**实验二：设备驱动实验**

**虚拟字符设备**

1. **实验目的**

1.了解Linux操作系统内核的构成；

2.了解Linux设备的分类；

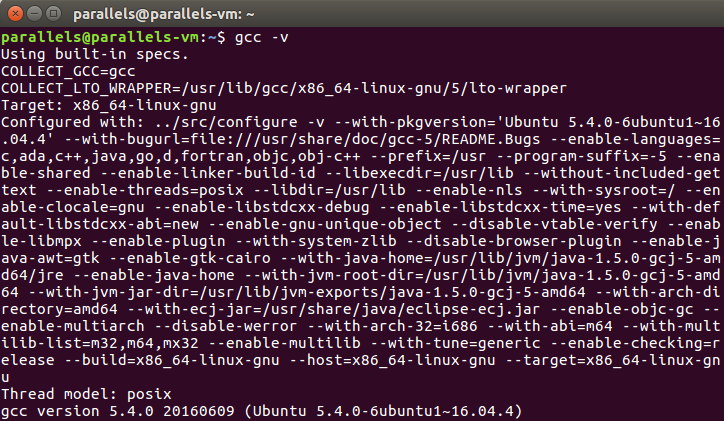
3.掌握编写驱动程序的基本概念和一般步骤；

4.熟悉模块的加载、卸载及驱动程序的调试；

5.掌握用C语言开发Linux内核程序的基本方法。

1. **实验准备**

1.检查是否安装gcc：



2.查看内核版本及路径：/usr/src/linux-headers-4.10.0-28-generic



1. **实验时间**

2019年12月02日

1. **实验内容**

设计一个虚拟字符驱动设备程序，具体要求为：

设备中有一个4字节的全局变量，对设备的读写即是对其中全局变量操作。

设计一个用户态程序：

能够对虚拟字符设备进行读写操作。

1. **基本原理**

1.设备描述结构：在任何一种驱动模型中，设备都会用内核中的一种结构来描述，字符设备在内核中使用struct cdev描述：

Struct cdev{

Struct kobject kobj;

Struct module \*owner;

Const struct file\_operations \*ops;//设备操作集

Struct list\_head list;

Dev\_t dev;//设备号

Unsigned int count;//设备数

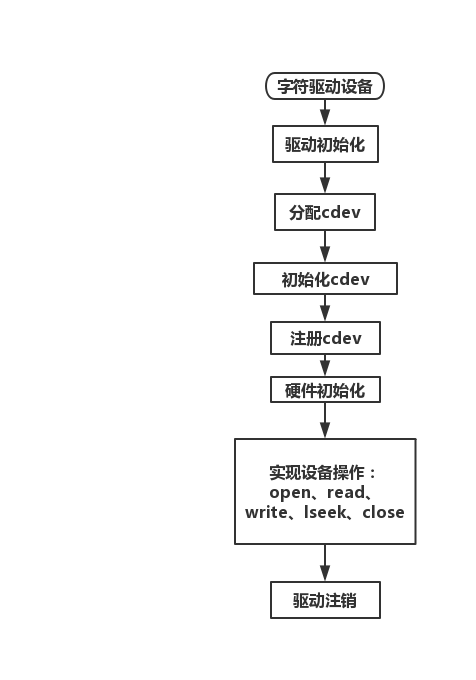
}

其中，字符设备文件和字符设备驱动通过主设备号建立连接。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。

2.开发者可自己选择一个数字作为主设备号，通过函数register\_chrdev\_region向内核申请使用，此方法的缺点在于，如果申请使用的设备号已被内核中其它的驱动使用了，则申请会失败。

3.在驱动退出时，应使用unregister\_chrdev\_region函数释放设备号。

1. **实现过程**
2. **流程图**

****

1. **核心代码**

（1）驱动初始化：

static int \_\_init globalvar\_init(void) //初始化模块的操作

{

int ret, err;

dev\_t devno=MKDEV(dev\_major, dev\_minor);

//注册设备号

if(dev\_major)

{

ret=register\_chrdev\_region(devno, 1, "globalvar");

