

电子科技大学

实验报告一

学生姓名：董文龙 学 号：202222080411 指导教师：詹思瑜

实验地点：主楼 A2-412 实验时间：2023/05/08

一、实验室名称：A2-412

二、实验项目名称：实现含有输入参数的 Hello 内核模块

三、实验学时：4

四、实验目的：

- 熟悉开发环境。
- 初步掌握 Linux 环境下内核模块编写、编译、调试及运行的基本流程。
- 熟悉内核编程的基本方法和步骤。

五、实验内容与要求：

1. 内容：

- 在 Ubuntu 16.04 环境下实现含输入参数的 Hello 内核模块
- 在内核模块中实现单链表
- 链表节点结构如下：

```
Struct listnode{  
    int val_  
    struct listnode* next_  
}
```

- 链表具有插入、删除、查找、打印基本操作。

2. 要求

- 理解内核模块与应用程序的区别；
- 初步掌握 Linux 环境下内核模块编写、编译、调试及运行的基本流程；
- 掌握内核模块参数传递方法；
- 在 Ubuntu 16.04 环境下实现含输入参数的 Hello 内核模块，可以按照输入的打印行数及姓名输出相应的信息。
- 链表中的基本操作的函数原型如下：

```
int insert(int val); //成功返回 1，失败返回-1
int delete(int val); //成功返回 1，失败返回-1
int lookup(int val); //找到返回 1，没有返回-1
void print(listnode* head) //从头至尾打印出链表内容
```

六、实验步骤:

6.1 编写内核模块

1. 头文件引入

```
#include <linux/module.h> //每个模块都要包括的头文件
#include <linux/kernel.h> //用到了 printk()函数
#include <linux/init.h>
#include <linux/slab.h>

int num = 10;

char *name = "wenlong dong";
```

2. 声明节点结构

```
// 声明节点结构
typedef struct node
{
    struct node *next; // 指向直接后继元素的指针
    int elem;           // 存储整形元素
} link;
```

3. 创建链表

```
// 创建链表的函数
link *initLink(void)
{
    link *p = (link *)kmalloc(sizeof(link), GFP_KERNEL); // 创建一个头结点
    link *temp = p;                                         // 声明一个指针指向头结
```

点，用于遍历链表

```
// 生成链表
for (int i = 1; i < 5; i++)
{
    // 创建节点并初始化
    link *a = (link *)kmalloc(sizeof(link), GFP_KERNEL);
    a->elem = i;
    a->next = NULL;
    // 建立新节点与直接前驱节点的逻辑关系
    temp->next = a;
    temp = temp->next;
}
return p;
}
```

4. 插入元素

// p 为原链表，elem 表示新数据元素，add 表示新元素要插入的位置

```
link *insertElem(link *p, int elem, int add)
{
    link *temp = p; // 创建临时结点 temp
    // 首先找到要插入位置的上一个结点
    for (int i = 1; i < add; i++)
    {
        temp = temp->next;
        if (temp == NULL)
        {
            printk(KERN_WARNING "插入位置无效\n");
            return p;
        }
    }
    // 创建插入结点 c
    link *c = (link *)kmalloc(sizeof(link), GFP_KERNEL);
    c->elem = elem;
    // 向链表中插入结点
    c->next = temp->next;
    temp->next = c;
    return p;
}
```

5. 删除元素

```

// p 为原链表，add 为要删除元素的值
link *delElem(link *p, int add)
{
    link *temp = p;
    // 遍历到被删除结点的上一个结点
    for (int i = 1; i < add; i++)
    {
        temp = temp->next;
        if (temp->next == NULL)
        {
            printk(KERN_WARNING "没有该结点\n");
            return p;
        }
    }
    link *del = temp->next;          // 单独设置一个指针指向被删除结点，以防丢失
    temp->next = temp->next->next; // 删除某个结点的方法就是更改前一个结点的指针域
    kfree(del);                    // 手动释放该结点，防止内存泄漏
    return p;
}

```

6. 查找元素

```

// p 为原链表，elem 表示被查找元素、
int selectElem(link *p, int elem)
{
    // 新建一个指针 t，初始化为头指针 p
    link *t = p;
    int i = 1;
    // 由于头节点的存在，因此 while 中的判断为 t->next
    while (t->next)
    {
        t = t->next;
        if (t->elem == elem)
        {
            return i;
        }
        i++;
    }
    // 程序执行至此处，表示查找失败
    return -1;
}

```

```
}
```

7. 更新元素

```
// 更新函数，其中，add 表示更改结点在链表中的位置，newElem 为新的数据域的值
link *updateElem(link *p, int add, int newElem)
{
    link *temp = p;
    temp = temp->next; // 在遍历之前，temp 指向首元结点
    // 遍历到待更新结点
    for (int i = 1; i < add; i++)
    {
        temp = temp->next;
    }
    temp->elem = newElem;
    return p;
}
```

8. 打印链表

```
void display(link *p)
{
    link *temp = p; // 将 temp 指针重新指向头结点
    // 只要 temp 指针指向的结点的 next 不是 Null，就执行输出语句。
    while (temp->next)
    {
        temp = temp->next;
        printk(KERN_INFO "%d ", temp->elem);
    }
    printk("\n");
}
```

9. 模块初始化函数

```
// num 和 name 即为内核加载过程中输入的参数
static int hello_init(void) // static 使得该文件以外无法访问
{
    // 只能使用内核里定义好的 C 函数，printk 会根据日志级别将指定信息输出到控制台
    // 或日志文件中，
    // KERN_ALERT 会输出到控制台
    for (int i = 0; i < num; i++)
    {
        // 注意 printk 和 C 语言中 printf 的区别
    }
}
```

```

        printk(KERN_INFO "hello,%s\n", name);
    }

    // 初始化链表 (1, 2, 3, 4)
    printk(KERN_INFO "初始化链表为: \n");
    link *p = initLink();
    display(p);
    printk(KERN_INFO "在第 4 的位置插入元素 5:\n");
    p = insertElem(p, 5, 4);
    display(p);
    printk(KERN_INFO "删除元素 3:\n");
    p = delElem(p, 3);
    display(p);
    printk(KERN_INFO "查找元素 2 的位置为:\n");
    int address = selectElem(p, 2);
    if (address == -1)
    {
        printk(KERN_INFO "没有该元素");
    }
    else
    {
        printk(KERN_INFO "元素 2 的位置为:%d\n", address);
    }
    printk(KERN_INFO "更改第 3 的位置上的数据为 7:\n");
    p = updateElem(p, 3, 7);
    display(p);

    return 0;
}

```

10. 模块退出函数

```

static void hello_exit(void)
{
    printk(KERN_NOTICE "Goodbye\n");
}

```

11. 其他

```

module_init(hello_init);
module_exit(hello_exit);

