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

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

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

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

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

#include "zfs_namecheck.h"
```

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

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

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

        return (NULL);
}

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

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

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

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

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

        (void) zfs_prop_get_table();
        if (dataset_namecheck(path, &why, &what) != 0) {
                if (hdl != NULL) {
                        switch (why) {
                        case NAME_ERR_TOOLONG:
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "name is too long"));
                                break;

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

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

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

                        case NAME_ERR_INVALCHAR:
                                zfs_error_aux(hdl,
                                    dgettext(TEXT_DOMAIN, "invalid character "
                                    "’%c’ in name"), what);
                                break;

                        case NAME_ERR_MULTIPLE_AT:
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "multiple ’@’ delimiters in name"));
                                break;

                        case NAME_ERR_NOLETTER:
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "pool doesn’t begin with a letter"));
                                break;

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

                        case NAME_ERR_DISKLIKE:
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "reserved disk name"));
                                break;
                        }
                }
```

```
                return (0);
        }

        if (!(type & ZFS_TYPE_SNAPSHOT) && strchr(path, ’@’) != NULL) {
                if (hdl != NULL)
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "snapshot delimiter ’@’ in filesystem name"));
                return (0);
        }

        if (type == ZFS_TYPE_SNAPSHOT && strchr(path, ’@’) == NULL) {
                if (hdl != NULL)
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "missing ’@’ delimiter in snapshot name"));
                return (0);
        }

        if (modifying && strchr(path, ’%’) != NULL) {
                if (hdl != NULL)
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "invalid character %c in name"), ’%’);
                return (0);
        }

        return (-1);
}

int
zfs_name_valid(const char *name, zfs_type_t type)
{
        if (type == ZFS_TYPE_POOL)
                return (zpool_name_valid(NULL, B_FALSE, name));
        return (zfs_validate_name(NULL, name, type, B_FALSE));
}

/*
 * This function takes the raw DSL properties, and filters out the user-defined
 * properties into a separate nvlist.
 */
static nvlist_t *
process_user_props(zfs_handle_t *zhp, nvlist_t *props)
{
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        nvpair_t *elem;
        nvlist_t *propval;
        nvlist_t *nvl;

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

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

                verify(nvpair_value_nvlist(elem, &propval) == 0);
                if (nvlist_add_nvlist(nvl, nvpair_name(elem), propval) != 0) {
                        nvlist_free(nvl);
                        (void) no_memory(hdl);
                        return (NULL);
                }
        }
        return (nvl);
}

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

        if ((zph = zpool_open_canfail(hdl, pool_name)) != NULL) {
                if (hdl->libzfs_pool_handles != NULL)
                        zph->zpool_next = hdl->libzfs_pool_handles;
                hdl->libzfs_pool_handles = zph;
        }
        return (zph);
}

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

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

/*
 * Returns a handle to the pool that contains the provided dataset.
 * If a handle to that pool already exists then that handle is returned.
 * Otherwise, a new handle is created and added to the list of handles.
 */
static zpool_handle_t *
zpool_handle(zfs_handle_t *zhp)
{
        char *pool_name;
        int len;
        zpool_handle_t *zph;

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

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

        free(pool_name);
        return (zph);
}

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

        while (zph != NULL) {
                next = zph->zpool_next;
                zpool_close(zph);
                zph = next;
        }
        hdl->libzfs_pool_handles = NULL;
}
```

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

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

        while (ioctl(hdl->libzfs_fd, ZFS_IOC_OBJSET_STATS, zc) != 0) {
                if (errno == ENOMEM) {
                        if (zcmd_expand_dst_nvlist(hdl, zc) != 0) {
                                return (-1);
                        }
                } else {
                        return (-1);
                }
        }
        return (0);
}

/*
 * Utility function to get the received properties of the given object.
 */
static int
get_recvd_props_ioctl(zfs_handle_t *zhp)
{
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        nvlist_t *recvdprops;
        zfs_cmd_t zc = { 0 };
        int err;

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

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

        while (ioctl(hdl->libzfs_fd, ZFS_IOC_OBJSET_RECVD_PROPS, &zc) != 0) {
                if (errno == ENOMEM) {
                        if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
                                return (-1);
                        }
                } else {
                        zcmd_free_nvlists(&zc);
                        return (-1);
                }
        }

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

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

        return (0);
}

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

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

```c
	if (zcmd_read_dst_nvlist(zhp->zfs_hdl, zc, &allprops) != 0) {
		return (-1);
	}

	/*
	 * XXX Why do we store the user props separately, in addition to
	 * storing them in zfs_props?
	 */
	if ((userprops = process_user_props(zhp, allprops)) == NULL) {
		nvlist_free(allprops);
		return (-1);
	}

	nvlist_free(zhp->zfs_props);
	nvlist_free(zhp->zfs_user_props);

	zhp->zfs_props = allprops;
	zhp->zfs_user_props = userprops;

	return (0);
}

static int
get_stats(zfs_handle_t *zhp)
{
	int rc = 0;
	zfs_cmd_t zc = { 0 };

	if (zcmd_alloc_dst_nvlist(zhp->zfs_hdl, &zc, 0) != 0)
		return (-1);
	if (get_stats_ioctl(zhp, &zc) != 0)
		rc = -1;
	else if (put_stats_zhdl(zhp, &zc) != 0)
		rc = -1;
	zcmd_free_nvlists(&zc);
	return (rc);
}

/*
 * Refresh the properties currently stored in the handle.
 */
void
zfs_refresh_properties(zfs_handle_t *zhp)
{
	(void) get_stats(zhp);
}

/*
 * Makes a handle from the given dataset name.  Used by zfs_open() and
 * zfs_iter_* to create child handles on the fly.
 */
static int
make_dataset_handle_common(zfs_handle_t *zhp, zfs_cmd_t *zc)
{
	if (put_stats_zhdl(zhp, zc) != 0)
		return (-1);

	/*
	 * We’ve managed to open the dataset and gather statistics.  Determine
	 * the high-level type.
	 */
	if (zhp->zfs_dmustats.dds_type == DMU_OST_ZVOL)
		zhp->zfs_head_type = ZFS_TYPE_VOLUME;
	else if (zhp->zfs_dmustats.dds_type == DMU_OST_ZFS)
		zhp->zfs_head_type = ZFS_TYPE_FILESYSTEM;
```

```c
	else
		abort();

	if (zhp->zfs_dmustats.dds_is_snapshot)
		zhp->zfs_type = ZFS_TYPE_SNAPSHOT;
	else if (zhp->zfs_dmustats.dds_type == DMU_OST_ZVOL)
		zhp->zfs_type = ZFS_TYPE_VOLUME;
	else if (zhp->zfs_dmustats.dds_type == DMU_OST_ZFS)
		zhp->zfs_type = ZFS_TYPE_FILESYSTEM;
	else
		abort();	/* we should never see any other types */

	if ((zhp->zpool_hdl = zpool_handle(zhp)) == NULL)
		return (-1);

	return (0);
}

zfs_handle_t *
make_dataset_handle(libzfs_handle_t *hdl, const char *path)
{
	zfs_cmd_t zc = { 0 };

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

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

	zhp->zfs_hdl = hdl;
	(void) strlcpy(zhp->zfs_name, path, sizeof (zhp->zfs_name));
	if (zcmd_alloc_dst_nvlist(hdl, &zc, 0) != 0) {
		free(zhp);
		return (NULL);
	}
	if (get_stats_ioctl(zhp, &zc) == -1) {
		zcmd_free_nvlists(&zc);
		free(zhp);
		return (NULL);
	}
	if (make_dataset_handle_common(zhp, &zc) == -1) {
		free(zhp);
		zhp = NULL;
	}
	zcmd_free_nvlists(&zc);
	return (zhp);
}

zfs_handle_t *
make_dataset_handle_zc(libzfs_handle_t *hdl, zfs_cmd_t *zc)
{
	zfs_handle_t *zhp = calloc(sizeof (zfs_handle_t), 1);

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

	zhp->zfs_hdl = hdl;
	(void) strlcpy(zhp->zfs_name, zc->zc_name, sizeof (zhp->zfs_name));
	if (make_dataset_handle_common(zhp, zc) == -1) {
		free(zhp);
		return (NULL);
	}
	return (zhp);
}

zfs_handle_t *
zfs_handle_dup(zfs_handle_t *zhp_orig)
```

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

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

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

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

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

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

```
                return (NULL);
        }

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

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

        return (zhp);
}

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

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

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

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

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

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

void
libzfs_mnttab_update(libzfs_handle_t *hdl)
{
        struct mnttab entry;
        rewind(hdl->libzfs_mnttab);
        while (getmntent(hdl->libzfs_mnttab, &entry) == 0) {
                mnttab_node_t *mtn;

                if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0)
                        continue;
                mtn = zfs_alloc(hdl, sizeof (mnttab_node_t));
                mtn->mtn_mt.mnt_special = zfs_strdup(hdl, entry.mnt_special);
                mtn->mtn_mt.mnt_mountp = zfs_strdup(hdl, entry.mnt_mountp);
                mtn->mtn_mt.mnt_fstype = zfs_strdup(hdl, entry.mnt_fstype);
                mtn->mtn_mt.mnt_mntopts = zfs_strdup(hdl, entry.mnt_mntopts);
                avl_add(&hdl->libzfs_mnttab_cache, mtn);
        }
}

void
libzfs_mnttab_fini(libzfs_handle_t *hdl)
{
        void *cookie = NULL;
        mnttab_node_t *mtn;

        while (mtn = avl_destroy_nodes(&hdl->libzfs_mnttab_cache, &cookie)) {
                free(mtn->mtn_mt.mnt_special);
                free(mtn->mtn_mt.mnt_mountp);
                free(mtn->mtn_mt.mnt_fstype);
                free(mtn->mtn_mt.mnt_mntopts);
                free(mtn);
        }
        avl_destroy(&hdl->libzfs_mnttab_cache);
}

void
libzfs_mnttab_cache(libzfs_handle_t *hdl, boolean_t enable)
{
        hdl->libzfs_mnttab_enable = enable;
}

int
libzfs_mnttab_find(libzfs_handle_t *hdl, const char *fsname,
    struct mnttab *entry)
{
        mnttab_node_t find;
        mnttab_node_t *mtn;

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

                if (avl_numnodes(&hdl->libzfs_mnttab_cache))
                        libzfs_mnttab_fini(hdl);
                rewind(hdl->libzfs_mnttab);
                srch.mnt_special = (char *)fsname;
                srch.mnt_fstype = MNTTYPE_ZFS;
                if (getmntany(hdl->libzfs_mnttab, entry, &srch) == 0)
                        return (0);
                else
                        return (ENOENT);
        }

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

        find.mtn_mt.mnt_special = (char *)fsname;
        mtn = avl_find(&hdl->libzfs_mnttab_cache, &find, NULL);
        if (mtn) {
                *entry = mtn->mtn_mt;
```

```
                return (0);
        }
        return (ENOENT);
}

void
libzfs_mnttab_add(libzfs_handle_t *hdl, const char *special,
    const char *mountp, const char *mntopts)
{
        mnttab_node_t *mtn;

        if (avl_numnodes(&hdl->libzfs_mnttab_cache) == 0)
                return;
        mtn = zfs_alloc(hdl, sizeof (mnttab_node_t));
        mtn->mtn_mt.mnt_special = zfs_strdup(hdl, special);
        mtn->mtn_mt.mnt_mountp = zfs_strdup(hdl, mountp);
        mtn->mtn_mt.mnt_fstype = zfs_strdup(hdl, MNTTYPE_ZFS);
        mtn->mtn_mt.mnt_mntopts = zfs_strdup(hdl, mntopts);
        avl_add(&hdl->libzfs_mnttab_cache, mtn);
}

void
libzfs_mnttab_remove(libzfs_handle_t *hdl, const char *fsname)
{
        mnttab_node_t find;
        mnttab_node_t *ret;

        find.mtn_mt.mnt_special = (char *)fsname;
        if (ret = avl_find(&hdl->libzfs_mnttab_cache, (void *)&find, NULL)) {
                avl_remove(&hdl->libzfs_mnttab_cache, ret);
                free(ret->mtn_mt.mnt_special);
                free(ret->mtn_mt.mnt_mountp);
                free(ret->mtn_mt.mnt_fstype);
                free(ret->mtn_mt.mnt_mntopts);
                free(ret);
        }
}

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

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

        *spa_version = zpool_get_prop_int(zpool_handle,
            ZPOOL_PROP_VERSION, NULL);
        return (0);
}

/*
 * The choice of reservation property depends on the SPA version.
 */
static int
zfs_which_resv_prop(zfs_handle_t *zhp, zfs_prop_t *resv_prop)
{
        int spa_version;

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

        if (spa_version >= SPA_VERSION_REFRESERVATION)
                *resv_prop = ZFS_PROP_REFRESERVATION;
        else
                *resv_prop = ZFS_PROP_RESERVATION;
```

```c
        return (0);
}

/*
 * Given an nvlist of properties to set, validates that they are correct, and
 * parses any numeric properties (index, boolean, etc) if they are specified as
 * strings.
 */
nvlist_t *
zfs_valid_proplist(libzfs_handle_t *hdl, zfs_type_t type, nvlist_t *nvl,
    uint64_t zoned, zfs_handle_t *zhp, const char *errbuf)
{
        nvpair_t *elem;
        uint64_t intval;
        char *strval;
        zfs_prop_t prop;
        nvlist_t *ret;
        int chosen_normal = -1;
        int chosen_utf = -1;

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

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

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

                prop = zfs_name_to_prop(propname);
                if (prop == ZPROP_INVAL && zfs_prop_user(propname)) {
                        /*
                         * This is a user property: make sure it’s a
                         * string, and that it’s less than ZAP_MAXNAMELEN.
                         */
                        if (nvpair_type(elem) != DATA_TYPE_STRING) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "’%s’ must be a string"), propname);
                                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                                goto error;
                        }

                        if (strlen(nvpair_name(elem)) >= ZAP_MAXNAMELEN) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "property name ’%s’ is too long"),
                                    propname);
                                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                                goto error;
                        }

                        (void) nvpair_value_string(elem, &strval);
                        if (nvlist_add_string(ret, propname, strval) != 0) {
                                (void) no_memory(hdl);
                                goto error;
                        }
                        continue;
                }

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

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

                if (prop == ZPROP_INVAL && zfs_prop_userquota(propname)) {
                        zfs_userquota_prop_t uqtype;
                        char newpropname[128];
                        char domain[128];
                        uint64_t rid;
                        uint64_t valary[3];

                        if (userquota_propname_decode(propname, zoned,
                            &uqtype, domain, sizeof (domain), &rid) != 0) {
                                zfs_error_aux(hdl,
                                    dgettext(TEXT_DOMAIN,
                                    "’%s’ has an invalid user/group name"),
                                    propname);
                                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                                goto error;
                        }

                        if (uqtype != ZFS_PROP_USERQUOTA &&
                            uqtype != ZFS_PROP_GROUPQUOTA) {
                                zfs_error_aux(hdl,
                                    dgettext(TEXT_DOMAIN, "’%s’ is readonly"),
                                    propname);
                                (void) zfs_error(hdl, EZFS_PROPREADONLY,
                                    errbuf);
                                goto error;
                        }

                        if (nvpair_type(elem) == DATA_TYPE_STRING) {
                                (void) nvpair_value_string(elem, &strval);
                                if (strcmp(strval, "none") == 0) {
                                        intval = 0;
                                } else if (zfs_nicestrtonum(hdl,
                                    strval, &intval) != 0) {
                                        (void) zfs_error(hdl,
                                            EZFS_BADPROP, errbuf);
                                        goto error;
                                }
                        } else if (nvpair_type(elem) ==
                            DATA_TYPE_UINT64) {
                                (void) nvpair_value_uint64(elem, &intval);
                                if (intval == 0) {
                                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                            "use ’none’ to disable "
                                            "userquota/groupquota"));
                                        goto error;
                                }
                        } else {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "’%s’ must be a number"), propname);
                                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                                goto error;
                        }

                        /*
                         * Encode the prop name as
                         * userquota@<hex-rid>-domain, to make it easy
                         * for the kernel to decode.
                         */
                        (void) snprintf(newpropname, sizeof (newpropname),
                            "%s%llx-%s", zfs_userquota_prop_prefixes[uqtype],
                            (longlong_t)rid, domain);
                        valary[0] = uqtype;
                        valary[1] = rid;
                        valary[2] = intval;
                        if (nvlist_add_uint64_array(ret, newpropname,
                            valary, 3) != 0) {
                                (void) no_memory(hdl);
                                goto error;
                        }
                        continue;
                } else if (prop == ZPROP_INVAL && zfs_prop_written(propname)) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "’%s’ is readonly"),
                            propname);
                        (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
                        goto error;
                }

                if (prop == ZPROP_INVAL) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "invalid property ’%s’"), propname);
                        (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                        goto error;
                }

                if (!zfs_prop_valid_for_type(prop, type)) {
                        zfs_error_aux(hdl,
                            dgettext(TEXT_DOMAIN, "’%s’ does not "
                            "apply to datasets of this type"), propname);
                        (void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
                        goto error;
                }

                if (zfs_prop_readonly(prop) &&
                    (!zfs_prop_setonce(prop) || zhp != NULL)) {
                        zfs_error_aux(hdl,
                            dgettext(TEXT_DOMAIN, "’%s’ is readonly"),
                            propname);
                        (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
                        goto error;
                }

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

                /*
                 * Perform some additional checks for specific properties.
                 */
                switch (prop) {
                case ZFS_PROP_VERSION:
                {
                        int version;

                        if (zhp == NULL)
                                break;
                        version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
                        if (intval < version) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "Can not downgrade; already at version %u"),
                                    version);
                                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                                goto error;
                        }
```

```
                        break;
                }

                case ZFS_PROP_RECORDSIZE:
                case ZFS_PROP_VOLBLOCKSIZE:
                        /* must be power of two within SPA_{MIN,MAX}BLOCKSIZE */
                        if (intval < SPA_MINBLOCKSIZE ||
                            intval > SPA_MAXBLOCKSIZE || !ISP2(intval)) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "’%s’ must be power of 2 from %u "
                                    "to %uk"), propname,
                                    (uint_t)SPA_MINBLOCKSIZE,
                                    (uint_t)SPA_MAXBLOCKSIZE >> 10);
                                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                                goto error;
                        }
                        break;

                case ZFS_PROP_MLSLABEL:
                {
                        /*
                         * Verify the mlslabel string and convert to
                         * internal hex label string.
                         */

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

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

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

                        /* Now translate to hex internal label string */
                        if (label_to_str(new_sl, &hex, M_INTERNAL,
                            DEF_NAMES) != 0) {
                                if (hex)
                                        free(hex);
                                goto badlabel;
                        }
                        m_label_free(new_sl);

                        /* If string is already in internal form, we’re done. */
                        if (strcmp(strval, hex) == 0) {
                                free(hex);
                                break;
                        }

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

                        break;

badlabel:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "invalid mlslabel ’%s’"), strval);
                        (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                        m_label_free(new_sl);   /* OK if null */
                        goto error;

                }

                case ZFS_PROP_MOUNTPOINT:
                {
                        namecheck_err_t why;

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

                        if (mountpoint_namecheck(strval, &why)) {
                                switch (why) {
                                case NAME_ERR_LEADING_SLASH:
                                        zfs_error_aux(hdl,
                                            dgettext(TEXT_DOMAIN,
                                            "’%s’ must be an absolute path, "
                                            "’none’, or ’legacy’"), propname);
                                        break;
                                case NAME_ERR_TOOLONG:
                                        zfs_error_aux(hdl,
                                            dgettext(TEXT_DOMAIN,
                                            "component of ’%s’ is too long"),
                                            propname);
                                        break;
                                }
                                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                                goto error;
                        }
                }

                        /*FALLTHRU*/

                case ZFS_PROP_SHARESMB:
                case ZFS_PROP_SHARENFS:
                        /*
                         * For the mountpoint and sharenfs or sharesmb
                         * properties, check if it can be set in a
                         * global/non-global zone based on
                         * the zoned property value:
                         *
                         *              global zone         non-global zone
                         * -------------------------------------------------
                         * zoned=on     mountpoint (no)     mountpoint (yes)
                         *              sharenfs (no)       sharenfs (no)
                         *              sharesmb (no)       sharesmb (no)
                         *
                         * zoned=off    mountpoint (yes)        N/A
                         *              sharenfs (yes)
                         *              sharesmb (yes)
                         */
                        if (zoned) {
                                if (getzoneid() == GLOBAL_ZONEID) {
                                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                            "’%s’ cannot be set on "
                                            "dataset in a non-global zone"),
                                            propname);
                                        (void) zfs_error(hdl, EZFS_ZONED,
                                            errbuf);
                                        goto error;
                                } else if (prop == ZFS_PROP_SHARENFS ||
                                    prop == ZFS_PROP_SHARESMB) {
                                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
```

```
                                            "’%s’ cannot be set in "
                                            "a non-global zone"), propname);
                                        (void) zfs_error(hdl, EZFS_ZONED,
                                            errbuf);
                                        goto error;
                                }
                        } else if (getzoneid() != GLOBAL_ZONEID) {
                                /*
                                 * If zoned property is ’off’, this must be in
                                 * a global zone. If not, something is wrong.
                                 */
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "’%s’ cannot be set while dataset "
                                    "’zoned’ property is set"), propname);
                                (void) zfs_error(hdl, EZFS_ZONED, errbuf);
                                goto error;
                        }

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

                                if (prop == ZFS_PROP_SHARESMB)
                                        proto = PROTO_SMB;
                                else
                                        proto = PROTO_NFS;

                                /*
                                 * Must be an valid sharing protocol
                                 * option string so init the libshare
                                 * in order to enable the parser and
                                 * then parse the options. We use the
                                 * control API since we don’t care about
                                 * the current configuration and don’t
                                 * want the overhead of loading it
                                 * until we actually do something.
                                 */

                                if (zfs_init_libshare(hdl,
                                    SA_INIT_CONTROL_API) != SA_OK) {
                                        /*
                                         * An error occurred so we can’t do
                                         * anything
                                         */
                                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                            "’%s’ cannot be set: problem "
                                            "in share initialization"),
                                            propname);
                                        (void) zfs_error(hdl, EZFS_BADPROP,
                                            errbuf);
                                        goto error;
                                }

                                if (zfs_parse_options(strval, proto) != SA_OK) {
                                        /*
                                         * There was an error in parsing so
                                         * deal with it by issuing an error
                                         * message and leaving after
                                         * uninitializing the the libshare
                                         * interface.
                                         */
                                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                            "’%s’ cannot be set to invalid "
                                            "options"), propname);
                                        (void) zfs_error(hdl, EZFS_BADPROP,
                                            errbuf);
                                        zfs_uninit_libshare(hdl);
                                        goto error;
                                }
                                zfs_uninit_libshare(hdl);
                        }

                        break;
                case ZFS_PROP_UTF8ONLY:
                        chosen_utf = (int)intval;
                        break;
                case ZFS_PROP_NORMALIZE:
                        chosen_normal = (int)intval;
                        break;
                }

                /*
                 * For changes to existing volumes, we have some additional
                 * checks to enforce.
                 */
                if (type == ZFS_TYPE_VOLUME && zhp != NULL) {
                        uint64_t volsize = zfs_prop_get_int(zhp,
                            ZFS_PROP_VOLSIZE);
                        uint64_t blocksize = zfs_prop_get_int(zhp,
                            ZFS_PROP_VOLBLOCKSIZE);
                        char buf[64];

                        switch (prop) {
                        case ZFS_PROP_RESERVATION:
                        case ZFS_PROP_REFRESERVATION:
                                if (intval > volsize) {
                                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                            "’%s’ is greater than current "
                                            "volume size"), propname);
                                        (void) zfs_error(hdl, EZFS_BADPROP,
                                            errbuf);
                                        goto error;
                                }
                                break;

                        case ZFS_PROP_VOLSIZE:
                                if (intval % blocksize != 0) {
                                        zfs_nicenum(blocksize, buf,
                                            sizeof (buf));
                                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                            "’%s’ must be a multiple of "
                                            "volume block size (%s)"),
                                            propname, buf);
                                        (void) zfs_error(hdl, EZFS_BADPROP,
                                            errbuf);
                                        goto error;
                                }

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

