Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[Lab 07] Kernel Data Structures

DIAG
Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v1

Linux Kernel Design Patterns

The kernel has to manage a significant amount of different data structures. Many objects are complex:

- variable size
- groups of other objects (collections)
- changing frequently over time

Performance and efficiency is fundamental. We need abstract data types: how to do that in C?

Abstract Data Types

Encapsulate the entire implementation of a data structure. Provide only a well-defined interface to manipulate objects/collections. Optimizations in the data structure implementation is directly spread across the whole source.

The data structures that we will see are implements these concepts.

Linux Kernel Design Patterns

If you want to deepen the design patterns for the linux kernel you can follow these articles:

- Part 1 https://lwn.net/Articles/336224/
- Part 2 https://lwn.net/Articles/336255/
- Part 3 https://lwn.net/Articles/336262/

Data Structures

The available data structures in the Linux kernel are:

- Linked Lists
- Queues
- Maps
- Trees
  - Binary Tree
  - Red-Black Tree
  - Radix Tree

Linked Lists

Linked Lists

The linked list is the simplest and the most common data structure in the linux Kernel.

Suppose that you want to create a linked list on your own. You will probably use the following approach.

```
struct fox {
        unsigned long  tail_length;  /* length in centimeters of tail */
        unsigned long  weight;       /* weight in kilograms */
        bool           is_fantastic; /* is this fox fantastic? */
        struct fox     *next;        /* next fox in linked list */
        struct fox     *prev;        /* previous fox in linked list */
};
```

Linked lists

For following the design patterns that we introduced, the Linux kernel follows a unique approach: instead of turning the entire struct intro a linked list, **it embeds a linked list node in the structure**.

For making you structure a list item you need to include the `list_head` struct in your one, from `include/list.h`.

```
struct list_head {
    struct list_head *next
    struct list_head *prev;
};
```

https://elixir.bootlin.com/linux/v5.11/source/tools/include/linux/types.h#L69

Linked Lists

For instance

```
struct my_list_item
{
        int my_field1;
        int my_field2;
        struct list_head list;
}
```

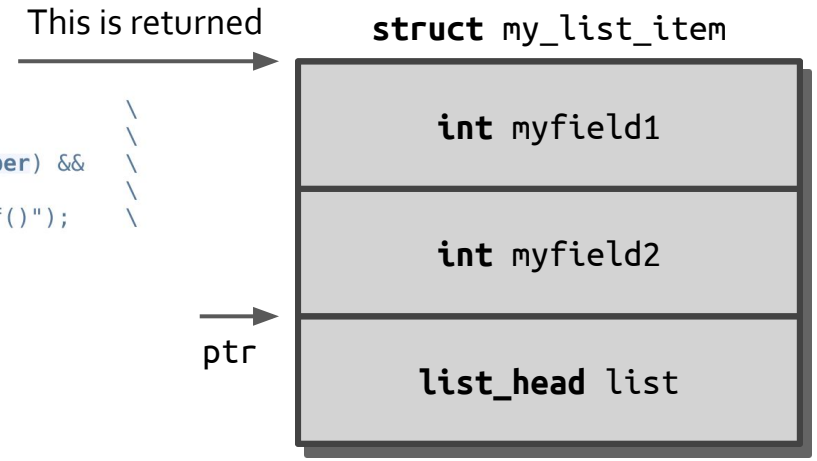
Now you can use all of the APIs available for managing lists, but all the methods are **generic** since they only accept `list_head` structures.

container_of

Using container of we can retrieve the parent structure containing any given member variable, only by computing offsets.

```
692  #define container_of(ptr, type, member) ({                          \
693          void *__mptr = (void *)(ptr);                               \
694          BUILD_BUG_ON_MSG(!__same_type(*(ptr), ((type *)0)->member) &&   \
695                           !__same_type(*(ptr), void),                \
696                           "pointer type mismatch in container_of()");    \
697          ((type *)(__mptr - offsetof(type, member))); })
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/kernel.h#L692



Head of list

For initializing the list you need to use the `INIT_LIST_HEAD` macro.

