

```
*****
```

```
8478 Wed May 1 01:43:39 2013
```

```
new/usr/src/cmd/ndmpd/ndmp/ndmpd_chkpkt.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;  
192  
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     }  
197  
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     }  
204  
205     p = strchr(snapname, '@') + 1;  
206     if (zfs_hold(zhp, p, jname, recursive, cleanup_fd) != 0) {  
207         if (zfs_hold(zhp, p, jname, recursive, B_FALSE, cleanup_fd) != 0) {  
208             NDMP_LOG(LOG_ERR, "Cannot hold snapshot %s", p);  
209             zfs_close(zhp);  
210             return (-1);  
211         }  
212         zfs_close(zhp);  
213     }  
_____unchanged_portion_omitted_____
```

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

1

```
*****
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 processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
_____unchanged_portion_omitted_____
5104 static int
5105 zfs_do_hold_rele_impl(int argc, char **argv, boolean_t holding)
5106 {
5107     int errors = 0;
5108     int i;
5109     const char *tag;
5110     boolean_t recursive = B_FALSE;
5111     const char *opts = holding ? "rt" : "r";
5112     int c;
5113
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     }
5126
5127     argc -= optind;
5128     argv += optind;
5129
5130     /* check number of arguments */
5131     if (argc < 2)
5132         usage(B_FALSE);
5133
5134     tag = argv[0];
5135     --argc;
5136     ++argv;
5137
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     }
5143
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];
5149
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';
5159
5160         zhp = zfs_open(g_zfs, parent,
5161                       ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
5162     }
5163 }
```

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

2

```
5162         if (zhp == NULL) {
5163             ++errors;
5164             continue;
5165         }
5166         if (holding) {
5167             if (zfs_hold(zhp, delim+1, tag, recursive, -1) != 0)
5168                 if (zfs_hold(zhp, delim+1, tag, recursive,
5169                               B_FALSE, -1) != 0)
5170                     ++errors;
5171             else {
5172                 if (zfs_release(zhp, delim+1, tag, recursive) != 0)
5173                     ++errors;
5174             }
5175             zfs_close(zhp);
5176         }
5177     }
5178     return (errors != 0);
5179 }
5180 _____unchanged_portion_omitted_____
5181 }
```

```
new/usr/src/lib/libzfs/common/libzfs.h
```

```
1
```

```
*****  
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_nv1(zfs_handle_t *, int, nvlist_t *);  
597     boolean_t, boolean_t, int);
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 "camount=off" on all modified filesystems */
632     boolean_t camountoff;  
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
```

new/usr/src/lib/libzfs/common/libzfs_dataset.c

1

```
*****
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 processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
unchanged_portion_omitted_
4080 static int
4081 zfs_hold_one(zfs_handle_t *zhp, void *arg)
4082 {
4083     struct holdarg *ha = arg;
4084     zfs_handle_t *szhp;
4084     char name[ZFS_MAXNAMELEN];
4085     int rv = 0;
4086
4087     (void) snprintf(name, sizeof (name),
4088                     "%s@%s", zhp->zfs_name, ha->snapname);
4089
4090     if (lzc_exists(name))
4091         szhp = make_dataset_handle(zhp->zfs_hdl, name);
4092     if (szhp) {
4093         fnvlist_add_string(ha->nvl, name, ha->tag);
4094         zfs_close(szhp);
4095     }
4096
4097     if (ha->recursive)
4098         rv = zfs_iter_filesystems(zhp, zfs_hold_one, ha);
4099     zfs_close(zhp);
4100     return (rv);
4101
4102 int
4103 zfs_hold(zfs_handle_t *zhp, const char *snapname, const char *tag,
4104           boolean_t recursive, int cleanup_fd)
4105           boolean_t recursive, boolean_t enoent_ok, int cleanup_fd)
4106 {
4107     int ret;
4108     struct holdarg ha;
4109     nvlist_t *errors;
4110     libzfs_handle_t *hdl = zhp->zfs_hdl;
4111     char errbuf[1024];
4112     nvpair_t *elem;
4113
4114     ha.nvl = fnvlist_alloc();
4115     ha.snapname = snapname;
4116     ha.tag = tag;
4117     ha.recursive = recursive;
4118     (void) zfs_hold_one(zfs_handle_dup(zhp), &ha);
4119     ret = zfs_hold_nvl(zhp, cleanup_fd, ha.nvl);
4120     ret = lzc_hold(ha.nvl, cleanup_fd, &errors);
4121     fnvlist_free(ha.nvl);
4122
4123     return (ret);
4124 }
4125
4126 int
4127 zfs_hold_nvl(zfs_handle_t *zhp, int cleanup_fd, nvlist_t *holds)
4128 {
4129     int ret;
4130     nvlist_t *errors;
4131     libzfs_handle_t *hdl = zhp->zfs_hdl;
4132     char errbuf[1024];
4133     nvpair_t *elem;
4134
4135     ret = lzc_hold(holds, cleanup_fd, &errors);
```

new/usr/src/lib/libzfs/common/libzfs_dataset.c

2

```
4127
4128 #endif /* ! codereview */
4129     if (ret == 0)
4130         return (0);
4131
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     }
4149
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'"),
4156                         nvpair_name(elem));
4157         switch (fnvpair_value_int32(elem)) {
4158             case E2BIG:
4159                 /*
4160                  * Temporary tags wind up having the ds object id
4161                  * prepended. So even if we passed the length check
4162                  * above, it's still possible for the tag to wind
4163                  * up being slightly too long.
4164
4165                 (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
4166                 break;
4167             case EINVAL:
4168                 (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
4169                 break;
4170             case EEXIST:
4171                 (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
4172                 break;
4173             case ENOENT:
4174                 if (enoent_ok)
4175                     return (ENOENT);
4176                 /* FALLTHROUGH */
4177             default:
4178                 (void) zfs_standard_error(hdl,
4179                                         fnvpair_value_int32(elem), errbuf);
4180         }
4181         fnvlist_free(errors);
4182         return (ret);
4183     }
4184
4185     struct releasearg {
4186         nvlist_t *nvl;
4187         const char *snapname;
4188         const char *tag;
4189         boolean_t recursive;
4190     };
4191 }
```

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

unchanged_portion_omitted

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

```
*****  
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         }  
842     }  
843 }
```

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

```

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 caseEFAULT:
853 case EROFS:
854 zfs_error_aux(hdl, strerror(errno));
855 return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));
857 default:
858 return (zfs_standard_error(hdl, errno, errbuf));
859 }
860 }
862 *sizep = zc.zc_objset_type;
864 return (0);
865 }

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

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

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

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

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

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

```

```

906     switch (errno) {
907         case EXDEV:
908             zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
909                           "not an earlier snapshot from the same fs"));
910             return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));
912
913         case ENOENT:
914             if (zfs_dataset_exists(hdl, zc.zc_name,
915                           ZFS_TYPE_SNAPSHOT)) {
916                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
917                               "incremental source (@%s) does not exist"),
918                               zc.zc_value);
919             }
920             return (zfs_error(hdl, EZFS_NOENT, errbuf));
921
922         case EDQUOT:
923         case EFBIG:
924         case EIO:
925         case ENOLINK:
926         case ENOSPC:
927         case ENOSTR:
928         case ENXIO:
929         case EPIPE:
930         case ERANGE:
931         case EFAULT:
932         case EROFS:
933             zfs_error_aux(hdl, strerror(errno));
934             return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));
935
936         default:
937             return (zfs_standard_error(hdl, errno, errbuf));
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
832     } unchanged_portion_omitted
833
834     static int
835     dump_snapshot(zfs_handle_t *zhp, void *arg)
836     {
837         send_dump_data_t *sdd = arg;
838         progress_arg_t pa = { 0 };
839         pthread_t tid;
840
841         char *thissnap;
842         int err;
843         boolean_t isfromsnap, istosnap, fromorigin;
844         boolean_t exclude = B_FALSE;
845
846         thissnap = strchr(zhp->zfs_name, '@') + 1;
847         isfromsnap = (sdd->fromsnap != NULL &&
848                       strcmp(sdd->fromsnap, thissnap) == 0);
849
850         if (!sdd->seenfrom && isfromsnap) {
851             gather_holds(zhp, sdd);
852             err = hold_for_send(zhp, sdd);
853             if (err == 0) {
854                 sdd->seenfrom = B_TRUE;
855                 (void) strcpy(sdd->prevsnap, thissnap);
856                 sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
857                 sdd->prevsnap_obj = zfs_prop_get_int(zhp,
858                                           ZFS_PROP_OBJSETID);
859             } else if (err == ENOENT) {
860                 err = 0;
861             }
862             zfs_close(zhp);
863             return (0);
864             return (err);
865         }
866
867         if (sdd->seento || !sdd->seenfrom) {
868             zfs_close(zhp);
869             return (0);
870         }
871
872         istosnap = (strcmp(sdd->tosnap, thissnap) == 0);
873         if (istosnap)
874             sdd->seento = B_TRUE;
875
876         if (!sdd->doall && !isfromsnap && !istosnap) {
877             if (sdd->replicate) {
878                 char *snapname;
879                 nvlist_t *snapprops;
880                 /*
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
```

```

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
1060         } else {
1061             exclude = B_TRUE;
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         if (sdd->parsable) {
1100             if (sdd->prevsnap[0] != '\0') {
1101                 (void) fprintf(stderr, "incremental\t%s\t%s",
1102                               sdd->prevsnap, zhp->zfs_name);
1103             } else {
1104                 (void) fprintf(stderr, "full\t%s",
1105                               zhp->zfs_name);
1106             }
1107             (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1108                           "send from @%s to %s"),
1109                           sdd->prevsnap, zhp->zfs_name);
1110         }
1111         if (err == 0) {
1112             if (sdd->parsable) {

