```
**********************************************************
    2278 Wed May 14 12:03:03 2014
new/usr/src/lib/libzfs/Makefile.com
zpool import is braindead
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright (c) 2012 by Delphix. All rights reserved.
# Copyright 2014 RackTop Systems.
#endif /* ! codereview */
#

LIBRARY= libzfs.a
VERS= .1

OBJS_SHARED=                    \
        zfeature_common.o       \
        zfs_comutil.o           \
        zfs_deleg.o             \
        zfs_fletcher.o          \
        zfs_namecheck.o         \
        zfs_prop.o              \
        zpool_prop.o            \
        zprop_common.o

OBJS_COMMON=                    \
        libzfs_changelist.o     \
        libzfs_config.o         \
        libzfs_dataset.o        \
        libzfs_diff.o           \
        libzfs_fru.o            \
        libzfs_import.o         \
        libzfs_iter.o           \
        libzfs_mount.o          \
        libzfs_pool.o           \
        libzfs_sendrecv.o       \
        libzfs_status.o         \
        libzfs_util.o

OBJECTS= $(OBJS_COMMON) $(OBJS_SHARED)

include ../../Makefile.lib

# libzfs must be installed in the root filesystem for mount(1M)
include ../../Makefile.rootfs
```



```
LIBS=   $(DYNLIB) $(LINTLIB)

SRCDIR =        ../common

INCS += -I$(SRCDIR)
INCS += -I../../../uts/common/fs/zfs
INCS += -I../../../common/zfs
INCS += -I../../libc/inc

C99MODE=        -xc99=%all
C99LMODE=       -Xc99=%all
LDLIBS +=       -lc -lm -ldevid -lgen -lnvpair -luutil -lavl -lefi \
        -ladm -lidmap -ltsol -lmd -lumem -lzfs_core
CPPFLAGS +=     $(INCS) -D_LARGEFILE64_SOURCE=1 -D_REENTRANT

CERRWARN +=     -_gcc=-Wno-switch
CERRWARN +=     -_gcc=-Wno-parentheses
CERRWARN +=     -_gcc=-Wno-uninitialized
CERRWARN +=     -_gcc=-Wno-unused-function

SRCS=   $(OBJS_COMMON:%.o=$(SRCDIR)/%.c)        \
        $(OBJS_SHARED:%.o=$(SRC)/common/zfs/%.c)
$(LINTLIB) := SRCS=     $(SRCDIR)/$(LINTSRC)

.KEEP_STATE:

all: $(LIBS)

lint: lintcheck

pics/%.o: ../../../common/zfs/%.c
        $(COMPILE.c) -o $@ $<
        $(POST_PROCESS_O)

include ../../Makefile.targ
```

```
**********************************************************
  114386 Wed May 14 12:03:03 2014
new/usr/src/lib/libzfs/common/libzfs_dataset.c
zpool import is braindead
**********************************************************
```

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

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 DEY Storage Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 Martin Matuska. All rights reserved.
 * Copyright (c) 2013 Steven Hartland. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 * Copyright 2014 RackTop Systems.
#endif /* ! codereview */
 */

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

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

#include "zfs_namecheck.h"
```



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

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

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

        return (NULL);
}

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

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

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

        assert(types & ZFS_TYPE_VOLUME);
```

```
128         return (dgettext(TEXT_DOMAIN, "volume"));
129 }

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;

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;

152                         case NAME_ERR_LEADING_SLASH:
153                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
154                                     "leading slash in name"));
155                                 break;

157                         case NAME_ERR_EMPTY_COMPONENT:
158                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
159                                     "empty component in name"));
160                                 break;

162                         case NAME_ERR_TRAILING_SLASH:
163                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
164                                     "trailing slash in name"));
165                                 break;

167                         case NAME_ERR_INVALCHAR:
168                                 zfs_error_aux(hdl,
169                                     dgettext(TEXT_DOMAIN, "invalid character "
170                                     "'%c' in name"), what);
171                                 break;

173                         case NAME_ERR_MULTIPLE_AT:
174                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
175                                     "multiple '@' delimiters in name"));
176                                 break;

178                         case NAME_ERR_NOLETTER:
179                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
180                                     "pool doesn't begin with a letter"));
181                                 break;

183                         case NAME_ERR_RESERVED:
184                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
185                                     "name is reserved"));
186                                 break;

188                         case NAME_ERR_DISKLIKE:
189                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
190                                     "reserved disk name"));
191                                 break;
192                         }
193                 }
```



```
195                 return (0);
196         }

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         }

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         }

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         }

219         return (-1);
220 }

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 }

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;

242         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
243                 (void) no_memory(hdl);
244                 return (NULL);
245         }

247         elem = NULL;
248         while ((elem = nvlist_next_nvpair(props, elem)) != NULL) {
249                 if (!zfs_prop_user(nvpair_name(elem)))
250                         continue;

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             (strncmp(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) strlcpy(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                         }
341                 } else {
342                         return (-1);
343                 }
344         }
345         return (0);
346 }

348 /*
349  * Utility function to get the received properties of the given object.
350  */
351 static int
352 get_recvd_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                         }
369                 } else {
370                         zcmd_free_nvlists(&zc);
371                         return (-1);
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_recvd_props);
381         zhp->zfs_recvd_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)
```

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

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

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

boolean_t
zfs_bookmark_exists(const char *path)
{
        nvlist_t *bmarks;
        nvlist_t *props;
        char fsname[ZFS_MAXNAMELEN];
        char *bmark_name;
        char *pound;
        int err;
        boolean_t rv;


        (void) strlcpy(fsname, path, sizeof (fsname));
        pound = strchr(fsname, '#');
        if (pound == NULL)
                return (B_FALSE);

        *pound = '\0';
        bmark_name = pound + 1;
        props = fnvlist_alloc();
        err = lzc_get_bookmarks(fsname, props, &bmarks);
```



```
        nvlist_free(props);
        if (err != 0) {
                nvlist_free(bmarks);
                return (B_FALSE);
        }

        rv = nvlist_exists(bmarks, bmark_name);
        nvlist_free(bmarks);
        return (rv);
}

zfs_handle_t *
make_bookmark_handle(zfs_handle_t *parent, const char *path,
    nvlist_t *bmark_props)
{
        zfs_handle_t *zhp = calloc(sizeof (zfs_handle_t), 1);

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

        /* Fill in the name. */
        zhp->zfs_hdl = parent->zfs_hdl;
        (void) strlcpy(zhp->zfs_name, path, sizeof (zhp->zfs_name));

        /* Set the property lists. */
        if (nvlist_dup(bmark_props, &zhp->zfs_props, 0) != 0) {
                free(zhp);
                return (NULL);
        }

        /* Set the types. */
        zhp->zfs_head_type = parent->zfs_head_type;
        zhp->zfs_type = ZFS_TYPE_BOOKMARK;

        if ((zhp->zpool_hdl = zpool_handle(zhp)) == NULL) {
                nvlist_free(zhp->zfs_props);
                free(zhp);
                return (NULL);
        }

