```
**********************************************************
   35728 Mon Feb 15 12:55:56 2016
new/usr/src/cmd/beadm/beadm.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 735 static int
 736 be_do_create(int argc, char **argv)
 737 {
 738         nvlist_t        *be_attrs;
 739         nvlist_t        *zfs_props = NULL;
 740         boolean_t       activate = B_FALSE;
 741         boolean_t       is_snap = B_FALSE;
 742         int             c;
 743         int             err = 1;
 744         char            *obe_name = NULL;
 745         char            *snap_name = NULL;
 746         char            *nbe_zpool = NULL;
 747         char            *nbe_name = NULL;
 748         char            *nbe_desc = NULL;
 749         char            *propname = NULL;
 750         char            *propval = NULL;
 751         char            *strval = NULL;

 753         while ((c = getopt(argc, argv, "ad:e:io:p:v")) != -1) {
 754                 switch (c) {
 755                 case 'a':
 756                         activate = B_TRUE;
 757                         break;
 758                 case 'd':
 759                         nbe_desc = optarg;
 760                         break;
 761                 case 'e':
 762                         obe_name = optarg;
 763                         break;
 764                 case 'o':
 765                         if (zfs_props == NULL && be_nvl_alloc(&zfs_props) != 0)
 766                                 return (1);

 768                         propname = optarg;
 769                         if ((propval = strchr(propname, '=')) == NULL) {
 770                                 (void) fprintf(stderr, _("missing "
 771                                     "'=' for -o option\n"));
 772                                 goto out2;
 773                         }
 774                         *propval = '\0';
 775                         propval++;
 776                         if (nvlist_lookup_string(zfs_props, propname,
 777                             &strval) == 0) {
 778                                 (void) fprintf(stderr, _("property '%s' "
 779                                     "specified multiple times\n"), propname);
 780                                 goto out2;

 782                         }
 783                         if (be_nvl_add_string(zfs_props, propname, propval)
 784                             != 0)
 785                                 goto out2;

 787                         break;
 788                 case 'p':
 789                         nbe_zpool = optarg;
 790                         break;
 791                 case 'v':
 792                         libbe_print_errors(B_TRUE);
 793                         break;
```



```
 794                 default:
 795                         usage();
 796                         goto out2;
 797                 }
 798         }

 800         argc -= optind;
 801         argv += optind;

 803         if (argc != 1) {
 804                 usage();
 805                 goto out2;
 806         }

 808         nbe_name = argv[0];

 810         if ((snap_name = strrchr(nbe_name, '@')) != NULL) {
 811                 if (snap_name[1] == '\0') {
 812                         usage();
 813                         goto out2;
 814                 }

 816                 snap_name[0] = '\0';
 817                 snap_name++;
 818                 is_snap = B_TRUE;
 819         }

 821         if (obe_name) {
 822                 if (is_snap) {
 823                         usage();
 824                         goto out2;
 825                 }

 827                 /*
 828                  * Check if obe_name is really a snapshot name.
 829                  * If so, split it out.
 830                  */
 831                 if ((snap_name = strrchr(obe_name, '@')) != NULL) {
 832                         if (snap_name[1] == '\0') {
 833                                 usage();
 834                                 goto out2;
 835                         }

 837                         snap_name[0] = '\0';
 838                         snap_name++;
 839                 }
 840         } else if (is_snap) {
 841                 obe_name = nbe_name;
 842                 nbe_name = NULL;
 843         }

 845         if (be_nvl_alloc(&be_attrs) != 0)
 846                 goto out2;


 849         if (zfs_props != NULL && be_nvl_add_nvlist(be_attrs,
 850             BE_ATTR_ORIG_BE_NAME, zfs_props) != 0)
 851                 goto out;

 853         if (obe_name != NULL && be_nvl_add_string(be_attrs,
 854             BE_ATTR_ORIG_BE_NAME, obe_name) != 0)
 855                 goto out;

 857         if (snap_name != NULL && be_nvl_add_string(be_attrs,
 858             BE_ATTR_SNAP_NAME, snap_name) != 0)
 859                 goto out;
```

```
861         if (nbe_zpool != NULL && be_nvl_add_string(be_attrs,
862             BE_ATTR_NEW_BE_POOL, nbe_zpool) != 0)
863                 goto out;
864
865         if (nbe_name != NULL && be_nvl_add_string(be_attrs,
866             BE_ATTR_NEW_BE_NAME, nbe_name) != 0)
867                 goto out;
868
869         if (nbe_desc != NULL && be_nvl_add_string(be_attrs,
870             BE_ATTR_NEW_BE_DESC, nbe_desc) != 0)
871                 goto out;
872
873         if (is_snap)
874                 err = be_create_snapshot(be_attrs);
875         else
876                 err = be_copy(be_attrs);
877
878         switch (err) {
879         case BE_SUCCESS:
880                 if (!is_snap && !nbe_name) {
881                         /*
882                          * We requested an auto named BE; find out the
883                          * name of the BE that was created for us and
884                          * the auto snapshot created from the original BE.
885                          */
886                         if (nvlist_lookup_string(be_attrs, BE_ATTR_NEW_BE_NAME,
887                             &nbe_name) != 0) {
888                                 (void) fprintf(stderr, _("failed to get %s "
889                                     "attribute\n"), BE_ATTR_NEW_BE_NAME);
890                                 break;
891                         } else
892                                 (void) printf(_("Auto named BE: %s\n"),
893                                     nbe_name);
894
895                         if (nvlist_lookup_string(be_attrs, BE_ATTR_SNAP_NAME,
896                             &snap_name) != 0) {
897                                 (void) fprintf(stderr, _("failed to get %s "
898                                     "attribute\n"), BE_ATTR_SNAP_NAME);
899                                 break;
900                         } else
901                                 (void) printf(_("Auto named snapshot: %s\n"),
902                                     snap_name);
903                 }
904
905                 if (!is_snap && activate) {
906                         char *args[] = { "activate", "", NULL };
907                         args[1] = nbe_name;
908                         optind = 1;
909
910                         err = be_do_activate(2, args);
911                         goto out;
912                 }
913
914                 (void) printf(_("Created successfully\n"));
915                 break;
916         case BE_ERR_BE_EXISTS:
917                 (void) fprintf(stderr, _("BE %s already exists\n."
918                     "Please choose a different BE name.\n"), nbe_name);
919                 break;
920         case BE_ERR_SS_EXISTS:
921                 (void) fprintf(stderr, _("BE %s snapshot %s already exists.\n"
922                     "Please choose a different snapshot name.\n"), obe_name,
923                     snap_name);
924                 break;
925         case BE_ERR_PERM:
```



```
926         case BE_ERR_ACCESS:
927                 if (is_snap)
928                         (void) fprintf(stderr, _("Unable to create snapshot "
929                             "%s.\n"), snap_name);
930                 else
931                         (void) fprintf(stderr, _("Unable to create %s.\n"),
932                             nbe_name);
933                 (void) fprintf(stderr, _("You have insufficient privileges to "
934                     "execute this command.\n"));
935                 break;
936         default:
937                 if (is_snap)
938                         (void) fprintf(stderr, _("Unable to create snapshot "
939                             "%s.\n"), snap_name);
940                 else
941                         (void) fprintf(stderr, _("Unable to create %s.\n"),
942                             nbe_name);
943                 (void) fprintf(stderr, "%s\n", be_err_to_str(err));
944         }
945
946 out:
947         nvlist_free(be_attrs);
948 out2:
949         if (zfs_props != NULL)
949         nvlist_free(zfs_props);
950
951         return (err);
952 }
```

_____***unchanged_portion_omitted_***

****
234065 Mon Feb 15 12:55:56 2016
new/usr/src/cmd/boot/bootadm/bootadm.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
3030 #define init_walk_args()        bzero(&walk_arg, sizeof (walk_arg))

3032 static void
3033 clear_walk_args(void)
3034 {
3035         if (walk_arg.old_nvlp)
3035         nvlist_free(walk_arg.old_nvlp);
3037         if (walk_arg.new_nvlp)
3036         nvlist_free(walk_arg.new_nvlp);
3037         if (walk_arg.sparcfile)
3038                 (void) fclose(walk_arg.sparcfile);
3039         walk_arg.old_nvlp = NULL;
3040         walk_arg.new_nvlp = NULL;
3041         walk_arg.sparcfile = NULL;
3042 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    7876 Mon Feb 15 12:55:57 2016
new/usr/src/cmd/cmd-inet/sbin/dhcpagent/defaults.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 146 /*
 147  * df_get_string(): gets the string value of a given user-tunable parameter
 148  *
 149  *   input: const char *: the interface the parameter applies to
 150  *          boolean_t: B_TRUE for DHCPv6, B_FALSE for IPv4 DHCP
 151  *          uint_t: the parameter number to look up
 152  *  output: const char *: the parameter's value, or default if not set
 153  *                        (must be copied by caller to be kept)
 154  *    NOTE: df_get_string() is both used by functions outside this source
 155  *          file to retrieve strings from the defaults file, *and*
 156  *          internally by other df_get_*() functions.
 157  */

 159 const char *
 160 df_get_string(const char *if_name, boolean_t isv6, uint_t param)
 161 {
 162         char                    *value;
 163         char                    paramstr[256];
 164         char                    name[256];
 165         struct stat             statbuf;
 166         static struct stat      df_statbuf;
 167         static boolean_t        df_unavail_msg = B_FALSE;
 168         static nvlist_t         *df_nvlist = NULL;

 170         if (param >= (sizeof (defaults) / sizeof (*defaults)))
 171                 return (NULL);

 173         if (stat(DHCP_AGENT_DEFAULTS, &statbuf) != 0) {
 174                 if (!df_unavail_msg) {
 175                         dhcpmsg(MSG_WARNING, "cannot access %s; using "
 176                             "built-in defaults", DHCP_AGENT_DEFAULTS);
 177                         df_unavail_msg = B_TRUE;
 178                 }
 179                 return (defaults[param].df_default);
 180         }

 182         /*
 183          * if our cached parameters are stale, rebuild.
 184          */

 186         if (statbuf.st_mtime != df_statbuf.st_mtime ||
 187             statbuf.st_size != df_statbuf.st_size) {
 188                 df_statbuf = statbuf;
 189                 if (df_nvlist != NULL)
 189                 nvlist_free(df_nvlist);
 190                 df_nvlist = df_build_cache();
 191         }

 193         if (isv6) {
 194                 (void) snprintf(name, sizeof (name), ".V6.%s",
 195                     defaults[param].df_name);
 196                 (void) snprintf(paramstr, sizeof (paramstr), "%s%s", if_name,
 197                     name);
 198         } else {
 199                 (void) strlcpy(name, defaults[param].df_name, sizeof (name));
 200                 (void) snprintf(paramstr, sizeof (paramstr), "%s.%s", if_name,
 201                     name);
 202         }
```



```
 204         /*
 205          * first look for 'if_name.[v6.]param', then '[v6.]param'.  if neither
 206          * has been set, use the built-in default.
 207          */

 209         if (nvlist_lookup_string(df_nvlist, paramstr, &value) == 0 ||
 210             nvlist_lookup_string(df_nvlist, name, &value) == 0)
 211                 return (value);

 213         return (defaults[param].df_default);
 214 }
______unchanged_portion_omitted_
```

****
124126 Mon Feb 15 12:55:57 2016
new/usr/src/cmd/cmd-inet/usr.lib/vrrpd/vrrpd.c
patch tsoome-feedback
****
_____***unchanged_portion_omitted_***

```
4436 static int
4437 vrrpd_post_event(const char *name, vrrp_state_t prev_st, vrrp_state_t st)
4438 {
4439         sysevent_id_t   eid;
4440         nvlist_t        *nvl = NULL;

4442         /*
4443          * sysevent is not supported in the non-global zone
4444          */
4445         if (getzoneid() != GLOBAL_ZONEID)
4446                 return (0);

4448         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0)
4449                 goto failed;

4451         if (nvlist_add_uint8(nvl, VRRP_EVENT_VERSION,
4452             VRRP_EVENT_CUR_VERSION) != 0)
4453                 goto failed;

4455         if (nvlist_add_string(nvl, VRRP_EVENT_ROUTER_NAME, name) != 0)
4456                 goto failed;

4458         if (nvlist_add_uint8(nvl, VRRP_EVENT_STATE, st) != 0)
4459                 goto failed;

4461         if (nvlist_add_uint8(nvl, VRRP_EVENT_PREV_STATE, prev_st) != 0)
4462                 goto failed;

4464         if (sysevent_post_event(EC_VRRP, ESC_VRRP_STATE_CHANGE,
4465             SUNW_VENDOR, VRRP_EVENT_PUBLISHER, nvl, &eid) == 0) {
4466                 nvlist_free(nvl);
4467                 return (0);
4468         }

4470 failed:
4471         vrrp_log(VRRP_ERR, "vrrpd_post_event(): ‘state change (%s --> %s)’ "
4472             "sysevent posting failed: %s", vrrp_state2str(prev_st),
4473             vrrp_state2str(st), strerror(errno));

4475         if (nvl != NULL)
4475         nvlist_free(nvl);
4476         return (-1);
4477 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  227576 Mon Feb 15 12:55:57 2016
new/usr/src/cmd/cmd-inet/usr.sbin/ipqosconf/ipqosconf.c
patch tsoome-feedback
patch cleanup
6659 nvlist_free(NULL) is a no-op
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License, Version 1.0 only
   6  * (the "License").  You may not use this file except in compliance
   7  * with the License.
   8  *
   9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  10  * or http://www.opensolaris.org/os/licensing.
  11  * See the License for the specific language governing permissions
  12  * and limitations under the License.
  13  *
  14  * When distributing Covered Code, include this CDDL HEADER in each
  15  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  16  * If applicable, add the following below this CDDL HEADER, with the
  17  * fields enclosed by brackets "[]" replaced with your own identifying
  18  * information: Portions Copyright [yyyy] [name of copyright owner]
  19  *
  20  * CDDL HEADER END
  21  */
  22 /*
  23  * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
  24  * Use is subject to license terms.
  25  */

  27 #pragma ident   "%Z%%M% %I%     %E% SMI"

  27 /* enable debug output and some debug asserts */
  28 #undef  _IPQOS_CONF_DEBUG

  30 #include <stdlib.h>
  31 #include <unistd.h>
  32 #include <libintl.h>
  33 #include <signal.h>
  34 #include <strings.h>
  35 #include <sys/nvpair.h>
  36 #include <stdio.h>
  37 #include <netinet/in.h>
  38 #include <arpa/inet.h>
  39 #include <ctype.h>
  40 #include <sys/socket.h>
  41 #include <limits.h>
  42 #include <netdb.h>
  43 #include <fcntl.h>
  44 #include <sys/types.h>
  45 #include <sys/stat.h>
  46 #include <errno.h>
  47 #include <libipp.h>
  48 #include <ipp/ipp_config.h>
  49 #include <ipp/ipgpc/ipgpc.h>
  50 #include <ipp/ipp.h>
  51 #ifdef  _IPQOS_CONF_DEBUG
  52 #include <assert.h>
  53 #endif
  54 #include <sys/sockio.h>
  55 #include <syslog.h>
  56 #include <stdarg.h>
  57 #include <libintl.h>
```



```
  58 #include <locale.h>
  59 #include <pwd.h>
  60 #include "ipqosconf.h"

  62 #if     defined(_IPQOS_CONF_DEBUG)

  64 /* debug level */
  65 static int ipqosconf_dbg_flgs =
  66 /*
  67  */
  68 RBK |
  69 MHME |
  70 KRET |
  71 DIFF |
  72 APPLY |
  73 L2 |
  74 L1 |
  75 L0 |
  76 0;



  80 #define IPQOSCDBG0(lvl, x)\
  81         if (lvl & ipqosconf_dbg_flgs)\
  82                 (void) fprintf(stderr, x)

  84 #define IPQOSCDBG1(lvl, x, y)\
  85         if (lvl & ipqosconf_dbg_flgs)\
  86                 (void) fprintf(stderr, x, y)

  88 #define IPQOSCDBG2(lvl, x, y, z)\
  89         if (lvl & ipqosconf_dbg_flgs)\
  90                 (void) fprintf(stderr, x, y, z)

  92 #define IPQOSCDBG3(lvl, x, y, z, a)\
  93         if (lvl & ipqosconf_dbg_flgs)\
  94                 (void) fprintf(stderr, x, y, z, a)

  96 #define IPQOSCDBG4(lvl, x, y, z, a, b)\
  97         if (lvl & ipqosconf_dbg_flgs)\
  98                 (void) fprintf(stderr, x, y, z, a, b)

 100 #define IPQOSCDBG5(lvl, x, y, z, a, b, c)\
 101         if (lvl & ipqosconf_dbg_flgs)\
 102                 (void) fprintf(stderr, x, y, z, a, b, c)

 104 #else   /* defined(_IPQOS_CONF_DEBUG) && !defined(lint) */

 106 #define IPQOSCDBG0(lvl, x)
 107 #define IPQOSCDBG1(lvl, x, y)
 108 #define IPQOSCDBG2(lvl, x, y, z)
 109 #define IPQOSCDBG3(lvl, x, y, z, a)
 110 #define IPQOSCDBG4(lvl, x, y, z, a, b)
 111 #define IPQOSCDBG5(lvl, x, y, z, a, b, c)

 113 #endif  /* defined(_IPQOS_CONF_DEBUG) */



 117 /* function prototypes */

 119 static int modify_params(char *, nvlist_t **, int, boolean_t);
 120 static int add_class(char *, char *, int, boolean_t, char *);
 121 static int modify_class(char *, char *, int, boolean_t, char *,
 122     enum ipp_flags);
 123 static int remove_class(char *, char *, int, enum ipp_flags);
```

```
124 static int add_filter(char *, ipqos_conf_filter_t *, int);
125 static int modify_filter(char *, ipqos_conf_filter_t *, int);
126 static int remove_filter(char *, char *, int, int);
127 static boolean_t arrays_equal(int *, int *, uint32_t);
128 static int diffclass(ipqos_conf_class_t *, ipqos_conf_class_t *);
129 static int diffparams(ipqos_conf_params_t *, ipqos_conf_params_t *, char *);
130 static int difffilter(ipqos_conf_filter_t *, ipqos_conf_filter_t *, char *);
131 static int add_filters(ipqos_conf_filter_t *, char *, int, boolean_t);
132 static int add_classes(ipqos_conf_class_t *, char *,  int, boolean_t);
133 static int modify_items(ipqos_conf_action_t *);
134 static int add_items(ipqos_conf_action_t *, boolean_t);
135 static int add_item(ipqos_conf_action_t *, boolean_t);
136 static int remove_items(ipqos_conf_action_t *, boolean_t);
137 static int remove_item(ipqos_conf_action_t *, boolean_t);
138 static int undo_modifys(ipqos_conf_action_t *, ipqos_conf_action_t *);
139 static int applydiff(ipqos_conf_action_t *, ipqos_conf_action_t *);
140 static int rollback(ipqos_conf_action_t *, ipqos_conf_action_t *);
141 static int rollback_recover(ipqos_conf_action_t *);
142 static ipqos_conf_class_t *classexist(char *, ipqos_conf_class_t *);
143 static ipqos_conf_filter_t *filterexist(char *, int, ipqos_conf_filter_t *);
144 static ipqos_conf_action_t *actionexist(char *, ipqos_conf_action_t *);
145 static int diffnvlists(nvlist_t *, nvlist_t *, char *, int *, place_t);
146 static int diffaction(ipqos_conf_action_t *, ipqos_conf_action_t *);
147 static int diffconf(ipqos_conf_action_t *, ipqos_conf_action_t *);
148 static int readllong(char *, long long *, char **);
149 static int readuint8(char *, uint8_t *, char **);
150 static int readuint16(char *, uint16_t *, char **);
151 static int readint16(char *, int16_t *, char **);
152 static int readint32(char *, int *, char **);
153 static int readuint32(char *, uint32_t *, char **);
154 static int readbool(char *, boolean_t *);
155 static void setmask(int, in6_addr_t *, int);
156 static int readtoken(FILE *, char **);
157 static nvpair_t *find_nvpair(nvlist_t *, char *);
158 static char *prepend_module_name(char *, char *);
159 static int readnvpair(FILE *, FILE *, nvlist_t **, nvpair_t **,
160     ipqos_nvtype_t *, place_t, char *);
161 static int add_aref(ipqos_conf_act_ref_t **, char *, char *);
162 static int readparams(FILE *, FILE *, char *, ipqos_conf_params_t *);
163 static int readclass(FILE *, char *, ipqos_conf_class_t **, char **, int);
164 static int readfilter(FILE *, FILE *, char *, ipqos_conf_filter_t **, char **,
165     int);
166 static FILE *validmod(char *, int *);
167 static int readaction(FILE *, ipqos_conf_action_t **);
168 static int actions_unique(ipqos_conf_action_t *, char **);
169 static int validconf(ipqos_conf_action_t *, int);
170 static int readconf(FILE *, ipqos_conf_action_t **);
171 static int flush(boolean_t *);
172 static int atomic_flush(boolean_t);
173 static int flushconf();
174 static int writeconf(ipqos_conf_action_t *, char *);
175 static int commitconf();
176 static int applyconf(char *ifile);
177 static int block_all_signals();
178 static int restore_all_signals();
179 static int unlock(int fd);
180 static int lock();
181 static int viewconf(int);
182 static void usage();
183 static int valid_name(char *);
184 static int in_cycle(ipqos_conf_action_t *);
185 static int readtype(FILE *, char *, char *, ipqos_nvtype_t *, str_val_nd_t **,
186     char *, boolean_t, place_t *);
187 static int read_int_array_info(char *, str_val_nd_t **, uint32_t *, int *,
188     int *, char *);
189 static str_val_nd_t *read_enum_nvs(char *, char *);
```



```
190 static int add_str_val_entry(str_val_nd_t **, char *, uint32_t);
191 static void free_str_val_entrys(str_val_nd_t *);
192 static void get_str_val_value_range(str_val_nd_t *, int *, int *);
193 static int read_enum_value(FILE *, char *, str_val_nd_t *, uint32_t *);
194 static int read_mapped_values(FILE *, nvlist_t **, char *, char *,
195     int);
196 static int read_int_array(FILE *, char *, int **, uint32_t, int, int,
197     str_val_nd_t *);
198 static int str_val_list_lookup(str_val_nd_t *, char *, uint32_t *);
199 static int parse_kparams(char *, ipqos_conf_params_t *, nvlist_t *);
200 static int parse_kclass(ipqos_conf_class_t *, nvlist_t *);
201 static int parse_kfilter(ipqos_conf_filter_t *, nvlist_t *);
202 static int parse_kaction(nvlist_t *, ipqos_actinfo_prm_t *);
203 static int readkconf(ipqos_conf_action_t **);
204 static void print_int_array(FILE *, int *, uint32_t, int, int, str_val_nd_t *,
205     int);
206 static void printrange(FILE *fp, uint32_t, uint32_t);
207 static void printenum(FILE *, uint32_t, str_val_nd_t *);
208 static void printproto(FILE *, uint8_t);
209 static void printport(FILE *, uint16_t);
210 static int printnvlist(FILE *, char *, nvlist_t *, int, ipqos_conf_filter_t *,
211     int, place_t);
212 static int virtual_action(char *);
213 static void free_arefs(ipqos_conf_act_ref_t *);
214 static void print_action_nm(FILE *, char *);
215 static int add_orig_ipqosconf(nvlist_t *);
216 static char *get_originator_nm(uint32_t);
217 static void mark_classes_filters_new(ipqos_conf_action_t *);
218 static void mark_classes_filters_del(ipqos_conf_action_t *);
219 static void mark_config_new(ipqos_conf_action_t *);
220 static int printifname(FILE *, int);
221 static int readifindex(char *, int *);
222 static void cleanup_string_table(char **, int);
223 static int domultihome(ipqos_conf_filter_t *, ipqos_conf_filter_t **,
224     boolean_t);
225 static int dup_filter(ipqos_conf_filter_t *, ipqos_conf_filter_t **, int, int,
226     void *, void *, int);
227 static void free_actions(ipqos_conf_action_t *);
228 static ipqos_conf_filter_t *alloc_filter();
229 static void free_filter(ipqos_conf_filter_t *);
230 static int read_curl_begin(FILE *);
231 static ipqos_conf_class_t *alloc_class(void);
232 static int diffclasses(ipqos_conf_action_t *old, ipqos_conf_action_t *new);
233 static int difffilters(ipqos_conf_action_t *old, ipqos_conf_action_t *new);
234 static int dup_class(ipqos_conf_class_t *src, ipqos_conf_class_t **dst);
235 static int add_action(ipqos_conf_action_t *act);
236 static int masktocidr(int af, in6_addr_t *mask);
237 static int read_perm_items(int, FILE *, char *, char ***, int *);
238 static int in_string_table(char *stable[], int size, char *string);
239 static void list_end(ipqos_list_el_t **listp, ipqos_list_el_t ***lendpp);
240 static void add_to_list(ipqos_list_el_t **listp, ipqos_list_el_t *el);
241 static int read_cfile_ver(FILE *, char *);
242 static char *quote_ws_string(const char *);
243 static int read_tfile_ver(FILE *, char *, char *);
244 static int ver_str_to_int(char *);
245 static void printuser(FILE *fp, uid_t uid);
246 static int readuser(char *str, uid_t *uid);

248 /*
249  * macros to call list functions with the more complex list element type
250  * cast to the skeletal type iqpos_list_el_t.
251  */
252 #define LIST_END(list, end)\
253         list_end((ipqos_list_el_t **)list,  (ipqos_list_el_t ***)end)
254 #define ADD_TO_LIST(list, el)\
255         add_to_list((ipqos_list_el_t **)list, (ipqos_list_el_t *)el)
```

```
257 /*
258  *      Macros to produce a quoted string containing the value of a
259  *      preprocessor macro. For example, if SIZE is defined to be 256,
260  *      VAL2STR(SIZE) is "256". This is used to construct format
261  *      strings for scanf-family functions below.
262  */
263 #define QUOTE(x)        #x
264 #define VAL2STR(x)      QUOTE(x)


267 /* globals */

269 /* table of supported parameter types and enum value */
270 static str_val_t nv_types[] = {
271 {"uint8",               IPQOS_DATA_TYPE_UINT8},
272 {"int16",               IPQOS_DATA_TYPE_INT16},
273 {"uint16",              IPQOS_DATA_TYPE_UINT16},
274 {"int32",               IPQOS_DATA_TYPE_INT32},
275 {"uint32",              IPQOS_DATA_TYPE_UINT32},
276 {"boolean",             IPQOS_DATA_TYPE_BOOLEAN},
277 {"string",              IPQOS_DATA_TYPE_STRING},
278 {"action",              IPQOS_DATA_TYPE_ACTION},
279 {"address",             IPQOS_DATA_TYPE_ADDRESS},
280 {"port",                IPQOS_DATA_TYPE_PORT},
281 {"protocol",            IPQOS_DATA_TYPE_PROTO},
282 {"enum",                IPQOS_DATA_TYPE_ENUM},
283 {"ifname",              IPQOS_DATA_TYPE_IFNAME},
284 {"mindex",              IPQOS_DATA_TYPE_M_INDEX},
285 {"int_array",           IPQOS_DATA_TYPE_INT_ARRAY},
286 {"user",                IPQOS_DATA_TYPE_USER},
287 {"",                    0}
288 };
```

*unchanged_portion_omitted_*

```
531 /*
532  * add a class to the kernel action action_name called class_name with
533  * stats set according to stats_enable and the first action set to
534  * first_action.
535  * RETURNS: IPQOS_CONF_ERR on error, else IPQOS_CONF_SUCCES.
536  */
537 static int
538 add_class(
539 char *action_name,
540 char *class_name,
541 int module_version,
542 boolean_t stats_enable,
543 char *first_action)
544 {

546         nvlist_t *nvl;

548         IPQOSCDBG4(APPLY, "add_class: action: %s, class: %s, "
549             "first_action: %s, stats: %s\n", action_name, class_name,
550             first_action, (stats_enable == B_TRUE ? "true" : "false"));


553         /* create nvlist */
554         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
555                 ipqos_msg(MT_ENOSTR, "nvlist_alloc");
556                 return (IPQOS_CONF_ERR);
557         }

559         /* add 'add class' config type */
560         if (nvlist_add_byte(nvl, IPP_CONFIG_TYPE, CLASSIFIER_ADD_CLASS) != 0) {
561                 ipqos_msg(MT_ENOSTR, "nvlist_add_byte");
```



```
562                 goto fail;
563         }

565         /*
566          * add module version
567          */
568         if (nvlist_add_uint32(nvl, IPP_MODULE_VERSION,
569             (uint32_t)module_version) != 0) {
570                 ipqos_msg(MT_ENOSTR, "nvlist_add_uint32");
571                 goto fail;
572         }

574         /* add class name */
575         if (nvlist_add_string(nvl, CLASSIFIER_CLASS_NAME, class_name) != 0) {
576                 ipqos_msg(MT_ENOSTR, "nvlist_add_string");
577                 goto fail;
578         }

580         /* add next action */
581         if (nvlist_add_string(nvl, CLASSIFIER_NEXT_ACTION, first_action) != 0) {
582                 ipqos_msg(MT_ENOSTR, "nvlist_add_string");
583                 goto fail;
584         }

586         /* add stats_enable */
587         if (nvlist_add_uint32(nvl, CLASSIFIER_CLASS_STATS_ENABLE,
588             (uint32_t)stats_enable) != 0) {
589                 ipqos_msg(MT_ENOSTR, "nvlist_add_uint32");
590                 goto fail;
591         }

593         /* add ipqosconf as originator */
594         if (add_orig_ipqosconf(nvl) != IPQOS_CONF_SUCCESS) {
595                 goto fail;
596         }

598         /* call lib to do modify */
599         if (ipp_action_modify(action_name, &nvl, 0) != 0) {

601                 /* ipgpc max classes */

603                 if (errno == ENOSPC &&
604                     strcmp(action_name, IPGPC_CLASSIFY) == 0) {
605                         ipqos_msg(MT_ERROR,
606                             gettext("Max number of classes reached in %s.\n"),
607                             IPGPC_NAME);

609                 /* other errors */

611                 } else {
612                         ipqos_msg(MT_ERROR,
613                             gettext("Failed to create class %s in action "
614                             "%s: %s.\n"), class_name, action_name,
615                             strerror(errno));
616                 }

618                 goto fail;
619         }

621         return (IPQOS_CONF_SUCCESS);
622 fail:
625         if (nvl != NULL)
623         nvlist_free(nvl);
624         return (IPQOS_CONF_ERR);
625 }
```

```
628 /*
629  * modify the class in the kernel action action_name called class_name with
630  * stats set according to stats_enable and the first action set to
631  * first_action.
632  * RETURNS: IPQOS_CONF_ERR on error, else IPQOS_CONF_SUCCES.
633  */
634 static int
635 modify_class(
636 char *action_name,
637 char *class_name,
638 int module_version,
639 boolean_t stats_enable,
640 char *first_action,
641 enum ipp_flags flags)
642 {

644         nvlist_t *nvl;

646         IPQOSCDBG5(APPLY, "modify_class: action: %s, class: %s, first: %s, "
647             "stats: %s, flags: %x\n", action_name, class_name, first_action,
648             stats_enable == B_TRUE ? "true" : "false", flags);


651         /* create nvlist */
652         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
653                 ipqos_msg(MT_ENOSTR, "nvlist_alloc");
654                 return (IPQOS_CONF_ERR);
655         }

657         /* add 'modify class' config type */
658         if (nvlist_add_byte(nvl, IPP_CONFIG_TYPE, CLASSIFIER_MODIFY_CLASS) !=
659             0) {
660                 ipqos_msg(MT_ENOSTR, "nvlist_add_byte");
661                 goto fail;
662         }

664         /*
665          * add module version
666          */
667         if (nvlist_add_uint32(nvl, IPP_MODULE_VERSION,
668             (uint32_t)module_version) != 0) {
669                 ipqos_msg(MT_ENOSTR, "nvlist_add_uint32");
670                 goto fail;
671         }

673         /* add class name */
674         if (nvlist_add_string(nvl, CLASSIFIER_CLASS_NAME, class_name) != 0) {
675                 ipqos_msg(MT_ENOSTR, "nvlist_add_string");
676                 goto fail;
677         }

679         /* add next action */
680         if (nvlist_add_string(nvl, CLASSIFIER_NEXT_ACTION, first_action) != 0) {
681                 ipqos_msg(MT_ENOSTR, "nvlist_add_string");
682                 goto fail;
683         }

685         /* add stats enable */
686         if (nvlist_add_uint32(nvl, CLASSIFIER_CLASS_STATS_ENABLE,
687             (uint32_t)stats_enable) != 0) {
688                 ipqos_msg(MT_ENOSTR, "nvlist_add_uint32");
689                 goto fail;
690         }

692         /* add originator ipqosconf */
```



```
693         if (add_orig_ipqosconf(nvl) != IPQOS_CONF_SUCCESS) {
694                 goto fail;
695         }

697         /* call lib to do modify */
698         if (ipp_action_modify(action_name, &nvl, flags) != 0) {

700                 /* generic error message */

702                 ipqos_msg(MT_ERROR,
703                     gettext("Modifying class %s in action %s failed: %s.\n"),
704                     class_name, action_name, strerror(errno));

706                 goto fail;
707         }

709         return (IPQOS_CONF_SUCCESS);
710 fail:
714         if (nvl != NULL)
711         nvlist_free(nvl);
712         return (IPQOS_CONF_ERR);
713 }

715 /*
716  * removes the class class_name from the kernel action action_name. The
717  * flags argument can currently be set to IPP_ACTION_DESTROY which will
718  * result in the action this class references being destroyed.
719  * RETURNS: IPQOS_CONF_ERR on error, else IPQOS_CONF_SUCCES.
720  */
721 static int
722 remove_class(
723 char *action_name,
724 char *class_name,
725 int module_version,
726 enum ipp_flags flags)
727 {

729         nvlist_t *nvl;

731         IPQOSCDBG3(APPLY, "remove_class: action: %s, class: %s, "
732             "flags: %x\n", action_name, class_name, flags);

734         /* allocate nvlist */
735         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
736                 ipqos_msg(MT_ENOSTR, "nvlist_alloc");
737                 return (IPQOS_CONF_ERR);
738         }

740         /* add 'remove class' config type */
741         if (nvlist_add_byte(nvl, IPP_CONFIG_TYPE, CLASSIFIER_REMOVE_CLASS) !=
742             0) {
743                 ipqos_msg(MT_ENOSTR, "nvlist_add_byte");
744                 goto fail;
745         }

747         /*
748          * add module version
749          */
750         if (nvlist_add_uint32(nvl, IPP_MODULE_VERSION,
751             (uint32_t)module_version) != 0) {
752                 ipqos_msg(MT_ENOSTR, "nvlist_add_uint32");
753                 goto fail;
754         }

756         /* add class name */
757         if (nvlist_add_string(nvl, CLASSIFIER_CLASS_NAME, class_name) != 0) {
```

```
 758                 ipqos_msg(MT_ENOSTR, "nvlist_add_string");
 759                 goto fail;
 760         }

 762         if (ipp_action_modify(action_name, &nvl, flags) != 0) {

 764                 /* generic error message */

 766                 ipqos_msg(MT_ERROR,
 767                     gettext("Removing class %s in action %s failed: %s.\n"),
 768                     class_name, action_name, strerror(errno));

 770                 goto fail;
 771         }

 773         return (IPQOS_CONF_SUCCESS);
 774 fail:
 779         if (nvl != NULL)
 775         nvlist_free(nvl);
 776         return (IPQOS_CONF_ERR);
 777 }
______unchanged_portion_omitted_

6333 /* frees up all memory occupied by a filter struct and its contents. */
6334 static void
6335 free_class(ipqos_conf_class_t *cls)
6336 {

6338         if (cls == NULL)
6339                 return;

6341         /* free its nvlist if present */

6348         if (cls->nvlist)
6343         nvlist_free(cls->nvlist);

6345         /* free its action refs if present */

6347         if (cls->alist)
6348                 free_arefs(cls->alist);

6350         /* finally free class itself */
6351         free(cls);
6352 }
______unchanged_portion_omitted_

6775 /*
6776  * free all the memory used by the action references in arefs.
6777  */
6778 static void
6779 free_arefs(
6780 ipqos_conf_act_ref_t *arefs)
6781 {

6783         ipqos_conf_act_ref_t *aref = arefs;
6784         ipqos_conf_act_ref_t *next;

6786         while (aref) {
6793                 if (aref->nvlist)
6787                 nvlist_free(aref->nvlist);
6788                 next = aref->next;
6789                 free(aref);
6790                 aref = next;
6791         }
6792 }
______unchanged_portion_omitted_
```

****
52678 Mon Feb 15 12:55:58 2016
new/usr/src/cmd/dcs/sparc/sun4u/ri_init.c
patch tsoome-feedback
****
_____***unchanged_portion_omitted_***

```
 660 /*
 661  * RCM capacity change request for cpus.
 662  */
 663 static int
 664 cpu_cap_request(ri_hdl_t *ri_hdl, rcmd_t *rcm)
 665 {
 666         cpuid_t         *syscpuids, *newcpuids;
 667         int             sysncpus, newncpus;
 668         rcm_info_t      *rcm_info = NULL;
 669         int             i, j, k;
 670         nvlist_t        *nvl;
 671         int             rv = 0;

 673         /* get all cpus in the system */
 674         if (syscpus(&syscpuids, &sysncpus) == -1)
 675                 return (-1);

 677         newncpus = sysncpus - rcm->ncpus;
 678         if ((newcpuids = calloc(newncpus, sizeof (cpuid_t))) == NULL) {
 679                 dprintf((stderr, "calloc: %s", strerror(errno)));
 680                 rv = -1;
 681                 goto out;
 682         }

 684         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
 685                 dprintf((stderr, "nvlist_alloc fail\n"));
 686                 rv = -1;
 687                 goto out;
 688         }

 690         /*
 691          * Construct the new cpu list.
 692          */
 693         for (i = 0, j = 0; i < sysncpus; i++) {
 694                 for (k = 0; k < rcm->ncpus; k++) {
 695                         if (rcm->cpus[k] == syscpuids[i]) {
 696                                 break;
 697                         }
 698                 }
 699                 if (k == rcm->ncpus) {
 700                         newcpuids[j++] = syscpuids[i];
 701                 }
 702         }

 704         if (nvlist_add_int32(nvl, "old_total", sysncpus) != 0 ||
 705             nvlist_add_int32(nvl, "new_total", newncpus) != 0 ||
 706             nvlist_add_int32_array(nvl, "old_cpu_list", syscpuids,
 707             sysncpus) != 0 ||
 708             nvlist_add_int32_array(nvl, "new_cpu_list", newcpuids,
 709             newncpus) != 0) {
 710                 dprintf((stderr, "nvlist_add fail\n"));
 711                 rv = -1;
 712                 goto out;
 713         }

 715 #ifdef DEBUG
 716         dprintf((stderr, "old_total=%d\n", sysncpus));
 717         for (i = 0; i < sysncpus; i++) {
 718                 dprintf((stderr, "old_cpu_list[%d]=%d\n", i, syscpuids[i]));
```



```
 719         }
 720         dprintf((stderr, "new_total=%d\n", newncpus));
 721         for (i = 0; i < newncpus; i++) {
 722                 dprintf((stderr, "new_cpu_list[%d]=%d\n", i, newcpuids[i]));
 723         }
 724 #endif /* DEBUG */

 726         (void) rcm_request_capacity_change(rcm->hdl, RCM_CPU_ALL,
 727             RCM_QUERY|RCM_SCOPE, nvl, &rcm_info);

 729         rv = add_rcm_clients(&ri_hdl->cpu_cap_clients, rcm, rcm_info, 0, NULL);

 731 out:
 732         s_free(syscpuids);
 733         s_free(newcpuids);
 734         if (nvl != NULL)
 734         nvlist_free(nvl);
 735         if (rcm_info != NULL)
 736                 rcm_free_info(rcm_info);

 738         return (rv);
 739 }
```

_____***unchanged_portion_omitted_***

```
1758 /*
1759  * Create and link attachment point handle.
1760  */
1761 static ri_ap_t *
1762 ri_ap_alloc(char *ap_id, ri_hdl_t *hdl)
1763 {
1764         ri_ap_t         *ap, *tmp;

1766         if ((ap = calloc(1, sizeof (*ap))) == NULL) {
1767                 dprintf((stderr, "calloc: %s\n", strerror(errno)));
1768                 return (NULL);
1769         }

1771         if (nvlist_alloc(&ap->conf_props, NV_UNIQUE_NAME, 0) != 0 ||
1772             nvlist_add_string(ap->conf_props, RI_AP_REQ_ID, ap_id) != 0) {
1774                 if (ap->conf_props != NULL)
1773                 nvlist_free(ap->conf_props);
1774                 free(ap);
1775                 return (NULL);
1776         }

1778         if ((tmp = hdl->aps) == NULL) {
1779                 hdl->aps = ap;
1780         } else {
1781                 while (tmp->next != NULL) {
1782                         tmp = tmp->next;
1783                 }
1784                 tmp->next = ap;
1785         }

1787         return (ap);
1788 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   21333 Mon Feb 15 12:55:58 2016
new/usr/src/cmd/dcs/sparc/sun4u/rsrc_info.c
patch tsoome-feedback
**********************************************************
```

```
/*
 * 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 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * Routines for traversing and packing/unpacking the handle
 * returned from ri_init.
 */

#include <stdlib.h>
#include <strings.h>
#include "rsrc_info.h"
#include "rsrc_info_impl.h"

static int ap_list_pack(ri_ap_t *, char **, size_t *, int);
static int ap_list_unpack(char *, size_t, ri_ap_t **);
static int ap_pack(ri_ap_t *, char **, size_t *, int);
static int ap_unpack(char *, size_t, ri_ap_t *);
static int dev_list_pack(ri_dev_t *, char **, size_t *, int);
static int dev_list_unpack(char *, size_t, ri_dev_t **);
static int dev_pack(ri_dev_t *, char **, size_t *, int);
static int dev_unpack(char *, size_t, ri_dev_t *);
static int client_list_pack(ri_client_t *, char **, size_t *, int);
static int client_list_unpack(char *, size_t, ri_client_t **);
static int client_pack(ri_client_t *, char **, size_t *, int);
static int client_unpack(char *, size_t, ri_client_t *);
static int pack_add_byte_array(nvlist_t *, char *, nvlist_t *, int);
static int lookup_unpack_byte_array(nvlist_t *, char *, nvlist_t **);
static void ri_ap_free(ri_ap_t *);

void
ri_fini(ri_hdl_t *hdl)
{
        ri_ap_t         *ap;
        ri_client_t     *client;

        if (hdl == NULL)
                return;
```



```
        while ((ap = hdl->aps) != NULL) {
                hdl->aps = ap->next;
                ri_ap_free(ap);
        }
        while ((client = hdl->cpu_cap_clients) != NULL) {
                hdl->cpu_cap_clients = client->next;
                ri_client_free(client);
        }
        while ((client = hdl->mem_cap_clients) != NULL) {
                hdl->mem_cap_clients = client->next;
                ri_client_free(client);
        }
        free(hdl);
}

static void
ri_ap_free(ri_ap_t *ap)
{
        ri_dev_t        *dev;

        assert(ap != NULL);

        if (ap->conf_props != NULL)
        nvlist_free(ap->conf_props);

        while ((dev = ap->cpus) != NULL) {
                ap->cpus = dev->next;
                ri_dev_free(dev);
        }
        while ((dev = ap->mems) != NULL) {
                ap->mems = dev->next;
                ri_dev_free(dev);
        }
        while ((dev = ap->ios) != NULL) {
                ap->ios = dev->next;
                ri_dev_free(dev);
        }
        free(ap);
}
_____unchanged_portion_omitted_

void
ri_client_free(ri_client_t *client)
{
        assert(client != NULL);

        if (client->usg_props != NULL)
        nvlist_free(client->usg_props);
        if (client->v_props != NULL)
        nvlist_free(client->v_props);
        free(client);
}

/*
 * Pack everything contained in the handle up inside out.
 */
int
ri_pack(ri_hdl_t *hdl, caddr_t *bufp, size_t *sizep, int encoding)
{
        nvlist_t        *nvl = NULL;
        char            *buf = NULL;
        size_t          size = 0;

        if (bufp == NULL || sizep == NULL)
                return (RI_INVAL);
```

```
138         *sizep = 0;
139         *bufp = NULL;

141         /*
142          * Check the handle. If it is NULL, there
143          * is nothing to pack, so we are done.
144          */
145         if (hdl == NULL) {
146                 return (RI_SUCCESS);
147         }

149         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
150                 dprintf((stderr, "nvlist_alloc fail\n", errno));
151                 goto fail;
152         }

154         if (nvlist_add_int32(nvl, RI_HDL_FLAGS, hdl->flags) != 0) {
155                 dprintf((stderr, "nvlist_add_int32 fail\n"));
156                 goto fail;
157         }

159         if (ap_list_pack(hdl->aps, &buf, &size, encoding) != 0 ||
160             nvlist_add_byte_array(nvl, RI_HDL_APS, (uchar_t *)buf, size) != 0) {
161                 goto fail;
162         }

164         s_free(buf);
165         if (client_list_pack(hdl->cpu_cap_clients, &buf, &size,
166             encoding) != 0 ||
167             nvlist_add_byte_array(nvl, RI_HDL_CPU_CAPS, (uchar_t *)buf,
168             size) != 0) {
169                 goto fail;
170         }

172         s_free(buf);
173         if (client_list_pack(hdl->mem_cap_clients, &buf, &size,
174             encoding) != 0 ||
175             nvlist_add_byte_array(nvl, RI_HDL_MEM_CAPS, (uchar_t *)buf,
176             size) != 0) {
177                 goto fail;
178         }

180         s_free(buf);
181         if (nvlist_pack(nvl, &buf, &size, encoding, 0) != 0) {
182                 dprintf((stderr, "nvlist_pack fail\n"));
183                 goto fail;
184         }

186         nvlist_free(nvl);
187         *bufp = buf;
188         *sizep = size;

190         return (RI_SUCCESS);

192 fail:
193         s_free(buf);
199         if (nvl != NULL)
194         nvlist_free(nvl);

196         return (RI_FAILURE);
197 }

199 /*
200  * Pack a list of attachment point handles.
201  */
```



```
202 static int
203 ap_list_pack(ri_ap_t *aplist, char **bufp, size_t *sizep, int encoding)
204 {
205         nvlist_t        *nvl = NULL;
206         char            *buf = NULL;
207         size_t          size;

209         assert(bufp != NULL && sizep != NULL);

211         *sizep = 0;
212         *bufp = NULL;

214         if (nvlist_alloc(&nvl, 0, 0) != 0) {
215                 dprintf((stderr, "nvlist_alloc fail\n"));
216                 return (-1);
217         }

219         while (aplist != NULL) {
220                 s_free(buf);
221                 if (ap_pack(aplist, &buf, &size, encoding) != 0)
222                         goto fail;

224                 if (nvlist_add_byte_array(nvl, RI_AP_T, (uchar_t *)buf,
225                     size) != 0) {
226                         dprintf((stderr, "nvlist_add_byte_array fail "
227                             "(%s)\n", RI_AP_T));
228                         goto fail;
229                 }
230                 aplist = aplist->next;
231         }

233         s_free(buf);
234         if (nvlist_pack(nvl, &buf, &size, encoding, 0) != 0) {
235                 dprintf((stderr, "nvlist_pack fail\n"));
236                 goto fail;
237         }

239         nvlist_free(nvl);
240         *bufp = buf;
241         *sizep = size;

243         return (0);

245 fail:
246         s_free(buf);
253         if (nvl != NULL)
247         nvlist_free(nvl);

249         return (-1);
250 }

252 /*
253  * Pack a list of ri_dev_t's.
254  */
255 static int
256 dev_list_pack(ri_dev_t *devlist, char **bufp, size_t *sizep, int encoding)
257 {
258         nvlist_t        *nvl = NULL;
259         char            *buf = NULL;
260         size_t          size = 0;

262         assert(bufp != NULL && sizep != NULL);

264         *sizep = 0;
265         *bufp = NULL;
```

```
267         if (nvlist_alloc(&nvl, 0, 0) != 0) {
268                 dprintf((stderr, "nvlist_alloc fail\n"));
269                 return (-1);
270         }

272         while (devlist != NULL) {
273                 s_free(buf);
274                 if (dev_pack(devlist, &buf, &size, encoding) != 0)
275                         goto fail;

277                 if (nvlist_add_byte_array(nvl, RI_DEV_T, (uchar_t *)buf,
278                     size) != 0) {
279                         dprintf((stderr, "nvlist_add_byte_array fail "
280                             "(%s)\n", RI_DEV_T));
281                         goto fail;
282                 }
283                 devlist = devlist->next;
284         }

286         s_free(buf);
287         if (nvlist_pack(nvl, &buf, &size, encoding, 0) != 0) {
288                 dprintf((stderr, "nvlist_pack fail\n"));
289                 goto fail;
290         }

292         nvlist_free(nvl);
293         *bufp = buf;
294         *sizep = size;

296         return (0);

298 fail:
299         s_free(buf);
307         if (nvl != NULL)
300         nvlist_free(nvl);

302         return (-1);
303 }

305 /*
306  * Pack a list of ri_client_t's.
307  */
308 static int
309 client_list_pack(ri_client_t *client_list, char **bufp, size_t *sizep,
310     int encoding)
311 {
312         nvlist_t        *nvl = NULL;
313         char            *buf = NULL;
314         size_t          size = 0;

316         assert(bufp != NULL && sizep != NULL);

318         *sizep = 0;
319         *bufp = NULL;

321         if (nvlist_alloc(&nvl, 0, 0) != 0) {
322                 dprintf((stderr, "nvlist_alloc fail\n"));
323                 return (-1);
324         }

326         while (client_list != NULL) {
327                 s_free(buf);
328                 if (client_pack(client_list, &buf, &size, encoding) != 0)
329                         goto fail;

331                 if (nvlist_add_byte_array(nvl, RI_CLIENT_T, (uchar_t *)buf,
```



```
332                     size) != 0) {
333                         dprintf((stderr, "nvlist_add_byte_array fail "
334                             "(%s)\n", RI_CLIENT_T));
335                         goto fail;
336                 }
337                 client_list = client_list->next;
338         }

340         s_free(buf);
341         if (nvlist_pack(nvl, &buf, &size, encoding, 0) != 0) {
342                 dprintf((stderr, "nvlist_pack fail\n"));
343                 goto fail;
344         }

346         nvlist_free(nvl);
347         *bufp = buf;
348         *sizep = size;

350         return (0);

352 fail:
353         s_free(buf);
362         if (nvl != NULL)
354         nvlist_free(nvl);

356         return (-1);
357 }

359 static int
360 ap_pack(ri_ap_t *ap, char **bufp, size_t *sizep, int encoding)
361 {
362         nvlist_t        *nvl = NULL;
363         char            *buf = NULL;
364         size_t          size = 0;

366         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
367                 dprintf((stderr, "nvlist_alloc fail\n"));
368                 return (-1);
369         }

371         if (pack_add_byte_array(ap->conf_props, RI_AP_PROPS, nvl,
372             encoding) != 0)
373                 goto fail;

375         if (dev_list_pack(ap->cpus, &buf, &size, encoding) != 0)
376                 goto fail;

378         if (nvlist_add_byte_array(nvl, RI_AP_CPUS, (uchar_t *)buf,
379             size) != 0) {
380                 dprintf((stderr, "nvlist_add_byte_array (%s)\n", RI_AP_CPUS));
381                 goto fail;
382         }

384         s_free(buf);
385         if (dev_list_pack(ap->mems, &buf, &size, encoding) != 0)
386                 goto fail;

388         if (nvlist_add_byte_array(nvl, RI_AP_MEMS, (uchar_t *)buf,
389             size) != 0) {
390                 dprintf((stderr, "nvlist_add_byte_array (%s)n", RI_AP_MEMS));
391                 goto fail;
392         }

394         s_free(buf);
395         if (dev_list_pack(ap->ios, &buf, &size, encoding) != 0)
396                 goto fail;
```

```
398         if (nvlist_add_byte_array(nvl, RI_AP_IOS, (uchar_t *)buf,
399             size) != 0) {
400                 dprintf((stderr, "nvlist_add_byte_array (%s)n", RI_AP_IOS));
401                 goto fail;
402         }

404         s_free(buf);
405         if (nvlist_pack(nvl, &buf, &size, encoding, 0) != 0) {
406                 dprintf((stderr, "nvlist_pack fail\n"));
407                 goto fail;
408         }

410         nvlist_free(nvl);
411         *bufp = buf;
412         *sizep = size;

414         return (0);

416 fail:
417         s_free(buf);
427         if (nvl != NULL)
418         nvlist_free(nvl);

420         return (-1);
421 }

423 static int
424 dev_pack(ri_dev_t *dev, char **bufp, size_t *sizep, int encoding)
425 {
426         nvlist_t        *nvl = NULL;
427         char            *buf = NULL;
428         size_t          size = 0;

430         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
431                 dprintf((stderr, "nvlist_alloc fail\n"));
432                 return (-1);
433         }

435         if (pack_add_byte_array(dev->conf_props, RI_DEV_PROPS, nvl,
436             encoding) != 0)
437                 goto fail;

439         if (client_list_pack(dev->rcm_clients, &buf, &size, encoding) != 0)
440                 goto fail;

442         if (nvlist_add_byte_array(nvl, RI_DEV_CLIENTS, (uchar_t *)buf,
443             size) != 0) {
444                 dprintf((stderr, "nvlist_add_byte_array (%s)n",
445                     RI_DEV_CLIENTS));
446                 goto fail;
447         }

449         s_free(buf);
450         if (nvlist_pack(nvl, &buf, &size, encoding, 0) != 0) {
451                 dprintf((stderr, "nvlist_pack fail\n"));
452                 goto fail;
453         }

455         nvlist_free(nvl);
456         *bufp = buf;
457         *sizep = size;

459         return (0);

461 fail:
```



```
462         s_free(buf);
473         if (nvl != NULL)
463         nvlist_free(nvl);

465         return (-1);
466 }

468 static int
469 client_pack(ri_client_t *client, char **bufp, size_t *sizep, int encoding)
470 {
471         nvlist_t        *nvl = NULL;
472         char            *buf = NULL;
473         size_t          size = 0;

475         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
476                 dprintf((stderr, "nvlist_alloc fail\n"));
477                 return (-1);
478         }

480         if (pack_add_byte_array(client->usg_props, RI_CLIENT_USAGE_PROPS,
481             nvl, encoding) != 0) {
482                 goto fail;
483         }

485         /*
486          * This will only be present if RI_VERBOSE was specified
487          * in the call to ri_init.
488          */
489         if (client->v_props != NULL && pack_add_byte_array(client->v_props,
490             RI_CLIENT_VERB_PROPS, nvl, encoding) != 0) {
491                 goto fail;
492         }

494         if (nvlist_pack(nvl, &buf, &size, encoding, 0) != 0) {
495                 dprintf((stderr, "nvlist_pack fail\n"));
496                 goto fail;
497         }

499         nvlist_free(nvl);
500         *bufp = buf;
501         *sizep = size;

503         return (0);

505 fail:
506         s_free(buf);
518         if (nvl != NULL)
507         nvlist_free(nvl);

509         return (-1);
510 }
_____unchanged_portion_omitted_

537 /*
538  * Unpack buf into ri_hdl_t.
539  */
540 int
541 ri_unpack(caddr_t buf, size_t size, ri_hdl_t **hdlp)
542 {
543         ri_hdl_t        *ri_hdl = NULL;
544         nvlist_t        *nvl = NULL;

546         if (hdlp == NULL)
547                 return (RI_INVAL);

549         *hdlp = NULL;
```

```
550         if ((ri_hdl = calloc(1, sizeof (*ri_hdl))) == NULL) {
551                 dprintf((stderr, "calloc: %s\n", strerror(errno)));
552                 return (RI_FAILURE);
553         }

555         if (nvlist_unpack(buf, size, &nvl, 0) != 0) {
556                 dprintf((stderr, "nvlist_unpack fail\n"));
557                 goto fail;
558         }

560         if (nvlist_lookup_int32(nvl, RI_HDL_FLAGS, &ri_hdl->flags) != 0) {
561                 dprintf((stderr, "nvlist_lookup_int32 fail (%s)\n",
562                     RI_HDL_FLAGS));
563                 goto fail;
564         }

566         buf = NULL;
567         size = 0;
568         if (nvlist_lookup_byte_array(nvl, RI_HDL_APS, (uchar_t **)&buf,
569             (uint_t *)&size) != 0) {
570                 dprintf((stderr, "nvlist_lookup_int32 fail (%s)\n",
571                     RI_HDL_APS));
572                 goto fail;
573         }

575         if (ap_list_unpack(buf, size, &ri_hdl->aps) != 0)
576                 goto fail;

578         buf = NULL;
579         size = 0;
580         if (nvlist_lookup_byte_array(nvl, RI_HDL_CPU_CAPS, (uchar_t **)&buf,
581             (uint_t *)&size) != 0) {
582                 dprintf((stderr, "nvlist_lookup_byte_array fail (%s)\n",
583                     RI_HDL_CPU_CAPS));
584                 goto fail;
585         }

587         if (client_list_unpack(buf, size, &ri_hdl->cpu_cap_clients) != 0)
588                 goto fail;

590         buf = NULL;
591         size = 0;
592         if (nvlist_lookup_byte_array(nvl, RI_HDL_MEM_CAPS, (uchar_t **)&buf,
593             (uint_t *)&size) != 0) {
594                 dprintf((stderr, "nvlist_lookup_byte_array fail (%s)\n",
595                     RI_HDL_MEM_CAPS));
596                 goto fail;
597         }

599         if (client_list_unpack(buf, size, &ri_hdl->mem_cap_clients) != 0)
600                 goto fail;

602         *hdlp = ri_hdl;

604         return (0);

606 fail:
607         free(ri_hdl);
620         if (nvl != NULL)
608         nvlist_free(nvl);

610         return (-1);
611 }

613 static int
614 ap_list_unpack(char *buf, size_t size, ri_ap_t **aps)
```



```
615 {
616         nvpair_t        *nvp = NULL;
617         nvlist_t        *nvl;
618         ri_ap_t         *aplist = NULL;
619         ri_ap_t         *prev = NULL;
620         ri_ap_t         *tmp = NULL;

622         if (nvlist_unpack(buf, size, &nvl, 0) != 0) {
623                 dprintf((stderr, "nvlist_unpack fail\n"));
624                 return (-1);
625         }

627         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
628                 assert(strcmp(nvpair_name(nvp), RI_AP_T) == 0 &&
629                     nvpair_type(nvp) == DATA_TYPE_BYTE_ARRAY);

631                 if ((tmp = calloc(1, sizeof (*tmp))) == NULL) {
632                         dprintf((stderr, "calloc: %s\n", strerror(errno)));
633                         goto fail;
634                 }

636                 buf = NULL;
637                 size = 0;
638                 if (nvpair_value_byte_array(nvp, (uchar_t **)&buf,
639                     (uint_t *)&size) != 0) {
640                         dprintf((stderr, "nvpair_value_byte_array fail\n"));
641                         goto fail;
642                 }

644                 if (ap_unpack(buf, size, tmp) != 0)
645                         goto fail;

647                 if (aplist == NULL) {
648                         prev = aplist = tmp;
649                 } else {
650                         prev->next = tmp;
651                         prev = tmp;
652                 }
653         }

655         nvlist_free(nvl);
656         *aps = aplist;

658         return (0);

660 fail:
674         if (nvl != NULL)
661         nvlist_free(nvl);
662         if (aplist != NULL) {
663                 while ((tmp = aplist) != NULL) {
664                         aplist = aplist->next;
665                         ri_ap_free(tmp);
666                 }
667         }

669         return (-1);
670 }

672 static int
673 dev_list_unpack(char *buf, size_t size, ri_dev_t **devs)
674 {
675         nvpair_t        *nvp = NULL;
676         nvlist_t        *nvl;
677         ri_dev_t        *devlist = NULL;
678         ri_dev_t        *prev = NULL;
679         ri_dev_t        *tmp = NULL;
```

```
681         if (nvlist_unpack(buf, size, &nvl, 0) != 0) {
682                 dprintf((stderr, "nvlist_unpack fail\n"));
683                 return (-1);
684         }

686         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
687                 assert(strcmp(nvpair_name(nvp), RI_DEV_T) == 0 &&
688                     nvpair_type(nvp) == DATA_TYPE_BYTE_ARRAY);

690                 if ((tmp = calloc(1, sizeof (*tmp))) == NULL) {
691                         dprintf((stderr, "calloc: %s\n", strerror(errno)));
692                         goto fail;
693                 }

695                 if (nvpair_value_byte_array(nvp, (uchar_t **)&buf,
696                     (uint_t *)&size) != 0) {
697                         dprintf((stderr, "nvpair_value_byte_array fail\n"));
698                         goto fail;
699                 }

701                 if (dev_unpack(buf, size, tmp) != 0)
702                         goto fail;

704                 if (devlist == NULL) {
705                         prev = devlist = tmp;
706                 } else {
707                         prev->next = tmp;
708                         prev = tmp;
709                 }
710         }

712         nvlist_free(nvl);
713         *devs = devlist;

715         return (0);

717 fail:
732         if (nvl != NULL)
718         nvlist_free(nvl);
719         if (devlist != NULL) {
720                 while ((tmp = devlist) != NULL) {
721                         devlist = devlist->next;
722                         ri_dev_free(tmp);
723                 }
724         }

726         return (-1);
727 }

729 static int
730 client_list_unpack(char *buf, size_t size, ri_client_t **clients)
731 {
732         nvpair_t        *nvp = NULL;
733         nvlist_t        *nvl;
734         ri_client_t     *client_list = NULL;
735         ri_client_t     *prev = NULL;
736         ri_client_t     *tmp = NULL;

738         if (nvlist_unpack(buf, size, &nvl, 0) != 0) {
739                 dprintf((stderr, "nvlist_unpack fail\n"));
740                 return (-1);
741         }

743         while ((nvp = nvlist_next_nvpair(nvl, nvp)) != NULL) {
744                 assert(strcmp(nvpair_name(nvp), RI_CLIENT_T) == 0);
```



```
745                 assert(nvpair_type(nvp) == DATA_TYPE_BYTE_ARRAY);

747                 if ((tmp = calloc(1, sizeof (*tmp))) == NULL) {
748                         dprintf((stderr, "calloc: %s\n", strerror(errno)));
749                         goto fail;
750                 }

752                 buf = NULL;
753                 size = 0;
754                 if (nvpair_value_byte_array(nvp, (uchar_t **)&buf,
755                     (uint_t *)&size) != 0) {
756                         dprintf((stderr, "nvpair_value_byte_array fail\n"));
757                         goto fail;
758                 }

760                 if (client_unpack(buf, size, tmp) != 0)
761                         goto fail;

763                 if (client_list == NULL) {
764                         prev = client_list = tmp;
765                 } else {
766                         prev->next = tmp;
767                         prev = tmp;
768                 }
769         }

771         nvlist_free(nvl);
772         *clients = client_list;

774         return (0);

776 fail:
792         if (nvl != NULL)
777         nvlist_free(nvl);
778         if (client_list != NULL) {
779                 while ((tmp = client_list) != NULL) {
780                         client_list = client_list->next;
781                         ri_client_free(tmp);
782                 }
783         }

785         return (-1);
786 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  211625 Mon Feb 15 12:55:58 2016
new/usr/src/cmd/devfsadm/devfsadm.c
patch tsoome-feedback
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

8186 /*
8187  * Build an nvlist containing all attributes for devfs events.
8188  * Returns nvlist pointer on success, NULL on failure.
8189  */
8190 static nvlist_t *
8191 build_event_attributes(char *class, char *subclass, char *node_path,
8192     di_node_t node, char *driver_name, int instance, char *minor)
8193 {
8194         nvlist_t *nvl;
8195         int err = 0;
8196         di_prop_t prop;
8197         int count;
8198         char *prop_name;
8199         int x;
8200         char *dev_name = NULL;
8201         int dev_name_lookup_err = 0;

8203         if ((err = nvlist_alloc(&nvl, NV_UNIQUE_NAME_TYPE, 0)) != 0) {
8204                 nvl = NULL;
8205                 goto out;
8206         }

8208         if ((err = nvlist_add_int32(nvl, EV_VERSION, EV_V1)) != 0)
8209                 goto out;

8211         if ((err = nvlist_add_string(nvl, DEV_PHYS_PATH, node_path)) != 0)
8212                 goto out;

8214         if (strcmp(class, EC_DEV_ADD) != 0 &&
8215             strcmp(class, EC_DEV_REMOVE) != 0)
8216                 return (nvl);

8218         if (driver_name == NULL || instance == -1)
8219                 goto out;

8221         if (strcmp(subclass, ESC_DISK) == 0) {
8222                 if ((dev_name = lookup_disk_dev_name(node_path)) == NULL) {
8223                         dev_name_lookup_err = 1;
8224                         goto out;
8225                 }
8226         } else if (strcmp(subclass, ESC_NETWORK) == 0) {
8227                 if ((dev_name = lookup_network_dev_name(node_path, driver_name))
8228                     == NULL) {
8229                         dev_name_lookup_err = 1;
8230                         goto out;
8231                 }
8232         } else if (strcmp(subclass, ESC_PRINTER) == 0) {
8233                 if ((dev_name = lookup_printer_dev_name(node_path)) == NULL) {
8234                         dev_name_lookup_err = 1;
8235                         goto out;
8236                 }
8237         } else if (strcmp(subclass, ESC_LOFI) == 0) {
8238                 /*
8239                  * The raw minor node is created or removed after the block
8240                  * node.  Lofi devfs events are dependent on this behavior.
8241                  * Generate the sysevent only for the raw minor node.
8242                  */
8243                 if (strstr(minor, "raw") == NULL) {
```



```
8244                         if (nvl) {
8245                                 nvlist_free(nvl);
8246                         }
8247                         return (NULL);
8248                 }
8249                 if ((dev_name = lookup_lofi_dev_name(node_path, minor)) ==
8250                     NULL) {
8251                         dev_name_lookup_err = 1;
8252                         goto out;
8253                 }
8254         }

8256         if (dev_name) {
8257                 if ((err = nvlist_add_string(nvl, DEV_NAME, dev_name)) != 0)
8258                         goto out;
8259                 free(dev_name);
8260                 dev_name = NULL;
8261         }

8263         if ((err = nvlist_add_string(nvl, DEV_DRIVER_NAME, driver_name)) != 0)
8264                 goto out;

8266         if ((err = nvlist_add_int32(nvl, DEV_INSTANCE, instance)) != 0)
8267                 goto out;

8269         if (strcmp(class, EC_DEV_ADD) == 0) {
8270                 /* add properties */
8271                 count = 0;
8272                 for (prop = di_prop_next(node, DI_PROP_NIL);
8273                     prop != DI_PROP_NIL && count < MAX_PROP_COUNT;
8274                     prop = di_prop_next(node, prop)) {

8276                         if (di_prop_devt(prop) != DDI_DEV_T_NONE)
8277                                 continue;

8279                         if ((x = add_property(nvl, prop)) == 0)
8280                                 count++;
8281                         else if (x == -1) {
8282                                 if ((prop_name = di_prop_name(prop)) == NULL)
8283                                         prop_name = "";
8284                                 err_print(PROP_ADD_FAILED, prop_name);
8285                                 goto out;
8286                         }
8287                 }
8288         }

8290         return (nvl);

8292 out:
8293         if (nvl)
8293         nvlist_free(nvl);

8295         if (dev_name)
8296                 free(dev_name);

8298         if (dev_name_lookup_err) {
8299                 /*
8300                  * If a lofi mount fails, the /devices node may well have
8301                  * disappeared by the time we run, so let's not complain.
8302                  */
8303                 if (strcmp(subclass, ESC_LOFI) != 0)
8304                         err_print(DEV_NAME_LOOKUP_FAILED, node_path);
8305         } else {
8306                 err_print(BUILD_EVENT_ATTR_FAILED, (err) ? strerror(err) : "");
8307         }
8308         return (NULL);
```

```
8309 }
_____unchanged_portion_omitted_

8350 static void
8351 process_syseventq()
8352 {
8353         (void) mutex_lock(&syseventq_mutex);
8354         while (syseventq_back != NULL) {
8355                 syseventq_t *tmp = syseventq_back;

8357                 vprint(CHATTY_MID, "sending queued event: %s, %s\n",
8358                     tmp->class, tmp->subclass);

8360                 log_event(tmp->class, tmp->subclass, tmp->nvl);

8362                 if (tmp->class != NULL)
8363                         free(tmp->class);
8364                 if (tmp->subclass != NULL)
8365                         free(tmp->subclass);
8367                 if (tmp->nvl != NULL)
8366                 nvlist_free(tmp->nvl);
8367                 syseventq_back = syseventq_back->next;
8368                 if (syseventq_back == NULL)
8369                         syseventq_front = NULL;
8370                 free(tmp);
8371         }
8372         (void) mutex_unlock(&syseventq_mutex);
8373 }
_____unchanged_portion_omitted_
```

****
73822 Mon Feb 15 12:55:58 2016
new/usr/src/cmd/fm/fmd/common/fmd_case.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
1315 static void
1316 fmd_case_destroy_suspects(fmd_case_impl_t *cip)
1317 {
1318         fmd_case_susp_t *cis, *ncis;

1320         ASSERT(MUTEX_HELD(&cip->ci_lock));

1322         if (cip->ci_proxy_asru)
1323                 fmd_free(cip->ci_proxy_asru, sizeof (uint8_t) *
1324                     cip->ci_nsuspects);
1325         if (cip->ci_diag_de)
1325         nvlist_free(cip->ci_diag_de);
1326         if (cip->ci_diag_asru)
1327                 fmd_free(cip->ci_diag_asru, sizeof (uint8_t) *
1328                     cip->ci_nsuspects);

1330         for (cis = cip->ci_suspects; cis != NULL; cis = ncis) {
1331                 ncis = cis->cis_next;
1332                 nvlist_free(cis->cis_nvl);
1333                 fmd_free(cis, sizeof (fmd_case_susp_t));
1334         }

1336         cip->ci_suspects = NULL;
1337         cip->ci_nsuspects = 0;
1338 }
```

_____***unchanged_portion_omitted_***

```
2428 void
2429 fmd_case_set_de_fmri(fmd_case_t *cp, nvlist_t *nvl)
2430 {
2431         fmd_case_impl_t *cip = (fmd_case_impl_t *)cp;

2434         if (cip->ci_diag_de)
2433         nvlist_free(cip->ci_diag_de);
2434         cip->ci_diag_de = nvl;
2435 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10325 Mon Feb 15 12:55:59 2016
new/usr/src/cmd/fm/fmd/common/fmd_event.c
patch tsoome-feedback
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
  24  * Use is subject to license terms.
  25  */

  27 #pragma ident   "%Z%%M% %I%     %E% SMI"

  27 #include <sys/fm/protocol.h>
  28 #include <limits.h>

  30 #include <fmd_alloc.h>
  31 #include <fmd_subr.h>
  32 #include <fmd_event.h>
  33 #include <fmd_string.h>
  34 #include <fmd_module.h>
  35 #include <fmd_case.h>
  36 #include <fmd_log.h>
  37 #include <fmd_time.h>
  38 #include <fmd_topo.h>
  39 #include <fmd_ctl.h>

  41 #include <fmd.h>

  43 static void
  44 fmd_event_nvwrap(fmd_event_impl_t *ep)
  45 {
  46         (void) nvlist_remove_all(ep->ev_nvl, FMD_EVN_TTL);
  47         (void) nvlist_remove_all(ep->ev_nvl, FMD_EVN_TOD);

  49         (void) nvlist_add_uint8(ep->ev_nvl,
  50             FMD_EVN_TTL, ep->ev_ttl);
  51         (void) nvlist_add_uint64_array(ep->ev_nvl,
  52             FMD_EVN_TOD, (uint64_t *)&ep->ev_time, 2);
  53 }
```

_____*unchanged_portion_omitted_*

```
 174 void
 175 fmd_event_destroy(fmd_event_t *e)
 176 {
 177         fmd_event_impl_t *ep = (fmd_event_impl_t *)e;
```



```
 179         ASSERT(MUTEX_HELD(&ep->ev_lock));
 180         ASSERT(ep->ev_refs == 0);

 182         /*
 183          * If the current state is RECEIVED (i.e. no module has accepted the
 184          * event) and the event was logged, then change the state to DISCARDED.
 185          */
 186         if (ep->ev_state == FMD_EVS_RECEIVED)
 187                 ep->ev_state = FMD_EVS_DISCARDED;

 189         /*
 190          * If the current state is DISCARDED, ACCEPTED, or DIAGNOSED and the
 191          * event has not yet been commited, then attempt to commit it now.
 192          */
 193         if (ep->ev_state != FMD_EVS_RECEIVED && (ep->ev_flags & (
 194             FMD_EVF_VOLATILE | FMD_EVF_REPLAY)) == FMD_EVF_REPLAY)
 195                 fmd_log_commit(ep->ev_log, e);

 197         if (ep->ev_log != NULL) {
 198                 if (ep->ev_flags & FMD_EVF_REPLAY)
 199                         fmd_log_decommit(ep->ev_log, e);
 200                 fmd_log_rele(ep->ev_log);
 201         }

 203         /*
 204          * Perform any event type-specific cleanup activities, and then free
 205          * the name-value pair list and underlying event data structure.
 206          */
 207         switch (ep->ev_type) {
 208         case FMD_EVT_TIMEOUT:
 209                 fmd_free(ep->ev_data, sizeof (fmd_modtimer_t));
 210                 break;
 211         case FMD_EVT_CLOSE:
 212         case FMD_EVT_PUBLISH:
 213                 fmd_case_rele(ep->ev_data);
 214                 break;
 215         case FMD_EVT_CTL:
 216                 fmd_ctl_fini(ep->ev_data);
 217                 break;
 218         case FMD_EVT_TOPO:
 219                 fmd_topo_rele(ep->ev_data);
 220                 break;
 221         }

 225         if (ep->ev_nvl != NULL)
 223         nvlist_free(ep->ev_nvl);

 225         fmd_free(ep, sizeof (fmd_event_impl_t));
 226 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   62351 Mon Feb 15 12:55:59 2016
new/usr/src/cmd/fm/fmd/common/fmd_xprt.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

1085 /*
1086  * This function creates a local suspect list. This is used when a suspect list
1087  * is created directly by an external source like fminject.
1088  */
1089 static void
1090 fmd_xprt_list_suspect_local(fmd_xprt_t *xp, nvlist_t *nvl)
1091 {
1092         nvlist_t **nvlp;
1093         nvlist_t *de_fmri, *de_fmri_dup = NULL;
1094         int64_t *diag_time;
1095         char *code = NULL;
1096         fmd_xprt_impl_t *xip = (fmd_xprt_impl_t *)xp;
1097         fmd_case_t *cp;
1098         uint_t nelem = 0, nelem2 = 0, i;
1099         boolean_t injected;

1101         fmd_module_lock(xip->xi_queue->eq_mod);
1102         cp = fmd_case_create(xip->xi_queue->eq_mod, NULL, NULL);
1103         if (cp == NULL) {
1104                 fmd_module_unlock(xip->xi_queue->eq_mod);
1105                 return;
1106         }

1108         /*
1109          * copy diag_code if present
1110          */
1111         (void) nvlist_lookup_string(nvl, FM_SUSPECT_DIAG_CODE, &code);
1112         if (code != NULL) {
1113                 fmd_case_impl_t *cip = (fmd_case_impl_t *)cp;

1115                 cip->ci_precanned = 1;
1116                 fmd_case_setcode(cp, code);
1117         }

1119         /*
1120          * copy suspects
1121          */
1122         (void) nvlist_lookup_nvlist_array(nvl, FM_SUSPECT_FAULT_LIST, &nvlp,
1123             &nelem);
1124         for (i = 0; i < nelem; i++) {
1125                 nvlist_t *flt_copy, *asru = NULL, *fru = NULL, *rsrc = NULL;
1126                 topo_hdl_t *thp;
1127                 char *loc = NULL;
1128                 int err;

1130                 thp = fmd_fmri_topo_hold(TOPO_VERSION);
1131                 (void) nvlist_xdup(nvlp[i], &flt_copy, &fmd.d_nva);
1132                 (void) nvlist_lookup_nvlist(nvlp[i], FM_FAULT_RESOURCE, &rsrc);

1134                 /*
1135                  * If no fru specified, get it from topo
1136                  */
1137                 if (nvlist_lookup_nvlist(nvlp[i], FM_FAULT_FRU, &fru) != 0 &&
1138                     rsrc && topo_fmri_fru(thp, rsrc, &fru, &err) == 0)
1139                         (void) nvlist_add_nvlist(flt_copy, FM_FAULT_FRU, fru);
1140                 /*
1141                  * If no asru specified, get it from topo
1142                  */
1143                 if (nvlist_lookup_nvlist(nvlp[i], FM_FAULT_ASRU, &asru) != 0 &&
```



```
1144                     rsrc && topo_fmri_asru(thp, rsrc, &asru, &err) == 0)
1145                         (void) nvlist_add_nvlist(flt_copy, FM_FAULT_ASRU, asru);
1146                 /*
1147                  * If no location specified, get it from topo
1148                  */
1149                 if (nvlist_lookup_string(nvlp[i], FM_FAULT_LOCATION,
1150                     &loc) != 0) {
1151                         if (fru && topo_fmri_label(thp, fru, &loc, &err) == 0)
1152                                 (void) nvlist_add_string(flt_copy,
1153                                     FM_FAULT_LOCATION, loc);
1154                         else if (rsrc && topo_fmri_label(thp, rsrc, &loc,
1155                             &err) == 0)
1156                                 (void) nvlist_add_string(flt_copy,
1157                                     FM_FAULT_LOCATION, loc);
1158                         if (loc)
1159                                 topo_hdl_strfree(thp, loc);
1160                 }
1161                 if (fru)
1161                 nvlist_free(fru);
1163                 if (asru)
1162                 nvlist_free(asru);
1165                 if (rsrc)
1163                 nvlist_free(rsrc);
1164                 fmd_fmri_topo_rele(thp);
1165                 fmd_case_insert_suspect(cp, flt_copy);
1166         }

1168         /*
1169          * copy diag_time if present
1170          */
1171         if (nvlist_lookup_int64_array(nvl, FM_SUSPECT_DIAG_TIME, &diag_time,
1172             &nelem2) == 0 && nelem2 >= 2)
1173                 fmd_case_settime(cp, diag_time[0], diag_time[1]);

1175         /*
1176          * copy DE fmri if present
1177          */
1178         if (nvlist_lookup_nvlist(nvl, FM_SUSPECT_DE, &de_fmri) == 0) {
1179                 (void) nvlist_xdup(de_fmri, &de_fmri_dup, &fmd.d_nva);
1180                 fmd_case_set_de_fmri(cp, de_fmri_dup);
1181         }

1183         /*
1184          * copy injected if present
1185          */
1186         if (nvlist_lookup_boolean_value(nvl, FM_SUSPECT_INJECTED,
1187             &injected) == 0 && injected)
1188                 fmd_case_set_injected(cp);

1190         fmd_case_transition(cp, FMD_CASE_SOLVED, FMD_CF_SOLVED);
1191         fmd_module_unlock(xip->xi_queue->eq_mod);
1192 }

1194 /*
1195  * This function is called to create a proxy case on receipt of a list.suspect
1196  * from the diagnosing side of the transport.
1197  */
1198 static void
1199 fmd_xprt_list_suspect(fmd_xprt_t *xp, nvlist_t *nvl)
1200 {
1201         fmd_xprt_impl_t *xip = (fmd_xprt_impl_t *)xp;
1202         nvlist_t **nvlp;
1203         uint_t nelem = 0, nelem2 = 0, i;
1204         int64_t *diag_time;
1205         topo_hdl_t *thp;
1206         char *class;
```

```
1207         nvlist_t *rsrc, *asru, *de_fmri, *de_fmri_dup = NULL;
1208         nvlist_t *flt_copy;
1209         int err;
1210         nvlist_t **asrua;
1211         uint8_t *proxy_asru = NULL;
1212         int got_proxy_asru = 0;
1213         int got_hc_rsrc = 0;
1214         int got_hc_asru = 0;
1215         int got_present_rsrc = 0;
1216         uint8_t *diag_asru = NULL;
1217         char *scheme;
1218         uint8_t *statusp;
1219         char *uuid, *code;
1220         fmd_case_t *cp;
1221         fmd_case_impl_t *cip;
1222         int need_update = 0;
1223         boolean_t injected;

1225         if (nvlist_lookup_string(nvl, FM_SUSPECT_UUID, &uuid) != 0)
1226                 return;
1227         if (nvlist_lookup_string(nvl, FM_SUSPECT_DIAG_CODE, &code) != 0)
1228                 return;
1229         (void) nvlist_lookup_nvlist_array(nvl, FM_SUSPECT_FAULT_LIST, &nvlp,
1230             &nelem);

1232         /*
1233          * In order to implement FMD_XPRT_HCONLY and FMD_XPRT_HC_PRESENT_ONLY
1234          * etc we first scan the suspects to see if
1235          * - there was an asru in the received fault
1236          * - there was an hc-scheme resource in the received fault
1237          * - any hc-scheme resource in the received fault is present in the
1238          *   local topology
1239          * - any hc-scheme resource in the received fault has an asru in the
1240          *   local topology
1241          */
1242         if (nelem > 0) {
1243                 asrua = fmd_zalloc(sizeof (nvlist_t *) * nelem, FMD_SLEEP);
1244                 proxy_asru = fmd_zalloc(sizeof (uint8_t) * nelem, FMD_SLEEP);
1245                 diag_asru = fmd_zalloc(sizeof (uint8_t) * nelem, FMD_SLEEP);
1246                 thp = fmd_fmri_topo_hold(TOPO_VERSION);
1247                 for (i = 0; i < nelem; i++) {
1248                         if (nvlist_lookup_nvlist(nvlp[i], FM_FAULT_ASRU,
1249                             &asru) == 0 && asru != NULL)
1250                                 diag_asru[i] = 1;
1251                         if (nvlist_lookup_string(nvlp[i], FM_CLASS,
1252                             &class) != 0 || strncmp(class, "fault", 5) != 0)
1253                                 continue;
1254                         /*
1255                          * If there is an hc-scheme asru, use that to find the
1256                          * real asru. Otherwise if there is an hc-scheme
1257                          * resource, work out the old asru from that.
1258                          * This order is to allow a two stage evaluation
1259                          * of the asru where a fault in the diagnosing side
1260                          * is in a component not visible to the proxy side,
1261                          * but prevents a component that is visible from
1262                          * working. So the diagnosing side sets the asru to
1263                          * the latter component (in hc-scheme as the diagnosing
1264                          * side doesn't know about the proxy side's virtual
1265                          * schemes), and then the proxy side can convert that
1266                          * to a suitable virtual scheme asru.
1267                          */
1268                         if (nvlist_lookup_nvlist(nvlp[i], FM_FAULT_ASRU,
1269                             &asru) == 0 && asru != NULL &&
1270                             nvlist_lookup_string(asru, FM_FMRI_SCHEME,
1271                             &scheme) == 0 &&
1272                             strcmp(scheme, FM_FMRI_SCHEME_HC) == 0) {
```



```
1273                                 got_hc_asru = 1;
1274                                 if (xip->xi_flags & FMD_XPRT_EXTERNAL)
1275                                         continue;
1276                                 if (topo_fmri_present(thp, asru, &err) != 0)
1277                                         got_present_rsrc = 1;
1278                                 if (topo_fmri_asru(thp, asru, &asrua[i],
1279                                     &err) == 0) {
1280                                         proxy_asru[i] =
1281                                             FMD_PROXY_ASRU_FROM_ASRU;
1282                                         got_proxy_asru = 1;
1283                                 }
1284                         } else if (nvlist_lookup_nvlist(nvlp[i],
1285                             FM_FAULT_RESOURCE, &rsrc) == 0 && rsrc != NULL &&
1286                             nvlist_lookup_string(rsrc, FM_FMRI_SCHEME,
1287                             &scheme) == 0 &&
1288                             strcmp(scheme, FM_FMRI_SCHEME_HC) == 0) {
1289                                 got_hc_rsrc = 1;
1290                                 if (xip->xi_flags & FMD_XPRT_EXTERNAL)
1291                                         continue;
1292                                 if (topo_fmri_present(thp, rsrc, &err) != 0)
1293                                         got_present_rsrc = 1;
1294                                 if (topo_fmri_asru(thp, rsrc, &asrua[i],
1295                                     &err) == 0) {
1296                                         proxy_asru[i] =
1297                                             FMD_PROXY_ASRU_FROM_RSRC;
1298                                         got_proxy_asru = 1;
1299                                 }
1300                         }
1301                 }
1302                 fmd_fmri_topo_rele(thp);
1303         }

1305         /*
1306          * If we're set up only to report hc-scheme faults, and
1307          * there aren't any, then just drop the event.
1308          */
1309         if (got_hc_rsrc == 0 && got_hc_asru == 0 &&
1310             (xip->xi_flags & FMD_XPRT_HCONLY)) {
1311                 if (nelem > 0) {
1312                         fmd_free(proxy_asru, sizeof (uint8_t) * nelem);
1313                         fmd_free(diag_asru, sizeof (uint8_t) * nelem);
1314                         fmd_free(asrua, sizeof (nvlist_t *) * nelem);
1315                 }
1316                 return;
1317         }

1319         /*
1320          * If we're set up only to report locally present hc-scheme
1321          * faults, and there aren't any, then just drop the event.
1322          */
1323         if (got_present_rsrc == 0 &&
1324             (xip->xi_flags & FMD_XPRT_HC_PRESENT_ONLY)) {
1325                 if (nelem > 0) {
1326                         for (i = 0; i < nelem; i++)
1330                                 if (asrua[i])
1327                                 nvlist_free(asrua[i]);
1328                         fmd_free(proxy_asru, sizeof (uint8_t) * nelem);
1329                         fmd_free(diag_asru, sizeof (uint8_t) * nelem);
1330                         fmd_free(asrua, sizeof (nvlist_t *) * nelem);
1331                 }
1332                 return;
1333         }

1335         /*
1336          * If fmd_case_recreate() returns NULL, UUID is already known.
1337          */
```

```
1338         fmd_module_lock(xip->xi_queue->eq_mod);
1339         if ((cp = fmd_case_recreate(xip->xi_queue->eq_mod, xp,
1340             FMD_CASE_UNSOLVED, uuid, code)) == NULL) {
1341                 if (nelem > 0) {
1342                         for (i = 0; i < nelem; i++)
1347                                 if (asrua[i])
1343                                 nvlist_free(asrua[i]);
1344                         fmd_free(proxy_asru, sizeof (uint8_t) * nelem);
1345                         fmd_free(diag_asru, sizeof (uint8_t) * nelem);
1346                         fmd_free(asrua, sizeof (nvlist_t *) * nelem);
1347                 }
1348                 fmd_module_unlock(xip->xi_queue->eq_mod);
1349                 return;
1350         }

1352         cip = (fmd_case_impl_t *)cp;
1353         cip->ci_diag_asru = diag_asru;
1354         cip->ci_proxy_asru = proxy_asru;
1355         for (i = 0; i < nelem; i++) {
1356                 (void) nvlist_xdup(nvlp[i], &flt_copy, &fmd.d_nva);
1357                 if (proxy_asru[i] != FMD_PROXY_ASRU_NOT_NEEDED) {
1358                         /*
1359                          * Copy suspects, but remove/replace asru first. Also if
1360                          * the original asru was hc-scheme use that as resource.
1361                          */
1362                         if (proxy_asru[i] == FMD_PROXY_ASRU_FROM_ASRU) {
1363                                 (void) nvlist_remove(flt_copy,
1364                                     FM_FAULT_RESOURCE, DATA_TYPE_NVLIST);
1365                                 (void) nvlist_lookup_nvlist(flt_copy,
1366                                     FM_FAULT_ASRU, &asru);
1367                                 (void) nvlist_add_nvlist(flt_copy,
1368                                     FM_FAULT_RESOURCE, asru);
1369                         }
1370                         (void) nvlist_remove(flt_copy, FM_FAULT_ASRU,
1371                             DATA_TYPE_NVLIST);
1372                         (void) nvlist_add_nvlist(flt_copy, FM_FAULT_ASRU,
1373                             asrua[i]);
1374                         nvlist_free(asrua[i]);
1375                 } else if (got_hc_asru == 0 &&
1376                     nvlist_lookup_nvlist(flt_copy, FM_FAULT_ASRU,
1377                     &asru) == 0 && asru != NULL) {
1378                         /*
1379                          * If we have an asru from diag side, but it's not
1380                          * in hc scheme, then we can't be sure what it
1381                          * represents, so mark as no retire.
1382                          */
1383                         (void) nvlist_add_boolean_value(flt_copy,
1384                             FM_SUSPECT_RETIRE, B_FALSE);
1385                 }
1386                 fmd_case_insert_suspect(cp, flt_copy);
1387         }
1388         /*
1389          * copy diag_time
1390          */
1391         if (nvlist_lookup_int64_array(nvl, FM_SUSPECT_DIAG_TIME, &diag_time,
1392             &nelem2) == 0 && nelem2 >= 2)
1393                 fmd_case_settime(cp, diag_time[0], diag_time[1]);
1394         /*
1395          * copy DE fmri
1396          */
1397         if (nvlist_lookup_nvlist(nvl, FM_SUSPECT_DE, &de_fmri) == 0) {
1398                 (void) nvlist_xdup(de_fmri, &de_fmri_dup, &fmd.d_nva);
1399                 fmd_case_set_de_fmri(cp, de_fmri_dup);
1400         }

1402         /*
```



```
1403          * copy injected if present
1404          */
1405         if (nvlist_lookup_boolean_value(nvl, FM_SUSPECT_INJECTED,
1406             &injected) == 0 && injected)
1407                 fmd_case_set_injected(cp);

1409         /*
1410          * Transition to solved. This will log the suspect list and create
1411          * the resource cache entries.
1412          */
1413         fmd_case_transition(cp, FMD_CASE_SOLVED, FMD_CF_SOLVED);

1415         /*
1416          * Update status if it is not simply "all faulty" (can happen if
1417          * list.suspects are being re-sent when the transport has reconnected).
1418          */
1419         (void) nvlist_lookup_uint8_array(nvl, FM_SUSPECT_FAULT_STATUS, &statusp,
1420             &nelem);
1421         for (i = 0; i < nelem; i++) {
1422                 if ((statusp[i] & (FM_SUSPECT_FAULTY | FM_SUSPECT_UNUSABLE |
1423                     FM_SUSPECT_NOT_PRESENT | FM_SUSPECT_DEGRADED)) !=
1424                     FM_SUSPECT_FAULTY)
1425                         need_update = 1;
1426         }
1427         if (need_update) {
1428                 fmd_case_update_status(cp, statusp, cip->ci_proxy_asru,
1429                     cip->ci_diag_asru);
1430                 fmd_case_update_containees(cp);
1431                 fmd_case_update(cp);
1432         }

1434         /*
1435          * if asru on proxy side, send an update back to the diagnosing side to
1436          * update UNUSABLE/DEGRADED.
1437          */
1438         if (got_proxy_asru)
1439                 fmd_case_xprt_updated(cp);

1441         if (nelem > 0)
1442                 fmd_free(asrua, sizeof (nvlist_t *) * nelem);
1443         fmd_module_unlock(xip->xi_queue->eq_mod);
1444 }
```

unchanged_portion_omitted_

```
**********************************************************
   20539 Mon Feb 15 12:55:59 2016
new/usr/src/cmd/fm/fminject/common/inj_defn.c
patch tsoome-feedback
**********************************************************
```

```
/*
 * 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 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * After having been declared, events, FMRIs and authorities must be defined
 * (instantiated) before they can be used as the subjects of commands.
 */

#include <sys/sysmacros.h>
#include <libnvpair.h>
#include <string.h>
#include <assert.h>

#include <inj_event.h>
#include <inj_err.h>
#include <inj_lex.h>
#include <inj_string.h>
#include <inj.h>

static inj_hash_t inj_defns[3];
static int inj_defns_initialized;

/* Intrinsics (signed and unsigned integer integer constants) */
typedef struct intr {
        uchar_t ei_signed;
        uchar_t ei_width;
} intr_t;
```