```

```

1113             (void) fprintf(stderr, "\t@%llu\n",
1114                           (longlong_t) size);
1115         } else {
1116             char buf[16];
1117             zfs_nicenum(size, buf, sizeof (buf));
1118             (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
1119                           " estimated size is %s\n"), buf);
1120         }
1121         sdd->size += size;
1122     } else {
1123         (void) fprintf(stderr, "\n");
1124     }
1125
1126     if (!sdd->dryrun) {
1127         /*
1128         * If progress reporting is requested, spawn a new thread to
1129         * poll ZFS_IOC_SEND_PROGRESS at a regular interval.
1130         */
1131         if (sdd->progress) {
1132             pa.pa_zhp = zhp;
1133             pa.pa_fd = sdd->outfd;
1134             pa.pa_parsable = sdd->parsable;
1135
1136             if (err = pthread_create(&tid, NULL,
1137                                     send_progress_thread, &pa)) {
1138                 zfs_close(zhp);
1139                 return (err);
1140             }
1141             if (err = dump_ioctl(zhp, sdd->prevsnap, sdd->prevsnap_obj,
1142                                   fromorigin, sdd->outfd, sdd->debugnv));
1143
1144             if (sdd->progress) {
1145                 (void) pthread_cancel(tid);
1146                 (void) pthread_join(tid, NULL);
1147             }
1148
1149             (void) strcpy(sdd->prevsnap, thissnap);
1150             sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
1151             zfs_close(zhp);
1152             return (err);
1153         }
1154     }
1155
1156     unchanged_portion_omitted_
1157
1158     /*
1159     * Generate a send stream for the dataset identified by the argument zhp.
1160     *
1161     * The content of the send stream is the snapshot identified by
1162     * 'tosnap'. Incremental streams are requested in two ways:
1163     * - from the snapshot identified by "fromsnap" (if non-null) or
1164     * - from the origin of the dataset identified by zhp, which must
1165     *   be a clone. In this case, "fromsnap" is null and "fromorigin"
1166     *   is TRUE.
1167     *
1168     * The send stream is recursive (i.e. dumps a hierarchy of snapshots) and
1169     * uses a special header (with a hdrtype field of DMU_COMPOUNDSTREAM)
1170     * if "replicate" is set. If "doall" is set, dump all the intermediate
1171     * snapshots. The DMU_COMPOUNDSTREAM header is used in the "doall"
1172     * case too. If "props" is set, send properties.
1173     */
1174     int
1175     zfs_send(zfs_handle_t *zhp, const char *fromsnap, const char *tosnap,
1176              sendflags_t *flags, int outfd, snapfilter_cb_t filter_func,

```

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

```

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 *fsss = 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;

1355     (void) snprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
1356             "cannot send '%s'"), zhp->zfs_name);

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     }

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     }

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     }

1392     if (flags->replicate || flags->doall || flags->props) {
1393         dmu_replay_record_t drr = { 0 };
1394         char *packbuf = NULL;
1395         size_t buflen = 0;
1396         zio_cksum_t zc = { 0 };

1398     if (flags->replicate || flags->props) {
1399         nvlist_t *hdrnv;

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));

```

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

```

        if (!flags->replicate) {
                VERIFY(0 == nvlist_add_boolean(hdrnv,
                                                "not_recursive"));
        }

        err = gather_nvlist(zhp->zfs_hdl, zhp->zfs_name,
                             fromsnap, tosnap, flags->replicate, &fss, &fsav1);
        if (err)
                goto err_out;
        VERIFY(0 == nvlist_add_nvlist(hdrnv, "fss", fss));
        err = nvlist_pack(hdrnv, &packbuf, &buflen,
                           NV_ENCODE_XDR, 0);
        if (debugnvp)
                *debugnvp = hdrnv;
        else
                nvlist_free(hdrnv);
        if (err) {
                fsav1_destroy(fsav1);
                nvlist_free(fss);
                goto stderr_out;
        }
}

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

        /* write header nvlist */
        if (err != -1 && packbuf != NULL) {
                err = cksum_and_write(packbuf, buflen, &zc,
                                      outfd);
        }
        free(packbuf);
        if (err == -1) {
                fsav1_destroy(fsav1);
                nvlist_free(fss);
                err = errno;
                goto stderr_out;
        }

        /* write end record */
        bzero(&drr, sizeof (drr));
        drr.drr_type = DRR_END;
        drr.drr_u.drr_end.drr_checksum = zc;
        err = write(outfd, &drr, sizeof (drr));
        if (err == -1) {
                fsav1_destroy(fsav1);
                nvlist_free(fss);
                err = errno;
                goto stderr_out;
        }
}

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     }

1541     if (sdd.snapholds != NULL) {
1542         /* Holds are required. */
1543         if (!flags->verbose) {
1544             /*
1545              * A verbose dry run wasn't done so do a non-verbose
1546              * dry run to gather snapshot hold's.
1547            */
1548             sdd.dryrun = B_TRUE;
1549             err = dump_filesystems(zhp, &sdd);
1550             sdd.dryrun = flags->dryrun;
1551         }
1553         if (err != 0) {
1554             fnvlist_free(sdd.snapholds);
1555             goto stderr_out;
1556         }
1558         err = zfs_hold_nvl(zhp, sdd.cleanup_fd, sdd.snapholds);
1559         fnvlist_free(sdd.snapholds);
1560         if (err != 0)
1561             goto stderr_out;
1562     }
1563 #endif /* ! codereview */
1564     err = dump_filesystems(zhp, &sdd);
1565     fsavl_destroy(fsavl);
1566     nvlist_free(fss);

1569     if (flags->dedup) {
1570         (void) close(pipefd[0]);
1571         (void) pthread_join(tid, NULL);
1572     }

1574     if (sdd.cleanup_fd != -1) {
1575         VERIFY(0 == close(sdd.cleanup_fd));
1576         sdd.cleanup_fd = -1;
1577     }

1579     if (!flags->dryrun && (flags->replicate || flags->doall ||
1580         flags->props)) {
1581         /*
1582          * write final end record. NB: want to do this even if
1583          * there was some error, because it might not be totally
1584          * failed.
1585        */
1586     dmu_replay_record_t drr = { 0 };
1587     drr.drr_type = DRR_END;
1588     if (write(outfd, &drr, sizeof (drr)) == -1) {
1589         return (zfs_standard_error(zhp->zfs_hdl,
1590             errno, errbuf));
1591     }
1592 }

1594     return (err || sdd.err);

1596 stderr_out:
1597     err = zfs_standard_error(zhp->zfs_hdl, err, errbuf);
1598 err_out:
1599     if (sdd.cleanup_fd != -1)
1600         VERIFY(0 == close(sdd.cleanup_fd));
1601     if (flags->dedup) {
1602         (void) pthread_cancel(tid);
1603         (void) pthread_join(tid, NULL);
1604         (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;

1649     buf = zfs_alloc(hdl, len);
1650     if (buf == NULL)
1651         return (ENOMEM);

1652     err = recv_read(hdl, fd, buf, len, byteswap, zc);
1653     if (err != 0) {
1654         free(buf);
1655         return (err);
1656     }

1657     err = nvlist_unpack(buf, len, nvp, 0);
1658     free(buf);
1659     if (err != 0) {
1660         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "invalid "
1661                         "stream (malformed nvlist)"));
1662         return (EINVAL);
1663     }
1664     return (0);
1665 }

1666 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;

1679     zhp = zfs_open(hdl, name, ZFS_TYPE_DATASET);
1680     if (zhp == NULL)
1681         return (-1);
1682     clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
1683                             flags->force ? MS_FORCE : 0);
1684     zfs_close(zhp);
1685     if (clp == NULL)
1686         return (-1);
1687     err = changelist_prefix(clp);
1688     if (err)
1689         return (err);

1690     zc.zc_objset_type = DMU_OST_ZFS;
1691     (void) strlcpy(zc.zc_name, name, sizeof (zc.zc_name));

1692     if (tryname) {
1693         (void) strcpy(newname, tryname);
1694         (void) strlcpy(zc.zc_value, tryname, sizeof (zc.zc_value));

1695         if (flags->verbose) {
1696             (void) printf("attempting rename %s to %s\n",
1697                           zc.zc_name, zc.zc_value);
1698         }
1699         err = ioctl(hdl->libzfs_fd, ZFS_IOC_RENAME, &zc);
1700         if (err == 0)
1701             changelist_rename(clp, name, tryname);
1702         } else {
1703             err = ENOENT;
1704         }

1705         if (err != 0 && strncmp(name + baselen, "recv-", 5) != 0) {
1706             seq++;

1707             (void) sprintf(newname, ZFS_MAXNAMELEN, "%.*srecv-%u-%u",
1708                           baselen, name, getpid(), seq);
1709             (void) strlcpy(zc.zc_value, newname, sizeof (zc.zc_value));

1710             if (flags->verbose) {
1711                 (void) printf("failed - trying rename %s to %s\n",
1712                               zc.zc_name, zc.zc_value);
1713             }
1714             err = ioctl(hdl->libzfs_fd, ZFS_IOC_RENAME, &zc);
1715             if (err == 0)
1716                 changelist_rename(clp, name, newname);
1717             if (err && flags->verbose) {
1718                 (void) printf("failed (%u) - "
1719                             "will try again on next pass\n", errno);
1720             }
1721             err = EAGAIN;
1722         } else if (flags->verbose) {
1723             if (err == 0)
1724                 (void) printf("success\n");
1725             else
1726                 (void) printf("failed (%u)\n", errno);
1727         }
1728     }
1729 }

1730     if (err == 0)
1731         (void) printf("success\n");
1732     else
1733         (void) printf("failed (%u)\n", errno);
1734 }

1735 }
```

```

1737     (void) changelist_postfix(clp);
1738     changelist_free(clp);
1740
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
1755     zhp = zfs_open(hdl, name, ZFS_TYPE_DATASET);
1756     if (zhp == NULL)
1757         return (-1);
1758     clp = changelist_gather(zhp, ZFS_PROP_NAME, 0,
1759       flags->force ? MS_FORCE : 0);
1760     if (zfs_get_type(zhp) == ZFS_TYPE_SNAPSHOT &&
1761         zfs_spa_version(zhp, &spa_version) == 0 &&
1762         spa_version >= SPA_VERSION_USERREFS)
1763         defer = B_TRUE;
1764     zfs_close(zhp);
1765     if (clp == NULL)
1766         return (-1);
1767     err = changelist_prefix(clp);
1768     if (err)
1769         return (err);
1770
1771     zc.zc_objset_type = DMU_OST_ZFS;
1772     zc.zc_defer_destroy = defer;
1773     (void) strlcpy(zc.zc_name, name, sizeof (zc.zc_name));
1774
1775     if (flags->verbose)
1776         (void) printf("attempting destroy %s\n", zc.zc_name);
1777     err = ioctl(hdl->libzfs_fd, ZFS_IOC_DESTROY, &zc);
1778     if (err == 0) {
1779         if (flags->verbose)
1780             (void) printf("success\n");
1781         changelist_remove(clp, zc.zc_name);
1782     }
1783
1784     (void) changelist_postfix(clp);
1785     changelist_free(clp);
1786
1787 /*
1788  * Deferred destroy might destroy the snapshot or only mark it to be
1789  * destroyed later, and it returns success in either case.
1790 */
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
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
1811     if (gtnd->skip != NULL &&
1812         strcmp(zhp->zfs_name, gtnd->skip) == 0) {
1813         return (0);
1814     }
1815
1816     if (zhp->zfs_dmustats.dds_guid == gtnd->guid) {
1817         (void) strcpy(gtnd->name, zhp->zfs_name);
1818         zfs_close(zhp);
1819         return (EXIST);
1820     }
1821
1822     err = zfs_iter_children(zhp, guid_to_name_cb, gtnd);
1823     zfs_close(zhp);
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
1845     gtnd.guid = guid;
1846     gtnd.name = name;
1847     gtnd.skip = NULL;
1848
1849     (void) strlcpy(pname, parent, sizeof (pname));
1850
1851     /*
1852      * Search progressively larger portions of the hierarchy. This will
1853      * select the "most local" version of the origin snapshot in the case
1854      * that there are multiple matching snapshots in the system.
1855      */
1856     while ((cp = strrchr(pname, '/')) != NULL) {
1857
1858         /* Chop off the last component and open the parent */
1859         *cp = '\0';
1860         zhp = make_dataset_handle(hdl, pname);
1861
1862         if (zhp == NULL)
1863             continue;
1864
1865         err = zfs_iter_children(zhp, guid_to_name_cb, &gtnd);
1866         zfs_close(zhp);
1867         if (err == EXIST)
1868             return (0);

