EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 15 – Workqueue in Linux Kernel Part 2**

Device Drivers

Linux Device Driver Tutorial Part 15 – Workqueue in Linux Kernel Part 2

This is the Series on Linux Device Driver. The aim of this series is to provide easy and practical examples that anyone can understand. In our previous tutorial we have seen the Workqueue in linux using the Static method through Device Driver Programming. Now we are going to see Linux Device Driver Tutorial Part 15 – Workqueue in Linux (Dynamic Creation Method).

1

Post Contents [hide]

Workqueue in Linux

Initialize work using Static Method

The below call creates a workqueue in Linux by the name `work` and the

function that gets scheduled in the queue is **work_fn**.

```
INIT_WORK(work,work_fn)
```

Where,

name: The name of the “work_struct” structure that has to be created.
func: The function to be scheduled in this workqueue.

Schedule work to the Workqueue

These below functions used to allocate the work to the queue.

Schedule_work

This function puts a job in the kernel-global workqueue if it was not already queued and leaves it in the same position on the kernel-global workqueue otherwise.

```
int schedule_work( struct work_struct *work );
```

where,

work – job to be done

Returns zero if *work* was already on the kernel-global workqueue and non-zero otherwise.

Scheduled_delayed_work

After waiting for a given time this function puts a job in the kernel-global workqueue.

```
int scheduled_delayed_work( struct delayed_work *dwork,
                            unsigned long delay );
```

where,

dwork – job to be done

delay – number of jiffies to wait or 0 for immediate execution

Schedule_work_on

This puts a job on a specific CPU.

```
int schedule_work_on( int cpu, struct work_struct *work );
```

where,

cpu– CPU to put the work task on

work– job to be done

Scheduled_delayed_work_on

After waiting for a given time this puts a job in the kernel-global workqueue on the specified CPU.

```
int scheduled_delayed_work_on(
int cpu, struct delayed_work *dwork, unsigned long
delay );
```

where,

cpu – cpu to put the work task on

dwork – job to be done

delay – number of jiffies to wait or 0 for immediate execution

Delete work from workqueue

There are also a number of helper functions that you can use to flush or cancel work on work queues. To flush a particular work item and block until the work is complete, you can make a call to **flush_work**. All work on a given work queue can be completed using a call to . In both cases, the caller blocks until the operation are complete. To flush the kernel-global work queue, call **flush_scheduled_work**.

```
int flush_work( struct work_struct *work );
void flush_scheduled_work( void );
```

Cancel Work from workqueue

You can cancel work if it is not already executing in a handler. A call to **cancel_work_sync** will terminate the work in the queue or block until the callback has finished (if the work is already in progress in the handler). If the work is delayed, you can use a call to **cancel_delayed_work_sync**.

```
int cancel_work_sync( struct work_struct *work );
int cancel_delayed_work_sync( struct delayed_work *dwork );
```

Check workqueue

Finally, you can find out whether a work item is pending (not yet executed by the handler) with a call to **work_pending** **or** **delayed_work_pending**.

```
work_pending( work );
delayed_work_pending( work );
```

