

new/usr/src/cmd/ndmpd/ndmp/ndmpd_chkpt.c

1

```
*****
8478 Fri May 24 00:51:01 2013
new/usr/src/cmd/ndmpd/ndmp/ndmpd_chkpt.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
_____unchanged_portion_omitted_____

184 /*
185  * Put a hold on snapshot
186  */
187 int
188 snapshot_hold(char *volname, char *snapname, char *jname, boolean_t recursive)
189 {
190     zfs_handle_t *zhp;
191     char *p;

193     if ((zhp = zfs_open(zlibh, volname, ZFS_TYPE_DATASET)) == 0) {
194         NDMP_LOG(LOG_ERR, "Cannot open volume %s.", volname);
195         return (-1);
196     }

198     if (cleanup_fd == -1 && (cleanup_fd = open(ZFS_DEV,
199         O_RDWR|O_EXCL)) < 0) {
200         NDMP_LOG(LOG_ERR, "Cannot open dev %d", errno);
201         zfs_close(zhp);
202         return (-1);
203     }

205     p = strchr(snapname, '@') + 1;
206     if (zfs_hold(zhp, p, jname, recursive, cleanup_fd) != 0) {
206     if (zfs_hold(zhp, p, jname, recursive, B_FALSE, cleanup_fd) != 0) {
207         NDMP_LOG(LOG_ERR, "Cannot hold snapshot %s", p);
208         zfs_close(zhp);
209         return (-1);
210     }
211     zfs_close(zhp);
212     return (0);
213 }
_____unchanged_portion_omitted_____
```

new/usr/src/cmd/zfs/zfs_main.c

1

```
*****
161499 Fri May 24 00:51:01 2013
new/usr/src/cmd/zfs/zfs_main.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
unchanged_portion_omitted

5104 static int
5105 zfs_do_hold_rele_impl(int argc, char **argv, boolean_t holding)
5106 {
5107     int errors = 0;
5108     int i;
5109     const char *tag;
5110     boolean_t recursive = B_FALSE;
5111     const char *opts = holding ? "rt" : "r";
5112     int c;

5114     /* check options */
5115     while ((c = getopt(argc, argv, opts)) != -1) {
5116         switch (c) {
5117             case 'r':
5118                 recursive = B_TRUE;
5119                 break;
5120             case '?':
5121                 (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5122                     optopt);
5123                 usage(B_FALSE);
5124             }
5125     }

5127     argc -= optind;
5128     argv += optind;

5130     /* check number of arguments */
5131     if (argc < 2)
5132         usage(B_FALSE);

5134     tag = argv[0];
5135     --argc;
5136     ++argv;

5138     if (holding && tag[0] == '.') {
5139         /* tags starting with '.' are reserved for libzfs */
5140         (void) fprintf(stderr, gettext("tag may not start with '.'\n"));
5141         usage(B_FALSE);
5142     }

5144     for (i = 0; i < argc; ++i) {
5145         zfs_handle_t *zhp;
5146         char parent[ZFS_MAXNAMELEN];
5147         const char *delim;
5148         char *path = argv[i];

5150         delim = strchr(path, '@');
5151         if (delim == NULL) {
5152             (void) fprintf(stderr,
5153                 gettext("%s' is not a snapshot\n"), path);
5154             ++errors;
5155             continue;
5156         }
5157         (void) strncpy(parent, path, delim - path);
5158         parent[delim - path] = '\0';

5160         zhp = zfs_open(g_zfs, parent,
5161             ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);

```

new/usr/src/cmd/zfs/zfs_main.c

2

```
5162         if (zhp == NULL) {
5163             ++errors;
5164             continue;
5165         }
5166         if (holding) {
5167             if (zfs_hold(zhp, delim+1, tag, recursive, -1) != 0)
5167                 if (zfs_hold(zhp, delim+1, tag, recursive,
5168                     B_FALSE, -1) != 0)
5168                     ++errors;
5169         } else {
5170             if (zfs_release(zhp, delim+1, tag, recursive) != 0)
5171                 ++errors;
5172         }
5173         zfs_close(zhp);
5174     }

5176     return (errors != 0);
5177 }
unchanged_portion_omitted

```

```
*****
26918 Fri May 24 00:51:01 2013
new/usr/src/lib/libzfs/common/libzfs.h
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
unchanged_portion_omitted

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

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

595 extern int zfs_promote(zfs_handle_t *);
596 extern int zfs_hold(zfs_handle_t *, const char *, const char *,
597     boolean_t, int);
598 extern int zfs_hold_nvlist(zfs_handle_t *, int, nvlist_t *);
599 extern int zfs_release(zfs_handle_t *, const char *, const char *, boolean_t);
600 extern int zfs_get_holds(zfs_handle_t *, nvlist_t **);
601 extern uint64_t zvol_volsize_to_reservation(uint64_t, nvlist_t *);

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

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

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

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

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

619     /*
620      * Only the tail of the sent snapshot path is appended to the
621      * destination to determine the received snapshot name (ie, -e).
622      */
623     boolean_t istail;

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

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

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

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

637     /* do not mount file systems as they are extracted (private) */
638     boolean_t nomount;
639 } recvflags_t;
unchanged_portion_omitted
```

```

*****
110822 Fri May 24 00:51:02 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
unchanged_portion_omitted_

4080 static int
4081 zfs_hold_one(zfs_handle_t *zhp, void *arg)
4082 {
4083     struct holdarg *ha = arg;
4084     zfs_handle_t *szhp;
4084     char name[ZFS_MAXNAMELEN];
4085     int rv = 0;

4087     (void) snprintf(name, sizeof (name),
4088         "%s@%s", zhp->zfs_name, ha->snapname);

4090     if (lzc_exists(name))
4091         szhp = make_dataset_handle(zhp->zfs_hdl, name);
4092     if (szhp) {
4091         fnvlist_add_string(ha->nvl, name, ha->tag);
4094         zfs_close(szhp);
4095     }

4093     if (ha->recursive)
4094         rv = zfs_iter_filesystems(zhp, zfs_hold_one, ha);
4095     zfs_close(zhp);
4096     return (rv);
4097 }

4099 int
4100 zfs_hold(zfs_handle_t *zhp, const char *snapname, const char *tag,
4101     boolean_t recursive, int cleanup_fd)
4102     boolean_t recursive, boolean_t enoent_ok, int cleanup_fd)
4103 {
4103     int ret;
4104     struct holdarg ha;
4109     nvlist_t *errors;
4110     libzfs_handle_t *hdl = zhp->zfs_hdl;
4111     char errbuf[1024];
4112     nvpair_t *elem;

4106     ha.nvl = fnvlist_alloc();
4107     ha.snapname = snapname;
4108     ha.tag = tag;
4109     ha.recursive = recursive;
4110     (void) zfs_hold_one(zfs_handle_dup(zhp), &ha);
4111     ret = zfs_hold_nvl(zhp, cleanup_fd, ha.nvl);
4119     ret = lzc_hold(ha.nvl, cleanup_fd, &errors);
4112     fnvlist_free(ha.nvl);

4114     return (ret);
4115 }

4117 int
4118 zfs_hold_nvl(zfs_handle_t *zhp, int cleanup_fd, nvlist_t *holds)
4119 {
4120     int ret;
4121     nvlist_t *errors;
4122     libzfs_handle_t *hdl = zhp->zfs_hdl;
4123     char errbuf[1024];
4124     nvpair_t *elem;
4125
4126     errors = NULL;

```

```

4127     ret = lzc_hold(holds, cleanup_fd, &errors);
4128
4129     if (ret == 0) {
4130         /* There may be errors even in the success case. */
4131         fnvlist_free(errors);
4122     if (ret == 0)
4132         return (0);
4133     }
4134 #endif /* ! codereview */

4136     if (nvlist_next_nvpair(errors, NULL) == NULL) {
4137         /* no hold-specific errors */
4138         (void) snprintf(errbuf, sizeof (errbuf),
4139             dgettext(TEXT_DOMAIN, "cannot hold"));
4140         switch (ret) {
4141             case ENOTSUP:
4142                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4143                     "pool must be upgraded"));
4144                 (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4145                 break;
4146             case EINVAL:
4147                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4148                 break;
4149             default:
4150                 (void) zfs_standard_error(hdl, ret, errbuf);
4151         }
4152     }

4154     for (elem = nvlist_next_nvpair(errors, NULL);
4155         elem != NULL;
4156         elem = nvlist_next_nvpair(errors, elem)) {
4157         (void) snprintf(errbuf, sizeof (errbuf),
4158             dgettext(TEXT_DOMAIN,
4159                 "cannot hold snapshot '%s'", nvpair_name(elem)));
4160         switch (fnvpair_value_int32(elem)) {
4161             case E2BIG:
4162                 /*
4163                  * Temporary tags wind up having the ds object id
4164                  * prepended. So even if we passed the length check
4165                  * above, it's still possible for the tag to wind
4166                  * up being slightly too long.
4167                  */
4168                 (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
4169                 break;
4170             case EINVAL:
4171                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4172                 break;
4173             case EEXIST:
4174                 (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
4175                 break;
4176             case ENOENT:
4177                 if (enoent_ok)
4178                     return (ENOENT);
4179                 /* FALLTHROUGH */
4180             default:
4181                 (void) zfs_standard_error(hdl,
4182                     fnvpair_value_int32(elem), errbuf);
4183         }
4184     }

4182     fnvlist_free(errors);
4183     return (ret);
4184 }

4138 struct releasearg {
4139     nvlist_t *nvl;

```

```

4140     const char *snapname;
4141     const char *tag;
4142     boolean_t recursive;
4143 };

4186 static int
4187 zfs_release_one(zfs_handle_t *zhp, void *arg)
4188 {
4189     struct holdarg *ha = arg;
4190     zfs_handle_t *szhp;
4191     char name[ZFS_MAXNAMELEN];
4192     int rv = 0;

4193     (void) snprintf(name, sizeof (name),
4194         "%s@%s", zhp->zfs_name, ha->snapname);

4196     if (lzc_exists(name)) {
4197         szhp = make_dataset_handle(zhp->zfs_hdl, name);
4198         if (szhp) {
4199             nvlist_t *holds = fnvlist_alloc();
4200             fnvlist_add_boolean(holds, ha->tag);
4201             fnvlist_add_nvlist(ha->nvl, name, holds);
4202             fnvlist_free(holds);
4203             zfs_close(szhp);
4204         }
4205     }

4203     if (ha->recursive)
4204         rv = zfs_iter_filesystems(zhp, zfs_release_one, ha);
4205     zfs_close(zhp);
4206     return (rv);
4207 }

4209 int
4210 zfs_release(zfs_handle_t *zhp, const char *snapname, const char *tag,
4211     boolean_t recursive)
4212 {
4213     int ret;
4214     struct holdarg ha;
4215     nvlist_t *errors;
4216     nvpair_t *elem;
4217     libzfs_handle_t *hdl = zhp->zfs_hdl;

4219     ha.nvl = fnvlist_alloc();
4220     ha.snapname = snapname;
4221     ha.tag = tag;
4222     ha.recursive = recursive;
4223     (void) zfs_release_one(zfs_handle_dup(zhp), &ha);
4224     errors = NULL;
4225 #endif /* ! codereview */
4226     ret = lzc_release(ha.nvl, &errors);
4227     fnvlist_free(ha.nvl);

4229     if (ret == 0) {
4230         /* There may be errors even in the success case. */
4231         fnvlist_free(errors);
4232     }
4233     if (ret == 0)
4234         return (0);
4235 #endif /* ! codereview */

4236     if (nvlist_next_nvpair(errors, NULL) == NULL) {
4237         /* no hold-specific errors */
4238         char errbuf[1024];

4240         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
4241             "cannot release"));

