```
**********************************************************
  149883 Thu Jan  8 13:02:09 2015
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
5515 dataset user hold doesn't reject empty tags
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Portions Copyright 2011 Martin Matuska
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2014, Joyent, Inc. All rights reserved.
 * Copyright (c) 2011, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 * Copyright (c) 2013 Steven Hartland. All rights reserved.
 * Copyright (c) 2015, Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2014, Nexenta Systems, Inc. All rights reserved.
 */

/*
 * ZFS ioctls.
 *
 * This file handles the ioctls to /dev/zfs, used for configuring ZFS storage
 * pools and filesystems, e.g. with /sbin/zfs and /sbin/zpool.
 *
 * There are two ways that we handle ioctls: the legacy way where almost
 * all of the logic is in the ioctl callback, and the new way where most
 * of the marshalling is handled in the common entry point, zfsdev_ioctl().
 *
 * Non-legacy ioctls should be registered by calling
 * zfs_ioctl_register() from zfs_ioctl_init().  The ioctl is invoked
 * from userland by lzc_ioctl().
 *
 * The registration arguments are as follows:
 *
 * const char *name
 *   The name of the ioctl.  This is used for history logging.  If the
 *   ioctl returns successfully (the callback returns 0), and allow_log
 *   is true, then a history log entry will be recorded with the input &
 *   output nvlists.  The log entry can be printed with "zpool history -i".
 *
 * zfs_ioc_t ioc
 *   The ioctl request number, which userland will pass to ioctl(2).
 *   The ioctl numbers can change from release to release, because
 *   the caller (libzfs) must be matched to the kernel.
 *
 * zfs_secpolicy_func_t *secpolicy
```



```
 *   This function will be called before the zfs_ioc_func_t, to
 *   determine if this operation is permitted.  It should return EPERM
 *   on failure, and 0 on success.  Checks include determining if the
 *   dataset is visible in this zone, and if the user has either all
 *   zfs privileges in the zone (SYS_MOUNT), or has been granted permission
 *   to do this operation on this dataset with "zfs allow".
 *
 * zfs_ioc_namecheck_t namecheck
 *   This specifies what to expect in the zfs_cmd_t:zc_name -- a pool
 *   name, a dataset name, or nothing.  If the name is not well-formed,
 *   the ioctl will fail and the callback will not be called.
 *   Therefore, the callback can assume that the name is well-formed
 *   (e.g. is null-terminated, doesn't have more than one '@' character,
 *   doesn't have invalid characters).
 *
 * zfs_ioc_poolcheck_t pool_check
 *   This specifies requirements on the pool state.  If the pool does
 *   not meet them (is suspended or is readonly), the ioctl will fail
 *   and the callback will not be called.  If any checks are specified
 *   (i.e. it is not POOL_CHECK_NONE), namecheck must not be NO_NAME.
 *   Multiple checks can be or-ed together (e.g. POOL_CHECK_SUSPENDED |
 *   POOL_CHECK_READONLY).
 *
 * boolean_t smush_outnvlist
 *   If smush_outnvlist is true, then the output is presumed to be a
 *   list of errors, and it will be "smushed" down to fit into the
 *   caller's buffer, by removing some entries and replacing them with a
 *   single "N_MORE_ERRORS" entry indicating how many were removed.  See
 *   nvlist_smush() for details.  If smush_outnvlist is false, and the
 *   outnvlist does not fit into the userland-provided buffer, then the
 *   ioctl will fail with ENOMEM.
 *
 * zfs_ioc_func_t *func
 *   The callback function that will perform the operation.
 *
 *   The callback should return 0 on success, or an error number on
 *   failure.  If the function fails, the userland ioctl will return -1,
 *   and errno will be set to the callback's return value.  The callback
 *   will be called with the following arguments:
 *
 *   const char *name
 *     The name of the pool or dataset to operate on, from
 *     zfs_cmd_t:zc_name.  The 'namecheck' argument specifies the
 *     expected type (pool, dataset, or none).
 *
 *   nvlist_t *innvl
 *     The input nvlist, deserialized from zfs_cmd_t:zc_nvlist_src.  Or
 *     NULL if no input nvlist was provided.  Changes to this nvlist are
 *     ignored.  If the input nvlist could not be deserialized, the
 *     ioctl will fail and the callback will not be called.
 *
 *   nvlist_t *outnvl
 *     The output nvlist, initially empty.  The callback can fill it in,
 *     and it will be returned to userland by serializing it into
 *     zfs_cmd_t:zc_nvlist_dst.  If it is non-empty, and serialization
 *     fails (e.g. because the caller didn't supply a large enough
 *     buffer), then the overall ioctl will fail.  See the
 *     'smush_nvlist' argument above for additional behaviors.
 *
 *     There are two typical uses of the output nvlist:
 *       - To return state, e.g. property values.  In this case,
 *         smush_outnvlist should be false.  If the buffer was not large
 *         enough, the caller will reallocate a larger buffer and try
 *         the ioctl again.
 *
 *       - To return multiple errors from an ioctl which makes on-disk
```

```
 127  *         changes.  In this case, smush_outnvlist should be true.
 128  *         Ioctls which make on-disk modifications should generally not
 129  *         use the outnvl if they succeed, because the caller can not
 130  *         distinguish between the operation failing, and
 131  *         deserialization failing.
