```
**********************************************************
  161089 Fri Oct 26 17:09:22 2012
new/usr/src/cmd/zfs/zfs_main.c
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
______unchanged_portion_omitted_

 201 #define NCOMMAND        (sizeof (command_table) / sizeof (command_table[0]))

 203 zfs_command_t *current_command;

 205 static const char *
 206 get_usage(zfs_help_t idx)
 207 {
 208         switch (idx) {
 209         case HELP_CLONE:
 210                 return (gettext("\tclone [-p] [-o property=value] ... "
 211                     "<snapshot> <filesystem|volume>\n"));
 212         case HELP_CREATE:
 213                 return (gettext("\tcreate [-p] [-o property=value] ... "
 214                     "<filesystem>\n"
 215                     "\tcreate [-ps] [-b blocksize] [-o property=value] ... "
 216                     "-V <size> <volume>\n"));
 217         case HELP_DESTROY:
 218                 return (gettext("\tdestroy [-fnpRrv] <filesystem|volume>\n"
 219                     "\tdestroy [-dnpRrv] "
 220                     "<filesystem|volume>@<snap>[%<snap>][,...]\n"));
 221         case HELP_GET:
 222                 return (gettext("\tget [-rHp] [-d max] "
 223                     "[-o \"all\" | field[,...]] [-t type[,...]] "
 224                     "[-s source[,...]]\n"
 225                     "\t    <\"all\" | property[,...]> "
 226                     "[filesystem|volume|snapshot] ...\n"));
 227         case HELP_INHERIT:
 228                 return (gettext("\tinherit [-rS] <property> "
 229                     "<filesystem|volume|snapshot> ...\n"));
 230         case HELP_UPGRADE:
 231                 return (gettext("\tupgrade [-v]\n"
 232                     "\tupgrade [-r] [-V version] <-a | filesystem ...>\n"));
 233         case HELP_LIST:
 234                 return (gettext("\tlist [-rH][-d max] "
 235                     "[-o property[,...]] [-t type[,...]] [-s property] ...\n"
 236                     "\t    [-S property] ... "
 237                     "[filesystem|volume|snapshot] ...\n"));
 238         case HELP_MOUNT:
 239                 return (gettext("\tmount\n"
 240                     "\tmount [-vO] [-o opts] <-a | filesystem>\n"));
 241         case HELP_PROMOTE:
 242                 return (gettext("\tpromote <clone-filesystem>\n"));
 243         case HELP_RECEIVE:
 244                 return (gettext("\treceive [-vnFu] <filesystem|volume|"
 245                 "snapshot>\n"
 246                 "\treceive [-vnFu] [-d | -e] <filesystem>\n"));
 247         case HELP_RENAME:
 248                 return (gettext("\trename [-f] <filesystem|volume|snapshot> "
 249                     "<filesystem|volume|snapshot>\n"
 250                     "\trename [-f] -p <filesystem|volume> <filesystem|volume>\n"
 251                     "\trename -r <snapshot> <snapshot>"));
 252         case HELP_ROLLBACK:
 253                 return (gettext("\trollback [-rRf] <snapshot>\n"));
 254         case HELP_SEND:
 255                 return (gettext("\tsend [-DnPpRv] [-[iI] snapshot] <snapshot>\n"
```



```
 256                     "\tsend -F [-nPRv] [-[iI] snapshot] <snapshot>\n"));
 255                 return (gettext("\tsend [-DnPpRv] [-[iI] snapshot] "
 256                     "<snapshot>\n"));
 257         case HELP_SET:
 258                 return (gettext("\tset <property=value> "
 259                     "<filesystem|volume|snapshot> ...\n"));
 260         case HELP_SHARE:
 261                 return (gettext("\tshare <-a | filesystem>\n"));
 262         case HELP_SNAPSHOT:
 263                 return (gettext("\tsnapshot [-r] [-o property=value] ... "
 264                     "<filesystem@snapname|volume@snapname> ...\n"));
 265         case HELP_UNMOUNT:
 266                 return (gettext("\tunmount [-f] "
 267                     "<-a | filesystem|mountpoint>\n"));
 268         case HELP_UNSHARE:
 269                 return (gettext("\tunshare "
 270                     "<-a | filesystem|mountpoint>\n"));
 271         case HELP_ALLOW:
 272                 return (gettext("\tallow <filesystem|volume>\n"
 273                     "\tallow [-ldug] "
 274                     "<\"everyone\"|user|group>[,...] <perm|@setname>[,...]\n"
 275                     "\t    <filesystem|volume>\n"
 276                     "\tallow [-ld] -e <perm|@setname>[,...] "
 277                     "<filesystem|volume>\n"
 278                     "\tallow -c <perm|@setname>[,...] <filesystem|volume>\n"
 279                     "\tallow -s @setname <perm|@setname>[,...] "
 280                     "<filesystem|volume>\n"));
 281         case HELP_UNALLOW:
 282                 return (gettext("\tunallow [-rldug] "
 283                     "<\"everyone\"|user|group>[,...]\n"
 284                     "\t    [<perm|@setname>[,...]] <filesystem|volume>\n"
 285                     "\tunallow [-rld] -e [<perm|@setname>[,...]] "
 286                     "<filesystem|volume>\n"
 287                     "\tunallow [-r] -c [<perm|@setname>[,...]] "
 288                     "<filesystem|volume>\n"
 289                     "\tunallow [-r] -s @setname [<perm|@setname>[,...]] "
 290                     "<filesystem|volume>\n"));
 291         case HELP_USERSPACE:
 292                 return (gettext("\tuserspace [-Hinp] [-o field[,...]] "
 293                     "[-s field] ...\n\t[-S field] ... "
 294                     "[-t type[,...]] <filesystem|snapshot>\n"));
 295         case HELP_GROUPSPACE:
 296                 return (gettext("\tgroupspace [-Hinp] [-o field[,...]] "
 297                     "[-s field] ...\n\t[-S field] ... "
 298                     "[-t type[,...]] <filesystem|snapshot>\n"));
 299         case HELP_HOLD:
 300                 return (gettext("\thold [-r] <tag> <snapshot> ...\n"));
 301         case HELP_HOLDS:
 302                 return (gettext("\tholds [-r] <snapshot> ...\n"));
 303         case HELP_RELEASE:
 304                 return (gettext("\trelease [-r] <tag> <snapshot> ...\n"));
 305         case HELP_DIFF:
 306                 return (gettext("\tdiff [-FHt] <snapshot> "
 307                     "[snapshot|filesystem]\n"));
 308         }

 310         abort();
 311         /* NOTREACHED */
 312 }
______unchanged_portion_omitted_

3508 /*
3509  * Send a backup stream to stdout.
3510  */
3511 static int
3512 zfs_do_send(int argc, char **argv)
```

```
3513 {
3514         char *fromname = NULL;
3515         char *toname = NULL;
3516         char *cp;
3517         zfs_handle_t *zhp;
3518         sendflags_t flags = { 0 };
3519         int c, err;
3520         nvlist_t *dbgnv = NULL;
3521         boolean_t extraverbose = B_FALSE;

3523         /* check options */
3524         while ((c = getopt(argc, argv, ":i:I:RDpvnPF")) != -1) {
3524         while ((c = getopt(argc, argv, ":i:I:RDpvnP")) != -1) {
3525                 switch (c) {
3526                 case 'i':
3527                         if (fromname)
3528                                 usage(B_FALSE);
3529                         fromname = optarg;
3530                         break;
3531                 case 'I':
3532                         if (fromname)
3533                                 usage(B_FALSE);
3534                         fromname = optarg;
3535                         flags.doall = B_TRUE;
3536                         break;
3537                 case 'R':
3538                         flags.replicate = B_TRUE;
3539                         break;
3540                 case 'p':
3541                         flags.props = B_TRUE;
3542                         break;
3543                 case 'P':
3544                         flags.parsable = B_TRUE;
3545                         flags.verbose = B_TRUE;
3546                         break;
3547                 case 'v':
3548                         if (flags.verbose)
3549                                 extraverbose = B_TRUE;
3550                         flags.verbose = B_TRUE;
3551                         flags.progress = B_TRUE;
3552                         break;
3553                 case 'D':
3554                         flags.dedup = B_TRUE;
3555                         break;
3556                 case 'n':
3557                         flags.dryrun = B_TRUE;
3558                         break;
3559                 case 'F':
3560                         flags.far = B_TRUE;
3561                         break;
3562 #endif /* ! codereview */
3563                 case ':':
3564                         (void) fprintf(stderr, gettext("missing argument for "
3565                             "'%c' option\n"), optopt);
3566                         usage(B_FALSE);
3567                         break;
3568                 case '?':
3569                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3570                             optopt);
3571                         usage(B_FALSE);
3572                 }
3573         }

3575         argc -= optind;
3576         argv += optind;
```



```
3578         /* check number of arguments */
3579         if (argc < 1) {
3580                 (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3581                 usage(B_FALSE);
3582         }
3583         if (argc > 1) {
3584                 (void) fprintf(stderr, gettext("too many arguments\n"));
3585                 usage(B_FALSE);
3586         }

3588         if (flags.far && (flags.dedup || flags.props)) {
3589                 (void) fprintf(stderr, gettext("options -D and -p are not "
3590                     "allowed with -F\n"));
3591                 usage(B_FALSE);
3592         }
3593 #endif /* ! codereview */
3594         if (!flags.dryrun && isatty(STDOUT_FILENO)) {
3595                 (void) fprintf(stderr,
3596                     gettext("Error: Stream can not be written to a terminal.\n"
3597                     "You must redirect standard output.\n"));
3598                 return (1);
3599         }

3601         cp = strchr(argv[0], '@');
3602         if (cp == NULL) {
3603                 (void) fprintf(stderr,
3604                     gettext("argument must be a snapshot\n"));
3605                 usage(B_FALSE);
3606         }
3607         *cp = '\0';
3608         toname = cp + 1;
3609         zhp = zfs_open(g_zfs, argv[0], ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
3610         if (zhp == NULL)
3611                 return (1);

3613         /*
3614          * If they specified the full path to the snapshot, chop off
3615          * everything except the short name of the snapshot, but special
3616          * case if they specify the origin.
3617          */
3618         if (fromname && (cp = strchr(fromname, '@')) != NULL) {
3619                 char origin[ZFS_MAXNAMELEN];
3620                 zprop_source_t src;

3622                 (void) zfs_prop_get(zhp, ZFS_PROP_ORIGIN,
3623                     origin, sizeof (origin), &src, NULL, 0, B_FALSE);

3625                 if (strcmp(origin, fromname) == 0) {
3626                         fromname = NULL;
3627                         flags.fromorigin = B_TRUE;
3628                 } else {
3629                         *cp = '\0';
3630                         if (cp != fromname && strcmp(argv[0], fromname)) {
3631                                 (void) fprintf(stderr,
3632                                     gettext("incremental source must be "
3633                                     "in same filesystem\n"));
3634                                 usage(B_FALSE);
3635                         }
3636                         fromname = cp + 1;
3637                         if (strchr(fromname, '@') || strchr(fromname, '/')) {
3638                                 (void) fprintf(stderr,
3639                                     gettext("invalid incremental source\n"));
3640                                 usage(B_FALSE);
3641                         }
3642                 }
3643         }
```

```
3645         if (flags.replicate && fromname == NULL)
3646                 flags.doall = B_TRUE;

3648         err = zfs_send(zhp, fromname, toname, &flags, STDOUT_FILENO, NULL, 0,
3649             extraverbose ? &dbgnv : NULL);

3651         if (extraverbose && dbgnv != NULL) {
3652                 /*
3653                  * dump_nvlist prints to stdout, but that's been
3654                  * redirected to a file.  Make it print to stderr
3655                  * instead.
3656                  */
3657                 (void) dup2(STDERR_FILENO, STDOUT_FILENO);
3658                 dump_nvlist(dbgnv, 0);
3659                 nvlist_free(dbgnv);
3660         }
3661         zfs_close(zhp);

3663         return (err != 0);
3664 }

3666 /*
3667  * zfs receive [-vnFu] [-d | -e] <fs@snap>
3668  *
3669  * Restore a backup stream from stdin.
3670  */
3671 static int
3672 zfs_do_receive(int argc, char **argv)
3673 {
3674         int c, err;
3675         recvflags_t flags = { 0 };

3677         /* check options */
3678         while ((c = getopt(argc, argv, ":denuvF")) != -1) {
3679                 switch (c) {
3680                 case 'd':
3681                         flags.isprefix = B_TRUE;
3682                         break;
3683                 case 'e':
3684                         flags.isprefix = B_TRUE;
3685                         flags.istail = B_TRUE;
3686                         break;
3687                 case 'n':
3688                         flags.dryrun = B_TRUE;
3689                         break;
3690                 case 'u':
3691                         flags.nomount = B_TRUE;
3692                         break;
3693                 case 'v':
3694                         flags.verbose = B_TRUE;
3695                         break;
3696                 case 'F':
3697                         flags.force = B_TRUE;
3698                         break;
3699                 case ':':
3700                         (void) fprintf(stderr, gettext("missing argument for "
3701                             "'%c' option\n"), optopt);
3702                         usage(B_FALSE);
3703                         break;
3704                 case '?':
3705                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3706                             optopt);
3707                         usage(B_FALSE);
3708                 }
3709         }
```



```
3711         argc -= optind;
3712         argv += optind;

3714         /* check number of arguments */
3715         if (argc < 1) {
3716                 (void) fprintf(stderr, gettext("missing snapshot argument\n"));
3717                 usage(B_FALSE);
3718         }
3719         if (argc > 1) {
3720                 (void) fprintf(stderr, gettext("too many arguments\n"));
3721                 usage(B_FALSE);
3722         }

3724         if (isatty(STDIN_FILENO)) {
3725                 (void) fprintf(stderr,
3726                     gettext("Error: Backup stream can not be read "
3727                     "from a terminal.\n"
3728                     "You must redirect standard input.\n"));
3729                 return (1);
3730         }

3732         err = zfs_receive(g_zfs, argv[0], &flags, STDIN_FILENO, NULL);

3734         return (err != 0);
3735 }

3737 /*
3738  * allow/unallow stuff
3739  */
3740 /* copied from zfs/sys/dsl_deleg.h */
3741 #define ZFS_DELEG_PERM_CREATE           "create"
3742 #define ZFS_DELEG_PERM_DESTROY          "destroy"
3743 #define ZFS_DELEG_PERM_SNAPSHOT         "snapshot"
3744 #define ZFS_DELEG_PERM_ROLLBACK         "rollback"
3745 #define ZFS_DELEG_PERM_CLONE            "clone"
3746 #define ZFS_DELEG_PERM_PROMOTE          "promote"
3747 #define ZFS_DELEG_PERM_RENAME           "rename"
3748 #define ZFS_DELEG_PERM_MOUNT            "mount"
3749 #define ZFS_DELEG_PERM_SHARE            "share"
3750 #define ZFS_DELEG_PERM_SEND             "send"
3751 #define ZFS_DELEG_PERM_RECEIVE          "receive"
3752 #define ZFS_DELEG_PERM_ALLOW            "allow"
3753 #define ZFS_DELEG_PERM_USERPROP         "userprop"
3754 #define ZFS_DELEG_PERM_VSCAN            "vscan" /* ??? */
3755 #define ZFS_DELEG_PERM_USERQUOTA        "userquota"
3756 #define ZFS_DELEG_PERM_GROUPQUOTA       "groupquota"
3757 #define ZFS_DELEG_PERM_USERUSED         "userused"
3758 #define ZFS_DELEG_PERM_GROUPUSED        "groupused"
3759 #define ZFS_DELEG_PERM_HOLD             "hold"
3760 #define ZFS_DELEG_PERM_RELEASE          "release"
3761 #define ZFS_DELEG_PERM_DIFF             "diff"

3763 #define ZFS_NUM_DELEG_NOTES ZFS_DELEG_NOTE_NONE

3765 static zfs_deleg_perm_tab_t zfs_deleg_perm_tbl[] = {
3766         { ZFS_DELEG_PERM_ALLOW, ZFS_DELEG_NOTE_ALLOW },
3767         { ZFS_DELEG_PERM_CLONE, ZFS_DELEG_NOTE_CLONE },
3768         { ZFS_DELEG_PERM_CREATE, ZFS_DELEG_NOTE_CREATE },
3769         { ZFS_DELEG_PERM_DESTROY, ZFS_DELEG_NOTE_DESTROY },
3770         { ZFS_DELEG_PERM_DIFF, ZFS_DELEG_NOTE_DIFF},
3771         { ZFS_DELEG_PERM_HOLD, ZFS_DELEG_NOTE_HOLD },
3772         { ZFS_DELEG_PERM_MOUNT, ZFS_DELEG_NOTE_MOUNT },
3773         { ZFS_DELEG_PERM_PROMOTE, ZFS_DELEG_NOTE_PROMOTE },
3774         { ZFS_DELEG_PERM_RECEIVE, ZFS_DELEG_NOTE_RECEIVE },
3775         { ZFS_DELEG_PERM_RELEASE, ZFS_DELEG_NOTE_RELEASE },
```

```
3776         { ZFS_DELEG_PERM_RENAME, ZFS_DELEG_NOTE_RENAME },
3777         { ZFS_DELEG_PERM_ROLLBACK, ZFS_DELEG_NOTE_ROLLBACK },
3778         { ZFS_DELEG_PERM_SEND, ZFS_DELEG_NOTE_SEND },
3779         { ZFS_DELEG_PERM_SHARE, ZFS_DELEG_NOTE_SHARE },
3780         { ZFS_DELEG_PERM_SNAPSHOT, ZFS_DELEG_NOTE_SNAPSHOT },

3782         { ZFS_DELEG_PERM_GROUPQUOTA, ZFS_DELEG_NOTE_GROUPQUOTA },
3783         { ZFS_DELEG_PERM_GROUPUSED, ZFS_DELEG_NOTE_GROUPUSED },
3784         { ZFS_DELEG_PERM_USERPROP, ZFS_DELEG_NOTE_USERPROP },
3785         { ZFS_DELEG_PERM_USERQUOTA, ZFS_DELEG_NOTE_USERQUOTA },
3786         { ZFS_DELEG_PERM_USERUSED, ZFS_DELEG_NOTE_USERUSED },
3787         { NULL, ZFS_DELEG_NOTE_NONE }
3788 };

3790 /* permission structure */
3791 typedef struct deleg_perm {
3792         zfs_deleg_who_type_t    dp_who_type;
3793         const char              *dp_name;
3794         boolean_t               dp_local;
3795         boolean_t               dp_descend;
3796 } deleg_perm_t;

3798 /* */
3799 typedef struct deleg_perm_node {
3800         deleg_perm_t            dpn_perm;

3802         uu_avl_node_t           dpn_avl_node;
3803 } deleg_perm_node_t;

3805 typedef struct fs_perm fs_perm_t;

3807 /* permissions set */
3808 typedef struct who_perm {
3809         zfs_deleg_who_type_t    who_type;
3810         const char              *who_name;              /* id */
3811         char                    who_ug_name[256];       /* user/group name */
3812         fs_perm_t               *who_fsperm;            /* uplink */

3814         uu_avl_t                *who_deleg_perm_avl;    /* permissions */
3815 } who_perm_t;

3817 /* */
3818 typedef struct who_perm_node {
3819         who_perm_t      who_perm;
3820         uu_avl_node_t   who_avl_node;
3821 } who_perm_node_t;

3823 typedef struct fs_perm_set fs_perm_set_t;
3824 /* fs permissions */
3825 struct fs_perm {
3826         const char              *fsp_name;

3828         uu_avl_t                *fsp_sc_avl;    /* sets,create */
3829         uu_avl_t                *fsp_uge_avl;   /* user,group,everyone */

3831         fs_perm_set_t           *fsp_set;       /* uplink */
3832 };

3834 /* */
3835 typedef struct fs_perm_node {
3836         fs_perm_t       fspn_fsperm;
3837         uu_avl_t        *fspn_avl;

3839         uu_list_node_t  fspn_list_node;
3840 } fs_perm_node_t;
```



```
3842 /* top level structure */
3843 struct fs_perm_set {
3844         uu_list_pool_t  *fsps_list_pool;
3845         uu_list_t       *fsps_list; /* list of fs_perms */

3847         uu_avl_pool_t   *fsps_named_set_avl_pool;
3848         uu_avl_pool_t   *fsps_who_perm_avl_pool;
3849         uu_avl_pool_t   *fsps_deleg_perm_avl_pool;
3850 };

3852 static inline const char *
3853 deleg_perm_type(zfs_deleg_note_t note)
3854 {
3855         /* subcommands */
3856         switch (note) {
3857                 /* SUBCOMMANDS */
3858                 /* OTHER */
3859         case ZFS_DELEG_NOTE_GROUPQUOTA:
3860         case ZFS_DELEG_NOTE_GROUPUSED:
3861         case ZFS_DELEG_NOTE_USERPROP:
3862         case ZFS_DELEG_NOTE_USERQUOTA:
3863         case ZFS_DELEG_NOTE_USERUSED:
3864                 /* other */
3865                 return (gettext("other"));
3866         default:
3867                 return (gettext("subcommand"));
3868         }
3869 }

3871 static int inline
3872 who_type2weight(zfs_deleg_who_type_t who_type)
3873 {
3874         int res;
3875         switch (who_type) {
3876                 case ZFS_DELEG_NAMED_SET_SETS:
3877                 case ZFS_DELEG_NAMED_SET:
3878                         res = 0;
3879                         break;
3880                 case ZFS_DELEG_CREATE_SETS:
3881                 case ZFS_DELEG_CREATE:
3882                         res = 1;
3883                         break;
3884                 case ZFS_DELEG_USER_SETS:
3885                 case ZFS_DELEG_USER:
3886                         res = 2;
3887                         break;
3888                 case ZFS_DELEG_GROUP_SETS:
3889                 case ZFS_DELEG_GROUP:
3890                         res = 3;
3891                         break;
3892                 case ZFS_DELEG_EVERYONE_SETS:
3893                 case ZFS_DELEG_EVERYONE:
3894                         res = 4;
3895                         break;
3896                 default:
3897                         res = -1;
3898         }

3900         return (res);
3901 }

3903 /* ARGSUSED */
3904 static int
3905 who_perm_compare(const void *larg, const void *rarg, void *unused)
3906 {
3907         const who_perm_node_t *l = larg;
```

```
3908         const who_perm_node_t *r = rarg;
3909         zfs_deleg_who_type_t ltype = l->who_perm.who_type;
3910         zfs_deleg_who_type_t rtype = r->who_perm.who_type;
3911         int lweight = who_type2weight(ltype);
3912         int rweight = who_type2weight(rtype);
3913         int res = lweight - rweight;
3914         if (res == 0)
3915                 res = strncmp(l->who_perm.who_name, r->who_perm.who_name,
3916                     ZFS_MAX_DELEG_NAME-1);

3918         if (res == 0)
3919                 return (0);
3920         if (res > 0)
3921                 return (1);
3922         else
3923                 return (-1);
3924 }

3926 /* ARGSUSED */
3927 static int
3928 deleg_perm_compare(const void *larg, const void *rarg, void *unused)
3929 {
3930         const deleg_perm_node_t *l = larg;
3931         const deleg_perm_node_t *r = rarg;
3932         int res =  strncmp(l->dpn_perm.dp_name, r->dpn_perm.dp_name,
3933             ZFS_MAX_DELEG_NAME-1);

3935         if (res == 0)
3936                 return (0);

3938         if (res > 0)
3939                 return (1);
3940         else
3941                 return (-1);
3942 }

3944 static inline void
3945 fs_perm_set_init(fs_perm_set_t *fspset)
3946 {
3947         bzero(fspset, sizeof (fs_perm_set_t));

3949         if ((fspset->fsps_list_pool = uu_list_pool_create("fsps_list_pool",
3950             sizeof (fs_perm_node_t), offsetof(fs_perm_node_t, fspn_list_node),
3951             NULL, UU_DEFAULT)) == NULL)
3952                 nomem();
3953         if ((fspset->fsps_list = uu_list_create(fspset->fsps_list_pool, NULL,
3954             UU_DEFAULT)) == NULL)
3955                 nomem();

3957         if ((fspset->fsps_named_set_avl_pool = uu_avl_pool_create(
3958             "named_set_avl_pool", sizeof (who_perm_node_t), offsetof(
3959             who_perm_node_t, who_avl_node), who_perm_compare,
3960             UU_DEFAULT)) == NULL)
3961                 nomem();

3963         if ((fspset->fsps_who_perm_avl_pool = uu_avl_pool_create(
3964             "who_perm_avl_pool", sizeof (who_perm_node_t), offsetof(
3965             who_perm_node_t, who_avl_node), who_perm_compare,
3966             UU_DEFAULT)) == NULL)
3967                 nomem();

3969         if ((fspset->fsps_deleg_perm_avl_pool = uu_avl_pool_create(
3970             "deleg_perm_avl_pool", sizeof (deleg_perm_node_t), offsetof(
3971             deleg_perm_node_t, dpn_avl_node), deleg_perm_compare, UU_DEFAULT))
3972             == NULL)
3973                 nomem();
```



```
3974 }

3976 static inline void fs_perm_fini(fs_perm_t *);
3977 static inline void who_perm_fini(who_perm_t *);

3979 static inline void
3980 fs_perm_set_fini(fs_perm_set_t *fspset)
3981 {
3982         fs_perm_node_t *node = uu_list_first(fspset->fsps_list);

3984         while (node != NULL) {
3985                 fs_perm_node_t *next_node =
3986                     uu_list_next(fspset->fsps_list, node);
3987                 fs_perm_t *fsperm = &node->fspn_fsperm;
3988                 fs_perm_fini(fsperm);
3989                 uu_list_remove(fspset->fsps_list, node);
3990                 free(node);
3991                 node = next_node;
3992         }

3994         uu_avl_pool_destroy(fspset->fsps_named_set_avl_pool);
3995         uu_avl_pool_destroy(fspset->fsps_who_perm_avl_pool);
3996         uu_avl_pool_destroy(fspset->fsps_deleg_perm_avl_pool);
3997 }

3999 static inline void
4000 deleg_perm_init(deleg_perm_t *deleg_perm, zfs_deleg_who_type_t type,
4001     const char *name)
4002 {
4003         deleg_perm->dp_who_type = type;
4004         deleg_perm->dp_name = name;
4005 }

4007 static inline void
4008 who_perm_init(who_perm_t *who_perm, fs_perm_t *fsperm,
4009     zfs_deleg_who_type_t type, const char *name)
4010 {
4011         uu_avl_pool_t   *pool;
4012         pool = fsperm->fsp_set->fsps_deleg_perm_avl_pool;

4014         bzero(who_perm, sizeof (who_perm_t));

4016         if ((who_perm->who_deleg_perm_avl = uu_avl_create(pool, NULL,
4017             UU_DEFAULT)) == NULL)
4018                 nomem();

4020         who_perm->who_type = type;
4021         who_perm->who_name = name;
4022         who_perm->who_fsperm = fsperm;
4023 }

4025 static inline void
4026 who_perm_fini(who_perm_t *who_perm)
4027 {
4028         deleg_perm_node_t *node = uu_avl_first(who_perm->who_deleg_perm_avl);

4030         while (node != NULL) {
4031                 deleg_perm_node_t *next_node =
4032                     uu_avl_next(who_perm->who_deleg_perm_avl, node);

4034                 uu_avl_remove(who_perm->who_deleg_perm_avl, node);
4035                 free(node);
4036                 node = next_node;
4037         }

4039         uu_avl_destroy(who_perm->who_deleg_perm_avl);
```

```
4040 }

4042 static inline void
4043 fs_perm_init(fs_perm_t *fsperm, fs_perm_set_t *fspset, const char *fsname)
4044 {
4045         uu_avl_pool_t   *nset_pool = fspset->fsps_named_set_avl_pool;
4046         uu_avl_pool_t   *who_pool = fspset->fsps_who_perm_avl_pool;

4048         bzero(fsperm, sizeof (fs_perm_t));

4050         if ((fsperm->fsp_sc_avl = uu_avl_create(nset_pool, NULL, UU_DEFAULT))
4051             == NULL)
4052                 nomem();

4054         if ((fsperm->fsp_uge_avl = uu_avl_create(who_pool, NULL, UU_DEFAULT))
4055             == NULL)
4056                 nomem();

4058         fsperm->fsp_set = fspset;
4059         fsperm->fsp_name = fsname;
4060 }

4062 static inline void
4063 fs_perm_fini(fs_perm_t *fsperm)
4064 {
4065         who_perm_node_t *node = uu_avl_first(fsperm->fsp_sc_avl);
4066         while (node != NULL) {
4067                 who_perm_node_t *next_node = uu_avl_next(fsperm->fsp_sc_avl,
4068                     node);
4069                 who_perm_t *who_perm = &node->who_perm;
4070                 who_perm_fini(who_perm);
4071                 uu_avl_remove(fsperm->fsp_sc_avl, node);
4072                 free(node);
4073                 node = next_node;
4074         }

4076         node = uu_avl_first(fsperm->fsp_uge_avl);
4077         while (node != NULL) {
4078                 who_perm_node_t *next_node = uu_avl_next(fsperm->fsp_uge_avl,
4079                     node);
4080                 who_perm_t *who_perm = &node->who_perm;
4081                 who_perm_fini(who_perm);
4082                 uu_avl_remove(fsperm->fsp_uge_avl, node);
4083                 free(node);
4084                 node = next_node;
4085         }

4087         uu_avl_destroy(fsperm->fsp_sc_avl);
4088         uu_avl_destroy(fsperm->fsp_uge_avl);
4089 }

4091 static void inline
4092 set_deleg_perm_node(uu_avl_t *avl, deleg_perm_node_t *node,
4093     zfs_deleg_who_type_t who_type, const char *name, char locality)
4094 {
4095         uu_avl_index_t idx = 0;

4097         deleg_perm_node_t *found_node = NULL;
4098         deleg_perm_t    *deleg_perm = &node->dpn_perm;

4100         deleg_perm_init(deleg_perm, who_type, name);

4102         if ((found_node = uu_avl_find(avl, node, NULL, &idx))
4103             == NULL)
4104                 uu_avl_insert(avl, node, idx);
4105         else {
```



```
4106                 node = found_node;
4107                 deleg_perm = &node->dpn_perm;
4108         }


4111         switch (locality) {
4112         case ZFS_DELEG_LOCAL:
4113                 deleg_perm->dp_local = B_TRUE;
4114                 break;
4115         case ZFS_DELEG_DESCENDENT:
4116                 deleg_perm->dp_descend = B_TRUE;
4117                 break;
4118         case ZFS_DELEG_NA:
4119                 break;
4120         default:
4121                 assert(B_FALSE); /* invalid locality */
4122         }
4123 }

4125 static inline int
4126 parse_who_perm(who_perm_t *who_perm, nvlist_t *nvl, char locality)
4127 {
4128         nvpair_t *nvp = NULL;
4129         fs_perm_set_t *fspset = who_perm->who_fsperm->fsp_set;
4130         uu_avl_t *avl = who_perm->who_deleg_perm_avl;
4131         zfs_deleg_who_type_t who_type = who_perm->who_type;

4133         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
4134                 const char *name = nvpair_name(nvp);
4135                 data_type_t type = nvpair_type(nvp);
4136                 uu_avl_pool_t *avl_pool = fspset->fsps_deleg_perm_avl_pool;
4137                 deleg_perm_node_t *node =
4138                     safe_malloc(sizeof (deleg_perm_node_t));

4140                 assert(type == DATA_TYPE_BOOLEAN);

4142                 uu_avl_node_init(node, &node->dpn_avl_node, avl_pool);
4143                 set_deleg_perm_node(avl, node, who_type, name, locality);
4144         }

4146         return (0);
4147 }

4149 static inline int
4150 parse_fs_perm(fs_perm_t *fsperm, nvlist_t *nvl)
4151 {
4152         nvpair_t *nvp = NULL;
4153         fs_perm_set_t *fspset = fsperm->fsp_set;

4155         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
4156                 nvlist_t *nvl2 = NULL;
4157                 const char *name = nvpair_name(nvp);
4158                 uu_avl_t *avl = NULL;
4159                 uu_avl_pool_t *avl_pool;
4160                 zfs_deleg_who_type_t perm_type = name[0];
4161                 char perm_locality = name[1];
4162                 const char *perm_name = name + 3;
4163                 boolean_t is_set = B_TRUE;
4164                 who_perm_t *who_perm = NULL;

4166                 assert('$' == name[2]);

4168                 if (nvpair_value_nvlist(nvp, &nvl2) != 0)
4169                         return (-1);

4171                 switch (perm_type) {
```

```
4172                 case ZFS_DELEG_CREATE:
4173                 case ZFS_DELEG_CREATE_SETS:
4174                 case ZFS_DELEG_NAMED_SET:
4175                 case ZFS_DELEG_NAMED_SET_SETS:
4176                         avl_pool = fspset->fsps_named_set_avl_pool;
4177                         avl = fsperm->fsp_sc_avl;
4178                         break;
4179                 case ZFS_DELEG_USER:
4180                 case ZFS_DELEG_USER_SETS:
4181                 case ZFS_DELEG_GROUP:
4182                 case ZFS_DELEG_GROUP_SETS:
4183                 case ZFS_DELEG_EVERYONE:
4184                 case ZFS_DELEG_EVERYONE_SETS:
4185                         avl_pool = fspset->fsps_who_perm_avl_pool;
4186                         avl = fsperm->fsp_uge_avl;
4187                         break;
4188                 }

4190                 if (is_set) {
4191                         who_perm_node_t *found_node = NULL;
4192                         who_perm_node_t *node = safe_malloc(
4193                             sizeof (who_perm_node_t));
4194                         who_perm = &node->who_perm;
4195                         uu_avl_index_t idx = 0;

4197                         uu_avl_node_init(node, &node->who_avl_node, avl_pool);
4198                         who_perm_init(who_perm, fsperm, perm_type, perm_name);

4200                         if ((found_node = uu_avl_find(avl, node, NULL, &idx))
4201                             == NULL) {
4202                                 if (avl == fsperm->fsp_uge_avl) {
4203                                         uid_t rid = 0;
4204                                         struct passwd *p = NULL;
4205                                         struct group *g = NULL;
4206                                         const char *nice_name = NULL;

4208                                         switch (perm_type) {
4209                                         case ZFS_DELEG_USER_SETS:
4210                                         case ZFS_DELEG_USER:
4211                                                 rid = atoi(perm_name);
4212                                                 p = getpwuid(rid);
4213                                                 if (p)
4214                                                         nice_name = p->pw_name;
4215                                                 break;
4216                                         case ZFS_DELEG_GROUP_SETS:
4217                                         case ZFS_DELEG_GROUP:
4218                                                 rid = atoi(perm_name);
4219                                                 g = getgrgid(rid);
4220                                                 if (g)
4221                                                         nice_name = g->gr_name;
4222                                                 break;
4223                                         }

4225                                         if (nice_name != NULL)
4226                                                 (void) strlcpy(
4227                                                     node->who_perm.who_ug_name,
4228                                                     nice_name, 256);
4229                                 }

4231                                 uu_avl_insert(avl, node, idx);
4232                         } else {
4233                                 node = found_node;
4234                                 who_perm = &node->who_perm;
4235                         }
4236                 }
```



```
4238                         (void) parse_who_perm(who_perm, nvl2, perm_locality);
4239                 }

4241                 return (0);
4242         }

4244         static inline int
4245         parse_fs_perm_set(fs_perm_set_t *fspset, nvlist_t *nvl)
4246         {
4247                 nvpair_t *nvp = NULL;
4248                 uu_avl_index_t idx = 0;

4250                 while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
4251                         nvlist_t *nvl2 = NULL;
4252                         const char *fsname = nvpair_name(nvp);
4253                         data_type_t type = nvpair_type(nvp);
4254                         fs_perm_t *fsperm = NULL;
4255                         fs_perm_node_t *node = safe_malloc(sizeof (fs_perm_node_t));
4256                         if (node == NULL)
4257                                 nomem();

4259                         fsperm = &node->fspn_fsperm;

4261                         assert(DATA_TYPE_NVLIST == type);

4263                         uu_list_node_init(node, &node->fspn_list_node,
4264                             fspset->fsps_list_pool);

4266                         idx = uu_list_numnodes(fspset->fsps_list);
4267                         fs_perm_init(fsperm, fspset, fsname);

4269                         if (nvpair_value_nvlist(nvp, &nvl2) != 0)
4270                                 return (-1);

4272                         (void) parse_fs_perm(fsperm, nvl2);

4274                         uu_list_insert(fspset->fsps_list, node, idx);
4275                 }

4277                 return (0);
4278         }

4280         static inline const char *
4281         deleg_perm_comment(zfs_deleg_note_t note)
4282         {
4283                 const char *str = "";

4285                 /* subcommands */
4286                 switch (note) {
4287                         /* SUBCOMMANDS */
4288                 case ZFS_DELEG_NOTE_ALLOW:
4289                         str = gettext("Must also have the permission that is being"
4290                             "\n\t\t\t\tallowed");
4291                         break;
4292                 case ZFS_DELEG_NOTE_CLONE:
4293                         str = gettext("Must also have the 'create' ability and 'mount'"
4294                             "\n\t\t\t\tability in the origin file system");
4295                         break;
4296                 case ZFS_DELEG_NOTE_CREATE:
4297                         str = gettext("Must also have the 'mount' ability");
4298                         break;
4299                 case ZFS_DELEG_NOTE_DESTROY:
4300                         str = gettext("Must also have the 'mount' ability");
4301                         break;
4302                 case ZFS_DELEG_NOTE_DIFF:
4303                         str = gettext("Allows lookup of paths within a dataset;"
```

```
4304                     "\n\t\t\t\tgiven an object number. Ordinary users need this"
4305                     "\n\t\t\t\tin order to use zfs diff");
4306                 break;
4307         case ZFS_DELEG_NOTE_HOLD:
4308                 str = gettext("Allows adding a user hold to a snapshot");
4309                 break;
4310         case ZFS_DELEG_NOTE_MOUNT:
4311                 str = gettext("Allows mount/umount of ZFS datasets");
4312                 break;
4313         case ZFS_DELEG_NOTE_PROMOTE:
4314                 str = gettext("Must also have the 'mount'\n\t\t\t\tand"
4315                     " 'promote' ability in the origin file system");
4316                 break;
4317         case ZFS_DELEG_NOTE_RECEIVE:
4318                 str = gettext("Must also have the 'mount' and 'create'"
4319                     " ability");
4320                 break;
4321         case ZFS_DELEG_NOTE_RELEASE:
4322                 str = gettext("Allows releasing a user hold which\n\t\t\t\t"
4323                     "might destroy the snapshot");
4324                 break;
4325         case ZFS_DELEG_NOTE_RENAME:
4326                 str = gettext("Must also have the 'mount' and 'create'"
4327                     "\n\t\t\t\tability in the new parent");
4328                 break;
4329         case ZFS_DELEG_NOTE_ROLLBACK:
4330                 str = gettext("");
4331                 break;
4332         case ZFS_DELEG_NOTE_SEND:
4333                 str = gettext("");
4334                 break;
4335         case ZFS_DELEG_NOTE_SHARE:
4336                 str = gettext("Allows sharing file systems over NFS or SMB"
4337                     "\n\t\t\t\tprotocols");
4338                 break;
4339         case ZFS_DELEG_NOTE_SNAPSHOT:
4340                 str = gettext("");
4341                 break;
4342 /*
4343  *      case ZFS_DELEG_NOTE_VSCAN:
4344  *              str = gettext("");
4345  *              break;
4346  */
4347                 /* OTHER */
4348         case ZFS_DELEG_NOTE_GROUPQUOTA:
4349                 str = gettext("Allows accessing any groupquota@... property");
4350                 break;
4351         case ZFS_DELEG_NOTE_GROUPUSED:
4352                 str = gettext("Allows reading any groupused@... property");
4353                 break;
4354         case ZFS_DELEG_NOTE_USERPROP:
4355                 str = gettext("Allows changing any user property");
4356                 break;
4357         case ZFS_DELEG_NOTE_USERQUOTA:
4358                 str = gettext("Allows accessing any userquota@... property");
4359                 break;
4360         case ZFS_DELEG_NOTE_USERUSED:
4361                 str = gettext("Allows reading any userused@... property");
4362                 break;
4363                 /* other */
4364         default:
4365                 str = "";
4366         }

4368         return (str);
4369 }
```



```
4371 struct allow_opts {
4372         boolean_t local;
4373         boolean_t descend;
4374         boolean_t user;
4375         boolean_t group;
4376         boolean_t everyone;
4377         boolean_t create;
4378         boolean_t set;
4379         boolean_t recursive; /* unallow only */
4380         boolean_t prt_usage;

4382         boolean_t prt_perms;
4383         char *who;
4384         char *perms;
4385         const char *dataset;
4386 };

4388 static inline int
4389 prop_cmp(const void *a, const void *b)
4390 {
4391         const char *str1 = *(const char **)a;
4392         const char *str2 = *(const char **)b;
4393         return (strcmp(str1, str2));
4394 }

4396 static void
4397 allow_usage(boolean_t un, boolean_t requested, const char *msg)
4398 {
4399         const char *opt_desc[] = {
4400                 "-h", gettext("show this help message and exit"),
4401                 "-l", gettext("set permission locally"),
4402                 "-d", gettext("set permission for descents"),
4403                 "-u", gettext("set permission for user"),
4404                 "-g", gettext("set permission for group"),
4405                 "-e", gettext("set permission for everyone"),
4406                 "-c", gettext("set create time permission"),
4407                 "-s", gettext("define permission set"),
4408                 /* unallow only */
4409                 "-r", gettext("remove permissions recursively"),
4410         };
4411         size_t unallow_size = sizeof (opt_desc) / sizeof (char *);
4412         size_t allow_size = unallow_size - 2;
4413         const char *props[ZFS_NUM_PROPS];
4414         int i;
4415         size_t count = 0;
4416         FILE *fp = requested ? stdout : stderr;
4417         zprop_desc_t *pdtbl = zfs_prop_get_table();
4418         const char *fmt = gettext("%-16s %-14s\t%s\n");

4420         (void) fprintf(fp, gettext("Usage: %s\n"), get_usage(un ? HELP_UNALLOW :
4421             HELP_ALLOW));
4422         (void) fprintf(fp, gettext("Options:\n"));
4423         for (int i = 0; i < (un ? unallow_size : allow_size); i++) {
4424                 const char *opt = opt_desc[i++];
4425                 const char *optdsc = opt_desc[i];
4426                 (void) fprintf(fp, gettext("  %-10s  %s\n"), opt, optdsc);
4427         }

4429         (void) fprintf(fp, gettext("\nThe following permissions are "
4430             "supported:\n\n"));
4431         (void) fprintf(fp, fmt, gettext("NAME"), gettext("TYPE"),
4432             gettext("NOTES"));
4433         for (i = 0; i < ZFS_NUM_DELEG_NOTES; i++) {
4434                 const char *perm_name = zfs_deleg_perm_tbl[i].z_perm;
4435                 zfs_deleg_note_t perm_note = zfs_deleg_perm_tbl[i].z_note;
```

```
4436                const char *perm_type = deleg_perm_type(perm_note);
4437                const char *perm_comment = deleg_perm_comment(perm_note);
4438                (void) fprintf(fp, fmt, perm_name, perm_type, perm_comment);
4439        }

4441        for (i = 0; i < ZFS_NUM_PROPS; i++) {
4442                zprop_desc_t *pd = &pdtbl[i];
4443                if (pd->pd_visible != B_TRUE)
4444                        continue;

4446                if (pd->pd_attr == PROP_READONLY)
4447                        continue;

4449                props[count++] = pd->pd_name;
4450        }
4451        props[count] = NULL;

4453        qsort(props, count, sizeof (char *), prop_cmp);

4455        for (i = 0; i < count; i++)
4456                (void) fprintf(fp, fmt, props[i], gettext("property"), "");

4458        if (msg != NULL)
4459                (void) fprintf(fp, gettext("\nzfs: error: %s"), msg);

4461        exit(requested ? 0 : 2);
4462 }

4464 static inline const char *
4465 munge_args(int argc, char **argv, boolean_t un, size_t expected_argc,
4466     char **permsp)
4467 {
4468        if (un && argc == expected_argc - 1)
4469                *permsp = NULL;
4470        else if (argc == expected_argc)
4471                *permsp = argv[argc - 2];
4472        else
4473                allow_usage(un, B_FALSE,
4474                    gettext("wrong number of parameters\n"));

4476        return (argv[argc - 1]);
4477 }

4479 static void
4480 parse_allow_args(int argc, char **argv, boolean_t un, struct allow_opts *opts)
4481 {
4482        int uge_sum = opts->user + opts->group + opts->everyone;
4483        int csuge_sum = opts->create + opts->set + uge_sum;
4484        int ldcsuge_sum = csuge_sum + opts->local + opts->descend;
4485        int all_sum = un ? ldcsuge_sum + opts->recursive : ldcsuge_sum;

4487        if (uge_sum > 1)
4488                allow_usage(un, B_FALSE,
4489                    gettext("-u, -g, and -e are mutually exclusive\n"));

4491        if (opts->prt_usage)
4492                if (argc == 0 && all_sum == 0)
4493                        allow_usage(un, B_TRUE, NULL);
4494                else
4495                        usage(B_FALSE);

4497        if (opts->set) {
4498                if (csuge_sum > 1)
4499                        allow_usage(un, B_FALSE,
4500                            gettext("invalid options combined with -s\n"));
```



```
4502                opts->dataset = munge_args(argc, argv, un, 3, &opts->perms);
4503                if (argv[0][0] != '@')
4504                        allow_usage(un, B_FALSE,
4505                            gettext("invalid set name: missing '@' prefix\n"));
4506                opts->who = argv[0];
4507        } else if (opts->create) {
4508                if (ldcsuge_sum > 1)
4509                        allow_usage(un, B_FALSE,
4510                            gettext("invalid options combined with -c\n"));
4511                opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4512        } else if (opts->everyone) {
4513                if (csuge_sum > 1)
4514                        allow_usage(un, B_FALSE,
4515                            gettext("invalid options combined with -e\n"));
4516                opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4517        } else if (uge_sum == 0 && argc > 0 && strcmp(argv[0], "everyone")
4518            == 0) {
4519                opts->everyone = B_TRUE;
4520                argc--;
4521                argv++;
4522                opts->dataset = munge_args(argc, argv, un, 2, &opts->perms);
4523        } else if (argc == 1 && !un) {
4524                opts->prt_perms = B_TRUE;
4525                opts->dataset = argv[argc-1];
4526        } else {
4527                opts->dataset = munge_args(argc, argv, un, 3, &opts->perms);
4528                opts->who = argv[0];
4529        }

4531        if (!opts->local && !opts->descend) {
4532                opts->local = B_TRUE;
4533                opts->descend = B_TRUE;
4534        }
4535 }

4537 static void
4538 store_allow_perm(zfs_deleg_who_type_t type, boolean_t local, boolean_t descend,
4539     const char *who, char *perms, nvlist_t *top_nvl)
4540 {
4541        int i;
4542        char ld[2] = { '\0', '\0' };
4543        char who_buf[ZFS_MAXNAMELEN+32];
4544        char base_type;
4545        char set_type;
4546        nvlist_t *base_nvl = NULL;
4547        nvlist_t *set_nvl = NULL;
4548        nvlist_t *nvl;

4550        if (nvlist_alloc(&base_nvl, NV_UNIQUE_NAME, 0) != 0)
4551                nomem();
4552        if (nvlist_alloc(&set_nvl, NV_UNIQUE_NAME, 0) !=  0)
4553                nomem();

4555        switch (type) {
4556        case ZFS_DELEG_NAMED_SET_SETS:
4557        case ZFS_DELEG_NAMED_SET:
4558                set_type = ZFS_DELEG_NAMED_SET_SETS;
4559                base_type = ZFS_DELEG_NAMED_SET;
4560                ld[0] = ZFS_DELEG_NA;
4561                break;
4562        case ZFS_DELEG_CREATE_SETS:
4563        case ZFS_DELEG_CREATE:
4564                set_type = ZFS_DELEG_CREATE_SETS;
4565                base_type = ZFS_DELEG_CREATE;
4566                ld[0] = ZFS_DELEG_NA;
4567                break;
```

```
4568         case ZFS_DELEG_USER_SETS:
4569         case ZFS_DELEG_USER:
4570                 set_type = ZFS_DELEG_USER_SETS;
4571                 base_type = ZFS_DELEG_USER;
4572                 if (local)
4573                         ld[0] = ZFS_DELEG_LOCAL;
4574                 if (descend)
4575                         ld[1] = ZFS_DELEG_DESCENDENT;
4576                 break;
4577         case ZFS_DELEG_GROUP_SETS:
4578         case ZFS_DELEG_GROUP:
4579                 set_type = ZFS_DELEG_GROUP_SETS;
4580                 base_type = ZFS_DELEG_GROUP;
4581                 if (local)
4582                         ld[0] = ZFS_DELEG_LOCAL;
4583                 if (descend)
4584                         ld[1] = ZFS_DELEG_DESCENDENT;
4585                 break;
4586         case ZFS_DELEG_EVERYONE_SETS:
4587         case ZFS_DELEG_EVERYONE:
4588                 set_type = ZFS_DELEG_EVERYONE_SETS;
4589                 base_type = ZFS_DELEG_EVERYONE;
4590                 if (local)
4591                         ld[0] = ZFS_DELEG_LOCAL;
4592                 if (descend)
4593                         ld[1] = ZFS_DELEG_DESCENDENT;
4594         }

4596         if (perms != NULL) {
4597                 char *curr = perms;
4598                 char *end = curr + strlen(perms);

4600                 while (curr < end) {
4601                         char *delim = strchr(curr, ',');
4602                         if (delim == NULL)
4603                                 delim = end;
4604                         else
4605                                 *delim = '\0';

4607                         if (curr[0] == '@')
4608                                 nvl = set_nvl;
4609                         else
4610                                 nvl = base_nvl;

4612                         (void) nvlist_add_boolean(nvl, curr);
4613                         if (delim != end)
4614                                 *delim = ',';
4615                         curr = delim + 1;
4616                 }

4618                 for (i = 0; i < 2; i++) {
4619                         char locality = ld[i];
4620                         if (locality == 0)
4621                                 continue;

4623                         if (!nvlist_empty(base_nvl)) {
4624                                 if (who != NULL)
4625                                         (void) snprintf(who_buf,
4626                                             sizeof (who_buf), "%c%c$%s",
4627                                             base_type, locality, who);
4628                                 else
4629                                         (void) snprintf(who_buf,
4630                                             sizeof (who_buf), "%c%c$",
4631                                             base_type, locality);

4633                                 (void) nvlist_add_nvlist(top_nvl, who_buf,
```



```
4634                                     base_nvl);
4635                         }


4638                         if (!nvlist_empty(set_nvl)) {
4639                                 if (who != NULL)
4640                                         (void) snprintf(who_buf,
4641                                             sizeof (who_buf), "%c%c$%s",
4642                                             set_type, locality, who);
4643                                 else
4644                                         (void) snprintf(who_buf,
4645                                             sizeof (who_buf), "%c%c$",
4646                                             set_type, locality);

4648                                 (void) nvlist_add_nvlist(top_nvl, who_buf,
4649                                     set_nvl);
4650                         }
4651                 }
4652         } else {
4653                 for (i = 0; i < 2; i++) {
4654                         char locality = ld[i];
4655                         if (locality == 0)
4656                                 continue;

4658                         if (who != NULL)
4659                                 (void) snprintf(who_buf, sizeof (who_buf),
4660                                     "%c%c$%s", base_type, locality, who);
4661                         else
4662                                 (void) snprintf(who_buf, sizeof (who_buf),
4663                                     "%c%c$", base_type, locality);
4664                         (void) nvlist_add_boolean(top_nvl, who_buf);

4666                         if (who != NULL)
4667                                 (void) snprintf(who_buf, sizeof (who_buf),
4668                                     "%c%c$%s", set_type, locality, who);
4669                         else
4670                                 (void) snprintf(who_buf, sizeof (who_buf),
4671                                     "%c%c$", set_type, locality);
4672                         (void) nvlist_add_boolean(top_nvl, who_buf);
4673                 }
4674         }
4675 }

4677 static int
4678 construct_fsacl_list(boolean_t un, struct allow_opts *opts, nvlist_t **nvlp)
4679 {
4680         if (nvlist_alloc(nvlp, NV_UNIQUE_NAME, 0) != 0)
4681                 nomem();

4683         if (opts->set) {
4684                 store_allow_perm(ZFS_DELEG_NAMED_SET, opts->local,
4685                     opts->descend, opts->who, opts->perms, *nvlp);
4686         } else if (opts->create) {
4687                 store_allow_perm(ZFS_DELEG_CREATE, opts->local,
4688                     opts->descend, NULL, opts->perms, *nvlp);
4689         } else if (opts->everyone) {
4690                 store_allow_perm(ZFS_DELEG_EVERYONE, opts->local,
4691                     opts->descend, NULL, opts->perms, *nvlp);
4692         } else {
4693                 char *curr = opts->who;
4694                 char *end = curr + strlen(curr);

4696                 while (curr < end) {
4697                         const char *who;
4698                         zfs_deleg_who_type_t who_type;
4699                         char *endch;
```

```
4700                        char *delim = strchr(curr, ',');
4701                        char errbuf[256];
4702                        char id[64];
4703                        struct passwd *p = NULL;
4704                        struct group *g = NULL;

4706                        uid_t rid;
4707                        if (delim == NULL)
4708                                delim = end;
4709                        else
4710                                *delim = '\0';

4712                        rid = (uid_t)strtol(curr, &endch, 0);
4713                        if (opts->user) {
4714                                who_type = ZFS_DELEG_USER;
4715                                if (*endch != '\0')
4716                                        p = getpwnam(curr);
4717                                else
4718                                        p = getpwuid(rid);

4720                                if (p != NULL)
4721                                        rid = p->pw_uid;
4722                                else {
4723                                        (void) snprintf(errbuf, 256, gettext(
4724                                            "invalid user %s"), curr);
4725                                        allow_usage(un, B_TRUE, errbuf);
4726                                }
4727                        } else if (opts->group) {
4728                                who_type = ZFS_DELEG_GROUP;
4729                                if (*endch != '\0')
4730                                        g = getgrnam(curr);
4731                                else
4732                                        g = getgrgid(rid);

4734                                if (g != NULL)
4735                                        rid = g->gr_gid;
4736                                else {
4737                                        (void) snprintf(errbuf, 256, gettext(
4738                                            "invalid group %s"),  curr);
4739                                        allow_usage(un, B_TRUE, errbuf);
4740                                }
4741                        } else {
4742                                if (*endch != '\0') {
4743                                        p = getpwnam(curr);
4744                                } else {
4745                                        p = getpwuid(rid);
4746                                }

4748                                if (p == NULL)
4749                                        if (*endch != '\0') {
4750                                                g = getgrnam(curr);
4751                                        } else {
4752                                                g = getgrgid(rid);
4753                                        }

4755                                if (p != NULL) {
4756                                        who_type = ZFS_DELEG_USER;
4757                                        rid = p->pw_uid;
4758                                } else if (g != NULL) {
4759                                        who_type = ZFS_DELEG_GROUP;
4760                                        rid = g->gr_gid;
4761                                } else {
4762                                        (void) snprintf(errbuf, 256, gettext(
4763                                            "invalid user/group %s"), curr);
4764                                        allow_usage(un, B_TRUE, errbuf);
4765                                }
```



```
4766                        }

4768                        (void) sprintf(id, "%u", rid);
4769                        who = id;

4771                        store_allow_perm(who_type, opts->local,
4772                            opts->descend, who, opts->perms, *nvlp);
4773                        curr = delim + 1;
4774                }
4775        }

4777        return (0);
4778 }

4780 static void
4781 print_set_creat_perms(uu_avl_t *who_avl)
4782 {
4783        const char *sc_title[] = {
4784                gettext("Permission sets:\n"),
4785                gettext("Create time permissions:\n"),
4786                NULL
4787        };
4788        const char **title_ptr = sc_title;
4789        who_perm_node_t *who_node = NULL;
4790        int prev_weight = -1;

4792        for (who_node = uu_avl_first(who_avl); who_node != NULL;
4793            who_node = uu_avl_next(who_avl, who_node)) {
4794                uu_avl_t *avl = who_node->who_perm.who_deleg_perm_avl;
4795                zfs_deleg_who_type_t who_type = who_node->who_perm.who_type;
4796                const char *who_name = who_node->who_perm.who_name;
4797                int weight = who_type2weight(who_type);
4798                boolean_t first = B_TRUE;
4799                deleg_perm_node_t *deleg_node;

4801                if (prev_weight != weight) {
4802                        (void) printf(*title_ptr++);
4803                        prev_weight = weight;
4804                }

4806                if (who_name == NULL || strnlen(who_name, 1) == 0)
4807                        (void) printf("\t");
4808                else
4809                        (void) printf("\t%s ", who_name);

4811                for (deleg_node = uu_avl_first(avl); deleg_node != NULL;
4812                    deleg_node = uu_avl_next(avl, deleg_node)) {
4813                        if (first) {
4814                                (void) printf("%s",
4815                                    deleg_node->dpn_perm.dp_name);
4816                                first = B_FALSE;
4817                        } else
4818                                (void) printf(",%s",
4819                                    deleg_node->dpn_perm.dp_name);
4820                }

4822                (void) printf("\n");
4823        }
4824 }

4826 static void inline
4827 print_uge_deleg_perms(uu_avl_t *who_avl, boolean_t local, boolean_t descend,
4828     const char *title)
4829 {
4830        who_perm_node_t *who_node = NULL;
4831        boolean_t prt_title = B_TRUE;
```

```
4832         uu_avl_walk_t *walk;

4834         if ((walk = uu_avl_walk_start(who_avl, UU_WALK_ROBUST)) == NULL)
4835                 nomem();

4837         while ((who_node = uu_avl_walk_next(walk)) != NULL) {
4838                 const char *who_name = who_node->who_perm.who_name;
4839                 const char *nice_who_name = who_node->who_perm.who_ug_name;
4840                 uu_avl_t *avl = who_node->who_perm.who_deleg_perm_avl;
4841                 zfs_deleg_who_type_t who_type = who_node->who_perm.who_type;
4842                 char delim = ' ';
4843                 deleg_perm_node_t *deleg_node;
4844                 boolean_t prt_who = B_TRUE;

4846                 for (deleg_node = uu_avl_first(avl);
4847                     deleg_node != NULL;
4848                     deleg_node = uu_avl_next(avl, deleg_node)) {
4849                         if (local != deleg_node->dpn_perm.dp_local ||
4850                             descend != deleg_node->dpn_perm.dp_descend)
4851                                 continue;

4853                         if (prt_who) {
4854                                 const char *who = NULL;
4855                                 if (prt_title) {
4856                                         prt_title = B_FALSE;
4857                                         (void) printf(title);
4858                                 }

4860                                 switch (who_type) {
4861                                 case ZFS_DELEG_USER_SETS:
4862                                 case ZFS_DELEG_USER:
4863                                         who = gettext("user");
4864                                         if (nice_who_name)
4865                                                 who_name  = nice_who_name;
4866                                         break;
4867                                 case ZFS_DELEG_GROUP_SETS:
4868                                 case ZFS_DELEG_GROUP:
4869                                         who = gettext("group");
4870                                         if (nice_who_name)
4871                                                 who_name  = nice_who_name;
4872                                         break;
4873                                 case ZFS_DELEG_EVERYONE_SETS:
4874                                 case ZFS_DELEG_EVERYONE:
4875                                         who = gettext("everyone");
4876                                         who_name = NULL;
4877                                 }

4879                                 prt_who = B_FALSE;
4880                                 if (who_name == NULL)
4881                                         (void) printf("\t%s", who);
4882                                 else
4883                                         (void) printf("\t%s %s", who, who_name);
4884                         }

4886                         (void) printf("%c%s", delim,
4887                             deleg_node->dpn_perm.dp_name);
4888                         delim = ',';
4889                 }

4891                 if (!prt_who)
4892                         (void) printf("\n");
4893         }

4895         uu_avl_walk_end(walk);
4896 }
```



```
4898 static void
4899 print_fs_perms(fs_perm_set_t *fspset)
4900 {
4901         fs_perm_node_t *node = NULL;
4902         char buf[ZFS_MAXNAMELEN+32];
4903         const char *dsname = buf;

4905         for (node = uu_list_first(fspset->fsps_list); node != NULL;
4906             node = uu_list_next(fspset->fsps_list, node)) {
4907                 uu_avl_t *sc_avl = node->fspn_fsperm.fsp_sc_avl;
4908                 uu_avl_t *uge_avl = node->fspn_fsperm.fsp_uge_avl;
4909                 int left = 0;

4911                 (void) snprintf(buf, ZFS_MAXNAMELEN+32,
4912                     gettext("---- Permissions on %s "),
4913                     node->fspn_fsperm.fsp_name);
4914                 (void) printf(dsname);
4915                 left = 70 - strlen(buf);
4916                 while (left-- > 0)
4917                         (void) printf("-");
4918                 (void) printf("\n");

4920                 print_set_creat_perms(sc_avl);
4921                 print_uge_deleg_perms(uge_avl, B_TRUE, B_FALSE,
4922                     gettext("Local permissions:\n"));
4923                 print_uge_deleg_perms(uge_avl, B_FALSE, B_TRUE,
4924                     gettext("Descendent permissions:\n"));
4925                 print_uge_deleg_perms(uge_avl, B_TRUE, B_TRUE,
4926                     gettext("Local+Descendent permissions:\n"));
4927         }
4928 }

4930 static fs_perm_set_t fs_perm_set = { NULL, NULL, NULL, NULL };

4932 struct deleg_perms {
4933         boolean_t un;
4934         nvlist_t *nvl;
4935 };

4937 static int
4938 set_deleg_perms(zfs_handle_t *zhp, void *data)
4939 {
4940         struct deleg_perms *perms = (struct deleg_perms *)data;
4941         zfs_type_t zfs_type = zfs_get_type(zhp);

4943         if (zfs_type != ZFS_TYPE_FILESYSTEM && zfs_type != ZFS_TYPE_VOLUME)
4944                 return (0);

4946         return (zfs_set_fsacl(zhp, perms->un, perms->nvl));
4947 }

4949 static int
4950 zfs_do_allow_unallow_impl(int argc, char **argv, boolean_t un)
4951 {
4952         zfs_handle_t *zhp;
4953         nvlist_t *perm_nvl = NULL;
4954         nvlist_t *update_perm_nvl = NULL;
4955         int error = 1;
4956         int c;
4957         struct allow_opts opts = { 0 };

4959         const char *optstr = un ? "ldugecsrh" : "ldugecsh";

4961         /* check opts */
4962         while ((c = getopt(argc, argv, optstr)) != -1) {
4963                 switch (c) {
```

```
4964                 case 'l':
4965                         opts.local = B_TRUE;
4966                         break;
4967                 case 'd':
4968                         opts.descend = B_TRUE;
4969                         break;
4970                 case 'u':
4971                         opts.user = B_TRUE;
4972                         break;
4973                 case 'g':
4974                         opts.group = B_TRUE;
4975                         break;
4976                 case 'e':
4977                         opts.everyone = B_TRUE;
4978                         break;
4979                 case 's':
4980                         opts.set = B_TRUE;
4981                         break;
4982                 case 'c':
4983                         opts.create = B_TRUE;
4984                         break;
4985                 case 'r':
4986                         opts.recursive = B_TRUE;
4987                         break;
4988                 case ':':
4989                         (void) fprintf(stderr, gettext("missing argument for "
4990                             "'%c' option\n"), optopt);
4991                         usage(B_FALSE);
4992                         break;
4993                 case 'h':
4994                         opts.prt_usage = B_TRUE;
4995                         break;
4996                 case '?':
4997                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
4998                             optopt);
4999                         usage(B_FALSE);
5000                 }
5001         }

5003         argc -= optind;
5004         argv += optind;

5006         /* check arguments */
5007         parse_allow_args(argc, argv, un, &opts);

5009         /* try to open the dataset */
5010         if ((zhp = zfs_open(g_zfs, opts.dataset, ZFS_TYPE_FILESYSTEM |
5011             ZFS_TYPE_VOLUME)) == NULL) {
5012                 (void) fprintf(stderr, "Failed to open dataset: %s\n",
5013                     opts.dataset);
5014                 return (-1);
5015         }

5017         if (zfs_get_fsacl(zhp, &perm_nvl) != 0)
5018                 goto cleanup2;

5020         fs_perm_set_init(&fs_perm_set);
5021         if (parse_fs_perm_set(&fs_perm_set, perm_nvl) != 0) {
5022                 (void) fprintf(stderr, "Failed to parse fsacl permissions\n");
5023                 goto cleanup1;
5024         }

5026         if (opts.prt_perms)
5027                 print_fs_perms(&fs_perm_set);
5028         else {
5029                 (void) construct_fsacl_list(un, &opts, &update_perm_nvl);
```



```
5030                 if (zfs_set_fsacl(zhp, un, update_perm_nvl) != 0)
5031                         goto cleanup0;

5033                 if (un && opts.recursive) {
5034                         struct deleg_perms data = { un, update_perm_nvl };
5035                         if (zfs_iter_filesystems(zhp, set_deleg_perms,
5036                             &data) != 0)
5037                                 goto cleanup0;
5038                 }
5039         }

5041         error = 0;

5043 cleanup0:
5044         nvlist_free(perm_nvl);
5045         if (update_perm_nvl != NULL)
5046                 nvlist_free(update_perm_nvl);
5047 cleanup1:
5048         fs_perm_set_fini(&fs_perm_set);
5049 cleanup2:
5050         zfs_close(zhp);

5052         return (error);
5053 }

5055 /*
5056  * zfs allow [-r] [-t] <tag> <snap> ...
5057  *
5058  *      -r      Recursively hold
5059  *      -t      Temporary hold (hidden option)
5060  *
5061  * Apply a user-hold with the given tag to the list of snapshots.
5062  */
5063 static int
5064 zfs_do_allow(int argc, char **argv)
5065 {
5066         return (zfs_do_allow_unallow_impl(argc, argv, B_FALSE));
5067 }

5069 /*
5070  * zfs unallow [-r] [-t] <tag> <snap> ...
5071  *
5072  *      -r      Recursively hold
5073  *      -t      Temporary hold (hidden option)
5074  *
5075  * Apply a user-hold with the given tag to the list of snapshots.
5076  */
5077 static int
5078 zfs_do_unallow(int argc, char **argv)
5079 {
5080         return (zfs_do_allow_unallow_impl(argc, argv, B_TRUE));
5081 }

5083 static int
5084 zfs_do_hold_rele_impl(int argc, char **argv, boolean_t holding)
5085 {
5086         int errors = 0;
5087         int i;
5088         const char *tag;
5089         boolean_t recursive = B_FALSE;
5090         boolean_t temphold = B_FALSE;
5091         const char *opts = holding ? "rt" : "r";
5092         int c;

5094         /* check options */
5095         while ((c = getopt(argc, argv, opts)) != -1) {
```

```
5096                switch (c) {
5097                case 'r':
5098                        recursive = B_TRUE;
5099                        break;
5100                case 't':
5101                        temphold = B_TRUE;
5102                        break;
5103                case '?':
5104                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5105                            optopt);
5106                        usage(B_FALSE);
5107                }
5108        }

5110        argc -= optind;
5111        argv += optind;

5113        /* check number of arguments */
5114        if (argc < 2)
5115                usage(B_FALSE);

5117        tag = argv[0];
5118        --argc;
5119        ++argv;

5121        if (holding && tag[0] == '.') {
5122                /* tags starting with '.' are reserved for libzfs */
5123                (void) fprintf(stderr, gettext("tag may not start with '.'\n"));
5124                usage(B_FALSE);
5125        }

5127        for (i = 0; i < argc; ++i) {
5128                zfs_handle_t *zhp;
5129                char parent[ZFS_MAXNAMELEN];
5130                const char *delim;
5131                char *path = argv[i];

5133                delim = strchr(path, '@');
5134                if (delim == NULL) {
5135                        (void) fprintf(stderr,
5136                            gettext("'%s' is not a snapshot\n"), path);
5137                        ++errors;
5138                        continue;
5139                }
5140                (void) strncpy(parent, path, delim - path);
5141                parent[delim - path] = '\0';

5143                zhp = zfs_open(g_zfs, parent,
5144                    ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
5145                if (zhp == NULL) {
5146                        ++errors;
5147                        continue;
5148                }
5149                if (holding) {
5150                        if (zfs_hold(zhp, delim+1, tag, recursive,
5151                            temphold, B_FALSE, -1, 0, 0) != 0)
5152                                ++errors;
5153                } else {
5154                        if (zfs_release(zhp, delim+1, tag, recursive) != 0)
5155                                ++errors;
5156                }
5157                zfs_close(zhp);
5158        }

5160        return (errors != 0);
5161 }
```



```
5163 /*
5164  * zfs hold [-r] [-t] <tag> <snap> ...
5165  *
5166  *      -r      Recursively hold
5167  *      -t      Temporary hold (hidden option)
5168  *
5169  * Apply a user-hold with the given tag to the list of snapshots.
5170  */
5171 static int
5172 zfs_do_hold(int argc, char **argv)
5173 {
5174         return (zfs_do_hold_rele_impl(argc, argv, B_TRUE));
5175 }

5177 /*
5178  * zfs release [-r] <tag> <snap> ...
5179  *
5180  *      -r      Recursively release
5181  *
5182  * Release a user-hold with the given tag from the list of snapshots.