```
                                }
                                break;
                        }
                }
        }

        /*
         * If normalization was chosen, but no UTF8 choice was made,
         * enforce rejection of non-UTF8 names.
         *
         * If normalization was chosen, but rejecting non-UTF8 names
         * was explicitly not chosen, it is an error.
         */
        if (chosen_normal > 0 && chosen_utf < 0) {
                if (nvlist_add_uint64(ret,
                    zfs_prop_to_name(ZFS_PROP_UTF8ONLY), 1) != 0) {
                        (void) no_memory(hdl);
                        goto error;
                }
        } else if (chosen_normal > 0 && chosen_utf == 0) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                    "’%s’ must be set ’on’ if normalization chosen"),
                    zfs_prop_to_name(ZFS_PROP_UTF8ONLY));
                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                goto error;
        }
        return (ret);

error:
        nvlist_free(ret);
        return (NULL);
}

int
zfs_add_synthetic_resv(zfs_handle_t *zhp, nvlist_t *nvl)
{
        uint64_t old_volsize;
        uint64_t new_volsize;
        uint64_t old_reservation;
        uint64_t new_reservation;
        zfs_prop_t resv_prop;
        nvlist_t *props;

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

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

        if ((zvol_volsize_to_reservation(old_volsize, props) !=
            old_reservation) || nvlist_exists(nvl,
            zfs_prop_to_name(resv_prop))) {
                fnvlist_free(props);
                return (0);
        }
        if (nvlist_lookup_uint64(nvl, zfs_prop_to_name(ZFS_PROP_VOLSIZE),
            &new_volsize) != 0) {
                fnvlist_free(props);
                return (-1);
        }
        new_reservation = zvol_volsize_to_reservation(new_volsize, props);
        fnvlist_free(props);

        if (nvlist_add_uint64(nvl, zfs_prop_to_name(resv_prop),
            new_reservation) != 0) {
                (void) no_memory(zhp->zfs_hdl);
                return (-1);
        }
        return (1);
}

void
zfs_setprop_error(libzfs_handle_t *hdl, zfs_prop_t prop, int err,
    char *errbuf)
{
        switch (err) {

        case ENOSPC:
                /*
                 * For quotas and reservations, ENOSPC indicates
                 * something different; setting a quota or reservation
                 * doesn’t use any disk space.
                 */
                switch (prop) {
                case ZFS_PROP_QUOTA:
                case ZFS_PROP_REFQUOTA:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "size is less than current used or "
                            "reserved space"));
                        (void) zfs_error(hdl, EZFS_PROPSPACE, errbuf);
                        break;

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

                default:
                        (void) zfs_standard_error(hdl, err, errbuf);
                        break;
                }
                break;

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

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

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

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

```
                        (void) zfs_error(hdl, EZFS_NOTSUP, errbuf);
                } else {
                        (void) zfs_standard_error(hdl, err, errbuf);
                }
                break;

        case EINVAL:
                if (prop == ZPROP_INVAL) {
                        (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                } else {
                        (void) zfs_standard_error(hdl, err, errbuf);
                }
                break;

        case EOVERFLOW:
                /*
                 * This platform can’t address a volume this big.
                 */
#ifdef _ILP32
                if (prop == ZFS_PROP_VOLSIZE) {
                        (void) zfs_error(hdl, EZFS_VOLTOOBIG, errbuf);
                        break;
                }
#endif
                /* FALLTHROUGH */
        default:
                (void) zfs_standard_error(hdl, err, errbuf);
        }
}

/*
 * Given a property name and value, set the property for the given dataset.
 */
int
zfs_prop_set(zfs_handle_t *zhp, const char *propname, const char *propval)
{
        zfs_cmd_t zc = { 0 };
        int ret = -1;
        prop_changelist_t *cl = NULL;
        char errbuf[1024];
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        nvlist_t *nvl = NULL, *realprops;
        zfs_prop_t prop;
        boolean_t do_prefix = B_TRUE;
        int added_resv;

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

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

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

        nvlist_free(nvl);
        nvl = realprops;

        prop = zfs_name_to_prop(propname);

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

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

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

        /*
         * We don’t want to unmount & remount the dataset when changing
         * its canmount property to ’on’ or ’noauto’.  We only use
         * the changelist logic to unmount when setting canmount=off.
         */
        if (prop == ZFS_PROP_CANMOUNT) {
                uint64_t idx;
                int err = zprop_string_to_index(prop, propval, &idx,
                    ZFS_TYPE_DATASET);
                if (err == 0 && idx != ZFS_CANMOUNT_OFF)
                        do_prefix = B_FALSE;
        }

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

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

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

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

        if (ret != 0) {
                zfs_setprop_error(hdl, prop, errno, errbuf);
                if (added_resv && errno == ENOSPC) {
                        /* clean up the volsize property we tried to set */
                        uint64_t old_volsize = zfs_prop_get_int(zhp,
                            ZFS_PROP_VOLSIZE);
                        nvlist_free(nvl);
                        zcmd_free_nvlists(&zc);
                        if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
                                goto error;
                        if (nvlist_add_uint64(nvl,
                            zfs_prop_to_name(ZFS_PROP_VOLSIZE),
                            old_volsize) != 0)
                                goto error;
                        if (zcmd_write_src_nvlist(hdl, &zc, nvl) != 0)
                                goto error;
                        (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc);
                }
        } else {
                if (do_prefix)
                        ret = changelist_postfix(cl);

                /*
                 * Refresh the statistics so the new property value
                 * is reflected.
```

```
                 */
                if (ret == 0)
                        (void) get_stats(zhp);
        }

error:
        nvlist_free(nvl);
        zcmd_free_nvlists(&zc);
        if (cl)
                changelist_free(cl);
        return (ret);
}

/*
 * Given a property, inherit the value from the parent dataset, or if received
 * is TRUE, revert to the received value, if any.
 */
int
zfs_prop_inherit(zfs_handle_t *zhp, const char *propname, boolean_t received)
{
        zfs_cmd_t zc = { 0 };
        int ret;
        prop_changelist_t *cl;
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char errbuf[1024];
        zfs_prop_t prop;

        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
            "cannot inherit %s for ’%s’"), propname, zhp->zfs_name);

        zc.zc_cookie = received;
        if ((prop = zfs_name_to_prop(propname)) == ZPROP_INVAL) {
                /*
                 * For user properties, the amount of work we have to do is very
                 * small, so just do it here.
                 */
                if (!zfs_prop_user(propname)) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "invalid property"));
                        return (zfs_error(hdl, EZFS_BADPROP, errbuf));
                }

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

                if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_INHERIT_PROP, &zc) != 0)
                        return (zfs_standard_error(hdl, errno, errbuf));

                return (0);
        }

        /*
         * Verify that this property is inheritable.
         */
        if (zfs_prop_readonly(prop))
                return (zfs_error(hdl, EZFS_PROPREADONLY, errbuf));

        if (!zfs_prop_inheritable(prop) && !received)
                return (zfs_error(hdl, EZFS_PROPNONINHERIT, errbuf));

        /*
         * Check to see if the value applies to this type
         */
        if (!zfs_prop_valid_for_type(prop, zhp->zfs_type))
                return (zfs_error(hdl, EZFS_PROPTYPE, errbuf));
        /*
         * Normalize the name, to get rid of shorthand abbreviations.
         */
        propname = zfs_prop_to_name(prop);
        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
        (void) strlcpy(zc.zc_value, propname, sizeof (zc.zc_value));

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

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

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

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

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

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

                /*
                 * Refresh the statistics so the new property is reflected.
                 */
                (void) get_stats(zhp);
        }

error:
        changelist_free(cl);
        return (ret);
}

/*
 * True DSL properties are stored in an nvlist.  The following two functions
 * extract them appropriately.
 */
static uint64_t
getprop_uint64(zfs_handle_t *zhp, zfs_prop_t prop, char **source)
{
        nvlist_t *nv;
        uint64_t value;

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

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

        return (value);
}

static char *
getprop_string(zfs_handle_t *zhp, zfs_prop_t prop, char **source)
{
        nvlist_t *nv;
        char *value;

        *source = NULL;
        if (nvlist_lookup_nvlist(zhp->zfs_props,
            zfs_prop_to_name(prop), &nv) == 0) {
                verify(nvlist_lookup_string(nv, ZPROP_VALUE, &value) == 0);
                (void) nvlist_lookup_string(nv, ZPROP_SOURCE, source);
        } else {
                verify(!zhp->zfs_props_table ||
                    zhp->zfs_props_table[prop] == B_TRUE);
                if ((value = (char *)zfs_prop_default_string(prop)) == NULL)
                        value = "";
                *source = "";
        }

        return (value);
}

static boolean_t
zfs_is_recvd_props_mode(zfs_handle_t *zhp)
{
        return (zhp->zfs_props == zhp->zfs_recvd_props);
}

static void
zfs_set_recvd_props_mode(zfs_handle_t *zhp, uint64_t *cookie)
{
        *cookie = (uint64_t)(uintptr_t)zhp->zfs_props;
        zhp->zfs_props = zhp->zfs_recvd_props;
}

static void
zfs_unset_recvd_props_mode(zfs_handle_t *zhp, uint64_t *cookie)
{
        zhp->zfs_props = (nvlist_t *)(uintptr_t)*cookie;
        *cookie = 0;
}

/*
 * Internal function for getting a numeric property.  Both zfs_prop_get() and
 * zfs_prop_get_int() are built using this interface.
 *
 * Certain properties can be overridden using 'mount -o'.  In this case, scan
 * the contents of the /etc/mnttab entry, searching for the appropriate options.
 * If they differ from the on-disk values, report the current values and mark
 * the source "temporary".
 */
static int
get_numeric_property(zfs_handle_t *zhp, zfs_prop_t prop, zprop_source_t *src,
    char **source, uint64_t *val)
{
        zfs_cmd_t zc = { 0 };
        nvlist_t *zplprops = NULL;
        struct mnttab mnt;
        char *mntopt_on = NULL;
        char *mntopt_off = NULL;
        boolean_t received = zfs_is_recvd_props_mode(zhp);

        *source = NULL;

        switch (prop) {
        case ZFS_PROP_ATIME:
                mntopt_on = MNTOPT_ATIME;
                mntopt_off = MNTOPT_NOATIME;
                break;

        case ZFS_PROP_DEVICES:
                mntopt_on = MNTOPT_DEVICES;
                mntopt_off = MNTOPT_NODEVICES;
                break;

        case ZFS_PROP_EXEC:
                mntopt_on = MNTOPT_EXEC;
                mntopt_off = MNTOPT_NOEXEC;
                break;

        case ZFS_PROP_READONLY:
                mntopt_on = MNTOPT_RO;
                mntopt_off = MNTOPT_RW;
                break;

        case ZFS_PROP_SETUID:
                mntopt_on = MNTOPT_SETUID;
                mntopt_off = MNTOPT_NOSETUID;
                break;

        case ZFS_PROP_XATTR:
                mntopt_on = MNTOPT_XATTR;
                mntopt_off = MNTOPT_NOXATTR;
                break;

        case ZFS_PROP_NBMAND:
                mntopt_on = MNTOPT_NBMAND;
                mntopt_off = MNTOPT_NONBMAND;
                break;
        }

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

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

                zhp->zfs_mntcheck = B_TRUE;
        }

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

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

        switch (prop) {
        case ZFS_PROP_ATIME:
        case ZFS_PROP_DEVICES:
        case ZFS_PROP_EXEC:
        case ZFS_PROP_READONLY:
        case ZFS_PROP_SETUID:
        case ZFS_PROP_XATTR:
        case ZFS_PROP_NBMAND:
                *val = getprop_uint64(zhp, prop, source);

                if (received)
                        break;

                if (hasmntopt(&mnt, mntopt_on) && !*val) {
                        *val = B_TRUE;
                        if (src)
                                *src = ZPROP_SRC_TEMPORARY;
                } else if (hasmntopt(&mnt, mntopt_off) && *val) {
                        *val = B_FALSE;
                        if (src)
                                *src = ZPROP_SRC_TEMPORARY;
                }
                break;

        case ZFS_PROP_CANMOUNT:
        case ZFS_PROP_VOLSIZE:
        case ZFS_PROP_QUOTA:
        case ZFS_PROP_REFQUOTA:
        case ZFS_PROP_RESERVATION:
        case ZFS_PROP_REFRESERVATION:
                *val = getprop_uint64(zhp, prop, source);

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

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

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

        case ZFS_PROP_VERSION:
        case ZFS_PROP_NORMALIZE:
        case ZFS_PROP_UTF8ONLY:
        case ZFS_PROP_CASE:
                if (!zfs_prop_valid_for_type(prop, zhp->zfs_head_type) ||
                    zcmd_alloc_dst_nvlist(zhp->zfs_hdl, &zc, 0) != 0)
                        return (-1);
                (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
                if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_OBJSET_ZPLPROPS, &zc)) {
                        zcmd_free_nvlists(&zc);
                        return (-1);
                }
                if (zcmd_read_dst_nvlist(zhp->zfs_hdl, &zc, &zplprops) != 0 ||
                    nvlist_lookup_uint64(zplprops, zfs_prop_to_name(prop),
                    val) != 0) {
                        zcmd_free_nvlists(&zc);
                        return (-1);
                }
                if (zplprops)
                        nvlist_free(zplprops);
                zcmd_free_nvlists(&zc);
                break;

        default:
                switch (zfs_prop_get_type(prop)) {
                case PROP_TYPE_NUMBER:
                case PROP_TYPE_INDEX:
                        *val = getprop_uint64(zhp, prop, source);
                        /*
                         * If we tried to use a default value for a
                         * readonly property, it means that it was not
                         * present.
                         */
                        if (zfs_prop_readonly(prop) &&
                            *source != NULL && (*source)[0] == '\0') {
                                *source = NULL;
                        }
                        break;

                case PROP_TYPE_STRING:
                default:
                        zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
                            "cannot get non-numeric property"));
                        return (zfs_error(zhp->zfs_hdl, EZFS_BADPROP,
                            dgettext(TEXT_DOMAIN, "internal error")));
                }
        }

        return (0);
}

/*
 * Calculate the source type, given the raw source string.
 */
static void
get_source(zfs_handle_t *zhp, zprop_source_t *srctype, char *source,
    char *statbuf, size_t statlen)
{
        if (statbuf == NULL || *srctype == ZPROP_SRC_TEMPORARY)
                return;

        if (source == NULL) {
                *srctype = ZPROP_SRC_NONE;
        } else if (source[0] == '\0') {
                *srctype = ZPROP_SRC_DEFAULT;
        } else if (strstr(source, ZPROP_SOURCE_VAL_RECVD) != NULL) {
                *srctype = ZPROP_SRC_RECEIVED;
        } else {
                if (strcmp(source, zhp->zfs_name) == 0) {
                        *srctype = ZPROP_SRC_LOCAL;
                } else {
                        (void) strlcpy(statbuf, source, statlen);
                        *srctype = ZPROP_SRC_INHERITED;
                }
        }

}

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

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

        prop = zfs_name_to_prop(propname);

        if (prop != ZPROP_INVAL) {
                uint64_t cookie;
                if (!nvlist_exists(zhp->zfs_recvd_props, propname))
                        return (-1);
                zfs_set_recvd_props_mode(zhp, &cookie);
                err = zfs_prop_get(zhp, prop, propbuf, proplen,
                    NULL, NULL, 0, literal);
                zfs_unset_recvd_props_mode(zhp, &cookie);
        } else {
                nvlist_t *propval;
                char *recvdval;
                if (nvlist_lookup_nvlist(zhp->zfs_recvd_props,
                    propname, &propval) != 0)
                        return (-1);
                verify(nvlist_lookup_string(propval, ZPROP_VALUE,
                    &recvdval) == 0);
                (void) strlcpy(propbuf, recvdval, proplen);
        }

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

static int
get_clones_string(zfs_handle_t *zhp, char *propbuf, size_t proplen)
{
        nvlist_t *value;
        nvpair_t *pair;

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

        propbuf[0] = '\0';
        for (pair = nvlist_next_nvpair(value, NULL); pair != NULL;
            pair = nvlist_next_nvpair(value, pair)) {
                if (propbuf[0] != '\0')
                        (void) strlcat(propbuf, ",", proplen);
                (void) strlcat(propbuf, nvpair_name(pair), proplen);
        }

        return (0);
}

struct get_clones_arg {
        uint64_t numclones;
        nvlist_t *value;
        const char *origin;
        char buf[ZFS_MAXNAMELEN];
};

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

        if (gca->numclones == 0) {
                zfs_close(zhp);
                return (0);
        }
        if (zfs_prop_get(zhp, ZFS_PROP_ORIGIN, gca->buf, sizeof (gca->buf),
            NULL, NULL, 0, B_TRUE) != 0)
                goto out;
        if (strcmp(gca->buf, gca->origin) == 0) {
                fnvlist_add_boolean(gca->value, zfs_get_name(zhp));
                gca->numclones--;
        }

out:
        (void) zfs_iter_children(zhp, get_clones_cb, gca);
        zfs_close(zhp);
        return (0);
}

nvlist_t *
zfs_get_clones_nvl(zfs_handle_t *zhp)
{
        nvlist_t *nv, *value;

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

                /*
                 * if this is a snapshot, then the kernel wasn't able
                 * to get the clones.  Do it by slowly iterating.
                 */
                if (zhp->zfs_type != ZFS_TYPE_SNAPSHOT)
                        return (NULL);
                if (nvlist_alloc(&nv, NV_UNIQUE_NAME, 0) != 0)
                        return (NULL);
                if (nvlist_alloc(&value, NV_UNIQUE_NAME, 0) != 0) {
                        nvlist_free(nv);
                        return (NULL);
                }

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

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

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

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

                if (gca.numclones != 0 ||
                    nvlist_add_nvlist(nv, ZPROP_VALUE, value) != 0 ||
                    nvlist_add_nvlist(zhp->zfs_props,
                    zfs_prop_to_name(ZFS_PROP_CLONES), nv) != 0) {
                        nvlist_free(nv);
                        nvlist_free(value);
                        return (NULL);
                }
                nvlist_free(nv);
                nvlist_free(value);
                verify(0 == nvlist_lookup_nvlist(zhp->zfs_props,
```

