```
**********************************************************
   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_

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

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

 595 extern int zfs_promote(zfs_handle_t *);
 596 extern int zfs_hold(zfs_handle_t *, const char *, const char *,
 597     boolean_t, boolean_t, int);
 598 extern int zfs_hold_add(zfs_handle_t *, const char *, const char *,
 599     boolean_t, nvlist_t *);
 600 extern int zfs_hold_apply(zfs_handle_t *, boolean_t, int, nvlist_t *);
 601 #endif /* ! codereview */
 602 extern int zfs_release(zfs_handle_t *, const char *, const char *, boolean_t);
 603 extern int zfs_get_holds(zfs_handle_t *, nvlist_t **);
 604 extern uint64_t zvol_volsize_to_reservation(uint64_t, nvlist_t *);

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

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

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

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

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

 622         /*
 623          * Only the tail of the sent snapshot path is appended to the
 624          * destination to determine the received snapshot name (ie, -e).
 625          */
 626         boolean_t istail;

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

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



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

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

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

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

 647 typedef enum diff_flags {
 648         ZFS_DIFF_PARSEABLE = 0x1,
 649         ZFS_DIFF_TIMESTAMP = 0x2,
 650         ZFS_DIFF_CLASSIFY = 0x4
 651 } diff_flags_t;

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

 656 /*
 657  * Miscellaneous functions.
 658  */
 659 extern const char *zfs_type_to_name(zfs_type_t);
 660 extern void zfs_refresh_properties(zfs_handle_t *);
 661 extern int zfs_name_valid(const char *, zfs_type_t);
 662 extern zfs_handle_t *zfs_path_to_zhandle(libzfs_handle_t *, char *, zfs_type_t);
 663 extern boolean_t zfs_dataset_exists(libzfs_handle_t *, const char *,
 664     zfs_type_t);
 665 extern int zfs_spa_version(zfs_handle_t *, int *);

 667 /*
 668  * Mount support functions.
 669  */
 670 extern boolean_t is_mounted(libzfs_handle_t *, const char *special, char **);
 671 extern boolean_t zfs_is_mounted(zfs_handle_t *, char **);
 672 extern int zfs_mount(zfs_handle_t *, const char *, int);
 673 extern int zfs_unmount(zfs_handle_t *, const char *, int);
 674 extern int zfs_unmountall(zfs_handle_t *, int);

 676 /*
 677  * Share support functions.
 678  */
 679 extern boolean_t zfs_is_shared(zfs_handle_t *);
 680 extern int zfs_share(zfs_handle_t *);
 681 extern int zfs_unshare(zfs_handle_t *);

 683 /*
 684  * Protocol-specific share support functions.
 685  */
 686 extern boolean_t zfs_is_shared_nfs(zfs_handle_t *, char **);
 687 extern boolean_t zfs_is_shared_smb(zfs_handle_t *, char **);
 688 extern int zfs_share_nfs(zfs_handle_t *);
 689 extern int zfs_share_smb(zfs_handle_t *);
 690 extern int zfs_shareall(zfs_handle_t *);
 691 extern int zfs_unshare_nfs(zfs_handle_t *, const char *);
 692 extern int zfs_unshare_smb(zfs_handle_t *, const char *);
 693 extern int zfs_unshareall_nfs(zfs_handle_t *);
 694 extern int zfs_unshareall_smb(zfs_handle_t *);
 695 extern int zfs_unshareall_bypath(zfs_handle_t *, const char *);
 696 extern int zfs_unshareall(zfs_handle_t *);
 697 extern int zfs_deleg_share_nfs(libzfs_handle_t *, char *, char *, char *,
 698     void *, void *, int, zfs_share_op_t);
```

```
700 /*
701  * When dealing with nvlists, verify() is extremely useful
702  */
703 #ifdef NDEBUG
704 #define verify(EX)      ((void)(EX))
705 #else
706 #define verify(EX)      assert(EX)
707 #endif

709 /*
710  * Utility function to convert a number to a human-readable form.
711  */
712 extern void zfs_nicenum(uint64_t, char *, size_t);
713 extern int zfs_nicestrtonum(libzfs_handle_t *, const char *, uint64_t *);

715 /*
716  * Given a device or file, determine if it is part of a pool.
717  */
718 extern int zpool_in_use(libzfs_handle_t *, int, pool_state_t *, char **,
719     boolean_t *);

721 /*
722  * Label manipulation.
723  */
724 extern int zpool_read_label(int, nvlist_t **);
725 extern int zpool_clear_label(int);

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

730 /*
731  * Management interfaces for SMB ACL files
732  */

734 int zfs_smb_acl_add(libzfs_handle_t *, char *, char *, char *);
735 int zfs_smb_acl_remove(libzfs_handle_t *, char *, char *, char *);
736 int zfs_smb_acl_purge(libzfs_handle_t *, char *, char *);
737 int zfs_smb_acl_rename(libzfs_handle_t *, char *, char *, char *, char *);

739 /*
740  * Enable and disable datasets within a pool by mounting/unmounting and
741  * sharing/unsharing them.
742  */
743 extern int zpool_enable_datasets(zpool_handle_t *, const char *, int);
744 extern int zpool_disable_datasets(zpool_handle_t *, boolean_t);

746 /*
747  * Mappings between vdev and FRU.
748  */
749 extern void libzfs_fru_refresh(libzfs_handle_t *);
750 extern const char *libzfs_fru_lookup(libzfs_handle_t *, const char *);
751 extern const char *libzfs_fru_devpath(libzfs_handle_t *, const char *);
752 extern boolean_t libzfs_fru_compare(libzfs_handle_t *, const char *,
753     const char *);
754 extern boolean_t libzfs_fru_notself(libzfs_handle_t *, const char *);
755 extern int zpool_fru_set(zpool_handle_t *, uint64_t, const char *);

757 #ifdef  __cplusplus
758 }
759 #endif

761 #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_

4103 int
4104 zfs_hold_add(zfs_handle_t *zhp, const char *snapname, const char *tag,
4105     boolean_t enoent_ok, nvlist_t *holds)
4106 {
4107         zfs_handle_t *szhp;
4108         char name[ZFS_MAXNAMELEN];
4109         char errbuf[1024];
4110         int ret;

4112         (void) snprintf(name, sizeof (name),
4113             "%s@%s", zhp->zfs_name, snapname);

4115         szhp = make_dataset_handle(zhp->zfs_hdl, name);
4116         if (szhp) {
4117                 fnvlist_add_string(holds, name, tag);
4118                 zfs_close(szhp);
4119                 return (0);
4120         }

4122         ret = ENOENT;
4123         if (enoent_ok)
4124                 return (ret);

4126         (void) snprintf(errbuf, sizeof (errbuf),
4127             dgettext(TEXT_DOMAIN, "cannot hold snapshot '%s@%s'"),
4128             zhp->zfs_name, snapname);
4129         (void) zfs_standard_error(zhp->zfs_hdl, ret, errbuf);

4131         return (ret);
4132 }

4134 int
4135 #endif /* ! codereview */
4136 zfs_hold(zfs_handle_t *zhp, const char *snapname, const char *tag,
4137     boolean_t recursive, boolean_t enoent_ok, int cleanup_fd)
4138 {
4139         int ret;
4140         struct holdarg ha;
4104         nvlist_t *errors;
4105         libzfs_handle_t *hdl = zhp->zfs_hdl;
4106         char errbuf[1024];
4107         nvpair_t *elem;

4142         ha.nvl = fnvlist_alloc();
```



```
4143         ha.snapname = snapname;
4144         ha.tag = tag;
4145         ha.recursive = recursive;
4146         (void) zfs_hold_one(zfs_handle_dup(zhp), &ha);
4147         ret = zfs_hold_apply(zhp, enoent_ok, cleanup_fd, ha.nvl);
4114         ret = lzc_hold(ha.nvl, cleanup_fd, &errors);
4148         fnvlist_free(ha.nvl);

4150         return (ret);
4151 }

4153 int
4154 zfs_hold_apply(zfs_handle_t *zhp, boolean_t enoent_ok, int cleanup_fd, nvlist_t
4155 {
4156         int ret;
4157         nvlist_t *errors;
4158         libzfs_handle_t *hdl = zhp->zfs_hdl;
4159         char errbuf[1024];
4160         nvpair_t *elem;

4162         ret = lzc_hold(holds, cleanup_fd, &errors);

4164 #endif /* ! codereview */
4165         if (ret == 0)
4166                 return (0);

4168         if (nvlist_next_nvpair(errors, NULL) == NULL) {
4169                 /* no hold-specific errors */
4170                 (void) snprintf(errbuf, sizeof (errbuf),
4171                     dgettext(TEXT_DOMAIN, "cannot hold"));
4172                 switch (ret) {
4173                 case ENOTSUP:
4174                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4175                             "pool must be upgraded"));
4176                         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4177                         break;
4178                 case EINVAL:
4179                         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4180                         break;
4181                 default:
4182                         (void) zfs_standard_error(hdl, ret, errbuf);
4183                 }
4184         }

4186         for (elem = nvlist_next_nvpair(errors, NULL);
4187             elem != NULL;
4188             elem = nvlist_next_nvpair(errors, elem)) {
4189                 (void) snprintf(errbuf, sizeof (errbuf),
4190                     dgettext(TEXT_DOMAIN,
4191                     "cannot hold snapshot '%s'"), nvpair_name(elem));
4192                 switch (fnvpair_value_int32(elem)) {
4193                 case E2BIG:
4194                         /*
4195                          * Temporary tags wind up having the ds object id
4196                          * prepended. So even if we passed the length check
4197                          * above, it's still possible for the tag to wind
4198                          * up being slightly too long.
4199                          */
4200                         (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
4201                         break;
4202                 case EINVAL:
4203                         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4204                         break;
4205                 case EEXIST:
4206                         (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
4207                         break;
```

```
4208                 case ENOENT:
4209                         if (enoent_ok)
4210                                 return (ENOENT);
4211                         /* FALLTHROUGH */
4212                 default:
4213                         (void) zfs_standard_error(hdl,
4214                             fnvpair_value_int32(elem), errbuf);
4215                 }
4216         }

4218         fnvlist_free(errors);
4219         return (ret);
4220 }

4117 struct releasearg {
4118         nvlist_t *nvl;
4119         const char *snapname;
4120         const char *tag;
4121         boolean_t recursive;
4122 };

4222 static int
4223 zfs_release_one(zfs_handle_t *zhp, void *arg)
4224 {
4225         struct holdarg *ha = arg;
4226         zfs_handle_t *szhp;
4227         char name[ZFS_MAXNAMELEN];
4228         int rv = 0;

4230         (void) snprintf(name, sizeof (name),
4231             "%s@%s", zhp->zfs_name, ha->snapname);

4233         szhp = make_dataset_handle(zhp->zfs_hdl, name);
4234         if (szhp) {
4235                 nvlist_t *holds = fnvlist_alloc();
4236                 fnvlist_add_boolean(holds, ha->tag);
4237                 fnvlist_add_nvlist(ha->nvl, name, holds);
4238                 fnvlist_free(holds);
4239 #endif /* ! codereview */
4240                 zfs_close(szhp);
4241         }

4243         if (ha->recursive)
4244                 rv = zfs_iter_filesystems(zhp, zfs_release_one, ha);
4245         zfs_close(zhp);
4246         return (rv);
4247 }

4249 int
4250 zfs_release(zfs_handle_t *zhp, const char *snapname, const char *tag,
4251     boolean_t recursive)
4252 {
4253         int ret;
4254         struct holdarg ha;
4255         nvlist_t *errors;
4256         nvpair_t *elem;
4257         libzfs_handle_t *hdl = zhp->zfs_hdl;

4259         ha.nvl = fnvlist_alloc();
4260         ha.snapname = snapname;
4261         ha.tag = tag;
4262         ha.recursive = recursive;
4263         (void) zfs_release_one(zfs_handle_dup(zhp), &ha);
4264         ret = lzc_release(ha.nvl, &errors);
4265         fnvlist_free(ha.nvl);
```



```
4267         if (ret == 0)
4268                 return (0);

4270         if (nvlist_next_nvpair(errors, NULL) == NULL) {
4271                 /* no hold-specific errors */
4272                 char errbuf[1024];

4274                 (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
4275                     "cannot release"));
4276                 switch (errno) {
4277                 case ENOTSUP:
4278                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4279                             "pool must be upgraded"));
4280                         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4281                         break;
4282                 default:
4283                         (void) zfs_standard_error_fmt(hdl, errno, errbuf);
4284                 }
4285         }

4287         for (elem = nvlist_next_nvpair(errors, NULL);
4288             elem != NULL;
4289             elem = nvlist_next_nvpair(errors, elem)) {
4290                 char errbuf[1024];

4292                 (void) snprintf(errbuf, sizeof (errbuf),
4293                     dgettext(TEXT_DOMAIN,
4294                     "cannot release hold from snapshot '%s'"),
4295                     nvpair_name(elem));
4296                 switch (fnvpair_value_int32(elem)) {
4297                 case ESRCH:
4298                         (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
4299                         break;
4300                 case EINVAL:
4301                         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4302                         break;
4303                 default:
4304                         (void) zfs_standard_error_fmt(hdl,
4305                             fnvpair_value_int32(elem), errbuf);
4306                 }
4307         }

4309         fnvlist_free(errors);
4310         return (ret);
4311 }

4313 int
4314 zfs_get_fsacl(zfs_handle_t *zhp, nvlist_t **nvl)
4315 {
4316         zfs_cmd_t zc = { 0 };
4317         libzfs_handle_t *hdl = zhp->zfs_hdl;
4318         int nvsz = 2048;
4319         void *nvbuf;
4320         int err = 0;
4321         char errbuf[1024];

4323         assert(zhp->zfs_type == ZFS_TYPE_VOLUME ||
4324             zhp->zfs_type == ZFS_TYPE_FILESYSTEM);

4326 tryagain:

4328         nvbuf = malloc(nvsz);
4329         if (nvbuf == NULL) {
4330                 err = (zfs_error(hdl, EZFS_NOMEM, strerror(errno)));
4331                 goto out;
4332         }
```

```
4334         zc.zc_nvlist_dst_size = nvsz;
4335         zc.zc_nvlist_dst = (uintptr_t)nvbuf;

4337         (void) strlcpy(zc.zc_name, zhp->zfs_name, ZFS_MAXNAMELEN);

4339         if (ioctl(hdl->libzfs_fd, ZFS_IOC_GET_FSACL, &zc) != 0) {
4340                 (void) snprintf(errbuf, sizeof (errbuf),
4341                     dgettext(TEXT_DOMAIN, "cannot get permissions on '%s'"),
4342                     zc.zc_name);
4343                 switch (errno) {
4344                 case ENOMEM:
4345                         free(nvbuf);
4346                         nvsz = zc.zc_nvlist_dst_size;
4347                         goto tryagain;

4349                 case ENOTSUP:
4350                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4351                             "pool must be upgraded"));
4352                         err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4353                         break;
4354                 case EINVAL:
4355                         err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4356                         break;
4357                 case ENOENT:
4358                         err = zfs_error(hdl, EZFS_NOENT, errbuf);
4359                         break;
4360                 default:
4361                         err = zfs_standard_error_fmt(hdl, errno, errbuf);
4362                         break;
4363                 }
4364         } else {
4365                 /* success */
4366                 int rc = nvlist_unpack(nvbuf, zc.zc_nvlist_dst_size, nvl, 0);
4367                 if (rc) {
4368                         (void) snprintf(errbuf, sizeof (errbuf), dgettext(
4369                             TEXT_DOMAIN, "cannot get permissions on '%s'"),
4370                             zc.zc_name);
4371                         err = zfs_standard_error_fmt(hdl, rc, errbuf);
4372                 }
4373         }

4375         free(nvbuf);
4376 out:
4377         return (err);
4378 }

4380 int
4381 zfs_set_fsacl(zfs_handle_t *zhp, boolean_t un, nvlist_t *nvl)
4382 {
4383         zfs_cmd_t zc = { 0 };
4384         libzfs_handle_t *hdl = zhp->zfs_hdl;
4385         char *nvbuf;
4386         char errbuf[1024];
4387         size_t nvsz;
4388         int err;

4390         assert(zhp->zfs_type == ZFS_TYPE_VOLUME ||
4391             zhp->zfs_type == ZFS_TYPE_FILESYSTEM);

4393         err = nvlist_size(nvl, &nvsz, NV_ENCODE_NATIVE);
4394         assert(err == 0);

4396         nvbuf = malloc(nvsz);

4398         err = nvlist_pack(nvl, &nvbuf, &nvsz, NV_ENCODE_NATIVE, 0);
```



```
4399         assert(err == 0);

4401         zc.zc_nvlist_src_size = nvsz;
4402         zc.zc_nvlist_src = (uintptr_t)nvbuf;
4403         zc.zc_perm_action = un;

4405         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

4407         if (zfs_ioctl(hdl, ZFS_IOC_SET_FSACL, &zc) != 0) {
4408                 (void) snprintf(errbuf, sizeof (errbuf),
4409                     dgettext(TEXT_DOMAIN, "cannot set permissions on '%s'"),
4410                     zc.zc_name);
4411                 switch (errno) {
4412                 case ENOTSUP:
4413                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4414                             "pool must be upgraded"));
4415                         err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4416                         break;
4417                 case EINVAL:
4418                         err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4419                         break;
4420                 case ENOENT:
4421                         err = zfs_error(hdl, EZFS_NOENT, errbuf);
4422                         break;
4423                 default:
4424                         err = zfs_standard_error_fmt(hdl, errno, errbuf);
4425                         break;
4426                 }
4427         }

4429         free(nvbuf);

4431         return (err);
4432 }

4434 int
4435 zfs_get_holds(zfs_handle_t *zhp, nvlist_t **nvl)
4436 {
4437         int err;
4438         char errbuf[1024];

4440         err = lzc_get_holds(zhp->zfs_name, nvl);

4442         if (err != 0) {
4443                 libzfs_handle_t *hdl = zhp->zfs_hdl;

4445                 (void) snprintf(errbuf, sizeof (errbuf),
4446                     dgettext(TEXT_DOMAIN, "cannot get holds for '%s'"),
4447                     zhp->zfs_name);
4448                 switch (err) {
4449                 case ENOTSUP:
4450                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4451                             "pool must be upgraded"));
4452                         err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4453                         break;
4454                 case EINVAL:
4455                         err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4456                         break;
4457                 case ENOENT:
4458                         err = zfs_error(hdl, EZFS_NOENT, errbuf);
4459                         break;
4460                 default:
4461                         err = zfs_standard_error_fmt(hdl, errno, errbuf);
4462                         break;
4463                 }
4464         }
```

```
4466         return (err);
4467 }

4469 uint64_t
4470 zvol_volsize_to_reservation(uint64_t volsize, nvlist_t *props)
4471 {
4472         uint64_t numdb;
4473         uint64_t nblocks, volblocksize;
4474         int ncopies;
4475         char *strval;

4477         if (nvlist_lookup_string(props,
4478             zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)
4479                 ncopies = atoi(strval);
4480         else
4481                 ncopies = 1;
4482         if (nvlist_lookup_uint64(props,
4483             zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
4484             &volblocksize) != 0)
4485                 volblocksize = ZVOL_DEFAULT_BLOCKSIZE;
4486         nblocks = volsize/volblocksize;
4487         /* start with metadnode L0-L6 */
4488         numdb = 7;
4489         /* calculate number of indirects */
4490         while (nblocks > 1) {
4491                 nblocks += DNODES_PER_LEVEL - 1;
4492                 nblocks /= DNODES_PER_LEVEL;
4493                 numdb += nblocks;
4494         }
4495         numdb *= MIN(SPA_DVAS_PER_BP, ncopies + 1);
4496         volsize *= ncopies;
4497         /*
4498          * this is exactly DN_MAX_INDBLKSHIFT when metadata isn't
4499          * compressed, but in practice they compress down to about
4500          * 1100 bytes
4501          */
4502         numdb *= 1ULL << DN_MAX_INDBLKSHIFT;
4503         volsize += numdb;
4504         return (volsize);
4505 }
```

```
**********************************************************
   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_

 782 /*
 783  * Routines specific to "zfs send"
 784  */
 785 typedef struct send_dump_data {
 786         /* these are all just the short snapname (the part after the @) */
 787         const char *fromsnap;
 788         const char *tosnap;
 789         char prevsnap[ZFS_MAXNAMELEN];
 790         uint64_t prevsnap_obj;
 791         boolean_t seenfrom, seento, replicate, doall, fromorigin;
 792         boolean_t verbose, dryrun, parsable, progress;
 793         int outfd;
 794         boolean_t err;
 795         nvlist_t *fss;
 796         nvlist_t *snapholds;
 797 #endif /* ! codereview */
 798         avl_tree_t *fsavl;
 799         snapfilter_cb_t *filter_cb;
 800         void *filter_cb_arg;
 801         nvlist_t *debugnv;
 802         char holdtag[ZFS_MAXNAMELEN];
 803         int cleanup_fd;
 804         uint64_t size;
 805 } send_dump_data_t;

 807 static int
 808 estimate_ioctl(zfs_handle_t *zhp, uint64_t fromsnap_obj,
 809     boolean_t fromorigin, uint64_t *sizep)
 810 {
 811         zfs_cmd_t zc = { 0 };
 812         libzfs_handle_t *hdl = zhp->zfs_hdl;

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

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

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



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

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

 834                 case ENOENT:
 835                         if (zfs_dataset_exists(hdl, zc.zc_name,
 836                             ZFS_TYPE_SNAPSHOT)) {
 837                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
 838                                     "incremental source (@%s) does not exist"),
 839                                     zc.zc_value);
 840                         }
 841                         return (zfs_error(hdl, EZFS_NOENT, errbuf));

 843                 case EDQUOT:
 844                 case EFBIG:
 845                 case EIO:
 846                 case ENOLINK:
 847                 case ENOSPC:
 848                 case ENOSTR:
 849                 case ENXIO:
 850                 case EPIPE:
 851                 case ERANGE:
 852                 case EFAULT:
 853                 case EROFS:
 854                         zfs_error_aux(hdl, strerror(errno));
 855                         return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));

 857                 default:
 858                         return (zfs_standard_error(hdl, errno, errbuf));
 859                 }
 860         }

 862         *sizep = zc.zc_objset_type;

 864         return (0);
 865 }

 867 /*
 868  * Dumps a backup of the given snapshot (incremental from fromsnap if it's not
 869  * NULL) to the file descriptor specified by outfd.
 870  */
 871 static int
 872 dump_ioctl(zfs_handle_t *zhp, const char *fromsnap, uint64_t fromsnap_obj,
 873     boolean_t fromorigin, int outfd, nvlist_t *debugnv)
 874 {
 875         zfs_cmd_t zc = { 0 };
 876         libzfs_handle_t *hdl = zhp->zfs_hdl;
 877         nvlist_t *thisdbg;

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

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

 888         VERIFY(0 == nvlist_alloc(&thisdbg, NV_UNIQUE_NAME, 0));
 889         if (fromsnap && fromsnap[0] != '\0') {
 890                 VERIFY(0 == nvlist_add_string(thisdbg,
 891                     "fromsnap", fromsnap));
```

```
892         }

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

899                 VERIFY(0 == nvlist_add_uint64(thisdbg, "error", errno));
900                 if (debugnv) {
901                         VERIFY(0 == nvlist_add_nvlist(debugnv,
902                             zhp->zfs_name, thisdbg));
903                 }
904                 nvlist_free(thisdbg);

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

912                 case ENOENT:
913                         if (zfs_dataset_exists(hdl, zc.zc_name,
914                             ZFS_TYPE_SNAPSHOT)) {
915                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
916                                     "incremental source (@%s) does not exist"),
917                                     zc.zc_value);
918                         }
919                         return (zfs_error(hdl, EZFS_NOENT, errbuf));

921                 case EDQUOT:
922                 case EFBIG:
923                 case EIO:
924                 case ENOLINK:
925                 case ENOSPC:
926                 case ENOSTR:
927                 case ENXIO:
928                 case EPIPE:
929                 case ERANGE:
930                 case EFAULT:
931                 case EROFS:
932                         zfs_error_aux(hdl, strerror(errno));
933                         return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));

935                 default:
936                         return (zfs_standard_error(hdl, errno, errbuf));
937                 }
938         }

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

944         return (0);
945 }

947 static int
948 hold_for_send(zfs_handle_t *zhp, send_dump_data_t *sdd)
949 {
950         zfs_handle_t *pzhp;
951         int error = 0;
952         char *thissnap;

954         assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);

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



