```
**********************************************************
   28038 Sat Aug  1 08:08:32 2015
new/usr/src/common/avl/avl.c
6091 avl_add doesn't assert on non-debug builds
Reviewed by: Andy Stormont <astormont@racktopsystems.com>
**********************************************************
/*
 * 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) 2014 by Delphix. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

/*
 * AVL - generic AVL tree implementation for kernel use
 *
 * A complete description of AVL trees can be found in many CS textbooks.
 *
 * Here is a very brief overview. An AVL tree is a binary search tree that is
 * almost perfectly balanced. By "almost" perfectly balanced, we mean that at
 * any given node, the left and right subtrees are allowed to differ in height
 * by at most 1 level.
 *
 * This relaxation from a perfectly balanced binary tree allows doing
 * insertion and deletion relatively efficiently. Searching the tree is
 * still a fast operation, roughly O(log(N)).
 *
 * The key to insertion and deletion is a set of tree manipulations called
 * rotations, which bring unbalanced subtrees back into the semi-balanced state.
 *
 * This implementation of AVL trees has the following peculiarities:
 *
 *      - The AVL specific data structures are physically embedded as fields
 *        in the "using" data structures.  To maintain generality the code
 *        must constantly translate between "avl_node_t *" and containing
 *        data structure "void *"s by adding/subtracting the avl_offset.
 *
 *      - Since the AVL data is always embedded in other structures, there is
 *        no locking or memory allocation in the AVL routines. This must be
 *        provided for by the enclosing data structure's semantics. Typically,
 *        avl_insert()/_add()/_remove()/avl_insert_here() require some kind of
 *        exclusive write lock. Other operations require a read lock.
```



```
 *
 *      - The implementation uses iteration instead of explicit recursion,
 *        since it is intended to run on limited size kernel stacks. Since
 *        there is no recursion stack present to move "up" in the tree,
 *        there is an explicit "parent" link in the avl_node_t.
 *
 *      - The left/right children pointers of a node are in an array.
 *        In the code, variables (instead of constants) are used to represent
 *        left and right indices.  The implementation is written as if it only
 *        dealt with left handed manipulations.  By changing the value assigned
 *        to "left", the code also works for right handed trees.  The
 *        following variables/terms are frequently used:
 *
 *              int left;       // 0 when dealing with left children,
 *                              // 1 for dealing with right children
 *
 *              int left_heavy; // -1 when left subtree is taller at some node,
 *                              // +1 when right subtree is taller
 *
 *              int right;      // will be the opposite of left (0 or 1)
 *              int right_heavy;// will be the opposite of left_heavy (-1 or 1)
 *
 *              int direction;  // 0 for "<" (ie. left child); 1 for ">" (right)
 *
 *        Though it is a little more confusing to read the code, the approach
 *        allows using half as much code (and hence cache footprint) for tree
 *        manipulations and eliminates many conditional branches.
 *
 *      - The avl_index_t is an opaque "cookie" used to find nodes at or
 *        adjacent to where a new value would be inserted in the tree. The value
 *        is a modified "avl_node_t *".  The bottom bit (normally 0 for a
 *        pointer) is set to indicate if that the new node has a value greater
 *        than the value of the indicated "avl_node_t *".
 *
 * Note - in addition to userland (e.g. libavl and libutil) and the kernel
 * (e.g. genunix), avl.c is compiled into ld.so and kmdb's genunix module,
 * which each have their own compilation environments and subsequent
 * requirements. Each of these environments must be considered when adding
 * dependencies from avl.c.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/debug.h>
#include <sys/avl.h>
#include <sys/cmn_err.h>

/*
 * Small arrays to translate between balance (or diff) values and child indices.
 *
 * Code that deals with binary tree data structures will randomly use
 * left and right children when examining a tree.  C "if()" statements
 * which evaluate randomly suffer from very poor hardware branch prediction.
 * In this code we avoid some of the branch mispredictions by using the
 * following translation arrays. They replace random branches with an
 * additional memory reference. Since the translation arrays are both very
 * small the data should remain efficiently in cache.
