

在Linux下写一个简单的驱动程序

本文首先描述了一个可以实际测试运行的驱动实例，然后由此去讨论Linux下驱动模板的要素，以及Linux上应用程序到驱动的执行过程。相信这样由浅入深、由具体实例到抽象理论的描述更容易初学者入手Linux驱动的大门。

一、一个简单的驱动程序实例

驱动文件hello.c



```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/delay.h>

#define HELLO_MAJOR 231
#define DEVICE_NAME "HelloModule"

static int hello_open(struct inode *inode, struct file *file){
    printk(KERN_EMERG "hello open.\n");
    return 0;
}

static int hello_write(struct file *file, const char __user * buf, size_t count, loff_t
*ppos){
    printk(KERN_EMERG "hello write.\n");
    return 0;
}

static struct file_operations hello_flops = {
    .owner    = THIS_MODULE,
    .open     = hello_open,
    .write    = hello_write,
};

static int __init hello_init(void){
    int ret;

    ret = register_chrdev(HELLO_MAJOR, DEVICE_NAME, &hello_flops);
    if (ret < 0) {
        printk(KERN_EMERG DEVICE_NAME " can't register major number.\n");
        return ret;
    }
    printk(KERN_EMERG DEVICE_NAME " initialized.\n");
    return 0;
}
```

```
static void __exit hello_exit(void){
    unregister_chrdev(HELLO_MAJOR, DEVICE_NAME);
    printk(KERN_EMERG DEVICE_NAME " removed.\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```



驱动文件主要包括函数hello_open、hello_write、hello_init、hello_exit，测试案例中并没有赋予驱动模块具有实际意义的功能，只是通过打印日志的方式告知控制台一些调试信息，这样我们就可以把握驱动程序的执行过程。

在使用printk打印的时候，在参数中加上了“KERN_EMERG”可以确保待打印信息输出到控制台上。由于printk打印分8个等级，等级高的被打印到控制台上，而等级低的却输出到日志文件中。

编译驱动所需的Makefile



```
ifneq ($(KERNELRELEASE),)
MODULE_NAME = hellomodule
$(MODULE_NAME)-objs := hello.o
obj-m := $(MODULE_NAME).o
else
KERNEL_DIR = /lib/modules/$(uname -r)/build
MODULEDIR := $(shell pwd)

.PHONY: modules
default: modules

modules:
    make -C $(KERNEL_DIR) M=$(MODULEDIR) modules

clean distclean:
    rm -f *.o *.mod.c *.*.cmd *.ko
    rm -rf .tmp_versions
endif
```



编译驱动文件需要一个合适的makefile，因为编译驱动的时候需要知道内核头文件，编译规则等。

测试驱动的上层应用代码hellotest.c



```
#include <fcntl.h>
#include <stdio.h>
```

```
int main(void)
{
    int fd;
    int val = 1;
    fd = open("/dev/hellodev", O_RDWR);
    if(fd < 0){
        printf("can't open!\n");
    }
    write(fd, &val, 4);
    return 0;
}
```



上层测试案例中，首先打开设备文件，然后向设备中写入数据。如此，则会调用驱动中对应的xxx_open和xxx_write函数，通过驱动程序的打印信息可以判断是否真的如愿执行了对应的函数。

二、驱动实例测试

测试的方法整体来说就是，编译驱动和上层测试应用；加载驱动，通过上层应用调用驱动；最后，卸载驱动。

1、编译驱动

```
#make
```

make命令，直接调用Makefile编译hello.c，最后会生成“hellomodule.ko”。

2、编译上层应用

```
#gcc hellotest.c -o hellotest
```

通过这条命令，就能编译出一个上层应用hellotest。

3、加载驱动

```
#insmod hellomodule.ko
```

insmod加载驱动的时候，会调用函数hello_init()，打印的调试信息如下。

```
root@daneiqi:/mnt/hgfs/share/hello# insmod hellomodule.ko
[ 6253.445447] HelloModule initialized.
```

此外，在“/proc/devices”中可以看到已经加载的模块。

```
226 drm
231 HelloModule
252 usbmon
```

4、创建节点

虽然已经加载了驱动hellomodule.ko，而且在/proc/devices文件中也看到了已经加载的模块HelloModule，但是这个模块仍然不能被使用，因为在设备目录/dev目录下还没有它对应的设备文件。所以，需要创建一个设备节点。

```
#mknod /dev/hellodev c 231 0
```

在`/proc/devices`中看到HelloModule模块的主设备号为231，创建节点的时候就是将设备文件`/dev/hellodev`与主设备号建立连接。这样在应用程序操作文件`/dev/hellodev`的时候，就会定位到模块HelloModule。

