****
162037 Wed Oct 16 18:29:02 2013
new/usr/src/cmd/zfs/zfs_main.c
4012 Upper limit of zfs set bounds check for refreservation on volumes is too lo
****
______***unchanged_portion_omitted_***

```
/*
 * zfs create [-p] [-o prop=value] ... fs
 * zfs create [-ps] [-b blocksize] [-o prop=value] ... -V vol size
 *
 * Create a new dataset.  This command can be used to create filesystems
 * and volumes.  Snapshot creation is handled by 'zfs snapshot'.
 * For volumes, the user must specify a size to be used.
 *
 * The '-s' flag applies only to volumes, and indicates that we should not try
 * to set the reservation for this volume.  By default we set a reservation
 * equal to the size for any volume.  For pools with SPA_VERSION >=
 * SPA_VERSION_REFRESERVATION, we set a refreservation instead.
 *
 * The '-p' flag creates all the non-existing ancestors of the target first.
 */
static int
zfs_do_create(int argc, char **argv)
{
        zfs_type_t type = ZFS_TYPE_FILESYSTEM;
        zfs_handle_t *zhp = NULL;
        uint64_t volsize;
        int c;
        boolean_t noreserve = B_FALSE;
        boolean_t bflag = B_FALSE;
        boolean_t parents = B_FALSE;
        int ret = 1;
        nvlist_t *props;
        uint64_t intval;
        int canmount = ZFS_CANMOUNT_OFF;

        if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0)
                nomem();

        /* check options */
        while ((c = getopt(argc, argv, ":V:b:so:p")) != -1) {
                switch (c) {
                case 'V':
                        type = ZFS_TYPE_VOLUME;
                        if (zfs_nicestrtonum(g_zfs, optarg, &intval) != 0) {
                                (void) fprintf(stderr, gettext("bad volume "
                                    "size '%s': %s\n"), optarg,
                                    libzfs_error_description(g_zfs));
                                goto error;
                        }

                        if (nvlist_add_uint64(props,
                            zfs_prop_to_name(ZFS_PROP_VOLSIZE), intval) != 0)
                                nomem();
                        volsize = intval;
                        break;
                case 'p':
                        parents = B_TRUE;
                        break;
                case 'b':
                        bflag = B_TRUE;
                        if (zfs_nicestrtonum(g_zfs, optarg, &intval) != 0) {
                                (void) fprintf(stderr, gettext("bad volume "
                                    "block size '%s': %s\n"), optarg,
                                    libzfs_error_description(g_zfs));
```

```
                                goto error;
                        }

                        if (nvlist_add_uint64(props,
                            zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
                            intval) != 0)
                                nomem();
                        break;
                case 'o':
                        if (parseprop(props))
                                goto error;
                        break;
                case 's':
                        noreserve = B_TRUE;
                        break;
                case ':':
                        (void) fprintf(stderr, gettext("missing size "
                            "argument\n"));
                        goto badusage;
                case '?':
                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
                            optopt);
                        goto badusage;
                }
        }

        if ((bflag || noreserve) && type != ZFS_TYPE_VOLUME) {
                (void) fprintf(stderr, gettext("'-s' and '-b' can only be "
                    "used when creating a volume\n"));
                goto badusage;
        }

        argc -= optind;
        argv += optind;

        /* check number of arguments */
        if (argc == 0) {
                (void) fprintf(stderr, gettext("missing %s argument\n"),
                    zfs_type_to_name(type));
                goto badusage;
        }
        if (argc > 1) {
