

```

*****
162042 Tue Oct 15 13:59:52 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_____

672 /*
673 * zfs create [-p] [-o prop=value] ... fs
674 * zfs create [-ps] [-b blocksize] [-o prop=value] ... -V vol size
675 *
676 * Create a new dataset. This command can be used to create filesystems
677 * and volumes. Snapshot creation is handled by 'zfs snapshot'.
678 * For volumes, the user must specify a size to be used.
679 *
680 * The '-s' flag applies only to volumes, and indicates that we should not try
681 * to set the reservation for this volume. By default we set a reservation
682 * equal to the size for any volume. For pools with SPA_VERSION >=
683 * SPA_VERSION_REFRESERVATION, we set a refreservation instead.
684 *
685 * The '-p' flag creates all the non-existing ancestors of the target first.
686 */
687 static int
688 zfs_do_create(int argc, char **argv)
689 {
690     zfs_type_t type = ZFS_TYPE_FILESYSTEM;
691     zfs_handle_t *zhp = NULL;
692     uint64_t volsize;
693     int c;
694     boolean_t noreserve = B_FALSE;
695     boolean_t bflag = B_FALSE;
696     boolean_t parents = B_FALSE;
697     int ret = 1;
698     nvlist_t *props;
699     uint64_t intval;
700     int canmount = ZFS_CANMOUNT_OFF;

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

705     /* check options */
706     while ((c = getopt(argc, argv, ":V:b:so:p")) != -1) {
707         switch (c) {
708             case 'V':
709                 type = ZFS_TYPE_VOLUME;
710                 if (zfs_nicestrtonum(g_zfs, optarg, &intval) != 0) {
711                     (void) fprintf(stderr, gettext("bad volume "
712 "size '%s': %s\n"), optarg,
713 libzfs_error_description(g_zfs));
714                     goto error;
715                 }

717                 if (nvlist_add_uint64(props,
718 zfs_prop_to_name(ZFS_PROP_VOLSIZE), intval) != 0)
719                     nomem();
720                 volsize = intval;
721                 break;
722             case 'p':
723                 parents = B_TRUE;
724                 break;
725             case 'b':
726                 bflag = B_TRUE;
727                 if (zfs_nicestrtonum(g_zfs, optarg, &intval) != 0) {
728                     (void) fprintf(stderr, gettext("bad volume "
729 "block size '%s': %s\n"), optarg,
730 libzfs_error_description(g_zfs));

```

```

731         goto error;
732     }

734     if (nvlist_add_uint64(props,
735 zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
736 intval) != 0)
737         nomem();
738     break;
739 case 'o':
740     if (parseprop(props))
741         goto error;
742     break;
743 case 's':
744     noreserve = B_TRUE;
745     break;
746 case ':':
747     (void) fprintf(stderr, gettext("missing size "
748 "argument\n"));
749     goto badusage;
750 case '?':
751     (void) fprintf(stderr, gettext("invalid option '%c'\n"),
752 optopt);
753     goto badusage;
754 }
755 }

757 if ((bflag || noreserve) && type != ZFS_TYPE_VOLUME) {
758     (void) fprintf(stderr, gettext("'s' and '-b' can only be "
759 "used when creating a volume\n"));
760     goto badusage;
761 }

763 argc -= optind;
764 argv += optind;

766 /* check number of arguments */
767 if (argc == 0) {
768     (void) fprintf(stderr, gettext("missing %s argument\n"),
769 zfs_type_to_name(type));
770     goto badusage;
771 }
772 if (argc > 1) {
773     (void) fprintf(stderr, gettext("too many arguments\n"));
774     goto badusage;
775 }

777 if (type == ZFS_TYPE_VOLUME && !noreserve) {
778     zpool_handle_t *zpool_handle;
779     nvlist_t *real_props;
780     uint64_t spa_version;
781     char *p;
782     zfs_prop_t resv_prop;
783     char *strval;
784     char msg[1024];
785     uint64_t volblocksize;
786     int ncopies;

788     if (p = strchr(argv[0], '/'))
789         *p = '\0';
790     zpool_handle = zpool_open(g_zfs, argv[0]);
791     if (p != NULL)
792         *p = '/';
793     if (zpool_handle == NULL)
794         goto error;
795     spa_version = zpool_get_prop_int(zpool_handle,
796 ZPOOL_PROP_VERSION, NULL);

```

```

797     zpool_close(zpool_handle);
798     if (spa_version >= SPA_VERSION_REFRESERVATION)
799         resv_prop = ZFS_PROP_REFRESERVATION;
800     else
801         resv_prop = ZFS_PROP_RESERVATION;
802
803     (void) snprintf(msg, sizeof (msg),
804         gettext("cannot create '%s'"), argv[0]);
805     if (props && (real_props = zfs_valid_proplist(g_zfs, type,
806         props, 0, NULL, msg)) == NULL)
807         goto error;
808
809     if (nvlist_lookup_string(real_props,
810         zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)
811         ncopies = atoi(strval);
812     else
813         ncopies = 1;
814     if (nvlist_lookup_uint64(real_props,
815         zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
816         &volblocksize) != 0)
817         volblocksize = ZVOL_DEFAULT_BLOCKSIZE;
818
819     volsize = zvol_volsize_to_reservation_impl(volsize,
820         volblocksize, ncopies);
821     volsize = zvol_volsize_to_reservation(volsize, real_props);
822     nvlist_free(real_props);
823
824     if (nvlist_lookup_string(props, zfs_prop_to_name(resv_prop),
825         &strval) != 0) {
826         if (nvlist_add_uint64(props,
827             zfs_prop_to_name(resv_prop), volsize) != 0) {
828             nvlist_free(props);
829             nomem();
830         }
831     }
832
833     if (parents && zfs_name_valid(argv[0], type)) {
834         /*
835          * Now create the ancestors of target dataset. If the target
836          * already exists and '-p' option was used we should not
837          * complain.
838          */
839         if (zfs_dataset_exists(g_zfs, argv[0], type)) {
840             ret = 0;
841             goto error;
842         }
843         if (zfs_create_ancestors(g_zfs, argv[0]) != 0)
844             goto error;
845     }
846
847     /* pass to libzfs */
848     if (zfs_create(g_zfs, argv[0], type, props) != 0)
849         goto error;
850
851     if ((zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_DATASET)) == NULL)
852         goto error;
853
854     ret = 0;
855     /*
856      * if the user doesn't want the dataset automatically mounted,
857      * then skip the mount/share step
858      */
859     if (zfs_prop_valid_for_type(ZFS_PROP_CANMOUNT, type))
860         canmount = zfs_prop_get_int(zhp, ZFS_PROP_CANMOUNT);

