

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

_____ 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 */
```

new/usr/src/lib/libuutil/common/mapfile-vers

```
*****
2564 Thu Sep 10 10:06:57 2015
new/usr/src/lib/libuutil/common/mapfile-vers
patch util
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 # Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
23 #

25 #
26 # MAPFILE HEADER START
27 #
28 # WARNING: STOP NOW. DO NOT MODIFY THIS FILE.
29 # Object versioning must comply with the rules detailed in
30 #
31 #     usr/src/lib/README.mapfiles
32 #
33 # You should not be making modifications here until you've read the most current
34 # copy of that file. If you need help, contact a gatekeeper for guidance.
35 #
36 # MAPFILE HEADER END
37 #

39 $mapfile_version 2

41 SYMBOL_VERSION SUNWprivate_1.1 {
42     global:
43         uu_alt_exit;
44         uu_avl_create;
45         uu_avl_destroy;
46         uu_avl_find;
47         uu_avl_first;
48         uu_avl_insert;
49         uu_avl_last;
50         uu_avl_nearest_next;
51         uu_avl_nearest_prev;
52         uu_avl_next;
53         uu_avl_node_fini;
54         uu_avl_node_init;
55         uu_avl_numnodes;
56         uu_avl_pool_create;
57         uu_avl_pool_destroy;
58         uu_avl_prev;
59         uu_avl_recreate;
60 #endif /* ! codereview */
61         uu_avl_remove;
```

1

new/usr/src/lib/libuutil/common/mapfile-vers

```
62         uu_avl_teardown;
63         uu_avl_walk;
64         uu_avl_walk_end;
65         uu_avl_walk_next;
66         uu_avl_walk_start;
67         uu_check_name;
68         uu_die;
69         uu_dprintf;
70         uu_dprintf_create;
71         uu_dprintf_destroy;
72         uu_dprintf_getname;
73         uu_dump;
74         uu_error;
75         uu_exit_fatal;
76         uu_exit_ok;
77         uu_exit_usage;
78         uu_free;
79         uu_getpname;
80         uu_list_create;
81         uu_list_destroy;
82         uu_list_find;
83         uu_list_first;
84         uu_list_insert;
85         uu_list_insert_after;
86         uu_list_insert_before;
87         uu_list_last;
88         uu_list_nearest_next;
89         uu_list_nearest_prev;
90         uu_list_next;
91         uu_list_node_fini;
92         uu_list_node_init;
93         uu_list_numnodes;
94         uu_list_pool_create;
95         uu_list_pool_destroy;
96         uu_list_prev;
97         uu_list_remove;
98         uu_list_teardown;
99         uu_list_walk;
100        uu_list_walk_end;
101        uu_list_walk_next;
102        uu_list_walk_start;
103        uu_memdup;
104        uu_msprintf;
105        uu_open_tmp;
106        uu_setpname;
107        uu_strbw;
108        uu_strcasseq;
109        uu_strdup;
110        uu_streq;
111        uu_strerror;
112        uu_strdup;
113        uu strtoint;
114        uu strtouint;
115        uu_vdie;
116        uu_vwarn;
117        uu_vxdie;
118        uu_warn;
119        uu_xdie;
120        uu_zalloc;
121        local:
122        *;
123    },
```

2

new/usr/src/lib/libuutil/common/uu_avl.c

```
*****
14180 Thu Sep 10 10:06:57 2015
new/usr/src/lib/libuutil/common/uu_avl.c
patch util
*****
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 2008 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 *
25 * Copyright 2015 Nexenta Systems, Inc. All rights reserved.
26 #endif /* ! codereview */
27 */

24 #pragma ident "%Z%%M% %I%     %E% SMI"
29 #include "libuutil_common.h"

31 #include <stdlib.h>
32 #include <string.h>
33 #include <unistd.h>
34 #include <sys/avl.h>

