Red-black Trees (rbtree) in Linux
January 18, 2007
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 anticipatory, deadline, and CFQ I/O schedulers all 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
    http://lwn.net/Articles/184495/

  Wikipedia entry on red-black trees
    http://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);
```

Support for Augmented rbtrees

Augmented rbtree is an rbtree with "some" additional data stored in each node. 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. rbtree user who wants this feature will have an augment callback function in rb_root initialized.

This callback function will be called from rbtree core routines whenever a node has a change in one or both of its children. It is the responsibility of the callback function to recalculate the additional data that is in the rb node using new children information. Note that if this new additional data affects the parent node's additional data, then callback function has to handle it and do the recursive updates.

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 descendents. 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:

```
find_lowest_match(lo, hi, node)
{
	lowest_match = NULL;
	while (node) {
		if (max_hi(node->left) > lo) {
			// Lowest overlap if any must be on left side
			node = node->left;
		} else if (overlap(lo, hi, node)) {
			lowest_match = node;
			break;
		} else if (lo > node->lo) {
			// Lowest overlap if any must be on right side
			node = node->right;
		} else {
			break;
		}
	}
	return lowest_match;
}
```

Finding exact match will be to first find lowest match and then to follow
successor nodes looking for exact match, until the start of a node is beyond
the hi value we are looking for.