Kernel Data Structures

Xiaoguang Wang

UIC COMPUTER SCIENCE

Recap

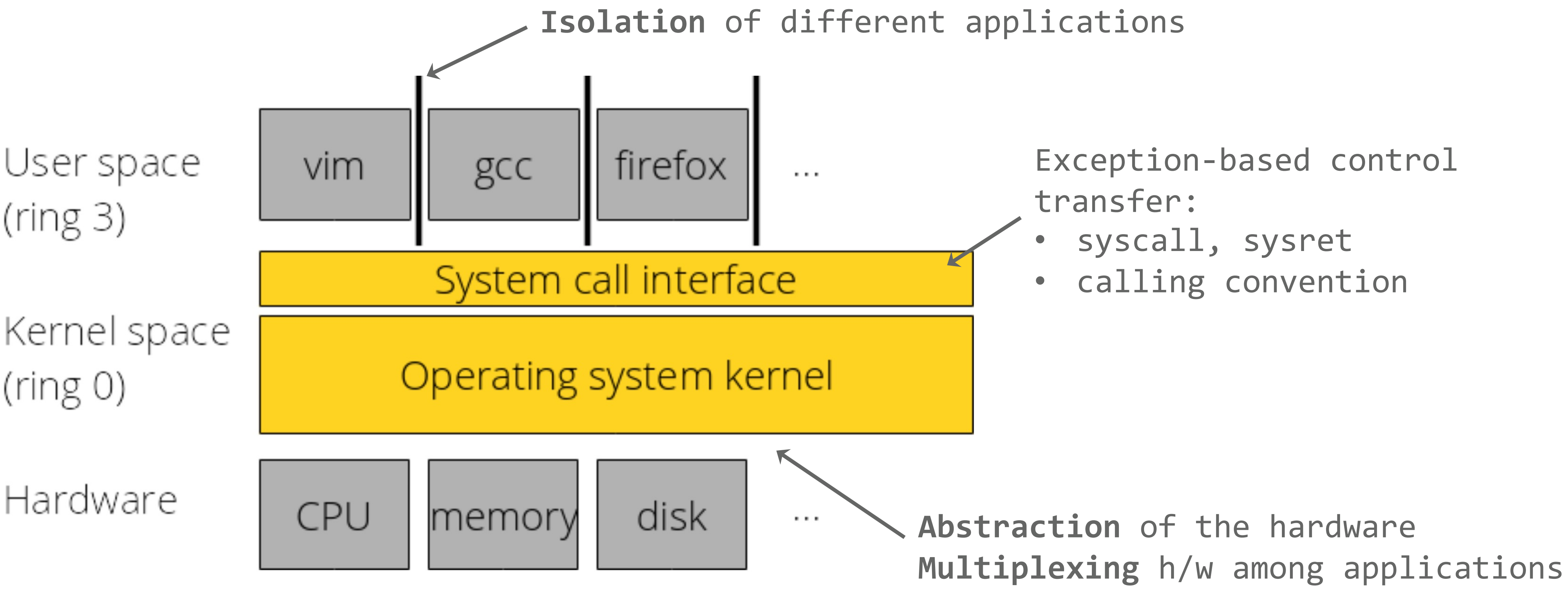
Tools

- git, tig, make, cscope, ctags, vim, emacs, tmux, ssh, etc.

