Laboratory 3: „Linux kernel data structures” (two weeks)

dr inż. Arkadiusz Chrobot
dr inż. Karol Tomaszewski

March 16, 2024

Contents

Introduction

This guide contains descriptions of ready-made implementations of data structures that are used in the Linux kernel code and have been made available for other developers creating kernel modules. Chapter 1 contains information, an API description and an example to illustrate the use of bidirectional list implementations. Chapter 2 describes the implementation of the FIFO queue along with its interface and an example of the module that uses it. Chapters 3 and 4 describe the structure, API and examples of using red-black trees and radix trees, respectively. Chapter 5 briefly lists other data structures that are available in the Linux kernel space, but are not described in more detail in this guide. The last chapter contains tasks to be done in the laboratory.

1. Bidirectional lists

The Linux kernel provides a universal implementation of the list, which is used by programmers who create the main kernel code, as well as programmers who create the kernel modules. This eliminates the need to develop subsequent versions of this structure for individual subsystems, reduces code maintenance costs, and makes programmers' work easier.

1.1. API description

The implementation of the list in the Linux kernel is based on a bidirectional cyclic list and is available after including the `linux/list.h` header file in the code. To create a list of structures of a given type, a field of type `struct list_head` should be included **inside of the definition of this resulting type**, which contains two pointers of the same `struct list_head` type named `next` and `prev`. The following macros and functions are available for handling the list:

`void INIT_LIST_HEAD(struct list_head *list)` - a function that is used to initialize list pointers in its single element. It accepts a pointer for the structure containing the list pointers to be initialized.

`LIST_HEAD(name)` - macro that is used to create the main element of the list, which is its „handle". The name of this element is passed to it.

`list_entry(ptr, type, member)` - a macro that allows to get a pointer to the structure **inside of which** the `struct list_head` field is contained. The first call argument is a pointer to a `struct list_head` structure, the second is the name of the structure type containing the field of type `struct list_head`, and the third argument is the name of the `struct list_head` field inside of a given structure.

`void list_add(struct list_head *new, struct list_head *head)` - a function that adds a new list item just `after` its main item. This allows this list to be used as a `stack`. As the first call argument, the function takes a pointer to the `struct list_head` structure contained in a new element of the list, and as a second argument it takes a pointer to the main element of the list.

`void list_add_tail(struct list_head *new, struct list_head *head)` - a function that adds a new list item just `before` its main item. Thanks to this, this list can be used as a FIFO queue. As the first call argument, the function takes a pointer to the `struct list_head` structure contained in a new element of the list, and as a second argument it takes a pointer to the main element of the list.

`void list_del(struct list_head *entry)` - a function that removes an element from the list. This does not mean that the memory for this element is being released! As a call argument it takes a pointer to a `struct list_head` field of the item being removed.

`void list_del_init(struct list_head *entry)` - a function that removes an element from the list and re-initializes its pointers. As a call argument it takes a pointer to a `struct list_head` field of the removed element.

`void list_move(struct list_head *list, struct list_head *head)` - the function removes an element from the list and places it on another list. The first call argument is the pointer to the `struct list_head` field of the item being moved, and the second argument is a pointer to the main element of the target list. The moved element is inserted just `after` the main element of the target list.

`void list_move_tail(struct list_head *list, struct list_head *head)` - the function removes an element from the list and places it on another list. The first call argument is the pointer to the `struct list_head` field of the item being moved, and the second argument is a pointer to the main element of the target list. The moved element is inserted just `before` the main element of the target list.

`int list_empty(const struct list_head *head)` - the function checks if the list is empty. If so, it returns value other than zero and it returns zero otherwise. As a call argument, it takes a pointer to the main element of the list.

`void list_splice(const struct list_head *list, struct list_head *head)` - the function combines two lists with each other. The list whose main element is indicated by the first call argument of the function is attached in full just `after` the main element of the second list, to which the pointer is passed by the second call argument of the function. It can be used to combine lists used as stacks.

`void list_splice_tail(struct list_head *list, struct list_head *head)` - the function combines two lists with each other. The list whose main element is indicated by the first call argument of the function is attached in full just `before` the main element of the second list, to which the pointer is passed by the second call argument of the function. This function can be used to combine lists used as FIFO queues.

`void list_splice_init(struct list_head *list, struct list_head *head)` - this function is used to connect two lists together, just like the `list_splice()` function, but the attached list is flushed and reinitialized.

`list_for_each(pos, head)` - macro that allows to browse the list elements, also called an *iterator*. The first calll argument of this macro is a pointer of type `struct list_head *`, which will indicate subsequent elements of the list in subsequent iterations. The second is the pointer to the main element of the list.