```

```

module_param(num, int, S_IRUGO);
module_param(name, charp, S_IRUGO);

MODULE_LICENSE("GPL"); // 没有指定 license 会出现 error
MODULE_AUTHOR("wenlong dong");
MODULE_DESCRIPTION("Hello Kernel");

```

6.2 编译与执行

1. 编写 Makefile

```

obj-m := hello.o
CFLAGS_hello.o := -std=gnu11 -Wno-declaration-after-statement
# 指定内核源码
KERNEL_DIR := /lib/modules/$(shell uname -r)/build
# 指向当前目录
PWD := $(shell pwd)
all:
    make -C $(KERNEL_DIR) M=$(PWD) modules
clean:
    make -C $(KERNEL_DIR) M=$(PWD) clean

```

2. 根据 Makefile 编译源代码，得到 hello.ko。然后进行如下操作：插入内核模块、查看系统日志、将内核模块移出内核。

```

sudo insmod hello.ko num=2 name=liyi
sudo rmmod hello

```

6.3 分析实验结果

1. 实验结果主要体现在系统日志中，截图如下：

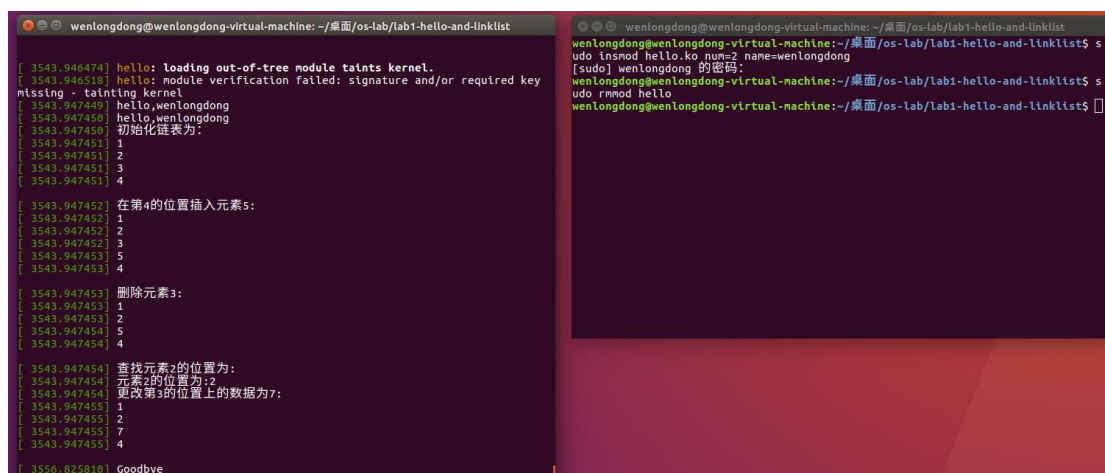


图 1 内核模块加载与卸载

2. 从内核日志中，我们可以看到，内核模块根据传入参数，打印了对应信息，

然后初始化单链表并进行了一些简单操作，在每次改变链表后都从头到尾打印了链表，打印结果符合预期。

3. 后来将内核模块移除，仍然符合实验结果。

七、总结及心得体会：

1. 在开始内核模块编程之前，我花时间熟悉了 Linux 环境下的开发工具和常用命令。这帮助我提高了工作效率，并能快速定位和解决问题。
2. 通过实验，我学会了编写简单的内核模块，包括初始化、清理和处理系统调用等。这提升了我对内核工作原理和编程概念的理解。
3. 由于内核模块运行在内核空间，错误可能导致系统崩溃或异常。我学会了使用调试工具来定位和解决问题，如内核调试器和日志输出。通过不断调试，我能快速找到问题并修复。
4. 内核编程需要耐心和细致。
5. 通过这次实验，我不仅掌握了内核编程的基础知识和技巧，还提升了问题解决和代码调试能力。内核编程是庞大而复杂的领域，我将继续深入学习和探索，并将所学应用到实际项目中。

八、对本实验过程及方法、手段的改进建议：

无

报告评分：

指导教师签字：

电子科技大学

实验报告二

学生姓名：董文龙 学 号：202222080411 指导教师：詹思瑜

实验地点：主楼 A2-412

实验时间：2023/05/15

一、实验室名称：A2-412

二、实验项目名称：实现含有输入参数的 Hello 内核模块

三、实验学时：4

四、实验目的：

- 学习 linux 内核编译和配置
- 深入理解 linux 系统调用
- 理解系统模式和用户模式的区别
- 掌握内核模块的编写方法，以扩展操作系统的功能

五、实验内容：

- 向 linux 内核增加新的系统调用，实现整数四则运算的系统功能
- 与用户空间等量的整数四则运算进行开销对比，分析测试结果
- 四则函数原型：long my_oper(int* result, int num1, int num2, char* op)
- 系统函数原型：long my_func(int count)，调用 my_oper 实现四则运算

六、实验步骤：

6.1 下载需要的内核

地址：<https://mirror.tuna.tsinghua.edu.cn/kernel>， 这里我们选择 4.10 版本，下载下来之后传到虚拟机中。

6.2 将下载的内核代码解压：

```
tar -xf linux-4.10
```

6.3 安装基本的工具和相关软件

```
sudo apt-get install libncurses*  
sudo apt-get install libssl-dev  
sudo apt-get install libelf-dev
```

6.4 新增系统调用号

修改内核目录 unistd_64.h 文件，路径：arch/sh/include/uapi/asm/unistd_64.h

添加新的系统调用号：#define __NR_my_func 400

6.5 修改系统调用向量表

修改文件 syscall_64.tbl，路径：arch/x86/entry/syscalls/syscall_64.tbl。添加 400 64

my_oper sys_my_func

6.6 修改系统调用向量表

在内核目录下修改 include/linux/syscalls.h，添加 asmlinkage long sys_my_func(int count);

6.7 添加系统调用的实现

在/linux-4.10.0/kernel/sys.c，添加具体的实现。

```
long my_oper(int* result, int num1, int num2, char* op)  
{  
    if(op)  
    {  
        if(*op == '+')  
            *result = num1 + num2;  
        else if(*op == '-')  
            *result = num1 - num2;  
        else if(*op == '*')  
            *result = num1 * num2;  
        else if(*op == '/')  
        {  
            if(num2 != 0)  
                *result = num1 / num2;  
            else  
                printk("divided number can't be zero.\n");  
        }  
    }  
}
```

```

    }
    else
        printk("operator is empth.\n");
    return 0;
}

```

SYSCALL_DEFINE1(my_func,int,count)

```

{
    printk("count is:%d\n", count);
    struct timeval tstart,tend;
    do_gettimeofday(&tstart);
    int i;
    int result;
    int times = count/4;
    for(i=0;i<times;++i)
    {
        char op_add = '+';
        my_oper(&result,10,10,&op_add);
    }
    printk("my_func op_add is ok. op_add count is:%d\n",i);
    for(i=0;i<times;++i)
    {
        char op_sub = '-';
        my_oper(&result,20,10,&op_sub);
    }
    printk("my_func op_sub is ok. op_sub count is:%d\n",i);
    for(i=0;i<times;++i)
    {
        char op_mul = '*';
        my_oper(&result,10,10,&op_mul);
    }
    printk("my_func op_mul is ok. op_mul count is:%d\n",i);
    for(i=0;i<times;++i)
    {
        char op_div = '\\';
        my_oper(&result,20,10,&op_div);
    }
}