 132  */

 134 #include <sys/types.h>
 135 #include <sys/param.h>
 136 #include <sys/errno.h>
 137 #include <sys/uio.h>
 138 #include <sys/buf.h>
 139 #include <sys/modctl.h>
 140 #include <sys/open.h>
 141 #include <sys/file.h>
 142 #include <sys/kmem.h>
 143 #include <sys/conf.h>
 144 #include <sys/cmn_err.h>
 145 #include <sys/stat.h>
 146 #include <sys/zfs_ioctl.h>
 147 #include <sys/zfs_vfsops.h>
 148 #include <sys/zfs_znode.h>
 149 #include <sys/zap.h>
 150 #include <sys/spa.h>
 151 #include <sys/spa_impl.h>
 152 #include <sys/vdev.h>
 153 #include <sys/priv_impl.h>
 154 #include <sys/dmu.h>
 155 #include <sys/dsl_dir.h>
 156 #include <sys/dsl_dataset.h>
 157 #include <sys/dsl_prop.h>
 158 #include <sys/dsl_deleg.h>
 159 #include <sys/dmu_objset.h>
 160 #include <sys/dmu_impl.h>
 161 #include <sys/dmu_tx.h>
 162 #include <sys/ddi.h>
 163 #include <sys/sunddi.h>
 164 #include <sys/sunldi.h>
 165 #include <sys/policy.h>
 166 #include <sys/zone.h>
 167 #include <sys/nvpair.h>
 168 #include <sys/pathname.h>
 169 #include <sys/mount.h>
 170 #include <sys/sdt.h>
 171 #include <sys/fs/zfs.h>
 172 #include <sys/zfs_ctldir.h>
 173 #include <sys/zfs_dir.h>
 174 #include <sys/zfs_onexit.h>
 175 #include <sys/zvol.h>
 176 #include <sys/dsl_scan.h>
 177 #include <sharefs/share.h>
 178 #include <sys/dmu_objset.h>
 179 #include <sys/dmu_send.h>
 180 #include <sys/dsl_destroy.h>
 181 #include <sys/dsl_bookmark.h>
 182 #include <sys/dsl_userhold.h>
 183 #include <sys/zfeature.h>

 185 #include "zfs_namecheck.h"
 186 #include "zfs_prop.h"
 187 #include "zfs_deleg.h"
 188 #include "zfs_comutil.h"

 190 extern struct modlfs zfs_modlfs;

 192 extern void zfs_init(void);
```



```
 193 extern void zfs_fini(void);

 195 ldi_ident_t zfs_li = NULL;
 196 dev_info_t *zfs_dip;

 198 uint_t zfs_fsyncer_key;
 199 extern uint_t rrw_tsd_key;
 200 static uint_t zfs_allow_log_key;

 202 typedef int zfs_ioc_legacy_func_t(zfs_cmd_t *);
 203 typedef int zfs_ioc_func_t(const char *, nvlist_t *, nvlist_t *);
 204 typedef int zfs_secpolicy_func_t(zfs_cmd_t *, nvlist_t *, cred_t *);

 206 typedef enum {
 207         NO_NAME,
 208         POOL_NAME,
 209         DATASET_NAME
 210 } zfs_ioc_namecheck_t;
_____unchanged_portion_omitted_

5098 /*
5099  * innvl: {
5100  *     "holds" -> { snapname -> holdname (string), ... }
5101  *     (optional) "cleanup_fd" -> fd (int32)
5102  * }
5103  *
5104  * outnvl: {
5105  *     snapname -> error value (int32)
5106  *     ...
5107  * }
5108  */
5109 /* ARGSUSED */
5110 static int
5111 zfs_ioc_hold(const char *pool, nvlist_t *args, nvlist_t *errlist)
5112 {
5113         nvpair_t *pair;
5114 #endif /* ! codereview */
5115         nvlist_t *holds;
5116         int cleanup_fd = -1;
5117         int error;
5118         minor_t minor = 0;

5120         error = nvlist_lookup_nvlist(args, "holds", &holds);
5121         if (error != 0)
5122                 return (SET_ERROR(EINVAL));

5124         /* make sure the user didn't pass us any invalid (empty) tags */
5125         for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
5126             pair = nvlist_next_nvpair(holds, pair)) {
5127                 char *htag;

5129                 error = nvpair_value_string(pair, &htag);
5130                 if (error != 0)
5131                         return (SET_ERROR(error));

5133                 if (strlen(htag) == 0)
5134                         return (SET_ERROR(EINVAL));
5135         }
5136 #endif /* ! codereview */

5138         if (nvlist_lookup_int32(args, "cleanup_fd", &cleanup_fd) == 0) {
5139                 error = zfs_onexit_fd_hold(cleanup_fd, &minor);
5140                 if (error != 0)
5141                         return (error);
5142         }
```

```
5144         error = dsl_dataset_user_hold(holds, minor, errlist);
5145         if (minor != 0)
5146                 zfs_onexit_fd_rele(cleanup_fd);
5147         return (error);
5148 }

5150 /*
5151  * innvl is not used.
5152  *
5153  * outnvl: {
5154  *    holdname -> time added (uint64 seconds since epoch)
5155  *    ...
5156  * }
5157  */
5158 /* ARGSUSED */
5159 static int
5160 zfs_ioc_get_holds(const char *snapname, nvlist_t *args, nvlist_t *outnvl)
5161 {
5162         return (dsl_dataset_get_holds(snapname, outnvl));
5163 }

5165 /*
5166  * innvl: {
5167  *     snapname -> { holdname, ... }
5168  *     ...
5169  * }
5170  *
5171  * outnvl: {
5172  *     snapname -> error value (int32)
5173  *     ...