`list_for_each_entry(pos, head, member)` - a macro that allows, to browse the list elements similarly to `list_for_each`, but using the pointer to the actual structure of its elements. This actual structure pointer is passed to the macro as the first argument. The second call argument is the pointer to

the first element of the list, and the last call argument is the name of the `struct list_head` field in the element.

`list_for_each_entry_reverse(pos, head, member)` - the macro accepts the same call arguments as previously described `list_for_each_entry()` macro, but it allows to browse the list elements in the reverse direction.

`list_for_each_safe(pos, n, head)` - the macro allows to browse the list elements and also to safely remove them from the list. The first call argument is a pointer to a list item, which will indicate subsequent items of the list during subsequent iterations. The second call argument is of the same type as the first and is used to temporarily store the address of list elements, and the third argument is a pointer to the main element of the list.

`list_for_each_entry_safe(pos, n, head, member)` - the macro works similarly to `list_for_each_entry`, and it also allows to safely remove elements from the list while browsing. It accepts the same arguments as the previous macro, but has one more argument that is passed to it in the second call argument. This is a pointer to a single list item that is used to temporarily store the addresses of list items.

`list_for_each_entry_safe_reverse(pos, n, head, member)` - a macro that allows to remove items from the list while browsing it similarly to `list_for_each_entry_safe`, but it browses the list in the opposite direction.

There are also other functions and macros related to list support, but they will not be described in this document.

1.2. Example

Listing 1 contains the source code of the module that uses the implementation of the list available in the Linux kernel.

Listing 1: Moduł korzystający z listy

```
#include<linux/module.h>
#include<linux/list.h>
#include<linux/slab.h>
#include<linux/random.h>

struct example_struct
{
        int random_value;
        struct list_head list_element;
};

static LIST_HEAD(head);

static int __init listmod_init(void)
{
        struct example_struct *element;
        struct list_head *entry;
        u8 i;

        for(i=0;i<4;i++) {
                element = (struct example_struct *)
                        kmalloc(sizeof(struct example_struct),GFP_KERNEL);
                if(!IS_ERR(element)) {
                        get_random_bytes((int *)&element->random_value,
                                         sizeof(element->random_value));
                        INIT_LIST_HEAD(&element->list_element);
                        list_add_tail(&element->list_element,&head);
```

```
28                  }
29          }
30          list_for_each(entry, &head) {
31                  element = list_entry(entry, struct example_struct, list_element);
32                  pr_notice("Element's value: %u\n",element->random_value);
33          }
34
35          return 0;
36  }
37
38  static void __exit listmod_exit(void)
39  {
40          struct example_struct *element, *next;
41
42           list_for_each_entry_safe(element, next, &head, list_element) {
43                   list_del(&element->list_element);
44                   pr_notice("Element's value: %u\n",element->random_value);
45                   kfree(element);
46           }
47  }
48
49  module_init(listmod_init);
50  module_exit(listmod_exit);
51
52  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
53  MODULE_LICENSE("GPL");
54  MODULE_DESCRIPTION("An exemplary kernel module that demonstrates the usage of a kernel list.");
55  MODULE_VERSION("1.0");
```

In the module source code from Listing 1 there are four header files attached. The first of them (line 1) contains the definitions of functions and macros for handling modules. The second header file (line 2) defines functions and macros for handling lists. The third file (line 3) contains the definition of the `kmalloc()` function, which will be used for dynamic allocation of memory to list elements, and the fourth header file (line 4) contains the definition of the `get_random_bytes()` function, a more detailed description of which will be provided later. It is used to get random values from the kernel random number generator[1]. Lines 6-10 define the type of the sample structure of list items. It contains a data field (`random_value`) and a field of type `struct list_head` with list pointers so that items can be combined into a list. In line 12 the main element of the list is declared. A list is created in the module constructor, and then the contents of the `random_value` field of its elements are printed to the kernel buffer. In line 16 of this function, a pointer is declared for a single element of the list. In line 17 there is a pointer declaration for a structure of type `struct list_head`, and in line 18 there is a declaration for the `for` loop counter. In this loop (lines 20-29) memory is allocated to four elements of the list (lines 21-22). If the allocation was successful, the `random_value` field of each such element is initialized with a random value taken from the random number generator using the `get_random_bytes()` function (line 24). This function returns nothing, but takes two call arguments to invoke. The first argument is the address of the variable to which the value should be stored, and the second argument is the number of bytes to be read from the generator. In case of the described module, it is equal to the size of the `random_value` field. After initializing this field, the `list_element` field is initialized, and then the entire element is added to the list, whose main element is `head`. On lines 30-33, the list is browsed using the `list_for_each`, and the values of the `random_value` fields of each of its elements are printed into the kernel buffer. Please note line 31, in which the pointer to the list item is obtained using the `list_entry` macro. This part of the initialization function also illustrates how to properly use the `list_for_each` macro. In the module destructor, the list is browsed using the `list_for_each_entry_safe` macro. While browsing, subsequent items from the list are removed, the contents of their `random_value` field goes to the kernel buffer, and the memory allocated to the item is released using the `kfree()` function. Please note the `element` and `next` pointer declarations in line 40 of the module and how they are passed to the macro.

