

```

*****
34345 Tue Aug 6 21:14:49 2013
new/usr/src/cmd/beam/beam.c
*** NO COMMENTS ***
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /*
27  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
28  * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
29 */

30 /*
31  * System includes
32 */

34 #include <assert.h>
35 #include <stdio.h>
36 #include <strings.h>
37 #include <libzfs.h>
38 #include <locale.h>
39 #include <langinfo.h>
40 #include <stdlib.h>
41 #include <wchar.h>
42 #include <sys/types.h>

44 #include "libbe.h"

46 #ifndef lint
47 #define _(x) gettext(x)
48 #else
49 #define _(x) (x)
50 #endif

52 #ifndef TEXT_DOMAIN
53 #define TEXT_DOMAIN "SYS_TEST"
54 #endif

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

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

```

```

61 static int be_do_destroy(int argc, char **argv);
62 static int be_do_list(int argc, char **argv);
63 static int be_do_mount(int argc, char **argv);
64 static int be_do_unmount(int argc, char **argv);
65 static int be_do_rename(int argc, char **argv);
66 static int be_do_rollback(int argc, char **argv);
67 static void usage(void);

69 /*
70  * single column name/width output format description
71  */
72 struct col_info {
73     const char *col_name;
74     size_t width;
75 };
76
77 unchanged_portion_omitted
78
357 static void
358 print_be_nodes(const char *be_name, boolean_t parsable, struct hdr_info *hdr,
359               be_node_list_t *nodes)
360 {
361     char buf[64];
362     char datetime[DT_BUF_LEN];
363     be_node_list_t *cur_be;

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

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

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

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

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

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

392         if (cur_be->be_active)
393             active[ai++] = 'N';
394         if (cur_be->be_active_on_boot) {
395             if (!cur_be->be_global_active)
396                 active[ai] = 'b';
397             else
398                 if (cur_be->be_active_on_boot)
399                     active[ai] = 'R';
400         }

401         nicenum(used, buf, sizeof(buf));
402         if (parsable)
403             (void) printf("%s;%s;%s;%s;%llu;%s;%ld\n",
404                          name,

```

```
405         cur_be->be_uid_str,
406         active,
407         (cur_be->be_mounted ? mntpt: ""),
408         used,
409         cur_be->be_policy_type,
410         creation);
411     else
412         (void) printf("%-*s %-*s %-*s %-*s %-*s %-*s\n",
413             hdr->cols[0].width, name,
414             hdr->cols[1].width, active,
415             hdr->cols[2].width, (cur_be->be_mounted ? mntpt:
416             "-"),
417             hdr->cols[3].width, buf,
418             hdr->cols[4].width, cur_be->be_policy_type,
419             hdr->cols[5].width, datetime);
420     }
421 }
unchanged_portion_omitted
```

```

*****
34518 Tue Aug 6 21:14:50 2013
new/usr/src/lib/libbe/common/be_activate.c
*** NO COMMENTS ***
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /*
27  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
28 */

30 #include <assert.h>
31 #include <libintl.h>
32 #include <libnvpair.h>
33 #include <libzfs.h>
34 #include <stdio.h>
35 #include <stdlib.h>
36 #include <string.h>
37 #include <errno.h>
38 #include <sys/mnttab.h>
39 #include <sys/types.h>
40 #include <sys/stat.h>
41 #include <unistd.h>

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

46 char    *mnttab = MNITAB;

48 /*
49  * Private function prototypes
50  */
51 static int set_bootfs(char *boot_rpool, char *be_root_ds);
52 static int set_canmount(be_node_list_t *, char *);
53 static int be_do_installgrub(be_transaction_data_t *);
54 static int be_get_grub_vers(be_transaction_data_t *, char **, char **);
55 static int get_ver_from_capfile(char *, char **);
56 static int be_promote_zone_ds(char *, char *);
57 static int be_promote_ds_callback(zfs_handle_t *, void *);

59 /* ***** */
60 /*                Public Functions                */
61 /* ***** */

```

```

63 /*
64  * Function:    be_activate
65  * Description: Calls _be_activate which activates the BE named in the
66  *              attributes passed in through be_attrs. The process of
67  *              activation sets the bootfs property of the root pool, resets
68  *              the canmount property to noauto, and sets the default in the
69  *              grub menu to the entry corresponding to the entry for the named
70  *              BE.
71  * Parameters:
72  *              be_attrs - pointer to nvlist_t of attributes being passed in.
73  *              The follow attribute values are used by this function:
74  *
75  *                      BE_ATTR_ORIG_BE_NAME          *required
76  * Return:
77  *              BE_SUCCESS - Success
78  *              be_errno_t - Failure
79  * Scope:
80  *              Public
81  */
82 int
83 be_activate(nvlist_t *be_attrs)
84 {
85     int    ret = BE_SUCCESS;
86     char   *be_name = NULL;

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

92     /* Get the BE name to activate */
93     if (nvlist_lookup_string(be_attrs, BE_ATTR_ORIG_BE_NAME, &be_name)
94         != 0) {
95         be_print_err(gettext("be_activate: failed to "
96             "lookup BE_ATTR_ORIG_BE_NAME attribute\n"));
97         be_zfs_fini();
98         return (BE_ERR_INVAL);
99     }

101     /* Validate BE name */
102     if (!be_valid_be_name(be_name)) {
103         be_print_err(gettext("be_activate: invalid BE name %s\n"),
104             be_name);
105         be_zfs_fini();
106         return (BE_ERR_INVAL);
107     }

109     ret = _be_activate(be_name);

111     be_zfs_fini();

113     return (ret);
114 }

116 /* ***** */
117 /*                Semi Private Functions                */
118 /* ***** */

120 /*
121  * Function:    _be_activate
122  * Description: This does the actual work described in be_activate.
123  * Parameters:
124  *              be_name - pointer to the name of BE to activate.
125  *
126  * Return:
127  *              BE_SUCCESS - Success

```

```

128 *          be_errnot_t - Failure
129 * Scope:
130 *          Public
131 */
132 int
133 _be_activate(char *be_name)
134 {
135     be_transaction_data_t cb = { 0 };
136     zfs_handle_t *zhp = NULL;
137     char root_ds[MAXPATHLEN];
138     char active_ds[MAXPATHLEN];
139     char *cur_vers = NULL, *new_vers = NULL;
140     be_node_list_t *be_nodes = NULL;
141     uuid_t uu = {0};
142     int entry, ret = BE_SUCCESS;
143     int zret = 0;
144
145     /*
146     * TODO: The BE needs to be validated to make sure that it is actually
147     * a bootable BE.
148     */
149
150     if (be_name == NULL)
151         return (BE_ERR_INVALID);
152
153     /* Set obe_name to be_name in the cb structure */
154     cb.obename = be_name;
155
156     /* find which zpool the be is in */
157     if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &cb)) == 0) {
158         be_print_err(gettext("be_activate: failed to "
159             "find zpool for BE (%s)\n"), cb.obename);
160         return (BE_ERR_BE_NOENT);
161     } else if (zret < 0) {
162         be_print_err(gettext("be_activate: "
163             "zpool_iter failed: %s\n"),
164             libzfs_error_description(g_zfs));
165         ret = zfs_err_to_be_err(g_zfs);
166         return (ret);
167     }
168
169     be_make_root_ds(cb.obename, cb.obename, root_ds, sizeof (root_ds));
170     cb.oberoot_ds = strdup(root_ds);
171
172     if (getzoneid() == GLOBAL_ZONEID) {
173         if (be_has_grub() && (ret = be_get_grub_vers(&cb, &cur_vers,
174             &new_vers)) != BE_SUCCESS) {
175             be_print_err(gettext("be_activate: failed to get grub "
176                 "versions from capability files.\n"));
177             return (ret);
178         }
179         if (cur_vers != NULL) {
180             /*
181             * We need to check to see if the version number from
182             * the BE being activated is greater than the current
183             * one.
184             */
185             if (new_vers != NULL &&
186                 atof(cur_vers) < atof(new_vers)) {
187                 if ((ret = be_do_installgrub(&cb))
188                     != BE_SUCCESS) {
189                     free(new_vers);
190                     free(cur_vers);
191                     return (ret);
192                 }
193                 free(new_vers);

```

```

194     }
195     free(cur_vers);
196     } else if (new_vers != NULL) {
197         if ((ret = be_do_installgrub(&cb)) != BE_SUCCESS) {
198             free(new_vers);
199             return (ret);
200         }
201         free(new_vers);
202     }
203     if (!be_has_menu_entry(root_ds, cb.obename, &entry)) {
204         if ((ret = be_append_menu(cb.obename, cb.obename,
205             NULL, NULL, NULL)) != BE_SUCCESS) {
206             be_print_err(gettext("be_activate: Failed to "
207                 "add BE (%s) to the GRUB menu\n"),
208                 cb.obename);
209             goto done;
210         }
211     }
212     if (be_has_grub()) {
213         if ((ret = be_change_grub_default(cb.obename,
214             cb.obename) != BE_SUCCESS) {
215             be_print_err(gettext("be_activate: failed to "
216                 "change the default entry in menu.lst\n"));
217             goto done;
218         }
219     }
220     }
221
222     if ((ret = _be_list(cb.obename, &be_nodes)) != BE_SUCCESS) {
223         return (ret);
224     }
225
226     if ((ret = set_canmount(be_nodes, "noauto")) != BE_SUCCESS) {
227         be_print_err(gettext("be_activate: failed to set "
228             "canmount dataset property\n"));
229         goto done;
230     }
231
232     if (getzoneid() == GLOBAL_ZONEID) {
233         if ((ret = set_bootfs(be_nodes->rpool,
234             root_ds)) != BE_SUCCESS) {
235             if ((ret = set_bootfs(be_nodes->rpool, root_ds)) != BE_SUCCESS) {
236                 be_print_err(gettext("be_activate: failed to set "
237                     "bootfs pool property for %s\n"), root_ds);
238                 goto done;
239             }
240         }
241     }
242     if ((zhp = zfs_open(g_zfs, root_ds, ZFS_TYPE_FILESYSTEM)) != NULL) {
243         /*
244         * We don't need to close the zfs handle at this
245         * point because The callback funtion
246         * be_promote_ds_callback() will close it for us.
247         */
248         if (be_promote_ds_callback(zhp, NULL) != 0) {
249             be_print_err(gettext("be_activate: "
250                 "failed to activate the "
251                 "datasets for %s: %s\n"),
252                 root_ds,
253                 libzfs_error_description(g_zfs));
254             ret = BE_ERR_PROMOTE;
255             goto done;
256         }
257     } else {
258         be_print_err(gettext("be_activate: failed to open "
259             "be_activate:: failed to open "

```

```

258     "dataset (%s): %s\n"), root_ds,
259     libzfs_error_description(g_zfs));
260     ret = zfs_err_to_be_err(g_zfs);
261     goto done;
262 }

264 if (getzoneid() == GLOBAL_ZONEID &&
265     be_get_uid(cb.obe_root_ds, &uu) == BE_SUCCESS &&
266     (ret = be_promote_zone_ds(cb.obe_name, cb.obe_root_ds))
267     != BE_SUCCESS) {
268     be_print_err(gettext("be_activate: failed to promote "
269     "the active zonepath datasets for zones in BE %s\n"),
270     cb.obe_name);
271 }

273 if (getzoneid() != GLOBAL_ZONEID) {
274     if (!be_zone_compare_uids(root_ds)) {
275         be_print_err(gettext("be_activate: activating zone "
276         "root dataset from non-active global BE is not "
277         "supported\n"));
278         ret = BE_ERR_NOTSUP;
279         goto done;
280     }
281     if ((zhp = zfs_open(g_zfs, root_ds,
282     ZFS_TYPE_FILESYSTEM)) == NULL) {
283         be_print_err(gettext("be_activate: failed to open "
284         "dataset (%s): %s\n"), root_ds,
285         libzfs_error_description(g_zfs));
286         ret = zfs_err_to_be_err(g_zfs);
287         goto done;
288     }
289     /* Find current active zone root dataset */
290     if ((ret = be_find_active_zone_root(zhp, cb.obe_zpool,
291     active_ds, sizeof(active_ds))) != BE_SUCCESS) {
292         be_print_err(gettext("be_activate: failed to find "
293         "active zone root dataset\n"));
294         ZFS_CLOSE(zhp);
295         goto done;
296     }
297     /* Do nothing if requested BE is already active */
298     if (strcmp(root_ds, active_ds) == 0) {
299         ret = BE_SUCCESS;
300         ZFS_CLOSE(zhp);
301         goto done;
302     }

304     /* Set active property for BE */
305     if (zfs_prop_set(zhp, BE_ZONE_ACTIVE_PROPERTY, "on") != 0) {
306         be_print_err(gettext("be_activate: failed to set "
307         "active property (%s): %s\n"), root_ds,
308         libzfs_error_description(g_zfs));
309         ret = zfs_err_to_be_err(g_zfs);
310         ZFS_CLOSE(zhp);
311         goto done;
312     }
313     ZFS_CLOSE(zhp);

315     /* Unset active property for old active root dataset */
316     if ((zhp = zfs_open(g_zfs, active_ds,
317     ZFS_TYPE_FILESYSTEM)) == NULL) {
318         be_print_err(gettext("be_activate: failed to open "
319         "dataset (%s): %s\n"), active_ds,
320         libzfs_error_description(g_zfs));
321         ret = zfs_err_to_be_err(g_zfs);
322         goto done;
323     }

```

```

324     if (zfs_prop_set(zhp, BE_ZONE_ACTIVE_PROPERTY, "off") != 0) {
325         be_print_err(gettext("be_activate: failed to unset "
326         "active property (%s): %s\n"), active_ds,
327         libzfs_error_description(g_zfs));
328         ret = zfs_err_to_be_err(g_zfs);
329         ZFS_CLOSE(zhp);
330         goto done;
331     }
332     ZFS_CLOSE(zhp);
333 }
334 done:
335     be_free_list(be_nodes);
336     return (ret);
337 }
_____unchanged_portion_omitted_

```

```

*****
86490 Tue Aug 6 21:14:51 2013
new/usr/src/lib/libbe/common/be_create.c
*** NO COMMENTS ***
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /*
27  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
28  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
29 */

30 /*
31  * System includes
32 */

34 #include <assert.h>
35 #include <ctype.h>
36 #include <errno.h>
37 #include <libgen.h>
38 #include <libintl.h>
39 #include <libnvpair.h>
40 #include <libzfs.h>
41 #include <stdio.h>
42 #include <stdlib.h>
43 #include <string.h>
44 #include <sys/mnttab.h>
45 #include <sys/mount.h>
46 #include <sys/stat.h>
47 #include <sys/types.h>
48 #include <sys/wait.h>
49 #include <unistd.h>

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

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

57 /* Private function prototypes */
58 static int _be_destroy(const char *, be_destroy_data_t *);
59 static int be_destroy_zones(char *, char *, be_destroy_data_t *);
60 static int be_destroy_zone_roots(char *, be_destroy_data_t *);

