```
**********************************************************
  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_

3378 /*
3379  * Destroys all the snapshots named in the nvlist.
3380  */
3381 int
3382 zfs_destroy_snaps_nvl(libzfs_handle_t *hdl, nvlist_t *snaps, boolean_t defer)
3383 {
3384         int ret;
3385         nvlist_t *errlist;

3387         ret = lzc_destroy_snaps(snaps, defer, &errlist);

3389         if (ret == 0)
3390                 return (0);

3392         if (nvlist_empty(errlist)) {
3393                 char errbuf[1024];
3394                 (void) snprintf(errbuf, sizeof (errbuf),
3395                     dgettext(TEXT_DOMAIN, "cannot destroy snapshots"));

3397                 ret = zfs_standard_error(hdl, ret, errbuf);
3398         }
3399         for (nvpair_t *pair = nvlist_next_nvpair(errlist, NULL);
3400             pair != NULL; pair = nvlist_next_nvpair(errlist, pair)) {
3401                 char errbuf[1024];
3402                 (void) snprintf(errbuf, sizeof (errbuf),
3403                     dgettext(TEXT_DOMAIN, "cannot destroy snapshot %s"),
3404                     nvpair_name(pair));

3406                 switch (fnvpair_value_int32(pair)) {
3407                 case EEXIST:
3408                         zfs_error_aux(hdl,
3409                             dgettext(TEXT_DOMAIN, "snapshot is cloned"));
3410                         ret = zfs_error(hdl, EZFS_EXISTS, errbuf);
3411                         break;
3412                 default:
3413                         ret = zfs_standard_error(hdl, errno, errbuf);
3414                         break;
3415                 }
3416         }

3418         nvlist_free(errlist);
3419 #endif /* ! codereview */
3420         return (ret);
3421 }

3423 /*
3424  * Clones the given dataset.  The target must be of the same type as the source.
3425  */
3426 int
3427 zfs_clone(zfs_handle_t *zhp, const char *target, nvlist_t *props)
3428 {
3429         char parent[ZFS_MAXNAMELEN];
3430         int ret;
3431         char errbuf[1024];
3432         libzfs_handle_t *hdl = zhp->zfs_hdl;
3433         uint64_t zoned;

3435         assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
```



```
3437         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3438             "cannot create '%s'"), target);

3440         /* validate the target/clone name */
3441         if (!zfs_validate_name(hdl, target, ZFS_TYPE_FILESYSTEM, B_TRUE))
3442                 return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));

3444         /* validate parents exist */
3445         if (check_parents(hdl, target, &zoned, B_FALSE, NULL) != 0)
3446                 return (-1);

3448         (void) parent_name(target, parent, sizeof (parent));

3450         /* do the clone */

3452         if (props) {
3453                 zfs_type_t type;
3454                 if (ZFS_IS_VOLUME(zhp)) {
3455                         type = ZFS_TYPE_VOLUME;
3456                 } else {
3457                         type = ZFS_TYPE_FILESYSTEM;
3458                 }
3459                 if ((props = zfs_valid_proplist(hdl, type, props, zoned,
3460                     zhp, errbuf)) == NULL)
3461                         return (-1);
3462         }

3464         ret = lzc_clone(target, zhp->zfs_name, props);
3465         nvlist_free(props);

3467         if (ret != 0) {
3468                 switch (errno) {

3470                 case ENOENT:
3471                         /*
3472                          * The parent doesn't exist.  We should have caught this
3473                          * above, but there may a race condition that has since
3474                          * destroyed the parent.
3475                          *
3476                          * At this point, we don't know whether it's the source
3477                          * that doesn't exist anymore, or whether the target
3478                          * dataset doesn't exist.
3479                          */
3480                         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
3481                             "no such parent '%s'"), parent);
3482                         return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));

3484                 case EXDEV:
3485                         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
3486                             "source and target pools differ"));
3487                         return (zfs_error(zhp->zfs_hdl, EZFS_CROSSTARGET,
3488                             errbuf));

3490                 default:
3491                         return (zfs_standard_error(zhp->zfs_hdl, errno,
3492                             errbuf));
3493                 }
3494         }

3496         return (ret);
3497 }

3499 /*
3500  * Promotes the given clone fs to be the clone parent.
3501  */
3502 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);
```

```
3635                        }
3636                }
3637        }

3639        nvlist_free(props);
3640        nvlist_free(errors);
3641        return (ret);
3642 }

3644 int
3645 zfs_snapshot(libzfs_handle_t *hdl, const char *path, boolean_t recursive,
3646     nvlist_t *props)
3647 {
3648        int ret;
3649        snapdata_t sd = { 0 };
3650        char fsname[ZFS_MAXNAMELEN];
3651        char *cp;
3652        zfs_handle_t *zhp;
3653        char errbuf[1024];

3655        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3656            "cannot snapshot %s"), path);

3658        if (!zfs_validate_name(hdl, path, ZFS_TYPE_SNAPSHOT, B_TRUE))
3659                return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));

3661        (void) strlcpy(fsname, path, sizeof (fsname));
3662        cp = strchr(fsname, '@');
3663        *cp = '\0';
3664        sd.sd_snapname = cp + 1;

3666        if ((zhp = zfs_open(hdl, fsname, ZFS_TYPE_FILESYSTEM |
3667            ZFS_TYPE_VOLUME)) == NULL) {
3668                return (-1);
3669        }

3671        verify(nvlist_alloc(&sd.sd_nvl, NV_UNIQUE_NAME, 0) == 0);
3672        if (recursive) {
3673                (void) zfs_snapshot_cb(zfs_handle_dup(zhp), &sd);
3674        } else {
3675                fnvlist_add_boolean(sd.sd_nvl, path);
3676        }

3678        ret = zfs_snapshot_nvl(hdl, sd.sd_nvl, props);
3679        nvlist_free(sd.sd_nvl);
3680        zfs_close(zhp);
3681        return (ret);
3682 }

3684 /*
3685  * Destroy any more recent snapshots.  We invoke this callback on any dependents
3686  * of the snapshot first.  If the 'cb_dependent' member is non-zero, then this
3687  * is a dependent and we should just destroy it without checking the transaction
3688  * group.
