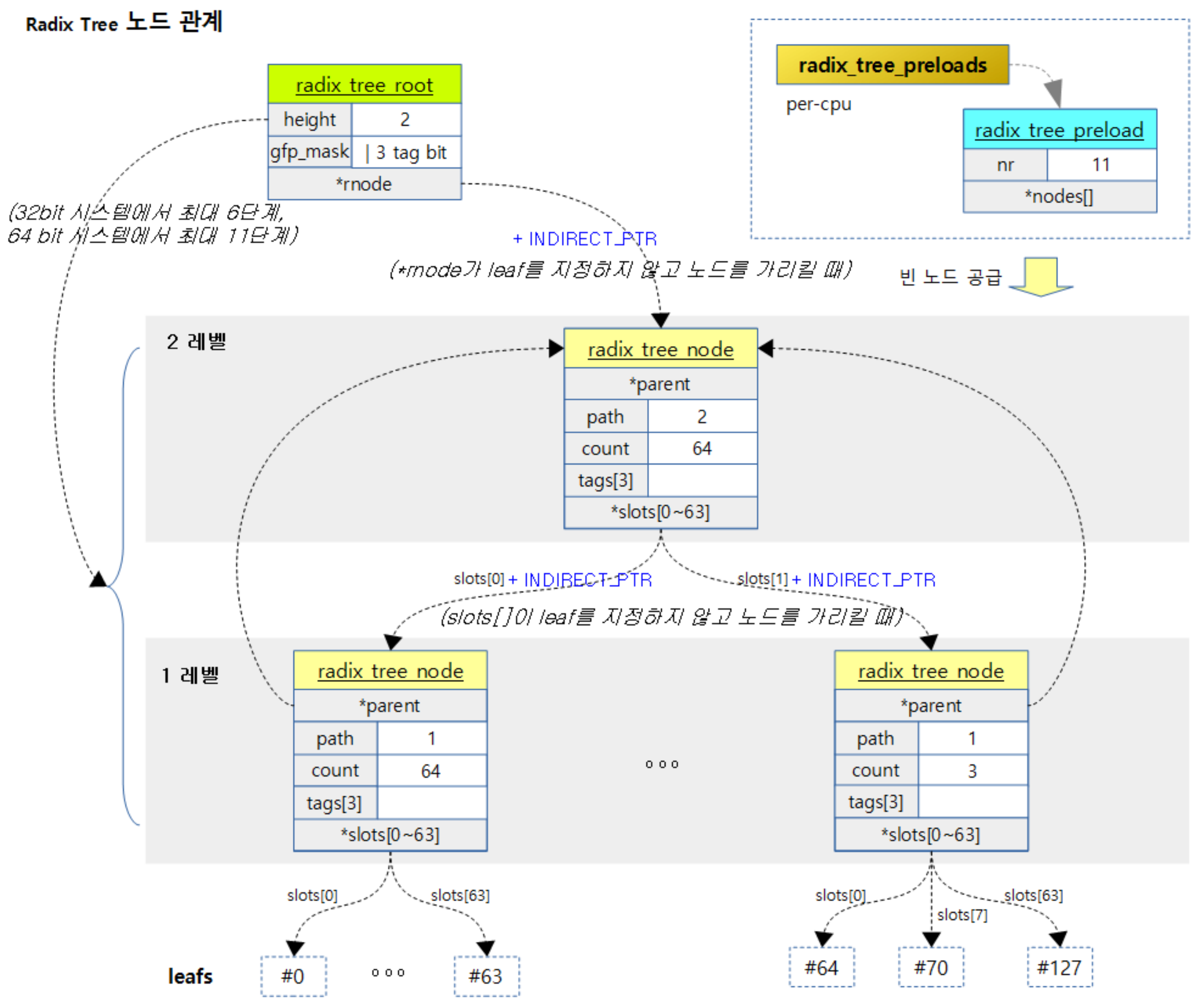
Radix Tree

2016-09-26 (http://jake.dothome.co.kr/radix-tree/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

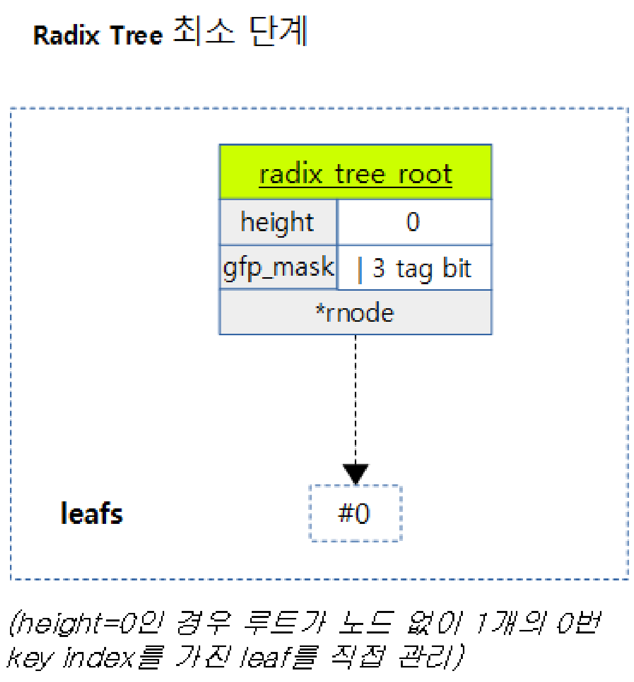
Radix Tree

- Dynamically, you can store the pointer value in a slot that corresponds to an integer index key.
- It's a good idea to use a small integer index key, as using a large index key from the start will expand and slow down the tree steps.
- Kernel version 2.6.17 has a lockless implementation.
  - 참고: radix-tree: updates and lockless (http://lwn.net/Articles/188483/)

The following diagram depicts the structure of a two-stage radix tree.

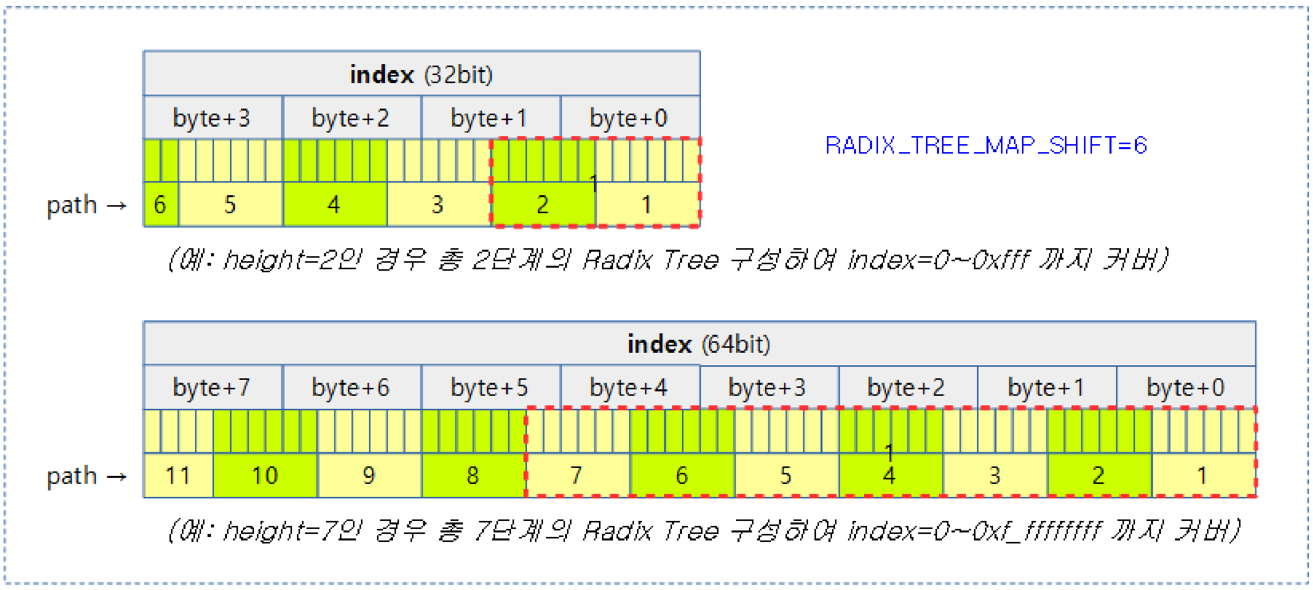


The following figure shows that the minimum level 0 of the Radix Tree can register the index key 0 times. If you add another numbered index key, the radix tree height will expand by the required steps.


