```
**********************************************************
   10274 Thu Sep 10 10:06:57 2015
new/usr/src/lib/libuutil/common/libuutil.h
patch uutil
**********************************************************
______unchanged_portion_omitted_

 304 typedef struct uu_avl_walk uu_avl_walk_t;

 306 typedef uintptr_t uu_avl_index_t;

 308 /*
 309  * avl trees: interface
 310  *
 311  * basic usage:
 312  *      typedef struct foo {
 313  *              ...
 314  *              uu_avl_node_t foo_node;
 315  *              ...
 316  *      } foo_t;
 317  *
 318  *      static int
 319  *      foo_compare(void *l_arg, void *r_arg, void *private)
 320  *      {
 321  *              foo_t *l = l_arg;
 322  *              foo_t *r = r_arg;
 323  *
 324  *              if (... l greater than r ...)
 325  *                      return (1);
 326  *              if (... l less than r ...)
 327  *                      return (-1);
 328  *              return (0);
 329  *      }
 330  *
 331  *      ...
 332  *              // at initialization time
 333  *              foo_pool = uu_avl_pool_create("foo_pool",
 334  *                  sizeof (foo_t), offsetof(foo_t, foo_node), foo_compare,
 335  *                  debugging? 0 : UU_AVL_POOL_DEBUG);
 336  *      ...
 337  */
 338 uu_avl_pool_t *uu_avl_pool_create(const char *, size_t, size_t,
 339     uu_compare_fn_t *, uint32_t);
 340 #define UU_AVL_POOL_DEBUG       0x00000001

 342 void uu_avl_pool_destroy(uu_avl_pool_t *);

 344 /*
 345  * usage:
 346  *
 347  *      foo_t *a;
 348  *      a = malloc(sizeof(*a));
 349  *      uu_avl_node_init(a, &a->foo_avl, pool);
 350  *      ...
 351  *      uu_avl_node_fini(a, &a->foo_avl, pool);
 352  *      free(a);
 353  */
 354 void uu_avl_node_init(void *, uu_avl_node_t *, uu_avl_pool_t *);
 355 void uu_avl_node_fini(void *, uu_avl_node_t *, uu_avl_pool_t *);

 357 uu_avl_t *uu_avl_create(uu_avl_pool_t *, void *_parent, uint32_t);
 358 #define UU_AVL_DEBUG    0x00000001

 360 void uu_avl_destroy(uu_avl_t *);        /* list must be empty */

 362 void uu_avl_recreate(uu_avl_t *);
```



```
 364 #endif /* ! codereview */
 365 size_t uu_avl_numnodes(uu_avl_t *);

 367 void *uu_avl_first(uu_avl_t *);
 368 void *uu_avl_last(uu_avl_t *);

 370 void *uu_avl_next(uu_avl_t *, void *);
 371 void *uu_avl_prev(uu_avl_t *, void *);

 373 int uu_avl_walk(uu_avl_t *, uu_walk_fn_t *, void *, uint32_t);

 375 uu_avl_walk_t *uu_avl_walk_start(uu_avl_t *, uint32_t);
 376 void *uu_avl_walk_next(uu_avl_walk_t *);
 377 void uu_avl_walk_end(uu_avl_walk_t *);

 379 void *uu_avl_find(uu_avl_t *, void *, void *, uu_avl_index_t *);
 380 void uu_avl_insert(uu_avl_t *, void *, uu_avl_index_t);

 382 void *uu_avl_nearest_next(uu_avl_t *, uu_avl_index_t);
 383 void *uu_avl_nearest_prev(uu_avl_t *, uu_avl_index_t);

 385 void *uu_avl_teardown(uu_avl_t *, void **);

 387 void uu_avl_remove(uu_avl_t *, void *);

 389 #ifdef  __cplusplus
 390 }
 391 #endif

 393 #endif  /* _LIBUUTIL_H */
```

```
**********************************************************
    2564 Thu Sep 10 10:06:57 2015
new/usr/src/lib/libuutil/common/mapfile-vers
patch uutil
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
#

#
# MAPFILE HEADER START
#
# WARNING:  STOP NOW.  DO NOT MODIFY THIS FILE.
# Object versioning must comply with the rules detailed in
#
#       usr/src/lib/README.mapfiles
#
# You should not be making modifications here until you've read the most current
# copy of that file. If you need help, contact a gatekeeper for guidance.
#
# MAPFILE HEADER END
#

$mapfile_version 2

SYMBOL_VERSION SUNWprivate_1.1 {
    global:
        uu_alt_exit;
        uu_avl_create;
        uu_avl_destroy;
        uu_avl_find;
        uu_avl_first;
        uu_avl_insert;
        uu_avl_last;
        uu_avl_nearest_next;
        uu_avl_nearest_prev;
        uu_avl_next;
        uu_avl_node_fini;
        uu_avl_node_init;
        uu_avl_numnodes;
        uu_avl_pool_create;
        uu_avl_pool_destroy;
        uu_avl_prev;
        uu_avl_recreate;
#endif /* ! codereview */
        uu_avl_remove;
```



```
        uu_avl_teardown;
        uu_avl_walk;
        uu_avl_walk_end;
        uu_avl_walk_next;
        uu_avl_walk_start;
        uu_check_name;
        uu_die;
        uu_dprintf;
        uu_dprintf_create;
        uu_dprintf_destroy;
