }
291
292     if (ve == NULL) {
293         if ((ve = zfs_alloc(hdl, sizeof (vdev_entry_t))) == NULL) {
294             nvlist_free(config);
295             return (-1);
296         }
297         ve->ve_guid = top_guid;
298         ve->ve_next = pe->pe_vdevs;
299         pe->pe_vdevs = ve;
300     }
301
302     /*
303      * Third, see if we have a config with a matching transaction group.  If
304      * so, then we do nothing.  Otherwise, add it to the list of known
305      * configs.
306      */
307     for (ce = ve->ve_configs; ce != NULL; ce = ce->ce_next) {
308         if (ce->ce_txg == txg)
309             break;
310     }
311
312     if (ce == NULL) {
313         if ((ce = zfs_alloc(hdl, sizeof (config_entry_t))) == NULL) {
314             nvlist_free(config);
315             return (-1);
316         }
317         ce->ce_txg = txg;
318         ce->ce_config = config;
319         ce->ce_next = ve->ve_configs;
320         ve->ve_configs = ce;
321     } else {
322         nvlist_free(config);
323     }
324
325     /*

```

```

326         /*
327          * At this point we've successfully added our config to the list of
328          * known configs.  The last thing to do is add the vdev guid -> path
329          * mappings so that we can fix up the configuration as necessary before
330          * doing the import.
331         */
332         if ((ne = zfs_alloc(hdl, sizeof (name_entry_t))) == NULL)
333             return (-1);
334
335         if ((ne->ne_name = zfs_strdup(hdl, path)) == NULL) {
336             free(ne);
337             return (-1);
338         }
339
340         ne->ne_guid = vdev_guid;
341         ne->ne_next = pl->names;
342         pl->names = ne;
343
344     }
345
346     /*
347      * Returns true if the named pool matches the given GUID.
348      */
349     static int
350     pool_active(libzfs_handle_t *hdl, const char *name, uint64_t guid,
351                 boolean_t *isactive)
352 {
353     zpool_handle_t *zhp;
354     uint64_t theguid;
355
356     if (zpool_open_silent(hdl, name, &zhp) != 0)
357         return (-1);
358
359     if (zhp == NULL) {
360         *isactive = B_FALSE;
361         return (0);
362     }
363
364     verify(nvlist_lookup_uint64(zhp->zpool_config, ZPOOL_CONFIG_POOL_GUID,
365                                 &theguid) == 0);
366
367     zpool_close(zhp);
368
369     *isactive = (theguid == guid);
370     return (0);
371 }
372
373 static nvlist_t *
374 refresh_config(libzfs_handle_t *hdl, nvlist_t *config)
375 {
376     nvlist_t *nvl;
377     zfs_cmd_t zc = { 0 };
378     int err;
379
380     if (zcmd_wwrite_conf_nvlist(hdl, &zc, config) != 0)
381         return (NULL);
382
383     if (zcmd_alloc_dst_nvlist(hdl, &zc,
384                               zc.zc_nvlist_conf_size * 2) != 0) {
385         zcmd_free_nvlists(&zc);
386         return (NULL);
387     }
388
389     while ((err = ioctl(hdl->libzfs_fd, ZFS_IOC_POOL_TRYIMPORT,
390                         &zc)) != 0 && errno == ENOMEM) {
391         if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {

```

```

392             zcmd_free_nvlists(&zc);
393             return (NULL);
394         }
395     }
396
397     if (err) {
398         zcmd_free_nvlists(&zc);
399         return (NULL);
400     }
401
402     if (zcmd_read_dst_nvlist(hdl, &zc, &nvl) != 0) {
403         zcmd_free_nvlists(&zc);
404         return (NULL);
405     }
406
407     zcmd_free_nvlists(&zc);
408     return (nvl);
409 }

410 /*
411  * Determine if the vdev id is a hole in the namespace.
412  */
413 boolean_t
414 vdev_is_hole(uint64_t *hole_array, uint_t holes, uint_t id)
415 {
416     for (int c = 0; c < holes; c++) {
417
418         /* Top-level is a hole */
419         if (hole_array[c] == id)
420             return (B_TRUE);
421     }
422
423     return (B_FALSE);
424 }

425 /*
426  * Convert our list of pools into the definitive set of configurations. We
427  * start by picking the best config for each toplevel vdev. Once that's done,
428  * we assemble the toplevel vdevs into a full config for the pool. We make a
429  * pass to fix up any incorrect paths, and then add it to the main list to
430  * return to the user.
431  */
432 static nvlist_t *
433 get_configs(libzfs_handle_t *hdl, pool_list_t *pl, boolean_t active_ok)
434 {
435     pool_entry_t *pe;
436     vdev_entry_t *ve;
437     config_entry_t *ce;
438
439     nvlist_t *ret = NULL, *config = NULL, *tmp = NULL, *nvtop, *nvroot;
440     nvlist_t *ret = NULL, *config = NULL, *tmp, *nvtop, *nvroot;
441     nvlist_t **spares, **l2cache;
442     uint_t i, nspares, nl2cache;
443     boolean_t config_seen;
444     uint64_t best_txg;
445     char *name, *hostname = NULL;
446     char *name, *hostname;
447     uint64_t guid;
448     uint_t children = 0;
449     nvlist_t **child = NULL;
450     uint_t holes;
451     uint64_t *hole_array, max_id;
452     uint_t c;
453     boolean_t isactive;
454     uint64_t hostid;
455     nvlist_t *nvl;
456     boolean_t found_one = B_FALSE;
457     boolean_t valid_top_config = B_FALSE;

```

```

457     if (nvlist_alloc(&ret, 0, 0) != 0)
458         goto nomem;
459
460     for (pe = pl->pools; pe != NULL; pe = pe->pe_next) {
461         uint64_t id, max_txg = 0;
462
463         if (nvlist_alloc(&config, NV_UNIQUE_NAME, 0) != 0)
464             goto nomem;
465         config_seen = B_FALSE;
466
467         /*
468          * Iterate over all toplevel vdevs. Grab the pool configuration
469          * from the first one we find, and then go through the rest and
470          * add them as necessary to the 'vdevs' member of the config.
471         */
472         for (ve = pe->pe_vdevs; ve != NULL; ve = ve->ve_next) {
473
474             /*
475              * Determine the best configuration for this vdev by
476              * selecting the config with the latest transaction
477              * group.
478             */
479             best_txg = 0;
480             for (ce = ve->ve_configs; ce != NULL;
481                  ce = ce->ce_next) {
482
483                 if (ce->ce_txg > best_txg) {
484                     tmp = ce->ce_config;
485                     best_txg = ce->ce_txg;
486                 }
487             }
488
489             /*
490              * We rely on the fact that the max txg for the
491              * pool will contain the most up-to-date information
492              * about the valid top-levels in the vdev namespace.
493             */
494             if (best_txg > max_txg) {
495                 (void) nvlist_remove(config,
496                                     ZPOOL_CONFIG_VDEV_CHILDREN,
497                                     DATA_TYPE_UINT64);
498                 (void) nvlist_remove(config,
499                                     ZPOOL_CONFIG_HOLE_ARRAY,
500                                     DATA_TYPE_UINT64_ARRAY);
501
502             max_txg = best_txg;
503             hole_array = NULL;
504             holes = 0;
505             max_id = 0;
506             valid_top_config = B_FALSE;
507
508             if (nvlist_lookup_uint64(tmp,
509                                     ZPOOL_CONFIG_VDEV_CHILDREN, &max_id) == 0) {
510                 verify(nvlist_add_uint64(config,
511                                     ZPOOL_CONFIG_VDEV_CHILDREN,
512                                     max_id) == 0);
513                 valid_top_config = B_TRUE;
514             }
515
516             if (nvlist_lookup_uint64_array(tmp,
517                                     ZPOOL_CONFIG_HOLE_ARRAY, &hole_array,
518                                     &holes) == 0) {
519                 verify(nvlist_add_uint64_array(config,
520                                     ZPOOL_CONFIG_HOLE_ARRAY,
521                                     hole_array, holes) == 0);
522             }
523         }
524     }
525
526     if (nvlist_free(config) != 0)
527         goto nomem;
528
529     if (nvlist_free(ret) != 0)
530         goto nomem;
531
532     return nomem;
533 }

```

```

522         }
523     }
524
525     if (!config_seen) {
526         /*
527          * Copy the relevant pieces of data to the pool
528          * configuration:
529          *
530          *   version
531          *   pool guid
532          *   name
533          *   comment (if available)
534          *   pool state
535          *   hostid (if available)
536          *   hostname (if available)
537          */
538
539     uint64_t state, version;
540     char *comment = NULL;
541
542     version = fnvlist_lookup_uint64(tmp,
543                                     ZPOOL_CONFIG_VERSION);
544     fnvlist_add_uint64(config,
545                         ZPOOL_CONFIG_VERSION, version);
546     guid = fnvlist_lookup_uint64(tmp,
547                                 ZPOOL_CONFIG_POOL_GUID);
548     fnvlist_add_uint64(config,
549                         ZPOOL_CONFIG_POOL_GUID, guid);
550     name = fnvlist_lookup_string(tmp,
551                                 ZPOOL_CONFIG_POOL_NAME);
552     fnvlist_add_string(config,
553                         ZPOOL_CONFIG_POOL_NAME, name);
554
555     if (fnvlist_lookup_string(tmp,
556                               ZPOOL_CONFIG_COMMENT, &comment) == 0)
557         fnvlist_add_string(config,
558                             ZPOOL_CONFIG_COMMENT, comment);
559
560     state = fnvlist_lookup_uint64(tmp,
561                                   ZPOOL_CONFIG_POOL_STATE);
562     fnvlist_add_uint64(config,
563                         ZPOOL_CONFIG_POOL_STATE, state);
564
565     hostid = 0;
566     if (fnvlist_lookup_uint64(tmp,
567                               ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
568         fnvlist_add_uint64(config,
569                             ZPOOL_CONFIG_HOSTID, hostid);
570         hostname = fnvlist_lookup_string(tmp,
571                                         ZPOOL_CONFIG_HOSTNAME);
572         fnvlist_add_string(config,
573                             ZPOOL_CONFIG_HOSTNAME, hostname);
574     }
575
576     config_seen = B_TRUE;
577 }
578
579 /*
580  * Add this top-level vdev to the child array.
581  */
582 verify(nvlist_lookup_nvlist(tmp,
583                            ZPOOL_CONFIG_VDEV_TREE, &nvtop) == 0);
584 verify(nvlist_lookup_uint64(nvtop, ZPOOL_CONFIG_ID,
585                            &id) == 0);
586
587 if (id >= children) {
588     nvlist_t **newchild;

```

```

589     newchild = zfs_alloc(hdl, (id + 1) *
590                          sizeof (nvlist_t *));
591     if (newchild == NULL)
592         goto nomem;
593
594     for (c = 0; c < children; c++)
595         newchild[c] = child[c];
596
597     free(child);
598     child = newchild;
599     children = id + 1;
600
601     if (nvlist_dup(nvtop, &child[id], 0) != 0)
602         goto nomem;
603
604 }
605
606 /*
607  * If we have information about all the top-levels then
608  * clean up the nvlist which we've constructed. This
609  * means removing any extraneous devices that are
610  * beyond the valid range or adding devices to the end
611  * of our array which appear to be missing.
612 */
613 if (valid_top_config) {
614     if (max_id < children) {
615         for (c = max_id; c < children; c++)
616             nvlist_free(child[c]);
617         children = max_id;
618     } else if (max_id > children) {
619         nvlist_t **newchild;
620
621         newchild = zfs_alloc(hdl, (max_id) *
622                             sizeof (nvlist_t *));
623         if (newchild == NULL)
624             goto nomem;
625
626         for (c = 0; c < children; c++)
627             newchild[c] = child[c];
628
629         free(child);
630         child = newchild;
631         children = max_id;
632     }
633
634     verify(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID,
635                                &guid) == 0);
636
637 /*
638  * The vdev namespace may contain holes as a result of
639  * device removal. We must add them back into the vdev
640  * tree before we process any missing devices.
641  */
642 if (holes > 0) {
643     ASSERT(valid_top_config);
644
645     for (c = 0; c < children; c++) {
646         nvlist_t *holey;
647
648         if (child[c] != NULL ||
649             !vdev_is_hole(hole_array, holes, c))
650             continue;
651
652         if (nvlist_alloc(&holey, NV_UNIQUE_NAME,
653

```