5174  * }
5175  */
5176 /* ARGSUSED */
5177 static int
5178 zfs_ioc_release(const char *pool, nvlist_t *holds, nvlist_t *errlist)
5179 {
5180         return (dsl_dataset_user_release(holds, errlist));
5181 }

5183 /*
5184  * inputs:
5185  * zc_name              name of new filesystem or snapshot
5186  * zc_value             full name of old snapshot
5187  *
5188  * outputs:
5189  * zc_cookie            space in bytes
5190  * zc_objset_type       compressed space in bytes
5191  * zc_perm_action       uncompressed space in bytes
5192  */
5193 static int
5194 zfs_ioc_space_written(zfs_cmd_t *zc)
5195 {
5196         int error;
5197         dsl_pool_t *dp;
5198         dsl_dataset_t *new, *old;

5200         error = dsl_pool_hold(zc->zc_name, FTAG, &dp);
5201         if (error != 0)
5202                 return (error);
5203         error = dsl_dataset_hold(dp, zc->zc_name, FTAG, &new);
5204         if (error != 0) {
5205                 dsl_pool_rele(dp, FTAG);
5206                 return (error);
5207         }
5208         error = dsl_dataset_hold(dp, zc->zc_value, FTAG, &old);
5209         if (error != 0) {
```



```
5210                 dsl_dataset_rele(new, FTAG);
5211                 dsl_pool_rele(dp, FTAG);
5212                 return (error);
5213         }

5215         error = dsl_dataset_space_written(old, new, &zc->zc_cookie,
5216             &zc->zc_objset_type, &zc->zc_perm_action);
5217         dsl_dataset_rele(old, FTAG);
5218         dsl_dataset_rele(new, FTAG);
5219         dsl_pool_rele(dp, FTAG);
5220         return (error);
5221 }

5223 /*
5224  * innvl: {
5225  *     "firstsnap" -> snapshot name
5226  * }
5227  *
5228  * outnvl: {
5229  *     "used" -> space in bytes
5230  *     "compressed" -> compressed space in bytes
5231  *     "uncompressed" -> uncompressed space in bytes
5232  * }
5233  */
5234 static int
5235 zfs_ioc_space_snaps(const char *lastsnap, nvlist_t *innvl, nvlist_t *outnvl)
5236 {
5237         int error;
5238         dsl_pool_t *dp;
5239         dsl_dataset_t *new, *old;
5240         char *firstsnap;
5241         uint64_t used, comp, uncomp;

5243         if (nvlist_lookup_string(innvl, "firstsnap", &firstsnap) != 0)
5244                 return (SET_ERROR(EINVAL));

5246         error = dsl_pool_hold(lastsnap, FTAG, &dp);
5247         if (error != 0)
5248                 return (error);

5250         error = dsl_dataset_hold(dp, lastsnap, FTAG, &new);
5251         if (error != 0) {
5252                 dsl_pool_rele(dp, FTAG);
5253                 return (error);
5254         }
5255         error = dsl_dataset_hold(dp, firstsnap, FTAG, &old);
5256         if (error != 0) {
5257                 dsl_dataset_rele(new, FTAG);
5258                 dsl_pool_rele(dp, FTAG);
5259                 return (error);
5260         }

5262         error = dsl_dataset_space_wouldfree(old, new, &used, &comp, &uncomp);
5263         dsl_dataset_rele(old, FTAG);
5264         dsl_dataset_rele(new, FTAG);
5265         dsl_pool_rele(dp, FTAG);
5266         fnvlist_add_uint64(outnvl, "used", used);
5267         fnvlist_add_uint64(outnvl, "compressed", comp);
5268         fnvlist_add_uint64(outnvl, "uncompressed", uncomp);
5269         return (error);
5270 }

5272 /*
5273  * innvl: {
5274  *     "fd" -> file descriptor to write stream to (int32)
5275  *     (optional) "fromsnap" -> full snap name to send an incremental from
```

```
5276  *     (optional) "largeblockok" -> (value ignored)
5277  *         indicates that blocks > 128KB are permitted
5278  *     (optional) "embedok" -> (value ignored)
5279  *         presence indicates DRR_WRITE_EMBEDDED records are permitted
5280  * }
5281  *
5282  * outnvl is unused
5283  */
5284 /* ARGSUSED */
5285 static int
5286 zfs_ioc_send_new(const char *snapname, nvlist_t *innvl, nvlist_t *outnvl)
5287 {
5288         int error;
5289         offset_t off;
5290         char *fromname = NULL;
5291         int fd;
5292         boolean_t largeblockok;
5293         boolean_t embedok;
5294
5295         error = nvlist_lookup_int32(innvl, "fd", &fd);
5296         if (error != 0)
5297                 return (SET_ERROR(EINVAL));
5298
5299         (void) nvlist_lookup_string(innvl, "fromsnap", &fromname);
5300
5301         largeblockok = nvlist_exists(innvl, "largeblockok");
5302         embedok = nvlist_exists(innvl, "embedok");
5303
5304         file_t *fp = getf(fd);
5305         if (fp == NULL)
5306                 return (SET_ERROR(EBADF));
5307
5308         off = fp->f_offset;
5309         error = dmu_send(snapname, fromname, embedok, largeblockok,
5310             fd, fp->f_vnode, &off);
5311
5312         if (VOP_SEEK(fp->f_vnode, fp->f_offset, &off, NULL) == 0)
5313                 fp->f_offset = off;
5314         releasef(fd);
5315         return (error);
5316 }
5317
5318 /*
5319  * Determine approximately how large a zfs send stream will be -- the number
5320  * of bytes that will be written to the fd supplied to zfs_ioc_send_new().
5321  *
5322  * innvl: {
5323  *     (optional) "fromsnap" -> full snap name to send an incremental from
5324  * }
5325  *
5326  * outnvl: {
5327  *     "space" -> bytes of space (uint64)
5328  * }
5329  */
5330 static int
5331 zfs_ioc_send_space(const char *snapname, nvlist_t *innvl, nvlist_t *outnvl)
5332 {
5333         dsl_pool_t *dp;
5334         dsl_dataset_t *fromsnap = NULL;
5335         dsl_dataset_t *tosnap;
5336         int error;
5337         char *fromname;
5338         uint64_t space;
5339
5340         error = dsl_pool_hold(snapname, FTAG, &dp);
5341         if (error != 0)
```



```
5342                 return (error);
5343
5344         error = dsl_dataset_hold(dp, snapname, FTAG, &tosnap);
5345         if (error != 0) {
5346                 dsl_pool_rele(dp, FTAG);
5347                 return (error);
5348         }
5349
5350         error = nvlist_lookup_string(innvl, "fromsnap", &fromname);
5351         if (error == 0) {
5352                 error = dsl_dataset_hold(dp, fromname, FTAG, &fromsnap);
5353                 if (error != 0) {
5354                         dsl_dataset_rele(tosnap, FTAG);
5355                         dsl_pool_rele(dp, FTAG);
5356                         return (error);
5357                 }
5358         }
5359
5360         error = dmu_send_estimate(tosnap, fromsnap, &space);
5361         fnvlist_add_uint64(outnvl, "space", space);
5362
5363         if (fromsnap != NULL)
5364                 dsl_dataset_rele(fromsnap, FTAG);
5365         dsl_dataset_rele(tosnap, FTAG);
5366         dsl_pool_rele(dp, FTAG);
5367         return (error);
5368 }
5369
5370
5371 static zfs_ioc_vec_t zfs_ioc_vec[ZFS_IOC_LAST - ZFS_IOC_FIRST];
5372
5373 static void
5374 zfs_ioctl_register_legacy(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5375     zfs_secpolicy_func_t *secpolicy, zfs_ioc_namecheck_t namecheck,
5376     boolean_t log_history, zfs_ioc_poolcheck_t pool_check)
5377 {
5378         zfs_ioc_vec_t *vec = &zfs_ioc_vec[ioc - ZFS_IOC_FIRST];
5379
5380         ASSERT3U(ioc, >=, ZFS_IOC_FIRST);
5381         ASSERT3U(ioc, <, ZFS_IOC_LAST);
5382         ASSERT3P(vec->zvec_legacy_func, ==, NULL);
5383         ASSERT3P(vec->zvec_func, ==, NULL);
5384
5385         vec->zvec_legacy_func = func;
5386         vec->zvec_secpolicy = secpolicy;
5387         vec->zvec_namecheck = namecheck;
5388         vec->zvec_allow_log = log_history;
5389         vec->zvec_pool_check = pool_check;
5390 }
5391
5392 /*
5393  * See the block comment at the beginning of this file for details on
5394  * each argument to this function.
