

```

*****
10388 Fri Sep 11 13:41:19 2015
new/usr/src/lib/libzfs/common/libzfs_config.c
6223 libzfs improperly uses an avl tree in namespace_reload
*****
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 2009 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25 */

27 /*
28  * Copyright (c) 2012 by Delphix. All rights reserved.
29  * Copyright 2015 Nexenta Systems, Inc. All rights reserved.
30 #endif /* !codereview */
31 */

33 /*
34  * The pool configuration repository is stored in /etc/zfs/zpool.cache as a
35  * single packed nvlist. While it would be nice to just read in this
36  * file from userland, this wouldn't work from a local zone. So we have to have
37  * a zpool ioctl to return the complete configuration for all pools. In the
38  * global zone, this will be identical to reading the file and unpacking it in
39  * userland.
40 */

42 #include <errno.h>
43 #include <sys/stat.h>
44 #include <fcntl.h>
45 #include <stddef.h>
46 #include <string.h>
47 #include <unistd.h>
48 #include <libintl.h>
49 #include <libuutil.h>

51 #include "libzfs_impl.h"

53 typedef struct config_node {
54     char      *cn_name;
55     nvlist_t  *cn_config;
56     uu_avl_node_t  cn_avl;
57 } config_node_t;

59 /* ARGSUSED */
60 static int
61 config_node_compare(const void *a, const void *b, void *unused)

```

```

62 {
63     int ret;

65     const config_node_t *ca = (config_node_t *)a;
66     const config_node_t *cb = (config_node_t *)b;

68     ret = strcmp(ca->cn_name, cb->cn_name);

70     if (ret < 0)
71         return (-1);
72     else if (ret > 0)
73         return (1);
74     else
75         return (0);
76 }

78 void
79 namespace_clear(libzfs_handle_t *hdl)
80 {
81     if (hdl->libzfs_ns_avl) {
82         config_node_t *cn;
83         void *cookie = NULL;

85         while ((cn = uu_avl_takedown(hdl->libzfs_ns_avl,
86             &cookie)) != NULL) {
87             nvlist_free(cn->cn_config);
88             free(cn->cn_name);
89             free(cn);
90         }

92         uu_avl_destroy(hdl->libzfs_ns_avl);
93         hdl->libzfs_ns_avl = NULL;
94     }

96     if (hdl->libzfs_ns_avlpool) {
97         uu_avl_pool_destroy(hdl->libzfs_ns_avlpool);
98         hdl->libzfs_ns_avlpool = NULL;
99     }
100 }

102 /*
103  * Loads the pool namespace, or re-loads it if the cache has changed.
104  */
105 static int
106 namespace_reload(libzfs_handle_t *hdl)
107 {
108     nvlist_t *config;
109     config_node_t *cn;
110     nvpair_t *elem;
111     zfs_cmd_t zc = { 0 };
112     void *cookie;

114     if (hdl->libzfs_ns_gen == 0) {
115         /*
116          * This is the first time we've accessed the configuration
117          * cache. Initialize the AVL tree and then fall through to the
118          * common code.
119          */
120         if ((hdl->libzfs_ns_avlpool = uu_avl_pool_create("config_pool",
121             sizeof (config_node_t),
122             offsetof(config_node_t, cn_avl),
123             config_node_compare, UU_DEFAULT)) == NULL)
124             return (no_memory(hdl));

126         if ((hdl->libzfs_ns_avl = uu_avl_create(hdl->libzfs_ns_avlpool,
127             NULL, UU_DEFAULT)) == NULL)

```

```

128         return (no_memory(hdl));
129     }

131     if (zcmd_alloc_dst_nvlist(hdl, &zdc, 0) != 0)
132         return (-1);

134     for (;;) {
135         zc.zc_cookie = hdl->libzfs_ns_gen;
136         if (ioctl(hdl->libzfs_fd, ZFS_IOC_POOL_CONFIGS, &zdc) != 0) {
137             switch (errno) {
138                 case EEXIST:
139                     /*
140                      * The namespace hasn't changed.
141                      */
142                     zcmd_free_nvlists(&zdc);
143                     return (0);

145                 case ENOMEM:
146                     if (zcmd_expand_dst_nvlist(hdl, &zdc) != 0) {
147                         zcmd_free_nvlists(&zdc);
148                         return (-1);
149                     }
150                     break;

152                 default:
153                     zcmd_free_nvlists(&zdc);
154                     return (zfs_standard_error(hdl, errno,
155                         dgettext(TEXT_DOMAIN, "failed to read "
156                             "pool configuration")));
157             }
158         } else {
159             hdl->libzfs_ns_gen = zc.zc_cookie;
160             break;
161         }
162     }

164     if (zcmd_read_dst_nvlist(hdl, &zdc, &config) != 0) {
165         zcmd_free_nvlists(&zdc);
166         return (-1);
167     }

169     zcmd_free_nvlists(&zdc);

171     /*
172     * Clear out any existing configuration information, and recreate
173     * the AVL tree.
174     * Clear out any existing configuration information.
175     */
176     cookie = NULL;
177     while ((cn = uu_avl_takedown(hdl->libzfs_ns_avl, &cookie)) != NULL) {
178         nvlist_free(cn->cn_config);
179         free(cn->cn_name);
180         free(cn);
181     }

182     uu_avl_recreate(hdl->libzfs_ns_avl);
183 #endif /* !codereview */

185     elem = NULL;
186     while ((elem = nvlist_next_nvpair(config, elem)) != NULL) {
187         nvlist_t *child;
188         uu_avl_index_t where;

190         if ((cn = zfs_alloc(hdl, sizeof (config_node_t))) == NULL) {
191             nvlist_free(config);
192             return (-1);

