```
**********************************************************
   42345 Wed Nov 11 10:43:14 2015
new/usr/src/lib/libbe/common/be_activate.c
patch zone-auto-create-be
**********************************************************
```

```
/*
 * 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 2015 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

#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 <errno.h>
#include <sys/mnttab.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/efi_partition.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 boolean_t be_do_install_mbr(char *, nvlist_t *);
static int be_do_installboot_helper(zpool_handle_t *, nvlist_t *, char *,
    char *, uint16_t);
static int be_do_installboot(be_transaction_data_t *, uint16_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                                 */
/* ********************************************************************* */

/*
 * 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);
}

/*
 * Function:    be_installboot
 * Description: Calls be_do_installboot to install/update bootloader on
 *              pool passed in through be_attrs. The primary consumer is
```

```
 *              bootadm command to avoid duplication of the code.
 * Parameters:
 *              be_attrs - pointer to nvlist_t of attributes being passed in.
 *                      The following attribute values are used:
 *
 *                      BE_ATTR_ORIG_BE_NAME            *required
 *                      BE_ATTR_ORIG_BE_POOL            *required
 *                      BE_ATTR_ORIG_BE_ROOT            *required
 *                      BE_ATTR_INSTALL_FLAGS           optional
 *
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Public
 */
int
be_installboot(nvlist_t *be_attrs)
{
        int             ret = BE_SUCCESS;
        uint16_t        flags = 0;
        uint16_t        verbose;
        be_transaction_data_t bt = { 0 };

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

        /* Set verbose early, so we get all messages */
        verbose = flags & BE_INSTALLBOOT_FLAG_VERBOSE;
        if (verbose == BE_INSTALLBOOT_FLAG_VERBOSE)
                libbe_print_errors(B_TRUE);

        ret = nvlist_lookup_string(be_attrs, BE_ATTR_ORIG_BE_NAME,
            &bt.obe_name);
        if (ret != 0) {
                be_print_err(gettext("be_installboot: failed to "
                    "lookup BE_ATTR_ORIG_BE_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }

        ret = nvlist_lookup_string(be_attrs, BE_ATTR_ORIG_BE_POOL,
            &bt.obe_zpool);
        if (ret != 0) {
                be_print_err(gettext("be_installboot: failed to "
                    "lookup BE_ATTR_ORIG_BE_POOL attribute\n"));
                return (BE_ERR_INVAL);
        }

        ret = nvlist_lookup_string(be_attrs, BE_ATTR_ORIG_BE_ROOT,
            &bt.obe_root_ds);
        if (ret != 0) {
                be_print_err(gettext("be_installboot: failed to "
                    "lookup BE_ATTR_ORIG_BE_ROOT attribute\n"));
                return (BE_ERR_INVAL);
        }

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

        ret = be_do_installboot(&bt, flags);
```



```
        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];
        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) {
                ret = be_do_installboot(&cb, BE_INSTALLBOOT_FLAG_NULL);
                if (ret != BE_SUCCESS)
                        return (ret);
```

```
260                 if (!be_has_menu_entry(root_ds, cb.obe_zpool, &entry)) {
261                         if ((ret = be_append_menu(cb.obe_name, cb.obe_zpool,
262                             NULL, NULL, NULL)) != BE_SUCCESS) {
263                                 be_print_err(gettext("be_activate: Failed to "
264                                     "add BE (%s) to the menu\n"),
265                                     cb.obe_name);
266                                 goto done;
267                         }
268                 }
269                 if (be_has_grub()) {
270                         if ((ret = be_change_grub_default(cb.obe_name,
271                             cb.obe_zpool)) != BE_SUCCESS) {
272                                 be_print_err(gettext("be_activate: failed to "
273                                     "change the default entry in menu.lst\n"));
274                                 goto done;
275                         }
276                 }
277         }

279         if ((ret = _be_list(cb.obe_name, &be_nodes)) != BE_SUCCESS) {
280                 return (ret);
281         }

283         if ((ret = set_canmount(be_nodes, "noauto")) != BE_SUCCESS) {
284                 be_print_err(gettext("be_activate: failed to set "
285                     "canmount dataset property\n"));
286                 goto done;
287         }

289         if (getzoneid() == GLOBAL_ZONEID) {
290                 if ((ret = set_bootfs(be_nodes->be_rpool,
291                     root_ds)) != BE_SUCCESS) {
292                         be_print_err(gettext("be_activate: failed to set "
293                             "bootfs pool property for %s\n"), root_ds);
294                         goto done;
295                 }
296         }

298         if ((zhp = zfs_open(g_zfs, root_ds, ZFS_TYPE_FILESYSTEM)) != NULL) {
299                 /*
300                  * We don't need to close the zfs handle at this
301                  * point because The callback funtion
302                  * be_promote_ds_callback() will close it for us.
303                  */
304                 if (be_promote_ds_callback(zhp, NULL) != 0) {
305                         be_print_err(gettext("be_activate: "
306                             "failed to activate the "
307                             "datasets for %s: %s\n"),
308                             root_ds,
309                             libzfs_error_description(g_zfs));
310                         ret = BE_ERR_PROMOTE;
311                         goto done;
312                 }
313         } else {
314                 be_print_err(gettext("be_activate: failed to open "
315                     "dataset (%s): %s\n"), root_ds,
316                     libzfs_error_description(g_zfs));
317                 ret = zfs_err_to_be_err(g_zfs);
318                 goto done;
319         }

321         if (getzoneid() == GLOBAL_ZONEID &&
322             be_get_uuid(cb.obe_root_ds, &uu) == BE_SUCCESS &&
323             (ret = be_promote_zone_ds(cb.obe_name, cb.obe_root_ds))
324             != BE_SUCCESS) {
325                 be_print_err(gettext("be_activate: failed to promote "
```



```
326                     "the active zonepath datasets for zones in BE %s\n"),
327                     cb.obe_name);
328         }

330         if (getzoneid() != GLOBAL_ZONEID) {
331                 if (!be_zone_compare_uuids(root_ds)) {
332                         be_print_err(gettext("be_activate: activating zone "
333                             "root dataset from non-active global BE is not "
334                             "supported\n"));
335                         ret = BE_ERR_NOTSUP;
336                         goto done;
337                 }
338                 if ((zhp = zfs_open(g_zfs, root_ds,
339                     ZFS_TYPE_FILESYSTEM)) == NULL) {
340                         be_print_err(gettext("be_activate: failed to open "
341                             "dataset (%s): %s\n"), root_ds,
342                             libzfs_error_description(g_zfs));
343                         ret = zfs_err_to_be_err(g_zfs);
344                         goto done;
345                 }
346                 /* Find current active zone root dataset */
347                 if ((ret = be_find_active_zone_root(zhp, cb.obe_zpool,
348                     active_ds, sizeof (active_ds))) != BE_SUCCESS) {
349                         be_print_err(gettext("be_activate: failed to find "
350                             "active zone root dataset\n"));
351                         ZFS_CLOSE(zhp);
352                         goto done;
353                 }
354                 /* Do nothing if requested BE is already active */
355                 if (strcmp(root_ds, active_ds) == 0) {
356                         ret = BE_SUCCESS;
357                         ZFS_CLOSE(zhp);
358                         goto done;
359                 }

361                 /* Set active property for BE */
362                 if (zfs_prop_set(zhp, BE_ZONE_ACTIVE_PROPERTY, "on") != 0) {
363                         be_print_err(gettext("be_activate: failed to set "
364                             "active property (%s): %s\n"), root_ds,
365                             libzfs_error_description(g_zfs));
366                         ret = zfs_err_to_be_err(g_zfs);
367                         ZFS_CLOSE(zhp);
368                         goto done;
369                 }
370                 ZFS_CLOSE(zhp);

372                 /* Unset active property for old active root dataset */
373                 if ((zhp = zfs_open(g_zfs, active_ds,
374                     ZFS_TYPE_FILESYSTEM)) == NULL) {
375                         be_print_err(gettext("be_activate: failed to open "
376                             "dataset (%s): %s\n"), active_ds,
377                             libzfs_error_description(g_zfs));
378                         ret = zfs_err_to_be_err(g_zfs);
379                         goto done;
380                 }
381                 if (zfs_prop_set(zhp, BE_ZONE_ACTIVE_PROPERTY, "off") != 0) {
382                         be_print_err(gettext("be_activate: failed to unset "
383                             "active property (%s): %s\n"), active_ds,
384                             libzfs_error_description(g_zfs));
385                         ret = zfs_err_to_be_err(g_zfs);
386                         ZFS_CLOSE(zhp);
387                         goto done;
388                 }
389                 ZFS_CLOSE(zhp);
390         }
391 done:
```

```
392         be_free_list(be_nodes);
393         return (ret);
394 }

396 /*
397  * Function:    be_activate_current_be
398  * Description: Set the currently "active" BE to be "active on boot"
399  * Paramters:
400  *              none
401  * Returns:
402  *              BE_SUCCESS - Success
403  *              be_errnot_t - Failure
404  * Scope:
405  *              Semi-private (library wide use only)
406  */
407 int
408 be_activate_current_be(void)
409 {
410         int ret = BE_SUCCESS;
411         be_transaction_data_t bt = { 0 };

413         if ((ret = be_find_current_be(&bt)) != BE_SUCCESS) {
414                 return (ret);
415         }

417         if ((ret = _be_activate(bt.obe_name)) != BE_SUCCESS) {
418                 be_print_err(gettext("be_activate_current_be: failed to "
419                     "activate %s\n"), bt.obe_name);
420                 return (ret);
421         }

423         return (BE_SUCCESS);
424 }

426 /*
427  * Function:    be_is_active_on_boot
428  * Description: Checks if the BE name passed in has the "active on boot"
429  *              property set to B_TRUE.
430  * Paramters:
431  *              be_name - the name of the BE to check
432  * Returns:
433  *              B_TRUE - if active on boot.
434  *              B_FALSE - if not active on boot.
435  * Scope:
436  *              Semi-private (library wide use only)
437  */
438 boolean_t
439 be_is_active_on_boot(char *be_name)
440 {
441         be_node_list_t *be_node = NULL;

443         if (be_name == NULL) {
444                 be_print_err(gettext("be_is_active_on_boot: "
445                     "be_name must not be NULL\n"));
446                 return (B_FALSE);
447         }

449         if (_be_list(be_name, &be_node) != BE_SUCCESS) {
450                 return (B_FALSE);
451         }

453         if (be_node == NULL) {
454                 return (B_FALSE);
455         }

457         if (be_node->be_active_on_boot) {
```



```
458                 be_free_list(be_node);
459                 return (B_TRUE);
460         } else {
461                 be_free_list(be_node);
462                 return (B_FALSE);
463         }
464 }

466 /* ************************************************************** */
467 /*                      Private Functions                         */
468 /* ************************************************************** */

470 /*
471  * Function:    set_bootfs
472  * Description: Sets the bootfs property on the boot pool to be the
473  *              root dataset of the activated BE.
474  * Parameters:
475  *              boot_pool - The pool we're setting bootfs in.
476  *              be_root_ds - The main dataset for the BE.
477  * Return:
478  *              BE_SUCCESS - Success
479  *              be_errno_t - Failure
480  * Scope:
481  *              Private
482  */
483 static int
484 set_bootfs(char *boot_rpool, char *be_root_ds)
485 {
486         zpool_handle_t *zhp;
487         int err = BE_SUCCESS;

489         if ((zhp = zpool_open(g_zfs, boot_rpool)) == NULL) {
490                 be_print_err(gettext("set_bootfs: failed to open pool "
491                     "(%s): %s\n"), boot_rpool, libzfs_error_description(g_zfs));
492                 err = zfs_err_to_be_err(g_zfs);
493                 return (err);
494         }

496         err = zpool_set_prop(zhp, "bootfs", be_root_ds);
497         if (err) {
498                 be_print_err(gettext("set_bootfs: failed to set "
499                     "bootfs property for pool %s: %s\n"), boot_rpool,
500                     libzfs_error_description(g_zfs));
501                 err = zfs_err_to_be_err(g_zfs);
502                 zpool_close(zhp);
503                 return (err);
504         }

506         zpool_close(zhp);
507         return (BE_SUCCESS);
508 }

510 /*
511  * Function:    set_canmount
512  * Description: Sets the canmount property on the datasets of the
513  *              activated BE.
514  * Parameters:
515  *              be_nodes - The be_node_t returned from be_list
516  *              value - The value of canmount we setting, on|off|noauto.
517  * Return:
518  *              BE_SUCCESS - Success
519  *              be_errno_t - Failure
520  * Scope:
521  *              Private
522  */
523 static int
```

```
set_canmount(be_node_list_t *be_nodes, char *value)
{
	char		ds_path[MAXPATHLEN];
	zfs_handle_t	*zhp = NULL;
	be_node_list_t	*list = be_nodes;
	int		err = BE_SUCCESS;

	while (list != NULL) {
		be_dataset_list_t *datasets = list->be_node_datasets;

		be_make_root_ds(list->be_rpool, list->be_node_name, ds_path,
		    sizeof (ds_path));

		if ((zhp = zfs_open(g_zfs, ds_path, ZFS_TYPE_DATASET)) ==
		    NULL) {
			be_print_err(gettext("set_canmount: failed to open "
			    "dataset (%s): %s\n"), ds_path,
			    libzfs_error_description(g_zfs));
			err = zfs_err_to_be_err(g_zfs);
			return (err);
		}
		if (zfs_prop_get_int(zhp, ZFS_PROP_MOUNTED)) {
			/*
			 * it's already mounted so we can't change the
			 * canmount property anyway.
			 */
			err = BE_SUCCESS;
		} else {
			err = zfs_prop_set(zhp,
			    zfs_prop_to_name(ZFS_PROP_CANMOUNT), value);
			if (err) {
				ZFS_CLOSE(zhp);
				be_print_err(gettext("set_canmount: failed to "
				    "set dataset property (%s): %s\n"),
				    ds_path, libzfs_error_description(g_zfs));
				err = zfs_err_to_be_err(g_zfs);
				return (err);
			}
		}
		ZFS_CLOSE(zhp);

		while (datasets != NULL) {
			be_make_root_ds(list->be_rpool,
			    datasets->be_dataset_name, ds_path,
			    sizeof (ds_path));

			if ((zhp = zfs_open(g_zfs, ds_path, ZFS_TYPE_DATASET))
			    == NULL) {
				be_print_err(gettext("set_canmount: failed to "
				    "open dataset %s: %s\n"), ds_path,
				    libzfs_error_description(g_zfs));
				err = zfs_err_to_be_err(g_zfs);
				return (err);
			}
			if (zfs_prop_get_int(zhp, ZFS_PROP_MOUNTED)) {
				/*
				 * it's already mounted so we can't change the
				 * canmount property anyway.
				 */
				err = BE_SUCCESS;
				ZFS_CLOSE(zhp);
				break;
			}
			err = zfs_prop_set(zhp,
			    zfs_prop_to_name(ZFS_PROP_CANMOUNT), value);
			if (err) {
```



```
				ZFS_CLOSE(zhp);
				be_print_err(gettext("set_canmount: "
				    "Failed to set property value %s "
				    "for dataset %s: %s\n"), value, ds_path,
				    libzfs_error_description(g_zfs));
				err = zfs_err_to_be_err(g_zfs);
				return (err);
			}
			ZFS_CLOSE(zhp);
			datasets = datasets->be_next_dataset;
		}
		list = list->be_next_node;
	}
	return (err);
}

/*
 * Function:	be_get_grub_vers
 * Description: Gets the grub version number from /boot/grub/capability. If
 *		capability file doesn't exist NULL is returned.
 * Parameters:
 *		bt - The transaction data for the BE we're getting the grub
 *		     version for.
 *		cur_vers - used to return the current version of grub from
 *			   the root pool.
 *		new_vers - used to return the grub version of the BE we're
 *			   activating.
 * Return:
 *		BE_SUCCESS - Success
 *		be_errno_t - Failed to find version
 * Scope:
 *		Private
 */
static int
be_get_grub_vers(be_transaction_data_t *bt, char **cur_vers, char **new_vers)
{
	zfs_handle_t	*zhp = NULL;
	zfs_handle_t	*pool_zhp = NULL;
	int ret = BE_SUCCESS;
	char cap_file[MAXPATHLEN];
	char *temp_mntpnt = NULL;
	char *zpool_mntpt = NULL;
	char *ptmp_mntpnt = NULL;
	char *orig_mntpnt = NULL;
	boolean_t be_mounted = B_FALSE;
	boolean_t pool_mounted = B_FALSE;

	if (!be_has_grub()) {
		be_print_err(gettext("be_get_grub_vers: Not supported on "
		    "this architecture\n"));
		return (BE_ERR_NOTSUP);
	}

	if (bt == NULL || bt->obe_name == NULL || bt->obe_zpool == NULL ||
	    bt->obe_root_ds == NULL) {
		be_print_err(gettext("be_get_grub_vers: Invalid BE\n"));
		return (BE_ERR_INVAL);
	}

	if ((pool_zhp = zfs_open(g_zfs, bt->obe_zpool, ZFS_TYPE_FILESYSTEM)) ==
	    NULL) {
		be_print_err(gettext("be_get_grub_vers: zfs_open failed: %s\n"),
		    libzfs_error_description(g_zfs));
		return (zfs_err_to_be_err(g_zfs));
	}
```

```
656         /*
657          * Check to see if the pool's dataset is mounted. If it isn't we'll
658          * attempt to mount it.
659          */
660         if ((ret = be_mount_pool(pool_zhp, &ptmp_mntpnt,
661             &orig_mntpnt, &pool_mounted)) != BE_SUCCESS) {
662                 be_print_err(gettext("be_get_grub_vers: pool dataset "
663                     "(%s) could not be mounted\n"), bt->obe_zpool);
664                 ZFS_CLOSE(pool_zhp);
665                 return (ret);
666         }

668         /*
669          * Get the mountpoint for the root pool dataset.
670          */
671         if (!zfs_is_mounted(pool_zhp, &zpool_mntpt)) {
672                 be_print_err(gettext("be_get_grub_vers: pool "
673                     "dataset (%s) is not mounted. Can't read the "
674                     "grub capability file.\n"), bt->obe_zpool);
675                 ret = BE_ERR_NO_MENU;
676                 goto cleanup;
677         }

679         /*
680          * get the version of the most recent grub update.
681          */
682         (void) snprintf(cap_file, sizeof (cap_file), "%s%s",
683             zpool_mntpt, BE_CAP_FILE);
684         free(zpool_mntpt);
685         zpool_mntpt = NULL;

687         if ((ret = get_ver_from_capfile(cap_file, cur_vers)) != BE_SUCCESS)
688                 goto cleanup;

690         if ((zhp = zfs_open(g_zfs, bt->obe_root_ds, ZFS_TYPE_FILESYSTEM)) ==
691             NULL) {
692                 be_print_err(gettext("be_get_grub_vers: failed to "
693                     "open BE root dataset (%s): %s\n"), bt->obe_root_ds,
694                     libzfs_error_description(g_zfs));
695                 free(cur_vers);
696                 ret = zfs_err_to_be_err(g_zfs);
697                 goto cleanup;
698         }
699         if (!zfs_is_mounted(zhp, &temp_mntpnt)) {
700                 if ((ret = _be_mount(bt->obe_name, &temp_mntpnt,
701                     BE_MOUNT_FLAG_NO_ZONES)) != BE_SUCCESS) {
702                         be_print_err(gettext("be_get_grub_vers: failed to "
703                             "mount BE (%s)\n"), bt->obe_name);
704                         free(*cur_vers);
705                         *cur_vers = NULL;
706                         ZFS_CLOSE(zhp);
707                         goto cleanup;
708                 }
709                 be_mounted = B_TRUE;
710         }
711         ZFS_CLOSE(zhp);

713         /*
714          * Now get the grub version for the BE being activated.
715          */
716         (void) snprintf(cap_file, sizeof (cap_file), "%s%s", temp_mntpnt,
717             BE_CAP_FILE);
718         ret = get_ver_from_capfile(cap_file, new_vers);
719         if (ret != BE_SUCCESS) {
720                 free(*cur_vers);
721                 *cur_vers = NULL;
```



```
722         }
723         if (be_mounted)
724                 (void) _be_unmount(bt->obe_name, 0);

726 cleanup:
727         if (pool_mounted) {
728                 int iret = BE_SUCCESS;
729                 iret = be_unmount_pool(pool_zhp, ptmp_mntpnt, orig_mntpnt);
730                 if (ret == BE_SUCCESS)
731                         ret = iret;
732                 free(orig_mntpnt);
733                 free(ptmp_mntpnt);
734         }
735         ZFS_CLOSE(pool_zhp);

737         free(temp_mntpnt);
738         return (ret);
739 }

741 /*
742  * Function:    get_ver_from_capfile
743  * Description: Parses the capability file passed in looking for the VERSION
744  *              line. If found the version is returned in vers, if not then
745  *              NULL is returned in vers.
746  *
747  * Parameters:
748  *              file - the path to the capability file we want to parse.
749  *              vers - the version string that will be passed back.
750  * Return:
751  *              BE_SUCCESS - Success
752  *              be_errno_t - Failed to find version
753  * Scope:
754  *              Private
755  */
756 static int
757 get_ver_from_capfile(char *file, char **vers)
758 {
759         FILE *fp = NULL;
760         char line[BUFSIZ];
761         char *last = NULL;
762         int err = BE_SUCCESS;
763         errno = 0;

765         if (!be_has_grub()) {
766                 be_print_err(gettext("get_ver_from_capfile: Not supported "
767                     "on this architecture\n"));
768                 return (BE_ERR_NOTSUP);
769         }

771         /*
772          * Set version string to NULL; the only case this shouldn't be set
773          * to be NULL is when we've actually found a version in the capability
774          * file, which is set below.
775          */
776         *vers = NULL;

778         /*
779          * If the capability file doesn't exist, we're returning success
780          * because on older releases, the capability file did not exist
781          * so this is a valid scenario.
782          */
783         if (access(file, F_OK) == 0) {
784                 if ((fp = fopen(file, "r")) == NULL) {
785                         err = errno;
786                         be_print_err(gettext("get_ver_from_capfile: failed to "
787                             "open file %s with error %s\n"), file,
```

```
788                             strerror(err));
789                     err = errno_to_be_err(err);
790                     return (err);
791             }

793             while (fgets(line, BUFSIZ, fp)) {
794                     char *tok = strtok_r(line, "=", &last);

796                     if (tok == NULL || tok[0] == '#') {
797                             continue;
798                     } else if (strcmp(tok, "VERSION") == 0) {
799                             *vers = strdup(last);
800                             break;
801                     }
802             }
803             (void) fclose(fp);
804     }

806     return (BE_SUCCESS);
807 }

809 /*
810  * To be able to boot EFI labeled disks, stage1 needs to be written
811  * into the MBR. We do not do this if we're on disks with a traditional
812  * fdisk partition table only, or if any foreign EFI partitions exist.
813  * In the trivial case of a whole-disk vdev we always write stage1 into
814  * the MBR.
815  */
816 static boolean_t
817 be_do_install_mbr(char *diskname, nvlist_t *child)
818 {
819     struct uuid allowed_uuids[] = {
820             EFI_UNUSED,
821             EFI_RESV1,
822             EFI_BOOT,
823             EFI_ROOT,
824             EFI_SWAP,
825             EFI_USR,
826             EFI_BACKUP,
827             EFI_RESV2,
828             EFI_VAR,
829             EFI_HOME,
830             EFI_ALTSCTR,
831             EFI_RESERVED,
832             EFI_SYSTEM,
833             EFI_BIOS_BOOT,
834             EFI_SYMC_PUB,
835             EFI_SYMC_CDS
836     };

838     uint64_t whole;
839     struct dk_gpt *gpt;
840     struct uuid *u;
841     int fd, npart, i, j;

843     (void) nvlist_lookup_uint64(child, ZPOOL_CONFIG_WHOLE_DISK,
844         &whole);

846     if (whole)
847             return (B_TRUE);

849     if ((fd = open(diskname, O_RDONLY|O_NDELAY)) < 0)
850             return (B_FALSE);

852     if ((npart = efi_alloc_and_read(fd, &gpt)) <= 0)
853             return (B_FALSE);
```



```
855     for (i = 0; i != npart; i++) {
856             int match = 0;

858             u = &gpt->efi_parts[i].p_guid;

860             for (j = 0;
861                 j != sizeof (allowed_uuids) / sizeof (struct uuid);
862                 j++)
863                     if (bcmp(u, &allowed_uuids[j],
864                         sizeof (struct uuid)) == 0)
865                             match++;

867             if (match == 0)
868                     return (B_FALSE);
869     }

871     return (B_TRUE);
872 }

874 static int
875 be_do_installboot_helper(zpool_handle_t *zphp, nvlist_t *child, char *stage1,
876     char *stage2, uint16_t flags)
877 {
878     char install_cmd[MAXPATHLEN];
879     char be_run_cmd_errbuf[BUFSIZ];
880     char be_run_cmd_outbuf[BUFSIZ];
881     char diskname[MAXPATHLEN];
882     char *vname;
883     char *path, *dsk_ptr;
884     char *flag = "";
885     int ret;
886     vdev_stat_t *vs;
887     uint_t vsc;

889     if (nvlist_lookup_string(child, ZPOOL_CONFIG_PATH, &path) != 0) {
890             be_print_err(gettext("be_do_installboot: "
891                 "failed to get device path\n"));
892             return (BE_ERR_NODEV);
893     }

895     if ((nvlist_lookup_uint64_array(child, ZPOOL_CONFIG_VDEV_STATS,
896         (uint64_t **)&vs, &vsc) != 0) ||
897         vs->vs_state < VDEV_STATE_DEGRADED) {
898             /*
899              * Don't try to run installgrub on a vdev that is not ONLINE
900              * or DEGRADED. Try to print a warning for each such vdev.
901              */
902             be_print_err(gettext("be_do_installboot: "
903                 "vdev %s is %s, can't install boot loader\n"),
904                 path, zpool_state_to_name(vs->vs_state, vs->vs_aux));
905             free(path);
906             return (BE_SUCCESS);
907     }

909     /*
910      * Modify the vdev path to point to the raw disk.
911      */
912     path = strdup(path);
913     if (path == NULL)
914             return (BE_ERR_NOMEM);

916     dsk_ptr = strstr(path, "/dsk/");
917     if (dsk_ptr != NULL) {
918             *dsk_ptr = '\0';
919             dsk_ptr++;
```

```
920         } else {
921                 dsk_ptr = "";
922         }
923
924         (void) snprintf(diskname, sizeof (diskname), "%s/r%s", path, dsk_ptr);
925         free(path);
926
927         vname = zpool_vdev_name(g_zfs, zphp, child, B_FALSE);
928         if (vname == NULL) {
929                 be_print_err(gettext("be_do_installboot: "
930                     "failed to get device name: %s\n"),
931                     libzfs_error_description(g_zfs));
932                 return (zfs_err_to_be_err(g_zfs));
933         }
934
935         if (be_is_isa("i386")) {
936                 uint16_t force = flags & BE_INSTALLBOOT_FLAG_FORCE;
937                 uint16_t mbr = flags & BE_INSTALLBOOT_FLAG_MBR;
938
939                 if (force == BE_INSTALLBOOT_FLAG_FORCE) {
940                         if (mbr == BE_INSTALLBOOT_FLAG_MBR ||
941                             be_do_install_mbr(diskname, child))
942                                 flag = "-F -m -f";
943                         else
944                                 flag = "-F";
945                 } else {
946                         if (mbr == BE_INSTALLBOOT_FLAG_MBR ||
947                             be_do_install_mbr(diskname, child))
948                                 flag = "-m -f";
949                 }
950
951                 (void) snprintf(install_cmd, sizeof (install_cmd),
952                     "%s %s %s %s %s", BE_INSTALL_GRUB, flag,
953                     stage1, stage2, diskname);
954         } else if (be_is_isa("sparc")) {
955                 if ((flags & BE_INSTALLBOOT_FLAG_FORCE) ==
956                     BE_INSTALLBOOT_FLAG_FORCE)
957                         flag = "-f -F zfs";
958                 else
959                         flag = "-F zfs";
960
961                 (void) snprintf(install_cmd, sizeof (install_cmd),
962                     "%s %s %s %s", BE_INSTALL_BOOT, flag, stage2, diskname);
963         } else {
964                 be_print_err(gettext("be_do_installboot: unsupported "
965                     "architecture.\n"));
966                 return (BE_ERR_BOOTFILE_INST);
967         }
968
969         *be_run_cmd_outbuf = '\0';
970         *be_run_cmd_errbuf = '\0';
971
972         ret = be_run_cmd(install_cmd, be_run_cmd_errbuf, BUFSIZ,
973             be_run_cmd_outbuf, BUFSIZ);
974
975         if (ret != BE_SUCCESS) {
976                 be_print_err(gettext("be_do_installboot: install "
977                     "failed for device %s.\n"), vname);
978                 ret = BE_ERR_BOOTFILE_INST;
979         }
980
981         be_print_err(gettext("  Command: \"%s\"\n"), install_cmd);
982         if (be_run_cmd_outbuf[0] != 0) {
983                 be_print_err(gettext("  Output:\n"));
984                 be_print_err("%s", be_run_cmd_outbuf);
985         }
```



```
986
987         if (be_run_cmd_errbuf[0] != 0) {
988                 be_print_err(gettext("  Errors:\n"));
989                 be_print_err("%s", be_run_cmd_errbuf);
990         }
991         free(vname);
992
993         return (ret);
994 }
995
996 /*
997  * Function:    be_do_copy_grub_cap
998  * Description: This function will copy grub capability file to BE.
999  *
1000 * Parameters:
1001 *              bt - The transaction data for the BE we're activating.
1002 * Return:
1003 *              BE_SUCCESS - Success
1004 *              be_errno_t - Failure
1005 *
1006 * Scope:
1007 *              Private
1008 */
1009 static int
1010 be_do_copy_grub_cap(be_transaction_data_t *bt)
1011 {
1012         zpool_handle_t  *zphp = NULL;
1013         zfs_handle_t    *zhp = NULL;
1014         char cap_file[MAXPATHLEN];
1015         char zpool_cap_file[MAXPATHLEN];
1016         char line[BUFSIZ];
1017         char *tmp_mntpnt = NULL;
1018         char *orig_mntpnt = NULL;
1019         char *pool_mntpnt = NULL;
1020         char *ptmp_mntpnt = NULL;
1021         FILE *cap_fp = NULL;
1022         FILE *zpool_cap_fp = NULL;
1023         int err = 0;
1024         int ret = BE_SUCCESS;
1025         boolean_t pool_mounted = B_FALSE;
1026         boolean_t be_mounted = B_FALSE;
1027
1028         /*
1029          * Copy the grub capability file from the BE we're activating
1030          * into the root pool.
1031          */
1032         zhp = zfs_open(g_zfs, bt->obe_zpool, ZFS_TYPE_FILESYSTEM);
1033         if (zhp == NULL) {
1034                 be_print_err(gettext("be_do_installboot: zfs_open "
1035                     "failed: %s\n"), libzfs_error_description(g_zfs));
1036                 zpool_close(zphp);
1037                 return (zfs_err_to_be_err(g_zfs));
1038         }
1039
1040         /*
1041          * Check to see if the pool's dataset is mounted. If it isn't we'll
1042          * attempt to mount it.
1043          */
1044         if ((ret = be_mount_pool(zhp, &ptmp_mntpnt,
1045             &orig_mntpnt, &pool_mounted)) != BE_SUCCESS) {
1046                 be_print_err(gettext("be_do_installboot: pool dataset "
1047                     "(%s) could not be mounted\n"), bt->obe_zpool);
1048                 ZFS_CLOSE(zhp);
1049                 zpool_close(zphp);
1050                 return (ret);
1051         }
```

```
1053         /*
1054          * Get the mountpoint for the root pool dataset.
1055          */
1056         if (!zfs_is_mounted(zhp, &pool_mntpnt)) {
1057                 be_print_err(gettext("be_do_installboot: pool "
1058                     "dataset (%s) is not mounted. Can't check the grub "
1059                     "version from the grub capability file.\n"), bt->obe_zpool);
1060                 ret = BE_ERR_NO_MENU;
1061                 goto done;
1062         }

1064         (void) snprintf(zpool_cap_file, sizeof (zpool_cap_file), "%s%s",
1065             pool_mntpnt, BE_CAP_FILE);

1067         free(pool_mntpnt);

1069         if ((zhp = zfs_open(g_zfs, bt->obe_root_ds, ZFS_TYPE_FILESYSTEM)) ==
1070             NULL) {
1071                 be_print_err(gettext("be_do_installboot: failed to "
1072                     "open BE root dataset (%s): %s\n"), bt->obe_root_ds,
1073                     libzfs_error_description(g_zfs));
1074                 ret = zfs_err_to_be_err(g_zfs);
1075                 goto done;
1076         }

1078         if (!zfs_is_mounted(zhp, &tmp_mntpnt)) {
1079                 if ((ret = _be_mount(bt->obe_name, &tmp_mntpnt,
1080                     BE_MOUNT_FLAG_NO_ZONES)) != BE_SUCCESS) {
1081                         be_print_err(gettext("be_do_installboot: failed to "
1082                             "mount BE (%s)\n"), bt->obe_name);
1083                         ZFS_CLOSE(zhp);
1084                         goto done;
1085                 }
1086                 be_mounted = B_TRUE;
1087         }
1088         ZFS_CLOSE(zhp);

1090         (void) snprintf(cap_file, sizeof (cap_file), "%s%s", tmp_mntpnt,
1091             BE_CAP_FILE);
1092         free(tmp_mntpnt);

1094         if ((cap_fp = fopen(cap_file, "r")) == NULL) {
1095                 err = errno;
1096                 be_print_err(gettext("be_do_installboot: failed to open grub "
1097                     "capability file\n"));
1098                 ret = errno_to_be_err(err);
1099                 goto done;
1100         }
1101         if ((zpool_cap_fp = fopen(zpool_cap_file, "w")) == NULL) {
1102                 err = errno;
1103                 be_print_err(gettext("be_do_installboot: failed to open new "
1104                     "grub capability file\n"));
1105                 ret = errno_to_be_err(err);
1106                 (void) fclose(cap_fp);
1107                 goto done;
1108         }

1110         while (fgets(line, BUFSIZ, cap_fp)) {
1111                 (void) fputs(line, zpool_cap_fp);
1112         }

1114         (void) fclose(zpool_cap_fp);
1115         (void) fclose(cap_fp);

1117 done:
```



```
1118         if (be_mounted)
1119                 (void) _be_unmount(bt->obe_name, 0);

1121         if (pool_mounted) {
1122                 int iret = 0;
1123                 iret = be_unmount_pool(zhp, ptmp_mntpnt, orig_mntpnt);
1124                 if (ret == BE_SUCCESS)
1125                         ret = iret;
1126                 free(orig_mntpnt);
1127                 free(ptmp_mntpnt);
1128         }
1129         return (ret);
1130 }

1132 /*
1133  * Function:    be_is_install_needed
1134  * Description: Check detached version files to detect if bootloader
1135  *              install/update is needed.
1136  *
1137  * Parameters:
1138  *              bt - The transaction data for the BE we're activating.
1139  *              update - set B_TRUE is update is needed.
1140  * Return:
1141  *              BE_SUCCESS - Success
1142  *              be_errno_t - Failure
1143  *
1144  * Scope:
1145  *              Private
1146  */
1147 static int
1148 be_is_install_needed(be_transaction_data_t *bt, boolean_t *update)
1149 {
1150         int     ret = BE_SUCCESS;
1151         char    *cur_vers = NULL, *new_vers = NULL;

1153         assert(bt != NULL);
1154         assert(update != NULL);

1156         if (!be_has_grub()) {
1157                 /*
1158                  * no detached versioning, let installboot to manage
1159                  * versioning.
1160                  */
1161                 *update = B_TRUE;
1162                 return (ret);
1163         }

1165         *update = B_FALSE;      /* set default */

1167         /*
1168          * We need to check to see if the version number from
1169          * the BE being activated is greater than the current
1170          * one.
1171          */
1172         ret = be_get_grub_vers(bt, &cur_vers, &new_vers);
1173         if (ret != BE_SUCCESS) {
1174                 be_print_err(gettext("be_activate: failed to get grub "
1175                     "versions from capability files.\n"));
1176                 return (ret);
1177         }
1178         /* update if we have both versions and can compare */
1179         if (cur_vers != NULL) {
1180                 if (new_vers != NULL) {
1181                         if (atof(cur_vers) < atof(new_vers))
1182                                 *update = B_TRUE;
1183                         free(new_vers);
```

```
1184                 }
1185                 free(cur_vers);
1186         } else if (new_vers != NULL) {
1187                 /* we only got new version - update */
1188                 *update = B_TRUE;
1189                 free(new_vers);
1190         }
1191         return (ret);
1192 }

1194 /*
1195  * Function:    be_do_installboot
1196  * Description: This function runs installgrub/installboot using the boot
1197  *              loader files from the BE we're activating and installing
1198  *              them on the pool the BE lives in.
1199  *
1200  * Parameters:
1201  *              bt - The transaction data for the BE we're activating.
1202  *              flags - flags for bootloader install
1203  * Return:
1204  *              BE_SUCCESS - Success
1205  *              be_errno_t - Failure
1206  *
1207  * Scope:
1208  *              Private
1209  */
1210 static int
1211 be_do_installboot(be_transaction_data_t *bt, uint16_t flags)
1212 {
1213         zpool_handle_t  *zphp = NULL;
1214         zfs_handle_t    *zhp = NULL;
1215         nvlist_t **child, *nv, *config;
1216         uint_t c, children = 0;
1217         char *tmp_mntpt = NULL;
1218         char stage1[MAXPATHLEN];
1219         char stage2[MAXPATHLEN];
1220         char *vname;
1221         int ret = BE_SUCCESS;
1222         boolean_t be_mounted = B_FALSE;
1223         boolean_t update = B_FALSE;

1225         /*
1226          * check versions. This call is to support detached
1227          * version implementation like grub. Embedded versioning is
1228          * checked by actual installer.
1229          */
1230         if ((flags & BE_INSTALLBOOT_FLAG_FORCE) != BE_INSTALLBOOT_FLAG_FORCE) {
1231                 ret = be_is_install_needed(bt, &update);
1232                 if (ret != BE_SUCCESS || update == B_FALSE)
1233                         return (ret);
1234         }

1236         if ((zhp = zfs_open(g_zfs, bt->obe_root_ds, ZFS_TYPE_FILESYSTEM)) ==
1237             NULL) {
1238                 be_print_err(gettext("be_do_installboot: failed to "
1239                     "open BE root dataset (%s): %s\n"), bt->obe_root_ds,
1240                     libzfs_error_description(g_zfs));
1241                 ret = zfs_err_to_be_err(g_zfs);
1242                 return (ret);
1243         }
1244         if (!zfs_is_mounted(zhp, &tmp_mntpt)) {
1245                 if ((ret = _be_mount(bt->obe_name, &tmp_mntpt,
1246                     BE_MOUNT_FLAG_NO_ZONES)) != BE_SUCCESS) {
1247                         be_print_err(gettext("be_do_installboot: failed to "
1248                             "mount BE (%s)\n"), bt->obe_name);
1249                         ZFS_CLOSE(zhp);
```



```
1250                         return (ret);
1251                 }
1252                 be_mounted = B_TRUE;
1253         }
1254         ZFS_CLOSE(zhp);

1256         if (be_has_grub()) {
1257                 (void) snprintf(stage1, sizeof (stage1), "%s%s",
1258                     tmp_mntpt, BE_GRUB_STAGE_1);
1259                 (void) snprintf(stage2, sizeof (stage2), "%s%s",
1260                     tmp_mntpt, BE_GRUB_STAGE_2);
1261         } else {
1262                 char *platform = be_get_platform();

1264                 if (platform == NULL) {
1265                         be_print_err(gettext("be_do_installboot: failed to "
1266                             "detect system platform name\n"));
1267                         if (be_mounted)
1268                                 (void) _be_unmount(bt->obe_name, 0);
1269                         free(tmp_mntpt);
1270                         return (BE_ERR_BOOTFILE_INST);
1271                 }

1273                 stage1[0] = '\0';       /* sparc has no stage1 */
1274                 (void) snprintf(stage2, sizeof (stage2),
1275                     "%s/usr/platform/%s%s", tmp_mntpt,
1276                     platform, BE_SPARC_BOOTBLK);
1277         }

1279         if ((zphp = zpool_open(g_zfs, bt->obe_zpool)) == NULL) {
1280                 be_print_err(gettext("be_do_installboot: failed to open "
1281                     "pool (%s): %s\n"), bt->obe_zpool,
1282                     libzfs_error_description(g_zfs));
1283                 ret = zfs_err_to_be_err(g_zfs);
1284                 if (be_mounted)
1285                         (void) _be_unmount(bt->obe_name, 0);
1286                 free(tmp_mntpt);
1287                 return (ret);
1288         }

1290         if ((config = zpool_get_config(zphp, NULL)) == NULL) {
1291                 be_print_err(gettext("be_do_installboot: failed to get zpool "
1292                     "configuration information. %s\n"),
1293                     libzfs_error_description(g_zfs));
1294                 ret = zfs_err_to_be_err(g_zfs);
1295                 goto done;
1296         }

1298         /*
1299          * Get the vdev tree
1300          */
1301         if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nv) != 0) {
1302                 be_print_err(gettext("be_do_installboot: failed to get vdev "
1303                     "tree: %s\n"), libzfs_error_description(g_zfs));
1304                 ret = zfs_err_to_be_err(g_zfs);
1305                 goto done;
1306         }

1308         if (nvlist_lookup_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN, &child,
1309             &children) != 0) {
1310                 be_print_err(gettext("be_do_installboot: failed to traverse "
1311                     "the vdev tree: %s\n"), libzfs_error_description(g_zfs));
1312                 ret = zfs_err_to_be_err(g_zfs);
1313                 goto done;
1314         }
1315         for (c = 0; c < children; c++) {
```

```
1316                uint_t i, nchildren = 0;
1317                nvlist_t **nvchild;
1318                vname = zpool_vdev_name(g_zfs, zphp, child[c], B_FALSE);
1319                if (vname == NULL) {
1320                        be_print_err(gettext(
1321                            "be_do_installboot: "
1322                            "failed to get device name: %s\n"),
1323                            libzfs_error_description(g_zfs));
1324                        ret = zfs_err_to_be_err(g_zfs);
1325                        goto done;
1326                }
1327                if (strcmp(vname, "mirror") == 0 || vname[0] != 'c') {
1328                        free(vname);

1330                        if (nvlist_lookup_nvlist_array(child[c],
1331                            ZPOOL_CONFIG_CHILDREN, &nvchild, &nchildren) != 0) {
1332                                be_print_err(gettext("be_do_installboot: "
1333                                    "failed to traverse the vdev tree: %s\n"),
1334                                    libzfs_error_description(g_zfs));
1335                                ret = zfs_err_to_be_err(g_zfs);
1336                                goto done;
1337                        }

1339                        for (i = 0; i < nchildren; i++) {
1340                                ret = be_do_installboot_helper(zphp, nvchild[i],
1341                                    stage1, stage2, flags);
1342                                if (ret != BE_SUCCESS)
1343                                        goto done;
1344                        }
1345                } else {
1346                        free(vname);

1348                        ret = be_do_installboot_helper(zphp, child[c], stage1,
1349                            stage2, flags);
1350                        if (ret != BE_SUCCESS)
1351                                goto done;
1352                }
1353        }

1355        if (be_has_grub()) {
1356                ret = be_do_copy_grub_cap(bt);
1357        }

1359 done:
1360        ZFS_CLOSE(zhp);
1361        if (be_mounted)
1362                (void) _be_unmount(bt->obe_name, 0);
1363        zpool_close(zphp);
1364        free(tmp_mntpt);
1365        return (ret);
1366 }

1368 /*
1369  * Function:    be_promote_zone_ds
1370  * Description: This function finds the zones for the BE being activated
1371  *              and the active zonepath dataset for each zone. Then each
1372  *              active zonepath dataset is promoted.
1373  *
1374  * Parameters:
1375  *              be_name - the name of the global zone BE that we need to
1376  *                        find the zones for.
1377  *              be_root_ds - the root dataset for be_name.
1378  * Return:
1379  *              BE_SUCCESS - Success
1380  *              be_errno_t - Failure
1381  *
```



```
1382  * Scope:
1383  *              Private
1384  */
1385 static int
1386 be_promote_zone_ds(char *be_name, char *be_root_ds)
1387 {
1388        char            *zone_ds = NULL;
1389        char            *temp_mntpt = NULL;
1390        char            origin[MAXPATHLEN];
1391        char            zoneroot_ds[MAXPATHLEN];
1392        zfs_handle_t    *zhp = NULL;
1393        zfs_handle_t    *z_zhp = NULL;
1394        zoneList_t      zone_list = NULL;
  28        zoneBrandList_t *brands = NULL;
1395        boolean_t       be_mounted = B_FALSE;
1396        int             zone_index = 0;
1397        int             err = BE_SUCCESS;

  33        /*
  34         * Get the supported zone brands so we can pass that
  35         * to z_get_nonglobal_zone_list_by_brand. Currently
  36         * only the ipkg and labeled brand zones are supported
  37         *
  38         */
  39        if ((brands = be_get_supported_brandlist()) == NULL) {
  40                be_print_err(gettext("be_promote_zone_ds: no supported "
  41                    "brands\n"));
  42                return (BE_SUCCESS);
  43        }

1399        if ((zhp = zfs_open(g_zfs, be_root_ds,
1400            ZFS_TYPE_FILESYSTEM)) == NULL) {
1401                be_print_err(gettext("be_promote_zone_ds: Failed to open "
1402                    "dataset (%s): %s\n"), be_root_ds,
1403                    libzfs_error_description(g_zfs));
1404                err = zfs_err_to_be_err(g_zfs);
  51                z_free_brand_list(brands);
1405                return (err);
1406        }

1408        if (!zfs_is_mounted(zhp, &temp_mntpt)) {
1409                if ((err = _be_mount(be_name, &temp_mntpt,
1410                    BE_MOUNT_FLAG_NO_ZONES)) != BE_SUCCESS) {
1411                        be_print_err(gettext("be_promote_zone_ds: failed to "
1412                            "mount the BE for zones procesing.\n"));
1413                        ZFS_CLOSE(zhp);
  61                        z_free_brand_list(brands);
1414                        return (err);
1415                }
1416                be_mounted = B_TRUE;
1417        }

1419        /*
1420         * Set the zone root to the temp mount point for the BE we just mounted.
1421         */
1422        z_set_zone_root(temp_mntpt);

1424        /*
1425         * If no zones are found, unmount the BE and move on.
  73         * Get all the zones based on the brands we're looking for. If no zones
  74         * are found that we're interested in unmount the BE and move on.
1426         */
1427        if ((zone_list = z_get_nonglobal_branded_zone_list()) == NULL) {
  76        if ((zone_list = z_get_nonglobal_zone_list_by_brand(brands)) == NULL) {
1428                if (be_mounted)
1429                        (void) _be_unmount(be_name, 0);
```

```
1430                ZFS_CLOSE(zhp);
  80                z_free_brand_list(brands);
1431                free(temp_mntpt);
1432                return (BE_SUCCESS);
1433        }
1434        for (zone_index = 0; z_zlist_get_zonename(zone_list, zone_index)
1435            != NULL; zone_index++) {
1436                char *zone_path = NULL;
1437                boolean_t auto_create;

1439                if (z_zlist_is_zone_auto_create_be(zone_list, zone_index,
1440                    &auto_create) != 0) {
1441                        be_print_err(gettext("be_promote_zone_ds: "
1442                            "Failed to get auto-create-be brand property\n"));
1443                        err = -1; // XXX
1444                        goto done;
1445                }

1447                if (!auto_create)
1448                        continue;
1449 #endif /* ! codereview */

1451                /* Skip zones that aren't at least installed */
1452                if (z_zlist_get_current_state(zone_list, zone_index) <
1453                    ZONE_STATE_INSTALLED)
1454                        continue;

1456                if (((zone_path =
1457                    z_zlist_get_zonepath(zone_list, zone_index)) == NULL) ||
1458                    ((zone_ds = be_get_ds_from_dir(zone_path)) == NULL) ||
1459                    !be_zone_supported(zone_ds))
1460                        continue;

1462                if (be_find_active_zone_root(zhp, zone_ds,
1463                    zoneroot_ds, sizeof (zoneroot_ds)) != 0) {
1464                        be_print_err(gettext("be_promote_zone_ds: "
1465                            "Zone does not have an active root "
1466                            "dataset, skipping this zone.\n"));
1467                        continue;
1468                }

1470                if ((z_zhp = zfs_open(g_zfs, zoneroot_ds,
1471                    ZFS_TYPE_FILESYSTEM)) == NULL) {
1472                        be_print_err(gettext("be_promote_zone_ds: "
1473                            "Failed to open dataset "
1474                            "(%s): %s\n"), zoneroot_ds,
1475                            libzfs_error_description(g_zfs));
1476                        err = zfs_err_to_be_err(g_zfs);
1477                        goto done;
1478                }

1480                if (zfs_prop_get(z_zhp, ZFS_PROP_ORIGIN, origin,
1481                    sizeof (origin), NULL, NULL, 0, B_FALSE) != 0) {
1482                        ZFS_CLOSE(z_zhp);
1483                        continue;
1484                }

1486                /*
1487                 * We don't need to close the zfs handle at this
1488                 * point because the callback funtion
1489                 * be_promote_ds_callback() will close it for us.
1490                 */
1491                if (be_promote_ds_callback(z_zhp, NULL) != 0) {
1492                        be_print_err(gettext("be_promote_zone_ds: "
1493                            "failed to activate the "
1494                            "datasets for %s: %s\n"),
```



```
1495                            zoneroot_ds,
1496                            libzfs_error_description(g_zfs));
1497                        err = BE_ERR_PROMOTE;
1498                        goto done;
1499                }
1500        }
1501 done:
1502        if (be_mounted)
1503                (void) _be_unmount(be_name, 0);
1504        ZFS_CLOSE(zhp);
1505        free(temp_mntpt);
  87        z_free_brand_list(brands);
1506        z_free_zone_list(zone_list);
1507        return (err);
1508 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   86468 Wed Nov 11 10:43:14 2015
new/usr/src/lib/libbe/common/be_create.c
patch zone-auto-create-be
**********************************************************
```

```
/*
 * 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 (c) 2014 by Delphix. All rights reserved.
 * Copyright 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

/*
 * 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"));
```

```
128                 return (BE_ERR_INVAL);
129         }

131         /* Validate new BE name */
132         if (!be_valid_be_name(bt.nbe_name)) {
133                 be_print_err(gettext("be_init: invalid BE name %s\n"),
134                     bt.nbe_name);
135                 return (BE_ERR_INVAL);
136         }

138         /* Get zpool name */
139         if (nvlist_lookup_string(be_attrs, BE_ATTR_NEW_BE_POOL, &bt.nbe_zpool)
140             != 0) {
141                 be_print_err(gettext("be_init: failed to lookup "
142                     "BE_ATTR_NEW_BE_POOL attribute\n"));
143                 return (BE_ERR_INVAL);
144         }

146         /* Get file system attributes */
147         nelem = 0;
148         if (nvlist_lookup_pairs(be_attrs, 0,
149             BE_ATTR_FS_NUM, DATA_TYPE_UINT16, &fs_num,
150             BE_ATTR_FS_NAMES, DATA_TYPE_STRING_ARRAY, &fs_names, &nelem,
151             NULL) != 0) {
152                 be_print_err(gettext("be_init: failed to lookup fs "
153                     "attributes\n"));
154                 return (BE_ERR_INVAL);
155         }
156         if (nelem != fs_num) {
157                 be_print_err(gettext("be_init: size of FS_NAMES array (%d) "
158                     "does not match FS_NUM (%d)\n"), nelem, fs_num);
159                 return (BE_ERR_INVAL);
160         }

162         /* Get shared file system attributes */
163         nelem = 0;
164         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
165             BE_ATTR_SHARED_FS_NUM, DATA_TYPE_UINT16, &shared_fs_num,
166             BE_ATTR_SHARED_FS_NAMES, DATA_TYPE_STRING_ARRAY, &shared_fs_names,
167             &nelem, NULL) != 0) {
168                 be_print_err(gettext("be_init: failed to lookup "
169                     "shared fs attributes\n"));
170                 return (BE_ERR_INVAL);
171         }
172         if (nelem != shared_fs_num) {
173                 be_print_err(gettext("be_init: size of SHARED_FS_NAMES "
174                     "array does not match SHARED_FS_NUM\n"));
175                 return (BE_ERR_INVAL);
176         }

178         /* Verify that nbe_zpool exists */
179         if ((zlp = zpool_open(g_zfs, bt.nbe_zpool)) == NULL) {
180                 be_print_err(gettext("be_init: failed to "
181                     "find existing zpool (%s): %s\n"), bt.nbe_zpool,
182                     libzfs_error_description(g_zfs));
183                 return (zfs_err_to_be_err(g_zfs));
184         }
185         zpool_close(zlp);

187         /*
188          * Verify BE container dataset in nbe_zpool exists.
189          * If not, create it.
190          */
191         if (!be_create_container_ds(bt.nbe_zpool))
192                 return (BE_ERR_CREATDS);
```



```
194         /*
195          * Verify that nbe_name doesn't already exist in some pool.
196          */
197         if ((zret = zpool_iter(g_zfs, be_exists_callback, bt.nbe_name)) > 0) {
198                 be_print_err(gettext("be_init: BE (%s) already exists\n"),
199                     bt.nbe_name);
200                 return (BE_ERR_BE_EXISTS);
201         } else if (zret < 0) {
202                 be_print_err(gettext("be_init: zpool_iter failed: %s\n"),
203                     libzfs_error_description(g_zfs));
204                 return (zfs_err_to_be_err(g_zfs));
205         }

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

211         /*
212          * Create property list for new BE root dataset.  If some
213          * zfs properties were already provided by the caller, dup
214          * that list.  Otherwise initialize a new property list.
215          */
216         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
217             BE_ATTR_ZFS_PROPERTIES, DATA_TYPE_NVLIST, &zfs_props, NULL)
218             != 0) {
219                 be_print_err(gettext("be_init: failed to lookup "
220                     "BE_ATTR_ZFS_PROPERTIES attribute\n"));
221                 return (BE_ERR_INVAL);
222         }
223         if (zfs_props != NULL) {
224                 /* Make sure its a unique nvlist */
225                 if (!(zfs_props->nvl_nvflag & NV_UNIQUE_NAME) &&
226                     !(zfs_props->nvl_nvflag & NV_UNIQUE_NAME_TYPE)) {
227                         be_print_err(gettext("be_init: ZFS property list "
228                             "not unique\n"));
229                         return (BE_ERR_INVAL);
230                 }

232                 /* Dup the list */
233                 if (nvlist_dup(zfs_props, &bt.nbe_zfs_props, 0) != 0) {
234                         be_print_err(gettext("be_init: failed to dup ZFS "
235                             "property list\n"));
236                         return (BE_ERR_NOMEM);
237                 }
238         } else {
239                 /* Initialize new nvlist */
240                 if (nvlist_alloc(&bt.nbe_zfs_props, NV_UNIQUE_NAME, 0) != 0) {
241                         be_print_err(gettext("be_init: internal "
242                             "error: out of memory\n"));
243                         return (BE_ERR_NOMEM);
244                 }
245         }

247         /* Set the mountpoint property for the root dataset */
248         if (nvlist_add_string(bt.nbe_zfs_props,
249             zfs_prop_to_name(ZFS_PROP_MOUNTPOINT), "/") != 0) {
250                 be_print_err(gettext("be_init: internal error "
251                     "out of memory\n"));
252                 ret = BE_ERR_NOMEM;
253                 goto done;
254         }

256         /* Set the 'canmount' property */
257         if (nvlist_add_string(bt.nbe_zfs_props,
258             zfs_prop_to_name(ZFS_PROP_CANMOUNT), "noauto") != 0) {
259                 be_print_err(gettext("be_init: internal error "
```

```
260                     "out of memory\n"));
261             ret = BE_ERR_NOMEM;
262             goto done;
263         }

265         /* Create BE root dataset for the new BE */
266         if (zfs_create(g_zfs, nbe_root_ds, ZFS_TYPE_FILESYSTEM,
267             bt.nbe_zfs_props) != 0) {
268             be_print_err(gettext("be_init: failed to "
269                 "create BE root dataset (%s): %s\n"), nbe_root_ds,
270                 libzfs_error_description(g_zfs));
271             ret = zfs_err_to_be_err(g_zfs);
272             goto done;
273         }

275         /* Set UUID for new BE */
276         if ((ret = be_set_uuid(nbe_root_ds)) != BE_SUCCESS) {
277             be_print_err(gettext("be_init: failed to "
278                 "set uuid for new BE\n"));
279         }

281         /*
282          * Clear the mountpoint property so that the non-shared
283          * file systems created below inherit their mountpoints.
284          */
285         (void) nvlist_remove(bt.nbe_zfs_props,
286             zfs_prop_to_name(ZFS_PROP_MOUNTPOINT), DATA_TYPE_STRING);

288         /* Create the new BE's non-shared file systems */
289         for (i = 0; i < fs_num && fs_names[i]; i++) {
290             /*
291              * If fs == "/", skip it;
292              * we already created the root dataset
293              */
294             if (strcmp(fs_names[i], "/") == 0)
295                 continue;

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

301             /* Create file system */
302             if (zfs_create(g_zfs, child_fs, ZFS_TYPE_FILESYSTEM,
303                 bt.nbe_zfs_props) != 0) {
304                 be_print_err(gettext("be_init: failed to create "
305                     "BE's child dataset (%s): %s\n"), child_fs,
306                     libzfs_error_description(g_zfs));
307                 ret = zfs_err_to_be_err(g_zfs);
308                 goto done;
309             }
310         }

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

316             if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0) {
317                 be_print_err(gettext("be_init: nvlist_alloc failed\n"));
318                 ret = BE_ERR_NOMEM;
319                 goto done;
320             }

322             for (i = 0; i < shared_fs_num; i++) {
323                 /* Generate string for shared file system */
324                 (void) snprintf(child_fs, sizeof (child_fs), "%s%s",
325                     bt.nbe_zpool, shared_fs_names[i]);
```



```
327                 if (nvlist_add_string(props,
328                     zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
329                     shared_fs_names[i]) != 0) {
330                     be_print_err(gettext("be_init: "
331                         "internal error: out of memory\n"));
332                     nvlist_free(props);
333                     ret = BE_ERR_NOMEM;
334                     goto done;
335                 }

337                 /* Create file system if it doesn't already exist */
338                 if (zfs_dataset_exists(g_zfs, child_fs,
339                     ZFS_TYPE_FILESYSTEM)) {
340                     continue;
341                 }
342                 if (zfs_create(g_zfs, child_fs, ZFS_TYPE_FILESYSTEM,
343                     props) != 0) {
344                     be_print_err(gettext("be_init: failed to "
345                         "create BE's shared dataset (%s): %s\n"),
346                         child_fs, libzfs_error_description(g_zfs));
347                     ret = zfs_err_to_be_err(g_zfs);
348                     nvlist_free(props);
349                     goto done;
350                 }
351             }

353             nvlist_free(props);
354         }

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

360         be_zfs_fini();

362         return (ret);
363 }

365 /*
366  * Function:    be_destroy
367  * Description: Destroy a BE and all of its children datasets, snapshots and
368  *              zones that belong to the parent BE.
369  * Parameters:
370  *              be_attrs - pointer to nvlist_t of attributes being passed in.
371  *                      The following attributes are used by this function:
372  *
373  *                      BE_ATTR_ORIG_BE_NAME            *required
374  *                      BE_ATTR_DESTROY_FLAGS           *optional
375  * Return:
376  *              BE_SUCCESS - Success
377  *              be_errno_t - Failure
378  * Scope:
379  *              Public
380  */
381 int
382 be_destroy(nvlist_t *be_attrs)
383 {
384         zfs_handle_t            *zhp = NULL;
385         be_transaction_data_t   bt = { 0 };
386         be_transaction_data_t   cur_bt = { 0 };
387         be_destroy_data_t       dd = { 0 };
388         int                     ret = BE_SUCCESS;
389         uint16_t                flags = 0;
390         boolean_t               bs_found = B_FALSE;
391         int                     zret;
```

```
392         char                    obe_root_ds[MAXPATHLEN];
393         char                    *mp = NULL;

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

399         /* Get name of BE to delete */
400         if (nvlist_lookup_string(be_attrs, BE_ATTR_ORIG_BE_NAME, &bt.obe_name)
401             != 0) {
402                 be_print_err(gettext("be_destroy: failed to lookup "
403                     "BE_ATTR_ORIG_BE_NAME attribute\n"));
404                 return (BE_ERR_INVAL);
405         }

407         /*
408          * Validate BE name. If valid, then check that the original BE is not
409          * the active BE. If it is the 'active' BE then return an error code
410          * since we can't destroy the active BE.
411          */
412         if (!be_valid_be_name(bt.obe_name)) {
413                 be_print_err(gettext("be_destroy: invalid BE name %s\n"),
414                     bt.obe_name);
415                 return (BE_ERR_INVAL);
416         } else if (bt.obe_name != NULL) {
417                 if ((ret = be_find_current_be(&cur_bt)) != BE_SUCCESS) {
418                         return (ret);
419                 }
420                 if (strcmp(cur_bt.obe_name, bt.obe_name) == 0) {
421                         return (BE_ERR_DESTROY_CURR_BE);
422                 }
423         }

425         /* Get destroy flags if provided */
426         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
427             BE_ATTR_DESTROY_FLAGS, DATA_TYPE_UINT16, &flags, NULL)
428             != 0) {
429                 be_print_err(gettext("be_destroy: failed to lookup "
430                     "BE_ATTR_DESTROY_FLAGS attribute\n"));
431                 return (BE_ERR_INVAL);
432         }

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

437         /* Find which zpool obe_name lives in */
438         if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) == 0) {
439                 be_print_err(gettext("be_destroy: failed to find zpool "
440                     "for BE (%s)\n"), bt.obe_name);
441                 return (BE_ERR_BE_NOENT);
442         } else if (zret < 0) {
443                 be_print_err(gettext("be_destroy: zpool_iter failed: %s\n"),
444                     libzfs_error_description(g_zfs));
445                 return (zfs_err_to_be_err(g_zfs));
446         }

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

453         if (getzoneid() != GLOBAL_ZONEID) {
454                 if (!be_zone_compare_uuids(bt.obe_root_ds)) {
455                         if (be_is_active_on_boot(bt.obe_name)) {
456                                 be_print_err(gettext("be_destroy: destroying "
457                                     "active zone root dataset from non-active "
```



```
458                                     "global BE is not supported\n"));
459                                 return (BE_ERR_NOTSUP);
460                         }
461                 }
462         }

464         /*
465          * Detect if the BE to destroy has the 'active on boot' property set.
466          * If so, set the 'active on boot' property on the the 'active' BE.
467          */
468         if (be_is_active_on_boot(bt.obe_name)) {
469                 if ((ret = be_activate_current_be()) != BE_SUCCESS) {
470                         be_print_err(gettext("be_destroy: failed to "
471                             "make the current BE 'active on boot'\n"));
472                         return (ret);
473                 }
474         }

476         /* Get handle to BE's root dataset */
477         if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_FILESYSTEM)) ==
478             NULL) {
479                 be_print_err(gettext("be_destroy: failed to "
480                     "open BE root dataset (%s): %s\n"), bt.obe_root_ds,
481                     libzfs_error_description(g_zfs));
482                 return (zfs_err_to_be_err(g_zfs));
483         }

485         /*
486          * Check if BE has snapshots and BE_DESTROY_FLAG_SNAPSHOTS
487          * is not set.
488          */
489         (void) zfs_iter_snapshots(zhp, be_has_snapshot_callback, &bs_found);
490         if (!dd.destroy_snaps && bs_found) {
491                 ZFS_CLOSE(zhp);
492                 return (BE_ERR_SS_EXISTS);
493         }

495         /* Get the UUID of the global BE */
496         if (getzoneid() == GLOBAL_ZONEID) {
497                 if (be_get_uuid(zfs_get_name(zhp),
498                     &dd.gz_be_uuid) != BE_SUCCESS) {
499                         be_print_err(gettext("be_destroy: BE has no "
500                         "UUID (%s)\n"), zfs_get_name(zhp));
501                 }
502         }

504         /*
505          * If the global BE is mounted, make sure we've been given the
506          * flag to forcibly unmount it.
507          */
508         if (zfs_is_mounted(zhp, &mp)) {
509                 if (!(dd.force_unmount)) {
510                         be_print_err(gettext("be_destroy: "
511                             "%s is currently mounted at %s, cannot destroy\n"),
512                             bt.obe_name, mp != NULL ? mp : "<unknown>");

514                         free(mp);
515                         ZFS_CLOSE(zhp);
516                         return (BE_ERR_MOUNTED);
517                 }
518                 free(mp);
519         }

521         /*
522          * Destroy the non-global zone BE's if we are in the global zone
523          * and there is a UUID associated with the global zone BE
```

```
524          */
525         if (getzoneid() == GLOBAL_ZONEID && !uuid_is_null(dd.gz_be_uuid)) {
526                 if ((ret = be_destroy_zones(bt.obe_name, bt.obe_root_ds, &dd))
527                     != BE_SUCCESS) {
528                         be_print_err(gettext("be_destroy: failed to "
529                             "destroy one or more zones for BE %s\n"),
530                             bt.obe_name);
531                         goto done;
532                 }
533         }
534
535         /* Unmount the BE if it was mounted */
536         if (zfs_is_mounted(zhp, NULL)) {
537                 if ((ret = _be_unmount(bt.obe_name, BE_UNMOUNT_FLAG_FORCE))
538                     != BE_SUCCESS) {
539                         be_print_err(gettext("be_destroy: "
540                             "failed to unmount %s\n"), bt.obe_name);
541                         ZFS_CLOSE(zhp);
542                         return (ret);
543                 }
544         }
545         ZFS_CLOSE(zhp);
546
547         /* Destroy this BE */
548         if ((ret = _be_destroy((const char *)bt.obe_root_ds, &dd))
549             != BE_SUCCESS) {
550                 goto done;
551         }
552
553         /* Remove BE's entry from the boot menu */
554         if (getzoneid() == GLOBAL_ZONEID) {
555                 if ((ret = be_remove_menu(bt.obe_name, bt.obe_zpool, NULL))
556                     != BE_SUCCESS) {
557                         be_print_err(gettext("be_destroy: failed to "