```

654             0) != 0)
655             goto nomem;
656
657         /*
658          * Holes in the namespace are treated as
659          * "hole" top-level vdevs and have a
660          * special flag set on them.
661         */
662         if (nvlist_add_string(holey,
663             ZPOOL_CONFIG_TYPE,
664             VDEV_TYPE_HOLE) != 0 ||
665             nvlist_add_uint64(holey,
666                 ZPOOL_CONFIG_ID, c) != 0 ||
667             nvlist_add_uint64(holey,
668                 ZPOOL_CONFIG_GUID, OULL) != 0)
669             goto nomem;
670         child[c] = holey;
671     }
672
673     /*
674      * Look for any missing top-level vdevs. If this is the case,
675      * create a faked up 'missing' vdev as a placeholder. We cannot
676      * simply compress the child array, because the kernel performs
677      * certain checks to make sure the vdev IDs match their location
678      * in the configuration.
679     */
680     for (c = 0; c < children; c++) {
681         if (child[c] == NULL) {
682             nvlist_t *missing;
683             if (nvlist_alloc(&missing, NV_UNIQUE_NAME,
684                 0) != 0)
685                 goto nomem;
686             if (nvlist_add_string(missing,
687                 ZPOOL_CONFIG_TYPE,
688                 VDEV_TYPE_MISSING) != 0 ||
689                 nvlist_add_uint64(missing,
690                     ZPOOL_CONFIG_ID, c) != 0 ||
691                 nvlist_add_uint64(missing,
692                     ZPOOL_CONFIG_GUID, OULL) != 0) {
693                 nvlist_free(missing);
694                 goto nomem;
695             }
696             child[c] = missing;
697         }
698     }
699
700     /*
701      * Put all of this pool's top-level vdevs into a root vdev.
702     */
703     if (nvlist_alloc(&nvroot, NV_UNIQUE_NAME, 0) != 0)
704         goto nomem;
705     if (nvlist_add_string(nvroot, ZPOOL_CONFIG_TYPE,
706         VDEV_TYPE_ROOT) != 0 ||
707         nvlist_add_uint64(nvroot, ZPOOL_CONFIG_ID, OULL) != 0 ||
708         nvlist_add_uint64(nvroot, ZPOOL_CONFIG_GUID, guid) != 0 ||
709         nvlist_add_nvlist_array(nvroot, ZPOOL_CONFIG_CHILDREN,
710             child, children) != 0) {
711             nvlist_free(nvroot);
712             goto nomem;
713         }
714
715     for (c = 0; c < children; c++)
716         nvlist_free(child[c]);
717     free(child);
718     children = 0;
719

```

```

720             child = NULL;
721
722             /*
723              * Go through and fix up any paths and/or devids based on our
724              * known list of vdev GUID -> path mappings.
725             */
726             if (fix_paths(nvroot, pl->names) != 0) {
727                 nvlist_free(nvroot);
728                 goto nomem;
729             }
730
731             /*
732              * Add the root vdev to this pool's configuration.
733             */
734             if (nvlist_add_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
735                 nvroot) != 0) {
736                 nvlist_free(nvroot);
737                 goto nomem;
738             }
739             nvlist_free(nvroot);
740
741             /*
742              * zdb uses this path to report on active pools that were
743              * imported or created using -R.
744             */
745             if (active_ok)
746                 goto add_pool;
747
748             /*
749              * Determine if this pool is currently active, in which case we
750              * can't actually import it.
751             */
752             verify(nvlist_lookup_string(config, ZPOOL_CONFIG_POOL_NAME,
753                 &name) == 0);
754             verify(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID,
755                 &guid) == 0);
756
757             if (pool_active(hdl, name, guid, &isactive) != 0)
758                 goto error;
759
760             if (isactive) {
761                 nvlist_free(config);
762                 config = NULL;
763                 continue;
764             }
765
766             if ((nvl = refresh_config(hdl, config)) == NULL) {
767                 nvlist_free(config);
768                 config = NULL;
769                 continue;
770             }
771
772             nvlist_free(config);
773             config = nvl;
774
775             /*
776              * Go through and update the paths for spares, now that we have
777              * them.
778             */
779             verify(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
780                 &nvroot) == 0);
781             if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
782                 &spares, &nspares) == 0) {
783                 for (i = 0; i < nspares; i++) {
784                     if (fix_paths(spares[i], pl->names) != 0)
785                         goto nomem;

```

```

786         }
787     }
788
789     /*
790      * Update the paths for l2cache devices.
791      */
792     if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
793         &l2cache, &n12cache) == 0) {
794         for (i = 0; i < n12cache; i++) {
795             if (fix_paths(l2cache[i], pl->names) != 0)
796                 goto nomem;
797         }
798     }
799
800     /*
801      * Restore the original information read from the actual label.
802      */
803     (void) nvlist_remove(config, ZPOOL_CONFIG_HOSTID,
804         DATA_TYPE_UINT64);
805     (void) nvlist_remove(config, ZPOOL_CONFIG_HOSTNAME,
806         DATA_TYPE_STRING);
807     if (hostid != 0) {
808         verify(nvlist_add_uint64(config, ZPOOL_CONFIG_HOSTID,
809             hostid) == 0);
810         verify(nvlist_add_string(config, ZPOOL_CONFIG_HOSTNAME,
811             hostname) == 0);
812     }
813
814 add_pool:
815     /*
816      * Add this pool to the list of configs.
817      */
818     verify(nvlist_lookup_string(config, ZPOOL_CONFIG_POOL_NAME,
819         &name) == 0);
820     if (nvlist_add_nvlist(ret, name, config) != 0)
821         goto nomem;
822
823     found_one = B_TRUE;
824     nvlist_free(config);
825     config = NULL;
826 }
827
828     if (!found_one) {
829         nvlist_free(ret);
830         ret = NULL;
831     }
832
833     return (ret);
834
835 nomem:
836     (void) no_memory(hdl);
837 error:
838     nvlist_free(config);
839     nvlist_free(ret);
840     for (c = 0; c < children; c++)
841         nvlist_free(child[c]);
842     free(child);
843
844     return (NULL);
845 }
846 unchanged_portion_omitted_
847
848 typedef struct slice_node {
849     char *sn_name;
850     nvlist_t *sn_config;
851     boolean_t sn_nozpool;

```

```

914         int sn_partno;
915         struct disk_node *sn_disk;
916         struct slice_node *sn_next;
917     } slice_node_t;
918
919     typedef struct disk_node {
920         char *dn_name;
921         int dn_dfd;
922         libzfs_handle_t *dn_hdl;
923         nvlist_t *dn_config;
924         struct slice_node *dn_slices;
925         struct disk_node *dn_next;
926     } disk_node_t;
927
928 #ifdef sparc
929 #define WHOLE_DISK      "s2"
930 #else
931 #define WHOLE_DISK      "p0"
932 #endif
933
934     typedef struct rdsk_node {
935         char *rn_name;
936         int rn_dfd;
937         libzfs_handle_t *rn_hdl;
938         nvlist_t *rn_config;
939         avl_tree_t *rn_avl;
940         avl_node_t rn_node;
941         boolean_t rn_nozpool;
942     } rdsk_node_t;
943
944     /*
945      * This function splits the slice from the device name. Currently it supports
946      * VTOC slices (s[0-16]) and DOS/FDISK partitions (p[0-4]). If this function
947      * is updated to support other slice types then the check_slices function will
948      * also need to be updated.
949      */
950     static boolean_t
951     get_disk_slice(libzfs_handle_t *hdl, char *disk, char **slice, int *partno)
952     static int
953     slice_cache_compare(const void *arg1, const void *arg2)
954     {
955         char *p;
956
957         if ((p = strrchr(disk, 's')) == NULL &&
958             (p = strrchr(disk, 'p')) == NULL)
959             return (B_FALSE);
960
961         if (!isdigit(p[1]))
962             return (B_FALSE);
963
964         const char *nm1 = ((rdsk_node_t *)arg1)->rn_name;
965         const char *nm2 = ((rdsk_node_t *)arg2)->rn_name;
966         char *nm1slice, *nm2slice;
967         int rv;
968
969         *slice = zfs_strdup(hdl, p);
970         *partno = atoi(p + 1);
971
972         /*
973          * slices zero and two are the most likely to provide results,
974          * so put those first
975          */
976         nm1slice = strstr(nm1, "s0");
977         nm2slice = strstr(nm2, "s0");
978         if (nm1slice && !nm2slice)
979             return (-1);
980
981         if (!nm1slice && nm2slice)
982             return (1);

```

```

525     }
526     nm1slice = strstr(nm1, "s2");
527     nm2slice = strstr(nm2, "s2");
528     if (nm1slice && !nm2slice) {
529         return (-1);
530     }
531     if (!nm1slice && nm2slice) {
532         return (1);
533     }
534
535     p = '\0';
536     return (B_TRUE);
537     rv = strcmp(nm1, nm2);
538     if (rv == 0)
539         return (0);
540     return (rv > 0 ? 1 : -1);
541 }
542
543 static void
544 check_one_slice(slice_node_t *slice, diskaddr_t size, uint_t blksz)
545 {
546     rdsk_node_t tmpnode;
547     rdsk_node_t *node;
548     char sname[MAXNAMELEN];
549
550     tmpnode.rn_name = &sname[0];
551     (void) snprintf(tmpnode.rn_name, MAXNAMELEN, "%s%u",
552                     diskname, partno);
553
554     /* protect against division by zero for disk labels that
555      * contain a bogus sector size
556      */
557     if (blksz == 0)
558         blksz = DEV_BSIZE;
559     /* too small to contain a zpool? */
560     if (size < (SPA_MINDEVSIZE / blksz))
561         slice->sn_nozpool = B_TRUE;
562     if ((size < (SPA_MINDEVSIZE / blksz)) &&
563         (node = avl_find(r, &tmpnode, NULL)))
564         node->rn_nozpool = B_TRUE;
565 }
566
567 static void
568 check_slices(slice_node_t *slices, int fd)
569 nozpool_all_slices(avl_tree_t *r, const char *sname)
570 {
571     char diskname[MAXNAMELEN];
572     char *ptr;
573     int i;
574
575     (void) strncpy(diskname, sname, MAXNAMELEN);
576     if (((ptr = strrchr(diskname, 's')) == NULL) &&
577         ((ptr = strrchr(diskname, 'p')) == NULL))
578         return;
579     ptr[0] = 's';
580     ptr[1] = '\0';
581     for (i = 0; i < NDKMAP; i++)
582         check_one_slice(r, diskname, i, 0, 1);
583     ptr[0] = 'p';
584     for (i = 0; i <= FD_NUMPART; i++)
585         check_one_slice(r, diskname, i, 0, 1);
586 }
587
588 static void

```

```

589 check_slices(avl_tree_t *r, int fd, const char *sname)
590 {
591     struct extvtoc vtoc;
592     struct dk_gpt *gpt;
593     slice_node_t *slice;
594     diskaddr_t size;
595     char diskname[MAXNAMELEN];
596     char *ptr;
597     int i;
598
599     (void) strncpy(diskname, sname, MAXNAMELEN);
600     if ((ptr = strrchr(diskname, 's')) == NULL || !isdigit(ptr[1]))
601         return;
602     ptr[1] = '\0';
603
604     if (read_extvtoc(fd, &vtoc) >= 0) {
605         for (slice = slices; slice; slice = slice->sn_next) {
606             if (slice->sn_name[0] == 'p')
607                 continue;
608             size = vtoc.v_part[slice->sn_partno].p_size;
609             check_one_slice(slice, size, vtoc.v_sectorsz);
610         }
611         for (i = 0; i < NDKMAP; i++)
612             check_one_slice(r, diskname, i,
613                             vtoc.v_part[i].p_size, vtoc.v_sectorsz);
614     } else if (efi_alloc_and_read(fd, &gpt) >= 0) {
615         for (slice = slices; slice; slice = slice->sn_next) {
616             /*
617              * on x86 we'll still have leftover links that point
618              * to slices s[9-15], so use NDKMAP instead
619              */
620             for (i = 0; i < NDKMAP; i++)
621                 check_one_slice(r, diskname, i,
622                                 gpt->efi_parts[i].p_size, gpt->efi_lbasize);
623             /* nodes p[1-4] are never used with EFI labels */
624             if (slice->sn_name[0] == 'p') {
625                 if (slice->sn_partno > 0)
626                     slice->sn_nozpool = B_TRUE;
627                 continue;
628             }
629             size = gpt->efi_parts[slice->sn_partno].p_size;
630             check_one_slice(slice, size, gpt->efi_lbasize);
631         }
632         ptr[0] = 'p';
633         for (i = 1; i <= FD_NUMPART; i++)
634             check_one_slice(r, diskname, i, 0, 1);
635     }
636     efi_free(gpt);
637 }
638
639 static void
640 zpool_open_func(void *arg)
641 {
642     disk_node_t *disk = arg;
643     rdsk_node_t *rn = arg;
644     struct stat64 statbuf;
645     slice_node_t *slice;
646
647     #endif /* ! codereview */
648     nvlist_t *config;
649     char *devname;
650     #endif /* ! codereview */
651     int fd;
652
653     /*
654      * If the disk has no slices we open it directly, otherwise we try
655      * to open the whole disk slice.
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1085
1086
1087
1088
1089
1089
1090
1091
1092
1093
1094
1095
1095
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1179
1180
1181
1182
1183
1184
1185
1185
1186
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473

```