5183  */
5184 static int
5185 zfs_do_release(int argc, char **argv)
5186 {
5187         return (zfs_do_hold_rele_impl(argc, argv, B_FALSE));
5188 }

5190 typedef struct holds_cbdata {
5191         boolean_t       cb_recursive;
5192         const char      *cb_snapname;
5193         nvlist_t        **cb_nvlp;
5194         size_t          cb_max_namelen;
5195         size_t          cb_max_taglen;
5196 } holds_cbdata_t;

5198 #define STRFTIME_FMT_STR "%a %b %e %k:%M %Y"
5199 #define DATETIME_BUF_LEN (32)
5200 /*
5201  *
5202  */
5203 static void
5204 print_holds(boolean_t scripted, size_t nwidth, size_t tagwidth, nvlist_t *nvl)
5205 {
5206         int i;
5207         nvpair_t *nvp = NULL;
5208         char *hdr_cols[] = { "NAME", "TAG", "TIMESTAMP" };
5209         const char *col;

5211         if (!scripted) {
5212                 for (i = 0; i < 3; i++) {
5213                         col = gettext(hdr_cols[i]);
5214                         if (i < 2)
5215                                 (void) printf("%-*s  ", i ? tagwidth : nwidth,
5216                                     col);
5217                         else
5218                                 (void) printf("%s\n", col);
5219                 }
5220         }

5222         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
5223                 char *zname = nvpair_name(nvp);
5224                 nvlist_t *nvl2;
5225                 nvpair_t *nvp2 = NULL;
5226                 (void) nvpair_value_nvlist(nvp, &nvl2);
5227                 while ((nvp2 = nvlist_next_nvpair(nvl2, nvp2)) != NULL) {
```

```
5228                        char tsbuf[DATETIME_BUF_LEN];
5229                        char *tagname = nvpair_name(nvp2);
5230                        uint64_t val = 0;
5231                        time_t time;
5232                        struct tm t;
5233                        char sep = scripted ? '\t' : ' ';
5234                        size_t sepnum = scripted ? 1 : 2;

5236                        (void) nvpair_value_uint64(nvp2, &val);
5237                        time = (time_t)val;
5238                        (void) localtime_r(&time, &t);
5239                        (void) strftime(tsbuf, DATETIME_BUF_LEN,
5240                            gettext(STRFTIME_FMT_STR), &t);

5242                        (void) printf("%-*s%*c%-*s%*c%s\n", nwidth, zname,
5243                            sepnum, sep, tagwidth, tagname, sepnum, sep, tsbuf);
5244                }
5245        }
5246 }

5248 /*
5249  * Generic callback function to list a dataset or snapshot.
5250  */
5251 static int
5252 holds_callback(zfs_handle_t *zhp, void *data)
5253 {
5254         holds_cbdata_t *cbp = data;
5255         nvlist_t *top_nvl = *cbp->cb_nvlp;
5256         nvlist_t *nvl = NULL;
5257         nvpair_t *nvp = NULL;
5258         const char *zname = zfs_get_name(zhp);
5259         size_t znamelen = strnlen(zname, ZFS_MAXNAMELEN);

5261         if (cbp->cb_recursive) {
5262                 const char *snapname;
5263                 char *delim  = strchr(zname, '@');
5264                 if (delim == NULL)
5265                         return (0);

5267                 snapname = delim + 1;
5268                 if (strcmp(cbp->cb_snapname, snapname))
5269                         return (0);
5270         }

5272         if (zfs_get_holds(zhp, &nvl) != 0)
5273                 return (-1);

5275         if (znamelen > cbp->cb_max_namelen)
5276                 cbp->cb_max_namelen  = znamelen;

5278         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
5279                 const char *tag = nvpair_name(nvp);
5280                 size_t taglen = strnlen(tag, MAXNAMELEN);
5281                 if (taglen > cbp->cb_max_taglen)
5282                         cbp->cb_max_taglen  = taglen;
5283         }

5285         return (nvlist_add_nvlist(top_nvl, zname, nvl));
5286 }

5288 /*
5289  * zfs holds [-r] <snap> ...
5290  *
5291  *      -r      Recursively hold
5292  */
5293 static int
```



```
5294 zfs_do_holds(int argc, char **argv)
5295 {
5296         int errors = 0;
5297         int c;
5298         int i;
5299         boolean_t scripted = B_FALSE;
5300         boolean_t recursive = B_FALSE;
5301         const char *opts = "rH";
5302         nvlist_t *nvl;

5304         int types = ZFS_TYPE_SNAPSHOT;
5305         holds_cbdata_t cb = { 0 };

5307         int limit = 0;
5308         int ret = 0;
5309         int flags = 0;

5311         /* check options */
5312         while ((c = getopt(argc, argv, opts)) != -1) {
5313                 switch (c) {
5314                 case 'r':
5315                         recursive = B_TRUE;
5316                         break;
5317                 case 'H':
5318                         scripted = B_TRUE;
5319                         break;
5320                 case '?':
5321                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5322                             optopt);
5323                         usage(B_FALSE);
5324                 }
5325         }

5327         if (recursive) {
5328                 types |= ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME;
5329                 flags |= ZFS_ITER_RECURSE;
5330         }

5332         argc -= optind;
5333         argv += optind;

5335         /* check number of arguments */
5336         if (argc < 1)
5337                 usage(B_FALSE);

5339         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
5340                 nomem();

5342         for (i = 0; i < argc; ++i) {
5343                 char *snapshot = argv[i];
5344                 const char *delim;
5345                 const char *snapname;

5347                 delim = strchr(snapshot, '@');
5348                 if (delim == NULL) {
5349                         (void) fprintf(stderr,
5350                             gettext("'%s' is not a snapshot\n"), snapshot);
5351                         ++errors;
5352                         continue;
5353                 }
5354                 snapname = delim + 1;
5355                 if (recursive)
5356                         snapshot[delim - snapshot] = '\0';

5358                 cb.cb_recursive = recursive;
5359                 cb.cb_snapname = snapname;
```

```
5360                cb.cb_nvlp = &nvl;

5362                /*
5363                 *  1. collect holds data, set format options
5364                 */
5365                ret = zfs_for_each(argc, argv, flags, types, NULL, NULL, limit,
5366                    holds_callback, &cb);
5367                if (ret != 0)
5368                        ++errors;
5369        }

5371        /*
5372         *  2. print holds data
5373         */
5374        print_holds(scripted, cb.cb_max_namelen, cb.cb_max_taglen, nvl);

5376        if (nvlist_empty(nvl))
5377                (void) printf(gettext("no datasets available\n"));

5379        nvlist_free(nvl);

5381        return (0 != errors);
5382 }

5384 #define CHECK_SPINNER 30
5385 #define SPINNER_TIME 3          /* seconds */
5386 #define MOUNT_TIME 5            /* seconds */

5388 static int
5389 get_one_dataset(zfs_handle_t *zhp, void *data)
5390 {
5391        static char *spin[] = { "-", "\\", "|", "/" };
5392        static int spinval = 0;
5393        static int spincheck = 0;
5394        static time_t last_spin_time = (time_t)0;
5395        get_all_cb_t *cbp = data;
5396        zfs_type_t type = zfs_get_type(zhp);

5398        if (cbp->cb_verbose) {
5399                if (--spincheck < 0) {
5400                        time_t now = time(NULL);
5401                        if (last_spin_time + SPINNER_TIME < now) {
5402                                update_progress(spin[spinval++ % 4]);
5403                                last_spin_time = now;
5404                        }
5405                        spincheck = CHECK_SPINNER;
5406                }
5407        }

5409        /*
5410         * Interate over any nested datasets.
5411         */
5412        if (zfs_iter_filesystems(zhp, get_one_dataset, data) != 0) {
5413                zfs_close(zhp);
5414                return (1);
5415        }

5417        /*
5418         * Skip any datasets whose type does not match.
5419         */
5420        if ((type & ZFS_TYPE_FILESYSTEM) == 0) {
5421                zfs_close(zhp);
5422                return (0);
5423        }
5424        libzfs_add_handle(cbp, zhp);
5425        assert(cbp->cb_used <= cbp->cb_alloc);
```



```
5427        return (0);
5428 }

5430 static void
5431 get_all_datasets(zfs_handle_t ***dslist, size_t *count, boolean_t verbose)
5432 {
5433        get_all_cb_t cb = { 0 };
5434        cb.cb_verbose = verbose;
5435        cb.cb_getone = get_one_dataset;

5437        if (verbose)
5438                set_progress_header(gettext("Reading ZFS config"));
5439        (void) zfs_iter_root(g_zfs, get_one_dataset, &cb);

5441        *dslist = cb.cb_handles;
5442        *count = cb.cb_used;

5444        if (verbose)
5445                finish_progress(gettext("done."));
5446 }

5448 /*
5449  * Generic callback for sharing or mounting filesystems.  Because the code is so
5450  * similar, we have a common function with an extra parameter to determine which
5451  * mode we are using.
5452  */
5453 #define OP_SHARE        0x1
5454 #define OP_MOUNT        0x2

5456 /*
5457  * Share or mount a dataset.
5458  */
5459 static int
5460 share_mount_one(zfs_handle_t *zhp, int op, int flags, char *protocol,
5461     boolean_t explicit, const char *options)
5462 {
5463        char mountpoint[ZFS_MAXPROPLEN];
5464        char shareopts[ZFS_MAXPROPLEN];
5465        char smbshareopts[ZFS_MAXPROPLEN];
5466        const char *cmdname = op == OP_SHARE ? "share" : "mount";
5467        struct mnttab mnt;
5468        uint64_t zoned, canmount;
5469        boolean_t shared_nfs, shared_smb;

5471        assert(zfs_get_type(zhp) & ZFS_TYPE_FILESYSTEM);

5473        /*
5474         * Check to make sure we can mount/share this dataset.  If we
5475         * are in the global zone and the filesystem is exported to a
5476         * local zone, or if we are in a local zone and the
5477         * filesystem is not exported, then it is an error.
5478         */
5479        zoned = zfs_prop_get_int(zhp, ZFS_PROP_ZONED);

5481        if (zoned && getzoneid() == GLOBAL_ZONEID) {
5482                if (!explicit)
5483                        return (0);

5485                (void) fprintf(stderr, gettext("cannot %s '%s': "
5486                    "dataset is exported to a local zone\n"), cmdname,
5487                    zfs_get_name(zhp));
5488                return (1);

5490        } else if (!zoned && getzoneid() != GLOBAL_ZONEID) {
5491                if (!explicit)
```

```
5492                        return (0);
5493
5494                (void) fprintf(stderr, gettext("cannot %s '%s': "
5495                    "permission denied\n"), cmdname,
5496                    zfs_get_name(zhp));
5497                return (1);
5498        }
5499
5500        /*
5501         * Ignore any filesystems which don't apply to us. This
5502         * includes those with a legacy mountpoint, or those with
5503         * legacy share options.
5504         */
5505        verify(zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
5506            sizeof (mountpoint), NULL, NULL, 0, B_FALSE) == 0);
5507        verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS, shareopts,
5508            sizeof (shareopts), NULL, NULL, 0, B_FALSE) == 0);
5509        verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB, smbshareopts,
5510            sizeof (smbshareopts), NULL, NULL, 0, B_FALSE) == 0);
5511
5512        if (op == OP_SHARE && strcmp(shareopts, "off") == 0 &&
5513            strcmp(smbshareopts, "off") == 0) {
5514                if (!explicit)
5515                        return (0);
5516
5517                (void) fprintf(stderr, gettext("cannot share '%s': "
5518                    "legacy share\n"), zfs_get_name(zhp));
5519                (void) fprintf(stderr, gettext("use share(1M) to "
5520                    "share this filesystem, or set "
5521                    "sharenfs property on\n"));
5522                return (1);
5523        }
5524
5525        /*
5526         * We cannot share or mount legacy filesystems. If the
5527         * shareopts is non-legacy but the mountpoint is legacy, we
5528         * treat it as a legacy share.
5529         */
5530        if (strcmp(mountpoint, "legacy") == 0) {
5531                if (!explicit)
5532                        return (0);
5533
5534                (void) fprintf(stderr, gettext("cannot %s '%s': "
5535                    "legacy mountpoint\n"), cmdname, zfs_get_name(zhp));
5536                (void) fprintf(stderr, gettext("use %s(1M) to "
5537                    "%s this filesystem\n"), cmdname, cmdname);
5538                return (1);
5539        }
5540
5541        if (strcmp(mountpoint, "none") == 0) {
5542                if (!explicit)
5543                        return (0);
5544
5545                (void) fprintf(stderr, gettext("cannot %s '%s': no "
5546                    "mountpoint set\n"), cmdname, zfs_get_name(zhp));
5547                return (1);
5548        }
5549
5550        /*
5551         * canmount     explicit        outcome
5552         * on           no              pass through
5553         * on           yes             pass through
5554         * off          no              return 0
5555         * off          yes             display error, return 1
5556         * noauto       no              return 0
5557         * noauto       yes             pass through
```



```
5558         */
5559        canmount = zfs_prop_get_int(zhp, ZFS_PROP_CANMOUNT);
5560        if (canmount == ZFS_CANMOUNT_OFF) {
5561                if (!explicit)
5562                        return (0);
5563
5564                (void) fprintf(stderr, gettext("cannot %s '%s': "
5565                    "'canmount' property is set to 'off'\n"), cmdname,
5566                    zfs_get_name(zhp));
5567                return (1);
5568        } else if (canmount == ZFS_CANMOUNT_NOAUTO && !explicit) {
5569                return (0);
5570        }
5571
5572        /*
5573         * At this point, we have verified that the mountpoint and/or
5574         * shareopts are appropriate for auto management. If the
5575         * filesystem is already mounted or shared, return (failing
5576         * for explicit requests); otherwise mount or share the
5577         * filesystem.
5578         */
5579        switch (op) {
5580        case OP_SHARE:
5581
5582                shared_nfs = zfs_is_shared_nfs(zhp, NULL);
5583                shared_smb = zfs_is_shared_smb(zhp, NULL);
5584
5585                if (shared_nfs && shared_smb ||
5586                    (shared_nfs && strcmp(shareopts, "on") == 0 &&
5587                    strcmp(smbshareopts, "off") == 0) ||
5588                    (shared_smb && strcmp(smbshareopts, "on") == 0 &&
5589                    strcmp(shareopts, "off") == 0)) {
5590                        if (!explicit)
5591                                return (0);
5592
5593                        (void) fprintf(stderr, gettext("cannot share "
5594                            "'%s': filesystem already shared\n"),
5595                            zfs_get_name(zhp));
5596                        return (1);
5597                }
5598
5599                if (!zfs_is_mounted(zhp, NULL) &&
5600                    zfs_mount(zhp, NULL, 0) != 0)
5601                        return (1);
5602
5603                if (protocol == NULL) {
5604                        if (zfs_shareall(zhp) != 0)
5605                                return (1);
5606                } else if (strcmp(protocol, "nfs") == 0) {
5607                        if (zfs_share_nfs(zhp))
5608                                return (1);
5609                } else if (strcmp(protocol, "smb") == 0) {
5610                        if (zfs_share_smb(zhp))
5611                                return (1);
5612                } else {
5613                        (void) fprintf(stderr, gettext("cannot share "
5614                            "'%s': invalid share type '%s' "
5615                            "specified\n"),
5616                            zfs_get_name(zhp), protocol);
5617                        return (1);
5618                }
5619
5620                break;
5621
5622        case OP_MOUNT:
5623                if (options == NULL)
```

```
5624                        mnt.mnt_mntopts = "";
5625                else
5626                        mnt.mnt_mntopts = (char *)options;

5628                if (!hasmntopt(&mnt, MNTOPT_REMOUNT) &&
5629                    zfs_is_mounted(zhp, NULL)) {
5630                        if (!explicit)
5631                                return (0);

5633                        (void) fprintf(stderr, gettext("cannot mount "
5634                            "'%s': filesystem already mounted\n"),
5635                            zfs_get_name(zhp));
5636                        return (1);
5637                }

5639                if (zfs_mount(zhp, options, flags) != 0)
5640                        return (1);
5641                break;
5642        }

5644        return (0);
5645 }

5647 /*
5648  * Reports progress in the form "(current/total)".  Not thread-safe.
5649  */
5650 static void
5651 report_mount_progress(int current, int total)
5652 {
5653        static time_t last_progress_time = 0;
5654        time_t now = time(NULL);
5655        char info[32];

5657        /* report 1..n instead of 0..n-1 */
5658        ++current;

5660        /* display header if we're here for the first time */
5661        if (current == 1) {
5662                set_progress_header(gettext("Mounting ZFS filesystems"));
5663        } else if (current != total && last_progress_time + MOUNT_TIME >= now) {
5664                /* too soon to report again */
5665                return;
5666        }

5668        last_progress_time = now;

5670        (void) sprintf(info, "(%d/%d)", current, total);

5672        if (current == total)
5673                finish_progress(info);
5674        else
5675                update_progress(info);
5676 }

5678 static void
5679 append_options(char *mntopts, char *newopts)
5680 {
5681        int len = strlen(mntopts);

5683        /* original length plus new string to append plus 1 for the comma */
5684        if (len + 1 + strlen(newopts) >= MNT_LINE_MAX) {
5685                (void) fprintf(stderr, gettext("the opts argument for "
5686                    "'%c' option is too long (more than %d chars)\n"),
5687                    "-o", MNT_LINE_MAX);
5688                usage(B_FALSE);
5689        }
```



```
5691        if (*mntopts)
5692                mntopts[len++] = ',';

5694        (void) strcpy(&mntopts[len], newopts);
5695 }

5697 static int
5698 share_mount(int op, int argc, char **argv)
5699 {
5700        int do_all = 0;
5701        boolean_t verbose = B_FALSE;
5702        int c, ret = 0;
5703        char *options = NULL;
5704        int flags = 0;

5706        /* check options */
5707        while ((c = getopt(argc, argv, op == OP_MOUNT ? ":avo:O" : "a"))
5708            != -1) {
5709                switch (c) {
5710                case 'a':
5711                        do_all = 1;
5712                        break;
5713                case 'v':
5714                        verbose = B_TRUE;
5715                        break;
5716                case 'o':
5717                        if (*optarg == '\0') {
5718                                (void) fprintf(stderr, gettext("empty mount "
5719                                    "options (-o) specified\n"));
5720                                usage(B_FALSE);
5721                        }

5723                        if (options == NULL)
5724                                options = safe_malloc(MNT_LINE_MAX + 1);

5726                        /* option validation is done later */
5727                        append_options(options, optarg);
5728                        break;

5730                case 'O':
5731                        flags |= MS_OVERLAY;
5732                        break;
5733                case ':':
5734                        (void) fprintf(stderr, gettext("missing argument for "
5735                            "'%c' option\n"), optopt);
5736                        usage(B_FALSE);
5737                        break;
5738                case '?':
5739                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5740                            optopt);
5741                        usage(B_FALSE);
5742                }
5743        }

5745        argc -= optind;
5746        argv += optind;

5748        /* check number of arguments */
5749        if (do_all) {
5750                zfs_handle_t **dslist = NULL;
5751                size_t i, count = 0;
5752                char *protocol = NULL;

5754                if (op == OP_SHARE && argc > 0) {
5755                        if (strcmp(argv[0], "nfs") != 0 &&
```

```
5756                            strcmp(argv[0], "smb") != 0) {
5757                                (void) fprintf(stderr, gettext("share type "
5758                                    "must be 'nfs' or 'smb'\n"));
5759                                usage(B_FALSE);
5760                        }
5761                        protocol = argv[0];
5762                        argc--;
5763                        argv++;
5764                }

5766                if (argc != 0) {
5767                        (void) fprintf(stderr, gettext("too many arguments\n"));
5768                        usage(B_FALSE);
5769                }

5771                start_progress_timer();
5772                get_all_datasets(&dslist, &count, verbose);

5774                if (count == 0)
5775                        return (0);

5777                qsort(dslist, count, sizeof (void *), libzfs_dataset_cmp);

5779                for (i = 0; i < count; i++) {
5780                        if (verbose)
5781                                report_mount_progress(i, count);

5783                        if (share_mount_one(dslist[i], op, flags, protocol,
5784                            B_FALSE, options) != 0)
5785                                ret = 1;
5786                        zfs_close(dslist[i]);
5787                }

5789                free(dslist);
5790        } else if (argc == 0) {
5791                struct mnttab entry;

5793                if ((op == OP_SHARE) || (options != NULL)) {
5794                        (void) fprintf(stderr, gettext("missing filesystem "
5795                            "argument (specify -a for all)\n"));
5796                        usage(B_FALSE);
5797                }

5799                /*
5800                 * When mount is given no arguments, go through /etc/mnttab and
5801                 * display any active ZFS mounts.  We hide any snapshots, since
5802                 * they are controlled automatically.
5803                 */
5804                rewind(mnttab_file);
5805                while (getmntent(mnttab_file, &entry) == 0) {
5806                        if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0 ||
5807                            strchr(entry.mnt_special, '@') != NULL)
5808                                continue;

5810                        (void) printf("%-30s  %s\n", entry.mnt_special,
5811                            entry.mnt_mountp);
5812                }

5814        } else {
5815                zfs_handle_t *zhp;

5817                if (argc > 1) {
5818                        (void) fprintf(stderr,
5819                            gettext("too many arguments\n"));
5820                        usage(B_FALSE);
5821                }
```



```
5823                if ((zhp = zfs_open(g_zfs, argv[0],
5824                    ZFS_TYPE_FILESYSTEM)) == NULL) {
5825                        ret = 1;
5826                } else {
5827                        ret = share_mount_one(zhp, op, flags, NULL, B_TRUE,
5828                            options);
5829                        zfs_close(zhp);
5830                }
5831        }

5833        return (ret);
5834 }

5836 /*
5837  * zfs mount -a [nfs]
5838  * zfs mount filesystem
5839  *
5840  * Mount all filesystems, or mount the given filesystem.
5841  */
5842 static int
5843 zfs_do_mount(int argc, char **argv)
5844 {
5845        return (share_mount(OP_MOUNT, argc, argv));
5846 }

5848 /*
5849  * zfs share -a [nfs | smb]
5850  * zfs share filesystem
5851  *
5852  * Share all filesystems, or share the given filesystem.
5853  */
5854 static int
5855 zfs_do_share(int argc, char **argv)
5856 {
5857        return (share_mount(OP_SHARE, argc, argv));
5858 }

5860 typedef struct unshare_unmount_node {
5861        zfs_handle_t    *un_zhp;
5862        char            *un_mountp;
5863        uu_avl_node_t   un_avlnode;
5864 } unshare_unmount_node_t;

5866 /* ARGSUSED */
5867 static int
5868 unshare_unmount_compare(const void *larg, const void *rarg, void *unused)
5869 {
5870        const unshare_unmount_node_t *l = larg;
5871        const unshare_unmount_node_t *r = rarg;

5873        return (strcmp(l->un_mountp, r->un_mountp));
5874 }

5876 /*
5877  * Convenience routine used by zfs_do_umount() and manual_unmount().  Given an
5878  * absolute path, find the entry /etc/mnttab, verify that its a ZFS filesystem,
5879  * and unmount it appropriately.
5880  */
5881 static int
5882 unshare_unmount_path(int op, char *path, int flags, boolean_t is_manual)
5883 {
5884        zfs_handle_t *zhp;
5885        int ret = 0;
5886        struct stat64 statbuf;
5887        struct extmnttab entry;
```

```
5888         const char *cmdname = (op == OP_SHARE) ? "unshare" : "unmount";
5889         ino_t path_inode;

5891         /*
5892          * Search for the path in /etc/mnttab.  Rather than looking for the
5893          * specific path, which can be fooled by non-standard paths (i.e. ".."
5894          * or "//"), we stat() the path and search for the corresponding
5895          * (major,minor) device pair.
5896          */
5897         if (stat64(path, &statbuf) != 0) {
5898                 (void) fprintf(stderr, gettext("cannot %s '%s': %s\n"),
5899                     cmdname, path, strerror(errno));
5900                 return (1);
5901         }
5902         path_inode = statbuf.st_ino;

5904         /*
5905          * Search for the given (major,minor) pair in the mount table.
5906          */
5907         rewind(mnttab_file);
5908         while ((ret = getextmntent(mnttab_file, &entry, 0)) == 0) {
5909                 if (entry.mnt_major == major(statbuf.st_dev) &&
5910                     entry.mnt_minor == minor(statbuf.st_dev))
5911                         break;
5912         }
5913         if (ret != 0) {
5914                 if (op == OP_SHARE) {
5915                         (void) fprintf(stderr, gettext("cannot %s '%s': not "
5916                             "currently mounted\n"), cmdname, path);
5917                         return (1);
5918                 }
5919                 (void) fprintf(stderr, gettext("warning: %s not in mnttab\n"),
5920                     path);
5921                 if ((ret = umount2(path, flags)) != 0)
5922                         (void) fprintf(stderr, gettext("%s: %s\n"), path,
5923                             strerror(errno));
5924                 return (ret != 0);
5925         }

5927         if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0) {
5928                 (void) fprintf(stderr, gettext("cannot %s '%s': not a ZFS "
5929                     "filesystem\n"), cmdname, path);
5930                 return (1);
5931         }

5933         if ((zhp = zfs_open(g_zfs, entry.mnt_special,
5934             ZFS_TYPE_FILESYSTEM)) == NULL)
5935                 return (1);

5937         ret = 1;
5938         if (stat64(entry.mnt_mountp, &statbuf) != 0) {
5939                 (void) fprintf(stderr, gettext("cannot %s '%s': %s\n"),
5940                     cmdname, path, strerror(errno));
5941                 goto out;
5942         } else if (statbuf.st_ino != path_inode) {
5943                 (void) fprintf(stderr, gettext("cannot "
5944                     "%s '%s': not a mountpoint\n"), cmdname, path);
5945                 goto out;
5946         }

5948         if (op == OP_SHARE) {
5949                 char nfs_mnt_prop[ZFS_MAXPROPLEN];
5950                 char smbshare_prop[ZFS_MAXPROPLEN];

5952                 verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS, nfs_mnt_prop,
5953                     sizeof (nfs_mnt_prop), NULL, NULL, 0, B_FALSE) == 0);
```



```
5954                 verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB, smbshare_prop,
5955                     sizeof (smbshare_prop), NULL, NULL, 0, B_FALSE) == 0);

5957                 if (strcmp(nfs_mnt_prop, "off") == 0 &&
5958                     strcmp(smbshare_prop, "off") == 0) {
5959                         (void) fprintf(stderr, gettext("cannot unshare "
5960                             "'%s': legacy share\n"), path);
5961                         (void) fprintf(stderr, gettext("use "
5962                             "unshare(1M) to unshare this filesystem\n"));
5963                 } else if (!zfs_is_shared(zhp)) {
5964                         (void) fprintf(stderr, gettext("cannot unshare '%s': "
5965                             "not currently shared\n"), path);
5966                 } else {
5967                         ret = zfs_unshareall_bypath(zhp, path);
5968                 }
5969         } else {
5970                 char mtpt_prop[ZFS_MAXPROPLEN];

5972                 verify(zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mtpt_prop,
5973                     sizeof (mtpt_prop), NULL, NULL, 0, B_FALSE) == 0);

5975                 if (is_manual) {
5976                         ret = zfs_unmount(zhp, NULL, flags);
5977                 } else if (strcmp(mtpt_prop, "legacy") == 0) {
5978                         (void) fprintf(stderr, gettext("cannot unmount "
5979                             "'%s': legacy mountpoint\n"),
5980                             zfs_get_name(zhp));
5981                         (void) fprintf(stderr, gettext("use umount(1M) "
5982                             "to unmount this filesystem\n"));
5983                 } else {
5984                         ret = zfs_unmountall(zhp, flags);
5985                 }
5986         }

5988 out:
5989         zfs_close(zhp);

5991         return (ret != 0);
5992 }

5994 /*
5995  * Generic callback for unsharing or unmounting a filesystem.
5996  */
5997 static int
5998 unshare_unmount(int op, int argc, char **argv)
5999 {
6000         int do_all = 0;
6001         int flags = 0;
6002         int ret = 0;
6003         int c;
6004         zfs_handle_t *zhp;
6005         char nfs_mnt_prop[ZFS_MAXPROPLEN];
6006         char sharesmb[ZFS_MAXPROPLEN];

6008         /* check options */
6009         while ((c = getopt(argc, argv, op == OP_SHARE ? "a" : "af")) != -1) {
6010                 switch (c) {
6011                 case 'a':
6012                         do_all = 1;
6013                         break;
6014                 case 'f':
6015                         flags = MS_FORCE;
6016                         break;
6017                 case '?':
6018                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
6019                             optopt);
```

```
6020                        usage(B_FALSE);
6021                }
6022        }

6024        argc -= optind;
6025        argv += optind;

6027        if (do_all) {
6028                /*
6029                 * We could make use of zfs_for_each() to walk all datasets in
6030                 * the system, but this would be very inefficient, especially
6031                 * since we would have to linearly search /etc/mnttab for each
6032                 * one.  Instead, do one pass through /etc/mnttab looking for
6033                 * zfs entries and call zfs_unmount() for each one.
6034                 *
6035                 * Things get a little tricky if the administrator has created
6036                 * mountpoints beneath other ZFS filesystems.  In this case, we
6037                 * have to unmount the deepest filesystems first.  To accomplish
6038                 * this, we place all the mountpoints in an AVL tree sorted by
6039                 * the special type (dataset name), and walk the result in
6040                 * reverse to make sure to get any snapshots first.
6041                 */
6042                struct mnttab entry;
6043                uu_avl_pool_t *pool;
6044                uu_avl_t *tree;
6045                unshare_unmount_node_t *node;
6046                uu_avl_index_t idx;
6047                uu_avl_walk_t *walk;

6049                if (argc != 0) {
6050                        (void) fprintf(stderr, gettext("too many arguments\n"));
6051                        usage(B_FALSE);
6052                }

6054                if (((pool = uu_avl_pool_create("unmount_pool",
6055                    sizeof (unshare_unmount_node_t),
6056                    offsetof(unshare_unmount_node_t, un_avlnode),
6057                    unshare_unmount_compare, UU_DEFAULT)) == NULL) ||
6058                    ((tree = uu_avl_create(pool, NULL, UU_DEFAULT)) == NULL))
6059                        nomem();

6061                rewind(mnttab_file);
6062                while (getmntent(mnttab_file, &entry) == 0) {

6064                        /* ignore non-ZFS entries */
6065                        if (strcmp(entry.mnt_fstype, MNTTYPE_ZFS) != 0)
6066                                continue;

6068                        /* ignore snapshots */
6069                        if (strchr(entry.mnt_special, '@') != NULL)
6070                                continue;

6072                        if ((zhp = zfs_open(g_zfs, entry.mnt_special,
6073                            ZFS_TYPE_FILESYSTEM)) == NULL) {
6074                                ret = 1;
6075                                continue;
6076                        }

6078                        switch (op) {
6079                        case OP_SHARE:
6080                                verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS,
6081                                    nfs_mnt_prop,
6082                                    sizeof (nfs_mnt_prop),
6083                                    NULL, NULL, 0, B_FALSE) == 0);
6084                                if (strcmp(nfs_mnt_prop, "off") != 0)
6085                                        break;
```



```
6086                                verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB,
6087                                    nfs_mnt_prop,
6088                                    sizeof (nfs_mnt_prop),
6089                                    NULL, NULL, 0, B_FALSE) == 0);
6090                                if (strcmp(nfs_mnt_prop, "off") == 0)
6091                                        continue;
6092                                break;
6093                        case OP_MOUNT:
6094                                /* Ignore legacy mounts */
6095                                verify(zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT,
6096                                    nfs_mnt_prop,
6097                                    sizeof (nfs_mnt_prop),
6098                                    NULL, NULL, 0, B_FALSE) == 0);
6099                                if (strcmp(nfs_mnt_prop, "legacy") == 0)
6100                                        continue;
6101                                /* Ignore canmount=noauto mounts */
6102                                if (zfs_prop_get_int(zhp, ZFS_PROP_CANMOUNT) ==
6103                                    ZFS_CANMOUNT_NOAUTO)
6104                                        continue;
6105                        default:
6106                                break;
6107                        }

6109                        node = safe_malloc(sizeof (unshare_unmount_node_t));
6110                        node->un_zhp = zhp;
6111                        node->un_mountp = safe_strdup(entry.mnt_mountp);

6113                        uu_avl_node_init(node, &node->un_avlnode, pool);

6115                        if (uu_avl_find(tree, node, NULL, &idx) == NULL) {
6116                                uu_avl_insert(tree, node, idx);
6117                        } else {
6118                                zfs_close(node->un_zhp);
6119                                free(node->un_mountp);
6120                                free(node);
6121                        }
6122                }

6124                /*
6125                 * Walk the AVL tree in reverse, unmounting each filesystem and
6126                 * removing it from the AVL tree in the process.
6127                 */
6128                if ((walk = uu_avl_walk_start(tree,
6129                    UU_WALK_REVERSE | UU_WALK_ROBUST)) == NULL)
6130                        nomem();

6132                while ((node = uu_avl_walk_next(walk)) != NULL) {
6133                        uu_avl_remove(tree, node);

6135                        switch (op) {
6136                        case OP_SHARE:
6137                                if (zfs_unshareall_bypath(node->un_zhp,
6138                                    node->un_mountp) != 0)
6139                                        ret = 1;
6140                                break;

6142                        case OP_MOUNT:
6143                                if (zfs_unmount(node->un_zhp,
6144                                    node->un_mountp, flags) != 0)
6145                                        ret = 1;
6146                                break;
6147                        }

6149                        zfs_close(node->un_zhp);
6150                        free(node->un_mountp);
6151                        free(node);
```

```
6152                }

6154                uu_avl_walk_end(walk);
6155                uu_avl_destroy(tree);
6156                uu_avl_pool_destroy(pool);

6158        } else {
6159                if (argc != 1) {
6160                        if (argc == 0)
6161                                (void) fprintf(stderr,
6162                                    gettext("missing filesystem argument\n"));
6163                        else
6164                                (void) fprintf(stderr,
6165                                    gettext("too many arguments\n"));
6166                        usage(B_FALSE);
6167                }

6169                /*
6170                 * We have an argument, but it may be a full path or a ZFS
6171                 * filesystem.  Pass full paths off to unmount_path() (shared by
6172                 * manual_unmount), otherwise open the filesystem and pass to
6173                 * zfs_unmount().
6174                 */
6175                if (argv[0][0] == '/')
6176                        return (unshare_unmount_path(op, argv[0],
6177                            flags, B_FALSE));

6179                if ((zhp = zfs_open(g_zfs, argv[0],
6180                    ZFS_TYPE_FILESYSTEM)) == NULL)
6181                        return (1);

6183                verify(zfs_prop_get(zhp, op == OP_SHARE ?
6184                    ZFS_PROP_SHARENFS : ZFS_PROP_MOUNTPOINT,
6185                    nfs_mnt_prop, sizeof (nfs_mnt_prop), NULL,
6186                    NULL, 0, B_FALSE) == 0);

6188                switch (op) {
6189                case OP_SHARE:
6190                        verify(zfs_prop_get(zhp, ZFS_PROP_SHARENFS,
6191                            nfs_mnt_prop,
6192                            sizeof (nfs_mnt_prop),
6193                            NULL, NULL, 0, B_FALSE) == 0);
6194                        verify(zfs_prop_get(zhp, ZFS_PROP_SHARESMB,
6195                            sharesmb, sizeof (sharesmb), NULL, NULL,
6196                            0, B_FALSE) == 0);

6198                        if (strcmp(nfs_mnt_prop, "off") == 0 &&
6199                            strcmp(sharesmb, "off") == 0) {
6200                                (void) fprintf(stderr, gettext("cannot "
6201                                    "unshare '%s': legacy share\n"),
6202                                    zfs_get_name(zhp));
6203                                (void) fprintf(stderr, gettext("use "
6204                                    "unshare(1M) to unshare this "
6205                                    "filesystem\n"));
6206                                ret = 1;
6207                        } else if (!zfs_is_shared(zhp)) {
6208                                (void) fprintf(stderr, gettext("cannot "
6209                                    "unshare '%s': not currently "
6210                                    "shared\n"), zfs_get_name(zhp));
6211                                ret = 1;
6212                        } else if (zfs_unshareall(zhp) != 0) {
6213                                ret = 1;
6214                        }
6215                        break;

6217                case OP_MOUNT:
```



```
6218                        if (strcmp(nfs_mnt_prop, "legacy") == 0) {
6219                                (void) fprintf(stderr, gettext("cannot "
6220                                    "unmount '%s': legacy "
6221                                    "mountpoint\n"), zfs_get_name(zhp));
6222                                (void) fprintf(stderr, gettext("use "
6223                                    "umount(1M) to unmount this "
6224                                    "filesystem\n"));
6225                                ret = 1;
6226                        } else if (!zfs_is_mounted(zhp, NULL)) {
6227                                (void) fprintf(stderr, gettext("cannot "
6228                                    "unmount '%s': not currently "
6229                                    "mounted\n"),
6230                                    zfs_get_name(zhp));
6231                                ret = 1;
6232                        } else if (zfs_unmountall(zhp, flags) != 0) {
6233                                ret = 1;
6234                        }
6235                        break;
6236                }

6238                zfs_close(zhp);
6239        }

6241        return (ret);
6242 }

6244 /*
6245  * zfs unmount -a
6246  * zfs unmount filesystem
6247  *
6248  * Unmount all filesystems, or a specific ZFS filesystem.
6249  */
6250 static int
6251 zfs_do_unmount(int argc, char **argv)
6252 {
6253        return (unshare_unmount(OP_MOUNT, argc, argv));
6254 }

6256 /*
6257  * zfs unshare -a
6258  * zfs unshare filesystem
6259  *
6260  * Unshare all filesystems, or a specific ZFS filesystem.
6261  */
6262 static int
6263 zfs_do_unshare(int argc, char **argv)
6264 {
6265        return (unshare_unmount(OP_SHARE, argc, argv));
6266 }

6268 /*
6269  * Called when invoked as /etc/fs/zfs/mount.  Do the mount if the mountpoint is
6270  * 'legacy'.  Otherwise, complain that use should be using 'zfs mount'.
6271  */
6272 static int
6273 manual_mount(int argc, char **argv)
6274 {
6275        zfs_handle_t *zhp;
6276        char mountpoint[ZFS_MAXPROPLEN];
6277        char mntopts[MNT_LINE_MAX] = { '\0' };
6278        int ret = 0;
6279        int c;
6280        int flags = 0;
6281        char *dataset, *path;

6283        /* check options */
```

```
6284         while ((c = getopt(argc, argv, ":mo:O")) != -1) {
6285                 switch (c) {
6286                 case 'o':
6287                         (void) strlcpy(mntopts, optarg, sizeof (mntopts));
6288                         break;
6289                 case 'O':
6290                         flags |= MS_OVERLAY;
6291                         break;
6292                 case 'm':
6293                         flags |= MS_NOMNTTAB;
6294                         break;
6295                 case ':':
6296                         (void) fprintf(stderr, gettext("missing argument for "
6297                             "'%c' option\n"), optopt);
6298                         usage(B_FALSE);
6299                         break;
6300                 case '?':
6301                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
6302                             optopt);
6303                         (void) fprintf(stderr, gettext("usage: mount [-o opts] "
6304                             "<path>\n"));
6305                         return (2);
6306                 }
6307         }

6309         argc -= optind;
6310         argv += optind;

6312         /* check that we only have two arguments */
6313         if (argc != 2) {
6314                 if (argc == 0)
6315                         (void) fprintf(stderr, gettext("missing dataset "
6316                             "argument\n"));
6317                 else if (argc == 1)
6318                         (void) fprintf(stderr,
6319                             gettext("missing mountpoint argument\n"));
6320                 else
6321                         (void) fprintf(stderr, gettext("too many arguments\n"));
6322                 (void) fprintf(stderr, "usage: mount <dataset> <mountpoint>\n");
6323                 return (2);
6324         }

6326         dataset = argv[0];
6327         path = argv[1];

6329         /* try to open the dataset */
6330         if ((zhp = zfs_open(g_zfs, dataset, ZFS_TYPE_FILESYSTEM)) == NULL)
6331                 return (1);

6333         (void) zfs_prop_get(zhp, ZFS_PROP_MOUNTPOINT, mountpoint,
6334             sizeof (mountpoint), NULL, NULL, 0, B_FALSE);

6336         /* check for legacy mountpoint and complain appropriately */
6337         ret = 0;
6338         if (strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) == 0) {
6339                 if (mount(dataset, path, MS_OPTIONSTR | flags, MNTTYPE_ZFS,
6340                     NULL, 0, mntopts, sizeof (mntopts)) != 0) {
6341                         (void) fprintf(stderr, gettext("mount failed: %s\n"),
6342                             strerror(errno));
6343                         ret = 1;
6344                 }
6345         } else {
6346                 (void) fprintf(stderr, gettext("filesystem '%s' cannot be "
6347                     "mounted using 'mount -F zfs'\n"), dataset);
6348                 (void) fprintf(stderr, gettext("Use 'zfs set mountpoint=%s' "
6349                     "instead.\n"), path);
```



```
6350                 (void) fprintf(stderr, gettext("If you must use 'mount -F zfs' "
6351                     "or /etc/vfstab, use 'zfs set mountpoint=legacy'.\n"));
6352                 (void) fprintf(stderr, gettext("See zfs(1M) for more "
6353                     "information.\n"));
6354                 ret = 1;
6355         }

6357         return (ret);
6358 }

6360 /*
6361  * Called when invoked as /etc/fs/zfs/umount.  Unlike a manual mount, we allow
6362  * unmounts of non-legacy filesystems, as this is the dominant administrative
6363  * interface.
6364  */
6365 static int
6366 manual_unmount(int argc, char **argv)
6367 {
6368         int flags = 0;
6369         int c;

6371         /* check options */
6372         while ((c = getopt(argc, argv, "f")) != -1) {
6373                 switch (c) {
6374                 case 'f':
6375                         flags = MS_FORCE;
6376                         break;
6377                 case '?':
6378                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
6379                             optopt);
6380                         (void) fprintf(stderr, gettext("usage: unmount [-f] "
6381                             "<path>\n"));
6382                         return (2);
6383                 }
6384         }

6386         argc -= optind;
6387         argv += optind;

6389         /* check arguments */
6390         if (argc != 1) {
6391                 if (argc == 0)
6392                         (void) fprintf(stderr, gettext("missing path "
6393                             "argument\n"));
6394                 else
6395                         (void) fprintf(stderr, gettext("too many arguments\n"));
6396                 (void) fprintf(stderr, gettext("usage: unmount [-f] <path>\n"));
6397                 return (2);
6398         }

6400         return (unshare_unmount_path(OP_MOUNT, argv[0], flags, B_TRUE));
6401 }

6403 static int
6404 find_command_idx(char *command, int *idx)
6405 {
6406         int i;

6408         for (i = 0; i < NCOMMAND; i++) {
6409                 if (command_table[i].name == NULL)
6410                         continue;

6412                 if (strcmp(command, command_table[i].name) == 0) {
6413                         *idx = i;
6414                         return (0);
6415                 }
```

```
6416         }
6417         return (1);
6418 }

6420 static int
6421 zfs_do_diff(int argc, char **argv)
6422 {
6423         zfs_handle_t *zhp;
6424         int flags = 0;
6425         char *tosnap = NULL;
6426         char *fromsnap = NULL;
6427         char *atp, *copy;
6428         int err = 0;
6429         int c;

6431         while ((c = getopt(argc, argv, "FHt")) != -1) {
6432                 switch (c) {
6433                 case 'F':
6434                         flags |= ZFS_DIFF_CLASSIFY;
6435                         break;
6436                 case 'H':
6437                         flags |= ZFS_DIFF_PARSEABLE;
6438                         break;
6439                 case 't':
6440                         flags |= ZFS_DIFF_TIMESTAMP;
6441                         break;
6442                 default:
6443                         (void) fprintf(stderr,
6444                             gettext("invalid option '%c'\n"), optopt);
6445                         usage(B_FALSE);
6446                 }
6447         }

6449         argc -= optind;
6450         argv += optind;

6452         if (argc < 1) {
6453                 (void) fprintf(stderr,
6454                 gettext("must provide at least one snapshot name\n"));
6455                 usage(B_FALSE);
6456         }

6458         if (argc > 2) {
6459                 (void) fprintf(stderr, gettext("too many arguments\n"));
6460                 usage(B_FALSE);
6461         }

6463         fromsnap = argv[0];
6464         tosnap = (argc == 2) ? argv[1] : NULL;

6466         copy = NULL;
6467         if (*fromsnap != '@')
6468                 copy = strdup(fromsnap);
6469         else if (tosnap)
6470                 copy = strdup(tosnap);
6471         if (copy == NULL)
6472                 usage(B_FALSE);

6474         if (atp = strchr(copy, '@'))
6475                 *atp = '\0';

6477         if ((zhp = zfs_open(g_zfs, copy, ZFS_TYPE_FILESYSTEM)) == NULL)
6478                 return (1);

6480         free(copy);
```



```
6482         /*
6483          * Ignore SIGPIPE so that the library can give us
6484          * information on any failure
6485          */
6486         (void) sigignore(SIGPIPE);

6488         err = zfs_show_diffs(zhp, STDOUT_FILENO, fromsnap, tosnap, flags);

6490         zfs_close(zhp);

6492         return (err != 0);
6493 }

6495 int
6496 main(int argc, char **argv)
6497 {
6498         int ret = 0;
6499         int i;
6500         char *progname;
6501         char *cmdname;

6503         (void) setlocale(LC_ALL, "");
6504         (void) textdomain(TEXT_DOMAIN);

6506         opterr = 0;

6508         if ((g_zfs = libzfs_init()) == NULL) {
6509                 (void) fprintf(stderr, gettext("internal error: failed to "
6510                     "initialize ZFS library\n"));
6511                 return (1);
6512         }

6514         zfs_save_arguments(argc, argv, history_str, sizeof (history_str));

6516         libzfs_print_on_error(g_zfs, B_TRUE);

6518         if ((mnttab_file = fopen(MNTTAB, "r")) == NULL) {
6519                 (void) fprintf(stderr, gettext("internal error: unable to "
6520                     "open %s\n"), MNTTAB);
6521                 return (1);
6522         }

6524         /*
6525          * This command also doubles as the /etc/fs mount and unmount program.
6526          * Determine if we should take this behavior based on argv[0].
6527          */
6528         progname = basename(argv[0]);
6529         if (strcmp(progname, "mount") == 0) {
6530                 ret = manual_mount(argc, argv);
6531         } else if (strcmp(progname, "umount") == 0) {
6532                 ret = manual_unmount(argc, argv);
6533         } else {
6534                 /*
6535                  * Make sure the user has specified some command.
6536                  */
6537                 if (argc < 2) {
6538                         (void) fprintf(stderr, gettext("missing command\n"));
6539                         usage(B_FALSE);
6540                 }

6542                 cmdname = argv[1];

6544                 /*
6545                  * The 'umount' command is an alias for 'unmount'
6546                  */
6547                 if (strcmp(cmdname, "umount") == 0)
```

```
6548                        cmdname = "unmount";

6550                /*
6551                 * The 'recv' command is an alias for 'receive'
6552                 */
6553                if (strcmp(cmdname, "recv") == 0)
6554                        cmdname = "receive";

6556                /*
6557                 * Special case '-?'
6558                 */
6559                if (strcmp(cmdname, "-?") == 0)
6560                        usage(B_TRUE);

6562                /*
6563                 * Run the appropriate command.
6564                 */
6565                libzfs_mnttab_cache(g_zfs, B_TRUE);
6566                if (find_command_idx(cmdname, &i) == 0) {
6567                        current_command = &command_table[i];
6568                        ret = command_table[i].func(argc - 1, argv + 1);
6569                } else if (strchr(cmdname, '=') != NULL) {
6570                        verify(find_command_idx("set", &i) == 0);
6571                        current_command = &command_table[i];
6572                        ret = command_table[i].func(argc, argv);
6573                } else {
6574                        (void) fprintf(stderr, gettext("unrecognized "
6575                            "command '%s'\n"), cmdname);
6576                        usage(B_FALSE);
6577                }
6578                libzfs_mnttab_cache(g_zfs, B_FALSE);
6579        }

6581        (void) fclose(mnttab_file);

6583        if (ret == 0 && log_history)
6584                (void) zpool_log_history(g_zfs, history_str);

6586        libzfs_fini(g_zfs);

6588        /*
6589         * The 'ZFS_ABORT' environment variable causes us to dump core on exit
6590         * for the purposes of running ::findleaks.
6591         */
6592        if (getenv("ZFS_ABORT") != NULL) {
6593                (void) printf("dumping core by request\n");
6594                abort();
6595        }

6597        return (ret);
6598 }
```

```
**********************************************************
   26949 Fri Oct 26 17:09:23 2012
new/usr/src/lib/libzfs/common/libzfs.h
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
______unchanged_portion_omitted_

 542 void libzfs_add_handle(get_all_cb_t *, zfs_handle_t *);
 543 int libzfs_dataset_cmp(const void *, const void *);

 545 /*
 546  * Functions to create and destroy datasets.
 547  */
 548 extern int zfs_create(libzfs_handle_t *, const char *, zfs_type_t,
 549     nvlist_t *);
 550 extern int zfs_create_ancestors(libzfs_handle_t *, const char *);
 551 extern int zfs_destroy(zfs_handle_t *, boolean_t);
 552 extern int zfs_destroy_snaps(zfs_handle_t *, char *, boolean_t);
 553 extern int zfs_destroy_snaps_nvl(zfs_handle_t *, nvlist_t *, boolean_t);
 554 extern int zfs_clone(zfs_handle_t *, const char *, nvlist_t *);
 555 extern int zfs_snapshot(libzfs_handle_t *, const char *, boolean_t, nvlist_t *);
 556 extern int zfs_snapshot_nvl(libzfs_handle_t *hdl, nvlist_t *snaps,
 557     nvlist_t *props);
 558 extern int zfs_rollback(zfs_handle_t *, zfs_handle_t *, boolean_t);
 559 extern int zfs_rename(zfs_handle_t *, const char *, boolean_t, boolean_t);

 561 typedef struct sendflags {
 562         /* print informational messages (ie, -v was specified) */
 563         boolean_t verbose;

 565         /* recursive send  (ie, -R) */
 566         boolean_t replicate;

 568         /* for incrementals, do all intermediate snapshots */
 569         boolean_t doall;

 571         /* if dataset is a clone, do incremental from its origin */
 572         boolean_t fromorigin;

 574         /* do deduplication */
 575         boolean_t dedup;

 577         /* send properties (ie, -p) */
 578         boolean_t props;

 580         /* do not send (no-op, ie. -n) */
 581         boolean_t dryrun;

 583         /* parsable verbose output (ie. -P) */
 584         boolean_t parsable;

 586         /* show progress (ie. -v) */
 587         boolean_t progress;

 589         /* send output as FAR-stream */
 590         boolean_t far;
 591 #endif /* ! codereview */
 592 } sendflags_t;

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

 596 extern int zfs_send(zfs_handle_t *, const char *, const char *,
```



```
 597     sendflags_t *, int, snapfilter_cb_t, void *, nvlist_t **);

 599 extern int zfs_promote(zfs_handle_t *);
 600 extern int zfs_hold(zfs_handle_t *, const char *, const char *, boolean_t,
 601     boolean_t, boolean_t, int, uint64_t, uint64_t);
 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 */
```

```
**********************************************************
   84681 Fri Oct 26 17:09:23 2012
new/usr/src/lib/libzfs/common/libzfs_sendrecv.c
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
______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, far;
 792         boolean_t verbose, dryrun, parsable, progress;
 793         int outfd;
 794         boolean_t err;
 795         nvlist_t *fss;
 796         avl_tree_t *fsavl;
 797         snapfilter_cb_t *filter_cb;
 798         void *filter_cb_arg;
 799         nvlist_t *debugnv;
 800         char holdtag[ZFS_MAXNAMELEN];
 801         int cleanup_fd;
 802         uint64_t size;
 803 } send_dump_data_t;
______unchanged_portion_omitted_

 865 /*
 866  * Dumps a backup of the given snapshot (incremental from fromsnap if it's not
 867  * NULL) to the file descriptor specified by outfd.
 868  */
 869 static int
 870 dump_ioctl(zfs_handle_t *zhp, const char *fromsnap, uint64_t fromsnap_obj,
 871     boolean_t fromorigin, int outfd, int far, nvlist_t *debugnv)
 871     boolean_t fromorigin, int outfd, nvlist_t *debugnv)
 872 {
 873         zfs_cmd_t zc = { 0 };
 874         libzfs_handle_t *hdl = zhp->zfs_hdl;
 875         nvlist_t *thisdbg;

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

 880         (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
 881         zc.zc_cookie = outfd;
 882         zc.zc_obj = fromorigin;
 883         zc.zc_sendobj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
 884         zc.zc_fromobj = fromsnap_obj;
 885         zc.zc_guid = far ? 2 : 0;
 886 #endif /* ! codereview */

 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);

 956         if (sdd->dryrun)
 957                 return (0);

 959         /*
```

```
960          * zfs_send() only opens a cleanup_fd for sends that need it,
961          * e.g. replication and doall.
962          */
963         if (sdd->cleanup_fd == -1)
964                 return (0);

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

971         /*
972          * It's OK if the parent no longer exists.  The send code will
973          * handle that error.
974          */
975         if (pzhp) {
976                 error = zfs_hold(pzhp, thissnap, sdd->holdtag,
977                     B_FALSE, B_TRUE, B_TRUE, sdd->cleanup_fd,
978                     zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID),
979                     zfs_prop_get_int(zhp, ZFS_PROP_CREATETXG));
980                 zfs_close(pzhp);
981         }

983         return (error);
984 }

986 static void *
987 send_progress_thread(void *arg)
988 {
989         progress_arg_t *pa = arg;

991         zfs_cmd_t zc = { 0 };
992         zfs_handle_t *zhp = pa->pa_zhp;
993         libzfs_handle_t *hdl = zhp->zfs_hdl;
994         unsigned long long bytes;
995         char buf[16];

997         time_t t;
998         struct tm *tm;

1000        assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
1001        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

1003        if (!pa->pa_parsable)
1004                (void) fprintf(stderr, "TIME        SENT   SNAPSHOT\n");

1006        /*
1007         * Print the progress from ZFS_IOC_SEND_PROGRESS every second.
1008         */
1009        for (;;) {
1010                (void) sleep(1);

1012                zc.zc_cookie = pa->pa_fd;
1013                if (zfs_ioctl(hdl, ZFS_IOC_SEND_PROGRESS, &zc) != 0)
1014                        return ((void *)-1);

1016                (void) time(&t);
1017                tm = localtime(&t);
1018                bytes = zc.zc_cookie;

1020                if (pa->pa_parsable) {
1021                        (void) fprintf(stderr, "%02d:%02d:%02d\t%llu\t%s\n",
1022                            tm->tm_hour, tm->tm_min, tm->tm_sec,
1023                            bytes, zhp->zfs_name);
1024                } else {
1025                        zfs_nicenum(bytes, buf, sizeof (buf));
```



```
1026                        (void) fprintf(stderr, "%02d:%02d:%02d   %5s   %s\n",
1027                            tm->tm_hour, tm->tm_min, tm->tm_sec,
1028                            buf, zhp->zfs_name);
1029                }
1030        }
1031 }

1033 static int
1034 dump_snapshot(zfs_handle_t *zhp, void *arg)
1035 {
1036         send_dump_data_t *sdd = arg;
1037         progress_arg_t pa = { 0 };
1038         pthread_t tid;

1040         char *thissnap;
1041         int err;
1042         boolean_t isfromsnap, istosnap, fromorigin;
1043         boolean_t exclude = B_FALSE;

1045         thissnap = strchr(zhp->zfs_name, '@') + 1;
1046         isfromsnap = (sdd->fromsnap != NULL &&
1047             strcmp(sdd->fromsnap, thissnap) == 0);

1049         if (!sdd->seenfrom && isfromsnap) {
1050                 err = hold_for_send(zhp, sdd);
1051                 if (err == 0) {
1052                         sdd->seenfrom = B_TRUE;
1053                         (void) strcpy(sdd->prevsnap, thissnap);
1054                         sdd->prevsnap_obj = zfs_prop_get_int(zhp,
1055                             ZFS_PROP_OBJSETID);
1056                 } else if (err == ENOENT) {
1057                         err = 0;
1058                 }
1059                 zfs_close(zhp);
1060                 return (err);
1061         }

1063         if (sdd->seento || !sdd->seenfrom) {
1064                 zfs_close(zhp);
1065                 return (0);
1066         }

1068         istosnap = (strcmp(sdd->tosnap, thissnap) == 0);
1069         if (istosnap)
1070                 sdd->seento = B_TRUE;

1072         if (!sdd->doall && !isfromsnap && !istosnap) {
1073                 if (sdd->replicate) {
1074                         char *snapname;
1075                         nvlist_t *snapprops;
1076                         /*
1077                          * Filter out all intermediate snapshots except origin
1078                          * snapshots needed to replicate clones.
1079                          */
1080                         nvlist_t *nvfs = fsavl_find(sdd->fsavl,
1081                             zhp->zfs_dmustats.dds_guid, &snapname);

1083                         VERIFY(0 == nvlist_lookup_nvlist(nvfs,
1084                             "snapprops", &snapprops));
1085                         VERIFY(0 == nvlist_lookup_nvlist(snapprops,
1086                             thissnap, &snapprops));
1087                         exclude = !nvlist_exists(snapprops, "is_clone_origin");
1088                 } else {
1089                         exclude = B_TRUE;
1090                 }
1091         }
```

```
1093         /*
1094          * If a filter function exists, call it to determine whether
1095          * this snapshot will be sent.
1096          */
1097         if (exclude || (sdd->filter_cb != NULL &&
1098             sdd->filter_cb(zhp, sdd->filter_cb_arg) == B_FALSE)) {
1099                 /*
1100                  * This snapshot is filtered out.  Don't send it, and don't
1101                  * set prevsnap_obj, so it will be as if this snapshot didn't
1102                  * exist, and the next accepted snapshot will be sent as
1103                  * an incremental from the last accepted one, or as the
1104                  * first (and full) snapshot in the case of a replication,
1105                  * non-incremental send.
1106                  */
1107                 zfs_close(zhp);
1108                 return (0);
1109         }

1111         err = hold_for_send(zhp, sdd);
1112         if (err) {
1113                 if (err == ENOENT)
1114                         err = 0;
1115                 zfs_close(zhp);
1116                 return (err);
1117         }

1119         fromorigin = sdd->prevsnap[0] == '\0' &&
1120             (sdd->fromorigin || sdd->replicate);

1122         if (sdd->verbose) {
1123                 uint64_t size;
1124                 err = estimate_ioctl(zhp, sdd->prevsnap_obj,
1125                     fromorigin, &size);

1127                 if (sdd->parsable) {
1128                         if (sdd->prevsnap[0] != '\0') {
1129                                 (void) fprintf(stderr, "incremental\t%s\t%s",
1130                                     sdd->prevsnap, zhp->zfs_name);
1131                         } else {
1132                                 (void) fprintf(stderr, "full\t%s",
1133                                     zhp->zfs_name);
1134                         }
1135                 } else {
1136                         (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1137                             "send from @%s to %s"),
1138                             sdd->prevsnap, zhp->zfs_name);
1139                 }
1140                 if (err == 0) {
1141                         if (sdd->parsable) {
1142                                 (void) fprintf(stderr, "\t%llu\n",
1143                                     (longlong_t)size);
1144                         } else {
1145                                 char buf[16];
1146                                 zfs_nicenum(size, buf, sizeof (buf));
1147                                 (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1148                                     " estimated size is %s\n"), buf);
1149                         }
1150                         sdd->size += size;
1151                 } else {
1152                         (void) fprintf(stderr, "\n");
1153                 }
1154         }

1156         if (!sdd->dryrun) {
1157                 /*
```



```
1158                  * If progress reporting is requested, spawn a new thread to
1159                  * poll ZFS_IOC_SEND_PROGRESS at a regular interval.
1160                  */
1161                 if (sdd->progress) {
1162                         pa.pa_zhp = zhp;
1163                         pa.pa_fd = sdd->outfd;
1164                         pa.pa_parsable = sdd->parsable;

1166                         if (err = pthread_create(&tid, NULL,
1167                             send_progress_thread, &pa)) {
1168                                 zfs_close(zhp);
1169                                 return (err);
1170                         }
1171                 }

1173                 err = dump_ioctl(zhp, sdd->prevsnap, sdd->prevsnap_obj,
1174                     fromorigin, sdd->outfd, sdd->far, sdd->debugnv);
 885                     fromorigin, sdd->outfd, sdd->debugnv);

1176                 if (sdd->progress) {
1177                         (void) pthread_cancel(tid);
1178                         (void) pthread_join(tid, NULL);
1179                 }
1180         }

1182         (void) strcpy(sdd->prevsnap, thissnap);
1183         sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
1184         zfs_close(zhp);
1185         return (err);
1186 }
______unchanged_portion_omitted_

1358 /*
1359  * Generate a send stream for the dataset identified by the argument zhp.
1360  *
1361  * The content of the send stream is the snapshot identified by
1362  * 'tosnap'.  Incremental streams are requested in two ways:
1363  *     - from the snapshot identified by "fromsnap" (if non-null) or
1364  *     - from the origin of the dataset identified by zhp, which must
1365  *       be a clone.  In this case, "fromsnap" is null and "fromorigin"
1366  *       is TRUE.
1367  *
1368  * The send stream is recursive (i.e. dumps a hierarchy of snapshots) and
1369  * uses a special header (with a hdrtype field of DMU_COMPOUNDSTREAM)
1370  * if "replicate" is set.  If "doall" is set, dump all the intermediate
1371  * snapshots. The DMU_COMPOUNDSTREAM header is used in the "doall"
1372  * case too. If "props" is set, send properties.
1373  */
1374 int
1375 zfs_send(zfs_handle_t *zhp, const char *fromsnap, const char *tosnap,
1376     sendflags_t *flags, int outfd, snapfilter_cb_t filter_func,
1377     void *cb_arg, nvlist_t **debugnvp)
1378 {
1379         char errbuf[1024];
1380         send_dump_data_t sdd = { 0 };
1381         int err = 0;
1382         nvlist_t *fss = NULL;
1383         avl_tree_t *fsavl = NULL;
1384         static uint64_t holdseq;
1385         int spa_version;
1386         pthread_t tid;
1387         int pipefd[2];
1388         dedup_arg_t dda = { 0 };
1389         int featureflags = 0;

1391         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
```

```
1392             "cannot send '%s'"), zhp->zfs_name);

1394         if (fromsnap && fromsnap[0] == '\0') {
1395                 zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
1396                     "zero-length incremental source"));
1397                 return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
1398         }

1400         if (zhp->zfs_type == ZFS_TYPE_FILESYSTEM) {
1401                 uint64_t version;
1402                 version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1403                 if (version >= ZPL_VERSION_SA) {
1404                         featureflags |= DMU_BACKUP_FEATURE_SA_SPILL;
1405                 }
1406         }

1408         if (flags->dedup && !flags->dryrun) {
1409                 featureflags |= (DMU_BACKUP_FEATURE_DEDUP |
1410                     DMU_BACKUP_FEATURE_DEDUPPROPS);
1411                 if (err = pipe(pipefd)) {
1412                         zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1413                         return (zfs_error(zhp->zfs_hdl, EZFS_PIPEFAILED,
1414                             errbuf));
1415                 }
1416                 dda.outputfd = outfd;
1417                 dda.inputfd = pipefd[1];
1418                 dda.dedup_hdl = zhp->zfs_hdl;
1419                 if (err = pthread_create(&tid, NULL, cksummer, &dda)) {
1420                         (void) close(pipefd[0]);
1421                         (void) close(pipefd[1]);
1422                         zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1423                         return (zfs_error(zhp->zfs_hdl,
1424                             EZFS_THREADCREATEFAILED, errbuf));
1425                 }
1426         }

1428         if (flags->replicate || flags->doall || flags->props) {
1429                 dmu_replay_record_t drr = { 0 };
1430                 char *packbuf = NULL;
1431                 size_t buflen = 0;
1432                 zio_cksum_t zc = { 0 };

1434                 if (flags->replicate || flags->props) {
1435                         nvlist_t *hdrnv;

1437                         VERIFY(0 == nvlist_alloc(&hdrnv, NV_UNIQUE_NAME, 0));
1438                         if (fromsnap) {
1439                                 VERIFY(0 == nvlist_add_string(hdrnv,
1440                                     "fromsnap", fromsnap));
1441                         }
1442                         VERIFY(0 == nvlist_add_string(hdrnv, "tosnap", tosnap));
1443                         if (!flags->replicate) {
1444                                 VERIFY(0 == nvlist_add_boolean(hdrnv,
1445                                     "not_recursive"));
1446                         }

1448                         err = gather_nvlist(zhp->zfs_hdl, zhp->zfs_name,
1449                             fromsnap, tosnap, flags->replicate, &fss, &fsavl);
1450                         if (err)
1451                                 goto err_out;
1452                         VERIFY(0 == nvlist_add_nvlist(hdrnv, "fss", fss));
1453                         err = nvlist_pack(hdrnv, &packbuf, &buflen,
1454                             NV_ENCODE_XDR, 0);
1455                         if (debugnvp)
1456                                 *debugnvp = hdrnv;
1457                         else
```



```
1458                             nvlist_free(hdrnv);
1459                     if (err) {
1460                             fsavl_destroy(fsavl);
1461                             nvlist_free(fss);
1462                             goto stderr_out;
1463                     }
1464             }

1466             if (!flags->dryrun && !flags->far) {
1177             if (!flags->dryrun) {
1467                     /* write first begin record */
1468                     drr.drr_type = DRR_BEGIN;
1469                     drr.drr_u.drr_begin.drr_magic = DMU_BACKUP_MAGIC;
1470                     DMU_SET_STREAM_HDRTYPE(drr.drr_u.drr_begin.
1471                         drr_versioninfo, DMU_COMPOUNDSTREAM);
1472                     DMU_SET_FEATUREFLAGS(drr.drr_u.drr_begin.
1473                         drr_versioninfo, featureflags);
1474                     (void) snprintf(drr.drr_u.drr_begin.drr_toname,
1475                         sizeof (drr.drr_u.drr_begin.drr_toname),
1476                         "%s@%s", zhp->zfs_name, tosnap);
1477                     drr.drr_payloadlen = buflen;
1478                     err = cksum_and_write(&drr, sizeof (drr), &zc, outfd);

1480                     /* write header nvlist */
1481                     if (err != -1 && packbuf != NULL) {
1482                             err = cksum_and_write(packbuf, buflen, &zc,
1483                                 outfd);
1484                     }
1485                     free(packbuf);
1486                     if (err == -1) {
1487                             fsavl_destroy(fsavl);
1488                             nvlist_free(fss);
1489                             err = errno;
1490                             goto stderr_out;
1491                     }

1493                     /* write end record */
1494                     bzero(&drr, sizeof (drr));
1495                     drr.drr_type = DRR_END;
1496                     drr.drr_u.drr_end.drr_checksum = zc;
1497                     err = write(outfd, &drr, sizeof (drr));
1498                     if (err == -1) {
1499                             fsavl_destroy(fsavl);
1500                             nvlist_free(fss);
1501                             err = errno;
1502                             goto stderr_out;
1503                     }

1505                     err = 0;
1506             }
1507         }

1509         /* dump each stream */
1510         sdd.fromsnap = fromsnap;
1511         sdd.tosnap = tosnap;
1512         if (flags->dedup)
1513                 sdd.outfd = pipefd[0];
1514         else
1515                 sdd.outfd = outfd;
1516         sdd.replicate = flags->replicate;
1517         sdd.doall = flags->doall;
1518         sdd.fromorigin = flags->fromorigin;
1519         sdd.fss = fss;
1520         sdd.fsavl = fsavl;
1521         sdd.verbose = flags->verbose;
1522         sdd.parsable = flags->parsable;
```

```
1523         sdd.progress = flags->progress;
1524         sdd.dryrun = flags->dryrun;
1525         sdd.far = flags->far;
1526 #endif /* ! codereview */
1527         sdd.filter_cb = filter_func;
1528         sdd.filter_cb_arg = cb_arg;
1529         if (debugnvp)
1530                 sdd.debugnv = *debugnvp;

1532         /*
1533          * Some flags require that we place user holds on the datasets that are
1534          * being sent so they don't get destroyed during the send. We can skip
1535          * this step if the pool is imported read-only since the datasets cannot
1536          * be destroyed.
1537          */
1538         if (!flags->dryrun && !zpool_get_prop_int(zfs_get_pool_handle(zhp),
1539             ZPOOL_PROP_READONLY, NULL) &&
1540             zfs_spa_version(zhp, &spa_version) == 0 &&
1541             spa_version >= SPA_VERSION_USERREFS &&
1542             (flags->doall || flags->replicate)) {
1543                 ++holdseq;
1544                 (void) snprintf(sdd.holdtag, sizeof (sdd.holdtag),
1545                     ".send-%d-%llu", getpid(), (u_longlong_t)holdseq);
1546                 sdd.cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
1547                 if (sdd.cleanup_fd < 0) {
1548                         err = errno;
1549                         goto stderr_out;
1550                 }
1551         } else {
1552                 sdd.cleanup_fd = -1;
1553         }
1554         if (flags->verbose) {
1555                 /*
1556                  * Do a verbose no-op dry run to get all the verbose output
1557                  * before generating any data.  Then do a non-verbose real
1558                  * run to generate the streams.
1559                  */
1560                 sdd.dryrun = B_TRUE;
1561                 err = dump_filesystems(zhp, &sdd);
1562                 sdd.dryrun = flags->dryrun;
1563                 sdd.verbose = B_FALSE;
1564                 if (flags->parsable) {
1565                         (void) fprintf(stderr, "size\t%llu\n",
1566                             (longlong_t)sdd.size);
1567                 } else {
1568                         char buf[16];
1569                         zfs_nicenum(sdd.size, buf, sizeof (buf));
1570                         (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1571                             "total estimated size is %s\n"), buf);
1572                 }
1573         }
1574         err = dump_filesystems(zhp, &sdd);
1575         fsavl_destroy(fsavl);
1576         nvlist_free(fss);

1578         if (flags->dedup) {
1579                 (void) close(pipefd[0]);
1580                 (void) pthread_join(tid, NULL);
1581         }

1583         if (sdd.cleanup_fd != -1) {
1584                 VERIFY(0 == close(sdd.cleanup_fd));
1585                 sdd.cleanup_fd = -1;
1586         }

1588         if (!flags->dryrun && !flags->far &&
```



```
1589             (flags->replicate || flags->doall || flags->props)) {
1236         if (!flags->dryrun && (flags->replicate || flags->doall ||
1237             flags->props)) {
1590                 /*
1591                  * write final end record.  NB: want to do this even if
1592                  * there was some error, because it might not be totally
1593                  * failed.
