专题10-字符设备驱动模型

一、使用字符驱动程序

在Linux系统中，驱动程序通常采用内核模块的 程序结构来进行编码。因此，编译/安装一个驱动程序，其实质就是编译/安装一个内核模块。

1.1、字符设备文件（设备节点）

通过字符设备文件，应用程 序可以使用相应的字符设备 驱动程序来控制字符设备。 创建字符设备文件的方法一 般有两种：

1.使用mknod命令

mknod /dev/文件名 c 主设备号 次设备号

2. 使用函数在驱动程序中创建

（后续课程介绍）

二、字符设备驱动模型

2.1、驱动模型

2.1.1、驱动初始化

2.1.1.1、分配设备描述结构

2.1.1.2、初始化设备描述结构

2.1.1.3、注册设备描述结构

2.1.1.4、硬件初始化

2.1.2、实现设备操作

2.1.3、驱动注销

2.2、字符设备驱动结构（cdev结构体）

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

2.2.1、设备号

2.2.1.1、通过命令ls -l /dev查看设备号

2.2.1.2、Linux内核中使用dev_t类型来定义设备号，dev_t这种类型其 实质为32位的unsigned int，其中高12位为主设备号，低20 位为次设备号.

问1：如果知道主设备号，次设备号，怎么组合成dev_t类型

答：dev_t dev = MKDEV(主设备号，次设备号)

问2: 如何从dev_t中分解出主设备号?

答: 主设备号 = MAJOR(dev_t dev)

问3: 如何从dev_t中分解出次设备号？

答: 次设备号=MINOR(dev_t dev)

2.2.1.3、如何为设备分配一个主设备号？

静态申请

开发者自己选择一个数字作为主设备号，然后通过函数register_chrdev_region向内核申请使用。

缺点：如果申请使用的设备号已经被内核中的其他驱动使用了，则申请失败。

动态分配

使用alloc_chrdev_region由内核分配一个可用的主设备号。

优点：因为内核知道哪些号已经被使用了，所以不会导致分 配到已经被使用的号。

2.2.1.4、设备号-注销

不论使用何种方法分配设备号，都应该在驱 动退出时，使用unregister_chrdev_region函数释放这些设备号。

2.2.2、操作函数集

Struct file_operations是一个函数指针的集合，定义能在 设备上进行的操作。结构中的函数指针指向驱动中的函数, 这些函数实现一个针对设备的操作, 对于不支持的操作则设 置函数指针为 NULL。例如：

```
struct file_operations dev_fops = {
    .llseek = NULL,
    .read = dev_read,
    .write = dev_write,
    .ioctl = dev_ioctl,
    .open = dev_open,
    .release = dev_release,
};
```

2.3、字符设备初始化

2.3.1、描述结构-分配

cdev变量的定义可以采用静态和动态两种办法

• 静态分配

```
struct cdev mdev;
```

• 动态分配

```
struct cdev *pdev = cdev_alloc();
```

2.3.2、描述结构-初始化

struct cdev的初始化使用cdev_init函数来完成。

```
cdev_init(struct cdev *cdev, const struct file_operations *fops)
```

参数：

cdev: 待初始化的cdev结构

fops: 设备对应的操作函数集

2.3.3、描述结构-注册

字符设备的注册使用cdev_add函数来完成。

```
cdev_add(struct cdev *p, dev_t dev, unsigned count)
```

参数：

p: 待添加到内核的字符设备结构

dev: 设备号

count: 该类设备的设备个数

2.4、硬件初始化

根据相应硬件的芯片手册，完成初始化。

2.5、实现设备操作

2.5.1、设备操作原型（也称为设备方法）。

```
int (*open) (struct inode *, struct file *)
```

打开设备，响应open系统

```
int (*release) (struct inode *, struct file *);
```

关闭设备，响应close系统调用

```
loff_t (*llseek) (struct file *, loff_t, int)
```

重定位读写指针，响应lseek系统调用

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)
```

从设备读取数据，响应read系统调用

```
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)
```

向设备写入数据，响应write系统调用

2.5.2、Struct file

在Linux系统中，每一个打开的文件，在内核中 都会关联一个struct file，它由内核在打开文件 时创建, 在文件关闭后释放。

重要成员：