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

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

        return (value);
}

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

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

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

        if (src)
                *src = ZPROP_SRC_NONE;

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

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

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

                str = getprop_string(zhp, prop, &source);

                if (str[0] == '/') {
                        char buf[MAXPATHLEN];
                        char *root = buf;
                        const char *relpath;

                        /*
                         * If we inherit the mountpoint, even from a dataset
                         * with a received value, the source will be the path of
                         * the dataset we inherit from. If source is
                         * ZPROP_SOURCE_VAL_RECVD, the received value is not
                         * inherited.
                         */
                        if (strcmp(source, ZPROP_SOURCE_VAL_RECVD) == 0) {
                                relpath = "";
                        } else {
                                relpath = zhp->zfs_name + strlen(source);
                                if (relpath[0] == '/')
                                        relpath++;
                        }

                        if ((zpool_get_prop(zhp->zpool_hdl,
                            ZPOOL_PROP_ALTROOT, buf, MAXPATHLEN, NULL)) ||
                            (strcmp(root, "-") == 0))
                                root[0] = '\0';
                        /*
                         * Special case an alternate root of '/'. This will
                         * avoid having multiple leading slashes in the
                         * mountpoint path.
                         */
                        if (strcmp(root, "/") == 0)
                                root++;

                        /*
                         * If the mountpoint is '/' then skip over this
                         * if we are obtaining either an alternate root or
                         * an inherited mountpoint.
                         */
                        if (str[1] == '\0' && (root[0] != '\0' ||
                            relpath[0] != '\0'))
                                str++;

                        if (relpath[0] == '\0')
                                (void) snprintf(propbuf, proplen, "%s%s",
                                    root, str);
                        else
                                (void) snprintf(propbuf, proplen, "%s%s%s%s",
                                    root, str, relpath[0] == '@' ? "" : "/",
                                    relpath);
                } else {
                        /* 'legacy' or 'none' */
                        (void) strlcpy(propbuf, str, proplen);
                }

                break;

        case ZFS_PROP_ORIGIN:
                (void) strlcpy(propbuf, getprop_string(zhp, prop, &source),
                    proplen);
                /*
                 * If there is no parent at all, return failure to indicate that
                 * it doesn't apply to this dataset.
                 */
```

```
                if (propbuf[0] == '\0')
                        return (-1);
                break;

        case ZFS_PROP_CLONES:
                if (get_clones_string(zhp, propbuf, proplen) != 0)
                        return (-1);
                break;

        case ZFS_PROP_QUOTA:
        case ZFS_PROP_REFQUOTA:
        case ZFS_PROP_RESERVATION:
        case ZFS_PROP_REFRESERVATION:

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

                /*
                 * If quota or reservation is 0, we translate this into 'none'
                 * (unless literal is set), and indicate that it's the default
                 * value.  Otherwise, we print the number nicely and indicate
                 * that its set locally.
                 */
                if (val == 0) {
                        if (literal)
                                (void) strlcpy(propbuf, "0", proplen);
                        else
                                (void) strlcpy(propbuf, "none", proplen);
                } else {
                        if (literal)
                                (void) snprintf(propbuf, proplen, "%llu",
                                    (u_longlong_t)val);
                        else
                                zfs_nicenum(val, propbuf, proplen);
                }
                break;

        case ZFS_PROP_REFRATIO:
        case ZFS_PROP_COMPRESSRATIO:
                if (get_numeric_property(zhp, prop, src, &source, &val) != 0)
                        return (-1);
                (void) snprintf(propbuf, proplen, "%llu.%02llux",
                    (u_longlong_t)(val / 100),
                    (u_longlong_t)(val % 100));
                break;

        case ZFS_PROP_TYPE:
                switch (zhp->zfs_type) {
                case ZFS_TYPE_FILESYSTEM:
                        str = "filesystem";
                        break;
                case ZFS_TYPE_VOLUME:
                        str = "volume";
                        break;
                case ZFS_TYPE_SNAPSHOT:
                        str = "snapshot";
                        break;
                default:
                        abort();
                }
                (void) snprintf(propbuf, proplen, "%s", str);
                break;

        case ZFS_PROP_MOUNTED:
                /*
                 * The 'mounted' property is a pseudo-property that described
                 * whether the filesystem is currently mounted.  Even though
                 * it's a boolean value, the typical values of "on" and "off"
                 * don't make sense, so we translate to "yes" and "no".
                 */
                if (get_numeric_property(zhp, ZFS_PROP_MOUNTED,
                    src, &source, &val) != 0)
                        return (-1);
                if (val)
                        (void) strlcpy(propbuf, "yes", proplen);
                else
                        (void) strlcpy(propbuf, "no", proplen);
                break;

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

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

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

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

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

                        if (str_to_label(propbuf, &new_sl, MAC_LABEL,
                            L_NO_CORRECTION, NULL) == -1) {
                                m_label_free(new_sl);
                                break;
                        }

                        if (label_to_str(new_sl, &ascii, M_LABEL,
                            DEF_NAMES) != 0) {
                                if (ascii)
                                        free(ascii);
                                m_label_free(new_sl);
                                break;
                        }
                        m_label_free(new_sl);

                        (void) strlcpy(propbuf, ascii, proplen);
                        free(ascii);
                }
                break;

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

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

        default:
                switch (zfs_prop_get_type(prop)) {
                case PROP_TYPE_NUMBER:
                        if (get_numeric_property(zhp, prop, src,
                            &source, &val) != 0)
                                return (-1);
                        if (literal)
                                (void) snprintf(propbuf, proplen, "%llu",
                                    (u_longlong_t)val);
                        else
                                zfs_nicenum(val, propbuf, proplen);
                        break;

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

                case PROP_TYPE_INDEX:
                        if (get_numeric_property(zhp, prop, src,
                            &source, &val) != 0)
                                return (-1);
                        if (zfs_prop_index_to_string(prop, val, &strval) != 0)
                                return (-1);
                        (void) strlcpy(propbuf, strval, proplen);
                        break;

                default:
                        abort();
                }
        }

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

        return (0);
}

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

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

        return (val);
}

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

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

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

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

        if (src)
                *src = ZPROP_SRC_NONE;

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

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

        return (0);
}

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

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

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

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

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

        domain[0] = ’\0’;

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

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

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

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

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

                errno = 0;
                *ridp = strtoull(cp, &end, 10);
                if (numericsid) {
                        free(numericsid);
                        numericsid = NULL;
                }
                if (errno != 0 || *end != ’\0’)
                        return (EINVAL);
        } else if (!isdigit(*cp)) {
                /*
                 * It’s a user/group name (eg "user") that needs to be
                 * turned into a uid/gid
                 */
                if (zoned && getzoneid() == GLOBAL_ZONEID)
                        return (ENOENT);
                if (isuser) {
                        struct passwd *pw;
                        pw = getpwnam(cp);
                        if (pw == NULL)
                                return (ENOENT);
                        *ridp = pw->pw_uid;
                } else {
                        struct group *gr;
                        gr = getgrnam(cp);
                        if (gr == NULL)
                                return (ENOENT);
                        *ridp = gr->gr_gid;
                }
        } else {
                /* It’s a user/group ID (eg "12345"). */
                uid_t id = strtoul(cp, &end, 10);
                idmap_rid_t rid;
                char *mapdomain;

                if (*end != ’\0’)
                        return (EINVAL);
                if (id > MAXUID) {
                        /* It’s an ephemeral ID. */
                        if (idmap_id_to_numeric_domain_rid(id, isuser,
                            &mapdomain, &rid) != 0)
                                return (ENOENT);
                        (void) strlcpy(domain, mapdomain, domainlen);
                        *ridp = rid;
                } else {
                        *ridp = id;
                }
        }

        ASSERT3P(numericsid, ==, NULL);
        return (0);
}

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

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

        err = userquota_propname_decode(propname,
            zfs_prop_get_int(zhp, ZFS_PROP_ZONED),
            typep, zc.zc_value, sizeof (zc.zc_value), &zc.zc_guid);
        zc.zc_objset_type = *typep;
        if (err)
                return (err);

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

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

```

int
zfs_prop_get_userquota_int(zfs_handle_t *zhp, const char *propname,
    uint64_t *propvalue)
{
        zfs_userquota_prop_t type;

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

int
zfs_prop_get_userquota(zfs_handle_t *zhp, const char *propname,
    char *propbuf, int proplen, boolean_t literal)
{
        int err;
        uint64_t propvalue;
        zfs_userquota_prop_t type;

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

        if (err)
                return (err);

        if (literal) {
                (void) snprintf(propbuf, proplen, "%llu", propvalue);
        } else if (propvalue == 0 &&
            (type == ZFS_PROP_USERQUOTA || type == ZFS_PROP_GROUPQUOTA)) {
                (void) strlcpy(propbuf, "none", proplen);
        } else {
                zfs_nicenum(propvalue, propbuf, proplen);
        }
        return (0);
}

int
zfs_prop_get_written_int(zfs_handle_t *zhp, const char *propname,
    uint64_t *propvalue)
{
        int err;
        zfs_cmd_t zc = { 0 };
        const char *snapname;

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

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

                (void) strlcpy(zc.zc_value, zhp->zfs_name,
                    sizeof (zc.zc_value));
                cp = strchr(zc.zc_value, ’@’);
                if (cp != NULL)
                        *cp = ’\0’;
                (void) strlcat(zc.zc_value, "@", sizeof (zc.zc_value));
                (void) strlcat(zc.zc_value, snapname, sizeof (zc.zc_value));
        }

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

```c
        *propvalue = zc.zc_cookie;
        return (0);
}

int
zfs_prop_get_written(zfs_handle_t *zhp, const char *propname,
    char *propbuf, int proplen, boolean_t literal)
{
        int err;
        uint64_t propvalue;

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

        if (err)
                return (err);

        if (literal) {
                (void) snprintf(propbuf, proplen, "%llu", propvalue);
        } else {
                zfs_nicenum(propvalue, propbuf, proplen);
        }
        return (0);
}

/*
 * Returns the name of the given zfs handle.
 */
const char *
zfs_get_name(const zfs_handle_t *zhp)
{
        return (zhp->zfs_name);
}

/*
 * Returns the type of the given zfs handle.
 */
zfs_type_t
zfs_get_type(const zfs_handle_t *zhp)
{
        return (zhp->zfs_type);
}

/*
 * Is one dataset name a child dataset of another?
 *
 * Needs to handle these cases:
 * Dataset 1    "a/foo"         "a/foo"         "a/foo"         "a/foo"
 * Dataset 2    "a/fo"          "a/foobar"      "a/bar/baz"     "a/foo/bar"
 * Descendant?  No.             No.             No.             Yes.
 */
static boolean_t
is_descendant(const char *ds1, const char *ds2)
{
        size_t d1len = strlen(ds1);

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

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

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

```c
 * pool).
 */
static int
parent_name(const char *path, char *buf, size_t buflen)
{
        char *slashp;

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

        if ((slashp = strrchr(buf, ’/’)) == NULL)
                return (-1);
        *slashp = ’\0’;

        return (0);
}

/*
 * If accept_ancestor is false, then check to make sure that the given path has
 * a parent, and that it exists.  If accept_ancestor is true, then find the
 * closest existing ancestor for the given path.  In prefixlen return the
 * length of already existing prefix of the given path.  We also fetch the
 * ’zoned’ property, which is used to validate property settings when creating
 * new datasets.
 */
static int
check_parents(libzfs_handle_t *hdl, const char *path, uint64_t *zoned,
    boolean_t accept_ancestor, int *prefixlen)
{
        zfs_cmd_t zc = { 0 };
        char parent[ZFS_MAXNAMELEN];
        char *slash;
        zfs_handle_t *zhp;
        char errbuf[1024];
        uint64_t is_zoned;

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

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

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

        /* check to see if the parent dataset exists */
        while ((zhp = make_dataset_handle(hdl, parent)) == NULL) {
                if (errno == ENOENT && accept_ancestor) {
                        /*
                         * Go deeper to find an ancestor, give up on top level.
                         */
                        if (parent_name(parent, parent, sizeof (parent)) != 0) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "no such pool ’%s’"), zc.zc_name);
                                return (zfs_error(hdl, EZFS_NOENT, errbuf));
                        }
                } else if (errno == ENOENT) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "parent does not exist"));
                        return (zfs_error(hdl, EZFS_NOENT, errbuf));
                } else
                        return (zfs_standard_error(hdl, errno, errbuf));
        }

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

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

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

        zfs_close(zhp);
        if (prefixlen != NULL)
                *prefixlen = strlen(parent);
        return (0);
}

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

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

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

                zfs_close(zhp);
                if (types & ds_type)
                        return (B_TRUE);
        }
        return (B_FALSE);
}

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

