

Kernel Data Structures

Advanced Operating Systems and Virtualization

Alessandro Pellegrini

A.Y. 2019/2020



SAPIENZA

UNIVERSITÀ DI ROMA

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



Circular Doubly-Linked Lists

- `/include/linux/list.h`

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



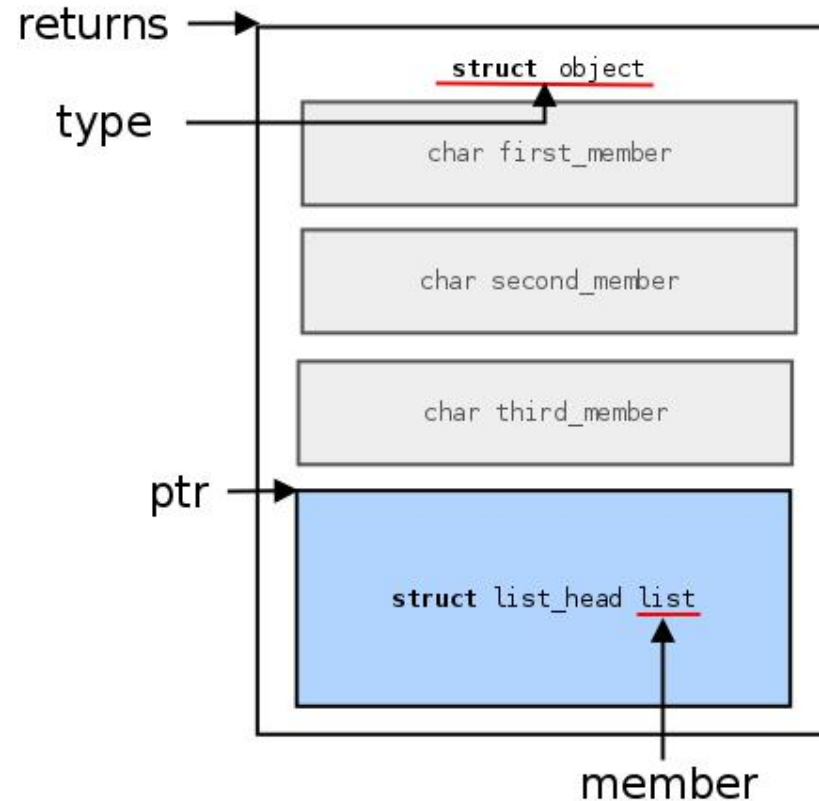
Circular Doubly-Linked Lists

```
struct my_struct {  
    int priority;  
    struct list_head list1;  
    struct list_head list1;  
    int other_member;  
};
```