```
loff_t f_pos /*文件读写指针*/
struct file_operations *f_op /*该文件所对应的操作*/
```

2.5.3、Struct inode

每一个存在于文件系统里面的文件都会关联一个 inode 结构，该结构主要用来记录文件物理上的 信息。因此，它和代表打开文件的file结构是不 同的。一个文件没有被打开时不会关联file结构， 但是却会关联一个inode 结构。

重要成员：

dev_t i_rdev：设备号

2.6、设备操作

2.6.1、设备操作-open

open设备方法是驱动程序用来为以后的操作 完成初始化准备工作的。在大部分驱动程序 中，open完成如下工作：

标明次设备号

启动设备

2.6.2、设备操作-release

release方法的作用正好与open相反。 这个设备方法有时也称为close，它应该：

关闭设备。

2.6.3、设备操作-read

read设备方法通常完成2件事情：

从设备中读取数据(属于硬件访问类操作)

将读取到的数据返回给应用程序

ssize_t (*read) (struct file *filp, char __user *buff, size_t count, loff_t *offp)

参数分析：

filp：与字符设备文件关联的file结构指针，由内核创建。

buff：从设备读取到的数据，需要保存到的位置。由read系统调用提供该参数。

count: 请求传输的数据量，由read系统调用提供该参数。

offp: 文件的读写位置，由内核从file结构中取出后，传递进来。

buff参数是来源于用户空间的指针，这类指针都不 能被内核代码直接引用，必须使用专门的函数

int copy_from_user(void *to, const void __user *from, int n)

int copy_to_user(void __user *to, const void *from, int n)

2.6.4、设备操作-write

write设备方法通常完成2件事情：

从应用程序提供的地址中取出数据

将数据写入设备(属于硬件访问类操作)

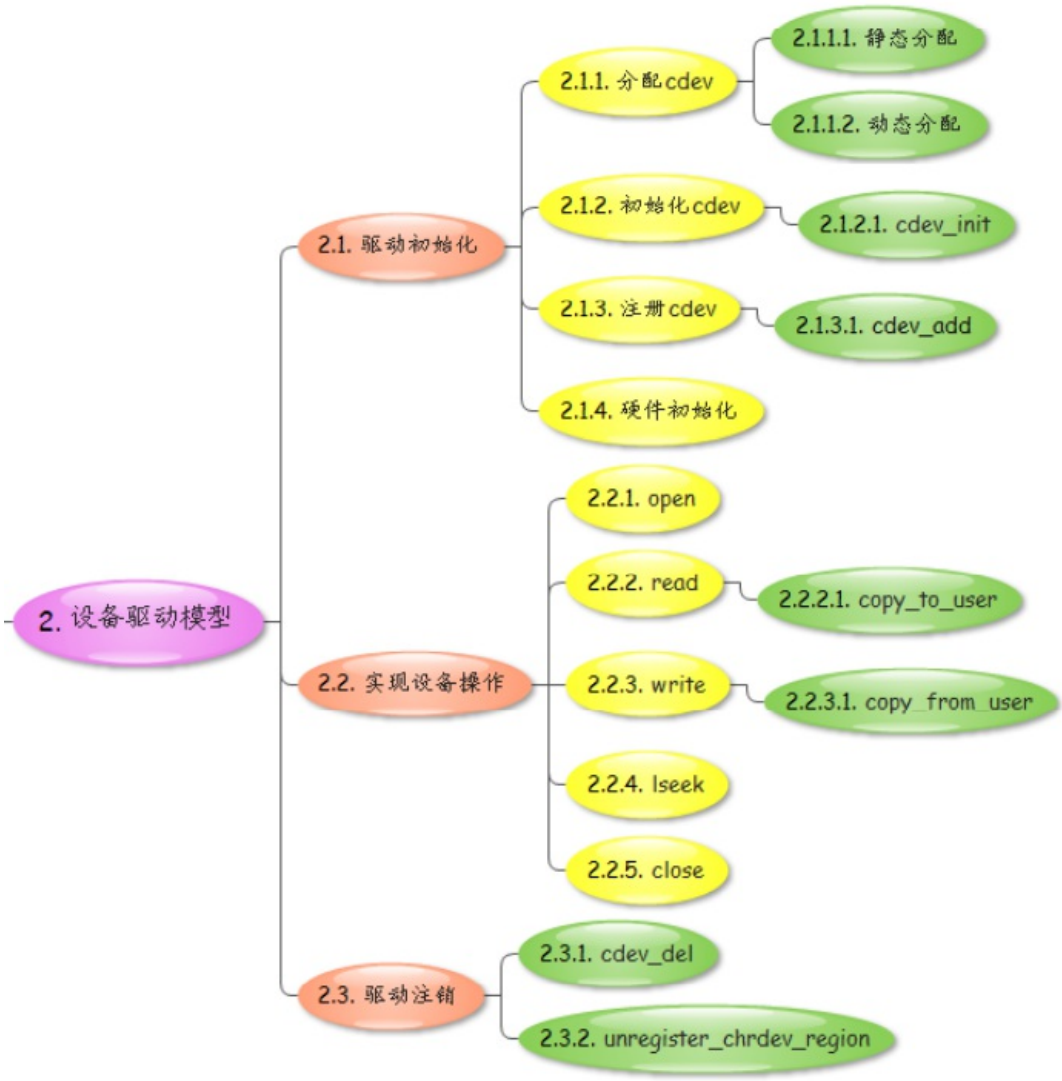
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)

其参数类似于read

2.7、驱动注销

当我们从内核中卸载驱动程序的时候，需要使 用cdev_del函数来完成字符设备的注销。

三、实现字符驱动

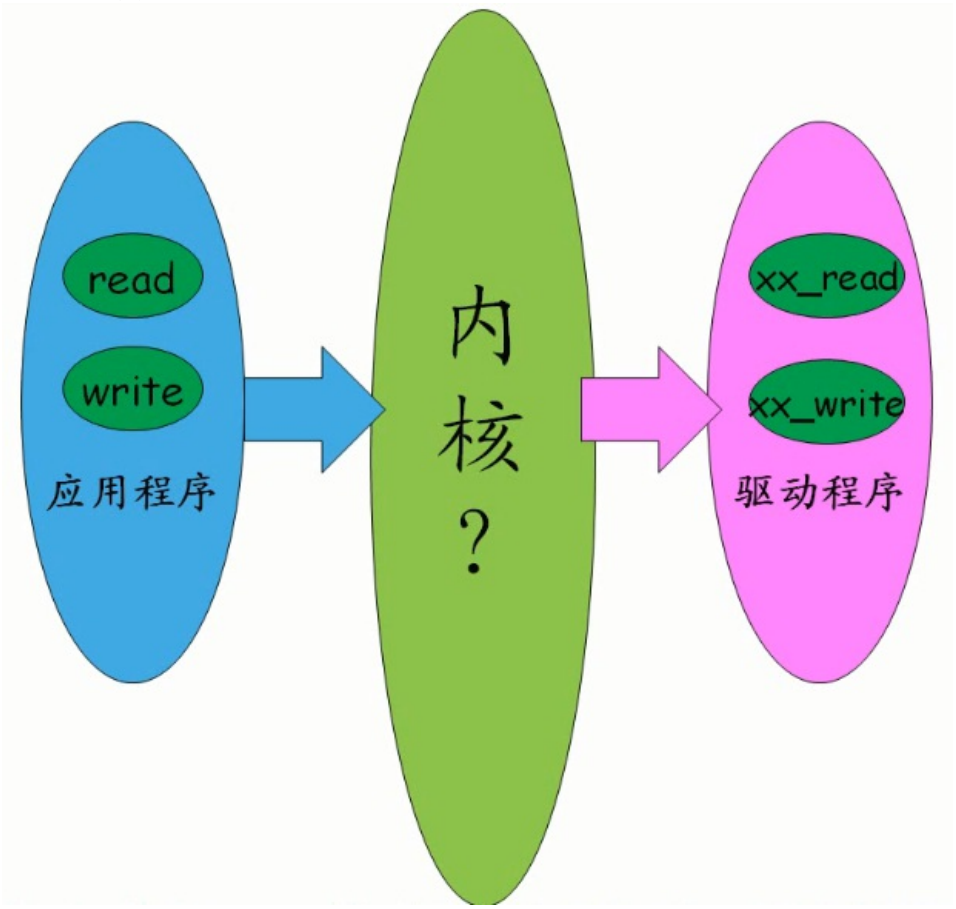