```
956         /*
957          * We process if snapholds is not NULL even if on a dry run as
958          * this is used to pre-calculate the required holds so they can
959          * be processed in one kernel request
800          * zfs_send() only opens a cleanup_fd for sends that need it,
801          * e.g. replication and doall.
960          */
961         if (sdd->snapholds == NULL)
803         if (sdd->cleanup_fd == -1)
962                 return (0);

964         thissnap = strchr(zhp->zfs_name, '@') + 1;
965         *(thissnap - 1) = '\0';
966         pzhp = zfs_open(zhp->zfs_hdl, zhp->zfs_name, ZFS_TYPE_DATASET);
967         *(thissnap - 1) = '@';

969         /*
970          * It's OK if the parent no longer exists.  The send code will
971          * handle that error.
972          */
973         if (pzhp) {
974                 error = zfs_hold_add(pzhp, thissnap, sdd->holdtag, B_TRUE,
975                     sdd->snapholds);
816                 error = zfs_hold(pzhp, thissnap, sdd->holdtag,
817                     B_FALSE, B_TRUE, sdd->cleanup_fd);
976                 zfs_close(pzhp);
977         }

979         return (error);
980 }
_____unchanged_portion_omitted_

1354 /*
1355  * Generate a send stream for the dataset identified by the argument zhp.
1356  *
1357  * The content of the send stream is the snapshot identified by
1358  * 'tosnap'.  Incremental streams are requested in two ways:
1359  *     - from the snapshot identified by "fromsnap" (if non-null) or
1360  *     - from the origin of the dataset identified by zhp, which must
1361  *       be a clone.  In this case, "fromsnap" is null and "fromorigin"
1362  *       is TRUE.
1363  *
1364  * The send stream is recursive (i.e. dumps a hierarchy of snapshots) and
1365  * uses a special header (with a hdrtype field of DMU_COMPOUNDSTREAM)
1366  * if "replicate" is set.  If "doall" is set, dump all the intermediate
1367  * snapshots. The DMU_COMPOUNDSTREAM header is used in the "doall"
1368  * case too. If "props" is set, send properties.
1369  */
1370 int
1371 zfs_send(zfs_handle_t *zhp, const char *fromsnap, const char *tosnap,
1372     sendflags_t *flags, int outfd, snapfilter_cb_t filter_func,
1373     void *cb_arg, nvlist_t **debugnvp)
1374 {
1375         char errbuf[1024];
1376         send_dump_data_t sdd = { 0 };
1377         int err = 0;
1378         nvlist_t *fss = NULL;
1379         avl_tree_t *fsavl = NULL;
1380         static uint64_t holdseq;
1381         int spa_version;
1382         pthread_t tid;
1383         int pipefd[2];
1384         dedup_arg_t dda = { 0 };
1385         int featureflags = 0;
```

