

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

1

```
*****
8478 Wed May  1 01:43:39 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_____
```

```

*****
161499 Wed May 1 01:43:40 2013
new/usr/src/cmd/zfs/zfs_main.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

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);

```

```

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 Wed May 1 01:43:40 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_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
```

```

*****
110624 Wed May 1 01:43:40 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.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_

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

```

```

4127
4128 #endif /* ! codereview */
4129     if (ret == 0)
4130         return (0);

4132     if (nvlist_next_nvpair(errors, NULL) == NULL) {
4133         /* no hold-specific errors */
4134         (void) snprintf(errbuf, sizeof (errbuf),
4135             dgettext(TEXT_DOMAIN, "cannot hold"));
4136         switch (ret) {
4137             case ENOTSUP:
4138                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
4139                     "pool must be upgraded"));
4140                 (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
4141                 break;
4142             case EINVAL:
4143                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4144                 break;
4145             default:
4146                 (void) zfs_standard_error(hdl, ret, errbuf);
4147         }
4148     }

4150     for (elem = nvlist_next_nvpair(errors, NULL);
4151         elem != NULL;
4152         elem = nvlist_next_nvpair(errors, elem)) {
4153         (void) snprintf(errbuf, sizeof (errbuf),
4154             dgettext(TEXT_DOMAIN,
4155                 "cannot hold snapshot '%s'", nvpair_name(elem)));
4156         switch (fnvpair_value_int32(elem)) {
4157             case E2BIG:
4158                 /*
4159                  * Temporary tags wind up having the ds object id
4160                  * prepended. So even if we passed the length check
4161                  * above, it's still possible for the tag to wind
4162                  * up being slightly too long.
4163                  */
4164                 (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
4165                 break;
4166             case EINVAL:
4167                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4168                 break;
4169             case EEXIST:
4170                 (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
4171                 break;
4172             case ENOENT:
4173                 if (enoent_ok)
4174                     return (ENOENT);
4175                 /* FALLTHROUGH */
4176             default:
4177                 (void) zfs_standard_error(hdl,
4178                     fnvpair_value_int32(elem), errbuf);
4179         }
4180     }

4178     fnvlist_free(errors);
4179     return (ret);
4180 }

4136 struct releasearg {
4137     nvlist_t *nvl;
4138     const char *snapname;
4139     const char *tag;
4140     boolean_t recursive;
4141 };

```

```
4182 static int
4183 zfs_release_one(zfs_handle_t *zhp, void *arg)
4184 {
4185     struct holdarg *ha = arg;
4186     zfs_handle_t *szhp;
4187     char name[ZFS_MAXNAMELEN];
4188     int rv = 0;
4189
4190     (void) snprintf(name, sizeof (name),
4191                    "%s@%s", zhp->zfs_name, ha->snapname);
4192
4193     if (lzc_exists(name)) {
4194         szhp = make_dataset_handle(zhp->zfs_hdl, name);
4195         if (szhp) {
4196             nvlist_t *holds = fnvlist_alloc();
4197             fnvlist_add_boolean(holds, ha->tag);
4198             fnvlist_add_nvlist(ha->nvl, name, holds);
4199             fnvlist_free(holds);
4200             zfs_close(szhp);
4201         }
4202     }
4203
4204     if (ha->recursive)
4205         rv = zfs_iter_filesystems(zhp, zfs_release_one, ha);
4206     zfs_close(zhp);
4207     return (rv);
4208 }
4209
4210 _____unchanged_portion_omitted_____
```

```

*****
84377 Wed May 1 01:43:41 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_CROSSTARGET, 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     thissnap = strchr(zhp->zfs_name, '@') + 1;
849     isfromsnap = (sdd->fromsnap != NULL &&
850         strcmp(sdd->fromsnap, thissnap) == 0);
851
852     if (!sdd->seenfrom && isfromsnap) {
853         gather_holds(zhp, sdd);
854         err = hold_for_send(zhp, sdd);
855         if (err == 0) {
856             sdd->seenfrom = B_TRUE;
857             (void) strcpy(sdd->prevsnap, thissnap);
858             sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
859             sdd->prevsnap_obj = zfs_prop_get_int(zhp,
860                 ZFS_PROP_OBJSETID);
861         } else if (err == ENOENT) {
862             err = 0;
863         }
864         zfs_close(zhp);
865         return (0);
866     }
867     return (err);
868 }
869
870 if (sdd->seento || !sdd->seenfrom) {
871     zfs_close(zhp);
872     return (0);
873 }
874
875 istosnap = (strcmp(sdd->tosnap, thissnap) == 0);
876 if (istosnap)
877     sdd->seento = B_TRUE;
878
879 if (!sdd->doall && !isfromsnap && !istosnap) {
880     if (sdd->replicate) {
881         char *snapname;
882         nvlist_t *snapprops;
883     }
884 }

```

```

1048         * Filter out all intermediate snapshots except origin
1049         * snapshots needed to replicate clones.
1050         */
1051         nvlist_t *nvfs = fsavl_find(sdd->fsavl,
1052         zhp->zfs_dmustats.dds_guid, &snapname);
1053
1054         VERIFY(0 == nvlist_lookup_nvlist(nvfs,
1055         "snapprops", &snapprops));
1056         VERIFY(0 == nvlist_lookup_nvlist(snapprops,
1057         "thissnap", &snapprops));
1058         exclude = !nvlist_exists(snapprops, "is_clone_origin");
1059     } else {
1060         exclude = B_TRUE;
1061     }
1062 }
1063
1064 /*
1065  * If a filter function exists, call it to determine whether
1066  * this snapshot will be sent.
1067  */
1068 if (exclude || (sdd->filter_cb != NULL &&
1069 sdd->filter_cb(zhp, sdd->filter_cb_arg) == B_FALSE)) {
1070     /*
1071      * This snapshot is filtered out. Don't send it, and don't
1072      * set prevsnap_obj, so it will be as if this snapshot didn't
1073      * exist, and the next accepted snapshot will be sent as
1074      * an incremental from the last accepted one, or as the
1075      * first (and full) snapshot in the case of a replication,
1076      * non-incremental send.
1077      */
1078     zfs_close(zhp);
1079     return (0);
1080 }
1081
1082 gather_holds(zhp, sdd);
1083 err = hold_for_send(zhp, sdd);
1084 if (err) {
1085     if (err == ENOENT)
1086         err = 0;
1087     zfs_close(zhp);
1088     return (err);
1089 }
1090
1091 fromorigin = sdd->prevsnap[0] == '\0' &&
1092 (sdd->fromorigin || sdd->replicate);
1093
1094 if (sdd->verbose) {
1095     uint64_t size;
1096     err = estimate_ioctl(zhp, sdd->prevsnap_obj,
1097         fromorigin, &size);
1098 }
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 if (err == 0) {
1115     if (sdd->parsable) {

```

```

1106         (void) fprintf(stderr, "\t%llu\n",
1107             (longlong_t)size);
1108     } else {
1109         char buf[16];
1110         zfs_nicenum(size, buf, sizeof (buf));
1111         (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1112             " estimated size is %s\n"), buf);
1113     }
1114     sdd->size += size;
1115 } else {
1116     (void) fprintf(stderr, "\n");
1117 }
1118 }
1119
1120 if (!sdd->dryrun) {
1121     /*
1122      * If progress reporting is requested, spawn a new thread to
1123      * poll ZFS_IOC_SEND_PROGRESS at a regular interval.
1124      */
1125     if (sdd->progress) {
1126         pa.pa_zhp = zhp;
1127         pa.pa_fd = sdd->outfd;
1128         pa.pa_parsable = sdd->parsable;
1129
1130         if (err = pthread_create(&tid, NULL,
1131             send_progress_thread, &pa)) {
1132             zfs_close(zhp);
1133             return (err);
1134         }
1135     }
1136
1137     err = dump_ioctl(zhp, sdd->prevsnap, sdd->prevsnap_obj,
1138         fromorigin, sdd->outfd, sdd->debugnv);
1139
1140     if (sdd->progress) {
1141         (void) pthread_cancel(tid);
1142         (void) pthread_join(tid, NULL);
1143     }
1144 }
1145
1146 (void) strcpy(sdd->prevsnap, thissnap);
1147 sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
1148 zfs_close(zhp);
1149 return (err);
1150 }
1151
1152 _____ unchanged portion omitted _____
1153
1154 1322 /*
1155 1323  * Generate a send stream for the dataset identified by the argument zhp.
1156 1324  *
1157 1325  * The content of the send stream is the snapshot identified by
1158 1326  * 'tosnap'. Incremental streams are requested in two ways:
1159 1327  * - from the snapshot identified by "fromsnap" (if non-null) or
1160 1328  * - from the origin of the dataset identified by zhp, which must
1161 1329  *   be a clone. In this case, "fromsnap" is null and "fromorigin"
1162 1330  *   is TRUE.
1163 1331  *
1164 1332  * The send stream is recursive (i.e. dumps a hierarchy of snapshots) and
1165 1333  * uses a special header (with a hdrtype field of DMU_COMPOUNDSTREAM)
1166 1334  * if "replicate" is set. If "doall" is set, dump all the intermediate
1167 1335  * snapshots. The DMU_COMPOUNDSTREAM header is used in the "doall"
1168 1336  * case too. If "props" is set, send properties.
1169 1337  */
1338 int
1339 zfs_send(zfs_handle_t *zhp, const char *fromsnap, const char *tosnap,
1340     sendflags_t *flags, int outfd, snapfilter_cb_t filter_func,

```



```

1341 void *cb_arg, nvlist_t **debugnvp)
1342 {
1343     char errbuf[1024];
1344     send_dump_data_t sdd = { 0 };
1345     int err = 0;
1346     nvlist_t *fss = NULL;
1347     avl_tree_t *fsavl = NULL;
1348     static uint64_t holdseq;
1349     int spa_version;
1350     pthread_t tid;
1351     int pipefd[2];
1352     dedup_arg_t dda = { 0 };
1353     int featureflags = 0;
1354
1355     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
1356         "cannot send '%s'"), zhp->zfs_name);
1357
1358     if (fromsnap && fromsnap[0] == '\0') {
1359         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
1360             "zero-length incremental source"));
1361         return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
1362     }
1363
1364     if (zhp->zfs_type == ZFS_TYPE_FILESYSTEM) {
1365         uint64_t version;
1366         version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
1367         if (version >= ZPL_VERSION_SA) {
1368             featureflags |= DMU_BACKUP_FEATURE_SA_SPILL;
1369         }
1370     }
1371
1372     if (flags->dedup && !flags->dryrun) {
1373         featureflags |= (DMU_BACKUP_FEATURE_DEDUP |
1374             DMU_BACKUP_FEATURE_DEDUPPROPS);
1375         if (err = pipe(pipefd)) {
1376             zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1377             return (zfs_error(zhp->zfs_hdl, EZFS_PIPEFAILED,
1378                 errbuf));
1379         }
1380         dda.outputfd = outfd;
1381         dda.inputfd = pipefd[1];
1382         dda.dedup_hdl = zhp->zfs_hdl;
1383         if (err = pthread_create(&tid, NULL, cksummer, &dda)) {
1384             (void) close(pipefd[0]);
1385             (void) close(pipefd[1]);
1386             zfs_error_aux(zhp->zfs_hdl, strerror(errno));
1387             return (zfs_error(zhp->zfs_hdl,
1388                 EZFS_THREADCREATEFAILED, errbuf));
1389         }
1390     }
1391
1392     if (flags->replicate || flags->doall || flags->props) {
1393         dmuf_replay_record_t drr = { 0 };
1394         char *packbuf = NULL;
1395         size_t buflen = 0;
1396         zio_cksum_t zc = { 0 };
1397
1398         if (flags->replicate || flags->props) {
1399             nvlist_t *hdrnv;
1400
1401             VERIFY(0 == nvlist_alloc(&hdrnv, NV_UNIQUE_NAME, 0));
1402             if (fromsnap) {
1403                 VERIFY(0 == nvlist_add_string(hdrnv,
1404                     "fromsnap", fromsnap));
1405             }
1406             VERIFY(0 == nvlist_add_string(hdrnv, "tosnap", tosnap));

