```
**********************************************************
  111785 Mon Aug 12 01:16:02 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.c
3996 want a libzfs_core API to rollback to latest snapshot
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

/*
 * Given a dataset, rollback to a specific snapshot, discarding any
 * data changes since then and making it the active dataset.
 *
 * Any snapshots more recent than the target are destroyed, along with
 * their dependents.
 */
int
zfs_rollback(zfs_handle_t *zhp, zfs_handle_t *snap, boolean_t force)
{
        rollback_data_t cb = { 0 };
        int err;
        zfs_cmd_t zc = { 0 };
        boolean_t restore_resv = 0;
        uint64_t old_volsize, new_volsize;
        zfs_prop_t resv_prop;

        assert(zhp->zfs_type == ZFS_TYPE_FILESYSTEM ||
            zhp->zfs_type == ZFS_TYPE_VOLUME);

        /*
         * Destroy all recent snapshots and their dependents.
         */
        cb.cb_force = force;
        cb.cb_target = snap->zfs_name;
        cb.cb_create = zfs_prop_get_int(snap, ZFS_PROP_CREATETXG);
        (void) zfs_iter_children(zhp, rollback_destroy, &cb);

        if (cb.cb_error)
                return (-1);

        /*
         * Now that we have verified that the snapshot is the latest,
         * rollback to the given snapshot.
         */

        if (zhp->zfs_type == ZFS_TYPE_VOLUME) {
                if (zfs_which_resv_prop(zhp, &resv_prop) < 0)
                        return (-1);
                old_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
                restore_resv =
                    (old_volsize == zfs_prop_get_int(zhp, resv_prop));
        }

        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

        if (ZFS_IS_VOLUME(zhp))
                zc.zc_objset_type = DMU_OST_ZVOL;
        else
                zc.zc_objset_type = DMU_OST_ZFS;

        /*
         * We rely on zfs_iter_children() to verify that there are no
         * newer snapshots for the given dataset.  Therefore, we can
         * simply pass the name on to the ioctl() call.  There is still
         * an unlikely race condition where the user has taken a
```

```
         * snapshot since we verified that this was the most recent.
         *
         */
        err = lzc_rollback(zhp->zfs_name, NULL, 0);
        if (err != 0) {
        if ((err = zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_ROLLBACK, &zc)) != 0) {
                (void) zfs_standard_error_fmt(zhp->zfs_hdl, errno,
                    dgettext(TEXT_DOMAIN, "cannot rollback '%s'"),
                    zhp->zfs_name);
                return (err);
        }

        /*
         * For volumes, if the pre-rollback volsize matched the pre-
         * rollback reservation and the volsize has changed then set
         * the reservation property to the post-rollback volsize.
         * Make a new handle since the rollback closed the dataset.
         */
        if ((zhp->zfs_type == ZFS_TYPE_VOLUME) &&
            (zhp = make_dataset_handle(zhp->zfs_hdl, zhp->zfs_name))) {
                if (restore_resv) {
                        new_volsize = zfs_prop_get_int(zhp, ZFS_PROP_VOLSIZE);
                        if (old_volsize != new_volsize)
                                err = zfs_prop_set_int(zhp, resv_prop,
                                    new_volsize);
                }
                zfs_close(zhp);
        }
        return (err);
}
______unchanged_portion_omitted_
```

```
************************************************************
   17237 Mon Aug 12 01:16:03 2013
new/usr/src/lib/libzfs_core/common/libzfs_core.c
3996 want a libzfs_core API to rollback to latest snapshot
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
************************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2013 Steven Hartland. All rights reserved.
 */

/*
 * LibZFS_Core (lzc) is intended to replace most functionality in libzfs.
 * It has the following characteristics:
 *
 *  - Thread Safe.  libzfs_core is accessible concurrently from multiple
 *  threads.  This is accomplished primarily by avoiding global data
 *  (e.g. caching).  Since it's thread-safe, there is no reason for a
 *  process to have multiple libzfs "instances".  Therefore, we store
 *  our few pieces of data (e.g. the file descriptor) in global
 *  variables.  The fd is reference-counted so that the libzfs_core
 *  library can be "initialized" multiple times (e.g. by different
 *  consumers within the same process).