5395  */
5396 static void
5397 zfs_ioctl_register(const char *name, zfs_ioc_t ioc, zfs_ioc_func_t *func,
5398     zfs_secpolicy_func_t *secpolicy, zfs_ioc_namecheck_t namecheck,
5399     zfs_ioc_poolcheck_t pool_check, boolean_t smush_outnvlist,
5400     boolean_t allow_log)
5401 {
5402         zfs_ioc_vec_t *vec = &zfs_ioc_vec[ioc - ZFS_IOC_FIRST];
5403
5404         ASSERT3U(ioc, >=, ZFS_IOC_FIRST);
5405         ASSERT3U(ioc, <, ZFS_IOC_LAST);
5406         ASSERT3P(vec->zvec_legacy_func, ==, NULL);
5407         ASSERT3P(vec->zvec_func, ==, NULL);
```

```
5409         /* if we are logging, the name must be valid */
5410         ASSERT(!allow_log || namecheck != NO_NAME);

5412         vec->zvec_name = name;
5413         vec->zvec_func = func;
5414         vec->zvec_secpolicy = secpolicy;
5415         vec->zvec_namecheck = namecheck;
5416         vec->zvec_pool_check = pool_check;
5417         vec->zvec_smush_outnvlist = smush_outnvlist;
5418         vec->zvec_allow_log = allow_log;
5419 }

5421 static void
5422 zfs_ioctl_register_pool(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5423     zfs_secpolicy_func_t *secpolicy, boolean_t log_history,
5424     zfs_ioc_poolcheck_t pool_check)
5425 {
5426         zfs_ioctl_register_legacy(ioc, func, secpolicy,
5427             POOL_NAME, log_history, pool_check);
5428 }

5430 static void
5431 zfs_ioctl_register_dataset_nolog(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5432     zfs_secpolicy_func_t *secpolicy, zfs_ioc_poolcheck_t pool_check)
5433 {
5434         zfs_ioctl_register_legacy(ioc, func, secpolicy,
5435             DATASET_NAME, B_FALSE, pool_check);
5436 }

5438 static void
5439 zfs_ioctl_register_pool_modify(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func)
5440 {
5441         zfs_ioctl_register_legacy(ioc, func, zfs_secpolicy_config,
5442             POOL_NAME, B_TRUE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5443 }

5445 static void
5446 zfs_ioctl_register_pool_meta(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5447     zfs_secpolicy_func_t *secpolicy)
5448 {
5449         zfs_ioctl_register_legacy(ioc, func, secpolicy,
5450             NO_NAME, B_FALSE, POOL_CHECK_NONE);
5451 }

5453 static void
5454 zfs_ioctl_register_dataset_read_secpolicy(zfs_ioc_t ioc,
5455     zfs_ioc_legacy_func_t *func, zfs_secpolicy_func_t *secpolicy)
5456 {
5457         zfs_ioctl_register_legacy(ioc, func, secpolicy,
5458             DATASET_NAME, B_FALSE, POOL_CHECK_SUSPENDED);
5459 }

5461 static void
5462 zfs_ioctl_register_dataset_read(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func)
5463 {
5464         zfs_ioctl_register_dataset_read_secpolicy(ioc, func,
5465             zfs_secpolicy_read);
5466 }

5468 static void
5469 zfs_ioctl_register_dataset_modify(zfs_ioc_t ioc, zfs_ioc_legacy_func_t *func,
5470         zfs_secpolicy_func_t *secpolicy)
5471 {
5472         zfs_ioctl_register_legacy(ioc, func, secpolicy,
5473             DATASET_NAME, B_TRUE, POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
```



```
5474 }

5476 static void
5477 zfs_ioctl_init(void)
5478 {
5479         zfs_ioctl_register("snapshot", ZFS_IOC_SNAPSHOT,
5480             zfs_ioc_snapshot, zfs_secpolicy_snapshot, POOL_NAME,
5481             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5483         zfs_ioctl_register("log_history", ZFS_IOC_LOG_HISTORY,
5484             zfs_ioc_log_history, zfs_secpolicy_log_history, NO_NAME,
5485             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_FALSE, B_FALSE);

5487         zfs_ioctl_register("space_snaps", ZFS_IOC_SPACE_SNAPS,
5488             zfs_ioc_space_snaps, zfs_secpolicy_read, DATASET_NAME,
5489             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5491         zfs_ioctl_register("send", ZFS_IOC_SEND_NEW,
5492             zfs_ioc_send_new, zfs_secpolicy_send_new, DATASET_NAME,
5493             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5495         zfs_ioctl_register("send_space", ZFS_IOC_SEND_SPACE,
5496             zfs_ioc_send_space, zfs_secpolicy_read, DATASET_NAME,
5497             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5499         zfs_ioctl_register("create", ZFS_IOC_CREATE,
5500             zfs_ioc_create, zfs_secpolicy_create_clone, DATASET_NAME,
5501             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5503         zfs_ioctl_register("clone", ZFS_IOC_CLONE,
5504             zfs_ioc_clone, zfs_secpolicy_create_clone, DATASET_NAME,
5505             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5507         zfs_ioctl_register("destroy_snaps", ZFS_IOC_DESTROY_SNAPS,
5508             zfs_ioc_destroy_snaps, zfs_secpolicy_destroy_snaps, POOL_NAME,
5509             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5511         zfs_ioctl_register("hold", ZFS_IOC_HOLD,
5512             zfs_ioc_hold, zfs_secpolicy_hold, POOL_NAME,
5513             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);
5514         zfs_ioctl_register("release", ZFS_IOC_RELEASE,
5515             zfs_ioc_release, zfs_secpolicy_release, POOL_NAME,
5516             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5518         zfs_ioctl_register("get_holds", ZFS_IOC_GET_HOLDS,
