Red-black Trees (rbtree) in Linux

Date: January 18, 2007
Author: Rob Landley <rob@landley.net>

What are red-black trees, and what are they for?

Red-black trees are a type of self-balancing binary search tree, used for storing sortable key/value data pairs. This differs from radix trees (which are used to efficiently store sparse arrays and thus use long integer indexes to insert/access/delete nodes) and hash tables (which are not kept sorted to be easily traversed in order, and must be tuned for a specific size and hash function where rbtrees scale gracefully storing arbitrary keys).

Red-black trees are similar to AVL trees, but provide faster real-time bounded worst case performance for insertion and deletion (at most two rotations and three rotations, respectively, to balance the tree), with slightly slower (but still O(log n)) lookup time.

To quote Linux Weekly News:

> There are a number of red-black trees in use in the kernel. The deadline and CFQ I/O schedulers employ rbtrees to track requests; the packet CD/DVD driver does the same. The high-resolution timer code uses an rbtree to organize outstanding timer requests. The ext3 filesystem tracks directory entries in a red-black tree. Virtual memory areas (VMAs) are tracked with red-black trees, as are epoll file descriptors, cryptographic keys, and network packets in the "hierarchical token bucket" scheduler.

This document covers use of the Linux rbtree implementation. For more information on the nature and implementation of Red Black Trees, see:

Linux Weekly News article on red-black trees
https://lwn.net/Articles/184495/
Wikipedia entry on red-black trees
https://en.wikipedia.org/wiki/Red-black_tree

Linux implementation of red-black trees

Linux's rbtree implementation lives in the file "lib/rbtree.c". To use it, "#include <linux/rbtree.h>".

The Linux rbtree implementation is optimized for speed, and thus has one less layer of indirection (and better cache locality) than more traditional tree implementations. Instead of using pointers to separate rb_node and data structures, each instance of struct rb_node is embedded in the data structure it organizes. And instead of using a comparison callback function pointer, users are expected to write their own tree search and insert functions which call the provided rbtree functions. Locking is also left up to the user of the rbtree code.

Creating a new rbtree

Data nodes in an rbtree tree are structures containing a struct rb_node member:

```
struct mytype {
      struct rb_node node;
      char *keystring;
};
```

When dealing with a pointer to the embedded struct rb_node, the containing data structure may be accessed with the standard container_of() macro. In addition, individual members may be accessed directly via rb_entry(node, type, member).

At the root of each rbtree is an rb_root structure, which is initialized to be empty via:

```
struct rb_root mytree = RB_ROOT;
```

Searching for a value in an rbtree

Writing a search function for your tree is fairly straightforward: start at the root, compare each value, and follow the left or right branch as necessary.

Example:

```
struct mytype *my_search(struct rb_root *root, char *string)
{
      struct rb_node *node = root->rb_node;

      while (node) {
```

```
                struct mytype *data = container_of(node, struct mytype, node);
                int result;

                result = strcmp(string, data->keystring);

                if (result < 0)
                        node = node->rb_left;
                else if (result > 0)
                        node = node->rb_right;
                else
                        return data;
        }
        return NULL;
}
```

Inserting data into an rbtree

Inserting data in the tree involves first searching for the place to insert the new node, then inserting the node and rebalancing ("recoloring") the tree.

The search for insertion differs from the previous search by finding the location of the pointer on which to graft the new node. The new node also needs a link to its parent node for rebalancing purposes.

Example:

```
int my_insert(struct rb_root *root, struct mytype *data)
{
        struct rb_node **new = &(root->rb_node), *parent = NULL;

        /* Figure out where to put new node */
        while (*new) {
                struct mytype *this = container_of(*new, struct mytype, node);
                int result = strcmp(data->keystring, this->keystring);

                parent = *new;
                if (result < 0)
                        new = &((*new)->rb_left);
                else if (result > 0)
                        new = &((*new)->rb_right);
                else
                        return FALSE;
        }

        /* Add new node and rebalance tree. */
        rb_link_node(&data->node, parent, new);
        rb_insert_color(&data->node, root);

        return TRUE;
}
```

Removing or replacing existing data in an rbtree

To remove an existing node from a tree, call:

```
void rb_erase(struct rb_node *victim, struct rb_root *tree);
```

Example:

```
struct mytype *data = mysearch(&mytree, "walrus");

if (data) {
        rb_erase(&data->node, &mytree);
        myfree(data);
}
```

To replace an existing node in a tree with a new one with the same key, call:

```
void rb_replace_node(struct rb_node *old, struct rb_node *new,
                     struct rb_root *tree);
```

Replacing a node this way does not re-sort the tree: If the new node doesn't have the same key as the old node, the rbtree will probably become corrupted.

