```
**********************************************************
  105876 Tue Sep 10 06:31:57 2013
new/usr/src/lib/libshare/common/libshare.c
4095 minor cleanup up libshare
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 RackTop Systems.
#endif /* ! codereview */
 */

/*
 * Share control API
 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <libxml/parser.h>
#include <libxml/tree.h>
#include "libshare.h"
#include "libshare_impl.h"
#include <libscf.h>
#include "scfutil.h"
#include <ctype.h>
#include <libintl.h>
#include <thread.h>
#include <synch.h>

#define DFS_LOCK_FILE   "/etc/dfs/fstypes"
#define SA_STRSIZE      256     /* max string size for names */

/*
 * internal object type values returned by sa_get_object_type()
 */
#define SA_TYPE_UNKNOWN         0
#define SA_TYPE_GROUP           1
#define SA_TYPE_SHARE           2
#define SA_TYPE_RESOURCE        3
#define SA_TYPE_OPTIONSET       4
#define SA_TYPE_ALTSPACE        5
```



```
/*
 * internal data structures
 */

extern struct sa_proto_plugin *sap_proto_list;

/* current SMF/SVC repository handle */
extern void getlegacyconfig(sa_handle_t, char *, xmlNodePtr *);
extern int gettransients(sa_handle_t, xmlNodePtr *);
extern int gettransients(sa_handle_impl_t, xmlNodePtr *);
extern char *sa_fstype(char *);
extern boolean_t sa_is_share(void *);
extern boolean_t sa_is_resource(void *);
extern int sa_is_share(void *);
extern int sa_is_resource(void *);
extern ssize_t scf_max_name_len; /* defined in scfutil during initialization */
extern boolean_t sa_group_is_zfs(sa_group_t);
extern boolean_t sa_path_is_zfs(char *);
extern int sa_group_is_zfs(sa_group_t);
extern int sa_path_is_zfs(char *);
extern int sa_zfs_set_sharenfs(sa_group_t, char *, int);
extern int sa_zfs_set_sharesmb(sa_group_t, char *, int);
extern void update_legacy_config(sa_handle_t);
extern int issubdir(char *, char *);
extern int sa_zfs_init(sa_handle_t);
extern void sa_zfs_fini(sa_handle_t);
extern int sa_zfs_init(sa_handle_impl_t);
extern void sa_zfs_fini(sa_handle_impl_t);
extern void sablocksigs(sigset_t *);
extern void saunblocksigs(sigset_t *);
static sa_group_t sa_get_optionset_parent(sa_optionset_t);
static char *get_node_attr(void *, char *);
extern void sa_update_sharetab_ts(sa_handle_t);

/*
 * Data structures for finding/managing the document root to access
 * handle mapping. The list isn't expected to grow very large so a
 * simple list is acceptable. The purpose is to provide a way to start
 * with a group or share and find the library handle needed for
 * various operations.
 */
mutex_t sa_global_lock;
struct doc2handle {
        struct doc2handle       *next;
        xmlNodePtr              root;
        sa_handle_t             handle;
        sa_handle_impl_t        handle;
};
_____unchanged_portion_omitted_

/*
 * Document root to active handle mapping functions.  These are only
 * used internally. A mutex is used to prevent access while the list
 * is changing. In general, the list will be relatively short - one
 * item per thread that has called sa_init().
 */

sa_handle_t
sa_handle_impl_t
get_handle_for_root(xmlNodePtr root)
{
        struct doc2handle *item;

        (void) mutex_lock(&sa_global_lock);
        for (item = sa_global_handles; item != NULL; item = item->next) {
                if (item->root == root)
```

```
252                         break;
253         }
254         (void) mutex_unlock(&sa_global_lock);
255         if (item != NULL)
256                 return (item->handle);
257         return (NULL);
258 }

260 static int
261 add_handle_for_root(xmlNodePtr root, sa_handle_t handle)
215 add_handle_for_root(xmlNodePtr root, sa_handle_impl_t handle)
262 {
263         struct doc2handle *item;
264         int ret = SA_NO_MEMORY;

266         item = (struct doc2handle *)calloc(sizeof (struct doc2handle), 1);
267         if (item != NULL) {
268                 item->root = root;
269                 item->handle = handle;
270                 (void) mutex_lock(&sa_global_lock);
271                 item->next = sa_global_handles;
272                 sa_global_handles = item;
273                 (void) mutex_unlock(&sa_global_lock);
274                 ret = SA_OK;
275         }
276         return (ret);
277 }
```

*unchanged_portion_omitted_*

```
309 /*
310  * sa_find_group_handle(sa_group_t group)
311  *
312  * Find the sa_handle_t for the configuration associated with this
313  * group.
314  */
315 sa_handle_t
316 sa_find_group_handle(sa_group_t group)
317 {
318         xmlNodePtr node = (xmlNodePtr)group;
319         sa_handle_t handle;

321         while (node != NULL) {
322                 if (strcmp((char *)(node->name), "sharecfg") == 0) {
323                         /* have the root so get the handle */
324                         handle = get_handle_for_root(node);
278                         handle = (sa_handle_t)get_handle_for_root(node);
325                         return (handle);
326                 }
327                 node = node->parent;
328         }
329         return (NULL);
330 }

332 /*
333  * set_legacy_timestamp(root, path, timevalue)
334  *
335  * add the current timestamp value to the configuration for use in
336  * determining when to update the legacy files.  For SMF, this
337  * property is kept in default/operation/legacy_timestamp
338  */

340 static void
341 set_legacy_timestamp(xmlNodePtr root, char *path, uint64_t tval)
342 {
343         xmlNodePtr node;
344         xmlChar *lpath = NULL;
```



```
345         sa_handle_t handle;
299         sa_handle_impl_t handle;

347         /* Have to have a handle or else we weren't initialized. */
348         handle = get_handle_for_root(root);
349         if (handle == NULL)
350                 return;

352         for (node = root->xmlChildrenNode; node != NULL;
353             node = node->next) {
354                 if (xmlStrcmp(node->name, (xmlChar *)"legacy") == 0) {
355                         /* a possible legacy node for this path */
356                         lpath = xmlGetProp(node, (xmlChar *)"path");
357                         if (lpath != NULL &&
358                             xmlStrcmp(lpath, (xmlChar *)path) == 0) {
359                                 xmlFree(lpath);
360                                 break;
361                         }
362                         if (lpath != NULL)
363                                 xmlFree(lpath);
364                 }
365         }
366         if (node == NULL) {
367                 /* need to create the first legacy timestamp node */
368                 node = xmlNewChild(root, NULL, (xmlChar *)"legacy", NULL);
369         }
370         if (node != NULL) {
371                 char tstring[32];
372                 int ret;

374                 (void) snprintf(tstring, sizeof (tstring), "%lld", tval);
375                 (void) xmlSetProp(node, (xmlChar *)"timestamp",
376                     (xmlChar *)tstring);
377                 (void) xmlSetProp(node, (xmlChar *)"path", (xmlChar *)path);
378                 /* now commit to SMF */
379                 ret = sa_get_instance(handle->scfhandle, "default");
380                 if (ret == SA_OK) {
381                         ret = sa_start_transaction(handle->scfhandle,
382                             "operation");
383                         if (ret == SA_OK) {
384                                 ret = sa_set_property(handle->scfhandle,
385                                     "legacy-timestamp", tstring);
386                                 if (ret == SA_OK) {
387                                         (void) sa_end_transaction(
388                                             handle->scfhandle, handle);
389                                 } else {
390                                         sa_abort_transaction(handle->scfhandle);
391                                 }
392                         }
393                 }
394         }
395 }
```

*unchanged_portion_omitted_*

```
598 /*
599  * check to see if group/share is persistent.
600  *
601  * "group" can be either an sa_group_t or an sa_share_t. (void *)
602  * works since both thse types are also void *.
603  * If the share is a ZFS share, mark it as persistent.
604  */
605 boolean_t
559 int
606 sa_is_persistent(void *group)
607 {
608         char *type;
```

```
609         boolean_t persist = B_TRUE;
563         int persist = 1;
610         sa_group_t grp;

612         type = sa_get_group_attr((sa_group_t)group, "type");
613         if (type != NULL) {
614                 if (strcmp(type, "transient") == 0)
615                         persist = B_FALSE;
569                         persist = 0;
616                 sa_free_attr_string(type);
617         }

619         grp = (sa_is_share(group)) ? sa_get_parent_group(group) : group;
620         if (sa_group_is_zfs(grp))
621                 persist = B_TRUE;
575                 persist = 1;

623         return (persist);
624 }
_____unchanged_portion_omitted_

816 /*
817  * sa_init(init_service)
818  *      Initialize the API
819  *      find all the shared objects
820  *      init the tables with all objects
821  *      read in the current configuration
822  */

824 #define GETPROP(prop)   scf_simple_prop_next_astring(prop)
825 #define CHECKTSTAMP(st, tval)   stat(SA_LEGACY_DFSTAB, &st) >= 0 && \
826         tval != TSTAMP(st.st_ctim)

828 sa_handle_t
829 sa_init(int init_service)
830 {
831         struct stat st;
832         int legacy = 0;
833         uint64_t tval = 0;
834         int lockfd;
835         sigset_t old;
836         int updatelegacy = B_FALSE;
837         scf_simple_prop_t *prop;
838         sa_handle_t handle;
792         sa_handle_impl_t handle;
839         int err;

841         handle = calloc(sizeof (struct sa_handle), 1);
795         handle = calloc(sizeof (struct sa_handle_impl), 1);

843         if (handle != NULL) {
844                 /*
845                  * Get protocol specific structures, but only if this
846                  * is the only handle.
847                  */
848                 (void) mutex_lock(&sa_global_lock);
849                 if (sa_global_handles == NULL)
850                         (void) proto_plugin_init();
851                 (void) mutex_unlock(&sa_global_lock);
852                 if (init_service & SA_INIT_SHARE_API) {
853                         /*
854                          * initialize access into libzfs. We use this
855                          * when collecting info about ZFS datasets and
856                          * shares.
857                          */
858                         if (sa_zfs_init(handle) == B_FALSE) {
```



```
859                             free(handle);
860                             (void) mutex_lock(&sa_global_lock);
861                             (void) proto_plugin_fini();
862                             (void) mutex_unlock(&sa_global_lock);
863                             return (NULL);
864                     }
865                     /*
866                      * since we want to use SMF, initialize an svc handle
867                      * and find out what is there.
868                      */
869                     handle->scfhandle = sa_scf_init(handle);
870                     if (handle->scfhandle != NULL) {
871                             /*
872                              * Need to lock the extraction of the
873                              * configuration if the dfstab file has
874                              * changed. Lock everything now and release if
875                              * not needed.  Use a file that isn't being
876                              * manipulated by other parts of the system in
877                              * order to not interfere with locking. Using
878                              * dfstab doesn't work.
879                              */
880                             sablocksigs(&old);
881                             lockfd = open(DFS_LOCK_FILE, O_RDWR);
882                             if (lockfd >= 0) {
883                                     extern int errno;
884                                     errno = 0;
885                                     (void) lockf(lockfd, F_LOCK, 0);
886                                     (void) mutex_lock(&sa_dfstab_lock);
887                                     /*
888                                      * Check whether we are going to need
889                                      * to merge any dfstab changes. This
890                                      * is done by comparing the value of
891                                      * legacy-timestamp with the current
892                                      * st_ctim of the file. If they are
893                                      * different, an update is needed and
894                                      * the file must remain locked until
895                                      * the merge is done in order to
896                                      * prevent multiple startups from
897                                      * changing the SMF repository at the
898                                      * same time.  The first to get the
899                                      * lock will make any changes before
900                                      * the others can read the repository.
901                                      */
902                                     prop = scf_simple_prop_get
903                                         (handle->scfhandle->handle,
904                                         (const char *)SA_SVC_FMRI_BASE
905                                         ":default", "operation",
906                                         "legacy-timestamp");
907                                     if (prop != NULL) {
908                                             char *i64;
909                                             i64 = GETPROP(prop);
910                                             if (i64 != NULL)
911                                                     tval = strtoull(i64,
912                                                         NULL, 0);
913                                             if (CHECKTSTAMP(st, tval))
914                                                     updatelegacy = B_TRUE;
915                                             scf_simple_prop_free(prop);
916                                     } else {
917                                             /*
918                                              * We haven't set the
919                                              * timestamp before so do it.
920                                              */
921                                             updatelegacy = B_TRUE;
922                                     }
923                                     if (updatelegacy == B_FALSE) {
924                                             (void) mutex_unlock(
```

```
925                                             &sa_dfstab_lock);
926                                     (void) lockf(lockfd, F_ULOCK,
927                                         0);
928                                     (void) close(lockfd);
929                             }

931                     }
932                     /*
933                      * It is essential that the document tree and
934                      * the internal list of roots to handles be
935                      * setup before anything that might try to
936                      * create a new object is called. The document
937                      * tree is the combination of handle->doc and
938                      * handle->tree. This allows searches,
939                      * etc. when all you have is an object in the
940                      * tree.
941                      */
942                     handle->doc = xmlNewDoc((xmlChar *)"1.0");
943                     handle->tree = xmlNewNode(NULL,
944                         (xmlChar *)"sharecfg");
945                     if (handle->doc != NULL &&
946                         handle->tree != NULL) {
947                             (void) xmlDocSetRootElement(handle->doc,
948                                 handle->tree);
949                             err = add_handle_for_root(handle->tree,
950                                 handle);
951                             if (err == SA_OK)
952                                     err = sa_get_config(
953                                         handle->scfhandle,
954                                         handle->tree, handle);
955                     } else {
956                             if (handle->doc != NULL)
957                                     xmlFreeDoc(handle->doc);
958                             if (handle->tree != NULL)
959                                     xmlFreeNode(handle->tree);
960                             err = SA_NO_MEMORY;
961                     }

963                     saunblocksigs(&old);

965                     if (err != SA_OK) {
966                             /*
967                              * If we couldn't add the tree handle
968                              * to the list, then things are going
969                              * to fail badly. Might as well undo
970                              * everything now and fail the
971                              * sa_init().
972                              */
973                             sa_fini(handle);
974                             if (updatelegacy == B_TRUE) {
975                                     (void) mutex_unlock(
976                                         &sa_dfstab_lock);
977                                     (void) lockf(lockfd,
978                                         F_ULOCK, 0);
979                                     (void) close(lockfd);
980                             }
981                             return (NULL);
982                     }

984                     if (tval == 0) {
985                             /*
986                              * first time so make sure
987                              * default is setup
988                              */
989                             verifydefgroupopts(handle);
990                     }
```



```
992                                     if (updatelegacy == B_TRUE) {
993                                             sablocksigs(&old);
994                                             getlegacyconfig(handle,
948                                             getlegacyconfig((sa_handle_t)handle,
995                                                 SA_LEGACY_DFSTAB, &handle->tree);
996                                             if (stat(SA_LEGACY_DFSTAB, &st) >= 0)
997                                                     set_legacy_timestamp(
998                                                         handle->tree,
999                                                         SA_LEGACY_DFSTAB,
1000                                                        TSTAMP(st.st_ctim));
1001                                            saunblocksigs(&old);
1002                                            /*
1003                                             * Safe to unlock now to allow
1004                                             * others to run
1005                                             */
1006                                            (void) mutex_unlock(&sa_dfstab_lock);
1007                                            (void) lockf(lockfd, F_ULOCK, 0);
1008                                            (void) close(lockfd);
1009                                    }
1010                                    /* Get sharetab timestamp */
1011                                    sa_update_sharetab_ts(handle);
965                                     sa_update_sharetab_ts((sa_handle_t)handle);

1013                                    /* Get lastupdate (transaction) timestamp */
1014                                    prop = scf_simple_prop_get(
1015                                        handle->scfhandle->handle,
1016                                        (const char *)SA_SVC_FMRI_BASE ":default",
1017                                        "state", "lastupdate");
1018                                    if (prop != NULL) {
1019                                            char *str;
1020                                            str =
1021                                                scf_simple_prop_next_astring(prop);
1022                                            if (str != NULL)
1023                                                    handle->tstrans =
1024                                                        strtoull(str, NULL, 0);
1025                                            else
1026                                                    handle->tstrans = 0;
1027                                            scf_simple_prop_free(prop);
1028                                    }
1029                                    legacy |= sa_get_zfs_shares(handle, "zfs");
1030                                    legacy |= gettransients(handle, &handle->tree);
1031                            }
1032                    }
1033            }
1034            return (handle);
988             return ((sa_handle_t)handle);
1035    }

1037    /*
1038     * sa_fini(handle)
1039     *      Uninitialize the API structures including the configuration
1040     *      data structures and ZFS related data.
1041     */

1043    void
1044    sa_fini(sa_handle_t handle)
1045    {
1046            if (handle != NULL) {
1000            sa_handle_impl_t impl_handle = (sa_handle_impl_t)handle;

1002            if (impl_handle != NULL) {
1047                    /*
1048                     * Free the config trees and any other data structures
1049                     * used in the handle.
1050                     */
```

```
1051                 if (handle->doc != NULL)
1052                         xmlFreeDoc(handle->doc);
1007                 if (impl_handle->doc != NULL)
1008                         xmlFreeDoc(impl_handle->doc);

1054                 /* Remove and free the entry in the global list. */
1055                 remove_handle_for_root(handle->tree);
1011                 remove_handle_for_root(impl_handle->tree);

1057                 /*
1058                  * If this was the last handle to release, unload the
1059                  * plugins that were loaded. Use a mutex in case
1060                  * another thread is reinitializing.
1061                  */
1062                 (void) mutex_lock(&sa_global_lock);
1063                 if (sa_global_handles == NULL)
1064                         (void) proto_plugin_fini();
1065                 (void) mutex_unlock(&sa_global_lock);

1067                 sa_scf_fini(handle->scfhandle);
1068                 sa_zfs_fini(handle);
1023                 sa_scf_fini(impl_handle->scfhandle);
1024                 sa_zfs_fini(impl_handle);

1070                 /* Make sure we free the handle */
1071                 free(handle);
1027                 free(impl_handle);

1073         }
1074 }
```

_____***unchanged_portion_omitted_***

```
1147 /*
1148  * sa_get_group(groupname)
1149  *      Return the "group" specified.  If groupname is NULL,
1150  *      return the first group of the list of groups.
1151  */
1152 sa_group_t
1153 sa_get_group(sa_handle_t handle, char *groupname)
1154 {
1155         xmlNodePtr node = NULL;
1156         char *subgroup = NULL;
1157         char *group = NULL;
1114         sa_handle_impl_t impl_handle = (sa_handle_impl_t)handle;

1159         if (handle != NULL && handle->tree != NULL) {
1116         if (impl_handle != NULL && impl_handle->tree != NULL) {
1160                 if (groupname != NULL) {
1161                         group = strdup(groupname);
1162                         if (group != NULL) {
1163                                 subgroup = strchr(group, '/');
1164                                 if (subgroup != NULL)
1165                                         *subgroup++ = '\0';
1166                         }
1167                 }
1168                 /*
1169                  * We want to find the, possibly, named group. If
1170                  * group is not NULL, then lookup the name. If it is
1171                  * NULL, we only do the find if groupname is also
1172                  * NULL. This allows lookup of the "first" group in
1173                  * the internal list.