```

```

    printk("my_func op_div is ok. op_div count is:%d\n",i);
    do_gettimeofday(&tend);
    printk("my_func running time is %ld usec\n", 1000000*(tend.tv_sec -
tstart.tv_sec)+(tend.tv_usec - tstart.tv_usec));
    return 0;
}

```

在此系统调用中，实现了加减乘除四则运算各一亿次，并计时后打印。

6.8 编译、运行内核源码

清除编译记录和临时文件： `sudo make mrproper`

配置内核： `sudo make menuconfig`，如下图 1 所示

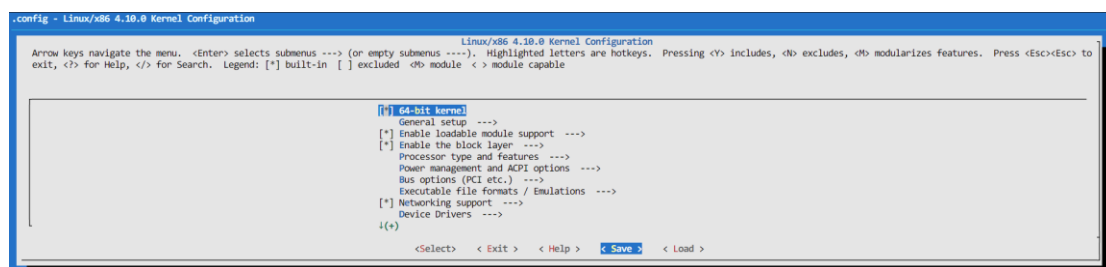


图 1 配置内核文件

save→exit

编译内核烧写镜像： `sudo make bzImage`。编译成功，如下图 2 所示

```

CC      arch/x86/boot/compressed/eboot.o
AS      arch/x86/boot/compressed/efi_stub_64.o
AS      arch/x86/boot/compressed/efi_thunk_64.o
DATAEL arch/x86/boot/compressed/vmlinux
LD      arch/x86/boot/compressed/vmlinux
ZOFFSET arch/x86/boot/zoffset.h
AS      arch/x86/boot/header.o
CC      arch/x86/boot/main.o
CC      arch/x86/boot/memory.o
CC      arch/x86/boot/pm.o
AS      arch/x86/boot/pmjump.o
CC      arch/x86/boot/printf.o
CC      arch/x86/boot/regs.o
CC      arch/x86/boot/string.o
CC      arch/x86/boot/tty.o
CC      arch/x86/boot/video.o
CC      arch/x86/boot/video-mode.o
CC      arch/x86/boot/version.o
CC      arch/x86/boot/video-vga.o
CC      arch/x86/boot/video-vesa.o
CC      arch/x86/boot/video-bios.o
LD      arch/x86/boot/setup.elf
OBJCOPY arch/x86/boot/setup.bin
OBJCOPY arch/x86/boot/vmlinux.bin
HOSTCC  arch/x86/boot/tools/build
BUILD   arch/x86/boot/bzImage
Setup is 17436 bytes (padded to 17920 bytes).
System is 7245 kB
CRC 7b602030
Kernel: arch/x86/boot/bzImage is ready (#1)

```

图 2 编译内核成功

编译模块: `sudo make modules`。编译成功，如下图 3 所示

```

IHEX      firmware/tigon/tg3_tso.bin
IHEX      firmware/tigon/tg3_tso5.bin
IHEX      firmware/3com/typhoon.bin
IHEX2FW   firmware/emi26/loader.fw
IHEX2FW   firmware/emi26/firmware.fw
IHEX2FW   firmware/emi26/bitstream.fw
IHEX2FW   firmware/emi62/loader.fw
IHEX2FW   firmware/emi62/bitstream.fw
IHEX2FW   firmware/emi62/spdif.fw
IHEX2FW   firmware/emi62/midi.fw
IHEX      firmware/kaweth/new_code.bin
IHEX      firmware/kaweth/trigger_code.bin
IHEX      firmware/kaweth/new_code_fix.bin
IHEX      firmware/kaweth/trigger_code_fix.bin
IHEX      firmware/ti_3410.fw
IHEX      firmware/ti_5052.fw
IHEX      firmware/mts_cdma.fw
IHEX      firmware/mts_gsm.fw
IHEX      firmware/mts_edge.fw
H16TOFW   firmware/edgeport/boot.fw
H16TOFW   firmware/edgeport/boot2.fw
H16TOFW   firmware/edgeport/down.fw
H16TOFW   firmware/edgeport/down2.fw
IHEX      firmware/edgeport/down3.bin
IHEX2FW   firmware/whiteheat_loader.fw
IHEX2FW   firmware/whiteheat.fw
IHEX2FW   firmware/keyspan_pda/keyspan_pda.fw
IHEX2FW   firmware/keyspan_pda/xircom_pgs.fw
IHEX      firmware/cpia2/stv0672_vp4.bin
IHEX      firmware/yam/1200.bin
IHEX      firmware/yam/9600.bin

```

图 3 编译模块成功

安装模块：sudo make modules_install，安装模块如下图 4 所示。

```

INSTALL /lib/firmware/korg/k1212.dsp
INSTALL /lib/firmware/ess/maestro3_assp_kernel.fw
INSTALL /lib/firmware/ess/maestro3_assp_minisrc.fw
INSTALL /lib/firmware/yamaha/ds1_ctrl.fw
INSTALL /lib/firmware/yamaha/ds1_dsp.fw
INSTALL /lib/firmware/yamaha/ds1e_ctrl.fw
INSTALL /lib/firmware/tehuti/bdx.bin
INSTALL /lib/firmware/tigon/tg3.bin
INSTALL /lib/firmware/tigon/tg3_tso.bin
INSTALL /lib/firmware/tigon/tg3_tso5.bin
INSTALL /lib/firmware/3com/typhoon.bin
INSTALL /lib/firmware/emi26/loader.fw
INSTALL /lib/firmware/emi26/firmware.fw
INSTALL /lib/firmware/emi26/bitstream.fw
INSTALL /lib/firmware/emi62/loader.fw
INSTALL /lib/firmware/emi62/bitstream.fw
INSTALL /lib/firmware/emi62/spdif.fw
INSTALL /lib/firmware/emi62/midi.fw
INSTALL /lib/firmware/kaweth/new_code.bin
INSTALL /lib/firmware/kaweth/trigger_code.bin
INSTALL /lib/firmware/kaweth/new_code_fix.bin
INSTALL /lib/firmware/kaweth/trigger_code_fix.bin
INSTALL /lib/firmware/ti_3410.fw
INSTALL /lib/firmware/ti_5052.fw
INSTALL /lib/firmware/mts_cdma.fw
INSTALL /lib/firmware/mts_gsm.fw
INSTALL /lib/firmware/mts_edge.fw
INSTALL /lib/firmware/edgeport/boot.fw
INSTALL /lib/firmware/edgeport/boot2.fw
INSTALL /lib/firmware/edgeport/down.fw
INSTALL /lib/firmware/edgeport/down2.fw
INSTALL /lib/firmware/edgeport/down3.bin
INSTALL /lib/firmware/whiteheat_loader.fw
INSTALL /lib/firmware/whiteheat.fw
INSTALL /lib/firmware/keyspan_pda/keyspan_pda.fw
INSTALL /lib/firmware/keyspan_pda/xircom_pgs.fw
INSTALL /lib/firmware/cpia2/stv0672_vp4.bin
INSTALL /lib/firmware/yam/1200.bin
INSTALL /lib/firmware/yam/9600.bin
DEPMOD 4.10.0