```

```

862     /*
863      * Mount and/or share the new filesystem as appropriate. We provide a
864      * verbose error message to let the user know that their filesystem was
865      * in fact created, even if we failed to mount or share it.
866      */
867     if (canmount == ZFS_CANMOUNT_ON) {
868         if (zfs_mount(zhp, NULL, 0) != 0) {
869             (void) fprintf(stderr, gettext("filesystem "
870                 "successfully created, but not mounted\n"));
871             ret = 1;
872         } else if (zfs_share(zhp) != 0) {
873             (void) fprintf(stderr, gettext("filesystem "
874                 "successfully created, but not shared\n"));
875             ret = 1;
876         }
877     }
878
879 error:
880     if (zhp)
881         zfs_close(zhp);
882     nvlist_free(props);
883     return (ret);
884 badusage:
885     nvlist_free(props);
886     usage(B_FALSE);
887     return (2);
888 }

```

\_\_\_\_\_unchanged\_portion\_omitted\_\_\_\_\_

```

*****
27130 Tue Oct 15 13:59:52 2013
new/usr/src/lib/libzfs/common/libzfs.h
4012 Upper limit of zfs set bounds check for reservation on volumes is too lo
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
25  * Copyright (c) 2012 by Delphix. All rights reserved.
26  * Copyright (c) 2012, Joyent, Inc. All rights reserved.
27  * Copyright (c) 2013 Steven Hartland. All rights reserved.
28  * Copyright 2013 DEY Storage Systems, Inc.
29 */

31 #ifndef _LIBZFS_H
32 #define _LIBZFS_H

34 #include <assert.h>
35 #include <libnvpair.h>
36 #include <sys/mnttab.h>
37 #include <sys/param.h>
38 #include <sys/types.h>
39 #include <sys/varargs.h>
40 #include <sys/fs/zfs.h>
41 #include <sys/avl.h>
42 #include <ucred.h>

44 #ifdef __cplusplus
45 extern "C" {
46 #endif

48 /*
49  * Miscellaneous ZFS constants
50  */
51 #define ZFS_MAXNAMELEN      MAXNAMELEN
52 #define ZPOOL_MAXNAMELEN   MAXNAMELEN
53 #define ZFS_MAXPROPLEN     MAXPATHLEN
54 #define ZPOOL_MAXPROPLEN   MAXPATHLEN

56 /*
57  * libzfs errors
58  */
59 typedef enum zfs_error {
60     EZFS_SUCCESS = 0,      /* no error -- success */
61     EZFS_NOMEM = 2000,    /* out of memory */