```

```

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

```

```

1473      /* dump each stream */
1474      sdd.fromsnap = fromsnap;
1475      sdd.tosnap = tosnap;
1476      if (flags->dedup)
1477          sdd.outfd = pipefd[0];
1478      else
1479          sdd.outfd = outfd;
1480      sdd.replicate = flags->replicate;
1481      sdd.doall = flags->doall;
1482      sdd.fromorigin = flags->fromorigin;
1483      sdd.fss = fss;
1484      sdd.fsavl = fsavl;
1485      sdd.verbose = flags->verbose;
1486      sdd.parsable = flags->parsable;
1487      sdd.progress = flags->progress;
1488      sdd.dryrun = flags->dryrun;
1489      sdd.filter_cb = filter_func;
1490      sdd.filter_cb_arg = cb_arg;
1491      if (debugnvp)
1492          sdd.debugnv = *debugnvp;

1494      /*
1495       * Some flags require that we place user holds on the datasets that are
1496       * being sent so they don't get destroyed during the send. We can skip
1497       * this step if the pool is imported read-only since the datasets cannot
1498       * be destroyed.
1499       */
1500      if (!flags->dryrun && !zpool_get_prop_int(zfs_get_pool_handle(zhp),
1501          ZPOOL_PROP_READONLY, NULL) &&
1502          zfs_spa_version(zhp, &spa_version) == 0 &&
1503          spa_version >= SPA_VERSION_USERREFS &&
1504          (flags->doall || flags->replicate)) {
1505          ++holdseq;
1506          (void) snprintf(sdd.holdtag, sizeof (sdd.holdtag),
1507              ".send-%d-%llu", getpid(), (u_longlong_t)holdseq);
1508          sdd.cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
1509          if (sdd.cleanup_fd < 0) {
1510              err = errno;
1511              goto stderr_out;
1512          }
1513          sdd.snapholds = fnvlist_alloc();
1514      #endif /* ! codereview */
1515      } else {
1516          sdd.cleanup_fd = -1;
1517          sdd.snapholds = NULL;
1518      #endif /* ! codereview */
1519      }
1520      if (flags->verbose) {
1521          /*
1522           * Do a verbose no-op dry run to get all the verbose output
1523           * before generating any data. Then do a non-verbose real
1524           * run to generate the streams.
1525           */
1526          sdd.dryrun = B_TRUE;
1527          err = dump_filesystems(zhp, &sdd);
1528          sdd.dryrun = flags->dryrun;
1529          sdd.verbose = B_FALSE;
1530          if (flags->parsable) {
1531              (void) fprintf(stderr, "size\t%llu\n",
1532                  (longlong_t)sdd.size);
1533          } else {
1534              char buf[16];
1535              zfs_nicenum(sdd.size, buf, sizeof (buf));
1536              (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1537                  "total estimated size is %s\n"), buf);
1538          }

```

```

1539      }
1540      if (sdd.snapholds != NULL) {
1541          /* Holds are required. */
1542          if (!flags->verbose) {
1543              /*
1544               * A verbose dry run wasn't done so do a non-verbose
1545               * dry run to gather snapshot hold's.
1546               */
1547              sdd.dryrun = B_TRUE;
1548              err = dump_filesystems(zhp, &sdd);
1549              sdd.dryrun = flags->dryrun;
1550          }
1551      }
1552      if (err != 0) {
1553          fnvlist_free(sdd.snapholds);
1554          goto stderr_out;
1555      }
1556      err = zfs_hold_nvl(zhp, sdd.cleanup_fd, sdd.snapholds);
1557      fnvlist_free(sdd.snapholds);
1558      if (err != 0)
1559          goto stderr_out;
1560      #endif /* ! codereview */
1561      err = dump_filesystems(zhp, &sdd);
1562      fsavl_destroy(fsavl);
1563      nvlist_free(fss);

1564      if (flags->dedup) {
1565          (void) close(pipefd[0]);
1566          (void) pthread_join(tid, NULL);
1567      }

1568      if (sdd.cleanup_fd != -1) {
1569          VERIFY(0 == close(sdd.cleanup_fd));
1570          sdd.cleanup_fd = -1;
1571      }

1572      if (!flags->dryrun && (flags->replicate || flags->doall ||
1573          flags->props)) {
1574          /*
1575           * write final end record. NB: want to do this even if
1576           * there was some error, because it might not be totally
1577           * failed.
1578           */
1579          dmuf_replay_record_t drr = { 0 };
1580          drr.drr_type = DRR_END;
1581          if (write(outfd, &drr, sizeof (drr)) == -1) {
1582              return (zfs_standard_error(zhp->zfs_hdl,
1583                  errno, errbuf));
1584          }
1585      }

1586      return (err || sdd.err);

1587      stderr_out:
1588      err = zfs_standard_error(zhp->zfs_hdl, err, errbuf);
1589      err_out:
1590      if (sdd.cleanup_fd != -1)
1591          VERIFY(0 == close(sdd.cleanup_fd));
1592      if (flags->dedup) {
1593          (void) pthread_cancel(tid);
1594          (void) pthread_join(tid, NULL);
1595          (void) close(pipefd[0]);

```

```

1605     }
1606     return (err);
1607 }

1609 /*
1610  * Routines specific to "zfs recv"
1611  */

1613 static int
1614 recv_read(libzfs_handle_t *hdl, int fd, void *buf, int ilen,
1615           boolean_t byteswap, zio_cksum_t *zc)
1616 {
1617     char *cp = buf;
1618     int rv;
1619     int len = ilen;

1621     do {
1622         rv = read(fd, cp, len);
1623         cp += rv;
1624         len -= rv;
1625     } while (rv > 0);

1627     if (rv < 0 || len != 0) {
1628         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
1629                                   "failed to read from stream"));
1630         return (zfs_error(hdl, EZFS_BADSTREAM, dgettext(TEXT_DOMAIN,
1631                                                         "cannot receive")));
1632     }

1634     if (zc) {
1635         if (byteswap)
1636             fletcher_4_incremental_byteswap(buf, ilen, zc);
1637         else
1638             fletcher_4_incremental_native(buf, ilen, zc);
1639     }
1640     return (0);
1641 }

1643 static int
1644 recv_read_nvlist(libzfs_handle_t *hdl, int fd, int len, nvlist_t **nvp,
1645                 boolean_t byteswap, zio_cksum_t *zc)
1646 {
1647     char *buf;
1648     int err;

1650     buf = zfs_alloc(hdl, len);
1651     if (buf == NULL)
1652         return (ENOMEM);

1654     err = recv_read(hdl, fd, buf, len, byteswap, zc);
1655     if (err != 0) {
1656         free(buf);
1657         return (err);
1658     }

1660     err = nvlist_unpack(buf, len, nvp, 0);
1661     free(buf);
1662     if (err != 0) {
1663         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
1664                                   "stream (malformed nvlist)"));
1665         return (EINVAL);
1666     }
1667     return (0);
1668 }

1670 static int

```

```

1671 recv_rename(libzfs_handle_t *hdl, const char *name, const char *tryname,
1672             int baselen, char *newname, recvflags_t *flags)
1673 {
1674     static int seq;
1675     zfs_cmd_t zc = { 0 };
1676     int err;
1677     prop_changelist_t *clp;
1678     zfs_handle_t *zhp;

1680     zhp = zfs_open(hdl, name, ZFS_TYPE_DATASET);
1681     if (zhp == NULL)
1682         return (-1);
1683     clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
1684                            flags->force ? MS_FORCE : 0);
1685     zfs_close(zhp);
1686     if (clp == NULL)
1687         return (-1);
1688     err = changelist_prefix(clp);
1689     if (err)
1690         return (err);

1692     zc.zc_objset_type = DMU_OST_ZFS;
1693     (void) strlcpy(zc.zc_name, name, sizeof (zc.zc_name));

1695     if (tryname) {
1696         (void) strcpy(newname, tryname);

1698         (void) strlcpy(zc.zc_value, tryname, sizeof (zc.zc_value));

1700         if (flags->verbose) {
1701             (void) printf("attempting rename %s to %s\n",
1702                          zc.zc_name, zc.zc_value);
1703         }
1704         err = ioctl(hdl->libzfs_fd, ZFS_IOC_RENAME, &zc);
1705         if (err == 0)
1706             changelist_rename(clp, name, tryname);
1707     } else {
1708         err = ENOENT;
1709     }

1711     if (err != 0 && strncmp(name + baselen, "recv-", 5) != 0) {
1712         seq++;

1714         (void) snprintf(newname, ZFS_MAXNAMELEN, "%.*srecv-%u-%u",
1715                        baselen, name, getpid(), seq);
1716         (void) strlcpy(zc.zc_value, newname, sizeof (zc.zc_value));

1718         if (flags->verbose) {
1719             (void) printf("failed - trying rename %s to %s\n",
1720                          zc.zc_name, zc.zc_value);
1721         }
1722         err = ioctl(hdl->libzfs_fd, ZFS_IOC_RENAME, &zc);
1723         if (err == 0)
1724             changelist_rename(clp, name, newname);
1725         if (err && flags->verbose) {
1726             (void) printf("failed (%u) - "
1727                          "will try again on next pass\n", errno);
1728         }
1729         err = EAGAIN;
1730     } else if (flags->verbose) {
1731         if (err == 0)
1732             (void) printf("success\n");
1733         else
1734             (void) printf("failed (%u)\n", errno);
1735     }

```

```

1737     (void) changelist_postfix(clp);
1738     changelist_free(clp);

1740     return (err);
1741 }

1743 static int
1744 recv_destroy(libzfs_handle_t *hdl, const char *name, int baselen,
1745             char *newname, recvflags_t *flags)
1746 {
1747     zfs_cmd_t zc = { 0 };
1748     int err = 0;
1749     prop_changelist_t *clp;
1750     zfs_handle_t *zhp;
1751     boolean_t defer = B_FALSE;
1752     int spa_version;

1754     zhp = zfs_open(hdl, name, ZFS_TYPE_DATASET);
1755     if (zhp == NULL)
1756         return (-1);
1757     clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
1758                          flags->force ? MS_FORCE : 0);
1759     if (zfs_get_type(zhp) == ZFS_TYPE_SNAPSHOT &&
1760         zfs_spa_version(zhp, &spa_version) == 0 &&
1761         spa_version >= SPA_VERSION_USERREFS)
1762         defer = B_TRUE;
1763     zfs_close(zhp);
1764     if (clp == NULL)
1765         return (-1);
1766     err = changelist_prefix(clp);
1767     if (err)
1768         return (err);

1770     zc.zc_objset_type = DMU_OST_ZFS;
1771     zc.zc_defer_destroy = defer;
1772     (void) strncpy(zc.zc_name, name, sizeof (zc.zc_name));

1774     if (flags->verbose)
1775         (void) printf("attempting destroy %s\n", zc.zc_name);
1776     err = ioctl(hdl->libzfs_fd, ZFS_IOC_DESTROY, &zc);
1777     if (err == 0) {
1778         if (flags->verbose)
1779             (void) printf("success\n");
1780         changelist_remove(clp, zc.zc_name);
1781     }

1783     (void) changelist_postfix(clp);
1784     changelist_free(clp);

1786     /*
1787      * Deferred destroy might destroy the snapshot or only mark it to be
1788      * destroyed later, and it returns success in either case.
1789      */
1790     if (err != 0 || (defer && zfs_dataset_exists(hdl, name,
1791         ZFS_TYPE_SNAPSHOT))) {
1792         err = recv_rename(hdl, name, NULL, baselen, newname, flags);
1793     }

1795     return (err);
1796 }

1798 typedef struct guid_to_name_data {
1799     uint64_t guid;
1800     char *name;
1801     char *skip;
1802 } guid_to_name_data_t;