Circular Doubly-Linked Lists

`container_of(ptr, type, member)`
illustrated explanation

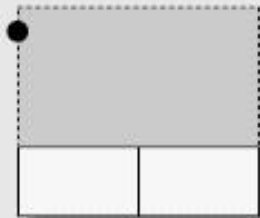


Circular Doubly-Linked Lists

Lists in
<linux/list.h>



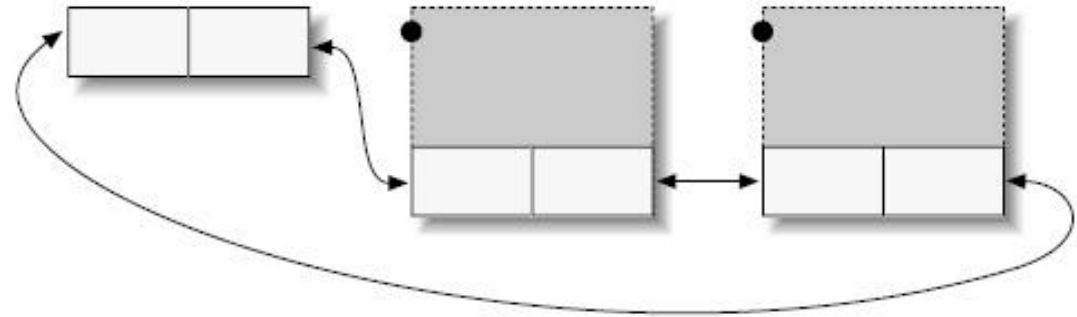
struct list_head



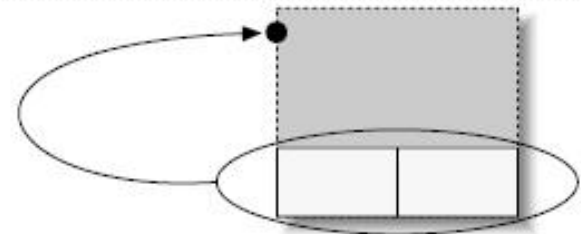
A custom structure
including a list_head



