Linux kernel data structure

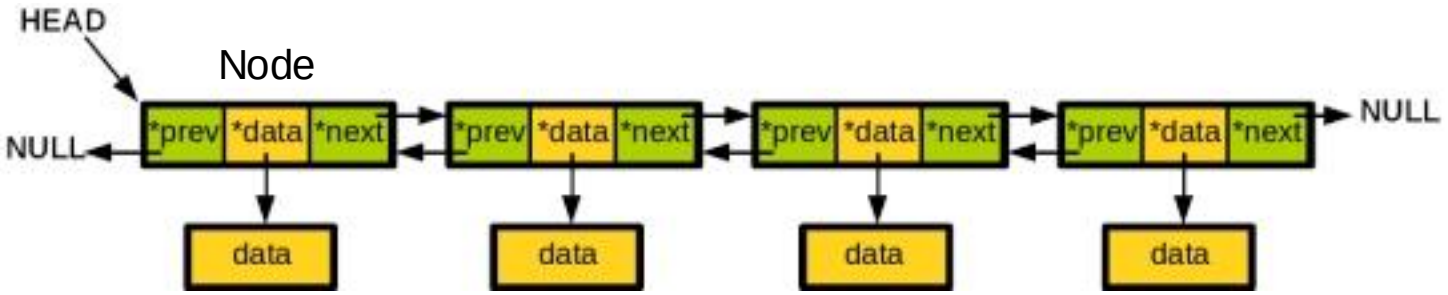
Linked List

Practical Class 10

Systems and Storage Laboratory
Department of Computer Science and Engineering
Chung-Ang University

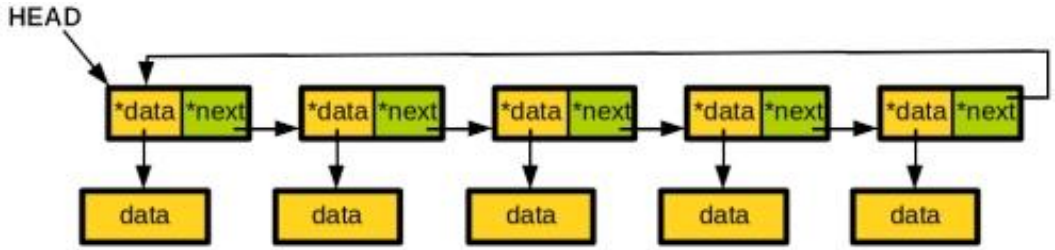
Doubly Linked List

- **Each node contains three fields**
  - Two link fields and one data field.
- **It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders**



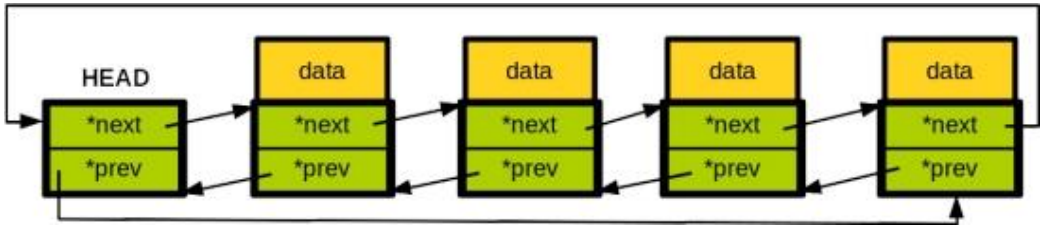
Circular Linked List

- **Circular linked list is a sequence of elements**
  - Every element has link to its next element
  - The last element has a link to the first element