```

```

4242     switch (errno) {
4243     case ENOTSUP:
4244         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4245             "pool must be upgraded"));
4246         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4247         break;
4248     default:
4249         (void) zfs_standard_error_fmt(hdl, errno, errbuf);
4250     }
4251 }

4253 for (elem = nvlist_next_nvpair(errors, NULL);
4254     elem != NULL;
4255     elem = nvlist_next_nvpair(errors, elem)) {
4256     char errbuf[1024];

4258     (void) snprintf(errbuf, sizeof (errbuf),
4259         dgettext(TEXT_DOMAIN,
4260             "cannot release hold from snapshot '%s'",
4261             nvpair_name(elem)));
4262     switch (fnvpair_value_int32(elem)) {
4263     case ESRCH:
4264         (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
4265         break;
4266     case EINVAL:
4267         (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4268         break;
4269     default:
4270         (void) zfs_standard_error_fmt(hdl,
4271             fnvpair_value_int32(elem), errbuf);
4272     }
4273 }

4275     fnvlist_free(errors);
4276     return (ret);
4277 }

4279 int
4280 zfs_get_fsacl(zfs_handle_t *zhp, nvlist_t **nvl)
4281 {
4282     zfs_cmd_t zc = { 0 };
4283     libzfs_handle_t *hdl = zhp->zfs_hdl;
4284     int nvsz = 2048;
4285     void *nvbuf;
4286     int err = 0;
4287     char errbuf[1024];

4289     assert(zhp->zfs_type == ZFS_TYPE_VOLUME ||
4290         zhp->zfs_type == ZFS_TYPE_FILESYSTEM);

4292     tryagain:

4294     nvbuf = malloc(nvsz);
4295     if (nvbuf == NULL) {
4296         err = (zfs_error(hdl, EZFS_NOMEM, strerror(errno)));
4297         goto out;
4298     }

4300     zc.zc_nvlist_dst_size = nvsz;
4301     zc.zc_nvlist_dst = (uintptr_t)nvbuf;

4303     (void) strncpy(zc.zc_name, zhp->zfs_name, ZFS_MAXNAMELEN);

4305     if (ioctl(hdl->libzfs_fd, ZFS_IOC_GET_FSACL, &zc) != 0) {
4306         (void) snprintf(errbuf, sizeof (errbuf),
4307             dgettext(TEXT_DOMAIN, "cannot get permissions on '%s'",

```

```

4308         zc.zc_name);
4309     switch (errno) {
4310     case ENOMEM:
4311         free(nvbuf);
4312         nvsz = zc.zc_nvlist_dst_size;
4313         goto tryagain;
4314
4315     case ENOTSUP:
4316         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4317         "pool must be upgraded"));
4318         err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4319         break;
4320     case EINVAL:
4321         err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4322         break;
4323     case ENOENT:
4324         err = zfs_error(hdl, EZFS_NOENT, errbuf);
4325         break;
4326     default:
4327         err = zfs_standard_error_fmt(hdl, errno, errbuf);
4328         break;
4329     }
4330 } else {
4331     /* success */
4332     int rc = nvlist_unpack(nvbuf, zc.zc_nvlist_dst_size, nvl, 0);
4333     if (rc) {
4334         (void) snprintf(errbuf, sizeof (errbuf), dgettext(
4335         TEXT_DOMAIN, "cannot get permissions on '%s'"),
4336         zc.zc_name);
4337         err = zfs_standard_error_fmt(hdl, rc, errbuf);
4338     }
4339 }
4340
4341 free(nvbuf);
4342 out:
4343 return (err);
4344 }
4345
4346 int
4347 zfs_set_fsacl(zfs_handle_t *zhp, boolean_t un, nvlist_t *nvl)
4348 {
4349     zfs_cmd_t zc = { 0 };
4350     libzfs_handle_t *hdl = zhp->zfs_hdl;
4351     char *nvbuf;
4352     char errbuf[1024];
4353     size_t nvsz;
4354     int err;
4355
4356     assert(zhp->zfs_type == ZFS_TYPE_VOLUME ||
4357     zhp->zfs_type == ZFS_TYPE_FILESYSTEM);
4358
4359     err = nvlist_size(nvl, &nvsz, NV_ENCODE_NATIVE);
4360     assert(err == 0);
4361
4362     nvbuf = malloc(nvsz);
4363
4364     err = nvlist_pack(nvl, &nvbuf, &nvsz, NV_ENCODE_NATIVE, 0);
4365     assert(err == 0);
4366
4367     zc.zc_nvlist_src_size = nvsz;
4368     zc.zc_nvlist_src = (uintptr_t)nvbuf;
4369     zc.zc_perm_action = un;
4370
4371     (void) strncpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
4372
4373     if (zfs_ioctl(hdl, ZFS_IOC_SET_FSACL, &zc) != 0) {

```

```

4374         (void) snprintf(errbuf, sizeof (errbuf),
4375         dgettext(TEXT_DOMAIN, "cannot set permissions on '%s'"),
4376         zc.zc_name);
4377         switch (errno) {
4378         case ENOTSUP:
4379             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4380             "pool must be upgraded"));
4381             err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4382             break;
4383         case EINVAL:
4384             err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4385             break;
4386         case ENOENT:
4387             err = zfs_error(hdl, EZFS_NOENT, errbuf);
4388             break;
4389         default:
4390             err = zfs_standard_error_fmt(hdl, errno, errbuf);
4391             break;
4392         }
4393     }
4394
4395     free(nvbuf);
4396
4397     return (err);
4398 }
4399
4400 int
4401 zfs_get_holds(zfs_handle_t *zhp, nvlist_t **nvl)
4402 {
4403     int err;
4404     char errbuf[1024];
4405
4406     err = lzfs_get_holds(zhp->zfs_name, nvl);
4407
4408     if (err != 0) {
4409         libzfs_handle_t *hdl = zhp->zfs_hdl;
4410
4411         (void) snprintf(errbuf, sizeof (errbuf),
4412         dgettext(TEXT_DOMAIN, "cannot get holds for '%s'"),
4413         zhp->zfs_name);
4414         switch (err) {
4415         case ENOTSUP:
4416             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4417             "pool must be upgraded"));
4418             err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
4419             break;
4420         case EINVAL:
4421             err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
4422             break;
4423         case ENOENT:
4424             err = zfs_error(hdl, EZFS_NOENT, errbuf);
4425             break;
4426         default:
4427             err = zfs_standard_error_fmt(hdl, errno, errbuf);
4428             break;
4429         }
4430     }
4431
4432     return (err);
4433 }
4434
4435 uint64_t
4436 zvol_volsize_to_reservation(uint64_t volsize, nvlist_t *props)
4437 {
4438     uint64_t numdb;
4439     uint64_t nblocks, volblocksize;

```

```
4440     int ncopies;
4441     char *strval;
4442
4443     if (nvlist_lookup_string(props,
4444         zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)
4445         ncopies = atoi(strval);
4446     else
4447         ncopies = 1;
4448     if (nvlist_lookup_uint64(props,
4449         zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
4450         &volblocksize) != 0)
4451         volblocksize = ZVOL_DEFAULT_BLOCKSIZE;
4452     nblocks = volsize/volblocksize;
4453     /* start with metadnode L0-L6 */
4454     numdb = 7;
4455     /* calculate number of indirects */
4456     while (nblocks > 1) {
4457         nblocks += DNODES_PER_LEVEL - 1;
4458         nblocks /= DNODES_PER_LEVEL;
4459         numdb += nblocks;
4460     }
4461     numdb *= MIN(SPA_DVAS_PER_BP, ncopies + 1);
4462     volsize *= ncopies;
4463     /*
4464      * this is exactly DN_MAX_INDBLKSHIFT when metadata isn't
4465      * compressed, but in practice they compress down to about
4466      * 1100 bytes
4467      */
4468     numdb *= 1ULL << DN_MAX_INDBLKSHIFT;
4469     volsize += numdb;
4470     return (volsize);
4471 }
```

```

*****
84475 Fri May 24 00:51:02 2013
new/usr/src/lib/libzfs/common/libzfs_sendrecv.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
unchanged_portion_omitted

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

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

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

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

823     if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_SEND, &zc) != 0) {
824         char errbuf[1024];
825         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
826 "warning: cannot estimate space for '%s'", zhp->zfs_name);

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

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

```

```

840     }
841     return (zfs_error(hdl, EZFS_NOENT, errbuf));

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

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

862 *sizep = zc.zc_objset_type;

864     return (0);
865 }

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

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

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

888     VERIFY(0 == nvlist_alloc(&thisdbg, NV_UNIQUE_NAME, 0));
889     if (fromsnap && fromsnap[0] != '\0') {
890         VERIFY(0 == nvlist_add_string(thisdbg,
891 "fromsnap", fromsnap));
892     }

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

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

```



```

906     switch (errno) {
907     case EXDEV:
908         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
909             "not an earlier snapshot from the same fs"));
910         return (zfs_error(hdl, EZFS_CROSTARGET, errbuf));
911
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));
920
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));
934
935     default:
936         return (zfs_standard_error(hdl, errno, errbuf));
937     }
938 }
939
940 if (debugnv)
941     VERIFY(0 == nvlist_add_nvlist(debugnv, zhp->zfs_name, thisdbg));
942 nvlist_free(thisdbg);
943
944 return (0);
945 }
946
947 static void
948 gather_holds(zfs_handle_t *zhp, send_dump_data_t *sdd)
949 static int
950 hold_for_send(zfs_handle_t *zhp, send_dump_data_t *sdd)
951 {
952     zfs_handle_t *pzhp;
953     int error = 0;
954     char *thissnap;
955
956     assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
957
958     if (sdd->dryrun)
959         return (0);
960
961     /*
962      * zfs_send() only sets snapholds for sends that need them,
963      * zfs_send() only opens a cleanup_fd for sends that need it,
964      * e.g. replication and doall.
965      */
966     if (sdd->snapholds == NULL)
967         return;
968     if (sdd->cleanup_fd == -1)
969         return (0);
970
971     thissnap = strchr(zhp->zfs_name, '@') + 1;