*_____unchanged_portion_omitted_*

```
void
inj_defn_destroy(inj_defn_t *defn)
{
        if (defn->defn_name != NULL)
                inj_strfree(defn->defn_name);

        if (defn->defn_nvl != NULL)
```



```
        nvlist_free(defn->defn_nvl);

        inj_defn_destroy_memlist(inj_list_next(&defn->defn_members));
}
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   31665 Mon Feb 15 12:55:59 2016
new/usr/src/cmd/fm/fmtopo/common/fmtopo.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 238 static void
 239 print_prop_nameval(topo_hdl_t *thp, tnode_t *node, nvlist_t *nvl)
 240 {
 241         int err;
 242         topo_type_t type;
 243         char *tstr, *propn, buf[48], *factype;
 244         nvpair_t *pv_nvp;
 245         int i;
 246         uint_t nelem;

 248         if ((pv_nvp = nvlist_next_nvpair(nvl, NULL)) == NULL)
 249                 return;

 251         /* Print property name */
 252         if ((pv_nvp = nvlist_next_nvpair(nvl, NULL)) == NULL ||
 253             nvpair_name(pv_nvp) == NULL ||
 254             strcmp(TOPO_PROP_VAL_NAME, nvpair_name(pv_nvp)) != 0) {
 255                 (void) fprintf(stderr, "%s: malformed property name\n",
 256                     g_pname);
 257                 return;
 258         } else {
 259                 (void) nvpair_value_string(pv_nvp, &propn);
 260         }

 262         if ((pv_nvp = nvlist_next_nvpair(nvl, pv_nvp)) == NULL ||
 263             nvpair_name(pv_nvp) == NULL ||
 264             strcmp(nvpair_name(pv_nvp), TOPO_PROP_VAL_TYPE) != 0 ||
 265             nvpair_type(pv_nvp) != DATA_TYPE_UINT32)  {
 266                 (void) fprintf(stderr, "%s: malformed property type for %s\n",
 267                     g_pname, propn);
 268                 return;
 269         } else {
 270                 (void) nvpair_value_uint32(pv_nvp, (uint32_t *)&type);
 271         }

 273         switch (type) {
 274                 case TOPO_TYPE_BOOLEAN: tstr = "boolean"; break;
 275                 case TOPO_TYPE_INT32: tstr = "int32"; break;
 276                 case TOPO_TYPE_UINT32: tstr = "uint32"; break;
 277                 case TOPO_TYPE_INT64: tstr = "int64"; break;
 278                 case TOPO_TYPE_UINT64: tstr = "uint64"; break;
 279                 case TOPO_TYPE_DOUBLE: tstr = "double"; break;
 280                 case TOPO_TYPE_STRING: tstr = "string"; break;
 281                 case TOPO_TYPE_FMRI: tstr = "fmri"; break;
 282                 case TOPO_TYPE_INT32_ARRAY: tstr = "int32[]"; break;
 283                 case TOPO_TYPE_UINT32_ARRAY: tstr = "uint32[]"; break;
 284                 case TOPO_TYPE_INT64_ARRAY: tstr = "int64[]"; break;
 285                 case TOPO_TYPE_UINT64_ARRAY: tstr = "uint64[]"; break;
 286                 case TOPO_TYPE_STRING_ARRAY: tstr = "string[]"; break;
 287                 case TOPO_TYPE_FMRI_ARRAY: tstr = "fmri[]"; break;
 288                 default: tstr = "unknown type";
 289         }

 291         (void) printf("    %-17s %-8s ", propn, tstr);

 293         /*
 294          * Get property value
 295          */
 296         if (nvpair_name(pv_nvp) == NULL ||
```



```
 297             (pv_nvp = nvlist_next_nvpair(nvl, pv_nvp)) == NULL) {
 298                 (void) fprintf(stderr, "%s: malformed property value\n",
 299                     g_pname);
 300                 return;
 301         }

 303         switch (nvpair_type(pv_nvp)) {
 304                 case DATA_TYPE_INT32: {
 305                         int32_t val;
 306                         (void) nvpair_value_int32(pv_nvp, &val);
 307                         (void) printf(" %d", val);
 308                         break;
 309                 }
 310                 case DATA_TYPE_UINT32: {
 311                         uint32_t val, type;
 312                         char val_str[49];
 313                         nvlist_t *fac, *rsrc = NULL;

 315                         (void) nvpair_value_uint32(pv_nvp, &val);
 316                         if (node == NULL || topo_node_flags(node) !=
 317                             TOPO_NODE_FACILITY)
 318                                 goto uint32_def;

 320                         if (topo_node_resource(node, &rsrc, &err) != 0)
 321                                 goto uint32_def;

 323                         if (nvlist_lookup_nvlist(rsrc, "facility", &fac) != 0)
 324                                 goto uint32_def;

 326                         if (nvlist_lookup_string(fac, FM_FMRI_FACILITY_TYPE,
 327                             &factype) != 0)
 328                                 goto uint32_def;

 330                         nvlist_free(rsrc);
 331                         rsrc = NULL;

 333                         /*
 334                          * Special case code to do friendlier printing of
 335                          * facility node properties
 336                          */
 337                         if ((strcmp(propn, TOPO_FACILITY_TYPE) == 0) &&
 338                             (strcmp(factype, TOPO_FAC_TYPE_SENSOR) == 0)) {
 339                                 topo_sensor_type_name(val, val_str, 48);
 340                                 (void) printf(" 0x%x (%s)", val, val_str);
 341                                 break;
 342                         } else if ((strcmp(propn, TOPO_FACILITY_TYPE) == 0) &&
 343                             (strcmp(factype, TOPO_FAC_TYPE_INDICATOR) == 0)) {
 344                                 topo_led_type_name(val, val_str, 48);
 345                                 (void) printf(" 0x%x (%s)", val, val_str);
 346                                 break;
 347                         } else if (strcmp(propn, TOPO_SENSOR_UNITS) == 0) {
 348                                 topo_sensor_units_name(val, val_str, 48);
 349                                 (void) printf(" 0x%x (%s)", val, val_str);
 350                                 break;
 351                         } else if (strcmp(propn, TOPO_LED_MODE) == 0) {
 352                                 topo_led_state_name(val, val_str, 48);
 353                                 (void) printf(" 0x%x (%s)", val, val_str);
 354                                 break;
 355                         } else if ((strcmp(propn, TOPO_SENSOR_STATE) == 0) &&
 356                             (strcmp(factype, TOPO_FAC_TYPE_SENSOR) == 0)) {
 357                                 if (topo_prop_get_uint32(node,
 358                                     TOPO_PGROUP_FACILITY, TOPO_FACILITY_TYPE,
 359                                     &type, &err) != 0) {
 360                                         goto uint32_def;
 361                                 }
 362                                 topo_sensor_state_name(type, val, val_str, 48);
```

```
363                                 (void) printf(" 0x%x (%s)", val, val_str);
364                                 break;
365                         }
366 uint32_def:
367                         (void) printf(" 0x%x", val);
368                         if (rsrc != NULL)
368                         nvlist_free(rsrc);
369                         break;
370                 }
371                 case DATA_TYPE_INT64: {
372                         int64_t val;
373                         (void) nvpair_value_int64(pv_nvp, &val);
374                         (void) printf(" %lld", (longlong_t)val);
375                         break;
376                 }
377                 case DATA_TYPE_UINT64: {
378                         uint64_t val;
379                         (void) nvpair_value_uint64(pv_nvp, &val);
380                         (void) printf(" 0x%llx", (u_longlong_t)val);
381                         break;
382                 }
383                 case DATA_TYPE_DOUBLE: {
384                         double val;
385                         (void) nvpair_value_double(pv_nvp, &val);
386                         (void) printf(" %lf", (double)val);
387                         break;
388                 }
389                 case DATA_TYPE_STRING: {
390                         char *val;
391                         (void) nvpair_value_string(pv_nvp, &val);
392                         if (!opt_V && strlen(val) > 48) {
393                                 (void) snprintf(buf, 48, "%s...", val);
394                                 (void) printf(" %s", buf);
395                         } else {
396                                 (void) printf(" %s", val);
397                         }
398                         break;
399                 }
400                 case DATA_TYPE_NVLIST: {
401                         nvlist_t *val;
402                         char *fmri;
403                         (void) nvpair_value_nvlist(pv_nvp, &val);
404                         if (topo_fmri_nvl2str(thp, val, &fmri, &err) != 0) {
405                                 if (opt_V)
406                                         nvlist_print(stdout, nvl);
407                                 break;
408                         }

410                         if (!opt_V && strlen(fmri) > 48) {
411                                 (void) snprintf(buf, 48, "%s", fmri);
412                                 (void) snprintf(&buf[45], 4, "%s", DOTS);
413                                 (void) printf(" %s", buf);
414                         } else {
415                                 (void) printf(" %s", fmri);
416                         }

418                         topo_hdl_strfree(thp, fmri);
419                         break;
420                 }
421                 case DATA_TYPE_INT32_ARRAY: {
422                         int32_t *val;

424                         (void) nvpair_value_int32_array(pv_nvp, &val, &nelem);
425                         (void) printf(" [ ");
426                         for (i = 0; i < nelem; i++)
427                                 (void) printf("%d ", val[i]);
```



```
428                         (void) printf("]");
429                         break;
430                 }
431                 case DATA_TYPE_UINT32_ARRAY: {
432                         uint32_t *val;

434                         (void) nvpair_value_uint32_array(pv_nvp, &val, &nelem);
435                         (void) printf(" [ ");
436                         for (i = 0; i < nelem; i++)
437                                 (void) printf("%u ", val[i]);
438                         (void) printf("]");
439                         break;
440                 }
441                 case DATA_TYPE_INT64_ARRAY: {
442                         int64_t *val;

444                         (void) nvpair_value_int64_array(pv_nvp, &val, &nelem);
445                         (void) printf(" [ ");
446                         for (i = 0; i < nelem; i++)
447                                 (void) printf("%lld ", val[i]);
448                         (void) printf("]");
449                         break;
450                 }
451                 case DATA_TYPE_UINT64_ARRAY: {
452                         uint64_t *val;

454                         (void) nvpair_value_uint64_array(pv_nvp, &val, &nelem);
455                         (void) printf(" [ ");
456                         for (i = 0; i < nelem; i++)
457                                 (void) printf("%llu ", val[i]);
458                         (void) printf("]");
459                         break;
460                 }
461                 case DATA_TYPE_STRING_ARRAY: {
462                         char **val;

464                         (void) nvpair_value_string_array(pv_nvp, &val, &nelem);
465                         (void) printf(" [ ");
466                         for (i = 0; i < nelem; i++)
467                                 (void) printf("\"%s\" ", val[i]);
468                         (void) printf("]");
469                         break;
470                 }
471                 default:
472                         (void) fprintf(stderr, " unknown data type (%d)",
473                             nvpair_type(pv_nvp));
474                         break;
475                 }
476                 (void) printf("\n");
477 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   19114 Mon Feb 15 12:55:59 2016
new/usr/src/cmd/fm/modules/common/disk-monitor/diskmon_conf.c
patch cleanup
6659 nvlist_free(NULL) is a no-op
**********************************************************
```

```
/*
 * 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 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * Disk & Indicator Monitor configuration file support routines
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <errno.h>
#include <limits.h>
#include <pthread.h>

#include "disk_monitor.h"
#include "util.h"
#include "topo_gather.h"

extern log_class_t g_verbose;

const char *
hotplug_state_string(hotplug_state_t state)
{
        switch (state & ~HPS_FAULTED) {
        default:
        case HPS_UNKNOWN:
                return ("Unknown");
        case HPS_ABSENT:
                return ("Absent");
        case HPS_PRESENT:
                return ("Present");
        case HPS_CONFIGURED:
```



```
                return ("Configured");
        case HPS_UNCONFIGURED:
                return ("Unconfigured");
        }
}
______unchanged_portion_omitted_

void
diskmon_free(diskmon_t *dmp)
{
        diskmon_t *nextp;

        /* Free the whole list */
        while (dmp != NULL) {
                nextp = dmp->next;

                if (dmp->props)
                nvlist_free(dmp->props);
                if (dmp->location)
                        dstrfree(dmp->location);
                if (dmp->ind_list)
                        ind_free(dmp->ind_list);
                if (dmp->indrule_list)
                        indrule_free(dmp->indrule_list);
                if (dmp->app_props)
                nvlist_free(dmp->app_props);
                if (dmp->frup)
                        dmfru_free(dmp->frup);
                dfree(dmp, sizeof (diskmon_t));

                dmp = nextp;
        }
}
______unchanged_portion_omitted_
```

```
**********************************************************
   20810 Mon Feb 15 12:55:59 2016
new/usr/src/cmd/fm/modules/common/disk-monitor/topo_gather.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_


 461 static int
 462 topo_add_bay(topo_hdl_t *thp, tnode_t *node, walk_diskmon_t *wdp)
 463 {
 464         diskmon_t *target_diskp = wdp->target;
 465         nvlist_t        *nvlp = find_disk_monitor_private_pgroup(node);
 466         nvlist_t        *prop_nvlp;
 467         nvpair_t        *nvp = NULL;
 468         char            *prop_name, *prop_value;
 469 #define PNAME_MAX 128
 470         char            pname[PNAME_MAX];
 471         char            msgbuf[MAX_CONF_MSG_LEN];
 472         char            *indicator_name, *indicator_action;
 473         char            *indrule_states, *indrule_actions;
 474         int             err = 0, i;
 475         conf_err_t      conferr;
 476         boolean_t       conf_failure = B_FALSE;
 477         char            *unadj_physid = NULL;
 478         char            physid[MAXPATHLEN];
 479         char            *label;
 480         nvlist_t        *diskprops = NULL;
 481         char            *cstr = NULL;
 482         indicator_t     *indp = NULL;
 483         indrule_t       *indrp = NULL;
 484         void            *p;
 485         diskmon_t       *diskp;
 486         void            *ptr;

 488         /* No private properties -- just ignore the port */
 489         if (nvlp == NULL)
 490                 return (0);

 492         /*
 493          * Look for a diskmon based on this node's FMRI string.
 494          * Once a diskmon has been created, it's not re-created.  This is
 495          * essential for the times when the tree-walk is called after a
 496          * disk is inserted (or removed) -- in that case, the disk node
 497          * handler simply updates the FRU information in the diskmon.
 498          */
 499         if ((p = fmri2ptr(thp, node, &cstr, &err)) != NULL) {

 501                 diskp = (diskmon_t *)p;

 503                 /*
 504                  * Delete the FRU information from the diskmon.  If a disk
 505                  * is connected, its FRU information will be refreshed by
 506                  * the disk node code.
 507                  */
 508                 if (diskp->frup && (target_diskp == NULL ||
 509                     diskp == target_diskp)) {
 510                         dm_assert(pthread_mutex_lock(&diskp->fru_mutex) == 0);
 511                         dmfru_free(diskp->frup);
 512                         diskp->frup = NULL;
 513                         dm_assert(pthread_mutex_unlock(&diskp->fru_mutex) == 0);
 514                 }

 516                 wdp->pfmri = cstr;
 517                 nvlist_free(nvlp);
 518                 return (0);
```



```
 519         }

 521         /*
 522          * Determine the physical path to the attachment point
 523          */
 524         if (topo_prop_get_string(node, TOPO_PGROUP_IO,
 525             TOPO_IO_AP_PATH, &unadj_physid, &err) == 0) {

 527                 adjust_dynamic_ap(unadj_physid, physid);
 528                 topo_hdl_strfree(thp, unadj_physid);
 529         } else {

 531                 /* unadj_physid cannot have been allocated */
 532                 if (cstr)
 533                         dstrfree(cstr);
 534                 nvlist_free(nvlp);
 535                 return (-1);
 536         }

 538         /*
 539          */

 541         /*
 542          * Process the properties.  If we encounter a property that
 543          * is not an indicator name, action, or rule, add it to the
 544          * disk's props list.
 545          */

 547         /* Process indicators */
 548         i = 0;
 549         indicator_name = NULL;
 550         indicator_action = NULL;
 551         do {
 552                 if (indicator_name != NULL && indicator_action != NULL) {

 554                         if (topoprop_indicator_add(&indp, indicator_name,
 555                             indicator_action) != 0) {

 557                                 conf_failure = B_TRUE;
 558                         }

 560                         topo_hdl_strfree(thp, indicator_name);
 561                         topo_hdl_strfree(thp, indicator_action);
 562                 }

 564                 (void) snprintf(pname, PNAME_MAX, BAY_IND_NAME "-%d", i);
 565                 if (topo_prop_get_string(node, DISK_MONITOR_PROPERTIES,
 566                     pname, &indicator_name, &err) != 0)
 567                         break;

 569                 (void) snprintf(pname, PNAME_MAX, BAY_IND_ACTION "-%d", i);
 570                 if (topo_prop_get_string(node, DISK_MONITOR_PROPERTIES,
 571                     pname, &indicator_action, &err) != 0)
 572                         break;

 574                 i++;
 575         } while (!conf_failure && indicator_name != NULL &&
 576             indicator_action != NULL);

 578         if (!conf_failure && indp != NULL &&
 579             (conferr = check_inds(indp)) != E_NO_ERROR) {
 580                 conf_error_msg(conferr, msgbuf, MAX_CONF_MSG_LEN, NULL);
 581                 log_msg(MM_CONF, "%s: Not adding disk to list\n", msgbuf);
 582                 conf_failure = B_TRUE;
 583         }
```

```
585         /* Process state rules and indicator actions */
586         i = 0;
587         indrule_states = NULL;
588         indrule_actions = NULL;
589         do {
590                 if (indrule_states != NULL && indrule_actions != NULL) {
592                         if (topoprop_indrule_add(&indrp, indrule_states,
593                             indrule_actions) != 0) {
595                                 conf_failure = B_TRUE;
596                         }
598                         topo_hdl_strfree(thp, indrule_states);
599                         topo_hdl_strfree(thp, indrule_actions);
600                 }
602                 (void) snprintf(pname, PNAME_MAX, BAY_INDRULE_STATES "-%d", i);
603                 if (topo_prop_get_string(node, DISK_MONITOR_PROPERTIES,
604                     pname, &indrule_states, &err) != 0)
605                         break;
607                 (void) snprintf(pname, PNAME_MAX, BAY_INDRULE_ACTIONS "-%d",
608                     i);
609                 if (topo_prop_get_string(node, DISK_MONITOR_PROPERTIES,
610                     pname, &indrule_actions, &err) != 0)
611                         break;
613                 i++;
614         } while (!conf_failure && indrule_states != NULL &&
615             indrule_actions != NULL);
617         if (!conf_failure && indrp != NULL && indp != NULL &&
618             ((conferr = check_indrules(indrp, (state_transition_t **)&ptr))
619             != E_NO_ERROR ||
620             (conferr = check_consistent_ind_indrules(indp, indrp,
621             (ind_action_t **)&ptr)) != E_NO_ERROR)) {
623                 conf_error_msg(conferr, msgbuf, MAX_CONF_MSG_LEN, ptr);
624                 log_msg(MM_CONF, "%s: Not adding disk to list\n", msgbuf);
625                 conf_failure = B_TRUE;
627         }
629         /*
630          * Now collect miscellaneous properties.
631          * Each property is stored as an embedded nvlist named
632          * TOPO_PROP_VAL.  The property name is stored in the value for
633          * key=TOPO_PROP_VAL_NAME and the property's value is
634          * stored in the value for key=TOPO_PROP_VAL_VAL.  This is all
635          * necessary so we can subtractively decode the properties that
636          * we do not directly handle (so that these properties are added to
637          * the per-disk properties nvlist), increasing flexibility.
638          */
639         (void) nvlist_alloc(&diskprops, NV_UNIQUE_NAME, 0);
640         while ((nvp = nvlist_next_nvpair(nvlp, nvp)) != NULL) {
641                 /* Only care about embedded nvlists named TOPO_PROP_VAL */
642                 if (nvpair_type(nvp) != DATA_TYPE_NVLIST ||
643                     strcmp(nvpair_name(nvp), TOPO_PROP_VAL) != 0 ||
644                     nvpair_value_nvlist(nvp, &prop_nvlp) != 0)
645                         continue;
647                 if (nonunique_nvlist_lookup_string(prop_nvlp,
648                     TOPO_PROP_VAL_NAME, &prop_name) != 0)
649                         continue;
```



```
651                 /* Filter out indicator properties */
652                 if (strstr(prop_name, BAY_IND_NAME) != NULL ||
653                     strstr(prop_name, BAY_IND_ACTION) != NULL ||
654                     strstr(prop_name, BAY_INDRULE_STATES) != NULL ||
655                     strstr(prop_name, BAY_INDRULE_ACTIONS) != NULL)
656                         continue;
658                 if (nonunique_nvlist_lookup_string(prop_nvlp, TOPO_PROP_VAL_VAL,
659                     &prop_value) != 0)
660                         continue;
662                 /* Add the property to the disk's prop list: */
663                 if (nvlist_add_string(diskprops, prop_name, prop_value) != 0)
664                         log_msg(MM_TOPO,
665                             "Could not add disk property '%s' with "
666                             "value '%s'\n", prop_name, prop_value);
667         }
669         nvlist_free(nvlp);
671         if (cstr != NULL) {
672                 namevalpr_t nvpr;
673                 nvlist_t *dmap_nvl;
675                 nvpr.name = DISK_AP_PROP_APID;
676                 nvpr.value = strncmp(physid, "/devices", 8) == 0 ?
677                     (physid + 8) : physid;
679                 /*
680                  * Set the diskmon's location to the value in this port's label.
681                  * If there's a disk plugged in, the location will be updated
682                  * to be the disk label (e.g. HD_ID_00).  Until a disk is
683                  * inserted, though, there won't be a disk libtopo node
684                  * created.
685                  */
687                 /* Pass physid without the leading "/devices": */
688                 dmap_nvl = namevalpr_to_nvlist(&nvpr);
690                 diskp = new_diskmon(dmap_nvl, indp, indrp, diskprops);
692                 if (topo_node_label(node, &label, &err) == 0) {
693                         diskp->location = dstrdup(label);
694                         topo_hdl_strfree(thp, label);
695                 } else
696                         diskp->location = dstrdup("unknown location");
698                 if (!conf_failure && diskp != NULL) {
699                         /* Add this diskmon to the disk list */
700                         cfgdata_add_diskmon(config_data, diskp);
701                         if (nvlist_add_uint64(g_topo2diskmon, cstr,
702                             (uint64_t)(uintptr_t)diskp) != 0) {
703                                 log_msg(MM_TOPO,
704                                     "Could not add pointer to nvlist "
705                                     "for '%s'!\n", cstr);
706                         }
707                 } else if (diskp != NULL) {
708                         diskmon_free(diskp);
709                 } else {
710                         if (dmap_nvl)
710                         nvlist_free(dmap_nvl);
711                         if (indp)
712                                 ind_free(indp);
713                         if (indrp)
714                                 indrule_free(indrp);
716                         if (diskprops)
```

```
715                         nvlist_free(diskprops);
716                 }

718                 wdp->pfmri = cstr;
719         }


722         return (0);
723 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  113638 Mon Feb 15 12:56:00 2016
new/usr/src/cmd/fm/modules/common/eversholt/fme.c
patch tsoome-feedback
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

1624 static void
1625 fme_receive_report(fmd_hdl_t *hdl, fmd_event_t *ffep,
1626     const char *eventstring, const struct ipath *ipp, nvlist_t *nvl)
1627 {
1628         struct event *ep;
1629         struct fme *fmep = NULL;
1630         struct fme *ofmep = NULL;
1631         struct fme *cfmep, *svfmep;
1632         int matched = 0;
1633         nvlist_t *defect;
1634         fmd_case_t *fmcase;
1635         char *reason;

1637         out(O_ALTFP|O_NONL, "fme_receive_report: ");
1638         ipath_print(O_ALTFP|O_NONL, eventstring, ipp);
1639         out(O_ALTFP|O_STAMP, NULL);

1641         /* decide which FME it goes to */
1642         for (fmep = FMElist; fmep; fmep = fmep->next) {
1643                 int prev_verbose;
1644                 unsigned long long my_delay = TIMEVAL_EVENTUALLY;
1645                 enum fme_state state;
1646                 nvlist_t *pre_peek_nvp = NULL;

1648                 if (fmep->overflow) {
1649                         if (!(fmd_case_closed(fmep->hdl, fmep->fmcase)))
1650                                 ofmep = fmep;

1652                         continue;
1653                 }

1655                 /*
1656                  * ignore solved or closed cases
1657                  */
1658                 if (fmep->posted_suspects ||
1659                     fmd_case_solved(fmep->hdl, fmep->fmcase) ||
1660                     fmd_case_closed(fmep->hdl, fmep->fmcase))
1661                         continue;

1663                 /* look up event in event tree for this FME */
1664                 if ((ep = itree_lookup(fmep->eventtree,
1665                     eventstring, ipp)) == NULL)
1666                         continue;

1668                 /* note observation */
1669                 fmep->ecurrent = ep;
1670                 if (ep->count++ == 0) {
1671                         /* link it into list of observations seen */
1672                         ep->observations = fmep->observations;
1673                         fmep->observations = ep;
1674                         ep->nvp = evnv_dupnvl(nvl);
1675                 } else {
1676                         /* use new payload values for peek */
1677                         pre_peek_nvp = ep->nvp;
1678                         ep->nvp = evnv_dupnvl(nvl);
1679                 }

1681                 /* tell hypothesise() not to mess with suspect list */
```



```
1682                 fmep->peek = 1;

1684                 /* don't want this to be verbose (unless Debug is set) */
1685                 prev_verbose = Verbose;
1686                 if (Debug == 0)
1687                         Verbose = 0;

1689                 lut_walk(fmep->eventtree, (lut_cb)clear_arrows, (void *)fmep);
1690                 state = hypothesise(fmep, fmep->e0, fmep->ull, &my_delay);

1692                 fmep->peek = 0;

1694                 /* put verbose flag back */
1695                 Verbose = prev_verbose;

1697                 if (state != FME_DISPROVED) {
1698                         /* found an FME that explains the ereport */
1699                         matched++;
1700                         out(O_ALTFP|O_NONL, "[");
1701                         ipath_print(O_ALTFP|O_NONL, eventstring, ipp);
1702                         out(O_ALTFP, " explained by FME%d]", fmep->id);

1704                         if (pre_peek_nvp)
1704                         nvlist_free(pre_peek_nvp);

1706                         if (ep->count == 1)
1707                                 serialize_observation(fmep, eventstring, ipp);

1709                         if (ffep) {
1710                                 fmd_case_add_ereport(hdl, fmep->fmcase, ffep);
1711                                 ep->ffep = ffep;
1712                         }

1714                         stats_counter_bump(fmep->Rcount);

1716                         /* re-eval FME */
1717                         fme_eval(fmep, ffep);
1718                 } else {

1720                         /* not a match, undo noting of observation */
1721                         fmep->ecurrent = NULL;
1722                         if (--ep->count == 0) {
1723                                 /* unlink it from observations */
1724                                 fmep->observations = ep->observations;
1725                                 ep->observations = NULL;
1726                                 nvlist_free(ep->nvp);
1727                                 ep->nvp = NULL;
1728                         } else {
1729                                 nvlist_free(ep->nvp);
1730                                 ep->nvp = pre_peek_nvp;
1731                         }
1732                 }
1733         }

1735         if (matched)
1736                 return; /* explained by at least one existing FME */

1738         /* clean up closed fmes */
1739         cfmep = ClosedFMEs;
1740         while (cfmep != NULL) {
1741                 svfmep = cfmep->next;
1742                 destroy_fme(cfmep);
1743                 cfmep = svfmep;
1744         }
1745         ClosedFMEs = NULL;
```

```
1747         if (ofmep) {
1748                 out(O_ALTFP|O_NONL, "[");
1749                 ipath_print(O_ALTFP|O_NONL, eventstring, ipp);
1750                 out(O_ALTFP, " ADDING TO OVERFLOW FME]");
1751                 if (ffep)
1752                         fmd_case_add_ereport(hdl, ofmep->fmcase, ffep);

1754                 return;

1756         } else if (Max_fme && (Open_fme_count >= Max_fme)) {
1757                 out(O_ALTFP|O_NONL, "[");
1758                 ipath_print(O_ALTFP|O_NONL, eventstring, ipp);
1759                 out(O_ALTFP, " MAX OPEN FME REACHED]");

1761                 fmcase = fmd_case_open(hdl, NULL);

1763                 /* Create overflow fme */
1764                 if ((fmep = newfme(eventstring, ipp, hdl, fmcase, ffep,
1765                     nvl)) == NULL) {
1766                         out(O_ALTFP|O_NONL, "[");
1767                         ipath_print(O_ALTFP|O_NONL, eventstring, ipp);
1768                         out(O_ALTFP, " CANNOT OPEN OVERFLOW FME]");
1769                         return;
1770                 }

1772                 Open_fme_count++;

1774                 init_fme_bufs(fmep);
1775                 fmep->overflow = B_TRUE;

1777                 if (ffep)
1778                         fmd_case_add_ereport(hdl, fmep->fmcase, ffep);

1780                 Undiag_reason = UD_VAL_MAXFME;
1781                 defect = fmd_nvl_create_fault(hdl,
1782                     undiag_2defect_str(Undiag_reason), 100, NULL, NULL, NULL);
1783                 reason = undiag_2reason_str(Undiag_reason, NULL);
1784                 (void) nvlist_add_string(defect, UNDIAG_REASON, reason);
1785                 FREE(reason);
1786                 fmd_case_add_suspect(hdl, fmep->fmcase, defect);
1787                 fmd_case_solve(hdl, fmep->fmcase);
1788                 Undiag_reason = UD_VAL_UNKNOWN;
1789                 return;
1790         }

1792         /* open a case */
1793         fmcase = fmd_case_open(hdl, NULL);

1795         /* start a new FME */
1796         if ((fmep = newfme(eventstring, ipp, hdl, fmcase, ffep, nvl)) == NULL) {
1797                 out(O_ALTFP|O_NONL, "[");
1798                 ipath_print(O_ALTFP|O_NONL, eventstring, ipp);
1799                 out(O_ALTFP, " CANNOT DIAGNOSE]");
1800                 return;
1801         }

1803         Open_fme_count++;

1805         init_fme_bufs(fmep);

1807         out(O_ALTFP|O_NONL, "[");
1808         ipath_print(O_ALTFP|O_NONL, eventstring, ipp);
1809         out(O_ALTFP, " created FME%d, case %s]", fmep->id,
1810             fmd_case_uuid(hdl, fmep->fmcase));

1812         ep = fmep->e0;
```



```
1813         ASSERT(ep != NULL);

1815         /* note observation */
1816         fmep->ecurrent = ep;
1817         if (ep->count++ == 0) {
1818                 /* link it into list of observations seen */
1819                 ep->observations = fmep->observations;
1820                 fmep->observations = ep;
1821                 ep->nvp = evnv_dupnvl(nvl);
1822                 serialize_observation(fmep, eventstring, ipp);
1823         } else {
1824                 /* new payload overrides any previous */
1825                 nvlist_free(ep->nvp);
1826                 ep->nvp = evnv_dupnvl(nvl);
1827         }

1829         stats_counter_bump(fmep->Rcount);

1831         if (ffep) {
1832                 fmd_case_add_ereport(hdl, fmep->fmcase, ffep);
1833                 fmd_case_setprincipal(hdl, fmep->fmcase, ffep);
1834                 fmep->e0r = ffep;
1835                 ep->ffep = ffep;
1836         }

1838         /* give the diagnosis algorithm a shot at the new FME state */
1839         fme_eval(fmep, ffep);
1840 }
_____unchanged_portion_omitted_

1967 static nvlist_t *
1968 node2fmri(struct node *n)
1969 {
1970         nvlist_t **pa, *f, *p;
1971         struct node *nc;
1972         uint_t depth = 0;
1973         char *numstr, *nullbyte;
1974         char *failure;
1975         int err, i;

1977         /* XXX do we need to be able to handle a non-T_NAME node? */
1978         if (n == NULL || n->t != T_NAME)
1979                 return (NULL);

1981         for (nc = n; nc != NULL; nc = nc->u.name.next) {
1982                 if (nc->u.name.child == NULL || nc->u.name.child->t != T_NUM)
1983                         break;
1984                 depth++;
1985         }

1987         if (nc != NULL) {
1988                 /* We bailed early, something went wrong */
1989                 return (NULL);
1990         }

1992         if ((err = nvlist_xalloc(&f, NV_UNIQUE_NAME, &Eft_nv_hdl)) != 0)
1993                 out(O_DIE|O_SYS, "alloc of fmri nvl failed");
1994         pa = alloca(depth * sizeof (nvlist_t *));
1995         for (i = 0; i < depth; i++)
1996                 pa[i] = NULL;

1998         err = nvlist_add_string(f, FM_FMRI_SCHEME, FM_FMRI_SCHEME_HC);
1999         err |= nvlist_add_uint8(f, FM_VERSION, FM_HC_SCHEME_VERSION);
2000         err |= nvlist_add_string(f, FM_FMRI_HC_ROOT, "");
2001         err |= nvlist_add_uint32(f, FM_FMRI_HC_LIST_SZ, depth);
2002         if (err != 0) {
```

```
2003                    failure = "basic construction of FMRI failed";
2004                    goto boom;
2005            }

2007            numbuf[MAXDIGITIDX] = '\0';
2008            nullbyte = &numbuf[MAXDIGITIDX];
2009            i = 0;

2011            for (nc = n; nc != NULL; nc = nc->u.name.next) {
2012                    err = nvlist_xalloc(&p, NV_UNIQUE_NAME, &Eft_nv_hdl);
2013                    if (err != 0) {
2014                            failure = "alloc of an hc-pair failed";
2015                            goto boom;
2016                    }
2017                    err = nvlist_add_string(p, FM_FMRI_HC_NAME, nc->u.name.s);
2018                    numstr = ulltostr(nc->u.name.child->u.ull, nullbyte);
2019                    err |= nvlist_add_string(p, FM_FMRI_HC_ID, numstr);
2020                    if (err != 0) {
2021                            failure = "construction of an hc-pair failed";
2022                            goto boom;
2023                    }
2024                    pa[i++] = p;
2025            }

2027            err = nvlist_add_nvlist_array(f, FM_FMRI_HC_LIST, pa, depth);
2028            if (err == 0) {
2029                    for (i = 0; i < depth; i++)
2031                            if (pa[i] != NULL)
2030                            nvlist_free(pa[i]);
2031                    return (f);
2032            }
2033            failure = "addition of hc-pair array to FMRI failed";

2035 boom:
2036            for (i = 0; i < depth; i++)
2039                    if (pa[i] != NULL)
2037                    nvlist_free(pa[i]);
2038            nvlist_free(f);
2039            out(O_DIE, "%s", failure);
2040            /*NOTREACHED*/
2041            return (NULL);
2042 }
```

_____unchanged_portion_omitted_

```
2050 static nvlist_t *
2051 ipath2fmri(struct ipath *ipath)
2052 {
2053            nvlist_t **pa, *f, *p;
2054            uint_t depth = 0;
2055            char *numstr, *nullbyte;
2056            char *failure;
2057            int err, i;
2058            struct ipath *ipp;

2060            for (ipp = ipath; ipp->s != NULL; ipp++)
2061                    depth++;

2063            if ((err = nvlist_xalloc(&f, NV_UNIQUE_NAME, &Eft_nv_hdl)) != 0)
2064                    out(O_DIE|O_SYS, "alloc of fmri nvl failed");
2065            pa = alloca(depth * sizeof (nvlist_t *));
2066            for (i = 0; i < depth; i++)
2067                    pa[i] = NULL;

2069            err = nvlist_add_string(f, FM_FMRI_SCHEME, FM_FMRI_SCHEME_HC);
2070            err |= nvlist_add_uint8(f, FM_VERSION, FM_HC_SCHEME_VERSION);
2071            err |= nvlist_add_string(f, FM_FMRI_HC_ROOT, "");
```



```
2072            err |= nvlist_add_uint32(f, FM_FMRI_HC_LIST_SZ, depth);
2073            if (err != 0) {
2074                    failure = "basic construction of FMRI failed";
2075                    goto boom;
2076            }

2078            numbuf[MAXDIGITIDX] = '\0';
2079            nullbyte = &numbuf[MAXDIGITIDX];
2080            i = 0;

2082            for (ipp = ipath; ipp->s != NULL; ipp++) {
2083                    err = nvlist_xalloc(&p, NV_UNIQUE_NAME, &Eft_nv_hdl);
2084                    if (err != 0) {
2085                            failure = "alloc of an hc-pair failed";
2086                            goto boom;
2087                    }
2088                    err = nvlist_add_string(p, FM_FMRI_HC_NAME, ipp->s);
2089                    numstr = ulltostr(ipp->i, nullbyte);
2090                    err |= nvlist_add_string(p, FM_FMRI_HC_ID, numstr);
2091                    if (err != 0) {
2092                            failure = "construction of an hc-pair failed";
2093                            goto boom;
2094                    }
2095                    pa[i++] = p;
2096            }

2098            err = nvlist_add_nvlist_array(f, FM_FMRI_HC_LIST, pa, depth);
2099            if (err == 0) {
2100                    for (i = 0; i < depth; i++)
2104                            if (pa[i] != NULL)
2101                            nvlist_free(pa[i]);
2102                    return (f);
2103            }
2104            failure = "addition of hc-pair array to FMRI failed";

2106 boom:
2107            for (i = 0; i < depth; i++)
2112                    if (pa[i] != NULL)
2108                    nvlist_free(pa[i]);
2109            nvlist_free(f);
2110            out(O_DIE, "%s", failure);
2111            /*NOTREACHED*/
2112            return (NULL);
2113 }
```

_____unchanged_portion_omitted_

```
2130 static void publish_suspects(struct fme *fmep, struct rsl *srl);

2132 /*
2133  *  rslfree -- free internal members of struct rsl not expected to be
2134  *      freed elsewhere.
2135  */
2136 static void
2137 rslfree(struct rsl *freeme)
2138 {
2144            if (freeme->asru != NULL)
2139            nvlist_free(freeme->asru);
2146            if (freeme->fru != NULL)
2140            nvlist_free(freeme->fru);
2141            if (freeme->rsrc != freeme->asru)
2148            if (freeme->rsrc != NULL && freeme->rsrc != freeme->asru)
2142                    nvlist_free(freeme->rsrc);
2143 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   57754 Mon Feb 15 12:56:00 2016
new/usr/src/cmd/fm/modules/common/eversholt/itree.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

1694 /*ARGSUSED*/
1695 static void
1696 itree_destructor(void *left, void *right, void *arg)
1697 {
1698         struct event *ep = (struct event *)right;
1699         struct bubble *nextbub, *bub;

1701         /* Free the properties */
1702         if (ep->props)
1703                 lut_free(ep->props, instances_destructor, NULL);

1705         /* Free the payload properties */
1706         if (ep->payloadprops)
1707                 lut_free(ep->payloadprops, payloadprops_destructor, NULL);

1709         /* Free the serd properties */
1710         if (ep->serdprops)
1711                 lut_free(ep->serdprops, serdprops_destructor, NULL);

1713         /* Free my bubbles */
1714         for (bub = ep->bubbles; bub != NULL; ) {
1715                 nextbub = bub->next;
1716                 /*
1717                  * Free arrows if they are FROM me.  Free arrowlists on
1718                  * other types of bubbles (but not the attached arrows,
1719                  * which will be freed when we free the originating
1720                  * bubble.
1721                  */
1722                 if (bub->t == B_FROM)
1723                         itree_free_arrowlists(bub, 1);
1724                 else
1725                         itree_free_arrowlists(bub, 0);
1726                 itree_free_bubble(bub);
1727                 bub = nextbub;
1728         }

1730         if (ep->nvp != NULL)
1730         nvlist_free(ep->nvp);
1731         alloc_xfree(ep, sizeof (*ep));
1732 }

1734 /*ARGSUSED*/
1735 static void
1736 itree_pruner(void *left, void *right, void *arg)
1737 {
1738         struct event *ep = (struct event *)right;
1739         struct bubble *nextbub, *bub;

1741         if (ep->keep_in_tree)
1742                 return;

1744         /* Free the properties */
1745         lut_free(ep->props, instances_destructor, NULL);

1747         /* Free the payload properties */
1748         lut_free(ep->payloadprops, payloadprops_destructor, NULL);

1750         /* Free the serd properties */
1751         lut_free(ep->serdprops, serdprops_destructor, NULL);
```



```
1753         /* Free my bubbles */
1754         for (bub = ep->bubbles; bub != NULL; ) {
1755                 nextbub = bub->next;
1756                 itree_prune_arrowlists(bub);
1757                 itree_free_bubble(bub);
1758                 bub = nextbub;
1759         }

1762         if (ep->nvp != NULL)
1761         nvlist_free(ep->nvp);
1762         ep->props = NULL;
1763         ep->payloadprops = NULL;
1764         ep->serdprops = NULL;
1765         ep->bubbles = NULL;
1766         ep->nvp = NULL;
1767 }
______unchanged_portion_omitted_
```

```
**********************************************************
   17480 Mon Feb 15 12:56:00 2016
new/usr/src/cmd/fm/modules/common/ext-event-transport/fmevt_inbound.c
patch tsoome-feedback
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 201 /*
 202  * Construct a "sw" scheme detector FMRI.
 203  *
 204  * We make no use of priv or pri.
 205  */
 206 /*ARGSUSED3*/
 207 static nvlist_t *
 208 fmevt_detector(nvlist_t *attr, char *ruleset, int user, int priv,
 209     fmev_pri_t pri)
 210 {
 211         char buf[FMEV_MAX_RULESET_LEN + 1];
 212         char *ns, *subsys;
 213         nvlist_t *obj, *dtcr, *site, *ctxt;
 214         char *execname = NULL;
 215         int32_t i32;
 216         int64_t i64;
 217         int err = 0;
 218         char *str;

 220         (void) strncpy(buf, ruleset, sizeof (buf));
 221         if (!fmevt_rs_burst(NULL, buf, &ns, &subsys, B_FALSE))
 222                 return (NULL);

 224         obj = fmd_nvl_alloc(fmevt_hdl, FMD_SLEEP);
 225         dtcr = fmd_nvl_alloc(fmevt_hdl, FMD_SLEEP);
 226         site = fmd_nvl_alloc(fmevt_hdl, FMD_SLEEP);
 227         ctxt = fmd_nvl_alloc(fmevt_hdl, FMD_SLEEP);

 229         if (obj == NULL || dtcr == NULL || site == NULL || ctxt == NULL) {
 230                 err++;
 231                 goto done;
 232         }

 234         /*
 235          * Build up 'object' nvlist.
 236          */
 237         if (nvlist_lookup_string(attr, "__fmev_execname", &execname) == 0)
 238                 err += nvlist_add_string(obj, FM_FMRI_SW_OBJ_PATH, execname);

 240         /*
 241          * Build up 'site' nvlist.  We should have source file and line
 242          * number and, if the producer was compiled with C99, function name.
 243          */
 244         if (nvlist_lookup_string(attr, "__fmev_file", &str) == 0) {
 245                 err += nvlist_add_string(site, FM_FMRI_SW_SITE_FILE, str);
 246                 (void) nvlist_remove(attr, "__fmev_file", DATA_TYPE_STRING);
 247         }

 249         if (nvlist_lookup_string(attr, "__fmev_func", &str) == 0) {
 250                 err += nvlist_add_string(site, FM_FMRI_SW_SITE_FUNC, str);
 251                 (void) nvlist_remove(attr, "__fmev_func", DATA_TYPE_STRING);
 252         }

 254         if (nvlist_lookup_int64(attr, "__fmev_line", &i64) == 0) {
 255                 err += nvlist_add_int64(site, FM_FMRI_SW_SITE_LINE, i64);
 256                 (void) nvlist_remove(attr, "__fmev_line", DATA_TYPE_INT64);
 257         }
```



```
 259         /*
 260          * Build up 'context' nvlist.  We do not include contract id at
 261          * this time.
 262          */

 264         err += nvlist_add_string(ctxt, FM_FMRI_SW_CTXT_ORIGIN,
 265             user ? "userland" : "kernel");

 267         if (execname) {
 268                 err += nvlist_add_string(ctxt, FM_FMRI_SW_CTXT_EXECNAME,
 269                     execname);
 270                 (void) nvlist_remove(attr, "__fmev_execname", DATA_TYPE_STRING);
 271         }

 273         if (nvlist_lookup_int32(attr, "__fmev_pid", &i32) == 0) {
 274                 err += nvlist_add_int32(ctxt, FM_FMRI_SW_CTXT_PID, i32);
 275                 (void) nvlist_remove(attr, "__fmev_pid", DATA_TYPE_INT32);
 276         }

 278         if (!isglobalzone)
 279                 err += nvlist_add_string(ctxt, FM_FMRI_SW_CTXT_ZONE, zonename);

 281         /* Put it all together */

 283         err += nvlist_add_uint8(dtcr, FM_VERSION, SW_SCHEME_VERSION0);
 284         err += nvlist_add_string(dtcr, FM_FMRI_SCHEME, FM_FMRI_SCHEME_SW);
 285         err += nvlist_add_nvlist(dtcr, FM_FMRI_SW_OBJ, obj);
 286         err += nvlist_add_nvlist(dtcr, FM_FMRI_SW_SITE, site);
 287         err += nvlist_add_nvlist(dtcr, FM_FMRI_SW_CTXT, ctxt);

 289 done:
 290         if (obj != NULL)
 290         nvlist_free(obj);
 292         if (site != NULL)
 291         nvlist_free(site);
 294         if (ctxt != NULL)
 292         nvlist_free(ctxt);

 294         if (err == 0) {
 295                 return (dtcr);
 296         } else {
 297                 nvlist_free(dtcr);
 298                 return (NULL);
 299         }
 300 }
______unchanged_portion_omitted_

 488 static int
 489 fmevt_cb(sysevent_t *sep, void *arg)
 490 {
 491         char *ruleset = NULL, *rawclass, *rawsubclass;
 492         uint32_t cbarg = (uint32_t)arg;
 493         nvlist_t *rawattr = NULL;
 494         struct fmevt_ppargs ea;
 495         nvlist_t *dtcr;
 496         int user, priv;
 497         fmev_pri_t pri;

 499         BUMPSTAT(raw_callbacks);

 501         if (cbarg & ~CBF_ALL)
 502                 fmd_hdl_abort(fmevt_hdl, "event receipt callback with "
 503                     "invalid flags\n");

 505         user = (cbarg & CBF_USER) != 0;
 506         priv = (cbarg & CBF_PRIV) != 0;
```

```
507         pri = (cbarg & CBF_HV ? FMEV_HIPRI : FMEV_LOPRI);

509         (void) pthread_mutex_lock(&fmevt_lock);

511         if (fmevt_exiting) {
512                 while (fmevt_xprt_refcnt > 0)
513                         (void) pthread_cond_wait(&fmevt_cv, &fmevt_lock);
514                 (void) pthread_mutex_unlock(&fmevt_lock);
515                 return (0);     /* discard event */
516         }

518         fmevt_xprt_refcnt++;
519         (void) pthread_mutex_unlock(&fmevt_lock);

521         ruleset = sysevent_get_vendor_name(sep);        /* must free */
522         rawclass = sysevent_get_class_name(sep);        /* valid with sep */
523         rawsubclass = sysevent_get_subclass_name(sep);  /* valid with sep */

525         if (sysevent_get_attr_list(sep, &rawattr) != 0) {
526                 BUMPSTAT(raw_noattrlist);
527                 goto done;
528         }

530         if ((dtcr = fmevt_detector(rawattr, ruleset, user, priv,
531             pri)) == NULL) {
532                 BUMPSTAT(raw_nodetector);
533                 goto done;
534         }

536         ea.pp_rawclass = rawclass;
537         ea.pp_rawsubclass = rawsubclass;
538         sysevent_get_time(sep, &ea.pp_hrt);
539         ea.pp_user = user;
540         ea.pp_priv = priv;
541         ea.pp_pri = pri;

543         fmevt_postprocess(ruleset, dtcr, rawattr, &ea);
544         nvlist_free(dtcr);
545 done:
546         (void) pthread_mutex_lock(&fmevt_lock);

548         if (--fmevt_xprt_refcnt == 0 && fmevt_exiting)
549                 (void) pthread_cond_broadcast(&fmevt_cv);

551         (void) pthread_mutex_unlock(&fmevt_lock);

553         if (ruleset)
554                 free(ruleset);

559         if (rawattr)
556         nvlist_free(rawattr);

558         return (0);     /* in all cases consider the event delivered */
559 }
```

*_____unchanged_portion_omitted_*

****
30747 Mon Feb 15 12:56:00 2016
new/usr/src/cmd/fm/modules/common/ip-transport/ip.c
patch tsoome-feedback
****
_____***unchanged_portion_omitted_***

```
1051 void
1052 _fmd_fini(fmd_hdl_t *hdl)
1053 {
1054         ip_quit++; /* set quit flag before signalling auxiliary threads */

1056         while (ip_xps != NULL)
1057                 ip_xprt_destroy(ip_xps);

1059         if (ip_auth != NULL)
1059         nvlist_free(ip_auth);

1061         ip_addr_cleanup();

1063         if (ip_domain_name != NULL)
1064                 fmd_prop_free_string(ip_hdl, ip_domain_name);

1066         fmd_hdl_unregister(hdl);
1067 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   18796 Mon Feb 15 12:56:00 2016
new/usr/src/cmd/fm/modules/common/sw-diag-response/subsidiary/panic/panic_diag.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 169 #define BUMPSTAT(stat)          swde_panic_stats.stat.fmds_value.ui64++

 171 static nvlist_t *
 172 panic_sw_fmri(fmd_hdl_t *hdl, char *object)
 173 {
 174         nvlist_t *fmri;
 175         nvlist_t *sw_obj;
 176         int err = 0;

 178         fmri = fmd_nvl_alloc(hdl, FMD_SLEEP);
 179         err |= nvlist_add_uint8(fmri, FM_VERSION, FM_SW_SCHEME_VERSION);
 180         err |= nvlist_add_string(fmri, FM_FMRI_SCHEME, FM_FMRI_SCHEME_SW);

 182         sw_obj = fmd_nvl_alloc(hdl, FMD_SLEEP);
 183         err |= nvlist_add_string(sw_obj, FM_FMRI_SW_OBJ_PATH, object);
 184         err |= nvlist_add_nvlist(fmri, FM_FMRI_SW_OBJ, sw_obj);
 185         if (sw_obj)
 185         nvlist_free(sw_obj);
 186         if (!err)
 187                 return (fmri);
 188         else
 189                 return (0);
 190 }
______unchanged_portion_omitted_
```

****
13132 Mon Feb 15 12:56:00 2016
new/usr/src/cmd/fm/modules/common/syslog-msgs/syslog.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
 178 static void
 179 free_notify_prefs(fmd_hdl_t *hdl, nvlist_t **prefs, uint_t nprefs)
 180 {
 181         int i;

 183         for (i = 0; i < nprefs; i++) {
 184                 if (prefs[i])
 184                 nvlist_free(prefs[i]);
 185         }

 187         fmd_hdl_free(hdl, prefs, sizeof (nvlist_t *) * nprefs);
 188 }
```

_____***unchanged_portion_omitted_***

****
13193 Mon Feb 15 12:56:00 2016
new/usr/src/cmd/fm/modules/sun4/cpumem-diagnosis/cmd_dimm.c
patch tsoome-feedback
****
_____***unchanged_portion_omitted_***

```
 78 nvlist_t *
 79 cmd_dimm_create_fault(fmd_hdl_t *hdl, cmd_dimm_t *dimm, const char *fltnm,
 80     uint_t cert)
 81 {
 82 #ifdef sun4v
 83         nvlist_t *flt, *nvlfru;
 84         /*
 85          * Do NOT issue hc scheme FRU FMRIs for ultraSPARC-T1 platforms.
 86          * The SP will misinterpret the FRU. Instead, reuse the ASRU FMRI
 87          *
 88          * Use the BR string as a distinguisher. BR (branch) is only
 89          * present in ultraSPARC-T2/T2plus DIMM unums
 90          */
 91         if (strstr(dimm->dimm_unum, "BR") == NULL) {
 92                 flt = cmd_nvl_create_fault(hdl, fltnm, cert,
 93                     dimm->dimm_asru_nvl, dimm->dimm_asru_nvl, NULL);
 94         } else {
 95                 nvlfru = cmd_mem2hc(hdl, dimm->dimm_asru_nvl);
 96                 flt = cmd_nvl_create_fault(hdl, fltnm, cert,
 97                     dimm->dimm_asru_nvl, nvlfru, NULL);
 98                 if (nvlfru != NULL)
 98                 nvlist_free(nvlfru);
 99         }
100         return (cmd_fault_add_location(hdl, flt, dimm->dimm_unum));
101 #else
102         return (cmd_nvl_create_fault(hdl, fltnm, cert, dimm->dimm_asru_nvl,
103             dimm->dimm_asru_nvl, NULL));
104 #endif /* sun4v */
105 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    2870 Mon Feb 15 12:56:01 2016
new/usr/src/cmd/fm/modules/sun4/cpumem-diagnosis/cmd_pageerr.c
patch tsoome-feedback
**********************************************************
```

```
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Fault-handling routines for page retirement faults
 */

#include <cmd_page.h>
#include <cmd.h>
#include <cmd_mem.h>

#include <errno.h>
#include <string.h>
#include <fm/fmd_api.h>
#include <sys/fm/protocol.h>
#ifdef sun4v
#include <cmd_hc_sun4v.h>
#include <cmd_dimm.h>
#endif

void
cmd_page_fault(fmd_hdl_t *hdl, nvlist_t *modasru, nvlist_t *modfru,
    fmd_event_t *ep, uint64_t afar)
{
        cmd_page_t *page = NULL;
        const char *uuid;
        nvlist_t *flt;
#ifdef sun4v
        nvlist_t *nvlfru;
#endif

        page = cmd_page_lookup(afar);
        if (page != NULL) {
                /*
                 * If the page has already been retired then *page
                 * would have been freed and recreated. Thus the
                 * flag would be 0x0 - check to see if the page
                 * is unusable (retired).
                 */
```



```
                if (page->page_flags & CMD_MEM_F_FAULTING ||
                    fmd_nvl_fmri_unusable(hdl, page->page_asru_nvl)) {
                        /* Page already faulted, don't fault again. */
                        page->page_flags |= CMD_MEM_F_FAULTING;
                        return;
                }
        } else {
                page = cmd_page_create(hdl, modasru, afar);
        }

        page->page_flags |= CMD_MEM_F_FAULTING;
        if (page->page_case.cc_cp == NULL)
                page->page_case.cc_cp = cmd_case_create(hdl,
                    &page->page_header, CMD_PTR_PAGE_CASE, &uuid);

#ifdef sun4v
        nvlfru = cmd_mem2hc(hdl, modfru);
        flt = cmd_nvl_create_fault(hdl, "fault.memory.page", 100,
            page->page_asru_nvl, nvlfru, NULL);
        flt = cmd_fault_add_location(hdl, flt, cmd_fmri_get_unum(modfru));
        if (nvlfru != NULL)
        nvlist_free(nvlfru);
#else /* sun4v */
        flt = cmd_nvl_create_fault(hdl, "fault.memory.page", 100,
            page->page_asru_nvl, modfru, NULL);
#endif /* sun4v */

        if (nvlist_add_boolean_value(flt, FM_SUSPECT_MESSAGE, B_FALSE) != 0)
                fmd_hdl_abort(hdl, "failed to add no-message member to fault");

        fmd_case_add_ereport(hdl, page->page_case.cc_cp, ep);
        fmd_case_add_suspect(hdl, page->page_case.cc_cp, flt);
        fmd_case_solve(hdl, page->page_case.cc_cp);
}
```

_____unchanged_portion_omitted_

```
**********************************************************
   13676 Mon Feb 15 12:56:01 2016
new/usr/src/cmd/fm/modules/sun4u/cpumem-diagnosis/cmd_oplerr.c
patch tsoome-feedback
**********************************************************
```

```
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * OPL platform specific functions for
 * CPU/Memory error diagnosis engine.
 */
#include <cmd.h>
#include <cmd_dimm.h>
#include <cmd_bank.h>
#include <cmd_page.h>
#include <cmd_opl.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/stat.h>

#include <sys/fm/protocol.h>
#include <sys/fm/io/opl_mc_fm.h>
#include <sys/async.h>
#include <sys/opl_olympus_regs.h>
#include <sys/fm/cpu/SPARC64-VI.h>
#include <sys/int_const.h>
#include <sys/mutex.h>
#include <sys/dditypes.h>
#include <opl/sys/mc-opl.h>

/*
 * The following is the common function for handling
 * memory UE with EID=MEM.
 * The error could be detected by either CPU/IO.
 */
cmd_evdisp_t
opl_ue_mem(fmd_hdl_t *hdl, fmd_event_t *ep, nvlist_t *nvl,
```



```
    int hdlr_type)
{
        nvlist_t *rsrc = NULL, *asru = NULL, *fru = NULL;
        uint64_t ubc_ue_log_reg, pa;
        cmd_page_t *page;

        if (nvlist_lookup_nvlist(nvl,
            FM_EREPORT_PAYLOAD_NAME_RESOURCE, &rsrc) != 0)
                return (CMD_EVD_BAD);

        switch (hdlr_type) {
        case CMD_OPL_HDLR_CPU:

                if (nvlist_lookup_uint64(nvl,
                    FM_EREPORT_PAYLOAD_NAME_SFAR, &pa) != 0)
                        return (CMD_EVD_BAD);

                fmd_hdl_debug(hdl, "cmd_ue_mem: pa=%llx\n",
                    (u_longlong_t)pa);
                break;

        case CMD_OPL_HDLR_IO:

                if (nvlist_lookup_uint64(nvl, OBERON_UBC_MUE,
                    &ubc_ue_log_reg) != 0)
                        return (CMD_EVD_BAD);

                pa = (ubc_ue_log_reg & UBC_UE_ADR_MASK);

                fmd_hdl_debug(hdl, "cmd_ue_mem: ue_log_reg=%llx\n",
                    (u_longlong_t)ubc_ue_log_reg);
                fmd_hdl_debug(hdl, "cmd_ue_mem: pa=%llx\n",
                    (u_longlong_t)pa);
                break;

        default:

                return (CMD_EVD_BAD);
        }

        if ((page = cmd_page_lookup(pa)) != NULL &&
            page->page_case.cc_cp != NULL &&
            fmd_case_solved(hdl, page->page_case.cc_cp))
                return (CMD_EVD_REDUND);

        if (nvlist_dup(rsrc, &asru, 0) != 0) {
                fmd_hdl_debug(hdl, "opl_ue_mem nvlist dup failed\n");
                return (CMD_EVD_BAD);
        }

        if (fmd_nvl_fmri_expand(hdl, asru) < 0) {
                nvlist_free(asru);
                CMD_STAT_BUMP(bad_mem_asru);
                return (CMD_EVD_BAD);
        }

        if ((fru = opl_mem_fru_create(hdl, asru)) == NULL) {
                nvlist_free(asru);
                return (CMD_EVD_BAD);
        }

        cmd_page_fault(hdl, asru, fru, ep, pa);
        nvlist_free(asru);
        nvlist_free(fru);
        return (CMD_EVD_OK);
}
```

```
127 /*
128  * The following is the main function to handle generating
129  * the sibling cpu suspect list for the CPU detected UE
130  * error cases.  This is to handle the
131  * multiple strand/core architecture on the OPL platform.
132  */
133 cmd_evdisp_t
134 cmd_opl_ue_cpu(fmd_hdl_t *hdl, fmd_event_t *ep,
135     const char *class, const char *fltname,
136     cmd_ptrsubtype_t ptr, cmd_cpu_t *cpu,
137     cmd_case_t *cc, uint8_t cpumask)
138 {
139         const char *uuid;
140         cmd_cpu_t *main_cpu, *sib_cpu;
141         nvlist_t *fmri;
142         cmd_list_t *cpu_list;
143         opl_cpu_t *opl_cpu;
144         uint32_t main_cpuid, nsusp = 1;
145         uint8_t cert;

147         fmd_hdl_debug(hdl,
148             "Enter OPL_CPUUE_HANDLER for class %x\n", class);

150         main_cpu = cpu;
151         main_cpuid = cpu->cpu_cpuid;

153         if (strcmp(fltname, "core") == 0)
154                 cpu_list = opl_cpulist_insert(hdl, cpu->cpu_cpuid,
155                     IS_CORE);
156         else if (strcmp(fltname, "chip") == 0)
157                 cpu_list = opl_cpulist_insert(hdl, cpu->cpu_cpuid,
158                     IS_CHIP);
159         else
160                 cpu_list = opl_cpulist_insert(hdl, cpu->cpu_cpuid,
161                     IS_STRAND);

163         for (opl_cpu = cmd_list_next(cpu_list); opl_cpu != NULL;
164             opl_cpu = cmd_list_next(opl_cpu)) {
165                 if (opl_cpu->oc_cpuid == main_cpuid) {
166                         sib_cpu = main_cpu;
167                         opl_cpu->oc_cmd_cpu = main_cpu;
168                 } else {
169                         fmri = cmd_cpu_fmri_create(opl_cpu->oc_cpuid, cpumask);
170                         if (fmri == NULL) {
171                                 opl_cpu->oc_cmd_cpu = NULL;
172                                 fmd_hdl_debug(hdl,
173                                     "missing asru, cpuid %u excluded\n",
174                                     opl_cpu->oc_cpuid);
175                                 continue;
176                         }

178                         sib_cpu = cmd_cpu_lookup(hdl, fmri, class,
179                             CMD_CPU_LEVEL_THREAD);
180                         if (sib_cpu == NULL || sib_cpu->cpu_faulting) {
183                                 if (fmri != NULL)
181                                 nvlist_free(fmri);
182                                 opl_cpu->oc_cmd_cpu = NULL;
183                                 fmd_hdl_debug(hdl,
184                                 "cpu not present, cpuid %u excluded\n",
185                                     opl_cpu->oc_cpuid);
186                                 continue;
187                         }
188                         opl_cpu->oc_cmd_cpu = sib_cpu;
192                         if (fmri != NULL)
189                         nvlist_free(fmri);
```



```
190                         nsusp++;
191                 }
192                 if (cpu->cpu_cpuid == main_cpuid) {
193                         if (cc->cc_cp != NULL &&
194                             fmd_case_solved(hdl, cc->cc_cp)) {
195                                 if (cpu_list != NULL)
196                                         opl_cpulist_free(hdl, cpu_list);
197                                 return (CMD_EVD_REDUND);
198                         }

200                         if (cc->cc_cp == NULL)
201                                 cc->cc_cp = cmd_case_create(hdl,
202                                     &cpu->cpu_header, ptr, &uuid);

204                         if (cc->cc_serdnm != NULL) {
205                                 fmd_hdl_debug(hdl,
206                                 "destroying existing %s state for class %x\n",
207                                     cc->cc_serdnm, class);
208                                 fmd_serd_destroy(hdl, cc->cc_serdnm);
209                                 fmd_hdl_strfree(hdl, cc->cc_serdnm);
210                                 cc->cc_serdnm = NULL;
211                                 fmd_case_reset(hdl, cc->cc_cp);
212                         }
213                         fmd_case_add_ereport(hdl, cc->cc_cp, ep);
214                 }
215         }
216         cert = opl_avg(100, nsusp);
217         for (opl_cpu = cmd_list_next(cpu_list); opl_cpu != NULL;
218             opl_cpu = cmd_list_next(opl_cpu)) {
219                 if (opl_cpu->oc_cmd_cpu != NULL) {
220                         nvlist_t *cpu_rsrc;

222                         cpu_rsrc = opl_cpursrc_create(hdl, opl_cpu->oc_cpuid);
223                         if (cpu_rsrc == NULL) {
224                                 fmd_hdl_debug(hdl,
225                                 "missing rsrc, cpuid %u excluded\n",
226                                     opl_cpu->oc_cpuid);
227                                 continue;
228                         }
229                         cmd_cpu_create_faultlist(hdl, cc->cc_cp,
230                             opl_cpu->oc_cmd_cpu, fltname, cpu_rsrc, cert);
231                         nvlist_free(cpu_rsrc);
232                 }
233         }
234         fmd_case_solve(hdl, cc->cc_cp);
235         if (cpu_list != NULL)
236                 opl_cpulist_free(hdl, cpu_list);
237         return (CMD_EVD_OK);
238 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   15764 Mon Feb 15 12:56:01 2016
new/usr/src/cmd/fm/modules/sun4v/cpumem-diagnosis/cmd_hc_sun4v.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 172 nvlist_t *
 173 cmd_mkboard_fru(fmd_hdl_t *hdl, char *frustr, char *serialstr, char *partstr) {

 175         char *nac, *nac_name;
 176         int n, i, len;
 177         nvlist_t *fru, **hc_list;

 179         if (frustr == NULL)
 180                 return (NULL);

 182         if ((nac_name = strstr(frustr, "MB")) == NULL)
 183                 return (NULL);

 185         len = strlen(nac_name) + 1;

 187         nac = fmd_hdl_zalloc(hdl, len, FMD_SLEEP);
 188         (void) strcpy(nac, nac_name);

 190         n = cmd_count_components(nac, '/');

 192         fmd_hdl_debug(hdl, "cmd_mkboard_fru: nac=%s components=%d\n", nac, n);

 194         hc_list = fmd_hdl_zalloc(hdl, sizeof (nvlist_t *)*n, FMD_SLEEP);

 196         for (i = 0; i < n; i++) {
 197                 (void) nvlist_alloc(&hc_list[i],
 198                     NV_UNIQUE_NAME|NV_UNIQUE_NAME_TYPE, 0);
 199         }

 201         if (cmd_breakup_components(nac, "/", hc_list) < 0) {
 202                 for (i = 0; i < n; i++) {
 203                         if (hc_list[i] != NULL)
 203                         nvlist_free(hc_list[i]);
 204                 }
 205                 fmd_hdl_free(hdl, hc_list, sizeof (nvlist_t *)*n);
 206                 fmd_hdl_free(hdl, nac, len);
 207                 return (NULL);
 208         }

 210         if (nvlist_alloc(&fru, NV_UNIQUE_NAME, 0) != 0) {
 211                 for (i = 0; i < n; i++) {
 213                         if (hc_list[i] != NULL)
 212                         nvlist_free(hc_list[i]);
 213                 }
 214                 fmd_hdl_free(hdl, hc_list, sizeof (nvlist_t *)*n);
 215                 fmd_hdl_free(hdl, nac, len);
 216                 return (NULL);
 217         }

 219         if (nvlist_add_uint8(fru, FM_VERSION, FM_HC_SCHEME_VERSION) != 0 ||
 220             nvlist_add_string(fru, FM_FMRI_SCHEME, FM_FMRI_SCHEME_HC) != 0 ||
 221             nvlist_add_string(fru, FM_FMRI_HC_ROOT, "") != 0 ||
 222             nvlist_add_uint32(fru, FM_FMRI_HC_LIST_SZ, n) != 0 ||
 223             nvlist_add_nvlist_array(fru, FM_FMRI_HC_LIST, hc_list, n) != 0) {
 224                 for (i = 0; i < n; i++) {
 227                         if (hc_list[i] != NULL)
 225                         nvlist_free(hc_list[i]);
 226                 }
 227                 fmd_hdl_free(hdl, hc_list, sizeof (nvlist_t *)*n);
```



```
 228                 fmd_hdl_free(hdl, nac, len);
 229                 nvlist_free(fru);
 230                 return (NULL);
 231         }

 233         for (i = 0; i < n; i++) {
 237                 if (hc_list[i] != NULL)
 234                 nvlist_free(hc_list[i]);
 235         }
 236         fmd_hdl_free(hdl, hc_list, sizeof (nvlist_t *)*n);
 237         fmd_hdl_free(hdl, nac, len);

 239         if ((serialstr != NULL &&
 240             nvlist_add_string(fru, FM_FMRI_HC_SERIAL_ID, serialstr) != 0) ||
 241             (partstr != NULL &&
 242             nvlist_add_string(fru, FM_FMRI_HC_PART, partstr) != 0)) {
 243                 nvlist_free(fru);
 244                 return (NULL);
 245         }

 247         return (fru);
 248 }

 250 nvlist_t *
 251 cmd_boardfru_create_fault(fmd_hdl_t *hdl, nvlist_t *asru, const char *fltnm,
 252     uint_t cert, char *loc)
 253 {
 254         nvlist_t *flt, *nvlfru;
 255         char *serialstr, *partstr;

 257         if ((loc == NULL) || (strcmp(loc, EMPTY_STR) == 0))
 258                 return (NULL);

 260         if (nvlist_lookup_string(asru, FM_FMRI_HC_SERIAL_ID, &serialstr) != 0)
 261                 serialstr = NULL;
 262         if (nvlist_lookup_string(asru, FM_FMRI_HC_PART, &partstr) != 0)
 263                 partstr = NULL;

 265         nvlfru = cmd_mkboard_fru(hdl, loc, serialstr, partstr);
 266         if (nvlfru == NULL)
 267                 return (NULL);

 269         flt = cmd_nvl_create_fault(hdl, fltnm, cert, nvlfru, nvlfru, NULL);
 270         flt = cmd_fault_add_location(hdl, flt, loc);
 275         if (nvlfru != NULL)
 271         nvlist_free(nvlfru);
 272         return (flt);
 273 }
_____unchanged_portion_omitted_

 589 nvlist_t *
 590 cmd_nvl_create_fault(fmd_hdl_t *hdl, const char *class, uint8_t cert,
 591     nvlist_t *asru, nvlist_t *fru, nvlist_t *rsrc)
 592 {
 593         nvlist_t *fllist;
 594         uint64_t offset, phyaddr;
 595         nvlist_t *hsp = NULL;

 597         rsrc = NULL;
 598         (void) nvlist_add_nvlist(fru, FM_FMRI_AUTHORITY,
 599             cmd.cmd_auth); /* not an error if this fails */

 601         if (strstr(class, "fault.memory.") != NULL) {
 602                 /*
 603                  * For T1 platform fault.memory.bank and fault.memory.dimm,
 604                  * do not issue the hc schmem for resource and fru
```

```
605                  */
606                 if (is_T1_platform(asru) && (strstr(class, ".page") == NULL)) {
607                         fllist = fmd_nvl_create_fault(hdl, class, cert, asru,
608                             fru, fru);
609                         return (fllist);
610                 }

612                 rsrc = get_mem_fault_resource(hdl, fru);
613                 /*
614                  * Need to append the phyaddr & offset into the
615                  * hc-specific of the fault.memory.page resource
616                  */
617                 if ((rsrc != NULL) && strstr(class, ".page") != NULL) {
618                         if (nvlist_alloc(&hsp, NV_UNIQUE_NAME, 0) == 0) {
619                                 if (nvlist_lookup_uint64(asru,
620                                     FM_FMRI_MEM_PHYSADDR, &phyaddr) == 0)
621                                         (void) (nvlist_add_uint64(hsp,
622                                             FM_FMRI_MEM_PHYSADDR,
623                                             phyaddr));

625                                 if (nvlist_lookup_uint64(asru,
626                                     FM_FMRI_MEM_OFFSET, &offset) == 0)
627                                         (void) nvlist_add_uint64(hsp,
628                                             FM_FMRI_HC_SPECIFIC_OFFSET, offset);

630                                 (void) nvlist_add_nvlist(rsrc,
631                                     FM_FMRI_HC_SPECIFIC, hsp);
632                         }
633                 }
634                 fllist = fmd_nvl_create_fault(hdl, class, cert, asru,
635                     fru, rsrc);
641                 if (hsp != NULL)
636                 nvlist_free(hsp);
637         } else {
638                 rsrc = get_cpu_fault_resource(hdl, asru);
639                 fllist = fmd_nvl_create_fault(hdl, class, cert, asru,
640                     fru, rsrc);
641         }

649         if (rsrc != NULL)
643         nvlist_free(rsrc);

645         return (fllist);
646 }
```

_____***unchanged_portion_omitted_***

****
20267 Mon Feb 15 12:56:01 2016
new/usr/src/cmd/fm/modules/sun4v/cpumem-diagnosis/cmd_memerr_arch.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
 482 /*ARGSUSED*/
 483 cmd_evdisp_t
 484 cmd_fw_defect(fmd_hdl_t *hdl, fmd_event_t *ep, nvlist_t *nvl, const char *class,
 485     cmd_errcl_t clcode)
 486 {
 487         const char *fltclass = NULL;
 488         nvlist_t *rsc = NULL;
 489         int solve = 0;

 491         if ((rsc = init_mb(hdl)) == NULL)
 492                 return (CMD_EVD_UNUSED);

 494         if (ERR_CLASS(class, FERG_INVALID) == 0) {
 495                 fltclass = "defect.fw.generic-sparc.erpt-gen";
 496         } else if (ERR_CLASS(class, DBU_ERROR) == 0) {
 497                 cmd_evdisp_t rc;
 498                 fltclass = "defect.fw.generic-sparc.addr-oob";
 499                 /*
 500                  * add dbu to nop error train
 501                  */
 502                 rc = cmd_xxcu_initial(hdl, ep, nvl, class, clcode,
 503                     CMD_XR_HDLR_NOP);
 504                 if (rc != 0)
 505                         fmd_hdl_debug(hdl,
 506                             "Failed to add error (%llx) to the train, rc = %d",
 507                             clcode, rc);
 508         } else {
 509                 fmd_hdl_debug(hdl, "Unexpected fw defect event %s", class);
 510         }

 512         if (fltclass) {
 513                 fmd_case_t *cp = NULL;
 514                 nvlist_t *fault = NULL;

 516                 fault = fmd_nvl_create_fault(hdl, fltclass, 100, NULL,
 517                     NULL, rsc);
 518                 if (fault != NULL) {
 519                         cp = fmd_case_open(hdl, NULL);
 520                         fmd_case_add_ereport(hdl, cp, ep);
 521                         fmd_case_add_suspect(hdl, cp, fault);
 522                         fmd_case_solve(hdl, cp);
 523                         solve = 1;
 524                 }
 525         }

 527         if (rsc)
 527         nvlist_free(rsc);

 529         return (solve ? CMD_EVD_OK : CMD_EVD_UNUSED);
 530 }
```

_____***unchanged_portion_omitted_***

****
7314 Mon Feb 15 12:56:01 2016
new/usr/src/cmd/fm/modules/sun4v/cpumem-retire/cma_cpu_sun4v.c
patch tsoome-feedback
****
_____***unchanged_portion_omitted_***

```
 262 static void
 263 cma_cpu_free(fmd_hdl_t *hdl, cma_cpu_t *cpu)
 264 {
 265         if (cpu->cpu_fmri != NULL)
 265         nvlist_free(cpu->cpu_fmri);
 266         if (cpu->cpu_uuid != NULL)
 267                 fmd_hdl_strfree(hdl, cpu->cpu_uuid);
 268         fmd_hdl_free(hdl, cpu, sizeof (cma_cpu_t));
 269 }
```

_____***unchanged_portion_omitted_***

****
13013 Mon Feb 15 12:56:01 2016
new/usr/src/cmd/fm/modules/sun4v/generic-mem/gmem_dimm.c
patch tsoome-feedback
****
_____***unchanged_portion_omitted_***

```
470 int
471 gmem_dimm_present(fmd_hdl_t *hdl, nvlist_t *asru)
472 {
473         char *sn;
474         nvlist_t *dimm = NULL;

476         if (nvlist_lookup_string(asru, FM_FMRI_HC_SERIAL_ID, &sn) != 0) {
477                 fmd_hdl_debug(hdl, "Unable to get dimm serial\n");
478                 return (0);
479         }
480         dimm = gmem_find_dimm_fru(hdl, sn);
481         if (dimm == NULL) {
482                 fmd_hdl_debug(hdl, "Dimm sn=%s is not present\n", sn);
483                 return (0);
484         }
485         if (dimm != NULL)
485         nvlist_free(dimm);
486         return (1);
487 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   23424 Mon Feb 15 12:56:02 2016
new/usr/src/cmd/fm/modules/sun4v/generic-mem/gmem_memerr.c
patch tsoome-feedback
**********************************************************
```

*_____unchanged_portion_omitted_*

```
157 /*
158  * fault the FRU of the common detector between two DIMMs
159  */
160 void
161 gmem_gen_datapath_fault(fmd_hdl_t *hdl, nvlist_t *det)
162 {
163         char *name, *id;
164         nvlist_t **hcl1, **hcl;
165         uint_t n;
166         int i, j;
167         fmd_case_t *cp;
168         nvlist_t *fltlist, *rsrc;
169         nvlist_t *fru = NULL;

171         if (nvlist_lookup_nvlist_array(det, FM_FMRI_HC_LIST, &hcl1, &n) < 0)
172                 return;

174         for (i = 0; i < n; i++) {
175                 (void) nvlist_lookup_string(hcl1[i], FM_FMRI_HC_NAME, &name);
176                 if (strcmp(name, "chip") == 0)
177                         break;
178         }

180         n = i + 1;
181         hcl = fmd_hdl_zalloc(hdl, sizeof (nvlist_t *) * n, FMD_SLEEP);
182         if (hcl == NULL)
183                 return;

185         for (i = 0; i < n; i++) {
186                 (void) nvlist_alloc(&hcl[i],
187                     NV_UNIQUE_NAME|NV_UNIQUE_NAME_TYPE, 0);
188         }

190         for (i = 0, j = 0; i < n; i++) {
191                 (void) nvlist_lookup_string(hcl1[i], FM_FMRI_HC_NAME, &name);
192                 (void) nvlist_lookup_string(hcl1[i], FM_FMRI_HC_ID, &id);
193                 (void) nvlist_add_string(hcl[j], FM_FMRI_HC_NAME, name);
194                 (void) nvlist_add_string(hcl[j], FM_FMRI_HC_ID, id);
195                 j++;
196                 if (strcmp(name, "chip") == 0)
197                         break;
198         }

200         if (nvlist_alloc(&rsrc,  NV_UNIQUE_NAME|NV_UNIQUE_NAME_TYPE, 0) != 0) {
201                 for (i = 0; i < n; i++) {
202                         if (hcl[i] != NULL)
202                         nvlist_free(hcl[i]);
203                 }
204                 fmd_hdl_free(hdl, hcl, sizeof (nvlist_t *) * n);
205         }

207         if (nvlist_add_uint8(rsrc, FM_VERSION, FM_HC_SCHEME_VERSION) != 0 ||
208             nvlist_add_string(rsrc, FM_FMRI_SCHEME, FM_FMRI_SCHEME_HC) != 0 ||
209             nvlist_add_string(rsrc, FM_FMRI_HC_ROOT, "") != 0 ||
210             nvlist_add_uint32(rsrc, FM_FMRI_HC_LIST_SZ, n) != 0 ||
211             nvlist_add_nvlist_array(rsrc, FM_FMRI_HC_LIST, hcl, n) != 0) {
212                 for (i = 0; i < n; i++) {
214                         if (hcl[i] != NULL)
213                         nvlist_free(hcl[i]);
```



```
214                 }
215                 fmd_hdl_free(hdl, hcl, sizeof (nvlist_t *) * n);
216                 nvlist_free(rsrc);
217         }

219         fru = gmem_find_fault_fru(hdl, rsrc);
220         if (fru != NULL) {
221                 cp = fmd_case_open(hdl, NULL);
222                 fltlist = fmd_nvl_create_fault(hdl, "fault.memory.datapath",
223                     100, fru, fru, fru);
224                 fmd_case_add_suspect(hdl, cp, fltlist);
225                 fmd_case_solve(hdl, cp);
226                 nvlist_free(fru);
227         }

229         for (i = 0; i < n; i++) {
232                 if (hcl[i] != NULL)
230                 nvlist_free(hcl[i]);
231         }

233         fmd_hdl_free(hdl, hcl, sizeof (nvlist_t *) * n);
234         nvlist_free(rsrc);
235 }
```