```

```

62     EZFS_BADPROP,         /* invalid property value */
63     EZFS_PROPREADONLY,    /* cannot set readonly property */
64     EZFS_PROPTYPE,        /* property does not apply to dataset type */
65     EZFS_PROPNONINHERIT, /* property is not inheritable */
66     EZFS_PROPSPACE,       /* bad quota or reservation */
67     EZFS_BADTYPE,         /* dataset is not of appropriate type */
68     EZFS_BUSY,            /* pool or dataset is busy */
69     EZFS_EXISTS,          /* pool or dataset already exists */
70     EZFS_NOENT,           /* no such pool or dataset */
71     EZFS_BADSTREAM,       /* bad backup stream */
72     EZFS_DSREADONLY,      /* dataset is readonly */
73     EZFS_VOLTOOBIG,       /* volume is too large for 32-bit system */
74     EZFS_INVALIDNAME,     /* invalid dataset name */
75     EZFS_BADRESTORE,      /* unable to restore to destination */
76     EZFS_BADBACKUP,       /* backup failed */
77     EZFS_BADTARGET,       /* bad attach/detach/replace target */
78     EZFS_NODEVICE,        /* no such device in pool */
79     EZFS_BADDEV,          /* invalid device to add */
80     EZFS_NOREPLICAS,      /* no valid replicas */
81     EZFS_RESILVERING,      /* currently resilvering */
82     EZFS_BADVERSION,       /* unsupported version */
83     EZFS_POOLUNAVAIL,     /* pool is currently unavailable */
84     EZFS_DEVOVERFLOW,     /* too many devices in one vdev */
85     EZFS_BADPATH,         /* must be an absolute path */
86     EZFS_CROSTARGET,       /* rename or clone across pool or dataset */
87     EZFS_ZONED,           /* used improperly in local zone */
88     EZFS_MOUNTFAILED,     /* failed to mount dataset */
89     EZFS_UMOUNTFAILED,    /* failed to unmount dataset */
90     EZFS_UNSHARENFSAILED, /* unshare(1M) failed */
91     EZFS_SHARENFSAILED,   /* share(1M) failed */
92     EZFS_PERM,            /* permission denied */
93     EZFS_NOSPC,           /* out of space */
94     EZFS_FAULT,           /* bad address */
95     EZFS_IO,              /* I/O error */
96     EZFS_INTR,           /* signal received */
97     EZFS_ISSPARE,         /* device is a hot spare */
98     EZFS_INVALIDCONFIG,   /* invalid vdev configuration */
99     EZFS_RECURSIVE,       /* recursive dependency */
100    EZFS_NOHISTORY,        /* no history object */
101    EZFS_POOLPROPS,        /* couldn't retrieve pool props */
102    EZFS_POOL_NOTSUP,      /* ops not supported for this type of pool */
103    EZFS_POOL_INVALIDARG, /* invalid argument for this pool operation */
104    EZFS_NAMETOOLONG,      /* dataset name is too long */
105    EZFS_OPENFAILED,       /* open of device failed */
106    EZFS_NOCAP,            /* couldn't get capacity */
107    EZFS_LABELFAILED,      /* write of label failed */
108    EZFS_BADWHO,           /* invalid permission who */
109    EZFS_BADPERM,          /* invalid permission */
110    EZFS_BADPERMSET,       /* invalid permission set name */
111    EZFS_NODELEGATION,     /* delegated administration is disabled */
112    EZFS_UNSHARESMBFAILED, /* failed to unshare over smb */
113    EZFS_SHARESMBFAILED,   /* failed to share over smb */
114    EZFS_BADCACHE,         /* bad cache file */
115    EZFS_ISL2CACHE,        /* device is for the level 2 ARC */
116    EZFS_VDEVNOTSUP,       /* unsupported vdev type */
117    EZFS_NOTSUP,           /* ops not supported on this dataset */
118    EZFS_ACTIVE_SPARE,     /* pool has active shared spare devices */
119    EZFS_UNPLAYED_LOGS,    /* log device has unplayed logs */
120    EZFS_REFTAG_RELE,      /* snapshot release: tag not found */
121    EZFS_REFTAG_HOLD,      /* snapshot hold: tag already exists */
122    EZFS_TAGTOOLONG,       /* snapshot hold/rele: tag too long */
123    EZFS_PIPEFAILED,       /* pipe create failed */
124    EZFS_THREADCREATEFAILED, /* thread create failed */
125    EZFS_POSTSPLIT_ONLINE, /* onlining a disk after splitting it */
126    EZFS_SCRUBBING,        /* currently scrubbing */
127    EZFS_NO_SCRUB,         /* no active scrub */

```

```
128     EZFS_DIFF,           /* general failure of zfs diff */
129     EZFS_DIFFDATA,      /* bad zfs diff data */
130     EZFS_POOLREADONLY,  /* pool is in read-only mode */
131     EZFS_UNKNOWN
132 } zfs_error_t;
unchanged portion omitted

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

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

597 extern int zfs_promote(zfs_handle_t *);
598 extern int zfs_hold(zfs_handle_t *, const char *, const char *,
599     boolean_t, int);
600 extern int zfs_hold_nvl(zfs_handle_t *, int, nvlist_t *);
601 extern int zfs_release(zfs_handle_t *, const char *, const char *, boolean_t);
602 extern int zfs_get_holds(zfs_handle_t *, nvlist_t **);
603 extern uint64_t zvol_volsize_to_reservation(uint64_t, nvlist_t *);
604 extern uint64_t zvol_volsize_to_reservation_impl(uint64_t volsize,
605     uint64_t volblocksize, int ncopies);

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

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

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

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

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

623     /*
624      * Only the tail of the sent snapshot path is appended to the
625      * destination to determine the received snapshot name (ie, -e).
626      */
627     boolean_t istail;

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

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

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

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

641     /* do not mount file systems as they are extracted (private) */
642     boolean_t nomount;
643 } recvflags_t;
unchanged portion omitted
```

```

*****
112221 Tue Oct 15 13:59:52 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.c
4012 Upper limit of zfs set bounds check for reservation on volumes is too lo
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 2013 by Delphix. All rights reserved.
25  * Copyright 2013 DEY Storage Systems, Inc.
26  * Copyright (c) 2012 DEY Storage Systems, Inc. All rights reserved.
27  * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
28  * Copyright (c) 2013 Martin Matuska. All rights reserved.
29  * Copyright (c) 2013 Steven Hartland. All rights reserved.
30 */

31 #include <ctype.h>
32 #include <errno.h>
33 #include <libintl.h>
34 #include <math.h>
35 #include <stdio.h>
36 #include <stdlib.h>
37 #include <strings.h>
38 #include <unistd.h>
39 #include <stddef.h>
40 #include <zone.h>
41 #include <fcntl.h>
42 #include <sys/mntent.h>
43 #include <sys/mount.h>
44 #include <priv.h>
45 #include <pwd.h>
46 #include <grp.h>
47 #include <stddef.h>
48 #include <ucred.h>
49 #include <idmap.h>
50 #include <aclutils.h>
51 #include <directory.h>

53 #include <sys/dnode.h>
54 #include <sys/spa.h>
55 #include <sys/zap.h>
56 #include <libzfs.h>

58 #include "zfs_namecheck.h"
59 #include "zfs_prop.h"
60 #include "libzfs_impl.h"