Programming

Driver Source Code

In that source code, When we read the **/dev/etx_device** interrupt will hit (To understand interrupts in Linux go to this tutorial). Whenever interrupt hits, I'm scheduling the work to the workqueue. I'm not going to do any job in both interrupt handler and workqueue function since it is a tutorial post. But in real workqueue in Linux, this function can be used to carry out any operations that need to be scheduled.

```
1  #include <linux/kernel.h>
2  #include <linux/init.h>
3  #include <linux/module.h>
4  #include <linux/kdev_t.h>
5  #include <linux/fs.h>
6  #include <linux/cdev.h>
7  #include <linux/device.h>
8  #include<linux/slab.h>                 //kmalloc()
9  #include<linux/uaccess.h>              //copy_to/from_user()
10 #include<linux/sysfs.h>
11 #include<linux/kobject.h>
12 #include <linux/interrupt.h>
```

```
#include <asm/io.h>
#include <linux/workqueue.h>            // Required for workqueues


#define IRQ_NO 11

/* Work structure */
static struct work_struct workqueue;

void workqueue_fn(struct work_struct *work);

/*Workqueue Function*/
void workqueue_fn(struct work_struct *work)
{
        printk(KERN_INFO "Executing Workqueue Function\n");
}

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

        return IRQ_HANDLED;
}

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);

/*************** 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);

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)
{
```

```
76          printk(KERN_INFO "Sysfs - Read!!!\n");
77          return sprintf(buf, "%d", etx_value);
78  }
79
80  static ssize_t sysfs_store(struct kobject *kobj,
81                  struct kobj_attribute *attr,const char *buf, size_t count)
82  {
83          printk(KERN_INFO "Sysfs - Write!!!\n");
84          sscanf(buf,"%d",&etx_value);
85          return count;
86  }
87
88  static int etx_open(struct inode *inode, struct file *file)
89  {
90          printk(KERN_INFO "Device File Opened...!!!\n");
91          return 0;
92  }
93
94  static int etx_release(struct inode *inode, struct file *file)
95  {
96          printk(KERN_INFO "Device File Closed...!!!\n");
97          return 0;
98  }
99
100 static ssize_t etx_read(struct file *filp,
101                 char __user *buf, size_t len, loff_t *off)
102 {
103         printk(KERN_INFO "Read function\n");
104         asm("int $0x3B");  // Corresponding to irq 11
105         return 0;
106 }
107 static ssize_t etx_write(struct file *filp,
108                 const char __user *buf, size_t len, loff_t *off)
109 {
110         printk(KERN_INFO "Write Function\n");
111         return 0;
112 }
113
114
115 static int __init etx_driver_init(void)
116 {
117         /*Allocating Major number*/
118         if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
119                 printk(KERN_INFO "Cannot allocate major number\n");
120                 return -1;
121         }
122         printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
123
124         /*Creating cdev structure*/
125         cdev_init(&etx_cdev,&fops);
126
127         /*Adding character device to the system*/
128         if((cdev_add(&etx_cdev,dev,1)) < 0){
129             printk(KERN_INFO "Cannot add the device to the system\n");
130             goto r_class;
131         }
132
133         /*Creating struct class*/
134         if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
135             printk(KERN_INFO "Cannot create the struct class\n");
136             goto r_class;
137         }
138
```

```
139         /*Creating device*/
140         if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
141             printk(KERN_INFO "Cannot create the Device 1\n");
142             goto r_device;
143         }
144 
145         /*Creating a directory in /sys/kernel/ */
146         kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
147 
148         /*Creating sysfs file for etx_value*/
149         if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
150                 printk(KERN_INFO"Cannot create sysfs file......\n");
151                 goto r_sysfs;
152         }
153         if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(ir
154             printk(KERN_INFO "my_device: cannot register IRQ ");
155                     goto irq;
156         }
157 
158         /*Creating work by Dynamic Method */
159         INIT_WORK(&workqueue,workqueue_fn);
160 
161         printk(KERN_INFO "Device Driver Insert...Done!!!\n");
162     return 0;
163 
164 irq:
165         free_irq(IRQ_NO,(void *)(irq_handler));
166 
167 r_sysfs:
168         kobject_put(kobj_ref);
169         sysfs_remove_file(kernel_kobj, &etx_attr.attr);
170 
171 r_device:
172         class_destroy(dev_class);
173 r_class:
174         unregister_chrdev_region(dev,1);
175         cdev_del(&etx_cdev);
176         return -1;
177 }
178 
179 void __exit etx_driver_exit(void)
180 {
181         free_irq(IRQ_NO,(void *)(irq_handler));
182         kobject_put(kobj_ref);
183         sysfs_remove_file(kernel_kobj, &etx_attr.attr);
184         device_destroy(dev_class,dev);
185         class_destroy(dev_class);
186         cdev_del(&etx_cdev);
187         unregister_chrdev_region(dev, 1);
188         printk(KERN_INFO "Device Driver Remove...Done!!!\n");
189 }
190 
191 module_init(etx_driver_init);
192 module_exit(etx_driver_exit);
193 
194 MODULE_LICENSE("GPL");
195 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
196 MODULE_DESCRIPTION("A simple device driver - Workqueue part 2");
197 MODULE_VERSION("1.11");
```

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**
- To trigger the interrupt read device file (**sudo cat /dev/etx_device**)
- Now see the Dmesg (**dmesg**)

```
linux@embetronicx-VirtualBox: dmesg

[11213.943071] Major = 246 Minor = 0
[11213.945181] Device Driver Insert...Done!!!
[11217.255727] Device File Opened...!!!
[11217.255747] Read function
[11217.255783] Shared IRQ: Interrupt Occurred
[11217.255845] Executing Workqueue Function
[11217.255860] Device File Closed...!!!
```

- We can able to see the print "**Shared IRQ: Interrupt Occurred**" and "**Executing Workqueue Function**"
- Unload the module using **sudo rmmod driver**

In our next tutorial we will discuss Workqueue using its own worker thread.

Like this:

Loading...

Related

Linux Device Driver Tutorial Part 14 – Workqueue in Linux Kernel Part 1
In "Device Drivers"

Linux Device Driver Tutorial Part 16 – Workqueue in Linux Kernel Part 3
In "Device Drivers"

Linux Device Driver Tutorial Part 20 – Tasklet | Static Method
In "Device Drivers"

1

Subscribe Connect with | Login

Join the discussion

This site uses Akismet to reduce spam. Learn how your comment data is processed.