*_____unchanged_portion_omitted_*

```
361 /*
362  * rule 5a checking. The check succeeds if
363  * - nretired >= 512
364  * - nretired >= 128 and (addr_hi - addr_low) / (nretired -1 ) > 512KB
365  */
366 static void
367 ce_thresh_check(fmd_hdl_t *hdl, gmem_dimm_t *dimm)
368 {
369         nvlist_t *flt, *rsrc;
370         fmd_case_t *cp;
371         uint_t nret;
372         uint64_t delta_addr = 0;

374         if (dimm->dimm_flags & GMEM_F_FAULTING)
375                 return;

377         nret = dimm->dimm_nretired;

379         if (nret < gmem.gm_low_ce_thresh)
380                 return;

382         if (dimm->dimm_phys_addr_hi >= dimm->dimm_phys_addr_low)
383                 delta_addr =
384                     (dimm->dimm_phys_addr_hi - dimm->dimm_phys_addr_low) /
385                     (nret - 1);

387         if (nret >= gmem.gm_max_retired_pages || delta_addr > GMEM_MQ_512KB) {

389                 fmd_hdl_debug(hdl, "ce_thresh_check succeeded nret=%d", nret);
390                 dimm->dimm_flags |= GMEM_F_FAULTING;
391                 gmem_dimm_dirty(hdl, dimm);

393                 cp = fmd_case_open(hdl, NULL);
394                 rsrc = gmem_find_dimm_rsc(hdl, dimm->dimm_serial);
395                 flt = fmd_nvl_create_fault(hdl, GMEM_FAULT_DIMM_PAGES,
396                     GMEM_FLTMAXCONF, NULL, gmem_dimm_fru(dimm), rsrc);
397                 fmd_case_add_suspect(hdl, cp, flt);
398                 fmd_case_solve(hdl, cp);
402                 if (rsrc != NULL)
399                 nvlist_free(rsrc);
400         }
```

```
401 }

403 /*
404  * rule 5b checking. The check succeeds if more than 120
405  * non-intermittent CEs are reported against one symbol
406  * position of one afar in 72 hours
407  */
408 static void
409 mq_5b_check(fmd_hdl_t *hdl, gmem_dimm_t *dimm)
410 {
411         nvlist_t *flt, *rsrc;
412         fmd_case_t *cp;
413         gmem_mq_t *ip, *next;
414         int cw;

416         for (cw = 0; cw < GMEM_MAX_CKWDS; cw++) {
417                 for (ip = gmem_list_next(&dimm->mq_root[cw]);
418                     ip != NULL; ip = next) {
419                         next = gmem_list_next(ip);
420                         if (ip->mq_dupce_count >= gmem.gm_dupce) {
421                                 fmd_hdl_debug(hdl,
422                                     "mq_5b_check succeeded: duplicate CE=%d",
423                                     ip->mq_dupce_count);
424                                 cp = fmd_case_open(hdl, NULL);
425                                 rsrc = gmem_find_dimm_rsc(hdl,
426                                     dimm->dimm_serial);
427                                 flt = fmd_nvl_create_fault(hdl,
428                                     GMEM_FAULT_DIMM_PAGES, GMEM_FLTMAXCONF,
429                                     NULL, gmem_dimm_fru(dimm), rsrc);
430                                 dimm->dimm_flags |= GMEM_F_FAULTING;
431                                 gmem_dimm_dirty(hdl, dimm);
432                                 fmd_case_add_suspect(hdl, cp, flt);
433                                 fmd_case_solve(hdl, cp);
438                                 if (rsrc != NULL)
434                                 nvlist_free(rsrc);
435                                 return;
436                         }
437                 }
438         }
439 }
```

*unchanged_portion_omitted_*

```
612 /*
613  * Check the MQSC index lists (one for each checkword) by making a
614  * complete pass through each list, checking if the criteria for
615  * Rule 4A has been met.  Rule 4A checking is done for each checkword.
616  *
617  * Rule 4A: fault a DIMM  "whenever Solaris reports two or more CEs from
618  * two or more different physical addresses on each of two or more different
619  * bit positions from the same DIMM within 72 hours of each other, and all
620  * the addresses are in the same relative checkword (that is, the AFARs
621  * are all the same modulo 64).  [Note: This means at least 4 CEs; two
622  * from one bit position, with unique addresses, and two from another,
623  * also with unique addresses, and the lower 6 bits of all the addresses
624  * are the same."
625  */

627 void
628 mq_check(fmd_hdl_t *hdl, gmem_dimm_t *dimm)
629 {
630         int upos_pairs, curr_upos, cw, i, j;
631         nvlist_t *flt, *rsc;
632         typedef struct upos_pair {
633                 int upos;
634                 gmem_mq_t *mq1;
635                 gmem_mq_t *mq2;
```



```
636         } upos_pair_t;
637         upos_pair_t upos_array[16]; /* max per cw = 2, * 8 cw's */
638         gmem_mq_t *ip;

640         /*
641          * Each upos_array[] member represents a pair of CEs for the same
642          * unit position (symbol) which is a 4 bit nibble.
643          * MQSC rule 4 requires pairs of CEs from the same symbol (same DIMM
644          * for rule 4A, and same DRAM for rule 4B) for a violation - this
645          * is why CE pairs are tracked.
646          */
647         upos_pairs = 0;
648         upos_array[0].mq1 = NULL;

650         for (cw = 0; cw < GMEM_MAX_CKWDS; cw++) {
651                 i = upos_pairs;
652                 curr_upos = -1;

654                 /*
655                  * mq_root[] is an array of cumulative lists of CEs
656                  * indexed by checkword where the list is in unit position
657                  * order. Loop through checking for duplicate unit position
658                  * entries (filled in at mq_create()).
659                  * The upos_array[] is filled in each time a duplicate
660                  * unit position is found; the first time through the loop
661                  * of a unit position sets curr_upos but does not fill in
662                  * upos_array[] until the second symbol is found.
663                  */
664                 for (ip = gmem_list_next(&dimm->mq_root[cw]); ip != NULL;
665                     ip = gmem_list_next(ip)) {
666                         if (curr_upos != ip->mq_unit_position) {
667                                 /* Set initial current position */
668                                 curr_upos = ip->mq_unit_position;
669                         } else if (i > upos_pairs &&
670                             curr_upos == upos_array[i-1].upos) {
671                                 /*
672                                  * Only keep track of CE pairs; skip
673                                  * triples, quads, etc...
674                                  */
675                                 continue;
676                         } else if (upos_array[i].mq1 == NULL) {
677                                 /* Have a pair. Add to upos_array[] */
678                                 fmd_hdl_debug(hdl, "pair:upos=%d",
679                                     curr_upos);
680                                 upos_array[i].upos = curr_upos;
681                                 upos_array[i].mq1 = gmem_list_prev(ip);
682                                 upos_array[i].mq2 = ip;
683                                 upos_array[++i].mq1 = NULL;
684                         }
685                 }
686                 if (i - upos_pairs >= 2) {
687                         /* Rule 4A violation */
688                         rsc = gmem_find_dimm_rsc(hdl, dimm->dimm_serial);
689                         flt = fmd_nvl_create_fault(hdl, GMEM_FAULT_DIMM_4A,
690                             GMEM_FLTMAXCONF, NULL, gmem_dimm_fru(dimm), rsc);
691                         for (j = upos_pairs; j < i; j++) {
692                                 fmd_case_add_ereport(hdl,
693                                     dimm->dimm_case.cc_cp,
694                                     upos_array[j].mq1->mq_ep);
695                                 fmd_case_add_ereport(hdl,
696                                     dimm->dimm_case.cc_cp,
697                                     upos_array[j].mq2->mq_ep);
698                         }
699                         dimm->dimm_flags |= GMEM_F_FAULTING;
700                         gmem_dimm_dirty(hdl, dimm);
701                         fmd_case_add_suspect(hdl, dimm->dimm_case.cc_cp, flt);
```

```
 702                         fmd_case_solve(hdl, dimm->dimm_case.cc_cp);
 708                         if (rsc != NULL)
 703                         nvlist_free(rsc);
 704                         return;
 705                 }
 706                 upos_pairs = i;
 707                 assert(upos_pairs < 16);
 708         }
 709 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    9031 Mon Feb 15 12:56:02 2016
new/usr/src/cmd/fm/modules/sun4v/generic-mem/gmem_page.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 246 /*ARGSUSED*/
 247 int
 248 gmem_page_unusable(fmd_hdl_t *hdl, gmem_page_t *page)
 249 {
 250         nvlist_t *asru = NULL;
 251         char *sn;

 253         if (nvlist_lookup_string(page->page_asru_nvl,
 254             FM_FMRI_HC_SERIAL_ID, &sn) != 0)
 255                 return (1);

 257         /*
 258          * get asru in mem scheme from topology
 259          */
 260         asru = gmem_find_dimm_asru(hdl, sn);
 261         if (asru == NULL)
 262                 return (1);

 264         (void) nvlist_add_string_array(asru, FM_FMRI_MEM_SERIAL_ID, &sn, 1);
 265         (void) nvlist_add_uint64(asru, FM_FMRI_MEM_PHYSADDR,
 266             page->page_physbase);
 267         (void) nvlist_add_uint64(asru, FM_FMRI_MEM_OFFSET, page->page_offset);

 269         if (fmd_nvl_fmri_unusable(hdl, asru)) {
 270                 nvlist_free(asru);
 271                 return (1);
 272         }

 274         if (asru != NULL)
 274         nvlist_free(asru);

 276         return (0);
 277 }
______unchanged_portion_omitted_


 316 int
 317 gmem_page_fault(fmd_hdl_t *hdl, nvlist_t *fru, nvlist_t *rsc,
 318     fmd_event_t *ep, uint64_t afar, uint64_t offset)
 319 {
 320         gmem_page_t *page = NULL;
 321         const char *uuid;
 322         nvlist_t *flt, *hsp;

 324         page = gmem_page_lookup(afar);
 325         if (page != NULL) {
 326                 if (page->page_flags & GMEM_F_FAULTING ||
 327                     gmem_page_unusable(hdl, page)) {
 329                         if (rsc != NULL)
 328                         nvlist_free(rsc);
 329                         page->page_flags |= GMEM_F_FAULTING;
 330                         return (0);
 331                 }
 332         } else {
 333                 page = gmem_page_create(hdl, fru, afar, offset);
 334         }

 336         page->page_flags |= GMEM_F_FAULTING;
 337         if (page->page_case.cc_cp == NULL)
```



```
 338                 page->page_case.cc_cp = gmem_case_create(hdl,
 339                     &page->page_header, GMEM_PTR_PAGE_CASE, &uuid);

 341         if (nvlist_lookup_nvlist(page->page_asru_nvl, FM_FMRI_HC_SPECIFIC,
 342             &hsp) == 0)
 343                 (void) nvlist_add_nvlist(rsc, FM_FMRI_HC_SPECIFIC, hsp);

 345         flt = fmd_nvl_create_fault(hdl, GMEM_FAULT_PAGE, 100, NULL, fru, rsc);
 348         if (rsc != NULL)
 346         nvlist_free(rsc);

 348         if (nvlist_add_boolean_value(flt, FM_SUSPECT_MESSAGE, B_FALSE) != 0)
 349                 fmd_hdl_abort(hdl, "failed to add no-message member to fault");

 351         fmd_case_add_ereport(hdl, page->page_case.cc_cp, ep);
 352         fmd_case_add_suspect(hdl, page->page_case.cc_cp, flt);
 353         fmd_case_solve(hdl, page->page_case.cc_cp);
 354         return (1);
 355 }
______unchanged_portion_omitted_
```

```
**********************************************************
    7367 Mon Feb 15 12:56:02 2016
new/usr/src/cmd/fm/schemes/hc/scheme.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 206 static int
 207 fru_compare(nvlist_t *r1, nvlist_t *r2)
 208 {
 209         topo_hdl_t *thp;
 210         nvlist_t *f1 = NULL, *f2 = NULL;
 211         nvlist_t **h1 = NULL, **h2 = NULL;
 212         uint_t h1sz, h2sz;
 213         int err, rc = 1;

 215         if ((thp = fmd_fmri_topo_hold(TOPO_VERSION)) == NULL)
 216                 return (fmd_fmri_set_errno(EINVAL));

 218         (void) topo_fmri_fru(thp, r1, &f1, &err);
 219         (void) topo_fmri_fru(thp, r2, &f2, &err);
 220         if (f1 != NULL && f2 != NULL) {
 221                 (void) nvlist_lookup_nvlist_array(f1, FM_FMRI_HC_LIST, &h1,
 222                     &h1sz);
 223                 (void) nvlist_lookup_nvlist_array(f2, FM_FMRI_HC_LIST, &h2,
 224                     &h2sz);
 225                 if (h1sz == h2sz && hclist_contains(h1, h1sz, h2, h2sz) == 1)
 226                         rc = 0;
 227         }

 229         fmd_fmri_topo_rele(thp);
 230         if (f1 != NULL)
 230         nvlist_free(f1);
 232         if (f2 != NULL)
 231         nvlist_free(f2);
 232         return (rc);
 233 }
______unchanged_portion_omitted_
```

```
**********************************************************
   15777 Mon Feb 15 12:56:02 2016
new/usr/src/cmd/fm/schemes/mem/mem.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 452 /*
 453  * We can only make a usable/unusable determination for pages.  Mem FMRIs
 454  * without page addresses will be reported as usable since Solaris has no
 455  * way at present to dynamically disable an entire DIMM or DIMM pair.
 456  */
 457 int
 458 fmd_fmri_unusable(nvlist_t *nvl)
 459 {
 460         uint64_t val1, val2;
 461         uint8_t version;
 462         int rc, err1 = 0, err2;
 463         nvlist_t *nvlcp = NULL;
 464         int retval;
 465         topo_hdl_t *thp;

 467         if (nvlist_lookup_uint8(nvl, FM_VERSION, &version) != 0 ||
 468             version > FM_MEM_SCHEME_VERSION)
 469                 return (fmd_fmri_set_errno(EINVAL));

 471         /*
 472          * If the mem-scheme topology exports this method unusable(), invoke it.
 473          */
 474         if ((thp = fmd_fmri_topo_hold(TOPO_VERSION)) == NULL)
 475                 return (fmd_fmri_set_errno(EINVAL));
 476         rc = topo_fmri_unusable(thp, nvl, &err1);
 477         fmd_fmri_topo_rele(thp);
 478         if (err1 != ETOPO_METHOD_NOTSUP)
 479                 return (rc);

 481         err1 = nvlist_lookup_uint64(nvl, FM_FMRI_MEM_OFFSET, &val1);
 482         err2 = nvlist_lookup_uint64(nvl, FM_FMRI_MEM_PHYSADDR, &val2);

 484         if (err1 == ENOENT && err2 == ENOENT)
 485                 return (0); /* no page, so assume it's still usable */

 487         if ((err1 != 0 && err1 != ENOENT) || (err2 != 0 && err2 != ENOENT))
 488                 return (fmd_fmri_set_errno(EINVAL));

 490         if ((rc = mem_unum_rewrite(nvl, &nvlcp)) != 0)
 491                 return (fmd_fmri_set_errno(rc));

 493         /*
 494          * Ask the kernel if the page is retired, using either the rewritten
 495          * hc FMRI or the original mem FMRI with the specified offset or PA.
 496          * Refer to the kernel's page_retire_check() for the error codes.
 497          */
 498         rc = page_isretired(nvlcp ? nvlcp : nvl, NULL);

 500         if (rc == FMD_AGENT_RETIRE_FAIL) {
 501                 /*
 502                  * The page is not retired and is not scheduled for retirement
 503                  * (i.e. no request pending and has not seen any errors)
 504                  */
 505                 retval = 0;
 506         } else if (rc == FMD_AGENT_RETIRE_DONE ||
 507             rc == FMD_AGENT_RETIRE_ASYNC) {
 508                 /*
 509                  * The page has been retired, is in the process of being
 510                  * retired, or doesn't exist.  The latter is valid if the page
```



```
 511                  * existed in the past but has been DR'd out.
 512                  */
 513                 retval = 1;
 514         } else {
 515                 /*
 516                  * Errors are only signalled to the caller if they're the
 517                  * caller's fault.  This isn't - it's a failure of the
 518                  * retirement-check code.  We'll whine about it and tell
 519                  * the caller the page is unusable.
 520                  */
 521                 fmd_fmri_warn("failed to determine page %s=%llx usability: "
 522                     "rc=%d errno=%d\n", err1 == 0 ? FM_FMRI_MEM_OFFSET :
 523                     FM_FMRI_MEM_PHYSADDR, err1 == 0 ? (u_longlong_t)val1 :
 524                     (u_longlong_t)val2, rc, errno);
 525                 retval = 1;
 526         }

 528         if (nvlcp)
 528                 nvlist_free(nvlcp);

 530         return (retval);
 531 }
______unchanged_portion_omitted_
```

```
**********************************************************
   70821 Mon Feb 15 12:56:02 2016
new/usr/src/cmd/fs.d/nfs/mountd/mountd.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

1496 /*
1497  * Determine whether two paths are within the same file system.
1498  * Returns nonzero (true) if paths are the same, zero (false) if
1499  * they are different.  If an error occurs, return false.
1500  *
1501  * Use the actual FSID if it's available (via getattrat()); otherwise,
1502  * fall back on st_dev.
1503  *
1504  * With ZFS snapshots, st_dev differs from the regular file system
1505  * versus the snapshot.  But the fsid is the same throughout.  Thus
1506  * the fsid is a better test.
1507  */
1508 static int
1509 same_file_system(const char *path1, const char *path2)
1510 {
1511         uint64_t fsid1, fsid2;
1512         struct stat64 st1, st2;
1513         nvlist_t *nvl1 = NULL;
1514         nvlist_t *nvl2 = NULL;

1516         if ((getattrat(AT_FDCWD, XATTR_VIEW_READONLY, path1, &nvl1) == 0) &&
1517             (getattrat(AT_FDCWD, XATTR_VIEW_READONLY, path2, &nvl2) == 0) &&
1518             (nvlist_lookup_uint64(nvl1, A_FSID, &fsid1) == 0) &&
1519             (nvlist_lookup_uint64(nvl2, A_FSID, &fsid2) == 0)) {
1520                 nvlist_free(nvl1);
1521                 nvlist_free(nvl2);
1522                 /*
1523                  * We have found fsid's for both paths.
1524                  */

1526                 if (fsid1 == fsid2)
1527                         return (B_TRUE);

1529                 return (B_FALSE);
1530         }

1532         if (nvl1 != NULL)
1532         nvlist_free(nvl1);
1534         if (nvl2 != NULL)
1533         nvlist_free(nvl2);

1535         /*
1536          * We were unable to find fsid's for at least one of the paths.
1537          * fall back on st_dev.
1538          */

1540         if (stat64(path1, &st1) < 0) {
1541                 syslog(LOG_NOTICE, "%s: %m", path1);
1542                 return (B_FALSE);
1543         }
1544         if (stat64(path2, &st2) < 0) {
1545                 syslog(LOG_NOTICE, "%s: %m", path2);
1546                 return (B_FALSE);
1547         }

1549         if (st1.st_dev == st2.st_dev)
1550                 return (B_TRUE);

1552         return (B_FALSE);
```



```
1553 }
______unchanged_portion_omitted_
```

```
**********************************************************
   17881 Mon Feb 15 12:56:02 2016
new/usr/src/cmd/hotplugd/hotplugd_door.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_
```

```
146 /*
147  * door_server()
148  *
149  *      This routine is the handler which responds to each door call.
150  *      Each incoming door call is expected to send a packed nvlist
151  *      of arguments which describe the requested action.  And each
152  *      response is sent back as a packed nvlist of results.
153  *
154  *      Results are always allocated on the heap.  A global list of
155  *      allocated result buffers is managed, and each one is tracked
156  *      by a unique sequence number.  The final step in the protocol
157  *      is for the caller to send a short response using the sequence
158  *      number when the buffer can be released.
159  */
160 /*ARGSUSED*/
161 static void
162 door_server(void *cookie, char *argp, size_t sz, door_desc_t *dp, uint_t ndesc)
163 {
164         nvlist_t        *args = NULL;
165         nvlist_t        *results = NULL;
166         hp_cmd_t        cmd;
167         int             rv;
168
169         dprintf("Door call: cookie=%p, argp=%p, sz=%d\n", cookie, (void *)argp,
170             sz);
171
172         /* Special case to free a results buffer */
173         if (sz == sizeof (uint64_t)) {
174                 free_buffer(*(uint64_t *)(uintptr_t)argp);
175                 (void) door_return(NULL, 0, NULL, 0);
176                 return;
177         }
178
179         /* Unpack the arguments nvlist */
180         if (nvlist_unpack(argp, sz, &args, 0) != 0) {
181                 log_err("Cannot unpack door arguments.\n");
182                 rv = EINVAL;
183                 goto fail;
184         }
185
186         /* Extract the requested command */
187         if (nvlist_lookup_int32(args, HPD_CMD, (int32_t *)&cmd) != 0) {
188                 log_err("Cannot decode door command.\n");
189                 rv = EINVAL;
190                 goto fail;
191         }
192
193         /* Implement the command */
194         switch (cmd) {
195         case HP_CMD_GETINFO:
196                 rv = cmd_getinfo(args, &results);
197                 break;
198         case HP_CMD_CHANGESTATE:
199                 rv = cmd_changestate(args, &results);
200                 break;
201         case HP_CMD_SETPRIVATE:
202         case HP_CMD_GETPRIVATE:
203                 rv = cmd_private(cmd, args, &results);
204                 break;
```



```
205         default:
206                 rv = EINVAL;
207                 break;
208         }
209
210         /* The arguments nvlist is no longer needed */
211         nvlist_free(args);
212         args = NULL;
213
214         /*
215          * If an nvlist was constructed for the results,
216          * then pack the results nvlist and return it.
217          */
218         if (results != NULL) {
219                 uint64_t        seqnum;
220                 char            *buf = NULL;
221                 size_t          len = 0;
222
223                 /* Add a sequence number to the results */
224                 seqnum = get_seqnum();
225                 if (nvlist_add_uint64(results, HPD_SEQNUM, seqnum) != 0) {
226                         log_err("Cannot add sequence number.\n");
227                         rv = EFAULT;
228                         goto fail;
229                 }
230
231                 /* Pack the results nvlist */
232                 if (nvlist_pack(results, &buf, &len,
233                     NV_ENCODE_NATIVE, 0) != 0) {
234                         log_err("Cannot pack door results.\n");
235                         rv = EFAULT;
236                         goto fail;
237                 }
238
239                 /* Link results buffer into list */
240                 add_buffer(seqnum, buf);
241
242                 /* The results nvlist is no longer needed */
243                 nvlist_free(results);
244
245                 /* Return the results */
246                 (void) door_return(buf, len, NULL, 0);
247                 return;
248         }
249
250         /* Return result code (when no nvlist) */
251         (void) door_return((char *)&rv, sizeof (int), NULL, 0);
252         return;
253
254 fail:
255         log_err("Door call failed (%s)\n", strerror(rv));
256         if (args != NULL)
256                 nvlist_free(args);
258         if (results != NULL)
257                 nvlist_free(results);
258         (void) door_return((char *)&rv, sizeof (int), NULL, 0);
259 }
```

```
______unchanged_portion_omitted_
```

```
**********************************************************
   50673 Mon Feb 15 12:56:03 2016
new/usr/src/cmd/mv/mv.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

1868 /* Copy extended system attributes from source to target */

1870 static int
1871 copy_sysattr(char *source, char *target)
1872 {
1873         struct dirent   *dp;
1874         nvlist_t        *response;
1875         int             error = 0;
1876         int             target_sa_support = 0;

1878         if (sysattr_support(source, _PC_SATTR_EXISTS) != 1)
1879                 return (0);

1881         if (open_source(source) != 0)
1882                 return (1);

1884         /*
1885          * Gets non default extended system attributes from the
1886          * source file to copy to the target. The target has
1887          * the defaults set when its created and thus  no need
1888          * to copy the defaults.
1889          */
1890         response = sysattr_list(cmd, srcfd, source);

1892         if (sysattr_support(target, _PC_SATTR_ENABLED) != 1) {
1893                 if (response != NULL) {
1894                         (void) fprintf(stderr,
1895                             gettext(
1896                             "%s: cannot preserve extended system "
1897                             "attribute, operation not supported on file"
1898                             " %s\n"), cmd, target);
1899                         error++;
1900                         goto out;
1901                 }
1902         } else {
1903                 target_sa_support = 1;
1904         }

1906         if (target_sa_support) {
1907                 if (srcdirp == NULL) {
1908                         if (open_target_srctarg_attrdirs(source,
1909                             target) !=  0) {
1910                                 error++;
1911                                 goto out;
1912                         }
1913                         if (open_attrdirp(source) != 0) {
1914                                 error++;
1915                                 goto out;
1916                         }
1917                 } else {
1918                         rewind_attrdir(srcdirp);
1919                 }
1920                 while ((dp = readdir(srcdirp)) != NULL) {
1921                         nvlist_t        *res;
1922                         int             ret;

1924                         if ((ret = traverse_attrfile(dp, source, target,
1925                             0)) == -1)
1926                                 continue;
```



```
1927                         else if (ret > 0) {
1928                                 ++error;
1929                                 goto out;
1930                         }
1931                         /*
1932                          * Gets non default extended system attributes from the
1933                          * attribute file to copy to the target. The target has
1934                          * the defaults set when its created and thus  no need
1935                          * to copy the defaults.
1936                          */
1937                         if (dp->d_name != NULL) {
1938                                 res = sysattr_list(cmd, srcattrfd, dp->d_name);
1939                                 if (res == NULL)
1940                                         goto next;

1942                         /*
1943                          * Copy non default extended system attributes of named
1944                          * attribute file.
1945                          */
1946                                 if (fsetattr(targattrfd,
1947                                     XATTR_VIEW_READWRITE, res) != 0) {
1948                                         ++error;
1949                                         (void) fprintf(stderr, gettext("%s: "
1950                                             "Failed to copy extended system "
1951                                             "attributes from attribute file "
1952                                             "%s of %s to %s\n"), cmd,
1953                                             dp->d_name, source, target);
1954                                 }
1955                         }
1956 next:
1957                         if (srcattrfd != -1)
1958                                 (void) close(srcattrfd);
1959                         if (targattrfd != -1)
1960                                 (void) close(targattrfd);
1961                         srcattrfd = targattrfd = -1;
1962                         if (res != NULL)
1962                         nvlist_free(res);
1963                 }
1964         }
1965         /* Copy source file non default extended system attributes to target */
1966         if (target_sa_support && (response != NULL) &&
1967             (fsetattr(targfd, XATTR_VIEW_READWRITE, response)) != 0) {
1968                 ++error;
1969                 (void) fprintf(stderr, gettext("%s: Failed to "
1970                     "copy extended system attributes from "
1971                     "%s to %s\n"), cmd, source, target);
1972         }
1973 out:
1975         if (response != NULL)
1974         nvlist_free(response);
1975         close_all();
1976         return (error == 0 ? 0 : 1);
1977 }
______unchanged_portion_omitted_
```

```
**********************************************************
   16318 Mon Feb 15 12:56:03 2016
new/usr/src/cmd/pcidr/pcidr.c
patch tsoome-feedback
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License, Version 1.0 only
   6  * (the "License").  You may not use this file except in compliance
   7  * with the License.
   8  *
   9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  10  * or http://www.opensolaris.org/os/licensing.
  11  * See the License for the specific language governing permissions
  12  * and limitations under the License.
  13  *
  14  * When distributing Covered Code, include this CDDL HEADER in each
  15  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  16  * If applicable, add the following below this CDDL HEADER, with the
  17  * fields enclosed by brackets "[]" replaced with your own identifying
  18  * information: Portions Copyright [yyyy] [name of copyright owner]
  19  *
  20  * CDDL HEADER END
  21  */
  22 /*
  23  * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
  24  * Use is subject to license terms.
  25  */

  27 #pragma ident   "%Z%%M% %I%     %E% SMI"

  27 #include <stdio.h>
  28 #include <stdlib.h>
  29 #include <unistd.h>
  30 #include <strings.h>
  31 #include <string.h>
  32 #include <errno.h>
  33 #include <sys/param.h>
  34 #include <sys/systeminfo.h>
  35 #include <sys/sysevent/eventdefs.h>
  36 #include <sys/sysevent/dr.h>
  37 #include <syslog.h>
  38 #include <libnvpair.h>
  39 #include <stdarg.h>
  40 #include <assert.h>
  41 #include <sys/stat.h>
  42 #include <dlfcn.h>
  43 #include <signal.h>
  44 #include <pcidr.h>

  46 /*
  47  * pcidr takes in arguments of the form specified in the help() routine
  48  * including a set of name=value pairs, then looks up a plugin (shared object)
  49  * based on <plugin_paths> and however find_plugin() operates.  The entry
  50  * point of the plugin is <PCIDR_PLUGIN_SYM> and has the type
  51  * <pcidr_plugin_t>.  Plugins must use the <PCIDR_PLUGIN_PROTO> macro to
  52  * define their entry point.
  53  *
  54  * The name=value arguments are intended to be used as a mechanism to pass
  55  * arbitrary sysevent attributes using the macro expansion capability provided
  56  * by the sysseventd SLM processing sysevent.conf files (i.e. specifying
  57  * "$attribute" arguments for the handler in a .conf file entry). They are
  58  * converted into an nvlist_t (see libnvpair(3LIB)) by converting the values
  59  * of recognized names into appropriate types using pcidr_name2type() and
```



```
  60  * leaving all others as string types. Because pcidr is used as a sysevent.conf
  61  * handler, the format of the value string for non-string attributes in each
  62  * name=value argument must match that used by the sysseventd macro capability
  63  *
  64  * The plugin will be passed this (nvlist_t *) along with a (pcidr_opt_t *) arg
  65  * for other options.  While pcidr does some basic checking of arguments, it
  66  * leaves any name=value check (after conversion) up to each plugin.  Note
  67  * that pcidr_check_attrs() is used by the default plugin and can be used by
  68  * any plugin that support the same or a superset of its attributes.  If the
  69  * default plugin supports additional publishers, it should be updated in
  70  * pcidr_check_attrs().
  71  *
  72  * See help() for an example of how pcidr can be specified in a sysevent.conf
  73  * file.
  74  */

  76 /*
  77  * plugin search paths (searched in order specified);
  78  * macros begin MACRO_BEGTOK and end with MACRO_ENDTOK;
  79  *
  80  * be sure to update parse_path() and its support functions whenever macros
  81  * are updated e.g. si_name2cmd(), as well as substring tokens (prefix or
  82  * suffix) used to recognize different types of macros e.g. SI_MACRO
  83  *
  84  * NOTE: if plugin search algorithm is changed starting with find_plugin(),
  85  * please update documentation here.
  86  *
  87  * macros:
  88  * SI_PLATFORM = cmd of same name in sysinfo(2)
  89  * SI_MACHINE = cmd of same name in sysinfo(2)
  90  */
  91 #define MACRO_BEGTOK    "${"
  92 #define MACRO_ENDTOK    "}"
  93 #define SI_MACRO        "SI_"

  95 static char *plugin_paths[] = {
  96         "/usr/platform/${SI_PLATFORM}/lib/pci/" PCIDR_PLUGIN_NAME,
  97         "/usr/platform/${SI_MACHINE}/lib/pci/" PCIDR_PLUGIN_NAME,
  98         "/usr/lib/pci/" PCIDR_PLUGIN_NAME,
  99 };
_____unchanged_portion_omitted_


 196 /*
 197  * argc: length of argv
 198  * argv: each string starting from index <argip> has the format "name=value"
 199  * argip: starting index in <argv>; also used to return ending index
 200  *
 201  * return: allocated nvlist on success, exits otherwise
 202  *
 203  * recognized names will have predetermined types, while all others will have
 204  * values of type string
 205  */
 206 static nvlist_t *
 207 parse_argv_attr(int argc, char **argv, int *argip)
 208 {
 209         char *fn = "parse_argv_attr";
 210         int rv, i;
 211         nvlist_t *attrlistp = NULL;
 212         char *eqp, *name, *value;
 213         data_type_t type;

 215         assert(*argip < argc);

 217         rv = nvlist_alloc(&attrlistp, NV_UNIQUE_NAME_TYPE, 0);
 218         if (rv != 0) {
```

```
219                 dprint(DDEBUG, "%s: nvlist_alloc() failed: rv = %d\n", fn, rv);
220                 goto ERR;
221         }

223         for (i = *argip; i < argc; i++) {
224                 eqp = strchr(argv[i], '=');
225                 if (eqp == NULL)
226                         goto ERR_ARG;
227                 *eqp = '\0';
228                 name = argv[i];
229                 value = eqp;
230                 value++;
231                 if (*name == '\0' || *value == '\0')
232                         goto ERR_ARG;

234                 if (pcidr_name2type(name, &type) != 0)
235                         type = DATA_TYPE_STRING;

237                 rv = nvadd(attrlistp, name, value, type);
238                 if (rv != 0) {
239                         dprint(DDEBUG, "%s: nvadd() failed: attribute \"%s\", "
240                             "value = %s, type = %d, rv = %d\n",
241                             fn, name, value, (int)type, rv);
242                         goto ERR;
243                 }
244                 *eqp = '=';
245         }

247         *argip = i;
248         return (attrlistp);

250         /*NOTREACHED*/
251 ERR_ARG:
252         if (eqp != NULL)
253                 *eqp = '=';
254         dprint(DDEBUG, "%s: bad attribute argv[%d]: \"%s\"\n", fn, i, argv[i]);
255 ERR:
258         if (attrlistp != NULL)
256         nvlist_free(attrlistp);
257         return (NULL);
258 }
```

unchanged_portion_omitted_

```
**********************************************************
   96450 Mon Feb 15 12:56:03 2016
new/usr/src/cmd/picl/plugins/common/devtree/picldevtree.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

3596 /*
3597  * This function is the event handler of this plug-in.
3598  *
3599  * It processes the following events:
3600  *
3601  *      PICLEVENT_SYSEVENT_DEVICE_ADDED
3602  *      PICLEVENT_SYSEVENT_DEVICE_REMOVED
3603  *      PICLEVENT_CPU_STATE_CHANGE
3604  *      PICLEVENT_DR_AP_STATE_CHANGE
3605  */
3606 /* ARGSUSED */
3607 static void
3608 picldevtree_evhandler(const char *ename, const void *earg, size_t size,
3609     void *cookie)
3610 {
3611         char                    *devfs_path;
3612         char                    ptreepath[PATH_MAX];
3613         char                    dipath[PATH_MAX];
3614         picl_nodehdl_t          plafh;
3615         picl_nodehdl_t          nodeh;
3616         nvlist_t                *nvlp;

3618         if ((earg == NULL) ||
3619             (ptree_get_node_by_path(PLATFORM_PATH, &plafh) != PICL_SUCCESS))
3620                 return;

3622         if (strcmp(ename, PICLEVENT_DR_AP_STATE_CHANGE) == 0) {
3623                 (void) setup_cpus(plafh);
3624                 if (picldevtree_debug > 1)
3625                         syslog(LOG_INFO, "picldevtree: event handler done\n");
3626                 return;
3627         }

3629         nvlp = NULL;
3630         if (nvlist_unpack((char *)earg, size, &nvlp, NULL) ||
3631             nvlist_lookup_string(nvlp, PICLEVENTARG_DEVFS_PATH, &devfs_path) ||
3632             strlen(devfs_path) > (PATH_MAX - sizeof (PLATFORM_PATH))) {
3633                 syslog(LOG_INFO, PICL_EVENT_DROPPED, ename);
3634                 if (nvlp)
3634                 nvlist_free(nvlp);
3635                 return;
3636         }

3638         (void) strlcpy(ptreepath, PLATFORM_PATH, PATH_MAX);
3639         (void) strlcat(ptreepath, devfs_path, PATH_MAX);
3640         (void) strlcpy(dipath, devfs_path, PATH_MAX);
3641         nvlist_free(nvlp);

3643         if (picldevtree_debug)
3644                 syslog(LOG_INFO, "picldevtree: event handler invoked ename:%s "
3645                     "ptreepath:%s\n", ename, ptreepath);

3647         if (strcmp(ename, PICLEVENT_CPU_STATE_CHANGE) == 0) {
3648                 goto done;
3649         }
3650         if (strcmp(ename, PICLEVENT_SYSEVENT_DEVICE_ADDED) == 0) {
3651                 di_node_t               devnode;
3652                 char            *strp;
3653                 picl_nodehdl_t  parh;
```



```
3654                 char            nodeclass[PICL_CLASSNAMELEN_MAX];
3655                 char            *nodename;
3656                 int             err;

3658                 /* If the node already exist, then nothing else to do here */
3659                 if (ptree_get_node_by_path(ptreepath, &nodeh) == PICL_SUCCESS)
3660                         return;

3662                 /* Skip if unable to find parent PICL node handle */
3663                 parh = plafh;
3664                 if (((strp = strrchr(ptreepath, '/')) != NULL) &&
3665                     (strp != strchr(ptreepath, '/'))) {
3666                         *strp = '\0';
3667                         if (ptree_get_node_by_path(ptreepath, &parh) !=
3668                             PICL_SUCCESS)
3669                                 return;
3670                 }

3672                 /*
3673                  * If parent is the root node
3674                  */
3675                 if (parh == plafh) {
3676                         ph = di_prom_init();
3677                         devnode = di_init(dipath, DINFOCPYALL);
3678                         if (devnode == DI_NODE_NIL) {
3679                                 if (ph != NULL) {
3680                                         di_prom_fini(ph);
3681                                         ph = NULL;
3682                                 }
3683                                 return;
3684                         }
3685                         nodename = di_node_name(devnode);
3686                         if (nodename == NULL) {
3687                                 di_fini(devnode);
3688                                 if (ph != NULL) {
3689                                         di_prom_fini(ph);
3690                                         ph = NULL;
3691                                 }
3692                                 return;
3693                         }

3695                         err = get_node_class(nodeclass, devnode, nodename);
3696                         if (err < 0) {
3697                                 di_fini(devnode);
3698                                 if (ph != NULL) {
3699                                         di_prom_fini(ph);
3700                                         ph = NULL;
3701                                 }
3702                                 return;
3703                         }
3704                         err = construct_devtype_node(plafh, nodename,
3705                             nodeclass, devnode, &nodeh);
3706                         if (err != PICL_SUCCESS) {
3707                                 di_fini(devnode);
3708                                 if (ph != NULL) {
3709                                         di_prom_fini(ph);
3710                                         ph = NULL;
3711                                 }
3712                                 return;
3713                         }
3714                         (void) update_subtree(nodeh, devnode);
3715                         (void) add_unitaddr_prop_to_subtree(nodeh);
3716                         if (ph != NULL) {
3717                                 di_prom_fini(ph);
3718                                 ph = NULL;
3719                         }
```

```
3720                        di_fini(devnode);
3721                        goto done;
3722                }

3724                /* kludge ... try without bus-addr first */
3725                if ((strp = strrchr(dipath, '@')) != NULL) {
3726                        char *p;

3728                        p = strrchr(dipath, '/');
3729                        if (p != NULL && strp > p) {
3730                                *strp = '\0';
3731                                devnode = di_init(dipath, DINFOCPYALL);
3732                                if (devnode != DI_NODE_NIL)
3733                                        di_fini(devnode);
3734                                *strp = '@';
3735                        }
3736                }
3737                /* Get parent devnode */
3738                if ((strp = strrchr(dipath, '/')) != NULL)
3739                        *++strp = '\0';
3740                devnode = di_init(dipath, DINFOCPYALL);
3741                if (devnode == DI_NODE_NIL)
3742                        return;
3743                ph = di_prom_init();
3744                (void) update_subtree(parh, devnode);
3745                (void) add_unitaddr_prop_to_subtree(parh);
3746                if (ph) {
3747                        di_prom_fini(ph);
3748                        ph = NULL;
3749                }
3750                di_fini(devnode);
3751        } else if (strcmp(ename, PICLEVENT_SYSEVENT_DEVICE_REMOVED) == 0) {
3752                char                    delclass[PICL_CLASSNAMELEN_MAX];
3753                char            *strp;

3755                /*
3756                 * if final element of path doesn't have a unit address
3757                 * then it is not uniquely identifiable - cannot remove
3758                 */
3759                if (((strp = strrchr(ptreepath, '/')) != NULL) &&
3760                    strchr(strp, '@') == NULL)
3761                        return;

3763                /* skip if can't find the node */
3764                if (ptree_get_node_by_path(ptreepath, &nodeh) != PICL_SUCCESS)
3765                        return;

3767                if (ptree_delete_node(nodeh) != PICL_SUCCESS)
3768                        return;

3770                if (picldevtree_debug)
3771                        syslog(LOG_INFO,
3772                            "picldevtree: deleted node nodeh:%llx\n", nodeh);
3773                if ((ptree_get_propval_by_name(nodeh,
3774                    PICL_PROP_CLASSNAME, delclass, PICL_CLASSNAMELEN_MAX) ==
3775                    PICL_SUCCESS) && IS_MC(delclass)) {
3776                        if (post_mc_event(PICLEVENT_MC_REMOVED, nodeh) !=
3777                            PICL_SUCCESS)
3778                                syslog(LOG_WARNING, PICL_EVENT_DROPPED,
3779                                    PICLEVENT_MC_REMOVED);
3780                } else
3781                        (void) ptree_destroy_node(nodeh);
3782        }
3783 done:
3784        (void) setup_cpus(plafh);
3785        (void) add_ffb_config_info(plafh);
```



```
3786        set_pci_pciex_deviceid(plafh);
3787        (void) set_sbus_slot(plafh);
3788        if (picldevtree_debug > 1)
3789                syslog(LOG_INFO, "picldevtree: event handler done\n");
3790 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10973 Mon Feb 15 12:56:03 2016
new/usr/src/cmd/picl/plugins/sun4v/piclsbl/piclsbl.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

193 /*
194  * Ontario SBL event handler, subscribed to:
195  *      PICLEVENT_SYSEVENT_DEVICE_ADDED
196  *      PICLEVENT_SYSEVENT_DEVICE_REMOVED
197  */
198 static void
199 piclsbl_handler(const char *ename, const void *earg, size_t size,
200                 void *cookie)
201 {
202         char            *devfs_path;
203         char            hdd_location[PICL_PROPNAMELEN_MAX];
204         nvlist_t        *nvlp = NULL;
205         pcp_msg_t       send_msg;
206         pcp_msg_t       recv_msg;
207         pcp_sbl_req_t   *req_ptr = NULL;
208         pcp_sbl_resp_t  *resp_ptr = NULL;
209         int             status = -1;
210         int             target;
211         disk_lookup_t   lookup;
212         int             channel_fd;

214         /*
215          * setup the request data to attach to the libpcp msg
216          */
217         if ((req_ptr = (pcp_sbl_req_t *)umem_zalloc(sizeof (pcp_sbl_req_t),
218             UMEM_DEFAULT)) == NULL)
219                 goto sbl_return;

221         /*
222          * This plugin serves to enable or disable the blue RAS
223          * 'ok-to-remove' LED that is on each of the 4 disks on the
224          * Ontario.  We catch the event via the picl handler, and
225          * if the event is DEVICE_ADDED for one of our onboard disks,
226          * then we'll be turning off the LED. Otherwise, if the event
227          * is DEVICE_REMOVED, then we turn it on.
228          */
229         if (strcmp(ename, PICLEVENT_SYSEVENT_DEVICE_ADDED) == 0)
230                 req_ptr->sbl_action = PCP_SBL_DISABLE;
231         else if (strcmp(ename, PICLEVENT_SYSEVENT_DEVICE_REMOVED) == 0)
232                 req_ptr->sbl_action = PCP_SBL_ENABLE;
233         else
234                 goto sbl_return;

236         /*
237          * retrieve the device's physical path from the event payload
238          */
239         if (nvlist_unpack((char *)earg, size, &nvlp, NULL))
240                 goto sbl_return;
241         if (nvlist_lookup_string(nvlp, "devfs-path", &devfs_path))
242                 goto sbl_return;

244         /*
245          * look for this disk in the picl tree, and if it's
246          * location indicates that it's one of our internal
247          * disks, then set sbl_id to incdicate which one.
248          * otherwise, return as it is not one of our disks.
249          */
250         lookup.path = strdup(devfs_path);
251         lookup.disk = NULL;
```



```
252         lookup.result = DISK_NOT_FOUND;

254         /* first, find the disk */
255         status = ptree_walk_tree_by_class(root_node, "disk", (void *)&lookup,
256             cb_find_disk);
257         if (status != PICL_SUCCESS)
258                 goto sbl_return;

260         if (lookup.result == DISK_FOUND) {
261                 /* now, lookup it's location in the node */
262                 status = ptree_get_propval_by_name(lookup.disk, "Location",
263                     (void *)&hdd_location, PICL_PROPNAMELEN_MAX);
264                 if (status != PICL_SUCCESS) {
265                         syslog(LOG_ERR, "piclsbl: failed hdd discovery");
266                         goto sbl_return;
267                 }
268         }

270         /*
271          * Strip off the target from the NAC name.
272          * The disk NAC will always be HDD#
273          */
274         if (strncmp(hdd_location, NAC_DISK_PREFIX,
275             strlen(NAC_DISK_PREFIX)) == 0) {
276                 (void) sscanf(hdd_location, "%*3s%d", &req_ptr->sbl_id);
277                 target = (int)req_ptr->sbl_id;
278         } else {
279                 /* this is not one of the onboard disks */
280                 goto sbl_return;
281         }

283         /*
284          * check the onboard RAID configuration for this disk. if it is
285          * a member of a RAID and is not the RAID itself, ignore the event
286          */
287         if (check_raid(target))
288                 goto sbl_return;

290         /*
291          * we have the information we need, init the platform channel.
292          * the platform channel driver will only allow one connection
293          * at a time on this socket. on the offchance that more than
294          * one event comes in, we'll retry to initialize this connection
295          * up to 3 times
296          */
297         if ((channel_fd = (*pcp_init_ptr)(LED_CHANNEL)) < 0) {
298                 /* failed to init; wait and retry up to 3 times */
299                 int s = PCPINIT_TIMEOUT;
300                 int retries = 0;
301                 while (++retries) {
302                         (void) sleep(s);
303                         if ((channel_fd = (*pcp_init_ptr)(LED_CHANNEL)) >= 0)
304                                 break;
305                         else if (retries == 3) {
306                                 syslog(LOG_ERR, "piclsbl: ",
307                                     "SC channel initialization failed");
308                                 goto sbl_return;
309                         }
310                         /* continue */
311                 }
312         }

314         /*
315          * populate the message for libpcp
316          */
317         send_msg.msg_type = PCP_SBL_CONTROL;
```

```
318         send_msg.sub_type = NULL;
319         send_msg.msg_len = sizeof (pcp_sbl_req_t);
320         send_msg.msg_data = (uint8_t *)req_ptr;

322         /*
323          * send the request, receive the response
324          */
325         if ((*pcp_send_recv_ptr)(channel_fd, &send_msg, &recv_msg,
326             PCPCOMM_TIMEOUT) < 0) {
327                 /* we either timed out or erred; either way try again */
328                 int s = PCPCOMM_TIMEOUT;
329                 (void) sleep(s);
330                 if ((*pcp_send_recv_ptr)(channel_fd, &send_msg, &recv_msg,
331                     PCPCOMM_TIMEOUT) < 0) {
332                         syslog(LOG_ERR, "piclsbl: communication failure");
333                         goto sbl_return;
334                 }
335         }

337         /*
338          * validate that this data was meant for us
339          */
340         if (recv_msg.msg_type != PCP_SBL_CONTROL_R) {
341                 syslog(LOG_ERR, "piclsbl: unbound packet received");
342                 goto sbl_return;
343         }

345         /*
346          * verify that the LED action has taken place
347          */
348         resp_ptr = (pcp_sbl_resp_t *)recv_msg.msg_data;
349         if (resp_ptr->status == PCP_SBL_ERROR) {
350                 syslog(LOG_ERR, "piclsbl: OK2RM LED action error");
351                 goto sbl_return;
352         }

354         /*
355          * ensure the LED action taken is the one requested
356          */
357         if ((req_ptr->sbl_action == PCP_SBL_DISABLE) &&
358             (resp_ptr->sbl_state != SBL_STATE_OFF))
359                 syslog(LOG_ERR, "piclsbl: OK2RM LED not OFF after disk "
360                     "configuration");
361         else if ((req_ptr->sbl_action == PCP_SBL_ENABLE) &&
362             (resp_ptr->sbl_state != SBL_STATE_ON))
363                 syslog(LOG_ERR, "piclsbl: OK2RM LED not ON after disk "
364                     "unconfiguration");
365         else if (resp_ptr->sbl_state == SBL_STATE_UNKNOWN)
366                 syslog(LOG_ERR, "piclsbl: OK2RM LED set to unknown state");

368 sbl_return:

370         (*pcp_close_ptr)(channel_fd);
371         if (req_ptr != NULL)
372                 umem_free(req_ptr, sizeof (pcp_sbl_req_t));
373         if (resp_ptr != NULL)
374                 free(resp_ptr);
375         if (nvlp != NULL)
375         nvlist_free(nvlp);
376 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   38605 Mon Feb 15 12:56:03 2016
new/usr/src/cmd/rcm_daemon/common/aggr_rcm.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

1256 /*
1257  * Send RCM_RESOURCE_LINK_NEW events to other modules about new aggregations.
1258  * Return 0 on success, -1 on failure.
1259  */
1260 static int
1261 aggr_notify_new_aggr(rcm_handle_t *hd, char *rsrc)
1262 {
1263         link_cache_t *node;
1264         dl_aggr_t *aggr;
1265         nvlist_t *nvl = NULL;
1266         uint64_t id;
1267         boolean_t is_only_port;
1268         int ret = -1;

1270         rcm_log_message(RCM_TRACE2, "AGGR: aggr_notify_new_aggr (%s)\n", rsrc);

1272         /* Check for the interface in the cache */
1273         (void) mutex_lock(&cache_lock);
1274         if ((node = cache_lookup(hd, rsrc, CACHE_REFRESH)) == NULL) {
1275                 rcm_log_message(RCM_TRACE1,
1276                     "AGGR: aggr_notify_new_aggr() unrecognized resource (%s)\n",
1277                     rsrc);
1278                 (void) mutex_unlock(&cache_lock);
1279                 return (0);
1280         }

1282         if (nvlist_alloc(&nvl, 0, 0) != 0) {
1283                 rcm_log_message(RCM_WARNING,
1284                     _("AGGR: failed to allocate nvlist\n"));
1285                 (void) mutex_unlock(&cache_lock);
1286                 goto done;
1287         }

1289         aggr = node->vc_aggr;
1290         is_only_port = (aggr->da_lastport == node->vc_linkid);

1292         if (is_only_port) {
1293                 rcm_log_message(RCM_TRACE2,
1294                     "AGGR: aggr_notify_new_aggr add (%u)\n",
1295                     aggr->da_aggrid);

1297                 id = aggr->da_aggrid;
1298                 if (nvlist_add_uint64(nvl, RCM_NV_LINKID, id) != 0) {
1299                         rcm_log_message(RCM_ERROR,
1300                             _("AGGR: failed to construct nvlist\n"));
1301                         (void) mutex_unlock(&cache_lock);
1302                         goto done;
1303                 }
1304         }

1306         (void) mutex_unlock(&cache_lock);

1308         /*
1309          * If this link is not the only port in the aggregation, the aggregation
1310          * is not new. No need to inform other consumers in that case.
1311          */
1312         if (is_only_port && rcm_notify_event(hd, RCM_RESOURCE_LINK_NEW,
1313             0, nvl, NULL) != RCM_SUCCESS) {
1314                 rcm_log_message(RCM_ERROR,
```



```
1315                     _("AGGR: failed to notify %s event for %s\n"),
1316                     RCM_RESOURCE_LINK_NEW, node->vc_resource);
1317                 goto done;
1318         }

1320         ret = 0;
1321 done:
1322         if (nvl != NULL)
1322         nvlist_free(nvl);
1323         return (ret);
1324 }
______unchanged_portion_omitted_
```

```
**********************************************************
   35975 Mon Feb 15 12:56:03 2016
new/usr/src/cmd/rcm_daemon/common/ibpart_rcm.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

1197 /*
1198  * Send RCM_RESOURCE_LINK_NEW events to other modules about new IBPARTs.
1199  * Return 0 on success, -1 on failure.
1200  */
1201 static int
1202 ibpart_notify_new_ibpart(rcm_handle_t *hd, char *rsrc)
1203 {
1204         link_cache_t *node;
1205         dl_ibpart_t *ibpart;
1206         nvlist_t *nvl = NULL;
1207         uint64_t id;
1208         int ret = -1;

1210         rcm_log_message(RCM_TRACE2, "IBPART: ibpart_notify_new_ibpart (%s)\n",
1211             rsrc);

1213         (void) mutex_lock(&cache_lock);
1214         if ((node = cache_lookup(hd, rsrc, CACHE_REFRESH)) == NULL) {
1215                 (void) mutex_unlock(&cache_lock);
1216                 return (0);
1217         }

1219         if (nvlist_alloc(&nvl, 0, 0) != 0) {
1220                 (void) mutex_unlock(&cache_lock);
1221                 rcm_log_message(RCM_WARNING,
1222                     _("IBPART: failed to allocate nvlist\n"));
1223                 goto done;
1224         }

1226         for (ibpart = node->pc_ibpart; ibpart != NULL;
1227             ibpart = ibpart->dlib_next) {
1228                 rcm_log_message(RCM_TRACE2, "IBPART: ibpart_notify_new_ibpart "
1229                     "add (%u)\n", ibpart->dlib_ibpart_id);

1231                 id = ibpart->dlib_ibpart_id;
1232                 if (nvlist_add_uint64(nvl, RCM_NV_LINKID, id) != 0) {
1233                         rcm_log_message(RCM_ERROR,
1234                             _("IBPART: failed to construct nvlist\n"));
1235                         (void) mutex_unlock(&cache_lock);
1236                         goto done;
1237                 }
1238         }
1239         (void) mutex_unlock(&cache_lock);

1241         if (rcm_notify_event(hd, RCM_RESOURCE_LINK_NEW, 0, nvl, NULL) !=
1242             RCM_SUCCESS) {
1243                 rcm_log_message(RCM_ERROR,
1244                     _("IBPART: failed to notify %s event for %s\n"),
1245                     RCM_RESOURCE_LINK_NEW, node->pc_resource);
1246                 goto done;
1247         }

1249         ret = 0;
1250 done:
1251         if (nvl != NULL)
1251         nvlist_free(nvl);
1252         return (ret);
1253 }
______unchanged_portion_omitted_
```

****
34391 Mon Feb 15 12:56:03 2016
new/usr/src/cmd/rcm_daemon/common/vlan_rcm.c
patch tsoome-feedback
****
_____***unchanged_portion_omitted_***

```
1166 /*
1167  * Send RCM_RESOURCE_LINK_NEW events to other modules about new VLANs.
1168  * Return 0 on success, -1 on failure.
1169  */
1170 static int
1171 vlan_notify_new_vlan(rcm_handle_t *hd, char *rsrc)
1172 {
1173         link_cache_t *node;
1174         dl_vlan_t *vlan;
1175         nvlist_t *nvl = NULL;
1176         uint64_t id;
1177         int ret = -1;

1179         rcm_log_message(RCM_TRACE2, "VLAN: vlan_notify_new_vlan (%s)\n", rsrc);

1181         (void) mutex_lock(&cache_lock);
1182         if ((node = cache_lookup(hd, rsrc, CACHE_REFRESH)) == NULL) {
1183                 (void) mutex_unlock(&cache_lock);
1184                 return (0);
1185         }

1187         if (nvlist_alloc(&nvl, 0, 0) != 0) {
1188                 (void) mutex_unlock(&cache_lock);
1189                 rcm_log_message(RCM_WARNING,
1190                     _("VLAN: failed to allocate nvlist\n"));
1191                 goto done;
1192         }

1194         for (vlan = node->vc_vlan; vlan != NULL; vlan = vlan->dv_next) {
1195                 rcm_log_message(RCM_TRACE2,
1196                     "VLAN: vlan_notify_new_vlan add (%u)\n",
1197                     vlan->dv_vlanid);

1199                 id = vlan->dv_vlanid;
1200                 if (nvlist_add_uint64(nvl, RCM_NV_LINKID, id) != 0) {
1201                         rcm_log_message(RCM_ERROR,
1202                             _("VLAN: failed to construct nvlist\n"));
1203                         (void) mutex_unlock(&cache_lock);
1204                         goto done;
1205                 }
1206         }
1207         (void) mutex_unlock(&cache_lock);

1209         if (rcm_notify_event(hd, RCM_RESOURCE_LINK_NEW, 0, nvl, NULL) !=
1210             RCM_SUCCESS) {
1211                 rcm_log_message(RCM_ERROR,
1212                     _("VLAN: failed to notify %s event for %s\n"),
1213                     RCM_RESOURCE_LINK_NEW, node->vc_resource);
1214                 goto done;
1215         }

1217         ret = 0;
1218 done:
1219         if (nvl != NULL)
1219                 nvlist_free(nvl);
1220         return (ret);
1221 }
```

_____***unchanged_portion_omitted_***

****
34693 Mon Feb 15 12:56:04 2016
new/usr/src/cmd/rcm_daemon/common/vnic_rcm.c
patch tsoome-feedback
****
_____unchanged_portion_omitted_

```
1178 /*
1179  * Send RCM_RESOURCE_LINK_NEW events to other modules about new VNICs.
1180  * Return 0 on success, -1 on failure.
1181  */
1182 static int
1183 vnic_notify_new_vnic(rcm_handle_t *hd, char *rsrc)
1184 {
1185         link_cache_t *node;
1186         dl_vnic_t *vnic;
1187         nvlist_t *nvl = NULL;
1188         uint64_t id;
1189         int ret = -1;

1191         rcm_log_message(RCM_TRACE2, "VNIC: vnic_notify_new_vnic (%s)\n", rsrc);

1193         (void) mutex_lock(&cache_lock);
1194         if ((node = cache_lookup(hd, rsrc, CACHE_REFRESH)) == NULL) {
1195                 (void) mutex_unlock(&cache_lock);
1196                 return (0);
1197         }

1199         if (nvlist_alloc(&nvl, 0, 0) != 0) {
1200                 (void) mutex_unlock(&cache_lock);
1201                 rcm_log_message(RCM_WARNING,
1202                     _("VNIC: failed to allocate nvlist\n"));
1203                 goto done;
1204         }

1206         for (vnic = node->vc_vnic; vnic != NULL; vnic = vnic->dlv_next) {
1207                 rcm_log_message(RCM_TRACE2,
1208                     "VNIC: vnic_notify_new_vnic add (%u)\n", vnic->dlv_vnic_id);

1210                 id = vnic->dlv_vnic_id;
1211                 if (nvlist_add_uint64(nvl, RCM_NV_LINKID, id) != 0) {
1212                         rcm_log_message(RCM_ERROR,
1213                             _("VNIC: failed to construct nvlist\n"));
1214                         (void) mutex_unlock(&cache_lock);
1215                         goto done;
1216                 }
1217         }
1218         (void) mutex_unlock(&cache_lock);

1220         if (rcm_notify_event(hd, RCM_RESOURCE_LINK_NEW, 0, nvl, NULL) !=
1221             RCM_SUCCESS) {
1222                 rcm_log_message(RCM_ERROR,
1223                     _("VNIC: failed to notify %s event for %s\n"),
1224                     RCM_RESOURCE_LINK_NEW, node->vc_resource);
1225                 goto done;
1226         }

1228         ret = 0;
1229 done:
1230         if (nvl != NULL)
1230         nvlist_free(nvl);
1231         return (ret);
1232 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   53869 Mon Feb 15 12:56:04 2016
new/usr/src/cmd/syseventd/modules/sysevent_conf_mod/sysevent_conf_mod.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

1636 /*
1637  * For a matching event specification, build the command to be
1638  * invoked in response to the event.  Building the command involves
1639  * expanding macros supplied in the event specification command
1640  * with values from the actual event.  These macros can be
1641  * the class/subclass/vendor/publisher strings, or arbitrary
1642  * attribute data attached to the event.
1643  *
1644  * This module does not invoke (fork/exec) the command itself,
1645  * since this module is running in the context of the syseventd
1646  * daemon, and fork/exec's done here interfere with the door
1647  * upcall delivering events from the kernel to the daemon.
1648  * Instead, we build a separate event and nvlist with the
1649  * attributes of the command to be invoked, and pass that on
1650  * to the syseventconfd daemon, which is basically a fork/exec
1651  * server on our behalf.
1652  *
1653  * Errors queuing the event are returned to syseventd with
1654  * EAGAIN, allowing syseventd to manage a limited number of
1655  * retries after a short delay.
1656  */
1657 static int
1658 queue_event(sysevent_t *ev, syseventtab_t *sep, sysevent_hdr_info_t *hdr)
1659 {
1660         str_t           *line;
1661         nvlist_t        *nvlist;
1662         char            *argv0;
1663         sysevent_t      *cmd_event;
1664         nvlist_t        *cmd_nvlist;
1665         cmdqueue_t      *new_cmd;

1667         if ((line = initstr(128)) == NULL)
1668                 return (1);

1670         if ((argv0 = strrchr(sep->se_path, '/')) == NULL) {
1671                 argv0 = sep->se_path;
1672         } else {
1673                 argv0++;
1674         }
1675         if (strcopys(line, argv0)) {
1676                 freestr(line);
1677                 return (1);
1678         }

1680         if (sep->se_args) {
1681                 if (strcats(line, " ")) {
1682                         freestr(line);
1683                         return (1);
1684                 }
1685                 if (strcats(line, sep->se_args)) {
1686                         freestr(line);
1687                         return (1);
1688                 }

1690                 if (sysevent_get_attr_list(ev, &nvlist) != 0) {
1691                         syslog(LOG_ERR, GET_ATTR_LIST_ERR,
1692                                 sep->se_conf_file, sep->se_lineno,
1693                                 strerror(errno));
1694                         freestr(line);
```



```
1695                         return (1);
1696                 }
1697                 if (expand_macros(ev, nvlist, sep, line, hdr)) {
1698                         freestr(line);
1699                         if (nvlist)
1699                         nvlist_free(nvlist);
1700                         return (1);
1701                 }
1703                 if (nvlist)
1702                 nvlist_free(nvlist);
1703         }

1705         if (debug_level >= DBG_EXEC) {
1706                 syseventd_print(DBG_EXEC, "%s, line %d: path = %s\n",
1707                         sep->se_conf_file, sep->se_lineno, sep->se_path);
1708                 syseventd_print(DBG_EXEC, "    cmd = %s\n", line->s_str);
1709         }

1711         cmd_nvlist = NULL;
1712         if ((errno = nvlist_alloc(&cmd_nvlist, NV_UNIQUE_NAME, 0)) != 0) {
1713                 freestr(line);
1714                 syslog(LOG_ERR, NVLIST_ALLOC_ERR,
1715                         sep->se_conf_file, sep->se_lineno,
1716                         strerror(errno));
1717                 return (1);
1718         }

1720         if ((errno = nvlist_add_string(cmd_nvlist, "path", sep->se_path)) != 0)
1721                 goto err;
1722         if ((errno = nvlist_add_string(cmd_nvlist, "cmd", line->s_str)) != 0)
1723                 goto err;
1724         if ((errno = nvlist_add_string(cmd_nvlist, "file",
1725             sep->se_conf_file)) != 0)
1726                 goto err;
1727         if ((errno = nvlist_add_int32(cmd_nvlist, "line", sep->se_lineno)) != 0)
1728                 goto err;
1729         if ((errno = nvlist_add_string(cmd_nvlist, "user", sep->se_user)) != 0)
1730                 goto err;

1732         if (sep->se_uid != (uid_t)0) {
1733                 if ((errno = nvlist_add_int32(cmd_nvlist, "uid",
1734                     sep->se_uid)) != 0)
1735                         goto err;
1736                 if ((errno = nvlist_add_int32(cmd_nvlist, "gid",
1737                     sep->se_gid)) != 0)
1738                         goto err;
1739         }

1741         cmd_event = sysevent_alloc_event(hdr->class, hdr->subclass, hdr->vendor,
1742                 hdr->publisher, cmd_nvlist);
1743         if (cmd_event == NULL) {
1744                 syslog(LOG_ERR, SYSEVENT_ALLOC_ERR,
1745                         sep->se_conf_file, sep->se_lineno,
1746                         strerror(errno));
1747                 nvlist_free(cmd_nvlist);
1748                 freestr(line);
1749                 return (1);
1750         }

1752         nvlist_free(cmd_nvlist);
1753         freestr(line);

1755         /*
1756          * Place cmd_event on queue to be transported to syseventconfd
1757          */
1758         if ((new_cmd = sc_malloc(sizeof (cmdqueue_t))) == NULL) {
```

```
1759                    sysevent_free(cmd_event);
1760                    return (1);
1761            }
1762            new_cmd->event = cmd_event;
1763            new_cmd->next = NULL;
1764            (void) mutex_lock(&cmdq_lock);
1765            if (cmdq == NULL) {
1766                    cmdq = new_cmd;
1767            } else {
1768                    cmdq_tail->next = new_cmd;
1769            }
1770            cmdq_cnt++;
1771            cmdq_tail = new_cmd;

1773            /*
1774             * signal queue flush thread
1775             */
1776            (void) cond_signal(&cmdq_cv);

1778            (void) mutex_unlock(&cmdq_lock);

1780            return (0);

1782 err:
1783            syslog(LOG_ERR, NVLIST_BUILD_ERR,
1784                    sep->se_conf_file, sep->se_lineno, strerror(errno));
1785            nvlist_free(cmd_nvlist);
1786            freestr(line);
1787            return (1);
1788 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  171250 Mon Feb 15 12:56:04 2016
new/usr/src/cmd/zfs/zfs_main.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

5254 static int
5255 zfs_do_allow_unallow_impl(int argc, char **argv, boolean_t un)
5256 {
5257         zfs_handle_t *zhp;
5258         nvlist_t *perm_nvl = NULL;
5259         nvlist_t *update_perm_nvl = NULL;
5260         int error = 1;
5261         int c;
5262         struct allow_opts opts = { 0 };

5264         const char *optstr = un ? "ldugecsrh" : "ldugecsh";

5266         /* check opts */
5267         while ((c = getopt(argc, argv, optstr)) != -1) {
5268                 switch (c) {
5269                 case 'l':
5270                         opts.local = B_TRUE;
5271                         break;
5272                 case 'd':
5273                         opts.descend = B_TRUE;
5274                         break;
5275                 case 'u':
5276                         opts.user = B_TRUE;
5277                         break;
5278                 case 'g':
5279                         opts.group = B_TRUE;
5280                         break;
5281                 case 'e':
5282                         opts.everyone = B_TRUE;
5283                         break;
5284                 case 's':
5285                         opts.set = B_TRUE;
5286                         break;
5287                 case 'c':
5288                         opts.create = B_TRUE;
5289                         break;
5290                 case 'r':
5291                         opts.recursive = B_TRUE;
5292                         break;
5293                 case ':':
5294                         (void) fprintf(stderr, gettext("missing argument for "
5295                             "'%c' option\n"), optopt);
5296                         usage(B_FALSE);
5297                         break;
5298                 case 'h':
5299                         opts.prt_usage = B_TRUE;
5300                         break;
5301                 case '?':
5302                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
5303                             optopt);
5304                         usage(B_FALSE);
5305                 }
5306         }

5308         argc -= optind;
5309         argv += optind;

5311         /* check arguments */
5312         parse_allow_args(argc, argv, un, &opts);
```



```
5314         /* try to open the dataset */
5315         if ((zhp = zfs_open(g_zfs, opts.dataset, ZFS_TYPE_FILESYSTEM |
5316             ZFS_TYPE_VOLUME)) == NULL) {
5317                 (void) fprintf(stderr, "Failed to open dataset: %s\n",
5318                     opts.dataset);
5319                 return (-1);
5320         }

5322         if (zfs_get_fsacl(zhp, &perm_nvl) != 0)
5323                 goto cleanup2;

5325         fs_perm_set_init(&fs_perm_set);
5326         if (parse_fs_perm_set(&fs_perm_set, perm_nvl) != 0) {
5327                 (void) fprintf(stderr, "Failed to parse fsacl permissions\n");
5328                 goto cleanup1;
5329         }

5331         if (opts.prt_perms)
5332                 print_fs_perms(&fs_perm_set);
5333         else {
5334                 (void) construct_fsacl_list(un, &opts, &update_perm_nvl);
5335                 if (zfs_set_fsacl(zhp, un, update_perm_nvl) != 0)
5336                         goto cleanup0;

5338                 if (un && opts.recursive) {
5339                         struct deleg_perms data = { un, update_perm_nvl };
5340                         if (zfs_iter_filesystems(zhp, set_deleg_perms,
5341                             &data) != 0)
5342                                 goto cleanup0;
5343                 }
5344         }

5346         error = 0;

5348 cleanup0:
5349         nvlist_free(perm_nvl);
5350         if (update_perm_nvl != NULL)
5350         nvlist_free(update_perm_nvl);
5351 cleanup1:
5352         fs_perm_set_fini(&fs_perm_set);
5353 cleanup2:
5354         zfs_close(zhp);

5356         return (error);
5357 }
______unchanged_portion_omitted_
```

```
**********************************************************
   44156 Mon Feb 15 12:56:04 2016
new/usr/src/cmd/zoneadm/zfs.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 357 /*
 358  * Make a ZFS clone on zonepath from snapshot_name.
 359  */
 360 static int
 361 clone_snap(char *snapshot_name, char *zonepath)
 362 {
 363         int             res = Z_OK;
 364         int             err;
 365         zfs_handle_t    *zhp;
 366         zfs_handle_t    *clone;
 367         nvlist_t        *props = NULL;

 369         if ((zhp = zfs_open(g_zfs, snapshot_name, ZFS_TYPE_SNAPSHOT)) == NULL)
 370                 return (Z_NO_ENTRY);

 372         (void) printf(gettext("Cloning snapshot %s\n"), snapshot_name);

 374         /*
 375          * We turn off zfs SHARENFS and SHARESMB properties on the
 376          * zoneroot dataset in order to prevent the GZ from sharing
 377          * NGZ data by accident.
 378          */
 379         if ((nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0) ||
 380             (nvlist_add_string(props, zfs_prop_to_name(ZFS_PROP_SHARENFS),
 381             "off") != 0) ||
 382             (nvlist_add_string(props, zfs_prop_to_name(ZFS_PROP_SHARESMB),
 383             "off") != 0)) {
 384                 if (props != NULL)
 384                 nvlist_free(props);
 385                 (void) fprintf(stderr, gettext("could not create ZFS clone "
 386                     "%s: out of memory\n"), zonepath);
 387                 return (Z_ERR);
 388         }

 390         err = zfs_clone(zhp, zonepath, props);
 391         zfs_close(zhp);

 393         nvlist_free(props);

 395         if (err != 0)
 396                 return (Z_ERR);

 398         /* create the mountpoint if necessary */
 399         if ((clone = zfs_open(g_zfs, zonepath, ZFS_TYPE_DATASET)) == NULL)
 400                 return (Z_ERR);

 402         /*
 403          * The clone has been created so we need to print a diagnostic
 404          * message if one of the following steps fails for some reason.
 405          */
 406         if (zfs_mount(clone, NULL, 0) != 0) {
 407                 (void) fprintf(stderr, gettext("could not mount ZFS clone "
 408                     "%s\n"), zfs_get_name(clone));
 409                 res = Z_ERR;

 411         } else if (clean_out_clone() != Z_OK) {
 412                 (void) fprintf(stderr, gettext("could not remove the "
 413                     "software inventory from ZFS clone %s\n"),
 414                     zfs_get_name(clone));
```



```
 415                 res = Z_ERR;
 416         }

 418         zfs_close(clone);
 419         return (res);
 420 }
______unchanged_portion_omitted_

 958 /*
 959  * Attempt to create a ZFS file system for the specified zonepath.
 960  * We either will successfully create a ZFS file system and get it mounted
 961  * on the zonepath or we don't.  The caller doesn't care since a regular
 962  * directory is used for the zonepath if no ZFS file system is mounted there.
 963  */
 964 void
 965 create_zfs_zonepath(char *zonepath)
 966 {
 967         zfs_handle_t    *zhp;
 968         char            zfs_name[MAXPATHLEN];
 969         nvlist_t        *props = NULL;

 971         if (path2name(zonepath, zfs_name, sizeof (zfs_name)) != Z_OK)
 972                 return;

 974         /* Check if the dataset already exists. */
 975         if ((zhp = zfs_open(g_zfs, zfs_name, ZFS_TYPE_DATASET)) != NULL) {
 976                 zfs_close(zhp);
 977                 return;
 978         }

 980         /*
 981          * We turn off zfs SHARENFS and SHARESMB properties on the
 982          * zoneroot dataset in order to prevent the GZ from sharing
 983          * NGZ data by accident.
 984          */
 985         if ((nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0) ||
 986             (nvlist_add_string(props, zfs_prop_to_name(ZFS_PROP_SHARENFS),
 987             "off") != 0) ||
 988             (nvlist_add_string(props, zfs_prop_to_name(ZFS_PROP_SHARESMB),
 989             "off") != 0)) {
 991                 if (props != NULL)
 990                 nvlist_free(props);
 991                 (void) fprintf(stderr, gettext("cannot create ZFS dataset %s: "
 992                     "out of memory\n"), zfs_name);
 993         }

 995         if (zfs_create(g_zfs, zfs_name, ZFS_TYPE_FILESYSTEM, props) != 0 ||
 996             (zhp = zfs_open(g_zfs, zfs_name, ZFS_TYPE_DATASET)) == NULL) {
 997                 (void) fprintf(stderr, gettext("cannot create ZFS dataset %s: "
 998                     "%s\n"), zfs_name, libzfs_error_description(g_zfs));
 999                 nvlist_free(props);
1000                 return;
1001         }

1003         nvlist_free(props);

1005         if (zfs_mount(zhp, NULL, 0) != 0) {
1006                 (void) fprintf(stderr, gettext("cannot mount ZFS dataset %s: "
1007                     "%s\n"), zfs_name, libzfs_error_description(g_zfs));
1008                 (void) zfs_destroy(zhp, B_FALSE);
1009         } else {
1010                 if (chmod(zonepath, S_IRWXU) != 0) {
1011                         (void) fprintf(stderr, gettext("file system %s "
1012                             "successfully created, but chmod %o failed: %s\n"),
1013                             zfs_name, S_IRWXU, strerror(errno));
1014                         (void) destroy_zfs(zonepath);
```

```
1015                 } else {
1016                         (void) printf(gettext("A ZFS file system has been "
1017                             "created for this zone.\n"));
1018                 }
1019         }

1021         zfs_close(zhp);
1022 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  144485 Mon Feb 15 12:56:04 2016
new/usr/src/cmd/zoneadmd/vplat.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

3320 static int
3321 get_rctls(zlog_t *zlogp, char **bufp, size_t *bufsizep)
3322 {
3323         nvlist_t *nvl = NULL;
3324         char *nvl_packed = NULL;
3325         size_t nvl_size = 0;
3326         nvlist_t **nvlv = NULL;
3327         int rctlcount = 0;
3328         int error = -1;
3329         zone_dochandle_t handle;
3330         struct zone_rctltab rctltab;
3331         rctlblk_t *rctlblk = NULL;
3332         uint64_t maxlwps;
3333         uint64_t maxprocs;

3335         *bufp = NULL;
3336         *bufsizep = 0;

3338         if ((handle = zonecfg_init_handle()) == NULL) {
3339                 zerror(zlogp, B_TRUE, "getting zone configuration handle");
3340                 return (-1);
3341         }
3342         if (zonecfg_get_snapshot_handle(zone_name, handle) != Z_OK) {
3343                 zerror(zlogp, B_FALSE, "invalid configuration");
3344                 zonecfg_fini_handle(handle);
3345                 return (-1);
3346         }

3348         rctltab.zone_rctl_valptr = NULL;
3349         if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
3350                 zerror(zlogp, B_TRUE, "%s failed", "nvlist_alloc");
3351                 goto out;
3352         }

3354         /*
3355          * Allow the administrator to control both the maximum number of
3356          * process table slots and the maximum number of lwps with just the
3357          * max-processes property.  If only the max-processes property is set,
3358          * we add a max-lwps property with a limit derived from max-processes.
3359          */
3360         if (zonecfg_get_aliased_rctl(handle, ALIAS_MAXPROCS, &maxprocs)
3361             == Z_OK &&
3362             zonecfg_get_aliased_rctl(handle, ALIAS_MAXLWPS, &maxlwps)
3363             == Z_NO_ENTRY) {
3364                 if (zonecfg_set_aliased_rctl(handle, ALIAS_MAXLWPS,
3365                     maxprocs * LWPS_PER_PROCESS) != Z_OK) {
3366                         zerror(zlogp, B_FALSE, "unable to set max-lwps alias");
3367                         goto out;
3368                 }
3369         }

3371         if (zonecfg_setrctlent(handle) != Z_OK) {
3372                 zerror(zlogp, B_FALSE, "%s failed", "zonecfg_setrctlent");
3373                 goto out;
3374         }

3376         if ((rctlblk = malloc(rctlblk_size())) == NULL) {
3377                 zerror(zlogp, B_TRUE, "memory allocation failed");
3378                 goto out;
```



```
3379         }
3380         while (zonecfg_getrctlent(handle, &rctltab) == Z_OK) {
3381                 struct zone_rctlvaltab *rctlval;
3382                 uint_t i, count;
3383                 const char *name = rctltab.zone_rctl_name;

3385                 /* zoneadm should have already warned about unknown rctls. */
3386                 if (!zonecfg_is_rctl(name)) {
3387                         zonecfg_free_rctl_value_list(rctltab.zone_rctl_valptr);
3388                         rctltab.zone_rctl_valptr = NULL;
3389                         continue;
3390                 }
3391                 count = 0;
3392                 for (rctlval = rctltab.zone_rctl_valptr; rctlval != NULL;
3393                     rctlval = rctlval->zone_rctlval_next) {
3394                         count++;
3395                 }
3396                 if (count == 0) {       /* ignore */
3397                         continue;       /* Nothing to free */
3398                 }
3399                 if ((nvlv = malloc(sizeof (*nvlv) * count)) == NULL)
3400                         goto out;
3401                 i = 0;
3402                 for (rctlval = rctltab.zone_rctl_valptr; rctlval != NULL;
3403                     rctlval = rctlval->zone_rctlval_next, i++) {
3404                         if (nvlist_alloc(&nvlv[i], NV_UNIQUE_NAME, 0) != 0) {
3405                                 zerror(zlogp, B_TRUE, "%s failed",
3406                                     "nvlist_alloc");
3407                                 goto out;
3408                         }
3409                         if (zonecfg_construct_rctlblk(rctlval, rctlblk)
3410                             != Z_OK) {
3411                                 zerror(zlogp, B_FALSE, "invalid rctl value: "
3412                                     "(priv=%s,limit=%s,action=%s)",
3413                                     rctlval->zone_rctlval_priv,
3414                                     rctlval->zone_rctlval_limit,
3415                                     rctlval->zone_rctlval_action);
3416                                 goto out;
3417                         }
3418                         if (!zonecfg_valid_rctl(name, rctlblk)) {
3419                                 zerror(zlogp, B_FALSE,
3420                                     "(priv=%s,limit=%s,action=%s) is not a "
3421                                     "valid value for rctl '%s'",
3422                                     rctlval->zone_rctlval_priv,
3423                                     rctlval->zone_rctlval_limit,
3424                                     rctlval->zone_rctlval_action,
3425                                     name);
3426                                 goto out;
3427                         }
3428                         if (nvlist_add_uint64(nvlv[i], "privilege",
3429                             rctlblk_get_privilege(rctlblk)) != 0) {
3430                                 zerror(zlogp, B_FALSE, "%s failed",
3431                                     "nvlist_add_uint64");
3432                                 goto out;
3433                         }
3434                         if (nvlist_add_uint64(nvlv[i], "limit",
3435                             rctlblk_get_value(rctlblk)) != 0) {
3436                                 zerror(zlogp, B_FALSE, "%s failed",
3437                                     "nvlist_add_uint64");
3438                                 goto out;
3439                         }
3440                         if (nvlist_add_uint64(nvlv[i], "action",
3441                             (uint_t)rctlblk_get_local_action(rctlblk, NULL))
3442                             != 0) {
3443                                 zerror(zlogp, B_FALSE, "%s failed",
3444                                     "nvlist_add_uint64");
```

```
3445                                    goto out;
3446                            }
3447                    }
3448                    zonecfg_free_rctl_value_list(rctltab.zone_rctl_valptr);
3449                    rctltab.zone_rctl_valptr = NULL;
3450                    if (nvlist_add_nvlist_array(nvl, (char *)name, nvlv, count)
3451                        != 0) {
3452                            zerror(zlogp, B_FALSE, "%s failed",
3453                                "nvlist_add_nvlist_array");
3454                            goto out;
3455                    }
3456                    for (i = 0; i < count; i++)
3457                            nvlist_free(nvlv[i]);
3458                    free(nvlv);
3459                    nvlv = NULL;
3460                    rctlcount++;
3461            }
3462            (void) zonecfg_endrctlent(handle);

3464            if (rctlcount == 0) {
3465                    error = 0;
3466                    goto out;
3467            }
3468            if (nvlist_pack(nvl, &nvl_packed, &nvl_size, NV_ENCODE_NATIVE, 0)
3469                != 0) {
3470                    zerror(zlogp, B_FALSE, "%s failed", "nvlist_pack");
3471                    goto out;
3472            }

3474            error = 0;
3475            *bufp = nvl_packed;
3476            *bufsizep = nvl_size;

3478 out:
3479            free(rctlblk);
3480            zonecfg_free_rctl_value_list(rctltab.zone_rctl_valptr);
3481            if (error && nvl_packed != NULL)
3482                    free(nvl_packed);
3483            if (nvl != NULL)
3483            nvlist_free(nvl);
3484            if (nvlv != NULL)
3485                    free(nvlv);
3486            if (handle != NULL)
3487                    zonecfg_fini_handle(handle);
3488            return (error);
3489 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  133502 Mon Feb 15 12:56:05 2016
new/usr/src/cmd/zpool/zpool_main.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

3204 /*
3205  * zpool split [-n] [-o prop=val] ...
3206  *              [-o mntopt] ...
3207  *              [-R altroot] <pool> <newpool> [<device> ...]
3208  *
3209  *      -n      Do not split the pool, but display the resulting layout if
3210  *              it were to be split.
3211  *      -o      Set property=value, or set mount options.
3212  *      -R      Mount the split-off pool under an alternate root.
3213  *
3214  * Splits the named pool and gives it the new pool name.  Devices to be split
3215  * off may be listed, provided that no more than one device is specified
3216  * per top-level vdev mirror.  The newly split pool is left in an exported
3217  * state unless -R is specified.
3218  *
3219  * Restrictions: the top-level of the pool pool must only be made up of
3220  * mirrors; all devices in the pool must be healthy; no device may be
3221  * undergoing a resilvering operation.
3222  */
3223 int
3224 zpool_do_split(int argc, char **argv)
3225 {
3226         char *srcpool, *newpool, *propval;
3227         char *mntopts = NULL;
3228         splitflags_t flags;
3229         int c, ret = 0;
3230         zpool_handle_t *zhp;
3231         nvlist_t *config, *props = NULL;

3233         flags.dryrun = B_FALSE;
3234         flags.import = B_FALSE;

3236         /* check options */
3237         while ((c = getopt(argc, argv, ":R:no:")) != -1) {
3238                 switch (c) {
3239                 case 'R':
3240                         flags.import = B_TRUE;
3241                         if (add_prop_list(
3242                             zpool_prop_to_name(ZPOOL_PROP_ALTROOT), optarg,
3243                             &props, B_TRUE) != 0) {
3244                                 if (props)
3244                                 nvlist_free(props);
3245                                 usage(B_FALSE);
3246                         }
3247                         break;
3248                 case 'n':
3249                         flags.dryrun = B_TRUE;
3250                         break;
3251                 case 'o':
3252                         if ((propval = strchr(optarg, '=')) != NULL) {
3253                                 *propval = '\0';
3254                                 propval++;
3255                                 if (add_prop_list(optarg, propval,
3256                                     &props, B_TRUE) != 0) {
3258                                         if (props)
3257                                         nvlist_free(props);
3258                                         usage(B_FALSE);
3259                                 }
3260                         } else {
```



```
3261                                 mntopts = optarg;
3262                         }
3263                         break;
3264                 case ':':
3265                         (void) fprintf(stderr, gettext("missing argument for "
3266                             "'%c' option\n"), optopt);
3267                         usage(B_FALSE);
3268                         break;
3269                 case '?':
3270                         (void) fprintf(stderr, gettext("invalid option '%c'\n"),
3271                             optopt);
3272                         usage(B_FALSE);
3273                         break;
3274                 }
3275         }

3277         if (!flags.import && mntopts != NULL) {
3278                 (void) fprintf(stderr, gettext("setting mntopts is only "
3279                     "valid when importing the pool\n"));
3280                 usage(B_FALSE);
3281         }

3283         argc -= optind;
3284         argv += optind;

3286         if (argc < 1) {
3287                 (void) fprintf(stderr, gettext("Missing pool name\n"));
3288                 usage(B_FALSE);
3289         }
3290         if (argc < 2) {
3291                 (void) fprintf(stderr, gettext("Missing new pool name\n"));
3292                 usage(B_FALSE);
3293         }

3295         srcpool = argv[0];
3296         newpool = argv[1];

3298         argc -= 2;
3299         argv += 2;

3301         if ((zhp = zpool_open(g_zfs, srcpool)) == NULL)
3302                 return (1);

3304         config = split_mirror_vdev(zhp, newpool, props, flags, argc, argv);
3305         if (config == NULL) {
3306                 ret = 1;
3307         } else {
3308                 if (flags.dryrun) {
3309                         (void) printf(gettext("would create '%s' with the "
3310                             "following layout:\n\n"), newpool);
3311                         print_vdev_tree(NULL, newpool, config, 0, B_FALSE);
3312                 }
3313                 nvlist_free(config);
3314         }

3316         zpool_close(zhp);

3318         if (ret != 0 || flags.dryrun || !flags.import)
3319                 return (ret);

3321         /*
3322          * The split was successful. Now we need to open the new
3323          * pool and import it.
3324          */
3325         if ((zhp = zpool_open_canfail(g_zfs, newpool)) == NULL)
3326                 return (1);
```

```
3327         if (zpool_get_state(zhp) != POOL_STATE_UNAVAIL &&
3328             zpool_enable_datasets(zhp, mntopts, 0) != 0) {
3329                 ret = 1;
3330                 (void) fprintf(stderr, gettext("Split was successful, but "
3331                     "the datasets could not all be mounted\n"));
3332                 (void) fprintf(stderr, gettext("Try doing '%s' with a "
3333                     "different altroot\n"), "zpool import");
3334         }
3335         zpool_close(zhp);

3337         return (ret);
3338 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   37828 Mon Feb 15 12:56:05 2016
new/usr/src/cmd/zpool/zpool_vdev.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

1367 nvlist_t *
1368 split_mirror_vdev(zpool_handle_t *zhp, char *newname, nvlist_t *props,
1369     splitflags_t flags, int argc, char **argv)
1370 {
1371         nvlist_t *newroot = NULL, **child;
1372         uint_t c, children;

1374         if (argc > 0) {
1375                 if ((newroot = construct_spec(argc, argv)) == NULL) {
1376                         (void) fprintf(stderr, gettext("Unable to build a "
1377                             "pool from the specified devices\n"));
1378                         return (NULL);
1379                 }

1381                 if (!flags.dryrun && make_disks(zhp, newroot) != 0) {
1382                         nvlist_free(newroot);
1383                         return (NULL);
1384                 }

1386                 /* avoid any tricks in the spec */
1387                 verify(nvlist_lookup_nvlist_array(newroot,
1388                     ZPOOL_CONFIG_CHILDREN, &child, &children) == 0);
1389                 for (c = 0; c < children; c++) {
1390                         char *path;
1391                         const char *type;
1392                         int min, max;

1394                         verify(nvlist_lookup_string(child[c],
1395                             ZPOOL_CONFIG_PATH, &path) == 0);
1396                         if ((type = is_grouping(path, &min, &max)) != NULL) {
1397                                 (void) fprintf(stderr, gettext("Cannot use "
1398                                     "'%s' as a device for splitting\n"), type);
1399                                 nvlist_free(newroot);
1400                                 return (NULL);
1401                         }
1402                 }
1403         }

1405         if (zpool_vdev_split(zhp, newname, &newroot, props, flags) != 0) {
1406                 if (newroot != NULL)
1406                 nvlist_free(newroot);
1407                 return (NULL);
1408         }

1410         return (newroot);
1411 }
______unchanged_portion_omitted_
```