558                             "remove BE %s from the boot menu\n"),
559                             bt.obe_root_ds);
560                         goto done;
561                 }
562         }
563
564 done:
565         be_zfs_fini();
566
567         return (ret);
568 }
569
570 /*
571  * Function:    be_copy
572  * Description: This function makes a copy of an existing BE.  If the original
573  *              BE and the new BE are in the same pool, it uses zfs cloning to
574  *              create the new BE, otherwise it does a physical copy.
575  *              If the original BE name isn't provided, it uses the currently
576  *              booted BE.  If the new BE name isn't provided, it creates an
577  *              auto named BE and returns that name to the caller.
578  * Parameters:
579  *              be_attrs - pointer to nvlist_t of attributes being passed in.
580  *                      The following attributes are used by this function:
581  *
582  *                      BE_ATTR_ORIG_BE_NAME            *optional
583  *                      BE_ATTR_SNAP_NAME               *optional
584  *                      BE_ATTR_NEW_BE_NAME             *optional
585  *                      BE_ATTR_NEW_BE_POOL             *optional
586  *                      BE_ATTR_NEW_BE_DESC             *optional
587  *                      BE_ATTR_ZFS_PROPERTIES          *optional
588  *                      BE_ATTR_POLICY                  *optional
589  *
```



```
590  *                      If the BE_ATTR_NEW_BE_NAME was not passed in, upon
591  *                      successful BE creation, the following attribute values
592  *                      will be returned to the caller by setting them in the
593  *                      be_attrs parameter passed in:
594  *
595  *                      BE_ATTR_SNAP_NAME
596  *                      BE_ATTR_NEW_BE_NAME
597  * Return:
598  *              BE_SUCCESS - Success
599  *              be_errno_t - Failure
600  * Scope:
601  *              Public
602  */
603 int
604 be_copy(nvlist_t *be_attrs)
605 {
606         be_transaction_data_t   bt = { 0 };
607         be_fs_list_data_t       fld = { 0 };
608         zfs_handle_t    *zhp = NULL;
609         zpool_handle_t  *zphp = NULL;
610         nvlist_t        *zfs_props = NULL;
611         uuid_t          uu = { 0 };
612         uuid_t          parent_uu = { 0 };
613         char            obe_root_ds[MAXPATHLEN];
614         char            nbe_root_ds[MAXPATHLEN];
615         char            ss[MAXPATHLEN];
616         char            *new_mp = NULL;
617         char            *obe_name = NULL;
618         boolean_t       autoname = B_FALSE;
619         boolean_t       be_created = B_FALSE;
620         int             i;
621         int             zret;
622         int             ret = BE_SUCCESS;
623         struct be_defaults be_defaults;
624
625         /* Initialize libzfs handle */
626         if (!be_zfs_init())
627                 return (BE_ERR_INIT);
628
629         /* Get original BE name */
630         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
631             BE_ATTR_ORIG_BE_NAME, DATA_TYPE_STRING, &obe_name, NULL) != 0) {
632                 be_print_err(gettext("be_copy: failed to lookup "
633                     "BE_ATTR_ORIG_BE_NAME attribute\n"));
634                 return (BE_ERR_INVAL);
635         }
636
637         if ((ret = be_find_current_be(&bt)) != BE_SUCCESS) {
638                 return (ret);
639         }
640
641         be_get_defaults(&be_defaults);
642
643         /* If original BE name not provided, use current BE */
644         if (obe_name != NULL) {
645                 bt.obe_name = obe_name;
646                 /* Validate original BE name */
647                 if (!be_valid_be_name(bt.obe_name)) {
648                         be_print_err(gettext("be_copy: "
649                             "invalid BE name %s\n"), bt.obe_name);
650                         return (BE_ERR_INVAL);
651                 }
652         }
653
654         if (be_defaults.be_deflt_rpool_container) {
655                 if ((zphp = zpool_open(g_zfs, bt.obe_zpool)) == NULL) {
```

```
656                         be_print_err(gettext("be_get_node_data: failed to "
657                             "open rpool (%s): %s\n"), bt.obe_zpool,
658                             libzfs_error_description(g_zfs));
659                         return (zfs_err_to_be_err(g_zfs));
660                 }
661                 if (be_find_zpool_callback(zphp, &bt) == 0) {
662                         return (BE_ERR_BE_NOENT);
663                 }
664         } else {
665                 /* Find which zpool obe_name lives in */
666                 if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) ==
667                     0) {
668                         be_print_err(gettext("be_copy: failed to "
669                             "find zpool for BE (%s)\n"), bt.obe_name);
670                         return (BE_ERR_BE_NOENT);
671                 } else if (zret < 0) {
672                         be_print_err(gettext("be_copy: "
673                             "zpool_iter failed: %s\n"),
674                             libzfs_error_description(g_zfs));
675                         return (zfs_err_to_be_err(g_zfs));
676                 }
677         }

679         /* Get snapshot name of original BE if one was provided */
680         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
681             BE_ATTR_SNAP_NAME, DATA_TYPE_STRING, &bt.obe_snap_name, NULL)
682             != 0) {
683                 be_print_err(gettext("be_copy: failed to lookup "
684                     "BE_ATTR_SNAP_NAME attribute\n"));
685                 return (BE_ERR_INVAL);
686         }

688         /* Get new BE name */
689         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
690             BE_ATTR_NEW_BE_NAME, DATA_TYPE_STRING, &bt.nbe_name, NULL)
691             != 0) {
692                 be_print_err(gettext("be_copy: failed to lookup "
693                     "BE_ATTR_NEW_BE_NAME attribute\n"));
694                 return (BE_ERR_INVAL);
695         }

697         /* Get zpool name to create new BE in */
698         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
699             BE_ATTR_NEW_BE_POOL, DATA_TYPE_STRING, &bt.nbe_zpool, NULL) != 0) {
700                 be_print_err(gettext("be_copy: failed to lookup "
701                     "BE_ATTR_NEW_BE_POOL attribute\n"));
702                 return (BE_ERR_INVAL);
703         }

705         /* Get new BE's description if one was provided */
706         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
707             BE_ATTR_NEW_BE_DESC, DATA_TYPE_STRING, &bt.nbe_desc, NULL) != 0) {
708                 be_print_err(gettext("be_copy: failed to lookup "
709                     "BE_ATTR_NEW_BE_DESC attribute\n"));
710                 return (BE_ERR_INVAL);
711         }

713         /* Get BE policy to create this snapshot under */
714         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
715             BE_ATTR_POLICY, DATA_TYPE_STRING, &bt.policy, NULL) != 0) {
716                 be_print_err(gettext("be_copy: failed to lookup "
717                     "BE_ATTR_POLICY attribute\n"));
718                 return (BE_ERR_INVAL);
719         }

721         /*
```



```
722          * Create property list for new BE root dataset.  If some
723          * zfs properties were already provided by the caller, dup
724          * that list.  Otherwise initialize a new property list.
725          */
726         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
727             BE_ATTR_ZFS_PROPERTIES, DATA_TYPE_NVLIST, &zfs_props, NULL)
728             != 0) {
729                 be_print_err(gettext("be_copy: failed to lookup "
730                     "BE_ATTR_ZFS_PROPERTIES attribute\n"));
731                 return (BE_ERR_INVAL);
732         }
733         if (zfs_props != NULL) {
734                 /* Make sure its a unique nvlist */
735                 if (!(zfs_props->nvl_nvflag & NV_UNIQUE_NAME) &&
736                     !(zfs_props->nvl_nvflag & NV_UNIQUE_NAME_TYPE)) {
737                         be_print_err(gettext("be_copy: ZFS property list "
738                             "not unique\n"));
739                         return (BE_ERR_INVAL);
740                 }

742                 /* Dup the list */
743                 if (nvlist_dup(zfs_props, &bt.nbe_zfs_props, 0) != 0) {
744                         be_print_err(gettext("be_copy: "
745                             "failed to dup ZFS property list\n"));
746                         return (BE_ERR_NOMEM);
747                 }
748         } else {
749                 /* Initialize new nvlist */
750                 if (nvlist_alloc(&bt.nbe_zfs_props, NV_UNIQUE_NAME, 0) != 0) {
751                         be_print_err(gettext("be_copy: internal "
752                             "error: out of memory\n"));
753                         return (BE_ERR_NOMEM);
754                 }
755         }

757         /*
758          * If new BE name provided, validate the BE name and then verify
759          * that new BE name doesn't already exist in some pool.
760          */
761         if (bt.nbe_name) {
762                 /* Validate original BE name */
763                 if (!be_valid_be_name(bt.nbe_name)) {
764                         be_print_err(gettext("be_copy: "
765                             "invalid BE name %s\n"), bt.nbe_name);
766                         ret = BE_ERR_INVAL;
767                         goto done;
768                 }

770                 /* Verify it doesn't already exist */
771                 if (getzoneid() == GLOBAL_ZONEID) {
772                         if ((zret = zpool_iter(g_zfs, be_exists_callback,
773                             bt.nbe_name)) > 0) {
774                                 be_print_err(gettext("be_copy: BE (%s) already "
775                                     "exists\n"), bt.nbe_name);
776                                 ret = BE_ERR_BE_EXISTS;
777                                 goto done;
778                         } else if (zret < 0) {
779                                 be_print_err(gettext("be_copy: zpool_iter "
780                                     "failed: %s\n"),
781                                     libzfs_error_description(g_zfs));
782                                 ret = zfs_err_to_be_err(g_zfs);
783                                 goto done;
784                         }
785                 } else {
786                         be_make_root_ds(bt.nbe_zpool, bt.nbe_name, nbe_root_ds,
787                             sizeof (nbe_root_ds));
```

```
788                         if (zfs_dataset_exists(g_zfs, nbe_root_ds,
789                             ZFS_TYPE_FILESYSTEM)) {
790                                 be_print_err(gettext("be_copy: BE (%s) already "
791                                     "exists\n"), bt.nbe_name);
792                                 ret = BE_ERR_BE_EXISTS;
793                                 goto done;
794                         }
795                 }
796         } else {
797                 /*
798                  * If an auto named BE is desired, it must be in the same
799                  * pool is the original BE.
800                  */
801                 if (bt.nbe_zpool != NULL) {
802                         be_print_err(gettext("be_copy: cannot specify pool "
803                             "name when creating an auto named BE\n"));
804                         ret = BE_ERR_INVAL;
805                         goto done;
806                 }

808                 /*
809                  * Generate auto named BE
810                  */
811                 if ((bt.nbe_name = be_auto_be_name(bt.obe_name))
812                     == NULL) {
813                         be_print_err(gettext("be_copy: "
814                             "failed to generate auto BE name\n"));
815                         ret = BE_ERR_AUTONAME;
816                         goto done;
817                 }

819                 autoname = B_TRUE;
820         }

822         /*
823          * If zpool name to create new BE in is not provided,
824          * create new BE in original BE's pool.
825          */
826         if (bt.nbe_zpool == NULL) {
827                 bt.nbe_zpool = bt.obe_zpool;
828         }

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

836         bt.obe_root_ds = obe_root_ds;
837         bt.nbe_root_ds = nbe_root_ds;

839         /*
840          * If an existing snapshot name has been provided to create from,
841          * verify that it exists for the original BE's root dataset.
842          */
843         if (bt.obe_snap_name != NULL) {

845                 /* Generate dataset name for snapshot to use. */
846                 (void) snprintf(ss, sizeof (ss), "%s@%s", bt.obe_root_ds,
847                     bt.obe_snap_name);

849                 /* Verify snapshot exists */
850                 if (!zfs_dataset_exists(g_zfs, ss, ZFS_TYPE_SNAPSHOT)) {
851                         be_print_err(gettext("be_copy: "
852                             "snapshot does not exist (%s): %s\n"), ss,
853                             libzfs_error_description(g_zfs));
```



```
854                         ret = BE_ERR_SS_NOENT;
855                         goto done;
856                 }
857         } else {
858                 /*
859                  * Else snapshot name was not provided, generate an
860                  * auto named snapshot to use as its origin.
861                  */
862                 if ((ret = _be_create_snapshot(bt.obe_name,
863                     &bt.obe_snap_name, bt.policy)) != BE_SUCCESS) {
864                         be_print_err(gettext("be_copy: "
865                             "failed to create auto named snapshot\n"));
866                         goto done;
867                 }

869                 if (nvlist_add_string(be_attrs, BE_ATTR_SNAP_NAME,
870                     bt.obe_snap_name) != 0) {
871                         be_print_err(gettext("be_copy: "
872                             "failed to add snap name to be_attrs\n"));
873                         ret = BE_ERR_NOMEM;
874                         goto done;
875                 }
876         }

878         /* Get handle to original BE's root dataset. */
879         if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_FILESYSTEM))
880             == NULL) {
881                 be_print_err(gettext("be_copy: failed to "
882                     "open BE root dataset (%s): %s\n"), bt.obe_root_ds,
883                     libzfs_error_description(g_zfs));
884                 ret = zfs_err_to_be_err(g_zfs);
885                 goto done;
886         }

888         /* If original BE is currently mounted, record its altroot. */
889         if (zfs_is_mounted(zhp, &bt.obe_altroot) && bt.obe_altroot == NULL) {
890                 be_print_err(gettext("be_copy: failed to "
891                     "get altroot of mounted BE %s: %s\n"),
892                     bt.obe_name, libzfs_error_description(g_zfs));
893                 ret = zfs_err_to_be_err(g_zfs);
894                 goto done;
895         }

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

899                 /* Do clone */

901                 /*
902                  * Iterate through original BE's datasets and clone
903                  * them to create new BE.  This call will end up closing
904                  * the zfs handle passed in whether it succeeds for fails.
905                  */
906                 if ((ret = be_clone_fs_callback(zhp, &bt)) != 0) {
907                         zhp = NULL;
908                         /* Creating clone BE failed */
909                         if (!autoname || ret != BE_ERR_BE_EXISTS) {
910                                 be_print_err(gettext("be_copy: "
911                                     "failed to clone new BE (%s) from "
912                                     "orig BE (%s)\n"),
913                                     bt.nbe_name, bt.obe_name);
914                                 ret = BE_ERR_CLONE;
915                                 goto done;
916                         }

918                         /*
919                          * We failed to create the new BE because a BE with
```

```
920                          * the auto-name we generated above has since come
921                          * into existence.  Regenerate a new auto-name
922                          * and retry.
923                          */
924                         for (i = 1; i < BE_AUTO_NAME_MAX_TRY; i++) {
925
926                                 /* Sleep 1 before retrying */
927                                 (void) sleep(1);
928
929                                 /* Generate new auto BE name */
930                                 free(bt.nbe_name);
931                                 if ((bt.nbe_name = be_auto_be_name(bt.obe_name))
932                                     == NULL) {
933                                         be_print_err(gettext("be_copy: "
934                                             "failed to generate auto "
935                                             "BE name\n"));
936                                         ret = BE_ERR_AUTONAME;
937                                         goto done;
938                                 }
939
940                                 /*
941                                  * Regenerate string for new BE's
942                                  * root dataset name
943                                  */
944                                 be_make_root_ds(bt.nbe_zpool, bt.nbe_name,
945                                     nbe_root_ds, sizeof (nbe_root_ds));
946                                 bt.nbe_root_ds = nbe_root_ds;
947
948                                 /*
949                                  * Get handle to original BE's root dataset.
950                                  */
951                                 if ((zhp = zfs_open(g_zfs, bt.obe_root_ds,
952                                     ZFS_TYPE_FILESYSTEM)) == NULL) {
953                                         be_print_err(gettext("be_copy: "
954                                             "failed to open BE root dataset "
955                                             "(%s): %s\n"), bt.obe_root_ds,
956                                             libzfs_error_description(g_zfs));
957                                         ret = zfs_err_to_be_err(g_zfs);
958                                         goto done;
959                                 }
960
961                                 /*
962                                  * Try to clone the BE again.  This
963                                  * call will end up closing the zfs
964                                  * handle passed in whether it
965                                  * succeeds or fails.
966                                  */
967                                 ret = be_clone_fs_callback(zhp, &bt);
968                                 zhp = NULL;
969                                 if (ret == 0) {
970                                         break;
971                                 } else if (ret != BE_ERR_BE_EXISTS) {
972                                         be_print_err(gettext("be_copy: "
973                                             "failed to clone new BE "
974                                             "(%s) from orig BE (%s)\n"),
975                                             bt.nbe_name, bt.obe_name);
976                                         ret = BE_ERR_CLONE;
977                                         goto done;
978                                 }
979                         }
980
981                         /*
982                          * If we've exhausted the maximum number of
983                          * tries, free the auto BE name and return
984                          * error.
985                          */
```



```
986                                 if (i == BE_AUTO_NAME_MAX_TRY) {
987                                         be_print_err(gettext("be_copy: failed "
988                                             "to create unique auto BE name\n"));
989                                         free(bt.nbe_name);
990                                         bt.nbe_name = NULL;
991                                         ret = BE_ERR_AUTONAME;
992                                         goto done;
993                                 }
994                         }
995                         zhp = NULL;
996
997                 } else {
998
999                         /* Do copy (i.e. send BE datasets via zfs_send/recv) */
1000
1001                        /*
1002                         * Verify BE container dataset in nbe_zpool exists.
1003                         * If not, create it.
1004                         */
1005                        if (!be_create_container_ds(bt.nbe_zpool)) {
1006                                ret = BE_ERR_CREATDS;
1007                                goto done;
1008                        }
1009
1010                        /*
1011                         * Iterate through original BE's datasets and send
1012                         * them to the other pool.  This call will end up closing
1013                         * the zfs handle passed in whether it succeeds or fails.
1014                         */
1015                        if ((ret = be_send_fs_callback(zhp, &bt)) != 0) {
1016                                be_print_err(gettext("be_copy: failed to "
1017                                    "send BE (%s) to pool (%s)\n"), bt.obe_name,
1018                                    bt.nbe_zpool);
1019                                ret = BE_ERR_COPY;
1020                                zhp = NULL;
1021                                goto done;
1022                        }
1023                        zhp = NULL;
1024                }
1025
1026                /*
1027                 * Set flag to note that the dataset(s) for the new BE have been
1028                 * successfully created so that if a failure happens from this point
1029                 * on, we know to cleanup these datasets.
1030                 */
1031                be_created = B_TRUE;
1032
1033                /*
1034                 * Validate that the new BE is mountable.
1035                 * Do not attempt to mount non-global zone datasets
1036                 * since they are not cloned yet.
1037                 */
1038                if ((ret = _be_mount(bt.nbe_name, &new_mp, BE_MOUNT_FLAG_NO_ZONES))
1039                    != BE_SUCCESS) {
1040                        be_print_err(gettext("be_copy: failed to "
1041                            "mount newly created BE\n"));
1042                        (void) _be_unmount(bt.nbe_name, 0);
1043                        goto done;
1044                }
1045
1046                /* Set UUID for new BE */
1047                if (getzoneid() == GLOBAL_ZONEID) {
1048                        if (be_set_uuid(bt.nbe_root_ds) != BE_SUCCESS) {
1049                                be_print_err(gettext("be_copy: failed to "
1050                                    "set uuid for new BE\n"));
1051                        }
```

```
1052         } else {
1053                 if ((ret = be_zone_get_parent_uuid(bt.obe_root_ds,
1054                     &parent_uu)) != BE_SUCCESS) {
1055                         be_print_err(gettext("be_copy: failed to get "
1056                             "parentbe uuid from orig BE\n"));
1057                         ret = BE_ERR_ZONE_NO_PARENTBE;
1058                         goto done;
1059                 } else if ((ret = be_zone_set_parent_uuid(bt.nbe_root_ds,
1060                     parent_uu)) != BE_SUCCESS) {
1061                         be_print_err(gettext("be_copy: failed to set "
1062                             "parentbe uuid for newly created BE\n"));
1063                         goto done;
1064                 }
1065         }

1067         /*
1068          * Process zones outside of the private BE namespace.
1069          * This has to be done here because we need the uuid set in the
1070          * root dataset of the new BE. The uuid is use to set the parentbe
1071          * property for the new zones datasets.
1072          */
1073         if (getzoneid() == GLOBAL_ZONEID &&
1074             be_get_uuid(bt.obe_root_ds, &uu) == BE_SUCCESS) {
1075                 if ((ret = be_copy_zones(bt.obe_name, bt.obe_root_ds,
1076                     bt.nbe_root_ds)) != BE_SUCCESS) {
1077                         be_print_err(gettext("be_copy: failed to process "
1078                             "zones\n"));
1079                         goto done;
1080                 }
1081         }

1083         /*
1084          * Generate a list of file systems from the original BE that are
1085          * legacy mounted.  We use this list to determine which entries in
1086          * vfstab we need to update for the new BE we've just created.
1087          */
1088         if ((ret = be_get_legacy_fs(bt.obe_name, bt.obe_root_ds, NULL, NULL,
1089             &fld)) != BE_SUCCESS) {
1090                 be_print_err(gettext("be_copy: failed to "
1091                     "get legacy mounted file system list for %s\n"),
1092                     bt.obe_name);
1093                 goto done;
1094         }

1096         /*
1097          * Update new BE's vfstab.
1098          */
1099         if ((ret = be_update_vfstab(bt.nbe_name, bt.obe_zpool, bt.nbe_zpool,
1100             &fld, new_mp)) != BE_SUCCESS) {
1101                 be_print_err(gettext("be_copy: failed to "
1102                     "update new BE's vfstab (%s)\n"), bt.nbe_name);
1103                 goto done;
1104         }

1106         /* Unmount the new BE */
1107         if ((ret = _be_unmount(bt.nbe_name, 0)) != BE_SUCCESS) {
1108                 be_print_err(gettext("be_copy: failed to "
1109                     "unmount newly created BE\n"));
1110                 goto done;
1111         }

1113         /*
1114          * Add boot menu entry for newly created clone
1115          */
1116         if (getzoneid() == GLOBAL_ZONEID &&
1117             (ret = be_append_menu(bt.nbe_name, bt.nbe_zpool,
```



```
1118                 NULL, bt.obe_root_ds, bt.nbe_desc)) != BE_SUCCESS) {
1119                     be_print_err(gettext("be_copy: failed to "
1120                         "add BE (%s) to boot menu\n"), bt.nbe_name);
1121                     goto done;
1122         }

1124         /*
1125          * If we succeeded in creating an auto named BE, set its policy
1126          * type and return the auto generated name to the caller by storing
1127          * it in the nvlist passed in by the caller.
1128          */
1129         if (autoname) {
1130                 /* Get handle to new BE's root dataset. */
1131                 if ((zhp = zfs_open(g_zfs, bt.nbe_root_ds,
1132                     ZFS_TYPE_FILESYSTEM)) == NULL) {
1133                         be_print_err(gettext("be_copy: failed to "
1134                             "open BE root dataset (%s): %s\n"), bt.nbe_root_ds,
1135                             libzfs_error_description(g_zfs));
1136                         ret = zfs_err_to_be_err(g_zfs);
1137                         goto done;
1138                 }

1140                 /*
1141                  * Set the policy type property into the new BE's root dataset
1142                  */
1143                 if (bt.policy == NULL) {
1144                         /* If no policy type provided, use default type */
1145                         bt.policy = be_default_policy();
1146                 }

1148                 if (zfs_prop_set(zhp, BE_POLICY_PROPERTY, bt.policy) != 0) {
1149                         be_print_err(gettext("be_copy: failed to "
1150                             "set BE policy for %s: %s\n"), bt.nbe_name,
1151                             libzfs_error_description(g_zfs));
1152                         ret = zfs_err_to_be_err(g_zfs);
1153                         goto done;
1154                 }

1156                 /*
1157                  * Return the auto generated name to the caller
1158                  */
1159                 if (bt.nbe_name) {
1160                         if (nvlist_add_string(be_attrs, BE_ATTR_NEW_BE_NAME,
1161                             bt.nbe_name) != 0) {
1162                                 be_print_err(gettext("be_copy: failed to "
1163                                     "add snap name to be_attrs\n"));
1164                         }
1165                 }
1166         }

1168 done:
1169         ZFS_CLOSE(zhp);
1170         be_free_fs_list(&fld);

1172         if (bt.nbe_zfs_props != NULL)
1173                 nvlist_free(bt.nbe_zfs_props);

1175         free(bt.obe_altroot);
1176         free(new_mp);

1178         /*
1179          * If a failure occurred and we already created the datasets for
1180          * the new boot environment, destroy them.
1181          */
1182         if (ret != BE_SUCCESS && be_created) {
1183                 be_destroy_data_t       cdd = { 0 };
```

```
1185                 cdd.force_unmount = B_TRUE;

1187                 be_print_err(gettext("be_copy: "
1188                     "destroying partially created boot environment\n"));

1190                 if (getzoneid() == GLOBAL_ZONEID && be_get_uuid(bt.nbe_root_ds,
1191                     &cdd.gz_be_uuid) == 0)
1192                         (void) be_destroy_zones(bt.nbe_name, bt.nbe_root_ds,
1193                             &cdd);

1195                 (void) _be_destroy(bt.nbe_root_ds, &cdd);
1196         }

1198         be_zfs_fini();

1200         return (ret);
1201 }

1203 /* ************************************************************** */
1204 /*                      Semi-Private Functions                    */
1205 /* ************************************************************** */

1207 /*
1208  * Function:    be_find_zpool_callback
1209  * Description: Callback function used to find the pool that a BE lives in.
1210  * Parameters:
1211  *              zlp - zpool_handle_t pointer for the current pool being
1212  *                      looked at.
1213  *              data - be_transaction_data_t pointer providing information
1214  *                      about the BE that's being searched for.
1215  *                      This function uses the obe_name member of this
1216  *                      parameter to use as the BE name to search for.
1217  *                      Upon successfully locating the BE, it populates
1218  *                      obe_zpool with the pool name that the BE is found in.
1219  * Returns:
1220  *              1 - BE exists in this pool.
1221  *              0 - BE does not exist in this pool.
1222  * Scope:
1223  *              Semi-private (library wide use only)
1224  */
1225 int
1226 be_find_zpool_callback(zpool_handle_t *zlp, void *data)
1227 {
1228         be_transaction_data_t   *bt = data;
1229         const char              *zpool =  zpool_get_name(zlp);
1230         char                    be_root_ds[MAXPATHLEN];

1232         /*
1233          * Generate string for the BE's root dataset
1234          */
1235         be_make_root_ds(zpool, bt->obe_name, be_root_ds, sizeof (be_root_ds));

1237         /*
1238          * Check if dataset exists
1239          */
1240         if (zfs_dataset_exists(g_zfs, be_root_ds, ZFS_TYPE_FILESYSTEM)) {
1241                 /* BE's root dataset exists in zpool */
1242                 bt->obe_zpool = strdup(zpool);
1243                 zpool_close(zlp);
1244                 return (1);
1245         }

1247         zpool_close(zlp);
1248         return (0);
1249 }
```



```
1251 /*
1252  * Function:    be_exists_callback
1253  * Description: Callback function used to find out if a BE exists.
1254  * Parameters:
1255  *              zlp - zpool_handle_t pointer to the current pool being
1256  *                      looked at.
1257  *              data - BE name to look for.
1258  * Return:
1259  *              1 - BE exists in this pool.
1260  *              0 - BE does not exist in this pool.
1261  * Scope:
1262  *              Semi-private (library wide use only)
1263  */
1264 int
1265 be_exists_callback(zpool_handle_t *zlp, void *data)
1266 {
1267         const char      *zpool = zpool_get_name(zlp);
1268         char            *be_name = data;
1269         char            be_root_ds[MAXPATHLEN];

1271         /*
1272          * Generate string for the BE's root dataset
1273          */
1274         be_make_root_ds(zpool, be_name, be_root_ds, sizeof (be_root_ds));

1276         /*
1277          * Check if dataset exists
1278          */
1279         if (zfs_dataset_exists(g_zfs, be_root_ds, ZFS_TYPE_FILESYSTEM)) {
1280                 /* BE's root dataset exists in zpool */
1281                 zpool_close(zlp);
1282                 return (1);
1283         }

1285         zpool_close(zlp);
1286         return (0);
1287 }

1289 /*
1290  * Function:    be_has_snapshots_callback
1291  * Description: Callback function used to find out if a BE has snapshots.
1292  * Parameters:
1293  *              zlp - zpool_handle_t pointer to the current pool being
1294  *                      looked at.
1295  *              data - be_snap_found_t pointer.
1296  * Return:
1297  *              1 - BE has no snapshots.
1298  *              0 - BE has snapshots.
1299  * Scope:
1300  *              Private
1301  */
1302 static int
1303 be_has_snapshot_callback(zfs_handle_t *zhp, void *data)
1304 {
1305         boolean_t *bs = data;
1306         if (zfs_get_name(zhp) == NULL) {
1307                 zfs_close(zhp);
1308                 return (1);
1309         }
1310         *bs = B_TRUE;
1311         zfs_close(zhp);
1312         return (0);
1313 }

1315 /*
```

```
 * Function:    be_set_uuid
 * Description: This function generates a uuid, unparses it into
 *              string representation, and sets that string into
 *              a zfs user property for a root dataset of a BE.
 *              The name of the user property used to store the
 *              uuid is org.opensolaris.libbe:uuid
 *
 * Parameters:
 *              root_ds - Root dataset of the BE to set a uuid on.
 * Return:
 *              be_errno_t - Failure
 *              BE_SUCCESS - Success
 * Scope:
 *              Semi-private (library wide ues only)
 */
int
be_set_uuid(char *root_ds)
{
        zfs_handle_t    *zhp = NULL;
        uuid_t          uu = { 0 };
        char            uu_string[UUID_PRINTABLE_STRING_LENGTH] = { 0 };
        int             ret = BE_SUCCESS;

        /* Generate a UUID and unparse it into string form */
        uuid_generate(uu);
        if (uuid_is_null(uu) != 0) {
                be_print_err(gettext("be_set_uuid: failed to "
                    "generate uuid\n"));
                return (BE_ERR_GEN_UUID);
        }
        uuid_unparse(uu, uu_string);

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

        /* Set uuid property for the BE */
        if (zfs_prop_set(zhp, BE_UUID_PROPERTY, uu_string) != 0) {
                be_print_err(gettext("be_set_uuid: failed to "
                    "set uuid property for BE: %s\n"),
                    libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
        }

        ZFS_CLOSE(zhp);

        return (ret);
}

/*
 * Function:    be_get_uuid
 * Description: This function gets the uuid string from a BE root
 *              dataset, parses it into internal format, and returns
 *              it the caller via a reference pointer passed in.
 *
 * Parameters:
 *              rootds - Root dataset of the BE to get the uuid from.
 *              uu - reference pointer to a uuid_t to return uuid in.
 * Return:
 *              be_errno_t - Failure
 *              BE_SUCCESS - Success
 * Scope:
```



```
 *              Semi-private (library wide use only)
 */
int
be_get_uuid(const char *root_ds, uuid_t *uu)
{
        zfs_handle_t    *zhp = NULL;
        nvlist_t        *userprops = NULL;
        nvlist_t        *propname = NULL;
        char            *uu_string = NULL;
        int             ret = BE_SUCCESS;

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

        /* Get user properties for BE's root dataset */
        if ((userprops = zfs_get_user_props(zhp)) == NULL) {
                be_print_err(gettext("be_get_uuid: failed to "
                    "get user properties for BE root dataset (%s): %s\n"),
                    root_ds, libzfs_error_description(g_zfs));
                ret = zfs_err_to_be_err(g_zfs);
                goto done;
        }

        /* Get UUID string from BE's root dataset user properties */
        if (nvlist_lookup_nvlist(userprops, BE_UUID_PROPERTY, &propname) != 0 ||
            nvlist_lookup_string(propname, ZPROP_VALUE, &uu_string) != 0) {
                /*
                 * This probably just means that the BE is simply too old
                 * to have a uuid or that we haven't created a uuid for
                 * this BE yet.
                 */
                be_print_err(gettext("be_get_uuid: failed to "
                    "get uuid property from BE root dataset user "
                    "properties.\n"));
                ret = BE_ERR_NO_UUID;
                goto done;
        }
        /* Parse uuid string into internal format */
        if (uuid_parse(uu_string, *uu) != 0 || uuid_is_null(*uu)) {
                be_print_err(gettext("be_get_uuid: failed to "
                    "parse uuid\n"));
                ret = BE_ERR_PARSE_UUID;
                goto done;
        }

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

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

/*
 * Function:    _be_destroy
 * Description: Destroy a BE and all of its children datasets and snapshots.
 *              This function is called for both global BEs and non-global BEs.
 *              The root dataset of either the global BE or non-global BE to be
 *              destroyed is passed in.
 * Parameters:
```

```
 *              root_ds - pointer to the name of the root dataset of the
 *                      BE to destroy.
 *              dd - pointer to a be_destroy_data_t structure.
 *
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Private
 */
static int
_be_destroy(const char *root_ds, be_destroy_data_t *dd)
{
        zfs_handle_t    *zhp = NULL;
        char            origin[MAXPATHLEN];
        char            parent[MAXPATHLEN];
        char            *snap = NULL;
        boolean_t       has_origin = B_FALSE;
        int             ret = BE_SUCCESS;

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

        /*
         * Demote this BE in case it has dependent clones.  This call
         * will end up closing the zfs handle passed in whether it
         * succeeds or fails.
         */
        if (be_demote_callback(zhp, NULL) != 0) {
                be_print_err(gettext("be_destroy: "
                    "failed to demote BE %s\n"), root_ds);
                return (BE_ERR_DEMOTE);
        }

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

        /*
         * Get the origin of this BE's root dataset.  This will be used
         * later to destroy the snapshots originally used to create this BE.
         */
        if (zfs_prop_get(zhp, ZFS_PROP_ORIGIN, origin, sizeof (origin), NULL,
            NULL, 0, B_FALSE) == 0) {
                (void) strlcpy(parent, origin, sizeof (parent));
                if (be_get_snap(parent, &snap) != BE_SUCCESS) {
                        ZFS_CLOSE(zhp);
                        be_print_err(gettext("be_destroy: failed to "
                            "get snapshot name from origin %s\n"), origin);
                        return (BE_ERR_INVAL);
                }
                has_origin = B_TRUE;
        }

        /*
```



```
         * Destroy the BE's root and its hierarchical children.  This call
         * will end up closing the zfs handle passed in whether it succeeds
         * or fails.
         */
        if (be_destroy_callback(zhp, dd) != 0) {
                be_print_err(gettext("be_destroy: failed to "
                    "destroy BE %s\n"), root_ds);
                ret = zfs_err_to_be_err(g_zfs);
                return (ret);
        }

        /* If BE has an origin */
        if (has_origin) {

                /*
                 * If origin snapshot doesn't have any other
                 * dependents, delete the origin.
                 */
                if ((zhp = zfs_open(g_zfs, origin, ZFS_TYPE_SNAPSHOT)) ==
                    NULL) {
                        be_print_err(gettext("be_destroy: failed to "
                            "open BE's origin (%s): %s\n"), origin,
                            libzfs_error_description(g_zfs));
                        ret = zfs_err_to_be_err(g_zfs);
                        return (ret);
                }

                /* If origin has dependents, don't delete it. */
                if (zfs_prop_get_int(zhp, ZFS_PROP_NUMCLONES) != 0) {
                        ZFS_CLOSE(zhp);
                        return (ret);
                }
                ZFS_CLOSE(zhp);

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

                /* Destroy the snapshot origin used to create this BE. */
                /*
                 * The boolean set to B_FALSE and passed to zfs_destroy_snaps()
                 * tells zfs to process and destroy the snapshots now.
                 * Otherwise the call will potentially return where the
                 * snapshot isn't actually destroyed yet, and ZFS is waiting
                 * until all the references to the snapshot have been
                 * released before actually destroying the snapshot.
                 */
                if (zfs_destroy_snaps(zhp, snap, B_FALSE) != 0) {
                        be_print_err(gettext("be_destroy: failed to "
                            "destroy original snapshots used to create "
                            "BE: %s\n"), libzfs_error_description(g_zfs));

                        /*
                         * If a failure happened because a clone exists,
                         * don't return a failure to the user.  Above, we're
                         * only checking that the root dataset's origin
                         * snapshot doesn't have dependent clones, but its
                         * possible that a subordinate dataset origin snapshot
                         * has a clone.  We really need to check for that
                         * before trying to destroy the origin snapshot.
```

```
1580                         */
1581                        if (libzfs_errno(g_zfs) != EZFS_EXISTS) {
1582                                ret = zfs_err_to_be_err(g_zfs);
1583                                ZFS_CLOSE(zhp);
1584                                return (ret);
1585                        }
1586                }
1587                ZFS_CLOSE(zhp);
1588        }

1590        return (ret);
1591 }

1593 /*
1594  * Function:    be_destroy_zones
1595  * Description: Find valid zone's and call be_destroy_zone_roots to destroy its
1596  *              corresponding dataset and all of its children datasets
1597  *              and snapshots.
1598  * Parameters:
1599  *              be_name - name of global boot environment being destroyed
1600  *              be_root_ds - root dataset of global boot environment being
1601  *                      destroyed.
1602  *              dd - be_destroy_data_t pointer
1603  * Return:
1604  *              BE_SUCCESS - Success
1605  *              be_errno_t - Failure
1606  * Scope:
1607  *              Private
1608  *
1609  * NOTES - Requires that the BE being deleted has no dependent BEs.  If it
1610  *         does, the destroy will fail.
1611  */
1612 static int
1613 be_destroy_zones(char *be_name, char *be_root_ds, be_destroy_data_t *dd)
1614 {
1615        int             i;
1616        int             ret = BE_SUCCESS;
1617        int             force_umnt = BE_UNMOUNT_FLAG_NULL;
1618        char            *zonepath = NULL;
1619        char            *zonename = NULL;
1620        char            *zonepath_ds = NULL;
1621        char            *mp = NULL;
1622        zoneList_t      zlist = NULL;
  26        zoneBrandList_t *brands = NULL;
1623        zfs_handle_t    *zhp = NULL;

1625        /* If zones are not implemented, then get out. */
1626        if (!z_zones_are_implemented()) {
1627                return (BE_SUCCESS);
1628        }

  34        /* Get list of supported brands */
  35        if ((brands = be_get_supported_brandlist()) == NULL) {
  36                be_print_err(gettext("be_destroy_zones: "
  37                    "no supported brands\n"));
  38                return (BE_SUCCESS);
  39        }

1630        /* Get handle to BE's root dataset */
1631        if ((zhp = zfs_open(g_zfs, be_root_ds, ZFS_TYPE_FILESYSTEM)) ==
1632            NULL) {
1633                be_print_err(gettext("be_destroy_zones: failed to "
1634                    "open BE root dataset (%s): %s\n"), be_root_ds,
1635                    libzfs_error_description(g_zfs));
  47                z_free_brand_list(brands);
1636                return (zfs_err_to_be_err(g_zfs));
```



```
1637        }

1639        /*
1640         * If the global BE is not mounted, we must mount it here to
1641         * gather data about the non-global zones in it.
1642         */
1643        if (!zfs_is_mounted(zhp, &mp)) {
1644                if ((ret = _be_mount(be_name, &mp,
1645                    BE_MOUNT_FLAG_NO_ZONES)) != BE_SUCCESS) {
1646                        be_print_err(gettext("be_destroy_zones: failed to "
1647                            "mount the BE (%s) for zones processing.\n"),
1648                            be_name);
1649                        ZFS_CLOSE(zhp);
  62                        z_free_brand_list(brands);
1650                        return (ret);
1651                }
1652        }
1653        ZFS_CLOSE(zhp);

1655        z_set_zone_root(mp);
1656        free(mp);

1658        /* Get list of zones. */
1659        if ((zlist = z_get_nonglobal_branded_zone_list()) == NULL)
  71        /* Get list of supported zones. */
  72        if ((zlist = z_get_nonglobal_zone_list_by_brand(brands)) == NULL) {
  73                z_free_brand_list(brands);
1660                return (BE_SUCCESS);
  75        }

1662        /* Unmount the BE before destroying the zones in it. */
1663        if (dd->force_unmount)
1664                force_umnt = BE_UNMOUNT_FLAG_FORCE;
1665        if ((ret = _be_unmount(be_name, force_umnt)) != BE_SUCCESS) {
1666                be_print_err(gettext("be_destroy_zones: failed to "
1667                    "unmount the BE (%s)\n"), be_name);
1668                goto done;
1669        }

1671        /* Iterate through the zones and destroy them. */
1672        for (i = 0; (zonename = z_zlist_get_zonename(zlist, i)) != NULL; i++) {
1673                boolean_t auto_create;

1675                if (z_zlist_is_zone_auto_create_be(zlist, i, &auto_create) != 0)
1676                        be_print_err(gettext("be_destroy_zones: failed to get "
1677                            "auto-create-be brand property\n"));
1678                        ret = -1; // XXX
1679                        goto done;
1680                }

1682                if (!auto_create)
1683                        continue;
1684 #endif /* ! codereview */

1686                /* Skip zones that aren't at least installed */
1687                if (z_zlist_get_current_state(zlist, i) < ZONE_STATE_INSTALLED)
1688                        continue;

1690                zonepath = z_zlist_get_zonepath(zlist, i);

1692                /*
1693                 * Get the dataset of this zonepath.  If its not
1694                 * a dataset, skip it.
1695                 */
1696                if ((zonepath_ds = be_get_ds_from_dir(zonepath)) == NULL)
1697                        continue;
```

```
1699                 /*
1700                  * Check if this zone is supported based on the
1701                  * dataset of its zonepath.
1702                  */
1703                 if (!be_zone_supported(zonepath_ds)) {
1704                         free(zonepath_ds);
1705                         continue;
1706                 }

1708                 /* Find the zone BE root datasets for this zone. */
1709                 if ((ret = be_destroy_zone_roots(zonepath_ds, dd))
1710                     != BE_SUCCESS) {
1711                         be_print_err(gettext("be_destroy_zones: failed to "
1712                             "find and destroy zone roots for zone %s\n"),
1713                             zonename);
1714                         free(zonepath_ds);
1715                         goto done;
1716                 }
1717                 free(zonepath_ds);
1718         }

1720 done:
  88         z_free_brand_list(brands);
1721         z_free_zone_list(zlist);

1723         return (ret);
1724 }
_____unchanged_portion_omitted_

1875 /*
1876  * Function:    be_copy_zones
1877  * Description: Find valid zones and clone them to create their
1878  *              corresponding datasets for the BE being created.
1879  * Parameters:
1880  *              obe_name - name of source global BE being copied.
1881  *              obe_root_ds - root dataset of source global BE being copied.
1882  *              nbe_root_ds - root dataset of target global BE.
1883  * Return:
1884  *              BE_SUCCESS - Success
1885  *              be_errno_t - Failure
1886  * Scope:
1887  *              Private
1888  */
1889 static int
1890 be_copy_zones(char *obe_name, char *obe_root_ds, char *nbe_root_ds)
1891 {
1892         int             i, num_retries;
1893         int             ret = BE_SUCCESS;
1894         int             iret = 0;
1895         char            *zonename = NULL;
1896         char            *zonepath = NULL;
1897         char            *zone_be_name = NULL;
1898         char            *temp_mntpt = NULL;
1899         char            *new_zone_be_name = NULL;
1900         char            zoneroot[MAXPATHLEN];
1901         char            zoneroot_ds[MAXPATHLEN];
1902         char            zone_container_ds[MAXPATHLEN];
1903         char            new_zoneroot_ds[MAXPATHLEN];
1904         char            ss[MAXPATHLEN];
1905         uuid_t          uu = { 0 };
1906         char            uu_string[UUID_PRINTABLE_STRING_LENGTH] = { 0 };
1907         be_transaction_data_t bt = { 0 };
1908         zfs_handle_t    *obe_zhp = NULL;
1909         zfs_handle_t    *nbe_zhp = NULL;
1910         zfs_handle_t    *z_zhp = NULL;
```



```
1911         zoneList_t      zlist = NULL;
 280         zoneBrandList_t *brands = NULL;
1912         boolean_t       mounted_here = B_FALSE;
1913         char            *snap_name = NULL;

1915         /* If zones are not implemented, then get out. */
1916         if (!z_zones_are_implemented()) {
1917                 return (BE_SUCCESS);
1918         }

 289         /* Get list of supported brands */
 290         if ((brands = be_get_supported_brandlist()) == NULL) {
 291                 be_print_err(gettext("be_copy_zones: "
 292                     "no supported brands\n"));
 293                 return (BE_SUCCESS);
 294         }

1920         /* Get handle to origin BE's root dataset */
1921         if ((obe_zhp = zfs_open(g_zfs, obe_root_ds, ZFS_TYPE_FILESYSTEM))
1922             == NULL) {
1923                 be_print_err(gettext("be_copy_zones: failed to open "
1924                     "the origin BE root dataset (%s) for zones processing: "
1925                     "%s\n"), obe_root_ds, libzfs_error_description(g_zfs));
1926                 return (zfs_err_to_be_err(g_zfs));
1927         }

1929         /* Get handle to newly cloned BE's root dataset */
1930         if ((nbe_zhp = zfs_open(g_zfs, nbe_root_ds, ZFS_TYPE_FILESYSTEM))
1931             == NULL) {
1932                 be_print_err(gettext("be_copy_zones: failed to open "
1933                     "the new BE root dataset (%s): %s\n"), nbe_root_ds,
1934                     libzfs_error_description(g_zfs));
1935                 ZFS_CLOSE(obe_zhp);
1936                 return (zfs_err_to_be_err(g_zfs));
1937         }

1939         /* Get the uuid of the newly cloned parent BE. */
1940         if (be_get_uuid(zfs_get_name(nbe_zhp), &uu) != BE_SUCCESS) {
1941                 be_print_err(gettext("be_copy_zones: "
1942                     "failed to get uuid for BE root "
1943                     "dataset %s\n"), zfs_get_name(nbe_zhp));
1944                 ZFS_CLOSE(nbe_zhp);
1945                 goto done;
1946         }
1947         ZFS_CLOSE(nbe_zhp);
1948         uuid_unparse(uu, uu_string);

1950         /*
1951          * If the origin BE is not mounted, we must mount it here to
1952          * gather data about the non-global zones in it.
1953          */
1954         if (!zfs_is_mounted(obe_zhp, &temp_mntpt)) {
1955                 if ((ret = _be_mount(obe_name, &temp_mntpt,
1956                     BE_MOUNT_FLAG_NULL)) != BE_SUCCESS) {
1957                         be_print_err(gettext("be_copy_zones: failed to "
1958                             "mount the BE (%s) for zones procesing.\n"),
1959                             obe_name);
1960                         goto done;
1961                 }
1962                 mounted_here = B_TRUE;
1963         }

1965         z_set_zone_root(temp_mntpt);

1967         /* Get list of zones. */
1968         if ((zlist = z_get_nonglobal_branded_zone_list()) == NULL) {
```

```
 343         /* Get list of supported zones. */
 344         if ((zlist = z_get_nonglobal_zone_list_by_brand(brands)) == NULL) {
1969                 ret = BE_SUCCESS;
1970                 goto done;
1971         }

1973         for (i = 0; (zonename = z_zlist_get_zonename(zlist, i)) != NULL; i++) {

1975                 be_fs_list_data_t       fld = { 0 };
1976                 char                    zonepath_ds[MAXPATHLEN];
1977                 char                    *ds = NULL;
1978                 boolean_t               auto_create;

1980                 if (z_zlist_is_zone_auto_create_be(zlist, i, &auto_create) != 0)
1981                         be_print_err(gettext("be_copy_zones: failed to get "
1982                             "auto-create-be brand property\n"));
1983                         ret = -1; // XXX
1984                 }

1986                 if (!auto_create)
1987                         continue;
1988 #endif /* ! codereview */

1990                 /* Get zonepath of zone */
1991                 zonepath = z_zlist_get_zonepath(zlist, i);

1993                 /* Skip zones that aren't at least installed */
1994                 if (z_zlist_get_current_state(zlist, i) < ZONE_STATE_INSTALLED)
1995                         continue;

1997                 /*
1998                  * Get the dataset of this zonepath.  If its not
1999                  * a dataset, skip it.
2000                  */
2001                 if ((ds = be_get_ds_from_dir(zonepath)) == NULL)
2002                         continue;

2004                 (void) strlcpy(zonepath_ds, ds, sizeof (zonepath_ds));
2005                 free(ds);
2006                 ds = NULL;

2008                 /* Get zoneroot directory */
2009                 be_make_zoneroot(zonepath, zoneroot, sizeof (zoneroot));

2011                 /* If zonepath dataset not supported, skip it. */
2012                 if (!be_zone_supported(zonepath_ds)) {
2013                         continue;
2014                 }

2016                 if ((ret = be_find_active_zone_root(obe_zhp, zonepath_ds,
2017                     zoneroot_ds, sizeof (zoneroot_ds))) != BE_SUCCESS) {
2018                         be_print_err(gettext("be_copy_zones: "
2019                             "failed to find active zone root for zone %s "
2020                             "in BE %s\n"), zonename, obe_name);
2021                         goto done;
2022                 }

2024                 be_make_container_ds(zonepath_ds, zone_container_ds,
2025                     sizeof (zone_container_ds));

2027                 if ((z_zhp = zfs_open(g_zfs, zoneroot_ds,
2028                     ZFS_TYPE_FILESYSTEM)) == NULL) {
2029                         be_print_err(gettext("be_copy_zones: "
2030                             "failed to open zone root dataset (%s): %s\n"),
2031                             zoneroot_ds, libzfs_error_description(g_zfs));
2032                         ret = zfs_err_to_be_err(g_zfs);
```



```
2033                         goto done;
2034                 }

2036                 zone_be_name =
2037                     be_get_zone_be_name(zoneroot_ds, zone_container_ds);

2039                 if ((new_zone_be_name = be_auto_zone_be_name(zone_container_ds,
2040                     zone_be_name)) == NULL) {
2041                         be_print_err(gettext("be_copy_zones: failed "
2042                             "to generate auto name for zone BE.\n"));
2043                         ret = BE_ERR_AUTONAME;
2044                         goto done;
2045                 }

2047                 if ((snap_name = be_auto_snap_name()) == NULL) {
2048                         be_print_err(gettext("be_copy_zones: failed to "
2049                             "generate snapshot name for zone BE.\n"));
2050                         ret = BE_ERR_AUTONAME;
2051                         goto done;
2052                 }

2054                 (void) snprintf(ss, sizeof (ss), "%s@%s", zoneroot_ds,
2055                     snap_name);

2057                 if (zfs_snapshot(g_zfs, ss, B_TRUE, NULL) != 0) {
2058                         be_print_err(gettext("be_copy_zones: "
2059                             "failed to snapshot zone BE (%s): %s\n"),
2060                             ss, libzfs_error_description(g_zfs));
2061                         if (libzfs_errno(g_zfs) == EZFS_EXISTS)
2062                                 ret = BE_ERR_ZONE_SS_EXISTS;
2063                         else
2064                                 ret = zfs_err_to_be_err(g_zfs);

2066                         goto done;
2067                 }

2069                 (void) snprintf(new_zoneroot_ds, sizeof (new_zoneroot_ds),
2070                     "%s/%s", zone_container_ds, new_zone_be_name);

2072                 bt.obe_name = zone_be_name;
2073                 bt.obe_root_ds = zoneroot_ds;
2074                 bt.obe_snap_name = snap_name;
2075                 bt.obe_altroot = temp_mntpt;
2076                 bt.nbe_name = new_zone_be_name;
2077                 bt.nbe_root_ds = new_zoneroot_ds;

2079                 if (nvlist_alloc(&bt.nbe_zfs_props, NV_UNIQUE_NAME, 0) != 0) {
2080                         be_print_err(gettext("be_copy_zones: "
2081                             "internal error: out of memory\n"));
2082                         ret = BE_ERR_NOMEM;
2083                         goto done;
2084                 }

2086                 /*
2087                  * The call to be_clone_fs_callback always closes the
2088                  * zfs_handle so there's no need to close z_zhp.
2089                  */
2090                 if ((iret = be_clone_fs_callback(z_zhp, &bt)) != 0) {
2091                         z_zhp = NULL;
2092                         if (iret != BE_ERR_BE_EXISTS) {
2093                                 be_print_err(gettext("be_copy_zones: "
2094                                     "failed to create zone BE clone for new "
2095                                     "zone BE %s\n"), new_zone_be_name);
2096                                 ret = iret;
2097                                 if (bt.nbe_zfs_props != NULL)
2098                                         nvlist_free(bt.nbe_zfs_props);
```

```
2099                            goto done;
2100                    }
2101                    /*
2102                     * We failed to create the new zone BE because a zone
2103                     * BE with the auto-name we generated above has since
2104                     * come into existence. Regenerate a new auto-name
2105                     * and retry.
2106                     */
2107                    for (num_retries = 1;
2108                        num_retries < BE_AUTO_NAME_MAX_TRY;
2109                        num_retries++) {

2111                            /* Sleep 1 before retrying */
2112                            (void) sleep(1);

2114                            /* Generate new auto zone BE name */
2115                            free(new_zone_be_name);
2116                            if ((new_zone_be_name = be_auto_zone_be_name(
2117                                zone_container_ds,
2118                                zone_be_name)) == NULL) {
2119                                be_print_err(gettext("be_copy_zones: "
2120                                    "failed to generate auto name "
2121                                    "for zone BE.\n"));
2122                                ret = BE_ERR_AUTONAME;
2123                                if (bt.nbe_zfs_props != NULL)
2124                                        nvlist_free(bt.nbe_zfs_props);
2125                                goto done;
2126                            }

2128                            (void) snprintf(new_zoneroot_ds,
2129                                sizeof (new_zoneroot_ds),
2130                                "%s/%s", zone_container_ds,
2131                                new_zone_be_name);
2132                            bt.nbe_name = new_zone_be_name;
2133                            bt.nbe_root_ds = new_zoneroot_ds;

2135                            /*
2136                             * Get handle to original zone BE's root
2137                             * dataset.
2138                             */
2139                            if ((z_zhp = zfs_open(g_zfs, zoneroot_ds,
2140                                ZFS_TYPE_FILESYSTEM)) == NULL) {
2141                                be_print_err(gettext("be_copy_zones: "
2142                                    "failed to open zone root "
2143                                    "dataset (%s): %s\n"),
2144                                    zoneroot_ds,
2145                                    libzfs_error_description(g_zfs));
2146                                ret = zfs_err_to_be_err(g_zfs);
2147                                if (bt.nbe_zfs_props != NULL)
2148                                        nvlist_free(bt.nbe_zfs_props);
2149                                goto done;
2150                            }

2152                            /*
2153                             * Try to clone the zone BE again. This
2154                             * call will end up closing the zfs
2155                             * handle passed in whether it
2156                             * succeeds or fails.
2157                             */
2158                            iret = be_clone_fs_callback(z_zhp, &bt);
2159                            z_zhp = NULL;
2160                            if (iret == 0) {
2161                                    break;
2162                            } else if (iret != BE_ERR_BE_EXISTS) {
2163                                    be_print_err(gettext("be_copy_zones: "
2164                                        "failed to create zone BE clone "
```



```
2165                                        "for new zone BE %s\n"),
2166                                        new_zone_be_name);
2167                                    ret = iret;
2168                                    if (bt.nbe_zfs_props != NULL)
2169                                            nvlist_free(bt.nbe_zfs_props);
2170                                    goto done;
2171                            }
2172                    }
2173                    /*
2174                     * If we've exhausted the maximum number of
2175                     * tries, free the auto zone BE name and return
2176                     * error.
2177                     */
2178                    if (num_retries == BE_AUTO_NAME_MAX_TRY) {
2179                            be_print_err(gettext("be_copy_zones: failed "
2180                                "to create a unique auto zone BE name\n"));
2181                            free(bt.nbe_name);
2182                            bt.nbe_name = NULL;
2183                            ret = BE_ERR_AUTONAME;
2184                            if (bt.nbe_zfs_props != NULL)
2185                                    nvlist_free(bt.nbe_zfs_props);
2186                            goto done;
2187                    }
2188            }

2190            if (bt.nbe_zfs_props != NULL)
2191                    nvlist_free(bt.nbe_zfs_props);

2193            z_zhp = NULL;

2195            if ((z_zhp = zfs_open(g_zfs, new_zoneroot_ds,
2196                ZFS_TYPE_FILESYSTEM)) == NULL) {
2197                    be_print_err(gettext("be_copy_zones: "
2198                        "failed to open the new zone BE root dataset "
2199                        "(%s): %s\n"), new_zoneroot_ds,
2200                        libzfs_error_description(g_zfs));
2201                    ret = zfs_err_to_be_err(g_zfs);
2202                    goto done;
2203            }

2205            if (zfs_prop_set(z_zhp, BE_ZONE_PARENTBE_PROPERTY,
2206                uu_string) != 0) {
2207                    be_print_err(gettext("be_copy_zones: "
2208                        "failed to set parentbe property\n"));
2209                    ZFS_CLOSE(z_zhp);
2210                    ret = zfs_err_to_be_err(g_zfs);
2211                    goto done;
2212            }

2214            if (zfs_prop_set(z_zhp, BE_ZONE_ACTIVE_PROPERTY, "on") != 0) {
2215                    be_print_err(gettext("be_copy_zones: "
2216                        "failed to set active property\n"));
2217                    ZFS_CLOSE(z_zhp);
2218                    ret = zfs_err_to_be_err(g_zfs);
2219                    goto done;
2220            }

2222            /*
2223             * Generate a list of file systems from the original
2224             * zone BE that are legacy mounted.  We use this list
2225             * to determine which entries in the vfstab we need to
2226             * update for the new zone BE we've just created.
2227             */
2228            if ((ret = be_get_legacy_fs(obe_name, obe_root_ds,
2229                zoneroot_ds, zoneroot, &fld)) != BE_SUCCESS) {
2230                    be_print_err(gettext("be_copy_zones: "
```

```
2231                            "failed to get legacy mounted file system "
2232                            "list for zone %s\n"), zonename);
2233                    ZFS_CLOSE(z_zhp);
2234                    goto done;
2235            }

2237            /*
2238             * Update new zone BE's vfstab.
2239             */
2240            if ((ret = be_update_zone_vfstab(z_zhp, bt.nbe_name,
2241                zonepath_ds, zonepath_ds, &fld)) != BE_SUCCESS) {
2242                    be_print_err(gettext("be_copy_zones: "
2243                        "failed to update new BE's vfstab (%s)\n"),
2244                        bt.nbe_name);
2245                    ZFS_CLOSE(z_zhp);
2246                    be_free_fs_list(&fld);
2247                    goto done;
2248            }

2250            be_free_fs_list(&fld);
2251            ZFS_CLOSE(z_zhp);
2252    }

2254 done:
2255    free(snap_name);
 354    if (brands != NULL)
 355            z_free_brand_list(brands);
2256    if (zlist != NULL)
2257            z_free_zone_list(zlist);

2259    if (mounted_here)
2260            (void) _be_unmount(obe_name, 0);

2262    ZFS_CLOSE(obe_zhp);
2263    return (ret);
2264 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   77577 Wed Nov 11 10:43:15 2015
new/usr/src/lib/libbe/common/be_mount.c
patch zone-auto-create-be
**********************************************************
```

```
/*
 * 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 2015 EveryCity Ltd.
 * Copyright 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

/*
 * 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;

/* Private function prototypes */
static int be_mount_callback(zfs_handle_t *, void *);
static int be_unmount_callback(zfs_handle_t *, void *);
static int be_get_legacy_fs_callback(zfs_handle_t *, void *);
static int fix_mountpoint(zfs_handle_t *);
static int fix_mountpoint_callback(zfs_handle_t *, void *);
static int get_mountpoint_from_vfstab(char *, const char *, char *, size_t,
    boolean_t);
static int loopback_mount_shared_fs(zfs_handle_t *, be_mount_data_t *);
static int loopback_mount_zonepath(const char *, be_mount_data_t *);
static int iter_shared_fs_callback(zfs_handle_t *, void *);
static int zpool_shared_fs_callback(zpool_handle_t *, void *);
static int unmount_shared_fs(be_unmount_data_t *);
static int add_to_fs_list(be_fs_list_data_t *, const char *);
static int be_mount_root(zfs_handle_t *, char *);
static int be_unmount_root(zfs_handle_t *, be_unmount_data_t *);
static int be_mount_zones(zfs_handle_t *, be_mount_data_t *);
static int be_unmount_zones(be_unmount_data_t *);
static int be_mount_one_zone(zfs_handle_t *, be_mount_data_t *, char *, char *,
    char *);
static int be_unmount_one_zone(be_unmount_data_t *, char *, char *, char *);
static int be_get_ds_from_dir_callback(zfs_handle_t *, void *);


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

/*
 * Function:    be_mount
 * Description: Mounts a BE and its subordinate datasets at a given mountpoint.
 * 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_MOUNTPOINT              *required
 *                      BE_ATTR_MOUNT_FLAGS             *optional
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Public
 */
int
be_mount(nvlist_t *be_attrs)
{
        char            *be_name = NULL;
        char            *mountpoint = NULL;
        uint16_t        flags = 0;
        int             ret = BE_SUCCESS;

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

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

