Kernel Korner

Dynamic Interrupt Request Allocation for Device Drivers

Dr B. Thangaraju

Abstract

Interrupts are how hardware gets software's attention. Here's how they work.

A computer cannot meet its requirements unless it communicates with its external devices. An interrupt is a communication gateway between the device and a processor. The allocation of an interrupt request line for a device and how the interrupt is handled play vital roles in device driver development. As the number of interrupt request lines in a system is limited, sharing an interrupt between devices is a must to access more devices. Any attempt to allocate an interrupt already in use, however, eventually crashes the system. This article explains the basics of the interrupt and the fundamentals of interrupt handling and includes an implementation of an interrupt request (IRQ) allocation for a character device.

The purpose of any device is to do some useful job, and to do so it should communicate with the microprocessor. When a processor wants to communicate with a device, it sends instructions to the device controller. A device controller controls the operation of a device. Similarly, if a device wants to reply to a processor that says new data is ready to be retrieved, the devices generate an interrupt to capture the processor's attention. An interrupt is a hardware mechanism that enables a device to communicate with a processor.

Until version 2.6, Linux had been non-preemptive, meaning that when a process is running in kernel mode, if any higher-priority process arrives in the ready-to-run queue, the lower-priority process cannot be preempted until it returns to user mode. But, an interrupt is allowed to divert CPU attention even though it is executing a process in kernel mode. This helps to improve the throughput of a system. When an interrupt occurs, the CPU suspends the current task and executes some other code, which responds to whatever event caused the interrupt.

Each device in a computer has a device controller, and it has a hardware pin that is used to assert when the device requires CPU service. This pin is attached to the corresponding interrupt pin in the CPU, which facilitates communication. The pin in the processor connected to the controller is called the interrupt request line. A CPU has several such pins so that many devices can be serviced by the processor. In a modern operating system, a programmable interrupt controller (PIC) is used to manage the IRQ lines between the processor and the various device controllers. The number of free IRQs in a system is restricted, but Linux has a mechanism to allow many pieces of hardware to share the same interrupts.

Interrupt servicing can be compared to a programmer's job. The programmer opens a mailbox and does his routine programming work. When new mail arrives, he is interrupted by a beep or by some other notification at the corner of the screen. Immediately, he saves the program and switches over to the mailbox. He then reads the mail, sends an acknowledgement and resumes his earlier work. A detailed reply listing the steps he has taken is sent later.

Similarly, when a CPU executes a process, a device can send an interrupt to the CPU regarding some task, for example, data is ready for transfer. When an interrupt comes, the CPU instantly saves the current value of
the program counter in the kernel mode stack and executes the corresponding interrupt service routine (ISR).

An ISR is a function situated in the kernel that determines the nature of the interrupt and performs whatever actions are needed, such as moving a block of data from hard disk to main memory. After executing the ISR, the CPU resumes the earlier process and executes.

A device driver is a software module in the kernel that waits for requests from the application program. Whenever an application wants to read data from a device, the corresponding device driver is invoked immediately, and the respective device is open for reading. If the system is waiting for slow hardware, it cannot do any useful job. One of the prime aims of kernel developers is to utilize system resources effectively. To avoid waiting for data from the hardware, the kernel gives this job to the device controller and resumes the stopped process. When reading completes, the device notifies the CPU through an interrupt. The processor then executes the corresponding ISR.

**Interrupt Classification**

Interrupts are divided into two broad categories, synchronous and asynchronous. Synchronous interrupts are generated by the CPU control unit when it is executing an instruction. The control unit issues an interrupt after terminating the instructions, hence the name synchronous interrupt. Asynchronous interrupts are created by hardware devices at random times with respect to the CPU clock. In the Intel context, the first one is called exceptions and the second is interrupts. Interrupt is identified by an unsigned one-byte integer called a vector. The vector ranges between 0 to 255. The first 32 (0–31) vectors are exceptions and non-maskable interrupts, which was explained in my article “Linux Signals for the Application Programmer”, LJ, March 2003. The range from 32–47 is assigned to maskable interrupts and is generated by IRQs (0–15 IRQ line numbers). The last range, from 48–255, is used to identify software interrupts; an example of this is interrupt 128 (int 0X80 assembly instructions), which is used to implement system calls.