 *
 *  - Committed Interface.  The libzfs_core interface will be committed,
 *  therefore consumers can compile against it and be confident that
 *  their code will continue to work on future releases of this code.
 *  Currently, the interface is Evolving (not Committed), but we intend
 *  to commit to it once it is more complete and we determine that it
 *  meets the needs of all consumers.
 *
 *  - Programatic Error Handling.  libzfs_core communicates errors with
 *  defined error numbers, and doesn't print anything to stdout/stderr.
 *
 *  - Thin Layer.  libzfs_core is a thin layer, marshaling arguments
 *  to/from the kernel ioctls.  There is generally a 1:1 correspondence
 *  between libzfs_core functions and ioctls to /dev/zfs.
 *
 *  - Clear Atomicity.  Because libzfs_core functions are generally 1:1
 *  with kernel ioctls, and kernel ioctls are general atomic, each
 *  libzfs_core function is atomic.  For example, creating multiple
 *  snapshots with a single call to lzc_snapshot() is atomic -- it
```

```
 *  can't fail with only some of the requested snapshots created, even
 *  in the event of power loss or system crash.
 *
 *  - Continued libzfs Support.  Some higher-level operations (e.g.
 *  support for "zfs send -R") are too complicated to fit the scope of
 *  libzfs_core.  This functionality will continue to live in libzfs.
 *  Where appropriate, libzfs will use the underlying atomic operations
 *  of libzfs_core.  For example, libzfs may implement "zfs send -R |
 *  zfs receive" by using individual "send one snapshot", rename,
 *  destroy, and "receive one snapshot" operations in libzfs_core.
 *  /sbin/zfs and /zbin/zpool will link with both libzfs and
 *  libzfs_core.  Other consumers should aim to use only libzfs_core,
 *  since that will be the supported, stable interface going forwards.
 */

#include <libzfs_core.h>
#include <ctype.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/nvpair.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/zfs_ioctl.h>

static int g_fd;
static pthread_mutex_t g_lock = PTHREAD_MUTEX_INITIALIZER;
static int g_refcount;

int
libzfs_core_init(void)
{
        (void) pthread_mutex_lock(&g_lock);
        if (g_refcount == 0) {
                g_fd = open("/dev/zfs", O_RDWR);
                if (g_fd < 0) {
                        (void) pthread_mutex_unlock(&g_lock);
                        return (errno);
                }
        }
        g_refcount++;
        (void) pthread_mutex_unlock(&g_lock);
        return (0);
}
______unchanged_portion_omitted_

/*
 * The simplest receive case: receive from the specified fd, creating the
 * specified snapshot.  Apply the specified properties a "received" properties
 * (which can be overridden by locally-set properties).  If the stream is a
 * clone, its origin snapshot must be specified by 'origin'.  The 'force'
 * flag will cause the target filesystem to be rolled back or destroyed if
 * necessary to receive.
 *
 * Return 0 on success or an errno on failure.
 *
 * Note: this interface does not work on dedup'd streams
 * (those with DMU_BACKUP_FEATURE_DEDUP).
 */
int
lzc_receive(const char *snapname, nvlist_t *props, const char *origin,
    boolean_t force, int fd)
```

```
{
        /*
         * The receive ioctl is still legacy, so we need to construct our own
         * zfs_cmd_t rather than using zfsc_ioctl().