 */
static const int  avl_child2balance[2]  = {-1, 1};
static const int  avl_balance2child[]   = {0, 0, 1};


/*
 * Walk from one node to the previous valued node (ie. an infix walk
 * towards the left). At any given node we do one of 2 things:
 *
```

```
 * - If there is a left child, go to it, then to it's rightmost descendant.
 *
 * - otherwise we return through parent nodes until we've come from a right
 *   child.
 *
 * Return Value:
 * NULL - if at the end of the nodes
 * otherwise next node
 */
void *
avl_walk(avl_tree_t *tree, void *oldnode, int left)
{
        size_t off = tree->avl_offset;
        avl_node_t *node = AVL_DATA2NODE(oldnode, off);
        int right = 1 - left;
        int was_child;


        /*
         * nowhere to walk to if tree is empty
         */
        if (node == NULL)
                return (NULL);

        /*
         * Visit the previous valued node. There are two possibilities:
         *
         * If this node has a left child, go down one left, then all
         * the way right.
         */
        if (node->avl_child[left] != NULL) {
                for (node = node->avl_child[left];
                    node->avl_child[right] != NULL;
                    node = node->avl_child[right])
                        ;
        /*
         * Otherwise, return thru left children as far as we can.
         */
        } else {
                for (;;) {
                        was_child = AVL_XCHILD(node);
                        node = AVL_XPARENT(node);
                        if (node == NULL)
                                return (NULL);
                        if (was_child == right)
                                break;
                }
        }

        return (AVL_NODE2DATA(node, off));
}

/*
 * Return the lowest valued node in a tree or NULL.
 * (leftmost child from root of tree)
 */
void *
avl_first(avl_tree_t *tree)
{
        avl_node_t *node;
        avl_node_t *prev = NULL;
        size_t off = tree->avl_offset;

        for (node = tree->avl_root; node != NULL; node = node->avl_child[0])
                prev = node;
```



```
        if (prev != NULL)
                return (AVL_NODE2DATA(prev, off));
        return (NULL);
}

/*
 * Return the highest valued node in a tree or NULL.
 * (rightmost child from root of tree)
 */
void *
avl_last(avl_tree_t *tree)
{
        avl_node_t *node;
        avl_node_t *prev = NULL;
        size_t off = tree->avl_offset;

        for (node = tree->avl_root; node != NULL; node = node->avl_child[1])
                prev = node;

        if (prev != NULL)
                return (AVL_NODE2DATA(prev, off));
        return (NULL);
}

/*
 * Access the node immediately before or after an insertion point.
 *
 * "avl_index_t" is a (avl_node_t *) with the bottom bit indicating a child
 *
 * Return value:
 *      NULL: no node in the given direction
 *      "void *"  of the found tree node
 */
void *
avl_nearest(avl_tree_t *tree, avl_index_t where, int direction)
{
        int child = AVL_INDEX2CHILD(where);
        avl_node_t *node = AVL_INDEX2NODE(where);
        void *data;
        size_t off = tree->avl_offset;

        if (node == NULL) {
                ASSERT(tree->avl_root == NULL);
                return (NULL);
        }
        data = AVL_NODE2DATA(node, off);
        if (child != direction)
                return (data);

        return (avl_walk(tree, data, direction));
}


/*
 * Search for the node which contains "value".  The algorithm is a
 * simple binary tree search.
 *
 * return value:
 *      NULL: the value is not in the AVL tree
 *              *where (if not NULL)  is set to indicate the insertion point
 *      "void *"  of the found tree node
 */
void *
avl_find(avl_tree_t *tree, const void *value, avl_index_t *where)
{
        avl_node_t *node;
```

```
        avl_node_t *prev = NULL;
        int child = 0;
        int diff;
        size_t off = tree->avl_offset;

        for (node = tree->avl_root; node != NULL;
            node = node->avl_child[child]) {

                prev = node;

                diff = tree->avl_compar(value, AVL_NODE2DATA(node, off));
                ASSERT(-1 <= diff && diff <= 1);
                if (diff == 0) {
#ifdef DEBUG
                        if (where != NULL)
                                *where = 0;
#endif
                        return (AVL_NODE2DATA(node, off));
                }
                child = avl_balance2child[1 + diff];

        }

        if (where != NULL)
                *where = AVL_MKINDEX(prev, child);

        return (NULL);
}


/*
 * Perform a rotation to restore balance at the subtree given by depth.