```

```

1804 static int
1805 guid_to_name_cb(zfs_handle_t *zhp, void *arg)
1806 {
1807     guid_to_name_data_t *gtnd = arg;
1808     int err;

1810     if (gtnd->skip != NULL &&
1811         strcmp(zhp->zfs_name, gtnd->skip) == 0) {
1812         return (0);
1813     }

1815     if (zhp->zfs_dmustats.dds_guid == gtnd->guid) {
1816         (void) strcpy(gtnd->name, zhp->zfs_name);
1817         zfs_close(zhp);
1818         return (EEXIST);
1819     }

1821     err = zfs_iter_children(zhp, guid_to_name_cb, gtnd);
1822     zfs_close(zhp);
1823     return (err);
1824 }

1826 /*
1827  * Attempt to find the local dataset associated with this guid.  In the case of
1828  * multiple matches, we attempt to find the "best" match by searching
1829  * progressively larger portions of the hierarchy.  This allows one to send a
1830  * tree of datasets individually and guarantee that we will find the source
1831  * guid within that hierarchy, even if there are multiple matches elsewhere.
1832  */
1833 static int
1834 guid_to_name(libzfs_handle_t *hdl, const char *parent, uint64_t guid,
1835             char *name)
1836 {
1837     /* exhaustive search all local snapshots */
1838     char pname[ZFS_MAXNAMELEN];
1839     guid_to_name_data_t gtnd;
1840     int err = 0;
1841     zfs_handle_t *zhp;
1842     char *cp;

1844     gtnd.guid = guid;
1845     gtnd.name = name;
1846     gtnd.skip = NULL;

1848     (void) strncpy(pname, parent, sizeof (pname));

1850     /*
1851      * Search progressively larger portions of the hierarchy.  This will
1852      * select the "most local" version of the origin snapshot in the case
1853      * that there are multiple matching snapshots in the system.
1854      */
1855     while ((cp = strrchr(pname, '/')) != NULL) {

1857         /* Chop off the last component and open the parent */
1858         *cp = '\0';
1859         zhp = make_dataset_handle(hdl, pname);

1861         if (zhp == NULL)
1862             continue;

1864         err = zfs_iter_children(zhp, guid_to_name_cb, &gtnd);
1865         zfs_close(zhp);
1866         if (err == EEXIST)
1867             return (0);

```



```

2133         fsname, fromsnap);
2134     }
2135     continue;
2136 }

2138 VERIFY(0 == nvlist_lookup_string(stream_nvfs,
2139     "name", &stream_fsname));
2140 VERIFY(0 == nvlist_lookup_uint64(stream_nvfs,
2141     "parentfromsnap", &stream_parent_fromsnap_guid));

2143 s1 = strrchr(fsname, '/');
2144 s2 = strrchr(stream_fsname, '/');

2146 /*
2147  * Check for rename. If the exact receive path is specified, it
2148  * does not count as a rename, but we still need to check the
2149  * datasets beneath it.
2150  */
2151 if ((stream_parent_fromsnap_guid != 0 &&
2152     parent_fromsnap_guid != 0 &&
2153     stream_parent_fromsnap_guid != parent_fromsnap_guid) ||
2154     ((flags->isprefix || strcmp(tofs, fsname) != 0) &&
2155     (s1 != NULL && (s2 != NULL) && strcmp(s1, s2) != 0)) {
2156     nvlist_t *parent;
2157     char tryname[ZFS_MAXNAMELEN];

2159     parent = fsavl_find(local_avl,
2160         stream_parent_fromsnap_guid, NULL);
2161     /*
2162      * NB: parent might not be found if we used the
2163      * tosnap for stream_parent_fromsnap_guid,
2164      * because the parent is a newly-created fs;
2165      * we'll be able to rename it after we recv the
2166      * new fs.
2167      */
2168     if (parent != NULL) {
2169         char *pname;

2171         VERIFY(0 == nvlist_lookup_string(parent, "name",
2172             &pname));
2173         (void) snprintf(tryname, sizeof(tryname),
2174             "%s%s", pname, strrchr(stream_fsname, '/'));
2175     } else {
2176         tryname[0] = '\0';
2177         if (flags->verbose) {
2178             (void) printf("local fs %s new parent "
2179                 "not found\n", fsname);
2180         }
2181     }

2183     newname[0] = '\0';

2185     error = recv_rename(hdl, fsname, tryname,
2186         strlen(tofs)+1, newname, flags);

2188     if (renamed != NULL && newname[0] != '\0') {
2189         VERIFY(0 == nvlist_add_boolean(renamed,
2190             newname));
2191     }

2193     if (error)
2194         needagain = B_TRUE;
2195     else
2196         progress = B_TRUE;
2197 }
2198 }

```

```

2200     fsavl_destroy(local_avl);
2201     nvlist_free(local_nv);

2203     if (needagain && progress) {
2204         /* do another pass to fix up temporary names */
2205         if (flags->verbose)
2206             (void) printf("another pass:\n");
2207         goto again;
2208     }

2210     return (needagain);
2211 }

2213 static int
2214 zfs_receive_package(libzfs_handle_t *hdl, int fd, const char *destname,
2215     recvflags_t *flags, dmuf_replay_record_t *drr, zio_cksum_t *zc,
2216     char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
2217 {
2218     nvlist_t *stream_nv = NULL;
2219     avl_tree_t *stream_avl = NULL;
2220     char *fromsnap = NULL;
2221     char *cp;
2222     char tofs[ZFS_MAXNAMELEN];
2223     char sendfs[ZFS_MAXNAMELEN];
2224     char errbuf[1024];
2225     dmuf_replay_record_t drre;
2226     int error;
2227     boolean_t anyerr = B_FALSE;
2228     boolean_t softerr = B_FALSE;
2229     boolean_t recursive;

2231     (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
2232         "cannot receive"));

2234     assert(drr->drr_type == DRR_BEGIN);
2235     assert(drr->drr_u.drr_begin.drr_magic == DMU_BACKUP_MAGIC);
2236     assert(DMU_GET_STREAM_HDRTYPE(drr->drr_u.drr_begin.drr_versioninfo) ==
2237         DMU_COMPOUNDSTREAM);

2239     /*
2240      * Read in the nvlist from the stream.
2241      */
2242     if (drr->drr_payloadlen != 0) {
2243         error = recv_read_nvlist(hdl, fd, drr->drr_payloadlen,
2244             &stream_nv, flags->byteswap, zc);
2245         if (error) {
2246             error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2247             goto out;
2248         }
2249     }

2251     recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
2252         ENOENT);

2254     if (recursive && strchr(destname, '@')) {
2255         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2256             "cannot specify snapshot name for multi-snapshot stream"));
2257         error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2258         goto out;
2259     }

2261     /*
2262      * Read in the end record and verify checksum.
2263      */
2264     if (0 != (error = recv_read(hdl, fd, &drre, sizeof(drre),

```

```

2265     flags->byteswap, NULL))
2266     goto out;
2267     if (flags->byteswap) {
2268         drre.drr_type = BSWAP_32(drre.drr_type);
2269         drre.drr_u.drr_end.drr_checksum.zc_word[0] =
2270             BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[0]);
2271         drre.drr_u.drr_end.drr_checksum.zc_word[1] =
2272             BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[1]);
2273         drre.drr_u.drr_end.drr_checksum.zc_word[2] =
2274             BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[2]);
2275         drre.drr_u.drr_end.drr_checksum.zc_word[3] =
2276             BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[3]);
2277     }
2278     if (drre.drr_type != DRR_END) {
2279         error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2280         goto out;
2281     }
2282     if (!ZIO_CHECKSUM_EQUAL(drre.drr_u.drr_end.drr_checksum, *zc) {
2283         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2284             "incorrect header checksum"));
2285         error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2286         goto out;
2287     }
2289     (void) nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap);
2291     if (drr->drr_payloadlen != 0) {
2292         nvlist_t *stream_fss;
2294         VERIFY(0 == nvlist_lookup_nvlist(stream_nv, "fss",
2295             &stream_fss));
2296         if ((stream_avl = fsavl_create(stream_fss)) == NULL) {
2297             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2298                 "couldn't allocate avl tree"));
2299             error = zfs_error(hdl, EZFS_NOMEM, errbuf);
2300             goto out;
2301         }
2303         if (fromsnap != NULL) {
2304             nvlist_t *renamed = NULL;
2305             nvpair_t *pair = NULL;
2307             (void) strlcpy(tofs, destname, ZFS_MAXNAMELEN);
2308             if (flags->isprefix) {
2309                 struct drr_begin *drrb = &drr->drr_u.drr_begin;
2310                 int i;
2312                 if (flags->istail) {
2313                     cp = strrchr(drrb->drr_toname, '/');
2314                     if (cp == NULL) {
2315                         (void) strlcat(tofs, "/",
2316                             ZFS_MAXNAMELEN);
2317                         i = 0;
2318                     } else {
2319                         i = (cp - drrb->drr_toname);
2320                     }
2321                 } else {
2322                     i = strcspn(drrb->drr_toname, "@/");
2323                 }
2324                 /* zfs_receive_one() will create_parents() */
2325                 (void) strlcat(tofs, &drrb->drr_toname[i],
2326                     ZFS_MAXNAMELEN);
2327                 *strchr(tofs, '@') = '\0';
2328             }
2330             if (recursive && !flags->dryrun && !flags->nomount) {

```

```

2331         VERIFY(0 == nvlist_alloc(&renamed,
2332             NV_UNIQUE_NAME, 0));
2333     }
2335     softerr = recv_incremental_replication(hdl, tofs, flags,
2336         stream_nv, stream_avl, renamed);
2338     /* Unmount renamed filesystems before receiving. */
2339     while ((pair = nvlist_next_nvpair(renamed,
2340         pair)) != NULL) {
2341         zfs_handle_t *zhp;
2342         prop_changelist_t *clp = NULL;
2344         zhp = zfs_open(hdl, nvpair_name(pair),
2345             ZFS_TYPE_FILESYSTEM);
2346         if (zhp != NULL) {
2347             clp = changelist_gather(zhp,
2348                 ZFS_PROP_MOUNTPOINT, 0, 0);
2349             zfs_close(zhp);
2350             if (clp != NULL) {
2351                 softerr |=
2352                     changelist_prefix(clp);
2353                 changelist_free(clp);
2354             }
2355         }
2356     }
2358     nvlist_free(renamed);
2359 }
2360 }
2362 /*
2363  * Get the fs specified by the first path in the stream (the top level
2364  * specified by 'zfs send') and pass it to each invocation of
2365  * zfs_receive_one().
2366  */
2367 (void) strlcpy(sendfs, drr->drr_u.drr_begin.drr_toname,
2368     ZFS_MAXNAMELEN);
2369 if ((cp = strchr(sendfs, '@')) != NULL)
2370     *cp = '\0';
2372 /* Finally, receive each contained stream */
2373 do {
2374     /*
2375      * we should figure out if it has a recoverable
2376      * error, in which case do a recv_skip() and drive on.
2377      * Note, if we fail due to already having this guid,
2378      * zfs_receive_one() will take care of it (ie,
2379      * recv_skip() and return 0).
2380      */
2381     error = zfs_receive_impl(hdl, destname, flags, fd,
2382         sendfs, stream_nv, stream_avl, top_zfs, cleanup_fd,
2383         action_handlep);
2384     if (error == ENODATA) {
2385         error = 0;
2386         break;
2387     }
2388     anyerr |= error;
2389 } while (error == 0);
2391 if (drr->drr_payloadlen != 0 && fromsnap != NULL) {
2392     /*
2393      * Now that we have the fs's they sent us, try the
2394      * renames again.
2395      */
2396     softerr = recv_incremental_replication(hdl, tofs, flags,

```



```

2397         stream_nv, stream_avl, NULL);
2398     }
2400 out:
2401     fsavl_destroy(stream_avl);
2402     if (stream_nv)
2403         nvlist_free(stream_nv);
2404     if (softerr)
2405         error = -2;
2406     if (anyerr)
2407         error = -1;
2408     return (error);
2409 }
2411 static void
2412 trunc_prop_errs(int truncated)
2413 {
2414     ASSERT(truncated != 0);
2416     if (truncated == 1)
2417         (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
2418             "1 more property could not be set\n"));
2419     else
2420         (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
2421             "%d more properties could not be set\n"), truncated);
2422 }
2424 static int
2425 recv_skip(libzfs_handle_t *hdl, int fd, boolean_t byteswap)
2426 {
2427     dmuf_replay_record_t *drr;
2428     void *buf = malloc(1<<20);
2429     char errbuf[1024];
2431     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2432         "cannot receive:"));
2434     /* XXX would be great to use lseek if possible... */
2435     drr = buf;
2437     while (recv_read(hdl, fd, drr, sizeof (dmuf_replay_record_t),
2438         byteswap, NULL) == 0) {
2439         if (byteswap)
2440             drr->drr_type = BSWAP_32(drr->drr_type);
2442         switch (drr->drr_type) {
2443         case DRR_BEGIN:
2444             /* NB: not to be used on v2 stream packages */
2445             if (drr->drr_payloadlen != 0) {
2446                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2447                     "invalid substream header"));
2448                 return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2449             }
2450             break;
2452         case DRR_END:
2453             free(buf);
2454             return (0);
2456         case DRR_OBJECT:
2457             if (byteswap) {
2458                 drr->drr_u.drr_object.drr_bonuslen =
2459                     BSWAP_32(drr->drr_u.drr_object.
2460                         drr_bonuslen);
2461             }
2462             (void) recv_read(hdl, fd, buf,