        uu_dprintf_getname;
        uu_dump;
        uu_error;
        uu_exit_fatal;
        uu_exit_ok;
        uu_exit_usage;
        uu_free;
        uu_getpname;
        uu_list_create;
        uu_list_destroy;
        uu_list_find;
        uu_list_first;
        uu_list_insert;
        uu_list_insert_after;
        uu_list_insert_before;
        uu_list_last;
        uu_list_nearest_next;
        uu_list_nearest_prev;
        uu_list_next;
        uu_list_node_fini;
        uu_list_node_init;
        uu_list_numnodes;
        uu_list_pool_create;
        uu_list_pool_destroy;
        uu_list_prev;
        uu_list_remove;
        uu_list_teardown;
        uu_list_walk;
        uu_list_walk_end;
        uu_list_walk_next;
        uu_list_walk_start;
        uu_memdup;
        uu_msprintf;
        uu_open_tmp;
        uu_setpname;
        uu_strbw;
        uu_strcaseeq;
        uu_strdup;
        uu_streq;
        uu_strerror;
        uu_strndup;
        uu_strtoint;
        uu_strtouint;
        uu_vdie;
        uu_vwarn;
        uu_vxdie;
        uu_warn;
        uu_xdie;
        uu_zalloc;
    local:
        *;
};
```

```
**********************************************************
   14180 Thu Sep 10 10:06:57 2015
new/usr/src/lib/libuutil/common/uu_avl.c
patch uutil
**********************************************************
```

```
/*
 * 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.
 *
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

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

#include "libuutil_common.h"

#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/avl.h>

static uu_avl_pool_t    uu_null_apool = { &uu_null_apool, &uu_null_apool };
static pthread_mutex_t  uu_apool_list_lock = PTHREAD_MUTEX_INITIALIZER;

/*
 * The index mark change on every insert and delete, to catch stale
 * references.
 *
 * We leave the low bit alone, since the avl code uses it.
 */
#define INDEX_MAX               (sizeof (uintptr_t) - 2)
#define INDEX_NEXT(m)           (((m) == INDEX_MAX)? 2 : ((m) + 2) & INDEX_MAX)

#define INDEX_DECODE(i)         ((i) & ~INDEX_MAX)
#define INDEX_ENCODE(p, n)      (((n) & ~INDEX_MAX) | (p)->ua_index)
#define INDEX_VALID(p, i)       (((i) & INDEX_MAX) == (p)->ua_index)
#define INDEX_CHECK(i)          (((i) & INDEX_MAX) != 0)

/*
 * When an element is inactive (not in a tree), we keep a marked pointer to
 * its containing pool in its first word, and a NULL pointer in its second.
 *
 * On insert, we use these to verify that it comes from the correct pool.
 */
#define NODE_ARRAY(p, n)        ((uintptr_t *)((uintptr_t)(n) + \
```



```
                                    (pp)->uap_nodeoffset))

#define POOL_TO_MARKER(pp) (((uintptr_t)(pp) | 1))

#define DEAD_MARKER             0xc4

uu_avl_pool_t *
uu_avl_pool_create(const char *name, size_t objsize, size_t nodeoffset,
    uu_compare_fn_t *compare_func, uint32_t flags)
{
        uu_avl_pool_t *pp, *next, *prev;

        if (name == NULL ||
            uu_check_name(name, UU_NAME_DOMAIN) == -1 ||
            nodeoffset + sizeof (uu_avl_node_t) > objsize ||
            compare_func == NULL) {
                uu_set_error(UU_ERROR_INVALID_ARGUMENT);
                return (NULL);
        }

        if (flags & ~UU_AVL_POOL_DEBUG) {
                uu_set_error(UU_ERROR_UNKNOWN_FLAG);
                return (NULL);
        }

        pp = uu_zalloc(sizeof (uu_avl_pool_t));
        if (pp == NULL) {
                uu_set_error(UU_ERROR_NO_MEMORY);
                return (NULL);
        }

        (void) strlcpy(pp->uap_name, name, sizeof (pp->uap_name));
        pp->uap_nodeoffset = nodeoffset;
        pp->uap_objsize = objsize;
        pp->uap_cmp = compare_func;
        if (flags & UU_AVL_POOL_DEBUG)
                pp->uap_debug = 1;
        pp->uap_last_index = 0;

        (void) pthread_mutex_init(&pp->uap_lock, NULL);

        pp->uap_null_avl.ua_next_enc = UU_PTR_ENCODE(&pp->uap_null_avl);
        pp->uap_null_avl.ua_prev_enc = UU_PTR_ENCODE(&pp->uap_null_avl);

        (void) pthread_mutex_lock(&uu_apool_list_lock);
        pp->uap_next = next = &uu_null_apool;
        pp->uap_prev = prev = next->uap_prev;
        next->uap_prev = pp;
        prev->uap_next = pp;
        (void) pthread_mutex_unlock(&uu_apool_list_lock);

        return (pp);
}
```

