

```

*****
10274 Fri Sep 11 13:41:17 2015
new/usr/src/lib/libuutil/common/libuutil.h
6222 libuutil could provide a way to re-create an AVL tree
*****
_____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_takedown(uu_avl_t *, void **);

387 void uu_avl_remove(uu_avl_t *, void *);

389 #ifdef __cplusplus
390 }
391 #endif

393 #endif /* _LIBUUTIL_H */

```

```

*****
2564 Fri Sep 11 13:41:17 2015
new/usr/src/lib/libutil/common/mapfile-vers
6222 libutil could provide a way to re-create an AVL tree
*****
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;

```

```

62         uu_avl_takedown;
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_takedown;
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_strcaseeq;
109        uu_strdup;
110        uu_streq;
111        uu_strerror;
112        uu_strndup;
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 };

```

```

*****
14180 Fri Sep 11 13:41:17 2015
new/usr/src/lib/libuutil/common/uu_avl.c
6222 libuutil could provide a way to re-create an AVL tree
*****
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) + \

```

```

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     }

80     if (flags & ~UU_AVL_POOL_DEBUG) {
81         uu_set_error(UU_ERROR_UNKNOWN_FLAG);
82         return (NULL);
83     }

85     pp = uu_zalloc(sizeof (uu_avl_pool_t));
86     if (pp == NULL) {
87         uu_set_error(UU_ERROR_NO_MEMORY);
88         return (NULL);
89     }

91     (void) strncpy(pp->uap_name, name, sizeof (pp->uap_name));
92     pp->uap_nodeoffset = nodeoffset;
93     pp->uap_objsize = objsize;
94     pp->uap_cmp = compare_func;
95     if (flags & UU_AVL_POOL_DEBUG)
96         pp->uap_debug = 1;
97     pp->uap_last_index = 0;

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

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

104     (void) pthread_mutex_lock(&uu_apool_list_lock);
105     pp->uap_next = next = &uu_null_apool;
106     pp->uap_prev = prev = next->uap_prev;
107     next->uap_prev = pp;
108     prev->uap_next = pp;
109     (void) pthread_mutex_unlock(&uu_apool_list_lock);

111     return (pp);
112 }

unchanged portion omitted

249 void
250 uu_avl_destroy(uu_avl_t *ap)
251 {
252     uu_avl_pool_t *pp = ap->ua_pool;

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         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);
271
272     ap->ua_pool = NULL;
273     avl_destroy(&ap->ua_tree);
274
275     uu_free(ap);
276 }
277
278 void
279 uu_avl_recreate(uu_avl_t *ap)
280 {
281     uu_avl_pool_t *pp = ap->ua_pool;
282
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 }
288
289 size_t
290 uu_avl_numnodes(uu_avl_t *ap)
291 {
292     return (avl_numnodes(&ap->ua_tree));
293 }
294
295 void *
296 uu_avl_first(uu_avl_t *ap)
297 {
298     return (avl_first(&ap->ua_tree));
299 }
300
301 void *
302 uu_avl_last(uu_avl_t *ap)
303 {
304     return (avl_last(&ap->ua_tree));
305 }
306
307 void *
308 uu_avl_next(uu_avl_t *ap, void *node)
309 {
310     return (AVL_NEXT(&ap->ua_tree, node));
311 }
312
313 void *
314 uu_avl_prev(uu_avl_t *ap, void *node)
315 {
316     return (AVL_PREV(&ap->ua_tree, node));
317 }
318
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;
323
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;
331
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);
336
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 }
344
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;
351
352     if (np == NULL)
353         return (NULL);
354
355     wp->uaw_next_result = (wp->uaw_dir > 0)? AVL_NEXT(t, np) :
356         AVL_PREV(t, np);
357
358     return (np);
359 }
360
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 }
373
374 uu_avl_walk_t *
375 uu_avl_walk_start(uu_avl_t *ap, uint32_t flags)
376 {
377     uu_avl_walk_t *wp;
378
379     if (flags & ~(UU_WALK_ROBUST | UU_WALK_REVERSE)) {
380         uu_set_error(UU_ERROR_UNKNOWN_FLAG);
381         return (NULL);
382     }
383
384     wp = uu_zalloc(sizeof (*wp));
385     if (wp == NULL) {
386         uu_set_error(UU_ERROR_NO_MEMORY);
387         return (NULL);
388     }
389
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 must 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_takedown(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 }
```