****
7159 Mon Feb 15 12:56:05 2016
new/usr/src/common/fsreparse/fs_reparse.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
 74 /*
 75  * reparse_free()
 76  *
 77  * Function to free memory of a nvlist allocated previously
 78  * by reparse_init().
 79  */
 80 void
 81 reparse_free(nvlist_t *nvl)
 82 {
 83         if (nvl)
 83         nvlist_free(nvl);
 84 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   75115 Mon Feb 15 12:56:05 2016
new/usr/src/common/nvpair/nvpair.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 527 /*
 528  * Frees all memory allocated for an nvpair (like embedded lists) with
 529  * the exception of the nvpair buffer itself.
 530  */
 531 static void
 532 nvpair_free(nvpair_t *nvp)
 533 {
 534         switch (NVP_TYPE(nvp)) {
 535         case DATA_TYPE_NVLIST:
 536                 nvlist_free(EMBEDDED_NVL(nvp));
 537                 break;
 538         case DATA_TYPE_NVLIST_ARRAY: {
 539                 nvlist_t **nvlp = EMBEDDED_NVL_ARRAY(nvp);
 540                 int i;

 542                 for (i = 0; i < NVP_NELEM(nvp); i++)
 543                         if (nvlp[i] != NULL)
 543                         nvlist_free(nvlp[i]);
 544                 break;
 545         }
 546         default:
 547                 break;
 548         }
 549 }
______unchanged_portion_omitted_
```

```
**********************************************************
   60793 Mon Feb 15 12:56:05 2016
new/usr/src/lib/cfgadm_plugins/ib/common/cfga_ib.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_


1619 /*
1620  * Function:
1621  *      cfga_list_ext
1622  * Input:
1623  *      ap_id           - The attachment point of an IB fabric
1624  *      ap_id_list      - The returned "list" information array
1625  *      nlistp          - Number of elements in the "list" information array
1626  *      options         - List command options passed by the cfgadm(1M)
1627  *      listopts        - "-s" specific options
1628  *      errstring       - This contains error msg if command fails
1629  *      flags           - Cfgadm(1m) flags
1630  * Output:
1631  *      NONE
1632  * Returns:
1633  *      If the command succeeded, cfgadm -l output otherwise an error
1634  * Description:
1635  *      Do cfgadm -l
1636  */
1637 /*ARGSUSED*/
1638 cfga_err_t
1639 cfga_list_ext(const char *ap_id, cfga_list_data_t **ap_id_list, int *nlistp,
1640     const char *options, const char *listopts, char **errstring,
1641     cfga_flags_t flags)
1642 {
1643         int                     expand = 0;
1644         int                     i, index, count;
1645         int                     show_dynamic = 0;
1646         size_t                  num_devices = 0;
1647         size_t                  num_hcas = 0;
1648         size_t                  snap_size = 0;
1649         uchar_t                 *snap_data = NULL;
1650         nvpair_t                *nvp = NULL;    /* for lint purposes */
1651         nvlist_t                *nvl = NULL;
1652         boolean_t               apid_matched = B_FALSE; /* for valid ap_id */
1653         cfga_ib_ret_t           rv = CFGA_IB_OK;
1654         cfga_list_data_t        *clp = NULL;

1656         if ((rv = ib_verify_params(ap_id, options, errstring)) != CFGA_IB_OK) {
1657                 (void) cfga_help(NULL, options, flags);
1658                 return (ib_err_msg(errstring, rv, ap_id, errno));
1659         }

1661         /* make sure we have a valid ap_id_list */
1662         if (ap_id_list == NULL || nlistp == NULL) {
1663                 DPRINTF("cfga_list_ext: list = NULL or nlistp = NULL\n");
1664                 (void) cfga_help(NULL, options, flags);
1665                 return (ib_err_msg(errstring, CFGA_IB_INVAL_ARG_ERR,
1666                     ap_id, errno));
1667         }

1669         DPRINTF("cfga_list_ext: ap_id = %s\n", ap_id);

1671         if ((flags & CFGA_FLAG_LIST_ALL) == CFGA_FLAG_LIST_ALL) {
1672                 expand = 1;             /* -a flag passed */
1673         }

1675         if (GET_DYN(ap_id) != NULL) {
1676                 show_dynamic = 1;
```



```
1677         }

1679         if ((expand == 1) &&    /* -a option passed */
1680             (strstr((char *)ap_id, IB_FABRIC_APID_STR) != NULL)) {
1681                 /*
1682                  * Figure out how many IOC/Port/Pseudo
1683                  * devices exist in the system?
1684                  */
1685                 if ((rv = ib_do_control_ioctl((char *)ap_id,
1686                     IBNEX_NUM_DEVICE_NODES, 0, 0, 0, &num_devices)) !=
1687                     CFGA_IB_OK) {
1688                         DPRINTF("cfga_list_ext: ib_do_control_ioctl "
1689                             "IBNEX_NUM_DEVICE_NODES failed :%d\n", rv);
1690                         if (errno == ENOENT)
1691                                 return (CFGA_APID_NOEXIST);
1692                         return (ib_err_msg(errstring, rv, ap_id, errno));
1693                 }

1695                 DPRINTF("cfga_list_ext: num_devices = %d\n", num_devices);
1696         }

1698         /* Figure out how many HCA nodes exist in the system. */
1699         if ((rv = ib_do_control_ioctl((char *)ap_id, IBNEX_NUM_HCA_NODES, 0, 0,
1700             0, &num_hcas)) != CFGA_IB_OK) {
1701                 DPRINTF("cfga_list_ext: ib_do_control_ioctl "
1702                     "IBNEX_NUM_HCA_NODES failed :%d\n", rv);
1703                 if (errno == ENOENT)
1704                         return (CFGA_APID_NOEXIST);
1705                 return (ib_err_msg(errstring, rv, ap_id, errno));
1706         }
1707         DPRINTF("cfga_list_ext: num_hcas = %d\n", num_hcas);

1709         /*
1710          * No HCAs or IOC/VPPA/Port/HCA_SVC/Pseudo devices seen (non-IB system)
1711          */
1712         if (!(num_hcas || num_devices)) {
1713                 DPRINTF("cfga_list_ext: no IB devices found\n");
1714                 return (CFGA_APID_NOEXIST);
1715         }

1717         /*
1718          * *nlistp contains to how many APIDs to show w/ cfgadm -l.
1719          * If ap_id is "fabric" then
1720          *      *nlistp is all Dynamic Apids + One more for "fabric"
1721          * If ap_id is "HCA" ap_id then
1722          *      *nlistp is 1
1723          * Note that each HCA is a static APID, so nlistp will be 1 always
1724          * and this function will be called N times for each of the N HCAs
1725          * in the host.
1726          */
1727         if (strstr((char *)ap_id, IB_FABRIC_APID_STR) != NULL) {
1728                 *nlistp = num_devices + 1;

1730         } else {
1731                 /* Assume it as a HCA ap_id */
1732                 *nlistp = 1;
1733         }

1735         /* Allocate storage for passing "list" info back */
1736         if ((*ap_id_list = (cfga_list_data_t *)calloc(*nlistp,
1737             sizeof (cfga_list_data_t))) == NULL) {
1738                 DPRINTF("cfga_list_ext: malloc for cfga_list_data_t failed. "
1739                     "errno: %d\n", errno);
1740                 return (ib_err_msg(errstring, CFGA_IB_ALLOC_FAIL,
1741                     ap_id, errno));
1742         }
```

```
1744         /*
1745          * Only static ap_id is ib_fabric:
1746          * If -a options isn't specified then only show the static ap_id.
1747          */
1748         if (!show_dynamic) {
1749                 clp = &(*ap_id_list[0]);

1751                 if ((rv = ib_fill_static_apids((char *)ap_id, clp)) !=
1752                     CFGA_IB_OK) {
1753                         S_FREE(*ap_id_list);
1754                         return (ib_err_msg(errstring, rv, ap_id, errno));
1755                 }
1756                 apid_matched = B_TRUE;
1757         }

1759         /*
1760          * No -a specified
1761          * No HCAs or IOC/VPPA/HCA_SVC/Port/Pseudo devices seen (non-IB system)
1762          */
1763         if (!expand || (!num_hcas && !num_devices)) {
1764                 if (!show_dynamic)
1765                         return (CFGA_OK);
1766         }

1768         if (strstr((char *)ap_id, IB_FABRIC_APID_STR) != NULL) {
1769                 rv = ib_do_control_ioctl((char *)ap_id, IBNEX_SNAPSHOT_SIZE,
1770                     IBNEX_GET_SNAPSHOT, IBNEX_DONOT_PROBE_FLAG,
1771                     (void **)&snap_data, &snap_size);
1772                 if (rv != 0) {
1773                         DPRINTF("cfga_list_ext: ib_do_control_ioctl "
1774                             "failed :%d\n", rv);
1775                         S_FREE(*ap_id_list);
1776                         S_FREE(snap_data);
1777                         return (ib_err_msg(errstring, rv, ap_id, errno));
1778                 }

1780                 if (nvlist_unpack((char *)snap_data, snap_size, &nvl, 0)) {
1781                         DPRINTF("cfga_list_ext: nvlist_unpack 1 failed %p\n",
1782                             snap_data);
1783                         S_FREE(*ap_id_list);
1784                         S_FREE(snap_data);
1785                         return (ib_err_msg(errstring, CFGA_IB_NVLIST_ERR,
1786                             ap_id, errno));
1787                 }

1789                 /*
1790                  * In kernel a nvlist is build per ap_id which contains
1791                  * information that is displayed using cfgadm -l.
1792                  * For IB devices only these 6 items are shown:
1793                  *      ap_id, type, occupant, receptacle, condition and info
1794                  *
1795                  * In addition, one could specify a dynamic ap_id from
1796                  * command-line. Then cfgadm -l should show only that
1797                  * ap_id and skip rest.
1798                  */
1799                 index = 1; count = 0;
1800                 while (nvp = nvlist_next_nvpair(nvl, nvp)) {
1801                         int32_t intval = 0;
1802                         int32_t node_type;
1803                         char    *info;
1804                         char    *nv_apid;
1805                         char    *name = nvpair_name(nvp);

1807                         /* start of with next device */
1808                         if (count == IB_NUM_NVPAIRS) {
```



```
1809                                 count = 0;
1810                                 ++index;
1811                         }

1813                         /*
1814                          * Check if the index doesn't go beyond the
1815                          * device number. If it goes, stop the loop
1816                          * here not to cause the heap corruption.
1817                          */
1818                         if (show_dynamic == 0 && index > num_devices)
1819                                 break;

1821                         /* fill up data into "clp" */
1822                         clp =  (show_dynamic != 0) ? &(*ap_id_list[0]) :
1823                             &(ap_id_list[0][index]);

1825                         /* First nvlist entry is "ap_id" always */
1826                         if (strcmp(name, IBNEX_NODE_APID_NVL) == 0) {
1827                                 (void) nvpair_value_string(nvp, &nv_apid);
1828                                 DPRINTF("cfga_list_ext: Name = %s, apid = %s\n",
1829                                     name, nv_apid);

1831                                 /*
1832                                  * If a dynamic ap_id is specified in the
1833                                  * command-line, skip all entries until
1834                                  * the one needed matches.
1835                                  */
1836                                 if (show_dynamic &&
1837                                     strstr(ap_id, nv_apid) == NULL) {
1838                                         DPRINTF("cfga_list_ext: NO MATCH\n");

1840                                         /*
1841                                          * skip rest of the entries of this
1842                                          * device.
1843                                          */
1844                                         for (i = 0; i < IB_NUM_NVPAIRS - 1; i++)
1845                                                 nvp = nvlist_next_nvpair(nvl,
1846                                                     nvp);
1847                                         count = 0;      /* reset it */
1848                                         continue;
1849                                 }

1851                                 apid_matched = B_TRUE;

1853                                 /* build the physical ap_id */
1854                                 if (strstr(ap_id, DYN_SEP) == NULL) {
1855                                         (void) snprintf(clp->ap_phys_id,
1856                                             sizeof (clp->ap_phys_id), "%s%s%s",
1857                                             ap_id, DYN_SEP, nv_apid);
1858                                 } else {
1859                                         (void) snprintf(clp->ap_phys_id,
1860                                             sizeof (clp->ap_phys_id), "%s",
1861                                             ap_id);
1862                                 }

1864                                 /* ensure that this is a valid apid */
1865                                 if (ib_verify_valid_apid(clp->ap_phys_id) !=
1866                                     0) {
1867                                         DPRINTF("cfga_list_ext: "
1868                                             "not a valid IB ap_id\n");
1869                                         S_FREE(*ap_id_list);
1870                                         S_FREE(snap_data);
1871                                         nvlist_free(nvl);
1872                                         return (ib_err_msg(errstring,
1873                                             CFGA_IB_AP_ERR, ap_id, errno));
1874                                 }
```

```
1876                                    /* build the logical ap_id */
1877                                    (void) snprintf(clp->ap_log_id,
1878                                        sizeof (clp->ap_log_id), "ib%s%s",
1879                                        DYN_SEP, nv_apid);
1880                                    DPRINTF("cfga_list_ext: ap_pi = %s, ap_li = %s,"
1881                                        "\nap_info = %s\n", clp->ap_phys_id,
1882                                        clp->ap_log_id, clp->ap_info);
1883                                    ++count;

1885                            } else if (strcmp(name, IBNEX_NODE_INFO_NVL) == 0) {
1886                                    (void) nvpair_value_string(nvp, &info);
1887                                    DPRINTF("cfga_list_ext: Name = %s, info = %s\n",
1888                                        name, info);
1889                                    (void) snprintf(clp->ap_info,
1890                                        sizeof (clp->ap_info), "%s", info);
1891                                    ++count;

1893                            } else if (strcmp(name, IBNEX_NODE_TYPE_NVL) == 0) {
1894                                    (void) nvpair_value_int32(nvp, &node_type);
1895                                    if (node_type == IBNEX_PORT_NODE_TYPE) {
1896                                            (void) snprintf(clp->ap_type,
1897                                                sizeof (clp->ap_type), "%s",
1898                                                IB_PORT_TYPE);
1899                                    } else if (node_type == IBNEX_VPPA_NODE_TYPE) {
1900                                            (void) snprintf(clp->ap_type,
1901                                                sizeof (clp->ap_type), "%s",
1902                                                IB_VPPA_TYPE);
1903                                    } else if (node_type ==
1904                                        IBNEX_HCASVC_NODE_TYPE) {
1905                                            (void) snprintf(clp->ap_type,
1906                                                sizeof (clp->ap_type), "%s",
1907                                                IB_HCASVC_TYPE);
1908                                    } else if (node_type == IBNEX_IOC_NODE_TYPE) {
1909                                            (void) snprintf(clp->ap_type,
1910                                                sizeof (clp->ap_type), "%s",
1911                                                IB_IOC_TYPE);
1912                                    } else if (node_type ==
1913                                        IBNEX_PSEUDO_NODE_TYPE) {
1914                                            (void) snprintf(clp->ap_type,
1915                                                sizeof (clp->ap_type), "%s",
1916                                                IB_PSEUDO_TYPE);
1917                                    }
1918                                    DPRINTF("cfga_list_ext: Name = %s, type = %x\n",
1919                                        name, intval);
1920                                    ++count;

1922                            } else if (strcmp(name, IBNEX_NODE_RSTATE_NVL) == 0) {
1923                                    (void) nvpair_value_int32(nvp, &intval);

1925                                    if (intval == AP_RSTATE_EMPTY)
1926                                            clp->ap_r_state = CFGA_STAT_EMPTY;
1927                                    else if (intval == AP_RSTATE_DISCONNECTED)
1928                                            clp->ap_r_state =
1929                                                CFGA_STAT_DISCONNECTED;
1930                                    else if (intval == AP_RSTATE_CONNECTED)
1931                                            clp->ap_r_state = CFGA_STAT_CONNECTED;
1932                                    DPRINTF("cfga_list_ext: Name = %s, "
1933                                        "rstate = %x\n", name, intval);
1934                                    ++count;

1936                            } else if (strcmp(name, IBNEX_NODE_OSTATE_NVL) == 0) {
1937                                    (void) nvpair_value_int32(nvp, &intval);

1939                                    if (intval == AP_OSTATE_CONFIGURED)
1940                                            clp->ap_o_state = CFGA_STAT_CONFIGURED;
```



```
1941                                    else if (intval == AP_OSTATE_UNCONFIGURED)
1942                                            clp->ap_o_state =
1943                                                CFGA_STAT_UNCONFIGURED;
1944                                    DPRINTF("cfga_list_ext: Name = %s, "
1945                                        "ostate = %x\n", name, intval);
1946                                    ++count;

1948                            } else if (strcmp(name, IBNEX_NODE_COND_NVL) == 0) {
1949                                    (void) nvpair_value_int32(nvp, &intval);

1951                                    if (intval == AP_COND_OK)
1952                                            clp->ap_cond = CFGA_COND_OK;
1953                                    else if (intval == AP_COND_FAILING)
1954                                            clp->ap_cond = CFGA_COND_FAILING;
1955                                    else if (intval == AP_COND_FAILED)
1956                                            clp->ap_cond = CFGA_COND_FAILED;
1957                                    else if (intval == AP_COND_UNUSABLE)
1958                                            clp->ap_cond = CFGA_COND_UNUSABLE;
1959                                    else if (intval == AP_COND_UNKNOWN)
1960                                            clp->ap_cond = CFGA_COND_UNKNOWN;
1961                                    DPRINTF("cfga_list_ext: Name = %s, "
1962                                        "condition = %x\n", name, intval);
1963                                    ++count;
1964                            }

1966                            clp->ap_class[0] = '\0'; /* Filled by libcfgadm */
1967                            clp->ap_busy = 0;
1968                            clp->ap_status_time = (time_t)-1;
1969                    } /* end of while */
1970            }

1972            S_FREE(snap_data);
1973            if (nvl)
1973            nvlist_free(nvl);

1975            /*
1976             * if a cmdline specified ap_id doesn't match the known list of ap_ids
1977             * then report an error right away
1978             */
1979            rv = (apid_matched ==  B_TRUE) ? CFGA_IB_OK : CFGA_IB_AP_ERR;
1980            return (ib_err_msg(errstring, rv, ap_id, errno));
1981 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   34016 Mon Feb 15 12:56:05 2016
new/usr/src/lib/cfgadm_plugins/sbd/common/ap_rcm.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

644 static int
645 ap_rcm_cap_cpu(apd_t *a, rcmd_t *rcm, rcm_handle_t *hd, uint_t flags,
646         rcm_info_t **rinfo, int cmd, int change)
647 {
648         int i;
649         int rv = RCM_FAILURE;
650         int ncpuids;
651         int oldncpuids;
652         int newncpuids;
653         char buf[32];
654         const char *fmt;
655         size_t size;
656         nvlist_t *nvl = NULL;
657         cpuid_t *cpuids = NULL;
658         cpuid_t *oldcpuids = NULL;
659         cpuid_t *newcpuids = NULL;

661         DBG("ap_rcm_cap_cpu(%p)\n", (void *)a);

663         /*
664          * Get the current number of configured cpus.
665          */
666         if (getsyscpuids(&ncpuids, &cpuids) == -1)
667                 return (rv);
668         else if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0) != 0) {
669                 free(cpuids);
670                 goto done;
671         }

673         if (change == 1)
674                 fmt = "(%d cpu)";
675         else
676                 fmt = "(%d cpus)";

678         size = sizeof (cpuid_t);

680         if (cmd == CMD_RCM_CAP_DEL) {
681                 /*
682                  * A delete request. rcm->cpuids represents the
683                  * cpus that will be unconfigured. The current
684                  * set of cpus, before the unconfigure operation,
685                  * are the old CPUs. The new CPUs are those
686                  * that would remain.
687                  */
688                 oldncpuids = ncpuids;
689                 oldcpuids = cpuids;

691                 /*
692                  * Fill newcpuids with the CPU IDs in the cpuids array,
693                  * but not in rcm->cpuids.
694                  */
695                 newcpuids = (cpuid_t *)calloc(ncpuids, size);
696                 if (newcpuids == NULL)
697                         goto done;

699                 newncpuids = 0;
700                 for (i = 0; i < ncpuids; i++) {
701                         if (!is_cpu_in_list(cpuids[i], rcm->cpuids, change))
702                                 newcpuids[newncpuids++] = cpuids[i];
```



```
703                 }
704         } else if (cmd == CMD_RCM_CAP_NOTIFY) {
705                 /*
706                  * An unconfigure capacity change notification. This
707                  * notification is sent after a DR unconfigure, whether
708                  * or not the DR was successful. rcm->cpuids represents
709                  * the CPUs that have been unconfigured.
710                  */

712                 /* New CPU IDs are the CPUs configured right now. */
713                 newncpuids = ncpuids;
714                 newcpuids = cpuids;

716                 /*
717                  * Old CPU IDs are the CPUs configured right now
718                  * in addition to those that have been unconfigured.
719                  * We build the old CPU ID list by concatenating
720                  * cpuids and rcm->cpuids.
721                  */
722                 oldcpuids = (cpuid_t *)calloc(ncpuids + change, size);
723                 if (oldcpuids == NULL)
724                         goto done;

726                 oldncpuids = 0;
727                 for (i = 0; i < ncpuids; i++) {
728                         if (!is_cpu_in_list(cpuids[i], rcm->cpuids, change))
729                                 oldcpuids[oldncpuids++] = cpuids[i];
730                 }
731                 for (i = 0; i < change; i++)
732                         oldcpuids[oldncpuids++] = rcm->cpuids[i];
733         } else {
734                 DBG("ap_rcm_cap_cpu: CPU capacity, old = %d, new = %d \n",
735                     rcm->capcpus, ncpuids);
736                 if (rcm->capcpus == ncpuids) {
737                         /* No real change in CPU capacity */
738                         rv = RCM_SUCCESS;
739                         goto done;
740                 }

742                 /*
743                  * An add notification.  rcm->cpuids represents the
744                  * cpus that have been configured.  The current
745                  * set of cpus, after the configure operation,
746                  * are the new CPU IDs.
747                  */
748                 newncpuids = ncpuids;
749                 newcpuids = cpuids;

751                 /*
752                  * Fill oldcpuids with the CPU IDs in the cpuids array,
753                  * but not in rcm->cpuids.
754                  */
755                 oldcpuids = (cpuid_t *)calloc(ncpuids, size);
756                 if (oldcpuids == NULL)
757                         goto done;

759                 oldncpuids = 0;
760                 for (i = 0; i < ncpuids; i++) {
761                         if (!is_cpu_in_list(cpuids[i], rcm->cpuids, change))
762                                 oldcpuids[oldncpuids++] = cpuids[i];
763                 }
764         }

766         DBG("oldcpuids: ");
767         for (i = 0; i < oldncpuids; i++)
768                 DBG("%d ", oldcpuids[i]);
```

```
769         DBG("\n");
770         DBG("change   : ");
771         for (i = 0; i < change; i++)
772                 DBG("%d ", rcm->cpuids[i]);
773         DBG("\n");
774         DBG("newcpuids: ");
775         for (i = 0; i < newncpuids; i++)
776                 DBG("%d ", newcpuids[i]);
777         DBG("\n");

779         if (nvlist_add_string(nvl, "state", "capacity") != 0 ||
780             nvlist_add_int32(nvl, "old_total", oldncpuids) != 0 ||
781             nvlist_add_int32(nvl, "new_total", newncpuids) != 0 ||
782             nvlist_add_int32_array(nvl, "old_cpu_list", oldcpuids,
783             oldncpuids) != 0 ||
784             nvlist_add_int32_array(nvl, "new_cpu_list", newcpuids,
785             newncpuids) != 0)
786                 goto done;

788         (void) snprintf(buf, sizeof (buf), fmt, change);
789         ap_msg(a, MSG_ISSUE, cmd, buf);

791         if (cmd == CMD_RCM_CAP_DEL) {
792                 rv = (*rcm->request_capacity_change)(hd, "SUNW_cpu",
793                     flags, nvl, rinfo);
794         } else {
795                 rv = (*rcm->notify_capacity_change)(hd, "SUNW_cpu",
796                     flags & ~RCM_FORCE, nvl, rinfo);
797         }

799 done:
800         if (nvl)
800         nvlist_free(nvl);
801         s_free(oldcpuids);
802         s_free(newcpuids);
803         return (rv);
804 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    7805 Mon Feb 15 12:56:06 2016
new/usr/src/lib/fm/libfmd_agent/common/fmd_agent.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

  73 static int
  74 cleanup_set_errno(fmd_agent_hdl_t *hdl, nvlist_t *innvl, nvlist_t *outnvl,
  75     int err)
  76 {
  77         if (innvl != NULL)
  77         nvlist_free(innvl);
  79         if (outnvl != NULL)
  78         nvlist_free(outnvl);
  79         return (fmd_agent_seterrno(hdl, err));
  80 }
______unchanged_portion_omitted_
```

****
4573 Mon Feb 15 12:56:06 2016
new/usr/src/lib/fm/libfmd_agent/i386/fmd_agent_i386.c
patch tsoome-feedback
****

```
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <libnvpair.h>
#include <sys/fcntl.h>
#include <sys/devfm.h>
#include <fmd_agent_impl.h>

static int
cleanup_set_errno(fmd_agent_hdl_t *hdl, nvlist_t *innvl, nvlist_t *outnvl,
    int err)
{
        if (innvl != NULL)
        nvlist_free(innvl);
        if (outnvl != NULL)
        nvlist_free(outnvl);
        return (fmd_agent_seterrno(hdl, err));
}
```

_____***unchanged_portion_omitted_***

****
13354 Mon Feb 15 12:56:06 2016
new/usr/src/lib/fm/libfmevent/common/fmev_publish.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
356 static fmev_err_t
357 do_publish(const char *file, const char *func, int64_t line,
358     const char *ruleset, const char *class, const char *subclass,
359     fmev_pri_t pri, nvlist_t *nvl, uint_t ntuples, va_list ap)
360 {
361         fmev_err_t rc = FMEVERR_INTERNAL;
362         boolean_t priv = B_TRUE;
363         nvlist_t *tmpnvl = NULL;
364         nvlist_t *pub;
365         evchan_t *evc;

367         if (nvl) {
368                 ASSERT(ntuples == 0);

370                 /*
371                  * Enforce NV_UNIQUE_NAME
372                  */
373                 if ((nvlist_nvflag(nvl) & NV_UNIQUE_NAME) != NV_UNIQUE_NAME)
374                         return (FMEVERR_NVLIST);

376                 pub = nvl;

378         } else if (ntuples != 0) {
379                 fmev_err_t err;

381                 err = va2nvl(&tmpnvl, ap, ntuples);
382                 if (err != FMEV_SUCCESS)
383                         return (err);

385                 pub = tmpnvl;
386         } else {
387                 /*
388                  * Even if the caller has no tuples to publish (just an event
389                  * class and subclass), we are going to add some detector
390                  * information so we need some nvlist.
391                  */
392                 if (nvlist_alloc(&tmpnvl, NV_UNIQUE_NAME, 0) != 0)
393                         return (FMEVERR_ALLOC);

395                 pub = tmpnvl;
396         }

398         evc = bind_channel(priv, pri);

400         if (evc == NULL) {
401                 rc = FMEVERR_INTERNAL;
402                 goto done;
403         }


406         /*
407          * Add detector information
408          */
409         if (file && nvlist_add_string(pub, "__fmev_file", file) != 0 ||
410             func && nvlist_add_string(pub, "__fmev_func", func) != 0 ||
411             line != -1 && nvlist_add_int64(pub, "__fmev_line", line) != 0 ||
412             nvlist_add_int32(pub, "__fmev_pid", getpid()) != 0 ||
413             nvlist_add_string(pub, "__fmev_execname", getexecname()) != 0) {
414                 rc = FMEVERR_ALLOC;
```



```
415                 goto done;
416         }

418         if (ruleset == NULL)
419                 ruleset = FMEV_RULESET_DEFAULT;

421         /*
422          * We abuse the GPEC publication arguments as follows:
423          *
424          * GPEC argument        Our usage
425          * -------------------- -----------------
426          * const char *class    Raw class
427          * const char *subclass Raw subclass
428          * const char *vendor   Ruleset name
429          * const char *pub_name Unused
430          * nvlist_t *attr_list  Event attributes
431          */
432         rc = (sysevent_evc_publish(evc, class, subclass, ruleset, "",
433             pub, EVCH_NOSLEEP) == 0) ? FMEV_SUCCESS : FMEVERR_TRANSPORT;

435 done:
436         /* Free a passed in nvlist iff success */
437         if (rc == FMEV_SUCCESS)
437         if (nvl && rc == FMEV_SUCCESS)
438                 nvlist_free(nvl);

440         if (tmpnvl)
440         nvlist_free(tmpnvl);

442         return (rc);
443 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   61134 Mon Feb 15 12:56:06 2016
new/usr/src/lib/fm/topo/libtopo/common/hc.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 947 /*ARGSUSED*/
 948 static int
 949 hc_fmri_str2nvl(topo_mod_t *mod, tnode_t *node, topo_version_t version,
 950     nvlist_t *in, nvlist_t **out)
 951 {
 952         nvlist_t **pa = NULL;
 953         nvlist_t *nf = NULL;
 954         nvlist_t *auth = NULL;
 955         nvlist_t *fac = NULL;
 956         char *str;
 957         char *serial = NULL, *part = NULL, *rev = NULL, *hcsn = NULL;
 958         int npairs, n;
 959         int i, e;

 961         if (version > TOPO_METH_STR2NVL_VERSION)
 962                 return (topo_mod_seterrno(mod, EMOD_VER_NEW));

 964         if (nvlist_lookup_string(in, "fmri-string", &str) != 0)
 965                 return (topo_mod_seterrno(mod, EMOD_METHOD_INVAL));

 967         /* We're expecting a string version of an hc scheme FMRI */
 968         if (strncmp(str, "hc://", 5) != 0)
 969                 return (topo_mod_seterrno(mod, EMOD_FMRI_MALFORM));

 971         if ((pa = make_hc_pairs(mod, str, &npairs)) == NULL)
 972                 return (topo_mod_seterrno(mod, EMOD_FMRI_MALFORM));

 974         if (make_hc_auth(mod, str, &serial, &part, &rev, &auth) < 0)
 975                 goto hcfmbail;

 977         if ((nf = hc_base_fmri_create(mod, auth, part, rev, serial)) == NULL)
 978                 goto hcfmbail;

 980         n = npairs;

 982         /*
 983          * If the last pair in hc-list is offset or physaddr, we move
 984          * it to hc-specific.
 985          */
 986         (void) nvlist_lookup_string(pa[npairs - 1], FM_FMRI_HC_NAME, &hcsn);
 987         if (strcmp(hcsn, FM_FMRI_HC_SPECIFIC_OFFSET) == 0 ||
 988             strcmp(hcsn, FM_FMRI_HC_SPECIFIC_PHYSADDR) == 0) {
 989                 char *hcid;
 990                 nvlist_t *hcsp;
 991                 uint64_t val;

 993                 (void) nvlist_lookup_string(pa[npairs - 1], FM_FMRI_HC_ID,
 994                     &hcid);
 995                 val = strtoull(hcid, NULL, 16);
 996                 if (topo_mod_nvalloc(mod, &hcsp, NV_UNIQUE_NAME) != 0)
 997                         goto hcfmbail;
 998                 if (nvlist_add_uint64(hcsp, hcsn, val) != 0 ||
 999                     nvlist_add_nvlist(nf, FM_FMRI_HC_SPECIFIC, hcsp) != 0) {
1000                         nvlist_free(hcsp);
1001                         goto hcfmbail;
1002                 }

1004                 nvlist_free(hcsp);
1005                 n--;
```



```
1006         }

1008         if ((e = nvlist_add_uint32(nf, FM_FMRI_HC_LIST_SZ, n)) == 0)
1009                 e = nvlist_add_nvlist_array(nf, FM_FMRI_HC_LIST, pa, n);
1010         if (e != 0) {
1011                 topo_mod_dprintf(mod, "construction of new hc nvl failed");
1012                 goto hcfmbail;
1013         }

1015         /*
1016          * Clean-up
1017          */
1018         for (i = 0; i < npairs; i++)
1019                 nvlist_free(pa[i]);
1020         topo_mod_free(mod, pa, npairs * sizeof (nvlist_t *));
1021         topo_mod_strfree(mod, serial);
1022         topo_mod_strfree(mod, part);
1023         topo_mod_strfree(mod, rev);
1024         nvlist_free(auth);

1026         if (make_facility(mod, str, &fac) == -1)
1027                 goto hcfmbail;

1029         if (fac != NULL) {
1030                 if (nvlist_add_nvlist(nf, FM_FMRI_FACILITY, fac) != 0)
1031                         goto hcfmbail;
1032         }

1034         *out = nf;

1036         return (0);

1038 hcfmbail:
1039         if (nf != NULL)
1039         nvlist_free(nf);
1040         for (i = 0; i < npairs; i++)
1041                 nvlist_free(pa[i]);
1042         topo_mod_free(mod, pa, npairs * sizeof (nvlist_t *));

1044         topo_mod_strfree(mod, serial);
1045         topo_mod_strfree(mod, part);
1046         topo_mod_strfree(mod, rev);
1047         nvlist_free(auth);
1048         nvlist_free(nf);
1049         return (topo_mod_seterrno(mod, EMOD_FMRI_MALFORM));
1050 }
______unchanged_portion_omitted_

1118 static nvlist_t *
1119 hc_fmri_create(topo_mod_t *mod, nvlist_t *pfmri, int version, const char *name,
1120     topo_instance_t inst, const nvlist_t *auth, const char *part,
1121     const char *rev, const char *serial)
1122 {
1123         int i;
1124         char str[21]; /* sizeof (UINT64_MAX) + '\0' */
1125         uint_t pelems = 0;
1126         nvlist_t **phcl = NULL;
1127         nvlist_t **hcl = NULL;
1128         nvlist_t *fmri = NULL;

1130         if (version > FM_HC_SCHEME_VERSION)
1131                 return (hc_create_seterror(mod,
1132                     hcl, pelems, fmri, EMOD_VER_OLD));
1133         else if (version < FM_HC_SCHEME_VERSION)
1134                 return (hc_create_seterror(mod,
1135                     hcl, pelems, fmri, EMOD_VER_NEW));
```

```
1137         /*
1138          * Check that the requested name is in our canonical list
1139          */
1140         if (hc_name_canonical(mod, name) == 0)
1141                 return (hc_create_seterror(mod,
1142                     hcl, pelems, fmri, EMOD_NONCANON));
1143         /*
1144          * Copy the parent's HC_LIST
1145          */
1146         if (pfmri != NULL) {
1147                 if (nvlist_lookup_nvlist_array(pfmri, FM_FMRI_HC_LIST,
1148                     &phcl, &pelems) != 0)
1149                         return (hc_create_seterror(mod,
1150                             hcl, pelems, fmri, EMOD_FMRI_MALFORM));
1151         }

1153         hcl = topo_mod_zalloc(mod, sizeof (nvlist_t *) * (pelems + 1));
1154         if (hcl == NULL)
1155                 return (hc_create_seterror(mod,  hcl, pelems, fmri,
1156                     EMOD_NOMEM));

1158         for (i = 0; i < pelems; ++i)
1159                 if (topo_mod_nvdup(mod, phcl[i], &hcl[i]) != 0)
1160                         return (hc_create_seterror(mod,
1161                             hcl, pelems, fmri, EMOD_FMRI_NVL));

1163         (void) snprintf(str, sizeof (str), "%d", inst);
1164         if ((hcl[i] = hc_list_create(mod, name, str)) == NULL)
1165                 return (hc_create_seterror(mod,
1166                     hcl, pelems, fmri, EMOD_FMRI_NVL));

1168         if ((fmri = hc_base_fmri_create(mod, auth, part, rev, serial)) == NULL)
1169                 return (hc_create_seterror(mod,
1170                     hcl, pelems, fmri, EMOD_FMRI_NVL));

1172         if (nvlist_add_nvlist_array(fmri, FM_FMRI_HC_LIST, hcl, pelems + 1)
1173             != 0)
1174                 return (hc_create_seterror(mod,
1175                     hcl, pelems, fmri, EMOD_FMRI_NVL));

1177         if (hcl != NULL) {
1178                 for (i = 0; i < pelems + 1; ++i) {
1180                         if (hcl[i] != NULL)
1179                         nvlist_free(hcl[i]);
1180                 }
1181                 topo_mod_free(mod, hcl, sizeof (nvlist_t *) * (pelems + 1));
1182         }

1184         return (fmri);
1185 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   13375 Mon Feb 15 12:56:06 2016
new/usr/src/lib/fm/topo/libtopo/common/sw.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 126 /*ARGSUSED*/
 127 static int
 128 sw_fmri_create(topo_mod_t *mod, tnode_t *node, topo_version_t version,
 129     nvlist_t *in, nvlist_t **out)
 130 {
 131         nvlist_t *args, *fmri = NULL, *obj = NULL, *site = NULL, *ctxt = NULL;
 132         topo_mod_errno_t moderr;
 133         int err = 0;

 135         char *obj_path, *obj_root;
 136         nvlist_t *obj_pkg;

 138         char *site_token, *site_module, *site_file, *site_func;
 139         int64_t site_line;

 141         char *ctxt_origin, *ctxt_execname, *ctxt_zone;
 142         int64_t ctxt_pid, ctxt_ctid;
 143         char **ctxt_stack;
 144         uint_t ctxt_stackdepth;


 147         if (version > TOPO_METH_FMRI_VERSION)
 148                 return (topo_mod_seterrno(mod, EMOD_VER_NEW));

 150         if (nvlist_lookup_nvlist(in, TOPO_METH_FMRI_ARG_NVL, &args) != 0)
 151                 return (topo_mod_seterrno(mod, EMOD_METHOD_INVAL));

 153         if (nvlist_lookup_string(args, "obj_path", &obj_path) != 0)
 154                 return (topo_mod_seterrno(mod, EMOD_NVL_INVAL));
 155         err |= sw_get_optl_string(args, "obj_root", &obj_root);
 156         err |= sw_get_optl_nvlist(args, "obj-pkg", &obj_pkg);

 158         err |= sw_get_optl_string(args, "site_token", &site_token);
 159         err |= sw_get_optl_string(args, "site_module", &site_module);
 160         err |= sw_get_optl_string(args, "site_file", &site_file);
 161         err |= sw_get_optl_string(args, "site_func", &site_func);
 162         err |= sw_get_optl_int64(args, "site_line", &site_line);

 164         err |= sw_get_optl_string(args, "ctxt_origin", &ctxt_origin);
 165         err |= sw_get_optl_string(args, "ctxt_execname", &ctxt_execname);
 166         err |= sw_get_optl_string(args, "ctxt_zone", &ctxt_zone);
 167         err |= sw_get_optl_int64(args, "ctxt_pid", &ctxt_pid);
 168         err |= sw_get_optl_int64(args, "ctxt_ctid", &ctxt_ctid);

 170         if (nvlist_lookup_string_array(args, "stack", &ctxt_stack,
 171             &ctxt_stackdepth) != 0) {
 172                 if (errno == ENOENT)
 173                         ctxt_stack = NULL;
 174                 else
 175                         err++;
 176         }

 178         if (err)
 179                 (void) topo_mod_seterrno(mod, EMOD_FMRI_NVL);

 181         if (topo_mod_nvalloc(mod, &fmri, NV_UNIQUE_NAME) != 0 ||
 182             topo_mod_nvalloc(mod, &obj, NV_UNIQUE_NAME) != 0) {
 183                 moderr = EMOD_NOMEM;
 184                 goto out;
```



```
 185         }

 187         /*
 188          * Add standard FMRI members 'version' and 'scheme'.
 189          */
 190         err |= nvlist_add_uint8(fmri, FM_VERSION, FM_SW_SCHEME_VERSION);
 191         err |= nvlist_add_string(fmri, FM_FMRI_SCHEME, FM_FMRI_SCHEME_SW);

 193         /*
 194          * Build up the 'object' nvlist.
 195          */
 196         err |= nvlist_add_string(obj, FM_FMRI_SW_OBJ_PATH, obj_path);
 197         err |= sw_add_optl_string(obj, FM_FMRI_SW_OBJ_ROOT, obj_root);
 198         if (obj_pkg)
 199                 err |= nvlist_add_nvlist(obj, FM_FMRI_SW_OBJ_PKG, obj_pkg);

 201         /*
 202          * Add 'object' to the fmri.
 203          */
 204         if (err == 0)
 205                 err |= nvlist_add_nvlist(fmri, FM_FMRI_SW_OBJ, obj);

 207         if (err) {
 208                 moderr = EMOD_NOMEM;
 209                 goto out;
 210         }

 212         /*
 213          * Do we have anything for a 'site' nvlist?
 214          */
 215         if (site_token == NULL && site_module == NULL && site_file == NULL &&
 216             site_func == NULL && site_line == -1)
 217                 goto context;

 219         /*
 220          * Allocate and build 'site' nvlist.
 221          */
 222         if (topo_mod_nvalloc(mod, &site, NV_UNIQUE_NAME) != 0) {
 223                 moderr = EMOD_NOMEM;
 224                 goto out;
 225         }

 227         err |= sw_add_optl_string(site, FM_FMRI_SW_SITE_TOKEN, site_token);
 228         err |= sw_add_optl_string(site, FM_FMRI_SW_SITE_MODULE, site_module);
 229         err |= sw_add_optl_string(site, FM_FMRI_SW_SITE_FILE, site_file);
 230         err |= sw_add_optl_string(site, FM_FMRI_SW_SITE_FUNC, site_func);
 231         if ((site_token || site_module || site_file || site_func) &&
 232             site_line != -1)
 233                 err |= nvlist_add_int64(site, FM_FMRI_SW_SITE_LINE, site_line);

 235         /*
 236          * Add 'site' to the fmri.
 237          */
 238         if (err == 0)
 239                 err |= nvlist_add_nvlist(fmri, FM_FMRI_SW_SITE, site);

 241         if (err) {
 242                 moderr = EMOD_NOMEM;
 243                 goto out;
 244         }

 246 context:
 247         /*
 248          * Do we have anything for a 'context' nvlist?
 249          */
 250         if (ctxt_origin || ctxt_execname || ctxt_zone ||
```

```
251             ctxt_pid != -1 || ctxt_ctid != -1 || ctxt_stack != NULL)
252                 goto out;

254         /*
255          * Allocate and build 'context' nvlist.
256          */
257         if (topo_mod_nvalloc(mod, &ctxt, NV_UNIQUE_NAME) != 0) {
258                 moderr = EMOD_NOMEM;
259                 goto out;
260         }

262         err |= sw_add_optl_string(ctxt, FM_FMRI_SW_CTXT_ORIGIN, ctxt_origin);
263         err |= sw_add_optl_string(ctxt, FM_FMRI_SW_CTXT_EXECNAME,
264             ctxt_execname);
265         err |= sw_add_optl_string(ctxt, FM_FMRI_SW_CTXT_ZONE, ctxt_zone);
266         if (ctxt_pid != -1)
267                 err |= nvlist_add_int64(ctxt, FM_FMRI_SW_CTXT_PID, ctxt_pid);
268         if (ctxt_ctid != -1)
269                 err |= nvlist_add_int64(ctxt, FM_FMRI_SW_CTXT_CTID, ctxt_ctid);
270         if (ctxt_stack != NULL)
271                 err |= nvlist_add_string_array(ctxt, FM_FMRI_SW_CTXT_STACK,
272                     ctxt_stack, ctxt_stackdepth);

274         /*
275          * Add 'context' to the fmri.
276          */
277         if (err == 0)
278                 err |= nvlist_add_nvlist(fmri, FM_FMRI_SW_CTXT, ctxt);

280         moderr = err ? EMOD_NOMEM : 0;
281 out:
282         if (moderr == 0)
283                 *out = fmri;
284         else

285         if (moderr != 0 && fmri)
285                 nvlist_free(fmri);

288         if (obj)
287         nvlist_free(obj);

291         if (site)
288         nvlist_free(site);

294         if (ctxt)
289         nvlist_free(ctxt);

291         return (moderr == 0 ? 0 : topo_mod_seterrno(mod, moderr));
292 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   28270 Mon Feb 15 12:56:06 2016
new/usr/src/lib/fm/topo/libtopo/common/topo_fmri.c
patch tsoome-feedback
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
  24  * Use is subject to license terms.
  25  */

  27 #include <ctype.h>
  28 #include <string.h>
  29 #include <limits.h>
  30 #include <fm/topo_mod.h>
  31 #include <fm/fmd_fmri.h>
  32 #include <sys/fm/protocol.h>
  33 #include <topo_alloc.h>
  34 #include <topo_error.h>
  35 #include <topo_hc.h>
  36 #include <topo_method.h>
  37 #include <topo_subr.h>
  38 #include <topo_string.h>

  40 /*
  41  * Topology node properties and method operations may be accessed by FMRI.
  42  * The FMRI used to perform property look-ups and method operations is
  43  * the FMRI contained in the matching topology node's protocol property
  44  * grouping for the resource property. The full range of fmd(1M)
  45  * scheme plugin operations are supported as long as a backend method is
  46  * supplied by a scheme-specific enumerator or the enumerator module that
  47  * created the matching topology node.  Support for fmd scheme operations
  48  * include:
  49  *
  50  *      - expand
  51  *      - present
  52  *      - replaced
  53  *      - contains
  54  *      - unusable
  55  *      - service_state
  56  *      - nvl2str
  57  *      - retire
  58  *      - unretire
  59  *
  60  * In addition, the following operations are supported per-FMRI:
  61  *
```



```
  62  *      - str2nvl: convert string-based FMRI to nvlist
  63  *      - compare: compare two FMRIs
  64  *      - asru: lookup associated ASRU property by FMRI
  65  *      - fru: lookup associated FRU by FMRI
  66  *      - create: an FMRI nvlist by scheme type
  67  *      - propery lookup
  68  *
  69  * These routines may only be called by consumers of a topology snapshot.
  70  * They may not be called by libtopo enumerator or method modules.
  71  */

  73 /*ARGSUSED*/
  74 static int
  75 set_error(topo_hdl_t *thp, int err, int *errp, char *method, nvlist_t *nvlp)
  76 {
  77         if (nvlp != NULL)
  77         nvlist_free(nvlp);

  79         topo_dprintf(thp, TOPO_DBG_ERR, "%s failed: %s\n", method,
  80             topo_strerror(err));

  82         *errp = err;
  83         return (-1);
  84 }

  86 /*ARGSUSED*/
  87 static nvlist_t *
  88 set_nverror(topo_hdl_t *thp, int err, int *errp, char *method, nvlist_t *nvlp)
  89 {
  91         if (nvlp != NULL)
  90         nvlist_free(nvlp);

  92         topo_dprintf(thp, TOPO_DBG_ERR, "%s failed: %s\n", method,
  93             topo_strerror(err));

  95         *errp = err;
  96         return (NULL);
  97 }
______unchanged_portion_omitted_

 558 int topo_fmri_setprop(topo_hdl_t *thp, nvlist_t *nvl, const char *pg,
 559     nvlist_t *prop, int flag, nvlist_t *args, int *err)
 560 {
 561         int rv;
 562         nvlist_t *in = NULL, *out = NULL;
 563         tnode_t *rnode;
 564         char *scheme;

 566         if (nvlist_lookup_string(nvl, FM_FMRI_SCHEME, &scheme) != 0)
 567                 return (set_error(thp, ETOPO_FMRI_MALFORM, err,
 568                     TOPO_METH_PROP_SET, in));

 570         if ((rnode = topo_hdl_root(thp, scheme)) == NULL)
 571                 return (set_error(thp, ETOPO_METHOD_NOTSUP, err,
 572                     TOPO_METH_PROP_SET, in));

 574         if (topo_hdl_nvalloc(thp, &in, NV_UNIQUE_NAME) != 0)
 575                 return (set_error(thp, ETOPO_FMRI_NVL, err,
 576                     TOPO_METH_PROP_SET, in));

 578         rv = nvlist_add_nvlist(in, TOPO_PROP_RESOURCE, nvl);
 579         rv |= nvlist_add_string(in, TOPO_PROP_GROUP, pg);
 580         rv |= nvlist_add_nvlist(in, TOPO_PROP_VAL, prop);
 581         rv |= nvlist_add_int32(in, TOPO_PROP_FLAG, (int32_t)flag);
 582         if (args != NULL)
 583                 rv |= nvlist_add_nvlist(in, TOPO_PROP_PARGS, args);
```

```
584         if (rv != 0)
585                 return (set_error(thp, ETOPO_FMRI_NVL, err,
586                     TOPO_METH_PROP_SET, in));

588         rv = topo_method_invoke(rnode, TOPO_METH_PROP_SET,
589             TOPO_METH_PROP_SET_VERSION, in, &out, err);

591         nvlist_free(in);

593         /* no return values */
596         if (out != NULL)
594         nvlist_free(out);

596         if (rv)
597                 return (-1);

599         return (0);

601 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    4872 Mon Feb 15 12:56:06 2016
new/usr/src/lib/fm/topo/libtopo/common/topo_parse.c
patch tsoome-feedback
**********************************************************
```

```
/*
 * 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 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <libxml/parser.h>
#include <fm/libtopo.h>
#include <topo_alloc.h>
#include <topo_error.h>
#include <topo_parse.h>
#include <topo_subr.h>

tf_info_t *
tf_info_new(topo_mod_t *mp, xmlDocPtr doc, xmlChar *scheme)
{
        tf_info_t *r;

        if ((r = topo_mod_zalloc(mp, sizeof (tf_info_t))) == NULL)
                return (NULL);
        r->tf_flags = TF_LIVE;
        if ((r->tf_scheme = topo_mod_strdup(mp, (char *)scheme)) == NULL) {
                tf_info_free(mp, r);
                return (NULL);
        }
        r->tf_xdoc = doc;
        return (r);
}
```

_____***unchanged_portion_omitted_***

```
void
tf_pad_free(topo_mod_t *mp, tf_pad_t *p)
{
        int n;
        if (p == NULL)
                return;
        if (p->tpad_pgs != NULL) {
                for (n = 0; n < p->tpad_pgcnt; n++)
                        if (p->tpad_pgs[n] != NULL)
```



```
                                nvlist_free(p->tpad_pgs[n]);
                topo_mod_free(mp,
                    p->tpad_pgs, p->tpad_pgcnt * sizeof (nvlist_t *));
        }
        tf_rdata_free(mp, p->tpad_child);
        tf_rdata_free(mp, p->tpad_sibs);
        topo_mod_free(mp, p, sizeof (tf_pad_t));
}
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   39038 Mon Feb 15 12:56:06 2016
new/usr/src/lib/fm/topo/libtopo/common/topo_prop.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 109 static int
 110 method_geterror(nvlist_t *nvl, int err, int *errp)
 111 {
 112         if (nvl != NULL)
 112         nvlist_free(nvl);

 114         *errp = err;

 116         return (-1);
 117 }

 119 static int
 120 prop_method_get(tnode_t *node, topo_propval_t *pv, topo_propmethod_t *pm,
 121     nvlist_t *pargs, int *err)
 122 {
 123         int ret;
 124         nvlist_t *args, *nvl;
 125         char *name;
 126         topo_type_t type;

 128         if (topo_hdl_nvalloc(pv->tp_hdl, &args, NV_UNIQUE_NAME) < 0 ||
 129             nvlist_add_nvlist(args, TOPO_PROP_ARGS, pm->tpm_args) != 0)
 130                 return (method_geterror(NULL, ETOPO_PROP_NVL, err));

 132         if (pargs != NULL)
 133                 if (nvlist_add_nvlist(args, TOPO_PROP_PARGS, pargs) != 0)
 134                         return (method_geterror(args, ETOPO_PROP_NVL, err));

 136         /*
 137          * Now, get the latest value
 138          *
 139          * Grab a reference to the property and then unlock the node.  This will
 140          * allow property methods to safely re-enter the prop_get codepath,
 141          * making it possible for property methods to access other property
 142          * values on the same node w\o causing a deadlock.
 143          */
 144         topo_prop_hold(pv);
 145         topo_node_unlock(node);
 146         if (topo_method_call(node, pm->tpm_name, pm->tpm_version,
 147             args, &nvl, err) < 0) {
 148                 topo_node_lock(node);
 149                 topo_prop_rele(pv);
 150                 return (method_geterror(args, *err, err));
 151         }
 152         topo_node_lock(node);
 153         topo_prop_rele(pv);

 155         nvlist_free(args);

 157         /* Verify the property contents */
 158         ret = nvlist_lookup_string(nvl, TOPO_PROP_VAL_NAME, &name);
 159         if (ret != 0 || strcmp(name, pv->tp_name) != 0)
 160                 return (method_geterror(nvl, ETOPO_PROP_NAME, err));

 162         ret = nvlist_lookup_uint32(nvl, TOPO_PROP_VAL_TYPE, (uint32_t *)&type);
 163         if (ret != 0 || type != pv->tp_type)
 164                 return (method_geterror(nvl, ETOPO_PROP_TYPE, err));

 166         /* Release the last value and re-assign to the new value */
```



```
 168         if (pv->tp_val != NULL)
 167         nvlist_free(pv->tp_val);
 168         pv->tp_val = nvl;

 170         return (0);
 171 }
______unchanged_portion_omitted_

 847 static int
 848 register_methoderror(tnode_t *node, topo_propmethod_t *pm, int *errp, int l,
 849     int err)
 850 {
 851         topo_hdl_t *thp = node->tn_hdl;

 853         if (pm != NULL) {
 854                 if (pm->tpm_name != NULL)
 855                         topo_hdl_strfree(thp, pm->tpm_name);
 858                 if (pm->tpm_args != NULL)
 856                 nvlist_free(pm->tpm_args);
 857                 topo_hdl_free(thp, pm, sizeof (topo_propmethod_t));
 858         }

 860         *errp = err;

 862         if (l != 0)
 863                 topo_node_unlock(node);

 865         return (-1);
 866 }
______unchanged_portion_omitted_

1298 static void
1299 propmethod_destroy(topo_hdl_t *thp, topo_propval_t *pv)
1300 {
1301         topo_propmethod_t *pm;

1303         pm = pv->tp_method;
1304         if (pm != NULL) {
1305                 if (pm->tpm_name != NULL)
1306                         topo_hdl_strfree(thp, pm->tpm_name);
1310                 if (pm->tpm_args != NULL)
1307                 nvlist_free(pm->tpm_args);
1308                 topo_hdl_free(thp, pm, sizeof (topo_propmethod_t));
1309                 pv->tp_method = NULL;
1310         }
1311 }

1313 static void
1314 topo_propval_destroy(topo_propval_t *pv)
1315 {
1316         topo_hdl_t *thp;

1318         if (pv == NULL)
1319                 return;

1321         thp = pv->tp_hdl;

1323         if (pv->tp_name != NULL)
1324                 topo_hdl_strfree(thp, pv->tp_name);

1330         if (pv->tp_val != NULL)
1326         nvlist_free(pv->tp_val);

1328         propmethod_destroy(thp, pv);

1330         topo_hdl_free(thp, pv, sizeof (topo_propval_t));
```

```
1331 }
______unchanged_portion_omitted_

1396 static int
1397 get_pgrp_seterror(tnode_t *node, nvlist_t *nvl, int *errp, int err)
1398 {
1399         topo_node_unlock(node);

1406         if (nvl != NULL)
1401         nvlist_free(nvl);

1403         *errp = err;

1405         return (-1);
1406 }
______unchanged_portion_omitted_

1468 static nvlist_t *
1469 get_all_seterror(tnode_t *node, nvlist_t *nvl, int *errp, int err)
1470 {
1471         topo_node_unlock(node);

1479         if (nvl != NULL)
1473         nvlist_free(nvl);

1475         *errp = err;

1477         return (NULL);
1478 }
______unchanged_portion_omitted_
```

```
**********************************************************
   57844 Mon Feb 15 12:56:07 2016
new/usr/src/lib/fm/topo/libtopo/common/topo_xml.c
patch tsoome-feedback
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 844 static int
 845 pmeth_record(topo_mod_t *mp, const char *pg_name, xmlNodePtr xn, tnode_t *tn,
 846     const char *rname, const char *ppgrp_name)
 847 {
 848         nvlist_t *arg_nvl = NULL;
 849         xmlNodePtr cn;
 850         xmlChar *meth_name = NULL, *prop_name = NULL;
 851         xmlChar *arg_name = NULL;
 852         uint64_t meth_ver, is_mutable = 0, is_nonvolatile = 0;
 853         topo_type_t prop_type;
 854         struct propmeth_data meth;
 855         int ret = 0, err;
 856         topo_type_t ptype;
 857         tnode_t *tmp;

 859         topo_dprintf(mp->tm_hdl, TOPO_DBG_XML, "pmeth_record: %s=%d "
 860             "(pgrp=%s)\n", topo_node_name(tn), topo_node_instance(tn), pg_name);

 862         /*
 863          * Get propmethod attribute values
 864          */
 865         if ((meth_name = xmlGetProp(xn, (xmlChar *)Name)) == NULL) {
 866                 topo_dprintf(mp->tm_hdl, TOPO_DBG_ERR,
 867                     "propmethod element lacks a name attribute\n");
 868                 return (topo_mod_seterrno(mp, ETOPO_PRSR_NOATTR));
 869         }
 870         if (xmlattr_to_int(mp, xn, Version, &meth_ver) < 0) {
 871                 topo_dprintf(mp->tm_hdl, TOPO_DBG_ERR,
 872                     "propmethod element lacks version attribute\n");
 873                 ret = topo_mod_seterrno(mp, ETOPO_PRSR_NOATTR);
 874                 goto pmr_done;
 875         }
 876         /*
 877          * The "mutable" and "nonvoltile" attributes are optional.  If not
 878          * specified we default to false (0)
 879          */
 880         (void) xmlattr_to_int(mp, xn, Mutable, &is_mutable);
 881         (void) xmlattr_to_int(mp, xn, Nonvolatile, &is_nonvolatile);

 883         if ((prop_name = xmlGetProp(xn, (xmlChar *)Propname)) == NULL) {
 884                 topo_dprintf(mp->tm_hdl, TOPO_DBG_ERR,
 885                     "propmethod element lacks propname attribute\n");
 886                 ret = topo_mod_seterrno(mp, ETOPO_PRSR_NOATTR);
 887                 goto pmr_done;
 888         }
 889         if ((prop_type = xmlattr_to_type(mp, xn, (xmlChar *)Proptype))
 890             == TOPO_TYPE_INVALID) {
 891                 topo_dprintf(mp->tm_hdl, TOPO_DBG_ERR,
 892                     "error decoding proptype attribute\n");
 893                 ret = topo_mod_seterrno(mp, ETOPO_PRSR_NOATTR);
 894                 goto pmr_done;
 895         }

 897         /*
 898          * Allocate method argument nvlist
 899          */
 900         if (topo_mod_nvalloc(mp, &arg_nvl, NV_UNIQUE_NAME) < 0) {
 901                 ret = topo_mod_seterrno(mp, ETOPO_NOMEM);
```



```
 902                 goto pmr_done;
 903         }

 905         /*
 906          * Iterate through the argval nodes and build the argval nvlist
 907          */
 908         for (cn = xn->xmlChildrenNode; cn != NULL; cn = cn->next) {
 909                 if (xmlStrcmp(cn->name, (xmlChar *)Argval) == 0) {
 910                         topo_dprintf(mp->tm_hdl, TOPO_DBG_XML,
 911                             "found argval element\n");
 912                         if ((arg_name = xmlGetProp(cn, (xmlChar *)Name))
 913                             == NULL) {
 914                                 topo_dprintf(mp->tm_hdl, TOPO_DBG_XML,
 915                                     "argval element lacks a name attribute\n");
 916                                 ret = topo_mod_seterrno(mp, ETOPO_PRSR_NOATTR);
 917                                 goto pmr_done;
 918                         }
 919                         if ((ptype = xmlattr_to_type(mp, cn, (xmlChar *)Type))
 920                             == TOPO_TYPE_INVALID) {
 921                                 ret = topo_mod_seterrno(mp, ETOPO_PRSR_BADTYPE);
 922                                 xmlFree(arg_name);
 923                                 break;
 924                         }
 925                         if (xlate_common(mp, cn, ptype, arg_nvl,
 926                             (const char *)arg_name) != 0) {
 927                                 ret = topo_mod_seterrno(mp, ETOPO_PRSR_BADTYPE);
 928                                 xmlFree(arg_name);
 929                                 break;
 930                         }
 931                 }
 932                 if (arg_name) {
 933                         xmlFree(arg_name);
 934                         arg_name = NULL;
 935                 }
 936         }

 938         if (ret != 0)
 939                 goto pmr_done;

 941         /*
 942          * Register the prop method for all of the nodes in our range
 943          */
 944         meth.pg_name = (const char *)pg_name;
 945         meth.prop_name = (const char *)prop_name;
 946         meth.prop_type = prop_type;
 947         meth.meth_name = (const char *)meth_name;
 948         meth.meth_ver = meth_ver;
 949         meth.arg_nvl = arg_nvl;

 951         /*
 952          * If the propgroup element is under a range element, we'll apply
 953          * the method to all of the topo nodes at this level with the same
 954          * range name.
 955          *
 956          * Otherwise, if the propgroup element is under a node element
 957          * then we'll simply register the method for this node.
 958          */
 959         if (strcmp(ppgrp_name, Range) == 0) {
 960                 for (tmp = tn; tmp != NULL; tmp = topo_child_next(NULL, tmp)) {
 961                         if (strcmp(rname, topo_node_name(tmp)) == 0) {
 962                                 if (register_method(mp, tmp, &meth) != 0) {
 963                                         ret = topo_mod_seterrno(mp,
 964                                             ETOPO_PRSR_REGMETH);
 965                                         goto pmr_done;
 966                                 }
 967                                 if (is_mutable) {
```

```
968                                     if (topo_prop_setmutable(tmp,
969                                         meth.pg_name, meth.prop_name, &err)
970                                         != 0) {
971                                             ret = topo_mod_seterrno(mp,
972                                                 ETOPO_PRSR_REGMETH);
973                                             goto pmr_done;
974                                     }
975                                 }
976                                 if (is_nonvolatile) {
977                                     if (topo_prop_setnonvolatile(tmp,
978                                         meth.pg_name, meth.prop_name, &err)
979                                         != 0) {
980                                             ret = topo_mod_seterrno(mp,
981                                                 ETOPO_PRSR_REGMETH);
982                                             goto pmr_done;
983                                     }
984                                 }
985                         }
986                 }
987         } else {
988                 if (register_method(mp, tn, &meth) != 0) {
989                         ret = topo_mod_seterrno(mp, ETOPO_PRSR_REGMETH);
990                         goto pmr_done;
991                 }
992                 if (is_mutable) {
993                         if (topo_prop_setmutable(tn, meth.pg_name,
994                             meth.prop_name, &err) != 0) {
995                                 ret = topo_mod_seterrno(mp,
996                                     ETOPO_PRSR_REGMETH);
997                                 goto pmr_done;
998                         }
999                 }
1000                if (is_nonvolatile) {
1001                        if (topo_prop_setnonvolatile(tn, meth.pg_name,
1002                            meth.prop_name, &err) != 0) {
1003                                ret = topo_mod_seterrno(mp,
1004                                    ETOPO_PRSR_REGMETH);
1005                                goto pmr_done;
1006                        }
1007                }
1008        }

1010 pmr_done:
1011        if (meth_name)
1012                xmlFree(meth_name);
1013        if (prop_name)
1014                xmlFree(prop_name);
1015        if (arg_nvl)
1015        nvlist_free(arg_nvl);
1016        return (ret);
1017 }


1020 static int
1021 pgroup_record(topo_mod_t *mp, xmlNodePtr pxn, tnode_t *tn, const char *rname,
1022     tf_pad_t *rpad, int pi, const char *ppgrp_name)
1023 {
1024        topo_stability_t nmstab, dstab;
1025        uint64_t ver;
1026        xmlNodePtr cn;
1027        xmlChar *name;
1028        nvlist_t **apl = NULL;
1029        nvlist_t *pgnvl = NULL;
1030        int pcnt = 0;
1031        int ai = 0;
1032        int e;
```



```
1034        topo_dprintf(mp->tm_hdl, TOPO_DBG_XML, "pgroup_record\n");
1035        if ((name = xmlGetProp(pxn, (xmlChar *)Name)) == NULL) {
1036                topo_dprintf(mp->tm_hdl, TOPO_DBG_ERR,
1037                    "propgroup lacks a name\n");
1038                return (topo_mod_seterrno(mp, ETOPO_PRSR_NOATTR));
1039        }
1040        if (xmlattr_to_int(mp, pxn, Version, &ver) < 0) {
1041                topo_dprintf(mp->tm_hdl, TOPO_DBG_ERR,
1042                    "propgroup lacks a version\n");
1043                return (topo_mod_seterrno(mp, ETOPO_PRSR_NOATTR));
1044        }
1045        if (xmlattr_to_stab(mp, pxn, Namestab, &nmstab) < 0) {
1046                topo_dprintf(mp->tm_hdl, TOPO_DBG_ERR,
1047                    "propgroup lacks name-stability\n");
1048                return (topo_mod_seterrno(mp, ETOPO_PRSR_NOATTR));
1049        }
1050        if (xmlattr_to_stab(mp, pxn, Datastab, &dstab) < 0) {
1051                topo_dprintf(mp->tm_hdl, TOPO_DBG_ERR,
1052                    "propgroup lacks data-stability\n");
1053                return (topo_mod_seterrno(mp, ETOPO_PRSR_NOATTR));
1054        }

1056        topo_dprintf(mp->tm_hdl, TOPO_DBG_XML, "pgroup %s\n", (char *)name);
1057        for (cn = pxn->xmlChildrenNode; cn != NULL; cn = cn->next) {
1058                if (xmlStrcmp(cn->name, (xmlChar *)Propval) == 0)
1059                        pcnt++;
1060        }

1062        if (topo_mod_nvalloc(mp, &pgnvl, NV_UNIQUE_NAME) < 0) {
1063                xmlFree(name);
1064                topo_dprintf(mp->tm_hdl, TOPO_DBG_ERR,
1065                    "failed to allocate propgroup nvlist\n");
1066                return (topo_mod_seterrno(mp, ETOPO_NOMEM));
1067        }

1069        e = nvlist_add_string(pgnvl, INV_PGRP_NAME, (char *)name);
1070        e |= nvlist_add_uint32(pgnvl, INV_PGRP_NMSTAB, nmstab);
1071        e |= nvlist_add_uint32(pgnvl, INV_PGRP_DSTAB, dstab);
1072        e |= nvlist_add_uint32(pgnvl, INV_PGRP_VER, ver);
1073        e |= nvlist_add_uint32(pgnvl, INV_PGRP_NPROP, pcnt);
1074        if (pcnt > 0)
1075                if (e != 0 ||
1076                    (apl = topo_mod_zalloc(mp, pcnt * sizeof (nvlist_t *)))
1077                    == NULL) {
1078                        xmlFree(name);
1079                        nvlist_free(pgnvl);
1080                        topo_dprintf(mp->tm_hdl, TOPO_DBG_ERR,
1081                            "failed to allocate nvlist array for properties"
1082                            "(e=%d)\n", e);
1083                        return (topo_mod_seterrno(mp, ETOPO_NOMEM));
1084                }
1085        for (cn = pxn->xmlChildrenNode; cn != NULL; cn = cn->next) {
1086                if (xmlStrcmp(cn->name, (xmlChar *)Propval) == 0) {
1087                        if (ai < pcnt) {
1088                                if ((apl[ai] = pval_record(mp, cn)) == NULL)
1089                                        break;
1090                        }
1091                        ai++;
1092                } else if (xmlStrcmp(cn->name, (xmlChar *)Prop_meth) == 0) {
1093                        if (pmeth_record(mp, (const char *)name, cn, tn, rname,
1094                            ppgrp_name) < 0)
1095                                break;
1096                }
1097        }
1098        xmlFree(name);
```

```
1099         if (pcnt > 0) {
1100                 e |= (ai != pcnt);
1101                 e |= nvlist_add_nvlist_array(pgnvl, INV_PGRP_ALLPROPS, apl,
1102                     pcnt);
1103                 for (ai = 0; ai < pcnt; ai++)
1105                         if (apl[ai] != NULL)
1104                         nvlist_free(apl[ai]);
1105                 topo_mod_free(mp, apl, pcnt * sizeof (nvlist_t *));
1106                 if (e != 0) {
1107                         nvlist_free(pgnvl);
1108                         return (-1);
1109                 }
1110         }
1111         rpad->tpad_pgs[pi] = pgnvl;
1112         return (0);
1113 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    9119 Mon Feb 15 12:56:07 2016
new/usr/src/lib/fm/topo/modules/SUNW,SPARC-Enterprise/ioboard/opl_hostbridge.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 102 /*
 103  * Create a root complex node.
 104  */
 105 static tnode_t *
 106 opl_rc_node_create(topo_mod_t *mp, tnode_t *parent, di_node_t dnode, int inst)
 107 {
 108         int err;
 109         tnode_t *rcn;
 110         const char *slot_name;
 111         char *dnpath;
 112         nvlist_t *mod;

 114         rcn = opl_node_create(mp, parent, PCIEX_ROOT, inst, (void *)dnode);
 115         if (rcn == NULL) {
 116                 return (NULL);
 117         }

 119         /*
 120          * If this root complex connects to a slot, it will have a
 121          * slot-names property.
 122          */
 123         slot_name = opl_get_slot_name(mp, dnode);
 124         if (slot_name) {
 125                 char fru_str[64];
 126                 nvlist_t *fru_fmri;
 127                 /* Add FRU fmri */
 128                 (void) snprintf(fru_str, sizeof (fru_str), "hc:///component=%s",
 129                     slot_name);
 130                 if (topo_mod_str2nvl(mp, fru_str, &fru_fmri) == 0) {
 131                         (void) topo_node_fru_set(rcn, fru_fmri, 0, &err);
 132                         nvlist_free(fru_fmri);
 133                 }
 134                 /* Add label */
 135                 (void) topo_node_label_set(rcn, (char *)slot_name, &err);
 136         } else {
 137                 /* Inherit parent FRU's label */
 138                 (void) topo_node_fru_set(rcn, NULL, 0, &err);
 139                 (void) topo_node_label_set(rcn, NULL, &err);
 140         }

 142         /*
 143          * Set ASRU to be the dev-scheme ASRU
 144          */
 145         if ((dnpath = di_devfs_path(dnode)) != NULL) {
 146                 nvlist_t *fmri;

 148                 fmri = topo_mod_devfmri(mp, FM_DEV_SCHEME_VERSION,
 149                     dnpath, NULL);
 150                 if (fmri == NULL) {
 151                         topo_mod_dprintf(mp,
 152                             "dev:///%s fmri creation failed.\n",
 153                             dnpath);
 154                         (void) topo_mod_seterrno(mp, err);
 155                         di_devfs_path_free(dnpath);
 156                         return (NULL);
 157                 }
 158                 if (topo_node_asru_set(rcn, fmri, 0, &err) < 0) {
 159                         topo_mod_dprintf(mp, "topo_node_asru_set failed\n");
 160                         (void) topo_mod_seterrno(mp, err);
```



```
 161                         nvlist_free(fmri);
 162                         di_devfs_path_free(dnpath);
 163                         return (NULL);
 164                 }
 165                 nvlist_free(fmri);
 166         } else {
 167                 topo_mod_dprintf(mp, "NULL di_devfs_path.\n");
 168         }

 170         /*
 171          * Set pciexrc properties for root complex nodes
 172          */

 174         /* Add the io and pci property groups */
 175         if (topo_pgroup_create(rcn, &io_pgroup, &err) < 0) {
 176                 topo_mod_dprintf(mp, "topo_pgroup_create failed\n");
 177                 di_devfs_path_free(dnpath);
 178                 (void) topo_mod_seterrno(mp, err);
 179                 return (NULL);
 180         }
 181         if (topo_pgroup_create(rcn, &pci_pgroup, &err) < 0) {
 182                 topo_mod_dprintf(mp, "topo_pgroup_create failed\n");
 183                 di_devfs_path_free(dnpath);
 184                 (void) topo_mod_seterrno(mp, err);
 185                 return (NULL);
 186         }
 187         /* Add the devfs path property */
 188         if (dnpath) {
 189                 if (topo_prop_set_string(rcn, TOPO_PGROUP_IO, TOPO_IO_DEV,
 190                     TOPO_PROP_IMMUTABLE, dnpath, &err) != 0) {
 191                         topo_mod_dprintf(mp, "Failed to set DEV property\n");
 192                         di_devfs_path_free(dnpath);
 193                         (void) topo_mod_seterrno(mp, err);
 194                 }
 195                 di_devfs_path_free(dnpath);
 196         }
 197         /* Oberon device type is always "pciex" */
 198         if (topo_prop_set_string(rcn, TOPO_PGROUP_IO, TOPO_IO_DEVTYPE,
 199             TOPO_PROP_IMMUTABLE, OPL_PX_DEVTYPE, &err) != 0) {
 200                 topo_mod_dprintf(mp, "Failed to set DEVTYPE property\n");
 201         }
 202         /* Oberon driver is always "px" */
 203         if (topo_prop_set_string(rcn, TOPO_PGROUP_IO, TOPO_IO_DRIVER,
 204             TOPO_PROP_IMMUTABLE, OPL_PX_DRV, &err) != 0) {
 205                 topo_mod_dprintf(mp, "Failed to set DRIVER property\n");
 206         }
 207         if ((mod = topo_mod_modfmri(mp, FM_MOD_SCHEME_VERSION, OPL_PX_DRV))
 208             == NULL || topo_prop_set_fmri(rcn, TOPO_PGROUP_IO,
 209             TOPO_IO_MODULE, TOPO_PROP_IMMUTABLE, mod,  &err) != 0) {
 210                 topo_mod_dprintf(mp, "Failed to set MODULE property\n");
 211         }
 212         if (mod != NULL)
 212         nvlist_free(mod);

 214         /* This is a PCIEX Root Complex */
 215         if (topo_prop_set_string(rcn, TOPO_PGROUP_PCI, TOPO_PCI_EXCAP,
 216             TOPO_PROP_IMMUTABLE, PCIEX_ROOT, &err) != 0) {
 217                 topo_mod_dprintf(mp, "Failed to set EXCAP property\n");
 218         }
 219         /* BDF of Oberon root complex is constant */
 220         if (topo_prop_set_string(rcn, TOPO_PGROUP_PCI,
 221             TOPO_PCI_BDF, TOPO_PROP_IMMUTABLE, OPL_PX_BDF, &err) != 0) {
 222                 topo_mod_dprintf(mp, "Failed to set EXCAP property\n");
 223         }

 225         /* Make room for children */
```

```
 226         (void) topo_node_range_create(mp, rcn, PCIEX_BUS, 0, OPL_BUS_MAX);
 227         return (rcn);
 228 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   28650 Mon Feb 15 12:56:07 2016
new/usr/src/lib/fm/topo/modules/common/disk/disk_common.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 109 /*
 110  * Set the properties of the disk node, from dev_di_node_t data.
 111  * Properties include:
 112  *      group: protocol  properties: resource, asru, label, fru
 113  *      group: authority properties: product-id, chasis-id, server-id
 114  *      group: io        properties: devfs-path, devid
 115  *      group: storage   properties:
 116  *              - logical-disk, disk-model, disk-manufacturer, serial-number
 117  *              - firmware-revision, capacity-in-bytes
 118  *
 119  * NOTE: the io and storage groups won't be present if the dnode passed in is
 120  * NULL. This happens when a disk is found through ses, but is not enumerated
 121  * in the devinfo tree.
 122  */
 123 static int
 124 disk_set_props(topo_mod_t *mod, tnode_t *parent,
 125     tnode_t *dtn, dev_di_node_t *dnode)
 126 {
 127         nvlist_t        *asru = NULL;
 128         char            *label = NULL;
 129         nvlist_t        *fmri = NULL;
 130         int             err;

 132         /* pull the label property down from our parent 'bay' node */
 133         if (topo_node_label(parent, &label, &err) != 0) {
 134                 topo_mod_dprintf(mod, "disk_set_props: "
 135                     "label error %s\n", topo_strerror(err));
 136                 goto error;
 137         }
 138         if (topo_node_label_set(dtn, label, &err) != 0) {
 139                 topo_mod_dprintf(mod, "disk_set_props: "
 140                     "label_set error %s\n", topo_strerror(err));
 141                 goto error;
 142         }

 144         /* get the resource fmri, and use it as the fru */
 145         if (topo_node_resource(dtn, &fmri, &err) != 0) {
 146                 topo_mod_dprintf(mod, "disk_set_props: "
 147                     "resource error: %s\n", topo_strerror(err));
 148                 goto error;
 149         }
 150         if (topo_node_fru_set(dtn, fmri, 0, &err) != 0) {
 151                 topo_mod_dprintf(mod, "disk_set_props: "
 152                     "fru_set error: %s\n", topo_strerror(err));
 153                 goto error;
 154         }

 156         /* create/set the authority group */
 157         if ((topo_pgroup_create(dtn, &disk_auth_pgroup, &err) != 0) &&
 158             (err != ETOPO_PROP_DEFD)) {
 159                 topo_mod_dprintf(mod, "disk_set_props: "
 160                     "create disk_auth error %s\n", topo_strerror(err));
 161                 goto error;
 162         }

 164         /* create the storage group */
 165         if (topo_pgroup_create(dtn, &storage_pgroup, &err) != 0) {
 166                 topo_mod_dprintf(mod, "disk_set_props: "
 167                     "create storage error %s\n", topo_strerror(err));
```



```
 168                 goto error;
 169         }

 171         /* no dnode was found for this disk - skip the io and storage groups */
 172         if (dnode == NULL) {
 173                 err = 0;
 174                 goto out;
 175         }

 177         /* form and set the asru */
 178         if ((asru = topo_mod_devfmri(mod, FM_DEV_SCHEME_VERSION,
 179             dnode->ddn_dpath, dnode->ddn_devid)) == NULL) {
 180                 err = ETOPO_FMRI_UNKNOWN;
 181                 topo_mod_dprintf(mod, "disk_set_props: "
 182                     "asru error %s\n", topo_strerror(err));
 183                 goto error;
 184         }
 185         if (topo_node_asru_set(dtn, asru, 0, &err) != 0) {
 186                 topo_mod_dprintf(mod, "disk_set_props: "
 187                     "asru_set error %s\n", topo_strerror(err));
 188                 goto error;
 189         }

 191         /* create/set the devfs-path and devid in the io group */
 192         if (topo_pgroup_create(dtn, &io_pgroup, &err) != 0) {
 193                 topo_mod_dprintf(mod, "disk_set_props: "
 194                     "create io error %s\n", topo_strerror(err));
 195                 goto error;
 196         }

 198         if (topo_prop_set_string(dtn, TOPO_PGROUP_IO, TOPO_IO_DEV_PATH,
 199             TOPO_PROP_IMMUTABLE, dnode->ddn_dpath, &err) != 0) {
 200                 topo_mod_dprintf(mod, "disk_set_props: "
 201                     "set dev error %s\n", topo_strerror(err));
 202                 goto error;
 203         }

 205         if (dnode->ddn_devid && topo_prop_set_string(dtn, TOPO_PGROUP_IO,
 206             TOPO_IO_DEVID, TOPO_PROP_IMMUTABLE, dnode->ddn_devid, &err) != 0) {
 207                 topo_mod_dprintf(mod, "disk_set_props: "
 208                     "set devid error %s\n", topo_strerror(err));
 209                 goto error;
 210         }

 212         if (dnode->ddn_ppath_count != 0 &&
 213             topo_prop_set_string_array(dtn, TOPO_PGROUP_IO, TOPO_IO_PHYS_PATH,
 214             TOPO_PROP_IMMUTABLE, (const char **)dnode->ddn_ppath,
 215             dnode->ddn_ppath_count, &err) != 0) {
 216                 topo_mod_dprintf(mod, "disk_set_props: "
 217                     "set phys-path error %s\n", topo_strerror(err));
 218                 goto error;
 219         }

 221         /* set the storage group public /dev name */
 222         if (dnode->ddn_lpath != NULL &&
 223             topo_prop_set_string(dtn, TOPO_PGROUP_STORAGE,
 224             TOPO_STORAGE_LOGICAL_DISK_NAME, TOPO_PROP_IMMUTABLE,
 225             dnode->ddn_lpath, &err) != 0) {
 226                 topo_mod_dprintf(mod, "disk_set_props: "
 227                     "set disk_name error %s\n", topo_strerror(err));
 228                 goto error;
 229         }

 231         /* populate other misc storage group properties */
 232         if (dnode->ddn_mfg && (topo_prop_set_string(dtn, TOPO_PGROUP_STORAGE,
 233             TOPO_STORAGE_MANUFACTURER, TOPO_PROP_IMMUTABLE,
```

```
234             dnode->ddn_mfg, &err) != 0)) {
235                 topo_mod_dprintf(mod, "disk_set_props: "
236                     "set mfg error %s\n", topo_strerror(err));
237                 goto error;
238         }
239         if (dnode->ddn_model && (topo_prop_set_string(dtn, TOPO_PGROUP_STORAGE,
240             TOPO_STORAGE_MODEL, TOPO_PROP_IMMUTABLE,
241             dnode->ddn_model, &err) != 0)) {
242                 topo_mod_dprintf(mod, "disk_set_props: "
243                     "set model error %s\n", topo_strerror(err));
244                 goto error;
245         }
246         if (dnode->ddn_serial && (topo_prop_set_string(dtn, TOPO_PGROUP_STORAGE,
247             TOPO_STORAGE_SERIAL_NUM, TOPO_PROP_IMMUTABLE,
248             dnode->ddn_serial, &err) != 0)) {
249                 topo_mod_dprintf(mod, "disk_set_props: "
250                     "set serial error %s\n", topo_strerror(err));
251                 goto error;
252         }
253         if (dnode->ddn_firm && (topo_prop_set_string(dtn, TOPO_PGROUP_STORAGE,
254             TOPO_STORAGE_FIRMWARE_REV, TOPO_PROP_IMMUTABLE,
255             dnode->ddn_firm, &err) != 0)) {
256                 topo_mod_dprintf(mod, "disk_set_props: "
257                     "set firm error %s\n", topo_strerror(err));
258                 goto error;
259         }
260         if (dnode->ddn_cap && (topo_prop_set_string(dtn, TOPO_PGROUP_STORAGE,
261             TOPO_STORAGE_CAPACITY, TOPO_PROP_IMMUTABLE,
262             dnode->ddn_cap, &err) != 0)) {
263                 topo_mod_dprintf(mod, "disk_set_props: "
264                     "set cap error %s\n", topo_strerror(err));
265                 goto error;
266         }
267         err = 0;

269 out:
269 out:    if (fmri)
270         nvlist_free(fmri);
271         if (label)
272                 topo_mod_strfree(mod, label);
273         if (asru)
273         nvlist_free(asru);
274         return (err);

276 error:  err = topo_mod_seterrno(mod, err);
277         goto out;
278 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   27788 Mon Feb 15 12:56:07 2016
new/usr/src/lib/fm/topo/modules/common/pcibus/did_props.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_


/*ARGSUSED*/
static int
FRU_set(tnode_t *tn, did_t *pd,
    const char *dpnm, const char *tpgrp, const char *tpnm)
{
        topo_mod_t *mp;
        char *nm;
        int e = 0, err = 0;

        nm = topo_node_name(tn);
        mp = did_mod(pd);

        /*
         * If this is a PCIEX_BUS and its parent is a PCIEX_ROOT,
         * check for a CPUBOARD predecessor.  If found, inherit its
         * parent's FRU.  Otherwise, continue with FRU set.
         */
        if ((strcmp(nm, PCIEX_BUS) == 0) &&
            (strcmp(topo_node_name(topo_node_parent(tn)), PCIEX_ROOT) == 0)) {

                if (use_predecessor_fru(tn, CPUBOARD) == 0)
                        return (0);
        }
        /*
         * If this topology node represents something other than an
         * ioboard or a device that implements a slot, inherit the
         * parent's FRU value.  If there is no label, inherit our
         * parent's FRU value.  Otherwise, munge up an fmri based on
         * the label.
         */
        if (strcmp(nm, IOBOARD) != 0 && strcmp(nm, PCI_DEVICE) != 0 &&
            strcmp(nm, PCIEX_DEVICE) != 0 && strcmp(nm, PCIEX_BUS) != 0) {
                (void) topo_node_fru_set(tn, NULL, 0, &e);
                return (0);
        }

        /*
         * If ioboard, set fru fmri to hc fmri
         */
        if (strcmp(nm, IOBOARD) == 0) {
                e = FRU_fmri_set(mp, tn);
                return (e);
        } else if (strcmp(nm, PCI_DEVICE) == 0 ||
            strcmp(nm, PCIEX_DEVICE) == 0 || strcmp(nm, PCIEX_BUS) == 0) {
                nvlist_t *in, *out;

                mp = did_mod(pd);
                if (topo_mod_nvalloc(mp, &in, NV_UNIQUE_NAME) != 0)
                        return (topo_mod_seterrno(mp, EMOD_FMRI_NVL));
                if (nvlist_add_uint64(in, "nv1", (uintptr_t)pd) != 0) {
                        nvlist_free(in);
                        return (topo_mod_seterrno(mp, EMOD_NOMEM));
                }
                if (topo_method_invoke(tn,
                    TOPO_METH_FRU_COMPUTE, TOPO_METH_FRU_COMPUTE_VERSION,
                    in, &out, &err) != 0) {
                        nvlist_free(in);
                        return (topo_mod_seterrno(mp, err));
```



```
                }
                nvlist_free(in);
                (void) topo_node_fru_set(tn, out, 0, &err);
                if (out != NULL)
                nvlist_free(out);
        } else
                (void) topo_node_fru_set(tn, NULL, 0, &err);

        return (0);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   24305 Mon Feb 15 12:56:07 2016
new/usr/src/lib/fm/topo/modules/i86pc/chip/chip_amd.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 636 static int
 637 amd_dramchan_create(topo_mod_t *mod, tnode_t *pnode, const char *name,
 638     nvlist_t *auth)
 639 {
 640         tnode_t *chnode;
 641         nvlist_t *fmri;
 642         char *socket;
 643         int i, nchan;
 644         nvlist_t *pfmri = NULL;
 645         int err, nerr = 0;

 647         /*
 648          * We will enumerate the number of channels present even if only
 649          * channel A is in use (i.e., running in 64-bit mode).  Only
 650          * the socket 754 package has a single channel.
 651          */
 652         if (topo_prop_get_string(pnode, PGNAME(MCT), "socket",
 653             &socket, &err) == 0 && strcmp(socket, "Socket 754") == 0)
 654                 nchan = 1;
 655         else
 656                 nchan = 2;

 658         topo_mod_strfree(mod, socket);

 660         if (topo_node_range_create(mod, pnode, name, 0, nchan - 1) < 0)
 661                 return (-1);

 663         (void) topo_node_fru(pnode, &pfmri, NULL, &err);

 665         for (i = 0; i < nchan; i++) {
 666                 if (mkrsrc(mod, pnode, name, i, auth, &fmri) != 0) {
 667                         whinge(mod, &nerr, "amd_dramchan_create: mkrsrc "
 668                             "failed\n");
 669                         continue;
 670                 }

 672                 if ((chnode = topo_node_bind(mod, pnode, name, i, fmri))
 673                     == NULL) {
 674                         nvlist_free(fmri);
 675                         whinge(mod, &nerr, "amd_dramchan_create: node bind "
 676                             "failed\n");
 677                         continue;
 678                 }

 680                 (void) topo_node_asru_set(chnode, fmri, 0, &err);
 681                 if (pfmri)
 682                         (void) topo_node_fru_set(chnode, pfmri, 0, &err);

 684                 nvlist_free(fmri);

 686                 (void) topo_pgroup_create(chnode, &chan_pgroup, &err);

 688                 (void) topo_prop_set_string(chnode, PGNAME(CHAN), "channel",
 689                     TOPO_PROP_IMMUTABLE, i == 0 ? "A" : "B", &err);
 690         }
 691         if (pfmri)
 691         nvlist_free(pfmri);

 693         return (nerr == 0 ? 0 : -1);
```



```
 694 }
______unchanged_portion_omitted_
```

```
**********************************************************
    8699 Mon Feb 15 12:56:07 2016
new/usr/src/lib/fm/topo/modules/sun4v/cpuboard/cpuboard_hostbridge.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

  79 /*
  80  * cpuboard_rc_node_create()
  81  * Description:
  82  *     Create a root complex node pciexrc
  83  * Parameters:
  84  *     mp: topo module pointer
  85  *     parent: topo parent node of the newly created pciexrc node
  86  *     dnode: Solaris device node of the root complex
  87  *     rcpath: Used to populated the dev property of the topo pciexrc node if
  88  *          the local host does not own the root complex.
  89  */
  90 static tnode_t *
  91 cpuboard_rc_node_create(topo_mod_t *mp, tnode_t *parent, di_node_t dnode,
  92     char *rcpath, int inst)
  93 {
  94         int err;
  95         tnode_t *rcn;
  96         char *dnpath;
  97         nvlist_t *mod;

  99         topo_mod_dprintf(mp, "cpuboard_rc_node_create:\n");

 101         rcn = cpuboard_node_create(mp, parent, PCIEX_ROOT, inst, (void *)dnode);
 102         if (rcn == NULL) {
 103                 return (NULL);
 104         }

 106         /* Inherit parent FRU's label */
 107         (void) topo_node_fru_set(rcn, NULL, 0, &err);
 108         (void) topo_node_label_set(rcn, NULL, &err);

 110         /*
 111          * Set ASRU to be the dev-scheme ASRU
 112          */
 113         if ((dnpath = di_devfs_path(dnode)) != NULL) {
 114                 nvlist_t *fmri;

 116                 /*
 117                  * The local host owns the root complex, so use the dev path
 118                  * from the di_devfs_path(), instead of the passed in rcpath,
 119                  * to populate the dev property.
 120                  */
 121                 rcpath = dnpath;
 122                 fmri = topo_mod_devfmri(mp, FM_DEV_SCHEME_VERSION,
 123                     dnpath, NULL);
 124                 if (fmri == NULL) {
 125                         topo_mod_dprintf(mp,
 126                             "dev:///%s fmri creation failed.\n",
 127                             dnpath);
 128                         (void) topo_mod_seterrno(mp, err);
 129                         di_devfs_path_free(dnpath);
 130                         return (NULL);
 131                 }
 132                 if (topo_node_asru_set(rcn, fmri, 0, &err) < 0) {
 133                         topo_mod_dprintf(mp, "topo_node_asru_set failed\n");
 134                         (void) topo_mod_seterrno(mp, err);
 135                         nvlist_free(fmri);
 136                         di_devfs_path_free(dnpath);
 137                         return (NULL);
```



```
 138                 }
 139                 nvlist_free(fmri);
 140         } else {
 141                 topo_mod_dprintf(mp, "NULL di_devfs_path.\n");
 142         }

 144         /*
 145          * Set pciexrc properties for root complex nodes
 146          */

 148         /* Add the io and pci property groups */
 149         if (topo_pgroup_create(rcn, &io_pgroup, &err) < 0) {
 150                 topo_mod_dprintf(mp, "topo_pgroup_create failed\n");
 151                 di_devfs_path_free(dnpath);
 152                 (void) topo_mod_seterrno(mp, err);
 153                 return (NULL);
 154         }
 155         if (topo_pgroup_create(rcn, &pci_pgroup, &err) < 0) {
 156                 topo_mod_dprintf(mp, "topo_pgroup_create failed\n");
 157                 di_devfs_path_free(dnpath);
 158                 (void) topo_mod_seterrno(mp, err);
 159                 return (NULL);
 160         }
 161         /* Add the devfs path property */
 162         if (rcpath) {
 163                 if (topo_prop_set_string(rcn, TOPO_PGROUP_IO, TOPO_IO_DEV,
 164                     TOPO_PROP_IMMUTABLE, rcpath, &err) != 0) {
 165                         topo_mod_dprintf(mp, "Failed to set DEV property\n");
 166                         (void) topo_mod_seterrno(mp, err);
 167                 }
 168         }
 169         if (dnpath) {
 170                 di_devfs_path_free(dnpath);
 171         }
 172         /* T5440 device type is always "pciex" */
 173         if (topo_prop_set_string(rcn, TOPO_PGROUP_IO, TOPO_IO_DEVTYPE,
 174             TOPO_PROP_IMMUTABLE, CPUBOARD_PX_DEVTYPE, &err) != 0) {
 175                 topo_mod_dprintf(mp, "Failed to set DEVTYPE property\n");
 176         }
 177         /* T5440 driver is always "px" */
 178         if (topo_prop_set_string(rcn, TOPO_PGROUP_IO, TOPO_IO_DRIVER,
 179             TOPO_PROP_IMMUTABLE, CPUBOARD_PX_DRV, &err) != 0) {
 180                 topo_mod_dprintf(mp, "Failed to set DRIVER property\n");
 181         }
 182         if ((mod = topo_mod_modfmri(mp, FM_MOD_SCHEME_VERSION, CPUBOARD_PX_DRV))
 183             == NULL || topo_prop_set_fmri(rcn, TOPO_PGROUP_IO,
 184             TOPO_IO_MODULE, TOPO_PROP_IMMUTABLE, mod,  &err) != 0) {
 185                 topo_mod_dprintf(mp, "Failed to set MODULE property\n");
 186         }
 187         if (mod != NULL)
 187         nvlist_free(mod);

 189         /* This is a PCIEX Root Complex */
 190         if (topo_prop_set_string(rcn, TOPO_PGROUP_PCI, TOPO_PCI_EXCAP,
 191             TOPO_PROP_IMMUTABLE, PCIEX_ROOT, &err) != 0) {
 192                 topo_mod_dprintf(mp, "Failed to set EXCAP property\n");
 193         }
 194         /* BDF of T5440 root complex is constant */
 195         if (topo_prop_set_string(rcn, TOPO_PGROUP_PCI,
 196             TOPO_PCI_BDF, TOPO_PROP_IMMUTABLE, CPUBOARD_PX_BDF, &err) != 0) {
 197                 topo_mod_dprintf(mp, "Failed to set EXCAP property\n");
 198         }

 200         /* Make room for children */
 201         (void) topo_node_range_create(mp, rcn, PCIEX_BUS, 0, CPUBOARD_MAX);
 202         return (rcn);
```

```
 203 }
______unchanged_portion_omitted_
```

```
**********************************************************
   86268 Mon Feb 15 12:56:07 2016
new/usr/src/lib/libbe/common/be_create.c
patch tsoome-feedback
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2014 by Delphix. All rights reserved.
 */

/*
 * System includes
 */

#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <libgen.h>
#include <libintl.h>
#include <libnvpair.h>
#include <libzfs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mnttab.h>
#include <sys/mount.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#include <libbe.h>
#include <libbe_priv.h>

/* Library wide variables */
libzfs_handle_t *g_zfs = NULL;

/* Private function prototypes */
static int _be_destroy(const char *, be_destroy_data_t *);
static int be_destroy_zones(char *, char *, be_destroy_data_t *);
static int be_destroy_zone_roots(char *, be_destroy_data_t *);
static int be_destroy_zone_roots_callback(zfs_handle_t *, void *);
static int be_copy_zones(char *, char *, char *);
static int be_clone_fs_callback(zfs_handle_t *, void *);
```



```
static int be_destroy_callback(zfs_handle_t *, void *);
static int be_send_fs_callback(zfs_handle_t *, void *);
static int be_demote_callback(zfs_handle_t *, void *);
static int be_demote_find_clone_callback(zfs_handle_t *, void *);
static int be_has_snapshot_callback(zfs_handle_t *, void *);
static int be_demote_get_one_clone(zfs_handle_t *, void *);
static int be_get_snap(char *, char **);
static int be_prep_clone_send_fs(zfs_handle_t *, be_transaction_data_t *,
    char *, int);
static boolean_t be_create_container_ds(char *);
static char *be_get_zone_be_name(char *root_ds, char *container_ds);
static int be_zone_root_exists_callback(zfs_handle_t *, void *);

/* ********************************************************************** */
/*                      Public Functions                                  */
/* ********************************************************************** */

/*
 * Function:    be_init
 * Description: Creates the initial datasets for a BE and leaves them
 *              unpopulated.  The resultant BE can be mounted but can't
 *              yet be activated or booted.
 * Parameters:
 *              be_attrs - pointer to nvlist_t of attributes being passed in.
 *                      The following attributes are used by this function:
 *
 *                      BE_ATTR_NEW_BE_NAME             *required
 *                      BE_ATTR_NEW_BE_POOL             *required
 *                      BE_ATTR_ZFS_PROPERTIES          *optional
 *                      BE_ATTR_FS_NAMES                *optional
 *                      BE_ATTR_FS_NUM                  *optional
 *                      BE_ATTR_SHARED_FS_NAMES         *optional
 *                      BE_ATTR_SHARED_FS_NUM           *optional
 * Return:
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Public
 */
int
be_init(nvlist_t *be_attrs)
{
        be_transaction_data_t   bt = { 0 };
        zpool_handle_t  *zlp;
        nvlist_t        *zfs_props = NULL;
        char            nbe_root_ds[MAXPATHLEN];
        char            child_fs[MAXPATHLEN];
        char            **fs_names = NULL;
        char            **shared_fs_names = NULL;
        uint16_t        fs_num = 0;
        uint16_t        shared_fs_num = 0;
        int             nelem;
        int             i;
        int             zret = 0, ret = BE_SUCCESS;

        /* Initialize libzfs handle */
        if (!be_zfs_init())
                return (BE_ERR_INIT);

        /* Get new BE name */
        if (nvlist_lookup_string(be_attrs, BE_ATTR_NEW_BE_NAME, &bt.nbe_name)
            != 0) {
                be_print_err(gettext("be_init: failed to lookup "
                    "BE_ATTR_NEW_BE_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }
```

```
129         /* Validate new BE name */
130         if (!be_valid_be_name(bt.nbe_name)) {
131                 be_print_err(gettext("be_init: invalid BE name %s\n"),
132                     bt.nbe_name);
133                 return (BE_ERR_INVAL);
134         }

136         /* Get zpool name */
137         if (nvlist_lookup_string(be_attrs, BE_ATTR_NEW_BE_POOL, &bt.nbe_zpool)
138             != 0) {
139                 be_print_err(gettext("be_init: failed to lookup "
140                     "BE_ATTR_NEW_BE_POOL attribute\n"));
141                 return (BE_ERR_INVAL);
142         }

144         /* Get file system attributes */
145         nelem = 0;
146         if (nvlist_lookup_pairs(be_attrs, 0,
147             BE_ATTR_FS_NUM, DATA_TYPE_UINT16, &fs_num,
148             BE_ATTR_FS_NAMES, DATA_TYPE_STRING_ARRAY, &fs_names, &nelem,
149             NULL) != 0) {
150                 be_print_err(gettext("be_init: failed to lookup fs "
151                     "attributes\n"));
152                 return (BE_ERR_INVAL);
153         }
154         if (nelem != fs_num) {
155                 be_print_err(gettext("be_init: size of FS_NAMES array (%d) "
156                     "does not match FS_NUM (%d)\n"), nelem, fs_num);
157                 return (BE_ERR_INVAL);
158         }

160         /* Get shared file system attributes */
161         nelem = 0;
162         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
163             BE_ATTR_SHARED_FS_NUM, DATA_TYPE_UINT16, &shared_fs_num,
164             BE_ATTR_SHARED_FS_NAMES, DATA_TYPE_STRING_ARRAY, &shared_fs_names,
165             &nelem, NULL) != 0) {
166                 be_print_err(gettext("be_init: failed to lookup "
167                     "shared fs attributes\n"));
168                 return (BE_ERR_INVAL);
169         }
170         if (nelem != shared_fs_num) {
171                 be_print_err(gettext("be_init: size of SHARED_FS_NAMES "
172                     "array does not match SHARED_FS_NUM\n"));
173                 return (BE_ERR_INVAL);
174         }

176         /* Verify that nbe_zpool exists */
177         if ((zlp = zpool_open(g_zfs, bt.nbe_zpool)) == NULL) {
178                 be_print_err(gettext("be_init: failed to "
179                     "find existing zpool (%s): %s\n"), bt.nbe_zpool,
180                     libzfs_error_description(g_zfs));
181                 return (zfs_err_to_be_err(g_zfs));
182         }
183         zpool_close(zlp);

185         /*
186          * Verify BE container dataset in nbe_zpool exists.
187          * If not, create it.
188          */
189         if (!be_create_container_ds(bt.nbe_zpool))
190                 return (BE_ERR_CREATDS);

192         /*
193          * Verify that nbe_name doesn't already exist in some pool.
```



```
194          */
195         if ((zret = zpool_iter(g_zfs, be_exists_callback, bt.nbe_name)) > 0) {
196                 be_print_err(gettext("be_init: BE (%s) already exists\n"),
197                     bt.nbe_name);
198                 return (BE_ERR_BE_EXISTS);
199         } else if (zret < 0) {
200                 be_print_err(gettext("be_init: zpool_iter failed: %s\n"),
201                     libzfs_error_description(g_zfs));
202                 return (zfs_err_to_be_err(g_zfs));
203         }

205         /* Generate string for BE's root dataset */
206         be_make_root_ds(bt.nbe_zpool, bt.nbe_name, nbe_root_ds,
207             sizeof (nbe_root_ds));

209         /*
210          * Create property list for new BE root dataset.  If some
211          * zfs properties were already provided by the caller, dup
212          * that list.  Otherwise initialize a new property list.
213          */
214         if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
215             BE_ATTR_ZFS_PROPERTIES, DATA_TYPE_NVLIST, &zfs_props, NULL)
216             != 0) {
217                 be_print_err(gettext("be_init: failed to lookup "
218                     "BE_ATTR_ZFS_PROPERTIES attribute\n"));
219                 return (BE_ERR_INVAL);
220         }
221         if (zfs_props != NULL) {
222                 /* Make sure its a unique nvlist */
223                 if (!(zfs_props->nvl_nvflag & NV_UNIQUE_NAME) &&
224                     !(zfs_props->nvl_nvflag & NV_UNIQUE_NAME_TYPE)) {
225                         be_print_err(gettext("be_init: ZFS property list "
226                             "not unique\n"));
227                         return (BE_ERR_INVAL);
228                 }

230                 /* Dup the list */
231                 if (nvlist_dup(zfs_props, &bt.nbe_zfs_props, 0) != 0) {
232                         be_print_err(gettext("be_init: failed to dup ZFS "
233                             "property list\n"));
234                         return (BE_ERR_NOMEM);
235                 }
236         } else {
237                 /* Initialize new nvlist */
238                 if (nvlist_alloc(&bt.nbe_zfs_props, NV_UNIQUE_NAME, 0) != 0) {
239                         be_print_err(gettext("be_init: internal "
240                             "error: out of memory\n"));
241                         return (BE_ERR_NOMEM);
242                 }
243         }

245         /* Set the mountpoint property for the root dataset */
246         if (nvlist_add_string(bt.nbe_zfs_props,
247             zfs_prop_to_name(ZFS_PROP_MOUNTPOINT), "/") != 0) {
248                 be_print_err(gettext("be_init: internal error "
249                     "out of memory\n"));
250                 ret = BE_ERR_NOMEM;
251                 goto done;
252         }

254         /* Set the 'canmount' property */
255         if (nvlist_add_string(bt.nbe_zfs_props,
256             zfs_prop_to_name(ZFS_PROP_CANMOUNT), "noauto") != 0) {
257                 be_print_err(gettext("be_init: internal error "
258                     "out of memory\n"));
259                 ret = BE_ERR_NOMEM;
```

```
260                 goto done;
261         }

263         /* Create BE root dataset for the new BE */
264         if (zfs_create(g_zfs, nbe_root_ds, ZFS_TYPE_FILESYSTEM,
265             bt.nbe_zfs_props) != 0) {
266                 be_print_err(gettext("be_init: failed to "
267                     "create BE root dataset (%s): %s\n"), nbe_root_ds,
268                     libzfs_error_description(g_zfs));
269                 ret = zfs_err_to_be_err(g_zfs);
270                 goto done;
271         }

273         /* Set UUID for new BE */
274         if ((ret = be_set_uuid(nbe_root_ds)) != BE_SUCCESS) {
275                 be_print_err(gettext("be_init: failed to "
276                     "set uuid for new BE\n"));
277         }

279         /*
280          * Clear the mountpoint property so that the non-shared
281          * file systems created below inherit their mountpoints.
282          */
283         (void) nvlist_remove(bt.nbe_zfs_props,
284             zfs_prop_to_name(ZFS_PROP_MOUNTPOINT), DATA_TYPE_STRING);

286         /* Create the new BE's non-shared file systems */
287         for (i = 0; i < fs_num && fs_names[i]; i++) {
288                 /*
289                  * If fs == "/", skip it;
290                  * we already created the root dataset
291                  */
292                 if (strcmp(fs_names[i], "/") == 0)
293                         continue;

295                 /* Generate string for file system */
296                 (void) snprintf(child_fs, sizeof (child_fs), "%s%s",
297                     nbe_root_ds, fs_names[i]);

299                 /* Create file system */
300                 if (zfs_create(g_zfs, child_fs, ZFS_TYPE_FILESYSTEM,
301                     bt.nbe_zfs_props) != 0) {
302                         be_print_err(gettext("be_init: failed to create "
303                             "BE's child dataset (%s): %s\n"), child_fs,
304                             libzfs_error_description(g_zfs));
305                         ret = zfs_err_to_be_err(g_zfs);
306                         goto done;
307                 }
308         }

310         /* Create the new BE's shared file systems */
311         if (shared_fs_num > 0) {
312                 nvlist_t        *props = NULL;

314                 if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0) {
315                         be_print_err(gettext("be_init: nvlist_alloc failed\n"));
316                         ret = BE_ERR_NOMEM;
317                         goto done;
318                 }

320                 for (i = 0; i < shared_fs_num; i++) {
321                         /* Generate string for shared file system */
322                         (void) snprintf(child_fs, sizeof (child_fs), "%s%s",
323                             bt.nbe_zpool, shared_fs_names[i]);

325                         if (nvlist_add_string(props,
```



```
326                             zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
327                             shared_fs_names[i]) != 0) {
328                                 be_print_err(gettext("be_init: "
329                                     "internal error: out of memory\n"));
330                                 nvlist_free(props);
331                                 ret = BE_ERR_NOMEM;
332                                 goto done;
333                         }

335                         /* Create file system if it doesn't already exist */
336                         if (zfs_dataset_exists(g_zfs, child_fs,
337                             ZFS_TYPE_FILESYSTEM)) {
338                                 continue;
339                         }
340                         if (zfs_create(g_zfs, child_fs, ZFS_TYPE_FILESYSTEM,
341                             props) != 0) {
342                                 be_print_err(gettext("be_init: failed to "
343                                     "create BE's shared dataset (%s): %s\n"),
344                                     child_fs, libzfs_error_description(g_zfs));
345                                 ret = zfs_err_to_be_err(g_zfs);
346                                 nvlist_free(props);
347                                 goto done;
348                         }
349                 }

351                 nvlist_free(props);
352         }

354 done:
355         if (bt.nbe_zfs_props != NULL)
355         nvlist_free(bt.nbe_zfs_props);

357         be_zfs_fini();

359         return (ret);
360 }
```

