```
**********************************************************
   34345 Tue Aug  6 21:14:49 2013
new/usr/src/cmd/beadm/beadm.c
*** NO COMMENTS ***
**********************************************************
/*
 * 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) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
 */

/*
 * System includes
 */

#include <assert.h>
#include <stdio.h>
#include <strings.h>
#include <libzfs.h>
#include <locale.h>
#include <langinfo.h>
#include <stdlib.h>
#include <wchar.h>
#include <sys/types.h>

#include "libbe.h"

#ifndef lint
#define _(x) gettext(x)
#else
#define _(x) (x)
#endif

#ifndef TEXT_DOMAIN
#define TEXT_DOMAIN "SYS_TEST"
#endif

#define DT_BUF_LEN (128)
#define NUM_COLS (6)

static int be_do_activate(int argc, char **argv);
static int be_do_create(int argc, char **argv);
```

```
static int be_do_destroy(int argc, char **argv);
static int be_do_list(int argc, char **argv);
static int be_do_mount(int argc, char **argv);
static int be_do_unmount(int argc, char **argv);
static int be_do_rename(int argc, char **argv);
static int be_do_rollback(int argc, char **argv);
static void usage(void);

/*
 * single column name/width output format description
 */
struct col_info {
        const char *col_name;
        size_t width;
};
______unchanged_portion_omitted_

static void
print_be_nodes(const char *be_name, boolean_t parsable, struct hdr_info *hdr,
    be_node_list_t *nodes)
{
        char buf[64];
        char datetime[DT_BUF_LEN];
        be_node_list_t  *cur_be;

        for (cur_be = nodes; cur_be != NULL; cur_be = cur_be->be_next_node) {
                char active[3] = "-\0";
                int ai = 0;
                const char *datetime_fmt = "%F %R";
                const char *name = cur_be->be_node_name;
                const char *mntpt = cur_be->be_mntpt;
                be_snapshot_list_t *snap = NULL;
                uint64_t used = cur_be->be_space_used;
                time_t creation = cur_be->be_node_creation;
                struct tm *tm;

                if (be_name != NULL && strcmp(be_name, name) != 0)
                        continue;

                if (parsable)
                        active[0] = '\0';

                tm = localtime(&creation);
                (void) strftime(datetime, DT_BUF_LEN, datetime_fmt, tm);

                for (snap = cur_be->be_node_snapshots; snap != NULL;
                    snap = snap->be_next_snapshot)
                        used += snap->be_snapshot_space_used;

                if (!cur_be->be_global_active)
                        active[ai++] = 'x';

                if (cur_be->be_active)
                        active[ai++] = 'N';
                if (cur_be->be_active_on_boot) {
                        if (!cur_be->be_global_active)
                                active[ai] = 'b';
                        else
                if (cur_be->be_active_on_boot)
                                active[ai] = 'R';
                }

                nicenum(used, buf, sizeof (buf));
                if (parsable)
                        (void) printf("%s;%s;%s;%s;%llu;%s;%ld\n",
                            name,
                            cur_be->be_uuid_str,
                            active,
                            (cur_be->be_mounted ? mntpt: ""),
                            used,
                            cur_be->be_policy_type,
                            creation);
                else
                        (void) printf("%-*s %-*s %-*s %-*s %-*s %-*s\n",
                            hdr->cols[0].width, name,
                            hdr->cols[1].width, active,
                            hdr->cols[2].width, (cur_be->be_mounted ? mntpt:
                            "-"),
                            hdr->cols[3].width, buf,
                            hdr->cols[4].width, cur_be->be_policy_type,
                            hdr->cols[5].width, datetime);
        }
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   34518 Tue Aug  6 21:14:50 2013
new/usr/src/lib/libbe/common/be_activate.c
*** NO COMMENTS ***
**********************************************************
```

```c
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 */

#include <assert.h>
#include <libintl.h>
#include <libnvpair.h>
#include <libzfs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/mnttab.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include <libbe.h>
#include <libbe_priv.h>

char    *mnttab = MNTTAB;

/*
 * Private function prototypes
 */
static int set_bootfs(char *boot_rpool, char *be_root_ds);
static int set_canmount(be_node_list_t *, char *);
static int be_do_installgrub(be_transaction_data_t *);
static int be_get_grub_vers(be_transaction_data_t *, char **, char **);
static int get_ver_from_capfile(char *, char **);
static int be_promote_zone_ds(char *, char *);
static int be_promote_ds_callback(zfs_handle_t *, void *);

/* ******************************************************************** */
/*                      Public Functions                                */
/* ******************************************************************** */
```

```c
/*
 * Function:    be_activate
 * Description: Calls _be_activate which activates the BE named in the
 *              attributes passed in through be_attrs. The process of
 *              activation sets the bootfs property of the root pool, resets
 *              the canmount property to noauto, and sets the default in the
 *              grub menu to the entry corresponding to the entry for the named
 *              BE.
 * Parameters:
 *              be_attrs - pointer to nvlist_t of attributes being passed in.
 *                      The follow attribute values are used by this function:
 *
 *                      BE_ATTR_ORIG_BE_NAME            *required
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Public
 */
int
be_activate(nvlist_t *be_attrs)
{
        int     ret = BE_SUCCESS;
        char    *be_name = NULL;

        /* Initialize libzfs handle */
        if (!be_zfs_init())
                return (BE_ERR_INIT);

        /* Get the BE name to activate */
        if (nvlist_lookup_string(be_attrs, BE_ATTR_ORIG_BE_NAME, &be_name)
            != 0) {
                be_print_err(gettext("be_activate: failed to "
                    "lookup BE_ATTR_ORIG_BE_NAME attribute\n"));
                be_zfs_fini();
                return (BE_ERR_INVAL);
        }

        /* Validate BE name */
        if (!be_valid_be_name(be_name)) {
                be_print_err(gettext("be_activate: invalid BE name %s\n"),
                    be_name);
                be_zfs_fini();
                return (BE_ERR_INVAL);
        }

        ret = _be_activate(be_name);

        be_zfs_fini();

        return (ret);
}

/* ******************************************************************** */
/*                      Semi Private Functions                          */
/* ******************************************************************** */

/*
 * Function:    _be_activate
 * Description: This does the actual work described in be_activate.
 * Parameters:
 *              be_name - pointer to the name of BE to activate.
 *
 * Return:
 *              BE_SUCCESS - Success
 *              be_errnot_t - Failure
 * Scope:
 *              Public
 */
int
_be_activate(char *be_name)
{
        be_transaction_data_t cb = { 0 };
        zfs_handle_t    *zhp = NULL;
        char            root_ds[MAXPATHLEN];
        char            active_ds[MAXPATHLEN];
        char            *cur_vers = NULL, *new_vers = NULL;
        be_node_list_t  *be_nodes = NULL;
        uuid_t          uu = {0};
        int             entry, ret = BE_SUCCESS;
        int             zret = 0;

        /*
         * TODO: The BE needs to be validated to make sure that it is actually
         * a bootable BE.
         */

        if (be_name == NULL)
                return (BE_ERR_INVAL);

        /* Set obe_name to be_name in the cb structure */
        cb.obe_name = be_name;

        /* find which zpool the be is in */
        if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &cb)) == 0) {
                be_print_err(gettext("be_activate: failed to "
                    "find zpool for BE (%s)\n"), cb.obe_name);
                return (BE_ERR_BE_NOENT);
        } else if (zret < 0) {
                be_print_err(gettext("be_activate: "
                    "zpool_iter failed: %s\n"),
                    libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
                return (ret);
        }

        be_make_root_ds(cb.obe_zpool, cb.obe_name, root_ds, sizeof (root_ds));
        cb.obe_root_ds = strdup(root_ds);

        if (getzoneid() == GLOBAL_ZONEID) {
                if (be_has_grub() && (ret = be_get_grub_vers(&cb, &cur_vers,
                    &new_vers)) != BE_SUCCESS) {
                        be_print_err(gettext("be_activate: failed to get grub "
                            "versions from capability files.\n"));
                        return (ret);
                }
                if (cur_vers != NULL) {
                        /*
                         * We need to check to see if the version number from
                         * the BE being activated is greater than the current
                         * one.
                         */
                        if (new_vers != NULL &&
                            atof(cur_vers) < atof(new_vers)) {
                                if ((ret = be_do_installgrub(&cb))
                                    != BE_SUCCESS) {
                                        free(new_vers);
                                        free(cur_vers);
                                        return (ret);
                                }
                                free(new_vers);
```

```
                                }
                                free(cur_vers);
                } else if (new_vers != NULL) {
                        if ((ret = be_do_installgrub(&cb)) != BE_SUCCESS) {
                                free(new_vers);
                                return (ret);
                        }
                        free(new_vers);
                }
                if (!be_has_menu_entry(root_ds, cb.obe_zpool, &entry)) {
                        if ((ret = be_append_menu(cb.obe_name, cb.obe_zpool,
                            NULL, NULL, NULL)) != BE_SUCCESS) {
                                be_print_err(gettext("be_activate: Failed to "
                                    "add BE (%s) to the GRUB menu\n"),
                                    cb.obe_name);
                                goto done;
                        }
                }
                if (be_has_grub()) {
                        if ((ret = be_change_grub_default(cb.obe_name,
                            cb.obe_zpool)) != BE_SUCCESS) {
                                be_print_err(gettext("be_activate: failed to "
                                    "change the default entry in menu.lst\n"));
                                goto done;
                        }
                }
        }

        if ((ret = _be_list(cb.obe_name, &be_nodes)) != BE_SUCCESS) {
                return (ret);
        }

        if ((ret = set_canmount(be_nodes, "noauto")) != BE_SUCCESS) {
                be_print_err(gettext("be_activate: failed to set "
                    "canmount dataset property\n"));
                goto done;
        }

        if (getzoneid() == GLOBAL_ZONEID) {
                if ((ret = set_bootfs(be_nodes->be_rpool,
                    root_ds)) != BE_SUCCESS) {
        if ((ret = set_bootfs(be_nodes->be_rpool, root_ds)) != BE_SUCCESS) {
                        be_print_err(gettext("be_activate: failed to set "
                            "bootfs pool property for %s\n"), root_ds);
                        goto done;
                }
        }

        if ((zhp = zfs_open(g_zfs, root_ds, ZFS_TYPE_FILESYSTEM)) != NULL) {
                /*
                 * We don't need to close the zfs handle at this
                 * point because The callback funtion
                 * be_promote_ds_callback() will close it for us.
                 */
                if (be_promote_ds_callback(zhp, NULL) != 0) {
                        be_print_err(gettext("be_activate: "
                            "failed to activate the "
                            "datasets for %s: %s\n"),
                            root_ds,
                            libzfs_error_description(g_zfs));
                        ret = BE_ERR_PROMOTE;
                        goto done;
                }
        } else {
                be_print_err(gettext("be_activate: failed to open "
                be_print_err(gettext("be_activate:: failed to open "
                    "dataset (%s): %s\n"), root_ds,
                    libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
                goto done;
        }

        if (getzoneid() == GLOBAL_ZONEID &&
            be_get_uuid(cb.obe_root_ds, &uu) == BE_SUCCESS &&
            (ret = be_promote_zone_ds(cb.obe_name, cb.obe_root_ds))
            != BE_SUCCESS) {
                be_print_err(gettext("be_activate: failed to promote "
                    "the active zonepath datasets for zones in BE %s\n"),
                    cb.obe_name);
        }

        if (getzoneid() != GLOBAL_ZONEID) {
                if (!be_zone_compare_uuids(root_ds)) {
                        be_print_err(gettext("be_activate: activating zone "
                            "root dataset from non-active global BE is not "
                            "supported\n"));
                        ret = BE_ERR_NOTSUP;
                        goto done;
                }
                if ((zhp = zfs_open(g_zfs, root_ds,
                    ZFS_TYPE_FILESYSTEM)) == NULL) {
                        be_print_err(gettext("be_activate: failed to open "
                            "dataset (%s): %s\n"), root_ds,
                            libzfs_error_description(g_zfs));
                        ret = zfs_err_to_be_err(g_zfs);
                        goto done;
                }
                /* Find current active zone root dataset */
                if ((ret = be_find_active_zone_root(zhp, cb.obe_zpool,
                    active_ds, sizeof (active_ds))) != BE_SUCCESS) {
                        be_print_err(gettext("be_activate: failed to find "
                            "active zone root dataset\n"));
                        ZFS_CLOSE(zhp);
                        goto done;
                }
                /* Do nothing if requested BE is already active */
                if (strcmp(root_ds, active_ds) == 0) {
                        ret = BE_SUCCESS;
                        ZFS_CLOSE(zhp);
                        goto done;
                }

                /* Set active property for BE */
                if (zfs_prop_set(zhp, BE_ZONE_ACTIVE_PROPERTY, "on") != 0) {
                        be_print_err(gettext("be_activate: failed to set "
                            "active property (%s): %s\n"), root_ds,
                            libzfs_error_description(g_zfs));
                        ret = zfs_err_to_be_err(g_zfs);
                        ZFS_CLOSE(zhp);
                        goto done;
                }
                ZFS_CLOSE(zhp);

                /* Unset active property for old active root dataset */
                if ((zhp = zfs_open(g_zfs, active_ds,
                    ZFS_TYPE_FILESYSTEM)) == NULL) {
                        be_print_err(gettext("be_activate: failed to open "
                            "dataset (%s): %s\n"), active_ds,
                            libzfs_error_description(g_zfs));
                        ret = zfs_err_to_be_err(g_zfs);
                        goto done;
                }
```

```
                if (zfs_prop_set(zhp, BE_ZONE_ACTIVE_PROPERTY, "off") != 0) {
                        be_print_err(gettext("be_activate: failed to unset "
                            "active property (%s): %s\n"), active_ds,
                            libzfs_error_description(g_zfs));
                        ret = zfs_err_to_be_err(g_zfs);
                        ZFS_CLOSE(zhp);
                        goto done;
                }
                ZFS_CLOSE(zhp);
        }
done:
        be_free_list(be_nodes);
        return (ret);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   86490 Tue Aug  6 21:14:51 2013
new/usr/src/lib/libbe/common/be_create.c
*** NO COMMENTS ***
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 */

/*
 * System includes
 */

#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <libgen.h>
#include <libintl.h>
#include <libnvpair.h>
#include <libzfs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mnttab.h>
#include <sys/mount.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#include <libbe.h>
#include <libbe_priv.h>

/* Library wide variables */
libzfs_handle_t *g_zfs = NULL;

/* Private function prototypes */
static int _be_destroy(const char *, be_destroy_data_t *);
static int be_destroy_zones(char *, char *, be_destroy_data_t *);
static int be_destroy_zone_roots(char *, be_destroy_data_t *);
```

```
static int be_destroy_zone_roots_callback(zfs_handle_t *, void *);
static int be_copy_zones(char *, char *, char *);
static int be_clone_fs_callback(zfs_handle_t *, void *);
static int be_destroy_callback(zfs_handle_t *, void *);
static int be_send_fs_callback(zfs_handle_t *, void *);
static int be_demote_callback(zfs_handle_t *, void *);
static int be_demote_find_clone_callback(zfs_handle_t *, void *);
static int be_has_snapshot_callback(zfs_handle_t *, void *);
static int be_demote_get_one_clone(zfs_handle_t *, void *);
static int be_get_snap(char *, char **);
static int be_prep_clone_send_fs(zfs_handle_t *, be_transaction_data_t *,
    char *, int);
static boolean_t be_create_container_ds(char *);
static char *be_get_zone_be_name(char *root_ds, char *container_ds);
static int be_zone_root_exists_callback(zfs_handle_t *, void *);

/* ******************************************************************** */
/*                      Public Functions                                */
/* ******************************************************************** */

/*
 * Function:    be_init
 * Description: Creates the initial datasets for a BE and leaves them
 *              unpopulated.  The resultant BE can be mounted but can't
 *              yet be activated or booted.
 * Parameters:
 *              be_attrs - pointer to nvlist_t of attributes being passed in.
 *                      The following attributes are used by this function:
 *
 *                      BE_ATTR_NEW_BE_NAME             *required
 *                      BE_ATTR_NEW_BE_POOL             *required
 *                      BE_ATTR_ZFS_PROPERTIES          *optional
 *                      BE_ATTR_FS_NAMES                *optional
 *                      BE_ATTR_FS_NUM                  *optional
 *                      BE_ATTR_SHARED_FS_NAMES         *optional
 *                      BE_ATTR_SHARED_FS_NUM           *optional
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Public
 */
int
be_init(nvlist_t *be_attrs)
{
        be_transaction_data_t   bt = { 0 };
        zpool_handle_t  *zlp;
        nvlist_t        *zfs_props = NULL;
        char            nbe_root_ds[MAXPATHLEN];
        char            child_fs[MAXPATHLEN];
        char            **fs_names = NULL;
        char            **shared_fs_names = NULL;
        uint16_t        fs_num = 0;
        uint16_t        shared_fs_num = 0;
        int             nelem;
        int             i;
        int             zret = 0, ret = BE_SUCCESS;

        /* Initialize libzfs handle */
        if (!be_zfs_init())
                return (BE_ERR_INIT);

        /* Get new BE name */
        if (nvlist_lookup_string(be_attrs, BE_ATTR_NEW_BE_NAME, &bt.nbe_name)
            != 0) {
                be_print_err(gettext("be_init: failed to lookup "
                    "BE_ATTR_NEW_BE_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }

        /* Validate new BE name */
        if (!be_valid_be_name(bt.nbe_name)) {
                be_print_err(gettext("be_init: invalid BE name %s\n"),
                    bt.nbe_name);
                return (BE_ERR_INVAL);
        }

        /* Get zpool name */
        if (nvlist_lookup_string(be_attrs, BE_ATTR_NEW_BE_POOL, &bt.nbe_zpool)
            != 0) {
                be_print_err(gettext("be_init: failed to lookup "
                    "BE_ATTR_NEW_BE_POOL attribute\n"));
                return (BE_ERR_INVAL);
        }

        /* Get file system attributes */
        nelem = 0;
        if (nvlist_lookup_pairs(be_attrs, 0,
            BE_ATTR_FS_NUM, DATA_TYPE_UINT16, &fs_num,
            BE_ATTR_FS_NAMES, DATA_TYPE_STRING_ARRAY, &fs_names, &nelem,
            NULL) != 0) {
                be_print_err(gettext("be_init: failed to lookup fs "
                    "attributes\n"));
                return (BE_ERR_INVAL);
        }
        if (nelem != fs_num) {
                be_print_err(gettext("be_init: size of FS_NAMES array (%d) "
                    "does not match FS_NUM (%d)\n"), nelem, fs_num);
                return (BE_ERR_INVAL);
        }

        /* Get shared file system attributes */
        nelem = 0;
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_SHARED_FS_NUM, DATA_TYPE_UINT16, &shared_fs_num,
            BE_ATTR_SHARED_FS_NAMES, DATA_TYPE_STRING_ARRAY, &shared_fs_names,
            &nelem, NULL) != 0) {
                be_print_err(gettext("be_init: failed to lookup "
                    "shared fs attributes\n"));
                return (BE_ERR_INVAL);
        }
        if (nelem != shared_fs_num) {
                be_print_err(gettext("be_init: size of SHARED_FS_NAMES "
                    "array does not match SHARED_FS_NUM\n"));
                return (BE_ERR_INVAL);
        }

        /* Verify that nbe_zpool exists */
        if ((zlp = zpool_open(g_zfs, bt.nbe_zpool)) == NULL) {
                be_print_err(gettext("be_init: failed to "
                    "find existing zpool (%s): %s\n"), bt.nbe_zpool,
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }
        zpool_close(zlp);

        /*
         * Verify BE container dataset in nbe_zpool exists.