An empty list



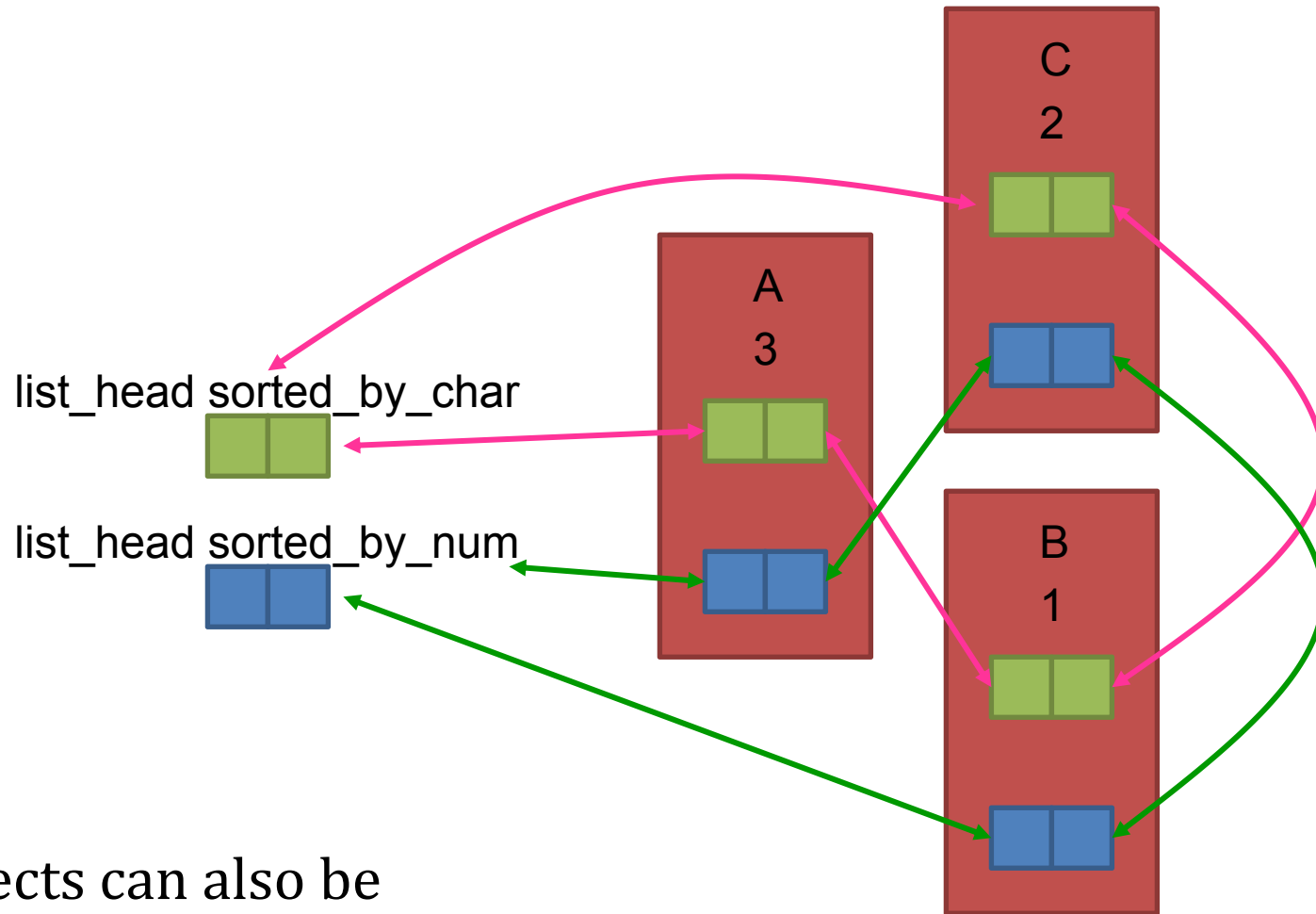
A list head with a two-item list



Effects of the list_entry macro



How to use Lists



Objects can also be
allocated into an array



Head of lists

- The head of the list is usually a standalone structure:

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

- If it is used as a global variable, it has to be initialized at compile time:

```
LIST_HEAD(todo_list);
```



Linked List API (partial)

- `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



List Traversal

```
void my_add_entry(struct my_struct *new) {
    struct list_head *ptr;
    struct my_struct *entry;
    for (ptr = my_list.next; ptr != &my_list; ptr = ptr->next) {
    entry = list_entry(ptr, struct my_struct, list);
    if (entry->priority < new->priority) {
        list_add_tail(&new->list, ptr);
        return;
    }
}
list_add_tail(&new->list, &my_list);
}
```



List Traversal

```
void my_add_entry(struct my_struct *new) {
    struct list_head *ptr;
    struct my_struct *entry;
    list_for_each(ptr, &todo_list) {
    entry = list_entry(ptr, struct my_struct, list);
    if (entry->priority < new->priority) {
        list_add_tail(&new->list, ptr);
        return;
    }
}
list_add_tail(&new->list, &my_list);
}
```



Hash Lists

- In some cases, storing two pointers in the head is a waste of memory (e.g., hash tables)

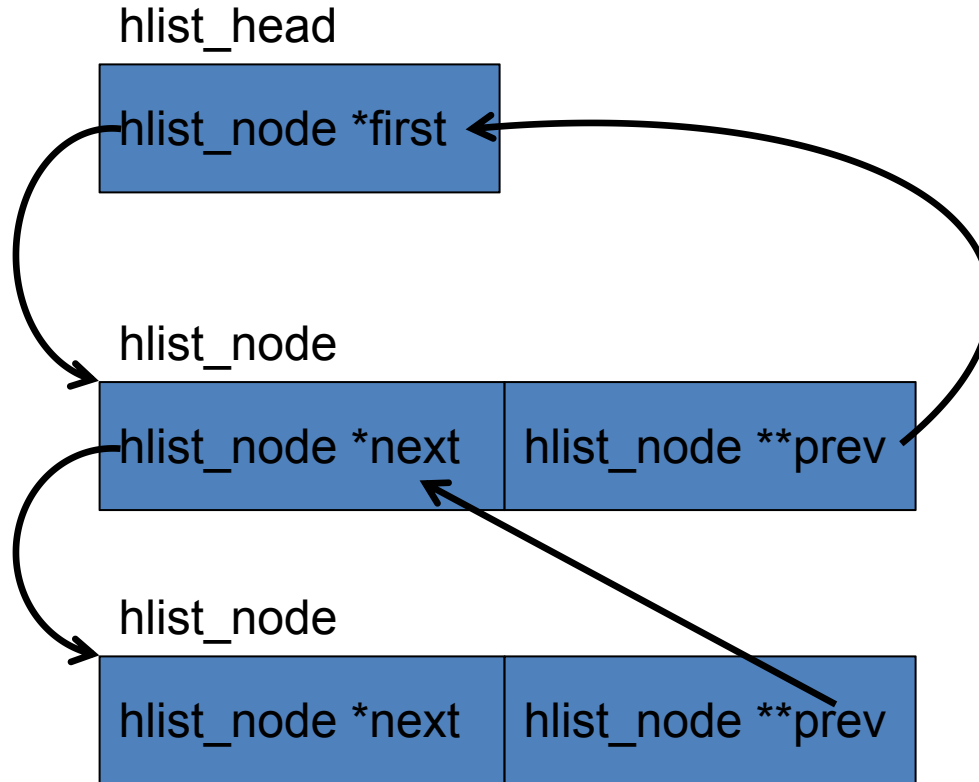
```
struct list_head {  
    struct list_head *next, *prev;  
};
```

```
struct hlist_head {  
    struct hlist_node *first;  
};
```