[1] Formally it is a pseudo-random number generator, but it meets the security requirements for cryptographic applications, hence it can be called a random number generator to distinguish it from typical pseudo-random number generators.

2. FIFO queues

Another data structure that has been universally implemented in the Linux kernel is the `fifo` queue. Although this structure can be also obtained using the bidirectional cyclic list described earlier, but the independent dedicated implementation of this queue is more effective, especially in the straight `producer-consumer` types of problems. The implementation of the FIFO queue in the Linux kernel is based on a cyclically indexed array. It allows to store and read items of variable size.

2.1. API description

To use the universal FIFO queue implementation, the `linux/kfifo.h` header file have to be enabled and a `struct kfifo` variable have to be defined. The following macros are designed to handle this variable:

`kfifo_alloc(fifo, size, gfp_mask)` - a macro that dynamically allocates memory to the FIFO queue. The first call argument is a pointer to a structure of type `struct kfifo`. The second is the size of all queue elements, which must be expressed by the power of two. The third argument is the assignment type marker. The macro returns zero if the allocation is successful, or an error code in case of an allocation failure.

`kfifo_init(fifo, buffer, size)` - the macro allows to initiate a `fifo` queue in the buffer to which memory was previously allocated. It is used instead of `kfifo_alloc`. The first call argument of this macro is a pointer to a structure of type `struct kfifo`, the second argument is a pointer of type `void *` to a buffer, and the third is the buffer size, which must be expressed by the power of two. The macro returns zero if the queue initialization has successfully completed, or an error code otherwise.

`DECLARE_KFIFO(fifo, type, size)` - a macro that allows to statically, i.e. during compilation, create a FIFO queue of a specific size. The first call argument of this macro is the name of the queue, the second is the name of the type of a single element, and the third argument is the size, which must be expressed by the power of two.

`INIT_KFIFO(fifo)` - macro that is used to initialize the queue created with `DECLARE_FIFO`. The only call argument is the name of this FIFO queue.

`kfifo_in(fifo, buf, n)` - a macro that allows to add an item (or items) to the queue. The first call argument is a pointer to the FIFO queue (of type `struct kfifo`), the second argument is a pointer to the inserted element, and the third argument is the size of the element. If the operation is successful, the macro will return the size of the added element.

`kfifo_out(fifo, buf, n)` - a macro that allows to remove an item (or items) from the queue. The first call argument is a pointer to the FIFO queue, the second argument is a pointer to the variable where the value of the element is to be stored, and the third argument is the size of the removed element. In case of success the macro will return the size of the removed item.

`kfifo_peek(fifo, val)` - the macro allows to read the value of an element (or elements) without removing it from the queue. As the first call argument it takes the pointer to the FIFO queue and the second argument is a pointer to the variable, where the value of this element is to be stored. In case of success the macro returns the size of the read element.

`kfifo_size(fifo)` - the macro returns the size of the FIFO queue. As the call argument it takes a pointer to this queue.

`kfifo_len(fifo)` - the macro returns the amount of space used in the queue, to which the pointer is passed as the call argument.

`kfifo_avail(fifo)` - the macro returns the amount of free space in the queue to which the pointer is passed as the call argument.

`kfifo_is_empty(fifo)` - the macro returns a non-zero value (true) if the queue to which the pointer is passed as the call argument is empty.

`kfifo_is_full(fifo)` - the macro returns a non-zero value (true) if the queue to which the pointer is passed as the call argument is full.

`kfifo_reset(fifo)` - deletes all contents of the FIFO queue, to which the pointer is passed the call argument.

`kfifo_free(fifo)` - the macro releases a FIFO queue memory that was previously allocated by the `kfifo_alloc` macro.

There are also other macros related to handling the FIFO queue that will not be described in this document.

2.2. Example

Listing 2 contains the source code of the module, that creates a FIFO queue and fills it with random numbers.