```

```

61 static int be_destroy_zone_roots_callback(zfs_handle_t *, void *);
62 static int be_copy_zones(char *, char *, char *);
63 static int be_clone_fs_callback(zfs_handle_t *, void *);
64 static int be_destroy_callback(zfs_handle_t *, void *);
65 static int be_send_fs_callback(zfs_handle_t *, void *);
66 static int be_demote_callback(zfs_handle_t *, void *);
67 static int be_demote_find_clone_callback(zfs_handle_t *, void *);
68 static int be_has_snapshot_callback(zfs_handle_t *, void *);
69 static int be_demote_get_one_clone(zfs_handle_t *, void *);
70 static int be_get_snap(char *, char **);
71 static int be_prep_clone_send_fs(zfs_handle_t *, be_transaction_data_t *,
72     char *, int);
73 static boolean_t be_create_container_ds(char *);
74 static char *be_get_zone_be_name(char *root_ds, char *container_ds);
75 static int be_zone_root_exists_callback(zfs_handle_t *, void *);

77 /* ***** */
78 /* Public Functions */
79 /* ***** */

81 /*
82  * Function: be_init
83  * Description: Creates the initial datasets for a BE and leaves them
84  * unpopulated. The resultant BE can be mounted but can't
85  * yet be activated or booted.
86  * Parameters:
87  * be_attrs - pointer to nvlist_t of attributes being passed in.
88  * The following attributes are used by this function:
89  *
90  * BE_ATTR_NEW_BE_NAME *required
91  * BE_ATTR_NEW_BE_POOL *required
92  * BE_ATTR_ZFS_PROPERTIES *optional
93  * BE_ATTR_FS_NAMES *optional
94  * BE_ATTR_FS_NUM *optional
95  * BE_ATTR_SHARED_FS_NAMES *optional
96  * BE_ATTR_SHARED_FS_NUM *optional
97  * Return:
98  * BE_SUCCESS - Success
99  * be_errno_t - Failure
100 * Scope:
101 * Public
102 */
103 int
104 be_init(nvlist_t *be_attrs)
105 {
106     be_transaction_data_t bt = { 0 };
107     zpool_handle_t *zlp;
108     nvlist_t *zfs_props = NULL;
109     char nbe_root_ds[MAXPATHLEN];
110     char child_fs[MAXPATHLEN];
111     char **fs_names = NULL;
112     char **shared_fs_names = NULL;
113     uint16_t fs_num = 0;
114     uint16_t shared_fs_num = 0;
115     int nelemt;
116     int i;
117     int zret = 0, ret = BE_SUCCESS;

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

123     /* Get new BE name */
124     if (nvlist_lookup_string(be_attrs, BE_ATTR_NEW_BE_NAME, &bt.nbe_name)
125         != 0) {
126         be_print_err(gettext("be_init: failed to lookup ")

```

```

127         "BE_ATTR_NEW_BE_NAME attribute\n"));
128     return (BE_ERR_INVALID);
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_INVALID);
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_INVALID);
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_INVALID);
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_INVALID);
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_INVALID);
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_INVALID);
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_INVALID);
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_INVALID);
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_INVALID);
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_INVALID);
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_INVALID);
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>");
513         }
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 }

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);

547 /* Destroy this BE */
548 if ((ret = _be_destroy((const char *)bt.obe_root_ds, &dd))
549     != BE_SUCCESS) {
550     goto done;
551 }

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 }

564 done:
565     be_zfs_fini();

567     return (ret);
568 }

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:  be_attrs - pointer to nvlist_t of attributes being passed in.
579 *              The following attributes are used by this function:
580 *
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;

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

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_INVALID);
635     }

637     if ((ret = be_find_current_be(&bt)) != BE_SUCCESS) {
638         return (ret);
639     }

641     be_get_defaults(&be_defaults);

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_INVALID);
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     }
678
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_INVALID);
686     }
687
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_INVALID);
695     }
696
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_INVALID);
703     }
704
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_INVALID);
711     }
712
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_INVALID);
719     }
720
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_INVALID);
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_INVALID);
740         }
741
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     }
756
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_INVALID;
767             goto done;
768         }
769
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                 if ((zret = zpool_iter(g_zfs, be_exists_callback, bt.nbe_name)
775                     > 0) {
776                     be_print_err(gettext("be_copy: BE (%s) already "
777                         "exists\n"), bt.nbe_name);
778                     ret = BE_ERR_BE_EXISTS;
779                     goto done;
780                 } else if (zret < 0) {
781                     be_print_err(gettext("be_copy: zpool_iter "

```



```

912     "orig BE (%s)\n"),
913     bt.nbe_name, bt.obe_name);
914     ret = BE_ERR_CLONE;
915     goto done;
916 }
917
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 } else {
997
998     /* Do copy (i.e. send BE datasets via zfs_send/recv) */
999
1000    /*
1001     * Verify BE container dataset in nbe_zpool exists.
1002     * If not, create it.
1003     */
1004    if (!be_create_container_ds(bt.nbe_zpool)) {
1005        ret = BE_ERR_CREATDS;
1006        goto done;
1007    }
1008
1009    /*
1010     * Iterate through original BE's datasets and send
1011     * them to the other pool. This call will end up closing
1012     * the zfs handle passed in whether it succeeds or fails.
1013     */
1014    if ((ret = be_send_fs_callback(zhp, &bt)) != 0) {
1015        be_print_err(gettext("be_copy: failed to "
1016            "send BE (%s) to pool (%s)\n"), bt.obe_name,
1017            bt.nbe_zpool);
1018        ret = BE_ERR_COPY;
1019        zhp = NULL;
1020        goto done;
1021    }
1022    zhp = NULL;
1023
1024    /*
1025     * Set flag to note that the dataset(s) for the new BE have been
1026     * successfully created so that if a failure happens from this point
1027     * on, we know to cleanup these datasets.
1028     */
1029    be_created = B_TRUE;
1030
1031    /*
1032     * Validate that the new BE is mountable.
1033     * Do not attempt to mount non-global zone datasets
1034     * since they are not cloned yet.
1035     */
1036    if ((ret = _be_mount(bt.nbe_name, &new_mp, BE_MOUNT_FLAG_NO_ZONES))
1037        != BE_SUCCESS) {
1038        be_print_err(gettext("be_copy: failed to "
1039            "mount newly created BE\n"));
1040        (void) _be_unmount(bt.nbe_name, 0);
1041        goto done;
1042    }
1043

```

```

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     }
1066
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     }
1082
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     }
1095
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     }
1105
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     }
1112
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     }
1123
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         }
1139
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         }
1147
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         }
1155
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     }
1167
1168 done:
1169     ZFS_CLOSE(zhp);
1170     be_free_fs_list(&fld);
1171
1172     if (bt.nbe_zfs_props != NULL)
1173         nvlist_free(bt.nbe_zfs_props);
1174
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 }
_____unchanged_portion_omitted_
```

```

*****
33179 Tue Aug 6 21:14:52 2013
new/usr/src/lib/libbe/common/be_list.c
*** NO COMMENTS ***
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /*
27  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
28  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
29 */

30 #include <assert.h>
31 #include <libintl.h>
32 #include <libnvpair.h>
33 #include <libzfs.h>
34 #include <stdio.h>
35 #include <stdlib.h>
36 #include <string.h>
37 #include <strings.h>
38 #include <sys/types.h>
39 #include <sys/stat.h>
40 #include <unistd.h>
41 #include <errno.h>

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

46 /*
47  * Callback data used for zfs_iter calls.
48  */
49 typedef struct list_callback_data {
50     char *zpool_name;
51     char *be_name;
52     be_node_list_t *be_nodes_head;
53     be_node_list_t *be_nodes;
54     char current_be[MAXPATHLEN];
55 } list_callback_data_t;
unchanged_portion_omitted

133 /* ***** */
134 /* Semi-Private Functions */
135 /* ***** */

```

```

137 /*
138  * Function: _be_list
139  * Description: This does the actual work described in be_list.
140  * Parameters:
141  *     be_name - The name of the BE to look up.
142  *     If NULL a list of all BEs will be returned.
143  *     be_nodes - A reference pointer to the list of BEs. The list
144  *     structure will be allocated here and must
145  *     be freed by a call to be_free_list. If there are no
146  *     BEs found on the system this reference will be
147  *     set to NULL.
148  * Return:
149  *     BE_SUCCESS - Success
150  *     be_errno_t - Failure
151  * Scope:
152  *     Semi-private (library wide use only)
153  */
154 int
155 _be_list(char *be_name, be_node_list_t **be_nodes)
156 {
157     list_callback_data_t cb = { 0 };
158     be_transaction_data_t bt = { 0 };
159     int ret = BE_SUCCESS;
160     zpool_handle_t *zphp;
161     char *rpool = NULL;
162     struct be_defaults be_defaults;

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

167     be_get_defaults(&be_defaults);

169     if (be_find_current_be(&bt) != BE_SUCCESS) {
170         /*
171          * We were unable to find a currently booted BE which
172          * probably means that we're not booted in a BE environment.
173          * None of the BE's will be marked as the active BE.
174          */
175         (void) strcpy(cb.current_be, "-");
176     } else {
177         (void) strncpy(cb.current_be, bt.obe_name,
178             sizeof (cb.current_be));
179         rpool = bt.obe_zpool;
180     }

182     /*
183      * If be_name is NULL we'll look for all BE's on the system.
184      * If not then we will only return data for the specified BE.
185      */
186     if (be_name != NULL)
187         cb.be_name = strdup(be_name);

189     if (be_defaults.be_deflt_rpool_container && rpool != NULL) {
190         if ((zphp = zpool_open(g_zfs, rpool)) == NULL) {
191             be_print_err(gettext("be_list: failed to "
192                 "be_get_node_data: failed to "
193                 "open rpool (%s): %s\n"), rpool,
194                 libzfs_error_description(g_zfs));
195             free(cb.be_name);
196             return (zfs_err_to_be_err(g_zfs));
197         }

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

```



```

201         if (cb.be_nodes_head != NULL) {
202             be_free_list(cb.be_nodes_head);
203             cb.be_nodes_head = NULL;
204             cb.be_nodes = NULL;
205         }
206         ret = BE_ERR_BE_NOENT;
207     }
208 }

210 if (cb.be_nodes_head == NULL) {
211     if (be_name != NULL)
212         be_print_err(gettext("be_list: BE (%s) does not "
213             "exist\n"), be_name);
214     else
215         be_print_err(gettext("be_list: No BE's found\n"));
216     ret = BE_ERR_BE_NOENT;
217 }

219 *be_nodes = cb.be_nodes_head;

221 free(cb.be_name);

223 be_sort_list(be_nodes);

225 return (ret);
226 }

```

unchanged_portion_omitted

```

796 /*
797 * Function:   be_get_node_data
798 * Description: Helper function used to collect all the information to fill
799 *              in the be_node_list structure to be returned by be_list.
800 * Parameters:
801 *             zhp - Handle to the root dataset for the BE whose information
802 *                 we're collecting.
803 *             be_node - a pointer to the node structure we're filling in.
804 *             be_name - The BE name of the node whose information we're
805 *                 collecting.
806 *             current_be - the name of the currently active BE.
807 *             be_ds - The dataset name for the BE.
808 *
809 * Returns:
810 *          BE_SUCCESS - Success
811 *          be_errno_t - Failure
812 *
813 * Scope:
814 *          Private
815 */
816 static int
817 be_get_node_data(
818     zfs_handle_t *zhp,
819     be_node_list_t *be_node,
820     char *be_name,
821     const char *rpool,
822     char *current_be,
823     char *be_ds)
824 {
825     char prop_buf[MAXPATHLEN];
826     nvlist_t *userprops = NULL;
827     nvlist_t *propval = NULL;
828     char *prop_str = NULL;
829     char *zone_prop_str = NULL;
830     char *grub_default_bootfs = NULL;
831     zpool_handle_t *zphp = NULL;
832     int err = 0;

```

```

834     if (be_node == NULL || be_name == NULL || current_be == NULL ||
835         be_ds == NULL) {
836         be_print_err(gettext("be_get_node_data: invalid arguments, "
837             "can not be NULL\n"));
838         return (BE_ERR_INVALID);
839     }

841     errno = 0;

843     be_node->be_root_ds = strdup(be_ds);
844     if ((err = errno) != 0 || be_node->be_root_ds == NULL) {
845         be_print_err(gettext("be_get_node_data: failed to "
846             "copy root dataset name\n"));
847         return (errno_to_be_err(err));
848     }

850     be_node->be_node_name = strdup(be_name);
851     if ((err = errno) != 0 || be_node->be_node_name == NULL) {
852         be_print_err(gettext("be_get_node_data: failed to "
853             "copy BE name\n"));
854         return (errno_to_be_err(err));
855     }
856     if (strncmp(be_name, current_be, MAXPATHLEN) == 0)
857         be_node->be_active = B_TRUE;
858     else
859         be_node->be_active = B_FALSE;

861     be_node->be_rpool = strdup(rpool);
862     if (be_node->be_rpool == NULL || (err = errno) != 0) {
863         be_print_err(gettext("be_get_node_data: failed to "
864             "copy root pool name\n"));
865         return (errno_to_be_err(err));
866     }

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

870     if (getzoneid() == GLOBAL_ZONEID) {
871         if ((zphp = zpool_open(g_zfs, rpool)) == NULL) {
872             be_print_err(gettext("be_get_node_data: failed to open "
873                 "pool (%s): %s\n"), rpool,
874                 libzfs_error_description(g_zfs));
875             be_print_err(gettext("be_get_node_data: failed to open pool "
876                 "(%s): %s\n"), rpool, libzfs_error_description(g_zfs));
877             return (zfs_err_to_be_err(g_zfs));
878         }

879         (void) zpool_get_prop(zphp, ZPOOL_PROP_BOOTFS, prop_buf,
880             ZFS_MAXPROPLEN, NULL);
881         if (be_has_grub() && (be_default_grub_bootfs(rpool,
882             &grub_default_bootfs) == BE_SUCCESS) &&
883             grub_default_bootfs != NULL)
884             (void) zpool_get_prop(zphp, ZPOOL_PROP_BOOTFS, prop_buf, ZFS_MAXPROPLEN,
885                 NULL);
886         if (be_has_grub() &&
887             (be_default_grub_bootfs(rpool, &grub_default_bootfs)
888             == BE_SUCCESS) && grub_default_bootfs != NULL)
889             if (strcmp(grub_default_bootfs, be_ds) == 0)
890                 be_node->be_active_on_boot = B_TRUE;
891             else
892                 be_node->be_active_on_boot = B_FALSE;
893         else if (prop_buf != NULL && strcmp(prop_buf, be_ds) == 0)
894             be_node->be_active_on_boot = B_TRUE;
895         else
896             be_node->be_active_on_boot = B_FALSE;

898     be_node->be_global_active = B_TRUE;