```
1387         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
1388             "cannot send '%s'"), zhp->zfs_name);
1389
1390         if (fromsnap && fromsnap[0] == '\0') {
1391                 zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
1392                     "zero-length incremental source"));
1393                 return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
1394         }
1395
1396         if (zhp->zfs_type == ZFS_TYPE_FILESYSTEM) {
1397                 uint64_t version;
1398                 version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1399                 if (version >= ZPL_VERSION_SA) {
1400                         featureflags |= DMU_BACKUP_FEATURE_SA_SPILL;
1401                 }
1402         }
1403
1404         if (flags->dedup && !flags->dryrun) {
1405                 featureflags |= (DMU_BACKUP_FEATURE_DEDUP |
1406                     DMU_BACKUP_FEATURE_DEDUPPROPS);
1407                 if (err = pipe(pipefd)) {
1408                         zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1409                         return (zfs_error(zhp->zfs_hdl, EZFS_PIPEFAILED,
1410                             errbuf));
1411                 }
1412                 dda.outputfd = outfd;
1413                 dda.inputfd = pipefd[1];
1414                 dda.dedup_hdl = zhp->zfs_hdl;
1415                 if (err = pthread_create(&tid, NULL, cksummer, &dda)) {
1416                         (void) close(pipefd[0]);
1417                         (void) close(pipefd[1]);
1418                         zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1419                         return (zfs_error(zhp->zfs_hdl,
1420                             EZFS_THREADCREATEFAILED, errbuf));
1421                 }
1422         }
1423
1424         if (flags->replicate || flags->doall || flags->props) {
1425                 dmu_replay_record_t drr = { 0 };
1426                 char *packbuf = NULL;
1427                 size_t buflen = 0;
1428                 zio_cksum_t zc = { 0 };
1429
1430                 if (flags->replicate || flags->props) {
1431                         nvlist_t *hdrnv;
1432
1433                         VERIFY(0 == nvlist_alloc(&hdrnv, NV_UNIQUE_NAME, 0));
1434                         if (fromsnap) {
1435                                 VERIFY(0 == nvlist_add_string(hdrnv,
1436                                     "fromsnap", fromsnap));
1437                         }
1438                         VERIFY(0 == nvlist_add_string(hdrnv, "tosnap", tosnap));
1439                         if (!flags->replicate) {
1440                                 VERIFY(0 == nvlist_add_boolean(hdrnv,
1441                                     "not_recursive"));
1442                         }
1443
1444                         err = gather_nvlist(zhp->zfs_hdl, zhp->zfs_name,
1445                             fromsnap, tosnap, flags->replicate, &fss, &fsavl);
1446                         if (err)
1447                                 goto err_out;
1448                         VERIFY(0 == nvlist_add_nvlist(hdrnv, "fss", fss));
1449                         err = nvlist_pack(hdrnv, &packbuf, &buflen,
1450                             NV_ENCODE_XDR, 0);
1451                         if (debugnvp)
1452                                 *debugnvp = hdrnv;
```



```
1453                         else
1454                                 nvlist_free(hdrnv);
1455                         if (err) {
1456                                 fsavl_destroy(fsavl);
1457                                 nvlist_free(fss);
1458                                 goto stderr_out;
1459                         }
1460                 }
1461
1462                 if (!flags->dryrun) {
1463                         /* write first begin record */
1464                         drr.drr_type = DRR_BEGIN;
1465                         drr.drr_u.drr_begin.drr_magic = DMU_BACKUP_MAGIC;
1466                         DMU_SET_STREAM_HDRTYPE(drr.drr_u.drr_begin.
1467                             drr_versioninfo, DMU_COMPOUNDSTREAM);
1468                         DMU_SET_FEATUREFLAGS(drr.drr_u.drr_begin.
1469                             drr_versioninfo, featureflags);
1470                         (void) snprintf(drr.drr_u.drr_begin.drr_toname,
1471                             sizeof (drr.drr_u.drr_begin.drr_toname),
1472                             "%s@%s", zhp->zfs_name, tosnap);
1473                         drr.drr_payloadlen = buflen;
1474                         err = cksum_and_write(&drr, sizeof (drr), &zc, outfd);
1475
1476                         /* write header nvlist */
1477                         if (err != -1 && packbuf != NULL) {
1478                                 err = cksum_and_write(packbuf, buflen, &zc,
1479                                     outfd);
1480                         }
1481                         free(packbuf);
1482                         if (err == -1) {
1483                                 fsavl_destroy(fsavl);
1484                                 nvlist_free(fss);
1485                                 err = errno;
1486                                 goto stderr_out;
1487                         }
1488
1489                         /* write end record */
1490                         bzero(&drr, sizeof (drr));
1491                         drr.drr_type = DRR_END;
1492                         drr.drr_u.drr_end.drr_checksum = zc;
1493                         err = write(outfd, &drr, sizeof (drr));
1494                         if (err == -1) {
1495                                 fsavl_destroy(fsavl);
1496                                 nvlist_free(fss);
1497                                 err = errno;
1498                                 goto stderr_out;
1499                         }
1500
1501                         err = 0;
1502                 }
1503         }
1504
1505         /* dump each stream */
1506         sdd.fromsnap = fromsnap;
1507         sdd.tosnap = tosnap;
1508         if (flags->dedup)
1509                 sdd.outfd = pipefd[0];
1510         else
1511                 sdd.outfd = outfd;
1512         sdd.replicate = flags->replicate;
1513         sdd.doall = flags->doall;
1514         sdd.fromorigin = flags->fromorigin;
1515         sdd.fss = fss;
1516         sdd.fsavl = fsavl;
1517         sdd.verbose = flags->verbose;
1518         sdd.parsable = flags->parsable;
```

```
1519         sdd.progress = flags->progress;
1520         sdd.dryrun = flags->dryrun;
1521         sdd.filter_cb = filter_func;
1522         sdd.filter_cb_arg = cb_arg;
1523         if (debugnvp)
1524                 sdd.debugnv = *debugnvp;

1526         /*
1527          * Some flags require that we place user holds on the datasets that are
1528          * being sent so they don't get destroyed during the send. We can skip
1529          * this step if the pool is imported read-only since the datasets cannot
1530          * be destroyed.
1531          */
1532         if (!flags->dryrun && !zpool_get_prop_int(zfs_get_pool_handle(zhp),
1533             ZPOOL_PROP_READONLY, NULL) &&
1534             zfs_spa_version(zhp, &spa_version) == 0 &&
1535             spa_version >= SPA_VERSION_USERREFS &&
1536             (flags->doall || flags->replicate)) {
1537                 ++holdseq;
1538                 (void) snprintf(sdd.holdtag, sizeof (sdd.holdtag),
1539                     ".send-%d-%llu", getpid(), (u_longlong_t)holdseq);
1540                 sdd.cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
1541                 if (sdd.cleanup_fd < 0) {
1542                         err = errno;
1543                         goto stderr_out;
1544                 }
1545                 sdd.snapholds = fnvlist_alloc();
1546 #endif /* ! codereview */
1547         } else {
1548                 sdd.cleanup_fd = -1;
1549                 sdd.snapholds = NULL;
1550 #endif /* ! codereview */
1551         }
1552         if (flags->verbose) {
1553                 /*
1554                  * Do a verbose no-op dry run to get all the verbose output
1555                  * before generating any data.  Then do a non-verbose real
1556                  * run to generate the streams.
1557                  */
1558                 sdd.dryrun = B_TRUE;
1559                 err = dump_filesystems(zhp, &sdd);
1560                 sdd.dryrun = flags->dryrun;
1561                 sdd.verbose = B_FALSE;
1562                 if (flags->parsable) {
1563                         (void) fprintf(stderr, "size\t%llu\n",
1564                             (longlong_t)sdd.size);
1565                 } else {
1566                         char buf[16];
1567                         zfs_nicenum(sdd.size, buf, sizeof (buf));
1568                         (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1569                             "total estimated size is %s\n"), buf);
1570                 }
1571         }

1573         if (sdd.snapholds != NULL) {
1574                 /* Holds are required */
1575                 if (!flags->verbose) {
1576                         /*
1577                          * A verbose dry run wasn't done so do a non-verbose
1578                          * dry run to collate snapshot hold's.
1579                          */
1580                         sdd.dryrun = B_TRUE;
1581                         err = dump_filesystems(zhp, &sdd);
1582                         sdd.dryrun = flags->dryrun;
1583                 }
```



```
1585                 if (err != 0) {
1586                         fnvlist_free(sdd.snapholds);
1587                         goto stderr_out;
1588                 }

1590                 err = zfs_hold_apply(zhp, B_TRUE, sdd.cleanup_fd, sdd.snapholds)
1591                 fnvlist_free(sdd.snapholds);
1592                 if (err != 0)
1593                         goto stderr_out;
1594         }
1595
1596 #endif /* ! codereview */
1597         err = dump_filesystems(zhp, &sdd);
1598         fsavl_destroy(fsavl);
1599         nvlist_free(fss);

1601         if (flags->dedup) {
1602                 (void) close(pipefd[0]);
1603                 (void) pthread_join(tid, NULL);
1604         }

1606         if (sdd.cleanup_fd != -1) {
1607                 VERIFY(0 == close(sdd.cleanup_fd));
1608                 sdd.cleanup_fd = -1;
1609         }

1611         if (!flags->dryrun && (flags->replicate || flags->doall ||
1612             flags->props)) {
1613                 /*
1614                  * write final end record.  NB: want to do this even if
1615                  * there was some error, because it might not be totally
1616                  * failed.
1617                  */
1618                 dmu_replay_record_t drr = { 0 };
1619                 drr.drr_type = DRR_END;
1620                 if (write(outfd, &drr, sizeof (drr)) == -1) {
1621                         return (zfs_standard_error(zhp->zfs_hdl,
1622                             errno, errbuf));
1623                 }
1624         }

1626         return (err || sdd.err);

1628 stderr_out:
1629         err = zfs_standard_error(zhp->zfs_hdl, err, errbuf);
1630 err_out:
1631         if (sdd.cleanup_fd != -1)
1632                 VERIFY(0 == close(sdd.cleanup_fd));
1633         if (flags->dedup) {
1634                 (void) pthread_cancel(tid);
1635                 (void) pthread_join(tid, NULL);
1636                 (void) close(pipefd[0]);
1637         }
1638         return (err);
1639 }

1641 /*
1642  * Routines specific to "zfs recv"
1643  */

1645 static int
1646 recv_read(libzfs_handle_t *hdl, int fd, void *buf, int ilen,
1647     boolean_t byteswap, zio_cksum_t *zc)
1648 {
1649         char *cp = buf;
1650         int rv;
```

```
1651         int len = ilen;

1653         do {
1654                 rv = read(fd, cp, len);
1655                 cp += rv;
1656                 len -= rv;
1657         } while (rv > 0);

1659         if (rv < 0 || len != 0) {
1660                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1661                     "failed to read from stream"));
1662                 return (zfs_error(hdl, EZFS_BADSTREAM, dgettext(TEXT_DOMAIN,
1663                     "cannot receive")));
1664         }

1666         if (zc) {
1667                 if (byteswap)
1668                         fletcher_4_incremental_byteswap(buf, ilen, zc);
1669                 else
1670                         fletcher_4_incremental_native(buf, ilen, zc);
1671         }
1672         return (0);
1673 }

1675 static int
1676 recv_read_nvlist(libzfs_handle_t *hdl, int fd, int len, nvlist_t **nvp,
1677     boolean_t byteswap, zio_cksum_t *zc)
1678 {
1679         char *buf;
1680         int err;

1682         buf = zfs_alloc(hdl, len);
1683         if (buf == NULL)
1684                 return (ENOMEM);

1686         err = recv_read(hdl, fd, buf, len, byteswap, zc);
1687         if (err != 0) {
1688                 free(buf);
1689                 return (err);
1690         }

1692         err = nvlist_unpack(buf, len, nvp, 0);
1693         free(buf);
1694         if (err != 0) {
1695                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
1696                     "stream (malformed nvlist)"));
1697                 return (EINVAL);
1698         }
1699         return (0);
1700 }

1702 static int
1703 recv_rename(libzfs_handle_t *hdl, const char *name, const char *tryname,
1704     int baselen, char *newname, recvflags_t *flags)
1705 {
1706         static int seq;
1707         zfs_cmd_t zc = { 0 };
1708         int err;
1709         prop_changelist_t *clp;
1710         zfs_handle_t *zhp;

1712         zhp = zfs_open(hdl, name, ZFS_TYPE_DATASET);
1713         if (zhp == NULL)
1714                 return (-1);
1715         clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
1716             flags->force ? MS_FORCE : 0);
```



```
1717         zfs_close(zhp);
1718         if (clp == NULL)
1719                 return (-1);
1720         err = changelist_prefix(clp);
1721         if (err)
1722                 return (err);

1724         zc.zc_objset_type = DMU_OST_ZFS;
1725         (void) strlcpy(zc.zc_name, name, sizeof (zc.zc_name));

1727         if (tryname) {
1728                 (void) strcpy(newname, tryname);

1730                 (void) strlcpy(zc.zc_value, tryname, sizeof (zc.zc_value));

1732                 if (flags->verbose) {
1733                         (void) printf("attempting rename %s to %s\n",
1734                             zc.zc_name, zc.zc_value);
1735                 }
1736                 err = ioctl(hdl->libzfs_fd, ZFS_IOC_RENAME, &zc);
1737                 if (err == 0)
1738                         changelist_rename(clp, name, tryname);
1739         } else {
1740                 err = ENOENT;
1741         }

1743         if (err != 0 && strncmp(name + baselen, "recv-", 5) != 0) {
1744                 seq++;

1746                 (void) snprintf(newname, ZFS_MAXNAMELEN, "%.*srecv-%u-%u",
1747                     baselen, name, getpid(), seq);
1748                 (void) strlcpy(zc.zc_value, newname, sizeof (zc.zc_value));

1750                 if (flags->verbose) {
1751                         (void) printf("failed - trying rename %s to %s\n",
1752                             zc.zc_name, zc.zc_value);
1753                 }
1754                 err = ioctl(hdl->libzfs_fd, ZFS_IOC_RENAME, &zc);
1755                 if (err == 0)
1756                         changelist_rename(clp, name, newname);
1757                 if (err && flags->verbose) {
1758                         (void) printf("failed (%u) - "
1759                             "will try again on next pass\n", errno);
1760                 }
1761                 err = EAGAIN;
1762         } else if (flags->verbose) {
1763                 if (err == 0)
1764                         (void) printf("success\n");
1765                 else
1766                         (void) printf("failed (%u)\n", errno);
1767         }