```

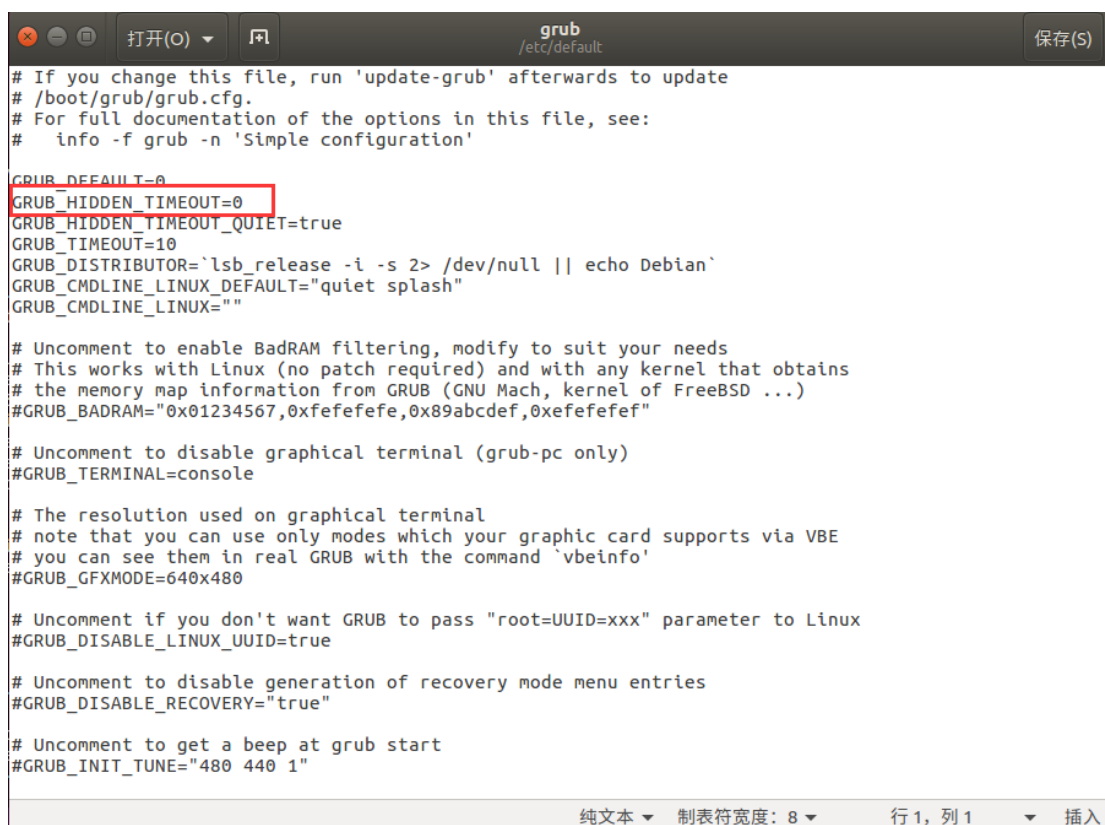
图 4 安装模块成功

安装内核：sudo make install，安装成功如下图 5 所示：

```
wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab2-new-system-call/linux-4.10$ sudo make install
sh ./arch/x86/boot/install.sh 4.10.0 arch/x86/boot/bzImage \
    System.map "/boot"
run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 4.10.0 /boot/vmlinuz-4.10.0
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 4.10.0 /boot/vmlinuz-4.10.0
update-initramfs: Generating /boot/initrd.img-4.10.0
run-parts: executing /etc/kernel/postinst.d/pm-utils 4.10.0 /boot/vmlinuz-4.10.0
run-parts: executing /etc/kernel/postinst.d/unattended-upgrades 4.10.0 /boot/vmlinuz-4.10.0
run-parts: executing /etc/kernel/postinst.d/update-notifier 4.10.0 /boot/vmlinuz-4.10.0
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 4.10.0 /boot/vmlinuz-4.10.0
Generating grub configuration file ...
Warning: Setting GRUB_TIMEOUT to a non-zero value when GRUB_HIDDEN_TIMEOUT is set is no longer supported.
Found linux image: /boot/vmlinuz-4.15.0-112-generic
Found initrd image: /boot/initrd.img-4.15.0-112-generic
Found linux image: /boot/vmlinuz-4.10.0
Found initrd image: /boot/initrd.img-4.10.0
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
```

图 5 安装内核成功

修改引导，gedit /etc/default/grub 如下图 6 所示



```
grub
/etc/default

# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
GRUB_HIDDEN_TIMEOUT=0
GRUB_HIDDEN_TIMEOUT_QUIET=true
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
GRUB_CMDLINE_LINUX=""

# Uncomment to enable BadRAM filtering, modify to suit your needs
# This works with Linux (no patch required) and with any kernel that obtains
# the memory map information from GRUB (GNU Mach, kernel of FreeBSD ...)
#GRUB_BADRAM="0x01234567,0xfefefefe,0x89abcdef,0xefefefef"

# Uncomment to disable graphical terminal (grub-pc only)
#GRUB_TERMINAL=console

# The resolution used on graphical terminal
# note that you can use only modes which your graphic card supports via VBE
# you can see them in real GRUB with the command `vbeinfo'
#GRUB_GFXMODE=640x480

# Uncomment if you don't want GRUB to pass "root=UUID=xxx" parameter to Linux
#GRUB_DISABLE_LINUX_UUID=true

# Uncomment to disable generation of recovery mode menu entries
#GRUB_DISABLE_RECOVERY="true"

# Uncomment to get a beep at grub start
#GRUB_INIT_TUNE="480 440 1"
```