3689  */
3690 typedef struct rollback_data {
3691        const char      *cb_target;             /* the snapshot */
3692        uint64_t        cb_create;              /* creation time reference */
3693        boolean_t       cb_error;
3694        boolean_t       cb_force;
3695 } rollback_data_t;

3697 static int
3698 rollback_destroy_dependent(zfs_handle_t *zhp, void *data)
3699 {
3700        rollback_data_t *cbp = data;
```



```
3701        prop_changelist_t *clp;

3703        /* We must destroy this clone; first unmount it */
3704        clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
3705            cbp->cb_force ? MS_FORCE: 0);
3706        if (clp == NULL || changelist_prefix(clp) != 0) {
3707                cbp->cb_error = B_TRUE;
3708                zfs_close(zhp);
3709                return (0);
3710        }
3711        if (zfs_destroy(zhp, B_FALSE) != 0)
3712                cbp->cb_error = B_TRUE;
3713        else
3714                changelist_remove(clp, zhp->zfs_name);
3715        (void) changelist_postfix(clp);
3716        changelist_free(clp);

3718        zfs_close(zhp);
3719        return (0);
3720 }

3722 static int
3723 rollback_destroy(zfs_handle_t *zhp, void *data)
3724 {
3725        rollback_data_t *cbp = data;

3727        if (zfs_prop_get_int(zhp, ZFS_PROP_CREATETXG) > cbp->cb_create) {
3728                cbp->cb_error |= zfs_iter_dependents(zhp, B_FALSE,
3729                    rollback_destroy_dependent, cbp);

3731                cbp->cb_error |= zfs_destroy(zhp, B_FALSE);
3732        }

3734        zfs_close(zhp);
3735        return (0);
3736 }

3738 /*
3739  * Given a dataset, rollback to a specific snapshot, discarding any
3740  * data changes since then and making it the active dataset.
3741  *
3742  * Any snapshots and bookmarks more recent than the target are
3743  * destroyed, along with their dependents (i.e. clones).
3744  */
3745 int
3746 zfs_rollback(zfs_handle_t *zhp, zfs_handle_t *snap, boolean_t force)
3747 {
3748        rollback_data_t cb = { 0 };
3749        int err;
3750        boolean_t restore_resv = 0;
3751        uint64_t old_volsize, new_volsize;
3752        zfs_prop_t resv_prop;

3754        assert(zhp->zfs_type == ZFS_TYPE_FILESYSTEM ||
3755            zhp->zfs_type == ZFS_TYPE_VOLUME);

3757        /*
3758         * Destroy all recent snapshots and their dependents.
3759         */
3760        cb.cb_force = force;
3761        cb.cb_target = snap->zfs_name;
3762        cb.cb_create = zfs_prop_get_int(snap, ZFS_PROP_CREATETXG);
3763        (void) zfs_iter_snapshots(zhp, rollback_destroy, &cb);
3764        (void) zfs_iter_bookmarks(zhp, rollback_destroy, &cb);

3766        if (cb.cb_error)
```

```
3767                 return (-1);

3769         /*
3770          * Now that we have verified that the snapshot is the latest,
3771          * rollback to the given snapshot.
3772          */

3774         if (zhp->zfs_type == ZFS_TYPE_VOLUME) {
3775                 if (zfs_which_resv_prop(zhp, &resv_prop) < 0)
3776                         return (-1);
3777                 old_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
3778                 restore_resv =
3779                     (old_volsize == zfs_prop_get_int(zhp, resv_prop));
3780         }

3782         /*
3783          * We rely on zfs_iter_children() to verify that there are no
3784          * newer snapshots for the given dataset.  Therefore, we can
3785          * simply pass the name on to the ioctl() call.  There is still
3786          * an unlikely race condition where the user has taken a
3787          * snapshot since we verified that this was the most recent.
3788          */
3789         err = lzc_rollback(zhp->zfs_name, NULL, 0);
3790         if (err != 0) {
3791                 (void) zfs_standard_error_fmt(zhp->zfs_hdl, errno,
3792                     dgettext(TEXT_DOMAIN, "cannot rollback '%s'"),
3793                     zhp->zfs_name);
3794                 return (err);
3795         }

3797         /*
3798          * For volumes, if the pre-rollback volsize matched the pre-
3799          * rollback reservation and the volsize has changed then set
3800          * the reservation property to the post-rollback volsize.
3801          * Make a new handle since the rollback closed the dataset.
3802          */
3803         if ((zhp->zfs_type == ZFS_TYPE_VOLUME) &&
3804             (zhp = make_dataset_handle(zhp->zfs_hdl, zhp->zfs_name))) {
3805                 if (restore_resv) {
3806                         new_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
3807                         if (old_volsize != new_volsize)
3808                                 err = zfs_prop_set_int(zhp, resv_prop,
3809                                     new_volsize);
3810                 }
3811                 zfs_close(zhp);
3812         }
3813         return (err);
3814 }

3816 /*
3817  * Renames the given dataset.