unchanged_portion_omitted

```
567 /*
568  * Function:    be_copy
569  * Description: This function makes a copy of an existing BE.  If the original
570  *              BE and the new BE are in the same pool, it uses zfs cloning to
571  *              create the new BE, otherwise it does a physical copy.
572  *              If the original BE name isn't provided, it uses the currently
573  *              booted BE.  If the new BE name isn't provided, it creates an
574  *              auto named BE and returns that name to the caller.
575  * Parameters:
576  *              be_attrs - pointer to nvlist_t of attributes being passed in.
577  *                      The following attributes are used by this function:
578  *
579  *                      BE_ATTR_ORIG_BE_NAME            *optional
580  *                      BE_ATTR_SNAP_NAME               *optional
581  *                      BE_ATTR_NEW_BE_NAME             *optional
582  *                      BE_ATTR_NEW_BE_POOL             *optional
583  *                      BE_ATTR_NEW_BE_DESC             *optional
584  *                      BE_ATTR_ZFS_PROPERTIES          *optional
585  *                      BE_ATTR_POLICY                  *optional
586  *
587  *                      If the BE_ATTR_NEW_BE_NAME was not passed in, upon
588  *                      successful BE creation, the following attribute values
589  *                      will be returned to the caller by setting them in the
590  *                      be_attrs parameter passed in:
591  *
592  *                      BE_ATTR_SNAP_NAME
593  *                      BE_ATTR_NEW_BE_NAME
594  * Return:
```

```
 *              BE_SUCCESS - Success
 *              be_errno_t - Failure
 * Scope:
 *              Public
 */
int
be_copy(nvlist_t *be_attrs)
{
        be_transaction_data_t   bt = { 0 };
        be_fs_list_data_t       fld = { 0 };
        zfs_handle_t    *zhp = NULL;
        zpool_handle_t  *zphp = NULL;
        nvlist_t        *zfs_props = NULL;
        uuid_t          uu = { 0 };
        uuid_t          parent_uu = { 0 };
        char            obe_root_ds[MAXPATHLEN];
        char            nbe_root_ds[MAXPATHLEN];
        char            ss[MAXPATHLEN];
        char            *new_mp = NULL;
        char            *obe_name = NULL;
        boolean_t       autoname = B_FALSE;
        boolean_t       be_created = B_FALSE;
        int             i;
        int             zret;
        int             ret = BE_SUCCESS;
        struct be_defaults be_defaults;

        /* Initialize libzfs handle */
        if (!be_zfs_init())
                return (BE_ERR_INIT);

        /* Get original BE name */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_ORIG_BE_NAME, DATA_TYPE_STRING, &obe_name, NULL) != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_ORIG_BE_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }

        if ((ret = be_find_current_be(&bt)) != BE_SUCCESS) {
                return (ret);
        }

        be_get_defaults(&be_defaults);

        /* If original BE name not provided, use current BE */
        if (obe_name != NULL) {
                bt.obe_name = obe_name;
                /* Validate original BE name */
                if (!be_valid_be_name(bt.obe_name)) {
                        be_print_err(gettext("be_copy: "
                            "invalid BE name %s\n"), bt.obe_name);
                        return (BE_ERR_INVAL);
                }
        }

        if (be_defaults.be_deflt_rpool_container) {
                if ((zphp = zpool_open(g_zfs, bt.obe_zpool)) == NULL) {
                        be_print_err(gettext("be_get_node_data: failed to "
                            "open rpool (%s): %s\n"), bt.obe_zpool,
                            libzfs_error_description(g_zfs));
                        return (zfs_err_to_be_err(g_zfs));
                }
                if (be_find_zpool_callback(zphp, &bt) == 0) {
                        return (BE_ERR_BE_NOENT);
                }
```



```
        } else {
                /* Find which zpool obe_name lives in */
                if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) ==
                    0) {
                        be_print_err(gettext("be_copy: failed to "
                            "find zpool for BE (%s)\n"), bt.obe_name);
                        return (BE_ERR_BE_NOENT);
                } else if (zret < 0) {
                        be_print_err(gettext("be_copy: "
                            "zpool_iter failed: %s\n"),
                            libzfs_error_description(g_zfs));
                        return (zfs_err_to_be_err(g_zfs));
                }
        }

        /* Get snapshot name of original BE if one was provided */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_SNAP_NAME, DATA_TYPE_STRING, &bt.obe_snap_name, NULL)
            != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_SNAP_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }

        /* Get new BE name */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_NEW_BE_NAME, DATA_TYPE_STRING, &bt.nbe_name, NULL)
            != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_NEW_BE_NAME attribute\n"));
                return (BE_ERR_INVAL);
        }

        /* Get zpool name to create new BE in */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_NEW_BE_POOL, DATA_TYPE_STRING, &bt.nbe_zpool, NULL) != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_NEW_BE_POOL attribute\n"));
                return (BE_ERR_INVAL);
        }

        /* Get new BE's description if one was provided */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_NEW_BE_DESC, DATA_TYPE_STRING, &bt.nbe_desc, NULL) != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_NEW_BE_DESC attribute\n"));
                return (BE_ERR_INVAL);
        }

        /* Get BE policy to create this snapshot under */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_POLICY, DATA_TYPE_STRING, &bt.policy, NULL) != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
                    "BE_ATTR_POLICY attribute\n"));
                return (BE_ERR_INVAL);
        }

        /*
         * Create property list for new BE root dataset.  If some
         * zfs properties were already provided by the caller, dup
         * that list.  Otherwise initialize a new property list.
         */
        if (nvlist_lookup_pairs(be_attrs, NV_FLAG_NOENTOK,
            BE_ATTR_ZFS_PROPERTIES, DATA_TYPE_NVLIST, &zfs_props, NULL)
            != 0) {
                be_print_err(gettext("be_copy: failed to lookup "
```

```
727                     "BE_ATTR_ZFS_PROPERTIES attribute\n"));
728                 return (BE_ERR_INVAL);
729         }
730         if (zfs_props != NULL) {
731                 /* Make sure its a unique nvlist */
732                 if (!(zfs_props->nvl_nvflag & NV_UNIQUE_NAME) &&
733                     !(zfs_props->nvl_nvflag & NV_UNIQUE_NAME_TYPE)) {
734                         be_print_err(gettext("be_copy: ZFS property list "
735                             "not unique\n"));
736                         return (BE_ERR_INVAL);
737                 }

739                 /* Dup the list */
740                 if (nvlist_dup(zfs_props, &bt.nbe_zfs_props, 0) != 0) {
741                         be_print_err(gettext("be_copy: "
742                             "failed to dup ZFS property list\n"));
743                         return (BE_ERR_NOMEM);
744                 }
745         } else {
746                 /* Initialize new nvlist */
747                 if (nvlist_alloc(&bt.nbe_zfs_props, NV_UNIQUE_NAME, 0) != 0) {
748                         be_print_err(gettext("be_copy: internal "
749                             "error: out of memory\n"));
750                         return (BE_ERR_NOMEM);
751                 }
752         }

754         /*
755          * If new BE name provided, validate the BE name and then verify
756          * that new BE name doesn't already exist in some pool.
757          */
758         if (bt.nbe_name) {
759                 /* Validate original BE name */
760                 if (!be_valid_be_name(bt.nbe_name)) {
761                         be_print_err(gettext("be_copy: "
762                             "invalid BE name %s\n"), bt.nbe_name);
763                         ret = BE_ERR_INVAL;
764                         goto done;
765                 }

767                 /* Verify it doesn't already exist */
768                 if (getzoneid() == GLOBAL_ZONEID) {
769                         if ((zret = zpool_iter(g_zfs, be_exists_callback,
770                             bt.nbe_name)) > 0) {
771                                 be_print_err(gettext("be_copy: BE (%s) already "
772                                     "exists\n"), bt.nbe_name);
773                                 ret = BE_ERR_BE_EXISTS;
774                                 goto done;
775                         } else if (zret < 0) {
776                                 be_print_err(gettext("be_copy: zpool_iter "
777                                     "failed: %s\n"),
778                                     libzfs_error_description(g_zfs));
779                                 ret = zfs_err_to_be_err(g_zfs);
780                                 goto done;
781                         }
782                 } else {
783                         be_make_root_ds(bt.nbe_zpool, bt.nbe_name, nbe_root_ds,
784                             sizeof (nbe_root_ds));
785                         if (zfs_dataset_exists(g_zfs, nbe_root_ds,
786                             ZFS_TYPE_FILESYSTEM)) {
787                                 be_print_err(gettext("be_copy: BE (%s) already "
788                                     "exists\n"), bt.nbe_name);
789                                 ret = BE_ERR_BE_EXISTS;
790                                 goto done;
791                         }
792                 }
```



```
793         } else {
794                 /*
795                  * If an auto named BE is desired, it must be in the same
796                  * pool is the original BE.
797                  */
798                 if (bt.nbe_zpool != NULL) {
799                         be_print_err(gettext("be_copy: cannot specify pool "
800                             "name when creating an auto named BE\n"));
801                         ret = BE_ERR_INVAL;
802                         goto done;
803                 }

805                 /*
806                  * Generate auto named BE
807                  */
808                 if ((bt.nbe_name = be_auto_be_name(bt.obe_name))
809                     == NULL) {
810                         be_print_err(gettext("be_copy: "
811                             "failed to generate auto BE name\n"));
812                         ret = BE_ERR_AUTONAME;
813                         goto done;
814                 }

816                 autoname = B_TRUE;
817         }

819         /*
820          * If zpool name to create new BE in is not provided,
821          * create new BE in original BE's pool.
822          */
823         if (bt.nbe_zpool == NULL) {
824                 bt.nbe_zpool = bt.obe_zpool;
825         }

827         /* Get root dataset names for obe_name and nbe_name */
828         be_make_root_ds(bt.obe_zpool, bt.obe_name, obe_root_ds,
829             sizeof (obe_root_ds));
830         be_make_root_ds(bt.nbe_zpool, bt.nbe_name, nbe_root_ds,
831             sizeof (nbe_root_ds));

833         bt.obe_root_ds = obe_root_ds;
834         bt.nbe_root_ds = nbe_root_ds;

836         /*
837          * If an existing snapshot name has been provided to create from,
838          * verify that it exists for the original BE's root dataset.
839          */
840         if (bt.obe_snap_name != NULL) {

842                 /* Generate dataset name for snapshot to use. */
843                 (void) snprintf(ss, sizeof (ss), "%s@%s", bt.obe_root_ds,
844                     bt.obe_snap_name);

846                 /* Verify snapshot exists */
847                 if (!zfs_dataset_exists(g_zfs, ss, ZFS_TYPE_SNAPSHOT)) {
848                         be_print_err(gettext("be_copy: "
849                             "snapshot does not exist (%s): %s\n"), ss,
850                             libzfs_error_description(g_zfs));
851                         ret = BE_ERR_SS_NOENT;
852                         goto done;
853                 }
854         } else {
855                 /*
856                  * Else snapshot name was not provided, generate an
857                  * auto named snapshot to use as its origin.
858                  */
```

```
859                 if ((ret = _be_create_snapshot(bt.obe_name,
860                     &bt.obe_snap_name, bt.policy)) != BE_SUCCESS) {
861                         be_print_err(gettext("be_copy: "
862                             "failed to create auto named snapshot\n"));
863                         goto done;
864                 }

866                 if (nvlist_add_string(be_attrs, BE_ATTR_SNAP_NAME,
867                     bt.obe_snap_name) != 0) {
868                         be_print_err(gettext("be_copy: "
869                             "failed to add snap name to be_attrs\n"));
870                         ret = BE_ERR_NOMEM;
871                         goto done;
872                 }
873         }

875         /* Get handle to original BE's root dataset. */
876         if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_FILESYSTEM))
877             == NULL) {
878                 be_print_err(gettext("be_copy: failed to "
879                     "open BE root dataset (%s): %s\n"), bt.obe_root_ds,
880                     libzfs_error_description(g_zfs));
881                 ret = zfs_err_to_be_err(g_zfs);
882                 goto done;
883         }

885         /* If original BE is currently mounted, record its altroot. */
886         if (zfs_is_mounted(zhp, &bt.obe_altroot) && bt.obe_altroot == NULL) {
887                 be_print_err(gettext("be_copy: failed to "
888                     "get altroot of mounted BE %s: %s\n"),
889                     bt.obe_name, libzfs_error_description(g_zfs));
890                 ret = zfs_err_to_be_err(g_zfs);
891                 goto done;
892         }

894         if (strcmp(bt.obe_zpool, bt.nbe_zpool) == 0) {

896                 /* Do clone */

898                 /*
899                  * Iterate through original BE's datasets and clone
900                  * them to create new BE.  This call will end up closing
901                  * the zfs handle passed in whether it succeeds for fails.
902                  */
903                 if ((ret = be_clone_fs_callback(zhp, &bt)) != 0) {
904                         zhp = NULL;
905                         /* Creating clone BE failed */
906                         if (!autoname || ret != BE_ERR_BE_EXISTS) {
907                                 be_print_err(gettext("be_copy: "
908                                     "failed to clone new BE (%s) from "
909                                     "orig BE (%s)\n"),
910                                     bt.nbe_name, bt.obe_name);
911                                 ret = BE_ERR_CLONE;
912                                 goto done;
913                         }

915                         /*
916                          * We failed to create the new BE because a BE with
917                          * the auto-name we generated above has since come
918                          * into existence.  Regenerate a new auto-name
919                          * and retry.
920                          */
921                         for (i = 1; i < BE_AUTO_NAME_MAX_TRY; i++) {

923                                 /* Sleep 1 before retrying */
924                                 (void) sleep(1);
```



```
926                                 /* Generate new auto BE name */
927                                 free(bt.nbe_name);
928                                 if ((bt.nbe_name = be_auto_be_name(bt.obe_name))
929                                     == NULL) {
930                                         be_print_err(gettext("be_copy: "
931                                             "failed to generate auto "
932                                             "BE name\n"));
933                                         ret = BE_ERR_AUTONAME;
934                                         goto done;
935                                 }

937                                 /*
938                                  * Regenerate string for new BE's
939                                  * root dataset name
940                                  */
941                                 be_make_root_ds(bt.nbe_zpool, bt.nbe_name,
942                                     nbe_root_ds, sizeof (nbe_root_ds));
943                                 bt.nbe_root_ds = nbe_root_ds;

945                                 /*
946                                  * Get handle to original BE's root dataset.
947                                  */
948                                 if ((zhp = zfs_open(g_zfs, bt.obe_root_ds,
949                                     ZFS_TYPE_FILESYSTEM)) == NULL) {
950                                         be_print_err(gettext("be_copy: "
951                                             "failed to open BE root dataset "
952                                             "(%s): %s\n"), bt.obe_root_ds,
953                                             libzfs_error_description(g_zfs));
954                                         ret = zfs_err_to_be_err(g_zfs);
955                                         goto done;
956                                 }

958                                 /*
959                                  * Try to clone the BE again.  This
960                                  * call will end up closing the zfs
961                                  * handle passed in whether it
962                                  * succeeds or fails.
963                                  */
964                                 ret = be_clone_fs_callback(zhp, &bt);
965                                 zhp = NULL;
966                                 if (ret == 0) {
967                                         break;
968                                 } else if (ret != BE_ERR_BE_EXISTS) {
969                                         be_print_err(gettext("be_copy: "
970                                             "failed to clone new BE "
971                                             "(%s) from orig BE (%s)\n"),
972                                             bt.nbe_name, bt.obe_name);
973                                         ret = BE_ERR_CLONE;
974                                         goto done;
975                                 }
976                         }

978                         /*
979                          * If we've exhausted the maximum number of
980                          * tries, free the auto BE name and return
981                          * error.
982                          */
983                         if (i == BE_AUTO_NAME_MAX_TRY) {
984                                 be_print_err(gettext("be_copy: failed "
985                                     "to create unique auto BE name\n"));
986                                 free(bt.nbe_name);
987                                 bt.nbe_name = NULL;
988                                 ret = BE_ERR_AUTONAME;
989                                 goto done;
990                         }
```

```
 991                 }
 992                 zhp = NULL;

 994         } else {

 996                 /* Do copy (i.e. send BE datasets via zfs_send/recv) */

 998                 /*
 999                  * Verify BE container dataset in nbe_zpool exists.
1000                  * If not, create it.
1001                  */
1002                 if (!be_create_container_ds(bt.nbe_zpool)) {
1003                         ret = BE_ERR_CREATDS;
1004                         goto done;
1005                 }

1007                 /*
1008                  * Iterate through original BE's datasets and send
1009                  * them to the other pool.  This call will end up closing
1010                  * the zfs handle passed in whether it succeeds or fails.
1011                  */
1012                 if ((ret = be_send_fs_callback(zhp, &bt)) != 0) {
1013                         be_print_err(gettext("be_copy: failed to "
1014                             "send BE (%s) to pool (%s)\n"), bt.obe_name,
1015                             bt.nbe_zpool);
1016                         ret = BE_ERR_COPY;
1017                         zhp = NULL;
1018                         goto done;
1019                 }
1020                 zhp = NULL;
1021         }

1023         /*
1024          * Set flag to note that the dataset(s) for the new BE have been
1025          * successfully created so that if a failure happens from this point
1026          * on, we know to cleanup these datasets.
1027          */
1028         be_created = B_TRUE;

1030         /*
1031          * Validate that the new BE is mountable.
1032          * Do not attempt to mount non-global zone datasets
1033          * since they are not cloned yet.
1034          */
1035         if ((ret = _be_mount(bt.nbe_name, &new_mp, BE_MOUNT_FLAG_NO_ZONES))
1036             != BE_SUCCESS) {
1037                 be_print_err(gettext("be_copy: failed to "
1038                     "mount newly created BE\n"));
1039                 (void) _be_unmount(bt.nbe_name, 0);
1040                 goto done;
1041         }

1043         /* Set UUID for new BE */
1044         if (getzoneid() == GLOBAL_ZONEID) {
1045                 if (be_set_uuid(bt.nbe_root_ds) != BE_SUCCESS) {
1046                         be_print_err(gettext("be_copy: failed to "
1047                             "set uuid for new BE\n"));
1048                 }
1049         } else {
1050                 if ((ret = be_zone_get_parent_uuid(bt.obe_root_ds,
1051                     &parent_uu)) != BE_SUCCESS) {
1052                         be_print_err(gettext("be_copy: failed to get "
1053                             "parentbe uuid from orig BE\n"));
1054                         ret = BE_ERR_ZONE_NO_PARENTBE;
1055                         goto done;
1056                 } else if ((ret = be_zone_set_parent_uuid(bt.nbe_root_ds,
```



```
1057                     parent_uu)) != BE_SUCCESS) {
1058                         be_print_err(gettext("be_copy: failed to set "
1059                             "parentbe uuid for newly created BE\n"));
1060                         goto done;
1061                 }
1062         }

1064         /*
1065          * Process zones outside of the private BE namespace.
1066          * This has to be done here because we need the uuid set in the
1067          * root dataset of the new BE. The uuid is use to set the parentbe
1068          * property for the new zones datasets.
1069          */
1070         if (getzoneid() == GLOBAL_ZONEID &&
1071             be_get_uuid(bt.obe_root_ds, &uu) == BE_SUCCESS) {
1072                 if ((ret = be_copy_zones(bt.obe_name, bt.obe_root_ds,
1073                     bt.nbe_root_ds)) != BE_SUCCESS) {
1074                         be_print_err(gettext("be_copy: failed to process "
1075                             "zones\n"));
1076                         goto done;
1077                 }
1078         }

1080         /*
1081          * Generate a list of file systems from the original BE that are
1082          * legacy mounted.  We use this list to determine which entries in
1083          * vfstab we need to update for the new BE we've just created.
1084          */
1085         if ((ret = be_get_legacy_fs(bt.obe_name, bt.obe_root_ds, NULL, NULL,
1086             &fld)) != BE_SUCCESS) {
1087                 be_print_err(gettext("be_copy: failed to "
1088                     "get legacy mounted file system list for %s\n"),
1089                     bt.obe_name);
1090                 goto done;
1091         }

1093         /*
1094          * Update new BE's vfstab.
1095          */
1096         if ((ret = be_update_vfstab(bt.nbe_name, bt.obe_zpool, bt.nbe_zpool,
1097             &fld, new_mp)) != BE_SUCCESS) {
1098                 be_print_err(gettext("be_copy: failed to "
1099                     "update new BE's vfstab (%s)\n"), bt.nbe_name);
1100                 goto done;
1101         }

1103         /* Unmount the new BE */
1104         if ((ret = _be_unmount(bt.nbe_name, 0)) != BE_SUCCESS) {
1105                 be_print_err(gettext("be_copy: failed to "
1106                     "unmount newly created BE\n"));
1107                 goto done;
1108         }

1110         /*
1111          * Add boot menu entry for newly created clone
1112          */
1113         if (getzoneid() == GLOBAL_ZONEID &&
1114             (ret = be_append_menu(bt.nbe_name, bt.nbe_zpool,
1115             NULL, bt.obe_root_ds, bt.nbe_desc)) != BE_SUCCESS) {
1116                 be_print_err(gettext("be_copy: failed to "
1117                     "add BE (%s) to boot menu\n"), bt.nbe_name);
1118                 goto done;
1119         }

1121         /*
1122          * If we succeeded in creating an auto named BE, set its policy
```

```
1123          * type and return the auto generated name to the caller by storing
1124          * it in the nvlist passed in by the caller.
1125          */
1126         if (autoname) {
1127                 /* Get handle to new BE's root dataset. */
1128                 if ((zhp = zfs_open(g_zfs, bt.nbe_root_ds,
1129                     ZFS_TYPE_FILESYSTEM)) == NULL) {
1130                         be_print_err(gettext("be_copy: failed to "
1131                             "open BE root dataset (%s): %s\n"), bt.nbe_root_ds,
1132                             libzfs_error_description(g_zfs));
1133                         ret = zfs_err_to_be_err(g_zfs);
1134                         goto done;
1135                 }

1137                 /*
1138                  * Set the policy type property into the new BE's root dataset
1139                  */
1140                 if (bt.policy == NULL) {
1141                         /* If no policy type provided, use default type */
1142                         bt.policy = be_default_policy();
1143                 }

1145                 if (zfs_prop_set(zhp, BE_POLICY_PROPERTY, bt.policy) != 0) {
1146                         be_print_err(gettext("be_copy: failed to "
1147                             "set BE policy for %s: %s\n"), bt.nbe_name,
1148                             libzfs_error_description(g_zfs));
1149                         ret = zfs_err_to_be_err(g_zfs);
1150                         goto done;
1151                 }

1153                 /*
1154                  * Return the auto generated name to the caller
1155                  */
1156                 if (bt.nbe_name) {
1157                         if (nvlist_add_string(be_attrs, BE_ATTR_NEW_BE_NAME,
1158                             bt.nbe_name) != 0) {
1159                                 be_print_err(gettext("be_copy: failed to "
1160                                     "add snap name to be_attrs\n"));
1161                         }
1162                 }
1163         }

1165 done:
1166         ZFS_CLOSE(zhp);
1167         be_free_fs_list(&fld);

1170         if (bt.nbe_zfs_props != NULL)
1169         nvlist_free(bt.nbe_zfs_props);

1171         free(bt.obe_altroot);
1172         free(new_mp);

1174         /*
1175          * If a failure occurred and we already created the datasets for
1176          * the new boot environment, destroy them.
1177          */
1178         if (ret != BE_SUCCESS && be_created) {
1179                 be_destroy_data_t       cdd = { 0 };

1181                 cdd.force_unmount = B_TRUE;

1183                 be_print_err(gettext("be_copy: "
1184                     "destroying partially created boot environment\n"));

1186                 if (getzoneid() == GLOBAL_ZONEID && be_get_uuid(bt.nbe_root_ds,
1187                     &cdd.gz_be_uuid) == 0)
```



```
1188                         (void) be_destroy_zones(bt.nbe_name, bt.nbe_root_ds,
1189                             &cdd);

1191                 (void) _be_destroy(bt.nbe_root_ds, &cdd);
1192         }

1194         be_zfs_fini();

1196         return (ret);
1197 }
```

_____***unchanged_portion_omitted_***

```
1872 /*
1873  * Function:    be_copy_zones
1874  * Description: Find valid zones and clone them to create their
1875  *              corresponding datasets for the BE being created.
1876  * Parameters:
1877  *              obe_name - name of source global BE being copied.
1878  *              obe_root_ds - root dataset of source global BE being copied.
1879  *              nbe_root_ds - root dataset of target global BE.
1880  * Return:
1881  *              BE_SUCCESS - Success
1882  *              be_errno_t - Failure
1883  * Scope:
1884  *              Private
1885  */
1886 static int
1887 be_copy_zones(char *obe_name, char *obe_root_ds, char *nbe_root_ds)
1888 {
1889         int             i, num_retries;
1890         int             ret = BE_SUCCESS;
1891         int             iret = 0;
1892         char            *zonename = NULL;
1893         char            *zonepath = NULL;
1894         char            *zone_be_name = NULL;
1895         char            *temp_mntpt = NULL;
1896         char            *new_zone_be_name = NULL;
1897         char            zoneroot[MAXPATHLEN];
1898         char            zoneroot_ds[MAXPATHLEN];
1899         char            zone_container_ds[MAXPATHLEN];
1900         char            new_zoneroot_ds[MAXPATHLEN];
1901         char            ss[MAXPATHLEN];
1902         uuid_t          uu = { 0 };
1903         char            uu_string[UUID_PRINTABLE_STRING_LENGTH] = { 0 };
1904         be_transaction_data_t bt = { 0 };
1905         zfs_handle_t    *obe_zhp = NULL;
1906         zfs_handle_t    *nbe_zhp = NULL;
1907         zfs_handle_t    *z_zhp = NULL;
1908         zoneList_t      zlist = NULL;
1909         zoneBrandList_t *brands = NULL;
1910         boolean_t       mounted_here = B_FALSE;
1911         char            *snap_name = NULL;

1913         /* If zones are not implemented, then get out. */
1914         if (!z_zones_are_implemented()) {
1915                 return (BE_SUCCESS);
1916         }

1918         /* Get list of supported brands */
1919         if ((brands = be_get_supported_brandlist()) == NULL) {
1920                 be_print_err(gettext("be_copy_zones: "
1921                     "no supported brands\n"));
1922                 return (BE_SUCCESS);
1923         }

1925         /* Get handle to origin BE's root dataset */
```

```
1926         if ((obe_zhp = zfs_open(g_zfs, obe_root_ds, ZFS_TYPE_FILESYSTEM))
1927             == NULL) {
1928                 be_print_err(gettext("be_copy_zones: failed to open "
1929                     "the origin BE root dataset (%s) for zones processing: "
1930                     "%s\n"), obe_root_ds, libzfs_error_description(g_zfs));
1931                 return (zfs_err_to_be_err(g_zfs));
1932         }

1934         /* Get handle to newly cloned BE's root dataset */
1935         if ((nbe_zhp = zfs_open(g_zfs, nbe_root_ds, ZFS_TYPE_FILESYSTEM))
1936             == NULL) {
1937                 be_print_err(gettext("be_copy_zones: failed to open "
1938                     "the new BE root dataset (%s): %s\n"), nbe_root_ds,
1939                     libzfs_error_description(g_zfs));
1940                 ZFS_CLOSE(obe_zhp);
1941                 return (zfs_err_to_be_err(g_zfs));
1942         }

1944         /* Get the uuid of the newly cloned parent BE. */
1945         if (be_get_uuid(zfs_get_name(nbe_zhp), &uu) != BE_SUCCESS) {
1946                 be_print_err(gettext("be_copy_zones: "
1947                     "failed to get uuid for BE root "
1948                     "dataset %s\n"), zfs_get_name(nbe_zhp));
1949                 ZFS_CLOSE(nbe_zhp);
1950                 goto done;
1951         }
1952         ZFS_CLOSE(nbe_zhp);
1953         uuid_unparse(uu, uu_string);

1955         /*
1956          * If the origin BE is not mounted, we must mount it here to
1957          * gather data about the non-global zones in it.
1958          */
1959         if (!zfs_is_mounted(obe_zhp, &temp_mntpt)) {
1960                 if ((ret = _be_mount(obe_name, &temp_mntpt,
1961                     BE_MOUNT_FLAG_NULL)) != BE_SUCCESS) {
1962                         be_print_err(gettext("be_copy_zones: failed to "
1963                             "mount the BE (%s) for zones procesing.\n"),
1964                             obe_name);
1965                         goto done;
1966                 }
1967                 mounted_here = B_TRUE;
1968         }

1970         z_set_zone_root(temp_mntpt);

1972         /* Get list of supported zones. */
1973         if ((zlist = z_get_nonglobal_zone_list_by_brand(brands)) == NULL) {
1974                 ret = BE_SUCCESS;
1975                 goto done;
1976         }

1978         for (i = 0; (zonename = z_zlist_get_zonename(zlist, i)) != NULL; i++) {

1980                 be_fs_list_data_t       fld = { 0 };
1981                 char                    zonepath_ds[MAXPATHLEN];
1982                 char                    *ds = NULL;

1984                 /* Get zonepath of zone */
1985                 zonepath = z_zlist_get_zonepath(zlist, i);

1987                 /* Skip zones that aren't at least installed */
1988                 if (z_zlist_get_current_state(zlist, i) < ZONE_STATE_INSTALLED)
1989                         continue;

1991                 /*
```



```
1992                  * Get the dataset of this zonepath.  If its not
1993                  * a dataset, skip it.
1994                  */
1995                 if ((ds = be_get_ds_from_dir(zonepath)) == NULL)
1996                         continue;

1998                 (void) strlcpy(zonepath_ds, ds, sizeof (zonepath_ds));
1999                 free(ds);
2000                 ds = NULL;

2002                 /* Get zoneroot directory */
2003                 be_make_zoneroot(zonepath, zoneroot, sizeof (zoneroot));

2005                 /* If zonepath dataset not supported, skip it. */
2006                 if (!be_zone_supported(zonepath_ds)) {
2007                         continue;
2008                 }

2010                 if ((ret = be_find_active_zone_root(obe_zhp, zonepath_ds,
2011                     zoneroot_ds, sizeof (zoneroot_ds))) != BE_SUCCESS) {
2012                         be_print_err(gettext("be_copy_zones: "
2013                             "failed to find active zone root for zone %s "
2014                             "in BE %s\n"), zonename, obe_name);
2015                         goto done;
2016                 }

2018                 be_make_container_ds(zonepath_ds, zone_container_ds,
2019                     sizeof (zone_container_ds));

2021                 if ((z_zhp = zfs_open(g_zfs, zoneroot_ds,
2022                     ZFS_TYPE_FILESYSTEM)) == NULL) {
2023                         be_print_err(gettext("be_copy_zones: "
2024                             "failed to open zone root dataset (%s): %s\n"),
2025                             zoneroot_ds, libzfs_error_description(g_zfs));
2026                         ret = zfs_err_to_be_err(g_zfs);
2027                         goto done;
2028                 }

2030                 zone_be_name =
2031                     be_get_zone_be_name(zoneroot_ds, zone_container_ds);

2033                 if ((new_zone_be_name = be_auto_zone_be_name(zone_container_ds,
2034                     zone_be_name)) == NULL) {
2035                         be_print_err(gettext("be_copy_zones: failed "
2036                             "to generate auto name for zone BE.\n"));
2037                         ret = BE_ERR_AUTONAME;
2038                         goto done;
2039                 }

2041                 if ((snap_name = be_auto_snap_name()) == NULL) {
2042                         be_print_err(gettext("be_copy_zones: failed to "
2043                             "generate snapshot name for zone BE.\n"));
2044                         ret = BE_ERR_AUTONAME;
2045                         goto done;
2046                 }

2048                 (void) snprintf(ss, sizeof (ss), "%s@%s", zoneroot_ds,
2049                     snap_name);

2051                 if (zfs_snapshot(g_zfs, ss, B_TRUE, NULL) != 0) {
2052                         be_print_err(gettext("be_copy_zones: "
2053                             "failed to snapshot zone BE (%s): %s\n"),
2054                             ss, libzfs_error_description(g_zfs));
2055                         if (libzfs_errno(g_zfs) == EZFS_EXISTS)
2056                                 ret = BE_ERR_ZONE_SS_EXISTS;
2057                         else
```

```
2058                                    ret = zfs_err_to_be_err(g_zfs);
2059
2060                            goto done;
2061                    }
2062
2063                    (void) snprintf(new_zoneroot_ds, sizeof (new_zoneroot_ds),
2064                        "%s/%s", zone_container_ds, new_zone_be_name);
2065
2066                    bt.obe_name = zone_be_name;
2067                    bt.obe_root_ds = zoneroot_ds;
2068                    bt.obe_snap_name = snap_name;
2069                    bt.obe_altroot = temp_mntpt;
2070                    bt.nbe_name = new_zone_be_name;
2071                    bt.nbe_root_ds = new_zoneroot_ds;
2072
2073                    if (nvlist_alloc(&bt.nbe_zfs_props, NV_UNIQUE_NAME, 0) != 0) {
2074                            be_print_err(gettext("be_copy_zones: "
2075                                "internal error: out of memory\n"));
2076                            ret = BE_ERR_NOMEM;
2077                            goto done;
2078                    }
2079
2080                    /*
2081                     * The call to be_clone_fs_callback always closes the
2082                     * zfs_handle so there's no need to close z_zhp.
2083                     */
2084                    if ((iret = be_clone_fs_callback(z_zhp, &bt)) != 0) {
2085                            z_zhp = NULL;
2086                            if (iret != BE_ERR_BE_EXISTS) {
2087                                    be_print_err(gettext("be_copy_zones: "
2088                                        "failed to create zone BE clone for new "
2089                                        "zone BE %s\n"), new_zone_be_name);
2090                                    ret = iret;
2091                                    if (bt.nbe_zfs_props != NULL)
2092                                    nvlist_free(bt.nbe_zfs_props);
2093                                    goto done;
2094                            }
2095                            /*
2096                             * We failed to create the new zone BE because a zone
2097                             * BE with the auto-name we generated above has since
2098                             * come into existence. Regenerate a new auto-name
2099                             * and retry.
2100                             */
2101                            for (num_retries = 1;
2102                                num_retries < BE_AUTO_NAME_MAX_TRY;
2103                                num_retries++) {
2104
2105                                    /* Sleep 1 before retrying */
2106                                    (void) sleep(1);
2107
2108                                    /* Generate new auto zone BE name */
2109                                    free(new_zone_be_name);
2110                                    if ((new_zone_be_name = be_auto_zone_be_name(
2111                                        zone_container_ds,
2112                                        zone_be_name)) == NULL) {
2113                                            be_print_err(gettext("be_copy_zones: "
2114                                                "failed to generate auto name "
2115                                                "for zone BE.\n"));
2116                                            ret = BE_ERR_AUTONAME;
2117                                            if (bt.nbe_zfs_props != NULL)
2118                                            nvlist_free(bt.nbe_zfs_props);
2119                                            goto done;
2120                                    }
2121
2122                                    (void) snprintf(new_zoneroot_ds,
2123                                        sizeof (new_zoneroot_ds),
```



```
2124                                        "%s/%s", zone_container_ds,
2125                                        new_zone_be_name);
2126                                    bt.nbe_name = new_zone_be_name;
2127                                    bt.nbe_root_ds = new_zoneroot_ds;
2128
2129                                    /*
2130                                     * Get handle to original zone BE's root
2131                                     * dataset.
2132                                     */
2133                                    if ((z_zhp = zfs_open(g_zfs, zoneroot_ds,
2134                                        ZFS_TYPE_FILESYSTEM)) == NULL) {
2135                                            be_print_err(gettext("be_copy_zones: "
2136                                                "failed to open zone root "
2137                                                "dataset (%s): %s\n"),
2138                                                zoneroot_ds,
2139                                                libzfs_error_description(g_zfs));
2140                                            ret = zfs_err_to_be_err(g_zfs);
2141                                            if (bt.nbe_zfs_props != NULL)
2142                                            nvlist_free(bt.nbe_zfs_props);
2143                                            goto done;
2144                                    }
2145
2146                                    /*
2147                                     * Try to clone the zone BE again. This
2148                                     * call will end up closing the zfs
2149                                     * handle passed in whether it
2150                                     * succeeds or fails.
2151                                     */
2152                                    iret = be_clone_fs_callback(z_zhp, &bt);
2153                                    z_zhp = NULL;
2154                                    if (iret == 0) {
2155                                            break;
2156                                    } else if (iret != BE_ERR_BE_EXISTS) {
2157                                            be_print_err(gettext("be_copy_zones: "
2158                                                "failed to create zone BE clone "
2159                                                "for new zone BE %s\n"),
2160                                                new_zone_be_name);
2161                                            ret = iret;
2162                                            if (bt.nbe_zfs_props != NULL)
2163                                            nvlist_free(bt.nbe_zfs_props);
2164                                            goto done;
2165                                    }
2166                            }
2167                            /*
2168                             * If we've exhausted the maximum number of
2169                             * tries, free the auto zone BE name and return
2170                             * error.
2171                             */
2172                            if (num_retries == BE_AUTO_NAME_MAX_TRY) {
2173                                    be_print_err(gettext("be_copy_zones: failed "
2174                                        "to create a unique auto zone BE name\n"));
2175                                    free(bt.nbe_name);
2176                                    bt.nbe_name = NULL;
2177                                    ret = BE_ERR_AUTONAME;
2178                                    if (bt.nbe_zfs_props != NULL)
2179                                    nvlist_free(bt.nbe_zfs_props);
2180                                    goto done;
2181                            }
2182                    }
2183
2184                    if (bt.nbe_zfs_props != NULL)
2185                    nvlist_free(bt.nbe_zfs_props);
2186
2187                    z_zhp = NULL;
2188
2189                    if ((z_zhp = zfs_open(g_zfs, new_zoneroot_ds,
```

```
2184                    ZFS_TYPE_FILESYSTEM)) == NULL) {
2185                        be_print_err(gettext("be_copy_zones: "
2186                            "failed to open the new zone BE root dataset "
2187                            "(%s): %s\n"), new_zoneroot_ds,
2188                            libzfs_error_description(g_zfs));
2189                        ret = zfs_err_to_be_err(g_zfs);
2190                        goto done;
2191                }

2193                if (zfs_prop_set(z_zhp, BE_ZONE_PARENTBE_PROPERTY,
2194                    uu_string) != 0) {
2195                        be_print_err(gettext("be_copy_zones: "
2196                            "failed to set parentbe property\n"));
2197                        ZFS_CLOSE(z_zhp);
2198                        ret = zfs_err_to_be_err(g_zfs);
2199                        goto done;
2200                }

2202                if (zfs_prop_set(z_zhp, BE_ZONE_ACTIVE_PROPERTY, "on") != 0) {
2203                        be_print_err(gettext("be_copy_zones: "
2204                            "failed to set active property\n"));
2205                        ZFS_CLOSE(z_zhp);
2206                        ret = zfs_err_to_be_err(g_zfs);
2207                        goto done;
2208                }

2210                /*
2211                 * Generate a list of file systems from the original
2212                 * zone BE that are legacy mounted.  We use this list
2213                 * to determine which entries in the vfstab we need to
2214                 * update for the new zone BE we've just created.
2215                 */
2216                if ((ret = be_get_legacy_fs(obe_name, obe_root_ds,
2217                    zoneroot_ds, zoneroot, &fld)) != BE_SUCCESS) {
2218                        be_print_err(gettext("be_copy_zones: "
2219                            "failed to get legacy mounted file system "
2220                            "list for zone %s\n"), zonename);
2221                        ZFS_CLOSE(z_zhp);
2222                        goto done;
2223                }

2225                /*
2226                 * Update new zone BE's vfstab.
2227                 */
2228                if ((ret = be_update_zone_vfstab(z_zhp, bt.nbe_name,
2229                    zonepath_ds, zonepath_ds, &fld)) != BE_SUCCESS) {
2230                        be_print_err(gettext("be_copy_zones: "
2231                            "failed to update new BE's vfstab (%s)\n"),
2232                            bt.nbe_name);
2233                        ZFS_CLOSE(z_zhp);
2234                        be_free_fs_list(&fld);
2235                        goto done;
2236                }

2238                be_free_fs_list(&fld);
2239                ZFS_CLOSE(z_zhp);
2240        }

2242 done:
2243        free(snap_name);
2244        if (brands != NULL)
2245                z_free_brand_list(brands);
2246        if (zlist != NULL)
2247                z_free_zone_list(zlist);

2249        if (mounted_here)
```



```
2250                (void) _be_unmount(obe_name, 0);

2252        ZFS_CLOSE(obe_zhp);
2253        return (ret);
2254 }
```

_____***unchanged_portion_omitted_***

****
23511 Mon Feb 15 12:56:08 2016
new/usr/src/lib/libbe/common/be_snapshot.c
patch tsoome-feedback
****
_____unchanged_portion_omitted_

```
379 /* ************************************************************** */
380 /*                      Semi-Private Functions                    */
381 /* ************************************************************** */

383 /*
384  * Function:    _be_create_snapshot
385  * Description: see be_create_snapshot
386  * Parameters:
387  *              be_name - The name of the BE that we're taking a snapshot of.
388  *              snap_name - The name of the snapshot we're creating. If
389  *                      snap_name is NULL an auto generated name will be used,
390  *                      and upon success, will return that name via this
391  *                      reference pointer.  The caller is responsible for
392  *                      freeing the returned name.
393  *              policy - The clean-up policy type. (library wide use only)
394  * Return:
395  *              BE_SUCCESS - Success
396  *              be_errno_t - Failure
397  * Scope:
398  *              Semi-private (library wide use only)
399  */
400 int
401 _be_create_snapshot(char *be_name, char **snap_name, char *policy)
402 {
403         be_transaction_data_t   bt = { 0 };
404         zfs_handle_t            *zhp = NULL;
405         nvlist_t                *ss_props = NULL;
406         char                    ss[MAXPATHLEN];
407         char                    root_ds[MAXPATHLEN];
408         int                     pool_version = 0;
409         int                     i = 0;
410         int                     zret = 0, ret = BE_SUCCESS;
411         boolean_t               autoname = B_FALSE;

413         /* Set parameters in bt structure */
414         bt.obe_name = be_name;
415         bt.obe_snap_name = *snap_name;
416         bt.policy = policy;

418         /* If original BE name not supplied, use current BE */
419         if (bt.obe_name == NULL) {
420                 if ((ret = be_find_current_be(&bt)) != BE_SUCCESS) {
421                         return (ret);
422                 }
423         }

425         /* Find which zpool obe_name lives in */
426         if ((zret = zpool_iter(g_zfs, be_find_zpool_callback, &bt)) == 0) {
427                 be_print_err(gettext("be_create_snapshot: failed to "
428                     "find zpool for BE (%s)\n"), bt.obe_name);
429                 return (BE_ERR_BE_NOENT);
430         } else if (zret < 0) {
431                 be_print_err(gettext("be_create_snapshot: "
432                     "zpool_iter failed: %s\n"),
433                     libzfs_error_description(g_zfs));
434                 return (zfs_err_to_be_err(g_zfs));
435         }
```



```
437         be_make_root_ds(bt.obe_zpool, bt.obe_name, root_ds,
438             sizeof (root_ds));
439         bt.obe_root_ds = root_ds;

441         if (getzoneid() != GLOBAL_ZONEID) {
442                 if (!be_zone_compare_uuids(bt.obe_root_ds)) {
443                         be_print_err(gettext("be_create_snapshot: creating "
444                             "snapshot for the zone root dataset from "
445                             "non-active global BE is not "
446                             "supported\n"));
447                         return (BE_ERR_NOTSUP);
448                 }
449         }

451         /* If BE policy not specified, use the default policy */
452         if (bt.policy == NULL) {
453                 bt.policy = be_default_policy();
454         } else {
455                 /* Validate policy type */
456                 if (!valid_be_policy(bt.policy)) {
457                         be_print_err(gettext("be_create_snapshot: "
458                             "invalid BE policy type (%s)\n"), bt.policy);
459                         return (BE_ERR_INVAL);
460                 }
461         }

463         /*
464          * If snapshot name not specified, set auto name flag and
465          * generate auto snapshot name.
466          */
467         if (bt.obe_snap_name == NULL) {
468                 autoname = B_TRUE;
469                 if ((bt.obe_snap_name = be_auto_snap_name())
470                     == NULL) {
471                         be_print_err(gettext("be_create_snapshot: "
472                             "failed to create auto snapshot name\n"));
473                         ret =  BE_ERR_AUTONAME;
474                         goto done;
475                 }
476         }

478         /* Generate the name of the snapshot to take. */
479         (void) snprintf(ss, sizeof (ss), "%s@%s", bt.obe_root_ds,
480             bt.obe_snap_name);

482         /* Get handle to BE's root dataset */
483         if ((zhp = zfs_open(g_zfs, bt.obe_root_ds, ZFS_TYPE_DATASET))
484             == NULL) {
485                 be_print_err(gettext("be_create_snapshot: "
486                     "failed to open BE root dataset (%s): %s\n"),
487                     bt.obe_root_ds, libzfs_error_description(g_zfs));
488                 ret = zfs_err_to_be_err(g_zfs);
489                 goto done;
490         }

492         /* Get the ZFS pool version of the pool where this dataset resides */
493         if (zfs_spa_version(zhp, &pool_version) != 0) {
494                 be_print_err(gettext("be_create_snapshot: failed to "
495                     "get ZFS pool version for %s: %s\n"), zfs_get_name(zhp),
496                     libzfs_error_description(g_zfs));
497         }

499         /*
500          * If ZFS pool version supports snapshot user properties, store
501          * cleanup policy there.  Otherwise don't set one - this snapshot
502          * will always inherit the cleanup policy from its parent.
```

```
503          */
504         if (getzoneid() == GLOBAL_ZONEID) {
505                 if (pool_version >= SPA_VERSION_SNAP_PROPS) {
506                         if (nvlist_alloc(&ss_props, NV_UNIQUE_NAME, 0) != 0) {
507                                 be_print_err(gettext("be_create_snapshot: "
508                                     "internal error: out of memory\n"));
509                                 return (BE_ERR_NOMEM);
510                         }
511                         if (nvlist_add_string(ss_props, BE_POLICY_PROPERTY,
512                             bt.policy) != 0) {
513                                 be_print_err(gettext("be_create_snapshot: "
514                                     "internal error: out of memory\n"));
515                                 nvlist_free(ss_props);
516                                 return (BE_ERR_NOMEM);
517                         }
518                 } else if (policy != NULL) {
519                         /*
520                          * If an explicit cleanup policy was requested
521                          * by the caller and we don't support it, error out.
522                          */
523                         be_print_err(gettext("be_create_snapshot: cannot set "
524                             "cleanup policy: ZFS pool version is %d\n"),
525                             pool_version);
526                         return (BE_ERR_NOTSUP);
527                 }
528         }

530         /* Create the snapshots recursively */
531         if (zfs_snapshot(g_zfs, ss, B_TRUE, ss_props) != 0) {
532                 if (!autoname || libzfs_errno(g_zfs) != EZFS_EXISTS) {
533                         be_print_err(gettext("be_create_snapshot: "
534                             "recursive snapshot of %s failed: %s\n"),
535                             ss, libzfs_error_description(g_zfs));

537                         if (libzfs_errno(g_zfs) == EZFS_EXISTS)
538                                 ret = BE_ERR_SS_EXISTS;
539                         else
540                                 ret = zfs_err_to_be_err(g_zfs);

542                         goto done;
543                 } else {
544                         for (i = 1; i < BE_AUTO_NAME_MAX_TRY; i++) {

546                                 /* Sleep 1 before retrying */
547                                 (void) sleep(1);

549                                 /* Generate new auto snapshot name. */
550                                 free(bt.obe_snap_name);
551                                 if ((bt.obe_snap_name =
552                                     be_auto_snap_name()) == NULL) {
553                                         be_print_err(gettext(
554                                             "be_create_snapshot: failed to "
555                                             "create auto snapshot name\n"));
556                                         ret = BE_ERR_AUTONAME;
557                                         goto done;
558                                 }

560                                 /* Generate string of the snapshot to take. */
561                                 (void) snprintf(ss, sizeof (ss), "%s@%s",
562                                     bt.obe_root_ds, bt.obe_snap_name);

564                                 /* Create the snapshots recursively */
565                                 if (zfs_snapshot(g_zfs, ss, B_TRUE, ss_props)
566                                     != 0) {
567                                         if (libzfs_errno(g_zfs) !=
568                                             EZFS_EXISTS) {
```



```
569                                                 be_print_err(gettext(
570                                                     "be_create_snapshot: "
571                                                     "recursive snapshot of %s "
572                                                     "failed: %s\n"), ss,
573                                                     libzfs_error_description(
574                                                     g_zfs));
575                                                 ret = zfs_err_to_be_err(g_zfs);
576                                                 goto done;
577                                         }
578                                 } else {
579                                         break;
580                                 }
581                         }

583                         /*
584                          * If we exhausted the maximum number of tries,
585                          * free the auto snap name and set error.
586                          */
587                         if (i == BE_AUTO_NAME_MAX_TRY) {
588                                 be_print_err(gettext("be_create_snapshot: "
589                                     "failed to create unique auto snapshot "
590                                     "name\n"));
591                                 free(bt.obe_snap_name);
592                                 bt.obe_snap_name = NULL;
593                                 ret = BE_ERR_AUTONAME;
594                         }
595                 }
596         }

598         /*
599          * If we succeeded in creating an auto named snapshot, store
600          * the name in the nvlist passed in by the caller.
601          */
602         if (autoname && bt.obe_snap_name) {
603                 *snap_name = bt.obe_snap_name;
604         }

606 done:
607         ZFS_CLOSE(zhp);

609         if (ss_props != NULL)
609         nvlist_free(ss_props);

611         return (ret);
612 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    8169 Mon Feb 15 12:56:08 2016
new/usr/src/lib/libcmdutils/common/process_xattrs.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 71 /*
 72  * mv_xattrs - Copies the content of the extended attribute files. Then
 73  *      moves the extended system attributes from the input attribute files
 74  *      to the target attribute files. Moves the extended system attributes
 75  *      from source to the target file. This function returns 0 on success
 76  *      and nonzero on error.
 77  */
 78 int
 79 mv_xattrs(char *cmd, char *infile, char *outfile, int sattr, int silent)
 80 {
 81         int srcfd = -1;
 82         int indfd = -1;
 83         int outdfd = -1;
 84         int tmpfd = -1;
 85         int sattrfd = -1;
 86         int tattrfd = -1;
 87         int asfd = -1;
 88         int atfd = -1;
 89         DIR *dirp = NULL;
 90         struct dirent *dp = NULL;
 91         char *etext = NULL;
 92         struct stat st1;
 93         struct stat st2;
 94         nvlist_t *response = NULL;
 95         nvlist_t *res = NULL;

 97         if ((srcfd = open(infile, O_RDONLY)) == -1) {
 98                 etext = dgettext(TEXT_DOMAIN, "cannot open source");
 99                 goto error;
100         }
101         if (sattr)
102                 response = sysattr_list(cmd, srcfd, infile);

104         if ((indfd = openat(srcfd, ".", O_RDONLY|O_XATTR)) == -1) {
105                 etext = dgettext(TEXT_DOMAIN, "cannot openat source");
106                 goto error;
107         }
108         if ((outdfd = attropen(outfile, ".", O_RDONLY)) == -1) {
109                 etext = dgettext(TEXT_DOMAIN, "cannot attropen target");
110                 goto error;
111         }
112         if ((tmpfd = dup(indfd)) == -1) {
113                 etext = dgettext(TEXT_DOMAIN, "cannot dup descriptor");
114                 goto error;

116         }
117         if ((dirp = fdopendir(tmpfd)) == NULL) {
118                 etext = dgettext(TEXT_DOMAIN, "cannot access source");
119                 goto error;
120         }
121         while ((dp = readdir(dirp)) != NULL) {
122                 if ((dp->d_name[0] == '.' && dp->d_name[1] == '\0') ||
123                     (dp->d_name[0] == '.' && dp->d_name[1] == '.' &&
124                     dp->d_name[2] == '\0') ||
125                     (sysattr_type(dp->d_name) == _RO_SATTR) ||
126                     (sysattr_type(dp->d_name) == _RW_SATTR))
127                         continue;

129                 if ((sattrfd = openat(indfd, dp->d_name,
```



```
130                     O_RDONLY)) == -1) {
131                         etext = dgettext(TEXT_DOMAIN,
132                             "cannot open src attribute file");
133                         goto error;
134                 }
135                 if (fstat(sattrfd, &st1) < 0) {
136                         etext = dgettext(TEXT_DOMAIN,
137                             "could not stat attribute file");
138                         goto error;
139                 }
140                 if ((tattrfd = openat(outdfd, dp->d_name,
141                     O_RDWR|O_CREAT|O_TRUNC, st1.st_mode)) == -1) {
142                         etext = dgettext(TEXT_DOMAIN,
143                             "cannot open target attribute file");
144                         goto error;
145                 }
146                 if (fstat(tattrfd, &st2) < 0) {
147                         etext = dgettext(TEXT_DOMAIN,
148                             "could not stat attribute file");
149                         goto error;
150                 }
151                 if (writefile(sattrfd, tattrfd, infile, outfile, dp->d_name,
152                     dp->d_name, &st1, &st2) != 0) {
153                         etext = dgettext(TEXT_DOMAIN,
154                             "failed to copy extended attribute "
155                             "from source to target");
156                         goto error;
157                 }

159                 errno = 0;
160                 if (sattr) {
161                         /*
162                          * Gets non default extended system attributes from
163                          * source to copy to target.
164                          */
165                         if (dp->d_name != NULL)
166                                 res = sysattr_list(cmd, sattrfd, dp->d_name);

168                         if (res != NULL &&
169                             get_attrdirs(indfd, outdfd, dp->d_name, &asfd,
170                             &atfd) != 0) {
171                                 etext = dgettext(TEXT_DOMAIN,
172                                     "Failed to open attribute files");
173                                 goto error;
174                         }
175                         /*
176                          * Copy extended system attribute from source
177                          * attribute file to target attribute file
178                          */
179                         if (res != NULL &&
180                             (renameat(asfd, VIEW_READWRITE, atfd,
181                             VIEW_READWRITE) != 0)) {
182                                 if (errno == EPERM)
183                                         etext = dgettext(TEXT_DOMAIN,
184                                             "Permission denied -"
185                                             "failed to move system attribute");
186                                 else
187                                         etext = dgettext(TEXT_DOMAIN,
188                                             "failed to move extended "
189                                             "system attribute");
190                                 goto error;
191                         }
192                 }
193                 if (sattrfd != -1)
194                         (void) close(sattrfd);
195                 if (tattrfd != -1)
```

```
196                         (void) close(tattrfd);
197                 if (asfd != -1)
198                         (void) close(asfd);
199                 if (atfd != -1)
200                         (void) close(atfd);
201                 if (res != NULL) {
202                         nvlist_free(res);
203                         res = NULL;
204                 }
205         }
206         errno = 0;
207         /* Copy extended system attribute from source to target */

209         if (response != NULL) {
210                 if (renameat(indfd, VIEW_READWRITE, outdfd,
211                     VIEW_READWRITE) == 0)
212                         goto done;

214                 if (errno == EPERM)
215                         etext = dgettext(TEXT_DOMAIN, "Permission denied");
216                 else
217                         etext = dgettext(TEXT_DOMAIN,
218                             "failed to move system attribute");
219         }
220 error:
221         if (res != NULL)
221         nvlist_free(res);
222         if (silent == 0 && etext != NULL) {
223                 if (!sattr)
224                         (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
225                             "%s: %s: cannot move extended attributes, "),
226                             cmd, infile);
227                 else
228                         (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
229                             "%s: %s: cannot move extended system "
230                             "attributes, "), cmd, infile);
231                 perror(etext);
232         }
233 done:
234         if (dirp)
235                 (void) closedir(dirp);
236         if (sattrfd != -1)
237                 (void) close(sattrfd);
238         if (tattrfd != -1)
239                 (void) close(tattrfd);
240         if (asfd != -1)
241                 (void) close(asfd);
242         if (atfd != -1)
243                 (void) close(atfd);
244         if (indfd != -1)
245                 (void) close(indfd);
246         if (outdfd != -1)
247                 (void) close(outdfd);
249         if (response != NULL)
248         nvlist_free(response);
249         if (etext != NULL)
250                 return (1);
251         else
252                 return (0);
253 }

255 /*
256  * The function returns non default extended system attribute list
257  * associated with 'fname' and returns NULL when an error has occured
258  * or when only extended system attributes other than archive,
259  * av_modified or crtime are set.
```



```
260  *
261  * The function returns system attribute list for the following cases:
262  *
263  *      - any extended system attribute other than the default attributes
264  *        ('archive', 'av_modified' and 'crtime') is set
265  *      - nvlist has NULL name string
266  *      - nvpair has data type of 'nvlist'
267  *      - default data type.
268  */

270 nvlist_t *
271 sysattr_list(char *cmd, int fd, char *fname)
272 {
273         boolean_t       value;
274         data_type_t     type;
275         nvlist_t        *response;
276         nvpair_t        *pair;
277         f_attr_t        fattr;
278         char            *name;

280         if (fgetattr(fd, XATTR_VIEW_READWRITE, &response) != 0) {
281                 (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
282                     "%s: %s: fgetattr failed\n"),
283                     cmd, fname);
284                 return (NULL);
285         }
286         pair = NULL;
287         while ((pair = nvlist_next_nvpair(response, pair)) != NULL) {

289                 name = nvpair_name(pair);

291                 if (name != NULL)
292                         fattr = name_to_attr(name);
293                 else
294                         return (response);

296                 type = nvpair_type(pair);
297                 switch (type) {
298                         case DATA_TYPE_BOOLEAN_VALUE:
299                                 if (nvpair_value_boolean_value(pair,
300                                     &value) != 0) {
301                                         (void) fprintf(stderr,
302                                             dgettext(TEXT_DOMAIN, "%s "
303                                             "nvpair_value_boolean_value "
304                                             "failed\n"), cmd);
305                                         continue;
306                                 }
307                                 if (value && fattr != F_ARCHIVE &&
308                                     fattr != F_AV_MODIFIED)
309                                         return (response);
310                                 break;
311                         case DATA_TYPE_UINT64_ARRAY:
312                                 if (fattr != F_CRTIME)
313                                         return (response);
314                                 break;
315                         case DATA_TYPE_NVLIST:
316                         default:
317                                 return (response);
318                 }
319         }
322         if (response != NULL)
320         nvlist_free(response);
321         return (NULL);
322 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10585 Mon Feb 15 12:56:08 2016
new/usr/src/lib/libcontract/common/libcontract.c
patch cleanup
6659 nvlist_free(NULL) is a no-op
**********************************************************
```

```
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/ctfs.h>
#include <sys/contract.h>
#include <string.h>
#include <libnvpair.h>
#include <assert.h>
#include <unistd.h>
#include <errno.h>
#include <libcontract.h>
#include "libcontract_impl.h"

/*
 * Common template routines
 */

int
ct_tmpl_activate(int fd)
{
        if (ioctl(fd, CT_TACTIVATE) == -1)
                return (errno);
        return (0);
}
```

*unchanged_portion_omitted_*

```
static int
ct_event_read_internal(int fd, int cmd, ct_evthdl_t *evt)
{
        char *event_buffer = NULL;
        int event_nbytes = 0;
        struct ctlib_event_info *info;
        ct_event_t *event;
        int error;

        info = malloc(sizeof (struct ctlib_event_info));
```



```
        if (info == NULL)
                return (errno);
        info->nvl = NULL;
        event = &info->event;

        for (;;) {
                event->ctev_nbytes = event_nbytes;
                event->ctev_buffer = event_buffer;
                do
                        error = ioctl(fd, cmd, event);
                while (error == -1 && errno == EINTR);
                if (error == -1) {
                        error = errno;
                        goto errout;
                }
                if (event->ctev_nbytes <= event_nbytes)
                        break;

                if (event_buffer)
                        free(event_buffer);
                event_nbytes = event->ctev_nbytes;
                event_buffer = malloc(event_nbytes);
                if (event_buffer == NULL) {
                        error = errno;
                        goto errout;
                }
        }

        if (event->ctev_goffset > 0 && (error = unpack_and_merge(&info->nvl,
            event->ctev_buffer, event->ctev_goffset)) != 0)
                goto errout;

        if (event->ctev_goffset < event->ctev_nbytes &&
            (error = unpack_and_merge(&info->nvl,
            event->ctev_buffer + event->ctev_goffset,
            event->ctev_nbytes - event->ctev_goffset)) != 0)
                goto errout;

        free(event_buffer);

        *evt = info;
        return (0);

errout:
        if (event_buffer)
                free(event_buffer);
        if (info) {
                if (info->nvl)
                nvlist_free(info->nvl);
                free(info);
        }
        return (error);
}
```