Listing 2: Module using the FIFO queue

```
#include<linux/module.h>
#include<linux/random.h>
#include<linux/kfifo.h>

#define NUMBER_OF_ELEMENTS 8

static struct kfifo fifo_queue;

static int __init fifomod_init(void)
{
        int returned_value;
        u32 random_value;
        unsigned int returned_size;

        returned_value =
                kfifo_alloc(&fifo_queue,NUMBER_OF_ELEMENTS*sizeof(random_value),GFP_KERNEL);
        if(returned_value) {
                pr_alert("Error allocating kfifo!\n");
                return -ENOMEM;
        }
        while(!kfifo_is_full(&fifo_queue)) {
                get_random_bytes(&random_value,sizeof(random_value));
                random_value %= 100;
                returned_size = kfifo_in(&fifo_queue,&random_value,sizeof(random_value));
                if(returned_size!=sizeof(random_value))
                        pr_alert("Enqueue error\n");
        }
        returned_size = kfifo_out_peek(&fifo_queue,&random_value,sizeof(random_value));
        if(returned_size!=sizeof(returned_size))
                pr_alert("Error peeking element form the queue!\n");
        pr_notice("Value of the first element in the queue: %u\n",random_value);

        return 0;
}

static void __exit fifomod_exit(void)
{
        int returned_size;
```

```
        u32 value;

        while(!kfifo_is_empty(&fifo_queue)) {
                returned_size = kfifo_out(&fifo_queue,&value,sizeof(value));
                if(returned_size!=sizeof(value))
                        pr_alert("Dequeue error!\n");
                pr_notice("Value from the queue: %u\n",value);
        }

        kfifo_free(&fifo_queue);
}

module_init(fifomod_init);
module_exit(fifomod_exit);

MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("An exemplary module demonstrating the usage of a kernel FIFO queue.");
MODULE_VERSION("1.0");
```

In the module's initialization function, a memory is allocated to a queue that can hold eight numbers of type `int`, and then a single- and a double-digit numbers are random and stored to this queue in a `while` loop, which is executed as long as there is space for new elements. Finally, the value of the number at the top of the queue is read and placed in the kernel buffer. The function checks the correctness of each queue operation. In the cleanup function of the module, in the `while` loop, subsequent elements are removed from the queue head and their value is placed in the kernel buffer. The correctness of the removal operation is checked every time. Looping ends when the FIFO queue is empty. Then the memory allocated to the queue is released. Other examples of using a queue can be found in the `sample/kfifo` subdirectory in the kernel source code directory.

3. Red-black trees

Red-black trees are the balanced BST trees. Balancing is achieved by giving each tree node an additional attribute (a label), which is called a `color`, and ensuring that each time a new element is inserted into the tree or an existing element is deleted, the following rules will apply[2]:

1. The root is always labelled as black.
2. Each node is labelled as black or red.
3. Descendants of the red node must always be labelled as black.
4. The leaves of the tree are always black.
5. All straight paths from the given node to any leaf subject to it contain the same number of black nodes.

If the tree does not comply with given rules after its modification, then by rotating certain sub-trees and/or changing the colors of the nodes, their compliance is restored.

3.1. API description

Because the red-black trees should be implemented universally in the system kernel, Linux kernel programmers provide only the means to transform any structures into red-black tree nodes and to balance it. Inserting and searching for elements in the tree, similar to those used in BST trees, must be implemented

[2]Different sources provide different sets of rules. The set of rules included in this guide comes - besides the rule No. 1, which is redundant to rule No. 4 - from the book *Introduction to Algorithms* by T. H. Cormen, Ch. E. Leiserson and R. L. Rivest, 1998 edition. It is fully sufficient for the proper functioning of such a structure.

by the application programmers[3]. Red-black trees can be used after including the `linux/rbtree.h` header file. The basic structure for a red-black tree is a `struct rb_node`. It should be declared as a field in the actual structure that is to be placed in the red-black tree, in a similar way as we placed a `struct list_head` structure in the actual list elements. The structure placed in the described tree must also have a field that acts as a key, i.e. the attribute by which the tree nodes will be ordered. The second important structure is `struct rb_root`, which is the root of the red-black tree. The following macros and functions have been defined to handle suchs trees:

`RB_ROOT` - the macro initializes the root of the red-black tree. Its value should be assigned to a variable of type `struct rb_root`.

`rb_entry(ptr, type, member)` - a macro that allows to get a pointer to the structure in which field of type `struct rb_node` is contained. The first call argument is a pointer to a `struct rb_node` structure, the second argument is the name of a structure containing the `struct rb_node` field, and the third argument is a `struct rb_node` field name.

`struct rb_node *rb_first(const struct rb_root *)` - function used to traverse (iterate) red-black tree nodes in non-descending order. As a call argument, it takes a pointer to the tree root (a variable of type `struct rb_root *`), and returns a pointer to a `struct rb_node` field contained in the first (leftmost) tree node.