```

```

894         free(grub_default_bootfs);
895         zpool_close(zphp);
896     } else {
897         if (be_zone_compare_uuids(be_node->be_root_ds))
898             be_node->be_global_active = B_TRUE;
899         else
900             be_node->be_global_active = B_FALSE;
901     }

903     /*
904     * If the dataset is mounted use the mount point
905     * returned from the zfs_is_mounted call. If the
906     * dataset is not mounted then pull the mount
907     * point information out of the zfs properties.
908     */
909     be_node->be_mounted = zfs_is_mounted(zhp,
910         &(be_node->be_mntpt));
911     if (!be_node->be_mounted) {
912         if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, prop_buf,
913             ZFS_MAXPROPLEN, NULL, NULL, 0, B_FALSE) == 0)
914             be_node->be_mntpt = strdup(prop_buf);
915         else
916             return (zfs_err_to_be_err(g_zfs));
917     }

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

922     /* Get all user properties used for libbe */
923     if ((userprops = zfs_get_user_props(zhp)) == NULL) {
924         be_node->be_policy_type = strdup(be_default_policy());
925     } else {
926         if (getzoneid() != GLOBAL_ZONEID) {
927             if (nvlist_lookup_nvlist(userprops,
928                 BE_ZONE_ACTIVE_PROPERTY, &zone_propval) != 0 ||
929                 zone_propval == NULL) {
930                 be_node->be_active_on_boot = B_FALSE;
931             } else {
932                 verify(nvlist_lookup_string(zone_propval,
933                     ZPROP_VALUE, &zone_prop_str) == 0);
934                 if (strcmp(zone_prop_str, "on") == 0) {
935                     be_node->be_active_on_boot = B_TRUE;
936                 } else {
937                     be_node->be_active_on_boot = B_FALSE;
938                 }
939             }
940         }

942         if (nvlist_lookup_nvlist(userprops, BE_POLICY_PROPERTY,
943             &propval) != 0 || propval == NULL) {
944             be_node->be_policy_type =
945                 strdup(be_default_policy());
946         } else {
947             verify(nvlist_lookup_string(propval, ZPROP_VALUE,
948                 &prop_str) == 0);
949             if (prop_str == NULL || strcmp(prop_str, "-") == 0 ||
950                 strcmp(prop_str, "") == 0)
951                 be_node->be_policy_type =
952                     strdup(be_default_policy());
953             else
954                 be_node->be_policy_type = strdup(prop_str);
955         }
956         if (getzoneid() != GLOBAL_ZONEID) {
957             if (nvlist_lookup_nvlist(userprops,
958                 BE_ZONE_PARENTBE_PROPERTY, &propval) != 0 &&

```

```

959         nvlist_lookup_string(propval, ZPROP_VALUE,
960             &prop_str) == 0 && nvlist_lookup_string(propval, ZPROP_VALUE,
961                 &prop_str) == 0) {
962             be_node->be_uuid_str = strdup(prop_str);
963         } else {
964             if (nvlist_lookup_nvlist(userprops, BE_UUID_PROPERTY,
965                 &propval) == 0 && nvlist_lookup_string(propval,
966                     ZPROP_VALUE, &prop_str) == 0) {
967                 be_node->be_uuid_str = strdup(prop_str);
968             }
969         }
970     }

972     /*
973     * Increment the dataset counter to include the root dataset
974     * of the BE.
975     */
976     be_node->be_node_num_datasets++;

978     return (BE_SUCCESS);
979 }

```

unchanged_portion_omitted

```

*****
77719 Tue Aug 6 21:14:52 2013
new/usr/src/lib/libbe/common/be_mount.c
*** NO COMMENTS ***
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
24 */
25 /*
26  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
27  * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
28 */

29 /*
30  * System includes
31 */
32 #include <assert.h>
33 #include <errno.h>
34 #include <libgen.h>
35 #include <libintl.h>
36 #include <libnvpair.h>
37 #include <libzfs.h>
38 #include <stdio.h>
39 #include <stdlib.h>
40 #include <string.h>
41 #include <sys/mntent.h>
42 #include <sys/mnttab.h>
43 #include <sys/mount.h>
44 #include <sys/stat.h>
45 #include <sys/types.h>
46 #include <sys/vfstab.h>
47 #include <sys/zone.h>
48 #include <sys/mkdev.h>
49 #include <unistd.h>

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

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

56 typedef struct dir_data {
57     char *dir;
58     char *ds;
59 } dir_data_t;
    unchanged_portion_omitted

```

```

227 /* ***** */
228 /*                               Semi-Private Functions                               */
229 /* ***** */

231 /*
232  * Function:    _be_mount
233  * Description: Mounts a BE.  If the altroot is not provided, this function
234  * will generate a temporary mountpoint to mount the BE at.  It
235  * will return this temporary mountpoint to the caller via the
236  * altroot reference pointer passed in.  This returned value is
237  * allocated on heap storage and is the responsibility of the
238  * caller to free.
239  * Parameters:
240  *     be_name - pointer to name of BE to mount.
241  *     altroot - reference pointer to altroot of where to mount BE.
242  *     flags - flag indicating special handling for mounting the BE
243  * Return:
244  *     BE_SUCCESS - Success
245  *     be_errno_t - Failure
246  * Scope:
247  *     Semi-private (library wide use only)
248 */
249 int
250 _be_mount(char *be_name, char **altroot, int flags)
251 {
252     be_transaction_data_t  bt = { 0 };
253     be_mount_data_t md = { 0 };
254     zfs_handle_t          *zhp;
255     char                   obe_root_ds[MAXPATHLEN];
256     char                   *mp = NULL;
257     char                   *tmp_altroot = NULL;
258     int                    ret = BE_SUCCESS, err = 0;
259     uuid_t                 uu = { 0 };
260     boolean_t              gen_tmp_altroot = B_FALSE;

262     if (be_name == NULL || altroot == NULL)
263         return (BE_ERR_INVALID);

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

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

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

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

```

```

293     /* Make sure BE's root dataset isn't already mounted somewhere */
294     if (zfs_is_mounted(zhp, &mp)) {
295         ZFS_CLOSE(zhp);
296         be_print_err(gettext("be_mount: %s is already mounted "
297             "at %s\n"), bt.obe_name, mp != NULL ? mp : "");
298         free(mp);
299         return (BE_ERR_MOUNTED);
300     }
301
302     /*
303     * Fix this BE's mountpoint if its root dataset isn't set to
304     * either 'legacy' or '/'.
305     */
306     if ((ret = fix_mountpoint(zhp)) != BE_SUCCESS) {
307         be_print_err(gettext("be_mount: mountpoint check "
308             "failed for %s\n"), bt.obe_root_ds);
309         ZFS_CLOSE(zhp);
310         return (ret);
311     }
312
313     /*
314     * If altroot not provided, create a temporary alternate root
315     * to mount on
316     */
317     if (*altroot == NULL) {
318         if ((ret = be_make_tmp_mountpoint(&tmp_altroot))
319             != BE_SUCCESS) {
320             be_print_err(gettext("be_mount: failed to "
321                 "make temporary mountpoint\n"));
322             ZFS_CLOSE(zhp);
323             return (ret);
324         }
325         gen_tmp_altroot = B_TRUE;
326     } else {
327         tmp_altroot = *altroot;
328     }
329
330     md.altroot = tmp_altroot;
331     md.shared_fs = flags & BE_MOUNT_FLAG_SHARED_FS;
332     md.shared_rw = flags & BE_MOUNT_FLAG_SHARED_RW;
333
334     /* Mount the BE's root file system */
335     if (getzoneid() == GLOBAL_ZONEID) {
336         if ((ret = be_mount_root(zhp, tmp_altroot)) != BE_SUCCESS) {
337             be_print_err(gettext("be_mount: failed to "
338                 "mount BE root file system\n"));
339             if (gen_tmp_altroot)
340                 free(tmp_altroot);
341             ZFS_CLOSE(zhp);
342             return (ret);
343         }
344     } else {
345         /* Legacy mount the zone root dataset */
346         if ((ret = be_mount_zone_root(zhp, &md)) != BE_SUCCESS) {
347             be_print_err(gettext("be_mount: failed to "
348                 "mount BE zone root file system\n"));
349             free(md.altroot);
350             ZFS_CLOSE(zhp);
351             return (ret);
352         }
353     }
354
355     /* Iterate through BE's children filesystems */
356     if ((err = zfs_iter_filesystems(zhp, be_mount_callback,
357         tmp_altroot)) != 0) {

```

```

358         be_print_err(gettext("be_mount: failed to "
359             "mount BE (%s) on %s\n"), bt.obe_name, tmp_altroot);
360         if (gen_tmp_altroot)
361             free(tmp_altroot);
362         ZFS_CLOSE(zhp);
363         return (err);
364     }
365
366     md.altroot = tmp_altroot;
367     md.shared_fs = flags & BE_MOUNT_FLAG_SHARED_FS;
368     md.shared_rw = flags & BE_MOUNT_FLAG_SHARED_RW;
369
370     /*
371     * Mount shared file systems if mount flag says so.
372     */
373     if (md.shared_fs) {
374         /*
375         * Mount all ZFS file systems not under the BE's root dataset
376         */
377         (void) zpool_iter(g_zfs, zpool_shared_fs_callback, &md);
378
379         /* TODO: Mount all non-ZFS file systems - Not supported yet */
380     }
381
382     /*
383     * If we're in the global zone and the global zone has a valid uuid,
384     * mount all supported non-global zones.
385     */
386     if (getzoneid() == GLOBAL_ZONEID &&
387         !(flags & BE_MOUNT_FLAG_NO_ZONES) &&
388         be_get_uuid(bt.obe_root_ds, &uu) == BE_SUCCESS) {
389         if ((ret = be_mount_zones(zhp, &md)) != BE_SUCCESS) {
390             (void) _be_unmount(bt.obe_name, 0);
391             if (gen_tmp_altroot)
392                 free(tmp_altroot);
393             ZFS_CLOSE(zhp);
394             return (ret);
395         }
396     }
397
398     ZFS_CLOSE(zhp);
399
400     /*
401     * If a NULL altroot was passed in, pass the generated altroot
402     * back to the caller in altroot.
403     */
404     if (gen_tmp_altroot)
405         *altroot = tmp_altroot;
406
407     return (BE_SUCCESS);
408 }
409
410 /*
411 * Function:     _be_unmount
412 * Description:  Unmount a BE.
413 * Parameters:
414 *     be_name - pointer to name of BE to unmount.
415 *     flags - flags for unmounting the BE.
416 * Returns:
417 *     BE_SUCCESS - Success
418 *     be_errno_t - Failure
419 * Scope:
420 *     Semi-private (library wide use only)
421 */
422 int
423 _be_unmount(char *be_name, int flags)

```

```

420 {
421     be_transaction_data_t   bt = { 0 };
422     be_unmount_data_t       ud = { 0 };
423     zfs_handle_t           *zhp;
424     uuid_t                  uu = { 0 };
425     char                    obe_root_ds[MAXPATHLEN];
426     char                    mountpoint[MAXPATHLEN];
427     char                    *mp = NULL;
428     int                     ret = BE_SUCCESS;
429     int                     zret = 0;
431
432     if (be_name == NULL)
433         return (BE_ERR_INVAL);
434
435     /* Set be_name as obe_name in bt structure */
436     bt.obe_name = be_name;
437
438     /* Find which zpool obe_name lives in */
439     if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) == 0) {
440         be_print_err(gettext("be_unmount: failed to "
441             "find zpool for BE (%s)\n"), bt.obe_name);
442         return (BE_ERR_BE_NOENT);
443     } else if (zret < 0) {
444         be_print_err(gettext("be_unmount: "
445             "zpool_iter failed: %s\n"),
446             libzfs_error_description(g_zfs));
447         ret = zfs_err_to_be_err(g_zfs);
448         return (ret);
449     }
450
451     /* Generate string for obe_name's root dataset */
452     be_make_root_ds(bt.obe_zpool, bt.obe_name, obe_root_ds,
453         sizeof (obe_root_ds));
454     bt.obe_root_ds = obe_root_ds;
455
456     /* Get handle to BE's root dataset */
457     if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_FILESYSTEM)) ==
458         NULL) {
459         be_print_err(gettext("be_unmount: failed to "
460             "open BE root dataset (%s): %s\n"), bt.obe_root_ds,
461             libzfs_error_description(g_zfs));
462         ret = zfs_err_to_be_err(g_zfs);
463         return (ret);
464     }
465
466     /* Make sure BE's root dataset is mounted somewhere */
467     if (!zfs_is_mounted(zhp, &mp)) {
468         be_print_err(gettext("be_unmount: "
469             "(%s) not mounted\n"), bt.obe_name);
470
471         /*
472          * BE is not mounted, fix this BE's mountpoint if its root
473          * dataset isn't set to either 'legacy' or '/'.
474          */
475         if ((ret = fix_mountpoint(zhp)) != BE_SUCCESS) {
476             be_print_err(gettext("be_unmount: mountpoint check "
477                 "failed for %s\n"), bt.obe_root_ds);
478             ZFS_CLOSE(zhp);
479             return (ret);
480         }
481
482         ZFS_CLOSE(zhp);
483         return (BE_ERR_NOTMOUNTED);
484     }

```

```

486     /*
487      * If we didn't get a mountpoint from the zfs_is_mounted call,
488      * try and get it from its property.
489      */
490     if (mp == NULL) {
491         if (zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
492             sizeof (mountpoint), NULL, NULL, 0, B_FALSE) != 0) {
493             be_print_err(gettext("be_unmount: failed to "
494                 "get mountpoint of (%s)\n"), bt.obe_name);
495             ZFS_CLOSE(zhp);
496             return (BE_ERR_ZFS);
497         }
498     } else {
499         (void) strncpy(mountpoint, mp, sizeof (mountpoint));
500         free(mp);
501     }
502
503     /* If BE mounted as current root, fail */
504     if (strcmp(mountpoint, "/") == 0) {
505         be_print_err(gettext("be_unmount: "
506             "cannot unmount currently running BE\n"));
507         ZFS_CLOSE(zhp);
508         return (BE_ERR_UMOUNT_CURR_BE);
509     }
510
511     ud.altroot = mountpoint;
512     ud.force = flags & BE_UNMOUNT_FLAG_FORCE;
513
514     /* Unmount all supported non-global zones if we're in the global zone */
515     if (getzoneid() == GLOBAL_ZONEID &&
516         be_get_uuid(bt.obe_root_ds, &uu) == BE_SUCCESS) {
517         if ((ret = be_unmount_zones(&ud)) != BE_SUCCESS) {
518             ZFS_CLOSE(zhp);
519             return (ret);
520         }
521     }
522
523     /* TODO: Unmount all non-ZFS file systems - Not supported yet */
524
525     /* Unmount all ZFS file systems not under the BE root dataset */
526     if ((ret = unmount_shared_fs(&ud)) != BE_SUCCESS) {
527         be_print_err(gettext("be_unmount: failed to "
528             "unmount shared file systems\n"));
529         ZFS_CLOSE(zhp);
530         return (ret);
531     }
532
533     /* Unmount all children datasets under the BE's root dataset */
534     if ((zret = zfs_iter_filesystems(zhp, be_unmount_callback,
535         &ud)) != 0) {
536         be_print_err(gettext("be_unmount: failed to "
537             "unmount BE (%s)\n"), bt.obe_name);
538         ZFS_CLOSE(zhp);
539         return (zret);
540     }
541
542     /* Unmount this BE's root filesystem */
543     if (getzoneid() == GLOBAL_ZONEID) {
544         if ((ret = be_unmount_root(zhp, &ud)) != BE_SUCCESS) {
545             ZFS_CLOSE(zhp);
546             return (ret);
547         }
548     } else {
549         if ((ret = be_unmount_zone_root(zhp, &ud)) != BE_SUCCESS) {
550             ZFS_CLOSE(zhp);
551             return (ret);

```

```

552     }
553 }

555     ZFS_CLOSE(zhp);

557     return (BE_SUCCESS);
558 }

560 /*
561  * Function:    be_mount_zone_root
562  * Description: Mounts the zone root dataset for a zone.
563  * Parameters:
564  *             zfs - zfs_handle_t pointer to zone root dataset
565  *             md - be_mount_data_t pointer to data for zone to be mounted
566  * Returns:
567  *            BE_SUCCESS - Success
568  *            be_errno_t - Failure
569  * Scope:
570  *            Semi-private (library wide use only)
571  */
572 int
573 be_mount_zone_root(zfs_handle_t *zhp, be_mount_data_t *md)
574 {
575     struct stat buf;
576     char    mountpoint[MAXPATHLEN];
577     int     err = 0;

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

588     /*
589     * Make sure zone's root dataset is set to 'legacy'. This is
590     * currently a requirement in this implementation of zones
591     * support.
592     */
593     if (strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) != 0) {
594         be_print_err(gettext("be_mount_zone_root: "
595             "zone root dataset mountpoint is not 'legacy'\n"));
596         return (BE_ERR_ZONE_ROOT_NOT_LEGACY);
597     }

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

609     /*
610     * Legacy mount the zone root dataset.
611     *
612     * As a workaround for 6176743, we mount the zone's root with the
613     * MS_OVERLAY option in case an alternate BE is mounted, and we're
614     * mounting the root for the zone from the current BE here. When an
615     * alternate BE is mounted, it ties up the zone's zoneroot directory
616     * for the current BE since the zone's zonename is loopback mounted
617     * from the current BE.