1174                  */
1175                 if (group != NULL || groupname == NULL)
1176                         node = find_group_by_name(handle->tree,
1133                         node = find_group_by_name(impl_handle->tree,
1177                             (xmlChar *)group);
```



```
1179                 /* if a subgroup, find it before returning */
1180                 if (subgroup != NULL && node != NULL)
1181                         node = find_group_by_name(node, (xmlChar *)subgroup);
1182         }
1183         if (node != NULL && (char *)group != NULL)
1184                 (void) sa_get_instance(handle->scfhandle, (char *)group);
1141                 (void) sa_get_instance(impl_handle->scfhandle, (char *)group);
1185         if (group != NULL)
1186                 free(group);
1187         return ((sa_group_t)(node));
1188 }
```

_____***unchanged_portion_omitted_***

```
1457 /*
1458  * _sa_add_share(group, sharepath, persist, *error, flags)
1459  *
1460  * Common code for all types of add_share. sa_add_share() is the
1461  * public API, we also need to be able to do this when parsing legacy
1462  * files and construction of the internal configuration while
1463  * extracting config info from SMF. "flags" indicates if some
1464  * protocols need relaxed rules while other don't. These values are
1465  * the featureset values defined in libshare.h.
1466  */

1468 sa_share_t
1469 _sa_add_share(sa_group_t group, char *sharepath, int persist, int *error,
1470     uint64_t flags)
1471 {
1472         xmlNodePtr node = NULL;
1473         int err;

1475         err  = SA_OK; /* assume success */

1477         node = xmlNewChild((xmlNodePtr)group, NULL, (xmlChar *)"share", NULL);
1478         if (node == NULL) {
1479                 if (error != NULL)
1480                         *error = SA_NO_MEMORY;
1481                 return (node);
1482         }

1484         (void) xmlSetProp(node, (xmlChar *)"path", (xmlChar *)sharepath);
1485         (void) xmlSetProp(node, (xmlChar *)"type",
1486             persist ? (xmlChar *)"persist" : (xmlChar *)"transient");
1487         if (flags != 0)
1488                 mark_excluded_protos(group, node, flags);
1489         if (persist != SA_SHARE_TRANSIENT) {
1490                 /*
1491                  * persistent shares come in two flavors: SMF and
1492                  * ZFS. Sort this one out based on target group and
1493                  * path type. Both NFS and SMB are supported. First,
1494                  * check to see if the protocol is enabled on the
1495                  * subgroup and then setup the share appropriately.
1496                  */
1497                 if (sa_group_is_zfs(group) &&
1498                     sa_path_is_zfs(sharepath)) {
1499                         if (sa_get_optionset(group, "nfs") != NULL)
1500                                 err = sa_zfs_set_sharenfs(group, sharepath, 1);
1501                         else if (sa_get_optionset(group, "smb") != NULL)
1502                                 err = sa_zfs_set_sharesmb(group, sharepath, 1);
1503                 } else {
1504                         sa_handle_t handle = sa_find_group_handle(group);
1505                         if (handle != NULL) {
1506                                 err = sa_commit_share(handle->scfhandle,
1461                         sa_handle_impl_t impl_handle;
```

```
1462                        impl_handle =
1463                            (sa_handle_impl_t)sa_find_group_handle(group);
1464                        if (impl_handle != NULL) {
1465                                err = sa_commit_share(impl_handle->scfhandle,
1507                                    group, (sa_share_t)node);
1508                        } else {
1509                                err = SA_SYSTEM_ERR;
1510                        }
1511                }
1512        }
1513        if (err == SA_NO_PERMISSION && persist & SA_SHARE_PARSER)
1514                /* called by the dfstab parser so could be a show */
1515                err = SA_OK;

1517        if (err != SA_OK) {
1518                /*
1519                 * we couldn't commit to the repository so undo
1520                 * our internal state to reflect reality.
1521                 */
1522                xmlUnlinkNode(node);
1523                xmlFreeNode(node);
1524                node = NULL;
1525        }

1527        if (error != NULL)
1528                *error = err;

1530        return (node);
1531 }
______unchanged_portion_omitted_

1704 /*
1705  * sa_remove_share(share)
1706  *
1707  * remove the specified share from its containing group.
1708  * Remove from the SMF or ZFS configuration space.
1709  */

1711 int
1712 sa_remove_share(sa_share_t share)
1713 {
1714        sa_group_t group;
1715        int ret = SA_OK;
1716        char *type;
1717        int transient = 0;
1718        char *groupname;
1719        char *zfs;

1721        type = sa_get_share_attr(share, "type");
1722        group = sa_get_parent_group(share);
1723        zfs = sa_get_group_attr(group, "zfs");
1724        groupname = sa_get_group_attr(group, "name");
1725        if (type != NULL && strcmp(type, "persist") != 0)
1726                transient = 1;
1727        if (type != NULL)
1728                sa_free_attr_string(type);

1730        /* remove the node from its group then free the memory */

1732        /*
1733         * need to test if "busy"
1734         */
1735        /* only do SMF action if permanent */
1736        if (!transient || zfs != NULL) {
1737                /* remove from legacy dfstab as well as possible SMF */
1738                ret = sa_delete_legacy(share, NULL);
```



```
1739                if (ret == SA_OK) {
1740                        if (!sa_group_is_zfs(group)) {
1741                                sa_handle_t handle =
1700                                sa_handle_impl_t impl_handle;
1701                                impl_handle = (sa_handle_impl_t)
1742                                    sa_find_group_handle(group);
1743                                if (handle != NULL) {
1703                                if (impl_handle != NULL) {
1744                                        ret = sa_delete_share(
1745                                            handle->scfhandle, group,
1705                                            impl_handle->scfhandle, group,
1746                                            share);
1747                                } else {
1748                                        ret = SA_SYSTEM_ERR;
1749                                }
1750                        } else {
1751                                char *sharepath = sa_get_share_attr(share,
1752                                    "path");
1753                                if (sharepath != NULL) {
1754                                        ret = sa_zfs_set_sharenfs(group,
1755                                            sharepath, 0);
1756                                        sa_free_attr_string(sharepath);
1757                                }
1758                        }
1759                }
1760        }
1761        if (groupname != NULL)
1762                sa_free_attr_string(groupname);
1763        if (zfs != NULL)
1764                sa_free_attr_string(zfs);

1766        xmlUnlinkNode((xmlNodePtr)share);
1767        xmlFreeNode((xmlNodePtr)share);
1768        return (ret);
1769 }

1771 /*
1772  * sa_move_share(group, share)
1773  *
1774  * move the specified share to the specified group.  Update SMF
1775  * appropriately.
1776  */

1778 int
1779 sa_move_share(sa_group_t group, sa_share_t share)
1780 {
1781        sa_group_t oldgroup;
1782        int ret = SA_OK;

1784        /* remove the node from its group then free the memory */

1786        oldgroup = sa_get_parent_group(share);
1787        if (oldgroup != group) {
1788                sa_handle_t handle;
1748                sa_handle_impl_t impl_handle;
1789                xmlUnlinkNode((xmlNodePtr)share);
1790                /*
1791                 * now that the share isn't in its old group, add to
1792                 * the new one
1793                 */
1794                (void) xmlAddChild((xmlNodePtr)group, (xmlNodePtr)share);
1795                /* need to deal with SMF */
1796                handle = sa_find_group_handle(group);
1797                if (handle != NULL) {
1756                impl_handle = (sa_handle_impl_t)sa_find_group_handle(group);
1757                if (impl_handle != NULL) {
```

```
1798                        /*
1799                         * need to remove from old group first and then add to
1800                         * new group. Ideally, we would do the other order but
1801                         * need to avoid having the share in two groups at the
1802                         * same time.
1803                         */
1804                        ret = sa_delete_share(handle->scfhandle, oldgroup,
1764                        ret = sa_delete_share(impl_handle->scfhandle, oldgroup,
1805                            share);
1806                        if (ret == SA_OK)
1807                                ret = sa_commit_share(handle->scfhandle,
1767                                ret = sa_commit_share(impl_handle->scfhandle,
1808                                    group, share);
1809                } else {
1810                        ret = SA_SYSTEM_ERR;
1811                }
1812        }
1813        return (ret);
1814 }
_____unchanged_portion_omitted_

1841 /*
1842  * _sa_create_group(handle, groupname)
1802  * _sa_create_group(impl_handle, groupname)
1843  *
1844  * Create a group in the document. The caller will need to deal with
1845  * configuration store and activation.
1846  */

1848 sa_group_t
1849 _sa_create_group(sa_handle_t handle, char *groupname)
1809 _sa_create_group(sa_handle_impl_t impl_handle, char *groupname)
1850 {
1851        xmlNodePtr node = NULL;

1853        if (sa_valid_group_name(groupname)) {
1854                node = xmlNewChild(handle->tree, NULL, (xmlChar *)"group",
1814                node = xmlNewChild(impl_handle->tree, NULL, (xmlChar *)"group",
1855                    NULL);
1856                if (node != NULL) {
1857                        (void) xmlSetProp(node, (xmlChar *)"name",
1858                            (xmlChar *)groupname);
1859                        (void) xmlSetProp(node, (xmlChar *)"state",
1860                            (xmlChar *)"enabled");
1861                }
1862        }
1863        return ((sa_group_t)node);
1864 }
_____unchanged_portion_omitted_

1889 /*
1890  * sa_create_group(groupname, *error)
1891  *
1892  * Create a new group with groupname.  Need to validate that it is a
1893  * legal name for SMF and the construct the SMF service instance of
1894  * svc:/network/shares/group to implement the group. All necessary
1895  * operational properties must be added to the group at this point
1896  * (via the SMF transaction model).
1897  */
1898 sa_group_t
1899 sa_create_group(sa_handle_t handle, char *groupname, int *error)
1900 {
1901        xmlNodePtr node = NULL;
1902        sa_group_t group;
1903        int ret;
1904        char rbacstr[SA_STRSIZE];
```



```
1865        sa_handle_impl_t impl_handle = (sa_handle_impl_t)handle;

1906        ret = SA_OK;

1908        if (handle == NULL || handle->scfhandle == NULL) {
1869        if (impl_handle == NULL || impl_handle->scfhandle == NULL) {
1909                ret = SA_SYSTEM_ERR;
1910                goto err;
1911        }

1913        group = sa_get_group(handle, groupname);
1914        if (group != NULL) {
1915                ret = SA_DUPLICATE_NAME;
1916        } else {
1917                if (sa_valid_group_name(groupname)) {
1918                        node = xmlNewChild(handle->tree, NULL,
1879                        node = xmlNewChild(impl_handle->tree, NULL,
1919                            (xmlChar *)"group", NULL);
1920                        if (node != NULL) {
1921                                (void) xmlSetProp(node, (xmlChar *)"name",
1922                                    (xmlChar *)groupname);
1923                                /* default to the group being enabled */
1924                                (void) xmlSetProp(node, (xmlChar *)"state",
1925                                    (xmlChar *)"enabled");
1926                                ret = sa_create_instance(handle->scfhandle,
1887                                ret = sa_create_instance(impl_handle->scfhandle,
1927                                    groupname);
1928                                if (ret == SA_OK) {
1929                                        ret = sa_start_transaction(
1930                                            handle->scfhandle,
1891                                            impl_handle->scfhandle,
1931                                            "operation");
1932                                }
1933                                if (ret == SA_OK) {
1934                                        ret = sa_set_property(
1935                                            handle->scfhandle,
1896                                            impl_handle->scfhandle,
1936                                            "state", "enabled");
1937                                        if (ret == SA_OK) {
1938                                                ret = sa_end_transaction(
1939                                                    handle->scfhandle,
1940                                                    handle);
1900                                                    impl_handle->scfhandle,
1901                                                    impl_handle);
1941                                        } else {
1942                                                sa_abort_transaction(
1943                                                    handle->scfhandle);
1904                                                    impl_handle->scfhandle);
1944                                        }
1945                                }
1946                                if (ret == SA_OK) {
1947                                        /* initialize the RBAC strings */
1948                                        ret = sa_start_transaction(
1949                                            handle->scfhandle,
1910                                            impl_handle->scfhandle,
1950                                            "general");
1951                                        if (ret == SA_OK) {
1952                                                (void) snprintf(rbacstr,
1953                                                    sizeof (rbacstr), "%s.%s",
1954                                                    SA_RBAC_MANAGE, groupname);
1955                                                ret = sa_set_property(
1956                                                    handle->scfhandle,
1917                                                    impl_handle->scfhandle,
1957                                                    "action_authorization",
1958                                                    rbacstr);
1959                                        }
```

```
1960                                    if (ret == SA_OK) {
1961                                            (void) snprintf(rbacstr,
1962                                                sizeof (rbacstr), "%s.%s",
1963                                                SA_RBAC_VALUE, groupname);
1964                                            ret = sa_set_property(
1965                                                handle->scfhandle,
1926                                                impl_handle->scfhandle,
1966                                                "value_authorization",
1967                                                rbacstr);
1968                                    }
1969                                    if (ret == SA_OK) {
1970                                            ret = sa_end_transaction(
1971                                                handle->scfhandle,
1972                                                handle);
1932                                                impl_handle->scfhandle,
1933                                                impl_handle);
1973                                    } else {
1974                                            sa_abort_transaction(
1975                                                handle->scfhandle);
1936                                                impl_handle->scfhandle);
1976                                    }
1977                            }
1978                            if (ret != SA_OK) {
1979                                    /*
1980                                     * Couldn't commit the group
1981                                     * so we need to undo
1982                                     * internally.
1983                                     */
1984                                    xmlUnlinkNode(node);
1985                                    xmlFreeNode(node);
1986                                    node = NULL;
1987                            }
1988                    } else {
1989                            ret = SA_NO_MEMORY;
1990                    }
1991            } else {
1992                    ret = SA_INVALID_NAME;
1993            }
1994    }
1995 err:
1996    if (error != NULL)
1997            *error = ret;
1998    return ((sa_group_t)node);
1999 }

2001 /*
2002  * sa_remove_group(group)
2003  *
2004  * Remove the specified group. This deletes from the SMF repository.
2005  * All property groups and properties are removed.
2006  */

2008 int
2009 sa_remove_group(sa_group_t group)
2010 {
2011    char *name;
2012    int ret = SA_OK;
2013    sa_handle_t handle;
1974    sa_handle_impl_t impl_handle;

2015    handle = sa_find_group_handle(group);
2016    if (handle != NULL) {
1976    impl_handle = (sa_handle_impl_t)sa_find_group_handle(group);
1977    if (impl_handle != NULL) {
2017            name = sa_get_group_attr(group, "name");
2018            if (name != NULL) {
```



```
2019                    ret = sa_delete_instance(handle->scfhandle, name);
1980                    ret = sa_delete_instance(impl_handle->scfhandle, name);
2020                    sa_free_attr_string(name);
2021            }
2022            xmlUnlinkNode((xmlNodePtr)group); /* make sure unlinked */
2023            xmlFreeNode((xmlNodePtr)group);   /* now it is gone */
2024    } else {
2025            ret = SA_SYSTEM_ERR;
2026    }
2027    return (ret);
2028 }
______unchanged_portion_omitted_

2101 /*
2102  * sa_set_group_attr(group, tag, value)
2103  *
2104  * set the specified tag/attribute on the group using value as its
2105  * value.
2106  *
2107  * This will result in setting the property in the SMF repository as
2108  * well as in the internal document.
2109  */

2111 int
2112 sa_set_group_attr(sa_group_t group, char *tag, char *value)
2113 {
2114    int ret;
2115    char *groupname;
2116    sa_handle_t handle;
2077    sa_handle_impl_t impl_handle;

2118    /*
2119     * ZFS group/subgroup doesn't need the handle so shortcut.
2120     */
2121    if (sa_group_is_zfs(group)) {
2122            set_node_attr((void *)group, tag, value);
2123            return (SA_OK);
2124    }

2126    handle = sa_find_group_handle(group);
2127    if (handle != NULL) {
2087    impl_handle = (sa_handle_impl_t)sa_find_group_handle(group);
2088    if (impl_handle != NULL) {
2128            groupname = sa_get_group_attr(group, "name");
2129            ret = sa_get_instance(handle->scfhandle, groupname);
2090            ret = sa_get_instance(impl_handle->scfhandle, groupname);
2130            if (ret == SA_OK) {
2131                    set_node_attr((void *)group, tag, value);
2132                    ret = sa_start_transaction(handle->scfhandle,
2093                    ret = sa_start_transaction(impl_handle->scfhandle,
2133                        "operation");
2134                    if (ret == SA_OK) {
2135                            ret = sa_set_property(handle->scfhandle,
2096                            ret = sa_set_property(impl_handle->scfhandle,
2136                                tag, value);
2137                            if (ret == SA_OK)
2138                                    ret = sa_end_transaction(
2139                                        handle->scfhandle,
2140                                        handle);
2100                                    impl_handle->scfhandle,
2101                                    impl_handle);
2141                            else
2142                                    sa_abort_transaction(
2143                                        handle->scfhandle);
2104                                    impl_handle->scfhandle);
2144                    }
```

```
2145                        if (ret == SA_SYSTEM_ERR)
2146                                ret = SA_NO_PERMISSION;
2147                }
2148                if (groupname != NULL)
2149                        sa_free_attr_string(groupname);
2150        } else {
2151                ret = SA_SYSTEM_ERR;
2152        }
2153        return (ret);
2154 }
```

_____unchanged_portion_omitted_

```
2187 /*
2188  * sa_set_share_attr(share, tag, value)
2189  *
2190  * Set the share attribute specified by tag to the specified value. In
2191  * the case of "resource", enforce a no duplicates in a group rule. If
2192  * the share is not transient, commit the changes to the repository
2193  * else just update the share internally.
2194  */

2196 int
2197 sa_set_share_attr(sa_share_t share, char *tag, char *value)
2198 {
2199        sa_group_t group;
2200        sa_share_t resource;
2201        int ret = SA_OK;

2203        group = sa_get_parent_group(share);

2205        /*
2206         * There are some attributes that may have specific
2207         * restrictions on them. Initially, only "resource" has
2208         * special meaning that needs to be checked. Only one instance
2209         * of a resource name may exist within a group.
2210         */

2212        if (strcmp(tag, "resource") == 0) {
2213                resource = sa_get_resource(group, value);
2214                if (resource != share && resource != NULL)
2215                        ret = SA_DUPLICATE_NAME;
2216        }
2217        if (ret == SA_OK) {
2218                set_node_attr((void *)share, tag, value);
2219                if (group != NULL) {
2220                        char *type;
2221                        /* we can probably optimize this some */
2222                        type = sa_get_share_attr(share, "type");
2223                        if (type == NULL || strcmp(type, "transient") != 0) {
2224                                sa_handle_t handle = sa_find_group_handle(
2185                                sa_handle_impl_t impl_handle;
2186                                impl_handle =
2187                                    (sa_handle_impl_t)sa_find_group_handle(
2225                                    group);
2226                                if (handle != NULL) {
2189                                if (impl_handle != NULL) {
2227                                        ret = sa_commit_share(
2228                                            handle->scfhandle, group,
2191                                            impl_handle->scfhandle, group,
2229                                            share);
2230                                } else {
2231                                        ret = SA_SYSTEM_ERR;
2232                                }
2233                        }
2234                        if (type != NULL)
2235                                sa_free_attr_string(type);
```



```
2236                }
2237        }
2238        return (ret);
2239 }
```

_____unchanged_portion_omitted_

```
2489 /*
2490  * sa_set_share_description(share, content)
2491  *
2492  * Set the description of share to content.