        return (zhp);
}

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

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

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

```
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         }

666         if (!(types & zhp->zfs_type)) {
667                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
668                 zfs_close(zhp);
669                 return (NULL);
670         }

672         return (zhp);
673 }

675 /*
676  * Release a ZFS handle.  Nothing to do but free the associated memory.
677  */
678 void
679 zfs_close(zfs_handle_t *zhp)
680 {
681         if (zhp->zfs_mntopts)
682                 free(zhp->zfs_mntopts);
683         nvlist_free(zhp->zfs_props);
684         nvlist_free(zhp->zfs_user_props);
685         nvlist_free(zhp->zfs_recvd_props);
686         free(zhp);
687 }

689 typedef struct mnttab_node {
690         struct mnttab mtn_mt;
691         avl_node_t mtn_node;
692 } mnttab_node_t;

694 static int
695 libzfs_mnttab_cache_compare(const void *arg1, const void *arg2)
696 {
697         const mnttab_node_t *mtn1 = arg1;
698         const mnttab_node_t *mtn2 = arg2;
699         int rv;

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

703         if (rv == 0)
704                 return (0);
705         return (rv > 0 ? 1 : -1);
706 }

708 void
709 libzfs_mnttab_init(libzfs_handle_t *hdl)
710 {
711         assert(avl_numnodes(&hdl->libzfs_mnttab_cache) == 0);
712         avl_create(&hdl->libzfs_mnttab_cache, libzfs_mnttab_cache_compare,
713             sizeof (mnttab_node_t), offsetof(mnttab_node_t, mtn_node));
714 }

716 void
717 libzfs_mnttab_update(libzfs_handle_t *hdl)
718 {
719         struct mnttab entry;

721         rewind(hdl->libzfs_mnttab);
```



```
722         while (getmntent(hdl->libzfs_mnttab, &entry) == 0) {
723                 mnttab_node_t *mtn;

725                 if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0)
726                         continue;
727                 mtn = zfs_alloc(hdl, sizeof (mnttab_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_mnttab_cache, mtn);
733         }
734 }

736 void
737 libzfs_mnttab_fini(libzfs_handle_t *hdl)
738 {
739         void *cookie = NULL;
740         mnttab_node_t *mtn;

742         while (mtn = avl_destroy_nodes(&hdl->libzfs_mnttab_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_mnttab_cache);
750 }

752 void
753 libzfs_mnttab_cache(libzfs_handle_t *hdl, boolean_t enable)
754 {
755         hdl->libzfs_mnttab_enable = enable;
756 }

758 int
759 libzfs_mnttab_find(libzfs_handle_t *hdl, const char *fsname,
760     struct mnttab *entry)
761 {
762         mnttab_node_t find;
763         mnttab_node_t *mtn;

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

768                 if (avl_numnodes(&hdl->libzfs_mnttab_cache))
769                         libzfs_mnttab_fini(hdl);
770                 rewind(hdl->libzfs_mnttab);
771                 srch.mnt_special = (char *)fsname;
772                 srch.mnt_fstype = MNTTYPE_ZFS;
773                 if (getmntany(hdl->libzfs_mnttab, entry, &srch) == 0)
774                         return (0);
775                 else
776                         return (ENOENT);
777         }

779         if (avl_numnodes(&hdl->libzfs_mnttab_cache) == 0)
780                 libzfs_mnttab_update(hdl);

782         find.mtn_mt.mnt_special = (char *)fsname;
783         mtn = avl_find(&hdl->libzfs_mnttab_cache, &find, NULL);
784         if (mtn) {
785                 *entry = mtn->mtn_mt;
786                 return (0);
787         }
```

```
788         return (ENOENT);
789 }

791 void
792 libzfs_mnttab_add(libzfs_handle_t *hdl, const char *special,
793     const char *mountp, const char *mntopts)
794 {
795         mnttab_node_t *mtn;

797         if (avl_numnodes(&hdl->libzfs_mnttab_cache) == 0)
798                 return;
799         mtn = zfs_alloc(hdl, sizeof (mnttab_node_t));
800         mtn->mtn_mt.mnt_special = zfs_strdup(hdl, special);
801         mtn->mtn_mt.mnt_mountp = zfs_strdup(hdl, mountp);
802         mtn->mtn_mt.mnt_fstype = zfs_strdup(hdl, MNTTYPE_ZFS);
803         mtn->mtn_mt.mnt_mntopts = zfs_strdup(hdl, mntopts);
804         avl_add(&hdl->libzfs_mnttab_cache, mtn);
805 }

807 void
808 libzfs_mnttab_remove(libzfs_handle_t *hdl, const char *fsname)
809 {
810         mnttab_node_t find;
811         mnttab_node_t *ret;

813         find.mtn_mt.mnt_special = (char *)fsname;
814         if (ret = avl_find(&hdl->libzfs_mnttab_cache, (void *)&find, NULL)) {
815                 avl_remove(&hdl->libzfs_mnttab_cache, ret);
816                 free(ret->mtn_mt.mnt_special);
817                 free(ret->mtn_mt.mnt_mountp);
818                 free(ret->mtn_mt.mnt_fstype);
819                 free(ret->mtn_mt.mnt_mntopts);
820                 free(ret);
821         }
822 }

824 int
825 zfs_spa_version(zfs_handle_t *zhp, int *spa_version)
826 {
827         zpool_handle_t *zpool_handle = zhp->zpool_hdl;

829         if (zpool_handle == NULL)
830                 return (-1);

832         *spa_version = zpool_get_prop_int(zpool_handle,
833             ZPOOL_PROP_VERSION, NULL);
834         return (0);
835 }

837 /*
838  * The choice of reservation property depends on the SPA version.
839  */
840 static int
841 zfs_which_resv_prop(zfs_handle_t *zhp, zfs_prop_t *resv_prop)
842 {
843         int spa_version;

845         if (zfs_spa_version(zhp, &spa_version) < 0)
846                 return (-1);

848         if (spa_version >= SPA_VERSION_REFRESERVATION)
849                 *resv_prop = ZFS_PROP_REFRESERVATION;
850         else
851                 *resv_prop = ZFS_PROP_RESERVATION;

853         return (0);
```



```
854 }

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

873         if (nvlist_alloc(&ret, NV_UNIQUE_NAME, 0) != 0) {
874                 (void) no_memory(hdl);
875                 return (NULL);
876         }

878         /*
879          * Make sure this property is valid and applies to this type.
880          */

882         elem = NULL;
883         while ((elem = nvlist_next_nvpair(nvl, elem)) != NULL) {
884                 const char *propname = nvpair_name(elem);

886                 prop = zfs_name_to_prop(propname);
887                 if (prop == ZPROP_INVAL && zfs_prop_user(propname)) {
888                         /*
889                          * This is a user property: make sure it's a
890                          * string, and that it's less than ZAP_MAXNAMELEN.
891                          */
892                         if (nvpair_type(elem) != DATA_TYPE_STRING) {
893                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
894                                     "'%s' must be a string"), propname);
895                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
896                                 goto error;
897                         }

899                         if (strlen(nvpair_name(elem)) >= ZAP_MAXNAMELEN) {
900                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
901                                     "property name '%s' is too long"),
902                                     propname);
903                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
904                                 goto error;
905                         }

907                         (void) nvpair_value_string(elem, &strval);
908                         if (nvlist_add_string(ret, propname, strval) != 0) {
909                                 (void) no_memory(hdl);
910                                 goto error;
911                         }
912                         continue;
913                 }

915                 /*
916                  * Currently, only user properties can be modified on
917                  * snapshots.
918                  */
919                 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             }

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];

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                     }

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_PROPREADONLY,
949                                 errbuf);
950                             goto error;
951                     }

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                     }

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) snprintf(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_PROPREADONLY, errbuf);
1001                    goto error;
1002            }

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            }

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"), propname);
1015                    (void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
1016                    goto error;
1017            }

1019            if (zfs_prop_readonly(prop) &&
1020                (!zfs_prop_setonce(prop) || zhp != NULL)) {
1021                    zfs_error_aux(hdl,
1022                        dgettext(TEXT_DOMAIN, "'%s' is readonly"),
1023                        propname);
1024                    (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
1025                    goto error;
1026            }

1028            if (zprop_parse_value(hdl, elem, prop, type, ret,
1029                &strval, &intval, errbuf) != 0)
1030                    goto error;

1032            /*
1033             * Perform some additional checks for specific properties.
1034             */
1035            switch (prop) {
1036            case ZFS_PROP_VERSION:
1037            {
1038                    int version;

1040                    if (zhp == NULL)
1041                            break;
1042                    version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1043                    if (intval < version) {
1044                            zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1045                                "Can not downgrade; already at version %u"),
1046                                version);
1047                            (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1048                            goto error;
1049                    }
1050                    break;
1051            }
```

```
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 ||
1057                             intval > SPA_MAXBLOCKSIZE || !ISP2(intval)) {
1058                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1059                                     "'%s' must be power of 2 from %u "
1060                                     "to %uk"), propname,
1061                                     (uint_t)SPA_MINBLOCKSIZE,
1062                                     (uint_t)SPA_MAXBLOCKSIZE >> 10);
1063                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1064                                 goto error;
1065                         }
1066                         break;

1068                 case ZFS_PROP_MLSLABEL:
1069                 {
1070                         /*
1071                          * Verify the mlslabel string and convert to
1072                          * internal hex label string.
1073                          */

1075                         m_label_t *new_sl;
1076                         char *hex = NULL;       /* internal label string */

1078                         /* Default value is already OK. */
1079                         if (strcasecmp(strval, ZFS_MLSLABEL_DEFAULT) == 0)
1080                                 break;

1082                         /* Verify the label can be converted to binary form */
1083                         if (((new_sl = m_label_alloc(MAC_LABEL)) == NULL) ||
1084                             (str_to_label(strval, &new_sl, MAC_LABEL,
1085                             L_NO_CORRECTION, NULL) == -1)) {
1086                                 goto badlabel;
1087                         }

1089                         /* Now translate to hex internal label string */
1090                         if (label_to_str(new_sl, &hex, M_INTERNAL,
1091                             DEF_NAMES) != 0) {
1092                                 if (hex)
1093                                         free(hex);
1094                                 goto badlabel;
1095                         }
1096                         m_label_free(new_sl);

1098                         /* If string is already in internal form, we're done. */
1099                         if (strcmp(strval, hex) == 0) {
1100                                 free(hex);
1101                                 break;
1102                         }

1104                         /* Replace the label string with the internal form. */
1105                         (void) nvlist_remove(ret, zfs_prop_to_name(prop),
1106                             DATA_TYPE_STRING);
1107                         verify(nvlist_add_string(ret, zfs_prop_to_name(prop),
1108                             hex) == 0);
1109                         free(hex);

1111                         break;

1113 badlabel:
1114                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1115                             "invalid mlslabel '%s'"), strval);
1116                         (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1117                         m_label_free(new_sl);   /* OK if null */
```



```
1118                         goto error;

1120                 }

1122                 case ZFS_PROP_MOUNTPOINT:
1123                 {
1124                         namecheck_err_t why;

1126                         if (strcmp(strval, ZFS_MOUNTPOINT_NONE) == 0 ||
1127                             strcmp(strval, ZFS_MOUNTPOINT_LEGACY) == 0)
1128                                 break;

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                 }

1150                         /*FALLTHRU*/

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                          *              global zone         non-global zone
1161                          * ------------------------------------------------------
1162                          * zoned=on     mountpoint (no)     mountpoint (yes)
1163                          *              sharenfs (no)       sharenfs (no)
1164                          *              sharesmb (no)       sharesmb (no)
1165                          *
1166                          * zoned=off    mountpoint (yes)        N/A
1167                          *              sharenfs (yes)
1168                          *              sharesmb (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                                    (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                    }

1200                    /*
1201                     * At this point, it is legitimate to set the
1202                     * property. Now we want to make sure that the
1203                     * property value is valid if it is sharenfs.
1204                     */
1205                    if ((prop == ZFS_PROP_SHARENFS ||
1206                        prop == ZFS_PROP_SHARESMB) &&
1207                        strcmp(strval, "on") != 0 &&
1208                        strcmp(strval, "off") != 0) {
1209                            zfs_share_proto_t proto;

1211                            if (prop == ZFS_PROP_SHARESMB)
1212                                    proto = PROTO_SMB;
1213                            else
1214                                    proto = PROTO_NFS;

1216                            /*
1217                             * Must be an valid sharing protocol
1218                             * option string so init the libshare
1219                             * in order to enable the parser and
1220                             * then parse the options. We use the
1221                             * control API since we don't care about
1222                             * the current configuration and don't
1223                             * want the overhead of loading it
1224                             * until we actually do something.
1225                             */

1227                            if (zfs_init_libshare(hdl,
1228                                SA_INIT_CONTROL_API) != SA_OK) {
1229                                    /*
1230                                     * An error occurred so we can't do
1231                                     * anything
1232                                     */
1233                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1234                                        "'%s' cannot be set: problem "
1235                                        "in share initialization"),
1236                                        propname);
1237                                    (void) zfs_error(hdl, EZFS_BADPROP,
1238                                        errbuf);
1239                                    goto error;
1240                            }

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