```

```

618     *
619     * TODO: The MS_OVERLAY option needs to be removed when 6176743
620     * is fixed.
621     */
622     if (mount(zfs_get_name(zhp), md->altroot, MS_OVERLAY, MNTTYPE_ZFS,
623         NULL, 0, NULL, 0) != 0) {
624         err = errno;
625         be_print_err(gettext("be_mount_zone_root: failed to "
626             "legacy mount zone root dataset (%s) at %s\n"),
627             zfs_get_name(zhp), md->altroot);
628         return (errno_to_be_err(err));
629     }

631     return (BE_SUCCESS);
632 }

```

_____unchanged_portion_omitted_____

```

*****
23535 Tue Aug 6 21:14:53 2013
new/usr/src/lib/libbe/common/be_snapshot.c
*** NO COMMENTS ***
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /*
27  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
28  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
29 */

30 /*
31  * System includes
32 */
33 #include <assert.h>
34 #include <libintl.h>
35 #include <libnvpair.h>
36 #include <libzfs.h>
37 #include <stdio.h>
38 #include <stdlib.h>
39 #include <string.h>
40 #include <sys/types.h>
41 #include <sys/stat.h>
42 #include <unistd.h>

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

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

52 /* ***** */
53 /* Public Functions */
54 /* ***** */

56 /*
57  * Function: be_create_snapshot
58  * Description: Creates a recursive snapshot of all the datasets within a BE.
59  * If the name of the BE to snapshot is not provided, it assumes
60  * we're snapshotting the currently running BE. If the snapshot

```

```

61 * name is not provided it creates an auto named snapshot, which
62 * will be returned to the caller upon success.
63 * Parameters:
64 * be_attrs - pointer to nvlist_t of attributes being passed in.
65 * The following attributes are used by this function:
66 *
67 * BE_ATTR_ORIG_BE_NAME *optional
68 * BE_ATTR_SNAP_NAME *optional
69 * BE_ATTR_POLICY *optional
70 *
71 * If the BE_ATTR_SNAP_NAME was not passed in, upon
72 * successful BE snapshot creation, the following
73 * attribute value will be returned to the caller by
74 * setting it in the be_attrs parameter passed in:
75 *
76 * BE_ATTR_SNAP_NAME
77 *
78 * Return:
79 * BE_SUCCESS - Success
80 * be_errno_t - Failure
81 * Scope:
82 * Public
83 */
84 int
85 be_create_snapshot(nvlist_t *be_attrs)
86 {
87     char *be_name = NULL;
88     char *snap_name = NULL;
89     char *policy = NULL;
90     boolean_t autoname = B_FALSE;
91     int ret = BE_SUCCESS;

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

97     /* Get original BE name if one was provided */
98     if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
99         BE_ATTR_ORIG_BE_NAME, DATA_TYPE_STRING, &be_name, NULL) != 0) {
100         be_print_err(gettext("be_create_snapshot: failed to "
101             "lookup BE_ATTR_ORIG_BE_NAME attribute\n"));
102         be_zfs_fini();
103         return (BE_ERR_INVALID);
104     }

106     /* Validate original BE name if one was provided */
107     if (be_name != NULL && !be_valid_be_name(be_name)) {
108         be_print_err(gettext("be_create_snapshot: "
109             "invalid BE name %s\n"), be_name);
110         be_zfs_fini();
111         return (BE_ERR_INVALID);
112     }

114     /* Get snapshot name to create if one was provided */
115     if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
116         BE_ATTR_SNAP_NAME, DATA_TYPE_STRING, &snap_name, NULL) != 0) {
117         be_print_err(gettext("be_create_snapshot: "
118             "failed to lookup BE_ATTR_SNAP_NAME attribute\n"));
119         be_zfs_fini();
120         return (BE_ERR_INVALID);
121     }

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

```

```

127         "failed to lookup BE_ATTR_POLICY attribute\n"));
128     be_zfs_fini();
129     return (BE_ERR_INVAL);
130 }
131
132 /*
133  * If no snap_name ws provided, we're going to create an
134  * auto named snapshot. Set flag so that we know to pass
135  * the auto named snapshot to the caller later.
136  */
137 if (snap_name == NULL)
138     autoname = B_TRUE;
139
140 if ((ret = _be_create_snapshot(be_name, &snap_name, policy))
141     == BE_SUCCESS) {
142     if (autoname == B_TRUE) {
143         /*
144          * Set auto named snapshot name in the
145          * nvlist passed in by the caller.
146          */
147         if (nvlist_add_string(be_attrs, BE_ATTR_SNAP_NAME,
148             snap_name) != 0) {
149             be_print_err(gettext("be_create_snapshot: "
150                 "failed to add auto snap name (%s) to "
151                 "be_attrs\n"), snap_name);
152             ret = BE_ERR_NOMEM;
153         }
154     }
155 }
156
157 be_zfs_fini();
158
159 return (ret);
160 }

```

unchanged portion omitted

```

223 /*
224  * Function:    be_rollback
225  * Description: Rolls back a BE and all of its children datasets to the
226  *              named snapshot. All of the BE's datasets must have the
227  *              named snapshot for this function to succeed. If the name
228  *              of the BE is not passed in, this function assumes we're
229  *              operating on the currently booted live BE.
230  *
231  * Note - This function does not check if the BE has any
232  *         younger snapshots than the one we're trying to rollback to.
233  *         If it does, then those younger snapshots and their dependent
234  *         clone file systems will get destroyed in the process of
235  *         rolling back.
236  *
237  * Parameters:
238  *     be_attrs - pointer to nvlist_t of attributes being passed in.
239  *              The following attributes are used by this function:
240  *
241  *              BE_ATTR_ORIG_BE_NAME      *optional
242  *              BE_ATTR_SNAP_NAME        *required
243  *
244  * Returns:
245  *     BE_SUCCESS - Success
246  *     be_errno_t - Failure
247  *
248  * Scope:
249  *     Public
250  */
251 int
252 be_rollback(nvlist_t *be_attrs)

```

```

253     be_transaction_data_t  bt = { 0 };
254     zfs_handle_t           *zhp = NULL;
255     zpool_handle_t         *zphp;
256     char                   obe_root_ds[MAXPATHLEN];
257     char                   *obe_name = NULL;
258     int                    zret = 0, ret = BE_SUCCESS;
259     struct be_defaults be_defaults;
260
261     /* Initialize libzfs handle */
262     if (!be_zfs_init())
263         return (BE_ERR_INIT);
264
265     if ((ret = be_find_current_be(&bt)) != BE_SUCCESS) {
266         return (ret);
267     }
268
269     /* Get original BE name if one was provided */
270     if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
271         BE_ATTR_ORIG_BE_NAME, DATA_TYPE_STRING, &obe_name, NULL) != 0) {
272         be_print_err(gettext("be_rollback: "
273             "failed to lookup BE_ATTR_ORIG_BE_NAME attribute\n"));
274         return (BE_ERR_INVAL);
275     }
276
277     be_get_defaults(&be_defaults);
278
279     /* If original BE name not provided, use current BE */
280     if (obe_name != NULL) {
281         bt.obe_name = obe_name;
282         /* Validate original BE name */
283         if (!be_valid_be_name(bt.obe_name)) {
284             be_print_err(gettext("be_rollback: "
285                 "invalid BE name %s\n"), bt.obe_name);
286             return (BE_ERR_INVAL);
287         }
288     }
289
290     /* Get snapshot name to rollback to */
291     if (nvlist_lookup_string(be_attrs, BE_ATTR_SNAP_NAME, &bt.obe_snap_name)
292         != 0) {
293         be_print_err(gettext("be_rollback: "
294             "failed to lookup BE_ATTR_SNAP_NAME attribute.\n"));
295         return (BE_ERR_INVAL);
296     }
297
298     if (be_defaults.be_deflt_rpool_container) {
299         if ((zphp = zpool_open(g_zfs, bt.obe_zpool)) == NULL) {
300             be_print_err(gettext("be_rollback: failed to "
301                 "open rpool (%s): %s\n"), bt.obe_zpool,
302                 libzfs_error_description(g_zfs));
303             return (zfs_err_to_be_err(g_zfs));
304         }
305         zret = be_find_zpool_callback(zphp, &bt);
306     } else {
307         /* Find which zpool obe_name lives in */
308         if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) ==
309             0) {
310             be_print_err(gettext("be_rollback: "
311                 "failed to find zpool for BE (%s)\n"), bt.obe_name);
312             return (BE_ERR_BE_NOENT);
313         } else if (zret < 0) {
314             be_print_err(gettext("be_rollback: "
315                 "zpool_iter failed: %s\n"),
316                 libzfs_error_description(g_zfs));
317             return (zfs_err_to_be_err(g_zfs));
318         }

```



```

319     }

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

326     if (getzoneid() != GLOBAL_ZONEID) {
327         if (!be_zone_compare_uuids(bt.obe_root_ds)) {
328             be_print_err(gettext("be_rollback: rolling back zone "
329                 "root dataset from non-active global BE is not "
330                 "supported\n"));
331             return (BE_ERR_NOTSUP);
332         }
333     }

335     /* Get handle to BE's root dataset */
336     if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_DATASET)) == NULL) {
337         be_print_err(gettext("be_rollback: "
338             "failed to open BE root dataset (%s): %s\n"),
339             bt.obe_root_ds, libzfs_error_description(g_zfs));
340         return (zfs_err_to_be_err(g_zfs));
341     }

343     /*
344      * Check that snapshot name exists for this BE and all of its
345      * children file systems. This call will end up closing the
346      * zfs handle passed in whether it succeeds or fails.
347      */
348     if ((ret = be_rollback_check_callback(zhp, bt.obe_snap_name)) != 0) {
349         zhp = NULL;
350         return (ret);
351     }

353     /* Get handle to BE's root dataset */
354     if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_DATASET)) == NULL) {
355         be_print_err(gettext("be_rollback: "
356             "failed to open BE root dataset (%s): %s\n"),
357             bt.obe_root_ds, libzfs_error_description(g_zfs));
358         return (zfs_err_to_be_err(g_zfs));
359     }

361     /*
362      * Iterate through a BE's datasets and roll them all back to
363      * the specified snapshot. This call will end up closing the
364      * zfs handle passed in whether it succeeds or fails.
365      */
366     if ((ret = be_rollback_callback(zhp, bt.obe_snap_name)) != 0) {
367         zhp = NULL;
368         be_print_err(gettext("be_rollback: "
369             "failed to rollback BE %s to %s\n"), bt.obe_name,
370             bt.obe_snap_name);
371         return (ret);
372     }
373     zhp = NULL;
374     be_zfs_fini();
375     return (BE_SUCCESS);
376 }

379 /* ***** */
380 /*           Semi-Private Functions           */
381 /* ***** */

383 /*
384  * Function:     _be_create_snapshot

```

```

385  * Description: see be_create_snapshot
386  * Parameters:
387  *           be_name - The name of the BE that we're taking a snapshot of.
388  *           snap_name - The name of the snapshot we're creating. If
389  *           snap_name is NULL an auto generated name will be used,
390  *           and upon success, will return that name via this
391  *           reference pointer. The caller is responsible for
392  *           freeing the returned name.
393  *           policy - The clean-up policy type. (library wide use only)
394  * Return:
395  *           BE_SUCCESS - Success
396  *           be_errno_t - Failure
397  * Scope:
398  *           Semi-private (library wide use only)
399  */
400 int
401 _be_create_snapshot(char *be_name, char **snap_name, char *policy)
402 {
403     be_transaction_data_t   bt = { 0 };
404     zfs_handle_t            *zhp = NULL;
405     nvlist_t                *ss_props = NULL;
406     char                    ss[MAXPATHLEN];
407     char                    root_ds[MAXPATHLEN];
408     int                     pool_version = 0;
409     int                     i = 0;
410     int                     zret = 0, ret = BE_SUCCESS;
411     boolean_t               autoname = B_FALSE;

413     /* Set parameters in bt structure */
414     bt.obe_name = be_name;
415     bt.obe_snap_name = *snap_name;
416     bt.policy = policy;

418     /* If original BE name not supplied, use current BE */
419     if (bt.obe_name == NULL) {
420         if ((ret = be_find_current_be(&bt)) != BE_SUCCESS) {
421             return (ret);
422         }
423     }

425     /* Find which zpool obe_name lives in */
426     if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) == 0) {
427         be_print_err(gettext("be_create_snapshot: failed to "
428             "find zpool for BE (%s)\n"), bt.obe_name);
429         return (BE_ERR_BE_NOENT);
430     } else if (zret < 0) {
431         be_print_err(gettext("be_create_snapshot: "
432             "zpool_iter failed: %s\n"),
433             libzfs_error_description(g_zfs));
434         return (zfs_err_to_be_err(g_zfs));
435     }

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

441     if (getzoneid() != GLOBAL_ZONEID) {
442         if (!be_zone_compare_uuids(bt.obe_root_ds)) {
443             be_print_err(gettext("be_create_snapshot: creating "
444                 "snapshot for the zone root dataset from "
445                 "non-active global BE is not "
446                 "supported\n"));
447             return (BE_ERR_NOTSUP);
448         }
449     }