Iterating through the elements stored in an rbtree (in sort order)

Four functions are provided for iterating through an rbtree's contents in sorted order. These work on arbitrary trees, and should not need to be modified or wrapped (except for locking purposes):

```
struct rb_node *rb_first(struct rb_root *tree);
struct rb_node *rb_last(struct rb_root *tree);
struct rb_node *rb_next(struct rb_node *node);
struct rb_node *rb_prev(struct rb_node *node);
```

To start iterating, call rb_first() or rb_last() with a pointer to the root of the tree, which will return a pointer to the node structure contained in the first or last element in the tree. To continue, fetch the next or previous node by calling rb_next() or rb_prev() on the current node. This will return NULL when there are no more nodes left.

The iterator functions return a pointer to the embedded struct rb_node, from which the containing data structure may be accessed with the container_of() macro, and individual members may be accessed directly via rb_entry(node, type, member).

Example:

```
struct rb_node *node;
for (node = rb_first(&mytree); node; node = rb_next(node))
      printk("key=%s\n", rb_entry(node, struct mytype, node)->keystring);
```

Cached rbtrees

Computing the leftmost (smallest) node is quite a common task for binary search trees, such as for traversals or users relying on a the particular order for their own logic. To this end, users can use 'struct rb_root_cached' to optimize O(logN) rb_first() calls to a simple pointer fetch avoiding potentially expensive tree iterations. This is done at negligible runtime overhead for maintenance; albeit larger memory footprint.

Similar to the rb_root structure, cached rbtrees are initialized to be empty via:

```
struct rb_root_cached mytree = RB_ROOT_CACHED;
```

Cached rbtree is simply a regular rb_root with an extra pointer to cache the leftmost node. This allows rb_root_cached to exist wherever rb_root does, which permits augmented trees to be supported as well as only a few extra interfaces:

```
struct rb_node *rb_first_cached(struct rb_root_cached *tree);
void rb_insert_color_cached(struct rb_node *, struct rb_root_cached *, bool);
void rb_erase_cached(struct rb_node *node, struct rb_root_cached *);
```

Both insert and erase calls have their respective counterpart of augmented trees:

```
void rb_insert_augmented_cached(struct rb_node *node, struct rb_root_cached *,
                                bool, struct rb_augment_callbacks *);
void rb_erase_augmented_cached(struct rb_node *, struct rb_root_cached *,
                               struct rb_augment_callbacks *);
```

Support for Augmented rbtrees

Augmented rbtree is an rbtree with "some" additional data stored in each node, where the additional data for node N must be a function of the contents of all nodes in the subtree rooted at N. This data can be used to augment some new functionality to rbtree. Augmented rbtree is an optional feature built on top of basic rbtree infrastructure. An rbtree user who wants this feature will have to call the augmentation functions with the user provided augmentation callback when inserting and erasing nodes.

C files implementing augmented rbtree manipulation must include <linux/rbtree_augmented.h> instead of <linux/rbtree.h>. Note that linux/rbtree_augmented.h exposes some rbtree implementations details you are not expected to rely on; please stick to the documented APIs there and do not include <linux/rbtree_augmented.h> from header files either so as to minimize chances of your users accidentally relying on such implementation details.

On insertion, the user must update the augmented information on the path leading to the inserted node, then call rb_link_node() as usual and rb_augment_inserted() instead of the usual rb_insert_color() call. If rb_augment_inserted() rebalances the rbtree, it will callback into a user provided function to update the augmented information on the affected subtrees.

When erasing a node, the user must call rb_erase_augmented() instead of rb_erase(). rb_erase_augmented() calls back into user provided functions to updated the augmented information on affected subtrees.

In both cases, the callbacks are provided through struct rb_augment_callbacks. 3 callbacks must be defined:

- A propagation callback, which updates the augmented value for a given node and its ancestors, up to a given stop point (or NULL to update all the way to the root).
- A copy callback, which copies the augmented value for a given subtree to a newly assigned subtree root.
- A tree rotation callback, which copies the augmented value for a given subtree to a newly assigned subtree root AND recomputes the augmented information for the former subtree root.

The compiled code for rb_erase_augmented() may inline the propagation and copy callbacks, which results in a large function, so each augmented rbtree user should have a single rb_erase_augmented() call site in order to limit compiled code size.

Sample usage

Interval tree is an example of augmented rb tree. Reference - "Introduction to Algorithms" by Cormen, Leiserson, Rivest and Stein.

More details about interval trees:

Classical rbtree has a single key and it cannot be directly used to store interval ranges like [lo:hi] and do a quick lookup for any overlap with a new lo:hi or to find whether there is an exact match for a new lo:hi.