1769         (void) changelist_postfix(clp);
1770         changelist_free(clp);

1772         return (err);
1773 }

1775 static int
1776 recv_destroy(libzfs_handle_t *hdl, const char *name, int baselen,
1777     char *newname, recvflags_t *flags)
1778 {
1779         zfs_cmd_t zc = { 0 };
1780         int err = 0;
1781         prop_changelist_t *clp;
1782         zfs_handle_t *zhp;
```

```
1783         boolean_t defer = B_FALSE;
1784         int spa_version;
1785
1786         zhp = zfs_open(hdl, name, ZFS_TYPE_DATASET);
1787         if (zhp == NULL)
1788                 return (-1);
1789         clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
1790             flags->force ? MS_FORCE : 0);
1791         if (zfs_get_type(zhp) == ZFS_TYPE_SNAPSHOT &&
1792             zfs_spa_version(zhp, &spa_version) == 0 &&
1793             spa_version >= SPA_VERSION_USERREFS)
1794                 defer = B_TRUE;
1795         zfs_close(zhp);
1796         if (clp == NULL)
1797                 return (-1);
1798         err = changelist_prefix(clp);
1799         if (err)
1800                 return (err);
1801
1802         zc.zc_objset_type = DMU_OST_ZFS;
1803         zc.zc_defer_destroy = defer;
1804         (void) strlcpy(zc.zc_name, name, sizeof (zc.zc_name));
1805
1806         if (flags->verbose)
1807                 (void) printf("attempting destroy %s\n", zc.zc_name);
1808         err = ioctl(hdl->libzfs_fd, ZFS_IOC_DESTROY, &zc);
1809         if (err == 0) {
1810                 if (flags->verbose)
1811                         (void) printf("success\n");
1812                 changelist_remove(clp, zc.zc_name);
1813         }
1814
1815         (void) changelist_postfix(clp);
1816         changelist_free(clp);
1817
1818         /*
1819          * Deferred destroy might destroy the snapshot or only mark it to be
1820          * destroyed later, and it returns success in either case.
1821          */
1822         if (err != 0 || (defer && zfs_dataset_exists(hdl, name,
1823             ZFS_TYPE_SNAPSHOT))) {
1824                 err = recv_rename(hdl, name, NULL, baselen, newname, flags);
1825         }
1826
1827         return (err);
1828 }
1829
1830 typedef struct guid_to_name_data {
1831         uint64_t guid;
1832         char *name;
1833         char *skip;
1834 } guid_to_name_data_t;
1835
1836 static int
1837 guid_to_name_cb(zfs_handle_t *zhp, void *arg)
1838 {
1839         guid_to_name_data_t *gtnd = arg;
1840         int err;
1841
1842         if (gtnd->skip != NULL &&
1843             strcmp(zhp->zfs_name, gtnd->skip) == 0) {
1844                 return (0);
1845         }
1846
1847         if (zhp->zfs_dmustats.dds_guid == gtnd->guid) {
1848                 (void) strcpy(gtnd->name, zhp->zfs_name);
```



```
1849                 zfs_close(zhp);
1850                 return (EEXIST);
1851         }
1852
1853         err = zfs_iter_children(zhp, guid_to_name_cb, gtnd);
1854         zfs_close(zhp);
1855         return (err);
1856 }
1857
1858 /*
1859  * Attempt to find the local dataset associated with this guid.  In the case of
1860  * multiple matches, we attempt to find the "best" match by searching
1861  * progressively larger portions of the hierarchy.  This allows one to send a
1862  * tree of datasets individually and guarantee that we will find the source
1863  * guid within that hierarchy, even if there are multiple matches elsewhere.
1864  */
1865 static int
1866 guid_to_name(libzfs_handle_t *hdl, const char *parent, uint64_t guid,
1867     char *name)
1868 {
1869         /* exhaustive search all local snapshots */
1870         char pname[ZFS_MAXNAMELEN];
1871         guid_to_name_data_t gtnd;
1872         int err = 0;
1873         zfs_handle_t *zhp;
1874         char *cp;
1875
1876         gtnd.guid = guid;
1877         gtnd.name = name;
1878         gtnd.skip = NULL;
1879
1880         (void) strlcpy(pname, parent, sizeof (pname));
1881
1882         /*
1883          * Search progressively larger portions of the hierarchy.  This will
1884          * select the "most local" version of the origin snapshot in the case
1885          * that there are multiple matching snapshots in the system.
1886          */
1887         while ((cp = strrchr(pname, '/')) != NULL) {
1888
1889                 /* Chop off the last component and open the parent */
1890                 *cp = '\0';
1891                 zhp = make_dataset_handle(hdl, pname);
1892
1893                 if (zhp == NULL)
1894                         continue;
1895
1896                 err = zfs_iter_children(zhp, guid_to_name_cb, &gtnd);
1897                 zfs_close(zhp);
1898                 if (err == EEXIST)
1899                         return (0);
1900
1901                 /*
1902                  * Remember the dataset that we already searched, so we
1903                  * skip it next time through.
1904                  */
1905                 gtnd.skip = pname;
1906         }
1907
1908         return (ENOENT);
1909 }
1910
1911 /*
1912  * Return +1 if guid1 is before guid2, 0 if they are the same, and -1 if
1913  * guid1 is after guid2.
1914  */
```

```
1915 static int
1916 created_before(libzfs_handle_t *hdl, avl_tree_t *avl,
1917     uint64_t guid1, uint64_t guid2)
1918 {
1919         nvlist_t *nvfs;
1920         char *fsname, *snapname;
1921         char buf[ZFS_MAXNAMELEN];
1922         int rv;
1923         zfs_handle_t *guid1hdl, *guid2hdl;
1924         uint64_t create1, create2;

1926         if (guid2 == 0)
1927                 return (0);
1928         if (guid1 == 0)
1929                 return (1);

1931         nvfs = fsavl_find(avl, guid1, &snapname);
1932         VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));
1933         (void) snprintf(buf, sizeof (buf), "%s@%s", fsname, snapname);
1934         guid1hdl = zfs_open(hdl, buf, ZFS_TYPE_SNAPSHOT);
1935         if (guid1hdl == NULL)
1936                 return (-1);

1938         nvfs = fsavl_find(avl, guid2, &snapname);
1939         VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));
1940         (void) snprintf(buf, sizeof (buf), "%s@%s", fsname, snapname);
1941         guid2hdl = zfs_open(hdl, buf, ZFS_TYPE_SNAPSHOT);
1942         if (guid2hdl == NULL) {
1943                 zfs_close(guid1hdl);
1944                 return (-1);
1945         }

1947         create1 = zfs_prop_get_int(guid1hdl, ZFS_PROP_CREATETXG);
1948         create2 = zfs_prop_get_int(guid2hdl, ZFS_PROP_CREATETXG);

1950         if (create1 < create2)
1951                 rv = -1;
1952         else if (create1 > create2)
1953                 rv = +1;
1954         else
1955                 rv = 0;

1957         zfs_close(guid1hdl);
1958         zfs_close(guid2hdl);

1960         return (rv);
1961 }

1963 static int
1964 recv_incremental_replication(libzfs_handle_t *hdl, const char *tofs,
1965     recvflags_t *flags, nvlist_t *stream_nv, avl_tree_t *stream_avl,
1966     nvlist_t *renamed)
1967 {
1968         nvlist_t *local_nv;
1969         avl_tree_t *local_avl;
1970         nvpair_t *fselem, *nextfselem;
1971         char *fromsnap;
1972         char newname[ZFS_MAXNAMELEN];
1973         int error;
1974         boolean_t needagain, progress, recursive;
1975         char *s1, *s2;

1977         VERIFY(0 == nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap));

1979         recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
1980             ENOENT);
```



```
1982         if (flags->dryrun)
1983                 return (0);

1985 again:
1986         needagain = progress = B_FALSE;

1988         if ((error = gather_nvlist(hdl, tofs, fromsnap, NULL,
1989             recursive, &local_nv, &local_avl)) != 0)
1990                 return (error);

1992         /*
1993          * Process deletes and renames
1994          */
1995         for (fselem = nvlist_next_nvpair(local_nv, NULL);
1996             fselem; fselem = nextfselem) {
1997                 nvlist_t *nvfs, *snaps;
1998                 nvlist_t *stream_nvfs = NULL;
1999                 nvpair_t *snapelem, *nextsnapelem;
2000                 uint64_t fromguid = 0;
2001                 uint64_t originguid = 0;
2002                 uint64_t stream_originguid = 0;
2003                 uint64_t parent_fromsnap_guid, stream_parent_fromsnap_guid;
2004                 char *fsname, *stream_fsname;

2006                 nextfselem = nvlist_next_nvpair(local_nv, fselem);

2008                 VERIFY(0 == nvpair_value_nvlist(fselem, &nvfs));
2009                 VERIFY(0 == nvlist_lookup_nvlist(nvfs, "snaps", &snaps));
2010                 VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));
2011                 VERIFY(0 == nvlist_lookup_uint64(nvfs, "parentfromsnap",
2012                     &parent_fromsnap_guid));
2013                 (void) nvlist_lookup_uint64(nvfs, "origin", &originguid);

2015                 /*
2016                  * First find the stream's fs, so we can check for
2017                  * a different origin (due to "zfs promote")
2018                  */
2019                 for (snapelem = nvlist_next_nvpair(snaps, NULL);
2020                     snapelem; snapelem = nvlist_next_nvpair(snaps, snapelem)) {
2021                         uint64_t thisguid;

2023                         VERIFY(0 == nvpair_value_uint64(snapelem, &thisguid));
2024                         stream_nvfs = fsavl_find(stream_avl, thisguid, NULL);

2026                         if (stream_nvfs != NULL)
2027                                 break;
2028                 }

2030                 /* check for promote */
2031                 (void) nvlist_lookup_uint64(stream_nvfs, "origin",
2032                     &stream_originguid);
2033                 if (stream_nvfs && originguid != stream_originguid) {
2034                         switch (created_before(hdl, local_avl,
2035                             stream_originguid, originguid)) {
2036                         case 1: {
2037                                 /* promote it! */
2038                                 zfs_cmd_t zc = { 0 };
2039                                 nvlist_t *origin_nvfs;
2040                                 char *origin_fsname;

2042                                 if (flags->verbose)
2043                                         (void) printf("promoting %s\n", fsname);

2045                                 origin_nvfs = fsavl_find(local_avl, originguid,
2046                                     NULL);
```

```
2047                                VERIFY(0 == nvlist_lookup_string(origin_nvfs,
2048                                    "name", &origin_fsname));
2049                                (void) strlcpy(zc.zc_value, origin_fsname,
2050                                    sizeof (zc.zc_value));
2051                                (void) strlcpy(zc.zc_name, fsname,
2052                                    sizeof (zc.zc_name));
2053                                error = zfs_ioctl(hdl, ZFS_IOC_PROMOTE, &zc);
2054                                if (error == 0)
2055                                        progress = B_TRUE;
2056                                break;
2057                        }
2058                        default:
2059                                break;
2060                        case -1:
2061                                fsavl_destroy(local_avl);
2062                                nvlist_free(local_nv);
2063                                return (-1);
2064                        }
2065                        /*
2066                         * We had/have the wrong origin, therefore our
2067                         * list of snapshots is wrong.  Need to handle
2068                         * them on the next pass.
2069                         */
2070                        needagain = B_TRUE;
2071                        continue;
2072                }

2074                for (snapelem = nvlist_next_nvpair(snaps, NULL);
2075                    snapelem; snapelem = nextsnapelem) {
2076                        uint64_t thisguid;
2077                        char *stream_snapname;
2078                        nvlist_t *found, *props;

2080                        nextsnapelem = nvlist_next_nvpair(snaps, snapelem);

2082                        VERIFY(0 == nvpair_value_uint64(snapelem, &thisguid));
2083                        found = fsavl_find(stream_avl, thisguid,
2084                            &stream_snapname);

2086                        /* check for delete */
2087                        if (found == NULL) {
2088                                char name[ZFS_MAXNAMELEN];

2090                                if (!flags->force)
2091                                        continue;

2093                                (void) snprintf(name, sizeof (name), "%s@%s",
2094                                    fsname, nvpair_name(snapelem));

2096                                error = recv_destroy(hdl, name,
2097                                    strlen(fsname)+1, newname, flags);
2098                                if (error)
2099                                        needagain = B_TRUE;
2100                                else
2101                                        progress = B_TRUE;
2102                                continue;
2103                        }

2105                        stream_nvfs = found;

2107                        if (0 == nvlist_lookup_nvlist(stream_nvfs, "snapprops",
2108                            &props) && 0 == nvlist_lookup_nvlist(props,
2109                            stream_snapname, &props)) {
2110                                zfs_cmd_t zc = { 0 };

2112                                zc.zc_cookie = B_TRUE; /* received */
```



```
2113                                (void) snprintf(zc.zc_name, sizeof (zc.zc_name),
2114                                    "%s@%s", fsname, nvpair_name(snapelem));
2115                                if (zcmd_write_src_nvlist(hdl, &zc,
2116                                    props) == 0) {
2117                                        (void) zfs_ioctl(hdl,
2118                                            ZFS_IOC_SET_PROP, &zc);
2119                                        zcmd_free_nvlists(&zc);
2120                                }
2121                        }

2123                        /* check for different snapname */
2124                        if (strcmp(nvpair_name(snapelem),
2125                            stream_snapname) != 0) {
2126                                char name[ZFS_MAXNAMELEN];
2127                                char tryname[ZFS_MAXNAMELEN];

2129                                (void) snprintf(name, sizeof (name), "%s@%s",
2130                                    fsname, nvpair_name(snapelem));
2131                                (void) snprintf(tryname, sizeof (name), "%s@%s",
2132                                    fsname, stream_snapname);