*_____unchanged_portion_omitted_*

```
void
uu_avl_destroy(uu_avl_t *ap)
{
        uu_avl_pool_t *pp = ap->ua_pool;

        if (ap->ua_debug) {
                if (avl_numnodes(&ap->ua_tree) != 0) {
                        uu_panic("uu_avl_destroy(%p): tree not empty\n",
                            (void *)ap);
                }
                if (ap->ua_null_walk.uaw_next != &ap->ua_null_walk ||
```

```
260                     ap->ua_null_walk.uaw_prev != &ap->ua_null_walk) {
261                         uu_panic("uu_avl_destroy(%p):  outstanding walkers\n",
262                             (void *)ap);
263                 }
264         }
265         (void) pthread_mutex_lock(&pp->uap_lock);
266         UU_AVL_PTR(ap->ua_next_enc)->ua_prev_enc = ap->ua_prev_enc;
267         UU_AVL_PTR(ap->ua_prev_enc)->ua_next_enc = ap->ua_next_enc;
268         (void) pthread_mutex_unlock(&pp->uap_lock);
269         ap->ua_prev_enc = UU_PTR_ENCODE(NULL);
270         ap->ua_next_enc = UU_PTR_ENCODE(NULL);

272         ap->ua_pool = NULL;
273         avl_destroy(&ap->ua_tree);

275         uu_free(ap);
276 }

278 void
279 uu_avl_recreate(uu_avl_t *ap)
280 {
281         uu_avl_pool_t *pp = ap->ua_pool;

283         avl_destroy(&ap->ua_tree);
284         avl_create(&ap->ua_tree, &uu_avl_node_compare, pp->uap_objsize,
285             pp->uap_nodeoffset);
286 #endif /* ! codereview */
287 }

289 size_t
290 uu_avl_numnodes(uu_avl_t *ap)
291 {
292         return (avl_numnodes(&ap->ua_tree));
293 }

295 void *
296 uu_avl_first(uu_avl_t *ap)
297 {
298         return (avl_first(&ap->ua_tree));
299 }

301 void *
302 uu_avl_last(uu_avl_t *ap)
303 {
304         return (avl_last(&ap->ua_tree));
305 }

307 void *
308 uu_avl_next(uu_avl_t *ap, void *node)
309 {
310         return (AVL_NEXT(&ap->ua_tree, node));
311 }

313 void *
314 uu_avl_prev(uu_avl_t *ap, void *node)
315 {
316         return (AVL_PREV(&ap->ua_tree, node));
317 }

319 static void
320 _avl_walk_init(uu_avl_walk_t *wp, uu_avl_t *ap, uint32_t flags)
321 {
322         uu_avl_walk_t *next, *prev;

324         int robust = (flags & UU_WALK_ROBUST);
325         int direction = (flags & UU_WALK_REVERSE)? -1 : 1;
```



```
327         (void) memset(wp, 0, sizeof (*wp));
328         wp->uaw_avl = ap;
329         wp->uaw_robust = robust;
330         wp->uaw_dir = direction;

332         if (direction > 0)
333                 wp->uaw_next_result = avl_first(&ap->ua_tree);
334         else
335                 wp->uaw_next_result = avl_last(&ap->ua_tree);

337         if (ap->ua_debug || robust) {
338                 wp->uaw_next = next = &ap->ua_null_walk;
339                 wp->uaw_prev = prev = next->uaw_prev;
340                 next->uaw_prev = wp;
341                 prev->uaw_next = wp;
342         }
343 }

345 static void *
346 _avl_walk_advance(uu_avl_walk_t *wp, uu_avl_t *ap)
347 {
348         void *np = wp->uaw_next_result;

350         avl_tree_t *t = &ap->ua_tree;

352         if (np == NULL)
353                 return (NULL);

355         wp->uaw_next_result = (wp->uaw_dir > 0)? AVL_NEXT(t, np) :
356             AVL_PREV(t, np);

358         return (np);
359 }

361 static void
362 _avl_walk_fini(uu_avl_walk_t *wp)
363 {
364         if (wp->uaw_next != NULL) {
365                 wp->uaw_next->uaw_prev = wp->uaw_prev;
366                 wp->uaw_prev->uaw_next = wp->uaw_next;
367                 wp->uaw_next = NULL;
368                 wp->uaw_prev = NULL;
369         }
370         wp->uaw_avl = NULL;
371         wp->uaw_next_result = NULL;
372 }

