****
117465 Mon May 11 16:18:20 2015
new/usr/src/lib/libzfs/common/libzfs_dataset.c
5918 Memory leak when zfs_destroy_snaps_nvl fails
****
______unchanged_portion_omitted_

```
/*
 * 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_empty(errlist)) {
                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;
                }
        }

        nvlist_free(errlist);
#endif /* ! codereview */
        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;

        if (zfs_prop_get_int(zhp, ZFS_PROP_INCONSISTENT) == 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);
```

```c
                        }
                }
        }

        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_force;
} rollback_data_t;

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

```c
        prop_changelist_t *clp;

        /* We must destroy this clone; first unmount it */
        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);
}

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

        if (zfs_prop_get_int(zhp, ZFS_PROP_CREATETXG) > cbp->cb_create) {
                cbp->cb_error |= zfs_iter_dependents(zhp, B_FALSE,
                    rollback_destroy_dependent, cbp);

                cbp->cb_error |= zfs_destroy(zhp, B_FALSE);
        }

        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 and bookmarks more recent than the target are
 * destroyed, along with their dependents (i.e. clones).
 */
int
zfs_rollback(zfs_handle_t *zhp, zfs_handle_t *snap, boolean_t force)
{
        rollback_data_t cb = { 0 };
        int err;
        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_snapshots(zhp, rollback_destroy, &cb);
        (void) zfs_iter_bookmarks(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));
        }

        /*
         * 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.
         */
        err = lzc_rollback(zhp->zfs_name, NULL, 0);
        if (err != 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 (zhp->zfs_type != ZFS_TYPE_SNAPSHOT) {
                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 (cl != NULL)
                        (void) changelist_postfix(cl);
        } else {
                if (cl != NULL) {
                        changelist_rename(cl, zfs_get_name(zhp), target);
                        ret = changelist_postfix(cl);
                }
        }

error:
        if (parentname != NULL) {
                free(parentname);
        }
        if (zhrp != NULL) {
                zfs_close(zhrp);
        }
        if (cl != NULL) {
                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,
    boolean_t literal)
```

```
{
        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 && !literal)
                        continue;

                if (entry->pl_prop != ZPROP_INVAL) {
                        if (zfs_prop_get(zhp, entry->pl_prop,
```

```
                            buf, sizeof (buf), NULL, NULL, 0, literal) == 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), literal) == 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), literal) == 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 (0);
                }
        }

        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;
        int error;
};

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

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

        if (lzc_exists(name))
                fnvlist_add_string(ha->nvl, name, ha->tag);

        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, int cleanup_fd)
{
        int ret;
        struct holdarg ha;

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

        if (nvlist_empty(ha.nvl)) {
                char errbuf[1024];

                fnvlist_free(ha.nvl);
                ret = ENOENT;
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN,
                    "cannot hold snapshot ’%s@%s’"),
                    zhp->zfs_name, snapname);
                (void) zfs_standard_error(zhp->zfs_hdl, ret, errbuf);
                return (ret);
        }

        ret = zfs_hold_nvl(zhp, cleanup_fd, ha.nvl);
        fnvlist_free(ha.nvl);

        return (ret);
}

int
zfs_hold_nvl(zfs_handle_t *zhp, int cleanup_fd, nvlist_t *holds)
{
```

```
        int ret;
        nvlist_t *errors;
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char errbuf[1024];
        nvpair_t *elem;

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

        if (ret == 0) {
                /* There may be errors even in the success case. */
                fnvlist_free(errors);
                return (0);
        }

        if (nvlist_empty(errors)) {
                /* 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;
                default:
                        (void) zfs_standard_error(hdl,
                            fnvpair_value_int32(elem), errbuf);
                }
        }