```

```

816     *(thissnap - 1) = '\0';
817     pzhp = zfs_open(zhp->zfs_hdl, zhp->zfs_name, ZFS_TYPE_DATASET);
818     *(thissnap - 1) = '@';
819
820     /*
821      * It's OK if the parent no longer exists. The send code will
822      * handle that error.
823      */
824     if (pzhp) {
825         error = zfs_hold(pzhp, thissnap, sdd->holdtag,
826             B_FALSE, B_TRUE, sdd->cleanup_fd);
827         zfs_close(pzhp);
828     }
829
830     fnvlist_add_string(sdd->snapholds, zhp->zfs_name, sdd->holdtag);
831     return (error);
832 }
833
834 unchanged_portion_omitted
835
836 static int
837 dump_snapshot(zfs_handle_t *zhp, void *arg)
838 {
839     send_dump_data_t *sdd = arg;
840     progress_arg_t pa = { 0 };
841     pthread_t tid;
842
843     char *thissnap;
844     int err;
845     boolean_t isfromsnap, istosnap, fromorigin;
846     boolean_t exclude = B_FALSE;
847
848     err = 0;
849 #endif /* !codereview */
850     thissnap = strchr(zhp->zfs_name, '@') + 1;
851     isfromsnap = (sdd->fromsnap != NULL &&
852         strcmp(sdd->fromsnap, thissnap) == 0);
853
854     if (!sdd->seenfrom && isfromsnap) {
855         gather_holds(zhp, sdd);
856         err = hold_for_send(zhp, sdd);
857         if (err == 0) {
858             sdd->seenfrom = B_TRUE;
859             (void) strcpy(sdd->prevsnap, thissnap);
860             sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
861             sdd->prevsnap_obj = zfs_prop_get_int(zhp,
862                 ZFS_PROP_OBJSETID);
863         } else if (err == ENOENT) {
864             err = 0;
865         }
866         zfs_close(zhp);
867         return (0);
868     }
869     return (err);
870 }
871
872 if (sdd->seento || !sdd->seenfrom) {
873     zfs_close(zhp);
874     return (0);
875 }
876
877 istosnap = (strcmp(sdd->tosnap, thissnap) == 0);
878 if (istosnap)
879     sdd->seento = B_TRUE;
880
881 if (!sdd->doall && !isfromsnap && !istosnap) {
882     if (sdd->replicate) {
883         char *snapname;

```

```

1047     nvlist_t *snapprops;
1048     /*
1049     * Filter out all intermediate snapshots except origin
1050     * snapshots needed to replicate clones.
1051     */
1052     nvlist_t *nvfs = fsavl_find(sdd->fsavl,
1053     zhp->zfs_dmustats.dds_guid, &snapname);
1054
1055     VERIFY(0 == nvlist_lookup_nvlist(nvfs,
1056     "snapprops", &snapprops));
1057     VERIFY(0 == nvlist_lookup_nvlist(snapprops,
1058     "thissnap", &snapprops));
1059     } else {
1060     } else {
1061     exclude = !nvlist_exists(snapprops, "is_clone_origin");
1062     exclude = B_TRUE;
1063     }
1064
1065     /*
1066     * If a filter function exists, call it to determine whether
1067     * this snapshot will be sent.
1068     */
1069     if (exclude || (sdd->filter_cb != NULL &&
1070     sdd->filter_cb(zhp, sdd->filter_cb_arg) == B_FALSE)) {
1071     /*
1072     * This snapshot is filtered out. Don't send it, and don't
1073     * set prevsnap_obj, so it will be as if this snapshot didn't
1074     * exist, and the next accepted snapshot will be sent as
1075     * an incremental from the last accepted one, or as the
1076     * first (and full) snapshot in the case of a replication,
1077     * non-incremental send.
1078     */
1079     zfs_close(zhp);
1080     return (0);
1081     }
1082
1083     gather_holds(zhp, sdd);
1084     err = hold_for_send(zhp, sdd);
1085     if (err) {
1086     if (err == ENOENT)
1087     err = 0;
1088     zfs_close(zhp);
1089     return (err);
1090     }
1091
1092     fromorigin = sdd->prevsnap[0] == '\0' &&
1093     (sdd->fromorigin || sdd->replicate);
1094
1095     if (sdd->verbose) {
1096     uint64_t size;
1097     err = estimate_ioctl(zhp, sdd->prevsnap_obj,
1098     fromorigin, &size);
1099
1100     if (sdd->parsable) {
1101     if (sdd->prevsnap[0] != '\0') {
1102     (void) fprintf(stderr, "incremental\t%s\t%s",
1103     sdd->prevsnap, zhp->zfs_name);
1104     } else {
1105     (void) fprintf(stderr, "full\t%s",
1106     zhp->zfs_name);
1107     }
1108     } else {
1109     (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1110     "send from @%s to %s"),
1111     sdd->prevsnap, zhp->zfs_name);
1112     }
1113     }
1114

```

```

1105     if (err == 0) {
1106     if (sdd->parsable) {
1107     (void) fprintf(stderr, "\t%llu\n",
1108     (longlong_t)size);
1109     } else {
1110     char buf[16];
1111     zfs_nicenum(size, buf, sizeof (buf));
1112     (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1113     " estimated size is %s\n"), buf);
1114     }
1115     sdd->size += size;
1116     } else {
1117     (void) fprintf(stderr, "\n");
1118     }
1119     }
1120
1121     if (!sdd->dryrun) {
1122     /*
1123     * If progress reporting is requested, spawn a new thread to
1124     * poll ZFS_IOC_SEND_PROGRESS at a regular interval.
1125     */
1126     if (sdd->progress) {
1127     pa.pa_zhp = zhp;
1128     pa.pa_fd = sdd->outfd;
1129     pa.pa_parsable = sdd->parsable;
1130
1131     if (err = pthread_create(&tid, NULL,
1132     send_progress_thread, &pa)) {
1133     zfs_close(zhp);
1134     return (err);
1135     }
1136     }
1137
1138     err = dump_ioctl(zhp, sdd->prevsnap, sdd->prevsnap_obj,
1139     fromorigin, sdd->outfd, sdd->debugnv);
1140
1141     if (sdd->progress) {
1142     (void) pthread_cancel(tid);
1143     (void) pthread_join(tid, NULL);
1144     }
1145     }
1146
1147     (void) strcpy(sdd->prevsnap, thissnap);
1148     sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
1149     zfs_close(zhp);
1150     return (err);
1151     }
1152
1153     unchanged_portion_omitted
1154
1155     1323 /*
1156     1324 * Generate a send stream for the dataset identified by the argument zhp.
1157     1325 *
1158     1326 * The content of the send stream is the snapshot identified by
1159     1327 * 'tosnap'. Incremental streams are requested in two ways:
1160     1328 * - from the snapshot identified by "fromsnap" (if non-null) or
1161     1329 * - from the origin of the dataset identified by zhp, which must
1162     1330 * be a clone. In this case, "fromsnap" is null and "fromorigin"
1163     1331 * is TRUE.
1164     1332 *
1165     1333 * The send stream is recursive (i.e. dumps a hierarchy of snapshots) and
1166     1334 * uses a special header (with a hdrtype field of DMU_COMPOUNDSTREAM)
1167     1335 * if "replicate" is set. If "doall" is set, dump all the intermediate
1168     1336 * snapshots. The DMU_COMPOUNDSTREAM header is used in the "doall"
1169     1337 * case too. If "props" is set, send properties.
1170     1338 */
1171     1339 int

```

```

1340 zfs_send(zfs_handle_t *zhp, const char *fromsnap, const char *tosnap,
1341          sendflags_t *flags, int outfd, snapfilter_cb_t filter_func,
1342          void *cb_arg, nvlist_t **debugnvp)
1343 {
1344     char errbuf[1024];
1345     send_dump_data_t sdd = { 0 };
1346     int err = 0;
1347     nvlist_t *fss = NULL;
1348     avl_tree_t *fsavl = NULL;
1349     static uint64_t holdseq;
1350     int spa_version;
1351     pthread_t tid = 0;
1352     pthread_t tid;
1353     int pipefd[2];
1354     dedup_arg_t dda = { 0 };
1355     int featureflags = 0;
1356
1357     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
1358              "cannot send '%s'", zhp->zfs_name);
1359
1360     if (fromsnap && fromsnap[0] == '\0') {
1361         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
1362              "zero-length incremental source"));
1363         return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
1364     }
1365
1366     if (zhp->zfs_type == ZFS_TYPE_FILESYSTEM) {
1367         uint64_t version;
1368         version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1369         if (version >= ZPL_VERSION_SA) {
1370             featureflags |= DMU_BACKUP_FEATURE_SA_SPILL;
1371         }
1372     }
1373
1374     if (flags->dedup && !flags->dryrun) {
1375         featureflags |= (DMU_BACKUP_FEATURE_DEDUP |
1376             DMU_BACKUP_FEATURE_DEDUPPROPS);
1377         if (err = pipe(pipefd)) {
1378             zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1379             return (zfs_error(zhp->zfs_hdl, EZFS_PIPEFAILED,
1380                 errbuf));
1381         }
1382         dda.outputfd = outfd;
1383         dda.inputfd = pipefd[1];
1384         dda.dedup_hdl = zhp->zfs_hdl;
1385         if (err = pthread_create(&tid, NULL, cksummer, &dda)) {
1386             (void) close(pipefd[0]);
1387             (void) close(pipefd[1]);
1388             zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1389             return (zfs_error(zhp->zfs_hdl,
1390                 EZFS_THREADCREATEFAILED, errbuf));
1391         }
1392     }
1393
1394     if (flags->replicate || flags->doall || flags->props) {
1395         dmu_replay_record_t drr = { 0 };
1396         char *packbuf = NULL;
1397         size_t buflen = 0;
1398         zio_cksum_t zc = { 0 };
1399
1400         if (flags->replicate || flags->props) {
1401             nvlist_t *hdrnv;
1402
1403             VERIFY(0 == nvlist_alloc(&hdrnv, NV_UNIQUE_NAME, 0));
1404             if (fromsnap) {
1405                 VERIFY(0 == nvlist_add_string(hdrnv,

```

```

1405         "fromsnap", fromsnap));
1406     }
1407     VERIFY(0 == nvlist_add_string(hdrnv, "tosnap", tosnap));
1408     if (!flags->replicate) {
1409         VERIFY(0 == nvlist_add_boolean(hdrnv,
1410             "not_recursive"));
1411     }
1412
1413     err = gather_nvlist(zhp->zfs_hdl, zhp->zfs_name,
1414         fromsnap, tosnap, flags->replicate, &fss, &fsavl);
1415     if (err)
1416         goto err_out;
1417     VERIFY(0 == nvlist_add_nvlist(hdrnv, "fss", fss));
1418     err = nvlist_pack(hdrnv, &packbuf, &buflen,
1419         NV_ENCODE_XDR, 0);
1420     if (debugnvp)
1421         *debugnvp = hdrnv;
1422     else
1423         nvlist_free(hdrnv);
1424     if (err)
1425         if (err) {
1426             fsavl_destroy(fsavl);
1427             nvlist_free(fss);
1428             goto stderr_out;
1429         }
1430
1431     if (!flags->dryrun) {
1432         /* write first begin record */
1433         drr.drr_type = DRR_BEGIN;
1434         drr.drr_u.drr_begin.drr_magic = DMU_BACKUP_MAGIC;
1435         DMU_SET_STREAM_HDRTYPE(drr.drr_u.drr_begin,
1436             drr_versioninfo, DMU_COMPOUNDSTREAM);
1437         DMU_SET_FEATUREFLAGS(drr.drr_u.drr_begin,
1438             drr_versioninfo, featureflags);
1439         (void) snprintf(drr.drr_u.drr_begin.drr_toname,
1440             sizeof (drr.drr_u.drr_begin.drr_toname),
1441             "%s%s", zhp->zfs_name, tosnap);
1442         drr.drr_payloadlen = buflen;
1443         err = cksum_and_write(&drr, sizeof (drr), &zc, outfd);
1444
1445         /* write header nvlist */
1446         if (err != -1 && packbuf != NULL) {
1447             err = cksum_and_write(packbuf, buflen, &zc,
1448                 outfd);
1449         }
1450         free(packbuf);
1451         if (err == -1) {
1452             fsavl_destroy(fsavl);
1453             nvlist_free(fss);
1454             err = errno;
1455             goto stderr_out;
1456         }
1457     }
1458
1459     /* write end record */
1460     bzero(&drr, sizeof (drr));
1461     drr.drr_type = DRR_END;
1462     drr.drr_u.drr_end.drr_checksum = zc;
1463     err = write(outfd, &drr, sizeof (drr));
1464     if (err == -1) {
1465         fsavl_destroy(fsavl);
1466         nvlist_free(fss);
1467         err = errno;
1468         goto stderr_out;
1469     }