```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

struct cdev mdev;
dev_t devno;

int dev1_regs[5];
int dev2_regs[5];

loff_t mem_lseek (struct file *filp, loff_t offset, int whence)
{
 loff_t new_pos = 0;
 switch (whence) {
  case SEEK_SET:
   new_pos = 0 + offset;
   break;

  case SEEK_CUR:
   new_pos = filp->f_pos + offset;
   break;

  case SEEK_END:
   new_pos = 5 * sizeof(int) + offset;
   break;

  default:
   break;
 }
```

```
 filp->f_pos = new_pos;

 return new_pos;
}

ssize_t mem_read (struct file *filp, char __user *buf, size_t size, loff_t *ppos)
{
 int *reg_base = filp->private_data;

 copy_to_user(buf, reg_base + (*ppos), size);

 filp->f_pos += size;

 return size;
}

ssize_t mem_write (struct file *filp, const char __user *buf, size_t size, loff_t *ppos)
{
 int *reg_base = filp->private_data;

 copy_from_user(reg_base + (*ppos), buf, size);

 filp->f_pos += size;

 return size;
}

int mem_open (struct inode *node, struct file *filp)
{
 int num = MINOR(node->i_rdev);

 if (num == 0)
  filp->private_data = dev1_regs;

 if (num == 1)
  filp->private_data = dev2_regs;

 return 0;
}

int mem_close (struct inode *node, struct file *filp)
{
 return 0;
}

struct file_operations memfops =
{
 .llseek = mem_lseek,
 .read = mem_read,
 .write = mem_write,
 .open = mem_open,
 .release = mem_close,
};

int memdev_init(void)
{
 cdev_init(&mdev, &memfops);

 alloc_chrdev_region(&devno, 0, 2, "memdev");

 cdev_add(&mdev, devno, 2);

 return 0;
}

void memdev_exit(void)
{
 cdev_del(&mdev);
```

```
 unregister_chrdev_region(devno, 2);
}


module_init(memdev_init);
module_exit(memdev_exit);
```

四。字符驱动访问揭秘



分析：应用程序访问驱动程序（read）

SVC：系统调用指令。