EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 17 – Linked List in Linux Kernel Part 1**

Device Drivers

Linux Device Driver Tutorial Part 17 – Linked List in Linux Kernel Part 1

X

provide easy and practical examples that anyone can understand. In our previous tutorials, we have seen workqueue. So this is the Linux Device Driver Tutorial Part 17 – Linked List in Linux Kernel Part 1.

Post Contents [hide]

X

Linux Device Driver Tutorial Part 17 – Linked List in Linux Kernel

Introduction about Linked List

A linked list is a data structure that consists of a sequence of nodes. Each node is composed of two fields: the **data field** and the **reference field** which is a pointer that points to the next node in the sequence.

X

Each node in the list is also called an element. A **head** pointer is used to track the first element in the linked list, therefore, it always points to the first element.

X

The elements do not necessarily occupy contiguous regions in memory and thus need to be linked together (each element in the list contains a pointer to the *next* element).

Advantages of Linked Lists

They are dynamic in nature which allocates the memory when required.
Insertion and deletion operations can be easily implemented.
Stacks and queues can be easily executed.
Linked List reduces the access time.

Disadvantages of Linked Lists

The memory is wasted as pointers require extra memory for storage.
No element can be accessed randomly; it has to access each node sequentially.
Reverse Traversing is difficult in the linked list.

Applications of Linked Lists

Linked lists are used to implement stacks, queues, graphs, etc.
Unlike an array, In Linked Lists, we don't need to know the size in advance.

Types of Linked Lists

There are three types of linked lists.

- Singly Linked List
- Doubly Linked List
- Circular Linked List

I'm not going to discuss its types. Let's get into the Linked List in the Linux kernel.

Linked List in Linux Kernel

X

3rd party library. It has built-in Linked List which is Doubly Linked List. It is defined in defined in **/lib/modules/$(uname -r)/build/include/linux/list.h**.

Normally we used to declared linked list as like below snippet.

```
struct my_list{
     int data,
     struct my_list *prev;
     struct my_list *next;
};
```

But if want to Implement in Linux, then you could write like below snippet.

```
struct my_list{
     struct list_head list;     //linux kernel list implementation
     int data;
};
```

Where struct list_head is declared in **list.h**.

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

Initialize Linked List Head

Before creating any node in the linked list, we should create a linked list's head node first. So below macro is used to create a head node.

LIST_HEAD(linked_list);

This macro will create the head node structure in the name of "**linked_list**" and it will initialize that to its own address.

For example,

I'm going to create the head node in the name of "**etx_linked_list**"

Let’s see how internally it handles this. The macro is defined like below in **list.h**.

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

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

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

So it will create like below.

```
struct list_head etx_linked_list = { &etx_linked_list , &etx_linked_list};
```

While creating the head node, it initializes the prev and next pointer to its own address. Which means that prev and next pointer points to itself. The node is empty If the node’s prev and next pointer points to itself.

Create Node in Linked List

You have to create your linked list node dynamically or statically. Your linked list node should have member defined in `struct list_head`. Using below inline function, we can initialize that **`struct list_head`**.

`INIT_LIST_HEAD(struct list_head *list);`

For Example, My node is like this.

```
struct my_list{
     struct list_head list;     //linux kernel list implementation
     int data;
};

struct my_list new_node;
```

So we have to initialize the list_head variable using **`INIT_LIST_HEAD`** inline function.

```
INIT_LIST_HEAD(&new_node.list);
```

Add after Head Node

After created that node, we need to add that node to the linked list. So we can use this inline function to do that.

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

Insert a new entry **after the specified head**. This is good for implementing stacks.

Where,

struct list_head * new – the new entry to be added

struct list_head * head – list head to add it after

For Example,

```
list_add(&new_node.list, &etx_linked_list);
```

Add before Head Node

Insert a new entry before the specified head. This is useful for implementing queues.

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

Where,

struct list_head * new – new entry to be added

struct list_head * head – list head to add before the head

For Example,

```
list_add_tail(&new_node.list, &etx_linked_list);
```

X

list_del

It will delete the entry node from the list. This function removes the entry node from the linked list by disconnect prev and next pointers from the list, but it doesn't free any memory space allocated for entry node.

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

Where,

`struct list_head * entry`– the element to delete from the list.

list_del_init

It will delete the entry node from the list and reinitialize it. This function removes the entry node from the linked list by disconnect prev and next pointers from the list, but it doesn't free any memory space allocated for entry node.

```
inline void list_del_init(struct list_head *entry);
```

Where,

`struct list_head * entry`– the element to delete from the list.