```

```

1869 /*
1870      * Remember the dataset that we already searched, so we
1871      * skip it next time through.
1872      */
1873      gtnd.skip = pname;
1874 }
1875
1876     return (ENOENT);
1877 }

1878 /*

1879 * Return +1 if guid1 is before guid2, 0 if they are the same, and -1 if
1880 * guid1 is after guid2.
1881 */
1882
1883 static int
1884 created_before(libzfs_handle_t *hdl, avl_tree_t *avl,
1885                 uint64_t guid1, uint64_t guid2)
1886 {
1887     nvlist_t *nvfs;
1888     char *fsname, *snapname;
1889     char buf[ZFS_MAXNAMELEN];
1890     int rv;
1891     zfs_handle_t *guid1hdl, *guid2hdl;
1892     uint64_t create1, create2;

1893     if (guid2 == 0)
1894         return (0);
1895     if (guid1 == 0)
1896         return (1);

1897     nvfs = fsavl_find(avl, guid1, &snapname);
1898     VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));
1899     (void) snprintf(buf, sizeof (buf), "%s@%s", fsname, snapname);
1900     guid1hdl = zfs_open(hdl, buf, ZFS_TYPE_SNAPSHOT);
1901     if (guid1hdl == NULL)
1902         return (-1);

1903     nvfs = fsavl_find(avl, guid2, &snapname);
1904     VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));
1905     (void) snprintf(buf, sizeof (buf), "%s@%s", fsname, snapname);
1906     guid2hdl = zfs_open(hdl, buf, ZFS_TYPE_SNAPSHOT);
1907     if (guid2hdl == NULL) {
1908         zfs_close(guid1hdl);
1909         return (-1);
1910     }

1911     create1 = zfs_prop_get_int(guid1hdl, ZFS_PROP_CREATETXG);
1912     create2 = zfs_prop_get_int(guid2hdl, ZFS_PROP_CREATETXG);

1913     if (create1 < create2)
1914         rv = -1;
1915     else if (create1 > create2)
1916         rv = +1;
1917     else
1918         rv = 0;

1919     zfs_close(guid1hdl);
1920     zfs_close(guid2hdl);

1921
1922     return (rv);
1923 }

1924 static int
1925 recv_incremental_replication(libzfs_handle_t *hdl, const char *tofs,
1926                               recvflags_t *flags, nvlist_t *stream_nv, avl_tree_t *stream_avl,
1927                               nvlist_t *renamed)

```

```

1935 {
1936     nvlist_t *local_nv;
1937     avl_tree_t *local_avl;
1938     nvpair_t *fselem, *nextfselem;
1939     char *fromsnap;
1940     char newname[ZFS_MAXNAMELEN];
1941     int error;
1942     boolean_t needagain, progress, recursive;
1943     char *s1, *s2;

1945     VERIFY(0 == nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap));

1947 recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
1948             ENOENT);

1950 if (flags->dryrun)
1951     return (0);

1953 again:
1954     needagain = progress = B_FALSE;

1956 if ((error = gather_nvlist(hdl, tofs, fromsnap, NULL,
1957                             recursive, &local_nv, &local_avl)) != 0)
1958     return (error);

1960 /*
1961  * Process deletes and renames
1962  */
1963 for (fselem = nvlist_next_nvpair(local_nv, NULL);
1964     fselem; fselem = nextfselem) {
1965     nvlist_t *nvfs, *snaps;
1966     nvlist_t *stream_nvfs = NULL;
1967     nvpair_t *snapelem, *nextsnapelem;
1968     uint64_t fromguid = 0;
1969     uint64_t originguid = 0;
1970     uint64_t stream_originguid = 0;
1971     uint64_t parent_fromsnap_guid, stream_parent_fromsnap_guid;
1972     char *fsname, *stream_fsname;

1974     nextfselem = nvlist_next_nvpair(local_nv, fselem);

1976     VERIFY(0 == nvpair_value_nvlist(fselem, &nvfs));
1977     VERIFY(0 == nvlist_lookup_nvlist(nvfs, "snaps", &snaps));
1978     VERIFY(0 == nvlist_lookup_string(nvfs, "name", &fsname));
1979     VERIFY(0 == nvlist_lookup_uint64(nvfs, "parentfromsnap",
1980                                     &parent_fromsnap_guid));
1981     (void) nvlist_lookup_uint64(nvfs, "origin", &originguid);

1983 /*
1984  * First find the stream's fs, so we can check for
1985  * a different origin (due to "zfs promote")
1986  */
1987 for (snapelem = nvlist_next_nvpair(snaps, NULL);
1988     snapelem; snapelem = nvlist_next_nvpair(snaps, snapelem)) {
1989     uint64_t thisguid;

1991     VERIFY(0 == nvpair_value_uint64(snapelem, &thisguid));
1992     stream_nvfs = fsavl_find(stream_avl, thisguid, NULL);

1994     if (stream_nvfs != NULL)
1995         break;
1996 }

1998 /* check for promote */
1999 (void) nvlist_lookup_uint64(stream_nvfs, "origin",
2000     &stream_originguid);

