Kernel Data Structure I

Dongyoon Lee

Summary of last lectures

- Tools
  - git, tig, make, cscope, ctags, vim, emacs, tmux, ssh, etc.
- System call
  - isolation, x86 ring architecture

Today: Kernel Data Structures

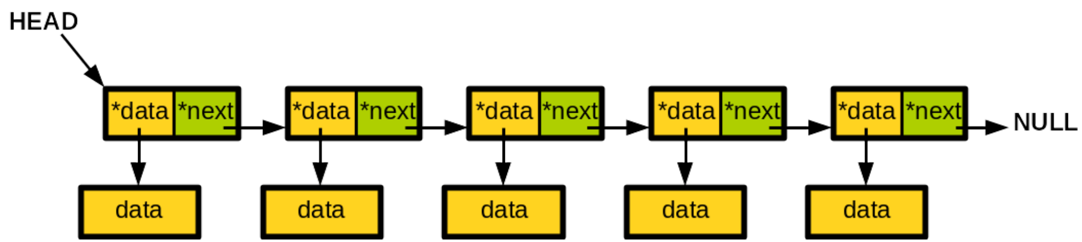
- Linked list
- Hash table
- Red-black tree
- Radix tree (next class)
- Bitmap (next class)

Why data structure is particularly important?

> *I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code and his data structures more important. Bad programmers worry about the code, Good programmers worry about data structures and their relationships. - Linus Torvalds*

Singly linked list (CS101)

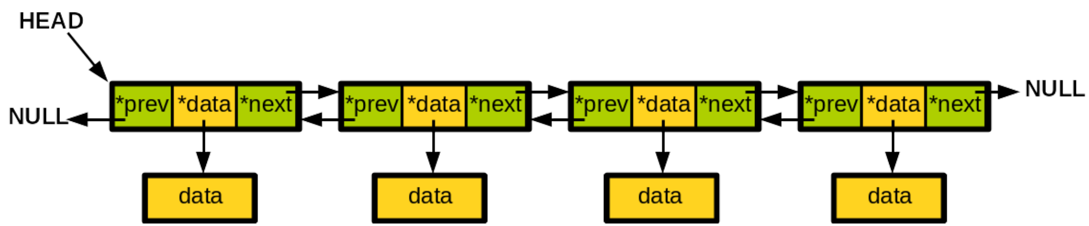
```
struct my_list_element {
    void                   *data; /* void pointer to point on a generic data */
    struct my_list_element *next; /* pointer to a next element */
};
```



- Starts from `HEAD` and terminates at `NULL`
- Traverses forward only
- When empty, `HEAD` is `NULL`

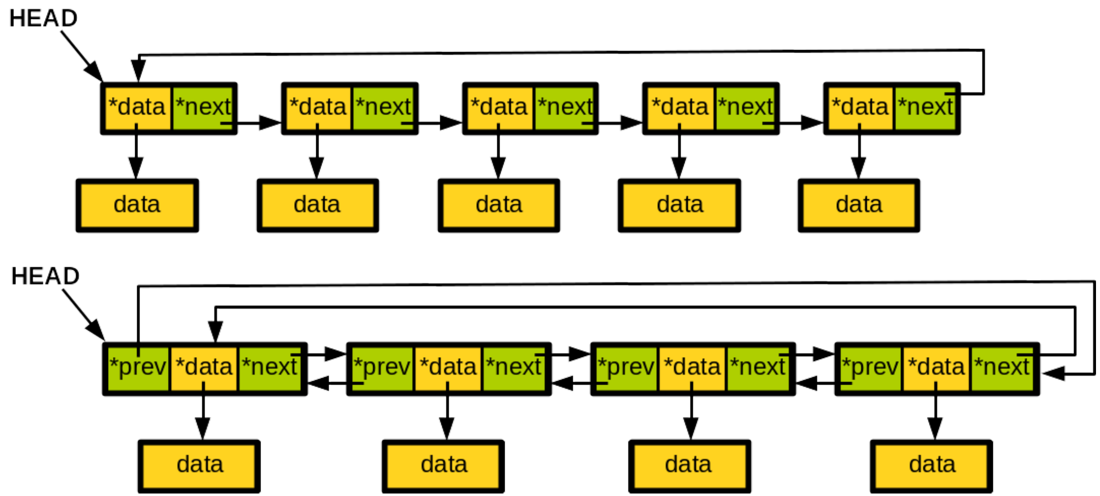
Doubly linked list (CS101)

```
struct my_list_element {
    void                   *data; /* void pointer to point on a generic data */
    struct my_list_element *prev; /* pointer to a previous element */
    struct my_list_element *next; /* pointer to a next element */
};
```