5519             zfs_ioc_get_holds, zfs_secpolicy_read, DATASET_NAME,
5520             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5522         zfs_ioctl_register("rollback", ZFS_IOC_ROLLBACK,
5523             zfs_ioc_rollback, zfs_secpolicy_rollback, DATASET_NAME,
5524             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_FALSE, B_TRUE);

5526         zfs_ioctl_register("bookmark", ZFS_IOC_BOOKMARK,
5527             zfs_ioc_bookmark, zfs_secpolicy_bookmark, POOL_NAME,
5528             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5530         zfs_ioctl_register("get_bookmarks", ZFS_IOC_GET_BOOKMARKS,
5531             zfs_ioc_get_bookmarks, zfs_secpolicy_read, DATASET_NAME,
5532             POOL_CHECK_SUSPENDED, B_FALSE, B_FALSE);

5534         zfs_ioctl_register("destroy_bookmarks", ZFS_IOC_DESTROY_BOOKMARKS,
5535             zfs_ioc_destroy_bookmarks, zfs_secpolicy_destroy_bookmarks,
5536             POOL_NAME,
5537             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY, B_TRUE, B_TRUE);

5539         /* IOCTLS that use the legacy function signature */
```

```
5541         zfs_ioctl_register_legacy(ZFS_IOC_POOL_FREEZE, zfs_ioc_pool_freeze,
5542             zfs_secpolicy_config, NO_NAME, B_FALSE, POOL_CHECK_READONLY);
5543
5544         zfs_ioctl_register_pool(ZFS_IOC_POOL_CREATE, zfs_ioc_pool_create,
5545             zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);
5546         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SCAN,
5547             zfs_ioc_pool_scan);
5548         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_UPGRADE,
5549             zfs_ioc_pool_upgrade);
5550         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ADD,
5551             zfs_ioc_vdev_add);
5552         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_REMOVE,
5553             zfs_ioc_vdev_remove);
5554         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SET_STATE,
5555             zfs_ioc_vdev_set_state);
5556         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_ATTACH,
5557             zfs_ioc_vdev_attach);
5558         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_DETACH,
5559             zfs_ioc_vdev_detach);
5560         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETPATH,
5561             zfs_ioc_vdev_setpath);
5562         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SETFRU,
5563             zfs_ioc_vdev_setfru);
5564         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_SET_PROPS,
5565             zfs_ioc_pool_set_props);
5566         zfs_ioctl_register_pool_modify(ZFS_IOC_VDEV_SPLIT,
5567             zfs_ioc_vdev_split);
5568         zfs_ioctl_register_pool_modify(ZFS_IOC_POOL_REGUID,
5569             zfs_ioc_pool_reguid);
5570
5571         zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_CONFIGS,
5572             zfs_ioc_pool_configs, zfs_secpolicy_none);
5573         zfs_ioctl_register_pool_meta(ZFS_IOC_POOL_TRYIMPORT,
5574             zfs_ioc_pool_tryimport, zfs_secpolicy_config);
5575         zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_FAULT,
5576             zfs_ioc_inject_fault, zfs_secpolicy_inject);
5577         zfs_ioctl_register_pool_meta(ZFS_IOC_CLEAR_FAULT,
5578             zfs_ioc_clear_fault, zfs_secpolicy_inject);
5579         zfs_ioctl_register_pool_meta(ZFS_IOC_INJECT_LIST_NEXT,
5580             zfs_ioc_inject_list_next, zfs_secpolicy_inject);
5581
5582         /*
5583          * pool destroy, and export don't log the history as part of
5584          * zfsdev_ioctl, but rather zfs_ioc_pool_export
5585          * does the logging of those commands.
5586          */
5587         zfs_ioctl_register_pool(ZFS_IOC_POOL_DESTROY, zfs_ioc_pool_destroy,
5588             zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);
5589         zfs_ioctl_register_pool(ZFS_IOC_POOL_EXPORT, zfs_ioc_pool_export,
5590             zfs_secpolicy_config, B_FALSE, POOL_CHECK_NONE);
5591
5592         zfs_ioctl_register_pool(ZFS_IOC_POOL_STATS, zfs_ioc_pool_stats,
5593             zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);
5594         zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_PROPS, zfs_ioc_pool_get_props,
5595             zfs_secpolicy_read, B_FALSE, POOL_CHECK_NONE);
5596
5597         zfs_ioctl_register_pool(ZFS_IOC_ERROR_LOG, zfs_ioc_error_log,
5598             zfs_secpolicy_inject, B_FALSE, POOL_CHECK_SUSPENDED);
5599         zfs_ioctl_register_pool(ZFS_IOC_DSOBJ_TO_DSNAME,
5600             zfs_ioc_dsobj_to_dsname,
5601             zfs_secpolicy_diff, B_FALSE, POOL_CHECK_SUSPENDED);
5602         zfs_ioctl_register_pool(ZFS_IOC_POOL_GET_HISTORY,
5603             zfs_ioc_pool_get_history,
5604             zfs_secpolicy_config, B_FALSE, POOL_CHECK_SUSPENDED);
```



```
5606         zfs_ioctl_register_pool(ZFS_IOC_POOL_IMPORT, zfs_ioc_pool_import,
5607             zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);
5608
5609         zfs_ioctl_register_pool(ZFS_IOC_CLEAR, zfs_ioc_clear,
5610             zfs_secpolicy_config, B_TRUE, POOL_CHECK_NONE);
5611         zfs_ioctl_register_pool(ZFS_IOC_POOL_REOPEN, zfs_ioc_pool_reopen,
5612             zfs_secpolicy_config, B_TRUE, POOL_CHECK_SUSPENDED);
5613
