```
**********************************************************
   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
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

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

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

#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stddef.h>
#include <string.h>
#include <unistd.h>
#include <libintl.h>
#include <libuutil.h>

#include "libzfs_impl.h"

typedef struct config_node {
        char            *cn_name;
        nvlist_t        *cn_config;
        uu_avl_node_t   cn_avl;
} config_node_t;

/* ARGSUSED */
static int
config_node_compare(const void *a, const void *b, void *unused)
```

```
{
        int ret;

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

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

        if (ret < 0)
                return (-1);
        else if (ret > 0)
                return (1);
        else
                return (0);
}

void
namespace_clear(libzfs_handle_t *hdl)
{
        if (hdl->libzfs_ns_avl) {
                config_node_t *cn;
                void *cookie = NULL;

                while ((cn = uu_avl_teardown(hdl->libzfs_ns_avl,
                    &cookie)) != NULL) {
                        nvlist_free(cn->cn_config);
                        free(cn->cn_name);
                        free(cn);
                }

                uu_avl_destroy(hdl->libzfs_ns_avl);
                hdl->libzfs_ns_avl = NULL;
        }

        if (hdl->libzfs_ns_avlpool) {
                uu_avl_pool_destroy(hdl->libzfs_ns_avlpool);
                hdl->libzfs_ns_avlpool = NULL;
        }
}

/*
 * Loads the pool namespace, or re-loads it if the cache has changed.
 */
static int
namespace_reload(libzfs_handle_t *hdl)
{
        nvlist_t *config;
        config_node_t *cn;
        nvpair_t *elem;
        zfs_cmd_t zc = { 0 };
        void *cookie;

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

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

```
                        return (no_memory(hdl));
        }

        if (zcmd_alloc_dst_nvlist(hdl, &zc, 0) != 0)
                return (-1);

        for (;;) {
                zc.zc_cookie = hdl->libzfs_ns_gen;
                if (ioctl(hdl->libzfs_fd, ZFS_IOC_POOL_CONFIGS, &zc) != 0) {
                        switch (errno) {
                        case EEXIST:
                                /*
                                 * The namespace hasn't changed.
                                 */
                                zcmd_free_nvlists(&zc);
                                return (0);

                        case ENOMEM:
                                if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
                                        zcmd_free_nvlists(&zc);
                                        return (-1);
                                }
                                break;

                        default:
                                zcmd_free_nvlists(&zc);
                                return (zfs_standard_error(hdl, errno,
                                    dgettext(TEXT_DOMAIN, "failed to read "
                                    "pool configuration")));
                        }
                } else {
                        hdl->libzfs_ns_gen = zc.zc_cookie;
                        break;
                }
        }

        if (zcmd_read_dst_nvlist(hdl, &zc, &config) != 0) {
                zcmd_free_nvlists(&zc);
                return (-1);
        }

        zcmd_free_nvlists(&zc);

        /*
         * Clear out any existing configuration information, and recreate
         * the AVL tree.
         * Clear out any existing configuration information.
         */
        cookie = NULL;
        while ((cn = uu_avl_teardown(hdl->libzfs_ns_avl, &cookie)) != NULL) {
                nvlist_free(cn->cn_config);
                free(cn->cn_name);
                free(cn);
        }