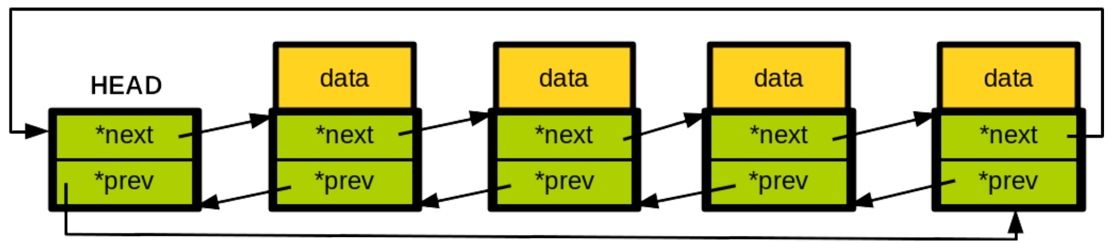
- Starts from `HEAD` and terminates at `NULL`
- Traverses forward and **backward**
- When empty, `HEAD` is `NULL`

Circular linked list (CS101)



- Starts from `HEAD` and terminates at `HEAD`
- When empty, `HEAD` is `NULL`
- **Easy to insert a new element at the end of a list**

Linux linked list



- Starts from `HEAD` and terminates at `HEAD`
- When empty, `HEAD` is **not** `NULL`
  - `prev` and `next` of `HEAD` points `HEAD`
  - `HEAD` is a sentinel node
- Easy to insert a new element at the end of a list
- **There is no exceptional case to handle `NULL`**

Linux linked list

- A circular doubly linked list
- Two differences from the typical design
    1. Embedding a linked list node in the structure
    2. Using a sentinel node as a list header
- `linux/include/linux/list.h`

Linux linked list

```
struct list_head {                  /* kernel linked list data structure */
    struct list_head *next, *prev;
};

struct car {
    struct list_head  list; /* add list_head instead of prev and next */
    unsigned int      max_speed; /* put data directly */
    unsigned int      drive_wheen_num;
    unsigned int      price_in_dollars;
};

struct list_head my_car_list; /* HEAD is also list_head */
```

- `struct list_head` is the key data structure
- `list_head` is embedded in the data structure
- Start of a list is also `list_head`, `my_car_list` → sentinel node

Getting a data from its `list_head`

- How to get the pointer of `struct car` from its `list`
  - use `list_entry(ptr, type, member)`
  - just a pointer arithmetic

```
struct car *amazing_car = list_entry(car_list_ptr, struct car, list);

/**
 * list_entry - get the struct for this entry
 * @ptr:    the &struct list_head pointer.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 */
#define list_entry(ptr, type, member) container_of(ptr, type, member)
#define container_of(ptr, type, member) ({ \
        const typeof( ((type *)0)->member ) *__mptr = (ptr); \
        (type *)( (char *)__mptr - offsetof(type,member) );})
#define offsetof(TYPE, MEMBER)  ((size_t)&((TYPE *)0)->MEMBER)
```

Defining a list

```
struct car *my_car = kmalloc(sizeof(*my_car), GFP_KERNEL);
my_car->max_speed = 150;
my_car->drive_wheel_num = 2;
my_car->price_in_dollars = 10000.0;
INIT_LIST_HEAD(&my_car->list);  /* initialize an element */

struct car my_car {
    .max_speed = 150,
    .drive_wheel_num = 2,
    .price_in_dollars = 10000,
    .list = LIST_HEAD_INIT(my_car.list), /* initialize an element */
}

LIST_HEAD(my_car_list); /* initialize the HEAD of a list */
```

- Initializing a `list_head`
  - `list_head->prev = &list_head`
  - `list_head->next = &list_head`

Manipulating a list: O(1)

```
/* Insert a new entry after the specified head */
void list_add(struct list_head *new, struct list_head *head);

/* Insert a new entry before the specified head */
void list_add_tail(struct list_head *new, struct list_head *head);

/* Delete a list entry
 * NOTE: You still have to take care of the memory deallocation if needed */
void list_del(struct list_head *entry);

/* Delete from one list and add as another's head */
void list_move(struct list_head *list, struct list_head *head);

/* Delete from one list and add as another's tail */
void list_move_tail(struct list_head *list, struct list_head *head);

/* Tests whether a list is empty */
int list_empty(const struct list_head *head);

/* Join two lists (merge a list to the specified head) */
void list_splice(const struct list_head *list, struct list_head *head);
```