```

```

2001     if (stream_nvfs && originguid != stream_originguid) {
2002         switch (created_before(hdl, local_avl,
2003                         stream_originguid, originguid)) {
2004             case 1: {
2005                 /* promote it! */
2006                 zfs_cmd_t zc = { 0 };
2007                 nvlist_t *origin_nvfs;
2008                 char *origin_fspath;
2009
2010                if (flags->verbose)
2011                    (void) printf("promoting %s\n", fspath);
2012
2013                origin_nvfs = fsavl_find(local_avl, originguid,
2014                                         NULL);
2015                VERIFY(0 == nvlist_lookup_string(origin_nvfs,
2016                                              "name", &origin_fspath));
2017                (void) strlcpy(zc.zc_value, origin_fspath,
2018                               sizeof (zc.zc_value));
2019                (void) strlcpy(zc.zc_name, fspath,
2020                               sizeof (zc.zc_name));
2021                error = zfs_ioctl(hdl, ZFS_IOC_PROMOTE, &zc);
2022                if (error == 0)
2023                    progress = B_TRUE;
2024                break;
2025            }
2026            default:
2027                break;
2028        case -1:
2029            fsavl_destroy(local_avl);
2030            nvlist_free(local_nv);
2031            return (-1);
2032        }
2033        /* We had/have the wrong origin, therefore our
2034        * list of snapshots is wrong. Need to handle
2035        * them on the next pass.
2036        */
2037        needagain = B_TRUE;
2038        continue;
2039    }
2040
2041    for (snapelem = nvlist_next_nvpair(snaps, NULL);
2042         snapelem; snapelem = nextsnapelem) {
2043        uint64_t thisguid;
2044        char *stream_snapname;
2045        nvlist_t *found, *props;
2046
2047        nextsnapelem = nvlist_next_nvpair(snaps, snapelem);
2048
2049        VERIFY(0 == nvpair_value_uint64(snapelem, &thisguid));
2050        found = fsavl_find(stream_avl, thisguid,
2051                           &stream_snapname);
2052
2053        /* check for delete */
2054        if (found == NULL) {
2055            char name[ZFS_MAXNAMELEN];
2056
2057            if (!flags->force)
2058                continue;
2059
2060            (void) sprintf(name, sizeof (name), "%s@%s",
2061                           fspath, nvpair_name(snapelem));
2062
2063            error = recv_destroy(hdl, name,
2064                                 strlen(fspath)+1, newname, flags);
2065            if (error)
2066

```

```

2067                     needagain = B_TRUE;
2068
2069             else
2070                 progress = B_TRUE;
2071             continue;
2072         }
2073
2074         stream_nvfs = found;
2075
2076         if (0 == nvlist_lookup_nvlist(stream_nvfs, "snapprops",
2077                                       &props) && 0 == nvlist_lookup_nvlist(props,
2078                                         stream_snapname, &props)) {
2079             zfs_cmd_t zc = { 0 };
2080
2081             zc.zc_cookie = B_TRUE; /* received */
2082             (void) snprintf(zc.zc_name, sizeof (zc.zc_name),
2083                            "%s@%s", fspath, nvpair_name(snapelem));
2084             if (zcmd_write_src_nvlist(hdl, &zc,
2085                                       props) == 0) {
2086                 (void) zfs_ioctl(hdl,
2087                                 ZFS_IOC_SET_PROP, &zc);
2088                 zcmd_free_nvlists(&zc);
2089             }
2090
2091             /* check for different snapname */
2092             if (strcmp(nvpair_name(snapelem),
2093                        stream_snapname) != 0) {
2094                 char name[ZFS_MAXNAMELEN];
2095                 char tryname[ZFS_MAXNAMELEN];
2096
2097                 (void) snprintf(name, sizeof (name), "%s@%s",
2098                                fspath, nvpair_name(snapelem));
2099                 (void) snprintf(tryname, sizeof (name), "%s@%s",
2100                                fspath, stream_snapname);
2101
2102                 error = recv_rename(hdl, name, tryname,
2103                                      strlen(fspath)+1, newname, flags);
2104                 if (error)
2105                     needagain = B_TRUE;
2106                 else
2107                     progress = B_TRUE;
2108             }
2109
2110             if (strcmp(stream_snapname, fromsnap) == 0)
2111                 fromguid = thisguid;
2112
2113
2114             /* check for delete */
2115             if (stream_nvfs == NULL) {
2116                 if (!flags->force)
2117                     continue;
2118
2119                 error = recv_destroy(hdl, fspath, strlen(tofs)+1,
2120                                       newname, flags);
2121                 if (error)
2122                     needagain = B_TRUE;
2123                 else
2124                     progress = B_TRUE;
2125                 continue;
2126             }
2127
2128             if (fromguid == 0) {
2129                 if (flags->verbose) {
2130                     (void) printf("local fs %s does not have "
2131                                  "freesnap (%s in stream); must have "
2132                                  "been deleted locally; ignoring\n",
2133

```

```

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
2160     parent = fsavl_find(local_avl,
2161         stream_parent_fromsnap_guid, NULL);
2162     /*
2163      * NB: parent might not be found if we used the
2164      * tosnap for stream_parent_fromsnap_guid,
2165      * because the parent is a newly-created fs;
2166      * we'll be able to rename it after we recv the
2167      * new fs.
2168 */
2169     if (parent != NULL) {
2170         char *pname;
2171
2172         VERIFY(0 == nvlist_lookup_string(parent, "name",
2173             &pname));
2174         (void) sprintf(tryname, sizeof(tryname),
2175             "%s%s", pname, strrchr(stream_fsname, '/'));
2176     } else {
2177         tryname[0] = '\0';
2178         if (flags->verbose) {
2179             (void) printf("local fs %s new parent "
2180                         "not found\n", fsname);
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;
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 }
2211 }
2213 static int
2214 zfs_receive_package(libzfs_handle_t *hdl, int fd, const char *destname,
2215     recvflags_t *flags, dmu_replay_record_t *dr, 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     dmu_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) sprintf(errbuf, sizeof(errbuf), dgettext(TEXT_DOMAIN,
2232         "cannot receive"));
2234     assert(dr->drr_type == DRR_BEGIN);
2235     assert(dr->drr_u.drr_begin.drr_magic == DMU_BACKUP_MAGIC);
2236     assert(DMU_GET_STREAM_HDRTYPE(dr->drr_u.drr_begin.drr_versioninfo) ==
2237         DMU_COMPOUNDSTREAM);
2239     /*
2240      * Read in the nvlist from the stream.
2241 */
2242     if (dr->drr_payloadlen != 0) {
2243         error = recv_read_nvlist(hdl, fd, dr->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),

```

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

21

```

2265     flags->byterwap, NULL)));
2266     goto out;
2267 }
2268 if (flags->byterwap) {
2269     drre.drr_type = BSWAP_32(drre.drr_type);
2270     drre.drr_u.drr_end.drr_checksum.zc_word[0] =
2271         BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[0]);
2272     drre.drr_u.drr_end.drr_checksum.zc_word[1] =
2273         BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[1]);
2274     drre.drr_u.drr_end.drr_checksum.zc_word[2] =
2275         BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[2]);
2276     drre.drr_u.drr_end.drr_checksum.zc_word[3] =
2277         BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[3]);
2278 }
2279 if (drre.drr_type != DRR_END) {
2280     error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2281     goto out;
2282 }
2283 if (!ZIO_CHECKSUM_EQUAL(drre.drr_u.drr_end.drr_checksum, *zc)) {
2284     zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2285         "incorrect header checksum"));
2286     error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2287     goto out;
2288 }
2289 (void) nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap);
2290 if (drr->drr_payloadlen != 0) {
2291     nvlist_t *stream_fss;
2292
2293     VERIFY(0 == nvlist_lookup_nvlist(stream_nv, "fss",
2294         &stream_fss));
2295     if ((stream_avl = fsavl_create(stream_fss)) == NULL) {
2296         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2297             "couldn't allocate avl tree"));
2298         error = zfs_error(hdl, EZFS_NOMEM, errbuf);
2299         goto out;
2300     }
2301
2302     if (fromsnap != NULL) {
2303         nvlist_t *renamed = NULL;
2304         nvpair_t *pair = NULL;
2305
2306         (void) strlcpy(tofs, destname, ZFS_MAXNAMELEN);
2307         if (flags->isprefix) {
2308             struct drr_begin *drrb = &drr->drr_u.drr_begin;
2309             int i;
2310
2311             if (flags->istail) {
2312                 cp = strrchr(drrb->drr_toname, '/');
2313                 if (cp == NULL) {
2314                     (void) strlcat(tofs, "/", ZFS_MAXNAMELEN);
2315                     i = 0;
2316                 } else {
2317                     i = (cp - drrb->drr_toname);
2318                 }
2319             } else {
2320                 i = strcspn(drrb->drr_toname, "/@");
2321             }
2322             /* zfs_receive_one() will create_parents() */
2323             (void) strlcat(tofs, &drrb->drr_toname[i],
2324                             ZFS_MAXNAMELEN);
2325             *strchr(tofs, '@') = '\0';
2326         }
2327     }
2328
2329     if (recursive && !flags->dryrun && !flags->nomount) {

