```
**********************************************************
   27038 Thu Apr 25 12:27:55 2013
new/usr/src/lib/libzfs/common/libzfs.h
Optimize creation and removal of temporary "user holds" placed on
snapshots by a zfs send, by ensuring all the required holds and
releases are done in a single dsl_sync_task.
Creation now collates the required holds during a dry run and
then uses a single lzc_hold call via zfs_hold_apply instead of
processing each snapshot in turn.
Defered (on exit) cleanup by the kernel is also now done in
dsl_sync_task by reusing dsl_dataset_user_release.
On a test with 11 volumes in a tree each with 8 snapshots on a
single HDD zpool this reduces the time required to perform a full
send from 20 seconds to under 0.8 seconds.
For reference eliminating the hold entirely reduces this 0.15
seconds.
While I'm here:-
* Remove some unused structures
* Fix nvlist_t leak in zfs_release_one
**********************************************************
______unchanged_portion_omitted_

typedef boolean_t (snapfilter_cb_t)(zfs_handle_t *, void *);

extern int zfs_send(zfs_handle_t *, const char *, const char *,
    sendflags_t *, int, snapfilter_cb_t, void *, nvlist_t **);

extern int zfs_promote(zfs_handle_t *);
extern int zfs_hold(zfs_handle_t *, const char *, const char *,
    boolean_t, boolean_t, int);
extern int zfs_hold_add(zfs_handle_t *, const char *, const char *,
    boolean_t, nvlist_t *);
extern int zfs_hold_apply(zfs_handle_t *, boolean_t, int, nvlist_t *);
#endif /* ! codereview */
extern int zfs_release(zfs_handle_t *, const char *, const char *, boolean_t);
extern int zfs_get_holds(zfs_handle_t *, nvlist_t **);
extern uint64_t zvol_volsize_to_reservation(uint64_t, nvlist_t *);

typedef int (*zfs_userspace_cb_t)(void *arg, const char *domain,
    uid_t rid, uint64_t space);

extern int zfs_userspace(zfs_handle_t *, zfs_userquota_prop_t,
    zfs_userspace_cb_t, void *);

extern int zfs_get_fsacl(zfs_handle_t *, nvlist_t **);
extern int zfs_set_fsacl(zfs_handle_t *, boolean_t, nvlist_t *);

typedef struct recvflags {
        /* print informational messages (ie, -v was specified) */
        boolean_t verbose;

        /* the destination is a prefix, not the exact fs (ie, -d) */
        boolean_t isprefix;

        /*
         * Only the tail of the sent snapshot path is appended to the
         * destination to determine the received snapshot name (ie, -e).
         */
        boolean_t istail;

        /* do not actually do the recv, just check if it would work (ie, -n) */
        boolean_t dryrun;

        /* rollback/destroy filesystems as necessary (eg, -F) */
        boolean_t force;
```

```
        /* set "canmount=off" on all modified filesystems */
        boolean_t canmountoff;

        /* byteswap flag is used internally; callers need not specify */
        boolean_t byteswap;

        /* do not mount file systems as they are extracted (private) */
        boolean_t nomount;
} recvflags_t;

extern int zfs_receive(libzfs_handle_t *, const char *, recvflags_t *,
    int, avl_tree_t *);

typedef enum diff_flags {
        ZFS_DIFF_PARSEABLE = 0x1,
        ZFS_DIFF_TIMESTAMP = 0x2,
        ZFS_DIFF_CLASSIFY = 0x4
} diff_flags_t;

extern int zfs_show_diffs(zfs_handle_t *, int, const char *, const char *,
    int);

/*
 * Miscellaneous functions.
 */
extern const char *zfs_type_to_name(zfs_type_t);
extern void zfs_refresh_properties(zfs_handle_t *);
extern int zfs_name_valid(const char *, zfs_type_t);
extern zfs_handle_t *zfs_path_to_zhandle(libzfs_handle_t *, char *, zfs_type_t);
extern boolean_t zfs_dataset_exists(libzfs_handle_t *, const char *,
    zfs_type_t);
extern int zfs_spa_version(zfs_handle_t *, int *);

/*
 * Mount support functions.
 */
extern boolean_t is_mounted(libzfs_handle_t *, const char *special, char **);
extern boolean_t zfs_is_mounted(zfs_handle_t *, char **);
extern int zfs_mount(zfs_handle_t *, const char *, int);
extern int zfs_unmount(zfs_handle_t *, const char *, int);
extern int zfs_unmountall(zfs_handle_t *, int);

/*
 * Share support functions.
 */
extern boolean_t zfs_is_shared(zfs_handle_t *);
extern int zfs_share(zfs_handle_t *);
extern int zfs_unshare(zfs_handle_t *);

/*
 * Protocol-specific share support functions.
 */
extern boolean_t zfs_is_shared_nfs(zfs_handle_t *, char **);
extern boolean_t zfs_is_shared_smb(zfs_handle_t *, char **);
extern int zfs_share_nfs(zfs_handle_t *);
extern int zfs_share_smb(zfs_handle_t *);
extern int zfs_shareall(zfs_handle_t *);
extern int zfs_unshare_nfs(zfs_handle_t *, const char *);
extern int zfs_unshare_smb(zfs_handle_t *, const char *);
extern int zfs_unshareall_nfs(zfs_handle_t *);
extern int zfs_unshareall_smb(zfs_handle_t *);
extern int zfs_unshareall_bypath(zfs_handle_t *, const char *);
extern int zfs_unshareall(zfs_handle_t *);
extern int zfs_deleg_share_nfs(libzfs_handle_t *, char *, char *, char *,
    void *, void *, int, zfs_share_op_t);
```

```
/*
 * When dealing with nvlists, verify() is extremely useful
 */
#ifdef NDEBUG
#define verify(EX)      ((void)(EX))
#else
#define verify(EX)      assert(EX)
#endif

/*
 * Utility function to convert a number to a human-readable form.
 */
extern void zfs_nicenum(uint64_t, char *, size_t);
extern int zfs_nicestrtonum(libzfs_handle_t *, const char *, uint64_t *);

/*
 * Given a device or file, determine if it is part of a pool.
 */
extern int zpool_in_use(libzfs_handle_t *, int, pool_state_t *, char **,
    boolean_t *);

/*
 * Label manipulation.
 */
extern int zpool_read_label(int, nvlist_t **);
extern int zpool_clear_label(int);

/* is this zvol valid for use as a dump device? */
extern int zvol_check_dump_config(char *);

/*
 * Management interfaces for SMB ACL files
 */

int zfs_smb_acl_add(libzfs_handle_t *, char *, char *, char *);
int zfs_smb_acl_remove(libzfs_handle_t *, char *, char *, char *);
int zfs_smb_acl_purge(libzfs_handle_t *, char *, char *);
int zfs_smb_acl_rename(libzfs_handle_t *, char *, char *, char *, char *);

/*
 * Enable and disable datasets within a pool by mounting/unmounting and
 * sharing/unsharing them.
 */
extern int zpool_enable_datasets(zpool_handle_t *, const char *, int);
extern int zpool_disable_datasets(zpool_handle_t *, boolean_t);

/*
 * Mappings between vdev and FRU.
 */
extern void libzfs_fru_refresh(libzfs_handle_t *);
extern const char *libzfs_fru_lookup(libzfs_handle_t *, const char *);
extern const char *libzfs_fru_devpath(libzfs_handle_t *, const char *);
extern boolean_t libzfs_fru_compare(libzfs_handle_t *, const char *,
    const char *);
extern boolean_t libzfs_fru_notself(libzfs_handle_t *, const char *);
extern int zpool_fru_set(zpool_handle_t *, uint64_t, const char *);

#ifdef  __cplusplus
}
#endif

#endif  /* _LIBZFS_H */
```

```
**********************************************************
  111585 Thu Apr 25 12:27:55 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.c
Optimize creation and removal of temporary "user holds" placed on
snapshots by a zfs send, by ensuring all the required holds and
releases are done in a single dsl_sync_task.
Creation now collates the required holds during a dry run and
then uses a single lzc_hold call via zfs_hold_apply instead of
processing each snapshot in turn.
Defered (on exit) cleanup by the kernel is also now done in
dsl_sync_task by reusing dsl_dataset_user_release.
On a test with 11 volumes in a tree each with 8 snapshots on a
single HDD zpool this reduces the time required to perform a full
send from 20 seconds to under 0.8 seconds.
For reference eliminating the hold entirely reduces this 0.15
seconds.
While I'm here:-
* Remove some unused structures
* Fix nvlist_t leak in zfs_release_one
**********************************************************
______unchanged_portion_omitted_
```

```
int
zfs_hold_add(zfs_handle_t *zhp, const char *snapname, const char *tag,
    boolean_t enoent_ok, nvlist_t *holds)
{
        zfs_handle_t *szhp;
        char name[ZFS_MAXNAMELEN];
        char errbuf[1024];
        int ret;

        (void) snprintf(name, sizeof (name),
            "%s@%s", zhp->zfs_name, snapname);

        szhp = make_dataset_handle(zhp->zfs_hdl, name);
        if (szhp) {
                fnvlist_add_string(holds, name, tag);
                zfs_close(szhp);
                return (0);
        }

        ret = ENOENT;
        if (enoent_ok)
                return (ret);

        (void) snprintf(errbuf, sizeof (errbuf),
            dgettext(TEXT_DOMAIN, "cannot hold snapshot '%s@%s'"),
            zhp->zfs_name, snapname);
        (void) zfs_standard_error(zhp->zfs_hdl, ret, errbuf);

        return (ret);
}

int
#endif /* ! codereview */
zfs_hold(zfs_handle_t *zhp, const char *snapname, const char *tag,
    boolean_t recursive, boolean_t enoent_ok, int cleanup_fd)
{
        int ret;
        struct holdarg ha;
        nvlist_t *errors;
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char errbuf[1024];
        nvpair_t *elem;

        ha.nvl = fnvlist_alloc();
```

```
        ha.snapname = snapname;
        ha.tag = tag;
        ha.recursive = recursive;
        (void) zfs_hold_one(zfs_handle_dup(zhp), &ha);
        ret = zfs_hold_apply(zhp, enoent_ok, cleanup_fd, ha.nvl);
        ret = lzc_hold(ha.nvl, cleanup_fd, &errors);
        fnvlist_free(ha.nvl);

        return (ret);
}

int
zfs_hold_apply(zfs_handle_t *zhp, boolean_t enoent_ok, int cleanup_fd, nvlist_t
{
        int ret;
        nvlist_t *errors;
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char errbuf[1024];
        nvpair_t *elem;

        ret = lzc_hold(holds, cleanup_fd, &errors);

#endif /* ! codereview */
        if (ret == 0)
                return (0);

        if (nvlist_next_nvpair(errors, NULL) == NULL) {
                /* no hold-specific errors */
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot hold"));
                switch (ret) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                default:
                        (void) zfs_standard_error(hdl, ret, errbuf);
                }
        }

        for (elem = nvlist_next_nvpair(errors, NULL);
            elem != NULL;
            elem = nvlist_next_nvpair(errors, elem)) {
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN,
                    "cannot hold snapshot '%s'"), nvpair_name(elem));
                switch (fnvpair_value_int32(elem)) {
                case E2BIG:
                        /*
                         * Temporary tags wind up having the ds object id
                         * prepended. So even if we passed the length check
                         * above, it's still possible for the tag to wind
                         * up being slightly too long.