2493  */

2495 int
2496 sa_set_share_description(sa_share_t share, char *content)
2497 {
2498        xmlNodePtr node;
2499        sa_group_t group;
2500        int ret = SA_OK;

2502        for (node = ((xmlNodePtr)share)->children; node != NULL;
2503            node = node->next) {
2504                if (xmlStrcmp(node->name, (xmlChar *)"description") == 0) {
2505                        break;
2506                }
2507        }
2508        /* no existing description but want to add */
2509        if (node == NULL && content != NULL) {
2510                /* add a description */
2511                node = _sa_set_share_description(share, content);
2512        } else if (node != NULL && content != NULL) {
2513                /* update a description */
2514                xmlNodeSetContent(node, (xmlChar *)content);
2515        } else if (node != NULL && content == NULL) {
2516                /* remove an existing description */
2517                xmlUnlinkNode(node);
2518                xmlFreeNode(node);
2519        }
2520        group = sa_get_parent_group(share);
2521        if (group != NULL &&
2522            sa_is_persistent(share) && (!sa_group_is_zfs(group))) {
2523                sa_handle_t handle = sa_find_group_handle(group);
2524                if (handle != NULL) {
2525                        ret = sa_commit_share(handle->scfhandle, group,
2486                        sa_handle_impl_t impl_handle;
2487                        impl_handle = (sa_handle_impl_t)sa_find_group_handle(group);
2488                        if (impl_handle != NULL) {
2489                                ret = sa_commit_share(impl_handle->scfhandle, group,
2526                                    share);
2527                        } else {
2528                                ret = SA_SYSTEM_ERR;
2529                        }
2530        }
2531        return (ret);
2532 }
```

_____unchanged_portion_omitted_

```
2590 /*
2591  * sa_create_optionset(group, proto)
2592  *
2593  * Create an optionset for the specified protocol in the specied
2594  * group. This is manifested as a property group within SMF.
2595  */

2597 sa_optionset_t
2598 sa_create_optionset(sa_group_t group, char *proto)
2599 {
```

```
2600         sa_optionset_t optionset;
2601         sa_group_t parent = group;
2602         sa_share_t share = NULL;
2603         int err = SA_OK;
2604         char *id = NULL;

2606         optionset = sa_get_optionset(group, proto);
2607         if (optionset != NULL) {
2608                 /* can't have a duplicate protocol */
2609                 optionset = NULL;
2610         } else {
2611                 /*
2612                  * Account for resource names being slightly
2613                  * different.
2614                  */
2615                 if (sa_is_share(group)) {
2616                         /*
2617                          * Transient shares do not have an "id" so not an
2618                          * error to not find one.
2619                          */
2620                         id = sa_get_share_attr((sa_share_t)group, "id");
2621                 } else if (sa_is_resource(group)) {
2622                         share = sa_get_resource_parent(
2623                             (sa_resource_t)group);
2624                         id = sa_get_resource_attr(share, "id");

2626                         /* id can be NULL if the group is transient (ZFS) */
2627                         if (id == NULL && sa_is_persistent(group))
2628                                 err = SA_NO_MEMORY;
2629                 }
2630                 if (err == SA_NO_MEMORY) {
2631                         /*
2632                          * Couldn't get the id for the share or
2633                          * resource. While this could be a
2634                          * configuration issue, it is most likely an
2635                          * out of memory. In any case, fail the create.
2636                          */
2637                         return (NULL);
2638                 }

2640                 optionset = (sa_optionset_t)xmlNewChild((xmlNodePtr)group,
2641                     NULL, (xmlChar *)"optionset", NULL);
2642                 /*
2643                  * only put to repository if on a group and we were
2644                  * able to create an optionset.
2645                  */
2646                 if (optionset != NULL) {
2647                         char oname[SA_STRSIZE];
2648                         char *groupname;

2650                         /*
2651                          * Need to get parent group in all cases, but also get
2652                          * the share if this is a resource.
2653                          */
2654                         if (sa_is_share(group)) {
2655                                 parent = sa_get_parent_group((sa_share_t)group);
2656                         } else if (sa_is_resource(group)) {
2657                                 share = sa_get_resource_parent(
2658                                     (sa_resource_t)group);
2659                                 parent = sa_get_parent_group(share);
2660                         }

2662                         sa_set_optionset_attr(optionset, "type", proto);

2664                         (void) sa_optionset_name(optionset, oname,
2665                             sizeof (oname), id);
```



```
2666                         groupname = sa_get_group_attr(parent, "name");
2667                         if (groupname != NULL && sa_is_persistent(group)) {
2668                                 sa_handle_t handle = sa_find_group_handle(
2632                                 sa_handle_impl_t impl_handle;
2633                                 impl_handle =
2634                                     (sa_handle_impl_t)sa_find_group_handle(
2669                                     group);
2670                                 assert(handle != NULL);
2671                                 if (handle != NULL) {
2636                                 assert(impl_handle != NULL);
2637                                 if (impl_handle != NULL) {
2672                                         (void) sa_get_instance(
2673                                             handle->scfhandle, groupname);
2639                                             impl_handle->scfhandle, groupname);
2674                                         (void) sa_create_pgroup(
2675                                             handle->scfhandle, oname);
2641                                             impl_handle->scfhandle, oname);
2676                                 }
2677                         }
2678                         if (groupname != NULL)
2679                                 sa_free_attr_string(groupname);
2680                 }
2681         }

2683         if (id != NULL)
2684                 sa_free_attr_string(id);
2685         return (optionset);
2686 }
______unchanged_portion_omitted_

2759 /*
2760  * sa_commit_properties(optionset, clear)
2761  *
2762  * Check if SMF or ZFS config and either update or abort the pending
2763  * changes.
2764  */

2766 int
2767 sa_commit_properties(sa_optionset_t optionset, int clear)
2768 {
2769         sa_group_t group;
2770         sa_group_t parent;
2771         int zfs = 0;
2772         int needsupdate = 0;
2773         int ret = SA_OK;
2774         sa_handle_t handle;
2740         sa_handle_impl_t impl_handle;

2776         group = sa_get_optionset_parent(optionset);
2777         if (group != NULL && (sa_is_share(group) || is_zfs_group(group))) {
2778                 /* only update ZFS if on a share */
2779                 parent = sa_get_parent_group(group);
2780                 zfs++;
2781                 if (parent != NULL && is_zfs_group(parent))
2782                         needsupdate = zfs_needs_update(group);
2783                 else
2784                         zfs = 0;
2785         }
2786         if (zfs) {
2787                 if (!clear && needsupdate)
2788                         ret = sa_zfs_update((sa_share_t)group);
2789         } else {
2790                 handle = sa_find_group_handle(group);
2791                 if (handle != NULL) {
2756                 impl_handle = (sa_handle_impl_t)sa_find_group_handle(group);
2757                 if (impl_handle != NULL) {
```

```
2792                        if (clear) {
2793                                (void) sa_abort_transaction(
2794                                    handle->scfhandle);
2760                                    impl_handle->scfhandle);
2795                        } else {
2796                                ret = sa_end_transaction(
2797                                    handle->scfhandle, handle);
2763                                    impl_handle->scfhandle, impl_handle);
2798                        }
2799                } else {
2800                        ret = SA_SYSTEM_ERR;
2801                }
2802        }
2803        return (ret);
2804 }

2806 /*
2807  * sa_destroy_optionset(optionset)
2808  *
2809  * Remove the optionset from its group. Update the repository to
2810  * reflect this change.
2811  */

2813 int
2814 sa_destroy_optionset(sa_optionset_t optionset)
2815 {
2816         char name[SA_STRSIZE];
2817         int len;
2818         int ret;
2819         char *id = NULL;
2820         sa_group_t group;
2821         int ispersist = 1;

2823         /* now delete the prop group */
2824         group = sa_get_optionset_parent(optionset);
2825         if (group != NULL) {
2826                 if (sa_is_resource(group)) {
2827                         sa_resource_t resource = group;
2828                         sa_share_t share = sa_get_resource_parent(resource);
2829                         group = sa_get_parent_group(share);
2830                         id = sa_get_share_attr(share, "id");
2831                 } else if (sa_is_share(group)) {
2832                         id = sa_get_share_attr((sa_share_t)group, "id");
2833                 }
2834                 ispersist = sa_is_persistent(group);
2835         }
2836         if (ispersist) {
2837                 sa_handle_t handle = sa_find_group_handle(group);
2803                 sa_handle_impl_t impl_handle;
2838                 len = sa_optionset_name(optionset, name, sizeof (name), id);
2839                 if (handle != NULL) {
2805                 impl_handle = (sa_handle_impl_t)sa_find_group_handle(group);
2806                 if (impl_handle != NULL) {
2840                         if (len > 0) {
2841                                 ret = sa_delete_pgroup(handle->scfhandle,
2808                                 ret = sa_delete_pgroup(impl_handle->scfhandle,
2842                                     name);
2843                         }
2844                 } else {
2845                         ret = SA_SYSTEM_ERR;
2846                 }
2847         }
2848         xmlUnlinkNode((xmlNodePtr)optionset);
2849         xmlFreeNode((xmlNodePtr)optionset);
2850         if (id != NULL)
2851                 sa_free_attr_string(id);
```



```
2852         return (ret);
2853 }
_____unchanged_portion_omitted_

2866 /*
2867  * sa_create_security(group, sectype, proto)
2868  *
2869  * Create a security optionset (one that has a type name and a
2870  * proto). Security is left over from a pure NFS implementation. The
2871  * naming will change in the future when the API is released.
2872  */
2873 sa_security_t
2874 sa_create_security(sa_group_t group, char *sectype, char *proto)
2875 {
2876         sa_security_t security;
2877         char *id = NULL;
2878         sa_group_t parent;
2879         char *groupname = NULL;

2881         if (group != NULL && sa_is_share(group)) {
2882                 id = sa_get_share_attr((sa_share_t)group, "id");
2883                 parent = sa_get_parent_group(group);
2884                 if (parent != NULL)
2885                         groupname = sa_get_group_attr(parent, "name");
2886         } else if (group != NULL) {
2887                 groupname = sa_get_group_attr(group, "name");
2888         }

2890         security = sa_get_security(group, sectype, proto);
2891         if (security != NULL) {
2892                 /* can't have a duplicate security option */
2893                 security = NULL;
2894         } else {
2895                 security = (sa_security_t)xmlNewChild((xmlNodePtr)group,
2896                     NULL, (xmlChar *)"security", NULL);
2897                 if (security != NULL) {
2898                         char oname[SA_STRSIZE];
2899                         sa_set_security_attr(security, "type", proto);

2901                         sa_set_security_attr(security, "sectype", sectype);
2902                         (void) sa_security_name(security, oname,
2903                             sizeof (oname), id);
2904                         if (groupname != NULL && sa_is_persistent(group)) {
2905                                 sa_handle_t handle = sa_find_group_handle(
2872                                 sa_handle_impl_t impl_handle;
2873                                 impl_handle =
2874                                     (sa_handle_impl_t)sa_find_group_handle(
2906                                     group);
2907                                 if (handle != NULL) {
2876                                 if (impl_handle != NULL) {
2908                                         (void) sa_get_instance(
2909                                             handle->scfhandle, groupname);
2878                                             impl_handle->scfhandle, groupname);
2910                                         (void) sa_create_pgroup(
2911                                             handle->scfhandle, oname);
2880                                             impl_handle->scfhandle, oname);
2912                                 }
2913                         }
2914                 }
2915         }
2916         if (id != NULL)
2917                 sa_free_attr_string(id);
2918         if (groupname != NULL)
2919                 sa_free_attr_string(groupname);
2920         return (security);
2921 }
```

```
2923 /*
2924  * sa_destroy_security(security)
2925  *
2926  * Remove the specified optionset from the document and the
2927  * configuration.
2928  */

2930 int
2931 sa_destroy_security(sa_security_t security)
2932 {
2933         char name[SA_STRSIZE];
2934         int len;
2935         int ret = SA_OK;
2936         char *id = NULL;
2937         sa_group_t group;
2938         int iszfs = 0;
2939         int ispersist = 1;

2941         group = sa_get_optionset_parent(security);

2943         if (group != NULL)
2944                 iszfs = sa_group_is_zfs(group);

2946         if (group != NULL && !iszfs) {
2947                 if (sa_is_share(group))
2948                         ispersist = sa_is_persistent(group);
2949                 id = sa_get_share_attr((sa_share_t)group, "id");
2950         }
2951         if (ispersist) {
2952                 len = sa_security_name(security, name, sizeof (name), id);
2953                 if (!iszfs && len > 0) {
2954                         sa_handle_t handle = sa_find_group_handle(group);
2955                         if (handle != NULL) {
2956                                 ret = sa_delete_pgroup(handle->scfhandle,
2923                         sa_handle_impl_t impl_handle;
2924                         impl_handle =
2925                             (sa_handle_impl_t)sa_find_group_handle(group);
2926                         if (impl_handle != NULL) {
2927                                 ret = sa_delete_pgroup(impl_handle->scfhandle,
2957                                     name);
2958                         } else {
2959                                 ret = SA_SYSTEM_ERR;
2960                         }
2961                 }
2962         }
2963         xmlUnlinkNode((xmlNodePtr)security);
2964         xmlFreeNode((xmlNodePtr)security);
2965         if (iszfs)
2966                 ret = sa_zfs_update(group);
2967         if (id != NULL)
2968                 sa_free_attr_string(id);
2969         return (ret);
2970 }
```

_____***unchanged_portion_omitted_***

```
3042 /*
3043  * sa_set_prop_by_prop(optionset, group, prop, type)
3044  *
3045  * Add/remove/update the specified property prop into the optionset or
3046  * share. If a share, sort out which property group based on GUID. In
3047  * all cases, the appropriate transaction is set (or ZFS share is
3048  * marked as needing an update)
3049  */

3051 static int
```



```
3052 sa_set_prop_by_prop(sa_optionset_t optionset, sa_group_t group,
3053                         sa_property_t prop, int type)
3054 {
3055         char *name;
3056         char *valstr;
3057         int ret = SA_OK;
3058         scf_transaction_entry_t *entry;
3059         scf_value_t *value;
3060         int opttype; /* 1 == optionset, 0 == security */
3061         char *id = NULL;
3062         int iszfs = 0;
3063         sa_group_t parent = NULL;
3064         sa_share_t share = NULL;
3065         sa_handle_t handle;
3036         sa_handle_impl_t impl_handle;
3066         scfutilhandle_t  *scf_handle;

3068         if (!sa_is_persistent(group)) {
3069                 /*
3070                  * if the group/share is not persistent we don't need
3071                  * to do anything here
3072                  */
3073                 return (SA_OK);
3074         }
3075         handle = sa_find_group_handle(group);
3076         if (handle == NULL || handle->scfhandle == NULL)
3046         impl_handle = (sa_handle_impl_t)sa_find_group_handle(group);
3047         if (impl_handle == NULL || impl_handle->scfhandle == NULL)
3077                 return (SA_SYSTEM_ERR);
3078         scf_handle = handle->scfhandle;
3049         scf_handle = impl_handle->scfhandle;
3079         name = sa_get_property_attr(prop, "type");
3080         valstr = sa_get_property_attr(prop, "value");
3081         entry = scf_entry_create(scf_handle->handle);
3082         opttype = is_nodetype((void *)optionset, "optionset");

3084         /*
3085          * Check for share vs. resource since they need slightly
3086          * different treatment given the hierarchy.
3087          */
3088         if (valstr != NULL && entry != NULL) {
3089                 if (sa_is_share(group)) {
3090                         parent = sa_get_parent_group(group);
3091                         share = (sa_share_t)group;
3092                         if (parent != NULL)
3093                                 iszfs = is_zfs_group(parent);
3094                 } else if (sa_is_resource(group)) {
3095                         share = sa_get_parent_group(group);
3096                         if (share != NULL)
3097                                 parent = sa_get_parent_group(share);
3098                 } else {
3099                         iszfs = is_zfs_group(group);
3100                 }
3101                 if (!iszfs) {
3102                         if (scf_handle->trans == NULL) {
3103                                 char oname[SA_STRSIZE];
3104                                 char *groupname = NULL;
3105                                 if (share != NULL) {
3106                                         if (parent != NULL)
3107                                                 groupname =
3108                                                     sa_get_group_attr(parent,
3109                                                     "name");
3110                                         id = sa_get_share_attr(
3111                                             (sa_share_t)share, "id");
3112                                 } else {
3113                                         groupname = sa_get_group_attr(group,
```

```
3114                                        "name");
3115                                }
3116                                if (groupname != NULL) {
3117                                        ret = sa_get_instance(scf_handle,
3118                                            groupname);
3119                                        sa_free_attr_string(groupname);
3120                                }
3121                                if (opttype)
3122                                        (void) sa_optionset_name(optionset,
3123                                            oname, sizeof (oname), id);
3124                                else
3125                                        (void) sa_security_name(optionset,
3126                                            oname, sizeof (oname), id);
3127                                ret = sa_start_transaction(scf_handle, oname);
3128                                if (id != NULL)
3129                                        sa_free_attr_string(id);
3130                        }
3131                        if (ret == SA_OK) {
3132                                switch (type) {
3133                                case SA_PROP_OP_REMOVE:
3134                                        ret = scf_transaction_property_delete(
3135                                            scf_handle->trans, entry, name);
3136                                        break;
3137                                case SA_PROP_OP_ADD:
3138                                case SA_PROP_OP_UPDATE:
3139                                        value = scf_value_create(
3140                                            scf_handle->handle);
3141                                        ret = add_or_update(scf_handle, type,
3142                                            value, entry, name, valstr);
3143                                        break;
3144                                }
3145                        }
3146                } else {
3147                        /*
3148                         * ZFS update. The calling function would have updated
3149                         * the internal XML structure. Just need to flag it as
3150                         * changed for ZFS.
3151                         */
3152                        zfs_set_update((sa_share_t)group);
3153                }
3154        }

3156        if (name != NULL)
3157                sa_free_attr_string(name);
3158        if (valstr != NULL)
3159                sa_free_attr_string(valstr);
3160        else if (entry != NULL)
3161                scf_entry_destroy(entry);

3163        if (ret == -1)
3164                ret = SA_SYSTEM_ERR;

3166        return (ret);
3167 }
```

*_____unchanged_portion_omitted_*

```
3217 /*
3218  * sa_add_property(object, property)
3219  *
3220  * Add the specified property to the object. Issue the appropriate
3221  * transaction or mark a ZFS object as needing an update.
3222  */

3224 int
3225 sa_add_property(void *object, sa_property_t property)
3226 {
```



```
3227        int ret = SA_OK;
3228        sa_group_t parent;
3229        sa_group_t group;
3230        char *proto;

3232        if (property != NULL) {
3233                sa_handle_t handle;
3234                handle = sa_find_group_handle((sa_group_t)object);
3235                /* It is legitimate to not find a handle */
3236                proto = sa_get_optionset_attr(object, "type");
3237                if ((ret = sa_valid_property(handle, object, proto,
3238                    property)) == SA_OK) {
3239                        property = (sa_property_t)xmlAddChild(
3240                            (xmlNodePtr)object, (xmlNodePtr)property);
3241                } else {
3242                        if (proto != NULL)
3243                                sa_free_attr_string(proto);
3244                        return (ret);
3245                }
3246                if (proto != NULL)
3247                        sa_free_attr_string(proto);
3248        }


3251        parent = sa_get_parent_group(object);
3252        if (!sa_is_persistent(parent))
3253                return (ret);

3255        if (sa_is_resource(parent)) {
3256                /*
3257                 * Resources are children of share.  Need to go up two
3258                 * levels to find the group but the parent needs to be
3259                 * the share at this point in order to get the "id".