        fnvlist_free(errors);
        return (ret);
}

static int
```

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

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

        if (lzc_get_holds(name, &existing_holds) != 0) {
                ha->error = ENOENT;
        } else if (!nvlist_exists(existing_holds, ha->tag)) {
                ha->error = ESRCH;
        } else {
                nvlist_t *torelease = fnvlist_alloc();
                fnvlist_add_boolean(torelease, ha->tag);
                fnvlist_add_nvlist(ha->nvl, name, torelease);
                fnvlist_free(torelease);
        }

        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 = NULL;
        nvpair_t *elem;
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char errbuf[1024];

        ha.nvl = fnvlist_alloc();
        ha.snapname = snapname;
        ha.tag = tag;
        ha.recursive = recursive;
        ha.error = 0;
        (void) zfs_release_one(zfs_handle_dup(zhp), &ha);

        if (nvlist_empty(ha.nvl)) {
                fnvlist_free(ha.nvl);
                ret = ha.error;
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN,
                    "cannot release hold from snapshot ’%s@%s’"),
                    zhp->zfs_name, snapname);
                if (ret == ESRCH) {
                        (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
                } else {
                        (void) zfs_standard_error(hdl, ret, errbuf);
                }
                return (ret);
        }

        ret = lzc_release(ha.nvl, &errors);
        fnvlist_free(ha.nvl);

        if (ret == 0) {
                /* There may be errors even in the success case. */
                fnvlist_free(errors);
```

```
                return (0);
        }

        if (nvlist_empty(errors)) {
                /* no hold-specific errors */
                (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)) {
                (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);
}

int
zfs_get_fsacl(zfs_handle_t *zhp, nvlist_t **nvl)
{
        zfs_cmd_t zc = { 0 };
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        int nvsz = 2048;
        void *nvbuf;
        int err = 0;
        char errbuf[1024];

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

tryagain:

        nvbuf = malloc(nvsz);
        if (nvbuf == NULL) {
                err = (zfs_error(hdl, EZFS_NOMEM, strerror(errno)));
                goto out;
        }

        zc.zc_nvlist_dst_size = nvsz;
        zc.zc_nvlist_dst = (uintptr_t)nvbuf;
        (void) strlcpy(zc.zc_name, zhp->zfs_name, ZFS_MAXNAMELEN);

        if (ioctl(hdl->libzfs_fd, ZFS_IOC_GET_FSACL, &zc) != 0) {
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot get permissions on ’%s’"),
                    zc.zc_name);
                switch (errno) {
                case ENOMEM:
                        free(nvbuf);
                        nvsz = zc.zc_nvlist_dst_size;
                        goto tryagain;

                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case ENOENT:
                        err = zfs_error(hdl, EZFS_NOENT, errbuf);
                        break;
                default:
                        err = zfs_standard_error_fmt(hdl, errno, errbuf);
                        break;
                }
        } else {
                /* success */
                int rc = nvlist_unpack(nvbuf, zc.zc_nvlist_dst_size, nvl, 0);
                if (rc) {
                        (void) snprintf(errbuf, sizeof (errbuf), dgettext(
                            TEXT_DOMAIN, "cannot get permissions on ’%s’"),
                            zc.zc_name);
                        err = zfs_standard_error_fmt(hdl, rc, errbuf);
                }
        }

        free(nvbuf);
out:
        return (err);
}

int
zfs_set_fsacl(zfs_handle_t *zhp, boolean_t un, nvlist_t *nvl)
{
        zfs_cmd_t zc = { 0 };
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char *nvbuf;
        char errbuf[1024];
        size_t nvsz;
        int err;

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

        err = nvlist_size(nvl, &nvsz, NV_ENCODE_NATIVE);
        assert(err == 0);

        nvbuf = malloc(nvsz);

        err = nvlist_pack(nvl, &nvbuf, &nvsz, NV_ENCODE_NATIVE, 0);
        assert(err == 0);

        zc.zc_nvlist_src_size = nvsz;
        zc.zc_nvlist_src = (uintptr_t)nvbuf;
```

```
        zc.zc_perm_action = un;

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

        if (zfs_ioctl(hdl, ZFS_IOC_SET_FSACL, &zc) != 0) {
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot set permissions on ’%s’"),
                    zc.zc_name);
                switch (errno) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case ENOENT:
                        err = zfs_error(hdl, EZFS_NOENT, errbuf);
                        break;
                default:
                        err = zfs_standard_error_fmt(hdl, errno, errbuf);
                        break;
                }
        }

        free(nvbuf);

        return (err);
}

int
zfs_get_holds(zfs_handle_t *zhp, nvlist_t **nvl)
{
        int err;
        char errbuf[1024];

        err = lzc_get_holds(zhp->zfs_name, nvl);

        if (err != 0) {
                libzfs_handle_t *hdl = zhp->zfs_hdl;

                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot get holds for ’%s’"),
                    zhp->zfs_name);
                switch (err) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case ENOENT:
                        err = zfs_error(hdl, EZFS_NOENT, errbuf);
                        break;
                default:
                        err = zfs_standard_error_fmt(hdl, errno, errbuf);
                        break;
                }
        }

        return (err);
}
```

```
/*
 * Convert the zvol’s volume size to an appropriate reservation.
 * Note: If this routine is updated, it is necessary to update the ZFS test
 * suite’s shell version in reservation.kshlib.
 */
uint64_t
zvol_volsize_to_reservation(uint64_t volsize, nvlist_t *props)
{
        uint64_t numdb;
        uint64_t nblocks, volblocksize;
        int ncopies;
        char *strval;

        if (nvlist_lookup_string(props,
            zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)
                ncopies = atoi(strval);
        else
                ncopies = 1;
        if (nvlist_lookup_uint64(props,
            zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
            &volblocksize) != 0)
                volblocksize = ZVOL_DEFAULT_BLOCKSIZE;
        nblocks = volsize/volblocksize;
        /* start with metadnode L0-L6 */
        numdb = 7;
        /* calculate number of indirects */
        while (nblocks > 1) {
                nblocks += DNODES_PER_LEVEL - 1;
                nblocks /= DNODES_PER_LEVEL;
                numdb += nblocks;
        }
        numdb *= MIN(SPA_DVAS_PER_BP, ncopies + 1);
        volsize *= ncopies;
        /*
         * this is exactly DN_MAX_INDBLKSHIFT when metadata isn’t
         * compressed, but in practice they compress down to about
         * 1100 bytes
         */
        numdb *= 1ULL << DN_MAX_INDBLKSHIFT;
        volsize += numdb;
        return (volsize);
}
```