# Linux内核实验

## 实验一

### 问题分析

内核模块和Makefile文件的基本结构如下：

//内核模块.c文件基本框架

​

#include<linux/module.h> //包含了对模块的结构定义以及模块的版本控制

​

MODULE\_LICENSE("GPL"); //声明GPL版权

static \_\_init module\_init(void){ //加载模块

。。。。。。

}

​

static \_\_exit module\_exit(void){ //卸载模块

。。。。。。

}

module\_init(module\_init);

module\_exit(module\_exit);

//Makefile文件基本框架

​

ifneq ($(KERNELRELEASE),)

obj-m :=main.o //指定将要编译的内核模块列表（一些.o文件）

​

else

KERNELDIR ?=/usr/lib/modules/$(shell uname -r)/build //内核源代码位置

PWD := $(shell pwd)

default:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules //编译连接目标

endif

.PHONY:clean

clean:

-rm \*.mod.c \*.o \*.order \*.symvers \*.ko

### helloworld源代码实现

helloworld.c:

#include<linux/module.h>

​

MODULE\_LICENSE("GPL");

​

int \_\_init hello\_init(void)

{

printk("hello init\n"); //printk函数，而不是之前学的printf

printk("hello,world!\n");

return 0;

}

​

void \_\_exit hello\_exit(void)

{

printk("hello exit\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

**makefile文件**

ifneq ($(KERNELRELEASE),)

obj-m := helloworld.o

else

KERNELDIR ?=/usr/src/kernels/5.10.0-4.25.0

PWD := $(shell pwd)

default:

