

Linux 驱动系列课-实验手册

(V1.1)

2023 年 4.30

修改历史

版本	作者	修改	备注
V1.0		内容创建	
V1.1		格式重排	

目 录

【实验 1】Linux 文件 IO 编程实验.....	4
《Linux 内核模块编程实验》	5
【实验 2】Linux 文件 IO 编程实验.....	5
1、编程：内核模块编程.....	5
2、编译：内核模块编译.....	6
3、运行：内核模块加载.....	6
《Linux 驱动接口实验》	7
【实验 3】CHRDEV 版注册字符设备驱动实验	7
1、编程.....	7
2、编译.....	8
3、运行测试.....	8
【实验 4】CDEV 版注册字符设备驱动实验.....	8
1、编程.....	8
2、编译.....	10
3、运行测试.....	10
【实验 5】打开关闭设备驱动.....	10
1、加载内核驱动.....	10
2、建立设备文件.....	12
3、执行应用测试.....	12
【实验 6】控制设备（IOCTL）	13
1、自定义头文件.....	13
2、驱动代码.....	13
2、建立设备文件.....	16
3、执行应用测试.....	16
【实验 7】读写设备（READ/WRITE）	17
1、内核驱动程序.....	17
2、应用测试程序.....	20

《Linux 系统编程实验》

【实验 1】Linux 文件 IO 编程实验

1、编程

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void)
{
    int fd;
    char buf[10]="hello\n";

    fd = open("/home/liy/test_file", O_RDWR | O_CREAT);
    write(fd, buf, sizeof(buf));
    close(fd);

    return 0;
}
```

1、头文件

2、man 手册

man 2 open

man 2 write

2、编译

```
gcc -o test_app test_app.c
```

3、运行

```
./test_app
```

4、思考

1) 程序运行状态切换如下:

- 用户态 -> 系统调用 -> 内核态 -> 返回用户态

2) 执行如下命令, 查看调用过程

- `strace ./test_app`

【说明】运行程序前加上 `strace`, 可以追踪到函数库调用过程。

《Linux 内核模块编程实验》

【实验 2】Linux 文件 IO 编程实验

1、编程: 内核模块编程

```
#include <linux/init.h>
#include <linux/module.h>
static int test_init(void)
{
    printk("hello kernel\n");
    return 0;
}
static void test_exit(void)
{
    printk("bye\n");
}

module_init(test_init);
module_exit(test_exit);

MODULE_LICENSE("GPL");
```

2、编译：内核模块编译

1) 内核模块编译 Makefile

```
obj-m := test.o

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

all:

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

clean:

    make -C $(KDIR) M=$(PWD) clean
```

注意：all 和 clean 下面的命令要严格用“tab”

2) 编译内核模块

在模块代码和模块 Makefile 所在目录执行“make”命令。

3、运行：内核模块加载

1) 加载模块

```
sudo insmod hello.ko
```

2) 查看日志

运行命令：dmesg

内核日志显示信息如下：

```
[156596.317933] hello world.
[156604.933930] hello exit!
```

3) 卸载模块

```
sudo rmmmod hello
```

注意：rmmmod 后面要加用 lsmod 查看到的模块名字'hello'而不是'hello.ko'

<<Linux 驱动接口实验>>

【实验 3】chrdev 版注册字符设备驱动实验

1、编程

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

struct file_operations test_fops;
int test_major;
int test_init(void)
{
    /* 动态分配代码 */
    test_major = register_chrdev(0, "test-driver", &test_fops); //返回大于
    0 的主设备号成功，返回非零失败

    if(test_major < 0){
        printk("register failed\n");
        return -1;
    }

    return 0;
}

void test_exit(void)
{
    unregister_chrdev(test_major, "test-driver");
}

module_init(test_init);
```

```
module_exit(test_exit);  
  
MODULE_LICENSE("GPL");
```

2、编译

1) 驱动模块编译 Makefile

```
Makefile  
obj-m := test_driver.o  
KDIR :=/lib/modules/$(shell uname -r)/build  
  
all:  
    make -C $(KDIR) M=$(PWD) modules  
clean:  
    make -C $(KDIR) M=$(PWD) clean
```

2) 执行模块编译命令

make

3、运行测试

- 1) 加载模块: insmod test_driver.ko
- 2) 查看是否注册成功: cat /proc/devices
- 3) 卸载模块: rmmod test_driver