Iterating over a list: O(n)

```
/**
 * list_for_each    -   iterate over a list
 * @pos:    the &struct list_head to use as a loop cursor.
 * @head:   the head for your list.
 */
#define list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)

/**
 * list_for_each_entry  -   iterate over list of given type
 * @pos:    the type * to use as a loop cursor.
 * @head:   the head for your list.
 * @member: the name of the list_head within the struct.
 */
#define list_for_each_entry(pos, head, member)              \
    for (pos = list_first_entry(head, typeof(*pos), member); \
         &pos->member != (head);                            \
         pos = list_next_entry(pos, member))
```

Iterating over a list: O(n)

```
/* Temporary variable needed to iterate: */
struct list_head p;

/* This will point on the actual data structures
 * (struct car)during the iteration: */
struct car *current_car;

list_for_each(p, &my_car_list) {
    current_car = list_entry(p, struct car, list);
    printk(KERN_INFO "Price: %lf\n", current_car->price_in_dollars);
}

/* Simpler: use list_for_each_entry */
list_for_each_entry(current_car, &my_car_list, list) {
    printk(KERN_INFO "Price: %lf\n", current_car->price_in_dollars);
}
```

- Backward iteration?
  - `list_for_each_entry_reverse(pos, head, member)`

Iterating while removing

```
#define list_for_each_safe(pos, next, head) ...
#define list_for_each_entry_safe(pos, next, head, member) ...

/* This will point on the actual data structures
 * (struct car) during the iteration: */
struct car *current_car, *next;
list_for_each_entry_safe(current_car, next, my_car_list, list) {
    printk(KERN_INFO "Price: %lf\n", current_car->price_in_dollars);
    list_del(current_car->list);
    kfree(current_car); /* if dynamically allocated using kmalloc */
}
```

- For each iteration, `next` points to the next node
  - Can safely remove the current node
  - Otherwise, can cause a use-after-free bug

Usages of linked lists in the kernel

- Kernel code makes extensive use of linked lists:
  - a list of threads under the same parent PID
  - a list of superblocks of a file system
  - and many more

Linux hash table

- A simple fixed-size open chaining hash table
  - The size of bucket array is fixed at initialization as a 2^N
  - Each bucket has a singly linked list to resolve hash collision.
  - Time complexity: O(1)

```
             Bucket
             +---+   Collision list
           0 |   |-->"John"-->"Kim"
 Key         +---+
  +-----> 1  |   |-->"Lisa"
  |          +---+
"Josh"     2 |   |-->"Min"
             +---+
           3 |   |
             +---+
```

Linux hash table

- A simple fixed-size chained hash table
  - The size of bucket array is fixed at initialization as a 2^N
  - Each bucket has a singly linked list to resolve hash collision.
  - Time complexity: O(1)

```
  Bucket
  +---+   Collision list
0 |   |-->"John"-->"Kim"
  +---+
1 |   |-->"Josh"-->"Lisa"
  +---+
2 |   |-->"Min"
  +---+
3 |   |
  +---+
```

Linux hash table

```
/* linux/include/linux/hashtable.h, types.h */
/* hash bucket */
struct hlist_head {
  struct hlist_node *first;
};

/* collision list */
struct hlist_node {
  /* Similar to list_head, hlist_node is embedded
   * into a data structure. */
  struct hlist_node *next;
  struct hlist_node **pprev; /* &prev->next */
};

  Bucket: array of hlist_head
  +---+   Collision list: hlist_node
0 |   |-->"John"-->"Kim"
  +---+
1 |   |-->"Josh"-->"Lisa"
  +---+
2 |   |-->"Min"
  +---+
3 |   |
  +---+
```

Linux hash table API

```
/**
 * Define a hashtable with 2^bits buckets
 */
#define DEFINE_HASHTABLE(name, bits) ...

/**
 * hash_init - initialize a hash table
 * @hashtable: hashtable to be initialized
 */
#define hash_init(hashtable) ...

/**
 * hash_add - add an object to a hashtable
 * @hashtable: hashtable to add to
 * @node: the &struct hlist_node of the object to be added
 * @key: the key of the object to be added
 */
#define hash_add(hashtable, node, key) ...
```