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

39 /*
40 * The index mark change on every insert and delete, to catch stale
41 * references.
42 *
43 * We leave the low bit alone, since the avl code uses it.
44 */
45 #define INDEX_MAX      (sizeof (uintptr_t) - 2)
46 #define INDEX_NEXT(m)   (((m) == INDEX_MAX)? 2 : ((m) + 2) & INDEX_MAX)

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

53 /*
54 * When an element is inactive (not in a tree), we keep a marked pointer to
55 * its containing pool in its first word, and a NULL pointer in its second.
56 *
57 * On insert, we use these to verify that it comes from the correct pool.
58 */
59 #define NODE_ARRAY(p, n)   ((uintptr_t *)((uintptr_t)(n) + \
```

1

new/usr/src/lib/libuutil/common/uu_avl.c

```
60                                     (pp)->uap_nodeoffset))
62 #define POOL_TO_MARKER(pp) (((uintptr_t)(pp) | 1))
64 #define DEAD_MARKER          0xc4
66 uu_avl_pool_t *
67 uu_avl_pool_create(const char *name, size_t objsize, size_t nodeoffset,
68                     uu_compare_fn_t *compare_func, uint32_t flags)
69 {
70     uu_avl_pool_t *pp, *next, *prev;
72     if (name == NULL ||
73         uu_check_name(name, UU_NAME_DOMAIN) == -1 ||
74         nodeoffset + sizeof (uu_avl_node_t) > objsize ||
75         compare_func == NULL) {
76         uu_set_error(UU_ERROR_INVALID_ARGUMENT);
77         return (NULL);
78     }
79     if (flags & UU_AVL_POOL_DEBUG) {
80         uu_set_error(UU_ERROR_UNKNOWN_FLAG);
81         return (NULL);
82     }
83
84     pp = uu_zalloc(sizeof (uu_avl_pool_t));
85     if (pp == NULL) {
86         uu_set_error(UU_ERROR_NO_MEMORY);
87         return (NULL);
88     }
89
90     (void) strlcpy(pp->uap_name, name, sizeof (pp->uap_name));
91     pp->uap_nodeoffset = nodeoffset;
92     pp->uap_objsize = objsize;
93     pp->uap_cmp = compare_func;
94     if (flags & UU_AVL_POOL_DEBUG)
95         pp->uap_debug = 1;
96     pp->uap_last_index = 0;
97
98     (void) pthread_mutex_init(&pp->uap_lock, NULL);
99
100    pp->uap_null_avl.ua_next_enc = UU_PTR_ENCODE(&pp->uap_null_avl);
101    pp->uap_null_avl.ua_prev_enc = UU_PTR_ENCODE(&pp->uap_null_avl);
102
103    (void) pthread_mutex_lock(&uu_apool_list_lock);
104    pp->uap_next = next = &uu_null_apool;
105    pp->uap_prev = prev = next->uap_prev;
106    next->uap_prev = pp;
107    prev->uap_next = pp;
108    (void) pthread_mutex_unlock(&uu_apool_list_lock);
109
110    return (pp);
111 }

112 }  

113 unchanged_portion_omitted
249 void
250 uu_avl_destroy(uu_avl_t *ap)
251 {
252     uu_avl_pool_t *pp = ap->ua_pool;
253
254     if (ap->ua_debug) {
255         if (avl_numnodes(&ap->ua_tree) != 0) {
256             uu_panic("uu_avl_destroy(%p): tree not empty\n",
257                     (void *)ap);
258         }
259     }
260     if (ap->ua_null_walk.uaw_next != &ap->ua_null_walk ||
```

2

```

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;
349
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
380     if (flags & ~(UU_WALK_ROBUST | UU_WALK_REVERSE)) {
381         uu_set_error(UU_ERROR_UNKNOWN_FLAG);
382         return (NULL);
383     }
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 }

399 void
400 uu_avl_walk_end(uu_avl_walk_t *wp)
401 {
402     _avl_walk_fini(wp);
403     uu_free(wp);
404 }
405 }