```

```

451 /* If BE policy not specified, use the default policy */
452 if (bt.policy == NULL) {
453     bt.policy = be_default_policy();
454 } else {
455     /* Validate policy type */
456     if (!valid_be_policy(bt.policy)) {
457         be_print_err(gettext("be_create_snapshot: "
458             "invalid BE policy type (%s)\n"), bt.policy);
459         return (BE_ERR_INVAL);
460     }
461 }
462
463 /*
464 * If snapshot name not specified, set auto name flag and
465 * generate auto snapshot name.
466 */
467 if (bt.obe_snap_name == NULL) {
468     autoname = B_TRUE;
469     if ((bt.obe_snap_name = be_auto_snap_name())
470         == NULL) {
471         be_print_err(gettext("be_create_snapshot: "
472             "failed to create auto snapshot name\n"));
473         ret = BE_ERR_AUTONAME;
474         goto done;
475     }
476 }
477
478 /* Generate the name of the snapshot to take. */
479 (void) snprintf(ss, sizeof (ss), "%s%s", bt.obe_root_ds,
480     bt.obe_snap_name);
481
482 /* Get handle to BE's root dataset */
483 if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_DATASET))
484     == NULL) {
485     be_print_err(gettext("be_create_snapshot: "
486         "failed to open BE root dataset (%s): %s\n"),
487         bt.obe_root_ds, libzfs_error_description(g_zfs));
488     ret = zfs_err_to_be_err(g_zfs);
489     goto done;
490 }
491
492 /* Get the ZFS pool version of the pool where this dataset resides */
493 if (zfs_spa_version(zhp, &pool_version) != 0) {
494     be_print_err(gettext("be_create_snapshot: failed to "
495         "get ZFS pool version for %s: %s\n"), zfs_get_name(zhp),
496         libzfs_error_description(g_zfs));
497 }
498
499 /*
500 * If ZFS pool version supports snapshot user properties, store
501 * cleanup policy there. Otherwise don't set one - this snapshot
502 * will always inherit the cleanup policy from its parent.
503 */
504 if (getzoneid() == GLOBAL_ZONEID) {
505     if (pool_version >= SPA_VERSION_SNAP_PROPS) {
506         if (nvlist_alloc(&ss_props, NV_UNIQUE_NAME, 0) != 0) {
507             be_print_err(gettext("be_create_snapshot: "
508                 "internal error: out of memory\n"));
509             be_print_err(gettext("be_create_snapshot: internal "
510                 "error: out of memory\n"));
511             return (BE_ERR_NOMEM);
512         }
513         if (nvlist_add_string(ss_props, BE_POLICY_PROPERTY,
514             bt.policy) != 0) {
515             be_print_err(gettext("be_create_snapshot: "
516                 "internal error: out of memory\n"));

```

```

491     if (nvlist_add_string(ss_props, BE_POLICY_PROPERTY, bt.policy)
492         != 0) {
493         be_print_err(gettext("be_create_snapshot: internal "
494             "error: out of memory\n"));
495         nvlist_free(ss_props);
496         return (BE_ERR_NOMEM);
497     }
498     } else if (policy != NULL) {
499         /*
500          * If an explicit cleanup policy was requested
501          * by the caller and we don't support it, error out.
502          */
503         be_print_err(gettext("be_create_snapshot: cannot set "
504             "cleanup policy: ZFS pool version is %d\n"),
505             pool_version);
506         "cleanup policy: ZFS pool version is %d\n", pool_version);
507         return (BE_ERR_NOTSUP);
508     }
509 }
510
511 /* Create the snapshots recursively */
512 if (zfs_snapshot(g_zfs, ss, B_TRUE, ss_props) != 0) {
513     if (!autoname || libzfs_errno(g_zfs) != EZFS_EXISTS) {
514         be_print_err(gettext("be_create_snapshot: "
515             "recursive snapshot of %s failed: %s\n"),
516             ss, libzfs_error_description(g_zfs));
517     }
518     if (libzfs_errno(g_zfs) == EZFS_EXISTS)
519         ret = BE_ERR_SS_EXISTS;
520     else
521         ret = zfs_err_to_be_err(g_zfs);
522     goto done;
523 } else {
524     for (i = 1; i < BE_AUTO_NAME_MAX_TRY; i++) {
525         /* Sleep 1 before retrying */
526         (void) sleep(1);
527
528         /* Generate new auto snapshot name. */
529         free(bt.obe_snap_name);
530         if ((bt.obe_snap_name =
531             be_auto_snap_name()) == NULL) {
532             be_print_err(gettext(
533                 "be_create_snapshot: failed to "
534                 "create auto snapshot name\n"));
535             ret = BE_ERR_AUTONAME;
536             goto done;
537         }
538
539         /* Generate string of the snapshot to take. */
540         (void) snprintf(ss, sizeof (ss), "%s%s",
541             bt.obe_root_ds, bt.obe_snap_name);
542
543         /* Create the snapshots recursively */
544         if (zfs_snapshot(g_zfs, ss, B_TRUE, ss_props)
545             != 0) {
546             if (libzfs_errno(g_zfs) !=
547                 EZFS_EXISTS) {
548                 be_print_err(gettext(
549                     "be_create_snapshot: "
550                     "recursive snapshot of %s "
551                     "failed: %s\n"), ss,
552                     libzfs_error_description(
553                         g_zfs));
554                 ret = zfs_err_to_be_err(g_zfs);

```

```
576                                     goto done;
577                                     } else {
578                                     } else {
579                                     } else {
580                                     } else {
581                                     } else {
582                                     } else {
583                                     } else {
584                                     } else {
585                                     } else {
586                                     } else {
587                                     } else {
588                                     } else {
589                                     } else {
590                                     } else {
591                                     } else {
592                                     } else {
593                                     } else {
594                                     } else {
595                                     } else {
596                                     } else {
597                                     } else {
598                                     } else {
599                                     } else {
600                                     } else {
601                                     } else {
602                                     } else {
603                                     } else {
604                                     } else {
605                                     } else {
606 done:
607     ZFS_CLOSE(zhp);
608
609     if (ss_props != NULL)
610         nvlist_free(ss_props);
611
612     return (ret);
613 }
_____unchanged_portion_omitted_____
```

```

*****
100368 Tue Aug 6 21:14:54 2013
new/usr/src/lib/libbe/common/be_utils.c
*** NO COMMENTS ***
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /*
27  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
28  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
29 */

31 /*
32  * System includes
33 */
34 #include <assert.h>
35 #include <errno.h>
36 #include <libgen.h>
37 #include <libintl.h>
38 #include <libnvpair.h>
39 #include <libzfs.h>
40 #include <libgen.h>
41 #include <stdio.h>
42 #include <stdlib.h>
43 #include <string.h>
44 #include <sys/stat.h>
45 #include <sys/types.h>
46 #include <sys/vfstab.h>
47 #include <sys/param.h>
48 #include <sys/systeminfo.h>
49 #include <ctype.h>
50 #include <time.h>
51 #include <unistd.h>
52 #include <fcntl.h>
53 #include <deflt.h>
54 #include <wait.h>
55 #include <libdevinfo.h>
56 #include <libgen.h>

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

```

```

61 /* Private function prototypes */
62 static int update_dataset(char *, int, char *, char *, char *);
63 static int _update_vfstab(char *, char *, char *, char *, be_fs_list_data_t *);
64 static int be_open_menu(char *, char *, FILE **, char *, boolean_t);
65 static int be_create_menu(char *, char *, FILE **, char *);
66 static char *be_get_auto_name(char *, char *, boolean_t);

68 /*
69  * Global error printing
70 */
71 boolean_t do_print = B_FALSE;

73 /*
74  * Private datatypes
75 */
76 typedef struct zone_be_name_cb_data {
77     char *base_be_name;
78     int num;
79 } zone_be_name_cb_data_t;
unchanged portion omitted

214 /*
215  * Function:    be_make_root_ds
216  * Description: Generate string for BE's root dataset given the pool
217  *              it lives in and the BE name.
218  * Parameters:
219  *              zpool - pointer zpool name.
220  *              be_name - pointer to BE name.
221  *              be_root_ds - pointer to buffer to return BE root dataset in.
222  *              be_root_ds_size - size of be_root_ds
223  * Returns:
224  *              None
225  * Scope:
226  *              Semi-private (library wide use only)
227 */
228 void
229 be_make_root_ds(const char *zpool, const char *be_name, char *be_root_ds,
230                int be_root_ds_size)
231 {
232     struct be_defaults be_defaults;
233     be_get_defaults(&be_defaults);
234     char *root_ds = NULL;

236     if (getzoneid() == GLOBAL_ZONEID) {
237         if (be_defaults.be_deflt_rpool_container) {
238             (void) snprintf(be_root_ds, be_root_ds_size,
239                            "%s/%s", zpool, be_name);
240         } else {
241             (void) snprintf(be_root_ds, be_root_ds_size,
242                            "%s/%s/%s", zpool, BE_CONTAINER_DS_NAME, be_name);
243         }
244     } else {
245         /*
246          * In non-global zone we can use path from mounted root dataset
247          * to generate BE's root dataset string.
248          */
249         if ((root_ds = be_get_ds_from_dir("/")) != NULL) {
250             (void) snprintf(be_root_ds, be_root_ds_size, "%s/%s",
251                            dirname(root_ds), be_name);
252         } else {
253             be_print_err(gettext("be_make_root_ds: zone root "
254                                "dataset is not mounted\n"));
255             return;
256         }
257     }
258     if (be_defaults.be_deflt_rpool_container)

```

```

235     (void) snprintf(be_root_ds, be_root_ds_size, "%s/%s", zpool,
236                    be_name);
237     else
238     (void) snprintf(be_root_ds, be_root_ds_size, "%s/%s/%s", zpool,
239                    BE_CONTAINER_DS_NAME, be_name);
258 }

260 /*
261 * Function:    be_make_container_ds
262 * Description: Generate string for the BE container dataset given a pool name.
263 * Parameters:  zpool - pointer zpool name.
264 *              container_ds - pointer to buffer to return BE container
265 *                  dataset in.
266 *              container_ds_size - size of container_ds
267 * Returns:     None
268 * Scope:      Semi-private (library wide use only)
272 */
273 void
274 be_make_container_ds(const char *zpool, char *container_ds,
275                    int container_ds_size)
276 {
277     struct be_defaults be_defaults;
278     be_get_defaults(&be_defaults);
279     char *root_ds = NULL;

281     if (getzoneid() == GLOBAL_ZONEID) {
282         if (be_defaults.be_deflt_rpool_container) {
283             (void) snprintf(container_ds, container_ds_size,
284                            "%s", zpool);
285         } else {
286             (void) snprintf(container_ds, container_ds_size,
287                            "%s/%s", zpool, BE_CONTAINER_DS_NAME);
288         }
289     } else {
290         if ((root_ds = be_get_ds_from_dir("/")) != NULL) {
291             (void) strncpy(container_ds, dirname(root_ds),
292                            container_ds_size);
293         } else {
294             be_print_err(gettext("be_make_container_ds: zone root "
295                                "dataset is not mounted\n"));
296             return;
297         }
298     }
299     if (be_defaults.be_deflt_rpool_container)
300         (void) snprintf(container_ds, container_ds_size, "%s", zpool);
301     else
302         (void) snprintf(container_ds, container_ds_size, "%s/%s", zpool,
303                        BE_CONTAINER_DS_NAME);

```

unchanged portion omitted

```

2447 /*
2448 * Function:    be_zpool_find_current_be_callback
2449 * Description: Callback function used to iterate through all existing pools
2450 *              to find the BE that is the currently booted BE.
2451 * Parameters:  zlp - zpool_handle_t pointer to the current pool being
2452 *                  looked at.
2453 *              data - be_transaction_data_t pointer.
2454 *              Upon successfully finding the current BE, the
2455 *                  obe_zpool member of this parameter is set to the
2456 *                  pool it is found in.
2457 * Return:

```

```

2459 *          1 - Found current BE in this pool.
2460 *          0 - Did not find current BE in this pool.
2461 * Scope:
2462 *          Semi-private (library wide use only)
2463 */
2464 int
2465 be_zpool_find_current_be_callback(zpool_handle_t *zlp, void *data)
2466 {
2467     be_transaction_data_t *bt = data;
2468     zfs_handle_t *zhp = NULL;
2469     const char *zpool = zpool_get_name(zlp);
2470     char be_container_ds[MAXPATHLEN];
2471     char *zpath = NULL;

2473     /*
2474     * Generate string for BE container dataset
2475     */
2476     if (getzoneid() != GLOBAL_ZONEID) {
2477         if ((zpath = be_get_ds_from_dir("/")) != NULL) {
2478             (void) strncpy(be_container_ds, dirname(zpath),
2479                            sizeof (be_container_ds));
2480         } else {
2481             be_print_err(gettext(
2482                "be_zpool_find_current_be_callback: "
2483                "zone root dataset is not mounted\n"));
2484             return (0);
2485         }
2486     } else {
2487         be_make_container_ds(zpool, be_container_ds,
2488                            sizeof (be_container_ds));
2489     }
2490     be_make_container_ds(zpool, be_container_ds, sizeof (be_container_ds));

2491     /*
2492     * Check if a BE container dataset exists in this pool.
2493     */
2494     if (!zfs_dataset_exists(g_zfs, be_container_ds, ZFS_TYPE_FILESYSTEM)) {
2495         zpool_close(zlp);
2496         return (0);
2497     }

2499     /*
2500     * Get handle to this zpool's BE container dataset.
2501     */
2502     if ((zhp = zfs_open(g_zfs, be_container_ds, ZFS_TYPE_FILESYSTEM)) ==
2503         NULL) {
2504         be_print_err(gettext("be_zpool_find_current_be_callback: "
2505                            "failed to open BE container dataset (%s)\n",
2506                            be_container_ds);
2507         zpool_close(zlp);
2508         return (0);
2509     }

2511     /*
2512     * Iterate through all potential BEs in this zpool
2513     */
2514     if (zfs_iter_filesystems(zhp, be_zfs_find_current_be_callback, bt)) {
2515         /*
2516         * Found current BE dataset; set obe_zpool
2517         */
2518         if ((bt->obe_zpool = strdup(zpool)) == NULL) {
2519             be_print_err(gettext(
2520                "be_zpool_find_current_be_callback: "
2521                "memory allocation failed\n"));
2522             ZFS_CLOSE(zhp);
2523             zpool_close(zlp);

```

new/usr/src/lib/libbe/common/be_utils.c

5

```
2524             return (0);
2525         }
2527         ZFS_CLOSE(zhp);
2528         zpool_close(zlp);
2529         return (1);
2530     }
2532     ZFS_CLOSE(zhp);
2533     zpool_close(zlp);
2535     return (0);
2536 }
unchanged_portion_omitted
```

```

*****
18420 Tue Aug 6 21:14:55 2013
new/usr/src/lib/libbe/common/be_zones.c
*** NO COMMENTS ***
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /*
27  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
28 */

30 /*
31  * System includes
32 */
33 #include <assert.h>
34 #include <errno.h>
35 #include <libintl.h>
36 #include <libnvpair.h>
37 #include <libzfs.h>
38 #include <stdio.h>
39 #include <stdlib.h>
40 #include <string.h>
41 #include <sys/mntent.h>
42 #include <sys/mnttab.h>
43 #include <sys/mount.h>
44 #include <sys/stat.h>
45 #include <sys/types.h>
46 #include <sys/vfstab.h>
47 #include <unistd.h>

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

52 typedef struct active_zone_root_data {
53     uid_t parent_uid;
54     char *zoneroot_ds;
55 } active_zone_root_data_t;
56 unchanged portion omitted

91 /*
92  * Function:    be_find_active_zone_root
93  * Description: This function will find the active zone root of a zone for
94  *              a given global BE. It will iterate all of the zone roots