```
1250                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1251                                        "'%s' cannot be set to invalid "
1252                                        "options"), propname);
1253                                    (void) zfs_error(hdl, EZFS_BADPROP,
1254                                        errbuf);
1255                                    zfs_uninit_libshare(hdl);
1256                                    goto error;
1257                            }
1258                            zfs_uninit_libshare(hdl);
1259                    }

1261                    break;
1262            case ZFS_PROP_UTF8ONLY:
1263                    chosen_utf = (int)intval;
1264                    break;
1265            case ZFS_PROP_NORMALIZE:
1266                    chosen_normal = (int)intval;
1267                    break;
1268            }

1270            /*
1271             * For changes to existing volumes, we have some additional
1272             * checks to enforce.
1273             */
1274            if (type == ZFS_TYPE_VOLUME && zhp != NULL) {
1275                    uint64_t volsize = zfs_prop_get_int(zhp,
1276                        ZFS_PROP_VOLSIZE);
1277                    uint64_t blocksize = zfs_prop_get_int(zhp,
1278                        ZFS_PROP_VOLBLOCKSIZE);
1279                    char buf[64];

1281                    switch (prop) {
1282                    case ZFS_PROP_RESERVATION:
1283                    case ZFS_PROP_REFRESERVATION:
1284                            if (intval > volsize) {
1285                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1286                                        "'%s' is greater than current "
1287                                        "volume size"), propname);
1288                                    (void) zfs_error(hdl, EZFS_BADPROP,
1289                                        errbuf);
1290                                    goto error;
1291                            }
1292                            break;

1294                    case ZFS_PROP_VOLSIZE:
1295                            if (intval % blocksize != 0) {
1296                                    zfs_nicenum(blocksize, buf,
1297                                        sizeof (buf));
1298                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1299                                        "'%s' must be a multiple of "
1300                                        "volume block size (%s)"),
1301                                        propname, buf);
1302                                    (void) zfs_error(hdl, EZFS_BADPROP,
1303                                        errbuf);
1304                                    goto error;
1305                            }

1307                            if (intval == 0) {
1308                                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1309                                        "'%s' cannot be zero"),
1310                                        propname);
1311                                    (void) zfs_error(hdl, EZFS_BADPROP,
1312                                        errbuf);
1313                                    goto error;
1314                            }
1315                            break;
```

```
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);
```



```
1382        fnvlist_free(props);

1384        if (nvlist_add_uint64(nvl, zfs_prop_to_name(resv_prop),
1385            new_reservation) != 0) {
1386                (void) no_memory(zhp->zfs_hdl);
1387                return (-1);
1388        }
1389        return (1);
1390 }

1392 void
1393 zfs_setprop_error(libzfs_handle_t *hdl, zfs_prop_t prop, int err,
1394     char *errbuf)
1395 {
1396        switch (err) {

1398        case ENOSPC:
1399                /*
1400                 * For quotas and reservations, ENOSPC indicates
1401                 * something different; setting a quota or reservation
1402                 * doesn't use any disk space.
1403                 */
1404                switch (prop) {
1405                case ZFS_PROP_QUOTA:
1406                case ZFS_PROP_REFQUOTA:
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;

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;

1420                default:
1421                        (void) zfs_standard_error(hdl, err, errbuf);
1422                        break;
1423                }
1424                break;

1426        case EBUSY:
1427                (void) zfs_standard_error(hdl, EBUSY, errbuf);
1428                break;

1430        case EROFS:
1431                (void) zfs_error(hdl, EZFS_DSREADONLY, errbuf);
1432                break;

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;

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 {
```

```
1448                        (void) zfs_standard_error(hdl, err, errbuf);
1449                }
1450                break;

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 EOVERFLOW:
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;
  30        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_get_int(zhp, 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        }

1516        if ((cl = changelist_gather(zhp, prop, 0, 0)) == NULL)
1517                goto error;

1519        if (prop == ZFS_PROP_MOUNTPOINT && changelist_haszonedchild(cl)) {
1520                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1521                    "child dataset with inherited mountpoint is used "
1522                    "in a non-global zone"));
1523                ret = zfs_error(hdl, EZFS_ZONED, errbuf);
1524                goto error;
1525        }

1527        /*
1528         * We don't want to unmount & remount the dataset when changing
1529         * its canmount property to 'on' or 'noauto'.  We only use
1530         * the changelist logic to unmount when setting canmount=off.
1531         */
1532        if (prop == ZFS_PROP_CANMOUNT) {
1533                uint64_t idx;
1534                int err = zprop_string_to_index(prop, propval, &idx,
1535                    ZFS_TYPE_DATASET);
1536                if (err == 0 && idx != ZFS_CANMOUNT_OFF)
1537                        do_prefix = B_FALSE;
1538        }

1540        if (do_prefix && (ret = changelist_prefix(cl)) != 0)
1541                goto error;

1543        /*
1544         * Execute the corresponding ioctl() to set this property.
1545         */
1546        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

1548        if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1549                goto error;

1551        ret = zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);

1553        if (ret != 0) {
1554                zfs_setprop_error(hdl, prop, errno, errbuf);
1555                if (added_resv && errno == ENOSPC) {
1556                        /* clean up the volsize property we tried to set */
1557                        uint64_t old_volsize = zfs_prop_get_int(zhp,
1558                            ZFS_PROP_VOLSIZE);
1559                        nvlist_free(nvl);
1560                        zcmd_free_nvlists(&zc);
1561                        if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
1562                                goto error;
1563                        if (nvlist_add_uint64(nvl,
1564                            zfs_prop_to_name(ZFS_PROP_VOLSIZE),
1565                            old_volsize) != 0)
1566                                goto error;
1567                        if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
1568                                goto error;
1569                        (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
1570                }
1571        } else {
1572                if (do_prefix)
1573                        ret = changelist_postfix(cl);

1575                /*
1576                 * Refresh the statistics so the new property value
1577                 * is reflected.
1578                 */
```

```
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;
1550         int rc;

3012         if (check_parents(hdl, path, NULL, B_TRUE, &prefix) != 0)
3013                 return (-1);

3015         if ((path_copy = strdup(path)) != NULL) {
3016                 rc = create_parents(hdl, path_copy, prefix);
3017                 free(path_copy);
3018         }
3019         if (path_copy == NULL || rc != 0)
3020                 return (-1);

3022         return (0);
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;
2172         uint64_t old_volsize, new_volsize;
3633         zfs_prop_t resv_prop;

3635         assert(zhp->zfs_type == ZFS_TYPE_FILESYSTEM ||
3636             zhp->zfs_type == ZFS_TYPE_VOLUME);

3638         /*
3639          * Destroy all recent snapshots and their dependents.
3640          */
3641         cb.cb_force = force;
3642         cb.cb_target = snap->zfs_name;
3643         cb.cb_create = zfs_prop_get_int(snap, ZFS_PROP_CREATETXG);
3644         (void) zfs_iter_snapshots(zhp, rollback_destroy, &cb);
3645         (void) zfs_iter_bookmarks(zhp, rollback_destroy, &cb);
```



```
3647         if (cb.cb_error)
3648                 return (-1);

3650         /*
3651          * Now that we have verified that the snapshot is the latest,
3652          * rollback to the given snapshot.
3653          */

3655         if (zhp->zfs_type == ZFS_TYPE_VOLUME) {
3656                 if (zfs_which_resv_prop(zhp, &resv_prop) < 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         }

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         }

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 }

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;
2244         int ret;
3705         zfs_cmd_t zc = { 0 };
3706         char *delim;
3707         prop_changelist_t *cl = NULL;
3708         zfs_handle_t *zhrp = NULL;
3709         char *parentname = NULL;
3710         char parent[ZFS_MAXNAMELEN];
```

```
3711        libzfs_handle_t *hdl = zhp->zfs_hdl;
3712        char errbuf[1024];

3714        /* if we have the same exact name, just return success */
3715        if (strcmp(zhp->zfs_name, target) == 0)
3716                return (0);

3718        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3719            "cannot rename to '%s'"), target);

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                }
3754                if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
3755                        return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3756        } else {
3757                if (recursive) {
3758                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3759                            "recursive rename must be a snapshot"));
3760                        return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
3761                }

3763                if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
3764                        return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));

3766                /* validate parents */
3767                if (check_parents(hdl, target, NULL, B_FALSE, NULL) != 0)
3768                        return (-1);

3770                /* make sure we're in the same pool */
3771                verify((delim = strchr(target, '/')) != NULL);
3772                if (strncmp(zhp->zfs_name, target, delim - target) != 0 ||
3773                    zhp->zfs_name[delim - target] != '/') {
3774                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3775                            "datasets must be within same pool"));
3776                        return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));
```



```
3777                }

3779                /* new name cannot be a child of the current dataset name */
3780                if (is_descendant(zhp->zfs_name, target)) {
3781                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3782                            "New dataset name cannot be a descendant of "
3783                            "current dataset name"));
3784                        return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3785                }
3786        }

3788        (void) snprintf(errbuf, sizeof (errbuf),
3789            dgettext(TEXT_DOMAIN, "cannot rename '%s'"), zhp->zfs_name);

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        }

3798        if (recursive) {

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                }

3813        } else {
3814                if ((cl = changelist_gather(zhp, ZFS_PROP_NAME, 0,
3815                    force_unmount ? MS_FORCE : 0)) == NULL)
3816                        return (-1);

