Linux Device Driver Tutorial Part 18 – Linked List in Linux Kernel Part 2

Linux Device Driver Tutorial Part 18 – Example Linked List in Linux Kernel

If you don't know the functions used in linked list, please refer this previous tutorial for the detailed explanation about all linked list functions.
So now we can directly enter into the Linux Linked List Kernel programming. I took the source code form the previous tutorial. First i will explain how this code works.

1. When we write the value to our device file using echo value > /dev/etx_value, it will invoke the interrupt. Because we configured the interrupt by using software. If you don't know how it works, Please refer this tutorial.
2. Interrupt will invoke the ISR function.
3. In ISR we are allocating work to the Workqueue.
4. Whenever Workqueue executing, we are creating Linked List Node and adding the Node to the Linked List.
5. When we are reading the driver using cat /dev/etx_device, printing all the nodes which is present in the Linked List using traverse.
6. When we are removing the driver using rmmod, it will removes all the nodes in Linked List and free the memory.

Note : We are not using the sysfs functions. So I kept empty sysfs functions

Creating Head Node

```
/*Declare and init the head node of the linked list*/
LIST_HEAD(Head_Node);
```

This will create the head node in the name of Head_Node and initialize that.

Creating Node and add that into Linked List

```
/*Creating Node*/
    temp_node = kmalloc(sizeof(struct my_list), GFP_KERNEL);

    /*Assgin the data that is received*/
    temp_node->data = etx_value;

    /*Init the list within the struct*/
    INIT_LIST_HEAD(&temp_node->list);

    /*Add Node to Linked List*/
    list_add_tail(&temp_node->list, &Head_Node);
```

This will create the node, assign the data to its member. Then finally add that node to the Linked List using list_add_tail. (*This part will be present in the workqueue function*)

Traversing Linked List

```
    struct my_list *temp;
    int count = 0;
    printk(KERN_INFO "Read function\n");

    /*Traversing Linked List and Print its Members*/
    list_for_each_entry(temp, &Head_Node, list) {
        printk(KERN_INFO "Node %d data = %d\n", count++, temp->data);
    }

    printk(KERN_INFO "Total Nodes = %d\n", count);
```

Here, we are traversing each nodes using `list_for_each_entry`and print those values. (*This part will be present in the read function*)

Deleting Linked List

```
 /* Go through the list and free the memory. */
    struct my_list *cursor, *temp;
    list_for_each_entry_safe(cursor, temp, &Head_Node, list) {
        list_del(&cursor->list);
        kfree(cursor);
    }
```

This will traverse the each node using list_for_each_entry_safe and delete that using list_del. Finally we need to free the memory which is allocated using kmalloc.

Programming

Driver Source Code

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include<linux/slab.h>                 //kmalloc()
#include<linux/uaccess.h>              //copy_to/from_user()
#include<linux/sysfs.h>
#include<linux/kobject.h>
#include <linux/interrupt.h>
#include <asm/io.h>
#include <linux/workqueue.h>           // Required for workqueues


#define IRQ_NO 11

volatile int etx_value = 0;

dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;
struct kobject *kobj_ref;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);

static struct workqueue_struct *own_workqueue;


static void workqueue_fn(struct work_struct *work);

static DECLARE_WORK(work, workqueue_fn);

/*Linked List Node*/
struct my_list{
    struct list_head list;    //linux kernel list implementation
    int data;
};
```

```
/*Declare and init the head node of the linked list*/
LIST_HEAD(Head_Node);

/*************** Driver Fuctions **********************/
static int etx_open(struct inode *inode, struct file *file);
static int etx_release(struct inode *inode, struct file *file);
static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len,loff_t * off);
static ssize_t etx_write(struct file *filp,
                const char *buf, size_t len, loff_t * off);

/*************** Sysfs Fuctions **********************/
static ssize_t sysfs_show(struct kobject *kobj,
                struct kobj_attribute *attr, char *buf);
static ssize_t sysfs_store(struct kobject *kobj,
                struct kobj_attribute *attr,const char *buf, size_t count);

struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
/*****************************************************/


/*Workqueue Function*/
static void workqueue_fn(struct work_struct *work)
{
        struct my_list *temp_node = NULL;

        printk(KERN_INFO "Executing Workqueue Function\n");

        /*Creating Node*/
        temp_node = kmalloc(sizeof(struct my_list), GFP_KERNEL);

        /*Assgin the data that is received*/
        temp_node->data = etx_value;

        /*Init the list within the struct*/
        INIT_LIST_HEAD(&temp_node->list);

        /*Add Node to Linked List*/
        list_add_tail(&temp_node->list, &Head_Node);
}


//Interrupt handler for IRQ 11.
static irqreturn_t irq_handler(int irq,void *dev_id) {
        printk(KERN_INFO "Shared IRQ: Interrupt Occurred\n");
        /*Allocating work to queue*/
        queue_work(own_workqueue, &work);
```