}

else

{

ret=alloc\_chrdev\_region(&devno, dev\_minor, 1, "globalvar");

dev\_major=MAJOR(devno);

}

if(ret<0)

{

printk("globalvar register failure!\n");

globalvar\_exit(); //如果注册设备号失败就退出。

return ret;

}

else

{

printk("globalvar register success!\n");

}

//为设备分配内核空间

my\_dev=kmalloc(sizeof(struct globalvar\_dev), GFP\_KERNEL);

if(!my\_dev)

{

ret=-ENOMEM;

printk("create device failed!\n");

}

else //初始化设备，添加设备

{

my\_dev->global\_var=0; //设备变量初始化为0

cdev\_init(&my\_dev->cdev, &globalvar\_fos); //初始化设备中的cdev结构

my\_dev->cdev.owner=THIS\_MODULE; //初始化cdev中的所有者字段

//向内核添加cdev结构，注意到此时用到了devno

err=cdev\_add(&my\_dev->cdev, devno, 1);

if(err<0) //如果添加字符设备失败，打印错误信息

printk("add charater device failure!\n");

else

printk("add charater device success!\n");

}

return ret;

}

（2）实现设备操作：

//打开操作

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

{

struct globalvar\_dev \*dev;

//根据inode结构的cdev字段，获得整个设备结构的指针

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

//分配并填写置于filp->private\_data里的数据结构，private\_data是跨系统调用时

//保存状态信息的非常有用的资源

filp->private\_data=dev;

return 0;

}

//读操作

ssize\_t globalvar\_read(struct file \*filp, char \*buf, size\_t len, loff\_t \*off)

{

struct globalvar\_dev \*dev=filp->private\_data; //获取已指向分配数据的指针

//将设备变量值复制到用户空间

if(copy\_to\_user(buf, &dev->global\_var, sizeof(int)))

return -EFAULT;

return sizeof(int); //返回读取数据字节数

}

//写操作

ssize\_t globalvar\_write(struct file \*filp, const char \*buf, size\_t len, loff\_t \*off)

{

struct globalvar\_dev \*dev=filp->private\_data; //获取已指向分配数据的指针

//将用户空间值复制到设备变量

if(copy\_from\_user(&dev->global\_var, buf, sizeof(int)))

return -EFAULT;

return sizeof(int);

}

（3）驱动注销：

static void \_\_exit globalvar\_exit(void) //退出模块时的操作

{

dev\_t devno=MKDEV(dev\_major, dev\_minor);

cdev\_del(&my\_dev->cdev); //从系统中移除一个字符设备

kfree(my\_dev); //释放自定义的设备结构

unregister\_chrdev\_region(devno, 1); //注销已注册的驱动程序

printk("globalvar unregister success!\n");

}

//释放操作

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

{

return 0;

}

（4）Makefile：

obj-m:=globalvar.o

KERNELDIR:=/usr/src/linux-headers-4.10.0-28-generic

PWD:=$(shell pwd)

modules:

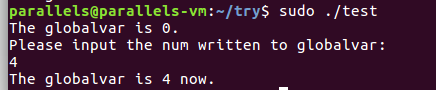
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

modules\_install:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules\_install