                (void) fprintf(stderr, gettext("too many arguments\n"));
                goto badusage;
        }

        if (type == ZFS_TYPE_VOLUME && !noreserve) {
                zpool_handle_t *zpool_handle;
                nvlist_t *real_props;
                uint64_t spa_version;
                char *p;
                zfs_prop_t resv_prop;
                char *strval;
                char msg[1024];
                uint64_t volblocksize;
                int ncopies;

                if (p = strchr(argv[0], '/'))
                        *p = '\0';
                zpool_handle = zpool_open(g_zfs, argv[0]);
                if (p != NULL)
                        *p = '/';
                if (zpool_handle == NULL)
                        goto error;
                spa_version = zpool_get_prop_int(zpool_handle,
                    ZPOOL_PROP_VERSION, NULL);
                zpool_close(zpool_handle);
                if (spa_version >= SPA_VERSION_REFRESERVATION)
                        resv_prop = ZFS_PROP_REFRESERVATION;
                else
                        resv_prop = ZFS_PROP_RESERVATION;

                (void) snprintf(msg, sizeof (msg),
                    gettext("cannot create '%s'"), argv[0]);
                if (props && (real_props = zfs_valid_proplist(g_zfs, type,
                    props, 0, NULL, msg)) == NULL)
                        goto error;

                if (nvlist_lookup_string(real_props,
                    zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)
                        ncopies = atoi(strval);
                else
                        ncopies = 1;
                if (nvlist_lookup_uint64(real_props,
                    zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
                    &volblocksize) != 0)
                        volblocksize = ZVOL_DEFAULT_BLOCKSIZE;

                volsize = zvol_volsize_to_reservation(volsize,
                    volblocksize, ncopies);
                volsize = zvol_volsize_to_reservation(volsize, real_props);
                nvlist_free(real_props);

                if (nvlist_lookup_string(props, zfs_prop_to_name(resv_prop),
                    &strval) != 0) {
                        if (nvlist_add_uint64(props,
                            zfs_prop_to_name(resv_prop), volsize) != 0) {
                                nvlist_free(props);
                                nomem();
                        }
                }
        }

        if (parents && zfs_name_valid(argv[0], type)) {
                /*
                 * Now create the ancestors of target dataset.  If the target
                 * already exists and '-p' option was used we should not
                 * complain.
                 */
                if (zfs_dataset_exists(g_zfs, argv[0], type)) {
                        ret = 0;
                        goto error;
                }
                if (zfs_create_ancestors(g_zfs, argv[0]) != 0)
                        goto error;
        }

        /* pass to libzfs */
        if (zfs_create(g_zfs, argv[0], type, props) != 0)
                goto error;

        if ((zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_DATASET)) == NULL)
                goto error;

        ret = 0;
        /*
         * if the user doesn't want the dataset automatically mounted,
         * then skip the mount/share step
         */
        if (zfs_prop_valid_for_type(ZFS_PROP_CANMOUNT, type))
                canmount = zfs_prop_get_int(zhp, ZFS_PROP_CANMOUNT);
```

```
        /*
         * Mount and/or share the new filesystem as appropriate.  We provide a
         * verbose error message to let the user know that their filesystem was
         * in fact created, even if we failed to mount or share it.
         */
        if (canmount == ZFS_CANMOUNT_ON) {
                if (zfs_mount(zhp, NULL, 0) != 0) {
                        (void) fprintf(stderr, gettext("filesystem "
                            "successfully created, but not mounted\n"));
                        ret = 1;
                } else if (zfs_share(zhp) != 0) {