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

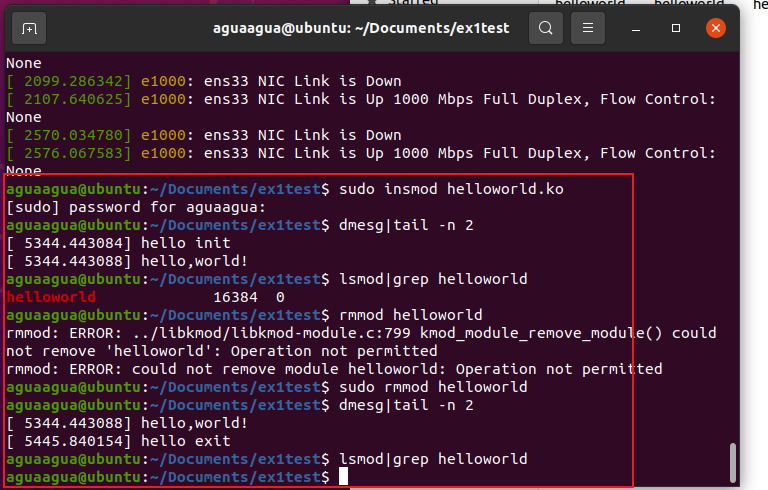
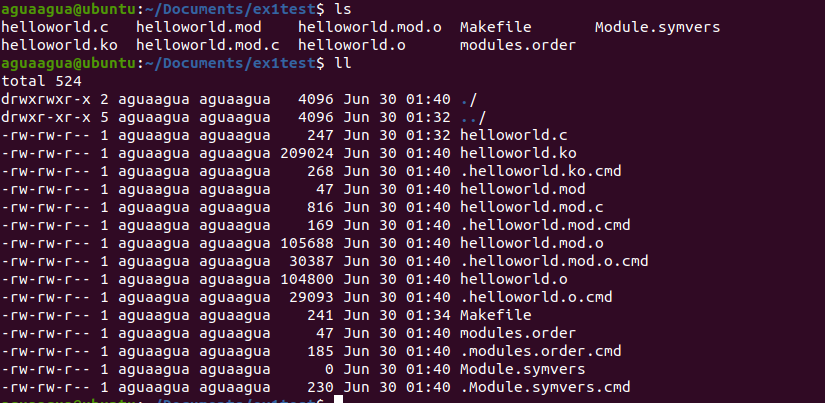
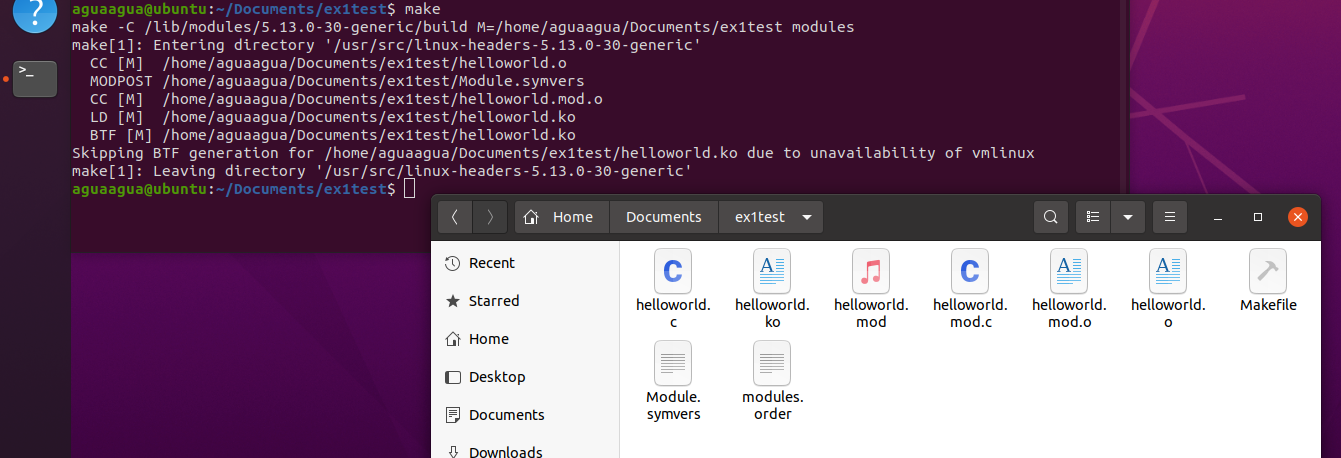
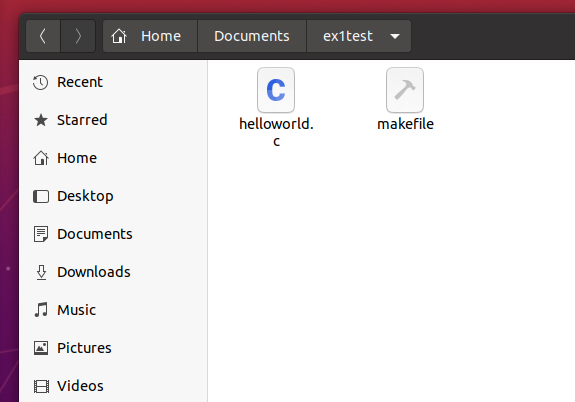
endif

.PHONY:clean

clean:

-rm \*.mod.c \*.o \*.order \*.symvers \*.ko

### 运行结果



## 实验二

### 问题分析

使用kmalloc和vmalloc分配内存， 大小分别为1k和8k， 1m，64m，使用%p打印指针的内存地址。

### kmalloc源代码实现

**Kmalloc.c源代码**

#include<linux/slab.h>

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

#define size 10

MODULE\_DESCRIPTION("memory allocation ");

MODULE\_AUTHOR("test");

MODULE\_LICENSE("GPL");

int alloc\_init(void)

{

void \*pointer;

pointer = kmalloc(1024, GFP\_KERNEL);

if(!pointer)

{ printk(KERN\_ALERT" Faild ");

//report error

}

printk(" I got : %zu Byes\n", ksize(pointer));

printk(" Address : %p\n", pointer);

kfree(pointer);

pointer = kmalloc(8192, GFP\_KERNEL);

if(!pointer)