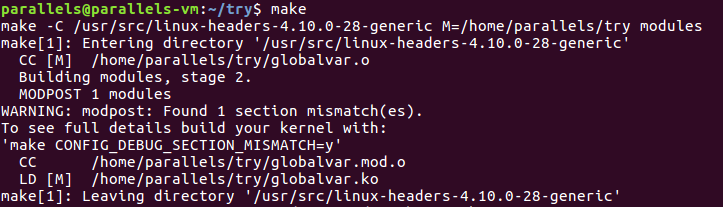
1. **运行结果**

使用test文件对字符驱动设备进行测试，结果表明驱动可以完成读写操作：

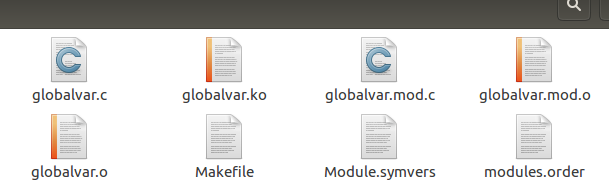


1. **运行截图**

（1）编译设备驱动程序：



（2）使用ismod命令加载驱动后，查看设备情况：

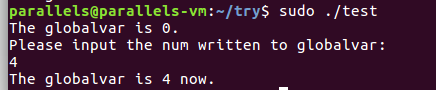




（3）查看设备号：



（4）执行测试文件：



1. **实验总结**
2. **实验过程分析**

本次实验是分两步进行的，先创建了字符设备驱动，编译并加载成功后，再编写test读写文件，测试驱动使用情况。实验过程中遇到了一些bug，总结如下：

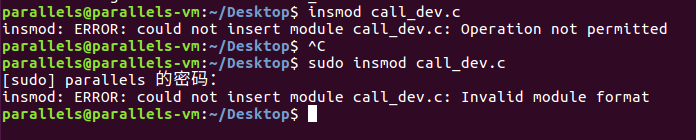
（1）make时报错nothiing to be done for ‘call\_dev.c’



在对设备驱动文件编译时，错误输入的命令行为make+文件名，正确的编译方法应该是cd进入文件目录下，直接使用make命令。

此外，在编译时曾遇到makefile文件内的错误，makefile文件内的命令行首格不应使用空格，而只能使用tab键。

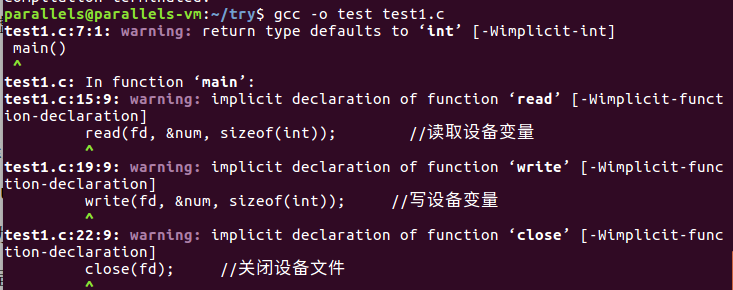
（2）insmod的两种错误

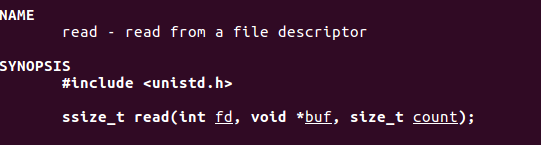


在使用insmod时，第一种错误是没有授予管理员权限，使用sudo即可。

使用insmod的第二种错误invaild module format，造成此错误的原因是makefile文件中的内核路径不对。应使用ls命令在/usr/src目录下查看内核版本，并将makefile中的路径具体为内核路径。

（3）编译测试文件时的警告





此类警告的原因是缺少对应的头文件声明，查看read相关的头文件，将其声明补入测试文件中即可成功编译。

1. **心得体会**

本次实验使我建立了对linux内核和驱动设备的深刻认识。在linux中，几乎一切都是文件，字符设备也是以文件形式存在于系统中，其与普通文件的区别在于，字符设备信息的前两列分别表示主设备号和此设备号，另外，字符设备也不像普通文件，在硬盘上占有正在运行的数据区。在linux用户空间中，要与内核驱动交互，首先需要为驱动程序申请一个一个设备号。主设备号用于标示与设备文件相连的驱动程序，次设备号被驱动程序用来辨别操作的是哪一个设备。

设备驱动是内核的一部分，能够完成：（1）对设备的初始化和释放；（2）把数据从内核传送到硬件和从硬件读取数据；（3）读取应用程序传送给设备文件的数据和回送应用程序请求的数据；（4）检测和处理设备出现的错误。

在此知识背景下，可以多加尝试建立不同的设备驱动程序，以实现更多复杂的功能。

附录 源代码

1.globalvar.c文件

#include <linux/module.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/uaccess.h>