```

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

```

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     }
2399
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 }
2410
2411 static void
2412 trunc_prop_errs(int truncated)
2413 {
2414     ASSERT(truncated != 0);
2415
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 }
2423
2424 static int
2425 recv_skip(libzfs_handle_t *hdl, int fd, boolean_t byteswap)
2426 {
2427     dmu_replay_record_t *drr;
2428     void *buf = malloc(1<<20);
2429     char errbuf[1024];
2430
2431     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2432         "cannot receive:"));
2433
2434     /* XXX would be great to use lseek if possible... */
2435     drr = buf;
2436
2437     while (recv_read(hdl, fd, drr, sizeof (dmu_replay_record_t),
2438         byteswap, NULL) == 0) {
2439         if (byteswap)
2440             drr->drr_type = BSWAP_32(drr->drr_type);
2441
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;
2451
2452             case DRR_END:
2453                 free(buf);
2454                 return (0);
2455
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         }
2463         (void) recv_read(hdl, fd, buf,

```

```

2463                     P2ROUNDUP(drr->drr_u.drr_object.drr_bonuslen, 8),
2464                     B_FALSE, NULL);
2465             break;
2466
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;
2487
2488             default:
2489                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2490                     "invalid record type"));
2491                 return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2492             }
2493         }
2494         free(buf);
2495         return (-1);
2496     }
2497 }
2498
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, dmu_replay_record_t *drr,
2505     dmu_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;
2524
2525     begin_time = time(NULL);
2526
2527     (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2528         "cannot receive"));

```

```

2530 recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
2531 ENOENT);
2532
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;
2539
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));
2545
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);
2553
2554     if (0 == nvlist_lookup_nvlist(fs, "snapprops", &props)) {
2555         VERIFY(0 == nvlist_lookup_nvlist(props,
2556             snapname, &snapprops_nvlist));
2557     }
2558
2559     if (ret != 0)
2560         return (-1);
2561 }
2562
2563 cp = NULL;
2564
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     }
2583
2584     chopprefix = strrchr(sendfs, '/');
2585
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     }
2596 }

```

```

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     }
2610
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
2621     } else {
2622     /* A snapshot was specified as an exact path (no -d or -e). */
2623     if (recursive) {
2624         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2625             "cannot specify snapshot name for multi-snapshot "
2626             "stream"));
2627         return (zfs_error(hdl, EZFS_BADSTREAM, errbuf));
2628     }
2629     chopprefix = drrb->drr_toname + strlen(drrb->drr_toname);
2630
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');
2636
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     }
2647
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         if (flags->verbose)

```

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

27

```

2661         (void) printf("found clone origin %s\n", zc.zc_string);
2662     }
2663
2664     stream_wantsnewfs = (drrb->drr_fromguid == NULL ||
2665                           (drrb->drr_flags & DRR_FLAG_CLONE));
2666
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"));
2674
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, strchr(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"));
2696
2697         (void) strcpy(zc.zc_name, zc.zc_value);
2698         *strchr(zc.zc_name, '@') = '\0';
2699
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     }
2717
2718     (void) strcpy(zc.zc_name, zc.zc_value);
2719     *strchr(zc.zc_name, '@') = '\0';
2720
2721     if (zfs_dataset_exists(hdl, zc.zc_name, ZFS_TYPE_DATASET)) {
2722         zfs_handle_t *zhp;
2723
2724         /*
2725          * Destination fs exists. Therefore this should either
2726          * be an incremental, or the stream specifies a new fs

```

new/usr/src/lib/libzfs/common/libzfs_sendrecv.c

```

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 }
2750
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 }
2756
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 }
2767
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_wantsnewfs ||
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 filesystem'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         }
2860     }
2861 }
2862
2863 if (err == 0) {
2864     nvlist_free(prop_errors);
2865 }
2866
2867 zc.zc_nvlist_dst = 0;
2868 zc.zc_nvlist_dst_size = 0;
2869 zcmd_free_nvlists(&zc);
2870
2871 if (err == 0 && snapprops_nvlist) {
2872     zfs_cmd_t zc2 = { 0 };
2873
2874     (void) strcpy(zc2.zc_name, zc.zc_value);
2875     zc2.zc_cookie = B_TRUE; /* received */
2876     if (zcmd_write_src_nvlist(hdl, &zc2, snapprops_nvlist) == 0) {
2877         (void) zfs_ioctl(hdl, ZFS_IOC_SET_PROP, &zc2);
2878         zcmd_free_nvlists(&zc2);
2879     }
2880
2881     if (err && (ioctl_errno == ENOENT || ioctl_errno == EEXIST)) {
2882         /*
2883          * It may be that this snapshot already exists,
2884          * in which case we want to consume & ignore it
2885          * rather than failing.
2886         */
2887         avl_tree_t *local_avl;
2888         nvlist_t *local_nv, *fs;
2889         cp = strchr(zc.zc_value, '@');
2890
2891         /*
2892          * XXX Do this faster by just iterating over snaps in
2893          * this fs. Also if zc_value does not exist, we will
2894          * get a strange "does not exist" error message.
2895         */
2896         *cp = '\0';
2897         if (gather_nvlist(hdl, zc.zc_value, NULL, NULL, B_FALSE,
2898                           &local_nv, &local_avl) == 0) {
2899             *cp = '@';
2900             fs = fsavl_find(local_avl, drrb->drr_toguid, NULL);
2901             fsavl_destroy(local_avl);
2902             nvlist_free(local_nv);
2903
2904             if (fs != NULL) {
2905                 if (flags->verbose) {
2906                     (void) printf("snap %s already exists; "
2907                                 "ignoring\n", zc.zc_value);
2908                 }
2909             }
2910             err = ioctl_err = recv_skip(hdl, infd,
2911                                         flags->byteswap);
2912         }
2913     }
2914
2915     *cp = '@';
2916 }
2917
2918 if (ioctl_err != 0) {
2919     switch (ioctl_errno) {
2920     case ENODEV:
2921         cp = strchr(zc.zc_value, '@');
2922     }
2923 }
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
3600
3601
3602
3603
3604
3605
3606
3607
3608
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
3650
3651
3652
3653
3654
3655
3656
3657
3658
3659
3660
3661
3662
3663
3664
3665
3666
3667
3668
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
3700
3701
3702
3703
3704
3705
3706
3707
3708
3709
3710
3711
3712
3713
3714
3715
3716
3717
3718
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
4000
4001
4002
4003
4004
4005
4006
4007
4008
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
4050
4051
4052
4053
4054
4055
4056
4057
4058
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
4100
4101
4102
4103
4104
4105
4106
4107
4108
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
4250
4251
4252
4253
4254
4255
4256
4257
4258
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
4350
4351
4352
4353
4354
4355
4356
4357
4358
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
4400
4401
4402
4403
4404
4405
4406
4407
4408
4409
4410
4411
4412
4413
4414
4415
4416
4417
4418
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
4450
4451
4452
4453
4454
4455
4456
4457
4458
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
4500
4501
4502
4503
4504
4505
4506
4507
4508
4509
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
4550
4551
4552
4553
4554
4555
4556
4557
4558
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
4600
4601
4602
4603
4604
4605
4606
4607
4608
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
4650
4651
4652
4653
4654
4655
4656
4657
4658
4659
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
4700
4701
4702
4703
4704
4705
4706
4707
4708
4709
4710
4711
4712
4713
4714
4715
4716
4717
4718
4719
4720
4721
4722
4723
4724
4725
4726
4727
4728
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
4750
4751
4752
4753
4754
4755
4756
4757
4758
4759
4760
4761
4762
4763
4764
4765
4766
4767
4768
4769
4770
4771
4772
4773
4774
4775
4776
4777
4778
4779
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799
4800
4801
4802
4803
4804
4805
4806
4807
4808
4809
4810
4811
4812
4813
4814
4815
4816
4817
4818
4819
4820
4821
4822
4823
4824
4825
4826
4827
4828
4829
4830
4831
4832
4833
4
```

```

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 EEXIST:
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  * Mount the target filesystem (if created). Also mount any
2975  * children of the target filesystem if we did a replication
2976  * receive (indicated by stream_avl being non-NULL).
2977  */
2978 cp = strchr(zc.zc_value, '@');
2979 if (cp && (ioctl_err == 0 || !newfs)) {
2980     zfs_handle_t *h;
2981
2982     *cp = '\0';
2983     h = zfs_open(hdl, zc.zc_value,
2984                 ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
2985     if (h != NULL) {
2986         if (h->zfs_type == ZFS_TYPE_VOLUME) {
2987             *cp = '@';
2988         } else if (newfs || stream_avl) {
2989             /*