3818  */
3819 int
3820 zfs_rename(zfs_handle_t *zhp, const char *target, boolean_t recursive,
3821     boolean_t force_unmount)
3822 {
3823         int ret;
3824         zfs_cmd_t zc = { 0 };
3825         char *delim;
3826         prop_changelist_t *cl = NULL;
3827         zfs_handle_t *zhrp = NULL;
3828         char *parentname = NULL;
3829         char parent[ZFS_MAXNAMELEN];
3830         libzfs_handle_t *hdl = zhp->zfs_hdl;
3831         char errbuf[1024];
```



```
3833         /* if we have the same exact name, just return success */
3834         if (strcmp(zhp->zfs_name, target) == 0)
3835                 return (0);

3837         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3838             "cannot rename to '%s'"), target);

3840         /*
3841          * Make sure the target name is valid
3842          */
3843         if (zhp->zfs_type == ZFS_TYPE_SNAPSHOT) {
3844                 if ((strchr(target, '@') == NULL) ||
3845                     *target == '@') {
3846                         /*
3847                          * Snapshot target name is abbreviated,
3848                          * reconstruct full dataset name
3849                          */
3850                         (void) strlcpy(parent, zhp->zfs_name,
3851                             sizeof (parent));
3852                         delim = strchr(parent, '@');
3853                         if (strchr(target, '@') == NULL)
3854                                 *(++delim) = '\0';
3855                         else
3856                                 *delim = '\0';
3857                         (void) strlcat(parent, target, sizeof (parent));
3858                         target = parent;
3859                 } else {
3860                         /*
3861                          * Make sure we're renaming within the same dataset.
3862                          */
3863                         delim = strchr(target, '@');
3864                         if (strncmp(zhp->zfs_name, target, delim - target)
3865                             != 0 || zhp->zfs_name[delim - target] != '@') {
3866                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3867                                     "snapshots must be part of same "
3868                                     "dataset"));
3869                                 return (zfs_error(hdl, EZFS_CROSSTARGET,
3870                                     errbuf));
3871                         }
3872                 }
3873                 if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
3874                         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3875         } else {
3876                 if (recursive) {
3877                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3878                             "recursive rename must be a snapshot"));
3879                         return (zfs_error(hdl, EZFS_BADTYPE, errbuf));
3880                 }

3882                 if (!zfs_validate_name(hdl, target, zhp->zfs_type, B_TRUE))
3883                         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));

3885                 /* validate parents */
3886                 if (check_parents(hdl, target, NULL, B_FALSE, NULL) != 0)
3887                         return (-1);

3889                 /* make sure we're in the same pool */
3890                 verify((delim = strchr(target, '/')) != NULL);
3891                 if (strncmp(zhp->zfs_name, target, delim - target) != 0 ||
3892                     zhp->zfs_name[delim - target] != '/') {
3893                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3894                             "datasets must be within same pool"));
3895                         return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));
3896                 }

3898                 /* new name cannot be a child of the current dataset name */
```

```
3899                 if (is_descendant(zhp->zfs_name, target)) {
3900                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3901                             "New dataset name cannot be a descendant of "
3902                             "current dataset name"));
3903                         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
3904                 }
3905         }

3907         (void) snprintf(errbuf, sizeof (errbuf),
3908             dgettext(TEXT_DOMAIN, "cannot rename '%s'"), zhp->zfs_name);

3910         if (getzoneid() == GLOBAL_ZONEID &&
3911             zfs_prop_get_int(zhp, ZFS_PROP_ZONED)) {
3912                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3913                     "dataset is used in a non-global zone"));
3914                 return (zfs_error(hdl, EZFS_ZONED, errbuf));
3915         }

3917         if (recursive) {
3918                 parentname = zfs_strdup(zhp->zfs_hdl, zhp->zfs_name);
3919                 if (parentname == NULL) {
3920                         ret = -1;
3921                         goto error;
3922                 }
3923                 delim = strchr(parentname, '@');
3924                 *delim = '\0';
3925                 zhrp = zfs_open(zhp->zfs_hdl, parentname, ZFS_TYPE_DATASET);
3926                 if (zhrp == NULL) {
3927                         ret = -1;
3928                         goto error;
3929                 }
3930         } else if (zhp->zfs_type != ZFS_TYPE_SNAPSHOT) {
3931                 if ((cl = changelist_gather(zhp, ZFS_PROP_NAME, 0,
3932                     force_unmount ? MS_FORCE : 0)) == NULL)
3933                         return (-1);

3935                 if (changelist_haszonedchild(cl)) {
3936                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3937                             "child dataset with inherited mountpoint is used "
3938                             "in a non-global zone"));
3939                         (void) zfs_error(hdl, EZFS_ZONED, errbuf);
3940                         goto error;
3941                 }

3943                 if ((ret = changelist_prefix(cl)) != 0)
3944                         goto error;
3945         }

3947         if (ZFS_IS_VOLUME(zhp))
3948                 zc.zc_objset_type = DMU_OST_ZVOL;
3949         else
3950                 zc.zc_objset_type = DMU_OST_ZFS;

3952         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
3953         (void) strlcpy(zc.zc_value, target, sizeof (zc.zc_value));

3955         zc.zc_cookie = recursive;

3957         if ((ret = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_RENAME, &zc)) != 0) {
3958                 /*
3959                  * if it was recursive, the one that actually failed will
3960                  * be in zc.zc_name
3961                  */
3962                 (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3963                     "cannot rename '%s'"), zc.zc_name);
```



```
3965                 if (recursive && errno == EEXIST) {
3966                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3967                             "a child dataset already has a snapshot "
3968                             "with the new name"));
3969                         (void) zfs_error(hdl, EZFS_EXISTS, errbuf);
3970                 } else {
3971                         (void) zfs_standard_error(zhp->zfs_hdl, errno, errbuf);
3972                 }

3974                 /*
3975                  * On failure, we still want to remount any filesystems that
3976                  * were previously mounted, so we don't alter the system state.