**IRQ Allocation**

A snapshot of interrupts already in use on the system is stored in the /proc directory. The `cat /proc/interrupt` command displays the data related to the interrupts. The following output was displayed on my machine:

```
CPU0
 0: 82821789        XT-PIC timer
 1:  122            XT-PIC i8042
 2:  0              XT-PIC cascade
 8:  1              XT-PIC rtc
10: 154190         XT-PIC eth0
12:  100           XT-PIC i8042
14: 21578          XT-PIC ide0
15:  18            XT-PIC ide1
NMI:  0
ERR:  0
```

The first column is the IRQ line (vector ranges from 32–47), and the next column is the number of times the interrupts are delivered in the CPU after booting the system. The third column is related to the PIC, and the last column is the list of the device names that have registered handlers for the corresponding interrupt.

The simplest way to load a device driver dynamically is first to find the unused IRQ line in the system. A `request_irq` function is used to allocate a specified IRQ line number for a device. The syntax for the `request_irq` follows and is declared in `linux/sched.h`:

```c
int request_irq (unsigned int irq,
```
void (*handler) (int, void *,
    struct pt_regs *),
unsigned long flags,
const char *device, void *dev_id);

The details of the arguments in this function are:

- **unsigned int irq**: interrupt number, which we want to request from the system.

- **void (*handler) (int, void *, struct pt_regs *)**: whenever an interrupt is generated, we have to write ISRs to handle the interrupt; otherwise, the processor simply acknowledges it and does nothing else for that interrupt. This argument is the pointer to the handler function. The syntax for the handler function is:

  ```c
  void
  handler (int irq, void *dev_id,
           struct pt_regs *regs);
  ```

  The first argument is the IRQ number, which we already have mentioned in the request_irq function. The second argument is a device identifier, using major and minor numbers to identify which device is in charge of the current interrupt event. The third argument is used to save the process' context in the kernel stack before the processor starts executing the interrupt handler function. This structure is used when the system resumes the execution of the earlier process. Normally, device driver writers need not worry about this argument.

- **unsigned long flags**: the flags variable is used for interrupt management. The SA_INTERRUPT flag is set for fast interrupt handler, and it disables all the maskable interrupt. SA_SHIRQ is set when we want to share the irq with more than one device, SA_PROBE is set if we are interested in probing a hardware device using the IRQ line, and SA_RANDOM is used to seed the kernel random number generator. For more details of this flag, see /usr/src/linux/drivers/char/random.c.

- **constant char *device**: a device name that holds the IRQ.

- **void *dev_id**: the device identifier—it's a pointer to the device structure. When the interrupt is shared, this field points to the particular device.

The request_irq function returns 0 on success and -EBUSY when the allocation has failed. EBUSY is the error number of 16, which is described in the /usr/src/linux/include/asm/errno.h file. The free_irq function releases the IRQ number from the device. The syntax for this function is:

```c
free_irq (unsigned int irq, void *dev_id);
```

The explanation for the arguments is the same as above.

An ISR is invoked whenever an interrupt occurs. The operations to be performed on the cause of the interrupt are described in the ISR. The kernel maintains a table in memory, which contains the addresses of the interrupt routines (interrupt vectors). When an interrupt occurs, the processor checks the address of the ISR in the interrupt vector table and then executes. The task of the ISR is to react to the device according to the nature of the interrupt, such as read or write data. Typically, the ISR wakes up sleeping processes on the device if the interrupt signals the event for which they are waiting.

The amount of time the processor takes to respond to an interrupt is called interrupt latency. Interrupt latency is composed of hardware propagation time, register saving time and software propagation time. Interrupt latency should be minimal to improve the system's performance; for this reason, the ISR should be short and disable interrupts only for a brief time. Other interrupts can occur while interrupts are disabled, but the
processor does not allow them until interrupts are re-enabled. If more than one interrupt is blocked, the
processor allows them in priority order when it is ready for interrupt service.

Device driver developers should disable interrupts in driver code only when necessary, because the system
does not update the system timers, transfer network packets to and from buffers and so on during the interrupt
disabling. Driver developers should write ISRs to release the processor for other tasks. In real-world
scenarios, however, ISRs handle lengthy tasks. In such situations, the ISR can do only the time-critical
communication with the hardware to disable the interrupt and use the tasklet to perform most of the actual
data transfer processing. The tasklet is the advanced feature in the latest Linux kernel that does certain
operations related to the interrupt during safe times. The tasklet is the software interrupt, and it can be
interrupted by other interrupts. The internals of the interrupts have been explained in detail by Bovet and
Cesati (see the on-line Resources), and the implementation of the interrupts in device driver perspective is
presented by Rubini and Corbet (see Resources).

