EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 14 – Workqueue in Linux Kernel Part 1**

Device Drivers

Linux Device Driver Tutorial Part 14 – Workqueue in Linux Kernel Part 1

X

carries on the discussion on character drivers and their implementation. This is Part 14 of the Linux device driver tutorial. In our previous tutorial, we have seen the Example of Interrupt through Device Driver Programming. Now we will see one of the Bottomhalf which is Workqueue in Linux Kernel.

Post Contents [hide]

Bottom Half

When Interrupt triggers, Interrupt Handler should be executed very quickly and it should not run for more time (it should not perform time-consuming tasks). If we have the interrupt handler which is doing more tasks then we need to divide into two halves.

1. Top Half
2. Bottom Half

Top Half is nothing but our interrupt handler. If our interrupt handler is doing less task, then the top half is more than enough. No need for the bottom half in that situation. But if we have more work to do when interrupt hits, then we need bottom half. The bottom half runs in the

There are 4 bottom half mechanisms are available in Linux:

1. Workqueue
2. Threaded IRQ
3. Softirq
4. Tasklet

In this tutorial, we will see Workqueue in Linux Kernel.

Workqueue in Linux Kernel

Work queues are added in the Linux kernel 2.6 version. Work queues are a different form of deferring work. Work queues defer work into a kernel thread; this bottom half always runs in process context. Because workqueue is allowing users to create a kernel thread and bind work to the kernel thread. So, this will run in process context and the work queue can sleep.

- Code deferred to a work queue has all the usual benefits of process context.
- Most importantly, work queues are schedulable and can therefore sleep.

Normally, it is easy to decide between using workqueue and softirq/tasklet:

There are two ways to implement Workqueue in Linux kernel.

1. Using global workqueue
2. Creating Own workqueue (We will see in the next tutorial)

Using Global Workqueue (Global Worker Thread)

In this tutorial, we will focus on this method.

In this method no need to create any workqueue or worker thread. So in this method, we only need to initialize work. We can initialize the work using two methods.

- Static method
- Dynamic method (We will see in the next tutorial)

Initialize work using Static Method

The below call creates a workqueue by the name and the function that we are passing in the second argument gets scheduled in the queue.

`DECLARE_WORK(name, void (*func)(void *))`

Where,

`name`*:* The name of the "work_struct" structure that has to be created.
`func`*:* The function to be scheduled in this workqueue.

Example

```
DECLARE_WORK(workqueue,workqueue_fn);
```

Schedule work to the Workqueue

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

X

```
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

X

```
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 the workqueue

X

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

Programming

Driver Source Code

I took the source code from the previous interrupt example tutorial. In that source code, When we read the **/dev/etx_device** the 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, this function can be used to carry out any operations that need to be scheduled.

```
#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


void workqueue_fn(struct work_struct *work);

/*Creating work by Static Method */
DECLARE_WORK(workqueue,workqueue_fn);

/*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) {
```

```

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)
{
        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");
        sscanf(buf,"%d",&etx_value);
        return count;
}

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

X

```

static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len, loff_t *off)
{
        printk(KERN_INFO "Read function\n");
        asm("int $0x3B");  // Corresponding to irq 11
        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");
        return 0;
}


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 1\n");
            goto r_device;
        }

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

        /*Creating sysfs file for etx_value*/
        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 *)(ir
            printk(KERN_INFO "my_device: cannot register IRQ ");
```

X

```

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)
{
        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 - Workqueue part 1");
MODULE_VERSION("1.10");
```

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**)

```
[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...!!!
```

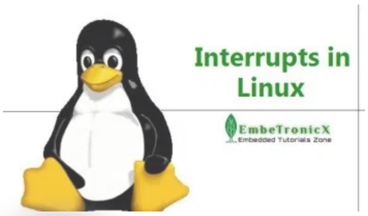
- We can able to see the print "**Shared IRQ: Interrupt Occurred**" and "**Executing Workqueue Function**"
- Unload the module using `sudo rmmod driver`

In our next tutorial, we will discuss Workqueue using the Dynamic method.

5

Article Rating

Related



Linux Device Driver Tutorial Part 12 -

Linux Device Driver Tutorial Part 20 – Tasklet |

Linux Device Driver Tutorial Part 13 –

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