```
128         /* Validate original BE name */
129         if (!be_valid_be_name(be_name)) {
130                 be_print_err(gettext("be_mount: invalid BE name %s\n"),
131                     be_name);
132                 return (BE_ERR_INVAL);
133         }

135         /* Get mountpoint */
136         if (nvlist_lookup_string(be_attrs, BE_ATTR_MOUNTPOINT, &mountpoint)
137             != 0) {
138                 be_print_err(gettext("be_mount: failed to lookup "
139                     "BE_ATTR_MOUNTPOINT attribute\n"));
140                 return (BE_ERR_INVAL);
141         }

143         /* Get flags */
144         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
145             BE_ATTR_MOUNT_FLAGS, DATA_TYPE_UINT16, &flags, NULL) != 0) {
146                 be_print_err(gettext("be_mount: failed to lookup "
147                     "BE_ATTR_MOUNT_FLAGS attribute\n"));
148                 return (BE_ERR_INVAL);
149         }

151         ret = _be_mount(be_name, &mountpoint, flags);

153         be_zfs_fini();

155         return (ret);
156 }

158 /*
159  * Function:    be_unmount
160  * Description: Unmounts a BE and its subordinate datasets.
161  * Parameters:
162  *              be_attrs - pointer to nvlist_t of attributes being passed in.
163  *                      The following attributes are used by this function:
164  *
165  *                      BE_ATTR_ORIG_BE_NAME            *required
166  *                      BE_ATTR_UNMOUNT_FLAGS           *optional
167  * Return:
168  *              BE_SUCCESS - Success
169  *              be_errno_t - Failure
170  * Scope:
171  *              Public
172  */
173 int
174 be_unmount(nvlist_t *be_attrs)
175 {
176         char            *be_name = NULL;
177         char            *be_name_mnt = NULL;
178         char            *ds = NULL;
179         uint16_t        flags = 0;
180         int             ret = BE_SUCCESS;

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

186         /* Get original BE name */
187         if (nvlist_lookup_string(be_attrs, BE_ATTR_ORIG_BE_NAME, &be_name)
188             != 0) {
189                 be_print_err(gettext("be_unmount: failed to lookup "
190                     "BE_ATTR_ORIG_BE_NAME attribute\n"));
191                 return (BE_ERR_INVAL);
192         }
```



```
194         /* Check if we have mountpoint argument instead of BE name */
195         if (be_name[0] == '/') {
196                 if ((ds = be_get_ds_from_dir(be_name)) != NULL) {
197                         if ((be_name_mnt = strrchr(ds, '/')) != NULL) {
198                                 be_name = be_name_mnt + 1;
199                         }
200                 } else {
201                         be_print_err(gettext("be_unmount: no datasets mounted "
202                             "at '%s'\n"), be_name);
203                         return (BE_ERR_INVAL);
204                 }
205         }

207         /* Validate original BE name */
208         if (!be_valid_be_name(be_name)) {
209                 be_print_err(gettext("be_unmount: invalid BE name %s\n"),
210                     be_name);
211                 return (BE_ERR_INVAL);
212         }

214         /* Get unmount flags */
215         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
216             BE_ATTR_UNMOUNT_FLAGS, DATA_TYPE_UINT16, &flags, NULL) != 0) {
217                 be_print_err(gettext("be_unmount: failed to loookup "
218                     "BE_ATTR_UNMOUNT_FLAGS attribute\n"));
219                 return (BE_ERR_INVAL);
220         }

222         ret = _be_unmount(be_name, flags);

224         be_zfs_fini();

226         return (ret);
227 }

229 /* ********************************************************************* */
230 /*                      Semi-Private Functions                           */
231 /* ********************************************************************* */

233 /*
234  * Function:    _be_mount
235  * Description: Mounts a BE.  If the altroot is not provided, this function
236  *              will generate a temporary mountpoint to mount the BE at.  It
237  *              will return this temporary mountpoint to the caller via the
238  *              altroot reference pointer passed in.  This returned value is
239  *              allocated on heap storage and is the repsonsibility of the
240  *              caller to free.
241  * Parameters:
242  *              be_name - pointer to name of BE to mount.
243  *              altroot - reference pointer to altroot of where to mount BE.
244  *              flags - flag indicating special handling for mounting the BE
245  * Return:
246  *              BE_SUCCESS - Success
247  *              be_errno_t - Failure
248  * Scope:
249  *              Semi-private (library wide use only)
250  */
251 int
252 _be_mount(char *be_name, char **altroot, int flags)
253 {
254         be_transaction_data_t   bt = { 0 };
255         be_mount_data_t md = { 0 };
256         zfs_handle_t    *zhp;
257         char            obe_root_ds[MAXPATHLEN];
258         char            *mp = NULL;
259         char            *tmp_altroot = NULL;
```

```
260         int             ret = BE_SUCCESS, err = 0;
261         uuid_t          uu = { 0 };
262         boolean_t       gen_tmp_altroot = B_FALSE;

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

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

270         /* Find which zpool obe_name lives in */
271         if ((err = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) == 0) {
272                 be_print_err(gettext("be_mount: failed to "
273                     "find zpool for BE (%s)\n"), bt.obe_name);
274                 return (BE_ERR_BE_NOENT);
275         } else if (err < 0) {
276                 be_print_err(gettext("be_mount: zpool_iter failed: %s\n"),
277                     libzfs_error_description(g_zfs));
278                 return (zfs_err_to_be_err(g_zfs));
279         }

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

286         /* Get handle to BE's root dataset */
287         if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_FILESYSTEM)) ==
288             NULL) {
289                 be_print_err(gettext("be_mount: failed to "
290                     "open BE root dataset (%s): %s\n"), bt.obe_root_ds,
291                     libzfs_error_description(g_zfs));
292                 return (zfs_err_to_be_err(g_zfs));
293         }

295         /* Make sure BE's root dataset isn't already mounted somewhere */
296         if (zfs_is_mounted(zhp, &mp)) {
297                 ZFS_CLOSE(zhp);
298                 be_print_err(gettext("be_mount: %s is already mounted "
299                     "at %s\n"), bt.obe_name, mp != NULL ? mp : "");
300                 free(mp);
301                 return (BE_ERR_MOUNTED);
302         }

304         /*
305          * Fix this BE's mountpoint if its root dataset isn't set to
306          * either 'legacy' or '/'.
307          */
308         if ((ret = fix_mountpoint(zhp)) != BE_SUCCESS) {
309                 be_print_err(gettext("be_mount: mountpoint check "
310                     "failed for %s\n"), bt.obe_root_ds);
311                 ZFS_CLOSE(zhp);
312                 return (ret);
313         }