Replace Node in Linked List

list_replace

This function is used to replace the old node with the new node.

```
inline void list_replace(struct list_head *old, struct
list_head *new);
```

Where,

`struct list_head * old`– the element to be replaced

X

list_replace_init

This function is used to replace the old node with the new node and reinitialize the old entry.

```
inline void list_replace_init(struct list_head *old, struct list_head *new);
```

Where,

`struct list_head * old`– the element to be replaced

`struct list_head * new`– the new element to insert

If ***old*** was empty, it will be overwritten.

Moving Node in Linked List

list_move

This will delete one list from the linked list and again adds to it after the head node.

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

Where,

`struct list_head * list` – the entry to move

`struct list_head * head`– the head that will precede our entry

list_move_tail

This will delete one list from the linked list and again adds to before the head node.

X

Where,

`struct list_head * list` – the entry to move

`struct list_head * head`– the head that will precede our entry

Rotate Node in Linked List

This will rotate the list to the left.

```
inline void list_rotate_left(struct list_head *head);
```

Where,

`head` – the head of the list

Test the Linked List Entry

list_is_last

This tests whether ***`list`*** is the last entry in the list ***head.***

```
inline int list_is_last(const struct list_head *list, const
                struct list_head *head);
```

Where,

`const struct list_head * list` – the entry to test

`const struct list_head * head` – the head of the list

It returns **1** if it is last entry otherwise **0**.

list_empty

X

```
inline int list_empty(const struct list_head *head);
```

Where,

`const struct list_head * head` – the head of the list

It returns **1** if it is empty otherwise **0**.

list_is_singular

This will tests whether a list has just one entry.

```
inline int list_is_singular(const struct list_head *head);
```

Where,

`const struct list_head * head` – the head of the list

It returns **1** if it has only one entry otherwise **0**.

Split Linked List into two part

This cut a list into two.

This helper moves the initial part of ***head***, up to and including ***entry***, from ***head*** to ***list***. You should pass on ***entry*** an element you know is on ***head***. ***list*** should be an empty list or a list you do not care about losing its data.

```
inline void list_cut_position(struct list_head *list, struct
        list_head *head, struct list_head *entry);
```

Where,

`struct list_head * list` – a new list to add all removed entries

X

itself and if so we won't cut the list

Join Two Linked Lists

This will join two lists, this is designed for stacks.

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

Where,

`const struct list_head * list` – the new list to add.

`struct list_head * head` – the place to add it in the first list.

Traverse Linked List

list_entry

This macro is used to get the struct for this entry.

```
list_entry(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.

list_for_each

This macro used to iterate over a list.

```
list_for_each(pos, head);
```

X

head – the head for your list.

So using those above two macros, we can traverse the linked list. We will see the example in the next tutorial. We can also use these below methods also.

list_for_each_entry

This is used to iterate over list of the given type.

```
list_for_each_entry(pos, head, 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_safe

This will iterate over the list of given type-safe against the removal of list entry.

```
list_for_each_entry_safe ( pos, n, head, member);
```

Where,

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.

X

list_for_each_prev

This will be used to iterate over a list backward.

```
list_for_each_prev(pos, head);
```

pos – the &struct list_head to use as a loop cursor.

head – the head for your list.

list_for_each_entry_reverse

This macro used to iterate backward over the list of the given type.

```
list_for_each_entry_reverse(pos, head, 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.

So, We have gone through all the functions which are useful for Kernel Linked List. Please go through the next tutorial (Part 2) for the Linked List sample program.

0

Article Rating

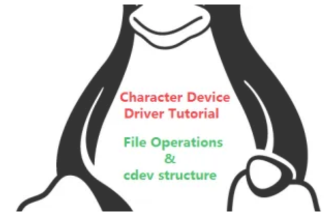
Share this:

X

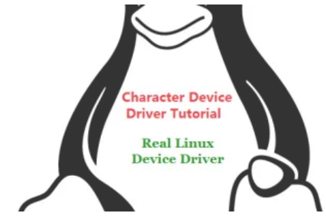
Loading...

Related

Linux Device Driver Tutorial Part 18 - Linked List in Linux Kernel Part 2
In "Device Drivers"



Linux Device Driver Tutorial Part 6 – Cdev structure and File Operations
In "Device Drivers"



Linux Device Driver Tutorial Part 7 - Linux Device Driver Tutorial Programming
In "Device Drivers"

X

Subscribe ▾ Connect with | Login

Be the First to Comment!

B I U S {} [+]

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

0 COMMENTS

X

X