Linux Kernel Data Structure

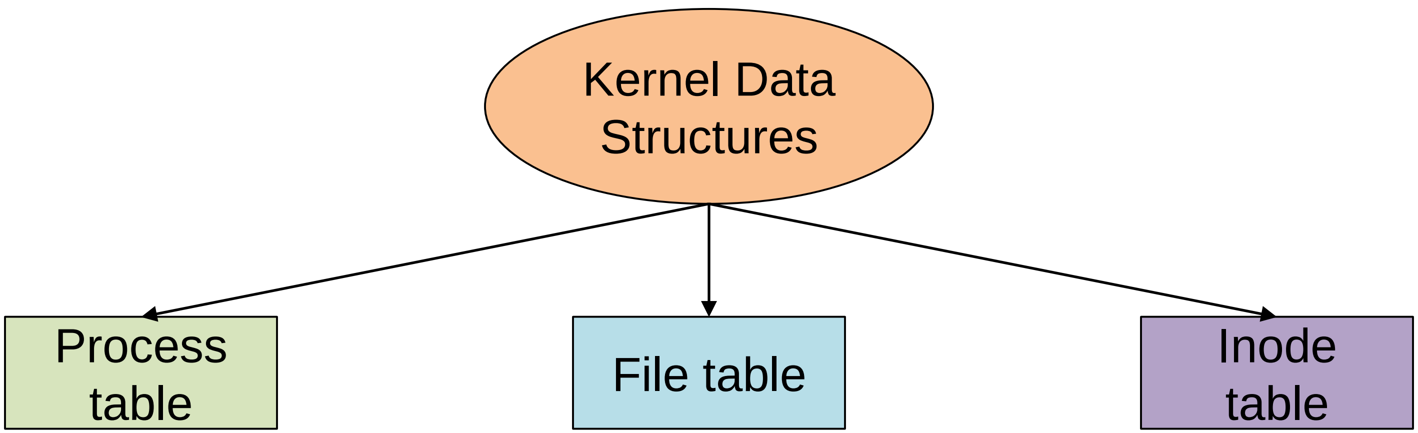
Chung-Ang University

Kernel Data Structure

- The kernel data structures are very important as they store data about the current state of the system
  - For example, if a new process is created in the system, a kernel data structure is created that contains the detail information or state about the process
  - Kernel data structures are only accessible by the kernel and its subsystems
  - They contain data as well as pointers to other data structures

Kernel Data Structure

- The kernel data structure stores and organizes a lot of information
  - For example, it has data about which processes are running in the system, their memory requirements, files in use, etc
  - To handle all this, three important structures are used
  - These are process table, file table, and inode table



The process table stores information about all the processes running in the system

The file table contains entries about all the files in the system

The inode table contains accessing information for the blocks of the file

Kernel Data Structures

- This chapter introduces several built-in data structures for use in Linux kernel code
  - As with any large software project, the Linux kernel provides these generic data structures to encourage code reuse
  - Kernel developers can use these data structures

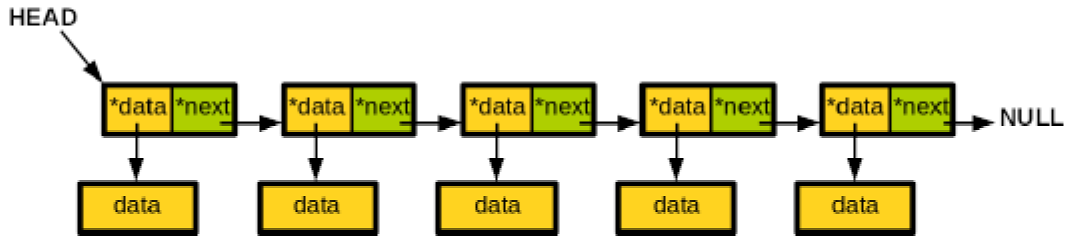
Types of Linux Kernel Data structure

- Linked list
- Red Black tree
- Hash table

Singly linked list

- Singly linked list is a basic linked list type
- Singly linked list is a collection of nodes linked together in a sequential way
- Each node of singly linked list contains a data field and an address field which contains the reference of the next node