315         /*
316          * If altroot not provided, create a temporary alternate root
317          * to mount on
318          */
319         if (*altroot == NULL) {
320                 if ((ret = be_make_tmp_mountpoint(&tmp_altroot))
321                     != BE_SUCCESS) {
322                         be_print_err(gettext("be_mount: failed to "
323                             "make temporary mountpoint\n"));
324                         ZFS_CLOSE(zhp);
325                         return (ret);
```



```
326                 }
327                 gen_tmp_altroot = B_TRUE;
328         } else {
329                 tmp_altroot = *altroot;
330         }

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

336         /* Mount the BE's root file system */
337         if (getzoneid() == GLOBAL_ZONEID) {
338                 if ((ret = be_mount_root(zhp, tmp_altroot)) != BE_SUCCESS) {
339                         be_print_err(gettext("be_mount: failed to "
340                             "mount BE root file system\n"));
341                         if (gen_tmp_altroot)
342                                 free(tmp_altroot);
343                         ZFS_CLOSE(zhp);
344                         return (ret);
345                 }
346         } else {
347                 /* Legacy mount the zone root dataset */
348                 if ((ret = be_mount_zone_root(zhp, &md)) != BE_SUCCESS) {
349                         be_print_err(gettext("be_mount: failed to "
350                             "mount BE zone root file system\n"));
351                         free(md.altroot);
352                         ZFS_CLOSE(zhp);
353                         return (ret);
354                 }
355         }

357         /* Iterate through BE's children filesystems */
358         if ((err = zfs_iter_filesystems(zhp, be_mount_callback,
359             tmp_altroot)) != 0) {
360                 be_print_err(gettext("be_mount: failed to "
361                     "mount BE (%s) on %s\n"), bt.obe_name, tmp_altroot);
362                 if (gen_tmp_altroot)
363                         free(tmp_altroot);
364                 ZFS_CLOSE(zhp);
365                 return (err);
366         }

368         /*
369          * Mount shared file systems if mount flag says so.
370          */
371         if (md.shared_fs) {
372                 /*
373                  * Mount all ZFS file systems not under the BE's root dataset
374                  */
375                 (void) zpool_iter(g_zfs, zpool_shared_fs_callback, &md);

377                 /* TODO: Mount all non-ZFS file systems - Not supported yet */
378         }

380         /*
381          * If we're in the global zone and the global zone has a valid uuid,
382          * mount all supported non-global zones.
383          */
384         if (getzoneid() == GLOBAL_ZONEID &&
385             !(flags & BE_MOUNT_FLAG_NO_ZONES) &&
386             be_get_uuid(bt.obe_root_ds, &uu) == BE_SUCCESS) {
387                 if (be_mount_zones(zhp, &md) != BE_SUCCESS) {
388                         ret = BE_ERR_NO_MOUNTED_ZONE;
389                 }
390         }
```

```
392         ZFS_CLOSE(zhp);
393
394         /*
395          * If a NULL altroot was passed in, pass the generated altroot
396          * back to the caller in altroot.
397          */
398         if (gen_tmp_altroot) {
399                 if (ret == BE_SUCCESS || ret == BE_ERR_NO_MOUNTED_ZONE)
400                         *altroot = tmp_altroot;
401                 else
402                         free(tmp_altroot);
403         }
404
405         return (ret);
406 }
407
408 /*
409  * Function:    _be_unmount
410  * Description: Unmount a BE.
411  * Parameters:
412  *              be_name - pointer to name of BE to unmount.
413  *              flags - flags for unmounting the BE.
414  * Returns:
415  *              BE_SUCCESS - Success
416  *              be_errno_t - Failure
417  * Scope:
418  *              Semi-private (library wide use only)
419  */
420 int
421 _be_unmount(char *be_name, int flags)
422 {
423         be_transaction_data_t   bt = { 0 };
424         be_unmount_data_t       ud = { 0 };
425         zfs_handle_t    *zhp;
426         uuid_t          uu = { 0 };
427         char            obe_root_ds[MAXPATHLEN];
428         char            mountpoint[MAXPATHLEN];
429         char            *mp = NULL;
430         int             ret = BE_SUCCESS;
431         int             zret = 0;
432
433         if (be_name == NULL)
434                 return (BE_ERR_INVAL);
435
436         /* Set be_name as obe_name in bt structure */
437         bt.obe_name = be_name;
438
439         /* Find which zpool obe_name lives in */
440         if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) == 0) {
441                 be_print_err(gettext("be_unmount: failed to "
442                     "find zpool for BE (%s)\n"), bt.obe_name);
443                 return (BE_ERR_BE_NOENT);
444         } else if (zret < 0) {
445                 be_print_err(gettext("be_unmount: "
446                     "zpool_iter failed: %s\n"),
447                     libzfs_error_description(g_zfs));
448                 ret = zfs_err_to_be_err(g_zfs);
449                 return (ret);
450         }
451
452         /* Generate string for obe_name's root dataset */
453         be_make_root_ds(bt.obe_zpool, bt.obe_name, obe_root_ds,
454             sizeof (obe_root_ds));
455         bt.obe_root_ds = obe_root_ds;
456
457         /* Get handle to BE's root dataset */
```



