**基于Linux的字符设备驱动程序的设计**

**1 选题意义**

驱动程序在 Linux 内核里扮演着特殊的角色. 它们是截然不同的"黑盒子", 使硬件的特殊的一部分响应定义好的内部编程接口. 它们完全隐藏了设备工作的细节. 用户的活动通过一套标准化的调用来进行,这些调用与特别的驱动是独立的; 设备驱动的角色就是将这些调用映射到作用于实际硬件的和设备相关的操作上. 这个编程接口是这样, 驱动可以与内核的其他部分分开建立, 并在需要的时候在运行时"插入". 这种模块化使得 Linux 驱动易写, 以致于目前有几百个驱动可用.

尽管编写设备代码并不一定比编写应用程序更困难，但它需要掌握一些新函数库，并考虑一些新问题，而这些问题是在应用程序空间里不曾遇到的。在应用程序空间写程序，内核能够为犯的一些错误提供一张安全网，但当我们工作在内核空间时，这张安全网已不复存在。因为内核代码对计算机有绝对的控制权，它能够阻止其他任何进程的执行，所以编写的设备代码绝对小心不能滥用这种权利。

在 Linux 设备驱动中，字符设备驱动较为基础,所以本次实验设计一个简单的字符设备驱动程序，然后通过模块机制加载该驱动，并通过一个测试程序来检验驱动设计的正确与否，并对出现的问题进行调试解决。

**2 技术路线**

模块实际上是一种目标对象文件(后缀名为ko )，没有链接，不能独立运行，但是其代码可以在运行时链接到系统中作为内核的一部分运行或从内核中取下，从而可以动态扩充内核的功能。模块有一个入口（init\_module()）和一个出口(exit\_module())函数，分别是模块加载和卸载时执行的操作，加载模块使用insmod命令，卸载使用rmmod命令。

字符设备以字节为单位进行数据处理，一般不适用缓存。大多数字符设备仅仅是数据通道，只能按照顺序读写。主设备号表示设备对应的驱动程序,次设备号用来区分具体设备的实例。LINUX为文件和设备提供一致的用户接口，对用户来说，设备文件与普通文件并无区别，设备文件也可以挂接到任何需要的地方。对于字符设备而言，file\_operations结构体中的成员函数是字符设备驱动程序设计的主体内容，这些函数实际会在应用程序进行Linux 的open()、write()、read()、close()等系统调用时最终被调用。驱动程序的三层界面 ：驱动程序与操作系统内核的接口，通过file\_operations数据结构来完成；驱动程序与系统引导的接口，这部分驱动程序对设备进行初始化；驱动程序与设备的接口，描述驱动程序如何与设备进行交互，这与具体设备密切相关。

**3 详细设计**

**3．1 cdev 结构体**

本论文基于虚拟的globalmem设备进行字符设备驱动，globalmem意味着“全局内存”，在globalmem字符设备驱动中会分配一片大小为GLOBALMEM\_ SIZE（4KB）的内存空间，并在驱动中提供针对该片内存的读写、控制和定位函数，以供用户空间的进程能通过Linux 系统调用访问这片内存。

在 Linux 2.6 内核中使用cdev结构体描述字符设备，cdev结构体的定义如下所示：

struct cdev

{

struct kobject kobj; /\* 内嵌的kobject对象\*/

struct module \*owner; /\*所属模块\*/

struct file\_operations \*ops; /\*文件操作结构体\*/

struct list\_head list;

dev\_t dev; /\*设备号\*/

unsigned int count;

};

cdev结构体的dev\_t 成员定义了设备号，为32 位，其中高12 位为主设备号，低20位为次设备号。使用下列宏可以从dev\_t获得主设备号和次设备号。

MAJOR(dev\_t dev)

MINOR(dev\_t dev)

而使用下列宏则可以通过主设备号和设备号生成dev\_t。

MKDEV(int major, int minor)

cdev 结构体的另一个重要成员file\_operations 定义了字符设备驱动提供给虚拟文件系统的接口函数。

Linux 2.6 内核提供了一组函数用于操作cdev结构体，如下所示：

void cdev\_init(struct cdev \*, struct file\_operations \*);

struct cdev \*cdev\_alloc(void);

void cdev\_put(struct cdev \*p);

int cdev\_add(struct cdev \*, dev\_t, unsigned);

void cdev\_del(struct cdev \*);

cdev\_init()函数用于初始化cdev 的成员，并建立cdev 和file\_operations 之间的连接，其源代码如下所示。

void cdev\_init(struct cdev \*cdev, struct file\_operations \*fops)