         */
        zfs_cmd_t zc = { 0 };
        char *atp;
        char *packed = NULL;
        size_t size;
        dmu_replay_record_t drr;
        int error;

        ASSERT3S(g_refcount, >, 0);

        /* zc_name is name of containing filesystem */
        (void) strlcpy(zc.zc_name, snapname, sizeof (zc.zc_name));
        atp = strchr(zc.zc_name, '@');
        if (atp == NULL)
                return (EINVAL);
        *atp = '\0';

        /* if the fs does not exist, try its parent. */
        if (!lzc_exists(zc.zc_name)) {
                char *slashp = strrchr(zc.zc_name, '/');
                if (slashp == NULL)
                        return (ENOENT);
                *slashp = '\0';

        }

        /* zc_value is full name of the snapshot to create */
        (void) strlcpy(zc.zc_value, snapname, sizeof (zc.zc_value));

        if (props != NULL) {
                /* zc_nvlist_src is props to set */
                packed = fnvlist_pack(props, &size);
                zc.zc_nvlist_src = (uint64_t)(uintptr_t)packed;
                zc.zc_nvlist_src_size = size;
        }

        /* zc_string is name of clone origin (if DRR_FLAG_CLONE) */
        if (origin != NULL)
                (void) strlcpy(zc.zc_string, origin, sizeof (zc.zc_string));

        /* zc_begin_record is non-byteswapped BEGIN record */
        error = recv_read(fd, &drr, sizeof (drr));
        if (error != 0)
                goto out;
        zc.zc_begin_record = drr.drr_u.drr_begin;

        /* zc_cookie is fd to read from */
        zc.zc_cookie = fd;

        /* zc guid is force flag */
        zc.zc_guid = force;

        /* zc_cleanup_fd is unused */
        zc.zc_cleanup_fd = -1;

        error = ioctl(g_fd, ZFS_IOC_RECV, &zc);
        if (error != 0)
                error = errno;

out:
        if (packed != NULL)
                fnvlist_pack_free(packed, size);
```

```
        free((void*)(uintptr_t)zc.zc_nvlist_dst);
        return (error);
}

/*
 * Roll back this filesystem or volume to its most recent snapshot.
 * If snapnamebuf is not NULL, it will be filled in with the name
 * of the most recent snapshot.
 *
 * Return 0 on success or an errno on failure.
 */
int
lzc_rollback(const char *fsname, char *snapnamebuf, int snapnamelen)
{
        nvlist_t *args;
        nvlist_t *result;
        int err;

        args = fnvlist_alloc();
        err = lzc_ioctl(ZFS_IOC_ROLLBACK, fsname, args, &result);
        nvlist_free(args);
        if (err == 0 && snapnamebuf != NULL) {
                const char *snapname = fnvlist_lookup_string(result, "target");
                (void) strlcpy(snapnamebuf, snapname, snapnamelen);
        }
        return (err);
}
```
______*unchanged_portion_omitted_*

```
**********************************************************
    2130 Mon Aug 12 01:16:05 2013
new/usr/src/lib/libzfs_core/common/libzfs_core.h
3996 want a libzfs_core API to rollback to latest snapshot
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _LIBZFS_CORE_H
#define _LIBZFS_CORE_H

#include <libnvpair.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/fs/zfs.h>

#ifdef  __cplusplus
extern "C" {
#endif

int libzfs_core_init(void);
void libzfs_core_fini(void);

int lzc_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t **errlist);
int lzc_create(const char *fsname, dmu_objset_type_t type, nvlist_t *props);
int lzc_clone(const char *fsname, const char *origin, nvlist_t *props);
int lzc_destroy_snaps(nvlist_t *snaps, boolean_t defer, nvlist_t **errlist);

int lzc_snaprange_space(const char *firstsnap, const char *lastsnap,
    uint64_t *usedp);

int lzc_hold(nvlist_t *holds, int cleanup_fd, nvlist_t **errlist);
int lzc_release(nvlist_t *holds, nvlist_t **errlist);
int lzc_get_holds(const char *snapname, nvlist_t **holdsp);

int lzc_send(const char *snapname, const char *fromsnap, int fd);
int lzc_receive(const char *snapname, nvlist_t *props, const char *origin,
    boolean_t force, int fd);
int lzc_send_space(const char *snapname, const char *fromsnap,
    uint64_t *result);
```

```
boolean_t lzc_exists(const char *dataset);

int lzc_rollback(const char *fsname, char *snapnamebuf, int snapnamelen);

#ifdef  __cplusplus
}
______unchanged_portion_omitted_
```

```
*********************************************************
    1593 Mon Aug 12 01:16:06 2013
new/usr/src/lib/libzfs_core/common/mapfile-vers
3996 want a libzfs_core API to rollback to latest snapshot
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
*********************************************************
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
# Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright (c) 2013 by Delphix. All rights reserved.