 *
 * This routine is used by both insertion and deletion. The return value
 * indicates:
 *       0 : subtree did not change height
 *      !0 : subtree was reduced in height
 *
 * The code is written as if handling left rotations, right rotations are
 * symmetric and handled by swapping values of variables right/left[_heavy]
 *
 * On input balance is the "new" balance at "node". This value is either
 * -2 or +2.
 */
static int
avl_rotation(avl_tree_t *tree, avl_node_t *node, int balance)
{
        int left = !(balance < 0);      /* when balance = -2, left will be 0 */
        int right = 1 - left;
        int left_heavy = balance >> 1;
        int right_heavy = -left_heavy;
        avl_node_t *parent = AVL_XPARENT(node);
        avl_node_t *child = node->avl_child[left];
        avl_node_t *cright;
        avl_node_t *gchild;
        avl_node_t *gright;
        avl_node_t *gleft;
        int which_child = AVL_XCHILD(node);
        int child_bal = AVL_XBALANCE(child);

        /* BEGIN CSTYLED */
        /*
         * case 1 : node is overly left heavy, the left child is balanced or
         * also left heavy. This requires the following rotation.
         *
         *                   (node bal:-2)
```



```
         *                    /           \
         *                   /             \
         *              (child bal:0 or -1)
         *              /    \
         *             /      \
         *                     cright
         *
         * becomes:
         *
         *              (child bal:1 or 0)
         *              /        \
         *             /          \
         *                         (node bal:-1 or 0)
         *                          /     \
         *                         /       \
         *                     cright
         *
         * we detect this situation by noting that child's balance is not
         * right_heavy.
         */
        /* END CSTYLED */
        if (child_bal != right_heavy) {

                /*
                 * compute new balance of nodes
                 *
                 * If child used to be left heavy (now balanced) we reduced
                 * the height of this sub-tree -- used in "return...;" below
                 */
                child_bal += right_heavy; /* adjust towards right */

                /*
                 * move "cright" to be node's left child
                 */
                cright = child->avl_child[right];
                node->avl_child[left] = cright;
                if (cright != NULL) {
                        AVL_SETPARENT(cright, node);
                        AVL_SETCHILD(cright, left);
                }

                /*
                 * move node to be child's right child
                 */
                child->avl_child[right] = node;
                AVL_SETBALANCE(node, -child_bal);
                AVL_SETCHILD(node, right);
                AVL_SETPARENT(node, child);

                /*
                 * update the pointer into this subtree
                 */
                AVL_SETBALANCE(child, child_bal);
                AVL_SETCHILD(child, which_child);
                AVL_SETPARENT(child, parent);
                if (parent != NULL)
                        parent->avl_child[which_child] = child;
                else
                        tree->avl_root = child;

                return (child_bal == 0);
        }

        /* BEGIN CSTYLED */
        /*
         * case 2 : When node is left heavy, but child is right heavy we use
```

```c
         * a different rotation.
         *
         *                   (node b:-2)
         *                    /   \
         *                   /     \
         *                  /       \
         *             (child b:+1)
         *              /     \
         *             /       \
         *                   (gchild b: != 0)
         *                     / \
         *                    /   \
         *                 gleft   gright
         *
         * becomes:
         *
         *              (gchild b:0)
         *              /       \
         *             /         \
         *            /           \
         *        (child b:?)   (node b:?)