1594                  */
1595                 dmu_replay_record_t drr = { 0 };
1596                 drr.drr_type = DRR_END;
1597                 if (write(outfd, &drr, sizeof (drr)) == -1) {
1598                         return (zfs_standard_error(zhp->zfs_hdl,
1599                             errno, errbuf));
1600                 }
1601         }

1603         return (err || sdd.err);

1605 stderr_out:
1606         err = zfs_standard_error(zhp->zfs_hdl, err, errbuf);
1607 err_out:
1608         if (sdd.cleanup_fd != -1)
1609                 VERIFY(0 == close(sdd.cleanup_fd));
1610         if (flags->dedup) {
1611                 (void) pthread_cancel(tid);
1612                 (void) pthread_join(tid, NULL);
1613                 (void) close(pipefd[0]);
1614         }
1615         return (err);
1616 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   42954 Fri Oct 26 17:09:23 2012
new/usr/src/uts/common/Makefile.files
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
#
# 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) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
# Copyright (c) 2012 by Delphix. All rights reserved.
#

#
# This Makefile defines all file modules for the directory uts/common
# and its children. These are the source files which may be considered
# common to all SunOS systems.

i386_CORE_OBJS += \
                atomic.o        \
                avintr.o        \
                pic.o

sparc_CORE_OBJS +=

COMMON_CORE_OBJS +=             \
                beep.o          \
                bitset.o        \
                bp_map.o        \
                brand.o         \
                cpucaps.o       \
                cmt.o           \
                cmt_policy.o    \
                cpu.o           \
                cpu_event.o     \
                cpu_intr.o      \
                cpu_pm.o        \
                cpupart.o       \
                cap_util.o      \
                disp.o          \
                group.o         \
                kstat_fr.o      \
                iscsiboot_prop.o        \
```



```
                lgrp.o          \
                lgrp_topo.o     \
                mmapobj.o       \
                mutex.o         \
                page_lock.o     \
                page_retire.o   \
                panic.o         \
                param.o         \
                pg.o            \
                pghw.o          \
                putnext.o       \
                rctl_proc.o     \
                rwlock.o        \
                seg_kmem.o      \
                softint.o       \
                string.o        \
                strtol.o        \
                strtoul.o       \
                strtoll.o       \
                strtoull.o      \
                thread_intr.o   \
                vm_page.o       \
                vm_pagelist.o   \
                zlib_obj.o      \
                clock_tick.o

CORE_OBJS +=    $(COMMON_CORE_OBJS) $($(MACH)_CORE_OBJS)

ZLIB_OBJS =     zutil.o zmod.o zmod_subr.o \
                adler32.o crc32.o deflate.o inffast.o \
                inflate.o inftrees.o trees.o

GENUNIX_OBJS += \
                access.o        \
                acl.o           \
                acl_common.o    \
                adjtime.o       \
                alarm.o         \
                aio_subr.o      \
                auditsys.o      \
                audit_core.o    \
                audit_zone.o    \
                audit_memory.o  \
                autoconf.o      \
                avl.o           \
                bdev_dsort.o    \
                bio.o           \
                bitmap.o        \
                blabel.o        \
                brandsys.o      \
                bz2blocksort.o  \
                bz2compress.o   \
                bz2decompress.o \
                bz2randtable.o  \
                bz2bzlib.o      \
                bz2crctable.o   \
                bz2huffman.o    \
                callb.o         \
                callout.o       \
                chdir.o         \
                chmod.o         \
                chown.o         \
                cladm.o         \
                class.o         \
                clock.o         \
                clock_highres.o \
```

```
124                 clock_realtime.o\
125                 close.o         \
126                 compress.o      \
127                 condvar.o       \
128                 conf.o          \
129                 console.o       \
130                 contract.o      \
131                 copyops.o       \
132                 core.o          \
133                 corectl.o       \
134                 cred.o          \
135                 cs_stubs.o      \
136                 dacf.o          \
137                 dacf_clnt.o     \
138                 damap.o \
139                 cyclic.o        \
140                 ddi.o           \
141                 ddifm.o         \
142                 ddi_hp_impl.o   \
143                 ddi_hp_ndi.o    \
144                 ddi_intr.o      \
145                 ddi_intr_impl.o \
146                 ddi_intr_irm.o  \
147                 ddi_nodeid.o    \
148                 ddi_timer.o     \
149                 devcfg.o        \
150                 devcache.o      \
151                 device.o        \
152                 devid.o         \
153                 devid_cache.o   \
154                 devid_scsi.o    \
155                 devid_smp.o     \
156                 devpolicy.o     \
157                 disp_lock.o     \
158                 dnlc.o          \
159                 driver.o        \
160                 dumpsubr.o      \
161                 driver_lyr.o    \
162                 dtrace_subr.o   \
163                 errorq.o        \
164                 etheraddr.o     \
165                 evchannels.o    \
166                 exacct.o        \
167                 exacct_core.o   \
168                 exec.o          \
169                 exit.o          \
170                 fbio.o          \
171                 fcntl.o         \
172                 fdbuffer.o      \
173                 fdsync.o        \
174                 fem.o           \
175                 ffs.o           \
176                 fio.o           \
177                 flock.o         \
178                 fm.o            \
179                 fork.o          \
180                 vpm.o           \
181                 fs_reparse.o    \
182                 fs_subr.o       \
183                 fsflush.o       \
184                 ftrace.o        \
185                 getcwd.o        \
186                 getdents.o      \
187                 getloadavg.o    \
188                 getpagesizes.o  \
189                 getpid.o        \
```



```
190                 gfs.o           \
191                 rusagesys.o     \
192                 gid.o           \
193                 groups.o        \
194                 grow.o          \
195                 hat_refmod.o    \
196                 id32.o          \
197                 id_space.o      \
198                 inet_ntop.o     \
199                 instance.o      \
200                 ioctl.o         \
201                 ip_cksum.o      \
202                 issetugid.o     \
203                 ippconf.o       \
204                 kcpc.o          \
205                 kdi.o           \
206                 kiconv.o        \
207                 klpd.o          \
208                 kmem.o          \
209                 ksyms_snapshot.o        \
210                 l_strplumb.o    \
211                 labelsys.o      \
212                 link.o          \
213                 list.o          \
214                 lockstat_subr.o \
215                 log_sysevent.o  \
216                 logsubr.o       \
217                 lookup.o        \
218                 lseek.o         \
219                 ltos.o          \
220                 lwp.o           \
221                 lwp_create.o    \
222                 lwp_info.o      \
223                 lwp_self.o      \
224                 lwp_sobj.o      \
225                 lwp_timer.o     \
226                 lwpsys.o        \
227                 main.o          \
228                 mmapobjsys.o    \
229                 memcntl.o       \
230                 memstr.o        \
231                 lgrpsys.o       \
232                 mkdir.o         \
233                 mknod.o         \
234                 mount.o         \
235                 move.o          \
236                 msacct.o        \
237                 multidata.o     \
238                 nbmlock.o       \
239                 ndifm.o         \
240                 nice.o          \
241                 netstack.o      \
242                 ntptime.o       \
243                 nvpair.o        \
244                 nvpair_alloc_system.o   \
245                 nvpair_alloc_fixed.o    \
246                 fnvpair.o       \
247                 octet.o         \
248                 open.o          \
249                 p_online.o      \
250                 pathconf.o      \
251                 pathname.o      \
252                 pause.o         \
253                 serializer.o    \
254                 pci_intr_lib.o  \
255                 pci_cap.o       \
```

```
256                 pcifm.o         \
257                 pgrp.o          \
258                 pgrpsys.o       \
259                 pid.o           \
260                 pkp_hash.o      \
261                 policy.o        \
262                 poll.o          \
263                 pool.o          \
264                 pool_pset.o     \
265                 port_subr.o     \
266                 ppriv.o         \
267                 printf.o        \
268                 priocntl.o      \
269                 priv.o          \
270                 priv_const.o    \
271                 proc.o          \
272                 procset.o       \
273                 processor_bind.o        \
274                 processor_info.o        \
275                 profil.o        \
276                 project.o       \
277                 qsort.o         \
278                 rctl.o          \
279                 rctlsys.o       \
280                 readlink.o      \
281                 refstr.o        \
282                 rename.o        \
283                 resolvepath.o   \
284                 retire_store.o  \
285                 process.o       \
286                 rlimit.o        \
287                 rmap.o          \
288                 rw.o            \
289                 rwstlock.o      \
290                 sad_conf.o      \
291                 sid.o           \
292                 sidsys.o        \
293                 sched.o         \
294                 schedctl.o      \
295                 sctp_crc32.o    \
296                 seg_dev.o       \
297                 seg_kp.o        \
298                 seg_kpm.o       \
299                 seg_map.o       \
300                 seg_vn.o        \
301                 seg_spt.o       \
302                 semaphore.o     \
303                 sendfile.o      \
304                 session.o       \
305                 share.o         \
306                 shuttle.o       \
307                 sig.o           \
308                 sigaction.o     \
309                 sigaltstack.o   \
310                 signotify.o     \
311                 sigpending.o    \
312                 sigprocmask.o   \
313                 sigqueue.o      \
314                 sigsendset.o    \
315                 sigsuspend.o    \
316                 sigtimedwait.o  \
317                 sleepq.o        \
318                 sock_conf.o     \
319                 space.o         \
320                 sscanf.o        \
321                 stat.o          \
```



```
322                 statfs.o        \
323                 statvfs.o       \
324                 stol.o          \
325                 str_conf.o      \
326                 strcalls.o      \
327                 stream.o        \
328                 streamio.o      \
329                 strext.o        \
330                 strsubr.o       \
331                 strsun.o        \
332                 subr.o          \
333                 sunddi.o        \
334                 sunmdi.o        \
335                 sunndi.o        \
336                 sunpci.o        \
337                 sunpm.o         \
338                 sundlpi.o       \
339                 suntpi.o        \
340                 swap_subr.o     \
341                 swap_vnops.o    \
342                 symlink.o       \
343                 sync.o          \
344                 sysclass.o      \
345                 sysconfig.o     \
346                 sysent.o        \
347                 sysfs.o         \
348                 systeminfo.o    \
349                 task.o          \
350                 taskq.o         \
351                 tasksys.o       \
352                 time.o          \
353                 timer.o         \
354                 times.o         \
355                 timers.o        \
356                 thread.o        \
357                 tlabel.o        \
358                 tnf_res.o       \
359                 turnstile.o     \
360                 tty_common.o    \
361                 u8_textprep.o   \
362                 uadmin.o        \
363                 uconv.o         \
364                 ucredsys.o      \
365                 uid.o           \
366                 umask.o         \
367                 umount.o        \
368                 uname.o         \
369                 unix_bb.o       \
370                 unlink.o        \
371                 urw.o           \
372                 utime.o         \
373                 utssys.o        \
374                 uucopy.o        \
375                 vfs.o           \
376                 vfs_conf.o      \
377                 vmem.o          \
378                 vm_anon.o       \
379                 vm_as.o         \
380                 vm_meter.o      \
381                 vm_pageout.o    \
382                 vm_pvn.o        \
383                 vm_rm.o         \
384                 vm_seg.o        \
385                 vm_subr.o       \
386                 vm_swap.o       \
387                 vm_usage.o      \
```

```
388                 vnode.o         \
389                 vuid_queue.o    \
390                 vuid_store.o    \
391                 waitq.o         \
392                 watchpoint.o    \
393                 yield.o         \
394                 scsi_confdata.o \
395                 xattr.o         \
396                 xattr_common.o  \
397                 xdr_mblk.o      \
398                 xdr_mem.o       \
399                 xdr.o           \
400                 xdr_array.o     \
401                 xdr_refer.o     \
402                 xhat.o          \
403                 zone.o

405 #
406 #       Stubs for the stand-alone linker/loader
407 #
408 sparc_GENSTUBS_OBJS =   \
409         kobj_stubs.o

411 i386_GENSTUBS_OBJS =

413 COMMON_GENSTUBS_OBJS =

415 GENSTUBS_OBJS += $(COMMON_GENSTUBS_OBJS) $($(MACH)_GENSTUBS_OBJS)

417 #
418 #       DTrace and DTrace Providers
419 #
420 DTRACE_OBJS += dtrace.o dtrace_isa.o dtrace_asm.o

422 SDT_OBJS += sdt_subr.o

424 PROFILE_OBJS += profile.o

426 SYSTRACE_OBJS += systrace.o

428 LOCKSTAT_OBJS += lockstat.o

430 FASTTRAP_OBJS += fasttrap.o fasttrap_isa.o

432 DCPC_OBJS += dcpc.o

434 #
435 #       Driver (pseudo-driver) Modules
436 #
437 IPP_OBJS +=     ippctl.o

439 AUDIO_OBJS += audio_client.o audio_ddi.o audio_engine.o \
440             audio_fltdata.o audio_format.o audio_ctrl.o \
441             audio_grc3.o audio_output.o audio_input.o \
442             audio_oss.o audio_sun.o

444 AUDIOEMU10K_OBJS += audioemu10k.o

446 AUDIOENS_OBJS += audioens.o

448 AUDIOVIA823X_OBJS += audiovia823x.o

450 AUDIOVIA97_OBJS += audiovia97.o

452 AUDIO1575_OBJS += audio1575.o
```



```
454 AUDIO810_OBJS += audio810.o

456 AUDIOCMI_OBJS += audiocmi.o

458 AUDIOCMIHD_OBJS += audiocmihd.o

460 AUDIOHD_OBJS += audiohd.o

462 AUDIOIXP_OBJS += audioixp.o

464 AUDIOLS_OBJS += audiols.o

466 AUDIOP16X_OBJS += audiop16x.o

468 AUDIOPCI_OBJS += audiopci.o

470 AUDIOSOLO_OBJS += audiosolo.o

472 AUDIOTS_OBJS += audiots.o

474 AC97_OBJS += ac97.o ac97_ad.o ac97_alc.o ac97_cmi.o

476 BLKDEV_OBJS += blkdev.o

478 CARDBUS_OBJS += cardbus.o cardbus_hp.o cardbus_cfg.o

480 CONSKBD_OBJS += conskbd.o

482 CONSMS_OBJS +=  consms.o

484 OLDPTY_OBJS +=  tty_ptyconf.o

486 PTC_OBJS +=     tty_pty.o

488 PTSL_OBJS +=    tty_pts.o

490 PTM_OBJS +=     ptm.o

492 MII_OBJS +=     mii.o mii_cicada.o mii_natsemi.o mii_intel.o mii_qualsemi.o \
493                 mii_marvell.o mii_realtek.o mii_other.o

495 PTS_OBJS +=     pts.o

497 PTY_OBJS +=     ptms_conf.o

499 SAD_OBJS +=     sad.o

501 MD4_OBJS +=     md4.o md4_mod.o

503 MD5_OBJS +=     md5.o md5_mod.o

505 SHA1_OBJS +=    sha1.o sha1_mod.o

507 SHA2_OBJS +=    sha2.o sha2_mod.o

509 IPGPC_OBJS +=   classifierddi.o classifier.o filters.o trie.o table.o \
510                 ba_table.o

512 DSCPMK_OBJS +=  dscpmk.o dscpmkddi.o

514 DLCOSMK_OBJS += dlcosmk.o dlcosmkddi.o

516 FLOWACCT_OBJS +=        flowacctddi.o flowacct.o

518 TOKENMT_OBJS += tokenmt.o tokenmtddi.o
```

```
520 TSWTCL_OBJS +=  tswtcl.o tswtclddi.o

522 ARP_OBJS +=     arpddi.o

524 ICMP_OBJS +=    icmpddi.o

526 ICMP6_OBJS +=   icmp6ddi.o

528 RTS_OBJS +=     rtsddi.o

530 IP_ICMP_OBJS =  icmp.o icmp_opt_data.o
531 IP_RTS_OBJS =   rts.o rts_opt_data.o
532 IP_TCP_OBJS =   tcp.o tcp_fusion.o tcp_opt_data.o tcp_sack.o tcp_stats.o \
533                 tcp_misc.o tcp_timers.o tcp_time_wait.o tcp_tpi.o tcp_output.o \
534                 tcp_input.o tcp_socket.o tcp_bind.o tcp_cluster.o tcp_tunables.o
535 IP_UDP_OBJS =   udp.o udp_opt_data.o udp_tunables.o udp_stats.o
536 IP_SCTP_OBJS =  sctp.o sctp_opt_data.o sctp_output.o \
537                 sctp_init.o sctp_input.o sctp_cookie.o \
538                 sctp_conn.o sctp_error.o sctp_snmp.o \
539                 sctp_tunables.o sctp_shutdown.o sctp_common.o \
540                 sctp_timer.o sctp_heartbeat.o sctp_hash.o \
541                 sctp_bind.o sctp_notify.o sctp_asconf.o \
542                 sctp_addr.o tn_ipopt.o tnet.o ip_netinfo.o \
543                 sctp_misc.o
544 IP_ILB_OBJS =   ilb.o ilb_nat.o ilb_conn.o ilb_alg_hash.o ilb_alg_rr.o

546 IP_OBJS +=      igmp.o ipmp.o ip.o ip6.o ip6_asp.o ip6_if.o ip6_ire.o \
547                 ip6_rts.o ip_if.o ip_ire.o ip_listutils.o ip_mroute.o \
548                 ip_multi.o ip2mac.o ip_ndp.o ip_rts.o ip_srcid.o \
549                 ipddi.o ipdrop.o mi.o nd.o tunables.o optcom.o snmpcom.o \
550                 ipsec_loader.o spd.o ipclassifier.o inet_common.o ip_squeue.o \
551                 squeue.o ip_sadb.o ip_ftable.o proto_set.o radix.o ip_dummy.o \
552                 ip_helper_stream.o ip_tunables.o \
553                 ip_output.o ip_input.o ip6_input.o ip6_output.o ip_arp.o \
554                 conn_opt.o ip_attr.o ip_dce.o \
555                 $(IP_ICMP_OBJS) \
556                 $(IP_RTS_OBJS) \
557                 $(IP_TCP_OBJS) \
558                 $(IP_UDP_OBJS) \
559                 $(IP_SCTP_OBJS) \
560                 $(IP_ILB_OBJS)

562 IP6_OBJS +=     ip6ddi.o

564 HOOK_OBJS +=    hook.o

566 NETI_OBJS +=    neti_impl.o neti_mod.o neti_stack.o

568 KEYSOCK_OBJS += keysockddi.o keysock.o keysock_opt_data.o

570 IPNET_OBJS +=   ipnet.o ipnet_bpf.o

572 SPDSOCK_OBJS += spdsockddi.o spdsock.o spdsock_opt_data.o

574 IPSECESP_OBJS += ipsecespddi.o ipsecesp.o

576 IPSECAH_OBJS += ipsecahddi.o ipsecah.o sadb.o

578 SPPP_OBJS +=    sppp.o sppp_dlpi.o sppp_mod.o s_common.o

580 SPPPTUN_OBJS += sppptun.o sppptun_mod.o

582 SPPPASYN_OBJS += spppasyn.o spppasyn_mod.o

584 SPPPCOMP_OBJS += spppcomp.o spppcomp_mod.o deflate.o bsd-comp.o vjcompress.o \
585                 zlib.o
```



```
587 TCP_OBJS +=     tcpddi.o

589 TCP6_OBJS +=    tcp6ddi.o

591 NCA_OBJS +=     ncaddi.o

593 SDP_SOCK_MOD_OBJS += sockmod_sdp.o socksdp.o socksdpsubr.o

595 SCTP_SOCK_MOD_OBJS += sockmod_sctp.o socksctp.o socksctpsubr.o

597 PFP_SOCK_MOD_OBJS += sockmod_pfp.o

599 RDS_SOCK_MOD_OBJS += sockmod_rds.o

601 RDS_OBJS +=     rdsddi.o rdssubr.o rds_opt.o rds_ioctl.o

603 RDSIB_OBJS +=   rdsib.o rdsib_ib.o rdsib_cm.o rdsib_ep.o rdsib_buf.o \
604                 rdsib_debug.o rdsib_sc.o

606 RDSV3_OBJS +=   af_rds.o rdsv3_ddi.o bind.o loop.o threads.o connection.o \
607                 transport.o cong.o sysctl.o message.o rds_recv.o send.o \
608                 stats.o info.o page.o rdma_transport.o ib_ring.o ib_rdma.o \
609                 ib_recv.o ib.o ib_send.o ib_sysctl.o ib_stats.o ib_cm.o \
610                 rdsv3_sc.o rdsv3_debug.o rdsv3_impl.o rdma.o rdsv3_af_thr.o

612 ISER_OBJS +=    iser.o iser_cm.o iser_cq.o iser_ib.o iser_idm.o \
613                 iser_resource.o iser_xfer.o

615 UDP_OBJS +=     udpddi.o

617 UDP6_OBJS +=    udp6ddi.o

619 SY_OBJS +=      gentty.o

621 TCO_OBJS +=     ticots.o

623 TCOO_OBJS +=    ticotsord.o

625 TCL_OBJS +=     ticlts.o

627 TL_OBJS +=      tl.o

629 DUMP_OBJS +=    dump.o

631 BPF_OBJS +=     bpf.o bpf_filter.o bpf_mod.o bpf_dlt.o bpf_mac.o

633 CLONE_OBJS +=   clone.o

635 CN_OBJS +=      cons.o

637 DLD_OBJS +=     dld_drv.o dld_proto.o dld_str.o dld_flow.o

639 DLS_OBJS +=     dls.o dls_link.o dls_mod.o dls_stat.o dls_mgmt.o

641 GLD_OBJS +=     gld.o gldutil.o

643 MAC_OBJS +=     mac.o  mac_bcast.o mac_client.o mac_datapath_setup.o mac_flow.o
644                 mac_hio.o mac_mod.o mac_ndd.o mac_provider.o mac_sched.o \
645                 mac_protect.o mac_soft_ring.o mac_stat.o mac_util.o

647 MAC_6TO4_OBJS +=        mac_6to4.o

649 MAC_ETHER_OBJS +=       mac_ether.o

651 MAC_IPV4_OBJS +=        mac_ipv4.o
```

```
653 MAC_IPV6_OBJS +=        mac_ipv6.o

655 MAC_WIFI_OBJS +=        mac_wifi.o

657 MAC_IB_OBJS +=          mac_ib.o

659 IPTUN_OBJS +=   iptun_dev.o iptun_ctl.o iptun.o

661 AGGR_OBJS +=    aggr_dev.o aggr_ctl.o aggr_grp.o aggr_port.o \
662                 aggr_send.o aggr_recv.o aggr_lacp.o

664 SOFTMAC_OBJS += softmac_main.o softmac_ctl.o softmac_capab.o \
665                 softmac_dev.o softmac_stat.o softmac_pkt.o softmac_fp.o

667 NET80211_OBJS += net80211.o net80211_proto.o net80211_input.o \
668                  net80211_output.o net80211_node.o net80211_crypto.o \
669                  net80211_crypto_none.o net80211_crypto_wep.o net80211_ioctl.o \
670                  net80211_crypto_tkip.o net80211_crypto_ccmp.o  \
671                  net80211_ht.o

673 VNIC_OBJS +=    vnic_ctl.o vnic_dev.o

675 SIMNET_OBJS +=  simnet.o

677 IB_OBJS +=      ibnex.o ibnex_ioctl.o ibnex_hca.o

679 IBCM_OBJS +=    ibcm_impl.o ibcm_sm.o ibcm_ti.o ibcm_utils.o ibcm_path.o \
680                 ibcm_arp.o ibcm_arp_link.o

682 IBDM_OBJS +=    ibdm.o

684 IBDMA_OBJS +=   ibdma.o

686 IBMF_OBJS +=    ibmf.o ibmf_impl.o ibmf_dr.o ibmf_wqe.o ibmf_ud_dest.o ibmf_mod.
687                 ibmf_send.o ibmf_recv.o ibmf_handlers.o ibmf_trans.o \
688                 ibmf_timers.o ibmf_msg.o ibmf_utils.o ibmf_rmpp.o \
689                 ibmf_saa.o ibmf_saa_impl.o ibmf_saa_utils.o ibmf_saa_events.o

691 IBTL_OBJS +=    ibtl_impl.o ibtl_util.o ibtl_mem.o ibtl_handlers.o ibtl_qp.o \
692                 ibtl_cq.o ibtl_wr.o ibtl_hca.o ibtl_chan.o ibtl_cm.o \
693                 ibtl_mcg.o ibtl_ibnex.o ibtl_srq.o ibtl_part.o

695 TAVOR_OBJS +=   tavor.o tavor_agents.o tavor_cfg.o tavor_ci.o tavor_cmd.o \
696                 tavor_cq.o tavor_event.o tavor_ioctl.o tavor_misc.o \
697                 tavor_mr.o tavor_qp.o tavor_qpmod.o tavor_rsrc.o \
698                 tavor_srq.o tavor_stats.o tavor_umap.o tavor_wr.o

700 HERMON_OBJS +=  hermon.o hermon_agents.o hermon_cfg.o hermon_ci.o hermon_cmd.o \
701                 hermon_cq.o hermon_event.o hermon_ioctl.o hermon_misc.o \
702                 hermon_mr.o hermon_qp.o hermon_qpmod.o hermon_rsrc.o \
703                 hermon_srq.o hermon_stats.o hermon_umap.o hermon_wr.o \
704                 hermon_fcoib.o hermon_fm.o

706 DAPLT_OBJS +=   daplt.o

708 SOL_OFS_OBJS += sol_cma.o sol_ib_cma.o sol_uobj.o \
709                 sol_ofs_debug_util.o sol_ofs_gen_util.o \
710                 sol_kverbs.o

712 SOL_UCMA_OBJS +=        sol_ucma.o

714 SOL_UVERBS_OBJS +=      sol_uverbs.o sol_uverbs_comp.o sol_uverbs_event.o \
715                         sol_uverbs_hca.o sol_uverbs_qp.o

717 SOL_UMAD_OBJS += sol_umad.o
```



```
719 KSTAT_OBJS +=   kstat.o

721 KSYMS_OBJS +=   ksyms.o

723 INSTANCE_OBJS += inst_sync.o

725 IWSCN_OBJS +=   iwscons.o

727 LOFI_OBJS +=    lofi.o LzmaDec.o

729 FSSNAP_OBJS +=  fssnap.o

731 FSSNAPIF_OBJS += fssnap_if.o

733 MM_OBJS +=      mem.o

735 PHYSMEM_OBJS += physmem.o

737 OPTIONS_OBJS += options.o

739 WINLOCK_OBJS += winlockio.o

741 PM_OBJS +=      pm.o
742 SRN_OBJS +=     srn.o

744 PSEUDO_OBJS +=  pseudonex.o

746 RAMDISK_OBJS += ramdisk.o

748 LLC1_OBJS += llc1.o

750 USBKBM_OBJS += usbkbm.o

752 USBWCM_OBJS += usbwcm.o

754 BOFI_OBJS += bofi.o

756 HID_OBJS += hid.o

758 HWA_RC_OBJS += hwarc.o

760 USBSKEL_OBJS += usbskel.o

762 USBVC_OBJS += usbvc.o usbvc_v4l2.o

764 HIDPARSER_OBJS += hidparser.o

766 USB_AC_OBJS += usb_ac.o

768 USB_AS_OBJS += usb_as.o

770 USB_AH_OBJS += usb_ah.o

772 USBMS_OBJS += usbms.o

774 USBPRN_OBJS += usbprn.o

776 UGEN_OBJS += ugen.o

778 USBSER_OBJS += usbser.o usbser_rseq.o

780 USBSACM_OBJS += usbsacm.o

782 USBSER_KEYSPAN_OBJS += usbser_keyspan.o keyspan_dsd.o keyspan_pipe.o
```

```
784 USBS49_FW_OBJS += keyspan_49fw.o

786 USBSPRL_OBJS += usbser_pl2303.o pl2303_dsd.o

788 WUSB_CA_OBJS += wusb_ca.o

790 USBFTDI_OBJS += usbser_uftdi.o uftdi_dsd.o

792 USBECM_OBJS += usbecm.o

794 WC_OBJS += wscons.o vcons.o

796 VCONS_CONF_OBJS += vcons_conf.o

798 SCSI_OBJS +=    scsi_capabilities.o scsi_confsubr.o scsi_control.o \
799                 scsi_data.o scsi_fm.o scsi_hba.o scsi_reset_notify.o \
800                 scsi_resource.o scsi_subr.o scsi_transport.o scsi_watch.o \
801                 smp_transport.o

803 SCSI_VHCI_OBJS +=               scsi_vhci.o mpapi_impl.o scsi_vhci_tpgs.o

805 SCSI_VHCI_F_SYM_OBJS +=         sym.o

807 SCSI_VHCI_F_TPGS_OBJS +=        tpgs.o

809 SCSI_VHCI_F_ASYM_SUN_OBJS +=    asym_sun.o

811 SCSI_VHCI_F_SYM_HDS_OBJS +=     sym_hds.o

813 SCSI_VHCI_F_TAPE_OBJS +=        tape.o

815 SCSI_VHCI_F_TPGS_TAPE_OBJS +=   tpgs_tape.o

817 SGEN_OBJS +=    sgen.o

819 SMP_OBJS +=     smp.o

821 SATA_OBJS +=    sata.o

823 USBA_OBJS +=    hcdi.o  usba.o  usbai.o hubdi.o parser.o genconsole.o \
824                 usbai_pipe_mgmt.o usbai_req.o usbai_util.o usbai_register.o \
825                 usba_devdb.o usba10_calls.o usba_ugen.o whcdi.o wa.o
826 USBA_WITHOUT_WUSB_OBJS +=       hcdi.o  usba.o  usbai.o hubdi.o parser.o gencons
827                 usbai_pipe_mgmt.o usbai_req.o usbai_util.o usbai_register.o \
828                 usba_devdb.o usba10_calls.o usba_ugen.o

830 USBA10_OBJS +=  usba10.o

832 RSM_OBJS +=     rsm.o   rsmka_pathmanager.o     rsmka_util.o

834 RSMOPS_OBJS +=  rsmops.o

836 S1394_OBJS +=   t1394.o t1394_errmsg.o s1394.o s1394_addr.o s1394_asynch.o \
837                 s1394_bus_reset.o s1394_cmp.o s1394_csr.o s1394_dev_disc.o \
838                 s1394_fa.o s1394_fcp.o \
839                 s1394_hotplug.o s1394_isoch.o s1394_misc.o h1394.o nx1394.o

841 HCI1394_OBJS += hci1394.o hci1394_async.o hci1394_attach.o hci1394_buf.o \
842                 hci1394_csr.o hci1394_detach.o hci1394_extern.o \
843                 hci1394_ioctl.o hci1394_isoch.o hci1394_isr.o \
844                 hci1394_ixl_comp.o hci1394_ixl_isr.o hci1394_ixl_misc.o \
845                 hci1394_ixl_update.o hci1394_misc.o hci1394_ohci.o \
846                 hci1394_q.o hci1394_s1394if.o hci1394_tlabel.o \
847                 hci1394_tlist.o hci1394_vendor.o

849 AV1394_OBJS +=  av1394.o av1394_as.o av1394_async.o av1394_cfgrom.o \
```



```
850                 av1394_cmp.o av1394_fcp.o av1394_isoch.o av1394_isoch_chan.o \
851                 av1394_isoch_recv.o av1394_isoch_xmit.o av1394_list.o \
852                 av1394_queue.o

854 DCAM1394_OBJS += dcam.o dcam_frame.o dcam_param.o dcam_reg.o \
855                 dcam_ring_buff.o

857 SCSA1394_OBJS += hba.o sbp2_driver.o sbp2_bus.o

859 SBP2_OBJS += cfgrom.o sbp2.o

861 PMODEM_OBJS += pmodem.o pmodem_cis.o cis.o cis_callout.o cis_handlers.o cis_para

863 DSW_OBJS +=     dsw.o dsw_dev.o ii_tree.o

865 NCALL_OBJS +=   ncall.o \
866                 ncall_stub.o

868 RDC_OBJS +=     rdc.o \
869                 rdc_dev.o \
870                 rdc_io.o \
871                 rdc_clnt.o \
872                 rdc_prot_xdr.o \
873                 rdc_svc.o \
874                 rdc_bitmap.o \
875                 rdc_health.o \
876                 rdc_subr.o \
877                 rdc_diskq.o

879 RDCSRV_OBJS +=  rdcsrv.o

881 RDCSTUB_OBJS += rdc_stub.o

883 SDBC_OBJS +=    sd_bcache.o \
884                 sd_bio.o \
885                 sd_conf.o \
886                 sd_ft.o \
887                 sd_hash.o \
888                 sd_io.o \
889                 sd_misc.o \
890                 sd_pcu.o \
891                 sd_tdaemon.o \
892                 sd_trace.o \
893                 sd_iob_impl0.o \
894                 sd_iob_impl1.o \
895                 sd_iob_impl2.o \
896                 sd_iob_impl3.o \
897                 sd_iob_impl4.o \
898                 sd_iob_impl5.o \
899                 sd_iob_impl6.o \
900                 sd_iob_impl7.o \
901                 safestore.o \
902                 safestore_ram.o

904 NSCTL_OBJS +=   nsctl.o \
905                 nsc_cache.o \
906                 nsc_disk.o \
907                 nsc_dev.o \
908                 nsc_freeze.o \
909                 nsc_gen.o \
910                 nsc_mem.o \
911                 nsc_ncallio.o \
912                 nsc_power.o \
913                 nsc_resv.o \
914                 nsc_rmspin.o \
915                 nsc_solaris.o \
```

```
916                 nsc_trap.o \
917                 nsc_list.o
918 UNISTAT_OBJS += spuni.o \
919                 spcs_s_k.o

921 NSKERN_OBJS +=  nsc_ddi.o \
922                 nsc_proc.o \
923                 nsc_raw.o \
924                 nsc_thread.o \
925                 nskernd.o

927 SV_OBJS +=      sv.o

929 PMCS_OBJS += pmcs_attach.o pmcs_ds.o pmcs_intr.o pmcs_nvram.o pmcs_sata.o \
930                 pmcs_scsa.o pmcs_smhba.o pmcs_subr.o pmcs_fwlog.o

932 PMCS8001FW_C_OBJS +=    pmcs_fw_hdr.o
933 PMCS8001FW_OBJS +=              $(PMCS8001FW_C_OBJS) SPCBoot.o ila.o firmware.o

935 #
936 #       Build up defines and paths.

938 ST_OBJS +=      st.o    st_conf.o

940 EMLXS_OBJS +=   emlxs_clock.o emlxs_dfc.o emlxs_dhchap.o emlxs_diag.o \
941                 emlxs_download.o emlxs_dump.o emlxs_els.o emlxs_event.o \
942                 emlxs_fcf.o emlxs_fcp.o emlxs_fct.o emlxs_hba.o emlxs_ip.o \
943                 emlxs_mbox.o emlxs_mem.o emlxs_msg.o emlxs_node.o \
944                 emlxs_pkt.o emlxs_sli3.o emlxs_sli4.o emlxs_solaris.o \
945                 emlxs_thread.o

947 EMLXS_FW_OBJS +=        emlxs_fw.o

949 OCE_OBJS +=     oce_buf.o oce_fm.o oce_gld.o oce_hw.o oce_intr.o oce_main.o \
950                 oce_mbx.o oce_mq.o oce_queue.o oce_rx.o oce_stat.o oce_tx.o \
951                 oce_utils.o

953 FCT_OBJS += discovery.o fct.o

955 QLT_OBJS += 2400.o 2500.o 8100.o qlt.o qlt_dma.o

957 SRPT_OBJS += srpt_mod.o srpt_ch.o srpt_cm.o srpt_ioc.o srpt_stp.o

959 FCOE_OBJS += fcoe.o fcoe_eth.o fcoe_fc.o

961 FCOET_OBJS += fcoet.o fcoet_eth.o fcoet_fc.o

963 FCOEI_OBJS += fcoei.o fcoei_eth.o fcoei_lv.o

965 ISCSIT_SHARED_OBJS += \
966                 iscsit_common.o

968 ISCSIT_OBJS +=  $(ISCSIT_SHARED_OBJS) \
969                 iscsit.o iscsit_tgt.o iscsit_sess.o iscsit_login.o \
970                 iscsit_text.o iscsit_isns.o iscsit_radiusauth.o \
971                 iscsit_radiuspacket.o iscsit_auth.o iscsit_authclient.o

973 PPPT_OBJS +=    alua_ic_if.o pppt.o pppt_msg.o pppt_tgt.o

975 STMF_OBJS += lun_map.o stmf.o

977 STMF_SBD_OBJS += sbd.o sbd_scsi.o sbd_pgr.o sbd_zvol.o

979 SYSMSG_OBJS +=  sysmsg.o

981 SES_OBJS +=     ses.o ses_sen.o ses_safte.o ses_ses.o
```



```
983 TNF_OBJS +=     tnf_buf.o       tnf_trace.o     tnf_writer.o    trace_init.o \
984                 trace_funcs.o   tnf_probe.o     tnf.o

986 LOGINDMUX_OBJS += logindmux.o

988 DEVINFO_OBJS += devinfo.o

990 DEVPOLL_OBJS += devpoll.o

992 DEVPOOL_OBJS += devpool.o

994 I8042_OBJS +=   i8042.o

996 KB8042_OBJS +=  \
997                 at_keyprocess.o \
998                 kb8042.o        \
999                 kb8042_keytables.o

1001 MOUSE8042_OBJS += mouse8042.o

1003 FDC_OBJS +=     fdc.o

1005 ASY_OBJS +=     asy.o

1007 ECPP_OBJS +=    ecpp.o

1009 VUIDM3P_OBJS += vuidmice.o vuidm3p.o

1011 VUIDM4P_OBJS += vuidmice.o vuidm4p.o

1013 VUIDM5P_OBJS += vuidmice.o vuidm5p.o

1015 VUIDPS2_OBJS += vuidmice.o vuidps2.o

1017 HPCSVC_OBJS += hpcsvc.o

1019 PCIE_MISC_OBJS += pcie.o pcie_fault.o pcie_hp.o pciehpc.o pcishpc.o pcie_pwr.o p

1021 PCIHPNEXUS_OBJS += pcihp.o

1023 OPENEEPR_OBJS += openprom.o

1025 RANDOM_OBJS += random.o

1027 PSHOT_OBJS += pshot.o

1029 GEN_DRV_OBJS += gen_drv.o

1031 TCLIENT_OBJS += tclient.o

1033 TPHCI_OBJS += tphci.o

1035 TVHCI_OBJS += tvhci.o

1037 EMUL64_OBJS += emul64.o emul64_bsd.o

1039 FCP_OBJS += fcp.o

1041 FCIP_OBJS += fcip.o

1043 FCSM_OBJS += fcsm.o

1045 FCTL_OBJS += fctl.o

1047 FP_OBJS += fp.o
```

```
1049 QLC_OBJS += ql_api.o ql_debug.o ql_hba_fru.o ql_init.o ql_iocb.o ql_ioctl.o \
1050         ql_isr.o ql_mbx.o ql_nx.o ql_xioctl.o ql_fw_table.o

1052 QLC_FW_2200_OBJS += ql_fw_2200.o

1054 QLC_FW_2300_OBJS += ql_fw_2300.o

1056 QLC_FW_2400_OBJS += ql_fw_2400.o

1058 QLC_FW_2500_OBJS += ql_fw_2500.o

1060 QLC_FW_6322_OBJS += ql_fw_6322.o

1062 QLC_FW_8100_OBJS += ql_fw_8100.o

1064 QLGE_OBJS += qlge.o qlge_dbg.o qlge_flash.o qlge_fm.o qlge_gld.o qlge_mpi.o

1066 ZCONS_OBJS += zcons.o

1068 NV_SATA_OBJS += nv_sata.o

1070 SI3124_OBJS += si3124.o

1072 AHCI_OBJS += ahci.o

1074 PCIIDE_OBJS += pci-ide.o

1076 PCEPP_OBJS += pcepp.o

1078 CPC_OBJS += cpc.o

1080 CPUID_OBJS += cpuid_drv.o

1082 SYSEVENT_OBJS += sysevent.o

1084 BL_OBJS += bl.o

1086 DRM_OBJS += drm_sunmod.o drm_kstat.o drm_agpsupport.o \
1087             drm_auth.o drm_bufs.o drm_context.o drm_dma.o \
1088             drm_drawable.o drm_drv.o drm_fops.o drm_ioctl.o drm_irq.o \
1089             drm_lock.o drm_memory.o drm_msg.o drm_pci.o drm_scatter.o \
1090             drm_cache.o drm_gem.o drm_mm.o ati_pcigart.o

1092 FM_OBJS += devfm.o devfm_machdep.o

1094 RTLS_OBJS +=    rtls.o

1096 #
1097 #                       exec modules
1098 #
1099 AOUTEXEC_OBJS +=aout.o

1101 ELFEXEC_OBJS += elf.o elf_notes.o old_notes.o

1103 INTPEXEC_OBJS +=intp.o

1105 SHBINEXEC_OBJS +=shbin.o

1107 JAVAEXEC_OBJS +=java.o

1109 #
1110 #                       file system modules
1111 #
1112 AUTOFS_OBJS +=  auto_vfsops.o auto_vnops.o auto_subr.o auto_xdr.o auto_sys.o
```



```
1114 CACHEFS_OBJS += cachefs_cnode.o         cachefs_cod.o \
1115                 cachefs_dir.o           cachefs_dlog.o  cachefs_filegrp.o \
1116                 cachefs_fscache.o       cachefs_ioctl.o cachefs_log.o \
1117                 cachefs_module.o \
1118                 cachefs_noopc.o         cachefs_resource.o \
1119                 cachefs_strict.o \
1120                 cachefs_subr.o          cachefs_vfsops.o \
1121                 cachefs_vnops.o

1123 DCFS_OBJS +=    dc_vnops.o

1125 DEVFS_OBJS +=   devfs_subr.o    devfs_vfsops.o  devfs_vnops.o

1127 DEV_OBJS  +=    sdev_subr.o     sdev_vfsops.o   sdev_vnops.o    \
1128                 sdev_ptsops.o   sdev_zvolops.o  sdev_comm.o     \
1129                 sdev_profile.o  sdev_ncache.o   sdev_netops.o   \
1130                 sdev_ipnetops.o \
1131                 sdev_vtops.o

1133 CTFS_OBJS +=    ctfs_all.o ctfs_cdir.o ctfs_ctl.o ctfs_event.o \
1134                 ctfs_latest.o ctfs_root.o ctfs_sym.o ctfs_tdir.o ctfs_tmpl.o

1136 OBJFS_OBJS +=   objfs_vfs.o     objfs_root.o    objfs_common.o \
1137                 objfs_odir.o    objfs_data.o

1139 FDFS_OBJS +=    fdops.o

1141 FIFO_OBJS +=    fifosubr.o      fifovnops.o

1143 PIPE_OBJS +=    pipe.o

1145 HSFS_OBJS +=    hsfs_node.o     hsfs_subr.o     hsfs_vfsops.o   hsfs_vnops.o \
1146                 hsfs_susp.o     hsfs_rrip.o     hsfs_susp_subr.o

1148 LOFS_OBJS +=    lofs_subr.o     lofs_vfsops.o   lofs_vnops.o

1150 NAMEFS_OBJS +=  namevfs.o       namevno.o

1152 NFS_OBJS +=     nfs_client.o    nfs_common.o    nfs_dump.o \
1153                 nfs_subr.o      nfs_vfsops.o    nfs_vnops.o \
1154                 nfs_xdr.o       nfs_sys.o       nfs_strerror.o \
1155                 nfs3_vfsops.o   nfs3_vnops.o    nfs3_xdr.o \
1156                 nfs_acl_vnops.o nfs_acl_xdr.o   nfs4_vfsops.o \
1157                 nfs4_vnops.o    nfs4_xdr.o      nfs4_idmap.o \
1158                 nfs4_shadow.o   nfs4_subr.o \
1159                 nfs4_attr.o     nfs4_rnode.o    nfs4_client.o \
1160                 nfs4_acache.o   nfs4_common.o   nfs4_client_state.o \
1161                 nfs4_callback.o nfs4_recovery.o nfs4_client_secinfo.o \
1162                 nfs4_client_debug.o     nfs_stats.o \
1163                 nfs4_acl.o      nfs4_stub_vnops.o       nfs_cmd.o

1165 NFSSRV_OBJS +=  nfs_server.o    nfs_srv.o       nfs3_srv.o \
1166                 nfs_acl_srv.o   nfs_auth.o      nfs_auth_xdr.o \
1167                 nfs_export.o    nfs_log.o       nfs_log_xdr.o \
1168                 nfs4_srv.o      nfs4_state.o    nfs4_srv_attr.o \
1169                 nfs4_srv_ns.o   nfs4_db.o       nfs4_srv_deleg.o \
1170                 nfs4_deleg_ops.o nfs4_srv_readdir.o nfs4_dispatch.o

1172 SMBSRV_SHARED_OBJS += \
1173                 smb_inet.o \
1174                 smb_match.o \
1175                 smb_msgbuf.o \
1176                 smb_oem.o \
1177                 smb_string.o \
1178                 smb_utf8.o \
1179                 smb_door_legacy.o \
```

```
1180                    smb_xdr.o \
1181                    smb_token.o \
1182                    smb_token_xdr.o \
1183                    smb_sid.o \
1184                    smb_native.o \
1185                    smb_netbios_util.o

1187 SMBSRV_OBJS +=    $(SMBSRV_SHARED_OBJS)                   \
1188                    smb_acl.o                               \
1189                    smb_alloc.o                             \
1190                    smb_close.o                             \
1191                    smb_common_open.o                       \
1192                    smb_common_transact.o                   \
1193                    smb_create.o                            \
1194                    smb_delete.o                            \
1195                    smb_directory.o                         \
1196                    smb_dispatch.o                          \
1197                    smb_echo.o                              \
1198                    smb_fem.o                               \
1199                    smb_find.o                              \
1200                    smb_flush.o                             \
1201                    smb_fsinfo.o                            \
1202                    smb_fsops.o                             \
1203                    smb_init.o                              \
1204                    smb_kdoor.o                             \
1205                    smb_kshare.o                            \
1206                    smb_kutil.o                             \
1207                    smb_lock.o                              \
1208                    smb_lock_byte_range.o                   \
1209                    smb_locking_andx.o                      \
1210                    smb_logoff_andx.o                       \
1211                    smb_mangle_name.o                       \
1212                    smb_mbuf_marshaling.o                   \
1213                    smb_mbuf_util.o                         \
1214                    smb_negotiate.o                         \
1215                    smb_net.o                               \
1216                    smb_node.o                              \
1217                    smb_nt_cancel.o                         \
1218                    smb_nt_create_andx.o                    \
1219                    smb_nt_transact_create.o                \
1220                    smb_nt_transact_ioctl.o                 \
1221                    smb_nt_transact_notify_change.o         \
1222                    smb_nt_transact_quota.o                 \
1223                    smb_nt_transact_security.o              \
1224                    smb_odir.o                              \
1225                    smb_ofile.o                             \
1226                    smb_open_andx.o                         \
1227                    smb_opipe.o                             \
1228                    smb_oplock.o                            \
1229                    smb_pathname.o                          \
1230                    smb_print.o                             \
1231                    smb_process_exit.o                      \
1232                    smb_query_fileinfo.o                    \
1233                    smb_read.o                              \
1234                    smb_rename.o                            \
1235                    smb_sd.o                                \
1236                    smb_seek.o                              \
1237                    smb_server.o                            \
1238                    smb_session.o                           \
1239                    smb_session_setup_andx.o                \
1240                    smb_set_fileinfo.o                      \
1241                    smb_signing.o                           \
1242                    smb_tree.o                              \
1243                    smb_trans2_create_directory.o           \
1244                    smb_trans2_dfs.o                        \
1245                    smb_trans2_find.o                       \
```



```
1246                    smb_tree_connect.o                      \
1247                    smb_unlock_byte_range.o                 \
1248                    smb_user.o                              \
1249                    smb_vfs.o                               \
1250                    smb_vops.o                              \
1251                    smb_vss.o                               \
1252                    smb_write.o                             \
1253                    smb_write_raw.o

1255 PCFS_OBJS +=      pc_alloc.o      pc_dir.o        pc_node.o       pc_subr.o \
1256                    pc_vfsops.o     pc_vnops.o

1258 PROC_OBJS +=      prcontrol.o     prioctl.o       prsubr.o        prusrio.o \
1259                    prvfsops.o      prvnops.o

1261 MNTFS_OBJS +=     mntvfsops.o     mntvnops.o

1263 SHAREFS_OBJS +=   sharetab.o      sharefs_vfsops.o        sharefs_vnops.o

1265 SPEC_OBJS +=      specsubr.o      specvfsops.o    specvnops.o

1267 SOCK_OBJS +=      socksubr.o      sockvfsops.o    sockparams.o    \
1268                    socksyscalls.o  socktpi.o       sockstr.o \
1269                    sockcommon_vnops.o      sockcommon_subr.o \
1270                    sockcommon_sops.o       sockcommon.o    \
1271                    sock_notsupp.o  socknotify.o \
1272                    nl7c.o          nl7curi.o       nl7chttp.o      nl7clogd.o \
1273                    nl7cnca.o       sodirect.o      sockfilter.o

1275 TMPFS_OBJS +=     tmp_dir.o       tmp_subr.o      tmp_tnode.o     tmp_vfsops.o \
1276                    tmp_vnops.o

1278 UDFS_OBJS +=      udf_alloc.o     udf_bmap.o      udf_dir.o       \
1279                    udf_inode.o     udf_subr.o      udf_vfsops.o    \
1280                    udf_vnops.o

1282 UFS_OBJS +=       ufs_alloc.o     ufs_bmap.o      ufs_dir.o       ufs_xattr.o \
1283                    ufs_inode.o     ufs_subr.o      ufs_tables.o    ufs_vfsops.o \
1284                    ufs_vnops.o     quota.o         quotacalls.o    quota_ufs.o \
1285                    ufs_filio.o     ufs_lockfs.o    ufs_thread.o    ufs_trans.o \
1286                    ufs_acl.o       ufs_panic.o     ufs_directio.o  ufs_log.o \
1287                    ufs_extvnops.o  ufs_snap.o      lufs.o          lufs_thread.o \
1288                    lufs_log.o      lufs_map.o      lufs_top.o      lufs_debug.o
1289 VSCAN_OBJS +=     vscan_drv.o     vscan_svc.o vscan_door.o

1291 NSMB_OBJS +=      smb_conn.o      smb_dev.o       smb_iod.o       smb_pass.o \
1292                    smb_rq.o        smb_sign.o      smb_smb.o       smb_subrs.o \
1293                    smb_time.o      smb_tran.o      smb_trantcp.o   smb_usr.o \
1294                    subr_mchain.o

1296 SMBFS_COMMON_OBJS += smbfs_ntacl.o
1297 SMBFS_OBJS +=      smbfs_vfsops.o  smbfs_vnops.o   smbfs_node.o    \
1298                    smbfs_acl.o     smbfs_client.o  smbfs_smb.o     \
1299                    smbfs_subr.o    smbfs_subr2.o   \
1300                    smbfs_rwlock.o  smbfs_xattr.o   \
1301                    $(SMBFS_COMMON_OBJS)


1304 #
1305 #                       LVM modules
1306 #
1307 MD_OBJS += md.o md_error.o md_ioctl.o md_mddb.o md_names.o \
1308         md_med.o md_rename.o md_subr.o

1310 MD_COMMON_OBJS = md_convert.o md_crc.o md_revchk.o
```

```
1312 MD_DERIVED_OBJS = metamed_xdr.o meta_basic_xdr.o

1314 SOFTPART_OBJS += sp.o sp_ioctl.o

1316 STRIPE_OBJS += stripe.o stripe_ioctl.o

1318 HOTSPARES_OBJS += hotspares.o

1320 RAID_OBJS += raid.o raid_ioctl.o raid_replay.o raid_resync.o raid_hotspare.o

1322 MIRROR_OBJS += mirror.o mirror_ioctl.o mirror_resync.o

1324 NOTIFY_OBJS += md_notify.o

1326 TRANS_OBJS += mdtrans.o trans_ioctl.o trans_log.o

1328 ZFS_COMMON_OBJS +=             \
1329         arc.o                  \
1330         bplist.o               \
1331         bpobj.o                \
1332         bptree.o               \
1333         dbuf.o                 \
1334         ddt.o                  \
1335         ddt_zap.o              \
1336         dmu.o                  \
1337         dmu_diff.o             \
1338         dmu_send.o             \
1339         dmu_object.o           \
1340         dmu_objset.o           \
1341         dmu_traverse.o         \
1342         dmu_tx.o               \
1343         dnode.o                \
1344         dnode_sync.o           \
1345         dsl_dir.o              \
1346         dsl_dataset.o          \
1347         dsl_deadlist.o         \
1348         dsl_pool.o             \
1349         dsl_synctask.o         \
1350         dmu_zfetch.o           \
1351         dsl_deleg.o            \
1352         dsl_prop.o             \
1353         dsl_scan.o             \
1354         zfeature.o             \
1355         gzip.o                 \
1356         lzjb.o                 \
1357         metaslab.o             \
1358         refcount.o             \
1359         sa.o                   \
1360         sha256.o               \
1361         spa.o                  \
1362         spa_config.o           \
1363         spa_errlog.o           \
1364         spa_history.o          \
1365         spa_misc.o             \
1366         space_map.o            \
1367         txg.o                  \
1368         uberblock.o            \
1369         unique.o               \
1370         vdev.o                 \
1371         vdev_cache.o           \
1372         vdev_file.o            \
1373         vdev_label.o           \
1374         vdev_mirror.o          \
1375         vdev_missing.o         \
1376         vdev_queue.o           \
1377         vdev_raidz.o           \
```



```
1378         vdev_root.o            \
1379         zap.o                  \
1380         zap_leaf.o             \
1381         zap_micro.o            \
1382         zfs_byteswap.o         \
1383         zfs_debug.o            \
1384         zfs_fm.o               \
1385         zfs_fuid.o             \
1386         zfs_sa.o               \
1387         zfs_znode.o            \
1388         zil.o                  \
1389         zio.o                  \
1390         zio_checksum.o         \
1391         zio_compress.o         \
1392         zio_inject.o           \
1393         zle.o                  \
1394         zrlock.o

1396 ZFS_SHARED_OBJS +=             \
1397         zfeature_common.o      \
1398         zfs_comutil.o          \
1399         zfs_deleg.o            \
1400         zfs_fletcher.o         \
1401         zfs_namecheck.o        \
1402         zfs_prop.o             \
1403         zpool_prop.o           \
1404         zprop_common.o

1406 ZFS_OBJS +=                    \
1407         $(ZFS_COMMON_OBJS)     \
1408         $(ZFS_SHARED_OBJS)     \
1409         vdev_disk.o            \
1410         zfs_acl.o              \
1411         zfs_ctldir.o           \
1412         zfs_dir.o              \
1413         zfs_ioctl.o            \
1414         zfs_log.o              \
1415         zfs_onexit.o           \
1416         zfs_replay.o           \
1417         zfs_rlock.o            \
1418         rrwlock.o              \
1419         zfs_vfsops.o           \
1420         zfs_vnops.o            \
1421         far.o                  \
1422         far_pass1.o            \
1423         far_pass2.o            \
1424         far_send.o             \
1425         far_crc32c.o           \
1426         far_count.o            \
1427 #endif /* ! codereview */
1428         zvol.o

1430 ZUT_OBJS +=                    \
1431         zut.o

1433 #
1434 #                       streams modules
1435 #
1436 BUFMOD_OBJS     +=      bufmod.o

1438 CONNLD_OBJS +=  connld.o

1440 DEDUMP_OBJS +=  dedump.o

1442 DRCOMPAT_OBJS +=        drcompat.o
```

```
1444 LDLINUX_OBJS += ldlinux.o

1446 LDTERM_OBJS +=  ldterm.o uwidth.o

1448 PCKT_OBJS +=    pckt.o

1450 PFMOD_OBJS +=   pfmod.o

1452 PTEM_OBJS +=    ptem.o

1454 REDIRMOD_OBJS += strredirm.o

1456 TIMOD_OBJS +=   timod.o

1458 TIRDWR_OBJS +=  tirdwr.o

1460 TTCOMPAT_OBJS +=ttcompat.o

1462 LOG_OBJS +=     log.o

1464 PIPEMOD_OBJS += pipemod.o

1466 RPCMOD_OBJS +=  rpcmod.o        clnt_cots.o     clnt_clts.o \
1467                 clnt_gen.o      clnt_perr.o     mt_rpcinit.o    rpc_calmsg.o \
1468                 rpc_prot.o      rpc_sztypes.o   rpc_subr.o      rpcb_prot.o \
1469                 svc.o           svc_clts.o      svc_gen.o       svc_cots.o \
1470                 rpcsys.o        xdr_sizeof.o    clnt_rdma.o     svc_rdma.o \
1471                 xdr_rdma.o      rdma_subr.o     xdrrdma_sizeof.o

1473 TLIMOD_OBJS +=  tlimod.o        t_kalloc.o      t_kbind.o       t_kclose.o \
1474                 t_kconnect.o    t_kfree.o       t_kgtstate.o    t_kopen.o \
1475                 t_krcvudat.o    t_ksndudat.o    t_kspoll.o      t_kunbind.o \
1476                 t_kutil.o

1478 RLMOD_OBJS += rlmod.o

1480 TELMOD_OBJS += telmod.o

1482 CRYPTMOD_OBJS += cryptmod.o

1484 KB_OBJS +=      kbd.o           keytables.o

1486 #
1487 #                       ID mapping module
1488 #
1489 IDMAP_OBJS +=   idmap_mod.o     idmap_kapi.o    idmap_xdr.o     idmap_cache.o

1491 #
1492 #                       scheduling class modules
1493 #
1494 SDC_OBJS +=             sysdc.o

1496 RT_OBJS +=              rt.o
1497 RT_DPTBL_OBJS +=        rt_dptbl.o

1499 TS_OBJS +=              ts.o
1500 TS_DPTBL_OBJS +=        ts_dptbl.o

1502 IA_OBJS +=              ia.o

1504 FSS_OBJS +=             fss.o

1506 FX_OBJS +=              fx.o
1507 FX_DPTBL_OBJS +=        fx_dptbl.o

1509 #
```



```
1510 #                       Inter-Process Communication (IPC) modules
1511 #
1512 IPC_OBJS +=     ipc.o

1514 IPCMSG_OBJS +=  msg.o

1516 IPCSEM_OBJS +=  sem.o

1518 IPCSHM_OBJS +=  shm.o

1520 #
1521 #                       bignum module
1522 #
1523 COMMON_BIGNUM_OBJS += bignum_mod.o bignumimpl.o

1525 BIGNUM_OBJS += $(COMMON_BIGNUM_OBJS) $(BIGNUM_PSR_OBJS)

1527 #
1528 #                       kernel cryptographic framework
1529 #
1530 KCF_OBJS +=     kcf.o kcf_callprov.o kcf_cbufcall.o kcf_cipher.o kcf_crypto.o \
1531                 kcf_cryptoadm.o kcf_ctxops.o kcf_digest.o kcf_dual.o \
1532                 kcf_keys.o kcf_mac.o kcf_mech_tabs.o kcf_miscapi.o \
1533                 kcf_object.o kcf_policy.o kcf_prov_lib.o kcf_prov_tabs.o \
1534                 kcf_sched.o kcf_session.o kcf_sign.o kcf_spi.o kcf_verify.o \
1535                 kcf_random.o modes.o ecb.o cbc.o ctr.o ccm.o gcm.o \
1536                 fips_random.o

1538 CRYPTOADM_OBJS += cryptoadm.o

1540 CRYPTO_OBJS +=  crypto.o

1542 DPROV_OBJS +=   dprov.o

1544 DCA_OBJS +=     dca.o dca_3des.o dca_debug.o dca_dsa.o dca_kstat.o dca_rng.o \
1545                 dca_rsa.o

1547 AESPROV_OBJS += aes.o aes_impl.o aes_modes.o

1549 ARCFOURPROV_OBJS += arcfour.o arcfour_crypt.o

1551 BLOWFISHPROV_OBJS += blowfish.o blowfish_impl.o

1553 ECCPROV_OBJS += ecc.o ec.o ec2_163.o ec2_mont.o ecdecode.o ecl_mult.o \
1554                 ecp_384.o ecp_jac.o ec2_193.o ecl.o ecp_192.o ecp_521.o \
1555                 ecp_jm.o ec2_233.o ecl_curve.o ecp_224.o ecp_aff.o \
1556                 ecp_mont.o ec2_aff.o ec_naf.o ecl_gf.o ecp_256.o mp_gf2m.o \
1557                 mpi.o mplogic.o mpmontg.o mpprime.o oid.o \
1558                 secitem.o ec2_test.o ecp_test.o

1560 RSAPROV_OBJS += rsa.o rsa_impl.o pkcs1.o

1562 SWRANDPROV_OBJS += swrand.o

1564 #
1565 #                       kernel SSL
1566 #
1567 KSSL_OBJS +=    kssl.o kssloctl.o

1569 KSSL_SOCKFIL_MOD_OBJS += ksslfilter.o ksslapi.o ksslrec.o

1571 #
1572 #                       misc. modules
1573 #

1575 C2AUDIT_OBJS += adr.o audit.o audit_event.o audit_io.o \
```

```
1576                audit_path.o audit_start.o audit_syscalls.o audit_token.o \
1577                audit_mem.o

1579 PCIC_OBJS +=    pcic.o

1581 RPCSEC_OBJS +=  secmod.o        sec_clnt.o      sec_svc.o       sec_gen.o \
1582                auth_des.o      auth_kern.o     auth_none.o     auth_loopb.o\
1583                authdesprt.o    authdesubr.o    authu_prot.o \
1584                key_call.o      key_prot.o      svc_authu.o     svcauthdes.o

1586 RPCSEC_GSS_OBJS +=      rpcsec_gssmod.o rpcsec_gss.o rpcsec_gss_misc.o \
1587                rpcsec_gss_utils.o svc_rpcsec_gss.o

1589 CONSCONFIG_OBJS += consconfig.o

1591 CONSCONFIG_DACF_OBJS  += consconfig_dacf.o consplat.o

1593 TEM_OBJS += tem.o tem_safe.o 6x10.o 7x14.o 12x22.o

1595 KBTRANS_OBJS +=                        \
1596                kbtrans.o               \
1597                kbtrans_keytables.o     \
1598                kbtrans_polled.o        \
1599                kbtrans_streams.o       \
1600                usb_keytables.o

1602 KGSSD_OBJS +=   gssd_clnt_stubs.o gssd_handle.o gssd_prot.o \
1603                gss_display_name.o gss_release_name.o gss_import_name.o \
1604                gss_release_buffer.o gss_release_oid_set.o gen_oids.o gssdmod.o

1606 KGSSD_DERIVED_OBJS = gssd_xdr.o

1608 KGSS_DUMMY_OBJS += dmech.o

1610 KSOCKET_OBJS += ksocket.o ksocket_mod.o

1612 CRYPTO= cksumtypes.o decrypt.o encrypt.o encrypt_length.o etypes.o \
1613         nfold.o verify_checksum.o prng.o block_size.o make_checksum.o\
1614         checksum_length.o hmac.o default_state.o mandatory_sumtype.o

1616 # crypto/des
1617 CRYPTO_DES= f_cbc.o f_cksum.o f_parity.o weak_key.o d3_cbc.o ef_crypto.o

1619 CRYPTO_DK= checksum.o derive.o dk_decrypt.o dk_encrypt.o

1621 CRYPTO_ARCFOUR= k5_arcfour.o

1623 # crypto/enc_provider
1624 CRYPTO_ENC= des.o des3.o arcfour_provider.o aes_provider.o

1626 # crypto/hash_provider
1627 CRYPTO_HASH= hash_kef_generic.o hash_kmd5.o hash_crc32.o hash_ksha1.o

1629 # crypto/keyhash_provider
1630 CRYPTO_KEYHASH= descbc.o k5_kmd5des.o k_hmac_md5.o

1632 # crypto/crc32
1633 CRYPTO_CRC32= crc32.o

1635 # crypto/old
1636 CRYPTO_OLD= old_decrypt.o old_encrypt.o

1638 # crypto/raw
1639 CRYPTO_RAW= raw_decrypt.o raw_encrypt.o

1641 K5_KRB= kfree.o copy_key.o \
```



```
1642         parse.o init_ctx.o \
1643         ser_adata.o ser_addr.o \
1644         ser_auth.o ser_cksum.o \
1645         ser_key.o ser_princ.o \
1646         serialize.o unparse.o \
1647         ser_actx.o

1649 K5_OS=  timeofday.o toffset.o \
1650         init_os_ctx.o c_ustime.o

1652 SEAL=
1653 # EXPORT DELETE START
1654 SEAL=   seal.o unseal.o
1655 # EXPORT DELETE END

1657 MECH=   delete_sec_context.o \
1658         import_sec_context.o \
1659         gssapi_krb5.o \
1660         k5seal.o k5unseal.o k5sealv3.o \
1661         ser_sctx.o \
1662         sign.o \
1663         util_crypt.o  \
1664         util_validate.o  util_ordering.o  \
1665         util_seqnum.o util_set.o util_seed.o \
1666         wrap_size_limit.o verify.o



1670 MECH_GEN= util_token.o


1673 KGSS_KRB5_OBJS += krb5mech.o \
1674         $(MECH) $(SEAL) $(MECH_GEN) \
1675         $(CRYPTO) $(CRYPTO_DES) $(CRYPTO_DK) $(CRYPTO_ARCFOUR) \
1676         $(CRYPTO_ENC) $(CRYPTO_HASH) \
1677         $(CRYPTO_KEYHASH) $(CRYPTO_CRC32) \
1678         $(CRYPTO_OLD) \
1679         $(CRYPTO_RAW) $(K5_KRB) $(K5_OS)

1681 DES_OBJS +=     des_crypt.o des_impl.o des_ks.o des_soft.o

1683 DLBOOT_OBJS +=  bootparam_xdr.o nfs_dlinet.o scan.o

1685 KRTLD_OBJS +=   kobj_bootflags.o getoptstr.o \
1686                 kobj.o kobj_kdi.o kobj_lm.o kobj_subr.o

1688 MOD_OBJS +=     modctl.o modsubr.o modsysfile.o modconf.o modhash.o

1690 STRPLUMB_OBJS += strplumb.o

1692 CPR_OBJS +=     cpr_driver.o cpr_dump.o \
1693                 cpr_main.o cpr_misc.o cpr_mod.o cpr_stat.o \
1694                 cpr_uthread.o

1696 PROF_OBJS +=    prf.o

1698 SE_OBJS += se_driver.o

1700 SYSACCT_OBJS += acct.o

1702 ACCTCTL_OBJS += acctctl.o

1704 EXACCTSYS_OBJS += exacctsys.o

1706 KAIO_OBJS += aio.o
```

```
1708 PCMCIA_OBJS += pcmcia.o cs.o cis.o cis_callout.o cis_handlers.o cis_params.o

1710 BUSRA_OBJS += busra.o

1712 PCS_OBJS += pcs.o

1714 PCAN_OBJS += pcan.o

1716 PCATA_OBJS += pcide.o pcdisk.o pclabel.o pcata.o

1718 PCSER_OBJS += pcser.o pcser_cis.o

1720 PCWL_OBJS += pcwl.o

1722 PSET_OBJS +=    pset.o

1724 OHCI_OBJS += ohci.o ohci_hub.o ohci_polled.o

1726 UHCI_OBJS += uhci.o uhciutil.o uhcitgt.o uhcihub.o uhcipolled.o

1728 EHCI_OBJS += ehci.o ehci_hub.o ehci_xfer.o ehci_intr.o ehci_util.o ehci_polled.o

1730 HUBD_OBJS += hubd.o

1732 USB_MID_OBJS += usb_mid.o

1734 USB_IA_OBJS += usb_ia.o

1736 UWBA_OBJS += uwba.o uwbai.o

1738 SCSA2USB_OBJS += scsa2usb.o usb_ms_bulkonly.o usb_ms_cbi.o

1740 HWAHC_OBJS += hwahc.o hwahc_util.o

1742 WUSB_DF_OBJS += wusb_df.o
1743 WUSB_FWMOD_OBJS += wusb_fwmod.o

1745 IPF_OBJS += ip_fil_solaris.o fil.o solaris.o ip_state.o ip_frag.o ip_nat.o \
1746             ip_proxy.o ip_auth.o ip_pool.o ip_htable.o ip_lookup.o \
1747             ip_log.o misc.o ip_compat.o ip_nat6.o drand48.o

1749 IBD_OBJS +=     ibd.o ibd_cm.o

1751 EIBNX_OBJS +=   enx_main.o enx_hdlrs.o enx_ibt.o enx_log.o enx_fip.o \
1752                 enx_misc.o enx_q.o enx_ctl.o

1754 EOIB_OBJS +=    eib_adm.o eib_chan.o eib_cmn.o eib_ctl.o eib_data.o \
1755                 eib_fip.o eib_ibt.o eib_log.o eib_mac.o eib_main.o \
1756                 eib_rsrc.o eib_svc.o eib_vnic.o

1758 DLPISTUB_OBJS += dlpistub.o

1760 SDP_OBJS +=     sdpddi.o

1762 TRILL_OBJS +=   trill.o

1764 CTF_OBJS += ctf_create.o ctf_decl.o ctf_error.o ctf_hash.o ctf_labels.o \
1765         ctf_lookup.o ctf_open.o ctf_types.o ctf_util.o ctf_subr.o ctf_mod.o

1767 SMBIOS_OBJS += smb_error.o smb_info.o smb_open.o smb_subr.o smb_dev.o

1769 RPCIB_OBJS += rpcib.o

1771 KMDB_OBJS += kdrv.o

1773 AFE_OBJS += afe.o
```



```
1775 BGE_OBJS += bge_main2.o bge_chip2.o bge_kstats.o bge_log.o bge_ndd.o \
1776                 bge_atomic.o bge_mii.o bge_send.o bge_recv2.o bge_mii_5906.o

1778 DMFE_OBJS += dmfe_log.o dmfe_main.o dmfe_mii.o

1780 EFE_OBJS += efe.o

1782 ELXL_OBJS += elxl.o

1784 HME_OBJS += hme.o

1786 IXGB_OBJS += ixgb.o ixgb_atomic.o ixgb_chip.o ixgb_gld.o ixgb_kstats.o \
1787                 ixgb_log.o ixgb_ndd.o ixgb_rx.o ixgb_tx.o ixgb_xmii.o

1789 NGE_OBJS += nge_main.o nge_atomic.o nge_chip.o nge_ndd.o nge_kstats.o \
1790                 nge_log.o nge_rx.o nge_tx.o nge_xmii.o

1792 PCN_OBJS += pcn.o

1794 RGE_OBJS += rge_main.o rge_chip.o rge_ndd.o rge_kstats.o rge_log.o rge_rxtx.o

1796 URTW_OBJS += urtw.o

1798 ARN_OBJS += arn_hw.o arn_eeprom.o arn_mac.o arn_calib.o arn_ani.o arn_phy.o arn_
1799                 arn_main.o arn_recv.o arn_xmit.o arn_rc.o

1801 ATH_OBJS += ath_aux.o ath_main.o ath_osdep.o ath_rate.o

1803 ATU_OBJS += atu.o

1805 IPW_OBJS += ipw2100_hw.o ipw2100.o

1807 IWI_OBJS += ipw2200_hw.o ipw2200.o

1809 IWH_OBJS += iwh.o

1811 IWK_OBJS += iwk2.o

1813 IWP_OBJS += iwp.o

1815 MWL_OBJS += mwl.o

1817 MWLFW_OBJS += mwlfw_mode.o

1819 WPI_OBJS += wpi.o

1821 RAL_OBJS += rt2560.o ral_rate.o

1823 RUM_OBJS += rum.o

1825 RWD_OBJS += rt2661.o

1827 RWN_OBJS += rt2860.o

1829 UATH_OBJS += uath.o

1831 UATHFW_OBJS += uathfw_mod.o

1833 URAL_OBJS += ural.o

1835 RTW_OBJS += rtw.o smc93cx6.o rtwphy.o rtwphyio.o

1837 ZYD_OBJS += zyd.o zyd_usb.o zyd_hw.o zyd_fw.o

1839 MXFE_OBJS += mxfe.o
```

```
1841 MPTSAS_OBJS += mptsas.o mptsas_impl.o mptsas_init.o mptsas_raid.o mptsas_smhba.o

1843 SFE_OBJS += sfe.o sfe_util.o

1845 BFE_OBJS += bfe.o

1847 BRIDGE_OBJS += bridge.o

1849 IDM_SHARED_OBJS += base64.o

1851 IDM_OBJS +=     $(IDM_SHARED_OBJS) \
1852                 idm.o idm_impl.o idm_text.o idm_conn_sm.o idm_so.o

1854 VR_OBJS += vr.o

1856 ATGE_OBJS += atge_main.o atge_l1e.o atge_mii.o atge_l1.o atge_l1c.o

1858 YGE_OBJS = yge.o

1860 #
1861 #       Build up defines and paths.