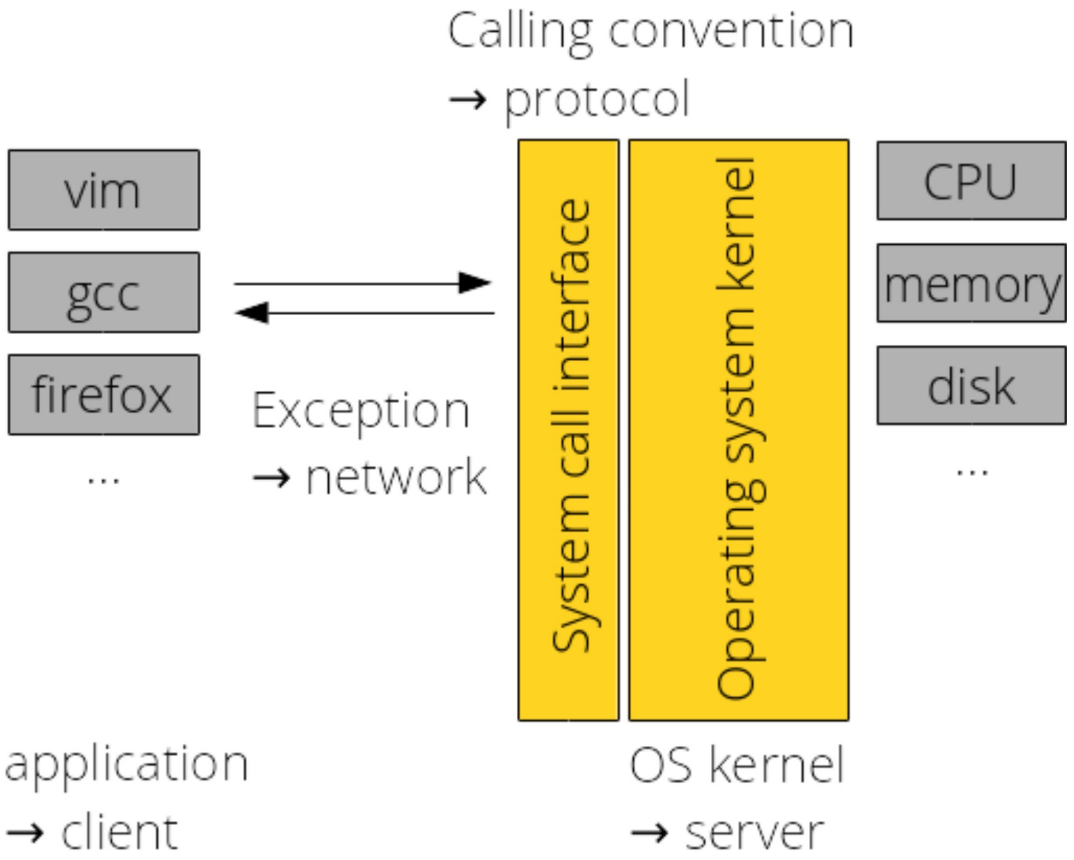
System calls

- isolation, x86 ring architecture

What is the operating system (again)?



Operating system in a different angle



Client/server programs

- exception: network request

sync network request

➔ async network request (non-block, no need to wait for the result)

FlexSC: Exception-less system calls

Why?

- The cost of system call = direct cost + indirect cost

How?

- Exception-less system call
- A syscall page to pass parameters
- A syscall thread
- Dynamic core specialization

"Exit-less" design used in other cases

Event-driven server (NGINIX)

- Exception-less System Calls for Event-driven Servers, USENIX ATC 2011

Virtualization

- ELI: bare-metal performance for I/O virtualization, ASPLOS 2012

SGX (Intel Software Guard eXtension)

- Eleos: ExitLess OS Services for SGX Enclaves, EuroSys 2017

Outline: Kernel Data Structures

Linked list

Hash table

Red-black tree

Radix tree

Bitmap

Why data structure is 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

Why data structure is important?

Help you understand the real Linux kernel code

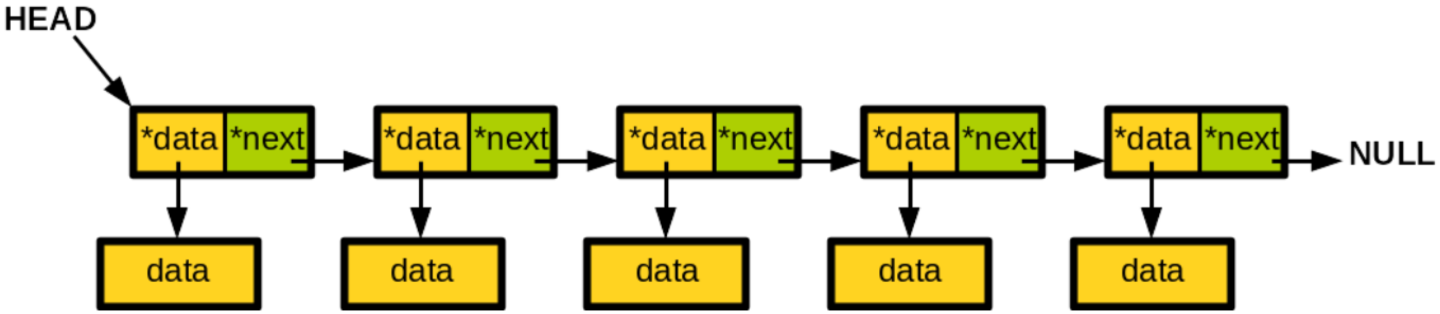
What’s going on with these code?

```
static struct kvm_task_sleep_node *_find_apf_task(struct
                                                   u32 tok
{
        struct hlist_node *p;

        hlist_for_each(p, &b->list) {
                struct kvm_task_sleep_node *n =
                        hlist_entry(p, typeof(*n), link);
                if (n->token == token)
                        return n;
        }

        return NULL;
}
```

```
/**
 * Disable HW interrupt  for HPD and HPDRX only since FLIP and VBLANK
 * will be disabled from manage_dm_interrupts on disable CRTC.
 */
for (src = DC_IRQ_SOURCE_HPD1; src <= DC_IRQ_SOURCE_HPD6RX; src++) {
        hnd_list_l = &adev->dm.irq_handler_list_low_tab[src];
        hnd_list_h = &adev->dm.irq_handler_list_high_tab[src];
        if (!list_empty(hnd_list_l) || !list_empty(hnd_list_h))
                dc_interrupt_set(adev->dm.dc, src, false);

        DM_IRQ_TABLE_UNLOCK(adev, irq_table_flags);

        if (!list_empty(hnd_list_l)) {
                list_for_each_safe (entry, tmp, hnd_list_l) {
                        handler = list_entry(
                                entry,
                                struct amdgpu_dm_irq_handler_data,
                                list);
                        flush_work(&handler->work);
                }
        }
        DM_IRQ_TABLE_LOCK(adev, irq_table_flags);
}
```