                         */
                        (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
                        break;
                case EINVAL:
                        (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case EEXIST:
                        (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
                        break;
                case ENOENT:
                        if (enoent_ok)
                                return (ENOENT);
                        /* FALLTHROUGH */
                default:
                        (void) zfs_standard_error(hdl,
                            fnvpair_value_int32(elem), errbuf);
                }
        }

        fnvlist_free(errors);
        return (ret);
}

struct releasearg {
        nvlist_t *nvl;
        const char *snapname;
        const char *tag;
        boolean_t recursive;
};

static int
zfs_release_one(zfs_handle_t *zhp, void *arg)
{
        struct holdarg *ha = arg;
        zfs_handle_t *szhp;
        char name[ZFS_MAXNAMELEN];
        int rv = 0;

        (void) snprintf(name, sizeof (name),
            "%s@%s", zhp->zfs_name, ha->snapname);

        szhp = make_dataset_handle(zhp->zfs_hdl, name);
        if (szhp) {
                nvlist_t *holds = fnvlist_alloc();
                fnvlist_add_boolean(holds, ha->tag);
                fnvlist_add_nvlist(ha->nvl, name, holds);
                fnvlist_free(holds);
#endif /* ! codereview */
                zfs_close(szhp);
        }

        if (ha->recursive)
                rv = zfs_iter_filesystems(zhp, zfs_release_one, ha);
        zfs_close(zhp);
        return (rv);
}

int
zfs_release(zfs_handle_t *zhp, const char *snapname, const char *tag,
    boolean_t recursive)
{
        int ret;
        struct holdarg ha;
        nvlist_t *errors;
        nvpair_t *elem;
        libzfs_handle_t *hdl = zhp->zfs_hdl;

        ha.nvl = fnvlist_alloc();
        ha.snapname = snapname;
        ha.tag = tag;
        ha.recursive = recursive;
        (void) zfs_release_one(zfs_handle_dup(zhp), &ha);
        ret = lzc_release(ha.nvl, &errors);
        fnvlist_free(ha.nvl);
```

```
        if (ret == 0)
                return (0);

        if (nvlist_next_nvpair(errors, NULL) == NULL) {
                /* no hold-specific errors */
                char errbuf[1024];

                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "cannot release"));
                switch (errno) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                default:
                        (void) zfs_standard_error_fmt(hdl, errno, errbuf);
                }
        }

        for (elem = nvlist_next_nvpair(errors, NULL);
            elem != NULL;
            elem = nvlist_next_nvpair(errors, elem)) {
                char errbuf[1024];

                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN,
                    "cannot release hold from snapshot '%s'"),
                    nvpair_name(elem));
                switch (fnvpair_value_int32(elem)) {
                case ESRCH:
                        (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
                        break;
                case EINVAL:
                        (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                default:
                        (void) zfs_standard_error_fmt(hdl,
                            fnvpair_value_int32(elem), errbuf);
                }
        }

        fnvlist_free(errors);
        return (ret);
}

int
zfs_get_fsacl(zfs_handle_t *zhp, nvlist_t **nvl)
{
        zfs_cmd_t zc = { 0 };
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        int nvsz = 2048;
        void *nvbuf;
        int err = 0;
        char errbuf[1024];

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

tryagain:

        nvbuf = malloc(nvsz);
        if (nvbuf == NULL) {
                err = (zfs_error(hdl, EZFS_NOMEM, strerror(errno)));
                goto out;
        }
        zc.zc_nvlist_dst_size = nvsz;
        zc.zc_nvlist_dst = (uintptr_t)nvbuf;

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

        if (ioctl(hdl->libzfs_fd, ZFS_IOC_GET_FSACL, &zc) != 0) {
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot get permissions on ’%s’"),
                    zc.zc_name);
                switch (errno) {
                case ENOMEM:
                        free(nvbuf);
                        nvsz = zc.zc_nvlist_dst_size;
                        goto tryagain;

                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case ENOENT:
                        err = zfs_error(hdl, EZFS_NOENT, errbuf);
                        break;
                default:
                        err = zfs_standard_error_fmt(hdl, errno, errbuf);
                        break;
                }
        } else {
                /* success */
                int rc = nvlist_unpack(nvbuf, zc.zc_nvlist_dst_size, nvl, 0);
                if (rc) {
                        (void) snprintf(errbuf, sizeof (errbuf), dgettext(
                            TEXT_DOMAIN, "cannot get permissions on ’%s’"),
                            zc.zc_name);
                        err = zfs_standard_error_fmt(hdl, rc, errbuf);
                }
        }

        free(nvbuf);
out:
        return (err);
}

int
zfs_set_fsacl(zfs_handle_t *zhp, boolean_t un, nvlist_t *nvl)
{
        zfs_cmd_t zc = { 0 };
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char *nvbuf;
        char errbuf[1024];
        size_t nvsz;
        int err;

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

        err = nvlist_size(nvl, &nvsz, NV_ENCODE_NATIVE);
        assert(err == 0);

        nvbuf = malloc(nvsz);

        err = nvlist_pack(nvl, &nvbuf, &nvsz, NV_ENCODE_NATIVE, 0);
```

```
        assert(err == 0);

        zc.zc_nvlist_src_size = nvsz;
        zc.zc_nvlist_src = (uintptr_t)nvbuf;
        zc.zc_perm_action = un;

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

        if (zfs_ioctl(hdl, ZFS_IOC_SET_FSACL, &zc) != 0) {
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot set permissions on ’%s’"),
                    zc.zc_name);
                switch (errno) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case ENOENT:
                        err = zfs_error(hdl, EZFS_NOENT, errbuf);
                        break;
                default:
                        err = zfs_standard_error_fmt(hdl, errno, errbuf);
                        break;
                }
        }

        free(nvbuf);

        return (err);
}

int
zfs_get_holds(zfs_handle_t *zhp, nvlist_t **nvl)
{
        int err;
        char errbuf[1024];

        err = lzc_get_holds(zhp->zfs_name, nvl);

        if (err != 0) {
                libzfs_handle_t *hdl = zhp->zfs_hdl;

                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot get holds for ’%s’"),
                    zhp->zfs_name);
                switch (err) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case ENOENT:
                        err = zfs_error(hdl, EZFS_NOENT, errbuf);
                        break;
                default:
                        err = zfs_standard_error_fmt(hdl, errno, errbuf);
                        break;
                }
        }
        return (err);
}

uint64_t
zvol_volsize_to_reservation(uint64_t volsize, nvlist_t *props)
{
        uint64_t numdb;
        uint64_t nblocks, volblocksize;
        int ncopies;
        char *strval;

        if (nvlist_lookup_string(props,
            zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)
                ncopies = atoi(strval);
        else
                ncopies = 1;
        if (nvlist_lookup_uint64(props,
            zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
            &volblocksize) != 0)
                volblocksize = ZVOL_DEFAULT_BLOCKSIZE;
        nblocks = volsize/volblocksize;
        /* start with metadnode L0-L6 */
        numdb = 7;
        /* calculate number of indirects */
        while (nblocks > 1) {
                nblocks += DNODES_PER_LEVEL - 1;
                nblocks /= DNODES_PER_LEVEL;
                numdb += nblocks;
        }
        numdb *= MIN(SPA_DVAS_PER_BP, ncopies + 1);
        volsize *= ncopies;
        /*
         * this is exactly DN_MAX_INDBLKSHIFT when metadata isn’t
         * compressed, but in practice they compress down to about
         * 1100 bytes
         */
        numdb *= 1ULL << DN_MAX_INDBLKSHIFT;
        volsize += numdb;
        return (volsize);
}
```

```
**********************************************************
   85020 Thu Apr 25 12:27:56 2013
new/usr/src/lib/libzfs/common/libzfs_sendrecv.c
Optimize creation and removal of temporary "user holds" placed on
snapshots by a zfs send, by ensuring all the required holds and
releases are done in a single dsl_sync_task.
Creation now collates the required holds during a dry run and
then uses a single lzc_hold call via zfs_hold_apply instead of
processing each snapshot in turn.
Defered (on exit) cleanup by the kernel is also now done in
dsl_sync_task by reusing dsl_dataset_user_release.
On a test with 11 volumes in a tree each with 8 snapshots on a
single HDD zpool this reduces the time required to perform a full
send from 20 seconds to under 0.8 seconds.
For reference eliminating the hold entirely reduces this 0.15
seconds.
While I’m here:-
* Remove some unused structures
* Fix nvlist_t leak in zfs_release_one
**********************************************************
______unchanged_portion_omitted_
```

```
/*
 * Routines specific to "zfs send"
 */
typedef struct send_dump_data {
        /* these are all just the short snapname (the part after the @) */
        const char *fromsnap;
        const char *tosnap;
        char prevsnap[ZFS_MAXNAMELEN];
        uint64_t prevsnap_obj;
        boolean_t seenfrom, seento, replicate, doall, fromorigin;
        boolean_t verbose, dryrun, parsable, progress;
        int outfd;
        boolean_t err;
        nvlist_t *fss;
        nvlist_t *snapholds;
#endif /* ! codereview */
        avl_tree_t *fsavl;
        snapfilter_cb_t *filter_cb;
        void *filter_cb_arg;
        nvlist_t *debugnv;
        char holdtag[ZFS_MAXNAMELEN];
        int cleanup_fd;
        uint64_t size;
} send_dump_data_t;

static int
estimate_ioctl(zfs_handle_t *zhp, uint64_t fromsnap_obj,
    boolean_t fromorigin, uint64_t *sizep)
{
        zfs_cmd_t zc = { 0 };
        libzfs_handle_t *hdl = zhp->zfs_hdl;

        assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
        assert(fromsnap_obj == 0 || !fromorigin);

        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
        zc.zc_obj = fromorigin;
        zc.zc_sendobj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
        zc.zc_fromobj = fromsnap_obj;
        zc.zc_guid = 1;  /* estimate flag */

        if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_SEND, &zc) != 0) {
                char errbuf[1024];
                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
```

```
                    "warning: cannot estimate space for ’%s’"), zhp->zfs_name);

                switch (errno) {
                case EXDEV:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "not an earlier snapshot from the same fs"));
                        return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));

                case ENOENT:
                        if (zfs_dataset_exists(hdl, zc.zc_name,
                            ZFS_TYPE_SNAPSHOT)) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "incremental source (@%s) does not exist"),
                                    zc.zc_value);
                        }
                        return (zfs_error(hdl, EZFS_NOENT, errbuf));

                case EDQUOT:
                case EFBIG:
                case EIO:
                case ENOLINK:
                case ENOSPC:
                case ENOSTR:
                case ENXIO:
                case EPIPE:
                case ERANGE:
                case EFAULT:
                case EROFS:
                        zfs_error_aux(hdl, strerror(errno));
                        return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));

                default:
                        return (zfs_standard_error(hdl, errno, errbuf));
                }
        }

        *sizep = zc.zc_objset_type;

        return (0);
}

/*
 * Dumps a backup of the given snapshot (incremental from fromsnap if it’s not
 * NULL) to the file descriptor specified by outfd.
 */
static int
dump_ioctl(zfs_handle_t *zhp, const char *fromsnap, uint64_t fromsnap_obj,
    boolean_t fromorigin, int outfd, nvlist_t *debugnv)
{
        zfs_cmd_t zc = { 0 };
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        nvlist_t *thisdbg;

        assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
        assert(fromsnap_obj == 0 || !fromorigin);

        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
        zc.zc_cookie = outfd;
        zc.zc_obj = fromorigin;
        zc.zc_sendobj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
        zc.zc_fromobj = fromsnap_obj;

        VERIFY(0 == nvlist_alloc(&thisdbg, NV_UNIQUE_NAME, 0));
        if (fromsnap && fromsnap[0] != ’\0’) {
                VERIFY(0 == nvlist_add_string(thisdbg,
                    "fromsnap", fromsnap));
        }

        if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_SEND, &zc) != 0) {
                char errbuf[1024];
                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "warning: cannot send ’%s’"), zhp->zfs_name);

                VERIFY(0 == nvlist_add_uint64(thisdbg, "error", errno));
                if (debugnv) {
                        VERIFY(0 == nvlist_add_nvlist(debugnv,
                            zhp->zfs_name, thisdbg));
                }
                nvlist_free(thisdbg);

                switch (errno) {
                case EXDEV:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "not an earlier snapshot from the same fs"));
                        return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));

                case ENOENT:
                        if (zfs_dataset_exists(hdl, zc.zc_name,
                            ZFS_TYPE_SNAPSHOT)) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "incremental source (@%s) does not exist"),
                                    zc.zc_value);
                        }
                        return (zfs_error(hdl, EZFS_NOENT, errbuf));

                case EDQUOT:
                case EFBIG:
                case EIO:
                case ENOLINK:
                case ENOSPC:
                case ENOSTR:
                case ENXIO:
                case EPIPE:
                case ERANGE:
                case EFAULT:
                case EROFS:
                        zfs_error_aux(hdl, strerror(errno));
                        return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));

                default:
                        return (zfs_standard_error(hdl, errno, errbuf));
                }
        }

        if (debugnv)
                VERIFY(0 == nvlist_add_nvlist(debugnv, zhp->zfs_name, thisdbg));
        nvlist_free(thisdbg);

        return (0);
}

static int
hold_for_send(zfs_handle_t *zhp, send_dump_data_t *sdd)
{
        zfs_handle_t *pzhp;
        int error = 0;
        char *thissnap;

        assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);

        if (sdd->dryrun)
                return (0);
```

```

        /*
         * We process if snapholds is not NULL even if on a dry run as
         * this is used to pre-calculate the required holds so they can
         * be processed in one kernel request
         * zfs_send() only opens a cleanup_fd for sends that need it,
         * e.g. replication and doall.
         */
        if (sdd->snapholds == NULL)
        if (sdd->cleanup_fd == -1)
                return (0);

        thissnap = strchr(zhp->zfs_name, ’@’) + 1;
        *(thissnap - 1) = ’\0’;
        pzhp = zfs_open(zhp->zfs_hdl, zhp->zfs_name, ZFS_TYPE_DATASET);
        *(thissnap - 1) = ’@’;

        /*
         * It’s OK if the parent no longer exists.  The send code will
         * handle that error.
         */
        if (pzhp) {
                error = zfs_hold_add(pzhp, thissnap, sdd->holdtag, B_TRUE,
                    sdd->snapholds);
                error = zfs_hold(pzhp, thissnap, sdd->holdtag,
                    B_FALSE, B_TRUE, sdd->cleanup_fd);
                zfs_close(pzhp);
        }

        return (error);
}
______unchanged_portion_omitted_

/*
 * Generate a send stream for the dataset identified by the argument zhp.
 *
 * The content of the send stream is the snapshot identified by
 * ’tosnap’.  Incremental streams are requested in two ways:
 *     - from the snapshot identified by "fromsnap" (if non-null) or
 *     - from the origin of the dataset identified by zhp, which must
 *       be a clone.  In this case, "fromsnap" is null and "fromorigin"
 *       is TRUE.
 *
 * The send stream is recursive (i.e. dumps a hierarchy of snapshots) and
 * uses a special header (with a hdrtype field of DMU_COMPOUNDSTREAM)
 * if "replicate" is set.  If "doall" is set, dump all the intermediate
 * snapshots. The DMU_COMPOUNDSTREAM header is used in the "doall"
 * case too. If "props" is set, send properties.