图 6 修改引导

使引导生效：sudo update-grub，重启之后，选择高级选项，选择重新编译的内核，如下图 7 所示。



图 7 选择 4.10 内核登录

6.9 编写用户态代码

```
#include <sys/syscall.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>

long my_oper(int* result, int num1, int num2, char* op)
{
    if(op)
    {
        if(*op == '+')
            *result = num1 + num2;
        else if(*op == '-')
            *result = num1 - num2;
        else if(*op == '*')
            *result = num1 * num2;
        else if(*op == '\\')
        {
```

```

        if(num2 != 0)
            *result = num1 / num2;
        else
            printf("divided number can't be zero.\n");
    }
}
else
    printf("operator is empth.\n");

return 0;
}

```

```

long optime(int count)

```

```

{
    printf("count is:%d\n", count);
    struct timeval tstart,tend;
    gettimeofday(&tstart, NULL);
    int i;
    int result;
    int times = count/4;
    for(i=0;i<times;++i)
    {
        char op_add = '+';
        my_oper(&result,10,10,&op_add);
    }
    printf("my_func op_add is ok. op_add count is:%d\n",i);

    for(i=0;i<times;++i)
    {
        char op_sub = '-';
        my_oper(&result,20,10,&op_sub);
    }
    printf("my_func op_sub is ok. op_sub count is:%d\n",i);

    for(i=0;i<times;++i)
    {
        char op_mul = '*';
        my_oper(&result,10,10,&op_mul);
    }
}

```



```

    }

    printf("my_func op_mul is ok. op_mul count is:%d\n",i);
    for(i=0;i<times;++i)
    {
        char op_div = '\\';
        my_oper(&result,20,10,&op_div);
    }
    printf("my_func op_div is ok. op_div count is:%d\n",i);
    gettimeofday(&tend, NULL);
    long exeTime = (tend.tv_sec-tstart.tv_sec)*1000000+(tend.tv_usec-tstart.tv_usec);
    printf("my_func running time is %ld usec\n", exeTime);
    return 0;
}

int main()
{
    int count = 10000*10000;
    long ret = syscall(400,count);
    printf("result is %ld\n",ret);
    optime(count);
    return 0;
}

```

6.10 分析实验结果

首先我们不对编译器进行优化，执行结果如下图 8 所示

```

wenlongdong@wenlongdong-virtual-machine: ~/桌面/os-lab/lab2-new-system-call
wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab2-new-system-call$ dmesg
4896.111174] count is:100000000
4896.111175] my_func op_add is ok. op_add count is:25000000
4896.111175] my_func op_sub is ok. op_sub count is:25000000
4896.111176] my_func op_mul is ok. op_mul count is:25000000
4896.111176] my_func op_div is ok. op_div count is:25000000
4896.111177] my_func running time is 2 usec

wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab2-new-system-call$ gcc
[sudo] wenlongdong 的密码:
wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab2-new-system-call$ gcc
time_compare.c
wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab2-new-system-call$ ./a.out
result is 0
count is:100000000
my_func op_add is ok. op_add count is:25000000
my_func op_sub is ok. op_sub count is:25000000
my_func op_mul is ok. op_mul count is:25000000
my_func op_div is ok. op_div count is:25000000
my_func running time is 275790 usec
wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab2-new-system-call$

```

图 8 未经编译优化的运行时间

从上述两个实验数据可以明显看出，测试文件输出了在用户态执行加减乘除各一亿次的时间，大概为 275790 微秒。而内核系统调用执行加减乘除各一亿次为 2 微秒。

虽然，内核态执行速度快于用户态，但不应该有如此大的速度差距。每则运算

都是重复进行相同数据的运算，重复一亿次，这样的 for 循环很容易被编译器优化掉。因此，重新编译测试代码，提高编译器优化等级。得到如下测试结果，如下图 9 所示。

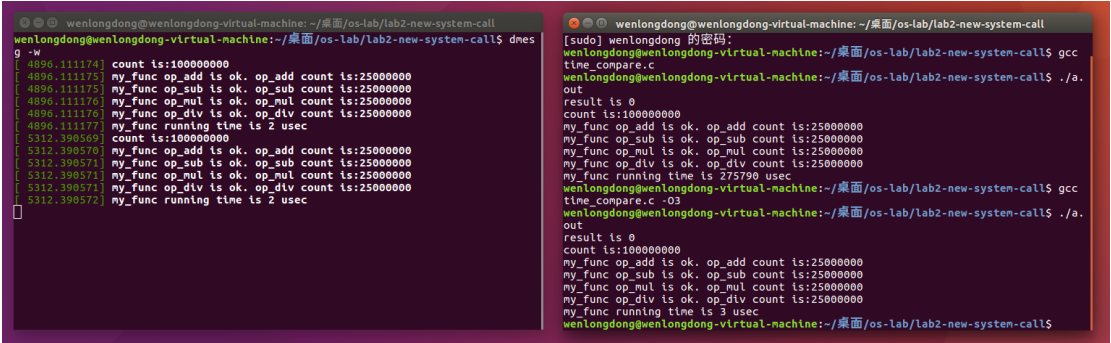


图 8 经过编译优化的运行时间

提高优化等级后，即使在用户态执行代码也能优化掉无意义的重复运算，大大缩短了执行时间。使得用户态和内核态执行时间达到非常接近的时间。

七、总结及心得体会：

- 1. 通过学习系统调用的原理和使用方式，了解了用户程序与内核之间的交互方式，以及如何通过系统调用实现对内核功能的访问。这使我能够更好地利用和扩展 Linux 操作系统的功能。
- 2. 了解到了编译器优化的强大之处。

八、对本实验过程及方法、手段的改进建议：

无

报告评分：

指导教师签字：

电子科技大学

实验报告三

学生姓名：董文龙 学 号：202222080411 指导教师：詹思瑜

实验地点：主楼 A2-412 实验时间：2023/05/22

一、实验室名称：A2-412

二、实验项目名称：动态模块及模块间通信

三、实验学时：4

四、实验目的：

- 掌握 Linux 内核添加动态模块加载执行原理
- 掌握 Linux 内核动态模块之间相互通信

五、实验内容与要求：

- 编写一个完成整数四则运算的动态模块
- 使用 insmod 和 rmmod 命令插入、删除模块
- 在模块初始化函数内，定义全局函数指针变量，令其指向模块运算函数，最后导出全局指针变量到全局符号表
- 新增一个动态模块，同实验 2 中的系统功能调用具有相同的接口。该动态模块调用被导出的函数指针，间接调用动态模块中的四则运算函数，完成整数四则运算
- 进行等量四则运算，测试，同实验二的测试结果对比