```

```

95 *          under a zonepath, find the zone roots that belong to the
96 *          specified global BE, and return the one that is active.
97 * Parameters:
98 *          be_zhp - zfs handle to global BE root dataset.
99 *          zonpath_ds - pointer to zone's zonepath dataset.
100 *          zoneroot_ds - pointer to a buffer to store the dataset name of
101 *                      the zone's zoneroot that's currently active for this
102 *                      given global BE..
103 *          zoneroot_ds_size - size of zoneroot_ds.
104 * Returns:
105 *          BE_SUCCESS - Success
106 *          be_errno_t - Failure
107 * Scope:
108 *          Semi-private (library wide use only)
109 */
110 int
111 be_find_active_zone_root(zfs_handle_t *be_zhp, char *zonpath_ds,
112                         char *zoneroot_ds, int zoneroot_ds_size)
113 {
114     active_zone_root_data_t    azr_data = { 0 };
115     zfs_handle_t              *zhp;
116     char                      zone_container_ds[MAXPATHLEN];
117     int                        ret = BE_SUCCESS;

119     /* Get the uuid of the parent global BE */
120     if (getzoneid() == GLOBAL_ZONEID) {
121         if ((ret = be_get_uuid(zfs_get_name(be_zhp),
122                                &azr_data.parent_uuid)) != BE_SUCCESS) {
123             be_print_err(gettext("be_find_active_zone_root: failed "
124                                 "to get uuid for BE root dataset %s\n"),
125                          zfs_get_name(be_zhp));
126         }
127         if ((ret = be_get_uuid(zfs_get_name(be_zhp), &azr_data.parent_uuid))
128             != BE_SUCCESS) {
129             be_print_err(gettext("be_find_active_zone_root: failed to "
130                                 "get uuid for BE root dataset %s\n"),
131                          zfs_get_name(be_zhp));
132             return (ret);
133         }
134     } else {
135         if ((ret = be_zone_get_parent_uuid(zfs_get_name(be_zhp),
136                                             &azr_data.parent_uuid)) != BE_SUCCESS) {
137             be_print_err(gettext("be_find_active_zone_root: failed "
138                                 "to get parentbe uuid for zone root dataset %s\n"),
139                          zfs_get_name(be_zhp));
140             return (ret);
141         }
142     }

143     /* Generate string for the root container dataset for this zone. */
144     be_make_container_ds(zonpath_ds, zone_container_ds,
145                         sizeof (zone_container_ds));

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

```

```

157     &azr_data)) != 0) {
158         be_print_err(gettext("be_find_active_zone_root: failed to "
159             "find active zone root in zonepath dataset %s: %s\n"),
160             zonepath_ds, be_err_to_str(ret));
161         goto done;
162     }
164     if (azr_data.zoneroot_ds != NULL) {
165         (void) strcpy(zoneroot_ds, azr_data.zoneroot_ds,
166             zoneroot_ds_size);
167         free(azr_data.zoneroot_ds);
168     } else {
169         be_print_err(gettext("be_find_active_zone_root: failed to "
170             "find active zone root in zonepath dataset %s\n"),
171             zonepath_ds);
172         ret = BE_ERR_ZONE_NO_ACTIVE_ROOT;
173     }
175 done:
176     ZFS_CLOSE(zhp);
177     return (ret);
178 }

```

unchanged portion omitted

```

394 /*
395  * Function:    be_zone_set_parent_uid
396  * Description: This function sets parentbe uuid into
397  *              a zfs user property for a root zone dataset.
398  * Parameters:  root_ds - Root zone dataset of the BE to set a uuid on.
399  * Return:     root_ds - Root zone dataset of the BE to set a uuid on.
400  *             be_errno_t - Failure
401  *             BE_SUCCESS - Success
402  * Scope:      Semi-private (library wide uses only)
403  */
404 int
405 be_zone_set_parent_uid(char *root_ds, uuid_t uu)
406 {
407     zfs_handle_t *zhp = NULL;
408     char uu_string[UUID_PRINTABLE_STRING_LENGTH];
409     int ret = BE_SUCCESS;
410
411     uuid_unparse(uu, uu_string);
412
413     /* Get handle to the root zone dataset. */
414     if ((zhp = zfs_open(g_zfs, root_ds, ZFS_TYPE_FILESYSTEM)) == NULL) {
415         be_print_err(gettext("be_zone_set_parent_uid: failed to "
416             "open root zone dataset (%s): %s\n"), root_ds,
417             libzfs_error_description(g_zfs));
418         return (zfs_err_to_be_err(g_zfs));
419     }
420
421     /* Set parentbe uuid property for the root zone dataset */
422     if (zfs_prop_set(zhp, BE_ZONE_PARENTBE_PROPERTY, uu_string) != 0) {
423         be_print_err(gettext("be_zone_set_parent_uid: failed to "
424             "set parentbe uuid property for root zone dataset: %s\n"),
425             libzfs_error_description(g_zfs));
426         ret = zfs_err_to_be_err(g_zfs);
427     }
428
429     ZFS_CLOSE(zhp);
430     return (ret);
431 }
432
433 */

```

```

436 * Function:    be_zone_compare_uuids
437 * Description: This function compare the parentbe uuid of the
438 *              current running root zone dataset with the parentbe
439 *              uuid of the given root zone dataset.
440 * Parameters:  root_ds - Root zone dataset of the BE to compare.
441 * Return:     B_TRUE - root dataset has right parentbe uuid
442 *             B_FALSE - root dataset has wrong parentbe uuid
443 * Scope:      Semi-private (library wide uses only)
444 */
445 boolean_t
446 be_zone_compare_uuids(char *root_ds)
447 {
448     char *active_ds;
449     uuid_t parent_uuid = {0};
450     uuid_t cur_parent_uuid = {0};
451
452     /* Get parentbe uuid from given zone root dataset */
453     if ((be_zone_get_parent_uuid(root_ds,
454         &parent_uuid)) != BE_SUCCESS) {
455         be_print_err(gettext("be_zone_compare_uuids: failed to get "
456             "parentbe uuid from the given BE\n"));
457         return (B_FALSE);
458     }
459
460     /* Find current running zone root dataset and get it's parentbe
461     * uuid property.
462     */
463     if ((active_ds = be_get_ds_from_dir("/") != NULL) {
464         if ((be_zone_get_parent_uuid(active_ds,
465             &cur_parent_uuid)) != BE_SUCCESS) {
466             be_print_err(gettext("be_zone_compare_uuids: failed "
467                 "to get parentbe uuid from the current running zone "
468                 "root dataset\n"));
469             return (B_FALSE);
470         }
471     } else {
472         be_print_err(gettext("be_zone_compare_uuids: zone root dataset "
473             "is not mounted\n"));
474         return (B_FALSE);
475     }
476
477     if (uuid_compare(parent_uuid, cur_parent_uuid) != 0) {
478         return (B_FALSE);
479     }
480
481     return (B_TRUE);
482 }
483
484
485
486
487
488 /* ***** Private Functions ***** */
489
490
491
492 /*
493  * Function:    be_find_active_zone_root_callback
494  * Description: This function is used as a callback to iterate over all of
495  *              a zone's root datasets, finding the one that is marked active
496  *              for the parent BE specified in the data passed in. The name
497  *              of the zone's active root dataset is returned in heap storage
498  *              in the active_zone_root_data_t structure passed in, so the
499  *              caller is responsible for freeing it.
500  * Parameters:  zhp - zfs_handle_t pointer to current dataset being processed

```



```
502 *      data - active_zone_root_data_t pointer
503 * Returns:
504 *      0 - Success
505 *      >0 - Failure
506 * Scope:
507 *      Private
508 */
509 static int
510 be_find_active_zone_root_callback(zfs_handle_t *zhp, void *data)
511 {
512     active_zone_root_data_t *azr_data = data;
513     uuid_t parent_uuid = { 0 };
514     int iret = 0;
515     int ret = 0;
516
517     if ((iret = be_zone_get_parent_uuid(zfs_get_name(zhp), &parent_uuid))
518         != BE_SUCCESS) {
519         be_print_err(gettext("be_find_active_zone_root_callback: "
520             "skipping zone root dataset (%s): %s\n"),
521             zfs_get_name(zhp), be_err_to_str(iret));
522         goto done;
523     }
524
525     if (uuid_compare(azr_data->parent_uuid, parent_uuid) == 0) {
526         /*
527          * Found a zone root dataset belonging to the right parent,
528          * check if its active.
529          */
530         if (be_zone_get_active(zhp)) {
531             /*
532              * Found active zone root dataset, if its already
533              * set in the callback data, that means this
534              * is the second one we've found. Return error.
535              */
536             if (azr_data->zoneroot_ds != NULL) {
537                 ret = BE_ERR_ZONE_MULTIPLE_ACTIVE;
538                 goto done;
539             }
540
541             azr_data->zoneroot_ds = strdup(zfs_get_name(zhp));
542             if (azr_data->zoneroot_ds == NULL) {
543                 ret = BE_ERR_NOMEM;
544             }
545         }
546     }
547
548 done:
549     ZFS_CLOSE(zhp);
550     return (ret);
551 }
552
553 _____unchanged_portion_omitted_____
```

```

*****
      8194 Tue Aug  6 21:14:55 2013
new/usr/src/lib/libbe/common/libbe.h
*** NO COMMENTS ***
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /*
27  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
28 */

30 #ifndef _LIBBE_H
31 #define _LIBBE_H

33 #include <libnvpair.h>
34 #include <uuid/uuid.h>
35 #include <libzfs.h>

37 #ifdef __cplusplus
38 extern "C" {
39 #endif

41 #define BE_ATTR_ORIG_BE_NAME      "orig_be_name"
42 #define BE_ATTR_ORIG_BE_POOL     "orig_be_pool"
43 #define BE_ATTR_SNAP_NAME        "snap_name"

45 #define BE_ATTR_NEW_BE_NAME      "new_be_name"
46 #define BE_ATTR_NEW_BE_POOL     "new_be_pool"
47 #define BE_ATTR_NEW_BE_DESC     "new_be_desc"
48 #define BE_ATTR_POLICY          "policy"
49 #define BE_ATTR_ZFS_PROPERTIES  "zfs_properties"

51 #define BE_ATTR_FS_NAMES        "fs_names"
52 #define BE_ATTR_FS_NUM          "fs_num"
53 #define BE_ATTR_SHARED_FS_NAMES "shared_fs_names"
54 #define BE_ATTR_SHARED_FS_NUM   "shared_fs_num"

56 #define BE_ATTR_MOUNTPOINT      "mountpoint"
57 #define BE_ATTR_MOUNT_FLAGS     "mount_flags"
58 #define BE_ATTR_UNMOUNT_FLAGS  "unmount_flags"
59 #define BE_ATTR_DESTROY_FLAGS  "destroy_flags"
60 #define BE_ATTR_ROOT_DS        "root_ds"
61 #define BE_ATTR_UUID_STR       "uuid_str"

```

```

63 #define BE_ATTR_ACTIVE          "active"
64 #define BE_ATTR_ACTIVE_ON_BOOT "active_boot"
65 #define BE_ATTR_GLOBAL_ACTIVE  "global_active"
66 #define BE_ATTR_SPACE         "space_used"
67 #define BE_ATTR_DATASET       "dataset"
68 #define BE_ATTR_STATUS        "status"
69 #define BE_ATTR_DATE          "date"
70 #define BE_ATTR_MOUNTED       "mounted"

72 /*
73  * libbe error codes
74  *
75  * NOTE: there is a copy of this enum in beadm/messages.py. To keep these
76  * in sync please make sure to add any new error messages at the end
77  * of this enumeration.
78  */
79 enum {
80     BE_SUCCESS = 0,
81     BE_ERR_ACCESS = 4000, /* permission denied */
82     BE_ERR_ACTIVATE_CURR, /* Activation of current BE failed */
83     BE_ERR_AUTONAME, /* auto naming failed */
84     BE_ERR_BE_NOENT, /* No such BE */
85     BE_ERR_BUSY, /* mount busy */
86     BE_ERR_CANCELED, /* operation canceled */
87     BE_ERR_CLONE, /* BE clone failed */
88     BE_ERR_COPY, /* BE copy failed */
89     BE_ERR_CREATDS, /* dataset creation failed */
90     BE_ERR_CURR_BE_NOT_FOUND, /* Can't find current BE */
91     BE_ERR_DESTROY, /* failed to destroy BE or snapshot */
92     BE_ERR_DEMOTE, /* BE demotion failed */
93     BE_ERR_DSTYPE, /* invalid dataset type */
94     BE_ERR_BE_EXISTS, /* BE exists */
95     BE_ERR_INIT, /* be_zfs_init failed */
96     BE_ERR_INTR, /* interrupted system call */
97     BE_ERR_INVALID, /* invalid argument */
98     BE_ERR_INVALIDPROP, /* invalid property for dataset */
99     BE_ERR_INVALIDMOUNTPOINT, /* Unexpected mountpoint */
100    BE_ERR_MOUNT, /* mount failed */
101    BE_ERR_MOUNTED, /* already mounted */
102    BE_ERR_NAMETOOLONG, /* name > BUFSIZ */
103    BE_ERR_NOENT, /* Doesn't exist */
104    BE_ERR_POOL_NOENT, /* No such pool */
105    BE_ERR_NODEV, /* No such device */
106    BE_ERR_NOTMOUNTED, /* File system not mounted */
107    BE_ERR_NOMEM, /* not enough memory */
108    BE_ERR_NONINHERIT, /* property is not inheritable for BE dataset */
109    BE_ERR_NXIO, /* No such device or address */
110    BE_ERR_NOSPC, /* No space on device */
111    BE_ERR_NOTSUP, /* Operation not supported */
112    BE_ERR_OPEN, /* open failed */
113    BE_ERR_PERM, /* Not owner */
114    BE_ERR_UNAVAIL, /* The BE is currently unavailable */
115    BE_ERR_PROMOTE, /* BE promotion failed */
116    BE_ERR_ROFS, /* read only file system */
117    BE_ERR_READONLYDS, /* read only dataset */
118    BE_ERR_READONLYPROP, /* read only property */
119    BE_ERR_SS_EXISTS, /* snapshot exists */
120    BE_ERR_SS_NOENT, /* No such snapshot */
121    BE_ERR_UMOUNT, /* unmount failed */
122    BE_ERR_UMOUNT_CURR_BE, /* Can't unmount current BE */
123    BE_ERR_UMOUNT_SHARED, /* unmount of shared File System failed */
124    BE_ERR_UNKNOWN, /* Unknown error */
125    BE_ERR_ZFS, /* ZFS returned an error */
126    BE_ERR_DESTROY_CURR_BE, /* Cannot destroy current BE */
127    BE_ERR_GEN_UUID, /* Failed to generate uuid */