5614         zfs_ioctl_register_dataset_read(ZFS_IOC_SPACE_WRITTEN,
5615             zfs_ioc_space_written);
5616         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_RECVD_PROPS,
5617             zfs_ioc_objset_recvd_props);
5618         zfs_ioctl_register_dataset_read(ZFS_IOC_NEXT_OBJ,
5619             zfs_ioc_next_obj);
5620         zfs_ioctl_register_dataset_read(ZFS_IOC_GET_FSACL,
5621             zfs_ioc_get_fsacl);
5622         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_STATS,
5623             zfs_ioc_objset_stats);
5624         zfs_ioctl_register_dataset_read(ZFS_IOC_OBJSET_ZPLPROPS,
5625             zfs_ioc_objset_zplprops);
5626         zfs_ioctl_register_dataset_read(ZFS_IOC_DATASET_LIST_NEXT,
5627             zfs_ioc_dataset_list_next);
5628         zfs_ioctl_register_dataset_read(ZFS_IOC_SNAPSHOT_LIST_NEXT,
5629             zfs_ioc_snapshot_list_next);
5630         zfs_ioctl_register_dataset_read(ZFS_IOC_SEND_PROGRESS,
5631             zfs_ioc_send_progress);
5632
5633         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_DIFF,
5634             zfs_ioc_diff, zfs_secpolicy_diff);
5635         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_STATS,
5636             zfs_ioc_obj_to_stats, zfs_secpolicy_diff);
5637         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_OBJ_TO_PATH,
5638             zfs_ioc_obj_to_path, zfs_secpolicy_diff);
5639         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_ONE,
5640             zfs_ioc_userspace_one, zfs_secpolicy_userspace_one);
5641         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_USERSPACE_MANY,
5642             zfs_ioc_userspace_many, zfs_secpolicy_userspace_many);
5643         zfs_ioctl_register_dataset_read_secpolicy(ZFS_IOC_SEND,
5644             zfs_ioc_send, zfs_secpolicy_send);
5645
5646         zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_PROP, zfs_ioc_set_prop,
5647             zfs_secpolicy_none);
5648         zfs_ioctl_register_dataset_modify(ZFS_IOC_DESTROY, zfs_ioc_destroy,
5649             zfs_secpolicy_destroy);
5650         zfs_ioctl_register_dataset_modify(ZFS_IOC_RENAME, zfs_ioc_rename,
5651             zfs_secpolicy_rename);
5652         zfs_ioctl_register_dataset_modify(ZFS_IOC_RECV, zfs_ioc_recv,
5653             zfs_secpolicy_recv);
5654         zfs_ioctl_register_dataset_modify(ZFS_IOC_PROMOTE, zfs_ioc_promote,
5655             zfs_secpolicy_promote);
5656         zfs_ioctl_register_dataset_modify(ZFS_IOC_INHERIT_PROP,
5657             zfs_ioc_inherit_prop, zfs_secpolicy_inherit_prop);
5658         zfs_ioctl_register_dataset_modify(ZFS_IOC_SET_FSACL, zfs_ioc_set_fsacl,
5659             zfs_secpolicy_set_fsacl);
5660
5661         zfs_ioctl_register_dataset_nolog(ZFS_IOC_SHARE, zfs_ioc_share,
5662             zfs_secpolicy_share, POOL_CHECK_NONE);
5663         zfs_ioctl_register_dataset_nolog(ZFS_IOC_SMB_ACL, zfs_ioc_smb_acl,
5664             zfs_secpolicy_smb_acl, POOL_CHECK_NONE);
5665         zfs_ioctl_register_dataset_nolog(ZFS_IOC_USERSPACE_UPGRADE,
5666             zfs_ioc_userspace_upgrade, zfs_secpolicy_userspace_upgrade,
5667             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5668         zfs_ioctl_register_dataset_nolog(ZFS_IOC_TMP_SNAPSHOT,
5669             zfs_ioc_tmp_snapshot, zfs_secpolicy_tmp_snapshot,
5670             POOL_CHECK_SUSPENDED | POOL_CHECK_READONLY);
5671 }
```

```
5673 int
5674 pool_status_check(const char *name, zfs_ioc_namecheck_t type,
5675     zfs_ioc_poolcheck_t check)
5676 {
5677         spa_t *spa;
5678         int error;

5680         ASSERT(type == POOL_NAME || type == DATASET_NAME);

5682         if (check & POOL_CHECK_NONE)
5683                 return (0);

5685         error = spa_open(name, &spa, FTAG);
5686         if (error == 0) {
5687                 if ((check & POOL_CHECK_SUSPENDED) && spa_suspended(spa))
5688                         error = SET_ERROR(EAGAIN);
5689                 else if ((check & POOL_CHECK_READONLY) && !spa_writeable(spa))
5690                         error = SET_ERROR(EROFS);
5691                 spa_close(spa, FTAG);
5692         }
5693         return (error);
5694 }

5696 /*
5697  * Find a free minor number.
5698  */
5699 minor_t
5700 zfsdev_minor_alloc(void)
5701 {
5702         static minor_t last_minor;
5703         minor_t m;

5705         ASSERT(MUTEX_HELD(&zfsdev_state_lock));

5707         for (m = last_minor + 1; m != last_minor; m++) {
5708                 if (m > ZFSDEV_MAX_MINOR)
5709                         m = 1;
5710                 if (ddi_get_soft_state(zfsdev_state, m) == NULL) {
5711                         last_minor = m;
5712                         return (m);
5713                 }
5714         }

5716         return (0);
5717 }

5719 static int
5720 zfs_ctldev_init(dev_t *devp)
5721 {
5722         minor_t minor;
5723         zfs_soft_state_t *zs;

5725         ASSERT(MUTEX_HELD(&zfsdev_state_lock));
5726         ASSERT(getminor(*devp) == 0);

5728         minor = zfsdev_minor_alloc();
5729         if (minor == 0)
5730                 return (SET_ERROR(ENXIO));

5732         if (ddi_soft_state_zalloc(zfsdev_state, minor) != DDI_SUCCESS)
5733                 return (SET_ERROR(EAGAIN));