2134                                error = recv_rename(hdl, name, tryname,
2135                                    strlen(fsname)+1, newname, flags);
2136                                if (error)
2137                                        needagain = B_TRUE;
2138                                else
2139                                        progress = B_TRUE;
2140                        }

2142                        if (strcmp(stream_snapname, fromsnap) == 0)
2143                                fromguid = thisguid;
2144                }

2146                /* check for delete */
2147                if (stream_nvfs == NULL) {
2148                        if (!flags->force)
2149                                continue;

2151                        error = recv_destroy(hdl, fsname, strlen(tofs)+1,
2152                            newname, flags);
2153                        if (error)
2154                                needagain = B_TRUE;
2155                        else
2156                                progress = B_TRUE;
2157                        continue;
2158                }

2160                if (fromguid == 0) {
2161                        if (flags->verbose) {
2162                                (void) printf("local fs %s does not have "
2163                                    "fromsnap (%s in stream); must have "
2164                                    "been deleted locally; ignoring\n",
2165                                    fsname, fromsnap);
2166                        }
2167                        continue;
2168                }

2170                VERIFY(0 == nvlist_lookup_string(stream_nvfs,
2171                    "name", &stream_fsname));
2172                VERIFY(0 == nvlist_lookup_uint64(stream_nvfs,
2173                    "parentfromsnap", &stream_parent_fromsnap_guid));

2175                s1 = strrchr(fsname, '/');
2176                s2 = strrchr(stream_fsname, '/');

2178                /*
```

```
2179                 * Check for rename. If the exact receive path is specified, it
2180                 * does not count as a rename, but we still need to check the
2181                 * datasets beneath it.
2182                 */
2183                if ((stream_parent_fromsnap_guid != 0 &&
2184                    parent_fromsnap_guid != 0 &&
2185                    stream_parent_fromsnap_guid != parent_fromsnap_guid) ||
2186                    ((flags->isprefix || strcmp(tofs, fsname) != 0) &&
2187                    (s1 != NULL) && (s2 != NULL) && strcmp(s1, s2) != 0)) {
2188                        nvlist_t *parent;
2189                        char tryname[ZFS_MAXNAMELEN];

2191                        parent = fsavl_find(local_avl,
2192                            stream_parent_fromsnap_guid, NULL);
2193                        /*
2194                         * NB: parent might not be found if we used the
2195                         * tosnap for stream_parent_fromsnap_guid,
2196                         * because the parent is a newly-created fs;
2197                         * we'll be able to rename it after we recv the
2198                         * new fs.
2199                         */
2200                        if (parent != NULL) {
2201                                char *pname;

2203                                VERIFY(0 == nvlist_lookup_string(parent, "name",
2204                                    &pname));
2205                                (void) snprintf(tryname, sizeof (tryname),
2206                                    "%s%s", pname, strrchr(stream_fsname, '/'));
2207                        } else {
2208                                tryname[0] = '\0';
2209                                if (flags->verbose) {
2210                                        (void) printf("local fs %s new parent "
2211                                            "not found\n", fsname);
2212                                }
2213                        }

2215                        newname[0] = '\0';

2217                        error = recv_rename(hdl, fsname, tryname,
2218                            strlen(tofs)+1, newname, flags);

2220                        if (renamed != NULL && newname[0] != '\0') {
2221                                VERIFY(0 == nvlist_add_boolean(renamed,
2222                                    newname));
2223                        }

2225                        if (error)
2226                                needagain = B_TRUE;
2227                        else
2228                                progress = B_TRUE;
2229                }
2230        }

2232        fsavl_destroy(local_avl);
2233        nvlist_free(local_nv);

2235        if (needagain && progress) {
2236                /* do another pass to fix up temporary names */
2237                if (flags->verbose)
2238                        (void) printf("another pass:\n");
2239                goto again;
2240        }

2242        return (needagain);
2243 }
```



```
2245 static int
2246 zfs_receive_package(libzfs_handle_t *hdl, int fd, const char *destname,
2247     recvflags_t *flags, dmu_replay_record_t *drr, zio_cksum_t *zc,
2248     char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
2249 {
2250         nvlist_t *stream_nv = NULL;
2251         avl_tree_t *stream_avl = NULL;
2252         char *fromsnap = NULL;
2253         char *cp;
2254         char tofs[ZFS_MAXNAMELEN];
2255         char sendfs[ZFS_MAXNAMELEN];
2256         char errbuf[1024];
2257         dmu_replay_record_t drre;
2258         int error;
2259         boolean_t anyerr = B_FALSE;
2260         boolean_t softerr = B_FALSE;
2261         boolean_t recursive;

2263         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2264             "cannot receive"));

2266         assert(drr->drr_type == DRR_BEGIN);
2267         assert(drr->drr_u.drr_begin.drr_magic == DMU_BACKUP_MAGIC);
2268         assert(DMU_GET_STREAM_HDRTYPE(drr->drr_u.drr_begin.drr_versioninfo) ==
2269             DMU_COMPOUNDSTREAM);

2271         /*
2272          * Read in the nvlist from the stream.
2273          */
2274         if (drr->drr_payloadlen != 0) {
2275                 error = recv_read_nvlist(hdl, fd, drr->drr_payloadlen,
2276                     &stream_nv, flags->byteswap, zc);
2277                 if (error) {
2278                         error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2279                         goto out;
2280                 }
2281         }

2283         recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
2284             ENOENT);

2286         if (recursive && strchr(destname, '@')) {
2287                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2288                     "cannot specify snapshot name for multi-snapshot stream"));
2289                 error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2290                 goto out;
2291         }

2293         /*
2294          * Read in the end record and verify checksum.
2295          */
2296         if (0 != (error = recv_read(hdl, fd, &drre, sizeof (drre),
2297             flags->byteswap, NULL)))
2298                 goto out;
2299         if (flags->byteswap) {
2300                 drre.drr_type = BSWAP_32(drre.drr_type);
2301                 drre.drr_u.drr_end.drr_checksum.zc_word[0] =
2302                     BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[0]);
2303                 drre.drr_u.drr_end.drr_checksum.zc_word[1] =
2304                     BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[1]);
2305                 drre.drr_u.drr_end.drr_checksum.zc_word[2] =
2306                     BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[2]);
2307                 drre.drr_u.drr_end.drr_checksum.zc_word[3] =
2308                     BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[3]);
2309         }
2310         if (drre.drr_type != DRR_END) {
```

```
2311                    error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2312                    goto out;
2313            }
2314            if (!ZIO_CHECKSUM_EQUAL(drre.drr_u.drr_end.drr_checksum, *zc)) {
2315                    zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2316                        "incorrect header checksum"));
2317                    error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2318                    goto out;
2319            }

2321            (void) nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap);

2323            if (drr->drr_payloadlen != 0) {
2324                    nvlist_t *stream_fss;

2326                    VERIFY(0 == nvlist_lookup_nvlist(stream_nv, "fss",
2327                        &stream_fss));
2328                    if ((stream_avl = fsavl_create(stream_fss)) == NULL) {
2329                            zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2330                                "couldn't allocate avl tree"));
2331                            error = zfs_error(hdl, EZFS_NOMEM, errbuf);
2332                            goto out;
2333                    }

2335                    if (fromsnap != NULL) {
2336                            nvlist_t *renamed = NULL;
2337                            nvpair_t *pair = NULL;

2339                            (void) strlcpy(tofs, destname, ZFS_MAXNAMELEN);
2340                            if (flags->isprefix) {
2341                                    struct drr_begin *drrb = &drr->drr_u.drr_begin;
2342                                    int i;

2344                                    if (flags->istail) {
2345                                            cp = strrchr(drrb->drr_toname, '/');
2346                                            if (cp == NULL) {
2347                                                    (void) strlcat(tofs, "/",
2348                                                        ZFS_MAXNAMELEN);
2349                                                    i = 0;
2350                                            } else {
2351                                                    i = (cp - drrb->drr_toname);
2352                                            }
2353                                    } else {
2354                                            i = strcspn(drrb->drr_toname, "/@");
2355                                    }
2356                                    /* zfs_receive_one() will create_parents() */
2357                                    (void) strlcat(tofs, &drrb->drr_toname[i],
2358                                        ZFS_MAXNAMELEN);
2359                                    *strchr(tofs, '@') = '\0';
2360                            }

2362                            if (recursive && !flags->dryrun && !flags->nomount) {
2363                                    VERIFY(0 == nvlist_alloc(&renamed,
2364                                        NV_UNIQUE_NAME, 0));
2365                            }

2367                            softerr = recv_incremental_replication(hdl, tofs, flags,
2368                                stream_nv, stream_avl, renamed);

2370                            /* Unmount renamed filesystems before receiving. */
2371                            while ((pair = nvlist_next_nvpair(renamed,
2372                                pair)) != NULL) {
2373                                zfs_handle_t *zhp;
2374                                prop_changelist_t *clp = NULL;

2376                                zhp = zfs_open(hdl, nvpair_name(pair),
```



```
2377                                    ZFS_TYPE_FILESYSTEM);
2378                                if (zhp != NULL) {
2379                                        clp = changelist_gather(zhp,
2380                                            ZFS_PROP_MOUNTPOINT, 0, 0);
2381                                        zfs_close(zhp);
2382                                        if (clp != NULL) {
2383                                                softerr |=
2384                                                    changelist_prefix(clp);
2385                                                changelist_free(clp);
2386                                        }
2387                                }
2388                        }

2390                        nvlist_free(renamed);
2391                }
2392        }

2394        /*
2395         * Get the fs specified by the first path in the stream (the top level
2396         * specified by 'zfs send') and pass it to each invocation of
2397         * zfs_receive_one().
2398         */
2399        (void) strlcpy(sendfs, drr->drr_u.drr_begin.drr_toname,
2400            ZFS_MAXNAMELEN);
2401        if ((cp = strchr(sendfs, '@')) != NULL)
2402                *cp = '\0';

2404        /* Finally, receive each contained stream */
2405        do {
2406                /*
2407                 * we should figure out if it has a recoverable
2408                 * error, in which case do a recv_skip() and drive on.
2409                 * Note, if we fail due to already having this guid,
2410                 * zfs_receive_one() will take care of it (ie,
2411                 * recv_skip() and return 0).
2412                 */
2413                error = zfs_receive_impl(hdl, destname, flags, fd,
2414                    sendfs, stream_nv, stream_avl, top_zfs, cleanup_fd,
2415                    action_handlep);
2416                if (error == ENODATA) {
2417                        error = 0;
2418                        break;
2419                }
2420                anyerr |= error;
2421        } while (error == 0);

2423        if (drr->drr_payloadlen != 0 && fromsnap != NULL) {
2424                /*
2425                 * Now that we have the fs's they sent us, try the
2426                 * renames again.
2427                 */
2428                softerr = recv_incremental_replication(hdl, tofs, flags,
2429                    stream_nv, stream_avl, NULL);
2430        }

2432 out:
2433        fsavl_destroy(stream_avl);
2434        if (stream_nv)
2435                nvlist_free(stream_nv);
2436        if (softerr)
2437                error = -2;
2438        if (anyerr)
2439                error = -1;
2440        return (error);
2441 }
```

```
2443 static void
2444 trunc_prop_errs(int truncated)
2445 {
2446         ASSERT(truncated != 0);

2448         if (truncated == 1)
2449                 (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
2450                     "1 more property could not be set\n"));
2451         else
2452                 (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
2453                     "%d more properties could not be set\n"), truncated);
2454 }

2456 static int
2457 recv_skip(libzfs_handle_t *hdl, int fd, boolean_t byteswap)
2458 {
2459         dmu_replay_record_t *drr;
2460         void *buf = malloc(1<<20);
2461         char errbuf[1024];

2463         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2464             "cannot receive:"));

2466         /* XXX would be great to use lseek if possible... */
2467         drr = buf;

2469         while (recv_read(hdl, fd, drr, sizeof (dmu_replay_record_t),
2470             byteswap, NULL) == 0) {
2471                 if (byteswap)
2472                         drr->drr_type = BSWAP_32(drr->drr_type);

2474                 switch (drr->drr_type) {
2475                 case DRR_BEGIN:
2476                         /* NB: not to be used on v2 stream packages */
2477                         if (drr->drr_payloadlen != 0) {
2478                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2479                                     "invalid substream header"));
2480                                 return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2481                         }
2482                         break;

2484                 case DRR_END:
2485                         free(buf);
2486                         return (0);

2488                 case DRR_OBJECT:
2489                         if (byteswap) {
2490                                 drr->drr_u.drr_object.drr_bonuslen =
2491                                     BSWAP_32(drr->drr_u.drr_object.
2492                                     drr_bonuslen);
2493                         }
2494                         (void) recv_read(hdl, fd, buf,
2495                             P2ROUNDUP(drr->drr_u.drr_object.drr_bonuslen, 8),
2496                             B_FALSE, NULL);
2497                         break;

2499                 case DRR_WRITE:
2500                         if (byteswap) {
2501                                 drr->drr_u.drr_write.drr_length =
2502                                     BSWAP_64(drr->drr_u.drr_write.drr_length);
2503                         }
2504                         (void) recv_read(hdl, fd, buf,
2505                             drr->drr_u.drr_write.drr_length, B_FALSE, NULL);
2506                         break;
2507                 case DRR_SPILL:
2508                         if (byteswap) {
```



```
2509                                 drr->drr_u.drr_write.drr_length =
2510                                     BSWAP_64(drr->drr_u.drr_spill.drr_length);
2511                         }
2512                         (void) recv_read(hdl, fd, buf,
2513                             drr->drr_u.drr_spill.drr_length, B_FALSE, NULL);
2514                         break;
2515                 case DRR_WRITE_BYREF:
2516                 case DRR_FREEOBJECTS:
2517                 case DRR_FREE:
2518                         break;

2520                 default:
2521                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2522                             "invalid record type"));
2523                         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2524                 }
2525         }