{

memset(cdev, 0, sizeof \*cdev);

INIT\_LIST\_HEAD(&cdev->list);

cdev->kobj.ktype = &ktype\_cdev\_default;

kobject\_init(&cdev->kobj);

cdev->ops = fops; /\*将传入的文件操作结构体指针赋值给cdev的ops\*/

}

cdev\_alloc()函数用于动态申请一个cdev内存，其源代码如代码如下所示：

struct cdev \*cdev\_alloc(void)

{

struct cdev \*p=kmalloc(sizeof(struct cdev),GFP\_KERNEL); /\*分配cdev的内存\*/

if (p) {

memset(p, 0, sizeof(struct cdev));

p->kobj.ktype = &ktype\_cdev\_dynamic;

INIT\_LIST\_HEAD(&p->list);

kobject\_init(&p->kobj);

}

return p;

}

cdev\_add()函数和cdev\_del()函数分别向系统添加和删除一个cdev，完成字符设备的注册和注销。对cdev\_add()的调用通常发生在字符设备驱动模块加载函数中，而对cdev\_del()函数的调用则通常发生在字符设备驱动模块卸载函数中。

**3.2 分配和释放设备号**

在调用cdev\_add() 函数向系统注册字符设备之前，应首先调用

register\_chrdev\_region()或alloc\_chrdev\_region()函数向系统申请设备号这两个函数的原型如下：

int register\_chrdev\_region(dev\_t from, unsigned count, const char\*name);

int alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count,const char \*name);

register\_chrdev\_region() 函数用于已知起始设备的设备号的情况；而alloc\_chrdev\_region()用于设备号未知，向系统动态申请未被占用的设备号的情况。函数调用成功之后，会把得到的设备号放入第一个参数dev 中。alloc\_chrdev\_region()与register\_chrdev\_region()对比的优点在于它会自动避开设备号重复的冲突。相 反 地 ， 在调用cdev\_del() 函数从系统注销字符设备之后，

unregister\_chrdev\_region()应该被调用以释放原先申请的设备号，这个函数的原型如下：

void unregister\_chrdev\_region(dev\_t from, unsigned count);

**3.3 file\_operations结构体**

file\_operations结构体中的成员函数是字符设备驱动程序设计的主体内容，这些函数实际会在应用程序进行Linux 的open()、write()、read()、close()等系统调用时最终被调用。

**3.4 Linux字符设备驱动的组成**

在 Linux 系统中，字符设备驱动由如下几个部分组成。

**3.4.1．字符设备驱动模块加载与卸载函数**

在字符设备驱动模块加载函数中应该实现设备号的申请和cdev的注册，而在卸载函数中应实现设备号的释放和cdev的注销。

我们通常习惯将设备定义为一个设备相关的结构体，其包含该设备所涉及的cdev、私有数据及信号量等信息。常见的设备结构体、模块加载和卸载函数形式如下所示：

//设备结构体

struct xxx\_dev\_t

{

struct cdev cdev;

...

} xxx\_dev;

//设备驱动模块加载函数

static int \_ \_init xxx\_init(void)

{

...

cdev\_init(&xxx\_dev.cdev, &xxx\_fops); //初始化cdev

xxx\_dev.cdev.owner = THIS\_MODULE;

//获取字符设备号

if (xxx\_major)

{

register\_chrdev\_region(xxx\_dev\_no, 1, DEV\_NAME);

}

else

{

alloc\_chrdev\_region(&xxx\_dev\_no, 0, 1, DEV\_NAME);

}

ret = cdev\_add(&xxx\_dev.cdev, xxx\_dev\_no, 1); //注册设备

...

}

/\*设备驱动模块卸载函数\*/

static void \_ \_exit xxx\_exit(void)

{

unregister\_chrdev\_region(xxx\_dev\_no, 1); //释放占用的设备号

cdev\_del(&xxx\_dev.cdev); //注销设备

...

}

**3.4.2．字符设备驱动的file\_operations 结构体中成员函数**

file\_operations 结构体中成员函数是字符设备驱动与内核的接口，是用户空间对Linux 进行系统调用最终的落实者。大多数字符设备驱动会实现read()、write()和ioctl()函数，常见的字符设备驱动的这3 个函数的形式如下所示：

/\* 读设备\*/

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

{

...

copy\_to\_user(buf, ..., ...);

...

