CSE 306 Operating Systems
Kernel Data Structures

YoungMin Kwon

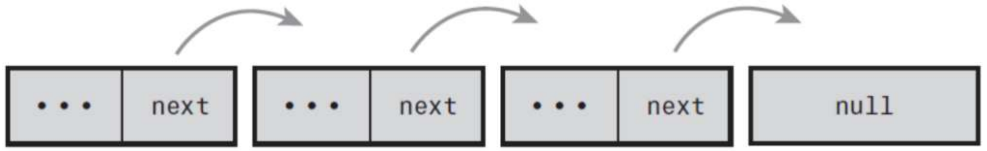
Kernel Data Structures

- Built-in data structures used in Linux kernel
  - Linked lists
  - Queues
  - Maps
  - Binary trees

Linked Lists

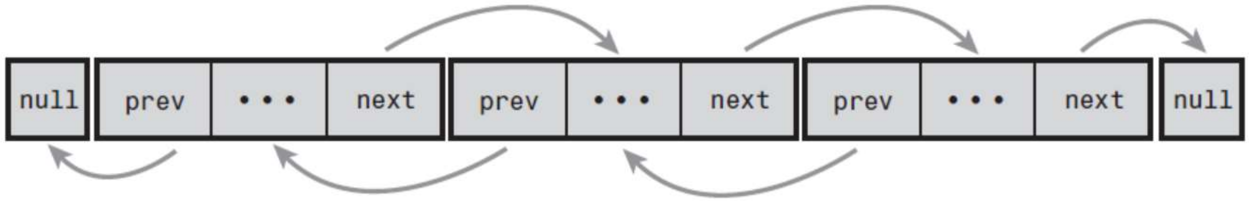
- Singly linked list

```
struct list_element {
    void *data;
    struct list_element *next;
};
```



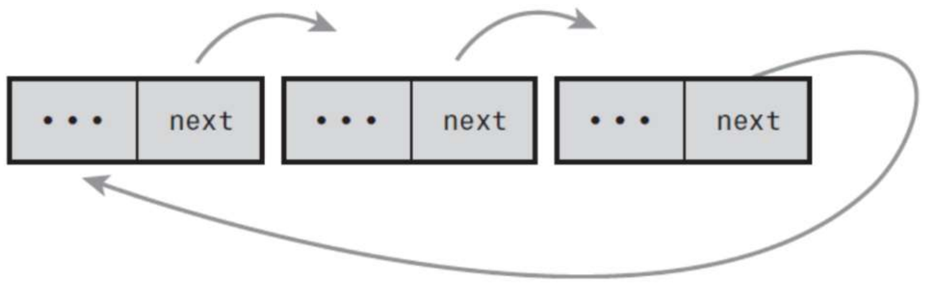
- Double linked list

```
struct list_element {
    void *data;
    struct list_element *next;
    struct list_element *prev;
};
```

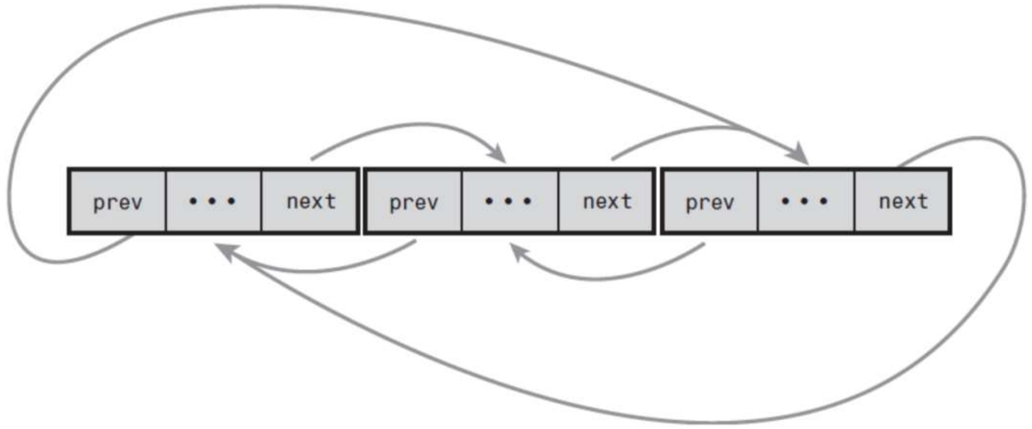


Circular Linked Lists

- Circular singly linked list



- Circular doubly linked list



Linked Lists (Linux Kernel's Implementation)