```

```

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 }
3002     if (clp) {
3003         err |= changelist_postfix(clp);
3004         changelist_free(clp);
3005     }
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     }
3020     if (err || ioctl_err)
3021         return (-1);
3023     *action_handlep = zc.zc_action_handle;
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_nicenum(bytes, buf1, sizeof (buf1));
3033         zfs_nicenum(bytes/delta, buf2, sizeof (buf2));
3035         (void) printf("received %sB stream in %lu seconds (%sB/sec)\n",
3036                         buf1, delta, buf2);
3037     }
3039     return (0);
3040 }
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     dm 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;
3055     (void) sprintf(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     }
3064
3065     /* read in the BEGIN record */
3066     if ((0 != (err = recv_read(hdl, infd, &drr, sizeof (drr), B_FALSE,
3067         &zcksum)))
3068         return (err);
3069
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     }
3074
3075     /* the kernel needs the non-byteswapped begin record */
3076     drr_noswap = drr;
3077
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;
3087
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     }
3098
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     }
3104
3105     featureflags = DMU_GET_FEATUREFLAGS(drrb->drr_versioninfo);
3106     hdrtypes = DMU_GET_STREAM_HDRTYPE(drrb->drr_versioninfo);
3107
3108     if (!DMU_STREAM_SUPPORTED(featureflags) ||
3109         (hdrtypes != DMU_SUBSTREAM && hdrtypes != 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     }
3115
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     }
3121
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_handlep));
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_handlep));
3144     }
3145
3146     /*
3147      * Restores a backup of tosnap from the file descriptor specified by infd.
3148      * Return 0 on total success, -2 if some things couldn't be
3149      * destroyed/rename/promoted, -1 if some things couldn't be received.
3150      * (-1 will override -2).
3151      */
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;
3161
3162         cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
3163         VERIFY(cleanup_fd >= 0);
3164
3165         err = zfs_receive_impl(hdl, tosnap, flags, infd, NULL, NULL,
3166             stream_avl, &top_zfs, cleanup_fd, &action_handle);
3167
3168         VERIFY(0 == close(cleanup_fd));
3169
3170         if (err == 0 && !flags->nomount && top_zfs) {
3171             zfs_handle_t *zhp;
3172             prop_changelist_t *clp;
3173
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 }
```

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

```
1
```

```
*****
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 processi
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 where 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
379     /* determine the pool name */
380     elem = nvlist_next_nvpair(holds, NULL);
381     if (elem == NULL)
382         return (0);
383     (void) strlcpy(pool, nvpair_name(elem), sizeof (pool));
384     pool[strcspn(pool, "/@")] = '\0';
385
386     args = fnvlist_alloc();
387     fnvlist_add_nvlist(args, "holds", holds);
388     if (cleanup_fd != -1)
389         fnvlist_add_int32(args, "cleanup_fd", cleanup_fd);
390
391     error = lzc_ioctl(ZFS_IOC_HOLD, pool, args, errlist);
392 }
```

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

```
2
```

```
387         nvlist_free(args);
388         return (error);
389     }
390
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
415      * The return value will be 0 if all holds were removed.
416
417      * Otherwise the return value will be the errno of a (unspecified) release
418      * that failed, no holds will be released, and the errlist will have an
419      * entry for each snapshot that has failed releases (name = snapshot).
420      * The value in the errlist will be the error code (int32) of a failed release.
421
422      */
423
424     int
425     lzc_release(nvlist_t *holds, nvlist_t **errlist)
426     {
427         char pool[MAXNAMELEN];
428         nvpair_t *elem;
429
430         /* determine the pool name */
431         elem = nvlist_next_nvpair(holds, NULL);
432         if (elem == NULL)
433             return (0);
434         (void) strlcpy(pool, nvpair_name(elem), sizeof (pool));
435         pool[strcspn(pool, "/@")] = '\0';
436
437         return (lzc_ioctl(ZFS_IOC_RELEASE, pool, holds, errlist));
438     }
439
440     _____unchanged_portion_omitted_____
441 }
```

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

```
*****
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 processi
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     dsl_dataset_user_release_tmp(dp, holds);
872 #endif /* ! codereview */
873     zap_cursor_fini(&zc);

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);
```

1

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

```
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
```

2

```

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. ioctl, zfs_ioc_*) are either read-only
985 * or modifying operations.
986 *
987 * Modifying operations should generally use dsl_sync_task(). The synctask
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;

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 }

```

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

1

```
*****
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 processi
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_syntask.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;

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

2

```
62         if (strlen(htag) > MAXNAMELEN)  
63             return (E2BIG);  
64         /* Temp holds have a more restricted length */  
65         if (temphold && strlen(htag) + MAX_TAG_PREFIX_LEN >= MAXNAMELEN)  
66             return (E2BIG);  
67  
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         }  
82         return (error);  
83     }  
84 }  
85  
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 */  
95  
96     if (spa_version(dp->dp_spa) < SPA_VERSION_USERREFS)  
97         return (SET_ERROR(ENOTSUP));  
98  
99     holds_possible = B_FALSE;  
100 }#endif /* ! codereview */  
101 for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;  
102     pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {  
103     int error = 0;  
104     dsl_dataset_t *ds;  
105     char *htag;  
106  
107     /* must be a snapshot */  
108     if (strchr(nvpair_name(pair), '@') == NULL)  
109         error = SET_ERROR(EINVAL);  
110  
111     if (error == 0)  
112         error = nvpair_value_string(pair, &htag);  
113     if (error == 0) {  
114         error = dsl_dataset_hold(dp,  
115                               nvpair_name(pair), FTAG, &ds);  
116  
117         if (error == ENOENT) {  
118             /*  
119             * We register ENOENT errors so they can be  
120             * correctly reported if needed, such as when  
121             * all holds fail.  
122             */  
123             if (dduha->dduha_errlist != NULL) {  
124                 fnvlist_add_int32(dduha->dduha_errlist,  
125                                   nvpair_name(pair), error);  
126             }
```

```

127         }
128         continue;
129     } /*endif /* ! codereview */
130     if (error == 0) {
131         error = dsl_dataset_user_hold_check_one(ds, htag,
132             dduha->dduha_minor != 0, tx);
133         dsl_dataset_rele(ds, FTAG);
134     }
135
136     if (error != 0) {
137         if (dduha->dduha_errlist != NULL) {
138             rv = error;
139             fnvlist_add_int32(dduha->dduha_errlist,
140                 nvpair_name(pair), error);
141         }
142         rv = error;
143     } else {
144         holds_possible = B_TRUE;
145     }
146 #endif /* ! codereview */
147 }
148
149 /*
150  * Check that at least one hold will possibly be created,
151  * otherwise fail.
152  */
153 if (rv == 0 && !holds_possible)
154     rv = ENOENT;
155
156 #endif /* ! codereview */
157 return (rv);
158 }
159
160
161 static void
162 dsl_dataset_user_hold_sync_one_impl(dsl_dataset_user_hold_arg_t *dduha,
163     dsl_dataset_t *ds, const char *htag, minor_t minor, uint64_t now,
164     dmu_tx_t *tx)
165 {
166     void
167     dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
168         minor_t minor, uint64_t now, dmu_tx_t *tx)
169 {
170     dsl_pool_t *dp = ds->ds_dir->dd_pool;
171     objset_t *mos = dp->dp_meta_objset;
172     uint64_t zapobj;
173
174     mutex_enter(&ds->ds_lock);
175     if (ds->ds_phys->ds_userrefs_obj == 0) {
176         /*
177          * This is the first user hold for this dataset. Create
178          * the userrefs zap object.
179          */
180         dmu_buf_will_dirty(ds->ds_dbuf, tx);
181         zapobj = ds->ds_phys->ds_userrefs_obj =
182             zap_create(mos, DMU_OT_USERREFS, DMU_OT_NONE, 0, tx);
183     } else {
184         zapobj = ds->ds_phys->ds_userrefs_obj;
185     }
186     ds->ds_userrefs++;
187     mutex_exit(&ds->ds_lock);
188
189     VERIFY0(zap_add(mos, zapobj, htag, 8, 1, &now, tx));
190
191     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) sprintf(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",
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_sspaname, &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_sspaname);
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_sspaname);
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
255 }
256
257     ASSERT(spa != NULL);
258     ca = kmalloc(sizeof (*ca), KM_SLEEP);
259
260     (void) strlcpy(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 }

267 void
268 dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
269     minor_t minor, uint64_t now, dmu_tx_t *tx)
270 {
271     dsl_dataset_user_hold_arg_t dduha;
272
273     dduha.dduha_spa = NULL;
274     dduha.dduha_holds = NULL;
275     dduha.dduha_tmpholds = fnvlist_alloc();
276     dduha.dduha_errlist = NULL;
277     dduha.dduha_minor = minor;
278     dduha.dduha_holds_created = B_FALSE;
279
280     dsl_dataset_user_hold_sync_one_impl(&dduha, ds, htag, minor, now, tx);
281     dsl_register_onexit_hold_cleanup(dsl_dataset_get_spa(ds),
282         dduha.dduha_tmpholds, minor);
283 }
284 }

285 #endif /* ! codereview */
286 static void
287 dsl_dataset_user_hold_sync(void *arg, dmu_tx_t *tx)
288 {
289     dsl_dataset_user_hold_arg_t *dduha = arg;
290     dsl_pool_t *dp = dmu_tx_pool(tx);
291     nvpair_t *pair;
292     uint64_t now = getrestime_sec();
293
294     for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;
295         pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {
296         dsl_dataset_t *ds;
297         char *name;
298         int error;
299
300         name = nvpair_name(pair);
301         error = dsl_dataset_hold(dp, name, FTAG, &ds);
302         if (error == 0) {
303             dsl_dataset_user_hold_sync_one_impl(dduha, ds,
304                 fnvpair_value_string(pair), dduha->dduha_minor,
305                 now, tx);
306             VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
307             dsl_dataset_user_hold_sync_one(ds, fnvpair_value_string(pair),
308                 dduha->dduha_minor, now, tx);
309             dsl_dataset_rele(ds, FTAG);
310             dduha->dduha_holds_created = B_TRUE;
311         } else if (dduha->dduha_errlist != NULL) {
312             /*
313             * We register ENOENT errors so they can be correctly
314             * reported if needed, such as when all holds fail.
315             */
316             fnvlist_add_int32(dduha->dduha_errlist, name, error);
317     }
318 }
319
320 #endif /* ! codereview */
321

```

