### 实验11 编写字符设备驱动

## 1实验目的

通过编写字符设备驱动，理解linux下的设备驱动程序，学习相关设备驱动知识，加强对内核编程的理解。

## 2实验内容

编写字符设备驱动，实现open、write、read、icotl、release操作，并与前面的内核模块编程相结合，向系统中注册一个字符设备，并在用户态对驱动进行测试

## 3实验原理

3.1设备驱动相关知识

设备驱动程序在内核中的角色：他们是一个个独立的“黑盒子”，使某个特定的硬件响应一个定义良好的内部编程接口，这些接口完全隐藏了设备的工作细节。（驱动程序除了对外提供特定的接口外，任何实现细节对应用程序都是不可见的。）用户的操作通过一组标准化的调用执行，而这些调用独立于特定的驱动程序。驱动程序任务是把这些标准化调用映射到实际硬件的设备特有操作。

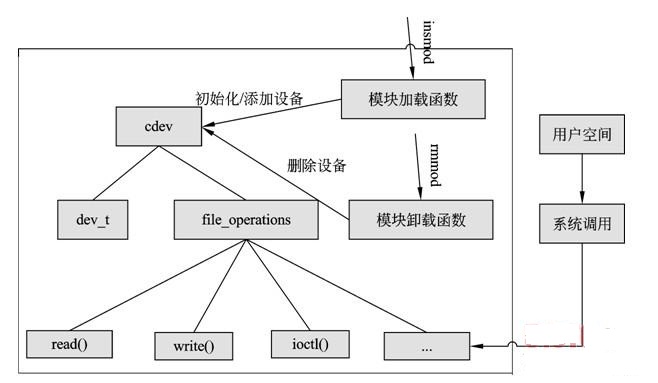
驱动程序就是应用程序与实际硬件之间的一个软件层，相同的硬件，不同的驱动程序可能提供不同的功能。实际的驱动程序设计要在许多要考虑的因素之间做出平衡。总的来说，驱动程序设计主要还是综合考虑下面三个方面的因素：提供给用户尽量多的选项、编写驱动程序要占用的时间以及尽量保持程序简单而不至于错误丛生。

其中linux的设备驱动程序的分类为：字符设备，块设备，网络设备。

3.2字符设备相关知识

字符设备是能够像字节流(类似文件)一样被访问的设备，有字符设备驱动程序来实现这种特性。字符设备驱动程序通常至少要实现open、close、read、write系统调用。字符设备可以通过文件系统节点来访问，这些设备文件和普通文件之间的唯一差别在于对普通文件的访问可以前后移动访问位置，而大多数字符设备是一个只能顺序访问的数据通道。一个字符设备是一种字节流设备，对设备的存取只能按顺序按字节的存取而不能随机访问，字符设备没有请求缓冲区，所有的访问请求都是按顺序执行的。但事实上现在一些高级字符设备也可以从指定位置一次读取一块数据。

3.3字符设备程序结构



**图 1字符设备驱动程序结构**

模块加载函数和模块卸载函数在之前的实验中已经学习过，在这里便不再说明。

3.3字符程序结构分析

1. 设备注册

内核使用struct cdev结构表示字符设备，所以在内核调用该设备操作之前，需要分配并注册一个或者多个该结构。字符设备的注册分为三个步骤：

1）分配cdev: struct cdev \*cdev\_alloc(void);

2）初始化cdev： void cdev\_init(struct cdev \*cdev, const struct

file\_operations \*fops)；

3)添加cdev： int cdev\_add(struct cdev \*p, dev\_t dev, unsigned count)

1. 设备操作的实现

file\_operations函数集的实现,这些操作将与设备编号连接起来。

\_\_user:用于文档，表明该指针是一个用户空间地址。

主要成员：open,  ioctl,  read,  write,  llseek

1. 设备注销

通过void cdev\_del(struct cdev \*p)函数将申请分配的cdev注销。

3.4字符程序重要数据结构

1. struct file：代表一个打开的文件描述符，系统中每一个打开的文件在内核中都有一个关联的struct file。它由内核在open时创建，并传递给在文件上操作的任何函数，直到最后关闭。当文件的所有实例都关闭之后，内核释放这个数据结构。
2. struct inode:用来记录文件的物理信息。它和代表打开的file结构是不同的。一个文件可以对应多个file结构，但只有一个inode结构。inode一般作为file\_operations结构中函数的参数传递过来。  
   　　inode译成中文就是索引节点。每个存储设备或存储设备的分区（存储设备是硬盘、软盘、U盘 ... ... ）被格式化为文件系统后，应该有两部份，一部份是inode，另一部份是Block，Block是用来存储数据用的。而inode呢，就是用来存储这些数据的信息，这些信息包括文件大小、属主、归属的用户组、读写权限等。inode为每个文件进行信息索引，所以就有了inode的数值。操作系统根据指令，能通过inode值最快的找到相对应的文件。
3. struct file\_operations。该结构体把操作与设备编号链接起来。