         * If not, create it.
         */
        if (!be_create_container_ds(bt.nbe_zpool))
                return (BE_ERR_CREATDS);
```

```

        /*
         * Verify that nbe_name doesn't already exist in some pool.
         */
        if ((zret = zpool_iter(g_zfs, be_exists_callback, bt.nbe_name)) > 0) {
                be_print_err(gettext("be_init: BE (%s) already exists\n"),
                    bt.nbe_name);
                return (BE_ERR_BE_EXISTS);
        } else if (zret < 0) {
                be_print_err(gettext("be_init: zpool_iter failed: %s\n"),
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /* Generate string for BE's root dataset */
        be_make_root_ds(bt.nbe_zpool, bt.nbe_name, nbe_root_ds,
            sizeof (nbe_root_ds));

        /*
         * Create property list for new BE root dataset.  If some
         * zfs properties were already provided by the caller, dup
         * that list.  Otherwise initialize a new property list.
         */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_ZFS_PROPERTIES, DATA_TYPE_NVLIST, &zfs_props, NULL)
            != 0) {
                be_print_err(gettext("be_init: failed to lookup "
                    "BE_ATTR_ZFS_PROPERTIES attribute\n"));
                return (BE_ERR_INVAL);
        }
        if (zfs_props != NULL) {
                /* Make sure its a unique nvlist */
                if (!(zfs_props->nvl_nvflag & NV_UNIQUE_NAME) &&
                    !(zfs_props->nvl_nvflag & NV_UNIQUE_NAME_TYPE)) {
                        be_print_err(gettext("be_init: ZFS property list "
                            "not unique\n"));
                        return (BE_ERR_INVAL);
                }

                /* Dup the list */
                if (nvlist_dup(zfs_props, &bt.nbe_zfs_props, 0) != 0) {
                        be_print_err(gettext("be_init: failed to dup ZFS "
                            "property list\n"));
                        return (BE_ERR_NOMEM);
                }
        } else {
                /* Initialize new nvlist */
                if (nvlist_alloc(&bt.nbe_zfs_props, NV_UNIQUE_NAME, 0) != 0) {
                        be_print_err(gettext("be_init: internal "
                            "error: out of memory\n"));
                        return (BE_ERR_NOMEM);
                }
        }

        /* Set the mountpoint property for the root dataset */
        if (nvlist_add_string(bt.nbe_zfs_props,
            zfs_prop_to_name(ZFS_PROP_MOUNTPOINT), "/") != 0) {
                be_print_err(gettext("be_init: internal error "
                    "out of memory\n"));
                ret = BE_ERR_NOMEM;
                goto done;
        }

        /* Set the 'canmount' property */
        if (nvlist_add_string(bt.nbe_zfs_props,
            zfs_prop_to_name(ZFS_PROP_CANMOUNT), "noauto") != 0) {
                be_print_err(gettext("be_init: internal error "
                    "out of memory\n"));
                ret = BE_ERR_NOMEM;
                goto done;
        }

        /* Create BE root dataset for the new BE */
        if (zfs_create(g_zfs, nbe_root_ds, ZFS_TYPE_FILESYSTEM,
            bt.nbe_zfs_props) != 0) {
                be_print_err(gettext("be_init: failed to "
                    "create BE root dataset (%s): %s\n"), nbe_root_ds,
                    libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
                goto done;
        }

        /* Set UUID for new BE */
        if ((ret = be_set_uuid(nbe_root_ds)) != BE_SUCCESS) {
                be_print_err(gettext("be_init: failed to "
                    "set uuid for new BE\n"));
        }

        /*
         * Clear the mountpoint property so that the non-shared
         * file systems created below inherit their mountpoints.
         */
        (void) nvlist_remove(bt.nbe_zfs_props,
            zfs_prop_to_name(ZFS_PROP_MOUNTPOINT), DATA_TYPE_STRING);

        /* Create the new BE's non-shared file systems */
        for (i = 0; i < fs_num && fs_names[i]; i++) {
                /*
                 * If fs == "/", skip it;
                 * we already created the root dataset
                 */
                if (strcmp(fs_names[i], "/") == 0)
                        continue;

                /* Generate string for file system */
                (void) snprintf(child_fs, sizeof (child_fs), "%s%s",
                    nbe_root_ds, fs_names[i]);

                /* Create file system */
                if (zfs_create(g_zfs, child_fs, ZFS_TYPE_FILESYSTEM,
                    bt.nbe_zfs_props) != 0) {
                        be_print_err(gettext("be_init: failed to create "
                            "BE's child dataset (%s): %s\n"), child_fs,
                            libzfs_error_description(g_zfs));
                        ret = zfs_err_to_be_err(g_zfs);
                        goto done;
                }
        }

        /* Create the new BE's shared file systems */
        if (shared_fs_num > 0) {
                nvlist_t        *props = NULL;

                if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0) {
                        be_print_err(gettext("be_init: nvlist_alloc failed\n"));
                        ret = BE_ERR_NOMEM;
                        goto done;
                }

                for (i = 0; i < shared_fs_num; i++) {
                        /* Generate string for shared file system */
                        (void) snprintf(child_fs, sizeof (child_fs), "%s%s",
```

```
                            bt.nbe_zpool, shared_fs_names[i]);

                        if (nvlist_add_string(props,
                            zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
                            shared_fs_names[i]) != 0) {
                                be_print_err(gettext("be_init: "
                                    "internal error: out of memory\n"));
                                nvlist_free(props);
                                ret = BE_ERR_NOMEM;
                                goto done;
                        }

                        /* Create file system if it doesn't already exist */
                        if (zfs_dataset_exists(g_zfs, child_fs,
                            ZFS_TYPE_FILESYSTEM)) {
                                continue;
                        }
                        if (zfs_create(g_zfs, child_fs, ZFS_TYPE_FILESYSTEM,
                            props) != 0) {
                                be_print_err(gettext("be_init: failed to "
                                    "create BE's shared dataset (%s): %s\n"),
                                    child_fs, libzfs_error_description(g_zfs));
                                ret = zfs_err_to_be_err(g_zfs);
                                nvlist_free(props);
                                goto done;
                        }
                }

                nvlist_free(props);
        }

done:
        if (bt.nbe_zfs_props != NULL)
                nvlist_free(bt.nbe_zfs_props);

        be_zfs_fini();

        return (ret);
}

/*
 * Function:    be_destroy
 * Description: Destroy a BE and all of its children datasets, snapshots and
 *              zones that belong to the parent BE.
 * Parameters:
 *              be_attrs - pointer to nvlist_t of attributes being passed in.
 *                      The following attributes are used by this function:
 *
 *                      BE_ATTR_ORIG_BE_NAME            *required
 *                      BE_ATTR_DESTROY_FLAGS           *optional
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Public
 */
int
be_destroy(nvlist_t *be_attrs)
{
        zfs_handle_t            *zhp = NULL;
        be_transaction_data_t   bt = { 0 };
        be_transaction_data_t   cur_bt = { 0 };
        be_destroy_data_t       dd = { 0 };
        int                     ret = BE_SUCCESS;
        uint16_t                flags = 0;
        boolean_t               bs_found = B_FALSE;
        int                     zret;
        char                    obe_root_ds[MAXPATHLEN];
        char                    *mp = NULL;

        /* Initialize libzfs handle */
        if (!be_zfs_init())
                return (BE_ERR_INIT);

        /* Get name of BE to delete */
        if (nvlist_lookup_string(be_attrs, BE_ATTR_ORIG_BE_NAME, &bt.obe_name)
            != 0) {
                be_print_err(gettext("be_destroy: failed to lookup "
                    "BE_ATTR_ORIG_BE_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }

        /*
         * Validate BE name. If valid, then check that the original BE is not
         * the active BE. If it is the 'active' BE then return an error code
         * since we can't destroy the active BE.
         */
        if (!be_valid_be_name(bt.obe_name)) {
                be_print_err(gettext("be_destroy: invalid BE name %s\n"),
                    bt.obe_name);
                return (BE_ERR_INVAL);
        } else if (bt.obe_name != NULL) {
                if ((ret = be_find_current_be(&cur_bt)) != BE_SUCCESS) {
                        return (ret);
                }
                if (strcmp(cur_bt.obe_name, bt.obe_name) == 0) {
                        return (BE_ERR_DESTROY_CURR_BE);
                }
        }

        /* Get destroy flags if provided */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_DESTROY_FLAGS, DATA_TYPE_UINT16, &flags, NULL)
            != 0) {
                be_print_err(gettext("be_destroy: failed to lookup "
                    "BE_ATTR_DESTROY_FLAGS attribute\n"));
                return (BE_ERR_INVAL);
        }

        dd.destroy_snaps = flags & BE_DESTROY_FLAG_SNAPSHOTS;
        dd.force_unmount = flags & BE_DESTROY_FLAG_FORCE_UNMOUNT;

        /* Find which zpool obe_name lives in */
        if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) == 0) {
                be_print_err(gettext("be_destroy: failed to find zpool "
                    "for BE (%s)\n"), bt.obe_name);
                return (BE_ERR_BE_NOENT);
        } else if (zret < 0) {
                be_print_err(gettext("be_destroy: zpool_iter failed: %s\n"),
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /* Generate string for obe_name's root dataset */
        be_make_root_ds(bt.obe_zpool, bt.obe_name, obe_root_ds,
            sizeof (obe_root_ds));
        bt.obe_root_ds = obe_root_ds;

        if (getzoneid() != GLOBAL_ZONEID) {
                if (!be_zone_compare_uuids(bt.obe_root_ds)) {
                        if (be_is_active_on_boot(bt.obe_name)) {
                                be_print_err(gettext("be_destroy: destroying "
```

```
                                    "active zone root dataset from non-active "
                                    "global BE is not supported\n"));
                                return (BE_ERR_NOTSUP);
                        }
                }
        }

        /*
         * Detect if the BE to destroy has the 'active on boot' property set.
         * If so, set the 'active on boot' property on the the 'active' BE.
         */
        if (be_is_active_on_boot(bt.obe_name)) {
                if ((ret = be_activate_current_be()) != BE_SUCCESS) {
                        be_print_err(gettext("be_destroy: failed to "
                            "make the current BE 'active on boot'\n"));
                        return (ret);
                }
        }

        /* Get handle to BE's root dataset */
        if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_FILESYSTEM)) ==
            NULL) {
                be_print_err(gettext("be_destroy: failed to "
                    "open BE root dataset (%s): %s\n"), bt.obe_root_ds,
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /*
         * Check if BE has snapshots and BE_DESTROY_FLAG_SNAPSHOTS
         * is not set.
         */
        (void) zfs_iter_snapshots(zhp, be_has_snapshot_callback, &bs_found);
        if (!dd.destroy_snaps && bs_found) {
                ZFS_CLOSE(zhp);
                return (BE_ERR_SS_EXISTS);
        }

        /* Get the UUID of the global BE */
        if (getzoneid() == GLOBAL_ZONEID) {
                if (be_get_uuid(zfs_get_name(zhp),
                    &dd.gz_be_uuid) != BE_SUCCESS) {
                        be_print_err(gettext("be_destroy: BE has no "
                        "UUID (%s)\n"), zfs_get_name(zhp));
        if (be_get_uuid(zfs_get_name(zhp), &dd.gz_be_uuid) != BE_SUCCESS) {
                be_print_err(gettext("be_destroy: BE has no UUID (%s)\n"),
                    zfs_get_name(zhp));
                }
        }

        /*
         * If the global BE is mounted, make sure we've been given the
         * flag to forcibly unmount it.
         */
        if (zfs_is_mounted(zhp, &mp)) {
                if (!(dd.force_unmount)) {
                        be_print_err(gettext("be_destroy: "
                            "%s is currently mounted at %s, cannot destroy\n"),
                            bt.obe_name, mp != NULL ? mp : "<unknown>");

                        free(mp);
                        ZFS_CLOSE(zhp);
                        return (BE_ERR_MOUNTED);
                }
                free(mp);
        }
        /*
         * Destroy the non-global zone BE's if we are in the global zone
         * and there is a UUID associated with the global zone BE
         */
        if (getzoneid() == GLOBAL_ZONEID && !uuid_is_null(dd.gz_be_uuid)) {
                if ((ret = be_destroy_zones(bt.obe_name, bt.obe_root_ds, &dd))
                    != BE_SUCCESS) {
                        be_print_err(gettext("be_destroy: failed to "
                            "destroy one or more zones for BE %s\n"),
                            bt.obe_name);
                        goto done;
                }
        }

        /* Unmount the BE if it was mounted */
        if (zfs_is_mounted(zhp, NULL)) {
                if ((ret = _be_unmount(bt.obe_name, BE_UNMOUNT_FLAG_FORCE))
                    != BE_SUCCESS) {
                        be_print_err(gettext("be_destroy: "
                            "failed to unmount %s\n"), bt.obe_name);
                        ZFS_CLOSE(zhp);
                        return (ret);
                }
        }
        ZFS_CLOSE(zhp);

        /* Destroy this BE */
        if ((ret = _be_destroy((const char *)bt.obe_root_ds, &dd))
            != BE_SUCCESS) {
                goto done;
        }

        /* Remove BE's entry from the boot menu */
        if (getzoneid() == GLOBAL_ZONEID) {
                if ((ret = be_remove_menu(bt.obe_name, bt.obe_zpool, NULL))
                    != BE_SUCCESS) {
                        be_print_err(gettext("be_destroy: failed to "
                            "remove BE %s from the boot menu\n"),
                            bt.obe_root_ds);
                        goto done;
                }
        }

done:
        be_zfs_fini();

        return (ret);
}

/*
 * Function:    be_copy
 * Description: This function makes a copy of an existing BE.  If the original
 *              BE and the new BE are in the same pool, it uses zfs cloning to
 *              create the new BE, otherwise it does a physical copy.
 *              If the original BE name isn't provided, it uses the currently
 *              booted BE.  If the new BE name isn't provided, it creates an
 *              auto named BE and returns that name to the caller.
 * Parameters:
 *              be_attrs - pointer to nvlist_t of attributes being passed in.
 *                      The following attributes are used by this function:
 *
 *                      BE_ATTR_ORIG_BE_NAME            *optional
 *                      BE_ATTR_SNAP_NAME               *optional
 *                      BE_ATTR_NEW_BE_NAME             *optional
 *                      BE_ATTR_NEW_BE_POOL             *optional
```

```
 *                      BE_ATTR_NEW_BE_DESC             *optional
 *                      BE_ATTR_ZFS_PROPERTIES          *optional
 *                      BE_ATTR_POLICY                  *optional
 *
 *                      If the BE_ATTR_NEW_BE_NAME was not passed in, upon
 *                      successful BE creation, the following attribute values
 *                      will be returned to the caller by setting them in the
 *                      be_attrs parameter passed in:
 *
 *                      BE_ATTR_SNAP_NAME
 *                      BE_ATTR_NEW_BE_NAME
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Public
 */
int
be_copy(nvlist_t *be_attrs)
{
        be_transaction_data_t   bt = { 0 };
        be_fs_list_data_t       fld = { 0 };
        zfs_handle_t    *zhp = NULL;
        zpool_handle_t  *zphp = NULL;
        nvlist_t        *zfs_props = NULL;
        uuid_t          uu = { 0 };
        uuid_t          parent_uu = { 0 };
        char            obe_root_ds[MAXPATHLEN];
        char            nbe_root_ds[MAXPATHLEN];
        char            ss[MAXPATHLEN];
        char            *new_mp = NULL;
        char            *obe_name = NULL;
        boolean_t       autoname = B_FALSE;
        boolean_t       be_created = B_FALSE;
        int             i;
        int             zret;
        int             ret = BE_SUCCESS;
        struct be_defaults be_defaults;

        /* Initialize libzfs handle */
        if (!be_zfs_init())
                return (BE_ERR_INIT);

        /* Get original BE name */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_ORIG_BE_NAME, DATA_TYPE_STRING, &obe_name, NULL) != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_ORIG_BE_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }

        if ((ret = be_find_current_be(&bt)) != BE_SUCCESS) {
                return (ret);
        }

        be_get_defaults(&be_defaults);

        /* If original BE name not provided, use current BE */
        if (obe_name != NULL) {
                bt.obe_name = obe_name;
                /* Validate original BE name */
                if (!be_valid_be_name(bt.obe_name)) {
                        be_print_err(gettext("be_copy: "
                            "invalid BE name %s\n"), bt.obe_name);
                        return (BE_ERR_INVAL);
                }
        }

        if (be_defaults.be_deflt_rpool_container) {
                if ((zphp = zpool_open(g_zfs, bt.obe_zpool)) == NULL) {
                        be_print_err(gettext("be_get_node_data: failed to "
                            "open rpool (%s): %s\n"), bt.obe_zpool,
                            libzfs_error_description(g_zfs));
                        return (zfs_err_to_be_err(g_zfs));
                }
                if (be_find_zpool_callback(zphp, &bt) == 0) {
                        return (BE_ERR_BE_NOENT);
                }
        } else {
                /* Find which zpool obe_name lives in */
                if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) ==
                    0) {
                        be_print_err(gettext("be_copy: failed to "
                            "find zpool for BE (%s)\n"), bt.obe_name);
                        return (BE_ERR_BE_NOENT);
                } else if (zret < 0) {
                        be_print_err(gettext("be_copy: "
                            "zpool_iter failed: %s\n"),
                            libzfs_error_description(g_zfs));
                        return (zfs_err_to_be_err(g_zfs));
                }
        }

        /* Get snapshot name of original BE if one was provided */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_SNAP_NAME, DATA_TYPE_STRING, &bt.obe_snap_name, NULL)
            != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_SNAP_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }

        /* Get new BE name */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_NEW_BE_NAME, DATA_TYPE_STRING, &bt.nbe_name, NULL)
            != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_NEW_BE_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }

        /* Get zpool name to create new BE in */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_NEW_BE_POOL, DATA_TYPE_STRING, &bt.nbe_zpool, NULL) != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_NEW_BE_POOL attribute\n"));
                return (BE_ERR_INVAL);
        }

        /* Get new BE's description if one was provided */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_NEW_BE_DESC, DATA_TYPE_STRING, &bt.nbe_desc, NULL) != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_NEW_BE_DESC attribute\n"));
                return (BE_ERR_INVAL);
        }

        /* Get BE policy to create this snapshot under */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_POLICY, DATA_TYPE_STRING, &bt.policy, NULL) != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_POLICY attribute\n"));
```

```
                return (BE_ERR_INVAL);
        }

        /*
         * Create property list for new BE root dataset.  If some
         * zfs properties were already provided by the caller, dup
         * that list.  Otherwise initialize a new property list.