 */
int
zfs_send(zfs_handle_t *zhp, const char *fromsnap, const char *tosnap,
    sendflags_t *flags, int outfd, snapfilter_cb_t filter_func,
    void *cb_arg, nvlist_t **debugnvp)
{
        char errbuf[1024];
        send_dump_data_t sdd = { 0 };
        int err = 0;
        nvlist_t *fss = NULL;
        avl_tree_t *fsavl = NULL;
        static uint64_t holdseq;
        int spa_version;
        pthread_t tid;
        int pipefd[2];
        dedup_arg_t dda = { 0 };
        int featureflags = 0;
        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
            "cannot send ’%s’"), zhp->zfs_name);

        if (fromsnap && fromsnap[0] == ’\0’) {
                zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
                    "zero-length incremental source"));
                return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
        }

        if (zhp->zfs_type == ZFS_TYPE_FILESYSTEM) {
                uint64_t version;
                version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
                if (version >= ZPL_VERSION_SA) {
                        featureflags |= DMU_BACKUP_FEATURE_SA_SPILL;
                }
        }

        if (flags->dedup && !flags->dryrun) {
                featureflags |= (DMU_BACKUP_FEATURE_DEDUP |
                    DMU_BACKUP_FEATURE_DEDUPPROPS);
                if (err = pipe(pipefd)) {
                        zfs_error_aux(zhp->zfs_hdl, strerror(errno));
                        return (zfs_error(zhp->zfs_hdl, EZFS_PIPEFAILED,
                            errbuf));
                }
                dda.outputfd = outfd;
                dda.inputfd = pipefd[1];
                dda.dedup_hdl = zhp->zfs_hdl;
                if (err = pthread_create(&tid, NULL, cksummer, &dda)) {
                        (void) close(pipefd[0]);
                        (void) close(pipefd[1]);
                        zfs_error_aux(zhp->zfs_hdl, strerror(errno));
                        return (zfs_error(zhp->zfs_hdl,
                            EZFS_THREADCREATEFAILED, errbuf));
                }
        }

        if (flags->replicate || flags->doall || flags->props) {
                dmu_replay_record_t drr = { 0 };
                char *packbuf = NULL;
                size_t buflen = 0;
                zio_cksum_t zc = { 0 };

                if (flags->replicate || flags->props) {
                        nvlist_t *hdrnv;

                        VERIFY(0 == nvlist_alloc(&hdrnv, NV_UNIQUE_NAME, 0));
                        if (fromsnap) {
                                VERIFY(0 == nvlist_add_string(hdrnv,
                                    "fromsnap", fromsnap));
                        }
                        VERIFY(0 == nvlist_add_string(hdrnv, "tosnap", tosnap));
                        if (!flags->replicate) {
                                VERIFY(0 == nvlist_add_boolean(hdrnv,
                                    "not_recursive"));
                        }

                        err = gather_nvlist(zhp->zfs_hdl, zhp->zfs_name,
                            fromsnap, tosnap, flags->replicate, &fss, &fsavl);
                        if (err)
                                goto err_out;
                        VERIFY(0 == nvlist_add_nvlist(hdrnv, "fss", fss));
                        err = nvlist_pack(hdrnv, &packbuf, &buflen,
                            NV_ENCODE_XDR, 0);
                        if (debugnvp)
                                *debugnvp = hdrnv;
```

```
                        else
                                nvlist_free(hdrnv);
                        if (err) {
                                fsavl_destroy(fsavl);
                                nvlist_free(fss);
                                goto stderr_out;
                        }
                }

                if (!flags->dryrun) {
                        /* write first begin record */
                        drr.drr_type = DRR_BEGIN;
                        drr.drr_u.drr_begin.drr_magic = DMU_BACKUP_MAGIC;
                        DMU_SET_STREAM_HDRTYPE(drr.drr_u.drr_begin.
                            drr_versioninfo, DMU_COMPOUNDSTREAM);
                        DMU_SET_FEATUREFLAGS(drr.drr_u.drr_begin.
                            drr_versioninfo, featureflags);
                        (void) snprintf(drr.drr_u.drr_begin.drr_toname,
                            sizeof (drr.drr_u.drr_begin.drr_toname),
                            "%s@%s", zhp->zfs_name, tosnap);
                        drr.drr_payloadlen = buflen;
                        err = cksum_and_write(&drr, sizeof (drr), &zc, outfd);

                        /* write header nvlist */
                        if (err != -1 && packbuf != NULL) {
                                err = cksum_and_write(packbuf, buflen, &zc,
                                    outfd);
                        }
                        free(packbuf);
                        if (err == -1) {
                                fsavl_destroy(fsavl);
                                nvlist_free(fss);
                                err = errno;
                                goto stderr_out;
                        }

                        /* write end record */
                        bzero(&drr, sizeof (drr));
                        drr.drr_type = DRR_END;
                        drr.drr_u.drr_end.drr_checksum = zc;
                        err = write(outfd, &drr, sizeof (drr));
                        if (err == -1) {
                                fsavl_destroy(fsavl);
                                nvlist_free(fss);
                                err = errno;
                                goto stderr_out;
                        }

                        err = 0;
                }
        }

        /* dump each stream */
        sdd.fromsnap = fromsnap;
        sdd.tosnap = tosnap;
        if (flags->dedup)
                sdd.outfd = pipefd[0];
        else
                sdd.outfd = outfd;
        sdd.replicate = flags->replicate;
        sdd.doall = flags->doall;
        sdd.fromorigin = flags->fromorigin;
        sdd.fss = fss;
        sdd.fsavl = fsavl;
        sdd.verbose = flags->verbose;
        sdd.parsable = flags->parsable;
        sdd.progress = flags->progress;
        sdd.dryrun = flags->dryrun;
        sdd.filter_cb = filter_func;
        sdd.filter_cb_arg = cb_arg;
        if (debugnvp)
                sdd.debugnv = *debugnvp;

        /*
         * Some flags require that we place user holds on the datasets that are
         * being sent so they don’t get destroyed during the send. We can skip
         * this step if the pool is imported read-only since the datasets cannot
         * be destroyed.
         */
        if (!flags->dryrun && !zpool_get_prop_int(zfs_get_pool_handle(zhp),
            ZPOOL_PROP_READONLY, NULL) &&
            zfs_spa_version(zhp, &spa_version) == 0 &&
            spa_version >= SPA_VERSION_USERREFS &&
            (flags->doall || flags->replicate)) {
                ++holdseq;
                (void) snprintf(sdd.holdtag, sizeof (sdd.holdtag),
                    ".send-%d-%llu", getpid(), (u_longlong_t)holdseq);
                sdd.cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
                if (sdd.cleanup_fd < 0) {
                        err = errno;
                        goto stderr_out;
                }
                sdd.snapholds = fnvlist_alloc();
#endif /* ! codereview */
        } else {
                sdd.cleanup_fd = -1;
                sdd.snapholds = NULL;
#endif /* ! codereview */
        }
        if (flags->verbose) {
                /*
                 * Do a verbose no-op dry run to get all the verbose output
                 * before generating any data.  Then do a non-verbose real
                 * run to generate the streams.
                 */
                sdd.dryrun = B_TRUE;
                err = dump_filesystems(zhp, &sdd);
                sdd.dryrun = flags->dryrun;
                sdd.verbose = B_FALSE;
                if (flags->parsable) {
                        (void) fprintf(stderr, "size\t%llu\n",
                            (longlong_t)sdd.size);
                } else {
                        char buf[16];
                        zfs_nicenum(sdd.size, buf, sizeof (buf));
                        (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
                            "total estimated size is %s\n"), buf);
                }
        }

        if (sdd.snapholds != NULL) {
                /* Holds are required */
                if (!flags->verbose) {
                        /*
                         * A verbose dry run wasn’t done so do a non-verbose
                         * dry run to collate snapshot hold’s.
                         */
                        sdd.dryrun = B_TRUE;
                        err = dump_filesystems(zhp, &sdd);
                        sdd.dryrun = flags->dryrun;
                }
```

```
                if (err != 0) {
                        fnvlist_free(sdd.snapholds);
                        goto stderr_out;
                }

                err = zfs_hold_apply(zhp, B_TRUE, sdd.cleanup_fd, sdd.snapholds)
                fnvlist_free(sdd.snapholds);
                if (err != 0)
                        goto stderr_out;
        }

#endif /* ! codereview */
        err = dump_filesystems(zhp, &sdd);
        fsavl_destroy(fsavl);
        nvlist_free(fss);

        if (flags->dedup) {
                (void) close(pipefd[0]);
                (void) pthread_join(tid, NULL);
        }

        if (sdd.cleanup_fd != -1) {
                VERIFY(0 == close(sdd.cleanup_fd));
                sdd.cleanup_fd = -1;
        }

        if (!flags->dryrun && (flags->replicate || flags->doall ||
            flags->props)) {
                /*
                 * write final end record.  NB: want to do this even if
                 * there was some error, because it might not be totally
                 * failed.
                 */
                dmu_replay_record_t drr = { 0 };
                drr.drr_type = DRR_END;
                if (write(outfd, &drr, sizeof (drr)) == -1) {
                        return (zfs_standard_error(zhp->zfs_hdl,
                            errno, errbuf));
                }
        }

        return (err || sdd.err);

stderr_out:
        err = zfs_standard_error(zhp->zfs_hdl, err, errbuf);
err_out:
        if (sdd.cleanup_fd != -1)
                VERIFY(0 == close(sdd.cleanup_fd));
        if (flags->dedup) {
                (void) pthread_cancel(tid);
                (void) pthread_join(tid, NULL);
                (void) close(pipefd[0]);
        }
        return (err);
}

/*
 * Routines specific to "zfs recv"
 */

static int
recv_read(libzfs_handle_t *hdl, int fd, void *buf, int ilen,
    boolean_t byteswap, zio_cksum_t *zc)
{
        char *cp = buf;
        int rv;
        int len = ilen;

        do {
                rv = read(fd, cp, len);
                cp += rv;
                len -= rv;
        } while (rv > 0);

        if (rv < 0 || len != 0) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                    "failed to read from stream"));
                return (zfs_error(hdl, EZFS_BADSTREAM, dgettext(TEXT_DOMAIN,
                    "cannot receive")));
        }

        if (zc) {
                if (byteswap)
                        fletcher_4_incremental_byteswap(buf, ilen, zc);
                else
                        fletcher_4_incremental_native(buf, ilen, zc);
        }
        return (0);
}

static int
recv_read_nvlist(libzfs_handle_t *hdl, int fd, int len, nvlist_t **nvp,
    boolean_t byteswap, zio_cksum_t *zc)
{
        char *buf;
        int err;

        buf = zfs_alloc(hdl, len);
        if (buf == NULL)
                return (ENOMEM);

        err = recv_read(hdl, fd, buf, len, byteswap, zc);
        if (err != 0) {
                free(buf);
                return (err);
        }

        err = nvlist_unpack(buf, len, nvp, 0);
        free(buf);
        if (err != 0) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
                    "stream (malformed nvlist)"));
                return (EINVAL);
        }
        return (0);
}

static int
recv_rename(libzfs_handle_t *hdl, const char *name, const char *tryname,
    int baselen, char *newname, recvflags_t *flags)
{
        static int seq;
        zfs_cmd_t zc = { 0 };
        int err;
        prop_changelist_t *clp;
        zfs_handle_t *zhp;

        zhp = zfs_open(hdl, name, ZFS_TYPE_DATASET);
        if (zhp == NULL)
                return (-1);
        clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
            flags->force ? MS_FORCE : 0);
```

```
        zfs_close(zhp);
        if (clp == NULL)
                return (-1);
        err = changelist_prefix(clp);
        if (err)
                return (err);

        zc.zc_objset_type = DMU_OST_ZFS;
        (void) strlcpy(zc.zc_name, name, sizeof (zc.zc_name));

        if (tryname) {
                (void) strcpy(newname, tryname);

                (void) strlcpy(zc.zc_value, tryname, sizeof (zc.zc_value));

                if (flags->verbose) {
                        (void) printf("attempting rename %s to %s\n",
                            zc.zc_name, zc.zc_value);
                }
                err = ioctl(hdl->libzfs_fd, ZFS_IOC_RENAME, &zc);
                if (err == 0)
                        changelist_rename(clp, name, tryname);
        } else {
                err = ENOENT;
        }

        if (err != 0 && strncmp(name + baselen, "recv-", 5) != 0) {
                seq++;

                (void) snprintf(newname, ZFS_MAXNAMELEN, "%.*srecv-%u-%u",
                    baselen, name, getpid(), seq);
                (void) strlcpy(zc.zc_value, newname, sizeof (zc.zc_value));

                if (flags->verbose) {
                        (void) printf("failed - trying rename %s to %s\n",
                            zc.zc_name, zc.zc_value);
                }
                err = ioctl(hdl->libzfs_fd, ZFS_IOC_RENAME, &zc);
                if (err == 0)
                        changelist_rename(clp, name, newname);
                if (err && flags->verbose) {
                        (void) printf("failed (%u) - "
                            "will try again on next pass\n", errno);
                }
                err = EAGAIN;
        } else if (flags->verbose) {
                if (err == 0)
                        (void) printf("success\n");
                else
                        (void) printf("failed (%u)\n", errno);
        }

        (void) changelist_postfix(clp);
        changelist_free(clp);

        return (err);
}

static int
recv_destroy(libzfs_handle_t *hdl, const char *name, int baselen,
    char *newname, recvflags_t *flags)
{
        zfs_cmd_t zc = { 0 };
        int err = 0;
        prop_changelist_t *clp;
        zfs_handle_t *zhp;
        boolean_t defer = B_FALSE;
        int spa_version;

        zhp = zfs_open(hdl, name, ZFS_TYPE_DATASET);
        if (zhp == NULL)
                return (-1);
        clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
            flags->force ? MS_FORCE : 0);
        if (zfs_get_type(zhp) == ZFS_TYPE_SNAPSHOT &&
            zfs_spa_version(zhp, &spa_version) == 0 &&
            spa_version >= SPA_VERSION_USERREFS)
                defer = B_TRUE;
        zfs_close(zhp);
        if (clp == NULL)
                return (-1);
        err = changelist_prefix(clp);
        if (err)
                return (err);

        zc.zc_objset_type = DMU_OST_ZFS;
        zc.zc_defer_destroy = defer;
        (void) strlcpy(zc.zc_name, name, sizeof (zc.zc_name));

        if (flags->verbose)
                (void) printf("attempting destroy %s\n", zc.zc_name);
        err = ioctl(hdl->libzfs_fd, ZFS_IOC_DESTROY, &zc);
        if (err == 0) {
                if (flags->verbose)
                        (void) printf("success\n");
                changelist_remove(clp, zc.zc_name);
        }

        (void) changelist_postfix(clp);
        changelist_free(clp);

        /*
         * Deferred destroy might destroy the snapshot or only mark it to be
         * destroyed later, and it returns success in either case.