3977                  */
3978                 if (cl != NULL)
3979                         (void) changelist_postfix(cl);
3980         } else {
3981                 if (cl != NULL) {
3982                         changelist_rename(cl, zfs_get_name(zhp), target);
3983                         ret = changelist_postfix(cl);
3984                 }
3985         }

3987 error:
3988         if (parentname != NULL) {
3989                 free(parentname);
3990         }
3991         if (zhrp != NULL) {
3992                 zfs_close(zhrp);
3993         }
3994         if (cl != NULL) {
3995                 changelist_free(cl);
3996         }
3997         return (ret);
3998 }

4000 nvlist_t *
4001 zfs_get_user_props(zfs_handle_t *zhp)
4002 {
4003         return (zhp->zfs_user_props);
4004 }

4006 nvlist_t *
4007 zfs_get_recvd_props(zfs_handle_t *zhp)
4008 {
4009         if (zhp->zfs_recvd_props == NULL)
4010                 if (get_recvd_props_ioctl(zhp) != 0)
4011                         return (NULL);
4012         return (zhp->zfs_recvd_props);
4013 }

4015 /*
4016  * This function is used by 'zfs list' to determine the exact set of columns to
4017  * display, and their maximum widths.  This does two main things:
4018  *
4019  *      - If this is a list of all properties, then expand the list to include
4020  *        all native properties, and set a flag so that for each dataset we look
4021  *        for new unique user properties and add them to the list.
4022  *
4023  *      - For non fixed-width properties, keep track of the maximum width seen
4024  *        so that we can size the column appropriately. If the user has
4025  *        requested received property values, we also need to compute the width
4026  *        of the RECEIVED column.
4027  */
4028 int
4029 zfs_expand_proplist(zfs_handle_t *zhp, zprop_list_t **plp, boolean_t received,
4030     boolean_t literal)
```

```
4031 {
4032         libzfs_handle_t *hdl = zhp->zfs_hdl;
4033         zprop_list_t *entry;
4034         zprop_list_t **last, **start;
4035         nvlist_t *userprops, *propval;
4036         nvpair_t *elem;
4037         char *strval;
4038         char buf[ZFS_MAXPROPLEN];

4040         if (zprop_expand_list(hdl, plp, ZFS_TYPE_DATASET) != 0)
4041                 return (-1);

4043         userprops = zfs_get_user_props(zhp);

4045         entry = *plp;
4046         if (entry->pl_all && nvlist_next_nvpair(userprops, NULL) != NULL) {
4047                 /*
4048                  * Go through and add any user properties as necessary.  We
4049                  * start by incrementing our list pointer to the first
4050                  * non-native property.
4051                  */
4052                 start = plp;
4053                 while (*start != NULL) {
4054                         if ((*start)->pl_prop == ZPROP_INVAL)
4055                                 break;
4056                         start = &(*start)->pl_next;
4057                 }

4059                 elem = NULL;
4060                 while ((elem = nvlist_next_nvpair(userprops, elem)) != NULL) {
4061                         /*
4062                          * See if we've already found this property in our list.
4063                          */
4064                         for (last = start; *last != NULL;
4065                             last = &(*last)->pl_next) {
4066                                 if (strcmp((*last)->pl_user_prop,
4067                                     nvpair_name(elem)) == 0)
4068                                         break;
4069                         }

4071                         if (*last == NULL) {
4072                                 if ((entry = zfs_alloc(hdl,
4073                                     sizeof (zprop_list_t))) == NULL ||
4074                                     ((entry->pl_user_prop = zfs_strdup(hdl,
4075                                     nvpair_name(elem)))) == NULL) {
4076                                         free(entry);
4077                                         return (-1);
4078                                 }

4080                                 entry->pl_prop = ZPROP_INVAL;
4081                                 entry->pl_width = strlen(nvpair_name(elem));
4082                                 entry->pl_all = B_TRUE;
4083                                 *last = entry;
4084                         }
4085                 }
4086         }

4088         /*
4089          * Now go through and check the width of any non-fixed columns
4090          */
4091         for (entry = *plp; entry != NULL; entry = entry->pl_next) {
4092                 if (entry->pl_fixed && !literal)
4093                         continue;

4095                 if (entry->pl_prop != ZPROP_INVAL) {
4096                         if (zfs_prop_get(zhp, entry->pl_prop,
```



```
4097                             buf, sizeof (buf), NULL, NULL, 0, literal) == 0) {
4098                                 if (strlen(buf) > entry->pl_width)
4099                                         entry->pl_width = strlen(buf);
4100                         }
4101                         if (received && zfs_prop_get_recvd(zhp,
4102                             zfs_prop_to_name(entry->pl_prop),
4103                             buf, sizeof (buf), literal) == 0)
4104                                 if (strlen(buf) > entry->pl_recvd_width)
4105                                         entry->pl_recvd_width = strlen(buf);
4106                 } else {
4107                         if (nvlist_lookup_nvlist(userprops, entry->pl_user_prop,
4108                             &propval) == 0) {
4109                                 verify(nvlist_lookup_string(propval,
4110                                     ZPROP_VALUE, &strval) == 0);
4111                                 if (strlen(strval) > entry->pl_width)
4112                                         entry->pl_width = strlen(strval);
4113                         }
4114                         if (received && zfs_prop_get_recvd(zhp,
4115                             entry->pl_user_prop,
4116                             buf, sizeof (buf), literal) == 0)
4117                                 if (strlen(buf) > entry->pl_recvd_width)
4118                                         entry->pl_recvd_width = strlen(buf);
4119                 }
4120         }

4122         return (0);
4123 }

4125 int
4126 zfs_deleg_share_nfs(libzfs_handle_t *hdl, char *dataset, char *path,
4127     char *resource, void *export, void *sharetab,
4128     int sharemax, zfs_share_op_t operation)
4129 {
4130         zfs_cmd_t zc = { 0 };
4131         int error;

4133         (void) strlcpy(zc.zc_name, dataset, sizeof (zc.zc_name));
4134         (void) strlcpy(zc.zc_value, path, sizeof (zc.zc_value));
4135         if (resource)
4136                 (void) strlcpy(zc.zc_string, resource, sizeof (zc.zc_string));
4137         zc.zc_share.z_sharedata = (uint64_t)(uintptr_t)sharetab;
4138         zc.zc_share.z_exportdata = (uint64_t)(uintptr_t)export;
4139         zc.zc_share.z_sharetype = operation;
4140         zc.zc_share.z_sharemax = sharemax;
4141         error = ioctl(hdl->libzfs_fd, ZFS_IOC_SHARE, &zc);
4142         return (error);
4143 }

4145 void
4146 zfs_prune_proplist(zfs_handle_t *zhp, uint8_t *props)
4147 {
4148         nvpair_t *curr;

4150         /*
4151          * Keep a reference to the props-table against which we prune the
4152          * properties.