*unchanged_portion_omitted_*

```
void
ct_event_free(ct_evthdl_t evthdl)
{
        struct ctlib_event_info *info = evthdl;

        if (info->nvl)
        nvlist_free(info->nvl);
        free(info);
}
```

*unchanged_portion_omitted_*

```
**********************************************************
   27625 Mon Feb 15 12:56:08 2016
new/usr/src/lib/libdiskmgt/common/entry.c
patch tsoome-feedback
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

561 /*
562  * Checks for overlapping slices.   If the given device is a slice, and it
563  * overlaps with any non-backup slice on the disk, return true with a detailed
564  * description similar to dm_inuse().
565  */
566 int
567 dm_isoverlapping(char *slicename, char **overlaps_with, int *errp)
568 {
569         dm_descriptor_t slice = NULL;
570         dm_descriptor_t *media = NULL;
571         dm_descriptor_t *slices = NULL;
572         int             i = 0;
573         uint32_t        in_snum;
574         uint64_t        start_block = 0;
575         uint64_t        end_block = 0;
576         uint64_t        media_size = 0;
577         uint64_t        size = 0;
578         nvlist_t        *media_attrs = NULL;
579         nvlist_t        *slice_attrs = NULL;
580         int             ret = 0;

582         slice = dm_get_descriptor_by_name(DM_SLICE, slicename, errp);
583         if (slice == NULL)
584                 goto out;

586         /*
587          * Get the list of slices be fetching the associated media, and then all
588          * associated slices.
589          */
590         media = dm_get_associated_descriptors(slice, DM_MEDIA, errp);
591         if (media == NULL || *media == NULL || *errp != 0)
592                 goto out;

594         slices = dm_get_associated_descriptors(*media, DM_SLICE, errp);
595         if (slices == NULL || *slices == NULL || *errp != 0)
596                 goto out;

598         media_attrs = dm_get_attributes(*media, errp);
599         if (media_attrs == NULL || *errp)
600                 goto out;

602         *errp = nvlist_lookup_uint64(media_attrs, DM_NACCESSIBLE, &media_size);
603         if (*errp != 0)
604                 goto out;

606         slice_attrs = dm_get_attributes(slice, errp);
607         if (slice_attrs == NULL || *errp != 0)
608                 goto out;

610         *errp = nvlist_lookup_uint64(slice_attrs, DM_START, &start_block);
611         if (*errp != 0)
612                 goto out;

614         *errp = nvlist_lookup_uint64(slice_attrs, DM_SIZE, &size);
615         if (*errp != 0)
616                 goto out;

618         *errp = nvlist_lookup_uint32(slice_attrs, DM_INDEX, &in_snum);
```



```
619         if (*errp != 0)
620                 goto out;

622         end_block = (start_block + size) - 1;

624         for (i = 0; slices[i]; i ++) {
625                 uint64_t other_start;
626                 uint64_t other_end;
627                 uint64_t other_size;
628                 uint32_t snum;

630                 nvlist_t *other_attrs = dm_get_attributes(slices[i], errp);

632                 if (other_attrs == NULL)
633                         continue;

635                 if (*errp != 0)
636                         goto out;

638                 *errp = nvlist_lookup_uint64(other_attrs, DM_START,
639                     &other_start);
640                 if (*errp) {
641                         nvlist_free(other_attrs);
642                         goto out;
643                 }

645                 *errp = nvlist_lookup_uint64(other_attrs, DM_SIZE,
646                     &other_size);

648                 if (*errp) {
649                         nvlist_free(other_attrs);
650                         ret = -1;
651                         goto out;
652                 }

654                 other_end = (other_size + other_start) - 1;

656                 *errp = nvlist_lookup_uint32(other_attrs, DM_INDEX,
657                     &snum);

659                 if (*errp) {
660                         nvlist_free(other_attrs);
661                         ret = -1;
662                         goto out;
663                 }

665                 /*
666                  * Check to see if there are > 2 overlapping regions
667                  * on this media in the same region as this slice.
668                  * This is done by assuming the following:
669                  *      Slice 2 is the backup slice if it is the size
670                  *      of the whole disk
671                  * If slice 2 is the overlap and slice 2 is the size of
672                  * the whole disk, continue. If another slice is found
673                  * that overlaps with our slice, return it.
674                  * There is the potential that there is more than one slice
675                  * that our slice overlaps with, however, we only return
676                  * the first overlapping slice we find.
677                  *
678                  */
679                 if (start_block >= other_start && start_block <= other_end) {
680                         if ((snum == 2 && (other_size == media_size)) ||
681                             snum == in_snum) {
682                                 continue;
683                         } else {
684                                 char *str = dm_get_name(slices[i], errp);
```

```
685                                 if (*errp != 0) {
686                                         nvlist_free(other_attrs);
687                                         ret = -1;
688                                         goto out;
689                                 }
690                                 *overlaps_with = strdup(str);
691                                 dm_free_name(str);
692                                 nvlist_free(other_attrs);
693                                 ret = 1;
694                                 goto out;
695                         }
696                 } else if (other_start >= start_block &&
697                     other_start <= end_block) {
698                         if ((snum == 2 && (other_size == media_size)) ||
699                             snum == in_snum) {
700                                 continue;
701                         } else {
702                                 char *str = dm_get_name(slices[i], errp);
703                                 if (*errp != 0) {
704                                         nvlist_free(other_attrs);
705                                         ret = -1;
706                                         goto out;
707                                 }
708                                 *overlaps_with = strdup(str);
709                                 dm_free_name(str);
710                                 nvlist_free(other_attrs);
711                                 ret = 1;
712                                 goto out;
713                         }
714                 }
715                 nvlist_free(other_attrs);
716         }

718 out:
719         if (media_attrs)
719         nvlist_free(media_attrs);
721         if (slice_attrs)
720         nvlist_free(slice_attrs);

722         if (slices)
723                 dm_free_descriptors(slices);
724         if (media)
725                 dm_free_descriptors(media);
726         if (slice)
727                 dm_free_descriptor(slice);

729         return (ret);
730 }
```

*unchanged_portion_omitted_*

```
839 /*
840  * Returns 'in use' details, if found, about a specific dev_name,
841  * based on the caller(who). It is important to note that it is possible
842  * for there to be more than one 'in use' statistic regarding a dev_name.
843  * The **msg parameter returns a list of 'in use' details. This message
844  * is formatted via gettext().
845  */
846 int
847 dm_inuse(char *dev_name, char **msg, dm_who_type_t who, int *errp)
848 {
849         nvlist_t *dev_stats = NULL;
850         char *by, *data;
851         nvpair_t *nvwhat = NULL;
852         nvpair_t *nvdesc = NULL;
853         int     found = 0;
854         int     err;
```



```
855         char    *dname = NULL;

857         *errp = 0;
858         *msg = NULL;

860         /*
861          * If the user doesn't want to do in use checking, return.
862          */

864         if (NOINUSE_SET)
865                 return (0);

867         dname = getfullblkname(dev_name);
868         /*
869          * If we cannot find the block name, we cannot check the device
870          * for in use statistics. So, return found, which is == 0.
871          */
872         if (dname == NULL || *dname == '\0') {
873                 return (found);
874         }

876         /*
877          * Slice stats for swap devices are only returned if mounted
878          * (e.g. /tmp).  Other devices or files being used for swap
879          * are ignored, so we add a special check here to use swapctl(2)
880          * to perform in-use checking.
881          */
882         if (ANY_ZPOOL_USE(who) && (err = dm_inuse_swap(dname, errp))) {

884                 /* on error, dm_inuse_swap sets errp */
885                 if (err < 0) {
886                         free(dname);
887                         return (err);
888                 }

890                 /* simulate a mounted swap device */
891                 (void) build_usage_string(dname, DM_USE_MOUNT, "swap", msg,
892                     &found, errp);

894                 /* if this fails, dm_get_usage_string changed */
895                 ASSERT(found == 1);

897                 free(dname);
898                 return (found);
899         }

901         dm_get_slice_stats(dname, &dev_stats, errp);
902         if (dev_stats == NULL) {
903                 /*
904                  * If there is an error, but it isn't a no device found error
905                  * return the error as recorded. Otherwise, with a full
906                  * block name, we might not be able to get the slice
907                  * associated, and will get an ENODEV error. For example,
908                  * an SVM metadevice will return a value from getfullblkname()
909                  * but libdiskmgt won't be able to find this device for
910                  * statistics gathering. This is expected and we should not
911                  * report errnoneous errors.
912                  */
913                 if (*errp) {
914                         if (*errp == ENODEV) {
915                                 *errp = 0;
916                         }
917                 }
918                 free(dname);
919                 return (found);
920         }
```

```
922         for (;;) {

924                 nvwhat = nvlist_next_nvpair(dev_stats, nvdesc);
925                 nvdesc = nvlist_next_nvpair(dev_stats, nvwhat);

927                 /*
928                  * End of the list found.
929                  */
930                 if (nvwhat == NULL || nvdesc == NULL) {
931                         break;
932                 }
933                 /*
934                  * Otherwise, we check to see if this client(who) cares
935                  * about this in use scenario
936                  */

938                 ASSERT(strcmp(nvpair_name(nvwhat), DM_USED_BY) == 0);
939                 ASSERT(strcmp(nvpair_name(nvdesc), DM_USED_NAME) == 0);
940                 /*
941                  * If we error getting the string value continue on
942                  * to the next pair(if there is one)
943                  */
944                 if (nvpair_value_string(nvwhat, &by)) {
945                         continue;
946                 }
947                 if (nvpair_value_string(nvdesc, &data)) {
948                         continue;
949                 }

951                 switch (who) {
952                         case DM_WHO_MKFS:
953                                 /*
954                                  * mkfs is not in use for these cases.
955                                  * All others are in use.
956                                  */
957                                 if (strcmp(by, DM_USE_LU) == 0 ||
958                                     strcmp(by, DM_USE_FS) == 0 ||
959                                     strcmp(by, DM_USE_EXPORTED_ZPOOL) == 0) {
960                                         break;
961                                 }
962                                 if (build_usage_string(dname,
963                                     by, data, msg, &found, errp) != 0) {
964                                         if (*errp) {
965                                                 goto out;
966                                         }
967                                 }
968                                 break;
969                         case DM_WHO_SWAP:
970                                 /*
971                                  * Not in use for this.
972                                  */
973                                 if (strcmp(by, DM_USE_DUMP) == 0 ||
974                                     strcmp(by, DM_USE_FS) == 0 ||
975                                     strcmp(by, DM_USE_EXPORTED_ZPOOL) == 0) {
976                                         break;
977                                 }
978                                 if (strcmp(by, DM_USE_LU) == 0 &&
979                                     strcmp(data, "-") == 0) {
980                                         break;
981                                 }
982                                 if (strcmp(by, DM_USE_VFSTAB) == 0 &&
983                                     strcmp(data, "") == 0) {
984                                         break;
985                                 }
986                                 if (build_usage_string(dname,
```



```
987                                     by, data, msg, &found, errp) != 0) {
988                                         if (*errp) {
989                                                 goto out;
990                                         }
991                                 }
992                                 break;
993                         case DM_WHO_DUMP:
994                                 /*
995                                  * Not in use for this.
996                                  */
997                                 if ((strcmp(by, DM_USE_MOUNT) == 0 &&
998                                     strcmp(data, "swap") == 0) ||
999                                     strcmp(by, DM_USE_DUMP) == 0 ||
1000                                    strcmp(by, DM_USE_FS) == 0 ||
1001                                    strcmp(by, DM_USE_EXPORTED_ZPOOL) == 0) {
1002                                        break;
1003                                }
1004                                if (build_usage_string(dname,
1005                                    by, data, msg, &found, errp)) {
1006                                        if (*errp) {
1007                                                goto out;
1008                                        }
1009                                }
1010                                break;

1012                        case DM_WHO_FORMAT:
1013                                if (strcmp(by, DM_USE_FS) == 0 ||
1014                                    strcmp(by, DM_USE_EXPORTED_ZPOOL) == 0)
1015                                        break;
1016                                if (build_usage_string(dname,
1017                                    by, data, msg, &found, errp) != 0) {
1018                                        if (*errp) {
1019                                                goto out;
1020                                        }
1021                                }
1022                                break;

1024                        case DM_WHO_ZPOOL_FORCE:
1025                                if (strcmp(by, DM_USE_FS) == 0 ||
1026                                    strcmp(by, DM_USE_EXPORTED_ZPOOL) == 0)
1027                                        break;
1028                                /* FALLTHROUGH */
1029                        case DM_WHO_ZPOOL:
1030                                if (build_usage_string(dname,
1031                                    by, data, msg, &found, errp) != 0) {
1032                                        if (*errp)
1033                                                goto out;
1034                                }
1035                                break;

1037                        case DM_WHO_ZPOOL_SPARE:
1038                                if (strcmp(by, DM_USE_SPARE_ZPOOL) != 0) {
1039                                        if (build_usage_string(dname, by,
1040                                            data, msg, &found, errp) != 0) {
1041                                                if (*errp)
1042                                                        goto out;
1043                                        }
1044                                }
1045                                break;

1047                        default:
1048                                /*
1049                                 * nothing found in use for this client
1050                                 * of libdiskmgt. Default is 'not in use'.
1051                                 */
1052                                break;
```

```
1053                    }
1054            }
1055 out:
1056            if (dname != NULL)
1057                    free(dname);
1060            if (dev_stats != NULL)
1058            nvlist_free(dev_stats);

1060            return (found);
1061 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   26125 Mon Feb 15 12:56:08 2016
new/usr/src/lib/libdladm/common/libdllink.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 465 /*
 466  * Case 2: rename an available physical link link1 to a REMOVED physical link
 467  *     link2.  As a result, link1 directly inherits all datalinks configured
 468  *     over link2 (linkid2).
 469  * Result: <linkid2, link2, link1_phymaj, link1_phyinst, link1_devname,
 470  *     link2_other_attr>
 471  */
 472 static dladm_status_t
 473 i_dladm_rename_link_c2(dladm_handle_t handle, datalink_id_t linkid1,
 474     datalink_id_t linkid2)
 475 {
 476         rcm_handle_t            *rcm_hdl = NULL;
 477         nvlist_t                *nvl = NULL;
 478         link_hold_arg_t         arg;
 479         dld_ioc_rename_t        dir;
 480         dladm_conf_t            conf1, conf2;
 481         char                    devname[MAXLINKNAMELEN];
 482         uint64_t                phymaj, phyinst;
 483         dladm_status_t          status = DLADM_STATUS_OK;

 485         /*
 486          * First check if linkid1 is associated with any persistent
 487          * aggregations or VLANs. If yes, return BUSY.
 488          */
 489         arg.linkid = linkid1;
 490         arg.holder = DATALINK_INVALID_LINKID;
 491         arg.flags = DLADM_OPT_PERSIST;
 492         (void) dladm_walk_datalink_id(i_dladm_aggr_link_hold, handle, &arg,
 493             DATALINK_CLASS_AGGR, DATALINK_ANY_MEDIATYPE, DLADM_OPT_PERSIST);
 494         if (arg.holder != DATALINK_INVALID_LINKID)
 495                 return (DLADM_STATUS_LINKBUSY);

 497         arg.flags = DLADM_OPT_PERSIST;
 498         (void) dladm_walk_datalink_id(i_dladm_vlan_link_hold, handle, &arg,
 499             DATALINK_CLASS_VLAN, DATALINK_ANY_MEDIATYPE, DLADM_OPT_PERSIST);
 500         if (arg.holder != DATALINK_INVALID_LINKID)
 501                 return (DLADM_STATUS_LINKBUSY);

 503         /*
 504          * Send DLDIOC_RENAME to request to rename link1's linkid to
 505          * be linkid2. This will check whether link1 is used by any
 506          * aggregations or VLANs, or is held by any application. If yes,
 507          * return failure.
 508          */
 509         dir.dir_linkid1 = linkid1;
 510         dir.dir_linkid2 = linkid2;
 511         if (ioctl(dladm_dld_fd(handle), DLDIOC_RENAME, &dir) < 0)
 512                 status = dladm_errno2status(errno);

 514         if (status != DLADM_STATUS_OK) {
 515                 return (status);
 516         }

 518         /*
 519          * Now change the phymaj, phyinst and devname associated with linkid1
 520          * to be associated with linkid2. Before doing that, the old active
 521          * linkprop of linkid1 should be deleted.
 522          */
 523         (void) dladm_set_linkprop(handle, linkid1, NULL, NULL, 0,
```



```
 524             DLADM_OPT_ACTIVE);

 526         if (((status = dladm_getsnap_conf(handle, linkid1, &conf1)) !=
 527             DLADM_STATUS_OK) ||
 528             ((status = dladm_get_conf_field(handle, conf1, FDEVNAME, devname,
 529             MAXLINKNAMELEN)) != DLADM_STATUS_OK) ||
 530             ((status = dladm_get_conf_field(handle, conf1, FPHYMAJ, &phymaj,
 531             sizeof (uint64_t))) != DLADM_STATUS_OK) ||
 532             ((status = dladm_get_conf_field(handle, conf1, FPHYINST, &phyinst,
 533             sizeof (uint64_t))) != DLADM_STATUS_OK) ||
 534             ((status = dladm_open_conf(handle, linkid2, &conf2)) !=
 535             DLADM_STATUS_OK)) {
 536                 dir.dir_linkid1 = linkid2;
 537                 dir.dir_linkid2 = linkid1;
 538                 (void) dladm_init_linkprop(handle, linkid1, B_FALSE);
 539                 (void) ioctl(dladm_dld_fd(handle), DLDIOC_RENAME, &dir);
 540                 return (status);
 541         }

 543         dladm_destroy_conf(handle, conf1);
 544         (void) dladm_set_conf_field(handle, conf2, FDEVNAME, DLADM_TYPE_STR,
 545             devname);
 546         (void) dladm_set_conf_field(handle, conf2, FPHYMAJ, DLADM_TYPE_UINT64,
 547             &phymaj);
 548         (void) dladm_set_conf_field(handle, conf2, FPHYINST,
 549             DLADM_TYPE_UINT64, &phyinst);
 550         (void) dladm_write_conf(handle, conf2);
 551         dladm_destroy_conf(handle, conf2);

 553         /*
 554          * Delete link1 and mark link2 up.
 555          */
 556         (void) dladm_remove_conf(handle, linkid1);
 557         (void) dladm_destroy_datalink_id(handle, linkid1, DLADM_OPT_ACTIVE |
 558             DLADM_OPT_PERSIST);
 559         (void) dladm_up_datalink_id(handle, linkid2);

 561         /*
 562          * Now generate the RCM_RESOURCE_LINK_NEW sysevent which can be
 563          * consumed by the RCM framework to restore all the datalink and
 564          * IP configuration.
 565          */
 566         status = DLADM_STATUS_FAILED;
 567         if ((nvlist_alloc(&nvl, 0, 0) != 0) ||
 568             (nvlist_add_uint64(nvl, RCM_NV_LINKID, linkid2) != 0)) {
 569                 goto done;
 570         }

 572         if (rcm_alloc_handle(NULL, 0, NULL, &rcm_hdl) != RCM_SUCCESS)
 573                 goto done;

 575         if (rcm_notify_event(rcm_hdl, RCM_RESOURCE_LINK_NEW, 0, nvl, NULL) ==
 576             RCM_SUCCESS) {
 577                 status = DLADM_STATUS_OK;
 578         }

 580 done:
 581         if (rcm_hdl != NULL)
 582                 (void) rcm_free_handle(rcm_hdl);
 583         if (nvl != NULL)
 583         nvlist_free(nvl);
 584         return (status);
 585 }
______unchanged_portion_omitted_
```

```
**********************************************************
   16995 Mon Feb 15 12:56:08 2016
new/usr/src/lib/libipp/libipp.c
patch tsoome-feedback
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (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 2001-2002 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <strings.h>
#include <string.h>
#include <fcntl.h>
#include <assert.h>
#include <libipp.h>
#include <libnvpair.h>
#include <ipp/ippctl.h>

/*
 * Debug macros
 */

#if     defined(DEBUG) && !defined(lint)
uint32_t        ipp_debug_flags =
/*
 * DBG_IO |
 */
DBG_ERR |
0;

#define DBG0(flags, fmt)                                                \
        do {                                                            \
                if (flags & ipp_debug_flags)                            \
                        fprintf(stderr, "libipp: " __FN__ ": " fmt);    \
        } while (0)

#define DBG1(flags, fmt, a)                                             \
```



```
        do {                                                            \
                if (flags & ipp_debug_flags)                            \
                        fprintf(stderr, "libipp: " __FN__ ": " fmt, a); \
        } while (0)

#define DBG2(flags, fmt, a, b)                                          \
        do {                                                            \
                if (flags & ipp_debug_flags)                            \
                        fprintf(stderr, "libipp: " __FN__ ": " fmt, a,  \
                            b);                                         \
        } while (0)

#define DBG3(flags, fmt, a, b, c)                                       \
        do {                                                            \
                if (flags & ipp_debug_flags)                            \
                        fprintf(stderr, "libipp: " __FN__ ": " fmt, a,  \
                            b, c);                                      \
        } while (0)

#else   /* defined(DEBUG) && !defined(lint) */
#define DBG0(flags, fmt)
#define DBG1(flags, fmt, a)
#define DBG2(flags, fmt, a, b)
#define DBG3(flags, fmt, a, b, c)
#endif  /* defined(DEBUG) && !defined(lint) */

/*
 * Control device node
 */

#define IPPCTL_DEVICE   "/devices/pseudo/ippctl@0:ctl"

/*
 * Structures.
 */

typedef struct array_desc_t {
        char    *name;
        char    **array;
        int     nelt;
} array_desc_t;
```

*unchanged_portion_omitted_*

```
#undef  __FN__

#define __FN__  "ipp_action_destroy"
int
ipp_action_destroy(
        const char      *aname,
        ipp_flags_t     flags)
{
        nvlist_t        *nvlp;
        int             rc;

        /*
         * Sanity check the arguments.
         */

        if (aname == NULL) {
                DBG0(DBG_ERR, "bad argument\n");
                errno = EINVAL;
                return (-1);
        }

        /*
         * Create an nvlist for our data as none is passed into the function.
         */
```

```
198         if ((rc = nvlist_alloc(&nvlp, NV_UNIQUE_NAME, 0)) != 0) {
199                 DBG0(DBG_ERR, "failed to allocate nvlist\n");
200                 nvlp = NULL;
201                 goto failed;
202         }

204         if ((rc = nvlist_add_byte(nvlp, IPPCTL_OP,
205             IPPCTL_OP_ACTION_DESTROY)) != 0) {
206                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_OP);
207                 goto failed;
208         }

210         if ((rc = nvlist_add_string(nvlp, IPPCTL_ANAME, (char *)aname)) != 0) {
211                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_ANAME);
212                 goto failed;
213         }

215         if ((rc = nvlist_add_uint32(nvlp, IPPCTL_FLAGS, flags)) != 0) {
216                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_FLAGS);
217                 goto failed;
218         }

220         /*
221          * Talk to the kernel.
222          */

224         return (dispatch(&nvlp, NULL, NULL));
225 failed:
228         if (nvlp != NULL)
226         nvlist_free(nvlp);
227         errno = rc;
228         return (-1);
229 }
_____unchanged_portion_omitted_
282 #undef  __FN__

284 #define __FN__  "ipp_action_info"
285 int
286 ipp_action_info(
287         const char      *aname,
288         int             (*fn)(nvlist_t *, void *),
289         void            *arg,
290         ipp_flags_t     flags)
291 {
292         nvlist_t        *nvlp;
293         int             rc;

295         /*
296          * Sanity check the arguments.
297          */

299         if (aname == NULL || fn == NULL) {
300                 DBG0(DBG_ERR, "bad argument\n");
301                 errno = EINVAL;
302                 return (-1);
303         }

305         /*
306          * Create an nvlist for our data.
307          */

309         if ((rc = nvlist_alloc(&nvlp, NV_UNIQUE_NAME, 0)) != 0) {
310                 DBG0(DBG_ERR, "failed to allocate nvlist\n");
311                 nvlp = NULL;
312         }
```



```
314         if ((rc = nvlist_add_byte(nvlp, IPPCTL_OP,
315             IPPCTL_OP_ACTION_INFO)) != 0) {
316                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_OP);
317                 goto failed;
318         }

320         if ((rc = nvlist_add_string(nvlp, IPPCTL_ANAME, (char *)aname)) != 0) {
321                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_ANAME);
322                 goto failed;
323         }

325         if ((rc = nvlist_add_uint32(nvlp, IPPCTL_FLAGS, flags)) != 0) {
326                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_FLAGS);
327                 goto failed;
328         }

330         /*
331          * Talk to the kernel.
332          */

334         return (dispatch(&nvlp, fn, arg));
335 failed:
339         if (nvlp != NULL)
336         nvlist_free(nvlp);
337         errno = rc;
338         return (-1);
339 }
340 #undef  __FN__

342 #define __FN__  "ipp_action_mod"
343 int
344 ipp_action_mod(
345         const char      *aname,
346         char            **modnamep)
347 {
348         nvlist_t        *nvlp;
349         int             rc;

351         /*
352          * Sanity check the arguments.
353          */

355         if (aname == NULL || modnamep == NULL) {
356                 DBG0(DBG_ERR, "bad argument\n");
357                 errno = EINVAL;
358                 return (-1);
359         }

361         /*
362          * Create an nvlist for our data.
363          */

365         if ((rc = nvlist_alloc(&nvlp, NV_UNIQUE_NAME, 0)) != 0) {
366                 DBG0(DBG_ERR, "failed to allocate nvlist\n");
367                 nvlp = NULL;
368                 goto failed;
369         }

371         if ((rc = nvlist_add_byte(nvlp, IPPCTL_OP,
372             IPPCTL_OP_ACTION_MOD)) != 0) {
373                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_OP);
374                 goto failed;
375         }

377         if ((rc = nvlist_add_string(nvlp, IPPCTL_ANAME, (char *)aname)) != 0) {
```

```
378                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_ANAME);
379                 goto failed;
380         }

382         /*
383          * Talk to the kernel.
384          */

386         return (dispatch(&nvlp, string_callback, (void *)modnamep));
387 failed:
392         if (nvlp != NULL)
388         nvlist_free(nvlp);
389         errno = rc;
390         return (-1);
391 }
392 #undef  __FN__

394 #define __FN__  "ipp_list_mods"
395 int
396 ipp_list_mods(
397         char            ***modname_arrayp,
398         int             *neltp)
399 {
400         nvlist_t        *nvlp;
401         array_desc_t    ad;
402         int             rc;

404         /*
405          * Sanity check the arguments.
406          */

408         if (modname_arrayp == NULL || neltp == NULL) {
409                 DBG0(DBG_ERR, "bad argument");
410                 errno = EINVAL;
411                 return (-1);
412         }

414         /*
415          * Create an nvlist for our data.
416          */

418         if ((rc = nvlist_alloc(&nvlp, NV_UNIQUE_NAME, 0)) != 0) {
419                 DBG0(DBG_ERR, "failed to allocate nvlist\n");
420                 nvlp = NULL;
421         }

423         if ((rc = nvlist_add_byte(nvlp, IPPCTL_OP,
424             IPPCTL_OP_LIST_MODS)) != 0) {
425                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_OP);
426                 goto failed;
427         }

429         /*
430          * Talk to the kernel.
431          */

433         ad.name = IPPCTL_MODNAME_ARRAY;
434         ad.array = NULL;
435         ad.nelt = 0;

437         if ((rc = dispatch(&nvlp, string_array_callback, (void *)&ad)) == 0) {
438                 *modname_arrayp = ad.array;
439                 *neltp = ad.nelt;
440         }

442         return (rc);
```



```
443 failed:
449         if (nvlp != NULL)
444         nvlist_free(nvlp);
445         errno = rc;
446         return (-1);
447 }
448 #undef  __FN__

450 #define __FN__  "ipp_mod_list_actions"
451 int
452 ipp_mod_list_actions(
453         const char      *modname,
454         char            ***aname_arrayp,
455         int             *neltp)
456 {
457         nvlist_t        *nvlp;
458         array_desc_t    ad;
459         int             rc;

461         /*
462          * Sanity check the arguments.
463          */

465         if (modname == NULL || aname_arrayp == NULL || neltp == NULL) {
466                 DBG0(DBG_ERR, "bad argument");
467                 errno = EINVAL;
468                 return (-1);
469         }

471         /*
472          * Create an nvlist for our data.
473          */

475         if ((rc = nvlist_alloc(&nvlp, NV_UNIQUE_NAME, 0)) != 0) {
476                 DBG0(DBG_ERR, "failed to allocate nvlist\n");
477                 nvlp = NULL;
478         }

480         if ((rc = nvlist_add_byte(nvlp, IPPCTL_OP,
481             IPPCTL_OP_MOD_LIST_ACTIONS)) != 0) {
482                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_OP);
483                 goto failed;
484         }

486         if ((rc = nvlist_add_string(nvlp, IPPCTL_MODNAME,
487             (char *)modname)) != 0) {
488                 DBG1(DBG_ERR, "failed to add '%s' to nvlist\n", IPPCTL_MODNAME);
489                 goto failed;
490         }

492         /*
493          * Talk to the kernel.
494          */

496         ad.name = IPPCTL_ANAME_ARRAY;
497         ad.array = NULL;
498         ad.nelt = 0;

500         if ((rc = dispatch(&nvlp, string_array_callback, (void *)&ad)) == 0) {
501                 *aname_arrayp = ad.array;
502                 *neltp = ad.nelt;
503         }

505         return (rc);
506 failed:
513         if (nvlp != NULL)
```

```
 507         nvlist_free(nvlp);
 508         errno = rc;
 509         return (-1);
 510 }
______unchanged_portion_omitted_
```

```
**********************************************************
   30897 Mon Feb 15 12:56:09 2016
new/usr/src/lib/libnwam/common/libnwam_values.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 380 void
 381 nwam_free_object_list(void *list)
 382 {
 383         if (list != NULL)
 383         nvlist_free(list);
 384 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   96418 Mon Feb 15 12:56:09 2016
new/usr/src/lib/libpool/common/pool_kernel.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

2384 /*
2385  * Update the specified property value.
2386  *
2387  * PO_FAIL is returned if an error is detected and the error code is updated
2388  * to indicate the cause of the error.
2389  */
2390 int
2391 pool_knl_put_property(pool_elem_t *pe, const char *name,
2392     const pool_value_t *val)
2393 {
2394         pool_knl_elem_t *pke = (pool_knl_elem_t *)pe;
2395         pool_knl_connection_t *prov =
2396             (pool_knl_connection_t *)(TO_CONF(pe))->pc_prov;
2397         nvpair_t *bp, *ap;
2398         pool_propput_undo_t *propput;
2399         nvlist_t *bl = NULL;
2400         const pool_prop_t *prop;

2402         if ((bp = pool_knl_find_nvpair(pke->pke_properties, name)) != NULL) {
2403                 if (nvlist_alloc(&bl, NV_UNIQUE_NAME_TYPE, 0) != 0) {
2404                         pool_seterror(POE_SYSTEM);
2405                         return (PO_FAIL);
2406                 }
2407                 if (nvlist_add_nvpair(bl, bp) != 0) {
2408                         nvlist_free(bl);
2409                         pool_seterror(POE_SYSTEM);
2410                         return (PO_FAIL);
2411                 }
2412         }
2413         if (pool_knl_nvlist_add_value(pke->pke_properties, name, val) !=
2414             PO_SUCCESS)
2415                 return (PO_FAIL);

2417         if (prov->pkc_log->l_state != LS_DO) {
2418                 if (bl)
2418                 nvlist_free(bl);
2419                 return (PO_SUCCESS);
2420         }
2421         /*
2422          * The remaining logic is setting up the arguments for the
2423          * POOL_PROPPUT ioctl and appending the details into the log.
2424          */
2425         if ((propput = malloc(sizeof (pool_propput_undo_t))) == NULL) {
2426                 pool_seterror(POE_SYSTEM);
2427                 return (PO_FAIL);
2428         }
2429         (void) memset(propput, 0, sizeof (pool_propput_undo_t));
2430         propput->ppu_blist = bl;

2432         ap = pool_knl_find_nvpair(pke->pke_properties, name);

2434         if (nvlist_alloc(&propput->ppu_alist, NV_UNIQUE_NAME_TYPE, 0) != 0) {
2435                 nvlist_free(propput->ppu_blist);
2436                 free(propput);
2437                 pool_seterror(POE_SYSTEM);
2438                 return (PO_FAIL);
2439         }
2440         if (nvlist_add_nvpair(propput->ppu_alist, ap) != 0) {
2441                 nvlist_free(propput->ppu_blist);
```



```
2442                 nvlist_free(propput->ppu_alist);
2443                 free(propput);
2444                 pool_seterror(POE_SYSTEM);
2445                 return (PO_FAIL);
2446         }

2448         if (nvlist_pack(propput->ppu_alist,
2449             (char **)&propput->ppu_ioctl.pp_o_buf,
2450             &propput->ppu_ioctl.pp_o_bufsize, NV_ENCODE_NATIVE, 0) != 0) {
2451                 pool_seterror(POE_SYSTEM);
2452                 return (PO_FAIL);
2453         }
2454         nvlist_free(propput->ppu_alist);
2455         propput->ppu_ioctl.pp_o_id_type = pool_elem_class(pe);
2456         if (pool_elem_class(pe) == PEC_RES_COMP ||
2457             pool_elem_class(pe) == PEC_RES_AGG)
2458                 propput->ppu_ioctl.pp_o_id_sub_type =
2459                     pool_resource_elem_class(pe);
2460         if (pool_elem_class(pe) == PEC_COMP)
2461                 propput->ppu_ioctl.pp_o_id_sub_type =
2462                     (pool_resource_elem_class_t)pool_component_elem_class(pe);

2464         propput->ppu_elem = pe;
2465         if ((prop = provider_get_prop(propput->ppu_elem, name)) != NULL) {
2466                 if (prop_is_readonly(prop) == PO_TRUE)
2467                         propput->ppu_doioctl |= KERNEL_PROP_RDONLY;
2468         }

2470         if (log_append(prov->pkc_log, POOL_PROPPUT, (void *)propput) !=
2471             PO_SUCCESS) {
2472                 nvlist_free(propput->ppu_blist);
2473                 free(propput);
2474                 return (PO_FAIL);
2475         }
2476         return (PO_SUCCESS);
2477 }
______unchanged_portion_omitted_

3401 /*
3402  * A log item stores state about the transaction it represents. This
3403  * function releases the resources associated with the transaction and
3404  * used to store the transaction state.
3405  */
3406 int
3407 log_item_release(log_item_t *li)
3408 {
3409         pool_create_undo_t *create;
3410         pool_destroy_undo_t *destroy;
3411         pool_assoc_undo_t *assoc;
3412         pool_dissoc_undo_t *dissoc;
3413         pool_propput_undo_t *propput;
3414         pool_proprm_undo_t *proprm;
3415         pool_xtransfer_undo_t *xtransfer;

3417         switch (li->li_op) {
3418         case POOL_CREATE:
3419                 create = (pool_create_undo_t *)li->li_details;

3421                 free(create);
3422                 break;
3423         case POOL_DESTROY:
3424                 destroy = (pool_destroy_undo_t *)li->li_details;

3426 #ifdef DEBUG
3427                 dprintf("log_item_release: POOL_DESTROY\n");
3428 #endif  /* DEBUG */
```

```
3430                 if (li->li_state == LS_UNDO) {
3431 #ifdef DEBUG
3432                         pool_elem_dprintf(destroy->pdu_elem);
3433 #endif  /* DEBUG */
3434                         pool_knl_elem_free((pool_knl_elem_t *)destroy->
3435                             pdu_elem, PO_TRUE);
3436                 }
3437                 free(destroy);
3438                 break;
3439         case POOL_ASSOC:
3440                 assoc = (pool_assoc_undo_t *)li->li_details;

3442                 free(assoc);
3443                 break;
3444         case POOL_DISSOC:
3445                 dissoc = (pool_dissoc_undo_t *)li->li_details;

3447                 free(dissoc);
3448                 break;
3449         case POOL_TRANSFER:
3450                 pool_seterror(POE_BADPARAM);
3451                 return (PO_FAIL);
3452         case POOL_XTRANSFER:
3453                 xtransfer = (pool_xtransfer_undo_t *)li->li_details;

3455                 free(xtransfer->pxu_rl);
3456                 free(xtransfer->pxu_ioctl.px_o_comp_list);
3457                 free(xtransfer);
3458                 break;
3459         case POOL_PROPPUT:
3460                 propput = (pool_propput_undo_t *)li->li_details;

3463                 if (propput->ppu_blist)
3462                 nvlist_free(propput->ppu_blist);
3463                 free(propput->ppu_ioctl.pp_o_buf);
3464                 free(propput);
3465                 break;
3466         case POOL_PROPRM:
3467                 proprm = (pool_proprm_undo_t *)li->li_details;

3469                 free(proprm);
3470                 break;
3471         default:
3472                 return (PO_FAIL);
3473         }
3474         return (PO_SUCCESS);
3475 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   33689 Mon Feb 15 12:56:09 2016
new/usr/src/lib/librcm/librcm.c
patch cleanup
6659 nvlist_free(NULL) is a no-op
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 #pragma ident   "%Z%%M% %I%     %E% SMI"

  26 #include "librcm_impl.h"
  27 #include "librcm_event.h"

  29 #ifdef  DEBUG
  30 static int rcm_debug = 1;
  31 #define dprintf(args) if (rcm_debug) (void) fprintf args
  32 #else
  33 #define dprintf(args) /* nothing */
  34 #endif  /* DEBUG */

  36 static int extract_info(nvlist_t *, rcm_info_t **);
  37 static int rcm_daemon_is_alive();
  38 static int rcm_common(int, rcm_handle_t *, char **, uint_t, void *,
  39     rcm_info_t **);
  40 static int rcm_direct_call(int, rcm_handle_t *, char **, uint_t, void *,
  41     rcm_info_t **);
  42 static int rcm_daemon_call(int, rcm_handle_t *, char **, uint_t, void *,
  43     rcm_info_t **);
  44 static int rcm_generate_nvlist(int, rcm_handle_t *, char **, uint_t, void *,
  45     char **, size_t *);
  46 static int rcm_check_permission(void);

  48 /*
  49  * Allocate a handle structure
  50  */
  51 /*ARGSUSED2*/
  52 int
  53 rcm_alloc_handle(char *modname, uint_t flag, void *arg, rcm_handle_t **hdp)
  54 {
  55         rcm_handle_t *hd;
  56         void *temp;
  57         char namebuf[MAXPATHLEN];
```



```
  59         if ((hdp == NULL) || (flag & ~RCM_ALLOC_HDL_MASK)) {
  60                 errno = EINVAL;
  61                 return (RCM_FAILURE);
  62         }

  64         if (rcm_check_permission() == 0) {
  65                 errno = EPERM;
  66                 return (RCM_FAILURE);
  67         }

  69         if ((hd = calloc(1, sizeof (*hd))) == NULL) {
  70                 return (RCM_FAILURE);
  71         }

  73         if (modname) {
  74                 (void) snprintf(namebuf, MAXPATHLEN, "%s%s", modname,
  75                         RCM_MODULE_SUFFIX);

  77                 if ((hd->modname = strdup(namebuf)) == NULL) {
  78                         free(hd);
  79                         return (RCM_FAILURE);
  80                 }

  82                 if ((temp = rcm_module_open(namebuf)) == NULL) {
  83                         free(hd->modname);
  84                         free(hd);
  85                         errno = EINVAL;
  86                         return (RCM_FAILURE);
  87                 }

  89                 rcm_module_close(temp);
  90         }

  92         if (flag & RCM_NOPID) {
  93                 hd->pid = (pid_t)0;
  94         } else {
  95                 hd->pid = (pid_t)getpid();
  96         }

  98         *hdp = hd;
  99         return (RCM_SUCCESS);
 100 }
______unchanged_portion_omitted_

 535 /*
 536  * RCM helper functions exposed to librcm callers.
 537  */

 539 /* Free linked list of registration info */
 540 void
 541 rcm_free_info(rcm_info_t *info)
 542 {
 543         while (info) {
 544                 rcm_info_t *tmp = info->next;

 548                 if (info->info)
 546                 nvlist_free(info->info);
 547                 free(info);

 549                 info = tmp;
 550         }
 551 }
______unchanged_portion_omitted_

1116 /*
1117  * Call into rcm_daemon door to process the request
```

```
 */
static int
rcm_daemon_call(int cmd, rcm_handle_t *hd, char **rsrcnames, uint_t flag,
    void *arg, rcm_info_t **infop)
{
        int errno_found;
        int daemon_errno = 0;
        int error = RCM_SUCCESS;
        int delay = 300;
        int maxdelay = 10000;   /* 10 seconds */
        char *nvl_packed = NULL;
        size_t nvl_size = 0;
        nvlist_t *ret = NULL;
        nvpair_t *nvp;
        size_t rsize = 0;
        rcm_info_t *info = NULL;

        errno = 0;

        /*
         * Decide whether to start the daemon
         */
        switch (cmd) {
        case CMD_GETINFO:
        case CMD_OFFLINE:
        case CMD_ONLINE:
        case CMD_REMOVE:
        case CMD_SUSPEND:
        case CMD_RESUME:
        case CMD_REGISTER:
        case CMD_UNREGISTER:
        case CMD_EVENT:
        case CMD_REQUEST_CHANGE:
        case CMD_NOTIFY_CHANGE:
        case CMD_GETSTATE:
                break;

        default:
                errno = EFAULT;
                return (RCM_FAILURE);
        }

        if (rcm_daemon_is_alive() != 1) {
                dprintf((stderr, "failed to start rcm_daemon\n"));
                errno = EFAULT;
                return (RCM_FAILURE);
        }

        /*
         * Generate a packed nvlist for the request
         */
        if (rcm_generate_nvlist(cmd, hd, rsrcnames, flag, arg, &nvl_packed,
            &nvl_size) < 0) {
                dprintf((stderr, "error in nvlist generation\n"));
                errno = EFAULT;
                return (RCM_FAILURE);
        }

        /*
         * Make the door call and get a return event. We go into a retry loop
         * when RCM_ET_EAGAIN is returned.
         */
retry:
        if (get_event_service(RCM_SERVICE_DOOR, (void *)nvl_packed, nvl_size,
            (void **)&ret, &rsize) < 0) {
                dprintf((stderr, "rcm_daemon call failed: %s\n",
```



```
                    strerror(errno)));
                free(nvl_packed);
                return (RCM_FAILURE);
        }

        assert(ret != NULL);

        /*
         * nvlist_lookup_* routines don't work because the returned nvlist
         * was nvlist_alloc'ed without the NV_UNIQUE_NAME flag.  Implement
         * a sequential search manually, which is fine since there is only
         * one RCM_RESULT value in the nvlist.
         */
        errno_found = 0;
        nvp = NULL;
        while (nvp = nvlist_next_nvpair(ret, nvp)) {
                if (strcmp(nvpair_name(nvp), RCM_RESULT) == 0) {
                        if (errno = nvpair_value_int32(nvp, &daemon_errno)) {
                                error = RCM_FAILURE;
                                goto out;
                        }
                        errno_found++;
                        break;
                }
        }
        if (errno_found == 0) {
                errno = EFAULT;
                error = RCM_FAILURE;
                goto out;
        }

        if (daemon_errno == EAGAIN) {
                /*
                 * Wait and retry
                 */
                dprintf((stderr, "retry door_call\n"));

                if (delay > maxdelay) {
                        errno = EAGAIN;
                        error = RCM_FAILURE;
                        goto out;
                }

                (void) poll(NULL, 0, delay);
                delay *= 2;             /* exponential back off */
                nvlist_free(ret);
                goto retry;
        }

        /*
         * The door call succeeded. Now extract info from returned event.
         */
        if (extract_info(ret, &info) != 0) {
                dprintf((stderr, "error in extracting event data\n"));
                errno = EFAULT;
                error = RCM_FAILURE;
                goto out;
        }

        if (infop)
                *infop = info;
        else
                rcm_free_info(info);

        if (daemon_errno) {
                if (daemon_errno > 0) {
```

```
1250                        errno = daemon_errno;
1251                        error = RCM_FAILURE;
1252                } else {
1253                        error = daemon_errno;
1254                }
1255        }

1257 out:
1258        if (nvl_packed)
1259                free(nvl_packed);
1263        if (ret)
1260        nvlist_free(ret);
1261        dprintf((stderr, "daemon call is done. error = %d, errno = %s\n", error,
1262            strerror(errno)));
1263        return (error);
1264 }
______unchanged_portion_omitted_

1324 /* Generate a packed nvlist for communicating with RCM daemon */
1325 static int
1326 rcm_generate_nvlist(int cmd, rcm_handle_t *hd, char **rsrcnames, uint_t flag,
1327     void *arg, char **nvl_packed, size_t *nvl_size)
1328 {
1329        int nrsrcnames;
1330        char *buf = NULL;
1331        size_t buflen = 0;
1332        nvlist_t *nvl = NULL;

1334        assert((nvl_packed != NULL) && (nvl_size != NULL));

1336        *nvl_size = 0;
1337        *nvl_packed = NULL;

1339        /* Allocate an empty nvlist */
1340        if ((errno = nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0)) > 0) {
1341                dprintf((stderr, "failed (nvlist_alloc=%s).\n",
1342                    strerror(errno)));
1343                return (-1);
1344        }

1346        /* Stuff in all the arguments for the daemon call */
1347        if (nvlist_add_int32(nvl, RCM_CMD, cmd) != 0) {
1348                dprintf((stderr, "failed (nvlist_add(CMD)=%s).\n",
1349                    strerror(errno)));
1350                goto fail;
1351        }
1352        if (rsrcnames) {
1353                nrsrcnames = 0;
1354                while (rsrcnames[nrsrcnames] != NULL)
1355                        nrsrcnames++;
1356                if (nvlist_add_string_array(nvl, RCM_RSRCNAMES, rsrcnames,
1357                    nrsrcnames) != 0) {
1358                        dprintf((stderr, "failed (nvlist_add(RSRCNAMES)=%s).\n",
1359                            strerror(errno)));
1360                        goto fail;
1361                }
1362        }
1363        if (hd->modname) {
1364                if (nvlist_add_string(nvl, RCM_CLIENT_MODNAME, hd->modname)
1365                    != 0) {
1366                        dprintf((stderr,
1367                            "failed (nvlist_add(CLIENT_MODNAME)=%s).\n",
1368                            strerror(errno)));
1369                        goto fail;
1370                }
1371        }
```



```
1372        if (hd->pid) {
1373                if (nvlist_add_uint64(nvl, RCM_CLIENT_ID, hd->pid) != 0) {
1374                        dprintf((stderr, "failed (nvlist_add(CLIENT_ID)=%s).\n",
1375                            strerror(errno)));
1376                        goto fail;
1377                }
1378        }
1379        if (flag) {
1380                if (nvlist_add_uint32(nvl, RCM_REQUEST_FLAG, flag) != 0) {
1381                        dprintf((stderr,
1382                            "failed (nvlist_add(REQUEST_FLAG)=%s).\n",
1383                            strerror(errno)));
1384                        goto fail;
1385                }
1386        }
1387        if (arg && cmd == CMD_SUSPEND) {
1388                if (nvlist_add_byte_array(nvl, RCM_SUSPEND_INTERVAL,
1389                    (uchar_t *)arg, sizeof (timespec_t)) != 0) {
1390                        dprintf((stderr,
1391                            "failed (nvlist_add(SUSPEND_INTERVAL)=%s).\n",
1392                            strerror(errno)));
1393                        goto fail;
1394                }
1395        }
1396        if (arg &&
1397            ((cmd == CMD_REQUEST_CHANGE) || (cmd == CMD_NOTIFY_CHANGE))) {
1398                if (errno = nvlist_pack(arg, &buf, &buflen, NV_ENCODE_NATIVE,
1399                    0)) {
1400                        dprintf((stderr,
1401                            "failed (nvlist_pack(CHANGE_DATA)=%s).\n",
1402                            strerror(errno)));
1403                        goto fail;
1404                }
1405                if (nvlist_add_byte_array(nvl, RCM_CHANGE_DATA, (uchar_t *)buf,
1406                    buflen) != 0) {
1407                        dprintf((stderr,
1408                            "failed (nvlist_add(CHANGE_DATA)=%s).\n",
1409                            strerror(errno)));
1410                        goto fail;
1411                }
1412        }
1413        if (arg && cmd == CMD_EVENT) {
1414                if (errno = nvlist_pack(arg, &buf, &buflen, NV_ENCODE_NATIVE,
1415                    0)) {
1416                        dprintf((stderr,
1417                            "failed (nvlist_pack(CHANGE_DATA)=%s).\n",
1418                            strerror(errno)));
1419                        goto fail;
1420                }
1421                if (nvlist_add_byte_array(nvl, RCM_EVENT_DATA, (uchar_t *)buf,
1422                    buflen) != 0) {
1423                        dprintf((stderr,
1424                            "failed (nvlist_add(EVENT_DATA)=%s).\n",
1425                            strerror(errno)));
1426                        goto fail;
1427                }
1428        }

1430        /* Pack the nvlist */
1431        if (errno = nvlist_pack(nvl, nvl_packed, nvl_size, NV_ENCODE_NATIVE,
1432            0)) {
1433                dprintf((stderr, "failed (nvlist_pack=%s).\n",
1434                    strerror(errno)));
1435                goto fail;
1436        }
```

```
1438         /* If an argument was packed intermediately, free the buffer */
1439         if (buf)
1440                 free(buf);

1442         /* Free the unpacked version of the nvlist and return the packed list */
1443         nvlist_free(nvl);
1444         return (0);

1446 fail:
1447         if (buf)
1448                 free(buf);
1453         if (nvl)
1449         nvlist_free(nvl);
1450         if (*nvl_packed)
1451                 free(*nvl_packed);
1452         *nvl_packed = NULL;
1453         *nvl_size = 0;
1454         return (-1);
1455 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   44946 Mon Feb 15 12:56:09 2016
new/usr/src/lib/libscf/common/notify_params.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

1687 /*
1688  * Specialized function to get fma notification parameters
1689  *
1690  * return SCF_SUCCESS or SCF_FAILED on
1691  *      SCF_ERROR_BACKEND_ACCESS
1692  *      SCF_ERROR_CONNECTION_BROKEN
1693  *      SCF_ERROR_DELETED
1694  *      SCF_ERROR_INTERNAL
1695  *      SCF_ERROR_INVALID_ARGUMENT
1696  *      SCF_ERROR_NO_MEMORY
1697  *      SCF_ERROR_NO_RESOURCES
1698  *      SCF_ERROR_NOT_FOUND
1699  *      SCF_ERROR_PERMISSION_DENIED
1700  */
1701 int
1702 _scf_get_fma_notify_params(const char *class, nvlist_t *nvl, int getsource)
1703 {
1704         scf_handle_t            *h = _scf_handle_create_and_bind(SCF_VERSION);
1705         scf_error_t             scf_e = scf_error();
1706         scf_instance_t          *i = scf_instance_create(h);
1707         scf_propertygroup_t     *pg = scf_pg_create(h);
1708         int r = SCF_FAILED;
1709         nvlist_t *params = NULL;
1710         char *pgname = NULL;

1712         if (h == NULL) {
1713                 /*
1714                  * use saved error if _scf_handle_create_and_bind() fails
1715                  */
1716                 (void) scf_set_error(scf_e);
1717                 goto cleanup;
1718         }
1719         if (i == NULL || pg == NULL)
1720                 goto cleanup;

1722         if (scf_handle_decode_fmri(h, SCF_NOTIFY_PARAMS_INST, NULL, NULL, i,
1723             NULL, NULL, SCF_DECODE_FMRI_EXACT) != SCF_SUCCESS) {
1724                 if (check_scf_error(scf_error(), errs_1)) {
1725                         goto cleanup;
1726                 }
1727         }

1729         if ((pgname = class_to_pgname(class)) == NULL)
1730                 goto cleanup;

1732         while (get_pg(NULL, i, pgname, pg, 0) != 0) {
1733                 if (scf_error() == SCF_ERROR_NOT_FOUND) {
1734                         char *p = strrchr(pgname, '.');

1736                         if (p != NULL) {
1737                                 *p = ',';
1738                                 /*
1739                                  * since the resulting string is shorter,
1740                                  * there is no risk of buffer overflow
1741                                  */
1742                                 (void) strcpy(p + 1, SCF_NOTIFY_PG_POSTFIX);
1743                                 continue;
1744                         }
1745                 }
```



```
1747                 if (check_scf_error(scf_error(), errs_1)) {
1748                         goto cleanup;
1749                 }
1750         }

1752         if (nvlist_alloc(&params, NV_UNIQUE_NAME, 0) != 0) {
1753                 (void) scf_set_error(SCF_ERROR_NO_MEMORY);
1754                 goto cleanup;
1755         }

1757         if (_scf_notify_get_params(pg, params) != SCF_SUCCESS)
1758                 goto cleanup;

1760         if (getsource && get_pg_source(pg, params) != SCF_SUCCESS)
1761                 goto cleanup;

1763         if (nvlist_add_nvlist_array(nvl, SCF_NOTIFY_PARAMS, &params, 1) != 0 ||
1764             nvlist_add_uint32(nvl, SCF_NOTIFY_NAME_VERSION,
1765             SCF_NOTIFY_PARAMS_VERSION) != 0) {
1766                 (void) scf_set_error(SCF_ERROR_NO_MEMORY);
1767                 goto cleanup;
1768         }

1770         r = SCF_SUCCESS;

1772 cleanup:
1773         if (params)
1773         nvlist_free(params);
1774         scf_pg_destroy(pg);
1775         scf_instance_destroy(i);
1776         scf_handle_destroy(h);
1777         free(pgname);

1779         return (r);
1780 }
______unchanged_portion_omitted_
```

```
**********************************************************
   61827 Mon Feb 15 12:56:09 2016
new/usr/src/lib/libsysevent/libsysevent.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 213 /*
 214  * sysevent_free - Free memory allocated for an event buffer
 215  */
 216 void
 217 sysevent_free(sysevent_t *ev)
 218 {
 219         nvlist_t *attr_list = (nvlist_t *)(uintptr_t)SE_ATTR_PTR(ev);

 221         if (attr_list)
 221         nvlist_free(attr_list);
 222         free(ev);
 223 }
______unchanged_portion_omitted_

1348 static int
1349 create_cached_registration(sysevent_handle_t *shp,
1350         class_lst_t **class_hash)
1351 {
1352         int i, j, new_class;
1353         char *class_name;
1354         uint_t num_elem;
1355         uchar_t *subscribers;
1356         nvlist_t *nvl;
1357         nvpair_t *nvpair;
1358         class_lst_t *clist;
1359         subclass_lst_t *sc_list;

1361         for (i = 0; i < CLASS_HASH_SZ + 1; ++i) {

1363                 if ((nvl = get_kernel_registration(SH_CHANNEL_NAME(shp), i))
1364                     == NULL) {
1365                         if (errno == ENOENT) {
1366                                 class_hash[i] = NULL;
1367                                 continue;
1368                         } else {
1369                                 goto create_failed;
1370                         }
1371                 }


1374                 nvpair = NULL;
1375                 if ((nvpair = nvlist_next_nvpair(nvl, nvpair)) == NULL) {
1376                         goto create_failed;
1377                 }

1379                 new_class = 1;
1380                 while (new_class) {
1381                         /* Extract the class name from the nvpair */
1382                         if (nvpair_value_string(nvpair, &class_name) != 0) {
1383                                 goto create_failed;
1384                         }
1385                         clist = (class_lst_t *)
1386                             calloc(1, sizeof (class_lst_t));
1387                         if (clist == NULL) {
1388                                 goto create_failed;
1389                         }

1391                         clist->cl_name = strdup(class_name);
1392                         if (clist->cl_name == NULL) {
```



```
1393                                 free(clist);
1394                                 goto create_failed;
1395                         }

1397                         /*
1398                          * Extract the subclass name and registration
1399                          * from the nvpair
1400                          */
1401                         if ((nvpair = nvlist_next_nvpair(nvl, nvpair))
1402                             == NULL) {
1403                                 free(clist->cl_name);
1404                                 free(clist);
1405                                 goto create_failed;
1406                         }

1408                         clist->cl_next = class_hash[i];
1409                         class_hash[i] = clist;

1411                         for (;;) {

1413                                 sc_list = (subclass_lst_t *)calloc(1,
1414                                     sizeof (subclass_lst_t));
1415                                 if (sc_list == NULL) {
1416                                         goto create_failed;
1417                                 }

1419                                 sc_list->sl_next = clist->cl_subclass_list;
1420                                 clist->cl_subclass_list = sc_list;

1422                                 sc_list->sl_name = strdup(nvpair_name(nvpair));
1423                                 if (sc_list->sl_name == NULL) {
1424                                         goto create_failed;
1425                                 }

1427                                 if (nvpair_value_byte_array(nvpair,
1428                                     &subscribers, &num_elem) != 0) {
1429                                         goto create_failed;
1430                                 }
1431                                 bcopy(subscribers, (uchar_t *)sc_list->sl_num,
1432                                     MAX_SUBSCRIBERS + 1);

1434                                 for (j = 1; j <= MAX_SUBSCRIBERS; ++j) {
1435                                         if (sc_list->sl_num[j] == 0)
1436                                                 continue;

1438                                         if (alloc_subscriber(shp, j, 1) != 0) {
1439                                                 goto create_failed;
1440                                         }
1441                                 }

1443                                 /*
1444                                  * Check next nvpair - either subclass or
1445                                  * class
1446                                  */
1447                                 if ((nvpair = nvlist_next_nvpair(nvl, nvpair))
1448                                     == NULL) {
1449                                         new_class = 0;
1450                                         break;
1451                                 } else if (strcmp(nvpair_name(nvpair),
1452                                     CLASS_NAME) == 0) {
1453                                         break;
1454                                 }
1455                         }
1456                 }
1457                 nvlist_free(nvl);
1458         }
```

```
1459         return (0);

1461 create_failed:
1462         dealloc_subscribers(shp);
1463         free_cached_registration(shp);
1465         if (nvl)
1464         nvlist_free(nvl);
1465         return (-1);

1467 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10291 Mon Feb 15 12:56:09 2016
new/usr/src/lib/libzfs/common/libzfs_config.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 257 /*
 258  * Refresh the vdev statistics associated with the given pool.  This is used in
 259  * iostat to show configuration changes and determine the delta from the last
 260  * time the function was called.  This function can fail, in case the pool has
 261  * been destroyed.
 262  */
 263 int
 264 zpool_refresh_stats(zpool_handle_t *zhp, boolean_t *missing)
 265 {
 266         zfs_cmd_t zc = { 0 };
 267         int error;
 268         nvlist_t *config;
 269         libzfs_handle_t *hdl = zhp->zpool_hdl;

 271         *missing = B_FALSE;
 272         (void) strcpy(zc.zc_name, zhp->zpool_name);

 274         if (zhp->zpool_config_size == 0)
 275                 zhp->zpool_config_size = 1 << 16;

 277         if (zcmd_alloc_dst_nvlist(hdl, &zc, zhp->zpool_config_size) != 0)
 278                 return (-1);

 280         for (;;) {
 281                 if (ioctl(zhp->zpool_hdl->libzfs_fd, ZFS_IOC_POOL_STATS,
 282                     &zc) == 0) {
 283                         /*
 284                          * The real error is returned in the zc_cookie field.
 285                          */
 286                         error = zc.zc_cookie;
 287                         break;
 288                 }

 290                 if (errno == ENOMEM) {
 291                         if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
 292                                 zcmd_free_nvlists(&zc);
 293                                 return (-1);
 294                         }
 295                 } else {
 296                         zcmd_free_nvlists(&zc);
 297                         if (errno == ENOENT || errno == EINVAL)
 298                                 *missing = B_TRUE;
 299                         zhp->zpool_state = POOL_STATE_UNAVAIL;
 300                         return (0);
 301                 }
 302         }

 304         if (zcmd_read_dst_nvlist(hdl, &zc, &config) != 0) {
 305                 zcmd_free_nvlists(&zc);
 306                 return (-1);
 307         }

 309         zcmd_free_nvlists(&zc);

 311         zhp->zpool_config_size = zc.zc_nvlist_dst_size;

 313         if (zhp->zpool_config != NULL) {
 314                 uint64_t oldtxg, newtxg;
```



```
 316                 verify(nvlist_lookup_uint64(zhp->zpool_config,
 317                     ZPOOL_CONFIG_POOL_TXG, &oldtxg) == 0);
 318                 verify(nvlist_lookup_uint64(config,
 319                     ZPOOL_CONFIG_POOL_TXG, &newtxg) == 0);

 321                 if (zhp->zpool_old_config != NULL)
 321                 nvlist_free(zhp->zpool_old_config);

 323                 if (oldtxg != newtxg) {
 324                         nvlist_free(zhp->zpool_config);
 325                         zhp->zpool_old_config = NULL;
 326                 } else {
 327                         zhp->zpool_old_config = zhp->zpool_config;
 328                 }
 329         }

 331         zhp->zpool_config = config;
 332         if (error)
 333                 zhp->zpool_state = POOL_STATE_UNAVAIL;
 334         else
 335                 zhp->zpool_state = POOL_STATE_ACTIVE;

 337         return (0);
 338 }
______unchanged_portion_omitted_
```

```
**********************************************************
  118243 Mon Feb 15 12:56:10 2016
new/usr/src/lib/libzfs/common/libzfs_dataset.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

1861 /*
1862  * Internal function for getting a numeric property.  Both zfs_prop_get() and
1863  * zfs_prop_get_int() are built using this interface.
1864  *
1865  * Certain properties can be overridden using 'mount -o'.  In this case, scan
1866  * the contents of the /etc/mnttab entry, searching for the appropriate options.
1867  * If they differ from the on-disk values, report the current values and mark
1868  * the source "temporary".
1869  */
1870 static int
1871 get_numeric_property(zfs_handle_t *zhp, zfs_prop_t prop, zprop_source_t *src,
1872     char **source, uint64_t *val)
1873 {
1874         zfs_cmd_t zc = { 0 };
1875         nvlist_t *zplprops = NULL;
1876         struct mnttab mnt;
1877         char *mntopt_on = NULL;
1878         char *mntopt_off = NULL;
1879         boolean_t received = zfs_is_recvd_props_mode(zhp);

1881         *source = NULL;

1883         switch (prop) {
1884         case ZFS_PROP_ATIME:
1885                 mntopt_on = MNTOPT_ATIME;
1886                 mntopt_off = MNTOPT_NOATIME;
1887                 break;

1889         case ZFS_PROP_DEVICES:
1890                 mntopt_on = MNTOPT_DEVICES;
1891                 mntopt_off = MNTOPT_NODEVICES;
1892                 break;

1894         case ZFS_PROP_EXEC:
1895                 mntopt_on = MNTOPT_EXEC;
1896                 mntopt_off = MNTOPT_NOEXEC;
1897                 break;

1899         case ZFS_PROP_READONLY:
1900                 mntopt_on = MNTOPT_RO;
1901                 mntopt_off = MNTOPT_RW;
1902                 break;

1904         case ZFS_PROP_SETUID:
1905                 mntopt_on = MNTOPT_SETUID;
1906                 mntopt_off = MNTOPT_NOSETUID;
1907                 break;

1909         case ZFS_PROP_XATTR:
1910                 mntopt_on = MNTOPT_XATTR;
1911                 mntopt_off = MNTOPT_NOXATTR;
1912                 break;

1914         case ZFS_PROP_NBMAND:
1915                 mntopt_on = MNTOPT_NBMAND;
1916                 mntopt_off = MNTOPT_NONBMAND;
1917                 break;
1918         }
```



```
1920         /*
1921          * Because looking up the mount options is potentially expensive
1922          * (iterating over all of /etc/mnttab), we defer its calculation until
1923          * we're looking up a property which requires its presence.
1924          */
1925         if (!zhp->zfs_mntcheck &&
1926             (mntopt_on != NULL || prop == ZFS_PROP_MOUNTED)) {
1927                 libzfs_handle_t *hdl = zhp->zfs_hdl;
1928                 struct mnttab entry;

1930                 if (libzfs_mnttab_find(hdl, zhp->zfs_name, &entry) == 0) {
1931                         zhp->zfs_mntopts = zfs_strdup(hdl,
1932                             entry.mnt_mntopts);
1933                         if (zhp->zfs_mntopts == NULL)
1934                                 return (-1);
1935                 }

1937                 zhp->zfs_mntcheck = B_TRUE;
1938         }

1940         if (zhp->zfs_mntopts == NULL)
1941                 mnt.mnt_mntopts = "";
1942         else
1943                 mnt.mnt_mntopts = zhp->zfs_mntopts;

1945         switch (prop) {
1946         case ZFS_PROP_ATIME:
1947         case ZFS_PROP_DEVICES:
1948         case ZFS_PROP_EXEC:
1949         case ZFS_PROP_READONLY:
1950         case ZFS_PROP_SETUID:
1951         case ZFS_PROP_XATTR:
1952         case ZFS_PROP_NBMAND:
1953                 *val = getprop_uint64(zhp, prop, source);

1955                 if (received)
1956                         break;

1958                 if (hasmntopt(&mnt, mntopt_on) && !*val) {
1959                         *val = B_TRUE;
1960                         if (src)
1961                                 *src = ZPROP_SRC_TEMPORARY;
1962                 } else if (hasmntopt(&mnt, mntopt_off) && *val) {
1963                         *val = B_FALSE;
1964                         if (src)
1965                                 *src = ZPROP_SRC_TEMPORARY;
1966                 }
1967                 break;

1969         case ZFS_PROP_CANMOUNT:
1970         case ZFS_PROP_VOLSIZE:
1971         case ZFS_PROP_QUOTA:
1972         case ZFS_PROP_REFQUOTA:
1973         case ZFS_PROP_RESERVATION:
1974         case ZFS_PROP_REFRESERVATION:
1975         case ZFS_PROP_FILESYSTEM_LIMIT:
1976         case ZFS_PROP_SNAPSHOT_LIMIT:
1977         case ZFS_PROP_FILESYSTEM_COUNT:
1978         case ZFS_PROP_SNAPSHOT_COUNT:
1979                 *val = getprop_uint64(zhp, prop, source);

1981                 if (*source == NULL) {
1982                         /* not default, must be local */
1983                         *source = zhp->zfs_name;
1984                 }
1985                 break;
```

```
1987         case ZFS_PROP_MOUNTED:
1988                 *val = (zhp->zfs_mntopts != NULL);
1989                 break;

1991         case ZFS_PROP_NUMCLONES:
1992                 *val = zhp->zfs_dmustats.dds_num_clones;
1993                 break;

1995         case ZFS_PROP_VERSION:
1996         case ZFS_PROP_NORMALIZE:
1997         case ZFS_PROP_UTF8ONLY:
1998         case ZFS_PROP_CASE:
1999                 if (!zfs_prop_valid_for_type(prop, zhp->zfs_head_type) ||
2000                     zcmd_alloc_dst_nvlist(zhp->zfs_hdl, &zc, 0) != 0)
2001                         return (-1);
2002                 (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
2003                 if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_OBJSET_ZPLPROPS, &zc)) {
2004                         zcmd_free_nvlists(&zc);
2005                         return (-1);
2006                 }
2007                 if (zcmd_read_dst_nvlist(zhp->zfs_hdl, &zc, &zplprops) != 0 ||
2008                     nvlist_lookup_uint64(zplprops, zfs_prop_to_name(prop),
2009                     val) != 0) {
2010                         zcmd_free_nvlists(&zc);
2011                         return (-1);
2012                 }
2013                 if (zplprops)
2013                 nvlist_free(zplprops);
2014                 zcmd_free_nvlists(&zc);
2015                 break;

2017         case ZFS_PROP_INCONSISTENT:
2018                 *val = zhp->zfs_dmustats.dds_inconsistent;
2019                 break;

2021         default:
2022                 switch (zfs_prop_get_type(prop)) {
2023                 case PROP_TYPE_NUMBER:
2024                 case PROP_TYPE_INDEX:
2025                         *val = getprop_uint64(zhp, prop, source);
2026                         /*
2027                          * If we tried to use a default value for a
2028                          * readonly property, it means that it was not
2029                          * present.
2030                          */
2031                         if (zfs_prop_readonly(prop) &&
2032                             *source != NULL && (*source)[0] == '\0') {
2033                                 *source = NULL;
2034                         }
2035                         break;

2037                 case PROP_TYPE_STRING:
2038                 default:
2039                         zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
2040                             "cannot get non-numeric property"));
2041                         return (zfs_error(zhp->zfs_hdl, EZFS_BADPROP,
2042                             dgettext(TEXT_DOMAIN, "internal error")));
2043                 }
2044         }

2046         return (0);
2047 }
```

*unchanged_portion_omitted_*

```
4198 static int
```



```
4199 zfs_smb_acl_mgmt(libzfs_handle_t *hdl, char *dataset, char *path,
4200     zfs_smb_acl_op_t cmd, char *resource1, char *resource2)
4201 {
4202         zfs_cmd_t zc = { 0 };
4203         nvlist_t *nvlist = NULL;
4204         int error;

4206         (void) strlcpy(zc.zc_name, dataset, sizeof (zc.zc_name));
4207         (void) strlcpy(zc.zc_value, path, sizeof (zc.zc_value));
4208         zc.zc_cookie = (uint64_t)cmd;

4210         if (cmd == ZFS_SMB_ACL_RENAME) {
4211                 if (nvlist_alloc(&nvlist, NV_UNIQUE_NAME, 0) != 0) {
4212                         (void) no_memory(hdl);
4213                         return (0);
4214                 }
4215         }

4217         switch (cmd) {
4218         case ZFS_SMB_ACL_ADD:
4219         case ZFS_SMB_ACL_REMOVE:
4220                 (void) strlcpy(zc.zc_string, resource1, sizeof (zc.zc_string));
4221                 break;
4222         case ZFS_SMB_ACL_RENAME:
4223                 if (nvlist_add_string(nvlist, ZFS_SMB_ACL_SRC,
4224                     resource1) != 0) {
4225                                 (void) no_memory(hdl);
4226                                 return (-1);
4227                 }
4228                 if (nvlist_add_string(nvlist, ZFS_SMB_ACL_TARGET,
4229                     resource2) != 0) {
4230                                 (void) no_memory(hdl);
4231                                 return (-1);
4232                 }
4233                 if (zcmd_write_src_nvlist(hdl, &zc, nvlist) != 0) {
4234                         nvlist_free(nvlist);
4235                         return (-1);
4236                 }
4237                 break;
4238         case ZFS_SMB_ACL_PURGE:
4239                 break;
4240         default:
4241                 return (-1);
4242         }
4243         error = ioctl(hdl->libzfs_fd, ZFS_IOC_SMB_ACL, &zc);
4245         if (nvlist)
4244         nvlist_free(nvlist);
4245         return (error);
4246 }
```