         */
        if (err != 0 || (defer && zfs_dataset_exists(hdl, name,
            ZFS_TYPE_SNAPSHOT))) {
                err = recv_rename(hdl, name, NULL, baselen, newname, flags);
        }

        return (err);
}

typedef struct guid_to_name_data {
        uint64_t guid;
        char *name;
        char *skip;
} guid_to_name_data_t;

static int
guid_to_name_cb(zfs_handle_t *zhp, void *arg)
{
        guid_to_name_data_t *gtnd = arg;
        int err;

        if (gtnd->skip != NULL &&
            strcmp(zhp->zfs_name, gtnd->skip) == 0) {
                return (0);
        }

        if (zhp->zfs_dmustats.dds_guid == gtnd->guid) {
                (void) strcpy(gtnd->name, zhp->zfs_name);
```

```
                zfs_close(zhp);
                return (EEXIST);
        }

        err = zfs_iter_children(zhp, guid_to_name_cb, gtnd);
        zfs_close(zhp);
        return (err);
}

/*
 * Attempt to find the local dataset associated with this guid.  In the case of
 * multiple matches, we attempt to find the "best" match by searching
 * progressively larger portions of the hierarchy.  This allows one to send a
 * tree of datasets individually and guarantee that we will find the source
 * guid within that hierarchy, even if there are multiple matches elsewhere.
 */
static int
guid_to_name(libzfs_handle_t *hdl, const char *parent, uint64_t guid,
    char *name)
{
        /* exhaustive search all local snapshots */
        char pname[ZFS_MAXNAMELEN];
        guid_to_name_data_t gtnd;
        int err = 0;
        zfs_handle_t *zhp;
        char *cp;

        gtnd.guid = guid;
        gtnd.name = name;
        gtnd.skip = NULL;

        (void) strlcpy(pname, parent, sizeof (pname));

        /*
         * Search progressively larger portions of the hierarchy.  This will
         * select the "most local" version of the origin snapshot in the case
         * that there are multiple matching snapshots in the system.
         */
        while ((cp = strrchr(pname, ’/’)) != NULL) {

                /* Chop off the last component and open the parent */
                *cp = ’\0’;
                zhp = make_dataset_handle(hdl, pname);

                if (zhp == NULL)
                        continue;

                err = zfs_iter_children(zhp, guid_to_name_cb, &gtnd);
                zfs_close(zhp);
                if (err == EEXIST)
                        return (0);

                /*
                 * Remember the dataset that we already searched, so we
                 * skip it next time through.
                 */
                gtnd.skip = pname;
        }

        return (ENOENT);
}

/*
 * Return +1 if guid1 is before guid2, 0 if they are the same, and -1 if
 * guid1 is after guid2.
 */
```

```
static int
created_before(libzfs_handle_t *hdl, avl_tree_t *avl,
    uint64_t guid1, uint64_t guid2)
{
        nvlist_t *nvfs;
        char *fsname, *snapname;
        char buf[ZFS_MAXNAMELEN];
        int rv;
        zfs_handle_t *guid1hdl, *guid2hdl;
        uint64_t create1, create2;

        if (guid2 == 0)
                return (0);
        if (guid1 == 0)
                return (1);

        nvfs = fsavl_find(avl, guid1, &snapname);
        VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));
        (void) snprintf(buf, sizeof (buf), "%s@%s", fsname, snapname);
        guid1hdl = zfs_open(hdl, buf, ZFS_TYPE_SNAPSHOT);
        if (guid1hdl == NULL)
                return (-1);

        nvfs = fsavl_find(avl, guid2, &snapname);
        VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));
        (void) snprintf(buf, sizeof (buf), "%s@%s", fsname, snapname);
        guid2hdl = zfs_open(hdl, buf, ZFS_TYPE_SNAPSHOT);
        if (guid2hdl == NULL) {
                zfs_close(guid1hdl);
                return (-1);
        }

        create1 = zfs_prop_get_int(guid1hdl, ZFS_PROP_CREATETXG);
        create2 = zfs_prop_get_int(guid2hdl, ZFS_PROP_CREATETXG);

        if (create1 < create2)
                rv = -1;
        else if (create1 > create2)
                rv = +1;
        else
                rv = 0;

        zfs_close(guid1hdl);
        zfs_close(guid2hdl);

        return (rv);
}

static int
recv_incremental_replication(libzfs_handle_t *hdl, const char *tofs,
    recvflags_t *flags, nvlist_t *stream_nv, avl_tree_t *stream_avl,
    nvlist_t *renamed)
{
        nvlist_t *local_nv;
        avl_tree_t *local_avl;
        nvpair_t *fselem, *nextfselem;
        char *fromsnap;
        char newname[ZFS_MAXNAMELEN];
        int error;
        boolean_t needagain, progress, recursive;
        char *s1, *s2;

        VERIFY(0 == nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap));

        recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
            ENOENT);
```

```
        if (flags->dryrun)
                return (0);

again:
        needagain = progress = B_FALSE;

        if ((error = gather_nvlist(hdl, tofs, fromsnap, NULL,
            recursive, &local_nv, &local_avl)) != 0)
                return (error);

        /*
         * Process deletes and renames
         */
        for (fselem = nvlist_next_nvpair(local_nv, NULL);
            fselem; fselem = nextfselem) {
                nvlist_t *nvfs, *snaps;
                nvlist_t *stream_nvfs = NULL;
                nvpair_t *snapelem, *nextsnapelem;
                uint64_t fromguid = 0;
                uint64_t originguid = 0;
                uint64_t stream_originguid = 0;
                uint64_t parent_fromsnap_guid, stream_parent_fromsnap_guid;
                char *fsname, *stream_fsname;

                nextfselem = nvlist_next_nvpair(local_nv, fselem);

                VERIFY(0 == nvpair_value_nvlist(fselem, &nvfs));
                VERIFY(0 == nvlist_lookup_nvlist(nvfs, "snaps", &snaps));
                VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));
                VERIFY(0 == nvlist_lookup_uint64(nvfs, "parentfromsnap",
                    &parent_fromsnap_guid));
                (void) nvlist_lookup_uint64(nvfs, "origin", &originguid);

                /*
                 * First find the stream’s fs, so we can check for
                 * a different origin (due to "zfs promote")
                 */
                for (snapelem = nvlist_next_nvpair(snaps, NULL);
                    snapelem; snapelem = nvlist_next_nvpair(snaps, snapelem)) {
                        uint64_t thisguid;

                        VERIFY(0 == nvpair_value_uint64(snapelem, &thisguid));
                        stream_nvfs = fsavl_find(stream_avl, thisguid, NULL);

                        if (stream_nvfs != NULL)
                                break;
                }

                /* check for promote */
                (void) nvlist_lookup_uint64(stream_nvfs, "origin",
                    &stream_originguid);
                if (stream_nvfs && originguid != stream_originguid) {
                        switch (created_before(hdl, local_avl,
                            stream_originguid, originguid)) {
                        case 1: {
                                /* promote it! */
                                zfs_cmd_t zc = { 0 };
                                nvlist_t *origin_nvfs;
                                char *origin_fsname;

                                if (flags->verbose)
                                        (void) printf("promoting %s\n", fsname);

                                origin_nvfs = fsavl_find(local_avl, originguid,
                                    NULL);
                                VERIFY(0 == nvlist_lookup_string(origin_nvfs,
                                    "name", &origin_fsname));
                                (void) strlcpy(zc.zc_value, origin_fsname,
                                    sizeof (zc.zc_value));
                                (void) strlcpy(zc.zc_name, fsname,
                                    sizeof (zc.zc_name));
                                error = zfs_ioctl(hdl, ZFS_IOC_PROMOTE, &zc);
                                if (error == 0)
                                        progress = B_TRUE;
                                break;
                        }
                        default:
                                break;
                        case -1:
                                fsavl_destroy(local_avl);
                                nvlist_free(local_nv);
                                return (-1);
                        }
                        /*
                         * We had/have the wrong origin, therefore our
                         * list of snapshots is wrong.  Need to handle
                         * them on the next pass.
                         */
                        needagain = B_TRUE;
                        continue;
                }

                for (snapelem = nvlist_next_nvpair(snaps, NULL);
                    snapelem; snapelem = nextsnapelem) {
                        uint64_t thisguid;
                        char *stream_snapname;
                        nvlist_t *found, *props;

                        nextsnapelem = nvlist_next_nvpair(snaps, snapelem);

                        VERIFY(0 == nvpair_value_uint64(snapelem, &thisguid));
                        found = fsavl_find(stream_avl, thisguid,
                            &stream_snapname);

                        /* check for delete */
                        if (found == NULL) {
                                char name[ZFS_MAXNAMELEN];

                                if (!flags->force)
                                        continue;

                                (void) snprintf(name, sizeof (name), "%s@%s",
                                    fsname, nvpair_name(snapelem));

                                error = recv_destroy(hdl, name,
                                    strlen(fsname)+1, newname, flags);
                                if (error)
                                        needagain = B_TRUE;
                                else
                                        progress = B_TRUE;
                                continue;
                        }

                        stream_nvfs = found;

                        if (0 == nvlist_lookup_nvlist(stream_nvfs, "snapprops",
                            &props) && 0 == nvlist_lookup_nvlist(props,
                            stream_snapname, &props)) {
                                zfs_cmd_t zc = { 0 };

                                zc.zc_cookie = B_TRUE; /* received */
```

```
                                (void) snprintf(zc.zc_name, sizeof (zc.zc_name),
                                    "%s@%s", fsname, nvpair_name(snapelem));
                                if (zcmd_write_src_nvlist(hdl, &zc,
                                    props) == 0) {
                                        (void) zfs_ioctl(hdl,
                                            ZFS_IOC_SET_PROP, &zc);
                                        zcmd_free_nvlists(&zc);
                                }
                        }

                        /* check for different snapname */
                        if (strcmp(nvpair_name(snapelem),
                            stream_snapname) != 0) {
                                char name[ZFS_MAXNAMELEN];
                                char tryname[ZFS_MAXNAMELEN];

                                (void) snprintf(name, sizeof (name), "%s@%s",
                                    fsname, nvpair_name(snapelem));
                                (void) snprintf(tryname, sizeof (name), "%s@%s",
                                    fsname, stream_snapname);

                                error = recv_rename(hdl, name, tryname,
                                    strlen(fsname)+1, newname, flags);
                                if (error)
                                        needagain = B_TRUE;
                                else
                                        progress = B_TRUE;
                        }

                        if (strcmp(stream_snapname, fromsnap) == 0)
                                fromguid = thisguid;
                }

                /* check for delete */
                if (stream_nvfs == NULL) {
                        if (!flags->force)
                                continue;

                        error = recv_destroy(hdl, fsname, strlen(tofs)+1,
                            newname, flags);
                        if (error)
                                needagain = B_TRUE;
                        else
                                progress = B_TRUE;
                        continue;
                }

                if (fromguid == 0) {
                        if (flags->verbose) {
                                (void) printf("local fs %s does not have "
                                    "fromsnap (%s in stream); must have "
                                    "been deleted locally; ignoring\n",
                                    fsname, fromsnap);
                        }
                        continue;
                }

                VERIFY(0 == nvlist_lookup_string(stream_nvfs,
                    "name", &stream_fsname));
                VERIFY(0 == nvlist_lookup_uint64(stream_nvfs,
                    "parentfromsnap", &stream_parent_fromsnap_guid));

                s1 = strrchr(fsname, ’/’);
                s2 = strrchr(stream_fsname, ’/’);

                /*
```

```
                 * Check for rename. If the exact receive path is specified, it
                 * does not count as a rename, but we still need to check the
                 * datasets beneath it.
                 */
                if ((stream_parent_fromsnap_guid != 0 &&
                    parent_fromsnap_guid != 0 &&
                    stream_parent_fromsnap_guid != parent_fromsnap_guid) ||
                    ((flags->isprefix || strcmp(tofs, fsname) != 0) &&
                    (s1 != NULL) && (s2 != NULL) && strcmp(s1, s2) != 0)) {
                        nvlist_t *parent;
                        char tryname[ZFS_MAXNAMELEN];

                        parent = fsavl_find(local_avl,
                            stream_parent_fromsnap_guid, NULL);
                        /*
                         * NB: parent might not be found if we used the
                         * tosnap for stream_parent_fromsnap_guid,
                         * because the parent is a newly-created fs;
                         * we'll be able to rename it after we recv the
                         * new fs.
                         */
                        if (parent != NULL) {
                                char *pname;

                                VERIFY(0 == nvlist_lookup_string(parent, "name",
                                    &pname));
                                (void) snprintf(tryname, sizeof (tryname),
                                    "%s%s", pname, strrchr(stream_fsname, '/'));
                        } else {
                                tryname[0] = '\0';
                                if (flags->verbose) {
                                        (void) printf("local fs %s new parent "
                                            "not found\n", fsname);
                                }
                        }

                        newname[0] = '\0';

                        error = recv_rename(hdl, fsname, tryname,
                            strlen(tofs)+1, newname, flags);

                        if (renamed != NULL && newname[0] != '\0') {
                                VERIFY(0 == nvlist_add_boolean(renamed,
                                    newname));
                        }

                        if (error)
                                needagain = B_TRUE;
                        else
                                progress = B_TRUE;
                }
        }

        fsavl_destroy(local_avl);
        nvlist_free(local_nv);

        if (needagain && progress) {
                /* do another pass to fix up temporary names */
                if (flags->verbose)
                        (void) printf("another pass:\n");
                goto again;
        }

        return (needagain);
}
```

```
static int
zfs_receive_package(libzfs_handle_t *hdl, int fd, const char *destname,
    recvflags_t *flags, dmu_replay_record_t *drr, zio_cksum_t *zc,
    char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
{
        nvlist_t *stream_nv = NULL;
        avl_tree_t *stream_avl = NULL;
        char *fromsnap = NULL;
        char *cp;
        char tofs[ZFS_MAXNAMELEN];
        char sendfs[ZFS_MAXNAMELEN];
        char errbuf[1024];
        dmu_replay_record_t drre;
        int error;
        boolean_t anyerr = B_FALSE;
        boolean_t softerr = B_FALSE;
        boolean_t recursive;

        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
            "cannot receive"));

        assert(drr->drr_type == DRR_BEGIN);
        assert(drr->drr_u.drr_begin.drr_magic == DMU_BACKUP_MAGIC);
        assert(DMU_GET_STREAM_HDRTYPE(drr->drr_u.drr_begin.drr_versioninfo) ==
            DMU_COMPOUNDSTREAM);

        /*
         * Read in the nvlist from the stream.