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                }

3826                if ((ret = changelist_prefix(cl)) != 0)
3827                        goto error;
3828        }

3830        if (ZFS_IS_VOLUME(zhp))
3831                zc.zc_objset_type = DMU_OST_ZVOL;
3832        else
3833                zc.zc_objset_type = DMU_OST_ZFS;

3835        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
3836        (void) strlcpy(zc.zc_value, target, sizeof (zc.zc_value));

3838        zc.zc_cookie = recursive;

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);

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                            (void) zfs_error(hdl, EZFS_EXISTS, errbuf);
3853                    } else {
3854                            (void) zfs_standard_error(zhp->zfs_hdl, errno, errbuf);
3855                    }

3857                    /*
3858                     * On failure, we still want to remount any filesystems that
3859                     * were previously mounted, so we don't alter the system state.
3860                     */
3861                    if (!recursive)
3862                            (void) changelist_postfix(cl);
3863            } else {
3864                    if (!recursive) {
3865                            changelist_rename(cl, zfs_get_name(zhp), target);
3866                            ret = changelist_postfix(cl);
3867                    }
3868            }

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_***

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

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright 2014 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2014 RackTop Systems.
#endif /* ! codereview */
 */

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

#include <ctype.h>
#include <devid.h>
#include <dirent.h>
#include <errno.h>
#include <libintl.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/vtoc.h>
#include <sys/dktp/fdisk.h>
#include <sys/efi_partition.h>
#include <thread_pool.h>

#include <sys/vdev_impl.h>
```



```
#include "libzfs.h"
#include "libzfs_impl.h"

/*
 * Intermediate structures used to gather configuration information.
 */
typedef struct config_entry {
        uint64_t                ce_txg;
        nvlist_t                *ce_config;
        struct config_entry     *ce_next;
} config_entry_t;

typedef struct vdev_entry {
        uint64_t                ve_guid;
        config_entry_t          *ve_configs;
        struct vdev_entry       *ve_next;
} vdev_entry_t;

typedef struct pool_entry {
        uint64_t                pe_guid;
        vdev_entry_t            *pe_vdevs;
        struct pool_entry       *pe_next;
} pool_entry_t;

typedef struct name_entry {
        char                    *ne_name;
        uint64_t                ne_guid;
        struct name_entry       *ne_next;
} name_entry_t;

typedef struct pool_list {
        pool_entry_t            *pools;
        name_entry_t            *names;
} pool_list_t;

static char *
get_devid(const char *path)
{
        int fd;
        ddi_devid_t devid;
        char *minor, *ret;

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

        minor = NULL;
        ret = NULL;
        if (devid_get(fd, &devid) == 0) {
                if (devid_get_minor_name(fd, &minor) == 0)
                        ret = devid_str_encode(devid, minor);
                if (minor != NULL)
                        devid_str_free(minor);
                devid_free(devid);
        }
        (void) close(fd);

        return (ret);
}


/*
 * Go through and fix up any path and/or devid information for the given vdev
 * configuration.
 */
static int
```

```
fix_paths(nvlist_t *nv, name_entry_t *names)
{
        nvlist_t **child;
        uint_t c, children;
        uint64_t guid;
        name_entry_t *ne, *best;
        char *path, *devid;
        int matched;

        if (nvlist_lookup_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN,
            &child, &children) == 0) {
                for (c = 0; c < children; c++)
                        if (fix_paths(child[c], names) != 0)
                                return (-1);
                return (0);
        }

        /*
         * This is a leaf (file or disk) vdev.  In either case, go through
         * the name list and see if we find a matching guid.  If so, replace
         * the path and see if we can calculate a new devid.
         *
         * There may be multiple names associated with a particular guid, in
         * which case we have overlapping slices or multiple paths to the same
         * disk.  If this is the case, then we want to pick the path that is
         * the most similar to the original, where "most similar" is the number
         * of matching characters starting from the end of the path.  This will
         * preserve slice numbers even if the disks have been reorganized, and
         * will also catch preferred disk names if multiple paths exist.
         */
        verify(nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) == 0);
        if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PATH, &path) != 0)
                path = NULL;

        matched = 0;
        best = NULL;
        for (ne = names; ne != NULL; ne = ne->ne_next) {
                if (ne->ne_guid == guid) {
                        const char *src, *dst;
                        int count;

                        if (path == NULL) {
                                best = ne;
                                break;
                        }

                        src = ne->ne_name + strlen(ne->ne_name) - 1;
                        dst = path + strlen(path) - 1;
                        for (count = 0; src >= ne->ne_name && dst >= path;
                            src--, dst--, count++)
                                if (*src != *dst)
                                        break;

                        /*
                         * At this point, 'count' is the number of characters
                         * matched from the end.
                         */
                        if (count > matched || best == NULL) {
                                best = ne;
                                matched = count;
                        }
                }
        }

        if (best == NULL)
                return (0);
```



```
        if (nvlist_add_string(nv, ZPOOL_CONFIG_PATH, best->ne_name) != 0)
                return (-1);

        if ((devid = get_devid(best->ne_name)) == NULL) {
                (void) nvlist_remove_all(nv, ZPOOL_CONFIG_DEVID);
        } else {
                if (nvlist_add_string(nv, ZPOOL_CONFIG_DEVID, devid) != 0)
                        return (-1);
                devid_str_free(devid);
        }

        return (0);
}

/*
 * Add the given configuration to the list of known devices.
 */
static int
add_config(libzfs_handle_t *hdl, pool_list_t *pl, const char *path,
    nvlist_t *config)
{
        uint64_t pool_guid, vdev_guid, top_guid, txg, state;
        pool_entry_t *pe;
        vdev_entry_t *ve;
        config_entry_t *ce;
        name_entry_t *ne;

        /*
         * If this is a hot spare not currently in use or level 2 cache
         * device, add it to the list of names to translate, but don't do
         * anything else.
         */
        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_STATE,
            &state) == 0 &&
            (state == POOL_STATE_SPARE || state == POOL_STATE_L2CACHE) &&
            nvlist_lookup_uint64(config, ZPOOL_CONFIG_GUID, &vdev_guid) == 0) {
                if ((ne = zfs_alloc(hdl, sizeof (name_entry_t))) == NULL)
                        return (-1);

                if ((ne->ne_name = zfs_strdup(hdl, path)) == NULL) {
                        free(ne);
                        return (-1);
                }
                ne->ne_guid = vdev_guid;
                ne->ne_next = pl->names;
                pl->names = ne;
                return (0);
        }

        /*
         * If we have a valid config but cannot read any of these fields, then
         * it means we have a half-initialized label.  In vdev_label_init()
         * we write a label with txg == 0 so that we can identify the device
         * in case the user refers to the same disk later on.  If we fail to
         * create the pool, we'll be left with a label in this state
         * which should not be considered part of a valid pool.
         */
        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID,
            &pool_guid) != 0 ||
            nvlist_lookup_uint64(config, ZPOOL_CONFIG_GUID,
            &vdev_guid) != 0 ||
            nvlist_lookup_uint64(config, ZPOOL_CONFIG_TOP_GUID,
            &top_guid) != 0 ||
            nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_TXG,
            &txg) != 0 || txg == 0) {
```

```
260                 nvlist_free(config);
261                 return (0);
262         }

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         }

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         }

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         }

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         }

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         }

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         }

325         /*
```



```
326          * At this point we've successfully added our config to the list of
327          * known configs.  The last thing to do is add the vdev guid -> path
328          * mappings so that we can fix up the configuration as necessary before
329          * doing the import.
330          */
331         if ((ne = zfs_alloc(hdl, sizeof (name_entry_t))) == NULL)
332                 return (-1);

334         if ((ne->ne_name = zfs_strdup(hdl, path)) == NULL) {
335                 free(ne);
336                 return (-1);
337         }

339         ne->ne_guid = vdev_guid;
340         ne->ne_next = pl->names;
341         pl->names = ne;

343         return (0);
344 }

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;

356         if (zpool_open_silent(hdl, name, &zhp) != 0)
357                 return (-1);

359         if (zhp == NULL) {
360                 *isactive = B_FALSE;
361                 return (0);
362         }

364         verify(nvlist_lookup_uint64(zhp->zpool_config, ZPOOL_CONFIG_POOL_GUID,
365             &theguid) == 0);

367         zpool_close(zhp);

369         *isactive = (theguid == guid);
370         return (0);
371 }

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;

380         if (zcmd_write_conf_nvlist(hdl, &zc, config) != 0)
381                 return (NULL);

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         }

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         }

397         if (err) {
398                 zcmd_free_nvlists(&zc);
399                 return (NULL);
400         }

402         if (zcmd_read_dst_nvlist(hdl, &zc, &nvl) != 0) {
403                 zcmd_free_nvlists(&zc);
404                 return (NULL);
405         }

407         zcmd_free_nvlists(&zc);
408         return (nvl);
409 }

411 /*
412  * Determine if the vdev id is a hole in the namespace.
413  */
414 boolean_t
415 vdev_is_hole(uint64_t *hole_array, uint_t holes, uint_t id)
416 {
417         for (int c = 0; c < holes; c++) {

419                 /* Top-level is a hole */
420                 if (hole_array[c] == id)
421                         return (B_TRUE);
422         }
423         return (B_FALSE);
424 }