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

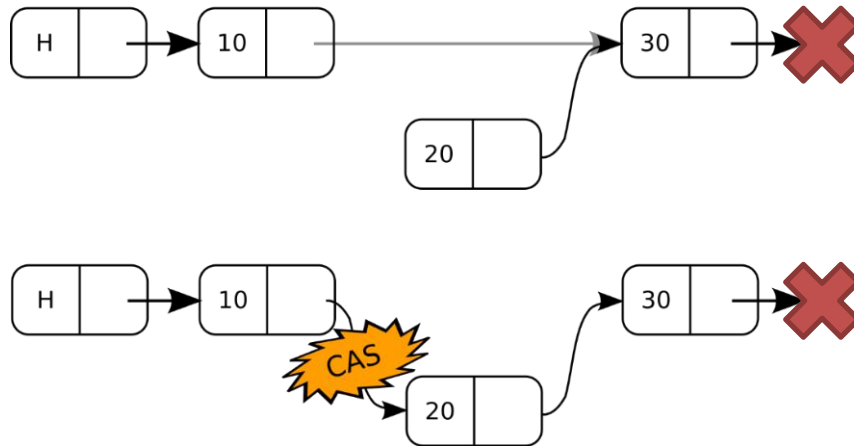
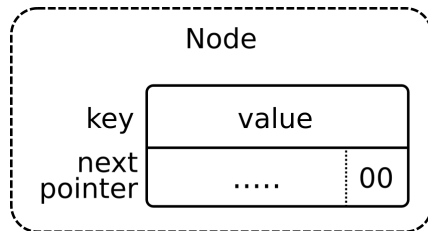


Hash Lists



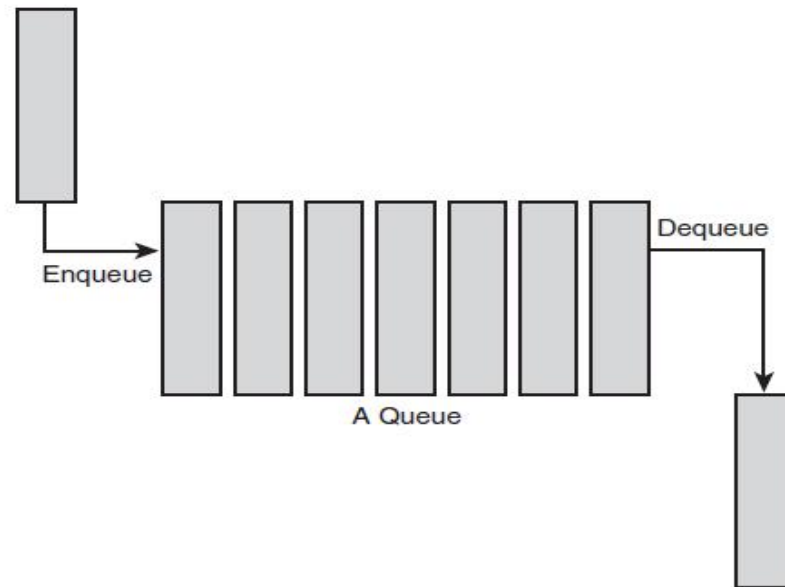
Lock-less Lists

- Singly-linked NULL-terminated non-blocking lists
- Based on compare and swap to update pointers
- If operations are carried out accessing only the single next pointer, RMW instructions allow concurrent access with no locking