六、实验步骤：

6.1 编写整数四则运算动态模块

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

```

#include <linux/module.h>
#include <linux/kernel.h>
MODULE_LICENSE("Dual BSD/GPL");
int ASMD_Operation(int *result, int num1, int num2, char *op)
{
    if (op)
    {
        if (*op == '+')
        {
            *result = num1 + num2;
        }
        else if (*op == '-')
        {
            *result = num1 - num2;
        }
        else if (*op == '*')
        {
            *result = num1 * num2;
        }
        else if (*op == '/')
        {
            if (num2 != 0)
                *result = num1 / num2;
            else
                printk("divided number can't be zero!\n");
        }
        else
            printk("unrecongized operator %c\n", *op);
    }
    else
    {
        printk("operation is empty.\n");
    }
    return 0;
}
int (*fptr_Operation)(int *, int, int, char *);
static int os4_init(void)
{

```

```

    printk(KERN_ALERT "os3 init...\n");
    fptr_Operation = &ASMD_Operation;
    return 0;
}
static void os4_exit(void)
{
    printk(KERN_ALERT "os3 exit...\n");
}
EXPORT_SYMBOL(fptr_Operation);
module_init(os4_init);
module_exit(os4_exit);

```

EXPORT_SYMBOL 用于 Linux 内核编程的宏，它允许一个内核模块导出符号（通常是函数指针），以便其他内核模块可以使用。这样做的目的是实现模块之间的相互通信和功能共享。

6.2 编写整数四则运算动态模块的 Makefile

```

obj-m:=os4_module.o
CFLAGS_os4_module.o := -std=gnu11 -Wno-declaration-after-statement

CURRENT_PATH :=$(shell pwd)
VERSION_NUM :=$(shell uname -r)
LINUX_PATH :=/usr/src/linux-headers-$(VERSION_NUM)
all :
    make -C $(LINUX_PATH) M=$(CURRENT_PATH) modules -Wdeclaration-after-statement
    rm *.order *.mod.c *.o *.symvers
clean :
    make -C $(LINUX_PATH) M=$(CURRENT_PATH) clean

```

6.3 编写、编译、安装新增动态模块

新增动态模块用于调用四则运算动态模块，进行四则运算，并计时后打印。

代码如下：

```

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/ktime.h>
#include <linux/delay.h>
MODULE_LICENSE("Dual BSD/GPL");

```

```

extern int (*fptr_Operation)(int *, int, int, char *);
struct timeval tstart, tend;
static int call_os4_module_init(void)
{
    printk("call_os4_module_init...\n");
    do_gettimeofday(&tstart);
    const int count = 25000000;
    int result;
    int i;
    for (i = 0; i < count; i++)
    {
        char op_add = '+';
        fptr_Operation(&result, 10, 10, &op_add);
    }
    for (i = 0; i < count; i++)
    {
        char op_sub = '-';
        fptr_Operation(&result, 20, 10, &op_sub);
    }
    for (i = 0; i < count; i++)
    {
        char op_mul = '*';
        fptr_Operation(&result, 10, 10, &op_mul);
    }
    for (i = 0; i < count; i++)
    {
        char op_div = '\\';
        fptr_Operation(&result, 20, 10, &op_div);
    }
    do_gettimeofday(&tend);
    printk("total time:%ld usec\n", 1000000 * (tend.tv_sec - tstart.tv_sec) + (tend.tv_usec -
tstart.tv_usec) / 1000);
    return 0;
}
static void call_os4_module_exit(void)
{
    printk("call_os4_module_exit...\n");
    return;
}

```

```
}  
module_init(call_os4_module_init);  
module_exit(call_os4_module_exit);
```

在此模块中，调用了四则运算模块提供的运算接口（函数指针），实现了动态模块之间的通信。

6.4 编写、编译、安装新增动态模块的 Makefile

```
obj-m:=call_os4_module_add_time.o  
CFLAGS_call_os4_module_add_time.o := -std=gnu11 -Wno-declaration-after-statement  
CURRENT_PATH :=$(shell pwd)  
VERSION_NUM :=$(shell uname -r)  
LINUX_PATH :=/usr/src/linux-headers-$(VERSION_NUM)  
all :  
    make -C $(LINUX_PATH) M=$(CURRENT_PATH) modules  
    rm *.order *.mod.c *.o *.symvers  
clean :  
    make -C $(LINUX_PATH) M=$(CURRENT_PATH) clean
```

6.5 分析实验结果

执行下面的命令，先后插入两个模块。注意，必须先插入四则运算模块，然后插入新增模块。

```
sudo insmod os4_module.ko  
cd call_os4_module/  
sudo insmod call_os4_module_add_time.ko  
sudo rmmod call_os4_module_add_time.ko  
sudo rmmod os4_module.ko
```

得到实现结果：

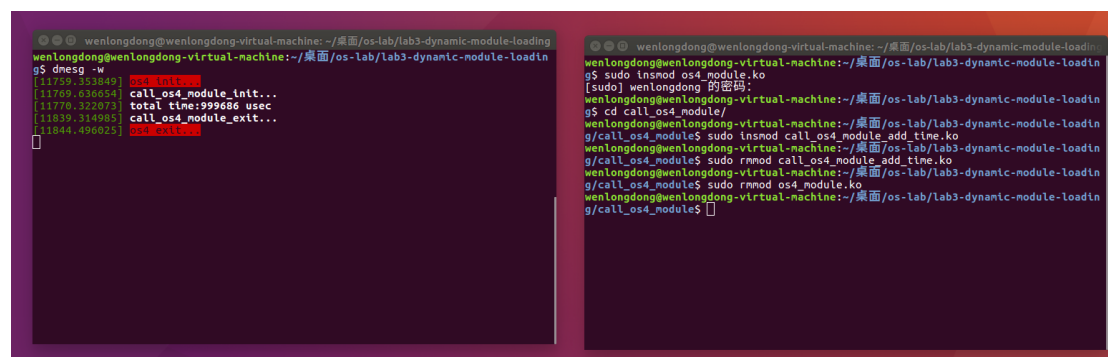


图 1 动态模块的加载与卸载

通过观察系统日志，我们发现在新增的模块中，通过调用四则运算模块的函数指针进行一亿次整数四则运算，耗时约为 999675 微秒。

与实验二的用户态执行时间进行对比后，发现动态模块的执行速度反而更慢了（用户态耗时约为 275790 秒）。

然而，在动态模块的执行过程中，并没有进行完全优化，即执行了一亿次函数调用，而非一次性完成，无法对此处的 for 循环进行优化。

这就导致了动态模块的执行速度反而比用户态更高了。

七、总结及心得体会：

1. 通过实验，编写一个能够完成整数四则运算的动态模块。在编写过程中，了解了模块的初始化函数和清理函数的编写，以及运算函数的实现。合理的模块设计对于提高代码的可维护性和扩展性非常重要。
2. 使用 insmod 和 rmmod 命令插入和删除模块是关键步骤。通过这些命令，我可以将编写好的动态模块插入到内核中进行测试，并在不需要时将其从内核中删除。这使得模块的开发和调试更加方便和灵活。
3. 在模块初始化函数内，定义全局函数指针变量并导出到全局符号表是重要的一步。通过这样做，可以在其他模块或系统功能中使用导出的函数指针变量，间接调用动态模块中的四则运算函数。这种间接调用方式降低了模块之间的耦合度，增加了模块的可扩展性。
4. 通过进行等量四则运算的测试，能够验证动态模块的正确性和性能，并将其与实验二的测试结果进行对比。同时，也能够观察到内核执行速度与用户态执行速度的差异。这对于评估内核执行效率和动态模块的优化空间非常有意义。