```
root@daneiqi:/mnt/hgfs/share/hello# mknod /dev/hellodev c 231 0
root@daneiqi:/mnt/hgfs/share/hello# ll /dev/hellodev
crw-r--r-- 1 root root 231, 0 Oct 27 16:42 /dev/hellodev
```

`/proc/devices` 与 `/dev`的区别

- `/proc/devices`中的设备是驱动程序生成的，它可产生一个major供mknod作为参数。这个文件中的内容显示的是当前挂载在系统的模块。当加载驱动HelloModule的时候，并没有生成一个对应的设备文件来对这个设备进行抽象封装，以供上层应用访问。
- `/dev`下的设备是通过mknod加上去的，用户通过此设备名来访问驱动。我以为可以将`/dev`下的文件看做是硬件模块的一个抽象封装，Linux下所有的设备都以文件的形式进行封装。

5、上层应用调用驱动

```
#./hellotest
```

hellotest应用程序先打开文件`"/dev/hellodev"`，然后向此文件中写入一个变量val。期间会调用底层驱动中的`hello_open`和`hello_write`函数，hellotest的运行结果如下所示。

```
root@daneiqi:/mnt/hgfs/share/hello# ./hellotest
[ 6471.490408] hello open.
[ 6471.490544] hello write.
```

6、卸载驱动

```
#rmmod hellomodule
```

insmod卸载驱动的时候，会调用函数`hello_exit()`，打印的调试信息如下。

```
root@daneiqi:/mnt/hgfs/share/hello# rmmod hellomodule
[ 6483.840708] HelloModule removed.
```

总结一个模块的操作流程：

- (1) 通过insmod命令注册module
- (2) 通过mknod命令在`/dev`目录下建立一个设备文件"xxx"，并通过主设备号与module建立连接
- (3) 应用程序层通过设备文件`/dev/xxx`对底层module进行操作

三、驱动模板

从宏观上把握了驱动程序的框架，然后再从细节上完善驱动的功能，这是开发驱动程序的一般步骤。驱动模板必备要素有头文件、初始化函数、退出函数、版权信息，常用的扩展要素是增加一些功能函数完善底层驱动的功能。

1、头文件

```
#include <linux/module.h>
```

```
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/delay.h>
```

init.h 定义了驱动的初始化和退出相关的函数
kernel.h 定义了经常用到的函数原型及宏定义
module.h 定义了内核模块相关的函数、变量及宏

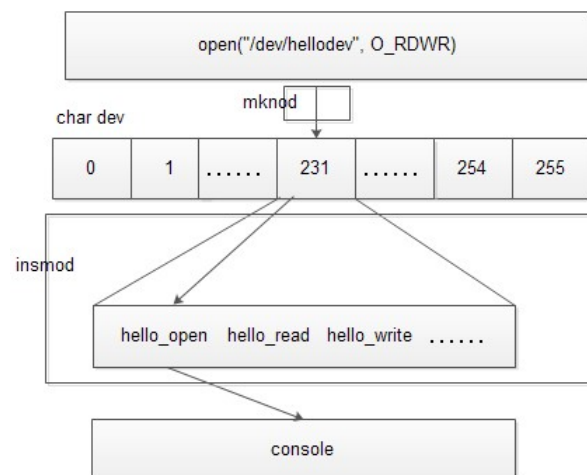
2、初始化函数

```
static int __init hello_init(void) {
    int ret;
    ret = register_chrdev(HELLO_MAJOR, DEVICE_NAME, &hello_flops);
    if (ret < 0) {
        printk(KERN_EMERG DEVICE_NAME " can't register major number.\n");
        return ret;
    }
    printk(KERN_EMERG DEVICE_NAME " initialized.\n");
    return 0;
}
module_init(hello_init);
```

当加载驱动到内核的时候，这个初始化函数就会被自动执行。

初始化函数顾名思义是用来初始化模块的，常用的功能是通过register_chrdev来注册函数。内核分配了一块内存（数组）专门用来存放字符设备的函数集，register_chrdev函数会在这个数组的HELLO_MAJOR位置将hello_flops中的内容进行填充，也就是将HelloModule的功能函数地址注册到设备管理内存集中。

形象的比喻好像是操作系统提供了很多的衣服架，注册设备就好像是把一个衣服挂到某一个衣服架上。衣服上有许多口袋，就好像每一个模块有许多功能程序接口。显然，如果想使用设备的某个功能，就可以先找到对应的衣服架，然后找到相应的口袋，去调用对应的函数，执行动作。



3、退出函数

```
static void __exit hello_exit(void) {
    unregister_chrdev(HELLO_MAJOR, DEVICE_NAME);
    printk(KERN_EMERG DEVICE_NAME " removed.\n");
}

module_exit(hello_exit);
```

当卸载驱动的时候，退出函数便会自动执行，完成一些列清楚工作。

在加载驱动的时候，我们向设备管理内存集中注册了该模块的相关功能函数。当卸载驱动的时候，就有必要将这个模块占用的内存空间清空。这样当其他的设备注册的时候便有更多的空间可以选择。


形象的比喻是，当卸载驱动的时候，就是把衣服从衣服架上取下来，这样衣服架就腾空了。

4、版权信息