3260                 */
3261                parent = sa_get_parent_group(parent);
3262                group = sa_get_parent_group(parent);
3263        } else if (sa_is_share(parent)) {
3264                group = sa_get_parent_group(parent);
3265        } else {
3266                group = parent;
3267        }

3269        if (property == NULL) {
3270                ret = SA_NO_MEMORY;
3271        } else {
3272                char oname[SA_STRSIZE];

3274                if (!is_zfs_group(group)) {
3275                        char *id = NULL;
3276                        sa_handle_t handle;
3247                        sa_handle_impl_t impl_handle;
3277                        scfutilhandle_t  *scf_handle;

3279                        handle = sa_find_group_handle(group);
3280                        if (handle == NULL ||
3281                            handle->scfhandle == NULL)
3250                        impl_handle = (sa_handle_impl_t)sa_find_group_handle(
3251                            group);
3252                        if (impl_handle == NULL ||
3253                            impl_handle->scfhandle == NULL)
3282                                ret = SA_SYSTEM_ERR;
3283                        if (ret == SA_OK) {
3284                                scf_handle = handle->scfhandle;
3256                                scf_handle = impl_handle->scfhandle;
3285                                if (sa_is_share((sa_group_t)parent)) {
3286                                        id = sa_get_share_attr(
```

```
3287                                        (sa_share_t)parent, "id");
3288                                }
3289                                if (scf_handle->trans == NULL) {
3290                                        if (is_nodetype(object, "optionset")) {
3291                                                (void) sa_optionset_name(
3292                                                    (sa_optionset_t)object,
3293                                                    oname, sizeof (oname), id);
3294                                        } else {
3295                                                (void) sa_security_name(
3296                                                    (sa_optionset_t)object,
3297                                                    oname, sizeof (oname), id);
3298                                        }
3299                                        ret = sa_start_transaction(scf_handle,
3300                                            oname);
3301                                }
3302                                if (ret == SA_OK) {
3303                                        char *name;
3304                                        char *value;
3305                                        name = sa_get_property_attr(property,
3306                                            "type");
3307                                        value = sa_get_property_attr(property,
3308                                            "value");
3309                                        if (name != NULL && value != NULL) {
3310                                                if (scf_handle->scf_state ==
3311                                                    SCH_STATE_INIT) {
3312                                                        ret = sa_set_property(
3313                                                            scf_handle, name,
3314                                                            value);
3315                                                }
3316                                        } else {
3317                                                ret = SA_CONFIG_ERR;
3318                                        }
3319                                        if (name != NULL)
3320                                                sa_free_attr_string(
3321                                                    name);
3322                                        if (value != NULL)
3323                                                sa_free_attr_string(value);
3324                                }
3325                                if (id != NULL)
3326                                        sa_free_attr_string(id);
3327                        }
3328                } else {
3329                        /*
3330                         * ZFS is a special case. We do want
3331                         * to allow editing property/security
3332                         * lists since we can have a better
3333                         * syntax and we also want to keep
3334                         * things consistent when possible.
3335                         *
3336                         * Right now, we defer until the
3337                         * sa_commit_properties so we can get
3338                         * them all at once. We do need to
3339                         * mark the share as "changed"
3340                         */
3341                        zfs_set_update((sa_share_t)parent);
3342                }
3343        }
3344        return (ret);
3345 }
______unchanged_portion_omitted_


3739 /*
3740  * sa_add_resource(share, resource, persist, &err)
3741  *
3742  * Adds a new resource name associated with share. The resource name
```



```
3743  * must be unique in the system and will be case insensitive (eventually).
3744  */

3746 sa_resource_t
3747 sa_add_resource(sa_share_t share, char *resource, int persist, int *error)
3748 {
3749        xmlNodePtr node;
3750        int err = SA_OK;
3751        sa_resource_t res;
3752        sa_group_t group;
3753        sa_handle_t handle;
3754        char istring[8]; /* just big enough for an integer value */
3755        int index;

3757        group = sa_get_parent_group(share);
3758        handle = sa_find_group_handle(group);
3759        res = sa_find_resource(handle, resource);
3760        if (res != NULL) {
3761                err = SA_DUPLICATE_NAME;
3762                res = NULL;
3763        } else {
3764                node = xmlNewChild((xmlNodePtr)share, NULL,
3765                    (xmlChar *)"resource", NULL);
3766                if (node != NULL) {
3767                        (void) xmlSetProp(node, (xmlChar *)"name",
3768                            (xmlChar *)resource);
3769                        (void) xmlSetProp(node, (xmlChar *)"type", persist ?
3770                            (xmlChar *)"persist" : (xmlChar *)"transient");
3771                        if (persist != SA_SHARE_TRANSIENT) {
3772                                index = _sa_get_next_resource_index(share);
3773                                (void) snprintf(istring, sizeof (istring), "%d",
3774                                    index);
3775                                (void) xmlSetProp(node, (xmlChar *)"id",
3776                                    (xmlChar *)istring);

3778                                if (!sa_is_persistent((sa_group_t)share))
3779                                        goto done;

3781                                if (!sa_group_is_zfs(group)) {
3782                                        /* ZFS doesn't use resource names */
3783                                        sa_handle_t handle;
3755                                        sa_handle_impl_t ihandle;

3785                                        handle = sa_find_group_handle(
3757                                        ihandle = (sa_handle_impl_t)
3758                                            sa_find_group_handle(
3786                                            group);
3787                                        if (handle != NULL)
3760                                        if (ihandle != NULL)
3788                                                err = sa_commit_share(
3789                                                    handle->scfhandle, group,
3762                                                    ihandle->scfhandle, group,
3790                                                    share);
3791                                        else
3792                                                err = SA_SYSTEM_ERR;
3793                                } else {
3794                                        err = sa_zfs_update((sa_share_t)group);
3795                                }
3796                        }
3797                }
3798        }
3799 done:
3800        if (error != NULL)
3801                *error = err;
3802        return ((sa_resource_t)node);
3803 }
```

```
3805 /*
3806  * sa_remove_resource(resource)
3807  *
3808  * Remove the resource name from the share (and the system)
3809  */

3811 int
3812 sa_remove_resource(sa_resource_t resource)
3813 {
3814         sa_share_t share;
3815         sa_group_t group;
3816         char *type;
3817         int ret = SA_OK;
3818         boolean_t transient = B_FALSE;
3819         sa_optionset_t opt;

3821         share = sa_get_resource_parent(resource);
3822         type = sa_get_share_attr(share, "type");
3823         group = sa_get_parent_group(share);


3826         if (type != NULL) {
3827                 if (strcmp(type, "persist") != 0)
3828                         transient = B_TRUE;
3829                 sa_free_attr_string(type);
3830         }

3832         /* Disable the resource for all protocols. */
3833         (void) sa_disable_resource(resource, NULL);

3835         /* Remove any optionsets from the resource. */
3836         for (opt = sa_get_optionset(resource, NULL);
3837             opt != NULL;
3838             opt = sa_get_next_optionset(opt))
3839                 (void) sa_destroy_optionset(opt);

3841         /* Remove from the share */
3842         xmlUnlinkNode((xmlNode *)resource);
3843         xmlFreeNode((xmlNode *)resource);

3845         /* only do SMF action if permanent and not ZFS */
3846         if (transient)
3847                 return (ret);

3849         if (!sa_group_is_zfs(group)) {
3850                 sa_handle_t handle = sa_find_group_handle(group);
3851                 if (handle != NULL)
3852                         ret = sa_commit_share(handle->scfhandle, group, share);
3823                 sa_handle_impl_t ihandle;
3824                 ihandle = (sa_handle_impl_t)sa_find_group_handle(group);
3825                 if (ihandle != NULL)
3826                         ret = sa_commit_share(ihandle->scfhandle, group, share);
3853                 else
3854                         ret = SA_SYSTEM_ERR;
3855         } else {
3856                 ret = sa_zfs_update((sa_share_t)group);
3857         }

3859         return (ret);
3860 }
```

_____***unchanged_portion_omitted_***

```
3892 /*
3893  * sa_rename_resource(resource, newname)
3894  *
```



```
3895  * Rename the resource to the new name, if it is unique.
3896  */

3898 int
3899 sa_rename_resource(sa_resource_t resource, char *newname)
3900 {
3901         sa_share_t share;
3902         sa_group_t group = NULL;
3903         sa_resource_t target;
3904         int ret = SA_CONFIG_ERR;
3905         sa_handle_t handle = NULL;

3907         share = sa_get_resource_parent(resource);
3908         if (share == NULL)
3909                 return (ret);

3911         group = sa_get_parent_group(share);
3912         if (group == NULL)
3913                 return (ret);

3915         handle = sa_find_group_handle(group);
3889         handle = (sa_handle_impl_t)sa_find_group_handle(group);
3916         if (handle == NULL)
3917                 return (ret);

3919         target = sa_find_resource(handle, newname);
3920         if (target != NULL) {
3921                 ret = SA_DUPLICATE_NAME;
3922         } else {
3923                 /*
3924                  * Everything appears to be valid at this
3925                  * point. Change the name of the active share and then
3926                  * update the share in the appropriate repository.
3927                  */
3928                 ret = proto_rename_resource(handle, group, resource, newname);
3929                 set_node_attr(resource, "name", newname);

3931                 if (!sa_is_persistent((sa_group_t)share))
3932                         return (ret);

3934                 if (!sa_group_is_zfs(group)) {
3935                         ret = sa_commit_share(handle->scfhandle, group,
3909                         sa_handle_impl_t ihandle = (sa_handle_impl_t)handle;
3910                         ret = sa_commit_share(ihandle->scfhandle, group,
3936                             share);
3937                 } else {
3938                         ret = sa_zfs_update((sa_share_t)group);
3939                 }
3940         }
3941         return (ret);
3942 }
```

_____***unchanged_portion_omitted_***

```
4309 /*
4310  * sa_set_resource_description(resource, content)
4311  *
4312  * Set the description of share to content.
4313  */

4315 int
4316 sa_set_resource_description(sa_resource_t resource, char *content)
4317 {
4318         xmlNodePtr node;
4319         sa_group_t group;
4320         sa_share_t share;
4321         int ret = SA_OK;
```

```
4323         for (node = ((xmlNodePtr)resource)->children;
4324             node != NULL;
4325             node = node->next) {
4326                 if (xmlStrcmp(node->name, (xmlChar *)"description") == 0) {
4327                         break;
4328                 }
4329         }

4331         /* no existing description but want to add */
4332         if (node == NULL && content != NULL) {
4333                 /* add a description */
4334                 node = _sa_set_share_description(resource, content);
4335         } else if (node != NULL && content != NULL) {
4336                 /* update a description */
4337                 xmlNodeSetContent(node, (xmlChar *)content);
4338         } else if (node != NULL && content == NULL) {
4339                 /* remove an existing description */
4340                 xmlUnlinkNode(node);
4341                 xmlFreeNode(node);
4342         }

4344         share = sa_get_resource_parent(resource);
4345         group = sa_get_parent_group(share);
4346         if (group != NULL &&
4347             sa_is_persistent(share) && (!sa_group_is_zfs(group))) {
4348                 sa_handle_t handle = sa_find_group_handle(group);
4349                 if (handle != NULL)
4350                         ret = sa_commit_share(handle->scfhandle,
4323                 sa_handle_impl_t impl_handle;
4324                 impl_handle = (sa_handle_impl_t)sa_find_group_handle(group);
4325                 if (impl_handle != NULL)
4326                         ret = sa_commit_share(impl_handle->scfhandle,
4351                             group, share);
4352                 else
4353                         ret = SA_SYSTEM_ERR;
4354         }
4355         return (ret);
4356 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   11482 Tue Sep 10 06:31:58 2013
new/usr/src/lib/libshare/common/libshare.h
4095 minor cleanup up libshare
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 RackTop Systems.
#endif /* ! codereview */
 */

/*
 * basic API declarations for share management
 */

#ifndef _LIBSHARE_H
#define _LIBSHARE_H

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/types.h>
#include <libnvpair.h>

/*
 * Basic datatypes for most functions
 */
typedef void *sa_group_t;
typedef void *sa_share_t;
typedef void *sa_property_t;
typedef void *sa_optionset_t;
typedef void *sa_security_t;
typedef void *sa_protocol_properties_t;
typedef void *sa_resource_t;

typedef struct sa_handle *sa_handle_t;  /* opaque handle to access core function
typedef void *sa_handle_t;      /* opaque handle to access core functions */

/*
 * defined error values
 */

#define SA_OK                   0
#define SA_NO_SUCH_PATH         1       /* provided path doesn't exist */
```



```
#define SA_NO_MEMORY            2       /* no memory for data structures */
#define SA_DUPLICATE_NAME       3       /* object name is already in use */
#define SA_BAD_PATH             4       /* not a full path */
#define SA_NO_SUCH_GROUP        5       /* group is not defined */
#define SA_CONFIG_ERR           6       /* system configuration error */
#define SA_SYSTEM_ERR           7       /* system error, use errno */
#define SA_SYNTAX_ERR           8       /* syntax error on command line */
#define SA_NO_PERMISSION        9       /* no permission for operation */
#define SA_BUSY                 10      /* resource is busy */
#define SA_NO_SUCH_PROP         11      /* property doesn't exist */
#define SA_INVALID_NAME         12      /* name of object is invalid */
#define SA_INVALID_PROTOCOL     13      /* specified protocol not valid */
#define SA_NOT_ALLOWED          14      /* operation not allowed */
#define SA_BAD_VALUE            15      /* bad value for property */
#define SA_INVALID_SECURITY     16      /* invalid security type */
#define SA_NO_SUCH_SECURITY     17      /* security set not found */
#define SA_VALUE_CONFLICT       18      /* property value conflict */
#define SA_NOT_IMPLEMENTED      19      /* plugin interface not implemented */
#define SA_INVALID_PATH         20      /* path is sub-dir of existing share */
#define SA_NOT_SUPPORTED        21      /* operation not supported for proto */
#define SA_PROP_SHARE_ONLY      22      /* property valid on share only */
#define SA_NOT_SHARED           23      /* path is not shared */
#define SA_NO_SUCH_RESOURCE     24      /* resource not found */
#define SA_RESOURCE_REQUIRED    25      /* resource name is required  */
#define SA_MULTIPLE_ERROR       26      /* multiple protocols reported error */
#define SA_PATH_IS_SUBDIR       27      /* check_path found path is subdir */
#define SA_PATH_IS_PARENTDIR    28      /* check_path found path is parent */
#define SA_NO_SECTION           29      /* protocol requires section info */
#define SA_NO_SUCH_SECTION      30      /* no section found */
#define SA_NO_PROPERTIES        31      /* no properties found */
#define SA_PASSWORD_ENC         32      /* passwords must be encrypted */
#define SA_SHARE_EXISTS         33      /* path or file is already shared */

/* API Initialization */
#define SA_INIT_SHARE_API       0x0001  /* init share specific interface */
#define SA_INIT_CONTROL_API     0x0002  /* init control specific interface */

/* not part of API returns */
#define SA_LEGACY_ERR           32      /* share/unshare error return */

/*
 * other defined values
 */

#define SA_MAX_NAME_LEN         100     /* must fit service instance name */
#define SA_MAX_RESOURCE_NAME    255     /* Maximum length of resource name */

/* Used in calls to sa_add_share() and sa_add_resource() */
#define SA_SHARE_TRANSIENT      0       /* shared but not across reboot */
#define SA_SHARE_LEGACY         1       /* share is in dfstab only */
#define SA_SHARE_PERMANENT      2       /* share goes to repository */

/* sa_check_path() related */
#define SA_CHECK_NORMAL         0       /* only check against active shares */
#define SA_CHECK_STRICT         1       /* check against all shares */

/* RBAC related */
#define SA_RBAC_MANAGE  "solaris.smf.manage.shares"
#define SA_RBAC_VALUE   "solaris.smf.value.shares"

/*
 * Feature set bit definitions
 */

#define SA_FEATURE_NONE         0x0000  /* no feature flags set */
#define SA_FEATURE_RESOURCE     0x0001  /* resource names are required */
```

```
#define SA_FEATURE_DFSTAB       0x0002  /* need to manage in dfstab */
#define SA_FEATURE_ALLOWSUBDIRS 0x0004  /* allow subdirs to be shared */
#define SA_FEATURE_ALLOWPARDIRS 0x0008  /* allow parent dirs to be shared */
#define SA_FEATURE_HAS_SECTIONS 0x0010  /* protocol supports sections */
#define SA_FEATURE_ADD_PROPERTIES       0x0020  /* can add properties */
#define SA_FEATURE_SERVER       0x0040  /* protocol supports server mode */

/*
 * legacy files
 */

#define SA_LEGACY_DFSTAB        "/etc/dfs/dfstab"
#define SA_LEGACY_SHARETAB      "/etc/dfs/sharetab"

/*
 * SMF related
 */

#define SA_SVC_FMRI_BASE        "svc:/network/shares/group"

/* initialization */
extern sa_handle_t sa_init(int);
extern void sa_fini(sa_handle_t);
extern int sa_update_config(sa_handle_t);
extern char *sa_errorstr(int);

/* protocol names */
extern int sa_get_protocols(char ***);
extern int sa_valid_protocol(char *);

/* group control (create, remove, etc) */
extern sa_group_t sa_create_group(sa_handle_t, char *, int *);
extern int sa_remove_group(sa_group_t);
extern sa_group_t sa_get_group(sa_handle_t, char *);
extern sa_group_t sa_get_next_group(sa_group_t);
extern char *sa_get_group_attr(sa_group_t, char *);
extern int sa_set_group_attr(sa_group_t, char *, char *);
extern sa_group_t sa_get_sub_group(sa_group_t);
extern int sa_valid_group_name(char *);

/* share control */
extern sa_share_t sa_add_share(sa_group_t, char *, int, int *);
extern int sa_check_path(sa_group_t, char *, int);
extern int sa_move_share(sa_group_t, sa_share_t);
extern int sa_remove_share(sa_share_t);
extern sa_share_t sa_get_share(sa_group_t, char *);
extern sa_share_t sa_find_share(sa_handle_t, char *);
extern sa_share_t sa_get_next_share(sa_share_t);
extern char *sa_get_share_attr(sa_share_t, char *);
extern char *sa_get_share_description(sa_share_t);
extern sa_group_t sa_get_parent_group(sa_share_t);
extern int sa_set_share_attr(sa_share_t, char *, char *);
extern int sa_set_share_description(sa_share_t, char *);
extern int sa_enable_share(sa_group_t, char *);
extern int sa_disable_share(sa_share_t, char *);
extern boolean_t sa_is_share(void *);
extern int sa_is_share(void *);

/* resource name related */
extern sa_resource_t sa_find_resource(sa_handle_t, char *);
extern sa_resource_t sa_get_resource(sa_group_t, char *);
extern sa_resource_t sa_get_next_resource(sa_resource_t);
extern sa_share_t sa_get_resource_parent(sa_resource_t);
extern sa_resource_t sa_get_share_resource(sa_share_t, char *);
extern sa_resource_t sa_add_resource(sa_share_t, char *, int, int *);
extern int sa_remove_resource(sa_resource_t);
```



```
extern char *sa_get_resource_attr(sa_resource_t, char *);
extern int sa_set_resource_attr(sa_resource_t, char *, char *);
extern int sa_set_resource_description(sa_resource_t, char *);
extern char *sa_get_resource_description(sa_resource_t);
extern int sa_enable_resource(sa_resource_t, char *);
extern int sa_disable_resource(sa_resource_t, char *);
extern int sa_rename_resource(sa_resource_t, char *);
extern void sa_fix_resource_name(char *);

/* data structure free calls */
extern void sa_free_attr_string(char *);
extern void sa_free_share_description(char *);

/* optionset control */
extern sa_optionset_t sa_get_optionset(sa_group_t, char *);
extern sa_optionset_t sa_get_next_optionset(sa_group_t);
extern char *sa_get_optionset_attr(sa_optionset_t, char *);
extern void sa_set_optionset_attr(sa_optionset_t, char *, char *);
extern sa_optionset_t sa_create_optionset(sa_group_t, char *);
extern int sa_destroy_optionset(sa_optionset_t);
extern sa_optionset_t sa_get_derived_optionset(void *, char *, int);
extern void sa_free_derived_optionset(sa_optionset_t);

/* property functions */
extern sa_property_t sa_get_property(sa_optionset_t, char *);
extern sa_property_t sa_get_next_property(sa_group_t);
extern char *sa_get_property_attr(sa_property_t, char *);
extern sa_property_t sa_create_section(char *, char *);
extern void sa_set_section_attr(sa_property_t, char *, char *);
extern sa_property_t sa_create_property(char *, char *);
extern int sa_add_property(void *, sa_property_t);
extern int sa_update_property(sa_property_t, char *);
extern int sa_remove_property(sa_property_t);
extern int sa_commit_properties(sa_optionset_t, int);
extern int sa_valid_property(sa_handle_t, void *, char *, sa_property_t);
extern boolean_t sa_is_persistent(void *);
extern int sa_is_persistent(void *);

/* security control */
extern sa_security_t sa_get_security(sa_group_t, char *, char *);
extern sa_security_t sa_get_next_security(sa_security_t);
extern char *sa_get_security_attr(sa_optionset_t, char *);
extern sa_security_t sa_create_security(sa_group_t, char *, char *);
extern int sa_destroy_security(sa_security_t);
extern void sa_set_security_attr(sa_security_t, char *, char *);
extern sa_optionset_t sa_get_all_security_types(void *, char *, int);
extern sa_security_t sa_get_derived_security(void *, char *, char *, int);
extern void sa_free_derived_security(sa_security_t);

/* protocol specific interfaces */
extern int sa_parse_legacy_options(sa_group_t, char *, char *);
extern char *sa_proto_legacy_format(char *, sa_group_t, int);
extern boolean_t sa_is_security(char *, char *);
extern int sa_is_security(char *, char *);
extern sa_protocol_properties_t sa_proto_get_properties(char *);
extern uint64_t sa_proto_get_featureset(char *);
extern sa_property_t sa_get_protocol_section(sa_protocol_properties_t, char *);
extern sa_property_t sa_get_next_protocol_section(sa_property_t, char *);
extern sa_property_t sa_get_protocol_property(sa_protocol_properties_t, char *);
extern sa_property_t sa_get_next_protocol_property(sa_property_t, char *);
extern int sa_set_protocol_property(sa_property_t, char *, char *);
extern char *sa_get_protocol_status(char *);
extern void sa_format_free(char *);
extern sa_protocol_properties_t sa_create_protocol_properties(char *);
extern int sa_add_protocol_property(sa_protocol_properties_t, sa_property_t);
extern int sa_proto_valid_prop(sa_handle_t, char *, sa_property_t,
```

```
256     sa_optionset_t);
257 extern int sa_proto_valid_space(char *, char *);
258 extern char *sa_proto_space_alias(char *, char *);
259 extern int sa_proto_get_transients(sa_handle_t, char *);
260 extern int sa_proto_notify_resource(sa_resource_t, char *);
261 extern int sa_proto_change_notify(sa_share_t, char *);
262 extern int sa_proto_delete_section(char *, char *);

264 /* handle legacy (dfstab/sharetab) files */
265 extern int sa_delete_legacy(sa_share_t, char *);
266 extern int sa_update_legacy(sa_share_t, char *);
267 extern int sa_update_sharetab(sa_share_t, char *);
268 extern int sa_delete_sharetab(sa_handle_t, char *, char *);

270 /* ZFS functions */
271 extern boolean_t sa_zfs_is_shared(sa_handle_t, char *);
272 extern boolean_t sa_group_is_zfs(sa_group_t);
273 extern boolean_t sa_path_is_zfs(char *);
242 extern int sa_zfs_is_shared(sa_handle_t, char *);
243 extern int sa_group_is_zfs(sa_group_t);
244 extern int sa_path_is_zfs(char *);
274 extern int sa_zfs_setprop(sa_handle_t, char *, nvlist_t *);

276 /* SA Handle specific functions */
277 extern sa_handle_t sa_find_group_handle(sa_group_t);

279 #ifdef  __cplusplus
280 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    6085 Tue Sep 10 06:31:58 2013
new/usr/src/lib/libshare/common/libshare_impl.h
4095 minor cleanup up libshare
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2013 RackTop Systems.