2527         free(buf);
2528         return (-1);
2529 }

2531 /*
2532  * Restores a backup of tosnap from the file descriptor specified by infd.
2533  */
2534 static int
2535 zfs_receive_one(libzfs_handle_t *hdl, int infd, const char *tosnap,
2536     recvflags_t *flags, dmu_replay_record_t *drr,
2537     dmu_replay_record_t *drr_noswap, const char *sendfs,
2538     nvlist_t *stream_nv, avl_tree_t *stream_avl, char **top_zfs, int cleanup_fd,
2539     uint64_t *action_handlep)
2540 {
2541         zfs_cmd_t zc = { 0 };
2542         time_t begin_time;
2543         int ioctl_err, ioctl_errno, err;
2544         char *cp;
2545         struct drr_begin *drrb = &drr->drr_u.drr_begin;
2546         char errbuf[1024];
2547         char prop_errbuf[1024];
2548         const char *chopprefix;
2549         boolean_t newfs = B_FALSE;
2550         boolean_t stream_wantsnewfs;
2551         uint64_t parent_snapguid = 0;
2552         prop_changelist_t *clp = NULL;
2553         nvlist_t *snapprops_nvlist = NULL;
2554         zprop_errflags_t prop_errflags;
2555         boolean_t recursive;

2557         begin_time = time(NULL);

2559         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2560             "cannot receive"));

2562         recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
2563             ENOENT);

2565         if (stream_avl != NULL) {
2566                 char *snapname;
2567                 nvlist_t *fs = fsavl_find(stream_avl, drrb->drr_toguid,
2568                     &snapname);
2569                 nvlist_t *props;
2570                 int ret;

2572                 (void) nvlist_lookup_uint64(fs, "parentfromsnap",
2573                     &parent_snapguid);
2574                 err = nvlist_lookup_nvlist(fs, "props", &props);
```

```
2575                if (err)
2576                        VERIFY(0 == nvlist_alloc(&props, NV_UNIQUE_NAME, 0));

2578                if (flags->canmountoff) {
2579                        VERIFY(0 == nvlist_add_uint64(props,
2580                            zfs_prop_to_name(ZFS_PROP_CANMOUNT), 0));
2581                }
2582                ret = zcmd_write_src_nvlist(hdl, &zc, props);
2583                if (err)
2584                        nvlist_free(props);

2586                if (0 == nvlist_lookup_nvlist(fs, "snapprops", &props)) {
2587                        VERIFY(0 == nvlist_lookup_nvlist(props,
2588                            snapname, &snapprops_nvlist));
2589                }

2591                if (ret != 0)
2592                        return (-1);
2593        }

2595        cp = NULL;

2597        /*
2598         * Determine how much of the snapshot name stored in the stream
2599         * we are going to tack on to the name they specified on the
2600         * command line, and how much we are going to chop off.
2601         *
2602         * If they specified a snapshot, chop the entire name stored in
2603         * the stream.
2604         */
2605        if (flags->istail) {
2606                /*
2607                 * A filesystem was specified with -e. We want to tack on only
2608                 * the tail of the sent snapshot path.
2609                 */
2610                if (strchr(tosnap, '@')) {
2611                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
2612                            "argument - snapshot not allowed with -e"));
2613                        return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2614                }

2616                chopprefix = strrchr(sendfs, '/');

2618                if (chopprefix == NULL) {
2619                        /*
2620                         * The tail is the poolname, so we need to
2621                         * prepend a path separator.
2622                         */
2623                        int len = strlen(drrb->drr_toname);
2624                        cp = malloc(len + 2);
2625                        cp[0] = '/';
2626                        (void) strcpy(&cp[1], drrb->drr_toname);
2627                        chopprefix = cp;
2628                } else {
2629                        chopprefix = drrb->drr_toname + (chopprefix - sendfs);
2630                }
2631        } else if (flags->isprefix) {
2632                /*
2633                 * A filesystem was specified with -d. We want to tack on
2634                 * everything but the first element of the sent snapshot path
2635                 * (all but the pool name).
2636                 */
2637                if (strchr(tosnap, '@')) {
2638                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
2639                            "argument - snapshot not allowed with -d"));
2640                        return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
```



```
2641                }

2643                chopprefix = strchr(drrb->drr_toname, '/');
2644                if (chopprefix == NULL)
2645                        chopprefix = strchr(drrb->drr_toname, '@');
2646        } else if (strchr(tosnap, '@') == NULL) {
2647                /*
2648                 * If a filesystem was specified without -d or -e, we want to
2649                 * tack on everything after the fs specified by 'zfs send'.
2650                 */
2651                chopprefix = drrb->drr_toname + strlen(sendfs);
2652        } else {
2653                /* A snapshot was specified as an exact path (no -d or -e). */
2654                if (recursive) {
2655                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2656                            "cannot specify snapshot name for multi-snapshot "
2657                            "stream"));
2658                        return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2659                }
2660                chopprefix = drrb->drr_toname + strlen(drrb->drr_toname);
2661        }

2663        ASSERT(strstr(drrb->drr_toname, sendfs) == drrb->drr_toname);
2664        ASSERT(chopprefix > drrb->drr_toname);
2665        ASSERT(chopprefix <= drrb->drr_toname + strlen(drrb->drr_toname));
2666        ASSERT(chopprefix[0] == '/' || chopprefix[0] == '@' ||
2667            chopprefix[0] == '\0');

2669        /*
2670         * Determine name of destination snapshot, store in zc_value.
2671         */
2672        (void) strcpy(zc.zc_value, tosnap);
2673        (void) strncat(zc.zc_value, chopprefix, sizeof (zc.zc_value));
2674        free(cp);
2675        if (!zfs_name_valid(zc.zc_value, ZFS_TYPE_SNAPSHOT)) {
2676                zcmd_free_nvlists(&zc);
2677                return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2678        }

2680        /*
2681         * Determine the name of the origin snapshot, store in zc_string.
2682         */
2683        if (drrb->drr_flags & DRR_FLAG_CLONE) {
2684                if (guid_to_name(hdl, zc.zc_value,
2685                    drrb->drr_fromguid, zc.zc_string) != 0) {
2686                        zcmd_free_nvlists(&zc);
2687                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2688                            "local origin for clone %s does not exist"),
2689                            zc.zc_value);
2690                        return (zfs_error(hdl, EZFS_NOENT, errbuf));
2691                }
2692                if (flags->verbose)
2693                        (void) printf("found clone origin %s\n", zc.zc_string);
2694        }

2696        stream_wantsnewfs = (drrb->drr_fromguid == NULL ||
2697            (drrb->drr_flags & DRR_FLAG_CLONE));

2699        if (stream_wantsnewfs) {
2700                /*
2701                 * if the parent fs does not exist, look for it based on
2702                 * the parent snap GUID
2703                 */
2704                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2705                    "cannot receive new filesystem stream"));
```

```
2707                (void) strcpy(zc.zc_name, zc.zc_value);
2708                cp = strrchr(zc.zc_name, '/');
2709                if (cp)
2710                        *cp = '\0';
2711                if (cp &&
2712                    !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2713                        char suffix[ZFS_MAXNAMELEN];
2714                        (void) strcpy(suffix, strrchr(zc.zc_value, '/'));
2715                        if (guid_to_name(hdl, zc.zc_name, parent_snapguid,
2716                            zc.zc_value) == 0) {
2717                                *strchr(zc.zc_value, '@') = '\0';
2718                                (void) strcat(zc.zc_value, suffix);
2719                        }
2720                }
2721        } else {
2722                /*
2723                 * if the fs does not exist, look for it based on the
2724                 * fromsnap GUID
2725                 */
2726                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2727                    "cannot receive incremental stream"));

2729                (void) strcpy(zc.zc_name, zc.zc_value);
2730                *strchr(zc.zc_name, '@') = '\0';

2732                /*
2733                 * If the exact receive path was specified and this is the
2734                 * topmost path in the stream, then if the fs does not exist we
2735                 * should look no further.
2736                 */
2737                if ((flags->isprefix || (*(chopprefix = drrb->drr_toname +
2738                    strlen(sendfs)) != '\0' && *chopprefix != '@')) &&
2739                    !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2740                        char snap[ZFS_MAXNAMELEN];
2741                        (void) strcpy(snap, strchr(zc.zc_value, '@'));
2742                        if (guid_to_name(hdl, zc.zc_name, drrb->drr_fromguid,
2743                            zc.zc_value) == 0) {
2744                                *strchr(zc.zc_value, '@') = '\0';
2745                                (void) strcat(zc.zc_value, snap);
2746                        }
2747                }
2748        }

2750        (void) strcpy(zc.zc_name, zc.zc_value);
2751        *strchr(zc.zc_name, '@') = '\0';

2753        if (zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2754                zfs_handle_t *zhp;

2756                /*
2757                 * Destination fs exists.  Therefore this should either
2758                 * be an incremental, or the stream specifies a new fs
2759                 * (full stream or clone) and they want us to blow it
2760                 * away (and have therefore specified -F and removed any
2761                 * snapshots).
2762                 */
2763                if (stream_wantsnewfs) {
2764                        if (!flags->force) {
2765                                zcmd_free_nvlists(&zc);
2766                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2767                                    "destination '%s' exists\n"
2768                                    "must specify -F to overwrite it"),
2769                                    zc.zc_name);
2770                                return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2771                        }
2772                        if (ioctl(hdl->libzfs_fd, ZFS_IOC_SNAPSHOT_LIST_NEXT,
```



```
2773                            &zc) == 0) {
2774                                zcmd_free_nvlists(&zc);
2775                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2776                                    "destination has snapshots (eg. %s)\n"
2777                                    "must destroy them to overwrite it"),
2778                                    zc.zc_name);
2779                                return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2780                        }
2781                }

2783                if ((zhp = zfs_open(hdl, zc.zc_name,
2784                    ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME)) == NULL) {
2785                        zcmd_free_nvlists(&zc);
2786                        return (-1);
2787                }

2789                if (stream_wantsnewfs &&
2790                    zhp->zfs_dmustats.dds_origin[0]) {
2791                        zcmd_free_nvlists(&zc);
2792                        zfs_close(zhp);
2793                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2794                            "destination '%s' is a clone\n"
2795                            "must destroy it to overwrite it"),
2796                            zc.zc_name);
2797                        return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2798                }

2800                if (!flags->dryrun && zhp->zfs_type == ZFS_TYPE_FILESYSTEM &&
2801                    stream_wantsnewfs) {
2802                        /* We can't do online recv in this case */
2803                        clp = changelist_gather(zhp, ZFS_PROP_NAME, 0, 0);
2804                        if (clp == NULL) {
2805                                zfs_close(zhp);
2806                                zcmd_free_nvlists(&zc);
2807                                return (-1);
2808                        }
2809                        if (changelist_prefix(clp) != 0) {
2810                                changelist_free(clp);
2811                                zfs_close(zhp);
2812                                zcmd_free_nvlists(&zc);
2813                                return (-1);
2814                        }
2815                }
2816                zfs_close(zhp);
2817        } else {
2818                /*
2819                 * Destination filesystem does not exist.  Therefore we better
2820                 * be creating a new filesystem (either from a full backup, or
2821                 * a clone).  It would therefore be invalid if the user
2822                 * specified only the pool name (i.e. if the destination name
2823                 * contained no slash character).
2824                 */
2825                if (!stream_wantsnewfs ||
2826                    (cp = strrchr(zc.zc_name, '/')) == NULL) {
2827                        zcmd_free_nvlists(&zc);
2828                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2829                            "destination '%s' does not exist"), zc.zc_name);
2830                        return (zfs_error(hdl, EZFS_NOENT, errbuf));
2831                }

2833                /*
2834                 * Trim off the final dataset component so we perform the
2835                 * recvbackup ioctl to the filesystems's parent.
2836                 */
2837                *cp = '\0';
```

```
2839                if (flags->isprefix && !flags->istail && !flags->dryrun &&
2840                    create_parents(hdl, zc.zc_value, strlen(tosnap)) != 0) {
2841                        zcmd_free_nvlists(&zc);
2842                        return (zfs_error(hdl, EZFS_BADRESTORE, errbuf));
2843                }

2845                newfs = B_TRUE;
2846        }

2848        zc.zc_begin_record = drr_noswap->drr_u.drr_begin;
2849        zc.zc_cookie = infd;
2850        zc.zc_guid = flags->force;
2851        if (flags->verbose) {
2852                (void) printf("%s %s stream of %s into %s\n",
2853                    flags->dryrun ? "would receive" : "receiving",
2854                    drrb->drr_fromguid ? "incremental" : "full",
2855                    drrb->drr_toname, zc.zc_value);
2856                (void) fflush(stdout);
2857        }

2859        if (flags->dryrun) {
2860                zcmd_free_nvlists(&zc);
2861                return (recv_skip(hdl, infd, flags->byteswap));
2862        }

2864        zc.zc_nvlist_dst = (uint64_t)(uintptr_t)prop_errbuf;
2865        zc.zc_nvlist_dst_size = sizeof (prop_errbuf);
2866        zc.zc_cleanup_fd = cleanup_fd;
2867        zc.zc_action_handle = *action_handlep;

2869        err = ioctl_err = zfs_ioctl(hdl, ZFS_IOC_RECV, &zc);
2870        ioctl_errno = errno;
2871        prop_errflags = (zprop_errflags_t)zc.zc_obj;

2873        if (err == 0) {
2874                nvlist_t *prop_errors;
2875                VERIFY(0 == nvlist_unpack((void *)(uintptr_t)zc.zc_nvlist_dst,
2876                    zc.zc_nvlist_dst_size, &prop_errors, 0));

2878                nvpair_t *prop_err = NULL;

2880                while ((prop_err = nvlist_next_nvpair(prop_errors,
2881                    prop_err)) != NULL) {
2882                        char tbuf[1024];
2883                        zfs_prop_t prop;
2884                        int intval;