1862 #
1863 LINT_DEFS       += -Dunix

1865 #
1866 #       This duality can be removed when the native and target compilers
1867 #       are the same (or at least recognize the same command line syntax!)
1868 #       It is a bug in the current compilation system that the assember
1869 #       can't process the -Y I, flag.
1870 #
1871 NATIVE_INC_PATH += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1872 AS_INC_PATH     += $(INC_PATH) -I$(UTSBASE)/common
1873 INCLUDE_PATH    += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common

1875 PCIEB_OBJS += pcieb.o

1877 #       Chelsio N110 10G NIC driver module
1878 #
1879 CH_OBJS = ch.o glue.o pe.o sge.o

1881 CH_COM_OBJS =   ch_mac.o ch_subr.o cspi.o espi.o ixf1010.o mc3.o mc4.o mc5.o \
1882                 mv88e1xxx.o mv88x201x.o my3126.o pm3393.o tp.o ulp.o \
1883                 vsc7321.o vsc7326.o xpak.o

1885 #
1886 #       PCI strings file
1887 #
1888 PCI_STRING_OBJS = pci_strings.o

1890 NET_DACF_OBJS += net_dacf.o

1892 #
1893 #       Xframe 10G NIC driver module
1894 #
1895 XGE_OBJS = xge.o xgell.o

1897 XGE_HAL_OBJS =  xgehal-channel.o xgehal-fifo.o xgehal-ring.o  xgehal-config.o \
1898                 xgehal-driver.o  xgehal-mm.o xgehal-stats.o  xgehal-device.o \
1899                 xge-queue.o  xgehal-mgmt.o xgehal-mgmtaux.o

1901 #
1902 #       e1000g module
1903 #
1904 E1000G_OBJS +=  e1000_80003es2lan.o e1000_82540.o e1000_82541.o e1000_82542.o \
1905                 e1000_82543.o e1000_82571.o e1000_api.o e1000_ich8lan.o \
```



```
1906                 e1000_mac.o e1000_manage.o e1000_nvm.o e1000_osdep.o \
1907                 e1000_phy.o e1000g_debug.o e1000g_main.o e1000g_alloc.o \
1908                 e1000g_tx.o e1000g_rx.o e1000g_stat.o

1910 #
1911 #       Intel 82575 1G NIC driver module
1912 #
1913 IGB_OBJS =      igb_82575.o igb_api.o igb_mac.o igb_manage.o \
1914                 igb_nvm.o igb_osdep.o igb_phy.o igb_buf.o \
1915                 igb_debug.o igb_gld.o igb_log.o igb_main.o \
1916                 igb_rx.o igb_stat.o igb_tx.o

1918 #
1919 #       Intel Pro/100 NIC driver module
1920 #
1921 IPRB_OBJS =     iprb.o

1923 #
1924 #       Intel 10GbE PCIE NIC driver module
1925 #
1926 IXGBE_OBJS =    ixgbe_82598.o ixgbe_82599.o ixgbe_api.o         \
1927                 ixgbe_common.o ixgbe_phy.o                      \
1928                 ixgbe_buf.o ixgbe_debug.o ixgbe_gld.o           \
1929                 ixgbe_log.o ixgbe_main.o                        \
1930                 ixgbe_osdep.o ixgbe_rx.o ixgbe_stat.o           \
1931                 ixgbe_tx.o  ixgbe_x540.o ixgbe_mbx.o

1933 #
1934 #       NIU 10G/1G driver module
1935 #
1936 NXGE_OBJS =     nxge_mac.o nxge_ipp.o nxge_rxdma.o               \
1937                 nxge_txdma.o nxge_txc.o nxge_main.o             \
1938                 nxge_hw.o nxge_fzc.o nxge_virtual.o             \
1939                 nxge_send.o nxge_classify.o nxge_fflp.o         \
1940                 nxge_fflp_hash.o nxge_ndd.o nxge_kstats.o       \
1941                 nxge_zcp.o nxge_fm.o nxge_espc.o nxge_hv.o      \
1942                 nxge_hio.o nxge_hio_guest.o nxge_intr.o

1944 NXGE_NPI_OBJS = \
1945                 npi.o npi_mac.o npi_ipp.o                       \
1946                 npi_txdma.o npi_rxdma.o npi_txc.o               \
1947                 npi_zcp.o npi_espc.o npi_fflp.o                 \
1948                 npi_vir.o

1950 NXGE_HCALL_OBJS =       \
1951                 nxge_hcall.o

1953 #
1954 #       kiconv modules
1955 #
1956 KICONV_EMEA_OBJS += kiconv_emea.o

1958 KICONV_JA_OBJS += kiconv_ja.o

1960 KICONV_KO_OBJS += kiconv_cck_common.o kiconv_ko.o

1962 KICONV_SC_OBJS += kiconv_cck_common.o kiconv_sc.o

1964 KICONV_TC_OBJS += kiconv_cck_common.o kiconv_tc.o

1966 #
1967 #       AAC module
1968 #
1969 AAC_OBJS = aac.o aac_ioctl.o

1971 #
```

```
#       sdcard modules
#
SDA_OBJS =      sda_cmd.o sda_host.o sda_init.o sda_mem.o sda_mod.o sda_slot.o
SDHOST_OBJS =   sdhost.o

#
#       hxge 10G driver module
#
HXGE_OBJS =     hxge_main.o hxge_vmac.o hxge_send.o             \
                hxge_txdma.o hxge_rxdma.o hxge_virtual.o        \
                hxge_fm.o hxge_fzc.o hxge_hw.o hxge_kstats.o    \
                hxge_ndd.o hxge_pfc.o                           \
                hpi.o hpi_vmac.o hpi_rxdma.o hpi_txdma.o        \
                hpi_vir.o hpi_pfc.o

#
#       MEGARAID_SAS module
#
MEGA_SAS_OBJS = megaraid_sas.o

#
#       MR_SAS module
#
MR_SAS_OBJS = mr_sas.o

#
#       ISCSI_INITIATOR module
#
ISCSI_INITIATOR_OBJS =  chap.o iscsi_io.o iscsi_thread.o        \
                        iscsi_ioctl.o iscsid.o iscsi.o          \
                        iscsi_login.o isns_client.o iscsiAuthClient.o   \
                        iscsi_lun.o iscsiAuthClientGlue.o       \
                        iscsi_net.o nvfile.o iscsi_cmd.o        \
                        iscsi_queue.o persistent.o iscsi_conn.o \
                        iscsi_sess.o radius_auth.o iscsi_crc.o  \
                        iscsi_stats.o radius_packet.o iscsi_doorclt.o   \
                        iscsi_targetparam.o utils.o kifconf.o

#
#       ntxn 10Gb/1Gb NIC driver module
#
NTXN_OBJS =     unm_nic_init.o unm_gem.o unm_nic_hw.o unm_ndd.o \
                        unm_nic_main.o unm_nic_isr.o unm_nic_ctx.o niu.o

#
#       Myricom 10Gb NIC driver module
#
MYRI10GE_OBJS = myri10ge.o myri10ge_lro.o

#       nulldriver module
#
NULLDRIVER_OBJS =       nulldriver.o

TPM_OBJS =      tpm.o tpm_hcall.o
```

****
117504 Fri Oct 26 17:09:24 2012
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
FAR: generating send-streams in portable format
This commit adds a switch '-F' to zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
****
_____***unchanged_portion_omitted_***

```
 683 int
 683 static int
 684 dsl_dataset_namelen(dsl_dataset_t *ds)
 685 {
 686         int result;

 688         if (ds == NULL) {
 689                 result = 3;     /* "mos" */
 690         } else {
 691                 result = dsl_dir_namelen(ds->ds_dir);
 692                 VERIFY(0 == dsl_dataset_get_snapname(ds));
 693                 if (ds->ds_snapname[0]) {
 694                         ++result;       /* adding one for the @-sign */
 695                         if (!MUTEX_HELD(&ds->ds_lock)) {
 696                                 mutex_enter(&ds->ds_lock);
 697                                 result += strlen(ds->ds_snapname);
 698                                 mutex_exit(&ds->ds_lock);
 699                         } else {
 700                                 result += strlen(ds->ds_snapname);
 701                         }
 702                 }
 703         }

 705         return (result);
 706 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   30762 Fri Oct 26 17:09:24 2012
new/usr/src/uts/common/fs/zfs/far.c
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * 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) 2012 STRATO AG. All rights reserved.
 */
#include <sys/zfs_context.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <sys/mkdev.h>
#include <sys/debug.h>
#include <sys/open.h>
#include <sys/zfs_ioctl.h>
#include <zfs_namecheck.h>
#include <sys/policy.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_prop.h>
#include <sys/zvol.h>
#include <sys/zap.h>
#include <sys/dsl_dataset.h>
#include <sys/dmu_traverse.h>
#include <sys/dsl_dir.h>
#include <sys/arc.h>
#include <sys/spa.h>
#include <sys/spa_impl.h>
#include <sys/sa.h>
#include <sys/sa_impl.h>
#include <sys/zfs_acl.h>
#include <sys/zfs_sa.h>
#include <sys/zfs_znode.h>
#include <sys/dbuf.h>
#include <sys/far.h>
#include <sys/far_impl.h>

/*
 * far_send generates a stream of filesystem data analogous to dmu_send.
 * The main difference is that the far-stream does not contain zfs-specific
 * data and can be replayed on any filesystem. It just contains commands like
 * MKDIR, CHMOD, RENAME etc.
 * The stream is generated in two passes. The first pass, PASS_LINK basically
```



```
 * creates all new files/directories and links, while the second pass,
 * PASS_UNLINK, does all the removal of old stuff.
 * Each pass enumerates all objects in inode order.
 * There are some corner cases:
 *   Files / directories can only be created if the parent already exists or
 * already has been created. If an object is encountered which parent does not
 * satisfy this condition, it is put back and its creation will be trigger
 * by the creation of the parent.
 *   A similar case applies on deletion. A directory can only be removed after
 * the last contained object has been removed. If a directory is not empty,
 * it is put back and the deletion of the last object in it triggers the
 * deletion.
 *   If an objects gets deleted, and a new object is created under the same
 * name, pass1 cannot create the object directly. So it is created under a
 * temporary name and gets renamed in pass2.
 *   If an object is deleted and a new object (of possibly diffent type)
 * created under the same inode and the same name, this change cannot be
 * detected by enumerating the containing directory (as name + inode are
 * unchanged). It is detected by a change of the inode generation number and
 * a flag is set for pass2. Creation is postponed. In pass2, all enumerated
 * directories are checked for this inode (although the entry is unchanged,
 * the directory has a bumped txg). If it is encountered, delete + create
 * happen both in pass2.
 *
 * There are lots of TODOs left:
 *  - add XATTR support
 *  - add path-caching
 *  - add a cache for brute-force parent search
 *  - add a cache for inode-search in a directory
 *  - use a hash instead of the linear list in far_count
 */
static int
far_dnode_changed(spa_t *spa, far_t *f, uint64_t dnobj,
    dnode_phys_t *from, arc_buf_t *frombuf, dnode_phys_t *to, arc_buf_t *tobuf);

/* copied from zfs_znode.c */
static int
far_sa_setup(objset_t *osp, sa_attr_type_t **sa_table)
{
        uint64_t sa_obj = 0;
        int error;

        error = zap_lookup(osp, MASTER_NODE_OBJ, ZFS_SA_ATTRS, 8, 1, &sa_obj);
        if (error != 0 && error != ENOENT)
                return (error);

        error = sa_setup(osp, sa_obj, zfs_attr_table, ZPL_END, sa_table);
        return (error);
}

static int
far_grab_sa_handle(objset_t *osp, uint64_t obj, sa_handle_t **hdlp,
    dmu_buf_t **db, void *tag)
{
        dmu_object_info_t doi;
        int error;

        if ((error = sa_buf_hold(osp, obj, tag, db)) != 0)
                return (error);

        dmu_object_info_from_db(*db, &doi);
        if ((doi.doi_bonus_type != DMU_OT_SA &&
            doi.doi_bonus_type != DMU_OT_ZNODE) ||
            (doi.doi_bonus_type == DMU_OT_ZNODE &&
            doi.doi_bonus_size < sizeof (znode_phys_t))) {
                sa_buf_rele(*db, tag);
```

```
124                 return (ENOTSUP);
125         }

127         error = sa_handle_get(osp, obj, NULL, SA_HDL_PRIVATE, hdlp);
128         if (error != 0) {
129                 sa_buf_rele(*db, tag);
130                 return (error);
131         }

133         return (0);
134 }

136 static void
137 far_release_sa_handle(sa_handle_t *hdl, dmu_buf_t *db, void *tag)
138 {
139         sa_handle_destroy(hdl);
140         sa_buf_rele(db, tag);
141 }

143 static int
144 far_find_from_bp(spa_t *spa, dnode_phys_t *dnp, blklevel_t *bl,
145     const zbookmark_t *zb, blkptr_t **bpp, arc_buf_t **pbuf)
146 {
147         uint32_t flags;
148         int epbs = dnp->dn_indblkshift - SPA_BLKPTRSHIFT;
149         int epbmask = (1 << epbs) - 1;
150         int level;
151         int slot;
152         uint64_t blkid;
153         uint64_t blk;
154         blklevel_t *blp;
155         zbookmark_t czb;
156         int i;

158         *bpp = NULL;
159         for (level = dnp->dn_nlevels - 1; level >= zb->zb_level; --level) {
160                 blkid = zb->zb_blkid >> (epbs * (level - zb->zb_level));
161                 blk = blkid >> epbs;
162                 slot = blk & epbmask;
163                 blp = bl + level;

165                 if (blp->bl_blk == blk)
166                         continue;

168                 for (i = 0; i <= level; ++i) {
169                         blklevel_t *b = bl + i;

171                         if (b->bl_buf)
172                                 arc_buf_remove_ref(b->bl_buf, &b->bl_buf);
173                         b->bl_bp = NULL;
174                         b->bl_buf = NULL;
175                         b->bl_blk = -1;
176                 }
177                 ASSERT(slot < blp[1].bl_nslots);
178                 if (BP_IS_HOLE(blp[1].bl_bp + slot)) {
179                         *bpp = NULL;
180                         return (0);
181                 }
182                 /*
183                  * load indblk
184                  */
185                 flags = ARC_WAIT;
186                 SET_BOOKMARK(&czb, zb->zb_objset, zb->zb_object, level, blkid);
187                 if (dsl_read(NULL, spa, blp[1].bl_bp + slot, blp[1].bl_buf,
188                     arc_getbuf_func, &blp->bl_buf, ZIO_PRIORITY_ASYNC_READ,
189                     ZIO_FLAG_CANFAIL, &flags, &czb) != 0)
```



```
190                         return (EIO);
191                 blp->bl_bp = blp->bl_buf->b_data;
192                 blp->bl_nslots = 1 << epbs;
193                 blp->bl_blk = blk;
194         }
195         slot = zb->zb_blkid & epbmask;
196         blp = bl + zb->zb_level;
197         ASSERT(slot < blp->bl_nslots);
198         *bpp = blp->bl_bp + slot;
199         *pbuf = blp->bl_buf;
200         if (BP_IS_HOLE(*bpp))
201                 *bpp = NULL;

203         return (0);
204 }

206 static int
207 far_file_cb(spa_t *spa, far_t *f, zbookmark_t *zb,
208     blkptr_t *bp, arc_buf_t *pbuf, void *ctx)
209 {
210         int err = 0;
211         blkptr_t *fbp;

213         if (issig(JUSTLOOKING) && issig(FORREAL))
214                 return (EINTR);

216         if (f->f_fromds && zb->zb_objset == f->f_fromds->ds_object)
217                 return (0);

219         if (bp == NULL) {
220                 arc_buf_t *fpbuf = NULL;
221                 zbookmark_t czb;

223                 ASSERT(f->f_fromds);
224                 SET_BOOKMARK(&czb, f->f_fromds->ds_object, zb->zb_object,
225                     zb->zb_level, zb->zb_blkid);
226                 err = far_find_from_bp(spa, f->f_dnp, f->f_filebl,
227                     &czb, &fbp, &fpbuf);
228                 if (err)
229                         return (err);
230                 if (fbp) {
231 #if 0
232                         /* XXX TODO callback for newly created hole */
233                         err = far_enum_bp(spa, da, &czb, fbp, fpbuf);
234                         if (err)
235                                 return (err);
236 #endif
237                 }
238         } else if (zb->zb_level == 0) {
239                 arc_buf_t *tbuf;
240                 uint32_t tflags = ARC_WAIT;
241                 int blksz = BP_GET_LSIZE(bp);

243                 if (dsl_read(NULL, spa, bp, pbuf,
244                     arc_getbuf_func, &tbuf, ZIO_PRIORITY_ASYNC_READ,
245                     ZIO_FLAG_CANFAIL, &tflags, zb) != 0)
246                         return (EIO);

248                 if (f->f_ops->far_file_data)
249                         err = f->f_ops->far_file_data(ctx, tbuf->b_data,
250                             zb->zb_blkid * blksz, blksz);

252                 (void) arc_buf_remove_ref(tbuf, &tbuf);
253         }
254         return (err);
255 }
```

```
257 static int
258 far_enum_bp(spa_t *spa, far_t *da, zbookmark_t *zb,
259     blkptr_t *bp, arc_buf_t *pbuf, uint64_t min_txg, void *ctx)
260 {
261         int err = 0;
262         arc_buf_t *buf = NULL;
263         uint32_t flags = ARC_WAIT;

265         if (BP_IS_HOLE(bp))
266                 return (0);

268         if (bp->blk_birth <= min_txg)
269                 return (0);

271         if (BP_GET_LEVEL(bp) > 0) {
272                 int i;
273                 int epb = BP_GET_LSIZE(bp) >> SPA_BLKPTRSHIFT;
274                 blkptr_t *cbp;
275                 zbookmark_t czb;

277                 if (dsl_read(NULL, spa, bp, pbuf, arc_getbuf_func, &buf,
278                     ZIO_PRIORITY_ASYNC_READ, ZIO_FLAG_CANFAIL, &flags, zb) != 0)
279                         return (EIO);
280                 cbp = buf->b_data;
281                 for (i = 0; i < epb; ++i, ++cbp) {
282                         SET_BOOKMARK(&czb, zb->zb_objset, zb->zb_object,
283                             zb->zb_level - 1, zb->zb_blkid * epb + i);
284                         err = far_enum_bp(spa, da, &czb, cbp, buf, min_txg,
285                             ctx);
286                         if (err)
287                                 goto out;
288                 }
289         } else if (BP_GET_TYPE(bp) == DMU_OT_DNODE) {
290                 int i;
291                 int epb = BP_GET_LSIZE(bp) >> DNODE_SHIFT;
292                 dnode_phys_t *dnp;

294                 if (dsl_read(NULL, spa, bp, pbuf, arc_getbuf_func, &buf,
295                     ZIO_PRIORITY_ASYNC_READ, ZIO_FLAG_CANFAIL,
296                     &flags, zb) != 0) {
297                         err = EIO;
298                         goto out;
299                 }
300                 dnp = buf->b_data;
301                 for (i = 0; i < epb; ++i, ++dnp) {
302                         uint64_t dnobj = zb->zb_blkid * epb + i;
303                         if (dnp->dn_type == DMU_OT_NONE)
304                                 continue;
305                         err = far_dnode_changed(spa, da, dnobj, dnp, buf,
306                             NULL, NULL);
307                         if (err)
308                                 goto out;
309                 }
310         } else {
311                 err = far_file_cb(spa, da, zb, bp, pbuf, ctx);
312         }
313 out:
314         if (buf)
315                 (void) arc_buf_remove_ref(buf, &buf);

317         return (err);
318 }

320 static int
321 far_cb(spa_t *spa, zilog_t *zilog, const blkptr_t *bp, arc_buf_t *pbuf,
```



```
322     const zbookmark_t *zb, const dnode_phys_t *dnp, void *arg)
323 {
324         int err = 0;
325         far_t *f = arg;
326         blkptr_t *fbp = NULL;
327         zbookmark_t czb;

329         if (issig(JUSTLOOKING) && issig(FORREAL))
330                 return (EINTR);

332         if (f->f_fromds)
333                 SET_BOOKMARK(&czb, f->f_fromds->ds_object, zb->zb_object,
334                     zb->zb_level, zb->zb_blkid);

336         if (zb->zb_object != DMU_META_DNODE_OBJECT)
337                 return (0);

339         if (bp == NULL) {
340                 arc_buf_t *fpbuf = NULL;

342                 if (!f->f_fromds)
343                         return (0);

345                 err = far_find_from_bp(spa, f->f_dnp, f->f_bl,
346                     &czb, &fbp, &fpbuf);
347                 if (err)
348                         return (EIO);
349                 if (fbp) {
350                         err = far_enum_bp(spa, f, &czb, fbp, fpbuf, 0, NULL);
351                         if (err)
352                                 return (EIO);
353                 }
354                 return (0);
355         } else if (zb->zb_level == 0) {
356                 dnode_phys_t *tblk;
357                 dnode_phys_t *fblk = NULL;
358                 arc_buf_t *tbuf;
359                 arc_buf_t *fbuf = NULL;
360                 arc_buf_t *fpbuf = NULL;
361                 uint32_t fflags = ARC_WAIT;
362                 uint32_t tflags = ARC_WAIT;
363                 int blksz = BP_GET_LSIZE(bp);
364                 int i;

366                 if (dsl_read(NULL, spa, bp, pbuf,
367                     arc_getbuf_func, &tbuf, ZIO_PRIORITY_ASYNC_READ,
368                     ZIO_FLAG_CANFAIL, &tflags, zb) != 0)
369                         return (EIO);
370                 tblk = tbuf->b_data;

372                 if (f->f_fromds) {
373                         err = far_find_from_bp(spa, f->f_dnp, f->f_bl, zb,
374                             &fbp, &fpbuf);
375                         if (err)
376                                 return (EIO);
377                 }
378                 if (fbp) {
379                         if (dsl_read(NULL, spa, fbp, fpbuf,
380                             arc_getbuf_func, &fbuf, ZIO_PRIORITY_ASYNC_READ,
381                             ZIO_FLAG_CANFAIL, &fflags, &czb) != 0) {
382                                 (void) arc_buf_remove_ref(tbuf, &tbuf);
383                                 return (EIO);
384                         }
385                         fblk = fbuf->b_data;
386                         if (blksz != BP_GET_LSIZE(fbp))
387                                 return (EIO);
```

```
388                 }
389                 for (i = 0; i < blksz >> DNODE_SHIFT; i++) {
390                         uint64_t dnobj = (zb->zb_blkid <<
391                             (DNODE_BLOCK_SHIFT - DNODE_SHIFT)) + i;
392                         err = 0;
393                         if (fbuf && (tblk[i].dn_type == DMU_OT_NONE) &&
394                             fblk[i].dn_type != DMU_OT_NONE) {
395                                 err = far_dnode_changed(spa, f, dnobj,
396                                     fblk + i, fbuf, NULL, NULL);
397                         } else if (fbuf) {
398                                 if (memcmp(tblk + i, fblk + i, sizeof (*tblk)))
399                                         err = far_dnode_changed(spa, f,
400                                             dnobj, fblk + i, fbuf, tblk + i,
401                                             tbuf);
402                         } else {
403                                 if (tblk[i].dn_type != DMU_OT_NONE)
404                                         err = far_dnode_changed(spa, f,
405                                             dnobj, NULL, NULL, tblk + i, tbuf);
406                         }
407                         if (err)
408                                 break;
409                 }
410                 (void) arc_buf_remove_ref(tbuf, &tbuf);
411                 if (fbuf)
412                         (void) arc_buf_remove_ref(fbuf, &fbuf);
413
414                 if (err)
415                         return (EIO);
416                 /* Don't care about the data blocks */
417                 return (TRAVERSE_VISIT_NO_CHILDREN);
418         }
419         return (0);
420 }
421
422 #define DIR_FROM        1
423 #define DIR_TO          2
424 static int
425 far_diff_dir(far_t *f, uint64_t dnobj, int dir, void *ctx)
426 {
427         zap_cursor_t zc;
428         zap_attribute_t *za;
429         int err;
430         objset_t *os1;
431         objset_t *os2;
432         uint64_t mask =  ZFS_DIRENT_OBJ(-1ULL);
433         uint64_t num;
434         uint64_t ix = 0;
435
436         if (dir == DIR_FROM) {
437                 os1 = f->f_fromsnap;
438                 os2 = f->f_tosnap;
439         } else {
440                 os1 = f->f_tosnap;
441                 os2 = f->f_fromsnap;
442         }
443
444         za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
445         for (zap_cursor_init(&zc, os1, dnobj);
446             (err = zap_cursor_retrieve(&zc, za)) == 0;
447             zap_cursor_advance(&zc), ++ix) {
448                 err = zap_lookup(os2, dnobj, za->za_name, sizeof (num), 1,
449                     &num);
450                 if (err && err != ENOENT)
451                         break;
452                 if (err == ENOENT) {
453                         if (dir == DIR_FROM) {
```



```
454                                 if (f->f_ops->far_dirent_del) {
455                                         err = f->f_ops->far_dirent_del(ctx,
456                                             za->za_name,
457                                             za->za_first_integer & mask);
458                                         if (err)
459                                                 goto out;
460                                 }
461                         } else {
462                                 if (f->f_ops->far_dirent_add) {
463                                         err = f->f_ops->far_dirent_add(ctx,
464                                             za->za_name,
465                                             za->za_first_integer & mask);
466                                         if (err)
467                                                 goto out;
468                                 }
469                         }
470                 } else if ((za->za_first_integer & mask) != (num & mask)) {
471                         if (dir == DIR_TO) {
472                                 /* report only once */
473                                 if (f->f_ops->far_dirent_mod) {
474                                         err = f->f_ops->far_dirent_mod(ctx,
475                                             za->za_name, num & mask,
476                                             za->za_first_integer & mask);
477                                         if (err)
478                                                 goto out;
479                                 }
480                         }
481                 } else {
482                         if (dir == DIR_TO) {
483                                 /* report only once */
484                                 if (f->f_ops->far_dirent_unmod) {
485                                         err = f->f_ops->far_dirent_unmod(ctx,
486                                             za->za_name, num & mask);
487                                         if (err)
488                                                 goto out;
489                                 }
490                         }
491                 }
492         }
493         err = 0;
494 out:
495         zap_cursor_fini(&zc);
496         kmem_free(za, sizeof (zap_attribute_t));
497
498         return (err);
499 }
500 static int
501 far_enum_dir(far_t *f, uint64_t dnobj, int dir, void *ctx)
502 {
503         zap_cursor_t zc;
504         zap_attribute_t *za;
505         int err;
506         objset_t *os;
507         uint64_t mask =  ZFS_DIRENT_OBJ(-1ULL);
508
509         if (dir == DIR_FROM)
510                 os = f->f_fromsnap;
511         else
512                 os = f->f_tosnap;
513
514         za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
515         for (zap_cursor_init(&zc, os, dnobj);
516             (err = zap_cursor_retrieve(&zc, za)) == 0;
517             zap_cursor_advance(&zc)) {
518                 if (dir == DIR_FROM) {
519                         if (f->f_ops->far_dirent_del) {
```

```
520                                 err = f->f_ops->far_dirent_del(ctx,
521                                     za->za_name, za->za_first_integer & mask);
522                                 if (err)
523                                         break;
524                         }
525                 } else {
526                         if (f->f_ops->far_dirent_add) {
527                                 err = f->f_ops->far_dirent_add(ctx,
528                                     za->za_name, za->za_first_integer & mask);
529                                 if (err)
530                                         break;
531                         }
532                 }
533         }
534         if (err == ENOENT)
535                 err = 0;
536
537         zap_cursor_fini(&zc);
538         kmem_free(za, sizeof (zap_attribute_t));
539
540         return (err);
541 }
542
543 static int
544 far_dnode_changed(spa_t *spa, far_t *f, uint64_t dnobj,
545     dnode_phys_t *from, arc_buf_t *frombuf, dnode_phys_t *to, arc_buf_t *tobuf)
546 {
547         int err = 0;
548         int type = 0;
549         far_info_t si;
550
551         if (dnobj == f->f_shares_dir)
552                 return (0);
553
554         if (to && to->dn_type != DMU_OT_PLAIN_FILE_CONTENTS &&
555             to->dn_type != DMU_OT_DIRECTORY_CONTENTS) {
556                 to = NULL;
557         }
558         if (from && from->dn_type != DMU_OT_PLAIN_FILE_CONTENTS &&
559             from->dn_type != DMU_OT_DIRECTORY_CONTENTS) {
560                 from = NULL;
561         }
562
563         if (from) {
564                 err = far_get_info(f, dnobj, FAR_OLD, &si, FI_ATTR_LINKS);
565                 if (err)
566                         return (err);
567                 if (si.si_nlinks == 0)
568                         from = NULL;
569         }
570         if (to) {
571                 err = far_get_info(f, dnobj, FAR_NEW, &si, FI_ATTR_LINKS);
572                 if (err)
573                         return (err);
574                 if (si.si_nlinks == 0)
575                         to = NULL;
576         }
577
578         if (!to && !from)
579                 return (0);
580
581         if (from) {
582                 if (from->dn_bonustype != DMU_OT_SA &&
583                     from->dn_bonustype != DMU_OT_ZNODE)
584                         return (EINVAL);
585         }
```



```
586         if (to) {
587                 if (to->dn_bonustype != DMU_OT_SA &&
588                     to->dn_bonustype != DMU_OT_ZNODE)
589                         return (EINVAL);
590         }
591
592         if (from)
593                 type = from->dn_type;
594         else if (to)
595                 type = to->dn_type;
596
597         err = 0;
598         if (type == DMU_OT_DIRECTORY_CONTENTS) {
599                 if (from && to) {
600                         if (f->f_ops->far_dir_mod)
601                                 err = f->f_ops->far_dir_mod(f, dnobj);
602                 } else if (from) {
603                         if (f->f_ops->far_dir_del)
604                                 err = f->f_ops->far_dir_del(f, dnobj);
605                 } else if (to) {
606                         if (f->f_ops->far_dir_add)
607                                 err = f->f_ops->far_dir_add(f, dnobj);
608                 }
609         } else if (type == DMU_OT_PLAIN_FILE_CONTENTS) {
610                 if (from && to) {
611                         if (f->f_ops->far_file_mod)
612                                 err = f->f_ops->far_file_mod(f, dnobj);
613                 } else if (from) {
614                         if (f->f_ops->far_file_del)
615                                 err = f->f_ops->far_file_del(f, dnobj);
616                 } else if (to) {
617                         if (f->f_ops->far_file_add)
618                                 err = f->f_ops->far_file_add(f, dnobj);
619                 }
620         } else {
621                 /* TODO other types, symlinks? */
622                 err = 0;
623         }
624         return (err);
625 }
626
627 typedef struct _far_search {
628         far_t *zs_f;
629         uint64_t zs_dnobj;
630         uint64_t zs_parent;
631         objset_t *zs_osp;
632 } far_search_t;
633
634 static int
635 search_cb(spa_t *spa, zilog_t *zilog, const blkptr_t *bp, arc_buf_t *pbuf,
636     const zbookmark_t *zb, const dnode_phys_t *dnp, void *arg)
637 {
638         far_search_t *zs = arg;
639         far_t *f = zs->zs_f;
640         arc_buf_t *buf;
641         uint32_t flags = ARC_WAIT;
642         int ebp;
643         int i;
644         int ret;
645
646         if (issig(JUSTLOOKING) && issig(FORREAL))
647                 return (EINTR);
648
649         if (zb->zb_object != DMU_META_DNODE_OBJECT)
650                 return (0);
```

```
652         if (zb->zb_level != 0)
653                 return (0);

655         if (!bp || BP_IS_HOLE(bp))
656                 return (0);

658         if (BP_GET_TYPE(bp) != DMU_OT_DNODE)
659                 return (0);

661         ebp = BP_GET_LSIZE(bp) >> DNODE_SHIFT;

663         if (dsl_read(NULL, spa, bp, pbuf,
664             arc_getbuf_func, &buf, ZIO_PRIORITY_ASYNC_READ,
665             ZIO_FLAG_CANFAIL, &flags, zb) != 0)
666                 return (EIO);
667         dnp = buf->b_data;

669         for (i = 0; i < ebp; ++i) {
670                 zap_cursor_t zc;
671                 zap_attribute_t *za;
672                 uint64_t mask = ZFS_DIRENT_OBJ(-1ULL);
673                 uint64_t ix = 0;
674                 uint64_t dnobj = (zb->zb_blkid <<
675                         (DNODE_BLOCK_SHIFT - DNODE_SHIFT)) + i;

677                 if (dnp[i].dn_type != DMU_OT_DIRECTORY_CONTENTS)
678                         continue;
679                 if (dnobj == f->f_shares_dir)
680                         continue;

682                 za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
683                 for (zap_cursor_init(&zc, zs->zs_osp, dnobj);
684                     (ret = zap_cursor_retrieve(&zc, za)) == 0;
685                     zap_cursor_advance(&zc), ++ix) {
686                         if ((za->za_first_integer & mask) ==
687                             (zs->zs_dnobj & mask)) {
688                                 zs->zs_parent = dnobj;
689                                 break;
690                         }
691                 }
692                 zap_cursor_fini(&zc);
693                 kmem_free(za, sizeof (zap_attribute_t));
694         }

696         (void) arc_buf_remove_ref(buf, &buf);

698         if (zs->zs_parent)
699                 return (EIO);   /* abort search */

701         return (TRAVERSE_VISIT_NO_CHILDREN);
702 }

704 static int
705 far_search_parent(far_t *f, uint64_t dnobj, far_which_t which,
706     uint64_t *parent)
707 {
708         dsl_dataset_t *ds;
709         far_search_t zs;
710         int ret;

712         if (which == FAR_OLD) {
713                 ds = f->f_fromds;
714                 zs.zs_osp = f->f_fromsnap;
715         } else {
716                 ds = f->f_tods;
717                 zs.zs_osp = f->f_tosnap;
```



```
718         }

720         zs.zs_f = f;
721         zs.zs_dnobj = dnobj;
722         zs.zs_parent = 0;
723         ret = traverse_dataset(ds, 0, TRAVERSE_PRE, search_cb, &zs);
724         if (zs.zs_parent) {
725                 *parent = zs.zs_parent;
726                 return (0);
727         }

729         return (ret ? ret : ENOENT);
730 }

732 int
733 far_get_info(far_t *f, uint64_t dnobj, far_which_t which,
734     far_info_t *sp, uint64_t flags)
735 {
736         int ret;
737         sa_handle_t *hdl = NULL;
738         dmu_buf_t *db;
739         objset_t *osp;
740         sa_bulk_attr_t bulk[13];
741         int count = 0;
742         sa_attr_type_t *sa_table;

744         if (which == FAR_OLD) {
745                 osp = f->f_fromsnap;
746                 if (!osp)
747                         return (ENOENT);
748                 sa_table = f->f_from_sa_table;
749         } else if (which == FAR_NEW) {
750                 osp = f->f_tosnap;
751                 sa_table = f->f_to_sa_table;
752         } else {
753                 return (EINVAL);
754         }

756         ret = far_grab_sa_handle(osp, dnobj, &hdl, &db, FTAG);
757         if (ret)
758                 return (ret);

760         if (flags & FI_ATTR_ATIME) {
761                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_ATIME], NULL,
762                     &sp->si_atime, sizeof (sp->si_atime));
763         }
764         if (flags & FI_ATTR_MTIME) {
765                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_MTIME], NULL,
766                     &sp->si_mtime, sizeof (sp->si_mtime));
767         }
768         if (flags & FI_ATTR_CTIME) {
769                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_CTIME], NULL,
770                     &sp->si_ctime, sizeof (sp->si_ctime));
771         }
772         if (flags & FI_ATTR_OTIME) {
773                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_CRTIME], NULL,
774                     &sp->si_otime, sizeof (sp->si_otime));
775         }
776         if (flags & FI_ATTR_MODE) {
777                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_MODE], NULL,
778                     &sp->si_mode, sizeof (sp->si_mode));
779         }
780         if (flags & FI_ATTR_SIZE) {
781                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_SIZE], NULL,
782                     &sp->si_size, sizeof (sp->si_size));
783         }
```

```
784         if (flags & FI_ATTR_PARENT) {
785                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_PARENT], NULL,
786                     &sp->si_parent, sizeof (sp->si_parent));
787         }
788         if (flags & FI_ATTR_LINKS) {
789                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_LINKS], NULL,
790                     &sp->si_nlinks, sizeof (sp->si_nlinks));
791         }
792         if (flags & FI_ATTR_RDEV) {
793                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_RDEV], NULL,
794                     &sp->si_rdev, sizeof (sp->si_rdev));
795         }
796         if (flags & FI_ATTR_UID) {
797                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_UID], NULL,
798                     &sp->si_uid, sizeof (sp->si_uid));
799         }
800         if (flags & FI_ATTR_GID) {
801                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_GID], NULL,
802                     &sp->si_gid, sizeof (sp->si_gid));
803         }
804         if (flags & FI_ATTR_GEN) {
805                 SA_ADD_BULK_ATTR(bulk, count, sa_table[ZPL_GEN], NULL,
806                     &sp->si_gen, sizeof (sp->si_gen));
807         }
808         /* XXX if you add things, also bump the size of bulk */
809         /* XXX XATTR */

811         /* XXX TODO get flags to check for xattrdir */
812         if (count) {
813                 ret = sa_bulk_lookup(hdl, bulk, count);
814                 if (ret)
815                         goto out;
816         }

818         if ((flags & FI_ATTR_PARENT) && sp->si_parent != dnobj) {
819                 far_info_t si;
820                 int good = 0;
821                 /*
822                  * verify parent. this is very expensive and only a workaround
823                  */
824                 ret = far_get_info(f, sp->si_parent, which, &si, FI_ATTR_MODE);
825                 if (ret && ret != ENOENT)
826                         goto out;
827                 if (ret == 0 && S_ISDIR(si.si_mode)) {
828                         ret = far_find_entry(f, sp->si_parent, dnobj, which,
829                             NULL);
830                         if (ret && ret != ENOENT)
831                                 goto out;
832                         if (ret == 0)
833                                 good = 1;
834                 }
835                 if (!good) {
836                         uint64_t parent;

838                         cmn_err(CE_NOTE, "parent wrong, do a brute force "
839                             "search for ino %"PRIu64"\n", dnobj);
840                         ret = far_search_parent(f, dnobj, which, &parent);
841                         if (ret == ENOENT) {
842                                 cmn_err(CE_NOTE, "no parent found\n");
843                                 ret = EINVAL;
844                                 goto out;
845                         }
846                         if (ret)
847                                 goto out;
848                         sp->si_parent = parent;
849                         cmn_err(CE_NOTE, "parent found, use %"PRIu64"\n",
```



```
850                             parent);
851                         /*
852                          * TODO add a bad parent cache to prevent additional
853                          * lookup in pass 2
854                          */
855                 }
856         }

858 out:
859         far_release_sa_handle(hdl, db, FTAG);
860         return (ret);
861 }

863 int
864 far_file_contents(far_t *f, uint64_t dnobj, void *ctx)
865 {
866         dnode_t *from = NULL;
867         dnode_t *to = NULL;
868         int err;
869         int i;
870         zbookmark_t czb;
871         spa_t *spa = f->f_tods->ds_dir->dd_pool->dp_spa;

873         if (f->f_fromds) {
874                 err = dnode_hold(f->f_fromsnap, dnobj, FTAG, &from);
875                 if (err && err != ENOENT)
876                         return (err);
877         }
878         if (from && from->dn_type != DMU_OT_PLAIN_FILE_CONTENTS) {
879                 dnode_rele(from, FTAG);
880                 from = NULL;
881         }
882         err = dnode_hold(f->f_tosnap, dnobj, FTAG, &to);
883         if (err)
884                 goto out;
885         if (to->dn_type != DMU_OT_PLAIN_FILE_CONTENTS) {
886                 err = EINVAL;
887                 goto out;
888         }
889         if (from) {
890                 f->f_filebl = kmem_zalloc(sizeof (blklevel_t)*from->dn_nlevels,
891                                           KM_SLEEP);
892                 for (i = 0; i < from->dn_nlevels; ++i)
893                         f->f_filebl[i].bl_blk = -1;
894                 i = from->dn_nlevels - 1;
895                 f->f_filebl[i].bl_nslots = from->dn_nblkptr;
896                 f->f_filebl[i].bl_bp = &from->dn_phys->dn_blkptr[0];
897                 f->f_filebl[i].bl_blk = 0;
898                 f->f_filebl[i].bl_buf = from->dn_dbuf->db_parent->db_buf;
899         }
900         for (i = 0; i < to->dn_nblkptr; ++i) {
901                 SET_BOOKMARK(&czb, f->f_tods->ds_object, dnobj,
902                     to->dn_nlevels - 1, i);
903                 err = far_enum_bp(spa, f, &czb, to->dn_phys->dn_blkptr + i,
904                     NULL, f->f_fromtxg, ctx);
905                 if (err)
906                         goto out;
907         }
908 out:
909         if (f->f_filebl) {
910                 kmem_free(f->f_filebl, sizeof (blklevel_t) * from->dn_nlevels);
911                 f->f_filebl = NULL;
912         }
913         if (from)
914                 dnode_rele(from, FTAG);
915         if (to)
```

```
916                 dnode_rele(to, FTAG);

918         return (err);
919 }

921 int
922 far_dir_contents(far_t *f, uint64_t dnobj, void *ctx)
923 {
924         dnode_t *from = NULL;
925         dnode_t *to = NULL;
926         int err;

928         if (f->f_fromds) {
929                 err = dnode_hold(f->f_fromsnap, dnobj, FTAG, &from);
930                 if (err && err != ENOENT)
931                         return (err);
932         }
933         if (from && from->dn_type != DMU_OT_DIRECTORY_CONTENTS) {
934                 dnode_rele(from, FTAG);
935                 from = NULL;
936         }
937         err = dnode_hold(f->f_tosnap, dnobj, FTAG, &to);
938         if (err && err != ENOENT)
939                 return (err);
940         if (to && to->dn_type != DMU_OT_DIRECTORY_CONTENTS) {
941                 dnode_rele(to, FTAG);
942                 to = NULL;
943         }

945         if (to && from) {
946                 err = far_diff_dir(f, dnobj, DIR_TO, ctx);
947                 if (err)
948                         goto out;
949                 err = far_diff_dir(f, dnobj, DIR_FROM, ctx);
950         } else if (to) {
951                 err = far_enum_dir(f, dnobj, DIR_TO, ctx);
952         } else if (from) {
953                 err = far_enum_dir(f, dnobj, DIR_FROM, ctx);
954         }
955 out:
956         if (from)
957                 dnode_rele(from, FTAG);
958         if (to)
959                 dnode_rele(to, FTAG);

961         return (err);
962 }

964 int
965 far_find_entry(far_t *f, uint64_t dirobj, uint64_t dnobj,
966     far_which_t which, char **name)
967 {
968         zap_cursor_t zc;
969         zap_attribute_t *za;
970         int err;
971         uint64_t mask = ZFS_DIRENT_OBJ(-1ULL);
972         struct objset *os;

974         if (which == FAR_OLD) {
975                 os = f->f_fromsnap;
976                 if (!os)
977                         return (ENOENT);
978         } else if (which == FAR_NEW) {
979                 os = f->f_tosnap;
980         } else {
981                 return (EINVAL);
```



```
982         }

984         if (name)
985                 *name = NULL;
986         za = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);
987         for (zap_cursor_init(&zc, os, dirobj);
988             (err = zap_cursor_retrieve(&zc, za)) == 0;
989             zap_cursor_advance(&zc)) {
990                 if ((za->za_first_integer & mask) == (dnobj & mask)) {
991                         if (name)
992                                 *name = za->za_name;
993                         break;
994                 }
995         }
996         zap_cursor_fini(&zc);
997         return (err);
998 }

1000 void
1001 far_free_name(char *name)
1002 {
1003         zap_attribute_t *za;

1005         if (!name)
1006                 return;

1008         za = (zap_attribute_t *)(name - offsetof(zap_attribute_t, za_name));
1009         kmem_free(za, sizeof (*za));
1010 }

1012 int
1013 far_lookup_entry(far_t *f, uint64_t dirobj, char *name,
1014     far_which_t which, uint64_t *dnobj)
1015 {
1016         struct objset *osp;
1017         int ret;

1019         if (which == FAR_OLD) {
1020                 osp = f->f_fromsnap;
1021                 if (!osp)
1022                         return (ENOENT);
1023         } else if (which == FAR_NEW) {
1024                 osp = f->f_tosnap;
1025         } else {
1026                 return (EINVAL);
1027         }

1029         ret =  zap_lookup(osp, dirobj, name, sizeof (*dnobj), 1, dnobj);
1030         if (ret)
1031                 return (ret);
1032         *dnobj = ZFS_DIRENT_OBJ(*dnobj);

1034         return (0);
1035 }

1037 int
1038 far_write(far_t *f, const uint8_t *data, int len)
1039 {
1040         ssize_t resid; /* have to get resid to get detailed errno */
1041         int err;

1043         err = vn_rdwr(UIO_WRITE, f->f_vp, (caddr_t)data,
1044             len, 0, UIO_SYSSPACE, FAPPEND, RLIM64_INFINITY, CRED(), &resid);
1045         *f->f_offp += len;

1047         return (err);
```

```
}

int
far_get_uuid(far_t *f, far_which_t which, uint8_t data[16])
{
        if (which == FAR_OLD && !f->f_fromds)
                return (ENOENT);

        LE_OUT64(data, f->f_tods->ds_dir->dd_pool->dp_spa->spa_config_guid);
        if (which == FAR_OLD) {
                LE_OUT64(data + 8, f->f_fromds->ds_phys->ds_guid);
        } else {
                LE_OUT64(data + 8, f->f_tods->ds_phys->ds_guid);
        }
        return (0);
}

int
far_get_ctransid(far_t *f, far_which_t which, uint64_t *ctransid)
{
        if (which == FAR_OLD && !f->f_fromds)
                return (ENOENT);

        if (which == FAR_OLD)
                *ctransid = f->f_fromds->ds_phys->ds_creation_txg;
        else
                *ctransid = f->f_tods->ds_phys->ds_creation_txg;
        return (0);
}

int
far_get_snapname(far_t *f, far_which_t which,
    char **name, int *len)
{
        dsl_dataset_t *ds;

        if (which == FAR_OLD && !f->f_fromds)
                return (ENOENT);

        if (which == FAR_OLD)
                ds = f->f_fromds;
        else
                ds = f->f_tods;

        *len = dsl_dataset_namelen(ds) + 1;
        *name = kmem_alloc(*len, KM_SLEEP);
        dsl_dataset_name(ds, *name);
        return (0);
}

int
far_read_symlink(far_t *f, uint64_t dnobj, far_which_t which,
    char **target, int *plen)
{
        int err;
        int ret;
        sa_handle_t *hdl = NULL;
        dmu_buf_t *db;
        objset_t *osp;
        dmu_object_info_t doi;
        sa_attr_type_t *sa_table;

        if (which == FAR_OLD) {
                osp = f->f_fromsnap;
                if (!osp)
                        return (EINVAL);
```



```
                sa_table = f->f_from_sa_table;
        } else if (which == FAR_NEW) {
                osp = f->f_tosnap;
                sa_table = f->f_to_sa_table;
        } else {
                return (EINVAL);
        }

        err = far_grab_sa_handle(osp, dnobj, &hdl, &db, FTAG);
        if (err)
                return (err);

        dmu_object_info_from_db(db, &doi);
        if (doi.doi_bonus_type == DMU_OT_SA) {
                int len;

                ret = sa_size(hdl, sa_table[ZPL_SYMLINK], &len);
                if (ret)
                        goto out;
                *target = kmem_alloc(len + 1, KM_SLEEP);
                *plen = len;
                (*target)[len] = 0;
                ret = sa_lookup(hdl, sa_table[ZPL_SYMLINK], *target, len + 1);
                if (ret)
                        kmem_free(*target, len + 1);
        } else {
                /*
                 * TODO read target from file data, the old way
                 * see zfs_readlink
                 */
                ret = EINVAL;
        }

out:
        far_release_sa_handle(hdl, db, FTAG);

        return (ret);
}

int
far_send(objset_t *tosnap, objset_t *fromsnap, int outfd, vnode_t *vp,
    offset_t *off)
{
        dsl_dataset_t *ds;
        dsl_dataset_t *fromds = NULL;
        int err = 0;
        far_t *f;
        arc_buf_t *buf = NULL;
        uint32_t flags;
        objset_phys_t *osp = NULL;
        int i;
        zbookmark_t zb;

        memset(&f, 0, sizeof (f));
        ds = tosnap->os_dsl_dataset;
        if (fromsnap)
                fromds = fromsnap->os_dsl_dataset;

        /* make certain we are looking at snapshots */
        if (!dsl_dataset_is_snapshot(ds) ||
            (fromds && !dsl_dataset_is_snapshot(fromds)))
                return (EINVAL);

        /* fromsnap must be earlier and from the same lineage as tosnap */
        if (fromds) {
                if (fromds->ds_phys->ds_creation_txg >=
```

```
1180                    ds->ds_phys->ds_creation_txg)
1181                        return (EXDEV);
1182
1183                if (fromds->ds_dir != ds->ds_dir)
1184                        return (EXDEV);
1185
1186                /*
1187                 * read root dnode from from-dataset
1188                 */
1189                flags = ARC_WAIT;
1190                SET_BOOKMARK(&zb, fromds->ds_object, ZB_ROOT_OBJECT,
1191                    ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
1192                err = dsl_read_nolock(NULL, fromds->ds_dir->dd_pool->dp_spa,
1193                    &fromds->ds_phys->ds_bp, arc_getbuf_func, &buf,
1194                    ZIO_PRIORITY_ASYNC_READ, ZIO_FLAG_CANFAIL, &flags, &zb);
1195                if (err)
1196                        return (err);
1197                osp = buf->b_data;
1198        }
1199        f = kmem_zalloc(sizeof (far_t), KM_SLEEP);
1200        f->f_vp = vp;
1201        f->f_offp = off;
1202        f->f_err = 0;
1203        f->f_fromds = fromds;
1204        f->f_tods = ds;
1205        f->f_fromsnap = fromsnap;
1206        f->f_tosnap = tosnap;
1207        f->f_dmu_sendarg.dsa_off = off;
1208        f->f_dmu_sendarg.dsa_outfd = outfd;
1209        f->f_dmu_sendarg.dsa_proc = curproc;
1210        f->f_dnp = osp ? &osp->os_meta_dnode : NULL;
1211        mutex_enter(&ds->ds_sendstream_lock);
1212        list_insert_head(&ds->ds_sendstreams, &f->f_dmu_sendarg);
1213        mutex_exit(&ds->ds_sendstream_lock);
1214
1215        if (fromds) {
1216                f->f_fromtxg = fromds->ds_phys->ds_creation_txg;
1217                f->f_bl = kmem_zalloc(sizeof (blklevel_t) *
1218                    f->f_dnp->dn_nlevels, KM_SLEEP);
1219                for (i = 0; i < f->f_dnp->dn_nlevels; ++i)
1220                        f->f_bl[i].bl_blk = -1;
1221                i = f->f_dnp->dn_nlevels - 1;
1222                f->f_bl[i].bl_nslots = f->f_dnp->dn_nblkptr;
1223                f->f_bl[i].bl_bp = &f->f_dnp->dn_blkptr[0];
1224                f->f_bl[i].bl_blk = 0;
1225
1226                err = far_sa_setup(fromsnap, &f->f_from_sa_table);
1227                if (err)
1228                        goto out;
1229        }
1230        err = far_sa_setup(tosnap, &f->f_to_sa_table);
1231        if (err)
1232                goto out;
1233
1234        err = zap_lookup(tosnap, MASTER_NODE_OBJ, ZFS_SHARES_DIR, 8, 1,
1235            &f->f_shares_dir);
1236        if (err && err != ENOENT)
1237                goto out;
1238
1239        err = far_start(f, &f->f_ops);
1240        if (err)
1241                goto out;
1242
1243
1244
1245        err = traverse_dataset(ds, f->f_fromtxg,
```



```
1246            TRAVERSE_PRE | TRAVERSE_PREFETCH_METADATA, far_cb, f);
1247        if (err) {
1248                far_abort(f);
1249                goto out;
1250        }
1251        err = far_start2(f, &f->f_ops);
1252        if (err) {
1253                goto out;
1254        }
1255        err = traverse_dataset(ds, f->f_fromtxg,
1256            TRAVERSE_PRE | TRAVERSE_PREFETCH_METADATA, far_cb, f);
1257        if (err) {
1258                far_abort(f);
1259                goto out;
1260        }
1261
1262        err = far_end(f);
1263        if (err)
1264                goto out;
1265
1266 out:
1267        if (fromds) {
1268                for (i = 0; i < f->f_dnp->dn_nlevels - 1; ++i) {
1269                        blklevel_t *b = f->f_bl + i;
1270                        if (b->bl_buf)
1271                                arc_buf_remove_ref(b->bl_buf, &b->bl_buf);
1272                }
1273                kmem_free(f->f_bl, sizeof (blklevel_t) * f->f_dnp->dn_nlevels);
1274        }
1275
1276        if (buf)
1277                arc_buf_remove_ref(buf, &buf);
1278
1279        mutex_enter(&ds->ds_sendstream_lock);
1280        list_remove(&ds->ds_sendstreams, &f->f_dmu_sendarg);
1281        mutex_exit(&ds->ds_sendstream_lock);
1282        kmem_free(f, sizeof (far_t));
1283
1284        return (err);
1285 }
1286 #endif /* ! codereview */
```

```
**********************************************************
    3366 Fri Oct 26 17:09:24 2012
new/usr/src/uts/common/fs/zfs/far_count.c
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * 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) 2012 STRATO AG. All rights reserved.
 */
#include <sys/zfs_context.h>
#include <sys/errno.h>
#include <sys/far_impl.h>

static void
far_count_value_free(mod_hash_val_t val)
{
        far_count_elem_t *fce = val;
        kmem_free(fce, sizeof (*fce));
}

static void
far_count_key_free(mod_hash_key_t key)
{
        kmem_free(key, sizeof (uint64_t));
}

static int
far_count_cmp(const void *aa, const void *bb)
{
        const far_count_elem_t *a = aa;
        const far_count_elem_t *b = bb;

        if (a->fce_ino < b->fce_ino)
                return -1;
        if (a->fce_ino > b->fce_ino)
                return 1;
        return 0;
}

int
far_add_count(far_counter_t *fc, uint64_t ino, uint64_t inc,
    uint64_t aux, uint64_t *new_count, uint64_t *old_aux)
{
```



```
        far_count_elem_t *fce;
        far_count_elem_t e = { .fce_ino = ino };
        avl_index_t where;

        fce = avl_find(&fc->fc_avl, &e, &where);

        if (!fce) {
                fce = kmem_alloc(sizeof (*fce), KM_SLEEP);
                fce->fce_count = 0;
                fce->fce_aux = 0;
                fce->fce_ino = ino;
                avl_insert(&fc->fc_avl, fce, where);
        }

        if (old_aux) {
                *old_aux = fce->fce_aux;
                fce->fce_aux = aux;
        }
        fce->fce_count += inc;

        if (new_count)
                *new_count = fce->fce_count;

        return (0);
}

int
far_get_count(far_counter_t *fc, uint64_t ino, uint64_t *count, uint64_t *aux)
{
        far_count_elem_t *fce;
        far_count_elem_t e = { .fce_ino = ino };

        fce = avl_find(&fc->fc_avl, &e, NULL);
        if (!fce) {
                if (count)
                        *count = 0;
                if (aux)
                        *aux = 0;
                return (ENOENT);
        } else {
                if (count)
                        *count = fce->fce_count;
                if (aux)
                        *aux = fce->fce_aux;
                return (0);
        }
}

void
far_free_count(far_counter_t *fc, uint64_t ino)
{
        far_count_elem_t *fce;
        far_count_elem_t e = { .fce_ino = ino };

        fce = avl_find(&fc->fc_avl, &e, NULL);
        if (!fce)
                return;
        avl_remove(&fc->fc_avl, fce);
        kmem_free(fce, sizeof (*fce));
}

int
far_count_init(far_counter_t *fc, char *name)
{
        avl_create(&fc->fc_avl, far_count_cmp, sizeof (far_count_elem_t),
            offsetof(far_count_elem_t, fce_avl_node));
```

```
124         fc->fc_name = name;

126         return 0;
127 }

129 int
130 far_count_fini(far_counter_t *fc)
131 {
132         far_count_elem_t *fce;
133         int ret = 0;

135         while ((fce = avl_first(&fc->fc_avl))) {
136                 /*
137                  * a count of zero might be left over if a file had > 1 links
138                  * and be replaced by a file with > 1 link. see test 041.034
139                  */
140                 if (fce->fce_count != 0) {
141                         cmn_err(CE_NOTE, "far_assert_count_empty: %s ino %"
142                             PRIu64 " count %" PRIu64"\n", fc->fc_name,
143                             fce->fce_ino, fce->fce_count);
144                         ++ret;
145                 }
146                 avl_remove(&fc->fc_avl, fce);
147                 kmem_free(fce, sizeof (*fce));
148         }
149         avl_destroy(&fc->fc_avl);

151         return (ret);
152 }
153 #endif /* ! codereview */
```

```
**********************************************************
    3884 Fri Oct 26 17:09:24 2012
new/usr/src/uts/common/fs/zfs/far_crc32c.c
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * 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
 */
/*
 * The crc32c algorithms are taken from sctp_crc32 implementation
 * common/inet/sctp_crc32.{c,h}, which in turn were taken from nxge_fflp_hash.c
 */

#include <sys/zfs_context.h>
#include <sys/far_crc32c.h>

static void far_crc32c_word(uint32_t *crcptr, const uint32_t *buf, int len);

/*
 * Fast CRC32C calculation algorithm.  The basic idea is to look at it
 * four bytes (one word) at a time, using four tables.  The
 * standard algorithm in RFC 3309 uses one table.
 */

#define CRC_32C_POLY 0x1EDC6F41L

/* The four CRC32c tables. */
static uint32_t crc32c_tab[4][256];
static int initialized;


static uint32_t
reflect_32(uint32_t b)
{
        int i;
        uint32_t rw = 0;

        for (i = 0; i < 32; i++) {
                if (b & 1) {
                        rw |= 1 << (31 - i);
                }
                b >>= 1;
        }
        return (rw);
}
```



```
#ifdef _BIG_ENDIAN
static uint32_t
flip32(uint32_t w)
{
        return (((w >> 24) | ((w >> 8) & 0xff00) |
            ((w << 8) & 0xff0000) | (w << 24)));
}
#endif

/*
 * Initialize the crc32c tables.
 */

void
far_crc32c_init(void)
{
        uint32_t index, bit, byte, crc;

        for (index = 0; index < 256; index++) {
                crc = reflect_32(index);
                for (byte = 0; byte < 4; byte++) {
                        for (bit = 0; bit < 8; bit++) {
                                crc = (crc & 0x80000000) ?
                                    (crc << 1) ^ CRC_32C_POLY : crc << 1;
                        }
#ifdef _BIG_ENDIAN
                        crc32c_tab[3 - byte][index] = flip32(reflect_32(crc));
#else
                        crc32c_tab[byte][index] = reflect_32(crc);
#endif
                }
        }
}

/*
 * Lookup  the crc32c for a byte stream
 */
static void
far_crc32c_byte(uint32_t *crcptr, const uint8_t *buf, int len)
{
        uint32_t crc;
        int i;

        crc = *crcptr;
        for (i = 0; i < len; i++) {
#ifdef _BIG_ENDIAN
                crc = (crc << 8) ^ crc32c_tab[3][buf[i] ^ (crc >> 24)];
#else
                crc = (crc >> 8) ^ crc32c_tab[0][buf[i] ^ (crc & 0xff)];
#endif
        }
        *crcptr = crc;
}

/*
 * Lookup  the crc32c for a 32 bit word stream
 * Lookup is done fro the 4 bytes in parallel
 * from the tables computed earlier
 *
 */
static void
far_crc32c_word(uint32_t *crcptr, const uint32_t *buf, int len)
{
        uint32_t w, crc;
        int i;
```

```
125         crc = *crcptr;
126         for (i = 0; i < len; i++) {
127                 w = crc ^ buf[i];
128                 crc = crc32c_tab[0][w >> 24] ^
129                     crc32c_tab[1][(w >> 16) & 0xff] ^
130                     crc32c_tab[2][(w >> 8) & 0xff] ^
131                     crc32c_tab[3][w & 0xff];
132         }
133         *crcptr = crc;
134 }

136 /*
137  * Lookup  the crc32c for a stream of bytes
138  *
139  * Tries to lookup the CRC on 4 byte words
140  * If the buffer is not 4 byte aligned, first compute
141  * with byte lookup until aligned. Then compute crc
142  * for each 4 bytes. If there are bytes left at the end of
143  * the buffer, then perform a byte lookup for the remaining bytes
144  *
145  *
146  */
147 uint32_t
148 far_crc32c(uint32_t crc32, const uint8_t *buf, int len)
149 {
150         int rem;

152         if (!initialized) {
153                 far_crc32c_init();
154                 initialized = 1;
155         }

157         rem = 4 - (((uintptr_t)buf) & 3);
158         if (rem != 0) {
159                 if (len < rem) {
160                         rem = len;
161                 }
162                 far_crc32c_byte(&crc32, buf, rem);
163                 buf = buf + rem;
164                 len = len - rem;
165         }
166         if (len > 3) {
167                 far_crc32c_word(&crc32, (const uint32_t *) buf, len / 4);
168         }
169         rem = len & 3;
170         if (rem != 0) {
171                 far_crc32c_byte(&crc32, buf + len - rem, rem);
172         }
173         return (crc32);
174 }
175 #endif /* ! codereview */
```

```
**********************************************************
   11286 Fri Oct 26 17:09:24 2012
new/usr/src/uts/common/fs/zfs/far_pass1.c
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * 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) 2012 STRATO AG. All rights reserved.