# Copyright (c) 2012 by Delphix. All rights reserved.
#
# MAPFILE HEADER START
#
# WARNING:  STOP NOW.  DO NOT MODIFY THIS FILE.
# Object versioning must comply with the rules detailed in
#
#       usr/src/lib/README.mapfiles
#
# You should not be making modifications here until you've read the most current
# copy of that file. If you need help, contact a gatekeeper for guidance.
#
# MAPFILE HEADER END
#

$mapfile_version 2

SYMBOL_VERSION ILLUMOS_0.1 {
    global:

        libzfs_core_fini;
        libzfs_core_init;
        lzc_clone;
        lzc_create;
        lzc_destroy_snaps;
        lzc_exists;
        lzc_get_holds;
        lzc_hold;
        lzc_receive;
        lzc_release;
        lzc_rollback;
        lzc_send;
        lzc_send_space;
        lzc_snaprange_space;
        lzc_snapshot;
```

```
    local:
        *;
};
______unchanged_portion_omitted_
```

```
**********************************************************
   83061 Mon Aug 12 01:16:07 2013
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
3996 want a libzfs_core API to rollback to latest snapshot
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

typedef struct dsl_dataset_rollback_arg {
        const char *ddra_fsname;
        void *ddra_owner;
        nvlist_t *ddra_result;
} dsl_dataset_rollback_arg_t;
______unchanged_portion_omitted_

static void
dsl_dataset_rollback_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_rollback_arg_t *ddra = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds, *clone;
        uint64_t cloneobj;
        char namebuf[ZFS_MAXNAMELEN];

        VERIFY0(dsl_dataset_hold(dp, ddra->ddra_fsname, FTAG, &ds));

        dsl_dataset_name(ds->ds_prev, namebuf);
        fnvlist_add_string(ddra->ddra_result, "target", namebuf);

        cloneobj = dsl_dataset_create_sync(ds->ds_dir, "%rollback",
            ds->ds_prev, DS_CREATE_FLAG_NODIRTY, kcred, tx);

        VERIFY0(dsl_dataset_hold_obj(dp, cloneobj, FTAG, &clone));

        dsl_dataset_clone_swap_sync_impl(clone, ds, tx);
        dsl_dataset_zero_zil(ds, tx);

        dsl_destroy_head_sync_impl(clone, tx);

        dsl_dataset_rele(clone, FTAG);
        dsl_dataset_rele(ds, FTAG);
}

/*
 * Rolls back the given filesystem or volume to the most recent snapshot.
 * The name of the most recent snapshot will be returned under key "target"
 * in the result nvlist.
 *
 * If owner != NULL:
 *
 * - The existing dataset MUST be owned by the specified owner at entry
 * - Upon return, dataset will still be held by the same owner, whether we
 *   succeed or not.
 *
 * This mode is required any time the existing filesystem is mounted.  See
 * notes above zfs_suspend_fs() for further details.
 */
int
dsl_dataset_rollback(const char *fsname, void *owner, nvlist_t *result)
dsl_dataset_rollback(const char *fsname, void *owner)
{
        dsl_dataset_rollback_arg_t ddra;

        ddra.ddra_fsname = fsname;
```

```
        ddra.ddra_owner = owner;
        ddra.ddra_result = result;

        return (dsl_sync_task(fsname, dsl_dataset_rollback_check,
            dsl_dataset_rollback_sync, &ddra, 1));
            dsl_dataset_rollback_sync, (void *)&ddra, 1));
}
______unchanged_portion_omitted_
```

****
10302 Mon Aug 12 01:16:08 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
3996 want a libzfs_core API to rollback to latest snapshot
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
****
______unchanged_portion_omitted_

```
/*
 * The max length of a temporary tag prefix is the number of hex digits
 * required to express UINT64_MAX plus one for the hyphen.