```

```

2463         P2ROUNDUP(drr->drr_u.drr_object.drr_bonuslen, 8),
2464         B_FALSE, NULL);
2465         break;
2467     case DRR_WRITE:
2468         if (byteswap) {
2469             drr->drr_u.drr_write.drr_length =
2470                 BSWAP_64(drr->drr_u.drr_write.drr_length);
2471         }
2472         (void) recv_read(hdl, fd, buf,
2473             drr->drr_u.drr_write.drr_length, B_FALSE, NULL);
2474         break;
2475     case DRR_SPILL:
2476         if (byteswap) {
2477             drr->drr_u.drr_write.drr_length =
2478                 BSWAP_64(drr->drr_u.drr_spill.drr_length);
2479         }
2480         (void) recv_read(hdl, fd, buf,
2481             drr->drr_u.drr_spill.drr_length, B_FALSE, NULL);
2482         break;
2483     case DRR_WRITE_BYREF:
2484     case DRR_FREEOBJECTS:
2485     case DRR_FREE:
2486         break;
2488     default:
2489         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2490             "invalid record type"));
2491         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2492     }
2493 }
2495     free(buf);
2496     return (-1);
2497 }
2499 /*
2500  * Restores a backup of tosnap from the file descriptor specified by infd.
2501  */
2502 static int
2503 zfs_receive_one(libzfs_handle_t *hdl, int infd, const char *tosnap,
2504     recvflags_t *flags, dmuf_replay_record_t *drr,
2505     dmuf_replay_record_t *drr_noswap, const char *sendfs,
2506     nvlist_t *stream_nv, avl_tree_t *stream_avl, char **top_zfs, int cleanup_fd,
2507     uint64_t *action_handlep)
2508 {
2509     zfs_cmd_t zc = { 0 };
2510     time_t begin_time;
2511     int ioctl_err, ioctl_errno, err;
2512     char *cp;
2513     struct drr_begin *drrb = &drr->drr_u.drr_begin;
2514     char errbuf[1024];
2515     char prop_errbuf[1024];
2516     const char *chopprefix;
2517     boolean_t newfs = B_FALSE;
2518     boolean_t stream_wantsnewfs;
2519     uint64_t parent_snapguid = 0;
2520     prop_changelist_t *clp = NULL;
2521     nvlist_t *snapprops_nvlist = NULL;
2522     zprop_errflags_t prop_errflags;
2523     boolean_t recursive;
2525     begin_time = time(NULL);
2527     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2528         "cannot receive:"));

```

```

2530     recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
2531     ENOENT);

2533     if (stream_avl != NULL) {
2534         char *snapname;
2535         nvlist_t *fs = fsavl_find(stream_avl, drrb->drr_toguid,
2536         &snapname);
2537         nvlist_t *props;
2538         int ret;

2540         (void) nvlist_lookup_uint64(fs, "parentfromsnap",
2541         &parent_snapguid);
2542         err = nvlist_lookup_nvlist(fs, "props", &props);
2543         if (err)
2544             VERIFY(0 == nvlist_alloc(&props, NV_UNIQUE_NAME, 0));

2546         if (flags->canmountoff) {
2547             VERIFY(0 == nvlist_add_uint64(props,
2548             zfs_prop_to_name(ZFS_PROP_CANMOUNT), 0));
2549         }
2550         ret = zcmd_write_src_nvlist(hdl, &zc, props);
2551         if (err)
2552             nvlist_free(props);

2554         if (0 == nvlist_lookup_nvlist(fs, "snapprops", &props)) {
2555             VERIFY(0 == nvlist_lookup_nvlist(props,
2556             snapname, &snapprops_nvlist));
2557         }

2559         if (ret != 0)
2560             return (-1);
2561     }

2563     cp = NULL;

2565     /*
2566     * Determine how much of the snapshot name stored in the stream
2567     * we are going to tack on to the name they specified on the
2568     * command line, and how much we are going to chop off.
2569     *
2570     * If they specified a snapshot, chop the entire name stored in
2571     * the stream.
2572     */
2573     if (flags->istail) {
2574         /*
2575         * A filesystem was specified with -e. We want to tack on only
2576         * the tail of the sent snapshot path.
2577         */
2578         if (strchr(tosnap, '@')) {
2579             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
2580             "argument - snapshot not allowed with -e"));
2581             return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2582         }

2584         chopprefix = strrchr(sendfs, '/');

2586         if (chopprefix == NULL) {
2587             /*
2588             * The tail is the poolname, so we need to
2589             * prepend a path separator.
2590             */
2591             int len = strlen(drrb->drr_toname);
2592             cp = malloc(len + 2);
2593             cp[0] = '/';
2594             (void) strcpy(&cp[1], drrb->drr_toname);

```

```

2595         chopprefix = cp;
2596     } else {
2597         chopprefix = drrb->drr_toname + (chopprefix - sendfs);
2598     }
2599     } else if (flags->isprefix) {
2600         /*
2601         * A filesystem was specified with -d. We want to tack on
2602         * everything but the first element of the sent snapshot path
2603         * (all but the pool name).
2604         */
2605         if (strchr(tosnap, '@')) {
2606             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
2607             "argument - snapshot not allowed with -d"));
2608             return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2609         }

2611         chopprefix = strchr(drrb->drr_toname, '/');
2612         if (chopprefix == NULL)
2613             chopprefix = strchr(drrb->drr_toname, '@');
2614     } else if (strchr(tosnap, '@') == NULL) {
2615         /*
2616         * If a filesystem was specified without -d or -e, we want to
2617         * tack on everything after the fs specified by 'zfs send'.
2618         */
2619         chopprefix = drrb->drr_toname + strlen(sendfs);
2620     } else {
2621         /* A snapshot was specified as an exact path (no -d or -e). */
2622         if (recursive) {
2623             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2624             "cannot specify snapshot name for multi-snapshot "
2625             "stream"));
2626             return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2627         }
2628         chopprefix = drrb->drr_toname + strlen(drrb->drr_toname);
2629     }

2631     ASSERT(strstr(drrb->drr_toname, sendfs) == drrb->drr_toname);
2632     ASSERT(chopprefix > drrb->drr_toname);
2633     ASSERT(chopprefix <= drrb->drr_toname + strlen(drrb->drr_toname));
2634     ASSERT(chopprefix[0] == '/' || chopprefix[0] == '@' ||
2635     chopprefix[0] == '\0');

2637     /*
2638     * Determine name of destination snapshot, store in zc_value.
2639     */
2640     (void) strcpy(zc.zc_value, tosnap);
2641     (void) strncat(zc.zc_value, chopprefix, sizeof (zc.zc_value));
2642     free(cp);
2643     if (!zfs_name_valid(zc.zc_value, ZFS_TYPE_SNAPSHOT)) {
2644         zcmd_free_nvlists(&zc);
2645         return (zfs_error(hdl, EZFS_INVALIDNAME, errbuf));
2646     }

2648     /*
2649     * Determine the name of the origin snapshot, store in zc_string.
2650     */
2651     if (drrb->drr_flags & DRR_FLAG_CLONE) {
2652         if (guid_to_name(hdl, zc.zc_value,
2653         drrb->drr_fromguid, zc.zc_string) != 0) {
2654             zcmd_free_nvlists(&zc);
2655             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2656             "local origin for clone %s does not exist"),
2657             zc.zc_value);
2658             return (zfs_error(hdl, EZFS_NOENT, errbuf));
2659         }
2660     }
2661     if (flags->verbose)

```

```

2661         (void) printf("found clone origin %s\n", zc.zc_string);
2662     }

2664     stream_wantsnewfs = (drrb->drr_fromguid == NULL ||
2665         (drrb->drr_flags & DRR_FLAG_CLONE));

2667     if (stream_wantsnewfs) {
2668         /*
2669          * if the parent fs does not exist, look for it based on
2670          * the parent snap GUID
2671          */
2672         (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
2673             "cannot receive new filesystem stream"));

2675         (void) strcpy(zc.zc_name, zc.zc_value);
2676         cp = strrchr(zc.zc_name, '/');
2677         if (cp)
2678             *cp = '\0';
2679         if (cp &&
2680             !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2681             char suffix[ZFS_MAXNAMELEN];
2682             (void) strcpy(suffix, strrchr(zc.zc_value, '/'));
2683             if (guid_to_name(hdl, zc.zc_name, parent_snapguid,
2684                 zc.zc_value) == 0) {
2685                 *strchr(zc.zc_value, '@') = '\0';
2686                 (void) strcat(zc.zc_value, suffix);
2687             }
2688         }
2689     } else {
2690         /*
2691          * if the fs does not exist, look for it based on the
2692          * fromsnap GUID
2693          */
2694         (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
2695             "cannot receive incremental stream"));

2697         (void) strcpy(zc.zc_name, zc.zc_value);
2698         *strchr(zc.zc_name, '@') = '\0';

2700         /*
2701          * If the exact receive path was specified and this is the
2702          * topmost path in the stream, then if the fs does not exist we
2703          * should look no further.
2704          */
2705         if ((!flags->isprefix || (*(chopprefix = drrb->drr_toname +
2706             strlen(sendfs)) != '\0' && *chopprefix != '@')) &&
2707             !zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2708             char snap[ZFS_MAXNAMELEN];
2709             (void) strcpy(snap, strchr(zc.zc_value, '@'));
2710             if (guid_to_name(hdl, zc.zc_name, drrb->drr_fromguid,
2711                 zc.zc_value) == 0) {
2712                 *strchr(zc.zc_value, '@') = '\0';
2713                 (void) strcat(zc.zc_value, snap);
2714             }
2715         }
2716     }

2718     (void) strcpy(zc.zc_name, zc.zc_value);
2719     *strchr(zc.zc_name, '@') = '\0';

2721     if (zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2722         zfs_handle_t *zhp;

2724         /*
2725          * Destination fs exists. Therefore this should either
2726          * be an incremental, or the stream specifies a new fs

```

```

2727         * (full stream or clone) and they want us to blow it
2728         * away (and have therefore specified -F and removed any
2729         * snapshots).
2730         */
2731         if (stream_wantsnewfs) {
2732             if (!flags->force) {
2733                 zcmd_free_nvlists(&zc);
2734                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2735                     "destination '%s' exists\n"
2736                     "must specify -F to overwrite it"),
2737                     zc.zc_name);
2738                 return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2739             }
2740             if (ioctl(hdl->libzfs_fd, ZFS_IOC_SNAPSHOT_LIST_NEXT,
2741                 &zc) == 0) {
2742                 zcmd_free_nvlists(&zc);
2743                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2744                     "destination has snapshots (eg. %s)\n"
2745                     "must destroy them to overwrite it"),
2746                     zc.zc_name);
2747                 return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2748             }
2749         }

2751         if ((zhp = zfs_open(hdl, zc.zc_name,
2752             ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME)) == NULL) {
2753             zcmd_free_nvlists(&zc);
2754             return (-1);
2755         }

2757         if (stream_wantsnewfs &&
2758             zhp->zfs_dmustats.dds_origin[0]) {
2759             zcmd_free_nvlists(&zc);
2760             zfs_close(zhp);
2761             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2762                 "destination '%s' is a clone\n"
2763                 "must destroy it to overwrite it"),
2764                 zc.zc_name);
2765             return (zfs_error(hdl, EZFS_EXISTS, errbuf));
2766         }

2768         if (!flags->dryrun && zhp->zfs_type == ZFS_TYPE_FILESYSTEM &&
2769             stream_wantsnewfs) {
2770             /* We can't do online recv in this case */
2771             clp = changelist_gather(zhp, ZFS_PROP_NAME, 0, 0);
2772             if (clp == NULL) {
2773                 zfs_close(zhp);
2774                 zcmd_free_nvlists(&zc);
2775                 return (-1);
2776             }
2777             if (changelist_prefix(clp) != 0) {
2778                 changelist_free(clp);
2779                 zfs_close(zhp);
2780                 zcmd_free_nvlists(&zc);
2781                 return (-1);
2782             }
2783         }
2784         zfs_close(zhp);
2785     } else {
2786         /*
2787          * Destination filesystem does not exist. Therefore we better
2788          * be creating a new filesystem (either from a full backup, or
2789          * a clone). It would therefore be invalid if the user
2790          * specified only the pool name (i.e. if the destination name
2791          * contained no slash character).
2792         */