}

/\* 写设备\*/

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

{

...

copy\_from\_user(..., buf, ...);

...

}

/\* ioctl函数\*/

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

{

...

switch (cmd)

{

case XXX\_CMD1:

...

break;

case XXX\_CMD2:

...

break;

default:

/\* 不能支持的命令\*/

return - ENOTTY;

}

return 0;

}

设备驱动的读函数中，filp是文件结构体指针，buf是用户空间内存的地址，该地址在内核空间不能直接读写，count 是要读的字节数，f\_pos 是读的位置相对于文件开头的偏移。设备驱动的写函数中，filp是文件结构体指针，buf是用户空间内存的地址，该地址在内核空间不能直接读写，count 是要写的字节数，f\_pos 是写的位置相对于文件开头的偏移。

由于内核空间与用户空间的内存不能直接互访，因此借助函数copy\_from\_user()完成用户空间到内核空间的复制，函数copy\_to\_user()完成内核空间到用户空间的复制。

copy\_from\_user()和copy\_to\_user()的原型如下所示：

unsigned long copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long count);

unsigned long copy\_to\_user(void \_ \_user \*to, const void \*from, unsigned long count);

上述函数均返回不能被复制的字节数，因此，如果完全复制成功，返回值为0。

读和写函数中的\_ \_user 是一个宏，表明其后的指针指向用户空间，这个宏定义如下：

#ifdef \_ \_CHECKER\_ \_

# define \_ \_user \_ \_attribute\_ \_((noderef, address\_space(1)))

#else

# define \_ \_user

#endif

I/O 控制函数的cmd参数为事先定义的I/O 控制命令，而arg为对应于该命令的参数。例如对于串行设备，如果SET\_BAUDRATE 是一个设置波特率的命令，那后面的arg就应该是波特率值。

在字符设备驱动中，需要定义一个file\_operations 的实例，并将具体设备驱动的函数赋值给file\_operations的成员，如代码下 所示。

struct file\_operations xxx\_fops =

{

.owner = THIS\_MODULE,

.read = xxx\_read,

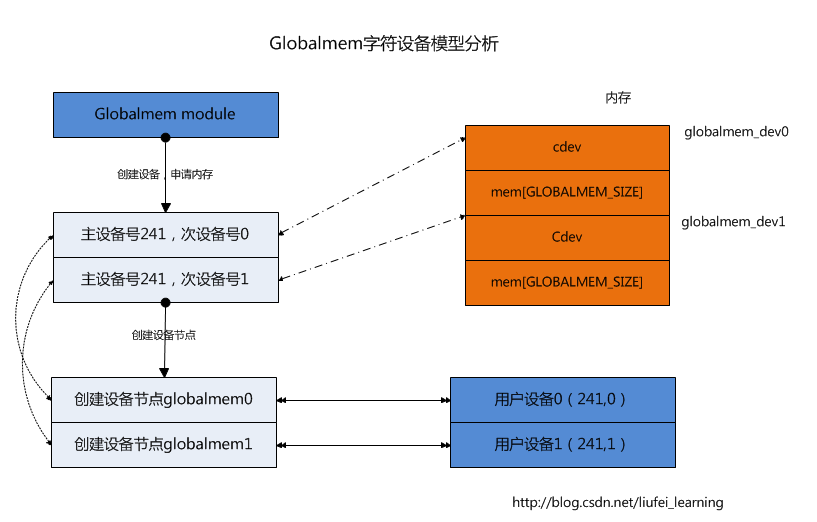
.write = xxx\_write,

.ioctl = xxx\_ioctl,

...