【实验 4】cdev 版注册字符设备驱动实验

1、编程

```
#include <linux/module.h>  
#include <linux/init.h>  
#include <linux/fs.h>  
#include <linux/cdev.h>  
  
int major, minor;  
int devno;  
struct cdev *test_cdev;
```



```
struct file_operations test_fops;

int test_init(void)
{
    int ret;
    major = 0;//动态分配
    if(major)//静态分配
    {
        minor = 0;
        devno = MKDEV(major, minor);
        ret = register_chrdev_region(devno, 1, "new-char");
    }
    else//动态分配
    {
        ret = alloc_chrdev_region(&devno, 0, 1, "new-char");
        printk("ma[%d], mi[%d]\n", MAJOR(devno), MINOR(devno));
    }
    if(ret){
        printk("register region failed\n");
        goto fail;
    }

    test_cdev = cdev_alloc();

    cdev_init(test_cdev, &test_fops);

    ret = cdev_add(test_cdev, devno, 1);
    if(ret){
        printk("cdev add failed\n");
        goto fail1;
    }
    printk("hello new char\n");
    return 0;
fail1:
    unregister_chrdev_region(devno, 1);
fail:
    return ret;
}

void test_exit(void)
{
    cdev_del(test_cdev);
}
```

```
        unregister_chrdev_region(devno, 1);
        printk("bye\n");
    }
module_init(test_init);
module_exit(test_exit);

MODULE_LICENSE("GPL");
```

2、编译

1) 驱动模块编译 Makefile

```
Makefile
obj-m := test_driver.o
KDIR := /lib/modules/$(shell uname -r)/build

all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

2) 执行模块编译命令

make

3、运行测试

1) 加载模块: insmod test_driver.ko

2) 查看日志: dmesg 看到动态分配成功, 并且通过宏得到了主次设备号

3) 查看是否注册成功: cat /proc/devices

4) 卸载模块: rmmod test_driver

【实验 5】打开关闭设备驱动

1、加载内核驱动

1) 编程

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

int test_open(struct inode *node, struct file *filp)
{
    printk("%s\n", __FUNCTION__);
    return 0;
}

int test_close(struct inode *node, struct file *filp)
{
    printk("%s\n", __FUNCTION__);
    return 0;
}

struct file_operations test_fops = {
    .open = test_open,
    .release = test_close,
};

int test_init(void)
{
    int ret;
    ret = register_chrdev(241, "test-driver", &test_fops);
    if(ret){
        printk("register failed\n");
        return -1;
    }

    return 0;
}

void test_exit(void)
{
    unregister_chrdev(241, "test-driver");
}

module_init(test_init);
module_exit(test_exit);
```

```
MODULE_LICENSE("GPL");
```

2) 编译

驱动模块编译 Makefile

```
Makefile
obj-m := test_driver.o
KDIR :=/lib/modules/$(shell uname -r)/build

all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

编译驱动模块：make

3) 加载

加载模块：insmod test_driver.ko

查看是否注册成功：cat /proc/devices

2、建立设备文件

查看设备号：cat /proc/devices

建立设备文件：mknod /dev/test_dev c 241 0

查看是否创建成功：ls -l /dev/test_dev

3、执行应用测试

1) 编程

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
    int fd;
    fd = open("/dev/test_dev", O_RDWR);
```

```
        if(fd < 0){
            perror("open failed\n");
            return -1;
        }

        close(fd);

        return 0;
    }
}
```

2) 编译

```
gcc -o test_app test_app.c
```

3) 运行

```
./test_app
```

dmesg 查看内核日志看是否成功调用了 test_driver 驱动的 test_open

【实验 6】控制设备（ioctl）

1、自定义头文件

一个简单的命令定义头文件，驱动和应用函数都要包含这个头文件：

```
/*test_cmd.h*/
#ifndef _TEST_CMD_H
#define _TEST_CMD_H

#define TEST_ON 0
#define TEST_OFF 1

#endif /*_TEST_CMD_H*/
```

2、驱动代码

1) 编程

```
#include <linux/init.h>
#include <linux/module.h>
```

```
#include <linux/fs.h>
#include <asm/uaccess.h>
#include "test_cmd.h"

void arm_on(void)
{
    //驱动硬件寄存器代码

    printk("%s\n", __FUNCTION__);
}

void arm_off(void)
{
    printk("%s\n", __FUNCTION__);
}

int test_open(struct inode *node, struct file *filp)
{
    printk("kernel %s\n", __FUNCTION__);
    return 0;
}

int test_close(struct inode *node, struct file *filp)
{
    printk("%s\n", __FUNCTION__);
    return 0;
}

long test_unlocked_ioctl (struct file *filp, unsigned int cmd, unsigned long
args)
{
    switch(cmd) {
        case TEST_ON:
            arm_on();
            break;
        case TEST_OFF:
            arm_off();
            break;
        default:
            printk("unknow ioctl cmd\n");
            return -1;
    }
}
```

```

    }
    return 0;
}

struct file_operations test_fops = {
    .open = test_open,
    .release = test_close,
    .unlocked_ioctl = test_unlocked_ioctl,
};

int test_init(void)
{
    int ret = 0;
    ret = register_chrdev(241, "test-driver", &test_fops);
    if(ret){
        printk("register failed\n");
        return -1;
    }

    printk("1 kernel register right\n");

    return 0;
}

void test_exit(void)
{
    unregister_chrdev(241, "test-driver");
}

module_init(test_init);
module_exit(test_exit);

MODULE_LICENSE("GPL");

```

2) 编译

驱动模块编译 Makefile

```

Makefile
obj-m := test_driver.o
KDIR :=/lib/modules/$(shell uname -r)/build