```

```

61 #include "zfs_deleg.h"

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

66 /*
67  * Given a single type (not a mask of types), return the type in a human
68  * readable form.
69  */
70 const char *
71 zfs_type_to_name(zfs_type_t type)
72 {
73     switch (type) {
74     case ZFS_TYPE_FILESYSTEM:
75         return (dgettext(TEXT_DOMAIN, "filesystem"));
76     case ZFS_TYPE_SNAPSHOT:
77         return (dgettext(TEXT_DOMAIN, "snapshot"));
78     case ZFS_TYPE_VOLUME:
79         return (dgettext(TEXT_DOMAIN, "volume"));
80     }

82     return (NULL);
83 }

    unchanged portion omitted

789 /*
790  * Given an nvlist of properties to set, validates that they are correct, and
791  * parses any numeric properties (index, boolean, etc) if they are specified as
792  * strings.
793  */
794 nvlist_t *
795 zfs_valid_proplist(libzfs_handle_t *hdl, zfs_type_t type, nvlist_t *nvl,
796   uint64_t zoned, zfs_handle_t *zhp, const char *errbuf)
797 {
798     nvpair_t *elem;
799     uint64_t intval;
800     char *strval;
801     zfs_prop_t prop;
802     nvlist_t *ret;
803     int chosen_normal = -1;
804     int chosen_utf = -1;

806     if (nvlist_alloc(&ret, NV_UNIQUE_NAME, 0) != 0) {
807         (void) no_memory(hdl);
808         return (NULL);
809     }

811     /*
812      * Make sure this property is valid and applies to this type.
813      */

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

819         prop = zfs_name_to_prop(propname);
820         if (prop == ZPROP_INVALID && zfs_prop_user(propname)) {
821             /*
822              * This is a user property: make sure it's a
823              * string, and that it's less than ZAP_MAXNAMELEN.
824              */
825             if (nvpair_type(elem) != DATA_TYPE_STRING) {
826                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
827                   "'%s' must be a string"), propname);
828                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
829                 goto error;

```

```

830         }
831
832         if (strlen(nvpair_name(elem)) >= ZAP_MAXNAMELEN) {
833             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
834                 "property name '%s' is too long"),
835                 propname);
836             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
837             goto error;
838         }
839
840         (void) nvpair_value_string(elem, &strval);
841         if (nvlist_add_string(ret, propname, strval) != 0) {
842             (void) no_memory(hdl);
843             goto error;
844         }
845         continue;
846     }
847
848     /*
849     * Currently, only user properties can be modified on
850     * snapshots.
851     */
852     if (type == ZFS_TYPE_SNAPSHOT) {
853         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
854             "this property can not be modified for snapshots"));
855         (void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
856         goto error;
857     }
858
859     if (prop == ZPROP_INVALID && zfs_prop_userquota(propname)) {
860         zfs_userquota_prop_t uqtype;
861         char newpropname[128];
862         char domain[128];
863         uint64_t rid;
864         uint64_t valary[3];
865
866         if (userquota_propname_decode(propname, zoned,
867             &uqtype, domain, sizeof(domain), &rid) != 0) {
868             zfs_error_aux(hdl,
869                 dgettext(TEXT_DOMAIN,
870                     "'%s' has an invalid user/group name"),
871                 propname);
872             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
873             goto error;
874         }
875
876         if (uqtype != ZFS_PROP_USERQUOTA &&
877             uqtype != ZFS_PROP_GROUPQUOTA) {
878             zfs_error_aux(hdl,
879                 dgettext(TEXT_DOMAIN, "'%s' is readonly"),
880                 propname);
881             (void) zfs_error(hdl, EZFS_PROPREADONLY,
882                 errbuf);
883             goto error;
884         }
885
886         if (nvpair_type(elem) == DATA_TYPE_STRING) {
887             (void) nvpair_value_string(elem, &strval);
888             if (strcmp(strval, "none") == 0) {
889                 intval = 0;
890             } else if (zfs_nicestrtonum(hdl,
891                 strval, &intval) != 0) {
892                 (void) zfs_error(hdl,
893                     EZFS_BADPROP, errbuf);
894                 goto error;
895             }

```

```

896         } else if (nvpair_type(elem) ==
897             DATA_TYPE_UINT64) {
898             (void) nvpair_value_uint64(elem, &intval);
899             if (intval == 0) {
900                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
901                     "use 'none' to disable "
902                     "userquota/groupquota"));
903                 goto error;
904             }
905         } else {
906             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
907                 "'%s' must be a number"), propname);
908             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
909             goto error;
910         }
911
912         /*
913         * Encode the prop name as
914         * userquota@<hex-rid>-domain, to make it easy
915         * for the kernel to decode.
916         */
917         (void) snprintf(newpropname, sizeof(newpropname),
918             "%s%llx-%s", zfs_userquota_prop_prefixes[uqtype],
919             (longlong_t)rid, domain);
920         valary[0] = uqtype;
921         valary[1] = rid;
922         valary[2] = intval;
923         if (nvlist_add_uint64_array(ret, newpropname,
924             valary, 3) != 0) {
925             (void) no_memory(hdl);
926             goto error;
927         }
928         continue;
929     } else if (prop == ZPROP_INVALID && zfs_prop_written(propname)) {
930         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
931             "'%s' is readonly"),
932             propname);
933         (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
934         goto error;
935     }
936
937     if (prop == ZPROP_INVALID) {
938         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
939             "invalid property '%s'"), propname);
940         (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
941         goto error;
942     }
943
944     if (!zfs_prop_valid_for_type(prop, type)) {
945         zfs_error_aux(hdl,
946             dgettext(TEXT_DOMAIN, "'%s' does not "
947                 "apply to datasets of this type"), propname);
948         (void) zfs_error(hdl, EZFS_PROPTYPE, errbuf);
949         goto error;
950     }
951
952     if (zfs_prop_readonly(prop) &&
953         (!zfs_prop_setonce(prop) || zhp != NULL)) {
954         zfs_error_aux(hdl,
955             dgettext(TEXT_DOMAIN, "'%s' is readonly"),
956             propname);
957         (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
958         goto error;
959     }
960
961     if (zprop_parse_value(hdl, elem, prop, type, ret,