However, rbtree can be augmented to store such interval ranges in a structured way making it possible to do efficient lookup and exact match.

This "extra information" stored in each node is the maximum hi (max_hi) value among all the nodes that are its descendants. This information can be maintained at each node just be looking at the node and its immediate children. And this will be used in O(log n) lookup for lowest match (lowest start address among all possible matches) with something like:

```
struct interval_tree_node *
interval_tree_first_match(struct rb_root *root,
                          unsigned long start, unsigned long last)
{
      struct interval_tree_node *node;

      if (!root->rb_node)
              return NULL;
      node = rb_entry(root->rb_node, struct interval_tree_node, rb);

      while (true) {
              if (node->rb.rb_left) {
                      struct interval_tree_node *left =
                              rb_entry(node->rb.rb_left,
                                       struct interval_tree_node, rb);
                      if (left->__subtree_last >= start) {
                              /*
                               * Some nodes in left subtree satisfy Cond2.
                               * Iterate to find the leftmost such node N.
                               * If it also satisfies Cond1, that's the match
                               * we are looking for. Otherwise, there is no
                               * matching interval as nodes to the right of N
                               * can't satisfy Cond1 either.
                               */
                              node = left;
                              continue;
                      }
              }
              if (node->start <= last) {              /* Cond1 */
                      if (node->last >= start)        /* Cond2 */
                              return node;    /* node is leftmost match */
                      if (node->rb.rb_right) {
                              node = rb_entry(node->rb.rb_right,
                                      struct interval_tree_node, rb);
                              if (node->__subtree_last >= start)
                                      continue;
                      }
              }
              return NULL;    /* No match */
      }
}
```

Insertion/removal are defined using the following augmented callbacks:

```
static inline unsigned long
compute_subtree_last(struct interval_tree_node *node)
{
      unsigned long max = node->last, subtree_last;
      if (node->rb.rb_left) {
              subtree_last = rb_entry(node->rb.rb_left,
                      struct interval_tree_node, rb)->__subtree_last;
              if (max < subtree_last)
                      max = subtree_last;
      }
      if (node->rb.rb_right) {
              subtree_last = rb_entry(node->rb.rb_right,
                      struct interval_tree_node, rb)->__subtree_last;
              if (max < subtree_last)
                      max = subtree_last;
      }
      return max;
}

static void augment_propagate(struct rb_node *rb, struct rb_node *stop)
{
      while (rb != stop) {
              struct interval_tree_node *node =
                      rb_entry(rb, struct interval_tree_node, rb);
              unsigned long subtree_last = compute_subtree_last(node);
```

```
                if (node->__subtree_last == subtree_last)
                        break;
                node->__subtree_last = subtree_last;
                rb = rb_parent(&node->rb);
        }
}

static void augment_copy(struct rb_node *rb_old, struct rb_node *rb_new)
{
        struct interval_tree_node *old =
                rb_entry(rb_old, struct interval_tree_node, rb);
        struct interval_tree_node *new =
                rb_entry(rb_new, struct interval_tree_node, rb);

        new->__subtree_last = old->__subtree_last;
}

static void augment_rotate(struct rb_node *rb_old, struct rb_node *rb_new)
{
        struct interval_tree_node *old =
                rb_entry(rb_old, struct interval_tree_node, rb);
        struct interval_tree_node *new =
                rb_entry(rb_new, struct interval_tree_node, rb);

        new->__subtree_last = old->__subtree_last;
        old->__subtree_last = compute_subtree_last(old);
}

static const struct rb_augment_callbacks augment_callbacks = {
        augment_propagate, augment_copy, augment_rotate
};

void interval_tree_insert(struct interval_tree_node *node,
                          struct rb_root *root)
{
        struct rb_node **link = &root->rb_node, *rb_parent = NULL;
        unsigned long start = node->start, last = node->last;
        struct interval_tree_node *parent;

        while (*link) {
                rb_parent = *link;
                parent = rb_entry(rb_parent, struct interval_tree_node, rb);
                if (parent->__subtree_last < last)
                        parent->__subtree_last = last;
                if (start < parent->start)
                        link = &parent->rb.rb_left;
                else
                        link = &parent->rb.rb_right;
        }

        node->__subtree_last = last;
        rb_link_node(&node->rb, rb_parent, link);
        rb_insert_augmented(&node->rb, root, &augment_callbacks);
}

void interval_tree_remove(struct interval_tree_node *node,
                          struct rb_root *root)
{
        rb_erase_augmented(&node->rb, root, &augment_callbacks);
}
```