         */
        if (drr->drr_payloadlen != 0) {
                error = recv_read_nvlist(hdl, fd, drr->drr_payloadlen,
                    &stream_nv, flags->byteswap, zc);
                if (error) {
                        error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
                        goto out;
                }
        }

        recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
            ENOENT);

        if (recursive && strchr(destname, '@')) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                    "cannot specify snapshot name for multi-snapshot stream"));
                error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
                goto out;
        }

        /*
         * Read in the end record and verify checksum.
         */
        if (0 != (error = recv_read(hdl, fd, &drre, sizeof (drre),
            flags->byteswap, NULL)))
                goto out;
        if (flags->byteswap) {
                drre.drr_type = BSWAP_32(drre.drr_type);
                drre.drr_u.drr_end.drr_checksum.zc_word[0] =
                    BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[0]);
                drre.drr_u.drr_end.drr_checksum.zc_word[1] =
                    BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[1]);
                drre.drr_u.drr_end.drr_checksum.zc_word[2] =
                    BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[2]);
                drre.drr_u.drr_end.drr_checksum.zc_word[3] =
                    BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[3]);
        }
        if (drre.drr_type != DRR_END) {
                error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
                goto out;
        }
        if (!ZIO_CHECKSUM_EQUAL(drre.drr_u.drr_end.drr_checksum, *zc)) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                    "incorrect header checksum"));
                error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
                goto out;
        }

        (void) nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap);

        if (drr->drr_payloadlen != 0) {
                nvlist_t *stream_fss;

                VERIFY(0 == nvlist_lookup_nvlist(stream_nv, "fss",
                    &stream_fss));
                if ((stream_avl = fsavl_create(stream_fss)) == NULL) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "couldn’t allocate avl tree"));
                        error = zfs_error(hdl, EZFS_NOMEM, errbuf);
                        goto out;
                }

                if (fromsnap != NULL) {
                        nvlist_t *renamed = NULL;
                        nvpair_t *pair = NULL;

                        (void) strlcpy(tofs, destname, ZFS_MAXNAMELEN);
                        if (flags->isprefix) {
                                struct drr_begin *drrb = &drr->drr_u.drr_begin;
                                int i;

                                if (flags->istail) {
                                        cp = strrchr(drrb->drr_toname, ’/’);
                                        if (cp == NULL) {
                                                (void) strlcat(tofs, "/",
                                                    ZFS_MAXNAMELEN);
                                                i = 0;
                                        } else {
                                                i = (cp - drrb->drr_toname);
                                        }
                                } else {
                                        i = strcspn(drrb->drr_toname, "/@");
                                }
                                /* zfs_receive_one() will create_parents() */
                                (void) strlcat(tofs, &drrb->drr_toname[i],
                                    ZFS_MAXNAMELEN);
                                *strchr(tofs, ’@’) = ’\0’;
                        }

                        if (recursive && !flags->dryrun && !flags->nomount) {
                                VERIFY(0 == nvlist_alloc(&renamed,
                                    NV_UNIQUE_NAME, 0));
                        }

                        softerr = recv_incremental_replication(hdl, tofs, flags,
                            stream_nv, stream_avl, renamed);

                        /* Unmount renamed filesystems before receiving. */
                        while ((pair = nvlist_next_nvpair(renamed,
                            pair)) != NULL) {
                                zfs_handle_t *zhp;
                                prop_changelist_t *clp = NULL;

                                zhp = zfs_open(hdl, nvpair_name(pair),
```

```
                                    ZFS_TYPE_FILESYSTEM);
                                if (zhp != NULL) {
                                        clp = changelist_gather(zhp,
                                            ZFS_PROP_MOUNTPOINT, 0, 0);
                                        zfs_close(zhp);
                                        if (clp != NULL) {
                                                softerr |=
                                                    changelist_prefix(clp);
                                                changelist_free(clp);
                                        }
                                }
                        }

                        nvlist_free(renamed);
                }
        }

        /*
         * Get the fs specified by the first path in the stream (the top level
         * specified by ’zfs send’) and pass it to each invocation of
         * zfs_receive_one().
         */
        (void) strlcpy(sendfs, drr->drr_u.drr_begin.drr_toname,
            ZFS_MAXNAMELEN);
        if ((cp = strchr(sendfs, ’@’)) != NULL)
                *cp = ’\0’;

        /* Finally, receive each contained stream */
        do {
                /*
                 * we should figure out if it has a recoverable
                 * error, in which case do a recv_skip() and drive on.
                 * Note, if we fail due to already having this guid,
                 * zfs_receive_one() will take care of it (ie,
                 * recv_skip() and return 0).
                 */
                error = zfs_receive_impl(hdl, destname, flags, fd,
                    sendfs, stream_nv, stream_avl, top_zfs, cleanup_fd,
                    action_handlep);
                if (error == ENODATA) {
                        error = 0;
                        break;
                }
                anyerr |= error;
        } while (error == 0);

        if (drr->drr_payloadlen != 0 && fromsnap != NULL) {
                /*
                 * Now that we have the fs’s they sent us, try the
                 * renames again.
                 */
                softerr = recv_incremental_replication(hdl, tofs, flags,
                    stream_nv, stream_avl, NULL);
        }

out:
        fsavl_destroy(stream_avl);
        if (stream_nv)
                nvlist_free(stream_nv);
        if (softerr)
                error = -2;
        if (anyerr)
                error = -1;
        return (error);
}
```

```
static void
trunc_prop_errs(int truncated)
{
        ASSERT(truncated != 0);

        if (truncated == 1)
                (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
                    "1 more property could not be set\n"));
        else
                (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
                    "%d more properties could not be set\n"), truncated);
}

static int
recv_skip(libzfs_handle_t *hdl, int fd, boolean_t byteswap)
{
        dmu_replay_record_t *drr;
        void *buf = malloc(1<<20);
        char errbuf[1024];

        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
            "cannot receive:"));

        /* XXX would be great to use lseek if possible... */
        drr = buf;

        while (recv_read(hdl, fd, drr, sizeof (dmu_replay_record_t),
            byteswap, NULL) == 0) {
                if (byteswap)
                        drr->drr_type = BSWAP_32(drr->drr_type);

                switch (drr->drr_type) {
                case DRR_BEGIN:
                        /* NB: not to be used on v2 stream packages */
                        if (drr->drr_payloadlen != 0) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "invalid substream header"));
                                return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
                        }
                        break;

                case DRR_END:
                        free(buf);
                        return (0);

                case DRR_OBJECT:
                        if (byteswap) {
                                drr->drr_u.drr_object.drr_bonuslen =
                                    BSWAP_32(drr->drr_u.drr_object.
                                    drr_bonuslen);
                        }
                        (void) recv_read(hdl, fd, buf,
                            P2ROUNDUP(drr->drr_u.drr_object.drr_bonuslen, 8),
                            B_FALSE, NULL);
                        break;

                case DRR_WRITE:
                        if (byteswap) {
                                drr->drr_u.drr_write.drr_length =
                                    BSWAP_64(drr->drr_u.drr_write.drr_length);
                        }
                        (void) recv_read(hdl, fd, buf,
                            drr->drr_u.drr_write.drr_length, B_FALSE, NULL);
                        break;
                case DRR_SPILL:
                        if (byteswap) {
```

```
                                drr->drr_u.drr_write.drr_length =
                                    BSWAP_64(drr->drr_u.drr_spill.drr_length);
                        }
                        (void) recv_read(hdl, fd, buf,
                            drr->drr_u.drr_spill.drr_length, B_FALSE, NULL);
                        break;
                case DRR_WRITE_BYREF:
                case DRR_FREEOBJECTS:
                case DRR_FREE:
                        break;

                default:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "invalid record type"));
                        return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
                }
        }

        free(buf);
        return (-1);
}

/*
 * Restores a backup of tosnap from the file descriptor specified by infd.
 */
static int
zfs_receive_one(libzfs_handle_t *hdl, int infd, const char *tosnap,
    recvflags_t *flags, dmu_replay_record_t *drr,
    dmu_replay_record_t *drr_noswap, const char *sendfs,
    nvlist_t *stream_nv, avl_tree_t *stream_avl, char **top_zfs, int cleanup_fd,
    uint64_t *action_handlep)
{
        zfs_cmd_t zc = { 0 };
        time_t begin_time;
        int ioctl_err, ioctl_errno, err;
        char *cp;
        struct drr_begin *drrb = &drr->drr_u.drr_begin;
        char errbuf[1024];
        char prop_errbuf[1024];
        const char *chopprefix;
        boolean_t newfs = B_FALSE;
        boolean_t stream_wantsnewfs;
        uint64_t parent_snapguid = 0;
        prop_changelist_t *clp = NULL;
        nvlist_t *snapprops_nvlist = NULL;
        zprop_errflags_t prop_errflags;
        boolean_t recursive;

        begin_time = time(NULL);

        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
            "cannot receive"));

        recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
            ENOENT);

        if (stream_avl != NULL) {
                char *snapname;
                nvlist_t *fs = fsavl_find(stream_avl, drrb->drr_toguid,
                    &snapname);
                nvlist_t *props;
                int ret;

                (void) nvlist_lookup_uint64(fs, "parentfromsnap",
                    &parent_snapguid);
                err = nvlist_lookup_nvlist(fs, "props", &props);
                if (err)
                        VERIFY(0 == nvlist_alloc(&props, NV_UNIQUE_NAME, 0));

                if (flags->canmountoff) {
                        VERIFY(0 == nvlist_add_uint64(props,
                            zfs_prop_to_name(ZFS_PROP_CANMOUNT), 0));
                }
                ret = zcmd_write_src_nvlist(hdl, &zc, props);
                if (err)
                        nvlist_free(props);

                if (0 == nvlist_lookup_nvlist(fs, "snapprops", &props)) {
                        VERIFY(0 == nvlist_lookup_nvlist(props,
                            snapname, &snapprops_nvlist));
                }

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

        cp = NULL;

        /*
         * Determine how much of the snapshot name stored in the stream
         * we are going to tack on to the name they specified on the
         * command line, and how much we are going to chop off.
         *
         * If they specified a snapshot, chop the entire name stored in
         * the stream.
         */
        if (flags->istail) {
                /*
                 * A filesystem was specified with -e. We want to tack on only
                 * the tail of the sent snapshot path.
                 */
                if (strchr(tosnap, '@')) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
                            "argument - snapshot not allowed with -e"));
                        return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
                }

                chopprefix = strrchr(sendfs, '/');

                if (chopprefix == NULL) {
                        /*
                         * The tail is the poolname, so we need to
                         * prepend a path separator.
                         */
                        int len = strlen(drrb->drr_toname);
                        cp = malloc(len + 2);
                        cp[0] = '/';
                        (void) strcpy(&cp[1], drrb->drr_toname);
                        chopprefix = cp;
                } else {
                        chopprefix = drrb->drr_toname + (chopprefix - sendfs);
                }
        } else if (flags->isprefix) {
                /*
                 * A filesystem was specified with -d. We want to tack on
                 * everything but the first element of the sent snapshot path
                 * (all but the pool name).
                 */
                if (strchr(tosnap, '@')) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
                            "argument - snapshot not allowed with -d"));
                        return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
```

```
                }

                chopprefix = strchr(drrb->drr_toname, '/');
                if (chopprefix == NULL)
                        chopprefix = strchr(drrb->drr_toname, '@');
        } else if (strchr(tosnap, '@') == NULL) {
                /*
                 * If a filesystem was specified without -d or -e, we want to
                 * tack on everything after the fs specified by 'zfs send'.
                 */
                chopprefix = drrb->drr_toname + strlen(sendfs);
        } else {
                /* A snapshot was specified as an exact path (no -d or -e). */
                if (recursive) {
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "cannot specify snapshot name for multi-snapshot "
                            "stream"));
                        return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
                }
                chopprefix = drrb->drr_toname + strlen(drrb->drr_toname);
        }

        ASSERT(strstr(drrb->drr_toname, sendfs) == drrb->drr_toname);
        ASSERT(chopprefix > drrb->drr_toname);
        ASSERT(chopprefix <= drrb->drr_toname + strlen(drrb->drr_toname));
        ASSERT(chopprefix[0] == '/' || chopprefix[0] == '@' ||
            chopprefix[0] == '\0');

        /*
         * Determine name of destination snapshot, store in zc_value.
         */
        (void) strcpy(zc.zc_value, tosnap);
        (void) strncat(zc.zc_value, chopprefix, sizeof (zc.zc_value));
        free(cp);
        if (!zfs_name_valid(zc.zc_value, ZFS_TYPE_SNAPSHOT)) {
                zcmd_free_nvlists(&zc);
                return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
        }

        /*
         * Determine the name of the origin snapshot, store in zc_string.