```

```

962         &strval, &intval, errbuf) != 0)
963             goto error;

965     /*
966      * Perform some additional checks for specific properties.
967      */
968     switch (prop) {
969     case ZFS_PROP_VERSION:
970     {
971         int version;

973         if (zhp == NULL)
974             break;
975         version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
976         if (intval < version) {
977             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
978                 "Can not downgrade; already at version %u"),
979                 version);
980             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
981             goto error;
982         }
983         break;
984     }

986     case ZFS_PROP_RECORDSIZE:
987     case ZFS_PROP_VOLBLOCKSIZE:
988         /* must be power of two within SPA_{MIN,MAX}BLOCKSIZE */
989         if (intval < SPA_MINBLOCKSIZE ||
990             intval > SPA_MAXBLOCKSIZE || !ISP2(intval)) {
991             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
992                 "'%s' must be power of 2 from %u "
993                 "to %uk"), propname,
994                 (uint_t)SPA_MINBLOCKSIZE,
995                 (uint_t)SPA_MAXBLOCKSIZE >> 10);
996             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
997             goto error;
998         }
999         break;

1001     case ZFS_PROP_MLSLABEL:
1002     {
1003         /*
1004          * Verify the mlslabel string and convert to
1005          * internal hex label string.
1006          */

1008         m_label_t *new_sl;
1009         char *hex = NULL;          /* internal label string */

1011         /* Default value is already OK. */
1012         if (strcasecmp(strval, ZFS_MLSLABEL_DEFAULT) == 0)
1013             break;

1015         /* Verify the label can be converted to binary form */
1016         if (((new_sl = m_label_alloc(MAC_LABEL)) == NULL) ||
1017             (str_to_label(strval, &new_sl, MAC_LABEL,
1018                 L_NO_CORRECTION, NULL) == -1)) {
1019             goto badlabel;
1020         }

1022         /* Now translate to hex internal label string */
1023         if (label_to_str(new_sl, &hex, M_INTERNAL,
1024             DEF_NAMES) != 0) {
1025             if (hex)
1026                 free(hex);
1027             goto badlabel;

```

```

1028         }
1029         m_label_free(new_sl);

1031         /* If string is already in internal form, we're done. */
1032         if (strcmp(strval, hex) == 0) {
1033             free(hex);
1034             break;
1035         }

1037         /* Replace the label string with the internal form. */
1038         (void) nvlist_remove(ret, zfs_prop_to_name(prop),
1039             DATA_TYPE_STRING);
1040         verify(nvlist_add_string(ret, zfs_prop_to_name(prop),
1041             hex) == 0);
1042         free(hex);

1044         break;

1046     badlabel:
1047         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1048             "invalid mlslabel '%s'"), strval);
1049         (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1050         m_label_free(new_sl);      /* OK if null */
1051         goto error;

1053     }

1055     case ZFS_PROP_MOUNTPOINT:
1056     {
1057         namecheck_err_t why;

1059         if (strcmp(strval, ZFS_MOUNTPOINT_NONE) == 0 ||
1060             strcmp(strval, ZFS_MOUNTPOINT_LEGACY) == 0)
1061             break;

1063         if (mountpoint_namecheck(strval, &why)) {
1064             switch (why) {
1065             case NAME_ERR_LEADING_SLASH:
1066                 zfs_error_aux(hdl,
1067                     dgettext(TEXT_DOMAIN,
1068                         "'%s' must be an absolute path, "
1069                         "'none', or 'legacy'"), propname);
1070                 break;
1071             case NAME_ERR_TOOLONG:
1072                 zfs_error_aux(hdl,
1073                     dgettext(TEXT_DOMAIN,
1074                         "component of '%s' is too long"),
1075                     propname);
1076                 break;
1077             }
1078             (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1079             goto error;
1080         }
1081     }

1083     /*FALLTHRU*/

1085     case ZFS_PROP_SHARESMB:
1086     case ZFS_PROP_SHARENFPS:
1087         /*
1088          * For the mountpoint and sharenfs or sharesmb
1089          * properties, check if it can be set in a
1090          * global/non-global zone based on
1091          * the zoned property value:
1092          *
1093          *

```

```

                global zone          non-global zone

```

```

1094 * -----
1095 * zoned=on      mountpoint (no)   mountpoint (yes)
1096 *              sharenfs (no)    sharenfs (no)
1097 *              sharesmb (no)    sharesmb (no)
1098 *
1099 * zoned=off    mountpoint (yes)   N/A
1100 *              sharenfs (yes)
1101 *              sharesmb (yes)
1102 */
1103 if (zoned) {
1104     if (getzoneid() == GLOBAL_ZONEID) {
1105         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1106             "'%s' cannot be set on "
1107             "dataset in a non-global zone"),
1108             propname);
1109         (void) zfs_error(hdl, EZFS_ZONED,
1110             errbuf);
1111         goto error;
1112     } else if (prop == ZFS_PROP_SHARENFS ||
1113         prop == ZFS_PROP_SHARESMB) {
1114         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1115             "'%s' cannot be set in "
1116             "a non-global zone"), propname);
1117         (void) zfs_error(hdl, EZFS_ZONED,
1118             errbuf);
1119         goto error;
1120     }
1121 } else if (getzoneid() != GLOBAL_ZONEID) {
1122     /*
1123      * If zoned property is 'off', this must be in
1124      * a global zone. If not, something is wrong.
1125      */
1126     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1127         "'%s' cannot be set while dataset "
1128         "'zoned' property is set"), propname);
1129     (void) zfs_error(hdl, EZFS_ZONED, errbuf);
1130     goto error;
1131 }
1132
1133 /*
1134  * At this point, it is legitimate to set the
1135  * property. Now we want to make sure that the
1136  * property value is valid if it is sharenfs.
1137  */
1138 if ((prop == ZFS_PROP_SHARENFS ||
1139     prop == ZFS_PROP_SHARESMB) &&
1140     strcmp(strval, "on") != 0 &&
1141     strcmp(strval, "off") != 0) {
1142     zfs_share_proto_t proto;
1143
1144     if (prop == ZFS_PROP_SHARESMB)
1145         proto = PROTO_SMB;
1146     else
1147         proto = PROTO_NFS;
1148
1149     /*
1150      * Must be a valid sharing protocol
1151      * option string so init the libshare
1152      * in order to enable the parser and
1153      * then parse the options. We use the
1154      * control API since we don't care about
1155      * the current configuration and don't
1156      * want the overhead of loading it
1157      * until we actually do something.
1158      */