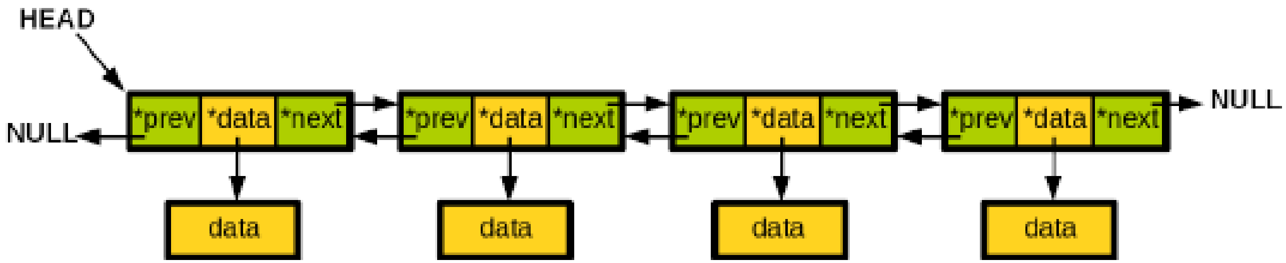
```
struct my_list_element{
        void *data; /* void pointer to point on a generic data */
        struct my_list_element *next; /* pointer to a next element */
};
```



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

Doubly linked list

- Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field

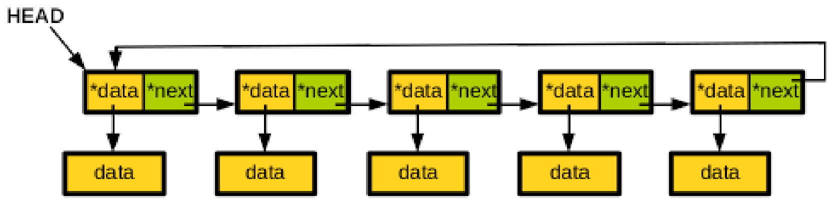



```
struct my_list_element{
        void *data; /* void pointer to point on a generic data */
        struct my_list_element *prev; /* pointer to a previous element */
        struct my_list_element *next; /* pointer to a next element */
};
```

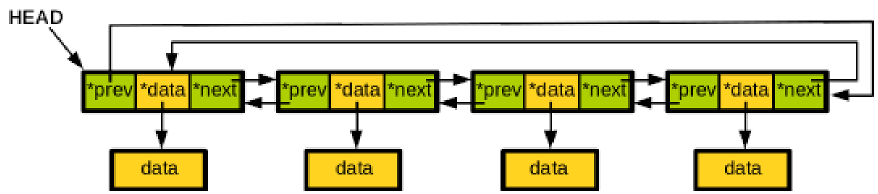
- Starts from **HEAD** and terminates at **NULL**
- Traverses forward and backward
- When empty, **HEAD** is **NULL**

Circular linked list

- Circular linked list is a linked list where all nodes are connected to form a circle
  - There is no NULL at the end
  - A circular linked list can be a singly circular linked list or doubly circular linked list



Singly circular linked list

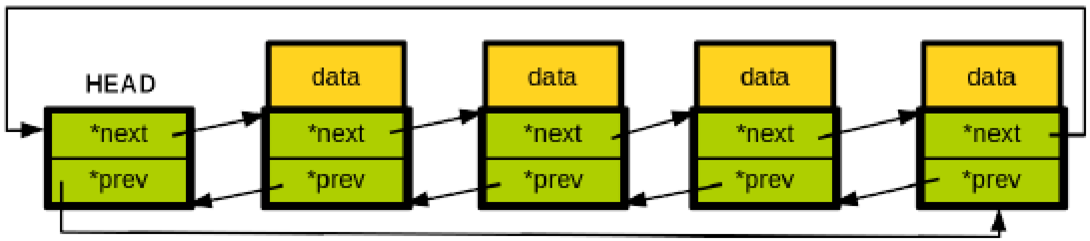


Doubly circular linked list

  - Starts from **HEAD** and terminates at **HEAD**
  - When empty, **HEAD** is **NULL**