4153          */
4154         zhp->zfs_props_table = props;

4156         curr = nvlist_next_nvpair(zhp->zfs_props, NULL);

4158         while (curr) {
4159                 zfs_prop_t zfs_prop = zfs_name_to_prop(nvpair_name(curr));
4160                 nvpair_t *next = nvlist_next_nvpair(zhp->zfs_props, curr);

4162                 /*
```

```
4163                 * User properties will result in ZPROP_INVAL, and since we
4164                 * only know how to prune standard ZFS properties, we always
4165                 * leave these in the list.  This can also happen if we
4166                 * encounter an unknown DSL property (when running older
4167                 * software, for example).
4168                 */
4169                if (zfs_prop != ZPROP_INVAL && props[zfs_prop] == B_FALSE)
4170                        (void) nvlist_remove(zhp->zfs_props,
4171                            nvpair_name(curr), nvpair_type(curr));
4172                curr = next;
4173        }
4174 }

4176 static int
4177 zfs_smb_acl_mgmt(libzfs_handle_t *hdl, char *dataset, char *path,
4178     zfs_smb_acl_op_t cmd, char *resource1, char *resource2)
4179 {
4180        zfs_cmd_t zc = { 0 };
4181        nvlist_t *nvlist = NULL;
4182        int error;

4184        (void) strlcpy(zc.zc_name, dataset, sizeof (zc.zc_name));
4185        (void) strlcpy(zc.zc_value, path, sizeof (zc.zc_value));
4186        zc.zc_cookie = (uint64_t)cmd;

4188        if (cmd == ZFS_SMB_ACL_RENAME) {
4189                if (nvlist_alloc(&nvlist, NV_UNIQUE_NAME, 0) != 0) {
4190                        (void) no_memory(hdl);
4191                        return (0);
4192                }
4193        }

4195        switch (cmd) {
4196        case ZFS_SMB_ACL_ADD:
4197        case ZFS_SMB_ACL_REMOVE:
4198                (void) strlcpy(zc.zc_string, resource1, sizeof (zc.zc_string));
4199                break;
4200        case ZFS_SMB_ACL_RENAME:
4201                if (nvlist_add_string(nvlist, ZFS_SMB_ACL_SRC,
4202                    resource1) != 0) {
4203                                (void) no_memory(hdl);
4204                                return (-1);
4205                }
4206                if (nvlist_add_string(nvlist, ZFS_SMB_ACL_TARGET,
4207                    resource2) != 0) {
4208                                (void) no_memory(hdl);
4209                                return (-1);
4210                }
4211                if (zcmd_write_src_nvlist(hdl, &zc, nvlist) != 0) {
4212                        nvlist_free(nvlist);
4213                        return (-1);
4214                }
4215                break;
4216        case ZFS_SMB_ACL_PURGE:
4217                break;
4218        default:
4219                return (-1);
4220        }
4221        error = ioctl(hdl->libzfs_fd, ZFS_IOC_SMB_ACL, &zc);
4222        if (nvlist)
4223                nvlist_free(nvlist);
4224        return (error);
4225 }

4227 int
4228 zfs_smb_acl_add(libzfs_handle_t *hdl, char *dataset,
```



```
4229     char *path, char *resource)
4230 {
4231        return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_ADD,
4232            resource, NULL));
4233 }

4235 int
4236 zfs_smb_acl_remove(libzfs_handle_t *hdl, char *dataset,
4237     char *path, char *resource)
4238 {
4239        return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_REMOVE,
4240            resource, NULL));
4241 }

4243 int
4244 zfs_smb_acl_purge(libzfs_handle_t *hdl, char *dataset, char *path)
4245 {
4246        return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_PURGE,
4247            NULL, NULL));
4248 }

4250 int
4251 zfs_smb_acl_rename(libzfs_handle_t *hdl, char *dataset, char *path,
4252     char *oldname, char *newname)
4253 {
4254        return (zfs_smb_acl_mgmt(hdl, dataset, path, ZFS_SMB_ACL_RENAME,
4255            oldname, newname));
4256 }

4258 int
4259 zfs_userspace(zfs_handle_t *zhp, zfs_userquota_prop_t type,
4260     zfs_userspace_cb_t func, void *arg)
4261 {
4262        zfs_cmd_t zc = { 0 };
4263        zfs_useracct_t buf[100];
4264        libzfs_handle_t *hdl = zhp->zfs_hdl;
4265        int ret;

4267        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

4269        zc.zc_objset_type = type;
4270        zc.zc_nvlist_dst = (uintptr_t)buf;

4272        for (;;) {
4273                zfs_useracct_t *zua = buf;

4275                zc.zc_nvlist_dst_size = sizeof (buf);
4276                if (zfs_ioctl(hdl, ZFS_IOC_USERSPACE_MANY, &zc) != 0) {
4277                        char errbuf[1024];

4279                        (void) snprintf(errbuf, sizeof (errbuf),