```
458         if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_FILESYSTEM)) ==
459             NULL) {
460                 be_print_err(gettext("be_unmount: failed to "
461                     "open BE root dataset (%s): %s\n"), bt.obe_root_ds,
462                     libzfs_error_description(g_zfs));
463                 ret = zfs_err_to_be_err(g_zfs);
464                 return (ret);
465         }
466
467         /* Make sure BE's root dataset is mounted somewhere */
468         if (!zfs_is_mounted(zhp, &mp)) {
469
470                 be_print_err(gettext("be_unmount: "
471                     "(%s) not mounted\n"), bt.obe_name);
472
473                 /*
474                  * BE is not mounted, fix this BE's mountpoint if its root
475                  * dataset isn't set to either 'legacy' or '/'.
476                  */
477                 if ((ret = fix_mountpoint(zhp)) != BE_SUCCESS) {
478                         be_print_err(gettext("be_unmount: mountpoint check "
479                             "failed for %s\n"), bt.obe_root_ds);
480                         ZFS_CLOSE(zhp);
481                         return (ret);
482                 }
483
484                 ZFS_CLOSE(zhp);
485                 return (BE_ERR_NOTMOUNTED);
486         }
487
488         /*
489          * If we didn't get a mountpoint from the zfs_is_mounted call,
490          * try and get it from its property.
491          */
492         if (mp == NULL) {
493                 if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
494                     sizeof (mountpoint), NULL, NULL, 0, B_FALSE) != 0) {
495                         be_print_err(gettext("be_unmount: failed to "
496                             "get mountpoint of (%s)\n"), bt.obe_name);
497                         ZFS_CLOSE(zhp);
498                         return (BE_ERR_ZFS);
499                 }
500         } else {
501                 (void) strlcpy(mountpoint, mp, sizeof (mountpoint));
502                 free(mp);
503         }
504
505         /* If BE mounted as current root, fail */
506         if (strcmp(mountpoint, "/") == 0) {
507                 be_print_err(gettext("be_unmount: "
508                     "cannot unmount currently running BE\n"));
509                 ZFS_CLOSE(zhp);
510                 return (BE_ERR_UMOUNT_CURR_BE);
511         }
512
513         ud.altroot = mountpoint;
514         ud.force = flags & BE_UNMOUNT_FLAG_FORCE;
515
516         /* Unmount all supported non-global zones if we're in the global zone */
517         if (getzoneid() == GLOBAL_ZONEID &&
518             be_get_uuid(bt.obe_root_ds, &uu) == BE_SUCCESS) {
519                 if ((ret = be_unmount_zones(&ud)) != BE_SUCCESS) {
520                         ZFS_CLOSE(zhp);
521                         return (ret);
522                 }
523         }
```

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

527         /* Unmount all ZFS file systems not under the BE root dataset */
528         if ((ret = unmount_shared_fs(&ud)) != BE_SUCCESS) {
529                 be_print_err(gettext("be_unmount: failed to "
530                     "unmount shared file systems\n"));
531                 ZFS_CLOSE(zhp);
532                 return (ret);
533         }

535         /* Unmount all children datasets under the BE's root dataset */
536         if ((zret = zfs_iter_filesystems(zhp, be_unmount_callback,
537             &ud)) != 0) {
538                 be_print_err(gettext("be_unmount: failed to "
539                     "unmount BE (%s)\n"), bt.obe_name);
540                 ZFS_CLOSE(zhp);
541                 return (zret);
542         }

544         /* Unmount this BE's root filesystem */
545         if (getzoneid() == GLOBAL_ZONEID) {
546                 if ((ret = be_unmount_root(zhp, &ud)) != BE_SUCCESS) {
547                         ZFS_CLOSE(zhp);
548                         return (ret);
549                 }
550         } else {
551                 if ((ret = be_unmount_zone_root(zhp, &ud)) != BE_SUCCESS) {
552                         ZFS_CLOSE(zhp);
553                         return (ret);
554                 }
555         }

557         ZFS_CLOSE(zhp);

559         return (BE_SUCCESS);
560 }

562 /*
563  * Function:    be_mount_zone_root
564  * Description: Mounts the zone root dataset for a zone.
565  * Parameters:
566  *              zfs - zfs_handle_t pointer to zone root dataset
567  *              md - be_mount_data_t pointer to data for zone to be mounted
568  * Returns:
569  *              BE_SUCCESS - Success
570  *              be_errno_t - Failure
571  * Scope:
572  *              Semi-private (library wide use only)
573  */
574 int
575 be_mount_zone_root(zfs_handle_t *zhp, be_mount_data_t *md)
576 {
577         struct stat buf;
578         char    mountpoint[MAXPATHLEN];
579         int     err = 0;

581         /* Get mountpoint property of dataset */
582         if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
583             sizeof (mountpoint), NULL, NULL, 0, B_FALSE) != 0) {
584                 be_print_err(gettext("be_mount_zone_root: failed to "
585                     "get mountpoint property for %s: %s\n"), zfs_get_name(zhp),
586                     libzfs_error_description(g_zfs));
587                 return (zfs_err_to_be_err(g_zfs));
588         }
```



```
590         /*
591          * Make sure zone's root dataset is set to 'legacy'.  This is
592          * currently a requirement in this implementation of zones
593          * support.
594          */
595         if (strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) != 0) {
596                 be_print_err(gettext("be_mount_zone_root: "
597                     "zone root dataset mountpoint is not 'legacy'\n"));
598                 return (BE_ERR_ZONE_ROOT_NOT_LEGACY);
599         }

601         /* Create the mountpoint if it doesn't exist */
602         if (lstat(md->altroot, &buf) != 0) {
603                 if (mkdirp(md->altroot, 0755) != 0) {
604                         err = errno;
605                         be_print_err(gettext("be_mount_zone_root: failed "
606                             "to create mountpoint %s\n"), md->altroot);
607                         return (errno_to_be_err(err));
608                 }
609         }

611         /*
612          * Legacy mount the zone root dataset.
613          *
614          * As a workaround for 6176743, we mount the zone's root with the
615          * MS_OVERLAY option in case an alternate BE is mounted, and we're
616          * mounting the root for the zone from the current BE here.  When an
617          * alternate BE is mounted, it ties up the zone's zoneroot directory
618          * for the current BE since the zone's zonepath is loopback mounted
619          * from the current BE.
620          *
621          * TODO: The MS_OVERLAY option needs to be removed when 6176743
622          * is fixed.
623          */
624         if (mount(zfs_get_name(zhp), md->altroot, MS_OVERLAY, MNTTYPE_ZFS,
625             NULL, 0, NULL, 0) != 0) {
626                 err = errno;
627                 be_print_err(gettext("be_mount_zone_root: failed to "
628                     "legacy mount zone root dataset (%s) at %s\n"),
629                     zfs_get_name(zhp), md->altroot);
630                 return (errno_to_be_err(err));
631         }

633         return (BE_SUCCESS);
634 }

636 /*
637  * Function:    be_unmount_zone_root
638  * Description: Unmounts the zone root dataset for a zone.
639  * Parameters:
640  *              zhp - zfs_handle_t pointer to zone root dataset
641  *              ud - be_unmount_data_t pointer to data for zone to be unmounted
642  * Returns:
643  *              BE_SUCCESS - Success
644  *              be_errno_t - Failure
645  * Scope:
646  *              Semi-private (library wise use only)
647  */
648 int
649 be_unmount_zone_root(zfs_handle_t *zhp, be_unmount_data_t *ud)
650 {
651         char    mountpoint[MAXPATHLEN];

653         /* Unmount the dataset */
654         if (zfs_unmount(zhp, NULL, ud->force ? MS_FORCE : 0) != 0) {
655                 be_print_err(gettext("be_unmount_zone_root: failed to "
```

```
656                     "unmount zone root dataset %s: %s\n"), zfs_get_name(zhp),
657                     libzfs_error_description(g_zfs));
658                 return (zfs_err_to_be_err(g_zfs));
659         }
660
661         /* Get the current mountpoint property for the zone root dataset */
662         if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
663             sizeof (mountpoint), NULL, NULL, 0, B_FALSE) != 0) {
664                 be_print_err(gettext("be_unmount_zone_root: failed to "
665                     "get mountpoint property for zone root dataset (%s): %s\n"),
666                     zfs_get_name(zhp), libzfs_error_description(g_zfs));
667                 return (zfs_err_to_be_err(g_zfs));
668         }
669
670         /* If mountpoint not already set to 'legacy', set it to 'legacy' */
671         if (strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) != 0) {
672                 if (zfs_prop_set(zhp, zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
673                     ZFS_MOUNTPOINT_LEGACY) != 0) {
674                         be_print_err(gettext("be_unmount_zone_root: "
675                             "failed to set mountpoint of zone root dataset "
676                             "%s to 'legacy': %s\n"), zfs_get_name(zhp),
677                             libzfs_error_description(g_zfs));
678                         return (zfs_err_to_be_err(g_zfs));
679                 }
680         }
681
682         return (BE_SUCCESS);
683 }
684
685 /*
686  * Function:    be_get_legacy_fs
687  * Description: This function iterates through all non-shared file systems
688  *              of a BE and finds the ones with a legacy mountpoint.  For
689  *              those file systems, it reads the BE's vfstab to get the
690  *              mountpoint.  If found, it adds that file system to the
691  *              be_fs_list_data_t passed in.
692  *
693  *              This function can be used to gather legacy mounted file systems
694  *              for both global BEs and non-global zone BEs.  To get data for
695  *              a non-global zone BE, the zoneroot_ds and zoneroot parameters
696  *              will be specified, otherwise they should be set to NULL.
697  * Parameters:
698  *              be_name - global BE name from which to get legacy file
699  *                      system list.
700  *              be_root_ds - root dataset of global BE.
701  *              zoneroot_ds - root dataset of zone.
702  *              zoneroot - zoneroot path of zone.
703  *              fld - be_fs_list_data_t pointer.
704  * Returns:
705  *              BE_SUCCESS - Success
706  *              be_errno_t - Failure
707  * Scope:
708  *              Semi-private (library wide use only)
709  */
710 int
711 be_get_legacy_fs(char *be_name, char *be_root_ds, char *zoneroot_ds,
712     char *zoneroot, be_fs_list_data_t *fld)
713 {
714         zfs_handle_t            *zhp = NULL;
715         char                    mountpoint[MAXPATHLEN];
716         boolean_t               mounted_here = B_FALSE;
717         boolean_t               zone_mounted_here = B_FALSE;
718         int                     ret = BE_SUCCESS, err = 0;
719
720         if (be_name == NULL || be_root_ds == NULL || fld == NULL)
721                 return (BE_ERR_INVAL);
```



```
723         /* Get handle to BE's root dataset */
724         if ((zhp = zfs_open(g_zfs, be_root_ds, ZFS_TYPE_FILESYSTEM))
725             == NULL) {
726                 be_print_err(gettext("be_get_legacy_fs: failed to "
727                     "open BE root dataset (%s): %s\n"), be_root_ds,
728                     libzfs_error_description(g_zfs));
729                 ret = zfs_err_to_be_err(g_zfs);
730                 return (ret);
731         }
732
733         /* If BE is not already mounted, mount it. */
734         if (!zfs_is_mounted(zhp, &fld->altroot)) {
735                 if ((ret = _be_mount(be_name, &fld->altroot,
736                     zoneroot_ds ? BE_MOUNT_FLAG_NULL :
737                     BE_MOUNT_FLAG_NO_ZONES)) != BE_SUCCESS) {
738                         be_print_err(gettext("be_get_legacy_fs: "
739                             "failed to mount BE %s\n"), be_name);
740                         goto cleanup;
741                 }
742
743                 mounted_here = B_TRUE;
744         } else if (fld->altroot == NULL) {
745                 be_print_err(gettext("be_get_legacy_fs: failed to "
746                     "get altroot of mounted BE %s: %s\n"),
747                     be_name, libzfs_error_description(g_zfs));
748                 ret = zfs_err_to_be_err(g_zfs);
749                 goto cleanup;
750         }
751
752         /*
753          * If a zone root dataset was passed in, we're wanting to get
754          * legacy mounted file systems for that zone, not the global
755          * BE.
756          */
757         if (zoneroot_ds != NULL) {
758                 be_mount_data_t         zone_md = { 0 };
759
760                 /* Close off handle to global BE's root dataset */
761                 ZFS_CLOSE(zhp);
762
763                 /* Get handle to zone's root dataset */
764                 if ((zhp = zfs_open(g_zfs, zoneroot_ds,
765                     ZFS_TYPE_FILESYSTEM)) == NULL) {
766                         be_print_err(gettext("be_get_legacy_fs: failed to "
767                             "open zone BE root dataset (%s): %s\n"),
768                             zoneroot_ds, libzfs_error_description(g_zfs));
769                         ret = zfs_err_to_be_err(g_zfs);
770                         goto cleanup;
771                 }
772
773                 /* Make sure the zone we're looking for is mounted */
774                 if (!zfs_is_mounted(zhp, &zone_md.altroot)) {
775                         char    zone_altroot[MAXPATHLEN];
776
777                         /* Generate alternate root path for zone */
778                         (void) snprintf(zone_altroot, sizeof (zone_altroot),
779                             "%s%s", fld->altroot, zoneroot);
780                         if ((zone_md.altroot = strdup(zone_altroot)) == NULL) {
781                                 be_print_err(gettext("be_get_legacy_fs: "
782                                     "memory allocation failed\n"));
783                                 ret = BE_ERR_NOMEM;
784                                 goto cleanup;
785                         }
786
787                         if ((ret = be_mount_zone_root(zhp, &zone_md))
```

```
788                             != BE_SUCCESS) {
789                                 be_print_err(gettext("be_get_legacy_fs: "
790                                     "failed to mount zone root %s\n"),
791                                     zoneroot_ds);
792                                 free(zone_md.altroot);
793                                 zone_md.altroot = NULL;
794                                 goto cleanup;
795                         }
796                         zone_mounted_here = B_TRUE;
797                 }
798
799                 free(fld->altroot);
800                 fld->altroot = zone_md.altroot;
801         }
802
803         /*
804          * If the root dataset is in the vfstab with a mountpoint of "/",
805          * add it to the list
806          */
807         if (get_mountpoint_from_vfstab(fld->altroot, zfs_get_name(zhp),
808             mountpoint, sizeof (mountpoint), B_FALSE) == BE_SUCCESS) {
809                 if (strcmp(mountpoint, "/") == 0) {
810                         if (add_to_fs_list(fld, zfs_get_name(zhp))
811                             != BE_SUCCESS) {
812                                 be_print_err(gettext("be_get_legacy_fs: "
813                                     "failed to add %s to fs list\n"),
814                                     zfs_get_name(zhp));
815                                 ret = BE_ERR_INVAL;
816                                 goto cleanup;
817                         }
818                 }
819         }
820
821         /* Iterate subordinate file systems looking for legacy mounts */
822         if ((ret = zfs_iter_filesystems(zhp, be_get_legacy_fs_callback,
823             fld)) != 0) {
824                 be_print_err(gettext("be_get_legacy_fs: "
825                     "failed to iterate  %s to get legacy mounts\n"),
826                     zfs_get_name(zhp));
827         }
828
829 cleanup:
830         /* If we mounted the zone BE, unmount it */
831         if (zone_mounted_here) {
832                 be_unmount_data_t       zone_ud = { 0 };
833
834                 zone_ud.altroot = fld->altroot;
835                 zone_ud.force = B_TRUE;
836                 if ((err = be_unmount_zone_root(zhp, &zone_ud)) != BE_SUCCESS) {
837                         be_print_err(gettext("be_get_legacy_fs: "
838                             "failed to unmount zone root %s\n"),
839                             zoneroot_ds);
840                         if (ret == BE_SUCCESS)
841                                 ret = err;
842                 }
843         }
844
845         /* If we mounted this BE, unmount it */
846         if (mounted_here) {
847                 if ((err = _be_unmount(be_name, 0)) != BE_SUCCESS) {
848                         be_print_err(gettext("be_get_legacy_fs: "
849                             "failed to unmount %s\n"), be_name);
850                         if (ret == BE_SUCCESS)
851                                 ret = err;
852                 }
853         }
```



```
855         ZFS_CLOSE(zhp);
856
857         free(fld->altroot);
858         fld->altroot = NULL;
859
860         return (ret);
861 }
862
863 /*
864  * Function:    be_free_fs_list
865  * Description: Function used to free the members of a be_fs_list_data_t
866  *                      structure.
867  * Parameters:
868  *              fld - be_fs_list_data_t pointer to free.
869  * Returns:
870  *              None
871  * Scope:
872  *              Semi-private (library wide use only)
873  */
874 void
875 be_free_fs_list(be_fs_list_data_t *fld)
876 {
877         int     i;
878
879         if (fld == NULL)
880                 return;
881
882         free(fld->altroot);
883
884         if (fld->fs_list == NULL)
885                 return;
886
887         for (i = 0; i < fld->fs_num; i++)
888                 free(fld->fs_list[i]);
889
890         free(fld->fs_list);
891 }
892
893 /*
894  * Function:    be_get_ds_from_dir(char *dir)
895  * Description: Given a directory path, find the underlying dataset mounted
896  *              at that directory path if there is one.   The returned name
897  *              is allocated in heap storage, so the caller is responsible
898  *              for freeing it.
899  * Parameters:
900  *              dir - char pointer of directory to find.
901  * Returns:
902  *              NULL - if directory is not mounted from a dataset.
903  *              name of dataset mounted at dir.
904  * Scope:
905  *              Semi-private (library wide use only)
906  */
907 char *
908 be_get_ds_from_dir(char *dir)
909 {
910         dir_data_t      dd = { 0 };
911         char            resolved_dir[MAXPATHLEN];
912
913         /* Make sure length of dir is within the max length */
914         if (dir == NULL || strlen(dir) >= MAXPATHLEN)
915                 return (NULL);
916
917         /* Resolve dir in case its lofs mounted */
918         (void) strlcpy(resolved_dir, dir, sizeof (resolved_dir));
919         z_resolve_lofs(resolved_dir, sizeof (resolved_dir));
```

```
921         dd.dir = resolved_dir;

923         (void) zfs_iter_root(g_zfs, be_get_ds_from_dir_callback, &dd);

925         return (dd.ds);
926 }

928 /*
929  * Function:    be_make_tmp_mountpoint
930  * Description: This function generates a random temporary mountpoint
931  *              and creates that mountpoint directory.  It returns the
932  *              mountpoint in heap storage, so the caller is responsible
933  *              for freeing it.
934  * Parameters:
935  *              tmp_mp - reference to pointer of where to store generated
936  *                       temporary mountpoint.
937  * Returns:
938  *              BE_SUCCESS - Success
939  *              be_errno_t - Failure
940  * Scope:
941  *              Semi-private (library wide use only)
942  */
943 int
944 be_make_tmp_mountpoint(char **tmp_mp)
945 {
946         int     err = 0;

948         if ((*tmp_mp = (char *)calloc(1, sizeof (BE_TMP_MNTPNT) + 1)) == NULL) {
949                 be_print_err(gettext("be_make_tmp_mountpoint: "
950                     "malloc failed\n"));
951                 return (BE_ERR_NOMEM);
952         }
953         (void) strlcpy(*tmp_mp, BE_TMP_MNTPNT, sizeof (BE_TMP_MNTPNT) + 1);
954         if (mkdtemp(*tmp_mp) == NULL) {
955                 err = errno;
956                 be_print_err(gettext("be_make_tmp_mountpoint: mkdtemp() failed "
957                     "for %s: %s\n"), *tmp_mp, strerror(err));
958                 free(*tmp_mp);
959                 *tmp_mp = NULL;
960                 return (errno_to_be_err(err));
961         }

963         return (BE_SUCCESS);
964 }

966 /*
967  * Function:    be_mount_pool
968  * Description: This function determines if the pool's datase is mounted
969  *              and if not it is used to mount the pool's dataset. The
970  *              function returns the current mountpoint if we are able
971  *              to mount the dataset.
972  * Parameters:
973  *              zhp - handle to the pool's dataset
974  *              tmp_mntpnt - The temporary mountpoint that the pool's
975  *                           dataset is mounted on. This is set only
976  *                           if the attempt to mount the dataset at it's
977  *                           set mountpoint fails, and we've used a
978  *                           temporary mount point for this dataset. It
979  *                           is expected that the caller will free this
980  *                           memory.
981  *              orig_mntpnt - The original mountpoint for the pool. If a
982  *                           temporary mount point was needed this will
983  *                           be used to reset the mountpoint property to
984  *                           it's original mountpoint. It is expected that
985  *                           the caller will free this memory.
```



```
986  *              pool_mounted - This flag indicates that the pool was mounted
987  *                             in this function.
988  * Returns:
989  *              BE_SUCCESS - Success
990  *              be_errno_t - Failure
991  * Scope:
992  *              Semi-private (library wide use only)
993  */
994 int
995 be_mount_pool(
996         zfs_handle_t *zhp,
997         char **tmp_mntpnt,
998         char **orig_mntpnt,
999         boolean_t *pool_mounted)
1000 {

1002         char            mountpoint[MAXPATHLEN];
1003         int             ret = 0;

1005         *tmp_mntpnt = NULL;
1006         *orig_mntpnt = NULL;
1007         *pool_mounted = B_FALSE;

1009         if (!zfs_is_mounted(zhp, NULL)) {
1010                 if (zfs_mount(zhp, NULL, 0) != 0) {
1011                         if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
1012                             sizeof (mountpoint), NULL, NULL, 0, B_FALSE) != 0) {
1013                                 be_print_err(gettext("be_mount_pool: failed to "
1014                                     "get mountpoint of (%s): %s\n"),
1015                                     zfs_get_name(zhp),
1016                                     libzfs_error_description(g_zfs));
1017                                 return (zfs_err_to_be_err(g_zfs));
1018                         }
1019                         if ((*orig_mntpnt = strdup(mountpoint)) == NULL) {
1020                                 be_print_err(gettext("be_mount_pool: memory "
1021                                     "allocation failed\n"));
1022                                 return (BE_ERR_NOMEM);
1023                         }
1024                         /*
1025                          * attempt to mount on a temp mountpoint
1026                          */
1027                         if ((ret = be_make_tmp_mountpoint(tmp_mntpnt))
1028                             != BE_SUCCESS) {
1029                                 be_print_err(gettext("be_mount_pool: failed "
1030                                     "to make temporary mountpoint\n"));
1031                                 free(*orig_mntpnt);
1032                                 *orig_mntpnt = NULL;
1033                                 return (ret);
1034                         }

1036                         if (zfs_prop_set(zhp,
1037                             zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
1038                             *tmp_mntpnt) != 0) {
1039                                 be_print_err(gettext("be_mount_pool: failed "
1040                                     "to set mountpoint of pool dataset %s to "
1041                                     "%s: %s\n"), zfs_get_name(zhp),
1042                                     *orig_mntpnt,
1043                                     libzfs_error_description(g_zfs));
1044                                 free(*tmp_mntpnt);
1045                                 free(*orig_mntpnt);
1046                                 *orig_mntpnt = NULL;
1047                                 *tmp_mntpnt = NULL;
1048                                 return (zfs_err_to_be_err(g_zfs));
1049                         }

1051                         if (zfs_mount(zhp, NULL, 0) != 0) {
```

```
1052                                be_print_err(gettext("be_mount_pool: failed "
1053                                    "to mount dataset %s at %s: %s\n"),
1054                                    zfs_get_name(zhp), *tmp_mntpnt,
1055                                    libzfs_error_description(g_zfs));
1056                                ret = zfs_err_to_be_err(g_zfs);
1057                                if (zfs_prop_set(zhp,
1058                                    zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
1059                                    mountpoint) != 0) {
1060                                        be_print_err(gettext("be_mount_pool: "
1061                                            "failed to set mountpoint of pool "
1062                                            "dataset %s to %s: %s\n"),
1063                                            zfs_get_name(zhp), *tmp_mntpnt,
1064                                            libzfs_error_description(g_zfs));
1065                                }
1066                                free(*tmp_mntpnt);
1067                                free(*orig_mntpnt);
1068                                *orig_mntpnt = NULL;
1069                                *tmp_mntpnt = NULL;
1070                                return (ret);
1071                        }
1072                }
1073                *pool_mounted = B_TRUE;
1074        }

1076        return (BE_SUCCESS);
1077 }

1079 /*
1080  * Function:    be_unmount_pool
1081  * Description: This function is used to unmount the pool's dataset if we
1082  *              mounted it previously using be_mount_pool().
1083  * Parameters:
1084  *              zhp - handle to the pool's dataset
1085  *              tmp_mntpnt - If a temprary mount point was used this will
1086  *                           be set. Since this was created in be_mount_pool
1087  *                           we will need to clean it up here.
1088  *              orig_mntpnt - The original mountpoint for the pool. This is
1089  *                            used to set the dataset mountpoint property
1090  *                            back to it's original value in the case where a
1091  *                            temporary mountpoint was used.
1092  * Returns:
1093  *              BE_SUCCESS - Success
1094  *              be_errno_t - Failure
1095  * Scope:
1096  *              Semi-private (library wide use only)
1097  */
1098 int
1099 be_unmount_pool(
1100         zfs_handle_t *zhp,
1101         char *tmp_mntpnt,
1102         char *orig_mntpnt)
1103 {
1104         if (zfs_unmount(zhp, NULL, 0) != 0) {
1105                 be_print_err(gettext("be_unmount_pool: failed to "
1106                     "unmount pool (%s): %s\n"), zfs_get_name(zhp),
1107                     libzfs_error_description(g_zfs));
1108                 return (zfs_err_to_be_err(g_zfs));
1109         }
1110         if (orig_mntpnt != NULL) {
1111                 if (tmp_mntpnt != NULL &&
1112                     strcmp(orig_mntpnt, tmp_mntpnt) != 0) {
1113                         (void) rmdir(tmp_mntpnt);
1114                 }
1115                 if (zfs_prop_set(zhp,
1116                     zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
1117                     orig_mntpnt) != 0) {
```



```
1118                        be_print_err(gettext("be_unmount_pool: failed "
1119                            "to set the mountpoint for dataset (%s) to "
1120                            "%s: %s\n"), zfs_get_name(zhp), orig_mntpnt,
1121                            libzfs_error_description(g_zfs));
1122                        return (zfs_err_to_be_err(g_zfs));
1123                }
1124        }

1126        return (BE_SUCCESS);
1127 }

1129 /* ************************************************************** */
1130 /*                      Private Functions                         */
1131 /* ************************************************************** */

1133 /*
1134  * Function:    be_mount_callback
1135  * Description: Callback function used to iterate through all of a BE's
1136  *              subordinate file systems and to mount them accordingly.
1137  * Parameters:
1138  *              zhp - zfs_handle_t pointer to current file system being
1139  *                    processed.
1140  *              data - pointer to the altroot of where to mount BE.
1141  * Returns:
1142  *              0 - Success
1143  *              be_errno_t - Failure
1144  * Scope:
1145  *              Private
1146  */
1147 static int
1148 be_mount_callback(zfs_handle_t *zhp, void *data)
1149 {
1150         zprop_source_t  sourcetype;
1151         const char      *fs_name = zfs_get_name(zhp);
1152         char            source[ZFS_MAXNAMELEN];
1153         char            *altroot = data;
1154         char            zhp_mountpoint[MAXPATHLEN];
1155         char            mountpoint[MAXPATHLEN];
1156         int             ret = 0;

1158         /* Get dataset's mountpoint and source values */
1159         if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, zhp_mountpoint,
1160             sizeof (zhp_mountpoint), &sourcetype, source, sizeof (source),
1161             B_FALSE) != 0) {
1162                 be_print_err(gettext("be_mount_callback: failed to "
1163                     "get mountpoint and sourcetype for %s\n"),
1164                     fs_name);
1165                 ZFS_CLOSE(zhp);
1166                 return (BE_ERR_ZFS);
1167         }

1169         /*
1170          * Set this filesystem's 'canmount' property to 'noauto' just incase
1171          * it's been set 'on'.  We do this so that when we change its
1172          * mountpoint zfs won't immediately try to mount it.
1173          */
1174         if (zfs_prop_set(zhp, zfs_prop_to_name(ZFS_PROP_CANMOUNT), "noauto")) {
1175                 be_print_err(gettext("be_mount_callback: failed to "
1176                     "set canmount to 'noauto' (%s)\n"), fs_name);
1177                 ZFS_CLOSE(zhp);
1178                 return (BE_ERR_ZFS);
1179         }

1181         /*
1182          * If the mountpoint is none, there's nothing to do, goto next.
1183          * If the mountpoint is legacy, legacy mount it with mount(2).
```

```
1184          * If the mountpoint is inherited, its mountpoint should
1185          * already be set.  If it's not, then explicitly fix-up
1186          * the mountpoint now by appending its explicitly set
1187          * mountpoint value to the BE mountpoint.
1188          */
1189         if (strcmp(zhp_mountpoint, ZFS_MOUNTPOINT_NONE) == 0) {
1190                 goto next;
1191         } else if (strcmp(zhp_mountpoint, ZFS_MOUNTPOINT_LEGACY) == 0) {
1192                 /*
1193                  * If the mountpoint is set to 'legacy', we need to
1194                  * dig into this BE's vfstab to figure out where to
1195                  * mount it, and just mount it via mount(2).
1196                  */
1197                 if (get_mountpoint_from_vfstab(altroot, fs_name,
1198                     mountpoint, sizeof (mountpoint), B_TRUE) == BE_SUCCESS) {

1200                         /* Legacy mount the file system */
1201                         if (mount(fs_name, mountpoint, MS_DATA,
1202                             MNTTYPE_ZFS, NULL, 0, NULL, 0) != 0) {
1203                                 be_print_err(
1204                                     gettext("be_mount_callback: "
1205                                     "failed to mount %s on %s\n"),
1206                                     fs_name, mountpoint);
1207                         }
1208                 } else {
1209                         be_print_err(
1210                             gettext("be_mount_callback: "
1211                             "no entry for %s in vfstab, "
1212                             "skipping ...\n"), fs_name);
1213                 }

1215                 goto next;

1217         } else if (sourcetype & ZPROP_SRC_INHERITED) {
1218                 /*
1219                  * If the mountpoint is inherited, its parent should have
1220                  * already been processed so its current mountpoint value
1221                  * is what its mountpoint ought to be.
1222                  */
1223                 (void) strlcpy(mountpoint, zhp_mountpoint, sizeof (mountpoint));
1224         } else if (sourcetype & ZPROP_SRC_LOCAL) {
1225                 /*
1226                  * Else process dataset with explicitly set mountpoint.
1227                  */
1228                 (void) snprintf(mountpoint, sizeof (mountpoint),
1229                     "%s%s", altroot, zhp_mountpoint);

1231                 /* Set the new mountpoint for the dataset */
1232                 if (zfs_prop_set(zhp,
1233                     zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
1234                     mountpoint)) {
1235                         be_print_err(gettext("be_mount_callback: "
1236                             "failed to set mountpoint for %s to "
1237                             "%s\n"), fs_name, mountpoint);
1238                         ZFS_CLOSE(zhp);
1239                         return (BE_ERR_ZFS);
1240                 }
1241         } else {
1242                 be_print_err(gettext("be_mount_callback: "
1243                     "mountpoint sourcetype of %s is %d, skipping ...\n"),
1244                     fs_name, sourcetype);

1246                 goto next;
1247         }

1249         /* Mount this filesystem */
```



```
1250         if (zfs_mount(zhp, NULL, 0) != 0) {
1251                 be_print_err(gettext("be_mount_callback: failed to "
1252                     "mount dataset %s at %s: %s\n"), fs_name, mountpoint,
1253                     libzfs_error_description(g_zfs));
1254                 /*
1255                  * Set this filesystem's 'mountpoint' property back to what
1256                  * it was
1257                  */
1258                 if (sourcetype & ZPROP_SRC_LOCAL &&
1259                     strcmp(zhp_mountpoint, ZFS_MOUNTPOINT_LEGACY) != 0) {
1260                         (void) zfs_prop_set(zhp,
1261                             zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
1262                             zhp_mountpoint);
1263                 }

1265                 ZFS_CLOSE(zhp);
1266                 return (BE_ERR_MOUNT);
1267         }

1269 next:
1270         /* Iterate through this dataset's children and mount them */
1271         if ((ret = zfs_iter_filesystems(zhp, be_mount_callback,
1272             altroot)) != 0) {
1273                 ZFS_CLOSE(zhp);
1274                 return (ret);
1275         }


1278         ZFS_CLOSE(zhp);
1279         return (0);
1280 }

1282 /*
1283  * Function:    be_unmount_callback
1284  * Description: Callback function used to iterate through all of a BE's
1285  *              subordinate file systems and to unmount them.
1286  * Parameters:
1287  *              zhp - zfs_handle_t pointer to current file system being
1288  *                    processed.
1289  *              data - pointer to the mountpoint of where BE is mounted.
1290  * Returns:
1291  *              0 - Success
1292  *              be_errno_t - Failure
1293  * Scope:
1294  *              Private
1295  */
1296 static int
1297 be_unmount_callback(zfs_handle_t *zhp, void *data)
1298 {
1299         be_unmount_data_t       *ud = data;
1300         zprop_source_t  sourcetype;
1301         const char      *fs_name = zfs_get_name(zhp);
1302         char            source[ZFS_MAXNAMELEN];
1303         char            mountpoint[MAXPATHLEN];
1304         char            *zhp_mountpoint;
1305         int             ret = 0;

1307         /* Iterate down this dataset's children first */
1308         if (zfs_iter_filesystems(zhp, be_unmount_callback, ud)) {
1309                 ret = BE_ERR_UMOUNT;
1310                 goto done;
1311         }

1313         /* Is dataset even mounted ? */
1314         if (!zfs_is_mounted(zhp, NULL))
1315                 goto done;
```

```
1317         /* Unmount this file system */
1318         if (zfs_unmount(zhp, NULL, ud->force ? MS_FORCE : 0) != 0) {
1319                 be_print_err(gettext("be_unmount_callback: "
1320                     "failed to unmount %s: %s\n"), fs_name,
1321                     libzfs_error_description(g_zfs));
1322                 ret = zfs_err_to_be_err(g_zfs);
1323                 goto done;
1324         }

1326         /* Get dataset's current mountpoint and source value */
1327         if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
1328             sizeof (mountpoint), &sourcetype, source, sizeof (source),
1329             B_FALSE) != 0) {
1330                 be_print_err(gettext("be_unmount_callback: "
1331                     "failed to get mountpoint and sourcetype for %s: %s\n"),
1332                     fs_name, libzfs_error_description(g_zfs));
1333                 ret = zfs_err_to_be_err(g_zfs);
1334                 goto done;
1335         }

1337         if (sourcetype & ZPROP_SRC_INHERITED) {
1338                 /*
1339                  * If the mountpoint is inherited we don't need to
1340                  * do anything.  When its parent gets processed
1341                  * its mountpoint will be set accordingly.
1342                  */
1343                 goto done;
1344         } else if (sourcetype & ZPROP_SRC_LOCAL) {

1346                 if (strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) == 0) {
1347                         /*
1348                          * If the mountpoint is set to 'legacy', its already
1349                          * been unmounted (from above call to zfs_unmount), and
1350                          * we don't need to do anything else with it.
1351                          */
1352                         goto done;

1354                 } else {
1355                         /*
1356                          * Else process dataset with explicitly set mountpoint.
1357                          */

1359                         /*
1360                          * Get this dataset's mountpoint relative to
1361                          * the BE's mountpoint.
1362                          */
1363                         if ((strncmp(mountpoint, ud->altroot,
1364                             strlen(ud->altroot)) == 0) &&
1365                             (mountpoint[strlen(ud->altroot)] == '/')) {

1367                                 zhp_mountpoint = mountpoint +
1368                                     strlen(ud->altroot);

1370                                 /* Set this dataset's mountpoint value */
1371                                 if (zfs_prop_set(zhp,
1372                                     zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
1373                                     zhp_mountpoint)) {
1374                                         be_print_err(
1375                                             gettext("be_unmount_callback: "
1376                                             "failed to set mountpoint for "
1377                                             "%s to %s: %s\n"), fs_name,
1378                                             zhp_mountpoint,
1379                                             libzfs_error_description(g_zfs));
1380                                         ret = zfs_err_to_be_err(g_zfs);
1381                                 }
```



```
1382                         } else {
1383                                 be_print_err(
1384                                     gettext("be_unmount_callback: "
1385                                     "%s not mounted under BE's altroot %s, "
1386                                     "skipping ...\n"), fs_name, ud->altroot);
1387                                 /*
1388                                  * fs_name is mounted but not under the
1389                                  * root for this BE.
1390                                  */
1391                                 ret = BE_ERR_INVALMOUNTPOINT;
1392                         }
1393                 }
1394         } else {
1395                 be_print_err(gettext("be_unmount_callback: "
1396                     "mountpoint sourcetype of %s is %d, skipping ...\n"),
1397                     fs_name, sourcetype);
1398                 ret = BE_ERR_ZFS;
1399         }

1401 done:
1402         /* Set this filesystem's 'canmount' property to 'noauto' */
1403         if (zfs_prop_set(zhp, zfs_prop_to_name(ZFS_PROP_CANMOUNT), "noauto")) {
1404                 be_print_err(gettext("be_unmount_callback: "
1405                     "failed to set canmount to 'noauto' (%s)\n"), fs_name);
1406                 if (ret == 0)
1407                         ret = BE_ERR_ZFS;
1408         }

1410         ZFS_CLOSE(zhp);
1411         return (ret);
1412 }

1414 /*
1415  * Function:    be_get_legacy_fs_callback
1416  * Description: The callback function is used to iterate through all
1417  *              non-shared file systems of a BE, finding ones that have
1418  *              a legacy mountpoint and an entry in the BE's vfstab.
1419  *              It adds these file systems to the callback data.
1420  * Parameters:
1421  *              zhp - zfs_handle_t pointer to current file system being
1422  *                    processed.
1423  *              data - be_fs_list_data_t pointer
1424  * Returns:
1425  *              0 - Success
1426  *              be_errno_t - Failure
1427  * Scope:
1428  *              Private
1429  */
1430 static int
1431 be_get_legacy_fs_callback(zfs_handle_t *zhp, void *data)
1432 {
1433         be_fs_list_data_t       *fld = data;
1434         const char              *fs_name = zfs_get_name(zhp);
1435         char                    zhp_mountpoint[MAXPATHLEN];
1436         char                    mountpoint[MAXPATHLEN];
1437         int                     ret = 0;

1439         /* Get this dataset's mountpoint property */
1440         if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, zhp_mountpoint,
1441             sizeof (zhp_mountpoint), NULL, NULL, 0, B_FALSE) != 0) {
1442                 be_print_err(gettext("be_get_legacy_fs_callback: "
1443                     "failed to get mountpoint for %s: %s\n"),
1444                     fs_name, libzfs_error_description(g_zfs));
1445                 ret = zfs_err_to_be_err(g_zfs);
1446                 ZFS_CLOSE(zhp);
1447                 return (ret);
```

```
1448         }
1449
1450         /*
1451          * If mountpoint is legacy, try to get its mountpoint from this BE's
1452          * vfstab.  If it exists in the vfstab, add this file system to the
1453          * callback data.
1454          */
1455         if (strcmp(zhp_mountpoint, ZFS_MOUNTPOINT_LEGACY) == 0) {
1456                 if (get_mountpoint_from_vfstab(fld->altroot, fs_name,
1457                     mountpoint, sizeof (mountpoint), B_FALSE) != BE_SUCCESS) {
1458                         be_print_err(gettext("be_get_legacy_fs_callback: "
1459                             "no entry for %s in vfstab, "
1460                             "skipping ...\n"), fs_name);
1461
1462                         goto next;
1463                 }
1464
1465                 /* Record file system into the callback data. */
1466                 if (add_to_fs_list(fld, zfs_get_name(zhp)) != BE_SUCCESS) {
1467                         be_print_err(gettext("be_get_legacy_fs_callback: "
1468                             "failed to add %s to fs list\n"), mountpoint);
1469                         ZFS_CLOSE(zhp);
1470                         return (BE_ERR_NOMEM);
1471                 }
1472         }
1473
1474 next:
1475         /* Iterate through this dataset's children file systems */
1476         if ((ret = zfs_iter_filesystems(zhp, be_get_legacy_fs_callback,
1477             fld)) != 0) {
1478                 ZFS_CLOSE(zhp);
1479                 return (ret);
1480         }
1481         ZFS_CLOSE(zhp);
1482         return (0);
1483 }
1484
1485 /*
1486  * Function:    add_to_fs_list
1487  * Description: Function used to add a file system to the fs_list array in
1488  *                      a be_fs_list_data_t structure.
1489  * Parameters:
1490  *              fld - be_fs_list_data_t pointer
1491  *              fs - file system to add
1492  * Returns:
1493  *              BE_SUCCESS - Success
1494  *              1 - Failure
1495  * Scope:
1496  *              Private
1497  */
1498 static int
1499 add_to_fs_list(be_fs_list_data_t *fld, const char *fs)
1500 {
1501         if (fld == NULL || fs == NULL)
1502                 return (1);
1503
1504         if ((fld->fs_list = (char **)realloc(fld->fs_list,
1505             sizeof (char *)*(fld->fs_num + 1))) == NULL) {
1506                 be_print_err(gettext("add_to_fs_list: "
1507                     "memory allocation failed\n"));
1508                 return (1);
1509         }
1510
1511         if ((fld->fs_list[fld->fs_num++] = strdup(fs)) == NULL) {
1512                 be_print_err(gettext("add_to_fs_list: "
1513                     "memory allocation failed\n"));
```



```
1514                 return (1);
1515         }
1516
1517         return (BE_SUCCESS);
1518 }
1519
1520 /*
1521  * Function:    zpool_shared_fs_callback
1522  * Description: Callback function used to iterate through all existing pools
1523  *              to find and mount all shared filesystems.  This function
1524  *              processes the pool's "pool data" dataset, then uses
1525  *              iter_shared_fs_callback to iterate through the pool's
1526  *              datasets.
1527  * Parameters:
1528  *              zlp - zpool_handle_t pointer to the current pool being
1529  *                    looked at.
1530  *              data - be_mount_data_t pointer
1531  * Returns:
1532  *              0 - Success
1533  *              be_errno_t - Failure
1534  * Scope:
1535  *              Private
1536  */
1537 static int
1538 zpool_shared_fs_callback(zpool_handle_t *zlp, void *data)
1539 {
1540         be_mount_data_t *md = data;
1541         zfs_handle_t    *zhp = NULL;
1542         const char      *zpool = zpool_get_name(zlp);
1543         int             ret = 0;
1544
1545         /*
1546          * Get handle to pool's "pool data" dataset
1547          */
1548         if ((zhp = zfs_open(g_zfs, zpool, ZFS_TYPE_FILESYSTEM)) == NULL) {
1549                 be_print_err(gettext("zpool_shared_fs: "
1550                     "failed to open pool dataset %s: %s\n"), zpool,
1551                     libzfs_error_description(g_zfs));
1552                 ret = zfs_err_to_be_err(g_zfs);
1553                 zpool_close(zlp);
1554                 return (ret);
1555         }
1556
1557         /* Process this pool's "pool data" dataset */
1558         (void) loopback_mount_shared_fs(zhp, md);
1559
1560         /* Interate through this pool's children */
1561         (void) zfs_iter_filesystems(zhp, iter_shared_fs_callback, md);
1562
1563         ZFS_CLOSE(zhp);
1564         zpool_close(zlp);
1565
1566         return (0);
1567 }
1568
1569 /*
1570  * Function:    iter_shared_fs_callback
1571  * Description: Callback function used to iterate through a pool's datasets
1572  *              to find and mount all shared filesystems.  It makes sure to
1573  *              find the BE container dataset of the pool, if it exists, and
1574  *              does not process and iterate down that path.
1575  *
1576  *              Note - This function iterates linearly down the
1577  *              hierarchical dataset paths and mounts things as it goes
1578  *              along.  It does not make sure that something deeper down
1579  *              a dataset path has an interim mountpoint for something
```

```
 *              processed earlier.
 *
 * Parameters:
 *              zhp - zfs_handle_t pointer to the current dataset being
 *                      processed.
 *              data - be_mount_data_t pointer
 * Returns:
 *              0 - Success
 *              be_errno_t - Failure
 * Scope:
 *              Private
 */
static int
iter_shared_fs_callback(zfs_handle_t *zhp, void *data)
{
        be_mount_data_t *md = data;
        const char      *name = zfs_get_name(zhp);
        char            container_ds[MAXPATHLEN];
        char            tmp_name[MAXPATHLEN];
        char            *pool;

        /* Get the pool's name */
        (void) strlcpy(tmp_name, name, sizeof (tmp_name));
        pool = strtok(tmp_name, "/");

        if (pool) {
                /* Get the name of this pool's container dataset */
                be_make_container_ds(pool, container_ds,
                    sizeof (container_ds));

                /*
                 * If what we're processing is this pool's BE container
                 * dataset, skip it.
                 */
                if (strcmp(name, container_ds) == 0) {
                        ZFS_CLOSE(zhp);
                        return (0);
                }
        } else {
                /* Getting the pool name failed, return error */
                be_print_err(gettext("iter_shared_fs_callback: "
                    "failed to get pool name from %s\n"), name);
                ZFS_CLOSE(zhp);
                return (BE_ERR_POOL_NOENT);
        }

        /* Mount this shared filesystem */
        (void) loopback_mount_shared_fs(zhp, md);

        /* Iterate this dataset's children file systems */
        (void) zfs_iter_filesystems(zhp, iter_shared_fs_callback, md);
        ZFS_CLOSE(zhp);

        return (0);
}

/*
 * Function:    loopback_mount_shared_fs
 * Description: This function loopback mounts a file system into the altroot
 *              area of the BE being mounted.  Since these are shared file
 *              systems, they are expected to be already mounted for the
 *              current BE, and this function just loopback mounts them into
 *              the BE mountpoint.  If they are not mounted for the current
 *              live system, they are skipped and not mounted into the BE
 *              we're mounting.
 * Parameters:
```



```
 *              zhp - zfs_handle_t pointer to the dataset to loopback mount
 *              md - be_mount_data_t pointer
 * Returns:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Private
 */
static int
loopback_mount_shared_fs(zfs_handle_t *zhp, be_mount_data_t *md)
{
        char            zhp_mountpoint[MAXPATHLEN];
        char            mountpoint[MAXPATHLEN];
        char            *mp = NULL;
        char            optstr[MAX_MNTOPT_STR];
        int             mflag = MS_OPTIONSTR;
        int             err;

        /*
         * Check if file system is currently mounted and not delegated
         * to a non-global zone (if we're in the global zone)
         */
        if (zfs_is_mounted(zhp, &mp) && (getzoneid() != GLOBAL_ZONEID ||
            !zfs_prop_get_int(zhp, ZFS_PROP_ZONED))) {
                /*
                 * If we didn't get a mountpoint from the zfs_is_mounted call,
                 * get it from the mountpoint property.
                 */
                if (mp == NULL) {
                        if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT,
                            zhp_mountpoint, sizeof (zhp_mountpoint), NULL,
                            NULL, 0, B_FALSE) != 0) {
                                be_print_err(
                                    gettext("loopback_mount_shared_fs: "
                                    "failed to get mountpoint property\n"));
                                return (BE_ERR_ZFS);
                        }
                } else {
                        (void) strlcpy(zhp_mountpoint, mp,
                            sizeof (zhp_mountpoint));
                        free(mp);
                }

                (void) snprintf(mountpoint, sizeof (mountpoint), "%s%s",
                    md->altroot, zhp_mountpoint);

                /* Mount it read-only if read-write was not requested */
                if (!md->shared_rw) {
                        mflag |= MS_RDONLY;
                }

                /* Add the "nosub" option to the mount options string */
                (void) strlcpy(optstr, MNTOPT_NOSUB, sizeof (optstr));

                /* Loopback mount this dataset at the altroot */
                if (mount(zhp_mountpoint, mountpoint, mflag, MNTTYPE_LOFS,
                    NULL, 0, optstr, sizeof (optstr)) != 0) {
                        err = errno;
                        be_print_err(gettext("loopback_mount_shared_fs: "
                            "failed to loopback mount %s at %s: %s\n"),
                            zhp_mountpoint, mountpoint, strerror(err));
                        return (BE_ERR_MOUNT);
                }
        }

        return (BE_SUCCESS);
```

```
}

/*
 * Function:    loopback_mount_zonepath
 * Description: This function loopback mounts a zonepath into the altroot
 *              area of the BE being mounted.
 * Parameters:
 *              zonepath - pointer to zone path in the current BE
 *              md - be_mount_data_t pointer
 * Returns:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Private
 */
static int
loopback_mount_zonepath(const char *zonepath, be_mount_data_t *md)
{
        FILE            *fp = (FILE *)NULL;
        struct stat     st;
        char            *p;
        char            *p1;
        char            *parent_dir;
        struct extmnttab        extmtab;
        dev_t           dev = NODEV;
        char            *parentmnt;
        char            alt_parentmnt[MAXPATHLEN];
        struct mnttab   mntref;
        char            altzonepath[MAXPATHLEN];
        char            optstr[MAX_MNTOPT_STR];
        int             mflag = MS_OPTIONSTR;
        int             ret;
        int             err;

        fp = fopen(MNTTAB, "r");
        if (fp == NULL) {
                err = errno;
                be_print_err(gettext("loopback_mount_zonepath: "
                    "failed to open /etc/mnttab\n"));
                return (errno_to_be_err(err));
        }

        /*
         * before attempting the loopback mount of zonepath under altroot,
         * we need to make sure that all intermediate file systems in the
         * zone path are also mounted under altroot
         */

        /* get the parent directory for zonepath */
        p = strrchr(zonepath, '/');
        if (p != NULL && p != zonepath) {
                if ((parent_dir = (char *)calloc(sizeof (char),
                    p - zonepath + 1)) == NULL) {
                        ret = BE_ERR_NOMEM;
                        goto done;
                }
                (void) strlcpy(parent_dir, zonepath, p - zonepath + 1);
                if (stat(parent_dir, &st) < 0) {
                        ret = errno_to_be_err(errno);
                        be_print_err(gettext("loopback_mount_zonepath: "
                            "failed to stat %s"),
                            parent_dir);
                        free(parent_dir);
                        goto done;
                }
                free(parent_dir);
```



```
                /*
                 * After the above stat call, st.st_dev contains ID of the
                 * device over which parent dir resides.
                 * Now, search mnttab and find mount point of parent dir device.
                 */

                resetmnttab(fp);
                while (getextmntent(fp, &extmtab, sizeof (extmtab)) == 0) {
                        dev = makedev(extmtab.mnt_major, extmtab.mnt_minor);
                        if (st.st_dev == dev && strcmp(extmtab.mnt_fstype,
                            MNTTYPE_ZFS) == 0) {
                                p1 = strchr(extmtab.mnt_special, '/');
                                if (p1 == NULL || strncmp(p1 + 1,
                                    BE_CONTAINER_DS_NAME, 4) != 0 ||
                                    (*(p1 + 5) != '/' && *(p1 + 5) != '\0')) {
                                        /*
                                         * if parent dir is in a shared file
                                         * system, check whether it is already
                                         * loopback mounted under altroot or
                                         * not.  It would have been mounted
                                         * already under altroot if it is in
                                         * a non-shared filesystem.
                                         */
                                        parentmnt = strdup(extmtab.mnt_mountp);
                                        (void) snprintf(alt_parentmnt,
                                            sizeof (alt_parentmnt), "%s%s",
                                            md->altroot, parentmnt);
                                        mntref.mnt_mountp = alt_parentmnt;
                                        mntref.mnt_special = parentmnt;
                                        mntref.mnt_fstype = MNTTYPE_LOFS;
                                        mntref.mnt_mntopts = NULL;
                                        mntref.mnt_time = NULL;
                                        resetmnttab(fp);
                                        if (getmntany(fp, (struct mnttab *)
                                            &extmtab, &mntref) != 0) {
                                                ret = loopback_mount_zonepath(
                                                    parentmnt, md);
                                                if (ret != BE_SUCCESS) {
                                                        free(parentmnt);
                                                        goto done;
                                                }
                                        }
                                        free(parentmnt);
                                }
                                break;
                        }
                }
        }


        if (!md->shared_rw) {
                mflag |= MS_RDONLY;
        }

        (void) snprintf(altzonepath, sizeof (altzonepath), "%s%s",
            md->altroot, zonepath);

        /* Add the "nosub" option to the mount options string */
        (void) strlcpy(optstr, MNTOPT_NOSUB, sizeof (optstr));

        /* Loopback mount this dataset at the altroot */
        if (mount(zonepath, altzonepath, mflag, MNTTYPE_LOFS,
            NULL, 0, optstr, sizeof (optstr)) != 0) {
                err = errno;
                be_print_err(gettext("loopback_mount_zonepath: "
```

```
1844                    "failed to loopback mount %s at %s: %s\n"),
1845                    zonepath, altzonepath, strerror(err));
1846                ret = BE_ERR_MOUNT;
1847                goto done;
1848        }
1849        ret = BE_SUCCESS;

1851 done :
1852        (void) fclose(fp);
1853        return (ret);
1854 }

1856 /*
1857  * Function:    unmount_shared_fs
1858  * Description: This function iterates through the mnttab and finds all
1859  *              loopback mount entries that reside within the altroot of
1860  *              where the BE is mounted, and unmounts it.
1861  * Parameters:
1862  *              ud - be_unmount_data_t pointer
1863  * Returns:
1864  *              BE_SUCCESS - Success
1865  *              be_errno_t - Failure
1866  * Scope:
1867  *              Private
1868  */
1869 static int
1870 unmount_shared_fs(be_unmount_data_t *ud)
1871 {
1872        FILE            *fp = NULL;
1873        struct mnttab   *table = NULL;
1874        struct mnttab   ent;
1875        struct mnttab   *entp = NULL;
1876        size_t          size = 0;
1877        int             read_chunk = 32;
1878        int             i;
1879        int             altroot_len;
1880        int             err = 0;

1882        errno = 0;

1884        /* Read in the mnttab into a table */
1885        if ((fp = fopen(MNTTAB, "r")) == NULL) {
1886                err = errno;
1887                be_print_err(gettext("unmount_shared_fs: "
1888                    "failed to open mnttab\n"));
1889                return (errno_to_be_err(err));
1890        }

1892        while (getmntent(fp, &ent) == 0) {
1893                if (size % read_chunk == 0) {
1894                        table = (struct mnttab *)realloc(table,
1895                            (size + read_chunk) * sizeof (ent));
1896                }
1897                entp = &table[size++];

1899                /*
1900                 * Copy over the current mnttab entry into our table,
1901                 * copying only the fields that we care about.
1902                 */
1903                (void) memset(entp, 0, sizeof (*entp));
1904                if ((entp->mnt_mountp = strdup(ent.mnt_mountp)) == NULL ||
1905                    (entp->mnt_fstype = strdup(ent.mnt_fstype)) == NULL) {
1906                        be_print_err(gettext("unmount_shared_fs: "
1907                            "memory allocation failed\n"));
1908                        return (BE_ERR_NOMEM);
1909                }
```



```
1910        }
1911        (void) fclose(fp);

1913        /*
1914         * Process the mnttab entries in reverse order, looking for
1915         * loopback mount entries mounted under our altroot.
1916         */
1917        altroot_len = strlen(ud->altroot);
1918        for (i = size; i > 0; i--) {
1919                entp = &table[i - 1];

1921                /* If not of type lofs, skip */
1922                if (strcmp(entp->mnt_fstype, MNTTYPE_LOFS) != 0)
1923                        continue;

1925                /* If inside the altroot, unmount it */
1926                if (strncmp(entp->mnt_mountp, ud->altroot, altroot_len) == 0 &&
1927                    entp->mnt_mountp[altroot_len] == '/') {
1928                        if (umount(entp->mnt_mountp) != 0) {
1929                                err = errno;
1930                                if (err == EBUSY) {
1931                                        (void) sleep(1);
1932                                        err = errno = 0;
1933                                        if (umount(entp->mnt_mountp) != 0)
1934                                                err = errno;
1935                                }
1936                                if (err != 0) {
1937                                        be_print_err(gettext(
1938                                            "unmount_shared_fs: "
1939                                            "failed to unmount shared file "
1940                                            "system %s: %s\n"),
1941                                            entp->mnt_mountp, strerror(err));
1942                                        return (errno_to_be_err(err));
1943                                }
1944                        }
1945                }
1946        }

1948        return (BE_SUCCESS);
1949 }

1951 /*
1952  * Function:    get_mountpoint_from_vfstab
1953  * Description: This function digs into the vfstab in the given altroot,
1954  *              and searches for an entry for the fs passed in.  If found,
1955  *              it returns the mountpoint of that fs in the mountpoint
1956  *              buffer passed in.  If the get_alt_mountpoint flag is set,
1957  *              it returns the mountpoint with the altroot prepended.
1958  * Parameters:
1959  *              altroot - pointer to the alternate root location
1960  *              fs - pointer to the file system name to look for in the
1961  *                      vfstab in altroot
1962  *              mountpoint - pointer to buffer of where the mountpoint of
1963  *                      fs will be returned.
1964  *              size_mp - size of mountpoint argument
1965  *              get_alt_mountpoint - flag to indicate whether or not the
1966  *                      mountpoint should be populated with the altroot
1967  *                      prepended.
1968  * Returns:
1969  *              BE_SUCCESS - Success
1970  *              1 - Failure
1971  * Scope:
1972  *              Private
1973  */
1974 static int
1975 get_mountpoint_from_vfstab(char *altroot, const char *fs, char *mountpoint,
```

```
1976     size_t size_mp, boolean_t get_alt_mountpoint)
1977 {
1978         struct vfstab   vp;
1979         FILE            *fp = NULL;
1980         char            alt_vfstab[MAXPATHLEN];

1982         /* Generate path to alternate root vfstab */
1983         (void) snprintf(alt_vfstab, sizeof (alt_vfstab), "%s/etc/vfstab",
1984             altroot);

1986         /* Open alternate root vfstab */
1987         if ((fp = fopen(alt_vfstab, "r")) == NULL) {
1988                 be_print_err(gettext("get_mountpoint_from_vfstab: "
1989                     "failed to open vfstab (%s)\n"), alt_vfstab);
1990                 return (1);
1991         }

1993         if (getvfsspec(fp, &vp, (char *)fs) == 0) {
1994                 /*
1995                  * Found entry for fs, grab its mountpoint.
1996                  * If the flag to prepend the altroot into the mountpoint
1997                  * is set, prepend it.  Otherwise, just return the mountpoint.
1998                  */
1999                 if (get_alt_mountpoint) {
2000                         (void) snprintf(mountpoint, size_mp, "%s%s", altroot,
2001                             vp.vfs_mountp);
2002                 } else {
2003                         (void) strlcpy(mountpoint, vp.vfs_mountp, size_mp);
2004                 }
2005         } else {
2006                 (void) fclose(fp);
2007                 return (1);
2008         }

2010         (void) fclose(fp);

2012         return (BE_SUCCESS);
2013 }

2015 /*
2016  * Function:    fix_mountpoint_callback
2017  * Description: This callback function is used to iterate through a BE's
2018  *              children filesystems to check if its mountpoint is currently
2019  *              set to be mounted at some specified altroot.  If so, fix it by
2020  *              removing altroot from the beginning of its mountpoint.
2021  *
2022  *              Note - There's no way to tell if a child filesystem's
2023  *              mountpoint isn't broken, and just happens to begin with
2024  *              the altroot we're looking for.  In this case, this function
2025  *              will errantly remove the altroot portion from the beginning
2026  *              of this filesystem's mountpoint.
2027  *
2028  * Parameters:
2029  *              zhp - zfs_handle_t pointer to filesystem being processed.
2030  *              data - altroot of where BE is to be mounted.
2031  * Returns:
2032  *              0 - Success
2033  *              be_errno_t - Failure
2034  * Scope:
2035  *              Private
2036  */
2037 static int
2038 fix_mountpoint_callback(zfs_handle_t *zhp, void *data)
2039 {
2040         zprop_source_t  sourcetype;
2041         char            source[ZFS_MAXNAMELEN];
```



```
2042         char            mountpoint[MAXPATHLEN];
2043         char            *zhp_mountpoint = NULL;
2044         char            *altroot = data;
2045         int             ret = 0;

2047         /* Get dataset's mountpoint and source values */
2048         if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
2049             sizeof (mountpoint), &sourcetype, source, sizeof (source),
2050             B_FALSE) != 0) {
2051                 be_print_err(gettext("fix_mountpoint_callback: "
2052                     "failed to get mountpoint and sourcetype for %s\n"),
2053                     zfs_get_name(zhp));
2054                 ZFS_CLOSE(zhp);
2055                 return (BE_ERR_ZFS);
2056         }

2058         /*
2059          * If the mountpoint is not inherited and the mountpoint is not
2060          * 'legacy', this file system potentially needs its mountpoint
2061          * fixed.
2062          */
2063         if (!(sourcetype & ZPROP_SRC_INHERITED) &&
2064             strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) != 0) {

2066                 /*
2067                  * Check if this file system's current mountpoint is
2068                  * under the altroot we're fixing it against.
2069                  */
2070                 if (strncmp(mountpoint, altroot, strlen(altroot)) == 0 &&
2071                     mountpoint[strlen(altroot)] == '/') {

2073                         /*
2074                          * Get this dataset's mountpoint relative to the
2075                          * altroot.
2076                          */
2077                         zhp_mountpoint = mountpoint + strlen(altroot);

2079                         /* Fix this dataset's mountpoint value */
2080                         if (zfs_prop_set(zhp,
2081                             zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
2082                             zhp_mountpoint)) {
2083                                 be_print_err(gettext("fix_mountpoint_callback: "
2084                                     "failed to set mountpoint for %s to "
2085                                     "%s: %s\n"), zfs_get_name(zhp),
2086                                     zhp_mountpoint,
2087                                     libzfs_error_description(g_zfs));
2088                                 ret = zfs_err_to_be_err(g_zfs);
2089                                 ZFS_CLOSE(zhp);
2090                                 return (ret);
2091                         }
2092                 }
2093         }

2095         /* Iterate through this dataset's children and fix them */
2096         if ((ret = zfs_iter_filesystems(zhp, fix_mountpoint_callback,
2097             altroot)) != 0) {
2098                 ZFS_CLOSE(zhp);
2099                 return (ret);
2100         }


2103         ZFS_CLOSE(zhp);
2104         return (0);
2105 }

2107 /*
```

```
 * Function:    be_mount_root
 * Description: This function mounts the root dataset of a BE at the
 *              specified altroot.
 * Parameters:
 *              zhp - zfs_handle_t pointer to root dataset of a BE that is
 *              to be mounted at altroot.
 *              altroot - location of where to mount the BE root.
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Private
 */
static int
be_mount_root(zfs_handle_t *zhp, char *altroot)
{
        char            mountpoint[MAXPATHLEN];

        /* 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_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));
        }

        /*
         * Set the canmount property for the BE's root dataset to 'noauto' just
         * in case it's been set to 'on'.  We do this so that when we change its
         * mountpoint, zfs won't immediately try to mount it.
         */
        if (zfs_prop_set(zhp, zfs_prop_to_name(ZFS_PROP_CANMOUNT), "noauto")
            != 0) {
                be_print_err(gettext("be_mount_root: failed to "
                    "set canmount property to 'noauto' (%s): %s\n"),
                    zfs_get_name(zhp), libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /* Set mountpoint for BE's root filesystem */
        if (zfs_prop_set(zhp, zfs_prop_to_name(ZFS_PROP_MOUNTPOINT), altroot)
            != 0) {
                be_print_err(gettext("be_mount_root: failed to "
                    "set mountpoint of %s to %s: %s\n"),
                    zfs_get_name(zhp), altroot,
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /* Mount the BE's root filesystem */
        if (zfs_mount(zhp, NULL, 0) != 0) {
                be_print_err(gettext("be_mount_root: failed to "
                    "mount dataset %s at %s: %s\n"), zfs_get_name(zhp),
                    altroot, libzfs_error_description(g_zfs));
                /*
                 * Set this BE's root filesystem 'mountpoint' property
                 * back to what it was before.
                 */
                (void) zfs_prop_set(zhp, zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
                    mountpoint);
                return (zfs_err_to_be_err(g_zfs));
        }

        return (BE_SUCCESS);
}
```



```
/*
 * Function:    be_unmount_root
 * Description: This function unmounts the root dataset of a BE, but before
 *              unmounting, it looks at the BE's vfstab to determine
 *              if the root dataset mountpoint should be left as 'legacy'
 *              or '/'.  If the vfstab contains an entry for this root
 *              dataset with a mountpoint of '/', it sets the mountpoint
 *              property to 'legacy'.
 *
 * Parameters:
 *              zhp - zfs_handle_t pointer of the BE root dataset that
 *              is currently mounted.
 *              ud - be_unmount_data_t pointer providing unmount data
 *              for the given BE root dataset.
 * Returns:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Private
 */
static int
be_unmount_root(zfs_handle_t *zhp, be_unmount_data_t *ud)
{
        char            mountpoint[MAXPATHLEN];
        boolean_t       is_legacy = B_FALSE;

        /* See if this is a legacy mounted root */
        if (get_mountpoint_from_vfstab(ud->altroot, zfs_get_name(zhp),
            mountpoint, sizeof (mountpoint), B_FALSE) == BE_SUCCESS &&
            strcmp(mountpoint, "/") == 0) {
                is_legacy = B_TRUE;
        }

        /* Unmount the dataset */
        if (zfs_unmount(zhp, NULL, ud->force ? MS_FORCE : 0) != 0) {
                be_print_err(gettext("be_unmount_root: failed to "
                    "unmount BE root dataset %s: %s\n"), zfs_get_name(zhp),
                    libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /* Set canmount property for this BE's root filesystem to noauto */
        if (zfs_prop_set(zhp, zfs_prop_to_name(ZFS_PROP_CANMOUNT), "noauto")
            != 0) {
                be_print_err(gettext("be_unmount_root: failed to "
                    "set canmount property for %s to 'noauto': %s\n"),
                    zfs_get_name(zhp), libzfs_error_description(g_zfs));
                return (zfs_err_to_be_err(g_zfs));
        }

        /*
         * Set mountpoint for BE's root dataset back to '/', or 'legacy'
         * if its a legacy mounted root.
         */
        if (zfs_prop_set(zhp, zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
            is_legacy ? ZFS_MOUNTPOINT_LEGACY : "/") != 0) {
                be_print_err(gettext("be_unmount_root: failed to "
                    "set mountpoint of %s to %s\n"), zfs_get_name(zhp),
                    is_legacy ? ZFS_MOUNTPOINT_LEGACY : "/");
                return (zfs_err_to_be_err(g_zfs));
        }

        return (BE_SUCCESS);
}
```

```
2240 /*
2241  * Function:    fix_mountpoint
2242  * Description: This function checks the mountpoint of an unmounted BE to make
2243  *              sure that it is set to either 'legacy' or '/'.  If it's not,
2244  *              then we're in a situation where an unmounted BE has some random
2245  *              mountpoint set for it.  (This could happen if the system was
2246  *              rebooted while an inactive BE was mounted).  This function
2247  *              attempts to fix its mountpoints.
2248  * Parameters:
2249  *              zhp - zfs_handle_t pointer to root dataset of the BE
2250  *              whose mountpoint needs to be checked.
2251  * Return:
2252  *              BE_SUCCESS - Success
2253  *              be_errno_t - Failure
2254  * Scope:
2255  *              Private
2256  */
2257 static int
2258 fix_mountpoint(zfs_handle_t *zhp)
2259 {
2260         be_unmount_data_t       ud = { 0 };
2261         char    *altroot = NULL;
2262         char    mountpoint[MAXPATHLEN];
2263         int     ret = BE_SUCCESS;

2265         /*
2266          * Record what this BE's root dataset mountpoint property is currently
2267          * set to.
2268          */
2269         if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
2270             sizeof (mountpoint), NULL, NULL, 0, B_FALSE) != 0) {
2271                 be_print_err(gettext("fix_mountpoint: failed to get "
2272                     "mountpoint property of (%s): %s\n"), zfs_get_name(zhp),
2273                     libzfs_error_description(g_zfs));
2274                 return (BE_ERR_ZFS);
2275         }

2277         /*
2278          * If the root dataset mountpoint is set to 'legacy' or '/', we're okay.
2279          */
2280         if (strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) == 0 ||
2281             strcmp(mountpoint, "/") == 0) {
2282                 return (BE_SUCCESS);
2283         }

2285         /*
2286          * Iterate through this BE's children datasets and fix
2287          * them if they need fixing.
2288          */
2289         if (zfs_iter_filesystems(zhp, fix_mountpoint_callback, mountpoint)
2290             != 0) {
2291                 return (BE_ERR_ZFS);
2292         }

2294         /*
2295          * The process of mounting and unmounting the root file system
2296          * will fix its mountpoint to correctly be either 'legacy' or '/'
2297          * since be_unmount_root will do the right thing by looking at
2298          * its vfstab.
2299          */

2301         /* Generate temporary altroot to mount the root file system */
2302         if ((ret = be_make_tmp_mountpoint(&altroot)) != BE_SUCCESS) {
2303                 be_print_err(gettext("fix_mountpoint: failed to "
2304                     "make temporary mountpoint\n"));
2305                 return (ret);
```



```
2306         }

2308         /* Mount and unmount the root. */
2309         if ((ret = be_mount_root(zhp, altroot)) != BE_SUCCESS) {
2310                 be_print_err(gettext("fix_mountpoint: failed to "
2311                     "mount BE root file system\n"));
2312                 goto cleanup;
2313         }
2314         ud.altroot = altroot;
2315         if ((ret = be_unmount_root(zhp, &ud)) != BE_SUCCESS) {
2316                 be_print_err(gettext("fix_mountpoint: failed to "
2317                     "unmount BE root file system\n"));
2318                 goto cleanup;
2319         }

2321 cleanup:
2322         free(altroot);

2324         return (ret);
2325 }

2327 /*
2328  * Function:    be_mount_zones
2329  * Description: This function finds all supported non-global zones in the
2330  *              given global BE and mounts them with respect to where the
2331  *              global BE is currently mounted.  The global BE datasets
2332  *              (including its shared datasets) are expected to already
2333  *              be mounted.
2334  * Parameters:
2335  *              be_zhp - zfs_handle_t pointer to the root dataset of the
2336  *                       global BE.
2337  *              md - be_mount_data_t pointer to data for global BE.
2338  * Returns:
2339  *              BE_SUCCESS - Success
2340  *              be_errno_t - Failure
2341  * Scope:
2342  *              Private
2343  */
2344 static int
2345 be_mount_zones(zfs_handle_t *be_zhp, be_mount_data_t *md)
2346 {
  28         zoneBrandList_t *brands = NULL;
2347         zoneList_t      zlst = NULL;
2348         char            *zonename = NULL;
2349         char            *zonepath = NULL;
2350         char            *zonepath_ds = NULL;
2351         int             k;
2352         int             ret = BE_SUCCESS;
2353         boolean_t       auto_create;
2354 #endif /* ! codereview */

2356         z_set_zone_root(md->altroot);

2358         zlst = z_get_nonglobal_branded_zone_list();
2359         if (zlst == NULL)
  35         if ((brands = be_get_supported_brandlist()) == NULL) {
  36                 be_print_err(gettext("be_mount_zones: "
  37                     "no supported brands\n"));
2360                 return (BE_SUCCESS);

2362         for (k = 0; (zonename = z_zlist_get_zonename(zlst, k)) != NULL; k++) {
2363                 if (z_zlist_is_zone_auto_create_be(zlst, k, &auto_create) != 0)
2364                         be_print_err(gettext("be_mount_zones: failed to"
2365                             " get auto-create-be brand property\n"));
2366                         goto done;
2367 #endif /* ! codereview */
```

```
2368                 }
2370                 if (!auto_create)
2371                         continue;
  39         zlst = z_get_nonglobal_zone_list_by_brand(brands);
  40         if (zlst == NULL) {
  41                 z_free_brand_list(brands);
  42                 return (BE_SUCCESS);
  43         }
  45         for (k = 0; (zonename = z_zlist_get_zonename(zlst, k)) != NULL; k++) {
2373                 if (z_zlist_get_current_state(zlst, k) ==
2374                     ZONE_STATE_INSTALLED) {
2375                         zonepath = z_zlist_get_zonepath(zlst, k);
2377                         /*
2378                          * Get the dataset of this zonepath in current BE.
2379                          * If its not a dataset, skip it.
2380                          */
2381                         if ((zonepath_ds = be_get_ds_from_dir(zonepath))
2382                             == NULL)
2383                                 continue;
2385                         /*
2386                          * Check if this zone is supported based on
2387                          * the dataset of its zonepath
2388                          */
2389                         if (!be_zone_supported(zonepath_ds)) {
2390                                 free(zonepath_ds);
2391                                 zonepath_ds = NULL;
2392                                 continue;
2393                         }
2395                         /*
2396                          * if BE's shared file systems are already mounted,
2397                          * zone path dataset would have already been lofs
2398                          * mounted under altroot. Otherwise, we need to do
2399                          * it here.
2400                          */
2401                         if (!md->shared_fs) {
2402                                 ret = loopback_mount_zonepath(zonepath, md);
2403                                 if (ret != BE_SUCCESS)
2404                                         goto done;
2405                         }
2408                         /* Mount this zone */
2409                         ret = be_mount_one_zone(be_zhp, md, zonename,
2410                             zonepath, zonepath_ds);
2412                         free(zonepath_ds);
2413                         zonepath_ds = NULL;
2415                         if (ret != BE_SUCCESS) {
2416                                 be_print_err(gettext("be_mount_zones: "
2417                                     "failed to mount zone %s under "
2418                                     "altroot %s\n"), zonename, md->altroot);
2419                                 goto done;
2420                         }
2421                 }
2422         }
2424 done:
  98         z_free_brand_list(brands);
2425         z_free_zone_list(zlst);
2426         /*
```



```
2427          * libinstzones caches mnttab and uses cached version for resolving lofs
2428          * mounts when we call z_resolve_lofs. It creates the cached version
2429          * when the first call to z_resolve_lofs happens. So, library's cached
2430          * mnttab doesn't contain entries for lofs mounts created in the above
2431          * loop. Because of this, subsequent calls to z_resolve_lofs would fail
2432          * to resolve these lofs mounts. So, here we destroy library's cached
2433          * mnttab to force its recreation when the next call to z_resolve_lofs
2434          * happens.
2435          */
2436         z_destroyMountTable();
2437         return (ret);
2438 }
2440 /*
2441  * Function:    be_unmount_zones
2442  * Description: This function finds all supported non-global zones in the
2443  *              given mounted global BE and unmounts them.
2444  * Parameters:
2445  *              ud - unmount_data_t pointer data for the global BE.
2446  * Returns:
2447  *              BE_SUCCESS - Success
2448  *              be_errno_t - Failure
2449  * Scope:
2450  *              Private
2451  */
2452 static int
2453 be_unmount_zones(be_unmount_data_t *ud)
2454 {
  129         zoneBrandList_t         *brands = NULL;
2455         zoneList_t              zlst = NULL;
2456         char                    *zonename = NULL;
2457         char                    *zonepath = NULL;
2458         char                    alt_zonepath[MAXPATHLEN];
2459         char                    *zonepath_ds = NULL;
2460         int                     k;
2461         int                     ret = BE_SUCCESS;
2462         boolean_t               auto_create;
2463 #endif /* ! codereview */
2465         z_set_zone_root(ud->altroot);
2467         zlst = z_get_nonglobal_branded_zone_list();
2468         if (zlst == NULL)
  137         if ((brands = be_get_supported_brandlist()) == NULL) {
  138                 be_print_err(gettext("be_unmount_zones: "
  139                     "no supported brands\n"));
2469                 return (BE_SUCCESS);
2471         for (k = 0; (zonename = z_zlist_get_zonename(zlst, k)) != NULL; k++) {
2472                 if (z_zlist_is_zone_auto_create_be(zlst, k, &auto_create) != 0)
2473                         be_print_err(gettext("be_unmount_zones: failed to"
2474                             " get auto-create-be brand property\n"));
2475                         goto done;
2476 #endif /* ! codereview */
2477                 }
2479                 if (!auto_create)
2480                         continue;
  141         zlst = z_get_nonglobal_zone_list_by_brand(brands);
  142         if (zlst == NULL) {
  143                 z_free_brand_list(brands);
  144                 return (BE_SUCCESS);
  145         }
  147         for (k = 0; (zonename = z_zlist_get_zonename(zlst, k)) != NULL; k++) {
2482                 if (z_zlist_get_current_state(zlst, k) ==
```

```
2483                     ZONE_STATE_INSTALLED) {
2484                         zonepath = z_zlist_get_zonepath(zlst, k);

2486                         /* Build zone's zonepath wrt the global BE altroot */
2487                         (void) snprintf(alt_zonepath, sizeof (alt_zonepath),
2488                             "%s%s", ud->altroot, zonepath);

2490                         /*
2491                          * Get the dataset of this zonepath.  If its not
2492                          * a dataset, skip it.
2493                          */
2494                         if ((zonepath_ds = be_get_ds_from_dir(alt_zonepath))
2495                             == NULL)
2496                                 continue;

2498                         /*
2499                          * Check if this zone is supported based on the
2500                          * dataset of its zonepath.
2501                          */
2502                         if (!be_zone_supported(zonepath_ds)) {
2503                                 free(zonepath_ds);
2504                                 zonepath_ds = NULL;
2505                                 continue;
2506                         }

2508                         /* Unmount this zone */
2509                         ret = be_unmount_one_zone(ud, zonename, zonepath,
2510                             zonepath_ds);

2512                         free(zonepath_ds);
2513                         zonepath_ds = NULL;

2515                         if (ret != BE_SUCCESS) {
2516                                 be_print_err(gettext("be_unmount_zones:"
2517                                     " failed to unmount zone %s from "
2518                                     "altroot %s\n"), zonename, ud->altroot);
2519                                 goto done;
2520                         }
2521                 }
2522         }

2524 done:
 191         z_free_brand_list(brands);
2525         z_free_zone_list(zlst);
2526         return (ret);
2527 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   18003 Wed Nov 11 10:43:15 2015
new/usr/src/lib/libbe/common/be_zones.c
patch zone-auto-create-be
**********************************************************
```

```
/*
 * 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 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

/*
 * 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;

typedef struct mounted_zone_root_data {
        char    *zone_altroot;
        char    *zoneroot_ds;
```



```
} mounted_zone_root_data_t;

/* Private function prototypes */
static int be_find_active_zone_root_callback(zfs_handle_t *, void *);
static int be_find_mounted_zone_root_callback(zfs_handle_t *, void *);
static boolean_t be_zone_get_active(zfs_handle_t *);


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

/*
 * Function:    be_make_zoneroot
 * Description: Generate a string for a zone's zoneroot given the
 *              zone's zonepath.
 * Parameters:
 *              zonepath - pointer to zonepath
 *              zoneroot - pointer to buffer to retrn zoneroot in.
 *              zoneroot_size - size of zoneroot
 * Returns:
 *              None
 * Scope:
 *              Semi-private (library wise use only)
 */
void
be_make_zoneroot(char *zonepath, char *zoneroot, int zoneroot_size)
{
        (void) snprintf(zoneroot, zoneroot_size, "%s/root", zonepath);
}

/*
 * 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));
```

```
128                         return (ret);
129                 }
130         } else {
131                 if ((ret = be_zone_get_parent_uuid(zfs_get_name(be_zhp),
132                     &azr_data.parent_uuid)) != BE_SUCCESS) {
133                         be_print_err(gettext("be_find_active_zone_root: failed "
134                             "to get parentbe uuid for zone root dataset %s\n"),
135                             zfs_get_name(be_zhp));
136                         return (ret);
137                 }
138         }

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

144         /* Get handle to this zone's root container dataset */
145         if ((zhp = zfs_open(g_zfs, zone_container_ds, ZFS_TYPE_FILESYSTEM))
146             == NULL) {
147                 be_print_err(gettext("be_find_active_zone_root: failed to "
148                     "open zone root container dataset (%s): %s\n"),
149                     zone_container_ds, libzfs_error_description(g_zfs));
150                 return (zfs_err_to_be_err(g_zfs));
151         }

153         /*
154          * Iterate through all of this zone's BEs, looking for ones
155          * that belong to the parent global BE, and finding the one
156          * that is marked active.
157          */
158         if ((ret = zfs_iter_filesystems(zhp, be_find_active_zone_root_callback,
159             &azr_data)) != 0) {
160                 be_print_err(gettext("be_find_active_zone_root: failed to "
161                     "find active zone root in zonepath dataset %s: %s\n"),
162                     zonepath_ds, be_err_to_str(ret));
163                 goto done;
164         }

166         if (azr_data.zoneroot_ds != NULL) {
167                 (void) strlcpy(zoneroot_ds, azr_data.zoneroot_ds,
168                     zoneroot_ds_size);
169                 free(azr_data.zoneroot_ds);
170         } else {
171                 be_print_err(gettext("be_find_active_zone_root: failed to "
172                     "find active zone root in zonepath dataset %s\n"),
173                     zonepath_ds);
174                 ret = BE_ERR_ZONE_NO_ACTIVE_ROOT;
175         }

177 done:
178         ZFS_CLOSE(zhp);
179         return (ret);
180 }

182 /*
183  * Function:    be_find_mounted_zone_root
184  * Description: This function will find the dataset mounted as the zoneroot
185  *              of a zone for a given mounted global BE.
186  * Parameters:
187  *              zone_altroot - path of zoneroot wrt the mounted global BE.
188  *              zonepath_ds - dataset of the zone's zonepath.
189  *              zoneroot_ds - pointer to a buffer to store the dataset of
190  *                      the zoneroot that currently mounted for this zone
191  *                      in the mounted global BE.
192  *              zoneroot_ds_size - size of zoneroot_ds
193  * Returns:
```



```
194  *              BE_SUCCESS - Success
195  *              be_errno_t - Failure
196  * Scope:
197  *              Semi-private (library wide use only)
198  */
199 int
200 be_find_mounted_zone_root(char *zone_altroot, char *zonepath_ds,
201     char *zoneroot_ds, int zoneroot_ds_size)
202 {
203         mounted_zone_root_data_t        mzr_data = { 0 };
204         zfs_handle_t    *zhp = NULL;
205         char            zone_container_ds[MAXPATHLEN];
206         int             ret = BE_SUCCESS;
207         int             zret = 0;

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

213         /* Get handle to this zone's root container dataset. */
214         if ((zhp = zfs_open(g_zfs, zone_container_ds, ZFS_TYPE_FILESYSTEM))
215             == NULL) {
216                 be_print_err(gettext("be_find_mounted_zone_root: failed to "
217                     "open zone root container dataset (%s): %s\n"),
218                     zone_container_ds, libzfs_error_description(g_zfs));
219                 return (zfs_err_to_be_err(g_zfs));
220         }

222         mzr_data.zone_altroot = zone_altroot;

224         /*
225          * Iterate through all of the zone's BEs, looking for the one
226          * that is currently mounted at the zone altroot in the mounted
227          * global BE.
228          */
229         if ((zret = zfs_iter_filesystems(zhp,
230             be_find_mounted_zone_root_callback, &mzr_data)) == 0) {
231                 be_print_err(gettext("be_find_mounted_zone_root: did not "
232                     "find mounted zone under altroot zonepath %s\n"),
233                     zonepath_ds);
234                 ret = BE_ERR_NO_MOUNTED_ZONE;
235                 goto done;
236         } else if (zret < 0) {
237                 be_print_err(gettext("be_find_mounted_zone_root: "
238                     "zfs_iter_filesystems failed: %s\n"),
239                     libzfs_error_description(g_zfs));
240                 ret = zfs_err_to_be_err(g_zfs);
241                 goto done;
242         }

244         if (mzr_data.zoneroot_ds != NULL) {
245                 (void) strlcpy(zoneroot_ds, mzr_data.zoneroot_ds,
246                     zoneroot_ds_size);
247                 free(mzr_data.zoneroot_ds);
248         }

250 done:
251         ZFS_CLOSE(zhp);
252         return (ret);
253 }

255 /*
256  * Function:    be_zone_supported
257  * Description: This function will determine if a zone is supported
258  *              based on its zonepath dataset.  The zonepath dataset
259  *              must:
```

```
 *                 - not be under any global BE root dataset.
 *                 - have a root container dataset underneath it.
 *
 * Parameters:
 *              zonepath_ds - name of dataset of the zonepath of the
 *              zone to check.
 * Returns:
 *              B_TRUE - zone is supported
 *              B_FALSE - zone is not supported
 * Scope:
 *              Semi-private (library wide use only)
 */
boolean_t
be_zone_supported(char *zonepath_ds)
{
        char    zone_container_ds[MAXPATHLEN];
        int     ret = 0;

        /*
         * Make sure the dataset for the zonepath is not hierarchically
         * under any reserved BE root container dataset of any pool.
         */
        if ((ret = zpool_iter(g_zfs, be_check_be_roots_callback,
            zonepath_ds)) > 0) {
                be_print_err(gettext("be_zone_supported: "
                    "zonepath dataset %s not supported\n"), zonepath_ds);
                return (B_FALSE);
        } else if (ret < 0) {
                be_print_err(gettext("be_zone_supported: "
                "zpool_iter failed: %s\n"),
                    libzfs_error_description(g_zfs));
                return (B_FALSE);
        }

        /*
         * Make sure the zonepath has a zone root container dataset
         * underneath it.
         */
        be_make_container_ds(zonepath_ds, zone_container_ds,
            sizeof (zone_container_ds));

        if (!zfs_dataset_exists(g_zfs, zone_container_ds,
            ZFS_TYPE_FILESYSTEM)) {
                be_print_err(gettext("be_zone_supported: "
                    "zonepath dataset (%s) does not have a zone root container "
                    "dataset, zone is not supported, skipping ...\n"),
                    zonepath_ds);
                return (B_FALSE);
        }

        return (B_TRUE);
}

/*
 * Function:    be_get_supported_brandlist
 * Desciption:  This functions retuns a list of supported brands in
 *              a zoneBrandList_t object.
 * Parameters:
 *              None
 * Returns:
 *              Failure - NULL if no supported brands found.
 *              Success - pointer to zoneBrandList structure.
 * Scope:
 *              Semi-private (library wide use only)
 */
zoneBrandList_t *
```



```
be_get_supported_brandlist(void)
{
        return (z_make_brand_list(BE_ZONE_SUPPORTED_BRANDS,
            BE_ZONE_SUPPORTED_BRANDS_DELIM));
}
```