Singly linked list (CS101)

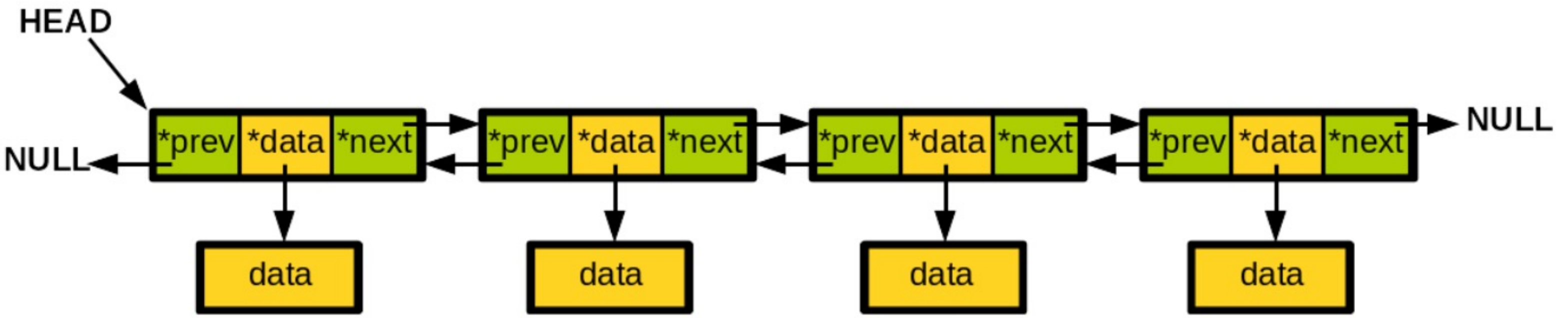
```c
struct my_list_element {
  void *data;                          /* void pointer to point on a generic data */
  struct my_list_element *next;        /* pointer to a next element */
};
```



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

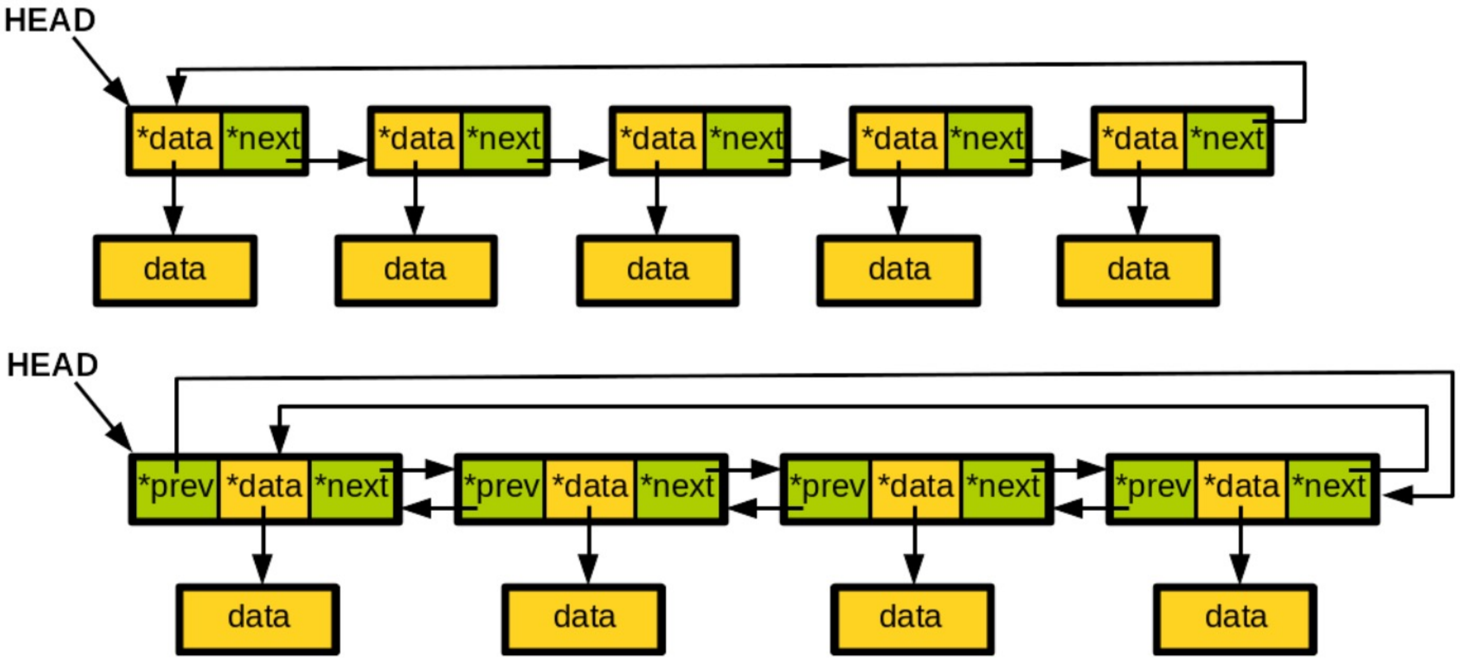
Doubly linked list (CS101)

```
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 */
};
```