```

```

2793     if (!stream_wantsnewsfs ||
2794         (cp = strrchr(zc.zc_name, '/') == NULL) {
2795         zcmd_free_nvlists(&zc);
2796         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2797             "destination '%s' does not exist"), zc.zc_name);
2798         return (zfs_error(hdl, EZFS_NOENT, errbuf));
2799     }
2800
2801     /*
2802     * Trim off the final dataset component so we perform the
2803     * recvbackup ioctl to the filesystems's parent.
2804     */
2805     *cp = '\0';
2806
2807     if (flags->isprefix && !flags->istail && !flags->dryrun &&
2808         create_parents(hdl, zc.zc_value, strlen(tosnap)) != 0) {
2809         zcmd_free_nvlists(&zc);
2810         return (zfs_error(hdl, EZFS_BADRESTORE, errbuf));
2811     }
2812
2813     newfs = B_TRUE;
2814 }
2815
2816 zc.zc_begin_record = drr_noswap->drr_u.drr_begin;
2817 zc.zc_cookie = infd;
2818 zc.zc_guid = flags->force;
2819 if (flags->verbose) {
2820     (void) printf("%s %s stream of %s into %s\n",
2821         flags->dryrun ? "would receive" : "receiving",
2822         drrb->drr_fromguid ? "incremental" : "full",
2823         drrb->drr_toname, zc.zc_value);
2824     (void) fflush(stdout);
2825 }
2826
2827 if (flags->dryrun) {
2828     zcmd_free_nvlists(&zc);
2829     return (recv_skip(hdl, infd, flags->byteswap));
2830 }
2831
2832 zc.zc_nvlist_dst = (uint64_t)(uintptr_t)prop_errbuf;
2833 zc.zc_nvlist_dst_size = sizeof (prop_errbuf);
2834 zc.zc_cleanup_fd = cleanup_fd;
2835 zc.zc_action_handle = *action_handlep;
2836
2837 err = ioctl_err = zfs_ioctl(hdl, ZFS_IOC_RECV, &zc);
2838 ioctl_errno = errno;
2839 prop_errflags = (zprop_errflags_t)zc.zc_obj;
2840
2841 if (err == 0) {
2842     nvlist_t *prop_errors;
2843     VERIFY(0 == nvlist_unpack((void *) (uintptr_t)zc.zc_nvlist_dst,
2844         zc.zc_nvlist_dst_size, &prop_errors, 0));
2845
2846     nvpair_t *prop_err = NULL;
2847
2848     while ((prop_err = nvlist_next_nvpair(prop_errors,
2849         prop_err)) != NULL) {
2850         char tbuf[1024];
2851         zfs_prop_t prop;
2852         int intval;
2853
2854         prop = zfs_name_to_prop(nvpair_name(prop_err));
2855         (void) nvpair_value_int32(prop_err, &intval);
2856         if (strcmp(nvpair_name(prop_err),
2857             ZPROP_N_MORE_ERRORS) == 0) {
2858             trunc_prop_errs(intval);

```

```

2859         break;
2860     } else {
2861         (void) snprintf(tbuf, sizeof (tbuf),
2862             dgettext(TEXT_DOMAIN,
2863                 "cannot receive %s property on %s"),
2864             nvpair_name(prop_err), zc.zc_name);
2865         zfs_setprop_error(hdl, prop, intval, tbuf);
2866     }
2867 }
2868 nvlist_free(prop_errors);
2869 }
2870
2871 zc.zc_nvlist_dst = 0;
2872 zc.zc_nvlist_dst_size = 0;
2873 zcmd_free_nvlists(&zc);
2874
2875 if (err == 0 && snapprops_nvlist) {
2876     zfs_cmd_t zc2 = { 0 };
2877
2878     (void) strcpy(zc2.zc_name, zc.zc_value);
2879     zc2.zc_cookie = B_TRUE; /* received */
2880     if (zcmd_write_src_nvlist(hdl, &zc2, snapprops_nvlist) == 0) {
2881         (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc2);
2882         zcmd_free_nvlists(&zc2);
2883     }
2884 }
2885
2886 if (err && (ioctl_errno == ENOENT || ioctl_errno == EEXIST)) {
2887     /*
2888     * It may be that this snapshot already exists,
2889     * in which case we want to consume & ignore it
2890     * rather than failing.
2891     */
2892     avl_tree_t *local_avl;
2893     nvlist_t *local_nv, *fs;
2894     cp = strchr(zc.zc_value, '@');
2895
2896     /*
2897     * XXX Do this faster by just iterating over snaps in
2898     * this fs. Also if zc_value does not exist, we will
2899     * get a strange "does not exist" error message.
2900     */
2901     *cp = '\0';
2902     if (gather_nvlist(hdl, zc.zc_value, NULL, NULL, B_FALSE,
2903         &local_nv, &local_avl) == 0) {
2904         *cp = '@';
2905         fs = fsavl_find(local_avl, drrb->drr_toguid, NULL);
2906         fsavl_destroy(local_avl);
2907         nvlist_free(local_nv);
2908
2909         if (fs != NULL) {
2910             if (flags->verbose) {
2911                 (void) printf("snap %s already exists; "
2912                     "ignoring\n", zc.zc_value);
2913             }
2914             err = ioctl_err = recv_skip(hdl, infd,
2915                 flags->byteswap);
2916         }
2917     }
2918     *cp = '@';
2919 }
2920
2921 if (ioctl_err != 0) {
2922     switch (ioctl_errno) {
2923     case ENODEV:
2924         cp = strchr(zc.zc_value, '@');

```

```

2925         *cp = '\0';
2926         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2927             "most recent snapshot of %s does not\n"
2928             "match incremental source"), zc.zc_value);
2929         (void) zfs_error(hdl, EZFS_BADRESTORE, errbuf);
2930         *cp = '@';
2931         break;
2932     case ETXTBSY:
2933         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2934             "destination %s has been modified\n"
2935             "since most recent snapshot"), zc.zc_name);
2936         (void) zfs_error(hdl, EZFS_BADRESTORE, errbuf);
2937         break;
2938     case EEXIST:
2939         cp = strchr(zc.zc_value, '@');
2940         if (newfs) {
2941             /* it's the containing fs that exists */
2942             *cp = '\0';
2943         }
2944         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2945             "destination already exists"));
2946         (void) zfs_error_fmt(hdl, EZFS_EXISTS,
2947             dgettext(TEXT_DOMAIN, "cannot restore to %s"),
2948             zc.zc_value);
2949         *cp = '@';
2950         break;
2951     case EINVAL:
2952         (void) zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2953         break;
2954     case ECKSUM:
2955         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2956             "invalid stream (checksum mismatch)"));
2957         (void) zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2958         break;
2959     case ENOTSUP:
2960         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2961             "pool must be upgraded to receive this stream.));
2962         (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
2963         break;
2964     case EDQUOT:
2965         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2966             "destination %s space quota exceeded"), zc.zc_name);
2967         (void) zfs_error(hdl, EZFS_NOSPC, errbuf);
2968         break;
2969     default:
2970         (void) zfs_standard_error(hdl, ioctl_errno, errbuf);
2971     }
2972 }
2973
2974 /*
2975  * Mount the target filesystem (if created). Also mount any
2976  * children of the target filesystem if we did a replication
2977  * receive (indicated by stream_avl being non-NULL).
2978  */
2979 cp = strchr(zc.zc_value, '@');
2980 if (cp && (ioctl_err == 0 || !newfs)) {
2981     zfs_handle_t *h;
2982
2983     *cp = '\0';
2984     h = zfs_open(hdl, zc.zc_value,
2985         ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
2986     if (h != NULL) {
2987         if (h->zfs_type == ZFS_TYPE_VOLUME) {
2988             *cp = '@';
2989         } else if (newfs || stream_avl) {
2990             /*

```

```

2991         * Track the first/top of hierarchy fs,
2992         * for mounting and sharing later.
2993         */
2994         if (top_zfs && *top_zfs == NULL)
2995             *top_zfs = zfs_strdup(hdl, zc.zc_value);
2996     }
2997     zfs_close(h);
2998 }
2999 *cp = '@';
3000 }
3001
3002 if (clp) {
3003     err |= changelist_postfix(clp);
3004     changelist_free(clp);
3005 }
3006
3007 if (prop_errflags & ZPROP_ERR_NOCLEAR) {
3008     (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "Warning: "
3009         "failed to clear unreceived properties on %s"),
3010         zc.zc_name);
3011     (void) fprintf(stderr, "\n");
3012 }
3013 if (prop_errflags & ZPROP_ERR_NORESTORE) {
3014     (void) fprintf(stderr, dgettext(TEXT_DOMAIN, "Warning: "
3015         "failed to restore original properties on %s"),
3016         zc.zc_name);
3017     (void) fprintf(stderr, "\n");
3018 }
3019
3020 if (err || ioctl_err)
3021     return (-1);
3022
3023 *action_handlep = zc.zc_action_handle;
3024
3025 if (flags->verbose) {
3026     char buf1[64];
3027     char buf2[64];
3028     uint64_t bytes = zc.zc_cookie;
3029     time_t delta = time(NULL) - begin_time;
3030     if (delta == 0)
3031         delta = 1;
3032     zfs_nicenumber(bytes, buf1, sizeof(buf1));
3033     zfs_nicenumber(bytes/delta, buf2, sizeof(buf1));
3034
3035     (void) printf("received %sB stream in %lu seconds (%sB/sec)\n",
3036         buf1, delta, buf2);
3037 }
3038
3039 return (0);
3040 }
3041
3042 static int
3043 zfs_receive_impl(libzfs_handle_t *hdl, const char *tosnap, recvflags_t *flags,
3044     int infd, const char *sendfs, nvlist_t *stream_nv, avl_tree_t *stream_avl,
3045     char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
3046 {
3047     int err;
3048     dmu_replay_record_t drr, drr_noswap;
3049     struct drr_begin *drrb = &drr.drr_u.drr_begin;
3050     char errbuf[1024];
3051     zio_cksum_t zcksum = { 0 };
3052     uint64_t featureflags;
3053     int hdrtype;
3054
3055     (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
3056         "cannot receive"));

```

```

3058     if (flags->isprefix &&
3059         !zfs_dataset_exists(hdl, tosnap, ZFS_TYPE_DATASET)) {
3060         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "specified fs "
3061             "%s does not exist"), tosnap);
3062         return (zfs_error(hdl, EZFS_NOENT, errbuf));
3063     }

3065     /* read in the BEGIN record */
3066     if (0 != (err = recv_read(hdl, infd, &drr, sizeof (drr), B_FALSE,
3067         &zcksum)))
3068         return (err);

3070     if (drr.drr_type == DRR_END || drr.drr_type == BSWAP_32(DRR_END)) {
3071         /* It's the double end record at the end of a package */
3072         return (ENODATA);
3073     }

3075     /* the kernel needs the non-byteswapped begin record */
3076     drr_noswap = drr;

3078     flags->byteswap = B_FALSE;
3079     if (drrb->drr_magic == BSWAP_64(DMU_BACKUP_MAGIC)) {
3080         /*
3081          * We computed the checksum in the wrong byteorder in
3082          * recv_read() above; do it again correctly.
3083          */
3084         bzero(&zcksum, sizeof (zio_cksum_t));
3085         fletcher_4_incremental_byteswap(&drr, sizeof (drr), &zcksum);
3086         flags->byteswap = B_TRUE;

3088         drr.drr_type = BSWAP_32(drr.drr_type);
3089         drr.drr_payloadlen = BSWAP_32(drr.drr_payloadlen);
3090         drrb->drr_magic = BSWAP_64(drrb->drr_magic);
3091         drrb->drr_versioninfo = BSWAP_64(drrb->drr_versioninfo);
3092         drrb->drr_creation_time = BSWAP_64(drrb->drr_creation_time);
3093         drrb->drr_type = BSWAP_32(drrb->drr_type);
3094         drrb->drr_flags = BSWAP_32(drrb->drr_flags);
3095         drrb->drr_toguid = BSWAP_64(drrb->drr_toguid);
3096         drrb->drr_fromguid = BSWAP_64(drrb->drr_fromguid);
3097     }

3099     if (drrb->drr_magic != DMU_BACKUP_MAGIC || drr.drr_type != DRR_BEGIN) {
3100         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
3101             "stream (bad magic number)"));
3102         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
3103     }

3105     featureflags = DMU_GET_FEATUREFLAGS(drrb->drr_versioninfo);
3106     hdrtype = DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo);

3108     if (!DMU_STREAM_SUPPORTED(featureflags) ||
3109         (hdrtype != DMU_SUBSTREAM && hdrtype != DMU_COMPOUNDSTREAM)) {
3110         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
3111             "stream has unsupported feature, feature flags = %lx"),
3112             featureflags);
3113         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
3114     }

3116     if (strchr(drrb->drr_toname, '@') == NULL) {
3117         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
3118             "stream (bad snapshot name)"));
3119         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
3120     }

3122     if (DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) == DMU_SUBSTREAM) {