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



```
457         if (nvlist_alloc(&ret, 0, 0) != 0)
458                 goto nomem;

460         for (pe = pl->pools; pe != NULL; pe = pe->pe_next) {
461                 uint64_t id, max_txg = 0;

463                 if (nvlist_alloc(&config, NV_UNIQUE_NAME, 0) != 0)
464                         goto nomem;
465                 config_seen = B_FALSE;

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) {

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) {

483                                 if (ce->ce_txg > best_txg) {
484                                         tmp = ce->ce_config;
485                                         best_txg = ce->ce_txg;
486                                 }
487                         }

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);

502                                 max_txg = best_txg;
503                                 hole_array = NULL;
504                                 holes = 0;
505                                 max_id = 0;
506                                 valid_top_config = B_FALSE;

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                                 }

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                     }

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                             uint64_t state, version;
539                             char *comment = NULL;

541                             version = fnvlist_lookup_uint64(tmp,
542                                 ZPOOL_CONFIG_VERSION);
543                             fnvlist_add_uint64(config,
544                                 ZPOOL_CONFIG_VERSION, version);
545                             guid = fnvlist_lookup_uint64(tmp,
546                                 ZPOOL_CONFIG_POOL_GUID);
547                             fnvlist_add_uint64(config,
548                                 ZPOOL_CONFIG_POOL_GUID, guid);
549                             name = fnvlist_lookup_string(tmp,
550                                 ZPOOL_CONFIG_POOL_NAME);
551                             fnvlist_add_string(config,
552                                 ZPOOL_CONFIG_POOL_NAME, name);

554                             if (nvlist_lookup_string(tmp,
555                                 ZPOOL_CONFIG_COMMENT, &comment) == 0)
556                                     fnvlist_add_string(config,
557                                         ZPOOL_CONFIG_COMMENT, comment);

559                             state = fnvlist_lookup_uint64(tmp,
560                                 ZPOOL_CONFIG_POOL_STATE);
561                             fnvlist_add_uint64(config,
562                                 ZPOOL_CONFIG_POOL_STATE, state);

564                             hostid = 0;
565                             if (nvlist_lookup_uint64(tmp,
566                                 ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
567                                     fnvlist_add_uint64(config,
568                                         ZPOOL_CONFIG_HOSTID, hostid);
569                                     hostname = fnvlist_lookup_string(tmp,
570                                         ZPOOL_CONFIG_HOSTNAME);
571                                     fnvlist_add_string(config,
572                                         ZPOOL_CONFIG_HOSTNAME, hostname);
573                             }

575                             config_seen = B_TRUE;
576                     }

578                     /*
579                      * Add this top-level vdev to the child array.
580                      */
581                     verify(nvlist_lookup_nvlist(tmp,
582                         ZPOOL_CONFIG_VDEV_TREE, &nvtop) == 0);
583                     verify(nvlist_lookup_uint64(nvtop, ZPOOL_CONFIG_ID,
584                         &id) == 0);

586                     if (id >= children) {
587                             nvlist_t **newchild;
```



```
589                                     newchild = zfs_alloc(hdl, (id + 1) *
590                                         sizeof (nvlist_t *));
591                                     if (newchild == NULL)
592                                             goto nomem;

594                                     for (c = 0; c < children; c++)
595                                             newchild[c] = child[c];

597                                     free(child);
598                                     child = newchild;
599                                     children = id + 1;
600                             }
601                             if (nvlist_dup(nvtop, &child[id], 0) != 0)
602                                     goto nomem;

604                     }

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;

621                                     newchild = zfs_alloc(hdl, (max_id) *
622                                         sizeof (nvlist_t *));
623                                     if (newchild == NULL)
624                                             goto nomem;

626                                     for (c = 0; c < children; c++)
627                                             newchild[c] = child[c];

629                                     free(child);
630                                     child = newchild;
631                                     children = max_id;
632                             }
633                     }

635                     verify(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID,
636                         &guid) == 0);

638                     /*
639                      * The vdev namespace may contain holes as a result of
640                      * device removal. We must add them back into the vdev
641                      * tree before we process any missing devices.
642                      */
643                     if (holes > 0) {
644                             ASSERT(valid_top_config);

646                             for (c = 0; c < children; c++) {
647                                     nvlist_t *holey;

649                                     if (child[c] != NULL ||
650                                         !vdev_is_hole(hole_array, holes, c))
651                                             continue;

653                                     if (nvlist_alloc(&holey, NV_UNIQUE_NAME,
```

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



```
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                 }

789                 /*
790                  * Update the paths for l2cache devices.
791                  */
792                 if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
793                     &l2cache, &nl2cache) == 0) {
794                         for (i = 0; i < nl2cache; i++) {
795                                 if (fix_paths(l2cache[i], pl->names) != 0)
796                                         goto nomem;
797                         }
798                 }

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                 }

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;

823                 found_one = B_TRUE;
824                 nvlist_free(config);
825                 config = NULL;
826         }

828         if (!found_one) {
829                 nvlist_free(ret);
830                 ret = NULL;
831         }

833         return (ret);

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);

844         return (NULL);
845 }
```