```

```

193     }

195     if ((cn->cn_name = zfs_strdup(hdl,
196         nvpair_name(elem))) == NULL) {
197         free(cn);
198         nvlist_free(config);
199         return (-1);
200     }

202     verify(nvpair_value_nvlist(elem, &child) == 0);
203     if (nvlist_dup(child, &cn->cn_config, 0) != 0) {
204         free(cn->cn_name);
205         free(cn);
206         nvlist_free(config);
207         return (no_memory(hdl));
208     }
209     verify(uu_avl_find(hdl->libzfs_ns_avl, cn, NULL, &where)
210         == NULL);

212     uu_avl_insert(hdl->libzfs_ns_avl, cn, where);
213 }

215     nvlist_free(config);
216     return (0);
217 }

219 /*
220 * Retrieve the configuration for the given pool. The configuration is a nvlist
221 * describing the vdevs, as well as the statistics associated with each one.
222 */
223 nvlist_t *
224 zpool_get_config(zpool_handle_t *zhp, nvlist_t **oldconfig)
225 {
226     if (oldconfig)
227         *oldconfig = zhp->zpool_old_config;
228     return (zhp->zpool_config);
229 }

231 /*
232 * Retrieves a list of enabled features and their refcounts and caches it in
233 * the pool handle.
234 */
235 nvlist_t *
236 zpool_get_features(zpool_handle_t *zhp)
237 {
238     nvlist_t *config, *features;

240     config = zpool_get_config(zhp, NULL);

242     if (config == NULL || !nvlist_exists(config,
243         ZPOOL_CONFIG_FEATURE_STATS)) {
244         int error;
245         boolean_t missing = B_FALSE;

247         error = zpool_refresh_stats(zhp, &missing);

249         if (error != 0 || missing)
250             return (NULL);

252         config = zpool_get_config(zhp, NULL);
253     }

255     verify(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_FEATURE_STATS,
256         &features) == 0);

258     return (features);

```

```

259 }
261 /*
262  * Refresh the vdev statistics associated with the given pool. This is used in
263  * iostat to show configuration changes and determine the delta from the last
264  * time the function was called. This function can fail, in case the pool has
265  * been destroyed.
266  */
267 int
268 zpool_refresh_stats(zpool_handle_t *zhp, boolean_t *missing)
269 {
270     zfs_cmd_t zc = { 0 };
271     int error;
272     nvlist_t *config;
273     libzfs_handle_t *hdl = zhp->zpool_hdl;
275     *missing = B_FALSE;
276     (void) strcpy(zc.zc_name, zhp->zpool_name);
278     if (zhp->zpool_config_size == 0)
279         zhp->zpool_config_size = 1 << 16;
281     if (zcmd_alloc_dst_nvlist(hdl, &zc, zhp->zpool_config_size) != 0)
282         return (-1);
284     for (;;) {
285         if (ioctl(zhp->zpool_hdl->libzfs_fd, ZFS_IOC_POOL_STATS,
286             &zc) == 0) {
287             /*
288              * The real error is returned in the zc_cookie field.
289              */
290             error = zc.zc_cookie;
291             break;
292         }
294         if (errno == ENOMEM) {
295             if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
296                 zcmd_free_nvlists(&zc);
297                 return (-1);
298             }
299         } else {
300             zcmd_free_nvlists(&zc);
301             if (errno == ENOENT || errno == EINVAL)
302                 *missing = B_TRUE;
303             zhp->zpool_state = POOL_STATE_UNAVAIL;
304             return (0);
305         }
306     }
308     if (zcmd_read_dst_nvlist(hdl, &zc, &config) != 0) {
309         zcmd_free_nvlists(&zc);
310         return (-1);
311     }
313     zcmd_free_nvlists(&zc);
315     zhp->zpool_config_size = zc.zc_nvlist_dst_size;
317     if (zhp->zpool_config != NULL) {
318         uint64_t oldtxg, newtxg;
320         verify(nvlist_lookup_uint64(zhp->zpool_config,
321             ZPOOL_CONFIG_POOL_TXG, &oldtxg) == 0);
322         verify(nvlist_lookup_uint64(config,
323             ZPOOL_CONFIG_POOL_TXG, &newtxg) == 0);