374 uu_avl_walk_t *
375 uu_avl_walk_start(uu_avl_t *ap, uint32_t flags)
376 {
377         uu_avl_walk_t *wp;

379         if (flags & ~(UU_WALK_ROBUST | UU_WALK_REVERSE)) {
380                 uu_set_error(UU_ERROR_UNKNOWN_FLAG);
381                 return (NULL);
382         }

384         wp = uu_zalloc(sizeof (*wp));
385         if (wp == NULL) {
386                 uu_set_error(UU_ERROR_NO_MEMORY);
387                 return (NULL);
388         }

390         _avl_walk_init(wp, ap, flags);
391         return (wp);
```

```
392 }
394 void *
395 uu_avl_walk_next(uu_avl_walk_t *wp)
396 {
397         return (_avl_walk_advance(wp, wp->uaw_avl));
398 }
400 void
401 uu_avl_walk_end(uu_avl_walk_t *wp)
402 {
403         _avl_walk_fini(wp);
404         uu_free(wp);
405 }
407 int
408 uu_avl_walk(uu_avl_t *ap, uu_walk_fn_t *func, void *private, uint32_t flags)
409 {
410         void *e;
411         uu_avl_walk_t my_walk;
413         int status = UU_WALK_NEXT;
415         if (flags & ~(UU_WALK_ROBUST | UU_WALK_REVERSE)) {
416                 uu_set_error(UU_ERROR_UNKNOWN_FLAG);
417                 return (-1);
418         }
420         _avl_walk_init(&my_walk, ap, flags);
421         while (status == UU_WALK_NEXT &&
422             (e = _avl_walk_advance(&my_walk, ap)) != NULL)
423                 status = (*func)(e, private);
424         _avl_walk_fini(&my_walk);
426         if (status >= 0)
427                 return (0);
428         uu_set_error(UU_ERROR_CALLBACK_FAILED);
429         return (-1);
430 }
432 void
433 uu_avl_remove(uu_avl_t *ap, void *elem)
434 {
435         uu_avl_walk_t *wp;
436         uu_avl_pool_t *pp = ap->ua_pool;
437         uintptr_t *na = NODE_ARRAY(pp, elem);
439         if (ap->ua_debug) {
440                 /*
441                  * invalidate outstanding uu_avl_index_ts.
442                  */
443                 ap->ua_index = INDEX_NEXT(ap->ua_index);
444         }
446         /*
447          * Robust walkers most be advanced, if we are removing the node
448          * they are currently using.  In debug mode, non-robust walkers
449          * are also on the walker list.
450          */
451         for (wp = ap->ua_null_walk.uaw_next; wp != &ap->ua_null_walk;
452             wp = wp->uaw_next) {
453                 if (wp->uaw_robust) {
454                         if (elem == wp->uaw_next_result)
455                                 (void) _avl_walk_advance(wp, ap);
456                 } else if (wp->uaw_next_result != NULL) {
457                         uu_panic("uu_avl_remove(%p, %p): active non-robust "
```



```
458                             "walker\n", (void *)ap, elem);
459                 }
460         }
462         avl_remove(&ap->ua_tree, elem);
464         na[0] = POOL_TO_MARKER(pp);
465         na[1] = 0;
466 }
468 void *
469 uu_avl_teardown(uu_avl_t *ap, void **cookie)
470 {
471         void *elem = avl_destroy_nodes(&ap->ua_tree, cookie);
473         if (elem != NULL) {
474                 uu_avl_pool_t *pp = ap->ua_pool;
475                 uintptr_t *na = NODE_ARRAY(pp, elem);
477                 na[0] = POOL_TO_MARKER(pp);
478                 na[1] = 0;
479         }
480         return (elem);
481 }
483 void *
484 uu_avl_find(uu_avl_t *ap, void *elem, void *private, uu_avl_index_t *out)
485 {
486         struct uu_avl_node_compare_info info;
487         void *result;
489         info.ac_compare = ap->ua_pool->uap_cmp;
490         info.ac_private = private;
491         info.ac_right = elem;
492         info.ac_found = NULL;