`struct rb_node *rb_next(const struct rb_node *)` - function called after `rb_first()`, in a loop. As a call argument, it takes a pointer to a `struct rb_node` field of the current red-black tree element, and returns a pointer to a field of the same type contained in the next node to visit, in a non-descending order. If such node does not exist, the function returns `NULL`.

`struct rb_node *rb_last(const struct rb_root *)` - a function that works similarly to `rb_first()`, but allows to start going through the red-black tree nodes in non-ascending order. Returns a pointer to the `struct rb_node` field contained in the last (rightmost) element of the tree.

`struct rb_node *rb_prev(const struct rb_node *)` - a function that is equivalent to `rb_next()` to traverse tree nodes in non-ascending order. It is used in the same way as the function mentioned.

`void rb_link_node(struct rb_node *node, struct rb_node *parent, struct rb_node **rb_link)` - a function that allows to attach a new node to a red-black tree. It takes three call arguments and does not return anything. The first call argument is a pointer to a `struct rb_node` field of the new node. The second argument is a pointer to a field of the same type located in the parent node that will point to that new node. The third argument is a pointer to one of the two pointer fields of a `struct rb_node` structure of the parent node (`rb_right` or `rb_left`) to which the new node is to be linked.

`void rb_insert_color(struct rb_node *, struct rb_root *)` - function that checks if the red-black tree needs balancing and performs this operation if necessary. It is called immediately after `rb_link_node()`. It takes two call arguments: a pointer to the the newly inserted tree node and a pointer to the tree root.

`void rb_replace_node(struct rb_node *victim, struct rb_node *new, struct rb_root *root)` - this function replaces the tree node indicated by the first call argument with the node indicated by the second call argument. The third argument is a pointer to the tree root. **Both, the replaced node and the replacement node, should have the same key, i.e. the value by which they are ordered in the red-black tree.** This function does not perform balancing operations, so it should not be used to replace nodes with other nodes with different keys.

`void rb_erase(struct rb_node *, struct rb_root *)` - a function that removes a node pointed by its first call argument from the red-black tree. It does not release the memory allocated to this node. The second call argument is a pointer to the tree root.

As with previous structures, this API description is not complete. It lists only the most important functions and macros for handling red-black trees.

[3]Information and examples of how to do this can be found in the kernel documentation, in the `linux/Documentation/rbtree.txt` file.

3.2. Example

Listing 3 contains the source code for an example kernel module that uses a red-black tree to store natural numbers. These numbers are both the key and the only value of the tree nodes.

Listing 3: Module using red-black tree

```
#include<linux/module.h>
#include<linux/rbtree.h>
#include<linux/slab.h>

struct example_struct
{
        int data;
        struct rb_node node;
};

static struct rb_root root = RB_ROOT;

static struct example_struct *find_node(struct rb_root *root, int number)
{
        struct rb_node *node = root->rb_node;

        while(node) {
                struct example_struct *current_data = rb_entry(node, struct example_struct, node);

                if(number<current_data->data)
                        node = node->rb_left;
                else if(number>current_data->data)
                        node = node->rb_right;
                else
                        return current_data;
        }
        return NULL;
}

static bool insert_node(struct rb_root *root, struct example_struct *node)
{
        struct rb_node **new_node = &(root->rb_node), *parent = NULL;

        while(*new_node) {
                struct example_struct *this = rb_entry(*new_node, struct example_struct, node);

                parent = *new_node;
                if(node->data<this->data)
                        new_node = &((*new_node)->rb_left);
                else if(node->data>this->data)
                        new_node = &((*new_node)->rb_right);
                else return false;
        }

        rb_link_node(&node->node,parent,new_node);
        rb_insert_color(&node->node,root);
        return true;
}

static int __init rbtreemod_init(void)
{
        int i;
        struct example_struct *node = NULL;