```
create_parents(libzfs_handle_t *hdl, char *target, int prefixlen)
{
        zfs_handle_t *h;
        char *cp;
        const char *opname;

        /* make sure prefix exists */
        cp = target + prefixlen;
        if (*cp != '/') {
                assert(strchr(cp, '/') == NULL);
                h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
        } else {
                *cp = '\0';
                h = zfs_open(hdl, target, ZFS_TYPE_FILESYSTEM);
                *cp = '/';
        }
        if (h == NULL)
                return (-1);
        zfs_close(h);

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

                *cp = '\0';

                h = make_dataset_handle(hdl, target);
                if (h) {
                        /* it already exists, nothing to do here */
                        zfs_close(h);
                        continue;
                }

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

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

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

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

                zfs_close(h);
        }

        return (0);

ancestorerr:
        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
            "failed to %s ancestor '%s'"), opname, target);
        return (-1);
}

/*
 * Creates non-existing ancestors of the given path.
 */
int
zfs_create_ancestors(libzfs_handle_t *hdl, const char *path)
{
        int prefix;
        char *path_copy;
        int rc;

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

        if ((path_copy = strdup(path)) != NULL) {
                rc = create_parents(hdl, path_copy, prefix);
                free(path_copy);
        }
        if (path_copy == NULL || rc != 0)
                return (-1);

        return (0);
}

/*
 * Create a new filesystem or volume.
 */
int
zfs_create(libzfs_handle_t *hdl, const char *path, zfs_type_t type,
    nvlist_t *props)
{
        int ret;
        uint64_t size = 0;
        uint64_t blocksize = zfs_prop_default_numeric(ZFS_PROP_VOLBLOCKSIZE);
        char errbuf[1024];
        uint64_t zoned;
        dmu_objset_type_t ost;

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

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

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

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

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

```
        else
                ost = DMU_OST_ZFS;

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

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

                if ((ret = nvlist_lookup_uint64(props,
                    zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
                    &blocksize)) != 0) {
                        if (ret == ENOENT) {
                                blocksize = zfs_prop_default_numeric(
                                    ZFS_PROP_VOLBLOCKSIZE);
                        } else {
                                nvlist_free(props);
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "missing volume block size"));
                                return (zfs_error(hdl, EZFS_BADPROP, errbuf));
                        }
                }

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

                if (size % blocksize != 0) {
                        nvlist_free(props);
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "volume size must be a multiple of volume block "
                            "size"));
                        return (zfs_error(hdl, EZFS_BADPROP, errbuf));
                }
        }

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

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

                switch (errno) {
                case ENOENT:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "no such parent ’%s’"), parent);
                        return (zfs_error(hdl, EZFS_NOENT, errbuf));

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

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

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

                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded to set this "
                            "property or value"));
                        return (zfs_error(hdl, EZFS_BADVERSION, errbuf));
#ifdef _ILP32
                case EOVERFLOW:
                        /*
                         * This platform can't address a volume this big.
                         */
                        if (type == ZFS_TYPE_VOLUME)
                                return (zfs_error(hdl, EZFS_VOLTOOBIG,
                                    errbuf));
#endif
                        /* FALLTHROUGH */
                default:
                        return (zfs_standard_error(hdl, errno, errbuf));
                }
        }

        return (0);
}

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

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

        if (ZFS_IS_VOLUME(zhp)) {
                zc.zc_objset_type = DMU_OST_ZVOL;
        } else {
                zc.zc_objset_type = DMU_OST_ZFS;
        }

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

        remove_mountpoint(zhp);
```

```
        return (0);
}

struct destroydata {
        nvlist_t *nvl;
        const char *snapname;
};

static int
zfs_check_snap_cb(zfs_handle_t *zhp, void *arg)
{
        struct destroydata *dd = arg;
        zfs_handle_t *szhp;
        char name[ZFS_MAXNAMELEN];
        int rv = 0;

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

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

        rv = zfs_iter_filesystems(zhp, zfs_check_snap_cb, dd);
        zfs_close(zhp);
        return (rv);
}

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

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

        if (nvlist_next_nvpair(dd.nvl, NULL) == NULL) {
                ret = zfs_standard_error_fmt(zhp->zfs_hdl, ENOENT,
                    dgettext(TEXT_DOMAIN, "cannot destroy '%s@%s'"),
                    zhp->zfs_name, snapname);
        } else {
                ret = zfs_destroy_snaps_nvl(zhp->zfs_hdl, dd.nvl, defer);
        }
        nvlist_free(dd.nvl);
        return (ret);
}

/*
 * Destroys all the snapshots named in the nvlist.
 */
int
zfs_destroy_snaps_nvl(libzfs_handle_t *hdl, nvlist_t *snaps, boolean_t defer)
{
        int ret;
        nvlist_t *errlist;

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

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

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

                switch (fnvpair_value_int32(pair)) {
                case EEXIST:
                        zfs_error_aux(hdl,
                            dgettext(TEXT_DOMAIN, "snapshot is cloned"));
                        ret = zfs_error(hdl, EZFS_EXISTS, errbuf);
                        break;
                default:
                        ret = zfs_standard_error(hdl, errno, errbuf);
                        break;
                }
        }

        return (ret);
}

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

        assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);

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

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

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

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

        /* do the clone */

        if (props) {
                zfs_type_t type;
                if (ZFS_IS_VOLUME(zhp)) {
                        type = ZFS_TYPE_VOLUME;
                } else {
```

```
                        type = ZFS_TYPE_FILESYSTEM;
                }
                if ((props = zfs_valid_proplist(hdl, type, props, zoned,
                    zhp, errbuf)) == NULL)
                        return (-1);
        }

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

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

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

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

                default:
                        return (zfs_standard_error(zhp->zfs_hdl, errno,
                            errbuf));
                }
        }

        return (ret);
}

/*
 * Promotes the given clone fs to be the clone parent.
 */
int
zfs_promote(zfs_handle_t *zhp)
{
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        zfs_cmd_t zc = { 0 };
        char parent[MAXPATHLEN];
        int ret;
        char errbuf[1024];

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

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

        (void) strlcpy(parent, zhp->zfs_dmustats.dds_origin, sizeof (parent));
        if (parent[0] == ’\0’) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                    "not a cloned filesystem"));
                return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
        }

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

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

                switch (save_errno) {
                case EEXIST:
                        /* There is a conflicting snapshot name. */
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "conflicting snapshot ’%s’ from parent ’%s’"),
                            zc.zc_string, parent);
                        return (zfs_error(hdl, EZFS_EXISTS, errbuf));

                default:
                        return (zfs_standard_error(hdl, save_errno, errbuf));
                }
        }
        return (ret);
}

typedef struct snapdata {
        nvlist_t *sd_nvl;
        const char *sd_snapname;
} snapdata_t;

static int
zfs_snapshot_cb(zfs_handle_t *zhp, void *arg)
{
        snapdata_t *sd = arg;
        char name[ZFS_MAXNAMELEN];
        int rv = 0;

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

        fnvlist_add_boolean(sd->sd_nvl, name);

        rv = zfs_iter_filesystems(zhp, zfs_snapshot_cb, sd);
        zfs_close(zhp);
        return (rv);
}

/*
 * Creates snapshots.  The keys in the snaps nvlist are the snapshots to be
 * created.
 */
int
zfs_snapshot_nvl(libzfs_handle_t *hdl, nvlist_t *snaps, nvlist_t *props)
{
        int ret;
        char errbuf[1024];
        nvpair_t *elem;
        nvlist_t *errors;

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

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

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

                /* validate the target name */
                if (!zfs_validate_name(hdl, snapname, ZFS_TYPE_SNAPSHOT,
                    B_TRUE)) {
                        (void) snprintf(errbuf, sizeof (errbuf),
                            dgettext(TEXT_DOMAIN,
                            "cannot create snapshot ’%s’"), snapname);
                        return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
                }
        }

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

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

        if (ret != 0) {
                boolean_t printed = B_FALSE;
                for (elem = nvlist_next_nvpair(errors, NULL);
                    elem != NULL;
                    elem = nvlist_next_nvpair(errors, elem)) {
                        (void) snprintf(errbuf, sizeof (errbuf),
                            dgettext(TEXT_DOMAIN,
                            "cannot create snapshot ’%s’"), nvpair_name(elem));
                        (void) zfs_standard_error(hdl,
                            fnvpair_value_int32(elem), errbuf);
                        printed = B_TRUE;
                }
                if (!printed) {
                        switch (ret) {
                        case EXDEV:
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "multiple snapshots of same "
                                    "fs not allowed"));
                                (void) zfs_error(hdl, EZFS_EXISTS, errbuf);

                                break;
                        default:
                                (void) zfs_standard_error(hdl, ret, errbuf);
                        }
                }
        }

        nvlist_free(props);
        nvlist_free(errors);
        return (ret);
}

int
zfs_snapshot(libzfs_handle_t *hdl, const char *path, boolean_t recursive,
    nvlist_t *props)
{
        int ret;
        snapdata_t sd = { 0 };
        char fsname[ZFS_MAXNAMELEN];
        char *cp;
        zfs_handle_t *zhp;
        char errbuf[1024];

        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
            "cannot snapshot %s"), path);
        if (!zfs_validate_name(hdl, path, ZFS_TYPE_SNAPSHOT, B_TRUE))
                return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));

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

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

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

        ret = zfs_snapshot_nvl(hdl, sd.sd_nvl, props);
        nvlist_free(sd.sd_nvl);
        zfs_close(zhp);
        return (ret);
}

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

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

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

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

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

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

```
                        zfs_close(zhp);
                        return (0);
                }
                if (zfs_destroy(zhp, B_FALSE) != 0)
                        cbp->cb_error = B_TRUE;
                else
                        changelist_remove(clp, zhp->zfs_name);
                (void) changelist_postfix(clp);
                changelist_free(clp);
        }

        zfs_close(zhp);
        return (0);
}

/*
 * Given a dataset, rollback to a specific snapshot, discarding any
 * data changes since then and making it the active dataset.
 *
 * Any snapshots more recent than the target are destroyed, along with
 * their dependents.
 */
int
zfs_rollback(zfs_handle_t *zhp, zfs_handle_t *snap, boolean_t force)
{
        rollback_data_t cb = { 0 };
        int err;
        zfs_cmd_t zc = { 0 };
        boolean_t restore_resv = 0;
        uint64_t old_volsize, new_volsize;
        zfs_prop_t resv_prop;

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

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

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

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

        if (zhp->zfs_type == ZFS_TYPE_VOLUME) {
                if (zfs_which_resv_prop(zhp, &resv_prop) < 0)
                        return (-1);
                old_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
                restore_resv =
                    (old_volsize == zfs_prop_get_int(zhp, resv_prop));
        }

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

        if (ZFS_IS_VOLUME(zhp))
                zc.zc_objset_type = DMU_OST_ZVOL;
        else
                zc.zc_objset_type = DMU_OST_ZFS;
        /*
         * We rely on zfs_iter_children() to verify that there are no
         * newer snapshots for the given dataset.  Therefore, we can
         * simply pass the name on to the ioctl() call.  There is still
         * an unlikely race condition where the user has taken a
         * snapshot since we verified that this was the most recent.
         *
         */
        if ((err = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_ROLLBACK, &zc)) != 0) {
                (void) zfs_standard_error_fmt(zhp->zfs_hdl, errno,
                    dgettext(TEXT_DOMAIN, "cannot rollback ’%s’"),
                    zhp->zfs_name);
                return (err);
        }

        /*
         * For volumes, if the pre-rollback volsize matched the pre-
         * rollback reservation and the volsize has changed then set
         * the reservation property to the post-rollback volsize.
         * Make a new handle since the rollback closed the dataset.
         */
        if ((zhp->zfs_type == ZFS_TYPE_VOLUME) &&
            (zhp = make_dataset_handle(zhp->zfs_hdl, zhp->zfs_name))) {
                if (restore_resv) {
                        new_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
                        if (old_volsize != new_volsize)
                                err = zfs_prop_set_int(zhp, resv_prop,
                                    new_volsize);
                }
                zfs_close(zhp);
        }
        return (err);
}

/*
 * Renames the given dataset.
 */
int
zfs_rename(zfs_handle_t *zhp, const char *target, boolean_t recursive,
    boolean_t force_unmount)
{
        int ret;
        zfs_cmd_t zc = { 0 };
        char *delim;
        prop_changelist_t *cl = NULL;
        zfs_handle_t *zhrp = NULL;
        char *parentname = NULL;
        char parent[ZFS_MAXNAMELEN];
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char errbuf[1024];

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

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

        /*
         * Make sure the target name is valid
         */
        if (zhp->zfs_type == ZFS_TYPE_SNAPSHOT) {
                if ((strchr(target, ’@’) == NULL) ||
                    *target == ’@’) {
                        /*
                         * Snapshot target name is abbreviated,
```

```
                         * reconstruct full dataset name
                         */
                        (void) strlcpy(parent, zhp->zfs_name,
                            sizeof (parent));
                        delim = strchr(parent, ’@’);
                        if (strchr(target, ’@’) == NULL)
                                *(++delim) = ’\0’;
                        else
                                *delim = ’\0’;
                        (void) strlcat(parent, target, sizeof (parent));
                        target = parent;
                } else {
                        /*
                         * Make sure we’re renaming within the same dataset.
                         */
                        delim = strchr(target, ’@’);
                        if (strncmp(zhp->zfs_name, target, delim - target)
                            != 0 || zhp->zfs_name[delim - target] != ’@’) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "snapshots must be part of same "
                                    "dataset"));
                                return (zfs_error(hdl, EZFS_CROSSTARGET,
                                    errbuf));
                        }
                }
                if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
                        return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
        } else {
                if (recursive) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "recursive rename must be a snapshot"));
                        return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
                }

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

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

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

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

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

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

        if (recursive) {

                parentname = zfs_strdup(zhp->zfs_hdl, zhp->zfs_name);
                if (parentname == NULL) {
                        ret = -1;
                        goto error;
                }
                delim = strchr(parentname, ’@’);
                *delim = ’\0’;
                zhrp = zfs_open(zhp->zfs_hdl, parentname, ZFS_TYPE_DATASET);
                if (zhrp == NULL) {
                        ret = -1;
                        goto error;
                }

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

                if (changelist_haszonedchild(cl)) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "child dataset with inherited mountpoint is used "
                            "in a non-global zone"));
                        (void) zfs_error(hdl, EZFS_ZONED, errbuf);
                        goto error;
                }

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

        if (ZFS_IS_VOLUME(zhp))
                zc.zc_objset_type = DMU_OST_ZVOL;
        else
                zc.zc_objset_type = DMU_OST_ZFS;

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

        zc.zc_cookie = recursive;

        if ((ret = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_RENAME, &zc)) != 0) {
                /*
                 * if it was recursive, the one that actually failed will
                 * be in zc.zc_name
                 */
                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "cannot rename ’%s’"), zc.zc_name);

                if (recursive && errno == EEXIST) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "a child dataset already has a snapshot "
                            "with the new name"));
                        (void) zfs_error(hdl, EZFS_EXISTS, errbuf);
                } else {
                        (void) zfs_standard_error(zhp->zfs_hdl, errno, errbuf);
                }

                /*
                 * On failure, we still want to remount any filesystems that
                 * were previously mounted, so we don’t alter the system state.
                 */
```

```
                if (!recursive)
                        (void) changelist_postfix(cl);
        } else {
                if (!recursive) {
                        changelist_rename(cl, zfs_get_name(zhp), target);
                        ret = changelist_postfix(cl);
                }
        }

error:
        if (parentname) {
                free(parentname);
        }
        if (zhrp) {
                zfs_close(zhrp);
        }
        if (cl) {
                changelist_free(cl);
        }
        return (ret);
}

nvlist_t *
zfs_get_user_props(zfs_handle_t *zhp)
{
        return (zhp->zfs_user_props);
}

nvlist_t *
zfs_get_recvd_props(zfs_handle_t *zhp)
{
        if (zhp->zfs_recvd_props == NULL)
                if (get_recvd_props_ioctl(zhp) != 0)
                        return (NULL);
        return (zhp->zfs_recvd_props);
}

/*
 * This function is used by ’zfs list’ to determine the exact set of columns to
 * display, and their maximum widths.  This does two main things:
 *
 *      - If this is a list of all properties, then expand the list to include
 *        all native properties, and set a flag so that for each dataset we look
 *        for new unique user properties and add them to the list.
 *
 *      - For non fixed-width properties, keep track of the maximum width seen
 *        so that we can size the column appropriately. If the user has
 *        requested received property values, we also need to compute the width
 *        of the RECEIVED column.
 */
int
zfs_expand_proplist(zfs_handle_t *zhp, zprop_list_t **plp, boolean_t received)
{
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        zprop_list_t *entry;
        zprop_list_t **last, **start;
        nvlist_t *userprops, *propval;
        nvpair_t *elem;
        char *strval;
        char buf[ZFS_MAXPROPLEN];

        if (zprop_expand_list(hdl, plp, ZFS_TYPE_DATASET) != 0)
                return (-1);

        userprops = zfs_get_user_props(zhp);
        entry = *plp;
        if (entry->pl_all && nvlist_next_nvpair(userprops, NULL) != NULL) {
                /*
                 * Go through and add any user properties as necessary.  We
                 * start by incrementing our list pointer to the first
                 * non-native property.
                 */
                start = plp;
                while (*start != NULL) {
                        if ((*start)->pl_prop == ZPROP_INVAL)
                                break;
                        start = &(*start)->pl_next;
                }

                elem = NULL;
                while ((elem = nvlist_next_nvpair(userprops, elem)) != NULL) {
                        /*
                         * See if we’ve already found this property in our list.
                         */
                        for (last = start; *last != NULL;
                            last = &(*last)->pl_next) {
                                if (strcmp((*last)->pl_user_prop,
                                    nvpair_name(elem)) == 0)
                                        break;
                        }

                        if (*last == NULL) {
                                if ((entry = zfs_alloc(hdl,
                                    sizeof (zprop_list_t))) == NULL ||
                                    ((entry->pl_user_prop = zfs_strdup(hdl,
                                    nvpair_name(elem)))) == NULL) {
                                        free(entry);
                                        return (-1);
                                }