                        (void) fprintf(stderr, gettext("filesystem "
                            "successfully created, but not shared\n"));
                        ret = 1;
                }
        }

error:
        if (zhp)
                zfs_close(zhp);
        nvlist_free(props);
        return (ret);
badusage:
        nvlist_free(props);
        usage(B_FALSE);
        return (2);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   27058 Wed Oct 16 18:29:02 2013
new/usr/src/lib/libzfs/common/libzfs.h
4012 Upper limit of zfs set bounds check for refreservation on volumes is too lo
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright (c) 2013 Steven Hartland. All rights reserved.
 * Copyright 2013 DEY Storage Systems, Inc.
 */

#ifndef _LIBZFS_H
#define _LIBZFS_H

#include <assert.h>
#include <libnvpair.h>
#include <sys/mnttab.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/varargs.h>
#include <sys/fs/zfs.h>
#include <sys/avl.h>
#include <ucred.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Miscellaneous ZFS constants
 */
#define ZFS_MAXNAMELEN          MAXNAMELEN
#define ZPOOL_MAXNAMELEN        MAXNAMELEN
#define ZFS_MAXPROPLEN          MAXPATHLEN
#define ZPOOL_MAXPROPLEN        MAXPATHLEN

/*
 * libzfs errors
 */
typedef enum zfs_error {
        EZFS_SUCCESS = 0,       /* no error -- success */
        EZFS_NOMEM = 2000,      /* out of memory */
```

```
        EZFS_BADPROP,           /* invalid property value */
        EZFS_PROPREADONLY,      /* cannot set readonly property */
        EZFS_PROPTYPE,          /* property does not apply to dataset type */
        EZFS_PROPNONINHERIT,    /* property is not inheritable */
        EZFS_PROPSPACE,         /* bad quota or reservation */
        EZFS_BADTYPE,           /* dataset is not of appropriate type */
        EZFS_BUSY,              /* pool or dataset is busy */
        EZFS_EXISTS,            /* pool or dataset already exists */
        EZFS_NOENT,             /* no such pool or dataset */
        EZFS_BADSTREAM,         /* bad backup stream */
        EZFS_DSREADONLY,        /* dataset is readonly */
        EZFS_VOLTOOBIG,         /* volume is too large for 32-bit system */
        EZFS_INVALIDNAME,       /* invalid dataset name */
        EZFS_BADRESTORE,        /* unable to restore to destination */
        EZFS_BADBACKUP,         /* backup failed */
        EZFS_BADTARGET,         /* bad attach/detach/replace target */
        EZFS_NODEVICE,          /* no such device in pool */
        EZFS_BADDEV,            /* invalid device to add */
        EZFS_NOREPLICAS,        /* no valid replicas */
        EZFS_RESILVERING,       /* currently resilvering */
        EZFS_BADVERSION,        /* unsupported version */
        EZFS_POOLUNAVAIL,       /* pool is currently unavailable */
        EZFS_DEVOVERFLOW,       /* too many devices in one vdev */
        EZFS_BADPATH,           /* must be an absolute path */
        EZFS_CROSSTARGET,       /* rename or clone across pool or dataset */
        EZFS_ZONED,             /* used improperly in local zone */
        EZFS_MOUNTFAILED,       /* failed to mount dataset */
        EZFS_UMOUNTFAILED,      /* failed to unmount dataset */
        EZFS_UNSHARENFSFAILED,  /* unshare(1M) failed */
        EZFS_SHARENFSFAILED,    /* share(1M) failed */
        EZFS_PERM,              /* permission denied */
        EZFS_NOSPC,             /* out of space */
        EZFS_FAULT,             /* bad address */
        EZFS_IO,                /* I/O error */
        EZFS_INTR,              /* signal received */
        EZFS_ISSPARE,           /* device is a hot spare */
        EZFS_INVALCONFIG,       /* invalid vdev configuration */
        EZFS_RECURSIVE,         /* recursive dependency */
        EZFS_NOHISTORY,         /* no history object */
        EZFS_POOLPROPS,         /* couldn't retrieve pool props */
        EZFS_POOL_NOTSUP,       /* ops not supported for this type of pool */
        EZFS_POOL_INVALARG,     /* invalid argument for this pool operation */
        EZFS_NAMETOOLONG,       /* dataset name is too long */
        EZFS_OPENFAILED,        /* open of device failed */