Radix Tree 최소 단계
radix_tree_root
height
0
gfp_mask
| 3 tag bit
*rnode
leafs
#0
(height=0인 경우 루트가 노드 없이 1개의 0번 key index를 가진 leaf를 직접 관리)

(http://jake.dothome.co.kr/wp-content/uploads/2016/09/radix_tree-1.png)

The following figure shows how the size of the index value determines the steps of the Radix Tree. (Step-by-step using 6bit)


index (32bit)
byte+3
byte+2
byte+1
byte+0
path →
6
5
4
3
2
1
RADIX_TREE_MAP_SHIFT=6
(예: height=2인 경우 총 2단계의 Radix Tree 구성하여 index=0~0xfff 까지 커버)
index (64bit)
byte+7
byte+6
byte+5
byte+4
byte+3
byte+2
byte+1
byte+0
path →
11
10
9
8
7
6
5
4
3
2
1
(예: height=7인 경우 총 7단계의 Radix Tree 구성하여 index=0~0xf_fffffff 까지 커버)

(http://jake.dothome.co.kr/wp-content/uploads/2016/09/radix_tree-3a.png)

Declaring a Radix Tree

There are two ways to declare and initialize a Radix tree:

- RADIX_TREE(name, mask);
- struct radix_tree_root my_tree;   INIT_RADIX_TREE(my_tree, gfp_mask);

include/linux/radix-tree.h

```
#define RADIX_TREE(name, mask) \
        struct radix_tree_root name = RADIX_TREE_INIT(mask)
```

Declare the radix_tree_root structure with the requested name and initialize it by assigning gfp_mask.

include/linux/radix-tree.h

```
#define RADIX_TREE_INIT(mask)   {
\
        .height = 0,
\
        .gfp_mask = (mask),
\
        .rnode = NULL,
\
}
```

include/linux/radix-tree.h

```
#define INIT_RADIX_TREE(root, mask)
\
do {
\
        (root)->height = 0;
\
        (root)->gfp_mask = (mask);
\
        (root)->rnode = NULL;
\
} while (0)
```

Adding and Deleting Radix Trees

There are commands to add and delete items in the Radix tree.

- radix_tree_insert(root, index, item)
- radix_tree_delete(root, index)

radix_tree_insert()

lib/radix-tree.c

```
/**
 *      radix_tree_insert    -    insert into a radix tree
 *      @root:          radix tree root
 *      @index:         index key
 *      @item:          item to insert
 *
 *      Insert an item into the radix tree at position @index.
 */
int radix_tree_insert(struct radix_tree_root *root,
                        unsigned long index, void *item)
{
        struct radix_tree_node *node;
        void **slot;
        int error;

        BUG_ON(radix_tree_is_indirect_ptr(item));

        error = __radix_tree_create(root, index, &node, &slot);
        if (error)
                return error;
        if (*slot != NULL)
                return -EEXIST;
        rcu_assign_pointer(*slot, item);

        if (node) {
                node->count++;
                BUG_ON(tag_get(node, 0, index & RADIX_TREE_MAP_MASK));
                BUG_ON(tag_get(node, 1, index & RADIX_TREE_MAP_MASK));
        } else {
                BUG_ON(root_tag_get(root, 0));
                BUG_ON(root_tag_get(root, 1));
        }

        return 0;
}
EXPORT_SYMBOL(radix_tree_insert);
```

Assign an item pointer to the slot that corresponds to the index in the radix tree.

- error = __radix_tree_create(root, index, &node, &slot);
    - Prepare a radix tree slot with the index key number.
- if (error) return error;
    - Returns an error in the event of an error.
- if (*slot != NULL) return -EEXIST;
    - If the slot is not null, it returns an error that it already exists.
- rcu_assign_pointer(*slot, item);
    - Assign an item to the slot.

__radix_tree_create()

lib/radix-tree.c

```
/**
 *      __radix_tree_create     -       create a slot in a radix tree
 *      @root:          radix tree root
 *      @index:         index key
 *      @nodep:         returns node
 *      @slotp:         returns slot
 *
 *      Create, if necessary, and return the node and slot for an item
 *      at position @index in the radix tree @root.
 *
 *      Until there is more than one item in the tree, no nodes are
 *      allocated and @root->rnode is used as a direct slot instead of
 *      pointing to a node, in which case *@nodep will be NULL.
 *
 *      Returns -ENOMEM, or 0 for success.
 */
```

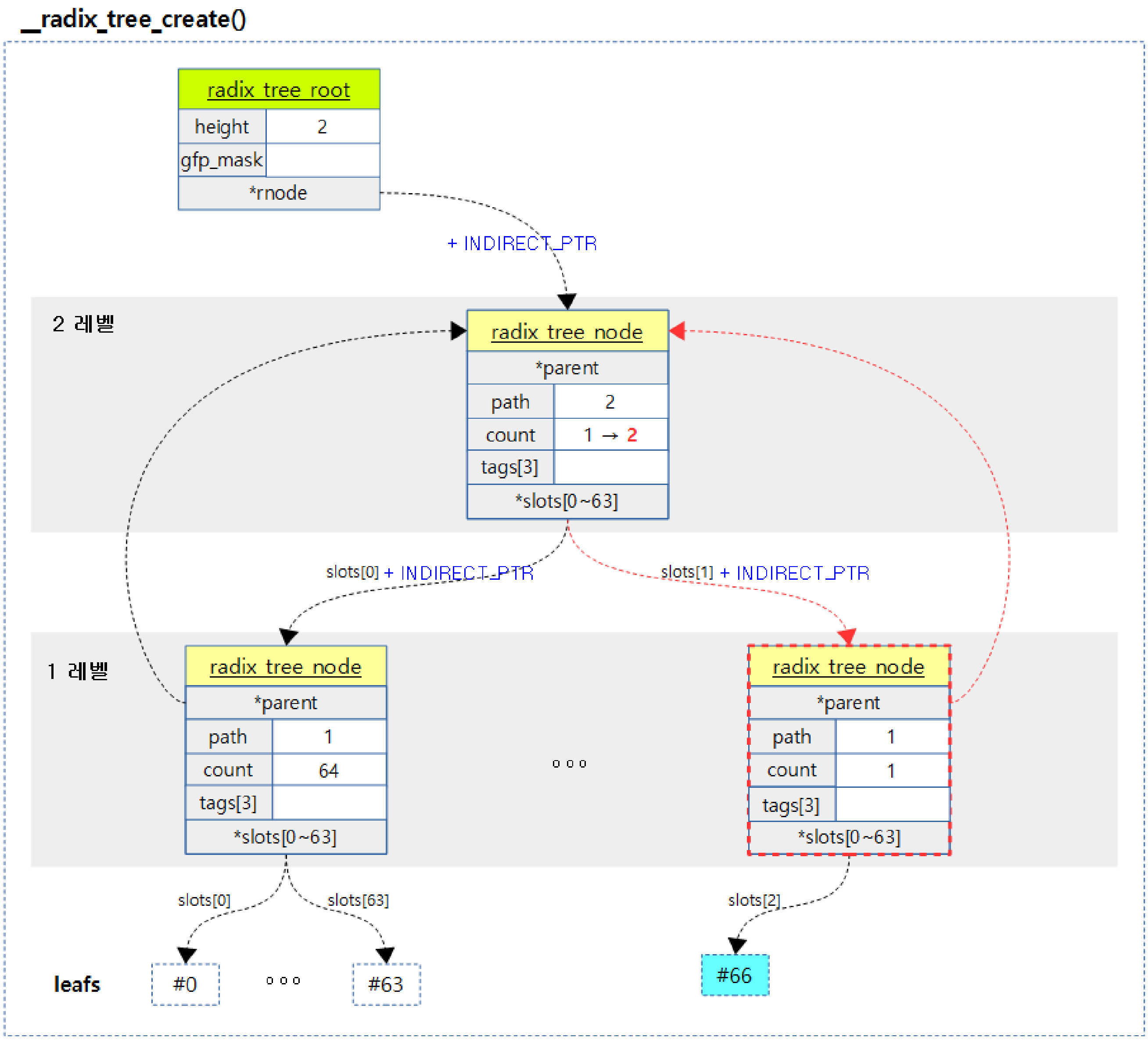
```
int __radix_tree_create(struct radix_tree_root *root, unsigned long index,
                        struct radix_tree_node **nodep, void ***slotp)
{
        struct radix_tree_node *node = NULL, *slot;
        unsigned int height, shift, offset;
        int error;

        /* Make sure the tree is high enough.  */
        if (index > radix_tree_maxindex(root->height)) {
                error = radix_tree_extend(root, index);
                if (error)
                        return error;
        }

        slot = indirect_to_ptr(root->rnode);

        height = root->height;
        shift = (height-1) * RADIX_TREE_MAP_SHIFT;

        offset = 0;                     /* uninitialised var warning */
        while (height > 0) {
                if (slot == NULL) {
                        /* Have to add a child node.  */
                        if (!(slot = radix_tree_node_alloc(root)))
                                return -ENOMEM;
                        slot->path = height;
                        slot->parent = node;
                        if (node) {
                                rcu_assign_pointer(node->slots[offset], slot);
                                node->count++;
                                slot->path |= offset << RADIX_TREE_HEIGHT_SHIFT;
                        } else
                                rcu_assign_pointer(root->rnode, ptr_to_indirect(slot));
                }

                /* Go a level down */
                offset = (index >> shift) & RADIX_TREE_MAP_MASK;
                node = slot;
                slot = node->slots[offset];
                shift -= RADIX_TREE_MAP_SHIFT;
                height--;
        }

        if (nodep)
                *nodep = node;
        if (slotp)
                *slotp = node ? node->slots + offset : (void **)&root->rnode;
        return 0;
}
```

Finds the node and slot corresponding to the index key. If you need to expand, do it as well.