                                entry->pl_prop = ZPROP_INVAL;
                                entry->pl_width = strlen(nvpair_name(elem));
                                entry->pl_all = B_TRUE;
                                *last = entry;
                        }
                }
        }

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

                if (entry->pl_prop != ZPROP_INVAL) {
                        if (zfs_prop_get(zhp, entry->pl_prop,
                            buf, sizeof (buf), NULL, NULL, 0, B_FALSE) == 0) {
                                if (strlen(buf) > entry->pl_width)
                                        entry->pl_width = strlen(buf);
                        }
                        if (received && zfs_prop_get_recvd(zhp,
                            zfs_prop_to_name(entry->pl_prop),
                            buf, sizeof (buf), B_FALSE) == 0)
                                if (strlen(buf) > entry->pl_recvd_width)
                                        entry->pl_recvd_width = strlen(buf);
                } else {
                        if (nvlist_lookup_nvlist(userprops, entry->pl_user_prop,
                            &propval) == 0) {
                                verify(nvlist_lookup_string(propval,
                                    ZPROP_VALUE, &strval) == 0);
```

```
                                if (strlen(strval) > entry->pl_width)
                                        entry->pl_width = strlen(strval);
                        }
                        if (received && zfs_prop_get_recvd(zhp,
                            entry->pl_user_prop,
                            buf, sizeof (buf), B_FALSE) == 0)
                                if (strlen(buf) > entry->pl_recvd_width)
                                        entry->pl_recvd_width = strlen(buf);
                }
        }

        return (0);
}

int
zfs_deleg_share_nfs(libzfs_handle_t *hdl, char *dataset, char *path,
    char *resource, void *export, void *sharetab,
    int sharemax, zfs_share_op_t operation)
{
        zfs_cmd_t zc = { 0 };
        int error;

        (void) strlcpy(zc.zc_name, dataset, sizeof (zc.zc_name));
        (void) strlcpy(zc.zc_value, path, sizeof (zc.zc_value));
        if (resource)
                (void) strlcpy(zc.zc_string, resource, sizeof (zc.zc_string));
        zc.zc_share.z_sharedata = (uint64_t)(uintptr_t)sharetab;
        zc.zc_share.z_exportdata = (uint64_t)(uintptr_t)export;
        zc.zc_share.z_sharetype = operation;
        zc.zc_share.z_sharemax = sharemax;
        error = ioctl(hdl->libzfs_fd, ZFS_IOC_SHARE, &zc);
        return (error);
}

void
zfs_prune_proplist(zfs_handle_t *zhp, uint8_t *props)
{
        nvpair_t *curr;

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

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

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

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

static int
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        return (0);
}

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

static int
zfs_hold_one(zfs_handle_t *zhp, void *arg)
{
        struct holdarg *ha = arg;
        zfs_handle_t *szhp;
        char name[ZFS_MAXNAMELEN];
        int rv = 0;

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

        szhp = make_dataset_handle(zhp->zfs_hdl, name);
        if (szhp) {
                fnvlist_add_string(ha->nvl, name, ha->tag);
                zfs_close(szhp);
        }

        if (ha->recursive)
                rv = zfs_iter_filesystems(zhp, zfs_hold_one, ha);
        zfs_close(zhp);
        return (rv);
}

int
zfs_hold(zfs_handle_t *zhp, const char *snapname, const char *tag,
    boolean_t recursive, boolean_t enoent_ok, int cleanup_fd)
{
        int ret;
        struct holdarg ha;
        nvlist_t *errors;
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char errbuf[1024];
        nvpair_t *elem;

        ha.nvl = fnvlist_alloc();
        ha.snapname = snapname;
        ha.tag = tag;
        ha.recursive = recursive;
        (void) zfs_hold_one(zfs_handle_dup(zhp), &ha);

        if (nvlist_next_nvpair(ha.nvl, NULL) == NULL) {
                fnvlist_free(ha.nvl);
                ret = ENOENT;
                if (!enoent_ok) {
                        (void) snprintf(errbuf, sizeof (errbuf),
                            dgettext(TEXT_DOMAIN,
                            "cannot hold snapshot '%s@%s'"),
                            zhp->zfs_name, snapname);
                        (void) zfs_standard_error(hdl, ret, errbuf);
                }
                return (ret);
        }

#endif /* ! codereview */
        ret = lzc_hold(ha.nvl, cleanup_fd, &errors);
        fnvlist_free(ha.nvl);

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

        if (nvlist_next_nvpair(errors, NULL) == NULL) {
                /* no hold-specific errors */
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot hold"));
                switch (ret) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
```

```
                        break;
                case EINVAL:
                        (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                default:
                        (void) zfs_standard_error(hdl, ret, errbuf);
                }
        }

        for (elem = nvlist_next_nvpair(errors, NULL);
            elem != NULL;
            elem = nvlist_next_nvpair(errors, elem)) {
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN,
                    "cannot hold snapshot '%s'"), nvpair_name(elem));
                switch (fnvpair_value_int32(elem)) {
                case E2BIG:
                        /*
                         * Temporary tags wind up having the ds object id
                         * prepended. So even if we passed the length check
                         * above, it's still possible for the tag to wind
                         * up being slightly too long.
                         */
                        (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
                        break;
                case EINVAL:
                        (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case EEXIST:
                        (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
                        break;
                case ENOENT:
                        if (enoent_ok)
                                return (ENOENT);
                        /* FALLTHROUGH */
                default:
                        (void) zfs_standard_error(hdl,
                            fnvpair_value_int32(elem), errbuf);
                }
        }

        fnvlist_free(errors);
        return (ret);
}

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

static int
zfs_release_one(zfs_handle_t *zhp, void *arg)
{
        struct holdarg *ha = arg;
        zfs_handle_t *szhp;
        char name[ZFS_MAXNAMELEN];
        int rv = 0;

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

        szhp = make_dataset_handle(zhp->zfs_hdl, name);
        if (szhp) {
                nvlist_t *holds = fnvlist_alloc();
                fnvlist_add_boolean(holds, ha->tag);
                fnvlist_add_nvlist(ha->nvl, name, holds);
                zfs_close(szhp);
        }

        if (ha->recursive)
                rv = zfs_iter_filesystems(zhp, zfs_release_one, ha);
        zfs_close(zhp);
        return (rv);
}

int
zfs_release(zfs_handle_t *zhp, const char *snapname, const char *tag,
    boolean_t recursive)
{
        int ret;
        struct holdarg ha;
        nvlist_t *errors;
        nvpair_t *elem;
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char errbuf[1024];
#endif /* ! codereview */

        ha.nvl = fnvlist_alloc();
        ha.snapname = snapname;
        ha.tag = tag;
        ha.recursive = recursive;
        (void) zfs_release_one(zfs_handle_dup(zhp), &ha);

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

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

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

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

                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "cannot release"));
                switch (errno) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                default:
                        (void) zfs_standard_error_fmt(hdl, errno, errbuf);
                }
        }

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

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

                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN,
                    "cannot release hold from snapshot '%s'"),
                    nvpair_name(elem));
                switch (fnvpair_value_int32(elem)) {
                case ESRCH:
                        (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
                        break;
                case EINVAL:
                        (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                default:
                        (void) zfs_standard_error_fmt(hdl,
                            fnvpair_value_int32(elem), errbuf);
                }
        }

        fnvlist_free(errors);
        return (ret);
}
______unchanged_portion_omitted_
```

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

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

#include <sys/dmu.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/dsl_deleg.h>
#include <sys/spa.h>
#include <sys/metaslab.h>
#include <sys/zap.h>
#include <sys/zio.h>
#include <sys/arc.h>
#include <sys/sunddi.h>
#include "zfs_namecheck.h"

static uint64_t dsl_dir_space_towrite(dsl_dir_t *dd);

/* ARGSUSED */
static void
dsl_dir_evict(dmu_buf_t *db, void *arg)
{
        dsl_dir_t *dd = arg;
        dsl_pool_t *dp = dd->dd_pool;
        int t;

        for (t = 0; t < TXG_SIZE; t++) {
                ASSERT(!txg_list_member(&dp->dp_dirty_dirs, dd, t));
                ASSERT(dd->dd_tempreserved[t] == 0);
                ASSERT(dd->dd_space_towrite[t] == 0);
        }
```

```
        if (dd->dd_parent)
                dsl_dir_rele(dd->dd_parent, dd);

        spa_close(dd->dd_pool->dp_spa, dd);

        /*
         * The props callback list should have been cleaned up by
         * objset_evict().
         */
        list_destroy(&dd->dd_prop_cbs);
        mutex_destroy(&dd->dd_lock);
        kmem_free(dd, sizeof (dsl_dir_t));
}

int
dsl_dir_hold_obj(dsl_pool_t *dp, uint64_t ddobj,
    const char *tail, void *tag, dsl_dir_t **ddp)
{
        dmu_buf_t *dbuf;
        dsl_dir_t *dd;
        int err;

        ASSERT(dsl_pool_config_held(dp));

        err = dmu_bonus_hold(dp->dp_meta_objset, ddobj, tag, &dbuf);
        if (err != 0)
                return (err);
        dd = dmu_buf_get_user(dbuf);
#ifdef ZFS_DEBUG
        {
                dmu_object_info_t doi;
                dmu_object_info_from_db(dbuf, &doi);
                ASSERT3U(doi.doi_type, ==, DMU_OT_DSL_DIR);
                ASSERT3U(doi.doi_bonus_size, >=, sizeof (dsl_dir_phys_t));
        }
#endif
        if (dd == NULL) {
                dsl_dir_t *winner;

                dd = kmem_zalloc(sizeof (dsl_dir_t), KM_SLEEP);
                dd->dd_object = ddobj;
                dd->dd_dbuf = dbuf;
                dd->dd_pool = dp;
                dd->dd_phys = dbuf->db_data;
                mutex_init(&dd->dd_lock, NULL, MUTEX_DEFAULT, NULL);

                list_create(&dd->dd_prop_cbs, sizeof (dsl_prop_cb_record_t),
                    offsetof(dsl_prop_cb_record_t, cbr_node));

                dsl_dir_snap_cmtime_update(dd);

                if (dd->dd_phys->dd_parent_obj) {
                        err = dsl_dir_hold_obj(dp, dd->dd_phys->dd_parent_obj,
                            NULL, dd, &dd->dd_parent);
                        if (err != 0)
                                goto errout;
                        if (tail) {
#ifdef ZFS_DEBUG
                                uint64_t foundobj;

                                err = zap_lookup(dp->dp_meta_objset,
                                    dd->dd_parent->dd_phys->dd_child_dir_zapobj,
                                    tail, sizeof (foundobj), 1, &foundobj);
                                ASSERT(err || foundobj == ddobj);
#endif
```

```
                                (void) strcpy(dd->dd_myname, tail);
                        } else {
                                err = zap_value_search(dp->dp_meta_objset,
                                    dd->dd_parent->dd_phys->dd_child_dir_zapobj,
                                    ddobj, 0, dd->dd_myname);
                        }
                        if (err != 0)
                                goto errout;
                } else {
                        (void) strcpy(dd->dd_myname, spa_name(dp->dp_spa));
                }

                if (dsl_dir_is_clone(dd)) {
                        dmu_buf_t *origin_bonus;
                        dsl_dataset_phys_t *origin_phys;

                        /*
                         * We can't open the origin dataset, because
                         * that would require opening this dsl_dir.
                         * Just look at its phys directly instead.
                         */
                        err = dmu_bonus_hold(dp->dp_meta_objset,
                            dd->dd_phys->dd_origin_obj, FTAG, &origin_bonus);
                        if (err != 0)
                                goto errout;
                        origin_phys = origin_bonus->db_data;
                        dd->dd_origin_txg =
                            origin_phys->ds_creation_txg;
                        dmu_buf_rele(origin_bonus, FTAG);
                }

                winner = dmu_buf_set_user_ie(dbuf, dd, &dd->dd_phys,
                    dsl_dir_evict);
                if (winner) {
                        if (dd->dd_parent)
                                dsl_dir_rele(dd->dd_parent, dd);
                        mutex_destroy(&dd->dd_lock);
                        kmem_free(dd, sizeof (dsl_dir_t));
                        dd = winner;
                } else {
                        spa_open_ref(dp->dp_spa, dd);
                }
        }

        /*
         * The dsl_dir_t has both open-to-close and instantiate-to-evict
         * holds on the spa.  We need the open-to-close holds because
         * otherwise the spa_refcnt wouldn't change when we open a
         * dir which the spa also has open, so we could incorrectly
         * think it was OK to unload/export/destroy the pool.  We need
         * the instantiate-to-evict hold because the dsl_dir_t has a
         * pointer to the dd_pool, which has a pointer to the spa_t.
         */
        spa_open_ref(dp->dp_spa, tag);
        ASSERT3P(dd->dd_pool, ==, dp);
        ASSERT3U(dd->dd_object, ==, ddobj);
        ASSERT3P(dd->dd_dbuf, ==, dbuf);
        *ddp = dd;
        return (0);

errout:
        if (dd->dd_parent)
                dsl_dir_rele(dd->dd_parent, dd);
        mutex_destroy(&dd->dd_lock);
        kmem_free(dd, sizeof (dsl_dir_t));
        dmu_buf_rele(dbuf, tag);
```

```
        return (err);
}

void
dsl_dir_rele(dsl_dir_t *dd, void *tag)
{
        dprintf_dd(dd, "%s\n", "");
        spa_close(dd->dd_pool->dp_spa, tag);
        dmu_buf_rele(dd->dd_dbuf, tag);
}

/* buf must be long enough (MAXNAMELEN + strlen(MOS_DIR_NAME) + 1 should do) */
void
dsl_dir_name(dsl_dir_t *dd, char *buf)
{
        if (dd->dd_parent) {
                dsl_dir_name(dd->dd_parent, buf);
                (void) strcat(buf, "/");
        } else {
                buf[0] = '\0';
        }
        if (!MUTEX_HELD(&dd->dd_lock)) {
                /*
                 * recursive mutex so that we can use
                 * dprintf_dd() with dd_lock held
                 */
                mutex_enter(&dd->dd_lock);
                (void) strcat(buf, dd->dd_myname);
                mutex_exit(&dd->dd_lock);
        } else {
                (void) strcat(buf, dd->dd_myname);
        }
}

/* Calculate name length, avoiding all the strcat calls of dsl_dir_name */
int
dsl_dir_namelen(dsl_dir_t *dd)
{
        int result = 0;

        if (dd->dd_parent) {
                /* parent's name + 1 for the "/" */
                result = dsl_dir_namelen(dd->dd_parent) + 1;
        }

        if (!MUTEX_HELD(&dd->dd_lock)) {
                /* see dsl_dir_name */
                mutex_enter(&dd->dd_lock);
                result += strlen(dd->dd_myname);
                mutex_exit(&dd->dd_lock);
        } else {
                result += strlen(dd->dd_myname);
        }

        return (result);
}

static int
getcomponent(const char *path, char *component, const char **nextp)
{
        char *p;

        if ((path == NULL) || (path[0] == '\0'))
                return (SET_ERROR(ENOENT));
        /* This would be a good place to reserve some namespace... */
        p = strpbrk(path, "/@");
        if (p && (p[1] == ’/’ || p[1] == ’@’)) {
                /* two separators in a row */
                return (SET_ERROR(EINVAL));
        }
        if (p == NULL || p == path) {
                /*
                 * if the first thing is an @ or /, it had better be an
                 * @ and it had better not have any more ats or slashes,
                 * and it had better have something after the @.
                 */
                if (p != NULL &&
                    (p[0] != ’@’ || strpbrk(path+1, "/@") || p[1] == ’\0’))
                        return (SET_ERROR(EINVAL));
                if (strlen(path) >= MAXNAMELEN)
                        return (SET_ERROR(ENAMETOOLONG));
                (void) strcpy(component, path);
                p = NULL;
        } else if (p[0] == ’/’) {
                if (p - path >= MAXNAMELEN)
                        return (SET_ERROR(ENAMETOOLONG));
                (void) strncpy(component, path, p - path);
                component[p - path] = ’\0’;
                p++;
        } else if (p[0] == ’@’) {
                /*
                 * if the next separator is an @, there better not be
                 * any more slashes.
                 */
                if (strchr(path, ’/’))
                        return (SET_ERROR(EINVAL));
                if (p - path >= MAXNAMELEN)
                        return (SET_ERROR(ENAMETOOLONG));
                (void) strncpy(component, path, p - path);
                component[p - path] = ’\0’;
        } else {
                panic("invalid p=%p", (void *)p);
        }
        *nextp = p;
        return (0);
}

/*
 * Return the dsl_dir_t, and possibly the last component which couldn’t
 * be found in *tail.  The name must be in the specified dsl_pool_t.  This
 * thread must hold the dp_config_rwlock for the pool.  Returns NULL if the
 * path is bogus, or if tail==NULL and we couldn’t parse the whole name.
 * (*tail)[0] == ’@’ means that the last component is a snapshot.
 */
int
dsl_dir_hold(dsl_pool_t *dp, const char *name, void *tag,
    dsl_dir_t **ddp, const char **tailp)
{
        char buf[MAXNAMELEN];
        const char *spaname, *next, *nextnext = NULL;
        int err;
        dsl_dir_t *dd;
        uint64_t ddobj;

        err = getcomponent(name, buf, &next);
        if (err != 0)
                return (err);

        /* Make sure the name is in the specified pool. */
        spaname = spa_name(dp->dp_spa);
        if (strcmp(buf, spaname) != 0)
                return (SET_ERROR(EINVAL));
```

```

        ASSERT(dsl_pool_config_held(dp));

        err = dsl_dir_hold_obj(dp, dp->dp_root_dir_obj, NULL, tag, &dd);
        if (err != 0) {
                return (err);
        }

        while (next != NULL) {
                dsl_dir_t *child_ds;
                err = getcomponent(next, buf, &nextnext);
                if (err != 0)
                        break;
                ASSERT(next[0] != ’\0’);
                if (next[0] == ’@’)
                        break;
                dprintf("looking up %s in obj%lld\n",
                    buf, dd->dd_phys->dd_child_dir_zapobj);

                err = zap_lookup(dp->dp_meta_objset,
                    dd->dd_phys->dd_child_dir_zapobj,
                    buf, sizeof (ddobj), 1, &ddobj);
                if (err != 0) {
                        if (err == ENOENT)
                                err = 0;
                        break;
                }

                err = dsl_dir_hold_obj(dp, ddobj, buf, tag, &child_ds);
                if (err != 0)
                        break;
                dsl_dir_rele(dd, tag);
                dd = child_ds;
                next = nextnext;
        }

        if (err != 0) {
                dsl_dir_rele(dd, tag);
                return (err);
        }

        /*
         * It’s an error if there’s more than one component left, or
         * tailp==NULL and there’s any component left.
         */
        if (next != NULL &&
            (tailp == NULL || (nextnext && nextnext[0] != ’\0’))) {
                /* bad path name */
                dsl_dir_rele(dd, tag);
                dprintf("next=%p (%s) tail=%p\n", next, next?next:"", tailp);
                err = SET_ERROR(ENOENT);
        }
        if (tailp != NULL)
                *tailp = next;
        *ddp = dd;
        return (err);
}

uint64_t
dsl_dir_create_sync(dsl_pool_t *dp, dsl_dir_t *pds, const char *name,
    dmu_tx_t *tx)
{
        objset_t *mos = dp->dp_meta_objset;
        uint64_t ddobj;
        dsl_dir_phys_t *ddphys;
        dmu_buf_t *dbuf;
        ddobj = dmu_object_alloc(mos, DMU_OT_DSL_DIR, 0,
            DMU_OT_DSL_DIR, sizeof (dsl_dir_phys_t), tx);
        if (pds) {
                VERIFY(0 == zap_add(mos, pds->dd_phys->dd_child_dir_zapobj,
                    name, sizeof (uint64_t), 1, &ddobj, tx));
        } else {
                /* it's the root dir */
                VERIFY(0 == zap_add(mos, DMU_POOL_DIRECTORY_OBJECT,
                    DMU_POOL_ROOT_DATASET, sizeof (uint64_t), 1, &ddobj, tx));
        }
        VERIFY(0 == dmu_bonus_hold(mos, ddobj, FTAG, &dbuf));
        dmu_buf_will_dirty(dbuf, tx);
        ddphys = dbuf->db_data;

        ddphys->dd_creation_time = gethrestime_sec();
        if (pds)
                ddphys->dd_parent_obj = pds->dd_object;
        ddphys->dd_props_zapobj = zap_create(mos,
            DMU_OT_DSL_PROPS, DMU_OT_NONE, 0, tx);
        ddphys->dd_child_dir_zapobj = zap_create(mos,
            DMU_OT_DSL_DIR_CHILD_MAP, DMU_OT_NONE, 0, tx);
        if (spa_version(dp->dp_spa) >= SPA_VERSION_USED_BREAKDOWN)
                ddphys->dd_flags |= DD_FLAG_USED_BREAKDOWN;
        dmu_buf_rele(dbuf, FTAG);

        return (ddobj);
}

boolean_t
dsl_dir_is_clone(dsl_dir_t *dd)
{
        return (dd->dd_phys->dd_origin_obj &&
            (dd->dd_pool->dp_origin_snap == NULL ||
            dd->dd_phys->dd_origin_obj !=
            dd->dd_pool->dp_origin_snap->ds_object));
}

void
dsl_dir_stats(dsl_dir_t *dd, nvlist_t *nv)
{
        mutex_enter(&dd->dd_lock);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USED,
            dd->dd_phys->dd_used_bytes);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_QUOTA, dd->dd_phys->dd_quota);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_RESERVATION,
            dd->dd_phys->dd_reserved);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_COMPRESSRATIO,
            dd->dd_phys->dd_compressed_bytes == 0 ? 100 :
            (dd->dd_phys->dd_uncompressed_bytes * 100 /
            dd->dd_phys->dd_compressed_bytes));
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_LOGICALUSED,
            dd->dd_phys->dd_uncompressed_bytes);
        if (dd->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
                dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USEDSNAP,
                    dd->dd_phys->dd_used_breakdown[DD_USED_SNAP]);
                dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USEDDS,
                    dd->dd_phys->dd_used_breakdown[DD_USED_HEAD]);
                dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USEDREFRESERV,
                    dd->dd_phys->dd_used_breakdown[DD_USED_REFRSRV]);
                dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USEDCHILD,
                    dd->dd_phys->dd_used_breakdown[DD_USED_CHILD] +
                    dd->dd_phys->dd_used_breakdown[DD_USED_CHILD_RSRV]);
        }
        mutex_exit(&dd->dd_lock);
```

```
        if (dsl_dir_is_clone(dd)) {
                dsl_dataset_t *ds;
                char buf[MAXNAMELEN];

                VERIFY0(dsl_dataset_hold_obj(dd->dd_pool,
                    dd->dd_phys->dd_origin_obj, FTAG, &ds));
                dsl_dataset_name(ds, buf);
                dsl_dataset_rele(ds, FTAG);
                dsl_prop_nvlist_add_string(nv, ZFS_PROP_ORIGIN, buf);
        }
}

void
dsl_dir_dirty(dsl_dir_t *dd, dmu_tx_t *tx)
{
        dsl_pool_t *dp = dd->dd_pool;

        ASSERT(dd->dd_phys);

        if (txg_list_add(&dp->dp_dirty_dirs, dd, tx->tx_txg)) {
                /* up the hold count until we can be written out */
                dmu_buf_add_ref(dd->dd_dbuf, dd);
        }
}

static int64_t
parent_delta(dsl_dir_t *dd, uint64_t used, int64_t delta)
{
        uint64_t old_accounted = MAX(used, dd->dd_phys->dd_reserved);
        uint64_t new_accounted = MAX(used + delta, dd->dd_phys->dd_reserved);
        return (new_accounted - old_accounted);
}

void
dsl_dir_sync(dsl_dir_t *dd, dmu_tx_t *tx)
{
        ASSERT(dmu_tx_is_syncing(tx));

        mutex_enter(&dd->dd_lock);
        ASSERT0(dd->dd_tempreserved[tx->tx_txg&TXG_MASK]);
        dprintf_dd(dd, "txg=%llu towrite=%lluK\n", tx->tx_txg,
            dd->dd_space_towrite[tx->tx_txg&TXG_MASK] / 1024);
        dd->dd_space_towrite[tx->tx_txg&TXG_MASK] = 0;
        mutex_exit(&dd->dd_lock);

        /* release the hold from dsl_dir_dirty */
        dmu_buf_rele(dd->dd_dbuf, dd);
}

static uint64_t
dsl_dir_space_towrite(dsl_dir_t *dd)
{
        uint64_t space = 0;
        int i;

        ASSERT(MUTEX_HELD(&dd->dd_lock));

        for (i = 0; i < TXG_SIZE; i++) {
                space += dd->dd_space_towrite[i&TXG_MASK];
                ASSERT3U(dd->dd_space_towrite[i&TXG_MASK], >=, 0);
        }
        return (space);
}

/*
 * How much space would dd have available if ancestor had delta applied
 * to it?  If ondiskonly is set, we’re only interested in what’s
 * on-disk, not estimated pending changes.
 */
uint64_t
dsl_dir_space_available(dsl_dir_t *dd,
    dsl_dir_t *ancestor, int64_t delta, int ondiskonly)
{
        uint64_t parentspace, myspace, quota, used;

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

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

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

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

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

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

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

        mutex_exit(&dd->dd_lock);

        return (myspace);
}

struct tempreserve {
```