```

```

1463         err = 0;
1464     }
1465 }

1467 /* dump each stream */
1468 sdd.fromsnap = fromsnap;
1469 sdd.tosnap = tosnap;
1470 if (tid != 0)
1471     if (flags->dedup)
1472         sdd.outfd = pipefd[0];
1473 else
1474     sdd.outfd = outfd;
1475 sdd.replicate = flags->replicate;
1476 sdd.doall = flags->doall;
1477 sdd.fromorigin = flags->fromorigin;
1478 sdd.fss = fss;
1479 sdd.fsavl = fsavl;
1480 sdd.verbose = flags->verbose;
1481 sdd.parsable = flags->parsable;
1482 sdd.progress = flags->progress;
1483 sdd.dryrun = flags->dryrun;
1484 sdd.filter_cb = filter_func;
1485 sdd.filter_cb_arg = cb_arg;
1486 if (debugnvp)
1487     sdd.debugnv = *debugnvp;

1488 /*
1489  * Some flags require that we place user holds on the datasets that are
1490  * being sent so they don't get destroyed during the send. We can skip
1491  * this step if the pool is imported read-only since the datasets cannot
1492  * be destroyed.
1493  */
1494 if (!flags->dryrun && !zpool_get_prop_int(zfs_get_pool_handle(zhp),
1495     ZPOOL_PROP_READONLY, NULL) &&
1496     zfs_spa_version(zhp, &spa_version) == 0 &&
1497     spa_version >= SPA_VERSION_USERREFS &&
1498     (flags->doall || flags->replicate)) {
1499     ++holdseq;
1500     (void) snprintf(sdd.holdtag, sizeof (sdd.holdtag),
1501         ".send-%d-%llu", getpid(), (u_longlong_t)holdseq);
1502     sdd.cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
1503     if (sdd.cleanup_fd < 0) {
1504         err = errno;
1505         goto stderr_out;
1506     }
1507     sdd.snapholds = fnvlist_alloc();
1508 #endif /* ! codereview */
1509     } else {
1510         sdd.cleanup_fd = -1;
1511         sdd.snapholds = NULL;
1512 #endif /* ! codereview */
1513     }
1514     if (flags->verbose || sdd.snapholds != NULL) {
1515         if (flags->verbose) {
1516             /*
1517              * Do a verbose no-op dry run to get all the verbose output
1518              * or to gather snapshot hold's before generating any data,
1519              * then do a non-verbose real run to generate the streams.
1520              * before generating any data. Then do a non-verbose real
1521              * run to generate the streams.
1522              */
1523             sdd.dryrun = B_TRUE;
1524             err = dump_filesystems(zhp, &sdd);
1525         }
1526         if (err != 0)
1527             goto stderr_out;

```

```

1526         if (flags->verbose) {
1527             sdd.dryrun = flags->dryrun;
1528             sdd.verbose = B_FALSE;
1529             if (flags->parsable) {
1530                 (void) fprintf(stderr, "size\t%llu\n",
1531                     (longlong_t)sdd.size);
1532             } else {
1533                 char buf[16];
1534                 zfs_nicenum(sdd.size, buf, sizeof (buf));
1535                 (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1536                     "total estimated size is %s\n"), buf);
1537             }
1538         }
1539     }
1540 }

1542 /* Ensure no snaps found is treated as an error. */
1543 if (!sdd.seento) {
1544     err = ENOENT;
1545     goto err_out;
1546 }

1548 /* Skip the second run if dryrun was requested. */
1549 if (flags->dryrun)
1550     goto err_out;

1552 if (sdd.snapholds != NULL) {
1553     err = zfs_hold_nvlist(zhp, sdd.cleanup_fd, sdd.snapholds);
1554     if (err != 0)
1555         goto stderr_out;
1556     fnvlist_free(sdd.snapholds);
1557     sdd.snapholds = NULL;
1558 }

1560 sdd.dryrun = B_FALSE;
1561 sdd.verbose = B_FALSE;
1562 }

1564 #endif /* ! codereview */
1565 err = dump_filesystems(zhp, &sdd);
1566 fsavl_destroy(fsavl);
1567 nvlist_free(fss);

1569 /* Ensure no snaps found is treated as an error. */
1570 if (err == 0 && !sdd.seento)
1571     err = ENOENT;

1573 if (tid != 0) {
1574     if (err != 0)
1575         (void) pthread_cancel(tid);
1576     (void) pthread_join(tid, NULL);
1577     if (flags->dedup) {
1578         (void) close(pipefd[0]);
1579         (void) pthread_join(tid, NULL);
1580     }
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->replicate || flags->doall ||
1589     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.

```

```
1587         */
1588         dmu_replay_record_t drr = { 0 };
1589         drr.drr_type = DRR_END;
1590         if (write(outfd, &drr, sizeof (drr)) == -1) {
1591             return (zfs_standard_error(zhp->zfs_hdl,
1592                                     errno, errbuf));
1593         }
1594     }
1596     return (err || sdd.err);
1598 stderr_out:
1599     err = zfs_standard_error(zhp->zfs_hdl, err, errbuf);
1600 err_out:
1601     fsavl_destroy(fsavl);
1602     nvlist_free(fss);
1603     fnvlist_free(sdd.snapholds);
1605 #endif /* ! codereview */
1606     if (sdd.cleanup_fd != -1)
1607         VERIFY(0 == close(sdd.cleanup_fd));
1608     if (tid != 0) {
1441         if (flags->dedup) {
1609             (void) pthread_cancel(tid);
1610             (void) pthread_join(tid, NULL);
1611             (void) close(pipefd[0]);
1612         }
1613     }
1614 }
1613     return (err);
1614 }
unchanged_portion_omitted
```

new/usr/src/lib/libzfs_core/common/libzfs_core.c

1

```
*****
17044 Fri May 24 00:51:02 2013
new/usr/src/lib/libzfs_core/common/libzfs_core.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
unchanged_portion_omitted_
```

```
241 /*
242  * Destroys snapshots.
243  *
244  * The keys in the snaps nvlist are the snapshots to be destroyed.
245  * They must all be in the same pool.
246  *
247  * Snapshots that do not exist will be silently ignored.
248  *
249  * If 'defer' is not set, and a snapshot has user holds or clones, the
250  * destroy operation will fail and none of the snapshots will be
251  * destroyed.
252  *
253  * If 'defer' is set, and a snapshot has user holds or clones, it will be
254  * marked for deferred destruction, and will be destroyed when the last hold
255  * or clone is removed/destroyed.
256  *
257  * The return value will be ENOENT if none of the snapshots existed.
258  *
259  #endif /* ! codereview */
260  * The return value will be 0 if all snapshots were destroyed (or marked for
261  * later destruction if 'defer' is set) or didn't exist to begin with and
262  * at least one snapshot was destroyed.
263  * later destruction if 'defer' is set) or didn't exist to begin with.
264  *
265  * Otherwise the return value will be the errno of a (unspecified) snapshot
266  * that failed, no snapshots will be destroyed, and the errlist will have an
267  * entry for each snapshot that failed. The value in the errlist will be
268  * the (int32) error code.
269  */
270  int
271  lzc_destroy_snaps(nvlist_t *snaps, boolean_t defer, nvlist_t **errlist)
272  {
273      nvpair_t *elem;
274      nvlist_t *args;
275      int error;
276      char pool[MAXNAMELEN];
277
278      /* determine the pool name */
279      elem = nvlist_next_nvpair(snaps, NULL);
280      if (elem == NULL)
281          return (0);
282      (void) strncpy(pool, nvpair_name(elem), sizeof (pool));
283      pool[strcspn(pool, "/@")] = '\0';
284
285      args = fnvlist_alloc();
286      fnvlist_add_nvlist(args, "snaps", snaps);
287      if (defer)
288          fnvlist_add_boolean(args, "defer");
289
290      error = lzc_ioctl(ZFS_IOC_DESTROY_SNAPS, pool, args, errlist);
291      nvlist_free(args);
292
293      return (error);
```