```

```

3123         char nonpackage_sendfs[ZFS_MAXNAMELEN];
3124         if (sendfs == NULL) {
3125             /*
3126              * We were not called from zfs_receive_package(). Get
3127              * the fs specified by 'zfs send'.
3128              */
3129             char *cp;
3130             (void) strlcpy(nonpackage_sendfs,
3131                 drr.drr_u.drr_begin.drr_toname, ZFS_MAXNAMELEN);
3132             if ((cp = strchr(nonpackage_sendfs, '@')) != NULL)
3133                 *cp = '\0';
3134             sendfs = nonpackage_sendfs;
3135         }
3136         return (zfs_receive_one(hdl, infd, tosnap, flags,
3137             &drr, &drr_noswap, sendfs, stream_nv, stream_avl,
3138             top_zfs, cleanup_fd, action_handle));
3139     } else {
3140         assert(DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo) ==
3141             DMU_COMPOUNDSTREAM);
3142         return (zfs_receive_package(hdl, infd, tosnap, flags,
3143             &drr, &zcksum, top_zfs, cleanup_fd, action_handle));
3144     }
3145 }

3147 /*
3148  * Restores a backup of tosnap from the file descriptor specified by infd.
3149  * Return 0 on total success, -2 if some things couldn't be
3150  * destroyed/renamed/promoted, -1 if some things couldn't be received.
3151  * (-1 will override -2).
3152  */
3153 int
3154 zfs_receive(libzfs_handle_t *hdl, const char *tosnap, recvflags_t *flags,
3155     int infd, avl_tree_t *stream_avl)
3156 {
3157     char *top_zfs = NULL;
3158     int err;
3159     int cleanup_fd;
3160     uint64_t action_handle = 0;

3162     cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
3163     VERIFY(cleanup_fd >= 0);

3165     err = zfs_receive_impl(hdl, tosnap, flags, infd, NULL, NULL,
3166         stream_avl, &top_zfs, cleanup_fd, &action_handle);

3168     VERIFY(0 == close(cleanup_fd));

3170     if (err == 0 && !flags->nomount && top_zfs) {
3171         zfs_handle_t *zhp;
3172         prop_changelist_t *clp;

3174         zhp = zfs_open(hdl, top_zfs, ZFS_TYPE_FILESYSTEM);
3175         if (zhp != NULL) {
3176             clp = changelist_gather(zhp, ZFS_PROP_MOUNTPOINT,
3177                 CL_GATHER_MOUNT_ALWAYS, 0);
3178             zfs_close(zhp);
3179             if (clp != NULL) {
3180                 /* mount and share received datasets */
3181                 err = changelist_postfix(clp);
3182                 changelist_free(clp);
3183             }
3184         }
3185         if (zhp == NULL || clp == NULL || err)
3186             err = -1;
3187     }
3188     if (top_zfs)

```

```
3189         free(top_zfs);
3191     return (err);
3192 }
```

```

*****
16856 Wed May 1 01:43:41 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

334 /*
335 * Create "user holds" on snapshots. If there is a hold on a snapshot,
336 * the snapshot can not be destroyed. (However, it can be marked for deletion
337 * by lzc_destroy_snaps(defer=B_TRUE).)
338 *
339 * The keys in the nvlist are snapshot names.
340 * The snapshots must all be in the same pool.
341 * The value is the name of the hold (string type).
342 *
343 * If cleanup_fd is not -1, it must be the result of open("/dev/zfs", O_EXCL).
344 * In this case, when the cleanup_fd is closed (including on process
345 * termination), the holds will be released. If the system is shut down
346 * uncleanly, the holds will be released when the pool is next opened
347 * or imported.
348 *
349 * Holds for snapshots which don't exist will be skipped and have an entry
350 * added to errlist, but will not cause an overall failure, except in the
351 * case that all holds were skipped.
352 *
353 * The return value will be 0 if the nvl holds was empty or all holds, for
354 * snapshots that existed, were successfully created and at least one hold
355 * was created.
356 *
357 * If none of the snapshots for the requested holds existed ENOENT will be
358 * returned.
359 *
360 * Otherwise the return value will be the errno of a (unspecified) hold that
361 * failed, no holds will be created.
362 *
363 * In all cases the errlist will have an entry for each hold that failed
364 * (name = snapshot), with its value being the error code (int32).
365 * The return value will be 0 if all holds were created. Otherwise the return
366 * value will be the errno of a (unspecified) hold that failed, no holds will
367 * be created, and the errlist will have an entry for each hold that
368 * failed (name = snapshot). The value in the errlist will be the error
369 * code (int32).
370 */
371 int
372 lzc_hold(nvlist_t *holds, int cleanup_fd, nvlist_t **errlist)
373 {
374     char pool[MAXNAMELEN];
375     nvlist_t *args;
376     nvpair_t *elem;
377     int error;

378     /* determine the pool name */
379     elem = nvlist_next_nvpair(holds, NULL);
380     if (elem == NULL)
381         return (0);
382     (void) strncpy(pool, nvpair_name(elem), sizeof (pool));
383     pool[strlen(pool, "/@")] = '\0';

384     args = fnvlist_alloc();
385     fnvlist_add_nvlist(args, "holds", holds);
386     if (cleanup_fd != -1)
387         fnvlist_add_int32(args, "cleanup_fd", cleanup_fd);

388     error = lzc_ioctl(ZFS_IOC_HOLD, pool, args, errlist);

```

```

387     nvlist_free(args);
388     return (error);
389 }

391 /*
392 * Release "user holds" on snapshots. If the snapshot has been marked for
393 * deferred destroy (by lzc_destroy_snaps(defer=B_TRUE)), it does not have
394 * any clones, and all the user holds are removed, then the snapshot will be
395 * destroyed.
396 *
397 * The keys in the nvlist are snapshot names.
398 * The snapshots must all be in the same pool.
399 * The value is a nvlist whose keys are the holds to remove.
400 *
401 * Holds which failed to release because they didn't exist will have an entry
402 * added to errlist, but will not cause an overall failure.
403 *
404 * The return value will be 0 if the nvl holds was empty or all holds, that
405 * existed, were successfully removed and at least one hold was removed.
406 *
407 * If none of the holds specified existed ENOENT will be returned.
408 *
409 * Otherwise the return value will be the errno of a (unspecified) hold that
410 * failed to release and no holds will be released.
411 *
412 * In all cases the errlist will have an entry for each hold that failed to
413 * release.
414 * The return value will be 0 if all holds were removed.
415 * Otherwise the return value will be the errno of a (unspecified) release
416 * that failed, no holds will be released, and the errlist will have an
417 * entry for each snapshot that has failed releases (name = snapshot).
418 * The value in the errlist will be the error code (int32) of a failed release.
419 */
420 int
421 lzc_release(nvlist_t *holds, nvlist_t **errlist)
422 {
423     char pool[MAXNAMELEN];
424     nvpair_t *elem;

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[strlen(pool, "/@")] = '\0';

431     return (lzc_ioctl(ZFS_IOC_RELEASE, pool, holds, errlist));
432 }
unchanged_portion_omitted

```



```

*****
29735 Wed May 1 01:43:41 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 }

```

```

*****
18953 Wed May 1 01:43:41 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_tmpholds;
43 #endif /* ! codereview */
44     nvlist_t *dduha_errlist;
45     minor_t dduha_minor;
46     boolean_t dduha_holds_created;
47 #endif /* ! codereview */
48 } dsl_dataset_user_hold_arg_t;

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

```

```

62     if (strlen(htag) > MAXNAMELEN)
63         return (E2BIG);
64     /* Tempholds have a more restricted length */
65     if (temphold && strlen(htag) + MAX_TAG_PREFIX_LEN >= MAXNAMELEN)
66         return (E2BIG);

68     /* tags must be unique (if ds already exists) */
69     if (ds != NULL) {
70         mutex_enter(&ds->ds_lock);
71         if (ds->ds_phys->ds_userrefs_obj != 0) {
72             uint64_t value;
73             error = zap_lookup(mos, ds->ds_phys->ds_userrefs_obj,
74                 htag, 8, 1, &value);
75             if (error == 0)
76                 error = SET_ERROR(EEXIST);
77             else if (error == ENOENT)
78                 error = 0;
79         }
80         mutex_exit(&ds->ds_lock);
81     }

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;
92     int rv = 0;
93     boolean_t holds_possible;
94 #endif /* ! codereview */

96     if (spa_version(dp->dp_spa) < SPA_VERSION_USERREFS)
97         return (SET_ERROR(ENOTSUP));

99     holds_possible = B_FALSE;

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

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

112         if (error == 0)
113             error = nvpair_value_string(pair, &htag);
114         if (error == 0) {
115             error = dsl_dataset_hold(dp,
116                 nvpair_name(pair), FTAG, &ds);

118             if (error == ENOENT) {
119                 /*
120                  * We register ENOENT errors so they can be
121                  * correctly reported if needed, such as when
122                  * all holds fail.
123                  */
124                 if (dduha->dduha_errlist != NULL) {
125                     fnvlist_add_int32(dduha->dduha_errlist,
126                         nvpair_name(pair), error);

```

```

127         }
128         continue;
129     }
130 #endif /* ! codereview */
131     }
132     if (error == 0) {
133         error = dsl_dataset_user_hold_check_one(ds, htag,
134         dduha->dduha_minor != 0, tx);
135         dsl_dataset_rele(ds, FTAG);
136     }
137
138     if (error != 0) {
139         if (dduha->dduha_errlist != NULL) {
140             rv = error;
141             fnvlist_add_int32(dduha->dduha_errlist,
142             nvpair_name(pair), error);
143         }
144     } else {
145         holds_possible = B_TRUE;
146     }
147 #endif /* ! codereview */
148 }
149
150 /*
151  * Check that at least one hold will possibly be created,
152  * otherwise fail.
153  */
154 if (rv == 0 && !holds_possible)
155     rv = ENOENT;
156
157 #endif /* ! codereview */
158 return (rv);
159 }
160
161
162 static void
163 dsl_dataset_user_hold_sync_one_impl(dsl_dataset_user_hold_arg_t *dduha,
164 dsl_dataset_t *ds, const char *htag, minor_t minor, uint64_t now,
165 dmu_tx_t *tx)
166 {
167     void
168     dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
169     minor_t minor, uint64_t now, dmu_tx_t *tx)
170 {
171     dsl_pool_t *dp = ds->ds_dir->dd_pool;
172     objset_t *mos = dp->dp_meta_objset;
173     uint64_t zapobj;
174
175     mutex_enter(&ds->ds_lock);
176     if (ds->ds_phys->ds_userrefs_obj == 0) {
177         /*
178          * This is the first user hold for this dataset. Create
179          * the userrefs zap object.
180          */
181         dmu_buf_will_dirty(ds->ds_dbuf, tx);
182         zapobj = ds->ds_phys->ds_userrefs_obj =
183         zap_create(mos, DMU_OT_USERREFS, DMU_OT_NONE, 0, tx);
184     } else {
185         zapobj = ds->ds_phys->ds_userrefs_obj;
186     }
187     ds->ds_userrefs++;
188     mutex_exit(&ds->ds_lock);
189
190     VERIFY0(zap_add(mos, zapobj, htag, 8, 1, &now, tx));
191
192     if (minor != 0) {