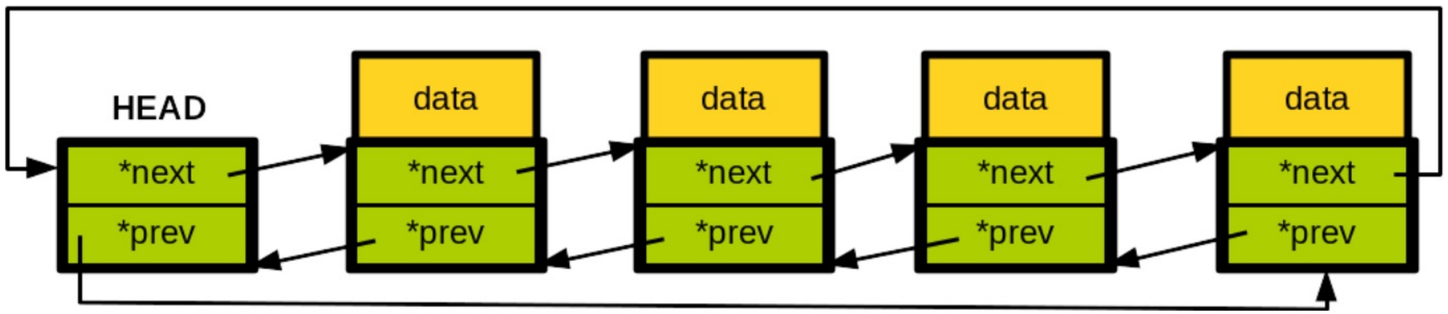
- Starts from HEAD and terminates at NULL
- Traverses forward and **backward**
- When empty, HEAD is NULL

Circular linked list (CS101)



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

Linux linked list



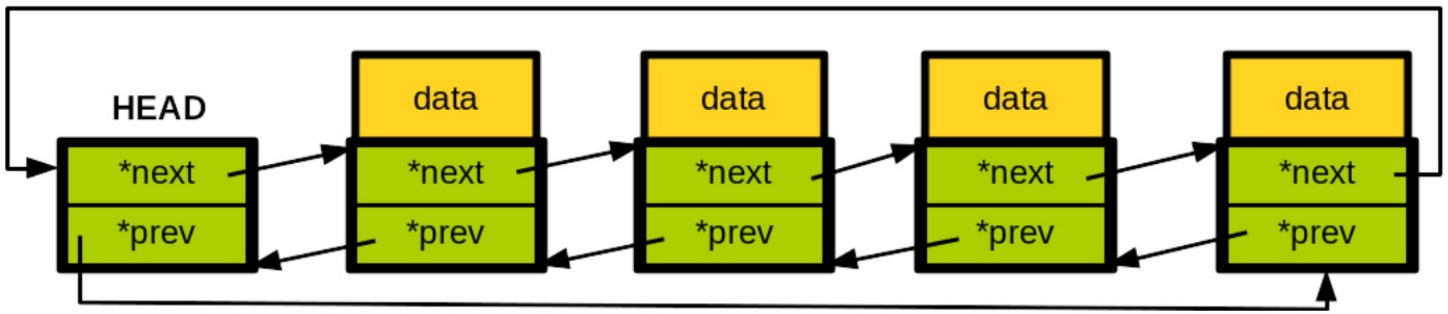
Starts from `HEAD` and terminates at `HEAD`