5735         *devp = makedevice(getemajor(*devp), minor);

5737         zs = ddi_get_soft_state(zfsdev_state, minor);
```



```
5738         zs->zss_type = ZSST_CTLDEV;
5739         zfs_onexit_init((zfs_onexit_t **)&zs->zss_data);

5741         return (0);
5742 }

5744 static void
5745 zfs_ctldev_destroy(zfs_onexit_t *zo, minor_t minor)
5746 {
5747         ASSERT(MUTEX_HELD(&zfsdev_state_lock));

5749         zfs_onexit_destroy(zo);
5750         ddi_soft_state_free(zfsdev_state, minor);
5751 }

5753 void *
5754 zfsdev_get_soft_state(minor_t minor, enum zfs_soft_state_type which)
5755 {
5756         zfs_soft_state_t *zp;

5758         zp = ddi_get_soft_state(zfsdev_state, minor);
5759         if (zp == NULL || zp->zss_type != which)
5760                 return (NULL);

5762         return (zp->zss_data);
5763 }

5765 static int
5766 zfsdev_open(dev_t *devp, int flag, int otyp, cred_t *cr)
5767 {
5768         int error = 0;

5770         if (getminor(*devp) != 0)
5771                 return (zvol_open(devp, flag, otyp, cr));

5773         /* This is the control device. Allocate a new minor if requested. */
5774         if (flag & FEXCL) {
5775                 mutex_enter(&zfsdev_state_lock);
5776                 error = zfs_ctldev_init(devp);
5777                 mutex_exit(&zfsdev_state_lock);
5778         }

5780         return (error);
5781 }

5783 static int
5784 zfsdev_close(dev_t dev, int flag, int otyp, cred_t *cr)
5785 {
5786         zfs_onexit_t *zo;
5787         minor_t minor = getminor(dev);

5789         if (minor == 0)
5790                 return (0);

5792         mutex_enter(&zfsdev_state_lock);
5793         zo = zfsdev_get_soft_state(minor, ZSST_CTLDEV);
5794         if (zo == NULL) {
5795                 mutex_exit(&zfsdev_state_lock);
5796                 return (zvol_close(dev, flag, otyp, cr));
5797         }
5798         zfs_ctldev_destroy(zo, minor);
5799         mutex_exit(&zfsdev_state_lock);

5801         return (0);
5802 }
```

```
5804 static int
5805 zfsdev_ioctl(dev_t dev, int cmd, intptr_t arg, int flag, cred_t *cr, int *rvalp)
5806 {
5807         zfs_cmd_t *zc;
5808         uint_t vecnum;
5809         int error, rc, len;
5810         minor_t minor = getminor(dev);
5811         const zfs_ioc_vec_t *vec;
5812         char *saved_poolname = NULL;
5813         nvlist_t *innvl = NULL;

5815         if (minor != 0 &&
5816             zfsdev_get_soft_state(minor, ZSST_CTLDEV) == NULL)
5817                 return (zvol_ioctl(dev, cmd, arg, flag, cr, rvalp));

5819         vecnum = cmd - ZFS_IOC_FIRST;
5820         ASSERT3U(getmajor(dev), ==, ddi_driver_major(zfs_dip));

5822         if (vecnum >= sizeof (zfs_ioc_vec) / sizeof (zfs_ioc_vec[0]))
5823                 return (SET_ERROR(EINVAL));
5824         vec = &zfs_ioc_vec[vecnum];

5826         zc = kmem_zalloc(sizeof (zfs_cmd_t), KM_SLEEP);

5828         error = ddi_copyin((void *)arg, zc, sizeof (zfs_cmd_t), flag);
5829         if (error != 0) {
5830                 error = SET_ERROR(EFAULT);
5831                 goto out;
5832         }

5834         zc->zc_iflags = flag & FKIOCTL;
5835         if (zc->zc_nvlist_src_size != 0) {
5836                 error = get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
5837                     zc->zc_iflags, &innvl);
5838                 if (error != 0)
5839                         goto out;
5840         }

5842         /*
5843          * Ensure that all pool/dataset names are valid before we pass down to
5844          * the lower layers.
5845          */
5846         zc->zc_name[sizeof (zc->zc_name) - 1] = '\0';
5847         switch (vec->zvec_namecheck) {
5848         case POOL_NAME:
5849                 if (pool_namecheck(zc->zc_name, NULL, NULL) != 0)
5850                         error = SET_ERROR(EINVAL);
5851                 else
5852                         error = pool_status_check(zc->zc_name,
5853                             vec->zvec_namecheck, vec->zvec_pool_check);
5854                 break;

5856         case DATASET_NAME:
5857                 if (dataset_namecheck(zc->zc_name, NULL, NULL) != 0)
5858                         error = SET_ERROR(EINVAL);
5859                 else
5860                         error = pool_status_check(zc->zc_name,
5861                             vec->zvec_namecheck, vec->zvec_pool_check);
5862                 break;

5864         case NO_NAME:
5865                 break;
5866         }


5869         if (error == 0 && !(flag & FKIOCTL))
```



```
5870                 error = vec->zvec_secpolicy(zc, innvl, cr);

5872         if (error != 0)
5873                 goto out;

5875         /* legacy ioctls can modify zc_name */
5876         len = strcspn(zc->zc_name, "/@#") + 1;
5877         saved_poolname = kmem_alloc(len, KM_SLEEP);
5878         (void) strlcpy(saved_poolname, zc->zc_name, len);

5880         if (vec->zvec_func != NULL) {
5881                 nvlist_t *outnvl;
5882                 int puterror = 0;
5883                 spa_t *spa;
5884                 nvlist_t *lognv = NULL;

5886                 ASSERT(vec->zvec_legacy_func == NULL);

5888                 /*
5889                  * Add the innvl to the lognv before calling the func,
5890                  * in case the func changes the innvl.