- if (index > radix_tree_maxindex(root->height)) { error = radix_tree_extend(root, index); if (error) return error; }
  - If the index key exceeds the maximum, the radix tree is expanded. If it is an error, it returns an error.
- slot = indirect_to_ptr(root->rnode);
  - Remove the bit (bit1) where RADIX_TREE_INDIRECT_PTR(0) is located from the ptr.
- height = root->height; shift = (height-1) * RADIX_TREE_MAP_SHIFT;

    - Determine the shift value with the height of the current radix tree.
        - e.g. height=3, RADIX_TREE_MAP_SHIFT=6
            - shift=12
- while (height > 0) {
    - If the height is greater than zero, it goes around the loop.
- if (slot == NULL) {
    - If the slot is empty
- if (!(slot = radix_tree_node_alloc(root))) return -ENOMEM;
    - You are assigned a radix tree node. If an error occurs, it returns an out-of-memory error.
- slot->path = height; slot->parent = node;
    - Assign height to the path of the slot, and node to parent.
- if (node) { rcu_assign_pointer(node->slots[offset], slot); node->count++; slot->path |= offset << RADIX_TREE_HEIGHT_SHIFT;
    - If it is not the root node, assign a slot to the node's slots[offset], increment the count, and assign the offset value to the path shifted to the left by RADIX_TREE_HEIGHT_SHIFT degree.
- } else rcu_assign_pointer(root->rnode, ptr_to_indirect(slot));
    - If it is the root node, assign root->rnode to the slot pointer plus RADIX_TREE_INDEIRECT_PTR(1).
- offset = (index >> shift) & RADIX_TREE_MAP_MASK;
    - Assign the index bit to be processed at the next level from the index value to offset.
- node = slot; slot = node->slots[offset];
    - Know the node of the next level.
- shift -= RADIX_TREE_MAP_SHIFT; height–;
    - Decreases shift by RADIX_TREE_MAP_SHIFT and height by 1.
- if (nodep) *nodep = node;
    - Given the argument nodep, assign node.
- if (slotp) *slotp = node ? node->slots + offset : (void **)&root->rnode;
    - If the argument slotp is given, the slot is substituted.
        - If node is null, it means that the radix_tree_root node manages the leaf directly without radix_tree_node.

The following diagram shows the process of creating and configuring the intermediate nodes needed to add the required index keys.



(http://jake.dothome.co.kr/wp-content/uploads/2016/09/radix_tree_create-1.png)

radix_tree_extend()

lib/radix-tree.c

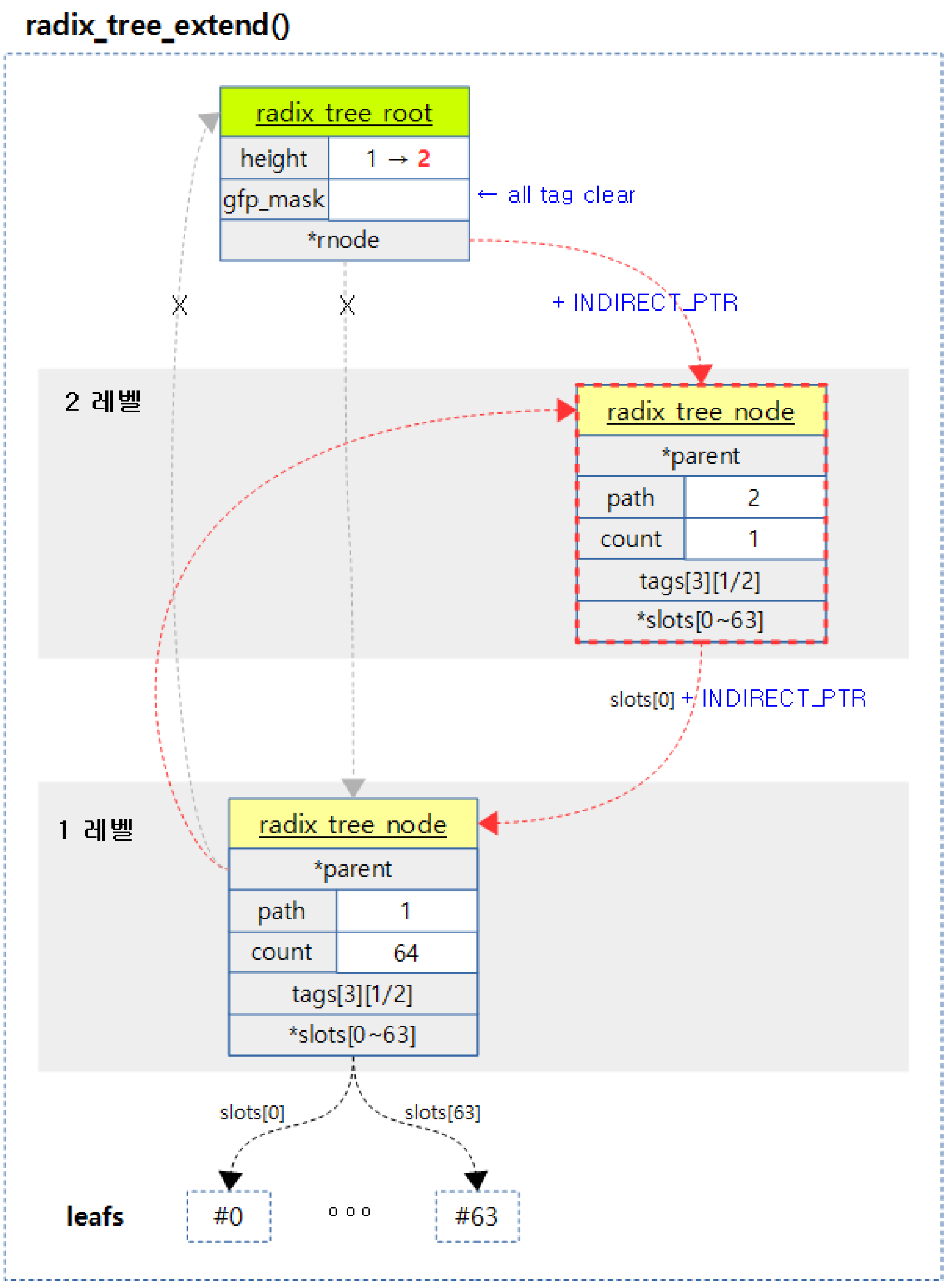
```
/*
 *      Extend a radix tree so it can store key @index.
 */
static int radix_tree_extend(struct radix_tree_root *root, unsigned long
index)
{
        struct radix_tree_node *node;
        struct radix_tree_node *slot;
        unsigned int height;
        int tag;

        /* Figure out what the height should be.  */
        height = root->height + 1;
        while (index > radix_tree_maxindex(height))
                height++;

        if (root->rnode == NULL) {
                root->height = height;
                goto out;
        }

        do {
                unsigned int newheight;
                if (!(node = radix_tree_node_alloc(root)))
                        return -ENOMEM;

                /* Propagate the aggregated tag info into the new root
*/
                for (tag = 0; tag < RADIX_TREE_MAX_TAGS; tag++) {
                        if (root_tag_get(root, tag))
                                tag_set(node, tag, 0);
                }

                /* Increase the height.  */
                newheight = root->height+1;
                BUG_ON(newheight & ~RADIX_TREE_HEIGHT_MASK);
                node->path = newheight;
                node->count = 1;
                node->parent = NULL;
                slot = root->rnode;
                if (newheight > 1) {
                        slot = indirect_to_ptr(slot);
                        slot->parent = node;
                }
                node->slots[0] = slot;
                node = ptr_to_indirect(node);
                rcu_assign_pointer(root->rnode, node);
                root->height = newheight;
        } while (height > root->height);
out:
        return 0;
}
```

To extend the Radix tree node, add a new root node and connect the existing node to the first slot of the newly created node. Perform these steps as many steps as you need to scale.