```
MODULE_LICENSE("GPL");
```

Linux内核是按照GPL发布的，同样Linux的驱动程序也要提供版权信息，否则当加载到内核中系统会给出警告信息。


5、功能函数



```
static int hello_open(struct inode *inode, struct file *file) {
    printk(KERN_EMERG "hello open.\n");
    return 0;
}

static int hello_write(struct file *file, const char __user * buf, size_t count, loff_t
*ppos) {
    printk(KERN_EMERG "hello write.\n");
    return 0;
}

static struct file_operations hello_flops = {
    .owner    =    THIS_MODULE,
    .open     =    hello_open,
    .write    =    hello_write,
};
```



功能函数虽然不是一个驱动模板所必须的，但是一个有实际意义的驱动程序一定包含功能函数。功能函数实际上定义了这个驱动程序为用户提供了哪些功能，也就是用户可以对一个硬件设备可以进行哪些操作。

常见的功能函数有xxx_open()、xxx_write()、xxx_read()、xxx_ioctl()、xxx_llseek()等。当上层应用调用open()、write()、read()、ioctl()、llseek()等这些函数的时候，经过层层调用最后到达底层，调用相应的功能函数。结构体file_operations中的成员定义了很多函数，实际应用可以只对其部分成员赋值，其定义如下。



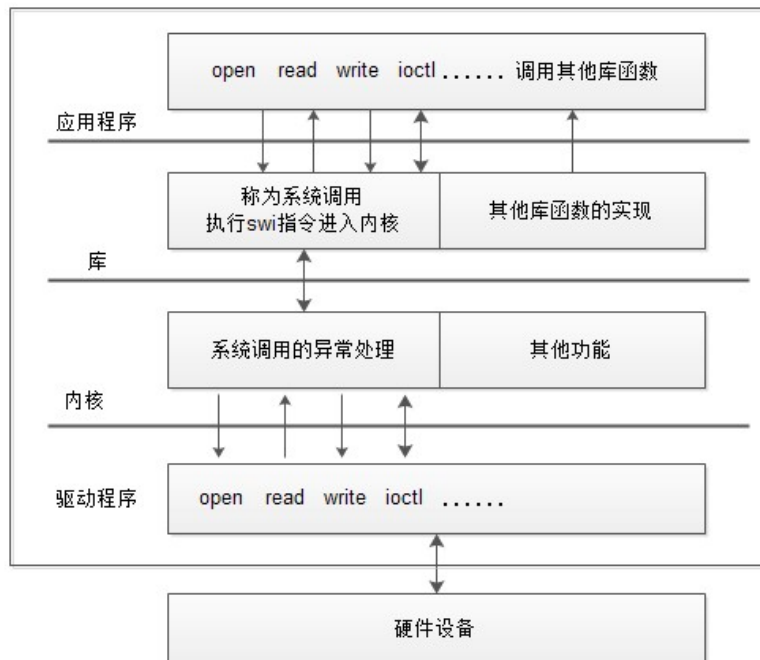


```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
    unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
    unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t,
    unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **);
};
```



四、从上层应用到底层驱动的执行过程

1、Linux系统的分层结构



Linux系统的分层结构为：应用层 ----> 库 ----> 内核 ----> 驱动程序 ----> 硬件设备。

2、从上层应用到底层驱动的执行过程

以“open("/dev/hellodev", O_RDWR)”函数的执行过程为例来说明。

- (1) 应用程序使用库提供的open函数打开代表hellodev的设备文件。
- (2) 库根据open函数传入的参数执行swi指令，这条指令会引起CPU异常，从而进入内核。
- (3) 内核的异常处理函数根据这些参数找到相应的驱动程序。
- (4) 执行相应的驱动程序。
- (5) 返回一个文件句柄给库，进而返回给应用程序。

3、驱动程序的执行特点

与应用程序不同，驱动程序从不主动运行，它是被动的：根据应用程序的要求进行初始化，根据应用程序的要求进行读写。驱动程序加载进内核，只是告诉内核“我在这里，我能做这些工作”，至于这些工作何时开始，则取决于应用程序。

驱动程序运行于“内核空间”，它是系统“信任”的一部分，驱动程序的错误有可能导致整个系统的崩溃。

参考资料：

[linux驱动开发框架](#)

《嵌入式Linux应用开发完全手册》

posted on 2015-10-27 15:57 [amanlikethis](#) 阅读(8868) 评论(0) [编辑](#) [收藏](#)