Linux Kernel Linked List

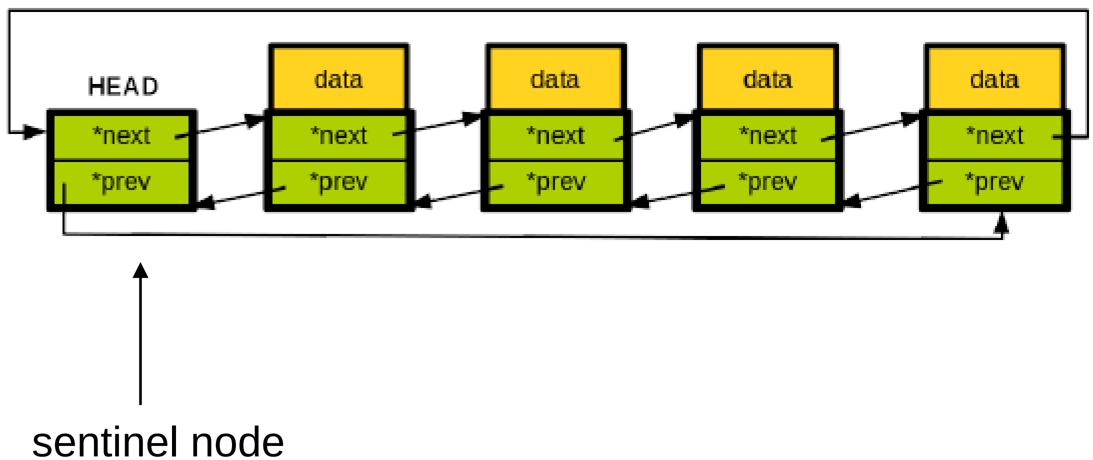
- **Linux kernel linked list**



  - 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 Kernel Linked List

- Linux kernel linked list is a circular doubly linked list
- Two differences from the typical design
  - Embedding a linked list node in the structure
  - Using a sentinel node as a list header
- linux/include/linux/list.h



Linux Kernel Linked List

- 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

```
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 */
```

Linux Kernel Linked List

- Getting a data from **list_head**
  - use list_entry (ptr, type, member)

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)

Linux Kernel Linked List

- **Linux kernel linked list APIs**

```
/* 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);

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

Linux Kernel Linked List

- **Linux kernel linked list APIs**

```
/**
* 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))
```

Linux Kernel Linked List

- **An example of Linux kernel linked list**

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/list.h>
#include <linux/slab.h> // for kmalloc

struct my_node {
        struct list_head list;
        int data;
};
```

```
int __init hello_module_init(void)
{
    struct_exmaple();
    printk("module init\n");
    return 0;
}

void __exit hello_module_cleanup(void)
{
    printk("Bye Module\n");
}

module_init(hello_module_init);
module_exit(hello_module_cleanup);
```

Linux linked list

- ## Linux linked list example

```
struct my_node {
        struct list_head list;
        int data;
};
```

```
#define list_for_each(pos, head) \
        for (pos = (head)->next; pos != (head); pos = pos->next)

#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))
```

```
void struct_exmaple(void)
{
        struct list_head my_list;

        /* initialize list */
        INIT_LIST_HEAD(&my_list);

        /* add list element */
        int i;
        for (i = 0; i<10;i++){
                struct my_node *new =  kmalloc(sizeof(struct my_node),GFP_KERNEL);
                new->data = i;
                list_add(&new->list, &my_list);
        }

        struct my_node *current_node; /* This will point on the actual data structures during the iteration */
        sturct list_head *p; /* Temporary variable needed to iterate; */
        list_for_each(p, &my_list){
                current_node = list_entry(p, struct my_node, list);
                printk("current value : %d\n", current_node->data);
        }
        list_for_each_entry(current_node, &my_list, list){
                printk("current value : %d\n", current_node->data);
        }
```