};

上述xxx\_fops 在代码cdev\_init（&xxx\_dev.cdev, &xxx\_fops)的语句中被建立与cdev的连接。



**4 程序源代码**

**4.1字符设备驱动主程序**

#include <linux/module.h>

#include <linux/types.h>

#include <linux/fs.h>

#include <linux/errno.h>

#include <linux/mm.h>

#include <linux/sched.h>

#include <linux/init.h>

#include <linux/cdev.h>

#include <asm/io.h>

#include <asm/system.h>

#include <asm/uaccess.h>

#define GLOBALMEM\_SIZE 0x1000 /\*全局内存最大 4KB\*/

#define MEM\_CLEAR 0x1 /\*清零全局内存\*/

#define GLOBALMEM\_MAJOR 251 /\*预设的 globalmem 的主设备号\*/

static globalmem\_major = GLOBALMEM\_MAJOR;

/\*globalmem 设备结构体\*/

struct globalmem\_dev

{

struct cdev cdev; /\*cdev 结构体\*/

unsigned char mem[GLOBALMEM\_SIZE]; /\*全局内存\*/

};

struct globalmem\_dev \*globalmem\_devp; /\*设备结构体指针\*/

/\*文件打开函数\*/

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

{

/\*将设备结构体指针赋值给文件私有数据指针\*/

filp-> private\_data = globalmem\_devp;

return 0;

}

/\*文件释放函数\*/

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

{

return 0;

}

/\* ioctl 设备控制函数 \*/

static int globalmem\_ioctl(struct inode \*inodep, struct file \*filp, unsigned int cmd, unsigned long arg)

{

struct globalmem\_dev \*dev = filp->private\_data;/\*获得设备结构体指针\*/

switch (cmd)

{

case MEM\_CLEAR:

memset(dev->mem, 0, GLOBALMEM\_SIZE);

printk(KERN\_INFO "globalmem is set to zero\n");

break;

default:

return EINVAL;

}

return 0;

}

/\*读函数\*/

static ssize\_t globalmem\_read(struct file \*filp, char \_\_user \*buf, size\_t size,loff\_t \*ppos)

{

unsigned long p = \*ppos;

unsigned int count = size;

int ret = 0;

struct globalmem\_dev \*dev = filp->private\_data; /\* 获得设备结构体指针\*/

/\*分析和获取有效的写长度\*/

if (p >= GLOBALMEM\_SIZE)

return count ? ENXIO:0;

if (count > GLOBALMEM\_SIZE-p)

count = GLOBALMEM\_SIZE-p;

/\*内核空间→用户空间\*/

if (copy\_to\_user(buf, (void\*)(dev->mem + p), count)) /\*返回不能复制的字节数\*/

{

ret =-EFAULT;

}

else

{

\*ppos += count;

ret = count;

printk(KERN\_INFO "read %d bytes(s) from %d\n", count, p);

}

return ret;

}

/\*写函数\*/

static ssize\_t globalmem\_write(struct file \*filp, const char \_\_user \*buf, size\_t size, loff\_t \*ppos)

{

unsigned long p = \*ppos;

unsigned int count = size;

int ret = 0;

struct globalmem\_dev \*dev = filp->private\_data; /\* 获得设备结构指针\*/

/\*分析和获取有效的写长度\*/

if (p >= GLOBALMEM\_SIZE)

return count ? ENXIO:0;

if (count > GLOBALMEM\_SIZE-p)

count = GLOBALMEM\_SIZE-p;

/\*用户空间→内核空间\*/

if (copy\_from\_user(dev->mem + p, buf, count))

ret =-EFAULT;

else

{

\*ppos += count;

ret = count;

printk(KERN\_INFO "written %d bytes(s) from %d\n", count, p);

}

return ret;

}

/\* seek 文件定位函数 \*/

static loff\_t globalmem\_llseek(struct file \*filp, loff\_t offset, int orig)

{

loff\_t ret = 0;

switch (orig)

{

/\*相对文件开始位置偏移\*/

case 0:

if (offset < 0)

{

ret = EINVAL;

break;

}

if ((unsigned int)offset > GLOBALMEM\_SIZE)

{

ret = EINVAL;

break;

}

filp->f\_pos = (unsigned int)offset;

ret = filp->f\_pos;

break;

/\*相对文件当前位置偏移\*/

case 1:

if ((filp->f\_pos + offset) > GLOBALMEM\_SIZE)

{

ret = EINVAL;

break;

}

if ((filp->f\_pos + offset) < 0)

{

ret = EINVAL;

break;

}

filp->f\_pos += offset;

ret = filp->f\_pos;

break;

default:

ret = EINVAL;

break;

}

return ret;

}

/\*文件操作结构体\*/

static const struct file\_operations globalmem\_fops =

{

.owner = THIS\_MODULE,

.llseek = globalmem\_llseek,

.read = globalmem\_read,

.write = globalmem\_write,

.ioctl = globalmem\_ioctl,

.open = globalmem\_open,

.release = globalmem\_release,

};

