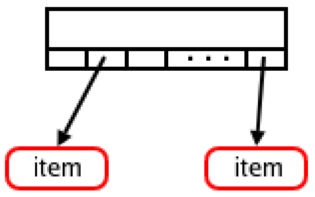
Trees I: Radix trees

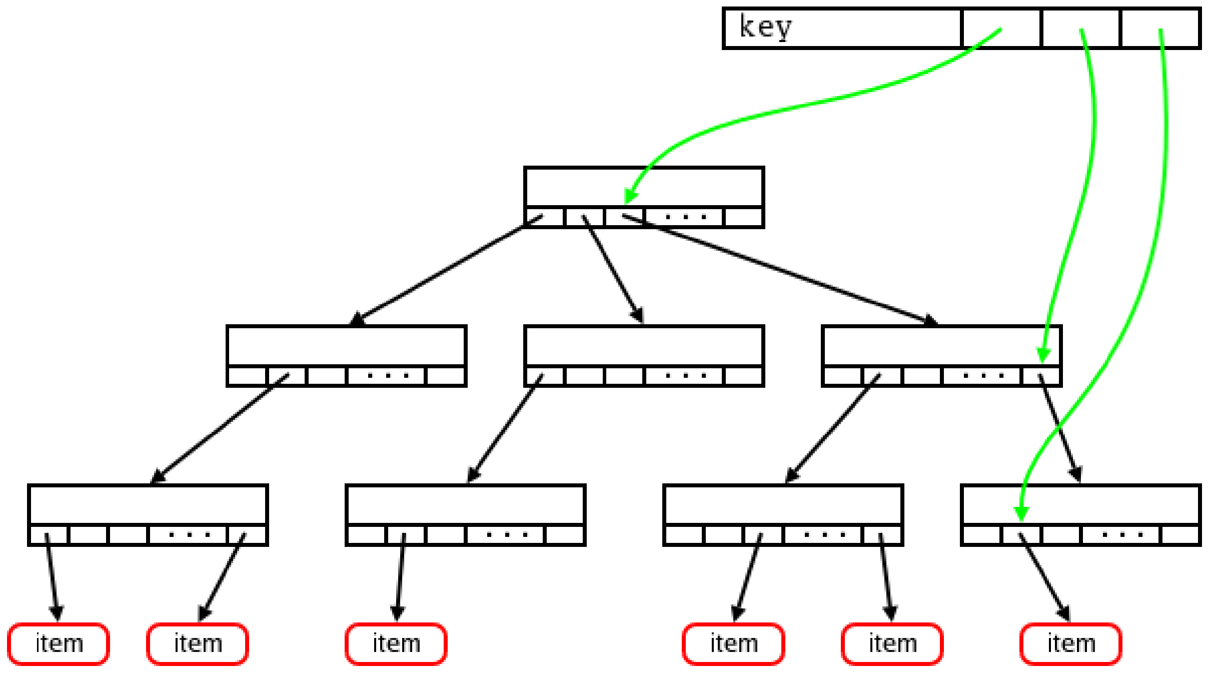
[Posted March 13, 2006 by corbet]

The kernel includes a number of library routines for the implementation of useful data structures. Among those are two types of trees: radix trees and red-black trees. This article will have a look at the radix tree API, with red-black trees to follow in the future.

Wikipedia has a radix tree article, but Linux radix trees are not well described by that article. A Linux radix tree is a mechanism by which a (pointer) value can be associated with a (long) integer key. It is reasonably efficient in terms of storage, and is quite quick on lookups. Additionally, radix trees in the Linux kernel have some features driven by kernel-specific needs, including the ability to associate tags with specific entries.



The cheesy diagram on the right shows a leaf node from a Linux radix tree. The node contains a number of slots, each of which can contain a pointer to something of interest to the creator of the tree. Empty slots contain a `NULL` pointer. These trees are quite broad - in the 2.6.16-rc kernels, there are 64 slots in each radix tree node. Slots are indexed by a portion of the (long) integer key. If the highest key value is less than 64, the entire tree can be represented with a single node. Normally, however, a rather larger set of keys is in use - otherwise, a simple array could have been used. So a larger tree might look something like this:



This tree is three levels deep. When the kernel goes to look up a specific key, the most significant six bits will be used to find the appropriate slot in the root node. The next six bits then index the slot in the middle node, and the least significant six bits will indicate the slot containing a pointer to the actual value. Nodes which have no children are not present in the tree, so a radix tree can provide efficient storage for sparse trees.