5891                  */
5892                 if (vec->zvec_allow_log) {
5893                         lognv = fnvlist_alloc();
5894                         fnvlist_add_string(lognv, ZPOOL_HIST_IOCTL,
5895                             vec->zvec_name);
5896                         if (!nvlist_empty(innvl)) {
5897                                 fnvlist_add_nvlist(lognv, ZPOOL_HIST_INPUT_NVL,
5898                                     innvl);
5899                         }
5900                 }

5902                 outnvl = fnvlist_alloc();
5903                 error = vec->zvec_func(zc->zc_name, innvl, outnvl);

5905                 if (error == 0 && vec->zvec_allow_log &&
5906                     spa_open(zc->zc_name, &spa, FTAG) == 0) {
5907                         if (!nvlist_empty(outnvl)) {
5908                                 fnvlist_add_nvlist(lognv, ZPOOL_HIST_OUTPUT_NVL,
5909                                     outnvl);
5910                         }
5911                         (void) spa_history_log_nvl(spa, lognv);
5912                         spa_close(spa, FTAG);
5913                 }
5914                 fnvlist_free(lognv);

5916                 if (!nvlist_empty(outnvl) || zc->zc_nvlist_dst_size != 0) {
5917                         int smusherror = 0;
5918                         if (vec->zvec_smush_outnvlist) {
5919                                 smusherror = nvlist_smush(outnvl,
5920                                     zc->zc_nvlist_dst_size);
5921                         }
5922                         if (smusherror == 0)
5923                                 puterror = put_nvlist(zc, outnvl);
5924                 }

5926                 if (puterror != 0)
5927                         error = puterror;

5929                 nvlist_free(outnvl);
5930         } else {
5931                 error = vec->zvec_legacy_func(zc);
5932         }

5934 out:
5935         nvlist_free(innvl);
```

```
5936         rc = ddi_copyout(zc, (void *)arg, sizeof (zfs_cmd_t), flag);
5937         if (error == 0 && rc != 0)
5938                 error = SET_ERROR(EFAULT);
5939         if (error == 0 && vec->zvec_allow_log) {
5940                 char *s = tsd_get(zfs_allow_log_key);
5941                 if (s != NULL)
5942                         strfree(s);
5943                 (void) tsd_set(zfs_allow_log_key, saved_poolname);
5944         } else {
5945                 if (saved_poolname != NULL)
5946                         strfree(saved_poolname);
5947         }

5949         kmem_free(zc, sizeof (zfs_cmd_t));
5950         return (error);
5951 }

5953 static int
5954 zfs_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
5955 {
5956         if (cmd != DDI_ATTACH)
5957                 return (DDI_FAILURE);

5959         if (ddi_create_minor_node(dip, "zfs", S_IFCHR, 0,
5960             DDI_PSEUDO, 0) == DDI_FAILURE)
5961                 return (DDI_FAILURE);

5963         zfs_dip = dip;

5965         ddi_report_dev(dip);

5967         return (DDI_SUCCESS);
5968 }

5970 static int
5971 zfs_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
5972 {
5973         if (spa_busy() || zfs_busy() || zvol_busy())
5974                 return (DDI_FAILURE);

5976         if (cmd != DDI_DETACH)
5977                 return (DDI_FAILURE);

5979         zfs_dip = NULL;

5981         ddi_prop_remove_all(dip);
5982         ddi_remove_minor_node(dip, NULL);

5984         return (DDI_SUCCESS);
5985 }

5987 /*ARGSUSED*/
5988 static int
5989 zfs_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
5990 {
5991         switch (infocmd) {
5992         case DDI_INFO_DEVT2DEVINFO:
5993                 *result = zfs_dip;
5994                 return (DDI_SUCCESS);

5996         case DDI_INFO_DEVT2INSTANCE:
5997                 *result = (void *)0;
5998                 return (DDI_SUCCESS);
5999         }

6001         return (DDI_FAILURE);
```



```
6002 }

6004 /*
6005  * OK, so this is a little weird.
6006  *
6007  * /dev/zfs is the control node, i.e. minor 0.
6008  * /dev/zvol/[r]dsk/pool/dataset are the zvols, minor > 0.
6009  *
6010  * /dev/zfs has basically nothing to do except serve up ioctls,
6011  * so most of the standard driver entry points are in zvol.c.
6012  */
6013 static struct cb_ops zfs_cb_ops = {
6014         zfsdev_open,    /* open */
6015         zfsdev_close,   /* close */
6016         zvol_strategy,  /* strategy */
6017         nodev,          /* print */
6018         zvol_dump,      /* dump */
6019         zvol_read,      /* read */
6020         zvol_write,     /* write */
6021         zfsdev_ioctl,   /* ioctl */
6022         nodev,          /* devmap */
6023         nodev,          /* mmap */
6024         nodev,          /* segmap */
6025         nochpoll,       /* poll */
6026         ddi_prop_op,    /* prop_op */
6027         NULL,           /* streamtab */
6028         D_NEW | D_MP | D_64BIT,         /* Driver compatibility flag */
6029         CB_REV,         /* version */
6030         nodev,          /* async read */
6031         nodev,          /* async write */
6032 };

6034 static struct dev_ops zfs_dev_ops = {
6035         DEVO_REV,       /* version */
6036         0,              /* refcnt */
6037         zfs_info,       /* info */
6038         nulldev,        /* identify */
6039         nulldev,        /* probe */
6040         zfs_attach,     /* attach */
6041         zfs_detach,     /* detach */
6042         nodev,          /* reset */
6043         &zfs_cb_ops,    /* driver operations */
6044         NULL,           /* no bus operations */
6045         NULL,           /* power */
6046         ddi_quiesce_not_needed, /* quiesce */
6047 };

6049 static struct modldrv zfs_modldrv = {
6050         &mod_driverops,
6051         "ZFS storage pool",
6052         &zfs_dev_ops
6053 };

6055 static struct modlinkage modlinkage = {
6056         MODREV_1,
6057         (void *)&zfs_modlfs,
6058         (void *)&zfs_modldrv,
6059         NULL
6060 };

6062 static void
6063 zfs_allow_log_destroy(void *arg)
6064 {
6065         char *poolname = arg;
6066         strfree(poolname);
6067 }
```

```
6069 int
6070 _init(void)
6071 {
6072         int error;

6074         spa_init(FREAD | FWRITE);
6075         zfs_init();
6076         zvol_init();
6077         zfs_ioctl_init();

6079         if ((error = mod_install(&modlinkage)) != 0) {
6080                 zvol_fini();
6081                 zfs_fini();
6082                 spa_fini();
6083                 return (error);
6084         }

6086         tsd_create(&zfs_fsyncer_key, NULL);
6087         tsd_create(&rrw_tsd_key, rrw_tsd_destroy);
6088         tsd_create(&zfs_allow_log_key, zfs_allow_log_destroy);

6090         error = ldi_ident_from_mod(&modlinkage, &zfs_li);
6091         ASSERT(error == 0);
6092         mutex_init(&zfs_share_lock, NULL, MUTEX_DEFAULT, NULL);

6094         return (0);
6095 }

6097 int
6098 _fini(void)
6099 {
6100         int error;

6102         if (spa_busy() || zfs_busy() || zvol_busy() || zio_injection_enabled)
6103                 return (SET_ERROR(EBUSY));

6105         if ((error = mod_remove(&modlinkage)) != 0)
6106                 return (error);

6108         zvol_fini();
6109         zfs_fini();
6110         spa_fini();
6111         if (zfs_nfsshare_inited)
6112                 (void) ddi_modclose(nfs_mod);
6113         if (zfs_smbshare_inited)
6114                 (void) ddi_modclose(smbsrv_mod);
6115         if (zfs_nfsshare_inited || zfs_smbshare_inited)
6116                 (void) ddi_modclose(sharefs_mod);

6118         tsd_destroy(&zfs_fsyncer_key);
6119         ldi_ident_release(zfs_li);
6120         zfs_li = NULL;
6121         mutex_destroy(&zfs_share_lock);

6123         return (error);
6124 }

6126 int
6127 _info(struct modinfo *modinfop)
6128 {
6129         return (mod_info(&modlinkage, modinfop));
6130 }
```