 */
#define MAX_TAG_PREFIX_LEN      17

#define dsl_dataset_is_snapshot(ds) \
        ((ds)->ds_phys->ds_num_children != 0)

#define DS_UNIQUE_IS_ACCURATE(ds)       \
        (((ds)->ds_phys->ds_flags & DS_FLAG_UNIQUE_ACCURATE) != 0)

int dsl_dataset_hold(struct dsl_pool *dp, const char *name, void *tag,
    dsl_dataset_t **dsp);
int dsl_dataset_hold_obj(struct dsl_pool *dp, uint64_t dsobj, void *tag,
    dsl_dataset_t **);
void dsl_dataset_rele(dsl_dataset_t *ds, void *tag);
int dsl_dataset_own(struct dsl_pool *dp, const char *name,
    void *tag, dsl_dataset_t **dsp);
int dsl_dataset_own_obj(struct dsl_pool *dp, uint64_t dsobj,
    void *tag, dsl_dataset_t **dsp);
void dsl_dataset_disown(dsl_dataset_t *ds, void *tag);
void dsl_dataset_name(dsl_dataset_t *ds, char *name);
boolean_t dsl_dataset_tryown(dsl_dataset_t *ds, void *tag);
uint64_t dsl_dataset_create_sync(dsl_dir_t *pds, const char *lastname,
    dsl_dataset_t *origin, uint64_t flags, cred_t *, dmu_tx_t *);
uint64_t dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
    uint64_t flags, dmu_tx_t *tx);
int dsl_dataset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors);
int dsl_dataset_promote(const char *name, char *conflsnap);
int dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
    boolean_t force);
int dsl_dataset_rename_snapshot(const char *fsname,
    const char *oldsnapname, const char *newsnapname, boolean_t recursive);
int dsl_dataset_snapshot_tmp(const char *fsname, const char *snapname,
    minor_t cleanup_minor, const char *htag);

blkptr_t *dsl_dataset_get_blkptr(dsl_dataset_t *ds);
void dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx);

spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);

boolean_t dsl_dataset_modified_since_snap(dsl_dataset_t *ds,
    dsl_dataset_t *snap);

void dsl_dataset_sync(dsl_dataset_t *os, zio_t *zio, dmu_tx_t *tx);

void dsl_dataset_block_born(dsl_dataset_t *ds, const blkptr_t *bp,
    dmu_tx_t *tx);
int dsl_dataset_block_kill(dsl_dataset_t *ds, const blkptr_t *bp,
    dmu_tx_t *tx, boolean_t async);
boolean_t dsl_dataset_block_freeable(dsl_dataset_t *ds, const blkptr_t *bp,
    uint64_t blk_birth);
uint64_t dsl_dataset_prev_snap_txg(dsl_dataset_t *ds);

void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
```

```
void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);
void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
void dsl_dataset_space(dsl_dataset_t *ds,
    uint64_t *refdbytesp, uint64_t *availbytesp,
    uint64_t *usedobjsp, uint64_t *availobjsp);
uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
    uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
    uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
boolean_t dsl_dataset_is_dirty(dsl_dataset_t *ds);

int dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf);

int dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
    uint64_t asize, uint64_t inflight, uint64_t *used,
    uint64_t *ref_rsrv);
int dsl_dataset_set_refquota(const char *dsname, zprop_source_t source,
    uint64_t quota);
int dsl_dataset_set_refreservation(const char *dsname, zprop_source_t source,
    uint64_t reservation);

boolean_t dsl_dataset_is_before(dsl_dataset_t *later, dsl_dataset_t *earlier);
void dsl_dataset_long_hold(dsl_dataset_t *ds, void *tag);
void dsl_dataset_long_rele(dsl_dataset_t *ds, void *tag);
boolean_t dsl_dataset_long_held(dsl_dataset_t *ds);

int dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
    dsl_dataset_t *origin_head, boolean_t force, void *owner, dmu_tx_t *tx);
void dsl_dataset_clone_swap_sync_impl(dsl_dataset_t *clone,
    dsl_dataset_t *origin_head, dmu_tx_t *tx);
int dsl_dataset_snapshot_check_impl(dsl_dataset_t *ds, const char *snapname,
    dmu_tx_t *tx, boolean_t recv);
void dsl_dataset_snapshot_sync_impl(dsl_dataset_t *ds, const char *snapname,
    dmu_tx_t *tx);

void dsl_dataset_remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj,
    dmu_tx_t *tx);
void dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds);
int dsl_dataset_get_snapname(dsl_dataset_t *ds);
int dsl_dataset_snap_lookup(dsl_dataset_t *ds, const char *name,
    uint64_t *value);
int dsl_dataset_snap_remove(dsl_dataset_t *ds, const char *name, dmu_tx_t *tx);
void dsl_dataset_set_refreservation_sync_impl(dsl_dataset_t *ds,
    zprop_source_t source, uint64_t value, dmu_tx_t *tx);
int dsl_dataset_rollback(const char *fsname, void *owner, nvlist_t *result);
int dsl_dataset_rollback(const char *fsname, void *owner);

#ifdef ZFS_DEBUG
#define dprintf_ds(ds, fmt, ...) do { \
        if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
        char *__ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
        dsl_dataset_name(ds, __ds_name); \
        dprintf("ds=%s " fmt, __ds_name, __VA_ARGS__); \
        kmem_free(__ds_name, MAXNAMELEN); \
        } \
_NOTE(CONSTCOND) } while (0)
#else
#define dprintf_ds(dd, fmt, ...)