Simple Implementation

Any kernel module includes a device driver that can be loaded with the existing kernel, even when the
system is running. I explain the basic dynamic IRQ allocation procedure in a simple module shown in Listing
1. The following simple character device driver code describes the dynamic allocation of an IRQ line for a
device named OurDevice. When you insert the module, the init_module function is executed. If it is allocated
successfully, an unused major number and register for the given IRQ number for the device and the
 corresponding printk message then is printed. From here, we could check the IRQ allocation in the /proc
directory. The given IRQ is released at the time the module is removed. The best place to register an IRQ
number is an open entry point of a driver code, which subsequently frees the IRQ in a release function.

Listing 1. my_module.c

```c
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/interrupt.h>

static struct file_operations fops;
static int Major, irq = 7;

static void OurISR (int irq, void *device,
    struct pt_regs *regs)
{ /* important and immediate time critical tasks */
}

static int __init my_init_module(void)
{
    int status;
    Major = register_chrdev(0, "OurDevice", &fops);

    if (Major == -1) {
        printk (" Dynamic Major number 
            "allocation failed\n");
        return Major;
    }

    status = request_irq(irq,
        (void *)OurISR,
        SA_INTERRUPT,
        "OurDevice", &fops);
```
```c
if (status == -EBUSY) {
    printk("IRQ number allocation failed\n");
    unregister_chrdev(Major, "OurDevice");
    return status;
}

printk("The module is successfully loaded\n");
printk("Major number for OurDevice:   %d\n", Major);
printk("IRQ number for OurDevice:     %d\n", irq);
return 0;
}

static void __exit my_cleanup_module (void)
{
    printk("Major number %d  IRQ number %d \
" "are released\n", Major, irq);
    free_irq(irq, &fops);
    unregister_chrdev(Major, "OurDevice");
    printk("The Module is successfully unloaded\n");
}

module_init (my_init_module);
module_exit (myCleanup_module);

MODULE_LICENSE("GPL");
```

The my_module.c file is compiled with the 2.6.0-0.test2.1.29 kernel. The kernel-2.6.0-0.test2.1.30.i586.rpm
was downloaded along with all the dependent RPMs and installed. The RPM was downloaded from
http://people.redhat.com/arjanv/2.5/RPMS.kernel, and the device driver program was compiled as follows:

gcc -Wall -O3 -finline-functions \  
-Wstrict-prototypes -falign-functions=4 \  
-I/lib/modules/2.6.0-0.test2.1.29/build/include \  
-I/lib/modules/2.6.0-0.test2.1.29/build/include/ \  
-asm/mach-default \  
-I./include -D__KERNEL__ -DMODULE -DEXPORT_SYMTAB \  
-DKBUILD_MODNAME=my_module -c my_module.c -o \  
my_module.o

After inserting my_module.o, if the major number and the IRQ allocation for the device are successful, the
 corresponding printk statement output can be seen. If the IRQ number already is in use by another device, the
 kernel unregisters the device and releases the major number. The $cat /proc/interrupt command displays
the following output:

```
CPU0
0:  82887219  XT-PIC  timer
1:  122  XT-PIC  i8042
2:  0  XT-PIC  cascade
7:  0  XT-PIC  OurDevice
8:  1  XT-PIC  rtc
10:  154769  XT-PIC  eth0
12:  100  XT-PIC  i8042
14:  21636  XT-PIC  ide0
15:  18  XT-PIC  idle
NMI:  0
ERR:  0
```

An entry of OurDevice along with the IRQ line can be seen in the output. When we remove the module, the
kernel frees the IRQ number, unregisters the device and releases the major number.
Conclusion

Hopefully, this article makes clear the fundamental concepts of interrupts and the interrupt handling routine. The discussion of the `request_irq` and `free_irq` function is useful when we use these concepts in device drivers. The dynamic IRQ allocation procedure has been explained with the simple character device driver code.

Acknowledgements

C. Surest is acknowledged for his help during the preparation of this article.

Resources for this article: [http://www.linuxjournal.com/article/8064](http://www.linuxjournal.com/article/8064).