2886                        prop = zfs_name_to_prop(nvpair_name(prop_err));
2887                        (void) nvpair_value_int32(prop_err, &intval);
2888                        if (strcmp(nvpair_name(prop_err),
2889                            ZPROP_N_MORE_ERRORS) == 0) {
2890                                trunc_prop_errs(intval);
2891                                break;
2892                        } else {
2893                                (void) snprintf(tbuf, sizeof (tbuf),
2894                                    dgettext(TEXT_DOMAIN,
2895                                    "cannot receive %s property on %s"),
2896                                    nvpair_name(prop_err), zc.zc_name);
2897                                zfs_setprop_error(hdl, prop, intval, tbuf);
2898                        }
2899                }
2900                nvlist_free(prop_errors);
2901        }

2903        zc.zc_nvlist_dst = 0;
2904        zc.zc_nvlist_dst_size = 0;
```



```
2905        zcmd_free_nvlists(&zc);

2907        if (err == 0 && snapprops_nvlist) {
2908                zfs_cmd_t zc2 = { 0 };

2910                (void) strcpy(zc2.zc_name, zc.zc_value);
2911                zc2.zc_cookie = B_TRUE; /* received */
2912                if (zcmd_write_src_nvlist(hdl, &zc2, snapprops_nvlist) == 0) {
2913                        (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc2);
2914                        zcmd_free_nvlists(&zc2);
2915                }
2916        }

2918        if (err && (ioctl_errno == ENOENT || ioctl_errno == EEXIST)) {
2919                /*
2920                 * It may be that this snapshot already exists,
2921                 * in which case we want to consume & ignore it
2922                 * rather than failing.
2923                 */
2924                avl_tree_t *local_avl;
2925                nvlist_t *local_nv, *fs;
2926                cp = strchr(zc.zc_value, '@');

2928                /*
2929                 * XXX Do this faster by just iterating over snaps in
2930                 * this fs.  Also if zc_value does not exist, we will
2931                 * get a strange "does not exist" error message.
2932                 */
2933                *cp = '\0';
2934                if (gather_nvlist(hdl, zc.zc_value, NULL, NULL, B_FALSE,
2935                    &local_nv, &local_avl) == 0) {
2936                        *cp = '@';
2937                        fs = fsavl_find(local_avl, drrb->drr_toguid, NULL);
2938                        fsavl_destroy(local_avl);
2939                        nvlist_free(local_nv);

2941                        if (fs != NULL) {
2942                                if (flags->verbose) {
2943                                        (void) printf("snap %s already exists; "
2944                                            "ignoring\n", zc.zc_value);
2945                                }
2946                                err = ioctl_err = recv_skip(hdl, infd,
2947                                    flags->byteswap);
2948                        }
2949                }
2950                *cp = '@';
2951        }

2953        if (ioctl_err != 0) {
2954                switch (ioctl_errno) {
2955                case ENODEV:
2956                        cp = strchr(zc.zc_value, '@');
2957                        *cp = '\0';
2958                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2959                            "most recent snapshot of %s does not\n"
2960                            "match incremental source"), zc.zc_value);
2961                        (void) zfs_error(hdl, EZFS_BADRESTORE, errbuf);
2962                        *cp = '@';
2963                        break;
2964                case ETXTBSY:
2965                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2966                            "destination %s has been modified\n"
2967                            "since most recent snapshot"), zc.zc_name);
2968                        (void) zfs_error(hdl, EZFS_BADRESTORE, errbuf);
2969                        break;
2970                case EEXIST:
```

```
2971                        cp = strchr(zc.zc_value, '@');
2972                        if (newfs) {
2973                                /* it's the containing fs that exists */
2974                                *cp = '\0';
2975                        }
2976                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2977                            "destination already exists"));
2978                        (void) zfs_error_fmt(hdl, EZFS_EXISTS,
2979                            dgettext(TEXT_DOMAIN, "cannot restore to %s"),
2980                            zc.zc_value);
2981                        *cp = '@';
2982                        break;
2983                case EINVAL:
2984                        (void) zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2985                        break;
2986                case ECKSUM:
2987                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2988                            "invalid stream (checksum mismatch)"));
2989                        (void) zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2990                        break;
2991                case ENOTSUP:
2992                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2993                            "pool must be upgraded to receive this stream."));
2994                        (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
2995                        break;
2996                case EDQUOT:
2997                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2998                            "destination %s space quota exceeded"), zc.zc_name);
2999                        (void) zfs_error(hdl, EZFS_NOSPC, errbuf);
3000                        break;
3001                default:
3002                        (void) zfs_standard_error(hdl, ioctl_errno, errbuf);
3003                }
3004        }

3006        /*
3007         * Mount the target filesystem (if created).  Also mount any
3008         * children of the target filesystem if we did a replication
3009         * receive (indicated by stream_avl being non-NULL).
3010         */
3011        cp = strchr(zc.zc_value, '@');
3012        if (cp && (ioctl_err == 0 || !newfs)) {
3013                zfs_handle_t *h;

3015                *cp = '\0';
3016                h = zfs_open(hdl, zc.zc_value,
3017                    ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3018                if (h != NULL) {
3019                        if (h->zfs_type == ZFS_TYPE_VOLUME) {
3020                                *cp = '@';
3021                        } else if (newfs || stream_avl) {
3022                                /*
3023                                 * Track the first/top of hierarchy fs,
3024                                 * for mounting and sharing later.
3025                                 */
3026                                if (top_zfs && *top_zfs == NULL)
3027                                        *top_zfs = zfs_strdup(hdl, zc.zc_value);
3028                        }
3029                        zfs_close(h);
3030                }
3031                *cp = '@';
3032        }

3034        if (clp) {
3035                err |= changelist_postfix(clp);
3036                changelist_free(clp);
```



```
3037        }

3039        if (prop_errflags & ZPROP_ERR_NOCLEAR) {
3040                (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "Warning: "
3041                    "failed to clear unreceived properties on %s"),
3042                    zc.zc_name);
3043                (void) fprintf(stderr, "\n");
3044        }
3045        if (prop_errflags & ZPROP_ERR_NORESTORE) {
3046                (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "Warning: "
3047                    "failed to restore original properties on %s"),
3048                    zc.zc_name);
3049                (void) fprintf(stderr, "\n");
3050        }

3052        if (err || ioctl_err)
3053                return (-1);

3055        *action_handlep = zc.zc_action_handle;

3057        if (flags->verbose) {
3058                char buf1[64];
3059                char buf2[64];
3060                uint64_t bytes = zc.zc_cookie;
3061                time_t delta = time(NULL) - begin_time;
3062                if (delta == 0)
3063                        delta = 1;
3064                zfs_nicenum(bytes, buf1, sizeof (buf1));
3065                zfs_nicenum(bytes/delta, buf2, sizeof (buf1));

3067                (void) printf("received %sB stream in %lu seconds (%sB/sec)\n",
3068                    buf1, delta, buf2);
3069        }

3071        return (0);
3072 }

3074 static int
3075 zfs_receive_impl(libzfs_handle_t *hdl, const char *tosnap, recvflags_t *flags,
3076     int infd, const char *sendfs, nvlist_t *stream_nv, avl_tree_t *stream_avl,
3077     char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
3078 {
3079        int err;
3080        dmu_replay_record_t drr, drr_noswap;
3081        struct drr_begin *drrb = &drr.drr_u.drr_begin;
3082        char errbuf[1024];
3083        zio_cksum_t zcksum = { 0 };
3084        uint64_t featureflags;
3085        int hdrtype;

3087        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
3088            "cannot receive"));

3090        if (flags->isprefix &&
3091            !zfs_dataset_exists(hdl, tosnap, ZFS_TYPE_DATASET)) {
3092                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "specified fs "
3093                    "(%s) does not exist"), tosnap);
3094                return (zfs_error(hdl, EZFS_NOENT, errbuf));
3095        }

3097        /* read in the BEGIN record */
3098        if (0 != (err = recv_read(hdl, infd, &drr, sizeof (drr), B_FALSE,
3099            &zcksum)))
3100                return (err);

3102        if (drr.drr_type == DRR_END || drr.drr_type == BSWAP_32(DRR_END)) {
```

```
3103                 /* It's the double end record at the end of a package */
3104                 return (ENODATA);
3105         }
3106
3107         /* the kernel needs the non-byteswapped begin record */
3108         drr_noswap = drr;
3109
3110         flags->byteswap = B_FALSE;
3111         if (drrb->drr_magic == BSWAP_64(DMU_BACKUP_MAGIC)) {
3112                 /*
3113                  * We computed the checksum in the wrong byteorder in
3114                  * recv_read() above; do it again correctly.
3115                  */
3116                 bzero(&zcksum, sizeof (zio_cksum_t));
3117                 fletcher_4_incremental_byteswap(&drr, sizeof (drr), &zcksum);
3118                 flags->byteswap = B_TRUE;
3119
3120                 drr.drr_type = BSWAP_32(drr.drr_type);
3121                 drr.drr_payloadlen = BSWAP_32(drr.drr_payloadlen);
3122                 drrb->drr_magic = BSWAP_64(drrb->drr_magic);
3123                 drrb->drr_versioninfo = BSWAP_64(drrb->drr_versioninfo);
3124                 drrb->drr_creation_time = BSWAP_64(drrb->drr_creation_time);
3125                 drrb->drr_type = BSWAP_32(drrb->drr_type);
3126                 drrb->drr_flags = BSWAP_32(drrb->drr_flags);
3127                 drrb->drr_toguid = BSWAP_64(drrb->drr_toguid);
3128                 drrb->drr_fromguid = BSWAP_64(drrb->drr_fromguid);
3129         }
3130
3131         if (drrb->drr_magic != DMU_BACKUP_MAGIC || drr.drr_type != DRR_BEGIN) {
3132                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
3133                     "stream (bad magic number)"));
3134                 return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
3135         }
3136
3137         featureflags = DMU_GET_FEATUREFLAGS(drrb->drr_versioninfo);
3138         hdrtype = DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo);
3139
3140         if (!DMU_STREAM_SUPPORTED(featureflags) ||
3141             (hdrtype != DMU_SUBSTREAM && hdrtype != DMU_COMPOUNDSTREAM)) {
3142                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3143                     "stream has unsupported feature, feature flags = %lx"),
3144                     featureflags);
3145                 return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
3146         }
3147
3148         if (strchr(drrb->drr_toname, '@') == NULL) {
3149                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
3150                     "stream (bad snapshot name)"));
3151                 return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
3152         }
3153
3154         if (DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) == DMU_SUBSTREAM) {
3155                 char nonpackage_sendfs[ZFS_MAXNAMELEN];
3156                 if (sendfs == NULL) {
3157                         /*
3158                          * We were not called from zfs_receive_package(). Get
3159                          * the fs specified by 'zfs send'.
3160                          */
3161                         char *cp;
3162                         (void) strlcpy(nonpackage_sendfs,
3163                             drr.drr_u.drr_begin.drr_toname, ZFS_MAXNAMELEN);
3164                         if ((cp = strchr(nonpackage_sendfs, '@')) != NULL)
3165                                 *cp = '\0';
3166                         sendfs = nonpackage_sendfs;
3167                 }
3168                 return (zfs_receive_one(hdl, infd, tosnap, flags,
```



```
3169                     &drr, &drr_noswap, sendfs, stream_nv, stream_avl,
3170                     top_zfs, cleanup_fd, action_handlep));
3171         } else {
3172                 assert(DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) ==
3173                     DMU_COMPOUNDSTREAM);
3174                 return (zfs_receive_package(hdl, infd, tosnap, flags,
3175                     &drr, &zcksum, top_zfs, cleanup_fd, action_handlep));
3176         }
3177 }
3178
3179 /*
3180  * Restores a backup of tosnap from the file descriptor specified by infd.
3181  * Return 0 on total success, -2 if some things couldn't be
3182  * destroyed/renamed/promoted, -1 if some things couldn't be received.
3183  * (-1 will override -2).
3184  */
3185 int
3186 zfs_receive(libzfs_handle_t *hdl, const char *tosnap, recvflags_t *flags,
3187     int infd, avl_tree_t *stream_avl)
3188 {
3189         char *top_zfs = NULL;
3190         int err;
3191         int cleanup_fd;
3192         uint64_t action_handle = 0;
3193
3194         cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
3195         VERIFY(cleanup_fd >= 0);
3196
3197         err = zfs_receive_impl(hdl, tosnap, flags, infd, NULL, NULL,
3198             stream_avl, &top_zfs, cleanup_fd, &action_handle);
3199
3200         VERIFY(0 == close(cleanup_fd));
3201
3202         if (err == 0 && !flags->nomount && top_zfs) {
3203                 zfs_handle_t *zhp;
3204                 prop_changelist_t *clp;
3205
3206                 zhp = zfs_open(hdl, top_zfs, ZFS_TYPE_FILESYSTEM);
3207                 if (zhp != NULL) {
3208                         clp = changelist_gather(zhp, ZFS_PROP_MOUNTPOINT,
3209                             CL_GATHER_MOUNT_ALWAYS, 0);
3210                         zfs_close(zhp);
3211                         if (clp != NULL) {
3212                                 /* mount and share received datasets */
3213                                 err = changelist_postfix(clp);
3214                                 changelist_free(clp);
3215                         }
3216                 }
3217                 if (zhp == NULL || clp == NULL || err)
3218                         err = -1;
3219         }
3220         if (top_zfs)
3221                 free(top_zfs);
3222
3223         return (err);
3224 }
```

```
**********************************************************
   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_

 828 /*
 829  * Walk through the pool-wide zap object of temporary snapshot user holds
 830  * and release them.
 831  */
 832 void
 833 dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp)
 834 {
 835         char *htag;
 836 #endif /* ! codereview */
 837         zap_attribute_t za;
 838         zap_cursor_t zc;
 839         objset_t *mos = dp->dp_meta_objset;
 840         uint64_t zapobj = dp->dp_tmp_userrefs_obj;
 841         uint64_t dsobj;
 842         nvlist_t *holds, *tags;
 843         dsl_dataset_t *ds;
 844         char name[MAXNAMELEN];
 845 #endif /* ! codereview */