- Embedding the list structure into data

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

struct data {
    int a;
    int b;
    struct list_head list;
};
```

  - `next` and `prev` are pointing to the embedded `list_head` structures **not** `data`.

  - How to access data from list?

Linux Kernel's Linked List

:to access the container

```
#define offsetof(st, m) ( (size_t) &(((st *)0)->m) )

int main() {
    struct data d;

    printf("d.a: &d: %p, &d.a: %p, &d.a - &d: %ld, offsetof(a): %ld\n",
        &d, &d.a, (void*)&d.a - (void*)&d, offsetof(struct data, a));

    printf("d.b: &d: %p, &d.b: %p, &d.b - &d: %ld, offsetof(b): %ld\n",
        &d, &d.b, (void*)&d.b - (void*)&d, offsetof(struct data, b));

    printf("d.list: &d: %p, &d.list: %p, &d.list - &d: %ld, "
           "offsetof(list): %ld\n",  &d, &d.list,
        (void*)&d.list - (void*)&d, offsetof(struct data, list));
}
```

```
d.a:    &d: 0x7ffe186697d0, &d.a:    0x7ffe186697d0, &d.a - &d:    0, offsetof(a):    0
d.b:    &d: 0x7ffe186697d0, &d.b:    0x7ffe186697d4, &d.b - &d:    4, offsetof(b):    4
d.list: &d: 0x7ffe186697d0, &d.list: 0x7ffe186697d8, &d.list - &d: 8, offsetof(list): 8
```

Linux Kernel’s Linked List

:to access the container

```
//A simple version of container_of
#define container_of_1(ptr, type, member) (                         \
    (type *)( (char*)ptr - offsetof(type, member) )                 )


//Type-check: if ptr is actually &type.member type
#define container_of(ptr, type, member) ({                          \
    const typeof( ((type*) 0)->member ) *__mptr = (ptr);            \
    (type *)( (char*)__mptr - offsetof(type, member) );             })
```

Linux Kernel's Linked List

:to access the container

```c
int main() {
    struct data d;

    printf("&d: %p, container_of(a): %p\n",
        &d, container_of(&d.a, struct data, a));

    printf("&d: %p, container_of(b): %p\n",
        &d, container_of(&d.b, struct data, b));

    printf("&d: %p, container_of(list): %p\n",
        &d, container_of(&d.list, struct data, list));
}
```

```
&d: 0x7ffe186697d0, container_of(a):    0x7ffe186697d0
&d: 0x7ffe186697d0, container_of(b):    0x7ffe186697d0
&d: 0x7ffe186697d0, container_of(list): 0x7ffe186697d0
```

Linux Kernel's Linked List : manipulating a linked list

- `LIST_HEAD(listhead)`
  - Defines a linked list head variable named listhead
- `INIT_LIST_HEAD(&list)`
  - Initializes list, a list_head type variable
- `list_add(sruct list_head* new, struct list_head *head)`
  - Inserts new immediately after the head node
- `list_add_tail(sruct list_head* new, struct list_head *head)`
  - Adds new at the end of the list (immediately before head)
- `list_del(struct list_head *entry)`
  - Removes the element entry from the list

Linux Kernel’s Linked List

:to access the container

```
#include <linux/list.h>
#include <linux/slab.h>
void list_alloc(void) {
    LIST_HEAD(head);    //define a list head

    //allocate and initialize a list entry
    struct data *d = kmalloc(sizeof(struct data), GFP_KERNEL);
    d->a = 0;
    d->b = 0;
    INIT_LIST_HEAD(&d->list);

    //add the entry to the list
    list_add(&d->list, &head);

    //remove the entry from the list
    list_del(&d->list);

    //free the memory
    kfree(d);
}
```

Linux Kernel's Linked List
: manipulating a linked list

- `list_for_each(pos, head)`
  - Traversing a linked list

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

- `list_for_each_entry(pos, head, member)`
  - Traversing a linked list with the container

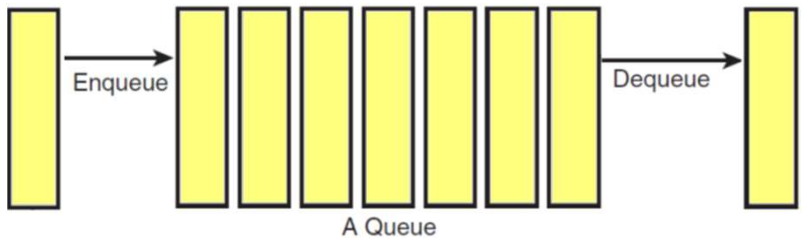
```
#define list_for_each_entry(pos, head, member)                   \
    for (pos = list_first_entry(head, typeof(*pos), member);  \
         &pos->member != (head);                              \
         pos = list_next_entry(pos, member))
```