八、对本实验过程及方法、手段的改进建议：

无

报告评分：

指导教师签字：

电子科技大学

实验报告四

学生姓名：董文龙 学 号：202222080411 指导教师：詹思瑜

实验地点：主楼 A2-412

实验时间：2023/05/29

一、实验室名称：A2-412

二、实验项目名称：

三、实验学时：4

四、实验目的：

- 理解 linux 设备字符驱动程序的基本原理
- 掌握 Linux 字符驱动设备驱动程序的框架结构
- 学会编写字符设备驱动程序

五、实验内容与要求：

在 linux 环境下编写实验一个基于内存的虚拟字节设备驱动程序，具备打开、关闭、读、写的功能

六、实验步骤：

6.1 编写字符驱动程序，包括设备注册，设备操作的实现，设备注销

头文件如下：

```
#pragma once
#include <linux/types.h>
#include <linux/cdev.h>
#include <linux/sched.h>
#include <linux/slab.h>
#include <asm/current.h>
```

```

#include <linux/init.h>
#include <linux/module.h>
#include <linux/device.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/uaccess.h>

#define DEVICE_NAME "charMsgDriver"
#define CLASS_NAME "charMsg"
#define MSG_COUNT 1024

// static int majorNumber;
// static char msg[MSG_COUNT] = {0}; // 用来保存从用户空间传输的数据，用于 write 函数
// static short size_of_msg;           // 记录用户空间传输的数据大小
// static int DeviceOpenNumbers = 0;
// static struct class *pcharMsgDriverClass = NULL;
// static struct device *pCharMsgDriverDevice = NULL;

// additional
#define MSG_COUNT 1024
#define BUFFER_SIZE 256
static const unsigned int MINOR_BASE = 0;
static const unsigned int MINOR_NUM = 2;
static unsigned int chardev_major;
static struct cdev chardev_cdev;
static struct class *chardev_class = NULL;

```

实现如下：

```

#include "charDevDriver.h"
#include "DriverFileOperation.h"
MODULE_LICENSE("GPL");
MODULE_AUTHOR("wenlongdong");
MODULE_DESCRIPTION("a simple linux char driver example");
/**
 * linux 操作系统中一切都是文件,设备也是文件。
 * linux/fs.h 定义 file_operations 结构体
 * 这里定义文件操作的相关联的回调函数

```

```

* 字符设备的基本功能包括打开,关闭,读取,写入
*/
struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = DriverOpen,
    .release = DriverRelease,
    .read = DriverRead,
    .write = DriverWrite,
};
static int __init charMsgDriverInit(void)
{
    int alloc_ret = 0;
    int cdev_err = 0;
    int minor;
    dev_t dev;
    printk(KERN_INFO "The charMsgDriverInit() function has been called.\n");
    alloc_ret = alloc_chrdev_region(&dev, MINOR_BASE, MINOR_NUM,
DEVICE_NAME);
    if (alloc_ret != 0)
    {
        printk(KERN_ERR "alloc_chrdev_region = %d\n", alloc_ret);
        return -1;
    }
    // Get the major number value in dev.
    chardev_major = MAJOR(dev);
    dev = MKDEV(chardev_major, MINOR_BASE);
    // initialize a cdev structure
    cdev_init(&chardev_cdev, &fops);
    chardev_cdev.owner = THIS_MODULE;
    // add a char device to the system
    cdev_err = cdev_add(&chardev_cdev, dev, MINOR_NUM);
    if (cdev_err != 0)
    {
        printk(KERN_ERR "cdev_add = %d\n", alloc_ret);
        unregister_chrdev_region(dev, MINOR_NUM);
        return -1;
    }
    chardev_class = class_create(THIS_MODULE, "chardev");

```

```

    if (IS_ERR(chardev_class))
    {
        printk(KERN_ERR "class_create\n");
        cdev_del(&chardev_cdev);
        unregister_chrdev_region(dev, MINOR_NUM);
        return -1;
    }
    for (minor = MINOR_BASE; minor < MINOR_BASE + MINOR_NUM; minor++)
    {
        device_create(chardev_class, NULL, MKDEV(chardev_major, minor), NULL,
"chardev%d", minor);
    }
    return 0;
}
static void __exit charMsgDriverExit(void)
{
    int minor;
    dev_t dev = MKDEV(chardev_major, MINOR_BASE);
    printk(KERN_INFO "The charMsgDriverExit() function has been called.\n");
    for (minor = MINOR_BASE; minor < MINOR_BASE + MINOR_NUM; minor++)
    {
        device_destroy(chardev_class, MKDEV(chardev_major, minor));
    }
    class_destroy(chardev_class);
    cdev_del(&chardev_cdev);
    unregister_chrdev_region(dev, MINOR_NUM);
}
module_init(charMsgDriverInit);
module_exit(charMsgDriverExit);

```

6.2 编写字符驱动程序文件操作

头文件如下：

```

#pragma once
#include "charDevDriver.h"
int DriverOpen(struct inode *pInodeStruct, struct file *pFileStruct);
int DriverRelease(struct inode *pInodeStruct, struct file *pFileStruct);
ssize_t DriverRead(struct file *pFileStruct, char __user *pBuffer, size_t count, loff_t *pOffset);
ssize_t DriverWrite(struct file *pFileStruct, const char __user *pBuffer, size_t count, loff_t
*pOffset);