#endif

#ifdef  __cplusplus
}
#endif

#endif /* _SYS_DSL_DATASET_H */
```

```
**********************************************************
  144478 Mon Aug 12 01:16:10 2013
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
3996 want a libzfs_core API to rollback to latest snapshot
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

/*
 * fsname is name of dataset to rollback (to most recent snapshot)
 * inputs:
 * zc_name      name of dataset to rollback (to most recent snapshot)
 *
 * innvl is not used.
 *
 * outnvl: "target" -> name of most recent snapshot
 * }
 * outputs:     none
 */
/* ARGSUSED */
static int
zfs_ioc_rollback(const char *fsname, nvlist_t *args, nvlist_t *outnvl)
zfs_ioc_rollback(zfs_cmd_t *zc)
{
        zfsvfs_t *zfsvfs;
        int error;

        if (getzfsvfs(fsname, &zfsvfs) == 0) {
        if (getzfsvfs(zc->zc_name, &zfsvfs) == 0) {
                error = zfs_suspend_fs(zfsvfs);
                if (error == 0) {
                        int resume_err;

                        error = dsl_dataset_rollback(fsname, zfsvfs, outnvl);
                        resume_err = zfs_resume_fs(zfsvfs, fsname);
                        error = dsl_dataset_rollback(zc->zc_name, zfsvfs);
                        resume_err = zfs_resume_fs(zfsvfs, zc->zc_name);
                        error = error ? error : resume_err;
                }
                VFS_RELE(zfsvfs->z_vfs);
        } else {
                error = dsl_dataset_rollback(fsname, NULL, outnvl);
                error = dsl_dataset_rollback(zc->zc_name, NULL);
        }
        return (error);
}
______unchanged_portion_omitted_

static void
zfs_ioctl_init(void)
{
        zfs_ioctl_register("snapshot", ZFS_IOC_SNAPSHOT,
            zfs_ioc_snapshot, zfs_secpolicy_snapshot, POOL_NAME,
            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

        zfs_ioctl_register("log_history", ZFS_IOC_LOG_HISTORY,
            zfs_ioc_log_history, zfs_secpolicy_log_history, NO_NAME,
            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_FALSE, B_FALSE);

        zfs_ioctl_register("space_snaps", ZFS_IOC_SPACE_SNAPS,
            zfs_ioc_space_snaps, zfs_secpolicy_read, DATASET_NAME,
            POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

        zfs_ioctl_register("send", ZFS_IOC_SEND_NEW,
```

```
            zfs_ioc_send_new, zfs_secpolicy_send_new, DATASET_NAME,
            POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

        zfs_ioctl_register("send_space", ZFS_IOC_SEND_SPACE,
            zfs_ioc_send_space, zfs_secpolicy_read, DATASET_NAME,
            POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

        zfs_ioctl_register("create", ZFS_IOC_CREATE,
            zfs_ioc_create, zfs_secpolicy_create_clone, DATASET_NAME,
            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

        zfs_ioctl_register("clone", ZFS_IOC_CLONE,
            zfs_ioc_clone, zfs_secpolicy_create_clone, DATASET_NAME,
            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

        zfs_ioctl_register("destroy_snaps", ZFS_IOC_DESTROY_SNAPS,
            zfs_ioc_destroy_snaps, zfs_secpolicy_destroy_snaps, POOL_NAME,
            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