 */
#include <sys/zfs_context.h>
#include <sys/errno.h>
#include <sys/stat.h>
#include <sys/far.h>
#include <sys/far_impl.h>

struct far_enum {
        far_t *fe_far;
        uint64_t fe_parent_ino;
        far_dirent_t *fe_dirent_chain;
};

struct far_file {
        far_t *ff_far;
        uint64_t ff_len;
        uint64_t ff_last_byte;
        uint64_t ff_ino;
        far_path_t *ff_path;
        far_dirent_t *ff_dirent;
};

static int far_file_data_pass1(void *far_filep, void *data, uint64_t off,
    uint64_t len);
static int far_dirent_add_pass1(void *far_enump, char *name, uint64_t ino);
static int far_dirent_mod_pass1(void *far_enump, char *name,
    uint64_t ino_old, uint64_t ino_new);
static int far_dir_add_pass1(far_t *f, uint64_t ino);
static int far_mod_pass1(far_t *f, uint64_t ino);

static far_ops_t _ops = {
        .far_dir_add = far_dir_add_pass1,
        .far_dir_mod = far_mod_pass1,
        .far_dirent_add = far_dirent_add_pass1,
        .far_dirent_mod = far_dirent_mod_pass1,
```



```
        .far_file_mod = far_mod_pass1,
        .far_file_data = far_file_data_pass1
};

static int far_file_add_genchange(far_t *f, uint64_t ino);

int
far_start(far_t *f, far_ops_t **ops)
{
        int ret;

        f->f_pass = PASS_LINK;
        far_count_init(&f->f_link_add_cnt, "link_add_cnt");
        far_count_init(&f->f_del_dir_cnt, "del_dir_cnt");
        far_count_init(&f->f_put_back_cnt, "put_back_cnt");
        far_send_init(f);

        *ops = &_ops;

        ret = far_send_start(f);
        if (ret) {
                far_abort(f);
                return (ret);
        }

        return (0);
}

static int
enum_dir(far_t *f, uint64_t ino, far_dirent_t *chain)
{
        struct far_enum fe = {
                .fe_far = f,
                .fe_parent_ino = ino,
                .fe_dirent_chain = chain
        };

        return (far_dir_contents(f, ino, &fe));
}

static int
far_file_data_pass1(void *far_filep, void *data, uint64_t off, uint64_t len)
{
        struct far_file *ff = far_filep;

        if (off + len > ff->ff_len)
                len = ff->ff_len - off;

        ff->ff_last_byte = off + len;

        return far_send_file_data(ff->ff_far, &ff->ff_path,
            ff->ff_dirent, ff->ff_ino, off, len, data);
}

static int
dirent_add_dir(far_t *f, far_dirent_t *dirent, uint64_t ino, int exists)
{
        far_info_t si_old;
        far_info_t si_new;
        int ret;

        ret = far_get_info(f, ino, FAR_OLD, &si_old,
                        FI_ATTR_PARENT | FI_ATTR_GEN | FI_ATTR_MODE);
        if (ret && ret != ENOENT)
                return (ret);
```

```
124         if (ret == 0) {
125                 ret = far_get_info(f, ino, FAR_NEW, &si_new,
126                                    FI_ATTR_GEN | FI_ATTR_MODE);
127                 if (ret)
128                         return (ret);
129
130                 if (si_old.si_gen == si_new.si_gen ||
131                     (S_ISDIR(si_new.si_mode) && S_ISDIR(si_old.si_mode)))
132                         return far_send_rename(f, dirent, ino,
133                                                si_old.si_parent, exists);
134                 return (0);
135         }
136
137         if (ino > f->f_current_ino)
138                 return (0);
139
140         /* dir is new */
141         ret = far_send_mkdir(f, dirent, ino, exists);
142         if (ret)
143                 return (ret);
144
145         return (enum_dir(f, ino, dirent));
146 }
147
148 int
149 far_dirent_add_file(far_t *f, far_dirent_t *dirent,
150     uint64_t ino, uint64_t mode, int exists)
151 {
152         far_info_t si_old;
153         far_info_t si_new;
154         int ret;
155         far_path_t *far_path;
156         uint64_t new_count;
157         uint64_t old_aux;
158
159         ret = far_get_info(f, ino, FAR_OLD, &si_old,
160                            FI_ATTR_GEN | FI_ATTR_PARENT);
161         if (ret && ret != ENOENT)
162                 return (ret);
163
164         if (ret == 0) {
165                 ret = far_get_info(f, ino, FAR_NEW, &si_new,
166                                    FI_ATTR_GEN);
167                 if (ret)
168                         return (ret);
169
170                 if (si_old.si_gen == si_new.si_gen)
171                         return far_send_link(f, dirent, ino, si_old.si_parent,
172                             FAR_OLD, exists);
173         }
174
175         /* file is new */
176         ret = far_add_count(&f->f_link_add_cnt, ino, 1, dirent->fd_parent_ino,
177             &new_count, &old_aux);
178         if (ret)
179                 return (ret);
180
181         if (new_count == 1)
182                 ret = far_send_create_file(f, dirent, ino,
183                                            exists, &far_path);
184         else
185                 ret = far_send_link(f, dirent, ino, old_aux, FAR_NEW, exists);
186         if (ret)
187                 return (ret);
188
189         ret = far_get_info(f, ino, FAR_NEW, &si_new,
```



```
190                            FI_ATTR_SIZE | FI_ATTR_LINKS);
191         ASSERT(ret == 0);
192         if (new_count == 1 && S_ISREG(mode)) {
193                 struct far_file ff;
194
195                 ff.ff_ino = ino;
196                 ff.ff_len = si_new.si_size;
197                 ff.ff_far = f;
198                 ff.ff_path = far_path;
199                 ff.ff_dirent = dirent;
200                 ff.ff_last_byte = 0;
201                 ret = far_file_contents(f, ino, &ff);
202                 far_path_free(ff.ff_path);
203                 if (ret)
204                         return (ret);
205                 if (ff.ff_last_byte != si_new.si_size) {
206                         /* sparse end */
207                         ret = far_send_truncate(f, NULL, ino, si_new.si_size);
208                         if (ret)
209                                 return (ret);
210                 }
211         }
212         if (new_count == si_new.si_nlinks)
213                 far_free_count(&f->f_link_add_cnt, ino);
214
215         return (0);
216 }
217
218 static int
219 dirent_add(far_t *f, far_dirent_t *dirent, uint64_t ino, int exists)
220 {
221         far_info_t si;
222         int ret;
223
224         ret = far_get_info(f, ino, FAR_NEW, &si, FI_ATTR_MODE);
225         if (ret)
226                 return (ret);
227
228         if (S_ISDIR(si.si_mode)) {
229                 return (dirent_add_dir(f, dirent, ino, exists));
230         } else {
231                 return (far_dirent_add_file(f, dirent, ino, si.si_mode,
232                     exists));
233         }
234 }
235
236 static int
237 far_dirent_add_pass1(void *far_enump, char *name, uint64_t ino)
238 {
239         struct far_enum *fe = far_enump;
240         far_dirent_t dirent = {
241                 .fd_name = name,
242                 .fd_parent_ino = fe->fe_parent_ino,
243                 .fd_prev = fe->fe_dirent_chain,
244         };
245
246         return (dirent_add(fe->fe_far, &dirent, ino, 0));
247 }
248
249 static int
250 far_dirent_mod_pass1(void *far_enump, char *name,
251     uint64_t ino_old, uint64_t ino_new)
252 {
253         struct far_enum *fe = far_enump;
254         far_dirent_t dirent = {
255                 .fd_name = name,
```

```
256                 .fd_parent_ino = fe->fe_parent_ino,
257                 .fd_prev = fe->fe_dirent_chain,
258         };

260         return (dirent_add(fe->fe_far, &dirent, ino_new, 1));
261 }

263 static int
264 far_file_add_genchange(far_t *f, uint64_t ino)
265 {
266         int ret;
267         char *name = NULL;

269         f->f_current_ino = ino;
270         f->f_current_path = NULL;

272         /*
273          * only called when generation has changed. TODO: move to own
274          * function
275          */
276         far_info_t si_old;
277         far_info_t si_new;
278         int same_name = 0;

280         ret = far_get_info(f, ino, FAR_OLD, &si_old, FI_ATTR_MODE |
281                            FI_ATTR_LINKS | FI_ATTR_PARENT);
282         if (ret)
283                 return (ret);
284         ret = far_get_info(f, ino, FAR_NEW, &si_new, FI_ATTR_MODE |
285                            FI_ATTR_LINKS | FI_ATTR_PARENT);
286         if (ret)
287                 return (ret);

289         if (si_old.si_nlinks > 1 && si_new.si_nlinks > 1)
290                 return far_add_count(&f->f_link_add_cnt, ino, 0, 0, NULL,
291                     NULL);

293         if (S_ISDIR(si_old.si_mode))
294                 return (0);

296         far_which_t from;
297         far_which_t to;
298         uint64_t new_ino;
299         uint64_t parent = si_new.si_parent;

301         if (si_old.si_nlinks == 1) {
302                 from = FAR_OLD;
303                 to = FAR_NEW;
304                 parent = si_old.si_parent;
305         } else if (si_old.si_nlinks > 1 && si_new.si_nlinks == 1) {
306                 from = FAR_NEW;
307                 to = FAR_OLD;
308                 parent = si_new.si_parent;
309         } else {
310                 return (EINVAL);
311         }

313         ret = far_find_entry(f, parent, ino, from, &name);
314         if (ret)
315                 return (ret);

317         ret = far_lookup_entry(f, parent, name, to, &new_ino);
318         if (ret && ret != ENOENT)
319                 goto out;
320         if (ret == 0 && new_ino == ino)
321                 same_name = 1;
```



```
323         if ((si_old.si_nlinks == 1 || si_new.si_nlinks == 1) && !same_name) {
324                 ret = 0;
325                 goto out;
326         }

328         far_dirent_t dirent = {
329                 .fd_parent_ino = parent,
330                 .fd_name = name,
331                 .fd_prev = NULL
332         };

334         ret = far_send_unlink(f, &dirent, ino);
335         if (ret)
336                 goto out;
337         ret = far_dirent_add_file(f, &dirent, ino, si_new.si_mode, 0);

339 out:
340         far_free_name(name);
341         return (ret);
342 }

344 static int
345 far_dir_add_pass1(far_t *f, uint64_t ino)
346 {
347         int ret;
348         uint64_t parent;
349         uint64_t first_parent = FAR_NO_INO;
350         far_info_t si;
351         far_dirent_t dirent;
352         int same_name = 0;
353         char *name = NULL;

355         f->f_current_ino = ino;
356         f->f_current_path = NULL;

358         parent = ino;
359         while (1) {
360                 /* the new parent must exist, otherwise the fs is wrong */
361                 ret = far_get_info(f, parent, FAR_NEW, &si,
362                                    FI_ATTR_PARENT);
363                 if (ret)
364                         return (ret);
365                 if (first_parent == FAR_NO_INO)
366                         first_parent = si.si_parent;

368                 ret = far_get_info(f, parent, FAR_OLD, &si, 0);
369                 if (ret && ret != ENOENT)
370                         return (ret);
371                 if (ret != ENOENT)
372                         break;

374                 /*
375                  * this check is only needed for a full send, on all
376                  * incrementals the parent already exists and it breaks out
377                  * above
378                  */
379                 if (parent == si.si_parent) {
380                         first_parent = FAR_NO_INO;
381                         break;
382                 }
383                 parent = si.si_parent;

385                 if (parent > ino)
386                         return (0);
387         }
```

```
389         /*
390          * check for same-name
391          */
392         if (first_parent != FAR_NO_INO) {
393                 ret = far_get_info(f, first_parent, FAR_OLD, &si,
394                     FI_ATTR_MODE);
395                 if (ret && ret != ENOENT)
396                         return (ret);
397                 if (ret == 0 && S_ISDIR(si.si_mode)) {
398                         uint64_t old_ino;

400                         ret = far_find_entry(f, first_parent, ino,
401                             FAR_NEW, &name);
402                         if (ret)
403                                 return (ret);

405                         ret = far_lookup_entry(f, first_parent, name,
406                             FAR_OLD, &old_ino);
407                         if (ret && ret != ENOENT) {
408                                 goto out;
409                         }
410                         if (ret == 0) {
411                                 same_name = 1;
412                                 dirent.fd_name = name;
413                                 dirent.fd_parent_ino = first_parent;
414                                 dirent.fd_prev = NULL;
415                                 if (old_ino == ino) {
416                                         ret = far_add_count(&f->f_link_add_cnt,
417                                             ino, 0, 0, NULL, NULL);
418                                         if (ret)
419                                                 goto out;
420                                 }
421                         }
422                 }
423         }
424         /* dir is new */
425         ret = far_send_mkdir(f, same_name ? &dirent : NULL, ino, same_name);
426         if (ret)
427                 goto out;

429         ret = enum_dir(f, ino, NULL);

431 out:
432         far_free_name(name);
433         return (ret);
434 }

436 static int
437 far_mod_pass1(far_t *f, uint64_t ino)
438 {
439         far_info_t si_old;
440         far_info_t si_new;
441         int ret;

443         f->f_current_ino = ino;
444         f->f_current_path = NULL;

446         ret = far_get_info(f, ino, FAR_NEW, &si_new, FI_ATTR_SIZE |
447                             FI_ATTR_MODE | FI_ATTR_GEN | FI_ATTR_UID |
448                             FI_ATTR_GID | FI_ATTR_SIZE);
449         if (ret)
450                 return (ret);
451         ret = far_get_info(f, ino, FAR_OLD, &si_old, FI_ATTR_GEN |
452                             FI_ATTR_MODE | FI_ATTR_UID |
453                             FI_ATTR_GID | FI_ATTR_SIZE);
```



```
454         if (ret)
455                 return (ret);

457         if (!(S_ISDIR(si_old.si_mode) && S_ISDIR(si_new.si_mode)) &&
458             si_new.si_gen != si_old.si_gen) {
459                 if (S_ISDIR(si_new.si_mode))
460                         return (far_dir_add_pass1(f, ino));
461                 else
462                         return (far_file_add_genchange(f, ino));
463         }

465         if (S_ISDIR(si_new.si_mode)) {
466                 ret = enum_dir(f, ino, NULL);
467                 if (ret)
468                         return (ret);
469         }

471         if (S_ISREG(si_new.si_mode)) {
472                 struct far_file ff;
473                 ff.ff_ino = ino;
474                 ff.ff_len = si_new.si_size;
475                 ff.ff_far = f;
476                 ff.ff_path = NULL;
477                 ff.ff_dirent = NULL;
478                 ff.ff_last_byte = 0;

480                 ret = far_file_contents(f, ino, &ff);
481                 far_path_free(ff.ff_path);
482                 if (ret)
483                         return (ret);
484                 if (si_new.si_size < si_old.si_size ||
485                     (si_new.si_size != si_old.si_size &&
486                     si_new.si_size != ff.ff_last_byte)) {
487                         ret = far_send_truncate(f, NULL, ino, si_new.si_size);
488                         if (ret)
489                                 return (ret);
490                 }
491         }

493         if (si_old.si_uid != si_new.si_uid || si_old.si_gid != si_new.si_gid) {
494                 ret = far_send_chown(f, NULL, ino, si_new.si_uid,
495                     si_new.si_gid);
496                 if (ret)
497                         return (ret);
498         }
499         if (si_old.si_mode != si_new.si_mode) {
500                 ret = far_send_chmod(f, NULL, ino, si_new.si_mode);
501                 if (ret)
502                         return (ret);
503         }

505         return (ret);
506 }
507 #endif /* ! codereview */
```

```
**********************************************************
   11476 Fri Oct 26 17:09:24 2012
new/usr/src/uts/common/fs/zfs/far_pass2.c
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * 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) 2012 STRATO AG. All rights reserved.
 */
#include <sys/zfs_context.h>
#include <sys/errno.h>
#include <sys/stat.h>
#include <sys/far.h>
#include <sys/far_impl.h>

struct far_enum {
        far_t *fe_far;
        uint64_t fe_parent_ino;
        uint64_t fe_del_dir_cnt;
        uint64_t fe_put_back_cnt;
        far_dirent_t *fe_dirent_chain;
};

struct far_file {
        far_t *ff_far;
        uint64_t ff_len;
        uint64_t ff_last_byte;
        uint64_t ff_ino;
        far_path_t *ff_path;
        far_dirent_t *ff_dirent;
};

static int far_dirent_del_pass2(void *far_enump, char *name, uint64_t ino);
static int far_dirent_mod_pass2(void *far_enump, char *name,
    uint64_t ino_old, uint64_t ino_new);
static int far_dirent_unmod_pass2(void *far_enump, char *name, uint64_t ino);
static int far_dir_del_pass2(far_t *f, uint64_t ino);
static int far_add_pass2(far_t *f, uint64_t ino);
static int far_mod_pass2(far_t *f, uint64_t ino);

far_ops_t _ops = {
        .far_dirent_del = far_dirent_del_pass2,
        .far_dirent_mod = far_dirent_mod_pass2,
```



```
        .far_dirent_unmod = far_dirent_unmod_pass2,
        .far_file_add = far_add_pass2,
        .far_file_mod = far_mod_pass2,
        .far_dir_add = far_add_pass2,
        .far_dir_del = far_dir_del_pass2,
        .far_dir_mod = far_mod_pass2
};

int
far_start2(far_t *f, far_ops_t **ops)
{
        f->f_pass = PASS_UNLINK;
        *ops = &_ops;

        return (0);
}

int
far_abort(far_t *f)
{
        far_send_fini(f);

        return (0);
}

int
far_end(far_t *f)
{
        int ret;
        int ret2;

        far_send_end(f);

        ret = far_count_fini(&f->f_link_add_cnt);
        ret += far_count_fini(&f->f_del_dir_cnt);
        ret += far_count_fini(&f->f_put_back_cnt);

        ret2 = far_abort(f);

        return (ret ? EIO : ret2);
}

static int
dir_del(far_t *f, uint64_t ino, uint64_t removed_entries)
{
        far_info_t si;
        uint64_t parent;
        int ret;
        int exists;
        uint64_t new_count;
        uint64_t left;

        while (1) {
                ret = far_get_info(f, ino, FAR_OLD, &si, FI_ATTR_GEN |
                                   FI_ATTR_PARENT | FI_ATTR_NENTRIES);
                if (ret)
                        return (ret);

                if (si.si_parent > f->f_current_ino)
                        return (0);

                if (removed_entries + 2 < si.si_nentries) /* '.' and '..' */
                        return (0);

                far_free_count(&f->f_del_dir_cnt, ino);
                ret = far_send_rmdir(f, NULL, ino);
```

```
124                 if (ret)
125                         return (ret);
126
127                 parent = si.si_parent;
128                 exists = far_get_info(f, parent, FAR_NEW, &si,
129                     FI_ATTR_MODE);
130                 if (exists && exists != ENOENT)
131                         return (exists);
132
133                 ret = far_get_count(&f->f_put_back_cnt, parent, &left, NULL);
134                 if (ret && ret != ENOENT)
135                         return (ret);
136                 if (left > 0) {
137                         ret = far_add_count(&f->f_put_back_cnt, parent, -1, 0,
138                             &left, NULL);
139                         if (ret)
140                                 return (ret);
141                 }
142                 if ((int64_t)left < 0)
143                         return (EINVAL);
144
145                 if (exists == 0 && S_ISDIR(si.si_mode)) {
146                         char *name;
147                         uint64_t new_ino;
148                         far_info_t si_new;
149                         int new;
150
151                         new = far_get_info(f, ino, FAR_NEW, &si_new,
152                                            FI_ATTR_MODE);
153                         if (new && new != ENOENT)
154                                 return (new);
155                         ret = far_find_entry(f, parent, ino, FAR_OLD, &name);
156                         if (ret)
157                                 return (ret);
158
159                         ret = far_lookup_entry(f, parent, name,
160                                                FAR_NEW, &new_ino);
161                         if (ret && ret != ENOENT) {
162                                 far_free_name(name);
163                                 return (ret);
164                         }
165                         if (ret == 0 && new_ino != ino) {
166                                 far_dirent_t dirent = {
167                                         .fd_name = name,
168                                         .fd_parent_ino = parent
169                                 };
170
171                                 ret = far_send_rename_from_tempname(f, &dirent,
172                                     ino, new_ino);
173                                 if (ret) {
174                                         far_free_name(name);
175                                         return (ret);
176                                 }
177                         } else if (ret == 0 && new_ino == ino &&
178                             !S_ISDIR(si_new.si_mode)) {
179                                 far_dirent_t dirent = {
180                                         .fd_name = name,
181                                         .fd_parent_ino = parent
182                                 };
183
184                                 ret = far_dirent_add_file(f, &dirent, ino,
185                                     si_new.si_mode, 0);
186                                 if (ret) {
187                                         far_free_name(name);
188                                         return (ret);
189                                 }
```



```
190                         }
191                         far_free_name(name);
192                 }
193
194                 if (left == 0) {
195                         far_free_count(&f->f_put_back_cnt, parent);
196                         if (exists == 0 && S_ISDIR(si.si_mode)) {
197                                 ret = far_send_mtime_update(f, NULL, parent);
198                                 if (ret)
199                                         return (ret);
200                         }
201                 }
202
203                 if (exists == 0 && S_ISDIR(si.si_mode))
204                         return (0);
205
206                 /* propagate deletion */
207                 ret = far_add_count(&f->f_del_dir_cnt, parent, 1, 0,
208                     &new_count, NULL);
209                 if (ret)
210                         return (ret);
211
212                 ino = parent;
213                 removed_entries = new_count;
214         }
215 }
216
217 static int
218 enum_dir(far_t *f, uint64_t ino, uint64_t *pput_back_cnt,
219     uint64_t *pdel_dir_cnt)
220 {
221         int ret;
222         struct far_enum fe = {
223                 .fe_far = f,
224                 .fe_parent_ino = ino,
225                 .fe_del_dir_cnt = 0,
226                 .fe_put_back_cnt = 0
227         };
228
229         ret = far_dir_contents(f, ino, &fe);
230         if (ret)
231                 return (ret);
232
233         if (pput_back_cnt)
234                 *pput_back_cnt = fe.fe_put_back_cnt;
235         if (pdel_dir_cnt)
236                 *pdel_dir_cnt = fe.fe_del_dir_cnt;
237
238         if (fe.fe_put_back_cnt) {
239                 ret = far_add_count(&f->f_put_back_cnt, ino,
240                     fe.fe_put_back_cnt, 0, NULL, NULL);
241                 if (ret)
242                         return (ret);
243         }
244
245         return (0);
246 }
247
248 static int
249 dirent_del_file(struct far_enum *fe, far_dirent_t *dirent,
250                 uint64_t ino, uint64_t remains)
251 {
252         int ret;
253
254         ret = far_send_unlink(fe->fe_far, dirent, ino);
255         if (ret)
```

```
256                 return (ret);

258         if (remains != FAR_NO_INO) {
259                 ret = far_send_rename_from_tempname(fe->fe_far, dirent,
260                     ino, remains);
261                 if (ret)
262                         return (ret);
263         }

265         fe->fe_del_dir_cnt++;

267         return (0);
268 }

270 static int
271 dirent_del_dir(struct far_enum *fe, far_dirent_t *dirent, uint64_t ino,
272     uint64_t remains)
273 {
274         int ret;
275         far_info_t si;
276         far_info_t si_old;
277         int new;
278         int old;
279         far_t *f = fe->fe_far;

281         new = far_get_info(f, ino, FAR_NEW, &si, FI_ATTR_GEN |
282                            FI_ATTR_MODE);
283         if (new && new != ENOENT)
284                 return (new);
285         old = far_get_info(f, ino, FAR_OLD, &si_old,
286                            FI_ATTR_NENTRIES | FI_ATTR_GEN |
287                            FI_ATTR_PARENT);
288         if (old)
289                 return (old);

291         /* new == 0 means the dir was renamed, which happened during pass 1 */
292         if (new == ENOENT ||
293             (si.si_gen != si_old.si_gen && !S_ISDIR(si.si_mode))) {
294                 uint64_t cnt;

296                 if (ino > f->f_current_ino) {
297                         ++fe->fe_put_back_cnt;
298                         return (0);
299                 }

301                 ret = far_get_count(&f->f_del_dir_cnt, ino, &cnt,
302                                     NULL);
303                 if (ret && ret != ENOENT)
304                         return (ret);
305                 /* 2 for '.' and '..' */
306                 if (cnt + 2 < si_old.si_nentries) {
307                         ++fe->fe_put_back_cnt;
308                         return (0);
309                 }

311                 far_free_count(&f->f_del_dir_cnt, ino);
312                 ret = far_send_rmdir(f, dirent, ino);
313                 if (ret)
314                         return (ret);
315         }
316         if (remains != FAR_NO_INO) {
317                 ret = far_send_rename_from_tempname(f, dirent, ino, remains);
318                 if (ret)
319                         return (ret);
320         }
321         fe->fe_del_dir_cnt++;
```



```
322         if (new == 0 && si.si_gen != si_old.si_gen && !S_ISDIR(si.si_mode) &&
323             si_old.si_parent == dirent->fd_parent_ino) {
324                 uint64_t parent = si_old.si_parent;
325                 far_info_t sip;
326                 char *name = NULL;

328                 ret = far_get_info(f, parent, FAR_OLD, &sip,
329                                    FI_ATTR_MODE);
330                 if (ret && ret != ENOENT)
331                         return (ret);
332                 if (ret == 0 && S_ISDIR(sip.si_mode)) {
333                         uint64_t old_ino;

335                         ret = far_find_entry(f, parent, ino, FAR_OLD, &name);
336                         if (ret)
337                                 return (ret);

339                         ret = far_lookup_entry(f, parent, name,
340                                                FAR_NEW, &old_ino);
341                         if (ret && ret != ENOENT) {
342                                 far_free_name(name);
343                                 return (ret);
344                         }
345                         if (ret == 0) {
346                                 ret = far_dirent_add_file(f, dirent, ino,
347                                     si.si_mode, 0);
348                                 if (ret)
349                                         return (ret);
350                         }
351                 }
352         }

354         return (0);
355 }

357 static int
358 dirent_del(struct far_enum *fe, far_dirent_t *dirent,
359     uint64_t ino, uint64_t remains)
360 {
361         far_info_t si;
362         int ret;

364         ret = far_get_info(fe->fe_far, ino, FAR_OLD, &si, FI_ATTR_MODE);
365         if (ret)
366                 return (ret);

368         if (S_ISDIR(si.si_mode)) {
369                 return (dirent_del_dir(fe, dirent, ino, remains));
370         } else {
371                 return (dirent_del_file(fe, dirent, ino, remains));
372         }
373 }

375 static int
376 far_dirent_del_pass2(void *far_enump, char *name, uint64_t ino)
377 {
378         struct far_enum *fe = far_enump;
379         far_dirent_t dirent = {
380                 .fd_name = name,
381                 .fd_parent_ino = fe->fe_parent_ino,
382                 .fd_prev = fe->fe_dirent_chain,
383         };

385         return (dirent_del(fe, &dirent, ino, FAR_NO_INO));
386 }
```

```
388 static int
389 far_dirent_mod_pass2(void *far_enump, char *name,
390     uint64_t ino_old, uint64_t ino_new)
391 {
392         struct far_enum *fe = far_enump;
393         far_dirent_t dirent = {
394                 .fd_name = name,
395                 .fd_parent_ino = fe->fe_parent_ino,
396                 .fd_prev = fe->fe_dirent_chain,
397         };
398
399         return (dirent_del(fe, &dirent, ino_old, ino_new));
400 }
401
402 static int
403 far_dirent_unmod_pass2(void *far_enump, char *name, uint64_t ino)
404 {
405         struct far_enum *fe = far_enump;
406         far_t *f = fe->fe_far;
407         int ret;
408         uint64_t cnt;
409         far_info_t si_old;
410         far_info_t si_new;
411         far_dirent_t dirent = {
412                 .fd_name = name,
413                 .fd_parent_ino = fe->fe_parent_ino,
414                 .fd_prev = fe->fe_dirent_chain,
415         };
416
417         ret = far_get_count(&f->f_link_add_cnt, ino, &cnt, NULL);
418         if (ret)
419                 return (ret == ENOENT ? 0 : ret);
420
421         ret = far_get_info(f, ino, FAR_OLD, &si_old, FI_ATTR_MODE);
422         if (ret)
423                 return (ret);
424
425         if (S_ISDIR(si_old.si_mode)) {
426                 return (dirent_del_dir(fe, &dirent, ino, 0));
427         }
428
429         ret = far_get_info(f, ino, FAR_NEW, &si_new, FI_ATTR_MODE);
430         if (ret)
431                 return (ret);
432         ret = far_send_unlink(f, &dirent, ino);
433         if (ret)
434                 return (ret);
435         if (S_ISDIR(si_new.si_mode)) {
436                 far_free_count(&f->f_link_add_cnt, ino);
437                 ret = far_send_rename_from_tempname(f, &dirent, ino, ino);
438                 if (ret)
439                         return (ret);
440                 ret = far_send_mtime_update(f, &dirent, ino);
441         } else {
442                 ret = far_dirent_add_file(f, &dirent, ino, si_new.si_mode, 0);
443         }
444
445         return (ret);
446 }
447
448 static int
449 far_add_pass2(far_t *f, uint64_t ino)
450 {
451         f->f_current_ino = ino;
452         f->f_current_path = NULL;
```



```
454         return (far_send_mtime_update(f, NULL, ino));
455 }
456
457 static int
458 far_dir_del_pass2(far_t *f, uint64_t ino)
459 {
460         uint64_t put_back_cnt;
461         uint64_t del_dir_cnt;
462         uint64_t new_count;
463         int ret;
464
465         f->f_current_ino = ino;
466         f->f_current_path = NULL;
467
468         ret = enum_dir(f, ino, &put_back_cnt, &del_dir_cnt);
469         if (ret)
470                 return (ret);
471
472         ret = far_add_count(&f->f_del_dir_cnt, ino, del_dir_cnt,
473             0, &new_count, NULL);
474         if (ret)
475                 return (ret);
476
477         if (put_back_cnt == 0) {
478                 ret = dir_del(f, ino, new_count);
479                 if (ret)
480                         return (ret);
481         }
482
483         return (0);
484 }
485
486 static int
487 far_mod_pass2(far_t *f, uint64_t ino)
488 {
489         far_info_t si_old;
490         far_info_t si_new;
491         int ret;
492         uint64_t put_back_cnt = 0;
493
494         f->f_current_ino = ino;
495         f->f_current_path = NULL;
496
497         ret = far_get_info(f, ino, FAR_NEW, &si_new, FI_ATTR_SIZE |
498                             FI_ATTR_MODE | FI_ATTR_GEN | FI_ATTR_UID |
499                             FI_ATTR_GID | FI_ATTR_SIZE);
500         if (ret)
501                 return (ret);
502         ret = far_get_info(f, ino, FAR_OLD, &si_old, FI_ATTR_GEN |
503                             FI_ATTR_MODE | FI_ATTR_UID |
504                             FI_ATTR_GID | FI_ATTR_SIZE);
505         if (ret)
506                 return (ret);
507
508         if (!(S_ISDIR(si_old.si_mode) && S_ISDIR(si_new.si_mode)) &&
509             si_new.si_gen != si_old.si_gen) {
510                 if (S_ISDIR(si_old.si_mode)) {
511                         ret = far_dir_del_pass2(f, ino);
512                         if (ret)
513                                 return (ret);
514                 }
515                 return (far_add_pass2(f, ino));
516         }
517
518         if (S_ISDIR(si_new.si_mode)) {
519                 ret = enum_dir(f, ino, &put_back_cnt, NULL);
```

```
520                 if (ret)
521                         return (ret);
522         }

524         if (put_back_cnt)
525                 return (0);

527         return (far_send_mtime_update(f, NULL, ino));
528 }
529 #endif /* ! codereview */
```

```
**********************************************************
   20980 Fri Oct 26 17:09:25 2012
new/usr/src/uts/common/fs/zfs/far_send.c
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * 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) 2012 STRATO AG. All rights reserved.
 */
#include <sys/zfs_context.h>
#include <sys/stat.h>
#include <sys/mkdev.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <sys/far.h>
#include <sys/far_impl.h>
#include <sys/far_crc32c.h>

#define TEMPNAME_PREFIX "far-tempname-"
/* 2^128 needs 39 digits in decimal */
#define TEMPNAME_SIZE (sizeof (TEMPNAME_PREFIX) + 39)

void
far_send_init(far_t *f)
{
        f->f_alloc_len = FAR_SEND_BUF_SIZE;
        f->f_buf = kmem_alloc(f->f_alloc_len, KM_SLEEP);
        f->f_size = 0;
}

void
far_send_fini(far_t *f)
{
        kmem_free(f->f_buf, f->f_alloc_len);
}

static int
far_send_reserve(far_t *f, void **buf, int len)
{
        int res = f->f_alloc_len - f->f_size;
        if (len > res)
                return (-E2BIG);
        *buf = f->f_buf + f->f_size;
```



```
        f->f_size += len;

        return (0);
}

static int
far_send_put(far_t *f, void *buf, int len)
{
        int ret;
        void *p;

        ret = far_send_reserve(f, &p, len);
        if (ret)
                return (ret);

        memcpy(p, buf, len);

        return (0);
}

static int
far_send_put_attr(far_t *f, uint16_t attr, void *buf, int len)
{
        far_attr_header_t hdr;
        int ret;

        LE_OUT16(&hdr.fa_type, attr);
        LE_OUT16(&hdr.fa_len, len);

        ret = far_send_put(f, &hdr, sizeof (hdr));
        if (ret)
                return (ret);
        return (far_send_put(f, buf, len));
}

static int
far_send_reserve_attr(far_t *f, uint16_t attr, void **buf, int len)
{
        far_attr_header_t hdr;
        int ret;

        LE_OUT16(&hdr.fa_type, attr);
        LE_OUT16(&hdr.fa_len, len);

        ret = far_send_put(f, &hdr, sizeof (hdr));
        if (ret)
                return (ret);
        return (far_send_reserve(f, buf, len));
}

static int
far_send_put_u64(far_t *f, uint16_t attr, uint64_t val)
{
        uint64_t v;

        LE_OUT64(&v, val);
        return (far_send_put_attr(f, attr, &v, sizeof (v)));
}

static int
far_send_put_time(far_t *f, uint16_t attr, far_time_t *t)
{
        char buf[12];

        LE_OUT64(buf, t->st_sec);
        LE_OUT32(buf + 8, t->st_nsec);
```

```
125         return (far_send_put_attr(f, attr, buf, sizeof (buf)));
126 }

128 static int
129 far_cmd_start(far_t *f, uint16_t cmd)
130 {
131         far_cmd_header_t ch;

133         memset(&ch, 0, sizeof (ch));
134         LE_OUT16(&ch.fc_cmd, cmd);
135         f->f_size = 0;
136         return (far_send_put(f, &ch, sizeof (ch)));
137 }

139 static int
140 far_cmd_send(far_t *f)
141 {
142         far_cmd_header_t *ch;
143         uint32_t crc;
144         int ret;

146         ch = (far_cmd_header_t *)f->f_buf;
147         LE_OUT32(&ch->fc_len, f->f_size - sizeof (*ch));
148         ch->fc_crc = 0;

150         crc = far_crc32c(0, f->f_buf, f->f_size);
151         LE_OUT32(&ch->fc_crc, crc);

153         ret = far_write(f, f->f_buf, f->f_size);
154         f->f_size = 0;

156         mutex_enter(&f->f_tods->ds_sendstream_lock);
157         *f->f_dmu_sendarg.dsa_off += f->f_size;
158         mutex_exit(&f->f_tods->ds_sendstream_lock);

160         return (ret);
161 }

163 static int
164 far_send_stream_header(far_t *f)
165 {
166         far_stream_header_t header;

168         strcpy(header.fs_magic, FAR_SEND_STREAM_MAGIC);
169         LE_OUT32(&header.fs_version, FAR_SEND_STREAM_VERSION);

171         return (far_write(f, (uint8_t *)&header, sizeof (header)));
172 }

174 static void
175 tempname(uint64_t ino, char *buf, int maxlen)
176 {
177         int l = sizeof (TEMPNAME_PREFIX) - 1;
178         memcpy(buf, TEMPNAME_PREFIX, MIN(maxlen, l));
179         snprintf(buf + l, maxlen - l, "%llu", (long long)ino);
180 }

182 static void
183 path_add_name(far_path_t **fp, char *name, int namelen)
184 {
185         far_path_t *new;

187         new = kmem_alloc(sizeof (*new) + namelen + 1, KM_SLEEP);
188         new->fp_next = *fp;
189         new->fp_len = namelen + 1;
```



```
190         new->fp_total_len = namelen + 1;
191         if (*fp)
192                 new->fp_total_len += (*fp)->fp_total_len;
193         memcpy(new->fp_buf, name, namelen);
194         new->fp_buf[namelen] = '\0';
195         *fp = new;
196 }

198 static void
199 path_copy(far_path_t *fp, char *b)
200 {
201         far_path_t *cur;

203         for (cur = fp; cur; cur = cur->fp_next) {
204                 *b = '/';
205                 memcpy(b + 1, cur->fp_buf, cur->fp_len - 1);
206                 b += cur->fp_len;
207         }
208 }

210 static void
211 path2buf(far_path_t *fp, char **buf, int *buf_len)
212 {
213         char *b;

215         *buf_len = fp->fp_total_len + 1; /* one for the trailing 0-byte */
216         *buf = b = kmem_alloc(*buf_len, KM_SLEEP);

218         path_copy(fp, b);

220         b[*buf_len - 1] = '\0';
221 }

223 static int
224 put_path(far_t *f, uint16_t attr, far_path_t *fp)
225 {
226         int ret;
227         void *p;

229         ret = far_send_reserve_attr(f, attr, &p, fp->fp_total_len);
230         if (ret)
231                 return (ret);
232         path_copy(fp, p);

234         return (0);
235 }

237 void
238 far_path_free(far_path_t *fp)
239 {
240         far_path_t *next;
241         while (fp) {
242                 next = fp->fp_next;
243                 kmem_free(fp, fp->fp_len + sizeof (*fp));
244                 fp = next;
245         }
246 }

248 static int
249 is_ino_run(far_t *f, uint64_t ino)
250 {
251         int ret;
252         far_info_t si;

254         while (1) {
255                 if (ino > f->f_current_ino)
```

```
256                         return (0);
257                 ret = far_get_info(f, ino, FAR_OLD, &si, 0);
258                 if (ret && ret != ENOENT)
259                         return (ret);
260                 if (ret != ENOENT)
261                         break;
262                 ret = far_get_info(f, ino, FAR_NEW, &si, FI_ATTR_PARENT);
263                 if (ret && ret != ENOENT)
264                         return (ret);
265                 if (ret)
266                         return (0);     /* ignore for now */
267                 ino = si.si_parent;
268         }
269         return (1);
270 }

272 static int
273 build_path(far_t *f, far_dirent_t *dirent, uint64_t ino,
274     int devise_tempname, far_which_t which_in, far_path_t **fp)
275 {
276         int ret = 0;
277         far_dirent_t *de;
278         far_info_t si;
279         far_which_t which;
280         far_dirent_t temp_dirent;
281         char temp_buf[TEMPNAME_SIZE];

283         if (devise_tempname) {
284                 if (!dirent)
285                         return (EINVAL);
286                 temp_dirent = *dirent;
287                 tempname(ino, temp_buf, sizeof (temp_buf));
288                 temp_dirent.fd_name = temp_buf;
289                 dirent = &temp_dirent;
290         }

292         *fp = NULL;

294         for (de = dirent; de; de = de->fd_prev) {
295                 path_add_name(fp, de->fd_name, strlen(de->fd_name));
296                 ino = de->fd_parent_ino;
297         }

299         /*
300          * XXX TODO check if f->f_current_path is set. if yes, use it instead.
301          * otherwise save result of loop below to f_current_path
302          */
303         while (1) {
304                 int namebuflen;
305                 char *name;
306                 char *t_name;
307                 uint64_t old_parent;
308                 uint64_t new_parent;
309                 uint64_t old_gen = 0;
310                 uint64_t new_gen = 0;
311                 uint64_t parent;
312                 int check_tempname;
313                 uint64_t old_mode = 0;

315                 old_parent = 0;
316                 new_parent = 0;
317                 check_tempname = 0;
318                 ret = far_get_info(f, ino, FAR_OLD, &si, FI_ATTR_PARENT |
319                     FI_ATTR_GEN | FI_ATTR_MODE);
320                 if (ret && ret != ENOENT)
321                         return (ret);
```



```
322                 if (ret == 0) {
323                         old_parent = si.si_parent;
324                         old_gen = si.si_gen;
325                         old_mode = si.si_mode;
326                 }
327                 ret = far_get_info(f, ino, FAR_NEW, &si, FI_ATTR_PARENT |
328                     FI_ATTR_GEN | FI_ATTR_MODE);
329                 if (ret && ret != ENOENT)
330                         return (ret);
331                 if (ret == 0) {
332                         new_parent = si.si_parent;
333                         new_gen = si.si_gen;
334                 }
335                 if (old_parent && new_parent && old_gen != new_gen &&
336                     !(S_ISDIR(old_mode) && S_ISDIR(si.si_mode))) {
337                         if (which_in == FAR_OLD) {
338                                 new_parent = 0;
339                         } else if (which_in == FAR_NEW) {
340                                 old_parent = 0;
341                                 if (S_ISDIR(si.si_mode))
342                                         check_tempname = 1;
343                         }
344                 }

346                 if (f->f_pass == PASS_LINK) {
347                         if (old_parent && !new_parent) {
348                                 which = FAR_OLD;
349                         } else if (!old_parent && new_parent) {
350                                 which = FAR_NEW;
351                         } else if (is_ino_run(f, new_parent)) {
352                                 check_tempname = 1;
353                                 which = FAR_NEW;
354                         } else {
355                                 which = FAR_OLD;
356                         }
357                 } else {
358                         if (old_parent && !new_parent) {
359                                 which = FAR_OLD;
360                         } else {
361                                 check_tempname = 1;
362                                 which = FAR_NEW;
363                         }
364                 }
365                 if (which == FAR_OLD)
366                         parent = old_parent;
367                 else
368                         parent = new_parent;
369                 if (parent == ino)
370                         break;
371                 ret = far_find_entry(f, parent, ino, which, &t_name);
372                 if (ret)
373                         return (ret);
374                 name = strdup(t_name);
375                 far_free_name(t_name);
376                 namebuflen = strlen(name) + 1;
377                 if (check_tempname) {
378                         far_info_t si_old;
379                         far_info_t si_new;

381                         ret = far_get_info(f, parent, FAR_OLD,
382                                                 &si_old, FI_ATTR_GEN);
383                         if (ret && ret != ENOENT)
384                                 return (ret);
385                         if (ret == 0) {
386                                 ret = far_get_info(f, parent, FAR_NEW,
387                                                         &si_new, FI_ATTR_GEN);
```

```
388                                 if (ret)
389                                         return (ret);
390                                 if (si_old.si_gen != si_new.si_gen)
391                                         check_tempname = 0;
392                         } else {
393                                 check_tempname = 0;
394                         }
395                 }
396                 if (check_tempname) {
397                         uint64_t old_ino;

399                         ret = far_lookup_entry(f, parent, name,
400                                                FAR_OLD, &old_ino);
401                         if (ret && ret != ENOENT) {
402                                 far_free_name(name);
403                                 return (ret);
404                         }
405                         if ((ret == 0 && old_ino != ino) ||
406                             (ret == 0 && S_ISDIR(si.si_mode) &&
407                             !S_ISDIR(old_mode) && old_ino == ino)) {
408                                 int ret;
409                                 uint64_t cnt = 1;

411                                 if (f->f_pass == PASS_UNLINK &&
412                                     new_parent < f->f_current_ino) {
413                                         ret = far_get_count(&f->f_put_back_cnt,
414                                             old_ino, &cnt, NULL);
415                                         if (ret && ret != ENOENT)
416                                                 return (ret);
417                                 }
418                                 if (cnt) {
419                                         kmem_free(name, namebuflen);
420                                         namebuflen = TEMPNAME_SIZE;
421                                         name = kmem_alloc(namebuflen, KM_SLEEP);
422                                         tempname(ino, name, namebuflen);
423                                 }
424                         }
425                 }
426                 ino = parent;
427                 path_add_name(fp, name, strlen(name));
428                 kmem_free(name, namebuflen);
429         }
430         if (*fp == NULL)
431                 path_add_name(fp, "", 0);

433         return (0);
434 }

436 int
437 far_send_start(far_t *f)
438 {
439         int ret;
440         uint8_t o_uuid[16];
441         uint8_t n_uuid[16];
442         uint64_t o_ctrans;
443         uint64_t n_ctrans;
444         char *path = NULL;
445         int len;
446         char *p;
447         int cmd = FAR_CMD_SUBVOL;

449         ret = far_send_stream_header(f);
450         if (ret) {
451                 far_abort(f);
452                 return (ret);
453         }
```



```
455         if ((ret = far_get_uuid(f, FAR_NEW, n_uuid)) ||
456             (ret = far_get_ctransid(f, FAR_NEW, &n_ctrans)) ||
457             (ret = far_get_snapname(f, FAR_NEW, &path, &len)))
458                 goto out;
459         /* for now, strip the pool name */
460         if ((p = strchr(path, '/')))
461                 ++p;
462         else
463                 p = path;
464         ret = far_get_uuid(f, FAR_OLD, o_uuid);
465         if (ret && ret != ENOENT)
466                 goto out;
467         if (ret == 0) {
468                 ret = far_get_ctransid(f, FAR_OLD, &o_ctrans);
469                 if (ret)
470                         goto out;
471                 cmd = FAR_CMD_SNAPSHOT;
472         }
473         if ((ret = far_cmd_start(f, cmd)) ||
474             (ret = far_send_put_attr(f, FAR_ATTR_PATH, p, strlen(p))) ||
475             (ret = far_send_put_u64(f, FAR_ATTR_CTRANSID, n_ctrans)) ||
476             (ret = far_send_put_attr(f, FAR_ATTR_UUID, n_uuid, 16)))
477                 goto out;
478         if (cmd == FAR_CMD_SNAPSHOT) {
479                 if ((ret = far_send_put_u64(f, FAR_ATTR_CLONE_CTRANSID,
480                     o_ctrans)) ||
481                     (ret = far_send_put_attr(f, FAR_ATTR_CLONE_UUID,
482                     o_uuid, 16)))
483                         goto out;
484         }
485         ret = far_cmd_send(f);

487 out:
488         kmem_free(path, len);

490         return (ret);
491 }

493 int
494 far_send_create_file(far_t *f, far_dirent_t *dirent, uint64_t ino,
495     int devise_tempname, far_path_t **path_ret)
496 {
497         far_path_t *path = NULL;
498         far_info_t si;
499         int ret;
500         int send_rdev = 0;
501         int cmd;
502         uint64_t rdev = 0;
503         char *symlink = NULL;
504         int symlen = 0;

506         ret = build_path(f, dirent, ino, devise_tempname, FAR_NEW, &path);
507         if (ret)
508                 goto out;

510         ret = far_get_info(f, ino, FAR_NEW, &si,
511                            FI_ATTR_MODE | FI_ATTR_UID | FI_ATTR_GID);
512         if (ret)
513                 goto out;

515         if (S_ISREG(si.si_mode)) {
516                 cmd = FAR_CMD_MKFILE;
517         } else if (S_ISDIR(si.si_mode)) {
518                 cmd = FAR_CMD_MKDIR;
519         } else if (S_ISLNK(si.si_mode)) {
```

```
520                 cmd = FAR_CMD_SYMLINK;
521                 ret = far_read_symlink(f, ino, FAR_NEW, &symlink, &symlen);
522                 if (ret)
523                         goto out;
524         } else if (S_ISCHR(si.si_mode) || S_ISBLK(si.si_mode)) {
525                 cmd = FAR_CMD_MKNOD;
526                 send_rdev = 1;
527         } else if (S_ISFIFO(si.si_mode)) {
528                 cmd = FAR_CMD_MKFIFO;
529         } else if (S_ISSOCK(si.si_mode)) {
530                 cmd = FAR_CMD_MKSOCK;
531         } else {
532                 /* unknown file type, ignore for now */
533                 return (0);
534         }

536         if (send_rdev) {
537                 far_info_t sirdev;
538                 uint64_t r_major;
539                 uint64_t r_minor;
540                 ret = far_get_info(f, ino, FAR_NEW, &sirdev, FI_ATTR_RDEV);
541                 if (ret)
542                         goto out;
543                 rdev = sirdev.si_rdev;

545                 /* XXX hardcodedly transform rdev to linux form */
546                 r_major = rdev >> 32;
547                 r_minor = rdev & 0xffffffful;
548                 rdev = ((r_minor & 0xff) | ((r_major & 0xfff) << 8) |
549                     ((r_minor >> 8) << 20) | ((r_major >> 12) << 44));
550         }
551         /* send MKFILE */
552         if ((ret = far_cmd_start(f, cmd)) ||
553             (ret = put_path(f, FAR_ATTR_PATH, path)) ||
554             (ret = far_send_put_u64(f, FAR_ATTR_INO, ino)))
555                 goto out;
556         if (send_rdev) {
557                 ret = far_send_put_u64(f, FAR_ATTR_RDEV, rdev);
558                 if (ret)
559                         goto out;
560                 ret = far_send_put_u64(f, FAR_ATTR_MODE, si.si_mode);
561                 if (ret)
562                         goto out;
563         }
564         if (S_ISLNK(si.si_mode)) {
565                 ret = far_send_put_attr(f, FAR_ATTR_PATH_LINK,
566                                         symlink, strlen(symlink));
567                 if (ret)
568                         goto out;
569         }
570         if ((ret = far_cmd_send(f)))
571                 goto out;

573         /* send CHOWN */
574         if ((ret = far_cmd_start(f, FAR_CMD_CHOWN)) ||
575             (ret = put_path(f, FAR_ATTR_PATH, path)) ||
576             (ret = far_send_put_u64(f, FAR_ATTR_UID, si.si_uid)) ||
577             (ret = far_send_put_u64(f, FAR_ATTR_GID, si.si_gid)) ||
578             (ret = far_cmd_send(f)))
579                 goto out;

581         /* send CHMOD, but not for symlinks */
582         if (!S_ISLNK(si.si_mode)) {
583                 if ((ret = far_cmd_start(f, FAR_CMD_CHMOD)) ||
584                     (ret = put_path(f, FAR_ATTR_PATH, path)) ||
585                     (ret = far_send_put_u64(f, FAR_ATTR_MODE,
```



```
586                         si.si_mode & 0xfff)) ||
587                         (ret = far_cmd_send(f)))
588                             goto out;
589         }

591 out:
592         if (ret == 0 && path_ret)
593                 *path_ret = path;
594         else
595                 far_path_free(path);
596         if (symlink)
597                 kmem_free(symlink, symlen);
598         return (ret);
599 }

601 int
602 far_send_link(far_t *f, far_dirent_t *new_dirent, uint64_t ino,
603     uint64_t old_parent_ino, far_which_t which, int devise_tempname)
604 {
605         far_path_t *new_path = NULL;
606         far_path_t *old_path = NULL;
607         int ret;
608         far_dirent_t old_dirent = {
609                 .fd_name = NULL,
610                 .fd_parent_ino = old_parent_ino,
611                 .fd_prev = NULL,
612         };

614         ret = far_find_entry(f, old_parent_ino, ino, which,
615             &old_dirent.fd_name);
616         if (ret)
617                 return (ret);
618         ret = build_path(f, &old_dirent, ino, 0, FAR_OLD, &old_path);
619         if (ret)
620                 goto out;

622         ret = build_path(f, new_dirent, ino, devise_tempname, FAR_NEW,
623             &new_path);
624         if (ret)
625                 goto out;

627         if ((ret = far_cmd_start(f, FAR_CMD_LINK)) ||
628             (ret = put_path(f, FAR_ATTR_PATH_LINK, old_path)) ||
629             (ret = put_path(f, FAR_ATTR_PATH, new_path)) ||
630             (ret = far_cmd_send(f)))
631                 goto out;

633         if (f->f_pass == PASS_UNLINK)
634                 ret = far_send_mtime_update(f, new_dirent, ino);
635 out:
636         if (old_dirent.fd_name)
637                 kmem_free(old_dirent.fd_name, strlen(old_dirent.fd_name) + 1);
638         far_path_free(old_path);
639         far_path_free(new_path);
640         return (ret);
641 }

643 int
644 far_send_mkdir(far_t *f, far_dirent_t *dirent,
645                 uint64_t ino, int devise_tempname)
646 {
647         far_path_t *path = NULL;
648         far_info_t si;
649         int ret;

651         ret = build_path(f, dirent, ino, devise_tempname, FAR_NEW, &path);
```

```
652         if (ret)
653                 goto out;
654
655         ret = far_get_info(f, ino, FAR_NEW, &si,
656                            FI_ATTR_UID | FI_ATTR_GID | FI_ATTR_MODE);
657         if (ret)
658                 goto out;
659
660         if (path->fp_total_len != 1) {
661                 /* don't send an mkdir for the root, but send chown/chmod */
662                 if ((ret = far_cmd_start(f, FAR_CMD_MKDIR)) ||
663                     (ret = put_path(f, FAR_ATTR_PATH, path)) ||
664                     (ret = far_send_put_u64(f, FAR_ATTR_INO, ino)) ||
665                     (ret = far_cmd_send(f)))
666                         goto out;
667         }
668
669         /* send CHOWN */
670         if ((ret = far_cmd_start(f, FAR_CMD_CHOWN)) ||
671             (ret = put_path(f, FAR_ATTR_PATH, path)) ||
672             (ret = far_send_put_u64(f, FAR_ATTR_UID, si.si_uid)) ||
673             (ret = far_send_put_u64(f, FAR_ATTR_GID, si.si_gid)) ||
674             (ret = far_cmd_send(f)))
675                 goto out;
676
677         /* send CHMOD */
678         if ((ret = far_cmd_start(f, FAR_CMD_CHMOD)) ||
679             (ret = put_path(f, FAR_ATTR_PATH, path)) ||
680             (ret = far_send_put_u64(f, FAR_ATTR_MODE,
681             si.si_mode & 0xfff)) ||
682             (ret = far_cmd_send(f)))
683                 goto out;
684
685 out:
686         far_path_free(path);
687         return (ret);
688 }
689
690 /* this one is only used for directory renames */
691 int
692 far_send_rename(far_t *f, far_dirent_t *new_dirent, uint64_t ino,
693     uint64_t old_parent_ino, int devise_tempname)
694 {
695         far_path_t *new_path = NULL;
696         far_path_t *old_path = NULL;
697         int ret;
698         far_dirent_t old_dirent = {
699                 .fd_name = NULL,
700                 .fd_parent_ino = old_parent_ino,
701                 .fd_prev = NULL,
702         };
703
704         ret = far_find_entry(f, old_parent_ino, ino, FAR_OLD,
705             &old_dirent.fd_name);
706         if (ret)
707                 return (ret);
708         ret = build_path(f, &old_dirent, ino, 0, FAR_OLD, &old_path);
709         if (ret)
710                 goto out;
711
712         ret = build_path(f, new_dirent, ino, devise_tempname, FAR_NEW,
713             &new_path);
714         if (ret)
715                 goto out;
716
717         if ((ret = far_cmd_start(f, FAR_CMD_RENAME)) ||
```



```
718             (ret = put_path(f, FAR_ATTR_PATH, old_path)) ||
719             (ret = put_path(f, FAR_ATTR_PATH_TO, new_path)) ||
720             (ret = far_cmd_send(f)))
721                 goto out;
722 out:
723         far_path_free(old_path);
724         far_path_free(new_path);
725         return (ret);
726 }
727
728 int
729 far_send_rename_from_tempname(far_t *f, far_dirent_t *dirent,
730     uint64_t ino, uint64_t old)
731 {
732         char buf[TEMPNAME_SIZE];
733         far_path_t *new_path = NULL;
734         far_path_t *old_path = NULL;
735         int ret;
736         far_dirent_t old_dirent;
737
738         tempname(old, buf, sizeof (buf));
739         old_dirent = *dirent;
740         old_dirent.fd_name = buf;
741
742         ret = build_path(f, &old_dirent, old, 0, FAR_OLD, &old_path);
743         if (ret)
744                 goto out;
745         ret = build_path(f, dirent, ino, 0, FAR_NEW, &new_path);
746         if (ret)
747                 goto out;
748
749         if ((ret = far_cmd_start(f, FAR_CMD_RENAME)) ||
750             (ret = put_path(f, FAR_ATTR_PATH, old_path)) ||
751             (ret = put_path(f, FAR_ATTR_PATH_TO, new_path)) ||
752             (ret = far_cmd_send(f)))
753                 goto out;
754
755         ret = far_send_mtime_update(f, dirent, old);
756
757 out:
758         far_path_free(old_path);
759         far_path_free(new_path);
760         return (ret);
761 }
762
763 int
764 far_send_unlink(far_t *f, far_dirent_t *dirent, uint64_t ino)
765 {
766         far_path_t *path = NULL;
767         int ret;
768
769         ret = build_path(f, dirent, ino, 0, FAR_OLD, &path);
770         if (ret)
771                 goto out;
772
773         if ((ret = far_cmd_start(f, FAR_CMD_UNLINK)) ||
774             (ret = put_path(f, FAR_ATTR_PATH, path)) ||
775             (ret = far_cmd_send(f)))
776                 goto out;
777
778 out:
779         far_path_free(path);
780         return (ret);
781 }
782
783 int
```

```
784 far_send_rmdir(far_t *f, far_dirent_t *dirent, uint64_t ino)
785 {
786         far_path_t *path;
787         int ret;
788
789         ret = build_path(f, dirent, ino, 0, FAR_OLD, &path);
790         if (ret)
791                 goto out;
792
793         if ((ret = far_cmd_start(f, FAR_CMD_RMDIR)) ||
794             (ret = put_path(f, FAR_ATTR_PATH, path)) ||
795             (ret = far_cmd_send(f)))
796                 goto out;
797
798 out:
799         far_path_free(path);
800         return (ret);
801 }
802
803 int
804 far_send_file_data(far_t *f, far_path_t **path_p,
805                    far_dirent_t *dirent, uint64_t ino,
806                    uint64_t off, uint64_t len, void *data)
807 {
808         int ret = 0;
809
810         if (!*path_p) {
811                 ret = build_path(f, dirent, ino, 0, FAR_NEW, path_p);
812                 if (ret)
813                         return (ret);
814         }
815
816         while (len) {
817                 uint64_t l = MIN(len, FAR_SEND_READ_SIZE);
818
819                 if ((ret = far_cmd_start(f, FAR_CMD_WRITE)) ||
820                     (ret = put_path(f, FAR_ATTR_PATH, *path_p)) ||
821                     (ret = far_send_put_u64(f, FAR_ATTR_FILE_OFFSET, off))||
822                     (ret = far_send_put_attr(f, FAR_ATTR_DATA, data, l)) ||
823                     (ret = far_cmd_send(f)))
824                         goto out;
825                 data += l;
826                 off += l;
827                 len -= l;
828         }
829
830 out:
831         return (ret);
832 }
833
834 int
835 far_send_mtime_update(far_t *f, far_dirent_t *dirent, uint64_t ino)
836 {
837         far_path_t *path = NULL;
838         int ret;
839         far_info_t si;
840
841         ret = far_get_info(f, ino, FAR_NEW, &si,
842                            FI_ATTR_ATIME | FI_ATTR_MTIME |
843                            FI_ATTR_CTIME | FI_ATTR_OTIME);
844         if (ret) {
845                 if (ret == ENOENT)
846                         ret = 0;
847                 goto out;
848         }
```



```
849
850         ret = build_path(f, dirent, ino, 0, FAR_NEW, &path);
851         if (ret)
852                 goto out;
853
854         if ((ret = far_cmd_start(f, FAR_CMD_UTIMES)) ||
855             (ret = put_path(f, FAR_ATTR_PATH, path)) ||
856             (ret = far_send_put_time(f, FAR_ATTR_ATIME, &si.si_atime)) ||
857             (ret = far_send_put_time(f, FAR_ATTR_MTIME, &si.si_mtime)) ||
858             (ret = far_send_put_time(f, FAR_ATTR_CTIME, &si.si_ctime)) ||
859             (ret = far_send_put_time(f, FAR_ATTR_OTIME, &si.si_otime)) ||
860             (ret = far_cmd_send(f)))
861                 goto out;
862
863 out:
864         far_path_free(path);
865         return (ret);
866 }
867
868 int
869 far_send_truncate(far_t *f, far_dirent_t *dirent, uint64_t ino,
870     uint64_t new_size)
871 {
872         far_path_t *path = NULL;
873         int ret;
874
875         ret = build_path(f, dirent, ino, 0, FAR_NEW, &path);
876         if (ret)
877                 return (ret);
878
879         if ((ret = far_cmd_start(f, FAR_CMD_TRUNCATE)) ||
880             (ret = put_path(f, FAR_ATTR_PATH, path)) ||
881             (ret = far_send_put_u64(f, FAR_ATTR_SIZE, new_size)) ||
882             (ret = far_cmd_send(f)))
883                 goto out;
884
885 out:
886         far_path_free(path);
887         return (ret);
888 }
889
890 int
891 far_send_chown(far_t *f, far_dirent_t *dirent, uint64_t ino,
892                    uint64_t new_uid, uint64_t new_gid)
893 {
894         far_path_t *path = NULL;
895         int ret;
896
897         ret = build_path(f, dirent, ino, 0, FAR_NEW, &path);
898         if (ret)
899                 return (ret);
900
901         if ((ret = far_cmd_start(f, FAR_CMD_CHOWN)) ||
902             (ret = put_path(f, FAR_ATTR_PATH, path)) ||
903             (ret = far_send_put_u64(f, FAR_ATTR_UID, new_uid)) ||
904             (ret = far_send_put_u64(f, FAR_ATTR_GID, new_gid)) ||
905             (ret = far_cmd_send(f)))
906                 goto out;
907
908 out:
909         far_path_free(path);
910         return (ret);
911 }
912
913 int
914 far_send_chmod(far_t *f, far_dirent_t *dirent, uint64_t ino,
915                    uint64_t new_mode)
```

```
916 {
917         far_path_t *path = NULL;
918         int ret;

920         ret = build_path(f, dirent, ino, 0, FAR_NEW, &path);
921         if (ret)
922                 return (ret);

924         if ((ret = far_cmd_start(f, FAR_CMD_CHMOD)) ||
925             (ret = put_path(f, FAR_ATTR_PATH, path)) ||
926             (ret = far_send_put_u64(f, FAR_ATTR_MODE, new_mode)) ||
927             (ret = far_cmd_send(f)))
928                 goto out;

930 out:
931         far_path_free(path);
932         return (ret);
933 }

935 int
936 far_send_end(far_t *f)
937 {
938         int ret;

940         if ((ret = far_cmd_start(f, FAR_CMD_END)) ||
941             (ret = far_cmd_send(f)))
942                 goto out;

944 out:
945         return (ret);
946 }
947 #endif /* ! codereview */
```

```
**********************************************************
   10469 Fri Oct 26 17:09:25 2012
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
______unchanged_portion_omitted_

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

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

 197 int dsl_dataset_hold(const char *name, void *tag, dsl_dataset_t **dsp);
 198 int dsl_dataset_hold_obj(struct dsl_pool *dp, uint64_t dsobj,
 199     void *tag, dsl_dataset_t **);
 200 int dsl_dataset_own(const char *name, boolean_t inconsistentok,
 201     void *tag, dsl_dataset_t **dsp);
 202 int dsl_dataset_own_obj(struct dsl_pool *dp, uint64_t dsobj,
 203     boolean_t inconsistentok, void *tag, dsl_dataset_t **dsp);
 204 void dsl_dataset_name(dsl_dataset_t *ds, char *name);
 205 int dsl_dataset_namelen(dsl_dataset_t *ds);
 206 #endif /* ! codereview */
 207 void dsl_dataset_rele(dsl_dataset_t *ds, void *tag);
 208 void dsl_dataset_disown(dsl_dataset_t *ds, void *tag);
 209 void dsl_dataset_drop_ref(dsl_dataset_t *ds, void *tag);
 210 boolean_t dsl_dataset_tryown(dsl_dataset_t *ds, boolean_t inconsistentok,
 211     void *tag);
 212 void dsl_dataset_make_exclusive(dsl_dataset_t *ds, void *tag);
 213 void dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
 214     minor_t minor);
 215 uint64_t dsl_dataset_create_sync(dsl_dir_t *pds, const char *lastname,
 216     dsl_dataset_t *origin, uint64_t flags, cred_t *, dmu_tx_t *);
 217 uint64_t dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
 218     uint64_t flags, dmu_tx_t *tx);
 219 int dsl_dataset_destroy(dsl_dataset_t *ds, void *tag, boolean_t defer);
 220 dsl_checkfunc_t dsl_dataset_destroy_check;
 221 dsl_syncfunc_t dsl_dataset_destroy_sync;
 222 dsl_syncfunc_t dsl_dataset_user_hold_sync;
 223 int dsl_dataset_snapshot_check(dsl_dataset_t *ds, const char *, dmu_tx_t *tx);
 224 void dsl_dataset_snapshot_sync(dsl_dataset_t *ds, const char *, dmu_tx_t *tx);
 225 int dsl_dataset_rename(char *name, const char *newname, boolean_t recursive);
 226 int dsl_dataset_promote(const char *name, char *conflsnap);
 227 int dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
 228     boolean_t force);
 229 int dsl_dataset_user_hold(char *dsname, char *snapname, char *htag,
 230     boolean_t recursive, boolean_t temphold, int cleanup_fd);
 231 int dsl_dataset_user_hold_for_send(dsl_dataset_t *ds, char *htag,
 232     boolean_t temphold);
 233 int dsl_dataset_user_release(char *dsname, char *snapname, char *htag,
 234     boolean_t recursive);
 235 int dsl_dataset_user_release_tmp(struct dsl_pool *dp, uint64_t dsobj,
 236     char *htag, boolean_t retry);
 237 int dsl_dataset_get_holds(const char *dsname, nvlist_t **nvp);

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

 242 spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);

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



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

 248 void dsl_dataset_block_born(dsl_dataset_t *ds, const blkptr_t *bp,
 249     dmu_tx_t *tx);
 250 int dsl_dataset_block_kill(dsl_dataset_t *ds, const blkptr_t *bp,
 251     dmu_tx_t *tx, boolean_t async);
 252 boolean_t dsl_dataset_block_freeable(dsl_dataset_t *ds, const blkptr_t *bp,
 253     uint64_t blk_birth);
 254 uint64_t dsl_dataset_prev_snap_txg(dsl_dataset_t *ds);

 256 void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
 257 void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);
 258 void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
 259 void dsl_dataset_space(dsl_dataset_t *ds,
 260     uint64_t *refdbytesp, uint64_t *availbytesp,
 261     uint64_t *usedobjsp, uint64_t *availobjsp);
 262 uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
 263 int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
 264     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
 265 int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
 266     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
 267 boolean_t dsl_dataset_is_dirty(dsl_dataset_t *ds);

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

 271 int dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
 272     uint64_t asize, uint64_t inflight, uint64_t *used,
 273     uint64_t *ref_rsrv);
 274 int dsl_dataset_set_quota(const char *dsname, zprop_source_t source,
 275     uint64_t quota);
 276 dsl_syncfunc_t dsl_dataset_set_quota_sync;
 277 int dsl_dataset_set_reservation(const char *dsname, zprop_source_t source,
 278     uint64_t reservation);

 280 int dsl_destroy_inconsistent(const char *dsname, void *arg);

 282 #ifdef ZFS_DEBUG
 283 #define dprintf_ds(ds, fmt, ...) do { \
 284         if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
 285         char *__ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
 286         dsl_dataset_name(ds, __ds_name); \
 287         dprintf("ds=%s " fmt, __ds_name, __VA_ARGS__); \
 288         kmem_free(__ds_name, MAXNAMELEN); \
 289         } \
 290 _NOTE(CONSTCOND) } while (0)
 291 #else
 292 #define dprintf_ds(dd, fmt, ...)
 293 #endif

 295 #ifdef  __cplusplus
 296 }
 297 #endif

 299 #endif /* _SYS_DSL_DATASET_H */
```

```
**********************************************************
    3321 Fri Oct 26 17:09:25 2012
new/usr/src/uts/common/fs/zfs/sys/far.h
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * 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) 2012 Alexander Block. All rights reserved.
 * Copyright (c) 2012 STRATO AG. All rights reserved.
 */

#ifndef _SYS_FAR_H
#define _SYS_FAR_H

#include <sys/inttypes.h>
#include <sys/types.h>
#include <sys/dmu.h>
#include <sys/vnode.h>

#ifdef  __cplusplus
extern "C" {
#endif

#define FAR_SEND_STREAM_MAGIC "btrfs-stream"
#define FAR_SEND_STREAM_VERSION 1

#define FAR_SEND_BUF_SIZE 65536
#define FAR_SEND_READ_SIZE 49152

typedef struct _far_stream_header {
        char            fs_magic[sizeof (FAR_SEND_STREAM_MAGIC)];
        uint32_t        fs_version;
} __attribute__((__packed__)) far_stream_header_t;

typedef struct _far_cmd_header {
        /* len of the payload, not including header */
        uint32_t        fc_len;
        uint16_t        fc_cmd;
        /* the crc includes the header, but with fc_crc assumed as 0 */
        uint32_t        fc_crc;
} __attribute__((__packed__)) far_cmd_header_t;
```



```
typedef struct _far_attr_header {
        uint16_t        fa_type;
        /* len of the payload, not including header */
        uint16_t        fa_len;
} __attribute__((__packed__)) far_attr_header_t;

/* commands */
#define FAR_CMD_SUBVOL          1
#define FAR_CMD_SNAPSHOT        2
#define FAR_CMD_MKFILE          3
#define FAR_CMD_MKDIR           4
#define FAR_CMD_MKNOD           5
#define FAR_CMD_MKFIFO          6
#define FAR_CMD_MKSOCK          7
#define FAR_CMD_SYMLINK 8
#define FAR_CMD_RENAME          9
#define FAR_CMD_LINK            10
#define FAR_CMD_UNLINK          11
#define FAR_CMD_RMDIR           12
#define FAR_CMD_SET_XATTR       13
#define FAR_CMD_REMOVE_XATTR    14
#define FAR_CMD_WRITE           15
#define FAR_CMD_CLONE           16
#define FAR_CMD_TRUNCATE        17
#define FAR_CMD_CHMOD           18
#define FAR_CMD_CHOWN           19
#define FAR_CMD_UTIMES          20
#define FAR_CMD_END             21
#define FAR_CMD_MAX             21

/* attributes */
#define FAR_ATTR_UUID                   1
#define FAR_ATTR_CTRANSID               2
#define FAR_ATTR_INO                    3
#define FAR_ATTR_SIZE                   4
#define FAR_ATTR_MODE                   5
#define FAR_ATTR_UID                    6
#define FAR_ATTR_GID                    7
#define FAR_ATTR_RDEV                   8
#define FAR_ATTR_CTIME                  9
#define FAR_ATTR_MTIME                  10
#define FAR_ATTR_ATIME                  11
#define FAR_ATTR_OTIME                  12
#define FAR_ATTR_XATTR_NAME             13
#define FAR_ATTR_XATTR_DATA             14
#define FAR_ATTR_PATH                   15
#define FAR_ATTR_PATH_TO                16
#define FAR_ATTR_PATH_LINK              17
#define FAR_ATTR_FILE_OFFSET            18
#define FAR_ATTR_DATA                   19
#define FAR_ATTR_CLONE_UUID             20
#define FAR_ATTR_CLONE_CTRANSID 21
#define FAR_ATTR_CLONE_PATH             22
#define FAR_ATTR_CLONE_OFFSET           23
#define FAR_ATTR_CLONE_LEN              24
#define FAR_ATTR_MAX                    24

int far_send(objset_t *tosnap, objset_t *fromsnap, int outfd, vnode_t *vp,
    offset_t *off);

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_FAR_H */
#endif /* ! codereview */
```

****
1022 Fri Oct 26 17:09:25 2012
new/usr/src/uts/common/fs/zfs/sys/far_crc32c.h
FAR: generating send-streams in portable format
This commit adds a switch '-F' to zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
****

```
/*
 * 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
 */

#ifndef _SYS_FAR_CRC32C_H
#define _SYS_FAR_CRC32C_H

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

uint32_t far_crc32c(uint32_t seed, const uint8_t *data, int len);

#endif /* _SYS_FAR_CRC32C_H */
#endif /* ! codereview */
```

```
**********************************************************
    7203 Fri Oct 26 17:09:25 2012
new/usr/src/uts/common/fs/zfs/sys/far_impl.h
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * 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) 2012 STRATO AG. All rights reserved.