```

```

1160 if (zfs_init_libshare(hdl,
1161     SA_INIT_CONTROL_API) != SA_OK) {
1162     /*
1163      * An error occurred so we can't do
1164      * anything
1165      */
1166     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1167         "'%s' cannot be set: problem "
1168         "in share initialization"),
1169         propname);
1170     (void) zfs_error(hdl, EZFS_BADPROP,
1171         errbuf);
1172     goto error;
1173 }
1174
1175 if (zfs_parse_options(strval, proto) != SA_OK) {
1176     /*
1177      * There was an error in parsing so
1178      * deal with it by issuing an error
1179      * message and leaving after
1180      * uninitializing the the libshare
1181      * interface.
1182      */
1183     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1184         "'%s' cannot be set to invalid "
1185         "options"), propname);
1186     (void) zfs_error(hdl, EZFS_BADPROP,
1187         errbuf);
1188     zfs_uninit_libshare(hdl);
1189     goto error;
1190 }
1191 zfs_uninit_libshare(hdl);
1192
1193 }
1194
1195 break;
1196 case ZFS_PROP_UTF8ONLY:
1197     chosen_utf = (int)intval;
1198     break;
1199 case ZFS_PROP_NORMALIZE:
1200     chosen_normal = (int)intval;
1201     break;
1202 }
1203
1204 /*
1205  * For changes to existing volumes, we have some additional
1206  * checks to enforce.
1207  */
1208 if (type == ZFS_TYPE_VOLUME && zhp != NULL) {
1209     uint64_t volsize = zfs_prop_get_int(zhp,
1210         ZFS_PROP_VOLSIZE);
1211     uint64_t blocksize = zfs_prop_get_int(zhp,
1212         ZFS_PROP_VOLBLOCKSIZE);
1213     int ncopies = zfs_prop_get_int(zhp, ZFS_PROP_COPIES);
1214     char buf[64];
1215
1216     switch (prop) {
1217     case ZFS_PROP_RESERVATION:
1218     case ZFS_PROP_REFRESERVATION:
1219         if (intval >
1220             zvol_volsize_to_reservation_impl(volsize,
1221                 blocksize, ncopies)) {
1222             if (intval > volsize) {
1223                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1224                     "'%s' is greater than current "
1225                     "volume size"), propname);
1226                 (void) zfs_error(hdl, EZFS_BADPROP,

```



```

1225         errbuf);
1226         goto error;
1227     }
1228     break;

1230     case ZFS_PROP_VOLSIZE:
1231         if (intval % blocksize != 0) {
1232             zfs_nicenum(blocksize, buf,
1233                 sizeof (buf));
1234             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1235                 "%s' must be a multiple of "
1236                 "volume block size (%s)"),
1237                 propname, buf);
1238             (void) zfs_error(hdl, EZFS_BADPROP,
1239                 errbuf);
1240             goto error;
1241         }

1243         if (intval == 0) {
1244             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1245                 "%s' cannot be zero"),
1246                 propname);
1247             (void) zfs_error(hdl, EZFS_BADPROP,
1248                 errbuf);
1249             goto error;
1250         }
1251     }
1252     break;
1253 }
1254 }

1256 /*
1257  * If normalization was chosen, but no UTF8 choice was made,
1258  * enforce rejection of non-UTF8 names.
1259  *
1260  * If normalization was chosen, but rejecting non-UTF8 names
1261  * was explicitly not chosen, it is an error.
1262  */
1263 if (chosen_normal > 0 && chosen_utf < 0) {
1264     if (nvlist_add_uint64(ret,
1265         zfs_prop_to_name(ZFS_PROP_UTF8ONLY), 1) != 0) {
1266         (void) no_memory(hdl);
1267         goto error;
1268     }
1269 } else if (chosen_normal > 0 && chosen_utf == 0) {
1270     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1271         "%s' must be set 'on' if normalization chosen"),
1272         zfs_prop_to_name(ZFS_PROP_UTF8ONLY));
1273     (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
1274     goto error;
1275 }
1276 return (ret);

1278 error:
1279     nvlist_free(ret);
1280     return (NULL);
1281 }

1283 int
1284 zfs_add_synthetic_resv(zfs_handle_t *zhp, nvlist_t *nvl)
1285 {
1286     uint64_t old_volsize;
1287     uint64_t new_volsize;
1288     uint64_t old_reservation;
1289     uint64_t new_reservation;
1290     zfs_prop_t resv_prop;