Linux linked list

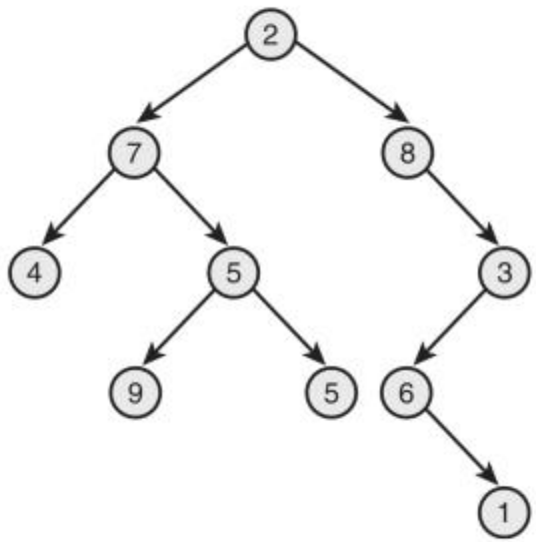
```
        /*  iterate over a list reversely*/
        list_for_each_entry_reverse(current_node, &my_list, list){
                printk("current value : %d\n", current_node->data);
        }

        /* delete list element */
        list_for_each_entry(current_node, &my_list, list){
                if(current_node->data == 2){
                        printk("current node value : %d\n", current_node->data);
                        list_del(&current_node->list);
                        kfree(current_node);
                }
        }

        /* iterate over a list */
        list_for_each_entry(current_node, &my_list, list){
                printk("current value : %d\n", current_node->data);
        }
}
```

Linux Kernel Linked List

- **Usage of linked list 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

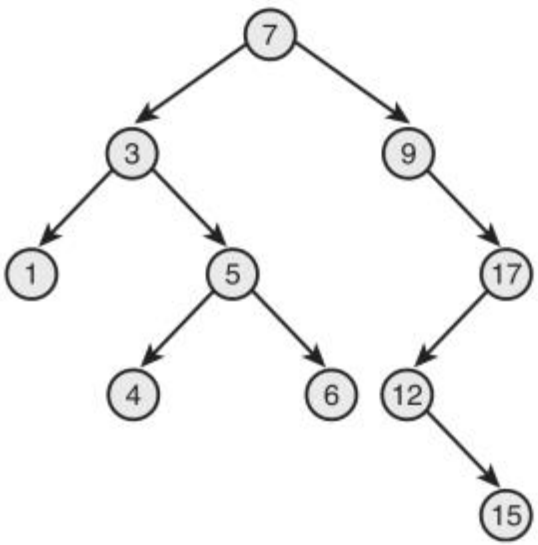
Red-Black Tree

- **Tree basics : binary tree**
- **Properties**
  - Nodes have zero, one, or two children
  - Root has no parent, other nodes have one



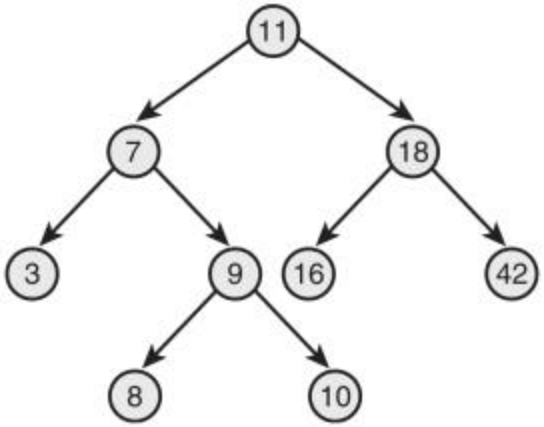
Red-Black Tree

- **Tree basics : binary search tree**
- **Properties**
  - Left children < parent
  - Right children > parent
  - Search and ordered traversal are efficient



