Linux Device Driver Tutorial Part 13 – Interrupts Example Program in Linux Kernel

Interrupt Example Program in Linux Kernel

Before writing any interrupt program, you should keep these following points in your mind.

1. Interrupt handlers can not enter sleep, so to avoid calls to some functions which has sleep.
2. When the interrupt handler has part of the code to enter the critical section, use spinlocks lock, rather than mutexes. Because if it couldn't take mutex it will go to sleep until it takes the mute.
3. Interrupt handlers can not exchange data with the user space.
4. The interrupt handlers must be executed as soon as possible. To ensure this, it is best to split the implementation into two parts, top half and bottom half. The top half of the handler will get the job done as soon as possible, and then work late on the bottom half, which can be done with softirqs or tasklets or workqueus.
5. Interrupt handlers can not be called repeatedly. When a handler is already executing, its corresponding IRQ must be disabled until the handler is done.
6. Interrupt handlers can be interrupted by higher authority handlers. If you want to avoid being interrupted by a highly qualified handlers, you can mark the interrupt handler as a fast handler. However, if too many are marked as fast handlers, the performance of the system will be degraded, because the interrupt latency will be longer.

Functions Related to Interrupt

Before programming we should know the basic functions which is useful for interrupts. This table explains the usage of all functions.

FUNCTION	DESCRIPTION
`request_irq`	Register an IRQ, the parameters are as follows:

FUNCTION	DESCRIPTION
(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev_id)	**irq**: IRQ number to allocate. **handler**: This is Interrupt handler function.This function will be invoked whenever the operating system receives the interrupt.The data type of return is irq_handler_t, if its return value is IRQ_HANDLED, it indicates that the processing is completed successfully, but if the return value is IRQ_NONE, the processing fails. **flags**: can be either zero or a bit mask of one or more of the flags defined in `linux/interrupt.h`. The most important of these flags are: IRQF_DISABLED IRQF_SAMPLE_RANDOM IRQF_SHARED IRQF_TIMER (Explained after this table) **name**: Used to identify the device name using this IRQ, for example, cat / proc / interrupts will list the IRQ number and device name. **dev_id**: IRQ shared by many devices. When an interrupt handler is freed, dev provides a unique cookie to enable the removal of only the desired interrupt handler from the interrupt line. Without this parameter, it would be impossible for the kernel to know which handler to remove on a given interrupt line. You can pass NULL here if the line is not shared, but you must pass a unique cookie if your interrupt line is shared. This pointer is also passed into the interrupt handler on each invocation. A common practice is to pass the driver's device structure. This pointer is unique and might be useful to have within the handlers. **Return** returns zero on success and nonzero value indicates an error. *request_irq() cannot be called from interrupt context (other situations where code cannot block), because it can block.*
free_irq(unsigned int irq, void *dev_id)	Release an IRQ registered by `request_irq()` with the following parameters: **irq**: IRQ number. **dev_id**: is the last parameter of request_irq. If the specified interrupt line is not shared, this function removes the handler and disables the line. If the interrupt line is shared, the handler identified via dev_id is removed, but the interrupt line is disabled only when the last handler

FUNCTION	DESCRIPTION
	is removed. With shared interrupt lines, a unique cookie is required to differentiate between the multiple handlers that can exist on a single line and enable `free_irq()` to remove only the correct handler. In either case (shared or unshared), if dev_id is non-NULL, it must match the desired handler. A call to `free_irq()` must be made from process context.
`enable_irq(unsigned int irq)`	Re-enable interrupt disabled by disable_irq or disable_irq_nosync.
`disable_irq(unsigned int irq)`	Disable an IRQ from issuing an interrupt.
`disable_irq_nosync(unsigned int irq)`	Disable an IRQ from issuing an interrupt, but wait until there is an interrupt handler being executed.
`in_irq()`	returns true when in interrupt handler
`in_interrupt()`	returns true when in interrupt handler or bottom half

Interrupts Flags

These are the second parameter of the function. It has several flags. Here I explained important flags.

- IRQF_DISABLED.
  - When set, this flag instructs the kernel to disable all interrupts when executing this interrupt handler.
  - When unset, interrupt handlers run with all interrupts except their own enabled.

  Most interrupt handlers do not set this flag, as disabling all interrupts is bad form. Its use is reserved for performance-sensitive interrupts that execute quickly. This flag is the current manifestation of the SA_INTERRUPT flag, which in the past distinguished between “fast” and “slow” interrupts.
- IRQF_SAMPLE_RANDOM. This flag specifies that interrupts generated by this device should contribute to the kernel entropy pool. The kernel entropy pool provides truly random numbers derived from various random events. If this flag is specified, the timing of interrupts from this device are fed to the pool as entropy. Do not set this if your device issues interrupts at a predictable rate (e.g. the system timer) or can be influenced by external attackers (e.g. a networking device). On the other hand, most other hardware generates interrupts at non deterministic times and is therefore a good source of entropy.
- IRQF_TIMER. This flag specifies that this handler processes interrupts for the system timer.
- IRQF_SHARED. This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line.