494         result = avl_find(&ap->ua_tree, &info, out);
495         if (out != NULL)
496                 *out = INDEX_ENCODE(ap, *out);
498         if (ap->ua_debug && result != NULL)
499                 uu_panic("uu_avl_find: internal error: avl_find succeeded\n");
501         return (info.ac_found);
502 }
504 void
505 uu_avl_insert(uu_avl_t *ap, void *elem, uu_avl_index_t idx)
506 {
507         if (ap->ua_debug) {
508                 uu_avl_pool_t *pp = ap->ua_pool;
509                 uintptr_t *na = NODE_ARRAY(pp, elem);
511                 if (na[1] != 0)
512                         uu_panic("uu_avl_insert(%p, %p, %p): node already "
513                             "in tree, or corrupt\n",
514                             (void *)ap, elem, (void *)idx);
515                 if (na[0] == 0)
516                         uu_panic("uu_avl_insert(%p, %p, %p): node not "
517                             "initialized\n",
518                             (void *)ap, elem, (void *)idx);
519                 if (na[0] != POOL_TO_MARKER(pp))
520                         uu_panic("uu_avl_insert(%p, %p, %p): node from "
521                             "other pool, or corrupt\n",
522                             (void *)ap, elem, (void *)idx);
```

```
524                 if (!INDEX_VALID(ap, idx))
525                         uu_panic("uu_avl_insert(%p, %p, %p): %s\n",
526                             (void *)ap, elem, (void *)idx,
527                             INDEX_CHECK(idx)? "outdated index" :
528                             "invalid index");
529
530                 /*
531                  * invalidate outstanding uu_avl_index_ts.
532                  */
533                 ap->ua_index = INDEX_NEXT(ap->ua_index);
534         }
535         avl_insert(&ap->ua_tree, elem, INDEX_DECODE(idx));
536 }
537
538 void *
539 uu_avl_nearest_next(uu_avl_t *ap, uu_avl_index_t idx)
540 {
541         if (ap->ua_debug && !INDEX_VALID(ap, idx))
542                 uu_panic("uu_avl_nearest_next(%p, %p): %s\n",
543                     (void *)ap, (void *)idx, INDEX_CHECK(idx)?
544                     "outdated index" : "invalid index");
545         return (avl_nearest(&ap->ua_tree, INDEX_DECODE(idx), AVL_AFTER));
546 }
547
548 void *
549 uu_avl_nearest_prev(uu_avl_t *ap, uu_avl_index_t idx)
550 {
551         if (ap->ua_debug && !INDEX_VALID(ap, idx))
552                 uu_panic("uu_avl_nearest_prev(%p, %p): %s\n",
553                     (void *)ap, (void *)idx, INDEX_CHECK(idx)?
554                     "outdated index" : "invalid index");
555         return (avl_nearest(&ap->ua_tree, INDEX_DECODE(idx), AVL_BEFORE));
556 }
557
558 /*
559  * called from uu_lockup() and uu_release(), as part of our fork1()-safety.
560  */
561 void
562 uu_avl_lockup(void)
563 {
564         uu_avl_pool_t *pp;
565
566         (void) pthread_mutex_lock(&uu_apool_list_lock);
567         for (pp = uu_null_apool.uap_next; pp != &uu_null_apool;
568             pp = pp->uap_next)
569                 (void) pthread_mutex_lock(&pp->uap_lock);
570 }
571
572 void
573 uu_avl_release(void)
574 {
575         uu_avl_pool_t *pp;
576
577         for (pp = uu_null_apool.uap_next; pp != &uu_null_apool;
578             pp = pp->uap_next)
579                 (void) pthread_mutex_unlock(&pp->uap_lock);
580         (void) pthread_mutex_unlock(&uu_apool_list_lock);
581 }
```

```
**********************************************************
   10388 Thu Sep 10 10:06:57 2015
new/usr/src/lib/libzfs/common/libzfs_config.c
patch big
**********************************************************
```