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

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

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

        mutex_enter(&dd->dd_lock);

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

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

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

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

        /*
         * Adjust the quota against the actual pool size at the root
         * minus any outstanding deferred frees.
         * To ensure that it’s possible to remove files from a full
         * pool without inducing transient overcommits, we throttle
         * netfree transactions against a quota that is slightly larger,
	 * but still within the pool’s allocation slop.  In cases where
	 * we’re very close to full, this will allow a steady trickle of
	 * removes to get through.
	 */
	if (dd->dd_parent == NULL) {
		spa_t *spa = dd->dd_pool->dp_spa;
		uint64_t poolsize = dsl_pool_adjustedsize(dd->dd_pool, netfree);
		deferred = metaslab_class_get_deferred(spa_normal_class(spa));
		if (poolsize - deferred < quota) {
			quota = poolsize - deferred;
			retval = ENOSPC;
		}
	}

	/*
	 * If they are requesting more space, and our current estimate
	 * is over quota, they get to try again unless the actual
	 * on-disk is over quota and there are no pending changes (which
	 * may free up space for us).
	 */
	if (used_on_disk + est_inflight >= quota) {
		if (est_inflight > 0 || used_on_disk < quota ||
		    (retval == ENOSPC && used_on_disk < quota + deferred))
			retval = ERESTART;
		dprintf_dd(dd, "failing: used=%lluK inflight = %lluK "
		    "quota=%lluK tr=%lluK err=%d\n",
		    used_on_disk>>10, est_inflight>>10,
		    quota>>10, asize>>10, retval);
		mutex_exit(&dd->dd_lock);
		return (SET_ERROR(retval));
	}

	/* We need to up our estimated delta before dropping dd_lock */
	dd->dd_tempreserved[txgidx] += asize;

	parent_rsrv = parent_delta(dd, used_on_disk + est_inflight,
	    asize - ref_rsrv);
	mutex_exit(&dd->dd_lock);

	tr = kmem_zalloc(sizeof (struct tempreserve), KM_SLEEP);
	tr->tr_ds = dd;
	tr->tr_size = asize;
	list_insert_tail(tr_list, tr);

	/* see if it’s OK with our parent */
	if (dd->dd_parent && parent_rsrv) {
		boolean_t ismos = (dd->dd_phys->dd_head_dataset_obj == 0);

		return (dsl_dir_tempreserve_impl(dd->dd_parent,
		    parent_rsrv, netfree, ismos, TRUE, tr_list, tx, FALSE));
	} else {
		return (0);
	}
}

/*
 * Reserve space in this dsl_dir, to be used in this tx’s txg.
 * After the space has been dirtied (and dsl_dir_willuse_space()
 * has been called), the reservation should be canceled, using
 * dsl_dir_tempreserve_clear().
 */
int
dsl_dir_tempreserve_space(dsl_dir_t *dd, uint64_t lsize, uint64_t asize,
    uint64_t fsize, uint64_t usize, void **tr_cookiep, dmu_tx_t *tx)
{
	int err;
```

```
	list_t *tr_list;

	if (asize == 0) {
		*tr_cookiep = NULL;
		return (0);
	}

	tr_list = kmem_alloc(sizeof (list_t), KM_SLEEP);
	list_create(tr_list, sizeof (struct tempreserve),
	    offsetof(struct tempreserve, tr_node));
	ASSERT3S(asize, >, 0);
	ASSERT3S(fsize, >=, 0);

	err = arc_tempreserve_space(lsize, tx->tx_txg);
	if (err == 0) {
		struct tempreserve *tr;

		tr = kmem_zalloc(sizeof (struct tempreserve), KM_SLEEP);
		tr->tr_size = lsize;
		list_insert_tail(tr_list, tr);

		err = dsl_pool_tempreserve_space(dd->dd_pool, asize, tx);
	} else {
		if (err == EAGAIN) {
			txg_delay(dd->dd_pool, tx->tx_txg,
			    MSEC2NSEC(10), MSEC2NSEC(10));
			err = SET_ERROR(ERESTART);
		}
		dsl_pool_memory_pressure(dd->dd_pool);
	}

	if (err == 0) {
		struct tempreserve *tr;

		tr = kmem_zalloc(sizeof (struct tempreserve), KM_SLEEP);
		tr->tr_dp = dd->dd_pool;
		tr->tr_size = asize;
		list_insert_tail(tr_list, tr);

		err = dsl_dir_tempreserve_impl(dd, asize, fsize >= asize,
		    FALSE, asize > usize, tr_list, tx, TRUE);
	}

	if (err != 0)
		dsl_dir_tempreserve_clear(tr_list, tx);
	else
		*tr_cookiep = tr_list;

	return (err);
}

/*
 * Clear a temporary reservation that we previously made with
 * dsl_dir_tempreserve_space().
 */
void
dsl_dir_tempreserve_clear(void *tr_cookie, dmu_tx_t *tx)
{
	int txgidx = tx->tx_txg & TXG_MASK;
	list_t *tr_list = tr_cookie;
	struct tempreserve *tr;

	ASSERT3U(tx->tx_txg, !=, 0);

	if (tr_cookie == NULL)
		return;
```

```
        while (tr = list_head(tr_list)) {
                if (tr->tr_dp) {
                        dsl_pool_tempreserve_clear(tr->tr_dp, tr->tr_size, tx);
                } else if (tr->tr_ds) {
                        mutex_enter(&tr->tr_ds->dd_lock);
                        ASSERT3U(tr->tr_ds->dd_tempreserved[txgidx], >=,
                            tr->tr_size);
                        tr->tr_ds->dd_tempreserved[txgidx] -= tr->tr_size;
                        mutex_exit(&tr->tr_ds->dd_lock);
                } else {
                        arc_tempreserve_clear(tr->tr_size);
                }
                list_remove(tr_list, tr);
                kmem_free(tr, sizeof (struct tempreserve));
        }

        kmem_free(tr_list, sizeof (list_t));
}

static void
dsl_dir_willuse_space_impl(dsl_dir_t *dd, int64_t space, dmu_tx_t *tx)
{
        int64_t parent_space;
        uint64_t est_used;

        mutex_enter(&dd->dd_lock);
        if (space > 0)
                dd->dd_space_towrite[tx->tx_txg & TXG_MASK] += space;

        est_used = dsl_dir_space_towrite(dd) + dd->dd_phys->dd_used_bytes;
        parent_space = parent_delta(dd, est_used, space);
        mutex_exit(&dd->dd_lock);

        /* Make sure that we clean up dd_space_to* */
        dsl_dir_dirty(dd, tx);

        /* XXX this is potentially expensive and unnecessary... */
        if (parent_space && dd->dd_parent)
                dsl_dir_willuse_space_impl(dd->dd_parent, parent_space, tx);
}

/*
 * Call in open context when we think we're going to write/free space,
 * eg. when dirtying data.  Be conservative (ie. OK to write less than
 * this or free more than this, but don't write more or free less).
 */
void
dsl_dir_willuse_space(dsl_dir_t *dd, int64_t space, dmu_tx_t *tx)
{
        dsl_pool_willuse_space(dd->dd_pool, space, tx);
        dsl_dir_willuse_space_impl(dd, space, tx);
}

/* call from syncing context when we actually write/free space for this dd */
void
dsl_dir_diduse_space(dsl_dir_t *dd, dd_used_t type,
    int64_t used, int64_t compressed, int64_t uncompressed, dmu_tx_t *tx)
{
        int64_t accounted_delta;
        boolean_t needlock = !MUTEX_HELD(&dd->dd_lock);

        ASSERT(dmu_tx_is_syncing(tx));
        ASSERT(type < DD_USED_NUM);

        if (needlock)
```

```
                mutex_enter(&dd->dd_lock);
        accounted_delta = parent_delta(dd, dd->dd_phys->dd_used_bytes, used);
        ASSERT(used >= 0 || dd->dd_phys->dd_used_bytes >= -used);
        ASSERT(compressed >= 0 ||
            dd->dd_phys->dd_compressed_bytes >= -compressed);
        ASSERT(uncompressed >= 0 ||
            dd->dd_phys->dd_uncompressed_bytes >= -uncompressed);
        dmu_buf_will_dirty(dd->dd_dbuf, tx);
        dd->dd_phys->dd_used_bytes += used;
        dd->dd_phys->dd_uncompressed_bytes += uncompressed;
        dd->dd_phys->dd_compressed_bytes += compressed;

        if (dd->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
                ASSERT(used > 0 ||
                    dd->dd_phys->dd_used_breakdown[type] >= -used);
                dd->dd_phys->dd_used_breakdown[type] += used;
#ifdef DEBUG
                dd_used_t t;
                uint64_t u = 0;
                for (t = 0; t < DD_USED_NUM; t++)
                        u += dd->dd_phys->dd_used_breakdown[t];
                ASSERT3U(u, ==, dd->dd_phys->dd_used_bytes);
#endif
        }
        if (needlock)
                mutex_exit(&dd->dd_lock);

        if (dd->dd_parent != NULL) {
                dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD,
                    accounted_delta, compressed, uncompressed, tx);
                dsl_dir_transfer_space(dd->dd_parent,
                    used - accounted_delta,
                    DD_USED_CHILD_RSRV, DD_USED_CHILD, tx);
        }
}

void
dsl_dir_transfer_space(dsl_dir_t *dd, int64_t delta,
    dd_used_t oldtype, dd_used_t newtype, dmu_tx_t *tx)
{
        boolean_t needlock = !MUTEX_HELD(&dd->dd_lock);

        ASSERT(dmu_tx_is_syncing(tx));
        ASSERT(oldtype < DD_USED_NUM);
        ASSERT(newtype < DD_USED_NUM);

        if (delta == 0 || !(dd->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN))
                return;

        if (needlock)
                mutex_enter(&dd->dd_lock);
        ASSERT(delta > 0 ?
            dd->dd_phys->dd_used_breakdown[oldtype] >= delta :
            dd->dd_phys->dd_used_breakdown[newtype] >= -delta);
        ASSERT(dd->dd_phys->dd_used_bytes >= ABS(delta));
        dmu_buf_will_dirty(dd->dd_dbuf, tx);
        dd->dd_phys->dd_used_breakdown[oldtype] -= delta;
        dd->dd_phys->dd_used_breakdown[newtype] += delta;
        if (needlock)
                mutex_exit(&dd->dd_lock);
}

typedef struct dsl_dir_set_qr_arg {
        const char *ddsqra_name;
        zprop_source_t ddsqra_source;
        uint64_t ddsqra_value;
```