Registering an Interrupt Handler

```
#define IRQ_NO 11

if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_handler))) {
        printk(KERN_INFO "my_device: cannot register IRQ ");
            goto irq;
    }
```

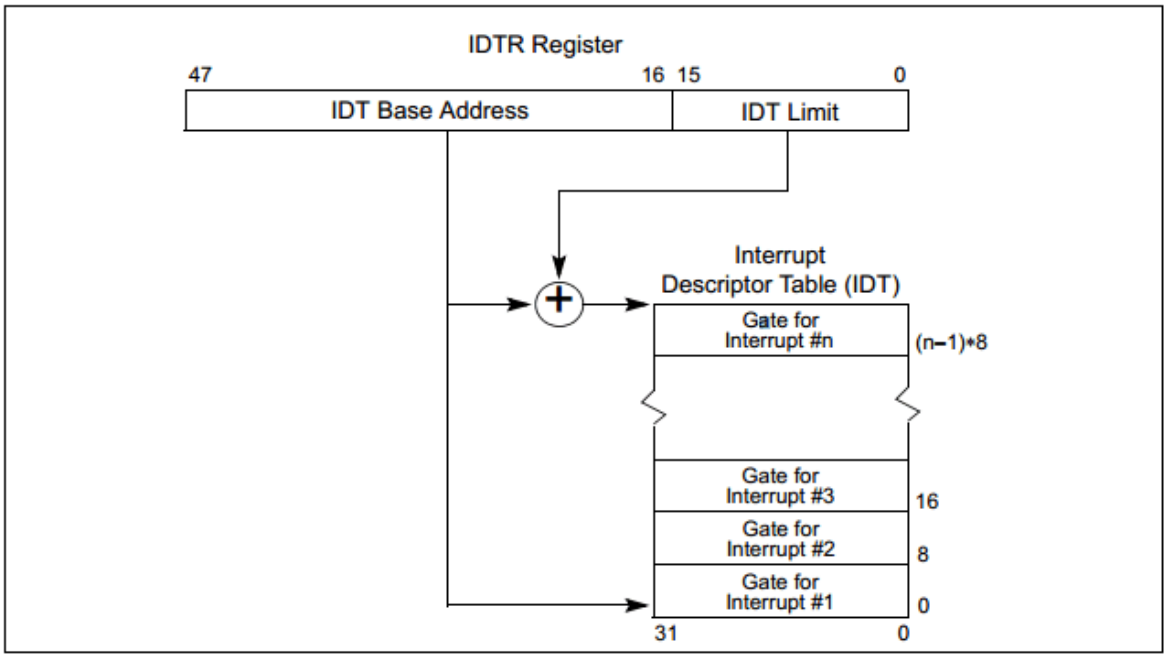
Freeing an Interrupt Handler

```
free_irq(IRQ_NO,(void *)(irq_handler));
```

Interrupt Handler

```
static irqreturn_t irq_handler(int irq,void *dev_id)
{
  printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
  return IRQ_HANDLED;
}
```

Programming
Interrupt can be coming from anywhere (any hardware) and anytime. In our tutorial we are not going to use any hardware. Here instead of using hardware, we are going to trigger interrupt by simulating. If you have only PC (without hardware), but you want to play around Interrupts in Linux you can follow our method.
Triggering Hardware Interrupt through Software
Intel processors handle interrupt using IDT (Interrupt Descriptor Table) . The IDT consists of 256 entries with each entry corresponding to a vector and of 8 bytes. All the entries are pointer to the interrupt handling function. The CPU uses IDTR to point to IDT. Relation between those two can be depicted as below,



An interrupt can be programmatically raised using 'int' instruction. For example , Linux system call was using int $0x80.
In linux IRQ to vector mapping is done in arch/x86/include/asm/irq_vectors.h.The used vector range is as follows ,

```
*  Vectors   0 ...  31 : system traps and exceptions - hardcoded events
*  Vectors  32 ... 127 : device interrupts
*  Vector  128         : legacy int80 syscall interface
*  Vectors 129 ... INVALIDATE_TLB_VECTOR_START-1 except 204 : device interrupts
*  Vectors INVALIDATE_TLB_VECTOR_START ... 255 : special interrupts
```

The IRQ0 is mapped to vector using the macro,
#define IRQ0_VECTOR (FIRST_EXTERNAL_VECTOR + 0x10)
where, FIRST_EXTERNAL_VECTOR = 0x20
So if we want to raise an interrupt IRQ11, programmatically we have to add 11 to vector of IRQ0.
0x20 + 0x10 + 11 = 0x3B (59 in Decimal).
Hence executing "asm("int $0x3B")" will raise interrupt IRQ 11.
The instruction will be executed while reading device file of our driver (/dev/etx_device).

Driver Source Code

Here I took the old source code from the sysfs tutorial. In that source I have just added interrupt code like request_irq, free_irq along with interrupt handler.
In this program interrupt will be triggered whenever we are reading device file of our driver (/dev/etx_device).
Whenever Interrupt triggers, it will prints the "**Shared IRQ: Interrupt Occurred**" Text

#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>

#define IRQ_NO 11

//Interrupt handler for IRQ 11.

static irqreturn_t irq_handler(int irq,void *dev_id) {
  printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
  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)
{
        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;
}

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)
{
        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 *)(irq_handler))) {
           printk(KERN_INFO "my_device: cannot register IRQ ");
                   goto irq;
        }
        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)
{
        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 - Interrupts");
MODULE_VERSION("1.9");
```

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

linux@embetronicx-VirtualBox: dmesg

[19743.366386] Major = 246 Minor = 0
[19743.370707] Device Driver Insert...Done!!!
[19745.580487] Device File Opened...!!!
[19745.580507] Read function
[19745.580531] Shared IRQ: Interrupt Occurred
[19745.580540] Device File Closed...!!!

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

This is the simple example using Interrupts in device driver. This is just a basic. You can also try using hardware. I hope this might helped you.