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

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

/*
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

/*
 * The pool configuration repository is stored in /etc/zfs/zpool.cache as a
 * single packed nvlist.  While it would be nice to just read in this
 * file from userland, this wouldn't work from a local zone.  So we have to have
 * a zpool ioctl to return the complete configuration for all pools.  In the
 * global zone, this will be identical to reading the file and unpacking it in
 * userland.
 */

#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stddef.h>
#include <string.h>
#include <unistd.h>
#include <libintl.h>
#include <libuutil.h>

#include "libzfs_impl.h"

typedef struct config_node {
        char            *cn_name;
        nvlist_t        *cn_config;
        uu_avl_node_t   cn_avl;
} config_node_t;

/* ARGSUSED */
static int
config_node_compare(const void *a, const void *b, void *unused)
```



```
{
        int ret;

        const config_node_t *ca = (config_node_t *)a;
        const config_node_t *cb = (config_node_t *)b;

        ret = strcmp(ca->cn_name, cb->cn_name);

        if (ret < 0)
                return (-1);
        else if (ret > 0)
                return (1);
        else
                return (0);
}

void
namespace_clear(libzfs_handle_t *hdl)
{
        if (hdl->libzfs_ns_avl) {
                config_node_t *cn;
                void *cookie = NULL;

                while ((cn = uu_avl_teardown(hdl->libzfs_ns_avl,
                    &cookie)) != NULL) {
                        nvlist_free(cn->cn_config);
                        free(cn->cn_name);
                        free(cn);
                }

                uu_avl_destroy(hdl->libzfs_ns_avl);
                hdl->libzfs_ns_avl = NULL;
        }

        if (hdl->libzfs_ns_avlpool) {
                uu_avl_pool_destroy(hdl->libzfs_ns_avlpool);
                hdl->libzfs_ns_avlpool = NULL;
        }
}

/*
 * Loads the pool namespace, or re-loads it if the cache has changed.
 */
static int
namespace_reload(libzfs_handle_t *hdl)
{
        nvlist_t *config;
        config_node_t *cn;
        nvpair_t *elem;
        zfs_cmd_t zc = { 0 };
        void *cookie;

        if (hdl->libzfs_ns_gen == 0) {
                /*
                 * This is the first time we've accessed the configuration
                 * cache.  Initialize the AVL tree and then fall through to the
                 * common code.
                 */
                if ((hdl->libzfs_ns_avlpool = uu_avl_pool_create("config_pool",
                    sizeof (config_node_t),
                    offsetof(config_node_t, cn_avl),
                    config_node_compare, UU_DEFAULT)) == NULL)
                        return (no_memory(hdl));

                if ((hdl->libzfs_ns_avl = uu_avl_create(hdl->libzfs_ns_avlpool,
                    NULL, UU_DEFAULT)) == NULL)
```

```
128                         return (no_memory(hdl));
129         }
130
131         if (zcmd_alloc_dst_nvlist(hdl, &zc, 0) != 0)
132                 return (-1);
133
134         for (;;) {
135                 zc.zc_cookie = hdl->libzfs_ns_gen;
136                 if (ioctl(hdl->libzfs_fd, ZFS_IOC_POOL_CONFIGS, &zc) != 0) {
137                         switch (errno) {
138                         case EEXIST:
139                                 /*
140                                  * The namespace hasn't changed.
141                                  */
142                                 zcmd_free_nvlists(&zc);
143                                 return (0);
144
145                         case ENOMEM:
146                                 if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
147                                         zcmd_free_nvlists(&zc);
148                                         return (-1);
149                                 }
150                                 break;
151
152                         default:
153                                 zcmd_free_nvlists(&zc);
154                                 return (zfs_standard_error(hdl, errno,
155                                     dgettext(TEXT_DOMAIN, "failed to read "
156                                     "pool configuration")));
157                         }
158                 } else {
159                         hdl->libzfs_ns_gen = zc.zc_cookie;
160                         break;
161                 }
162         }
163
164         if (zcmd_read_dst_nvlist(hdl, &zc, &config) != 0) {
165                 zcmd_free_nvlists(&zc);
166                 return (-1);
167         }
168
169         zcmd_free_nvlists(&zc);
170
171         /*
172          * Clear out any existing configuration information, and recreate
173          * the AVL tree.
 29          * Clear out any existing configuration information.
174          */
175         cookie = NULL;
176         while ((cn = uu_avl_teardown(hdl->libzfs_ns_avl, &cookie)) != NULL) {
177                 nvlist_free(cn->cn_config);
178                 free(cn->cn_name);
179                 free(cn);
180         }
181
182         uu_avl_recreate(hdl->libzfs_ns_avl);
183 #endif /* ! codereview */
184
185         elem = NULL;
186         while ((elem = nvlist_next_nvpair(config, elem)) != NULL) {
187                 nvlist_t *child;
188                 uu_avl_index_t where;
189
190                 if ((cn = zfs_alloc(hdl, sizeof (config_node_t))) == NULL) {
191                         nvlist_free(config);
192                         return (-1);
```