```

1018     */
1019     if (disk->dn_slices == NULL)
1020         devname = strdup(disk->dn_name);
1021     else
1022         (void) asprintf(&devname, "%s WHOLE_DISK, disk->dn_name);
1023
1024     if (devname == NULL) {
1025         (void) no_memory(disk->dn_hdl);
1026         if (rn->rn_nozpool)
1027             return;
1028
1029     if ((fd = openat64(disk->dn_dfd, devname, O_RDONLY)) < 0) {
1030         free(devname);
1031         if ((fd = openat64(rn->rn_dfd, rn->rn_name, O_RDONLY)) < 0) {
1032             /* symlink to a device that's no longer there */
1033             if (errno == ENOENT)
1034                 nozpool_all_slices(rn->rn_avl, rn->rn_name);
1035             return;
1036         }
1037         /*
1038          * Ignore failed stats. We only want regular
1039          * files, character devs and block devs.
1040         */
1041         if (fstat64(fd, &statbuf) != 0 ||
1042             (!S_ISREG(statbuf.st_mode) &&
1043              !S_ISCHR(statbuf.st_mode) &&
1044              !S_ISBLK(statbuf.st_mode))) {
1045             (void) close(fd);
1046             free(devname);
1047         #endif /* ! codereview */
1048         }
1049     }
1050     /* this file is too small to hold a zpool */
1051     if (S_ISREG(statbuf.st_mode) && statbuf.st_size < SPA_MINDEVSIZE) {
1052         if (S_ISREG(statbuf.st_mode) &&
1053             statbuf.st_size < SPA_MINDEVSIZE) {
1054             (void) close(fd);
1055             free(devname);
1056         #endif /* ! codereview */
1057         }
1058         check_slices(disk->dn_slices, fd);
1059         check_slices(rn->rn_avl, fd, rn->rn_name);
1060
1061     /*
1062      * If we're working with the device directly (it has no slices)
1063      * then we can just read the config and we're done.
1064     */
1065     if (disk->dn_slices == NULL) {
1066         if (zpool_read_label(fd, &config) != 0) {
1067             (void) no_memory(disk->dn_hdl);
1068             (void) close(fd);
1069             free(devname);
1070             return;
1071         }
1072         disk->dn_config = config;
1073         if ((zpool_read_label(fd, &config)) != 0) {
1074             (void) close(fd);
1075             free(devname);
1076     }

```

```

653         (void) no_memory(rn->rn_hdl);
654         return;
655     }
656
657 #endif /* ! codereview */
658     (void) close(fd);
659     free(devname);
660
661     /*
662      * Go through and read the label off each slice. The check_slices
663      * function has already performed some basic checks and set the
664      * sn_nozpool flag on any slices which just can't contain a zpool.
665      */
666     for (slice = disk->dn_slices; slice; slice = slice->sn_next) {
667         if (slice->sn_nozpool == B_TRUE)
668             continue;
669
670         (void) asprintf(&devname, "%s%s", disk->dn_name,
671                         slice->sn_name);
672
673         if (devname == NULL) {
674             (void) no_memory(disk->dn_hdl);
675             free(devname);
676             return;
677         }
678     #endif /* ! codereview */
679
680     if ((fd = openat64(disk->dn_dfd, devname, O_RDONLY)) < 0) {
681         free(devname);
682         continue;
683     }
684
685     if ((zpool_read_label(fd, &config)) != 0) {
686         (void) no_memory(disk->dn_hdl);
687         (void) close(fd);
688         free(devname);
689         return;
690     }
691     #endif /* ! codereview */
692
693     slice->sn_config = config;
694     (void) close(fd);
695     free(devname);
696     rn->rn_config = config;
697     if (config != NULL) {
698         assert(rn->rn_nozpool == B_FALSE);
699     }
700 }
701
702 unchanged_portion_omitted_
703
704 /*
705  * Given a list of directories to search, find all pools stored on disk. This
706  * includes partial pools which are not available to import. If no args are
707  * given (argc is 0), then the default directory (/dev/dsk) is searched.
708  * poolname or guid (but not both) are provided by the caller when trying
709  * to import a specific pool.
710 */
711 static nvlist_t *
712 zpool_find_import_impl(libzfs_handle_t *hdl, importargs_t *iarg)
713 {
714     int i, dirs = iarg->paths;
715     DIR *dirp = NULL;
716     struct dirent64 *dp;
717     char path[MAXPATHLEN];
718     char *end, **dir = iarg->path;
719     size_t pathleft;

```

