kernel hacker 修炼之道——李万鹏

男儿立志出乡关， 学不成名死不还。 埋骨何须桑梓地， 人生无处不青山。 ——西乡隆盛诗

kernel hacker修炼之道之驱动-按键

分类： linux驱动编程 2011-04-02 22:17 1541人阅读 评论(10) 收藏 举报

浅析linux驱动之按键

作者：李万鹏

按键程序使用了驱动的很多知识。有中断，阻塞，等待队列，linux设备驱动模型等。
使用中断处理的步骤是：

1. 向内核注册中断
2. 实现中断处理函数。

安装中断的函数是：

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

释放的函数：

这里主要是申请中断信号线，这是一个非常宝贵的资源。request_irq中的参数dev_id用于共享中断信号线的时候，因为需要中断处理的设备肯定比中断信号线多的，所以一旦某一个触发了中断，内核需要在这条中断线上寻找发生中断的设备，由于大家使用的中断号是一样的，所以通过dev_id来识别，dev_id是唯一的。使用完要注意释放着宝贵的资源。从下边的程序可以看到注册中断是在open的时候，为什么不在注册模块的函数中呢，因为如果是在注册函数中除非卸载模块，这样这个驱动会一直占用中断号，而自己可能不用，这样就白白浪费了宝贵的资源。
中断处理的函数原型是：

第一个参数是irq，第二个是dev_id，第三个是struct pt_reg *regs，很少用，它保存了处理器进入中断代码之前的处理器上下文快照。注意返回值，如果处理程序发现其设备的确需要处理，则应返回IRQ_HANDLED；否则，返回值应该是IRQ_NONE。

在读取的时候使用的是阻塞机制，也就是说如果用户程序获得不到数据，就阻塞。此时那个进程进入休眠状态，被CPU的调度器从运行队列搬到等待队列。Linux内核中阻塞是通过等待队列来实现的。在Linux中，一个等待队列通过一个“等待队列头(wait queue head)”来管理。等待队列头是一个类型为wait_queue_head_t的结构体，定义在<linux/wait.h>中。

可通过如下方法静态定义并初始化一个等待队列：

或者用动态的方法：

Linux内核中的休眠方式是使用wait_event宏,如下：

```
 wait_event(queue,condition);
wait_event_interruptible(queue,condition);
wait_event_timeout(queue,condition,timeout);
wait_event_interruptible_timeout(queue,condition,timeout);
```

queue是等待队列头，condition是等待条件。如果condition为0，则进行阻塞；否则，不阻塞。wait_event_interruptible宏与wait_event宏的区别是wait_event_interruptible是可以被信号中断的。当进程休眠时，它将期待某个条件在未来成为真；当一个进程被唤醒时，它必须再次检测它所等待的条件的确为真。
用来唤醒等待队列的函数：