        EZFS_NOCAP,             /* couldn't get capacity */
        EZFS_LABELFAILED,       /* write of label failed */
        EZFS_BADWHO,            /* invalid permission who */
        EZFS_BADPERM,           /* invalid permission */
        EZFS_BADPERMSET,        /* invalid permission set name */
        EZFS_NODELEGATION,      /* delegated administration is disabled */
        EZFS_UNSHARESMBFAILED,  /* failed to unshare over smb */
        EZFS_SHARESMBFAILED,    /* failed to share over smb */
        EZFS_BADCACHE,          /* bad cache file */
        EZFS_ISL2CACHE,         /* device is for the level 2 ARC */
        EZFS_VDEVNOTSUP,        /* unsupported vdev type */
        EZFS_NOTSUP,            /* ops not supported on this dataset */
        EZFS_ACTIVE_SPARE,      /* pool has active shared spare devices */
        EZFS_UNPLAYED_LOGS,     /* log device has unplayed logs */
        EZFS_REFTAG_RELE,       /* snapshot release: tag not found */
        EZFS_REFTAG_HOLD,       /* snapshot hold: tag already exists */
        EZFS_TAGTOOLONG,        /* snapshot hold/rele: tag too long */
        EZFS_PIPEFAILED,        /* pipe create failed */
        EZFS_THREADCREATEFAILED, /* thread create failed */
        EZFS_POSTSPLIT_ONLINE,  /* onlining a disk after splitting it */
        EZFS_SCRUBBING,         /* currently scrubbing */
        EZFS_NO_SCRUB,          /* no active scrub */
        EZFS_DIFF,              /* general failure of zfs diff */
        EZFS_DIFFDATA,          /* bad zfs diff data */
        EZFS_POOLREADONLY,      /* pool is in read-only mode */
        EZFS_UNKNOWN
} zfs_error_t;
______unchanged_portion_omitted_

typedef boolean_t (snapfilter_cb_t)(zfs_handle_t *, void *);

extern int zfs_send(zfs_handle_t *, const char *, const char *,
    sendflags_t *, int, snapfilter_cb_t, void *, nvlist_t **);

extern int zfs_promote(zfs_handle_t *);
extern int zfs_hold(zfs_handle_t *, const char *, const char *,
    boolean_t, int);
extern int zfs_hold_nvl(zfs_handle_t *, int, nvlist_t *);
extern int zfs_release(zfs_handle_t *, const char *, const char *, boolean_t);
extern int zfs_get_holds(zfs_handle_t *, nvlist_t **);
extern uint64_t zvol_volsize_to_reservation(uint64_t volsize,
    uint64_t volblocksize, int ncopies);
extern uint64_t zvol_volsize_to_reservation(uint64_t, nvlist_t *);

typedef int (*zfs_userspace_cb_t)(void *arg, const char *domain,
    uid_t rid, uint64_t space);

extern int zfs_userspace(zfs_handle_t *, zfs_userquota_prop_t,
    zfs_userspace_cb_t, void *);

extern int zfs_get_fsacl(zfs_handle_t *, nvlist_t **);
extern int zfs_set_fsacl(zfs_handle_t *, boolean_t, nvlist_t *);

typedef struct recvflags {
        /* print informational messages (ie, -v was specified) */
        boolean_t verbose;

        /* the destination is a prefix, not the exact fs (ie, -d) */
        boolean_t isprefix;

        /*
         * Only the tail of the sent snapshot path is appended to the
         * destination to determine the received snapshot name (ie, -e).
         */
        boolean_t istail;

        /* do not actually do the recv, just check if it would work (ie, -n) */
        boolean_t dryrun;

        /* rollback/destroy filesystems as necessary (eg, -F) */
        boolean_t force;

        /* set "canmount=off" on all modified filesystems */
        boolean_t canmountoff;

        /* byteswap flag is used internally; callers need not specify */
        boolean_t byteswap;

        /* do not mount file systems as they are extracted (private) */
        boolean_t nomount;
} recvflags_t;
______unchanged_portion_omitted_
```

```
**********************************************************
  111439 Wed Oct 16 18:29:02 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.c
4012 Upper limit of zfs set bounds check for refreservation on volumes is too lo
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

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

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

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

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