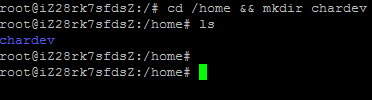
而在字符设备驱动编写过程中遇到的库函数便不再说明，请自己查阅资料。

## 4实验步骤

4.1编写字符驱动程序

步骤1：使用cd&&mkdir命令创建程序所在目录

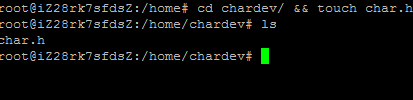
命令：# cd /home && mkdir chardev



**图表 1创建目录结果**

步骤2：进入该目录创建字符设备驱动程序头文件char.h

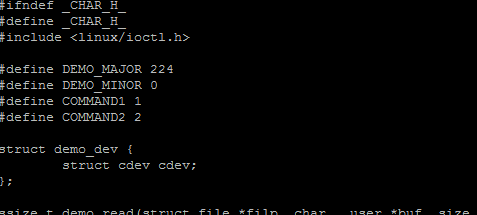
命令：# cd chardev && touch char.h



**图表 2执行cd,touch指令结果**

步骤3：使用vi指令编写该头文件

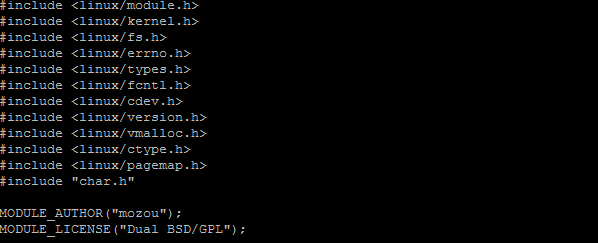
命令：#vi char.h



**图表 3编写char.h**

步骤4：使用vi在同目录下编写源文件

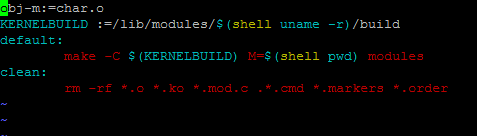
命令：# vi char.c



**图表 4编写char.c**

步骤5：编写makefile文件进行编译

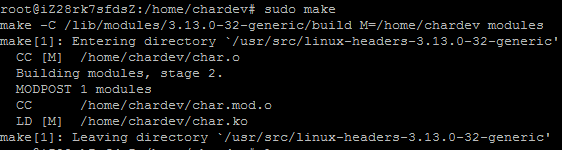
命令：#vi Makefile



**图表 5Makefile文件**

步骤6：编译字符驱动程序

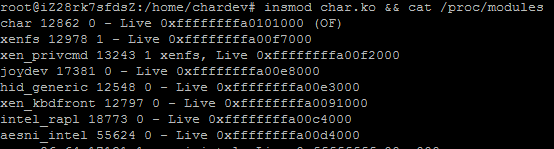
命令：#sudo make



**图表 6使用make编译结果**

步骤7：加载驱动程序char.ko，并查看是否安装正确

命令：# insmod char.ko && cat /proc/modules



**图表 7安装驱动并查看**

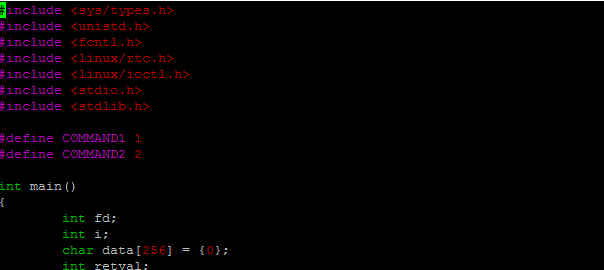
步骤8：创建设备节点

命令：# mknod /dev/mozou c 224 0

此处的节点设备号要与驱动程序中的注册的设备号相同

步骤9：创建测试代码

命令：# vi test.c



**图表 8测试程序**

步骤10：编译测试代码并运行结构

命令：#gcc test.c -o test

命令：# ./test



**图表 9运行测试代码**

4.2实验参考源代码

1.char.h

#ifndef \_CHAR\_H\_

#define \_CHAR\_H\_

#include <linux/ioctl.h>

#define DEMO\_MAJOR 224

#define DEMO\_MINOR 0

#define COMMAND1 1

#define COMMAND2 2

struct demo\_dev {

struct cdev cdev;

};

ssize\_t demo\_read(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*f\_pos);

ssize\_t demo\_write(struct file \*filp, const char \_\_user \*buf, size\_t count, loff\_t \*f\_pos);

loff\_t demo\_llseek(struct file \*filp, loff\_t off, int whence);

int demo\_ioctl(struct inode \*inode, struct file \*filp, unsigned int cmd, unsigned long arg);

#endif

1. char.c

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/errno.h>

#include <linux/types.h>

#include <linux/fcntl.h>

#include <linux/cdev.h>

#include <linux/version.h>

#include <linux/vmalloc.h>

#include <linux/slab.h>

#include <linux/ctype.h>

#include <linux/pagemap.h>

#include "char.h"