```
} dsl_dir_set_qr_arg_t;

static int
dsl_dir_set_quota_check(void *arg, dmu_tx_t *tx)
{
        dsl_dir_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        int error;
        uint64_t towrite, newval;

        error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
        if (error != 0)
                return (error);

        error = dsl_prop_predict(ds->ds_dir, "quota",
            ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
        if (error != 0) {
                dsl_dataset_rele(ds, FTAG);
                return (error);
        }

        if (newval == 0) {
                dsl_dataset_rele(ds, FTAG);
                return (0);
        }

        mutex_enter(&ds->ds_dir->dd_lock);
        /*
         * If we are doing the preliminary check in open context, and
         * there are pending changes, then don’t fail it, since the
         * pending changes could under-estimate the amount of space to be
         * freed up.
         */
        towrite = dsl_dir_space_towrite(ds->ds_dir);
        if ((dmu_tx_is_syncing(tx) || towrite == 0) &&
            (newval < ds->ds_dir->dd_phys->dd_reserved ||
            newval < ds->ds_dir->dd_phys->dd_used_bytes + towrite)) {
                error = SET_ERROR(ENOSPC);
        }
        mutex_exit(&ds->ds_dir->dd_lock);
        dsl_dataset_rele(ds, FTAG);
        return (error);
}

static void
dsl_dir_set_quota_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dir_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        uint64_t newval;

        VERIFY0(dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds));

        if (spa_version(dp->dp_spa) >= SPA_VERSION_RECVD_PROPS) {
#endif /* ! codereview */
                dsl_prop_set_sync_impl(ds, zfs_prop_to_name(ZFS_PROP_QUOTA),
                    ddsqra->ddsqra_source, sizeof (ddsqra->ddsqra_value), 1,
                    &ddsqra->ddsqra_value, tx);

                VERIFY0(dsl_prop_get_int_ds(ds,
                    zfs_prop_to_name(ZFS_PROP_QUOTA), &newval));
        } else {
                newval = ddsqra->ddsqra_value;
                spa_history_log_internal_ds(ds, "set", tx, "%s=%lld",
```

```
                    zfs_prop_to_name(ZFS_PROP_QUOTA), (longlong_t)newval);
        }
#endif /* ! codereview */

        dmu_buf_will_dirty(ds->ds_dir->dd_dbuf, tx);
        mutex_enter(&ds->ds_dir->dd_lock);
        ds->ds_dir->dd_phys->dd_quota = newval;
        mutex_exit(&ds->ds_dir->dd_lock);
        dsl_dataset_rele(ds, FTAG);
}

int
dsl_dir_set_quota(const char *ddname, zprop_source_t source, uint64_t quota)
{
        dsl_dir_set_qr_arg_t ddsqra;

        ddsqra.ddsqra_name = ddname;
        ddsqra.ddsqra_source = source;
        ddsqra.ddsqra_value = quota;

        return (dsl_sync_task(ddname, dsl_dir_set_quota_check,
            dsl_dir_set_quota_sync, &ddsqra, 0));
}

int
dsl_dir_set_reservation_check(void *arg, dmu_tx_t *tx)
{
        dsl_dir_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        dsl_dir_t *dd;
        uint64_t newval, used, avail;
        int error;

        error = dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds);
        if (error != 0)
                return (error);
        dd = ds->ds_dir;

        /*
         * If we are doing the preliminary check in open context, the
         * space estimates may be inaccurate.
         */
        if (!dmu_tx_is_syncing(tx)) {
                dsl_dataset_rele(ds, FTAG);
                return (0);
        }

        error = dsl_prop_predict(ds->ds_dir,
            zfs_prop_to_name(ZFS_PROP_RESERVATION),
            ddsqra->ddsqra_source, ddsqra->ddsqra_value, &newval);
        if (error != 0) {
                dsl_dataset_rele(ds, FTAG);
                return (error);
        }

        mutex_enter(&dd->dd_lock);
        used = dd->dd_phys->dd_used_bytes;
        mutex_exit(&dd->dd_lock);

        if (dd->dd_parent) {
                avail = dsl_dir_space_available(dd->dd_parent,
                    NULL, 0, FALSE);
        } else {
                avail = dsl_pool_adjustedsize(dd->dd_pool, B_FALSE) - used;
        }
        if (MAX(used, newval) > MAX(used, dd->dd_phys->dd_reserved)) {
                uint64_t delta = MAX(used, newval) -
                    MAX(used, dd->dd_phys->dd_reserved);

                if (delta > avail ||
                    (dd->dd_phys->dd_quota > 0 &&
                    newval > dd->dd_phys->dd_quota))
                        error = SET_ERROR(ENOSPC);
        }

        dsl_dataset_rele(ds, FTAG);
        return (error);
}

void
dsl_dir_set_reservation_sync_impl(dsl_dir_t *dd, uint64_t value, dmu_tx_t *tx)
{
        uint64_t used;
        int64_t delta;

        dmu_buf_will_dirty(dd->dd_dbuf, tx);

        mutex_enter(&dd->dd_lock);
        used = dd->dd_phys->dd_used_bytes;
        delta = MAX(used, value) - MAX(used, dd->dd_phys->dd_reserved);
        dd->dd_phys->dd_reserved = value;

        if (dd->dd_parent != NULL) {
                /* Roll up this additional usage into our ancestors */
                dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD_RSRV,
                    delta, 0, 0, tx);
        }
        mutex_exit(&dd->dd_lock);
}


static void
dsl_dir_set_reservation_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dir_set_qr_arg_t *ddsqra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        uint64_t newval;

        VERIFY0(dsl_dataset_hold(dp, ddsqra->ddsqra_name, FTAG, &ds));

        if (spa_version(dp->dp_spa) >= SPA_VERSION_RECVD_PROPS) {
                dsl_prop_set_sync_impl(ds,
                    zfs_prop_to_name(ZFS_PROP_RESERVATION),
        dsl_prop_set_sync_impl(ds, zfs_prop_to_name(ZFS_PROP_RESERVATION),
                    ddsqra->ddsqra_source, sizeof (ddsqra->ddsqra_value), 1,
                    &ddsqra->ddsqra_value, tx);

                VERIFY0(dsl_prop_get_int_ds(ds,
                    zfs_prop_to_name(ZFS_PROP_RESERVATION), &newval));
        } else {
                newval = ddsqra->ddsqra_value;
                spa_history_log_internal_ds(ds, "set", tx, "%s=%lld",
                    zfs_prop_to_name(ZFS_PROP_RESERVATION),
                    (longlong_t)newval);
        }
#endif /* ! codereview */

        dsl_dir_set_reservation_sync_impl(ds->ds_dir, newval, tx);
        dsl_dataset_rele(ds, FTAG);
```

```
}

int
dsl_dir_set_reservation(const char *ddname, zprop_source_t source,
    uint64_t reservation)
{
        dsl_dir_set_qr_arg_t ddsqra;

        ddsqra.ddsqra_name = ddname;
        ddsqra.ddsqra_source = source;
        ddsqra.ddsqra_value = reservation;

        return (dsl_sync_task(ddname, dsl_dir_set_reservation_check,
            dsl_dir_set_reservation_sync, &ddsqra, 0));
}

static dsl_dir_t *
closest_common_ancestor(dsl_dir_t *ds1, dsl_dir_t *ds2)
{
        for (; ds1; ds1 = ds1->dd_parent) {
                dsl_dir_t *dd;
                for (dd = ds2; dd; dd = dd->dd_parent) {
                        if (ds1 == dd)
                                return (dd);
                }
        }
        return (NULL);
}

/*
 * If delta is applied to dd, how much of that delta would be applied to
 * ancestor?  Syncing context only.
 */
static int64_t
would_change(dsl_dir_t *dd, int64_t delta, dsl_dir_t *ancestor)
{
        if (dd == ancestor)
                return (delta);

        mutex_enter(&dd->dd_lock);
        delta = parent_delta(dd, dd->dd_phys->dd_used_bytes, delta);
        mutex_exit(&dd->dd_lock);
        return (would_change(dd->dd_parent, delta, ancestor));
}

typedef struct dsl_dir_rename_arg {
        const char *ddra_oldname;
        const char *ddra_newname;
} dsl_dir_rename_arg_t;

/* ARGSUSED */
static int
dsl_valid_rename(dsl_pool_t *dp, dsl_dataset_t *ds, void *arg)
{
        int *deltap = arg;
        char namebuf[MAXNAMELEN];

        dsl_dataset_name(ds, namebuf);

        if (strlen(namebuf) + *deltap >= MAXNAMELEN)
                return (SET_ERROR(ENAMETOOLONG));
        return (0);
}

static int
dsl_dir_rename_check(void *arg, dmu_tx_t *tx)
```

```
{
        dsl_dir_rename_arg_t *ddra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dir_t *dd, *newparent;
        const char *mynewname;
        int error;
        int delta = strlen(ddra->ddra_newname) - strlen(ddra->ddra_oldname);

        /* target dir should exist */
        error = dsl_dir_hold(dp, ddra->ddra_oldname, FTAG, &dd, NULL);
        if (error != 0)
                return (error);

        /* new parent should exist */
        error = dsl_dir_hold(dp, ddra->ddra_newname, FTAG,
            &newparent, &mynewname);
        if (error != 0) {
                dsl_dir_rele(dd, FTAG);
                return (error);
        }

        /* can’t rename to different pool */
        if (dd->dd_pool != newparent->dd_pool) {
                dsl_dir_rele(newparent, FTAG);
                dsl_dir_rele(dd, FTAG);
                return (SET_ERROR(ENXIO));
        }

        /* new name should not already exist */
        if (mynewname == NULL) {
                dsl_dir_rele(newparent, FTAG);
                dsl_dir_rele(dd, FTAG);
                return (SET_ERROR(EEXIST));
        }

        /* if the name length is growing, validate child name lengths */
        if (delta > 0) {
                error = dmu_objset_find_dp(dp, dd->dd_object, dsl_valid_rename,
                    &delta, DS_FIND_CHILDREN | DS_FIND_SNAPSHOTS);
                if (error != 0) {
                        dsl_dir_rele(newparent, FTAG);
                        dsl_dir_rele(dd, FTAG);
                        return (error);
                }
        }

        if (newparent != dd->dd_parent) {
                /* is there enough space? */
                uint64_t myspace =
                    MAX(dd->dd_phys->dd_used_bytes, dd->dd_phys->dd_reserved);

                /* no rename into our descendant */
                if (closest_common_ancestor(dd, newparent) == dd) {
                        dsl_dir_rele(newparent, FTAG);
                        dsl_dir_rele(dd, FTAG);
                        return (SET_ERROR(EINVAL));
                }

                error = dsl_dir_transfer_possible(dd->dd_parent,
                    newparent, myspace);
                if (error != 0) {
                        dsl_dir_rele(newparent, FTAG);
                        dsl_dir_rele(dd, FTAG);
                        return (error);
                }
        }
```

```

        dsl_dir_rele(newparent, FTAG);
        dsl_dir_rele(dd, FTAG);
        return (0);
}

static void
dsl_dir_rename_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dir_rename_arg_t *ddra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dir_t *dd, *newparent;
        const char *mynewname;
        int error;
        objset_t *mos = dp->dp_meta_objset;

        VERIFY0(dsl_dir_hold(dp, ddra->ddra_oldname, FTAG, &dd, NULL));
        VERIFY0(dsl_dir_hold(dp, ddra->ddra_newname, FTAG, &newparent,
            &mynewname));

        /* Log this before we change the name. */
        spa_history_log_internal_dd(dd, "rename", tx,
            "-> %s", ddra->ddra_newname);

        if (newparent != dd->dd_parent) {
                dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD,
                    -dd->dd_phys->dd_used_bytes,
                    -dd->dd_phys->dd_compressed_bytes,
                    -dd->dd_phys->dd_uncompressed_bytes, tx);
                dsl_dir_diduse_space(newparent, DD_USED_CHILD,
                    dd->dd_phys->dd_used_bytes,
                    dd->dd_phys->dd_compressed_bytes,
                    dd->dd_phys->dd_uncompressed_bytes, tx);

                if (dd->dd_phys->dd_reserved > dd->dd_phys->dd_used_bytes) {
                        uint64_t unused_rsrv = dd->dd_phys->dd_reserved -
                            dd->dd_phys->dd_used_bytes;

                        dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD_RSRV,
                            -unused_rsrv, 0, 0, tx);
                        dsl_dir_diduse_space(newparent, DD_USED_CHILD_RSRV,
                            unused_rsrv, 0, 0, tx);
                }
        }

        dmu_buf_will_dirty(dd->dd_dbuf, tx);

        /* remove from old parent zapobj */
        error = zap_remove(mos, dd->dd_parent->dd_phys->dd_child_dir_zapobj,
            dd->dd_myname, tx);
        ASSERT0(error);

        (void) strcpy(dd->dd_myname, mynewname);
        dsl_dir_rele(dd->dd_parent, dd);
        dd->dd_phys->dd_parent_obj = newparent->dd_object;
        VERIFY0(dsl_dir_hold_obj(dp,
            newparent->dd_object, NULL, dd, &dd->dd_parent));

        /* add to new parent zapobj */
        VERIFY0(zap_add(mos, newparent->dd_phys->dd_child_dir_zapobj,
            dd->dd_myname, 8, 1, &dd->dd_object, tx));

        dsl_prop_notify_all(dd);

        dsl_dir_rele(newparent, FTAG);
        dsl_dir_rele(dd, FTAG);
```

```
}

int
dsl_dir_rename(const char *oldname, const char *newname)
{
        dsl_dir_rename_arg_t ddra;

        ddra.ddra_oldname = oldname;
        ddra.ddra_newname = newname;

        return (dsl_sync_task(oldname,
            dsl_dir_rename_check, dsl_dir_rename_sync, &ddra, 3));
}

int
dsl_dir_transfer_possible(dsl_dir_t *sdd, dsl_dir_t *tdd, uint64_t space)
{
        dsl_dir_t *ancestor;
        int64_t adelta;
        uint64_t avail;

        ancestor = closest_common_ancestor(sdd, tdd);
        adelta = would_change(sdd, -space, ancestor);
        avail = dsl_dir_space_available(tdd, ancestor, adelta, FALSE);
        if (avail < space)
                return (SET_ERROR(ENOSPC));

        return (0);
}

timestruc_t
dsl_dir_snap_cmtime(dsl_dir_t *dd)
{
        timestruc_t t;

        mutex_enter(&dd->dd_lock);
        t = dd->dd_snap_cmtime;
        mutex_exit(&dd->dd_lock);

        return (t);
}

void
dsl_dir_snap_cmtime_update(dsl_dir_t *dd)
{
        timestruc_t t;

        gethrestime(&t);
        mutex_enter(&dd->dd_lock);
        dd->dd_snap_cmtime = t;
        mutex_exit(&dd->dd_lock);
}
```

```
**********************************************************
   29134 Fri May 17 22:54:38 2013
new/usr/src/uts/common/fs/zfs/dsl_prop.c
3699 zfs hold or release of a non-existent snapshot does not output error
3739 cannot set zfs quota or reservation on pool version < 22
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Eric Shrock <eric.schrock@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 Martin Matuska. All rights reserved.
#endif /* ! codereview */
 */

#include <sys/zfs_context.h>
#include <sys/dmu.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/spa.h>
#include <sys/zap.h>
#include <sys/fs/zfs.h>

#include "zfs_prop.h"

#define ZPROP_INHERIT_SUFFIX "$inherit"
#define ZPROP_RECVD_SUFFIX "$recvd"

static int
dodefault(const char *propname, int intsz, int numints, void *buf)
{
        zfs_prop_t prop;

        /*
         * The setonce properties are read-only, BUT they still
         * have a default value that can be used as the initial
         * value.
         */
        if ((prop = zfs_name_to_prop(propname)) == ZPROP_INVAL ||
            (zfs_prop_readonly(prop) && !zfs_prop_setonce(prop)))
                return (SET_ERROR(ENOENT));

        if (zfs_prop_get_type(prop) == PROP_TYPE_STRING) {
                if (intsz != 1)
                        return (SET_ERROR(EOVERFLOW));
                (void) strncpy(buf, zfs_prop_default_string(prop),
                    numints);
        } else {
                if (intsz != 8 || numints < 1)
                        return (SET_ERROR(EOVERFLOW));

                *(uint64_t *)buf = zfs_prop_default_numeric(prop);
        }

        return (0);
}

int
dsl_prop_get_dd(dsl_dir_t *dd, const char *propname,
    int intsz, int numints, void *buf, char *setpoint, boolean_t snapshot)
{
        int err = ENOENT;
        dsl_dir_t *target = dd;
        objset_t *mos = dd->dd_pool->dp_meta_objset;
        zfs_prop_t prop;
        boolean_t inheritable;
        boolean_t inheriting = B_FALSE;
        char *inheritstr;
        char *recvdstr;

        ASSERT(dsl_pool_config_held(dd->dd_pool));

        if (setpoint)
                setpoint[0] = '\0';

        prop = zfs_name_to_prop(propname);
        inheritable = (prop == ZPROP_INVAL || zfs_prop_inheritable(prop));
        inheritstr = kmem_asprintf("%s%s", propname, ZPROP_INHERIT_SUFFIX);
        recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

        /*
         * Note: dd may become NULL, therefore we shouldn't dereference it
         * after this loop.
         */
        for (; dd != NULL; dd = dd->dd_parent) {
                if (dd != target || snapshot) {
                        if (!inheritable)
                                break;
                        inheriting = B_TRUE;
                }

                /* Check for a local value. */
                err = zap_lookup(mos, dd->dd_phys->dd_props_zapobj, propname,
                    intsz, numints, buf);
                if (err != ENOENT) {
                        if (setpoint != NULL && err == 0)
                                dsl_dir_name(dd, setpoint);
                        break;
                }

                /*
                 * Skip the check for a received value if there is an explicit
                 * inheritance entry.
                 */
                err = zap_contains(mos, dd->dd_phys->dd_props_zapobj,
                    inheritstr);
                if (err != 0 && err != ENOENT)
                        break;
                if (err == ENOENT) {
                        /* Check for a received value. */
                        err = zap_lookup(mos, dd->dd_phys->dd_props_zapobj,
                            recvdstr, intsz, numints, buf);
                        if (err != ENOENT) {
                                if (setpoint != NULL && err == 0) {
                                        if (inheriting) {
                                                dsl_dir_name(dd, setpoint);
                                        } else {
                                                (void) strcpy(setpoint,
                                                    ZPROP_SOURCE_VAL_RECVD);
                                        }
                                }
                                break;
                        }
                }

                /*
                 * If we found an explicit inheritance entry, err is zero even
                 * though we haven't yet found the value, so reinitializing err
                 * at the end of the loop (instead of at the beginning) ensures
                 * that err has a valid post-loop value.
                 */
                err = SET_ERROR(ENOENT);
        }

        if (err == ENOENT)
                err = dodefault(propname, intsz, numints, buf);

        strfree(inheritstr);
        strfree(recvdstr);

        return (err);
}

int
dsl_prop_get_ds(dsl_dataset_t *ds, const char *propname,
    int intsz, int numints, void *buf, char *setpoint)
{
        zfs_prop_t prop = zfs_name_to_prop(propname);
        boolean_t inheritable;
        boolean_t snapshot;
        uint64_t zapobj;