- height = root->height + 1; while (index > radix_tree_maxindex(height)) height++;
    - Set the height value to the level that the radix tree can handle at the moment.
- if (root->rnode == NULL) { root->height = height; goto out; }
    - If the slot is empty, set the root->height value and exit the function.
- do { unsigned int newheight; if (!(node = radix_tree_node_alloc(root))) return -ENOMEM;
    - Create a new root node.
- for (tag = 0; tag < RADIX_TREE_MAX_TAGS; tag++) { if (root_tag_get(root, tag)) tag_set(node, tag, 0); }
    - It loops up to the maximum number of tag bits, and if each tag bit on the root node is 1, it initializes to 0.
    - The tag bits are used in order from __GFP_BITS_SHIFT (25) bits of the gfp_mask.
- newheight = root->height+1; node->path = newheight;
    - Assign the path of the new root node by increasing the height value by 1.
- node->count = 1; node->parent = NULL;
    - Assign 1 to the count of the root node and specify that there is no parent node.
- slot = root->rnode; if (newheight > 1) { slot = indirect_to_ptr(slot); slot->parent = node; } node->slots[0] = slot;
    - Connect the existing root node to the first slot of the root node.
- node = ptr_to_indirect(node); rcu_assign_pointer(root->rnode, node);
    - root->rnode에

- root->height = newheight;
  - Set the hieght value for the new root node.
- } while (height > root->height);
  - If the height is greater than the height of the current root node, the loop is repeated to create a new root node.

The following illustration shows the radix tree from step 1 expanding to phase 2.



(http://jake.dothome.co.kr/wp-content/uploads/2016/09/radix_tree_extend-1.png)

radix_tree_node_alloc()

lib/radix-tree.c

```
/*
 * This assumes that the caller has performed appropriate preallocation,
and
```

```
 * that the caller has pinned this thread of control to the current CPU.
 */
static struct radix_tree_node *
radix_tree_node_alloc(struct radix_tree_root *root)
{
        struct radix_tree_node *ret = NULL;
        gfp_t gfp_mask = root_gfp_mask(root);

        /*
         * Preload code isn't irq safe and it doesn't make sence to use
         * preloading in the interrupt anyway as all the allocations have to
         * be atomic. So just do normal allocation when in interrupt.
         */
        if (!(gfp_mask & __GFP_WAIT) && !in_interrupt()) {
                struct radix_tree_preload *rtp;

                /*
                 * Provided the caller has preloaded here, we will always
                 * succeed in getting a node here (and never reach
                 * kmem_cache_alloc)
                 */
                rtp = this_cpu_ptr(&radix_tree_preloads);
                if (rtp->nr) {
                        ret = rtp->nodes[rtp->nr - 1];
                        rtp->nodes[rtp->nr - 1] = NULL;
                        rtp->nr--;
                }
                /*
                 * Update the allocation stack trace as this is more useful
                 * for debugging.
                 */
                kmemleak_update_trace(ret);
        }
        if (ret == NULL)
                ret = kmem_cache_alloc(radix_tree_node_cachep, gfp_mask);

        BUG_ON(radix_tree_is_indirect_ptr(ret));
        return ret;
}
```

You will be assigned a radix tree node.

- Returns the nodes that are ready in the radix_tree_preloads.
- If there are no nodes ready in the radix_tree_preloads, they will be allocated from the slub cache.

- gfp_t gfp_mask = root_gfp_mask(root);
  - Finds the value of the root node's gfp_mask minus the tag.
- if (!(gfp_mask & __GFP_WAIT) && !in_interrupt()) {
  - If the interrupt is not being processed and there is no __GFP_WAIT request
- rtp = this_cpu_ptr(&radix_tree_preloads); if (rtp->nr) { ret = rtp->nodes[rtp->nr – 1]; rtp->nodes[rtp->nr – 1] = NULL; rtp->nr–; }
  - If there is a node managed by a global radix_tree_preloads structure, pull one node.
- if (ret == NULL) ret = kmem_cache_alloc(radix_tree_node_cachep, gfp_mask);
  - If the node is not prepared by preload, it will be allocated from the slub cache.

radix_tree_delete()

lib/radix-tree.c

```
/**
 *      radix_tree_delete    -    delete an item from a radix tree
 *      @root:          radix tree root
 *      @index:         index key
 *
 *      Remove the item at @index from the radix tree rooted at @root.
 *
 *      Returns the address of the deleted item, or NULL if it was not present.
 */
void *radix_tree_delete(struct radix_tree_root *root, unsigned long index)
{
        return radix_tree_delete_item(root, index, NULL);
}
EXPORT_SYMBOL(radix_tree_delete);
```

Remove the request index key entry from the Radix tree.

radix_tree_delete_item()

lib/radix-tree.c

```
/**
 *      radix_tree_delete_item    -    delete an item from a radix tree
 *      @root:          radix tree root
 *      @index:         index key
 *      @item:          expected item
 *
 *      Remove @item at @index from the radix tree rooted at @root.
 *
 *      Returns the address of the deleted item, or NULL if it was not present
 *      or the entry at the given @index was not @item.
 */
void *radix_tree_delete_item(struct radix_tree_root *root,
                             unsigned long index, void *item)
{
        struct radix_tree_node *node;
        unsigned int offset;
        void **slot;
        void *entry;
        int tag;

        entry = __radix_tree_lookup(root, index, &node, &slot);
        if (!entry)
                return NULL;

        if (item && entry != item)
                return NULL;

        if (!node) {
                root_tag_clear_all(root);
                root->rnode = NULL;
                return entry;
        }

        offset = index & RADIX_TREE_MAP_MASK;

        /*
         * Clear all tags associated with the item to be deleted.
         * This way of doing it would be inefficient, but seldom is any set.
         */
        for (tag = 0; tag < RADIX_TREE_MAX_TAGS; tag++) {
                if (tag_get(node, tag, offset))
                        radix_tree_tag_clear(root, index, tag);
        }

        node->slots[offset] = NULL;
        node->count--;

        __radix_tree_delete_node(root, node);

        return entry;
}
EXPORT_SYMBOL(radix_tree_delete_item);
```

Removes the request index key entry from the Radix tree and returns the removed entry.

- entry = __radix_tree_lookup(root, index, &node, &slot);
    - Search the Radix tree for the request index key entry.
- if (item && entry != item) return NULL;
    - If the entry address and the item address are different (mismatch) found by searching, it returns null.
- if (!node) { root_tag_clear_all(root); root->rnode = NULL; return entry; }
    - If it is not a node, i.e. if the key index is zero, it clears the tag and ptr value from the root and returns the corresponding data ptr value.
- offset = index & RADIX_TREE_MAP_MASK;
    - offset corresponding to the index key requested by the current node
    - 0~RADIX_TREE_MAP_SIZE(63)
- for (tag = 0; tag < RADIX_TREE_MAX_TAGS; tag++) { if (tag_get(node, tag, offset)) radix_tree_tag_clear(root, index, tag); }
    - If the tag located in offset is set, clear the three tags corresponding to the requested index key. If all 3 surrounding tags managed by the same node are missing, proceed to the parent node and clear the tags.
- node->slots[offset] = NULL; node->count–;
    - Empty a slot and reduce the usage counter by 1.
- __radix_tree_delete_node(root, node);
    - If you don't need a Radix tree node, delete it.
- return entry;
    - Returns the deleted entries.

__radix_tree_lookup()

lib/radix-tree.c

```
/**
 *      __radix_tree_lookup     -       lookup an item in a radix tree
 *      @root:          radix tree root
 *      @index:         index key
 *      @nodep:         returns node
 *      @slotp:         returns slot
 *
 *      Lookup and return the item at position @index in the radix
 *      tree @root.
 *
 *      Until there is more than one item in the tree, no nodes are
 *      allocated and @root->rnode is used as a direct slot instead of
 *      pointing to a node, in which case *@nodep will be NULL.
 */
void *__radix_tree_lookup(struct radix_tree_root *root, unsigned long index,
                          struct radix_tree_node **nodep, void ***slotp)
{
        struct radix_tree_node *node, *parent;
        unsigned int height, shift;
        void **slot;

        node = rcu_dereference_raw(root->rnode);
        if (node == NULL)
                return NULL;

        if (!radix_tree_is_indirect_ptr(node)) {
                if (index > 0)
                        return NULL;

                if (nodep)
                        *nodep = NULL;
                if (slotp)
                        *slotp = (void **)&root->rnode;
                return node;
        }
        node = indirect_to_ptr(node);

        height = node->path & RADIX_TREE_HEIGHT_MASK;
        if (index > radix_tree_maxindex(height))
                return NULL;

        shift = (height-1) * RADIX_TREE_MAP_SHIFT;

        do {
                parent = node;
                slot = node->slots + ((index >> shift) & RADIX_TREE_MAP_MASK);
                node = rcu_dereference_raw(*slot);
                if (node == NULL)
                        return NULL;

                shift -= RADIX_TREE_MAP_SHIFT;
                height--;
        } while (height > 0);

        if (nodep)
                *nodep = parent;
        if (slotp)
                *slotp = slot;
        return node;
}
```

Search the Radix tree for the request index key entry. If not found, it returns null.