Linux hash table API

```
/**
 * hash_for_each - iterate over a hashtable
 * @name: hashtable to iterate
 * @bkt: integer to use as bucket loop cursor
 * @obj: the type * to use as a loop cursor for each entry
 * @member: the name of the hlist_node within the struct
 */
#define hash_for_each(name, bkt, obj, member) ...

            +---+
          0 |   |-->"John"<-->"Kim"
            +---+
          1 |   |-->"Josh"<-->"Lisa"
            +---+
          2 |   |-->"Min"
            +---+
          3 |   |
            +---+
```

Linux hash table API

```
/**
 * hash_for_each_possible - iterate over all possible objects hashing to the
 * same bucket
 * @name: hashtable to iterate
 * @obj: the type * to use as a loop cursor for each entry
 * @member: the name of the hlist_node within the struct
 * @key: the key of the objects to iterate over
 */
#define hash_for_each_possible(name, obj, member, key) ...

            +---+
          1 |   |-->"Josh"<-->"Lisa"
            +---+

/**
 * hash_del - remove an object from a hashtable
 * @node: &struct hlist_node of the object to remove
 */
void hash_del(struct hlist_node *node);
```

Linux hash table example

- Transparent hugepage
  - finds physically consecutive 4KB pages
  - remaps consecutive 4KB pages to a 2MB page (huge page)
  - saves TLB entries and improves memory access performance by reduing TLB miss
  - maintains per-process memory structure, `struct mm_struct`

Linux hash table example

```
/* linux/mm/khugepaged.c */

#define MM_SLOTS_HASH_BITS 10
static DEFINE_HASHTABLE(mm_slots_hash, MM_SLOTS_HASH_BITS);

/* struct mm_slot - hash lookup from mm to mm_slot
 * @hash: hash collision list
 * @mm: the mm that this information is valid for
 */
struct mm_slot {
    struct hlist_node hash;  /* hlist_node is embedded like list_head */
    struct mm_struct *mm;
};
```

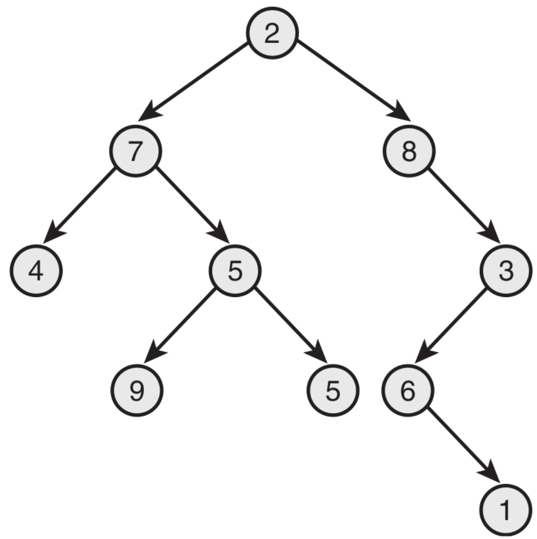
Linux hash table example

```
/* add an mm_slot into the hash table
 * use the mm pointer as a key */
static void insert_to_mm_slots_hash(struct mm_struct *mm,
                    struct mm_slot *mm_slot)
{
    mm_slot->mm = mm;
    hash_add(mm_slots_hash, &mm_slot->hash, (long)mm);
}

/* iterate the chained list of a bucket to find an entry */
static struct mm_slot *get_mm_slot(struct mm_struct *mm)
{
    struct mm_slot *mm_slot;

    hash_for_each_possible(mm_slots_hash, mm_slot, hash, (unsigned long)mm)
        if (mm == mm_slot->mm)
            return mm_slot;

    return NULL;
}
```

