Linux字符设备驱动框架

字符设备是Linux三大设备之一(另外两种是块设备，网络设备)，字符设备就是字节流形式通讯的I/O设备,绝大部分设备都是字符设备，常见的字符设备包括鼠标、键盘、显示器、串口等等，当我们执行ls -l /dev的时候，就能看到大量的设备文件，c就是字符设备，b就是块设备，网络设备没有对应的设备文件。编写一个外部模块的字符设备驱动，除了要实现编写一个模块所需要的代码之外，还需要编写作为一个字符设备的代码。

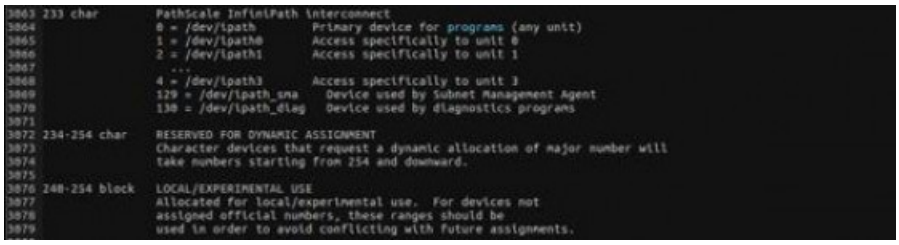
驱动模型

Linux一切皆文件，那么作为一个设备文件，它的操作方法接口封装在struct file_operations，当我们写一个驱动的时候，一定要实现相应的接口，这样才能使这个驱动可用，Linux的内核中大量使用"注册+回调"机制进行驱动程序的编写，所谓注册回调，简单的理解，就是当我们open一个设备文件的时候，其实是通过VFS找到相应的inode，并执行此前创建这个设备文件时注册在inode中的open函数，其他函数也是如此，所以，为了让我们写的驱动能够正常的被应用程序操作，首先要做的就是实现相应的方法，然后再创建相应的设备文件。

```
#include <linux/cdev.h> //for struct cdev
#include <linux/fs.h>   //for struct file
#include <asm-generic/uaccess.h>    //for copy_to_user
#include <linux/errno.h>            //for error number

static int ma = 0;
static int mi = 0;
const int count = 3;/* 准备操作方法集 *//*
struct file_operations {
    struct module *owner;   //THIS_MODULE

    //读设备
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    //写设备
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);

    //映射内核空间到用户空间
    int (*mmap) (struct file *, struct vm_area_struct *);

    //读写设备参数、读设备状态、控制设备
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);

    //打开设备
    int (*open) (struct inode *, struct file *);
    //关闭设备
    int (*release) (struct inode *, struct file *);

    //刷新设备
    int (*flush) (struct file *, fl_owner_t id);

    //文件定位
    loff_t (*llseek) (struct file *, loff_t, int);

    //异步通知
    int (*fasync) (int, struct file *, int);
    //POLL机制
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    。。。
};
*/

ssize_t myread(struct file *filep, char __user * user_buf, size_t size,
loff_t* offset)
{    return 0;
}
struct file fops = {
    .owner = THIS_MODULE,
    .read = myread，
    ...
};/* 字符设备对象类型
struct cdev {
    struct kobject kobj;
    struct module *owner;      //模块所有者（THIS_MODULE），用于模块计数
    const struct file_operations *ops;//操作方法集（分工:打开、关闭、读/写、...）
    struct list_head list;
    dev_t dev;                          //设备号（第一个）
    unsigned int count;           //设备数量
};
*/static int __init chrdev_init(void){
    ...    /* 构造cdev设备对象 */
    struct cdev *cdev_alloc(void);    /* 初始化cdev设备对象 */
    void cdev_init(struct cdev*, const struct file_opeartions*);
```

```
 /* 申请设备号，静态or动态*/
    /* 为字符设备静态申请第一个设备号 */
    int register_chrdev_region(dev_t from, unsigned count, const char* name);
  /* 为字符设备动态申请第一个设备号 */
    int alloc_chrdev_region(dev_t* dev, unsigned baseminor, unsigned count,
const char* name);

    ma = MAJOR(dev)     //从dev_t数据中得到主设备号
    mi = MINOR(dev)     //从dev_t数据中得到次设备号
    MKDEV(ma,1) //将主设备号和次设备号组合成设备号,多用于批量创建/删除设备文件

    /* 注册字符设备对象cdev到内核 */
    int cdev_add(struct cdev* , dev_t, unsigned);
    ...
}static void __exit chrdev_exit(void){
    ...    /* cdev_del()、cdev_put()二选一 */
    /* 从内核注销cdev设备对象 */
    void cdev_del(struct cdev* );    /* 从内核注销cdev设备对象 */
    void cdev_put(stuct cdev *);    /* 回收设备号 */
    void unregister_chrdev_region(dev_t from, unsigned count);
```