```

```

1291     uint64_t volblocksize;
1292     int ncopies;
1293     nvlist_t *props;

1294     /*
1295      * If this is an existing volume, and someone is setting the volsize,
1296      * make sure that it matches the reservation, or add it if necessary.
1297      */
1298     old_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
1299     if (zfs_which_resv_prop(zhp, &resv_prop) < 0)
1300         return (-1);
1301     old_reservation = zfs_prop_get_int(zhp, resv_prop);
1302     volblocksize = zfs_prop_get_int(zhp, ZFS_PROP_VOLBLOCKSIZE);
1303     ncopies = zfs_prop_get_int(zhp, ZFS_PROP_COPIES);

1305     if ((zvol_volsize_to_reservation_impl(old_volsize, volblocksize,
1306         ncopies) != old_reservation) || nvlist_exists(nvl,
1307         zfs_prop_to_name(resv_prop)))
1308         props = fnvlist_alloc();
1309     fnvlist_add_uint64(props, zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
1310         zfs_prop_get_int(zhp, ZFS_PROP_VOLBLOCKSIZE));

1312     if ((zvol_volsize_to_reservation(old_volsize, props) !=
1313         old_reservation) || nvlist_exists(nvl,
1314         zfs_prop_to_name(resv_prop))) {
1315         fnvlist_free(props);
1316         return (0);
1317     }

1319     if (nvlist_lookup_uint64(nvl, zfs_prop_to_name(ZFS_PROP_VOLSIZE),
1320         &new_volsize) != 0)
1321         &new_volsize != 0) {
1322         fnvlist_free(props);
1323         return (-1);
1324     }
1325     new_reservation = zvol_volsize_to_reservation_impl(new_volsize,
1326         volblocksize, ncopies);
1327     new_reservation = zvol_volsize_to_reservation(new_volsize, props);
1328     fnvlist_free(props);

1330     if (nvlist_add_uint64(nvl, zfs_prop_to_name(resv_prop),
1331         new_reservation) != 0) {
1332         (void) no_memory(zhp->zfs_hdl);
1333         return (-1);
1334     }
1335     return (1);
1336 }

unchanged_portion_omitted

4460 /*
4461  * Convert the zvol's volume size to an appropriate reservation. This is a
4462  * convenience front-end to zvol_volsize_to_reservation_impl.
4463  * Convert the zvol's volume size to an appropriate reservation.
4464  * Note: If this routine is updated, it is necessary to update the ZFS test
4465  * suite's shell version in reservation.kshlib.
4466  */
4467 uint64_t
4468 zvol_volsize_to_reservation(uint64_t volsize, nvlist_t *props)
4469 {
4470     uint64_t volblocksize;
4471     uint64_t numdb;
4472     uint64_t nblocks, volblocksize;
4473     int ncopies;
4474     char *strval;

4476     if (nvlist_lookup_string(props,
4477         zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)

```

```
4475         ncopies = atoi(strval);
4476     else
4477         ncopies = 1;
4478     if (nvlist_lookup_uint64(props,
4479         zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
4480         &volblocksize) != 0)
4481         volblocksize = ZVOL_DEFAULT_BLOCKSIZE;
4483     return (zvol_volsize_to_reservation_impl(volsize, volblocksize,
4484         ncopies));
4485 }
4487 /*
4488  * Computes the required reservation to completely contain all blocks of a
4489  * zvol at a given volsize.
4490  */
4491 uint64_t
4492 zvol_volsize_to_reservation_impl(uint64_t volsize, uint64_t volblocksize,
4493     int ncopies)
4494 {
4495     uint64_t numdb;
4496     uint64_t nblocks;
4498     nblocks = volsize/volblocksize;
4499     /* start with metadnode L0-L6 */
4500     numdb = 7;
4501     /* calculate number of indirects */
4502     while (nblocks > 1) {
4503         nblocks += DNODES_PER_LEVEL - 1;
4504         nblocks /= DNODES_PER_LEVEL;
4505         numdb += nblocks;
4506     }
4507     numdb *= MIN(SPA_DVAS_PER_BP, ncopies + 1);
4508     volsize *= ncopies;
4509     /*
4510      * this is exactly DN_MAX_INDBLKSHIFT when metadata isn't
4511      * compressed, but in practice they compress down to about
4512      * 1100 bytes
4513      */
4514     numdb *= 1ULL << DN_MAX_INDBLKSHIFT;
4515     volsize += numdb;
4516     return (volsize);
4517 }
4518 unchanged_portion_omitted
```

```

*****
5439 Tue Oct 15 13:59:52 2013
new/usr/src/lib/libzfs/common/mapfile-vers
4012 Upper limit of zfs set bounds check for reservation on volumes is too lo
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 # Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
22 # Copyright 2011 Nexenta Systems, Inc. All rights reserved.
23 # Copyright (c) 2012 by Delphix. All rights reserved.
24 #
25 # MAPFILE HEADER START
26 #
27 # WARNING: STOP NOW. DO NOT MODIFY THIS FILE.
28 # Object versioning must comply with the rules detailed in
29 #
30 #     usr/src/lib/README.mapfiles
31 #
32 # You should not be making modifications here until you've read the most current
33 # copy of that file. If you need help, contact a gatekeeper for guidance.
34 #
35 # MAPFILE HEADER END
36 #
37
38 $mapfile_version 2
39
40 SYMBOL_VERSION SUNWprivate_1.1 {
41     global:
42         fletcher_2_native;
43         fletcher_2_byteswap;
44         fletcher_4_native;
45         fletcher_4_byteswap;
46         fletcher_4_incremental_native;
47         fletcher_4_incremental_byteswap;
48         libzfs_add_handle;
49         libzfs_dataset_cmp;
50         libzfs_errno;
51         libzfs_error_action;
52         libzfs_error_description;
53         libzfs_fini;
54         libzfs_fru_compare;
55         libzfs_fru_devpath;
56         libzfs_fru_lookup;
57         libzfs_fru_notself;
58         libzfs_fru_refresh;
59         libzfs_init;
60         libzfs_mnttab_cache;
61         libzfs_print_on_error;