406 int
407 uu_avl_walk(uu_avl_t *ap, uu_walk_fn_t *func, void *private, uint32_t flags)
408 {
409     void *e;
410     uu_avl_walk_t my_walk;
411
412     int status = UU_WALK_NEXT;
413
414     if (flags & ~(UU_WALK_ROBUST | UU_WALK_REVERSE)) {
415         uu_set_error(UU_ERROR_UNKNOWN_FLAG);
416         return (-1);
417     }
418
419     _avl_walk_init(&my_walk, ap, flags);
420     while (status == UU_WALK_NEXT &&
421            (e = _avl_walk_advance(&my_walk, ap)) != NULL)
422         status = (*func)(e, private);
423     _avl_walk_fini(&my_walk);
424
425     if (status >= 0)
426         return (0);
427     uu_set_error(UU_ERROR_CALLBACK_FAILED);
428
429     return (-1);
430 }

431 void
432 uu_avl_remove(uu_avl_t *ap, void *elem)
433 {
434     uu_avl_walk_t *wp;
435     uu_avl_pool_t *pp = ap->ua_pool;
436     uintptr_t *na = NODE_ARRAY(pp, elem);
437
438     if (ap->ua_debug) {
439         /*
440          * invalidate outstanding uu_avl_index_ts.
441          */
442         ap->ua_index = INDEX_NEXT(ap->ua_index);
443     }
444
445     /*
446      * Robust walkers must be advanced, if we are removing the node
447      * they are currently using. In debug mode, non-robust walkers
448      * are also on the walker list.
449      */
450     for (wp = ap->ua_null_walk.uaw_next; wp != &ap->ua_null_walk;
451         wp = wp->uaw_next) {
452         if (wp->uaw_robust) {
453             if (elem == wp->uaw_next_result)
454                 (void) _avl_walk_advance(wp, ap);
455         } else if (wp->uaw_next_result != NULL) {
456             uu_panic("uu_avl_remove(%p, %p): active non-robust "
457

```

```

458                                         "walker\n", (void *)ap, elem);
459     }
460 }
461
462     avl_remove(&ap->ua_tree, elem);
463     na[0] = POOL_TO_MARKER(pp);
464     na[1] = 0;
465 }
466 }

467 void *
468 uu_avl_teardown(uu_avl_t *ap, void **cookie)
469 {
470     void *elem = avl_destroy_nodes(&ap->ua_tree, cookie);
471
472     if (elem != NULL) {
473         uu_avl_pool_t *pp = ap->ua_pool;
474         uintptr_t *na = NODE_ARRAY(pp, elem);
475
476         na[0] = POOL_TO_MARKER(pp);
477         na[1] = 0;
478     }
479
480     return (elem);
481 }

482 void *
483 uu_avl_find(uu_avl_t *ap, void *elem, void *private, uu_avl_index_t *out)
484 {
485     struct uu_avl_node_compare_info info;
486     void *result;
487
488     info.ac_COMPARE = ap->ua_pool->uap_CMP;
489     info.ac_private = private;
490     info.ac_right = elem;
491     info.ac_found = NULL;
492
493     result = avl_find(&ap->ua_tree, &info, out);
494     if (out != NULL)
495         *out = INDEX_ENCODE(ap, *out);
496
497     if (ap->ua_debug && result != NULL)
498         uu_panic("uu_avl_find: internal error: avl_find succeeded\n");
499
500     return (info.ac_found);
501 }

502 }

503 void
504 uu_avl_insert(uu_avl_t *ap, void *elem, uu_avl_index_t idx)
505 {
506     if (ap->ua_debug) {
507         uu_avl_pool_t *pp = ap->ua_pool;
508         uintptr_t *na = NODE_ARRAY(pp, elem);
509
510         if (na[1] != 0)
511             uu_panic("uu_avl_insert(%p, %p, %p): node already "
512                     "in tree, or corrupt\n",
513                     (void *)ap, elem, (void *)idx);
514         if (na[0] == 0)
515             uu_panic("uu_avl_insert(%p, %p, %p): node not "
516                     "initialized\n",
517                     (void *)ap, elem, (void *)idx);
518         if (na[0] != POOL_TO_MARKER(pp))
519             uu_panic("uu_avl_insert(%p, %p, %p): node from "
520                     "other pool, or corrupt\n",
521                     (void *)ap, elem, (void *)idx);
522     }
523 }

