回顾：

1.linux内核字符设备驱动相关内容

1.1.理念

1.2.字符设备文件

c

主设备号

次设备号

设备文件名

mknod

仅仅在open时使用

1.3.设备号

dev_t

12

20

MKDEV

MAJOR

MINOR

宝贵资源

alloc_chrdev_region

unregister_chrdev_region

主设备号：应用找到驱动

次设备号：驱动分区硬件个体

1.4.linux内核描述字符设备驱动数据结构

struct cdev

.dev

.count

.ops

配套函数：

cdev_init

cdev_add

cdev_del

1.5.字符设备驱动描述硬件操作接口的数据结构

```
struct file_operations {
  .open
  .release
  .read
  .write
};
```

open/release:可以根据用户的实际需求不用初始化

应用程序open/close永远返回成功

read:

1.分配内核缓冲区

2.获取硬件信息将信息拷贝到内核缓冲区

3.拷贝内核缓冲区数据到用户缓冲区

write:

1.分配内核缓冲区

2.拷贝用户缓冲区数据到内核缓冲区

3.拷贝内核缓冲区到硬件

注意：第二个形参buf:保存用户缓冲区的首地址,驱动不能直接访问,需要利用内存拷贝函数

copy_from_user/copy_to_user

2.linux内核字符设备驱动硬件操作接口之ioctl

2.1.掌握ioctl系统调用函数：

函数原型：
```
int ioctl(int fd, int request, ...);
```
函数功能：
1.不仅仅能够向设备发送控制命令(例如开关灯命令)
2.还能够跟硬件设备进行数据的读写操作

参数：
fd：设备文件描述符
request:向设备发送的控制命令,命令需要自己定义
        例如：
        #define LED_ON   (0x100001)
        #define LED_OFF (0x100002)
...:如果应用程序要传递第三个参数,第三个参数要传递用户缓冲区
    的首地址,将来底层驱动可以访问这个用户缓冲区的首地址
    同样底层驱动不能直接访问,需要利用内存拷贝函数
返回值：成功返回0，失败返回-1
参考代码：
```
//传递两个参数：
//开灯
ioctl(fd, LED_ON);
//关灯
ioctl(fd, LED_OFF);
```
说明：仅仅发送命令

```
//传递三个参数：
//开第一个灯：
int uindex = 1;
ioctl(fd, LED_ON, &uindex);

//关第一个灯：
  int uindex = 1;
ioctl(fd, LED_OFF, &uindex);
```
说明：不仅仅发送命令,还传递用户缓冲区的首地址,完成和设备的读或者写

2.2.ioctl对应的底层驱动的接口
回忆C编程：
```
int a = 0x12345678;
int *p = &a;
printf("a = %#x\n", *p);
```
等价于：
```
unsigned long p = &a;
printf("a = %#x\n", *(int *)p);

ioctl对应的底层驱动的接口
struct file_operations {
      long (*unlocked_ioctl)(struct file *file,
                             unsigned int cmd,
                             unsigned long arg)
};
```
调用关系：
应用ioctl->C库ioctl->软中断->内核sys_ioctl->驱动unlocked_ioctl接口
接口功能：
1.不仅仅向设备发送控制命令
2.还能够和设备进行数据的读或者写操作
参数：
file：文件指针

cmd:保存用户传递过来的参数,保存应用ioctl的第二个参数
arg：如果应用ioctl传递第三个参数,arg保存用户缓冲区的
首地址,内核不允许直接直接访问(int kindex=*(int *)arg)
需要利用内存拷贝函数,使用时注意数据类型的转换

案例：利用ioctl实现开关任意一个灯
实施步骤：
虚拟机执行：

```
mkdir /opt/drivers/day04/1.0 -p
cd /opt/drivers/day04/1.0
vim led_drv.c
vim Makefile
vim led_test.c
make
arm-linux-gcc -o led_test led_test.c
cp led_drv.ko led_test /opt/rootfs/home/drivers
```

ARM执行：

```
insmod /home/drivers/led_drv.ko
cat /proc/devices
mknod /dev/myled c 主设备号 0
/home/drivers/led_test on 1
/home/drivers/led_test on 2
/home/drivers/led_test off 1
/home/drivers/led_test off 2
```

3.linux内核字符设备驱动之设备文件的自动创建
3.1.设备文件手动创建
mknod /dev/设备文件名 c 主设备号 次设备号
3.2.设备文件自动创建实施步骤：
1.保证根文件系统rootfs具有mdev可执行程序
mdev可执行程序将来会帮你自动创建设备文件

```
which is mdev
/sbin/mdev
```

2.保证根文件系统rootfs的启动脚本etc/init.d/rcS必须有
以下两句话：

```
/bin/mount -a
echo /sbin/mdev > /proc/sys/kernel/hotplug
```