/\*初始化并注册 cdev\*/

static void globalmem\_setup\_cdev(struct globalmem\_dev \*dev, int index)

{

int err, devno = MKDEV(globalmem\_major, index);

cdev\_init(&dev->cdev, &globalmem\_fops);

dev->cdev.owner = THIS\_MODULE;

dev->cdev.ops = &globalmem\_fops;

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

if (err)

printk(KERN\_NOTICE "Error %d adding LED%d", err, index);

}

/\*设备驱动模块加载函数\*/

int globalmem\_init(void)

{

int result;

dev\_t devno = MKDEV(globalmem\_major, 0);

/\* 申请设备号\*/

if (globalmem\_major)

result = register\_chrdev\_region(devno, 1, "globalmem");

else /\* 动态申请设备号 \*/

{

result = alloc\_chrdev\_region(&devno, 0, 1, "globalmem");

globalmem\_major = MAJOR(devno);

}

if (result < 0)

return result;

/\* 动态申请设备结构体的内存\*/

globalmem\_devp = kmalloc(sizeof(struct globalmem\_dev), GFP\_KERNEL);

/\*申请失败\*/

if (!globalmem\_devp)

{

result = ENOMEM;

goto fail\_malloc;

}

memset(globalmem\_devp, 0, sizeof(struct globalmem\_dev));

globalmem\_setup\_cdev(globalmem\_devp, 0);

return 0;

fail\_malloc: unregister\_chrdev\_region(devno, 1);

return result;

}

/\*模块卸载函数\*/

void globalmem\_exit(void)

{

cdev\_del(&globalmem\_devp->cdev); /\*注销 cdev\*/

/\*释放设备结构体内存\*/

kfree(globalmem\_devp);

unregister\_chrdev\_region(MKDEV(globalmem\_major, 0), 1); /\*释放备号\*/

}

MODULE\_AUTHOR("Song Baohua");

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

module\_param(globalmem\_major, int, S\_IRUGO);

module\_init(globalmem\_init);

module\_exit(globalmem\_exit);

**4.2测试程序：**

//测试函数定义

#include <sys/types.h>

#include <sys/stat.h>

#include <string.h>

#include <stdio.h>

#include <fcntl.h>

#define MAX\_SIZE 0x1000 //定义最大缓冲区为4KB

int main(void)

{

int fd,choice;

char buf[MAX\_SIZE];

char tmp[MAX\_SIZE];

memset(tmp, 0, sizeof(tmp));

memset(buf, 0, sizeof(buf));//初始化内存空间

init:

//选择界面

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("向设备写数据 1 \n");

printf("从设备读数据 2 \n");

printf("退出测试程序 0 \n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

scan:

printf("请输入选择项：\n");

scanf("%d",&choice);//录入选择项

if(choice != 0 && choice != 1 && choice != 2) {

printf("请输入正确的选择（0或1或2）！\n");

goto scan;

}

//选择写

else if(choice == 1) {fd = open("/dev/global\_mem\_driver", O\_RDWR, S\_IRUSR|S\_IWUSR);//打开设备

if (fd > 0) {

printf("请输入您要写入的字符串(注意只能输入字母或数字)：\n");

getchar(); // 用于消除换行符 "\n"

scanf("%[a-zA-Z0-9]%",buf);//录入要输入的字符串printf("正将您输入的字符串 [%s] 写入字符设备chardev.\n", buf);

printf("共写了[%d]个字符\n", write(fd, &buf, strlen(buf)+1)-1);//将字符串写入设备，并显示写入的字符个数

printf("\n");

}

else {

return -1;

}

close(fd);//释放设备

}

//选择读

else if(choice == 2) {fd = open("/dev/global\_mem\_driver ", O\_RDWR, S\_IRUSR|S\_IWUSR);//打开设备

if (fd > 0) {

printf("开始从设备中读取字符...\n");

printf("要读的个数为：[%d]\n", read(fd, &tmp, 50));//从设备中读出字符窜

printf("读取到的字符串：[%s].\n", tmp);

printf("\n");

}

else {

return -1;

}

close(fd);//释放设备

}

//选择退出

else {

return 0;

}

goto init;

return 0;