 */

/*
#endif /* ! codereview */
 * basic declarations for implementation of the share management
 * libraries.
 */

#ifndef _LIBSHARE_IMPL_H
#define _LIBSHARE_IMPL_H

#include <libshare.h>
#include <libscf.h>
#include <scfutil.h>
#include <libzfs.h>
#include <sharefs/share.h>
#include <sharefs/sharetab.h>

#ifdef __cplusplus
extern "C" {
#endif

/* directory to find plugin modules in */
#define SA_LIB_DIR      "/usr/lib/fs"

/* default group name for dfstab file */
#define SA_DEFAULT_FILE_GRP     "sys"

typedef void *sa_phandle_t;

#define SA_PLUGIN_VERSION       1
struct sa_plugin_ops {
        int     sa_version;                     /* version number */
        char    *sa_protocol;                   /* protocol name */
        int     (*sa_init)();
        void    (*sa_fini)();
        int     (*sa_share)(sa_share_t);        /* start sharing */
        int     (*sa_unshare)(sa_share_t, char *); /* stop sharing */
        int     (*sa_valid_prop)(sa_handle_t, sa_property_t,
            sa_optionset_t); /* validate */
        int     (*sa_valid_space)(char *);      /* is name valid optionspace? */
        int     (*sa_security_prop)(char *);    /* property is security */
        int     (*sa_legacy_opts)(sa_group_t, char *); /* parse legacy opts */
        char   *(*sa_legacy_format)(sa_group_t, int);
        int     (*sa_set_proto_prop)(sa_property_t);    /* set a property */
        sa_protocol_properties_t (*sa_get_proto_set)(); /* get properties */
        char   *(*sa_get_proto_status)();
        char   *(*sa_space_alias)(char *);
        int     (*sa_update_legacy)(sa_share_t);
        int     (*sa_delete_legacy)(sa_share_t);
        int     (*sa_change_notify)(sa_share_t);
        int     (*sa_enable_resource)(sa_resource_t);
        int     (*sa_disable_resource)(sa_resource_t);
        uint64_t (*sa_features)(void);
        int     (*sa_get_transient_shares)(sa_handle_t); /* add transients */
        int     (*sa_notify_resource)(sa_resource_t);
        int     (*sa_rename_resource)(sa_handle_t, sa_resource_t, char *);
        int     (*sa_run_command)(int, int, char **); /* proto specific */
        int     (*sa_command_help)();
        int     (*sa_delete_proto_section)(char *);
};

struct sa_proto_handle {
        int                     sa_num_proto;
        char                    **sa_proto;
        struct sa_plugin_ops    **sa_ops;
};

typedef struct propertylist {
        struct propertylist     *pl_next;
        int                     pl_type;
        union propval {
            sa_optionset_t      pl_optionset;
            sa_security_t       pl_security;
            void                *pl_void;
        }                       pl_value;
} property_list_t;

/* internal version of sa_handle_t */
struct sa_handle {
typedef struct sa_handle_impl {
        uint64_t        flags;
        scfutilhandle_t *scfhandle;
        libzfs_handle_t *zfs_libhandle;
        zfs_handle_t    **zfs_list;
        size_t          zfs_list_count;
        xmlNodePtr      tree;
        xmlDocPtr       doc;
        uint64_t        tssharetab;
        uint64_t        tstrans;
};
} *sa_handle_impl_t;

extern int sa_proto_share(char *, sa_share_t);
extern int sa_proto_unshare(sa_share_t, char *, char *);
extern int sa_proto_valid_prop(sa_handle_t, char *, sa_property_t,
    sa_optionset_t);
extern int sa_proto_security_prop(char *, char *);
extern int sa_proto_legacy_opts(char *, sa_group_t, char *);
```

```
126 extern int sa_proto_share_resource(char *, sa_resource_t);
127 extern int sa_proto_unshare_resource(char *, sa_resource_t);

129 /* internal utility functions */
130 extern sa_optionset_t sa_get_derived_optionset(sa_group_t, char *, int);
131 extern void sa_free_derived_optionset(sa_optionset_t);
132 extern sa_optionset_t sa_get_all_security_types(void *, char *, int);
133 extern sa_security_t sa_get_derived_security(void *, char *, char *, int);
134 extern void sa_free_derived_security(sa_security_t);
135 extern sa_protocol_properties_t sa_create_protocol_properties(char *);
136 extern int sa_start_transaction(scfutilhandle_t *, char *);
137 extern int sa_end_transaction(scfutilhandle_t *, sa_handle_t);
 57 extern int sa_end_transaction(scfutilhandle_t *, sa_handle_impl_t);
138 extern void sa_abort_transaction(scfutilhandle_t *);
139 extern int sa_commit_share(scfutilhandle_t *, sa_group_t, sa_share_t);
140 extern int sa_set_property(scfutilhandle_t *, char *, char *);
141 extern void sa_free_fstype(char *fstyp);
142 extern int sa_delete_share(scfutilhandle_t *, sa_group_t, sa_share_t);
143 extern int sa_delete_instance(scfutilhandle_t *, char *);
144 extern int sa_create_pgroup(scfutilhandle_t *, char *);
145 extern int sa_delete_pgroup(scfutilhandle_t *, char *);
146 extern void sa_fillshare(sa_share_t share, char *proto, struct share *sh);
147 extern void sa_emptyshare(struct share *sh);

149 /* ZFS functions */
150 extern int sa_get_zfs_shares(sa_handle_t, char *);
151 extern int sa_zfs_update(sa_share_t);
152 extern int sa_share_zfs(sa_share_t, sa_resource_t, char *, share_t *,
153     void *, zfs_share_op_t);
154 extern int sa_sharetab_fill_zfs(sa_share_t share, struct share *sh,
155     char *proto);

157 /* plugin specific functions */
158 extern int proto_plugin_init();
159 extern void proto_plugin_fini();
160 extern int sa_proto_set_property(char *, sa_property_t);
161 extern int sa_proto_delete_legacy(char *, sa_share_t);
162 extern int sa_proto_update_legacy(char *, sa_share_t);
163 extern int sa_proto_rename_resource(sa_handle_t, char *,
164     sa_resource_t, char *);

166 #define PL_TYPE_PROPERTY        0
167 #define PL_TYPE_SECURITY        1

169 /* values only used by the internal dfstab/sharetab parser */
170 #define SA_SHARE_PARSER         0x1000

172 /* plugin handler only */
173 struct sa_proto_plugin {
174         struct sa_proto_plugin  *plugin_next;
175         struct sa_plugin_ops    *plugin_ops;
176         void                    *plugin_handle;
177 };
```

unchanged_portion_omitted_

```
**********************************************************
   36729 Tue Sep 10 06:31:58 2013
new/usr/src/lib/libshare/common/libshare_zfs.c
4095 minor cleanup up libshare
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2012 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 RackTop Systems.
#endif /* ! codereview */
 */

#include <stdio.h>
#include <libzfs.h>
#include <string.h>
#include <strings.h>
#include <errno.h>
#include <libshare.h>
#include "libshare_impl.h"
#include <libintl.h>
#include <sys/mnttab.h>
#include <sys/mntent.h>

extern sa_share_t _sa_add_share(sa_group_t, char *, int, int *, uint64_t);
extern sa_group_t _sa_create_zfs_group(sa_group_t, char *);
extern char *sa_fstype(char *);
extern void set_node_attr(void *, char *, char *);
extern boolean_t sa_is_share(void *);
extern int sa_is_share(void *);
extern void sa_update_sharetab_ts(sa_handle_t);

/*
 * File system specific code for ZFS. The original code was stolen
 * from the "zfs" command and modified to better suit this library's
 * usage.
 */

typedef struct get_all_cbdata {
        zfs_handle_t    **cb_handles;
        size_t          cb_alloc;
        size_t          cb_used;
        uint_t          cb_types;
} get_all_cbdata_t;
```



```
/*
 * sa_zfs_init(handle)
 * sa_zfs_init(impl_handle)
 *
 * Initialize an access handle into libzfs.  The handle needs to stay
 * around until sa_zfs_fini() in order to maintain the cache of
 * mounts.
 */

int
sa_zfs_init(sa_handle_t handle)
sa_zfs_init(sa_handle_impl_t impl_handle)
{
        handle->zfs_libhandle = libzfs_init();
        if (handle->zfs_libhandle != NULL) {
                libzfs_print_on_error(handle->zfs_libhandle, B_TRUE);
        impl_handle->zfs_libhandle = libzfs_init();
        if (impl_handle->zfs_libhandle != NULL) {
                libzfs_print_on_error(impl_handle->zfs_libhandle, B_TRUE);
                return (B_TRUE);
        }
        return (B_FALSE);
}

/*
 * sa_zfs_fini(handle)
 * sa_zfs_fini(impl_handle)
 *
 * cleanup data structures and the libzfs handle used for accessing
 * zfs file share info.
 */

void
sa_zfs_fini(sa_handle_t handle)
sa_zfs_fini(sa_handle_impl_t impl_handle)
{
        if (handle->zfs_libhandle != NULL) {
                if (handle->zfs_list != NULL) {
                        zfs_handle_t **zhp = handle->zfs_list;
        if (impl_handle->zfs_libhandle != NULL) {
                if (impl_handle->zfs_list != NULL) {
                        zfs_handle_t **zhp = impl_handle->zfs_list;
                        size_t i;

                        /*
                         * Contents of zfs_list need to be freed so we
                         * don't lose ZFS handles.
                         */
                        for (i = 0; i < handle->zfs_list_count; i++) {
                        for (i = 0; i < impl_handle->zfs_list_count; i++) {
                                zfs_close(zhp[i]);
                        }
                        free(handle->zfs_list);
                        handle->zfs_list = NULL;
                        handle->zfs_list_count = 0;
                        free(impl_handle->zfs_list);
                        impl_handle->zfs_list = NULL;
                        impl_handle->zfs_list_count = 0;
                }

                libzfs_fini(handle->zfs_libhandle);
                handle->zfs_libhandle = NULL;
                libzfs_fini(impl_handle->zfs_libhandle);
                impl_handle->zfs_libhandle = NULL;
        }
```

```
 109 }
_____unchanged_portion_omitted_

 176 /*
 177  * get_all_filesystems(zfs_handle_t ***fslist, size_t *count)
 178  *
 179  * iterate through all ZFS file systems starting at the root. Returns
 180  * a count and an array of handle pointers. Allocating is only done
 181  * once. The caller does not need to free since it will be done at
 182  * sa_zfs_fini() time.
 183  */

 185 static void
 186 get_all_filesystems(sa_handle_t handle,
 169 get_all_filesystems(sa_handle_impl_t impl_handle,
 187                         zfs_handle_t ***fslist, size_t *count)
 188 {
 189         get_all_cbdata_t cb = { 0 };
 190         cb.cb_types = ZFS_TYPE_FILESYSTEM;

 192         if (handle->zfs_list != NULL) {
 193                 *fslist = handle->zfs_list;
 194                 *count = handle->zfs_list_count;
 175         if (impl_handle->zfs_list != NULL) {
 176                 *fslist = impl_handle->zfs_list;
 177                 *count = impl_handle->zfs_list_count;
 195                 return;
 196         }

 198         (void) zfs_iter_root(handle->zfs_libhandle,
 181         (void) zfs_iter_root(impl_handle->zfs_libhandle,
 199             get_one_filesystem, &cb);

 201         handle->zfs_list = *fslist = cb.cb_handles;
 202         handle->zfs_list_count = *count = cb.cb_used;
 184         impl_handle->zfs_list = *fslist = cb.cb_handles;
 185         impl_handle->zfs_list_count = *count = cb.cb_used;
 203 }
_____unchanged_portion_omitted_

 263 /*
 264  * get_zfs_dataset(handle, path)
 247  * get_zfs_dataset(impl_handle, path)
 265  *
 266  * get the name of the ZFS dataset the path is equivalent to.  The
 267  * dataset name is used for get/set of ZFS properties since libzfs
 268  * requires a dataset to do a zfs_open().
 269  */

 271 static char *
 272 get_zfs_dataset(sa_handle_t handle, char *path,
 255 get_zfs_dataset(sa_handle_impl_t impl_handle, char *path,
 273     boolean_t search_mnttab)
 274 {
 275         size_t i, count = 0;
 276         char *dataset = NULL;
 277         zfs_handle_t **zlist;
 278         char mountpoint[ZFS_MAXPROPLEN];
 279         char canmount[ZFS_MAXPROPLEN];

 281         get_all_filesystems(handle, &zlist, &count);
 264         get_all_filesystems(impl_handle, &zlist, &count);
 282         qsort(zlist, count, sizeof (void *), mountpoint_compare);
 283         for (i = 0; i < count; i++) {
 284                 /* must have a mountpoint */
 285                 if (zfs_prop_get(zlist[i], ZFS_PROP_MOUNTPOINT, mountpoint,
```



```
 286                     sizeof (mountpoint), NULL, NULL, 0, B_FALSE) != 0) {
 287                         /* no mountpoint */
 288                         continue;
 289                 }

 291                 /* mountpoint must be a path */
 292                 if (strcmp(mountpoint, ZFS_MOUNTPOINT_NONE) == 0 ||
 293                     strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) == 0) {
 294                         /*
 295                          * Search mmttab for mountpoint and get dataset.
 296                          */

 298                         if (search_mnttab == B_TRUE &&
 299                             get_legacy_mountpoint(path, mountpoint,
 300                             sizeof (mountpoint), NULL, 0) == 0) {
 301                                 dataset = mountpoint;
 302                                 break;
 303                         }
 304                         continue;
 305                 }

 307                 /* canmount must be set */
 308                 canmount[0] = '\0';
 309                 if (zfs_prop_get(zlist[i], ZFS_PROP_CANMOUNT, canmount,
 310                     sizeof (canmount), NULL, NULL, 0, B_FALSE) != 0 ||
 311                     strcmp(canmount, "off") == 0)
 312                         continue;

 314                 /*
 315                  * have a mountable handle but want to skip those marked none
 316                  * and legacy
 317                  */
 318                 if (strcmp(mountpoint, path) == 0) {
 319                         dataset = (char *)zfs_get_name(zlist[i]);
 320                         break;
 321                 }

 323         }

 325         if (dataset != NULL)
 326                 dataset = strdup(dataset);

 328         return (dataset);
 329 }

 331 /*
 332  * get_zfs_property(dataset, property)
 333  *
 334  * Get the file system property specified from the ZFS dataset.
 335  */

 337 static char *
 338 get_zfs_property(sa_handle_t handle, char *dataset, zfs_prop_t property)
 321 get_zfs_property(char *dataset, zfs_prop_t property)
 339 {
 340         zfs_handle_t *z_fs;
 323         zfs_handle_t *handle = NULL;
 341         char shareopts[ZFS_MAXPROPLEN];
 325         libzfs_handle_t *libhandle;

 343         z_fs = zfs_open(handle->zfs_libhandle, dataset, ZFS_TYPE_FILESYSTEM);
 344         if (z_fs != NULL) {
 345                 if (zfs_prop_get(z_fs, property, shareopts,
 327         libhandle = libzfs_init();
 328         if (libhandle != NULL) {
 329                 handle = zfs_open(libhandle, dataset, ZFS_TYPE_FILESYSTEM);
```

```
330                 if (handle != NULL) {
331                         if (zfs_prop_get(handle, property, shareopts,
346                     sizeof (shareopts), NULL, NULL, 0,
347                     B_FALSE) == 0) {
348                         zfs_close(z_fs);
334                                 zfs_close(handle);
335                                 libzfs_fini(libhandle);
349                         return (strdup(shareopts));
350                 }
351                 zfs_close(z_fs);
338                         zfs_close(handle);
339                 }
340                 libzfs_fini(libhandle);
352         }
353         return (NULL);
354 }

356 /*
357  * sa_zfs_is_shared(handle, path)
358  *
359  * Check to see if the ZFS path provided has the sharenfs option set
360  * or not.