```

```
55          for(i=0;i<5;i++) {
56                  node = (struct example_struct *)kmalloc(sizeof(struct example_struct), GFP_KERNEL);
57                  if(!IS_ERR(node)) {
58                          node->data = i;
59                          if(!insert_node(&root,node))
60                                  pr_alert("Error inserting new node!\n");
61
62                  } else
63                          pr_alert("Error allocating memory for new node: %ld\n",PTR_ERR(node));
64
65          }
66
67          for(i=0;i<5;i++) {
68                  node = find_node(&root,i);
69                  if(node)
70                          pr_notice("Value of the node: %d\n",node->data);
71                  else
72                          pr_alert("Error retrieving node from red-black tree!\n");
73          }
74
75          return 0;
76  }
77
78  static void __exit rbtreemod_exit(void)
79  {
80          int i;
81          struct example_struct *node = NULL;
82
83          for(i=0;i<5;i++) {
84                  node = find_node(&root,i);
85                  if(node) {
86                          rb_erase(&node->node, &root);
87                          kfree(node);
88                  } else
89                          pr_alert("Error retrieving node from red-black tree!\n");
90          }
91  }
92
93  module_init(rbtreemod_init);
94  module_exit(rbtreemod_exit);
95  MODULE_LICENSE("GPL");
96  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
97  MODULE_DESCRIPTION("A module that demonstrated the usage of a kernel red-black tree implementation.");
98  MODULE_VERSION("1.0");
```

Lines 5-9 define the structural type describing the structure of each node of the red-black tree. There is a field named `data` for numbers that will be the keys and values of nodes in this tree and a field named `node` of type `struct rb_node`, which contains pointers `rb_right` and `rb_left`. Similarly as with lists, this field allows to join nodes into a tree structure. In line 11, the variable named `root` has been declared and initialized, that will be the root of the red-black tree. The module code also defines two functions named `find_node()` and `insert_node()`. The first of them searches for a node in the tree with the key value given by its parameter named `number`. Due to the limited size of the kernel stack (typically only 2 pages of memory), the use of recursion by kernel functions is not recommended. Therefore, the `find_node()` function searches the tree using a `while` loop. In line 15 in the local pointer named `node` the address of the `rb_node` field is stored. This is the only field in the `struct rb_root` that contains the address of the tree root. Next, in the `while` loop, with the use of `rb_entry` macro the address of the `struct example_struct` structure is obtained. This structure has an embedded field pointed by the node pointer (line 18). It is simply a tree node. If the value stored in the `number` field of this node is greater than that contained in the number parameter, then the `node` pointer is set to the left child of

that node (if it exists), and if smaller, the `node` pointer is set to the right child (if it exists). If both values are the same, the current node address is returned (line 25). If the value given by the `number` parameter is not in the tree, then the while loop will end and `NULL` will be returned (line 27). The `insert_node()` function is more complicated. It is responsible for the placement of a new node in the red-black tree. It will return `true` if it succeeds and `false` otherwise. This function has two call parameters: a pointer to the root of the tree and a pointer to the element that should be inserted into it. In line 32, two pointers are declared and initialized. The first pointer (`new_node`) is a double-purpose pointer that will initially contain the root address of the tree. It will be used to „move" around the tree and will eventually contain the address of the tree node to which the new node will be attached. The second pointer (`parent`) will point to the parent of the node whose address will be in `new_node`, so its value is initially `NULL` (the root has no parent). In the `while` loop, in line 35 the address of the `struct example_struct` structure is obtained and stored in the `this` pointer. The mentioned `struct example_struct` wraps around the `node` field pointed by the `*new_node` pointer. In line 37 the address of the `node` field pointed by the `*new_node` pointer is stored in the `parent` pointer. Then the value of the `number` field contained in the new node is compared to the value of the same field in the currently visited node. If it is smaller, the `new_node` pointer is „set" to the left child of the current node (if it exists), and if the value is larger, then `new_node` pointer is „set" to the right child of that node. If those values are equal, the function ends and returns `false` because the implementation of the red-black tree that is implemented in the Linux kernel assumes that the keys in the tree do not repeat. If a node is found whose corresponding child does not exist, then the execution of the `while` loop is terminated and the new node becomes that child, i.e. it is inserted into the tree with `rb_link_node()` function and the `rb_insert_color()` function is called in case the tree requires to be balanced. After that, the `insert_node()` function terminates and returns `true`. In the module's initialization function, 5 nodes are created, which receive the values of the `number` field as follows: `0`, `1`, `2`, `3`, `4`. They are then inserted into the tree by calling the `insert_node()` function. Then, in the next loop, these nodes are searched and their value is placed in the kernel buffer. In the cleanup function, nodes are searched by key, removed from the tree, and memory allocated for them is released.

4. Radix trees

Radix trees are a variation of *trie* trees that are designed to save memory space. In Linux, these structures allow to associate a key, which is an integer, with a pointer to a variable containing the correct value, i.e. create key-value pairs[4]. In addition, the Linux implementation allows to assign a labels to the elements of the tree. Each node in the tree usually contains an array of 64 pointers. Each of these elements is indexed by a number of type `long int`. Typically, however, amount of keys in radix trees is larger than 64, so more than one node is needed to remember the values associated with it. When searching the tree, the six oldest bits in the key are used to find the appropriate pointer in the table at the root of the tree. The next six bits is indexing the corresponding pointer in the table of the node indicated by the previous node, and so on. This happens until the six youngest bits allow locating the pointer in the table in the leaf of the tree that indicates the correct data.