Radix trees have a few users in the mainline kernel tree. The PowerPC architecture uses a tree to map between real and virtual IRQ numbers. The NFS code attaches a tree to relevant `inode` structures to keep track of outstanding requests. The most widespread use of radix trees, however, is in the memory management code.

The `address_space` structure used to keep track of backing store contains a radix tree which tracks in-core pages tied to that mapping. Among other things, this tree allows the memory management code to quickly find pages which are dirty or under writeback.

As is typical with kernel data structures, there are two modes for declaring and initializing radix trees:

```
#include <linux/radix-tree.h>

RADIX_TREE(name, gfp_mask);  /* Declare and initialize */

struct radix_tree_root my_tree;
INIT_RADIX_TREE(my_tree, gfp_mask);
```

The first form declares and initializes a radix tree with the given `name`; the second form performs the initialization at run time. In either case, a `gfp_mask` must be provided to tell the code how memory allocations are to be performed. If radix tree operations (insertions, in particular) are to be performed in atomic context, the given mask should be `GFP_ATOMIC`.

The functions for adding and removing entries are straightforward:

```
int radix_tree_insert(struct radix_tree_root *tree, unsigned long key,
                      void *item);
void *radix_tree_delete(struct radix_tree_root *tree, unsigned long key);
```

A call to `radix_tree_insert()` will cause the given `item` to be inserted (associated with `key`) in the given `tree`. This operation may require memory allocations; should an allocation fail, the insertion will fail and the return value will be `-ENOMEM`. The code will refuse to overwrite an existing entry; if `key` already exists in the tree, `radix_tree_insert()` will return `-EEXIST`. On success, the return value is zero. `radix_tree_delete()` removes the item associated with `key` from `tree`, returning a pointer to that item if it was present.

There are situations where failure to insert an item into a radix tree can be a significant problem. To help avoid such situations, a pair of specialized functions are provided:

```
int radix_tree_preload(gfp_t gfp_mask);
void radix_tree_preload_end(void);
```

This function will attempt to allocate sufficient memory (using the given `gfp_mask`) to guarantee that the next radix tree insertion cannot fail. The allocated structures are stored in a per-CPU variable, meaning that the calling function must perform the insertion before it can schedule or be moved to a different processor. To that end, `radix_tree_preload()` will, when successful, return with preemption disabled; the caller must eventually ensure that preemption is enabled again by calling `radix_tree_preload_end()`. On failure, `-ENOMEM` is returned and preemption is *not* disabled.

Radix tree lookups can be done in a few ways:

```
void *radix_tree_lookup(struct radix_tree_root *tree, unsigned long key);
void **radix_tree_lookup_slot(struct radix_tree_root *tree, unsigned long key);
unsigned int radix_tree_gang_lookup(struct radix_tree_root *root,
                                    void **results,
                                    unsigned long first_index,
                                    unsigned int max_items);
```

The simplest form, `radix_tree_lookup()`, looks for `key` in the `tree` and returns the associated item (or `NULL` on failure). `radix_tree_lookup_slot()` will, instead, return a pointer to the slot holding the pointer to the item. The caller can, then, change the pointer to associate a new item with the `key`. If the item does not exist, however, `radix_tree_lookup_slot()` will not create a slot for it, so this function cannot be used in place of `radix_tree_insert()`.

Finally, a call to `radix_tree_gang_lookup()` will return up to `max_items` items in `results`, with ascending key values starting at `first_index`. The number of items returned may be less than requested, but a short return (other than zero) does not imply that there are no more values in the tree.

One should note that none of the radix tree functions perform any sort of locking internally. It is up to the caller to ensure that multiple threads do not corrupt the tree or get into other sorts of unpleasant race conditions. Nick Piggin currently has a patch circulating which would use read-copy-update to free tree nodes; this patch would allow lookup operations to be performed without locking as long as (1) the resulting pointer is only used in atomic context, and (2) the calling code avoids creating race conditions of its own. It is not clear when that patch might be merged, however.

The radix tree code supports a feature called "tags," wherein specific bits can be set on items in the tree. Tags are used, for example, to mark memory pages which are dirty or under writeback. The API for working with tags is:

```
    void *radix_tree_tag_set(struct radix_tree_root *tree,
                             unsigned long key, int tag);
    void *radix_tree_tag_clear(struct radix_tree_root *tree,
                             unsigned long key, int tag);
    int radix_tree_tag_get(struct radix_tree_root *tree,
                             unsigned long key, int tag);
```

`radix_tree_tag_set()` will set the given `tag` on the item indexed by `key`; it is an error to attempt to set a tag on a nonexistent key. The return value will be a pointer to the tagged item. While `tag` looks like an arbitrary integer, the code as currently written allows for a maximum of two tags. Use of any tag value other than zero or one will silently corrupt memory in some undesirable place; consider yourself warned.