_unchanged_portion_omitted_

```
**********************************************************
    7704 Wed Nov 11 10:43:15 2015
new/usr/src/lib/libbe/common/libbe_priv.h
patch zone-auto-create-be
**********************************************************
```

```
/*
 * 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 2015 Toomas Soome <tsoome@me.com>
 */

#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_GRUB_STAGE_1         "/boot/grub/stage1"
#define BE_GRUB_STAGE_2         "/boot/grub/stage2"
#define BE_INSTALL_BOOT         "/usr/sbin/installboot"
#define BE_SPARC_BOOTBLK        "/lib/fs/zfs/bootblk"

#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 *);
```

```
169 /* be_utils.c */
170 boolean_t be_zfs_init(void);
171 void be_zfs_fini(void);
172 void be_make_root_ds(const char *, const char *, char *, int);
173 void be_make_container_ds(const char *, char *, int);
174 char *be_make_name_from_ds(const char *, char *);
175 int be_append_menu(char *, char *, char *, char *, char *);
176 int be_remove_menu(char *, char *, char *);
177 int be_update_menu(char *, char *, char *, char *);
178 int be_default_grub_bootfs(const char *, char **);
179 boolean_t be_has_menu_entry(char *, char *, int *);
180 int be_run_cmd(char *, char *, int, char *, int);
181 int be_change_grub_default(char *, char *);
182 int be_update_vfstab(char *, char *, char *, be_fs_list_data_t *, char *);
183 int be_update_zone_vfstab(zfs_handle_t *, char *, char *, char *,
184     be_fs_list_data_t *);
185 int be_maxsize_avail(zfs_handle_t *, uint64_t *);
186 char *be_auto_snap_name(void);
187 char *be_auto_be_name(char *);
188 char *be_auto_zone_be_name(char *, char *);
189 char *be_default_policy(void);
190 boolean_t valid_be_policy(char *);
191 boolean_t be_valid_auto_snap_name(char *);
192 boolean_t be_valid_be_name(const char *);
193 void be_print_err(char *, ...);
194 int be_find_current_be(be_transaction_data_t *);
195 int zfs_err_to_be_err(libzfs_handle_t *);
196 int errno_to_be_err(int);

198 /* be_activate.c */
199 int _be_activate(char *);
200 int be_activate_current_be(void);
201 boolean_t be_is_active_on_boot(char *);

203 /* be_zones.c */
204 void be_make_zoneroot(char *, char *, int);
205 int be_find_active_zone_root(zfs_handle_t *, char *, char *, int);
206 int be_find_mounted_zone_root(char *, char *, char *, int);
207 boolean_t be_zone_supported(char *);
210 zoneBrandList_t *be_get_supported_brandlist(void);
208 int be_zone_get_parent_uuid(const char *, uuid_t *);
209 int be_zone_set_parent_uuid(char *, uuid_t);
210 boolean_t be_zone_compare_uuids(char *);

212 /* check architecture functions */
213 char *be_get_default_isa(void);
214 char *be_get_platform(void);
215 boolean_t be_is_isa(char *);
216 boolean_t be_has_grub(void);

218 /* callback functions */
219 int be_exists_callback(zpool_handle_t *, void *);
220 int be_find_zpool_callback(zpool_handle_t *, void *);
221 int be_zpool_find_current_be_callback(zpool_handle_t *, void *);
222 int be_zfs_find_current_be_callback(zfs_handle_t *, void *);
223 int be_check_be_roots_callback(zpool_handle_t *, void *);

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

228 #ifdef __cplusplus
229 }
```