        ASSERT(dsl_pool_config_held(ds->ds_dir->dd_pool));
        inheritable = (prop == ZPROP_INVAL || zfs_prop_inheritable(prop));
        snapshot = (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds));
        zapobj = (ds->ds_phys == NULL ? 0 : ds->ds_phys->ds_props_obj);

        if (zapobj != 0) {
                objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
                int err;

                ASSERT(snapshot);

                /* Check for a local value. */
                err = zap_lookup(mos, zapobj, propname, intsz, numints, buf);
                if (err != ENOENT) {
                        if (setpoint != NULL && err == 0)
                                dsl_dataset_name(ds, setpoint);
                        return (err);
                }

                /*
                 * Skip the check for a received value if there is an explicit
```

```
                 * inheritance entry.
                 */
                if (inheritable) {
                        char *inheritstr = kmem_asprintf("%s%s", propname,
                            ZPROP_INHERIT_SUFFIX);
                        err = zap_contains(mos, zapobj, inheritstr);
                        strfree(inheritstr);
                        if (err != 0 && err != ENOENT)
                                return (err);
                }

                if (err == ENOENT) {
                        /* Check for a received value. */
                        char *recvdstr = kmem_asprintf("%s%s", propname,
                            ZPROP_RECVD_SUFFIX);
                        err = zap_lookup(mos, zapobj, recvdstr,
                            intsz, numints, buf);
                        strfree(recvdstr);
                        if (err != ENOENT) {
                                if (setpoint != NULL && err == 0)
                                        (void) strcpy(setpoint,
                                            ZPROP_SOURCE_VAL_RECVD);
                                return (err);
                        }
                }
        }

        return (dsl_prop_get_dd(ds->ds_dir, propname,
            intsz, numints, buf, setpoint, snapshot));
}

/*
 * Register interest in the named property.  We'll call the callback
 * once to notify it of the current property value, and again each time
 * the property changes, until this callback is unregistered.
 *
 * Return 0 on success, errno if the prop is not an integer value.
 */
int
dsl_prop_register(dsl_dataset_t *ds, const char *propname,
    dsl_prop_changed_cb_t *callback, void *cbarg)
{
        dsl_dir_t *dd = ds->ds_dir;
        dsl_pool_t *dp = dd->dd_pool;
        uint64_t value;
        dsl_prop_cb_record_t *cbr;
        int err;

        ASSERT(dsl_pool_config_held(dp));

        err = dsl_prop_get_int_ds(ds, propname, &value);
        if (err != 0)
                return (err);

        cbr = kmem_alloc(sizeof (dsl_prop_cb_record_t), KM_SLEEP);
        cbr->cbr_ds = ds;
        cbr->cbr_propname = kmem_alloc(strlen(propname)+1, KM_SLEEP);
        (void) strcpy((char *)cbr->cbr_propname, propname);
        cbr->cbr_func = callback;
        cbr->cbr_arg = cbarg;
        mutex_enter(&dd->dd_lock);
        list_insert_head(&dd->dd_prop_cbs, cbr);
        mutex_exit(&dd->dd_lock);

        cbr->cbr_func(cbr->cbr_arg, value);
        return (0);
```

```
}

int
dsl_prop_get(const char *dsname, const char *propname,
    int intsz, int numints, void *buf, char *setpoint)
{
        objset_t *os;
        int error;

        error = dmu_objset_hold(dsname, FTAG, &os);
        if (error != 0)
                return (error);

        error = dsl_prop_get_ds(dmu_objset_ds(os), propname,
            intsz, numints, buf, setpoint);

        dmu_objset_rele(os, FTAG);
        return (error);
}

/*
 * Get the current property value.  It may have changed by the time this
 * function returns, so it is NOT safe to follow up with
 * dsl_prop_register() and assume that the value has not changed in
 * between.
 *
 * Return 0 on success, ENOENT if ddname is invalid.
 */
int
dsl_prop_get_integer(const char *ddname, const char *propname,
    uint64_t *valuep, char *setpoint)
{
        return (dsl_prop_get(ddname, propname, 8, 1, valuep, setpoint));
}

int
dsl_prop_get_int_ds(dsl_dataset_t *ds, const char *propname,
    uint64_t *valuep)
{
        return (dsl_prop_get_ds(ds, propname, 8, 1, valuep, NULL));
}

/*
 * Predict the effective value of the given special property if it were set with
 * the given value and source. This is not a general purpose function. It exists
 * only to handle the special requirements of the quota and reservation
 * properties. The fact that these properties are non-inheritable greatly
 * simplifies the prediction logic.
 *
 * Returns 0 on success, a positive error code on failure, or -1 if called with
 * a property not handled by this function.
 */
int
dsl_prop_predict(dsl_dir_t *dd, const char *propname,
    zprop_source_t source, uint64_t value, uint64_t *newvalp)
{
        zfs_prop_t prop = zfs_name_to_prop(propname);
        objset_t *mos;
        uint64_t zapobj;
        uint64_t version;
        char *recvdstr;
        int err = 0;

        switch (prop) {
        case ZFS_PROP_QUOTA:
        case ZFS_PROP_RESERVATION:
```

```
        case ZFS_PROP_REFQUOTA:
        case ZFS_PROP_REFRESERVATION:
                break;
        default:
                return (-1);
        }

        mos = dd->dd_pool->dp_meta_objset;
        zapobj = dd->dd_phys->dd_props_zapobj;
        recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

        version = spa_version(dd->dd_pool->dp_spa);
        if (version < SPA_VERSION_RECVD_PROPS) {
                if (source & ZPROP_SRC_NONE)
                        source = ZPROP_SRC_NONE;
                else if (source & ZPROP_SRC_RECEIVED)
                        source = ZPROP_SRC_LOCAL;
        }

        switch (source) {
        case ZPROP_SRC_NONE:
                /* Revert to the received value, if any. */
                err = zap_lookup(mos, zapobj, recvdstr, 8, 1, newvalp);
                if (err == ENOENT)
                        *newvalp = 0;
                break;
        case ZPROP_SRC_LOCAL:
                *newvalp = value;
                break;
        case ZPROP_SRC_RECEIVED:
                /*
                 * If there’s no local setting, then the new received value will
                 * be the effective value.
                 */
                err = zap_lookup(mos, zapobj, propname, 8, 1, newvalp);
                if (err == ENOENT)
                        *newvalp = value;
                break;
        case (ZPROP_SRC_NONE | ZPROP_SRC_RECEIVED):
                /*
                 * We’re clearing the received value, so the local setting (if
                 * it exists) remains the effective value.
                 */
                err = zap_lookup(mos, zapobj, propname, 8, 1, newvalp);
                if (err == ENOENT)
                        *newvalp = 0;
                break;
        default:
                panic("unexpected property source: %d", source);
        }

        strfree(recvdstr);

        if (err == ENOENT)
                return (0);

        return (err);
}

/*
 * Unregister this callback.  Return 0 on success, ENOENT if ddname is
 * invalid, ENOMSG if no matching callback registered.
 */
int
dsl_prop_unregister(dsl_dataset_t *ds, const char *propname,
    dsl_prop_changed_cb_t *callback, void *cbarg)
```

```
{
        dsl_dir_t *dd = ds->ds_dir;
        dsl_prop_cb_record_t *cbr;

        mutex_enter(&dd->dd_lock);
        for (cbr = list_head(&dd->dd_prop_cbs);
            cbr; cbr = list_next(&dd->dd_prop_cbs, cbr)) {
                if (cbr->cbr_ds == ds &&
                    cbr->cbr_func == callback &&
                    cbr->cbr_arg == cbarg &&
                    strcmp(cbr->cbr_propname, propname) == 0)
                        break;
        }

        if (cbr == NULL) {
                mutex_exit(&dd->dd_lock);
                return (SET_ERROR(ENOMSG));
        }

        list_remove(&dd->dd_prop_cbs, cbr);
        mutex_exit(&dd->dd_lock);
        kmem_free((void*)cbr->cbr_propname, strlen(cbr->cbr_propname)+1);
        kmem_free(cbr, sizeof (dsl_prop_cb_record_t));

        return (0);
}

boolean_t
dsl_prop_hascb(dsl_dataset_t *ds)
{
        dsl_dir_t *dd = ds->ds_dir;
        boolean_t rv = B_FALSE;
        dsl_prop_cb_record_t *cbr;

        mutex_enter(&dd->dd_lock);
        for (cbr = list_head(&dd->dd_prop_cbs); cbr;
            cbr = list_next(&dd->dd_prop_cbs, cbr)) {
                if (cbr->cbr_ds == ds) {
                        rv = B_TRUE;
                        break;
                }
        }
        mutex_exit(&dd->dd_lock);
        return (rv);
}

/* ARGSUSED */
static int
dsl_prop_notify_all_cb(dsl_pool_t *dp, dsl_dataset_t *ds, void *arg)
{
        dsl_dir_t *dd = ds->ds_dir;
        dsl_prop_cb_record_t *cbr;

        mutex_enter(&dd->dd_lock);
        for (cbr = list_head(&dd->dd_prop_cbs); cbr;
            cbr = list_next(&dd->dd_prop_cbs, cbr)) {
                uint64_t value;

                if (dsl_prop_get_ds(cbr->cbr_ds, cbr->cbr_propname,
                    sizeof (value), 1, &value, NULL) == 0)
                        cbr->cbr_func(cbr->cbr_arg, value);
        }
        mutex_exit(&dd->dd_lock);

        return (0);
}
```

```

/*
 * Update all property values for ddobj & its descendants.  This is used
 * when renaming the dir.
 */
void
dsl_prop_notify_all(dsl_dir_t *dd)
{
        dsl_pool_t *dp = dd->dd_pool;
        ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));
        (void) dmu_objset_find_dp(dp, dd->dd_object, dsl_prop_notify_all_cb,
            NULL, DS_FIND_CHILDREN);
}

static void
dsl_prop_changed_notify(dsl_pool_t *dp, uint64_t ddobj,
    const char *propname, uint64_t value, int first)
{
        dsl_dir_t *dd;
        dsl_prop_cb_record_t *cbr;
        objset_t *mos = dp->dp_meta_objset;
        zap_cursor_t zc;
        zap_attribute_t *za;
        int err;

        ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));
        err = dsl_dir_hold_obj(dp, ddobj, NULL, FTAG, &dd);
        if (err)
                return;

        if (!first) {
                /*
                 * If the prop is set here, then this change is not
                 * being inherited here or below; stop the recursion.
                 */
                err = zap_contains(mos, dd->dd_phys->dd_props_zapobj, propname);
                if (err == 0) {
                        dsl_dir_rele(dd, FTAG);
                        return;
                }
                ASSERT3U(err, ==, ENOENT);
        }

        mutex_enter(&dd->dd_lock);
        for (cbr = list_head(&dd->dd_prop_cbs); cbr;
            cbr = list_next(&dd->dd_prop_cbs, cbr)) {
                uint64_t propobj = cbr->cbr_ds->ds_phys->ds_props_obj;

                if (strcmp(cbr->cbr_propname, propname) != 0)
                        continue;

                /*
                 * If the property is set on this ds, then it is not
                 * inherited here; don’t call the callback.
                 */
                if (propobj && 0 == zap_contains(mos, propobj, propname))
                        continue;

                cbr->cbr_func(cbr->cbr_arg, value);
        }
        mutex_exit(&dd->dd_lock);

        za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
        for (zap_cursor_init(&zc, mos,
            dd->dd_phys->dd_child_dir_zapobj);
            zap_cursor_retrieve(&zc, za) == 0;
            zap_cursor_advance(&zc)) {
                dsl_prop_changed_notify(dp, za->za_first_integer,
                    propname, value, FALSE);
        }
        kmem_free(za, sizeof (zap_attribute_t));
        zap_cursor_fini(&zc);
        dsl_dir_rele(dd, FTAG);
}

void
dsl_prop_set_sync_impl(dsl_dataset_t *ds, const char *propname,
    zprop_source_t source, int intsz, int numints, const void *value,
    dmu_tx_t *tx)
{
        objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
        uint64_t zapobj, intval, dummy;
        int isint;
        char valbuf[32];
        const char *valstr = NULL;
        char *inheritstr;
        char *recvdstr;
        char *tbuf = NULL;
        int err;
        uint64_t version = spa_version(ds->ds_dir->dd_pool->dp_spa);

        isint = (dodefault(propname, 8, 1, &intval) == 0);

        if (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds)) {
                ASSERT(version >= SPA_VERSION_SNAP_PROPS);
                if (ds->ds_phys->ds_props_obj == 0) {
                        dmu_buf_will_dirty(ds->ds_dbuf, tx);
                        ds->ds_phys->ds_props_obj =
                            zap_create(mos,
                            DMU_OT_DSL_PROPS, DMU_OT_NONE, 0, tx);
                }
                zapobj = ds->ds_phys->ds_props_obj;
        } else {
                zapobj = ds->ds_dir->dd_phys->dd_props_zapobj;
        }

        if (version < SPA_VERSION_RECVD_PROPS) {
                zfs_prop_t prop = zfs_name_to_prop(propname);
                if (prop == ZFS_PROP_QUOTA || prop == ZFS_PROP_RESERVATION)
                        return;

                if (source & ZPROP_SRC_NONE)
                        source = ZPROP_SRC_NONE;
                else if (source & ZPROP_SRC_RECEIVED)
                        source = ZPROP_SRC_LOCAL;
        }

        inheritstr = kmem_asprintf("%s%s", propname, ZPROP_INHERIT_SUFFIX);
        recvdstr = kmem_asprintf("%s%s", propname, ZPROP_RECVD_SUFFIX);

        switch (source) {
        case ZPROP_SRC_NONE:
                /*
                 * revert to received value, if any (inherit -S)
                 * - remove propname
                 * - remove propname$inherit
                 */
                err = zap_remove(mos, zapobj, propname, tx);
                ASSERT(err == 0 || err == ENOENT);
                err = zap_remove(mos, zapobj, inheritstr, tx);
                ASSERT(err == 0 || err == ENOENT);
                break;
```

```
        case ZPROP_SRC_LOCAL:
                /*
                 * remove propname$inherit
                 * set propname -> value
                 */
                err = zap_remove(mos, zapobj, inheritstr, tx);
                ASSERT(err == 0 || err == ENOENT);
                VERIFY0(zap_update(mos, zapobj, propname,
                    intsz, numints, value, tx));
                break;
        case ZPROP_SRC_INHERITED:
                /*
                 * explicitly inherit
                 * - remove propname
                 * - set propname$inherit
                 */
                err = zap_remove(mos, zapobj, propname, tx);
                ASSERT(err == 0 || err == ENOENT);
                if (version >= SPA_VERSION_RECVD_PROPS &&
                    dsl_prop_get_int_ds(ds, ZPROP_HAS_RECVD, &dummy) == 0) {
                        dummy = 0;
                        VERIFY0(zap_update(mos, zapobj, inheritstr,
                            8, 1, &dummy, tx));
                }
                break;
        case ZPROP_SRC_RECEIVED:
                /*
                 * set propname$recvd -> value
                 */
                err = zap_update(mos, zapobj, recvdstr,
                    intsz, numints, value, tx);
                ASSERT(err == 0);
                break;
        case (ZPROP_SRC_NONE | ZPROP_SRC_LOCAL | ZPROP_SRC_RECEIVED):
                /*
                 * clear local and received settings
                 * - remove propname
                 * - remove propname$inherit
                 * - remove propname$recvd
                 */
                err = zap_remove(mos, zapobj, propname, tx);
                ASSERT(err == 0 || err == ENOENT);
                err = zap_remove(mos, zapobj, inheritstr, tx);
                ASSERT(err == 0 || err == ENOENT);
                /* FALLTHRU */
        case (ZPROP_SRC_NONE | ZPROP_SRC_RECEIVED):
                /*
                 * remove propname$recvd
                 */
                err = zap_remove(mos, zapobj, recvdstr, tx);
                ASSERT(err == 0 || err == ENOENT);
                break;
        default:
                cmn_err(CE_PANIC, "unexpected property source: %d", source);
        }

        strfree(inheritstr);
        strfree(recvdstr);

        if (isint) {
                VERIFY0(dsl_prop_get_int_ds(ds, propname, &intval));

                if (ds->ds_phys != NULL && dsl_dataset_is_snapshot(ds)) {
                        dsl_prop_cb_record_t *cbr;
                        /*
                         * It’s a snapshot; nothing can inherit this
                         * property, so just look for callbacks on this
                         * ds here.
                         */
                        mutex_enter(&ds->ds_dir->dd_lock);
                        for (cbr = list_head(&ds->ds_dir->dd_prop_cbs); cbr;
                            cbr = list_next(&ds->ds_dir->dd_prop_cbs, cbr)) {
                                if (cbr->cbr_ds == ds &&
                                    strcmp(cbr->cbr_propname, propname) == 0)
                                        cbr->cbr_func(cbr->cbr_arg, intval);
                        }
                        mutex_exit(&ds->ds_dir->dd_lock);
                } else {
                        dsl_prop_changed_notify(ds->ds_dir->dd_pool,
                            ds->ds_dir->dd_object, propname, intval, TRUE);
                }

                (void) snprintf(valbuf, sizeof (valbuf),
                    "%lld", (longlong_t)intval);
                valstr = valbuf;
        } else {
                if (source == ZPROP_SRC_LOCAL) {
                        valstr = value;
                } else {
                        tbuf = kmem_alloc(ZAP_MAXVALUELEN, KM_SLEEP);
                        if (dsl_prop_get_ds(ds, propname, 1,
                            ZAP_MAXVALUELEN, tbuf, NULL) == 0)
                                valstr = tbuf;
                }
        }

        spa_history_log_internal_ds(ds, (source == ZPROP_SRC_NONE ||
            source == ZPROP_SRC_INHERITED) ? "inherit" : "set", tx,
            "%s=%s", propname, (valstr == NULL ? "" : valstr));

        if (tbuf != NULL)
                kmem_free(tbuf, ZAP_MAXVALUELEN);
}
```

______***unchanged_portion_omitted_***