```
193                 }
194
195                 if ((cn->cn_name = zfs_strdup(hdl,
196                     nvpair_name(elem))) == NULL) {
197                         free(cn);
198                         nvlist_free(config);
199                         return (-1);
200                 }
201
202                 verify(nvpair_value_nvlist(elem, &child) == 0);
203                 if (nvlist_dup(child, &cn->cn_config, 0) != 0) {
204                         free(cn->cn_name);
205                         free(cn);
206                         nvlist_free(config);
207                         return (no_memory(hdl));
208                 }
209                 verify(uu_avl_find(hdl->libzfs_ns_avl, cn, NULL, &where)
210                     == NULL);
211
212                 uu_avl_insert(hdl->libzfs_ns_avl, cn, where);
213         }
214
215         nvlist_free(config);
216         return (0);
217 }
218
219 /*
220  * Retrieve the configuration for the given pool.  The configuration is a nvlist
221  * describing the vdevs, as well as the statistics associated with each one.
222  */
223 nvlist_t *
224 zpool_get_config(zpool_handle_t *zhp, nvlist_t **oldconfig)
225 {
226         if (oldconfig)
227                 *oldconfig = zhp->zpool_old_config;
228         return (zhp->zpool_config);
229 }
230
231 /*
232  * Retrieves a list of enabled features and their refcounts and caches it in
233  * the pool handle.
234  */
235 nvlist_t *
236 zpool_get_features(zpool_handle_t *zhp)
237 {
238         nvlist_t *config, *features;
239
240         config = zpool_get_config(zhp, NULL);
241
242         if (config == NULL || !nvlist_exists(config,
243             ZPOOL_CONFIG_FEATURE_STATS)) {
244                 int error;
245                 boolean_t missing = B_FALSE;
246
247                 error = zpool_refresh_stats(zhp, &missing);
248
249                 if (error != 0 || missing)
250                         return (NULL);
251
252                 config = zpool_get_config(zhp, NULL);
253         }
254
255         verify(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_FEATURE_STATS,
256             &features) == 0);
257
258         return (features);
```

```
259 }

261 /*
262  * Refresh the vdev statistics associated with the given pool.  This is used in
263  * iostat to show configuration changes and determine the delta from the last
264  * time the function was called.  This function can fail, in case the pool has
265  * been destroyed.
266  */
267 int
268 zpool_refresh_stats(zpool_handle_t *zhp, boolean_t *missing)
269 {
270         zfs_cmd_t zc = { 0 };
271         int error;
272         nvlist_t *config;
273         libzfs_handle_t *hdl = zhp->zpool_hdl;

275         *missing = B_FALSE;
276         (void) strcpy(zc.zc_name, zhp->zpool_name);

278         if (zhp->zpool_config_size == 0)
279                 zhp->zpool_config_size = 1 << 16;

281         if (zcmd_alloc_dst_nvlist(hdl, &zc, zhp->zpool_config_size) != 0)
282                 return (-1);

284         for (;;) {
285                 if (ioctl(zhp->zpool_hdl->libzfs_fd, ZFS_IOC_POOL_STATS,
286                     &zc) == 0) {
287                         /*
288                          * The real error is returned in the zc_cookie field.
289                          */
290                         error = zc.zc_cookie;
291                         break;
292                 }

294                 if (errno == ENOMEM) {
295                         if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
296                                 zcmd_free_nvlists(&zc);
297                                 return (-1);
298                         }
299                 } else {
300                         zcmd_free_nvlists(&zc);
301                         if (errno == ENOENT || errno == EINVAL)
302                                 *missing = B_TRUE;
303                         zhp->zpool_state = POOL_STATE_UNAVAIL;
304                         return (0);
305                 }
306         }