unchanged_portion_omitted_

```
**********************************************************
   28681 Wed Nov 11 10:43:15 2015
new/usr/src/lib/libbrand/common/libbrand.c
patch zone-auto-create-be
**********************************************************
```

```
/*
 * 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) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011, Joyent, Inc. All rights reserved.
 * Copyright 2014 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

#include <assert.h>
#include <dirent.h>
#include <errno.h>
#include <fnmatch.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>
#include <synch.h>
#include <sys/brand.h>
#include <sys/fcntl.h>
#include <sys/param.h>
#include <sys/stat.h>
#include <sys/systeminfo.h>
#include <sys/types.h>
#include <thread.h>
#include <zone.h>

#include <libbrand_impl.h>
#include <libbrand.h>

#define DTD_ELEM_ATTACH         ((const xmlChar *) "attach")
#define DTD_ELEM_BOOT           ((const xmlChar *) "boot")
#define DTD_ELEM_BRAND          ((const xmlChar *) "brand")
#define DTD_ELEM_CLONE          ((const xmlChar *) "clone")
#define DTD_ELEM_COMMENT        ((const xmlChar *) "comment")
#define DTD_ELEM_DETACH         ((const xmlChar *) "detach")
#define DTD_ELEM_DEVICE         ((const xmlChar *) "device")
#define DTD_ELEM_GLOBAL_MOUNT   ((const xmlChar *) "global_mount")
#define DTD_ELEM_HALT           ((const xmlChar *) "halt")
#define DTD_ELEM_INITNAME       ((const xmlChar *) "initname")
#define DTD_ELEM_INSTALL        ((const xmlChar *) "install")
```



```
#define DTD_ELEM_INSTALLOPTS    ((const xmlChar *) "installopts")
#define DTD_ELEM_LOGIN_CMD      ((const xmlChar *) "login_cmd")
#define DTD_ELEM_FORCELOGIN_CMD ((const xmlChar *) "forcedlogin_cmd")
#define DTD_ELEM_MODNAME        ((const xmlChar *) "modname")
#define DTD_ELEM_MOUNT          ((const xmlChar *) "mount")
#define DTD_ELEM_RESTARTINIT    ((const xmlChar *) "restartinit")
#define DTD_ELEM_POSTATTACH     ((const xmlChar *) "postattach")
#define DTD_ELEM_POSTCLONE      ((const xmlChar *) "postclone")
#define DTD_ELEM_POSTINSTALL    ((const xmlChar *) "postinstall")
#define DTD_ELEM_POSTSNAP       ((const xmlChar *) "postsnap")
#define DTD_ELEM_POSTSTATECHG   ((const xmlChar *) "poststatechange")
#define DTD_ELEM_PREDETACH      ((const xmlChar *) "predetach")
#define DTD_ELEM_PRESNAP        ((const xmlChar *) "presnap")
#define DTD_ELEM_PRESTATECHG    ((const xmlChar *) "prestatechange")
#define DTD_ELEM_PREUNINSTALL   ((const xmlChar *) "preuninstall")
#define DTD_ELEM_PRIVILEGE      ((const xmlChar *) "privilege")
#define DTD_ELEM_QUERY          ((const xmlChar *) "query")
#define DTD_ELEM_SHUTDOWN       ((const xmlChar *) "shutdown")
#define DTD_ELEM_SYMLINK        ((const xmlChar *) "symlink")
#define DTD_ELEM_SYSBOOT        ((const xmlChar *) "sysboot")
#define DTD_ELEM_UNINSTALL      ((const xmlChar *) "uninstall")
#define DTD_ELEM_USER_CMD       ((const xmlChar *) "user_cmd")
#define DTD_ELEM_VALIDSNAP      ((const xmlChar *) "validatesnap")
#define DTD_ELEM_VERIFY_CFG     ((const xmlChar *) "verify_cfg")
#define DTD_ELEM_VERIFY_ADM     ((const xmlChar *) "verify_adm")

#define DTD_ATTR_ALLOWEXCL      ((const xmlChar *) "allow-exclusive-ip")
#define DTD_ATTR_ARCH           ((const xmlChar *) "arch")
#define DTD_ATTR_AUTO_CREATE_BE ((const xmlChar *) "auto-create-be")
#endif /* ! codereview */
#define DTD_ATTR_DIRECTORY      ((const xmlChar *) "directory")
#define DTD_ATTR_IPTYPE         ((const xmlChar *) "ip-type")
#define DTD_ATTR_MATCH          ((const xmlChar *) "match")
#define DTD_ATTR_MODE           ((const xmlChar *) "mode")
#define DTD_ATTR_NAME           ((const xmlChar *) "name")
#define DTD_ATTR_OPT            ((const xmlChar *) "opt")
#define DTD_ATTR_PATH           ((const xmlChar *) "path")
#define DTD_ATTR_SET            ((const xmlChar *) "set")
#define DTD_ATTR_SOURCE         ((const xmlChar *) "source")
#define DTD_ATTR_SPECIAL        ((const xmlChar *) "special")
#define DTD_ATTR_TARGET         ((const xmlChar *) "target")
#define DTD_ATTR_TYPE           ((const xmlChar *) "type")

#define DTD_ENTITY_TRUE         "true"
#define DTD_ENTITY_FALSE        "false"
#endif /* ! codereview */

static volatile boolean_t       libbrand_initialized = B_FALSE;
static char                     i_curr_arch[MAXNAMELEN];
static char                     i_curr_zone[ZONENAME_MAX];

/*ARGSUSED*/
static void
brand_error_func(void *ctx, const char *msg, ...)
{
        /*
         * Ignore error messages from libxml
         */
}

static boolean_t
libbrand_initialize()
{
        static mutex_t initialize_lock = DEFAULTMUTEX;

        (void) mutex_lock(&initialize_lock);
```

```
129         if (libbrand_initialized) {
130                 (void) mutex_unlock(&initialize_lock);
131                 return (B_TRUE);
132         }

134         if (sysinfo(SI_ARCHITECTURE, i_curr_arch, sizeof (i_curr_arch)) < 0) {
135                 (void) mutex_unlock(&initialize_lock);
136                 return (B_FALSE);
137         }

139         if (getzonenamebyid(getzoneid(), i_curr_zone,
140             sizeof (i_curr_zone)) < 0) {
141                 (void) mutex_unlock(&initialize_lock);
142                 return (B_FALSE);
143         }

145         /*
146          * Note that here we're initializing per-process libxml2
147          * state.  By doing so we're implicitly assuming that
148          * no other code in this process is also trying to
149          * use libxml2.  But in most case we know this not to
150          * be true since we're almost always used in conjunction
151          * with libzonecfg, which also uses libxml2.  Lucky for
152          * us, libzonecfg initializes libxml2 to essentially
153          * the same defaults as we're using below.
154          */
155         (void) xmlLineNumbersDefault(1);
156         xmlLoadExtDtdDefaultValue |= XML_DETECT_IDS;
157         xmlDoValidityCheckingDefaultValue = 1;
158         (void) xmlKeepBlanksDefault(0);
159         xmlGetWarningsDefaultValue = 0;
160         xmlSetGenericErrorFunc(NULL, brand_error_func);

162         libbrand_initialized = B_TRUE;
163         (void) mutex_unlock(&initialize_lock);
164         return (B_TRUE);
165 }

167 static const char *
168 get_curr_arch(void)
169 {
170         if (!libbrand_initialize())
171                 return (NULL);

173         return (i_curr_arch);
174 }

176 static const char *
177 get_curr_zone(void)
178 {
179         if (!libbrand_initialize())
180                 return (NULL);

182         return (i_curr_zone);
183 }

185 /*
186  * Internal function to open an XML file
187  *
188  * Returns the XML doc pointer, or NULL on failure.  It will validate the
189  * document, as well as removing any comments from the document structure.
190  */
191 static xmlDocPtr
192 open_xml_file(const char *file)
193 {
```



```
194         xmlDocPtr doc;
195         xmlValidCtxtPtr cvp;
196         int valid;

198         if (!libbrand_initialize())
199                 return (NULL);

201         /*
202          * Parse the file
203          */
204         if ((doc = xmlParseFile(file)) == NULL)
205                 return (NULL);

207         /*
208          * Validate the file
209          */
210         if ((cvp = xmlNewValidCtxt()) == NULL) {
211                 xmlFreeDoc(doc);
212                 return (NULL);
213         }
214         cvp->error = brand_error_func;
215         cvp->warning = brand_error_func;
216         valid = xmlValidateDocument(cvp, doc);
217         xmlFreeValidCtxt(cvp);
218         if (valid == 0) {
219                 xmlFreeDoc(doc);
220                 return (NULL);
221         }

223         return (doc);
224 }
225 /*
226  * Open a handle to the named brand.
227  *
228  * Returns a handle to the named brand, which is used for all subsequent brand
229  * interaction, or NULL if unable to open or initialize the brand.
230  */
231 brand_handle_t
232 brand_open(const char *name)
233 {
234         struct brand_handle *bhp;
235         char path[MAXPATHLEN];
236         xmlNodePtr node;
237         xmlChar *property;
238         struct stat statbuf;

240         /*
241          * Make sure brand name isn't too long
242          */
243         if (strlen(name) >= MAXNAMELEN)
244                 return (NULL);

246         /*
247          * Check that the brand exists
248          */
249         (void) snprintf(path, sizeof (path), "%s/%s", BRAND_DIR, name);

251         if (stat(path, &statbuf) != 0)
252                 return (NULL);

254         /*
255          * Allocate brand handle
256          */
257         if ((bhp = malloc(sizeof (struct brand_handle))) == NULL)
258                 return (NULL);
259         bzero(bhp, sizeof (struct brand_handle));
```

```
261         (void) strcpy(bhp->bh_name, name);

263         /*
264          * Open the configuration file
265          */
266         (void) snprintf(path, sizeof (path), "%s/%s/%s", BRAND_DIR, name,
267             BRAND_CONFIG);
268         if ((bhp->bh_config = open_xml_file(path)) == NULL) {
269                 brand_close((brand_handle_t)bhp);
270                 return (NULL);
271         }

273         /*
274          * Verify that the name of the brand matches the directory in which it
275          * is installed.
276          */
277         if ((node = xmlDocGetRootElement(bhp->bh_config)) == NULL) {
278                 brand_close((brand_handle_t)bhp);
279                 return (NULL);
280         }

282         if (xmlStrcmp(node->name, DTD_ELEM_BRAND) != 0) {
283                 brand_close((brand_handle_t)bhp);
284                 return (NULL);
285         }

287         if ((property = xmlGetProp(node, DTD_ATTR_NAME)) == NULL) {
288                 brand_close((brand_handle_t)bhp);
289                 return (NULL);
290         }

292         if (strcmp((char *)property, name) != 0) {
293                 xmlFree(property);
294                 brand_close((brand_handle_t)bhp);
295                 return (NULL);
296         }
297         xmlFree(property);

299         /*
300          * Open handle to platform configuration file.
301          */
302         (void) snprintf(path, sizeof (path), "%s/%s/%s", BRAND_DIR, name,
303             BRAND_PLATFORM);
304         if ((bhp->bh_platform = open_xml_file(path)) == NULL) {
305                 brand_close((brand_handle_t)bhp);
306                 return (NULL);
307         }

309         return ((brand_handle_t)bhp);
310 }

312 /*
313  * Closes the given brand handle
314  */
315 void
316 brand_close(brand_handle_t bh)
317 {
318         struct brand_handle *bhp = (struct brand_handle *)bh;
319         if (bhp->bh_platform != NULL)
320                 xmlFreeDoc(bhp->bh_platform);
321         if (bhp->bh_config != NULL)
322                 xmlFreeDoc(bhp->bh_config);
323         free(bhp);
324 }
```



```
326 static int
327 i_substitute_tokens(const char *sbuf, char *dbuf, int dbuf_size,
328     const char *zonename, const char *zonepath, const char *username,
329     const char *curr_zone)
330 {
331         int dst, src;

333         /*
334          * Walk through the characters, substituting values as needed.
335          */
336         dbuf[0] = '\0';
337         dst = 0;
338         for (src = 0; src < strlen((char *)sbuf) && dst < dbuf_size; src++) {
339                 if (sbuf[src] != '%') {
340                         dbuf[dst++] = sbuf[src];
341                         continue;
342                 }

344                 switch (sbuf[++src]) {
345                 case '%':
346                         dst += strlcpy(dbuf + dst, "%", dbuf_size - dst);
347                         break;
348                 case 'R':
349                         if (zonepath == NULL)
350                                 break;
351                         dst += strlcpy(dbuf + dst, zonepath, dbuf_size - dst);
352                         break;
353                 case 'u':
354                         if (username == NULL)
355                                 break;
356                         dst += strlcpy(dbuf + dst, username, dbuf_size - dst);
357                         break;
358                 case 'Z':
359                         if (curr_zone == NULL)
360                                 break;
361                         /* name of the zone we're running in */
362                         dst += strlcpy(dbuf + dst, curr_zone, dbuf_size - dst);
363                         break;
364                 case 'z':
365                         /* name of the zone we're operating on */
366                         if (zonename == NULL)
367                                 break;
368                         dst += strlcpy(dbuf + dst, zonename, dbuf_size - dst);
369                         break;
370                 }
371         }

373         if (dst >= dbuf_size)
374                 return (-1);

376         dbuf[dst] = '\0';
377         return (0);
378 }

380 /*
381  * Retrieve the given tag from the brand.
382  * Perform the following substitutions as necessary:
383  *
384  *      %%      %
385  *      %u      Username
386  *      %z      Name of target zone
387  *      %Z      Name of current zone
388  *      %R      Zonepath of zone
389  *
390  * Returns 0 on success, -1 on failure.
391  */
```

```
392 static int
393 brand_get_value(struct brand_handle *bhp, const char *zonename,
394     const char *zonepath, const char *username, const char *curr_zone,
395     char *buf, size_t len, const xmlChar *tagname,
396     boolean_t substitute, boolean_t optional)
397 {
398         xmlNodePtr node;
399         xmlChar *content;
400         int err = 0;

402         /*
403          * Retrieve the specified value from the XML doc
404          */
405         if ((node = xmlDocGetRootElement(bhp->bh_config)) == NULL)
406                 return (-1);

408         if (xmlStrcmp(node->name, DTD_ELEM_BRAND) != 0)
409                 return (-1);

411         for (node = node->xmlChildrenNode; node != NULL;
412             node = node->next) {
413                 if (xmlStrcmp(node->name, tagname) == 0)
414                         break;
415         }

417         if (node == NULL) {
418                 if (optional) {
419                         buf[0] = '\0';
420                         return (0);
421                 } else {
422                         return (-1);
423                 }
424         }

426         if ((content = xmlNodeGetContent(node)) == NULL)
427                 return (-1);

429         if (strlen((char *)content) == 0) {
430                 /*
431                  * If the entry in the config file is empty, check to see
432                  * whether this is an optional field.  If so, we return the
433                  * empty buffer.  If not, we return an error.
434                  */
435                 if (optional) {
436                         buf[0] = '\0';
437                 } else {
438                         err = -1;
439                 }
440         } else {
441                 /* Substitute token values as needed. */
442                 if (substitute) {
443                         if (i_substitute_tokens((char *)content, buf, len,
444                             zonename, zonepath, username, curr_zone) != 0)
445                                 err = -1;
446                 } else {
447                         if (strlcpy(buf, (char *)content, len) >= len)
448                                 err = -1;
449                 }
450         }

452         xmlFree(content);

454         return (err);
455 }

457 int
```



```
458 brand_get_attach(brand_handle_t bh, const char *zonename,
459     const char *zonepath, char *buf, size_t len)
460 {
461         struct brand_handle *bhp = (struct brand_handle *)bh;
462         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
463             buf, len, DTD_ELEM_ATTACH, B_TRUE, B_TRUE));
464 }

466 int
467 brand_get_boot(brand_handle_t bh, const char *zonename,
468     const char *zonepath, char *buf, size_t len)
469 {
470         struct brand_handle *bhp = (struct brand_handle *)bh;
471         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
472             buf, len, DTD_ELEM_BOOT, B_TRUE, B_TRUE));
473 }

475 int
476 brand_get_brandname(brand_handle_t bh, char *buf, size_t len)
477 {
478         struct brand_handle *bhp = (struct brand_handle *)bh;
479         if (len <= strlen(bhp->bh_name))
480                 return (-1);

482         (void) strcpy(buf, bhp->bh_name);

484         return (0);
485 }

487 int
488 brand_get_clone(brand_handle_t bh, const char *zonename,
489     const char *zonepath, char *buf, size_t len)
490 {
491         struct brand_handle *bhp = (struct brand_handle *)bh;
492         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
493             buf, len, DTD_ELEM_CLONE, B_TRUE, B_TRUE));
494 }

496 int
497 brand_get_detach(brand_handle_t bh, const char *zonename,
498     const char *zonepath, char *buf, size_t len)
499 {
500         struct brand_handle *bhp = (struct brand_handle *)bh;
501         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
502             buf, len, DTD_ELEM_DETACH, B_TRUE, B_TRUE));
503 }

505 int
506 brand_get_halt(brand_handle_t bh, const char *zonename,
507     const char *zonepath, char *buf, size_t len)
508 {
509         struct brand_handle *bhp = (struct brand_handle *)bh;
510         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
511             buf, len, DTD_ELEM_HALT, B_TRUE, B_TRUE));
512 }

514 int
515 brand_get_shutdown(brand_handle_t bh, const char *zonename,
516     const char *zonepath, char *buf, size_t len)
517 {
518         struct brand_handle *bhp = (struct brand_handle *)bh;
519         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
520             buf, len, DTD_ELEM_SHUTDOWN, B_TRUE, B_TRUE));
521 }

523 int
```

```
524 brand_get_initname(brand_handle_t bh, char *buf, size_t len)
525 {
526         struct brand_handle *bhp = (struct brand_handle *)bh;
527         return (brand_get_value(bhp, NULL, NULL, NULL, NULL,
528             buf, len, DTD_ELEM_INITNAME, B_FALSE, B_FALSE));
529 }

531 boolean_t
532 brand_restartinit(brand_handle_t bh)
533 {
534         struct brand_handle *bhp = (struct brand_handle *)bh;
535         char val[80];

537         if (brand_get_value(bhp, NULL, NULL, NULL, NULL,
538             val, sizeof (val), DTD_ELEM_RESTARTINIT, B_FALSE, B_FALSE) != 0)
539                 return (B_TRUE);

541         if (strcmp(val, "false") == 0)
542                 return (B_FALSE);
543         return (B_TRUE);
544 }

546 int
547 brand_get_login_cmd(brand_handle_t bh, const char *username,
548     char *buf, size_t len)
549 {
550         struct brand_handle *bhp = (struct brand_handle *)bh;
551         const char *curr_zone = get_curr_zone();
552         return (brand_get_value(bhp, NULL, NULL, username, curr_zone,
553             buf, len, DTD_ELEM_LOGIN_CMD, B_TRUE, B_FALSE));
554 }

556 int
557 brand_get_forcedlogin_cmd(brand_handle_t bh, const char *username,
558     char *buf, size_t len)
559 {
560         struct brand_handle *bhp = (struct brand_handle *)bh;
561         const char *curr_zone = get_curr_zone();
562         return (brand_get_value(bhp, NULL, NULL, username, curr_zone,
563             buf, len, DTD_ELEM_FORCELOGIN_CMD, B_TRUE, B_FALSE));
564 }

566 int
567 brand_get_user_cmd(brand_handle_t bh, const char *username,
568     char *buf, size_t len)
569 {
570         struct brand_handle *bhp = (struct brand_handle *)bh;

572         return (brand_get_value(bhp, NULL, NULL, username, NULL,
573             buf, len, DTD_ELEM_USER_CMD, B_TRUE, B_FALSE));
574 }

576 int
577 brand_get_install(brand_handle_t bh, const char *zonename,
578     const char *zonepath, char *buf, size_t len)
579 {
580         struct brand_handle *bhp = (struct brand_handle *)bh;
581         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
582             buf, len, DTD_ELEM_INSTALL, B_TRUE, B_FALSE));
583 }

585 int
586 brand_get_installopts(brand_handle_t bh, char *buf, size_t len)
587 {
588         struct brand_handle *bhp = (struct brand_handle *)bh;
589         return (brand_get_value(bhp, NULL, NULL, NULL, NULL,
```



```
590             buf, len, DTD_ELEM_INSTALLOPTS, B_FALSE, B_TRUE));
591 }

593 int
594 brand_get_modname(brand_handle_t bh, char *buf, size_t len)
595 {
596         struct brand_handle *bhp = (struct brand_handle *)bh;
597         return (brand_get_value(bhp, NULL, NULL, NULL, NULL,
598             buf, len, DTD_ELEM_MODNAME, B_FALSE, B_TRUE));
599 }

601 int
602 brand_get_postattach(brand_handle_t bh, const char *zonename,
603     const char *zonepath, char *buf, size_t len)
604 {
605         struct brand_handle *bhp = (struct brand_handle *)bh;
606         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
607             buf, len, DTD_ELEM_POSTATTACH, B_TRUE, B_TRUE));
608 }

610 int
611 brand_get_postclone(brand_handle_t bh, const char *zonename,
612     const char *zonepath, char *buf, size_t len)
613 {
614         struct brand_handle *bhp = (struct brand_handle *)bh;
615         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
616             buf, len, DTD_ELEM_POSTCLONE, B_TRUE, B_TRUE));
617 }

619 int
620 brand_get_postinstall(brand_handle_t bh, const char *zonename,
621     const char *zonepath, char *buf, size_t len)
622 {
623         struct brand_handle *bhp = (struct brand_handle *)bh;
624         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
625             buf, len, DTD_ELEM_POSTINSTALL, B_TRUE, B_TRUE));
626 }

628 int
629 brand_get_postsnap(brand_handle_t bh, const char *zonename,
630     const char *zonepath, char *buf, size_t len)
631 {
632         struct brand_handle *bhp = (struct brand_handle *)bh;
633         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
634             buf, len, DTD_ELEM_POSTSNAP, B_TRUE, B_TRUE));
635 }

637 int
638 brand_get_poststatechange(brand_handle_t bh, const char *zonename,
639     const char *zonepath, char *buf, size_t len)
640 {
641         struct brand_handle *bhp = (struct brand_handle *)bh;
642         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
643             buf, len, DTD_ELEM_POSTSTATECHG, B_TRUE, B_TRUE));
644 }

646 int
647 brand_get_predetach(brand_handle_t bh, const char *zonename,
648     const char *zonepath, char *buf, size_t len)
649 {
650         struct brand_handle *bhp = (struct brand_handle *)bh;
651         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
652             buf, len, DTD_ELEM_PREDETACH, B_TRUE, B_TRUE));
653 }

655 int
```

```
656 brand_get_presnap(brand_handle_t bh, const char *zonename,
657     const char *zonepath, char *buf, size_t len)
658 {
659         struct brand_handle *bhp = (struct brand_handle *)bh;
660         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
661             buf, len, DTD_ELEM_PRESNAP, B_TRUE, B_TRUE));
662 }

664 int
665 brand_get_prestatechange(brand_handle_t bh, const char *zonename,
666     const char *zonepath, char *buf, size_t len)
667 {
668         struct brand_handle *bhp = (struct brand_handle *)bh;
669         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
670             buf, len, DTD_ELEM_PRESTATECHG, B_TRUE, B_TRUE));
671 }

673 int
674 brand_get_preuninstall(brand_handle_t bh, const char *zonename,
675     const char *zonepath, char *buf, size_t len)
676 {
677         struct brand_handle *bhp = (struct brand_handle *)bh;
678         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
679             buf, len, DTD_ELEM_PREUNINSTALL, B_TRUE, B_TRUE));
680 }

682 int
683 brand_get_query(brand_handle_t bh, const char *zonename,
684     const char *zonepath, char *buf, size_t len)
685 {
686         struct brand_handle *bhp = (struct brand_handle *)bh;
687         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
688             buf, len, DTD_ELEM_QUERY, B_TRUE, B_TRUE));
689 }

691 int
692 brand_get_uninstall(brand_handle_t bh, const char *zonename,
693     const char *zonepath, char *buf, size_t len)
694 {
695         struct brand_handle *bhp = (struct brand_handle *)bh;
696         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
697             buf, len, DTD_ELEM_UNINSTALL, B_TRUE, B_TRUE));
698 }

700 int
701 brand_get_validatesnap(brand_handle_t bh, const char *zonename,
702     const char *zonepath, char *buf, size_t len)
703 {
704         struct brand_handle *bhp = (struct brand_handle *)bh;
705         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
706             buf, len, DTD_ELEM_VALIDSNAP, B_TRUE, B_TRUE));
707 }

709 int
710 brand_get_verify_cfg(brand_handle_t bh, char *buf, size_t len)
711 {
712         struct brand_handle *bhp = (struct brand_handle *)bh;
713         return (brand_get_value(bhp, NULL, NULL, NULL, NULL,
714             buf, len, DTD_ELEM_VERIFY_CFG, B_FALSE, B_TRUE));
715 }

717 int
718 brand_get_verify_adm(brand_handle_t bh, const char *zonename,
719     const char *zonepath, char *buf, size_t len)
720 {
721         struct brand_handle *bhp = (struct brand_handle *)bh;
```



```
722         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
723             buf, len, DTD_ELEM_VERIFY_ADM, B_TRUE, B_TRUE));
724 }

726 int
727 brand_get_sysboot(brand_handle_t bh, const char *zonename,
728     const char *zonepath, char *buf, size_t len)
729 {
730         struct brand_handle *bhp = (struct brand_handle *)bh;
731         return (brand_get_value(bhp, zonename, zonepath, NULL, NULL,
732             buf, len, DTD_ELEM_SYSBOOT, B_TRUE, B_TRUE));
733 }

735 boolean_t
736 brand_allow_exclusive_ip(brand_handle_t bh)
737 {
738         struct brand_handle     *bhp = (struct brand_handle *)bh;
739         xmlNodePtr              node;
740         xmlChar                 *allow_excl;
741         boolean_t               ret;

743         assert(bhp != NULL);

745         if ((node = xmlDocGetRootElement(bhp->bh_platform)) == NULL)
746                 return (B_FALSE);

748         allow_excl = xmlGetProp(node, DTD_ATTR_ALLOWEXCL);
749         if (allow_excl == NULL)
750                 return (B_FALSE);

752         /* Note: only return B_TRUE if it's "true" */
753         if (strcmp((char *)allow_excl, DTD_ENTITY_TRUE) == 0)
754                 ret = B_TRUE;
755         else
756                 ret = B_FALSE;

758         xmlFree(allow_excl);

760         return (ret);
761 }

763 boolean_t
764 brand_auto_create_be(brand_handle_t bh)
765 {
766         struct brand_handle     *bhp = (struct brand_handle *)bh;
767         xmlNodePtr              node;
768         xmlChar                 *auto_create_be;
769         boolean_t               ret;

771         assert(bhp != NULL);

773         if ((node = xmlDocGetRootElement(bhp->bh_platform)) == NULL)
774                 return (B_FALSE);

776         auto_create_be = xmlGetProp(node, DTD_ATTR_AUTO_CREATE_BE);
777         if (auto_create_be == NULL)
778                 return (B_FALSE);

780         /* Note: only return B_FALSE if it's "false" */
781         if (strcmp((char *)auto_create_be, DTD_ENTITY_FALSE) == 0)
782                 ret = B_FALSE;
783         else
784                 ret = B_TRUE;

786         xmlFree(auto_create_be);
787 #endif /* ! codereview */
```

```
789         return (ret);
790 }

792 /*
793  * Iterate over brand privileges
794  *
795  * Walks the brand config, searching for <privilege> elements, calling the
796  * specified callback for each.  Returns 0 on success, or -1 on failure.
797  */
798 int
799 brand_config_iter_privilege(brand_handle_t bh,
800     int (*func)(void *, priv_iter_t *), void *data)
801 {
802         struct brand_handle     *bhp = (struct brand_handle *)bh;
803         xmlNodePtr              node;
804         xmlChar                 *name, *set, *iptype;
805         priv_iter_t             priv_iter;
806         int                     ret;

808         if ((node = xmlDocGetRootElement(bhp->bh_config)) == NULL)
809                 return (-1);

811         for (node = node->xmlChildrenNode; node != NULL; node = node->next) {

813                 if (xmlStrcmp(node->name, DTD_ELEM_PRIVILEGE) != 0)
814                         continue;

816                 name = xmlGetProp(node, DTD_ATTR_NAME);
817                 set = xmlGetProp(node, DTD_ATTR_SET);
818                 iptype = xmlGetProp(node, DTD_ATTR_IPTYPE);

820                 if (name == NULL || set == NULL || iptype == NULL) {
821                         if (name != NULL)
822                                 xmlFree(name);
823                         if (set != NULL)
824                                 xmlFree(set);
825                         if (iptype != NULL)
826                                 xmlFree(iptype);
827                         return (-1);
828                 }

830                 priv_iter.pi_name = (char *)name;
831                 priv_iter.pi_set = (char *)set;
832                 priv_iter.pi_iptype = (char *)iptype;

834                 ret = func(data, &priv_iter);

836                 xmlFree(name);
837                 xmlFree(set);
838                 xmlFree(iptype);

840                 if (ret != 0)
841                         return (-1);
842         }

844         return (0);
845 }

847 static int
848 i_brand_platform_iter_mounts(struct brand_handle *bhp, const char *zonepath,
849     int (*func)(void *, const char *, const char *, const char *,
850     const char *), void *data, const xmlChar *mount_type)
851 {
852         xmlNodePtr node;
853         xmlChar *special, *dir, *type, *opt;
```



```
854         char special_exp[MAXPATHLEN];
855         char opt_exp[MAXPATHLEN];
856         int ret;

858         if ((node = xmlDocGetRootElement(bhp->bh_platform)) == NULL)
859                 return (-1);

861         for (node = node->xmlChildrenNode; node != NULL; node = node->next) {

863                 if (xmlStrcmp(node->name, mount_type) != 0)
864                         continue;

866                 special = xmlGetProp(node, DTD_ATTR_SPECIAL);
867                 dir = xmlGetProp(node, DTD_ATTR_DIRECTORY);
868                 type = xmlGetProp(node, DTD_ATTR_TYPE);
869                 opt = xmlGetProp(node, DTD_ATTR_OPT);
870                 if ((special == NULL) || (dir == NULL) || (type == NULL) ||
871                     (opt == NULL)) {
872                         ret = -1;
873                         goto next;
874                 }

876                 /* Substitute token values as needed. */
877                 if ((ret = i_substitute_tokens((char *)special,
878                     special_exp, sizeof (special_exp),
879                     NULL, zonepath, NULL, NULL)) != 0)
880                         goto next;

882                 /* opt might not be defined */
883                 if (strlen((const char *)opt) == 0) {
884                         xmlFree(opt);
885                         opt = NULL;
886                 } else {
887                         if ((ret = i_substitute_tokens((char *)opt,
888                             opt_exp, sizeof (opt_exp),
889                             NULL, zonepath, NULL, NULL)) != 0)
890                                 goto next;
891                 }

893                 ret = func(data, (char *)special_exp, (char *)dir,
894                     (char *)type, ((opt != NULL) ? opt_exp : NULL));

896 next:
897                 if (special != NULL)
898                         xmlFree(special);
899                 if (dir != NULL)
900                         xmlFree(dir);
901                 if (type != NULL)
902                         xmlFree(type);
903                 if (opt != NULL)
904                         xmlFree(opt);
905                 if (ret != 0)
906                         return (-1);
907         }
908         return (0);
909 }


912 /*
913  * Iterate over global platform filesystems
914  *
915  * Walks the platform, searching for <global_mount> elements, calling the
916  * specified callback for each.  Returns 0 on success, or -1 on failure.
917  *
918  * Perform the following substitutions as necessary:
919  *
```

```
 *      %R      Zonepath of zone
 */
int
brand_platform_iter_gmounts(brand_handle_t bh, const char *zonepath,
    int (*func)(void *, const char *, const char *, const char *,
    const char *), void *data)
{
        struct brand_handle *bhp = (struct brand_handle *)bh;
        return (i_brand_platform_iter_mounts(bhp, zonepath, func, data,
            DTD_ELEM_GLOBAL_MOUNT));
}

/*
 * Iterate over non-global zone platform filesystems
 *
 * Walks the platform, searching for <mount> elements, calling the
 * specified callback for each.  Returns 0 on success, or -1 on failure.
 */
int
brand_platform_iter_mounts(brand_handle_t bh, int (*func)(void *,
    const char *, const char *, const char *, const char *), void *data)
{
        struct brand_handle *bhp = (struct brand_handle *)bh;
        return (i_brand_platform_iter_mounts(bhp, NULL, func, data,
            DTD_ELEM_MOUNT));
}

/*
 * Iterate over platform symlinks
 *
 * Walks the platform, searching for <symlink> elements, calling the
 * specified callback for each.  Returns 0 on success, or -1 on failure.
 */
int
brand_platform_iter_link(brand_handle_t bh,
    int (*func)(void *, const char *, const char *), void *data)
{
        struct brand_handle *bhp = (struct brand_handle *)bh;
        xmlNodePtr node;
        xmlChar *source, *target;
        int ret;

        if ((node = xmlDocGetRootElement(bhp->bh_platform)) == NULL)
                return (-1);

        for (node = node->xmlChildrenNode; node != NULL; node = node->next) {

                if (xmlStrcmp(node->name, DTD_ELEM_SYMLINK) != 0)
                        continue;

                source = xmlGetProp(node, DTD_ATTR_SOURCE);
                target = xmlGetProp(node, DTD_ATTR_TARGET);

                if (source == NULL || target == NULL) {
                        if (source != NULL)
                                xmlFree(source);
                        if (target != NULL)
                                xmlFree(target);
                        return (-1);
                }

                ret = func(data, (char *)source, (char *)target);

                xmlFree(source);
                xmlFree(target);
```



```
                if (ret != 0)
                        return (-1);
        }

        return (0);
}

/*
 * Iterate over platform devices
 *
 * Walks the platform, searching for <device> elements, calling the
 * specified callback for each.  Returns 0 on success, or -1 on failure.
 */
int
brand_platform_iter_devices(brand_handle_t bh, const char *zonename,
    int (*func)(void *, const char *, const char *), void *data,
    const char *curr_iptype)
{
        struct brand_handle     *bhp = (struct brand_handle *)bh;
        const char              *curr_arch = get_curr_arch();
        xmlNodePtr              node;
        xmlChar                 *match, *name, *arch, *iptype;
        char                    match_exp[MAXPATHLEN];
        boolean_t               err = B_FALSE;
        int                     ret = 0;


        assert(bhp != NULL);
        assert(zonename != NULL);
        assert(func != NULL);
        assert(curr_iptype != NULL);

        if ((node = xmlDocGetRootElement(bhp->bh_platform)) == NULL)
                return (-1);

        for (node = node->xmlChildrenNode; node != NULL; node = node->next) {

                if (xmlStrcmp(node->name, DTD_ELEM_DEVICE) != 0)
                        continue;

                match = xmlGetProp(node, DTD_ATTR_MATCH);
                name = xmlGetProp(node, DTD_ATTR_NAME);
                arch = xmlGetProp(node, DTD_ATTR_ARCH);
                iptype = xmlGetProp(node, DTD_ATTR_IPTYPE);
                if ((match == NULL) || (name == NULL) || (arch == NULL) ||
                    (iptype == NULL)) {
                        err = B_TRUE;
                        goto next;
                }

                /* check if the arch matches */
                if ((strcmp((char *)arch, "all") != 0) &&
                    (strcmp((char *)arch, curr_arch) != 0))
                        goto next;

                /* check if the iptype matches */
                if ((strcmp((char *)iptype, "all") != 0) &&
                    (strcmp((char *)iptype, curr_iptype) != 0))
                        goto next;

                /* Substitute token values as needed. */
                if ((ret = i_substitute_tokens((char *)match,
                    match_exp, sizeof (match_exp),
                    zonename, NULL, NULL)) != 0) {
                        err = B_TRUE;
                        goto next;
```

```
1052                }

1054                /* name might not be defined */
1055                if (strlen((const char *)name) == 0) {
1056                        xmlFree(name);
1057                        name = NULL;
1058                }

1060                /* invoke the callback */
1061                ret = func(data, (const char *)match_exp, (const char *)name);

1063 next:
1064                if (match != NULL)
1065                        xmlFree(match);
1066                if (name != NULL)
1067                        xmlFree(name);
1068                if (arch != NULL)
1069                        xmlFree(arch);
1070                if (iptype != NULL)
1071                        xmlFree(iptype);
1072                if (err)
1073                        return (-1);
1074                if (ret != 0)
1075                        return (-1);
1076        }

1078        return (0);
1079 }
```

```
**********************************************************
    4871 Wed Nov 11 10:43:15 2015
new/usr/src/lib/libbrand/common/libbrand.h
patch zone-auto-create-be
**********************************************************
```

```
/*
 * 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) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011, Joyent, Inc. All rights reserved.
 * Copyright 2014 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

#ifndef _LIBBRAND_H
#define _LIBBRAND_H

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/types.h>

typedef struct __brand_handle *brand_handle_t;

typedef struct priv_iter_s {
        char    *pi_name;
        char    *pi_set;
        char    *pi_iptype;
} priv_iter_t;

extern brand_handle_t brand_open(const char *);
extern void brand_close(brand_handle_t);

extern boolean_t brand_allow_exclusive_ip(brand_handle_t);
extern boolean_t brand_auto_create_be(brand_handle_t);
#endif /* ! codereview */

extern int brand_get_attach(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_boot(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_brandname(brand_handle_t, char *, size_t);
extern int brand_get_clone(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_detach(brand_handle_t, const char *, const char *,
```



```
    char *, size_t);
extern int brand_get_shutdown(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_halt(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_initname(brand_handle_t, char *, size_t);
extern boolean_t brand_restartinit(brand_handle_t);
extern int brand_get_install(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_installopts(brand_handle_t, char *, size_t);
extern int brand_get_login_cmd(brand_handle_t, const char *, char *, size_t);
extern int brand_get_forcedlogin_cmd(brand_handle_t, const char *,
    char *, size_t);
extern int brand_get_modname(brand_handle_t, char *, size_t);
extern int brand_get_postattach(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_postclone(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_postinstall(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_postsnap(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_poststatechange(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_predetach(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_presnap(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_prestatechange(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_preuninstall(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_query(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_uninstall(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_validatesnap(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_user_cmd(brand_handle_t, const char *, char *, size_t);
extern int brand_get_verify_cfg(brand_handle_t, char *, size_t);
extern int brand_get_verify_adm(brand_handle_t, const char *, const char *,
    char *, size_t);
extern int brand_get_sysboot(brand_handle_t, const char *, const char *, char *,
    size_t);

extern int brand_config_iter_privilege(brand_handle_t,
    int (*func)(void *, priv_iter_t *), void *);

extern int brand_platform_iter_devices(brand_handle_t, const char *,
    int (*)(void *, const char *, const char *), void *, const char *);
extern int brand_platform_iter_gmounts(brand_handle_t, const char *,
    int (*)(void *, const char *, const char *, const char *, const char *),
    void *);
extern int brand_platform_iter_link(brand_handle_t, int (*)(void *,
    const char *, const char *), void *);
extern int brand_platform_iter_mounts(brand_handle_t, int (*)(void *,
    const char *, const char *, const char *, const char *), void *);

#ifdef  __cplusplus
}
#endif

#endif  /* _LIBBRAND_H */
```

```
**********************************************************
    2302 Wed Nov 11 10:43:16 2015
new/usr/src/lib/libbrand/common/mapfile-vers
patch zone-auto-create-be
**********************************************************
```

```
#
# 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) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright (c) 2011, Joyent, Inc. All rights reserved.
# Copyright 2014 Nexenta Systems, Inc. All rights reserved.
#

#
# MAPFILE HEADER START
#
# WARNING:  STOP NOW.  DO NOT MODIFY THIS FILE.
# Object versioning must comply with the rules detailed in
#
#       usr/src/lib/README.mapfiles
#
# You should not be making modifications here until you've read the most current
# copy of that file. If you need help, contact a gatekeeper for guidance.
#
# MAPFILE HEADER END
#

$mapfile_version 2

SYMBOL_VERSION SUNWprivate {
    global:
        brand_allow_exclusive_ip;
        brand_auto_create_be;
#endif /* ! codereview */
        brand_close;
        brand_config_iter_privilege;
        brand_get_attach;
        brand_get_boot;
        brand_get_brandname;
        brand_get_clone;
        brand_get_detach;
        brand_get_forcedlogin_cmd;
        brand_get_halt;
        brand_get_initname;
        brand_get_install;
        brand_get_installopts;
        brand_get_login_cmd;
        brand_get_modname;
```



```
        brand_get_postattach;
        brand_get_postclone;
        brand_get_postinstall;
        brand_get_postsnap;
        brand_get_poststatechange;
        brand_get_predetach;
        brand_get_presnap;
        brand_get_prestatechange;
        brand_get_preuninstall;
        brand_get_query;
        brand_get_sysboot;
        brand_get_shutdown;
        brand_get_uninstall;
        brand_get_user_cmd;
        brand_get_validatesnap;
        brand_get_verify_adm;
        brand_get_verify_cfg;
        brand_open;
        brand_platform_iter_devices;
        brand_platform_iter_gmounts;
        brand_platform_iter_link;
        brand_platform_iter_mounts;
        brand_restartinit;
    local:
        *;
};
```

```
**********************************************************
    2023 Wed Nov 11 10:43:16 2015
new/usr/src/lib/libinstzones/Makefile.com
patch zone-auto-create-be
**********************************************************
```

```
#
# 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 2009 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# Copyright 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#
#endif /* ! codereview */

LIBRARY=        libinstzones.a
VERS=           .1

OBJECTS =       \
                zones_args.o \
                zones_exec.o \
                zones_locks.o \
                zones_paths.o \
                zones_states.o \
                zones_str.o \
                zones_utils.o \
                zones_lofs.o \
                zones.o

# include library definitions
include $(SRC)/lib/Makefile.lib

SRCDIR=         ../common

POFILE =        libinstzones.po
MSGFILES =      $(OBJECTS:%.o=../common/%.i)
CLEANFILES +=   $(MSGFILES)

# openssl forces us to ignore dubious pointer casts, thanks to its clever
# use of macros for stack management.
LINTFLAGS=      -umx -errtags \
                -erroff=E_BAD_PTR_CAST_ALIGN,E_BAD_PTR_CAST
$(LINTLIB):=    SRCS = $(SRCDIR)/$(LINTSRC)

CERRWARN +=     -_gcc=-Wno-parentheses
CERRWARN +=     -_gcc=-Wno-clobbered
CERRWARN +=     -_gcc=-Wno-address
```



```
LIBS = $(DYNLIB) $(LINTLIB)

DYNFLAGS += $(ZLAZYLOAD)

LDLIBS +=       -lc -lcontract -lzonecfg -lbrand
LDLIBS +=       -lc -lcontract -lzonecfg

CFLAGS +=       $(CCVERBOSE)
CPPFLAGS +=     -I$(SRCDIR)

.KEEP_STATE:

all:    $(LIBS)

$(POFILE): $(MSGFILES)
        $(BUILDPO.msgfiles)

_msg: $(MSGDOMAINPOFILE)

lint:   lintcheck

# include library targets
include $(SRC)/lib/Makefile.targ
include $(SRC)/Makefile.msg.targ
```

```
**********************************************************
    2389 Wed Nov 11 10:43:16 2015
new/usr/src/lib/libinstzones/common/mapfile-vers
patch zone-auto-create-be
**********************************************************
```

```
#
# 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 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
#

#
# MAPFILE HEADER START
#
# WARNING:  STOP NOW.  DO NOT MODIFY THIS FILE.
# Object versioning must comply with the rules detailed in
#
#       usr/src/lib/README.mapfiles
#
# You should not be making modifications here until you've read the most current
# copy of that file. If you need help, contact a gatekeeper for guidance.
#
# MAPFILE HEADER END
#

$mapfile_version 2

SYMBOL_VERSION SUNWprivate {
    global:
        UmountAllZones;
        z_brands_are_implemented;
        z_canoninplace;
        z_createMountTable;
        z_create_zone_admin_file;
        z_destroyMountTable;
        z_ExecCmdArray;
        z_ExecCmdList;
        z_free_brand_list;
        z_free_zone_list;
        z_get_nonglobal_zone_list;
        z_get_nonglobal_branded_zone_list;
#endif /* ! codereview */
        z_get_nonglobal_zone_list_by_brand;
        z_get_zonename;
        z_global_only;
```



```
        z_isPathWritable;
        z_is_zone_branded;
        z_is_zone_brand_in_list;
        z_lock_this_zone;
        z_lock_zones;
        z_make_brand_list;
        z_make_zone_root;
        z_mount_in_lz;
        z_non_global_zones_exist;
        z_on_zone_spec;
        z_path_canonize;
        z_resolve_lofs;
        z_running_in_global_zone;
        z_set_output_functions;
        z_set_zone_root;
        z_set_zone_spec;
        z_umount_lz_mount;
        z_unlock_this_zone;
        z_unlock_zones;
        z_verify_zone_spec;
        z_zlist_change_zone_state;
        z_zlist_get_current_state;
        z_zlist_get_original_state;
        z_zlist_get_scratch;
        z_zlist_get_zonename;
        z_zlist_get_zonepath;
        z_zlist_is_zone_auto_create_be;
#endif /* ! codereview */
        z_zlist_is_zone_runnable;
        z_zlist_restore_zone_state;
        z_zone_exec;
        z_zones_are_implemented;
  local:
        *;
};
```

```
**********************************************************
   60008 Wed Nov 11 10:43:16 2015
new/usr/src/lib/libinstzones/common/zones.c
patch zone-auto-create-be
**********************************************************
```

```
/*
 * 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 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */


/*
 * Module:      zones.c
 * Group:       libinstzones
 * Description: Provide "zones" interface for install consolidation code
 *
 * Public Methods:
 *  z_create_zone_admin_file - Given a location to create the file, and
 *      optionally an existing administration file, generate an
 *      administration file that can be used to perform "non-interactive"
 *      operations in a non-global zone.
 *  z_free_zone_list - free contents of zoneList_t object
 *  z_get_nonglobal_zone_list - return zoneList_t object describing all
 *      non-global native zones
 *  z_get_nonglobal_branded_zone_list - return zoneList_t object describing
 *      all branded non-global zones
#endif /* ! codereview */
 *  z_get_nonglobal_zone_list_by_brand - return zoneList_t object describing
 *      all non-global zones matching the list of zone brands passed in.
 *  z_free_brand_list - free contents of a zoneBrandList_t object
 *  z_make_brand_list - return a zoneBrandList_t object describing the list
 *      of all zone brands passed in.
 *  z_get_zonename - return the name of the current zone
 *  z_global_only - Determine if the global zone is only zone on the spec list
 *  z_lock_this_zone - lock this zone
 *  z_lock_zones - lock specified zones
 *  z_mount_in_lz - Mount global zone directory in specified zone's root file
 *      system
 *  z_non_global_zones_exist - Determine if any non-global native zones exist
 *  z_on_zone_spec - Determine if named zone is on the zone_spec list
 *  z_running_in_global_zone - Determine if running in the "global" zone
 *  z_set_output_functions - Link program specific output functions
 *  z_set_zone_root - Set root for zones library operations
 *  z_set_zone_spec - Set list of zones on which actions will be performed
```



```
 *  z_umount_lz_mount - Unmount directory mounted with z_mount_in_lz
 *  z_unlock_this_zone - unlock this zone
 *  z_unlock_zones - unlock specified zones
 *  z_verify_zone_spec - Verify list of zones on which actions will be performed
 *  z_zlist_change_zone_state - Change the current state of the specified zone
 *  z_zlist_get_current_state - Determine the current kernel state of the
 *      specified zone
 *  z_zlist_get_original_state - Return the original kernal state of the
 *      specified zone
 *  z_zlist_get_scratch - Determine name of scratch zone
 *  z_zlist_get_zonename - Determine name of specified zone
 *  z_zlist_get_zonepath - Determine zonepath of specified zone
 *  z_zlist_restore_zone_state - Return the zone to the state it was originally
 *      in
 *  z_zone_exec - Execute a Unix command in a specified zone and return results
 *  z_zones_are_implemented - Determine if any zone operations can be performed
 *  z_is_zone_branded - determine if zone has a non-native brand
 *  z_is_zone_brand_in_list - determine if the zone's brand matches the
 *      brand list passed in.
 *  z_brands_are_implemented - determine if branded zones are implemented on
 *                      this system
 */

/*
 * System includes
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <string.h>
#include <strings.h>
#include <sys/stat.h>
#include <stdarg.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <signal.h>
#include <stropts.h>
#include <wait.h>
#include <zone.h>
#include <sys/brand.h>
#include <libintl.h>
#include <locale.h>
#include <libzonecfg.h>
#include <libcontract.h>
#include <sys/contract/process.h>
#include <sys/ctfs.h>
#include <assert.h>
#include <dlfcn.h>
#include <link.h>
#include <time.h>

/*
 * local includes
 */

/*
 * When _INSTZONES_LIB_Z_DEFINE_GLOBAL_DATA is defined,
 * instzones_lib.h will define the z_global_data structure.
 * Otherwise an extern to the structure is inserted.
```

```
 */

#define _INSTZONES_LIB_Z_DEFINE_GLOBAL_DATA
#include "instzones_lib.h"
#include "zones_strings.h"

/*
 * Private structures
 */

#define CLUSTER_BRAND_NAME      "cluster"

/* maximum number of arguments to exec() call */

#define UUID_FORMAT     "%02d%02d%02d%03d-%02d%02d%02d%d-%016llx"

/*
 * Library Function Prototypes
 */

#define streq(a, b) (strcmp((a), (b)) == 0)

/*
 * Local Function Prototypes
 */

/*
 * global internal (private) declarations
 */

/*
 * ************************************************************************
 * global external (public) functions
 * ************************************************************************
 */

/*
 * Name:        z_create_zone_admin_file
 * Description: Given a location to create the file, and optionally an existing
 *              administration file, generate an administration file that
 *              can be used to perform "non-interactive" operations in a
 *              non-global zone.
 * Arguments:   a_zoneAdminFilename - pointer to string representing the
 *                      full path of zone admin file to create
 *              a_userAdminFilename - pointer to string representing the path
 *                      to an existing "user" administration file - the
 *                      administration file created will contain the
 *                      settings contained in this file, modified as
 *                      appropriate to supress any interaction;
 *                      If this is == NULL then the administration file
 *                      created will not contain any extra settings
 * Returns:     boolean_t
 *                      == B_TRUE - admin file created
 *                      == B_FALSE - failed to create admin file
 */

boolean_t
z_create_zone_admin_file(char *a_zoneAdminFilename, char *a_userAdminFilename)
{
        FILE    *zFp;
        FILE    *uFp = (FILE *)NULL;

        /* entry assertions */

        assert(a_zoneAdminFilename != NULL);
        assert(*a_zoneAdminFilename != '\0');
```



```
        /* create temporary zone admin file */

        zFp = fopen(a_zoneAdminFilename, "w");
        if (zFp == (FILE *)NULL) {
                return (B_FALSE);
        }

        /* open user admin file if specified */

        if (a_userAdminFilename != (char *)NULL) {
                uFp = fopen(a_userAdminFilename, "r");
        }

        /* create default admin file for zone pkg ops if no user admin file */

        if (uFp == (FILE *)NULL) {
                /* create default admin file */
                (void) fprintf(zFp, "action=nocheck\nauthentication=nocheck\n"
                    "basedir=default\nconflict=nocheck\nidepend=nocheck\n"
                    "instance=unique\npartial=nocheck\nrdepend=nocheck\n"
                    "runlevel=nocheck\nsetuid=nocheck\nspace=nocheck\n"
                    "mail=\n");
        } else for (;;) {
                /* copy user admin file substitute/change appropriate entries */
                char    buf[LINE_MAX+1];
                char    *p;

                /* read next line of user admin file */

                p = fgets(buf, sizeof (buf), uFp);
                if (p == (char *)NULL) {
                        (void) fclose(uFp);
                        break;
                }

                /* modify / replace / accept as appropriate */

                if (strncmp(buf, "instance=quit", 13) == 0) {
                        (void) fprintf(zFp, "%s", "instance=unique\n");
                        /*LINTED*/
                } else if (strncmp(buf, "keystore=", 9) == 0) {
                } else if (strncmp(buf, "action=", 7) == 0) {
                        (void) fprintf(zFp, "action=nocheck\n");
                } else if (strncmp(buf, "authentication=", 15) == 0) {
                        (void) fprintf(zFp, "authentication=nocheck\n");
                } else if (strncmp(buf, "conflict=", 9) == 0) {
                        (void) fprintf(zFp, "conflict=nocheck\n");
                } else if (strncmp(buf, "idepend=", 8) == 0) {
                        (void) fprintf(zFp, "idepend=nocheck\n");
                } else if (strncmp(buf, "mail=", 5) == 0) {
                        (void) fprintf(zFp, "mail=\n");
                } else if (strncmp(buf, "partial=", 8) == 0) {
                        (void) fprintf(zFp, "partial=nocheck\n");
                } else if (strncmp(buf, "rdepend=", 8) == 0) {
                        (void) fprintf(zFp, "rdepend=nocheck\n");
                } else if (strncmp(buf, "runlevel=", 9) == 0) {
                        (void) fprintf(zFp, "runlevel=nocheck\n");
                } else if (strncmp(buf, "setuid=", 7) == 0) {
                        (void) fprintf(zFp, "setuid=nocheck\n");
                } else if (strncmp(buf, "space=", 6) == 0) {
                        (void) fprintf(zFp, "space=nocheck\n");
                } else {
                        (void) fprintf(zFp, "%s", buf);
                }
        }
```

```
261         /* close admin file and return success */

263         (void) fclose(zFp);
264         return (B_TRUE);
265 }

267 /*
268  * Name:        z_brands_are_implemented
269  * Description: Determine if any branded zones may be present
270  * Arguments:   void
271  * Returns:     boolean_t
272  *                      == B_TRUE - branded zones are supported
273  *                      == B_FALSE - branded zones are not supported
274  */

276 boolean_t
277 z_brands_are_implemented(void)
278 {
279 static  boolean_t       _brandsImplementedDetermined = B_FALSE;
280 static  boolean_t       _brandsAreImplemented = B_FALSE;

282         /* if availability has not been determined, cache it now */

284         if (!_brandsImplementedDetermined) {
285                 _brandsImplementedDetermined = B_TRUE;
286                 _brandsAreImplemented = _z_brands_are_implemented();
287                 if (_brandsAreImplemented) {
288                         _z_echoDebug(DBG_BRANDS_ARE_IMPLEMENTED);
289                 } else {
290                         _z_echoDebug(DBG_BRANDS_NOT_IMPLEMENTED);
291                 }
292         }

294         /* return cached answer */

296         return (_brandsAreImplemented);
297 }

299 /*
300  * Name:        z_free_zone_list
301  * Description: free contents of zoneList_t object
302  * Arguments:   a_zlst - handle to zoneList_t object to free
303  * Returns:     void
304  */

306 void
307 z_free_zone_list(zoneList_t a_zlst)
308 {
309         int     numzones;

311         /* ignore empty list */

313         if (a_zlst == (zoneList_t)NULL) {
314                 return;
315         }

317         /* free each entry in the zone list */

319         for (numzones = 0; a_zlst[numzones]._zlName != (char *)NULL;
320             numzones++) {
321                 zoneListElement_t *zelm = &a_zlst[numzones];

323                 /* free zone name string */

325                 free(zelm->_zlName);
```



```
327                 /* free zonepath string */

329                 if (zelm->_zlPath != (char *)NULL) {
330                         free(zelm->_zlPath);
331                 }

333         }

335         /* free handle to the list */

337         free(a_zlst);
338 }

340 /*
341  * Name:        z_get_nonglobal_zone_list
342  * Description: return zoneList_t object describing all non-global
343  *              native zones - branded zones are not included in list
344  * Arguments:   None.
345  * Returns:     zoneList_t
346  *                      == NULL - error, list could not be generated
347  *                      != NULL - success, list returned
348  * NOTE:        Any zoneList_t returned is placed in new storage for the
349  *              calling function. The caller must use 'z_free_zone_list' to
350  *              dispose of the storage once the list is no longer needed.
351  */

353 zoneList_t
354 z_get_nonglobal_zone_list(void)
355 {
356         zoneList_t zones;
357         zoneBrandList_t *brands = NULL;

359         if ((brands = z_make_brand_list("native cluster", " ")) == NULL)
360                 return (NULL);

362         zones = z_get_nonglobal_zone_list_by_brand(brands);

364         z_free_brand_list(brands);

366         return (zones);
367 }

369 /*
370  * Name:        z_free_brand_list
371  * Description: Free contents of zoneBrandList_t object
372  * Arguments:   brands - pointer to zoneBrandList_t object to free
373  * Returns:     void
374  */
375 void
376 z_free_brand_list(zoneBrandList_t *brands)
377 {
378         while (brands != NULL) {
379                 zoneBrandList_t *temp = brands;
380                 free(brands->string_ptr);
381                 brands = brands->next;
382                 free(temp);
383         }
384 }

386 /*
387  * Name:        z_make_brand_list
388  * Description: Given a string with a list of brand name delimited by
389  *              the delimeter passed in, build a zoneBrandList_t structure
390  *              with the list of brand names and return it to the caller.
391  * Arguments:
```

```
 *               brands - const char pointer to string list of brand names
 *               delim - const char pointer to string representing the
 *                       delimeter for brands string.
 * Returns:      zoneBrandList_t *
 *                       == NULL - error, list could not be generated
 *                       != NULL - success, list returned
 * NOTE:         Any zoneBrandList_t returned is placed in new storage for the
 *               calling function.  The caller must use 'z_free_brand_list' to
 *               dispose of the storage once the list is no longer needed.
 */
zoneBrandList_t *
z_make_brand_list(const char *brands, const char *delim)
{
        zoneBrandList_t *brand = NULL, *head = NULL;
        char            *blist = NULL;
        char            *str = NULL;

        if ((blist = strdup(brands)) == NULL)
                return (NULL);

        if ((str = strtok(blist, delim)) != NULL) {
                if ((brand = (zoneBrandList_t *)
                    malloc(sizeof (struct _zoneBrandList))) == NULL) {
                        return (NULL);
                }

                head = brand;
                brand->string_ptr = strdup(str);
                brand->next = NULL;

                while ((str = strtok(NULL, delim)) != NULL) {
                        if ((brand->next = (zoneBrandList_t *)
                            malloc(sizeof (struct _zoneBrandList))) == NULL) {
                                return (NULL);
                        }

                        brand = brand->next;
                        brand->string_ptr = strdup(str);
                        brand->next = NULL;
                }
        }

        free(blist);
        return (head);
}

static zoneList_t
i_get_nonglobal_branded_zone_list(boolean_t (*include)(struct zoneent *,
    void *), void *arg)
/*
 * Name:         z_get_nonglobal_zone_list_by_brand
 * Description: return zoneList_t object describing all non-global
 *               zones matching the list of brands passed in.
 * Arguments:    brands - The list of zone brands to look for.
 * Returns:      zoneList_t
 *                       == NULL - error, list could not be generated
 *                       != NULL - success, list returned
 * NOTE:         Any zoneList_t returned is placed in new storage for the
 *               calling function. The caller must use 'z_free_zone_list' to
 *               dispose of the storage once the list is no longer needed.
 */
zoneList_t
z_get_nonglobal_zone_list_by_brand(zoneBrandList_t *brands)
{
        FILE            *zoneIndexFP;
        int             numzones = 0;
```



```
        struct zoneent  *ze;
        zoneList_t      zlst = NULL;
        FILE            *mapFP;
        char            zonename[ZONENAME_MAX];
        zone_spec_t     *zent;

        /* if zones are not implemented, return empty list */

        if (!z_zones_are_implemented()) {
                return ((zoneList_t)NULL);
        }

        /*
         * Open the zone index file.  Note that getzoneent_private() handles
         * NULL.
         */
        zoneIndexFP = setzoneent();

        mapFP = zonecfg_open_scratch("", B_FALSE);

        /* index file open; scan all zones; see if any are at least installed */

        while ((ze = getzoneent_private(zoneIndexFP)) != NULL) {
                zone_state_t    st;

                /* skip the global zone */

                if (strcmp(ze->zone_name, GLOBAL_ZONENAME) == 0) {
                        free(ze);
                        continue;
                }

                /*
                 * skip any zones the filter function doesn't like
                 * skip any zones with brands not on the brand list
                 */
                if (include != NULL && !include(ze, arg)) {
                if (!z_is_zone_brand_in_list(ze->zone_name, brands)) {
                        free(ze);
                        continue;
                }

                /*
                 * If the user specified an explicit zone list, then ignore any
                 * zones that aren't on that list.
                 */
                if ((zent = _z_global_data._zone_spec) != NULL) {
                        while (zent != NULL) {
                                if (strcmp(zent->zl_name, ze->zone_name) == 0)
                                        break;
                                zent = zent->zl_next;
                        }
                        if (zent == NULL) {
                                free(ze);
                                continue;
                        }
                }

                /* non-global zone: create entry for this zone */

                if (numzones == 0) {
                        zlst = (zoneList_t)_z_calloc(
                            sizeof (zoneListElement_t)*2);
                } else {
                        zlst = (zoneList_t)_z_realloc(zlst,
                            sizeof (zoneListElement_t)*(numzones+2));
```

```
508                         (void) memset(&zlst[numzones], 0L,
509                             sizeof (zoneListElement_t)*2);
510                 }
511
512                 /*
513                  * remember the zone name, zonepath and the current
514                  * zone state of the zone.
515                  */
516                 zlst[numzones]._zlName = _z_strdup(ze->zone_name);
517                 zlst[numzones]._zlPath = _z_strdup(ze->zone_path);
518                 zlst[numzones]._zlOrigInstallState = ze->zone_state;
519                 zlst[numzones]._zlCurrInstallState = ze->zone_state;
520
521                 /* get the zone kernel status */
522
523                 if (zone_get_state(ze->zone_name, &st) != Z_OK) {
524                         st = ZONE_STATE_INCOMPLETE;
525                 }
526
527                 _z_echoDebug(DBG_ZONES_NGZ_LIST_STATES,
528                     ze->zone_name, ze->zone_state, st);
529
530                 /*
531                  * For a scratch zone, we need to know the kernel zone name.
532                  */
533                 if (zonecfg_in_alt_root() && mapFP != NULL &&
534                     zonecfg_find_scratch(mapFP, ze->zone_name,
535                     zonecfg_get_root(), zonename, sizeof (zonename)) != -1) {
536                         free(zlst[numzones]._zlScratchName);
537                         zlst[numzones]._zlScratchName = _z_strdup(zonename);
538                 }
539
540                 /*
541                  * remember the current kernel status of the zone.
542                  */
543
544                 zlst[numzones]._zlOrigKernelStatus = st;
545                 zlst[numzones]._zlCurrKernelStatus = st;
546
547                 numzones++;
548                 free(ze);
549         }
550
551         /* close the index file */
552         endzoneent(zoneIndexFP);
553
554         if (mapFP != NULL)
555                 zonecfg_close_scratch(mapFP);
556
557         /* return generated list */
558
559         return (zlst);
560 }
561
562 /*
563  * Name:        z_get_nonglobal_branded_zone_list
564  * Description: return zoneList_t object describing all non-global
565  * Returns:     zoneList_t
566  *                      == NULL - error, list could not be generated
567  *                      != NULL - success, list returned
568  * NOTE:        Any zoneList_t returned is placed in new storage for the
569  *              calling function. The caller must use 'z_free_zone_list' to
570  *              dispose of the storage once the list is no longer needed.
571  */
572 zoneList_t
573 z_get_nonglobal_branded_zone_list(void)
```



```
574 {
575         return (i_get_nonglobal_branded_zone_list(NULL, NULL));
576 }
577
578 static boolean_t
579 X(struct zoneent *ze, void *arg)
580 {
581         zoneBrandList_t *brands = arg;
582
583         return (z_is_zone_brand_in_list(ze->zone_name, brands));
584 }
585
586 /*
587  * Name:        z_get_nonglobal_zone_list_by_brand
588  * Description: return zoneList_t object describing all non-global
589  *              zones matching the list of brands passed in.