         */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_ZFS_PROPERTIES, DATA_TYPE_NVLIST, &zfs_props, NULL)
            != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_ZFS_PROPERTIES attribute\n"));
                return (BE_ERR_INVAL);
        }
        if (zfs_props != NULL) {
                /* Make sure its a unique nvlist */
                if (!(zfs_props->nvl_nvflag & NV_UNIQUE_NAME) &&
                    !(zfs_props->nvl_nvflag & NV_UNIQUE_NAME_TYPE)) {
                        be_print_err(gettext("be_copy: ZFS property list "
                            "not unique\n"));
                        return (BE_ERR_INVAL);
                }

                /* Dup the list */
                if (nvlist_dup(zfs_props, &bt.nbe_zfs_props, 0) != 0) {
                        be_print_err(gettext("be_copy: "
                            "failed to dup ZFS property list\n"));
                        return (BE_ERR_NOMEM);
                }
        } else {
                /* Initialize new nvlist */
                if (nvlist_alloc(&bt.nbe_zfs_props, NV_UNIQUE_NAME, 0) != 0) {
                        be_print_err(gettext("be_copy: internal "
                            "error: out of memory\n"));
                        return (BE_ERR_NOMEM);
                }
        }

        /*
         * If new BE name provided, validate the BE name and then verify
         * that new BE name doesn't already exist in some pool.
         */
        if (bt.nbe_name) {
                /* Validate original BE name */
                if (!be_valid_be_name(bt.nbe_name)) {
                        be_print_err(gettext("be_copy: "
                            "invalid BE name %s\n"), bt.nbe_name);
                        ret = BE_ERR_INVAL;
                        goto done;
                }

                /* Verify it doesn't already exist */
                if (getzoneid() == GLOBAL_ZONEID) {
                        if ((zret = zpool_iter(g_zfs, be_exists_callback,
                            bt.nbe_name)) > 0) {
                if ((zret = zpool_iter(g_zfs, be_exists_callback, bt.nbe_name))
                    > 0) {
                                be_print_err(gettext("be_copy: BE (%s) already "
                                    "exists\n"), bt.nbe_name);
                                ret = BE_ERR_BE_EXISTS;
                                goto done;
                        } else if (zret < 0) {
                                be_print_err(gettext("be_copy: zpool_iter "
                                    "failed: %s\n"),
                                    libzfs_error_description(g_zfs));
                        be_print_err(gettext("be_copy: zpool_iter failed: "
                            "%s\n"), libzfs_error_description(g_zfs));
                                ret = zfs_err_to_be_err(g_zfs);
                                goto done;
                        }
                } else {
                        be_make_root_ds(bt.nbe_zpool, bt.nbe_name, nbe_root_ds,
                            sizeof (nbe_root_ds));
                        if (zfs_dataset_exists(g_zfs, nbe_root_ds,
                            ZFS_TYPE_FILESYSTEM)) {
                                be_print_err(gettext("be_copy: BE (%s) already "
                                    "exists\n"), bt.nbe_name);
                                ret = BE_ERR_BE_EXISTS;
                                goto done;
                        }
                }
        } else {
                /*
                 * If an auto named BE is desired, it must be in the same
                 * pool is the original BE.
                 */
                if (bt.nbe_zpool != NULL) {
                        be_print_err(gettext("be_copy: cannot specify pool "
                            "name when creating an auto named BE\n"));
                        ret = BE_ERR_INVAL;
                        goto done;
                }

                /*
                 * Generate auto named BE
                 */
                if ((bt.nbe_name = be_auto_be_name(bt.obe_name))
                    == NULL) {
                        be_print_err(gettext("be_copy: "
                            "failed to generate auto BE name\n"));
                        ret = BE_ERR_AUTONAME;
                        goto done;
                }

                autoname = B_TRUE;
        }

        /*
         * If zpool name to create new BE in is not provided,
         * create new BE in original BE's pool.
         */
        if (bt.nbe_zpool == NULL) {
                bt.nbe_zpool = bt.obe_zpool;
        }

        /* Get root dataset names for obe_name and nbe_name */
        be_make_root_ds(bt.obe_zpool, bt.obe_name, obe_root_ds,
            sizeof (obe_root_ds));
        be_make_root_ds(bt.nbe_zpool, bt.nbe_name, nbe_root_ds,
            sizeof (nbe_root_ds));

        bt.obe_root_ds = obe_root_ds;
        bt.nbe_root_ds = nbe_root_ds;

        /*
         * If an existing snapshot name has been provided to create from,
         * verify that it exists for the original BE's root dataset.
         */
        if (bt.obe_snap_name != NULL) {

                /* Generate dataset name for snapshot to use. */
```

```
                (void) snprintf(ss, sizeof (ss), "%s@%s", bt.obe_root_ds,
                    bt.obe_snap_name);

                /* Verify snapshot exists */
                if (!zfs_dataset_exists(g_zfs, ss, ZFS_TYPE_SNAPSHOT)) {
                        be_print_err(gettext("be_copy: "
                            "snapshot does not exist (%s): %s\n"), ss,
                            libzfs_error_description(g_zfs));
                        ret = BE_ERR_SS_NOENT;
                        goto done;
                }
        } else {
                /*
                 * Else snapshot name was not provided, generate an
                 * auto named snapshot to use as its origin.
                 */
                if ((ret = _be_create_snapshot(bt.obe_name,
                    &bt.obe_snap_name, bt.policy)) != BE_SUCCESS) {
                        be_print_err(gettext("be_copy: "
                            "failed to create auto named snapshot\n"));
                        goto done;
                }

                if (nvlist_add_string(be_attrs, BE_ATTR_SNAP_NAME,
                    bt.obe_snap_name) != 0) {
                        be_print_err(gettext("be_copy: "
                            "failed to add snap name to be_attrs\n"));
                        ret = BE_ERR_NOMEM;
                        goto done;
                }
        }

        /* Get handle to original BE's root dataset. */
        if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_FILESYSTEM))
            == NULL) {
                be_print_err(gettext("be_copy: failed to "
                    "open BE root dataset (%s): %s\n"), bt.obe_root_ds,
                    libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
                goto done;
        }

        /* If original BE is currently mounted, record its altroot. */
        if (zfs_is_mounted(zhp, &bt.obe_altroot) && bt.obe_altroot == NULL) {
                be_print_err(gettext("be_copy: failed to "
                    "get altroot of mounted BE %s: %s\n"),
                    bt.obe_name, libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
                goto done;
        }

        if (strcmp(bt.obe_zpool, bt.nbe_zpool) == 0) {

                /* Do clone */

                /*
                 * Iterate through original BE's datasets and clone
                 * them to create new BE.  This call will end up closing
                 * the zfs handle passed in whether it succeeds for fails.
                 */
                if ((ret = be_clone_fs_callback(zhp, &bt)) != 0) {
                        zhp = NULL;
                        /* Creating clone BE failed */
                        if (!autoname || ret != BE_ERR_BE_EXISTS) {
                                be_print_err(gettext("be_copy: "
                                    "failed to clone new BE (%s) from "
                                    "orig BE (%s)\n"),
                                    bt.nbe_name, bt.obe_name);
                                ret = BE_ERR_CLONE;
                                goto done;
                        }

                        /*
                         * We failed to create the new BE because a BE with
                         * the auto-name we generated above has since come
                         * into existence.  Regenerate a new auto-name
                         * and retry.
                         */
                        for (i = 1; i < BE_AUTO_NAME_MAX_TRY; i++) {

                                /* Sleep 1 before retrying */
                                (void) sleep(1);

                                /* Generate new auto BE name */
                                free(bt.nbe_name);
                                if ((bt.nbe_name = be_auto_be_name(bt.obe_name))
                                    == NULL) {
                                        be_print_err(gettext("be_copy: "
                                            "failed to generate auto "
                                            "BE name\n"));
                                        ret = BE_ERR_AUTONAME;
                                        goto done;
                                }

                                /*
                                 * Regenerate string for new BE's
                                 * root dataset name
                                 */
                                be_make_root_ds(bt.nbe_zpool, bt.nbe_name,
                                    nbe_root_ds, sizeof (nbe_root_ds));
                                bt.nbe_root_ds = nbe_root_ds;

                                /*
                                 * Get handle to original BE's root dataset.
                                 */
                                if ((zhp = zfs_open(g_zfs, bt.obe_root_ds,
                                    ZFS_TYPE_FILESYSTEM)) == NULL) {
                                        be_print_err(gettext("be_copy: "
                                            "failed to open BE root dataset "
                                            "(%s): %s\n"), bt.obe_root_ds,
                                            libzfs_error_description(g_zfs));
                                        ret = zfs_err_to_be_err(g_zfs);
                                        goto done;
                                }

                                /*
                                 * Try to clone the BE again.  This
                                 * call will end up closing the zfs
                                 * handle passed in whether it
                                 * succeeds or fails.
                                 */
                                ret = be_clone_fs_callback(zhp, &bt);
                                zhp = NULL;
                                if (ret == 0) {
                                        break;
                                } else if (ret != BE_ERR_BE_EXISTS) {
                                        be_print_err(gettext("be_copy: "
                                            "failed to clone new BE "
                                            "(%s) from orig BE (%s)\n"),
                                            bt.nbe_name, bt.obe_name);
                                        ret = BE_ERR_CLONE;
                                        goto done;
```

```
                                        }
                                }

                                /*
                                 * If we've exhausted the maximum number of
                                 * tries, free the auto BE name and return
                                 * error.
                                 */
                                if (i == BE_AUTO_NAME_MAX_TRY) {
                                        be_print_err(gettext("be_copy: failed "
                                            "to create unique auto BE name\n"));
                                        free(bt.nbe_name);
                                        bt.nbe_name = NULL;
                                        ret = BE_ERR_AUTONAME;
                                        goto done;
                                }
                        }
                        zhp = NULL;

                } else {

                        /* Do copy (i.e. send BE datasets via zfs_send/recv) */

                        /*
                         * Verify BE container dataset in nbe_zpool exists.
                         * If not, create it.
                         */
                        if (!be_create_container_ds(bt.nbe_zpool)) {
                                ret = BE_ERR_CREATDS;
                                goto done;
                        }

                        /*
                         * Iterate through original BE's datasets and send
                         * them to the other pool.  This call will end up closing
                         * the zfs handle passed in whether it succeeds or fails.
                         */
                        if ((ret = be_send_fs_callback(zhp, &bt)) != 0) {
                                be_print_err(gettext("be_copy: failed to "
                                    "send BE (%s) to pool (%s)\n"), bt.obe_name,
                                    bt.nbe_zpool);
                                ret = BE_ERR_COPY;
                                zhp = NULL;
                                goto done;
                        }
                        zhp = NULL;
                }

                /*
                 * Set flag to note that the dataset(s) for the new BE have been
                 * successfully created so that if a failure happens from this point
                 * on, we know to cleanup these datasets.
                 */
                be_created = B_TRUE;

                /*
                 * Validate that the new BE is mountable.
                 * Do not attempt to mount non-global zone datasets
                 * since they are not cloned yet.
                 */
                if ((ret = _be_mount(bt.nbe_name, &new_mp, BE_MOUNT_FLAG_NO_ZONES))
                    != BE_SUCCESS) {
                        be_print_err(gettext("be_copy: failed to "
                            "mount newly created BE\n"));
                        (void) _be_unmount(bt.nbe_name, 0);
                        goto done;
```

```
	}

	/* Set UUID for new BE */
	if (getzoneid() == GLOBAL_ZONEID) {
		if (be_set_uuid(bt.nbe_root_ds) != BE_SUCCESS) {
			be_print_err(gettext("be_copy: failed to "
			    "set uuid for new BE\n"));
		}
	} else {
		if ((ret = be_zone_get_parent_uuid(bt.obe_root_ds,
		    &parent_uu)) != BE_SUCCESS) {
			be_print_err(gettext("be_copy: failed to get "
			    "parentbe uuid from orig BE\n"));
			ret = BE_ERR_ZONE_NO_PARENTBE;
			goto done;
		} else if ((ret = be_zone_set_parent_uuid(bt.nbe_root_ds,
		    parent_uu)) != BE_SUCCESS) {
			be_print_err(gettext("be_copy: failed to set "
			    "parentbe uuid for newly created BE\n"));
			goto done;
		}
	}

	/*
	 * Process zones outside of the private BE namespace.
	 * This has to be done here because we need the uuid set in the
	 * root dataset of the new BE. The uuid is use to set the parentbe
	 * property for the new zones datasets.
	 */
	if (getzoneid() == GLOBAL_ZONEID &&
	    be_get_uuid(bt.obe_root_ds, &uu) == BE_SUCCESS) {
		if ((ret = be_copy_zones(bt.obe_name, bt.obe_root_ds,
		    bt.nbe_root_ds)) != BE_SUCCESS) {
			be_print_err(gettext("be_copy: failed to process "
			    "zones\n"));
			goto done;
		}
	}

	/*
	 * Generate a list of file systems from the original BE that are
	 * legacy mounted.  We use this list to determine which entries in
	 * vfstab we need to update for the new BE we've just created.
	 */
	if ((ret = be_get_legacy_fs(bt.obe_name, bt.obe_root_ds, NULL, NULL,
	    &fld)) != BE_SUCCESS) {
		be_print_err(gettext("be_copy: failed to "
		    "get legacy mounted file system list for %s\n"),
		    bt.obe_name);
		goto done;
	}

	/*
	 * Update new BE's vfstab.
	 */
	if ((ret = be_update_vfstab(bt.nbe_name, bt.obe_zpool, bt.nbe_zpool,
	    &fld, new_mp)) != BE_SUCCESS) {
		be_print_err(gettext("be_copy: failed to "
		    "update new BE's vfstab (%s)\n"), bt.nbe_name);
		goto done;
	}

	/* Unmount the new BE */
	if ((ret = _be_unmount(bt.nbe_name, 0)) != BE_SUCCESS) {
		be_print_err(gettext("be_copy: failed to "
		    "unmount newly created BE\n"));
```

```
		goto done;
	}

	/*
	 * Add boot menu entry for newly created clone
	 */
	if (getzoneid() == GLOBAL_ZONEID &&
	    (ret = be_append_menu(bt.nbe_name, bt.nbe_zpool,
	    NULL, bt.obe_root_ds, bt.nbe_desc)) != BE_SUCCESS) {
		be_print_err(gettext("be_copy: failed to "
		    "add BE (%s) to boot menu\n"), bt.nbe_name);
		goto done;
	}

	/*
	 * If we succeeded in creating an auto named BE, set its policy
	 * type and return the auto generated name to the caller by storing
	 * it in the nvlist passed in by the caller.
	 */
	if (autoname) {
		/* Get handle to new BE's root dataset. */
		if ((zhp = zfs_open(g_zfs, bt.nbe_root_ds,
		    ZFS_TYPE_FILESYSTEM)) == NULL) {
			be_print_err(gettext("be_copy: failed to "
			    "open BE root dataset (%s): %s\n"), bt.nbe_root_ds,
			    libzfs_error_description(g_zfs));
			ret = zfs_err_to_be_err(g_zfs);
			goto done;
		}

		/*
		 * Set the policy type property into the new BE's root dataset
		 */
		if (bt.policy == NULL) {
			/* If no policy type provided, use default type */
			bt.policy = be_default_policy();
		}

		if (zfs_prop_set(zhp, BE_POLICY_PROPERTY, bt.policy) != 0) {
			be_print_err(gettext("be_copy: failed to "
			    "set BE policy for %s: %s\n"), bt.nbe_name,
			    libzfs_error_description(g_zfs));
			ret = zfs_err_to_be_err(g_zfs);
			goto done;
		}

		/*
		 * Return the auto generated name to the caller
		 */
		if (bt.nbe_name) {
			if (nvlist_add_string(be_attrs, BE_ATTR_NEW_BE_NAME,
			    bt.nbe_name) != 0) {
				be_print_err(gettext("be_copy: failed to "
				    "add snap name to be_attrs\n"));
			}
		}
	}

done:
	ZFS_CLOSE(zhp);
	be_free_fs_list(&fld);

	if (bt.nbe_zfs_props != NULL)
		nvlist_free(bt.nbe_zfs_props);

	free(bt.obe_altroot);
```

```
        free(new_mp);

        /*
         * If a failure occurred and we already created the datasets for
         * the new boot environment, destroy them.
         */
        if (ret != BE_SUCCESS && be_created) {
                be_destroy_data_t       cdd = { 0 };

                cdd.force_unmount = B_TRUE;

                be_print_err(gettext("be_copy: "
                    "destroying partially created boot environment\n"));

                if (getzoneid() == GLOBAL_ZONEID && be_get_uuid(bt.nbe_root_ds,
                    &cdd.gz_be_uuid) == 0)
                        (void) be_destroy_zones(bt.nbe_name, bt.nbe_root_ds,
                            &cdd);

                (void) _be_destroy(bt.nbe_root_ds, &cdd);
        }

        be_zfs_fini();

        return (ret);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   33179 Tue Aug  6 21:14:52 2013
new/usr/src/lib/libbe/common/be_list.c
*** NO COMMENTS ***
**********************************************************
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 */

#include <assert.h>
#include <libintl.h>
#include <libnvpair.h>
#include <libzfs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>

#include <libbe.h>
#include <libbe_priv.h>

/*
 * Callback data used for zfs_iter calls.
 */
typedef struct list_callback_data {
        char *zpool_name;
        char *be_name;
        be_node_list_t *be_nodes_head;
        be_node_list_t *be_nodes;
        char current_be[MAXPATHLEN];
} list_callback_data_t;
______unchanged_portion_omitted_

/* ******************************************************************** */
/*                      Semi-Private Functions                          */
/* ******************************************************************** */
```

```
/*
 * Function:    _be_list
 * Description: This does the actual work described in be_list.
 * Parameters:
 *              be_name - The name of the BE to look up.
 *                        If NULL a list of all BEs will be returned.
 *              be_nodes - A reference pointer to the list of BEs. The list
 *                         structure will be allocated here and must
 *                         be freed by a call to be_free_list. If there are no
 *                         BEs found on the system this reference will be
 *                         set to NULL.