*unchanged_portion_omitted_*

****
41521 Mon Feb 15 12:56:10 2016
new/usr/src/lib/libzfs/common/libzfs_import.c
6659 nvlist_free(NULL) is a no-op
****

_____unchanged_portion_omitted_

```
1118 /*
1119  * Given a list of directories to search, find all pools stored on disk.  This
1120  * includes partial pools which are not available to import.  If no args are
1121  * given (argc is 0), then the default directory (/dev/dsk) is searched.
1122  * poolname or guid (but not both) are provided by the caller when trying
1123  * to import a specific pool.
1124  */
1125 static nvlist_t *
1126 zpool_find_import_impl(libzfs_handle_t *hdl, importargs_t *iarg)
1127 {
1128         int i, dirs = iarg->paths;
1129         struct dirent64 *dp;
1130         char path[MAXPATHLEN];
1131         char *end, **dir = iarg->path;
1132         size_t pathleft;
1133         nvlist_t *ret = NULL;
1134         static char *default_dir = "/dev/dsk";
1135         pool_list_t pools = { 0 };
1136         pool_entry_t *pe, *penext;
1137         vdev_entry_t *ve, *venext;
1138         config_entry_t *ce, *cenext;
1139         name_entry_t *ne, *nenext;
1140         avl_tree_t slice_cache;
1141         rdsk_node_t *slice;
1142         void *cookie;

1144         if (dirs == 0) {
1145                 dirs = 1;
1146                 dir = &default_dir;
1147         }

1149         /*
1150          * Go through and read the label configuration information from every
1151          * possible device, organizing the information according to pool GUID
1152          * and toplevel GUID.
1153          */
1154         for (i = 0; i < dirs; i++) {
1155                 tpool_t *t;
1156                 char *rdsk;
1157                 int dfd;
1158                 boolean_t config_failed = B_FALSE;
1159                 DIR *dirp;

1161                 /* use realpath to normalize the path */
1162                 if (realpath(dir[i], path) == 0) {
1163                         (void) zfs_error_fmt(hdl, EZFS_BADPATH,
1164                             dgettext(TEXT_DOMAIN, "cannot open '%s'"), dir[i]);
1165                         goto error;
1166                 }
1167                 end = &path[strlen(path)];
1168                 *end++ = '/';
1169                 *end = 0;
1170                 pathleft = &path[sizeof (path)] - end;

1172                 /*
1173                  * Using raw devices instead of block devices when we're
1174                  * reading the labels skips a bunch of slow operations during
1175                  * close(2) processing, so we replace /dev/dsk with /dev/rdsk.
1176                  */
```



```
1177                 if (strcmp(path, "/dev/dsk/") == 0)
1178                         rdsk = "/dev/rdsk/";
1179                 else
1180                         rdsk = path;

1182                 if ((dfd = open64(rdsk, O_RDONLY)) < 0 ||
1183                     (dirp = fdopendir(dfd)) == NULL) {
1184                         if (dfd >= 0)
1185                                 (void) close(dfd);
1186                         zfs_error_aux(hdl, strerror(errno));
1187                         (void) zfs_error_fmt(hdl, EZFS_BADPATH,
1188                             dgettext(TEXT_DOMAIN, "cannot open '%s'"),
1189                             rdsk);
1190                         goto error;
1191                 }

1193                 avl_create(&slice_cache, slice_cache_compare,
1194                     sizeof (rdsk_node_t), offsetof(rdsk_node_t, rn_node));
1195                 /*
1196                  * This is not MT-safe, but we have no MT consumers of libzfs
1197                  */
1198                 while ((dp = readdir64(dirp)) != NULL) {
1199                         const char *name = dp->d_name;
1200                         if (name[0] == '.' &&
1201                             (name[1] == 0 || (name[1] == '.' && name[2] == 0)))
1202                                 continue;

1204                         slice = zfs_alloc(hdl, sizeof (rdsk_node_t));
1205                         slice->rn_name = zfs_strdup(hdl, name);
1206                         slice->rn_avl = &slice_cache;
1207                         slice->rn_dfd = dfd;
1208                         slice->rn_hdl = hdl;
1209                         slice->rn_nozpool = B_FALSE;
1210                         avl_add(&slice_cache, slice);
1211                 }
1212                 /*
1213                  * create a thread pool to do all of this in parallel;
1214                  * rn_nozpool is not protected, so this is racy in that
1215                  * multiple tasks could decide that the same slice can
1216                  * not hold a zpool, which is benign.  Also choose
1217                  * double the number of processors; we hold a lot of
1218                  * locks in the kernel, so going beyond this doesn't
1219                  * buy us much.
1220                  */
1221                 t = tpool_create(1, 2 * sysconf(_SC_NPROCESSORS_ONLN),
1222                     0, NULL);
1223                 for (slice = avl_first(&slice_cache); slice;
1224                     (slice = avl_walk(&slice_cache, slice,
1225                     AVL_AFTER)))
1226                         (void) tpool_dispatch(t, zpool_open_func, slice);
1227                 tpool_wait(t);
1228                 tpool_destroy(t);

1230                 cookie = NULL;
1231                 while ((slice = avl_destroy_nodes(&slice_cache,
1232                     &cookie)) != NULL) {
1233                         if (slice->rn_config != NULL && !config_failed) {
1234                                 nvlist_t *config = slice->rn_config;
1235                                 boolean_t matched = B_TRUE;

1237                                 if (iarg->poolname != NULL) {
1238                                         char *pname;

1240                                         matched = nvlist_lookup_string(config,
1241                                             ZPOOL_CONFIG_POOL_NAME,
1242                                             &pname) == 0 &&
```

```
1243                                        strcmp(iarg->poolname, pname) == 0;
1244                                } else if (iarg->guid != 0) {
1245                                        uint64_t this_guid;

1247                                        matched = nvlist_lookup_uint64(config,
1248                                            ZPOOL_CONFIG_POOL_GUID,
1249                                            &this_guid) == 0 &&
1250                                            iarg->guid == this_guid;
1251                                }
1252                                if (!matched) {
1253                                        nvlist_free(config);
1254                                } else {
1255                                        /*
1256                                         * use the non-raw path for the config
1257                                         */
1258                                        (void) strlcpy(end, slice->rn_name,
1259                                            pathleft);
1260                                        if (add_config(hdl, &pools, path,
1261                                            config) != 0)
1262                                                config_failed = B_TRUE;
1263                                }
1264                        }
1265                        free(slice->rn_name);
1266                        free(slice);
1267                }
1268                avl_destroy(&slice_cache);

1270                (void) closedir(dirp);

1272                if (config_failed)
1273                        goto error;
1274        }

1276        ret = get_configs(hdl, &pools, iarg->can_be_active);

1278 error:
1279        for (pe = pools.pools; pe != NULL; pe = penext) {
1280                penext = pe->pe_next;
1281                for (ve = pe->pe_vdevs; ve != NULL; ve = venext) {
1282                        venext = ve->ve_next;
1283                        for (ce = ve->ve_configs; ce != NULL; ce = cenext) {
1284                                cenext = ce->ce_next;
1285                                if (ce->ce_config)
1285                                nvlist_free(ce->ce_config);
1286                                free(ce);
1287                        }
1288                        free(ve);
1289                }
1290                free(pe);
1291        }

1293        for (ne = pools.names; ne != NULL; ne = nenext) {
1294                nenext = ne->ne_next;
1295                free(ne->ne_name);
1296                free(ne);
1297        }

1299        return (ret);
1300 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  102319 Mon Feb 15 12:56:10 2016
new/usr/src/lib/libzfs/common/libzfs_pool.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

1022 /*
1023  * Close the handle.  Simply frees the memory associated with the handle.
1024  */
1025 void
1026 zpool_close(zpool_handle_t *zhp)
1027 {
1028         if (zhp->zpool_config)
1028         nvlist_free(zhp->zpool_config);
1030         if (zhp->zpool_old_config)
1029         nvlist_free(zhp->zpool_old_config);
1032         if (zhp->zpool_props)
1030         nvlist_free(zhp->zpool_props);
1031         free(zhp);
1032 }
______unchanged_portion_omitted_

1514 /*
1515  * zpool_import() is a contracted interface. Should be kept the same
1516  * if possible.
1517  *
1518  * Applications should use zpool_import_props() to import a pool with
1519  * new properties value to be set.
1520  */
1521 int
1522 zpool_import(libzfs_handle_t *hdl, nvlist_t *config, const char *newname,
1523     char *altroot)
1524 {
1525         nvlist_t *props = NULL;
1526         int ret;

1528         if (altroot != NULL) {
1529                 if (nvlist_alloc(&props, NV_UNIQUE_NAME, 0) != 0) {
1530                         return (zfs_error_fmt(hdl, EZFS_NOMEM,
1531                             dgettext(TEXT_DOMAIN, "cannot import '%s'"),
1532                             newname));
1533                 }

1535                 if (nvlist_add_string(props,
1536                     zpool_prop_to_name(ZPOOL_PROP_ALTROOT), altroot) != 0 ||
1537                     nvlist_add_string(props,
1538                     zpool_prop_to_name(ZPOOL_PROP_CACHEFILE), "none") != 0) {
1539                         nvlist_free(props);
1540                         return (zfs_error_fmt(hdl, EZFS_NOMEM,
1541                             dgettext(TEXT_DOMAIN, "cannot import '%s'"),
1542                             newname));
1543                 }
1544         }

1546         ret = zpool_import_props(hdl, config, newname, props,
1547             ZFS_IMPORT_NORMAL);
1551         if (props)
1548         nvlist_free(props);
1549         return (ret);
1550 }
______unchanged_portion_omitted_

2808 /*
2809  * Split a mirror pool.  If newroot points to null, then a new nvlist
2810  * is generated and it is the responsibility of the caller to free it.
```



```
2811  */
2812 int
2813 zpool_vdev_split(zpool_handle_t *zhp, char *newname, nvlist_t **newroot,
2814     nvlist_t *props, splitflags_t flags)
2815 {
2816         zfs_cmd_t zc = { 0 };
2817         char msg[1024];
2818         nvlist_t *tree, *config, **child, **newchild, *newconfig = NULL;
2819         nvlist_t **varray = NULL, *zc_props = NULL;
2820         uint_t c, children, newchildren, lastlog = 0, vcount, found = 0;
2821         libzfs_handle_t *hdl = zhp->zpool_hdl;
2822         uint64_t vers;
2823         boolean_t freelist = B_FALSE, memory_err = B_TRUE;
2824         int retval = 0;

2826         (void) snprintf(msg, sizeof (msg),
2827             dgettext(TEXT_DOMAIN, "Unable to split %s"), zhp->zpool_name);

2829         if (!zpool_name_valid(hdl, B_FALSE, newname))
2830                 return (zfs_error(hdl, EZFS_INVALIDNAME, msg));

2832         if ((config = zpool_get_config(zhp, NULL)) == NULL) {
2833                 (void) fprintf(stderr, gettext("Internal error: unable to "
2834                     "retrieve pool configuration\n"));
2835                 return (-1);
2836         }

2838         verify(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &tree)
2839             == 0);
2840         verify(nvlist_lookup_uint64(config, ZPOOL_CONFIG_VERSION, &vers) == 0);

2842         if (props) {
2843                 prop_flags_t flags = { .create = B_FALSE, .import = B_TRUE };
2844                 if ((zc_props = zpool_valid_proplist(hdl, zhp->zpool_name,
2845                     props, vers, flags, msg)) == NULL)
2846                         return (-1);
2847         }

2849         if (nvlist_lookup_nvlist_array(tree, ZPOOL_CONFIG_CHILDREN, &child,
2850             &children) != 0) {
2851                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2852                     "Source pool is missing vdev tree"));
2857                 if (zc_props)
2853                 nvlist_free(zc_props);
2854                 return (-1);
2855         }

2857         varray = zfs_alloc(hdl, children * sizeof (nvlist_t *));
2858         vcount = 0;

2860         if (*newroot == NULL ||
2861             nvlist_lookup_nvlist_array(*newroot, ZPOOL_CONFIG_CHILDREN,
2862             &newchild, &newchildren) != 0)
2863                 newchildren = 0;

2865         for (c = 0; c < children; c++) {
2866                 uint64_t is_log = B_FALSE, is_hole = B_FALSE;
2867                 char *type;
2868                 nvlist_t **mchild, *vdev;
2869                 uint_t mchildren;
2870                 int entry;

2872                 /*
2873                  * Unlike cache & spares, slogs are stored in the
2874                  * ZPOOL_CONFIG_CHILDREN array.  We filter them out here.
2875                  */
```

```
2876                (void) nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_IS_LOG,
2877                    &is_log);
2878                (void) nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_IS_HOLE,
2879                    &is_hole);
2880                if (is_log || is_hole) {
2881                        /*
2882                         * Create a hole vdev and put it in the config.
2883                         */
2884                        if (nvlist_alloc(&vdev, NV_UNIQUE_NAME, 0) != 0)
2885                                goto out;
2886                        if (nvlist_add_string(vdev, ZPOOL_CONFIG_TYPE,
2887                            VDEV_TYPE_HOLE) != 0)
2888                                goto out;
2889                        if (nvlist_add_uint64(vdev, ZPOOL_CONFIG_IS_HOLE,
2890                            1) != 0)
2891                                goto out;
2892                        if (lastlog == 0)
2893                                lastlog = vcount;
2894                        varray[vcount++] = vdev;
2895                        continue;
2896                }
2897                lastlog = 0;
2898                verify(nvlist_lookup_string(child[c], ZPOOL_CONFIG_TYPE, &type)
2899                    == 0);
2900                if (strcmp(type, VDEV_TYPE_MIRROR) != 0) {
2901                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2902                            "Source pool must be composed only of mirrors\n"));
2903                        retval = zfs_error(hdl, EZFS_INVALCONFIG, msg);
2904                        goto out;
2905                }

2907                verify(nvlist_lookup_nvlist_array(child[c],
2908                    ZPOOL_CONFIG_CHILDREN, &mchild, &mchildren) == 0);

2910                /* find or add an entry for this top-level vdev */
2911                if (newchildren > 0 &&
2912                    (entry = find_vdev_entry(zhp, mchild, mchildren,
2913                    newchild, newchildren)) >= 0) {
2914                        /* We found a disk that the user specified. */
2915                        vdev = mchild[entry];
2916                        ++found;
2917                } else {
2918                        /* User didn't specify a disk for this vdev. */
2919                        vdev = mchild[mchildren - 1];
2920                }

2922                if (nvlist_dup(vdev, &varray[vcount++], 0) != 0)
2923                        goto out;
2924        }

2926        /* did we find every disk the user specified? */
2927        if (found != newchildren) {
2928                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "Device list must "
2929                    "include at most one disk from each mirror"));
2930                retval = zfs_error(hdl, EZFS_INVALCONFIG, msg);
2931                goto out;
2932        }

2934        /* Prepare the nvlist for populating. */
2935        if (*newroot == NULL) {
2936                if (nvlist_alloc(newroot, NV_UNIQUE_NAME, 0) != 0)
2937                        goto out;
2938                freelist = B_TRUE;
2939                if (nvlist_add_string(*newroot, ZPOOL_CONFIG_TYPE,
2940                    VDEV_TYPE_ROOT) != 0)
2941                        goto out;
```



```
2942        } else {
2943                verify(nvlist_remove_all(*newroot, ZPOOL_CONFIG_CHILDREN) == 0);
2944        }

2946        /* Add all the children we found */
2947        if (nvlist_add_nvlist_array(*newroot, ZPOOL_CONFIG_CHILDREN, varray,
2948            lastlog == 0 ? vcount : lastlog) != 0)
2949                goto out;

2951        /*
2952         * If we're just doing a dry run, exit now with success.
2953         */
2954        if (flags.dryrun) {
2955                memory_err = B_FALSE;
2956                freelist = B_FALSE;
2957                goto out;
2958        }

2960        /* now build up the config list & call the ioctl */
2961        if (nvlist_alloc(&newconfig, NV_UNIQUE_NAME, 0) != 0)
2962                goto out;

2964        if (nvlist_add_nvlist(newconfig,
2965            ZPOOL_CONFIG_VDEV_TREE, *newroot) != 0 ||
2966            nvlist_add_string(newconfig,
2967            ZPOOL_CONFIG_POOL_NAME, newname) != 0 ||
2968            nvlist_add_uint64(newconfig, ZPOOL_CONFIG_VERSION, vers) != 0)
2969                goto out;

2971        /*
2972         * The new pool is automatically part of the namespace unless we
2973         * explicitly export it.
2974         */
2975        if (!flags.import)
2976                zc.zc_cookie = ZPOOL_EXPORT_AFTER_SPLIT;
2977        (void) strlcpy(zc.zc_name, zhp->zpool_name, sizeof (zc.zc_name));
2978        (void) strlcpy(zc.zc_string, newname, sizeof (zc.zc_string));
2979        if (zcmd_write_conf_nvlist(hdl, &zc, newconfig) != 0)
2980                goto out;
2981        if (zc_props != NULL && zcmd_write_src_nvlist(hdl, &zc, zc_props) != 0)
2982                goto out;

2984        if (zfs_ioctl(hdl, ZFS_IOC_VDEV_SPLIT, &zc) != 0) {
2985                retval = zpool_standard_error(hdl, errno, msg);
2986                goto out;
2987        }

2989        freelist = B_FALSE;
2990        memory_err = B_FALSE;

2992 out:
2993        if (varray != NULL) {
2994                int v;

2996                for (v = 0; v < vcount; v++)
2997                        nvlist_free(varray[v]);
2998                free(varray);
2999        }
3000        zcmd_free_nvlists(&zc);
3006        if (zc_props)
3001                nvlist_free(zc_props);
3008        if (newconfig)
3002                nvlist_free(newconfig);
3003        if (freelist) {
3004                nvlist_free(*newroot);
3005                *newroot = NULL;
```

```
3006         }

3008         if (retval != 0)
3009                 return (retval);

3011         if (memory_err)
3012                 return (no_memory(hdl));

3014         return (0);
3015 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   97298 Mon Feb 15 12:56:10 2016
new/usr/src/lib/libzfs/common/libzfs_sendrecv.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

2558 static int
2559 zfs_receive_package(libzfs_handle_t *hdl, int fd, const char *destname,
2560     recvflags_t *flags, dmu_replay_record_t *drr, zio_cksum_t *zc,
2561     char **top_zfs, int cleanup_fd, uint64_t *action_handlep)
2562 {
2563         nvlist_t *stream_nv = NULL;
2564         avl_tree_t *stream_avl = NULL;
2565         char *fromsnap = NULL;
2566         char *sendsnap = NULL;
2567         char *cp;
2568         char tofs[ZFS_MAXNAMELEN];
2569         char sendfs[ZFS_MAXNAMELEN];
2570         char errbuf[1024];
2571         dmu_replay_record_t drre;
2572         int error;
2573         boolean_t anyerr = B_FALSE;
2574         boolean_t softerr = B_FALSE;
2575         boolean_t recursive;

2577         (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
2578             "cannot receive"));

2580         assert(drr->drr_type == DRR_BEGIN);
2581         assert(drr->drr_u.drr_begin.drr_magic == DMU_BACKUP_MAGIC);
2582         assert(DMU_GET_STREAM_HDRTYPE(drr->drr_u.drr_begin.drr_versioninfo) ==
2583             DMU_COMPOUNDSTREAM);

2585         /*
2586          * Read in the nvlist from the stream.
2587          */
2588         if (drr->drr_payloadlen != 0) {
2589                 error = recv_read_nvlist(hdl, fd, drr->drr_payloadlen,
2590                     &stream_nv, flags->byteswap, zc);
2591                 if (error) {
2592                         error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2593                         goto out;
2594                 }
2595         }

2597         recursive = (nvlist_lookup_boolean(stream_nv, "not_recursive") ==
2598             ENOENT);

2600         if (recursive && strchr(destname, '@')) {
2601                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2602                     "cannot specify snapshot name for multi-snapshot stream"));
2603                 error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2604                 goto out;
2605         }

2607         /*
2608          * Read in the end record and verify checksum.
2609          */
2610         if (0 != (error = recv_read(hdl, fd, &drre, sizeof (drre),
2611             flags->byteswap, NULL)))
2612                 goto out;
2613         if (flags->byteswap) {
2614                 drre.drr_type = BSWAP_32(drre.drr_type);
2615                 drre.drr_u.drr_end.drr_checksum.zc_word[0] =
2616                     BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[0]);
```



```
2617                 drre.drr_u.drr_end.drr_checksum.zc_word[1] =
2618                     BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[1]);
2619                 drre.drr_u.drr_end.drr_checksum.zc_word[2] =
2620                     BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[2]);
2621                 drre.drr_u.drr_end.drr_checksum.zc_word[3] =
2622                     BSWAP_64(drre.drr_u.drr_end.drr_checksum.zc_word[3]);
2623         }
2624         if (drre.drr_type != DRR_END) {
2625                 error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2626                 goto out;
2627         }
2628         if (!ZIO_CHECKSUM_EQUAL(drre.drr_u.drr_end.drr_checksum, *zc)) {
2629                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2630                     "incorrect header checksum"));
2631                 error = zfs_error(hdl, EZFS_BADSTREAM, errbuf);
2632                 goto out;
2633         }

2635         (void) nvlist_lookup_string(stream_nv, "fromsnap", &fromsnap);

2637         if (drr->drr_payloadlen != 0) {
2638                 nvlist_t *stream_fss;

2640                 VERIFY(0 == nvlist_lookup_nvlist(stream_nv, "fss",
2641                     &stream_fss));
2642                 if ((stream_avl = fsavl_create(stream_fss)) == NULL) {
2643                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
2644                             "couldn't allocate avl tree"));
2645                         error = zfs_error(hdl, EZFS_NOMEM, errbuf);
2646                         goto out;
2647                 }

2649                 if (fromsnap != NULL) {
2650                         nvlist_t *renamed = NULL;
2651                         nvpair_t *pair = NULL;

2653                         (void) strlcpy(tofs, destname, ZFS_MAXNAMELEN);
2654                         if (flags->isprefix) {
2655                                 struct drr_begin *drrb = &drr->drr_u.drr_begin;
2656                                 int i;

2658                                 if (flags->istail) {
2659                                         cp = strrchr(drrb->drr_toname, '/');
2660                                         if (cp == NULL) {
2661                                                 (void) strlcat(tofs, "/",
2662                                                     ZFS_MAXNAMELEN);
2663                                                 i = 0;
2664                                         } else {
2665                                                 i = (cp - drrb->drr_toname);
2666                                         }
2667                                 } else {
2668                                         i = strcspn(drrb->drr_toname, "/@");
2669                                 }
2670                                 /* zfs_receive_one() will create_parents() */
2671                                 (void) strlcat(tofs, &drrb->drr_toname[i],
2672                                     ZFS_MAXNAMELEN);
2673                                 *strchr(tofs, '@') = '\0';
2674                         }

2676                         if (recursive && !flags->dryrun && !flags->nomount) {
2677                                 VERIFY(0 == nvlist_alloc(&renamed,
2678                                     NV_UNIQUE_NAME, 0));
2679                         }

2681                         softerr = recv_incremental_replication(hdl, tofs, flags,
2682                             stream_nv, stream_avl, renamed);
```

```
2684                        /* Unmount renamed filesystems before receiving. */
2685                        while ((pair = nvlist_next_nvpair(renamed,
2686                            pair)) != NULL) {
2687                                zfs_handle_t *zhp;
2688                                prop_changelist_t *clp = NULL;

2690                                zhp = zfs_open(hdl, nvpair_name(pair),
2691                                    ZFS_TYPE_FILESYSTEM);
2692                                if (zhp != NULL) {
2693                                        clp = changelist_gather(zhp,
2694                                            ZFS_PROP_MOUNTPOINT, 0, 0);
2695                                        zfs_close(zhp);
2696                                        if (clp != NULL) {
2697                                                softerr |=
2698                                                    changelist_prefix(clp);
2699                                                changelist_free(clp);
2700                                        }
2701                                }
2702                        }

2704                        nvlist_free(renamed);
2705                }
2706        }

2708        /*
2709         * Get the fs specified by the first path in the stream (the top level
2710         * specified by 'zfs send') and pass it to each invocation of
2711         * zfs_receive_one().
2712         */
2713        (void) strlcpy(sendfs, drr->drr_u.drr_begin.drr_toname,
2714            ZFS_MAXNAMELEN);
2715        if ((cp = strchr(sendfs, '@')) != NULL) {
2716                *cp = '\0';
2717                /*
2718                 * Find the "sendsnap", the final snapshot in a replication
2719                 * stream.  zfs_receive_one() handles certain errors
2720                 * differently, depending on if the contained stream is the
2721                 * last one or not.
2722                 */
2723                sendsnap = (cp + 1);
2724        }

2726        /* Finally, receive each contained stream */
2727        do {
2728                /*
2729                 * we should figure out if it has a recoverable
2730                 * error, in which case do a recv_skip() and drive on.
2731                 * Note, if we fail due to already having this guid,
2732                 * zfs_receive_one() will take care of it (ie,
2733                 * recv_skip() and return 0).
2734                 */
2735                error = zfs_receive_impl(hdl, destname, NULL, flags, fd,
2736                    sendfs, stream_nv, stream_avl, top_zfs, cleanup_fd,
2737                    action_handlep, sendsnap);
2738                if (error == ENODATA) {
2739                        error = 0;
2740                        break;
2741                }
2742                anyerr |= error;
2743        } while (error == 0);

2745        if (drr->drr_payloadlen != 0 && fromsnap != NULL) {
2746                /*
2747                 * Now that we have the fs's they sent us, try the
2748                 * renames again.
```



```
2749                 */
2750                softerr = recv_incremental_replication(hdl, tofs, flags,
2751                    stream_nv, stream_avl, NULL);
2752        }

2754 out:
2755        fsavl_destroy(stream_avl);
2756        if (stream_nv)
2757                nvlist_free(stream_nv);
2758        if (softerr)
2759                error = -2;
2760        if (anyerr)
2761                error = -1;
2762        return (error);
2763 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   28379 Mon Feb 15 12:56:10 2016
new/usr/src/lib/pylibbe/common/libbe_py.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_
```

```
 142 /*
 143  * Function:    beCopy
 144  * Description: Convert Python args to nvlist pairs and call libbe:be_copy
 145  *              to create a Boot Environment
 146  * Parameters:
 147  *   args -     pointer to a python object containing:
 148  *     trgtBeName - The name of the BE to create
 149  *     srcBeName - The name of the BE used to create trgtBeName (optional)
 150  *     rpool - The pool to create the new BE in (optional)
 151  *     srcSnapName - The snapshot name (optional)
 152  *     beNameProperties - The properties to use when creating
 153  *                        the BE (optional)
 154  *
 155  * Returns a pointer to a python object. That Python object will consist of
 156  * the return code and optional attributes, trgtBeName and snapshotName
 157  *      BE_SUCCESS, [trgtBeName], [trgtSnapName] - Success
 158  *      1, [trgtBeName], [trgtSnapName] - Failure
 159  * Scope:
 160  *      Public
 161  */
 162 /* ARGSUSED */
 163 PyObject *
 164 beCopy(PyObject *self, PyObject *args)
 165 {
 166         char    *trgtBeName = NULL;
 167         char    *srcBeName = NULL;
 168         char    *srcSnapName = NULL;
 169         char    *trgtSnapName = NULL;
 170         char    *rpool = NULL;
 171         char    *beDescription = NULL;
 172         Py_ssize_t      pos = 0;
 173         int             ret = BE_PY_SUCCESS;
 174         nvlist_t        *beAttrs = NULL;
 175         nvlist_t        *beProps = NULL;
 176         PyObject        *beNameProperties = NULL;
 177         PyObject        *pkey = NULL;
 178         PyObject        *pvalue = NULL;
 179         PyObject        *retVals = NULL;

 181         if (!PyArg_ParseTuple(args, "|zzzzOz", &trgtBeName, &srcBeName,
 182             &srcSnapName, &rpool, &beNameProperties, &beDescription)) {
 183                 return (Py_BuildValue("[iss]", BE_PY_ERR_PARSETUPLE,
 184                     NULL, NULL));
 185         }

 187         if (!convertPyArgsToNvlist(&beAttrs, 10,
 188             BE_ATTR_NEW_BE_NAME, trgtBeName,
 189             BE_ATTR_ORIG_BE_NAME, srcBeName,
 190             BE_ATTR_SNAP_NAME, srcSnapName,
 191             BE_ATTR_NEW_BE_POOL, rpool,
 192             BE_ATTR_NEW_BE_DESC, beDescription)) {
 193                 nvlist_free(beAttrs);
 194                 return (Py_BuildValue("[iss]", BE_PY_ERR_NVLIST, NULL, NULL));
 195         }

 197         if (beNameProperties != NULL) {
 198                 if (nvlist_alloc(&beProps, NV_UNIQUE_NAME, 0) != 0) {
 199                         (void) printf("nvlist_alloc failed.\n");
 200                         nvlist_free(beAttrs);
```



```
 201                         return (Py_BuildValue("[iss]", BE_PY_ERR_NVLIST,
 202                             NULL, NULL));
 203                 }
 204                 while (PyDict_Next(beNameProperties, &pos, &pkey, &pvalue)) {
 205                         if (!convertPyArgsToNvlist(&beProps, 2,
 206                             PyString_AsString(pkey),
 207                             PyString_AsString(pvalue))) {
 208                                 nvlist_free(beProps);
 209                                 nvlist_free(beAttrs);
 210                                 return (Py_BuildValue("[iss]", BE_PY_ERR_NVLIST,
 211                                     NULL, NULL));
 212                         }
 213                 }
 214         }

 216         if (beProps != NULL && beAttrs != NULL &&
 217             nvlist_add_nvlist(beAttrs, BE_ATTR_ZFS_PROPERTIES,
 218             beProps) != 0) {
 219                 nvlist_free(beProps);
 220                 nvlist_free(beAttrs);
 221                 return (Py_BuildValue("[iss]", BE_PY_ERR_NVLIST,
 222                     NULL, NULL));
 223         }

 225         nvlist_free(beProps);
 225         if (beProps != NULL) nvlist_free(beProps);

 227         if (trgtBeName == NULL) {
 228                 /*
 229                  * Caller wants to get back the BE_ATTR_NEW_BE_NAME and
 230                  * BE_ATTR_SNAP_NAME
 231                  */
 232                 if ((ret = be_copy(beAttrs)) != BE_SUCCESS) {
 233                         nvlist_free(beAttrs);
 234                         return (Py_BuildValue("[iss]", ret, NULL, NULL));
 235                 }

 237                 /*
 238                  * When no trgtBeName is passed to be_copy, be_copy
 239                  * returns an auto generated beName and snapshot name.
 240                  */
 241                 if (nvlist_lookup_string(beAttrs, BE_ATTR_NEW_BE_NAME,
 242                     &trgtBeName) != 0) {
 243                         nvlist_free(beAttrs);
 244                         return (Py_BuildValue("[iss]", BE_PY_ERR_NVLIST,
 245                             NULL, NULL));
 246                 }
 247                 if (nvlist_lookup_string(beAttrs, BE_ATTR_SNAP_NAME,
 248                     &trgtSnapName) != 0) {
 249                         nvlist_free(beAttrs);
 250                         return (Py_BuildValue("[iss]", BE_PY_ERR_NVLIST,
 251                             NULL, NULL));
 252                 }

 254                 retVals = Py_BuildValue("[iss]", BE_PY_SUCCESS,
 255                     trgtBeName, trgtSnapName);
 256                 nvlist_free(beAttrs);
 257                 return (retVals);

 259         } else {
 260                 ret = be_copy(beAttrs);
 261                 nvlist_free(beAttrs);
 262                 return (Py_BuildValue("[iss]", ret, NULL, NULL));
 263         }
 264 }
```

```
______unchanged_portion_omitted_
```

```
**********************************************************
   14985 Mon Feb 15 12:56:10 2016
new/usr/src/man/man3nvpair/nvlist_alloc.3nvpair
patch more-manpage
patch cleanup
**********************************************************
   1 '\" te
   2 .\" Copyright (c) 2004, Sun Microsystems, Inc. All Rights Reserved.
   3 .\" The contents of this file are subject to the terms of the Common Development
   4 .\" You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE or http:
   5 .\" When distributing Covered Code, include this CDDL HEADER in each file and in
   6 .TH NVLIST_ALLOC 3NVPAIR "Feb 15, 2016"
   6 .TH NVLIST_ALLOC 3NVPAIR "Feb 2, 2004"
   7 .SH NAME
   8 nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup,
   9 nvlist_merge, nvlist_xalloc, nvlist_xpack, nvlist_xunpack, nvlist_xdup,
  10 nvlist_lookup_nv_alloc, nv_alloc_init, nv_alloc_reset, nv_alloc_fini \- manage
  11 a name-value pair list
  12 .SH SYNOPSIS
  13 .LP
  14 .nf
  15 \fBcc\fR [ \fIflag\fR... ] \fIfile\fR... \fB-lnvpair\fR [ \fIlibrary\fR... ]
  16 #include <libnvpair.h>

  18 \fBint\fR \fBnvlist_alloc\fR(\fBnvlist_t **\fR\fInvlp\fR, \fBuint_t\fR \fInvflag
  19 .fi

  21 .LP
  22 .nf
  23 \fBint\fR \fBnvlist_xalloc\fR(\fBnvlist_t **\fR\fInvlp\fR, \fBuint_t\fR \fInvfla
  24      \fBnv_alloc_t *\fR \fInva\fR);
  25 .fi

  27 .LP
  28 .nf
  29 \fBvoid\fR \fBnvlist_free\fR(\fBnvlist_t *\fR\fInvl\fR);
  30 .fi

  32 .LP
  33 .nf
  34 \fBint\fR \fBnvlist_size\fR(\fBnvlist_t *\fR\fInvl\fR, \fBsize_t *\fR\fIsize\fR,
  35 .fi

  37 .LP
  38 .nf
  39 \fBint\fR \fBnvlist_pack\fR(\fBnvlist_t *\fR\fInvl\fR, \fBchar **\fR\fIbufp\fR,
  40      \fBint\fR \fIencoding\fR, \fBint\fR \fIflag\fR);
  41 .fi

  43 .LP
  44 .nf
  45 \fBint\fR \fBnvlist_xpack\fR(\fBnvlist_t *\fR\fInvl\fR, \fBchar **\fR\fIbufp\fR,
  46      \fBint\fR \fIencoding\fR, \fBnv_alloc_t *\fR \fInva\fR);
  47 .fi

  49 .LP
  50 .nf
  51 \fBint\fR \fBnvlist_unpack\fR(\fBchar *\fR\fIbuf\fR, \fBsize_t\fR \fIbuflen\fR,
  52      \fBint\fR \fIflag\fR);
  53 .fi

  55 .LP
  56 .nf
  57 \fBint\fR \fBnvlist_xunpack\fR(\fBchar *\fR\fIbuf\fR, \fBsize_t\fR \fIbuflen\fR,
  58      \fBnv_alloc_t *\fR \fInva\fR);
  59 .fi
```



```
  61 .LP
  62 .nf
  63 \fBint\fR \fBnvlist_dup\fR(\fBnvlist_t *\fR\fInvl\fR, \fBnvlist_t **\fR\fInvlp\f
  64 .fi

  66 .LP
  67 .nf
  68 \fBint\fR \fBnvlist_xdup\fR(\fBnvlist_t *\fR\fInvl\fR, \fBnvlist_t **\fR\fInvlp\
  69      \fBnv_alloc_t *\fR \fInva\fR);
  70 .fi

  72 .LP
  73 .nf
  74 \fBint\fR \fBnvlist_merge\fR(\fBnvlist_t *\fR\fIdst\fR, \fBnvlist_t *\fR\fInvl\f
  75 .fi

  77 .LP
  78 .nf
  79 \fBnv_alloc_t *\fR \fBnvlist_lookup_nv_alloc\fR(\fBnvlist_t *\fR\fInvl\fR);
  80 .fi

  82 .LP
  83 .nf
  84 \fBint\fR \fBnv_alloc_init\fR(\fBnv_alloc_t *\fR\fInva\fR, \fBconst nv_alloc_ops
  85      \fB/*\fR \fIargs\fR */ ...);
  86 .fi

  88 .LP
  89 .nf
  90 \fBvoid\fR \fBnv_alloc_reset\fR(\fBnv_alloc_t *\fR\fInva\fR);
  91 .fi

  93 .LP
  94 .nf
  95 \fBvoid\fR \fBnv_alloc_fini\fR(\fBnv_alloc_t *\fR\fInva\fR);
  96 .fi

  98 .SH PARAMETERS
  99 .sp
 100 .ne 2
 101 .na
 102 \fB\fInvlp\fR\fR
 103 .ad
 104 .RS 12n
 105 Address of a pointer to \fBnvlist_t\fR.
 106 .RE

 108 .sp
 109 .ne 2
 110 .na
 111 \fB\fInvflag\fR\fR
 112 .ad
 113 .RS 12n
 114 Specify bit fields defining \fBnvlist\fR properties:
 115 .sp
 116 .ne 2
 117 .na
 118 \fB\fBNV_UNIQUE_NAME\fR\fR
 119 .ad
 120 .RS 23n
 121 The \fBnvpair\fR names are unique.
 122 .RE

 124 .sp
 125 .ne 2
```

```
126 .na
127 \fB\fBNV_UNIQUE_NAME_TYPE\fR\fR
128 .ad
129 .RS 23n
130 Name-data type combination is unique.
131 .RE

133 .RE

135 .sp
136 .ne 2
137 .na
138 \fB\fIflag\fR\fR
139 .ad
140 .RS 12n
141 Specify 0. Reserved for future use.
142 .RE

144 .sp
145 .ne 2
146 .na
147 \fB\fInvl\fR\fR
148 .ad
149 .RS 12n
150 The \fBnvlist_t\fR to be processed.
151 .RE

153 .sp
154 .ne 2
155 .na
156 \fB\fIdst\fR\fR
157 .ad
158 .RS 12n
159 The destination \fBnvlist_t\fR.
160 .RE

162 .sp
163 .ne 2
164 .na
165 \fB\fIsize\fR\fR
166 .ad
167 .RS 12n
168 Pointer to buffer to contain the encoded size.
169 .RE

171 .sp
172 .ne 2
173 .na
174 \fB\fIbufp\fR\fR
175 .ad
176 .RS 12n
177 Address of buffer to pack \fBnvlist\fR into. Must be 8-byte aligned. If
178 \fINULL\fR, library will allocate memory.
179 .RE

181 .sp
182 .ne 2
183 .na
184 \fB\fIbuf\fR\fR
185 .ad
186 .RS 12n
187 Buffer containing packed \fBnvlist\fR.
188 .RE

190 .sp
191 .ne 2
```



```
192 .na
193 \fB\fIbuflen\fR\fR
194 .ad
195 .RS 12n
196 Size of buffer \fIbufp\fR or \fIbuf\fR points to.
197 .RE

199 .sp
200 .ne 2
201 .na
202 \fB\fIencoding\fR\fR
203 .ad
204 .RS 12n
205 Encoding method for packing.
206 .RE

208 .sp
209 .ne 2
210 .na
211 \fB\fInvo\fR\fR
212 .ad
213 .RS 12n
214 Pluggable allocator operations pointer (\fBnv_alloc_ops_t\fR).
215 .RE

217 .sp
218 .ne 2
219 .na
220 \fB\fInva\fR\fR
221 .ad
222 .RS 12n
223 A pointer to an \fBnv_alloc_t\fR structure to be used for the specified
224 \fBnvlist_t\fR.
225 .RE

227 .SH DESCRIPTION
228 .SS "List Manipulation"
229 .sp
230 .LP
231 The \fBnvlist_alloc()\fR function allocates a new name-value pair list and
232 updates \fInvlp\fR to point to the handle. The \fInvflag\fR argument specifies
233 \fBnvlist\fR properties to remain persistent across packing, unpacking, and
234 duplication. If \fBNV_UNIQUE_NAME\fR was specified for \fInvflag\fR, existing
235 nvpairs with matching names are removed before the new nvpair is added. If
236 \fBNV_UNIQUE_NAME_TYPE\fR was specified for \fInvflag\fR, existing nvpairs with
237 matching names and data types are removed before the new nvpair is added. See
238 \fBnvlist_add_byte\fR(3NVPAIR) for more information.
239 .sp
240 .LP
241 The \fBnvlist_xalloc()\fR function is identical to \fBnvlist_alloc()\fR except
242 that \fBnvlist_xalloc()\fR can use a different allocator, as described in the
243 Pluggable Allocators section.
244 .sp
245 .LP
246 The \fBnvlist_free()\fR function frees a name-value pair list. If \fInvl\fR
247 is a null pointer, no action occurs.
246 The \fBnvlist_free()\fR function frees a name-value pair list.
248 .sp
249 .LP
250 The \fBnvlist_size()\fR function returns the minimum size of a contiguous
251 buffer large enough to pack \fInvl\fR. The \fIencoding\fR parameter specifies
252 the method of encoding when packing \fInvl\fR. Supported encoding methods are:
253 .sp
254 .ne 2
255 .na
256 \fB\fBNV_ENCODE_NATIVE\fR\fR
```

```
.ad
.RS 20n
Straight \fBbcopy()\fR as described in \fBbcopy\fR(3C).
.RE

.sp
.ne 2
.na
\fB\fBNV_ENCODE_XDR\fR\fR
.ad
.RS 20n
Use XDR encoding, suitable for sending to another host.
.RE

.sp
.LP
The \fBnvlist_pack()\fR function packs \fInvl\fR into contiguous memory
starting at *\fIbufp\fR. The \fIencoding\fR parameter specifies the method of
encoding (see above).
.RS +4
.TP
.ie t \(bu
.el o
If *\fIbufp\fR is not \fINULL\fR, *\fIbufp\fR is expected to be a
caller-allocated buffer of size *\fIbuflen\fR.
.RE
.RS +4
.TP
.ie t \(bu
.el o
If *\fIbufp\fR is \fINULL\fR, the library will allocate memory and update
*\fIbufp\fR to point to the memory and update *\fIbuflen\fR to contain the size
of the allocated memory.
.RE
.sp
.LP
The \fBnvlist_xpack()\fR function is identical to \fBnvlist_pack()\fR except
that \fBnvlist_xpack()\fR can use a different allocator.
.sp
.LP
The \fBnvlist_unpack()\fR function takes a buffer with a packed \fBnvlist_t\fR
and unpacks it into a searchable \fBnvlist_t\fR. The library allocates memory
for \fBnvlist_t\fR. The caller is responsible for freeing the memory by calling
\fBnvlist_free()\fR.
.sp
.LP
The \fBnvlist_xunpack()\fR function is identical to \fBnvlist_unpack()\fR
except that \fBnvlist_xunpack()\fR can use a different allocator.
.sp
.LP
The \fBnvlist_dup()\fR function makes a copy of \fInvl\fR and updates
\fInvlp\fR to point to the copy.
.sp
.LP
The \fBnvlist_xdup()\fR function is identical to \fBnvlist_dup()\fR except that
\fBnvlist_xdup()\fR can use a different allocator.
.sp
.LP
The \fBnvlist_merge()\fR function adds copies of all name-value pairs from
\fInvl\fR to \fIdst\fR.  Name-value pairs in \fIdst\fR are replaced with
name-value pairs from \fInvl\fR that have identical names (if \fIdst\fR has the
type \fBNV_UNIQUE_NAME\fR) or identical names and types (if \fIdst\fR has the
type \fBNV_UNIQUE_NAME_TYPE\fR).
.sp
.LP
The \fBnvlist_lookup_nv_alloc()\fR function retrieves the pointer to the
```



```
allocator that was used when manipulating a name-value pair list.
.SS "Pluggable Allocators"
.SS "Using Pluggable Allocators"
.sp
.LP
The \fBnv_alloc_init()\fR, \fBnv_alloc_reset()\fR and \fBnv_alloc_fini()\fR
functions provide an interface to specify the allocator to be used when
manipulating a name-value pair list.
.sp
.LP
The \fBnv_alloc_init()\fR function determines the allocator properties and puts
them into the \fInva\fR argument. The application must specify the \fInv_arg\fR
and \fInvo\fR arguments and an optional variable argument list. The optional
arguments are passed to the (*\fBnv_ao_init()\fR) function.
.sp
.LP
The \fInva\fR argument must be passed to \fBnvlist_xalloc()\fR,
\fBnvlist_xpack()\fR, \fBnvlist_xunpack()\fR and \fBnvlist_xdup()\fR.
.sp
.LP
The \fBnv_alloc_reset()\fR function is responsible for resetting the allocator
properties to the data specified by \fBnv_alloc_init()\fR. When no
(*\fBnv_ao_reset()\fR) function is specified, \fBnv_alloc_reset()\fR has no
effect.
.sp
.LP
The \fBnv_alloc_fini()\fR function destroys the allocator properties determined
by \fBnv_alloc_init()\fR. When a (*\fBnv_ao_fini()\fR) function is specified,
it is called from \fBnv_alloc_fini()\fR.
.sp
.LP
The disposition of the allocated objects and the memory used to store them is
left to the allocator implementation.
.sp
.LP
The \fBnv_alloc_nosleep\fR \fBnv_alloc_t\fR can be used with
\fBnvlist_xalloc()\fR to mimic the behavior of \fBnvlist_alloc()\fR.
.sp
.LP
The nvpair allocator framework provides a pointer to the operation structure of
a fixed buffer allocator. This allocator, \fBnv_fixed_ops\fR, uses a
pre-allocated buffer for memory allocations. It is intended primarily for
kernel use and is described on \fBnvlist_alloc\fR(9F).
.sp
.LP
An example program that uses the pluggable allocator functionality is provided
on \fBnvlist_alloc\fR(9F).
.SS "Creating Pluggable Allocators"
.sp
.LP
Any producer of name-value pairs can specify its own allocator functions. The
application must provide the following pluggable allocator operations:
.sp
.in +2
.nf
int (*nv_ao_init)(nv_alloc_t *nva, va_list nv_valist);
void (*nv_ao_fini)(nv_alloc_t *nva);
void *(*nv_ao_alloc)(nv_alloc_t *nva, size_t sz);
void (*nv_ao_reset)(nv_alloc_t *nva);
void (*nv_ao_free)(nv_alloc_t *nva, void *buf, size_t sz);
.fi
.in -2

.sp
.LP
The \fInva\fR argument of the allocator implementation is always the first
```

```
389 argument.
390 .sp
391 .LP
392 The optional (*\fBnv_ao_init()\fR) function is responsible for filling the data
393 specified by \fBnv_alloc_init()\fR into the \fInva_arg\fR argument.  The
394 (*\fBnv_ao_init()\fR) function is only called when \fBnv_alloc_init()\fR is
395 executed.
396 .sp
397 .LP
398 The optional (*\fBnv_ao_fini()\fR) function is responsible for the cleanup of
399 the allocator implementation. It is called by \fBnv_alloc_fini()\fR.
400 .sp
401 .LP
402 The required (*\fBnv_ao_alloc()\fR) function is used in the nvpair allocation
403 framework for memory allocation. The \fIsz\fR argument specifies the size of
404 the requested buffer.
405 .sp
406 .LP
407 The optional (*\fBnv_ao_reset()\fR) function is responsible for resetting the
408 \fInva_arg\fR argument to the data specified by \fBnv_alloc_init()\fR.
409 .sp
410 .LP
411 The required (*\fBnv_ao_free()\fR) function is used in the nvpair allocator
412 framework for memory deallocation. The \fIbuf\fR argument is a pointer to a
413 block previously allocated by the (*\fBnv_ao_alloc()\fR) function. The size
414 argument \fIsz\fR must exactly match the original allocation.
415 .sp
416 .LP
417 The disposition of the allocated objects and the memory used to store them is
418 left to the allocator implementation.
419 .SH RETURN VALUES
420 .sp
421 .LP
422 These functions return 0 on success and an error value on failure.
423 .sp
424 .LP
425 The \fBnvlist_lookup_nv_alloc()\fR function returns a pointer to an allocator.
426 .SH ERRORS
427 .sp
428 .LP
429 These functions will fail if:
430 .sp
431 .ne 2
432 .na
433 \fB\fBEINVAL\fR\fR
434 .ad
435 .RS 10n
436 There is an invalid argument.
437 .RE

439 .sp
440 .LP
441 The \fBnvlist_alloc()\fR, \fBnvlist_dup()\fR, \fBnvlist_pack()\fR,
442 \fBnvlist_unpack()\fR, \fBnvlist_merge()\fR, \fBnvlist_xalloc()\fR,
443 \fBnvlist_xdup()\fR, \fBnvlist_xpack()\fR, and \fBnvlist_xunpack()\fR functions
444 will fail if:
445 .sp
446 .ne 2
447 .na
448 \fB\fBENOMEM\fR\fR
449 .ad
450 .RS 10n
451 There is insufficient memory.
452 .RE

454 .sp
```



```
455 .LP
456 The \fBnvlist_pack()\fR, \fBnvlist_unpack()\fR, \fBnvlist_xpack()\fR, and
457 \fBnvlist_xunpack()\fR functions will fail if:
458 .sp
459 .ne 2
460 .na
461 \fB\fBEFAULT\fR\fR
462 .ad
463 .RS 11n
464 An encode/decode error occurs.
465 .RE

467 .sp
468 .ne 2
469 .na
470 \fB\fBENOTSUP\fR\fR
471 .ad
472 .RS 11n
473 An encode/decode method is not supported.
474 .RE

476 .SH EXAMPLES
477 .sp
478 .in +2
479 .nf
480 /*
481  * Program to create an nvlist.
482  */
483 #include <stdio.h>
484 #include <sys/types.h>
485 #include <string.h>
486 #include <libnvpair.h>

488 /* generate a packed nvlist */
489 static int
490 create_packed_nvlist(char **buf, uint_t *buflen, int encode)
491 {
492     uchar_t bytes[] = {0xaa, 0xbb, 0xcc, 0xdd};
493     int32_t int32[] = {3, 4, 5};
494     char *strs[] = {"child0", "child1", "child2"};
495     int err;
496     nvlist_t *nvl;

498     err = nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0);    /* allocate list */
499     if (err) {
500         (void) printf("nvlist_alloc() failed\en");
501         return (err);
502     }

504     /* add a value of some types */
505     if ((nvlist_add_byte(nvl, "byte", bytes[0]) != 0) ||
506         (nvlist_add_int32(nvl, "int32", int32[0]) != 0) ||
507         (nvlist_add_int32_array(nvl, "int32_array", int32, 3) != 0) ||
508         (nvlist_add_string_array(nvl, "string_array", strs, 3) != 0)) {
509         nvlist_free(nvl);
510         return (-1);
511     }

513     err = nvlist_size(nvl, buflen, encode);
514     if (err) {
515         (void) printf("nvlist_size: %s\en", strerror(err));
516         nvlist_free(nvl);
517         return (err);
518     }

520     /* pack into contig. memory */
```

```
 521     err = nvlist_pack(nvl, buf, buflen, encode, 0);
 522     if (err)
 523         (void) printf("nvlist_pack: %s\en", strerror(err));

 525     /* free the original list */
 526     nvlist_free(nvl);
 527     return (err);
 528 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   15073 Mon Feb 15 12:56:11 2016
new/usr/src/man/man9f/nvlist_alloc.9f
patch more-manpage
patch cleanup
**********************************************************
'\" te
.\" Copyright (c) 2006, Sun Microsystems, Inc. All Rights Reserved.
.\" The contents of this file are subject to the terms of the Common Development
.\" You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE or http:
.\" When distributing Covered Code, include this CDDL HEADER in each file and in
.TH NVLIST_ALLOC 9F "Feb 15, 2016"
.TH NVLIST_ALLOC 9F "Jan 16, 2006"
.SH NAME
nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup,
nv_alloc_init, nv_alloc_fini, nvlist_xalloc, nvlist_xpack, nvlist_xunpack,
nvlist_xdup, nvlist_merge \- Manage a name-value pair list
.SH SYNOPSIS
.LP
.nf
#include <sys/nvpair.h>

List Manipulation:

\fBint\fR \fBnvlist_alloc\fR(\fBnvlist_t **\fR\fInvlp\fR, \fBuint_t\fR \fInvflag
     \fBint\fR \fIkmflag\fR);
.fi

.LP
.nf
\fBint\fR \fBnvlist_xalloc\fR(\fBnvlist_t **\fR\fInvlp\fR, \fBuint_t\fR \fInvfla
.fi

.LP
.nf
\fBvoid\fR \fBnvlist_free\fR(\fBnvlist_t *\fR\fInvl\fR);
.fi

.LP
.nf
\fBint\fR \fBnvlist_size\fR(\fBnvlist_t *\fR\fInvl\fR, \fBsize_t *\fR\fIsize\fR,
.fi

.LP
.nf
\fBint\fR \fBnvlist_pack\fR(\fBnvlist_t *\fR\fInvl\fR, \fBchar **\fR\fIbufp\fR,
     \fBint\fR \fIflag\fR);
.fi

.LP
.nf
\fBint\fR \fBnvlist_xpack\fR(\fBnvlist_t *\fR\fInvl\fR, \fBchar **\fR\fIbufp\fR,
     \fBnv_alloc_t *\fR\fInva\fR);
.fi

.LP
.nf
\fBint\fR \fBnvlist_unpack\fR(\fBchar *\fR\fIbuf\fR, \fBsize_t\fR \fIbuflen\fR,
.fi

.LP
.nf
\fBint\fR \fBnvlist_xunpack\fR(\fBchar *\fR\fIbuf\fR, \fBsize_t\fR \fIbuflen\fR,
     \fBnv_alloc_t *\fR\fInva\fR);
.fi
```



```
.LP
.nf
\fBint\fR \fBnvlist_dup\fR(\fBnvlist_t *\fR\fInvl\fR, \fBnvlist_t **\fR\fInvlp\f
.fi

.LP
.nf
\fBint\fR \fBnvlist_xdup\fR(\fBnvlist_t *\fR\fInvl\fR, \fBnvlist_t **\fR\fInvlp\
.fi

.LP
.nf
\fBint\fR \fBnvlist_merge\fR(\fBnvlist_t *\fR\fIdst\fR, \fBnvlist_t *\fR\fInvl\f
.fi

.LP
.nf
Pluggable Allocator Configuration:

\fBnv_alloc_t *\fR\fBnvlist_lookup_nv_alloc\fR(\fBnvlist_t *);\fR
.fi

.LP
.nf
\fBint\fR \fBnv_alloc_init\fR(\fBnv_alloc_t *\fR\fInva\fR,
     \fBconst nv_alloc_ops_t *\fR \fInvo,\fR/* args */ ...);
.fi

.LP
.nf
\fBvoid\fR  \fBnv_alloc_reset\fR(\fBnv_alloc_t  *\fR\fInva\fR);
.fi

.LP
.nf
\fBvoid\fR  \fBnv_alloc_fini\fR(\fBnv_alloc_t *\fR\fInva\fR);
.fi

.LP
.nf
Pluggable Allocation Initialization with Fixed Allocator:

\fBint\fR \fBnv_alloc_init\fR(\fBnv_alloc_t *\fR\fInva\fR,
     \fBnv_fixed_ops\fR, \fBvoid *\fR \fIbufptr\fR,  \fBsize_t\fR sz);
.fi

.SH INTERFACE LEVEL
.sp
.LP
Solaris DDI specific (Solaris DDI)
.SH PARAMETERS
.sp
.ne 2
.na
\fB\fInvlp\fR\fR
.ad
.RS 12n
Address of a pointer to list of name-value pairs (\fBnvlist_t\fR).
.RE

.sp
.ne 2
.na
\fB\fInvflag\fR\fR
.ad
.RS 12n
```

```
126 Specify bit fields defining \fBnvlist_t\fR properties:
127 .sp
128 .ne 2
129 .na
130 \fB\fBNV_UNIQUE_NAME\fR\fR
131 .ad
132 .RS 23n
133 \fBnvpair\fR names are unique.
134 .RE

136 .sp
137 .ne 2
138 .na
139 \fB\fBNV_UNIQUE_NAME_TYPE\fR\fR
140 .ad
141 .RS 23n
142 Name-data type combination is unique
143 .RE

145 .RE

147 .sp
148 .ne 2
149 .na
150 \fB\fIkmflag\fR\fR
151 .ad
152 .RS 12n
153 Kernel memory allocation policy, either \fBKM_SLEEP\fR or \fBKM_NOSLEEP\fR.
154 .RE

156 .sp
157 .ne 2
158 .na
159 \fB\fInvl\fR\fR
160 .ad
161 .RS 12n
162 \fBnvlist_t\fR to be processed.
163 .RE

165 .sp
166 .ne 2
167 .na
168 \fB\fIdst\fR\fR
169 .ad
170 .RS 12n
171 Destination \fBnvlist_t\fR.
172 .RE

174 .sp
175 .ne 2
176 .na
177 \fB\fIsize\fR\fR
178 .ad
179 .RS 12n
180 Pointer to buffer to contain the encoded size.
181 .RE

183 .sp
184 .ne 2
185 .na
186 \fB\fIbufp\fR\fR
187 .ad
188 .RS 12n
189 Address of buffer to pack \fBnvlist\fR into. Must be 8-byte aligned. If NULL,
190 library will allocate memory.
191 .RE
```



```
193 .sp
194 .ne 2
195 .na
196 \fB\fIbuf\fR\fR
197 .ad
198 .RS 12n
199 Buffer containing packed \fBnvlist_t\fR.
200 .RE

202 .sp
203 .ne 2
204 .na
205 \fB\fIbuflen\fR\fR
206 .ad
207 .RS 12n
208 Size of buffer \fIbufp\fR or \fIbuf\fR points to.
209 .RE

211 .sp
212 .ne 2
213 .na
214 \fB\fIencoding\fR\fR
215 .ad
216 .RS 12n
217 Encoding method for packing.
218 .RE

220 .sp
221 .ne 2
222 .na
223 \fB\fInvo\fR\fR
224 .ad
225 .RS 12n
226 Pluggable allocator operations pointer (nv_alloc_ops_t).
227 .RE

229 .sp
230 .ne 2
231 .na
232 \fB\fInva\fR\fR
233 .ad
234 .RS 12n
235 Points to a nv_alloc_t structure to be used for the specified \fBnvlist_t\fR.
236 .RE

238 .SH DESCRIPTION
239 .sp
240 .LP
241 List Manipulation:
242 .sp
243 .LP
244 The \fBnvlist_alloc()\fR function allocates a new name-value pair list and
245 updates \fInvlp\fR to point to the handle. The argument \fInvflag\fR specifies
246 \fBnvlist_t\fR properties to remain persistent across packing, unpacking, and
247 duplication.
248 .sp
249 .LP
250 If \fBNV_UNIQUE_NAME\fR is specified for nvflag, existing nvpairs with matching
251 names are removed before the new nvpair is added. If \fBNV_UNIQUE_NAME_TYPE\fR
252 is specified for nvflag, existing nvpairs with matching names and data types
253 are removed before the new nvpair is added. See \fBnvlist_add_byte\fR(9F) for
254 more details.
255 .sp
256 .LP
257 The \fBnvlist_xalloc()\fR function differs from \fBnvlist_alloc()\fR in that
```

\fBnvlist_xalloc()\fR can use a different allocator, as described in the Pluggable Allocators section.
.sp
.LP
The \fBnvlist_free()\fR function frees a name-value pair list. If \fInvl\fR is a null pointer, no action occurs.
The \fBnvlist_free()\fR function frees a name-value pair list.
.sp
.LP
The \fBnvlist_size()\fR function returns the minimum size of a contiguous buffer large enough to pack \fInvl\fR. The \fIencoding\fR parameter specifies the method of encoding when packing \fInvl\fR. Supported encoding methods are:
.sp
.ne 2
.na
\fB\fBNV_ENCODE_NATIVE\fR\fR
.ad
.RS 20n
Straight \fBbcopy()\fR as described in \fBbcopy\fR(9F).
.RE

.sp
.ne 2
.na
\fB\fBNV_ENCODE_XDR\fR\fR
.ad
.RS 20n
Use XDR encoding, suitable for sending to another host.
.RE

.sp
.LP
The \fBnvlist_pack()\fR function packs \fInvl\fR into contiguous memory starting at *\fIbufp\fR. The \fIencoding\fR parameter specifies the method of encoding (see above).
.RS +4
.TP
.ie t \(bu
.el o
If *\fIbufp\fR is not NULL, *\fIbufp\fR is expected to be a caller-allocated buffer of size *\fIbuflen\fR. The \fIkmflag\fR argument is ignored.
.RE
.RS +4
.TP
.ie t \(bu
.el o
If *\fIbufp\fR is NULL, the library allocates memory and updates *\fIbufp\fR to point to the memory and updates *\fIbuflen\fR to contain the size of the allocated memory. The value of \fIkmflag\fR indicates the memory allocation policy
.RE
.sp
.LP
The \fBnvlist_xpack()\fR function differs from \fBnvlist_pack()\fR in that \fBnvlist_xpack()\fR can use a different allocator.
.sp
.LP
The \fBnvlist_unpack()\fR function takes a buffer with a packed \fBnvlist_t\fR and unpacks it into a searchable \fBnvlist_t\fR. The library allocates memory for \fBnvlist_t\fR. The caller is responsible for freeing the memory by calling \fBnvlist_free()\fR.
.sp
.LP
The \fBnvlist_xunpack()\fR function differs from \fBnvlist_unpack()\fR in that \fBnvlist_xunpack()\fR can use a different allocator.
.sp

.LP
The \fBnvlist_dup()\fR function makes a copy of \fInvl\fR and updates \fInvlp\fR to point to the copy.
.sp
.LP
The \fBnvlist_xdup()\fR function differs from \fBnvlist_dup()\fR in that \fBnvlist_xdup()\fR can use a different allocator.
.sp
.LP
The \fBnvlist_merge()\fR function adds copies of all name-value pairs from \fBnvlist_t\fR \fInvl\fR to \fBnvlist_t dst\fR. Name-value pairs in dst are replaced with name-value pairs from \fBnvl\fR which have identical names (if dst has the type \fBNV_UNIQUE_NAME\fR), or identical names and types (if dst has the type \fBNV_UNIQUE_NAME_TYPE\fR).
.sp
.LP
The \fBnvlist_lookup_nv_alloc()\fR function retrieves the pointer to the allocator used when manipulating a name-value pair list.
.SS "PLUGGABLE ALLOCATORS"
.sp
.LP
Using Pluggable Allocators:
.sp
.LP
The \fBnv_alloc_init()\fR, \fBnv_alloc_reset()\fR and \fBnv_alloc_fini()\fR functions provide an interface that specifies the allocator to be used when manipulating a name-value pair list.
.sp
.LP
The \fBnv_alloc_init()\fR determines allocator properties and puts them into the \fInva\fR argument. You need to specify the \fInv_arg\fR argument, the \fInvo\fR argument and an optional variable argument list. The optional arguments are passed to the (*\fBnv_ao_init()\fR) function.
.sp
.LP
The \fInva\fR argument must be passed to \fBnvlist_xalloc()\fR, \fBnvlist_xpack()\fR, \fBnvlist_xunpack()\fR and \fBnvlist_xdup()\fR.
.sp
.LP
The \fBnv_alloc_reset()\fR function resets the allocator properties to the data specified by \fBnv_alloc_init()\fR. When no (*\fBnv_ao_reset()\fR) function is specified, \fBnv_alloc_reset()\fR is without effect.
.sp
.LP
The \fBnv_alloc_fini()\fR destroys the allocator properties determined by \fBnv_alloc_init()\fR. When a (*\fBnv_ao_fini()\fR) routine is specified, it is called from \fBnv_alloc_fini()\fR.
.sp
.LP
The disposition of the allocated objects and the memory used to store them is left to the allocator implementation.
.sp
.LP
The ‘nv_alloc_sleep’ and ‘nv_alloc_nosleep’ nv_alloc_t pointers may be used with nvlist_xalloc to mimic the behavior of nvlist_alloc with KM_SLEEP and KM_NOSLEEP, respectively.
.sp
.in +2
.nf
o nv_alloc_nosleep
o nv_alloc_sleep
.fi
.in -2

.sp
.LP

```
389 The nvpair framework provides a fixed-buffer allocator, accessible via
390 nv_fixed_ops.
391 .sp
392 .in +2
393 .nf
394 o  nv_fixed_ops
395 .fi
396 .in -2

398 .sp
399 .LP
400 Given a buffer size and address, the fixed-buffer allocator allows for the
401 creation of nvlists in contexts where malloc or kmem_alloc services may not be
402 available. The fixed-buffer allocator is designed primarily to support the
403 creation of nvlists.
404 .sp
405 .LP
406 Memory freed using \fBnvlist_free()\fR, pair-removal, or similar routines is
407 not reclaimed.
408 .sp
409 .LP
410 When used to initialize the fixed-buffer allocator, nv_alloc_init should be
411 called as follows:
412 .sp
413 .in +2
414 .nf
415 int nv_alloc_init(nv_alloc_t *nva, const nv_alloc_ops_t *nvo,
416     void *bufptr, size_t sz);
417 .fi
418 .in -2

420 .sp
421 .LP
422 When invoked on a fixed-buffer, the \fBnv_alloc_reset()\fR function resets the
423 fixed buffer and prepares it for re-use. The framework consumer is responsible
424 for freeing the buffer passed to \fBnv_alloc_init()\fR.
425 .SS "CREATING PLUGGABLE ALLOCATORS"
426 .sp
427 .LP
428 Any producer of name-value pairs may possibily specify his own allocator
429 routines. You must provide the following pluggable allocator operations in the
430 allocator implementation.
431 .sp
432 .in +2
433 .nf
434 int (*nv_ao_init)(nv_alloc_t *nva, va_list nv_valist);
435 void (*nv_ao_fini)(nv_alloc_t *nva);
436 void *(*nv_ao_alloc)(nv_alloc_t *nva, size_t sz);
437 void (*nv_ao_reset)(nv_alloc_t *nva);
438 void (*nv_ao_free)(nv_alloc_t *nva, void *buf, size_t sz);
439 .fi
440 .in -2

442 .sp
443 .LP
444 The \fInva\fR argument of the allocator implementation is always the first
445 argument.
446 .sp
447 .LP
448 The optional (*\fBnv_ao_init()\fR ) function is responsible for filling the
449 data specified by \fBnv_alloc_init()\fR into the \fBnva_arg()\fR argument.  The
450 (*\fBnv_ao_init()\fR) function is called only when \fBnv_alloc_init()\fR is
451 executed.
452 .sp
453 .LP
454 The optional (*\fBnv_ao_fini()\fR) function is responsible for the cleanup of
```



```
455 the allocator implementation. It is called by \fBnv_alloc_fini()\fR.
456 .sp
457 .LP
458 The required (*\fBnv_ao_alloc()\fR) function is used in the nvpair allocation
459 framework for memory allocation. The sz argument specifies the size of the
460 requested buffer.
461 .sp
462 .LP
463 The optional (*\fBnv_ao_reset()\fR) function is responsible for resetting the
464 nva_arg argument to the data specified by \fBnv_alloc_init()\fR.
465 .sp
466 .LP
467 The required (*\fBnv_ao_free()\fR) function is used in the nvpair allocator
468 framework for memory de-allocation. The argument buf is a pointer to a block
469 previously allocated by (*\fBnv_ao_alloc()\fR) function. The size argument sz
470 must exactly match the original allocation.
471 .sp
472 .LP
473 The disposition of the allocated objects and the memory used to store them is
474 left to the allocator implementation.
475 .SH RETURN VALUES
476 .sp
477 .LP
478 For \fBnvlist_alloc()\fR, \fBnvlist_dup()\fR, \fBnvlist_xalloc()\fR, and
479 \fBnvlist_xdup()\fR:
480 .sp
481 .ne 2
482 .na
483 \fB\fB0\fR\fR
484 .ad
485 .RS 10n
486 success
487 .RE

489 .sp
490 .ne 2
491 .na
492 \fB\fBEINVAL\fR\fR
493 .ad
494 .RS 10n
495 invalid argument
496 .RE

498 .sp
499 .ne 2
500 .na
501 \fB\fBENOMEM\fR\fR
502 .ad
503 .RS 10n
504 insufficient memory
505 .RE

507 .sp
508 .LP
509 For \fBnvlist_pack()\fR, \fBnvlist_unpack()\fR, \fBnvlist_xpack()\fR, and
510 \fBnvlist_xunpack()\fR:
511 .sp
512 .ne 2
513 .na
514 \fB\fB0\fR\fR
515 .ad
516 .RS 11n
517 success
518 .RE

520 .sp
```

```
.ne 2
.na
\fB\fBEINVAL\fR\fR
.ad
.RS 11n
invalid argument
.RE

.sp
.ne 2
.na
\fB\fBENOMEM\fR\fR
.ad
.RS 11n
insufficient memory
.RE

.sp
.ne 2
.na
\fB\fBEFAULT\fR\fR
.ad
.RS 11n
encode/decode error
.RE

.sp
.ne 2
.na
\fB\fBENOTSUP\fR\fR
.ad
.RS 11n
encode/decode method not supported
.RE

.sp
.LP
For \fBnvlist_size()\fR:
.sp
.ne 2
.na
\fB\fB0\fR\fR
.ad
.RS 10n
success
.RE

.sp
.ne 2
.na
\fB\fBEINVAL\fR\fR
.ad
.RS 10n
invalid argument
.RE

.sp
.LP
For \fBnvlist_lookup_nv_alloc()\fR:
.sp
.LP
pointer to the allocator
.SH USAGE
.sp
.LP
The fixed-buffer allocator is very simple allocator. It uses a pre-allocated
```



```
buffer for memory allocations and it can be used in interrupt context. You are
responsible for allocation and de-allocation for the pre-allocated buffer.
.SH EXAMPLES
.sp
.in +2
.nf
  /*
   * using the fixed-buffer allocator.
   */
   #include <sys/nvpair.h>

  /* initialize the nvpair allocator framework */
  static nv_alloc_t *
  init(char *buf, size_t size)
  {
       nv_alloc_t *nvap;

       if ((nvap = kmem_alloc(sizeof(nv_alloc_t), KM_SLEEP)) == NULL)
           return (NULL);

       if (nv_alloc_init(nvap, nv_fixed_ops, buf, size) == 0)
           return (nvap);

       return (NULL);
   }

   static void
   fini(nv_alloc_t *nvap)
   {
         nv_alloc_fini(nvap);
         kmem_free(nvap, sizeof(nv_alloc_t));
   }
    static int
    interrupt_context(nv_alloc_t *nva)
    {
       nvlist_t *nvl;
       int error;

       if ((error = nvlist_xalloc(&nvl, NV_UNIQUE_NAME, nva)) != 0)
            return (-1);

       if ((error = nvlist_add_int32(nvl, "name", 1234)) == 0)
            error = send_nvl(nvl);

       nvlist_free(nvl);
       return (error);
      }
.fi
.in -2

.SH CONTEXT
.sp
.LP
The \fBnvlist_alloc()\fR, \fBnvlist_pack()\fR, \fBnvlist_unpack()\fR, and
\fBnvlist_dup()\fR functions can be called from interrupt context only if the
\fBKM_NOSLEEP\fR flag is set. They can be called from user context with any
valid flag.
.sp
.LP
The \fBnvlist_xalloc()\fR, \fBnvlist_xpack()\fR, \fBnvlist_xunpack()\fR, and
\fBnvlist_xdup()\fR functions can be called from interrupt context only if (1)
the default allocator is used and the \fBKM_NOSLEEP\fR flag is set or (2) the
specified allocator did not sleep for free memory (for example, it uses a
pre-allocated buffer for memory allocations).
.sp
.LP
```

653 These functions can be called from user or kernel context with any valid flag.

****
2430 Mon Feb 15 12:56:11 2016
new/usr/src/uts/common/avs/ns/nsctl/nsc_trap.c
6659 nvlist_free(NULL) is a no-op
****

```
/*
 * 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.
 */

#include <sys/ddi.h>
#include <sys/sunddi.h>

#ifdef DS_DDICT
#include "../contract.h"
#endif

#define SVE_STE_CLASS   "SVE_STE"
#define SVE_II_CLASS    "SVE_II"
#define SVE_CACHE_CLASS "SVE_CACHE"

void
nsc_do_sysevent(char *driver_name, char *trap_messages, int errorno,
        int alertlevel, char *component, dev_info_t *info_dip)
{
#if !defined(DS_DDICT) && !defined(_SunOS_5_6) && \
        !defined(_SunOS_5_7) && !defined(_SunOS_5_8)

        nvlist_t *attr_list;
        int rc;

        attr_list = NULL;
        rc = nvlist_alloc(&attr_list, NV_UNIQUE_NAME_TYPE, KM_SLEEP);
        if (rc != 0) {
                goto out;
        }
        rc = nvlist_add_int32(attr_list, "alertlevel", alertlevel);
        if (rc != 0) {
                goto out;
        }
        rc = nvlist_add_string(attr_list, "messagevalue", trap_messages);
        if (rc != 0) {
                goto out;
        }
        rc = nvlist_add_int32(attr_list, "errorno", errorno);
        if (rc != 0) {
```



```
                goto out;
        }
        if (strcmp(driver_name, "sdbc") == 0)
                rc = ddi_log_sysevent(info_dip, DDI_VENDOR_SUNW,
                    SVE_CACHE_CLASS, component, attr_list, NULL, DDI_SLEEP);
        else if (strcmp(driver_name, "ste") == 0)
                rc = ddi_log_sysevent(info_dip, DDI_VENDOR_SUNW,
                    SVE_STE_CLASS, component, attr_list, NULL, DDI_SLEEP);
        else if (strcmp(driver_name, "ii") == 0)
                rc = ddi_log_sysevent(info_dip, DDI_VENDOR_SUNW,
                    SVE_II_CLASS, component, attr_list, NULL, DDI_SLEEP);
out:
        if (attr_list)
        nvlist_free(attr_list);

        if (rc != 0) {
                cmn_err(CE_WARN, "!%s: unable to log sysevent %d:%s and %d",
                    driver_name, errorno, trap_messages, alertlevel);
        }
#endif  /* which O/S? */
}
```

*unchanged_portion_omitted_*

```
**********************************************************
   68464 Mon Feb 15 12:56:11 2016
new/usr/src/uts/common/contract/device.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

1531 /*
1532  * Core routine called by event-specific routines when an event occurs.
1533  * Determines if an event should be be published, and if it is to be
1534  * published, whether a negotiation should take place. Also implements
1535  * NEGEND events which publish the final disposition of an event after
1536  * negotiations are complete.
1537  *
1538  * When an event occurs on a minor node, this routine walks the list of
1539  * contracts hanging off a devinfo node and for each contract on the affected
1540  * dip, evaluates the following cases
1541  *
1542  *      a. an event that is synchronous, breaks the contract and NONEG not set
1543  *              - bumps up the outstanding negotiation counts on the dip
1544  *              - marks the dip as undergoing negotiation (devi_ct_neg)
1545  *              - event of type CTE_NEG is published
1546  *      b. an event that is synchronous, breaks the contract and NONEG is set
1547  *              - sets the final result to CT_NACK, event is blocked
1548  *              - does not publish an event
1549  *      c. event is asynchronous and breaks the contract
1550  *              - publishes a critical event irrespect of whether the NONEG
1551  *                flag is set, since the contract will be broken and contract
1552  *                owner needs to be informed.
1553  *      d. No contract breakage but the owner has subscribed to the event
1554  *              - publishes the event irrespective of the NONEG event as the
1555  *                owner has explicitly subscribed to the event.
1556  *      e. NEGEND event
1557  *              - publishes a critical event. Should only be doing this if
1558  *                if NONEG is not set.
1559  *      f. all other events
1560  *              - Since a contract is not broken and this event has not been
1561  *                subscribed to, this event does not need to be published for
1562  *                for this contract.
1563  *
1564  *      Once an event is published, what happens next depends on the type of
1565  *      event:
1566  *
1567  *      a. NEGEND event
1568  *              - cleanup all state associated with the preceding negotiation
1569  *                and return CT_ACK to the caller of contract_device_publish()
1570  *      b. NACKed event
1571  *              - One or more contracts had the NONEG term, so the event was
1572  *                blocked. Return CT_NACK to the caller.
1573  *      c. Negotiated event
1574  *              - Call wait_for_acks() to wait for responses from contract
1575  *              holders. The end result is either CT_ACK (event is permitted),
1576  *              CT_NACK (event is blocked) or CT_NONE (no contract owner)
1577  *              responded. This result is returned back to the caller.
1578  *      d. All other events
1579  *              - If the event was asynchronous (i.e. not negotiated) or
1580  *              a contract was not broken return CT_ACK to the caller.
1581  */
1582 static uint_t
1583 contract_device_publish(dev_info_t *dip, dev_t dev, int spec_type,
1584     uint_t evtype, nvlist_t *tnvl)
1585 {
1586         cont_device_t *ctd;
1587         uint_t result = CT_NONE;
1588         uint64_t evid = 0;
1589         uint64_t nevid = 0;
```



```
1590         char *path = NULL;
1591         int negend;
1592         int match;
1593         int sync = 0;
1594         contract_t *ct;
1595         ct_kevent_t *event;
1596         nvlist_t *nvl;
1597         int broken = 0;

1599         ASSERT(dip);
1600         ASSERT(dev != NODEV && dev != DDI_DEV_T_NONE);
1601         ASSERT((dev == DDI_DEV_T_ANY && spec_type == 0) ||
1602             (spec_type == S_IFBLK || spec_type == S_IFCHR));
1603         ASSERT(evtype == 0 || (evtype & CT_DEV_ALLEVENT));

1605         /* Is this a synchronous state change ? */
1606         if (evtype != CT_EV_NEGEND) {
1607                 sync = is_sync_neg(get_state(dip), evtype);
1608                 /* NOP if unsupported transition */
1609                 if (sync == -2 || sync == -1) {
1610                         DEVI(dip)->devi_flags |= DEVI_CT_NOP;
1611                         result = (sync == -2) ? CT_ACK : CT_NONE;
1612                         goto out;
1613                 }
1614                 CT_DEBUG((CE_NOTE, "publish: is%s sync state change",
1615                     sync ? "" : " not"));
1616         } else if (DEVI(dip)->devi_flags & DEVI_CT_NOP) {
1617                 DEVI(dip)->devi_flags &= ~DEVI_CT_NOP;
1618                 result = CT_ACK;
1619                 goto out;
1620         }

1622         path = kmem_alloc(MAXPATHLEN, KM_SLEEP);
1623         (void) ddi_pathname(dip, path);

1625         mutex_enter(&(DEVI(dip)->devi_ct_lock));

1627         /*
1628          * Negotiation end - set the state of the device in the contract
1629          */
1630         if (evtype == CT_EV_NEGEND) {
1631                 CT_DEBUG((CE_NOTE, "publish: negend: setting cond state"));
1632                 set_cond_state(dip);
1633         }

1635         /*
1636          * If this device didn't go through negotiation, don't publish
1637          * a NEGEND event - simply release the barrier to allow other
1638          * device events in.
1639          */
1640         negend = 0;
1641         if (evtype == CT_EV_NEGEND && !DEVI(dip)->devi_ct_neg) {
1642                 CT_DEBUG((CE_NOTE, "publish: no negend reqd. release barrier"));
1643                 ct_barrier_release(dip);
1644                 mutex_exit(&(DEVI(dip)->devi_ct_lock));
1645                 result = CT_ACK;
1646                 goto out;
1647         } else if (evtype == CT_EV_NEGEND) {
1648                 /*
1649                  * There are negotiated contract breakages that
1650                  * need a NEGEND event
1651                  */
1652                 ASSERT(ct_barrier_held(dip));
1653                 negend = 1;
1654                 CT_DEBUG((CE_NOTE, "publish: setting negend flag"));
1655         } else {
```

```
1656                 /*
1657                  * This is a new event, not a NEGEND event. Wait for previous
1658                  * contract events to complete.
1659                  */
1660                 ct_barrier_acquire(dip);
1661         }


1664         match = 0;
1665         for (ctd = list_head(&(DEVI(dip)->devi_ct)); ctd != NULL;
1666             ctd = list_next(&(DEVI(dip)->devi_ct), ctd)) {

1668                 ctid_t ctid;
1669                 size_t len = strlen(path);

1671                 mutex_enter(&ctd->cond_contract.ct_lock);

1673                 ASSERT(ctd->cond_dip == dip);
1674                 ASSERT(ctd->cond_minor);
1675                 ASSERT(strncmp(ctd->cond_minor, path, len) == 0 &&
1676                     ctd->cond_minor[len] == ':');

1678                 if (dev != DDI_DEV_T_ANY && dev != ctd->cond_devt) {
1679                         mutex_exit(&ctd->cond_contract.ct_lock);
1680                         continue;
1681                 }
1682                 if (dev != DDI_DEV_T_ANY && spec_type != ctd->cond_spec) {
1683                         mutex_exit(&ctd->cond_contract.ct_lock);
1684                         continue;
1685                 }

1687                 /* We have a matching contract */
1688                 match = 1;
1689                 ctid = ctd->cond_contract.ct_id;
1690                 CT_DEBUG((CE_NOTE, "publish: found matching contract: %d",
1691                     ctid));

1693                 /*
1694                  * There are 4 possible cases
1695                  * 1. A contract is broken (dev not in acceptable state) and
1696                  *    the state change is synchronous - start negotiation
1697                  *    by sending a CTE_NEG critical event.
1698                  * 2. A contract is broken and the state change is
1699                  *    asynchronous - just send a critical event and
1700                  *    break the contract.
1701                  * 3. Contract is not broken, but consumer has subscribed
1702                  *    to the event as a critical or informative event
1703                  *    - just send the appropriate event
1704                  * 4. contract waiting for negend event - just send the critical
1705                  *    NEGEND event.
1706                  */
1707                 broken = 0;
1708                 if (!negend && !(evtype & ctd->cond_aset)) {
1709                         broken = 1;
1710                         CT_DEBUG((CE_NOTE, "publish: Contract broken: %d",
1711                             ctid));
1712                 }

1714                 /*
1715                  * Don't send event if
1716                  *      - contract is not broken AND
1717                  *      - contract holder has not subscribed to this event AND
1718                  *      - contract not waiting for a NEGEND event
1719                  */
1720                 if (!broken && !EVSENDP(ctd, evtype) &&
1721                     !ctd->cond_neg) {
```



```
1722                         CT_DEBUG((CE_NOTE, "contract_device_publish(): "
1723                             "contract (%d): no publish reqd: event %d",
1724                             ctd->cond_contract.ct_id, evtype));
1725                         mutex_exit(&ctd->cond_contract.ct_lock);
1726                         continue;
1727                 }

1729                 /*
1730                  * Note: need to kmem_zalloc() the event so mutexes are
1731                  * initialized automatically
1732                  */
1733                 ct = &ctd->cond_contract;
1734                 event = kmem_zalloc(sizeof (ct_kevent_t), KM_SLEEP);
1735                 event->cte_type = evtype;

1737                 if (broken && sync) {
1738                         CT_DEBUG((CE_NOTE, "publish: broken + sync: "
1739                             "ctid: %d", ctid));
1740                         ASSERT(!negend);
1741                         ASSERT(ctd->cond_currev_id == 0);
1742                         ASSERT(ctd->cond_currev_type == 0);
1743                         ASSERT(ctd->cond_currev_ack == 0);
1744                         ASSERT(ctd->cond_neg == 0);
1745                         if (ctd->cond_noneg) {
1746                                 /* Nothing to publish. Event has been blocked */
1747                                 CT_DEBUG((CE_NOTE, "publish: sync and noneg:"
1748                                     "not publishing blocked ev: ctid: %d",
1749                                     ctid));
1750                                 result = CT_NACK;
1751                                 kmem_free(event, sizeof (ct_kevent_t));
1752                                 mutex_exit(&ctd->cond_contract.ct_lock);
1753                                 continue;
1754                         }
1755                         event->cte_flags = CTE_NEG; /* critical neg. event */
1756                         ctd->cond_currev_type = event->cte_type;
1757                         ct_barrier_incr(dip);
1758                         DEVI(dip)->devi_ct_neg = 1; /* waiting for negend */
1759                         ctd->cond_neg = 1;
1760                 } else if (broken && !sync) {
1761                         CT_DEBUG((CE_NOTE, "publish: broken + async: ctid: %d",
1762                             ctid));
1763                         ASSERT(!negend);
1764                         ASSERT(ctd->cond_currev_id == 0);
1765                         ASSERT(ctd->cond_currev_type == 0);
1766                         ASSERT(ctd->cond_currev_ack == 0);
1767                         ASSERT(ctd->cond_neg == 0);
1768                         event->cte_flags = 0; /* critical event */
1769                 } else if (EVSENDP(ctd, event->cte_type)) {
1770                         CT_DEBUG((CE_NOTE, "publish: event suscrib: ctid: %d",
1771                             ctid));
1772                         ASSERT(!negend);
1773                         ASSERT(ctd->cond_currev_id == 0);
1774                         ASSERT(ctd->cond_currev_type == 0);
1775                         ASSERT(ctd->cond_currev_ack == 0);
1776                         ASSERT(ctd->cond_neg == 0);
1777                         event->cte_flags = EVINFOP(ctd, event->cte_type) ?
1778                             CTE_INFO : 0;
1779                 } else if (ctd->cond_neg) {
1780                         CT_DEBUG((CE_NOTE, "publish: NEGEND: ctid: %d", ctid));
1781                         ASSERT(negend);
1782                         ASSERT(ctd->cond_noneg == 0);
1783                         nevid = ctd->cond_contract.ct_nevent ?
1784                             ctd->cond_contract.ct_nevent->cte_id : 0;
1785                         ASSERT(ctd->cond_currev_id == nevid);
1786                         event->cte_flags = 0;   /* NEGEND is always critical */
1787                         ctd->cond_currev_id = 0;
```

```
1788                    ctd->cond_currev_type = 0;
1789                    ctd->cond_currev_ack = 0;
1790                    ctd->cond_neg = 0;
1791            } else {
1792                    CT_DEBUG((CE_NOTE, "publish: not publishing event for "
1793                        "ctid: %d, evtype: %d",
1794                        ctd->cond_contract.ct_id, event->cte_type));
1795                    ASSERT(!negend);
1796                    ASSERT(ctd->cond_currev_id == 0);
1797                    ASSERT(ctd->cond_currev_type == 0);
1798                    ASSERT(ctd->cond_currev_ack == 0);
1799                    ASSERT(ctd->cond_neg == 0);
1800                    kmem_free(event, sizeof (ct_kevent_t));
1801                    mutex_exit(&ctd->cond_contract.ct_lock);
1802                    continue;
1803            }

1805            nvl = NULL;
1806            if (tnvl) {
1807                    VERIFY(nvlist_dup(tnvl, &nvl, 0) == 0);
1808                    if (negend) {
1809                            int32_t newct = 0;
1810                            ASSERT(ctd->cond_noneg == 0);
1811                            VERIFY(nvlist_add_uint64(nvl, CTS_NEVID, nevid)
1812                                == 0);
1813                            VERIFY(nvlist_lookup_int32(nvl, CTS_NEWCT,
1814                                &newct) == 0);
1815                            VERIFY(nvlist_add_int32(nvl, CTS_NEWCT,
1816                                newct == 1 ? 0 :
1817                                ctd->cond_contract.ct_id) == 0);
1818                            CT_DEBUG((CE_NOTE, "publish: negend: ctid: %d "
1819                                "CTS_NEVID: %llu, CTS_NEWCT: %s",
1820                                ctid, (unsigned long long)nevid,
1821                                newct ? "success" : "failure"));

1823                    }
1824            }

1826            if (ctd->cond_neg) {
1827                    ASSERT(ctd->cond_contract.ct_ntime.ctm_start == -1);
1828                    ASSERT(ctd->cond_contract.ct_qtime.ctm_start == -1);
1829                    ctd->cond_contract.ct_ntime.ctm_start = ddi_get_lbolt();
1830                    ctd->cond_contract.ct_qtime.ctm_start =
1831                        ctd->cond_contract.ct_ntime.ctm_start;
1832            }

1834            /*
1835             * by holding the dip's devi_ct_lock we ensure that
1836             * all ACK/NACKs are held up until we have finished
1837             * publishing to all contracts.
1838             */
1839            mutex_exit(&ctd->cond_contract.ct_lock);
1840            evid = cte_publish_all(ct, event, nvl, NULL);
1841            mutex_enter(&ctd->cond_contract.ct_lock);

1843            if (ctd->cond_neg) {
1844                    ASSERT(!negend);
1845                    ASSERT(broken);
1846                    ASSERT(sync);
1847                    ASSERT(!ctd->cond_noneg);
1848                    CT_DEBUG((CE_NOTE, "publish: sync break, setting evid"
1849                        ": %d", ctid));
1850                    ctd->cond_currev_id = evid;
1851            } else if (negend) {
1852                    ctd->cond_contract.ct_ntime.ctm_start = -1;
1853                    ctd->cond_contract.ct_qtime.ctm_start = -1;
```



```
1854                    }
1855                    mutex_exit(&ctd->cond_contract.ct_lock);
1856            }

1858            /*
1859             * If "negend" set counter back to initial state (-1) so that
1860             * other events can be published. Also clear the negotiation flag
1861             * on dip.
1862             *
1863             * 0 .. n are used for counting.
1864             * -1 indicates counter is available for use.
1865             */
1866            if (negend) {
1867                    /*
1868                     * devi_ct_count not necessarily 0. We may have
1869                     * timed out in which case, count will be non-zero.
1870                     */
1871                    ct_barrier_release(dip);
1872                    DEVI(dip)->devi_ct_neg = 0;
1873                    CT_DEBUG((CE_NOTE, "publish: negend: reset dip state: dip=%p",
1874                        (void *)dip));
1875            } else if (DEVI(dip)->devi_ct_neg) {
1876                    ASSERT(match);
1877                    ASSERT(!ct_barrier_empty(dip));
1878                    CT_DEBUG((CE_NOTE, "publish: sync count=%d, dip=%p",
1879                        DEVI(dip)->devi_ct_count, (void *)dip));
1880            } else {
1881                    /*
1882                     * for non-negotiated events or subscribed events or no
1883                     * matching contracts
1884                     */
1885                    ASSERT(ct_barrier_empty(dip));
1886                    ASSERT(DEVI(dip)->devi_ct_neg == 0);
1887                    CT_DEBUG((CE_NOTE, "publish: async/non-nego/subscrib/no-match: "
1888                        "dip=%p", (void *)dip));

1890                    /*
1891                     * only this function when called from contract_device_negend()
1892                     * can reset the counter to READY state i.e. -1. This function
1893                     * is so called for every event whether a NEGEND event is needed
1894                     * or not, but the negend event is only published if the event
1895                     * whose end they signal is a negotiated event for the contract.
1896                     */
1897            }

1899            if (!match) {
1900                    /* No matching contracts */
1901                    CT_DEBUG((CE_NOTE, "publish: No matching contract"));
1902                    result = CT_NONE;
1903            } else if (result == CT_NACK) {
1904                    /* a non-negotiable contract exists and this is a neg. event */
1905                    CT_DEBUG((CE_NOTE, "publish: found 1 or more NONEG contract"));
1906                    (void) wait_for_acks(dip, dev, spec_type, evtype);
1907            } else if (DEVI(dip)->devi_ct_neg) {
1908                    /* one or more contracts going through negotations  */
1909                    CT_DEBUG((CE_NOTE, "publish: sync contract: waiting"));
1910                    result = wait_for_acks(dip, dev, spec_type, evtype);
1911            } else {
1912                    /* no negotiated contracts or no broken contracts or NEGEND */
1913                    CT_DEBUG((CE_NOTE, "publish: async/no-break/negend"));
1914                    result = CT_ACK;
1915            }

1917            /*
1918             * Release the lock only now so that the only point where we
1919             * drop the lock is in wait_for_acks(). This is so that we don't
```

```
1920          * miss cv_signal/cv_broadcast from contract holders
1921          */
1922         CT_DEBUG((CE_NOTE, "publish: dropping devi_ct_lock"));
1923         mutex_exit(&(DEVI(dip)->devi_ct_lock));

1925 out:
1926         if (tnvl)
1926         nvlist_free(tnvl);
1927         if (path)
1928                 kmem_free(path, MAXPATHLEN);


1931         CT_DEBUG((CE_NOTE, "publish: result = %s", result_str(result)));
1932         return (result);
1933 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   24334 Mon Feb 15 12:56:11 2016
new/usr/src/uts/common/fs/dev/sdev_ncache.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 279 /*
 280  * Pack internal format cache data to a single nvlist.
 281  * Used when writing the nvlist file.
 282  * Note this is called indirectly by the nvpflush daemon.
 283  */
 284 static int
 285 sdev_ncache_pack_list(nvf_handle_t fd, nvlist_t **ret_nvl)
 286 {
 287         nvlist_t        *nvl, *sub_nvl;
 288         nvp_devname_t   *np;
 289         int             rval;
 290         list_t          *listp;

 292         ASSERT(fd == sdevfd_handle);
 293         ASSERT(RW_WRITE_HELD(nvf_lock(fd)));

 295         rval = nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP);
 296         if (rval != 0) {
 297                 nvf_error("%s: nvlist alloc error %d\n",
 298                     nvf_cache_name(fd), rval);
 299                 return (DDI_FAILURE);
 300         }

 302         listp = nvf_list(sdevfd_handle);
 303         if ((np = list_head(listp)) != NULL) {
 304                 ASSERT(list_next(listp, np) == NULL);

 306                 rval = nvlist_alloc(&sub_nvl, NV_UNIQUE_NAME, KM_SLEEP);
 307                 if (rval != 0) {
 308                         nvf_error("%s: nvlist alloc error %d\n",
 309                             nvf_cache_name(fd), rval);
 310                         sub_nvl = NULL;
 311                         goto err;
 312                 }

 314                 rval = nvlist_add_string_array(sub_nvl,
 315                     DP_DEVNAME_NCACHE_ID, np->nvp_paths, np->nvp_npaths);
 316                 if (rval != 0) {
 317                         nvf_error("%s: nvlist add error %d (sdev)\n",
 318                             nvf_cache_name(fd), rval);
 319                         goto err;
 320                 }

 322                 rval = nvlist_add_int32_array(sub_nvl,
 323                     DP_DEVNAME_NC_EXPIRECNT_ID,
 324                     np->nvp_expirecnts, np->nvp_npaths);
 325                 if (rval != 0) {
 326                         nvf_error("%s: nvlist add error %d (sdev)\n",
 327                             nvf_cache_name(fd), rval);
 328                         goto err;
 329                 }

 331                 rval = nvlist_add_nvlist(nvl, DP_DEVNAME_ID, sub_nvl);
 332                 if (rval != 0) {
 333                         nvf_error("%s: nvlist add error %d (sublist)\n",
 334                             nvf_cache_name(fd), rval);
 335                         goto err;
 336                 }
 337                 nvlist_free(sub_nvl);
```



```
 338         }

 340         *ret_nvl = nvl;
 341         return (DDI_SUCCESS);

 343 err:
 344         if (sub_nvl)
 344         nvlist_free(sub_nvl);
 345         nvlist_free(nvl);
 346         *ret_nvl = NULL;
 347         return (DDI_FAILURE);
 348 }
______unchanged_portion_omitted_
```

```
**********************************************************
   73712 Mon Feb 15 12:56:11 2016
new/usr/src/uts/common/fs/dev/sdev_subr.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 150 kmem_cache_t    *sdev_node_cache;       /* sdev_node cache */
 151 int             devtype;                /* fstype */

 153 /* static */
 154 static struct vnodeops *sdev_get_vop(struct sdev_node *);
 155 static void sdev_set_no_negcache(struct sdev_node *);
 156 static fs_operation_def_t *sdev_merge_vtab(const fs_operation_def_t []);
 157 static void sdev_free_vtab(fs_operation_def_t *);

 159 static void
 160 sdev_prof_free(struct sdev_node *dv)
 161 {
 162         ASSERT(!SDEV_IS_GLOBAL(dv));
 163         if (dv->sdev_prof.dev_name)
 163         nvlist_free(dv->sdev_prof.dev_name);
 165         if (dv->sdev_prof.dev_map)
 164         nvlist_free(dv->sdev_prof.dev_map);
 167         if (dv->sdev_prof.dev_symlink)
 165         nvlist_free(dv->sdev_prof.dev_symlink);
 169         if (dv->sdev_prof.dev_glob_incdir)
 166         nvlist_free(dv->sdev_prof.dev_glob_incdir);
 171         if (dv->sdev_prof.dev_glob_excdir)
 167         nvlist_free(dv->sdev_prof.dev_glob_excdir);
 168         bzero(&dv->sdev_prof, sizeof (dv->sdev_prof));
 169 }
______unchanged_portion_omitted_
```

```
**********************************************************
  100266 Mon Feb 15 12:56:11 2016
new/usr/src/uts/common/fs/nfs/nfs4_state.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

1049 /*
1050  * DSS: distributed stable storage.
1051  * Unpack the list of paths passed by nfsd.
1052  * Use nvlist_alloc(9F) to manage the data.
1053  * The caller is responsible for allocating and freeing the buffer.
1054  */
1055 int
1056 rfs4_dss_setpaths(char *buf, size_t buflen)
1057 {
1058         int error;

1060         /*
1061          * If this is a "warm start", i.e. we previously had DSS paths,
1062          * preserve the old paths.
1063          */
1064         if (rfs4_dss_paths != NULL) {
1065                 /*
1066                  * Before we lose the ptr, destroy the nvlist and pathnames
1067                  * array from the warm start before this one.
1068                  */
1069                 if (rfs4_dss_oldpaths)
1069                 nvlist_free(rfs4_dss_oldpaths);
1070                 rfs4_dss_oldpaths = rfs4_dss_paths;
1071         }

1073         /* unpack the buffer into a searchable nvlist */
1074         error = nvlist_unpack(buf, buflen, &rfs4_dss_paths, KM_SLEEP);
1075         if (error)
1076                 return (error);

1078         /*
1079          * Search the nvlist for the pathnames nvpair (which is the only nvpair
1080          * in the list, and record its location.
1081          */
1082         error = nvlist_lookup_string_array(rfs4_dss_paths, NFS4_DSS_NVPAIR_NAME,
1083             &rfs4_dss_newpaths, &rfs4_dss_numnewpaths);
1084         return (error);
1085 }
______unchanged_portion_omitted_


1412 /*
1413  * Used at server shutdown to cleanup all of the NFSv4 server's structures
1414  * and other state.
1415  */
1416 void
1417 rfs4_state_fini()
1418 {
1419         rfs4_database_t *dbp;

1421         mutex_enter(&rfs4_state_lock);

1423         if (rfs4_server_state == NULL) {
1424                 mutex_exit(&rfs4_state_lock);
1425                 return;
1426         }

1428         rfs4_client_clrst = NULL;
```



```
1430         rfs4_set_deleg_policy(SRV_NEVER_DELEGATE);
1431         dbp = rfs4_server_state;
1432         rfs4_server_state = NULL;

1434         /*
1435          * Cleanup the CPR callback.
1436          */
1437         if (cpr_id)
1438                 (void) callb_delete(cpr_id);

1440         rw_destroy(&rfs4_findclient_lock);

1442         /* First stop all of the reaper threads in the database */
1443         rfs4_database_shutdown(dbp);
1444         /* clean up any dangling stable storage structures */
1445         rfs4_ss_fini();
1446         /* Now actually destroy/release the database and its tables */
1447         rfs4_database_destroy(dbp);

1449         /* Reset the cache timers for next time */
1450         rfs4_client_cache_time = 0;
1451         rfs4_openowner_cache_time = 0;
1452         rfs4_state_cache_time = 0;
1453         rfs4_lo_state_cache_time = 0;
1454         rfs4_lockowner_cache_time = 0;
1455         rfs4_file_cache_time = 0;
1456         rfs4_deleg_state_cache_time = 0;

1458         mutex_exit(&rfs4_state_lock);

1460         /* destroy server instances and current instance ptr */
1461         rfs4_servinst_destroy_all();

1463         /* reset the "first NFSv4 request" status */
1464         rfs4_seen_first_compound = 0;

1466         /* DSS: distributed stable storage */
1468         if (rfs4_dss_oldpaths)
1467         nvlist_free(rfs4_dss_oldpaths);
1470         if (rfs4_dss_paths)
1468         nvlist_free(rfs4_dss_paths);
1469         rfs4_dss_paths = rfs4_dss_oldpaths = NULL;
1470 }
______unchanged_portion_omitted_
```

```
**********************************************************
   31115 Mon Feb 15 12:56:12 2016
new/usr/src/uts/common/fs/smbsrv/smb_kshare.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 364 /*
 365  * A list of shares in nvlist format can be sent down
 366  * from userspace thourgh the IOCTL interface. The nvlist
 367  * is unpacked here and all the shares in the list will
 368  * be exported.
 369  */
 370 int
 371 smb_kshare_export_list(smb_ioc_share_t *ioc)
 372 {
 373         smb_server_t    *sv = NULL;
 374         nvlist_t        *shrlist = NULL;
 375         nvlist_t         *share;
 376         nvpair_t         *nvp;
 377         smb_kshare_t     *shr;
 378         char            *shrname;
 379         int             rc;

 381         if ((rc = smb_server_lookup(&sv)) != 0)
 382                 return (rc);

 384         if (!smb_export_isready(sv)) {
 385                 rc = ENOTACTIVE;
 386                 goto out;
 387         }

 389         rc = nvlist_unpack(ioc->shr, ioc->shrlen, &shrlist, KM_SLEEP);
 390         if (rc != 0)
 391                 goto out;

 393         for (nvp = nvlist_next_nvpair(shrlist, NULL); nvp != NULL;
 394             nvp = nvlist_next_nvpair(shrlist, nvp)) {

 396                 /*
 397                  * Since this loop can run for a while we want to exit
 398                  * as soon as the server state is anything but RUNNING
 399                  * to allow shutdown to proceed.
 400                  */
 401                 if (sv->sv_state != SMB_SERVER_STATE_RUNNING)
 402                         goto out;

 404                 if (nvpair_type(nvp) != DATA_TYPE_NVLIST)
 405                         continue;

 407                 shrname = nvpair_name(nvp);
 408                 ASSERT(shrname);

 410                 if ((rc = nvpair_value_nvlist(nvp, &share)) != 0) {
 411                         cmn_err(CE_WARN, "export[%s]: failed accessing",
 412                             shrname);
 413                         continue;
 414                 }

 416                 if ((shr = smb_kshare_decode(share)) == NULL) {
 417                         cmn_err(CE_WARN, "export[%s]: failed decoding",
 418                             shrname);
 419                         continue;
 420                 }

 422                 /* smb_kshare_export consumes shr so it's not leaked */
```



```
 423                 if ((rc = smb_kshare_export(sv, shr)) != 0) {
 424                         smb_kshare_destroy(shr);
 425                         continue;
 426                 }
 427         }
 428         rc = 0;

 430 out:
 431         if (shrlist != NULL)
 431         nvlist_free(shrlist);
 432         smb_server_release(sv);
 433         return (rc);
 434 }

 436 /*
 437  * This function is invoked when a share is disabled to disconnect trees
 438  * and close files.  Cleaning up may involve VOP and/or VFS calls, which
 439  * may conflict/deadlock with stuck threads if something is amiss with the
 440  * file system.  Queueing the request for asynchronous processing allows the
 441  * call to return immediately so that, if the unshare is being done in the
 442  * context of a forced unmount, the forced unmount will always be able to
 443  * proceed (unblocking stuck I/O and eventually allowing all blocked unshare
 444  * processes to complete).
 445  *
 446  * The path lookup to find the root vnode of the VFS in question and the
 447  * release of this vnode are done synchronously prior to any associated
 448  * unmount.  Doing these asynchronous to an associated unmount could run
 449  * the risk of a spurious EBUSY for a standard unmount or an EIO during
 450  * the path lookup due to a forced unmount finishing first.
 451  */
 452 int
 453 smb_kshare_unexport_list(smb_ioc_share_t *ioc)
 454 {
 455         smb_server_t    *sv = NULL;
 456         smb_unshare_t   *ux;
 457         nvlist_t        *shrlist = NULL;
 458         nvpair_t        *nvp;
 459         boolean_t       unexport = B_FALSE;
 460         char            *shrname;
 461         int             rc;

 463         if ((rc = smb_server_lookup(&sv)) != 0)
 464                 return (rc);

 466         if ((rc = nvlist_unpack(ioc->shr, ioc->shrlen, &shrlist, 0)) != 0)
 467                 goto out;

 469         for (nvp = nvlist_next_nvpair(shrlist, NULL); nvp != NULL;
 470             nvp = nvlist_next_nvpair(shrlist, nvp)) {
 471                 if (nvpair_type(nvp) != DATA_TYPE_NVLIST)
 472                         continue;

 474                 shrname = nvpair_name(nvp);
 475                 ASSERT(shrname);

 477                 if ((rc = smb_kshare_unexport(sv, shrname)) != 0)
 478                         continue;

 480                 ux = kmem_cache_alloc(smb_kshare_cache_unexport, KM_SLEEP);
 481                 (void) strlcpy(ux->us_sharename, shrname, MAXNAMELEN);

 483                 smb_slist_insert_tail(&sv->sv_export.e_unexport_list, ux);
 484                 unexport = B_TRUE;
 485         }

 487         if (unexport)
```

```
488                 smb_thread_signal(&sv->sv_export.e_unexport_thread);
489         rc = 0;

491 out:
493         if (shrlist != NULL)
492         nvlist_free(shrlist);
493         smb_server_release(sv);
494         return (rc);
495 }
```

unchanged_portion_omitted_

****
182245 Mon Feb 15 12:56:12 2016
new/usr/src/uts/common/fs/zfs/spa.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
5986 static void
5987 spa_sync_config_object(spa_t *spa, dmu_tx_t *tx)
5988 {
5989         nvlist_t *config;