```
        return IRQ_HANDLED;
}

static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = etx_read,
        .write          = etx_write,
        .open           = etx_open,
        .release        = etx_release,
};

static ssize_t sysfs_show(struct kobject *kobj,
                struct kobj_attribute *attr, char *buf)
{
        printk(KERN_INFO "Sysfs - Read!!!\n");
        return sprintf(buf, "%d", etx_value);
}

static ssize_t sysfs_store(struct kobject *kobj,
                struct kobj_attribute *attr,const char *buf, size_t count)
{
        printk(KERN_INFO "Sysfs - Write!!!\n");
        return count;
}

static int etx_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Opened...!!!\n");
        return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Closed...!!!\n");
        return 0;
}

static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len, loff_t *off)
{
        struct my_list *temp;
        int count = 0;
        printk(KERN_INFO "Read function\n");

        /*Traversing Linked List and Print its Members*/
        list_for_each_entry(temp, &Head_Node, list) {
```

```
        printk(KERN_INFO "Node %d data = %d\n", count++, temp->data);
    }

    printk(KERN_INFO "Total Nodes = %d\n", count);
    return 0;
}
static ssize_t etx_write(struct file *filp,
            const char __user *buf, size_t len, loff_t *off)
{
    printk(KERN_INFO "Write Function\n");
    /*Copying data from user space*/
    sscanf(buf,"%d",&etx_value);
    /* Triggering Interrupt */
    asm("int $0x3B");  // Corresponding to irq 11
    return len;
}


static int __init etx_driver_init(void)
{
    /*Allocating Major number*/
    if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
            printk(KERN_INFO "Cannot allocate major number\n");
            return -1;
    }
    printk(KERN_INFO "Major = %d Minor = %d n",MAJOR(dev), MINOR(dev));

    /*Creating cdev structure*/
    cdev_init(&etx_cdev,&fops);

    /*Adding character device to the system*/
    if((cdev_add(&etx_cdev,dev,1)) < 0){
        printk(KERN_INFO "Cannot add the device to the system\n");
        goto r_class;
    }

    /*Creating struct class*/
    if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
        printk(KERN_INFO "Cannot create the struct class\n");
        goto r_class;
    }

    /*Creating device*/
    if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
        printk(KERN_INFO "Cannot create the Device \n");
        goto r_device;
    }
```

```
        /*Creating a directory in /sys/kernel/ */
        kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);

        /*Creating sysfs file*/
        if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
                printk(KERN_INFO"Cannot create sysfs file......\n");
                goto r_sysfs;
        }
        if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_handler))) {
           printk(KERN_INFO "my_device: cannot register IRQ \n");
                   goto irq;
        }

        /*Creating workqueue */
        own_workqueue = create_workqueue("own_wq");

        printk(KERN_INFO "Device Driver Insert...Done!!!\n");
    return 0;

irq:
        free_irq(IRQ_NO,(void *)(irq_handler));

r_sysfs:
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);

r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        cdev_del(&etx_cdev);
        return -1;
}

void __exit etx_driver_exit(void)
{

        /* Go through the list and free the memory. */
        struct my_list *cursor, *temp;
        list_for_each_entry_safe(cursor, temp, &Head_Node, list) {
           list_del(&cursor->list);
           kfree(cursor);
        }

        /* Delete workqueue */
        destroy_workqueue(own_workqueue);
        free_irq(IRQ_NO,(void *)(irq_handler));
        kobject_put(kobj_ref);
```

```
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
        printk(KERN_INFO "Device Driver Remove...Done!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
MODULE_DESCRIPTION("A simple device driver - Kernel Linked List");
MODULE_VERSION("1.13");
```

MakeFile

```
obj-m += driver.o

KDIR = /lib/modules/$(shell uname -r)/build

all:
   make -C $(KDIR) M=$(shell pwd) modules

clean:
   make -C $(KDIR) M=$(shell pwd) clean
```

Building and Testing Driver

- Build the driver by using Makefile (*sudo make*)
- Load the driver using `sudo insmod driver.ko`
- `sudo su`
- To trigger interrupt read device file (`cat /dev/etx_device`)
- Now see the Dmesg (`dmesg`)

[5310.125001] Major = 246 Minor = 0 n
[5310.133127] Device Driver Insert...Done!!!
[5346.839872] Device File Opened...!!!
[5346.839950] Read function
[5346.839954] Total Nodes = 0
[5346.839982] Device File Closed...!!!

- By this time there is no nodes available.
- So now write the value to driver using `echo 10 > /dev/etx_device`

- By this time, One node has been added to the linked list.
- To test that read the device file using `cat /dev/etx device`
- Now see the Dmesg (`dmesg`)

[5346.839982] Device File Closed…!!!
[5472.408239] Device File Opened…!!!
[5472.408266] Write Function
[5472.408293] Shared IRQ: Interrupt Occurred
[5472.408309] Device File Closed…!!!
[5472.409037] Executing Workqueue Function
[5551.996018] Device File Opened…!!!
[5551.996040] Read function
[5551.996044] Node 0 data = 10
[5551.996046] Total Nodes = 1
[5551.996052] Device File Closed…!!!

- Our value has added to the list.
- You can also write many times to create and add the node to linked list
- Unload the module using `rmmod driver`