#include <linux/slab.h>

MODULE\_LICENSE("GPL");

int globalvar\_open(struct inode \*, struct file \*);

int globalvar\_release(struct inode \*, struct file \*);

ssize\_t globalvar\_read(struct file \*, char \*, size\_t, loff\_t \*);

ssize\_t globalvar\_write(struct file \*, const char \*, size\_t, loff\_t \*);

int dev\_major=0;

int dev\_minor=0;

struct file\_operations globalvar\_fos=

{

.owner=THIS\_MODULE,

.open=globalvar\_open,

.release=globalvar\_release,

.read=globalvar\_read,

.write=globalvar\_write,

};

struct globalvar\_dev

{

int global\_var; //代表要操作的设备

struct cdev cdev; //内核中表示字符设备的结构

};

struct globalvar\_dev \*my\_dev;

static void \_\_exit globalvar\_exit(void) //退出模块时的操作

{

dev\_t devno=MKDEV(dev\_major, dev\_minor);

cdev\_del(&my\_dev->cdev); //从系统中移除一个字符设备

kfree(my\_dev); //释放自定义的设备结构

unregister\_chrdev\_region(devno, 1); //注销已注册的驱动程序

printk("globalvar unregister success!\n");

}

static int \_\_init globalvar\_init(void) //初始化模块的操作

{

int ret, err;

dev\_t devno=MKDEV(dev\_major, dev\_minor);

//注册设备号

if(dev\_major)

{

ret=register\_chrdev\_region(devno, 1, "globalvar");

}

else

{

ret=alloc\_chrdev\_region(&devno, dev\_minor, 1, "globalvar");

dev\_major=MAJOR(devno);

}

if(ret<0)

{

printk("globalvar register failure!\n");

globalvar\_exit(); //如果注册设备号失败就退出。这个有问题？

return ret;

}

else

{

printk("globalvar register success!\n");

}

//为设备分配内核空间

my\_dev=kmalloc(sizeof(struct globalvar\_dev), GFP\_KERNEL);

if(!my\_dev)

{

ret=-ENOMEM;

printk("create device failed!\n");

}

else //初始化设备，添加设备

{

my\_dev->global\_var=0; //设备变量初始化为0

cdev\_init(&my\_dev->cdev, &globalvar\_fos); //初始化设备中的cdev结构

my\_dev->cdev.owner=THIS\_MODULE; //初始化cdev中的所有者字段

//向内核添加cdev结构，注意到此时用到了devno

err=cdev\_add(&my\_dev->cdev, devno, 1);

if(err<0) //如果添加字符设备失败，打印错误信息

printk("add charater device failure!\n");

else

printk("add charater device success!\n");

}

return ret;

}

//打开操作

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

{

struct globalvar\_dev \*dev;

//根据inode结构的cdev字段，获得整个设备结构的指针

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

//分配并填写置于filp->private\_data里的数据结构，private\_data是跨系统调用时

//保存状态信息的非常有用的资源

filp->private\_data=dev;

return 0;

}

//释放操作

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

{

return 0;

}

//读操作

ssize\_t globalvar\_read(struct file \*filp, char \*buf, size\_t len, loff\_t \*off)

{

struct globalvar\_dev \*dev=filp->private\_data; //获取已指向分配数据的指针

//将设备变量值复制到用户空间

if(copy\_to\_user(buf, &dev->global\_var, sizeof(int)))

return -EFAULT;

return sizeof(int); //返回读取数据字节数

}

//写操作

ssize\_t globalvar\_write(struct file \*filp, const char \*buf, size\_t len, loff\_t \*off)

{

struct globalvar\_dev \*dev=filp->private\_data; //获取已指向分配数据的指针

//将用户空间值复制到设备变量

if(copy\_from\_user(&dev->global\_var, buf, sizeof(int)))

return -EFAULT;

return sizeof(int);

}

module\_init(globalvar\_init);

module\_exit(globalvar\_exit);

2.Makefile文件

obj-m:=globalvar.o

KERNELDIR:=/usr/src/linux-headers-4.10.0-28-generic

PWD:=$(shell pwd)

modules:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

modules\_install:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules\_install

3.test文件

obj-m:=globalvar.o

KERNELDIR:=/usr/src/linux-headers-4.10.0-28-generic

PWD:=$(shell pwd)

modules:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

modules\_install:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules\_install