- node = rcu_dereference_raw(root->rnode); if (node == NULL) return NULL;
    - Obtain the address of the node connected to the root.
- if (!radix_tree_is_indirect_ptr(node)) {
    - If you have a direct data value and not a node,
- if (index > 0) return NULL;
    - If the request index is greater than 0, it is not found, and returns null.
    - The only time the root has direct data is when the key index is 0.

- if (nodep) *nodep = NULL;
  - Since it is found at the root, there is no Radix tree node, so null is assigned to the output argument nodep.
- if (slotp) *slotp = (void **)&root->rnode; return node;
  - Assign the value of the slot address (the rnode acts as a single slot) to the output argument slotp and return the data ptr value.
- node = indirect_to_ptr(node);
  - Remove unnecessary flag bits, leaving only the actual node address.
  - The current node value is the highest-ranking Radix tree node address.
- height = node->path & RADIX_TREE_HEIGHT_MASK; if (index > radix_tree_maxindex(height)) return NULL;
  - Returns null if the request index key value exceeds the value managed by the height step of the top-level node.
- shift = (height-1) * RADIX_TREE_MAP_SHIFT;
  - Determine the number of bits to shift to handle the top-level node.
- do { parent = node; slot = node->slots + ((index >> shift) & RADIX_TREE_MAP_MASK); node = rcu_dereference_raw(*slot); if (node == NULL) return NULL; shift -= RADIX_TREE_MAP_SHIFT; height–; } while (height > 0);
  - It goes around the loop from the top node through the bottom node to the leaf, looking for the node that is connected.
- if (nodep) *nodep = parent;
  - Assign the output argument nodep to the last node that manages the leaf.
- if (slotp) *slotp = slot; return node;
  - Assign the slot address to the output argument slotp and return the data ptr value.

radix_tree_tag_clear()

lib/radix-tree.c

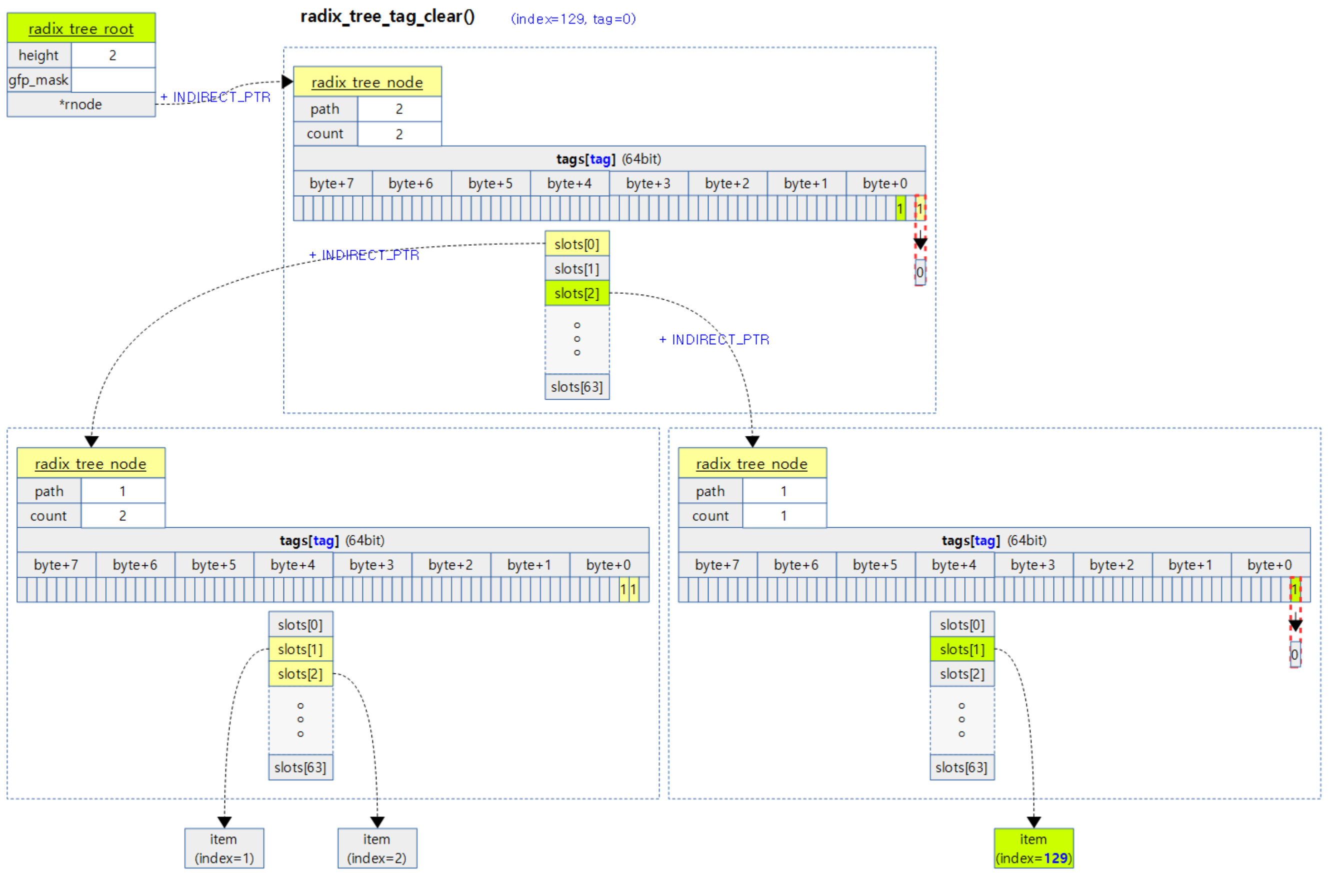
```
/**
 *      radix_tree_tag_clear - clear a tag on a radix tree node
 *      @root:          radix tree root
 *      @index:         index key
 *      @tag:           tag index
 *
 *      Clear the search tag (which must be < RADIX_TREE_MAX_TAGS)
 *      corresponding to @index in the radix tree.  If
 *      this causes the leaf node to have no tags set then clear the tag
in the
 *      next-to-leaf node, etc.
 *
 *      Returns the address of the tagged item on success, else NULL.  i
e:
 *      has the same return value and semantics as radix_tree_lookup().
 */
void *radix_tree_tag_clear(struct radix_tree_root *root,
                        unsigned long index, unsigned int tag)
{
        struct radix_tree_node *node = NULL;
        struct radix_tree_node *slot = NULL;
        unsigned int height, shift;
        int uninitialized_var(offset);

        height = root->height;
        if (index > radix_tree_maxindex(height))
                goto out;

        shift = height * RADIX_TREE_MAP_SHIFT;
        slot = indirect_to_ptr(root->rnode);

        while (shift) {
                if (slot == NULL)
                        goto out;

                shift -= RADIX_TREE_MAP_SHIFT;
                offset = (index >> shift) & RADIX_TREE_MAP_MASK;
                node = slot;
                slot = slot->slots[offset];
        }

        if (slot == NULL)
                goto out;

        while (node) {
                if (!tag_get(node, tag, offset))
                        goto out;
                tag_clear(node, tag, offset);
                if (any_tag_set(node, tag))
                        goto out;

                index >>= RADIX_TREE_MAP_SHIFT;
                offset = index & RADIX_TREE_MAP_MASK;
                node = node->parent;
        }

        /* clear the root's tag bit */
        if (root_tag_get(root, tag))
                root_tag_clear(root, tag);

out:
        return slot;
}
EXPORT_SYMBOL(radix_tree_tag_clear);
```

Clear the tag that corresponds to the requested index key. If all 64 tags managed by the same node are not present, proceed to the parent node and clear the tags.

- height = root->height; if (index > radix_tree_maxindex(height)) goto out;
  - Returns null if the value of the index key in the request exceeds the steps managed by the Radix tree.
- shift = height * RADIX_TREE_MAP_SHIFT;
  - After a while, we'll go around the loop and use the index key to shift the bits we need for each step, so we'll set the shift value to a step one step higher than the top.
- slot = indirect_to_ptr(root->rnode);
  - A slot refers to the top-level node.
- while (shift) { if (slot == NULL) goto out; shift -= RADIX_TREE_MAP_SHIFT; offset = (index >> shift) & RADIX_TREE_MAP_MASK; node = slot; slot = slot->slots[offset]; }
  - Go around the loop to the bottom and assign the lowest node address and slot value to the node and slot.
- if (slot == NULL) goto out;
  - Returns null if the slot is already empty.

- while (node) { if (!tag_get(node, tag, offset)) goto out; tag_clear(node, tag, offset); if (any_tag_set(node, tag)) goto out; index >>= RADIX_TREE_MAP_SHIFT; offset = index & RADIX_TREE_MAP_MASK; node = node->parent; }
    - It loops from the lowest node to the topmost node and clears the tag associated with the index key.
    - If the tag associated with the current node's index key is already empty, the function will exit.
    - If there are still 64 other tag bits associated with the current node during the process, the function exits without proceeding to the top.
- if (root_tag_get(root, tag)) root_tag_clear(root, tag);
    - This means that all tags have been cleared, even on the top-level node, so we will also delete the tags in the root.