1 COMMENT Oldest

madhab

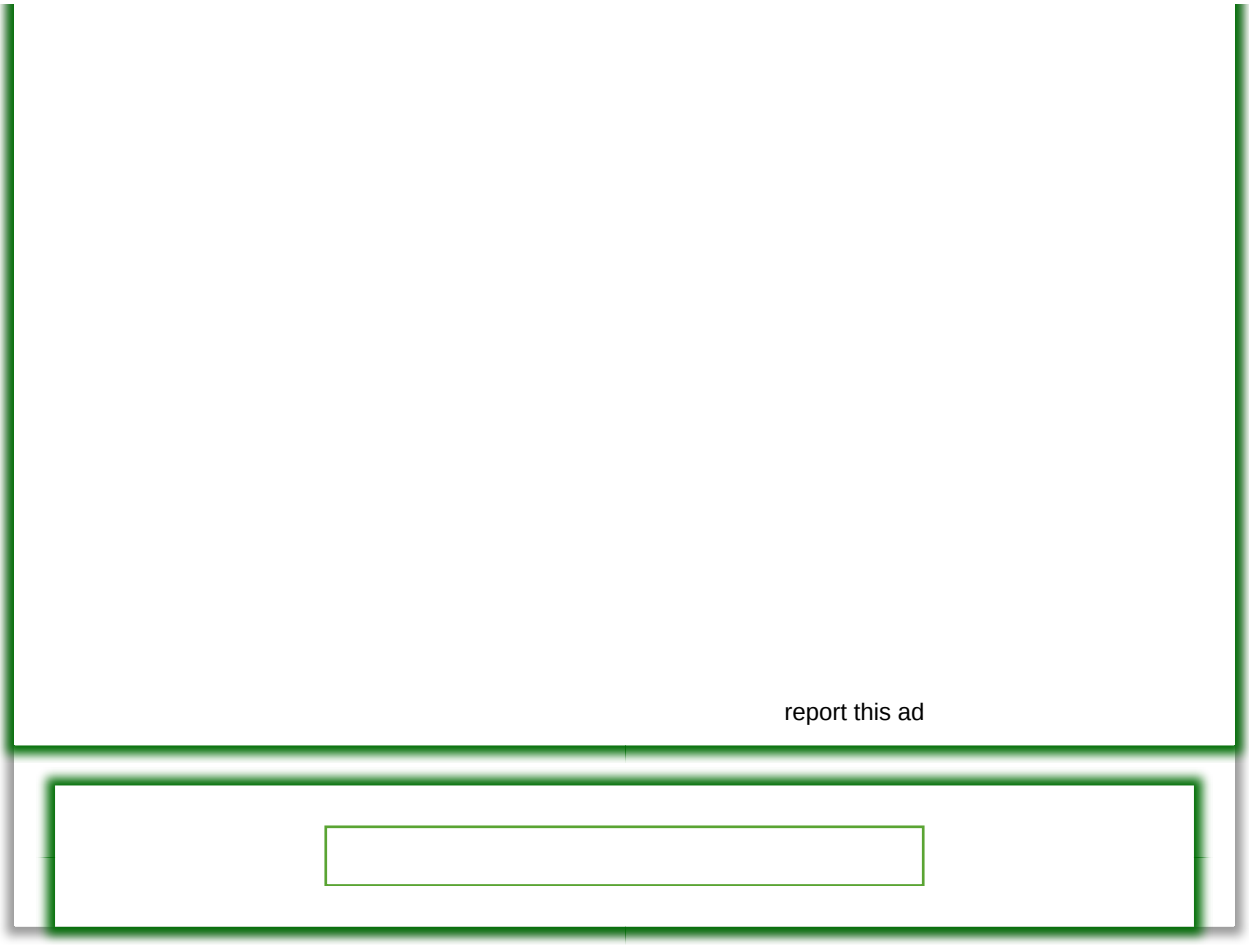
August 12, 2019 3:10 AM

when i am running cat /dev/etx_device and executing dmesg it shows
Device Driver Insert...Done!!!
[269946.176056] Device File Opened...!!!
[269946.176062] Read function
[269946.176064] Shared IRQ: Interrupt Occurred
[269946.176071] Device File Closed...!!!
[269946.176106] Executing Workqueue Function
but in you case it is
[11217.255747] Read function
[11217.255783] Shared IRQ: Interrupt Occurred
[11217.255845] Executing Workqueue Function
[11217.255860] Device File Closed...!!!

so can you please tell me both are same or any difference is there

Loading...

0 Reply



1