```

1165     nvlist_t *ret = NULL;
1166     static char *default_dir = "/dev/dsk";
1167     pool_list_t pools = { 0 };
1168     pool_entry_t *pe, *penext;
1169     vdev_entry_t *ve, *venext;
1170     config_entry_t *ce, *cenext;
1171     name_entry_t *ne, *nenext;
1172     avl_tree_t slice_cache;
1173     rdsk_node_t *slice;
1174     void *cookie;
1175
1176     if (dirs == 0) {
1177         dirs = 1;
1178         dir = &default_dir;
1179     }
1180
1181     /*
1182      * Go through and read the label configuration information from every
1183      * possible device, organizing the information according to pool GUID
1184      * and toplevel GUID.
1185     */
1186     for (i = 0; i < dirs; i++) {
1187         tpool_t *t;
1188         char *rdsk;
1189         int dfd;
1190         disk_node_t *disks = NULL, *curdisk = NULL;
1191         slice_node_t *curslice = NULL;
1192 #endif /* ! codereview */
1193
1194         /* use realpath to normalize the path */
1195         if (realpath(dir[i], path) == 0) {
1196             (void) zfs_error_fmt(hdl, EZFS_BADPATH,
1197                                 dgettext(TEXT_DOMAIN, "cannot open '%s'"),
1198                                 dir[i]);
1199             goto error;
1200         }
1201         end = &path[strlen(path)];
1202         *end++ = '/';
1203         *end = 0;
1204         pathleft = &path[sizeof (path)] - end;
1205
1206         /*
1207          * Using raw devices instead of block devices when we're
1208          * reading the labels skips a bunch of slow operations during
1209          * close(2) processing, so we replace /dev/dsk with /dev/rdsk.
1210          */
1211         if (strcmp(path, "/dev/dsk/") == 0)
1212             rdsks = "/dev/rdsk/";
1213         else
1214             rdsks = path;
1215
1216         if ((dfd = open64(rdsks, O_RDONLY)) < 0 ||
1217             (dirp = fdopendir(dfd)) == NULL) {
1218             zfs_error_aux(hdl, strerror(errno));
1219             (void) zfs_error_fmt(hdl, EZFS_BADPATH,
1220                                 dgettext(TEXT_DOMAIN, "cannot open '%s'"),
1221                                 rdsks);
1222             goto error;
1223         }
1224
1225         avl_create(&slice_cache, slice_cache_compare,
1226                   sizeof (rdsk_node_t), offsetof(rdsk_node_t, rn_node));
1227
1228         /*
1229          * This is not MT-safe, but we have no MT consumers of libzfs
1230         */
1231         while ((dp = readdir64(dirp)) != NULL) {
1232             boolean_t isslice;
1233
1234             if (dp->d_name[0] == '.' && (dp->d_name[1] == '\0' ||
1235                 (dp->d_name[1] == '.' && dp->d_name[2] == '\0'))) {
1236                 const char *name = dp->d_name;
1237                 if (name[0] == '.' && (name[1] == 0 || (name[1] == '.' && name[2] == 0)))
1238                     continue;
1239
1240                 name = zfs_strdup(hdl, dp->d_name);
1241
1242                 if (isslice == B_FALSE || curdisk == NULL ||
1243                     strcmp(curdisk->dn_name, name) != 0) {
1244
1245                     newdisk = zfs_alloc(hdl, sizeof (disk_node_t));
1246                     newdisk->dn_name = name;
1247                     newdisk->dn_dfd = dfd;
1248                     newdisk->dn_hdl = hdl;
1249
1250                     if (curdisk != NULL)
1251                         curdisk->dn_next = newdisk;
1252                     else
1253                         disks = newdisk;
1254
1255                     curdisk = newdisk;
1256                     curslice = NULL;
1257
1258                     assert(curdisk != NULL);
1259
1260                     /*
1261                      * Add a new slice node to the current disk node.
1262                      * We do this for all slices including zero slices.
1263                     */
1264                     if (isslice == B_TRUE) {
1265                         slice_node_t *newslice;
1266
1267                         newslice = zfs_alloc(hdl,
1268                                   sizeof (slice_node_t));
1269                         newslice->sn_name = sname;
1270                         newslice->sn_partno = partno;
1271                         newslice->sn_disk = curdisk;
1272
1273                         if (curslice != NULL)
1274                             curslice->sn_next = newslice;
1275                         else
1276                             curdisk->dn_slices = newslice;
1277
1278                         curslice = newslice;
1279
1280                         slice = zfs_alloc(hdl, sizeof (rdsk_node_t));
1281                         slice->rn_name = zfs_strdup(hdl, name);
1282                         slice->rn_avl = &slice_cache;
1283                         slice->rn_dfd = dfd;
1284                         slice->rn_hdl = hdl;
1285                         slice->rn_nozpool = B_FALSE;
1286                         avl_add(&slice_cache, slice);
1287
1288                     }
1289
1290                 }
1291
1292             }
1293
1294             if (dp->d_name[0] == '.' && (dp->d_name[1] == '\0' ||
1295                 (dp->d_name[1] == '.' && dp->d_name[2] == '\0'))) {
1296                 const char *name = dp->d_name;
1297                 if (name[0] == '.' && (name[1] == 0 || (name[1] == '.' && name[2] == 0)))
1298                     continue;
1299
1300                 name = zfs_strdup(hdl, dp->d_name);
1301
1302                 if (isslice == B_FALSE || curdisk == NULL ||
1303                     strcmp(curdisk->dn_name, name) != 0) {
1304
1305                     newdisk = zfs_alloc(hdl, sizeof (disk_node_t));
1306                     newdisk->dn_name = name;
1307                     newdisk->dn_dfd = dfd;
1308                     newdisk->dn_hdl = hdl;
1309
1310                     if (curdisk != NULL)
1311                         curdisk->dn_next = newdisk;
1312                     else
1313                         disks = newdisk;
1314
1315                     curdisk = newdisk;
1316                     curslice = NULL;
1317
1318                     assert(curdisk != NULL);
1319
1320                     /*
1321                      * Add a new slice node to the current disk node.
1322                      * We do this for all slices including zero slices.
1323                     */
1324                     if (isslice == B_TRUE) {
1325                         slice_node_t *newslice;
1326
1327                         newslice = zfs_alloc(hdl,
1328                                   sizeof (slice_node_t));
1329                         newslice->sn_name = sname;
1330                         newslice->sn_partno = partno;
1331                         newslice->sn_disk = curdisk;
1332
1333                         if (curslice != NULL)
1334                             curslice->sn_next = newslice;
1335                         else
1336                             curdisk->dn_slices = newslice;
1337
1338                         curslice = newslice;
1339
1340                         slice = zfs_alloc(hdl, sizeof (rdsk_node_t));
1341                         slice->rn_name = zfs_strdup(hdl, name);
1342                         slice->rn_avl = &slice_cache;
1343                         slice->rn_dfd = dfd;
1344                         slice->rn_hdl = hdl;
1345                         slice->rn_nozpool = B_FALSE;
1346                         avl_add(&slice_cache, slice);
1347
1348                     }
1349
1350                 }
1351
1352             }
1353
1354         }
1355
1356     }
1357
1358     if (ret != NULL)
1359         nvlist_free(ret);
1360
1361     if (curdisk != NULL)
1362         disk_free(curdisk);
1363
1364     if (newdisk != NULL)
1365         disk_free(newdisk);
1366
1367     if (curslice != NULL)
1368         slice_free(curslice);
1369
1370     if (slice != NULL)
1371         slice_free(slice);
1372
1373     if (disks != NULL)
1374         disk_free(disks);
1375
1376     if (newdisk != NULL)
1377         disk_free(newdisk);
1378
1379     if (curslice != NULL)
1380         slice_free(curslice);
1381
1382     if (slice != NULL)
1383         slice_free(slice);
1384
1385     if (disks != NULL)
1386         disk_free(disks);
1387
1388     if (newdisk != NULL)
1389         disk_free(newdisk);
1390
1391     if (curslice != NULL)
1392         slice_free(curslice);
1393
1394     if (slice != NULL)
1395         slice_free(slice);
1396
1397     if (disks != NULL)
1398         disk_free(disks);
1399
1400     if (newdisk != NULL)
1401         disk_free(newdisk);
1402
1403     if (curslice != NULL)
1404         slice_free(curslice);
1405
1406     if (slice != NULL)
1407         slice_free(slice);
1408
1409     if (disks != NULL)
1410         disk_free(disks);
1411
1412     if (newdisk != NULL)
1413         disk_free(newdisk);
1414
1415     if (curslice != NULL)
1416         slice_free(curslice);
1417
1418     if (slice != NULL)
1419         slice_free(slice);
1420
1421     if (disks != NULL)
1422         disk_free(disks);
1423
1424     if (newdisk != NULL)
1425         disk_free(newdisk);
1426
1427     if (curslice != NULL)
1428         slice_free(curslice);
1429
1430     if (slice != NULL)
1431         slice_free(slice);
1432
1433     if (disks != NULL)
1434         disk_free(disks);
1435
1436     if (newdisk != NULL)
1437         disk_free(newdisk);
1438
1439     if (curslice != NULL)
1440         slice_free(curslice);
1441
1442     if (slice != NULL)
1443         slice_free(slice);
1444
1445     if (disks != NULL)
1446         disk_free(disks);
1447
1448     if (newdisk != NULL)
1449         disk_free(newdisk);
1450
1451     if (curslice != NULL)
1452         slice_free(curslice);
1453
1454     if (slice != NULL)
1455         slice_free(slice);
1456
1457     if (disks != NULL)
1458         disk_free(disks);
1459
1460     if (newdisk != NULL)
1461         disk_free(newdisk);
1462
1463     if (curslice != NULL)
1464         slice_free(curslice);
1465
1466     if (slice != NULL)
1467         slice_free(slice);
1468
1469     if (disks != NULL)
1470         disk_free(disks);
1471
1472     if (newdisk != NULL)
1473         disk_free(newdisk);
1474
1475     if (curslice != NULL)
1476         slice_free(curslice);
1477
1478     if (slice != NULL)
1479         slice_free(slice);
1480
1481     if (disks != NULL)
1482         disk_free(disks);
1483
1484     if (newdisk != NULL)
1485         disk_free(newdisk);
1486
1487     if (curslice != NULL)
1488         slice_free(curslice);
1489
1490     if (slice != NULL)
1491         slice_free(slice);
1492
1493     if (disks != NULL)
1494         disk_free(disks);
1495
1496     if (newdisk != NULL)
1497         disk_free(newdisk);
1498
1499     if (curslice != NULL)
1500         slice_free(curslice);
1501
1502     if (slice != NULL)
1503         slice_free(slice);
1504
1505     if (disks != NULL)
1506         disk_free(disks);
1507
1508     if (newdisk != NULL)
1509         disk_free(newdisk);
1510
1511     if (curslice != NULL)
1512         slice_free(curslice);
1513
1514     if (slice != NULL)
1515         slice_free(slice);
1516
1517     if (disks != NULL)
1518         disk_free(disks);
1519
1520     if (newdisk != NULL)
1521         disk_free(newdisk);
1522
1523     if (curslice != NULL)
1524         slice_free(curslice);
1525
1526     if (slice != NULL)
1527         slice_free(slice);
1528
1529     if (disks != NULL)
1530         disk_free(disks);
1531
1532     if (newdisk != NULL)
1533         disk_free(newdisk);
1534
1535     if (curslice != NULL)
1536         slice_free(curslice);
1537
1538     if (slice != NULL)
1539         slice_free(slice);
1540
1541     if (disks != NULL)
1542         disk_free(disks);
1543
1544     if (newdisk != NULL)
1545         disk_free(newdisk);
1546
1547     if (curslice != NULL)
1548         slice_free(curslice);
1549
1550     if (slice != NULL)
1551         slice_free(slice);
1552
1553     if (disks != NULL)
1554         disk_free(disks);
1555
1556     if (newdisk != NULL)
1557         disk_free(newdisk);
1558
1559     if (curslice != NULL)
1560         slice_free(curslice);
1561
1562     if (slice != NULL)
1563         slice_free(slice);
1564
1565     if (disks != NULL)
1566         disk_free(disks);
1567
1568     if (newdisk != NULL)
1569         disk_free(newdisk);
1570
1571     if (curslice != NULL)
1572         slice_free(curslice);
1573
1574     if (slice != NULL)
1575         slice_free(slice);
1576
1577     if (disks != NULL)
1578         disk_free(disks);
1579
1580     if (newdisk != NULL)
1581         disk_free(newdisk);
1582
1583     if (curslice != NULL)
1584         slice_free(curslice);
1585
1586     if (slice != NULL)
1587         slice_free(slice);
1588
1589     if (disks != NULL)
1590         disk_free(disks);
1591
1592     if (newdisk != NULL)
1593         disk_free(newdisk);
1594
1595     if (curslice != NULL)
1596         slice_free(curslice);
1597
1598     if (slice != NULL)
1599         slice_free(slice);
1600
1601     if (disks != NULL)
1602         disk_free(disks);
1603
1604     if (newdisk != NULL)
1605         disk_free(newdisk);
1606
1607     if (curslice != NULL)
1608         slice_free(curslice);
1609
1610     if (slice != NULL)
1611         slice_free(slice);
1612
1613     if (disks != NULL)
1614         disk_free(disks);
1615
1616     if (newdisk != NULL)
1617         disk_free(newdisk);
1618
1619     if (curslice != NULL)
1620         slice_free(curslice);
1621
1622     if (slice != NULL)
1623         slice_free(slice);
1624
1625     if (disks != NULL)
1626         disk_free(disks);
1627
1628     if (newdisk != NULL)
1629         disk_free(newdisk);
1630
1631     if (curslice != NULL)
1632         slice_free(curslice);
1633
1634     if (slice != NULL)
1635         slice_free(slice);
1636
1637     if (disks != NULL)
1638         disk_free(disks);
1639
1640     if (newdisk != NULL)
1641         disk_free(newdisk);
1642
1643     if (curslice != NULL)
1644         slice_free(curslice);
1645
1646     if (slice != NULL)
1647         slice_free(slice);
1648
1649     if (disks != NULL)
1650         disk_free(disks);
1651
1652     if (newdisk != NULL)
1653         disk_free(newdisk);
1654
1655     if (curslice != NULL)
1656         slice_free(curslice);
1657
1658     if (slice != NULL)
1659         slice_free(slice);
1660
1661     if (disks != NULL)
1662         disk_free(disks);
1663
1664     if (newdisk != NULL)
1665         disk_free(newdisk);
1666
1667     if (curslice != NULL)
1668         slice_free(curslice);
1669
1670     if (slice != NULL)
1671         slice_free(slice);
1672
1673     if (disks != NULL)
1674         disk_free(disks);
1675
1676     if (newdisk != NULL)
1677         disk_free(newdisk);
1678
1679     if (curslice != NULL)
1680         slice_free(curslice);
1681
1682     if (slice != NULL)
1683         slice_free(slice);
1684
1685     if (disks != NULL)
1686         disk_free(disks);
1687
1688     if (newdisk != NULL)
1689         disk_free(newdisk);
1690
1691     if (curslice != NULL)
1692         slice_free(curslice);
1693
1694     if (slice != NULL)
1695         slice_free(slice);
1696
1697     if (disks != NULL)
1698         disk_free(disks);
1699
1700     if (newdisk != NULL)
1701         disk_free(newdisk);
1702
1703     if (curslice != NULL)
1704         slice_free(curslice);
1705
1706     if (slice != NULL)
1707         slice_free(slice);
1708
1709     if (disks != NULL)
1710         disk_free(disks);
1711
1712     if (newdisk != NULL)
1713         disk_free(newdisk);
1714
1715     if (curslice != NULL)
1716         slice_free(curslice);
1717
1718     if (slice != NULL)
1719         slice_free(slice);
1720
1721     if (disks != NULL)
1722         disk_free(disks);
1723
1724     if (newdisk != NULL)
1725         disk_free(newdisk);
1726
1727     if (curslice != NULL)
1728         slice_free(curslice);
1729
1730     if (slice != NULL)
1731         slice_free(slice);
1732
1733     if (disks != NULL)
1734         disk_free(disks);
1735
1736     if (newdisk != NULL)
1737         disk_free(newdisk);
1738
1739     if (curslice != NULL)
1740         slice_free(curslice);
1741
1742     if (slice != NULL)
1743         slice_free(slice);
1744
1745     if (disks != NULL)
1746         disk_free(disks);
1747
1748     if (newdisk != NULL)
1749         disk_free(newdisk);
1750
1751     if (curslice != NULL)
1752         slice_free(curslice);
1753
1754     if (slice != NULL)
1755         slice_free(slice);
1756
1757     if (disks != NULL)
1758         disk_free(disks);
1759
1760     if (newdisk != NULL)
1761         disk_free(newdisk);
1762
1763     if (curslice != NULL)
1764         slice_free(curslice);
1765
1766     if (slice != NULL)
1767         slice_free(slice);
1768
1769     if (disks != NULL)
1770         disk_free(disks);
1771
1772     if (newdisk != NULL)
1773         disk_free(newdisk);
1774
1775     if (curslice != NULL)
1776         slice_free(curslice);
1777
1778     if (slice != NULL)
1779         slice_free(slice);
1780
1781     if (disks != NULL)
1782         disk_free(disks);
1783
1784     if (newdisk != NULL)
1785         disk_free(newdisk);
1786
1787     if (curslice != NULL)
1788         slice_free(curslice);
1789
1790     if (slice != NULL)
1791         slice_free(slice);
1792
1793     if (disks != NULL)
1794         disk_free(disks);
1795
1796     if (newdisk != NULL)
1797         disk_free(newdisk);
1798
1799     if (curslice != NULL)
1800         slice_free(curslice);
1801
1802     if (slice != NULL)
1803         slice_free(slice);
1804
1805     if (disks != NULL)
1806         disk_free(disks);
1807
1808     if (newdisk != NULL)
1809         disk_free(newdisk);
1810
1811     if (curslice != NULL)
1812         slice_free(curslice);
1813
1814     if (slice != NULL)
1815         slice_free(slice);
1816
1817     if (disks != NULL)
1818         disk_free(disks);
1819
1820     if (newdisk != NULL)
1821         disk_free(newdisk);
1822
1823     if (curslice != NULL)
1824         slice_free(curslice);
1825
1826     if (slice != NULL)
1827         slice_free(slice);
1828
1829     if (disks != NULL)
1830         disk_free(disks);
1831
1832     if (newdisk != NULL)
1833         disk_free(newdisk);
1834
1835     if (curslice != NULL)
1836         slice_free(curslice);
1837
1838     if (slice != NULL)
1839         slice_free(slice);
1840
1841     if (disks != NULL)
1842         disk_free(disks);
1843
1844     if (newdisk != NULL)
1845         disk_free(newdisk);
1846
1847     if (curslice != NULL)
1848         slice_free(curslice);
1849
1850     if (slice != NULL)
1851         slice_free(slice);
1852
1853     if (disks != NULL)
1854         disk_free(disks);
1855
1856     if (newdisk != NULL)
1857         disk_free(newdisk);
1858
1859     if (curslice != NULL)
1860         slice_free(curslice);
1861
1862     if (slice != NULL)
1863         slice_free(slice);
1864
1865     if (disks != NULL)
1866         disk_free(disks);
1867
1868     if (newdisk != NULL)
1869         disk_free(newdisk);
1870
1871     if (curslice != NULL)
1872         slice_free(curslice);
1873
1874     if (slice != NULL)
1875         slice_free(slice);
1876
1877     if (disks != NULL)
1878         disk_free(disks);
1879
1880     if (newdisk != NULL)
1881         disk_free(newdisk);
1882
1883     if (curslice != NULL)
1884         slice_free(curslice);
1885
1886     if (slice != NULL)
1887         slice_free(slice);
1888
1889     if (disks != NULL)
1890         disk_free(disks);
1891
1892     if (newdisk != NULL)
1893         disk_free(newdisk);
1894
1895     if (curslice != NULL)
1896         slice_free(curslice);
1897
1898     if (slice != NULL)
1899         slice_free(slice);
1899
1900     if (disks != NULL)
1901         disk_free(disks);
1902
1903     if (newdisk != NULL)
1904         disk_free(newdisk);
1905
1906     if (curslice != NULL)
1907         slice_free(curslice);
1908
1909     if (slice != NULL)
1910         slice_free(slice);
1910
1911     if (disks != NULL)
1912         disk_free(disks);
1913
1914     if (newdisk != NULL)
1915         disk_free(newdisk);
1916
1917     if (curslice != NULL)
1918         slice_free(curslice);
1919
1920     if (slice != NULL)
1921         slice_free(slice);
1921
1922     if (disks != NULL)
1923         disk_free(disks);
1924
1925     if (newdisk != NULL)
1926         disk_free(newdisk);
1927
1928     if (curslice != NULL)
1929         slice_free(curslice);
1930
1931     if (slice != NULL)
1932         slice_free(slice);
1932
1933     if (disks != NULL)
1934         disk_free(disks);
1935
1936     if (newdisk != NULL)
1937         disk_free(newdisk);
1938
1939     if (curslice != NULL)
1940         slice_free(curslice);
1940
1941     if (slice != NULL)
1942         slice_free(slice);
1942
1943     if (disks != NULL)
1944         disk_free(disks);
1945
1946     if (newdisk != NULL)
1947         disk_free(newdisk);
1948
1949     if (curslice != NULL)
1950         slice_free(curslice);
1950
1951     if (slice != NULL)
1952         slice_free(slice);
1952
1953     if (disks != NULL)
1954         disk_free(disks);
1955
1956     if (newdisk != NULL)
1957         disk_free(newdisk);
1958
1959     if (curslice != NULL)
1960         slice_free(curslice);
1960
1961     if (slice != NULL)
1962         slice_free(slice);
1962
1963     if (disks != NULL)
1964         disk_free(disks);
1965
1966     if (newdisk != NULL)
1967         disk_free(newdisk);
1968
1969     if (curslice != NULL)
1970         slice_free(curslice);
1970
1971     if (slice != NULL)
1972         slice_free(slice);
1972
1973     if (disks != NULL)
1974         disk_free(disks);
1975
1976     if (newdisk != NULL)
1977         disk_free(newdisk);
1978
1979     if (curslice != NULL)
1980         slice_free(curslice);
1980
1981     if (slice != NULL)
1982         slice_free(slice);
1982
1983     if (disks != NULL)
1984         disk_free(disks);
1985
1986     if (newdisk != NULL)
1987         disk_free(newdisk);
1988
1989     if (curslice != NULL)
1990         slice_free(curslice);
1990
1991     if (slice != NULL)
1992         slice_free(slice);
1992
1993     if (disks != NULL)
1994         disk_free(disks);
1995
1996     if (newdisk != NULL)
1997         disk_free(newdisk);
1998
1999     if (curslice != NULL)
2000         slice_free(curslice);
2000
2001     if (slice != NULL)
2002         slice_free(slice);
2002
2003     if (disks != NULL)
2004         disk_free(disks);
2005
2006     if (newdisk != NULL)
2007         disk_free(newdisk);
2008
2009     if (curslice != NULL)
2010         slice_free(curslice);
2010
2011     if (slice != NULL)
2012         slice_free(slice);
2012
2013     if (disks != NULL)
2014         disk_free(disks);
2015
2016     if (newdisk != NULL)
2017         disk_free(newdisk);
2018
2019     if (curslice != NULL)
2020         slice_free(curslice);
2020
2021     if (slice != NULL)
2022         slice_free(slice);
2022
2023     if (disks != NULL)
2024         disk_free(disks);
2025
2026     if (newdisk != NULL)
2027         disk_free(newdisk);
2028
2029     if (curslice != NULL)
2030         slice_free(curslice);
2030
2031     if (slice != NULL)
2032         slice_free(slice);
2032
2033     if (disks != NULL)
2034         disk_free(disks);
2035
2036     if (newdisk != NULL)
2037         disk_free(newdisk);
2038
2039     if (curslice != NULL)
2040         slice_free(curslice);
2040
2041     if (slice != NULL)
2042         slice_free(slice);
2042
2043     if (disks != NULL)
2044         disk_free(disks);
2045
2046     if (newdisk != NULL)
2047         disk_free(newdisk);
2048
2049     if (curslice != NULL)
2050         slice_free(curslice);
2050
2051     if (slice != NULL)
2052         slice_free(slice);
2052
2053     if (disks != NULL)
2054         disk_free(disks);
2055
2056     if (newdisk != NULL)
2057         disk_free(newdisk);
2058
2059     if (curslice != NULL)
2060         slice_free(curslice);
2060
2061     if (slice != NULL)
2062         slice_free(slice);
2062
2063     if (disks != NULL)
2064         disk_free(disks);
2065
2066     if (newdisk != NULL)
2067         disk_free(newdisk);
2068
2069     if (curslice != NULL)
2070         slice_free(curslice);
2070
2071     if (slice != NULL)
2072         slice_free(slice);
2072
2073     if (disks != NULL)
2074         disk_free(disks);
2075
2076     if (newdisk != NULL)
2077         disk_free(newdisk);
2078
2079     if (curslice != NULL)
2080         slice_free(curslice);
2080
2081     if (slice != NULL)
2082         slice_free(slice);
2082
2083     if (disks != NULL)
2084         disk_free(disks);
2085
2086     if (newdisk != NULL)
2087         disk_free(newdisk);
2088
2089     if (curslice != NULL)
2090         slice_free(curslice);
2090
2091     if (slice != NULL)
2092         slice_free(slice);
2092
2093     if (disks != NULL)
2094         disk_free(disks);
2095
2096     if (newdisk != NULL)
2097         disk_free(newdisk);
2098
2099     if (curslice != NULL)
2100         slice_free(curslice);
2100
2101     if (slice != NULL)
2102         slice_free(slice);
2102
2103     if (disks != NULL)
2104         disk_free(disks);
2105
2106     if (newdisk != NULL)
2107         disk_free(newdisk);
2108
2109     if (curslice != NULL)
2110         slice_free(curslice);
2110
2111     if (slice != NULL)
2112         slice_free(slice);
2112
2113     if (disks != NULL)
2114         disk_free(disks);
2115
2116     if (newdisk != NULL)
2117         disk_free(newdisk);
2118
2119     if (curslice != NULL)
2120         slice_free(curslice);
2120
2121     if (slice != NULL)
2122         slice_free(slice);
2122
2123     if (disks != NULL)
2124         disk_free(disks);
2125
2126     if (newdisk != NULL)
2127         disk_free(newdisk);
2128
2129     if (curslice != NULL)
2130         slice_free(curslice);
2130
2131     if (slice != NULL)
2132         slice_free(slice);
2132
2133     if (disks != NULL)
2134         disk_free(disks);
2135
2136     if (newdisk != NULL)
2137         disk_free(newdisk);
2138
2139     if (curslice != NULL)
2140         slice_free(curslice);
2140
2141     if (slice != NULL)
2142         slice_free(slice);
2142
2143     if (disks != NULL)
2144         disk_free(disks);
2145
2146     if (newdisk != NULL)
2147         disk_free(newdisk);
2148
2149     if (curslice != NULL)
2150         slice_free(curslice);
2150
2151     if (slice != NULL)
2152         slice_free(slice);
2152
2153     if (disks != NULL)
2154         disk_free(disks);
2155
2156     if (newdisk != NULL)
2157         disk_free(newdisk);
2158
2159     if (curslice != NULL)
2160         slice_free(curslice);
2160
2161     if (slice != NULL)
2162         slice_free(slice);
2162
2163     if (disks != NULL)
2164         disk_free(disks);
2165
2166     if (newdisk != NULL)
2167         disk_free(newdisk);
2168
2169     if (curslice != NULL)
2170         slice_free(curslice);
2170
2171     if (slice != NULL)
2172         slice_free(slice);
2172
2173     if (disks != NULL)
2174         disk_free(disks);
2175
2176     if (newdisk != NULL)
2177         disk_free(newdisk);
2178
2179     if (curslice != NULL)
2180         slice_free(curslice);
2180
2181     if (slice != NULL)
2182         slice_free(slice);
2182
2183     if (disks != NULL)
2184         disk_free(disks);
2185
2186     if (newdisk != NULL)
2187         disk_free(newdisk);
2188
2189     if (curslice != NULL)
2190         slice_free(curslice);
2190
2191     if (slice != NULL)
2192         slice_free(slice);
2192
2193     if (disks != NULL)
2194         disk_free(disks);
2195
2196     if (newdisk != NULL)
2197         disk_free(newdisk);
2198
2199     if (curslice != NULL)
2200         slice_free(curslice);
2200
2201     if (slice != NULL)
2202         slice_free(slice);
2202
2203     if (disks != NULL)
2204         disk_free(disks);
2205
2206     if (newdisk != NULL)
2207         disk_free(newdisk);
2208
2209     if (curslice != NULL)
2210         slice_free(curslice);
2210
2211     if (slice != NULL)
2212         slice_free(slice);
2212
2213     if (disks != NULL)
2214         disk_free(disks);
2215
2216     if (newdisk != NULL)
2217         disk_free(newdisk);
2218
2219     if (curslice != NULL)
2220         slice_free(curslice);
2220
2221     if (slice != NULL)
2222         slice_free(slice);
2222
2223     if (disks != NULL)
2224         disk_free(disks);
2225
2226     if (newdisk != NULL)
2227         disk_free(newdisk);
2228
2229     if (curslice != NULL)
2230         slice_free(curslice);
2230
2231     if (slice != NULL)
2232         slice_free(slice);
2232
2233     if (disks != NULL)
2234         disk_free(disks);
2235
2236     if (newdisk != NULL)
2237         disk_free(newdisk);
2238
2239     if (curslice != NULL)
2240         slice_free(curslice);
2240
2241     if (slice != NULL)
2242         slice_free(slice);
2242
2243     if (disks != NULL)
2244         disk_free(disks);
2245
2246     if (newdisk != NULL)
2247         disk_free(newdisk);
2248
2249     if (curslice != NULL)
2250         slice_free(curslice);
2250
2251     if (slice != NULL)
2252         slice_free(slice);
2252
2253     if (disks != NULL)
2254         disk_free(disks);
2255
2256     if (newdisk != NULL)
2257         disk_free(newdisk);
2258
2259     if (curslice != NULL)
2260         slice_free(curslice);
2260
2261     if (slice != NULL)
2262         slice_free(slice);
2262
2263     if (disks != NULL)
2264         disk_free(disks);
2265
2266     if (newdisk != NULL)
2267         disk_free(newdisk);
2268
2269     if (curslice != NULL)
2270         slice_free(curslice);
2270
2271     if
```