```
struct list_head todo_list;
INIT_LIST_HEAD(&todo_list);
```

If the list variable is global, then it must be initialized at compile time.

```
LIST_HEAD(todo_list);
```

API

```
list_add(struct list_head *new, struct list_head *head);
list_add_tail(struct list_head *new, struct list_head *head);
list_del(struct list_head *entry);
list_del_init(struct list_head *entry); // To later relink
list_move(struct list_head *entry, struct list_head *head);
list_move_tail(struct list_head *entry, struct list_head *head);
list_empty(struct list_head *head); // Non-zero if empty
```

How to iterate over the list?

API

Traversing

For traversing a list you can use `list_for_each(p, list)` or ever `list_for_each_entry_reverse` or `list_for_each_entry_safe` if you want to eliminate items while traversing the list. Many other functions are available in `include/list.h`.

```
struct list_head *p;
struct fox *f;
list_for_each(p, &fox_list) {
    /* f points to the structure in which the list is embedded */
    f = container_of(p, struct fox, list);
    /* or  */
    f = list_item(p, struct fox, list);
}
```

Lock-less Linked Lists

There is also a special kind of NULL terminated singly linked lists which are lockless in particular cases. If operations are carried out accessing only the single next pointer, RMW instructions allow concurrent access with no locking. They are based on the atomic asm instruction `cmpxchg` (classic compare-and-swap).

```
struct llist_head {
     struct llist_node *first;
};

struct llist_node {
     struct llist_node *next;
};
```

```
*            |   add    | del_first |  del_all
* add        |    -     |     -     |     -
* del_first  |          |     L     |     L
* del_all    |          |           |     -

- = locking not needed
L = locking needed
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/llist.h

Hash Tables

Hash Tables

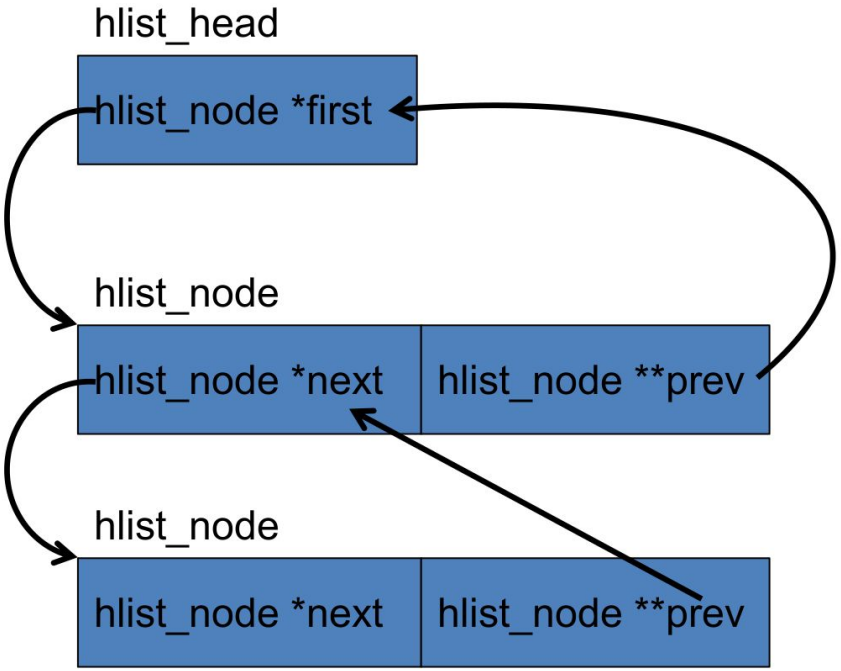
In some cases, for improving search efficiency we can use hash tables.

```
struct list_head {
    struct list_head *next, *prev;
};

struct hlist_head {
    struct hlist_node *first;
};

struct hlist_node {
    struct hlist_node *next, **pprev;
};
```

Hash Tables


hlist_head
hlist_node *first
hlist_node
hlist_node *next
hlist_node **prev
hlist_node
hlist_node *next
hlist_node **prev