MODULE\_AUTHOR("mozou");

MODULE\_LICENSE("Dual BSD/GPL");

struct demo\_dev \*demo\_devices;

static unsigned char demo\_inc = 0;//全局变量，每次只能打开一个设备

static u8 demo\_buffer[256];

int demo\_open(struct inode \*inode, struct file \*filp)

{

struct demo\_dev \*dev;

if (demo\_inc > 0) return -ERESTARTSYS;

demo\_inc++;

dev = container\_of(inode->i\_cdev, struct demo\_dev, cdev);

filp->private\_data = dev;

return 0;

}

int demo\_release(struct inode \*inode, struct file \*filp)

{

demo\_inc--;

return 0;

}

ssize\_t demo\_read(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*f\_pos)

{

int result;

loff\_t pos = \*f\_pos; //pos: offset

if (pos >= 256) {

result = 0;

goto out;

}

if (count > (256 - pos))

count = 256 - pos;

pos += count;

if (copy\_to\_user(buf, demo\_buffer+\*f\_pos, count)) {

count = -EFAULT;

goto out;

}

\*f\_pos = pos;

out:

return count;

}

ssize\_t demo\_write(struct file \*filp, const char \_\_user \*buf, size\_t count, loff\_t \*f\_pos)

{

ssize\_t retval = -ENOMEM;

loff\_t pos = \*f\_pos;

if (pos > 256)

goto out;

if (count > (256 - pos))

count = 256 - pos;

pos += count;

if (copy\_from\_user(demo\_buffer+\*f\_pos, buf, count)) {

retval = -EFAULT;

goto out;

}

\*f\_pos = pos;

retval = count;

out:

return retval;

}

int demo\_ioctl(struct inode \*inode, struct file \*filp, unsigned int cmd, unsigned long arg)

{

if (cmd == COMMAND1) {

printk("ioctl command 1 successfully\n");

return 0;

}

if (cmd == COMMAND2) {

printk("ioctl command 2 successfully\n");

return 0;

}

printk("ioctl error\n");

return -EFAULT;

}

loff\_t demo\_llseek(struct file \*filp, loff\_t off, int whence)

{

loff\_t pos;

pos = filp->f\_pos;

switch (whence) {

case 0:

pos = off;

break;

case 1:

pos += off;

break;

case 2:

default:

return -EINVAL;

}

if ((pos > 256) || (pos < 0))

return -EINVAL;

return filp->f\_pos = pos;

}

struct file\_operations demo\_fops = {

.owner = THIS\_MODULE,

.llseek = demo\_llseek,

.read = demo\_read,

.write = demo\_write,

.ioctl = demo\_ioctl,

.open = demo\_open,

.release = demo\_release,

};

void demo\_cleanup\_module(void)

{

dev\_t devno = MKDEV(DEMO\_MAJOR, DEMO\_MINOR);

if (demo\_devices) {

cdev\_del(&demo\_devices->cdev);

kfree(demo\_devices);

}

unregister\_chrdev\_region(devno, 1);

}

int demo\_init\_module(void)

{

int result;

dev\_t dev = 0;

dev = MKDEV(DEMO\_MAJOR, DEMO\_MINOR);

result = register\_chrdev\_region(dev, 1, "DEMO");

if (result < 0) {

printk(KERN\_WARNING "DEMO: can't get major %d\n", DEMO\_MAJOR);

return result;

}

demo\_devices = kmalloc(sizeof(struct demo\_dev), GFP\_KERNEL);

if (!demo\_devices) {

result = -ENOMEM;

goto fail;

}

memset(demo\_devices, 0, sizeof(struct demo\_dev));

cdev\_init(&demo\_devices->cdev, &demo\_fops);

demo\_devices->cdev.owner = THIS\_MODULE;

demo\_devices->cdev.ops = &demo\_fops; //将创建的字符设备与 //file\_operations中各函数操作 //连接起来

result = cdev\_add(&demo\_devices->cdev, dev, 1);

if (result) {

printk(KERN\_NOTICE "error %d adding demo\n", result);

goto fail;

}

return 0;

fail:

demo\_cleanup\_module();

return result;

}

module\_init(demo\_init\_module);

module\_exit(demo\_cleanup\_module);

1. Makefile文件

obj-m:=char.o

KERNELBUILD :=/lib/modules/$(shell uname -r)/build

default:

make -C $(KERNELBUILD) M=$(shell pwd) modules

clean:

rm -rf \*.o \*.ko \*.mod.c .\*.cmd \*.markers \*.order

## 5实验总结

经过本次实验，对linux操作系统的内核模块有了新的理解与认识。在学习字符设备驱动编程中，对上次实验做了巩固。对用户态和内核态的关系和联系有了更深的了解。对linux下的驱动程序有了直观的认识。

## 6参考文献

Linux设备驱动程序第3版 第3章字符驱动

## 7实验拓展

扩充字符设备驱动程序功能，对itocl函数在内核上的应用做加深了解。