*unchanged_portion_omitted_*

```
910 typedef struct slice_node {
911         char *sn_name;
912         nvlist_t *sn_config;
913         boolean_t sn_nozpool;
```



```
914         int sn_partno;
915         struct disk_node *sn_disk;
916         struct slice_node *sn_next;
917 } slice_node_t;

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;

928 #ifdef  sparc
929 #define WHOLE_DISK      "s2"
930 #else
931 #define WHOLE_DISK      "p0"
932 #endif
496 typedef struct rdsk_node {
497         char *rn_name;
498         int rn_dfd;
499         libzfs_handle_t *rn_hdl;
500         nvlist_t *rn_config;
501         avl_tree_t *rn_avl;
502         avl_node_t rn_node;
503         boolean_t rn_nozpool;
504 } rdsk_node_t;
```

```
934 /*
935  * This function splits the slice from the device name.  Currently it supports
936  * VTOC slices (s[0-16]) and DOS/FDISK partitions (p[0-4]).  If this function
937  * is updated to support other slice types then the check_slices function will
938  * also need to be updated.
939  */
940 static boolean_t
941 get_disk_slice(libzfs_handle_t *hdl, char *disk, char **slice, int *partno)
506 static int
507 slice_cache_compare(const void *arg1, const void *arg2)
942 {
943         char *p;

945         if ((p = strrchr(disk, 's')) == NULL &&
946             (p = strrchr(disk, 'p')) == NULL)
947                 return (B_FALSE);

949         if (!isdigit(p[1]))
950                 return (B_FALSE);
509         const char  *nm1 = ((rdsk_node_t *)arg1)->rn_name;
510         const char  *nm2 = ((rdsk_node_t *)arg2)->rn_name;
511         char *nm1slice, *nm2slice;
512         int rv;
```

```
952         *slice = zfs_strdup(hdl, p);
953         *partno = atoi(p + 1);
514         /*
515          * slices zero and two are the most likely to provide results,
516          * so put those first
517          */
518         nm1slice = strstr(nm1, "s0");
519         nm2slice = strstr(nm2, "s0");
520         if (nm1slice && !nm2slice) {
521                 return (-1);
522         }
523         if (!nm1slice && nm2slice) {
524                 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         }

955         p = '\0';
956         return (B_TRUE);
535         rv = strcmp(nm1, nm2);
536         if (rv == 0)
537                 return (0);
538         return (rv > 0 ? 1 : -1);
957 }

959 static void
960 check_one_slice(slice_node_t *slice, diskaddr_t size, uint_t blksz)
542 check_one_slice(avl_tree_t *r, char *diskname, uint_t partno,
543     diskaddr_t size, uint_t blksz)
961 {
545         rdsk_node_t tmpnode;
546         rdsk_node_t *node;
547         char sname[MAXNAMELEN];

549         tmpnode.rn_name = &sname[0];
550         (void) snprintf(tmpnode.rn_name, MAXNAMELEN, "%s%u",
551             diskname, partno);
962         /*
963          * protect against division by zero for disk labels that
964          * contain a bogus sector size
965          */
966         if (blksz == 0)
967                 blksz = DEV_BSIZE;
968         /* too small to contain a zpool? */
969         if (size < (SPA_MINDEVSIZE / blksz))
970                 slice->sn_nozpool = B_TRUE;
559         if ((size < (SPA_MINDEVSIZE / blksz)) &&
560             (node = avl_find(r, &tmpnode, NULL)))
561                 node->rn_nozpool = B_TRUE;
971 }

973 static void
974 check_slices(slice_node_t *slices, int fd)
565 nozpool_all_slices(avl_tree_t *r, const char *sname)
566 {
567         char diskname[MAXNAMELEN];
568         char *ptr;
569         int i;

571         (void) strncpy(diskname, sname, MAXNAMELEN);
572         if (((ptr = strrchr(diskname, 's')) == NULL) &&
573             ((ptr = strrchr(diskname, 'p')) == NULL))
574                 return;
575         ptr[0] = 's';
576         ptr[1] = '\0';
577         for (i = 0; i < NDKMAP; i++)
578                 check_one_slice(r, diskname, i, 0, 1);
579         ptr[0] = 'p';
580         for (i = 0; i <= FD_NUMPART; i++)
581                 check_one_slice(r, diskname, i, 0, 1);
582 }

584 static void
```



```
585 check_slices(avl_tree_t *r, int fd, const char *sname)
975 {
976         struct extvtoc vtoc;
977         struct dk_gpt *gpt;
978         slice_node_t *slice;
979         diskaddr_t size;
589         char diskname[MAXNAMELEN];
590         char *ptr;
591         int i;

593         (void) strncpy(diskname, sname, MAXNAMELEN);
594         if ((ptr = strrchr(diskname, 's')) == NULL || !isdigit(ptr[1]))
595                 return;
596         ptr[1] = '\0';

981         if (read_extvtoc(fd, &vtoc) >= 0) {
982                 for (slice = slices; slice; slice = slice->sn_next) {
983                         if (slice->sn_name[0] == 'p')
984                                 continue;
985                         size = vtoc.v_part[slice->sn_partno].p_size;
986                         check_one_slice(slice, size, vtoc.v_sectorsz);
987                 }
599                 for (i = 0; i < NDKMAP; i++)
600                         check_one_slice(r, diskname, i,
601                             vtoc.v_part[i].p_size, vtoc.v_sectorsz);
988         } else if (efi_alloc_and_read(fd, &gpt) >= 0) {
989                 for (slice = slices; slice; slice = slice->sn_next) {
603                 /*
604                  * on x86 we'll still have leftover links that point
605                  * to slices s[9-15], so use NDKMAP instead
606                  */
607                 for (i = 0; i < NDKMAP; i++)
608                         check_one_slice(r, diskname, i,
609                             gpt->efi_parts[i].p_size, gpt->efi_lbasize);
990                         /* nodes p[1-4] are never used with EFI labels */
991                         if (slice->sn_name[0] == 'p') {
992                                 if (slice->sn_partno > 0)
993                                         slice->sn_nozpool = B_TRUE;
994                                 continue;
995                         }
996                         size = gpt->efi_parts[slice->sn_partno].p_size;
997                         check_one_slice(slice, size, gpt->efi_lbasize);
998                 }
611                 ptr[0] = 'p';
612                 for (i = 1; i <= FD_NUMPART; i++)
613                         check_one_slice(r, diskname, i, 0, 1);
999                 efi_free(gpt);
1000         }
1001 }

1003 static void
1004 zpool_open_func(void *arg)
1005 {
1006         disk_node_t *disk = arg;
621         rdsk_node_t *rn = arg;
1007         struct stat64 statbuf;
1008         slice_node_t *slice;
1009 #endif /* ! codereview */
1010         nvlist_t *config;
1011         char *devname;
1012 #endif /* ! codereview */
1013         int fd;

1015         /*
1016          * If the disk has no slices we open it directly, otherwise we try
1017          * to open the whole disk slice.
```

```
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);

1024         if (devname == NULL) {
1025                 (void) no_memory(disk->dn_hdl);
 623         if (rn->rn_nozpool)
1026                 return;
1027         }

1029         if ((fd = openat64(disk->dn_dfd, devname, O_RDONLY)) < 0) {
1030                 free(devname);
 625         if ((fd = openat64(rn->rn_dfd, rn->rn_name, O_RDONLY)) < 0) {
 626                 /* symlink to a device that's no longer there */
 627                 if (errno == ENOENT)
 628                         nozpool_all_slices(rn->rn_avl, rn->rn_name);
1031                 return;
1032         }
1033         /*
1034          * Ignore failed stats.  We only want regular
1035          * files, character devs and block devs.
1036          */
1037         if (fstat64(fd, &statbuf) != 0 ||
1038             (!S_ISREG(statbuf.st_mode) &&
1039             !S_ISCHR(statbuf.st_mode) &&
1040             !S_ISBLK(statbuf.st_mode))) {
1041                 (void) close(fd);
1042                 free(devname);
1043 #endif /* ! codereview */
1044                 return;
1045         }
1046         /* this file is too small to hold a zpool */
1047         if (S_ISREG(statbuf.st_mode) && statbuf.st_size < SPA_MINDEVSIZE) {
 640         if (S_ISREG(statbuf.st_mode) &&
 641             statbuf.st_size < SPA_MINDEVSIZE) {
1048                 (void) close(fd);
1049                 free(devname);
1050 #endif /* ! codereview */
1051                 return;
1052         } else if (!S_ISREG(statbuf.st_mode) && disk->dn_slices != NULL) {
 643         } else if (!S_ISREG(statbuf.st_mode)) {
1053                 /*
1054                  * Try to read the disk label first so we don't have to
1055                  * open a bunch of minor nodes that can't have a zpool.
1056                  */
1057                 check_slices(disk->dn_slices, fd);
 648                 check_slices(rn->rn_avl, fd, rn->rn_name);
1058         }

1060         /*
1061          * If we're working with the device directly (it has no slices)
1062          * then we can just read the config and we're done.
1063          */
1064         if (disk->dn_slices == NULL) {
1065                 if (zpool_read_label(fd, &config) != 0) {
1066                         (void) no_memory(disk->dn_hdl);
1067                         (void) close(fd);
1068                         free(devname);
1069                         return;
1070                 }
1071                 disk->dn_config = config;
 651         if ((zpool_read_label(fd, &config)) != 0) {
1072                 (void) close(fd);
1073                 free(devname);
```



```
 653                 (void) no_memory(rn->rn_hdl);
1074                 return;
1075         }

1077 #endif /* ! codereview */
1078         (void) close(fd);
1079         free(devname);

1081         /*
1082          * Go through and read the label off each slice.  The check_slices
1083          * function has already performed some basic checks and set the
1084          * sn_nozpool flag on any slices which just can't contain a zpool.
1085          */
1086         for (slice = disk->dn_slices; slice; slice = slice->sn_next) {
1087                 if (slice->sn_nozpool == B_TRUE)
1088                         continue;

1090                 (void) asprintf(&devname, "%s%s", disk->dn_name,
1091                     slice->sn_name);

1093                 if (devname == NULL) {
1094                         (void) no_memory(disk->dn_hdl);
1095                         free(devname);
1096                         return;
1097                 }
1098 #endif /* ! codereview */

1100                 if ((fd = openat64(disk->dn_dfd, devname, O_RDONLY)) < 0) {
1101                         free(devname);
1102                         continue;
1103                 }

1105                 if ((zpool_read_label(fd, &config)) != 0) {
1106                         (void) no_memory(disk->dn_hdl);
1107                         (void) close(fd);
1108                         free(devname);
1109                         return;
1110                 }
1111 #endif /* ! codereview */

1113                 slice->sn_config = config;
1114                 (void) close(fd);
1115                 free(devname);
 656         rn->rn_config = config;
 657         if (config != NULL) {
 658                 assert(rn->rn_nozpool == B_FALSE);
1116         }
1117 }
```