361  */

363 boolean_t
364 sa_zfs_is_shared(sa_handle_t handle, char *path)
352 int
353 sa_zfs_is_shared(sa_handle_t sahandle, char *path)
365 {
366         int ret = B_FALSE;
355         int ret = 0;
367         char *dataset;
368         zfs_handle_t *z_fs = NULL;
357         zfs_handle_t *handle = NULL;
369         char shareopts[ZFS_MAXPROPLEN];
359         libzfs_handle_t *libhandle;

371         dataset = get_zfs_dataset(handle, path, B_FALSE);
361         dataset = get_zfs_dataset((sa_handle_t)sahandle, path, B_FALSE);
372         if (dataset != NULL) {
373                 z_fs = zfs_open(handle->zfs_libhandle, dataset,
363                 libhandle = libzfs_init();
364                 if (libhandle != NULL) {
365                         handle = zfs_open(libhandle, dataset,
374                     ZFS_TYPE_FILESYSTEM);
375                 if (z_fs != NULL) {
376                         if (zfs_prop_get(z_fs, ZFS_PROP_SHARENFS,
367                         if (handle != NULL) {
368                                 if (zfs_prop_get(handle, ZFS_PROP_SHARENFS,
377                             shareopts, sizeof (shareopts), NULL, NULL,
378                             0, B_FALSE) == 0 &&
379                             strcmp(shareopts, "off") != 0) {
380                                 ret = B_TRUE; /* it is shared */
372                                         ret = 1; /* it is shared */
381                         }
382                         zfs_close(z_fs);
374                                 zfs_close(handle);
375                         }
376                         libzfs_fini(libhandle);
383                 }
384                 free(dataset);
385         }
386         return (ret);
387 }
_____unchanged_portion_omitted_
```



```
716 /*
717  * sa_get_zfs_shares(handle, groupname)
718  *
719  * Walk the mnttab for all zfs mounts and determine which are
720  * shared. Find or create the appropriate group/sub-group to contain
721  * the shares.
722  *
723  * All shares are in a sub-group that will hold the properties. This
724  * allows representing the inherited property model.
725  *
726  * One area of complication is if "sharenfs" is set at one level of
727  * the directory tree and "sharesmb" is set at a different level, the
728  * a sub-group must be formed at the lower level for both
729  * protocols. That is the nature of the problem in CR 6667349.
730  */

732 int
733 sa_get_zfs_shares(sa_handle_t handle, char *groupname)
734 {
735         sa_group_t zfsgroup;
736         boolean_t nfs;
737         boolean_t nfs_inherited;
738         boolean_t smb;
739         boolean_t smb_inherited;
740         zfs_handle_t **zlist;
741         char nfsshareopts[ZFS_MAXPROPLEN];
742         char smbshareopts[ZFS_MAXPROPLEN];
743         sa_share_t share;
744         zprop_source_t source;
745         char nfssourcestr[ZFS_MAXPROPLEN];
746         char smbsourcestr[ZFS_MAXPROPLEN];
747         char mountpoint[ZFS_MAXPROPLEN];
748         size_t count = 0, i;
749         libzfs_handle_t *zfs_libhandle;
750         int err = SA_OK;

752         /*
753          * If we can't access libzfs, don't bother doing anything.
754          */
755         zfs_libhandle = handle->zfs_libhandle;
749         zfs_libhandle = ((sa_handle_impl_t)handle)->zfs_libhandle;
756         if (zfs_libhandle == NULL)
757                 return (SA_SYSTEM_ERR);

759         zfsgroup = find_or_create_group(handle, groupname, NULL, &err);
760         /* Not an error, this could be a legacy condition */
761         if (zfsgroup == NULL)
762                 return (SA_OK);

764         /*
765          * need to walk the mounted ZFS pools and datasets to
766          * find shares that are possible.
767          */
768         get_all_filesystems(handle, &zlist, &count);
762         get_all_filesystems((sa_handle_impl_t)handle, &zlist, &count);
769         qsort(zlist, count, sizeof (void *), mountpoint_compare);

771         for (i = 0; i < count; i++) {
772                 char *dataset;

774                 source = ZPROP_SRC_ALL;
775                 /* If no mountpoint, skip. */
776                 if (zfs_prop_get(zlist[i], ZFS_PROP_MOUNTPOINT,
777                     mountpoint, sizeof (mountpoint), NULL, NULL, 0,
778                     B_FALSE) != 0)
779                         continue;
```

```
781                 /*
782                  * zfs_get_name value must not be freed. It is just a
783                  * pointer to a value in the handle.
784                  */
785                 if ((dataset = (char *)zfs_get_name(zlist[i])) == NULL)
786                         continue;
787
788                 /*
789                  * only deal with "mounted" file systems since
790                  * unmounted file systems can't actually be shared.
791                  */
792
793                 if (!zfs_is_mounted(zlist[i], NULL))
794                         continue;
795
796                 nfs = nfs_inherited = B_FALSE;
797
798                 if (zfs_prop_get(zlist[i], ZFS_PROP_SHARENFS, nfsshareopts,
799                     sizeof (nfsshareopts), &source, nfssourcestr,
800                     ZFS_MAXPROPLEN, B_FALSE) == 0 &&
801                     strcmp(nfsshareopts, "off") != 0) {
802                         if (source & ZPROP_SRC_INHERITED)
803                                 nfs_inherited = B_TRUE;
804                         else
805                                 nfs = B_TRUE;
806                 }
807
808                 smb = smb_inherited = B_FALSE;
809                 if (zfs_prop_get(zlist[i], ZFS_PROP_SHARESMB, smbshareopts,
810                     sizeof (smbshareopts), &source, smbsourcestr,
811                     ZFS_MAXPROPLEN, B_FALSE) == 0 &&
812                     strcmp(smbshareopts, "off") != 0) {
813                         if (source & ZPROP_SRC_INHERITED)
814                                 smb_inherited = B_TRUE;
815                         else
816                                 smb = B_TRUE;
817                 }
818
819                 /*
820                  * If the mountpoint is already shared, it must be a
821                  * non-ZFS share. We want to remove the share from its
822                  * parent group and reshare it under ZFS.
823                  */
824                 share = sa_find_share(handle, mountpoint);
825                 if (share != NULL &&
826                     (nfs || smb || nfs_inherited || smb_inherited)) {
827                         err = sa_remove_share(share);
828                         share = NULL;
829                 }
830
831                 /*
832                  * At this point, we have the information needed to
833                  * determine what to do with the share.
834                  *
835                  * If smb or nfs is set, we have a new sub-group.
836                  * If smb_inherit and/or nfs_inherit is set, then
837                  * place on an existing sub-group. If both are set,
838                  * the existing sub-group is the closest up the tree.
839                  */
840                 if (nfs || smb) {
841                         /*
842                          * Non-inherited is the straightforward
843                          * case. sa_zfs_process_share handles it
844                          * directly. Make sure that if the "other"
845                          * protocol is inherited, that we treat it as
```



```
846                          * non-inherited as well.
847                          */
848                         if (nfs || nfs_inherited) {
849                                 err = sa_zfs_process_share(handle, zfsgroup,
850                                     share, mountpoint, "nfs",
851                                     0, nfsshareopts,
852                                     nfssourcestr, dataset);
853                                 share = sa_find_share(handle, mountpoint);
854                         }
855                         if (smb || smb_inherited) {
856                                 err = sa_zfs_process_share(handle, zfsgroup,
857                                     share, mountpoint, "smb",
858                                     0, smbshareopts,
859                                     smbsourcestr, dataset);
860                         }
861                 } else if (nfs_inherited || smb_inherited) {
862                         char *grpdataset;
863                         /*
864                          * If we only have inherited groups, it is
865                          * important to find the closer of the two if
866                          * the protocols are set at different
867                          * levels. The closest sub-group is the one we
868                          * want to work with.
869                          */
870                         if (nfs_inherited && smb_inherited) {
871                                 if (strcmp(nfssourcestr, smbsourcestr) <= 0)
872                                         grpdataset = nfssourcestr;
873                                 else
874                                         grpdataset = smbsourcestr;
875                         } else if (nfs_inherited) {
876                                 grpdataset = nfssourcestr;
877                         } else if (smb_inherited) {
878                                 grpdataset = smbsourcestr;
879                         }
880                         if (nfs_inherited) {
881                                 err = sa_zfs_process_share(handle, zfsgroup,
882                                     share, mountpoint, "nfs",
883                                     ZPROP_SRC_INHERITED, nfsshareopts,
884                                     grpdataset, dataset);
885                                 share = sa_find_share(handle, mountpoint);
886                         }
887                         if (smb_inherited) {
888                                 err = sa_zfs_process_share(handle, zfsgroup,
889                                     share, mountpoint, "smb",
890                                     ZPROP_SRC_INHERITED, smbshareopts,
891                                     grpdataset, dataset);
892                         }
893                 }
894         }
895         /*
896          * Don't need to free the "zlist" variable since it is only a
897          * pointer to a cached value that will be freed when
898          * sa_fini() is called.
899          */
900         return (err);
901 }

903 #define COMMAND         "/usr/sbin/zfs"

905 /*
906  * sa_zfs_set_sharenfs(group, path, on)
907  *
908  * Update the "sharenfs" property on the path. If on is true, then set
909  * to the properties on the group or "on" if no properties are
910  * defined. Set to "off" if on is false.
911  */
```

```
913 int
914 sa_zfs_set_sharenfs(sa_group_t group, char *path, int on)
915 {
916         int ret = SA_NOT_IMPLEMENTED;
917         char *command;

919         command = malloc(ZFS_MAXPROPLEN * 2);
920         if (command != NULL) {
921                 char *opts = NULL;
922                 char *dataset = NULL;
923                 FILE *pfile;
924                 sa_handle_t handle;
918                 sa_handle_impl_t impl_handle;
925                 /* for now, NFS is always available for "zfs" */
926                 if (on) {
927                         opts = sa_proto_legacy_format("nfs", group, 1);
928                         if (opts != NULL && strlen(opts) == 0) {
929                                 free(opts);
930                                 opts = strdup("on");
931                         }
932                 }

934                 handle = sa_find_group_handle(group);
935                 assert(handle != NULL);
936                 if (handle != NULL)
937                         dataset = get_zfs_dataset(handle, path, B_FALSE);
928                 impl_handle = (sa_handle_impl_t)sa_find_group_handle(group);
929                 assert(impl_handle != NULL);
930                 if (impl_handle != NULL)
931                         dataset = get_zfs_dataset(impl_handle, path, B_FALSE);
938                 else
939                         ret = SA_SYSTEM_ERR;

941                 if (dataset != NULL) {
942                         (void) snprintf(command, ZFS_MAXPROPLEN * 2,
943                             "%s set sharenfs=\"%s\" %s", COMMAND,
944                             opts != NULL ? opts : "off", dataset);
945                         pfile = popen(command, "r");
946                         if (pfile != NULL) {
947                                 ret = pclose(pfile);
948                                 if (ret != 0)
949                                         ret = SA_SYSTEM_ERR;
950                         }
951                 }
952                 if (opts != NULL)
953                         free(opts);
954                 if (dataset != NULL)
955                         free(dataset);
956                 free(command);
957         }
958         return (ret);
959 }
```

*unchanged_portion_omitted_*

```
1011 /*
1012  * sa_zfs_set_sharesmb(group, path, on)
1013  *
1014  * Update the "sharesmb" property on the path. If on is true, then set
1015  * to the properties on the group or "on" if no properties are
1016  * defined. Set to "off" if on is false.
1017  */

1019 int
1020 sa_zfs_set_sharesmb(sa_group_t group, char *path, int on)
1021 {
```



```
1022         int ret = SA_NOT_IMPLEMENTED;
1023         char *command;
1024         sa_share_t share;

1026         /* In case SMB not enabled */
1027         if (sa_get_optionset(group, "smb") == NULL)
1028                 return (SA_NOT_SUPPORTED);

1030         command = malloc(ZFS_MAXPROPLEN * 2);
1031         if (command != NULL) {
1032                 char *opts = NULL;
1033                 char *dataset = NULL;
1034                 FILE *pfile;
1035                 sa_handle_t handle;
1029                 sa_handle_impl_t impl_handle;

1037                 if (on) {
1038                         char *newopt;

1040                         share = sa_get_share(group, NULL);
1041                         opts = sa_proto_legacy_format("smb", share, 1);
1042                         if (opts != NULL && strlen(opts) == 0) {
1043                                 free(opts);
1044                                 opts = strdup("on");
1045                         }
1046                         newopt = add_resources(opts, share);
1047                         free(opts);
1048                         opts = newopt;
1049                 }

1051                 handle = sa_find_group_handle(group);
1052                 assert(handle != NULL);
1053                 if (handle != NULL)
1054                         dataset = get_zfs_dataset(handle, path, B_FALSE);
1045                 impl_handle = (sa_handle_impl_t)sa_find_group_handle(group);
1046                 assert(impl_handle != NULL);
1047                 if (impl_handle != NULL)
1048                         dataset = get_zfs_dataset(impl_handle, path, B_FALSE);
1055                 else
1056                         ret = SA_SYSTEM_ERR;

1058                 if (dataset != NULL) {
1059                         (void) snprintf(command, ZFS_MAXPROPLEN * 2,
1060                             "echo %s set sharesmb=\"%s\" %s", COMMAND,
1061                             opts != NULL ? opts : "off", dataset);
1062                         pfile = popen(command, "r");
1063                         if (pfile != NULL) {
1064                                 ret = pclose(pfile);
1065                                 if (ret != 0)
1066                                         ret = SA_SYSTEM_ERR;
1067                         }
1068                 }
1069                 if (opts != NULL)
1070                         free(opts);
1071                 if (dataset != NULL)
1072                         free(dataset);
1073                 free(command);
1074         }
1075         return (ret);
1076 }

1078 /*
1079  * sa_zfs_update(group)
1080  *
1081  * call back to ZFS to update the share if necessary.
1082  * Don't do it if it isn't a real change.
```

```
 */
int
sa_zfs_update(sa_group_t group)
{
        sa_optionset_t protopt;
        sa_group_t parent;
        char *command;
        char *optstring;
        int ret = SA_OK;
        int doupdate = 0;
        FILE *pfile;

        if (sa_is_share(group))
                parent = sa_get_parent_group(group);
        else
                parent = group;

        if (parent != NULL) {
                command = malloc(ZFS_MAXPROPLEN * 2);
                if (command == NULL)
                        return (SA_NO_MEMORY);

                *command = '\0';
                for (protopt = sa_get_optionset(parent, NULL); protopt != NULL;
                    protopt = sa_get_next_optionset(protopt)) {

                        char *proto = sa_get_optionset_attr(protopt, "type");
                        char *path;
                        char *dataset = NULL;
                        char *zfsopts = NULL;

                        if (sa_is_share(group)) {
                                path = sa_get_share_attr((sa_share_t)group,
                                    "path");
                                if (path != NULL) {
                                        sa_handle_t handle;
                                        sa_handle_impl_t impl_handle;

                                        handle = sa_find_group_handle(
                                        impl_handle = sa_find_group_handle(
                                            group);
                                        if (handle != NULL)
                                        if (impl_handle != NULL)
                                                dataset = get_zfs_dataset(
                                                    handle, path, B_FALSE);
                                                    impl_handle, path, B_FALSE);
                                        else
                                                ret = SA_SYSTEM_ERR;

                                        sa_free_attr_string(path);
                                }
                        } else {
                                dataset = sa_get_group_attr(group, "name");
                        }
                        /* update only when there is an optstring found */
                        doupdate = 0;
                        if (proto != NULL && dataset != NULL) {
                                sa_handle_t handle;

#endif /* ! codereview */
                                optstring = sa_proto_legacy_format(proto,
                                    group, 1);
                                handle = sa_find_group_handle(group);
                                zfsopts = get_zfs_property(handle, dataset,
                                zfsopts = get_zfs_property(dataset,
                                    ZFS_PROP_SHARENFS);
```



```
                                if (optstring != NULL && zfsopts != NULL) {
                                        if (strcmp(optstring, zfsopts) != 0)
                                                doupdate++;
                                }
                                if (doupdate) {
                                        if (optstring != NULL &&
                                            strlen(optstring) > 0) {
                                                (void) snprintf(command,
                                                    ZFS_MAXPROPLEN * 2,
                                                    "%s set share%s=%s %s",
                                                    COMMAND, proto,
                                                    optstring, dataset);
                                        } else {
                                                (void) snprintf(command,
                                                    ZFS_MAXPROPLEN * 2,
                                                    "%s set share%s=on %s",
                                                    COMMAND, proto,
                                                    dataset);
                                        }
                                        pfile = popen(command, "r");
                                        if (pfile != NULL)
                                                ret = pclose(pfile);
                                        switch (ret) {
                                        default:
                                        case 1:
                                                ret = SA_SYSTEM_ERR;
                                                break;
                                        case 2:
                                                ret = SA_SYNTAX_ERR;
                                                break;
                                        case 0:
                                                break;
                                        }
                                }
                                if (optstring != NULL)
                                        free(optstring);
                                if (zfsopts != NULL)
                                        free(zfsopts);
                        }
                        if (proto != NULL)
                                sa_free_attr_string(proto);
                        if (dataset != NULL)
                                free(dataset);
                }
                free(command);
        }
        return (ret);
}

/*
 * sa_group_is_zfs(group)
 *
 * Given the group, determine if the zfs attribute is set.
 */

boolean_t
int
sa_group_is_zfs(sa_group_t group)
{
        char *zfs;
        int ret = B_FALSE;
        int ret = 0;

        zfs = sa_get_group_attr(group, "zfs");
        if (zfs != NULL) {
```

```
1208                 ret = B_TRUE;
1196                 ret = 1;
1209                 sa_free_attr_string(zfs);
1210         }
1211         return (ret);
1212 }

1214 /*
1215  * sa_path_is_zfs(path)
1216  *
1217  * Check to see if the file system path represents is of type "zfs".