```c
#include <linux/list.h>
#include <linux/slab.h>
#include <linux/kernel.h>
#include <linux/syscalls.h>
struct cdata {
    char c;
    struct list_head list;
};

void test_list(void) {
    LIST_HEAD(head);
    struct cdata *cd, *next;
    char *s = "Hello World!";

    //build a list
    for(; *s; s++) {
        struct cdata *cd = kmalloc(
                    sizeof(struct cdata),
                    GFP_KERNEL);
        cd->c = *s;
        INIT_LIST_HEAD(&cd->list);
        list_add(&cd->list, &head);
    }

    //traverse the list
    list_for_each_entry(
        cd, &head, list) {
        printk("%c\n", cd->c);
    }

    //empty the list
    list_for_each_entry_safe(
        cd, next, &head, list) {
        list_del(&cd->list);
        kfree(cd);
    }
}

SYSCALL_DEFINE0(data_structures) {
    printk("Test List\n");
    test_list();

    return 0;
}
```

Queues

- A buffer with the property that the First data In is the First data Out of the queue (FIFO)

- kfifo: Linux kernel's implementation of queue



kfifo

- `int kfifo_alloc(struct kfifo *fifo, unsigned int size, gfp_t gfp_mask)`
  - Creates a kfifo
  - size must be a power of two
- `static inline void kfifo_reset(struct kfifo *fifo)`
  - Removes all contents from the queue
- `void kfifo_free(struct kfifo *fifo)`
  - Destroys the queue

kfifo

- `unsigned int kfifo_in(struct kfifo *fifo, const void *from, unsigned int len)`
  - Enqueues data of length len byte to fifo
- `unsigned int kfifo_out(struct kfifo *fifo, const void *to, unsigned int len)`
  - Dequeues data of length len byte from fifo
- `unsigned int kfifo_out_peek(struct kfifo *fifo, const void *to, unsigned int len, unsigned offset)`
  - Reads data of length len byte from the offset position of fifo without removing them

kfifo

```c
#include <linux/kfifo.h>
#include <linux/kernel.h>
void test_queue(void) {
    struct kfifo fifo;
    char *s = "Hello World!", c;

    if(kfifo_alloc(&fifo, 1024, GFP_KERNEL)) // create kfifo
        return;

    for(; *s; s++)
        kfifo_in(&fifo, s, 1);              // enqueue

    while(kfifo_size(&fifo) > 0) {
        if(kfifo_out(&fifo, &c, 1) != 1)    // dequeue
            break;
        printk("%c\n", c);
    }