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

Linux linked list



A circular doubly linked list

Two differences from the typical design

1. Use a sentinel node as a list header
2. Embed a linked list node in the data structure

Linux linked list

Definition: `include/linux/types.h` **APIs/impl**: `include/linux/list.h` (kern v6.1)

```
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 price_in_dollars;
};

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

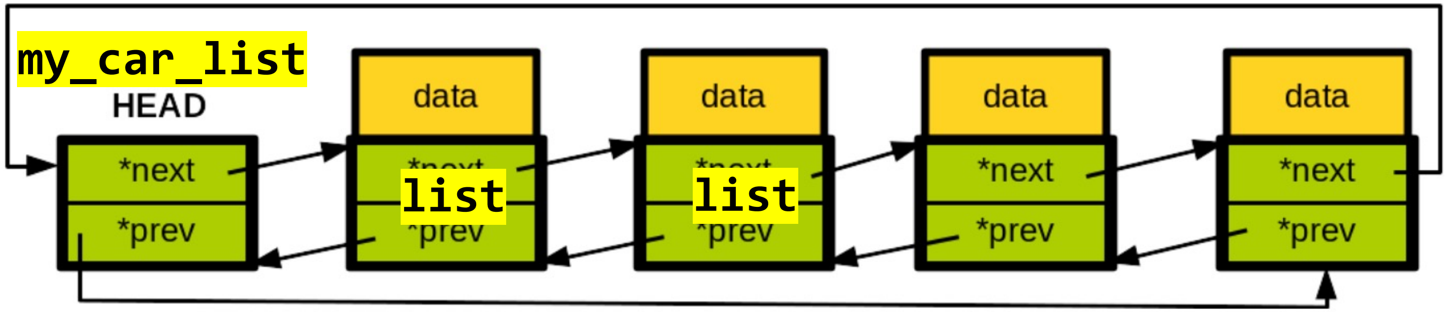
The traditional way of implementing a doubly linked list:

```
struct car {
    struct car * prev;
    struct car * next;
    unsigned int max_speed;
    unsigned int price_in_dollars;
};
```

Linux linked list

Definition: `include/linux/types.h` **APIs/impl**: `include/linux/list.h` (kern v6.1)

```
struct list_head {     /* kernel linked list data structure */
    struct list_head *next, *prev;
};
struct car {
    struct list_head list; /* add list_head in
    unsigned int max_speed; /* put data di
    unsigned int price_in_dollars;
};
struct list_head my_car_list;     /* HEAD is also list_head */
```



Linux linked list

What's the benefit of having a `list_head` struct in the data structure?

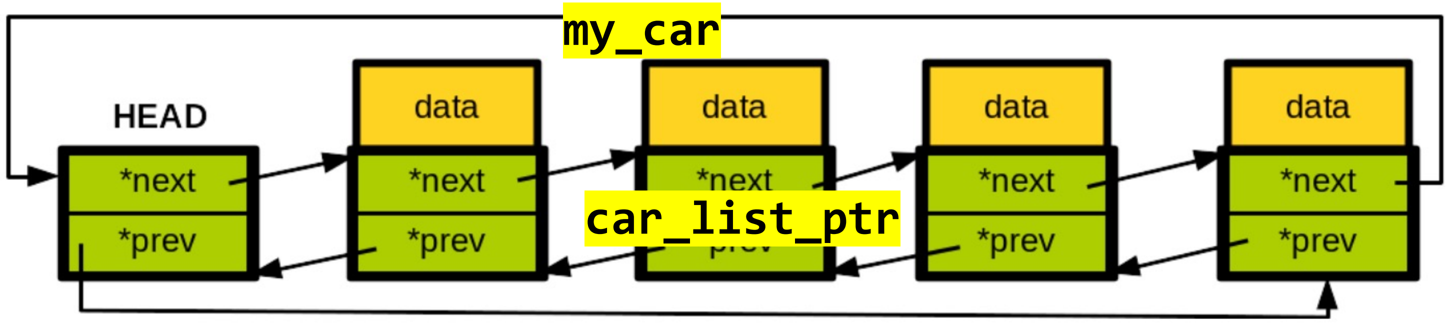
- List APIs only need to accept the `list_head` structures
  - e.g., `list_add(struct list_head *new, struct list_head *head)`
  - manipulate the `list_head` object
- If we want to find the parent structure, use `container_of()`

Getting a data element from `list_head`

How to get the pointer of the containing data structure (`struct car`) from its list?

- Use `list_entry(ptr, type, member)`
- Just a pointer arithmetic

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




```
struct list_head {
    struct list_head *next, *prev;
};
struct car {
    struct list_head list;
    unsigned int max_speed;
    unsigned int price_in_dollars;
};
struct list_head my_car_list;
```

Getting a data element from `list_head`

How to get the pointer of `struct car` from its list?

- Use `list_entry(ptr, type, member)`

```c
struct car *my_car = list_entry(car_list_ptr, struct car, list);
/* list_entry - get the struct for this entry  */
#define list_entry(ptr, type, member) container_of(ptr, type, member)
#define container_of(ptr, type, member) ({ \
        void *__mptr = (void *)(ptr); \
        static_assert(..pointer type mismatch check...); \
        (type *)( (char *)__mptr - offsetof(type, member) );})
```

Define a linked list

Static (compile-time) definition:

```
struct car my_car {
  .max_speed = 150,
  .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 */
```

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)
```

Define a linked list

Dynamic (runtime) definition, most commonly used:

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

Initializing a `list_head`

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
        WRITE_ONCE(list->next, list);
        WRITE_ONCE(list->prev, list);
}
```

Manipulate 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);
```

Manipulate a list: O(1)

Examples:

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

list_add(&my_car->list, &my_car_list);
list_del(&my_car->list);
```

Iterate over a list: O(n)

`list_for_each(p, head)`, `list_for_each_entry(p, head, mem)`