        zfs_ioctl_register("hold", ZFS_IOC_HOLD,
            zfs_ioc_hold, zfs_secpolicy_hold, POOL_NAME,
            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);
        zfs_ioctl_register("release", ZFS_IOC_RELEASE,
            zfs_ioc_release, zfs_secpolicy_release, POOL_NAME,
            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

        zfs_ioctl_register("get_holds", ZFS_IOC_GET_HOLDS,
            zfs_ioc_get_holds, zfs_secpolicy_read, DATASET_NAME,
            POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

        zfs_ioctl_register("rollback", ZFS_IOC_ROLLBACK,
            zfs_ioc_rollback, zfs_secpolicy_rollback, DATASET_NAME,
            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_FALSE, B_TRUE);

        /* IOCTLS that use the legacy function signature */

        zfs_ioctl_register_legacy(ZFS_IOC_POOL_FREEZE, zfs_ioc_pool_freeze,
            zfs_secpolicy_config, NO_NAME, B_FALSE, POOL_CHECK_READONLY);

        zfs_ioctl_register_pool(ZFS_IOC_POOL_CREATE, zfs_ioc_pool_create,
            zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);
        zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SCAN,
            zfs_ioc_pool_scan);
        zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_UPGRADE,
            zfs_ioc_pool_upgrade);
        zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ADD,
            zfs_ioc_vdev_add);
        zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_REMOVE,
            zfs_ioc_vdev_remove);
        zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SET_STATE,
            zfs_ioc_vdev_set_state);
        zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ATTACH,
            zfs_ioc_vdev_attach);
        zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_DETACH,
            zfs_ioc_vdev_detach);
        zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETPATH,
            zfs_ioc_vdev_setpath);
        zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETFRU,
            zfs_ioc_vdev_setfru);
        zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SET_PROPS,
            zfs_ioc_pool_set_props);
        zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SPLIT,
            zfs_ioc_vdev_split);
        zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_REGUID,
            zfs_ioc_pool_reguid);
        zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_CONFIGS,
            zfs_ioc_pool_configs, zfs_secpolicy_none);
        zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_TRYIMPORT,
            zfs_ioc_pool_tryimport, zfs_secpolicy_config);
        zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_FAULT,
            zfs_ioc_inject_fault, zfs_secpolicy_inject);
        zfs_ioctl_register_pool_meta(ZFS_IOC_CLEAR_FAULT,
            zfs_ioc_clear_fault, zfs_secpolicy_inject);
        zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_LIST_NEXT,
            zfs_ioc_inject_list_next, zfs_secpolicy_inject);

        /*
         * pool destroy, and export don't log the history as part of
         * zfsdev_ioctl, but rather zfs_ioc_pool_export
         * does the logging of those commands.