    kfifo_free(&fifo);                      // destroy kfifo
}
```

Maps

- A map, aka an associative array, is a collection of a key and a value associated with the key
  - Three basic operations
    - `Add(key, value),`
    - `Remove(key),`
    - `value = Lookup(key)`
- idr (id allocator)
  - Linux kernel's simple and efficient map data structure.
  - Provides a mapping from a unique id (UID) to a pointer

Idr (id allocator)

- Initializing an idr
  - `void idr_init(struct idr *idp)`
- Allocating a new UID
  - `int idr_alloc(struct idr *idp, void *ptr, int start, int end, gfp_t gfp_mask)`
    - Allocate a new UID in [start, end) and associate it with `ptr`.
    - When end is non-positive, it means the max
  - `int idr_preload(gfp_t gfp_mask)`
    - Allocates memory and disables preemption
  - `int idr_preload_end(void)`
    - Reenables preemption

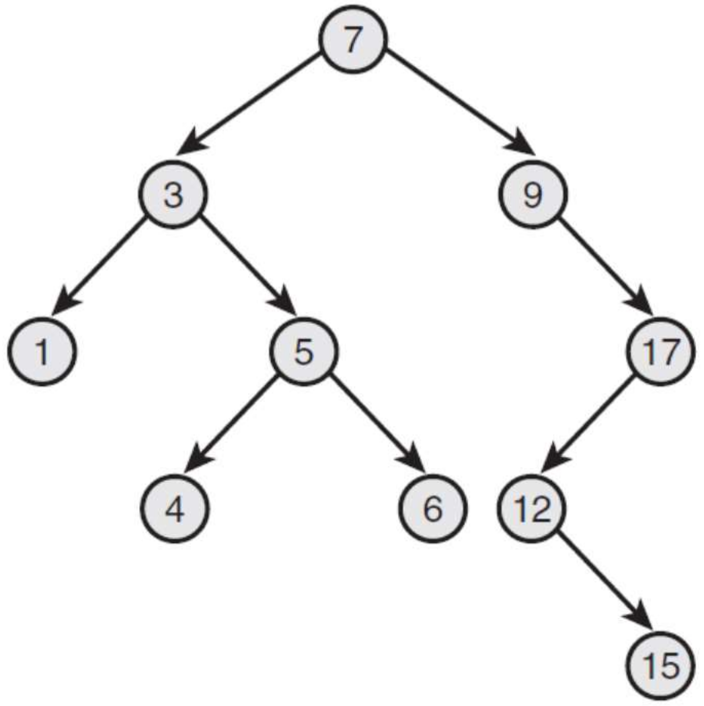
idr

- Looking up a UID
  - `void *idr_find(struct idr *idp, int id)`
- Removing a UID
  - `void idr_remove(struct idr *idp, int id)`
- Destroying an idr
  - `void idr_destroy(struct idr *idp)`

idr

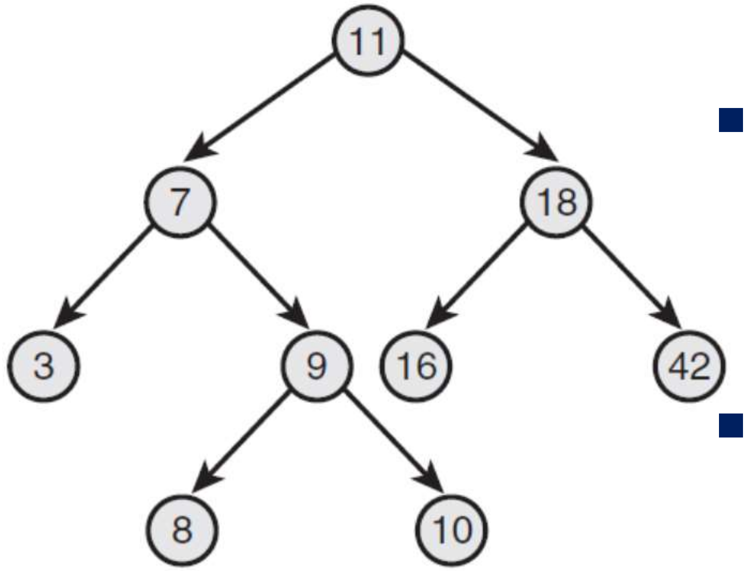
```
#include <linux/idr.h>
#include <linux/kernel.h>
#define HW  "Hello World!"
void test_idr(void) {
    struct idr map;
    const int n = sizeof(HW);
    char *s = HW;
    int ids[n], i;

    idr_init(&map);  //initialize idr

    for(i = 0; *s; s++, i++) {
        //allocate a new id and associate it with s
        ids[i] = idr_alloc(&map, s, 0/*start*/, n/*end*/, GFP_KERNEL);
    }

    while(i > 0) {
        i--;
        printk("%c\n", *(char*)idr_find(&map, ids[i])); //lookup id
        idr_remove(&map, ids[i]); //remove id
    }