```

```

189         char name[MAXNAMELEN];
190         nvlist_t *tags;
191
192 #endif /* ! codereview */
193     VERIFY0(dsl_pool_user_hold(dp, ds->ds_object,
194     htag, now, tx));
195     (void) snprintf(name, sizeof(name), "%llx",
196     (u_longlong_t)ds->ds_object);
197
198     if (nvlist_lookup_nvlist(dduha->dduha_tmpholds, name, &tags) !=
199     tags = fnvlist_alloc();
200     fnvlist_add_boolean(tags, htag);
201     fnvlist_add_nvlist(dduha->dduha_tmpholds, name, tags);
202     fnvlist_free(tags);
203     } else {
204         fnvlist_add_boolean(tags, htag);
205     }
206     dsl_register_onexit_hold_cleanup(ds, htag, minor);
207
208     spa_history_log_internal_ds(ds, "hold", tx,
209     "tag=%s temp=%d refs=%llu",
210     htag, minor != 0, ds->ds_userrefs);
211 }
212
213 typedef struct zfs_hold_cleanup_arg {
214     char zhca_spaname[MAXNAMELEN];
215     uint64_t zhca_spa_load_guid;
216     nvlist_t *zhca_holds;
217 } zfs_hold_cleanup_arg_t;
218
219 static void
220 dsl_dataset_user_release_onexit(void *arg)
221 {
222     zfs_hold_cleanup_arg_t *ca = (zfs_hold_cleanup_arg_t *)arg;
223     spa_t *spa;
224     int error;
225
226     error = spa_open(ca->zhca_spaname, &spa, FTAG);
227     if (error != 0) {
228         zfs_dbgmsg("couldn't release holds on pool=%s "
229         "because pool is no longer loaded",
230         ca->zhca_spaname);
231         return;
232     }
233     if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
234         zfs_dbgmsg("couldn't release holds on pool=%s "
235         "because pool is no longer loaded (guid doesn't match)",
236         ca->zhca_spaname);
237         spa_close(spa, FTAG);
238         return;
239     }
240
241     (void) dsl_dataset_user_release_tmp(spa_get_dsl(spa), ca->zhca_holds);
242     fnvlist_free(ca->zhca_holds);
243     kmem_free(ca, sizeof(zfs_hold_cleanup_arg_t));
244     spa_close(spa, FTAG);
245 }
246
247 static void
248 dsl_register_onexit_hold_cleanup(spa_t *spa, nvlist_t *holds, minor_t minor)
249 {
250     zfs_hold_cleanup_arg_t *ca;
251
252     if (minor == 0 || nvlist_next_nvpair(holds, NULL) == NULL) {
253         fnvlist_free(holds);

```

```

254         return;
255     }

257     ASSERT(spa != NULL);
258     ca = kmem_alloc(sizeof (*ca), KM_SLEEP);

260     (void) strcpy(ca->zhca_spaname, spa_name(spa),
261                 sizeof (ca->zhca_spaname));
262     ca->zhca_spa_load_guid = spa_load_guid(spa);
263     ca->zhca_holds = holds;
264     VERIFY0(zfs_onexit_add_cb(minor,
265                             dsl_dataset_user_release_onexit, ca, NULL));
266 }

268 void
269 dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
270                               minor_t minor, uint64_t now, dmu_tx_t *tx)
271 {
272     dsl_dataset_user_hold_arg_t dduha;

274     dduha.dduha_spa = NULL;
275     dduha.dduha_holds = NULL;
276     dduha.dduha_tmpholds = fnvlist_alloc();
277     dduha.dduha_errlist = NULL;
278     dduha.dduha_minor = minor;
279     dduha.dduha_holds_created = B_FALSE;

281     dsl_dataset_user_hold_sync_one_impl(&dduha, ds, htag, minor, now, tx);
282     dsl_register_onexit_hold_cleanup(dsl_dataset_get_spa(ds),
283                                     dduha.dduha_tmpholds, minor);
284 }

286 #endif /* ! codereview */
287 static void
288 dsl_dataset_user_hold_sync(void *arg, dmu_tx_t *tx)
289 {
290     dsl_dataset_user_hold_arg_t *dduha = arg;
291     dsl_pool_t *dp = dmu_tx_pool(tx);
292     nvpair_t *pair;
293     uint64_t now = gethrstime_sec();

295     for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;
296          pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {
297         dsl_dataset_t *ds;
298         char *name;
299         int error;

301         name = nvpair_name(pair);
302         error = dsl_dataset_hold(dp, name, FTAG, &ds);
303         if (error == 0) {
304             dsl_dataset_user_hold_sync_one_impl(dduha, ds,
305                                                 fnvpair_value_string(pair), dduha->dduha_minor,
306                                                 now, tx);
307             VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
308             dsl_dataset_user_hold_sync_one(ds, fnvpair_value_string(pair),
309                                           dduha->dduha_minor, now, tx);
310             dsl_dataset_rele(ds, FTAG);
311             dduha->dduha_holds_created = B_TRUE;
312         } else if (dduha->dduha_errlist != NULL) {
313             /*
314              * We register ENOENT errors so they can be correctly
315              * reported if needed, such as when all holds fail.
316              */
317             fnvlist_add_int32(dduha->dduha_errlist, name, error);
318         }
319     }
320 }

```

```

317     }
318     dduha->dduha_spa = dp->dp_spa;
319 #endif /* ! codereview */
320 }

322 /*
323  * The full semantics of this function are described in the comment above
324  * lzc_hold().
325  *
326  * To summarize:
327  * #endif /* ! codereview */
328  * holds is nvl of snapname -> holdname
329  * errlist will be filled in with snapname -> error
330  * 81 * if cleanup_minor is not 0, the holds will be temporary, cleaned up
331  * 82 * when the process exits.
332  *
333  * The snapshots must all be in the same pool.
334  *
335  * Holds for snapshots that don't exist will be skipped.
336  *
337  * If none of the snapshots for requested holds exist then ENOENT will be
338  * returned.
339  *
340  * If cleanup_minor is not 0, the holds will be temporary, which will be cleaned
341  * up when the process exits.
342  *
343  * On success all the holds, for snapshots that existed, will be created and 0
344  * will be returned.
345  *
346  * On failure no holds will be created, the errlist will be filled in,
347  * and an errno will returned.
348  *
349  * In all cases the errlist will contain entries for holds where the snapshot
350  * didn't exist.
351  * 84 * if any fails, all will fail.
352  */
353 int
354 dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor, nvlist_t *errlist)
355 {
356     dsl_dataset_user_hold_arg_t dduha;
357     nvpair_t *pair;
358     int ret;
359 #endif /* ! codereview */

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

365     dduha.dduha_spa = NULL;
366 #endif /* ! codereview */
367     dduha.dduha_holds = holds;
368     dduha.dduha_tmpholds = fnvlist_alloc();
369 #endif /* ! codereview */
370     dduha.dduha_errlist = errlist;
371     dduha.dduha_minor = cleanup_minor;
372     dduha.dduha_holds_created = B_FALSE;
373 #endif /* ! codereview */

375     ret = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
376                       dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds));
377     if (ret == 0) {
378         /* Check we created at least one hold. */
379         if (dduha.dduha_holds_created) {
380             dsl_register_onexit_hold_cleanup(dduha.dduha_spa,
381                                             dduha.dduha_tmpholds, cleanup_minor);
382         } else {

```

```

380         fnvlist_free(dduha.dduha_tmpholds);
381         ret = ENOENT;
382     }
383     } else {
384         fnvlist_free(dduha.dduha_tmpholds);
385     }
387     return (ret);
91     return (dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
92         dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds)));
388 }

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

393 #endif /* ! codereview */
394 typedef struct dsl_dataset_user_release_arg {
395     dsl_holdfunc_t *ddura_holdfunc;
396 #endif /* ! codereview */
397     nvlist_t *ddura_holds;
398     nvlist_t *ddura_todelete;
399     nvlist_t *ddura_errlist;
400     boolean_t ddura_holds_found;
401 #endif /* ! codereview */
402 } dsl_dataset_user_release_arg_t;

404 /* Place a dataset hold on the snapshot identified by passed dsobj string */
405 static
406 int dsl_dataset_hold_byobj(dsl_pool_t *dp, const char *dsobj, void *tag,
407     dsl_dataset_t **dsp)
408 {
409     return dsl_dataset_hold_obj(dp, strtonum(dsobj, NULL), tag, dsp);
410 }

412 #endif /* ! codereview */
413 static int
414 dsl_dataset_user_release_check_one(dsl_dataset_user_release_arg_t *ddura,
415     dsl_dataset_t *ds, nvlist_t *holds, boolean_t *todelete)
416 {
95     dsl_dataset_user_release_check_one(dsl_dataset_t *ds,
96         nvlist_t *holds, boolean_t *todelete)
417 {
418     uint64_t zapobj;
419     nvpair_t *pair;
420     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
421     int error;
422     int numholds = 0;
423     int ret;
424 #endif /* ! codereview */

425     *todelete = B_FALSE;
426     ret = 0;
427 #endif /* ! codereview */

429     if (!dsl_dataset_is_snapshot(ds))
430         return (SET_ERROR(EINVAL));

432     zapobj = ds->ds_phys->ds_userrefs_obj;
433     if (zapobj == 0)
434         return (SET_ERROR(ESRCH));

436     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
437         pair = nvlist_next_nvpair(holds, pair)) {
103         /* Make sure the hold exists */
438         uint64_t tmp;
440 #endif /* ! codereview */

```

```

441         error = zap_lookup(mos, zapobj, nvpair_name(pair), 8, 1, &tmp);
442         /* Non-existent holds aren't always fatal. */
443         if (error == ENOENT) {
444             ret = error;
445             continue;
446         }
105         if (error == ENOENT)
106             error = SET_ERROR(ESRCH);
447         if (error != 0)
448             return (error);
449         numholds++;
450     }

452     if (DS_IS_DEFER_DESTROY(ds) && ds->ds_phys->ds_num_children == 1 &&
453         ds->ds_userrefs == numholds) {
454         /* we need to destroy the snapshot as well */

456         if (dsl_dataset_long_held(ds))
457             return (SET_ERROR(EBUSY));
458         *todelete = B_TRUE;
459     }

461     if (numholds != 0)
462         ddura->ddura_holds_found = B_TRUE;

464     return (ret);
120     return (0);
465 }

467 static int
468 dsl_dataset_user_release_check(void *arg, dmu_tx_t *tx)
469 {
470     dsl_dataset_user_release_arg_t *ddura = arg;
471     dsl_holdfunc_t *holdfunc = ddura->ddura_holdfunc;
472 #endif /* ! codereview */
473     dsl_pool_t *dp = dmu_tx_pool(tx);
474     nvpair_t *pair;
475     int rv = 0;

477     if (!dmu_tx_is_syncing(tx))
478         return (0);

480     for (pair = nvlist_next_nvpair(ddura->ddura_holds, NULL); pair != NULL;
481         pair = nvlist_next_nvpair(ddura->ddura_holds, pair)) {
482         const char *name = nvpair_name(pair);
483         int error;
484         dsl_dataset_t *ds;
485         nvlist_t *holds;

487         error = nvpair_value_nvlist(pair, &holds);
488         if (error != 0)
489             return (SET_ERROR(EINVAL));

491         error = holdfunc(dp, name, FTAG, &ds);
127         error = dsl_dataset_hold(dp, name, FTAG, &ds);
492         if (error == 0) {
493             boolean_t deleteme;
494             error = dsl_dataset_user_release_check_one(ddura, ds,
130                 error = dsl_dataset_user_release_check_one(ds,
495                     holds, &deleteme);
496             /*
497              * Don't check for error == 0 as deleteme is only set
498              * to B_TRUE if it's correct to do so despite the error
499              * e.g. ENOENT.
500              */
501             if (deleteme) {