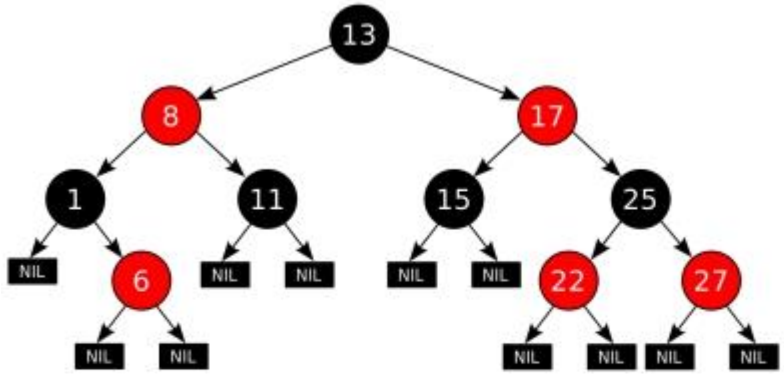
Red-Black Tree

- **Tree basics : balanced binary search tree**
- **Properties**
  - Depth of all leaves differs by at most one



Red-Black Tree

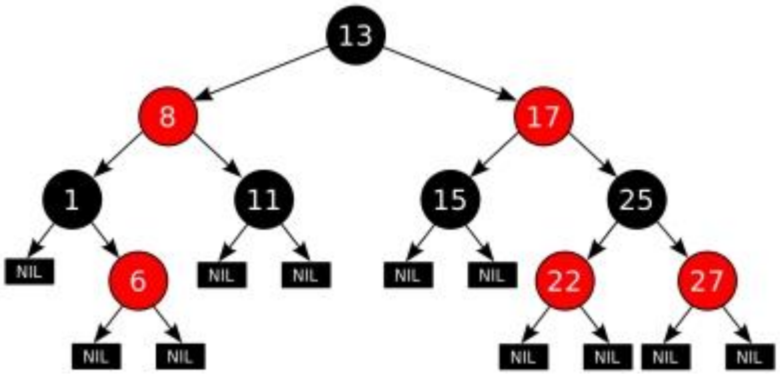
- **Tree basics : red-black tree**
- **Self-balancing binary search tree**
- **Properties**
  - A type of self-balancing binary search tree
  - Nodes: red or black
  - Leaves: black, no data



Linux Kernel Red-Black Tree

- **Self-balancing binary search tree**

```
/* 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;
};
```



Linux Kernel Red-Black Tree

- **red-black tree APIs**

```
/* 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 Kernel Red-Black Tree

- **Red-Black Tree example**

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/rbtree.h>
#include <linux/slab.h> // for kmalloc

#define FALSE 0
#define TRUE 1

struct my_node {
		struct rb_node node;
		int key;
		int value;
};
```

```
int __init hello_module_init(void)
{

		RB_exmaple();
		printk("module init\n");
		return 0;
}

void __exit hello_module_cleanup(void)
{
		printk("Bye Module\n");
}

module_init(hello_module_init);
module_exit(hello_module_cleanup);
```