The following figure shows that when deleting tag 129 for index 0, the tag of the lower node is deleted first, and if all the tags of the same node are missing, the tag of the higher node is also deleted.



(http://jake.dothome.co.kr/wp-content/uploads/2016/09/radix_tree_tag_clear-1a.png)

__radix_tree_delete_node()

lib/radix-tree.c

```
/**
 *      __radix_tree_delete_node    -    try to free node after clearing
a slot
 *      @root:          radix tree root
 *      @node:          node containing @index
 *
 *      After clearing the slot at @index in @node from radix tree
 *      rooted at @root, call this function to attempt freeing the
 *      node and shrinking the tree.
 *
 *      Returns %true if @node was freed, %false otherwise.
 */
bool __radix_tree_delete_node(struct radix_tree_root *root,
                              struct radix_tree_node *node)
{
        bool deleted = false;

        do {
                struct radix_tree_node *parent;

                if (node->count) {
                        if (node == indirect_to_ptr(root->rnode)) {
                                radix_tree_shrink(root);
                                if (root->height == 0)
                                        deleted = true;
                        }
                        return deleted;
                }

                parent = node->parent;
                if (parent) {
                        unsigned int offset;

                        offset = node->path >> RADIX_TREE_HEIGHT_SHIFT;
                        parent->slots[offset] = NULL;
                        parent->count--;
                } else {
                        root_tag_clear_all(root);
                        root->height = 0;
                        root->rnode = NULL;
                }

                radix_tree_node_free(node);
                deleted = true;

                node = parent;
        } while (node);

        return deleted;
}
```

After shrinking on the request node, delete it if there are no available slots, and repeat by going around the loop to the parent node. Returns true if one or more of them have been deleted.

- do { struct radix_tree_node *parent; if (node->count) { if (node == indirect_to_ptr(root->rnode)) { radix_tree_shrink(root); if (root->height == 0) deleted = true; } return deleted; }
  - If the count of the requested node is greater than 0, it will try to shrink and return the result.
- parent = node->parent; if (parent) { unsigned int offset; offset = node->path >> RADIX_TREE_HEIGHT_SHIFT; parent->slots[offset] = NULL; parent->count–;
  - If there is a parent node, disconnect from the parent node to the current node and reduce the count.
- } else { root_tag_clear_all(root); root->height = 0; root->rnode = NULL; }
  - Remove the top-level node.
  - If the parent node does not exist, clear all root tags, set height to 0, and disconnect item.
- radix_tree_node_free(node); deleted = true; node = parent; } while (node);

- Remove the current node, select its parent, and continue.
  - The parent node is also removed when the count reaches zero.

radix_tree_shrink()

lib/radix-tree.c

```
/**
 *      radix_tree_shrink    -    shrink height of a radix tree to minimal
 *      @root           radix tree root
 */
static inline void radix_tree_shrink(struct radix_tree_root *root)
{
        /* try to shrink tree height */
        while (root->height > 0) {
                struct radix_tree_node *to_free = root->rnode;
                struct radix_tree_node *slot;

                BUG_ON(!radix_tree_is_indirect_ptr(to_free));
                to_free = indirect_to_ptr(to_free);

                /*
                 * The candidate node has more than one child, or its child
                 * is not at the leftmost slot, we cannot shrink.
                 */
                if (to_free->count != 1)
                        break;
                if (!to_free->slots[0])
                        break;

                /*
                 * We don't need rcu_assign_pointer(), since we are simply
                 * moving the node from one part of the tree to another: if it
                 * was safe to dereference the old pointer to it
                 * (to_free->slots[0]), it will be safe to dereference the new
                 * one (root->rnode) as far as dependent read barriers go.
                 */
                slot = to_free->slots[0];
                if (root->height > 1) {
                        slot->parent = NULL;
                        slot = ptr_to_indirect(slot);
                }
                root->rnode = slot;
                root->height--;

                /*
                 * We have a dilemma here. The node's slot[0] must not be
                 * NULLed in case there are concurrent lookups expecting to
                 * find the item. However if this was a bottom-level node,
                 * then it may be subject to the slot pointer being visible
                 * to callers dereferencing it. If item corresponding to
                 * slot[0] is subsequently deleted, these callers would expect
                 * their slot to become empty sooner or later.
                 *
```

```
		 * For example, lockless pagecache will look up a slot, deref
		 * the page pointer, and if the page is 0 refcount it means it
		 * was concurrently deleted from pagecache so try the deref
		 * again. Fortunately there is already a requirement for logic
		 * to retry the entire slot lookup -- the indirect pointer
		 * problem (replacing direct root node with an indirect pointer
		 * also results in a stale slot). So tag the slot as indirect
		 * to force callers to retry.
		 */
		if (root->height == 0)
			*((unsigned long *)&to_free->slots[0]) |=
						RADIX_TREE_INDIRECT_PTR;

		radix_tree_node_free(to_free);
	}
}
```

If you can reduce the Radix tree steps, delete the unnecessary Radix tree nodes and reduce the steps.

- If you can loop around and remove the top-level node, remove it to reduce the Radix tree steps.
- If there is only slot 0 of the top-level node, delete the current node and change the next node to the top-level node.

- while (root->height > 0) { struct radix_tree_node *to_free = root->rnode;
  - If the Radix tree step is more than 1 step, it takes the top-level node connected to the root.
- to_free = indirect_to_ptr(to_free);
  - Remove RADIX_TREE_INDIRECT_PTR bits from the top-level node pointer.
- if (to_free->count != 1) break;
  - If there is not one slot managed by the node, it will stop shrinking and exit.
- if (!to_free->slots[0]) break;
  - If the remaining slot is not 0, stop shrinking and exit.
- slot = to_free->slots[0]; if (root->height > 1) { slot->parent = NULL; slot = ptr_to_indirect(slot); } root->rnode = slot; root->height–;
  - Make the next node connected to the first slot the top-level node.
  - Put null in parent, point the root to the next node, and decrement the height value.
- if (root->height == 0) *((unsigned long *)&to_free->slots[0]) |= RADIX_TREE_INDIRECT_PTR;
  - If the root height value is 0, the value connected to the first slot of the node to be deleted adds RADIX_TREE_INDIRECT_PTR bits, even if it is item.
- radix_tree_node_free(to_free);
  - Use the RCU method to remove nodes.

radix_tree_node_free()

lib/radix-tree.c

```
static inline void
```

```
radix_tree_node_free(struct radix_tree_node *node)
{
        call_rcu(&node->rcu_head, radix_tree_node_rcu_free);
}
```

The Radix tree node is removed in an RCU manner and returned to the slub cache (free).

radix_tree_node_rcu_free()

lib/radix-tree.c

```
static void radix_tree_node_rcu_free(struct rcu_head *head)
{
        struct radix_tree_node *node =
                        container_of(head, struct radix_tree_node, rcu_h
ead);
        int i;

        /*
         * must only free zeroed nodes into the slab. radix_tree_shrink
         * can leave us with a non-NULL entry in the first slot, so clea
r
         * that here to make sure.
         */
        for (i = 0; i < RADIX_TREE_MAX_TAGS; i++)
                tag_clear(node, i, 0);

        node->slots[0] = NULL;
        node->count = 0;

        kmem_cache_free(radix_tree_node_cachep, node);
}
```

Remove the tag from the Radix tree node, assign null to slots[0], reduce the count to 0, and return it to the Radix tree node's slub cache. (free)

Preload Radix Tree Nodes

radix_tree_preload()

lib/radix-tree.c

```
/*
 * Load up this CPU's radix_tree_node buffer with sufficient objects to
 * ensure that the addition of a single element in the tree cannot fai
l.  On
 * success, return zero, with preemption disabled.  On error, return -EN
OMEM
 * with preemption not disabled.
 *
 * To make use of this facility, the radix tree must be initialised with
out
 * __GFP_WAIT being passed to INIT_RADIX_TREE().
 */
int radix_tree_preload(gfp_t gfp_mask)
{
        /* Warn on non-sensical use... */
        WARN_ON_ONCE(!(gfp_mask & __GFP_WAIT));
        return __radix_tree_preload(gfp_mask);
}
EXPORT_SYMBOL(radix_tree_preload);
```

The global per-CPU type radix tree preload structure is pre-allocated with empty radix tree nodes and is prepared to fill them up.