```

```

325         if (zhp->zpool_old_config != NULL)
326             nvlist_free(zhp->zpool_old_config);
328         if (oldtxg != newtxg) {
329             nvlist_free(zhp->zpool_config);
330             zhp->zpool_old_config = NULL;
331         } else {
332             zhp->zpool_old_config = zhp->zpool_config;
333         }
334     }
336     zhp->zpool_config = config;
337     if (error)
338         zhp->zpool_state = POOL_STATE_UNAVAIL;
339     else
340         zhp->zpool_state = POOL_STATE_ACTIVE;
342     return (0);
343 }
345 /*
346  * If the __ZFS_POOL_RESTRICT environment variable is set we only iterate over
347  * pools it lists.
348  *
349  * This is an undocumented feature for use during testing only.
350  *
351  * This function returns B_TRUE if the pool should be skipped
352  * during iteration.
353  */
354 static boolean_t
355 check_restricted(const char *poolname)
356 {
357     static boolean_t initialized = B_FALSE;
358     static char *restricted = NULL;
360     const char *cur, *end;
361     int len, namelen;
363     if (!initialized) {
364         initialized = B_TRUE;
365         restricted = getenv("__ZFS_POOL_RESTRICT");
366     }
368     if (NULL == restricted)
369         return (B_FALSE);
371     cur = restricted;
372     namelen = strlen(poolname);
373     do {
374         end = strchr(cur, ' ');
375         len = (NULL == end) ? strlen(cur) : (end - cur);
377         if (len == namelen && 0 == strcmp(cur, poolname, len)) {
378             return (B_FALSE);
379         }
381         cur += (len + 1);
382     } while (NULL != end);
384     return (B_TRUE);
385 }
387 /*
388  * Iterate over all pools in the system.
389  */
390 int

```

```

391 zpool_iter(libzfs_handle_t *hdl, zpool_iter_f func, void *data)
392 {
393     config_node_t *cn;
394     zpool_handle_t *zhp;
395     int ret;
396
397     /*
398      * If someone makes a recursive call to zpool_iter(), we want to avoid
399      * refreshing the namespace because that will invalidate the parent
400      * context. We allow recursive calls, but simply re-use the same
401      * namespace AVL tree.
402      */
403     if (!hdl->libzfs_pool_iter && namespace_reload(hdl) != 0)
404         return (-1);
405
406     hdl->libzfs_pool_iter++;
407     for (cn = uu_avl_first(hdl->libzfs_ns_avl); cn != NULL;
408          cn = uu_avl_next(hdl->libzfs_ns_avl, cn)) {
409
410         if (check_restricted(cn->cn_name))
411             continue;
412
413         if (zpool_open_silent(hdl, cn->cn_name, &zhp) != 0) {
414             hdl->libzfs_pool_iter--;
415             return (-1);
416         }
417
418         if (zhp == NULL)
419             continue;
420
421         if ((ret = func(zhp, data)) != 0) {
422             hdl->libzfs_pool_iter--;
423             return (ret);
424         }
425     }
426     hdl->libzfs_pool_iter--;
427
428     return (0);
429 }
430
431 /*
432  * Iterate over root datasets, calling the given function for each. The zfs
433  * handle passed each time must be explicitly closed by the callback.
434  */
435 int
436 zfs_iter_root(libzfs_handle_t *hdl, zfs_iter_f func, void *data)
437 {
438     config_node_t *cn;
439     zfs_handle_t *zhp;
440     int ret;
441
442     if (namespace_reload(hdl) != 0)
443         return (-1);
444
445     for (cn = uu_avl_first(hdl->libzfs_ns_avl); cn != NULL;
446          cn = uu_avl_next(hdl->libzfs_ns_avl, cn)) {
447
448         if (check_restricted(cn->cn_name))
449             continue;
450
451         if ((zhp = make_dataset_handle(hdl, cn->cn_name)) == NULL)
452             continue;
453
454         if ((ret = func(zhp, data)) != 0)
455             return (ret);
456     }

```

```

458     return (0);
459 }

```