 */

#ifndef _SYS_FAR_IMPL_H
#define _SYS_FAR_IMPL_H

#include <sys/inttypes.h>
#include <sys/types.h>
#include <sys/cmn_err.h>
#include <sys/spa.h>
#include <sys/arc.h>
#include <sys/dsl_dataset.h>
#include <sys/dnode.h>
#include <sys/sa.h>
#include <sys/avl.h>
#include <sys/dmu_impl.h>

#define FAR_NO_INO      0

enum pass {
        PASS_LINK,
        PASS_UNLINK
};

typedef struct _far_count_elem {
        avl_node_t      fce_avl_node;
        uint64_t        fce_ino;
        uint64_t        fce_count;
        uint64_t        fce_aux;
} far_count_elem_t;

typedef struct _far_counter {
        avl_tree_t fc_avl;
        char *fc_name;
} far_counter_t;
```



```
typedef struct blklevel {
        uint64_t        bl_blk;
        int             bl_nslots;
        blkptr_t        *bl_bp;
        arc_buf_t       *bl_buf;
} blklevel_t;

typedef struct _far {
        enum pass f_pass;
        struct _far_ops *f_ops;
        far_counter_t f_del_dir_cnt;
        far_counter_t f_put_back_cnt;
        far_counter_t f_link_add_cnt;
        uint64_t f_current_ino;
        struct _far_path *f_current_path;
        int             f_alloc_len;
        uint8_t         *f_buf;
        int             f_size;
        struct vnode    *f_vp;          /* file to which we are reporting */
        offset_t        *f_offp;
        int             f_err;          /* error that stopped diff search */
        dsl_dataset_t   *f_fromds;
        dsl_dataset_t   *f_tods;
        objset_t        *f_fromsnap;
        objset_t        *f_tosnap;
        dnode_phys_t    *f_dnp;
        blklevel_t      *f_bl;
        blklevel_t      *f_filebl;
        uint64_t        f_fromtxg;
        sa_attr_type_t  *f_from_sa_table;
        sa_attr_type_t  *f_to_sa_table;
        uint64_t        f_shares_dir;
        dmu_sendarg_t   f_dmu_sendarg;
} far_t;

typedef struct _far_ops {
        int (*far_dir_add)(far_t *f, uint64_t ino);
        int (*far_dir_del)(far_t *f, uint64_t ino);
        int (*far_dir_mod)(far_t *f, uint64_t ino);
        int (*far_dirent_add)(void *far_enump, char *name, uint64_t ino);
        int (*far_dirent_del)(void *far_enump, char *name, uint64_t ino);
        int (*far_dirent_mod)(void *far_enump, char *name,
            uint64_t ino_old, uint64_t ino_new);
        int (*far_dirent_unmod)(void *far_enump, char *name, uint64_t ino);
        int (*far_file_add)(far_t *f, uint64_t ino);
        int (*far_file_del)(far_t *f, uint64_t ino);
        int (*far_file_mod)(far_t *f, uint64_t ino);
        int (*far_file_data)(void *far_filep, void *data, uint64_t off,
            uint64_t len);
} far_ops_t;

typedef struct _far_path {
        struct _far_path *fp_next;
        int fp_len;
        int fp_total_len;
        char fp_buf[0];
} far_path_t;

typedef struct _far_dirent {
        char *fd_name;
        uint64_t fd_parent_ino;
        struct _far_dirent *fd_prev;
} far_dirent_t;

typedef enum _far_which {
```

```
124         FAR_UNDEF,
125         FAR_OLD,
126         FAR_NEW
127 } far_which_t;

129 typedef struct _far_time {
130         uint64_t        st_sec;
131         uint64_t        st_nsec;
132 } far_time_t;

134 #define FI_ATTR_ATIME           (1 <<  0)
135 #define FI_ATTR_MTIME           (1 <<  1)
136 #define FI_ATTR_CTIME           (1 <<  2)
137 #define FI_ATTR_OTIME           (1 <<  3)
138 #define FI_ATTR_MODE            (1 <<  4)
139 #define FI_ATTR_SIZE            (1 <<  5)
140 #define FI_ATTR_NENTRIES        (1 <<  5)
141 #define FI_ATTR_PARENT          (1 <<  6)
142 #define FI_ATTR_LINKS           (1 <<  7)
143 #define FI_ATTR_RDEV            (1 <<  8)
144 #define FI_ATTR_UID             (1 <<  9)
145 #define FI_ATTR_GID             (1 << 10)
146 #define FI_ATTR_GEN             (1 << 11)
147 /* XXX TODO xattr, acl, dacl */

149 #undef si_uid   /* XXX defined in siginfo.h */
150 #undef si_gid   /* XXX defined in siginfo.h */
151 typedef struct _far_info {
152         uint64_t        si_nlinks;
153         uint64_t        si_parent;
154         union {
155                 uint64_t        si_nentries;
156                 uint64_t        si_size;
157         };
158         far_time_t      si_atime;
159         far_time_t      si_mtime;
160         far_time_t      si_ctime;
161         far_time_t      si_otime;
162         uint64_t        si_mode;
163         /* XXX TODO xattr */
164         uint64_t        si_rdev;
165         uint64_t        si_uid;
166         uint64_t        si_gid;
167         uint64_t        si_gen;
168 } far_info_t;

170 int far_start(far_t *f, far_ops_t **);
171 int far_start2(far_t *f, far_ops_t **);
172 int far_abort(far_t *f);
173 int far_end(far_t *f);

175 int far_dirent_add_file(far_t *f, far_dirent_t *dirent,
176     uint64_t ino, uint64_t mode, int exists);
177 void far_path_free(far_path_t *fp);

179 int far_get_info(far_t *f, uint64_t dnobj, far_which_t which,
180     far_info_t *sp, uint64_t flags);

182 typedef int (*far_file_cb_t)(void *ctx, void *data, int len);
183 int far_file_contents(far_t *f, uint64_t dnobj, void *ctx);
184 int far_dir_contents(far_t *f, uint64_t dnobj, void *ctx);
185 int far_find_entry(far_t *f, uint64_t dirobj, uint64_t dnobj,
186     far_which_t which, char **name);
187 void far_free_name(char *name);
188 int far_lookup_entry(far_t *f, uint64_t dirobj, char *name,
189     far_which_t which, uint64_t *dnobj);
```



```
190 int far_write(far_t *f, const uint8_t *data, int len);
191 int far_get_uuid(far_t *f, far_which_t which, uint8_t data[16]);
192 int far_get_ctransid(far_t *f, far_which_t which,
193     uint64_t *ctransid);
194 int far_get_snapname(far_t *f, far_which_t which,
195     char **name, int *len);
196 int far_read_symlink(far_t *f, uint64_t dnobj, far_which_t which,
197     char **target, int *plen);

199 int far_send_start(far_t *f);
200 int far_send_create_file(far_t *f, far_dirent_t *dirent, uint64_t ino,
201 int devise_tempname, far_path_t **path_ret);
202 int far_send_link(far_t *f, far_dirent_t *new_dirent, uint64_t ino,
203     uint64_t old_parent_ino, far_which_t which,
204 int devise_tempname);
205 int far_send_mkdir(far_t *f, far_dirent_t *dirent, uint64_t ino,
206 int devise_tempname);
207 int far_send_rename(far_t *f, far_dirent_t *dirent, uint64_t ino,
208     uint64_t old_parent_ino, int devise_tempname);
209 int far_send_rename_from_tempname(far_t *f, far_dirent_t *dirent,
210     uint64_t ino, uint64_t old);
211 int far_send_unlink(far_t *f, far_dirent_t *dirent, uint64_t ino);
212 int far_send_rmdir(far_t *f, far_dirent_t *dirent, uint64_t ino);
213 /* TODO: make **path *path or do we really still alloc it? */
214 int far_send_file_data(far_t *f,
215     far_path_t **path, far_dirent_t *dirent,
216     uint64_t ino, uint64_t off, uint64_t len, void *data);
217 int far_send_mtime_update(far_t *f, far_dirent_t *dirent, uint64_t ino);
218 int far_send_truncate(far_t *f, far_dirent_t *dirent, uint64_t ino,
219     uint64_t new_size);
220 int far_send_chown(far_t *f, far_dirent_t *dirent, uint64_t ino,
221     uint64_t new_uid, uint64_t new_gid);
222 int far_send_chmod(far_t *f, far_dirent_t *dirent, uint64_t ino,
223     uint64_t new_mode);
224 int far_send_end(far_t *f);
225 int far_count_init(far_counter_t *fc, char *name);
226 void far_count_destroy(far_counter_t *fc);
227 int far_add_count(far_counter_t *fc, uint64_t ino, uint64_t inc,
228     uint64_t aux, uint64_t *new_count, uint64_t *old_aux);
229 void far_free_count(far_counter_t *fc, uint64_t ino);
230 int far_get_count(far_counter_t *fc, uint64_t ino, uint64_t *new_count,
231     uint64_t *old_aux);
232 int far_count_fini(far_counter_t *fc);

234 void far_send_init(far_t *f);
235 void far_send_fini(far_t *f);

237 #endif /* _SYS_FAR_IMPL_H */
238 #endif /* ! codereview */
```

```
**********************************************************
  143125 Fri Oct 26 17:09:25 2012
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
FAR: generating send-streams in portable format
This commit adds a switch '-F' to  zfs send. This set, zfs send generates
a stream in FAR-format instead of the traditional zfs stream format. The
generated send stream is compatible with the stream generated from 'btrfs send'
and can in principle easily be received to any filesystem.
**********************************************************
```

```
/*
 * 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.
 * Portions Copyright 2011 Martin Matuska
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

/*
 * ZFS ioctls.
 *
 * This file handles the ioctls to /dev/zfs, used for configuring ZFS storage
 * pools and filesystems, e.g. with /sbin/zfs and /sbin/zpool.
 *
 * There are two ways that we handle ioctls: the legacy way where almost
 * all of the logic is in the ioctl callback, and the new way where most
 * of the marshalling is handled in the common entry point, zfsdev_ioctl().
 *
 * Non-legacy ioctls should be registered by calling
 * zfs_ioctl_register() from zfs_ioctl_init().  The ioctl is invoked
 * from userland by lzc_ioctl().
 *
 * The registration arguments are as follows:
 *
 * const char *name
 *   The name of the ioctl.  This is used for history logging.  If the
 *   ioctl returns successfully (the callback returns 0), and allow_log
 *   is true, then a history log entry will be recorded with the input &
 *   output nvlists.  The log entry can be printed with "zpool history -i".
 *
 * zfs_ioc_t ioc
 *   The ioctl request number, which userland will pass to ioctl(2).
 *   The ioctl numbers can change from release to release, because
 *   the caller (libzfs) must be matched to the kernel.
 *
 * zfs_secpolicy_func_t *secpolicy
```



```
 *   This function will be called before the zfs_ioc_func_t, to
 *   determine if this operation is permitted.  It should return EPERM
 *   on failure, and 0 on success.  Checks include determining if the
 *   dataset is visible in this zone, and if the user has either all
 *   zfs privileges in the zone (SYS_MOUNT), or has been granted permission
 *   to do this operation on this dataset with "zfs allow".
 *
 * zfs_ioc_namecheck_t namecheck
 *   This specifies what to expect in the zfs_cmd_t:zc_name -- a pool
 *   name, a dataset name, or nothing.  If the name is not well-formed,
 *   the ioctl will fail and the callback will not be called.
 *   Therefore, the callback can assume that the name is well-formed
 *   (e.g. is null-terminated, doesn't have more than one '@' character,
 *   doesn't have invalid characters).
 *
 * zfs_ioc_poolcheck_t pool_check
 *   This specifies requirements on the pool state.  If the pool does
 *   not meet them (is suspended or is readonly), the ioctl will fail
 *   and the callback will not be called.  If any checks are specified
 *   (i.e. it is not POOL_CHECK_NONE), namecheck must not be NO_NAME.
 *   Multiple checks can be or-ed together (e.g. POOL_CHECK_SUSPENDED |
 *   POOL_CHECK_READONLY).
 *
 * boolean_t smush_outnvlist
 *   If smush_outnvlist is true, then the output is presumed to be a
 *   list of errors, and it will be "smushed" down to fit into the
 *   caller's buffer, by removing some entries and replacing them with a
 *   single "N_MORE_ERRORS" entry indicating how many were removed.  See
 *   nvlist_smush() for details.  If smush_outnvlist is false, and the
 *   outnvlist does not fit into the userland-provided buffer, then the
 *   ioctl will fail with ENOMEM.
 *
 * zfs_ioc_func_t *func
 *   The callback function that will perform the operation.
 *
 *   The callback should return 0 on success, or an error number on
 *   failure.  If the function fails, the userland ioctl will return -1,
 *   and errno will be set to the callback's return value.  The callback
 *   will be called with the following arguments:
 *
 *   const char *name
 *     The name of the pool or dataset to operate on, from
 *     zfs_cmd_t:zc_name.  The 'namecheck' argument specifies the
 *     expected type (pool, dataset, or none).
 *
 *   nvlist_t *innvl
 *     The input nvlist, deserialized from zfs_cmd_t:zc_nvlist_src.  Or
 *     NULL if no input nvlist was provided.  Changes to this nvlist are
 *     ignored.  If the input nvlist could not be deserialized, the
 *     ioctl will fail and the callback will not be called.
 *
 *   nvlist_t *outnvl
 *     The output nvlist, initially empty.  The callback can fill it in,
 *     and it will be returned to userland by serializing it into
 *     zfs_cmd_t:zc_nvlist_dst.  If it is non-empty, and serialization
 *     fails (e.g. because the caller didn't supply a large enough
 *     buffer), then the overall ioctl will fail.  See the
 *     'smush_nvlist' argument above for additional behaviors.
 *
 *     There are two typical uses of the output nvlist:
 *       - To return state, e.g. property values.  In this case,
 *         smush_outnvlist should be false.  If the buffer was not large
 *         enough, the caller will reallocate a larger buffer and try
 *         the ioctl again.
 *
 *       - To return multiple errors from an ioctl which makes on-disk
```

```
 *         changes.  In this case, smush_outnvlist should be true.
 *         Ioctls which make on-disk modifications should generally not
 *         use the outnvl if they succeed, because the caller can not
 *         distinguish between the operation failing, and
 *         deserialization failing.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/buf.h>
#include <sys/modctl.h>
#include <sys/open.h>
#include <sys/file.h>
#include <sys/kmem.h>
#include <sys/conf.h>
#include <sys/cmn_err.h>
#include <sys/stat.h>
#include <sys/zfs_ioctl.h>
#include <sys/zfs_vfsops.h>
#include <sys/zfs_znode.h>
#include <sys/zap.h>
#include <sys/spa.h>
#include <sys/spa_impl.h>
#include <sys/vdev.h>
#include <sys/priv_impl.h>
#include <sys/dmu.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_deleg.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_impl.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sunldi.h>
#include <sys/policy.h>
#include <sys/zone.h>
#include <sys/nvpair.h>
#include <sys/pathname.h>
#include <sys/mount.h>
#include <sys/sdt.h>
#include <sys/fs/zfs.h>
#include <sys/zfs_ctldir.h>
#include <sys/zfs_dir.h>
#include <sys/zfs_onexit.h>
#include <sys/zvol.h>
#include <sys/dsl_scan.h>
#include <sharefs/share.h>
#include <sys/dmu_objset.h>
#include <sys/far.h>
#endif /* ! codereview */

#include "zfs_namecheck.h"
#include "zfs_prop.h"
#include "zfs_deleg.h"
#include "zfs_comutil.h"

extern struct modlfs zfs_modlfs;

extern void zfs_init(void);
extern void zfs_fini(void);

ldi_ident_t zfs_li = NULL;
dev_info_t *zfs_dip;
```



```
uint_t zfs_fsyncer_key;
extern uint_t rrw_tsd_key;
static uint_t zfs_allow_log_key;

typedef int zfs_ioc_legacy_func_t(zfs_cmd_t *);
typedef int zfs_ioc_func_t(const char *, nvlist_t *, nvlist_t *);
typedef int zfs_secpolicy_func_t(zfs_cmd_t *, nvlist_t *, cred_t *);

typedef enum {
        NO_NAME,
        POOL_NAME,
        DATASET_NAME
} zfs_ioc_namecheck_t;

typedef enum {
        POOL_CHECK_NONE         = 1 << 0,
        POOL_CHECK_SUSPENDED    = 1 << 1,
        POOL_CHECK_READONLY     = 1 << 2,
} zfs_ioc_poolcheck_t;

typedef struct zfs_ioc_vec {
        zfs_ioc_legacy_func_t   *zvec_legacy_func;
        zfs_ioc_func_t          *zvec_func;
        zfs_secpolicy_func_t    *zvec_secpolicy;
        zfs_ioc_namecheck_t     zvec_namecheck;
        boolean_t               zvec_allow_log;
        zfs_ioc_poolcheck_t     zvec_pool_check;
        boolean_t               zvec_smush_outnvlist;
        const char              *zvec_name;
} zfs_ioc_vec_t;

/* This array is indexed by zfs_userquota_prop_t */
static const char *userquota_perms[] = {
        ZFS_DELEG_PERM_USERUSED,
        ZFS_DELEG_PERM_USERQUOTA,
        ZFS_DELEG_PERM_GROUPUSED,
        ZFS_DELEG_PERM_GROUPQUOTA,
};

static int zfs_ioc_userspace_upgrade(zfs_cmd_t *zc);
static int zfs_check_settable(const char *name, nvpair_t *property,
    cred_t *cr);
static int zfs_check_clearable(char *dataset, nvlist_t *props,
    nvlist_t **errors);
static int zfs_fill_zplprops_root(uint64_t, nvlist_t *, nvlist_t *,
    boolean_t *);
int zfs_set_prop_nvlist(const char *, zprop_source_t, nvlist_t *, nvlist_t *);
static int get_nvlist(uint64_t nvl, uint64_t size, int iflag, nvlist_t **nvp);

/* _NOTE(PRINTFLIKE(4)) - this is printf-like, but lint is too whiney */
void
__dprintf(const char *file, const char *func, int line, const char *fmt, ...)
{
        const char *newfile;
        char buf[512];
        va_list adx;

        /*
         * Get rid of annoying "../common/" prefix to filename.
         */
        newfile = strrchr(file, '/');
        if (newfile != NULL) {
                newfile = newfile + 1; /* Get rid of leading / */
        } else {
                newfile = file;
```

```
256         }

258         va_start(adx, fmt);
259         (void) vsnprintf(buf, sizeof (buf), fmt, adx);
260         va_end(adx);

262         /*
263          * To get this data, use the zfs-dprintf probe as so:
264          * dtrace -q -n 'zfs-dprintf \
265          *      /stringof(arg0) == "dbuf.c"/ \
266          *      {printf("%s: %s", stringof(arg1), stringof(arg3))}'
267          * arg0 = file name
268          * arg1 = function name
269          * arg2 = line number
270          * arg3 = message
271          */
272         DTRACE_PROBE4(zfs__dprintf,
273             char *, newfile, char *, func, int, line, char *, buf);
274 }

276 static void
277 history_str_free(char *buf)
278 {
279         kmem_free(buf, HIS_MAX_RECORD_LEN);
280 }

282 static char *
283 history_str_get(zfs_cmd_t *zc)
284 {
285         char *buf;

287         if (zc->zc_history == NULL)
288                 return (NULL);

290         buf = kmem_alloc(HIS_MAX_RECORD_LEN, KM_SLEEP);
291         if (copyinstr((void *)(uintptr_t)zc->zc_history,
292             buf, HIS_MAX_RECORD_LEN, NULL) != 0) {
293                 history_str_free(buf);
294                 return (NULL);
295         }

297         buf[HIS_MAX_RECORD_LEN -1] = '\0';

299         return (buf);
300 }

302 /*
303  * Check to see if the named dataset is currently defined as bootable
304  */
305 static boolean_t
306 zfs_is_bootfs(const char *name)
307 {
308         objset_t *os;

310         if (dmu_objset_hold(name, FTAG, &os) == 0) {
311                 boolean_t ret;
312                 ret = (dmu_objset_id(os) == spa_bootfs(dmu_objset_spa(os)));
313                 dmu_objset_rele(os, FTAG);
314                 return (ret);
315         }
316         return (B_FALSE);
317 }

319 /*
320  * zfs_earlier_version
321  *
```



```
322  *      Return non-zero if the spa version is less than requested version.
323  */
324 static int
325 zfs_earlier_version(const char *name, int version)
326 {
327         spa_t *spa;

329         if (spa_open(name, &spa, FTAG) == 0) {
330                 if (spa_version(spa) < version) {
331                         spa_close(spa, FTAG);
332                         return (1);
333                 }
334                 spa_close(spa, FTAG);
335         }
336         return (0);
337 }

339 /*
340  * zpl_earlier_version
341  *
342  * Return TRUE if the ZPL version is less than requested version.
343  */
344 static boolean_t
345 zpl_earlier_version(const char *name, int version)
346 {
347         objset_t *os;
348         boolean_t rc = B_TRUE;

350         if (dmu_objset_hold(name, FTAG, &os) == 0) {
351                 uint64_t zplversion;

353                 if (dmu_objset_type(os) != DMU_OST_ZFS) {
354                         dmu_objset_rele(os, FTAG);
355                         return (B_TRUE);
356                 }
357                 /* XXX reading from non-owned objset */
358                 if (zfs_get_zplprop(os, ZFS_PROP_VERSION, &zplversion) == 0)
359                         rc = zplversion < version;
360                 dmu_objset_rele(os, FTAG);
361         }
362         return (rc);
363 }

365 static void
366 zfs_log_history(zfs_cmd_t *zc)
367 {
368         spa_t *spa;
369         char *buf;

371         if ((buf = history_str_get(zc)) == NULL)
372                 return;

374         if (spa_open(zc->zc_name, &spa, FTAG) == 0) {
375                 if (spa_version(spa) >= SPA_VERSION_ZPOOL_HISTORY)
376                         (void) spa_history_log(spa, buf);
377                 spa_close(spa, FTAG);
378         }
379         history_str_free(buf);
380 }

382 /*
383  * Policy for top-level read operations (list pools).  Requires no privileges,
384  * and can be used in the local zone, as there is no associated dataset.
385  */
386 /* ARGSUSED */
387 static int
```

```
388 zfs_secpolicy_none(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
389 {
390         return (0);
391 }

393 /*
394  * Policy for dataset read operations (list children, get statistics).  Requires
395  * no privileges, but must be visible in the local zone.
396  */
397 /* ARGSUSED */
398 static int
399 zfs_secpolicy_read(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
400 {
401         if (INGLOBALZONE(curproc) ||
402             zone_dataset_visible(zc->zc_name, NULL))
403                 return (0);

405         return (ENOENT);
406 }

408 static int
409 zfs_dozonecheck_impl(const char *dataset, uint64_t zoned, cred_t *cr)
410 {
411         int writable = 1;

413         /*
414          * The dataset must be visible by this zone -- check this first
415          * so they don't see EPERM on something they shouldn't know about.
416          */
417         if (!INGLOBALZONE(curproc) &&
418             !zone_dataset_visible(dataset, &writable))
419                 return (ENOENT);

421         if (INGLOBALZONE(curproc)) {
422                 /*
423                  * If the fs is zoned, only root can access it from the
424                  * global zone.
425                  */
426                 if (secpolicy_zfs(cr) && zoned)
427                         return (EPERM);
428         } else {
429                 /*
430                  * If we are in a local zone, the 'zoned' property must be set.
431                  */
432                 if (!zoned)
433                         return (EPERM);

435                 /* must be writable by this zone */
436                 if (!writable)
437                         return (EPERM);
438         }
439         return (0);
440 }

442 static int
443 zfs_dozonecheck(const char *dataset, cred_t *cr)
444 {
445         uint64_t zoned;

447         if (dsl_prop_get_integer(dataset, "zoned", &zoned, NULL))
448                 return (ENOENT);

450         return (zfs_dozonecheck_impl(dataset, zoned, cr));
451 }

453 static int
```



```
454 zfs_dozonecheck_ds(const char *dataset, dsl_dataset_t *ds, cred_t *cr)
455 {
456         uint64_t zoned;

458         rw_enter(&ds->ds_dir->dd_pool->dp_config_rwlock, RW_READER);
459         if (dsl_prop_get_ds(ds, "zoned", 8, 1, &zoned, NULL)) {
460                 rw_exit(&ds->ds_dir->dd_pool->dp_config_rwlock);
461                 return (ENOENT);
462         }
463         rw_exit(&ds->ds_dir->dd_pool->dp_config_rwlock);

465         return (zfs_dozonecheck_impl(dataset, zoned, cr));
466 }

468 static int
469 zfs_secpolicy_write_perms(const char *name, const char *perm, cred_t *cr)
470 {
471         int error;
472         dsl_dataset_t *ds;

474         error = dsl_dataset_hold(name, FTAG, &ds);
475         if (error != 0)
476                 return (error);

478         error = zfs_dozonecheck_ds(name, ds, cr);
479         if (error == 0) {
480                 error = secpolicy_zfs(cr);
481                 if (error)
482                         error = dsl_deleg_access_impl(ds, perm, cr);
483         }

485         dsl_dataset_rele(ds, FTAG);
486         return (error);
487 }

489 static int
490 zfs_secpolicy_write_perms_ds(const char *name, dsl_dataset_t *ds,
491     const char *perm, cred_t *cr)
492 {
493         int error;

495         error = zfs_dozonecheck_ds(name, ds, cr);
496         if (error == 0) {
497                 error = secpolicy_zfs(cr);
498                 if (error)
499                         error = dsl_deleg_access_impl(ds, perm, cr);
500         }
501         return (error);
502 }

504 /*
505  * Policy for setting the security label property.
506  *
507  * Returns 0 for success, non-zero for access and other errors.
508  */
509 static int
510 zfs_set_slabel_policy(const char *name, char *strval, cred_t *cr)
511 {
512         char            ds_hexsl[MAXNAMELEN];
513         bslabel_t       ds_sl, new_sl;
514         boolean_t       new_default = FALSE;
515         uint64_t        zoned;
516         int             needed_priv = -1;
517         int             error;

519         /* First get the existing dataset label. */
```

```
520         error = dsl_prop_get(name, zfs_prop_to_name(ZFS_PROP_MLSLABEL),
521             1, sizeof (ds_hexsl), &ds_hexsl, NULL);
522         if (error)
523                 return (EPERM);
524
525         if (strcasecmp(strval, ZFS_MLSLABEL_DEFAULT) == 0)
526                 new_default = TRUE;
527
528         /* The label must be translatable */
529         if (!new_default && (hexstr_to_label(strval, &new_sl) != 0))
530                 return (EINVAL);
531
532         /*
533          * In a non-global zone, disallow attempts to set a label that
534          * doesn't match that of the zone; otherwise no other checks
535          * are needed.
536          */
537         if (!INGLOBALZONE(curproc)) {
538                 if (new_default || !blequal(&new_sl, CR_SL(CRED())))
539                         return (EPERM);
540                 return (0);
541         }
542
543         /*
544          * For global-zone datasets (i.e., those whose zoned property is
545          * "off", verify that the specified new label is valid for the
546          * global zone.
547          */
548         if (dsl_prop_get_integer(name,
549             zfs_prop_to_name(ZFS_PROP_ZONED), &zoned, NULL))
550                 return (EPERM);
551         if (!zoned) {
552                 if (zfs_check_global_label(name, strval) != 0)
553                         return (EPERM);
554         }
555
556         /*
557          * If the existing dataset label is nondefault, check if the
558          * dataset is mounted (label cannot be changed while mounted).
559          * Get the zfsvfs; if there isn't one, then the dataset isn't
560          * mounted (or isn't a dataset, doesn't exist, ...).
561          */
562         if (strcasecmp(ds_hexsl, ZFS_MLSLABEL_DEFAULT) != 0) {
563                 objset_t *os;
564                 static char *setsl_tag = "setsl_tag";
565
566                 /*
567                  * Try to own the dataset; abort if there is any error,
568                  * (e.g., already mounted, in use, or other error).
569                  */
570                 error = dmu_objset_own(name, DMU_OST_ZFS, B_TRUE,
571                     setsl_tag, &os);
572                 if (error)
573                         return (EPERM);
574
575                 dmu_objset_disown(os, setsl_tag);
576
577                 if (new_default) {
578                         needed_priv = PRIV_FILE_DOWNGRADE_SL;
579                         goto out_check;
580                 }
581
582                 if (hexstr_to_label(strval, &new_sl) != 0)
583                         return (EPERM);
584
585                 if (blstrictdom(&ds_sl, &new_sl))
```



```
586                         needed_priv = PRIV_FILE_DOWNGRADE_SL;
587                 else if (blstrictdom(&new_sl, &ds_sl))
588                         needed_priv = PRIV_FILE_UPGRADE_SL;
589         } else {
590                 /* dataset currently has a default label */
591                 if (!new_default)
592                         needed_priv = PRIV_FILE_UPGRADE_SL;
593         }
594
595 out_check:
596         if (needed_priv != -1)
597                 return (PRIV_POLICY(cr, needed_priv, B_FALSE, EPERM, NULL));
598         return (0);
599 }
600
601 static int
602 zfs_secpolicy_setprop(const char *dsname, zfs_prop_t prop, nvpair_t *propval,
603     cred_t *cr)
604 {
605         char *strval;
606
607         /*
608          * Check permissions for special properties.
609          */
610         switch (prop) {
611         case ZFS_PROP_ZONED:
612                 /*
613                  * Disallow setting of 'zoned' from within a local zone.
614                  */
615                 if (!INGLOBALZONE(curproc))
616                         return (EPERM);
617                 break;
618
619         case ZFS_PROP_QUOTA:
620                 if (!INGLOBALZONE(curproc)) {
621                         uint64_t zoned;
622                         char setpoint[MAXNAMELEN];
623                         /*
624                          * Unprivileged users are allowed to modify the
625                          * quota on things *under* (ie. contained by)
626                          * the thing they own.
627                          */
628                         if (dsl_prop_get_integer(dsname, "zoned", &zoned,
629                             setpoint))
630                                 return (EPERM);
631                         if (!zoned || strlen(dsname) <= strlen(setpoint))
632                                 return (EPERM);
633                 }
634                 break;
635
636         case ZFS_PROP_MLSLABEL:
637                 if (!is_system_labeled())
638                         return (EPERM);
639
640                 if (nvpair_value_string(propval, &strval) == 0) {
641                         int err;
642
643                         err = zfs_set_slabel_policy(dsname, strval, CRED());
644                         if (err != 0)
645                                 return (err);
646                 }
647                 break;
648         }
649
650         return (zfs_secpolicy_write_perms(dsname, zfs_prop_to_name(prop), cr));
651 }
```

```
653 /* ARGSUSED */
654 static int
655 zfs_secpolicy_set_fsacl(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
656 {
657         int error;

659         error = zfs_dozonecheck(zc->zc_name, cr);
660         if (error)
661                 return (error);

663         /*
664          * permission to set permissions will be evaluated later in
665          * dsl_deleg_can_allow()
666          */
667         return (0);
668 }

670 /* ARGSUSED */
671 static int
672 zfs_secpolicy_rollback(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
673 {
674         return (zfs_secpolicy_write_perms(zc->zc_name,
675             ZFS_DELEG_PERM_ROLLBACK, cr));
676 }

678 /* ARGSUSED */
679 static int
680 zfs_secpolicy_send(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
681 {
682         spa_t *spa;
683         dsl_pool_t *dp;
684         dsl_dataset_t *ds;
685         char *cp;
686         int error;

688         /*
689          * Generate the current snapshot name from the given objsetid, then
690          * use that name for the secpolicy/zone checks.
691          */
692         cp = strchr(zc->zc_name, '@');
693         if (cp == NULL)
694                 return (EINVAL);
695         error = spa_open(zc->zc_name, &spa, FTAG);
696         if (error)
697                 return (error);

699         dp = spa_get_dsl(spa);
700         rw_enter(&dp->dp_config_rwlock, RW_READER);
701         error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &ds);
702         rw_exit(&dp->dp_config_rwlock);
703         spa_close(spa, FTAG);
704         if (error)
705                 return (error);

707         dsl_dataset_name(ds, zc->zc_name);

709         error = zfs_secpolicy_write_perms_ds(zc->zc_name, ds,
710             ZFS_DELEG_PERM_SEND, cr);
711         dsl_dataset_rele(ds, FTAG);

713         return (error);
714 }

716 /* ARGSUSED */
717 static int
```



```
718 zfs_secpolicy_send_new(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
719 {
720         return (zfs_secpolicy_write_perms(zc->zc_name,
721             ZFS_DELEG_PERM_SEND, cr));
722 }

724 /* ARGSUSED */
725 static int
726 zfs_secpolicy_deleg_share(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
727 {
728         vnode_t *vp;
729         int error;

731         if ((error = lookupname(zc->zc_value, UIO_SYSSPACE,
732             NO_FOLLOW, NULL, &vp)) != 0)
733                 return (error);

735         /* Now make sure mntpnt and dataset are ZFS */

737         if (vp->v_vfsp->vfs_fstype != zfsfstype ||
738             (strcmp((char *)refstr_value(vp->v_vfsp->vfs_resource),
739             zc->zc_name) != 0)) {
740                 VN_RELE(vp);
741                 return (EPERM);
742         }

744         VN_RELE(vp);
745         return (dsl_deleg_access(zc->zc_name,
746             ZFS_DELEG_PERM_SHARE, cr));
747 }

749 int
750 zfs_secpolicy_share(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
751 {
752         if (!INGLOBALZONE(curproc))
753                 return (EPERM);

755         if (secpolicy_nfs(cr) == 0) {
756                 return (0);
757         } else {
758                 return (zfs_secpolicy_deleg_share(zc, innvl, cr));
759         }
760 }

762 int
763 zfs_secpolicy_smb_acl(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
764 {
765         if (!INGLOBALZONE(curproc))
766                 return (EPERM);

768         if (secpolicy_smb(cr) == 0) {
769                 return (0);
770         } else {
771                 return (zfs_secpolicy_deleg_share(zc, innvl, cr));
772         }
773 }

775 static int
776 zfs_get_parent(const char *datasetname, char *parent, int parentsize)
777 {
778         char *cp;

780         /*
781          * Remove the @bla or /bla from the end of the name to get the parent.
782          */
783         (void) strncpy(parent, datasetname, parentsize);
```

```
784         cp = strrchr(parent, '@');
785         if (cp != NULL) {
786                 cp[0] = '\0';
787         } else {
788                 cp = strrchr(parent, '/');
789                 if (cp == NULL)
790                         return (ENOENT);
791                 cp[0] = '\0';
792         }
794         return (0);
795 }
797 int
798 zfs_secpolicy_destroy_perms(const char *name, cred_t *cr)
799 {
800         int error;
802         if ((error = zfs_secpolicy_write_perms(name,
803             ZFS_DELEG_PERM_MOUNT, cr)) != 0)
804                 return (error);
806         return (zfs_secpolicy_write_perms(name, ZFS_DELEG_PERM_DESTROY, cr));
807 }
809 /* ARGSUSED */
810 static int
811 zfs_secpolicy_destroy(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
812 {
813         return (zfs_secpolicy_destroy_perms(zc->zc_name, cr));
814 }
816 /*
817  * Destroying snapshots with delegated permissions requires
818  * descendant mount and destroy permissions.
819  */
820 /* ARGSUSED */
821 static int
822 zfs_secpolicy_destroy_snaps(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
823 {
824         nvlist_t *snaps;
825         nvpair_t *pair, *nextpair;
826         int error = 0;
828         if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
829                 return (EINVAL);
830         for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
831             pair = nextpair) {
832                 dsl_dataset_t *ds;
834                 nextpair = nvlist_next_nvpair(snaps, pair);
835                 error = dsl_dataset_hold(nvpair_name(pair), FTAG, &ds);
836                 if (error == 0) {
837                         dsl_dataset_rele(ds, FTAG);
838                 } else if (error == ENOENT) {
839                         /*
840                          * Ignore any snapshots that don't exist (we consider
841                          * them "already destroyed").  Remove the name from the
842                          * nvl here in case the snapshot is created between
843                          * now and when we try to destroy it (in which case
844                          * we don't want to destroy it since we haven't
845                          * checked for permission).
846                          */
847                         fnvlist_remove_nvpair(snaps, pair);
848                         error = 0;
849                         continue;
```



```
850                 } else {
851                         break;
852                 }
853                 error = zfs_secpolicy_destroy_perms(nvpair_name(pair), cr);
854                 if (error != 0)
855                         break;
856         }
858         return (error);
859 }
861 int
862 zfs_secpolicy_rename_perms(const char *from, const char *to, cred_t *cr)
863 {
864         char    parentname[MAXNAMELEN];
865         int     error;
867         if ((error = zfs_secpolicy_write_perms(from,
868             ZFS_DELEG_PERM_RENAME, cr)) != 0)
869                 return (error);
871         if ((error = zfs_secpolicy_write_perms(from,
872             ZFS_DELEG_PERM_MOUNT, cr)) != 0)
873                 return (error);
875         if ((error = zfs_get_parent(to, parentname,
876             sizeof (parentname))) != 0)
877                 return (error);
879         if ((error = zfs_secpolicy_write_perms(parentname,
880             ZFS_DELEG_PERM_CREATE, cr)) != 0)
881                 return (error);
883         if ((error = zfs_secpolicy_write_perms(parentname,
884             ZFS_DELEG_PERM_MOUNT, cr)) != 0)
885                 return (error);
887         return (error);
888 }
890 /* ARGSUSED */
891 static int
892 zfs_secpolicy_rename(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
893 {
894         return (zfs_secpolicy_rename_perms(zc->zc_name, zc->zc_value, cr));
895 }
897 /* ARGSUSED */
898 static int
899 zfs_secpolicy_promote(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
900 {
901         char    parentname[MAXNAMELEN];
902         objset_t *clone;
903         int error;
905         error = zfs_secpolicy_write_perms(zc->zc_name,
906             ZFS_DELEG_PERM_PROMOTE, cr);
907         if (error)
908                 return (error);
910         error = dmu_objset_hold(zc->zc_name, FTAG, &clone);
912         if (error == 0) {
913                 dsl_dataset_t *pclone = NULL;
914                 dsl_dir_t *dd;
915                 dd = clone->os_dsl_dataset->ds_dir;
```

```
917                 rw_enter(&dd->dd_pool->dp_config_rwlock, RW_READER);
918                 error = dsl_dataset_hold_obj(dd->dd_pool,
919                     dd->dd_phys->dd_origin_obj, FTAG, &pclone);
920                 rw_exit(&dd->dd_pool->dp_config_rwlock);
921                 if (error) {
922                         dmu_objset_rele(clone, FTAG);
923                         return (error);
924                 }
926                 error = zfs_secpolicy_write_perms(zc->zc_name,
927                     ZFS_DELEG_PERM_MOUNT, cr);
929                 dsl_dataset_name(pclone, parentname);
930                 dmu_objset_rele(clone, FTAG);
931                 dsl_dataset_rele(pclone, FTAG);
932                 if (error == 0)
933                         error = zfs_secpolicy_write_perms(parentname,
934                             ZFS_DELEG_PERM_PROMOTE, cr);
935         }
936         return (error);
937 }
939 /* ARGSUSED */
940 static int
941 zfs_secpolicy_recv(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
942 {
943         int error;
945         if ((error = zfs_secpolicy_write_perms(zc->zc_name,
946             ZFS_DELEG_PERM_RECEIVE, cr)) != 0)
947                 return (error);
949         if ((error = zfs_secpolicy_write_perms(zc->zc_name,
950             ZFS_DELEG_PERM_MOUNT, cr)) != 0)
951                 return (error);
953         return (zfs_secpolicy_write_perms(zc->zc_name,
954             ZFS_DELEG_PERM_CREATE, cr));
955 }
957 int
958 zfs_secpolicy_snapshot_perms(const char *name, cred_t *cr)
959 {
960         return (zfs_secpolicy_write_perms(name,
961             ZFS_DELEG_PERM_SNAPSHOT, cr));
962 }
964 /*
965  * Check for permission to create each snapshot in the nvlist.
966  */
967 /* ARGSUSED */
968 static int
969 zfs_secpolicy_snapshot(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
970 {
971         nvlist_t *snaps;
972         int error;
973         nvpair_t *pair;
975         if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
976                 return (EINVAL);
977         for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
978             pair = nvlist_next_nvpair(snaps, pair)) {
979                 char *name = nvpair_name(pair);
980                 char *atp = strchr(name, '@');
```



```
982                 if (atp == NULL) {
983                         error = EINVAL;
984                         break;
985                 }
986                 *atp = '\0';
987                 error = zfs_secpolicy_snapshot_perms(name, cr);
988                 *atp = '@';
989                 if (error != 0)
990                         break;
991         }
992         return (error);
993 }
995 /* ARGSUSED */
996 static int
997 zfs_secpolicy_log_history(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
998 {
999         /*
1000          * Even root must have a proper TSD so that we know what pool
1001          * to log to.
1002          */
1003         if (tsd_get(zfs_allow_log_key) == NULL)
1004                 return (EPERM);
1005         return (0);
1006 }
1008 static int
1009 zfs_secpolicy_create_clone(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1010 {
1011         char    parentname[MAXNAMELEN];
1012         int     error;
1013         char    *origin;
1015         if ((error = zfs_get_parent(zc->zc_name, parentname,
1016             sizeof (parentname))) != 0)
1017                 return (error);
1019         if (nvlist_lookup_string(innvl, "origin", &origin) == 0 &&
1020             (error = zfs_secpolicy_write_perms(origin,
1021             ZFS_DELEG_PERM_CLONE, cr)) != 0)
1022                 return (error);
1024         if ((error = zfs_secpolicy_write_perms(parentname,
1025             ZFS_DELEG_PERM_CREATE, cr)) != 0)
1026                 return (error);
1028         return (zfs_secpolicy_write_perms(parentname,
1029             ZFS_DELEG_PERM_MOUNT, cr));
1030 }
1032 /*
1033  * Policy for pool operations - create/destroy pools, add vdevs, etc.  Requires
1034  * SYS_CONFIG privilege, which is not available in a local zone.
1035  */
1036 /* ARGSUSED */
1037 static int
1038 zfs_secpolicy_config(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
1039 {
1040         if (secpolicy_sys_config(cr, B_FALSE) != 0)
1041                 return (EPERM);
1043         return (0);
1044 }
1046 /*
1047  * Policy for object to name lookups.
```

```
 */
/* ARGSUSED */
static int
zfs_secpolicy_diff(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        int error;

        if ((error = secpolicy_sys_config(cr, B_FALSE)) == 0)
                return (0);

        error = zfs_secpolicy_write_perms(zc->zc_name, ZFS_DELEG_PERM_DIFF, cr);
        return (error);
}

/*
 * Policy for fault injection.  Requires all privileges.
 */
/* ARGSUSED */
static int
zfs_secpolicy_inject(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        return (secpolicy_zinject(cr));
}

/* ARGSUSED */
static int
zfs_secpolicy_inherit_prop(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        zfs_prop_t prop = zfs_name_to_prop(zc->zc_value);

        if (prop == ZPROP_INVAL) {
                if (!zfs_prop_user(zc->zc_value))
                        return (EINVAL);
                return (zfs_secpolicy_write_perms(zc->zc_name,
                    ZFS_DELEG_PERM_USERPROP, cr));
        } else {
                return (zfs_secpolicy_setprop(zc->zc_name, prop,
                    NULL, cr));
        }
}

static int
zfs_secpolicy_userspace_one(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        int err = zfs_secpolicy_read(zc, innvl, cr);
        if (err)
                return (err);

        if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
                return (EINVAL);

        if (zc->zc_value[0] == 0) {
                /*
                 * They are asking about a posix uid/gid.  If it's
                 * themself, allow it.
                 */
                if (zc->zc_objset_type == ZFS_PROP_USERUSED ||
                    zc->zc_objset_type == ZFS_PROP_USERQUOTA) {
                        if (zc->zc_guid == crgetuid(cr))
                                return (0);
                } else {
                        if (groupmember(zc->zc_guid, cr))
                                return (0);
                }
        }
```



```
        return (zfs_secpolicy_write_perms(zc->zc_name,
            userquota_perms[zc->zc_objset_type], cr));
}

static int
zfs_secpolicy_userspace_many(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        int err = zfs_secpolicy_read(zc, innvl, cr);
        if (err)
                return (err);

        if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
                return (EINVAL);

        return (zfs_secpolicy_write_perms(zc->zc_name,
            userquota_perms[zc->zc_objset_type], cr));
}

/* ARGSUSED */
static int
zfs_secpolicy_userspace_upgrade(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        return (zfs_secpolicy_setprop(zc->zc_name, ZFS_PROP_VERSION,
            NULL, cr));
}

/* ARGSUSED */
static int
zfs_secpolicy_hold(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        return (zfs_secpolicy_write_perms(zc->zc_name,
            ZFS_DELEG_PERM_HOLD, cr));
}

/* ARGSUSED */
static int
zfs_secpolicy_release(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        return (zfs_secpolicy_write_perms(zc->zc_name,
            ZFS_DELEG_PERM_RELEASE, cr));
}

/*
 * Policy for allowing temporary snapshots to be taken or released
 */
static int
zfs_secpolicy_tmp_snapshot(zfs_cmd_t *zc, nvlist_t *innvl, cred_t *cr)
{
        /*
         * A temporary snapshot is the same as a snapshot,
         * hold, destroy and release all rolled into one.
         * Delegated diff alone is sufficient that we allow this.
         */
        int error;

        if ((error = zfs_secpolicy_write_perms(zc->zc_name,
            ZFS_DELEG_PERM_DIFF, cr)) == 0)
                return (0);

        error = zfs_secpolicy_snapshot_perms(zc->zc_name, cr);
        if (!error)
                error = zfs_secpolicy_hold(zc, innvl, cr);
        if (!error)
                error = zfs_secpolicy_release(zc, innvl, cr);
        if (!error)
                error = zfs_secpolicy_destroy(zc, innvl, cr);
```

```
1180         return (error);
1181 }

1183 /*
1184  * Returns the nvlist as specified by the user in the zfs_cmd_t.
1185  */
1186 static int
1187 get_nvlist(uint64_t nvl, uint64_t size, int iflag, nvlist_t **nvp)
1188 {
1189         char *packed;
1190         int error;
1191         nvlist_t *list = NULL;

1193         /*
1194          * Read in and unpack the user-supplied nvlist.
1195          */
1196         if (size == 0)
1197                 return (EINVAL);

1199         packed = kmem_alloc(size, KM_SLEEP);

1201         if ((error = ddi_copyin((void *)(uintptr_t)nvl, packed, size,
1202             iflag)) != 0) {
1203                 kmem_free(packed, size);
1204                 return (error);
1205         }

1207         if ((error = nvlist_unpack(packed, size, &list, 0)) != 0) {
1208                 kmem_free(packed, size);
1209                 return (error);
1210         }

1212         kmem_free(packed, size);

1214         *nvp = list;
1215         return (0);
1216 }

1218 /*
1219  * Reduce the size of this nvlist until it can be serialized in 'max' bytes.
1220  * Entries will be removed from the end of the nvlist, and one int32 entry
1221  * named "N_MORE_ERRORS" will be added indicating how many entries were
1222  * removed.
1223  */
1224 static int
1225 nvlist_smush(nvlist_t *errors, size_t max)
1226 {
1227         size_t size;

1229         size = fnvlist_size(errors);

1231         if (size > max) {
1232                 nvpair_t *more_errors;
1233                 int n = 0;

1235                 if (max < 1024)
1236                         return (ENOMEM);

1238                 fnvlist_add_int32(errors, ZPROP_N_MORE_ERRORS, 0);
1239                 more_errors = nvlist_prev_nvpair(errors, NULL);

1241                 do {
1242                         nvpair_t *pair = nvlist_prev_nvpair(errors,
1243                             more_errors);
1244                         fnvlist_remove_nvpair(errors, pair);
1245                         n++;
```



```
1246                         size = fnvlist_size(errors);
1247                 } while (size > max);

1249                 fnvlist_remove_nvpair(errors, more_errors);
1250                 fnvlist_add_int32(errors, ZPROP_N_MORE_ERRORS, n);
1251                 ASSERT3U(fnvlist_size(errors), <=, max);
1252         }

1254         return (0);
1255 }

1257 static int
1258 put_nvlist(zfs_cmd_t *zc, nvlist_t *nvl)
1259 {
1260         char *packed = NULL;
1261         int error = 0;
1262         size_t size;

1264         size = fnvlist_size(nvl);

1266         if (size > zc->zc_nvlist_dst_size) {
1267                 error = ENOMEM;
1268         } else {
1269                 packed = fnvlist_pack(nvl, &size);
1270                 if (ddi_copyout(packed, (void *)(uintptr_t)zc->zc_nvlist_dst,
1271                     size, zc->zc_iflags) != 0)
1272                         error = EFAULT;
1273                 fnvlist_pack_free(packed, size);
1274         }

1276         zc->zc_nvlist_dst_size = size;
1277         zc->zc_nvlist_dst_filled = B_TRUE;
1278         return (error);
1279 }

1281 static int
1282 getzfsvfs(const char *dsname, zfsvfs_t **zfvp)
1283 {
1284         objset_t *os;
1285         int error;

1287         error = dmu_objset_hold(dsname, FTAG, &os);
1288         if (error)
1289                 return (error);
1290         if (dmu_objset_type(os) != DMU_OST_ZFS) {
1291                 dmu_objset_rele(os, FTAG);
1292                 return (EINVAL);
1293         }

1295         mutex_enter(&os->os_user_ptr_lock);
1296         *zfvp = dmu_objset_get_user(os);
1297         if (*zfvp) {
1298                 VFS_HOLD((*zfvp)->z_vfs);
1299         } else {
1300                 error = ESRCH;
1301         }
1302         mutex_exit(&os->os_user_ptr_lock);
1303         dmu_objset_rele(os, FTAG);
1304         return (error);
1305 }

1307 /*
1308  * Find a zfsvfs_t for a mounted filesystem, or create our own, in which
1309  * case its z_vfs will be NULL, and it will be opened as the owner.
1310  * If 'writer' is set, the z_teardown_lock will be held for RW_WRITER,
1311  * which prevents all vnode ops from running.
```

```
 */
static int
zfsvfs_hold(const char *name, void *tag, zfsvfs_t **zfvp, boolean_t writer)
{
        int error = 0;

        if (getzfsvfs(name, zfvp) != 0)
                error = zfsvfs_create(name, zfvp);
        if (error == 0) {
                rrw_enter(&(*zfvp)->z_teardown_lock, (writer) ? RW_WRITER :
                    RW_READER, tag);
                if ((*zfvp)->z_unmounted) {
                        /*
                         * XXX we could probably try again, since the unmounting
                         * thread should be just about to disassociate the
                         * objset from the zfsvfs.
                         */
                        rrw_exit(&(*zfvp)->z_teardown_lock, tag);
                        return (EBUSY);
                }
        }
        return (error);
}

static void
zfsvfs_rele(zfsvfs_t *zfsvfs, void *tag)
{
        rrw_exit(&zfsvfs->z_teardown_lock, tag);

        if (zfsvfs->z_vfs) {
                VFS_RELE(zfsvfs->z_vfs);
        } else {
                dmu_objset_disown(zfsvfs->z_os, zfsvfs);
                zfsvfs_free(zfsvfs);
        }
}

static int
zfs_ioc_pool_create(zfs_cmd_t *zc)
{
        int error;
        nvlist_t *config, *props = NULL;
        nvlist_t *rootprops = NULL;
        nvlist_t *zplprops = NULL;

        if (error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
            zc->zc_iflags, &config))
                return (error);

        if (zc->zc_nvlist_src_size != 0 && (error =
            get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
            zc->zc_iflags, &props))) {
                nvlist_free(config);
                return (error);
        }

        if (props) {
                nvlist_t *nvl = NULL;
                uint64_t version = SPA_VERSION;

                (void) nvlist_lookup_uint64(props,
                    zpool_prop_to_name(ZPOOL_PROP_VERSION), &version);
                if (!SPA_VERSION_IS_SUPPORTED(version)) {
                        error = EINVAL;
                        goto pool_props_bad;
                }
```



```
                (void) nvlist_lookup_nvlist(props, ZPOOL_ROOTFS_PROPS, &nvl);
                if (nvl) {
                        error = nvlist_dup(nvl, &rootprops, KM_SLEEP);
                        if (error != 0) {
                                nvlist_free(config);
                                nvlist_free(props);
                                return (error);
                        }
                        (void) nvlist_remove_all(props, ZPOOL_ROOTFS_PROPS);
                }
                VERIFY(nvlist_alloc(&zplprops, NV_UNIQUE_NAME, KM_SLEEP) == 0);
                error = zfs_fill_zplprops_root(version, rootprops,
                    zplprops, NULL);
                if (error)
                        goto pool_props_bad;
        }

        error = spa_create(zc->zc_name, config, props, zplprops);

        /*
         * Set the remaining root properties
         */
        if (!error && (error = zfs_set_prop_nvlist(zc->zc_name,
            ZPROP_SRC_LOCAL, rootprops, NULL)) != 0)
                (void) spa_destroy(zc->zc_name);

pool_props_bad:
        nvlist_free(rootprops);
        nvlist_free(zplprops);
        nvlist_free(config);
        nvlist_free(props);

        return (error);
}

static int
zfs_ioc_pool_destroy(zfs_cmd_t *zc)
{
        int error;
        zfs_log_history(zc);
        error = spa_destroy(zc->zc_name);
        if (error == 0)
                zvol_remove_minors(zc->zc_name);
        return (error);
}

static int
zfs_ioc_pool_import(zfs_cmd_t *zc)
{
        nvlist_t *config, *props = NULL;
        uint64_t guid;
        int error;

        if ((error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
            zc->zc_iflags, &config)) != 0)
                return (error);

        if (zc->zc_nvlist_src_size != 0 && (error =
            get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
            zc->zc_iflags, &props))) {
                nvlist_free(config);
                return (error);
        }

        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID, &guid) != 0 ||
            guid != zc->zc_guid)
```

```
1444                 error = EINVAL;
1445         else
1446                 error = spa_import(zc->zc_name, config, props, zc->zc_cookie);

1448         if (zc->zc_nvlist_dst != 0) {
1449                 int err;

1451                 if ((err = put_nvlist(zc, config)) != 0)
1452                         error = err;
1453         }

1455         nvlist_free(config);

1457         if (props)
1458                 nvlist_free(props);

1460         return (error);
1461 }

1463 static int
1464 zfs_ioc_pool_export(zfs_cmd_t *zc)
1465 {
1466         int error;
1467         boolean_t force = (boolean_t)zc->zc_cookie;
1468         boolean_t hardforce = (boolean_t)zc->zc_guid;

1470         zfs_log_history(zc);
1471         error = spa_export(zc->zc_name, NULL, force, hardforce);
1472         if (error == 0)
1473                 zvol_remove_minors(zc->zc_name);
1474         return (error);
1475 }

1477 static int
1478 zfs_ioc_pool_configs(zfs_cmd_t *zc)
1479 {
1480         nvlist_t *configs;
1481         int error;

1483         if ((configs = spa_all_configs(&zc->zc_cookie)) == NULL)
1484                 return (EEXIST);

1486         error = put_nvlist(zc, configs);

1488         nvlist_free(configs);

1490         return (error);
1491 }

1493 /*
1494  * inputs:
1495  * zc_name              name of the pool
1496  *
1497  * outputs:
1498  * zc_cookie            real errno
1499  * zc_nvlist_dst        config nvlist
1500  * zc_nvlist_dst_size   size of config nvlist
1501  */
1502 static int
1503 zfs_ioc_pool_stats(zfs_cmd_t *zc)
1504 {
1505         nvlist_t *config;
1506         int error;
1507         int ret = 0;

1509         error = spa_get_stats(zc->zc_name, &config, zc->zc_value,
```



```
1510             sizeof (zc->zc_value));

1512         if (config != NULL) {
1513                 ret = put_nvlist(zc, config);
1514                 nvlist_free(config);

1516                 /*
1517                  * The config may be present even if 'error' is non-zero.
1518                  * In this case we return success, and preserve the real errno
1519                  * in 'zc_cookie'.
1520                  */
1521                 zc->zc_cookie = error;
1522         } else {
1523                 ret = error;
1524         }

1526         return (ret);
1527 }

1529 /*
1530  * Try to import the given pool, returning pool stats as appropriate so that
1531  * user land knows which devices are available and overall pool health.
1532  */
1533 static int
1534 zfs_ioc_pool_tryimport(zfs_cmd_t *zc)
1535 {
1536         nvlist_t *tryconfig, *config;
1537         int error;

1539         if ((error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1540             zc->zc_iflags, &tryconfig)) != 0)
1541                 return (error);

1543         config = spa_tryimport(tryconfig);

1545         nvlist_free(tryconfig);

1547         if (config == NULL)
1548                 return (EINVAL);

1550         error = put_nvlist(zc, config);
1551         nvlist_free(config);

1553         return (error);
1554 }

1556 /*
1557  * inputs:
1558  * zc_name              name of the pool
1559  * zc_cookie            scan func (pool_scan_func_t)
1560  */
1561 static int
1562 zfs_ioc_pool_scan(zfs_cmd_t *zc)
1563 {
1564         spa_t *spa;
1565         int error;

1567         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1568                 return (error);

1570         if (zc->zc_cookie == POOL_SCAN_NONE)
1571                 error = spa_scan_stop(spa);
1572         else
1573                 error = spa_scan(spa, zc->zc_cookie);

1575         spa_close(spa, FTAG);
```

```
1577         return (error);
1578 }

1580 static int
1581 zfs_ioc_pool_freeze(zfs_cmd_t *zc)
1582 {
1583         spa_t *spa;
1584         int error;

1586         error = spa_open(zc->zc_name, &spa, FTAG);
1587         if (error == 0) {
1588                 spa_freeze(spa);
1589                 spa_close(spa, FTAG);
1590         }
1591         return (error);
1592 }

1594 static int
1595 zfs_ioc_pool_upgrade(zfs_cmd_t *zc)
1596 {
1597         spa_t *spa;
1598         int error;

1600         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1601                 return (error);

1603         if (zc->zc_cookie < spa_version(spa) ||
1604             !SPA_VERSION_IS_SUPPORTED(zc->zc_cookie)) {
1605                 spa_close(spa, FTAG);
1606                 return (EINVAL);
1607         }

1609         spa_upgrade(spa, zc->zc_cookie);
1610         spa_close(spa, FTAG);

1612         return (error);
1613 }

1615 static int
1616 zfs_ioc_pool_get_history(zfs_cmd_t *zc)
1617 {
1618         spa_t *spa;
1619         char *hist_buf;
1620         uint64_t size;
1621         int error;

1623         if ((size = zc->zc_history_len) == 0)
1624                 return (EINVAL);

1626         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1627                 return (error);

1629         if (spa_version(spa) < SPA_VERSION_ZPOOL_HISTORY) {
1630                 spa_close(spa, FTAG);
1631                 return (ENOTSUP);
1632         }

1634         hist_buf = kmem_alloc(size, KM_SLEEP);
1635         if ((error = spa_history_get(spa, &zc->zc_history_offset,
1636             &zc->zc_history_len, hist_buf)) == 0) {
1637                 error = ddi_copyout(hist_buf,
1638                     (void *)(uintptr_t)zc->zc_history,
1639                     zc->zc_history_len, zc->zc_iflags);
1640         }
```



```
1642         spa_close(spa, FTAG);
1643         kmem_free(hist_buf, size);
1644         return (error);
1645 }

1647 static int
1648 zfs_ioc_pool_reguid(zfs_cmd_t *zc)
1649 {
1650         spa_t *spa;
1651         int error;

1653         error = spa_open(zc->zc_name, &spa, FTAG);
1654         if (error == 0) {
1655                 error = spa_change_guid(spa);
1656                 spa_close(spa, FTAG);
1657         }
1658         return (error);
1659 }

1661 static int
1662 zfs_ioc_dsobj_to_dsname(zfs_cmd_t *zc)
1663 {
1664         int error;

1666         if (error = dsl_dsobj_to_dsname(zc->zc_name, zc->zc_obj, zc->zc_value))
1667                 return (error);

1669         return (0);
1670 }

1672 /*
1673  * inputs:
1674  * zc_name              name of filesystem
1675  * zc_obj               object to find
1676  *
1677  * outputs:
1678  * zc_value             name of object
1679  */
1680 static int
1681 zfs_ioc_obj_to_path(zfs_cmd_t *zc)
1682 {
1683         objset_t *os;
1684         int error;

1686         /* XXX reading from objset not owned */
1687         if ((error = dmu_objset_hold(zc->zc_name, FTAG, &os)) != 0)
1688                 return (error);
1689         if (dmu_objset_type(os) != DMU_OST_ZFS) {
1690                 dmu_objset_rele(os, FTAG);
1691                 return (EINVAL);
1692         }
1693         error = zfs_obj_to_path(os, zc->zc_obj, zc->zc_value,
1694             sizeof (zc->zc_value));
1695         dmu_objset_rele(os, FTAG);

1697         return (error);
1698 }

1700 /*
1701  * inputs:
1702  * zc_name              name of filesystem
1703  * zc_obj               object to find
1704  *
1705  * outputs:
1706  * zc_stat              stats on object
1707  * zc_value             path to object
```

```
 */
static int
zfs_ioc_obj_to_stats(zfs_cmd_t *zc)
{
        objset_t *os;
        int error;

        /* XXX reading from objset not owned */
        if ((error = dmu_objset_hold(zc->zc_name, FTAG, &os)) != 0)
                return (error);
        if (dmu_objset_type(os) != DMU_OST_ZFS) {
                dmu_objset_rele(os, FTAG);
                return (EINVAL);
        }
        error = zfs_obj_to_stats(os, zc->zc_obj, &zc->zc_stat, zc->zc_value,
            sizeof (zc->zc_value));
        dmu_objset_rele(os, FTAG);

        return (error);
}

static int
zfs_ioc_vdev_add(zfs_cmd_t *zc)
{
        spa_t *spa;
        int error;
        nvlist_t *config, **l2cache, **spares;
        uint_t nl2cache = 0, nspares = 0;

        error = spa_open(zc->zc_name, &spa, FTAG);
        if (error != 0)
                return (error);

        error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
            zc->zc_iflags, &config);
        (void) nvlist_lookup_nvlist_array(config, ZPOOL_CONFIG_L2CACHE,
            &l2cache, &nl2cache);

        (void) nvlist_lookup_nvlist_array(config, ZPOOL_CONFIG_SPARES,
            &spares, &nspares);

        /*
         * A root pool with concatenated devices is not supported.
         * Thus, can not add a device to a root pool.
         *
         * Intent log device can not be added to a rootpool because
         * during mountroot, zil is replayed, a seperated log device
         * can not be accessed during the mountroot time.
         *
         * l2cache and spare devices are ok to be added to a rootpool.
         */
        if (spa_bootfs(spa) != 0 && nl2cache == 0 && nspares == 0) {
                nvlist_free(config);
                spa_close(spa, FTAG);
                return (EDOM);
        }

        if (error == 0) {
                error = spa_vdev_add(spa, config);
                nvlist_free(config);
        }
        spa_close(spa, FTAG);
        return (error);
}

/*
```



```
 * inputs:
 * zc_name              name of the pool
 * zc_nvlist_conf       nvlist of devices to remove
 * zc_cookie            to stop the remove?
 */
static int
zfs_ioc_vdev_remove(zfs_cmd_t *zc)
{
        spa_t *spa;
        int error;

        error = spa_open(zc->zc_name, &spa, FTAG);
        if (error != 0)
                return (error);
        error = spa_vdev_remove(spa, zc->zc_guid, B_FALSE);
        spa_close(spa, FTAG);
        return (error);
}

static int
zfs_ioc_vdev_set_state(zfs_cmd_t *zc)
{
        spa_t *spa;
        int error;
        vdev_state_t newstate = VDEV_STATE_UNKNOWN;

        if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
                return (error);
        switch (zc->zc_cookie) {
        case VDEV_STATE_ONLINE:
                error = vdev_online(spa, zc->zc_guid, zc->zc_obj, &newstate);
                break;

        case VDEV_STATE_OFFLINE:
                error = vdev_offline(spa, zc->zc_guid, zc->zc_obj);
                break;

        case VDEV_STATE_FAULTED:
                if (zc->zc_obj != VDEV_AUX_ERR_EXCEEDED &&
                    zc->zc_obj != VDEV_AUX_EXTERNAL)
                        zc->zc_obj = VDEV_AUX_ERR_EXCEEDED;

                error = vdev_fault(spa, zc->zc_guid, zc->zc_obj);
                break;

        case VDEV_STATE_DEGRADED:
                if (zc->zc_obj != VDEV_AUX_ERR_EXCEEDED &&
                    zc->zc_obj != VDEV_AUX_EXTERNAL)
                        zc->zc_obj = VDEV_AUX_ERR_EXCEEDED;

                error = vdev_degrade(spa, zc->zc_guid, zc->zc_obj);
                break;

        default:
                error = EINVAL;
        }
        zc->zc_cookie = newstate;
        spa_close(spa, FTAG);
        return (error);
}

static int
zfs_ioc_vdev_attach(zfs_cmd_t *zc)
{
        spa_t *spa;
        int replacing = zc->zc_cookie;
```

```
1840         nvlist_t *config;
1841         int error;

1843         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1844                 return (error);

1846         if ((error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1847             zc->zc_iflags, &config)) == 0) {
1848                 error = spa_vdev_attach(spa, zc->zc_guid, config, replacing);
1849                 nvlist_free(config);
1850         }

1852         spa_close(spa, FTAG);
1853         return (error);
1854 }

1856 static int
1857 zfs_ioc_vdev_detach(zfs_cmd_t *zc)
1858 {
1859         spa_t *spa;
1860         int error;

1862         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1863                 return (error);

1865         error = spa_vdev_detach(spa, zc->zc_guid, 0, B_FALSE);

1867         spa_close(spa, FTAG);
1868         return (error);
1869 }

1871 static int
1872 zfs_ioc_vdev_split(zfs_cmd_t *zc)
1873 {
1874         spa_t *spa;
1875         nvlist_t *config, *props = NULL;
1876         int error;
1877         boolean_t exp = !!(zc->zc_cookie & ZPOOL_EXPORT_AFTER_SPLIT);

1879         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
1880                 return (error);

1882         if (error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1883             zc->zc_iflags, &config)) {
1884                 spa_close(spa, FTAG);
1885                 return (error);
1886         }

1888         if (zc->zc_nvlist_src_size != 0 && (error =
1889             get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
1890             zc->zc_iflags, &props))) {
1891                 spa_close(spa, FTAG);
1892                 nvlist_free(config);
1893                 return (error);
1894         }

1896         error = spa_vdev_split_mirror(spa, zc->zc_string, config, props, exp);

1898         spa_close(spa, FTAG);

1900         nvlist_free(config);
1901         nvlist_free(props);

1903         return (error);
1904 }
```



```
1906 static int
1907 zfs_ioc_vdev_setpath(zfs_cmd_t *zc)
1908 {
1909         spa_t *spa;
1910         char *path = zc->zc_value;
1911         uint64_t guid = zc->zc_guid;
1912         int error;

1914         error = spa_open(zc->zc_name, &spa, FTAG);
1915         if (error != 0)
1916                 return (error);

1918         error = spa_vdev_setpath(spa, guid, path);
1919         spa_close(spa, FTAG);
1920         return (error);
1921 }

1923 static int
1924 zfs_ioc_vdev_setfru(zfs_cmd_t *zc)
1925 {
1926         spa_t *spa;
1927         char *fru = zc->zc_value;
1928         uint64_t guid = zc->zc_guid;
1929         int error;

1931         error = spa_open(zc->zc_name, &spa, FTAG);
1932         if (error != 0)
1933                 return (error);

1935         error = spa_vdev_setfru(spa, guid, fru);
1936         spa_close(spa, FTAG);
1937         return (error);
1938 }

1940 static int
1941 zfs_ioc_objset_stats_impl(zfs_cmd_t *zc, objset_t *os)
1942 {
1943         int error = 0;
1944         nvlist_t *nv;

1946         dmu_objset_fast_stat(os, &zc->zc_objset_stats);

1948         if (zc->zc_nvlist_dst != 0 &&
1949             (error = dsl_prop_get_all(os, &nv)) == 0) {
1950                 dmu_objset_stats(os, nv);
1951                 /*
1952                  * NB: zvol_get_stats() will read the objset contents,
1953                  * which we aren't supposed to do with a
1954                  * DS_MODE_USER hold, because it could be
1955                  * inconsistent.  So this is a bit of a workaround...
1956                  * XXX reading with out owning
1957                  */
1958                 if (!zc->zc_objset_stats.dds_inconsistent &&
1959                     dmu_objset_type(os) == DMU_OST_ZVOL) {
1960                         error = zvol_get_stats(os, nv);
1961                         if (error == EIO)
1962                                 return (error);
1963                         VERIFY0(error);
1964                 }
1965                 error = put_nvlist(zc, nv);
1966                 nvlist_free(nv);
1967         }

1969         return (error);
1970 }
```

```
1972 /*
1973  * inputs:
1974  * zc_name              name of filesystem
1975  * zc_nvlist_dst_size   size of buffer for property nvlist
1976  *
1977  * outputs:
1978  * zc_objset_stats      stats
1979  * zc_nvlist_dst        property nvlist
1980  * zc_nvlist_dst_size   size of property nvlist
1981  */
1982 static int
1983 zfs_ioc_objset_stats(zfs_cmd_t *zc)
1984 {
1985         objset_t *os = NULL;
1986         int error;

1988         if (error = dmu_objset_hold(zc->zc_name, FTAG, &os))
1989                 return (error);

1991         error = zfs_ioc_objset_stats_impl(zc, os);

1993         dmu_objset_rele(os, FTAG);

1995         return (error);
1996 }

1998 /*
1999  * inputs:
2000  * zc_name              name of filesystem
2001  * zc_nvlist_dst_size   size of buffer for property nvlist
2002  *
2003  * outputs:
2004  * zc_nvlist_dst        received property nvlist
2005  * zc_nvlist_dst_size   size of received property nvlist
2006  *
2007  * Gets received properties (distinct from local properties on or after
2008  * SPA_VERSION_RECVD_PROPS) for callers who want to differentiate received from
2009  * local property values.
2010  */
2011 static int
2012 zfs_ioc_objset_recvd_props(zfs_cmd_t *zc)
2013 {
2014         objset_t *os = NULL;
2015         int error;
2016         nvlist_t *nv;

2018         if (error = dmu_objset_hold(zc->zc_name, FTAG, &os))
2019                 return (error);

2021         /*
2022          * Without this check, we would return local property values if the
2023          * caller has not already received properties on or after
2024          * SPA_VERSION_RECVD_PROPS.
2025          */
2026         if (!dsl_prop_get_hasrecvd(os)) {
2027                 dmu_objset_rele(os, FTAG);
2028                 return (ENOTSUP);
2029         }

2031         if (zc->zc_nvlist_dst != 0 &&
2032             (error = dsl_prop_get_received(os, &nv)) == 0) {
2033                 error = put_nvlist(zc, nv);
2034                 nvlist_free(nv);
2035         }

2037         dmu_objset_rele(os, FTAG);
```



```
2038         return (error);
2039 }

2041 static int
2042 nvl_add_zplprop(objset_t *os, nvlist_t *props, zfs_prop_t prop)
2043 {
2044         uint64_t value;
2045         int error;

2047         /*
2048          * zfs_get_zplprop() will either find a value or give us
2049          * the default value (if there is one).
2050          */
2051         if ((error = zfs_get_zplprop(os, prop, &value)) != 0)
2052                 return (error);
2053         VERIFY(nvlist_add_uint64(props, zfs_prop_to_name(prop), value) == 0);
2054         return (0);
2055 }

2057 /*
2058  * inputs:
2059  * zc_name              name of filesystem
2060  * zc_nvlist_dst_size   size of buffer for zpl property nvlist
2061  *
2062  * outputs:
2063  * zc_nvlist_dst        zpl property nvlist
2064  * zc_nvlist_dst_size   size of zpl property nvlist
2065  */
2066 static int
2067 zfs_ioc_objset_zplprops(zfs_cmd_t *zc)
2068 {
2069         objset_t *os;
2070         int err;

2072         /* XXX reading without owning */
2073         if (err = dmu_objset_hold(zc->zc_name, FTAG, &os))
2074                 return (err);

2076         dmu_objset_fast_stat(os, &zc->zc_objset_stats);

2078         /*
2079          * NB: nvl_add_zplprop() will read the objset contents,
2080          * which we aren't supposed to do with a DS_MODE_USER
2081          * hold, because it could be inconsistent.