Linux hash table example

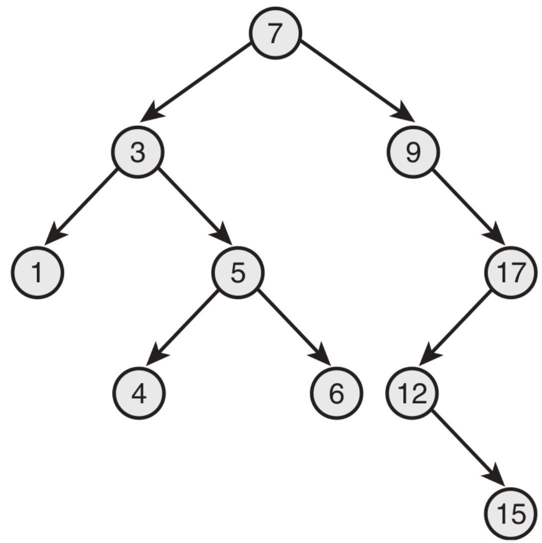
```
/* remove an entry after finding it */
void __khugepaged_exit(struct mm_struct *mm)
{
    struct mm_slot *mm_slot;

    spin_lock(&khugepaged_mm_lock);
    mm_slot = get_mm_slot(mm);
    if (mm_slot && khugepaged_scan.mm_slot != mm_slot) {
        hash_del(&mm_slot->hash);
        list_del(&mm_slot->mm_node);
        free = 1;
    }
    spin_unlock(&khugepaged_mm_lock);

    clear_bit(MMF_VM_HUGEPAGE, &mm->flags);
    free_mm_slot(mm_slot);
    mmdrop(mm);
    /* ... */
}
```

Tree basics: binary tree



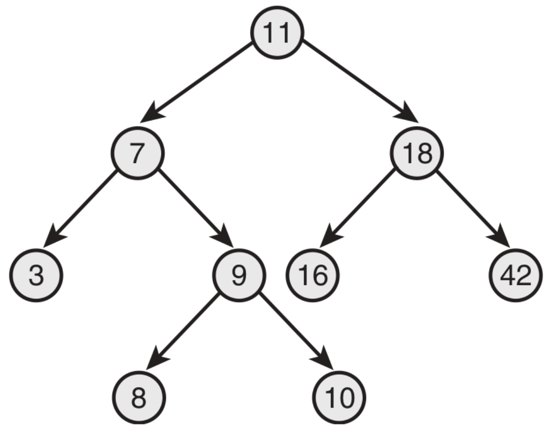
- Nodes have zero, one, or two children
- Root has no parent, other nodes have one

Tree basics: binary search tree



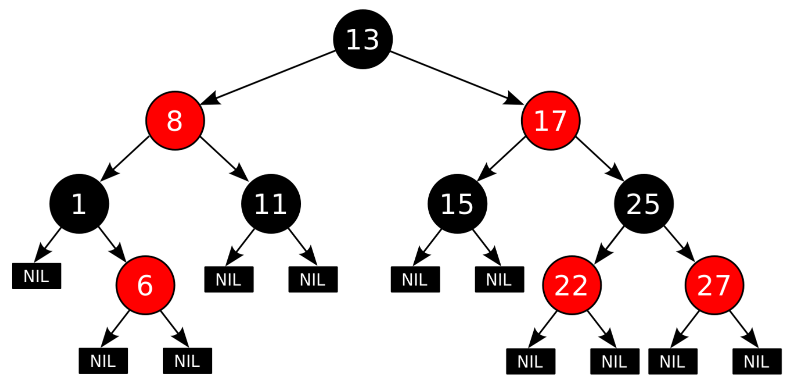
- Left children < parent
- Right children > parent
- Search and ordered traversal are efficient

Tree basics: balanced binary search tree



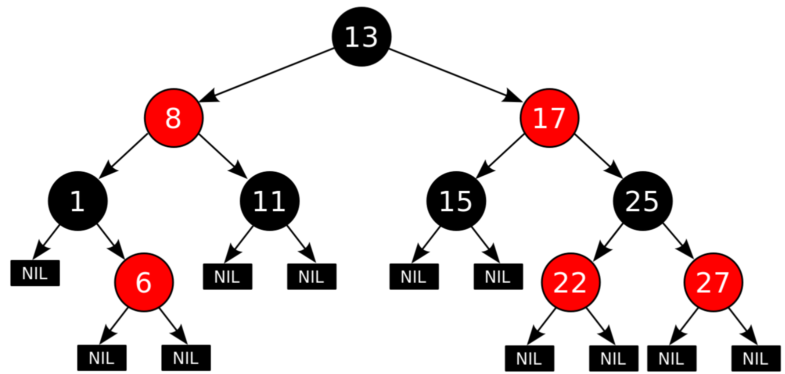
- Depth of all leaves differs by at most one
- Puts a boundary on the worst case operations