308         if (zcmd_read_dst_nvlist(hdl, &zc, &config) != 0) {
309                 zcmd_free_nvlists(&zc);
310                 return (-1);
311         }

313         zcmd_free_nvlists(&zc);

315         zhp->zpool_config_size = zc.zc_nvlist_dst_size;

317         if (zhp->zpool_config != NULL) {
318                 uint64_t oldtxg, newtxg;

320                 verify(nvlist_lookup_uint64(zhp->zpool_config,
321                     ZPOOL_CONFIG_POOL_TXG, &oldtxg) == 0);
322                 verify(nvlist_lookup_uint64(config,
323                     ZPOOL_CONFIG_POOL_TXG, &newtxg) == 0);
```



```
325                 if (zhp->zpool_old_config != NULL)
326                         nvlist_free(zhp->zpool_old_config);

328                 if (oldtxg != newtxg) {
329                         nvlist_free(zhp->zpool_config);
330                         zhp->zpool_old_config = NULL;
331                 } else {
332                         zhp->zpool_old_config = zhp->zpool_config;
333                 }
334         }

336         zhp->zpool_config = config;
337         if (error)
338                 zhp->zpool_state = POOL_STATE_UNAVAIL;
339         else
340                 zhp->zpool_state = POOL_STATE_ACTIVE;

342         return (0);
343 }

345 /*
346  * If the __ZFS_POOL_RESTRICT environment variable is set we only iterate over
347  * pools it lists.
348  *
349  * This is an undocumented feature for use during testing only.
350  *
351  * This function returns B_TRUE if the pool should be skipped
352  * during iteration.
353  */
354 static boolean_t
355 check_restricted(const char *poolname)
356 {
357         static boolean_t initialized = B_FALSE;
358         static char *restricted = NULL;

360         const char *cur, *end;
361         int len, namelen;

363         if (!initialized) {
364                 initialized = B_TRUE;
365                 restricted = getenv("__ZFS_POOL_RESTRICT");
366         }

368         if (NULL == restricted)
369                 return (B_FALSE);

371         cur = restricted;
372         namelen = strlen(poolname);
373         do {
374                 end = strchr(cur, ' ');
375                 len = (NULL == end) ? strlen(cur) : (end - cur);

377                 if (len == namelen && 0 == strncmp(cur, poolname, len)) {
378                         return (B_FALSE);
379                 }

381                 cur += (len + 1);
382         } while (NULL != end);

384         return (B_TRUE);
385 }

387 /*
388  * Iterate over all pools in the system.
389  */
390 int
```

```
zpool_iter(libzfs_handle_t *hdl, zpool_iter_f func, void *data)
{
        config_node_t *cn;
        zpool_handle_t *zhp;
        int ret;

        /*
         * If someone makes a recursive call to zpool_iter(), we want to avoid
         * refreshing the namespace because that will invalidate the parent
         * context.  We allow recursive calls, but simply re-use the same
         * namespace AVL tree.
         */
        if (!hdl->libzfs_pool_iter && namespace_reload(hdl) != 0)
                return (-1);

        hdl->libzfs_pool_iter++;
        for (cn = uu_avl_first(hdl->libzfs_ns_avl); cn != NULL;
            cn = uu_avl_next(hdl->libzfs_ns_avl, cn)) {

                if (check_restricted(cn->cn_name))
                        continue;

                if (zpool_open_silent(hdl, cn->cn_name, &zhp) != 0) {
                        hdl->libzfs_pool_iter--;
                        return (-1);
                }

                if (zhp == NULL)
                        continue;

                if ((ret = func(zhp, data)) != 0) {
                        hdl->libzfs_pool_iter--;
                        return (ret);
                }
        }
        hdl->libzfs_pool_iter--;

        return (0);
}

/*
 * Iterate over root datasets, calling the given function for each.  The zfs
 * handle passed each time must be explicitly closed by the callback.
 */
int
zfs_iter_root(libzfs_handle_t *hdl, zfs_iter_f func, void *data)
{
        config_node_t *cn;
        zfs_handle_t *zhp;
        int ret;

        if (namespace_reload(hdl) != 0)
                return (-1);

        for (cn = uu_avl_first(hdl->libzfs_ns_avl); cn != NULL;
            cn = uu_avl_next(hdl->libzfs_ns_avl, cn)) {

                if (check_restricted(cn->cn_name))
                        continue;

                if ((zhp = make_dataset_handle(hdl, cn->cn_name)) == NULL)
                        continue;

                if ((ret = func(zhp, data)) != 0)
                        return (ret);
        }
```



```
        return (0);
}
```