2082          */
2083         if (zc->zc_nvlist_dst != NULL &&
2084             !zc->zc_objset_stats.dds_inconsistent &&
2085             dmu_objset_type(os) == DMU_OST_ZFS) {
2086                 nvlist_t *nv;

2088                 VERIFY(nvlist_alloc(&nv, NV_UNIQUE_NAME, KM_SLEEP) == 0);
2089                 if ((err = nvl_add_zplprop(os, nv, ZFS_PROP_VERSION)) == 0 &&
2090                     (err = nvl_add_zplprop(os, nv, ZFS_PROP_NORMALIZE)) == 0 &&
2091                     (err = nvl_add_zplprop(os, nv, ZFS_PROP_UTF8ONLY)) == 0 &&
2092                     (err = nvl_add_zplprop(os, nv, ZFS_PROP_CASE)) == 0)
2093                         err = put_nvlist(zc, nv);
2094                 nvlist_free(nv);
2095         } else {
2096                 err = ENOENT;
2097         }
2098         dmu_objset_rele(os, FTAG);
2099         return (err);
2100 }

2102 static boolean_t
2103 dataset_name_hidden(const char *name)
```

```
2104 {
2105         /*
2106          * Skip over datasets that are not visible in this zone,
2107          * internal datasets (which have a $ in their name), and
2108          * temporary datasets (which have a % in their name).
2109          */
2110         if (strchr(name, '$') != NULL)
2111                 return (B_TRUE);
2112         if (strchr(name, '%') != NULL)
2113                 return (B_TRUE);
2114         if (!INGLOBALZONE(curproc) && !zone_dataset_visible(name, NULL))
2115                 return (B_TRUE);
2116         return (B_FALSE);
2117 }

2119 /*
2120  * inputs:
2121  * zc_name              name of filesystem
2122  * zc_cookie            zap cursor
2123  * zc_nvlist_dst_size   size of buffer for property nvlist
2124  *
2125  * outputs:
2126  * zc_name              name of next filesystem
2127  * zc_cookie            zap cursor
2128  * zc_objset_stats      stats
2129  * zc_nvlist_dst        property nvlist
2130  * zc_nvlist_dst_size   size of property nvlist
2131  */
2132 static int
2133 zfs_ioc_dataset_list_next(zfs_cmd_t *zc)
2134 {
2135         objset_t *os;
2136         int error;
2137         char *p;
2138         size_t orig_len = strlen(zc->zc_name);

2140 top:
2141         if (error = dmu_objset_hold(zc->zc_name, FTAG, &os)) {
2142                 if (error == ENOENT)
2143                         error = ESRCH;
2144                 return (error);
2145         }

2147         p = strrchr(zc->zc_name, '/');
2148         if (p == NULL || p[1] != '\0')
2149                 (void) strlcat(zc->zc_name, "/", sizeof (zc->zc_name));
2150         p = zc->zc_name + strlen(zc->zc_name);

2152         /*
2153          * Pre-fetch the datasets.  dmu_objset_prefetch() always returns 0
2154          * but is not declared void because its called by dmu_objset_find().
2155          */
2156         if (zc->zc_cookie == 0) {
2157                 uint64_t cookie = 0;
2158                 int len = sizeof (zc->zc_name) - (p - zc->zc_name);

2160                 while (dmu_dir_list_next(os, len, p, NULL, &cookie) == 0) {
2161                         if (!dataset_name_hidden(zc->zc_name))
2162                                 (void) dmu_objset_prefetch(zc->zc_name, NULL);
2163                 }
2164         }

2166         do {
2167                 error = dmu_dir_list_next(os,
2168                     sizeof (zc->zc_name) - (p - zc->zc_name), p,
2169                     NULL, &zc->zc_cookie);
```



```
2170                 if (error == ENOENT)
2171                         error = ESRCH;
2172         } while (error == 0 && dataset_name_hidden(zc->zc_name));
2173         dmu_objset_rele(os, FTAG);

2175         /*
2176          * If it's an internal dataset (ie. with a '$' in its name),
2177          * don't try to get stats for it, otherwise we'll return ENOENT.
2178          */
2179         if (error == 0 && strchr(zc->zc_name, '$') == NULL) {
2180                 error = zfs_ioc_objset_stats(zc); /* fill in the stats */
2181                 if (error == ENOENT) {
2182                         /* We lost a race with destroy, get the next one. */
2183                         zc->zc_name[orig_len] = '\0';
2184                         goto top;
2185                 }
2186         }
2187         return (error);
2188 }

2190 /*
2191  * inputs:
2192  * zc_name              name of filesystem
2193  * zc_cookie            zap cursor
2194  * zc_nvlist_dst_size   size of buffer for property nvlist
2195  *
2196  * outputs:
2197  * zc_name              name of next snapshot
2198  * zc_objset_stats      stats
2199  * zc_nvlist_dst        property nvlist
2200  * zc_nvlist_dst_size   size of property nvlist
2201  */
2202 static int
2203 zfs_ioc_snapshot_list_next(zfs_cmd_t *zc)
2204 {
2205         objset_t *os;
2206         int error;

2208 top:
2209         if (zc->zc_cookie == 0)
2210                 (void) dmu_objset_find(zc->zc_name, dmu_objset_prefetch,
2211                     NULL, DS_FIND_SNAPSHOTS);

2213         error = dmu_objset_hold(zc->zc_name, FTAG, &os);
2214         if (error)
2215                 return (error == ENOENT ? ESRCH : error);

2217         /*
2218          * A dataset name of maximum length cannot have any snapshots,
2219          * so exit immediately.
2220          */
2221         if (strlcat(zc->zc_name, "@", sizeof (zc->zc_name)) >= MAXNAMELEN) {
2222                 dmu_objset_rele(os, FTAG);
2223                 return (ESRCH);
2224         }

2226         error = dmu_snapshot_list_next(os,
2227             sizeof (zc->zc_name) - strlen(zc->zc_name),
2228             zc->zc_name + strlen(zc->zc_name), &zc->zc_obj, &zc->zc_cookie,
2229             NULL);

2231         if (error == 0) {
2232                 dsl_dataset_t *ds;
2233                 dsl_pool_t *dp = os->os_dsl_dataset->ds_dir->dd_pool;

2235                 /*
```

```
2236                 * Since we probably don't have a hold on this snapshot,
2237                 * it's possible that the objsetid could have been destroyed
2238                 * and reused for a new objset. It's OK if this happens during
2239                 * a zfs send operation, since the new createtxg will be
2240                 * beyond the range we're interested in.
2241                 */
2242                rw_enter(&dp->dp_config_rwlock, RW_READER);
2243                error = dsl_dataset_hold_obj(dp, zc->zc_obj, FTAG, &ds);
2244                rw_exit(&dp->dp_config_rwlock);
2245                if (error) {
2246                        if (error == ENOENT) {
2247                                /* Racing with destroy, get the next one. */
2248                                *strchr(zc->zc_name, '@') = '\0';
2249                                dmu_objset_rele(os, FTAG);
2250                                goto top;
2251                        }
2252                } else {
2253                        objset_t *ossnap;

2255                        error = dmu_objset_from_ds(ds, &ossnap);
2256                        if (error == 0)
2257                                error = zfs_ioc_objset_stats_impl(zc, ossnap);
2258                        dsl_dataset_rele(ds, FTAG);
2259                }
2260        } else if (error == ENOENT) {
2261                error = ESRCH;
2262        }

2264        dmu_objset_rele(os, FTAG);
2265        /* if we failed, undo the @ that we tacked on to zc_name */
2266        if (error)
2267                *strchr(zc->zc_name, '@') = '\0';
2268        return (error);
2269 }

2271 static int
2272 zfs_prop_set_userquota(const char *dsname, nvpair_t *pair)
2273 {
2274        const char *propname = nvpair_name(pair);
2275        uint64_t *valary;
2276        unsigned int vallen;
2277        const char *domain;
2278        char *dash;
2279        zfs_userquota_prop_t type;
2280        uint64_t rid;
2281        uint64_t quota;
2282        zfsvfs_t *zfsvfs;
2283        int err;

2285        if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2286                nvlist_t *attrs;
2287                VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
2288                if (nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2289                    &pair) != 0)
2290                        return (EINVAL);
2291        }

2293        /*
2294         * A correctly constructed propname is encoded as
2295         * userquota@<rid>-<domain>.
2296         */
2297        if ((dash = strchr(propname, '-')) == NULL ||
2298            nvpair_value_uint64_array(pair, &valary, &vallen) != 0 ||
2299            vallen != 3)
2300                return (EINVAL);
```



```
2302        domain = dash + 1;
2303        type = valary[0];
2304        rid = valary[1];
2305        quota = valary[2];

2307        err = zfsvfs_hold(dsname, FTAG, &zfsvfs, B_FALSE);
2308        if (err == 0) {
2309                err = zfs_set_userquota(zfsvfs, type, domain, rid, quota);
2310                zfsvfs_rele(zfsvfs, FTAG);
2311        }

2313        return (err);
2314 }

2316 /*
2317  * If the named property is one that has a special function to set its value,
2318  * return 0 on success and a positive error code on failure; otherwise if it is
2319  * not one of the special properties handled by this function, return -1.
2320  *
2321  * XXX: It would be better for callers of the property interface if we handled
2322  * these special cases in dsl_prop.c (in the dsl layer).
2323  */
2324 static int
2325 zfs_prop_set_special(const char *dsname, zprop_source_t source,
2326     nvpair_t *pair)
2327 {
2328        const char *propname = nvpair_name(pair);
2329        zfs_prop_t prop = zfs_name_to_prop(propname);
2330        uint64_t intval;
2331        int err;

2333        if (prop == ZPROP_INVAL) {
2334                if (zfs_prop_userquota(propname))
2335                        return (zfs_prop_set_userquota(dsname, pair));
2336                return (-1);
2337        }

2339        if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2340                nvlist_t *attrs;
2341                VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
2342                VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2343                    &pair) == 0);
2344        }

2346        if (zfs_prop_get_type(prop) == PROP_TYPE_STRING)
2347                return (-1);

2349        VERIFY(0 == nvpair_value_uint64(pair, &intval));

2351        switch (prop) {
2352        case ZFS_PROP_QUOTA:
2353                err = dsl_dir_set_quota(dsname, source, intval);
2354                break;
2355        case ZFS_PROP_REFQUOTA:
2356                err = dsl_dataset_set_quota(dsname, source, intval);
2357                break;
2358        case ZFS_PROP_RESERVATION:
2359                err = dsl_dir_set_reservation(dsname, source, intval);
2360                break;
2361        case ZFS_PROP_REFRESERVATION:
2362                err = dsl_dataset_set_reservation(dsname, source, intval);
2363                break;
2364        case ZFS_PROP_VOLSIZE:
2365                err = zvol_set_volsize(dsname, ddi_driver_major(zfs_dip),
2366                    intval);
2367                break;
```

```
2368         case ZFS_PROP_VERSION:
2369         {
2370                 zfsvfs_t *zfsvfs;

2372                 if ((err = zfsvfs_hold(dsname, FTAG, &zfsvfs, B_TRUE)) != 0)
2373                         break;

2375                 err = zfs_set_version(zfsvfs, intval);
2376                 zfsvfs_rele(zfsvfs, FTAG);

2378                 if (err == 0 && intval >= ZPL_VERSION_USERSPACE) {
2379                         zfs_cmd_t *zc;

2381                         zc = kmem_zalloc(sizeof (zfs_cmd_t), KM_SLEEP);
2382                         (void) strcpy(zc->zc_name, dsname);
2383                         (void) zfs_ioc_userspace_upgrade(zc);
2384                         kmem_free(zc, sizeof (zfs_cmd_t));
2385                 }
2386                 break;
2387         }

2389         default:
2390                 err = -1;
2391         }

2393         return (err);
2394 }

2396 /*
2397  * This function is best effort. If it fails to set any of the given properties,
2398  * it continues to set as many as it can and returns the last error
2399  * encountered. If the caller provides a non-NULL errlist, it will be filled in
2400  * with the list of names of all the properties that failed along with the
2401  * corresponding error numbers.
2402  *
2403  * If every property is set successfully, zero is returned and errlist is not
2404  * modified.
2405  */
2406 int
2407 zfs_set_prop_nvlist(const char *dsname, zprop_source_t source, nvlist_t *nvl,
2408     nvlist_t *errlist)
2409 {
2410         nvpair_t *pair;
2411         nvpair_t *propval;
2412         int rv = 0;
2413         uint64_t intval;
2414         char *strval;
2415         nvlist_t *genericnvl = fnvlist_alloc();
2416         nvlist_t *retrynvl = fnvlist_alloc();

2418 retry:
2419         pair = NULL;
2420         while ((pair = nvlist_next_nvpair(nvl, pair)) != NULL) {
2421                 const char *propname = nvpair_name(pair);
2422                 zfs_prop_t prop = zfs_name_to_prop(propname);
2423                 int err = 0;

2425                 /* decode the property value */
2426                 propval = pair;
2427                 if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2428                         nvlist_t *attrs;
2429                         attrs = fnvpair_value_nvlist(pair);
2430                         if (nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2431                             &propval) != 0)
2432                                 err = EINVAL;
2433                 }
```



```
2435                 /* Validate value type */
2436                 if (err == 0 && prop == ZPROP_INVAL) {
2437                         if (zfs_prop_user(propname)) {
2438                                 if (nvpair_type(propval) != DATA_TYPE_STRING)
2439                                         err = EINVAL;
2440                         } else if (zfs_prop_userquota(propname)) {
2441                                 if (nvpair_type(propval) !=
2442                                     DATA_TYPE_UINT64_ARRAY)
2443                                         err = EINVAL;
2444                         } else {
2445                                 err = EINVAL;
2446                         }
2447                 } else if (err == 0) {
2448                         if (nvpair_type(propval) == DATA_TYPE_STRING) {
2449                                 if (zfs_prop_get_type(prop) != PROP_TYPE_STRING)
2450                                         err = EINVAL;
2451                         } else if (nvpair_type(propval) == DATA_TYPE_UINT64) {
2452                                 const char *unused;

2454                                 intval = fnvpair_value_uint64(propval);

2456                                 switch (zfs_prop_get_type(prop)) {
2457                                 case PROP_TYPE_NUMBER:
2458                                         break;
2459                                 case PROP_TYPE_STRING:
2460                                         err = EINVAL;
2461                                         break;
2462                                 case PROP_TYPE_INDEX:
2463                                         if (zfs_prop_index_to_string(prop,
2464                                             intval, &unused) != 0)
2465                                                 err = EINVAL;
2466                                         break;
2467                                 default:
2468                                         cmn_err(CE_PANIC,
2469                                             "unknown property type");
2470                                 }
2471                         } else {
2472                                 err = EINVAL;
2473                         }
2474                 }

2476                 /* Validate permissions */
2477                 if (err == 0)
2478                         err = zfs_check_settable(dsname, pair, CRED());

2480                 if (err == 0) {
2481                         err = zfs_prop_set_special(dsname, source, pair);
2482                         if (err == -1) {
2483                                 /*
2484                                  * For better performance we build up a list of
2485                                  * properties to set in a single transaction.
2486                                  */
2487                                 err = nvlist_add_nvpair(genericnvl, pair);
2488                         } else if (err != 0 && nvl != retrynvl) {
2489                                 /*
2490                                  * This may be a spurious error caused by
2491                                  * receiving quota and reservation out of order.
2492                                  * Try again in a second pass.
2493                                  */
2494                                 err = nvlist_add_nvpair(retrynvl, pair);
2495                         }
2496                 }

2498                 if (err != 0) {
2499                         if (errlist != NULL)
```

```
2500                                fnvlist_add_int32(errlist, propname, err);
2501                        rv = err;
2502                }
2503        }

2505        if (nvl != retrynvl && !nvlist_empty(retrynvl)) {
2506                nvl = retrynvl;
2507                goto retry;
2508        }

2510        if (!nvlist_empty(genericnvl) &&
2511            dsl_props_set(dsname, source, genericnvl) != 0) {
2512                /*
2513                 * If this fails, we still want to set as many properties as we
2514                 * can, so try setting them individually.
2515                 */
2516                pair = NULL;
2517                while ((pair = nvlist_next_nvpair(genericnvl, pair)) != NULL) {
2518                        const char *propname = nvpair_name(pair);
2519                        int err = 0;

2521                        propval = pair;
2522                        if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2523                                nvlist_t *attrs;
2524                                attrs = fnvpair_value_nvlist(pair);
2525                                propval = fnvlist_lookup_nvpair(attrs,
2526                                    ZPROP_VALUE);
2527                        }

2529                        if (nvpair_type(propval) == DATA_TYPE_STRING) {
2530                                strval = fnvpair_value_string(propval);
2531                                err = dsl_prop_set(dsname, propname, source, 1,
2532                                    strlen(strval) + 1, strval);
2533                        } else {
2534                                intval = fnvpair_value_uint64(propval);
2535                                err = dsl_prop_set(dsname, propname, source, 8,
2536                                    1, &intval);
2537                        }

2539                        if (err != 0) {
2540                                if (errlist != NULL) {
2541                                        fnvlist_add_int32(errlist, propname,
2542                                            err);
2543                                }
2544                                rv = err;
2545                        }
2546                }
2547        }
2548        nvlist_free(genericnvl);
2549        nvlist_free(retrynvl);

2551        return (rv);
2552 }

2554 /*
2555  * Check that all the properties are valid user properties.
2556  */
2557 static int
2558 zfs_check_userprops(const char *fsname, nvlist_t *nvl)
2559 {
2560        nvpair_t *pair = NULL;
2561        int error = 0;

2563        while ((pair = nvlist_next_nvpair(nvl, pair)) != NULL) {
2564                const char *propname = nvpair_name(pair);
2565                char *valstr;
```



```
2567                if (!zfs_prop_user(propname) ||
2568                    nvpair_type(pair) != DATA_TYPE_STRING)
2569                        return (EINVAL);

2571                if (error = zfs_secpolicy_write_perms(fsname,
2572                    ZFS_DELEG_PERM_USERPROP, CRED()))
2573                        return (error);

2575                if (strlen(propname) >= ZAP_MAXNAMELEN)
2576                        return (ENAMETOOLONG);

2578                VERIFY(nvpair_value_string(pair, &valstr) == 0);
2579                if (strlen(valstr) >= ZAP_MAXVALUELEN)
2580                        return (E2BIG);
2581        }
2582        return (0);
2583 }

2585 static void
2586 props_skip(nvlist_t *props, nvlist_t *skipped, nvlist_t **newprops)
2587 {
2588        nvpair_t *pair;

2590        VERIFY(nvlist_alloc(newprops, NV_UNIQUE_NAME, KM_SLEEP) == 0);

2592        pair = NULL;
2593        while ((pair = nvlist_next_nvpair(props, pair)) != NULL) {
2594                if (nvlist_exists(skipped, nvpair_name(pair)))
2595                        continue;

2597                VERIFY(nvlist_add_nvpair(*newprops, pair) == 0);
2598        }
2599 }

2601 static int
2602 clear_received_props(objset_t *os, const char *fs, nvlist_t *props,
2603     nvlist_t *skipped)
2604 {
2605        int err = 0;
2606        nvlist_t *cleared_props = NULL;
2607        props_skip(props, skipped, &cleared_props);
2608        if (!nvlist_empty(cleared_props)) {
2609                /*
2610                 * Acts on local properties until the dataset has received
2611                 * properties at least once on or after SPA_VERSION_RECVD_PROPS.
2612                 */
2613                zprop_source_t flags = (ZPROP_SRC_NONE |
2614                    (dsl_prop_get_hasrecvd(os) ? ZPROP_SRC_RECEIVED : 0));
2615                err = zfs_set_prop_nvlist(fs, flags, cleared_props, NULL);
2616        }
2617        nvlist_free(cleared_props);
2618        return (err);
2619 }

2621 /*
2622  * inputs:
2623  * zc_name              name of filesystem
2624  * zc_value             name of property to set
2625  * zc_nvlist_src{_size} nvlist of properties to apply
2626  * zc_cookie            received properties flag
2627  *
2628  * outputs:
2629  * zc_nvlist_dst{_size} error for each unapplied received property
2630  */
2631 static int
```

```
2632 zfs_ioc_set_prop(zfs_cmd_t *zc)
2633 {
2634         nvlist_t *nvl;
2635         boolean_t received = zc->zc_cookie;
2636         zprop_source_t source = (received ? ZPROP_SRC_RECEIVED :
2637             ZPROP_SRC_LOCAL);
2638         nvlist_t *errors;
2639         int error;

2641         if ((error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
2642             zc->zc_iflags, &nvl)) != 0)
2643                 return (error);

2645         if (received) {
2646                 nvlist_t *origprops;
2647                 objset_t *os;

2649                 if (dmu_objset_hold(zc->zc_name, FTAG, &os) == 0) {
2650                         if (dsl_prop_get_received(os, &origprops) == 0) {
2651                                 (void) clear_received_props(os,
2652                                     zc->zc_name, origprops, nvl);
2653                                 nvlist_free(origprops);
2654                         }

2656                         dsl_prop_set_hasrecvd(os);
2657                         dmu_objset_rele(os, FTAG);
2658                 }
2659         }

2661         errors = fnvlist_alloc();
2662         error = zfs_set_prop_nvlist(zc->zc_name, source, nvl, errors);

2664         if (zc->zc_nvlist_dst != NULL && errors != NULL) {
2665                 (void) put_nvlist(zc, errors);
2666         }

2668         nvlist_free(errors);
2669         nvlist_free(nvl);
2670         return (error);
2671 }

2673 /*
2674  * inputs:
2675  * zc_name              name of filesystem
2676  * zc_value             name of property to inherit
2677  * zc_cookie            revert to received value if TRUE
2678  *
2679  * outputs:             none
2680  */
2681 static int
2682 zfs_ioc_inherit_prop(zfs_cmd_t *zc)
2683 {
2684         const char *propname = zc->zc_value;
2685         zfs_prop_t prop = zfs_name_to_prop(propname);
2686         boolean_t received = zc->zc_cookie;
2687         zprop_source_t source = (received
2688             ? ZPROP_SRC_NONE           /* revert to received value, if any */
2689             : ZPROP_SRC_INHERITED);    /* explicitly inherit */

2691         if (received) {
2692                 nvlist_t *dummy;
2693                 nvpair_t *pair;
2694                 zprop_type_t type;
2695                 int err;

2697                 /*
```



```
2698                  * zfs_prop_set_special() expects properties in the form of an
2699                  * nvpair with type info.
2700                  */
2701                 if (prop == ZPROP_INVAL) {
2702                         if (!zfs_prop_user(propname))
2703                                 return (EINVAL);

2705                         type = PROP_TYPE_STRING;
2706                 } else if (prop == ZFS_PROP_VOLSIZE ||
2707                     prop == ZFS_PROP_VERSION) {
2708                         return (EINVAL);
2709                 } else {
2710                         type = zfs_prop_get_type(prop);
2711                 }

2713                 VERIFY(nvlist_alloc(&dummy, NV_UNIQUE_NAME, KM_SLEEP) == 0);

2715                 switch (type) {
2716                 case PROP_TYPE_STRING:
2717                         VERIFY(0 == nvlist_add_string(dummy, propname, ""));
2718                         break;
2719                 case PROP_TYPE_NUMBER:
2720                 case PROP_TYPE_INDEX:
2721                         VERIFY(0 == nvlist_add_uint64(dummy, propname, 0));
2722                         break;
2723                 default:
2724                         nvlist_free(dummy);
2725                         return (EINVAL);
2726                 }

2728                 pair = nvlist_next_nvpair(dummy, NULL);
2729                 err = zfs_prop_set_special(zc->zc_name, source, pair);
2730                 nvlist_free(dummy);
2731                 if (err != -1)
2732                         return (err); /* special property already handled */
2733         } else {
2734                 /*
2735                  * Only check this in the non-received case. We want to allow
2736                  * 'inherit -S' to revert non-inheritable properties like quota
2737                  * and reservation to the received or default values even though
2738                  * they are not considered inheritable.
2739                  */
2740                 if (prop != ZPROP_INVAL && !zfs_prop_inheritable(prop))
2741                         return (EINVAL);
2742         }

2744         /* property name has been validated by zfs_secpolicy_inherit_prop() */
2745         return (dsl_prop_set(zc->zc_name, zc->zc_value, source, 0, 0, NULL));
2746 }

2748 static int
2749 zfs_ioc_pool_set_props(zfs_cmd_t *zc)
2750 {
2751         nvlist_t *props;
2752         spa_t *spa;
2753         int error;
2754         nvpair_t *pair;

2756         if (error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
2757             zc->zc_iflags, &props))
2758                 return (error);

2760         /*
2761          * If the only property is the configfile, then just do a spa_lookup()
2762          * to handle the faulted case.
2763          */
```

```
2764         pair = nvlist_next_nvpair(props, NULL);
2765         if (pair != NULL && strcmp(nvpair_name(pair),
2766             zpool_prop_to_name(ZPOOL_PROP_CACHEFILE)) == 0 &&
2767             nvlist_next_nvpair(props, pair) == NULL) {
2768                 mutex_enter(&spa_namespace_lock);
2769                 if ((spa = spa_lookup(zc->zc_name)) != NULL) {
2770                         spa_configfile_set(spa, props, B_FALSE);
2771                         spa_config_sync(spa, B_FALSE, B_TRUE);
2772                 }
2773                 mutex_exit(&spa_namespace_lock);
2774                 if (spa != NULL) {
2775                         nvlist_free(props);
2776                         return (0);
2777                 }
2778         }

2780         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0) {
2781                 nvlist_free(props);
2782                 return (error);
2783         }

2785         error = spa_prop_set(spa, props);

2787         nvlist_free(props);
2788         spa_close(spa, FTAG);

2790         return (error);
2791 }

2793 static int
2794 zfs_ioc_pool_get_props(zfs_cmd_t *zc)
2795 {
2796         spa_t *spa;
2797         int error;
2798         nvlist_t *nvp = NULL;

2800         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0) {
2801                 /*
2802                  * If the pool is faulted, there may be properties we can still
2803                  * get (such as altroot and cachefile), so attempt to get them
2804                  * anyway.
2805                  */
2806                 mutex_enter(&spa_namespace_lock);
2807                 if ((spa = spa_lookup(zc->zc_name)) != NULL)
2808                         error = spa_prop_get(spa, &nvp);
2809                 mutex_exit(&spa_namespace_lock);
2810         } else {
2811                 error = spa_prop_get(spa, &nvp);
2812                 spa_close(spa, FTAG);
2813         }

2815         if (error == 0 && zc->zc_nvlist_dst != NULL)
2816                 error = put_nvlist(zc, nvp);
2817         else
2818                 error = EFAULT;

2820         nvlist_free(nvp);
2821         return (error);
2822 }

2824 /*
2825  * inputs:
2826  * zc_name              name of filesystem
2827  * zc_nvlist_src{_size} nvlist of delegated permissions
2828  * zc_perm_action       allow/unallow flag
2829  *
```



```
2830  * outputs:             none
2831  */
2832 static int
2833 zfs_ioc_set_fsacl(zfs_cmd_t *zc)
2834 {
2835         int error;
2836         nvlist_t *fsaclnv = NULL;

2838         if ((error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
2839             zc->zc_iflags, &fsaclnv)) != 0)
2840                 return (error);

2842         /*
2843          * Verify nvlist is constructed correctly
2844          */
2845         if ((error = zfs_deleg_verify_nvlist(fsaclnv)) != 0) {
2846                 nvlist_free(fsaclnv);
2847                 return (EINVAL);
2848         }

2850         /*
2851          * If we don't have PRIV_SYS_MOUNT, then validate
2852          * that user is allowed to hand out each permission in
2853          * the nvlist(s)
2854          */

2856         error = secpolicy_zfs(CRED());
2857         if (error) {
2858                 if (zc->zc_perm_action == B_FALSE) {
2859                         error = dsl_deleg_can_allow(zc->zc_name,
2860                             fsaclnv, CRED());
2861                 } else {
2862                         error = dsl_deleg_can_unallow(zc->zc_name,
2863                             fsaclnv, CRED());
2864                 }
2865         }

2867         if (error == 0)
2868                 error = dsl_deleg_set(zc->zc_name, fsaclnv, zc->zc_perm_action);

2870         nvlist_free(fsaclnv);
2871         return (error);
2872 }

2874 /*
2875  * inputs:
2876  * zc_name              name of filesystem
2877  *
2878  * outputs:
2879  * zc_nvlist_src{_size} nvlist of delegated permissions
2880  */
2881 static int
2882 zfs_ioc_get_fsacl(zfs_cmd_t *zc)
2883 {
2884         nvlist_t *nvp;
2885         int error;

2887         if ((error = dsl_deleg_get(zc->zc_name, &nvp)) == 0) {
2888                 error = put_nvlist(zc, nvp);
2889                 nvlist_free(nvp);
2890         }

2892         return (error);
2893 }

2895 /*
```

```
 * Search the vfs list for a specified resource.  Returns a pointer to it
 * or NULL if no suitable entry is found. The caller of this routine
 * is responsible for releasing the returned vfs pointer.
 */
static vfs_t *
zfs_get_vfs(const char *resource)
{
        struct vfs *vfsp;
        struct vfs *vfs_found = NULL;

        vfs_list_read_lock();
        vfsp = rootvfs;
        do {
                if (strcmp(refstr_value(vfsp->vfs_resource), resource) == 0) {
                        VFS_HOLD(vfsp);
                        vfs_found = vfsp;
                        break;
                }
                vfsp = vfsp->vfs_next;
        } while (vfsp != rootvfs);
        vfs_list_unlock();
        return (vfs_found);
}

/* ARGSUSED */
static void
zfs_create_cb(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx)
{
        zfs_creat_t *zct = arg;

        zfs_create_fs(os, cr, zct->zct_zplprops, tx);
}

#define ZFS_PROP_UNDEFINED      ((uint64_t)-1)

/*
 * inputs:
 * createprops          list of properties requested by creator
 * default_zplver       zpl version to use if unspecified in createprops
 * fuids_ok             fuids allowed in this version of the spa?
 * os                   parent objset pointer (NULL if root fs)
 *
 * outputs:
 * zplprops     values for the zplprops we attach to the master node object
 * is_ci        true if requested file system will be purely case-insensitive
 *
 * Determine the settings for utf8only, normalization and
 * casesensitivity.  Specific values may have been requested by the
 * creator and/or we can inherit values from the parent dataset.  If
 * the file system is of too early a vintage, a creator can not
 * request settings for these properties, even if the requested
 * setting is the default value.  We don't actually want to create dsl
 * properties for these, so remove them from the source nvlist after
 * processing.
 */
static int
zfs_fill_zplprops_impl(objset_t *os, uint64_t zplver,
    boolean_t fuids_ok, boolean_t sa_ok, nvlist_t *createprops,
    nvlist_t *zplprops, boolean_t *is_ci)
{
        uint64_t sense = ZFS_PROP_UNDEFINED;
        uint64_t norm = ZFS_PROP_UNDEFINED;
        uint64_t u8 = ZFS_PROP_UNDEFINED;

        ASSERT(zplprops != NULL);
```



```
        /*
         * Pull out creator prop choices, if any.
         */
        if (createprops) {
                (void) nvlist_lookup_uint64(createprops,
                    zfs_prop_to_name(ZFS_PROP_VERSION), &zplver);
                (void) nvlist_lookup_uint64(createprops,
                    zfs_prop_to_name(ZFS_PROP_NORMALIZE), &norm);
                (void) nvlist_remove_all(createprops,
                    zfs_prop_to_name(ZFS_PROP_NORMALIZE));
                (void) nvlist_lookup_uint64(createprops,
                    zfs_prop_to_name(ZFS_PROP_UTF8ONLY), &u8);
                (void) nvlist_remove_all(createprops,
                    zfs_prop_to_name(ZFS_PROP_UTF8ONLY));
                (void) nvlist_lookup_uint64(createprops,
                    zfs_prop_to_name(ZFS_PROP_CASE), &sense);
                (void) nvlist_remove_all(createprops,
                    zfs_prop_to_name(ZFS_PROP_CASE));
        }

        /*
         * If the zpl version requested is whacky or the file system
         * or pool is version is too "young" to support normalization
         * and the creator tried to set a value for one of the props,
         * error out.
         */
        if ((zplver < ZPL_VERSION_INITIAL || zplver > ZPL_VERSION) ||
            (zplver >= ZPL_VERSION_FUID && !fuids_ok) ||
            (zplver >= ZPL_VERSION_SA && !sa_ok) ||
            (zplver < ZPL_VERSION_NORMALIZATION &&
            (norm != ZFS_PROP_UNDEFINED || u8 != ZFS_PROP_UNDEFINED ||
            sense != ZFS_PROP_UNDEFINED)))
                return (ENOTSUP);

        /*
         * Put the version in the zplprops
         */
        VERIFY(nvlist_add_uint64(zplprops,
            zfs_prop_to_name(ZFS_PROP_VERSION), zplver) == 0);

        if (norm == ZFS_PROP_UNDEFINED)
                VERIFY(zfs_get_zplprop(os, ZFS_PROP_NORMALIZE, &norm) == 0);
        VERIFY(nvlist_add_uint64(zplprops,
            zfs_prop_to_name(ZFS_PROP_NORMALIZE), norm) == 0);

        /*
         * If we're normalizing, names must always be valid UTF-8 strings.
         */
        if (norm)
                u8 = 1;
        if (u8 == ZFS_PROP_UNDEFINED)
                VERIFY(zfs_get_zplprop(os, ZFS_PROP_UTF8ONLY, &u8) == 0);
        VERIFY(nvlist_add_uint64(zplprops,
            zfs_prop_to_name(ZFS_PROP_UTF8ONLY), u8) == 0);

        if (sense == ZFS_PROP_UNDEFINED)
                VERIFY(zfs_get_zplprop(os, ZFS_PROP_CASE, &sense) == 0);
        VERIFY(nvlist_add_uint64(zplprops,
            zfs_prop_to_name(ZFS_PROP_CASE), sense) == 0);

        if (is_ci)
                *is_ci = (sense == ZFS_CASE_INSENSITIVE);

        return (0);
}
```

```
3028 static int
3029 zfs_fill_zplprops(const char *dataset, nvlist_t *createprops,
3030     nvlist_t *zplprops, boolean_t *is_ci)
3031 {
3032         boolean_t fuids_ok, sa_ok;
3033         uint64_t zplver = ZPL_VERSION;
3034         objset_t *os = NULL;
3035         char parentname[MAXNAMELEN];
3036         char *cp;
3037         spa_t *spa;
3038         uint64_t spa_vers;
3039         int error;

3041         (void) strlcpy(parentname, dataset, sizeof (parentname));
3042         cp = strrchr(parentname, '/');
3043         ASSERT(cp != NULL);
3044         cp[0] = '\0';

3046         if ((error = spa_open(dataset, &spa, FTAG)) != 0)
3047                 return (error);

3049         spa_vers = spa_version(spa);
3050         spa_close(spa, FTAG);

3052         zplver = zfs_zpl_version_map(spa_vers);
3053         fuids_ok = (zplver >= ZPL_VERSION_FUID);
3054         sa_ok = (zplver >= ZPL_VERSION_SA);

3056         /*
3057          * Open parent object set so we can inherit zplprop values.
3058          */
3059         if ((error = dmu_objset_hold(parentname, FTAG, &os)) != 0)
3060                 return (error);

3062         error = zfs_fill_zplprops_impl(os, zplver, fuids_ok, sa_ok, createprops,
3063             zplprops, is_ci);
3064         dmu_objset_rele(os, FTAG);
3065         return (error);
3066 }

3068 static int
3069 zfs_fill_zplprops_root(uint64_t spa_vers, nvlist_t *createprops,
3070     nvlist_t *zplprops, boolean_t *is_ci)
3071 {
3072         boolean_t fuids_ok;
3073         boolean_t sa_ok;
3074         uint64_t zplver = ZPL_VERSION;
3075         int error;

3077         zplver = zfs_zpl_version_map(spa_vers);
3078         fuids_ok = (zplver >= ZPL_VERSION_FUID);
3079         sa_ok = (zplver >= ZPL_VERSION_SA);

3081         error = zfs_fill_zplprops_impl(NULL, zplver, fuids_ok, sa_ok,
3082             createprops, zplprops, is_ci);
3083         return (error);
3084 }

3086 /*
3087  * innvl: {
3088  *     "type" -> dmu_objset_type_t (int32)
3089  *     (optional) "props" -> { prop -> value }
3090  * }
3091  *
3092  * outnvl: propname -> error code (int32)
3093  */
```



```
3094 static int
3095 zfs_ioc_create(const char *fsname, nvlist_t *innvl, nvlist_t *outnvl)
3096 {
3097         int error = 0;
3098         zfs_creat_t zct = { 0 };
3099         nvlist_t *nvprops = NULL;
3100         void (*cbfunc)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx);
3101         int32_t type32;
3102         dmu_objset_type_t type;
3103         boolean_t is_insensitive = B_FALSE;

3105         if (nvlist_lookup_int32(innvl, "type", &type32) != 0)
3106                 return (EINVAL);
3107         type = type32;
3108         (void) nvlist_lookup_nvlist(innvl, "props", &nvprops);

3110         switch (type) {
3111         case DMU_OST_ZFS:
3112                 cbfunc = zfs_create_cb;
3113                 break;

3115         case DMU_OST_ZVOL:
3116                 cbfunc = zvol_create_cb;
3117                 break;

3119         default:
3120                 cbfunc = NULL;
3121                 break;
3122         }
3123         if (strchr(fsname, '@') ||
3124             strchr(fsname, '%'))
3125                 return (EINVAL);

3127         zct.zct_props = nvprops;

3129         if (cbfunc == NULL)
3130                 return (EINVAL);

3132         if (type == DMU_OST_ZVOL) {
3133                 uint64_t volsize, volblocksize;

3135                 if (nvprops == NULL)
3136                         return (EINVAL);
3137                 if (nvlist_lookup_uint64(nvprops,
3138                     zfs_prop_to_name(ZFS_PROP_VOLSIZE), &volsize) != 0)
3139                         return (EINVAL);

3141                 if ((error = nvlist_lookup_uint64(nvprops,
3142                     zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
3143                     &volblocksize)) != 0 && error != ENOENT)
3144                         return (EINVAL);

3146                 if (error != 0)
3147                         volblocksize = zfs_prop_default_numeric(
3148                             ZFS_PROP_VOLBLOCKSIZE);

3150                 if ((error = zvol_check_volblocksize(
3151                     volblocksize)) != 0 ||
3152                     (error = zvol_check_volsize(volsize,
3153                     volblocksize)) != 0)
3154                         return (error);
3155         } else if (type == DMU_OST_ZFS) {
3156                 int error;

3158                 /*
3159                  * We have to have normalization and
```

```
3160                 * case-folding flags correct when we do the
3161                 * file system creation, so go figure them out
3162                 * now.
3163                 */
3164                VERIFY(nvlist_alloc(&zct.zct_zplprops,
3165                    NV_UNIQUE_NAME, KM_SLEEP) == 0);
3166                error = zfs_fill_zplprops(fsname, nvprops,
3167                    zct.zct_zplprops, &is_insensitive);
3168                if (error != 0) {
3169                        nvlist_free(zct.zct_zplprops);
3170                        return (error);
3171                }
3172        }

3174        error = dmu_objset_create(fsname, type,
3175            is_insensitive ? DS_FLAG_CI_DATASET : 0, cbfunc, &zct);
3176        nvlist_free(zct.zct_zplprops);

3178        /*
3179         * It would be nice to do this atomically.
3180         */
3181        if (error == 0) {
3182                error = zfs_set_prop_nvlist(fsname, ZPROP_SRC_LOCAL,
3183                    nvprops, outnvl);
3184                if (error != 0)
3185                        (void) dmu_objset_destroy(fsname, B_FALSE);
3186        }
3187        return (error);
3188 }

3190 /*
3191  * innvl: {
3192  *     "origin" -> name of origin snapshot
3193  *     (optional) "props" -> { prop -> value }
3194  * }
3195  *
3196  * outnvl: propname -> error code (int32)
3197  */
3198 static int
3199 zfs_ioc_clone(const char *fsname, nvlist_t *innvl, nvlist_t *outnvl)
3200 {
3201        int error = 0;
3202        nvlist_t *nvprops = NULL;
3203        char *origin_name;
3204        dsl_dataset_t *origin;

3206        if (nvlist_lookup_string(innvl, "origin", &origin_name) != 0)
3207                return (EINVAL);
3208        (void) nvlist_lookup_nvlist(innvl, "props", &nvprops);

3210        if (strchr(fsname, '@') ||
3211            strchr(fsname, '%'))
3212                return (EINVAL);

3214        if (dataset_namecheck(origin_name, NULL, NULL) != 0)
3215                return (EINVAL);

3217        error = dsl_dataset_hold(origin_name, FTAG, &origin);
3218        if (error)
3219                return (error);

3221        error = dmu_objset_clone(fsname, origin, 0);
3222        dsl_dataset_rele(origin, FTAG);
3223        if (error)
3224                return (error);
```



```
3226        /*
3227         * It would be nice to do this atomically.
3228         */
3229        if (error == 0) {
3230                error = zfs_set_prop_nvlist(fsname, ZPROP_SRC_LOCAL,
3231                    nvprops, outnvl);
3232                if (error != 0)
3233                        (void) dmu_objset_destroy(fsname, B_FALSE);
3234        }
3235        return (error);
3236 }

3238 /*
3239  * innvl: {
3240  *     "snaps" -> { snapshot1, snapshot2 }
3241  *     (optional) "props" -> { prop -> value (string) }
3242  * }
3243  *
3244  * outnvl: snapshot -> error code (int32)
3245  *
3246  */
3247 static int
3248 zfs_ioc_snapshot(const char *poolname, nvlist_t *innvl, nvlist_t *outnvl)
3249 {
3250        nvlist_t *snaps;
3251        nvlist_t *props = NULL;
3252        int error, poollen;
3253        nvpair_t *pair;

3255        (void) nvlist_lookup_nvlist(innvl, "props", &props);
3256        if ((error = zfs_check_userprops(poolname, props)) != 0)
3257                return (error);

3259        if (!nvlist_empty(props) &&
3260            zfs_earlier_version(poolname, SPA_VERSION_SNAP_PROPS))
3261                return (ENOTSUP);

3263        if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
3264                return (EINVAL);
3265        poollen = strlen(poolname);
3266        for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
3267            pair = nvlist_next_nvpair(snaps, pair)) {
3268                const char *name = nvpair_name(pair);
3269                const char *cp = strchr(name, '@');

3271                /*
3272                 * The snap name must contain an @, and the part after it must
3273                 * contain only valid characters.
3274                 */
3275                if (cp == NULL || snapshot_namecheck(cp + 1, NULL, NULL) != 0)
3276                        return (EINVAL);

3278                /*
3279                 * The snap must be in the specified pool.
3280                 */
3281                if (strncmp(name, poolname, poollen) != 0 ||
3282                    (name[poollen] != '/' && name[poollen] != '@'))
3283                        return (EXDEV);

3285                /* This must be the only snap of this fs. */
3286                for (nvpair_t *pair2 = nvlist_next_nvpair(snaps, pair);
3287                    pair2 != NULL; pair2 = nvlist_next_nvpair(snaps, pair2)) {
3288                        if (strncmp(name, nvpair_name(pair2), cp - name + 1)
3289                            == 0) {
3290                                return (EXDEV);
3291                        }
```

```
3292                 }
3293         }

3295         error = dmu_objset_snapshot(snaps, props, outnvl);
3296         return (error);
3297 }

3299 /*
3300  * innvl: "message" -> string
3301  */
3302 /* ARGSUSED */
3303 static int
3304 zfs_ioc_log_history(const char *unused, nvlist_t *innvl, nvlist_t *outnvl)
3305 {
3306         char *message;
3307         spa_t *spa;
3308         int error;
3309         char *poolname;

3311         /*
3312          * The poolname in the ioctl is not set, we get it from the TSD,
3313          * which was set at the end of the last successful ioctl that allows
3314          * logging.  The secpolicy func already checked that it is set.
3315          * Only one log ioctl is allowed after each successful ioctl, so
3316          * we clear the TSD here.
3317          */
3318         poolname = tsd_get(zfs_allow_log_key);
3319         (void) tsd_set(zfs_allow_log_key, NULL);
3320         error = spa_open(poolname, &spa, FTAG);
3321         strfree(poolname);
3322         if (error != 0)
3323                 return (error);

3325         if (nvlist_lookup_string(innvl, "message", &message) != 0)  {
3326                 spa_close(spa, FTAG);
3327                 return (EINVAL);
3328         }

3330         if (spa_version(spa) < SPA_VERSION_ZPOOL_HISTORY) {
3331                 spa_close(spa, FTAG);
3332                 return (ENOTSUP);
3333         }

3335         error = spa_history_log(spa, message);
3336         spa_close(spa, FTAG);
3337         return (error);
3338 }

3340 /* ARGSUSED */
3341 int
3342 zfs_unmount_snap(const char *name, void *arg)
3343 {
3344         vfs_t *vfsp;
3345         int err;

3347         if (strchr(name, '@') == NULL)
3348                 return (0);

3350         vfsp = zfs_get_vfs(name);
3351         if (vfsp == NULL)
3352                 return (0);

3354         if ((err = vn_vfswlock(vfsp->vfs_vnodecovered)) != 0) {
3355                 VFS_RELE(vfsp);
3356                 return (err);
3357         }
```



```
3358         VFS_RELE(vfsp);

3360         /*
3361          * Always force the unmount for snapshots.
3362          */
3363         return (dounmount(vfsp, MS_FORCE, kcred));
3364 }

3366 /*
3367  * innvl: {
3368  *     "snaps" -> { snapshot1, snapshot2 }
3369  *     (optional boolean) "defer"
3370  * }
3371  *
3372  * outnvl: snapshot -> error code (int32)
3373  *
3374  */
3375 static int
3376 zfs_ioc_destroy_snaps(const char *poolname, nvlist_t *innvl, nvlist_t *outnvl)
3377 {
3378         int poollen;
3379         nvlist_t *snaps;
3380         nvpair_t *pair;
3381         boolean_t defer;

3383         if (nvlist_lookup_nvlist(innvl, "snaps", &snaps) != 0)
3384                 return (EINVAL);
3385         defer = nvlist_exists(innvl, "defer");

3387         poollen = strlen(poolname);
3388         for (pair = nvlist_next_nvpair(snaps, NULL); pair != NULL;
3389             pair = nvlist_next_nvpair(snaps, pair)) {
3390                 const char *name = nvpair_name(pair);

3392                 /*
3393                  * The snap must be in the specified pool.
3394                  */
3395                 if (strncmp(name, poolname, poollen) != 0 ||
3396                     (name[poollen] != '/' && name[poollen] != '@'))
3397                         return (EXDEV);

3399                 /*
3400                  * Ignore failures to unmount; dmu_snapshots_destroy_nvl()
3401                  * will deal with this gracefully (by filling in outnvl).
3402                  */
3403                 (void) zfs_unmount_snap(name, NULL);
3404         }

3406         return (dmu_snapshots_destroy_nvl(snaps, defer, outnvl));
3407 }

3409 /*
3410  * inputs:
3411  * zc_name              name of dataset to destroy
3412  * zc_objset_type       type of objset
3413  * zc_defer_destroy     mark for deferred destroy
3414  *
3415  * outputs:             none
3416  */
3417 static int
3418 zfs_ioc_destroy(zfs_cmd_t *zc)
3419 {
3420         int err;
3421         if (strchr(zc->zc_name, '@') && zc->zc_objset_type == DMU_OST_ZFS) {
3422                 err = zfs_unmount_snap(zc->zc_name, NULL);
3423                 if (err)
```

```
3424                        return (err);
3425        }

3427        err = dmu_objset_destroy(zc->zc_name, zc->zc_defer_destroy);
3428        if (zc->zc_objset_type == DMU_OST_ZVOL && err == 0)
3429                (void) zvol_remove_minor(zc->zc_name);
3430        return (err);
3431 }

3433 /*
3434  * inputs:
3435  * zc_name      name of dataset to rollback (to most recent snapshot)
3436  *
3437  * outputs:     none
3438  */
3439 static int
3440 zfs_ioc_rollback(zfs_cmd_t *zc)
3441 {
3442        dsl_dataset_t *ds, *clone;
3443        int error;
3444        zfsvfs_t *zfsvfs;
3445        char *clone_name;

3447        error = dsl_dataset_hold(zc->zc_name, FTAG, &ds);
3448        if (error)
3449                return (error);

3451        /* must not be a snapshot */
3452        if (dsl_dataset_is_snapshot(ds)) {
3453                dsl_dataset_rele(ds, FTAG);
3454                return (EINVAL);
3455        }

3457        /* must have a most recent snapshot */
3458        if (ds->ds_phys->ds_prev_snap_txg < TXG_INITIAL) {
3459                dsl_dataset_rele(ds, FTAG);
3460                return (EINVAL);
3461        }

3463        /*
3464         * Create clone of most recent snapshot.
3465         */
3466        clone_name = kmem_asprintf("%s/%%rollback", zc->zc_name);
3467        error = dmu_objset_clone(clone_name, ds->ds_prev, DS_FLAG_INCONSISTENT);
3468        if (error)
3469                goto out;

3471        error = dsl_dataset_own(clone_name, B_TRUE, FTAG, &clone);
3472        if (error)
3473                goto out;

3475        /*
3476         * Do clone swap.
3477         */
3478        if (getzfsvfs(zc->zc_name, &zfsvfs) == 0) {
3479                error = zfs_suspend_fs(zfsvfs);
3480                if (error == 0) {
3481                        int resume_err;

3483                        if (dsl_dataset_tryown(ds, B_FALSE, FTAG)) {
3484                                error = dsl_dataset_clone_swap(clone, ds,
3485                                    B_TRUE);
3486                                dsl_dataset_disown(ds, FTAG);
3487                                ds = NULL;
3488                        } else {
3489                                error = EBUSY;
```



```
3490                        }
3491                        resume_err = zfs_resume_fs(zfsvfs, zc->zc_name);
3492                        error = error ? error : resume_err;
3493                }
3494                VFS_RELE(zfsvfs->z_vfs);
3495        } else {
3496                if (dsl_dataset_tryown(ds, B_FALSE, FTAG)) {
3497                        error = dsl_dataset_clone_swap(clone, ds, B_TRUE);
3498                        dsl_dataset_disown(ds, FTAG);
3499                        ds = NULL;
3500                } else {
3501                        error = EBUSY;
3502                }
3503        }

3505        /*
3506         * Destroy clone (which also closes it).
3507         */
3508        (void) dsl_dataset_destroy(clone, FTAG, B_FALSE);

3510 out:
3511        strfree(clone_name);
3512        if (ds)
3513                dsl_dataset_rele(ds, FTAG);
3514        return (error);
3515 }

3517 /*
3518  * inputs:
3519  * zc_name      old name of dataset
3520  * zc_value     new name of dataset
3521  * zc_cookie    recursive flag (only valid for snapshots)
3522  *
3523  * outputs:     none
3524  */
3525 static int
3526 zfs_ioc_rename(zfs_cmd_t *zc)
3527 {
3528        boolean_t recursive = zc->zc_cookie & 1;

3530        zc->zc_value[sizeof (zc->zc_value) - 1] = '\0';
3531        if (dataset_namecheck(zc->zc_value, NULL, NULL) != 0 ||
3532            strchr(zc->zc_value, '%'))
3533                return (EINVAL);

3535        /*
3536         * Unmount snapshot unless we're doing a recursive rename,
3537         * in which case the dataset code figures out which snapshots
3538         * to unmount.
3539         */
3540        if (!recursive && strchr(zc->zc_name, '@') != NULL &&
3541            zc->zc_objset_type == DMU_OST_ZFS) {
3542                int err = zfs_unmount_snap(zc->zc_name, NULL);
3543                if (err)
3544                        return (err);
3545        }
3546        if (zc->zc_objset_type == DMU_OST_ZVOL)
3547                (void) zvol_remove_minor(zc->zc_name);
3548        return (dmu_objset_rename(zc->zc_name, zc->zc_value, recursive));
3549 }

3551 static int
3552 zfs_check_settable(const char *dsname, nvpair_t *pair, cred_t *cr)
3553 {
3554        const char *propname = nvpair_name(pair);
3555        boolean_t issnap = (strchr(dsname, '@') != NULL);
```

```
3556         zfs_prop_t prop = zfs_name_to_prop(propname);
3557         uint64_t intval;
3558         int err;

3560         if (prop == ZPROP_INVAL) {
3561                 if (zfs_prop_user(propname)) {
3562                         if (err = zfs_secpolicy_write_perms(dsname,
3563                             ZFS_DELEG_PERM_USERPROP, cr))
3564                                 return (err);
3565                         return (0);
3566                 }

3568                 if (!issnap && zfs_prop_userquota(propname)) {
3569                         const char *perm = NULL;
3570                         const char *uq_prefix =
3571                             zfs_userquota_prop_prefixes[ZFS_PROP_USERQUOTA];
3572                         const char *gq_prefix =
3573                             zfs_userquota_prop_prefixes[ZFS_PROP_GROUPQUOTA];

3575                         if (strncmp(propname, uq_prefix,
3576                             strlen(uq_prefix)) == 0) {
3577                                 perm = ZFS_DELEG_PERM_USERQUOTA;
3578                         } else if (strncmp(propname, gq_prefix,
3579                             strlen(gq_prefix)) == 0) {
3580                                 perm = ZFS_DELEG_PERM_GROUPQUOTA;
3581                         } else {
3582                                 /* USERUSED and GROUPUSED are read-only */
3583                                 return (EINVAL);
3584                         }

3586                         if (err = zfs_secpolicy_write_perms(dsname, perm, cr))
3587                                 return (err);
3588                         return (0);
3589                 }

3591                 return (EINVAL);
3592         }

3594         if (issnap)
3595                 return (EINVAL);

3597         if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
3598                 /*
3599                  * dsl_prop_get_all_impl() returns properties in this
3600                  * format.
3601                  */
3602                 nvlist_t *attrs;
3603                 VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
3604                 VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
3605                     &pair) == 0);
3606         }

3608         /*
3609          * Check that this value is valid for this pool version
3610          */
3611         switch (prop) {
3612         case ZFS_PROP_COMPRESSION:
3613                 /*
3614                  * If the user specified gzip compression, make sure
3615                  * the SPA supports it. We ignore any errors here since
3616                  * we'll catch them later.
3617                  */
3618                 if (nvpair_type(pair) == DATA_TYPE_UINT64 &&
3619                     nvpair_value_uint64(pair, &intval) == 0) {
3620                         if (intval >= ZIO_COMPRESS_GZIP_1 &&
3621                             intval <= ZIO_COMPRESS_GZIP_9 &&
```



```
3622                             zfs_earlier_version(dsname,
3623                             SPA_VERSION_GZIP_COMPRESSION)) {
3624                                 return (ENOTSUP);
3625                         }

3627                         if (intval == ZIO_COMPRESS_ZLE &&
3628                             zfs_earlier_version(dsname,
3629                             SPA_VERSION_ZLE_COMPRESSION))
3630                                 return (ENOTSUP);

3632                         /*
3633                          * If this is a bootable dataset then
3634                          * verify that the compression algorithm
3635                          * is supported for booting. We must return
3636                          * something other than ENOTSUP since it
3637                          * implies a downrev pool version.
3638                          */
3639                         if (zfs_is_bootfs(dsname) &&
3640                             !BOOTFS_COMPRESS_VALID(intval)) {
3641                                 return (ERANGE);
3642                         }
3643                 }
3644                 break;

3646         case ZFS_PROP_COPIES:
3647                 if (zfs_earlier_version(dsname, SPA_VERSION_DITTO_BLOCKS))
3648                         return (ENOTSUP);
3649                 break;

3651         case ZFS_PROP_DEDUP:
3652                 if (zfs_earlier_version(dsname, SPA_VERSION_DEDUP))
3653                         return (ENOTSUP);
3654                 break;

3656         case ZFS_PROP_SHARESMB:
3657                 if (zpl_earlier_version(dsname, ZPL_VERSION_FUID))
3658                         return (ENOTSUP);
3659                 break;

3661         case ZFS_PROP_ACLINHERIT:
3662                 if (nvpair_type(pair) == DATA_TYPE_UINT64 &&
3663                     nvpair_value_uint64(pair, &intval) == 0) {
3664                         if (intval == ZFS_ACL_PASSTHROUGH_X &&
3665                             zfs_earlier_version(dsname,
3666                             SPA_VERSION_PASSTHROUGH_X))
3667                                 return (ENOTSUP);
3668                 }
3669                 break;
3670         }

3672         return (zfs_secpolicy_setprop(dsname, prop, pair, CRED()));
3673 }

3675 /*
3676  * Removes properties from the given props list that fail permission checks
3677  * needed to clear them and to restore them in case of a receive error. For each
3678  * property, make sure we have both set and inherit permissions.
3679  *
3680  * Returns the first error encountered if any permission checks fail. If the
3681  * caller provides a non-NULL errlist, it also gives the complete list of names
3682  * of all the properties that failed a permission check along with the
3683  * corresponding error numbers. The caller is responsible for freeing the
3684  * returned errlist.
3685  *
3686  * If every property checks out successfully, zero is returned and the list
3687  * pointed at by errlist is NULL.
```

```
 */
static int
zfs_check_clearable(char *dataset, nvlist_t *props, nvlist_t **errlist)
{
        zfs_cmd_t *zc;
        nvpair_t *pair, *next_pair;
        nvlist_t *errors;
        int err, rv = 0;

        if (props == NULL)
                return (0);

        VERIFY(nvlist_alloc(&errors, NV_UNIQUE_NAME, KM_SLEEP) == 0);

        zc = kmem_alloc(sizeof (zfs_cmd_t), KM_SLEEP);
        (void) strcpy(zc->zc_name, dataset);
        pair = nvlist_next_nvpair(props, NULL);
        while (pair != NULL) {
                next_pair = nvlist_next_nvpair(props, pair);

                (void) strcpy(zc->zc_value, nvpair_name(pair));
                if ((err = zfs_check_settable(dataset, pair, CRED())) != 0 ||
                    (err = zfs_secpolicy_inherit_prop(zc, NULL, CRED())) != 0) {
                        VERIFY(nvlist_remove_nvpair(props, pair) == 0);
                        VERIFY(nvlist_add_int32(errors,
                            zc->zc_value, err) == 0);
                }
                pair = next_pair;
        }
        kmem_free(zc, sizeof (zfs_cmd_t));

        if ((pair = nvlist_next_nvpair(errors, NULL)) == NULL) {
                nvlist_free(errors);
                errors = NULL;
        } else {
                VERIFY(nvpair_value_int32(pair, &rv) == 0);
        }

        if (errlist == NULL)
                nvlist_free(errors);
        else
                *errlist = errors;

        return (rv);
}

static boolean_t
propval_equals(nvpair_t *p1, nvpair_t *p2)
{
        if (nvpair_type(p1) == DATA_TYPE_NVLIST) {
                /* dsl_prop_get_all_impl() format */
                nvlist_t *attrs;
                VERIFY(nvpair_value_nvlist(p1, &attrs) == 0);
                VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
                    &p1) == 0);
        }

        if (nvpair_type(p2) == DATA_TYPE_NVLIST) {
                nvlist_t *attrs;
                VERIFY(nvpair_value_nvlist(p2, &attrs) == 0);
                VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
                    &p2) == 0);
        }

        if (nvpair_type(p1) != nvpair_type(p2))
                return (B_FALSE);
```



```
        if (nvpair_type(p1) == DATA_TYPE_STRING) {
                char *valstr1, *valstr2;

                VERIFY(nvpair_value_string(p1, (char **)&valstr1) == 0);
                VERIFY(nvpair_value_string(p2, (char **)&valstr2) == 0);
                return (strcmp(valstr1, valstr2) == 0);
        } else {
                uint64_t intval1, intval2;

                VERIFY(nvpair_value_uint64(p1, &intval1) == 0);
                VERIFY(nvpair_value_uint64(p2, &intval2) == 0);
                return (intval1 == intval2);
        }
}

/*
 * Remove properties from props if they are not going to change (as determined
 * by comparison with origprops). Remove them from origprops as well, since we
 * do not need to clear or restore properties that won't change.
 */
static void
props_reduce(nvlist_t *props, nvlist_t *origprops)
{
        nvpair_t *pair, *next_pair;

        if (origprops == NULL)
                return; /* all props need to be received */

        pair = nvlist_next_nvpair(props, NULL);
        while (pair != NULL) {
                const char *propname = nvpair_name(pair);
                nvpair_t *match;

                next_pair = nvlist_next_nvpair(props, pair);

                if ((nvlist_lookup_nvpair(origprops, propname,
                    &match) != 0) || !propval_equals(pair, match))
                        goto next; /* need to set received value */

                /* don't clear the existing received value */
                (void) nvlist_remove_nvpair(origprops, match);
                /* don't bother receiving the property */
                (void) nvlist_remove_nvpair(props, pair);
next:
                pair = next_pair;
        }
}

#ifdef  DEBUG
static boolean_t zfs_ioc_recv_inject_err;
#endif

/*
 * inputs:
 * zc_name              name of containing filesystem
 * zc_nvlist_src{_size} nvlist of properties to apply
 * zc_value             name of snapshot to create
 * zc_string            name of clone origin (if DRR_FLAG_CLONE)
 * zc_cookie            file descriptor to recv from
 * zc_begin_record      the BEGIN record of the stream (not byteswapped)
 * zc_guid              force flag
 * zc_cleanup_fd        cleanup-on-exit file descriptor
 * zc_action_handle     handle for this guid/ds mapping (or zero on first call)
 *
 * outputs:
```

```
 * zc_cookie            number of bytes read
 * zc_nvlist_dst{_size} error for each unapplied received property
 * zc_obj               zprop_errflags_t
 * zc_action_handle     handle for this guid/ds mapping
 */
static int
zfs_ioc_recv(zfs_cmd_t *zc)
{
        file_t *fp;
        objset_t *os;
        dmu_recv_cookie_t drc;
        boolean_t force = (boolean_t)zc->zc_guid;
        int fd;
        int error = 0;
        int props_error = 0;
        nvlist_t *errors;
        offset_t off;
        nvlist_t *props = NULL; /* sent properties */
        nvlist_t *origprops = NULL; /* existing properties */
        objset_t *origin = NULL;
        char *tosnap;
        char tofs[ZFS_MAXNAMELEN];
        boolean_t first_recvd_props = B_FALSE;

        if (dataset_namecheck(zc->zc_value, NULL, NULL) != 0 ||
            strchr(zc->zc_value, '@') == NULL ||
            strchr(zc->zc_value, '%'))
                return (EINVAL);

        (void) strcpy(tofs, zc->zc_value);
        tosnap = strchr(tofs, '@');
        *tosnap++ = '\0';

        if (zc->zc_nvlist_src != NULL &&
            (error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
            zc->zc_iflags, &props)) != 0)
                return (error);

        fd = zc->zc_cookie;
        fp = getf(fd);
        if (fp == NULL) {
                nvlist_free(props);
                return (EBADF);
        }

        VERIFY(nvlist_alloc(&errors, NV_UNIQUE_NAME, KM_SLEEP) == 0);

        if (props && dmu_objset_hold(tofs, FTAG, &os) == 0) {
                if ((spa_version(os->os_spa) >= SPA_VERSION_RECVD_PROPS) &&
                    !dsl_prop_get_hasrecvd(os)) {
                        first_recvd_props = B_TRUE;
                }

                /*
                 * If new received properties are supplied, they are to
                 * completely replace the existing received properties, so stash
                 * away the existing ones.
                 */
                if (dsl_prop_get_received(os, &origprops) == 0) {
                        nvlist_t *errlist = NULL;
                        /*
                         * Don't bother writing a property if its value won't
                         * change (and avoid the unnecessary security checks).
                         *
                         * The first receive after SPA_VERSION_RECVD_PROPS is a
                         * special case where we blow away all local properties
```



```
                         * regardless.
                         */
                        if (!first_recvd_props)
                                props_reduce(props, origprops);
                        if (zfs_check_clearable(tofs, origprops,
                            &errlist) != 0)
                                (void) nvlist_merge(errors, errlist, 0);
                        nvlist_free(errlist);
                }

                dmu_objset_rele(os, FTAG);
        }

        if (zc->zc_string[0]) {
                error = dmu_objset_hold(zc->zc_string, FTAG, &origin);
                if (error)
                        goto out;
        }

        error = dmu_recv_begin(tofs, tosnap, zc->zc_top_ds,
            &zc->zc_begin_record, force, origin, &drc);
        if (origin)
                dmu_objset_rele(origin, FTAG);
        if (error)
                goto out;

        /*
         * Set properties before we receive the stream so that they are applied
         * to the new data. Note that we must call dmu_recv_stream() if
         * dmu_recv_begin() succeeds.
         */
        if (props) {
                if (dmu_objset_from_ds(drc.drc_logical_ds, &os) == 0) {
                        if (drc.drc_newfs) {
                                if (spa_version(os->os_spa) >=
                                    SPA_VERSION_RECVD_PROPS)
                                        first_recvd_props = B_TRUE;
                        } else if (origprops != NULL) {
                                if (clear_received_props(os, tofs, origprops,
                                    first_recvd_props ? NULL : props) != 0)
                                        zc->zc_obj |= ZPROP_ERR_NOCLEAR;
                        } else {
                                zc->zc_obj |= ZPROP_ERR_NOCLEAR;
                        }
                        dsl_prop_set_hasrecvd(os);
                } else if (!drc.drc_newfs) {
                        zc->zc_obj |= ZPROP_ERR_NOCLEAR;
                }

                (void) zfs_set_prop_nvlist(tofs, ZPROP_SRC_RECEIVED,
                    props, errors);
        }

        if (zc->zc_nvlist_dst_size != 0 &&
            (nvlist_smush(errors, zc->zc_nvlist_dst_size) != 0 ||
            put_nvlist(zc, errors) != 0)) {
                /*
                 * Caller made zc->zc_nvlist_dst less than the minimum expected
                 * size or supplied an invalid address.
                 */
                props_error = EINVAL;
        }

        off = fp->f_offset;
        error = dmu_recv_stream(&drc, fp->f_vnode, &off, zc->zc_cleanup_fd,
            &zc->zc_action_handle);
```

```
3953         if (error == 0) {
3954                 zfsvfs_t *zfsvfs = NULL;
3955
3956                 if (getzfsvfs(tofs, &zfsvfs) == 0) {
3957                         /* online recv */
3958                         int end_err;
3959
3960                         error = zfs_suspend_fs(zfsvfs);
3961                         /*
3962                          * If the suspend fails, then the recv_end will
3963                          * likely also fail, and clean up after itself.
3964                          */
3965                         end_err = dmu_recv_end(&drc);
3966                         if (error == 0)
3967                                 error = zfs_resume_fs(zfsvfs, tofs);
3968                         error = error ? error : end_err;
3969                         VFS_RELE(zfsvfs->z_vfs);
3970                 } else {
3971                         error = dmu_recv_end(&drc);
3972                 }
3973         }
3974
3975         zc->zc_cookie = off - fp->f_offset;
3976         if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
3977                 fp->f_offset = off;
3978
3979 #ifdef  DEBUG
3980         if (zfs_ioc_recv_inject_err) {
3981                 zfs_ioc_recv_inject_err = B_FALSE;
3982                 error = 1;
3983         }
3984 #endif
3985         /*
3986          * On error, restore the original props.
3987          */
3988         if (error && props) {
3989                 if (dmu_objset_hold(tofs, FTAG, &os) == 0) {
3990                         if (clear_received_props(os, tofs, props, NULL) != 0) {
3991                                 /*
3992                                  * We failed to clear the received properties.
3993                                  * Since we may have left a $recvd value on the
3994                                  * system, we can't clear the $hasrecvd flag.
3995                                  */
3996                                 zc->zc_obj |= ZPROP_ERR_NORESTORE;
3997                         } else if (first_recvd_props) {
3998                                 dsl_prop_unset_hasrecvd(os);
3999                         }
4000                         dmu_objset_rele(os, FTAG);
4001                 } else if (!drc.drc_newfs) {
4002                         /* We failed to clear the received properties. */
4003                         zc->zc_obj |= ZPROP_ERR_NORESTORE;
4004                 }
4005
4006                 if (origprops == NULL && !drc.drc_newfs) {
4007                         /* We failed to stash the original properties. */
4008                         zc->zc_obj |= ZPROP_ERR_NORESTORE;
4009                 }
4010
4011                 /*
4012                  * dsl_props_set() will not convert RECEIVED to LOCAL on or
4013                  * after SPA_VERSION_RECVD_PROPS, so we need to specify LOCAL
4014                  * explictly if we're restoring local properties cleared in the
4015                  * first new-style receive.
4016                  */
4017                 if (origprops != NULL &&
```



```
4018                     zfs_set_prop_nvlist(tofs, (first_recvd_props ?
4019                     ZPROP_SRC_LOCAL : ZPROP_SRC_RECEIVED),
4020                     origprops, NULL) != 0) {
4021                         /*
4022                          * We stashed the original properties but failed to
4023                          * restore them.
4024                          */
4025                         zc->zc_obj |= ZPROP_ERR_NORESTORE;
4026                 }
4027         }
4028 out:
4029         nvlist_free(props);
4030         nvlist_free(origprops);
4031         nvlist_free(errors);
4032         releasef(fd);
4033
4034         if (error == 0)
4035                 error = props_error;
4036
4037         return (error);
4038 }
4039
4040 /*
4041  * inputs:
4042  * zc_name      name of snapshot to send
4043  * zc_cookie    file descriptor to send stream to
4044  * zc_obj       fromorigin flag (mutually exclusive with zc_fromobj)
4045  * zc_sendobj   objsetid of snapshot to send
4046  * zc_fromobj   objsetid of incremental fromsnap (may be zero)
4047  * zc_guid      bit 0: if set, estimate size of stream only. zc_cookie is
4048  *              ignored. output size in zc_objset_type.
4049  * zc_guid      bit 1: if set, send output in FAR-format
 175  * zc_guid      if set, estimate size of stream only.  zc_cookie is ignored.