}

**4.3 Makefile文件：**

TARGET=global\_mem\_driver

OBJS=global\_mem\_driver.o

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

PWD= $(shell pwd)

obj-m:=$(TARGET).o

default:

make -C $(KDIR) SUBDIRS=$(PWD) modules

clean:

rm -f \*.o \*.ko \*.mod.c

-include $(KDIR)/Rules.make

现在我来说明一下这个Makefile：  
obj-m ：这个变量是指定你要声称哪些模块模块的格式为 obj-m := <模块名>.o。  
KDIR ：这是我们正在运行的操作系统内核编译目录。也就是编译模块需要的环境  
PWD ：这是当前工作路径$(shell   )是make的一个内置函数。用来执行shell命令。

**5 编译及调试过程分析**

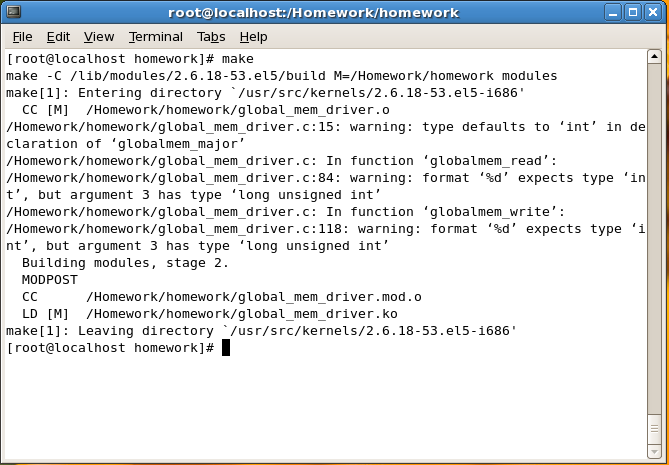


图1 make命令后截图