```

1283         * create a thread pool to do all of this in parallel;
1284         * choose double the number of processors; we hold a lot
1285         * of locks in the kernel, so going beyond this doesn't
1286         * buy us much. Each disk (and any slices it might have)
1287         * is handled inside a single thread.
1288     754         * rn_nozpool is not protected, so this is racy in that
1289     755         * multiple tasks could decide that the same slice can
1290     756         * not hold a zpool, which is benign. Also choose
1291     757         * double the number of processors; we hold a lot of
1292     758         * locks in the kernel, so going beyond this doesn't
1293     759         * buy us much.
1294     */
1295     t = tpool_create(1, 2 * sysconf(_SC_NPROCESSORS_ONLN),
1296                      0, NULL);
1297     for (curdisk = disks; curdisk; curdisk = curdisk->dn_next)
1298         (void) tpool_dispatch(t, zpool_open_func, curdisk);
1299     for (slice = avl_first(&slice_cache); slice;
1300          (slice = avl_walk(&slice_cache, slice,
1301                           AVL_AFTER)))
1302         (void) tpool_dispatch(t, zpool_open_func, slice);
1303     tpool_wait(t);
1304     tpool_destroy(t);

1305     curdisk = disks;
1306     while (curdisk != NULL) {
1307         nvlist_t *config;
1308         disk_node_t *prevdisk;

1309         /*
1310          * If the device has slices we examine the config on
1311          * each of those. If not we use the config directly
1312          * from the device instead.
1313          */
1314         curslice = curdisk->dn_slices;

1315         if (curslice != NULL)
1316             config = curslice->sn_config;
1317         else
1318             config = curdisk->dn_config;

1319         do {
1320             cookie = NULL;
1321             while ((slice = avl_destroy_nodes(&slice_cache,
1322                                              &cookie)) != NULL) {
1323                 if (slice->rn_config != NULL) {
1324                     nvlist_t *config = slice->rn_config;
1325                     boolean_t matched = B_TRUE;

1326                     if (config == NULL)
1327                         goto next;

1328 #endif /* ! codereview */
1329             if (iarg->poolname != NULL) {
1330                 char *pname;

1331                 matched = nvlist_lookup_string(config,
1332                                               ZPOOL_CONFIG_POOL_NAME,
1333                                               &pname) == 0 &&
1334                     strcmp(iarg->poolname, pname) == 0;
1335             } else if (iarg->guid != 0) {
1336                 uint64_t this_guid;

1337                 matched = nvlist_lookup_uint64(config,
1338                                               ZPOOL_CONFIG_POOL_GUID,
1339                                               &this_guid) == 0 &&
1340                     iarg->guid == this_guid;

```

```

1341                                         }
1342 #endif /* ! codereview */
1343         }
1344         if (!matched) {
1345             nvlist_free(config);
1346             goto next;
1347             config = NULL;
1348             continue;
1349     }

1350     /* use the non-raw path for the config */
1351     if (curslice != NULL)
1352         (void) snprintf(end, pathleft, "%s%s",
1353                        curdisk->dn_name,
1354                        curslice->sn_name);
1355     else
1356         (void) strlcpy(end, curdisk->dn_name,
1357                        pathleft);
1358     (void) strlcpy(end, slice->rn_name, pathleft);
1359     if (add_config(hdl, &pools, path, config) != 0)
1360         goto error;

1361     next:
1362     /*
1363      * If we're looking at slices free this one
1364      * and go move onto the next.
1365      */
1366     if (curslice != NULL) {
1367         slice_node_t *prevslice;
1368         prevslice = curslice;
1369         curslice = curslice->sn_next;
1370         free(prevslice->sn_name);
1371         free(prevslice);
1372         if (curslice != NULL) {
1373             config = curslice->sn_config;
1374         }
1375 #endif /* ! codereview */
1376     }
1377     } while (curslice != NULL);