```

实现如下：

```
#include "DriverFileOperation.h"

struct data
{
    unsigned char buffer[BUFFER_SIZE];
};

int DriverOpen(struct inode *inode, struct file *file)
{
    // DeviceOpenNumbers++;
    // printk(DEVICE_NAME ": device opened %d times.\n", DeviceOpenNumbers);
    // return 0;
    char *str = "";
    int ret;
    struct data *p = kmalloc(sizeof(struct data), GFP_KERNEL);
    printk(KERN_ALERT "The DriverOpen() function has been called.\n");
    if (p == NULL)
    {
        printk(KERN_ERR "kmalloc - Null");
        return -ENOMEM;
    }
    ret = strncpy(p->buffer, str, sizeof(p->buffer));
    if (ret > strlen(str))
    {
        printk(KERN_ERR "strncpy - too long (%d)", ret);
    }
    file->private_data = p;
    return 0;
}

int DriverRelease(struct inode *inode, struct file *file)
{
    // printk(DEVICE_NAME ": close invoked.\n");
    // return 0;
    printk(KERN_ALERT "The DriverRelease() function has been called.\n");
    if (file->private_data)
    {
        kfree(file->private_data);
        file->private_data = NULL;
    }
}
```

```

    }
    return 0;
}

ssize_t DriverRead(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    // int error_count = 0;
    // error_count = copy_to_user(pBuffer, msg, size_of_msg);
    // if(error_count==0)
    //{
    //     printk(DEVICE_NAME ": send %d characters to user.\n", size_of_msg);
    //     return (size_of_msg=0);
    // }
    // else
    //{
    //     printk(DEVICE_NAME ": failed to read from device.\n");
    //     return -EFAULT;
    // }
    struct data *p = filp->private_data;
    printk(KERN_ALERT "The DriverRead() function has been called.\n");
    if (count > BUFFER_SIZE)
    {
        count = BUFFER_SIZE;
    }
    if (copy_to_user(buf, p->buffer, count) != 0)
    {
        return -EFAULT;
    }
    return count;
}

ssize_t DriverWrite(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
    // sprintf(msg, "%s(%d letters)", pBuffer, count);
    // size_of_msg = strlen(msg);
    // printk(DEVICE_NAME " : received %d characters from the user.\n " , count);
    // return 0;

```

```

struct data *p = filp->private_data;
printk(KERN_ALERT "The DriverWrite() function has been called.\n");
// printk("Before calling the copy_from_user() function : %p, %s", p->buffer, p->buffer);
if (copy_from_user(p->buffer, buf, count) != 0)
{
    return -EFAULT;
}
// printk("After calling the copy_from_user() function : %p, %s", p->buffer, p->buffer);
return count;
}

```

6.3 编写 makefile

```

obj-m := charMsgModule.o
charMsgModule-objs := charDevDriver.o DriverFileOperation.o
CFLAGS_charMsgModule.o := -std=gnu11 -Wno-declaration-after-statement
ONFIG_MODULE_SIG=n
# 指定内核源码
KERNEL_DIR := /lib/modules/$(shell uname -r)/build
# 指向当前目录
PWD := $(shell pwd)
all:
    make -C $(KERNEL_DIR) M=$(PWD) modules
    rm *.order *.mod.c *.o *.symvers
clean:
    make -C $(KERNEL_DIR) M=$(PWD) clean

```

6.4 编写测试

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>

#define BUFFER_LENGTH 1024
static char msgBuffer[BUFFER_LENGTH];

int main()
{
    int ret, fd;
    char toSend[BUFFER_LENGTH] = {0};
}

```

```
printf("\n");

// 1. 打开设备 0
fd = open("/dev/chardev0", O_RDWR);
if (fd < 0)
{
    perror("failed to open /dev/chardev0 device.\n");
    return -1;
}

// 吸入除了换行符以外的字符
printf("type something to the kernel module:\n");
scanf("%[^n]%*c", toSend);
printf("writing message to the kernel.\n");

// 2. 向设备中写入数据
ret = write(fd, toSend, strlen(toSend));
if (ret < 0)
{
    perror("failed to write the message\n");
    return -1;
}

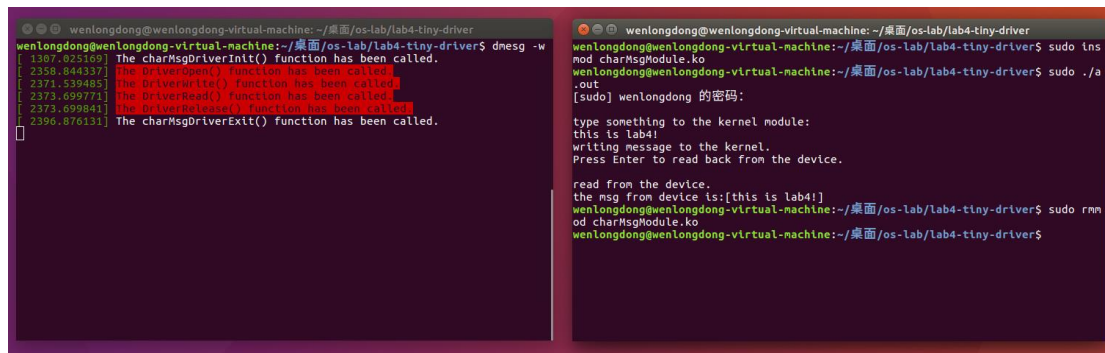
// 3. 回车从设备中读数据
printf("Press Enter to read back from the device.\n");
getchar();
printf("read from the device.\n");
ret = read(fd, msgBuffer, strlen(toSend));
if (ret < 0)
{
    perror("failed to read message from the device.\n");
    return -1;
}
printf("the msg from device is:[%s]\n", msgBuffer);
return 0;
}
```

6.5 分析实验结果

按以下操作执行


```
sudo insmod charMsgModule.ko
sudo ./a.out
sudo rmmod charMsgModule.ko
```

执行结果如下：



```
wenlongdong@wenlongdong-virtual-machine: ~/桌面/os-lab/lab4-tiny-driver
wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab4-tiny-driver$ dmesg -w
[ 1307.825169] The charMsgDriverInit() function has been called.
[ 2358.844337] The DriverOpen() function has been called.
[ 2371.539485] The DriverWrite() function has been called.
[ 2373.699771] The DriverRead() function has been called.
[ 2373.699841] The DriverRelease() function has been called.
[ 2390.876131] The charMsgDriverExit() function has been called.
^

wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab4-tiny-driver$ sudo ins
mod charMsgModule.ko
wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab4-tiny-driver$ sudo ./a
.out
[sudo] wenlongdong 的密码:
type something to the kernel module:
this is lab4!
writing message to the kernel.
Press Enter to read back from the device.

read from the device.
the msg from device is:[this is lab4!]
wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab4-tiny-driver$ sudo rm
od charMsgModule.ko
wenlongdong@wenlongdong-virtual-machine:~/桌面/os-lab/lab4-tiny-driver$
```

图 1 字符的装载与卸载

向程序输入“this is a lab4!”，程序会自动将字符写入设备。然后，按下回车，下达从字符设备读取的命令。可以看到，程序从字符设备读取到了我们写入的字符，并正确展示了出来。然后，观察系统日志，这里记录了内核动态模块的信息。

七、总结及心得体会：

1. 在编写字符设备驱动程序时，驱动程序主要由模块初始化函数、文件操作函数和清理函数组成。模块初始化函数用于注册设备，文件操作函数负责处理设备文件的读写操作，而清理函数用于释放资源。这样的框架结构使得设备驱动程序具有灵活性和可扩展性。
2. 学习和编写 Linux 字符设备驱动程序是一次有益的实践经验。通过深入理解驱动程序的基本原理和掌握驱动程序的框架结构能够更自信地进行设备驱动程序的开发，并在 Linux 系统中实现对设备的灵活控制。

八、对本实验过程及方法、手段的改进建议：

无

报告评分：

指导教师签字：