    idr_destroy(&map); //destroy idr
}
```

Binary Trees



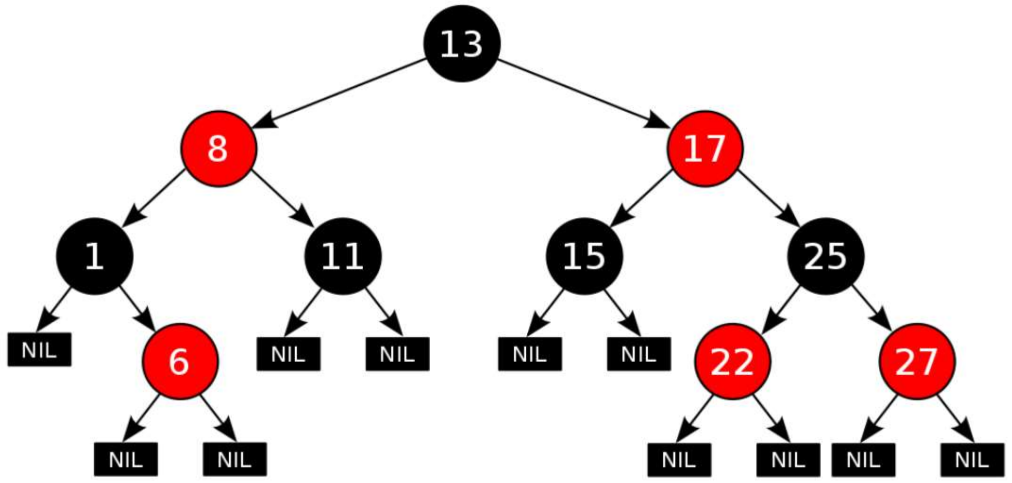
- Binary Search Tree
  - Left subtree of the root has nodes with values less than the root
  - Right subtree of the root has nodes with values larger than the root
  - All subtrees are also binary search trees

Balanced Binary Search Tree



- Depth of a node
  - # of parent nodes to the root node
- Height of a tree
  - The depth of the deepest node in the tree
- Balanced binary search tree
  - A binary search tree in which the depth of all leaves differs by at most one

Red-Black Tree



- Kind-of-balanced tree
  - All nodes are either **red** or **black**
  - Leaf nodes are **black**
  - Leaf nodes do not contain data
  - All non-leaf nodes have two children
  - If a node is **red**, both of its children are **black**
  - Every path from a node to a leaf node has the same number of **black** nodes
- The deepest leaf node has a depth of no more than double that of the shallowest leaf

rbtree

- To initialize a rbtree
  - `struct rb_root = RB_ROOT;`
- To traverse the tree
  - `struct rb_node *rb_first(struct rb_root *tree);`
  - `struct rb_node *rb_last(struct rb_root *tree);`
  - `struct rb_node *rb_next(struct rb_node *node);`
  - `struct rb_node *rb_prev(struct rb_node *node);`
- To get the containing element
  - `rb_entry(pointer, type, member)`

rbtree

- To insert a node
  - Insert a new node at a given spot
    - `void rb_link_node(struct rb_node *new_node, struct rb_node *parent, struct rb_node **link)`
  - Rebalance the tree
    - `void rb_insert_color(struct rb_node *new_node, struct rb_root *tree)`
- To remove a node from a tree
  - `void rb_erase(struct rb_node *victim, struct rb_root *tree)`
- Removing nodes while traversing the tree
  - `rbtree_postorder_for_each_entry_safe(pos, n/*temporary storage*/, root, field)`

rbtree

```c
#include <linux/rbtree.h>
#include <linux/slab.h>
#include <linux/kernel.h>

struct kdata{
    char k;
    struct rb_node node;
};

struct kdata* rbfind(struct rb_root *root, char c) {
    struct rb_node *n = root->rb_node;
    while(n) {
        struct kdata *d = rb_entry(n, struct kdata, node);
        if(c > d->k)
            n = n->rb_right;
        else if(c < d->k)
            n = n->rb_left;
        else
            return d;
    }
    return NULL;
}
```

rbtree

```c
void rbinsert(struct rb_root *root, struct kdata *kd) {
    struct rb_node **p = &root->rb_node;
    struct rb_node *parent = NULL;
    char c = kd->k;

    while(*p) {
        struct kdata *d = rb_entry(*p, struct kdata, node);
        parent = *p;
        if(c > d->k)
            p = &((*p)->rb_right);
        else //if(c <= d->k)
            p = &((*p)->rb_left);
    }

    rb_link_node(&kd->node, parent, p);
    rb_insert_color(&kd->node, root);
}
```

```
void test_rbtree(void) {
    struct rb_root root = RB_ROOT;
    struct rb_node *n;
    struct kdata *kd, *t;
    char *s = "Hello World!";

    printk("%s\n", __FUNCTION__);
    for(; *s; s++) {
        kd = kmalloc(sizeof(struct kdata), GFP_KERNEL);
        kd->k = *s;
        rbinsert(&root, kd);
    }

    for(n = rb_first(&root); n; n = rb_next(n)) {
        kd = rb_entry(n, struct kdata, node);
        printk("%c\n", kd->k);
    }

    rbtree_postorder_for_each_entry_safe(kd, t, &root, node) {
        rb_erase(&kd->node, &root);
        kfree(kd);
    }
}
```