罗嗦一句，如果使用静态申请设备号，那么最大的问题就是不要与已知的设备号相冲突，内核在文档"Documentation/devices.txt"中已经注明了哪些主设备号被使用了，从中可以看出，在2^12个主设备号中，我们能够使用的范围是240-255以及261-2^12-1的部分，这也可以解释为什么我们动态申请的时候，设备号经常是250的原因。此外，通过这个文件，我们也可以看出，"主设备号表征一类设备"，但是字符/块设备本身就可以被分为好多类，所以内核给他们每一类都分配了主设备号。



实现read，write

Linux下各个进程都有自己独立的进程空间，即使是将内核的数据映射到用户进程，该数据的PID也会自动转变为该用户进程的PID，由于这种机制的存在，我们不能直接将数据从内核空间和用户空间进行拷贝，而需要专门的拷贝数据函数/宏：

```
long copy_from_user(void *to, const void __user * from, unsigned long n)
long copy_to_user(void __user *to, const void *from, unsigned long n)
```

这两个函数可以将内核空间的数据拷贝到回调该函数的用户进程的用户进程空间，有了这两个函数，内核中的read,write就可以实现内核空间和用户空间的数据拷贝。

```
ssize_t myread(struct file *filep, char __user * user_buf, size_t size, loff_t* offset)
{
    long ret = 0;
    size = size > MAX_KBUF?MAX_KBUF:size;
        if(copy_to_user(user_buf, kbuf,size)
                return -EAGAIN;
    }
    return 0;
}
```

实现ioctl

ioctl是Linux专门为用户层控制设备设计的系统调用接口，这个接口具有极大的灵活性，我们的设备打算让用户通过哪些命令实现哪些功能，都可以通过它来实现，ioctl在操作方法集中对应的函数指针是long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);，其中的命令和参数完全由驱动指定，通常命令会写在一个头文件中以供应用层和驱动层遵守同样的通信协议，Linux建议如图所示的方式定义ioctl()命令

```
设备类型    序列号     方向       数据尺寸
8bit        8bit    2bit    13/14bit
```

设备类型字段为一个幻数，可以是0~0xff之间的数，内核中的"ioctl-number.txt"给出了一个推荐的和已经被使用的幻数(但是已经好久没人维护了)，新设备驱动定义幻数的时候要避免与其冲突。

序列号字段表示当前命令是整个ioctl命令中的第几个，从1开始计数。

方向字段为2bit，表示数据的传输方向，可能的值是：_IOC_NONE，_IOC_READ，_IOC_WRITE和

_IOC_READ|_IOC_WRITE。

数据尺寸字段表示涉及的用户数据的大小，这个成员的宽度依赖于体系结构，通常是13或14位。

内核还定义了_IO()，_IOR()，_IOW()，_IOWR()这4个宏来辅助生成这种格式的命令。这几个宏的作用是根据传入的type(设备类型字段)，nr(序列号字段)和size(数据长度字段)和方向字段移位组合生成命令码。

内核中还预定义了一些I/O控制命令，如果某设备驱动中包含了与预定义命令一样的命令码，这些命令会被当做预定义命令被内核处理而不是被设备驱动处理，有如下4种:

FIOCLEX:即file ioctl close on exec 对文件设置专用的标志，通知内核当exec()系统带哦用发生时自动关闭打开的文件

FIONCLEX:即file ioctl not close on exec，清除由FIOCLEX设置的标志

FIOQSIZE:获得一个文件或目录的大小，当用于设备文件时，返回一个ENOTTY错误

FIONBIO:即file ioctl non-blocking I/O 这个调用修改flip->f_flags中的O_NONBLOCK标志

实例

```
//mycmd.h...#include <asm/ioctl.h>#define CMDT 'A'#define KARG_SIZE
36struct karg{    int kval;    char kbuf[KARG_SIZE];
};#define CMD_OFF _IO(CMDT,0)#define CMD_ON  _IO(CMDT,1)#define CMD_R
 _IOR(CMDT,2,struct karg)#define CMD_W   _IOW(CMDT,3,struct karg)...
```

```
//chrdev.c

static long demo_ioctl(struct file *filp, unsigned int cmd, unsigned long arg){
    static struct karg karg = {
        .kval = 0,
        .kbuf = {0},
    };
    struct karg *usr_arg;
    switch(cmd){
        case CMD_ON:        /* 开灯 */
        break;
        case CMD_OFF:        /* 关灯 */
        break;
        case CMD_R:
        if(_IOC_SIZE(cmd) != sizeof(karg)){
            return -EINVAL;
        }
        usr_arg = (struct karg *)arg;

        if(copy_to_user(usr_arg, &karg, sizeof(karg))){
            return -EAGAIN;
        }
        break;
        case CMD_W:
        if(_IOC_SIZE(cmd) != sizeof(karg)){
            return -EINVAL;
        }
        usr_arg = (struct karg *)arg;
        if(copy_from_user(&karg, usr_arg, sizeof(karg))){                    return -EAGAIN;
        }
        break;
        default:
        ;
    };
    return 0;}
```

创建设备文件

插入的设备模块，我们就可以使用cat /proc/devices命令查看当前系统注册的设备，但是我们还没有创建相应的设备文件，用户也就不能通过文件访问这个设备。设备文件的inode应该是包含了这个设备的设备号，操作方法集指针等信息，这样我们就可以通过设备文件找到相应的inode进而访问设备。创建设备文件的方法有两种，手动创建或自动创建，手动创建设备文件就是使用mknod /dev/xxx 设备类型 主设备号 次设备号的命令创建，所以首先需要使用cat /proc/devices查看设备的主设备号并通过源码找到设备的次设备号，需要注意的是，理论上设备文件可以放置在任何文件加夹，但是放到"/dev"才符合Linux的设备管理机制，这里面的devtmpfs是专门设计用来管理设备文件的文件系统。设备文件创建好之后就会和创建时指定的设备绑定，即使设备已经被卸载了，如要删除设备文件，只需要像删除普通文件一样rm即可。理论上模块名(lsmod),设备名(/proc/devices)，设备文件名(/dev)并没有什么关系，完全可以不一样，但是原则上还是建议将三者进行统一，便于管理。

除了使用蹩脚的手动创建设备节点的方式，我们还可以在设备源码中使用相应的措施使设备一旦被加载就自动创建设备文件，自动创建设备文件需要我们在编译内核的时候或制作根文件系统的时候就好相应的配置:

```
Device Drivers --->
    Generic Driver Options --->
        [*]Maintain a devtmpfs filesystem to mount at /dev
        [*] Automount devtmpfs at /dev,after the kernel mounted the rootfs
```

OR

制作根文件系统的启动脚本写入

```
mount -t sysfs none sysfs /sys
mdev -s //udev也行
```

有了这些准备，只需要导出相应的设备信息到"/sys"就可以按照我们的要求自动创建设备文件。内核给我们提供了相关的API

```
class_create(owner,name);
struct device *device_create_vargs(struct class *cls, struct device
*parent,dev_t devt, void *drvdata,const char *fmt, va_list vargs);

void class_destroy(struct class *cls);
void device_destroy(struct class *cls, dev_t devt);
```

有了这几个函数，我们就可以在设备的xxx_init()和xxx_exit()中分别填写以下的代码就可以实现自动的创建删除设备文件

```
/* 在/sys中导出设备类信息 */
    cls = class_create(THIS_MODULE,DEV_NAME);
        /* 在cls指向的类中创建一组(个)设备文件 */
    for(i= minor;i<(minor+cnt);i++){
        devp = device_create(cls,NULL,MKDEV(major,i),NULL,"%s%d",DEV_NAME,i);
    }
```

```
/* 在cls指向的类中删除一组(个)设备文件 */
    for(i= minor;i<(minor+cnt);i++){
        device_destroy(cls,MKDEV(major,i));
    }
        /* 在/sys中删除设备类信息 */
    class_destroy(cls);             //一定要先卸载device再卸载class
```

完成了这些工作，一个简单的字符设备驱动就搭建完成了，现在就可以写一个用户程序进行测试了^ - ^