524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
255
```

```
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 }
```

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

```
*****
10388 Thu Sep 10 10:06:57 2015
new/usr/src/lib/libzfs/common/libzfs_config.c
patch big
*****
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 /*
28 * Copyright (c) 2012 by Delphix. All rights reserved.
29 * Copyright 2015 Nexenta Systems, Inc. All rights reserved.
30 #endif /* ! codereview */
31 */
33 /*
34 * The pool configuration repository is stored in /etc/zfs/zpool.cache as a
35 * single packed nvlist. While it would be nice to just read in this
36 * file from userland, this wouldn't work from a local zone. So we have to have
37 * a zpool ioctl to return the complete configuration for all pools. In the
38 * global zone, this will be identical to reading the file and unpacking it in
39 * userland.
40 */
42 #include <errno.h>
43 #include <sys/stat.h>
44 #include <fcntl.h>
45 #include <stddef.h>
46 #include <string.h>
47 #include <unistd.h>
48 #include <libintl.h>
49 #include <libuutil.h>
51 #include "libzfs_impl.h"
53 typedef struct config_node {
54     char          *cn_name;
55     nvlist_t      *cn_config;
56     uu_avl_node_t cn_avl;
57 } config_node_t;
59 /* ARGSUSED */
60 static int
61 config_node_compare(const void *a, const void *b, void *unused)
```

1

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

```
62 {
63     int ret;
65     const config_node_t *ca = (config_node_t *)a;
66     const config_node_t *cb = (config_node_t *)b;
68     ret = strcmp(ca->cn_name, cb->cn_name);
70     if (ret < 0)
71         return (-1);
72     else if (ret > 0)
73         return (1);
74     else
75         return (0);
76 }
78 void
79 namespace_clear(libzfs_handle_t *hdl)
80 {
81     if (hdl->libzfs_ns_avl) {
82         config_node_t *cn;
83         void *cookie = NULL;
85         while ((cn = uu_avl_teardown(hdl->libzfs_ns_avl,
86             &cookie)) != NULL) {
87             nvlist_free(cn->cn_config);
88             free(cn->cn_name);
89             free(cn);
90         }
92         uu_avl_destroy(hdl->libzfs_ns_avl);
93         hdl->libzfs_ns_avl = NULL;
94     }
96     if (hdl->libzfs_ns_avlpool) {
97         uu_avl_pool_destroy(hdl->libzfs_ns_avlpool);
98         hdl->libzfs_ns_avlpool = NULL;
99     }
102 /*
103 * Loads the pool namespace, or re-loads it if the cache has changed.
104 */
105 static int
106 namespace_reload(libzfs_handle_t *hdl)
107 {
108     nvlist_t *config;
109     config_node_t *cn;
110     nvpair_t *elem;
111     zfs_cmd_t zc = { 0 };
112     void *cookie;
114     if (hdl->libzfs_ns_gen == 0) {
115         /*
116             * This is the first time we've accessed the configuration
117             * cache. Initialize the AVL tree and then fall through to the
118             * common code.
119         */
120         if ((hdl->libzfs_ns_avlpool = uu_avl_pool_create("config_pool",
121             sizeof(config_node_t),
122             offsetof(config_node_t, cn_avl),
123             config_node_compare, UU_DEFAULT)) == NULL)
124             return (no_memory(hdl));
126         if ((hdl->libzfs_ns_avl = uu_avl_create(hdl->libzfs_ns_avlpool,
127             NULL, UU_DEFAULT)) == NULL)
```

2