_____***unchanged_portion_omitted_***

```
1149 /*
1150  * Given a list of directories to search, find all pools stored on disk.  This
1151  * includes partial pools which are not available to import.  If no args are
1152  * given (argc is 0), then the default directory (/dev/dsk) is searched.
1153  * poolname or guid (but not both) are provided by the caller when trying
1154  * to import a specific pool.
1155  */
1156 static nvlist_t *
1157 zpool_find_import_impl(libzfs_handle_t *hdl, importargs_t *iarg)
1158 {
1159         int i, dirs = iarg->paths;
1160         DIR *dirp = NULL;
1161         struct dirent64 *dp;
1162         char path[MAXPATHLEN];
1163         char *end, **dir = iarg->path;
1164         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;
 715         avl_tree_t slice_cache;
 716         rdsk_node_t *slice;
1172         void *cookie;

1174         if (dirs == 0) {
1175                 dirs = 1;
1176                 dir = &default_dir;
1177         }

1179         /*
1180          * Go through and read the label configuration information from every
1181          * possible device, organizing the information according to pool GUID
1182          * and toplevel GUID.
1183          */
1184         for (i = 0; i < dirs; i++) {
1185                 tpool_t *t;
1186                 char *rdsk;
1187                 int dfd;
1188                 disk_node_t *disks = NULL, *curdisk = NULL;
1189                 slice_node_t *curslice = NULL;
1190 #endif /* ! codereview */

1192                 /* use realpath to normalize the path */
1193                 if (realpath(dir[i], path) == 0) {
1194                         (void) zfs_error_fmt(hdl, EZFS_BADPATH,
1195                             dgettext(TEXT_DOMAIN, "cannot open '%s'"), dir[i]);
1196                         goto error;
1197                 }
1198                 end = &path[strlen(path)];
1199                 *end++ = '/';
1200                 *end = 0;
1201                 pathleft = &path[sizeof (path)] - end;

1203                 /*
1204                  * Using raw devices instead of block devices when we're
1205                  * reading the labels skips a bunch of slow operations during
1206                  * close(2) processing, so we replace /dev/dsk with /dev/rdsk.
1207                  */
1208                 if (strcmp(path, "/dev/dsk/") == 0)
1209                         rdsk = "/dev/rdsk/";
1210                 else
1211                         rdsk = path;

1213                 if ((dfd = open64(rdsk, O_RDONLY)) < 0 ||
1214                     (dirp = fdopendir(dfd)) == NULL) {
1215                         zfs_error_aux(hdl, strerror(errno));
1216                         (void) zfs_error_fmt(hdl, EZFS_BADPATH,
1217                             dgettext(TEXT_DOMAIN, "cannot open '%s'"),
1218                             rdsk);
1219                         goto error;
1220                 }

 733                 avl_create(&slice_cache, slice_cache_compare,
 734                     sizeof (rdsk_node_t), offsetof(rdsk_node_t, rn_node));
1222                 /*
1223                  * This is not MT-safe, but we have no MT consumers of libzfs
1224                  */
1225                 while ((dp = readdir64(dirp)) != NULL) {
1226                         boolean_t isslice;
```



```
1227                         char *name, *sname;
1228                         int partno;

1230                         if (dp->d_name[0] == '.' && (dp->d_name[1] == '\0' ||
1231                             (dp->d_name[1] == '.' && dp->d_name[2] == '\0')))
 739                         const char *name = dp->d_name;
 740                         if (name[0] == '.' &&
 741                             (name[1] == 0 || (name[1] == '.' && name[2] == 0)))
1232                                 continue;

1234                         name = zfs_strdup(hdl, dp->d_name);

1236                         /*
1237                          * We create a new disk node every time we encounter
1238                          * a disk with no slices or the disk name changes.
1239                          */
1240                         isslice = get_disk_slice(hdl, name, &sname, &partno);
1241                         if (isslice == B_FALSE || curdisk == NULL ||
1242                             strcmp(curdisk->dn_name, name) != 0) {
1243                                 disk_node_t *newdisk;

1245                                 newdisk = zfs_alloc(hdl, sizeof (disk_node_t));
1246                                 newdisk->dn_name = name;
1247                                 newdisk->dn_dfd = dfd;
1248                                 newdisk->dn_hdl = hdl;

1250                                 if (curdisk != NULL)
1251                                         curdisk->dn_next = newdisk;
1252                                 else
1253                                         disks = newdisk;

1255                                 curdisk = newdisk;
1256                                 curslice = NULL;
1257                         }

1259                         assert(curdisk != NULL);

1261                         /*
1262                          * Add a new slice node to the current disk node.
1263                          * We do this for all slices including zero slices.
1264                          */
1265                         if (isslice == B_TRUE) {
1266                                 slice_node_t *newslice;

1268                                 newslice = zfs_alloc(hdl,
1269                                     sizeof (slice_node_t));
1270                                 newslice->sn_name = sname;
1271                                 newslice->sn_partno = partno;
1272                                 newslice->sn_disk = curdisk;

1274                                 if (curslice != NULL)
1275                                         curslice->sn_next = newslice;
1276                                 else
1277                                         curdisk->dn_slices = newslice;

1279                                 curslice = newslice;
1280                         }
 744                         slice = zfs_alloc(hdl, sizeof (rdsk_node_t));
 745                         slice->rn_name = zfs_strdup(hdl, name);
 746                         slice->rn_avl = &slice_cache;
 747                         slice->rn_dfd = dfd;
 748                         slice->rn_hdl = hdl;
 749                         slice->rn_nozpool = B_FALSE;
 750                         avl_add(&slice_cache, slice);
1281                 }
1282                 /*
```

```
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.
 754                  * rn_nozpool is not protected, so this is racy in that
 755                  * multiple tasks could decide that the same slice can
 756                  * not hold a zpool, which is benign.  Also choose
 757                  * double the number of processors; we hold a lot of
 758                  * locks in the kernel, so going beyond this doesn't
 759                  * buy us much.
1288                  */
1289                 t = tpool_create(1, 2 * sysconf(_SC_NPROCESSORS_ONLN),
1290                     0, NULL);
1291                 for (curdisk = disks; curdisk; curdisk = curdisk->dn_next)
1292                         (void) tpool_dispatch(t, zpool_open_func, curdisk);
 763                 for (slice = avl_first(&slice_cache); slice;
 764                     (slice = avl_walk(&slice_cache, slice,
 765                     AVL_AFTER)))
 766                         (void) tpool_dispatch(t, zpool_open_func, slice);
1293                 tpool_wait(t);
1294                 tpool_destroy(t);

1296                 curdisk = disks;
1297                 while (curdisk != NULL) {
1298                         nvlist_t *config;
1299                         disk_node_t *prevdisk;

1301                         /*
1302                          * If the device has slices we examine the config on
1303                          * each of those.  If not we use the config directly
1304                          * from the device instead.
1305                          */
1306                         curslice = curdisk->dn_slices;

1308                         if (curslice != NULL)
1309                                 config = curslice->sn_config;
1310                         else
1311                                 config = curdisk->dn_config;

1313                         do {
 770                 cookie = NULL;
 771                 while ((slice = avl_destroy_nodes(&slice_cache,
 772                     &cookie)) != NULL) {
 773                         if (slice->rn_config != NULL) {
 774                                 nvlist_t *config = slice->rn_config;
1314                                 boolean_t matched = B_TRUE;

1316                                 if (config == NULL)
1317                                         goto next;

1319 #endif /* ! codereview */
1320                                 if (iarg->poolname != NULL) {
1321                                         char *pname;

1323                                         matched = nvlist_lookup_string(config,
1324                                             ZPOOL_CONFIG_POOL_NAME,
1325                                             &pname) == 0 &&
1326                                             strcmp(iarg->poolname, pname) == 0;
1327                                 } else if (iarg->guid != 0) {
1328                                         uint64_t this_guid;

1330                                         matched = nvlist_lookup_uint64(config,
1331                                             ZPOOL_CONFIG_POOL_GUID,
1332                                             &this_guid) == 0 &&
1333                                             iarg->guid == this_guid;
```



```
1334                                 }

1336 #endif /* ! codereview */
1337                                 if (!matched) {
1338                                         nvlist_free(config);
1339                                         goto next;
 777                                         config = NULL;
 778                                         continue;
1340                                 }

1342 #endif /* ! codereview */
1343                                 /* use the non-raw path for the config */
1344                                 if (curslice != NULL)
1345                                         (void) snprintf(end, pathleft, "%s%s",
1346                                             curdisk->dn_name,
1347                                             curslice->sn_name);
1348                                 else
1349                                         (void) strlcpy(end, curdisk->dn_name,
1350                                             pathleft);
 780                                 (void) strlcpy(end, slice->rn_name, pathleft);
1351                                 if (add_config(hdl, &pools, path, config) != 0)
1352                                         goto error;

1354 next:
1355                                 /*
1356                                  * If we're looking at slices free this one
1357                                  * and go move onto the next.
1358                                  */
1359                                 if (curslice != NULL) {
1360                                         slice_node_t *prevslice;

1362                                         prevslice = curslice;
1363                                         curslice = curslice->sn_next;

1365                                         free(prevslice->sn_name);
1366                                         free(prevslice);

1368                                         if (curslice != NULL) {
1369                                                 config = curslice->sn_config;
1370                                         }
1371 #endif /* ! codereview */
1372                                 }
1373                         } while (curslice != NULL);

1375                         /*
1376                          * Free this disk and move onto the next one.
1377                          */
1378                         prevdisk = curdisk;
1379                         curdisk = curdisk->dn_next;

1381                         free(prevdisk->dn_name);
1382                         free(prevdisk);
 783                         free(slice->rn_name);
 784                         free(slice);
1383                 }
 786                 avl_destroy(&slice_cache);

1385                 (void) closedir(dirp);
1386                 dirp = NULL;
1387         }

1389         ret = get_configs(hdl, &pools, iarg->can_be_active);

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         }

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         }

1414         if (dirp)
1415                 (void) closedir(dirp);

1417         return (ret);
1418 }
```

_____***unchanged_portion_omitted_***

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

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

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright 2014 RackTop Systems.
#endif /* ! codereview */
 */

/*
 * Internal utility routines for the ZFS library.
 */

#include <errno.h>
#include <fcntl.h>
#include <libintl.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <ctype.h>
#include <math.h>
#include <sys/mnttab.h>
#include <sys/mntent.h>
#include <sys/types.h>

#include <libzfs.h>
#include <libzfs_core.h>

#include "libzfs_impl.h"
#include "zfs_prop.h"
#include "zfeature_common.h"

int
libzfs_errno(libzfs_handle_t *hdl)
{
        return (hdl->libzfs_error);
}

const char *
```



```
libzfs_error_action(libzfs_handle_t *hdl)
{
        return (hdl->libzfs_action);
}

const char *
libzfs_error_description(libzfs_handle_t *hdl)
{
        if (hdl->libzfs_desc[0] != '\0')
                return (hdl->libzfs_desc);

        switch (hdl->libzfs_error) {
        case EZFS_NOMEM:
                return (dgettext(TEXT_DOMAIN, "out of memory"));
        case EZFS_BADPROP:
                return (dgettext(TEXT_DOMAIN, "invalid property value"));
        case EZFS_PROPREADONLY:
                return (dgettext(TEXT_DOMAIN, "read-only property"));
        case EZFS_PROPTYPE:
                return (dgettext(TEXT_DOMAIN, "property doesn't apply to "
                    "datasets of this type"));
        case EZFS_PROPNONINHERIT:
                return (dgettext(TEXT_DOMAIN, "property cannot be inherited"));
        case EZFS_PROPSPACE:
                return (dgettext(TEXT_DOMAIN, "invalid quota or reservation"));
        case EZFS_BADTYPE:
                return (dgettext(TEXT_DOMAIN, "operation not applicable to "
                    "datasets of this type"));
        case EZFS_BUSY:
                return (dgettext(TEXT_DOMAIN, "pool or dataset is busy"));
        case EZFS_EXISTS:
                return (dgettext(TEXT_DOMAIN, "pool or dataset exists"));
        case EZFS_NOENT:
                return (dgettext(TEXT_DOMAIN, "no such pool or dataset"));
        case EZFS_BADSTREAM:
                return (dgettext(TEXT_DOMAIN, "invalid backup stream"));
        case EZFS_DSREADONLY:
                return (dgettext(TEXT_DOMAIN, "dataset is read-only"));
        case EZFS_VOLTOOBIG:
                return (dgettext(TEXT_DOMAIN, "volume size exceeds limit for "
                    "this system"));
        case EZFS_INVALIDNAME:
                return (dgettext(TEXT_DOMAIN, "invalid name"));
        case EZFS_BADRESTORE:
                return (dgettext(TEXT_DOMAIN, "unable to restore to "
                    "destination"));
        case EZFS_BADBACKUP:
                return (dgettext(TEXT_DOMAIN, "backup failed"));
        case EZFS_BADTARGET:
                return (dgettext(TEXT_DOMAIN, "invalid target vdev"));
        case EZFS_NODEVICE:
                return (dgettext(TEXT_DOMAIN, "no such device in pool"));
        case EZFS_BADDEV:
                return (dgettext(TEXT_DOMAIN, "invalid device"));
        case EZFS_NOREPLICAS:
                return (dgettext(TEXT_DOMAIN, "no valid replicas"));
        case EZFS_RESILVERING:
                return (dgettext(TEXT_DOMAIN, "currently resilvering"));
        case EZFS_BADVERSION:
                return (dgettext(TEXT_DOMAIN, "unsupported version or "
                    "feature"));
        case EZFS_POOLUNAVAIL:
                return (dgettext(TEXT_DOMAIN, "pool is unavailable"));
        case EZFS_DEVOVERFLOW:
                return (dgettext(TEXT_DOMAIN, "too many devices in one vdev"));
        case EZFS_BADPATH:
```

```
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_UMOUNTFAILED:
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_RELE:
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 }

241 /*PRINTFLIKE2*/
242 void
243 zfs_error_aux(libzfs_handle_t *hdl, const char *fmt, ...)
244 {
245         va_list ap;

247         va_start(ap, fmt);