unchanged_portion_omitted_

337 /*

new/usr/src/lib/libzfs_core/common/libzfs_core.c

2

```
338  * Create "user holds" on snapshots. If there is a hold on a snapshot,
339  * the snapshot can not be destroyed. (However, it can be marked for deletion
340  * by lzc_destroy_snaps(defer=B_TRUE).)
341  *
342  * The keys in the nvlist are snapshot names.
343  * The snapshots must all be in the same pool.
344  * The value is the name of the hold (string type).
345  *
346  * If cleanup_fd is not -1, it must be the result of open("/dev/zfs", O_EXCL).
347  * In this case, when the cleanup_fd is closed (including on process
348  * termination), the holds will be released. If the system is shut down
349  * uncleanly, the holds will be released when the pool is next opened
350  * or imported.
351  *
352  * Holds for snapshots which don't exist will be skipped and have an entry
353  * added to errlist, but will not cause an overall failure, except in the
354  * case that all holds were skipped.
355  *
356  * The return value will be ENOENT if none of the snapshots for the requested
357  * holds existed.
358  *
359  * The return value will be 0 if the nvl holds was empty or all holds, for
360  * snapshots that existed, were successfully created and at least one hold
361  * was created.
362  *
363  * Otherwise the return value will be the errno of a (unspecified) hold that
364  * failed and no holds will be created.
365  *
366  * In all cases the errlist will have an entry for each hold that failed
367  * (name = snapshot), with its value being the error code (int32).
368  * The return value will be 0 if all holds were created. Otherwise the return
369  * value will be the errno of a (unspecified) hold that failed, no holds will
370  * be created, and the errlist will have an entry for each hold that
371  * failed (name = snapshot). The value in the errlist will be the error
372  * code (int32).
373  */
374  int
375  lzc_hold(nvlist_t *holds, int cleanup_fd, nvlist_t **errlist)
376  {
377      char pool[MAXNAMELEN];
378      nvlist_t *args;
379      nvpair_t *elem;
380      int error;
381
382      /* determine the pool name */
383      elem = nvlist_next_nvpair(holds, NULL);
384      if (elem == NULL)
385          return (0);
386      (void) strncpy(pool, nvpair_name(elem), sizeof (pool));
387      pool[strcspn(pool, "/@")] = '\0';
388
389      args = fnvlist_alloc();
390      fnvlist_add_nvlist(args, "holds", holds);
391      if (cleanup_fd != -1)
392          fnvlist_add_int32(args, "cleanup_fd", cleanup_fd);
393
394      error = lzc_ioctl(ZFS_IOC_HOLD, pool, args, errlist);
395      nvlist_free(args);
396      return (error);
397 }
398
399 /*
400  * Release "user holds" on snapshots. If the snapshot has been marked for
401  * deferred destroy (by lzc_destroy_snaps(defer=B_TRUE)), it does not have
402  * any clones, and all the user holds are removed, then the snapshot will be
403  * destroyed.
```

```
399 *
400 * The keys in the nvlist are snapshot names.
401 * The snapshots must all be in the same pool.
402 * The value is a nvlist whose keys are the holds to remove.
403 *
404 * Holds which failed to release because they didn't exist will have an entry
405 * added to errlist, but will not cause an overall failure, except in the
406 * case that all releases were skipped.
407 *
408 * The return value will be ENOENT if none of the specified holds existed.
409 *
410 * The return value will be 0 if the nvl holds was empty or all holds, that
411 * existed, were successfully removed and at least one hold was removed.
412 *
413 * Otherwise the return value will be the errno of a (unspecified) hold that
414 * failed to release and no holds will be released.
415 *
416 * In all cases the errlist will have an entry for each hold that failed to
417 * to release.
389 * The return value will be 0 if all holds were removed.
390 * Otherwise the return value will be the errno of a (unspecified) release
391 * that failed, no holds will be released, and the errlist will have an
392 * entry for each snapshot that has failed releases (name = snapshot).
393 * The value in the errlist will be the error code (int32) of a failed release.
418 */
419 int
420 lzc_release(nvlist_t *holds, nvlist_t **errlist)
421 {
422     char pool[MAXNAMELEN];
423     nvpair_t *elem;
424
425     /* determine the pool name */
426     elem = nvlist_next_nvpair(holds, NULL);
427     if (elem == NULL)
428         return (0);
429     (void) strncpy(pool, nvpair_name(elem), sizeof (pool));
430     pool[strcspn(pool, "@/")] = '\0';
431
432     return (lzc_ioctl(ZFS_IOC_RELEASE, pool, holds, errlist));
433 }
unchanged_portion_omitted
```

```

*****
29735 Fri May 24 00:51:03 2013
new/usr/src/uts/common/fs/zfs/dsl_pool.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
_____unchanged_portion_omitted_____

828 /*
829  * Walk through the pool-wide zap object of temporary snapshot user holds
830  * and release them.
831  */
832 void
833 dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp)
834 {
835     zap_attribute_t za;
836     zap_cursor_t zc;
837     objset_t *mos = dp->dp_meta_objset;
838     uint64_t zapobj = dp->dp_tmp_userrefs_obj;
839     nvlist_t *holds;
840 #endif /* !codereview */

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

846     holds = fnvlist_alloc();

848 #endif /* !codereview */
849     for (zap_cursor_init(&zc, mos, zapobj);
850          zap_cursor_retrieve(&zc, &za) == 0;
851          zap_cursor_advance(&zc)) {
852         char *htag;
853         uint64_t dsobj;
854         nvlist_t *tags;
855 #endif /* !codereview */

857         htag = strchr(za.za_name, '-');
858         *htag = '\0';
859         ++htag;
860         if (nvlist_lookup_nvlist(holds, za.za_name, &tags) != 0) {
861             tags = fnvlist_alloc();
862             fnvlist_add_boolean(tags, htag);
863             fnvlist_add_nvlist(holds, za.za_name, tags);
864             fnvlist_free(tags);
865         } else {
866             fnvlist_add_boolean(tags, htag);
867         }
868         dsobj = strtonum(za.za_name, NULL);
869         dsl_dataset_user_release_tmp(dp, dsobj, htag);
870     }
871 #endif /* !codereview */
872     zap_cursor_fini(&zc);
873 }

875 /*
876  * Create the pool-wide zap object for storing temporary snapshot holds.
877  */
878 void
879 dsl_pool_user_hold_create_obj(dsl_pool_t *dp, dmu_tx_t *tx)
880 {
881     objset_t *mos = dp->dp_meta_objset;
883     ASSERT(dp->dp_tmp_userrefs_obj == 0);

```

```

884     ASSERT(dmu_tx_is_syncing(tx));

886     dp->dp_tmp_userrefs_obj = zap_create_link(mos, DMU_OT_USERREFS,
887        DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_TMP_USERREFS, tx);
888 }

890 static int
891 dsl_pool_user_hold_rele_impl(dsl_pool_t *dp, uint64_t dsobj,
892     const char *tag, uint64_t now, dmu_tx_t *tx, boolean_t holding)
893 {
894     objset_t *mos = dp->dp_meta_objset;
895     uint64_t zapobj = dp->dp_tmp_userrefs_obj;
896     char *name;
897     int error;

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

902     /*
903      * If the pool was created prior to SPA_VERSION_USERREFS, the
904      * zap object for temporary holds might not exist yet.
905      */
906     if (zapobj == 0) {
907         if (holding) {
908             dsl_pool_user_hold_create_obj(dp, tx);
909             zapobj = dp->dp_tmp_userrefs_obj;
910         } else {
911             return (SET_ERROR(ENOENT));
912         }
913     }

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

922     return (error);
923 }

925 /*
926  * Add a temporary hold for the given dataset object and tag.
927  */
928 int
929 dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj, const char *tag,
930     uint64_t now, dmu_tx_t *tx)
931 {
932     return (dsl_pool_user_hold_rele_impl(dp, dsobj, tag, now, tx, B_TRUE));
933 }

935 /*
936  * Release a temporary hold for the given dataset object and tag.
937  */
938 int
939 dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj, const char *tag,
940     dmu_tx_t *tx)
941 {
942     return (dsl_pool_user_hold_rele_impl(dp, dsobj, tag, NULL,
943         tx, B_FALSE));
944 }

946 /*
947  * DSL Pool Configuration Lock
948  *
949  * The dp_config_rwlock protects against changes to DSL state (e.g. dataset

```



```

950 * creation / destruction / rename / property setting). It must be held for
951 * read to hold a dataset or dsl_dir. I.e. you must call
952 * dsl_pool_config_enter() or dsl_pool_hold() before calling
953 * dsl_{dataset,dir}_hold{obj}. In most circumstances, the dp_config_rwlock
954 * must be held continuously until all datasets and dsl_dirs are released.
955 *
956 * The only exception to this rule is that if a "long hold" is placed on
957 * a dataset, then the dp_config_rwlock may be dropped while the dataset
958 * is still held. The long hold will prevent the dataset from being
959 * destroyed -- the destroy will fail with EBUSY. A long hold can be
960 * obtained by calling dsl_dataset_long_hold(), or by "owning" a dataset
961 * (by calling dsl_{dataset,objset}_{try}own{obj}).
962 *
963 * Legitimate long-holders (including owners) should be long-running, cancelable
964 * tasks that should cause "zfs destroy" to fail. This includes DMU
965 * consumers (i.e. a ZPL filesystem being mounted or ZVOL being open),
966 * "zfs send", and "zfs diff". There are several other long-holders whose
967 * uses are suboptimal (e.g. "zfs promote", and zil_suspend()).
968 *
969 * The usual formula for long-holding would be:
970 * dsl_pool_hold()
971 * dsl_dataset_hold()
972 * ... perform checks ...
973 * dsl_dataset_long_hold()
974 * dsl_pool_rele()
975 * ... perform long-running task ...
976 * dsl_dataset_long_rele()
977 * dsl_dataset_rele()
978 *
979 * Note that when the long hold is released, the dataset is still held but
980 * the pool is not held. The dataset may change arbitrarily during this time
981 * (e.g. it could be destroyed). Therefore you shouldn't do anything to the
982 * dataset except release it.
983 *
984 * User-initiated operations (e.g. ioctls, zfs_ioc_*) are either read-only
985 * or modifying operations.
986 *
987 * Modifying operations should generally use dsl_sync_task(). The sync task
988 * infrastructure enforces proper locking strategy with respect to the
989 * dp_config_rwlock. See the comment above dsl_sync_task() for details.
990 *
991 * Read-only operations will manually hold the pool, then the dataset, obtain
992 * information from the dataset, then release the pool and dataset.
993 * dmu_objset_{hold,rele}() are convenience routines that also do the pool
994 * hold/rele.
995 */
997 int
998 dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp)
999 {
1000     spa_t *spa;
1001     int error;
1002
1003     error = spa_open(name, &spa, tag);
1004     if (error == 0) {
1005         *dp = spa_get_dsl(spa);
1006         dsl_pool_config_enter(*dp, tag);
1007     }
1008     return (error);
1009 }
1011 void
1012 dsl_pool_rele(dsl_pool_t *dp, void *tag)
1013 {
1014     dsl_pool_config_exit(dp, tag);
1015     spa_close(dp->dp_spa, tag);

```

```

1016 }
1018 void
1019 dsl_pool_config_enter(dsl_pool_t *dp, void *tag)
1020 {
1021     /*
1022      * We use a "reentrant" reader-writer lock, but not reentrantly.
1023      *
1024      * The rrwlock can (with the track_all flag) track all reading threads,
1025      * which is very useful for debugging which code path failed to release
1026      * the lock, and for verifying that the *current* thread does hold
1027      * the lock.
1028      *
1029      * (Unlike a rwlock, which knows that N threads hold it for
1030      * read, but not *which* threads, so rw_held(RW_READER) returns TRUE
1031      * if any thread holds it for read, even if this thread doesn't).
1032      */
1033     ASSERT(!rrw_held(&dp->dp_config_rwlock, RW_READER));
1034     rrw_enter(&dp->dp_config_rwlock, RW_READER, tag);
1035 }
1037 void
1038 dsl_pool_config_exit(dsl_pool_t *dp, void *tag)
1039 {
1040     rrw_exit(&dp->dp_config_rwlock, tag);
1041 }
1043 boolean_t
1044 dsl_pool_config_held(dsl_pool_t *dp)
1045 {
1046     return (RRW_LOCK_HELD(&dp->dp_config_rwlock));
1047 }

```

```

*****
17754 Fri May 24 00:51:03 2013
new/usr/src/uts/common/fs/zfs/dsl_userhold.c
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2013 by Delphix. All rights reserved.
24 */

26 #include <sys/zfs_context.h>
27 #include <sys/dsl_userhold.h>
28 #include <sys/dsl_dataset.h>
29 #include <sys/dsl_destroy.h>
30 #include <sys/dsl_synctask.h>
31 #include <sys/dmu_tx.h>
32 #include <sys/zfs_onexit.h>
33 #include <sys/dsl_pool.h>
34 #include <sys/dsl_dir.h>
35 #include <sys/zfs_ioctl.h>
36 #include <sys/zap.h>

38 typedef struct dsl_dataset_user_hold_arg {
39     spa_t *dduha_spa;
40 #endif /* ! codereview */
41     nvlist_t *dduha_holds;
42     nvlist_t *dduha_chkholds;
43     nvlist_t *dduha_tmpholds;
44 #endif /* ! codereview */
45     nvlist_t *dduha_errlist;
46     minor_t dduha_minor;
47 } dsl_dataset_user_hold_arg_t;

49 /*
50 * If you add new checks here, you may need to add additional checks to the
51 * "temporary" case in snapshot_check() in dmu_objset.c.
52 */
53 int
54 dsl_dataset_user_hold_check_one(dsl_dataset_t *ds, const char *htag,
55     boolean_t temphold, dmu_tx_t *tx)
56 {
57     dsl_pool_t *dp = dmu_tx_pool(tx);
58     objset_t *mos = dp->dp_meta_objset;
59     int error = 0;