1218  */

1220 boolean_t
1208 int
1221 sa_path_is_zfs(char *path)
1222 {
1223         char *fstype;
1224         int ret = B_FALSE;
1212         int ret = 0;

1226         fstype = sa_fstype(path);
1227         if (fstype != NULL && strcmp(fstype, "zfs") == 0)
1228                 ret = B_TRUE;
1216                 ret = 1;
1229         if (fstype != NULL)
1230                 sa_free_fstype(fstype);
1231         return (ret);
1232 }
______unchanged_portion_omitted_

1251 #define SMAX(i, j)      \
1252         if ((j) > (i)) { \
1253                 (i) = (j); \
1254         }

1256 int
1257 sa_share_zfs(sa_share_t share, sa_resource_t resource, char *path, share_t *sh,
1258     void *exportdata, zfs_share_op_t operation)
1259 {
1248         libzfs_handle_t *libhandle;
1260         sa_group_t group;
1261         sa_handle_t handle;
1250         sa_handle_t sahandle;
1262         char *dataset;
1263         int err = EINVAL;
1264         int i, j;
1265         char newpath[MAXPATHLEN];
1266         char *pathp;
1267         char *resource_name;
1268 #endif /* ! codereview */

1270         /*
1271          * First find the dataset name
1272          */
1273         if ((group = sa_get_parent_group(share)) == NULL)  {
1274                 return (EINVAL);
1275         }
1276         if ((handle = sa_find_group_handle(group)) == NULL) {
1256         if ((sahandle = sa_find_group_handle(group)) == NULL) {
1277                 return (EINVAL);
1278         }

1280         /*
1281          * If get_zfs_dataset fails, see if it is a subdirectory
1282          */
```



```
1284         pathp = path;
1285         while ((dataset = get_zfs_dataset(handle, pathp, B_TRUE)) == NULL) {
1265         while ((dataset = get_zfs_dataset(sahandle, pathp, B_TRUE)) == NULL) {
1286                 char *p;

1288                 if (pathp == path) {
1289                         (void) strlcpy(newpath, path, sizeof (newpath));
1290                         pathp = newpath;
1291                 }

1293                 /*
1294                  * Make sure only one leading '/' This condition came
1295                  * about when using HAStoragePlus which insisted on
1296                  * putting an extra leading '/' in the ZFS path
1297                  * name. The problem is fixed in other areas, but this
1298                  * will catch any other ways that a double slash might
1299                  * get introduced.
1300                  */
1301                 while (*pathp == '/' && *(pathp + 1) == '/')
1302                         pathp++;

1304                 /*
1305                  * chop off part of path, but if we are at root then
1306                  * make sure path is a /
1307                  */
1308                 if ((strlen(pathp) > 1) && (p = strrchr(pathp, '/'))) {
1309                         if (pathp == p) {
1310                                 *(p + 1) = '\0';  /* skip over /, root case */
1311                         } else {
1312                                 *p = '\0';
1313                         }
1314                 } else {
1315                         return (EINVAL);
1316                 }
1317         }

1299         libhandle = libzfs_init();
1300         if (libhandle != NULL) {
1301                 char *resource_name;

1319         i = (sh->sh_path ? strlen(sh->sh_path) : 0);
1320         sh->sh_size = i;

1322         j = (sh->sh_res ? strlen(sh->sh_res) : 0);
1323         sh->sh_size += j;
1324         SMAX(i, j);

1326         j = (sh->sh_fstype ? strlen(sh->sh_fstype) : 0);
1327         sh->sh_size += j;
1328         SMAX(i, j);

1330         j = (sh->sh_opts ? strlen(sh->sh_opts) : 0);
1331         sh->sh_size += j;
1332         SMAX(i, j);

1334         j = (sh->sh_descr ? strlen(sh->sh_descr) : 0);
1335         sh->sh_size += j;
1336         SMAX(i, j);

1338         resource_name = sa_get_resource_attr(resource, "name");

1340         err = zfs_deleg_share_nfs(handle->zfs_libhandle, dataset, path,
1324                 err = zfs_deleg_share_nfs(libhandle, dataset, path,
1341             resource_name, exportdata, sh, i, operation);
1342         if (err == SA_OK)
```

```
1343                 sa_update_sharetab_ts(handle);
1327                         sa_update_sharetab_ts(sahandle);
1344         else
1345                 err = errno;
1346         if (resource_name != NULL)
1330                 if (resource_name)
1347                 sa_free_attr_string(resource_name);

1333                 libzfs_fini(libhandle);
1334         }
1349         free(dataset);
1350         return (err);
1351 }

1353 /*
1354  * sa_get_zfs_handle(handle)
1355  *
1356  * Given an sa_handle_t, return the libzfs_handle_t *. This is only
1357  * used internally by libzfs. Needed in order to avoid including
1358  * libshare_impl.h in libzfs.
1359  */

1361 libzfs_handle_t *
1362 sa_get_zfs_handle(sa_handle_t handle)
1363 {
1364         return (handle->zfs_libhandle);
1350         sa_handle_impl_t implhandle = (sa_handle_impl_t)handle;

1352         return (implhandle->zfs_libhandle);
1353 }

1355 /*
1356  * sa_get_zfs_info(libzfs, path, mountpoint, dataset)
1357  *
1358  * Find the ZFS dataset and mountpoint for a given path
1359  */
1360 int
1361 sa_zfs_get_info(libzfs_handle_t *libzfs, char *path, char *mountpointp,
1362     char *datasetp)
1363 {
1364         get_all_cbdata_t cb = { 0 };
1365         int i;
1366         char mountpoint[ZFS_MAXPROPLEN];
1367         char dataset[ZFS_MAXPROPLEN];
1368         char canmount[ZFS_MAXPROPLEN];
1369         char *dp;
1370         int count;
1371         int ret = 0;

1373         cb.cb_types = ZFS_TYPE_FILESYSTEM;

1375         if (libzfs == NULL)
1376                 return (0);

1378         (void) zfs_iter_root(libzfs, get_one_filesystem, &cb);
1379         count = cb.cb_used;

1381         qsort(cb.cb_handles, count, sizeof (void *), mountpoint_compare);
1382         for (i = 0; i < count; i++) {
1383                 /* must have a mountpoint */
1384                 if (zfs_prop_get(cb.cb_handles[i], ZFS_PROP_MOUNTPOINT,
1385                     mountpoint, sizeof (mountpoint),
1386                     NULL, NULL, 0, B_FALSE) != 0) {
1387                         /* no mountpoint */
1388                         continue;
1389                 }
```



```
1391                 /* mountpoint must be a path */
1392                 if (strcmp(mountpoint, ZFS_MOUNTPOINT_NONE) == 0 ||
1393                     strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) == 0) {
1394                         /*
1395                          * Search mmttab for mountpoint
1396                          */

1398                         if (get_legacy_mountpoint(path, dataset,
1399                             ZFS_MAXPROPLEN, mountpoint,
1400                             ZFS_MAXPROPLEN) == 0) {
1401                                 ret = 1;
1402                                 break;
1403                         }
1404                         continue;
1405                 }

1407                 /* canmount must be set */
1408                 canmount[0] = '\0';
1409                 if (zfs_prop_get(cb.cb_handles[i], ZFS_PROP_CANMOUNT, canmount,
1410                     sizeof (canmount), NULL, NULL, 0, B_FALSE) != 0 ||
1411                     strcmp(canmount, "off") == 0)
1412                         continue;

1414                 /*
1415                  * have a mountable handle but want to skip those marked none
1416                  * and legacy
1417                  */
1418                 if (strcmp(mountpoint, path) == 0) {
1419                         dp = (char *)zfs_get_name(cb.cb_handles[i]);
1420                         if (dp != NULL) {
1421                                 if (datasetp != NULL)
1422                                         (void) strcpy(datasetp, dp);
1423                                 if (mountpointp != NULL)
1424                                         (void) strcpy(mountpointp, mountpoint);
1425                                 ret = 1;
1426                         }
1427                         break;
1428                 }

1430         }

1432         return (ret);
1365 }

1367 /*
1368  * This method builds values for "sharesmb" property from the
1369  * nvlist argument. The values are returned in sharesmb_val variable.
1370  */
1371 static int
1372 sa_zfs_sprintf_new_prop(nvlist_t *nvl, char *sharesmb_val)
1373 {
1374         char cur_val[ZFS_MAXPROPLEN];
1442         char cur_val[MAXPATHLEN];
1375         char *name, *val;
1376         nvpair_t *cur;
1377         int err = 0;

1379         cur = nvlist_next_nvpair(nvl, NULL);
1380         while (cur != NULL) {
1381                 name = nvpair_name(cur);
1382                 err = nvpair_value_string(cur, &val);
1383                 if ((err != 0) || (name == NULL) || (val == NULL))
1384                         return (-1);

1386                 (void) snprintf(cur_val, ZFS_MAXPROPLEN, "%s=%s,", name, val);
```

```
1387                (void) strlcat(sharesmb_val, cur_val, ZFS_MAXPROPLEN);
1454                (void) snprintf(cur_val, MAXPATHLEN, "%s=%s,", name, val);
1455                (void) strlcat(sharesmb_val, cur_val, MAXPATHLEN);

1389                cur = nvlist_next_nvpair(nvl, cur);
1390        }

1392        return (0);
1393 }

1395 /*
1396  * This method builds values for "sharesmb" property from values
1397  * already existing on the share. The properties set via sa_zfs_sprint_new_prop
1398  * method are passed in sharesmb_val. If a existing property is already
1399  * set via sa_zfs_sprint_new_prop method, then they are not appended
1400  * to the sharesmb_val string. The returned sharesmb_val string is a combination
1401  * of new and existing values for 'sharesmb' property.
1402  */
1403 static int
1404 sa_zfs_sprintf_existing_prop(zfs_handle_t *handle, char *sharesmb_val)
1405 {
1406        char shareopts[ZFS_MAXPROPLEN], cur_val[ZFS_MAXPROPLEN];
1474        char shareopts[ZFS_MAXPROPLEN], cur_val[MAXPATHLEN];
1407        char *token, *last, *value;

1409        if (zfs_prop_get(handle, ZFS_PROP_SHARESMB, shareopts,
1410            sizeof (shareopts), NULL, NULL, 0, B_FALSE) != 0)
1411                return (-1);

1413        if (strstr(shareopts, "=") == NULL)
1414                return (0);

1416        for (token = strtok_r(shareopts, ",", &last); token != NULL;
1417            token = strtok_r(NULL, ",", &last)) {
1418                value = strchr(token, '=');
1419                if (value == NULL)
1420                        return (-1);
1421                *value++ = '\0';

1423                (void) snprintf(cur_val, ZFS_MAXPROPLEN, "%s=", token);
1491                (void) snprintf(cur_val, MAXPATHLEN, "%s=", token);
1424                if (strstr(sharesmb_val, cur_val) == NULL) {
1425                        (void) strlcat(cur_val, value, ZFS_MAXPROPLEN);
1426                        (void) strlcat(cur_val, ",", ZFS_MAXPROPLEN);
1427                        (void) strlcat(sharesmb_val, cur_val, ZFS_MAXPROPLEN);
1493                        (void) strlcat(cur_val, value, MAXPATHLEN);
1494                        (void) strlcat(cur_val, ",", MAXPATHLEN);
1495                        (void) strlcat(sharesmb_val, cur_val, MAXPATHLEN);
1428                }
1429        }

1431        return (0);
1432 }

1434 /*
1435  * Sets the share properties on a ZFS share. For now, this method sets only
1436  * the "sharesmb" property.
1437  *
1438  * This method includes building a comma seperated name-value string to be
1439  * set on the "sharesmb" property of a ZFS share. This name-value string is
1440  * build in 2 steps:
1441  *    - New property values given as name-value pair are set first.
1442  *    - Existing optionset properties, which are not part of the new properties
1443  *      passed in step 1, are appended to the newly set properties.
1444  */
1445 int
```



```
1446 sa_zfs_setprop(sa_handle_t handle, char *path, nvlist_t *nvl)
1447 {
1448        zfs_handle_t *z_fs;
1449        char sharesmb_val[ZFS_MAXPROPLEN];
1517        libzfs_handle_t *z_lib;
1518        char sharesmb_val[MAXPATHLEN];
1450        char *dataset, *lastcomma;

1452        if (nvlist_empty(nvl))
1453                return (0);

1455        if ((handle == NULL) || (path == NULL))
1456                return (-1);

1458        if ((dataset = get_zfs_dataset(handle, path, B_FALSE)) == NULL)
1459                return (-1);

1461        z_fs = zfs_open(handle->zfs_libhandle, dataset, ZFS_TYPE_DATASET);
1530        if ((z_lib = libzfs_init()) == NULL) {
1531                free(dataset);
1532                return (-1);
1533        }

1535        z_fs = zfs_open(z_lib, dataset, ZFS_TYPE_DATASET);
1462        if (z_fs == NULL) {
1463                free(dataset);
1538                libzfs_fini(z_lib);
1464                return (-1);
1465        }

1467        bzero(sharesmb_val, ZFS_MAXPROPLEN);
1542        bzero(sharesmb_val, MAXPATHLEN);
1468        if (sa_zfs_sprintf_new_prop(nvl, sharesmb_val) != 0) {
1469                free(dataset);
1470                zfs_close(z_fs);
1546                libzfs_fini(z_lib);
1471                return (-1);
1472        }

1474        if (sa_zfs_sprintf_existing_prop(z_fs, sharesmb_val) != 0) {
1475                free(dataset);
1476                zfs_close(z_fs);
1553                libzfs_fini(z_lib);
1477                return (-1);
1478        }

1480        lastcomma = strrchr(sharesmb_val, ',');
1481        if ((lastcomma != NULL) && (lastcomma[1] == '\0'))
1482                *lastcomma = '\0';

1484        (void) zfs_prop_set(z_fs, zfs_prop_to_name(ZFS_PROP_SHARESMB),
1485            sharesmb_val);
1486        free(dataset);
1487        zfs_close(z_fs);
1565        libzfs_fini(z_lib);

1489        return (0);
1490 }
______unchanged_portion_omitted_
```

```
**********************************************************
   55425 Tue Sep 10 06:31:59 2013
new/usr/src/lib/libshare/common/libsharecore.c
4095 minor cleanup up libshare
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 RackTop Systems.
#endif /* ! codereview */
 */

/*
 * core library for common functions across all config store types
 * and file systems to be exported. This includes legacy dfstab/sharetab
 * parsing. Need to eliminate XML where possible.
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <limits.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <libxml/parser.h>
#include <libxml/tree.h>
#include "libshare.h"
#include "libshare_impl.h"
#include <fcntl.h>
#include <thread.h>
#include <grp.h>
#include <limits.h>
#include <sys/param.h>
#include <signal.h>
#include <libintl.h>
#include <dirent.h>

#include <sharefs/share.h>
#include "sharetab.h"

#define DFSTAB_NOTICE_LINES     5
static char *notice[DFSTAB_NOTICE_LINES] =      {
        "# Do not modify this file directly.\n",
        "# Use the sharemgr(1m) command for all share management\n",
```



```
        "# This file is reconstructed and only maintained for backward\n",
        "# compatibility. Configuration lines could be lost.\n",
        "#\n"
};

#define STRNCAT(x, y, z)        (xmlChar *)strncat((char *)x, (char *)y, z)

/* will be much smaller, but this handles bad syntax in the file */
#define MAXARGSFORSHARE 256

static mutex_t sharetab_lock = DEFAULTMUTEX;
extern mutex_t sa_dfstab_lock;

/* used internally only */
typedef
struct sharelist {
    struct sharelist *next;
    int   persist;
    char *path;
    char *resource;
    char *fstype;
    char *options;
    char *description;
    char *group;
    char *origline;
    int lineno;
} xfs_sharelist_t;
static void parse_dfstab(sa_handle_t, char *, xmlNodePtr);
extern char *_sa_get_token(char *);
static void dfs_free_list(xfs_sharelist_t *);
/* prototypes */
void getlegacyconfig(sa_handle_t, char *, xmlNodePtr *);
extern sa_share_t _sa_add_share(sa_group_t, char *, int, int *, uint64_t);
extern sa_group_t _sa_create_group(sa_handle_t, char *);
extern sa_group_t _sa_create_group(sa_handle_impl_t, char *);
static void outdfstab(FILE *, xfs_sharelist_t *);
extern int _sa_remove_optionset(sa_optionset_t);
extern int set_node_share(void *, char *, char *);
extern void set_node_attr(void *, char *, char *);

/*
 * sablocksigs(*sigs)
 *
 * block important signals for a critical region. Arg is a pointer to
 * a sigset_t that is used later for the unblock.
 */
void
sablocksigs(sigset_t *sigs)
{
        sigset_t new;

        if (sigs != NULL) {
                (void) sigprocmask(SIG_BLOCK, NULL, &new);
                (void) sigaddset(&new, SIGHUP);
                (void) sigaddset(&new, SIGINT);
                (void) sigaddset(&new, SIGQUIT);
                (void) sigaddset(&new, SIGTSTP);
                (void) sigprocmask(SIG_SETMASK, &new, sigs);
        }
}
_____unchanged_portion_omitted_

/*
 * sa_is_security(optname, proto)
 *
 * Check to see if optname is a security (named optionset) specific
```

```
 743  * property for the specified protocol.
 744  */

 746 boolean_t
 675 int
 747 sa_is_security(char *optname, char *proto)
 748 {
 749         int ret = B_FALSE;
 678         int ret = 0;
 750         if (proto != NULL)
 751                 ret = sa_proto_security_prop(proto, optname);
 752         return (ret);
 753 }
______unchanged_portion_omitted_

 774 /*
 775  * sa_is_share(object)
 776  *
 777  * returns true of the object is of type "share".
 778  */

 780 boolean_t
 709 int
 781 sa_is_share(void *object)
 782 {
 783         if (object != NULL) {
 784                 if (strcmp((char *)((xmlNodePtr)object)->name, "share") == 0)
 785                 return (B_TRUE);
 714                 return (1);
 786         }
 787         return (B_FALSE);
 716         return (0);
 788 }
 789 /*
 790  * sa_is_resource(object)
 791  *
 792  * returns true of the object is of type "share".
 793  */

 795 boolean_t
 724 int
 796 sa_is_resource(void *object)
 797 {
 798         if (object != NULL) {
 799                 if (strcmp((char *)((xmlNodePtr)object)->name, "resource") == 0)
 800                         return (B_TRUE);
 729                         return (1);
 801         }
 802         return (B_FALSE);
 731         return (0);
 803 }
______unchanged_portion_omitted_

1435 /*
1436  * parse_sharetab(handle)
1437  *
1438  * Read the /etc/dfs/sharetab file and see which entries don't exist
1439  * in the repository. These shares are marked transient.  We also need
1440  * to see if they are ZFS shares since ZFS bypasses the SMF
1441  * repository.
1442  */

1444 int
1445 parse_sharetab(sa_handle_t handle)
1446 {
1447         xfs_sharelist_t *list, *tmplist;
```



```
1448         int err = 0;
1449         sa_share_t share;
1450         sa_group_t group;
1451         sa_group_t lgroup;
1452         char *groupname;
1453         int legacy = 0;
1454         char shareopts[MAXNAMLEN];

1456         list = get_share_list(&err);
1457         if (list == NULL)
1458                 return (legacy);

1460         lgroup = sa_get_group(handle, "default");

1462         for (tmplist = list; tmplist != NULL; tmplist = tmplist->next) {
1463                 group = NULL;
1464                 share = sa_find_share(handle, tmplist->path);
1465                 if (share != NULL) {
1466                         /*
1467                          * If this is a legacy share, mark as shared so we
1468                          * only update sharetab appropriately. We also keep
1469                          * the sharetab options in order to display for legacy
1470                          * share with no arguments.
1471                          */
1472                         set_node_attr(share, "shared", "true");
1473                         (void) snprintf(shareopts, MAXNAMLEN, "shareopts-%s",
1474                             tmplist->fstype);
1475                         set_node_attr(share, shareopts, tmplist->options);
1476                         continue;
1477                 }

1479                 /*
1480                  * This share is transient so needs to be
1481                  * added. Initially, this will be under
1482                  * default(legacy) unless it is a ZFS
1483                  * share. If zfs, we need a zfs group.