4.1. API description

To use the radix tree in the kernel module, the `radix-tree.h` header file have to be included in the source code. It includes, among the others, the following macros and functions related to handling of such trees:

`RADIX_TREE` - a macro that allows to create a variable of type `struct radix_tree_root` that is the root of a radix tree. It takes two call arguments: the name of this variable and the assignment type marker.

`int radix_tree_insert(struct radix_tree_root *root, unsigned long index, void *entry)` - function that allows to insert a new value into the radix tree. It takes three call arguments: a pointer to the root of the tree, a key that specifies the location of this value in the tree, and a pointer of type `void *`

[4]More information about the implementation of trees in the Linux kernel can be found in the article entitled "Trees I: Radix trees" by Johnathan Corbet: `https://lwn.net/Articles/175432/`

to a variable containing this value. If the insertion is successful, the function returns `0`. A non-zero value is returned otherwise.

`void *radix_tree_lookup(struct radix_tree_root *, unsigned long)` - the function allows searching for a value in the raddix tree based on the given key. It takes two call arguments: a tree root pointer and a key. Returns a pointer to a variable containing the search value or `NULL` if no such value exists.

`void *radix_tree_delete(struct radix_tree_root *, unsigned long)` - the function removes the value associated with a given key from the tree, returning a pointer to a variable containing this value (of type `void *`). As a call argument, it takes a pointer to the root of the radix tree and the key associated with the search value. If the value being deleted does not exist, the function returns `NULL`.

The API description for the radix trees in this document is limited only to the functions and macros used in the example module.

4.2. Example

Listing 4 contains the source code of an example module that uses a radix tree to store key-value pairs, where the values are random numbers of type `int`.

Listing 4: Module using radix tree

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

#define UPPER_LIMIT 65535

static RADIX_TREE(root,GFP_KERNEL);

static int __init radixtreemod_init(void)
{
        long int i;
        int *random_number=NULL, *data = NULL;

        for(i=1;i<UPPER_LIMIT;i<<=3) {
                random_number = (int *)kmalloc(sizeof(int), GFP_KERNEL);
                if(IS_ERR(random_number)) {
                        pr_alert("Error allocating memory: %ld\n",PTR_ERR(random_number));
                        return -ENOMEM;
                }
                get_random_bytes(random_number,sizeof(int));
                *random_number%=2017;
                if(radix_tree_insert(&root,i,(void *)random_number))
                        pr_alert("Error inserting item to radix tree!\n");
        }

        for(i=1;i<UPPER_LIMIT;i<<=3) {
                data = (int *)radix_tree_lookup(&root,i);
                if(data)
                        pr_notice("Value retrieved from radix tree: %d for index: %ld\n",*data,i);
                else
                        pr_alert("Error retrieving data from tree!\n");
        }

        return 0;
}
```

```
37
38 static void __exit radixtreemod_exit(void)
39 {
40         int *data=NULL;
41         long int i;
42
43         for(i=1;i<UPPER_LIMIT;i<<=3) {
44                 data  = (int *)radix_tree_delete(&root,i);
45                 if(data) {
46                         pr_notice("Value retrieved from radix tree: %d for index: %ld\n",*data,i);
47                         kfree(data);
48                 } else
49                         pr_alert("Error retrieving data from tree!\n");
50         }
51 }
52
53 module_init(radixtreemod_init);
54 module_exit(radixtreemod_exit);
55
56 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
57 MODULE_LICENSE("GPL");
58 MODULE_DESCRIPTION("A module that demonstrates the usage of kernel radix tree implementation.");
59 MODULE_VERSION("1.0");
```

The constant variable `UPPER_LIMIT` specifies the upper range of the key value. Subsequent keys for the value will be determined by multiplying their predecessors by eight, thanks to which the tree will contain more than one node. In line 8 of the module a variable named `root` is created, which will be a node of the radix tree. The second argument of the `RADIX_TREE` macro defines how memory is allocated to subsequent nodes of this tree. In the module's initialization function, in the first `for` loop, dynamic variables are created into which random numbers are stored. These variables are then inserted into the radix tree with the keys specified by the loop counter value. In the second `for` loop, the values of these variables are searched for by the key in the radix tree and printed, together with the keys, to the kernel buffer. In the module cleanup function, dynamic variables that store values are removed according to the keys in the radix tree. Their values, along with the keys, are printed in the kernel buffer, and then the memory allocated to them is released.

5. Other data structures

The Linux kernel also has implementations of other data structures, such as *maps*, which in Linux are called *idr* (from *Integer ID management*), or cyclic unidirectional lists that are used in the kernel to create hash tables. However, they will not be described in this guide. A list of processes that collects descriptors of all processes working in Linux system can be considered an example of the application of the structures described earlier. Each process descriptor (of type `struct task_struct`) has a field of type `struct list_head`, which allows it to be included in such a list. In addition, Linux developers have defined three macros to handle this list. The `for_each_process` macro allows to iterate through the entire list of processes. As a call argument, it takes a pointer to the process descriptor, which „moves" through the list. The `next_task` macro takes the pointer to the process descriptor in the list as a call argument and returns the pointer to the next process descriptor. Finally the `next_prev` macro takes the same argument as `next_task` macro and returns the pointer to the previous process descriptor in the list. Listing 5 contains the source code of the module whose init and cleanup functions review the list of process descriptors and print out the name, `pid` and status of each process.

Listing 5: Module reviewing the list of processes