Linux Kernel Red-Black Tree

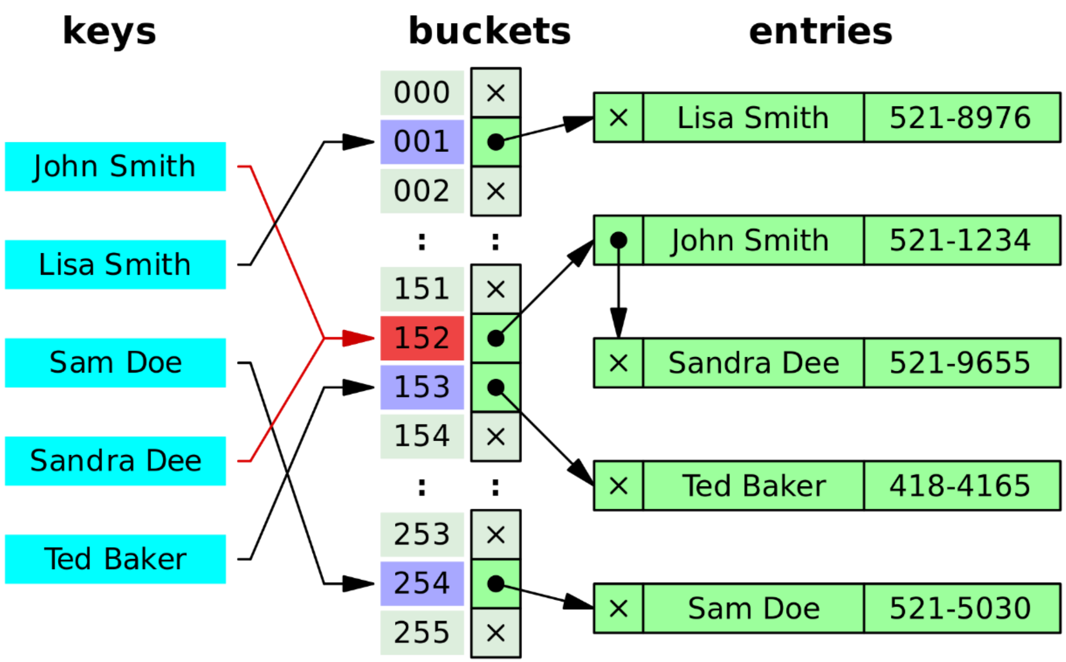
```
void RB_exmaple(void)
{
        struct rb_root root_node = RB_ROOT;
        int i;

       /* rb_node create and insert */
        for(i=0;i<20;i++){
                struct my_node *new_node = kmalloc(sizeof(struct my_type),GFP_KERNEL);
                if(!new_node)
                        return NULL;
                new_node->value = i*10;
                new_node->key = i;
                ret = rb_insert_color(new_node, &root_node);
        }

        /* rb_tree traversal using iterator */
        struct rb_node *iter_node;
        for (iter_node = rb_first(&root_node); iter_node; iter_node = rb_next(iter_node))
                printk("(key,value)=(%d.%d)\n",    \
                        rb_entry(iter_node, struct my_node, node)->key,    \
                        rb_entry(iter_node, struct my_node, node)->value);

        /* rb_tree delete node */
        rb_erase(new_node, &root_node);
}
```

Red-Black Tree

- **Usage of Red-Black tree in the kernel**
  - 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

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 or doubly linked list to resolve hash collision



Hash table

▪ Hash table APIs

```
/**
* 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) ...
```

Hash table

- ## Hash table APIs

```
/**
* 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) ...

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

Hash table

- **Hash table example**

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/hashtable.h>
#include <linux/slab.h> // for kmalloc

#define MY_HASH_BITS 2

struct my_node {
		u32 key;
		int value;
		struct hlist_node hnode;
};
```

```
int __init hello_module_init(void)
{

		hash_exmaple();
		printk("module init\n");
		return 0;
}

void __exit hello_module_cleanup(void)
{
		printk("Bye Module\n");
}

module_init(hello_module_init);
module_exit(hello_module_cleanup);
```

Hash table

```
void hash_exmaple(void)
{
        DEFINE_HASHTABLE(my_hash, MY_HASH_BITS);
        hash_init(my_hash);

        /* (key,value) insert */
        int i;
        for(i=0;i<10;i++){
                struct my_node *new = kmalloc(sizeof(struct my_node), GFP_KERNEL);
                new->value = i * 10;
                new->key = i;
                memset(&new->hnode, 0, sizeof(struct hlist_node));
                hash_add(my_hash,&new->hnode,new->key);
        }

        /* You can complete hash_for_each() function!!! */
        hash_for_each(…..){
                ……
        }

        /* You can complete hash_del() function!!! */
        hash_del(….){
                ……
        }
}
```

Hash table

- **Usage of hash table in the kernel**
  - 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**