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

wake_up会唤醒等待在给定queue上的所有进程。wake_up_interruptible只会唤醒那些执行可中断休眠的进程。

```
 #include <linux/fs.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <linux/platform_device.h>
#include <linux/types.h>
#include <asm/uaccess.h>
#include <asm/irq.h>
#include <linux/irq.h>
#include <mach/hardware.h>
#include <linux/slab.h>
#define key_major 234
#define key_minor 0

dev_t dev_num;
struct cdev * keyp;
struct class *key_class;
#define DEVICE_NAME "key_driver"
volatile int press_cnt[] = {0,0,0,0};
volatile bool ev_press = 0;
DECLARE_WAIT_QUEUE_HEAD(key_wait);

struct irq_key_descriptor{
unsigned int irq;
unsigned int flags;
const char *dev_name;
};

struct irq_key_descriptor key_irq[] = {
{IRQ_EINT0,IRQF_DISABLED,"key0"},
{IRQ_EINT2,IRQF_DISABLED,"key1"},
};

irqreturn_t key_interrupt(int irq, void *dev_id){
volatile int *press_cnt = (volatile int *)dev_id;
*press_cnt = *press_cnt + 1;
ev_press = 1;
wake_up_interruptible(&key_wait);
return IRQ_HANDLED;
}

int key_open(struct inode *inode, struct file *filp){
int i;
unsigned long err;
```

```
for(i = 0; i < sizeof(key_irq)/sizeof(key_irq[0]); i++){
err = request_irq(key_irq[i].irq, key_interrupt,key_irq[i].flags,key_irq[i].dev_name,
(void*)&press_cnt[i]);
if(err)
break;
}
if(err){
i--;
for(;i>=0;i--)
free_irq(key_irq[i].irq,(void*)&press_cnt[i]);
return -EBUSY;
}
return 0;
}

int key_close(struct inode *inode, struct file *filp){
int i;
for(i = 0; i < sizeof(key_irq)/sizeof(key_irq[0]); i++){
free_irq(key_irq[i].irq,(void*)&press_cnt[i]);
}
return 0;
}

ssize_t key_read(struct file *filp, char __user *buf, size_t count , loff_t * offp){
unsigned long err;
wait_event_interruptible(key_wait, ev_press);
ev_press = 0;
err = copy_to_user(buf,(const void *) press_cnt,count);
memset((void*)press_cnt, 0, sizeof(press_cnt));
return err?-EFAULT:0;
}

struct file_operations key_ops = {
.owner = THIS_MODULE,
.open = key_open,
.release = key_close,
.read = key_read,
};

void key_cdev_setup(void){
int err;
cdev_init(keyp,&key_ops);
keyp->owner = THIS_MODULE;
keyp->ops = &key_ops;
err = cdev_add(keyp,dev_num,1);
if(IS_ERR(&err))
printk(KERN_NOTICE "Error %d adding key_driver",err);
}

static int __init keyp_init(void){
int ret;
if(key_major){
dev_num = MKDEV(key_major, key_minor);
ret = register_chrdev_region(dev_num, 1, DEVICE_NAME);
}else{
ret = alloc_chrdev_region(&dev_num, key_minor, 1, DEVICE_NAME);
}
if(ret < 0){
printk(KERN_WARNING "key: can't get major %d\n",key_major);
return ret;
}
keyp = kmalloc(sizeof(struct cdev), GFP_KERNEL);
if(!keyp){
ret = -ENOMEM;
}
memset(keyp,0,sizeof(struct cdev));
```

```
key_cdev_setup();
key_class = class_create(THIS_MODULE,DEVICE_NAME);
if(IS_ERR(key_class)){
printk("Error:Failed to create key_class\n");
return -1;
}
device_create(key_class,NULL,dev_num,NULL,DEVICE_NAME);
printk(DEVICE_NAME "initialized\n");
return 0;
}

static void __exit keyp_exit(void){
cdev_del(keyp);
kfree(keyp);
unregister_chrdev_region(dev_num,1);
}

module_init(keyp_init);
module_exit(keyp_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("liwanpeng");
```

用户测试程序：

```
 #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>

int main(int argc,char **argv)
{
int i;
int ret;
int fd;
int press_cnt[4];

fd=open("/dev/key_driver",0);
if(fd<0) {
printf("Can't open /dev/key_driver \n");
return -1;
}
//这是个无限循环，进程有可能在read函数中休眠，当有按键按下时，
//它才返回
while(1) {
ret = read(fd,press_cnt,sizeof(press_cnt));
if(ret<0) {
printf("read err !\n");
continue;
}
//如果被按下的次数不为0，打印出来
for(i=0;i<sizeof(press_cnt)/sizeof(press_cnt[0]);i++) {
if(press_cnt[i])
printf("Key%d has been pressed %d times \n",i+1,press_cnt[i]);
}

}

}
```

Makefile:

```
 ifneq ($(KERNELRELEASE), )
obj-m:=key_driver.o
else
KERNELSRC :=/home/hacker/linux-2.6.30.4
modules:
make -C $(KERNELSRC) SUBDIRS=$(PWD) $@
```

```
clean:
rm -f *.o *.ko *.mod.c *~
endif
```

效果：

观察效果图，有的时候是按1次，有的是2，3，7等，看看用户程序，那里一直在读，如果按键不按下，则没有任何输出，因为读取进程被阻塞了，如果安下在读后按的次数会被清零。那应该都显示1啊，因为驱动程序中每按一次，在中断程序中唤醒等待进程，然后读取函数中进行了清零，应该每次都为1。可是，进程被唤醒后从等待队列搬到运行队列，在运行队列需要排队的，也就是说，可能不会立即获得时间片，这样下次中断又进行了加1，所以。。。哈哈