4280                            dgettext(TEXT_DOMAIN,
4281                            "cannot get used/quota for %s"), zc.zc_name);
4282                        return (zfs_standard_error_fmt(hdl, errno, errbuf));
4283                }
4284                if (zc.zc_nvlist_dst_size == 0)
4285                        break;

4287                while (zc.zc_nvlist_dst_size > 0) {
4288                        if ((ret = func(arg, zua->zu_domain, zua->zu_rid,
4289                            zua->zu_space)) != 0)
4290                                return (ret);
4291                        zua++;
4292                        zc.zc_nvlist_dst_size -= sizeof (zfs_useracct_t);
4293                }
4294        }
```

```
4296         return (0);
4297 }

4299 struct holdarg {
4300         nvlist_t *nvl;
4301         const char *snapname;
4302         const char *tag;
4303         boolean_t recursive;
4304         int error;
4305 };

4307 static int
4308 zfs_hold_one(zfs_handle_t *zhp, void *arg)
4309 {
4310         struct holdarg *ha = arg;
4311         char name[ZFS_MAXNAMELEN];
4312         int rv = 0;

4314         (void) snprintf(name, sizeof (name),
4315             "%s@%s", zhp->zfs_name, ha->snapname);

4317         if (lzc_exists(name))
4318                 fnvlist_add_string(ha->nvl, name, ha->tag);

4320         if (ha->recursive)
4321                 rv = zfs_iter_filesystems(zhp, zfs_hold_one, ha);
4322         zfs_close(zhp);
4323         return (rv);
4324 }

4326 int
4327 zfs_hold(zfs_handle_t *zhp, const char *snapname, const char *tag,
4328     boolean_t recursive, int cleanup_fd)
4329 {
4330         int ret;
4331         struct holdarg ha;

4333         ha.nvl = fnvlist_alloc();
4334         ha.snapname = snapname;
4335         ha.tag = tag;
4336         ha.recursive = recursive;
4337         (void) zfs_hold_one(zfs_handle_dup(zhp), &ha);

4339         if (nvlist_empty(ha.nvl)) {
4340                 char errbuf[1024];

4342                 fnvlist_free(ha.nvl);
4343                 ret = ENOENT;
4344                 (void) snprintf(errbuf, sizeof (errbuf),
4345                     dgettext(TEXT_DOMAIN,
4346                     "cannot hold snapshot '%s@%s'"),
4347                     zhp->zfs_name, snapname);
4348                 (void) zfs_standard_error(zhp->zfs_hdl, ret, errbuf);
4349                 return (ret);
4350         }

4352         ret = zfs_hold_nvl(zhp, cleanup_fd, ha.nvl);
4353         fnvlist_free(ha.nvl);

4355         return (ret);
4356 }

4358 int
4359 zfs_hold_nvl(zfs_handle_t *zhp, int cleanup_fd, nvlist_t *holds)
4360 {
```



```
4361         int ret;
4362         nvlist_t *errors;
4363         libzfs_handle_t *hdl = zhp->zfs_hdl;
4364         char errbuf[1024];
4365         nvpair_t *elem;

4367         errors = NULL;
4368         ret = lzc_hold(holds, cleanup_fd, &errors);

4370         if (ret == 0) {
4371                 /* There may be errors even in the success case. */
4372                 fnvlist_free(errors);
4373                 return (0);
4374         }

4376         if (nvlist_empty(errors)) {
4377                 /* no hold-specific errors */
4378                 (void) snprintf(errbuf, sizeof (errbuf),
4379                     dgettext(TEXT_DOMAIN, "cannot hold"));
4380                 switch (ret) {
4381                 case ENOTSUP:
4382                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4383                             "pool must be upgraded"));
4384                         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4385                         break;
4386                 case EINVAL:
4387                         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4388                         break;
4389                 default:
4390                         (void) zfs_standard_error(hdl, ret, errbuf);
4391                 }
4392         }

4394         for (elem = nvlist_next_nvpair(errors, NULL);
4395             elem != NULL;
4396             elem = nvlist_next_nvpair(errors, elem)) {
4397                 (void) snprintf(errbuf, sizeof (errbuf),
4398                     dgettext(TEXT_DOMAIN,
4399                     "cannot hold snapshot '%s'"), nvpair_name(elem));
4400                 switch (fnvpair_value_int32(elem)) {
4401                 case E2BIG:
4402                         /*
4403                          * Temporary tags wind up having the ds object id
4404                          * prepended. So even if we passed the length check
4405                          * above, it's still possible for the tag to wind
4406                          * up being slightly too long.