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Semi-private (library wide use only)
 */
int
_be_list(char *be_name, be_node_list_t **be_nodes)
{
        list_callback_data_t cb = { 0 };
        be_transaction_data_t bt = { 0 };
        int ret = BE_SUCCESS;
        zpool_handle_t *zphp;
        char *rpool = NULL;
        struct be_defaults be_defaults;

        if (be_nodes == NULL)
                return (BE_ERR_INVAL);

        be_get_defaults(&be_defaults);

        if (be_find_current_be(&bt) != BE_SUCCESS) {
                /*
                 * We were unable to find a currently booted BE which
                 * probably means that we're not booted in a BE envoronment.
                 * None of the BE's will be marked as the active BE.
                 */
                (void) strcpy(cb.current_be, "-");
        } else {
                (void) strncpy(cb.current_be, bt.obe_name,
                    sizeof (cb.current_be));
                rpool = bt.obe_zpool;
        }

        /*
         * If be_name is NULL we'll look for all BE's on the system.
         * If not then we will only return data for the specified BE.
         */
        if (be_name != NULL)
                cb.be_name = strdup(be_name);

        if (be_defaults.be_deflt_rpool_container && rpool != NULL) {
                if ((zphp = zpool_open(g_zfs, rpool)) == NULL) {
                        be_print_err(gettext("be_list: failed to "
                        be_print_err(gettext("be_get_node_data: failed to "
                            "open rpool (%s): %s\n"), rpool,
                            libzfs_error_description(g_zfs));
                        free(cb.be_name);
                        return (zfs_err_to_be_err(g_zfs));
                }

                ret = be_get_list_callback(zphp, &cb);
        } else {
                if ((zpool_iter(g_zfs, be_get_list_callback, &cb)) != 0) {
```

```
			if (cb.be_nodes_head != NULL) {
				be_free_list(cb.be_nodes_head);
				cb.be_nodes_head = NULL;
				cb.be_nodes = NULL;
			}
			ret = BE_ERR_BE_NOENT;
		}
	}

	if (cb.be_nodes_head == NULL) {
		if (be_name != NULL)
			be_print_err(gettext("be_list: BE (%s) does not "
			    "exist\n"), be_name);
		else
			be_print_err(gettext("be_list: No BE’s found\n"));
		ret = BE_ERR_BE_NOENT;
	}

	*be_nodes = cb.be_nodes_head;

	free(cb.be_name);

	be_sort_list(be_nodes);

	return (ret);
}
______unchanged_portion_omitted_

/*
 * Function:	be_get_node_data
 * Description:	Helper function used to collect all the information to fill
 *		in the be_node_list structure to be returned by be_list.
 * Parameters:
 *		zhp - Handle to the root dataset for the BE whose information
 *		      we’re collecting.
 *		be_node - a pointer to the node structure we’re filling in.
 *		be_name - The BE name of the node whose information we’re
 *		          collecting.
 *		current_be - the name of the currently active BE.
 *		be_ds - The dataset name for the BE.
 *
 * Returns:
 *		BE_SUCCESS - Success
 *		be_errno_t - Failure
 * Scope:
 *		Private
 */
static int
be_get_node_data(
	zfs_handle_t *zhp,
	be_node_list_t *be_node,
	char *be_name,
	const char *rpool,
	char *current_be,
	char *be_ds)
{
	char prop_buf[MAXPATHLEN];
	nvlist_t *userprops = NULL;
	nvlist_t *propval = NULL;
	nvlist_t *zone_propval = NULL;
	char *prop_str = NULL;
	char *zone_prop_str = NULL;
	char *grub_default_bootfs = NULL;
	zpool_handle_t *zphp = NULL;
	int err = 0;
```

```
	if (be_node == NULL || be_name == NULL || current_be == NULL ||
	    be_ds == NULL) {
		be_print_err(gettext("be_get_node_data: invalid arguments, "
		    "can not be NULL\n"));
		return (BE_ERR_INVAL);
	}

	errno = 0;

	be_node->be_root_ds = strdup(be_ds);
	if ((err = errno) != 0 || be_node->be_root_ds == NULL) {
		be_print_err(gettext("be_get_node_data: failed to "
		    "copy root dataset name\n"));
		return (errno_to_be_err(err));
	}

	be_node->be_node_name = strdup(be_name);
	if ((err = errno) != 0 || be_node->be_node_name == NULL) {
		be_print_err(gettext("be_get_node_data: failed to "
		    "copy BE name\n"));
		return (errno_to_be_err(err));
	}
	if (strncmp(be_name, current_be, MAXPATHLEN) == 0)
		be_node->be_active = B_TRUE;
	else
		be_node->be_active = B_FALSE;

	be_node->be_rpool = strdup(rpool);
	if (be_node->be_rpool == NULL || (err = errno) != 0) {
		be_print_err(gettext("be_get_node_data: failed to "
		    "copy root pool name\n"));
		return (errno_to_be_err(err));
	}

	be_node->be_space_used = zfs_prop_get_int(zhp, ZFS_PROP_USED);

	if (getzoneid() == GLOBAL_ZONEID) {
		if ((zphp = zpool_open(g_zfs, rpool)) == NULL) {
			be_print_err(gettext("be_get_node_data: failed to open "
			    "pool (%s): %s\n"), rpool,
			    libzfs_error_description(g_zfs));
		be_print_err(gettext("be_get_node_data: failed to open pool "
		    "(%s): %s\n"), rpool, libzfs_error_description(g_zfs));
			return (zfs_err_to_be_err(g_zfs));
		}

		(void) zpool_get_prop(zphp, ZPOOL_PROP_BOOTFS, prop_buf,
		    ZFS_MAXPROPLEN, NULL);
		if (be_has_grub() && (be_default_grub_bootfs(rpool,
		    &grub_default_bootfs) == BE_SUCCESS) &&
		    grub_default_bootfs != NULL)
	(void) zpool_get_prop(zphp, ZPOOL_PROP_BOOTFS, prop_buf, ZFS_MAXPROPLEN,
	    NULL);
	if (be_has_grub() &&
	    (be_default_grub_bootfs(rpool, &grub_default_bootfs)
	    == BE_SUCCESS) && grub_default_bootfs != NULL)
			if (strcmp(grub_default_bootfs, be_ds) == 0)
				be_node->be_active_on_boot = B_TRUE;
			else
				be_node->be_active_on_boot = B_FALSE;
		else if (prop_buf != NULL && strcmp(prop_buf, be_ds) == 0)
			be_node->be_active_on_boot = B_TRUE;
		else
			be_node->be_active_on_boot = B_FALSE;

		be_node->be_global_active = B_TRUE;
```

```
                free(grub_default_bootfs);
                zpool_close(zphp);
        } else {
                if (be_zone_compare_uuids(be_node->be_root_ds))
                        be_node->be_global_active = B_TRUE;
                else
                        be_node->be_global_active = B_FALSE;
        }

        /*
         * If the dataset is mounted use the mount point
         * returned from the zfs_is_mounted call. If the
         * dataset is not mounted then pull the mount
         * point information out of the zfs properties.
         */
        be_node->be_mounted = zfs_is_mounted(zhp,
            &(be_node->be_mntpt));
        if (!be_node->be_mounted) {
                if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, prop_buf,
                    ZFS_MAXPROPLEN, NULL, NULL, 0, B_FALSE) == 0)
                        be_node->be_mntpt = strdup(prop_buf);
                else
                        return (zfs_err_to_be_err(g_zfs));
        }

        be_node->be_node_creation = (time_t)zfs_prop_get_int(zhp,
            ZFS_PROP_CREATION);

        /* Get all user properties used for libbe */
        if ((userprops = zfs_get_user_props(zhp)) == NULL) {
                be_node->be_policy_type = strdup(be_default_policy());
        } else {
                if (getzoneid() != GLOBAL_ZONEID) {
                        if (nvlist_lookup_nvlist(userprops,
                            BE_ZONE_ACTIVE_PROPERTY, &zone_propval) != 0 ||
                            zone_propval == NULL) {
                                be_node->be_active_on_boot = B_FALSE;
                        } else {
                                verify(nvlist_lookup_string(zone_propval,
                                    ZPROP_VALUE, &zone_prop_str) == 0);
                                if (strcmp(zone_prop_str, "on") == 0) {
                                        be_node->be_active_on_boot = B_TRUE;
                                } else {
                                        be_node->be_active_on_boot = B_FALSE;
                                }
                        }
                }

                if (nvlist_lookup_nvlist(userprops, BE_POLICY_PROPERTY,
                    &propval) != 0 || propval == NULL) {
                        be_node->be_policy_type =
                            strdup(be_default_policy());
                } else {
                        verify(nvlist_lookup_string(propval, ZPROP_VALUE,
                            &prop_str) == 0);
                        if (prop_str == NULL || strcmp(prop_str, "-") == 0 ||
                            strcmp(prop_str, "") == 0)
                                be_node->be_policy_type =
                                    strdup(be_default_policy());
                        else
                                be_node->be_policy_type = strdup(prop_str);
                }
                if (getzoneid() != GLOBAL_ZONEID) {
                        if (nvlist_lookup_nvlist(userprops,
                            BE_ZONE_PARENTBE_PROPERTY, &propval) != 0 &&
```

```
                            nvlist_lookup_string(propval, ZPROP_VALUE,

                if (nvlist_lookup_nvlist(userprops, BE_UUID_PROPERTY, &propval)
                    == 0 && nvlist_lookup_string(propval, ZPROP_VALUE,
                            &prop_str) == 0) {
                                be_node->be_uuid_str = strdup(prop_str);
                        }
                } else {
                        if (nvlist_lookup_nvlist(userprops, BE_UUID_PROPERTY,
                            &propval) == 0 && nvlist_lookup_string(propval,
                            ZPROP_VALUE, &prop_str) == 0) {
                                be_node->be_uuid_str = strdup(prop_str);
                        }
                }
        }

        /*
         * Increment the dataset counter to include the root dataset
         * of the BE.
         */
        be_node->be_node_num_datasets++;

        return (BE_SUCCESS);
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   77719 Tue Aug  6 21:14:52 2013
new/usr/src/lib/libbe/common/be_mount.c
*** NO COMMENTS ***
**********************************************************
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
 */

/*
 * System includes
 */
#include <assert.h>
#include <errno.h>
#include <libgen.h>
#include <libintl.h>
#include <libnvpair.h>
#include <libzfs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mntent.h>
#include <sys/mnttab.h>
#include <sys/mount.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/vfstab.h>
#include <sys/zone.h>
#include <sys/mkdev.h>
#include <unistd.h>

#include <libbe.h>
#include <libbe_priv.h>

#define BE_TMP_MNTPNT           "/tmp/.be.XXXXXX"

typedef struct dir_data {
        char *dir;
        char *ds;
} dir_data_t;
______unchanged_portion_omitted_
```

```
/* ******************************************************************** */
/*                      Semi-Private Functions                          */
/* ******************************************************************** */

/*
 * Function:    _be_mount
 * Description: Mounts a BE.  If the altroot is not provided, this function
 *              will generate a temporary mountpoint to mount the BE at.  It
 *              will return this temporary mountpoint to the caller via the
 *              altroot reference pointer passed in.  This returned value is
 *              allocated on heap storage and is the repsonsibility of the
 *              caller to free.
 * Parameters:
 *              be_name - pointer to name of BE to mount.
 *              altroot - reference pointer to altroot of where to mount BE.
 *              flags - flag indicating special handling for mounting the BE
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Semi-private (library wide use only)
 */
int
_be_mount(char *be_name, char **altroot, int flags)
{
        be_transaction_data_t   bt = { 0 };
        be_mount_data_t md = { 0 };
        zfs_handle_t    *zhp;
        char            obe_root_ds[MAXPATHLEN];
        char            *mp = NULL;
        char            *tmp_altroot = NULL;
        int             ret = BE_SUCCESS, err = 0;
        uuid_t          uu = { 0 };
        boolean_t       gen_tmp_altroot = B_FALSE;

        if (be_name == NULL || altroot == NULL)
                return (BE_ERR_INVAL);

        /* Set be_name as obe_name in bt structure */
        bt.obe_name = be_name;

        /* Find which zpool obe_name lives in */
        if ((err = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) == 0) {
                be_print_err(gettext("be_mount: failed to "
                    "find zpool for BE (%s)\n"), bt.obe_name);
                return (BE_ERR_BE_NOENT);
        } else if (err < 0) {
                be_print_err(gettext("be_mount: zpool_iter failed: %s\n"),
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /* Generate string for obe_name's root dataset */
        be_make_root_ds(bt.obe_zpool, bt.obe_name, obe_root_ds,
            sizeof (obe_root_ds));
        bt.obe_root_ds = obe_root_ds;

        /* Get handle to BE's root dataset */
        if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_FILESYSTEM)) ==
            NULL) {
                be_print_err(gettext("be_mount: failed to "
                    "open BE root dataset (%s): %s\n"), bt.obe_root_ds,
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }
        /* Make sure BE’s root dataset isn’t already mounted somewhere */
        if (zfs_is_mounted(zhp, &mp)) {
                ZFS_CLOSE(zhp);
                be_print_err(gettext("be_mount: %s is already mounted "
                    "at %s\n"), bt.obe_name, mp != NULL ? mp : "");
                free(mp);
                return (BE_ERR_MOUNTED);
        }

        /*
         * Fix this BE’s mountpoint if its root dataset isn’t set to
         * either ’legacy’ or ’/’.
         */
        if ((ret = fix_mountpoint(zhp)) != BE_SUCCESS) {
                be_print_err(gettext("be_mount: mountpoint check "
                    "failed for %s\n"), bt.obe_root_ds);
                ZFS_CLOSE(zhp);
                return (ret);
        }

        /*
         * If altroot not provided, create a temporary alternate root
         * to mount on
         */
        if (*altroot == NULL) {
                if ((ret = be_make_tmp_mountpoint(&tmp_altroot))
                    != BE_SUCCESS) {
                        be_print_err(gettext("be_mount: failed to "
                            "make temporary mountpoint\n"));
                        ZFS_CLOSE(zhp);
                        return (ret);
                }
                gen_tmp_altroot = B_TRUE;
        } else {
                tmp_altroot = *altroot;
        }

        md.altroot = tmp_altroot;
        md.shared_fs = flags & BE_MOUNT_FLAG_SHARED_FS;
        md.shared_rw = flags & BE_MOUNT_FLAG_SHARED_RW;

        /* Mount the BE’s root file system */
        if (getzoneid() == GLOBAL_ZONEID) {
                if ((ret = be_mount_root(zhp, tmp_altroot)) != BE_SUCCESS) {
                        be_print_err(gettext("be_mount: failed to "
                            "mount BE root file system\n"));
                        if (gen_tmp_altroot)
                                free(tmp_altroot);
                        ZFS_CLOSE(zhp);
                        return (ret);
                }
        } else {
                /* Legacy mount the zone root dataset */
                if ((ret = be_mount_zone_root(zhp, &md)) != BE_SUCCESS) {
                        be_print_err(gettext("be_mount: failed to "
                            "mount BE zone root file system\n"));
                        free(md.altroot);
                        ZFS_CLOSE(zhp);
                        return (ret);
                }
        }

        /* Iterate through BE’s children filesystems */
        if ((err = zfs_iter_filesystems(zhp, be_mount_callback,
            tmp_altroot)) != 0) {
```

```
                be_print_err(gettext("be_mount: failed to "
                    "mount BE (%s) on %s\n"), bt.obe_name, tmp_altroot);
                if (gen_tmp_altroot)
                        free(tmp_altroot);
                ZFS_CLOSE(zhp);
                return (err);
        }

        md.altroot = tmp_altroot;
        md.shared_fs = flags & BE_MOUNT_FLAG_SHARED_FS;
        md.shared_rw = flags & BE_MOUNT_FLAG_SHARED_RW;

        /*
         * Mount shared file systems if mount flag says so.
         */
        if (md.shared_fs) {
                /*
                 * Mount all ZFS file systems not under the BE’s root dataset
                 */
                (void) zpool_iter(g_zfs, zpool_shared_fs_callback, &md);

                /* TODO: Mount all non-ZFS file systems - Not supported yet */
        }

        /*
         * If we’re in the global zone and the global zone has a valid uuid,
         * mount all supported non-global zones.
         */
        if (getzoneid() == GLOBAL_ZONEID &&
            !(flags & BE_MOUNT_FLAG_NO_ZONES) &&
            be_get_uuid(bt.obe_root_ds, &uu) == BE_SUCCESS) {
                if ((ret = be_mount_zones(zhp, &md)) != BE_SUCCESS) {
                        (void) _be_unmount(bt.obe_name, 0);
                        if (gen_tmp_altroot)
                                free(tmp_altroot);
                        ZFS_CLOSE(zhp);
                        return (ret);
                }
        }

        ZFS_CLOSE(zhp);

        /*
         * If a NULL altroot was passed in, pass the generated altroot
         * back to the caller in altroot.
         */
        if (gen_tmp_altroot)
                *altroot = tmp_altroot;

        return (BE_SUCCESS);
}

/*
 * Function:    _be_unmount
 * Description: Unmount a BE.
 * Parameters:
 *              be_name - pointer to name of BE to unmount.
 *              flags - flags for unmounting the BE.
 * Returns:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Semi-private (library wide use only)
 */
int
_be_unmount(char *be_name, int flags)
```

```
{
        be_transaction_data_t   bt = { 0 };
        be_unmount_data_t       ud = { 0 };
        zfs_handle_t    *zhp;
        uuid_t          uu = { 0 };
        char            obe_root_ds[MAXPATHLEN];
        char            mountpoint[MAXPATHLEN];
        char            *mp = NULL;
        int             ret = BE_SUCCESS;
        int             zret = 0;

        if (be_name == NULL)
                return (BE_ERR_INVAL);

        /* Set be_name as obe_name in bt structure */
        bt.obe_name = be_name;

        /* Find which zpool obe_name lives in */
        if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) == 0) {
                be_print_err(gettext("be_unmount: failed to "
                    "find zpool for BE (%s)\n"), bt.obe_name);
                return (BE_ERR_BE_NOENT);
        } else if (zret < 0) {
                be_print_err(gettext("be_unmount: "
                    "zpool_iter failed: %s\n"),
                    libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
                return (ret);
        }

        /* Generate string for obe_name's root dataset */
        be_make_root_ds(bt.obe_zpool, bt.obe_name, obe_root_ds,
            sizeof (obe_root_ds));
        bt.obe_root_ds = obe_root_ds;

        /* Get handle to BE's root dataset */
        if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_FILESYSTEM)) ==
            NULL) {
                be_print_err(gettext("be_unmount: failed to "
                    "open BE root dataset (%s): %s\n"), bt.obe_root_ds,
                    libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
                return (ret);
        }

        /* Make sure BE's root dataset is mounted somewhere */
        if (!zfs_is_mounted(zhp, &mp)) {

                be_print_err(gettext("be_unmount: "
                    "(%s) not mounted\n"), bt.obe_name);

                /*
                 * BE is not mounted, fix this BE's mountpoint if its root
                 * dataset isn't set to either 'legacy' or '/'.