```

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.
174      * Clear out any existing configuration information.
175      */
176     cookie = NULL;
177     while ((cn = uu_avl_teardown(hdl->libzfs_ns_avl, &cookie)) != NULL) {
178         nvlist_free(cn->cn_config);
179         free(cn->cn_name);
180         free(cn);
181     }
182
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);
205             nvlist_free(config);
206             return (no_memory(hdl));
207         }
208         verify(uu_avl_find(hdl->libzfs_ns_avl, cn, NULL, &where)
209               == NULL);
210
211         uu_avl_insert(hdl->libzfs_ns_avl, cn, where);
212     }
213
214     nvlist_free(config);
215     return (0);
216 }
217
218 /*
219  * Retrieve the configuration for the given pool.  The configuration is a nvlist
220  * describing the vdevs, as well as the statistics associated with each one.
221  */
222 nvlist_t *
223 zpool_get_config(zpool_handle_t *zhp, nvlist_t **oldconfig)
224 {
225     if (oldconfig)
226         *oldconfig = zhp->zpool_old_config;
227     return (zhp->zpool_config);
228 }
229
230 /*
231  * Retrieves a list of enabled features and their refcounts and caches it in
232  * the pool handle.
233  */
234 nvlist_t *
235 zpool_get_features(zpool_handle_t *zhp)
236 {
237     nvlist_t *config, *features;
238
239     config = zpool_get_config(zhp, NULL);
240
241     if (config == NULL || !nvlist_exists(config,
242                                         ZPOOL_CONFIG_FEATURE_STATS)) {
243         int error;
244         boolean_t missing = B_FALSE;
245
246         error = zpool_refresh_stats(zhp, &missing);
247
248         if (error != 0 || missing)
249             return (NULL);
250
251         config = zpool_get_config(zhp, NULL);
252     }
253
254     verify(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_FEATURE_STATS,
255                               &features) == 0);
256
257     return (features);
258 }
```

```

259 }
260 /*
261 * Refresh the vdev statistics associated with the given pool. This is used in
262 * iostat to show configuration changes and determine the delta from the last
263 * time the function was called. This function can fail, in case the pool has
264 * been destroyed.
265 */
266 int
267 zpool_refresh_stats(zpool_handle_t *zhp, boolean_t *missing)
268 {
269     zfs_cmd_t zc = { 0 };
270     int error;
271     nvlist_t *config;
272     libzfs_handle_t *hdl = zhp->zpool_hdl;
273
274     *missing = B_FALSE;
275     (void) strcpy(zc.zc_name, zhp->zpool_name);
276
277     if (zhp->zpool_config_size == 0)
278         zhp->zpool_config_size = 1 << 16;
279
280     if (zcmd_alloc_dst_nvlist(hdl, &zc, zhp->zpool_config_size) != 0)
281         return (-1);
282
283     for (;;) {
284         if (ioctl(zhp->zpool_hdl->libzfs_fd, ZFS_IOC_POOL_STATS,
285                   &zc) == 0) {
286             /*
287             * The real error is returned in the zc_cookie field.
288             */
289             error = zc.zc_cookie;
290             break;
291         }
292
293         if (errno == ENOMEM) {
294             if (zcmd_expand_dst_nvlist(hdl, &zc) != 0) {
295                 zcmd_free_nvlists(&zc);
296                 return (-1);
297             }
298         } else {
299             zcmd_free_nvlists(&zc);
300             if (errno == ENOENT || errno == EINVAL)
301                 *missing = B_TRUE;
302             zhp->zpool_state = POOL_STATE_UNAVAIL;
303             return (0);
304         }
305     }
306
307     if (zcmd_read_dst_nvlist(hdl, &zc, &config) != 0) {
308         zcmd_free_nvlists(&zc);
309         return (-1);
310     }
311
312     zcmd_free_nvlists(&zc);
313
314     zhp->zpool_config_size = zc.zc_nvlist_dst_size;
315
316     if (zhp->zpool_config != NULL) {
317         uint64_t oldtxg, newtxg;
318
319         verify(nvlist_lookup_uint64(zhp->zpool_config,
320                                     ZPOOL_CONFIG_POOL_TXG, &oldtxg) == 0);
321         verify(nvlist_lookup_uint64(config,
322                                     ZPOOL_CONFIG_POOL_TXG, &newtxg) == 0);
323

```