4407                          */
4408                         (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
4409                         break;
4410                 case EINVAL:
4411                         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4412                         break;
4413                 case EEXIST:
4414                         (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
4415                         break;
4416                 default:
4417                         (void) zfs_standard_error(hdl,
4418                             fnvpair_value_int32(elem), errbuf);
4419                 }
4420         }

4422         fnvlist_free(errors);
4423         return (ret);
4424 }

4426 static int
```

```
4427 zfs_release_one(zfs_handle_t *zhp, void *arg)
4428 {
4429         struct holdarg *ha = arg;
4430         char name[ZFS_MAXNAMELEN];
4431         int rv = 0;
4432         nvlist_t *existing_holds;
4433
4434         (void) snprintf(name, sizeof (name),
4435             "%s@%s", zhp->zfs_name, ha->snapname);
4436
4437         if (lzc_get_holds(name, &existing_holds) != 0) {
4438                 ha->error = ENOENT;
4439         } else if (!nvlist_exists(existing_holds, ha->tag)) {
4440                 ha->error = ESRCH;
4441         } else {
4442                 nvlist_t *torelease = fnvlist_alloc();
4443                 fnvlist_add_boolean(torelease, ha->tag);
4444                 fnvlist_add_nvlist(ha->nvl, name, torelease);
4445                 fnvlist_free(torelease);
4446         }
4447
4448         if (ha->recursive)
4449                 rv = zfs_iter_filesystems(zhp, zfs_release_one, ha);
4450         zfs_close(zhp);
4451         return (rv);
4452 }
4453
4454 int
4455 zfs_release(zfs_handle_t *zhp, const char *snapname, const char *tag,
4456     boolean_t recursive)
4457 {
4458         int ret;
4459         struct holdarg ha;
4460         nvlist_t *errors = NULL;
4461         nvpair_t *elem;
4462         libzfs_handle_t *hdl = zhp->zfs_hdl;
4463         char errbuf[1024];
4464
4465         ha.nvl = fnvlist_alloc();
4466         ha.snapname = snapname;
4467         ha.tag = tag;
4468         ha.recursive = recursive;
4469         ha.error = 0;
4470         (void) zfs_release_one(zfs_handle_dup(zhp), &ha);
4471
4472         if (nvlist_empty(ha.nvl)) {
4473                 fnvlist_free(ha.nvl);
4474                 ret = ha.error;
4475                 (void) snprintf(errbuf, sizeof (errbuf),
4476                     dgettext(TEXT_DOMAIN,
4477                     "cannot release hold from snapshot '%s@%s'"),
4478                     zhp->zfs_name, snapname);
4479                 if (ret == ESRCH) {
4480                         (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
4481                 } else {
4482                         (void) zfs_standard_error(hdl, ret, errbuf);
4483                 }
4484                 return (ret);
4485         }
4486
4487         ret = lzc_release(ha.nvl, &errors);
4488         fnvlist_free(ha.nvl);
4489
4490         if (ret == 0) {
4491                 /* There may be errors even in the success case. */
4492                 fnvlist_free(errors);
```



```
4493                 return (0);
4494         }
4495
4496         if (nvlist_empty(errors)) {
4497                 /* no hold-specific errors */
4498                 (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
4499                     "cannot release"));
4500                 switch (errno) {
4501                 case ENOTSUP:
4502                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4503                             "pool must be upgraded"));
4504                         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4505                         break;
4506                 default:
4507                         (void) zfs_standard_error_fmt(hdl, errno, errbuf);
4508                 }
4509         }
4510
4511         for (elem = nvlist_next_nvpair(errors, NULL);
4512             elem != NULL;
4513             elem = nvlist_next_nvpair(errors, elem)) {
4514                 (void) snprintf(errbuf, sizeof (errbuf),
4515                     dgettext(TEXT_DOMAIN,
4516                     "cannot release hold from snapshot '%s'"),
4517                     nvpair_name(elem));
4518                 switch (fnvpair_value_int32(elem)) {
4519                 case ESRCH:
4520                         (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
4521                         break;
4522                 case EINVAL:
4523                         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4524                         break;
4525                 default:
4526                         (void) zfs_standard_error_fmt(hdl,
4527                             fnvpair_value_int32(elem), errbuf);
4528                 }
4529         }
4530
4531         fnvlist_free(errors);
4532         return (ret);
4533 }
4534
4535 int
4536 zfs_get_fsacl(zfs_handle_t *zhp, nvlist_t **nvl)
4537 {
4538         zfs_cmd_t zc = { 0 };
4539         libzfs_handle_t *hdl = zhp->zfs_hdl;
4540         int nvsz = 2048;
4541         void *nvbuf;
4542         int err = 0;
4543         char errbuf[1024];
4544
4545         assert(zhp->zfs_type == ZFS_TYPE_VOLUME ||
4546             zhp->zfs_type == ZFS_TYPE_FILESYSTEM);
4547
4548 tryagain:
4549
4550         nvbuf = malloc(nvsz);
4551         if (nvbuf == NULL) {
4552                 err = (zfs_error(hdl, EZFS_NOMEM, strerror(errno)));
4553                 goto out;
4554         }
4555
4556         zc.zc_nvlist_dst_size = nvsz;
4557         zc.zc_nvlist_dst = (uintptr_t)nvbuf;
```

```
4559         (void) strlcpy(zc.zc_name, zhp->zfs_name, ZFS_MAXNAMELEN);

4561         if (ioctl(hdl->libzfs_fd, ZFS_IOC_GET_FSACL, &zc) != 0) {
4562                 (void) snprintf(errbuf, sizeof (errbuf),
4563                     dgettext(TEXT_DOMAIN, "cannot get permissions on '%s'"),
4564                     zc.zc_name);
4565                 switch (errno) {
4566                 case ENOMEM:
4567                         free(nvbuf);
4568                         nvsz = zc.zc_nvlist_dst_size;
4569                         goto tryagain;