```

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 * if cleanup_minor is not 0, the holds will be temporary, cleaned up
331 * 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 *
338 * If cleanup minor is not 0, the holds will be temporary, which will be cleaned
339 * up when the process exits.
340 *
341 * On success all the holds, for snapshots that existed, will be created and 0
342 * will be returned.
343 *
344 * On failure no holds will be created, the errlist will be filled in,
345 * and an errno will returned.
346 *
347 * In all cases the errlist will contain entries for holds where the snapshot
348 * didn't exist.
349 * if any fails, all will fail.
350 */
351 int
352 dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor, nvlist_t *errlist)
353 {
354     dsl_dataset_user_hold_arg_t dduha;
355     nvpair_t *pair;
356     int ret;
357 #endif /* ! codereview */

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

361     dduha.dduha_spa = NULL;
362 #endif /* ! codereview */
363     dduha.dduha_holds = holds;
364     dduha.dduha_tmpholds = fnvlist_alloc();
365 #endif /* ! codereview */
366     dduha.dduha_errlist = errlist;
367     dduha.dduha_minor = cleanup_minor;
368     dduha.dduha_holds_created = B_FALSE;
369 #endif /* ! codereview */

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

```

```

380             fnvlist_free(dduha.dduha_tmpholds);
381             ret = ENOENT;
382         } else {
383             fnvlist_free(dduha.dduha_tmpholds);
384         }
385     }
386
387     return (ret);
388 }
389 typedef int (dsl_holdfunc_t)(dsl_pool_t *dp, const char *name, void *tag,
390     dsl_dataset_t **dsp);
391
392 #endif /* ! codereview */
393 typedef struct dsl_dataset_user_release_arg {
394     dsl_holdfunc_t *ddura_holdfunc;
395     nvlist_t *ddura_holds;
396     nvlist_t *ddura_todelete;
397     nvlist_t *ddura_errlist;
398     boolean_t ddura_holds_found;
399 } dsl_dataset_user_release_arg_t;
400
401 #endif /* ! codereview */
402 }
403
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 }
411
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 {
417     uint64_t zapobj;
418     nvpair_t *pair;
419     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
420     int error;
421     int numholds = 0;
422     int ret;
423 #endif /* ! codereview */
424
425     *todelete = B_FALSE;
426     ret = 0;
427 #endif /* ! codereview */
428
429     if (!dsl_dataset_is_snapshot(ds))
430         return (SET_ERROR(EINVAL));
431
432     zapobj = ds->ds_phys->ds_userrefs_obj;
433     if (zapobj == 0)
434         return (SET_ERROR(ESRCH));
435
436     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
437         pair = nvlist_next_nvpair(holds, pair)) {
438         /* Make sure the hold exists */
439         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     }
447     if (error == ENOENT)
448         error = SET_ERROR(ESRCH);
449     if (error != 0)
450         return (error);
451     numholds++;
452
453     if (DS_IS_DEFER_DESTROY(ds) && ds->ds_phys->ds_num_children == 1 &&
454         ds->ds_userrefs == numholds) {
455         /* we need to destroy the snapshot as well */
456         if (dsl_dataset_long_hold(ds))
457             return (SET_ERROR(EBUSY));
458         *todelete = B_TRUE;
459     }
460
461     if (numholds != 0)
462         ddura->ddura_holds_found = B_TRUE;
463
464     return (ret);
465 }
466
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;
476
477     if (!dmu_tx_is_syncing(tx))
478         return (0);
479
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;
486
487         error = nvpair_value_nvlist(pair, &holds);
488         if (error != 0)
489             return (SET_ERROR(EINVAL));
490
491         error = holdfunc(dp, name, FTAG, &ds);
492         error = dsl_dataset_hold(dp, name, FTAG, &ds);
493         if (error == 0) {
494             boolean_t deleteme;
495             error = dsl_dataset_user_release_check_one(ddura, ds,
496                 error = dsl_dataset_user_release_check_one(ds,
497                     holds, &deleteme);
498
499             /*
500              * Don't check for error == 0 as deleteme is only set
501              * to B_TRUE if it's correct to do so despite the error
502              * e.g. ENOENT.
503             */
504             if (deleteme) {

```

```

132         if (error == 0 && deleteme) {
133             fnvlist_add_boolean(ddura->ddura_todelete,
134                                 name);
135         }
136     dsl_dataset_rele(ds, FTAG);
137 }
138 if (error != 0) {
139     if (ddura->ddura_errlist != NULL) {
140         fnvlist_add_int32(ddura->ddura_errlist,
141                           name, error);
142     }
143     /* Non-existent holds aren't always fatal. */
144     if (error != ENOENT)
145         if (error != ENOENT)
146             rv = error;
147     }
148 /*
149  * None of the specified holds existed so avoid the overhead of a sync
150  * and return ENOENT.
151  */
152 if (rv == 0 && !ddura->ddura_holds_found)
153     rv = ENOENT;
154
155 #endif /* ! codereview */
156     return (rv);
157 }
158
159 /*
160  * static void
161  * dsl_dataset_user_release_sync_one(dsl_dataset_user_release_arg_t *ddura,
162  *                                   dsl_dataset_t *ds, nvlist_t *holds, dmu_tx_t *tx)
163  * dsl_dataset_user_release_sync_one(dsl_dataset_t *ds, nvlist_t *holds,
164  *                                   dmu_tx_t *tx)
165  */
166 static void
167 dsl_dataset_user_release_sync_one(dsl_dataset_user_release_arg_t *ddura,
168                                   dsl_dataset_t *ds, nvlist_t *holds,
169                                   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     int error;
175     nvpair_t *pair;
176
177     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
178          pair = nvlist_next_nvpair(holds, pair)) {
179         uint64_t zapobj;
180         int error;
181         char *name;
182
183         name = nvpair_name(pair);
184
185         /* Remove temporary hold if one exists. */
186         error = dsl_pool_user_release(dp, ds->ds_object, name, tx);
187         ds->ds_userrefs--;
188         error = dsl_pool_user_release(dp, ds->ds_object,
189                                       nvpair_name(pair), tx);
190         VERIFY(error == 0 || error == ENOENT);
191
192         /* Remove user hold if one exists. */
193         /* Endif /* ! codereview */
194         zapobj = ds->ds_phys->ds_userrefs_obj;
195         error = zap_remove(mos, zapobj, name, tx);
196         if (error == ENOENT)
197             continue;
198         VERIFY(error == 0);
199
200         /* Only if we removed a hold do we decrement userrefs. */
201         mutex_enter(&ds->ds_lock);
202     }
203 }
```

```

560         ds->ds_userrefs--;
561         mutex_exit(&ds->ds_lock);
562
563         ddura->ddura_holds_found = B_TRUE;
564         VERIFY0(zap_remove(mos, zapobj, nvpair_name(pair), tx));
565
566         spa_history_log_internal_ds(ds, "release", tx,
567                                     "tag=%s refs=%lld", nvpair_name(pair),
568                                     (longlong_t)ds->ds_userrefs);
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             }
613             /* The full semantics of this function are described in the comment above
614             * lzc_release().
615             *
616             * To summarize:
617             * Releases holds specified in the nvl holds.
618             */
619 #endif /* ! codereview */
620             * holds is nvl of snapname -> { holdname, ... }
```

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

11