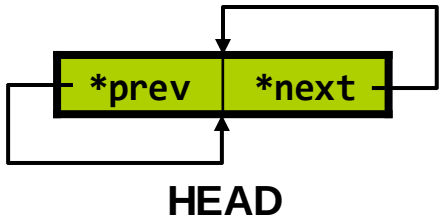


Circular Doubly Linked List

- **Linux 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

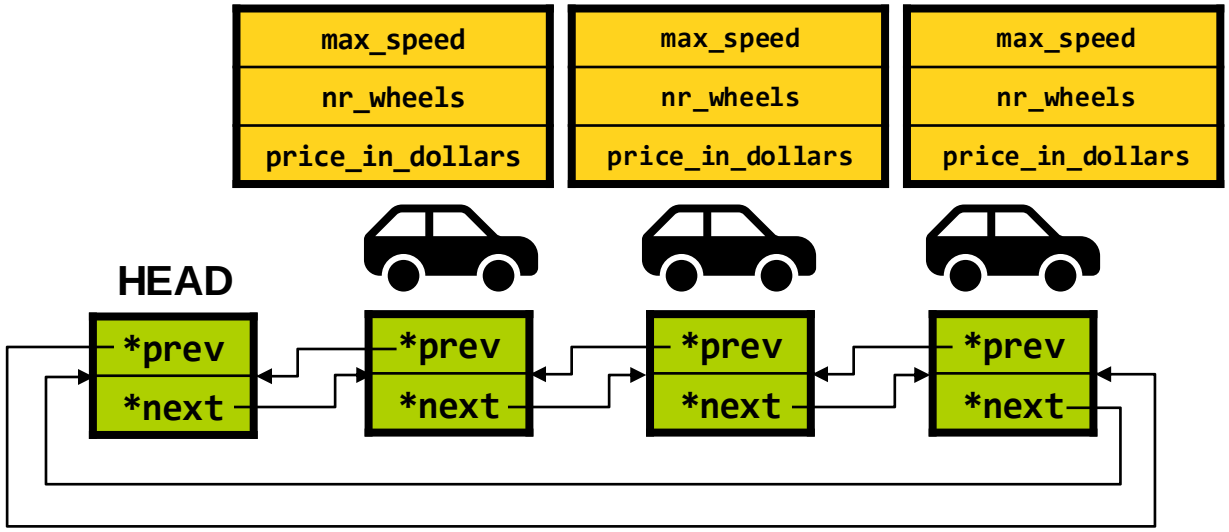


- **If the list contains only a sentinel node, then the list is circularly linked via the sentinel node.**
  - List is empty




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

Linux Linked List Example




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

Linked List APIs

- **Search (not safe while deleting entries)**
  - **list_for_each:** iterate over a list

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

    - **pos:** the `&struct list_head` to use as a loop cursor.
    - **head:** the head for your list.

  - **list_for_each_prev:** iterate over a list backwards

```
#define list_for_each_prev(pos, head) \
    for (pos = (head)->prev; pos != (head); pos = pos->prev)
```

Linked List APIs

- **Search (safe for deleting entries)**
  - **list_for_each_safe:** iterate over a list safe against removal of list entry

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

    - **pos:** the `&struct list_head` to use as a loop cursor.
    - **n:** another `&struct list_head` to use as temporary storage
    - **head:** the head for your list.

  - **list_for_each_prev_safe:** iterate over a list backwards safe against removal of list entry

```
#define list_for_each_prev_safe(pos, n, head) \
    for (pos = (head)->prev, n = pos->prev; \
         pos != (head); \
         pos = n, n = pos->prev)
```

Linked List APIs

❖ Grab list element

- **list_entry:** get the struct for this entry

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

  - **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.

- Ex)

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

struct list_head cars;
...
list_entry(&cars, struct car, list);
```

Linked List APIs

- **Search (not safe while deleting entries)**
  - **list_for_each_entry:** iterate over list of given type

```
#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))
```

    - **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.

  - **list_for_each_entry_reverse**: iterate backwards over list of given type.

```
#define list_for_each_entry_reverse(pos, head, member)          \
    for (pos = list_last_entry(head, typeof(*pos), member);     \
         &pos->member != (head);                                \
         pos = list_prev_entry(pos, member))
```

Linked List APIs

❖ Search (safe for deleting entries)

- **list_for_each_entry_safe:** iterate over list of given type safe against removal of list entry

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

  - **pos:** the type * to use as a loop cursor.
  - **n:** another type * to use as temporary storage.
  - **head:** the head for your list.
  - **member:** the name of the list_head within the struct.

- **list_for_each_entry_reverse**: iterate backwards over list of given type, safe against removal

```
#define list_for_each_entry_safe_reverse(pos, n, head, member)      \
    for (pos = list_last_entry(head, typeof(*pos), member),     \
        n = list_prev_entry(pos, member);                       \
         &pos->member != (head);                                \
         pos = n, n = list_prev_entry(n, member))
```

Linked List APIs

Insert

- **list_add:** Insert a new entry after the specified head
  - Good for implementing stacks