         *         /  \          /   \
         *        /    \        /     \
         *            gleft   gright
         *
         * computing the new balances is more complicated. As an example:
         *       if gchild was right_heavy, then child is now left heavy
         *              else it is balanced
         */
        /* END CSTYLED */
        gchild = child->avl_child[right];
        gleft = gchild->avl_child[left];
        gright = gchild->avl_child[right];

        /*
         * move gright to left child of node and
         *
         * move gleft to right child of node
         */
        node->avl_child[left] = gright;
        if (gright != NULL) {
                AVL_SETPARENT(gright, node);
                AVL_SETCHILD(gright, left);
        }

        child->avl_child[right] = gleft;
        if (gleft != NULL) {
                AVL_SETPARENT(gleft, child);
                AVL_SETCHILD(gleft, right);
        }

        /*
         * move child to left child of gchild and
         *
         * move node to right child of gchild and
         *
         * fixup parent of all this to point to gchild
         */
        balance = AVL_XBALANCE(gchild);
        gchild->avl_child[left] = child;
        AVL_SETBALANCE(child, (balance == right_heavy ? left_heavy : 0));
        AVL_SETPARENT(child, gchild);
        AVL_SETCHILD(child, left);

        gchild->avl_child[right] = node;
        AVL_SETBALANCE(node, (balance == left_heavy ? right_heavy : 0));
```



```c
        AVL_SETPARENT(node, gchild);
        AVL_SETCHILD(node, right);

        AVL_SETBALANCE(gchild, 0);
        AVL_SETPARENT(gchild, parent);
        AVL_SETCHILD(gchild, which_child);
        if (parent != NULL)
                parent->avl_child[which_child] = gchild;
        else
                tree->avl_root = gchild;

        return (1);     /* the new tree is always shorter */
}


/*
 * Insert a new node into an AVL tree at the specified (from avl_find()) place.
 *
 * Newly inserted nodes are always leaf nodes in the tree, since avl_find()
 * searches out to the leaf positions.  The avl_index_t indicates the node
 * which will be the parent of the new node.
 *
 * After the node is inserted, a single rotation further up the tree may
 * be necessary to maintain an acceptable AVL balance.
 */
void
avl_insert(avl_tree_t *tree, void *new_data, avl_index_t where)
{
        avl_node_t *node;
        avl_node_t *parent = AVL_INDEX2NODE(where);
        int old_balance;
        int new_balance;
        int which_child = AVL_INDEX2CHILD(where);
        size_t off = tree->avl_offset;

        ASSERT(tree);
#ifdef _LP64
        ASSERT(((uintptr_t)new_data & 0x7) == 0);
#endif

        node = AVL_DATA2NODE(new_data, off);

        /*
         * First, add the node to the tree at the indicated position.
         */
        ++tree->avl_numnodes;

        node->avl_child[0] = NULL;
        node->avl_child[1] = NULL;

        AVL_SETCHILD(node, which_child);
        AVL_SETBALANCE(node, 0);
        AVL_SETPARENT(node, parent);
        if (parent != NULL) {
                ASSERT(parent->avl_child[which_child] == NULL);
                parent->avl_child[which_child] = node;
        } else {
                ASSERT(tree->avl_root == NULL);
                tree->avl_root = node;
        }
        /*
         * Now, back up the tree modifying the balance of all nodes above the
         * insertion point. If we get to a highly unbalanced ancestor, we
         * need to do a rotation.  If we back out of the tree we are done.
         * If we brought any subtree into perfect balance (0), we are also done.
         */
```

```
        for (;;) {
                node = parent;
                if (node == NULL)
                        return;

                /*
                 * Compute the new balance
                 */
                old_balance = AVL_XBALANCE(node);
                new_balance = old_balance + avl_child2balance[which_child];

                /*
                 * If we introduced equal balance, then we are done immediately
                 */
                if (new_balance == 0) {
                        AVL_SETBALANCE(node, 0);
                        return;
                }

                /*
                 * If both old and new are not zero we went
                 * from -1 to -2 balance, do a rotation.
                 */
                if (old_balance != 0)
                        break;

                AVL_SETBALANCE(node, new_balance);
                parent = AVL_XPARENT(node);
                which_child = AVL_XCHILD(node);
        }

        /*
         * perform a rotation to fix the tree and return
         */
        (void) avl_rotation(tree, node, new_balance);
}

/*
 * Insert "new_data" in "tree" in the given "direction" either after or
 * before (AVL_AFTER, AVL_BEFORE) the data "here".
 *
 * Insertions can only be done at empty leaf points in the tree, therefore
 * if the given child of the node is already present we move to either
 * the AVL_PREV or AVL_NEXT and reverse the insertion direction. Since
 * every other node in the tree is a leaf, this always works.