4571                 case ENOTSUP:
4572                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4573                             "pool must be upgraded"));
4574                         err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4575                         break;
4576                 case EINVAL:
4577                         err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4578                         break;
4579                 case ENOENT:
4580                         err = zfs_error(hdl, EZFS_NOENT, errbuf);
4581                         break;
4582                 default:
4583                         err = zfs_standard_error_fmt(hdl, errno, errbuf);
4584                         break;
4585                 }
4586         } else {
4587                 /* success */
4588                 int rc = nvlist_unpack(nvbuf, zc.zc_nvlist_dst_size, nvl, 0);
4589                 if (rc) {
4590                         (void) snprintf(errbuf, sizeof (errbuf), dgettext(
4591                             TEXT_DOMAIN, "cannot get permissions on '%s'"),
4592                             zc.zc_name);
4593                         err = zfs_standard_error_fmt(hdl, rc, errbuf);
4594                 }
4595         }

4597         free(nvbuf);
4598 out:
4599         return (err);
4600 }

4602 int
4603 zfs_set_fsacl(zfs_handle_t *zhp, boolean_t un, nvlist_t *nvl)
4604 {
4605         zfs_cmd_t zc = { 0 };
4606         libzfs_handle_t *hdl = zhp->zfs_hdl;
4607         char *nvbuf;
4608         char errbuf[1024];
4609         size_t nvsz;
4610         int err;

4612         assert(zhp->zfs_type == ZFS_TYPE_VOLUME ||
4613             zhp->zfs_type == ZFS_TYPE_FILESYSTEM);

4615         err = nvlist_size(nvl, &nvsz, NV_ENCODE_NATIVE);
4616         assert(err == 0);

4618         nvbuf = malloc(nvsz);

4620         err = nvlist_pack(nvl, &nvbuf, &nvsz, NV_ENCODE_NATIVE, 0);
4621         assert(err == 0);

4623         zc.zc_nvlist_src_size = nvsz;
4624         zc.zc_nvlist_src = (uintptr_t)nvbuf;
```



```
4625         zc.zc_perm_action = un;

4627         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

4629         if (zfs_ioctl(hdl, ZFS_IOC_SET_FSACL, &zc) != 0) {
4630                 (void) snprintf(errbuf, sizeof (errbuf),
4631                     dgettext(TEXT_DOMAIN, "cannot set permissions on '%s'"),
4632                     zc.zc_name);
4633                 switch (errno) {
4634                 case ENOTSUP:
4635                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4636                             "pool must be upgraded"));
4637                         err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4638                         break;
4639                 case EINVAL:
4640                         err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4641                         break;
4642                 case ENOENT:
4643                         err = zfs_error(hdl, EZFS_NOENT, errbuf);
4644                         break;
4645                 default:
4646                         err = zfs_standard_error_fmt(hdl, errno, errbuf);
4647                         break;
4648                 }
4649         }

4651         free(nvbuf);

4653         return (err);
4654 }

4656 int
4657 zfs_get_holds(zfs_handle_t *zhp, nvlist_t **nvl)
4658 {
4659         int err;
4660         char errbuf[1024];

4662         err = lzc_get_holds(zhp->zfs_name, nvl);

4664         if (err != 0) {
4665                 libzfs_handle_t *hdl = zhp->zfs_hdl;

4667                 (void) snprintf(errbuf, sizeof (errbuf),
4668                     dgettext(TEXT_DOMAIN, "cannot get holds for '%s'"),
4669                     zhp->zfs_name);
4670                 switch (err) {
4671                 case ENOTSUP:
4672                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4673                             "pool must be upgraded"));
4674                         err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4675                         break;
4676                 case EINVAL:
4677                         err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4678                         break;
4679                 case ENOENT:
4680                         err = zfs_error(hdl, EZFS_NOENT, errbuf);
4681                         break;
4682                 default:
4683                         err = zfs_standard_error_fmt(hdl, errno, errbuf);
4684                         break;
4685                 }
4686         }

4688         return (err);
4689 }
```

```
4691 /*
4692  * Convert the zvol's volume size to an appropriate reservation.
4693  * Note: If this routine is updated, it is necessary to update the ZFS test
4694  * suite's shell version in reservation.kshlib.
4695  */
4696 uint64_t
4697 zvol_volsize_to_reservation(uint64_t volsize, nvlist_t *props)
4698 {
4699         uint64_t numdb;
4700         uint64_t nblocks, volblocksize;
4701         int ncopies;
4702         char *strval;

4704         if (nvlist_lookup_string(props,
4705             zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)
4706                 ncopies = atoi(strval);
4707         else
4708                 ncopies = 1;
4709         if (nvlist_lookup_uint64(props,
4710             zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
4711             &volblocksize) != 0)
4712                 volblocksize = ZVOL_DEFAULT_BLOCKSIZE;
4713         nblocks = volsize/volblocksize;
4714         /* start with metadnode L0-L6 */
4715         numdb = 7;
4716         /* calculate number of indirects */
4717         while (nblocks > 1) {
4718                 nblocks += DNODES_PER_LEVEL - 1;
4719                 nblocks /= DNODES_PER_LEVEL;
4720                 numdb += nblocks;
4721         }
4722         numdb *= MIN(SPA_DVAS_PER_BP, ncopies + 1);
4723         volsize *= ncopies;
4724         /*
4725          * this is exactly DN_MAX_INDBLKSHIFT when metadata isn't
4726          * compressed, but in practice they compress down to about
4727          * 1100 bytes
4728          */
4729         numdb *= 1ULL << DN_MAX_INDBLKSHIFT;
4730         volsize += numdb;
4731         return (volsize);
4732 }
```