__radix_tree_preload()

lib/radix-tree.c

```
/*
 * Load up this CPU's radix_tree_node buffer with sufficient objects to
 * ensure that the addition of a single element in the tree cannot fai
l.  On
 * success, return zero, with preemption disabled.  On error, return -EN
OMEM
 * with preemption not disabled.
 *
 * To make use of this facility, the radix tree must be initialised with
out
 * __GFP_WAIT being passed to INIT_RADIX_TREE().
 */
static int __radix_tree_preload(gfp_t gfp_mask)
{
        struct radix_tree_preload *rtp;
        struct radix_tree_node *node;
        int ret = -ENOMEM;

        preempt_disable();
        rtp = this_cpu_ptr(&radix_tree_preloads);
        while (rtp->nr < ARRAY_SIZE(rtp->nodes)) {
                preempt_enable();
                node = kmem_cache_alloc(radix_tree_node_cachep, gfp_mas
k);
                if (node == NULL)
                        goto out;
                preempt_disable();
                rtp = this_cpu_ptr(&radix_tree_preloads);
                if (rtp->nr < ARRAY_SIZE(rtp->nodes))
                        rtp->nodes[rtp->nr++] = node;
                else
                        kmem_cache_free(radix_tree_node_cachep, node);
        }
        ret = 0;
out:
        return ret;
}
```

The global per-cpu type radix tree preload structure is pre-allocated with empty radix tree nodes and is prepared to fill them up.

- If the allocation fails from the slub cache in the interim, the preemption is exited with enabled. If successful, the preemption exits with disable.
- The size of the system determines the maximum number of radix tree nodes.
    - 32bit=11
    - 64bit=21

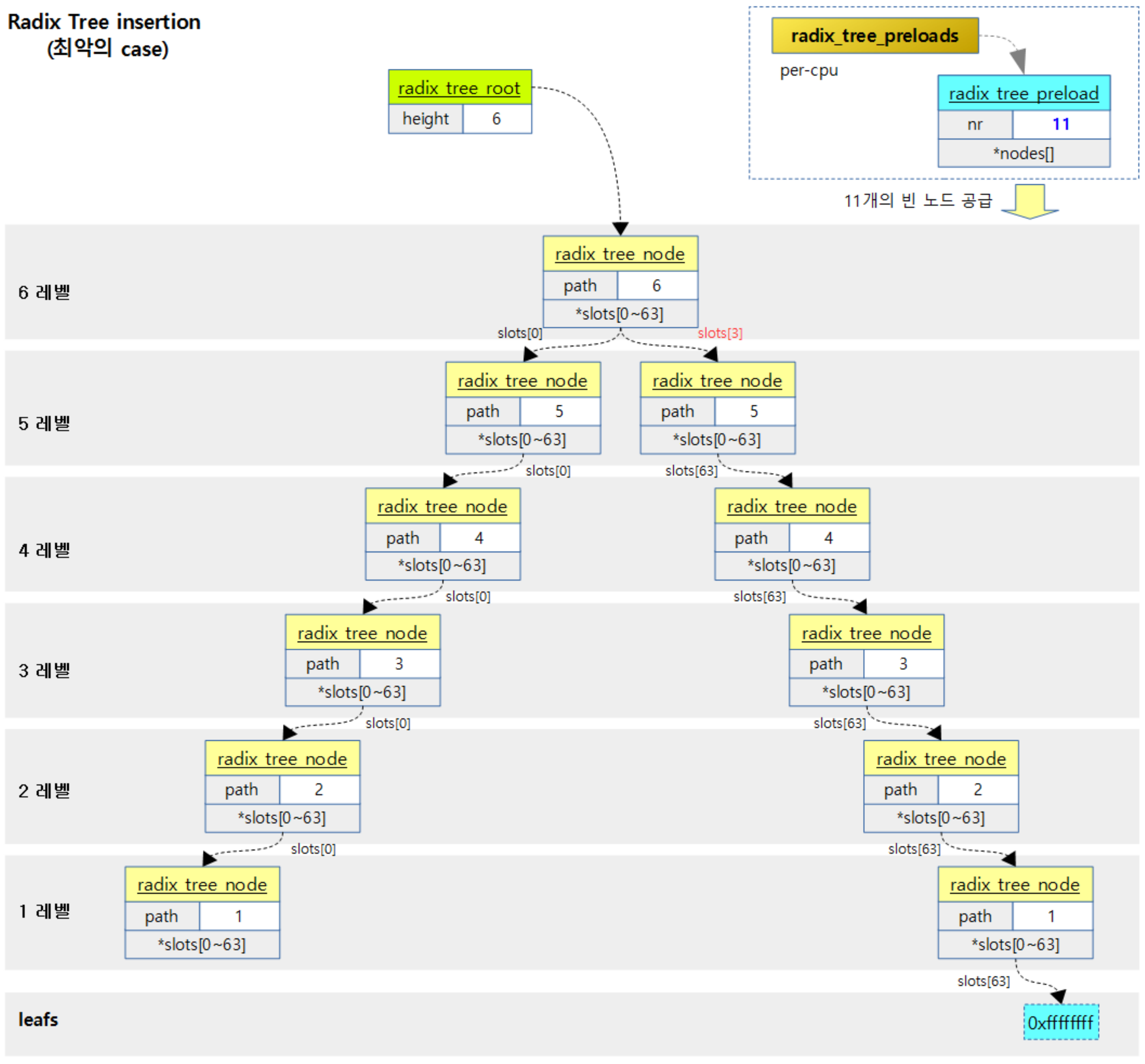
lib/radix-tree.c

```
/*
 * The radix tree is variable-height, so an insert operation not only ha
s
```

```
 * to build the branch to its corresponding item, it also has to build the
 * branch to existing items if the size has to be increased (by
 * radix_tree_extend).
 *
 * The worst case is a zero height tree with just a single item at index 0,
 * and then inserting an item at index ULONG_MAX. This requires 2 new branches
 * of RADIX_TREE_MAX_PATH size to be created, with only the root node shared.
 * Hence:
 */
#define RADIX_TREE_PRELOAD_SIZE (RADIX_TREE_MAX_PATH * 2 - 1)
```

- RADIX_TREE_PRELOAD_SIZE
  - The Radix tree consists of variable steps (level, height), and a single insert requires the allocation of multiple radix tree nodes in the worst case, so the maximum number of available allocation is pre-allocated in the radix tree preload buffer.
    - e.g. worst 32bit system case
      - If you use the index key 0xffffffff with a height of 0, you will need up to 11 RCU tree nodes.
        - We need 6 Radix tree nodes to scale up to 6 levels 6 times.
        - In order to create a radix tree node that matches the index key, you need 6 nodes, one for each of the 1~5 steps except for the 5th step.
  - Therefore, according to the maximum number required as above, the size is determined according to the system as follows.
    - 32bit=11
    - 64bit=21

The following figure shows that if the radix_tree_root is in operation at level 0 and the maximum value of the long is requested as the index key, it will be expanded to level 32, which is the maximum level of a 6-bit system, requiring a total of 11 radix_tree_node.



(http://jake.dothome.co.kr/wp-content/uploads/2016/09/RADIX_TREE_PRELOAD_SIZE-1b.png)

Other Functions

ptr_to_indirect()

lib/radix-tree.c

```
static inline void *ptr_to_indirect(void *ptr)
{
        return (void *)((unsigned long)ptr | RADIX_TREE_INDIRECT_PTR);
}
```

If it doesn't point to a leaf, but to a radix tree node, add a RADIX_TREE_INDIRECT_PTR to the ptr and store it.

indirect_to_ptr()

lib/radix-tree.c

```
static inline void *indirect_to_ptr(void *ptr)
```

```
{
        return (void *)((unsigned long)ptr & ~RADIX_TREE_INDIRECT_PTR);
}
```

Exclude RADIX_TREE_INDIRECT_PTR from the PTR.

root_gfp_mask()

lib/radix-tree.c

```
static inline gfp_t root_gfp_mask(struct radix_tree_root *root)
{
        return root->gfp_mask & __GFP_BITS_MASK;
}
```

Returns the pure gfp_mask of the gfp_mask stored in the root, excluding the tag bits.

tag_set()

lib/radix-tree.c

```
static inline void tag_set(struct radix_tree_node *node, unsigned int tag,
                int offset)
{
        __set_bit(offset, node->tags[tag]);
}
```

Set the offset bit of the tags[tag] of the Radix tree node.

- Maximum tag array is 3

tag_clear()

lib/radix-tree.c

```
static inline void tag_clear(struct radix_tree_node *node, unsigned int tag,
                int offset)
{
        __clear_bit(offset, node->tags[tag]);
}
```

Clear the offset bit of the tags[tag] of the Radix tree node.

- Maximum tag array is 3

tag_get()

lib/radix-tree.c

```
static inline int tag_get(struct radix_tree_node *node, unsigned int tag,
                int offset)
{
        return test_bit(offset, node->tags[tag]);
}
```

Get the offset bit state of the tags [tag] of the Radix tree node.