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

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

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

```
                }

                if ((cn->cn_name = zfs_strdup(hdl,
                    nvpair_name(elem))) == NULL) {
                        free(cn);
                        nvlist_free(config);
                        return (-1);
                }

                verify(nvpair_value_nvlist(elem, &child) == 0);
                if (nvlist_dup(child, &cn->cn_config, 0) != 0) {
                        free(cn->cn_name);
                        free(cn);
                        nvlist_free(config);
                        return (no_memory(hdl));
                }
                verify(uu_avl_find(hdl->libzfs_ns_avl, cn, NULL, &where)
                    == NULL);

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

        nvlist_free(config);
        return (0);
}

/*
 * Retrieve the configuration for the given pool.  The configuration is a nvlist
 * describing the vdevs, as well as the statistics associated with each one.
 */
nvlist_t *
zpool_get_config(zpool_handle_t *zhp, nvlist_t **oldconfig)
{
        if (oldconfig)
                *oldconfig = zhp->zpool_old_config;
        return (zhp->zpool_config);
}

/*
 * Retrieves a list of enabled features and their refcounts and caches it in
 * the pool handle.
 */
nvlist_t *
zpool_get_features(zpool_handle_t *zhp)
{
        nvlist_t *config, *features;

        config = zpool_get_config(zhp, NULL);

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

                error = zpool_refresh_stats(zhp, &missing);

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

                config = zpool_get_config(zhp, NULL);
        }

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

        return (features);
```

```c
}

/*
 * Refresh the vdev statistics associated with the given pool.  This is used in
 * iostat to show configuration changes and determine the delta from the last
 * time the function was called.  This function can fail, in case the pool has
 * been destroyed.
 */
int
zpool_refresh_stats(zpool_handle_t *zhp, boolean_t *missing)
{
        zfs_cmd_t zc = { 0 };
        int error;
        nvlist_t *config;
        libzfs_handle_t *hdl = zhp->zpool_hdl;

        *missing = B_FALSE;
        (void) strcpy(zc.zc_name, zhp->zpool_name);

        if (zhp->zpool_config_size == 0)
                zhp->zpool_config_size = 1 << 16;

        if (zcmd_alloc_dst_nvlist(hdl, &zc, zhp->zpool_config_size) != 0)
                return (-1);

        for (;;) {
                if (ioctl(zhp->zpool_hdl->libzfs_fd, ZFS_IOC_POOL_STATS,
                    &zc) == 0) {
                        /*
                         * The real error is returned in the zc_cookie field.
                         */
                        error = zc.zc_cookie;
                        break;
                }

                if (errno == ENOMEM) {
                        if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
                                zcmd_free_nvlists(&zc);
                                return (-1);
                        }
                } else {
                        zcmd_free_nvlists(&zc);
                        if (errno == ENOENT || errno == EINVAL)
                                *missing = B_TRUE;
                        zhp->zpool_state = POOL_STATE_UNAVAIL;
                        return (0);
                }
        }

        if (zcmd_read_dst_nvlist(hdl, &zc, &config) != 0) {
                zcmd_free_nvlists(&zc);
                return (-1);
        }

        zcmd_free_nvlists(&zc);

        zhp->zpool_config_size = zc.zc_nvlist_dst_size;

        if (zhp->zpool_config != NULL) {
                uint64_t oldtxg, newtxg;

                verify(nvlist_lookup_uint64(zhp->zpool_config,
                    ZPOOL_CONFIG_POOL_TXG, &oldtxg) == 0);
                verify(nvlist_lookup_uint64(config,
                    ZPOOL_CONFIG_POOL_TXG, &newtxg) == 0);
```

```c
                if (zhp->zpool_old_config != NULL)
                        nvlist_free(zhp->zpool_old_config);

                if (oldtxg != newtxg) {
                        nvlist_free(zhp->zpool_config);
                        zhp->zpool_old_config = NULL;
                } else {
                        zhp->zpool_old_config = zhp->zpool_config;
                }
        }

        zhp->zpool_config = config;
        if (error)
                zhp->zpool_state = POOL_STATE_UNAVAIL;
        else
                zhp->zpool_state = POOL_STATE_ACTIVE;

        return (0);
}

/*
 * If the __ZFS_POOL_RESTRICT environment variable is set we only iterate over
 * pools it lists.
 *
 * This is an undocumented feature for use during testing only.
 *
 * This function returns B_TRUE if the pool should be skipped
 * during iteration.
 */
static boolean_t
check_restricted(const char *poolname)
{
        static boolean_t initialized = B_FALSE;
        static char *restricted = NULL;

        const char *cur, *end;
        int len, namelen;

        if (!initialized) {
                initialized = B_TRUE;
                restricted = getenv("__ZFS_POOL_RESTRICT");
        }

        if (NULL == restricted)
                return (B_FALSE);

        cur = restricted;
        namelen = strlen(poolname);
        do {
                end = strchr(cur, ' ');
                len = (NULL == end) ? strlen(cur) : (end - cur);

                if (len == namelen && 0 == strncmp(cur, poolname, len)) {
                        return (B_FALSE);
                }

                cur += (len + 1);
        } while (NULL != end);

        return (B_TRUE);
}

/*
 * Iterate over all pools in the system.
 */
int
```

```
zpool_iter(libzfs_handle_t *hdl, zpool_iter_f func, void *data)
{
        config_node_t *cn;
        zpool_handle_t *zhp;
        int ret;

        /*
         * If someone makes a recursive call to zpool_iter(), we want to avoid
         * refreshing the namespace because that will invalidate the parent
         * context.  We allow recursive calls, but simply re-use the same
         * namespace AVL tree.
         */
        if (!hdl->libzfs_pool_iter && namespace_reload(hdl) != 0)
                return (-1);

        hdl->libzfs_pool_iter++;
        for (cn = uu_avl_first(hdl->libzfs_ns_avl); cn != NULL;
            cn = uu_avl_next(hdl->libzfs_ns_avl, cn)) {

                if (check_restricted(cn->cn_name))
                        continue;

                if (zpool_open_silent(hdl, cn->cn_name, &zhp) != 0) {
                        hdl->libzfs_pool_iter--;
                        return (-1);
                }

                if (zhp == NULL)
                        continue;

                if ((ret = func(zhp, data)) != 0) {
                        hdl->libzfs_pool_iter--;
                        return (ret);
                }
        }
        hdl->libzfs_pool_iter--;

        return (0);
}

/*
 * Iterate over root datasets, calling the given function for each.  The zfs
 * handle passed each time must be explicitly closed by the callback.
 */
int
zfs_iter_root(libzfs_handle_t *hdl, zfs_iter_f func, void *data)
{
        config_node_t *cn;
        zfs_handle_t *zhp;
        int ret;

        if (namespace_reload(hdl) != 0)
                return (-1);

        for (cn = uu_avl_first(hdl->libzfs_ns_avl); cn != NULL;
            cn = uu_avl_next(hdl->libzfs_ns_avl, cn)) {

                if (check_restricted(cn->cn_name))
                        continue;

                if ((zhp = make_dataset_handle(hdl, cn->cn_name)) == NULL)
                        continue;

                if ((ret = func(zhp, data)) != 0)
                        return (ret);
        }
```

```

        return (0);
}
```