Hash Tables

APIs

The interface to Levin's hash table is fairly straightforward. The API is defined in `linux/hashtable.h`.

```
DEFINE_HASHTABLE(name, bits)
hash_init(name);
hash_add(name, node, key);
hash_for_each(name, bkt, obj, member)
hash_for_each_possible(name, obj, member, key)
hash_del(node);
```

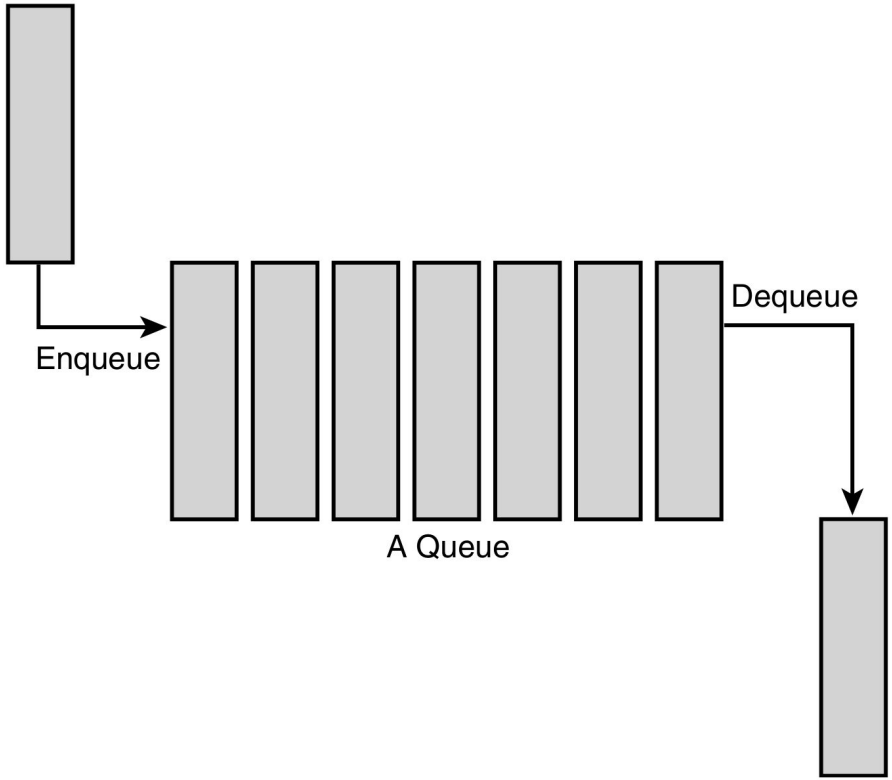
..and many others

https://elixir.bootlin.com/linux/v5.11/source/include/linux/hashtable.h

https://lwn.net/Articles/510202/

Queues

Queues



Love, Robert. Linux kernel development. Pearson Education, 2010.

Queues

V5.11

The implementation of the FIFO queue in the kernel is called `kfifo`.

APIs

```
int kfifo_alloc(struct kfifo *fifo, unsigned int size, gfp_t gfp_mask);
unsigned int kfifo_in(struct kfifo *fifo, const void *from, unsigned int len);
unsigned int kfifo_out(struct kfifo *fifo, void *to, unsigned int len);
static inline unsigned int kfifo_size(struct kfifo *fifo);
static inline unsigned int kfifo_len(struct kfifo *fifo);
static inline unsigned int kfifo_avail(struct kfifo *fifo);
static inline int kfifo_is_empty(struct kfifo *fifo);
static inline int kfifo_is_full(struct kfifo *fifo);
static inline void kfifo_reset(struct kfifo *fifo);
void kfifo_free(struct kfifo *fifo);
```

Maps

Maps

A map, also known as an associative array, is a collection of unique keys, where each key is associated with a specific value.The relationship between a key and its value is called a mapping. Maps support at least three operations:

- Add (key, value)
- Remove (key)
- value = Lookup (key)

The Linux kernel provides a simple and efficient map data structure, but it is not a general-purpose map. Instead, it is designed for one specific use case: mapping a unique identification number (UID) to a pointer. The data structure is called `idr`.