```
#include "zfs_deleg.h"

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

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

        return (NULL);
}
______unchanged_portion_omitted_

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

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

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

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

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

```
			}

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

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

		/*
		 * Currently, only user properties can be modified on
		 * snapshots.
		 */
		if (type == ZFS_TYPE_SNAPSHOT) {
			zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
			    "this property can not be modified for snapshots"));
			(void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
			goto error;
		}

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

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

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

			if (nvpair_type(elem) == DATA_TYPE_STRING) {
				(void) nvpair_value_string(elem, &strval);
				if (strcmp(strval, "none") == 0) {
					intval = 0;
				} else if (zfs_nicestrtonum(hdl,
				    strval, &intval) != 0) {
					(void) zfs_error(hdl,
					    EZFS_BADPROP, errbuf);
					goto error;
				}
```

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

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

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

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

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

		if (zprop_parse_value(hdl, elem, prop, type, ret,
```

```c
                    &strval, &intval, errbuf) != 0)
                        goto error;

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

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

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

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

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

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

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

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

```c
                        }
                        m_label_free(new_sl);

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

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

                        break;

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

                }

                case ZFS_PROP_MOUNTPOINT:
                {
                        namecheck_err_t why;

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

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

                        /*FALLTHRU*/

                case ZFS_PROP_SHARESMB:
                case ZFS_PROP_SHARENFS:
                        /*
                         * For the mountpoint and sharenfs or sharesmb
                         * properties, check if it can be set in a
                         * global/non-global zone based on
                         * the zoned property value:
                         *
                         *              global zone         non-global zone
```

```
			 * --------------------------------------------------
			 * zoned=on     mountpoint (no)     mountpoint (yes)
			 *              sharenfs (no)       sharenfs (no)
			 *              sharesmb (no)       sharesmb (no)
			 *
			 * zoned=off    mountpoint (yes)        N/A
			 *              sharenfs (yes)
			 *              sharesmb (yes)
			 */
			if (zoned) {
				if (getzoneid() == GLOBAL_ZONEID) {
					zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
					    "'%s' cannot be set on "
					    "dataset in a non-global zone"),
					    propname);
					(void) zfs_error(hdl, EZFS_ZONED,
					    errbuf);
					goto error;
				} else if (prop == ZFS_PROP_SHARENFS ||
				    prop == ZFS_PROP_SHARESMB) {
					zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
					    "'%s' cannot be set in "
					    "a non-global zone"), propname);
					(void) zfs_error(hdl, EZFS_ZONED,
					    errbuf);
					goto error;
				}
			} else if (getzoneid() != GLOBAL_ZONEID) {
				/*
				 * If zoned property is 'off', this must be in
				 * a global zone. If not, something is wrong.
				 */
				zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
				    "'%s' cannot be set while dataset "
				    "'zoned' property is set"), propname);
				(void) zfs_error(hdl, EZFS_ZONED, errbuf);
				goto error;
			}

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

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

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

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

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

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

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

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

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

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

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

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

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

```
        uint64_t volblocksize;
        int ncopies;
        nvlist_t *props;

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

        if ((zvol_volsize_to_reservation(old_volsize, volblocksize,
            ncopies) != old_reservation) || nvlist_exists(nvl,
            zfs_prop_to_name(resv_prop)))
        props = fnvlist_alloc();
        fnvlist_add_uint64(props, zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
            zfs_prop_get_int(zhp, ZFS_PROP_VOLBLOCKSIZE));

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

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

/*
 * Computes the required reservation to completely contain all blocks of a
 * zvol at a given volsize.
 * 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, uint64_t volblocksize,
    int ncopies)
zvol_volsize_to_reservation(uint64_t volsize, nvlist_t *props)
{
        uint64_t numdb;
        uint64_t nblocks;
        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);
}
```
______***unchanged_portion_omitted_***