5991         if (list_is_empty(&spa->spa_config_dirty_list))
5992                 return;

5994         spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);

5996         config = spa_config_generate(spa, spa->spa_root_vdev,
5997             dmu_tx_get_txg(tx), B_FALSE);

5999         /*
6000          * If we're upgrading the spa version then make sure that
6001          * the config object gets updated with the correct version.
6002          */
6003         if (spa->spa_ubsync.ub_version < spa->spa_uberblock.ub_version)
6004                 fnvlist_add_uint64(config, ZPOOL_CONFIG_VERSION,
6005                     spa->spa_uberblock.ub_version);

6007         spa_config_exit(spa, SCL_STATE, FTAG);

6009         if (spa->spa_config_syncing)
6009         nvlist_free(spa->spa_config_syncing);
6010         spa->spa_config_syncing = config;

6012         spa_sync_nvlist(spa, spa->spa_config_object, config, tx);
6013 }
```

_____***unchanged_portion_omitted_***

****
15411 Mon Feb 15 12:56:12 2016
new/usr/src/uts/common/fs/zfs/spa_config.c
patch tsoome-feedback
****
_____***unchanged_portion_omitted_***

```
 343 void
 344 spa_config_set(spa_t *spa, nvlist_t *config)
 345 {
 346         mutex_enter(&spa->spa_props_lock);
 347         if (spa->spa_config != NULL)
 347         nvlist_free(spa->spa_config);
 348         spa->spa_config = config;
 349         mutex_exit(&spa->spa_props_lock);
 350 }
```

_____***unchanged_portion_omitted_***

****
154666 Mon Feb 15 12:56:12 2016
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
1528 static int
1529 zfs_ioc_pool_import(zfs_cmd_t *zc)
1530 {
1531         nvlist_t *config, *props = NULL;
1532         uint64_t guid;
1533         int error;

1535         if ((error = get_nvlist(zc->zc_nvlist_conf, zc->zc_nvlist_conf_size,
1536             zc->zc_iflags, &config)) != 0)
1537                 return (error);

1539         if (zc->zc_nvlist_src_size != 0 && (error =
1540             get_nvlist(zc->zc_nvlist_src, zc->zc_nvlist_src_size,
1541             zc->zc_iflags, &props))) {
1542                 nvlist_free(config);
1543                 return (error);
1544         }

1546         if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID, &guid) != 0 ||
1547             guid != zc->zc_guid)
1548                 error = SET_ERROR(EINVAL);
1549         else
1550                 error = spa_import(zc->zc_name, config, props, zc->zc_cookie);

1552         if (zc->zc_nvlist_dst != 0) {
1553                 int err;

1555                 if ((err = put_nvlist(zc, config)) != 0)
1556                         error = err;
1557         }

1559         nvlist_free(config);

1561         if (props)
1561         nvlist_free(props);

1563         return (error);
1564 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   97055 Mon Feb 15 12:56:12 2016
new/usr/src/uts/common/io/comstar/lu/stmf_sbd/sbd.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

3590 int
3591 sbd_zvolget(char *zvol_name, char **comstarprop)
3592 {
3593         ldi_handle_t    zfs_lh;
3594         nvlist_t        *nv = NULL, *nv2;
3595         zfs_cmd_t       *zc;
3596         char            *ptr;
3597         int size = 1024;
3598         int unused;
3599         int rc;

3601         if ((rc = ldi_open_by_name("/dev/zfs", FREAD | FWRITE, kcred,
3602             &zfs_lh, sbd_zfs_ident)) != 0) {
3603                 cmn_err(CE_WARN, "ldi_open %d", rc);
3604                 return (ENXIO);
3605         }

3607         zc = kmem_zalloc(sizeof (zfs_cmd_t), KM_SLEEP);
3608         (void) strlcpy(zc->zc_name, zvol_name, sizeof (zc->zc_name));
3609 again:
3610         zc->zc_nvlist_dst = (uint64_t)(intptr_t)kmem_alloc(size,
3611             KM_SLEEP);
3612         zc->zc_nvlist_dst_size = size;
3613         rc = ldi_ioctl(zfs_lh, ZFS_IOC_OBJSET_STATS, (intptr_t)zc,
3614             FKIOCTL, kcred, &unused);
3615         /*
3616          * ENOMEM means the list is larger than what we've allocated
3617          * ldi_ioctl will fail with ENOMEM only once
3618          */
3619         if (rc == ENOMEM) {
3620                 int newsize;
3621                 newsize = zc->zc_nvlist_dst_size;
3622                 kmem_free((void *)(uintptr_t)zc->zc_nvlist_dst, size);
3623                 size = newsize;
3624                 goto again;
3625         } else if (rc != 0) {
3626                 goto out;
3627         }
3628         rc = nvlist_unpack((char *)(uintptr_t)zc->zc_nvlist_dst,
3629             zc->zc_nvlist_dst_size, &nv, 0);
3630         ASSERT(rc == 0);        /* nvlist_unpack should not fail */
3631         if ((rc = nvlist_lookup_nvlist(nv, "stmf_sbd_lu", &nv2)) == 0) {
3632                 rc = nvlist_lookup_string(nv2, ZPROP_VALUE, &ptr);
3633                 if (rc != 0) {
3634                         cmn_err(CE_WARN, "couldn't get value");
3635                 } else {
3636                         *comstarprop = kmem_alloc(strlen(ptr) + 1,
3637                             KM_SLEEP);
3638                         (void) strcpy(*comstarprop, ptr);
3639                 }
3640         }
3641 out:
3642         if (nv != NULL)
3642         nvlist_free(nv);
3643         kmem_free((void *)(uintptr_t)zc->zc_nvlist_dst, size);
3644         kmem_free(zc, sizeof (zfs_cmd_t));
3645         (void) ldi_close(zfs_lh, FREAD|FWRITE, kcred);

3647         return (rc);
```



```
3648 }
______unchanged_portion_omitted_
```

```
**********************************************************
   94801 Mon Feb 15 12:56:13 2016
new/usr/src/uts/common/io/comstar/port/iscsit/iscsit.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 437 /* ARGSUSED */
 438 static int
 439 iscsit_drv_ioctl(dev_t drv, int cmd, intptr_t argp, int flag, cred_t *cred,
 440     int *retval)
 441 {
 442         iscsit_ioc_set_config_t         setcfg;
 443         iscsit_ioc_set_config32_t       setcfg32;
 444         char                            *cfg_pnvlist = NULL;
 445         nvlist_t                        *cfg_nvlist = NULL;
 446         it_config_t                     *cfg = NULL;
 447         idm_status_t                    idmrc;
 448         int                             rc = 0;

 450         if (drv_priv(cred) != 0) {
 451                 return (EPERM);
 452         }

 454         mutex_enter(&iscsit_global.global_state_mutex);

 456         /*
 457          * Validate ioctl requests against global service state
 458          */
 459         switch (iscsit_global.global_svc_state) {
 460         case ISE_ENABLED:
 461                 if (cmd == ISCSIT_IOC_DISABLE_SVC) {
 462                         iscsit_global.global_svc_state = ISE_DISABLING;
 463                 } else if (cmd == ISCSIT_IOC_ENABLE_SVC) {
 464                         /* Already enabled */
 465                         mutex_exit(&iscsit_global.global_state_mutex);
 466                         return (0);
 467                 } else {
 468                         iscsit_global.global_svc_state = ISE_BUSY;
 469                 }
 470                 break;
 471         case ISE_DISABLED:
 472                 if (cmd == ISCSIT_IOC_ENABLE_SVC) {
 473                         iscsit_global.global_svc_state = ISE_ENABLING;
 474                 } else if (cmd == ISCSIT_IOC_DISABLE_SVC) {
 475                         /* Already disabled */
 476                         mutex_exit(&iscsit_global.global_state_mutex);
 477                         return (0);
 478                 } else {
 479                         rc = EFAULT;
 480                 }
 481                 break;
 482         case ISE_BUSY:
 483         case ISE_ENABLING:
 484         case ISE_DISABLING:
 485                 rc = EAGAIN;
 486                 break;
 487         case ISE_DETACHED:
 488         default:
 489                 rc = EFAULT;
 490                 break;
 491         }

 493         mutex_exit(&iscsit_global.global_state_mutex);
 494         if (rc != 0)
 495                 return (rc);
```



```
 497         /* Handle ioctl request (enable/disable have already been handled) */
 498         switch (cmd) {
 499         case ISCSIT_IOC_SET_CONFIG:
 500                 /* Any errors must set state back to ISE_ENABLED */
 501                 switch (ddi_model_convert_from(flag & FMODELS)) {
 502                 case DDI_MODEL_ILP32:
 503                         if (ddi_copyin((void *)argp, &setcfg32,
 504                             sizeof (iscsit_ioc_set_config32_t), flag) != 0) {
 505                                 rc = EFAULT;
 506                                 goto cleanup;
 507                         }

 509                         setcfg.set_cfg_pnvlist =
 510                             (char *)((uintptr_t)setcfg32.set_cfg_pnvlist);
 511                         setcfg.set_cfg_vers = setcfg32.set_cfg_vers;
 512                         setcfg.set_cfg_pnvlist_len =
 513                             setcfg32.set_cfg_pnvlist_len;
 514                         break;
 515                 case DDI_MODEL_NONE:
 516                         if (ddi_copyin((void *)argp, &setcfg,
 517                             sizeof (iscsit_ioc_set_config_t), flag) != 0) {
 518                                 rc = EFAULT;
 519                                 goto cleanup;
 520                         }
 521                         break;
 522                 default:
 523                         rc = EFAULT;
 524                         goto cleanup;
 525                 }

 527                 /* Check API version */
 528                 if (setcfg.set_cfg_vers != ISCSIT_API_VERS0) {
 529                         rc = EINVAL;
 530                         goto cleanup;
 531                 }

 533                 /* Config is in packed nvlist format so unpack it */
 534                 cfg_pnvlist = kmem_alloc(setcfg.set_cfg_pnvlist_len,
 535                     KM_SLEEP);
 536                 ASSERT(cfg_pnvlist != NULL);

 538                 if (ddi_copyin(setcfg.set_cfg_pnvlist, cfg_pnvlist,
 539                     setcfg.set_cfg_pnvlist_len, flag) != 0) {
 540                         rc = EFAULT;
 541                         goto cleanup;
 542                 }

 544                 rc = nvlist_unpack(cfg_pnvlist, setcfg.set_cfg_pnvlist_len,
 545                     &cfg_nvlist, KM_SLEEP);
 546                 if (rc != 0) {
 547                         goto cleanup;
 548                 }

 550                 /* Translate nvlist */
 551                 rc = it_nv_to_config(cfg_nvlist, &cfg);
 552                 if (rc != 0) {
 553                         cmn_err(CE_WARN, "Configuration is invalid");
 554                         goto cleanup;
 555                 }

 557                 /* Update config */
 558                 rc = iscsit_config_merge(cfg);
 559                 /* FALLTHROUGH */

 561 cleanup:
```

```
562                 if (cfg)
563                         it_config_free_cmn(cfg);
564                 if (cfg_pnvlist)
565                         kmem_free(cfg_pnvlist, setcfg.set_cfg_pnvlist_len);
566                 if (cfg_nvlist)
566                 nvlist_free(cfg_nvlist);

568                 /*
569                  * Now that the reconfig is complete set our state back to
570                  * enabled.
571                  */
572                 mutex_enter(&iscsit_global.global_state_mutex);
573                 iscsit_global.global_svc_state = ISE_ENABLED;
574                 mutex_exit(&iscsit_global.global_state_mutex);
575                 break;
576         case ISCSIT_IOC_ENABLE_SVC: {
577                 iscsit_hostinfo_t hostinfo;

579                 if (ddi_copyin((void *)argp, &hostinfo.length,
580                     sizeof (hostinfo.length), flag) != 0) {
581                         mutex_enter(&iscsit_global.global_state_mutex);
582                         iscsit_global.global_svc_state = ISE_DISABLED;
583                         mutex_exit(&iscsit_global.global_state_mutex);
584                         return (EFAULT);
585                 }

587                 if (hostinfo.length > sizeof (hostinfo.fqhn))
588                         hostinfo.length = sizeof (hostinfo.fqhn);

590                 if (ddi_copyin((void *)((caddr_t)argp +
591                     sizeof (hostinfo.length)), &hostinfo.fqhn,
592                     hostinfo.length, flag) != 0) {
593                         mutex_enter(&iscsit_global.global_state_mutex);
594                         iscsit_global.global_svc_state = ISE_DISABLED;
595                         mutex_exit(&iscsit_global.global_state_mutex);
596                         return (EFAULT);
597                 }

599                 idmrc = iscsit_enable_svc(&hostinfo);
600                 mutex_enter(&iscsit_global.global_state_mutex);
601                 if (idmrc == IDM_STATUS_SUCCESS) {
602                         iscsit_global.global_svc_state = ISE_ENABLED;
603                 } else {
604                         rc = EIO;
605                         iscsit_global.global_svc_state = ISE_DISABLED;
606                 }
607                 mutex_exit(&iscsit_global.global_state_mutex);
608                 break;
609         }
610         case ISCSIT_IOC_DISABLE_SVC:
611                 iscsit_disable_svc();
612                 mutex_enter(&iscsit_global.global_state_mutex);
613                 iscsit_global.global_svc_state = ISE_DISABLED;
614                 mutex_exit(&iscsit_global.global_state_mutex);
615                 break;

617         default:
618                 rc = EINVAL;
619                 mutex_enter(&iscsit_global.global_state_mutex);
620                 iscsit_global.global_svc_state = ISE_ENABLED;
621                 mutex_exit(&iscsit_global.global_state_mutex);
622         }

624         return (rc);
625 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   56010 Mon Feb 15 12:56:13 2016
new/usr/src/uts/common/io/comstar/port/pppt/alua_ic_if.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_


/*
 * Function prototypes.
 */

/*
 * Helpers for msg_alloc routines, used when the msg payload is
 * the same for multiple types of messages.
 */
static stmf_ic_msg_t *stmf_ic_reg_dereg_lun_msg_alloc(
    stmf_ic_msg_type_t msg_type, uint8_t *lun_id,
    char *lu_provider_name, uint16_t cb_arg_len,
    uint8_t *cb_arg, stmf_ic_msgid_t msgid);

static stmf_ic_msg_t *stmf_ic_session_create_destroy_msg_alloc(
    stmf_ic_msg_type_t msg_type,
    stmf_scsi_session_t *session,
    stmf_ic_msgid_t msgid);

static stmf_ic_msg_t *stmf_ic_echo_request_reply_msg_alloc(
    stmf_ic_msg_type_t msg_type,
    uint32_t data_len,
    uint8_t *data,
    stmf_ic_msgid_t msgid);

/*
 * Msg free routines.
 */
static void stmf_ic_reg_port_msg_free(stmf_ic_reg_port_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);
static void stmf_ic_dereg_port_msg_free(stmf_ic_dereg_port_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);
static void stmf_ic_reg_dereg_lun_msg_free(stmf_ic_reg_dereg_lun_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);
static void stmf_ic_scsi_cmd_msg_free(stmf_ic_scsi_cmd_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);
static void stmf_ic_scsi_data_msg_free(stmf_ic_scsi_data_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);
static void stmf_ic_scsi_data_xfer_done_msg_free(
    stmf_ic_scsi_data_xfer_done_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);
static void stmf_ic_scsi_status_msg_free(stmf_ic_scsi_status_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);
static void stmf_ic_r2t_msg_free(stmf_ic_r2t_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);
static void stmf_ic_status_msg_free(stmf_ic_status_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);
static void stmf_ic_session_create_destroy_msg_free(
    stmf_ic_session_create_destroy_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);
static void stmf_ic_echo_request_reply_msg_free(
    stmf_ic_echo_request_reply_msg_t *m,
    stmf_ic_msg_construction_method_t cmethod);

/*
 * Marshaling routines.
 */
static nvlist_t *stmf_ic_msg_marshal(stmf_ic_msg_t *msg);
static int stmf_ic_reg_port_msg_marshal(nvlist_t *nvl, void *msg);
```



```
static int stmf_ic_dereg_port_msg_marshal(nvlist_t *nvl, void *msg);
static int stmf_ic_reg_dereg_lun_msg_marshal(nvlist_t *nvl, void *msg);
static int stmf_ic_scsi_cmd_msg_marshal(nvlist_t *nvl, void *msg);
static int stmf_ic_scsi_data_msg_marshal(nvlist_t *nvl, void *msg);
static int stmf_ic_scsi_data_xfer_done_msg_marshal(nvlist_t *nvl, void *msg);
static int stmf_ic_scsi_status_msg_marshal(nvlist_t *nvl, void *msg);
static int stmf_ic_r2t_msg_marshal(nvlist_t *nvl, void *msg);
static int stmf_ic_status_msg_marshal(nvlist_t *nvl, void *msg);
static int stmf_ic_session_create_destroy_msg_marshal(nvlist_t *nvl, void *msg);
static int stmf_ic_echo_request_reply_msg_marshal(nvlist_t *nvl, void *msg);
static int stmf_ic_scsi_devid_desc_marshal(nvlist_t *parent_nvl,
        char *sdid_name, scsi_devid_desc_t *sdid);
static int stmf_ic_remote_port_marshal(nvlist_t *parent_nvl,
        char *rport_name, stmf_remote_port_t *rport);

/*
 * Unmarshaling routines.
 */
static stmf_ic_msg_t *stmf_ic_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_reg_port_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_dereg_port_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_reg_dereg_lun_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_scsi_cmd_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_scsi_data_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_scsi_data_xfer_done_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_scsi_status_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_r2t_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_status_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_session_create_destroy_msg_unmarshal(nvlist_t *nvl);
static void *stmf_ic_echo_request_reply_msg_unmarshal(nvlist_t *nvl);
static scsi_devid_desc_t *stmf_ic_lookup_scsi_devid_desc_and_unmarshal(
    nvlist_t *nvl, char *field_name);
static scsi_devid_desc_t *stmf_ic_scsi_devid_desc_unmarshal(
    nvlist_t *nvl_devid);
static uint8_t *stmf_ic_uint8_array_unmarshal(nvlist_t *nvl, char *field_name,
        uint64_t len, uint8_t *buf);
static char *stmf_ic_string_unmarshal(nvlist_t *nvl, char *field_name);
static stmf_remote_port_t *stmf_ic_lookup_remote_port_and_unmarshal(
        nvlist_t *nvl, char *field_name);
static stmf_remote_port_t *stmf_ic_remote_port_unmarshal(nvlist_t *nvl);

/*
 * Transmit and recieve routines.
 */
stmf_ic_msg_status_t stmf_ic_transmit(char *buf, size_t size);

/*
 * Utilities.
 */
static stmf_ic_msg_t *stmf_ic_alloc_msg_header(stmf_ic_msg_type_t msg_type,
        stmf_ic_msgid_t msgid);
static size_t sizeof_scsi_devid_desc(int ident_length);
static char *stmf_ic_strdup(char *str);
static scsi_devid_desc_t *scsi_devid_desc_dup(scsi_devid_desc_t *did);
static stmf_remote_port_t *remote_port_dup(stmf_remote_port_t *rport);
static void scsi_devid_desc_free(scsi_devid_desc_t *did);
static inline void stmf_ic_nvlookup_warn(const char *func, char *field);

/*
 * Send a message out over the interconnect, in the process marshalling
 * the arguments.
 *
 * After being sent, the message is freed.
 */
stmf_ic_msg_status_t
stmf_ic_tx_msg(stmf_ic_msg_t *msg)
```

```
326 {
327         size_t size = 0;
328         nvlist_t *nvl = NULL;
329         char *buf = NULL;
330         int err = 0;
331         stmf_ic_msg_status_t status = STMF_IC_MSG_SUCCESS;

333         nvl = stmf_ic_msg_marshal(msg);
334         if (!nvl) {
335                 cmn_err(CE_WARN, "stmf_ic_tx_msg: marshal failed");
336                 status = STMF_IC_MSG_INTERNAL_ERROR;
337                 goto done;
338         }

340         err = nvlist_size(nvl, &size, NV_ENCODE_XDR);
341         if (err) {
342                 status = STMF_IC_MSG_INTERNAL_ERROR;
343                 goto done;
344         }

346         buf = kmem_alloc(size, KM_SLEEP);
347         err = nvlist_pack(nvl, &buf, &size, NV_ENCODE_XDR, 0);
348         if (err) {
349                 status = STMF_IC_MSG_INTERNAL_ERROR;
350                 goto done;
351         }

353         /* push the bits out on the wire */

355         status = stmf_ic_transmit(buf, size);

357 done:
358         if (nvl)
358         nvlist_free(nvl);

360         if (buf)
361                 kmem_free(buf, size);

363         stmf_ic_msg_free(msg);


366         return (status);
367 }
```

*unchanged_portion_omitted_*

```
895 /*
896  * msg free routines.
897  */
898 void
899 stmf_ic_msg_free(stmf_ic_msg_t *msg)
900 {
901         stmf_ic_msg_construction_method_t cmethod =
902             (msg->icm_nvlist ? STMF_UNMARSHAL : STMF_CONSTRUCTOR);

904         switch (msg->icm_msg_type) {
905         case STMF_ICM_REGISTER_PROXY_PORT:
906                 stmf_ic_reg_port_msg_free(
907                     (stmf_ic_reg_port_msg_t *)msg->icm_msg, cmethod);
908                 break;

910         case STMF_ICM_DEREGISTER_PROXY_PORT:
911                 stmf_ic_dereg_port_msg_free(
912                     (stmf_ic_dereg_port_msg_t *)msg->icm_msg, cmethod);
913                 break;

915         case STMF_ICM_LUN_ACTIVE:
```



```
916         case STMF_ICM_REGISTER_LUN:
917         case STMF_ICM_DEREGISTER_LUN:
918                 stmf_ic_reg_dereg_lun_msg_free(
919                     (stmf_ic_reg_dereg_lun_msg_t *)msg->icm_msg, cmethod);
920                 break;

922         case STMF_ICM_SCSI_CMD:
923                 stmf_ic_scsi_cmd_msg_free(
924                     (stmf_ic_scsi_cmd_msg_t *)msg->icm_msg, cmethod);
925                 break;

927         case STMF_ICM_SCSI_DATA:
928                 stmf_ic_scsi_data_msg_free(
929                     (stmf_ic_scsi_data_msg_t *)msg->icm_msg, cmethod);
930                 break;

932         case STMF_ICM_SCSI_DATA_XFER_DONE:
933                 stmf_ic_scsi_data_xfer_done_msg_free(
934                     (stmf_ic_scsi_data_xfer_done_msg_t *)msg->icm_msg, cmethod);
935                 break;

937         case STMF_ICM_SCSI_STATUS:
938                 stmf_ic_scsi_status_msg_free(
939                     (stmf_ic_scsi_status_msg_t *)msg->icm_msg, cmethod);
940                 break;

942         case STMF_ICM_R2T:
943                 stmf_ic_r2t_msg_free(
944                     (stmf_ic_r2t_msg_t *)msg->icm_msg, cmethod);
945                 break;

947         case STMF_ICM_STATUS:
948                 stmf_ic_status_msg_free(
949                     (stmf_ic_status_msg_t *)msg->icm_msg, cmethod);
950                 break;

952         case STMF_ICM_SESSION_CREATE:
953         case STMF_ICM_SESSION_DESTROY:
954                 stmf_ic_session_create_destroy_msg_free(
955                     (stmf_ic_session_create_destroy_msg_t *)msg->icm_msg,
956                     cmethod);
957                 break;

959         case STMF_ICM_ECHO_REQUEST:
960         case STMF_ICM_ECHO_REPLY:
961                 stmf_ic_echo_request_reply_msg_free(
962                     (stmf_ic_echo_request_reply_msg_t *)msg->icm_msg, cmethod);
963                 break;

965         case STMF_ICM_MAX_MSG_TYPE:
966                 ASSERT(0);
967                 break;

969         default:
970                 ASSERT(0);
971         }

974         if (msg->icm_nvlist)
973         nvlist_free(msg->icm_nvlist);

975         kmem_free(msg, sizeof (*msg));
976 }
```

*unchanged_portion_omitted_*

```
1096 /*
```

```
 * Marshaling routines.
 */

static nvlist_t *
stmf_ic_msg_marshal(stmf_ic_msg_t *msg)
{
        nvlist_t *nvl = NULL;
        int rc = 0;

        rc = nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP);
        if (rc)
                goto done;

        NVLIST_ADD_FIELD(uint8, msg, icm_msg_type);
        NVLIST_ADD_FIELD(uint64, msg, icm_msgid);

        switch (msg->icm_msg_type) {
        case STMF_ICM_REGISTER_PROXY_PORT:
                rc = stmf_ic_reg_port_msg_marshal(nvl, msg->icm_msg);
                break;


        case STMF_ICM_DEREGISTER_PROXY_PORT:
                rc = stmf_ic_dereg_port_msg_marshal(nvl, msg->icm_msg);
                break;

        case STMF_ICM_LUN_ACTIVE:
        case STMF_ICM_REGISTER_LUN:
        case STMF_ICM_DEREGISTER_LUN:
                rc = stmf_ic_reg_dereg_lun_msg_marshal(nvl, msg->icm_msg);
                break;

        case STMF_ICM_SCSI_CMD:
                rc = stmf_ic_scsi_cmd_msg_marshal(nvl, msg->icm_msg);
                break;

        case STMF_ICM_SCSI_DATA:
                rc = stmf_ic_scsi_data_msg_marshal(nvl, msg->icm_msg);
                break;

        case STMF_ICM_SCSI_DATA_XFER_DONE:
                rc = stmf_ic_scsi_data_xfer_done_msg_marshal(nvl, msg->icm_msg);
                break;

        case STMF_ICM_SCSI_STATUS:
                rc = stmf_ic_scsi_status_msg_marshal(nvl, msg->icm_msg);
                break;

        case STMF_ICM_R2T:
                rc = stmf_ic_r2t_msg_marshal(nvl, msg->icm_msg);
                break;

        case STMF_ICM_STATUS:
                rc = stmf_ic_status_msg_marshal(nvl, msg->icm_msg);
                break;

        case STMF_ICM_SESSION_CREATE:
        case STMF_ICM_SESSION_DESTROY:
                rc = stmf_ic_session_create_destroy_msg_marshal(nvl,
                    msg->icm_msg);
                break;

        case STMF_ICM_ECHO_REQUEST:
        case STMF_ICM_ECHO_REPLY:
                rc = stmf_ic_echo_request_reply_msg_marshal(nvl,
                    msg->icm_msg);
```



```
                break;

        case STMF_ICM_MAX_MSG_TYPE:
                ASSERT(0);
                break;

        default:
                ASSERT(0);
        }

done:
        if (!rc)
                return (nvl);

        if (nvl)
        nvlist_free(nvl);

        return (NULL);
}
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  215677 Mon Feb 15 12:56:13 2016
new/usr/src/uts/common/io/comstar/stmf/stmf.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

2431 int
2432 stmf_load_ppd_ioctl(stmf_ppioctl_data_t *ppi, uint64_t *ppi_token,
2433     uint32_t *err_ret)
2434 {
2435         stmf_i_port_provider_t          *ipp;
2436         stmf_i_lu_provider_t            *ilp;
2437         stmf_pp_data_t                  *ppd;
2438         nvlist_t                        *nv;
2439         int                             s;
2440         int                             ret;

2442         *err_ret = 0;

2444         if ((ppi->ppi_lu_provider + ppi->ppi_port_provider) != 1) {
2445                 return (EINVAL);
2446         }

2448         mutex_enter(&stmf_state.stmf_lock);
2449         for (ppd = stmf_state.stmf_ppdlist; ppd != NULL; ppd = ppd->ppd_next) {
2450                 if (ppi->ppi_lu_provider) {
2451                         if (!ppd->ppd_lu_provider)
2452                                 continue;
2453                 } else if (ppi->ppi_port_provider) {
2454                         if (!ppd->ppd_port_provider)
2455                                 continue;
2456                 }
2457                 if (strncmp(ppi->ppi_name, ppd->ppd_name, 254) == 0)
2458                         break;
2459         }

2461         if (ppd == NULL) {
2462                 /* New provider */
2463                 s = strlen(ppi->ppi_name);
2464                 if (s > 254) {
2465                         mutex_exit(&stmf_state.stmf_lock);
2466                         return (EINVAL);
2467                 }
2468                 s += sizeof (stmf_pp_data_t) - 7;

2470                 ppd = kmem_zalloc(s, KM_NOSLEEP);
2471                 if (ppd == NULL) {
2472                         mutex_exit(&stmf_state.stmf_lock);
2473                         return (ENOMEM);
2474                 }
2475                 ppd->ppd_alloc_size = s;
2476                 (void) strcpy(ppd->ppd_name, ppi->ppi_name);

2478                 /* See if this provider already exists */
2479                 if (ppi->ppi_lu_provider) {
2480                         ppd->ppd_lu_provider = 1;
2481                         for (ilp = stmf_state.stmf_ilplist; ilp != NULL;
2482                             ilp = ilp->ilp_next) {
2483                                 if (strcmp(ppi->ppi_name,
2484                                     ilp->ilp_lp->lp_name) == 0) {
2485                                         ppd->ppd_provider = ilp;
2486                                         ilp->ilp_ppd = ppd;
2487                                         break;
2488                                 }
2489                         }
```



```
2490                 } else {
2491                         ppd->ppd_port_provider = 1;
2492                         for (ipp = stmf_state.stmf_ipplist; ipp != NULL;
2493                             ipp = ipp->ipp_next) {
2494                                 if (strcmp(ppi->ppi_name,
2495                                     ipp->ipp_pp->pp_name) == 0) {
2496                                         ppd->ppd_provider = ipp;
2497                                         ipp->ipp_ppd = ppd;
2498                                         break;
2499                                 }
2500                         }
2501                 }

2503                 /* Link this ppd in */
2504                 ppd->ppd_next = stmf_state.stmf_ppdlist;
2505                 stmf_state.stmf_ppdlist = ppd;
2506         }

2508         /*
2509          * User is requesting that the token be checked.
2510          * If there was another set after the user's get
2511          * it's an error
2512          */
2513         if (ppi->ppi_token_valid) {
2514                 if (ppi->ppi_token != ppd->ppd_token) {
2515                         *err_ret = STMF_IOCERR_PPD_UPDATED;
2516                         mutex_exit(&stmf_state.stmf_lock);
2517                         return (EINVAL);
2518                 }
2519         }

2521         if ((ret = nvlist_unpack((char *)ppi->ppi_data,
2522             (size_t)ppi->ppi_data_size, &nv, KM_NOSLEEP)) != 0) {
2523                 mutex_exit(&stmf_state.stmf_lock);
2524                 return (ret);
2525         }

2527         /* Free any existing lists and add this one to the ppd */
2528         if (ppd->ppd_nv)
2528         nvlist_free(ppd->ppd_nv);
2529         ppd->ppd_nv = nv;

2531         /* set the token for writes */
2532         ppd->ppd_token++;
2533         /* return token to caller */
2534         if (ppi_token) {
2535                 *ppi_token = ppd->ppd_token;
2536         }

2538         /* If there is a provider registered, do the notifications */
2539         if (ppd->ppd_provider) {
2540                 uint32_t cb_flags = 0;

2542                 if (stmf_state.stmf_config_state == STMF_CONFIG_INIT)
2543                         cb_flags |= STMF_PCB_STMF_ONLINING;
2544                 if (ppi->ppi_lu_provider) {
2545                         ilp = (stmf_i_lu_provider_t *)ppd->ppd_provider;
2546                         if (ilp->ilp_lp->lp_cb == NULL)
2547                                 goto bail_out;
2548                         ilp->ilp_cb_in_progress = 1;
2549                         mutex_exit(&stmf_state.stmf_lock);
2550                         ilp->ilp_lp->lp_cb(ilp->ilp_lp,
2551                             STMF_PROVIDER_DATA_UPDATED, ppd->ppd_nv, cb_flags);
2552                         mutex_enter(&stmf_state.stmf_lock);
2553                         ilp->ilp_cb_in_progress = 0;
2554                 } else {
```

```
2555                        ipp = (stmf_i_port_provider_t *)ppd->ppd_provider;
2556                        if (ipp->ipp_pp->pp_cb == NULL)
2557                                goto bail_out;
2558                        ipp->ipp_cb_in_progress = 1;
2559                        mutex_exit(&stmf_state.stmf_lock);
2560                        ipp->ipp_pp->pp_cb(ipp->ipp_pp,
2561                            STMF_PROVIDER_DATA_UPDATED, ppd->ppd_nv, cb_flags);
2562                        mutex_enter(&stmf_state.stmf_lock);
2563                        ipp->ipp_cb_in_progress = 0;
2564                }
2565        }

2567 bail_out:
2568        mutex_exit(&stmf_state.stmf_lock);

2570        return (0);
2571 }

2573 void
2574 stmf_delete_ppd(stmf_pp_data_t *ppd)
2575 {
2576        stmf_pp_data_t **pppd;

2578        ASSERT(mutex_owned(&stmf_state.stmf_lock));
2579        if (ppd->ppd_provider) {
2580                if (ppd->ppd_lu_provider) {
2581                        ((stmf_i_lu_provider_t *)
2582                            ppd->ppd_provider)->ilp_ppd = NULL;
2583                } else {
2584                        ((stmf_i_port_provider_t *)
2585                            ppd->ppd_provider)->ipp_ppd = NULL;
2586                }
2587                ppd->ppd_provider = NULL;
2588        }

2590        for (pppd = &stmf_state.stmf_ppdlist; *pppd != NULL;
2591            pppd = &((*pppd)->ppd_next)) {
2592                if (*pppd == ppd)
2593                        break;
2594        }

2596        if (*pppd == NULL)
2597                return;

2599        *pppd = ppd->ppd_next;
2601        if (ppd->ppd_nv)
2600        nvlist_free(ppd->ppd_nv);

2602        kmem_free(ppd, ppd->ppd_alloc_size);
2603 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    9979 Mon Feb 15 12:56:13 2016
new/usr/src/uts/common/io/devfm.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 227 /*ARGSUSED*/
 228 static int
 229 fm_ioctl(dev_t dev, int cmd, intptr_t data, int flag, cred_t *cred, int *rvalp)
 230 {
 231         char *buf;
 232         int err;
 233         uint_t model;
 234         const fm_subr_t *subr;
 235         uint32_t vers;
 236         fm_ioc_data_t fid;
 237         nvlist_t *invl = NULL, *onvl = NULL;
 238 #ifdef _MULTI_DATAMODEL
 239         fm_ioc_data32_t fid32;
 240 #endif

 242         if (getminor(dev) != 0)
 243                 return (ENXIO);

 245         for (subr = fm_subrs; subr->cmd != cmd; subr++)
 246                 if (subr->cmd == -1)
 247                         return (ENOTTY);

 249         if (subr->priv && (flag & FWRITE) == 0 &&
 250             secpolicy_sys_config(CRED(), 0) != 0)
 251                 return (EPERM);

 253         model = ddi_model_convert_from(flag & FMODELS);

 255         switch (model) {
 256 #ifdef _MULTI_DATAMODEL
 257         case DDI_MODEL_ILP32:
 258                 if (ddi_copyin((void *)data, &fid32,
 259                     sizeof (fm_ioc_data32_t), flag) != 0)
 260                         return (EFAULT);
 261                 fid.fid_version = fid32.fid_version;
 262                 fid.fid_insz = fid32.fid_insz;
 263                 fid.fid_inbuf = (caddr_t)(uintptr_t)fid32.fid_inbuf;
 264                 fid.fid_outsz = fid32.fid_outsz;
 265                 fid.fid_outbuf = (caddr_t)(uintptr_t)fid32.fid_outbuf;
 266                 break;
 267 #endif /* _MULTI_DATAMODEL */
 268         case DDI_MODEL_NONE:
 269         default:
 270                 if (ddi_copyin((void *)data, &fid, sizeof (fm_ioc_data_t),
 271                     flag) != 0)
 272                         return (EFAULT);
 273         }

 275         if (nvlist_lookup_uint32(fm_vers_nvl, subr->version, &vers) != 0 ||
 276             fid.fid_version != vers)
 277                 return (ENOTSUP);

 279         if (fid.fid_insz > FM_IOC_MAXBUFSZ)
 280                 return (ENAMETOOLONG);
 281         if (fid.fid_outsz > FM_IOC_OUT_MAXBUFSZ)
 282                 return (EINVAL);

 284         /*
 285          * Copy in and unpack the input nvlist.
```



```
 286          */
 287         if (fid.fid_insz != 0 && fid.fid_inbuf != (caddr_t)0) {
 288                 buf = kmem_alloc(fid.fid_insz, KM_SLEEP);
 289                 if (ddi_copyin(fid.fid_inbuf, buf, fid.fid_insz, flag) != 0) {
 290                         kmem_free(buf, fid.fid_insz);
 291                         return (EFAULT);
 292                 }
 293                 err = nvlist_unpack(buf, fid.fid_insz, &invl, KM_SLEEP);
 294                 kmem_free(buf, fid.fid_insz);
 295                 if (err != 0)
 296                         return (err);
 297         }

 299         err = subr->func(cmd, invl, &onvl);

 301         if (invl != NULL)
 301         nvlist_free(invl);

 303         if (err != 0) {
 305                 if (onvl != NULL)
 304                 nvlist_free(onvl);
 305                 return (err);
 306         }

 308         /*
 309          * If the output nvlist contains any data, pack it and copyout.
 310          */
 311         if (onvl != NULL) {
 312                 size_t sz;

 314                 if ((err = nvlist_size(onvl, &sz, NV_ENCODE_NATIVE)) != 0) {
 315                         nvlist_free(onvl);
 316                         return (err);
 317                 }
 318                 if (sz > fid.fid_outsz) {
 319                         nvlist_free(onvl);
 320                         return (ENAMETOOLONG);
 321                 }

 323                 buf = kmem_alloc(sz, KM_SLEEP);
 324                 if ((err = nvlist_pack(onvl, &buf, &sz, NV_ENCODE_NATIVE,
 325                     KM_SLEEP)) != 0) {
 326                         kmem_free(buf, sz);
 327                         nvlist_free(onvl);
 328                         return (err);
 329                 }
 330                 nvlist_free(onvl);
 331                 if (ddi_copyout(buf, fid.fid_outbuf, sz, flag) != 0) {
 332                         kmem_free(buf, sz);
 333                         return (EFAULT);
 334                 }
 335                 kmem_free(buf, sz);
 336                 fid.fid_outsz = sz;

 338                 switch (model) {
 339 #ifdef _MULTI_DATAMODEL
 340                 case DDI_MODEL_ILP32:
 341                         fid32.fid_outsz = (size32_t)fid.fid_outsz;
 342                         if (ddi_copyout(&fid32, (void *)data,
 343                             sizeof (fm_ioc_data32_t), flag) != 0)
 344                                 return (EFAULT);
 345                         break;
 346 #endif /* _MULTI_DATAMODEL */
 347                 case DDI_MODEL_NONE:
 348                 default:
 349                         if (ddi_copyout(&fid, (void *)data,
```

```
350                             sizeof (fm_ioc_data_t), flag) != 0)
351                                 return (EFAULT);
352                 }
353         }

355         return (err);
356 }
______unchanged_portion_omitted_

420 int
421 _fini(void)
422 {
423         int ret;

425         if ((ret = mod_remove(&modlinkage)) == 0) {
428                 if (fm_vers_nvl != NULL)
426                 nvlist_free(fm_vers_nvl);
427         }

429         return (ret);
430 }
______unchanged_portion_omitted_
```

```
**********************************************************
   34359 Mon Feb 15 12:56:13 2016
new/usr/src/uts/common/ipp/ippctl.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_
1218 #undef  __FN__

1220 #define __FN__  "ippctl_action_modify"
1221 static int
1222 ippctl_action_modify(
1223         char            *aname,
1224         nvlist_t        *nvlp,
1225         ipp_flags_t     flags)
1226 {
1227         ipp_action_id_t aid;
1228         int             ipp_rc;
1229         int             rc;

1231         /*
1232          * Look up the action id and modify the action.
1233          */

1235         aid = ipp_action_lookup(aname);
1236         FREE_TEXT(aname);

1238         ipp_rc = ipp_action_modify(aid, &nvlp, flags);

1240         /*
1241          * Add an nvlist containing the kernel return code to the
1242          * set of nvlists to pass back to libipp.
1243          */

1245         if ((rc = ippctl_set_rc(ipp_rc)) != 0) {
1246                 if (nvlp != NULL)
1246                 nvlist_free(nvlp);
1247                 return (rc);
1248         }

1250         /*
1251          * If the module passed back an nvlist, add this as
1252          * well.
1253          */

1255         if (nvlp != NULL) {
1256                 rc = ippctl_callback(nvlp, NULL);
1257                 nvlist_free(nvlp);
1258         } else
1259                 rc = 0;

1261         return (rc);
1262 }
______unchanged_portion_omitted_
```

```
**********************************************************
   69448 Mon Feb 15 12:56:14 2016
new/usr/src/uts/common/os/contract.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

1820 /*
1821  * cte_rele
1822  *
1823  * Releases a hold on the specified event.  If the caller had the last
1824  * reference, frees the event and releases its hold on the contract
1825  * that generated it.
1826  */
1827 static void
1828 cte_rele(ct_kevent_t *e)
1829 {
1830         mutex_enter(&e->cte_lock);
1831         ASSERT(e->cte_refs > 0);
1832         if (--e->cte_refs) {
1833                 mutex_exit(&e->cte_lock);
1834                 return;
1835         }

1837         contract_rele(e->cte_contract);

1839         mutex_destroy(&e->cte_lock);
1840         if (e->cte_data)
1840         nvlist_free(e->cte_data);
1842         if (e->cte_gdata)
1841         nvlist_free(e->cte_gdata);
1842         kmem_free(e, sizeof (ct_kevent_t));
1843 }
______unchanged_portion_omitted_
```

```
**********************************************************
   42349 Mon Feb 15 12:56:14 2016
new/usr/src/uts/common/os/damap.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

1093 static void
1094 dam_deact_cleanup(dam_t *mapp, id_t addrid, char *addrstr,
1095     damap_deact_rsn_t deact_rsn)
1096 {
1097         dam_da_t *passp;

1099         passp = ddi_get_soft_state(mapp->dam_da, addrid);
1100         ASSERT(passp);
1101         if (mapp->dam_deactivate_cb)
1102                 (*mapp->dam_deactivate_cb)(mapp->dam_activate_arg,
1103                     ddi_strid_id2str(mapp->dam_addr_hash, addrid),
1104                     addrid, passp->da_ppriv, deact_rsn);

1106         /*
1107          * clear the active bit and free the backing info for
1108          * this address
1109          */
1110         mutex_enter(&mapp->dam_lock);
1111         bitset_del(&mapp->dam_active_set, addrid);
1112         passp->da_ppriv = NULL;
1113         if (passp->da_nvl)
1113         nvlist_free(passp->da_nvl);
1114         passp->da_nvl = NULL;
1115         passp->da_ppriv_rpt = NULL;
1117         if (passp->da_nvl_rpt)
1116         nvlist_free(passp->da_nvl_rpt);
1117         passp->da_nvl_rpt = NULL;

1119         DTRACE_PROBE3(damap__addr__deactivate__end,
1120             char *, mapp->dam_name, dam_t *, mapp,
1121             char *, addrstr);

1123         (void) dam_addr_release(mapp, addrid);
1124         mutex_exit(&mapp->dam_lock);
1125 }
______unchanged_portion_omitted_

1627 /*
1628  * release an address report
1629  */
1630 static void
1631 dam_addr_report_release(dam_t *mapp, id_t addrid)
1632 {
1633         dam_da_t *passp;
1634         char *addrstr = damap_id2addr((damap_t *)mapp, addrid);

1636         DTRACE_PROBE3(damap__addr__report__release,
1637             char *, mapp->dam_name, dam_t *, mapp,
1638             char *, addrstr);

1640         ASSERT(mutex_owned(&mapp->dam_lock));
1641         passp = ddi_get_soft_state(mapp->dam_da, addrid);
1642         ASSERT(passp);
1643         /*
1644          * clear the report bit
1645          * if the address has a registered deactivation handler and
1646          * we are holding a private data pointer and the address has not
1647          * stabilized, deactivate the address (private data).
1648          */
```



```
1649         bitset_del(&mapp->dam_report_set, addrid);
1650         if (!DAM_IS_STABLE(mapp, addrid) && mapp->dam_deactivate_cb &&
1651             passp->da_ppriv_rpt) {
1652                 mutex_exit(&mapp->dam_lock);
1653                 (*mapp->dam_deactivate_cb)(mapp->dam_activate_arg,
1654                     ddi_strid_id2str(mapp->dam_addr_hash, addrid),
1655                     addrid, passp->da_ppriv_rpt, DAMAP_DEACT_RSN_UNSTBL);
1656                 mutex_enter(&mapp->dam_lock);
1657         }
1658         passp->da_ppriv_rpt = NULL;
1661         if (passp->da_nvl_rpt)
1659         nvlist_free(passp->da_nvl_rpt);
1660 }
______unchanged_portion_omitted_
```

****
30818 Mon Feb 15 12:56:14 2016
new/usr/src/uts/common/os/devid_cache.c
6659 nvlist_free(NULL) is a no-op
****
_____**unchanged_portion_omitted_**

```
 287 /*
 288  * Pack the list of devid cache elements into a single nvlist
 289  * Used when writing the nvlist file.
 290  */
 291 static int
 292 devid_cache_pack_list(nvf_handle_t fd, nvlist_t **ret_nvl)
 293 {
 294         nvlist_t        *nvl, *sub_nvl;
 295         nvp_devid_t     *np;
 296         int             rval;
 297         list_t          *listp;

 299         ASSERT(RW_WRITE_HELD(nvf_lock(dcfd_handle)));

 301         rval = nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP);
 302         if (rval != 0) {
 303                 nvf_error("%s: nvlist alloc error %d\n",
 304                     nvf_cache_name(fd), rval);
 305                 return (DDI_FAILURE);
 306         }

 308         listp = nvf_list(fd);
 309         for (np = list_head(listp); np; np = list_next(listp, np)) {
 310                 if (np->nvp_devid == NULL)
 311                         continue;
 312                 NVP_DEVID_DEBUG_PATH(np->nvp_devpath);
 313                 rval = nvlist_alloc(&sub_nvl, NV_UNIQUE_NAME, KM_SLEEP);
 314                 if (rval != 0) {
 315                         nvf_error("%s: nvlist alloc error %d\n",
 316                             nvf_cache_name(fd), rval);
 317                         sub_nvl = NULL;
 318                         goto err;
 319                 }

 321                 rval = nvlist_add_byte_array(sub_nvl, DP_DEVID_ID,
 322                     (uchar_t *)np->nvp_devid,
 323                     ddi_devid_sizeof(np->nvp_devid));
 324                 if (rval == 0) {
 325                         NVP_DEVID_DEBUG_DEVID(np->nvp_devid);
 326                 } else {
 327                         nvf_error(
 328                             "%s: nvlist add error %d (devid)\n",
 329                             nvf_cache_name(fd), rval);
 330                         goto err;
 331                 }

 333                 rval = nvlist_add_nvlist(nvl, np->nvp_devpath, sub_nvl);
 334                 if (rval != 0) {
 335                         nvf_error("%s: nvlist add error %d (sublist)\n",
 336                             nvf_cache_name(fd), rval);
 337                         goto err;
 338                 }
 339                 nvlist_free(sub_nvl);
 340         }

 342         *ret_nvl = nvl;
 343         return (DDI_SUCCESS);

 345 err:
```



```
 346         if (sub_nvl)
 346         nvlist_free(sub_nvl);
 347         nvlist_free(nvl);
 348         *ret_nvl = NULL;
 349         return (DDI_FAILURE);
 350 }
```

_____**unchanged_portion_omitted_**

****
59695 Mon Feb 15 12:56:14 2016
new/usr/src/uts/common/os/evchannels.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
1139 /*
1140  * Unbind: Free bind structure. Remove channel if last binding was freed.
1141  */
1142 static void
1143 evch_chunbind(evch_bind_t *bp)
1144 {
1145         struct evch_globals *eg;
1146         evch_chan_t *chp = bp->bd_channel;

1148         eg = zone_getspecific(evch_zone_key, curproc->p_zone);
1149         ASSERT(eg != NULL);

1151         mutex_enter(&eg->evch_list_lock);
1152         mutex_enter(&chp->ch_mutex);
1153         ASSERT(chp->ch_bindings > 0);
1154         chp->ch_bindings--;
1155         kmem_free(bp, sizeof (evch_bind_t));
1156         if (chp->ch_bindings == 0 && evch_dl_getnum(&chp->ch_subscr) == 0 &&
1157             (chp->ch_nevents == 0 || chp->ch_holdpend != CH_HOLD_PEND_INDEF)) {
1158                 /*
1159                  * No more bindings and no persistent subscriber(s).  If there
1160                  * are no events in the channel then destroy the channel;
1161                  * otherwise destroy the channel only if we're not holding
1162                  * pending events indefinitely.
1163                  */
1164                 mutex_exit(&chp->ch_mutex);
1165                 evch_dl_del(&eg->evch_list, &chp->ch_link);
1166                 evch_evq_destroy(chp->ch_queue);
1167                 if (chp->ch_propnvl)
1167                 nvlist_free(chp->ch_propnvl);
1168                 mutex_destroy(&chp->ch_mutex);
1169                 mutex_destroy(&chp->ch_pubmx);
1170                 cv_destroy(&chp->ch_pubcv);
1171                 kmem_free(chp->ch_name, chp->ch_namelen);
1172                 kmem_free(chp, sizeof (evch_chan_t));
1173         } else
1174                 mutex_exit(&chp->ch_mutex);
1175         mutex_exit(&eg->evch_list_lock);
1176 }
```

_____***unchanged_portion_omitted_***

```
1567 static void
1568 evch_chsetpropnvl(evch_bind_t *bp, nvlist_t *nvl)
1569 {
1570         evch_chan_t *chp = bp->bd_channel;

1572         mutex_enter(&chp->ch_mutex);

1575         if (chp->ch_propnvl)
1574         nvlist_free(chp->ch_propnvl);

1576         chp->ch_propnvl = nvl;
1577         chp->ch_propnvlgen++;

1579         mutex_exit(&chp->ch_mutex);
1580 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   11124 Mon Feb 15 12:56:14 2016
new/usr/src/uts/common/os/retire_store.c
patch cleanup
6659 nvlist_free(NULL) is a no-op
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 #pragma ident   "%Z%%M% %I%     %E% SMI"

  26 #include <sys/ddi.h>
  27 #include <sys/sunddi.h>
  28 #include <sys/sunndi.h>
  29 #include <sys/ddi_impldefs.h>
  30 #include <sys/ddi_implfuncs.h>
  31 #include <sys/list.h>
  32 #include <sys/reboot.h>
  33 #include <sys/sysmacros.h>
  34 #include <sys/console.h>
  35 #include <sys/devcache.h>

  37 /*
  38  * The nvpair name in the I/O retire specific sub-nvlist
  39  */
  40 #define RIO_STORE_VERSION_STR   "rio-store-version"
  41 #define RIO_STORE_MAGIC_STR     "rio-store-magic"
  42 #define RIO_STORE_FLAGS_STR     "rio-store-flags"

  44 #define RIO_STORE_VERSION_1     1
  45 #define RIO_STORE_VERSION       RIO_STORE_VERSION_1

  47 /*
  48  * decoded retire list element
  49  */

  51 typedef enum rio_store_flags {
  52         RIO_STORE_F_INVAL = 0,
  53         RIO_STORE_F_RETIRED = 1,
  54         RIO_STORE_F_BYPASS = 2
  55 } rio_store_flags_t;
_____unchanged_portion_omitted_

 245 static int
```



```
 246 rio_store_encode(nvf_handle_t nvfh, nvlist_t **ret_nvl)
 247 {
 248         nvlist_t        *nvl;
 249         nvlist_t        *line_nvl;
 250         list_t          *listp;
 251         rio_store_t     *rsp;
 252         int             rval;

 254         ASSERT(nvfh == rio_store_handle);
 255         ASSERT(RW_WRITE_HELD(nvf_lock(nvfh)));

 257         *ret_nvl = NULL;

 259         nvl = NULL;
 260         rval = nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP);
 261         if (rval != 0) {
 262                 return (DDI_FAILURE);
 263         }

 265         listp = nvf_list(nvfh);
 266         for (rsp = list_head(listp); rsp; rsp = list_next(listp, rsp)) {
 267                 int flag_mask = RIO_STORE_F_RETIRED|RIO_STORE_F_BYPASS;
 268                 int flags;
 269                 ASSERT(rsp->rst_devpath);
 270                 ASSERT(!(rsp->rst_flags & ~flag_mask));

 272                 line_nvl = NULL;
 273                 rval = nvlist_alloc(&line_nvl, NV_UNIQUE_NAME, KM_SLEEP);
 274                 if (rval != 0) {
 275                         line_nvl = NULL;
 276                         goto error;
 277                 }

 279                 rval = nvlist_add_int32(line_nvl, RIO_STORE_VERSION_STR,
 280                         RIO_STORE_VERSION);
 281                 if (rval != 0) {
 282                         goto error;
 283                 }
 284                 rval = nvlist_add_int32(line_nvl, RIO_STORE_MAGIC_STR,
 285                         RIO_STORE_MAGIC);
 286                 if (rval != 0) {
 287                         goto error;
 288                 }

 290                 /* don't save the bypass flag */
 291                 flags = RIO_STORE_F_RETIRED;
 292                 rval = nvlist_add_int32(line_nvl, RIO_STORE_FLAGS_STR,
 293                         flags);
 294                 if (rval != 0) {
 295                         goto error;
 296                 }

 298                 rval = nvlist_add_nvlist(nvl, rsp->rst_devpath, line_nvl);
 299                 if (rval != 0) {
 300                         goto error;
 301                 }
 302                 nvlist_free(line_nvl);
 303                 line_nvl = NULL;
 304         }

 306         *ret_nvl = nvl;
 307         STORE_DBG((CE_NOTE, "packed retire list into nvlist"));
 308         return (DDI_SUCCESS);

 310 error:
 313         if (line_nvl)
```

```
 311         nvlist_free(line_nvl);
 312         ASSERT(nvl);
 313         nvlist_free(nvl);
 314         return (DDI_FAILURE);
 315 }
______unchanged_portion_omitted_
```

****
238255 Mon Feb 15 12:56:14 2016
new/usr/src/uts/common/os/sunmdi.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
7980 /*
7981  * Build nvlist using the information in the vhci cache.
7982  * See the comment in mainnvl_to_vhcache() for the format of the nvlist.
7983  * Returns nvl on success, NULL on failure.
7984  */
7985 static nvlist_t *
7986 vhcache_to_mainnvl(mdi_vhci_cache_t *vhcache)
7987 {
7988         mdi_vhcache_phci_t *cphci;
7989         uint_t phci_count;
7990         char **phcis;
7991         nvlist_t *nvl;
7992         int err, i;

7994         if ((err = nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP)) != 0) {
7995                 nvl = NULL;
7996                 goto out;
7997         }

7999         if ((err = nvlist_add_int32(nvl, MDI_NVPNAME_VERSION,
8000             MDI_VHCI_CACHE_VERSION)) != 0)
8001                 goto out;

8003         rw_enter(&vhcache->vhcache_lock, RW_READER);
8004         if (vhcache->vhcache_phci_head == NULL) {
8005                 rw_exit(&vhcache->vhcache_lock);
8006                 return (nvl);
8007         }

8009         phci_count = 0;
8010         for (cphci = vhcache->vhcache_phci_head; cphci != NULL;
8011             cphci = cphci->cphci_next)
8012                 cphci->cphci_id = phci_count++;

8014         /* build phci pathname list */
8015         phcis = kmem_alloc(sizeof (char *) * phci_count, KM_SLEEP);
8016         for (cphci = vhcache->vhcache_phci_head, i = 0; cphci != NULL;
8017             cphci = cphci->cphci_next, i++)
8018                 phcis[i] = i_ddi_strdup(cphci->cphci_path, KM_SLEEP);

8020         err = nvlist_add_string_array(nvl, MDI_NVPNAME_PHCIS, phcis,
8021             phci_count);
8022         free_string_array(phcis, phci_count);

8024         if (err == 0 &&
8025             (err = vhcache_to_caddrmapnvl(vhcache, nvl)) == 0) {
8026                 rw_exit(&vhcache->vhcache_lock);
8027                 return (nvl);
8028         }

8030         rw_exit(&vhcache->vhcache_lock);
8031 out:
8032         if (nvl)
8032         nvlist_free(nvl);
8033         return (NULL);
8034 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   64137 Mon Feb 15 12:56:15 2016
new/usr/src/uts/common/os/sunndi.c
patch tsoome-feedback
**********************************************************
______unchanged_portion_omitted_

 409 /*
 410  * free all space allocated to a handle.
 411  */
 412 void
 413 ndi_dc_freehdl(struct devctl_iocdata *dcp)
 414 {
 415         ASSERT(dcp != NULL);

 417         if (dcp->c_nodename != NULL)
 418                 kmem_free(dcp->c_nodename, MAXNAMELEN);

 420         if (dcp->c_unitaddr != NULL)
 421                 kmem_free(dcp->c_unitaddr, MAXNAMELEN);

 423         if (dcp->nvl_user != NULL)
 423         nvlist_free(dcp->nvl_user);

 425         kmem_free(dcp, sizeof (*dcp));
 426 }
______unchanged_portion_omitted_
```

```
**********************************************************
  193067 Mon Feb 15 12:56:15 2016
new/usr/src/uts/common/os/zone.c
patch tsoome-feedback
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

4246 /*
4247  * System call to create/initialize a new zone named 'zone_name', rooted
4248  * at 'zone_root', with a zone-wide privilege limit set of 'zone_privs',
4249  * and initialized with the zone-wide rctls described in 'rctlbuf', and
4250  * with labeling set by 'match', 'doi', and 'label'.
4251  *
4252  * If extended error is non-null, we may use it to return more detailed
4253  * error information.
4254  */
4255 static zoneid_t
4256 zone_create(const char *zone_name, const char *zone_root,
4257     const priv_set_t *zone_privs, size_t zone_privssz,
4258     caddr_t rctlbuf, size_t rctlbufsz,
4259     caddr_t zfsbuf, size_t zfsbufsz, int *extended_error,
4260     int match, uint32_t doi, const bslabel_t *label,
4261     int flags)
4262 {
4263         struct zsched_arg zarg;
4264         nvlist_t *rctls = NULL;
4265         proc_t *pp = curproc;
4266         zone_t *zone, *ztmp;
4267         zoneid_t zoneid;
4268         int error;
4269         int error2 = 0;
4270         char *str;
4271         cred_t *zkcr;
4272         boolean_t insert_label_hash;

4274         if (secpolicy_zone_config(CRED()) != 0)
4275                 return (set_errno(EPERM));

4277         /* can't boot zone from within chroot environment */
4278         if (PTOU(pp)->u_rdir != NULL && PTOU(pp)->u_rdir != rootdir)
4279                 return (zone_create_error(ENOTSUP, ZE_CHROOTED,
4280                     extended_error));

4282         zone = kmem_zalloc(sizeof (zone_t), KM_SLEEP);
4283         zoneid = zone->zone_id = id_alloc(zoneid_space);
4284         zone->zone_status = ZONE_IS_UNINITIALIZED;
4285         zone->zone_pool = pool_default;
4286         zone->zone_pool_mod = gethrtime();
4287         zone->zone_psetid = ZONE_PS_INVAL;
4288         zone->zone_ncpus = 0;
4289         zone->zone_ncpus_online = 0;
4290         zone->zone_restart_init = B_TRUE;
4291         zone->zone_brand = &native_brand;
4292         zone->zone_initname = NULL;
4293         mutex_init(&zone->zone_lock, NULL, MUTEX_DEFAULT, NULL);
4294         mutex_init(&zone->zone_nlwps_lock, NULL, MUTEX_DEFAULT, NULL);
4295         mutex_init(&zone->zone_mem_lock, NULL, MUTEX_DEFAULT, NULL);
4296         cv_init(&zone->zone_cv, NULL, CV_DEFAULT, NULL);
4297         list_create(&zone->zone_ref_list, sizeof (zone_ref_t),
4298             offsetof(zone_ref_t, zref_linkage));
4299         list_create(&zone->zone_zsd, sizeof (struct zsd_entry),
4300             offsetof(struct zsd_entry, zsd_linkage));
4301         list_create(&zone->zone_datasets, sizeof (zone_dataset_t),
4302             offsetof(zone_dataset_t, zd_linkage));
4303         list_create(&zone->zone_dl_list, sizeof (zone_dl_t),
```



```
4304             offsetof(zone_dl_t, zdl_linkage));
4305         rw_init(&zone->zone_mlps.mlpl_rwlock, NULL, RW_DEFAULT, NULL);
4306         rw_init(&zone->zone_mntfs_db_lock, NULL, RW_DEFAULT, NULL);

4308         if (flags & ZCF_NET_EXCL) {
4309                 zone->zone_flags |= ZF_NET_EXCL;
4310         }

4312         if ((error = zone_set_name(zone, zone_name)) != 0) {
4313                 zone_free(zone);
4314                 return (zone_create_error(error, 0, extended_error));
4315         }

4317         if ((error = zone_set_root(zone, zone_root)) != 0) {
4318                 zone_free(zone);
4319                 return (zone_create_error(error, 0, extended_error));
4320         }
4321         if ((error = zone_set_privset(zone, zone_privs, zone_privssz)) != 0) {
4322                 zone_free(zone);
4323                 return (zone_create_error(error, 0, extended_error));
4324         }

4326         /* initialize node name to be the same as zone name */
4327         zone->zone_nodename = kmem_alloc(_SYS_NMLN, KM_SLEEP);
4328         (void) strncpy(zone->zone_nodename, zone->zone_name, _SYS_NMLN);
4329         zone->zone_nodename[_SYS_NMLN - 1] = '\0';

4331         zone->zone_domain = kmem_alloc(_SYS_NMLN, KM_SLEEP);
4332         zone->zone_domain[0] = '\0';
4333         zone->zone_hostid = HW_INVALID_HOSTID;
4334         zone->zone_shares = 1;
4335         zone->zone_shmmax = 0;
4336         zone->zone_ipc.ipcq_shmmni = 0;
4337         zone->zone_ipc.ipcq_semmni = 0;
4338         zone->zone_ipc.ipcq_msgmni = 0;
4339         zone->zone_bootargs = NULL;
4340         zone->zone_fs_allowed = NULL;
4341         zone->zone_initname =
4342             kmem_alloc(strlen(zone_default_initname) + 1, KM_SLEEP);
4343         (void) strcpy(zone->zone_initname, zone_default_initname);
4344         zone->zone_nlwps = 0;
4345         zone->zone_nlwps_ctl = INT_MAX;
4346         zone->zone_nprocs = 0;
4347         zone->zone_nprocs_ctl = INT_MAX;
4348         zone->zone_locked_mem = 0;
4349         zone->zone_locked_mem_ctl = UINT64_MAX;
4350         zone->zone_max_swap = 0;
4351         zone->zone_max_swap_ctl = UINT64_MAX;
4352         zone->zone_max_lofi = 0;
4353         zone->zone_max_lofi_ctl = UINT64_MAX;
4354         zone0.zone_lockedmem_kstat = NULL;
4355         zone0.zone_swapresv_kstat = NULL;

4357         /*
4358          * Zsched initializes the rctls.
4359          */
4360         zone->zone_rctls = NULL;

4362         if ((error = parse_rctls(rctlbuf, rctlbufsz, &rctls)) != 0) {
4363                 zone_free(zone);
4364                 return (zone_create_error(error, 0, extended_error));
4365         }

4367         if ((error = parse_zfs(zone, zfsbuf, zfsbufsz)) != 0) {
4368                 zone_free(zone);
4369                 return (set_errno(error));
```

```
4370         }
4371
4372         /*
4373          * Read in the trusted system parameters:
4374          * match flag and sensitivity label.
4375          */
4376         zone->zone_match = match;
4377         if (is_system_labeled() && !(zone->zone_flags & ZF_IS_SCRATCH)) {
4378                 /* Fail if requested to set doi to anything but system's doi */
4379                 if (doi != 0 && doi != default_doi) {
4380                         zone_free(zone);
4381                         return (set_errno(EINVAL));
4382                 }
4383                 /* Always apply system's doi to the zone */
4384                 error = zone_set_label(zone, label, default_doi);
4385                 if (error != 0) {
4386                         zone_free(zone);
4387                         return (set_errno(error));
4388                 }
4389                 insert_label_hash = B_TRUE;
4390         } else {
4391                 /* all zones get an admin_low label if system is not labeled */
4392                 zone->zone_slabel = l_admin_low;
4393                 label_hold(l_admin_low);
4394                 insert_label_hash = B_FALSE;
4395         }
4396
4397         /*
4398          * Stop all lwps since that's what normally happens as part of fork().
4399          * This needs to happen before we grab any locks to avoid deadlock
4400          * (another lwp in the process could be waiting for the held lock).
4401          */
4402         if (curthread != pp->p_agenttp && !holdlwps(SHOLDFORK)) {
4403                 zone_free(zone);
4404                 if (rctls)
4404                 nvlist_free(rctls);
4405                 return (zone_create_error(error, 0, extended_error));
4406         }
4407
4408         if (block_mounts(zone) == 0) {
4409                 mutex_enter(&pp->p_lock);
4410                 if (curthread != pp->p_agenttp)
4411                         continuelwps(pp);
4412                 mutex_exit(&pp->p_lock);
4413                 zone_free(zone);
4414                 if (rctls)
4415                 nvlist_free(rctls);
4415                 return (zone_create_error(error, 0, extended_error));
4416         }
4417
4418         /*
4419          * Set up credential for kernel access.  After this, any errors
4420          * should go through the dance in errout rather than calling
4421          * zone_free directly.
4422          */
4423         zone->zone_kcred = crdup(kcred);
4424         crsetzone(zone->zone_kcred, zone);
4425         priv_intersect(zone->zone_privset, &CR_PPRIV(zone->zone_kcred));
4426         priv_intersect(zone->zone_privset, &CR_EPRIV(zone->zone_kcred));
4427         priv_intersect(zone->zone_privset, &CR_IPRIV(zone->zone_kcred));
4428         priv_intersect(zone->zone_privset, &CR_LPRIV(zone->zone_kcred));
4429
4430         mutex_enter(&zonehash_lock);
4431         /*
4432          * Make sure zone doesn't already exist.
4433          *
```



```
4434          * If the system and zone are labeled,
4435          * make sure no other zone exists that has the same label.
4436          */
4437         if ((ztmp = zone_find_all_by_name(zone->zone_name)) != NULL ||
4438             (insert_label_hash &&
4439             (ztmp = zone_find_all_by_label(zone->zone_slabel)) != NULL)) {
4440                 zone_status_t status;
4441
4442                 status = zone_status_get(ztmp);
4443                 if (status == ZONE_IS_READY || status == ZONE_IS_RUNNING)
4444                         error = EEXIST;
4445                 else
4446                         error = EBUSY;
4447
4448                 if (insert_label_hash)
4449                         error2 = ZE_LABELINUSE;
4450
4451                 goto errout;
4452         }
4453
4454         /*
4455          * Don't allow zone creations which would cause one zone's rootpath to
4456          * be accessible from that of another (non-global) zone.
4457          */
4458         if (zone_is_nested(zone->zone_rootpath)) {
4459                 error = EBUSY;
4460                 goto errout;
4461         }
4462
4463         ASSERT(zonecount != 0);         /* check for leaks */
4464         if (zonecount + 1 > maxzones) {
4465                 error = ENOMEM;
4466                 goto errout;
4467         }
4468
4469         if (zone_mount_count(zone->zone_rootpath) != 0) {
4470                 error = EBUSY;
4471                 error2 = ZE_AREMOUNTS;
4472                 goto errout;
4473         }
4474
4475         /*
4476          * Zone is still incomplete, but we need to drop all locks while
4477          * zsched() initializes this zone's kernel process.  We
4478          * optimistically add the zone to the hashtable and associated
4479          * lists so a parallel zone_create() doesn't try to create the
4480          * same zone.
4481          */
4482         zonecount++;
4483         (void) mod_hash_insert(zonehashbyid,
4484             (mod_hash_key_t)(uintptr_t)zone->zone_id,
4485             (mod_hash_val_t)(uintptr_t)zone);
4486         str = kmem_alloc(strlen(zone->zone_name) + 1, KM_SLEEP);
4487         (void) strcpy(str, zone->zone_name);
4488         (void) mod_hash_insert(zonehashbyname, (mod_hash_key_t)str,
4489             (mod_hash_val_t)(uintptr_t)zone);
4490         if (insert_label_hash) {
4491                 (void) mod_hash_insert(zonehashbylabel,
4492                     (mod_hash_key_t)zone->zone_slabel, (mod_hash_val_t)zone);
4493                 zone->zone_flags |= ZF_HASHED_LABEL;
4494         }
4495
4496         /*
4497          * Insert into active list.  At this point there are no 'hold's
4498          * on the zone, but everyone else knows not to use it, so we can
4499          * continue to use it.  zsched() will do a zone_hold() if the
```

```
4500          * newproc() is successful.
4501          */
4502         list_insert_tail(&zone_active, zone);
4503         mutex_exit(&zonehash_lock);

4505         zarg.zone = zone;
4506         zarg.nvlist = rctls;
4507         /*
4508          * The process, task, and project rctls are probably wrong;
4509          * we need an interface to get the default values of all rctls,
4510          * and initialize zsched appropriately.  I'm not sure that that
4511          * makes much of a difference, though.
4512          */
4513         error = newproc(zsched, (void *)&zarg, syscid, minclsyspri, NULL, 0);
4514         if (error != 0) {
4515                 /*
4516                  * We need to undo all globally visible state.
4517                  */
4518                 mutex_enter(&zonehash_lock);
4519                 list_remove(&zone_active, zone);
4520                 if (zone->zone_flags & ZF_HASHED_LABEL) {
4521                         ASSERT(zone->zone_slabel != NULL);
4522                         (void) mod_hash_destroy(zonehashbylabel,
4523                             (mod_hash_key_t)zone->zone_slabel);
4524                 }
4525                 (void) mod_hash_destroy(zonehashbyname,
4526                     (mod_hash_key_t)(uintptr_t)zone->zone_name);
4527                 (void) mod_hash_destroy(zonehashbyid,
4528                     (mod_hash_key_t)(uintptr_t)zone->zone_id);
4529                 ASSERT(zonecount > 1);
4530                 zonecount--;
4531                 goto errout;
4532         }

4534         /*
4535          * Zone creation can't fail from now on.
4536          */

4538         /*
4539          * Create zone kstats
4540          */
4541         zone_kstat_create(zone);

4543         /*
4544          * Let the other lwps continue.
4545          */
4546         mutex_enter(&pp->p_lock);
4547         if (curthread != pp->p_agenttp)
4548                 continuelwps(pp);
4549         mutex_exit(&pp->p_lock);

4551         /*
4552          * Wait for zsched to finish initializing the zone.
4553          */
4554         zone_status_wait(zone, ZONE_IS_READY);
4555         /*
4556          * The zone is fully visible, so we can let mounts progress.
4557          */
4558         resume_mounts(zone);
4561         if (rctls)
4559                 nvlist_free(rctls);

4561         return (zoneid);

4563 errout:
4564         mutex_exit(&zonehash_lock);
```



```
4565         /*
4566          * Let the other lwps continue.
4567          */
4568         mutex_enter(&pp->p_lock);
4569         if (curthread != pp->p_agenttp)
4570                 continuelwps(pp);
4571         mutex_exit(&pp->p_lock);

4573         resume_mounts(zone);
4577         if (rctls)
4574         nvlist_free(rctls);
4575         /*
4576          * There is currently one reference to the zone, a cred_ref from
4577          * zone_kcred.  To free the zone, we call crfree, which will call
4578          * zone_cred_rele, which will call zone_free.
4579          */
4580         ASSERT(zone->zone_cred_ref == 1);
4581         ASSERT(zone->zone_kcred->cr_ref == 1);
4582         ASSERT(zone->zone_ref == 0);
4583         zkcr = zone->zone_kcred;
4584         zone->zone_kcred = NULL;
4585         crfree(zkcr);                           /* triggers call to zone_free */
4586         return (zone_create_error(error, error2, extended_error));
4587 }
______unchanged_portion_omitted_

6859 static int
6860 zone_remove_datalink(zoneid_t zoneid, datalink_id_t linkid)
6861 {
6862         zone_dl_t *zdl;
6863         zone_t *zone;
6864         int err = 0;

6866         if ((zone = zone_find_by_id(zoneid)) == NULL)
6867                 return (set_errno(EINVAL));

6869         mutex_enter(&zone->zone_lock);
6870         if ((zdl = zone_find_dl(zone, linkid)) == NULL) {
6871                 err = ENXIO;
6872         } else {
6873                 list_remove(&zone->zone_dl_list, zdl);
6878                 if (zdl->zdl_net != NULL)
6874                 nvlist_free(zdl->zdl_net);
6875                 kmem_free(zdl, sizeof (zone_dl_t));
6876         }
6877         mutex_exit(&zone->zone_lock);
6878         zone_rele(zone);
6879         return (err == 0 ? 0 : set_errno(err));
6880 }
______unchanged_portion_omitted_
```

****
61155 Mon Feb 15 12:56:15 2016
new/usr/src/uts/common/xen/os/xvdi.c
patch tsoome-feedback
****
_____***unchanged_portion_omitted_***

```
1689 /*
1690  * Notify hotplug script running in userland
1691  */
1692 int
1693 xvdi_post_event(dev_info_t *dip, xendev_hotplug_cmd_t hpc)
1694 {
1695         struct xendev_ppd *pdp;
1696         nvlist_t *attr_list = NULL;
1697         i_xd_cfg_t *xdcp;
1698         sysevent_id_t eid;
1699         int err;
1700         char devname[256]; /* XXPV dme: ? */

1702         pdp = ddi_get_parent_data(dip);
1703         ASSERT(pdp != NULL);

1705         xdcp = i_xvdi_devclass2cfg(pdp->xd_devclass);
1706         ASSERT(xdcp != NULL);

1708         (void) snprintf(devname, sizeof (devname) - 1, "%s%d",
1709             ddi_driver_name(dip),  ddi_get_instance(dip));

1711         err = nvlist_alloc(&attr_list, NV_UNIQUE_NAME, KM_NOSLEEP);
1712         if (err != DDI_SUCCESS)
1713                 goto failure;

1715         err = nvlist_add_int32(attr_list, "domain", pdp->xd_domain);
1716         if (err != DDI_SUCCESS)
1717                 goto failure;
1718         err = nvlist_add_int32(attr_list, "vdev", pdp->xd_vdevnum);
1719         if (err != DDI_SUCCESS)
1720                 goto failure;
1721         err = nvlist_add_string(attr_list, "devclass", xdcp->xsdev);
1722         if (err != DDI_SUCCESS)
1723                 goto failure;
1724         err = nvlist_add_string(attr_list, "device", devname);
1725         if (err != DDI_SUCCESS)
1726                 goto failure;
1727         err = nvlist_add_string(attr_list, "fob",
1728             ((pdp->xd_xsdev.frontend == 1) ? "frontend" : "backend"));
1729         if (err != DDI_SUCCESS)
1730                 goto failure;

1732         switch (hpc) {
1733         case XEN_HP_ADD:
1734                 err = ddi_log_sysevent(dip, DDI_VENDOR_SUNW, "EC_xendev",
1735                     "add", attr_list, &eid, DDI_NOSLEEP);
1736                 break;
1737         case XEN_HP_REMOVE:
1738                 err = ddi_log_sysevent(dip, DDI_VENDOR_SUNW, "EC_xendev",
1739                     "remove", attr_list, &eid, DDI_NOSLEEP);
1740                 break;
1741         default:
1742                 err = DDI_FAILURE;
1743                 goto failure;
1744         }

1746 failure:
1747         if (attr_list != NULL)
```



```
1747                 nvlist_free(attr_list);

1749         return (err);
1750 }
```
_____***unchanged_portion_omitted_***

****
13969 Mon Feb 15 12:56:15 2016
new/usr/src/uts/intel/io/intel_nb5000/intel_nbdrv.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
 302 static void
 303 inb_create_nvl()
 304 {
 305         nvlist_t *nvl;

 307         (void) nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP);
 308         (void) nvlist_add_uint8(nvl, MCINTEL_NVLIST_VERSTR,
 309             MCINTEL_NVLIST_VERS);
 310         (void) nvlist_add_string(nvl, "memory-controller", inb_mc_name());
 311         if (nb_chipset == INTEL_NB_5100)
 312                 (void) nvlist_add_uint8(nvl, MCINTEL_NVLIST_NMEM,
 313                     (uint8_t)nb_number_memory_controllers);
 314         inb_dimmlist(nvl);

 316         if (inb_mc_nvl)
 316         nvlist_free(inb_mc_nvl);
 317         inb_mc_nvl = nvl;
 318 }
```

_____***unchanged_portion_omitted_***

****
8623 Mon Feb 15 12:56:15 2016
new/usr/src/uts/intel/io/intel_nhm/dimm_topo.c
6659 nvlist_free(NULL) is a no-op
****
_____***unchanged_portion_omitted_***

```
 252 void
 253 inhm_create_nvl(int chip)
 254 {
 255         nvlist_t *nvl;

 257         (void) nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP);
 258         (void) nvlist_add_uint8(nvl, MCINTEL_NVLIST_VERSTR,
 259             MCINTEL_NVLIST_VERS);
 260         (void) nvlist_add_string(nvl, MCINTEL_NVLIST_MEM, inhm_mc_name());
 261         (void) nvlist_add_uint8(nvl, MCINTEL_NVLIST_NMEM, 1);
 262         (void) nvlist_add_uint8(nvl, MCINTEL_NVLIST_NRANKS, 4);
 263         inhm_dimmlist(chip, nvl);

 265         if (inhm_mc_nvl[chip])
 265         nvlist_free(inhm_mc_nvl[chip]);
 266         inhm_mc_nvl[chip] = nvl;
 267 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   96774 Mon Feb 15 12:56:16 2016
new/usr/src/uts/intel/io/pci/pci_boot.c
6659 nvlist_free(NULL) is a no-op
**********************************************************
______unchanged_portion_omitted_

 328 static int
 329 pci_cache_pack_nvlist(nvf_handle_t hdl, nvlist_t **ret_nvl)
 330 {
 331         int             rval;
 332         nvlist_t        *nvl, *sub_nvl;
 333         list_t          *listp;
 334         pua_node_t      *pua;
 335         char            buf[13];

 337         ASSERT(RW_WRITE_HELD(nvf_lock(hdl)));

 339         rval = nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP);
 340         if (rval != DDI_SUCCESS) {
 341                 nvf_error("%s: nvlist alloc error %d\n",
 342                     nvf_cache_name(hdl), rval);
 343                 return (DDI_FAILURE);
 344         }

 346         sub_nvl = NULL;
 347         rval = nvlist_alloc(&sub_nvl, NV_UNIQUE_NAME, KM_SLEEP);
 348         if (rval != DDI_SUCCESS)
 349                 goto error;

 351         listp = nvf_list(hdl);
 352         for (pua = list_head(listp); pua != NULL;
 353             pua = list_next(listp, pua)) {
 354                 (void) snprintf(buf, sizeof (buf), "%d", pua->pua_index);
 355                 rval = nvlist_add_int32(sub_nvl, buf, pua->pua_addr);
 356                 if (rval != DDI_SUCCESS)
 357                         goto error;
 358         }

 360         rval = nvlist_add_nvlist(nvl, "table", sub_nvl);
 361         if (rval != DDI_SUCCESS)
 362                 goto error;
 363         nvlist_free(sub_nvl);

 365         *ret_nvl = nvl;
 366         return (DDI_SUCCESS);

 368 error:
 369         if (sub_nvl)
 370         nvlist_free(sub_nvl);
 371         ASSERT(nvl);
 372         nvlist_free(nvl);
 373         *ret_nvl = NULL;
 374         return (DDI_FAILURE);
 375 }
______unchanged_portion_omitted_
```