         */
        if (drrb->drr_flags & DRR_FLAG_CLONE) {
                if (guid_to_name(hdl, zc.zc_value,
                    drrb->drr_fromguid, zc.zc_string) != 0) {
                        zcmd_free_nvlists(&zc);
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "local origin for clone %s does not exist"),
                            zc.zc_value);
                        return (zfs_error(hdl, EZFS_NOENT, errbuf));
                }
                if (flags->verbose)
                        (void) printf("found clone origin %s\n", zc.zc_string);
        }

        stream_wantsnewfs = (drrb->drr_fromguid == NULL ||
            (drrb->drr_flags & DRR_FLAG_CLONE));

        if (stream_wantsnewfs) {
                /*
                 * if the parent fs does not exist, look for it based on
                 * the parent snap GUID
                 */
                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "cannot receive new filesystem stream"));
                (void) strcpy(zc.zc_name, zc.zc_value);
                cp = strrchr(zc.zc_name, '/');
                if (cp)
                        *cp = '\0';
                if (cp &&
                    !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
                        char suffix[ZFS_MAXNAMELEN];
                        (void) strcpy(suffix, strrchr(zc.zc_value, '/'));
                        if (guid_to_name(hdl, zc.zc_name, parent_snapguid,
                            zc.zc_value) == 0) {
                                *strchr(zc.zc_value, '@') = '\0';
                                (void) strcat(zc.zc_value, suffix);
                        }
                }
        } else {
                /*
                 * if the fs does not exist, look for it based on the
                 * fromsnap GUID
                 */
                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "cannot receive incremental stream"));

                (void) strcpy(zc.zc_name, zc.zc_value);
                *strchr(zc.zc_name, '@') = '\0';

                /*
                 * If the exact receive path was specified and this is the
                 * topmost path in the stream, then if the fs does not exist we
                 * should look no further.
                 */
                if ((flags->isprefix || (*(chopprefix = drrb->drr_toname +
                    strlen(sendfs)) != '\0' && *chopprefix != '@')) &&
                    !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
                        char snap[ZFS_MAXNAMELEN];
                        (void) strcpy(snap, strchr(zc.zc_value, '@'));
                        if (guid_to_name(hdl, zc.zc_name, drrb->drr_fromguid,
                            zc.zc_value) == 0) {
                                *strchr(zc.zc_value, '@') = '\0';
                                (void) strcat(zc.zc_value, snap);
                        }
                }
        }

        (void) strcpy(zc.zc_name, zc.zc_value);
        *strchr(zc.zc_name, '@') = '\0';

        if (zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
                zfs_handle_t *zhp;

                /*
                 * Destination fs exists.  Therefore this should either
                 * be an incremental, or the stream specifies a new fs
                 * (full stream or clone) and they want us to blow it
                 * away (and have therefore specified -F and removed any
                 * snapshots).
                 */
                if (stream_wantsnewfs) {
                        if (!flags->force) {
                                zcmd_free_nvlists(&zc);
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "destination '%s' exists\n"
                                    "must specify -F to overwrite it"),
                                    zc.zc_name);
                                return (zfs_error(hdl, EZFS_EXISTS, errbuf));
                        }
                        if (ioctl(hdl->libzfs_fd, ZFS_IOC_SNAPSHOT_LIST_NEXT,
```

```
                            &zc) == 0) {
                                zcmd_free_nvlists(&zc);
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "destination has snapshots (eg. %s)\n"
                                    "must destroy them to overwrite it"),
                                    zc.zc_name);
                                return (zfs_error(hdl, EZFS_EXISTS, errbuf));
                        }
                }

                if ((zhp = zfs_open(hdl, zc.zc_name,
                    ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME)) == NULL) {
                        zcmd_free_nvlists(&zc);
                        return (-1);
                }

                if (stream_wantsnewfs &&
                    zhp->zfs_dmustats.dds_origin[0]) {
                        zcmd_free_nvlists(&zc);
                        zfs_close(zhp);
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "destination '%s' is a clone\n"
                            "must destroy it to overwrite it"),
                            zc.zc_name);
                        return (zfs_error(hdl, EZFS_EXISTS, errbuf));
                }

                if (!flags->dryrun && zhp->zfs_type == ZFS_TYPE_FILESYSTEM &&
                    stream_wantsnewfs) {
                        /* We can't do online recv in this case */
                        clp = changelist_gather(zhp, ZFS_PROP_NAME, 0, 0);
                        if (clp == NULL) {
                                zfs_close(zhp);
                                zcmd_free_nvlists(&zc);
                                return (-1);
                        }
                        if (changelist_prefix(clp) != 0) {
                                changelist_free(clp);
                                zfs_close(zhp);
                                zcmd_free_nvlists(&zc);
                                return (-1);
                        }
                }
                zfs_close(zhp);
        } else {
                /*
                 * Destination filesystem does not exist.  Therefore we better
                 * be creating a new filesystem (either from a full backup, or
                 * a clone).  It would therefore be invalid if the user
                 * specified only the pool name (i.e. if the destination name
                 * contained no slash character).
                 */
                if (!stream_wantsnewfs ||
                    (cp = strrchr(zc.zc_name, '/')) == NULL) {
                        zcmd_free_nvlists(&zc);
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "destination '%s' does not exist"), zc.zc_name);
                        return (zfs_error(hdl, EZFS_NOENT, errbuf));
                }

                /*
                 * Trim off the final dataset component so we perform the
                 * recvbackup ioctl to the filesystems's parent.
                 */
                *cp = '\0';
                if (flags->isprefix && !flags->istail && !flags->dryrun &&
                    create_parents(hdl, zc.zc_value, strlen(tosnap)) != 0) {
                        zcmd_free_nvlists(&zc);
                        return (zfs_error(hdl, EZFS_BADRESTORE, errbuf));
                }

                newfs = B_TRUE;
        }

        zc.zc_begin_record = drr_noswap->drr_u.drr_begin;
        zc.zc_cookie = infd;
        zc.zc_guid = flags->force;
        if (flags->verbose) {
                (void) printf("%s %s stream of %s into %s\n",
                    flags->dryrun ? "would receive" : "receiving",
                    drrb->drr_fromguid ? "incremental" : "full",
                    drrb->drr_toname, zc.zc_value);
                (void) fflush(stdout);
        }

        if (flags->dryrun) {
                zcmd_free_nvlists(&zc);
                return (recv_skip(hdl, infd, flags->byteswap));
        }

        zc.zc_nvlist_dst = (uint64_t)(uintptr_t)prop_errbuf;
        zc.zc_nvlist_dst_size = sizeof (prop_errbuf);
        zc.zc_cleanup_fd = cleanup_fd;
        zc.zc_action_handle = *action_handlep;

        err = ioctl_err = zfs_ioctl(hdl, ZFS_IOC_RECV, &zc);
        ioctl_errno = errno;
        prop_errflags = (zprop_errflags_t)zc.zc_obj;

        if (err == 0) {
                nvlist_t *prop_errors;
                VERIFY(0 == nvlist_unpack((void *)(uintptr_t)zc.zc_nvlist_dst,
                    zc.zc_nvlist_dst_size, &prop_errors, 0));

                nvpair_t *prop_err = NULL;

                while ((prop_err = nvlist_next_nvpair(prop_errors,
                    prop_err)) != NULL) {
                        char tbuf[1024];
                        zfs_prop_t prop;
                        int intval;

                        prop = zfs_name_to_prop(nvpair_name(prop_err));
                        (void) nvpair_value_int32(prop_err, &intval);
                        if (strcmp(nvpair_name(prop_err),
                            ZPROP_N_MORE_ERRORS) == 0) {
                                trunc_prop_errs(intval);
                                break;
                        } else {
                                (void) snprintf(tbuf, sizeof (tbuf),
                                    dgettext(TEXT_DOMAIN,
                                    "cannot receive %s property on %s"),
                                    nvpair_name(prop_err), zc.zc_name);
                                zfs_setprop_error(hdl, prop, intval, tbuf);
                        }
                }
                nvlist_free(prop_errors);
        }

        zc.zc_nvlist_dst = 0;
        zc.zc_nvlist_dst_size = 0;
```

```
        zcmd_free_nvlists(&zc);

        if (err == 0 && snapprops_nvlist) {
                zfs_cmd_t zc2 = { 0 };

                (void) strcpy(zc2.zc_name, zc.zc_value);
                zc2.zc_cookie = B_TRUE; /* received */
                if (zcmd_write_src_nvlist(hdl, &zc2, snapprops_nvlist) == 0) {
                        (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc2);
                        zcmd_free_nvlists(&zc2);
                }
        }

        if (err && (ioctl_errno == ENOENT || ioctl_errno == EEXIST)) {
                /*
                 * It may be that this snapshot already exists,
                 * in which case we want to consume & ignore it
                 * rather than failing.
                 */
                avl_tree_t *local_avl;
                nvlist_t *local_nv, *fs;
                cp = strchr(zc.zc_value, ’@’);

                /*
                 * XXX Do this faster by just iterating over snaps in
                 * this fs.  Also if zc_value does not exist, we will
                 * get a strange "does not exist" error message.
                 */
                *cp = ’\0’;
                if (gather_nvlist(hdl, zc.zc_value, NULL, NULL, B_FALSE,
                    &local_nv, &local_avl) == 0) {
                        *cp = ’@’;
                        fs = fsavl_find(local_avl, drrb->drr_toguid, NULL);
                        fsavl_destroy(local_avl);
                        nvlist_free(local_nv);

                        if (fs != NULL) {
                                if (flags->verbose) {
                                        (void) printf("snap %s already exists; "
                                            "ignoring\n", zc.zc_value);
                                }
                                err = ioctl_err = recv_skip(hdl, infd,
                                    flags->byteswap);
                        }
                }
                *cp = ’@’;
        }

        if (ioctl_err != 0) {
                switch (ioctl_errno) {
                case ENODEV:
                        cp = strchr(zc.zc_value, ’@’);
                        *cp = ’\0’;
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "most recent snapshot of %s does not\n"
                            "match incremental source"), zc.zc_value);
                        (void) zfs_error(hdl, EZFS_BADRESTORE, errbuf);
                        *cp = ’@’;
                        break;
                case ETXTBSY:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "destination %s has been modified\n"
                            "since most recent snapshot"), zc.zc_name);
                        (void) zfs_error(hdl, EZFS_BADRESTORE, errbuf);
                        break;
                case EEXIST:
                        cp = strchr(zc.zc_value, ’@’);
                        if (newfs) {
                                /* it’s the containing fs that exists */
                                *cp = ’\0’;
                        }
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "destination already exists"));
                        (void) zfs_error_fmt(hdl, EZFS_EXISTS,
                            dgettext(TEXT_DOMAIN, "cannot restore to %s"),
                            zc.zc_value);
                        *cp = ’@’;
                        break;
                case EINVAL:
                        (void) zfs_error(hdl, EZFS_BADSTREAM, errbuf);
                        break;
                case ECKSUM:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "invalid stream (checksum mismatch)"));
                        (void) zfs_error(hdl, EZFS_BADSTREAM, errbuf);
                        break;
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded to receive this stream."));
                        (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EDQUOT:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "destination %s space quota exceeded"), zc.zc_name);
                        (void) zfs_error(hdl, EZFS_NOSPC, errbuf);
                        break;
                default:
                        (void) zfs_standard_error(hdl, ioctl_errno, errbuf);
                }
        }

        /*
         * Mount the target filesystem (if created).  Also mount any
         * children of the target filesystem if we did a replication
         * receive (indicated by stream_avl being non-NULL).
         */
        cp = strchr(zc.zc_value, ’@’);
        if (cp && (ioctl_err == 0 || !newfs)) {
                zfs_handle_t *h;

                *cp = ’\0’;
                h = zfs_open(hdl, zc.zc_value,
                    ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
                if (h != NULL) {
                        if (h->zfs_type == ZFS_TYPE_VOLUME) {
                                *cp = ’@’;
                        } else if (newfs || stream_avl) {
                                /*
                                 * Track the first/top of hierarchy fs,
                                 * for mounting and sharing later.
                                 */
                                if (top_zfs && *top_zfs == NULL)
                                        *top_zfs = zfs_strdup(hdl, zc.zc_value);
                        }
                        zfs_close(h);
                }
                *cp = ’@’;
        }

        if (clp) {
                err |= changelist_postfix(clp);
                changelist_free(clp);
```

```
        }

        if (prop_errflags & ZPROP_ERR_NOCLEAR) {
                (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "Warning: "
                    "failed to clear unreceived properties on %s"),
                    zc.zc_name);
                (void) fprintf(stderr, "\n");
        }
        if (prop_errflags & ZPROP_ERR_NORESTORE) {
                (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "Warning: "
                    "failed to restore original properties on %s"),
                    zc.zc_name);
                (void) fprintf(stderr, "\n");
        }

        if (err || ioctl_err)
                return (-1);

        *action_handlep = zc.zc_action_handle;

        if (flags->verbose) {
                char buf1[64];
                char buf2[64];
                uint64_t bytes = zc.zc_cookie;
                time_t delta = time(NULL) - begin_time;
                if (delta == 0)
                        delta = 1;
                zfs_nicenum(bytes, buf1, sizeof (buf1));
                zfs_nicenum(bytes/delta, buf2, sizeof (buf1));

                (void) printf("received %sB stream in %lu seconds (%sB/sec)\n",
                    buf1, delta, buf2);
        }

        return (0);
}

static int
zfs_receive_impl(libzfs_handle_t *hdl, const char *tosnap, recvflags_t *flags,
    int infd, const char *sendfs, nvlist_t *stream_nv, avl_tree_t *stream_avl,
    char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
{
        int err;
        dmu_replay_record_t drr, drr_noswap;
        struct drr_begin *drrb = &drr.drr_u.drr_begin;
        char errbuf[1024];
        zio_cksum_t zcksum = { 0 };
        uint64_t featureflags;
        int hdrtype;

        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
            "cannot receive"));

        if (flags->isprefix &&
            !zfs_dataset_exists(hdl, tosnap, ZFS_TYPE_DATASET)) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "specified fs "
                    "(%s) does not exist"), tosnap);
                return (zfs_error(hdl, EZFS_NOENT, errbuf));
        }

        /* read in the BEGIN record */
        if (0 != (err = recv_read(hdl, infd, &drr, sizeof (drr), B_FALSE,
            &zcksum)))
                return (err);

        if (drr.drr_type == DRR_END || drr.drr_type == BSWAP_32(DRR_END)) {
                /* It’s the double end record at the end of a package */
                return (ENODATA);
        }

        /* the kernel needs the non-byteswapped begin record */
        drr_noswap = drr;

        flags->byteswap = B_FALSE;
        if (drrb->drr_magic == BSWAP_64(DMU_BACKUP_MAGIC)) {
                /*
                 * We computed the checksum in the wrong byteorder in
                 * recv_read() above; do it again correctly.