1378     /*
1379      * Free this disk and move onto the next one.
1380      */
1381     prevdisk = curdisk;
1382     curdisk = curdisk->dn_next;

1383     free(prevdisk->dn_name);
1384     free(prevdisk);
1385     free(slice->rn_name);
1386     free(slice);
1387 }

1388     avl_destroy(&slice_cache);

1389     (void) closedir(dirp);
1390     dirp = NULL;
1391 error:
1392     for (pe = pools.pools; pe != NULL; pe = penext) {
1393         penext = pe->pe_next;

```

```
1394     for (ve = pe->pe_vdevs; ve != NULL; ve = venext) {
1395         venext = ve->ve_next;
1396         for (ce = ve->ve_configs; ce != NULL; ce = cenext) {
1397             cenext = ce->ce_next;
1398             if (ce->ce_config)
1399                 nvlist_free(ce->ce_config);
1400             free(ce);
1401         }
1402         free(ve);
1403     }
1404     free(pe);
1405 }
1406
1407 for (ne = pools.names; ne != NULL; ne = nenext) {
1408     nenext = ne->ne_next;
1409     if (ne->ne_name)
1410         free(ne->ne_name);
1411     free(ne);
1412 }
1413
1414 if (dirp)
1415     (void) closedir(dirp);
1416
1417 return (ret);
1418 }
```

unchanged\_portion\_omitted

```
new/usr/src/lib/libzfs/common/libzfs_util.c
```

```
*****
36417 Wed May 14 12:03:03 2014
new/usr/src/lib/libzfs/common/libzfs_util.c
zpool import is braindead
*****
```

```
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) 2013, Joyent, Inc. All rights reserved.
25 * Copyright (c) 2012 by Delphix. All rights reserved.
26 * Copyright 2014 RackTop Systems.
27 #endif /* ! codereview */
28 */
```

```
30 /*
31 * Internal utility routines for the ZFS library.
32 */
```

```
34 #include <errno.h>
35 #include <fcntl.h>
36 #include <libintl.h>
37 #include <stdarg.h>
38 #include <stdio.h>
39 #include <stdlib.h>
40 #include <strings.h>
41 #include <unistd.h>
42 #include <ctype.h>
43 #include <math.h>
44 #include <sys/mnttab.h>
45 #include <sys/mntent.h>
46 #include <sys/types.h>
```

```
48 #include <libzfs.h>
49 #include <libzfs_core.h>
```

```
51 #include "libzfs_impl.h"
52 #include "zfs_prop.h"
53 #include "zfeature_common.h"
```

```
55 int
56 libzfs_errno(libzfs_handle_t *hdl)
57 {
58     return (hdl->libzfs_error);
59 }
```

```
61 const char *
```

```
1
```

```
new/usr/src/lib/libzfs/common/libzfs_util.c
```

```
62 libzfs_error_action(libzfs_handle_t *hdl)
63 {
64     return (hdl->libzfs_action);
65 }
```

```
66 const char *
67 libzfs_error_description(libzfs_handle_t *hdl)
68 {
69     if (hdl->libzfs_desc[0] != '\0')
70         return (hdl->libzfs_desc);
```

```
73     switch (hdl->libzfs_error) {
74     case EZFS_NOMEM:
75         return (dgettext(TEXT_DOMAIN, "out of memory"));
76     case EZFS_BADPROP:
77         return (dgettext(TEXT_DOMAIN, "invalid property value"));
78     case EZFS_PROP_READONLY:
79         return (dgettext(TEXT_DOMAIN, "read-only property"));
80     case EZFS_PROPTYPE:
81         return (dgettext(TEXT_DOMAIN, "property doesn't apply to "
82                         "datasets of this type"));
83     case EZFS_PROPNONINHERIT:
84         return (dgettext(TEXT_DOMAIN, "property cannot be inherited"));
85     case EZFS_PROPSPACE:
86         return (dgettext(TEXT_DOMAIN, "invalid quota or reservation"));
87     case EZFS_BADTYPE:
88         return (dgettext(TEXT_DOMAIN, "operation not applicable to "
89                         "datasets of this type"));
90     case EZFS_BUSY:
91         return (dgettext(TEXT_DOMAIN, "pool or dataset is busy"));
92     case EZFS_EXISTS:
93         return (dgettext(TEXT_DOMAIN, "pool or dataset exists"));
94     case EZFS_NOENT:
95         return (dgettext(TEXT_DOMAIN, "no such pool or dataset"));
96     case EZFS_BADSTREAM:
97         return (dgettext(TEXT_DOMAIN, "invalid backup stream"));
98     case EZFS_DSREADONLY:
99         return (dgettext(TEXT_DOMAIN, "dataset is read-only"));
100    case EZFS_VOLTOOBIG:
101        return (dgettext(TEXT_DOMAIN, "volume size exceeds limit for "
102                      "this system"));
103    case EZFS_INVALIDNAME:
104        return (dgettext(TEXT_DOMAIN, "invalid name"));
105    case EZFS_BADRESTORE:
106        return (dgettext(TEXT_DOMAIN, "unable to restore to "
107                      "destination"));
108    case EZFS_BADBACKUP:
109        return (dgettext(TEXT_DOMAIN, "backup failed"));
110    case EZFS_BADTARGET:
111        return (dgettext(TEXT_DOMAIN, "invalid target vdev"));
112    case EZFS_NODEVICE:
113        return (dgettext(TEXT_DOMAIN, "no such device in pool"));
114    case EZFS_BADDEV:
115        return (dgettext(TEXT_DOMAIN, "invalid device"));
116    case EZFS_NOREPLICAS:
117        return (dgettext(TEXT_DOMAIN, "no valid replicas"));
118    case EZFS_RESILVERING:
119        return (dgettext(TEXT_DOMAIN, "currently resilvering"));
120    case EZFS_BADVERSION:
121        return (dgettext(TEXT_DOMAIN, "unsupported version or "
122                      "feature"));
123    case EZFS_POOLUNAVAIL:
124        return (dgettext(TEXT_DOMAIN, "pool is unavailable"));
125    case EZFS_DEVOVERFLOW:
126        return (dgettext(TEXT_DOMAIN, "too many devices in one vdev"));
127    case EZFS_BADPATH:
```

```
2
```

```

128     return (dgettext(TEXT_DOMAIN, "must be an absolute path"));
129 case EZFS_CROSSTARGET:
130     return (dgettext(TEXT_DOMAIN, "operation crosses datasets or "
131                     "pools"));
132 case EZFS_ZONED:
133     return (dgettext(TEXT_DOMAIN, "dataset in use by local zone"));
134 case EZFS_MOUNTFAILED:
135     return (dgettext(TEXT_DOMAIN, "mount failed"));
136 case EZFS_UNMOUNTFAILED:
137     return (dgettext(TEXT_DOMAIN, "umount failed"));
138 case EZFS_UNSHARENFSFAILED:
139     return (dgettext(TEXT_DOMAIN, "unshare(1M) failed"));
140 case EZFS_SHARENFSFAILED:
141     return (dgettext(TEXT_DOMAIN, "share(1M) failed"));
142 case EZFS_UNSHARESMBFAILED:
143     return (dgettext(TEXT_DOMAIN, "smb remove share failed"));
144 case EZFS_SHARESMBFAILED:
145     return (dgettext(TEXT_DOMAIN, "smb add share failed"));
146 case EZFS_PERM:
147     return (dgettext(TEXT_DOMAIN, "permission denied"));
148 case EZFS_NOSPC:
149     return (dgettext(TEXT_DOMAIN, "out of space"));
150 case EZFS_FAULT:
151     return (dgettext(TEXT_DOMAIN, "bad address"));
152 case EZFS_IO:
153     return (dgettext(TEXT_DOMAIN, "I/O error"));
154 case EZFS_INTR:
155     return (dgettext(TEXT_DOMAIN, "signal received"));
156 case EZFS_ISSPARE:
157     return (dgettext(TEXT_DOMAIN, "device is reserved as a hot "
158                      "spare"));
159 case EZFS_INVALCONFIG:
160     return (dgettext(TEXT_DOMAIN, "invalid vdev configuration"));
161 case EZFS_RECURSIVE:
162     return (dgettext(TEXT_DOMAIN, "recursive dataset dependency"));
163 case EZFS_NOHISTORY:
164     return (dgettext(TEXT_DOMAIN, "no history available"));
165 case EZFS_POOLPROPS:
166     return (dgettext(TEXT_DOMAIN, "failed to retrieve "
167                      "pool properties"));
168 case EZFS_POOL_NOTSUP:
169     return (dgettext(TEXT_DOMAIN, "operation not supported "
170                      "on this type of pool"));
171 case EZFS_POOL_INVALARG:
172     return (dgettext(TEXT_DOMAIN, "invalid argument for "
173                      "this pool operation"));
174 case EZFS_NAMETOOLONG:
175     return (dgettext(TEXT_DOMAIN, "dataset name is too long"));
176 case EZFS_OPENFAILED:
177     return (dgettext(TEXT_DOMAIN, "open failed"));
178 case EZFS_NOCAP:
179     return (dgettext(TEXT_DOMAIN,
180                     "disk capacity information could not be retrieved"));
181 case EZFS_LABELFAILED:
182     return (dgettext(TEXT_DOMAIN, "write of label failed"));
183 case EZFS_BADWHO:
184     return (dgettext(TEXT_DOMAIN, "invalid user/group"));
185 case EZFS_BADPERM:
186     return (dgettext(TEXT_DOMAIN, "invalid permission"));
187 case EZFS_BADPERMSET:
188     return (dgettext(TEXT_DOMAIN, "invalid permission set name"));
189 case EZFS_NODELEGATION:
190     return (dgettext(TEXT_DOMAIN, "delegated administration is "
191                      "disabled on pool"));
192 case EZFS_BADCACHE:
193     return (dgettext(TEXT_DOMAIN, "invalid or missing cache file"));

```

```

194     case EZFS_ISL2CACHE:
195         return (dgettext(TEXT_DOMAIN, "device is in use as a cache"));
196 case EZFS_VDEVNOTSUP:
197     return (dgettext(TEXT_DOMAIN, "vdev specification is not "
198                     "supported"));
199 case EZFS_NOTSUP:
200     return (dgettext(TEXT_DOMAIN, "operation not supported "
201                     "on this dataset"));
202 case EZFS_ACTIVE_SPARE:
203     return (dgettext(TEXT_DOMAIN, "pool has active shared spare "
204                     "device"));
205 case EZFS_UNPLAYED_LOGS:
206     return (dgettext(TEXT_DOMAIN, "log device has unplayed intent "
207                     "logs"));
208 case EZFS_REFTAG_RELEASE:
209     return (dgettext(TEXT_DOMAIN, "no such tag on this dataset"));
210 case EZFS_REFTAG_HOLD:
211     return (dgettext(TEXT_DOMAIN, "tag already exists on this "
212                     "dataset"));
213 case EZFS_TAGTOOLONG:
214     return (dgettext(TEXT_DOMAIN, "tag too long"));
215 case EZFS_PIPEFAILED:
216     return (dgettext(TEXT_DOMAIN, "pipe create failed"));
217 case EZFS_THREADCREATEFAILED:
218     return (dgettext(TEXT_DOMAIN, "thread create failed"));
219 case EZFS_POSTSPLIT_ONLINE:
220     return (dgettext(TEXT_DOMAIN, "disk was split from this pool "
221                     "into a new one"));
222 case EZFS_SCRUBBING:
223     return (dgettext(TEXT_DOMAIN, "currently scrubbing; "
224                     "use 'zpool scrub -s' to cancel current scrub"));
225 case EZFS_NO_SCRUB:
226     return (dgettext(TEXT_DOMAIN, "there is no active scrub"));
227 case EZFS_DIFF:
228     return (dgettext(TEXT_DOMAIN, "unable to generate diffs"));
229 case EZFS_DIFFDATA:
230     return (dgettext(TEXT_DOMAIN, "invalid diff data"));
231 case EZFS_POOLREADONLY:
232     return (dgettext(TEXT_DOMAIN, "pool is read-only"));
233 case EZFS_UNKNOWN:
234     return (dgettext(TEXT_DOMAIN, "unknown error"));
235 default:
236     assert(hdl->libzfs_error == 0);
237     return (dgettext(TEXT_DOMAIN, "no error"));
238 }
239 }

240 /*PRINTFLIKE2*/
241 void
242 zfs_error_aux(libzfs_handle_t *hdl, const char *fmt, ...)
243 {
244     va_list ap;
245     va_start(ap, fmt);
246     va_start(ap, fmt);
247     (void) vsnprintf(hdl->libzfs_desc, sizeof(hdl->libzfs_desc),
248                      fmt, ap);
249     hdl->libzfs_desc_active = 1;
250     va_end(ap);
251     va_end(ap);
252 }
253