```

```

61     ASSERT(RRW_READ_HELD(&dp->dp_config_rwlock));

63 #endif /* ! codereview */
64     if (strlen(htag) > MAXNAMELEN)
65         return (E2BIG);
66     /* Tempholds have a more restricted length */
67     if (temphold && strlen(htag) + MAX_TAG_PREFIX_LEN >= MAXNAMELEN)
68         return (E2BIG);

70     /* tags must be unique (if ds already exists) */
71     if (ds != NULL && ds->ds_phys->ds_userrefs_obj != 0) {
39     if (ds != NULL) {
40         mutex_enter(&ds->ds_lock);
41         if (ds->ds_phys->ds_userrefs_obj != 0) {
72             uint64_t value;

74 #endif /* ! codereview */
75         error = zap_lookup(mos, ds->ds_phys->ds_userrefs_obj,
76             htag, 8, 1, &value);
77         if (error == 0)
78             error = SET_ERROR(EEXIST);
79         else if (error == ENOENT)
80             error = 0;
81     }
43     mutex_exit(&ds->ds_lock);
44 }

83     return (error);
84 }

86 static int
87 dsl_dataset_user_hold_check(void *arg, dmu_tx_t *tx)
88 {
89     dsl_dataset_user_hold_arg_t *dduha = arg;
90     dsl_pool_t *dp = dmu_tx_pool(tx);
91     nvpair_t *pair;
55     int rv = 0;

93     if (spa_version(dp->dp_spa) < SPA_VERSION_USERREFS)
94         return (SET_ERROR(ENOTSUP));

96     if (!dmu_tx_is_syncing(tx))
97         return (0);

99 #endif /* ! codereview */
100     for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;
101         pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {
102         dsl_dataset_t *ds;
103 #endif /* ! codereview */
104         int error = 0;
105         char *htag, *name;
106         dsl_dataset_t *ds;
107         char *htag;

107         /* must be a snapshot */
108         name = nvpair_name(pair);
109         if (strchr(name, '@') == NULL)
64         if (strchr(nvpair_name(pair), '@') == NULL)
110             error = SET_ERROR(EINVAL);

112         if (error == 0)
113             error = nvpair_value_string(pair, &htag);

115         if (error == 0)
116             error = dsl_dataset_hold(dp, name, FTAG, &ds);

```

```

69     if (error == 0) {
70         error = dsl_dataset_hold(dp,
71             nvpair_name(pair), FTAG, &ds);
72     }
118    if (error == 0) {
119        error = dsl_dataset_user_hold_check_one(ds, htag,
120            dduha->dduha_minor != 0, tx);
121        dsl_dataset_rele(ds, FTAG);
122    }

124    if (error == 0) {
125        fnvlist_add_string(dduha->dduha_chkhlds, name, htag);
126    } else {
127        /*
128         * We register ENOENT errors so they can be correctly
129         * reported if needed, such as when all holds fail.
130         */
131        fnvlist_add_int32(dduha->dduha_errlist, name, error);
132        if (error != ENOENT)
133            return (error);
79    if (error != 0) {
80        rv = error;
81        fnvlist_add_int32(dduha->dduha_errlist,
82            nvpair_name(pair), error);
134    }
135 }

137 /* Return ENOENT if no holds would be created. */
138 if (nvlist_next_nvpair(dduha->dduha_chkhlds, NULL) == NULL)
139     return (ENOENT);

141     return (0);
85     return (rv);
142 }

145 static void
146 dsl_dataset_user_hold_sync_one_impl(nvlist_t *tmpholds, dsl_dataset_t *ds,
147     const char *htag, minor_t minor, uint64_t now, dmu_tx_t *tx)
88 void
89 dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
90     minor_t minor, uint64_t now, dmu_tx_t *tx)
148 {
149     dsl_pool_t *dp = ds->ds_dir->dd_pool;
150     objset_t *mos = dp->dp_meta_objset;
151     uint64_t zapobj;

153     ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

96     mutex_enter(&ds->ds_lock);
155     if (ds->ds_phys->ds_userrefs_obj == 0) {
156         /*
157          * This is the first user hold for this dataset. Create
158          * the userrefs zap object.
159          */
160         dmu_buf_will_dirty(ds->ds_dbuf, tx);
161         zapobj = ds->ds_phys->ds_userrefs_obj =
162             zap_create(mos, DMU_OT_USERREFS, DMU_OT_NONE, 0, tx);
163     } else {
164         zapobj = ds->ds_phys->ds_userrefs_obj;
165     }
166     ds->ds_userrefs++;
109     mutex_exit(&ds->ds_lock);

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

```

```

170     if (minor != 0) {
171         char name[MAXNAMELEN];
172         nvlist_t *tags;

174 #endif /* ! codereview */
175         VERIFY0(dsl_pool_user_hold(dp, ds->ds_object,
176             htag, now, tx));
177         (void) snprintf(name, sizeof(name), "%llx",
178             (u_longlong_t)ds->ds_object);

180         if (nvlist_lookup_nvlist(tmpholds, name, &tags) != 0) {
181             tags = fnvlist_alloc();
182             fnvlist_add_boolean(tags, htag);
183             fnvlist_add_nvlist(tmpholds, name, tags);
184             fnvlist_free(tags);
185         } else {
186             fnvlist_add_boolean(tags, htag);
187         }
114         dsl_register_onexit_hold_cleanup(ds, htag, minor);
188     }

190     spa_history_log_internal_ds(ds, "hold", tx,
191         "tag=%s temp=%d refs=%llu",
192         htag, minor != 0, ds->ds_userrefs);
193 }

195 typedef struct zfs_hold_cleanup_arg {
196     char zhca_spaname[MAXNAMELEN];
197     uint64_t zhca_spa_load_guid;
198     nvlist_t *zhca_holds;
199 } zfs_hold_cleanup_arg_t;

201 static void
202 dsl_dataset_user_release_onexit(void *arg)
203 {
204     zfs_hold_cleanup_arg_t *ca = (zfs_hold_cleanup_arg_t *)arg;
205     spa_t *spa;
206     int error;

208     error = spa_open(ca->zhca_spaname, &spa, FTAG);
209     if (error != 0) {
210         zfs_dbgmsg("couldn't release holds on pool=%s "
211             "because pool is no longer loaded",
212             ca->zhca_spaname);
213         return;
214     }
215     if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
216         zfs_dbgmsg("couldn't release holds on pool=%s "
217             "because pool is no longer loaded (guid doesn't match)",
218             ca->zhca_spaname);
219         spa_close(spa, FTAG);
220         return;
221     }

223     (void) dsl_dataset_user_release_tmp(spa_get_dsl(spa), ca->zhca_holds);
224     fnvlist_free(ca->zhca_holds);
225     kmem_free(ca, sizeof(zfs_hold_cleanup_arg_t));
226     spa_close(spa, FTAG);
227 }

229 static void
230 dsl_register_onexit_hold_cleanup(spa_t *spa, nvlist_t *holds, minor_t minor)
231 {
232     zfs_hold_cleanup_arg_t *ca;

234     if (minor == 0 || nvlist_next_nvpair(holds, NULL) == NULL) {

```

```

235         fnvlist_free(holds);
236         return;
237     }

239     ASSERT(spa != NULL);
240     ca = kmem_alloc(sizeof (*ca), KM_SLEEP);

242     (void) strcpy(ca->zhca_spaname, spa_name(spa),
243                 sizeof (ca->zhca_spaname));
244     ca->zhca_spa_load_guid = spa_load_guid(spa);
245     ca->zhca_holds = holds;
246     VERIFY0(zfs_onexit_add_cb(minor,
247                               dsl_dataset_user_release_onexit, ca, NULL));
248 }

250 void
251 dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
252                               minor_t minor, uint64_t now, dmu_tx_t *tx)
253 {
254     nvlist_t *tmpholds;

256     tmpholds = fnvlist_alloc();

258     dsl_dataset_user_hold_sync_one_impl(tmpholds, ds, htag, minor, now, tx);
259     dsl_register_onexit_hold_cleanup(dsl_dataset_get_spa(ds), tmpholds,
260                                    minor);
261 }

263 #endif /* ! codereview */
264 static void
265 dsl_dataset_user_hold_sync(void *arg, dmu_tx_t *tx)
266 {
267     dsl_dataset_user_hold_arg_t *dduha = arg;
268     dsl_pool_t *dp = dmu_tx_pool(tx);
269     nvpair_t *pair;
270     uint64_t now = gethrstime_sec();

272     for (pair = nvlist_next_nvpair(dduha->dduha_chkholds, NULL);
273          pair != NULL;
274          pair = nvlist_next_nvpair(dduha->dduha_chkholds, pair)) {
275         for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;
276              pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {
277             dsl_dataset_t *ds;

278             VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
279             dsl_dataset_user_hold_sync_one_impl(dduha->dduha_tmpholds, ds,
280                                                fnvpair_value_string(pair), dduha->dduha_minor, now, tx);
281             dsl_dataset_user_hold_sync_one(ds, fnvpair_value_string(pair),
282                                           dduha->dduha_minor, now, tx);
283             dsl_dataset_rele(ds, FTAG);
284         }
285     }
286     dduha->dduha_spa = dp->dp_spa;
287 #endif /* ! codereview */
288 }

287 /*
288  * The full semantics of this function are described in the comment above
289  * lzc_hold().
290  *
291  * To summarize:
292  * #endif /* ! codereview */
293  * holds is nvl of snapname -> holdname
294  * errlist will be filled in with snapname -> error
295  * if cleanup_minor is not 0, the holds will be temporary, cleaned up
296  * when the process exits.

```

```

295  *
296  * The snapshots must all be in the same pool.
297  *
298  * Holds for snapshots that don't exist will be skipped.
299  *
300  * If none of the snapshots for requested holds exist then ENOENT will be
301  * returned.
302  *
303  * If cleanup_minor is not 0, the holds will be temporary, which will be cleaned
304  * up when the process exits.
305  *
306  * On success all the holds, for snapshots that existed, will be created and 0
307  * will be returned.
308  *
309  * On failure no holds will be created, the errlist will be filled in,
310  * and an errno will be returned.
311  *
312  * In all cases the errlist will contain entries for holds where the snapshot
313  * didn't exist.
314  * if any fails, all will fail.
315  */
316 int
317 dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor, nvlist_t *errlist)
318 {
319     dsl_dataset_user_hold_arg_t dduha;
320     nvpair_t *pair;
321     int ret;
322 #endif /* ! codereview */

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

327     dduha.dduha_spa = NULL;
328 #endif /* ! codereview */
329     dduha.dduha_holds = holds;
330     dduha.dduha_chkholds = fnvlist_alloc();
331     dduha.dduha_tmpholds = fnvlist_alloc();
332 #endif /* ! codereview */
333     dduha.dduha_errlist = errlist;
334     dduha.dduha_minor = cleanup_minor;

336     ret = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
337                        dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds));

339     /* dsl_register_onexit_hold_cleanup() always frees the passed holds. */
340     dsl_register_onexit_hold_cleanup(dduha.dduha_spa, dduha.dduha_tmpholds,
341                                     cleanup_minor);
342     fnvlist_free(dduha.dduha_chkholds);

344     return (ret);
345     return (dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
346                          dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds)));
347 }

347 typedef int (dsl_holdfunc_t)(dsl_pool_t *dp, const char *name, void *tag,
348                             dsl_dataset_t **dsp);

350 #endif /* ! codereview */
351 typedef struct dsl_dataset_user_release_arg {
352     dsl_holdfunc_t *ddura_holdfunc;
353 #endif /* ! codereview */
354     nvlist_t *ddura_holds;
355     nvlist_t *ddura_todelete;
356     nvlist_t *ddura_errlist;
357     nvlist_t *ddura_chkholds;