说明：
/bin/mount -a:将来系统会自动解析etc/fstab文件，进行
一系列的挂接动作
echo /sbin/mdev > /proc/sys/kernel/hotplug：将来驱动
创建设备文件时,会解析hotplug文件,驱动最终启动mdev来帮
驱动创建设备文件
3.保证根文件系统rootfs必须有etc/fstab文件,文件内容如下：

```
proc            /proc        proc   defaults 0 0
sysfs           /sys         sysfs  defaults 0 0
```

将/proc,/sys目录分别作为procfs,sysfs两种虚拟文件系统
的入口,这两个目录下的内容都是内核自己创建,创建的内存
存在于内存中
4.驱动程序只需调用以下四个函数即可完成设备文件的自动
创建和自动删除

```
struct class *cls; //定义一个设备类指针
//定义一个设备类,设备类名为tarena(类似长树枝)
cls = class_create(THIS_MODULE, "tarena");
//创建设备文件(类似长苹果)
```

```
    device_create(cls, NULL, 设备号, NULL, 设备文件名);
    例如：
    //自动在/dev/创建一个名为myled的设备文件
    device_create(cls, NULL, dev, NULL, "myled");

    //删除设备文件(摘苹果)
    device_destroy(cls, dev);

    //删除设备类(砍树枝)
    class_destroy(cls);

    案例：在ioctl实现的设备驱动中添加设备文件自动创建功能

4.linux内核字符设备驱动之通过次设备号区分硬件个体
  4.1.明确：多个硬件设备个体可以作为一个硬件看待,驱动也能够
            通过软件进行区分
  4.2.实现思路：
  1.驱动管理的硬件特性相似
    四个UART,Nand的多个分区
    不能把LED,按键放在一起
  2.主设备号为一个,驱动为一个,驱动共享
    cdev共享
    file_operations共享
    .open
    .read
    .write
    .release
    .unlocked_ioctl
    都共享
  3.多个硬件个体都有对应的设备文件
  4.驱动通过次设备号区分,次设备号的个数和硬件个体的数据一致

  4.3.了解两个数据结构：struct inode, struct file
  struct inode {
        dev_t   i_rdev; //保存设备文件的设备号信息
        struct cdev *i_cdev; //指向驱动定义初始化的字符设备对象led_cdev
        ...
  };
  作用：描述一个文件的物理上的信息(文件UID,GID,时间信息,大小等)
  生命周期：文件一旦被创建(mknod),内核就会创建一个文件对应的inode对象
            文件一旦被销毁(rm),内核就会删除文件对应的inode对象
  注意：struct file_operations中的open,release接口的第一个
  形参struct inode *inode,此指针就是指向内核创建的inode对象
  所以,驱动可以通过inode指针获取到设备号信息：inode->i_rdev

  struct file {
        const struct file_operations *f_op;//指向驱动定义初始化的硬件操作接口对
象led_fops
        ...
  };
  作用：描述一个文件被打开以后的信息
  生命周期：一旦文件被成功打开(open),内核就会创建一个file对象来描述
            一个文件被打开以后的状态属性
            一旦文件被关闭(close),内核就会销毁对应的file对象

  总结：一个文件只有一个inode对象,但是可以有多个file对象
```

切记：通过file对象指针获取inode对象指针的方法：
内核源码：fbmem.c
```
struct inode *inode = file->f_path.dentry->d_inode;
```
提取次设备号：
```
int minor = MINOR(inode->i_rdev);
```

案例：编写设备驱动,实现通过次设备号来区分两个LED
```
mknod /dev/myled1 c  250 0
mknod /dev/myled2 c  250 1
```

****

二.linux内核混杂设备驱动开发相关内容

1.概念

混杂设备本质还是字符设备,只是混杂设备的主设备号由内核已经定义为,为10,将来各个混杂设备个体通过次设备号来进行区分

2.混杂设备驱动的数据结构

```
struct miscdevice {
      int minor;
      const char *name;
      const struct file_operations *fops;
      ...
};
```

minor:混杂设备对应的次设备号,一般初始化时指定MISC_DYNAMIC_MINOR,表明让内核帮你分配一个次设备号

name:设备文件名,并且设备文件由内核帮你自动创建

fops：混杂设备具有的硬件操作接口

配套函数：
```
misc_register(&混杂设备对象);//注册混杂设备到内核
misc_deregister(&混杂设备对象);//卸载混杂设备
```

案例：利用混杂设备实现LED驱动,给用户提供的接口为ioctl
同上

vim分屏显示：
进入vim的命令行模式输入：
vs 文件名 //左右分屏
sp 文件名 //上下分屏
屏幕切换：ctrl+ww