Love, Robert. Linux kernel development. Pearson Education, 2010.

Maps

V5.11

APIs

```
struct idr id_huh; /* statically define idr structure */
idr_init(&id_huh); /* initialize provided idr structure */
```

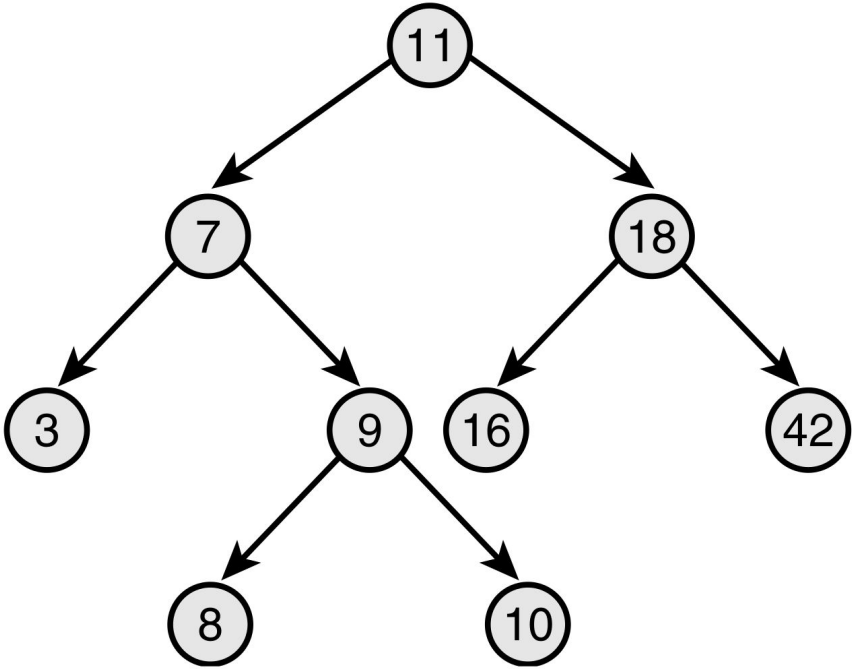
Once you have an `idr` set up, you can allocate a new UID with `idr_alloc`.

```
112  void idr_preload(gfp_t gfp_mask);
113
114  int idr_alloc(struct idr *, void *ptr, int start, int end, gfp_t);
115  int __must_check idr_alloc_u32(struct idr *, void *ptr, u32 *id,
116                                 unsigned long max, gfp_t);
117  int idr_alloc_cyclic(struct idr *, void *ptr, int start, int end, gfp_t);
118  void *idr_remove(struct idr *, unsigned long id);
119  void *idr_find(const struct idr *, unsigned long id);
120  int idr_for_each(const struct idr *,
121                   int (*fn)(int id, void *p, void *data), void *data);
122  void *idr_get_next(struct idr *, int *nextid);
123  void *idr_get_next_ul(struct idr *, unsigned long *nextid);
124  void *idr_replace(struct idr *, void *, unsigned long id);
125  void idr_destroy(struct idr *);
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/idr.h#L112

Trees

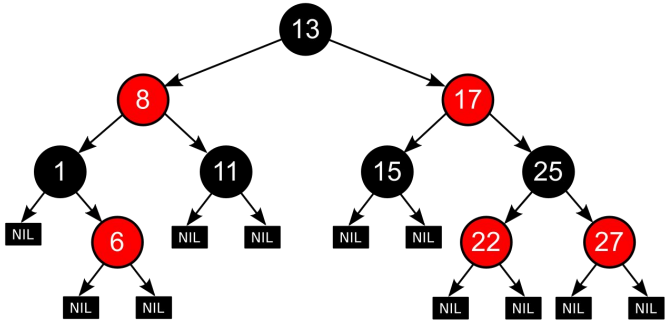
Binary Search Trees



Red-Black Trees

Linux has the API for a self-balancing binary search tree called Red-Black Tree. It is based on the following assumptions:

1. All nodes are either red or black.
2. Leaf nodes are black.
3. Leaf nodes do not contain data.
4. All non-leaf nodes have two children.
5. If a node is red, both of its children are black.
6. The path from a node to one of its leaves contains the same number of black nodes as the shortest path to any of its other leaves.



Red-Black Trees

They are defined in `/include/linux/rbtree.h`.

For initializing you need to use a special node:

```
struct rb_root root = RB_ROOT;
```

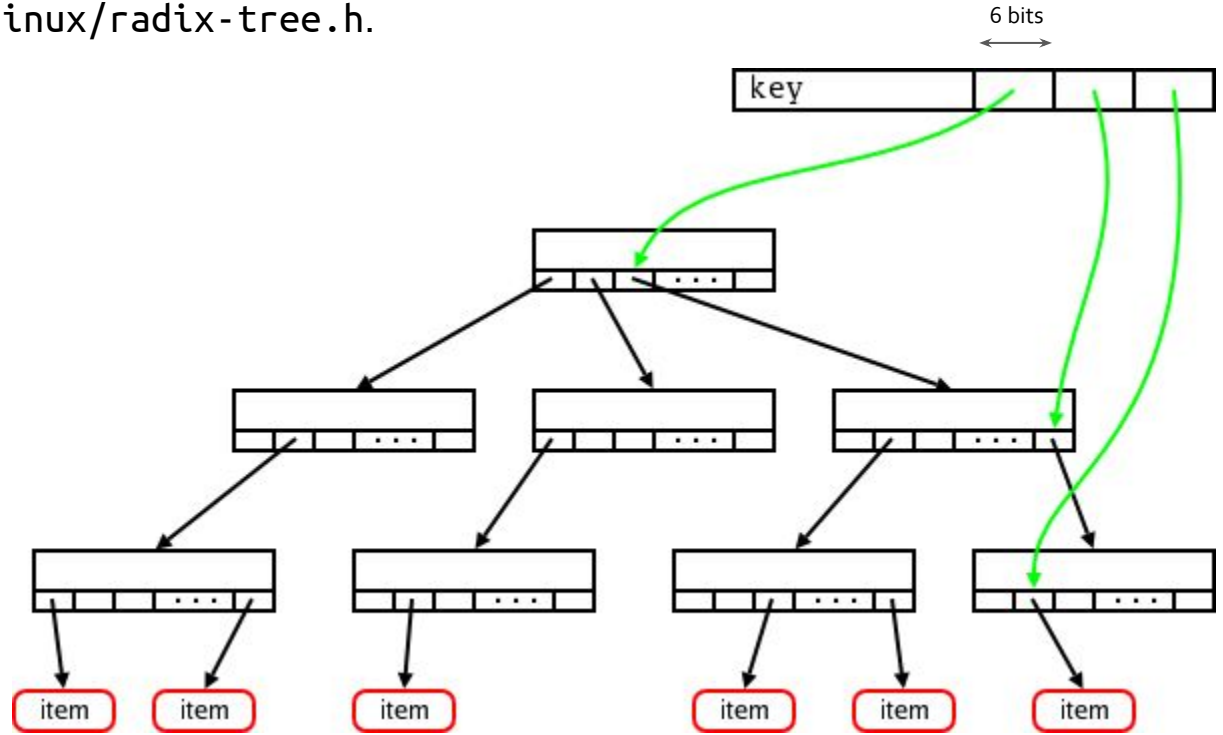
The API provides functions to:

- get the payload of a node: `rb_entry()`
- insert a node: `rb_link_node()`
- set the color (trigger rebalance) `rb_insert_color()`
- remove a node: `rb_erase()`

Traversal must be implemented by hand, see "Love, Robert. Linux kernel development. Pearson Education, 2010" for examples.

Radix Tree

Is the kind of tree on which IDR Maps (`/include/linux/idr.h`) are based. It is defined on `/include/linux/radix-tree.h`.



https://lwn.net/Articles/175432/

Advanced Operating Systems and Virtualization

[Lab 07] Kernel Data Structures

LECTURER

Gabriele **Proietti Mattia**

gpm.name · proiettimattia@diag.uniroma1.it

DIAG