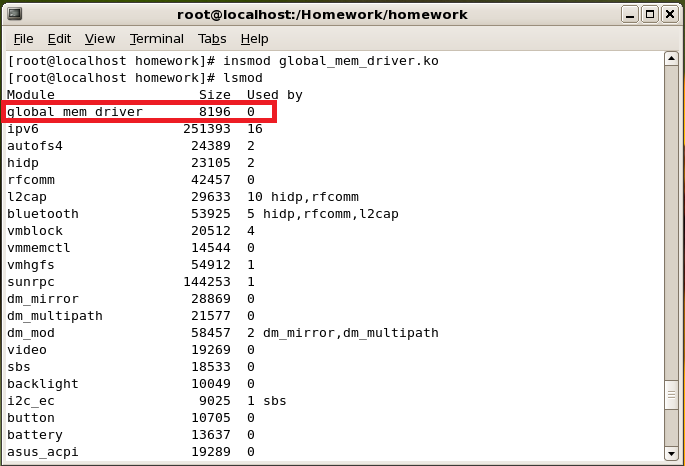


图2 将global\_mem\_driver模块加载成功

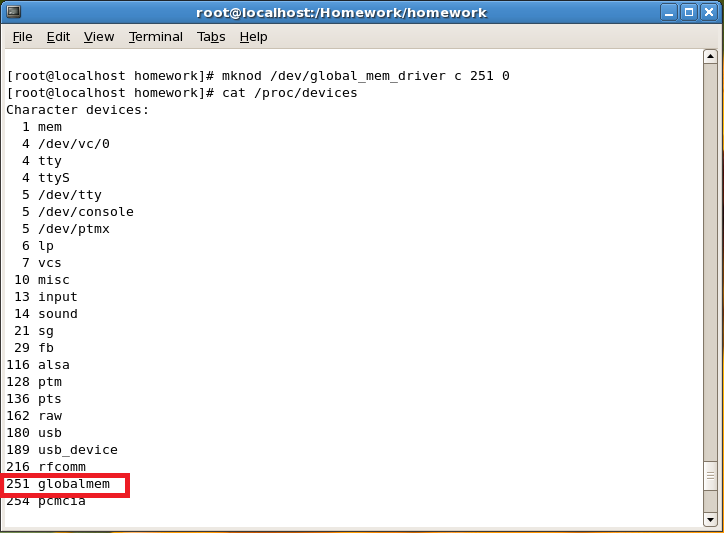


图3 创建设备节点并显示字符设备号

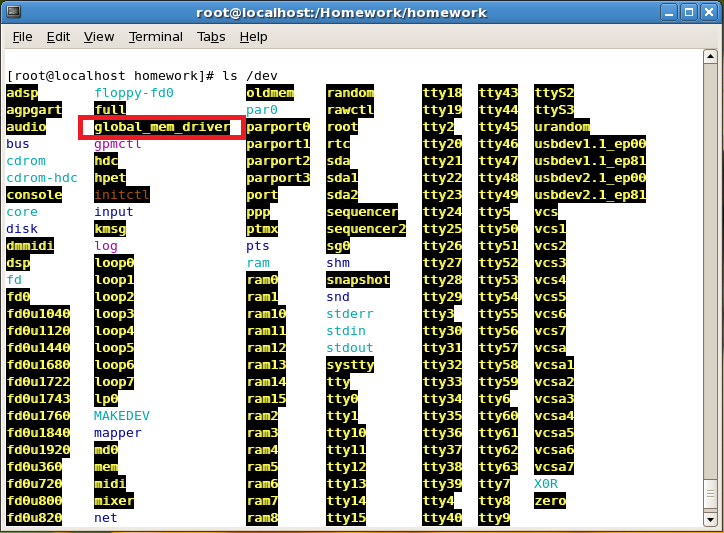


图5 显示创建的字符设备

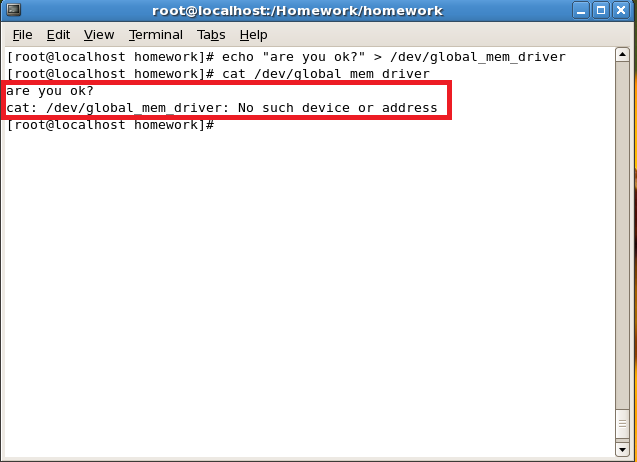


图6 将字符写进字符设备并回显

**备注：**

可以看到在图6中用红色圈框起的部分，第二行有可以看出有一个问题，我们通过strace cat /dev/global\_mem\_driver追踪如下：

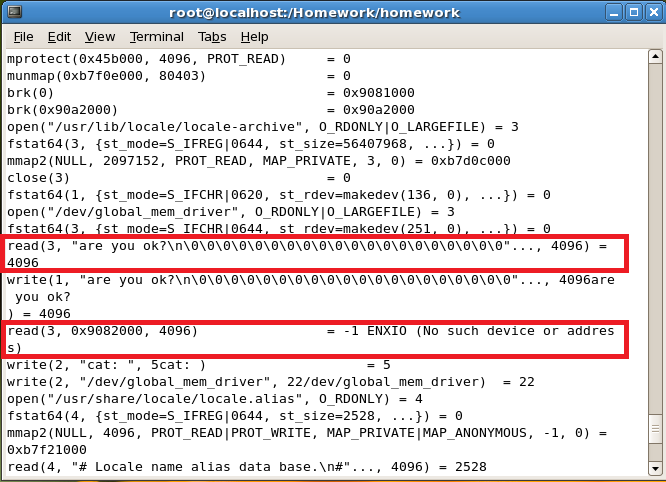


图 7 strace 进行追踪的结果

cat命令read设备内容，当返回值是大于0的时候，cat会继续进行read操作（每次读4096字节）。程序例子中第一次读4096个字节后，返回值4096，指针偏移也增加了4096，当第二次再进行读操作的时候其位置指针loff\_t \*ppos已经在4096的位置，所以在进行下图所示时出错。