```

21 * errlist will be filled in with snapname -> error
22 *
23 * If tmpdp is not NULL the names for holds should be the dbobj's of snapshots,
24 * otherwise they should be the names of snapshots.
25 *
26 * As a release may cause snapshots to be destroyed this tries to ensure they
27 * aren't mounted.
28 *
29 * The release of non-existent holds are skipped.
30 *
31 * At least one hold must have been released for the this function to succeed
32 * and return 0.
33 * if any fails, all will fail.
34 */
35
36 static int
37 dsl_dataset_user_release_impl(nvlist_t *holds, nvlist_t *errlist,
38     dsl_pool_t *tmpdp)
39 {
40     int
41     dsl_dataset_user_release_arg_t ddura;
42     nvpair_t *pair;
43     char *pool;
44 #endif /* ! codereview */
45     int error;
46
47     pair = nvlist_next_nvpair(holds, NULL);
48     if (pair == NULL)
49         return (0);
50
51 #ifdef _KERNEL
52     /*
53      * The release may cause snapshots to be destroyed; make sure they
54      * are not mounted.
55     */
56     if (tmpdp != NULL) {
57         /* Temporary holds are specified by dbobj. */
58         ddura.ddura_holdfunc = dsl_dataset_hold_byobj;
59         pool = spa_name(tmpdp->dp_spa);
60
61         ddura.ddura_holds = holds;
62         ddura.ddura_errlist = errlist;
63         ddura.ddura_todelete = fnvlist_alloc();
64
65         dsl_pool_config_enter(tmpdp, FTAG);
66         for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
67              pair = nvlist_next_nvpair(holds, pair)) {
68             error = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_release_check,
69             dsl_dataset_user_release_sync, &ddura, fnvlist_num_pairs(holds));
70             fnvlist_free(ddura.ddura_todelete);
71         }
72         return (error);
73     }
74
75     typedef struct dsl_dataset_user_release_tmp_arg {
76         uint64_t ddurta_dsoobj;
77         nvlist_t *ddurta_holds;
78         boolean_t ddurta_deleteme;
79     } dsl_dataset_user_release_tmp_arg_t;
80
81     static int
82     dsl_dataset_user_release_tmp_check(void *arg, dmu_tx_t *tx)
83     {
84         dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
85         dsl_pool_t *dp = dmu_tx_pool(tx);
86         dsl_dataset_t *ds;
87         int error;
88
89         if (ddurta->ddurta_deleteme) {
90             if (ddurta->ddurta_holds != NULL) {
91                 if (nvlist_isempty(ddurta->ddurta_holds))
92                     error = 0;
93                 else
94                     error = dsl_dataset_user_release(ds, ddurta->ddurta_holds,
95                         dsl_dataset_user_release_check, dsl_dataset_user_release_sync,
96                         &ddura, fnvlist_num_pairs(ds));
97             }
98         }
99     }
100 }
```

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

```

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
231 static void
232 dsl_dataset_user_release_tmp_sync(void *arg, dmu_tx_t *tx)
233 {
234     dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
235     dsl_pool_t *dp = dmu_tx_pool(tx);
236     dsl_dataset_t *ds;
237
238     VERIFY0(dsl_dataset_hold_obj(dp, ddurta->ddurta_dsobj, FTAG, &ds));
239     dsl_dataset_user_release_sync_one(ds, ddurta->ddurta_holds, tx);
240     if (ddurta->ddurta_deleteme) {
241         ASSERT(ds->ds_userrefs == 0 &&
242                ds->ds_phys->ds_num_children == 1 &&
243                DS_IS_DEFER_DESTROY(ds));
244         dsl_destroy_snapshot_sync_impl(ds, B_FALSE, tx);
245     }
246     dsl_dataset_rele(ds, FTAG);
247 }
248
249 /* Called at spa_load time to release a stale temporary user hold.
250  * Also called by the onexit code.
251 */
252 void
253 dsl_dataset_user_release_tmp(dsl_pool_t *dp, uint64_t dsobj, const char *htag)
254 {
255     dsl_dataset_user_release_tmp_arg_t ddurta;
256     dsl_dataset_t *ds;
257     int error;
258
259 #ifdef _KERNEL
260     /* Make sure it is not mounted. */
261     dsl_pool_config_enter(dp, FTAG);
262     error = dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds);
263     if (error == 0) {
264         char name[MAXNAMELEN];
265         dsl_dataset_name(ds, name);
266         dsl_dataset_rele(ds, FTAG);
267         dsl_pool_config_exit(dp, FTAG);
268         zfs_unmount_snap(name);
269     }
270     dsl_pool_config_exit(tmpdp, FTAG);
271 #endif /* ! codereview */
272 } else {
273     /* Non-temporary holds are specified by name. */
274     ddura.ddura_holdfunc = dsl_dataset_hold;
275     pool = nvpair_name(pair);
276
277     for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
278          pair = nvlist_next_nvpair(holds, pair))
279         zfs_unmount_snap(nvpair_name(pair));

```

```

268         dsl_pool_config_exit(dp, FTAG);
682     }
683 #endiff
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;
288
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 }
700
701     return (error);
296     error = spa_open(ca->zhca_sspaname, &spa, FTAG);
297     if (error != 0) {
298         zfs_dbgmmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
299             "because pool is no longer loaded",
300             ca->zhca_sspaname, ca->zhca_dsobj, ca->zhca_htag);
301         return;
302     }
303     if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
304         zfs_dbgmmsg("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_sspaname, ca->zhca_dsobj, ca->zhca_htag);
307         spa_close(spa, FTAG);
308     }
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);
315     }
316
317     /* holds is nvl of snapdsobj -> { holdname, ... } */
318     */
319 #endiff /* ! codereview */
320 void
321 dsl_dataset_user_release_tmp(struct dsl_pool *dp, nvlist_t *holds)
322 dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
323     minor_t minor)
324 {
325     ASSERT(dp != NULL);
326     (void) dsl_dataset_user_release_impl(holds, NULL, dp);
327     zfs_hold_cleanup_arg_t *ca = kmem_alloc(sizeof (*ca), KM_SLEEP);
328     spa_t *spa = dsl_dataset_get_spa(ds);
329     (void) strlcpy(ca->zhca_sspaname, spa_name(spa),
330         sizeof (ca->zhca_sspaname));
331     ca->zhca_spa_load_guid = spa_load_guid(spa);
332     ca->zhca_dsobj = ds->ds_object;
333     (void) strlcpy(ca->zhca_htag, htag, sizeof (ca->zhca_htag));
334     VERIFY0(zfs_onexit_add_cb(minor,
335         dsl_dataset_user_release_onexit, ca, NULL));
336
337     unchanged_portion_omitted_

```

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

1

```
*****
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 processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
*****
_____unchanged_portion_omitted_____
166 /*
167 * The max length of a temporary tag prefix is the number of hex digits
168 * required to express UINT64_MAX plus one for the hyphen.
169 */
170 #define MAX_TAG_PREFIX_LEN 17
172 #define dsl_dataset_is_snapshot(ds) \
173     ((ds)->ds_phys->ds_num_children != 0)
175 #define DS_UNIQUE_IS_ACCURATE(ds) \
176     (((ds)->ds_phys->ds_flags & DS_FLAG_UNIQUE_ACCURATE) != 0)
178 int dsl_dataset_hold(struct dsl_pool *dp, const char *name, void *tag,
179     dsl_dataset_t **dsp);
180 int dsl_dataset_hold_obj(struct dsl_pool *dp, uint64_t dsobj, void *tag,
181     dsl_dataset_t **);
182 void dsl_dataset_rele(dsl_dataset_t *ds, void *tag);
183 int dsl_dataset_own(struct dsl_pool *dp, const char *name,
184     void *tag, dsl_dataset_t **dsp);
185 int dsl_dataset_own_obj(struct dsl_pool *dp, uint64_t dsobj,
186     void *tag, dsl_dataset_t **dsp);
187 void dsl_dataset_disown(dsl_dataset_t *ds, void *tag);
188 void dsl_dataset_name(dsl_dataset_t *ds, char *name);
189 boolean_t dsl_dataset_tryown(dsl_dataset_t *ds, void *tag);
190 void dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
191     minor_t minor);
190 uint64_t dsl_dataset_create_sync(dsl_dir_t *pds, const char *lastname,
191     dsl_dataset_t *origin, uint64_t flags, cred_t *, dmu_tx_t *);
192 uint64_t dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
193     uint64_t flags, dmu_tx_t *tx);
194 int dsl_dataset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors);
195 int dsl_dataset_promote(const char *name, char *confsnap);
196 int dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
197     boolean_t force);
198 int dsl_dataset_rename_snapshot(const char *fsname,
199     const char *oldsnapname, const char *newsnapname, boolean_t recursive);
200 int dsl_dataset_snapshot_tmp(const char *fsname, const char *snapname,
201     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);
```

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

2

```
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 *refdbytesp, 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_dname(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 = kmalloc(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(dd, fmt, ...)
279 #endif
281 #ifdef __cplusplus
282 }
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 */
27
28 #ifndef _SYS_DSL_USERHOLD_H
29 #define _SYS_DSL_USERHOLD_H
30
31 #include <sys/nvpair.h>
32 #include <sys/types.h>
33
34 #ifdef __cplusplus
35 extern "C" {
36 #endif
37
38 struct dsl_pool;
39 struct dsl_dataset;
40 struct dmu_tx;
41
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_tmpl(struct dsl_pool *dp, nvlist_t *holds);
47 void dsl_dataset_user_release_tmpr(struct dsl_pool *dp, uint64_t dsobj,
48     const char *htag);
49 int dsl_dataset_user_hold_check_one(struct dsl_dataset *ds, const char *htag,
50     boolean_t temphold, struct dmu_tx *tx);
51 void dsl_dataset_user_hold_sync_one(struct dsl_dataset *ds, const char *htag,
52     minor_t minor, uint64_t now, struct dmu_tx *tx);
53
54 #ifdef __cplusplus
55 }
56 
```

unchanged_portion_omitted_

```
*****
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 * innv1: {
4970 *     snapname -> { holdname, ... }
4971 *     ...
4972 * }
4973 *
4974 * outnv1: {
4975 *     snapname -> error value (int32)
4976 *     ...
4977 * }
4978 */
4979 /* ARGSUSED */
4980 static int
4981 zfs_ioc_release(const char *pool, nvlist_t *holds, nvlist_t *errlist)
4982 {
4983     nvpair_t *pair;
```

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

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