 *
 * To help developers using this interface, we assert that the new node
 * is correctly ordered at every step of the way in DEBUG kernels.
 */
void
avl_insert_here(
        avl_tree_t *tree,
        void *new_data,
        void *here,
        int direction)
{
        avl_node_t *node;
        int child = direction;  /* rely on AVL_BEFORE == 0, AVL_AFTER == 1 */
#ifdef DEBUG
        int diff;
#endif

        ASSERT(tree != NULL);
        ASSERT(new_data != NULL);
        ASSERT(here != NULL);
        ASSERT(direction == AVL_BEFORE || direction == AVL_AFTER);
```



```
        /*
         * If corresponding child of node is not NULL, go to the neighboring
         * node and reverse the insertion direction.
         */
        node = AVL_DATA2NODE(here, tree->avl_offset);

#ifdef DEBUG
        diff = tree->avl_compar(new_data, here);
        ASSERT(-1 <= diff && diff <= 1);
        ASSERT(diff != 0);
        ASSERT(diff > 0 ? child == 1 : child == 0);
#endif

        if (node->avl_child[child] != NULL) {
                node = node->avl_child[child];
                child = 1 - child;
                while (node->avl_child[child] != NULL) {
#ifdef DEBUG
                        diff = tree->avl_compar(new_data,
                            AVL_NODE2DATA(node, tree->avl_offset));
                        ASSERT(-1 <= diff && diff <= 1);
                        ASSERT(diff != 0);
                        ASSERT(diff > 0 ? child == 1 : child == 0);
#endif
                        node = node->avl_child[child];
                }
#ifdef DEBUG
                diff = tree->avl_compar(new_data,
                    AVL_NODE2DATA(node, tree->avl_offset));
                ASSERT(-1 <= diff && diff <= 1);
                ASSERT(diff != 0);
                ASSERT(diff > 0 ? child == 1 : child == 0);
#endif
        }
        ASSERT(node->avl_child[child] == NULL);

        avl_insert(tree, new_data, AVL_MKINDEX(node, child));
}

/*
 * Add a new node to an AVL tree.
 */
void
avl_add(avl_tree_t *tree, void *new_node)
{
        avl_index_t where;

        /*
         * This is unfortunate.  We want to call panic() here, even for
         * non-DEBUG kernels.  In userland, however, we can't depend on anything
         * in libc or else the rtld build process gets confused.
         * Thankfully, rtld provides us with its own assfail() so we can use
         * that here.  We use assfail() directly to get a nice error message
         * in the core - much like what panic() does for crashdumps.
         * in libc or else the rtld build process gets confused.  So, all we can
         * do in userland is resort to a normal ASSERT().
         */
        if (avl_find(tree, new_node, &where) != NULL)
#ifdef _KERNEL
                panic("avl_find() succeeded inside avl_add()");
#else
                (void) assfail("avl_find() succeeded inside avl_add()",
                    __FILE__, __LINE__);
                ASSERT(0);
#endif
```

```
        avl_insert(tree, new_node, where);
}
```

______*unchanged_portion_omitted_*

****
1341 Sat Aug 1 08:08:32 2015
new/usr/src/lib/libavl/Makefile.com
6091 avl_add doesn’t assert on non-debug builds
Reviewed by: Andy Stormont <astormont@racktopsystems.com>
****

```
#
# 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.
#
# ident "%Z%%M% %I%     %E% SMI"
#

LIBRARY=        libavl.a
VERS=           .1
OBJECTS=        avl.o

include ../../Makefile.lib
include ../../Makefile.rootfs

LIBS =          $(DYNLIB) $(LINTLIB)
SRCS =          $(COMDIR)/avl.c

$(LINTLIB) :=   SRCS = $(SRCDIR)/$(LINTSRC)

COMDIR =        $(SRC)/common/avl

LDLIBS +=       -lc
#endif /* ! codereview */
CFLAGS +=       $(CCVERBOSE)

.KEEP_STATE:

all : $(LIBS)

lint : lintcheck

pics/%.o:        $(COMDIR)/%.c
        $(COMPILE.c) -o $@ $<
        $(POST_PROCESS_O)

include ../../Makefile.targ
```