```
/**
 * 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; !list_is_head(pos, (head)); pos = pos->next)
```

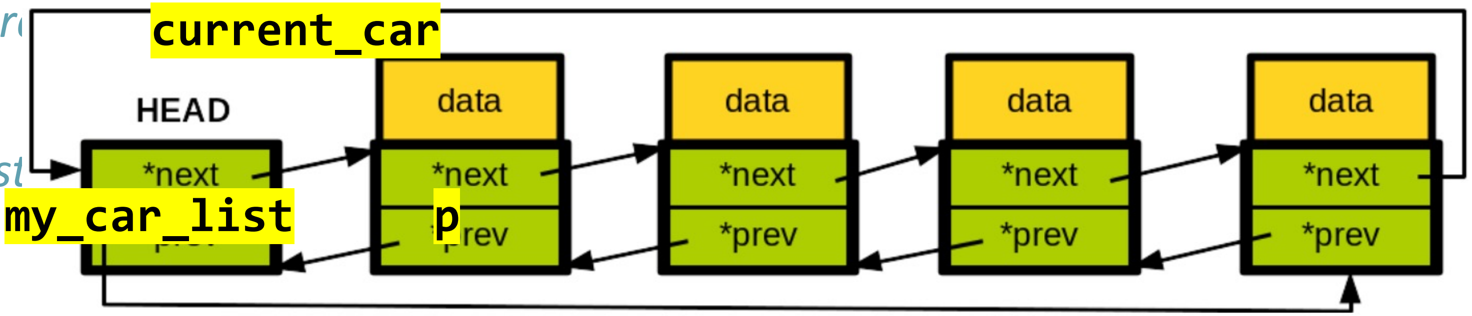
```
/**
 * list_for_each_entry  -       iterate over list of given type
 * @pos:        the type * to use as a loop cursor.
 * @head:       the head for your list.
 * @member:     the name of the list_head within the struct.
 */
#define list_for_each_entry(pos, head, member)                         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

Iterate over a list: O(n)

```
/* Temporary variable needed to iterate: */
struct list_head p;
/* This will point to 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: %ld\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: %ld\n", current_car->price_in_dollars);
}
```

Iterate over a list: O(n)

```
/* Temporary variable needed to iter
struct list_head p;
/* This will point to the actual data st
struct car *current_car;
```




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

Iterating while removing

```
#define list_for_each_safe(pos, next, head) ...
#define list_for_each_entry_safe(pos, next, head, member) ...
/* This will point to 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: %ld\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

Linked list usage in the Linux 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

UIC COMPUTER SCIENCE

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)

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

UIC COMPUTER SCIENCE

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)

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

Linux hash table

```c
/* 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 |   |-->"Xiaoguang"
   +---+
 3 |   |
   +---+
```

Linux hash table API

```
/* Define a hash table with 2^bits buckets   */
#define DEFINE_HASHTABLE(name, bits) ...
/* hash_init - initialize a hash table */
#define hash_init(hashtable) ...
/* hash_add - add an object to a hash table */
#define hash_add(hashtable, node, key) ...
```

Linux hash table API

```
/* hash_for_each - iterate over a hashtable */
#define hash_for_each(name, bkt, obj, member) ...
```

An integer to use as bucket loop cursor

The hash table

The type * to use as a loop cursor for each entry

The name of the hlist_node within the struct

```
         name
        +----+
bkt  0  |    |-->"John"<-->"Kim"
        +----+
     1  |    |-->"Josh"<-->”Lisa"
        +----+
     2  |    |-->"Xiaoguang"
        +----+
     3  |    |
        +----+
```

Linux hash table API

```
/* iterate over all possible objects hashing to the same bucket*/
#define hash_for_each_possible(name, obj, member, key) ...

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

/* hash_del - remove an object from a hashtable*/
void hash_del(struct hlist_node *node);
```

Linux hash table example

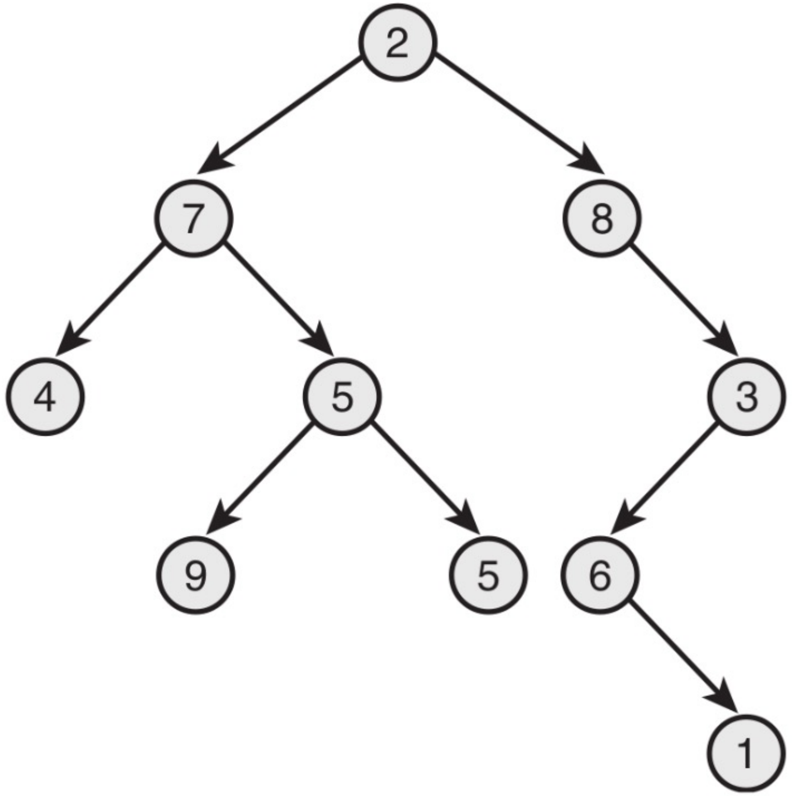
Network, device drivers, file systems, etc.