```

```
128 BE_ERR_PARSE_UUID, /* Failed to parse uuid */
129 BE_ERR_NO_UUID, /* BE has no uuid */
130 BE_ERR_ZONE_NO_PARENTBE, /* Zone root dataset has no parent uuid */
131 BE_ERR_ZONE_MULTIPLE_ACTIVE, /* Zone has multiple active roots */
132 BE_ERR_ZONE_NO_ACTIVE_ROOT, /* Zone has no active root for this BE */
133 BE_ERR_ZONE_ROOT_NOT_LEGACY, /* Zone root dataset mntpt is not legacy */
134 BE_ERR_NO_MOUNTED_ZONE, /* Zone not mounted in alternate BE */
135 BE_ERR_MOUNT_ZONEROOT, /* Failed to mount a zone root */
136 BE_ERR_UMOUNT_ZONEROOT, /* Failed to unmount a zone root */
137 BE_ERR_ZONES_UNMOUNT, /* Unable to unmount a zone. */
138 BE_ERR_FAULT, /* Bad Address */
139 BE_ERR_RENAME_ACTIVE, /* Renaming the active BE is not supported */
140 BE_ERR_NO_MENU, /* Missing boot menu file */
141 BE_ERR_DEV_BUSY, /* Device is Busy */
142 BE_ERR_BAD_MENU_PATH, /* Invalid path for menu.lst file */
143 BE_ERR_ZONE_SS_EXISTS, /* zone snapshot already exists */
144 BE_ERR_ADD_SPLASH_ICT, /* Add_splash_image ICT failed */
145 BE_ERR_BOOTFILE_INST, /* Error installing boot files */
146 BE_ERR_EXTCMD /* External command error */
147 } be_errno_t;
```

unchanged_portion_omitted

```
170 typedef struct be_node_list {
171     boolean_t be_mounted; /* is BE currently mounted */
172     boolean_t be_active_on_boot; /* is this BE active on boot */
173     boolean_t be_active; /* is this BE active currently */
174     boolean_t be_global_active; /* is zone's BE associated with */
175     /* an active global BE */
176     uint64_t be_space_used;
177     char *be_node_name;
178     char *be_rpool;
179     char *be_root_ds;
180     char *be_mntpt;
181     char *be_policy_type; /* cleanup policy type */
182     char *be_uuid_str; /* string representation of uuid */
183     time_t be_node_creation; /* Date/time stamp when created */
184     struct be_dataset_list *be_node_datasets;
185     uint_t be_node_num_datasets;
186     struct be_snapshot_list *be_node_snapshots;
187     uint_t be_node_num_snapshots;
188     struct be_node_list *be_next_node;
189 } be_node_list_t;
```

unchanged_portion_omitted

```

*****
7663 Tue Aug 6 21:14:56 2013
new/usr/src/lib/libbe/common/libbe_priv.h
*** NO COMMENTS ***
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /*
27  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
28  * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
29 */

30 #ifndef _LIBBE_PRIV_H
31 #define _LIBBE_PRIV_H

33 #include <libnvpair.h>
34 #include <libzfs.h>
35 #include <instzones_api.h>

37 #ifdef __cplusplus
38 extern "C" {
39 #endif

41 #define ARCH_LENGTH          MAXNAMELEN
42 #define BE_AUTO_NAME_MAX_TRY 3
43 #define BE_AUTO_NAME_DELIM  '-'
44 #define BE_DEFAULTS         "/etc/default/be"
45 #define BE_DFLT_BENAME_STARTS "BENAME_STARTS_WITH="
46 #define BE_CONTAINER_DS_NAME "ROOT"
47 #define BE_DEFAULT_CONSOLE  "text"
48 #define BE_POLICY_PROPERTY  "org.opensolaris.libbe:policy"
49 #define BE_UUID_PROPERTY    "org.opensolaris.libbe:uuid"
50 #define BE_PLCY_STATIC      "static"
51 #define BE_PLCY_VOLATILE    "volatile"
52 #define BE_GRUB_MENU        "/boot/grub/menu.lst"
53 #define BE_SPARC_MENU       "/boot/menu.lst"
54 #define BE_GRUB_COMMENT     "#===== End of LIBBE entry ====="
55 #define BE_GRUB_SPLASH      "splashimage /boot/solaris.xpm"
56 #define BE_GRUB_FOREGROUND  "foreground 343434"
57 #define BE_GRUB_BACKGROUND  "background F7FBFF"
58 #define BE_GRUB_DEFAULT     "default 0"
59 #define BE_WHITE_SPACE      "\t\r\n"
60 #define BE_CAP_FILE         "/boot/grub/capability"

```

```

61 #define BE_INSTALL_GRUB     "/sbin/installgrub"
62 #define BE_STAGE_1         "/boot/grub/stage1"
63 #define BE_STAGE_2         "/boot/grub/stage2"
64 #define ZFS_CLOSE(_zhp) \
65     if (_zhp) { \
66         zfs_close(_zhp); \
67         _zhp = NULL; \
68     }

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

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

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

81 typedef struct be_transaction_data {
82     char *obe_name; /* Original BE name */
83     char *obe_root_ds; /* Original BE root dataset */
84     char *obe_zpool; /* Original BE pool */
85     char *obe_snap_name; /* Original BE snapshot name */
86     char *obe_altroot; /* Original BE altroot */
87     char *nbe_name; /* New BE name */
88     char *nbe_root_ds; /* New BE root dataset */
89     char *nbe_zpool; /* New BE pool */
90     char *nbe_desc; /* New BE description */
91     nvlist_t *nbe_zfs_props; /* New BE dataset properties */
92     char *policy; /* BE policy type */
93 } be_transaction_data_t;
_____
unchanged portion omitted

139 /* Library globals */
140 extern libzfs_handle_t *g_zfs;
141 extern boolean_t do_print;

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

147 /* be_list.c */
148 int be_list(char *, be_node_list_t **);
149 int be_get_zone_be_list(char *, char *, be_node_list_t **);

151 /* be_mount.c */
152 int be_mount(char *, char **, int);
153 int be_unmount(char *, int);
154 int be_mount_pool(zfs_handle_t *, char **, char **, boolean_t *);
155 int be_unmount_pool(zfs_handle_t *, char **, char *);
156 int be_mount_zone_root(zfs_handle_t *, be_mount_data_t *);
157 int be_unmount_zone_root(zfs_handle_t *, be_unmount_data_t *);
158 int be_get_legacy_fs(char *, char *, char *, char *, be_fs_list_data_t *);
159 void be_free_fs_list(be_fs_list_data_t *);
160 char *be_get_ds_from_dir(char *);
161 int be_make_tmp_mountpoint(char **);

163 /* be_snapshot.c */
164 int be_create_snapshot(char *, char **, char *);
165 int be_destroy_snapshot(char *, char *);

167 /* be_utils.c */
168 boolean_t be_zfs_init(void);
169 void be_zfs_fini(void);

```

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

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

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

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

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

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

226 #ifdef __cplusplus
227 }
228 unchanged portion omitted
```

```

*****
14898 Tue Aug 6 21:14:57 2013
new/usr/src/man/man1m/beam.1m
*** NO COMMENTS ***
*****
1 \" te
2 .\" Copyright 2013 Nexenta Systems, Inc. All rights reserved.
3 .TH BEADM 1M \"Jul 25, 2013\"
2 .\" Copyright 2012 Nexenta Systems, Inc. All rights reserved.
3 .TH BEADM 1M \"Feb 26, 2011\"
4 .SH NAME
5 beam \- utility for managing zfs boot environments
6 .SH SYNOPSIS
7 .LP
8 .nf
9 \fBbeam\fR \fBcreate\fR [\fB-a\fR] [\fB-d\fR \fBdescription\fR]
10 [\fB-e\fR \fBnon-activeBeName\fR | \fBibeName@snapshot\fR]
11 [\fB-o\fR \fBproperty=value\fR] ... [\fB-p\fR \fBzpool\fR]
12 [\fB-v\fR] \fBibeName\fR
13 .fi

15 .LP
16 .nf
17 \fBbeam\fR \fBcreate\fR [\fB-v\fR] \fBibeName@snapshot\fR
18 .fi

20 .LP
21 .nf
22 \fBbeam\fR \fBdestroy\fR [\fB-fsv\fR] \fBibeName\fR | \fBibeName@snapshot\fR
23 .fi

25 .LP
26 .nf
27 \fBbeam\fR \fBlist\fR [\fB-a\fR | \fB-ds\fR] [\fB-H\fR] [\fB-v\fR] [\fBibeName\fR]
28 .fi

30 .LP
31 .nf
32 \fBbeam\fR \fBmount\fR [\fB-v\fR] \fBibeName\fR \fBimountpoint\fR
33 .fi

35 .LP
36 .nf
37 \fBbeam\fR \fBunmount\fR [\fB-fv\fR] \fBibeName\fR | \fBimountpoint\fR
38 .fi

40 .LP
41 .nf
42 \fBbeam\fR \fBrename\fR [\fB-v\fR] \fBibeName\fR \fBnewBeName\fR
43 .fi

45 .LP
46 .nf
47 \fBbeam\fR \fBactivate\fR [\fB-v\fR] \fBibeName\fR
48 .fi

50 .LP
51 .nf
52 \fBbeam\fR \fBrollback\fR [\fB-v\fR] \fBibeName\fR \fBisnapshot\fR
53 .fi

55 .LP
56 .nf
57 \fBbeam\fR \fBrollback\fR [\fB-v\fR] \fBibeName@snapshot\fR
58 .fi

```

```

60 .SH DESCRIPTION
61 The \fBbeam\fR command is the user interface for managing zfs Boot
62 Environments (BEs). This utility is intended to be used by System
63 Administrators who want to manage multiple Solaris Instances on a single
64 system.
65 .sp
66 The \fBbeam\fR command supports the following operations:
67 .RS +4
68 .TP
69 .ie t \(\bu
70 .el -
71 Create a new BE, based on the active BE.
72 .RE
73 .RS +4
74 .TP
75 .ie t \(\bu
76 .el -
77 Create a new BE, based on an inactive BE.
78 .RE
79 .RS +4
80 .TP
81 .ie t \(\bu
82 .el -
83 Create a snapshot of an existing BE.
84 .RE
85 .RS +4
86 .TP
87 .ie t \(\bu
88 .el -
89 Create a new BE, based on an existing snapshot.
90 .RE
91 .RS +4
92 .TP
93 .ie t \(\bu
94 .el -
95 Create a new BE, and copy it to a different zpool.
96 .RE
97 .RS +4
98 .TP
99 .ie t \(\bu
100 .el -
101 Activate an existing, inactive BE.
102 .RE
103 .RS +4
104 .TP
105 .ie t \(\bu
106 .el -
107 Mount a BE.
108 .RE
109 .RS +4
110 .TP
111 .ie t \(\bu
112 .el -
113 Unmount a BE.
114 .RE
115 .RS +4
116 .TP
117 .ie t \(\bu
118 .el -
119 Destroy a BE.
120 .RE
121 .RS +4
122 .TP
123 .ie t \(\bu
124 .el -
125 Destroy a snapshot of a BE.

```

```

126 .RE
127 .RS +4
128 .TP
129 .ie t \(\bu
130 .el -
131 Rename an existing, inactive BE.
132 .RE
133 .RS +4
134 .TP
135 .ie t \(\bu
136 .el -
137 Roll back a BE to an existing snapshot of a BE.
138 .RE
139 .RS +4
140 .TP
141 .ie t \(\bu
142 .el -
143 Display information about your snapshots and datasets.
144 .RE

146 .SH SUBCOMMANDS
147 The \fBbeadm\fR command has the subcommands and options listed
148 below. Also see
149 EXAMPLES below.
150 .sp
151 .ne 2
152 .na
153 \fBbeadm\fR
154 .ad
155 .sp .6
156 .RS 4n
157 Displays command usage.
158 .RE

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

168 .ad
169 .sp .6
170 .RS 4n
171 Creates a new boot environment named \fIbeName\fR. If the \fB-e\fR option is
172 not
173 provided, the new boot environment will be created as a clone of the
174 currently
175 running boot environment. If the \fB-d\fR option is provided then the
176 description is
177 also used as the title for the BE's entry in the GRUB menu for
178 x86 systems or
179 in the boot menu for SPARC systems. If the \fB-d\fR option is
180 not provided, \fIbeName\fR
181 will be used as the title.
182 .sp
183 .ne 2
184 .na
185 \fB-a\fR
186 .ad
187 .sp .6
188 .RS 4n
189 Activate the newly created BE upon creation. The default is to not activate
190 the newly created BE.
191 .RE

```

```

192 .sp
193 .ne 2
194 .na
195 \fB-d\fR \fIdescription\fR
196 .ad
197 .sp .6
198 .RS 4n
199 Create a new BE with a description associated with it.
200 .RE
201 .sp
202 .ne 2
203 .na
204 \fB-e\fR \fInon-activeBeName\fR
205 .ad
206 .sp .6
207 .RS 4n
208 Create a new BE from an existing inactive BE.
209 .RE
210 .sp
211 .ne 2
212 .na
213 \fB-e\fR \fIbeName@snapshot\fR
214 .ad
215 .sp .6
216 .RS 4n
217 Create a new BE from an existing snapshot of the BE named beName.
218 .RE
219 .sp
220 .ne 2
221 .na
222 \fB-o\fR \fIproperty=value\fR
223 .ad
224 .sp .6
225 .RS 4n
226 Create the datasets for new BE with specific ZFS properties. Multiple
227 \fB-o\fR
228 options can be specified. See \fBzfs\fR(1M) for more information on
229 the
230 \fB-o\fR option.
231 .RE
232 .sp
233 .ne 2
234 .na
235 \fB-p\fR \fIzpool\fR
236 .ad
237 .sp .6
238 .RS 4n
239 Create the new BE in the specified zpool. If this is not provided, the
240 default
241 behavior is to create the new BE in the same pool as as the origin BE.
242 This option is not supported in non-global zone.
243 .RE
244 .sp
245 .ne 2
246 .na
247 \fB-v\fR
248 .ad
249 .sp .6
250 .RS 4n
251 Verbose mode. Displays verbose error messages from \fBbeadm\fR.
252 .RE
253 .RE

255 .sp
256 .ne 2
257 .na

```

```

258 \fBbeadm\fR \fBcreate\fR [\fB-v\fR] \fIbeName@snapshot\fR
259 .ad
260 .sp .6
261 .RS 4n
262 Creates a snapshot of the existing BE named beName.
263 .sp
264 .ne 2
265 .na
266 \fB-v\fR
267 .ad
268 .sp .6
269 .RS 4n
270 Verbose mode. Displays verbose error messages from \fBbeadm\fR.
271 .RE
272 .RE