Tags can be removed with `radix_tree_tag_clear()`; once again, the return value is a pointer to the (un)tagged item. The function `radix_tree_tag_get()` will check whether the item indexed by `key` has the given `tag` set; the return value is zero if `key` is not present, -1 if `key` is present but `tag` is not set, and +1 otherwise. This function is currently commented out in the source, however, since no in-tree code uses it.

There are two other functions for querying tags:

```
    int radix_tree_tagged(struct radix_tree_root *tree, int tag);
    unsigned int radix_tree_gang_lookup_tag(struct radix_tree_root *tree,
                                            void **results,
                                            unsigned long first_index,
                                            unsigned int max_items,
                                            int tag);
```

`radix_tree_tagged()` returns a non-zero value if any item in the tree bears the given `tag`. A list of items with a given tag can be obtained with `radix_tree_gang_lookup_tag()`.

In concluding, we can note one other interesting aspect of the radix tree API: there is no function for destroying a radix tree. It is, evidently, assumed that radix trees will last forever. In practice, deleting all items from a radix tree will free all memory associated with it other than the root node, which can then be disposed of normally.

Index entries for this article	
Kernel	Radix tree

(Log in to post comments)

Node size and cache-line ping pong on SMP

Posted Mar 17, 2006 21:27 UTC (Fri) by **jzbiciak** (guest, #5246) [Link]

I wonder what's magic about 64, other than that it's 6 bits? On a 32-bit architecture, that's 256 bytes of storage for the 64 pointers. On a machine like the Athlon, cache lines are 64 bytes, which means each node of the tree occupies 4 cache lines. In an SMP context, is there an advantage to sizing the nodes down to occupy a single cache line, or is there a sort of hashing benefit to having each node require multiple cache lines? e.g. Would I get more pingponging or less if the nodes were smaller? Does the answer change based on # of CPUs?

Reply to this comment

Trees I: Radix trees

Posted Mar 19, 2006 8:09 UTC (Sun) by **ncm** (subscriber, #165) [Link]

The designer of Judy trees has expressed wonder at why anybody ever talks about binary trees (e.g. red-black trees) any more, since his measurements indicate that no matter what you do, their performance always stinks compared to modern cache-aware techniques. The only reasonable explanation is the same as the reason university graduates, once, all knew ancient Greek, and had studied geometry but not calculus.

Reply to this comment

Trees I: Radix trees

Posted Jun 30, 2006 23:22 UTC (Fri) by **wahern** (subscriber, #37304) [Link]

The author/inventor put it well himself when he said, "Well I cannot describe Judy in 10 minutes -- what possessed me?"

Source: http://judy.sourceforge.net/doc/10minutes.htm

Simplicity is often a very valuable quality, especially in software development.

Reply to this comment

Judy licensing

Posted Jul 18, 2007 7:59 UTC (Wed) by **iler** (guest, #46313) [Link]

Another reason, besides simlpicity, is licensing.

Is Judy GPLed ?

Reply to this comment

Judy licensing

Posted Oct 23, 2008 21:31 UTC (Thu) by **bcl** (subscriber, #17631) [Link]

LGPL according to the article linked above

Reply to this comment

Trees I: Radix trees

Posted Jul 22, 2013 10:25 UTC (Mon) by **puchuu** (guest, #92032) [Link]

Judy author sold himself and his ideas to HP. Why linux should collect HP patents? (see USA patent 6735595)

Also the fastest hash table in the world is "burst trie", not judy arrays. Linux can use fork of "burst trie" idea named "hat trie" in future. It is more memory efficient. These ideas are free, no patents.

Reply to this comment

Trees I: Radix trees

Posted May 28, 2015 18:44 UTC (Thu) by **aaabr** (guest, #102856) [Link]

How well would this apply to large addresses (i.e IPv6)? It has 64 slots, so will it be able to hold addresses much larger than 64 bits? Thank you

Reply to this comment

Trees I: Radix trees

Posted Sep 25, 2019 19:37 UTC (Wed) by **Naarayanan** (guest, #134635) [Link]

How do you iterate/traverse through a radix tree and print out its contents to the console. Can we use gang_lookup() or is there any other API for that?

Reply to this comment