```
struct xfrm_state *mlx5e_ipsec_sadb_rx_lookup(struct mlx5e_ipsec *ipsec,
                                              unsigned int handle)
{
        struct mlx5e_ipsec_sa_entry *sa_entry;
        struct xfrm_state *ret = NULL;

        rcu_read_lock();
        hash_for_each_possible_rcu(ipsec->sadb_rx, sa_entry, hlist, handle)
                if (sa_entry->handle == handle) {
                        ret = sa_entry->x;
                        xfrm_state_hold(ret);
                        break;
                }
        rcu_read_unlock();

        return ret;
}
```

Binary tree

Nodes have zero, one, or two children

Root has no parent; other nodes have one



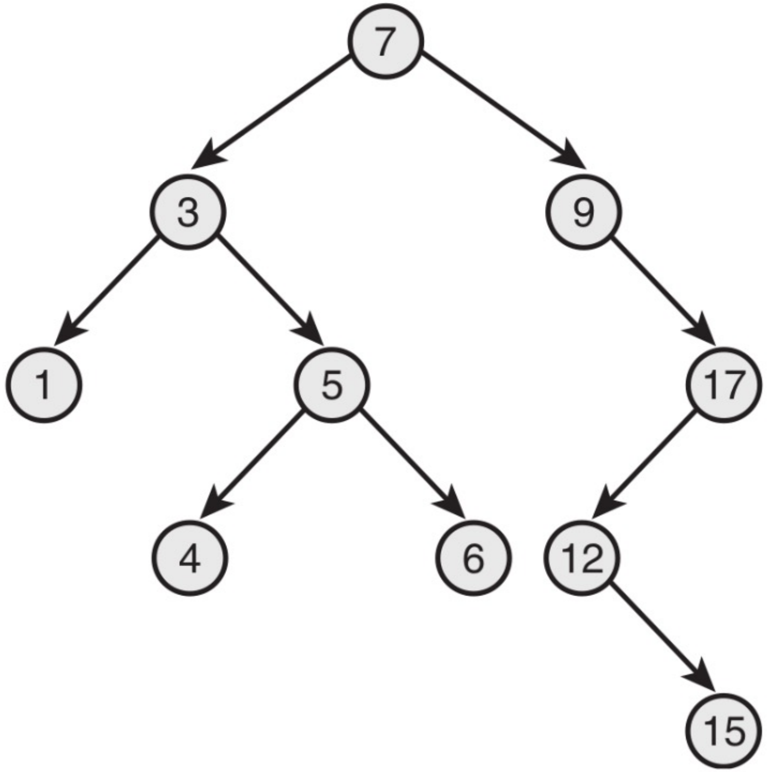
Binary search tree

It's a binary tree

Left children < parent

Right children > parent

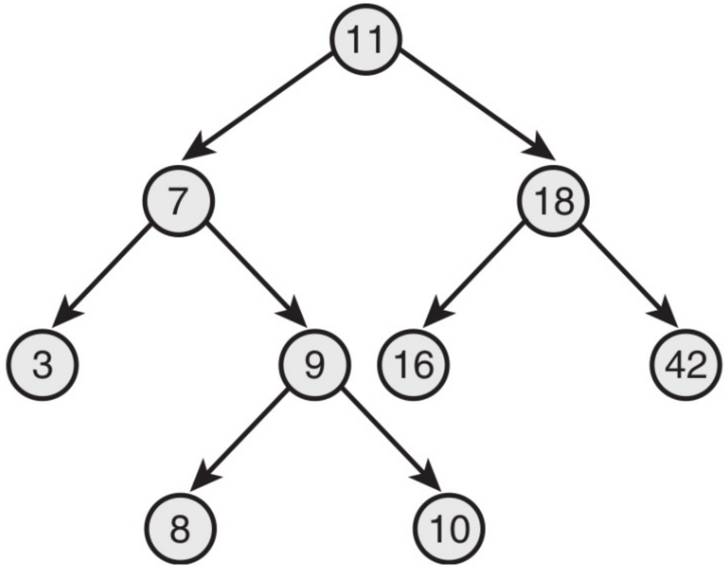
Search and ordered traversal are efficient



Balanced binary search tree

Depth of all leaves differ by at most one

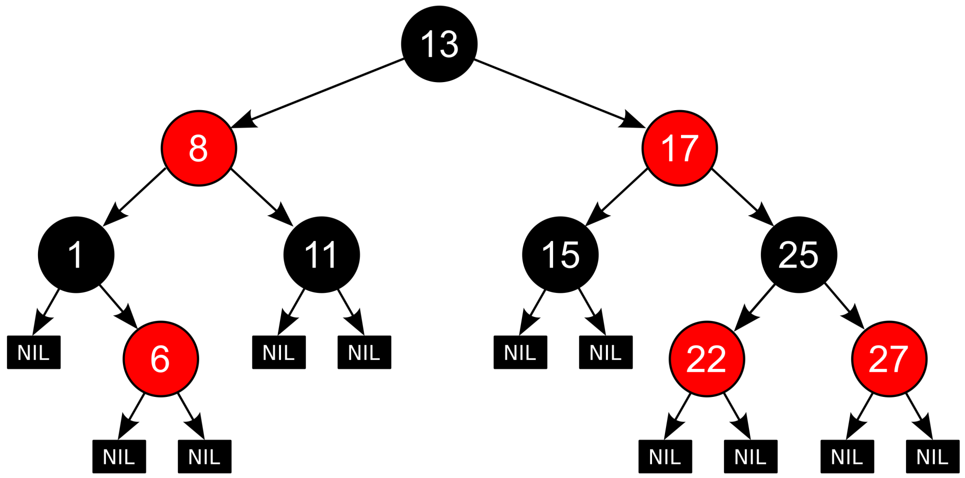
Puts a boundary on the worst-case operations



red-black tree

A type of self-balancing binary search tree

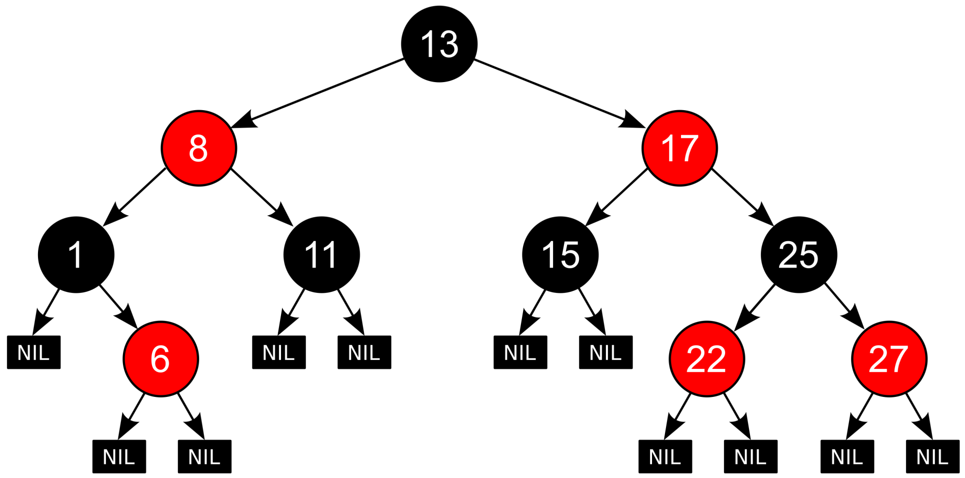
- Nodes: red or black
- Leaves: black, no data



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)

```
struct rb_node {                                  /* include/linux/rbtree.h,  lib/rbtree.c */
  unsigned long __rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;
};
struct rb_root {                                  /* Root of a rbtree */
  struct rb_node *rb_node;
};
#define RB_ROOT (struct rb_root) { NULL, }
#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)

```
/* 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))

/* 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 *);
```

Linux red-black tree (or rbtree)

```c
/* insert node into tree */
static __always_inline void
rb_add(struct rb_node *node, struct rb_root *tree,
      bool (*less)(struct rb_node *, const struct rb_node *));
/* find key in tree tree */
static __always_inline struct rb_node *
rb_find(const void *key, const struct rb_root *tree,
      int (*cmp)(const void *key, const struct rb_node *));
```

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`

Design patterns of kernel data structures

Embedding its pointer structure

- `list_head`, `hlist_node`, `rb_node`
- Programmers have full control of placement of fields in the structure (put fields closer 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

A toolbox rather than a complete solution

- None of Linux list, hash table, and rbtree provides a search function
- Build your own using the given primitives

Caller locks

- Choose to have the caller take locks

Further reading

LKD3 – chapter 6 (kernel data structures)

[How does the kernel implement Hashtables?]

[LWN: A generic hash table]

[LWN: Tree II: red-black trees]

UIC COMPUTER SCIENCE

Next lecture

- More kernel data structures
- Kernel modules

Feedback