274 .sp
275 .ne 2
276 .na
277 \fBbeadm\fR \fBdestroy\fR [\fB-fFsv\fR] \fIbeName\fR | \fIbeName@snapshot\fR
278 .ad
279 .sp .6
280 .RS 4n
281 Destroys the boot environment named \fIbeName\fR or destroys an existing
282 snapshot of
283 the boot environment named \fIbeName@snapshot\fR. Destroying a
284 boot environment
285 will also destroy all snapshots of that boot environment. Use
286 this command
287 with caution.
288 .sp
289 .ne 2
290 .na
291 \fB-f\fR
292 .ad
293 .sp .6
294 .RS 4n
295 Forcefully unmount the boot environment if it is currently mounted.
296 .RE
297 .sp
298 .ne 2
299 .na
300 \fB-F\fR
301 .ad
302 .sp .6
303 .RS 4n
304 Force the action without prompting to verify the destruction of the boot
305 environment.
306 .RE
307 .sp
308 .ne 2
309 .na
310 \fB-s\fR
311 .ad
312 .sp .6
313 .RS 4n
314 Destroy all snapshots of the boot
315 environment.
316 .RE
317 .sp
318 .ne 2
319 .na
320 \fB-v\fR
321 .ad
322 .sp .6
323 .RS 4n

```

```

324 Verbose mode. Displays verbose error messages from \fBbeadm\fR.
325 .RE
326 .RE

328 .sp
329 .ne 2
330 .na
331 \fBbeadm\fR \fBlist\fR [\fB-a\fR | \fB-ds\fR] [\fB-H\fR] [\fB-v\fR] [\fIbeName\f
332 .ad
333 .sp .6
334 .RS 4n
335 Lists information about the existing boot environment named \fIbeName\fR, or
336 lists
337 information for all boot environments if \fIbeName\fR is not provided.
338 The 'Active'
339 field indicates whether the boot environment is active now,
340 represented
341 by 'N'; active on reboot, represented by 'R'; or both, represented
342 by 'NR'. In non-global zone the 'Active' field also indicates whether the
343 boot environment has a non-active parent BE, represented by 'x'; is active
344 on boot in a non-active parent BE, represented by 'b'. Activate, rollback
345 and snapshot operations for boot environments from non-active global parent
346 BE aren't supported, destroy is allowed if these boot environments aren't
347 active on boot.
348 .sp
349 Each line in the machine parsable output has the boot environment name as the
350 first field. The 'Space' field is displayed in bytes and the 'Created' field
351 is displayed in UTC format. The \fB-H\fR option used with no other options
352 gives
353 the boot environment's uuid in the second field. This field will be
354 blank if
355 the boot environment does not have a uuid. See the EXAMPLES section.
356 In non-global zones, this field shows the uuid of the parent BE.
357 .sp
358 .ne 2
359 .na
360 \fB-a\fR
361 .ad
362 .sp .6
363 .RS 4n
364 Lists all available information about the boot environment. This includes
365 subordinate file systems and snapshots.
366 .RE
367 .sp
368 .ne 2
369 .na
370 \fB-d\fR
371 .ad
372 .sp .6
373 .RS 4n
374 Lists information about all subordinate file systems belonging to the boot
375 environment.
376 .RE
377 .sp
378 .ne 2
379 .na
380 \fB-s\fR
381 .ad
382 .sp .6
383 .RS 4n
384 Lists information about the snapshots of the boot environment.
385 .RE
386 .sp
387 .ne 2
388 .na

```



```

389 \fB-H\fR
390 .ad
391 .sp .6
392 .RS 4n
393 Do not list header information. Each field in the list information is
394 separated by a semicolon.
395 .RE
396 .sp
397 .ne 2
398 .na
399 \fB-v\fR
400 .ad
401 .sp .6
402 .RS 4n
403 Verbose mode. Displays verbose error messages from \fBbeadm\fR.
404 .RE
405 .RE

407 .sp
408 .ne 2
409 .na
410 \fBbeadm\fR \fBmount\fR [\fB-v\fR] \fIbeName\fR \fImountpoint\fR
411 .ad
412 .sp .6
413 .RS 4n
414 Mounts a boot environment named beName at mountpoint. mountpoint must be an
415 already existing empty directory.
416 .sp
417 .ne 2
418 .na
419 \fB-v\fR
420 .ad
421 .sp .6
422 .RS 4n
423 Verbose mode. Displays verbose error messages from \fBbeadm\fR.
424 .RE
425 .RE

427 .sp
428 .ne 2
429 .na
430 \fBbeadm\fR \fBunmount\fR [\fB-fv\fR] \fIbeName\fR | \fImountpoint\fR
431 .ad
432 .sp .6
433 .RS 4n
434 Unmounts the boot environment named beName. The command can also be given a path
435 beName mount point on the system.
436 .sp
437 .ne 2
438 .na
439 \fB-f\fR
440 .ad
441 .sp .6
442 .RS 4n
443 Forcefully unmount the boot environment even if its currently busy.
444 .RE
445 .sp
446 .ne 2
447 .na
448 \fB-v\fR
449 .ad
450 .sp .6
451 .RS 4n
452 Verbose mode. Displays verbose error messages from \fBbeadm\fR.
453 .RE
454 .RE

```

```

456 .sp
457 .ne 2
458 .na
459 \fBbeadm\fR \fBrename\fR [\fB-v\fR] \fIbeName\fR \fInewBeName\fR
460 .ad
461 .sp .6
462 .RS 4n
463 Renames the boot environment named \fIbeName\fR to \fInewBeName\fR.
464 .sp
465 .ne 2
466 .na
467 \fB-v\fR
468 .ad
469 .sp .6
470 .RS 4n
471 Verbose mode. Displays verbose error messages from \fBbeadm\fR.
472 .RE
473 .RE

475 .sp
476 .ne 2
477 .na
478 \fBbeadm\fR \fBrollback\fR [\fB-v\fR] \fIbeName\fR \fIsnapshot\fR | \fIbeName@sn
479 .ad
480 .sp .6
481 .RS 4n
482 Roll back the boot environment named \fIbeName\fR to existing snapshot
483 of the boot environment named \fIbeName@snapshot\fR.
484 .sp
485 .ne 2
486 .na
487 \fB-v\fR
488 .ad
489 .sp .6
490 .RS 4n
491 Verbose mode. Displays verbose error messages from \fBbeadm\fR.
492 .RE
493 .RE

495 .sp
496 .ne 2
497 .na
498 \fBbeadm\fR \fBactivate\fR [\fB-v\fR] \fIbeName\fR
499 .ad
500 .sp .6
501 .RS 4n
502 Makes beName the active BE on next reboot.
503 .sp
504 .ne 2
505 .na
506 \fB-v\fR
507 .ad
508 .sp .6
509 .RS 4n
510 Verbose mode. Displays verbose error messages from \fBbeadm\fR.
511 .RE
512 .RE

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

520 .SH EXAMPLES

```

```

521 .LP
522 \fBExample 1\fR: Create a new BE named BE1, by cloning the current live BE.
523 .sp
524 .in +2
525 .nf
526 \fB# beadm create BE1\fR
527 .fi
528 .in -2
529 .sp

531 .LP
532 \fBExample 2\fR: Create a new BE named BE2, by cloning the existing inactive
533 BE
534 named BE1.
535 .sp
536 .in +2
537 .nf
538 \fB# beadm create -e BE1 BE2\fR
539 .fi
540 .in -2
541 .sp

543 .LP
544 \fBExample 3\fR: Create a snapshot named now of the existing BE named BE1.
545 .sp
546 .in +2
547 .nf
548 \fB# beadm create BE1@now\fR
549 .fi
550 .in -2
551 .sp

553 .LP
554 \fBExample 4\fR: Create a new BE named BE3, by cloning an existing snapshot of
555 BE1.
556 .sp
557 .in +2
558 .nf
559 \fB# beadm create -e BE1@now BE3\fR
560 .fi
561 .in -2
562 .sp

564 .LP
565 \fBExample 5\fR: Create a new BE named BE4 based on the currently running BE.
566 Create the new BE in rpool2.
567 .sp
568 .in +2
569 .nf
570 \fB# beadm create -p rpool2 BE4\fR
571 .fi
572 .in -2
573 .sp

575 .LP
576 \fBExample 6\fR: Create a new BE named BE5 based on the currently running BE.
577 Create the new BE in rpool2, and create its datasets with compression turned
578 on.
579 .sp
580 .in +2
581 .nf
582 \fB# beadm create -p rpool2 -o compression=on BE5\fR
583 .fi
584 .in -2
585 .sp

```

```

587 .LP
588 \fBExample 7\fR: Create a new BE named BE6 based on the currently running BE
589 and provide a description for it.
590 .sp
591 .in +2
592 .nf
593 \fB# beadm create -d "BE6 used as test environment" BE6\fR
594 .fi
595 .in -2
596 .sp

598 .LP
599 \fBExample 8\fR: Activate an existing, inactive BE named BE3.
600 .sp
601 .in +2
602 .nf
603 \fB# beadm activate BE3\fR
604 .fi
605 .in -2
606 .sp

608 .LP
609 \fBExample 9\fR: Mount the BE named BE3 at /mnt.
610 .sp
611 .in +2
612 .nf
613 \fB# beadm mount BE3 /mnt\fR
614 .fi
615 .in -2
616 .sp

618 .LP
619 \fBExample 10\fR: Unmount the mounted BE named BE3.
620 .sp
621 .in +2
622 .nf
623 \fB# beadm unmount BE3\fR
624 .fi
625 .in -2
626 .sp

628 .LP
629 \fBExample 11\fR: Destroy the BE named BE3 without verification.
630 .sp
631 .in +2
632 .nf
633 \fB# beadm destroy -f BE3\fR
634 .fi
635 .in -2
636 .sp

638 .LP
639 \fBExample 12\fR: Destroy the snapshot named now of BE1.
640 .sp
641 .in +2
642 .nf
643 \fB# beadm destroy BE1@now\fR
644 .fi
645 .in -2
646 .sp

648 .LP
649 \fBExample 13\fR: Rename the existing, inactive BE named BE1 to BE3.
650 .sp
651 .in +2
652 .nf

```

```

653 \fB# beadm rename BE1 BE3\fR
654 .fi
655 .in -2
656 .sp

658 .LP
659 \fBExample 14\fR: Roll back the BE named BE1 to snapshot BE1@now.
660 .sp
661 .in +2
662 .nf
663 \fB# beadm rollback BE1 BE1@now\fR
664 .fi
665 .in -2
666 .sp

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

671 .sp
672 .in +2
673 .nf
674 \fB# beadm list\fR
675 BE Active Mountpoint Space Policy Created
676 --
677 BE2 - - 72.0K static 2008-05-21 12:26
678 BE3 - - 332.0K static 2008-08-26 10:28
679 BE4 - - 15.78M static 2008-09-05 18:20
680 BE5 NR / 7.25G static 2008-09-09 16:53
681 .fi
682 .in -2
683 .sp

685 .LP
686 \fBExample 16\fR: List all existing boot environments and list all dataset and
687 snapshot information about those boot environments.

689 .sp
690 .in +2
691 .nf
692 \fB# beadm list -d -s\fR

694 BE/Dataset/Snapshot Active Mountpoint Space Policy Created
695 -----
696 BE2
697 p/ROOT/BE2 - - 36.0K static 2008-05-21 12:26
698 p/ROOT/BE2/opt - - 18.0K static 2008-05-21 16:26
699 p/ROOT/BE2/opt@now - - 0 static 2008-09-08 22:43
700 p/ROOT/BE2@now - - 0 static 2008-09-08 22:43
701 BE3
702 p/ROOT/BE3 - - 192.0K static 2008-08-26 10:28
703 p/ROOT/BE3/opt - - 86.0K static 2008-08-26 10:28
704 p/ROOT/BE3/opt/local - - 36.0K static 2008-08-28 10:58
705 BE4
706 p/ROOT/BE4 - - 15.78M static 2008-09-05 18:20
707 BE5
708 p/ROOT/BE5 NR / 6.10G static 2008-09-09 16:53
709 p/ROOT/BE5/opt - /opt 24.55M static 2008-09-09 16:53
710 p/ROOT/BE5/opt@bar - - 18.38M static 2008-09-10 00:59
711 p/ROOT/BE5/opt@foo - - 18.38M static 2008-06-10 16:37
712 p/ROOT/BE5@bar - - 139.44M static 2008-09-10 00:59
713 p/ROOT/BE5@foo - - 912.85M static 2008-06-10 16:37
714 .fi
715 .in -2
716 .sp

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

```

```

720 .sp
721 .in +2
722 .nf
723 \fB# beadm list -a BE5\fR

725 BE/Dataset/Snapshot Active Mountpoint Space Policy Created
726 -----
727 BE5
728 p/ROOT/BE5 NR / 6.10G static 2008-09-09 16:53
729 p/ROOT/BE5/opt - /opt 24.55M static 2008-09-09 16:53
730 p/ROOT/BE5/opt@bar - - 18.38M static 2008-09-10 00:59
731 p/ROOT/BE5/opt@foo - - 18.38M static 2008-06-10 16:37
732 p/ROOT/BE5@bar - - 139.44M static 2008-09-10 00:59
733 p/ROOT/BE5@foo - - 912.85M static 2008-06-10 16:37
734 .fi
735 .in -2
736 .sp

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

742 .sp
743 .in +2
744 .nf
745 \fB# beadm list -H\fR

747 BE2;;;55296;static;1211397974
748 BE3;;;339968;static;1219771706
749 BE4;;;16541696;static;1220664051
750 BE5;215b8387-4968-627c-d2d0-f4a011414bab;NR;;7786206208;static;1221004384
751 .fi
752 .in -2
753 .sp

755 .SH EXIT STATUS
756 .sp
757 .LP
758 The following exit values are returned:
759 .sp
760 .ne 2
761 .na
762 \fB0\fR
763 .ad
764 .sp .6
765 .RS 4n
766 Successful completion
767 .RE

769 .sp
770 .ne 2
771 .na
772 \fB>0\fR
773 .ad
774 .sp .6
775 .RS 4n
776 Failure
777 .RE

780 .SH FILES
781 .sp
782 .LP
783 .sp
784 .ne 2

```

```
785 .na
786 \fB/var/log/beadm/<beName>/create.log.<yyyymmdd_hhmmss>\fR
787 .ad
788 .sp .6
789 .RS 4n
790 Log used for capturing beadm create output
791 .sp
792 .nf
793 \fIyyyymmdd_hhmmss\fR - 20071130_140558
794 \fIyy\fR - year; 2007
795 \fImm\fR - month; 11
796 \fIdd\fR - day; 30
797 \fIhh\fR - hour; 14
798 \fImm\fR - minute; 05
799 \fIss\fR - second; 58
800 .fi
801 .in -2
802 .sp
803 .RE
804 .sp
805 .LP
806 .sp
807 .ne 2
808 .na
809 \fB/etc/default/be\fR
810 .ad
811 .sp .6
812 .RS 4n
813 Contains default value for BENAME_STARTS_WITH parameter
814 .sp
815 .RE

817 .SH ATTRIBUTES
818 .sp
819 .LP
820 See \fBattributes\fR(5) for descriptions of the following attributes:
821 .sp

823 .sp
824 .TS
825 box;
826 c | c
827 l | l .
828 ATTRIBUTE TYPE ATTRIBUTE VALUE
829 -
830 Interface Stability Uncommitted
831 .TE

834 .SH SEE ALSO
835 .sp
836 .LP
837 .BR zfs (1M)
```