```

```

132         if (error == 0 && deleteme) {
502             fnvlist_add_boolean(ddura->ddura_todelete,
503                 name);
504         }
505         dsl_dataset_rele(ds, FTAG);
506     }
507     if (error != 0) {
508         if (ddura->ddura_errlist != NULL) {
509             fnvlist_add_int32(ddura->ddura_errlist,
510                 name, error);
511         }
512         /* Non-existent holds aren't always fatal. */
513         if (error != ENOENT)
514 #endif /* ! codereview */
515             rv = error;
516     }
517 }
518
519 /*
520  * None of the specified holds existed so avoid the overhead of a sync
521  * and return ENOENT.
522  */
523 if (rv == 0 && lddura->ddura_holds_found)
524     rv = ENOENT;
525
526 #endif /* ! codereview */
527 return (rv);
528 }
529
530 static void
531 dsl_dataset_user_release_sync_one(dsl_dataset_user_release_arg_t *ddura,
532     dsl_dataset_t *ds, nvlist_t *holds, dmu_tx_t *tx)
533 dsl_dataset_user_release_sync_one(dsl_dataset_t *ds, nvlist_t *holds,
534     dmu_tx_t *tx)
535 {
536     dsl_pool_t *dp = ds->ds_dir->dd_pool;
537     objset_t *mos = dp->dp_meta_objset;
538     uint64_t zapobj;
539     int error;
540     nvpair_t *pair;
541
542     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
543         pair = nvlist_next_nvpair(holds, pair)) {
544         uint64_t zapobj;
545         int error;
546         char *name;
547
548         name = nvpair_name(pair);
549
550         /* Remove temporary hold if one exists. */
551         error = dsl_pool_user_release(dp, ds->ds_object, name, tx);
552         ds->ds_userrefs--;
553         error = dsl_pool_user_release(dp, ds->ds_object,
554             nvpair_name(pair), tx);
555         VERIFY(error == 0 || error == ENOENT);
556
557         /* Remove user hold if one exists. */
558 #endif /* ! codereview */
559         zapobj = ds->ds_phys->ds_userrefs_obj;
560         error = zap_remove(mos, zapobj, name, tx);
561         if (error == ENOENT)
562             continue;
563         VERIFY0(error);
564
565         /* Only if we removed a hold do we decrement userrefs. */
566         mutex_enter(&ds->ds_lock);

```

```

560         ds->ds_userrefs--;
561         mutex_exit(&ds->ds_lock);
562
563         ddura->ddura_holds_found = B_TRUE;
564     }
565     spa_history_log_internal_ds(ds, "release", tx,
566         "tag=%s refs=%lld", nvpair_name(pair),
567         (longlong_t)ds->ds_userrefs);
568 }
569 }
570
571 static void
572 dsl_dataset_user_release_sync(void *arg, dmu_tx_t *tx)
573 {
574     dsl_dataset_user_release_arg_t *ddura = arg;
575     dsl_holdfunc_t *holdfunc = ddura->ddura_holdfunc;
576 #endif /* ! codereview */
577     dsl_pool_t *dp = dmu_tx_pool(tx);
578     nvpair_t *pair;
579
580     /*
581      * Even though check suggested that at least one of our holds where
582      * found this may have changed. Recalculate ddura_holds_found so that
583      * we can return ENOENT from the caller in the case that no holds
584      * where actually released.
585      */
586     ddura->ddura_holds_found = B_FALSE;
587
588 #endif /* ! codereview */
589     for (pair = nvlist_next_nvpair(ddura->ddura_holds, NULL); pair != NULL;
590         pair = nvlist_next_nvpair(ddura->ddura_holds, pair)) {
591         dsl_dataset_t *ds;
592         int error;
593 #endif /* ! codereview */
594
595         error = holdfunc(dp, nvpair_name(pair), FTAG, &ds);
596         if (error == ENOENT)
597             continue;
598         VERIFY0(error);
599
600         dsl_dataset_user_release_sync_one(ddura, ds,
601             VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
602             dsl_dataset_user_release_sync_one(ds,
603                 fnvpair_value_nvlist(pair), tx);
604         if (nvlist_exists(ddura->ddura_todelete, nvpair_name(pair))) {
605             if (nvlist_exists(ddura->ddura_todelete,
606                 nvpair_name(pair))) {
607                 ASSERT(ds->ds_userrefs == 0 &&
608                     ds->ds_phys->ds_num_children == 1 &&
609                     DS_IS_DEFER_DESTROY(ds));
610                 dsl_destroy_snapshot_sync_impl(ds, B_FALSE, tx);
611             }
612             dsl_dataset_rele(ds, FTAG);
613         }
614     }
615 }
616
617 /*
618  * The full semantics of this function are described in the comment above
619  * lz_c_release().
620  *
621  * To summarize:
622  * Releases holds specified in the nvl holds.
623  *
624 #endif /* ! codereview */
625 * holds is nvl of snapname -> { holdname, ... }

```

```

621 * errlist will be filled in with snapname -> error
622 *
623 * If tmpdp is not NULL the names for holds should be the dbobj's of snapshots,
624 * otherwise they should be the names of snapshots.
625 *
626 * As a release may cause snapshots to be destroyed this tries to ensure they
627 * aren't mounted.
628 *
629 * The release of non-existent holds are skipped.
630 *
631 * At least one hold must have been released for the this function to succeed
632 * and return 0.
633 * if any fails, all will fail.
634 */
635 static int
636 dsl_dataset_user_release_impl(nvlist_t *holds, nvlist_t *errlist,
637 dsl_pool_t *tmpdp)
638 {
639     dsl_dataset_user_release_arg_t ddura;
640     nvpair_t *pair;
641     char *pool;
642 #endif /* ! codereview */
643     int error;

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

647 #ifdef _KERNEL
648     /*
649      * The release may cause snapshots to be destroyed; make sure they
650      * are not mounted.
651      */
652     if (tmpdp != NULL) {
653         /* Temporary holds are specified by dbobj. */
654         ddura.ddura_holdfunc = dsl_dataset_hold_byobj;
655         pool = spa_name(tmpdp->dp_spa);
656         ddura.ddura_holds = holds;
657         ddura.ddura_errlist = errlist;
658         ddura.ddura_todelete = fnvlist_alloc();

659         dsl_pool_config_enter(tmpdp, FTAG);
660         for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
661             pair = nvlist_next_nvpair(holds, pair)) {
662             error = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_release_check,
663                 dsl_dataset_user_release_sync, &ddura, fnvlist_num_pairs(holds));
664             fnvlist_free(ddura.ddura_todelete);
665             return (error);
666         }
667     }

668 typedef struct dsl_dataset_user_release_tmp_arg {
669     uint64_t ddurta_dsobj;
670     nvlist_t *ddurta_holds;
671     boolean_t ddurta_deleteme;
672 } dsl_dataset_user_release_tmp_arg_t;

673 static int
674 dsl_dataset_user_release_tmp_check(void *arg, dmu_tx_t *tx)
675 {
676     dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
677     dsl_pool_t *dp = dmu_tx_pool(tx);
678     dsl_dataset_t *ds;
679     int error;

```

```

216     if (!dmu_tx_is_syncing(tx))
217         return (0);

218
219     error = dsl_dataset_hold_obj(dp, ddurta->ddurta_dsobj, FTAG, &ds);
220     if (error)
221         return (error);

222
223     error = dsl_dataset_hold_byobj(tmpdp, nvpair_name(pair),
224 FTAG, &ds);
225     error = dsl_dataset_user_release_check_one(ds,
226 ddurta->ddurta_holds, &ddurta->ddurta_deleteme);
227     dsl_dataset_rele(ds, FTAG);
228     return (error);
229 }

230 static void
231 dsl_dataset_user_release_tmp_sync(void *arg, dmu_tx_t *tx)
232 {
233     dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
234     dsl_pool_t *dp = dmu_tx_pool(tx);
235     dsl_dataset_t *ds;

236     VERIFY0(dsl_dataset_hold_obj(dp, ddurta->ddurta_dsobj, FTAG, &ds));
237     dsl_dataset_user_release_sync_one(ds, ddurta->ddurta_holds, tx);
238     if (ddurta->ddurta_deleteme) {
239         ASSERT(ds->ds_userrefs == 0 &&
240             ds->ds_phys->ds_num_children == 1 &&
241             DS_IS_DEFER_DESTROY(ds));
242         dsl_destroy_snapshot_sync_impl(ds, B_FALSE, tx);
243     }
244     dsl_dataset_rele(ds, FTAG);
245 }

246 /*
247 * Called at spa_load time to release a stale temporary user hold.
248 * Also called by the onexit code.
249 */
250 void
251 dsl_dataset_user_release_tmp(dsl_pool_t *dp, uint64_t dsobj, const char *htag)
252 {
253     dsl_dataset_user_release_tmp_arg_t ddurta;
254     dsl_dataset_t *ds;
255     int error;

256
257 #ifdef _KERNEL
258     /* Make sure it is not mounted. */
259     dsl_pool_config_enter(dp, FTAG);
260     error = dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds);
261     if (error == 0) {
262         char name[MAXNAMELEN];
263         dsl_dataset_name(ds, name);
264         dsl_dataset_rele(ds, FTAG);
265         dsl_pool_config_exit(dp, FTAG);
266         zfs_unmount_snap(name);
267     }
268     dsl_pool_config_exit(tmpdp, FTAG);
269 #endif /* ! codereview */
270     } else {
271         /* Non-temporary holds are specified by name. */
272         ddura.ddura_holdfunc = dsl_dataset_hold;
273         pool = nvpair_name(pair);
274
275         for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
276             pair = nvlist_next_nvpair(holds, pair))
277             zfs_unmount_snap(nvpair_name(pair));

```



```

268         dsl_pool_config_exit(dp, FTAG);
682     }
683 #endif

685     ddura.ddura_holds = holds;
686     ddura.ddura_errlist = errlist;
687     ddura.ddura_todelete = fnvlist_alloc();
688     ddura.ddura_holds_found = B_FALSE;
272     ddurta.ddurta_dsobj = dsobj;
273     ddurta.ddurta_holds = fnvlist_alloc();
274     fnvlist_add_boolean(ddurta.ddurta_holds, htag);

276     (void) dsl_sync_task(spa_name(dp->dp_spa),
277         dsl_dataset_user_release_tmp_check,
278         dsl_dataset_user_release_tmp_sync, &ddurta, 1);
279     fnvlist_free(ddurta.ddurta_holds);
280 }

690     error = dsl_sync_task(pool, dsl_dataset_user_release_check,
691         dsl_dataset_user_release_sync, &ddura,
692         fnvlist_num_pairs(holds));
693     fnvlist_free(ddura.ddura_todelete);
282 typedef struct zfs_hold_cleanup_arg {
283     char zhca_spaname[MAXNAMELEN];
284     uint64_t zhca_spa_load_guid;
285     uint64_t zhca_dsobj;
286     char zhca_htag[MAXNAMELEN];
287 } zfs_hold_cleanup_arg_t;

695     /* If at least one hold wasn't removed return ENOENT. */
696     if (error == 0 && !ddura.ddura_holds_found)
697         error = ENOENT;
289 static void
290 dsl_dataset_user_release_onexit(void *arg)
291 {
292     zfs_hold_cleanup_arg_t *ca = arg;
293     spa_t *spa;
294     int error;

699     return (error);
700 }
296     error = spa_open(ca->zhca_spaname, &spa, FTAG);
297     if (error != 0) {
298         zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
299             "because pool is no longer loaded",
300             ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
301         return;
302     }
303     if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
304         zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
305             "because pool is no longer loaded (guid doesn't match)",
306             ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
307         spa_close(spa, FTAG);
308         return;
309     }

702 /*
703 * holds is nvl of snapname -> { holdname, ... }
704 * errlist will be filled in with snapname -> error
705 *
706 * if any fails, all will fail.
707 */
708 int
709 dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist)
710 {
711     return dsl_dataset_user_release_impl(holds, errlist, NULL);

```

```

311     dsl_dataset_user_release_tmp(spa_get_dsl(spa),
312         ca->zhca_dsobj, ca->zhca_htag);
313     kmem_free(ca, sizeof (zfs_hold_cleanup_arg_t));
314     spa_close(spa, FTAG);
712 }

714 /*
715 * holds is nvl of snapdsobj -> { holdname, ... }
716 */
717 #endif /* ! codereview */
718 void
719 dsl_dataset_user_release_tmp(struct dsl_pool *dp, nvlist_t *holds)
317 dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
318     minor_t minor)
720 {
721     ASSERT(dp != NULL);
722     (void) dsl_dataset_user_release_impl(holds, NULL, dp);
320     zfs_hold_cleanup_arg_t *ca = kmem_alloc(sizeof (*ca), KM_SLEEP);
321     spa_t *spa = dsl_dataset_get_spa(ds);
322     (void) strncpy(ca->zhca_spaname, spa_name(spa),
323         sizeof (ca->zhca_spaname));
324     ca->zhca_spa_load_guid = spa_load_guid(spa);
325     ca->zhca_dsobj = ds->ds_object;
326     (void) strncpy(ca->zhca_htag, htag, sizeof (ca->zhca_htag));
327     VERIFY0(zfs_onexit_add_cb(minor,
328         dsl_dataset_user_release_onexit, ca, NULL));
723 }
_____unchanged_portion_omitted_

```

```

*****
10147 Wed May 1 01:43:41 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 UINTE64_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 *tx);
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 Wed May 1 01:43:42 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 Wed May 1 01:43:42 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