254 static void
255 zfs_verror(libzfs_handle_t *hdl, int error, const char *fmt, va_list ap)
256 {
257     (void) vsnprintf(hdl->libzfs_action, sizeof(hdl->libzfs_action),
258

```

new/usr/src/lib/libzfs/common/libzfs\_util.c

5

```
260     fmt, ap);
261     hdl->libzfs_error = error;
263
264     if (hdl->libzfs_desc_active)
265         hdl->libzfs_desc_active = 0;
266     else
267         hdl->libzfs_desc[0] = '\0';
268
269     if (hdl->libzfs_printerr) {
270         if (error == EZFS_UNKNOWN) {
271             (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "internal "
272                           "error: %s\n"), libzfs_error_description(hdl));
273             abort();
274         }
275
276         (void) fprintf(stderr, "%s: %s\n", hdl->libzfs_action,
277                       libzfs_error_description(hdl));
278         if (error == EZFS_NOMEM)
279             exit(1);
280     }
281 }
282 int
283 zfs_error(libzfs_handle_t *hdl, int error, const char *msg)
284 {
285     return (zfs_error_fmt(hdl, error, "%s", msg));
286 }
287
288 /*PRINTFLIKE3*/
289 int
290 zfs_error_fmt(libzfs_handle_t *hdl, int error, const char *fmt, ...)
291 {
292     va_list ap;
293
294     va_start(ap, fmt);
295
296     zfs_verror(hdl, error, fmt, ap);
297
298     va_end(ap);
299
300     return (-1);
301 }
302
303 static int
304 zfs_common_error(libzfs_handle_t *hdl, int error, const char *fmt,
305                   va_list ap)
306 {
307     switch (error) {
308     case EPERM:
309     case EACCES:
310         zfs_verror(hdl, EZFS_PERM, fmt, ap);
311         return (-1);
312
313     case ECANCELED:
314         zfs_verror(hdl, EZFS_NODELEGATION, fmt, ap);
315         return (-1);
316
317     case EIO:
318         zfs_verror(hdl, EZFS_IO, fmt, ap);
319         return (-1);
320
321     case EFAULT:
322         zfs_verror(hdl, EZFS_FAULT, fmt, ap);
323         return (-1);
324
325     case EINTR:
```

new/usr/src/lib/libzfs/common/libzfs\_util.c

```

326             zfs_verror(hdl, EZFS_INTR, fmt, ap);
327         }
328     }
329     return (0);
330 }
331 }

333 int
334 zfs_standard_error(libzfs_handle_t *hdl, int error, const char *msg)
335 {
336     return (zfs_standard_error_fmt(hdl, error, "%s", msg));
337 }

338 /*PRNTFLIKE3*/
339 int
340 zfs_standard_error_fmt(libzfs_handle_t *hdl, int error, const char *fmt, ...)
341 {
342     va_list ap;
343
344     va_start(ap, fmt);
345
346     if (zfs_common_error(hdl, error, fmt, ap) != 0) {
347         va_end(ap);
348         return (-1);
349     }
350
351     switch (error) {
352     case ENXIO:
353     case ENODEV:
354     case EPIPE:
355         zfs_verror(hdl, EZFS_IO, fmt, ap);
356         break;
357
358     case ENOENT:
359         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
360                         "dataset does not exist"));
361         zfs_verror(hdl, EZFS_NOENT, fmt, ap);
362         break;
363
364     case ENOSPC:
365     case EDQUOT:
366         zfs_verror(hdl, EZFS_NOSPC, fmt, ap);
367         return (-1);
368
369     case EEXIST:
370         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
371                         "dataset already exists"));
372         zfs_verror(hdl, EZFS_EXISTS, fmt, ap);
373         break;
374
375     case EBUSY:
376         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
377                         "dataset is busy"));
378         zfs_verror(hdl, EZFS_BUSY, fmt, ap);
379         break;
380
381     case EROFS:
382         zfs_verror(hdl, EZFS_POOLREADONLY, fmt, ap);
383         break;
384
385     case ENAMETOOLONG:
386         zfs_verror(hdl, EZFS_NAMETOOLONG, fmt, ap);
387         break;
388
389     case ENOTSUP:
390         zfs_verror(hdl, EZFS_BADVERSION, fmt, ap);
391         break;
392
393     case EAGAIN:
394         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,

```

```

392         "pool I/O is currently suspended"));
393         zfs_verror(hdl, EZFS_POOLUNAVAIL, fmt, ap);
394         break;
395     default:
396         zfs_error_aux(hdl, strerror(error));
397         zfs_verror(hdl, EZFS_UNKNOWN, fmt, ap);
398         break;
399     }
400
401     va_end(ap);
402     return (-1);
403 }
404
405 int
406 zpool_standard_error(libzfs_handle_t *hdl, int error, const char *msg)
407 {
408     return (zpool_standard_error_fmt(hdl, error, "%s", msg));
409 }
410
411 /*PRINTFLIKE3*/
412 int
413 zpool_standard_error_fmt(libzfs_handle_t *hdl, int error, const char *fmt, ...)
414 {
415     va_list ap;
416
417     va_start(ap, fmt);
418
419     if (zfs_common_error(hdl, error, fmt, ap) != 0) {
420         va_end(ap);
421         return (-1);
422     }
423
424     switch (error) {
425     case ENODEV:
426         zfs_verror(hdl, EZFS_NODEVICE, fmt, ap);
427         break;
428
429     case ENOENT:
430         zfs_error_aux(hdl,
431                     dgettext(TEXT_DOMAIN, "no such pool or dataset"));
432         zfs_verror(hdl, EZFS_NOENT, fmt, ap);
433         break;
434
435     case EEXIST:
436         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
437                     "pool already exists"));
438         zfs_verror(hdl, EZFS_EXISTS, fmt, ap);
439         break;
440
441     case EBUSY:
442         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "pool is busy"));
443         zfs_verror(hdl, EZFS_BUSY, fmt, ap);
444         break;
445
446     case ENXIO:
447         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
448                     "one or more devices is currently unavailable"));
449         zfs_verror(hdl, EZFS_BADDEV, fmt, ap);
450         break;
451
452     case ENAMETOOLONG:
453         zfs_verror(hdl, EZFS_DEVOVERFLOW, fmt, ap);
454         break;
455
456     case ENOTSUP:
457         zfs_verror(hdl, EZFS_POOL_NOTSUP, fmt, ap);

```

```

458         break;
459
460     case EINVAL:
461         zfs_verror(hdl, EZFS_POOL_INVALARG, fmt, ap);
462         break;
463
464     case ENOSPC:
465     case EDQUOT:
466         zfs_verror(hdl, EZFS_NOSPC, fmt, ap);
467         return (-1);
468
469     case EAGAIN:
470         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
471                     "pool I/O is currently suspended"));
472         zfs_verror(hdl, EZFS_POOLUNAVAIL, fmt, ap);
473         break;
474
475     case EROFS:
476         zfs_verror(hdl, EZFS_POOLREADONLY, fmt, ap);
477         break;
478
479     default:
480         zfs_error_aux(hdl, strerror(error));
481         zfs_verror(hdl, EZFS_UNKNOWN, fmt, ap);
482     }
483
484     va_end(ap);
485     return (-1);
486 }
487
488 /*
489  * Display an out of memory error message and abort the current program.
490  */
491 int
492 no_memory(libzfs_handle_t *hdl)
493 {
494     return (zfs_error(hdl, EZFS_NOMEM, "internal error"));
495 }
496
497 /*
498  * A safe form of malloc() which will die if the allocation fails.
499  */
500 void *
501 zfs_alloc(libzfs_handle_t *hdl, size_t size)
502 {
503     void *data;
504
505     if ((data = calloc(1, size)) == NULL)
506         (void) no_memory(hdl);
507
508     return (data);
509 }
510
511 /*
512  * A safe form of asprintf() which will die if the allocation fails.
513  */
514 /*PRINTFLIKE2*/
515 char *
516 zfs_asprintf(libzfs_handle_t *hdl, const char *fmt, ...)
517 {
518     va_list ap;
519     char *ret;
520     int err;
521
522     va_start(ap, fmt);

```

```

524     err = vasprintf(&ret, fmt, ap);
526     va_end(ap);
528     if (err < 0)
529         (void) no_memory(hdl);
531     return (ret);
532 }
534 /*
535  * A safe form of realloc(), which also zeroes newly allocated space.
536  */
537 void *
538 zfs_realloc(libzfs_handle_t *hdl, void *ptr, size_t oldsize, size_t newsize)
539 {
540     void *ret;
542     if ((ret = realloc(ptr, newsize)) == NULL) {
543         (void) no_memory(hdl);
544         return (NULL);
545     }
547     bzero((char *)ret + oldsize, (newsize - oldsize));
548     return (ret);
549 }
551 /*
552  * A safe form of strdup() which will die if the allocation fails.
553  */
554 char *
555 zfs_strdup(libzfs_handle_t *hdl, const char *str)
556 {
557     char *ret;
559     if ((ret = strdup(str)) == NULL)
560         (void) no_memory(hdl);
562     return (ret);
563 }
565 /*
566  * Convert a number to an appropriately human-readable output.
567  */
568 void
569 zfs_nicenum(uint64_t num, char *buf, size_t buflen)
570 {
571     uint64_t n = num;
572     int index = 0;
573     char u;
575     while (n >= 1024) {
576         n /= 1024;
577         index++;
578     }
580     u = "KMGTPE"[index];
582     if (index == 0) {
583         (void) snprintf(buf, buflen, "%llu", n);
584     } else if ((num & ((1ULL << 10 * index) - 1)) == 0) {
585         /*
586          * If this is an even multiple of the base, always display
587          * without any decimal precision.
588          */
589         (void) snprintf(buf, buflen, "%llu%c", n, u);

```

```

590     } else {
591         /*
592          * We want to choose a precision that reflects the best choice
593          * for fitting in 5 characters. This can get rather tricky when
594          * we have numbers that are very close to an order of magnitude.
595          * For example, when displaying 10239 (which is really 9.999K),
596          * we want only a single place of precision for 10.0K. We could
597          * develop some complex heuristics for this, but it's much
598          * easier just to try each combination in turn.
599         */
600         int i;
601         for (i = 2; i >= 0; i--) {
602             if (snprintf(buf, buflen, "%.f%c", i,
603                         (double)num / (1ULL << 10 * index), u) <= 5)
604                 break;
605         }
606     }
607 }
609 void
610 libzfs_print_on_error(libzfs_handle_t *hdl, boolean_t printerr)
611 {
612     hdl->libzfs_printer = printerr;
613 }
615 libzfs_handle_t *
616 libzfs_init(void)
617 {
618     libzfs_handle_t *hdl;
620     if ((hdl = calloc(1, sizeof (libzfs_handle_t))) == NULL) {
621         return (NULL);
622     }
624     if ((hdl->libzfs_fd = open(ZFS_DEV, O_RDWR)) < 0) {
625         free(hdl);
626         return (NULL);
627     }
629     if ((hdl->libzfs_mnttab = fopen(MNTTAB, "r")) == NULL) {
630         (void) close(hdl->libzfs_fd);
631         free(hdl);
632         return (NULL);
633     }
635     hdl->libzfs_sharetab = fopen("/etc/dfs/sharetab", "r");
637     if (libzfs_core_init() != 0) {
638         (void) close(hdl->libzfs_fd);
639         (void) fclose(hdl->libzfs_mnttab);
640         (void) fclose(hdl->libzfs_sharetab);
641         free(hdl);
642         return (NULL);
643     }
645     zfs_prop_init();
646     zpool_prop_init();
647     zpool_feature_init();
648     libzfs_mnttab_init(hdl);
650     return (hdl);
651 }
653 void
654 libzfs_fini(libzfs_handle_t *hdl)
655 {

```

```

656     (void) close(hdl->libzfs_fd);
657     if (hdl->libzfs_mnttab)
658         (void) fclose(hdl->libzfs_mnttab);
659     if (hdl->libzfs_sharetab)
660         (void) fclose(hdl->libzfs_sharetab);
661     zfs_uninit_libshare(hdl);
662     zpool_free_handles(hdl);
663     libzfs_fru_clear(hdl, B_TRUE);
664     namespace_clear(hdl);
665     libzfs_mnttab_fini(hdl);
666     libzfs_core_fini();
667     free(hdl);
668 }

670 libzfs_handle_t *
671 zpool_get_handle(zpool_handle_t *zhp)
672 {
673     return (zhp->zpool_hdl);
674 }

676 libzfs_handle_t *
677 zfs_get_handle(zfs_handle_t *zhp)
678 {
679     return (zhp->zfs_hdl);
680 }

682 zpool_handle_t *
683 zfs_get_pool_handle(const zfs_handle_t *zhp)
684 {
685     return (zhp->zpool_hdl);
686 }

688 /*
689  * Given a name, determine whether or not it's a valid path
690  * (starts with '/' or "./"). If so, walk the mnttab trying
691  * to match the device number. If not, treat the path as an
692  * fs/vol/snap name.
693 */
694 zfs_handle_t *
695 zfs_path_to_zhandle(libzfs_handle_t *hdl, char *path, zfs_type_t argtype)
696 {
697     struct stat64 statbuf;
698     struct extmnttab entry;
699     int ret;