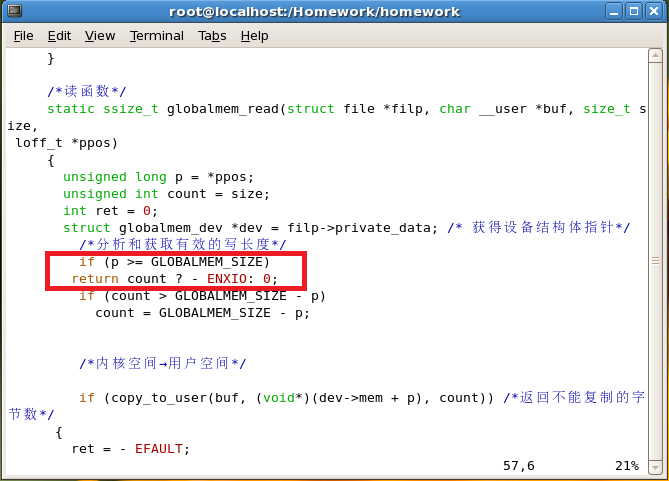


图8 驱动源程序设计的缺陷

其实这也算不上一个bug，因为我们并需要read函数在一次读请求反复的去读设备，请求一次读一次缓冲去就足够了，所以这个地方不会对测试程序有任何影响。

**6 测试结果**

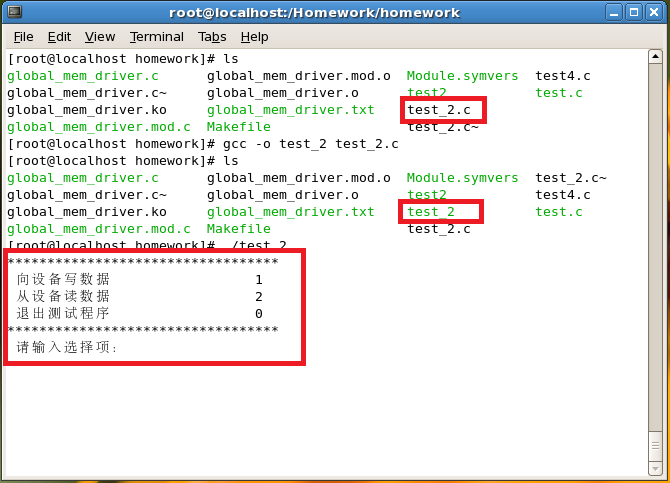


图9 测试程序编译及初始界面

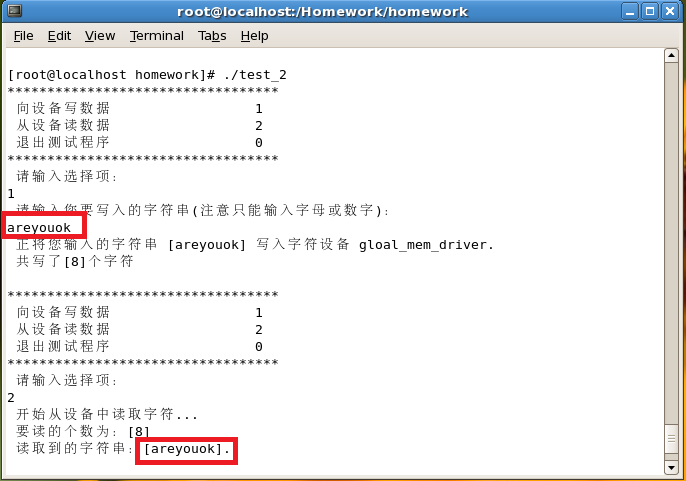


图 10 字符驱动读写操作测试

**7 工作小结**

本次实验让我对linux的驱动机制有了初步认识，懂得了驱动添加，驱动调用，驱动卸载的步骤。同时还加深了我对linux模块机制的了解，掌握了模块加载，模块卸载的方法。当然在编程过程中碰到了很多困难，不过通过查阅相关资料和网站，并经过自己的思考，终于取得了最终的结果。在最后的测试程序调试过程中，徐晓龙师兄给予了很大的帮助，在这里表示感谢！虽然在本次实验的过程中增加了一些对linux驱动和模块的了解，但是关于一些细节东西还有待于研究！

**参考文献：**

[1] 鸟哥 王世江 .鸟哥的Linux私房菜（基础篇）274-292.

[2] Jonahan Corbet**.** LINUX设备驱动程序（第三版）[M] **.** 北京

中国电力出版社，2006**.**

[3] 宋宝华 .Linux 设备驱动详解.

[4] Neil Matthew，Richard Stones**.** LINUX程序设计（第三版）[

北京：人民邮电出版社，2007**.**.

[5] 陈莉君. 深入分析linux 内核源代码 437-445

[6] 周来超，傅成华，刘宏玉**.** 基于Linux2.6内核的字符设备驱

程序设计[J] **.** 数字技术与应用，2010，7**.**

[7]李善平 陈文智.边干边学-Linux 内核指导 337-384