```
void list_add(struct list_head *new, struct list_head *head);
```

- **list_add_tail:** Insert a new entry before the specified head
  - Useful for implementing queues

```
void list_add_tail(struct list_head *new, struct list_head *head);
```

Delete

- **list_del:** Delete a list entry
  - You still have to take care of the memory deallocation if needed

```
void list_del(struct list_head *entry);
```

Linked List APIs

Move

- **list_move:** Delete from one list and add as another's head

```
void list_move(struct list_head *list, struct list_head *head);
```

- **list_move_tail:** Delete from one list and add as another's tail

```
void list_move_tail(struct list_head *list, struct list_head *head);
```

Join

- **list_splice:** Join two lists
  - Merge a list to the specified head

```
void list_splice(const struct list_head *list, struct list_head *head);
```

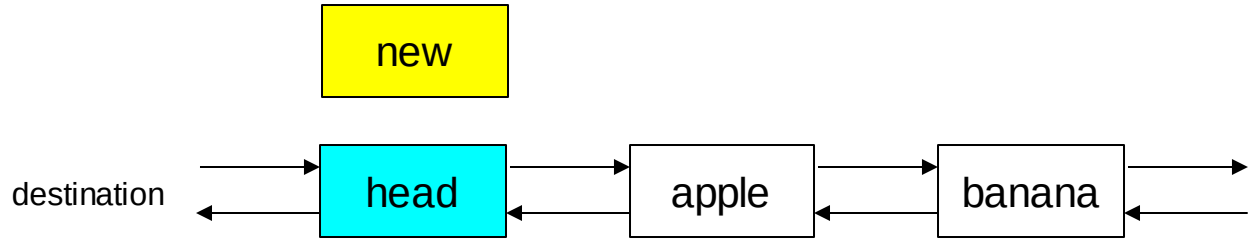
list_add Explained

- **Insert a new entry after the specified head**

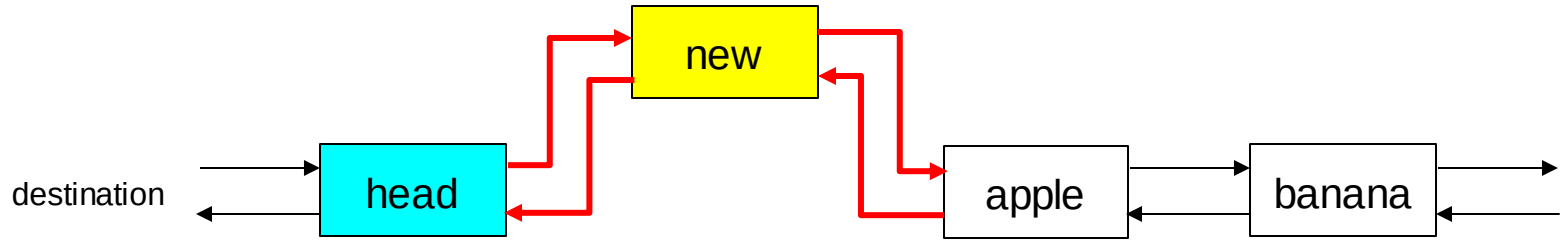
```
void list_add(struct list_head *new, struct list_head *head);
```

New entry to add — *Destination list head*

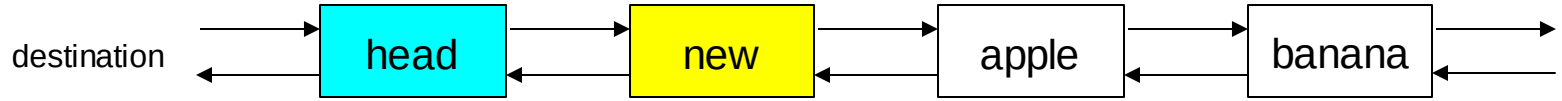
1. Initial State



2. Insert a new entry between `head` and `head->next`



3. Result



list_del Explained

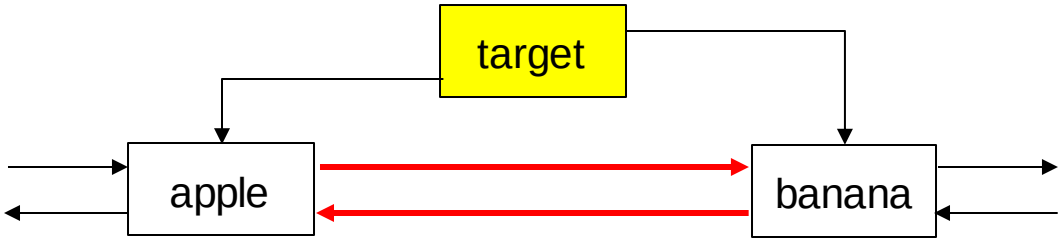
❖ Delete a list entry