                 */
                if ((ret = fix_mountpoint(zhp)) != BE_SUCCESS) {
                        be_print_err(gettext("be_unmount: mountpoint check "
                            "failed for %s\n"), bt.obe_root_ds);
                        ZFS_CLOSE(zhp);
                        return (ret);
                }

                ZFS_CLOSE(zhp);
                return (BE_ERR_NOTMOUNTED);
        }
```

```
        /*
         * If we didn't get a mountpoint from the zfs_is_mounted call,
         * try and get it from its property.
         */
        if (mp == NULL) {
                if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
                    sizeof (mountpoint), NULL, NULL, 0, B_FALSE) != 0) {
                        be_print_err(gettext("be_unmount: failed to "
                            "get mountpoint of (%s)\n"), bt.obe_name);
                        ZFS_CLOSE(zhp);
                        return (BE_ERR_ZFS);
                }
        } else {
                (void) strlcpy(mountpoint, mp, sizeof (mountpoint));
                free(mp);
        }

        /* If BE mounted as current root, fail */
        if (strcmp(mountpoint, "/") == 0) {
                be_print_err(gettext("be_unmount: "
                    "cannot unmount currently running BE\n"));
                ZFS_CLOSE(zhp);
                return (BE_ERR_UMOUNT_CURR_BE);
        }

        ud.altroot = mountpoint;
        ud.force = flags & BE_UNMOUNT_FLAG_FORCE;

        /* Unmount all supported non-global zones if we're in the global zone */
        if (getzoneid() == GLOBAL_ZONEID &&
            be_get_uuid(bt.obe_root_ds, &uu) == BE_SUCCESS) {
                if ((ret = be_unmount_zones(&ud)) != BE_SUCCESS) {
                        ZFS_CLOSE(zhp);
                        return (ret);
                }
        }

        /* TODO: Unmount all non-ZFS file systems - Not supported yet */

        /* Unmount all ZFS file systems not under the BE root dataset */
        if ((ret = unmount_shared_fs(&ud)) != BE_SUCCESS) {
                be_print_err(gettext("be_unmount: failed to "
                    "unmount shared file systems\n"));
                ZFS_CLOSE(zhp);
                return (ret);
        }

        /* Unmount all children datasets under the BE's root dataset */
        if ((zret = zfs_iter_filesystems(zhp, be_unmount_callback,
            &ud)) != 0) {
                be_print_err(gettext("be_unmount: failed to "
                    "unmount BE (%s)\n"), bt.obe_name);
                ZFS_CLOSE(zhp);
                return (zret);
        }

        /* Unmount this BE's root filesystem */
        if (getzoneid() == GLOBAL_ZONEID) {
                if ((ret = be_unmount_root(zhp, &ud)) != BE_SUCCESS) {
                        ZFS_CLOSE(zhp);
                        return (ret);
                }
        } else {
                if ((ret = be_unmount_zone_root(zhp, &ud)) != BE_SUCCESS) {
                        ZFS_CLOSE(zhp);
                        return (ret);
                **}**
        **}**

        ZFS_CLOSE(zhp);

        return (BE_SUCCESS);
}

/*
 * Function:    be_mount_zone_root
 * Description: Mounts the zone root dataset for a zone.
 * Parameters:
 *              zfs - zfs_handle_t pointer to zone root dataset
 *              md - be_mount_data_t pointer to data for zone to be mounted
 * Returns:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Semi-private (library wide use only)
 */
int
be_mount_zone_root(zfs_handle_t *zhp, be_mount_data_t *md)
{
        **struct stat buf;**
        char    mountpoint[MAXPATHLEN];
        int     err = 0;

        /* Get mountpoint property of dataset */
        if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
            sizeof (mountpoint), NULL, NULL, 0, B_FALSE) != 0) {
                be_print_err(gettext("be_mount_zone_root: failed to "
                    "get mountpoint property for %s: %s\n"), zfs_get_name(zhp),
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /*
         * Make sure zone's root dataset is set to 'legacy'.  This is
         * currently a requirement in this implementation of zones
         * support.
         */
        if (strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) != 0) {
                be_print_err(gettext("be_mount_zone_root: "
                    "zone root dataset mountpoint is not 'legacy'\n"));
                return (BE_ERR_ZONE_ROOT_NOT_LEGACY);
        }

        **/* Create the mountpoint if it doesn't exist */**
        **if (lstat(md->altroot, &buf) != 0) {**
                **if (mkdirp(md->altroot, 0755) != 0) {**
                        **err = errno;**
                        **be_print_err(gettext("be_mount_zone_root: failed "**
                            **"to create mountpoint %s\n"), md->altroot);**
                        **return (errno_to_be_err(err));**
                **}**
        **}**

        /*
         * Legacy mount the zone root dataset.
         *
         * As a workaround for 6176743, we mount the zone's root with the
         * MS_OVERLAY option in case an alternate BE is mounted, and we're
         * mounting the root for the zone from the current BE here.  When an
         * alternate BE is mounted, it ties up the zone's zoneroot directory
         * for the current BE since the zone's zonepath is loopback mounted
         * from the current BE.
```

```
         *
         * TODO: The MS_OVERLAY option needs to be removed when 6176743
         * is fixed.
         */
        if (mount(zfs_get_name(zhp), md->altroot, MS_OVERLAY, MNTTYPE_ZFS,
            NULL, 0, NULL, 0) != 0) {
                err = errno;
                be_print_err(gettext("be_mount_zone_root: failed to "
                    "legacy mount zone root dataset (%s) at %s\n"),
                    zfs_get_name(zhp), md->altroot);
                return (errno_to_be_err(err));
        }

        return (BE_SUCCESS);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   23535 Tue Aug  6 21:14:53 2013
new/usr/src/lib/libbe/common/be_snapshot.c
*** NO COMMENTS ***
**********************************************************
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 */

/*
 * System includes
 */
#include <assert.h>
#include <libintl.h>
#include <libnvpair.h>
#include <libzfs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include <libbe.h>
#include <libbe_priv.h>

/* Private function prototypes */
static int be_rollback_check_callback(zfs_handle_t *, void *);
static int be_rollback_callback(zfs_handle_t *, void *);


/* ******************************************************************** */
/*                      Public Functions                                */
/* ******************************************************************** */

/*
 * Function:    be_create_snapshot
 * Description: Creates a recursive snapshot of all the datasets within a BE.
 *              If the name of the BE to snapshot is not provided, it assumes
 *              we’re snapshotting the currently running BE.  If the snapshot
```

```
 *              name is not provided it creates an auto named snapshot, which
 *              will be returned to the caller upon success.
 * Parameters:
 *              be_attrs - pointer to nvlist_t of attributes being passed in.
 *                      The following attributes are used by this function:
 *
 *                      BE_ATTR_ORIG_BE_NAME            *optional
 *                      BE_ATTR_SNAP_NAME               *optional
 *                      BE_ATTR_POLICY                  *optional
 *
 *                      If the BE_ATTR_SNAP_NAME was not passed in, upon
 *                      successful BE snapshot creation, the following
 *                      attribute value will be returned to the caller by
 *                      setting it in the be_attrs parameter passed in:
 *
 *                      BE_ATTR_SNAP_NAME
 *
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Public
 */
int
be_create_snapshot(nvlist_t *be_attrs)
{
        char            *be_name = NULL;
        char            *snap_name = NULL;
        char            *policy = NULL;
        boolean_t       autoname = B_FALSE;
        int             ret = BE_SUCCESS;

        /* Initialize libzfs handle */
        if (!be_zfs_init())
                return (BE_ERR_INIT);

        /* Get original BE name if one was provided */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_ORIG_BE_NAME, DATA_TYPE_STRING, &be_name, NULL) != 0) {
                be_print_err(gettext("be_create_snapshot: failed to "
                    "lookup BE_ATTR_ORIG_BE_NAME attribute\n"));
                be_zfs_fini();
                return (BE_ERR_INVAL);
        }

        /* Validate original BE name if one was provided */
        if (be_name != NULL && !be_valid_be_name(be_name)) {
                be_print_err(gettext("be_create_snapshot: "
                    "invalid BE name %s\n"), be_name);
                be_zfs_fini();
                return (BE_ERR_INVAL);
        }

        /* Get snapshot name to create if one was provided */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_SNAP_NAME, DATA_TYPE_STRING, &snap_name, NULL) != 0) {
                be_print_err(gettext("be_create_snapshot: "
                    "failed to lookup BE_ATTR_SNAP_NAME attribute\n"));
                be_zfs_fini();
                return (BE_ERR_INVAL);
        }

        /* Get BE policy to create this snapshot under */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_POLICY, DATA_TYPE_STRING, &policy, NULL) != 0) {
                be_print_err(gettext("be_create_snapshot: "
```

```c
                    "failed to lookup BE_ATTR_POLICY attribute\n"));
                be_zfs_fini();
                return (BE_ERR_INVAL);
        }

        /*
         * If no snap_name ws provided, we're going to create an
         * auto named snapshot.  Set flag so that we know to pass
         * the auto named snapshot to the caller later.
         */
        if (snap_name == NULL)
                autoname = B_TRUE;

        if ((ret = _be_create_snapshot(be_name, &snap_name, policy))
            == BE_SUCCESS) {
                if (autoname == B_TRUE) {
                        /*
                         * Set auto named snapshot name in the
                         * nvlist passed in by the caller.
                         */
                        if (nvlist_add_string(be_attrs, BE_ATTR_SNAP_NAME,
                            snap_name) != 0) {
                                be_print_err(gettext("be_create_snapshot: "
                                    "failed to add auto snap name (%s) to "
                                    "be_attrs\n"), snap_name);
                                ret = BE_ERR_NOMEM;
                        }
                }
        }

        be_zfs_fini();

        return (ret);
}
```

______***unchanged_portion_omitted_***

```c
/*
 * Function:    be_rollback
 * Description: Rolls back a BE and all of its children datasets to the
 *              named snapshot.  All of the BE's datasets must have the
 *              named snapshot for this function to succeed.  If the name
 *              of the BE is not passed in, this function assumes we're
 *              operating on the currently booted live BE.
 *
 *              Note - This function does not check if the BE has any
 *              younger snapshots than the one we're trying to rollback to.
 *              If it does, then those younger snapshots and their dependent
 *              clone file systems will get destroyed in the process of
 *              rolling back.
 *
 * Parameters:
 *              be_attrs - pointer to nvlist_t of attributes being passed in.
 *                         The following attributes are used by this function:
 *
 *                         BE_ATTR_ORIG_BE_NAME         *optional
 *                         BE_ATTR_SNAP_NAME            *required
 *
 * Returns:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Public
 */
int
be_rollback(nvlist_t *be_attrs)
{
```

```c
        be_transaction_data_t   bt = { 0 };
        zfs_handle_t            *zhp = NULL;
        zpool_handle_t          *zphp;
        char                    obe_root_ds[MAXPATHLEN];
        char                    *obe_name = NULL;
        int                     zret = 0, ret = BE_SUCCESS;
        struct be_defaults be_defaults;

        /* Initialize libzfs handle */
        if (!be_zfs_init())
                return (BE_ERR_INIT);

        if ((ret = be_find_current_be(&bt)) != BE_SUCCESS) {
                return (ret);
        }

        /* Get original BE name if one was provided */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_ORIG_BE_NAME, DATA_TYPE_STRING, &obe_name, NULL) != 0) {
                be_print_err(gettext("be_rollback: "
                    "failed to lookup BE_ATTR_ORIG_BE_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }

        be_get_defaults(&be_defaults);

        /* If original BE name not provided, use current BE */
        if (obe_name != NULL) {
                bt.obe_name = obe_name;
                /* Validate original BE name  */
                if (!be_valid_be_name(bt.obe_name)) {
                        be_print_err(gettext("be_rollback: "
                            "invalid BE name %s\n"), bt.obe_name);
                        return (BE_ERR_INVAL);
                }
        }

        /* Get snapshot name to rollback to */
        if (nvlist_lookup_string(be_attrs, BE_ATTR_SNAP_NAME, &bt.obe_snap_name)
            != 0) {
                be_print_err(gettext("be_rollback: "
                    "failed to lookup BE_ATTR_SNAP_NAME attribute.\n"));
                return (BE_ERR_INVAL);
        }

        if (be_defaults.be_deflt_rpool_container) {
                if ((zphp = zpool_open(g_zfs, bt.obe_zpool)) == NULL) {
                        be_print_err(gettext("be_rollback: failed to "
                            "open rpool (%s): %s\n"), bt.obe_zpool,
                            libzfs_error_description(g_zfs));
                        return (zfs_err_to_be_err(g_zfs));
                }
                zret = be_find_zpool_callback(zphp, &bt);
        } else {
                /* Find which zpool obe_name lives in */
                if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) ==
                    0) {
                        be_print_err(gettext("be_rollback: "
                            "failed to find zpool for BE (%s)\n"), bt.obe_name);
                        return (BE_ERR_BE_NOENT);
                } else if (zret < 0) {
                        be_print_err(gettext("be_rollback: "
                            "zpool_iter failed: %s\n"),
                            libzfs_error_description(g_zfs));
                        return (zfs_err_to_be_err(g_zfs));
                }
        }

        /* Generate string for BE's root dataset */
        be_make_root_ds(bt.obe_zpool, bt.obe_name, obe_root_ds,
            sizeof (obe_root_ds));
        bt.obe_root_ds = obe_root_ds;

        if (getzoneid() != GLOBAL_ZONEID) {
                if (!be_zone_compare_uuids(bt.obe_root_ds)) {
                        be_print_err(gettext("be_rollback: rolling back zone "
                            "root dataset from non-active global BE is not "
                            "supported\n"));
                        return (BE_ERR_NOTSUP);
                }
        }

        /* Get handle to BE's root dataset */
        if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_DATASET)) == NULL) {
                be_print_err(gettext("be_rollback: "
                    "failed to open BE root dataset (%s): %s\n"),
                    bt.obe_root_ds, libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /*
         * Check that snapshot name exists for this BE and all of its
         * children file systems.  This call will end up closing the
         * zfs handle passed in whether it succeeds or fails.
         */
        if ((ret = be_rollback_check_callback(zhp, bt.obe_snap_name)) != 0) {
                zhp = NULL;
                return (ret);
        }

        /* Get handle to BE's root dataset */
        if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_DATASET)) == NULL) {
                be_print_err(gettext("be_rollback: "
                    "failed to open BE root dataset (%s): %s\n"),
                    bt.obe_root_ds, libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /*
         * Iterate through a BE's datasets and roll them all back to
         * the specified snapshot.  This call will end up closing the
         * zfs handle passed in whether it succeeds or fails.
         */
        if ((ret = be_rollback_callback(zhp, bt.obe_snap_name)) != 0) {
                zhp = NULL;
                be_print_err(gettext("be_rollback: "
                    "failed to rollback BE %s to %s\n"), bt.obe_name,
                    bt.obe_snap_name);
                return (ret);
        }
        zhp = NULL;
        be_zfs_fini();
        return (BE_SUCCESS);
}


/* ******************************************************************** */
/*                      Semi-Private Functions                          */
/* ******************************************************************** */

/*
 * Function:    _be_create_snapshot
```

```
 * Description: see be_create_snapshot
 * Parameters:
 *              be_name - The name of the BE that we're taking a snapshot of.
 *              snap_name - The name of the snapshot we're creating. If
 *                      snap_name is NULL an auto generated name will be used,
 *                      and upon success, will return that name via this
 *                      reference pointer.  The caller is responsible for
 *                      freeing the returned name.
 *              policy - The clean-up policy type. (library wide use only)
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Semi-private (library wide use only)
 */
int
_be_create_snapshot(char *be_name, char **snap_name, char *policy)
{
        be_transaction_data_t   bt = { 0 };
        zfs_handle_t            *zhp = NULL;
        nvlist_t                *ss_props = NULL;
        char                    ss[MAXPATHLEN];
        char                    root_ds[MAXPATHLEN];
        int                     pool_version = 0;
        int                     i = 0;
        int                     zret = 0, ret = BE_SUCCESS;
        boolean_t               autoname = B_FALSE;

        /* Set parameters in bt structure */
        bt.obe_name = be_name;
        bt.obe_snap_name = *snap_name;
        bt.policy = policy;

        /* If original BE name not supplied, use current BE */
        if (bt.obe_name == NULL) {
                if ((ret = be_find_current_be(&bt)) != BE_SUCCESS) {
                        return (ret);
                }
        }

        /* Find which zpool obe_name lives in */
        if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) == 0) {
                be_print_err(gettext("be_create_snapshot: failed to "
                    "find zpool for BE (%s)\n"), bt.obe_name);
                return (BE_ERR_BE_NOENT);
        } else if (zret < 0) {
                be_print_err(gettext("be_create_snapshot: "
                    "zpool_iter failed: %s\n"),
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        be_make_root_ds(bt.obe_zpool, bt.obe_name, root_ds,
            sizeof (root_ds));
        bt.obe_root_ds = root_ds;

        if (getzoneid() != GLOBAL_ZONEID) {
                if (!be_zone_compare_uuids(bt.obe_root_ds)) {
                        be_print_err(gettext("be_create_snapshot: creating "
                            "snapshot for the zone root dataset from "
                            "non-active global BE is not "
                            "supported\n"));
                        return (BE_ERR_NOTSUP);
                }
        }
        /* If BE policy not specified, use the default policy */
        if (bt.policy == NULL) {
                bt.policy = be_default_policy();
        } else {
                /* Validate policy type */
                if (!valid_be_policy(bt.policy)) {
                        be_print_err(gettext("be_create_snapshot: "
                            "invalid BE policy type (%s)\n"), bt.policy);
                        return (BE_ERR_INVAL);
                }
        }

        /*
         * If snapshot name not specified, set auto name flag and
         * generate auto snapshot name.
         */
        if (bt.obe_snap_name == NULL) {
                autoname = B_TRUE;
                if ((bt.obe_snap_name = be_auto_snap_name())
                    == NULL) {
                        be_print_err(gettext("be_create_snapshot: "
                            "failed to create auto snapshot name\n"));
                        ret =  BE_ERR_AUTONAME;
                        goto done;
                }
        }

        /* Generate the name of the snapshot to take. */
        (void) snprintf(ss, sizeof (ss), "%s@%s", bt.obe_root_ds,
            bt.obe_snap_name);

        /* Get handle to BE's root dataset */
        if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_DATASET))
            == NULL) {
                be_print_err(gettext("be_create_snapshot: "
                    "failed to open BE root dataset (%s): %s\n"),
                    bt.obe_root_ds, libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
                goto done;
        }

        /* Get the ZFS pool version of the pool where this dataset resides */
        if (zfs_spa_version(zhp, &pool_version) != 0) {
                be_print_err(gettext("be_create_snapshot: failed to "
                    "get ZFS pool version for %s: %s\n"), zfs_get_name(zhp),
                    libzfs_error_description(g_zfs));
        }

        /*
         * If ZFS pool version supports snapshot user properties, store
         * cleanup policy there.  Otherwise don't set one - this snapshot
         * will always inherit the cleanup policy from its parent.