- Maximum tag array is 3

root_tag_set()

lib/radix-tree.c

```
static inline void root_tag_set(struct radix_tree_root *root, unsigned int tag)
{
        root->gfp_mask |= (__force gfp_t)(1 << (tag + __GFP_BITS_SHIFT));
}
```

Set the requested tag bit at the root of the Radix tree.

root_tag_clear()

lib/radix-tree.c

```
static inline void root_tag_clear(struct radix_tree_root *root, unsigned int tag)
{
        root->gfp_mask &= (__force gfp_t)~(1 << (tag + __GFP_BITS_SHIFT));
}
```

Clear the requested tag bit at the root of the Radix tree.

root_tag_clear_all()

lib/radix-tree.c

```
static inline void root_tag_clear_all(struct radix_tree_root *root)
{
        root->gfp_mask &= __GFP_BITS_MASK;
}
```

Clear the entire tag bit (3 in total) at the root of the Radix tree.

root_tag_get()

lib/radix-tree.c

```
static inline int root_tag_get(struct radix_tree_root *root, unsigned int tag)
{
        return (__force unsigned)root->gfp_mask & (1 << (tag + __GFP_BITS_SHIFT));
}
```

Radix fetches the requested tag bit state of the root of the tree.

Related Constants

lib/radix-tree.c

```
#define RADIX_TREE_MAX_TAGS 3

#ifdef __KERNEL__
#define RADIX_TREE_MAP_SHIFT    (CONFIG_BASE_SMALL ? 4 : 6)
#else
#define RADIX_TREE_MAP_SHIFT    3       /* For more stressful testing */
#endif

#define RADIX_TREE_MAP_SIZE     (1UL << RADIX_TREE_MAP_SHIFT)
#define RADIX_TREE_MAP_MASK     (RADIX_TREE_MAP_SIZE-1)

#define RADIX_TREE_TAG_LONGS    \
        ((RADIX_TREE_MAP_SIZE + BITS_PER_LONG - 1) / BITS_PER_LONG)

#define RADIX_TREE_INDEX_BITS  (8 /* CHAR_BIT */ * sizeof(unsigned lon
g))
#define RADIX_TREE_MAX_PATH (DIV_ROUND_UP(RADIX_TREE_INDEX_BITS, \
                                          RADIX_TREE_MAP_SHIFT))

/* Height component in node->path */
#define RADIX_TREE_HEIGHT_SHIFT (RADIX_TREE_MAX_PATH + 1)
#define RADIX_TREE_HEIGHT_MASK  ((1UL << RADIX_TREE_HEIGHT_SHIFT) - 1)

/* Internally used bits of node->count */
#define RADIX_TREE_COUNT_SHIFT  (RADIX_TREE_MAP_SHIFT + 1)
#define RADIX_TREE_COUNT_MASK   ((1UL << RADIX_TREE_COUNT_SHIFT) - 1)
```

The values below are for the CONFIG_BASE_SMALL kernel options disabled.

- RADIX_TREE_MAX_TAGS
  - Maximum number of tags in the Radix tree
  - 3
- RADIX_TREE_MAP_SHIFT
  - 6
- RADIX_TREE_MAP_SIZE
  - The map size used by the Radix tree
  - 64
- RADIX_TREE_MAP_MASK
  - Map Mask Used by Radix Trees
  - 0x3f
- RADIX_TREE_TAG_LONGS
  - Radix Tree Tag Length
  - 32 bit=2, 64 bit=1
    - 32 long values required for 64 bits on a 2-bit system
- RADIX_TREE_INDEX_BITS
  - Radix tree index bits
    - 32bit=32, 64bit=64
- RADIX_TREE_MAX_PATH
  - Maximum number of passes in the Radix tree
    - 32 bit=6, 64 bit=11
- RADIX_TREE_HEIGHT_SHIFT

    - Number of shifts for Radix tree height
    - 32 bit=7, 64 bit=12
- RADIX_TREE_HEIGHT_MASK
    - Masks for Radix tree height
    - 32 bit=0x3f, 64 bit=0x7ff
- RADIX_TREE_COUNT_SHIFT
    - Radix tree count
    - 7
- RADIX_TREE_COUNT_MASK
    - Mask for Radix tree count
    - 0x3f

Structure

radix_tree_root Struct

include/linux/radix-tree.h

```
/* root tags are stored in gfp_mask, shifted by __GFP_BITS_SHIFT */
struct radix_tree_root {
        unsigned int            height;
        gfp_t                   gfp_mask;
        struct radix_tree_node  __rcu *rnode;
};
```

- height
    - Number of steps managed by the radix tree (0~N)
    - In step 0, there is no Radix tree node, only a slot for one index key 0 times.
    - The maximum number of steps varies depending on the size of the system.
        - 32bit: 6
        - 64bit: 11
    - If the height is 0, it means that no index key has been registered, or only one index key has been registered.
- gfp_mask
    - Whenever radix_tree_node receives an allocation, it requests memory allocation to the slub cache, which contains a gfp_mask to use.
    - In addition, three tag bits are used.
- rnode
    - It points to a node or acts as a slot corresponding to a single 0 index key to store a pointer to a leaf.
        - If you want it to point to the radix_tree_node, which is the highest node, add RADIX_TREE_INDIRECT_PTR bits to it.
        - If only one 0 index key is used, the rnode acts as a single slot and stores the item directly without creating a radix_tree_node.

radix_tree_node Structure

include/linux/radix-tree.h

```
struct radix_tree_node {
        unsigned int    path;   /* Offset in parent & height from the bo
ttom */
        unsigned int    count;
        union {
                struct {
                        /* Used when ascending tree */
                        struct radix_tree_node *parent;
                        /* For tree user */
                        void *private_data;
                };
                /* Used when freeing node */
                struct rcu_head rcu_head;
        };
        /* For tree user */
        struct list_head private_list;
        void __rcu      *slots[RADIX_TREE_MAP_SIZE];
        unsigned long   tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
};
```

- path
    - The level of the current node
    - The lowest level starts at 1.
- count
    - Number of slots in use
    - 0~up to RADIX_TREE_MAP_SIZE(64)
- *parent
    - Point to the parent node.
- *private_data
- rcu_head
    - This is used to delete a node using the RCU.
- private_list
- *slots[]
    - Store up to RADIX_TREE_MAP_SIZE (64) items that point to the next node or corresponding to a leaf.
        - If you want it to point to a node, add a RADIX_TREE_INDIRECT_PTR to it.
- tags[]
    - A bitmap with a total of 3 tags
    - Each bitmap had space to process RADIX_TREE_MAP_SIZE (64) bits.
        - The last of the double arrays of tags[][] is used only for actual declarations, and only as a one-dimensional array of tags[] in the actual processing routines.
    - Whether an item is stored in a slot is expressed using a bitmap.
        - If the tag bit is 1, it means that the slot at that bit location has been used.

radix_tree_preload Struct

lib/radix-tree.c

```
/*
```

```
 * Per-cpu pool of preloaded nodes
 */
struct radix_tree_preload {
        int nr;
        struct radix_tree_node *nodes[RADIX_TREE_PRELOAD_SIZE];
};
```

- No
  - The number of empty radix_tree_node structs currently allocated to the CPU.
- *nodes
  - radix_tree_node assigned pointers are in order in the array.

consultation

- Trees I: Radix trees (https://lwn.net/Articles/175432/) | LWN.net
- radix_tree_init() (http://jake.dothome.co.kr/radix_tree_init) | Qc

6 thoughts to “Radix Tree”

MAX
2018-06-21 16:24 (http://jake.dothome.co.kr/radix-tree/#comment-178403)

Radix treeI refer to it while studying. The explanation and organization are neat. Thanks for the great material.

RESPONSE (/RADIX-TREE/?REPLYTOCOM=178403#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2018-06-22 07:25 (http://jake.dothome.co.kr/radix-tree/#comment-178408)

Thank you for being helpful.

RESPONSE (/RADIX-TREE/?REPLYTOCOM=178408#RESPOND)

SHY
2020-07-14 11:59 (http://jake.dothome.co.kr/radix-tree/#comment-262405)

Thanks for the great material😢

RESPONSE (/RADIX-TREE/?REPLYTOCOM=262405#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2020-07-14 14:50 (http://jake.dothome.co.kr/radix-tree/#comment-262435)

I appreciate it. Have a nice day. ^^

RESPONSE (/RADIX-TREE/?REPLYTOCOM=262435#RESPOND)

HOLY CRAP
2021-07-24 01:11 (http://jake.dothome.co.kr/radix-tree/#comment-305856)

It was so well organized, which was a great help. I appreciate it.

RESPONSE (/RADIX-TREE/?REPLYTOCOM=305856#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-07-24 11:41 (http://jake.dothome.co.kr/radix-tree/#comment-305866)

I appreciate it. Have a great weekend. ^^

RESPONSE (/RADIX-TREE/?REPLYTOCOM=305866#RESPOND)

LEAVE A COMMENT

Your email will not be published. Required fields are marked with *

Comments

name *

email *

Website

WRITE A COMMENT

< context_tracking_init() (http://jake.dothome.co.kr/context_tracking_init/)

radix_tree_init() > (http://jake.dothome.co.kr/radix_tree_init/)