1484                  */
1485                 if (tmplist->resource != NULL &&
1486                     (groupname = strchr(tmplist->resource, '@')) != NULL) {
1487                         /* There is a defined group */
1488                         *groupname++ = '\0';
1489                         group = sa_get_group(handle, groupname);
1490                         if (group != NULL) {
1491                                 share = _sa_add_share(group, tmplist->path,
1492                                     SA_SHARE_TRANSIENT, &err,
1493                                     (uint64_t)SA_FEATURE_NONE);
1494                         } else {
1495                                 /*
1496                                  * While this case shouldn't
1497                                  * occur very often, it does
1498                                  * occur out of a "zfs set
1499                                  * sharenfs=off" when the
1500                                  * dataset is also set to
1501                                  * canmount=off. A warning
1502                                  * will then cause the zfs
1503                                  * command to abort. Since we
1504                                  * add it to the default list,
1505                                  * everything works properly
1506                                  * anyway and the library
1507                                  * doesn't need to give a
1508                                  * warning.
1509                                  */
1510                                 share = _sa_add_share(lgroup,
1511                                     tmplist->path, SA_SHARE_TRANSIENT,
1512                                     &err, (uint64_t)SA_FEATURE_NONE);
1513                         }
```

```
1514                } else {
1515                        if (sa_zfs_is_shared(handle, tmplist->path)) {
1516                                group = sa_get_group(handle, "zfs");
1517                                if (group == NULL) {
1518                                        group = sa_create_group(handle,
1519                                            "zfs", &err);
1520                                        if (group == NULL &&
1521                                            err == SA_NO_PERMISSION) {
1522                                                group = _sa_create_group(
1523                                                    handle, "zfs");
1452                                                    (sa_handle_impl_t)
1453                                                    handle,
1454                                                    "zfs");
1524                                        }
1525                                        if (group != NULL) {
1526                                                (void) sa_create_optionset(
1527                                                    group, tmplist->fstype);
1528                                                (void) sa_set_group_attr(group,
1529                                                    "zfs", "true");
1530                                        }
1531                                }
1532                                if (group != NULL) {
1533                                        share = _sa_add_share(group,
1534                                            tmplist->path, SA_SHARE_TRANSIENT,
1535                                            &err, (uint64_t)SA_FEATURE_NONE);
1536                                }
1537                        } else {
1538                                share = _sa_add_share(lgroup, tmplist->path,
1539                                    SA_SHARE_TRANSIENT, &err,
1540                                    (uint64_t)SA_FEATURE_NONE);
1541                        }
1542                }
1543                if (share == NULL)
1544                        (void) printf(dgettext(TEXT_DOMAIN,
1545                            "Problem with transient: %s\n"), sa_errorstr(err));
1546                if (share != NULL)
1547                        set_node_attr(share, "shared", "true");
1548                if (err == SA_OK) {
1549                        if (tmplist->options != NULL &&
1550                            strlen(tmplist->options) > 0) {
1551                                (void) sa_parse_legacy_options(share,
1552                                    tmplist->options, tmplist->fstype);
1553                        }
1554                        if (tmplist->resource != NULL &&
1555                            strcmp(tmplist->resource, "-") != 0)
1556                                set_node_attr(share, "resource",
1557                                    tmplist->resource);
1558                        if (tmplist->description != NULL) {
1559                                xmlNodePtr node;
1560                                node = xmlNewChild((xmlNodePtr)share, NULL,
1561                                    (xmlChar *)"description", NULL);
1562                                xmlNodeSetContent(node,
1563                                    (xmlChar *)tmplist->description);
1564                        }
1565                        legacy = 1;
1566                }
1567        }
1568        dfs_free_list(list);
1569        return (legacy);
1570 }

1572 /*
1573  * Get the transient shares from the sharetab (or other) file.  since
1574  * these are transient, they only appear in the working file and not
1575  * in a repository.
1576  */
```



```
1577 int
1578 gettransients(sa_handle_t handle, xmlNodePtr *root)
1509 gettransients(sa_handle_impl_t ihandle, xmlNodePtr *root)
1579 {
1580         int legacy = 0;
1581         int numproto;
1582         char **protocols = NULL;
1583         int i;

1585         if (root != NULL) {
1586                 if (*root == NULL)
1587                         *root = xmlNewNode(NULL, (xmlChar *)"sharecfg");
1588                 if (*root != NULL) {
1589                         legacy = parse_sharetab(handle);
1520                         legacy = parse_sharetab(ihandle);
1590                         numproto = sa_get_protocols(&protocols);
1591                         for (i = 0; i < numproto; i++)
1592                                 legacy |= sa_proto_get_transients(
1593                                     handle, protocols[i]);
1524                                     (sa_handle_t)ihandle, protocols[i]);
1594                         if (protocols != NULL)
1595                                 free(protocols);
1596                 }
1597         }
1598         return (legacy);
1599 }
______unchanged_portion_omitted_

2081 /*
2082  * sa_update_sharetab_ts(handle)
2083  *
2084  * Update the internal timestamp of when sharetab was last
2085  * changed. This needs to be public for ZFS to get at it.
2086  */

2088 void
2089 sa_update_sharetab_ts(sa_handle_t handle)
2090 {
2091         struct stat st;
2023         sa_handle_impl_t implhandle = (sa_handle_impl_t)handle;

2093         if (handle != NULL && stat(SA_LEGACY_SHARETAB, &st) == 0)
2094                 handle->tssharetab = TSTAMP(st.st_mtim);
2025         if (implhandle != NULL && stat(SA_LEGACY_SHARETAB, &st) == 0)
2026                 implhandle->tssharetab = TSTAMP(st.st_mtim);
2095 }
______unchanged_portion_omitted_

2168 /*
2169  * sa_needs_refresh(handle)
2170  *
2171  * Returns B_TRUE if the internal cache needs to be refreshed do to a
2172  * change by another process.  B_FALSE returned otherwise.
2173  */
2174 boolean_t
2175 sa_needs_refresh(sa_handle_t handle)
2176 {
2109         sa_handle_impl_t implhandle = (sa_handle_impl_t)handle;
2177         struct stat st;
2178         char *str;
2179         uint64_t tstamp;
2180         scf_simple_prop_t *prop;

2182         if (handle == NULL)
2183                 return (B_TRUE);
```

```
2185         /*
2186          * If sharetab has changed, then there was an external
2187          * change. Check sharetab first since it is updated by ZFS as
2188          * well as sharemgr.  This is where external ZFS changes are
2189          * caught.
2190          */
2191         if (stat(SA_LEGACY_SHARETAB, &st) == 0 &&
2192             TSTAMP(st.st_mtim) != handle->tssharetab)
2125             TSTAMP(st.st_mtim) != implhandle->tssharetab)
2193                 return (B_TRUE);

2195         /*
2196          * If sharetab wasn't changed, check whether there were any
2197          * SMF transactions that modified the config but didn't
2198          * initiate a share.  This is less common but does happen.
2199          */
2200         prop = scf_simple_prop_get(handle->scfhandle->handle,
2133         prop = scf_simple_prop_get(implhandle->scfhandle->handle,
2201             (const char *)SA_SVC_FMRI_BASE ":default", "state",
2202             "lastupdate");
2203         if (prop != NULL) {
2204                 str = scf_simple_prop_next_astring(prop);
2205                 if (str != NULL)
2206                         tstamp = strtoull(str, NULL, 0);
2207                 else
2208                         tstamp = 0;
2209                 scf_simple_prop_free(prop);
2210                 if (tstamp != handle->tstrans)
2143                 if (tstamp != implhandle->tstrans)
2211                         return (B_TRUE);
2212         }

2214         return (B_FALSE);
2215 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   49543 Tue Sep 10 06:31:59 2013
new/usr/src/lib/libshare/common/scfutil.c
4095 minor cleanup up libshare
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2013 RackTop Systems.
 */

#endif /* ! codereview */
/* helper functions for using libscf with sharemgr */

#include <libscf.h>
#include <libxml/parser.h>
#include <libxml/tree.h>
#include "libshare.h"
#include "libshare_impl.h"
#include "scfutil.h"
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <uuid/uuid.h>
#include <sys/param.h>
#include <signal.h>
#include <sys/time.h>
#include <libintl.h>

ssize_t scf_max_name_len;
extern struct sa_proto_plugin *sap_proto_list;
extern sa_handle_t get_handle_for_root(xmlNodePtr);
static void set_transaction_tstamp(sa_handle_t);
extern sa_handle_impl_t get_handle_for_root(xmlNodePtr);
static void set_transaction_tstamp(sa_handle_impl_t);
/*
 * The SMF facility uses some properties that must exist. We want to
 * skip over these when processing protocol options.
 */
static char *skip_props[] = {
        "modify_authorization",
        "action_authorization",
```



```
        "value_authorization",
        NULL
};
_____unchanged_portion_omitted_

/*
 * sa_scf_init()
 *
 * Must be called before using any of the SCF functions. Called by
 * sa_init() during the API setup.
 */

scfutilhandle_t *
sa_scf_init(sa_handle_t ihandle)
sa_scf_init(sa_handle_impl_t ihandle)
{
        scfutilhandle_t *handle;

        scf_max_name_len = scf_limit(SCF_LIMIT_MAX_NAME_LENGTH);
        if (scf_max_name_len <= 0)
                scf_max_name_len = SA_MAX_NAME_LEN + 1;

        handle = calloc(1, sizeof (scfutilhandle_t));
        if (handle == NULL)
                return (handle);

        ihandle->scfhandle = handle;
        handle->scf_state = SCH_STATE_INITIALIZING;
        handle->handle = scf_handle_create(SCF_VERSION);
        if (handle->handle == NULL) {
                free(handle);
                handle = NULL;
                (void) printf("libshare could not access SMF repository: %s\n",
                    scf_strerror(scf_error()));
                return (handle);
        }
        if (scf_handle_bind(handle->handle) != 0)
                goto err;

        handle->scope = scf_scope_create(handle->handle);
        handle->service = scf_service_create(handle->handle);
        handle->pg = scf_pg_create(handle->handle);

        /* Make sure we have sufficient SMF running */
        handle->instance = scf_instance_create(handle->handle);
        if (handle->scope == NULL || handle->service == NULL ||
            handle->pg == NULL || handle->instance == NULL)
                goto err;
        if (scf_handle_get_scope(handle->handle,
            SCF_SCOPE_LOCAL, handle->scope) != 0)
                goto err;
        if (scf_scope_get_service(handle->scope,
            SA_GROUP_SVC_NAME, handle->service) != 0)
                goto err;

        handle->scf_state = SCH_STATE_INIT;
        if (sa_get_instance(handle, "default") != SA_OK) {
                sa_group_t defgrp;
                defgrp = sa_create_group(ihandle, "default", NULL);
                defgrp = sa_create_group((sa_handle_t)ihandle, "default", NULL);
                /* Only NFS enabled for "default" group. */
                if (defgrp != NULL)
                        (void) sa_create_optionset(defgrp, "nfs");
        }

        return (handle);
```

```
 157         /* Error handling/unwinding */
 158 err:
 159         (void) sa_scf_fini(handle);
 160         (void) printf("libshare SMF initialization problem: %s\n",
 161             scf_strerror(scf_error()));
 162         return (NULL);
 163 }
______unchanged_portion_omitted_


1349 /*
1350  * sa_end_transaction(scfhandle, sahandle)
1351  *
1352  * Commit the changes that were added to the transaction in the
1353  * handle. Do all necessary cleanup.
1354  */

1356 int
1357 sa_end_transaction(scfutilhandle_t *handle, sa_handle_t sahandle)
1333 sa_end_transaction(scfutilhandle_t *handle, sa_handle_impl_t sahandle)
1358 {
1359         int ret = SA_OK;

1361         if (handle == NULL || handle->trans == NULL || sahandle == NULL) {
1362                 ret = SA_SYSTEM_ERR;
1363         } else {
1364                 if (scf_transaction_commit(handle->trans) < 0)
1365                         ret = SA_SYSTEM_ERR;
1366                 scf_transaction_destroy_children(handle->trans);
1367                 scf_transaction_destroy(handle->trans);
1368                 if (ret == SA_OK)
1369                         set_transaction_tstamp(sahandle);
1370                 handle->trans = NULL;
1371         }
1372         return (ret);
1373 }
______unchanged_portion_omitted_

1393 /*
1394  * set_transaction_tstamp(sahandle)
1395  *
1396  * After a successful transaction commit, update the timestamp of the
1397  * last transaction. This lets us detect changes from other processes.
1398  */
1399 static void
1400 set_transaction_tstamp(sa_handle_t sahandle)
1376 set_transaction_tstamp(sa_handle_impl_t sahandle)
1401 {
1402         char tstring[32];
1403         struct timeval tv;
1404         scfutilhandle_t *scfhandle;

1406         if (sahandle == NULL || sahandle->scfhandle == NULL)
1407                 return;

1409         scfhandle = sahandle->scfhandle;

1411         if (sa_get_instance(scfhandle, "default") != SA_OK)
1412                 return;

1414         if (gettimeofday(&tv, NULL) != 0)
1415                 return;

1417         if (sa_start_transaction(scfhandle, "*state") != SA_OK)
1418                 return;
```



```
1420         sahandle->tstrans = TSTAMP((*(timestruc_t *)&tv));
1421         (void) snprintf(tstring, sizeof (tstring), "%lld", sahandle->tstrans);
1422         if (sa_set_property(sahandle->scfhandle, "lastupdate", tstring) ==
1423             SA_OK) {
1424                 /*
1425                  * While best if it succeeds, a failure doesn't cause
1426                  * problems and we will ignore it anyway.
1427                  */
1428                 (void) scf_transaction_commit(scfhandle->trans);
1429                 scf_transaction_destroy_children(scfhandle->trans);
1430                 scf_transaction_destroy(scfhandle->trans);
1431         } else {
1432                 sa_abort_transaction(scfhandle);
1433         }
1434 }
______unchanged_portion_omitted_

1650 /*
1651  * sa_commit_share(handle, group, share)
1652  *
1653  *      Commit this share to the repository.
1654  *      properties are added if they exist but can be added later.
1655  *      Need to add to dfstab and sharetab, if appropriate.
1656  */
1657 int
1658 sa_commit_share(scfutilhandle_t *handle, sa_group_t group, sa_share_t share)
1659 {
1660         int ret = SA_OK;
1661         char *groupname;
1662         char *name;
1663         char *description;
1664         char *sharename;
1665         ssize_t proplen;
1666         char *propstring;

1668         /*
1669          * Don't commit in the zfs group. We do commit legacy
1670          * (default) and all other groups/shares. ZFS is handled
1671          * through the ZFS configuration rather than SMF.
1672          */

1674         groupname = sa_get_group_attr(group, "name");
1675         if (groupname != NULL) {
1676                 if (strcmp(groupname, "zfs") == 0) {
1677                         /*
1678                          * Adding to the ZFS group will result in the sharenfs
1679                          * property being set but we don't want to do anything
1680                          * SMF related at this point.
1681                          */
1682                         sa_free_attr_string(groupname);
1683                         return (ret);
1684                 }
1685         }

1687         proplen = get_scf_limit(SCF_LIMIT_MAX_VALUE_LENGTH);
1688         propstring = malloc(proplen);
1689         if (propstring == NULL)
1690                 ret = SA_NO_MEMORY;

1692         if (groupname != NULL && ret == SA_OK) {
1693                 ret = sa_get_instance(handle, groupname);
1694                 sa_free_attr_string(groupname);
1695                 groupname = NULL;
1696                 sharename = sa_get_share_attr(share, "id");
1697                 if (sharename == NULL) {
```

```
1698                        /* slipped by */
1699                        char shname[SA_SHARE_UUID_BUFLEN];
1700                        generate_unique_sharename(shname);
1701                        (void) xmlSetProp((xmlNodePtr)share, (xmlChar *)"id",
1702                            (xmlChar *)shname);
1703                        sharename = strdup(shname);
1704                }
1705                if (sharename != NULL) {
1706                        sigset_t old, new;
1707                        /*
1708                         * Have a share name allocated so create a pgroup for
1709                         * it. It may already exist, but that is OK.  In order
1710                         * to avoid creating a share pgroup that doesn't have
1711                         * a path property, block signals around the critical
1712                         * region of creating the share pgroup and props.
1713                         */
1714                        (void) sigprocmask(SIG_BLOCK, NULL, &new);
1715                        (void) sigaddset(&new, SIGHUP);
1716                        (void) sigaddset(&new, SIGINT);
1717                        (void) sigaddset(&new, SIGQUIT);
1718                        (void) sigaddset(&new, SIGTSTP);
1719                        (void) sigprocmask(SIG_SETMASK, &new, &old);

1721                        ret = sa_create_pgroup(handle, sharename);
1722                        if (ret == SA_OK) {
1723                                /*
1724                                 * Now start the transaction for the
1725                                 * properties that define this share. They may
1726                                 * exist so attempt to update before create.
1727                                 */
1728                                ret = sa_start_transaction(handle, sharename);
1729                        }
1730                        if (ret == SA_OK) {
1731                                name = sa_get_share_attr(share, "path");
1732                                if (name != NULL) {
1733                                        /*
1734                                         * There needs to be a path
1735                                         * for a share to exist.
1736                                         */
1737                                        ret = sa_set_property(handle, "path",
1738                                            name);
1739                                        sa_free_attr_string(name);
1740                                } else {
1741                                        ret = SA_NO_MEMORY;
1742                                }
1743                        }
1744                        if (ret == SA_OK) {
1745                                name = sa_get_share_attr(share, "drive-letter");
1746                                if (name != NULL) {
1747                                        /* A drive letter may exist for SMB */
1748                                        ret = sa_set_property(handle,
1749                                            "drive-letter", name);
1750                                        sa_free_attr_string(name);
1751                                }
1752                        }
1753                        if (ret == SA_OK) {
1754                                name = sa_get_share_attr(share, "exclude");
1755                                if (name != NULL) {
1756                                        /*
1757                                         * In special cases need to
1758                                         * exclude proto enable.
1759                                         */
1760                                        ret = sa_set_property(handle,
1761                                            "exclude", name);
1762                                        sa_free_attr_string(name);
1763                                }
```



```
1764                        }
1765                        if (ret == SA_OK) {
1766                                /*
1767                                 * If there are resource names, bundle them up
1768                                 * and save appropriately.
1769                                 */
1770                                ret = sa_set_resource_property(handle, share);
1771                        }

1773                        if (ret == SA_OK) {
1774                                description = sa_get_share_description(share);
1775                                if (description != NULL) {
1776                                        ret = sa_set_property(handle,
1777                                            "description",
1778                                            description);
1779                                        sa_free_share_description(description);
1780                                }
1781                        }
1782                        /* Make sure we cleanup the transaction */
1783                        if (ret == SA_OK) {
1784                                sa_handle_t sahandle;
1785                                sahandle = sa_find_group_handle(group);
1760                                sa_handle_impl_t sahandle;
1761                                sahandle = (sa_handle_impl_t)
1762                                    sa_find_group_handle(group);
1786                                if (sahandle != NULL)
1787                                        ret = sa_end_transaction(handle,
1788                                            sahandle);
1789                                else
1790                                        ret = SA_SYSTEM_ERR;
1791                        } else {
1792                                sa_abort_transaction(handle);
1793                        }

1795                        (void) sigprocmask(SIG_SETMASK, &old, NULL);

1797                        free(sharename);
1798                }
1799        }
1800        if (ret == SA_SYSTEM_ERR) {
1801                int err = scf_error();
1802                if (err == SCF_ERROR_PERMISSION_DENIED)
1803                        ret = SA_NO_PERMISSION;
1804        }
1805        if (propstring != NULL)
1806                free(propstring);
1807        if (groupname != NULL)
1808                sa_free_attr_string(groupname);

1810        return (ret);
1811 }
```

*_____unchanged_portion_omitted_*