590  * Arguments:   brands - The list of zone brands to look for.
591  * Returns:     zoneList_t
592  *                      == NULL - error, list could not be generated
593  *                      != NULL - success, list returned
594  * NOTE:        Any zoneList_t returned is placed in new storage for the
595  *              calling function. The caller must use 'z_free_zone_list' to
596  *              dispose of the storage once the list is no longer needed.
597  */
598 zoneList_t
599 z_get_nonglobal_zone_list_by_brand(zoneBrandList_t *brands)
600 {
601         return (i_get_nonglobal_branded_zone_list(X, brands));
602 }
603
604 /*
605 #endif /* ! codereview */
606  * Name:        z_get_zonename
607  * Description: return the name of the current zone
608  * Arguments:   void
609  * Returns:     char *
610  *                      - pointer to string representing the name of the current
611  *                      zone
612  * NOTE:        Any string returned is placed in new storage for the
613  *              calling function. The caller must use 'Free' to dispose
614  *              of the storage once the string is no longer needed.
615  */
616
617 char *
618 z_get_zonename(void)
619 {
620         ssize_t         zonenameLen;
621         char            zonename[ZONENAME_MAX];
622         zoneid_t        zoneid = (zoneid_t)-1;
623
624         /* if zones are not implemented, return "" */
625
626         if (!z_zones_are_implemented()) {
627                 return (_z_strdup(""));
628         }
629
630         /* get the zone i.d. of the current zone */
631
632         zoneid = getzoneid();
633
634         /* get the name of the current zone */
635
636         zonenameLen = getzonenamebyid(zoneid, zonename, sizeof (zonename));
637
638         /* return "" if could not get zonename */
```

```
640         if (zonenameLen < 1) {
641                 return (_z_strdup(""));
642         }

644         return (_z_strdup(zonename));
645 }

647 /*
648  * Name:         z_global_only
649  * Description: Determine if the global zone is only zone on the spec list.
650  * Arguments:   None
651  * Returns:     B_TRUE if global zone is the only zone on the list,
652  *              B_FALSE otherwise.
653  */

655 boolean_t
656 z_global_only(void)
657 {
658         /* return true if zones are not implemented - treate as global zone */

660         if (!z_zones_are_implemented()) {
661                 return (B_TRUE);
662         }

664         /* return true if this is the global zone */

666         if (_z_global_data._zone_spec != NULL &&
667             _z_global_data._zone_spec->zl_next == NULL &&
668             strcmp(_z_global_data._zone_spec->zl_name, GLOBAL_ZONENAME) == 0) {
669                 return (B_TRUE);
670         }

672         /* return false - not the global zone */

674         return (B_FALSE);
675 }

677 /*
678  * Name:         z_lock_this_zone
679  * Description: lock this zone
680  * Arguments:   a_lflags - [RO, *RO] - (ZLOCKS_T)
681  *                      Flags indicating which locks to acquire
682  * Returns:     boolean_t
683  *                      == B_TRUE - success specified locks acquired
684  *                      == B_FALSE - failure specified locks not acquired
685  * NOTE: the lock objects for "this zone" are maintained internally.
686  */

688 boolean_t
689 z_lock_this_zone(ZLOCKS_T a_lflags)
690 {
691         boolean_t       b;
692         char            *zoneName;
693         pid_t           pid = (pid_t)0;

695         /* entry assertions */

697         assert(a_lflags != ZLOCKS_NONE);

699         /* entry debugging info */

701         _z_echoDebug(DBG_ZONES_LCK_THIS, a_lflags);

703         zoneName = z_get_zonename();
704         pid = getpid();
```



```
706         /* lock zone administration */

708         if (a_lflags & ZLOCKS_ZONE_ADMIN) {
709                 b = _z_lock_zone_object(&_z_global_data._z_ObjectLocks,
710                     zoneName, LOBJ_ZONEADMIN, pid,
711                     MSG_ZONES_LCK_THIS_ZONEADM,
712                     ERR_ZONES_LCK_THIS_ZONEADM);
713                 if (!b) {
714                         (void) free(zoneName);
715                         return (B_FALSE);
716                 }
717         }

719         /* lock package administration always */

721         if (a_lflags & ZLOCKS_PKG_ADMIN) {
722                 b = _z_lock_zone_object(&_z_global_data._z_ObjectLocks,
723                     zoneName, LOBJ_PKGADMIN, pid,
724                     MSG_ZONES_LCK_THIS_PKGADM,
725                     ERR_ZONES_LCK_THIS_PKGADM);
726                 if (!b) {
727                         (void) z_unlock_this_zone(a_lflags);
728                         (void) free(zoneName);
729                         return (B_FALSE);
730                 }
731         }

733         (void) free(zoneName);

735         return (B_TRUE);
736 }

738 /*
739  * Name:         z_lock_zones
740  * Description: lock specified zones
741  * Arguments:   a_zlst - zoneList_t object describing zones to lock
742  *              a_lflags - [RO, *RO] - (ZLOCKS_T)
743  *                      Flags indicating which locks to acquire
744  * Returns:     boolean_t
745  *                      == B_TRUE - success, zones locked
746  *                      == B_FALSE - failure, zones not locked
747  */

749 boolean_t
750 z_lock_zones(zoneList_t a_zlst, ZLOCKS_T a_lflags)
751 {
752         boolean_t       b;
753         int             i;

755         /* entry assertions */

757         assert(a_lflags != ZLOCKS_NONE);

759         /* entry debugging info */

761         _z_echoDebug(DBG_ZONES_LCK_ZONES, a_lflags);

763         /* if zones are not implemented, return TRUE */

765         if (z_zones_are_implemented() == B_FALSE) {
766                 _z_echoDebug(DBG_ZONES_LCK_ZONES_UNIMP);
767                 return (B_TRUE);
768         }

770         /* lock this zone first before locking other zones */
```

```
772         b = z_lock_this_zone(a_lflags);
773         if (b == B_FALSE) {
774                 return (b);
775         }

777         /* ignore empty list */

779         if (a_zlst == (zoneList_t)NULL) {
780                 _z_echoDebug(DBG_ZONES_LCK_ZONES_NOZONES);
781                 return (B_FALSE);
782         }

784         /* zones exist */

786         _z_echoDebug(DBG_ZONES_LCK_ZONES_EXIST);

788         /*
789          * lock each listed zone that is currently running
790          */

792         for (i = 0; (a_zlst[i]._zlName != (char *)NULL); i++) {
793                 /* ignore zone if already locked */
794                 if (a_zlst[i]._zlStatus & ZST_LOCKED) {
795                         continue;
796                 }

798                 /* ignore zone if not running */
799                 if (a_zlst[i]._zlCurrKernelStatus != ZONE_STATE_RUNNING &&
800                     a_zlst[i]._zlCurrKernelStatus != ZONE_STATE_MOUNTED) {
801                         continue;
802                 }

804                 /*
805                  * mark zone locked - if interrupted out during lock, an attempt
806                  * will be made to release the lock
807                  */
808                 a_zlst[i]._zlStatus |= ZST_LOCKED;

810                 /* lock this zone */
811                 b = _z_lock_zone(&a_zlst[i], a_lflags);

813                 /* on failure unlock all zones and return error */
814                 if (b != B_TRUE) {
815                         _z_program_error(ERR_ZONES_LCK_ZONES_FAILED,
816                             a_zlst[i]._zlName);
817                         (void) z_unlock_zones(a_zlst, a_lflags);
818                         return (B_FALSE);
819                 }
820         }

822         /* success */

824         return (B_TRUE);
825 }

827 /*
828  * Name:        z_mount_in_lz
829  * Description: Mount global zone directory in specified zone's root file system
830  * Arguments:   r_lzMountPoint - pointer to handle to string - on success, the
831  *                      full path to the mount point relative to the global zone
832  *                      root file system is returned here - this is needed to
833  *                      unmount the directory when it is no longer needed
834  *              r_lzRootPath - pointer to handle to string - on success, the
835  *                      full path to the mount point relative to the specified
836  *                      zone's root file system is returned here - this is
837  *                      passed to any command executing in the specified zone to
```



```
838  *                      access the directory mounted
839  *              a_zoneName - pointer to string representing the name of the zone
840  *                      to mount the specified global zone directory in
841  *              a_gzPath - pointer to string representing the full absolute path
842  *                      of the global zone directory to LOFS mount inside of the
843  *                      specified non-global zone
844  *              a_mountPointPrefix - pointer to string representing the prefix
845  *                      to be used when creating the mount point name in the
846  *                      specified zone's root directory
847  * Returns:     boolean_t
848  *                      == B_TRUE - global zone directory mounted successfully
849  *                      == B_FALSE - failed to mount directory in specified zone
850  * NOTE:        Any strings returned is placed in new storage for the
851  *              calling function. The caller must use 'Free' to dispose
852  *              of the storage once the strings are no longer needed.
853  */

855 boolean_t
856 z_mount_in_lz(char **r_lzMountPoint, char **r_lzRootPath, char *a_zoneName,
857         char *a_gzPath, char *a_mountPointPrefix)
858 {
859         char            lzRootPath[MAXPATHLEN] = {'\0'};
860         char            uuid[MAXPATHLEN] = {'\0'};
861         char            gzMountPoint[MAXPATHLEN] = {'\0'};
862         char            lzMountPoint[MAXPATHLEN] = {'\0'};
863         hrtime_t        hretime;
864         int             err;
865         int             slen;
866         struct tm       tstruct;
867         time_t          thetime;
868         zoneid_t        zid;

870         /* entry assertions */

872         assert(a_zoneName != (char *)NULL);
873         assert(*a_zoneName != '\0');
874         assert(a_gzPath != (char *)NULL);
875         assert(*a_gzPath != '\0');
876         assert(r_lzMountPoint != (char **)NULL);
877         assert(r_lzRootPath != (char **)NULL);

879         /* entry debugging info */

881         _z_echoDebug(DBG_ZONES_MOUNT_IN_LZ_ENTRY, a_zoneName, a_gzPath);

883         /* reset returned non-global zone mount point path handle */

885         *r_lzMountPoint = (char *)NULL;
886         *r_lzRootPath = (char *)NULL;

888         /* if zones are not implemented, return FALSE */

890         if (z_zones_are_implemented() == B_FALSE) {
891                 return (B_FALSE);
892         }

894         /* error if global zone path is not absolute */

896         if (*a_gzPath != '/') {
897                 _z_program_error(ERR_GZPATH_NOT_ABSOLUTE, a_gzPath);
898                 return (B_FALSE);
899         }

901         /* error if global zone path does not exist */

903         if (_z_is_directory(a_gzPath) != 0) {
```

```
904                 _z_program_error(ERR_GZPATH_NOT_DIR, a_gzPath, strerror(errno));
905                 return (B_FALSE);
906         }

908         /* verify that specified non-global zone exists */

910         err = zone_get_id(a_zoneName, &zid);
911         if (err != Z_OK) {
912                 _z_program_error(ERR_GET_ZONEID, a_zoneName,
913                     zonecfg_strerror(err));
914                 return (B_FALSE);
915         }

917         /* obtain global zone path to non-global zones root file system */

919         err = zone_get_rootpath(a_zoneName, lzRootPath, sizeof (lzRootPath));
920         if (err != Z_OK) {
921                 _z_program_error(ERR_NO_ZONE_ROOTPATH, a_zoneName,
922                     zonecfg_strerror(err));
923                 return (B_FALSE);
924         }

926         if (lzRootPath[0] == '\0') {
927                 _z_program_error(ERR_ROOTPATH_EMPTY, a_zoneName);
928                 return (B_FALSE);
929         }

931         /*
932          * lofs resolve the non-global zone's root path first in case
933          * its in a path that's been lofs mounted read-only.
934          */
935         z_resolve_lofs(lzRootPath, sizeof (lzRootPath));

937         /* verify that the root path exists */

939         if (_z_is_directory(lzRootPath) != 0) {
940                 _z_program_error(ERR_LZROOT_NOTDIR, lzRootPath,
941                     strerror(errno));
942                 return (B_FALSE);
943         }

945         /*
946          * generate a unique key - the key is the same length as unique uid
947          * but contains different information that is as unique as can be made;
948          * include current hires time (nanosecond real timer). Such a unique
949          * i.d. will look like:
950          *              0203104092-1145345-0004e94d6af481a0
951          */

953         hretime = gethrtime();

955         thetime = time((time_t *)NULL);
956         (void) localtime_r(&thetime, &tstruct);

958         slen = snprintf(uuid, sizeof (uuid),
959             UUID_FORMAT,
960             tstruct.tm_mday, tstruct.tm_mon, tstruct.tm_year,
961             tstruct.tm_yday, tstruct.tm_hour, tstruct.tm_min,
962             tstruct.tm_sec,     tstruct.tm_wday, hretime);
963         if (slen > sizeof (uuid)) {
964                 _z_program_error(ERR_GZMOUNT_SNPRINTFUUID_FAILED,
965                     UUID_FORMAT, sizeof (uuid));
966                 return (B_FALSE);
967         }

969         /* create the global zone mount point */
```



```
971         slen = snprintf(gzMountPoint, sizeof (gzMountPoint), "%s/.SUNW_%s_%s",
972             lzRootPath,
973             a_mountPointPrefix ? a_mountPointPrefix : "zones", uuid);
974         if (slen > sizeof (gzMountPoint)) {
975                 _z_program_error(ERR_GZMOUNT_SNPRINTFGMP_FAILED,
976                     "%s/.SUNW_%s_%s", lzRootPath,
977                     a_mountPointPrefix ? a_mountPointPrefix : "zones",
978                     uuid, sizeof (gzMountPoint));
979                 return (B_FALSE);
980         }

982         slen = snprintf(lzMountPoint, sizeof (lzMountPoint), "%s",
983             gzMountPoint+strlen(lzRootPath));
984         if (slen > sizeof (lzMountPoint)) {
985                 _z_program_error(ERR_GZMOUNT_SNPRINTFLMP_FAILED,
986                     "%s", gzMountPoint+strlen(lzRootPath),
987                     sizeof (lzMountPoint));
988                 return (B_FALSE);
989         }

991         _z_echoDebug(DBG_MNTPT_NAMES, a_gzPath, a_zoneName, gzMountPoint,
992             lzMountPoint);

994         /* error if the mount point already exists */

996         if (_z_is_directory(gzMountPoint) == 0) {
997                 _z_program_error(ERR_ZONEROOT_NOTDIR, gzMountPoint,
998                     a_zoneName, strerror(errno));
999                 return (B_FALSE);
1000        }

1002        /* create the temporary mount point */

1004        if (mkdir(gzMountPoint, 0600) != 0) {
1005                _z_program_error(ERR_MNTPT_MKDIR, gzMountPoint, a_zoneName,
1006                    strerror(errno));
1007                return (B_FALSE);
1008        }

1010        /* mount the global zone path on the non-global zone root file system */

1012        err = mount(a_gzPath, gzMountPoint, MS_RDONLY|MS_DATA, "lofs",
1013            (char *)NULL, 0, (char *)NULL, 0);
1014        if (err != 0) {
1015                _z_program_error(ERR_GZMOUNT_FAILED, a_gzPath,
1016                    gzMountPoint, a_zoneName, strerror(errno));
1017                return (B_FALSE);
1018        }

1020        /* success - return both mountpoints to caller */

1022        *r_lzMountPoint = _z_strdup(gzMountPoint);

1024        *r_lzRootPath = _z_strdup(lzMountPoint);

1026        /* return success */

1028        return (B_TRUE);
1029 }

1031 /*
1032  * Name:        z_non_global_zones_exist
1033  * Description: Determine if any non-global native zones exist
1034  * Arguments:   None.
1035  * Returns:     boolean_t
```

```
 *      == B_TRUE - at least one non-global native zone exists
 *      == B_FALSE - no non-global native zone exists
 */

boolean_t
z_non_global_zones_exist(void)
{
        FILE            *zoneIndexFP;
        boolean_t       anyExist = B_FALSE;
        struct zoneent  *ze;
        zone_spec_t     *zent;

        /* if zones are not implemented, return FALSE */

        if (z_zones_are_implemented() == B_FALSE) {
                return (B_FALSE);
        }

        /* determine if any zones are configured */
        zoneIndexFP = setzoneent();
        if (zoneIndexFP == NULL) {
                return (B_FALSE);
        }

        /* index file open; scan all zones; see if any are at least installed */

        while ((ze = getzoneent_private(zoneIndexFP)) != NULL) {
                /*
                 * If the user specified an explicit zone list, then ignore any
                 * zones that aren't on that list.
                 */
                if ((zent = _z_global_data._zone_spec) != NULL) {
                        while (zent != NULL) {
                                if (strcmp(zent->zl_name, ze->zone_name) == 0)
                                        break;
                                zent = zent->zl_next;
                        }
                        if (zent == NULL) {
                                free(ze);
                                continue;
                        }
                }

                /* skip the global zone */
                if (strcmp(ze->zone_name, GLOBAL_ZONENAME) == 0) {
                        free(ze);
                        continue;
                }

                /* skip any branded zones */
                if (z_is_zone_branded(ze->zone_name)) {
                        free(ze);
                        continue;
                }

                /* is this zone installed? */
                if (ze->zone_state >= ZONE_STATE_INSTALLED) {
                        free(ze);
                        anyExist = B_TRUE;
                        break;
                }
                free(ze);
        }

        /* close the index file */
```



```
        endzoneent(zoneIndexFP);

        /* return results */

        return (anyExist);
}

/*
 * Name:        z_on_zone_spec
 * Description: Determine if named zone is on the zone_spec list.
 * Arguments:   Pointer to name to test.
 * Returns:     B_TRUE if named zone is on the list or if the user specified
 *              no list at all (all zones is the default), B_FALSE otherwise.
 */

boolean_t
z_on_zone_spec(const char *zonename)
{
        zone_spec_t     *zent;

        /* entry assertions */

        assert(zonename != NULL);
        assert(*zonename != '\0');

        /* return true if zones not implemented or no zone spec list defined */

        if (!z_zones_are_implemented() || _z_global_data._zone_spec == NULL) {
                return (B_TRUE);
        }

        /* return true if named zone is on the zone spec list */

        for (zent = _z_global_data._zone_spec;
            zent != NULL; zent = zent->zl_next) {
                if (strcmp(zent->zl_name, zonename) == 0)
                        return (B_TRUE);
        }

        /* named zone is not on the zone spec list */

        return (B_FALSE);
}

/*
 * Name:        z_running_in_global_zone
 * Description: Determine if running in the "global" zone
 * Arguments:   void
 * Returns:     boolean_t
 *                      == B_TRUE - running in global zone
 *                      == B_FALSE - not running in global zone
 */

boolean_t
z_running_in_global_zone(void)
{
        static  boolean_t       _zoneIdDetermined = B_FALSE;
        static  boolean_t       _zoneIsGlobal = B_FALSE;

        /* if ID has not been determined, cache it now */

        if (!_zoneIdDetermined) {
                _zoneIdDetermined = B_TRUE;
                _zoneIsGlobal = _z_running_in_global_zone();
        }
```

```
1168         return (_zoneIsGlobal);
1169 }

1171 /*
1172  * Name:        z_set_output_functions
1173  * Description: Link program specific output functions to this library.
1174  * Arguments:   a_echo_fcn - (_z_printf_fcn_t)
1175  *                      Function to call to cause "normal operation" messages
1176  *                      to be output/displayed
1177  *              a_echo_debug_fcn - (_z_printf_fcn_t)
1178  *                      Function to call to cause "debugging" messages
1179  *                      to be output/displayed
1180  *              a_progerr_fcn - (_z_printf_fcn_t)
1181  *                      Function to call to cause "program error" messages
1182  *                      to be output/displayed
1183  * Returns:     void
1184  * NOTE:        If NULL is specified for any function, then the functionality
1185  *              associated with that function is disabled.
1186  * NOTE:        The function pointers provided must call a function that
1187  *              takes two arguments:
1188  *                      function(char *format, char *message)
1189  *              Any registered function will be called like:
1190  *                      function("%s", "message")
1191  */

1193 void
1194 z_set_output_functions(_z_printf_fcn_t a_echo_fcn,
1195     _z_printf_fcn_t a_echo_debug_fcn,
1196     _z_printf_fcn_t a_progerr_fcn)
1197 {
1198         _z_global_data._z_echo = a_echo_fcn;
1199         _z_global_data._z_echo_debug = a_echo_debug_fcn;
1200         _z_global_data._z_progerr = a_progerr_fcn;
1201 }

1203 /*
1204  * Name:        z_set_zone_root
1205  * Description: Set root for zones library operations
1206  * Arguments:   Path to root of boot environment containing zone; must be
1207  *              absolute.
1208  * Returns:     None.
1209  * NOTE:        Must be called before performing any zone-related operations.
1210  *              (Currently called directly by set_inst_root() during -R
1211  *              argument handling.)
1212  */

1214 void
1215 z_set_zone_root(const char *zroot)
1216 {
1217         char *rootdir;

1219         /* if zones are not implemented, just return */

1221         if (!z_zones_are_implemented())
1222                 return;

1224         /* entry assertions */

1226         assert(zroot != NULL);

1228         rootdir = _z_strdup((char *)zroot);
1229         z_canoninplace(rootdir);

1231         if (strcmp(rootdir, "/") == 0) {
1232                 rootdir[0] = '\0';
1233         }
```



```
1235         /* free any existing cached root path */
1236         if (*_z_global_data._z_root_dir != '\0') {
1237                 free(_z_global_data._z_root_dir);
1238                 _z_global_data._z_root_dir = NULL;
1239         }

1241         /* store duplicate of new zone root path */

1243         if (*rootdir != '\0') {
1244                 _z_global_data._z_root_dir = _z_strdup(rootdir);
1245         } else {
1246                 _z_global_data._z_root_dir = "";
1247         }

1249         /* set zone root path */

1251         zonecfg_set_root(rootdir);

1253         free(rootdir);
1254 }

1256 /*
1257  * Name:        z_set_zone_spec
1258  * Description: Set list of zones on which actions will be performed.
1259  * Arguments:   Whitespace-separated list of zone names.
1260  * Returns:     0 on success, -1 on error.
1261  * NOTES:       Will call _z_program_error if argument can't be parsed or
1262  *              memory not available.
1263  */

1265 int
1266 z_set_zone_spec(const char *zlist)
1267 {
1268         const char      *zend;
1269         ptrdiff_t       zlen;
1270         zone_spec_t     *zent;
1271         zone_spec_t     *zhead;
1272         zone_spec_t     **znextp = &zhead;

1274         /* entry assertions */

1276         assert(zlist != NULL);

1278         /* parse list to zone_spec_t list, store in global data */

1280         for (;;) {
1281                 while (isspace(*zlist)) {
1282                         zlist++;
1283                 }
1284                 if (*zlist == '\0') {
1285                         break;
1286                 }
1287                 for (zend = zlist; *zend != '\0'; zend++) {
1288                         if (isspace(*zend)) {
1289                                 break;
1290                         }
1291                 }
1292                 zlen = ((ptrdiff_t)zend) - ((ptrdiff_t)zlist);
1293                 if (zlen >= ZONENAME_MAX) {
1294                         _z_program_error(ERR_ZONE_NAME_ILLEGAL, zlen, zlist);
1295                         return (-1);
1296                 }
1297                 zent = _z_malloc(sizeof (*zent));
1298                 (void) memcpy(zent->zl_name, zlist, zlen);
1299                 zent->zl_name[zlen] = '\0';
```

```
1300                zent->zl_used = B_FALSE;
1301                *znextp = zent;
1302                znextp = &zent->zl_next;
1303                zlist = zend;
1304        }
1305        *znextp = NULL;

1307        if (zhead == NULL) {
1308                _z_program_error(ERR_ZONE_LIST_EMPTY);
1309                return (-1);
1310        }

1312        _z_global_data._zone_spec = zhead;
1313        return (0);
1314 }

1316 /*
1317  * Name:        z_umount_lz_mount
1318  * Description: Unmount directory mounted with z_mount_in_lz
1319  * Arguments:   a_lzMountPointer - pointer to string returned by z_mount_in_lz
1320  * Returns:     boolean_t
1321  *                      == B_TRUE - successfully unmounted directory
1322  *                      == B_FALSE - failed to unmount directory
1323  */

1325 boolean_t
1326 z_umount_lz_mount(char *a_lzMountPoint)
1327 {
1328        int     err;

1330        /* entry assertions */

1332        assert(a_lzMountPoint != (char *)NULL);
1333        assert(*a_lzMountPoint != '\0');

1335        /* entry debugging info */

1337        _z_echoDebug(DBG_ZONES_UNMOUNT_FROM_LZ_ENTRY, a_lzMountPoint);

1339        /* if zones are not implemented, return TRUE */

1341        if (z_zones_are_implemented() == B_FALSE) {
1342                return (B_FALSE);
1343        }

1345        /* error if global zone path is not absolute */

1347        if (*a_lzMountPoint != '/') {
1348                _z_program_error(ERR_LZMNTPT_NOT_ABSOLUTE, a_lzMountPoint);
1349                return (B_FALSE);
1350        }

1352        /* verify mount point exists */

1354        if (_z_is_directory(a_lzMountPoint) != 0) {
1355                _z_program_error(ERR_LZMNTPT_NOTDIR, a_lzMountPoint,
1356                    strerror(errno));
1357                return (B_FALSE);
1358        }

1360        /* unmount */

1362        err = umount2(a_lzMountPoint, 0);
1363        if (err != 0) {
1364                _z_program_error(ERR_GZUMOUNT_FAILED, a_lzMountPoint,
1365                    strerror(errno));
```



```
1366                return (B_FALSE);
1367        }

1369        /* remove the mount point */

1371        (void) remove(a_lzMountPoint);

1373        /* return success */

1375        return (B_TRUE);
1376 }

1378 /*
1379  * Name:        z_unlock_this_zone
1380  * Description: unlock this zone
1381  * Arguments:   a_lflags - [RO, *RO] - (ZLOCKS_T)
1382  *                      Flags indicating which locks to release
1383  * Returns:     boolean_t
1384  *                      == B_TRUE - success specified locks released
1385  *                      == B_FALSE - failure specified locks may not be released
1386  * NOTE: the lock objects for "this zone" are maintained internally.
1387  */

1389 boolean_t
1390 z_unlock_this_zone(ZLOCKS_T a_lflags)
1391 {
1392        boolean_t       b;
1393        boolean_t       errors = B_FALSE;
1394        char            *zoneName;

1396        /* entry assertions */

1398        assert(a_lflags != ZLOCKS_NONE);

1400        /* entry debugging info */

1402        _z_echoDebug(DBG_ZONES_ULK_THIS, a_lflags);

1404        /* return if no objects locked */

1406        if ((_z_global_data._z_ObjectLocks == (char *)NULL) ||
1407            (*_z_global_data._z_ObjectLocks == '\0')) {
1408                return (B_TRUE);
1409        }

1411        zoneName = z_get_zonename();

1413        /* unlock package administration */

1415        if (a_lflags & ZLOCKS_PKG_ADMIN) {
1416                b = _z_unlock_zone_object(&_z_global_data._z_ObjectLocks,
1417                    zoneName, LOBJ_PKGADMIN, ERR_ZONES_ULK_THIS_PACKAGE);
1418                if (!b) {
1419                        errors = B_TRUE;
1420                }
1421        }

1423        /* unlock zone administration */

1425        if (a_lflags & ZLOCKS_ZONE_ADMIN) {
1426                b = _z_unlock_zone_object(&_z_global_data._z_ObjectLocks,
1427                    zoneName, LOBJ_ZONEADMIN, ERR_ZONES_ULK_THIS_ZONES);
1428                if (!b) {
1429                        errors = B_TRUE;
1430                }
1431        }
```

```
1433         (void) free(zoneName);
1434         return (!errors);
1435 }

1437 /*
1438  * Name:        z_unlock_zones
1439  * Description: unlock specified zones
1440  * Arguments:   a_zlst - zoneList_t object describing zones to unlock
1441  *              a_lflags - [RO, *RO] - (ZLOCKS_T)
1442  *                      Flags indicating which locks to release
1443  * Returns:     boolean_t
1444  *                      == B_TRUE - success, zones unlocked
1445  *                      == B_FALSE - failure, zones not unlocked
1446  */

1448 boolean_t
1449 z_unlock_zones(zoneList_t a_zlst, ZLOCKS_T a_lflags)
1450 {
1451         boolean_t       b;
1452         boolean_t       errors = B_FALSE;
1453         int             i;

1455         /* entry assertions */

1457         assert(a_lflags != ZLOCKS_NONE);

1459         /* entry debugging info */

1461         _z_echoDebug(DBG_ZONES_ULK_ZONES, a_lflags);

1463         /* if zones are not implemented, return TRUE */

1465         if (z_zones_are_implemented() == B_FALSE) {
1466                 _z_echoDebug(DBG_ZONES_ULK_ZONES_UNIMP);
1467                 return (B_TRUE);
1468         }

1470         /* ignore empty list */

1472         if (a_zlst == (zoneList_t)NULL) {
1473                 _z_echoDebug(DBG_ZONES_ULK_ZONES_NOZONES);
1474                 /* unlock this zone before returning */
1475                 return (z_unlock_this_zone(a_lflags));
1476         }

1478         /* zones exist */

1480         _z_echoDebug(DBG_ZONES_ULK_ZONES_EXIST);

1482         /*
1483          * unlock each listed zone that is currently running
1484          */

1486         for (i = 0; (a_zlst[i]._zlName != (char *)NULL); i++) {
1487                 /* ignore zone if not locked */
1488                 if (!(a_zlst[i]._zlStatus & ZST_LOCKED)) {
1489                         continue;
1490                 }

1492                 /* ignore zone if not running */
1493                 if (a_zlst[i]._zlCurrKernelStatus != ZONE_STATE_RUNNING &&
1494                     a_zlst[i]._zlCurrKernelStatus != ZONE_STATE_MOUNTED) {
1495                         continue;
1496                 }
```



```
1498                 /* unlock this zone */
1499                 b = _z_unlock_zone(&a_zlst[i], a_lflags);

1501                 if (b != B_TRUE) {
1502                         errors = B_TRUE;
1503                 } else {
1504                         /* mark zone as unlocked */
1505                         a_zlst[i]._zlStatus &= ~ZST_LOCKED;
1506                 }
1507         }

1509         /* unlock this zone */

1511         if (z_unlock_this_zone(a_lflags) != B_TRUE) {
1512                 errors = B_TRUE;
1513         }

1515         return (errors);
1516 }

1518 /*
1519  * Name:        z_verify_zone_spec
1520  * Description: Verify list of zones on which actions will be performed.
1521  * Arguments:   None.
1522  * Returns:     0 on success, -1 on error.
1523  * NOTES:       Will call _z_program_error if there are zones on the specified
1524  *              list that don't exist on the system. Requires that
1525  *              z_set_zone_root is called first (if it is called at all).
1526  */

1528 int
1529 z_verify_zone_spec(void)
1530 {
1531         FILE            *zoneIndexFP;
1532         boolean_t       errors;
1533         char            zoneIndexPath[MAXPATHLEN];
1534         struct zoneent  *ze;
1535         zone_spec_t     *zent;

1537         if (!z_zones_are_implemented()) {
1538                 _z_program_error(ERR_ZONES_NOT_IMPLEMENTED);
1539                 return (-1);
1540         }

1542         zoneIndexFP = setzoneent();
1543         if (zoneIndexFP == NULL) {
1544                 _z_program_error(ERR_ZONEINDEX_OPEN, zoneIndexPath,
1545                     strerror(errno));
1546                 return (-1);
1547         }

1549         while ((ze = getzoneent_private(zoneIndexFP)) != NULL) {
1550                 for (zent = _z_global_data._zone_spec;
1551                     zent != NULL; zent = zent->zl_next) {
1552                         if (strcmp(zent->zl_name, ze->zone_name) == 0) {
1553                                 zent->zl_used = B_TRUE;
1554                                 break;
1555                         }
1556                 }
1557                 free(ze);
1558         }
1559         endzoneent(zoneIndexFP);

1561         errors = B_FALSE;
1562         for (zent = _z_global_data._zone_spec;
1563             zent != NULL; zent = zent->zl_next) {
```

```
1564                 if (!zent->zl_used) {
1565                         _z_program_error(ERR_ZONE_NONEXISTENT, zent->zl_name);
1566                         errors = B_TRUE;
1567                 }
1568         }
1569         return (errors ? -1 : 0);
1570 }

1572 /*
1573  * Name:        z_zlist_change_zone_state
1574  * Description: Change the current state of the specified zone
1575  * Arguments:   a_zlst - handle to zoneList_t object describing all zones
1576  *              a_zoneIndex - index into a_zlst of the zone to return the
1577  *              a_newState - the state to put the specified zone in
1578  * Returns:     boolean_t
1579  *                      == B_TRUE - the zone is in the new state
1580  *                      == B_FALSE - unable to transition the zone to the
1581  *                                   specified state
1582  * NOTE:        This changes the "current kernel" state of the specified
1583  *              zone. For example, to boot the zone, change the state
1584  *              to "ZONE_STATE_RUNNING". To halt the zone, change the
1585  *              state to "ZONE_STATE_INSTALLED".
1586  */

1588 boolean_t
1589 z_zlist_change_zone_state(zoneList_t a_zlst, int a_zoneIndex,
1590         zone_state_t a_newState)
1591 {
1592         int     i;

1594         /* entry debugging info */

1596         _z_echoDebug(DBG_ZONES_CHG_Z_STATE_ENTRY, a_zoneIndex, a_newState);

1598         /* ignore empty list */

1600         if (a_zlst == (zoneList_t)NULL) {
1601                 return (B_FALSE);
1602         }

1604         /* find the specified zone in the list */

1606         for (i = 0; (i != a_zoneIndex) &&
1607             (a_zlst[i]._zlName != (char *)NULL); i++)
1608                 ;

1610         /* return error if the specified zone does not exist */

1612         if (a_zlst[i]._zlName == (char *)NULL) {
1613                 return (B_FALSE);
1614         }

1616         /* return success if the zone is already in this state */

1618         if (a_zlst[i]._zlCurrKernelStatus == a_newState) {
1619                 return (B_TRUE);
1620         }

1622         /* take action on new state to set zone to */

1624         _z_echoDebug(DBG_ZONES_CHG_Z_STATE, a_zlst[i]._zlName,
1625             a_zlst[i]._zlCurrKernelStatus, a_newState);

1627         switch (a_newState) {
1628         case ZONE_STATE_RUNNING:
1629         case ZONE_STATE_MOUNTED:
```



```
1630                 /* these states mean "boot the zone" */
1631                 return (_z_make_zone_running(&a_zlst[i]));

1633         case ZONE_STATE_DOWN:
1634         case ZONE_STATE_INSTALLED:
1635                 /* these states mean "halt the zone" */
1636                 return (_z_make_zone_down(&a_zlst[i]));

1638         case ZONE_STATE_READY:
1639                 return (_z_make_zone_ready(&a_zlst[i]));

1641         case ZONE_STATE_CONFIGURED:
1642         case ZONE_STATE_INCOMPLETE:
1643         case ZONE_STATE_SHUTTING_DOWN:
1644         default:
1645                 /* do not know how to change zone to this state */
1646                 return (B_FALSE);
1647         }
1648 }

1650 /*
1651  * Name:        z_is_zone_branded
1652  * Description: Determine whether zone has a non-native brand
1653  * Arguments:   a_zoneName - name of the zone to check for branding
1654  * Returns:     boolean_t
1655  *                      == B_TRUE - zone has a non-native brand
1656  *                      == B_FALSE - zone is native
1657  */
1658 boolean_t
1659 z_is_zone_branded(char *zoneName)
1660 {
1661         char                    brandname[MAXNAMELEN];
1662         int                     err;

1664         /* if zones are not implemented, return FALSE */
1665         if (!z_zones_are_implemented()) {
1666                 return (B_FALSE);
1667         }

1669         /* if brands are not implemented, return FALSE */
1670         if (!z_brands_are_implemented()) {
1671                 return (B_FALSE);
1672         }

1674         err = zone_get_brand(zoneName, brandname, sizeof (brandname));
1675         if (err != Z_OK) {
1676                 _z_program_error(ERR_BRAND_GETBRAND, zonecfg_strerror(err));
1677                 return (B_FALSE);
1678         }

1680         /*
1681          * Both "native" and "cluster" are native brands
1682          * that use the standard facilities in the areas
1683          * of packaging/installation/update.
1684          */
1685         if (streq(brandname, NATIVE_BRAND_NAME) ||
1686             streq(brandname, CLUSTER_BRAND_NAME)) {
1687                 return (B_FALSE);
1688         } else {
1689                 return (B_TRUE);
1690         }
1691 }

1693 /*
1694  * Name:        z_is_zone_brand_in_list
1695  * Description: Determine whether zone's brand has a match in the list
```

```
 *              brands passed in.
 * Arguments:   zoneName - name of the zone to check for branding
 *              list - list of brands to check the zone against
 * Returns:     boolean_t
 *                      == B_TRUE - zone has a matching brand
 *                      == B_FALSE - zone brand is not in list
 */
boolean_t
z_is_zone_brand_in_list(char *zoneName, zoneBrandList_t *list)
{
        char                    brandname[MAXNAMELEN];
        int                     err;
        zoneBrandList_t         *sp;

        if (zoneName == NULL || list == NULL)
                return (B_FALSE);

        /* if zones are not implemented, return FALSE */
        if (!z_zones_are_implemented()) {
                return (B_FALSE);
        }

        /* if brands are not implemented, return FALSE */
        if (!z_brands_are_implemented()) {
                return (B_FALSE);
        }

        err = zone_get_brand(zoneName, brandname, sizeof (brandname));
        if (err != Z_OK) {
                _z_program_error(ERR_BRAND_GETBRAND, zonecfg_strerror(err));
                return (B_FALSE);
        }

        for (sp = list; sp != NULL; sp = sp->next) {
                if (sp->string_ptr != NULL &&
                    strcmp(sp->string_ptr, brandname) == 0) {
                        return (B_TRUE);
                }
        }

        return (B_FALSE);
}

/*
 * Name:        z_zlist_get_current_state
 * Description: Determine the current kernel state of the specified zone
 * Arguments:   a_zlst - handle to zoneList_t object describing all zones
 *              a_zoneIndex - index into a_zlst of the zone to return
 * Returns:     zone_state_t
 *                      The current state of the specified zone is returned
 */

zone_state_t
z_zlist_get_current_state(zoneList_t a_zlst, int a_zoneIndex)
{
        int     i;

        /* ignore empty list */

        if (a_zlst == (zoneList_t)NULL) {
                return (ZONE_STATE_INCOMPLETE);
        }

        /* find the specified zone in the list */

        for (i = 0; (i != a_zoneIndex) &&
```



```
            (a_zlst[i]._zlName != (char *)NULL); i++)
                ;

        /* return error if the specified zone does not exist */

        if (a_zlst[i]._zlName == (char *)NULL) {
                return (ZONE_STATE_INCOMPLETE);
        }

        /* return selected zone's current kernel state */

        _z_echoDebug(DBG_ZONES_GET_ZONE_STATE,
            a_zlst[i]._zlName ? a_zlst[i]._zlName : "",
            a_zlst[i]._zlCurrKernelStatus);

        return (a_zlst[i]._zlCurrKernelStatus);
}

/*
 * Name:        z_zlist_get_original_state
 * Description: Return the original kernal state of the specified zone
 * Arguments:   a_zlst - handle to zoneList_t object describing all zones
 *              a_zoneIndex - index into a_zlst of the zone to return the
 * Returns:     zone_state_t
 *                      The original state of the specified zone is returned.
 *                      This is the state of the zone when the zoneList_t
 *                      object was first generated.
 */

zone_state_t
z_zlist_get_original_state(zoneList_t a_zlst, int a_zoneIndex)
{
        int     i;

        /* ignore empty list */

        if (a_zlst == (zoneList_t)NULL) {
                return (ZONE_STATE_INCOMPLETE);
        }

        /* find the specified zone in the list */

        for (i = 0; (i != a_zoneIndex) &&
            (a_zlst[i]._zlName != (char *)NULL); i++)
                ;

        /* return error if the specified zone does not exist */

        if (a_zlst[i]._zlName == (char *)NULL) {
                return (ZONE_STATE_INCOMPLETE);
        }

        /* return selected zone's original kernel state */

        return (a_zlst[i]._zlOrigKernelStatus);
}

/*
 * Name:        z_zlist_get_scratch
 * Description: Determine name of scratch zone
 * Arguments:   a_zlst - handle to zoneList_t object describing all zones
 *              a_zoneIndex - index into a_zlst of the zone to use
 * Return:      char *
 *                      == NULL - zone name could not be determined
 *                      != NULL - pointer to string representing scratch zone
 * NOTE:        Any name returned is placed in static storage that must
```

```
 *              NEVER be free()ed by the caller.
 */

char *
z_zlist_get_scratch(zoneList_t a_zlst, int a_zoneIndex)
{
        int     i;

        /* ignore empty list */

        if (a_zlst == NULL)
                return (NULL);

        /* find the specified zone in the list */

        for (i = 0; i != a_zoneIndex; i++) {
                if (a_zlst[i]._zlName == NULL)
                        return (NULL);
        }

        /* return selected zone's scratch name */

        return (a_zlst[i]._zlScratchName == NULL ? a_zlst[i]._zlName :
            a_zlst[i]._zlScratchName);
}

/*
 * Name:        z_zlist_get_zonename
 * Description: Determine name of specified zone
 * Arguments:   a_zlst - handle to zoneList_t object describing all zones
 *              a_zoneIndex - index into a_zlst of the zone to return the
 * Return:      char *
 *                      == NULL - zone name could not be determined
 *                      != NULL - pointer to string representing zone name
 * NOTE:        Any zoneList_t returned is placed in static storage that must
 *              NEVER be free()ed by the caller.
 */

char *
z_zlist_get_zonename(zoneList_t a_zlst, int a_zoneIndex)
{
        int     i;

        /* ignore empty list */

        if (a_zlst == (zoneList_t)NULL) {
                return ((char *)NULL);
        }

        /* find the specified zone in the list */

        for (i = 0; (i != a_zoneIndex) &&
            (a_zlst[i]._zlName != (char *)NULL); i++)
                ;

        /* return error if the specified zone does not exist */

        if (a_zlst[i]._zlName == (char *)NULL) {
                return (NULL);
        }

        /* return selected zone's name */

        return (a_zlst[i]._zlName);
}
```



```
/*
 * Name:        z_zlist_get_zonepath
 * Description: Determine zonepath of specified zone
 * Arguments:   a_zlst - handle to zoneList_t object describing all zones
 *              a_zoneIndex - index into a_zlst of the zone to return
 * Return:      char *
 *                      == NULL - zonepath could not be determined
 *                      != NULL - pointer to string representing zonepath
 * NOTE:        Any zoneList_t returned is placed in static storage that must
 *              NEVER be free()ed by the caller.
 */

char *
z_zlist_get_zonepath(zoneList_t a_zlst, int a_zoneIndex)
{
        int     i;

        /* ignore empty list */

        if (a_zlst == (zoneList_t)NULL) {
                return ((char *)NULL);
        }

        /* find the specified zone in the list */

        for (i = 0; (i != a_zoneIndex) &&
            (a_zlst[i]._zlName != (char *)NULL); i++)
                ;

        /* return error if the specified zone does not exist */

        if (a_zlst[i]._zlName == (char *)NULL) {
                return (NULL);
        }

        /* return selected zone's zonepath */

        return (a_zlst[i]._zlPath);
}

int
z_zlist_is_zone_auto_create_be(zoneList_t zlst, int idx, boolean_t *ret)
{
        char brandname[MAXNAMELEN];
        brand_handle_t bh;

        if (zone_get_brand(z_zlist_get_zonename(zlst, idx), brandname,
            sizeof (brandname)) != Z_OK)
                return (-1);

        bh = brand_open(brandname);
        if (bh == NULL)
                return (-1);

        *ret = brand_auto_create_be(bh);

        brand_close(bh);

        return (0);
#endif /* ! codereview */
}

boolean_t
z_zlist_is_zone_runnable(zoneList_t a_zlst, int a_zoneIndex)
{
        int     i;
```

```
1961         /* if zones are not implemented, return error */

1963         if (z_zones_are_implemented() == B_FALSE) {
1964                 return (B_FALSE);
1965         }

1967         /* ignore empty list */

1969         if (a_zlst == (zoneList_t)NULL) {
1970                 return (B_FALSE);
1971         }

1973         /* find the specified zone in the list */

1975         for (i = 0; (i != a_zoneIndex) &&
1976             (a_zlst[i]._zlName != (char *)NULL); i++)
1977                 ;

1979         /* return error if the specified zone does not exist */

1981         if (a_zlst[i]._zlName == (char *)NULL) {
1982                 return (B_FALSE);
1983         }

1985         /* choose based on current state */

1987         switch (a_zlst[i]._zlCurrKernelStatus) {
1988         case ZONE_STATE_RUNNING:
1989         case ZONE_STATE_MOUNTED:
1990                 /* already running */
1991                 return (B_TRUE);

1993         case ZONE_STATE_INSTALLED:
1994         case ZONE_STATE_DOWN:
1995         case ZONE_STATE_READY:
1996         case ZONE_STATE_SHUTTING_DOWN:
1997                 /* return false if the zone cannot be booted */

1999                 if (a_zlst[i]._zlStatus & ZST_NOT_BOOTABLE) {
2000                         return (B_FALSE);
2001                 }

2003                 return (B_TRUE);

2005         case ZONE_STATE_CONFIGURED:
2006         case ZONE_STATE_INCOMPLETE:
2007         default:
2008                 /* cannot transition (boot) these states */
2009                 return (B_FALSE);
2010         }
2011 }

2013 /*
2014  * Name:        z_zlist_restore_zone_state
2015  * Description: Return the zone to the state it was originally in
2016  * Arguments:   a_zlst - handle to zoneList_t object describing all zones
2017  *              a_zoneIndex - index into a_zlst of the zone to return the
2018  * Returns:     boolean_t
2019  *                      == B_TRUE - the zone's state has been restored
2020  *                      == B_FALSE - unable to transition the zone to its
2021  *                              original state
2022  */

2024 boolean_t
2025 z_zlist_restore_zone_state(zoneList_t a_zlst, int a_zoneIndex)
```



```
2026 {
2027         int             i;

2029         /* ignore empty list */

2031         if (a_zlst == (zoneList_t)NULL) {
2032                 return (B_FALSE);
2033         }

2035         /* find the specified zone in the list */

2037         for (i = 0; (i != a_zoneIndex) &&
2038             (a_zlst[i]._zlName != (char *)NULL); i++)
2039                 ;

2041         /* return error if the specified zone does not exist */

2043         if (a_zlst[i]._zlName == (char *)NULL) {
2044                 return (B_FALSE);
2045         }

2047         /* transition the zone back to its original state */

2049         return (z_zlist_change_zone_state(a_zlst,
2050             a_zoneIndex, a_zlst[i]._zlOrigKernelStatus));
2051 }

2053 /*
2054  * Name:        z_zone_exec
2055  * Description: Execute a Unix command in a specified zone and return results
2056  * Arguments:   a_zoneName - pointer to string representing the name of the zone
2057  *                      to execute the specified command in
2058  *              a_path - pointer to string representing the full path *in the
2059  *                      non-global zone named by a_zoneName* of the Unix command
2060  *                      to be executed
2061  *              a_argv[] - Pointer to array of character strings representing
2062  *                      the arguments to be passed to the Unix command. The list
2063  *                      must be termianted with an element that is (char *)NULL
2064  *              NOTE: a_argv[0] is the "command name" passed to the command
2065  *              a_stdoutPath - Pointer to string representing the path to a file
2066  *                      into which all output to "stdout" from the Unix command
2067  *                      is placed.
2068  *                      == (char *)NULL - leave stdout open and pass through
2069  *                      == "/dev/null" - discard stdout output
2070  *              a_strerrPath - Pointer to string representing the path to a file
2071  *                      into which all output to "stderr" from the Unix command
2072  *                      is placed.
2073  *                      == (char *)NULL - leave stderr open and pass through
2074  *                      == "/dev/null" - discard stderr output
2075  *              a_fds - Pointer to array of integers representing file
2076  *                      descriptors to remain open during the call - all
2077  *                      file descriptors above STDERR_FILENO not in this
2078  *                      list will be closed.
2079  * Returns:     int
2080  *                      The return (exit) code from the specified Unix command
2081  *                      Special return codes:
2082  *                      -1 : failure to exec process
2083  *                      -2 : could not create contract for greenline
2084  *                      -3 : fork() failed
2085  *                      -4 : could not open stdout capture file
2086  *                      -5 : error from 'waitpid' other than EINTR
2087  *                      -6 : zones are not supported
2088  * NOTE:        All file descriptores other than 0, 1 and 2 are closed except
2089  *              for those file descriptors listed in the a_fds array.
2090  */
```

```
2092 int
2093 z_zone_exec(const char *a_zoneName, const char *a_path, char *a_argv[],
2094         char *a_stdoutPath, char *a_stderrPath, int *a_fds)
2095 {
2096         int                     final_status;
2097         int                     lerrno;
2098         int                     status;
2099         int                     tmpl_fd;
2100         pid_t                   child_pid;
2101         pid_t                   result_pid;
2102         struct sigaction        nact;
2103         struct sigaction        oact;
2104         void                    (*funcSighup)();
2105         void                    (*funcSigint)();

2107         /* if zones are not implemented, return TRUE */

2109         if (z_zones_are_implemented() == B_FALSE) {
2110                 return (-6);    /* -6 : zones are not supported */
2111         }

2113         if ((tmpl_fd = _zexec_init_template()) == -1) {
2114                 _z_program_error(ERR_CANNOT_CREATE_CONTRACT, strerror(errno));
2115                 return (-2);    /* -2 : could not create greenline contract */
2116         }

2118         /*
2119          * hold SIGINT/SIGHUP signals and reset signal received counter;
2120          * after the fork1() the parent and child need to setup their respective
2121          * interrupt handling and release the hold on the signals
2122          */

2124         (void) sighold(SIGINT);
2125         (void) sighold(SIGHUP);

2127         _z_global_data._z_SigReceived = 0;      /* no signals received */

2129         /*
2130          * fork off a new process to execute command in;
2131          * fork1() is used instead of vfork() so the child process can
2132          * perform operations that would modify the parent process if
2133          * vfork() were used
2134          */

2136         child_pid = fork1();

2138         if (child_pid < 0) {
2139                 /*
2140                  * ************************************************************
2141                  * fork failed!
2142                  * ************************************************************
2143                  */

2145                 (void) ct_tmpl_clear(tmpl_fd);
2146                 (void) close(tmpl_fd);
2147                 _z_program_error(ERR_FORK, strerror(errno));

2149                 /* release hold on signals */

2151                 (void) sigrelse(SIGHUP);
2152                 (void) sigrelse(SIGINT);

2154                 return (-3);    /* -3 : fork() failed */
2155         }

2157         if (child_pid == 0) {
```



```
2158                 int     i;

2160                 /*
2161                  * ************************************************************
2162                  * This is the forked (child) process
2163                  * ************************************************************
2164                  */

2166                 (void) ct_tmpl_clear(tmpl_fd);
2167                 (void) close(tmpl_fd);

2169                 /* reset any signals to default */

2171                 for (i = 0; i < NSIG; i++) {
2172                         (void) sigset(i, SIG_DFL);
2173                 }

2175                 /*
2176                  * close all file descriptors not in the a_fds list
2177                  */

2179                 (void) fdwalk(&_z_close_file_descriptors, (void *)a_fds);

2181                 /*
2182                  * if a file for stdout is present, open the file and use the
2183                  * file to capture stdout from the _zexec process
2184                  */

2186                 if (a_stdoutPath != (char *)NULL) {
2187                         int     stdoutfd;

2189                         stdoutfd = open(a_stdoutPath,
2190                             O_WRONLY|O_CREAT|O_TRUNC, 0600);
2191                         if (stdoutfd < 0) {
2192                                 _z_program_error(ERR_CAPTURE_FILE, a_stdoutPath,
2193                                     strerror(errno));
2194                                 return (-4);
2195                         }

2197                         (void) dup2(stdoutfd, STDOUT_FILENO);
2198                         (void) close(stdoutfd);
2199                 }

2201                 /*
2202                  * if a file for stderr is present, open the file and use the
2203                  * file to capture stderr from the _zexec process
2204                  */

2206                 if (a_stderrPath != (char *)NULL) {
2207                         int     stderrfd;

2209                         stderrfd = open(a_stderrPath,
2210                             O_WRONLY|O_CREAT|O_TRUNC, 0600);
2211                         if (stderrfd < 0) {
2212                                 _z_program_error(ERR_CAPTURE_FILE, a_stderrPath,
2213                                     strerror(errno));
2214                                 return (-4);
2215                         }

2217                         (void) dup2(stderrfd, STDERR_FILENO);
2218                         (void) close(stderrfd);
2219                 }

2221                 /* release all held signals */

2223                 (void) sigrelse(SIGHUP);
```

```
2224                (void) sigrelse(SIGINT);

2226                /* execute command in the specified non-global zone */

2228                _exit(_zexec(a_zoneName, a_path, a_argv));
2229        }

2231        /*
2232         * ********************************************************************
2233         * This is the forking (parent) process
2234         * ********************************************************************
2235         */

2237        /* register child process i.d. so signal handlers can pass signal on */

2239        _z_global_data._z_ChildProcessId = child_pid;

2241        /*
2242         * setup signal handlers for SIGINT and SIGHUP and release hold
2243         */

2245        /* hook SIGINT to _z_sig_trap() */

2247        nact.sa_handler = _z_sig_trap;
2248        nact.sa_flags = SA_RESTART;
2249        (void) sigemptyset(&nact.sa_mask);

2251        if (sigaction(SIGINT, &nact, &oact) < 0) {
2252                funcSigint = SIG_DFL;
2253        } else {
2254                funcSigint = oact.sa_handler;
2255        }

2257        /* hook SIGHUP to _z_sig_trap() */

2259        nact.sa_handler = _z_sig_trap;
2260        nact.sa_flags = SA_RESTART;
2261        (void) sigemptyset(&nact.sa_mask);

2263        if (sigaction(SIGHUP, &nact, &oact) < 0) {
2264                funcSighup = SIG_DFL;
2265        } else {
2266                funcSighup = oact.sa_handler;
2267        }

2269        /* release hold on signals */

2271        (void) sigrelse(SIGHUP);
2272        (void) sigrelse(SIGINT);

2274        (void) ct_tmpl_clear(tmpl_fd);
2275        (void) close(tmpl_fd);

2277        /*
2278         * wait for the process to exit, reap child exit status
2279         */

2281        for (;;) {
2282                result_pid = waitpid(child_pid, &status, 0L);
2283                lerrno = (result_pid == -1 ? errno : 0);

2285                /* break loop if child process status reaped */

2287                if (result_pid != -1) {
2288                        break;
2289                }
```



```
2291                /* break loop if not interrupted out of waitpid */

2293                if (errno != EINTR) {
2294                        break;
2295                }
2296        }

2298        /* reset child process i.d. so signal handlers do not pass signals on */

2300        _z_global_data._z_ChildProcessId = -1;

2302        /*
2303         * If the child process terminated due to a call to exit(), then
2304         * set results equal to the 8-bit exit status of the child process;
2305         * otherwise, set the exit status to "-1" indicating that the child
2306         * exited via a signal.
2307         */

2309        if (WIFEXITED(status)) {
2310                final_status = WEXITSTATUS(status);
2311                if ((_z_global_data._z_SigReceived != 0) &&
2312                    (final_status == 0)) {
2313                        final_status = 1;
2314                }
2315        } else {
2316                final_status = -1;      /* -1 : failure to exec process */
2317        }

2319        /* determine proper exit code */

2321        if (result_pid == -1) {
2322                final_status = -5;      /* -5 : error from waitpid not EINTR */
2323        } else if (_z_global_data._z_SigReceived != 0) {
2324                final_status = -7;      /* -7 : interrupt received */
2325        }

2327        /*
2328         * reset signal handlers
2329         */

2331        /* reset SIGINT */

2333        nact.sa_handler = funcSigint;
2334        nact.sa_flags = SA_RESTART;
2335        (void) sigemptyset(&nact.sa_mask);

2337        (void) sigaction(SIGINT, &nact, (struct sigaction *)NULL);

2339        /* reset SIGHUP */

2341        nact.sa_handler = funcSighup;
2342        nact.sa_flags = SA_RESTART;
2343        (void) sigemptyset(&nact.sa_mask);

2345        (void) sigaction(SIGHUP, &nact, (struct sigaction *)NULL);

2347        /*
2348         * if signal received during command execution, interrupt
2349         * this process now.
2350         */

2352        if (_z_global_data._z_SigReceived != 0) {
2353                (void) kill(getpid(), SIGINT);
2354        }
```

```
2356         /* set errno and return */

2358         errno = lerrno;

2360         return (final_status);
2361 }

2363 /*
2364  * Name:         z_zones_are_implemented
2365  * Description: Determine if any zone operations can be performed
2366  * Arguments:   void
2367  * Returns:     boolean_t
2368  *                      == B_TRUE - zone operations are available
2369  *                      == B_FALSE - no zone operations can be done
2370  */

2372 boolean_t
2373 z_zones_are_implemented(void)
2374 {
2375         static  boolean_t       _zonesImplementedDetermined = B_FALSE;
2376         static  boolean_t       _zonesAreImplemented = B_FALSE;

2378         /* if availability has not been determined, cache it now */

2380         if (!_zonesImplementedDetermined) {
2381                 _zonesImplementedDetermined = B_TRUE;
2382                 _zonesAreImplemented = _z_zones_are_implemented();
2383                 if (!_zonesAreImplemented) {
2384                         _z_echoDebug(DBG_ZONES_NOT_IMPLEMENTED);
2385                 } else {
2386                         _z_echoDebug(DBG_ZONES_ARE_IMPLEMENTED);
2387                 }
2388         }

2390         return (_zonesAreImplemented);
2391 }
```

```
**********************************************************
    5655 Wed Nov 11 10:43:16 2015
new/usr/src/lib/libinstzones/hdrs/instzones_api.h
patch zone-auto-create-be
**********************************************************
```

```
/*
 * 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 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

#ifndef _INSTZONES_API_H
#define _INSTZONES_API_H


/*
 * Module:      instzones_api.h
 * Group:       libinstzones
 * Description: This module contains the libinstzones API data structures,
 *              constants, and function prototypes.
 */

/*
 * required includes
 */

/* System includes */

#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <termios.h>
#include <sys/mman.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <libzonecfg.h>

/*
 * C++ prefix
 */

#ifdef __cplusplus
extern "C" {
#endif
```



```
/* function prototypes */

/* PRINTFLIKE1 */
typedef void (*_z_printf_fcn_t)(char *a_format, ...);

/* zone list structure */

typedef struct _zoneListElement_t *zoneList_t;

/* zone brand list structure */

typedef struct _zoneBrandList zoneBrandList_t;

/* flag for zone locking functions */

typedef unsigned long ZLOCKS_T;

/* flags for zone locking */

#define ZLOCKS_ZONE_ADMIN       ((ZLOCKS_T)0x00000001)  /* zone admin */
#define ZLOCKS_PKG_ADMIN        ((ZLOCKS_T)0x00000002)  /* package admin */
#define ZLOCKS_ALL              ((ZLOCKS_T)0xFFFFFFFF)  /* all locks */
#define ZLOCKS_NONE             ((ZLOCKS_T)0x00000000)  /* no locks */

/*
 * external function definitions
 */

/* zones.c */

extern boolean_t        z_zones_are_implemented(void);
extern void             z_set_zone_root(const char *zroot);
extern int              z_zlist_is_zone_auto_create_be(zoneList_t, int,
                                boolean_t *);
#endif /* ! codereview */
extern boolean_t        z_zlist_is_zone_runnable(zoneList_t a_zoneList,
                                int a_zoneIndex);
extern boolean_t        z_zlist_restore_zone_state(zoneList_t a_zoneList,
                                int a_zoneIndex);
extern boolean_t        z_zlist_change_zone_state(zoneList_t a_zoneList,
                                int a_zoneIndex, zone_state_t a_newState);
extern char             *z_get_zonename(void);
extern zone_state_t     z_zlist_get_current_state(zoneList_t a_zoneList,
                                int a_zoneIndex);
extern zone_state_t     z_zlist_get_original_state(zoneList_t a_zoneList,
                                int a_zoneIndex);
extern int              z_zoneExecCmdArray(int *r_status, char **r_results,
                                char *a_inputFile, char *a_path, char *a_argv[],
                                const char *a_zoneName, int *a_fds);
extern int              z_zone_exec(const char *zonename, const char *path,
                                char *argv[], char *a_stdoutPath,
                                char *a_stderrPath, int *a_fds);
extern boolean_t        z_create_zone_admin_file(char *a_zoneAdminFilename,
                                char *a_userAdminFilename);
extern void             z_free_zone_list(zoneList_t a_zoneList);
extern zoneList_t       z_get_nonglobal_zone_list(void);
extern zoneList_t       z_get_nonglobal_branded_zone_list(void);
#endif /* ! codereview */
extern zoneList_t       z_get_nonglobal_zone_list_by_brand(zoneBrandList_t *);
extern void             z_free_brand_list(zoneBrandList_t *a_brandList);
extern zoneBrandList_t  *z_make_brand_list(const char *brandList,
                                const char *delim);
extern boolean_t        z_lock_zones(zoneList_t a_zlst, ZLOCKS_T a_lflags);
extern boolean_t        z_non_global_zones_exist(void);
```

```
128 extern boolean_t        z_running_in_global_zone(void);
129 extern void             z_set_output_functions(_z_printf_fcn_t a_echo_fcn,
130                                 _z_printf_fcn_t a_echo_debug_fcn,
131                                 _z_printf_fcn_t a_progerr_fcn);
132 extern int              z_set_zone_spec(const char *zlist);
133 extern int              z_verify_zone_spec(void);
134 extern boolean_t        z_on_zone_spec(const char *zonename);
135 extern boolean_t        z_global_only(void);
136 extern boolean_t        z_unlock_zones(zoneList_t a_zlst, ZLOCKS_T a_lflags);
137 extern boolean_t        z_lock_this_zone(ZLOCKS_T a_lflags);
138 extern boolean_t        z_unlock_this_zone(ZLOCKS_T a_lflags);
139 extern char             *z_zlist_get_zonename(zoneList_t a_zoneList,
140                                 int a_zoneId);
141 extern char             *z_zlist_get_zonepath(zoneList_t a_zoneList,
142                                 int a_zoneId);
143 extern char             *z_zlist_get_scratch(zoneList_t a_zoneList,
144                                 int a_zoneId);
145 extern boolean_t        z_umount_lz_mount(char *a_lzMountPoint);
146 extern boolean_t        z_mount_in_lz(char **r_lzMountPoint,
147                                 char **r_lzRootPath,
148                                 char *a_zoneName, char *a_gzPath,
149                                 char *a_mountPointPrefix);
150 extern boolean_t        z_is_zone_branded(char *zoneName);
151 extern boolean_t        z_is_zone_brand_in_list(char *zoneName,
152                             zoneBrandList_t *brands);
153 extern boolean_t        z_zones_are_implemented(void);

155 /* zones_exec.c */
156 extern int              z_ExecCmdArray(int *r_status, char **r_results,
157                                 char *a_inputFile, char *a_cmd, char **a_args);
158 /*VARARGS*/
159 extern int              z_ExecCmdList(int *r_status, char **r_results,
160                                 char *a_inputFile, char *a_cmd, ...);

162 /* zones_paths.c */
163 extern char             *z_make_zone_root(char *);
164 extern void             z_path_canonize(char *file);
165 extern void             z_canoninplace(char *file);

167 /* zones_lofs.c */
168 extern void z_destroyMountTable(void);
169 extern int z_createMountTable(void);
170 extern int z_isPathWritable(const char *);
171 extern void z_resolve_lofs(char *path, size_t);

173 /* zones_states.c */
174 extern int UmountAllZones(char *mntpnt);

176 /*
177  * C++ postfix
178  */

180 #ifdef __cplusplus
181 }
182 #endif

184 #endif /* _INSTZONES_API_H */
```