                 */
                bzero(&zcksum, sizeof (zio_cksum_t));
                fletcher_4_incremental_byteswap(&drr, sizeof (drr), &zcksum);
                flags->byteswap = B_TRUE;

                drr.drr_type = BSWAP_32(drr.drr_type);
                drr.drr_payloadlen = BSWAP_32(drr.drr_payloadlen);
                drrb->drr_magic = BSWAP_64(drrb->drr_magic);
                drrb->drr_versioninfo = BSWAP_64(drrb->drr_versioninfo);
                drrb->drr_creation_time = BSWAP_64(drrb->drr_creation_time);
                drrb->drr_type = BSWAP_32(drrb->drr_type);
                drrb->drr_flags = BSWAP_32(drrb->drr_flags);
                drrb->drr_toguid = BSWAP_64(drrb->drr_toguid);
                drrb->drr_fromguid = BSWAP_64(drrb->drr_fromguid);
        }

        if (drrb->drr_magic != DMU_BACKUP_MAGIC || drr.drr_type != DRR_BEGIN) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
                    "stream (bad magic number)"));
                return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
        }

        featureflags = DMU_GET_FEATUREFLAGS(drrb->drr_versioninfo);
        hdrtype = DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo);

        if (!DMU_STREAM_SUPPORTED(featureflags) ||
            (hdrtype != DMU_SUBSTREAM && hdrtype != DMU_COMPOUNDSTREAM)) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                    "stream has unsupported feature, feature flags = %lx"),
                    featureflags);
                return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
        }

        if (strchr(drrb->drr_toname, ’@’) == NULL) {
                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
                    "stream (bad snapshot name)"));
                return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
        }

        if (DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) == DMU_SUBSTREAM) {
                char nonpackage_sendfs[ZFS_MAXNAMELEN];
                if (sendfs == NULL) {
                        /*
                         * We were not called from zfs_receive_package(). Get
                         * the fs specified by ’zfs send’.
                         */
                        char *cp;
                        (void) strlcpy(nonpackage_sendfs,
                            drr.drr_u.drr_begin.drr_toname, ZFS_MAXNAMELEN);
                        if ((cp = strchr(nonpackage_sendfs, ’@’)) != NULL)
                                *cp = ’\0’;
                        sendfs = nonpackage_sendfs;
                }
                return (zfs_receive_one(hdl, infd, tosnap, flags,
```

```
                    &drr, &drr_noswap, sendfs, stream_nv, stream_avl,
                    top_zfs, cleanup_fd, action_handlep));
        } else {
                assert(DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) ==
                    DMU_COMPOUNDSTREAM);
                return (zfs_receive_package(hdl, infd, tosnap, flags,
                    &drr, &zcksum, top_zfs, cleanup_fd, action_handlep));
        }
}

/*
 * Restores a backup of tosnap from the file descriptor specified by infd.
 * Return 0 on total success, -2 if some things couldn’t be
 * destroyed/renamed/promoted, -1 if some things couldn’t be received.
 * (-1 will override -2).
 */
int
zfs_receive(libzfs_handle_t *hdl, const char *tosnap, recvflags_t *flags,
    int infd, avl_tree_t *stream_avl)
{
        char *top_zfs = NULL;
        int err;
        int cleanup_fd;
        uint64_t action_handle = 0;

        cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
        VERIFY(cleanup_fd >= 0);

        err = zfs_receive_impl(hdl, tosnap, flags, infd, NULL, NULL,
            stream_avl, &top_zfs, cleanup_fd, &action_handle);

        VERIFY(0 == close(cleanup_fd));

        if (err == 0 && !flags->nomount && top_zfs) {
                zfs_handle_t *zhp;
                prop_changelist_t *clp;

                zhp = zfs_open(hdl, top_zfs, ZFS_TYPE_FILESYSTEM);
                if (zhp != NULL) {
                        clp = changelist_gather(zhp, ZFS_PROP_MOUNTPOINT,
                            CL_GATHER_MOUNT_ALWAYS, 0);
                        zfs_close(zhp);
                        if (clp != NULL) {
                                /* mount and share received datasets */
                                err = changelist_postfix(clp);
                                changelist_free(clp);
                        }
                }
                if (zhp == NULL || clp == NULL || err)
                        err = -1;
        }
        if (top_zfs)
                free(top_zfs);

        return (err);
}
```

```
**********************************************************
   29987 Thu Apr 25 12:27:56 2013
new/usr/src/uts/common/fs/zfs/dsl_pool.c
Optimize creation and removal of temporary "user holds" placed on
snapshots by a zfs send, by ensuring all the required holds and
releases are done in a single dsl_sync_task.
Creation now collates the required holds during a dry run and
then uses a single lzc_hold call via zfs_hold_apply instead of
processing each snapshot in turn.
Defered (on exit) cleanup by the kernel is also now done in
dsl_sync_task by reusing dsl_dataset_user_release.
On a test with 11 volumes in a tree each with 8 snapshots on a
single HDD zpool this reduces the time required to perform a full
send from 20 seconds to under 0.8 seconds.
For reference eliminating the hold entirely reduces this 0.15
seconds.
While I'm here:-
* Remove some unused structures
* Fix nvlist_t leak in zfs_release_one
**********************************************************
______unchanged_portion_omitted_
```

```
/*
 * Walk through the pool-wide zap object of temporary snapshot user holds
 * and release them.
 */
void
dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp)
{
        char *htag;
#endif /* ! codereview */
        zap_attribute_t za;
        zap_cursor_t zc;
        objset_t *mos = dp->dp_meta_objset;
        uint64_t zapobj = dp->dp_tmp_userrefs_obj;
        uint64_t dsobj;
        nvlist_t *holds, *tags;
        dsl_dataset_t *ds;
        char name[MAXNAMELEN];
#endif /* ! codereview */

        if (zapobj == 0)
                return;
        ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);

        holds = fnvlist_alloc();

        dsl_pool_config_enter(dp, FTAG);
#endif /* ! codereview */
        for (zap_cursor_init(&zc, mos, zapobj);
            zap_cursor_retrieve(&zc, &za) == 0;
            zap_cursor_advance(&zc)) {
                char *htag;
                uint64_t dsobj;

                htag = strchr(za.za_name, '-');
                *htag = '\0';
                ++htag;
                dsobj = strtonum(za.za_name, NULL);
                if (dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds) == 0) {
                        dsl_dataset_name(ds, name);
                        if (nvlist_lookup_nvlist(holds, name, &tags) != 0) {
                                tags = fnvlist_alloc();
                                fnvlist_add_boolean(tags, htag);
                                fnvlist_add_nvlist(holds, name, tags);
                                fnvlist_free(tags);
```

```
                        } else {
                                fnvlist_add_boolean(tags, htag);
                        }
                        dsl_dataset_rele(ds, FTAG);
                }
                dsl_dataset_user_release_tmp(dp, dsobj, htag);
        }
        dsl_pool_config_exit(dp, FTAG);
        dsl_dataset_user_release(holds, NULL);
        fnvlist_free(holds);
#endif /* ! codereview */
        zap_cursor_fini(&zc);
}

/*
 * Create the pool-wide zap object for storing temporary snapshot holds.
 */
void
dsl_pool_user_hold_create_obj(dsl_pool_t *dp, dmu_tx_t *tx)
{
        objset_t *mos = dp->dp_meta_objset;

        ASSERT(dp->dp_tmp_userrefs_obj == 0);
        ASSERT(dmu_tx_is_syncing(tx));

        dp->dp_tmp_userrefs_obj = zap_create_link(mos, DMU_OT_USERREFS,
            DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_TMP_USERREFS, tx);
}

static int
dsl_pool_user_hold_rele_impl(dsl_pool_t *dp, uint64_t dsobj,
    const char *tag, uint64_t now, dmu_tx_t *tx, boolean_t holding)
{
        objset_t *mos = dp->dp_meta_objset;
        uint64_t zapobj = dp->dp_tmp_userrefs_obj;
        char *name;
        int error;

        ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);
        ASSERT(dmu_tx_is_syncing(tx));

        /*
         * If the pool was created prior to SPA_VERSION_USERREFS, the
         * zap object for temporary holds might not exist yet.
         */
        if (zapobj == 0) {
                if (holding) {
                        dsl_pool_user_hold_create_obj(dp, tx);
                        zapobj = dp->dp_tmp_userrefs_obj;
                } else {
                        return (SET_ERROR(ENOENT));
                }
        }

        name = kmem_asprintf("%llx-%s", (u_longlong_t)dsobj, tag);
        if (holding)
                error = zap_add(mos, zapobj, name, 8, 1, &now, tx);
        else
                error = zap_remove(mos, zapobj, name, tx);
        strfree(name);

        return (error);
}

/*
 * Add a temporary hold for the given dataset object and tag.
 */
int
dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj, const char *tag,
    uint64_t now, dmu_tx_t *tx)
{
        return (dsl_pool_user_hold_rele_impl(dp, dsobj, tag, now, tx, B_TRUE));
}

/*
 * Release a temporary hold for the given dataset object and tag.
 */
int
dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj, const char *tag,
    dmu_tx_t *tx)
{
        return (dsl_pool_user_hold_rele_impl(dp, dsobj, tag, NULL,
            tx, B_FALSE));
}

/*
 * DSL Pool Configuration Lock
 *
 * The dp_config_rwlock protects against changes to DSL state (e.g. dataset
 * creation / destruction / rename / property setting).  It must be held for
 * read to hold a dataset or dsl_dir.  I.e. you must call
 * dsl_pool_config_enter() or dsl_pool_hold() before calling
 * dsl_{dataset,dir}_hold{_obj}.  In most circumstances, the dp_config_rwlock
 * must be held continuously until all datasets and dsl_dirs are released.
 *
 * The only exception to this rule is that if a "long hold" is placed on
 * a dataset, then the dp_config_rwlock may be dropped while the dataset
 * is still held.  The long hold will prevent the dataset from being
 * destroyed -- the destroy will fail with EBUSY.  A long hold can be
 * obtained by calling dsl_dataset_long_hold(), or by "owning" a dataset
 * (by calling dsl_{dataset,objset}_{try}own{_obj}).
 *
 * Legitimate long-holders (including owners) should be long-running, cancelable
 * tasks that should cause "zfs destroy" to fail.  This includes DMU
 * consumers (i.e. a ZPL filesystem being mounted or ZVOL being open),
 * "zfs send", and "zfs diff".  There are several other long-holders whose
 * uses are suboptimal (e.g. "zfs promote", and zil_suspend()).
 *
 * The usual formula for long-holding would be:
 * dsl_pool_hold()
 * dsl_dataset_hold()
 * ... perform checks ...
 * dsl_dataset_long_hold()
 * dsl_pool_rele()
 * ... perform long-running task ...
 * dsl_dataset_long_rele()
 * dsl_dataset_rele()
 *
 * Note that when the long hold is released, the dataset is still held but
 * the pool is not held.  The dataset may change arbitrarily during this time
 * (e.g. it could be destroyed).  Therefore you shouldn't do anything to the
 * dataset except release it.
 *
 * User-initiated operations (e.g. ioctls, zfs_ioc_*()) are either read-only
 * or modifying operations.
 *
 * Modifying operations should generally use dsl_sync_task().  The synctask
 * infrastructure enforces proper locking strategy with respect to the
 * dp_config_rwlock.  See the comment above dsl_sync_task() for details.
 *
 * Read-only operations will manually hold the pool, then the dataset, obtain
 * information from the dataset, then release the pool and dataset.
```

```
 * dmu_objset_{hold,rele}() are convenience routines that also do the pool
 * hold/rele.
 */

int
dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp)
{
        spa_t *spa;
        int error;

        error = spa_open(name, &spa, tag);
        if (error == 0) {
                *dp = spa_get_dsl(spa);
                dsl_pool_config_enter(*dp, tag);
        }
        return (error);
}

void
dsl_pool_rele(dsl_pool_t *dp, void *tag)
{
        dsl_pool_config_exit(dp, tag);
        spa_close(dp->dp_spa, tag);
}

void
dsl_pool_config_enter(dsl_pool_t *dp, void *tag)
{
        /*
         * We use a "reentrant" reader-writer lock, but not reentrantly.
         *
         * The rrwlock can (with the track_all flag) track all reading threads,
         * which is very useful for debugging which code path failed to release
         * the lock, and for verifying that the *current* thread does hold
         * the lock.
         *
         * (Unlike a rwlock, which knows that N threads hold it for
         * read, but not *which* threads, so rw_held(RW_READER) returns TRUE
         * if any thread holds it for read, even if this thread doesn't).
         */
        ASSERT(!rrw_held(&dp->dp_config_rwlock, RW_READER));
        rrw_enter(&dp->dp_config_rwlock, RW_READER, tag);
}

void
dsl_pool_config_exit(dsl_pool_t *dp, void *tag)
{
        rrw_exit(&dp->dp_config_rwlock, tag);
}

boolean_t
dsl_pool_config_held(dsl_pool_t *dp)
{
        return (RRW_LOCK_HELD(&dp->dp_config_rwlock));
}
```

```
**********************************************************
   11744 Thu Apr 25 12:27:56 2013
new/usr/src/uts/common/fs/zfs/dsl_userhold.c
Optimize creation and removal of temporary "user holds" placed on
snapshots by a zfs send, by ensuring all the required holds and
releases are done in a single dsl_sync_task.
Creation now collates the required holds during a dry run and
then uses a single lzc_hold call via zfs_hold_apply instead of
processing each snapshot in turn.
Defered (on exit) cleanup by the kernel is also now done in
dsl_sync_task by reusing dsl_dataset_user_release.