```
void list_del(struct list_head *entry);
```

Target entry to delete from its list

1. Initial State

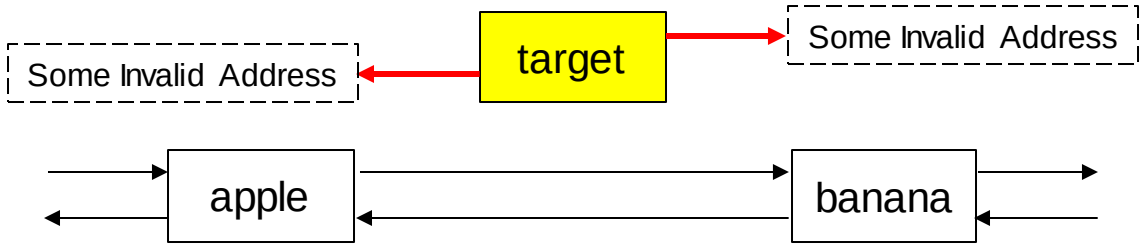
2. Detach the target entry from its list



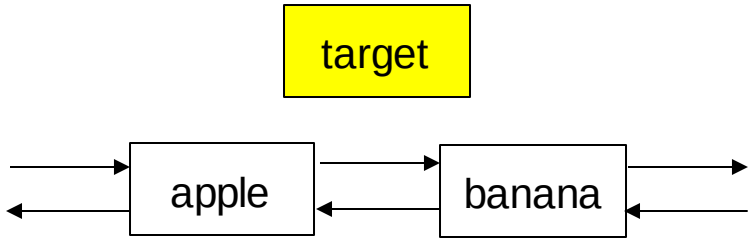
list_del Explained

❖ Delete a list entry

3. Make the entry forget about its list



4. Result
   - You have to free the entry from the memory by yourself



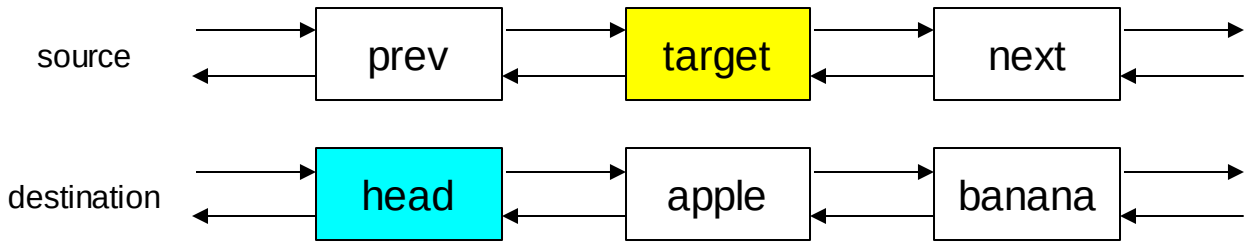
list_move Explained

❖ Delete from one list and add as another's head

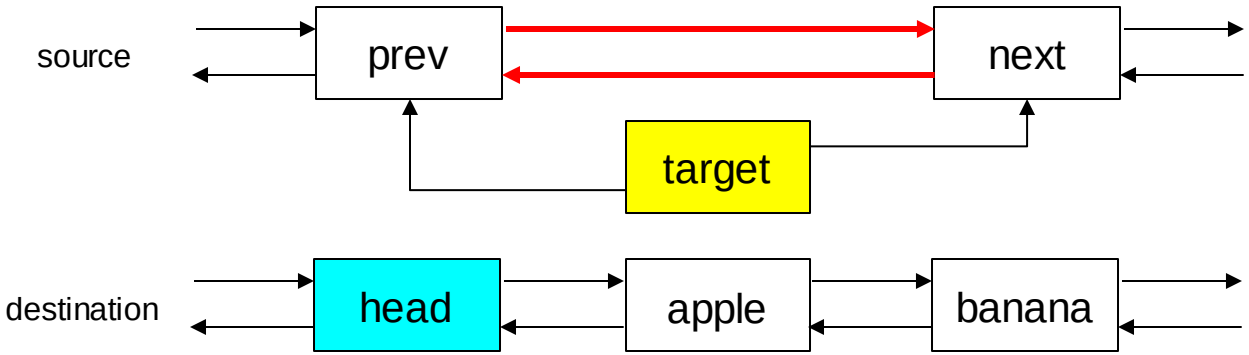
```
void list_move(struct list_head *list, struct list_head *head);
```

Source list's target entry — *Destination list head*

1. Initial State



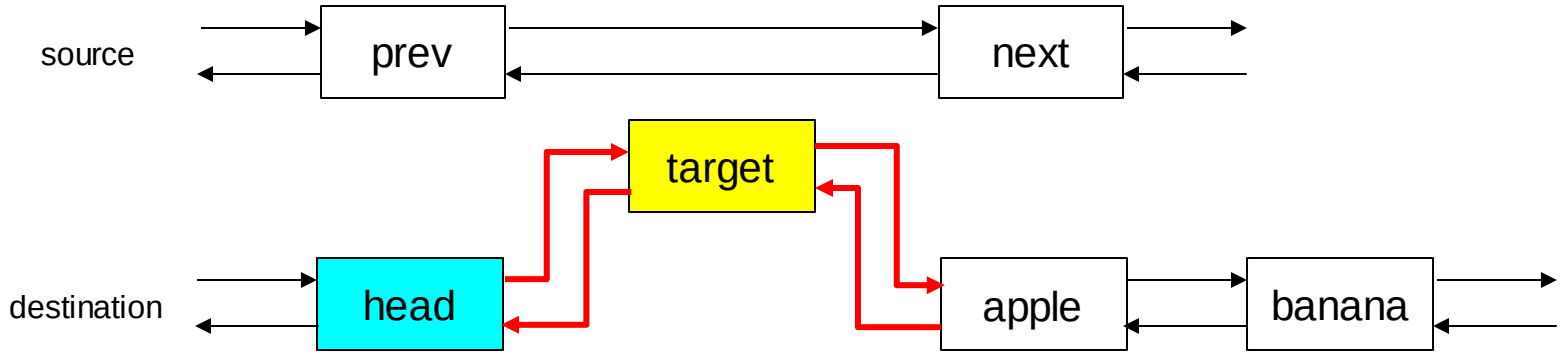
2. Detach target from the source list



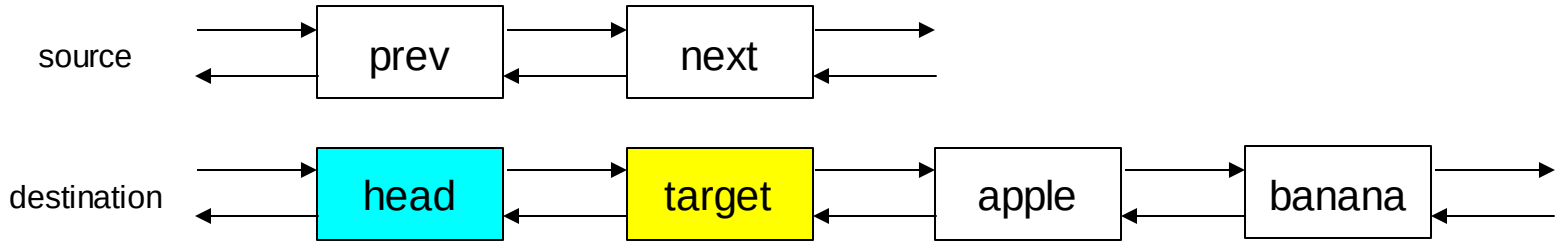
list_move Explained

- **Delete from one list and add as another's head**

3. Add target to the destination list



4. Result



Linked List Example

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

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

    /* list element add */
    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->entry,&my_list);
    }

    struct my_node *current_node = NULL;
    /* check list */
    list_for_each_entry(current_node, &my_list, entry) {
        printk("current value : %d\n", current_node->data);
    }
```

Linked List Example

```
    /* check list reverse*/
    list_for_each_entry_reverse(current_node, &my_list, entry) {
        printk("current value : %d\n", current_node->data);
    }

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

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

Linked List Usage

- **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 ...