Tree basics: red-black tree



- A type of self-balancing binary search tree
  - Nodes: red or black
  - Leaves: black, no data

Tree basics: red-black tree



- Following properties are maintained during tree modifications:
  - 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.
- Fast search, insert, delete operations: $O(log^N)$

Linux red-black tree (or rbtree)

```
/* linux/include/linux/rbtree.h
 * linux/lib/rbtree.c */

/* Rbtree node, which is embedded to your data structure like
 * list_head and hlist node */
struct rb_node {
    unsigned long  __rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
};

/* Root of a rbtree */
struct rb_root {
    struct rb_node *rb_node;
};

#define RB_ROOT (struct rb_root) { NULL, }

/* A macro to access data from rb_node */
#define rb_entry(ptr, type, member) container_of(ptr, type, member)
#define rb_parent(r) ((struct rb_node *)((r)->__rb_parent_color & ~3))
```

Linux red-black tree (or rbtree)

```
/* Find logical next and previous nodes in a tree */
struct rb_node *rb_next(const struct rb_node *);
struct rb_node *rb_prev(const struct rb_node *);
struct rb_node *rb_first(const struct rb_root *);
struct rb_node *rb_last(const struct rb_root *);

/* Insert a new node under a parent connected via rb_link */
void rb_link_node(struct rb_node *node, struct rb_node *parent,
                  struct rb_node **rb_link);

/* Re-balance an rbtree after inserting a node if necessary */
void rb_insert_color(struct rb_node *, struct rb_root *);

/* Delete a node */
void rb_erase(struct rb_node *, struct rb_root *);
```

Linux red-black tree example

- Completely Fair Scheduling (CFS)
  - Default task scheduler in Linux
  - Each task has `vruntime`, which presents how much time a task has run
  - CFS always picks a process with the smallest `vruntime` for fairness
  - Per-task `vruntime` structure is maintained in a rbtree

Linux red-black tree example

```
/* linux/include/linux/sched.h
 * linux/kernel/sched/fair.c, sched.h */

/* Define an rbtree */
struct cfs_rq {
    struct rb_root tasks_timeline; /* contains sched_entity */
};

/* Data structure of a task */
struct sched_entity {
    struct rb_node  run_node; /* embed a rb_node */
    u64             vruntime; /* vruntime is the key of task_timeline */
};

/* Initialize an rbtree */
void init_cfs_rq(struct cfs_rq *cfs_rq)
{
    cfs_rq->tasks_timeline = RB_ROOT;
}
```

Linux red-black tree example

```
/* Enqueue an entity into the rb-tree: */
void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node; /* root node */
    struct rb_node *parent = NULL;
    struct sched_entity *entry;

    /* Traverse the rbtree to find the right place to insert */
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        if (se->vruntime < entry->vruntime) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
        }
    }

    /* Insert a new node */
    rb_link_node(&se->run_node, parent, link);
    /* Re-balance the rbtree if necessary */
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Linux red-black tree example

```
/* Delete a node */
void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    rb_erase(&s->run_node, &cfs_rq->tasks_timeline);
}

/* Pick the first enitiy, which has the smallest vruntime,
 * for scheduling */
struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
{
    return rb_first(&cfs_rq->tasks_timeline);
}
```

Design patterns of kernel data structures

- Embedding its pointer structure
  - `list_head`, `hlist_node`, `rb_node`
  - The programmer has full control of placement of fields in the structure in case they need to put important fields close together to improve cache utilization
  - A structure can easily be on two or more lists quite independently, simply by having multiple `list_head` fields
  - `container_of`, `list_entry`, and `rb_entry` are used to get its embedding data structure

Design patterns of kernel data structures

- Tool box rather than a complete solution for generic service
  - Sometimes it is best not to provide a complete solution for a generic service, but rather to provide a suite of tools that can be used to build custom solutions.
  - None of Linux list, hash table, and rbtree provides a `search` function.
  - You should build your own using given low-level primitives

Design patterns of kernel data structures

- Caller locks
  - When there is any doubt, choose to have the caller take locks rather than the callee. This puts more control in that hands of the client of a function.

Further readings

- LWN: Linux kernel design patterns: part 2
- LWN: A generic hash table
- Hash Tables—Theory and Practice
- LWN: Relativistic hash tables
- LWN: Trees II: red-black trees
- Transparent Hugepage Support
- CFS Scheduler

Next lecture

- More kernel data structures
  - radix tree, bitmap
- Kernel modules