Queues

- Producer/consumer model



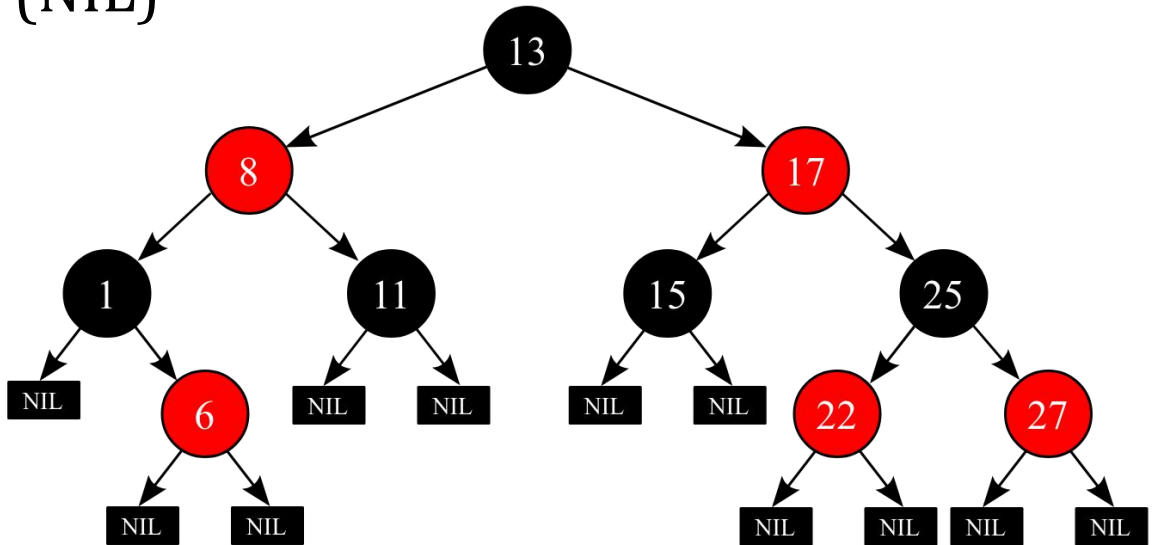
Queues

- Called `kfifo` in `/include/linux/kfifo.h`
- Two main operations:
 - Enqueue: `kfifo_in()`
 - Dequeue: `kfifo_out()`
- Creation:
 - `kfifo_alloc(struct kfifo *fifo, unsigned int size, gfp_t gfp_mask)`
- Removal:
 - `kfifo_free(struct kfifo *fifo)`



Red-Black Trees

- Self-balancing binary search tree
- Properties:
 - Each node is either black or red
 - Each path to leaf traverses the same number of black nodes
 - Each red node has two black children
 - All leaves are black (NIL)