 176  *              output size in zc_objset_type.
4050  *
4051  * outputs: none
4052  */
4053 static int
4054 zfs_ioc_send(zfs_cmd_t *zc)
4055 {
4056         objset_t *fromsnap = NULL;
4057         objset_t *tosnap;
4058         int error;
4059         offset_t off;
4060         dsl_dataset_t *ds;
4061         dsl_dataset_t *dsfrom = NULL;
4062         spa_t *spa;
4063         dsl_pool_t *dp;
4064         boolean_t estimate = ((zc->zc_guid & 1) != 0);
4065         boolean_t far = ((zc->zc_guid & 2) != 0);
 191         boolean_t estimate = (zc->zc_guid != 0);
4066
4067         error = spa_open(zc->zc_name, &spa, FTAG);
4068         if (error)
4069                 return (error);
4070
4071         dp = spa_get_dsl(spa);
4072         rw_enter(&dp->dp_config_rwlock, RW_READER);
4073         error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &ds);
4074         rw_exit(&dp->dp_config_rwlock);
4075         spa_close(spa, FTAG);
4076         if (error)
4077                 return (error);
4078
4079         error = dmu_objset_from_ds(ds, &tosnap);
4080         if (error) {
```

```
4081                 dsl_dataset_rele(ds, FTAG);
4082                 return (error);
4083         }

4085         if (zc->zc_fromobj != 0) {
4086                 rw_enter(&dp->dp_config_rwlock, RW_READER);
4087                 error = dsl_dataset_hold_obj(dp, zc->zc_fromobj, FTAG, &dsfrom);
4088                 rw_exit(&dp->dp_config_rwlock);
4089                 if (error) {
4090                         dsl_dataset_rele(ds, FTAG);
4091                         return (error);
4092                 }
4093                 error = dmu_objset_from_ds(dsfrom, &fromsnap);
4094                 if (error) {
4095                         dsl_dataset_rele(dsfrom, FTAG);
4096                         dsl_dataset_rele(ds, FTAG);
4097                         return (error);
4098                 }
4099         }

4101         if (zc->zc_obj) {
4102                 dsl_pool_t *dp = ds->ds_dir->dd_pool;

4104                 if (fromsnap != NULL) {
4105                         dsl_dataset_rele(dsfrom, FTAG);
4106                         dsl_dataset_rele(ds, FTAG);
4107                         return (EINVAL);
4108                 }

4110                 if (dsl_dir_is_clone(ds->ds_dir)) {
4111                         rw_enter(&dp->dp_config_rwlock, RW_READER);
4112                         error = dsl_dataset_hold_obj(dp,
4113                             ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &dsfrom);
4114                         rw_exit(&dp->dp_config_rwlock);
4115                         if (error) {
4116                                 dsl_dataset_rele(ds, FTAG);
4117                                 return (error);
4118                         }
4119                         error = dmu_objset_from_ds(dsfrom, &fromsnap);
4120                         if (error) {
4121                                 dsl_dataset_rele(dsfrom, FTAG);
4122                                 dsl_dataset_rele(ds, FTAG);
4123                                 return (error);
4124                         }
4125                 }
4126         }

4128         if (estimate) {
4129                 error = dmu_send_estimate(tosnap, fromsnap,
4130                     &zc->zc_objset_type);
4131         } else {
4132                 file_t *fp = getf(zc->zc_cookie);
4133                 if (fp == NULL) {
4134                         dsl_dataset_rele(ds, FTAG);
4135                         if (dsfrom)
4136                                 dsl_dataset_rele(dsfrom, FTAG);
4137                         return (EBADF);
4138                 }

4140                 off = fp->f_offset;
4141                 if (!far)
4142 #endif /* ! codereview */
4143                         error = dmu_send(tosnap, fromsnap,
4144                             zc->zc_cookie, fp->f_vnode, &off);
4145                 else
4146                         error = far_send(tosnap, fromsnap,
```



```
4147                             zc->zc_cookie, fp->f_vnode, &off);
4148 #endif /* ! codereview */

4150                 if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
4151                         fp->f_offset = off;
4152                 releasef(zc->zc_cookie);
4153         }
4154         if (dsfrom)
4155                 dsl_dataset_rele(dsfrom, FTAG);
4156         dsl_dataset_rele(ds, FTAG);
4157         return (error);
4158 }

4160 /*
4161  * inputs:
4162  * zc_name      name of snapshot on which to report progress
4163  * zc_cookie    file descriptor of send stream
4164  *
4165  * outputs:
4166  * zc_cookie    number of bytes written in send stream thus far
4167  */
4168 static int
4169 zfs_ioc_send_progress(zfs_cmd_t *zc)
4170 {
4171         dsl_dataset_t *ds;
4172         dmu_sendarg_t *dsp = NULL;
4173         int error;

4175         if ((error = dsl_dataset_hold(zc->zc_name, FTAG, &ds)) != 0)
4176                 return (error);

4178         mutex_enter(&ds->ds_sendstream_lock);

4180         /*
4181          * Iterate over all the send streams currently active on this dataset.
4182          * If there's one which matches the specified file descriptor _and_ the
4183          * stream was started by the current process, return the progress of
4184          * that stream.
4185          */
4186         for (dsp = list_head(&ds->ds_sendstreams); dsp != NULL;
4187             dsp = list_next(&ds->ds_sendstreams, dsp)) {
4188                 if (dsp->dsa_outfd == zc->zc_cookie &&
4189                     dsp->dsa_proc == curproc)
4190                         break;
4191         }

4193         if (dsp != NULL)
4194                 zc->zc_cookie = *(dsp->dsa_off);
4195         else
4196                 error = ENOENT;

4198         mutex_exit(&ds->ds_sendstream_lock);
4199         dsl_dataset_rele(ds, FTAG);
4200         return (error);
4201 }

4203 static int
4204 zfs_ioc_inject_fault(zfs_cmd_t *zc)
4205 {
4206         int id, error;

4208         error = zio_inject_fault(zc->zc_name, (int)zc->zc_guid, &id,
4209             &zc->zc_inject_record);

4211         if (error == 0)
4212                 zc->zc_guid = (uint64_t)id;
```

```
4214         return (error);
4215 }

4217 static int
4218 zfs_ioc_clear_fault(zfs_cmd_t *zc)
4219 {
4220         return (zio_clear_fault((int)zc->zc_guid));
4221 }

4223 static int
4224 zfs_ioc_inject_list_next(zfs_cmd_t *zc)
4225 {
4226         int id = (int)zc->zc_guid;
4227         int error;

4229         error = zio_inject_list_next(&id, zc->zc_name, sizeof (zc->zc_name),
4230             &zc->zc_inject_record);

4232         zc->zc_guid = id;

4234         return (error);
4235 }

4237 static int
4238 zfs_ioc_error_log(zfs_cmd_t *zc)
4239 {
4240         spa_t *spa;
4241         int error;
4242         size_t count = (size_t)zc->zc_nvlist_dst_size;

4244         if ((error = spa_open(zc->zc_name, &spa, FTAG)) != 0)
4245                 return (error);

4247         error = spa_get_errlog(spa, (void *)(uintptr_t)zc->zc_nvlist_dst,
4248             &count);
4249         if (error == 0)
4250                 zc->zc_nvlist_dst_size = count;
4251         else
4252                 zc->zc_nvlist_dst_size = spa_get_errlog_size(spa);

4254         spa_close(spa, FTAG);

4256         return (error);
4257 }

4259 static int
4260 zfs_ioc_clear(zfs_cmd_t *zc)
4261 {
4262         spa_t *spa;
4263         vdev_t *vd;
4264         int error;

4266         /*
4267          * On zpool clear we also fix up missing slogs
4268          */
4269         mutex_enter(&spa_namespace_lock);
4270         spa = spa_lookup(zc->zc_name);
4271         if (spa == NULL) {
4272                 mutex_exit(&spa_namespace_lock);
4273                 return (EIO);
4274         }
4275         if (spa_get_log_state(spa) == SPA_LOG_MISSING) {
4276                 /* we need to let spa_open/spa_load clear the chains */
4277                 spa_set_log_state(spa, SPA_LOG_CLEAR);
4278         }
```



```
4279         spa->spa_last_open_failed = 0;
4280         mutex_exit(&spa_namespace_lock);

4282         if (zc->zc_cookie & ZPOOL_NO_REWIND) {
4283                 error = spa_open(zc->zc_name, &spa, FTAG);
4284         } else {
4285                 nvlist_t *policy;
4286                 nvlist_t *config = NULL;

4288                 if (zc->zc_nvlist_src == NULL)
4289                         return (EINVAL);

4291                 if ((error = get_nvlist(zc->zc_nvlist_src,
4292                     zc->zc_nvlist_src_size, zc->zc_iflags, &policy)) == 0) {
4293                         error = spa_open_rewind(zc->zc_name, &spa, FTAG,
4294                             policy, &config);
4295                         if (config != NULL) {
4296                                 int err;

4298                                 if ((err = put_nvlist(zc, config)) != 0)
4299                                         error = err;
4300                                 nvlist_free(config);
4301                         }
4302                         nvlist_free(policy);
4303                 }
4304         }

4306         if (error)
4307                 return (error);

4309         spa_vdev_state_enter(spa, SCL_NONE);

4311         if (zc->zc_guid == 0) {
4312                 vd = NULL;
4313         } else {
4314                 vd = spa_lookup_by_guid(spa, zc->zc_guid, B_TRUE);
4315                 if (vd == NULL) {
4316                         (void) spa_vdev_state_exit(spa, NULL, ENODEV);
4317                         spa_close(spa, FTAG);
4318                         return (ENODEV);
4319                 }
4320         }

4322         vdev_clear(spa, vd);

4324         (void) spa_vdev_state_exit(spa, NULL, 0);

4326         /*
4327          * Resume any suspended I/Os.
4328          */
4329         if (zio_resume(spa) != 0)
4330                 error = EIO;

4332         spa_close(spa, FTAG);

4334         return (error);
4335 }

4337 static int
4338 zfs_ioc_pool_reopen(zfs_cmd_t *zc)
4339 {
4340         spa_t *spa;
4341         int error;

4343         error = spa_open(zc->zc_name, &spa, FTAG);
4344         if (error)
```

```
4345                 return (error);

4347         spa_vdev_state_enter(spa, SCL_NONE);

4349         /*
4350          * If a resilver is already in progress then set the
4351          * spa_scrub_reopen flag to B_TRUE so that we don't restart
4352          * the scan as a side effect of the reopen. Otherwise, let
4353          * vdev_open() decided if a resilver is required.
4354          */
4355         spa->spa_scrub_reopen = dsl_scan_resilvering(spa->spa_dsl_pool);
4356         vdev_reopen(spa->spa_root_vdev);
4357         spa->spa_scrub_reopen = B_FALSE;

4359         (void) spa_vdev_state_exit(spa, NULL, 0);
4360         spa_close(spa, FTAG);
4361         return (0);
4362 }
4363 /*
4364  * inputs:
4365  * zc_name      name of filesystem
4366  * zc_value     name of origin snapshot
4367  *
4368  * outputs:
4369  * zc_string    name of conflicting snapshot, if there is one
4370  */
4371 static int
4372 zfs_ioc_promote(zfs_cmd_t *zc)
4373 {
4374         char *cp;

4376         /*
4377          * We don't need to unmount *all* the origin fs's snapshots, but
4378          * it's easier.
4379          */
4380         cp = strchr(zc->zc_value, '@');
4381         if (cp)
4382                 *cp = '\0';
4383         (void) dmu_objset_find(zc->zc_value,
4384             zfs_unmount_snap, NULL, DS_FIND_SNAPSHOTS);
4385         return (dsl_dataset_promote(zc->zc_name, zc->zc_string));
4386 }

4388 /*
4389  * Retrieve a single {user|group}{used|quota}@... property.
4390  *
4391  * inputs:
4392  * zc_name      name of filesystem
4393  * zc_objset_type zfs_userquota_prop_t
4394  * zc_value     domain name (eg. "S-1-234-567-89")
4395  * zc_guid      RID/UID/GID
4396  *
4397  * outputs:
4398  * zc_cookie    property value
4399  */
4400 static int
4401 zfs_ioc_userspace_one(zfs_cmd_t *zc)
4402 {
4403         zfsvfs_t *zfsvfs;
4404         int error;

4406         if (zc->zc_objset_type >= ZFS_NUM_USERQUOTA_PROPS)
4407                 return (EINVAL);

4409         error = zfsvfs_hold(zc->zc_name, FTAG, &zfsvfs, B_FALSE);
4410         if (error)
```



```
4411                 return (error);

4413         error = zfs_userspace_one(zfsvfs,
4414             zc->zc_objset_type, zc->zc_value, zc->zc_guid, &zc->zc_cookie);
4415         zfsvfs_rele(zfsvfs, FTAG);

4417         return (error);
4418 }

4420 /*
4421  * inputs:
4422  * zc_name              name of filesystem
4423  * zc_cookie            zap cursor
4424  * zc_objset_type       zfs_userquota_prop_t
4425  * zc_nvlist_dst[_size] buffer to fill (not really an nvlist)
4426  *
4427  * outputs:
4428  * zc_nvlist_dst[_size] data buffer (array of zfs_useracct_t)
4429  * zc_cookie    zap cursor
4430  */
4431 static int
4432 zfs_ioc_userspace_many(zfs_cmd_t *zc)
4433 {
4434         zfsvfs_t *zfsvfs;
4435         int bufsize = zc->zc_nvlist_dst_size;

4437         if (bufsize <= 0)
4438                 return (ENOMEM);

4440         int error = zfsvfs_hold(zc->zc_name, FTAG, &zfsvfs, B_FALSE);
4441         if (error)
4442                 return (error);

4444         void *buf = kmem_alloc(bufsize, KM_SLEEP);

4446         error = zfs_userspace_many(zfsvfs, zc->zc_objset_type, &zc->zc_cookie,
4447             buf, &zc->zc_nvlist_dst_size);

4449         if (error == 0) {
4450                 error = xcopyout(buf,
4451                     (void *)(uintptr_t)zc->zc_nvlist_dst,
4452                     zc->zc_nvlist_dst_size);
4453         }
4454         kmem_free(buf, bufsize);
4455         zfsvfs_rele(zfsvfs, FTAG);

4457         return (error);
4458 }

4460 /*
4461  * inputs:
4462  * zc_name              name of filesystem
4463  *
4464  * outputs:
4465  * none
4466  */
4467 static int
4468 zfs_ioc_userspace_upgrade(zfs_cmd_t *zc)
4469 {
4470         objset_t *os;
4471         int error = 0;
4472         zfsvfs_t *zfsvfs;

4474         if (getzfsvfs(zc->zc_name, &zfsvfs) == 0) {
4475                 if (!dmu_objset_userused_enabled(zfsvfs->z_os)) {
4476                         /*
```

```
4477                         * If userused is not enabled, it may be because the
4478                         * objset needs to be closed & reopened (to grow the
4479                         * objset_phys_t).  Suspend/resume the fs will do that.
4480                         */
4481                        error = zfs_suspend_fs(zfsvfs);
4482                        if (error == 0)
4483                                error = zfs_resume_fs(zfsvfs, zc->zc_name);
4484                }
4485                if (error == 0)
4486                        error = dmu_objset_userspace_upgrade(zfsvfs->z_os);
4487                VFS_RELE(zfsvfs->z_vfs);
4488        } else {
4489                /* XXX kind of reading contents without owning */
4490                error = dmu_objset_hold(zc->zc_name, FTAG, &os);
4491                if (error)
4492                        return (error);

4494                error = dmu_objset_userspace_upgrade(os);
4495                dmu_objset_rele(os, FTAG);
4496        }

4498        return (error);
4499 }

4501 /*
4502  * We don't want to have a hard dependency
4503  * against some special symbols in sharefs
4504  * nfs, and smbsrv.  Determine them if needed when
4505  * the first file system is shared.
4506  * Neither sharefs, nfs or smbsrv are unloadable modules.
4507  */
4508 int (*znfsexport_fs)(void *arg);
4509 int (*zshare_fs)(enum sharefs_sys_op, share_t *, uint32_t);
4510 int (*zsmbexport_fs)(void *arg, boolean_t add_share);

4512 int zfs_nfsshare_inited;
4513 int zfs_smbshare_inited;

4515 ddi_modhandle_t nfs_mod;
4516 ddi_modhandle_t sharefs_mod;
4517 ddi_modhandle_t smbsrv_mod;
4518 kmutex_t zfs_share_lock;

4520 static int
4521 zfs_init_sharefs()
4522 {
4523        int error;

4525        ASSERT(MUTEX_HELD(&zfs_share_lock));
4526        /* Both NFS and SMB shares also require sharetab support. */
4527        if (sharefs_mod == NULL && ((sharefs_mod =
4528            ddi_modopen("fs/sharefs",
4529            KRTLD_MODE_FIRST, &error)) == NULL)) {
4530                return (ENOSYS);
4531        }
4532        if (zshare_fs == NULL && ((zshare_fs =
4533            (int (*)(enum sharefs_sys_op, share_t *, uint32_t))
4534            ddi_modsym(sharefs_mod, "sharefs_impl", &error)) == NULL)) {
4535                return (ENOSYS);
4536        }
4537        return (0);
4538 }

4540 static int
4541 zfs_ioc_share(zfs_cmd_t *zc)
4542 {
```



```
4543        int error;
4544        int opcode;

4546        switch (zc->zc_share.z_sharetype) {
4547        case ZFS_SHARE_NFS:
4548        case ZFS_UNSHARE_NFS:
4549                if (zfs_nfsshare_inited == 0) {
4550                        mutex_enter(&zfs_share_lock);
4551                        if (nfs_mod == NULL && ((nfs_mod = ddi_modopen("fs/nfs",
4552                            KRTLD_MODE_FIRST, &error)) == NULL)) {
4553                                mutex_exit(&zfs_share_lock);
4554                                return (ENOSYS);
4555                        }
4556                        if (znfsexport_fs == NULL &&
4557                            ((znfsexport_fs = (int (*)(void *))
4558                            ddi_modsym(nfs_mod,
4559                            "nfs_export", &error)) == NULL)) {
4560                                mutex_exit(&zfs_share_lock);
4561                                return (ENOSYS);
4562                        }
4563                        error = zfs_init_sharefs();
4564                        if (error) {
4565                                mutex_exit(&zfs_share_lock);
4566                                return (ENOSYS);
4567                        }
4568                        zfs_nfsshare_inited = 1;
4569                        mutex_exit(&zfs_share_lock);
4570                }
4571                break;
4572        case ZFS_SHARE_SMB:
4573        case ZFS_UNSHARE_SMB:
4574                if (zfs_smbshare_inited == 0) {
4575                        mutex_enter(&zfs_share_lock);
4576                        if (smbsrv_mod == NULL && ((smbsrv_mod =
4577                            ddi_modopen("drv/smbsrv",
4578                            KRTLD_MODE_FIRST, &error)) == NULL)) {
4579                                mutex_exit(&zfs_share_lock);
4580                                return (ENOSYS);
4581                        }
4582                        if (zsmbexport_fs == NULL && ((zsmbexport_fs =
4583                            (int (*)(void *, boolean_t))ddi_modsym(smbsrv_mod,
4584                            "smb_server_share", &error)) == NULL)) {
4585                                mutex_exit(&zfs_share_lock);
4586                                return (ENOSYS);
4587                        }
4588                        error = zfs_init_sharefs();
4589                        if (error) {
4590                                mutex_exit(&zfs_share_lock);
4591                                return (ENOSYS);
4592                        }
4593                        zfs_smbshare_inited = 1;
4594                        mutex_exit(&zfs_share_lock);
4595                }
4596                break;
4597        default:
4598                return (EINVAL);
4599        }

4601        switch (zc->zc_share.z_sharetype) {
4602        case ZFS_SHARE_NFS:
4603        case ZFS_UNSHARE_NFS:
4604                if (error =
4605                    znfsexport_fs((void *)
4606                    (uintptr_t)zc->zc_share.z_exportdata))
4607                        return (error);
4608                break;
```

```
4609         case ZFS_SHARE_SMB:
4610         case ZFS_UNSHARE_SMB:
4611                 if (error = zsmbexport_fs((void *)
4612                     (uintptr_t)zc->zc_share.z_exportdata,
4613                     zc->zc_share.z_sharetype == ZFS_SHARE_SMB ?
4614                     B_TRUE: B_FALSE)) {
4615                         return (error);
4616                 }
4617                 break;
4618         }

4620         opcode = (zc->zc_share.z_sharetype == ZFS_SHARE_NFS ||
4621             zc->zc_share.z_sharetype == ZFS_SHARE_SMB) ?
4622             SHAREFS_ADD : SHAREFS_REMOVE;

4624         /*
4625          * Add or remove share from sharetab
4626          */
4627         error = zshare_fs(opcode,
4628             (void *)(uintptr_t)zc->zc_share.z_sharedata,
4629             zc->zc_share.z_sharemax);

4631         return (error);

4633 }

4635 ace_t full_access[] = {
4636         {(uid_t)-1, ACE_ALL_PERMS, ACE_EVERYONE, 0}
4637 };

4639 /*
4640  * inputs:
4641  * zc_name              name of containing filesystem
4642  * zc_obj               object # beyond which we want next in-use object #
4643  *
4644  * outputs:
4645  * zc_obj               next in-use object #
4646  */
4647 static int
4648 zfs_ioc_next_obj(zfs_cmd_t *zc)
4649 {
4650         objset_t *os = NULL;
4651         int error;

4653         error = dmu_objset_hold(zc->zc_name, FTAG, &os);
4654         if (error)
4655                 return (error);

4657         error = dmu_object_next(os, &zc->zc_obj, B_FALSE,
4658             os->os_dsl_dataset->ds_phys->ds_prev_snap_txg);

4660         dmu_objset_rele(os, FTAG);
4661         return (error);
4662 }

4664 /*
4665  * inputs:
4666  * zc_name              name of filesystem
4667  * zc_value             prefix name for snapshot
4668  * zc_cleanup_fd        cleanup-on-exit file descriptor for calling process
4669  *
4670  * outputs:
4671  * zc_value             short name of new snapshot
4672  */
4673 static int
4674 zfs_ioc_tmp_snapshot(zfs_cmd_t *zc)
```



```
4675 {
4676         char *snap_name;
4677         int error;

4679         snap_name = kmem_asprintf("%s@%s-%016llx", zc->zc_name, zc->zc_value,
4680             (u_longlong_t)ddi_get_lbolt64());

4682         if (strlen(snap_name) >= MAXPATHLEN) {
4683                 strfree(snap_name);
4684                 return (E2BIG);
4685         }

4687         error = dmu_objset_snapshot_tmp(snap_name, "%temp", zc->zc_cleanup_fd);
4688         if (error != 0) {
4689                 strfree(snap_name);
4690                 return (error);
4691         }

4693         (void) strcpy(zc->zc_value, strchr(snap_name, '@') + 1);
4694         strfree(snap_name);
4695         return (0);
4696 }

4698 /*
4699  * inputs:
4700  * zc_name              name of "to" snapshot
4701  * zc_value             name of "from" snapshot
4702  * zc_cookie            file descriptor to write diff data on
4703  *
4704  * outputs:
4705  * dmu_diff_record_t's to the file descriptor
4706  */
4707 static int
4708 zfs_ioc_diff(zfs_cmd_t *zc)
4709 {
4710         objset_t *fromsnap;
4711         objset_t *tosnap;
4712         file_t *fp;
4713         offset_t off;
4714         int error;

4716         error = dmu_objset_hold(zc->zc_name, FTAG, &tosnap);
4717         if (error)
4718                 return (error);

4720         error = dmu_objset_hold(zc->zc_value, FTAG, &fromsnap);
4721         if (error) {
4722                 dmu_objset_rele(tosnap, FTAG);
4723                 return (error);
4724         }

4726         fp = getf(zc->zc_cookie);
4727         if (fp == NULL) {
4728                 dmu_objset_rele(fromsnap, FTAG);
4729                 dmu_objset_rele(tosnap, FTAG);
4730                 return (EBADF);
4731         }

4733         off = fp->f_offset;

4735         error = dmu_diff(tosnap, fromsnap, fp->f_vnode, &off);

4737         if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
4738                 fp->f_offset = off;
4739         releasef(zc->zc_cookie);
```

```
4741         dmu_objset_rele(fromsnap, FTAG);
4742         dmu_objset_rele(tosnap, FTAG);
4743         return (error);
4744 }

4746 /*
4747  * Remove all ACL files in shares dir
4748  */
4749 static int
4750 zfs_smb_acl_purge(znode_t *dzp)
4751 {
4752         zap_cursor_t    zc;
4753         zap_attribute_t zap;
4754         zfsvfs_t *zfsvfs = dzp->z_zfsvfs;
4755         int error;

4757         for (zap_cursor_init(&zc, zfsvfs->z_os, dzp->z_id);
4758             (error = zap_cursor_retrieve(&zc, &zap)) == 0;
4759             zap_cursor_advance(&zc)) {
4760                 if ((error = VOP_REMOVE(ZTOV(dzp), zap.za_name, kcred,
4761                     NULL, 0)) != 0)
4762                         break;
4763         }
4764         zap_cursor_fini(&zc);
4765         return (error);
4766 }

4768 static int
4769 zfs_ioc_smb_acl(zfs_cmd_t *zc)
4770 {
4771         vnode_t *vp;
4772         znode_t *dzp;
4773         vnode_t *resourcevp = NULL;
4774         znode_t *sharedir;
4775         zfsvfs_t *zfsvfs;
4776         nvlist_t *nvlist;
4777         char *src, *target;
4778         vattr_t vattr;
4779         vsecattr_t vsec;
4780         int error = 0;

4782         if ((error = lookupname(zc->zc_value, UIO_SYSSPACE,
4783             NO_FOLLOW, NULL, &vp)) != 0)
4784                 return (error);

4786         /* Now make sure mntpnt and dataset are ZFS */

4788         if (vp->v_vfsp->vfs_fstype != zfsfstype ||
4789             (strcmp((char *)refstr_value(vp->v_vfsp->vfs_resource),
4790             zc->zc_name) != 0)) {
4791                 VN_RELE(vp);
4792                 return (EINVAL);
4793         }

4795         dzp = VTOZ(vp);
4796         zfsvfs = dzp->z_zfsvfs;
4797         ZFS_ENTER(zfsvfs);

4799         /*
4800          * Create share dir if its missing.
4801          */
4802         mutex_enter(&zfsvfs->z_lock);
4803         if (zfsvfs->z_shares_dir == 0) {
4804                 dmu_tx_t *tx;

4806                 tx = dmu_tx_create(zfsvfs->z_os);
```



```
4807                 dmu_tx_hold_zap(tx, MASTER_NODE_OBJ, TRUE,
4808                     ZFS_SHARES_DIR);
4809                 dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, FALSE, NULL);
4810                 error = dmu_tx_assign(tx, TXG_WAIT);
4811                 if (error) {
4812                         dmu_tx_abort(tx);
4813                 } else {
4814                         error = zfs_create_share_dir(zfsvfs, tx);
4815                         dmu_tx_commit(tx);
4816                 }
4817                 if (error) {
4818                         mutex_exit(&zfsvfs->z_lock);
4819                         VN_RELE(vp);
4820                         ZFS_EXIT(zfsvfs);
4821                         return (error);
4822                 }
4823         }
4824         mutex_exit(&zfsvfs->z_lock);

4826         ASSERT(zfsvfs->z_shares_dir);
4827         if ((error = zfs_zget(zfsvfs, zfsvfs->z_shares_dir, &sharedir)) != 0) {
4828                 VN_RELE(vp);
4829                 ZFS_EXIT(zfsvfs);
4830                 return (error);
4831         }

4833         switch (zc->zc_cookie) {
4834         case ZFS_SMB_ACL_ADD:
4835                 vattr.va_mask = AT_MODE|AT_UID|AT_GID|AT_TYPE;
4836                 vattr.va_type = VREG;
4837                 vattr.va_mode = S_IFREG|0777;
4838                 vattr.va_uid = 0;
4839                 vattr.va_gid = 0;

4841                 vsec.vsa_mask = VSA_ACE;
4842                 vsec.vsa_aclentp = &full_access;
4843                 vsec.vsa_aclentsz = sizeof (full_access);
4844                 vsec.vsa_aclcnt = 1;

4846                 error = VOP_CREATE(ZTOV(sharedir), zc->zc_string,
4847                     &vattr, EXCL, 0, &resourcevp, kcred, 0, NULL, &vsec);
4848                 if (resourcevp)
4849                         VN_RELE(resourcevp);
4850                 break;

4852         case ZFS_SMB_ACL_REMOVE:
4853                 error = VOP_REMOVE(ZTOV(sharedir), zc->zc_string, kcred,
4854                     NULL, 0);
4855                 break;

4857         case ZFS_SMB_ACL_RENAME:
4858                 if ((error = get_nvlist(zc->zc_nvlist_src,
4859                     zc->zc_nvlist_src_size, zc->zc_iflags, &nvlist)) != 0) {
4860                         VN_RELE(vp);
4861                         ZFS_EXIT(zfsvfs);
4862                         return (error);
4863                 }
4864                 if (nvlist_lookup_string(nvlist, ZFS_SMB_ACL_SRC, &src) ||
4865                     nvlist_lookup_string(nvlist, ZFS_SMB_ACL_TARGET,
4866                     &target)) {
4867                         VN_RELE(vp);
4868                         VN_RELE(ZTOV(sharedir));
4869                         ZFS_EXIT(zfsvfs);
4870                         nvlist_free(nvlist);
4871                         return (error);
4872                 }
```

```
4873                 error = VOP_RENAME(ZTOV(sharedir), src, ZTOV(sharedir), target,
4874                     kcred, NULL, 0);
4875                 nvlist_free(nvlist);
4876                 break;
4877
4878         case ZFS_SMB_ACL_PURGE:
4879                 error = zfs_smb_acl_purge(sharedir);
4880                 break;
4881
4882         default:
4883                 error = EINVAL;
4884                 break;
4885         }
4886
4887         VN_RELE(vp);
4888         VN_RELE(ZTOV(sharedir));
4889
4890         ZFS_EXIT(zfsvfs);
4891
4892         return (error);
4893 }
4894
4895 /*
4896  * inputs:
4897  * zc_name              name of filesystem
4898  * zc_value             short name of snap
4899  * zc_string            user-supplied tag for this hold
4900  * zc_cookie            recursive flag
4901  * zc_temphold          set if hold is temporary
4902  * zc_cleanup_fd        cleanup-on-exit file descriptor for calling process
4903  * zc_sendobj           if non-zero, the objid for zc_name@zc_value
4904  * zc_createtxg         if zc_sendobj is non-zero, snap must have zc_createtxg
4905  *
4906  * outputs:             none
4907  */
4908 static int
4909 zfs_ioc_hold(zfs_cmd_t *zc)
4910 {
4911         boolean_t recursive = zc->zc_cookie;
4912         spa_t *spa;
4913         dsl_pool_t *dp;
4914         dsl_dataset_t *ds;
4915         int error;
4916         minor_t minor = 0;
4917
4918         if (snapshot_namecheck(zc->zc_value, NULL, NULL) != 0)
4919                 return (EINVAL);
4920
4921         if (zc->zc_sendobj == 0) {
4922                 return (dsl_dataset_user_hold(zc->zc_name, zc->zc_value,
4923                     zc->zc_string, recursive, zc->zc_temphold,
4924                     zc->zc_cleanup_fd));
4925         }
4926
4927         if (recursive)
4928                 return (EINVAL);
4929
4930         error = spa_open(zc->zc_name, &spa, FTAG);
4931         if (error)
4932                 return (error);
4933
4934         dp = spa_get_dsl(spa);
4935         rw_enter(&dp->dp_config_rwlock, RW_READER);
4936         error = dsl_dataset_hold_obj(dp, zc->zc_sendobj, FTAG, &ds);
4937         rw_exit(&dp->dp_config_rwlock);
4938         spa_close(spa, FTAG);
```



```
4939         if (error)
4940                 return (error);
4941
4942         /*
4943          * Until we have a hold on this snapshot, it's possible that
4944          * zc_sendobj could've been destroyed and reused as part
4945          * of a later txg.  Make sure we're looking at the right object.
4946          */
4947         if (zc->zc_createtxg != ds->ds_phys->ds_creation_txg) {
4948                 dsl_dataset_rele(ds, FTAG);
4949                 return (ENOENT);
4950         }
4951
4952         if (zc->zc_cleanup_fd != -1 && zc->zc_temphold) {
4953                 error = zfs_onexit_fd_hold(zc->zc_cleanup_fd, &minor);
4954                 if (error) {
4955                         dsl_dataset_rele(ds, FTAG);
4956                         return (error);
4957                 }
4958         }
4959
4960         error = dsl_dataset_user_hold_for_send(ds, zc->zc_string,
4961             zc->zc_temphold);
4962         if (minor != 0) {
4963                 if (error == 0) {
4964                         dsl_register_onexit_hold_cleanup(ds, zc->zc_string,
4965                             minor);
4966                 }
4967                 zfs_onexit_fd_rele(zc->zc_cleanup_fd);
4968         }
4969         dsl_dataset_rele(ds, FTAG);
4970
4971         return (error);
4972 }
4973
4974 /*
4975  * inputs:
4976  * zc_name      name of dataset from which we're releasing a user hold
4977  * zc_value     short name of snap
4978  * zc_string    user-supplied tag for this hold
4979  * zc_cookie    recursive flag
4980  *
4981  * outputs:     none
4982  */
4983 static int
4984 zfs_ioc_release(zfs_cmd_t *zc)
4985 {
4986         boolean_t recursive = zc->zc_cookie;
4987
4988         if (snapshot_namecheck(zc->zc_value, NULL, NULL) != 0)
4989                 return (EINVAL);
4990
4991         return (dsl_dataset_user_release(zc->zc_name, zc->zc_value,
4992             zc->zc_string, recursive));
4993 }
4994
4995 /*
4996  * inputs:
4997  * zc_name              name of filesystem
4998  *
4999  * outputs:
5000  * zc_nvlist_src{_size} nvlist of snapshot holds
5001  */
5002 static int
5003 zfs_ioc_get_holds(zfs_cmd_t *zc)
5004 {
```

```
5005         nvlist_t *nvp;
5006         int error;

5008         if ((error = dsl_dataset_get_holds(zc->zc_name, &nvp)) == 0) {
5009                 error = put_nvlist(zc, nvp);
5010                 nvlist_free(nvp);
5011         }

5013         return (error);
5014 }

5016 /*
5017  * inputs:
5018  * zc_name              name of new filesystem or snapshot
5019  * zc_value             full name of old snapshot
5020  *
5021  * outputs:
5022  * zc_cookie            space in bytes
5023  * zc_objset_type       compressed space in bytes
5024  * zc_perm_action       uncompressed space in bytes
5025  */
5026 static int
5027 zfs_ioc_space_written(zfs_cmd_t *zc)
5028 {
5029         int error;
5030         dsl_dataset_t *new, *old;

5032         error = dsl_dataset_hold(zc->zc_name, FTAG, &new);
5033         if (error != 0)
5034                 return (error);
5035         error = dsl_dataset_hold(zc->zc_value, FTAG, &old);
5036         if (error != 0) {
5037                 dsl_dataset_rele(new, FTAG);
5038                 return (error);
5039         }

5041         error = dsl_dataset_space_written(old, new, &zc->zc_cookie,
5042             &zc->zc_objset_type, &zc->zc_perm_action);
5043         dsl_dataset_rele(old, FTAG);
5044         dsl_dataset_rele(new, FTAG);
5045         return (error);
5046 }
5047 /*
5048  * innvl: {
5049  *     "firstsnap" -> snapshot name
5050  * }
5051  *
5052  * outnvl: {
5053  *     "used" -> space in bytes
5054  *     "compressed" -> compressed space in bytes
5055  *     "uncompressed" -> uncompressed space in bytes
5056  * }
5057  */
5058 static int
5059 zfs_ioc_space_snaps(const char *lastsnap, nvlist_t *innvl, nvlist_t *outnvl)
5060 {
5061         int error;
5062         dsl_dataset_t *new, *old;
5063         char *firstsnap;
5064         uint64_t used, comp, uncomp;

5066         if (nvlist_lookup_string(innvl, "firstsnap", &firstsnap) != 0)
5067                 return (EINVAL);

5069         error = dsl_dataset_hold(lastsnap, FTAG, &new);
5070         if (error != 0)
```



```
5071                 return (error);
5072         error = dsl_dataset_hold(firstsnap, FTAG, &old);
5073         if (error != 0) {
5074                 dsl_dataset_rele(new, FTAG);
5075                 return (error);
5076         }

5078         error = dsl_dataset_space_wouldfree(old, new, &used, &comp, &uncomp);
5079         dsl_dataset_rele(old, FTAG);
5080         dsl_dataset_rele(new, FTAG);
5081         fnvlist_add_uint64(outnvl, "used", used);
5082         fnvlist_add_uint64(outnvl, "compressed", comp);
5083         fnvlist_add_uint64(outnvl, "uncompressed", uncomp);
5084         return (error);
5085 }

5087 /*
5088  * innvl: {
5089  *     "fd" -> file descriptor to write stream to (int32)
5090  *     (optional) "fromsnap" -> full snap name to send an incremental from
5091  * }
5092  *
5093  * outnvl is unused
5094  */
5095 /* ARGSUSED */
5096 static int
5097 zfs_ioc_send_new(const char *snapname, nvlist_t *innvl, nvlist_t *outnvl)
5098 {
5099         objset_t *fromsnap = NULL;
5100         objset_t *tosnap;
5101         int error;
5102         offset_t off;
5103         char *fromname;
5104         int fd;
5105         int far;
5106 #endif /* ! codereview */

5108         error = nvlist_lookup_int32(innvl, "fd", &fd);
5109         if (error != 0)
5110                 return (EINVAL);

5112         error = nvlist_lookup_int32(innvl, "far", &far);
5113         if (error != 0 && error != ENOENT)
5114                 return (EINVAL);

5116 #endif /* ! codereview */
5117         error = dmu_objset_hold(snapname, FTAG, &tosnap);
5118         if (error)
5119                 return (error);

5121         error = nvlist_lookup_string(innvl, "fromsnap", &fromname);
5122         if (error == 0) {
5123                 error = dmu_objset_hold(fromname, FTAG, &fromsnap);
5124                 if (error) {
5125                         dmu_objset_rele(tosnap, FTAG);
5126                         return (error);
5127                 }
5128         }

5130         file_t *fp = getf(fd);
5131         if (fp == NULL) {
5132                 dmu_objset_rele(tosnap, FTAG);
5133                 if (fromsnap != NULL)
5134                         dmu_objset_rele(fromsnap, FTAG);
5135                 return (EBADF);
5136         }
```

```
5138         off = fp->f_offset;
5139         if (!far)
5140 #endif /* ! codereview */
5141                 error = dmu_send(tosnap, fromsnap, fd, fp->f_vnode, &off);
5142         else
5143                 error = far_send(tosnap, fromsnap, fd, fp->f_vnode, &off);
5144 #endif /* ! codereview */

5146         if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
5147                 fp->f_offset = off;
5148         releasef(fd);
5149         if (fromsnap != NULL)
5150                 dmu_objset_rele(fromsnap, FTAG);
5151         dmu_objset_rele(tosnap, FTAG);
5152         return (error);
5153 }

5155 /*
5156  * Determine approximately how large a zfs send stream will be -- the number
5157  * of bytes that will be written to the fd supplied to zfs_ioc_send_new().
5158  *
5159  * innvl: {
5160  *     (optional) "fromsnap" -> full snap name to send an incremental from
5161  * }
5162  *
5163  * outnvl: {
5164  *     "space" -> bytes of space (uint64)
5165  * }
5166  */
5167 static int
5168 zfs_ioc_send_space(const char *snapname, nvlist_t *innvl, nvlist_t *outnvl)
5169 {
5170         objset_t *fromsnap = NULL;
5171         objset_t *tosnap;
5172         int error;
5173         char *fromname;
5174         uint64_t space;

5176         error = dmu_objset_hold(snapname, FTAG, &tosnap);
5177         if (error)
5178                 return (error);

5180         error = nvlist_lookup_string(innvl, "fromsnap", &fromname);
5181         if (error == 0) {
5182                 error = dmu_objset_hold(fromname, FTAG, &fromsnap);
5183                 if (error) {
5184                         dmu_objset_rele(tosnap, FTAG);
5185                         return (error);
5186                 }
5187         }

5189         error = dmu_send_estimate(tosnap, fromsnap, &space);
5190         fnvlist_add_uint64(outnvl, "space", space);

5192         if (fromsnap != NULL)
5193                 dmu_objset_rele(fromsnap, FTAG);
5194         dmu_objset_rele(tosnap, FTAG);
5195         return (error);
5196 }


5199 static zfs_ioc_vec_t zfs_ioc_vec[ZFS_IOC_LAST - ZFS_IOC_FIRST];

5201 static void
5202 zfs_ioctl_register_legacy(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
```



```
5203     zfs_secpolicy_func_t *secpolicy, zfs_ioc_namecheck_t namecheck,
5204     boolean_t log_history, zfs_ioc_poolcheck_t pool_check)
5205 {
5206         zfs_ioc_vec_t *vec = &zfs_ioc_vec[ioc - ZFS_IOC_FIRST];

5208         ASSERT3U(ioc, >=, ZFS_IOC_FIRST);
5209         ASSERT3U(ioc, <, ZFS_IOC_LAST);
5210         ASSERT3P(vec->zvec_legacy_func, ==, NULL);
5211         ASSERT3P(vec->zvec_func, ==, NULL);

5213         vec->zvec_legacy_func = func;
5214         vec->zvec_secpolicy = secpolicy;
5215         vec->zvec_namecheck = namecheck;
5216         vec->zvec_allow_log = log_history;
5217         vec->zvec_pool_check = pool_check;
5218 }

5220 /*
5221  * See the block comment at the beginning of this file for details on
5222  * each argument to this function.
5223  */
5224 static void
5225 zfs_ioctl_register(const char *name, zfs_ioc_t ioc, zfs_ioc_func_t *func,
5226     zfs_secpolicy_func_t *secpolicy, zfs_ioc_namecheck_t namecheck,
5227     zfs_ioc_poolcheck_t pool_check, boolean_t smush_outnvlist,
5228     boolean_t allow_log)
5229 {
5230         zfs_ioc_vec_t *vec = &zfs_ioc_vec[ioc - ZFS_IOC_FIRST];

5232         ASSERT3U(ioc, >=, ZFS_IOC_FIRST);
5233         ASSERT3U(ioc, <, ZFS_IOC_LAST);
5234         ASSERT3P(vec->zvec_legacy_func, ==, NULL);
5235         ASSERT3P(vec->zvec_func, ==, NULL);

5237         /* if we are logging, the name must be valid */
5238         ASSERT(!allow_log || namecheck != NO_NAME);

5240         vec->zvec_name = name;
5241         vec->zvec_func = func;
5242         vec->zvec_secpolicy = secpolicy;
5243         vec->zvec_namecheck = namecheck;
5244         vec->zvec_pool_check = pool_check;
5245         vec->zvec_smush_outnvlist = smush_outnvlist;
5246         vec->zvec_allow_log = allow_log;
5247 }

5249 static void
5250 zfs_ioctl_register_pool(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5251     zfs_secpolicy_func_t *secpolicy, boolean_t log_history,
5252     zfs_ioc_poolcheck_t pool_check)
5253 {
5254         zfs_ioctl_register_legacy(ioc, func, secpolicy,
5255             POOL_NAME, log_history, pool_check);
5256 }

5258 static void
5259 zfs_ioctl_register_dataset_nolog(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5260     zfs_secpolicy_func_t *secpolicy, zfs_ioc_poolcheck_t pool_check)
5261 {
5262         zfs_ioctl_register_legacy(ioc, func, secpolicy,
5263             DATASET_NAME, B_FALSE, pool_check);
5264 }

5266 static void
5267 zfs_ioctl_register_pool_modify(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func)
5268 {
```

```
5269         zfs_ioctl_register_legacy(ioc, func, zfs_secpolicy_config,
5270             POOL_NAME, B_TRUE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5271 }

5273 static void
5274 zfs_ioctl_register_pool_meta(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5275     zfs_secpolicy_func_t *secpolicy)
5276 {
5277         zfs_ioctl_register_legacy(ioc, func, secpolicy,
5278             NO_NAME, B_FALSE, POOL_CHECK_NONE);
5279 }

5281 static void
5282 zfs_ioctl_register_dataset_read_secpolicy(zfs_ioc_t ioc,
5283     zfs_ioc_legacy_func_t *func, zfs_secpolicy_func_t *secpolicy)
5284 {
5285         zfs_ioctl_register_legacy(ioc, func, secpolicy,
5286             DATASET_NAME, B_FALSE, POOL_CHECK_SUSPENDED);
5287 }

5289 static void
5290 zfs_ioctl_register_dataset_read(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func)
5291 {
5292         zfs_ioctl_register_dataset_read_secpolicy(ioc, func,
5293             zfs_secpolicy_read);
5294 }

5296 static void
5297 zfs_ioctl_register_dataset_modify(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5298         zfs_secpolicy_func_t *secpolicy)
5299 {
5300         zfs_ioctl_register_legacy(ioc, func, secpolicy,
5301             DATASET_NAME, B_TRUE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5302 }

5304 static void
5305 zfs_ioctl_init(void)
5306 {
5307         zfs_ioctl_register("snapshot", ZFS_IOC_SNAPSHOT,
5308             zfs_ioc_snapshot, zfs_secpolicy_snapshot, POOL_NAME,
5309             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5311         zfs_ioctl_register("log_history", ZFS_IOC_LOG_HISTORY,
5312             zfs_ioc_log_history, zfs_secpolicy_log_history, NO_NAME,
5313             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_FALSE, B_FALSE);

5315         zfs_ioctl_register("space_snaps", ZFS_IOC_SPACE_SNAPS,
5316             zfs_ioc_space_snaps, zfs_secpolicy_read, DATASET_NAME,
5317             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5319         zfs_ioctl_register("send", ZFS_IOC_SEND_NEW,
5320             zfs_ioc_send_new, zfs_secpolicy_send_new, DATASET_NAME,
5321             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5323         zfs_ioctl_register("send_space", ZFS_IOC_SEND_SPACE,
5324             zfs_ioc_send_space, zfs_secpolicy_read, DATASET_NAME,
5325             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5327         zfs_ioctl_register("create", ZFS_IOC_CREATE,
5328             zfs_ioc_create, zfs_secpolicy_create_clone, DATASET_NAME,
5329             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5331         zfs_ioctl_register("clone", ZFS_IOC_CLONE,
5332             zfs_ioc_clone, zfs_secpolicy_create_clone, DATASET_NAME,
5333             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);
```



```
5335         zfs_ioctl_register("destroy_snaps", ZFS_IOC_DESTROY_SNAPS,
5336             zfs_ioc_destroy_snaps, zfs_secpolicy_destroy_snaps, POOL_NAME,
5337             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5339         /* IOCTLS that use the legacy function signature */

5341         zfs_ioctl_register_legacy(ZFS_IOC_POOL_FREEZE, zfs_ioc_pool_freeze,
5342             zfs_secpolicy_config, NO_NAME, B_FALSE, POOL_CHECK_READONLY);

5344         zfs_ioctl_register_pool(ZFS_IOC_POOL_CREATE, zfs_ioc_pool_create,
5345             zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);
5346         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SCAN,
5347             zfs_ioc_pool_scan);
5348         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_UPGRADE,
5349             zfs_ioc_pool_upgrade);
5350         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ADD,
5351             zfs_ioc_vdev_add);
5352         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_REMOVE,
5353             zfs_ioc_vdev_remove);
5354         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SET_STATE,
5355             zfs_ioc_vdev_set_state);
5356         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ATTACH,
5357             zfs_ioc_vdev_attach);
5358         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_DETACH,
5359             zfs_ioc_vdev_detach);
5360         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETPATH,
5361             zfs_ioc_vdev_setpath);
5362         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETFRU,
5363             zfs_ioc_vdev_setfru);
5364         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SET_PROPS,
5365             zfs_ioc_pool_set_props);
5366         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SPLIT,
5367             zfs_ioc_vdev_split);
5368         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_REGUID,
5369             zfs_ioc_pool_reguid);

5371         zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_CONFIGS,
5372             zfs_ioc_pool_configs, zfs_secpolicy_none);
5373         zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_TRYIMPORT,
5374             zfs_ioc_pool_tryimport, zfs_secpolicy_config);
5375         zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_FAULT,
5376             zfs_ioc_inject_fault, zfs_secpolicy_inject);
5377         zfs_ioctl_register_pool_meta(ZFS_IOC_CLEAR_FAULT,
5378             zfs_ioc_clear_fault, zfs_secpolicy_inject);
5379         zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_LIST_NEXT,
5380             zfs_ioc_inject_list_next, zfs_secpolicy_inject);

5382         /*
5383          * pool destroy, and export don't log the history as part of
5384          * zfsdev_ioctl, but rather zfs_ioc_pool_export
5385          * does the logging of those commands.
5386          */
5387         zfs_ioctl_register_pool(ZFS_IOC_POOL_DESTROY, zfs_ioc_pool_destroy,
5388             zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);
5389         zfs_ioctl_register_pool(ZFS_IOC_POOL_EXPORT, zfs_ioc_pool_export,
5390             zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);

5392         zfs_ioctl_register_pool(ZFS_IOC_POOL_STATS, zfs_ioc_pool_stats,
5393             zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);
5394         zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_PROPS, zfs_ioc_pool_get_props,
5395             zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);

5397         zfs_ioctl_register_pool(ZFS_IOC_ERROR_LOG, zfs_ioc_error_log,
5398             zfs_secpolicy_inject, B_FALSE, POOL_CHECK_SUSPENDED);
5399         zfs_ioctl_register_pool(ZFS_IOC_DSOBJ_TO_DSNAME,
5400             zfs_ioc_dsobj_to_dsname,
```

```
5401             zfs_secpolicy_diff, B_FALSE, POOL_CHECK_SUSPENDED);
5402         zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_HISTORY,
5403             zfs_ioc_pool_get_history,
5404             zfs_secpolicy_config, B_FALSE, POOL_CHECK_SUSPENDED);
5405
5406         zfs_ioctl_register_pool(ZFS_IOC_POOL_IMPORT, zfs_ioc_pool_import,
5407             zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);
5408
5409         zfs_ioctl_register_pool(ZFS_IOC_CLEAR, zfs_ioc_clear,
5410             zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);
5411         zfs_ioctl_register_pool(ZFS_IOC_POOL_REOPEN, zfs_ioc_pool_reopen,
5412             zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);
5413
5414         zfs_ioctl_register_dataset_read(ZFS_IOC_SPACE_WRITTEN,
5415             zfs_ioc_space_written);
5416         zfs_ioctl_register_dataset_read(ZFS_IOC_GET_HOLDS,
5417             zfs_ioc_get_holds);
5418         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_RECVD_PROPS,
5419             zfs_ioc_objset_recvd_props);
5420         zfs_ioctl_register_dataset_read(ZFS_IOC_NEXT_OBJ,
5421             zfs_ioc_next_obj);
5422         zfs_ioctl_register_dataset_read(ZFS_IOC_GET_FSACL,
5423             zfs_ioc_get_fsacl);
5424         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_STATS,
5425             zfs_ioc_objset_stats);
5426         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_ZPLPROPS,
5427             zfs_ioc_objset_zplprops);
5428         zfs_ioctl_register_dataset_read(ZFS_IOC_DATASET_LIST_NEXT,
5429             zfs_ioc_dataset_list_next);
5430         zfs_ioctl_register_dataset_read(ZFS_IOC_SNAPSHOT_LIST_NEXT,
5431             zfs_ioc_snapshot_list_next);
5432         zfs_ioctl_register_dataset_read(ZFS_IOC_SEND_PROGRESS,
5433             zfs_ioc_send_progress);
5434
5435         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_DIFF,
5436             zfs_ioc_diff, zfs_secpolicy_diff);
5437         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_STATS,
5438             zfs_ioc_obj_to_stats, zfs_secpolicy_diff);
5439         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_PATH,
5440             zfs_ioc_obj_to_path, zfs_secpolicy_diff);
5441         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_ONE,
5442             zfs_ioc_userspace_one, zfs_secpolicy_userspace_one);
5443         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_MANY,
5444             zfs_ioc_userspace_many, zfs_secpolicy_userspace_many);
5445         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_SEND,
5446             zfs_ioc_send, zfs_secpolicy_send);
5447
5448         zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_PROP, zfs_ioc_set_prop,
5449             zfs_secpolicy_none);
5450         zfs_ioctl_register_dataset_modify(ZFS_IOC_DESTROY, zfs_ioc_destroy,
5451             zfs_secpolicy_destroy);
5452         zfs_ioctl_register_dataset_modify(ZFS_IOC_ROLLBACK, zfs_ioc_rollback,
5453             zfs_secpolicy_rollback);
5454         zfs_ioctl_register_dataset_modify(ZFS_IOC_RENAME, zfs_ioc_rename,
5455             zfs_secpolicy_rename);
5456         zfs_ioctl_register_dataset_modify(ZFS_IOC_RECV, zfs_ioc_recv,
5457             zfs_secpolicy_recv);
5458         zfs_ioctl_register_dataset_modify(ZFS_IOC_PROMOTE, zfs_ioc_promote,
5459             zfs_secpolicy_promote);
5460         zfs_ioctl_register_dataset_modify(ZFS_IOC_HOLD, zfs_ioc_hold,
5461             zfs_secpolicy_hold);
5462         zfs_ioctl_register_dataset_modify(ZFS_IOC_RELEASE, zfs_ioc_release,
5463             zfs_secpolicy_release);
5464         zfs_ioctl_register_dataset_modify(ZFS_IOC_INHERIT_PROP,
5465             zfs_ioc_inherit_prop, zfs_secpolicy_inherit_prop);
5466         zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_FSACL, zfs_ioc_set_fsacl,
```



```
5467             zfs_secpolicy_set_fsacl);
5468
5469         zfs_ioctl_register_dataset_nolog(ZFS_IOC_SHARE, zfs_ioc_share,
5470             zfs_secpolicy_share, POOL_CHECK_NONE);
5471         zfs_ioctl_register_dataset_nolog(ZFS_IOC_SMB_ACL, zfs_ioc_smb_acl,
5472             zfs_secpolicy_smb_acl, POOL_CHECK_NONE);
5473         zfs_ioctl_register_dataset_nolog(ZFS_IOC_USERSPACE_UPGRADE,
5474             zfs_ioc_userspace_upgrade, zfs_secpolicy_userspace_upgrade,
5475             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5476         zfs_ioctl_register_dataset_nolog(ZFS_IOC_TMP_SNAPSHOT,
5477             zfs_ioc_tmp_snapshot, zfs_secpolicy_tmp_snapshot,
5478             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5479 }
5480
5481 int
5482 pool_status_check(const char *name, zfs_ioc_namecheck_t type,
5483     zfs_ioc_poolcheck_t check)
5484 {
5485         spa_t *spa;
5486         int error;
5487
5488         ASSERT(type == POOL_NAME || type == DATASET_NAME);
5489
5490         if (check & POOL_CHECK_NONE)
5491                 return (0);
5492
5493         error = spa_open(name, &spa, FTAG);
5494         if (error == 0) {
5495                 if ((check & POOL_CHECK_SUSPENDED) && spa_suspended(spa))
5496                         error = EAGAIN;
5497                 else if ((check & POOL_CHECK_READONLY) && !spa_writeable(spa))
5498                         error = EROFS;
5499                 spa_close(spa, FTAG);
5500         }
5501         return (error);
5502 }
5503
5504 /*
5505  * Find a free minor number.
5506  */
5507 minor_t
5508 zfsdev_minor_alloc(void)
5509 {
5510         static minor_t last_minor;
5511         minor_t m;
5512
5513         ASSERT(MUTEX_HELD(&zfsdev_state_lock));
5514
5515         for (m = last_minor + 1; m != last_minor; m++) {
5516                 if (m > ZFSDEV_MAX_MINOR)
5517                         m = 1;
5518                 if (ddi_get_soft_state(zfsdev_state, m) == NULL) {
5519                         last_minor = m;
5520                         return (m);
5521                 }
5522         }
5523
5524         return (0);
5525 }
5526
5527 static int
5528 zfs_ctldev_init(dev_t *devp)
5529 {
5530         minor_t minor;
5531         zfs_soft_state_t *zs;
```

```
5533         ASSERT(MUTEX_HELD(&zfsdev_state_lock));
5534         ASSERT(getminor(*devp) == 0);

5536         minor = zfsdev_minor_alloc();
5537         if (minor == 0)
5538                 return (ENXIO);

5540         if (ddi_soft_state_zalloc(zfsdev_state, minor) != DDI_SUCCESS)
5541                 return (EAGAIN);

5543         *devp = makedevice(getemajor(*devp), minor);

5545         zs = ddi_get_soft_state(zfsdev_state, minor);
5546         zs->zss_type = ZSST_CTLDEV;
5547         zfs_onexit_init((zfs_onexit_t **)&zs->zss_data);

5549         return (0);
5550 }

5552 static void
5553 zfs_ctldev_destroy(zfs_onexit_t *zo, minor_t minor)
5554 {
5555         ASSERT(MUTEX_HELD(&zfsdev_state_lock));

5557         zfs_onexit_destroy(zo);
5558         ddi_soft_state_free(zfsdev_state, minor);
5559 }

5561 void *
5562 zfsdev_get_soft_state(minor_t minor, enum zfs_soft_state_type which)
5563 {
5564         zfs_soft_state_t *zp;

5566         zp = ddi_get_soft_state(zfsdev_state, minor);
5567         if (zp == NULL || zp->zss_type != which)
5568                 return (NULL);

5570         return (zp->zss_data);
5571 }

5573 static int
5574 zfsdev_open(dev_t *devp, int flag, int otyp, cred_t *cr)
5575 {
5576         int error = 0;

5578         if (getminor(*devp) != 0)
5579                 return (zvol_open(devp, flag, otyp, cr));

5581         /* This is the control device. Allocate a new minor if requested. */
5582         if (flag & FEXCL) {
5583                 mutex_enter(&zfsdev_state_lock);
5584                 error = zfs_ctldev_init(devp);
5585                 mutex_exit(&zfsdev_state_lock);
5586         }

5588         return (error);
5589 }

5591 static int
5592 zfsdev_close(dev_t dev, int flag, int otyp, cred_t *cr)
5593 {
5594         zfs_onexit_t *zo;
5595         minor_t minor = getminor(dev);

5597         if (minor == 0)
5598                 return (0);
```



```
5600         mutex_enter(&zfsdev_state_lock);
5601         zo = zfsdev_get_soft_state(minor, ZSST_CTLDEV);
5602         if (zo == NULL) {
5603                 mutex_exit(&zfsdev_state_lock);
5604                 return (zvol_close(dev, flag, otyp, cr));
5605         }
5606         zfs_ctldev_destroy(zo, minor);
5607         mutex_exit(&zfsdev_state_lock);

5609         return (0);
5610 }

5612 static int
5613 zfsdev_ioctl(dev_t dev, int cmd, intptr_t arg, int flag, cred_t *cr, int *rvalp)
5614 {
5615         zfs_cmd_t *zc;
5616         uint_t vecnum;
5617         int error, rc, len;
5618         minor_t minor = getminor(dev);
5619         const zfs_ioc_vec_t *vec;
5620         char *saved_poolname = NULL;
5621         nvlist_t *innvl = NULL;

5623         if (minor != 0 &&
5624             zfsdev_get_soft_state(minor, ZSST_CTLDEV) == NULL)
5625                 return (zvol_ioctl(dev, cmd, arg, flag, cr, rvalp));

5627         vecnum = cmd - ZFS_IOC_FIRST;
5628         ASSERT3U(getmajor(dev), ==, ddi_driver_major(zfs_dip));

5630         if (vecnum >= sizeof (zfs_ioc_vec) / sizeof (zfs_ioc_vec[0]))
5631                 return (EINVAL);
5632         vec = &zfs_ioc_vec[vecnum];

5634         zc = kmem_zalloc(sizeof (zfs_cmd_t), KM_SLEEP);

5636         error = ddi_copyin((void *)arg, zc, sizeof (zfs_cmd_t), flag);
5637         if (error != 0) {
5638                 error = EFAULT;
5639                 goto out;
5640         }

5642         zc->zc_iflags = flag & FKIOCTL;
5643         if (zc->zc_nvlist_src_size != 0) {
5644                 error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
5645                     zc->zc_iflags, &innvl);
5646                 if (error != 0)
5647                         goto out;
5648         }

5650         /*
5651          * Ensure that all pool/dataset names are valid before we pass down to
5652          * the lower layers.
5653          */
5654         zc->zc_name[sizeof (zc->zc_name) - 1] = '\0';
5655         switch (vec->zvec_namecheck) {
5656         case POOL_NAME:
5657                 if (pool_namecheck(zc->zc_name, NULL, NULL) != 0)
5658                         error = EINVAL;
5659                 else
5660                         error = pool_status_check(zc->zc_name,
5661                             vec->zvec_namecheck, vec->zvec_pool_check);
5662                 break;

5664         case DATASET_NAME:
```

```
5665                 if (dataset_namecheck(zc->zc_name, NULL, NULL) != 0)
5666                         error = EINVAL;
5667                 else
5668                         error = pool_status_check(zc->zc_name,
5669                             vec->zvec_namecheck, vec->zvec_pool_check);
5670                 break;

5672         case NO_NAME:
5673                 break;
5674         }


5677         if (error == 0 && !(flag & FKIOCTL))
5678                 error = vec->zvec_secpolicy(zc, innvl, cr);

5680         if (error != 0)
5681                 goto out;

5683         /* legacy ioctls can modify zc_name */
5684         len = strcspn(zc->zc_name, "/@") + 1;
5685         saved_poolname = kmem_alloc(len, KM_SLEEP);
5686         (void) strlcpy(saved_poolname, zc->zc_name, len);

5688         if (vec->zvec_func != NULL) {
5689                 nvlist_t *outnvl;
5690                 int puterror = 0;
5691                 spa_t *spa;
5692                 nvlist_t *lognv = NULL;

5694                 ASSERT(vec->zvec_legacy_func == NULL);

5696                 /*
5697                  * Add the innvl to the lognv before calling the func,
5698                  * in case the func changes the innvl.
5699                  */
5700                 if (vec->zvec_allow_log) {
5701                         lognv = fnvlist_alloc();
5702                         fnvlist_add_string(lognv, ZPOOL_HIST_IOCTL,
5703                             vec->zvec_name);
5704                         if (!nvlist_empty(innvl)) {
5705                                 fnvlist_add_nvlist(lognv, ZPOOL_HIST_INPUT_NVL,
5706                                     innvl);
5707                         }
5708                 }

5710                 outnvl = fnvlist_alloc();
5711                 error = vec->zvec_func(zc->zc_name, innvl, outnvl);

5713                 if (error == 0 && vec->zvec_allow_log &&
5714                     spa_open(zc->zc_name, &spa, FTAG) == 0) {
5715                         if (!nvlist_empty(outnvl)) {
5716                                 fnvlist_add_nvlist(lognv, ZPOOL_HIST_OUTPUT_NVL,
5717                                     outnvl);
5718                         }
5719                         (void) spa_history_log_nvl(spa, lognv);
5720                         spa_close(spa, FTAG);
5721                 }
5722                 fnvlist_free(lognv);

5724                 if (!nvlist_empty(outnvl) || zc->zc_nvlist_dst_size != 0) {
5725                         int smusherror = 0;
5726                         if (vec->zvec_smush_outnvlist) {
5727                                 smusherror = nvlist_smush(outnvl,
5728                                     zc->zc_nvlist_dst_size);
5729                         }
5730                         if (smusherror == 0)
```



```
5731                                 puterror = put_nvlist(zc, outnvl);
5732                 }

5734                 if (puterror != 0)
5735                         error = puterror;

5737                 nvlist_free(outnvl);
5738         } else {
5739                 error = vec->zvec_legacy_func(zc);
5740         }

5742 out:
5743         nvlist_free(innvl);
5744         rc = ddi_copyout(zc, (void *)arg, sizeof (zfs_cmd_t), flag);
5745         if (error == 0 && rc != 0)
5746                 error = EFAULT;
5747         if (error == 0 && vec->zvec_allow_log) {
5748                 char *s = tsd_get(zfs_allow_log_key);
5749                 if (s != NULL)
5750                         strfree(s);
5751                 (void) tsd_set(zfs_allow_log_key, saved_poolname);
5752         } else {
5753                 if (saved_poolname != NULL)
5754                         strfree(saved_poolname);
5755         }

5757         kmem_free(zc, sizeof (zfs_cmd_t));
5758         return (error);
5759 }

5761 static int
5762 zfs_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
5763 {
5764         if (cmd != DDI_ATTACH)
5765                 return (DDI_FAILURE);

5767         if (ddi_create_minor_node(dip, "zfs", S_IFCHR, 0,
5768             DDI_PSEUDO, 0) == DDI_FAILURE)
5769                 return (DDI_FAILURE);

5771         zfs_dip = dip;

5773         ddi_report_dev(dip);

5775         return (DDI_SUCCESS);
5776 }

5778 static int
5779 zfs_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
5780 {
5781         if (spa_busy() || zfs_busy() || zvol_busy())
5782                 return (DDI_FAILURE);

5784         if (cmd != DDI_DETACH)
5785                 return (DDI_FAILURE);

5787         zfs_dip = NULL;

5789         ddi_prop_remove_all(dip);
5790         ddi_remove_minor_node(dip, NULL);

5792         return (DDI_SUCCESS);
5793 }

5795 /*ARGSUSED*/
5796 static int
```

```
5797 zfs_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
5798 {
5799         switch (infocmd) {
5800         case DDI_INFO_DEVT2DEVINFO:
5801                 *result = zfs_dip;
5802                 return (DDI_SUCCESS);

5804         case DDI_INFO_DEVT2INSTANCE:
5805                 *result = (void *)0;
5806                 return (DDI_SUCCESS);
5807         }

5809         return (DDI_FAILURE);
5810 }

5812 /*
5813  * OK, so this is a little weird.
5814  *
5815  * /dev/zfs is the control node, i.e. minor 0.
5816  * /dev/zvol/[r]dsk/pool/dataset are the zvols, minor > 0.
5817  *
5818  * /dev/zfs has basically nothing to do except serve up ioctls,
5819  * so most of the standard driver entry points are in zvol.c.
5820  */
5821 static struct cb_ops zfs_cb_ops = {
5822         zfsdev_open,    /* open */
5823         zfsdev_close,   /* close */
5824         zvol_strategy,  /* strategy */
5825         nodev,          /* print */
5826         zvol_dump,      /* dump */
5827         zvol_read,      /* read */
5828         zvol_write,     /* write */
5829         zfsdev_ioctl,   /* ioctl */
5830         nodev,          /* devmap */
5831         nodev,          /* mmap */
5832         nodev,          /* segmap */
5833         nochpoll,       /* poll */
5834         ddi_prop_op,    /* prop_op */
5835         NULL,           /* streamtab */
5836         D_NEW | D_MP | D_64BIT,         /* Driver compatibility flag */
5837         CB_REV,         /* version */
5838         nodev,          /* async read */
5839         nodev,          /* async write */
5840 };

5842 static struct dev_ops zfs_dev_ops = {
5843         DEVO_REV,       /* version */
5844         0,              /* refcnt */
5845         zfs_info,       /* info */
5846         nulldev,        /* identify */
5847         nulldev,        /* probe */
5848         zfs_attach,     /* attach */
5849         zfs_detach,     /* detach */
5850         nodev,          /* reset */
5851         &zfs_cb_ops,    /* driver operations */
5852         NULL,           /* no bus operations */
5853         NULL,           /* power */
5854         ddi_quiesce_not_needed, /* quiesce */
5855 };

5857 static struct modldrv zfs_modldrv = {
5858         &mod_driverops,
5859         "ZFS storage pool",
5860         &zfs_dev_ops
5861 };
```



```
5863 static struct modlinkage modlinkage = {
5864         MODREV_1,
5865         (void *)&zfs_modlfs,
5866         (void *)&zfs_modldrv,
5867         NULL
5868 };

5870 static void
5871 zfs_allow_log_destroy(void *arg)
5872 {
5873         char *poolname = arg;
5874         strfree(poolname);
5875 }

5877 int
5878 _init(void)
5879 {
5880         int error;

5882         spa_init(FREAD | FWRITE);
5883         zfs_init();
5884         zvol_init();
5885         zfs_ioctl_init();

5887         if ((error = mod_install(&modlinkage)) != 0) {
5888                 zvol_fini();
5889                 zfs_fini();
5890                 spa_fini();
5891                 return (error);
5892         }

5894         tsd_create(&zfs_fsyncer_key, NULL);
5895         tsd_create(&rrw_tsd_key, rrw_tsd_destroy);
5896         tsd_create(&zfs_allow_log_key, zfs_allow_log_destroy);

5898         error = ldi_ident_from_mod(&modlinkage, &zfs_li);
5899         ASSERT(error == 0);
5900         mutex_init(&zfs_share_lock, NULL, MUTEX_DEFAULT, NULL);

5902         return (0);
5903 }

5905 int
5906 _fini(void)
5907 {
5908         int error;

5910         if (spa_busy() || zfs_busy() || zvol_busy() || zio_injection_enabled)
5911                 return (EBUSY);

5913         if ((error = mod_remove(&modlinkage)) != 0)
5914                 return (error);

5916         zvol_fini();
5917         zfs_fini();
5918         spa_fini();
5919         if (zfs_nfsshare_inited)
5920                 (void) ddi_modclose(nfs_mod);
5921         if (zfs_smbshare_inited)
5922                 (void) ddi_modclose(smbsrv_mod);
5923         if (zfs_nfsshare_inited || zfs_smbshare_inited)
5924                 (void) ddi_modclose(sharefs_mod);

5926         tsd_destroy(&zfs_fsyncer_key);
5927         ldi_ident_release(zfs_li);
5928         zfs_li = NULL;
```

```
5929         mutex_destroy(&zfs_share_lock);

5931         return (error);
5932 }

5934 int
5935 _info(struct modinfo *modinfop)
5936 {
5937         return (mod_info(&modlinkage, modinfop));
5938 }
```