```

```

358 #endif /* ! codereview */
359 } dsl_dataset_user_release_arg_t;

361 /* Place a dataset hold on the snapshot identified by passed dsobj string */
362 static int
363 dsl_dataset_hold_obj_string(dsl_pool_t *dp, const char *dsobj, void *tag,
364     dsl_dataset_t **dsp)
365 {
366     return dsl_dataset_hold_obj(dp, strtonum(dsobj, NULL), tag, dsp);
367 }

369 #endif /* ! codereview */
370 static int
371 dsl_dataset_user_release_check_one(dsl_dataset_user_release_arg_t *ddura,
372     dsl_dataset_t *ds, nvlist_t *holds, const char *name)
373 dsl_dataset_user_release_check_one(dsl_dataset_t *ds,
374     nvlist_t *holds, boolean_t *todelete)
375 {
376     uint64_t zapobj;
377     nvpair_t *pair;
378     nvlist_t *holds_found;
379 #endif /* ! codereview */
380     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
381     int ret, numholds;
382     int error;
383     int numholds = 0;

151     *todelete = B_FALSE;

381     if (!dsl_dataset_is_snapshot(ds))
382         return (SET_ERROR(EINVAL));

384     zapobj = ds->ds_phys->ds_userrefs_obj;
385     if (zapobj == 0)
386         return (SET_ERROR(ESRCH));

388     ret = 0;
389     numholds = 0;
390     holds_found = fnvlist_alloc();

392 #endif /* ! codereview */
393     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
394         pair = nvlist_next_nvpair(holds, pair)) {
160         /* Make sure the hold exists */
395         uint64_t tmp;
396         int error;
397         const char *name;

399         name = nvpair_name(pair);
400         error = zap_lookup(mos, zapobj, name, 8, 1, &tmp);

402         /* Non-existent holds aren't always an error. */
162         error = zap_lookup(mos, zapobj, nvpair_name(pair), 8, 1, &tmp);
403         if (error == ENOENT)
404             continue;

406         if (error != 0) {
407             fnvlist_free(holds_found);
164             error = SET_ERROR(ESRCH);
165             if (error != 0)
408                 return (error);
409         }

411         fnvlist_add_boolean(holds_found, name);
412 #endif /* ! codereview */
413     numholds++;

```

```

414     }

416     if (DS_IS_DEFER_DESTROY(ds) && ds->ds_phys->ds_num_children == 1 &&
417         ds->ds_userrefs == numholds) {
418         /* we need to destroy the snapshot as well */
419         if (dsl_dataset_long_held(ds)) {
420             fnvlist_free(holds_found);

168             if (dsl_dataset_long_held(ds))
421                 return (SET_ERROR(EBUSY));
170             *todelete = B_TRUE;
422         }
423         fnvlist_add_boolean(ddura->ddura_todelete, name);
424     }

426     if (numholds == 0)
427         ret = ENOENT;
428     else
429         fnvlist_add_nvlist(ddura->ddura_chkhlds, name, holds_found);
430     fnvlist_free(holds_found);

432     return (ret);
172     return (0);
433 }

435 static int
436 dsl_dataset_user_release_check(void *arg, dmu_tx_t *tx)
437 {
438     dsl_dataset_user_release_arg_t *ddura;
439     dsl_holdfunc_t *holdfunc;
440     dsl_pool_t *dp;
178     dsl_dataset_user_release_arg_t *ddura = arg;
179     dsl_pool_t *dp = dmu_tx_pool(tx);
441     nvpair_t *pair;
181     int rv = 0;

443     if (!dmu_tx_is_syncing(tx))
444         return (0);

446     ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

448     dp = dmu_tx_pool(tx);
449     ddura = (dsl_dataset_user_release_arg_t *)arg;
450     holdfunc = ddura->ddura_holdfunc;

452 #endif /* ! codereview */
453     for (pair = nvlist_next_nvpair(ddura->ddura_holds, NULL); pair != NULL;
454         pair = nvlist_next_nvpair(ddura->ddura_holds, pair)) {
455         const char *name;
186         const char *name = nvpair_name(pair);
456         int error;
457         dsl_dataset_t *ds;
458         nvlist_t *holds;

460         name = nvpair_name(pair);
461 #endif /* ! codereview */
462         error = nvpair_value_nvlist(pair, &holds);
463         if (error != 0)
464             error = (SET_ERROR(EINVAL));
465         if (error == 0)
466             error = holdfunc(dp, name, FTAG, &ds);
191         return (SET_ERROR(EINVAL));

193         error = dsl_dataset_hold(dp, name, FTAG, &ds);
467         if (error == 0) {
468             error = dsl_dataset_user_release_check_one(ddura, ds,

```

```

469         holds, name);
195         boolean_t deleteme;
196         error = dsl_dataset_user_release_check_one(ds,
197             holds, &deleteme);
198         if (error == 0 && deleteme) {
199             fnvlist_add_boolean(ddura->ddura_todelete,
200                 name);
201         }
470         dsl_dataset_rele(ds, FTAG);
471     }
472     if (error != 0) {
473         if (ddura->ddura_errlist != NULL) {
474             fnvlist_add_int32(ddura->ddura_errlist, name,
475                 error);
206             fnvlist_add_int32(ddura->ddura_errlist,
207                 name, error);
476         }
477         /* Non-existent holds aren't always an error. */
478         if (error != ENOENT)
479             return (error);
209         rv = error;
480     }
481 }

483 /*
484  * Return ENOENT if none of the holds existed avoiding the overhead
485  * of a sync.
486  */
487 if (nvlist_next_nvpair(ddura->ddura_chkholds, NULL) == NULL)
488     return (ENOENT);

490 return (0);
212 return (rv);
491 }

493 static void
494 dsl_dataset_user_release_sync_one(dsl_dataset_user_release_arg_t *ddura,
495     dsl_dataset_t *ds, nvlist_t *holds, dmu_tx_t *tx)
216 dsl_dataset_user_release_sync_one(dsl_dataset_t *ds, nvlist_t *holds,
217     dmu_tx_t *tx)
496 {
497     dsl_pool_t *dp = ds->ds_dir->dd_pool;
498     objset_t *mos = dp->dp_meta_objset;
221     uint64_t zapobj;
222     int error;
499     nvpair_t *pair;

501     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
502         pair = nvlist_next_nvpair(holds, pair)) {
503         uint64_t zapobj;
504         int error;
505         const char *name;

507         name = nvpair_name(pair);

509         /* Remove temporary hold if one exists. */
510         error = dsl_pool_user_release(dp, ds->ds_object, name, tx);
227         ds->ds_userrefs--;
228         error = dsl_pool_user_release(dp, ds->ds_object,
229             nvpair_name(pair), tx);
511         VERIFY(error == 0 || error == ENOENT);

513         /* Remove user hold if one exists. */
514 #endif /* ! codereview */
515         zapobj = ds->ds_phys->ds_userrefs_obj;
516         error = zap_remove(mos, zapobj, name, tx);

```

```

517         if (error == ENOENT)
518             continue;
519         VERIFY0(error);

521         /* Only if we removed a hold do we decrement ds_userrefs. */
522         ds->ds_userrefs--;
231         VERIFY0(zap_remove(mos, zapobj, nvpair_name(pair), tx));

524         spa_history_log_internal_ds(ds, "release", tx,
525             "tag=%s refs=%lld", nvpair_name(pair),
526             (longlong_t)ds->ds_userrefs);
527     }
528 }

530 static void
531 dsl_dataset_user_release_sync(void *arg, dmu_tx_t *tx)
532 {
533     dsl_dataset_user_release_arg_t *ddura = arg;
534     dsl_holdfunc_t *holdfunc = ddura->ddura_holdfunc;
535 #endif /* ! codereview */
536     dsl_pool_t *dp = dmu_tx_pool(tx);
537     nvpair_t *pair;

539     ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

541     for (pair = nvlist_next_nvpair(ddura->ddura_chkholds, NULL);
542         pair != NULL; pair = nvlist_next_nvpair(ddura->ddura_chkholds,
543             pair)) {
243         for (pair = nvlist_next_nvpair(ddura->ddura_holds, NULL); pair != NULL;
244             pair = nvlist_next_nvpair(ddura->ddura_holds, pair)) {
544             dsl_dataset_t *ds;
545             const char *name;
546 #endif /* ! codereview */

548             name = nvpair_name(pair);
549             VERIFY0(holdfunc(dp, name, FTAG, &ds));

551             dsl_dataset_user_release_sync_one(ddura, ds,
246             VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
247             dsl_dataset_user_release_sync_one(ds,
552                 fnvpair_value_nvlist(pair), tx);
553             if (nvlist_exists(ddura->ddura_todelete, name)) {
249                 if (nvlist_exists(ddura->ddura_todelete,
250                     nvpair_name(pair))) {
554                     ASSERT(ds->ds_userrefs == 0 &&
555                         ds->ds_phys->ds_num_children == 1 &&
556                         DS_IS_DEFER_DESTROY(ds));
557                     dsl_destroy_snapshot_sync_impl(ds, B_FALSE, tx);
558                 }
559                 dsl_dataset_rele(ds, FTAG);
560             }
561 }

563 /*
564  * The full semantics of this function are described in the comment above
565  * lz_c_release().
566  *
567  * To summarize:
568  * Releases holds specified in the nvl holds.
569  *
570 #endif /* ! codereview */
571 * holds is nvl of snapname -> { holdname, ... }
572 * errlist will be filled in with snapname -> error
573 *
574 * If tmpdp is not NULL the names for holds should be the dsobj's of snapshots,
575 * otherwise they should be the names of shapshots.