On a test with 11 volumes in a tree each with 8 snapshots on a
single HDD zpool this reduces the time required to perform a full
send from 20 seconds to under 0.8 seconds.
For reference eliminating the hold entirely reduces this 0.15
seconds.
While I'm here:-
* Remove some unused structures
* Fix nvlist_t leak in zfs_release_one
**********************************************************
______unchanged_portion_omitted_
```

```
void
dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
    minor_t minor, uint64_t now, dmu_tx_t *tx)
{
        dsl_pool_t *dp = ds->ds_dir->dd_pool;
        objset_t *mos = dp->dp_meta_objset;
        uint64_t zapobj;

        mutex_enter(&ds->ds_lock);
        if (ds->ds_phys->ds_userrefs_obj == 0) {
                /*
                 * This is the first user hold for this dataset.  Create
                 * the userrefs zap object.
                 */
                dmu_buf_will_dirty(ds->ds_dbuf, tx);
                zapobj = ds->ds_phys->ds_userrefs_obj =
                    zap_create(mos, DMU_OT_USERREFS, DMU_OT_NONE, 0, tx);
        } else {
                zapobj = ds->ds_phys->ds_userrefs_obj;
        }
        ds->ds_userrefs++;
        mutex_exit(&ds->ds_lock);

        VERIFY0(zap_add(mos, zapobj, htag, 8, 1, &now, tx));

        if (minor != 0) {
                VERIFY0(dsl_pool_user_hold(dp, ds->ds_object,
                    htag, now, tx));
                dsl_register_onexit_hold_cleanup(ds, htag, minor);
        }

        spa_history_log_internal_ds(ds, "hold", tx,
            "tag=%s temp=%d refs=%llu",
            htag, minor != 0, ds->ds_userrefs);
}

static void
dsl_dataset_user_hold_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_user_hold_arg_t *dduha = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        nvpair_t *pair;
        uint64_t now = gethrestime_sec();
```

```
        for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;
            pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {
                dsl_dataset_t *ds;

#endif /* ! codereview */
                VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
                dsl_dataset_user_hold_sync_one(ds, fnvpair_value_string(pair),
                    dduha->dduha_minor, now, tx);
                dsl_dataset_rele(ds, FTAG);
        }
}

/*
 * holds is nvl of snapname -> holdname
 * errlist will be filled in with snapname -> error
 * if cleanup_minor is not 0, the holds will be temporary, cleaned up
 * when the process exits.
 *
 * if any fails, all will fail.
 */
int
dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor, nvlist_t *errlist)
{
        dsl_dataset_user_hold_arg_t dduha;
        nvpair_t *pair;
        int ret;
#endif /* ! codereview */

        pair = nvlist_next_nvpair(holds, NULL);
        if (pair == NULL)
                return (0);

        dduha.dduha_holds = holds;
        dduha.dduha_errlist = errlist;
        dduha.dduha_minor = cleanup_minor;

        ret = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
            dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds));
        if (ret == 0)
                dsl_register_onexit_hold_cleanup(holds, cleanup_minor);

        return (ret);
        return (dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
            dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds)));
}
______unchanged_portion_omitted_

/*
 * holds is nvl of snapname -> { holdname, ... }
 * errlist will be filled in with snapname -> error
 *
 * if any fails, all will fail.
 */
int
dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist)
{
        dsl_dataset_user_release_arg_t ddura;
        nvpair_t *pair, *pair2;
        nvpair_t *pair;
        int error;

        pair = nvlist_next_nvpair(holds, NULL);
        if (pair == NULL)
                return (0);

#ifdef _KERNEL
```

```
        /*
         * The release may cause the snapshot to be destroyed; make sure it
         * is not mounted.
         */
        for (pair2 = pair; pair2 != NULL;
            pair2 = nvlist_next_nvpair(holds, pair2)) {
                zfs_unmount_snap(nvpair_name(pair2));
        }
#endif

#endif /* ! codereview */
        ddura.ddura_holds = holds;
        ddura.ddura_errlist = errlist;
        ddura.ddura_todelete = fnvlist_alloc();

        error = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_release_check,
            dsl_dataset_user_release_sync, &ddura, fnvlist_num_pairs(holds));
        fnvlist_free(ddura.ddura_todelete);
        return (error);
}

static void
dsl_dataset_user_release_onexit(void *arg)
typedef struct dsl_dataset_user_release_tmp_arg {
        uint64_t ddurta_dsobj;
        nvlist_t *ddurta_holds;
        boolean_t ddurta_deleteme;
} dsl_dataset_user_release_tmp_arg_t;

static int
dsl_dataset_user_release_tmp_check(void *arg, dmu_tx_t *tx)
{
        nvlist_t *holds = arg;
        dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;
        int error;

        (void) dsl_dataset_user_release(holds, NULL);
        fnvlist_free(holds);
        if (!dmu_tx_is_syncing(tx))
                return (0);

        error = dsl_dataset_hold_obj(dp, ddurta->ddurta_dsobj, FTAG, &ds);
        if (error)
                return (error);

        error = dsl_dataset_user_release_check_one(ds,
            ddurta->ddurta_holds, &ddurta->ddurta_deleteme);
        dsl_dataset_rele(ds, FTAG);
        return (error);
}

void
dsl_register_onexit_hold_cleanup(nvlist_t *holds, minor_t minor)
static void
dsl_dataset_user_release_tmp_sync(void *arg, dmu_tx_t *tx)
{
        nvlist_t *ca;
        nvpair_t *pair;
        char *htag;
        dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;

        VERIFY0(dsl_dataset_hold_obj(dp, ddurta->ddurta_dsobj, FTAG, &ds));
```

```
        dsl_dataset_user_release_sync_one(ds, ddurta->ddurta_holds, tx);
        if (ddurta->ddurta_deleteme) {
                ASSERT(ds->ds_userrefs == 0 &&
                    ds->ds_phys->ds_num_children == 1 &&
                    DS_IS_DEFER_DESTROY(ds));
                dsl_destroy_snapshot_sync_impl(ds, B_FALSE, tx);
        }
        dsl_dataset_rele(ds, FTAG);
}

        ca = fnvlist_alloc();
        /*
         * Convert from hold format: nvl of snapname -> holdname
         * to release format: nvl of snapname -> { holdname, ... }
/*
 * Called at spa_load time to release a stale temporary user hold.
 * Also called by the onexit code.
         */
        for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
            pair = nvlist_next_nvpair(holds, pair)) {
                if (nvpair_value_string(pair, &htag) == 0) {
                        nvlist_t *tags;
void
dsl_dataset_user_release_tmp(dsl_pool_t *dp, uint64_t dsobj, const char *htag)
{
        dsl_dataset_user_release_tmp_arg_t ddurta;
        dsl_dataset_t *ds;
        int error;

                        tags = fnvlist_alloc();
                        fnvlist_add_boolean(tags, htag);
                        fnvlist_add_nvlist(ca, nvpair_name(pair), tags);
                        fnvlist_free(tags);
                }
#ifdef _KERNEL
        /* Make sure it is not mounted. */
        dsl_pool_config_enter(dp, FTAG);
        error = dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds);
        if (error == 0) {
                char name[MAXNAMELEN];
                dsl_dataset_name(ds, name);
                dsl_dataset_rele(ds, FTAG);
                dsl_pool_config_exit(dp, FTAG);
                zfs_unmount_snap(name);
        } else {
                dsl_pool_config_exit(dp, FTAG);
        }
#endif

        ddurta.ddurta_dsobj = dsobj;
        ddurta.ddurta_holds = fnvlist_alloc();
        fnvlist_add_boolean(ddurta.ddurta_holds, htag);

        (void) dsl_sync_task(spa_name(dp->dp_spa),
            dsl_dataset_user_release_tmp_check,
            dsl_dataset_user_release_tmp_sync, &ddurta, 1);
        fnvlist_free(ddurta.ddurta_holds);
}

typedef struct zfs_hold_cleanup_arg {
        char zhca_spaname[MAXNAMELEN];
        uint64_t zhca_spa_load_guid;
        uint64_t zhca_dsobj;
        char zhca_htag[MAXNAMELEN];
} zfs_hold_cleanup_arg_t;
```

```
static void
dsl_dataset_user_release_onexit(void *arg)
{
        zfs_hold_cleanup_arg_t *ca = arg;
        spa_t *spa;
        int error;

        error = spa_open(ca->zhca_spaname, &spa, FTAG);
        if (error != 0) {
                zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
                    "because pool is no longer loaded",
                    ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
                return;
        }
        if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
                zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
                    "because pool is no longer loaded (guid doesn't match)",
                    ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
                spa_close(spa, FTAG);
                return;
        }

        dsl_dataset_user_release_tmp(spa_get_dsl(spa),
            ca->zhca_dsobj, ca->zhca_htag);
        kmem_free(ca, sizeof (zfs_hold_cleanup_arg_t));
        spa_close(spa, FTAG);
}

void
dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
    minor_t minor)
{
        zfs_hold_cleanup_arg_t *ca = kmem_alloc(sizeof (*ca), KM_SLEEP);
        spa_t *spa = dsl_dataset_get_spa(ds);
        (void) strlcpy(ca->zhca_spaname, spa_name(spa),
            sizeof (ca->zhca_spaname));
        ca->zhca_spa_load_guid = spa_load_guid(spa);
        ca->zhca_dsobj = ds->ds_object;
        (void) strlcpy(ca->zhca_htag, htag, sizeof (ca->zhca_htag));
        VERIFY0(zfs_onexit_add_cb(minor,
            dsl_dataset_user_release_onexit, ca, NULL));
}
```

______**_unchanged_portion_omitted_**

```
**********************************************************
   10218 Thu Apr 25 12:27:56 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
Optimize creation and removal of temporary "user holds" placed on
snapshots by a zfs send, by ensuring all the required holds and
releases are done in a single dsl_sync_task.
Creation now collates the required holds during a dry run and
then uses a single lzc_hold call via zfs_hold_apply instead of
processing each snapshot in turn.
Defered (on exit) cleanup by the kernel is also now done in
dsl_sync_task by reusing dsl_dataset_user_release.
On a test with 11 volumes in a tree each with 8 snapshots on a
single HDD zpool this reduces the time required to perform a full
send from 20 seconds to under 0.8 seconds.
For reference eliminating the hold entirely reduces this 0.15
seconds.
While I'm here:-
* Remove some unused structures
* Fix nvlist_t leak in zfs_release_one
**********************************************************
______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);
void dsl_register_onexit_hold_cleanup(nvlist_t *holds, minor_t minor);
void dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
    minor_t minor);
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_lastsnap(dsl_dataset_t *ds);

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 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);
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);

#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
}
```

______unchanged_portion_omitted_

```
**********************************************************
    1753 Thu Apr 25 12:27:56 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_userhold.h
Optimize creation and removal of temporary "user holds" placed on
snapshots by a zfs send, by ensuring all the required holds and
releases are done in a single dsl_sync_task.
Creation now collates the required holds during a dry run and
then uses a single lzc_hold call via zfs_hold_apply instead of
processing each snapshot in turn.
Defered (on exit) cleanup by the kernel is also now done in
dsl_sync_task by reusing dsl_dataset_user_release.
On a test with 11 volumes in a tree each with 8 snapshots on a
single HDD zpool this reduces the time required to perform a full
send from 20 seconds to under 0.8 seconds.
For reference eliminating the hold entirely reduces this 0.15
seconds.
While I'm here:-
* Remove some unused structures
* Fix nvlist_t leak in zfs_release_one
**********************************************************
```

```
/*
 * 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) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */

#ifndef _SYS_DSL_USERHOLD_H
#define _SYS_DSL_USERHOLD_H

#include <sys/nvpair.h>
#include <sys/types.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct dsl_pool;
struct dsl_dataset;
struct dmu_tx;

int dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor,
    nvlist_t *errlist);
int dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist);
int dsl_dataset_get_holds(const char *dsname, nvlist_t *nvl);
void dsl_dataset_user_release_tmp(struct dsl_pool *dp, uint64_t dsobj,
```

```
    const char *htag);
int dsl_dataset_user_hold_check_one(struct dsl_dataset *ds, const char *htag,
    boolean_t temphold, struct dmu_tx *tx);
void dsl_dataset_user_hold_sync_one(struct dsl_dataset *ds, const char *htag,
    minor_t minor, uint64_t now, struct dmu_tx *tx);

#ifdef  __cplusplus
}
______unchanged_portion_omitted_
```

```
**********************************************************
  143884 Thu Apr 25 12:27:57 2013
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
Optimize creation and removal of temporary "user holds" placed on
snapshots by a zfs send, by ensuring all the required holds and
releases are done in a single dsl_sync_task.
Creation now collates the required holds during a dry run and
then uses a single lzc_hold call via zfs_hold_apply instead of
processing each snapshot in turn.
Defered (on exit) cleanup by the kernel is also now done in
dsl_sync_task by reusing dsl_dataset_user_release.
On a test with 11 volumes in a tree each with 8 snapshots on a
single HDD zpool this reduces the time required to perform a full
send from 20 seconds to under 0.8 seconds.
For reference eliminating the hold entirely reduces this 0.15
seconds.
While I’m here:-
* Remove some unused structures
* Fix nvlist_t leak in zfs_release_one
**********************************************************
______unchanged_portion_omitted_
```

```
/*
 * innvl: {
 *     snapname -> { holdname, ... }
 *     ...
 * }
 *
 * outnvl: {
 *     snapname -> error value (int32)
 *     ...
 * }
 */
/* ARGSUSED */
static int
zfs_ioc_release(const char *pool, nvlist_t *holds, nvlist_t *errlist)
{
        nvpair_t *pair;

        /*
         * The release may cause the snapshot to be destroyed; make sure it
         * is not mounted.
         */
        for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
            pair = nvlist_next_nvpair(holds, pair))
                zfs_unmount_snap(nvpair_name(pair));

        return (dsl_dataset_user_release(holds, errlist));
}
______unchanged_portion_omitted_
```