700     if (path[0] != '/' && strncmp(path, "./", strlen("./")) != 0) {
701         /*
702          * It's not a valid path, assume it's a name of type 'argtype'.
703          */
704         return (zfs_open(hdl, path, argtype));
705     }
706 }

708     if (stat64(path, &statbuf) != 0)
709         (void) fprintf(stderr, "%s: %s\n", path, strerror(errno));
710     return (NULL);
711 }

713     rewind(hdl->libzfs_mnttab);
714     while ((ret = getextmntent(hdl->libzfs_mnttab, &entry, 0)) == 0) {
715         if (makedevice(entry.mnt_major, entry.mnt_minor) ==
716             statbuf.st_dev) {
717             break;
718         }
719     }
720     if (ret != 0) {
721         return (NULL);

```

```

722     }

724     if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0) {
725         (void) fprintf(stderr, gettext("%s: not a ZFS filesystem\n"),
726                     path);
727         return (NULL);
728     }

730     return (zfs_open(hdl, entry.mnt_special, ZFS_TYPE_FILESYSTEM));
731 }

733 /*
734  * Initialize the zc_nvlist_dst member to prepare for receiving an nvlist from
735  * an ioctl().
736 */
737 int
738 zcmd_alloc_dst_nvlist(libzfs_handle_t *hdl, zfs_cmd_t *zc, size_t len)
739 {
740     if (len == 0)
741         len = 16 * 1024;
742     zc->zc_nvlist_dst_size = len;
743     if ((zc->zc_nvlist_dst = (uint64_t)(uintptr_t)
744          zfs_alloc(hdl, zc->zc_nvlist_dst_size)) == NULL)
745         return (-1);

747     return (0);
748 }

750 /*
751  * Called when an ioctl() which returns an nvlist fails with ENOMEM. This will
752  * expand the nvlist to the size specified in 'zc_nvlist_dst_size', which was
753  * filled in by the kernel to indicate the actual required size.
754 */
755 int
756 zcmd_expand_dst_nvlist(libzfs_handle_t *hdl, zfs_cmd_t *zc)
757 {
758     free((void *)zc->zc_nvlist_dst);
759     if ((zc->zc_nvlist_dst = (uint64_t)(uintptr_t)
760          zfs_alloc(hdl, zc->zc_nvlist_dst_size)) == NULL)
761         return (-1);

764     return (0);
765 }

767 /*
768  * Called to free the src and dst nvlists stored in the command structure.
769  */
770 void
771 zcmd_free_nvlists(zfs_cmd_t *zc)
772 {
773     free((void *)zc->zc_nvlist_conf);
774     free((void *)zc->zc_nvlist_src);
775     free((void *)zc->zc_nvlist_dst);
776 }

778 static int
779 zcmd_write_nvlist_com(libzfs_handle_t *hdl, uint64_t *outnv, uint64_t *outlen,
780                      nvlist_t *nvl)
781 {
782     char *packed;
783     size_t len;

785     verify(nvlist_size(nvl, &len, NV_ENCODE_NATIVE) == 0);

787     if ((packed = zfs_alloc(hdl, len)) == NULL)

```

```

788         return (-1);
790     verify(nvlist_pack(nvl, &packed, &len, NV_ENCODE_NATIVE, 0) == 0);
792     *outnv = (uint64_t)(uintptr_t)packed;
793     *outlen = len;
795     return (0);
796 }

798 int
799 zcmd_write_conf_nvlist(libzfs_handle_t *hdl, zfs_cmd_t *zc, nvlist_t *nvl)
800 {
801     return (zcmd_write_nvlist_com(hdl, &zc->zc_nvlist_conf,
802         &zc->zc_nvlist_conf_size, nvl));
803 }

805 int
806 zcmd_write_src_nvlist(libzfs_handle_t *hdl, zfs_cmd_t *zc, nvlist_t *nvl)
807 {
808     return (zcmd_write_nvlist_com(hdl, &zc->zc_nvlist_src,
809         &zc->zc_nvlist_src_size, nvl));
810 }

812 /**
813 * Unpacks an nvlist from the ZFS ioctl command structure.
814 */
815 int
816 zcmd_read_dst_nvlist(libzfs_handle_t *hdl, zfs_cmd_t *zc, nvlist_t **nvp)
817 {
818     if (nvlist_unpack((void *) (uintptr_t) zc->zc_nvlist_dst,
819         zc->zc_nvlist_dst_size, nvp, 0) != 0)
820         return (no_memory(hdl));
821
822     return (0);
823 }

825 int
826 zfs_ioctl(libzfs_handle_t *hdl, int request, zfs_cmd_t *zc)
827 {
828     return (ioctl(hdl->libzfs_fd, request, zc));
829 }

831 /**
832 * =====
833 * API shared by zfs and zpool property management
834 * =====
835 */

837 static void
838 zprop_print_headers(zprop_get_cbdata_t *cbp, zfs_type_t type)
839 {
840     zprop_list_t *pl = cbp->cb_proplist;
841     int i;
842     char *title;
843     size_t len;

845     cbp->cb_first = B_FALSE;
846     if (cbp->cb_scripted)
847         return;
848
849     /*
850      * Start with the length of the column headers.
851     */
852     cbp->cb_colwidths[GET_COL_NAME] = strlen(dgettext(TEXT_DOMAIN, "NAME"));
853     cbp->cb_colwidths[GET_COL_PROPERTY] = strlen(dgettext(TEXT_DOMAIN,

```

```

854         "PROPERTY"));
855     cbp->cb_colwidths[GET_COL_VALUE] = strlen(dgettext(TEXT_DOMAIN,
856         "VALUE"));
857     cbp->cb_colwidths[GET_COL_RECV] = strlen(dgettext(TEXT_DOMAIN,
858         "RECEIVED"));
859     cbp->cb_colwidths[GET_COL_SOURCE] = strlen(dgettext(TEXT_DOMAIN,
860         "SOURCE"));

862     /* first property is always NAME */
863     assert(cbp->cb_proplist->pl_prop ==
864         ((type == ZFS_TYPE_POOL) ? ZPOOL_PROP_NAME : ZFS_PROP_NAME));
865
866     /*
867      * Go through and calculate the widths for each column. For the
868      * 'source' column, we kludge it up by taking the worst-case scenario of
869      * inheriting from the longest name. This is acceptable because in the
870      * majority of cases 'SOURCE' is the last column displayed, and we don't
871      * use the width anyway. Note that the 'VALUE' column can be oversized,
872      * if the name of the property is much longer than any values we find.
873     */
874     for (pl = cbp->cb_proplist; pl != NULL; pl = pl->pl_next) {
875         /*
876          * 'PROPERTY' column
877        */
878         if (pl->pl_prop != ZPROP_INVALID) {
879             const char *propname = (type == ZFS_TYPE_POOL) ?
880                 zpool_prop_to_name(pl->pl_prop) :
881                 zfs_prop_to_name(pl->pl_prop);
882
883             len = strlen(propname);
884             if (len > cbp->cb_colwidths[GET_COL_PROPERTY])
885                 cbp->cb_colwidths[GET_COL_PROPERTY] = len;
886         } else {
887             len = strlen(pl->pl_user_prop);
888             if (len > cbp->cb_colwidths[GET_COL_PROPERTY])
889                 cbp->cb_colwidths[GET_COL_PROPERTY] = len;
890         }
891
892         /*
893          * 'VALUE' column. The first property is always the 'name'
894          * property that was tacked on either by /sbin/zfs's
895          * zfs_do_get() or when calling zprop_expand_list(), so we
896          * ignore its width. If the user specified the name property
897          * to display, then it will be later in the list in any case.
898        */
899         if (pl != cbp->cb_proplist &&
900             pl->pl_width > cbp->cb_colwidths[GET_COL_VALUE])
901             cbp->cb_colwidths[GET_COL_VALUE] = pl->pl_width;
902
903         /*
904          * 'RECEIVED' column.
905        */
906         if (pl != cbp->cb_proplist &&
907             pl->pl_recv_width > cbp->cb_colwidths[GET_COL_RECV])
908             cbp->cb_colwidths[GET_COL_RECV] = pl->pl_recv_width;
909
910         /*
911          * 'NAME' and 'SOURCE' columns
912        */
913         if (pl->pl_prop == (type == ZFS_TYPE_POOL ? ZPOOL_PROP_NAME :
914             ZFS_PROP_NAME) &&
915             pl->pl_width > cbp->cb_colwidths[GET_COL_NAME]) {
916             cbp->cb_colwidths[GET_COL_NAME] = pl->pl_width;
917             cbp->cb_colwidths[GET_COL_SOURCE] = pl->pl_width +
918                 strlen(dgettext(TEXT_DOMAIN, "inherited from"));
919         }
920     }
921 }
```

```

920 /*
921  * Now go through and print the headers.
922  */
923 for (i = 0; i < ZFS_GET_NCOLS; i++) {
924     switch (cbp->cb_columns[i]) {
925         case GET_COL_NAME:
926             title = dgettext(TEXT_DOMAIN, "NAME");
927             break;
928         case GET_COL_PROPERTY:
929             title = dgettext(TEXT_DOMAIN, "PROPERTY");
930             break;
931         case GET_COL_VALUE:
932             title = dgettext(TEXT_DOMAIN, "VALUE");
933             break;
934         case GET_COL_RECVD:
935             title = dgettext(TEXT_DOMAIN, "RECEIVED");
936             break;
937         case GET_COL_SOURCE:
938             title = dgettext(TEXT_DOMAIN, "SOURCE");
939             break;
940         default:
941             title = NULL;
942     }
943
944     if (title != NULL) {
945         if (i == (ZFS_GET_NCOLS - 1) ||
946             cbp->cb_columns[i + 1] == GET_COL_NONE)
947             (void) printf("%s", title);
948         else
949             (void) printf("%-*s ",
950                           cbp->cb_colwidths[cbp->cb_columns[i]],
951                           title);
952     }
953 }
954 (void) printf("\n");
955 }

956 /*
957  * Display a single line of output, according to the settings in the callback
958  * structure.
959  */
960 */
961 void
962 zprop_print_one_property(const char *name, zprop_get_cbdata_t *cbp,
963     const char *propname, const char *value, zprop_source_t sourcetype,
964     const char *source, const char *recvд_value)
965 {
966     int i;
967     const char *str = NULL;
968     const char *str;
969     char buf[128];

970     /*
971      * Ignore those source types that the user has chosen to ignore.
972      */
973     if ((sourcetype & cbp->cb_sources) == 0)
974         return;

975     if (cbp->cb_first)
976         zprop_print_headers(cbp, cbp->cb_type);

977     for (i = 0; i < ZFS_GET_NCOLS; i++) {
978         switch (cbp->cb_columns[i]) {
979             case GET_COL_NAME:
980                 str = name;
981                 break;
982             case GET_COL_SOURCE:
983                 str = source;
984                 break;
985             case GET_COL_RECVD:
986                 str = recvд_value;
987                 break;
988             case GET_COL_PROPERTY:
989                 str = propname;
990                 break;
991             case GET_COL_VALUE:
992                 str = value;
993                 break;
994             default:
995                 str = NULL;
996             }
997
998         if (str != NULL)
999             (void) printf("%-*s ", cbp->cb_colwidths[i], str);
1000     }
1001
1002     (void) printf("\n");
1003 }
```

```

985     case GET_COL_PROPERTY:
986         str = propname;
987         break;
988
989     case GET_COL_VALUE:
990         str = value;
991         break;
992
993     case GET_COL_SOURCE:
994         switch (sourcetype) {
995             case ZPROP_SRC_NONE:
996                 str = "-";
997                 break;
998
999             case ZPROP_SRC_DEFAULT:
1000                 str = "default";
1001                 break;
1002
1003             case ZPROP_SRC_LOCAL:
1004                 str = "local";
1005                 break;
1006
1007             case ZPROP_SRC_TEMPORARY:
1008                 str = "temporary";
1009                 break;
1010
1011             case ZPROP_SRC_INHERITED:
1012                 (void) sprintf(buf, sizeof (buf),
1013                               "inherited from %s", source);
1014                 str = buf;
1015                 break;
1016             case ZPROP_SRC_RECEIVED:
1017                 str = "received";
1018                 break;
1019         }
1020         break;
1021
1022     case GET_COL_RECVD:
1023         str = (recvd_value == NULL ? "-" : recvd_value);
1024         break;
1025
1026     default:
1027         continue;
1028     }
1029
1030     if (cbp->cb_columns[i + 1] == GET_COL_NONE)
1031         (void) printf("%s", str);
1032     else if (cbp->cb_scripted)
1033         (void) printf("%s\t", str);
1034     else
1035         (void) printf("%-*s ",
1036                       cbp->cb_colwidths[cbp->cb_columns[i]],
1037                       str);
1038     }
1039
1040     (void) printf("\n");
1041 }

```

unchanged portion omitted