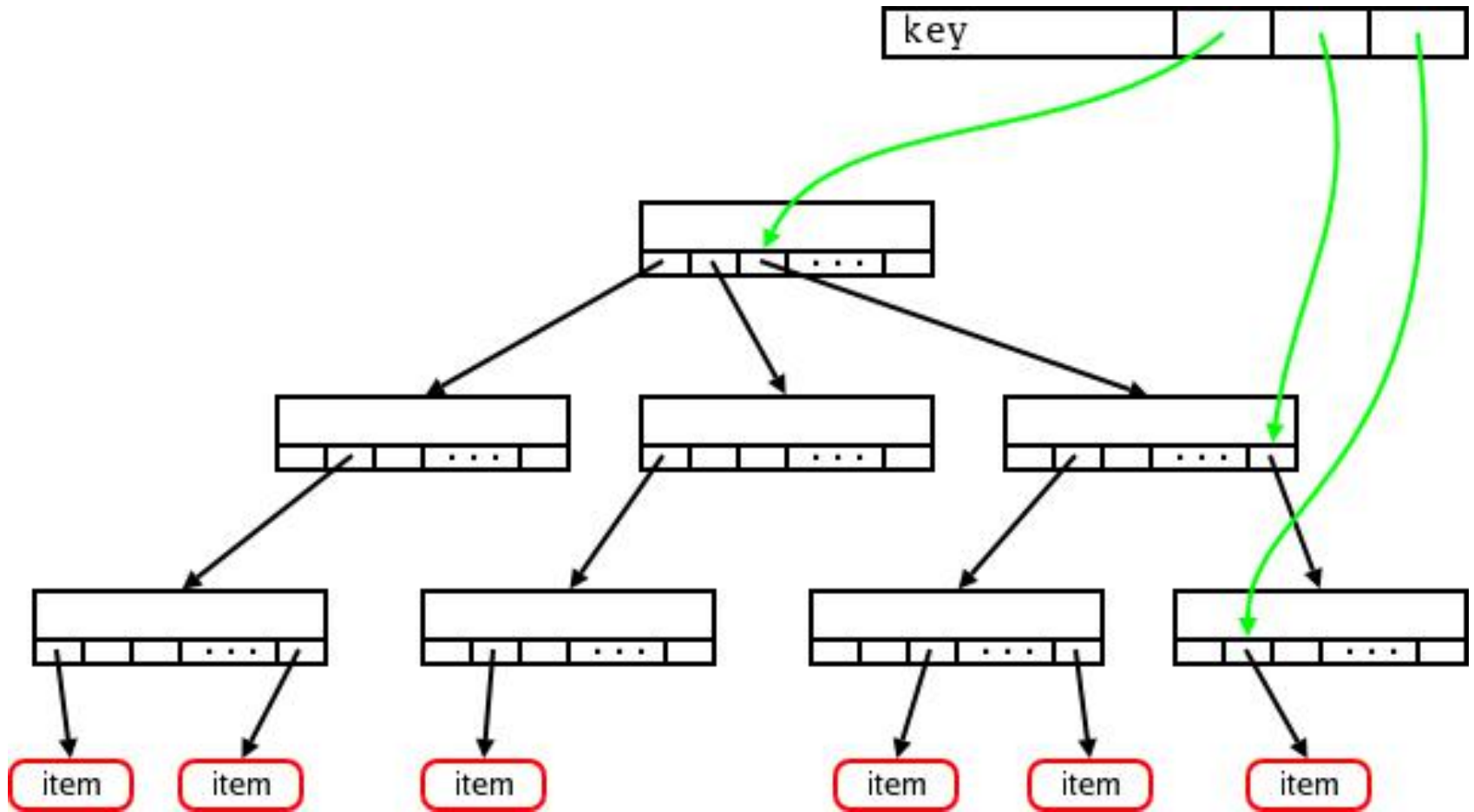


Red-Black Trees

- Defined in `/include/linux/rbtree.h`
- Initialization:
 - `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 rebalancing): `rb_insert_color()`
 - remove a node: `rb_erase()`
- Traversal must be implemented by hand (what should the default implementation compare?)



Radix Tree



Radix Tree

- There are two different implementations:
 - `/include/linux/radix-tree.h`
 - `/include/linux/idr.h` (simpler, based on the former)
- Both provide a mapping from a number (unsigned long) to a pointer (void *)
- They can be used to implement (sparse) maps
 - Empty nodes are not kept in the representation



idr Example

- This code allows any to cores to compete at allocating an ID.

```
again:
    if (idr_pre_get(&my_idr, GFP_KERNEL) == 0) {
        /* No memory, give up entirely */
    }
    spin_lock(&my_lock);
    result = idr_get_new(&my_idr, &target, &id);
    if (result == -EAGAIN) {
        sigh();
        spin_unlock(&my_lock);
        goto again;
    }
```

← can sleep, no lock



Per-CPU Variables

- They are variables referenced with the same name
- Depending on the core on which the code runs, this name is automatically mapped to different storage
- They are based on a reserved zone in the linear addressing space
- Macros allows to retrieve the actual address for the running core



Per-CPU Variables

- Definition and usage:

```
DEFINE_PER_CPU(int, x);  
int z;  
z = this_cpu_read(x);
```

- This is compiled to:

```
movl %gs:x, %eax
```



Per-CPU Variables

- The %gs segment points to a per-CPU area
 - This works only because we have a different GDT for each CPU!