```
1 #include<linux/module.h>
2 #include<linux/sched/signal.h>
3
```

```c
static int __init init_tasklist(void)
{
        struct task_struct *p;

        for_each_process(p) {
                pr_info("Process name is: %s, and its pid is: %i. ",p->comm, task_pid_nr(p));
                if(p->state==TASK_RUNNING)
                        pr_info("Process state is TASK_RUNNING\n");
                if(p->state==TASK_INTERRUPTIBLE)
                        pr_info("Process state is TASK_INTERRUPTIBLE\n");
                if(p->state==TASK_IDLE)
                        pr_info("Process state is TASK_IDLE\n");
                if(p->state==TASK_UNINTERRUPTIBLE)
                        pr_info("Process state is TASK_UNINTERRUPTIBLE\n");
                if(p->state==TASK_STOPPED)
                        pr_info("Proces state is TASK_STOPPED\n");
        }
        return 0;
}

static void __exit exit_tasklist(void) {

        struct task_struct *p;

        for_each_process(p) {
                pr_info("Process name is: %s, and its pid is: %i. ",p->comm, task_pid_nr(p));
                if(p->state==TASK_RUNNING)
                        pr_info("Process state is TASK_RUNNING.\n");
                if(p->state==TASK_INTERRUPTIBLE)
                        pr_info("Process state is TASK_INTERRUPTIBLE.\n");
                if(p->state==TASK_IDLE)
                        pr_info("Process state is TASK_IDLE\n");
                if(p->state==TASK_UNINTERRUPTIBLE)
                        pr_info("Process state is TASK_UNINTERRUPTIBLE.\n");
                if(p->state==TASK_STOPPED)
                        pr_info("Proces state is TASK_STOPPED.\n");
        }
}

module_init(init_tasklist);
module_exit(exit_tasklist);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple module that prints name, pid and state of every active process.");
MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
MODULE_VERSION("1.0");
```

Please notice, that the `pid` member of the descriptor is not accessed directly, but with the use of the `task_pid_nr()` function. This is a way recommended by Linux kernel programmers. Also, the module displays information about a process state, which was not discussed in the lecture. This is the `TASK_IDLE` state, that is similar to `TASK_UNINTERRUPTIBLE`, but statistics of processes that are in that state don't contribute to calculations of the system average load.

Exercises

1. `[7 points]` Write a program for a user space that will run until the user ends it by entering the `q` character. Write a module that finds the descriptor of this program in the process list and writes the information you selected from this descriptor to the kernel buffer. The descriptor structure type definition is in the `linux/sched.h` header file.

2. `[3 points]` Write a module in which you will use the implementation of the list to create a stack. Provide the values for stack elements through module parameters.

3. `[3 points]` Write a module in which you will use the list implementation to create a FIFO queue. Provide the values for queue elements through module parameters.

4. `[5 points]` Write a module in which you will use the FIFO queue implementation to store strings of varying lengths. Provide these strings through the module parameters.

5. `[5 points]` Repeat the command from task 4, but use the red-black tree instead of the FIFO queue.

6. `[7 points]` Repeat the command from task 5, but use the radix tree instead of the red-black tree.

7. `[5 points]` Use the red-black tree in the module to store structures that will have three fields. The first field will store the key that is a natural number, the second field will store the random lowercase letter, and the third will be of type `struct rb_node`. Print the values from this tree in the kernel buffer.

8. `[7 points]` Expand the module 3.2 so that more nodes are added to the red-black tree. Use the functions described in the chapter on this structure to place the values of these nodes in the buffer in a non-descending order. Also check how the function that replaces tree nodes works.