         */
        if (getzoneid() == GLOBAL_ZONEID) {
                if (pool_version >= SPA_VERSION_SNAP_PROPS) {
                        if (nvlist_alloc(&ss_props, NV_UNIQUE_NAME, 0) != 0) {
                                be_print_err(gettext("be_create_snapshot: "
                                    "internal error: out of memory\n"));
                        be_print_err(gettext("be_create_snapshot: internal "
                            "error: out of memory\n"));
                                return (BE_ERR_NOMEM);
                        }
                        if (nvlist_add_string(ss_props, BE_POLICY_PROPERTY,
                            bt.policy) != 0) {
                                be_print_err(gettext("be_create_snapshot: "
                                    "internal error: out of memory\n"));
```

```
                if (nvlist_add_string(ss_props, BE_POLICY_PROPERTY, bt.policy)
                    != 0) {
                        be_print_err(gettext("be_create_snapshot: internal "
                            "error: out of memory\n"));
                                nvlist_free(ss_props);
                                return (BE_ERR_NOMEM);
                        }
                } else if (policy != NULL) {
                        /*
                         * If an explicit cleanup policy was requested
                         * by the caller and we don't support it, error out.
                         */
                        be_print_err(gettext("be_create_snapshot: cannot set "
                            "cleanup policy: ZFS pool version is %d\n"),
                            pool_version);
                    "cleanup policy: ZFS pool version is %d\n"), pool_version);
                        return (BE_ERR_NOTSUP);
                }
        }

        /* Create the snapshots recursively */
        if (zfs_snapshot(g_zfs, ss, B_TRUE, ss_props) != 0) {
                if (!autoname || libzfs_errno(g_zfs) != EZFS_EXISTS) {
                        be_print_err(gettext("be_create_snapshot: "
                            "recursive snapshot of %s failed: %s\n"),
                            ss, libzfs_error_description(g_zfs));

                        if (libzfs_errno(g_zfs) == EZFS_EXISTS)
                                ret = BE_ERR_SS_EXISTS;
                        else
                                ret = zfs_err_to_be_err(g_zfs);

                        goto done;
                } else {
                        for (i = 1; i < BE_AUTO_NAME_MAX_TRY; i++) {

                                /* Sleep 1 before retrying */
                                (void) sleep(1);

                                /* Generate new auto snapshot name. */
                                free(bt.obe_snap_name);
                                if ((bt.obe_snap_name =
                                    be_auto_snap_name()) == NULL) {
                                        be_print_err(gettext(
                                            "be_create_snapshot: failed to "
                                            "create auto snapshot name\n"));
                                        ret = BE_ERR_AUTONAME;
                                        goto done;
                                }

                                /* Generate string of the snapshot to take. */
                                (void) snprintf(ss, sizeof (ss), "%s@%s",
                                    bt.obe_root_ds, bt.obe_snap_name);

                                /* Create the snapshots recursively */
                                if (zfs_snapshot(g_zfs, ss, B_TRUE, ss_props)
                                    != 0) {
                                        if (libzfs_errno(g_zfs) !=
                                            EZFS_EXISTS) {
                                                be_print_err(gettext(
                                                    "be_create_snapshot: "
                                                    "recursive snapshot of %s "
                                                    "failed: %s\n"), ss,
                                                    libzfs_error_description(
                                                    g_zfs));
                                                ret = zfs_err_to_be_err(g_zfs);
                                                goto done;
                                        }
                                } else {
                                        break;
                                }
                        }

                        /*
                         * If we exhausted the maximum number of tries,
                         * free the auto snap name and set error.
                         */
                        if (i == BE_AUTO_NAME_MAX_TRY) {
                                be_print_err(gettext("be_create_snapshot: "
                                    "failed to create unique auto snapshot "
                                    "name\n"));
                                free(bt.obe_snap_name);
                                bt.obe_snap_name = NULL;
                                ret = BE_ERR_AUTONAME;
                        }
                }
        }

        /*
         * If we succeeded in creating an auto named snapshot, store
         * the name in the nvlist passed in by the caller.
         */
        if (autoname && bt.obe_snap_name) {
                *snap_name = bt.obe_snap_name;
        }

done:
        ZFS_CLOSE(zhp);

        if (ss_props != NULL)
                nvlist_free(ss_props);

        return (ret);
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
  100368 Tue Aug  6 21:14:54 2013
new/usr/src/lib/libbe/common/be_utils.c
*** NO COMMENTS ***
**********************************************************
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 */


/*
 * System includes
 */
#include <assert.h>
#include <errno.h>
#include <libgen.h>
#include <libintl.h>
#include <libnvpair.h>
#include <libzfs.h>
#include <libgen.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/vfstab.h>
#include <sys/param.h>
#include <sys/systeminfo.h>
#include <ctype.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <deflt.h>
#include <wait.h>
#include <libdevinfo.h>
#include <libgen.h>

#include <libbe.h>
#include <libbe_priv.h>
```

```
/* Private function prototypes */
static int update_dataset(char *, int, char *, char *, char *);
static int _update_vfstab(char *, char *, char *, char *, be_fs_list_data_t *);
static int be_open_menu(char *, char *, FILE **, char *, boolean_t);
static int be_create_menu(char *, char *, FILE **, char *);
static char *be_get_auto_name(char *, char *, boolean_t);

/*
 * Global error printing
 */
boolean_t do_print = B_FALSE;

/*
 * Private datatypes
 */
typedef struct zone_be_name_cb_data {
        char *base_be_name;
        int num;
} zone_be_name_cb_data_t;
______unchanged_portion_omitted_

/*
 * Function:    be_make_root_ds
 * Description: Generate string for BE's root dataset given the pool
 *              it lives in and the BE name.
 * Parameters:
 *              zpool - pointer zpool name.
 *              be_name - pointer to BE name.
 *              be_root_ds - pointer to buffer to return BE root dataset in.
 *              be_root_ds_size - size of be_root_ds
 * Returns:
 *              None
 * Scope:
 *              Semi-private (library wide use only)
 */
void
be_make_root_ds(const char *zpool, const char *be_name, char *be_root_ds,
    int be_root_ds_size)
{
        struct be_defaults be_defaults;
        be_get_defaults(&be_defaults);
        char    *root_ds = NULL;

        if (getzoneid() == GLOBAL_ZONEID) {
                if (be_defaults.be_deflt_rpool_container) {
                        (void) snprintf(be_root_ds, be_root_ds_size,
                            "%s/%s", zpool, be_name);
                } else {
                        (void) snprintf(be_root_ds, be_root_ds_size,
                            "%s/%s/%s", zpool, BE_CONTAINER_DS_NAME, be_name);
                }
        } else {
                /*
                 * In non-global zone we can use path from mounted root dataset
                 * to generate BE's root dataset string.
                 */
                if ((root_ds = be_get_ds_from_dir("/")) != NULL) {
                        (void) snprintf(be_root_ds, be_root_ds_size, "%s/%s",
                            dirname(root_ds), be_name);
                } else {
                        be_print_err(gettext("be_make_root_ds: zone root "
                            "dataset is not mounted\n"));
                        return;
                }
        }
        if (be_defaults.be_deflt_rpool_container)
```

```
                (void) snprintf(be_root_ds, be_root_ds_size, "%s/%s", zpool,
                    be_name);
        else
                (void) snprintf(be_root_ds, be_root_ds_size, "%s/%s/%s", zpool,
                    BE_CONTAINER_DS_NAME, be_name);
}

/*
 * Function:    be_make_container_ds
 * Description: Generate string for the BE container dataset given a pool name.
 * Parameters:
 *              zpool - pointer zpool name.
 *              container_ds - pointer to buffer to return BE container
 *                      dataset in.
 *              container_ds_size - size of container_ds
 * Returns:
 *              None
 * Scope:
 *              Semi-private (library wide use only)
 */
void
be_make_container_ds(const char *zpool,  char *container_ds,
    int container_ds_size)
{
        struct be_defaults be_defaults;
        be_get_defaults(&be_defaults);
        char    *root_ds = NULL;

        if (getzoneid() == GLOBAL_ZONEID) {
                if (be_defaults.be_deflt_rpool_container) {
                        (void) snprintf(container_ds, container_ds_size,
                            "%s", zpool);
                } else {
                        (void) snprintf(container_ds, container_ds_size,
                            "%s/%s", zpool, BE_CONTAINER_DS_NAME);
                }
        } else {
                if ((root_ds = be_get_ds_from_dir("/")) != NULL) {
                        (void) strlcpy(container_ds, dirname(root_ds),
                            container_ds_size);
                } else {
                        be_print_err(gettext("be_make_container_ds: zone root "
                            "dataset is not mounted\n"));
                        return;
                }
        }
        if (be_defaults.be_deflt_rpool_container)
                (void) snprintf(container_ds, container_ds_size, "%s", zpool);
        else
                (void) snprintf(container_ds, container_ds_size, "%s/%s", zpool,
                    BE_CONTAINER_DS_NAME);
}
______unchanged_portion_omitted_

/*
 * Function:    be_zpool_find_current_be_callback
 * Description: Callback function used to iterate through all existing pools
 *              to find the BE that is the currently booted BE.
 * Parameters:
 *              zlp - zpool_handle_t pointer to the current pool being
 *                      looked at.
 *              data - be_transaction_data_t pointer.
 *                      Upon successfully finding the current BE, the
 *                      obe_zpool member of this parameter is set to the
 *                      pool it is found in.
 * Return:
```

```
 *              1 - Found current BE in this pool.
 *              0 - Did not find current BE in this pool.
 * Scope:
 *              Semi-private (library wide use only)
 */
int
be_zpool_find_current_be_callback(zpool_handle_t *zlp, void *data)
{
        be_transaction_data_t   *bt = data;
        zfs_handle_t            *zhp = NULL;
        const char              *zpool =  zpool_get_name(zlp);
        char                    be_container_ds[MAXPATHLEN];
        char                    *zpath = NULL;

        /*
         * Generate string for BE container dataset
         */
        if (getzoneid() != GLOBAL_ZONEID) {
                if ((zpath = be_get_ds_from_dir("/")) != NULL) {
                        (void) strlcpy(be_container_ds, dirname(zpath),
                            sizeof (be_container_ds));
                } else {
                        be_print_err(gettext(
                            "be_zpool_find_current_be_callback: "
                            "zone root dataset is not mounted\n"));
                        return (0);
                }
        } else {
                be_make_container_ds(zpool, be_container_ds,
                    sizeof (be_container_ds));
        }
        be_make_container_ds(zpool, be_container_ds, sizeof (be_container_ds));

        /*
         * Check if a BE container dataset exists in this pool.
         */
        if (!zfs_dataset_exists(g_zfs, be_container_ds, ZFS_TYPE_FILESYSTEM)) {
                zpool_close(zlp);
                return (0);
        }

        /*
         * Get handle to this zpool's BE container dataset.
         */
        if ((zhp = zfs_open(g_zfs, be_container_ds, ZFS_TYPE_FILESYSTEM)) ==
            NULL) {
                be_print_err(gettext("be_zpool_find_current_be_callback: "
                    "failed to open BE container dataset (%s)\n"),
                    be_container_ds);
                zpool_close(zlp);
                return (0);
        }

        /*
         * Iterate through all potential BEs in this zpool
         */
        if (zfs_iter_filesystems(zhp, be_zfs_find_current_be_callback, bt)) {
                /*
                 * Found current BE dataset; set obe_zpool
                 */
                if ((bt->obe_zpool = strdup(zpool)) == NULL) {
                        be_print_err(gettext(
                            "be_zpool_find_current_be_callback: "
                            "memory allocation failed\n"));
                        ZFS_CLOSE(zhp);
                        zpool_close(zlp);
                        return (0);
                }

                ZFS_CLOSE(zhp);
                zpool_close(zlp);
                return (1);
        }

        ZFS_CLOSE(zhp);
        zpool_close(zlp);

        return (0);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   18420 Tue Aug  6 21:14:55 2013
new/usr/src/lib/libbe/common/be_zones.c
*** NO COMMENTS ***
**********************************************************
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 */

/*
 * System includes
 */
#include <assert.h>
#include <errno.h>
#include <libintl.h>
#include <libnvpair.h>
#include <libzfs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mntent.h>
#include <sys/mnttab.h>
#include <sys/mount.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/vfstab.h>
#include <unistd.h>

#include <libbe.h>
#include <libbe_priv.h>

typedef struct active_zone_root_data {
        uuid_t  parent_uuid;
        char    *zoneroot_ds;
} active_zone_root_data_t;
______unchanged_portion_omitted_

/*
 * Function:    be_find_active_zone_root
 * Description: This function will find the active zone root of a zone for
 *              a given global BE.  It will iterate all of the zone roots
```

```
 *              under a zonepath, find the zone roots that belong to the
 *              specified global BE, and return the one that is active.
 * Parameters:
 *              be_zhp - zfs handle to global BE root dataset.
 *              zonepath_ds - pointer to zone's zonepath dataset.
 *              zoneroot_ds - pointer to a buffer to store the dataset name of
 *                      the zone's zoneroot that's currently active for this
 *                      given global BE..
 *              zoneroot-ds_size - size of zoneroot_ds.
 * Returns:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Semi-private (library wide use only)
 */
int
be_find_active_zone_root(zfs_handle_t *be_zhp, char *zonepath_ds,
    char *zoneroot_ds, int zoneroot_ds_size)
{
        active_zone_root_data_t         azr_data = { 0 };
        zfs_handle_t                    *zhp;
        char                            zone_container_ds[MAXPATHLEN];
        int                             ret = BE_SUCCESS;

        /* Get the uuid of the parent global BE */
        if (getzoneid() == GLOBAL_ZONEID) {
                if ((ret = be_get_uuid(zfs_get_name(be_zhp),
                    &azr_data.parent_uuid)) != BE_SUCCESS) {
                        be_print_err(gettext("be_find_active_zone_root: failed "
                            "to get uuid for BE root dataset %s\n"),
                            zfs_get_name(be_zhp));
        if ((ret = be_get_uuid(zfs_get_name(be_zhp), &azr_data.parent_uuid))
            != BE_SUCCESS) {
                be_print_err(gettext("be_find_active_zone_root: failed to "
                    "get uuid for BE root dataset %s\n"), zfs_get_name(be_zhp));
                        return (ret);
                }
        } else {
                if ((ret = be_zone_get_parent_uuid(zfs_get_name(be_zhp),
                    &azr_data.parent_uuid)) != BE_SUCCESS) {
                        be_print_err(gettext("be_find_active_zone_root: failed "
                            "to get parentbe uuid for zone root dataset %s\n"),
                            zfs_get_name(be_zhp));
                        return (ret);
                }
        }

        /* Generate string for the root container dataset  for this zone. */
        be_make_container_ds(zonepath_ds, zone_container_ds,
            sizeof (zone_container_ds));

        /* Get handle to this zone's root container dataset */
        if ((zhp = zfs_open(g_zfs, zone_container_ds, ZFS_TYPE_FILESYSTEM))
            == NULL) {
                be_print_err(gettext("be_find_active_zone_root: failed to "
                    "open zone root container dataset (%s): %s\n"),
                    zone_container_ds, libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /*
         * Iterate through all of this zone's BEs, looking for ones
         * that belong to the parent global BE, and finding the one
         * that is marked active.
         */
        if ((ret = zfs_iter_filesystems(zhp, be_find_active_zone_root_callback,
            &azr_data)) != 0) {
                be_print_err(gettext("be_find_active_zone_root: failed to "
                    "find active zone root in zonepath dataset %s: %s\n"),
                    zonepath_ds, be_err_to_str(ret));
                goto done;
        }

        if (azr_data.zoneroot_ds != NULL) {
                (void) strlcpy(zoneroot_ds, azr_data.zoneroot_ds,
                    zoneroot_ds_size);
                free(azr_data.zoneroot_ds);
        } else {
                be_print_err(gettext("be_find_active_zone_root: failed to "
                    "find active zone root in zonepath dataset %s\n"),
                    zonepath_ds);
                ret = BE_ERR_ZONE_NO_ACTIVE_ROOT;
        }

done:
        ZFS_CLOSE(zhp);
        return (ret);
}
```
______***unchanged_portion_omitted_***

```
/*
 * Function:    be_zone_set_parent_uuid
 * Description: This function sets parentbe uuid into
 *              a zfs user property for a root zone dataset.
 * Parameters:
 *              root_ds - Root zone dataset of the BE to set a uuid on.
 * Return:
 *              be_errno_t - Failure
 *              BE_SUCCESS - Success
 * Scope:
 *              Semi-private (library wide uses only)
 */
int
be_zone_set_parent_uuid(char *root_ds, uuid_t uu)
{
        zfs_handle_t    *zhp = NULL;
        char            uu_string[UUID_PRINTABLE_STRING_LENGTH];
        int             ret = BE_SUCCESS;

        uuid_unparse(uu, uu_string);

        /* Get handle to the root zone dataset. */
        if ((zhp = zfs_open(g_zfs, root_ds, ZFS_TYPE_FILESYSTEM)) == NULL) {
                be_print_err(gettext("be_zone_set_parent_uuid: failed to "
                    "open root zone dataset (%s): %s\n"), root_ds,
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /* Set parentbe uuid property for the root zone dataset */
        if (zfs_prop_set(zhp, BE_ZONE_PARENTBE_PROPERTY, uu_string) != 0) {
                be_print_err(gettext("be_zone_set_parent_uuid: failed to "
                    "set parentbe uuid property for root zone dataset: %s\n"),
                    libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
        }

        ZFS_CLOSE(zhp);
        return (ret);
}

/*
```

```
 * Function:    be_zone_compare_uuids
 * Description: This function compare the parentbe uuid of the
 *              current running root zone dataset with the parentbe
 *              uuid of the given root zone dataset.
 * Parameters:
 *              root_ds - Root zone dataset of the BE to compare.
 * Return:
 *              B_TRUE - root dataset has right parentbe uuid
 *              B_FALSE - root dataset has wrong parentbe uuid
 * Scope:
 *              Semi-private (library wide uses only)
 */
boolean_t
be_zone_compare_uuids(char *root_ds)
{
        char            *active_ds;
        uuid_t          parent_uuid = {0};
        uuid_t          cur_parent_uuid = {0};

        /* Get parentbe uuid from given zone root dataset */
        if ((be_zone_get_parent_uuid(root_ds,
            &parent_uuid)) != BE_SUCCESS) {
                be_print_err(gettext("be_zone_compare_uuids: failed to get "
                    "parentbe uuid from the given BE\n"));
                return (B_FALSE);
        }

        /*
         * Find current running zone root dataset and get it's parentbe
         * uuid property.