```

325         if (zhp->zpool_old_config != NULL)
326             nvlist_free(zhp->zpool_old_config);
327
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     }
335
336     zhp->zpool_config = config;
337     if (error)
338         zhp->zpool_state = POOL_STATE_UNAVAIL;
339     else
340         zhp->zpool_state = POOL_STATE_ACTIVE;
341
342     return (0);
343 }
344 /*
345 * If the __ZFS_POOL_RESTRICT environment variable is set we only iterate over
346 * pools it lists.
347 *
348 * This is an undocumented feature for use during testing only.
349 *
350 * This function returns B_TRUE if the pool should be skipped
351 * during iteration.
352 */
353 static boolean_t
354 check_restricted(const char *poolname)
355 {
356     static boolean_t initialized = B_FALSE;
357     static char *restricted = NULL;
358
359     const char *cur, *end;
360     int len, namelen;
361
362     if (!initialized) {
363         initialized = B_TRUE;
364         restricted = getenv("__ZFS_POOL_RESTRICT");
365     }
366
367     if (NULL == restricted)
368         return (B_FALSE);
369
370     cur = restricted;
371     namelen = strlen(poolname);
372     do {
373         end = strchr(cur, ' ');
374         len = (NULL == end) ? strlen(cur) : (end - cur);
375
376         if (len == namelen && 0 == strncmp(cur, poolname, len)) {
377             return (B_FALSE);
378         }
379     }
380     cur += (len + 1);
381     } while (NULL != end);
382
383     return (B_TRUE);
384 }
385
386 /*
387 * Iterate over all pools in the system.
388 */
389 int
390 int

```

```

391 zpool_iter(libzfs_handle_t *hdl, zpool_iter_f func, void *data)
392 {
393     config_node_t *cn;
394     zpool_handle_t *zhp;
395     int ret;
396
397     /*
398      * If someone makes a recursive call to zpool_iter(), we want to avoid
399      * refreshing the namespace because that will invalidate the parent
400      * context. We allow recursive calls, but simply re-use the same
401      * namespace AVL tree.
402      */
403     if (!hdl->libzfs_pool_iter && namespace_reload(hdl) != 0)
404         return (-1);
405
406     hdl->libzfs_pool_iter++;
407     for (cn = uu_avl_first(hdl->libzfs_ns_avl); cn != NULL;
408          cn = uu_avl_next(hdl->libzfs_ns_avl, cn)) {
409
410         if (check_restricted(cn->cn_name))
411             continue;
412
413         if (zpool_open_silent(hdl, cn->cn_name, &zhp) != 0) {
414             hdl->libzfs_pool_iter--;
415             return (-1);
416         }
417
418         if (zhp == NULL)
419             continue;
420
421         if ((ret = func(zhp, data)) != 0) {
422             hdl->libzfs_pool_iter--;
423             return (ret);
424         }
425     }
426     hdl->libzfs_pool_iter--;
427
428     return (0);
429 }
430
431 /*
432  * Iterate over root datasets, calling the given function for each. The zfs
433  * handle passed each time must be explicitly closed by the callback.
434  */
435 int
436 zfs_iter_root(libzfs_handle_t *hdl, zfs_iter_f func, void *data)
437 {
438     config_node_t *cn;
439     zfs_handle_t *zhp;
440     int ret;
441
442     if (namespace_reload(hdl) != 0)
443         return (-1);
444
445     for (cn = uu_avl_first(hdl->libzfs_ns_avl); cn != NULL;
446          cn = uu_avl_next(hdl->libzfs_ns_avl, cn)) {
447
448         if (check_restricted(cn->cn_name))
449             continue;
450
451         if ((zhp = make_dataset_handle(hdl, cn->cn_name)) == NULL)
452             continue;
453
454         if ((ret = func(zhp, data)) != 0)
455             return (ret);
456     }

```

```

458         return (0);
459     }

```