         */
        zfs_ioctl_register_pool(ZFS_IOC_POOL_DESTROY, zfs_ioc_pool_destroy,
            zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);
        zfs_ioctl_register_pool(ZFS_IOC_POOL_EXPORT, zfs_ioc_pool_export,
            zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);

        zfs_ioctl_register_pool(ZFS_IOC_POOL_STATS, zfs_ioc_pool_stats,
            zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);
        zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_PROPS, zfs_ioc_pool_get_props,
            zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);

        zfs_ioctl_register_pool(ZFS_IOC_ERROR_LOG, zfs_ioc_error_log,
            zfs_secpolicy_inject, B_FALSE, POOL_CHECK_SUSPENDED);
        zfs_ioctl_register_pool(ZFS_IOC_DSOBJ_TO_DSNAME,
            zfs_ioc_dsobj_to_dsname,
            zfs_secpolicy_diff, B_FALSE, POOL_CHECK_SUSPENDED);
        zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_HISTORY,
            zfs_ioc_pool_get_history,
            zfs_secpolicy_config, B_FALSE, POOL_CHECK_SUSPENDED);

        zfs_ioctl_register_pool(ZFS_IOC_POOL_IMPORT, zfs_ioc_pool_import,
            zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);

        zfs_ioctl_register_pool(ZFS_IOC_CLEAR, zfs_ioc_clear,
            zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);
        zfs_ioctl_register_pool(ZFS_IOC_POOL_REOPEN, zfs_ioc_pool_reopen,
            zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);

        zfs_ioctl_register_dataset_read(ZFS_IOC_SPACE_WRITTEN,
            zfs_ioc_space_written);
        zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_RECVD_PROPS,
            zfs_ioc_objset_recvd_props);
        zfs_ioctl_register_dataset_read(ZFS_IOC_NEXT_OBJ,
            zfs_ioc_next_obj);
        zfs_ioctl_register_dataset_read(ZFS_IOC_GET_FSACL,
            zfs_ioc_get_fsacl);
        zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_STATS,
            zfs_ioc_objset_stats);
        zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_ZPLPROPS,
            zfs_ioc_objset_zplprops);
        zfs_ioctl_register_dataset_read(ZFS_IOC_DATASET_LIST_NEXT,
            zfs_ioc_dataset_list_next);
        zfs_ioctl_register_dataset_read(ZFS_IOC_SNAPSHOT_LIST_NEXT,
            zfs_ioc_snapshot_list_next);
        zfs_ioctl_register_dataset_read(ZFS_IOC_SEND_PROGRESS,
            zfs_ioc_send_progress);

        zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_DIFF,
            zfs_ioc_diff, zfs_secpolicy_diff);
        zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_STATS,
            zfs_ioc_obj_to_stats, zfs_secpolicy_diff);
```

```
        zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_PATH,
            zfs_ioc_obj_to_path, zfs_secpolicy_diff);
        zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_ONE,
            zfs_ioc_userspace_one, zfs_secpolicy_userspace_one);
        zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_MANY,
            zfs_ioc_userspace_many, zfs_secpolicy_userspace_many);
        zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_SEND,
            zfs_ioc_send, zfs_secpolicy_send);

        zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_PROP, zfs_ioc_set_prop,
            zfs_secpolicy_none);
        zfs_ioctl_register_dataset_modify(ZFS_IOC_DESTROY, zfs_ioc_destroy,
            zfs_secpolicy_destroy);
        zfs_ioctl_register_dataset_modify(ZFS_IOC_ROLLBACK, zfs_ioc_rollback,
            zfs_secpolicy_rollback);
        zfs_ioctl_register_dataset_modify(ZFS_IOC_RENAME, zfs_ioc_rename,
            zfs_secpolicy_rename);
        zfs_ioctl_register_dataset_modify(ZFS_IOC_RECV, zfs_ioc_recv,
            zfs_secpolicy_recv);
        zfs_ioctl_register_dataset_modify(ZFS_IOC_PROMOTE, zfs_ioc_promote,
            zfs_secpolicy_promote);
        zfs_ioctl_register_dataset_modify(ZFS_IOC_INHERIT_PROP,
            zfs_ioc_inherit_prop, zfs_secpolicy_inherit_prop);
        zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_FSACL, zfs_ioc_set_fsacl,
            zfs_secpolicy_set_fsacl);

        zfs_ioctl_register_dataset_nolog(ZFS_IOC_SHARE, zfs_ioc_share,
            zfs_secpolicy_share, POOL_CHECK_NONE);
        zfs_ioctl_register_dataset_nolog(ZFS_IOC_SMB_ACL, zfs_ioc_smb_acl,
            zfs_secpolicy_smb_acl, POOL_CHECK_NONE);
        zfs_ioctl_register_dataset_nolog(ZFS_IOC_USERSPACE_UPGRADE,
            zfs_ioc_userspace_upgrade, zfs_secpolicy_userspace_upgrade,
            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
        zfs_ioctl_register_dataset_nolog(ZFS_IOC_TMP_SNAPSHOT,
            zfs_ioc_tmp_snapshot, zfs_secpolicy_tmp_snapshot,
            POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
}
```

______***unchanged_portion_omitted_***