249         (void) vsnprintf(hdl->libzfs_desc, sizeof (hdl->libzfs_desc),
250             fmt, ap);
251         hdl->libzfs_desc_active = 1;

253         va_end(ap);
254 }

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

```
260             fmt, ap);
261         hdl->libzfs_error = error;
263         if (hdl->libzfs_desc_active)
264                 hdl->libzfs_desc_active = 0;
265         else
266                 hdl->libzfs_desc[0] = '\0';
268         if (hdl->libzfs_printerr) {
269                 if (error == EZFS_UNKNOWN) {
270                         (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "internal "
271                             "error: %s\n"), libzfs_error_description(hdl));
272                         abort();
273                 }
275                 (void) fprintf(stderr, "%s: %s\n", hdl->libzfs_action,
276                     libzfs_error_description(hdl));
277                 if (error == EZFS_NOMEM)
278                         exit(1);
279         }
280 }
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 }
288 /*PRINTFLIKE3*/
289 int
290 zfs_error_fmt(libzfs_handle_t *hdl, int error, const char *fmt, ...)
291 {
292         va_list ap;
294         va_start(ap, fmt);
296         zfs_verror(hdl, error, fmt, ap);
298         va_end(ap);
300         return (-1);
301 }
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);
313         case ECANCELED:
314                 zfs_verror(hdl, EZFS_NODELEGATION, fmt, ap);
315                 return (-1);
317         case EIO:
318                 zfs_verror(hdl, EZFS_IO, fmt, ap);
319                 return (-1);
321         case EFAULT:
322                 zfs_verror(hdl, EZFS_FAULT, fmt, ap);
323                 return (-1);
325         case EINTR:
```



```
326                 zfs_verror(hdl, EZFS_INTR, fmt, ap);
327                 return (-1);
328         }
330         return (0);
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 }
339 /*PRINTFLIKE3*/
340 int
341 zfs_standard_error_fmt(libzfs_handle_t *hdl, int error, const char *fmt, ...)
342 {
343         va_list ap;
345         va_start(ap, fmt);
347         if (zfs_common_error(hdl, error, fmt, ap) != 0) {
348                 va_end(ap);
349                 return (-1);
350         }
352         switch (error) {
353         case ENXIO:
354         case ENODEV:
355         case EPIPE:
356                 zfs_verror(hdl, EZFS_IO, fmt, ap);
357                 break;
359         case ENOENT:
360                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
361                     "dataset does not exist"));
362                 zfs_verror(hdl, EZFS_NOENT, fmt, ap);
363                 break;
365         case ENOSPC:
366         case EDQUOT:
367                 zfs_verror(hdl, EZFS_NOSPC, fmt, ap);
368                 return (-1);
370         case EEXIST:
371                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
372                     "dataset already exists"));
373                 zfs_verror(hdl, EZFS_EXISTS, fmt, ap);
374                 break;
376         case EBUSY:
377                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
378                     "dataset is busy"));
379                 zfs_verror(hdl, EZFS_BUSY, fmt, ap);
380                 break;
381         case EROFS:
382                 zfs_verror(hdl, EZFS_POOLREADONLY, fmt, ap);
383                 break;
384         case ENAMETOOLONG:
385                 zfs_verror(hdl, EZFS_NAMETOOLONG, fmt, ap);
386                 break;
387         case ENOTSUP:
388                 zfs_verror(hdl, EZFS_BADVERSION, fmt, ap);
389                 break;
390         case EAGAIN:
391                 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         }

401         va_end(ap);
402         return (-1);
403 }

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 }

411 /*PRINTFLIKE3*/
412 int
413 zpool_standard_error_fmt(libzfs_handle_t *hdl, int error, const char *fmt, ...)
414 {
415         va_list ap;

417         va_start(ap, fmt);

419         if (zfs_common_error(hdl, error, fmt, ap) != 0) {
420                 va_end(ap);
421                 return (-1);
422         }

424         switch (error) {
425         case ENODEV:
426                 zfs_verror(hdl, EZFS_NODEVICE, fmt, ap);
427                 break;

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;

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;

441         case EBUSY:
442                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "pool is busy"));
443                 zfs_verror(hdl, EZFS_BUSY, fmt, ap);
444                 break;

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;

452         case ENAMETOOLONG:
453                 zfs_verror(hdl, EZFS_DEVOVERFLOW, fmt, ap);
454                 break;

456         case ENOTSUP:
457                 zfs_verror(hdl, EZFS_POOL_NOTSUP, fmt, ap);
```



```
458                 break;

460         case EINVAL:
461                 zfs_verror(hdl, EZFS_POOL_INVALARG, fmt, ap);
462                 break;

464         case ENOSPC:
465         case EDQUOT:
466                 zfs_verror(hdl, EZFS_NOSPC, fmt, ap);
467                 return (-1);

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;

475         case EROFS:
476                 zfs_verror(hdl, EZFS_POOLREADONLY, fmt, ap);
477                 break;

479         default:
480                 zfs_error_aux(hdl, strerror(error));
481                 zfs_verror(hdl, EZFS_UNKNOWN, fmt, ap);
482         }

484         va_end(ap);
485         return (-1);
486 }

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 }

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;

505         if ((data = calloc(1, size)) == NULL)
506                 (void) no_memory(hdl);

508         return (data);
509 }

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;

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_printerr = 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 }
669
670 libzfs_handle_t *
671 zpool_get_handle(zpool_handle_t *zhp)
672 {
673         return (zhp->zpool_hdl);
674 }
675
676 libzfs_handle_t *
677 zfs_get_handle(zfs_handle_t *zhp)
678 {
679         return (zhp->zfs_hdl);
680 }
681
682 zpool_handle_t *
683 zfs_get_pool_handle(const zfs_handle_t *zhp)
684 {
685         return (zhp->zpool_hdl);
686 }
687
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
701         if (path[0] != '/' && strncmp(path, "./", strlen("./")) != 0) {
702                 /*
703                  * It's not a valid path, assume it's a name of type 'argtype'.
704                  */
705                 return (zfs_open(hdl, path, argtype));
706         }
707
708         if (stat64(path, &statbuf) != 0) {
709                 (void) fprintf(stderr, "%s: %s\n", path, strerror(errno));
710                 return (NULL);
711         }
712
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         }
723
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         }
729
730         return (zfs_open(hdl, entry.mnt_special, ZFS_TYPE_FILESYSTEM));
731 }
732
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);
746
747         return (0);
748 }
749
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 *)(uintptr_t)zc->zc_nvlist_dst);
759         if ((zc->zc_nvlist_dst = (uint64_t)(uintptr_t)
760             zfs_alloc(hdl, zc->zc_nvlist_dst_size))
761             == NULL)
762                 return (-1);
763
764         return (0);
765 }
766
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 *)(uintptr_t)zc->zc_nvlist_conf);
774         free((void *)(uintptr_t)zc->zc_nvlist_src);
775         free((void *)(uintptr_t)zc->zc_nvlist_dst);
776 }
777
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;
784
785         verify(nvlist_size(nvl, &len, NV_ENCODE_NATIVE) == 0);
786
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 **nvlp)
817 {
818         if (nvlist_unpack((void *)(uintptr_t)zc->zc_nvlist_dst,
819             zc->zc_nvlist_dst_size, nvlp, 0) != 0)
820                 return (no_memory(hdl));
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;
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_RECVD] = 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));
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_INVAL) {
879                         const char *propname = (type == ZFS_TYPE_POOL) ?
880                             zpool_prop_to_name(pl->pl_prop) :
881                             zfs_prop_to_name(pl->pl_prop);
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                 }
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;
903                 /* 'RECEIVED' column. */
904                 if (pl != cbp->cb_proplist &&
905                     pl->pl_recvd_width > cbp->cb_colwidths[GET_COL_RECVD])
906                         cbp->cb_colwidths[GET_COL_RECVD] = pl->pl_recvd_width;
908                 /*
909                  * 'NAME' and 'SOURCE' columns
910                  */
911                 if (pl->pl_prop == (type == ZFS_TYPE_POOL ? ZPOOL_PROP_NAME :
912                     ZFS_PROP_NAME) &&
913                     pl->pl_width > cbp->cb_colwidths[GET_COL_NAME]) {
914                         cbp->cb_colwidths[GET_COL_NAME] = pl->pl_width;
915                         cbp->cb_colwidths[GET_COL_SOURCE] = pl->pl_width +
916                             strlen(dgettext(TEXT_DOMAIN, "inherited from"));
917                 }
918         }
```

```
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                 }

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 }

957 /*
958  * Display a single line of output, according to the settings in the callback
959  * structure.
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 *recvd_value)
965 {
966         int i;
967         const char *str = NULL;
 26         const char *str;
968         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;

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

979         for (i = 0; i < ZFS_GET_NCOLS; i++) {
980                 switch (cbp->cb_columns[i]) {
981                 case GET_COL_NAME:
982                         str = name;
983                         break;
```



```
985                 case GET_COL_PROPERTY:
986                         str = propname;
987                         break;

989                 case GET_COL_VALUE:
990                         str = value;
991                         break;

993                 case GET_COL_SOURCE:
994                         switch (sourcetype) {
995                         case ZPROP_SRC_NONE:
996                                 str = "-";
997                                 break;

999                         case ZPROP_SRC_DEFAULT:
1000                                str = "default";
1001                                break;

1003                        case ZPROP_SRC_LOCAL:
1004                                str = "local";
1005                                break;

1007                        case ZPROP_SRC_TEMPORARY:
1008                                str = "temporary";
1009                                break;

1011                        case ZPROP_SRC_INHERITED:
1012                                (void) snprintf(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;

1022                case GET_COL_RECVD:
1023                        str = (recvd_value == NULL ? "-" : recvd_value);
1024                        break;

1026                default:
1027                        continue;
1028                }

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        }

1040        (void) printf("\n");
1041 }
```

*unchanged_portion_omitted_*