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

 851         holds = fnvlist_alloc();

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

 858                 htag = strchr(za.za_name, '-');
 859                 *htag = '\0';
 860                 ++htag;
 861                 dsobj = strtonum(za.za_name, NULL);
 862                 if (dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds) == 0) {
 863                         dsl_dataset_name(ds, name);
 864                         if (nvlist_lookup_nvlist(holds, name, &tags) != 0) {
 865                                 tags = fnvlist_alloc();
 866                                 fnvlist_add_boolean(tags, htag);
 867                                 fnvlist_add_nvlist(holds, name, tags);
 868                                 fnvlist_free(tags);
```



```
 869                         } else {
 870                                 fnvlist_add_boolean(tags, htag);
 871                         }
 872                         dsl_dataset_rele(ds, FTAG);
 873                 }
 842                 dsl_dataset_user_release_tmp(dp, dsobj, htag);
 874         }
 875         dsl_pool_config_exit(dp, FTAG);
 876         dsl_dataset_user_release(holds, NULL);
 877         fnvlist_free(holds);
 878 #endif /* ! codereview */
 879         zap_cursor_fini(&zc);
 880 }

 882 /*
 883  * Create the pool-wide zap object for storing temporary snapshot holds.
 884  */
 885 void
 886 dsl_pool_user_hold_create_obj(dsl_pool_t *dp, dmu_tx_t *tx)
 887 {
 888         objset_t *mos = dp->dp_meta_objset;

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

 893         dp->dp_tmp_userrefs_obj = zap_create_link(mos, DMU_OT_USERREFS,
 894             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_TMP_USERREFS, tx);
 895 }

 897 static int
 898 dsl_pool_user_hold_rele_impl(dsl_pool_t *dp, uint64_t dsobj,
 899     const char *tag, uint64_t now, dmu_tx_t *tx, boolean_t holding)
 900 {
 901         objset_t *mos = dp->dp_meta_objset;
 902         uint64_t zapobj = dp->dp_tmp_userrefs_obj;
 903         char *name;
 904         int error;

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

 909         /*
 910          * If the pool was created prior to SPA_VERSION_USERREFS, the
 911          * zap object for temporary holds might not exist yet.
 912          */
 913         if (zapobj == 0) {
 914                 if (holding) {
 915                         dsl_pool_user_hold_create_obj(dp, tx);
 916                         zapobj = dp->dp_tmp_userrefs_obj;
 917                 } else {
 918                         return (SET_ERROR(ENOENT));
 919                 }
 920         }

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

 929         return (error);
 930 }

 932 /*
 933  * 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_
```

```
122 void
123 dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
124     minor_t minor, uint64_t now, dmu_tx_t *tx)
125 {
126         dsl_pool_t *dp = ds->ds_dir->dd_pool;
127         objset_t *mos = dp->dp_meta_objset;
128         uint64_t zapobj;
129
130         mutex_enter(&ds->ds_lock);
131         if (ds->ds_phys->ds_userrefs_obj == 0) {
132                 /*
133                  * This is the first user hold for this dataset.  Create
134                  * the userrefs zap object.
135                  */
136                 dmu_buf_will_dirty(ds->ds_dbuf, tx);
137                 zapobj = ds->ds_phys->ds_userrefs_obj =
138                     zap_create(mos, DMU_OT_USERREFS, DMU_OT_NONE, 0, tx);
139         } else {
140                 zapobj = ds->ds_phys->ds_userrefs_obj;
141         }
142         ds->ds_userrefs++;
143         mutex_exit(&ds->ds_lock);
144
145         VERIFY0(zap_add(mos, zapobj, htag, 8, 1, &now, tx));
146
147         if (minor != 0) {
148                 VERIFY0(dsl_pool_user_hold(dp, ds->ds_object,
149                     htag, now, tx));
150                 dsl_register_onexit_hold_cleanup(ds, htag, minor);
150         }
151
152         spa_history_log_internal_ds(ds, "hold", tx,
153             "tag=%s temp=%d refs=%llu",
154             htag, minor != 0, ds->ds_userrefs);
155 }
156
157 static void
158 dsl_dataset_user_hold_sync(void *arg, dmu_tx_t *tx)
159 {
160         dsl_dataset_user_hold_arg_t *dduha = arg;
161         dsl_pool_t *dp = dmu_tx_pool(tx);
162         nvpair_t *pair;
163         uint64_t now = gethrestime_sec();
```



```
165         for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;
166             pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {
167                 dsl_dataset_t *ds;
168
169 #endif /* ! codereview */
170                 VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
171                 dsl_dataset_user_hold_sync_one(ds, fnvpair_value_string(pair),
172                     dduha->dduha_minor, now, tx);
173                 dsl_dataset_rele(ds, FTAG);
174         }
175 }
176
177 /*
178  * holds is nvl of snapname -> holdname
179  * errlist will be filled in with snapname -> error
180  * if cleanup_minor is not 0, the holds will be temporary, cleaned up
181  * when the process exits.
182  *
183  * if any fails, all will fail.
184  */
185 int
186 dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor, nvlist_t *errlist)
187 {
188         dsl_dataset_user_hold_arg_t dduha;
189         nvpair_t *pair;
190         int ret;
191 #endif /* ! codereview */
192
193         pair = nvlist_next_nvpair(holds, NULL);
194         if (pair == NULL)
195                 return (0);
196
197         dduha.dduha_holds = holds;
198         dduha.dduha_errlist = errlist;
199         dduha.dduha_minor = cleanup_minor;
200
201         ret = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
202             dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds));
203         if (ret == 0)
204                 dsl_register_onexit_hold_cleanup(holds, cleanup_minor);
205
206         return (ret);
169         return (dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
170             dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds)));
207 }
```

```
______unchanged_portion_omitted_
```

```
351 /*
352  * holds is nvl of snapname -> { holdname, ... }
353  * errlist will be filled in with snapname -> error
354  *
355  * if any fails, all will fail.
356  */
357 int
358 dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist)
359 {
360         dsl_dataset_user_release_arg_t ddura;
361         nvpair_t *pair, *pair2;
325         nvpair_t *pair;
362         int error;
363
364         pair = nvlist_next_nvpair(holds, NULL);
365         if (pair == NULL)
366                 return (0);
367
368 #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;
```

```
420 static void
421 dsl_dataset_user_release_onexit(void *arg)
422 {
423         zfs_hold_cleanup_arg_t *ca = arg;
424         spa_t *spa;
425         int error;

427         error = spa_open(ca->zhca_spaname, &spa, FTAG);
428         if (error != 0) {
429                 zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
430                     "because pool is no longer loaded",
431                     ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
432                 return;
433         }
434         if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
435                 zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
436                     "because pool is no longer loaded (guid doesn't match)",
437                     ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
438                 spa_close(spa, FTAG);
439                 return;
440         }

442         dsl_dataset_user_release_tmp(spa_get_dsl(spa),
443             ca->zhca_dsobj, ca->zhca_htag);
444         kmem_free(ca, sizeof (zfs_hold_cleanup_arg_t));
445         spa_close(spa, FTAG);
446 }

448 void
449 dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
450     minor_t minor)
451 {
452         zfs_hold_cleanup_arg_t *ca = kmem_alloc(sizeof (*ca), KM_SLEEP);
453         spa_t *spa = dsl_dataset_get_spa(ds);
454         (void) strlcpy(ca->zhca_spaname, spa_name(spa),
455             sizeof (ca->zhca_spaname));
456         ca->zhca_spa_load_guid = spa_load_guid(spa);
457         ca->zhca_dsobj = ds->ds_object;
458         (void) strlcpy(ca->zhca_htag, htag, sizeof (ca->zhca_htag));
422         VERIFY0(zfs_onexit_add_cb(minor,
423             dsl_dataset_user_release_onexit, ca, NULL));
424 }
```

_____***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_

 166 /*
 167  * The max length of a temporary tag prefix is the number of hex digits
 168  * required to express UINT64_MAX plus one for the hyphen.
 169  */
 170 #define MAX_TAG_PREFIX_LEN      17

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

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

 178 int dsl_dataset_hold(struct dsl_pool *dp, const char *name, void *tag,
 179     dsl_dataset_t **dsp);
 180 int dsl_dataset_hold_obj(struct dsl_pool *dp, uint64_t dsobj, void *tag,
 181     dsl_dataset_t **);
 182 void dsl_dataset_rele(dsl_dataset_t *ds, void *tag);
 183 int dsl_dataset_own(struct dsl_pool *dp, const char *name,
 184     void *tag, dsl_dataset_t **dsp);
 185 int dsl_dataset_own_obj(struct dsl_pool *dp, uint64_t dsobj,
 186     void *tag, dsl_dataset_t **dsp);
 187 void dsl_dataset_disown(dsl_dataset_t *ds, void *tag);
 188 void dsl_dataset_name(dsl_dataset_t *ds, char *name);
 189 boolean_t dsl_dataset_tryown(dsl_dataset_t *ds, void *tag);
 190 void dsl_register_onexit_hold_cleanup(nvlist_t *holds, minor_t minor);
 190 void dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
 191     minor_t minor);
 191 uint64_t dsl_dataset_create_sync(dsl_dir_t *pds, const char *lastname,
 192     dsl_dataset_t *origin, uint64_t flags, cred_t *, dmu_tx_t *);
 193 uint64_t dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
 194     uint64_t flags, dmu_tx_t *tx);
 195 int dsl_dataset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors);
 196 int dsl_dataset_promote(const char *name, char *conflsnap);
 197 int dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
 198     boolean_t force);
 199 int dsl_dataset_rename_snapshot(const char *fsname,
 200     const char *oldsnapname, const char *newsnapname, boolean_t recursive);
 201 int dsl_dataset_snapshot_tmp(const char *fsname, const char *snapname,
 202     minor_t cleanup_minor, const char *htag);

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

 207 spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);
```



```
 209 boolean_t dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds);

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

 213 void dsl_dataset_block_born(dsl_dataset_t *ds, const blkptr_t *bp,
 214     dmu_tx_t *tx);
 215 int dsl_dataset_block_kill(dsl_dataset_t *ds, const blkptr_t *bp,
 216     dmu_tx_t *tx, boolean_t async);
 217 boolean_t dsl_dataset_block_freeable(dsl_dataset_t *ds, const blkptr_t *bp,
 218     uint64_t blk_birth);
 219 uint64_t dsl_dataset_prev_snap_txg(dsl_dataset_t *ds);

 221 void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
 222 void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);
 223 void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
 224 void dsl_dataset_space(dsl_dataset_t *ds,
 225     uint64_t *refdbytesp, uint64_t *availbytesp,
 226     uint64_t *usedobjsp, uint64_t *availobjsp);
 227 uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
 228 int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
 229     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
 230 int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
 231     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
 232 boolean_t dsl_dataset_is_dirty(dsl_dataset_t *ds);

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

 236 int dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
 237     uint64_t asize, uint64_t inflight, uint64_t *used,
 238     uint64_t *ref_rsrv);
 239 int dsl_dataset_set_refquota(const char *dsname, zprop_source_t source,
 240     uint64_t quota);
 241 int dsl_dataset_set_refreservation(const char *dsname, zprop_source_t source,
 242     uint64_t reservation);

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

 249 int dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
 250     dsl_dataset_t *origin_head, boolean_t force);
 251 void dsl_dataset_clone_swap_sync_impl(dsl_dataset_t *clone,
 252     dsl_dataset_t *origin_head, dmu_tx_t *tx);
 253 int dsl_dataset_snapshot_check_impl(dsl_dataset_t *ds, const char *snapname,
 254     dmu_tx_t *tx);
 255 void dsl_dataset_snapshot_sync_impl(dsl_dataset_t *ds, const char *snapname,
 256     dmu_tx_t *tx);

 258 void dsl_dataset_remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj,
 259     dmu_tx_t *tx);
 260 void dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds);
 261 int dsl_dataset_get_snapname(dsl_dataset_t *ds);
 262 int dsl_dataset_snap_lookup(dsl_dataset_t *ds, const char *name,
 263     uint64_t *value);
 264 int dsl_dataset_snap_remove(dsl_dataset_t *ds, const char *name, dmu_tx_t *tx);
 265 void dsl_dataset_set_refreservation_sync_impl(dsl_dataset_t *ds,
 266     zprop_source_t source, uint64_t value, dmu_tx_t *tx);
 267 int dsl_dataset_rollback(const char *fsname);

 269 #ifdef ZFS_DEBUG
 270 #define dprintf_ds(ds, fmt, ...) do { \
 271         if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
 272         char *__ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
 273         dsl_dataset_name(ds, __ds_name); \
```

```
 274         dprintf("ds=%s " fmt, __ds_name, __VA_ARGS__); \
 275         kmem_free(__ds_name, MAXNAMELEN); \
 276         } \
 277 _NOTE(CONSTCOND) } while (0)
 278 #else
 279 #define dprintf_ds(dd, fmt, ...)
 280 #endif

 282 #ifdef  __cplusplus
 283 }
```

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
}
#endif

_____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_***

```
4968 /*
4969  * innvl: {
4970  *     snapname -> { holdname, ... }
4971  *     ...
4972  * }
4973  *
4974  * outnvl: {
4975  *     snapname -> error value (int32)
4976  *     ...
4977  * }
4978  */
4979 /* ARGSUSED */
4980 static int
4981 zfs_ioc_release(const char *pool, nvlist_t *holds, nvlist_t *errlist)
4982 {
4983         nvpair_t *pair;

4985         /*
4986          * The release may cause the snapshot to be destroyed; make sure it
4987          * is not mounted.
4988          */
4989         for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
4990             pair = nvlist_next_nvpair(holds, pair))
4991                 zfs_unmount_snap(nvpair_name(pair));

4983         return (dsl_dataset_user_release(holds, errlist));
4984 }
```

_____***unchanged_portion_omitted_***