         */
        if ((active_ds = be_get_ds_from_dir("/")) != NULL) {
                if ((be_zone_get_parent_uuid(active_ds,
                    &cur_parent_uuid)) != BE_SUCCESS) {
                        be_print_err(gettext("be_zone_compare_uuids: failed "
                        "to get parentbe uuid from the current running zone "
                        "root dataset\n"));
                        return (B_FALSE);
                }
        } else {
                be_print_err(gettext("be_zone_compare_uuids: zone root dataset "
                    "is not mounted\n"));
                return (B_FALSE);
        }

        if (uuid_compare(parent_uuid, cur_parent_uuid) != 0) {
                return (B_FALSE);
        }

        return (B_TRUE);
}

/* ******************************************************************** */
/*                      Private Functions                               */
/* ******************************************************************** */

/*
 * Function:    be_find_active_zone_root_callback
 * Description: This function is used as a callback to iterate over all of
 *              a zone's root datasets, finding the one that is marked active
 *              for the parent BE specified in the data passed in.  The name
 *              of the zone's active root dataset is returned in heap storage
 *              in the active_zone_root_data_t structure passed in, so the
 *              caller is responsible for freeing it.
 * Parameters:
 *              zhp - zfs_handle_t pointer to current dataset being processed
 *              data - active_zone_root_data_t pointer
 * Returns:
 *              0 - Success
 *              >0 - Failure
 * Scope:
 *              Private
 */
static int
be_find_active_zone_root_callback(zfs_handle_t *zhp, void *data)
{
        active_zone_root_data_t *azr_data = data;
        uuid_t                  parent_uuid = { 0 };
        int                     iret = 0;
        int                     ret = 0;

        if ((iret = be_zone_get_parent_uuid(zfs_get_name(zhp), &parent_uuid))
            != BE_SUCCESS) {
                be_print_err(gettext("be_find_active_zone_root_callback: "
                    "skipping zone root dataset (%s): %s\n"),
                    zfs_get_name(zhp), be_err_to_str(iret));
                goto done;
        }

        if (uuid_compare(azr_data->parent_uuid, parent_uuid) == 0) {
                /*
                 * Found a zone root dataset belonging to the right parent,
                 * check if its active.
                 */
                if (be_zone_get_active(zhp)) {
                        /*
                         * Found active zone root dataset, if its already
                         * set in the callback data, that means this
                         * is the second one we've found.  Return error.
                         */
                        if (azr_data->zoneroot_ds != NULL) {
                                ret = BE_ERR_ZONE_MULTIPLE_ACTIVE;
                                goto done;
                        }

                        azr_data->zoneroot_ds = strdup(zfs_get_name(zhp));
                        if (azr_data->zoneroot_ds == NULL) {
                                ret = BE_ERR_NOMEM;
                        }
                }
        }

done:
        ZFS_CLOSE(zhp);
        return (ret);
}
```

______***unchanged_portion_omitted_***

```
************************************************************
    8194 Tue Aug  6 21:14:55 2013
new/usr/src/lib/libbe/common/libbe.h
*** NO COMMENTS ***
************************************************************
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 */

#ifndef _LIBBE_H
#define _LIBBE_H

#include <libnvpair.h>
#include <uuid/uuid.h>
#include <libzfs.h>

#ifdef __cplusplus
extern "C" {
#endif

#define BE_ATTR_ORIG_BE_NAME    "orig_be_name"
#define BE_ATTR_ORIG_BE_POOL    "orig_be_pool"
#define BE_ATTR_SNAP_NAME       "snap_name"

#define BE_ATTR_NEW_BE_NAME     "new_be_name"
#define BE_ATTR_NEW_BE_POOL     "new_be_pool"
#define BE_ATTR_NEW_BE_DESC     "new_be_desc"
#define BE_ATTR_POLICY          "policy"
#define BE_ATTR_ZFS_PROPERTIES  "zfs_properties"

#define BE_ATTR_FS_NAMES        "fs_names"
#define BE_ATTR_FS_NUM          "fs_num"
#define BE_ATTR_SHARED_FS_NAMES "shared_fs_names"
#define BE_ATTR_SHARED_FS_NUM   "shared_fs_num"

#define BE_ATTR_MOUNTPOINT      "mountpoint"
#define BE_ATTR_MOUNT_FLAGS     "mount_flags"
#define BE_ATTR_UNMOUNT_FLAGS   "unmount_flags"
#define BE_ATTR_DESTROY_FLAGS   "destroy_flags"
#define BE_ATTR_ROOT_DS         "root_ds"
#define BE_ATTR_UUID_STR        "uuid_str"
```

```
#define BE_ATTR_ACTIVE          "active"
#define BE_ATTR_ACTIVE_ON_BOOT  "active_boot"
#define BE_ATTR_GLOBAL_ACTIVE   "global_active"
#define BE_ATTR_SPACE           "space_used"
#define BE_ATTR_DATASET         "dataset"
#define BE_ATTR_STATUS          "status"
#define BE_ATTR_DATE            "date"
#define BE_ATTR_MOUNTED         "mounted"

/*
 * libbe error codes
 *
 * NOTE: there is a copy of this enum in beadm/messages.py. To keep these
 *       in sync please make sure to add any new error messages at the end
 *       of this enumeration.
 */
enum {
        BE_SUCCESS = 0,
        BE_ERR_ACCESS = 4000,   /* permission denied */
        BE_ERR_ACTIVATE_CURR,   /* Activation of current BE failed */
        BE_ERR_AUTONAME,        /* auto naming failed */
        BE_ERR_BE_NOENT,        /* No such BE */
        BE_ERR_BUSY,            /* mount busy */
        BE_ERR_CANCELED,        /* operation canceled */
        BE_ERR_CLONE,           /* BE clone failed */
        BE_ERR_COPY,            /* BE copy failed */
        BE_ERR_CREATDS,         /* dataset creation failed */
        BE_ERR_CURR_BE_NOT_FOUND,       /* Can’t find current BE */
        BE_ERR_DESTROY,         /* failed to destroy BE or snapshot */
        BE_ERR_DEMOTE,          /* BE demotion failed */
        BE_ERR_DSTYPE,          /* invalid dataset type */
        BE_ERR_BE_EXISTS,       /* BE exists */
        BE_ERR_INIT,            /* be_zfs_init failed */
        BE_ERR_INTR,            /* interupted system call */
        BE_ERR_INVAL,           /* invalid argument */
        BE_ERR_INVALPROP,       /* invalid property for dataset */
        BE_ERR_INVALMOUNTPOINT, /* Unexpected mountpoint */
        BE_ERR_MOUNT,           /* mount failed */
        BE_ERR_MOUNTED,         /* already mounted */
        BE_ERR_NAMETOOLONG,     /* name > BUFSIZ */
        BE_ERR_NOENT,           /* Doesn’t exist */
        BE_ERR_POOL_NOENT,      /* No such pool */
        BE_ERR_NODEV,           /* No such device */
        BE_ERR_NOTMOUNTED,      /* File system not mounted */
        BE_ERR_NOMEM,           /* not enough memory */
        BE_ERR_NONINHERIT,      /* property is not inheritable for BE dataset */
        BE_ERR_NXIO,            /* No such device or address */
        BE_ERR_NOSPC,           /* No space on device */
        BE_ERR_NOTSUP,          /* Operation not supported */
        BE_ERR_OPEN,            /* open failed */
        BE_ERR_PERM,            /* Not owner */
        BE_ERR_UNAVAIL,         /* The BE is currently unavailable */
        BE_ERR_PROMOTE,         /* BE promotion failed */
        BE_ERR_ROFS,            /* read only file system */
        BE_ERR_READONLYDS,      /* read only dataset */
        BE_ERR_READONLYPROP,    /* read only property */
        BE_ERR_SS_EXISTS,       /* snapshot exists */
        BE_ERR_SS_NOENT,        /* No such snapshot */
        BE_ERR_UMOUNT,          /* unmount failed */
        BE_ERR_UMOUNT_CURR_BE,  /* Can’t unmount current BE */
        BE_ERR_UMOUNT_SHARED,   /* unmount of shared File System failed */
        BE_ERR_UNKNOWN,         /* Unknown error */
        BE_ERR_ZFS,             /* ZFS returned an error */
        BE_ERR_DESTROY_CURR_BE, /* Cannot destroy current BE */
        BE_ERR_GEN_UUID,        /* Failed to generate uuid */
        BE_ERR_PARSE_UUID,      /* Failed to parse uuid */
        BE_ERR_NO_UUID,         /* BE has no uuid */
        BE_ERR_ZONE_NO_PARENTBE,    /* Zone root dataset has no parent uuid */
        BE_ERR_ZONE_MULTIPLE_ACTIVE, /* Zone has multiple active roots */
        BE_ERR_ZONE_NO_ACTIVE_ROOT, /* Zone has no active root for this BE */
        BE_ERR_ZONE_ROOT_NOT_LEGACY, /* Zone root dataset mntpt is not legacy */
        BE_ERR_NO_MOUNTED_ZONE, /* Zone not mounted in alternate BE */
        BE_ERR_MOUNT_ZONEROOT,  /* Failed to mount a zone root */
        BE_ERR_UMOUNT_ZONEROOT, /* Failed to unmount a zone root */
        BE_ERR_ZONES_UNMOUNT,   /* Unable to unmount a zone. */
        BE_ERR_FAULT,           /* Bad Address */
        BE_ERR_RENAME_ACTIVE,   /* Renaming the active BE is not supported */
        BE_ERR_NO_MENU,         /* Missing boot menu file */
        BE_ERR_DEV_BUSY,        /* Device is Busy */
        BE_ERR_BAD_MENU_PATH,   /* Invalid path for menu.lst file */
        BE_ERR_ZONE_SS_EXISTS,  /* zone snapshot already exists */
        BE_ERR_ADD_SPLASH_ICT,  /* Add_splash_image ICT failed */
        BE_ERR_BOOTFILE_INST,   /* Error installing boot files */
        BE_ERR_EXTCMD           /* External command error */
} be_errno_t;
```
______***unchanged_portion_omitted_***

```
typedef struct be_node_list {
        boolean_t be_mounted;           /* is BE currently mounted */
        boolean_t be_active_on_boot;    /* is this BE active on boot */
        boolean_t be_active;            /* is this BE active currently */
        boolean_t be_global_active;     /* is zone's BE associated with */
                                        /* an active global BE */
        uint64_t be_space_used;
        char *be_node_name;
        char *be_rpool;
        char *be_root_ds;
        char *be_mntpt;
        char *be_policy_type;           /* cleanup policy type */
        char *be_uuid_str;              /* string representation of uuid */
        time_t be_node_creation;        /* Date/time stamp when created */
        struct be_dataset_list *be_node_datasets;
        uint_t be_node_num_datasets;
        struct be_snapshot_list *be_node_snapshots;
        uint_t be_node_num_snapshots;
        struct be_node_list *be_next_node;
} be_node_list_t;
```
______***unchanged_portion_omitted_***

```
**********************************************************
    7663 Tue Aug  6 21:14:56 2013
new/usr/src/lib/libbe/common/libbe_priv.h
*** NO COMMENTS ***
**********************************************************
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 */

#ifndef _LIBBE_PRIV_H
#define _LIBBE_PRIV_H

#include <libnvpair.h>
#include <libzfs.h>
#include <instzones_api.h>

#ifdef __cplusplus
extern "C" {
#endif

#define ARCH_LENGTH             MAXNAMELEN
#define BE_AUTO_NAME_MAX_TRY    3
#define BE_AUTO_NAME_DELIM      '-'
#define BE_DEFAULTS             "/etc/default/be"
#define BE_DFLT_BENAME_STARTS   "BENAME_STARTS_WITH="
#define BE_CONTAINER_DS_NAME    "ROOT"
#define BE_DEFAULT_CONSOLE      "text"
#define BE_POLICY_PROPERTY      "org.opensolaris.libbe:policy"
#define BE_UUID_PROPERTY        "org.opensolaris.libbe:uuid"
#define BE_PLCY_STATIC          "static"
#define BE_PLCY_VOLATILE        "volatile"
#define BE_GRUB_MENU            "/boot/grub/menu.lst"
#define BE_SPARC_MENU           "/boot/menu.lst"
#define BE_GRUB_COMMENT         "#============ End of LIBBE entry ============="
#define BE_GRUB_SPLASH          "splashimage /boot/solaris.xpm"
#define BE_GRUB_FOREGROUND      "foreground 343434"
#define BE_GRUB_BACKGROUND      "background F7FBFF"
#define BE_GRUB_DEFAULT         "default 0"
#define BE_WHITE_SPACE          " \t\r\n"
#define BE_CAP_FILE             "/boot/grub/capability"
```

```
#define BE_INSTALL_GRUB         "/sbin/installgrub"
#define BE_STAGE_1              "/boot/grub/stage1"
#define BE_STAGE_2              "/boot/grub/stage2"
#define ZFS_CLOSE(_zhp) \
        if (_zhp) { \
                zfs_close(_zhp); \
                _zhp = NULL; \
        }

#define BE_ZONE_PARENTBE_PROPERTY       "org.opensolaris.libbe:parentbe"
#define BE_ZONE_ACTIVE_PROPERTY         "org.opensolaris.libbe:active"
#define BE_ZONE_SUPPORTED_BRANDS        "ipkg labeled"
#define BE_ZONE_SUPPORTED_BRANDS_DELIM  " "

/* Maximum length for the BE name. */
#define BE_NAME_MAX_LEN         64

#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))

typedef struct be_transaction_data {
        char            *obe_name;      /* Original BE name */
        char            *obe_root_ds;   /* Original BE root dataset */
        char            *obe_zpool;     /* Original BE pool */
        char            *obe_snap_name; /* Original BE snapshot name */
        char            *obe_altroot;   /* Original BE altroot */
        char            *nbe_name;      /* New BE name */
        char            *nbe_root_ds;   /* New BE root dataset */
        char            *nbe_zpool;     /* New BE pool */
        char            *nbe_desc;      /* New BE description */
        nvlist_t        *nbe_zfs_props; /* New BE dataset properties */
        char            *policy;        /* BE policy type */
} be_transaction_data_t;
______unchanged_portion_omitted_

/* Library globals */
extern libzfs_handle_t *g_zfs;
extern boolean_t do_print;

/* be_create.c */
int be_set_uuid(char *);
int be_get_uuid(const char *, uuid_t *);

/* be_list.c */
int _be_list(char *, be_node_list_t **);
int be_get_zone_be_list(char *, char *, be_node_list_t **);

/* be_mount.c */
int _be_mount(char *, char **, int);
int _be_unmount(char *, int);
int be_mount_pool(zfs_handle_t *, char **, char **, boolean_t *);
int be_unmount_pool(zfs_handle_t *, char *, char *);
int be_mount_zone_root(zfs_handle_t *, be_mount_data_t *);
int be_unmount_zone_root(zfs_handle_t *, be_unmount_data_t *);
int be_get_legacy_fs(char *, char *, char *, char *, be_fs_list_data_t *);
void be_free_fs_list(be_fs_list_data_t *);
char *be_get_ds_from_dir(char *);
int be_make_tmp_mountpoint(char **);

/* be_snapshot.c */
int _be_create_snapshot(char *, char **, char *);
int _be_destroy_snapshot(char *, char *);

/* be_utils.c */
boolean_t be_zfs_init(void);
void be_zfs_fini(void);
void be_make_root_ds(const char *, const char *, char *, int);
void be_make_container_ds(const char *, char *, int);
char *be_make_name_from_ds(const char *, char *);
int be_append_menu(char *, char *, char *, char *, char *);
int be_remove_menu(char *, char *, char *);
int be_update_menu(char *, char *, char *, char *);
int be_default_grub_bootfs(const char *, char **);
boolean_t be_has_menu_entry(char *, char *, int *);
int be_run_cmd(char *, char *, int, char *, int);
int be_change_grub_default(char *, char *);
int be_update_vfstab(char *, char *, char *, be_fs_list_data_t *, char *);
int be_update_zone_vfstab(zfs_handle_t *, char *, char *, char *,
    be_fs_list_data_t *);
int be_maxsize_avail(zfs_handle_t *, uint64_t *);
char *be_auto_snap_name(void);
char *be_auto_be_name(char *);
char *be_auto_zone_be_name(char *, char *);
char *be_default_policy(void);
boolean_t valid_be_policy(char *);
boolean_t be_valid_auto_snap_name(char *);
boolean_t be_valid_be_name(const char *);
void be_print_err(char *, ...);
int be_find_current_be(be_transaction_data_t *);
int zfs_err_to_be_err(libzfs_handle_t *);
int errno_to_be_err(int);

/* be_activate.c */
int _be_activate(char *);
int be_activate_current_be(void);
boolean_t be_is_active_on_boot(char *);

/* be_zones.c */
void be_make_zoneroot(char *, char *, int);
int be_find_active_zone_root(zfs_handle_t *, char *, char *, int);
int be_find_mounted_zone_root(char *, char *, char *, int);
boolean_t be_zone_supported(char *);
zoneBrandList_t *be_get_supported_brandlist(void);
int be_zone_get_parent_uuid(const char *, uuid_t *);
int be_zone_set_parent_uuid(char *, uuid_t);
boolean_t be_zone_compare_uuids(char *);

/* check architecture functions */
char *be_get_default_isa(void);
boolean_t be_is_isa(char *);
boolean_t be_has_grub(void);

/* callback functions */
int be_exists_callback(zpool_handle_t *, void *);
int be_find_zpool_callback(zpool_handle_t *, void *);
int be_zpool_find_current_be_callback(zpool_handle_t *, void *);
int be_zfs_find_current_be_callback(zfs_handle_t *, void *);
int be_check_be_roots_callback(zpool_handle_t *, void *);

/* defaults */
void be_get_defaults(struct be_defaults *defaults);

#ifdef __cplusplus
}
```
______***unchanged_portion_omitted_***

```
************************************************************
   14898 Tue Aug  6 21:14:57 2013
new/usr/src/man/man1m/beadm.1m
*** NO COMMENTS ***
************************************************************
'\" te
.\" Copyright 2013 Nexenta Systems, Inc. All rights reserved.
.TH BEADM 1M "Jul 25, 2013"
.\" Copyright 2012 Nexenta Systems, Inc. All rights reserved.