```

```
all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

编译驱动模块：make

3) 加载

加载模块：insmod test_driver.ko

查看是否注册成功：cat /proc/devices

2、建立设备文件

查看设备号：cat /proc/devices

建立设备文件：mknod /dev/test_dev c 241 0

查看是否创建成功：ls -l /dev/test_dev

3、执行应用测试

1) 编程

```
#include <sys/ioctl.h>
#include <linux/types.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <strings.h>

#include "test_cmd.h"

int main(int argc, char *argv[])
{
    int fd;
    char buf[10];
    fd = open("/dev/test_dev", O_RDWR);
    if(fd < 0) {
```



```

        perror("open failed\n");
        return -1;
    }
    ioctl(fd, TEST_ON);

    return 0;
}

```

2) 编译

```
gcc -o test_app test_app.c
```

3、运行

```
./test_app
```

dmesg 查看内核日志显示调用到内核的 arm_on 函数

【实验 7】读写设备（read/write）

1、内核驱动程序

1) 编程

```

#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>

//第一步：添加读写有关头文件
#include <asm/uaccess.h> //添加 copy_to_user 与 copy_from_user 的头文件
#define BUF_SIZE 1024 //内核 buffer 的 size
static char tmpbuf[BUF_SIZE]; //内核空间的数据读写数组

int test_open(struct inode *node, struct file *filp)
{
    printk("%s\n", __FUNCTION__);
    return 0;
}

int test_close(struct inode *node, struct file *filp)
{

```

```

    printk("%s\n", __FUNCTION__);
    return 0;
}

```

//第二步：添加设备驱动操作接口函数 write 和 read，这两个函数在应用程序调用 write 及 read 时调用。

```

static ssize_t test_chardev_read(struct file *file, char __user *buf, size_t
const count, loff_t *offset)
{
    if(count < BUF_SIZE)    //读写大小检查
    {
        if(copy_to_user(buf, tmpbuf, count))//执行完成后返回还需拷贝的字
节数。成功为 0
        {
            printk(KERN_ALERT "copy to user fail \n");
            return -EFAULT;
        }
    }else{
        printk(KERN_ALERT "read size must be less than %d\n",
BUF_SIZE);
        return -EINVAL;
    }

    *offset += count;    //文件位置指针更新
    return count;    //返回文件成功读写的字节数
}

```

```

static ssize_t test_chardev_write(struct file *file, const char __user
*buf, size_t const count, loff_t *offset)
{
    if(count < BUF_SIZE)    //读写大小检查
    {
        if(copy_from_user(tmpbuf, buf, count)) //执行完成后返回还需拷贝
的字节数。成功为 0
        {
            printk(KERN_ALERT "copy from user fail \n");
            return -EFAULT;
        }
    }else{

        printk(KERN_ALERT "size must be less than %d\n", BUF_SIZE);
    }
}

```

```

        return -EINVAL;
    }

    *offset += count;    //文件位置指针更新
    return count;        //返回文件成功读写的字节数
}

//第三步：完成 file_operations 的赋值，为该字符设备添加读写操作接口
static struct file_operations chardev_fops={
    .owner = THIS_MODULE,
    .open = test_open,
    .release = test_close,
    .read = test_chardev_read,
    .write = test_chardev_write,
};

//加载模块，注册字符设备驱动
int test_init(void)
{
    int ret;
    ret = register_chrdev(241, "test-driver", &test_fops);
    if(ret){
        printk("register failed\n");
        return -1;
    }

    return 0;
}

void test_exit(void)
{
    unregister_chrdev(241, "test-driver");
}

module_init(test_init);
module_exit(test_exit);
MODULE_LICENSE("GPL");

```

2) 编译

第一步：进入 chr_drv 实验目

第二步：在实验目录 char_drv.c 中按上述代码修改。

第三步：执行 make，编译生成的内核模块 char_drv.ko

3) 运行

第一步：加载模块：insmod char_drv.ko

第二步：通过参看系统文件来查看驱动的加载与卸载情况。

- cat /dev/devices
- 在 Character devices: 列表中查找驱动中注册的对应名字跟主设备号的设备信息。

第三步：执行测试程序看读写的返回结果。

第四步：卸载模块：rmmod char_drv.ko

2、应用测试程序

1) 编程

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <string.h>

//添加要读写的数组及字符串
static char sz[] = "this is a test string\n";
static char readback[1024];

int main(int argc, char **argv)
{
    int fd;
    fd = open("/dev/test", O_RDWR);
    if(fd>0) {
        printf("I am testing my device...\n");
        /*添加读写的测试*/
        write(fd, sz, strlen(sz));
        read(fd, readback, strlen(sz) + 1);
        printf("the string I read back is : %s\n", readback);
        close(fd);
    }
    return 0;
}
```

2) 编译

编译: `arm-linux-gcc test.c -o test`

3) 运行

下载: 使用超级终端下载到开发板, 在开发板输入 `rx test` 然后选择超级终端的文件发送功能, 使用 `xmodem` 协议。

执行: 执行成功可以看到 `the string I read back is : this is a test string`