```

```

576 *
577 * As a release may cause snapshots to be destroyed this tries to ensure they
578 * aren't mounted.
579 *
580 * The release of non-existent holds are skipped.
581 *
582 * At least one hold must have been released for the this function to succeed
583 * and return 0.
261 * if any fails, all will fail.
584 */
585 static int
586 dsl_dataset_user_release_impl(nvlist_t *holds, nvlist_t *errlist,
587 dsl_pool_t *tmpdp)
263 int
264 dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist)
588 {
589     dsl_dataset_user_release_arg_t ddura;
590     nvpair_t *pair;
591     char *pool;
592 #endif /* ! codereview */
593     int error;

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

599 #ifdef _KERNEL
600     /*
601      * The release may cause snapshots to be destroyed; make sure they
602      * are not mounted.
603      */
604     if (tmpdp != NULL) {
605         /* Temporary holds are specified by dsojb string. */
606         ddura.ddura_holdfunc = dsl_dataset_hold_obj_string;
607         pool = spa_name(tmpdp->dp_spa);
608         ddura.ddura_holds = holds;
609         ddura.ddura_errlist = errlist;
610         ddura.ddura_todelete = fvnlist_alloc();
609         dsl_pool_config_enter(tmpdp, FTAG);
610         for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
611             pair = nvlist_next_nvpair(holds, pair)) {
612             error = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_release_check,
613                 dsl_dataset_user_release_sync, &ddura, fvnlist_num_pairs(holds));
614             fvnlist_free(ddura.ddura_todelete);
615             return (error);
616         }

617     typedef struct dsl_dataset_user_release_tmp_arg {
618         uint64_t ddurta_dsojb;
619         nvlist_t *ddurta_holds;
620         boolean_t ddurta_deleteme;
621     } dsl_dataset_user_release_tmp_arg_t;

622     static int
623     dsl_dataset_user_release_tmp_check(void *arg, dmu_tx_t *tx)
624     {
625         dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
626         dsl_pool_t *dp = dmu_tx_pool(tx);
627         dsl_dataset_t *ds;
628         int error;

629         error = dsl_dataset_hold_obj_string(tmpdp,
630             nvpair_name(pair), FTAG, &ds);
631         if (!dmu_tx_is_syncing(tx))
632             return (0);

```

```

295     error = dsl_dataset_hold_obj(dp, ddurta->ddurta_dsojb, FTAG, &ds);
296     if (error)
297         return (error);

299     error = dsl_dataset_user_release_check_one(ds,
300         ddurta->ddurta_holds, &ddurta->ddurta_deleteme);
301     dsl_dataset_rele(ds, FTAG);
302     return (error);
303 }

305 static void
306 dsl_dataset_user_release_tmp_sync(void *arg, dmu_tx_t *tx)
307 {
308     dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
309     dsl_pool_t *dp = dmu_tx_pool(tx);
310     dsl_dataset_t *ds;

312     VERIFY0(dsl_dataset_hold_obj(dp, ddurta->ddurta_dsojb, FTAG, &ds));
313     dsl_dataset_user_release_sync_one(ds, ddurta->ddurta_holds, tx);
314     if (ddurta->ddurta_deleteme) {
315         ASSERT(ds->ds_userrefs == 0 &&
316             ds->ds_phys->ds_num_children == 1 &&
317             DS_IS_DEFER_DESTROY(ds));
318         dsl_destroy_snapshot_sync_impl(ds, B_FALSE, tx);
319     }
320     dsl_dataset_rele(ds, FTAG);
321 }

323 /*
324  * Called at spa_load time to release a stale temporary user hold.
325  * Also called by the onexit code.
326  */
327 void
328 dsl_dataset_user_release_tmp(dsl_pool_t *dp, uint64_t dsojb, const char *htag)
329 {
330     dsl_dataset_user_release_tmp_arg_t ddurta;
331     dsl_dataset_t *ds;
332     int error;

334 #ifdef _KERNEL
335     /* Make sure it is not mounted. */
336     dsl_pool_config_enter(dp, FTAG);
337     error = dsl_dataset_hold_obj(dp, dsojb, FTAG, &ds);
338     if (error == 0) {
339         char name[MAXNAMELEN];
340         dsl_dataset_name(ds, name);
341         dsl_dataset_rele(ds, FTAG);
342         dsl_pool_config_exit(dp, FTAG);
343         zfs_unmount_snap(name);
344     }
345     dsl_pool_config_exit(tmpdp, FTAG);
346 #endif /* ! codereview */
347     } else {
348         /* Non-temporary holds are specified by name. */
349         ddura.ddura_holdfunc = dsl_dataset_hold;
350         pool = nvpair_name(pair);
351         for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
352             pair = nvlist_next_nvpair(holds, pair)) {
353             zfs_unmount_snap(nvpair_name(pair));
354         }
355         dsl_pool_config_exit(dp, FTAG);
356     }
357 #endif

```

```

637     ddura.ddura_holds = holds;
638     ddura.ddura_errlist = errlist;
639     ddura.ddura_todelete = fnvlist_alloc();
640     ddura.ddura_chkholds = fnvlist_alloc();

642     error = dsl_sync_task(pool, dsl_dataset_user_release_check,
643         dsl_dataset_user_release_sync, &ddura,
644         fnvlist_num_pairs(holds));
645     fnvlist_free(ddura.ddura_todelete);
646     fnvlist_free(ddura.ddura_chkholds);

648     return (error);
348     ddurta.ddurta_dsobj = dsobj;
349     ddurta.ddurta_holds = fnvlist_alloc();
350     fnvlist_add_boolean(ddurta.ddurta_holds, htag);

352     (void) dsl_sync_task(spa_name(dp->dp_spa),
353         dsl_dataset_user_release_tmp_check,
354         dsl_dataset_user_release_tmp_sync, &ddurta, 1);
355     fnvlist_free(ddurta.ddurta_holds);
649 }

```

```

651 /*
652  * holds is nvl of snapname -> { holdname, ... }
653  * errlist will be filled in with snapname -> error
654  */
655 int
656 dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist)
358 typedef struct zfs_hold_cleanup_arg {
359     char zhca_spaname[MAXNAMELEN];
360     uint64_t zhca_spa_load_guid;
361     uint64_t zhca_dsobj;
362     char zhca_htag[MAXNAMELEN];
363 } zfs_hold_cleanup_arg_t;

```

```

365 static void
366 dsl_dataset_user_release_onexit(void *arg)
657 {
658     return dsl_dataset_user_release_impl(holds, errlist, NULL);
368     zfs_hold_cleanup_arg_t *ca = arg;
369     spa_t *spa;
370     int error;

372     error = spa_open(ca->zhca_spaname, &spa, FTAG);
373     if (error != 0) {
374         zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
375             "because pool is no longer loaded",
376             ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
377         return;
378     }
379     if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
380         zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
381             "because pool is no longer loaded (guid doesn't match)",
382             ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
383         spa_close(spa, FTAG);
384         return;
385     }

387     dsl_dataset_user_release_tmp(spa_get_dsl(spa),
388         ca->zhca_dsobj, ca->zhca_htag);
389     kmem_free(ca, sizeof (zfs_hold_cleanup_arg_t));
390     spa_close(spa, FTAG);
659 }

```

```
661 /*
```

```

662  * holds is nvl of snapsobj -> { holdname, ... }
663  */
664 #endif /* ! codereview */
665 void
666 dsl_dataset_user_release_tmp(struct dsl_pool *dp, nvlist_t *holds)
393 dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
394     minor_t minor)
667 {
668     ASSERT(dp != NULL);
669     (void) dsl_dataset_user_release_impl(holds, NULL, dp);
396     zfs_hold_cleanup_arg_t *ca = kmem_alloc(sizeof (*ca), KM_SLEEP);
397     spa_t *spa = dsl_dataset_get_spa(ds);
398     (void) strncpy(ca->zhca_spaname, spa_name(spa),
399         sizeof (ca->zhca_spaname));
400     ca->zhca_spa_load_guid = spa_load_guid(spa);
401     ca->zhca_dsobj = ds->ds_object;
402     (void) strncpy(ca->zhca_htag, htag, sizeof (ca->zhca_htag));
403     VERIFY0(zfs_onexit_add_cb(minor,
404         dsl_dataset_user_release_onexit, ca, NULL));
670 }
    _____unchanged_portion_omitted_____

```



```

*****
10147 Fri May 24 00:51:03 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
3740 Poor ZFS send / receive performance due to snapshot hold / release process!
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
_____unchanged_portion_omitted_____

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

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

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

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

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

206 spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);

208 boolean_t dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds);

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

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

220 void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
221 void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);

```

```

222 void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
223 void dsl_dataset_space(dsl_dataset_t *ds,
224     uint64_t *refdbbytesp, uint64_t *availbytesp,
225     uint64_t *usedobjsp, uint64_t *availobjsp);
226 uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
227 int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
228     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
229 int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
230     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
231 boolean_t dsl_dataset_is_dirty(dsl_dataset_t *ds);

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

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

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

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

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

268 #ifdef ZFS_DEBUG
269 #define dprintf_ds(ds, fmt, ...) do { \
270     if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
271         char *__ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
272         dsl_dataset_name(ds, __ds_name); \
273         dprintf("ds=%s " fmt, __ds_name, _VA_ARGS__); \
274         kmem_free(__ds_name, MAXNAMELEN); \
275     } \
276     _NOTE(CONSTCOND) } while (0)
277 #else
278 #define dprintf_ds(ds, fmt, ...)
279 #endif

281 #ifdef __cplusplus
282 }
_____unchanged_portion_omitted_____

```

new/usr/src/uts/common/fs/zfs/sys/dsl_userhold.h

1

1826 Fri May 24 00:51:03 2013

new/usr/src/uts/common/fs/zfs/sys/dsl_userhold.h

3740 Poor ZFS send / receive performance due to snapshot hold / release processi

Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>

```
2 /*
3  * CDDL HEADER START
4  *
5  * The contents of this file are subject to the terms of the
6  * Common Development and Distribution License (the "License").
7  * You may not use this file except in compliance with the License.
8  *
9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 * or http://www.opensolaris.org/os/licensing.
11 * See the License for the specific language governing permissions
12 * and limitations under the License.
13 *
14 * When distributing Covered Code, include this CDDL HEADER in each
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 * If applicable, add the following below this CDDL HEADER, with the
17 * fields enclosed by brackets "[]" replaced with your own identifying
18 * information: Portions Copyright [yyyy] [name of copyright owner]
19 *
20 * CDDL HEADER END
21 */
22 /*
23 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24 * Copyright (c) 2012 by Delphix. All rights reserved.
25 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
26 */

28 #ifndef _SYS_DSL_USERHOLD_H
29 #define _SYS_DSL_USERHOLD_H

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

34 #ifdef __cplusplus
35 extern "C" {
36 #endif

38 struct dsl_pool;
39 struct dsl_dataset;
40 struct dmu_tx;

42 int dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor,
43     nvlist_t *errlist);
44 int dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist);
45 int dsl_dataset_get_holds(const char *dsname, nvlist_t *nvl);
46 void dsl_dataset_user_release_tmp(struct dsl_pool *dp, nvlist_t *holds);
46 void dsl_dataset_user_release_tmp(struct dsl_pool *dp, uint64_t dsobj,
47     const char *htag);
47 int dsl_dataset_user_hold_check_one(struct dsl_dataset *ds, const char *htag,
48     boolean_t temphold, struct dmu_tx *tx);
49 void dsl_dataset_user_hold_sync_one(struct dsl_dataset *ds, const char *htag,
50     minor_t minor, uint64_t now, struct dmu_tx *tx);

52 #ifdef __cplusplus
53 }
_____unchanged_portion_omitted_____
```

new/usr/src/uts/common/fs/zfs/zfs_ioctl.c

1

143884 Fri May 24 00:51:03 2013

new/usr/src/uts/common/fs/zfs/zfs_ioctl.c

3740 Poor ZFS send / receive performance due to snapshot hold / release processi

Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>

unchanged_portion_omitted

```
4968 /*
4969  * innvl: {
4970  *     snapname -> { holdname, ... }
4971  *     ...
4972  * }
4973  *
4974  * outnvl: {
4975  *     snapname -> error value (int32)
4976  *     ...
4977  * }
4978  */
4979 /* ARGSUSED */
4980 static int
4981 zfs_ioc_release(const char *pool, nvlist_t *holds, nvlist_t *errlist)
4982 {
4983     nvpair_t *pair;
4984
4985     /*
4986      * The release may cause the snapshot to be destroyed; make sure it
4987      * is not mounted.
4988      */
4989     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
4990          pair = nvlist_next_nvpair(holds, pair))
4991         zfs_unmount_snap(nvpair_name(pair));
4983     return (dsl_dataset_user_release(holds, errlist));
4984 }
```

unchanged_portion_omitted