.TH BEADM 1M "Feb 26, 2011"
.SH NAME
beadm \- utility for managing zfs boot environments
.SH SYNOPSIS
.LP
.nf
\fBbeadm\fR \fBcreate\fR [\fB-a\fR] [\fB-d\fR \fIdescription\fR]
     [\fB-e\fR \fInon-activeBeName\fR | \fIbeName@snapshot\fR]
     [\fB-o\fR \fIproperty=value\fR] ... [\fB-p\fR \fIzpool\fR]
     [\fB-v\fR] \fIbeName\fR
.fi

.LP
.nf
\fBbeadm\fR \fBcreate\fR [\fB-v\fR] \fIbeName@snapshot\fR
.fi

.LP
.nf
\fBbeadm\fR \fBdestroy\fR [\fB-fFsv\fR] \fIbeName\fR | \fIbeName@snapshot\fR
.fi

.LP
.nf
\fBbeadm\fR \fBlist\fR [\fB-a\fR | \fB-ds\fR] [\fB-H\fR] [\fB-v\fR] [\fIbeName\f
.fi

.LP
.nf
\fBbeadm\fR \fBmount\fR [\fB-v\fR] \fIbeName\fR \fImountpoint\fR
.fi

.LP
.nf
\fBbeadm\fR \fBunmount\fR [\fB-fv\fR] \fIbeName\fR | \fImountpoint\fR
.fi

.LP
.nf
\fBbeadm\fR \fBrename\fR [\fB-v\fR] \fIbeName\fR \fInewBeName\fR
.fi

.LP
.nf
\fBbeadm\fR \fBactivate\fR [\fB-v\fR] \fIbeName\fR
.fi

.LP
.nf
\fBbeadm\fR \fBrollback\fR [\fB-v\fR] \fIbeName\fR \fIsnapshot\fR
.fi

.LP
.nf
\fBbeadm\fR \fBrollback\fR [\fB-v\fR] \fIbeName@snapshot\fR
.fi
```

```
.SH DESCRIPTION
The \fBbeadm\fR command is the user interface for managing zfs Boot
Environments (BEs). This utility is intended to be used by System
Administrators who want to manage multiple Solaris Instances on a single
system.
.sp
The \fBbeadm\fR command supports the following operations:
.RS +4
.TP
.ie t \(bu
.el -
Create a new BE, based on the active BE.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Create a new BE, based on an inactive BE.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Create a snapshot of an existing BE.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Create a new BE, based on an existing snapshot.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Create a new BE, and copy it to a different zpool.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Activate an existing, inactive BE.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Mount a BE.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Unmount a BE.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Destroy a BE.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Destroy a snapshot of a BE.
```

```
.RE
.RS +4
.TP
.ie t \(bu
.el -
Rename an existing, inactive BE.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Roll back a BE to an existing snapshot of a BE.
.RE
.RS +4
.TP
.ie t \(bu
.el -
Display information about your snapshots and datasets.
.RE

.SH SUBCOMMANDS
The \fBbeadm\fR command has the subcommands and options listed
below. Also see
EXAMPLES below.
.sp
.ne 2
.na
\fBbeadm\fR
.ad
.sp .6
.RS 4n
Displays command usage.
.RE

.sp
.ne 2
.na
\fBbeadm\fR \fBcreate\fR [\fB-a\fR] [\fB-d\fR \fIdescription\fR]
     [\fB-e\fR \fInon-activeBeName\fR | \fIbeName@snapshot\fR]
     [\fB-o\fR \fIproperty=value\fR] ... [\fB-p\fR \fIzpool\fR]
     [\fB-v\fR] \fIbeName\fR

.ad
.sp .6
.RS 4n
Creates a new boot environment named \fIbeName\fR.  If the \fB-e\fR option is
not
provided, the new boot environment will be created as a clone of the
currently
running boot environment. If the \fB-d\fR option is provided then the
description is
also used as the title for the BE’s entry in the GRUB menu for
x86 systems or
in the boot menu for SPARC systems. If the \fB-d\fR option is
not provided, \fIbeName\fR
will be used as the title.
.sp
.ne 2
.na
\fB-a\fR
.ad
.sp .6
.RS 4n
Activate the newly created BE upon creation.  The default is to not activate
the newly created BE.
.RE
```

```
.sp
.ne 2
.na
\fB-d\fR \fIdescription\fR
.ad
.sp .6
.RS 4n
Create a new BE with a description associated with it.
.RE
.sp
.ne 2
.na
\fB-e\fR \fInon-activeBeName\fR
.ad
.sp .6
.RS 4n
Create a new BE from an existing inactive BE.
.RE
.sp
.ne 2
.na
\fB-e\fR \fIbeName@snapshot\fR
.ad
.sp .6
.RS 4n
Create a new BE from an existing snapshot of the BE named beName.
.RE
.sp
.ne 2
.na
\fB-o\fR \fIproperty=value\fR
.ad
.sp .6
.RS 4n
Create the datasets for new BE with specific ZFS properties.  Multiple
\fB-o\fR
options can be specified.  See \fBzfs\fR(1M) for more information on
the
\fB-o\fR option.
.RE
.sp
.ne 2
.na
\fB-p\fR \fIzpool\fR
.ad
.sp .6
.RS 4n
Create the new BE in the specified zpool.  If this is not provided, the
default
behavior is to create the new BE in the same pool as as the origin BE.
This option is not supported in non-global zone.
.RE
.sp
.ne 2
.na
\fB-v\fR
.ad
.sp .6
.RS 4n
Verbose mode. Displays verbose error messages from \fBbeadm\fR.
.RE
.RE

.sp
.ne 2
.na
```

```
\fBbeadm\fR \fBcreate\fR [\fB-v\fR] \fIbeName@snapshot\fR
.ad
.sp .6
.RS 4n
Creates a snapshot of the existing BE named beName.
.sp
.ne 2
.na
\fB-v\fR
.ad
.sp .6
.RS 4n
Verbose mode. Displays verbose error messages from \fBbeadm\fR.
.RE
.RE

.sp
.ne 2
.na
\fBbeadm\fR \fBdestroy\fR [\fB-fFsv\fR] \fIbeName\fR | \fIbeName@snapshot\fR
.ad
.sp .6
.RS 4n
Destroys the boot environment named \fIbeName\fR or destroys an existing
snapshot of
the boot environment named \fIbeName@snapshot\fR.  Destroying a
boot environment
will also destroy all snapshots of that boot environment.  Use
this command
with caution.
.sp
.ne 2
.na
\fB-f\fR
.ad
.sp .6
.RS 4n
Forcefully unmount the boot environment if it is currently mounted.
.RE
.sp
.ne 2
.na
\fB-F\fR
.ad
.sp .6
.RS 4n
Force the action without prompting to verify the destruction of the boot
environment.
.RE
.sp
.ne 2
.na
\fB-s\fR
.ad
.sp .6
.RS 4n
Destroy all snapshots of the boot
environment.
.RE
.sp
.ne 2
.na
\fB-v\fR
.ad
.sp .6
.RS 4n
```

```
Verbose mode. Displays verbose error messages from \fBbeadm\fR.
.RE
.RE

.sp
.ne 2
.na
\fBbeadm\fR \fBlist\fR [\fB-a\fR | \fB-ds\fR] [\fB-H\fR] [\fB-v\fR] [\fIbeName\f
.ad
.sp .6
.RS 4n
Lists information about the existing boot environment named \fIbeName\fR, or
lists
information for all boot environments if \fIbeName\fR is not provided.
The 'Active'
field indicates whether the boot environment is active now,
represented
by 'N'; active on reboot, represented by 'R'; or both, represented
by 'NR'. In non-global zone the 'Active' field also indicates whether the
boot environment has a non-active parent BE, represented by 'x'; is active
on boot in a non-active parent BE, represented by 'b'. Activate, rollback
and snapshot operations for boot environments from non-active global parent
BE aren't supported, destroy is allowed if these boot environments aren't
active on boot.
by 'NR'.
.sp
Each line in the machine parasable output has the boot environment name as the
first field.  The 'Space' field is displayed in bytes and the 'Created' field
is displayed in UTC format.  The \fB-H\fR option used with no other options
gives
the boot environment's uuid in the second field.  This field will be
blank if
the boot environment does not have a uuid. See the EXAMPLES section.
In non-global zones, this field shows the uuid of the parent BE.
.sp
.ne 2
.na
\fB-a\fR
.ad
.sp .6
.RS 4n
Lists all available information about the boot environment.  This includes
subordinate file systems and snapshots.
.RE
.sp
.ne 2
.na
\fB-d\fR
.ad
.sp .6
.RS 4n
Lists information about all subordinate file systems belonging to the boot
environment.
.RE
.sp
.ne 2
.na
\fB-s\fR
.ad
.sp .6
.RS 4n
Lists information about the snapshots of the boot environment.
.RE
.sp
.ne 2
.na
```

```
\fB-H\fR
.ad
.sp .6
.RS 4n
Do not list header information.  Each field in the list information is
separated by a semicolon.
.RE
.sp
.ne 2
.na
\fB-v\fR
.ad
.sp .6
.RS 4n
Verbose mode. Displays verbose error messages from \fBbeadm\fR.
.RE
.RE

.sp
.ne 2
.na
\fBbeadm\fR \fBmount\fR [\fB-v\fR] \fIbeName\fR \fImountpoint\fR
.ad
.sp .6
.RS 4n
Mounts a boot environment named beName at mountpoint.  mountpoint must be an
already existing empty directory.
.sp
.ne 2
.na
\fB-v\fR
.ad
.sp .6
.RS 4n
Verbose mode. Displays verbose error messages from \fBbeadm\fR.
.RE
.RE

.sp
.ne 2
.na
\fBbeadm\fR \fBunmount\fR [\fB-fv\fR] \fIbeName\fR | \fImountpoint\fR
.ad
.sp .6
.RS 4n
Unmounts the boot environment named beName. The command can also be given a path
beName mount point on the system.
.sp
.ne 2
.na
\fB-f\fR
.ad
.sp .6
.RS 4n
Forcefully unmount the boot environment even if its currently busy.
.RE
.sp
.ne 2
.na
\fB-v\fR
.ad
.sp .6
.RS 4n
Verbose mode. Displays verbose error messages from \fBbeadm\fR.
.RE
.RE
```

```

.sp
.ne 2
.na
\fBbeadm\fR \fBrename\fR [\fB-v\fR] \fIbeName\fR \fInewBeName\fR
.ad
.sp .6
.RS 4n
Renames the boot environment named \fIbeName\fR to \fInewBeName\fR.
.sp
.ne 2
.na
\fB-v\fR
.ad
.sp .6
.RS 4n
Verbose mode. Displays verbose error messages from \fBbeadm\fR.
.RE
.RE

.sp
.ne 2
.na
\fBbeadm\fR \fBrollback\fR [\fB-v\fR] \fIbeName\fR \fIsnapshot\fR | \fIbeName@sn
.ad
.sp .6
.RS 4n
Roll back the boot environment named \fIbeName\fR to existing snapshot
of the boot environment named \fIbeName@snapshot\fR.
.sp
.ne 2
.na
\fB-v\fR
.ad
.sp .6
.RS 4n
Verbose mode. Displays verbose error messages from \fBbeadm\fR.
.RE
.RE

.sp
.ne 2
.na
\fBbeadm\fR \fBactivate\fR [\fB-v\fR] \fIbeName\fR
.ad
.sp .6
.RS 4n
Makes beName the active BE on next reboot.
.sp
.ne 2
.na
\fB-v\fR
.ad
.sp .6
.RS 4n
Verbose mode. Displays verbose error messages from \fBbeadm\fR.
.RE
.RE

.SH ALTERNATE BE LOCATION
.LP
The alternate BE location outside rpool/ROOT can be configured
by modifying the BENAME_STARTS_WITH parameter in /etc/default/be.
For example: BENAME_STARTS_WITH=rootfs

.SH EXAMPLES
```

```
.LP
\fBExample 1\fR: Create a new BE named BE1, by cloning the current live BE.
.sp
.in +2
.nf
\fB# beadm create BE1\fR
.fi
.in -2
.sp

.LP
\fBExample 2\fR: Create a new BE named BE2, by cloning the existing inactive
BE
named BE1.
.sp
.in +2
.nf
\fB# beadm create -e BE1 BE2\fR
.fi
.in -2
.sp

.LP
\fBExample 3\fR: Create a snapshot named now of the existing BE named BE1.
.sp
.in +2
.nf
\fB# beadm create BE1@now\fR
.fi
.in -2
.sp

.LP
\fBExample 4\fR: Create a new BE named BE3, by cloning an existing snapshot of
BE1.
.sp
.in +2
.nf
\fB# beadm create -e BE1@now BE3\fR
.fi
.in -2
.sp

.LP
\fBExample 5\fR: Create a new BE named BE4 based on the currently running BE.
Create the new BE in rpool2.
.sp
.in +2
.nf
\fB# beadm create -p rpool2 BE4\fR
.fi
.in -2
.sp

.LP
\fBExample 6\fR: Create a new BE named BE5 based on the currently running BE.
Create the new BE in rpool2, and create its datasets with compression turned
on.
.sp
.in +2
.nf
\fB# beadm create -p rpool2 -o compression=on BE5\fR
.fi
.in -2
.sp
```

```
.LP
\fBExample 7\fR: Create a new BE named BE6 based on the currently running BE
and provide a description for it.
.sp
.in +2
.nf
\fB# beadm create -d "BE6 used as test environment" BE6\fR
.fi
.in -2
.sp

.LP
\fBExample 8\fR: Activate an existing, inactive BE named BE3.
.sp
.in +2
.nf
\fB# beadm activate BE3\fR
.fi
.in -2
.sp

.LP
\fBExample 9\fR: Mount the BE named BE3 at /mnt.
.sp
.in +2
.nf
\fB# beadm mount BE3 /mnt\fR
.fi
.in -2
.sp

.LP
\fBExample 10\fR: Unmount the mounted BE named BE3.
.sp
.in +2
.nf
\fB# beadm unmount BE3\fR
.fi
.in -2
.sp

.LP
\fBExample 11\fR: Destroy the BE named BE3 without verification.
.sp
.in +2
.nf
\fB# beadm destroy -f BE3\fR
.fi
.in -2
.sp

.LP
\fBExample 12\fR: Destroy the snapshot named now of BE1.
.sp
.in +2
.nf
\fB# beadm destroy BE1@now\fR
.fi
.in -2
.sp

.LP
\fBExample 13\fR: Rename the existing, inactive BE named BE1 to BE3.
.sp
.in +2
.nf
```

```
\fB# beadm rename BE1 BE3\fR
.fi
.in -2
.sp

.LP
\fBExample 14\fR: Roll back the BE named BE1 to snapshot BE1@now.
.sp
.in +2
.nf
\fB# beadm rollback BE1 BE1@now\fR
.fi
.in -2
.sp

.LP
\fBExample 15\fR: List all existing boot environments.

.sp
.in +2
.nf
\fB# beadm list\fR
BE  Active Mountpoint Space  Policy Created
--  ------ ---------- -----  ------ -------
BE2 -      -          72.0K  static 2008-05-21 12:26
BE3 -      -          332.0K static 2008-08-26 10:28
BE4 -      -          15.78M static 2008-09-05 18:20
BE5 NR     /          7.25G  static 2008-09-09 16:53
.fi
.in -2
.sp

.LP
\fBExample 16\fR: List all existing boot environmets and list all dataset and
snapshot information about those bootenvironments.

.sp
.in +2
.nf
\fB# beadm list -d -s\fR

BE/Dataset/Snapshot     Active Mountpoint Space   Policy Created
-------------------     ------ ---------- -----   ------ -------
BE2
   p/ROOT/BE2           -      -          36.0K   static 2008-05-21 12:26
   p/ROOT/BE2/opt       -      -          18.0K   static 2008-05-21 16:26
   p/ROOT/BE2/opt@now   -      -          0       static 2008-09-08 22:43
   p/ROOT/BE2@now       -      -          0       static 2008-09-08 22:43
BE3
   p/ROOT/BE3           -      -          192.0K  static 2008-08-26 10:28
   p/ROOT/BE3/opt       -      -          86.0K   static 2008-08-26 10:28
   p/ROOT/BE3/opt/local -      -          36.0K   static 2008-08-28 10:58
BE4
   p/ROOT/BE4           -      -          15.78M  static 2008-09-05 18:20
BE5
   p/ROOT/BE5           NR     /          6.10G   static 2008-09-09 16:53
   p/ROOT/BE5/opt       -      /opt       24.55M  static 2008-09-09 16:53
   p/ROOT/BE5/opt@bar   -      -          18.38M  static 2008-09-10 00:59
   p/ROOT/BE5/opt@foo   -      -          18.38M  static 2008-06-10 16:37
   p/ROOT/BE5@bar       -      -          139.44M static 2008-09-10 00:59
   p/ROOT/BE5@foo       -      -          912.85M static 2008-06-10 16:37
.fi
.in -2
.sp

\fBExample 17\fR: List all dataset and snapshot information about BE5
```

```
.sp
.in +2
.nf
\fB# beadm list -a BE5\fR

BE/Dataset/Snapshot   Active Mountpoint Space   Policy Created
-------------------   ------ ---------- -----   ------ -------
BE5
   p/ROOT/BE5         NR     /          6.10G   static 2008-09-09 16:53
   p/ROOT/BE5/opt     -      /opt       24.55M  static 2008-09-09 16:53
   p/ROOT/BE5/opt@bar -      -          18.38M  static 2008-09-10 00:59
   p/ROOT/BE5/opt@foo -      -          18.38M  static 2008-06-10 16:37
   p/ROOT/BE5@bar     -      -          139.44M static 2008-09-10 00:59
   p/ROOT/BE5@foo     -      -          912.85M static 2008-06-10 16:37
.fi
.in -2
.sp

.LP
\fBExample 18\fR: List machine parsable information about all boot
environments.

.sp
.in +2
.nf
\fB# beadm list -H\fR

BE2;;;;55296;static;1211397974
BE3;;;;339968;static;1219771706
BE4;;;;16541696;static;1220664051
BE5;215b8387-4968-627c-d2d0-f4a011414bab;NR;/;7786206208;static;1221004384
.fi
.in -2
.sp

.SH EXIT STATUS
.sp
.LP
The following exit values are returned:
.sp
.ne 2
.na
\fB0\fR
.ad
.sp .6
.RS 4n
Successful completion
.RE

.sp
.ne 2
.na
\fB>0\fR
.ad
.sp .6
.RS 4n
Failure
.RE


.SH FILES
.sp
.LP
.sp
.ne 2
```

```
.na
\fB/var/log/beadm/<beName>/create.log.<yyyymmdd_hhmmss>\fR
.ad
.sp .6
.RS 4n
Log used for capturing beadm create output
.sp
.nf
\fIyyyymmdd_hhmmss\fR - 20071130_140558
\fIyy\fR - year; 2007
\fImm\fR - month; 11
\fIdd\fR - day; 30
\fIhh\fR - hour; 14
\fImm\fR - minute; 05
\fIss\fR - second; 58
.fi
.in -2
.sp
.RE
.sp
.LP
.sp
.ne 2
.na
\fB/etc/default/be\fR
.ad
.sp .6
.RS 4n
Contains default value for BENAME_STARTS_WITH parameter
.sp
.RE

.SH ATTRIBUTES
.sp
.LP
See \fBattributes\fR(5) for descriptions of the  following  attributes:
.sp

.sp
.TS
box;
c | c
l | l .
ATTRIBUTE TYPE  ATTRIBUTE VALUE
_
Interface Stability     Uncommitted
.TE


.SH SEE ALSO
.sp
.LP
.BR zfs (1M)
```