```

```

62     spa_feature_table;
63     zfs_allocatable_devs;
64     zfs_asprintf;
65     zfs_clone;
66     zfs_close;
67     zfs_create;
68     zfs_create_ancestors;
69     zfs_dataset_exists;
70     zfs_deleg_share_nfs;
71     zfs_destroy;
72     zfs_destroy_snaps;
73     zfs_destroy_snaps_nvl;
74     zfs_expand_proplist;
75     zfs_get_handle;
76     zfs_get_holds;
77     zfs_get_name;
78     zfs_get_pool_handle;
79     zfs_get_user_props;
80     zfs_get_type;
81     zfs_handle_dup;
82     zfs_history_event_names;
83     zfs_hold;
84     zfs_is_mounted;
85     zfs_is_shared;
86     zfs_is_shared_nfs;
87     zfs_is_shared_smb;
88     zfs_iter_children;
89     zfs_iter_dependents;
90     zfs_iter_filesystems;
91     zfs_iter_root;
92     zfs_iter_snapshots;
93     zfs_iter_snapshots_sorted;
94     zfs_iter_snapspec;
95     zfs_mount;
96     zfs_name_to_prop;
97     zfs_name_valid;
98     zfs_nicenum;
99     zfs_nicestrtonum;
100    zfs_open;
101    zfs_path_to_zhandle;
102    zfs_promote;
103    zfs_prop_align_right;
104    zfs_prop_column_name;
105    zfs_prop_default_numeric;
106    zfs_prop_default_string;
107    zfs_prop_get;
108    zfs_prop_get_int;
109    zfs_prop_get_numeric;
110    zfs_prop_get_recvd;
111    zfs_prop_get_table;
112    zfs_prop_get_userquota_int;
113    zfs_prop_get_userquota;
114    zfs_prop_get_written_int;
115    zfs_prop_get_written;
116    zfs_prop_inherit;
117    zfs_prop_inheritable;
118    zfs_prop_init;
119    zfs_prop_is_string;
120    zfs_prop_readonly;
121    zfs_prop_set;
122    zfs_prop_string_to_index;
123    zfs_prop_to_name;
124    zfs_prop_user;
125    zfs_prop_userquota;
126    zfs_prop_valid_for_type;
127    zfs_prop_values;

```

```

128     zfs_prop_written;
129     zfs_prune_proplist;
130     zfs_receive;
131     zfs_refresh_properties;
132     zfs_release;
133     zfs_rename;
134     zfs_rollback;
135     zfs_save_arguments;
136     zfs_send;
137     zfs_share;
138     zfs_shareall;
139     zfs_share_nfs;
140     zfs_share_smb;
141     zfs_show_diffs;
142     zfs_smb_acl_add;
143     zfs_smb_acl_purge;
144     zfs_smb_acl_remove;
145     zfs_smb_acl_rename;
146     zfs_snapshot;
147     zfs_snapshot_nv1;
148     zfs_spa_version;
149     zfs_spa_version_map;
150     zfs_type_to_name;
151     zfs_unmount;
152     zfs_unmountall;
153     zfs_unshare;
154     zfs_unshare_nfs;
155     zfs_unshare_smb;
156     zfs_unshareall;
157     zfs_unshareall_bypath;
158     zfs_unshareall_nfs;
159     zfs_unshareall_smb;
160     zfs_userspace;
161     zfs_valid_proplist;
162     zfs_get_fsacl;
163     zfs_set_fsacl;
164     zfs_userquota_prop_prefixes;
165     zfs_zpl_version_map;
166     zpool_add;
167     zpool_clear;
168     zpool_clear_label;
169     zpool_close;
170     zpool_create;
171     zpool_destroy;
172     zpool_disable_datasets;
173     zpool_dump_ddt;
174     zpool_enable_datasets;
175     zpool_expand_proplist;
176     zpool_explain_recover;
177     zpool_export;
178     zpool_export_force;
179     zpool_find_import;
180     zpool_find_import_cached;
181     zpool_find_vdev;
182     zpool_find_vdev_by_physpath;
183     zpool_fru_set;
184     zpool_get_config;
185     zpool_get_errlog;
186     zpool_get_features;
187     zpool_get_handle;
188     zpool_get_history;
189     zpool_get_name;
190     zpool_get_physpath;
191     zpool_get_prop;
192     zpool_get_prop_int;
193     zpool_get_state;

```

```

194     zpool_get_status;
195     zpool_history_unpack;
196     zpool_import;
197     zpool_import_props;
198     zpool_import_status;
199     zpool_in_use;
200     zpool_is_bootable;
201     zpool_iter;
202     zpool_label_disk;
203     zpool_log_history;
204     zpool_mount_datasets;
205     zpool_name_to_prop;
206     zpool_obj_to_path;
207     zpool_open;
208     zpool_open_canfail;
209     zpool_print_unsup_feat;
210     zpool_prop_align_right;
211     zpool_prop_column_name;
212     zpool_prop_feature;
213     zpool_prop_get_feature;
214     zpool_prop_readonly;
215     zpool_prop_to_name;
216     zpool_prop_unsupported;
217     zpool_prop_values;
218     zpool_read_label;
219     zpool_refresh_stats;
220     zpool_reguid;
221     zpool_reopen;
222     zpool_scan;
223     zpool_search_import;
224     zpool_set_prop;
225     zpool_state_to_name;
226     zpool_unmount_datasets;
227     zpool_upgrade;
228     zpool_vdev_attach;
229     zpool_vdev_clear;
230     zpool_vdev_degrade;
231     zpool_vdev_detach;
232     zpool_vdev_fault;
233     zpool_vdev_name;
234     zpool_vdev_offline;
235     zpool_vdev_online;
236     zpool_vdev_remove;
237     zpool_vdev_split;
238     zprop_free_list;
239     zprop_get_list;
240     zprop_iter;
241     zprop_print_one_property;
242     zprop_width;
243     zvol_check_dump_config;
244     zvol_volsize_to_reservation;
245     zvol_volsize_to_reservation_impl;
246     local:
247         *;
248 };
_____unchanged_portion_omitted_____

```