{ printk(KERN\_ALERT" Faild ");

//report error

}

printk(" I got : %zu Byes\n", ksize(pointer));

printk(" Address : %p\n", pointer);

kfree(pointer);

return 0;

}

void alloc\_exit(void)

{

printk(KERN\_ALERT"Time to exit");

}

module\_init(alloc\_init);

module\_exit(alloc\_exit);

**makefile文件**

MOD = kmalloc

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

PWD :=$(shell pwd)

obj-m = $(MOD).o

all:

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

clean:

$(MAKE) -C $(KPATH) M=$(PWD) clean

insmod: all

sudo rmmod $(MOD).ko; true

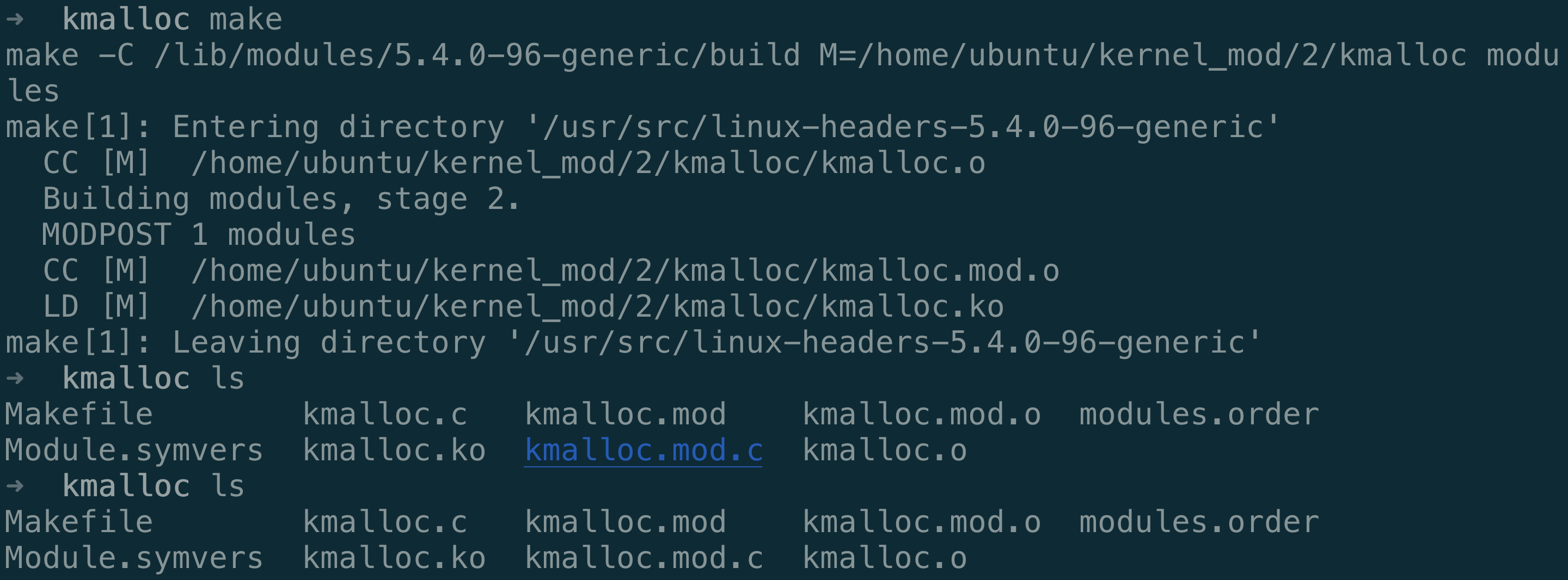
sudo insmod $(MOD).ko

log:

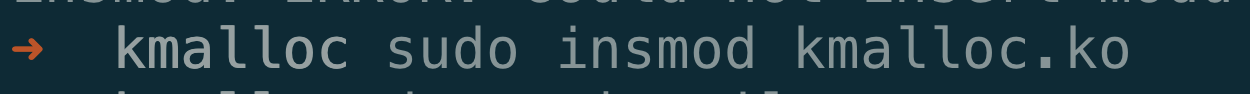
tail -f /var/log/messages

### kmalloc运行结果

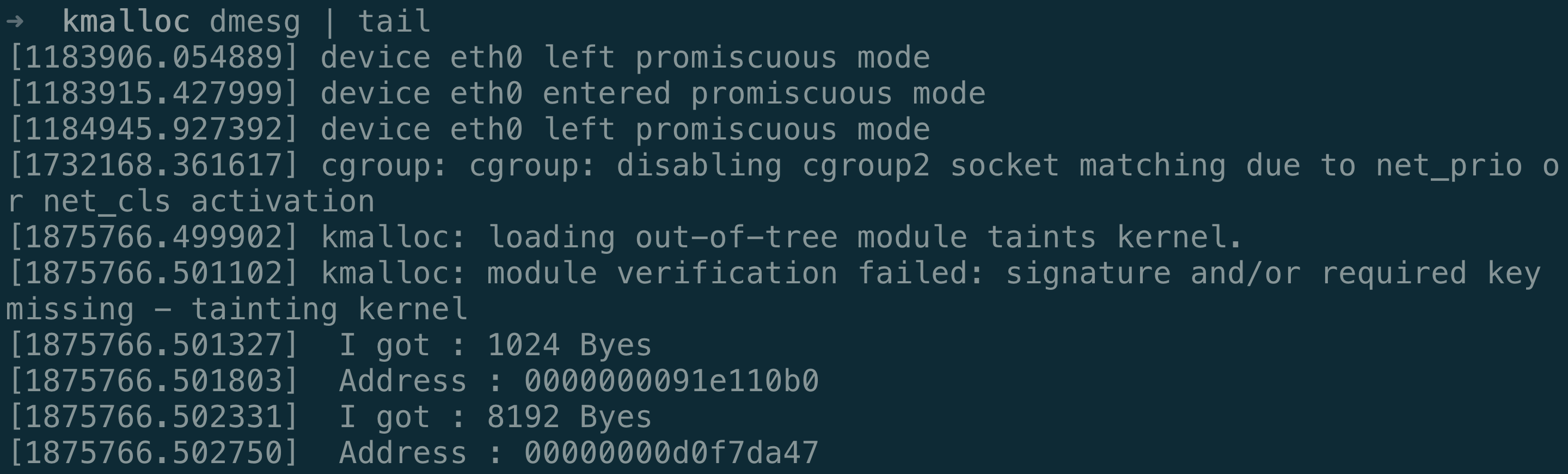
使用makefile编译内核模块



安装内核模块



dmesg显示内核信息



### vmalloc源代码实现

#include<linux/slab.h>

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

#define size 10

MODULE\_DESCRIPTION("memory allocation ");

MODULE\_AUTHOR("test");

MODULE\_LICENSE("GPL");

int alloc\_init(void)

{

void \*pointer;

// 8k

pointer = vmalloc(8192);

if(!pointer)

{ printk(KERN\_ALERT" Faild ");

//report error

}

printk(" I got : %zu Byes\n", ksize(pointer));

printk(" Address : %p\n", pointer);

// 1m

pointer = vmalloc(1048576);

if(!pointer)

{ printk(KERN\_ALERT" Faild ");

//report error

}

printk(" I got : %zu Byes\n", ksize(pointer));

printk(" Address : %p\n", pointer);

// 64m

// pointer = vmalloc(67108864);

pointer = vmalloc(6710886);

if(!pointer)

{ printk(KERN\_ALERT" Faild ");

//report error

}

printk(" I got : %zu Byes\n", ksize(pointer));

printk(" Address : %p\n", pointer);

kfree(pointer);

return 0;

}

void alloc\_exit(void)

{

printk(KERN\_ALERT"Time to exit");

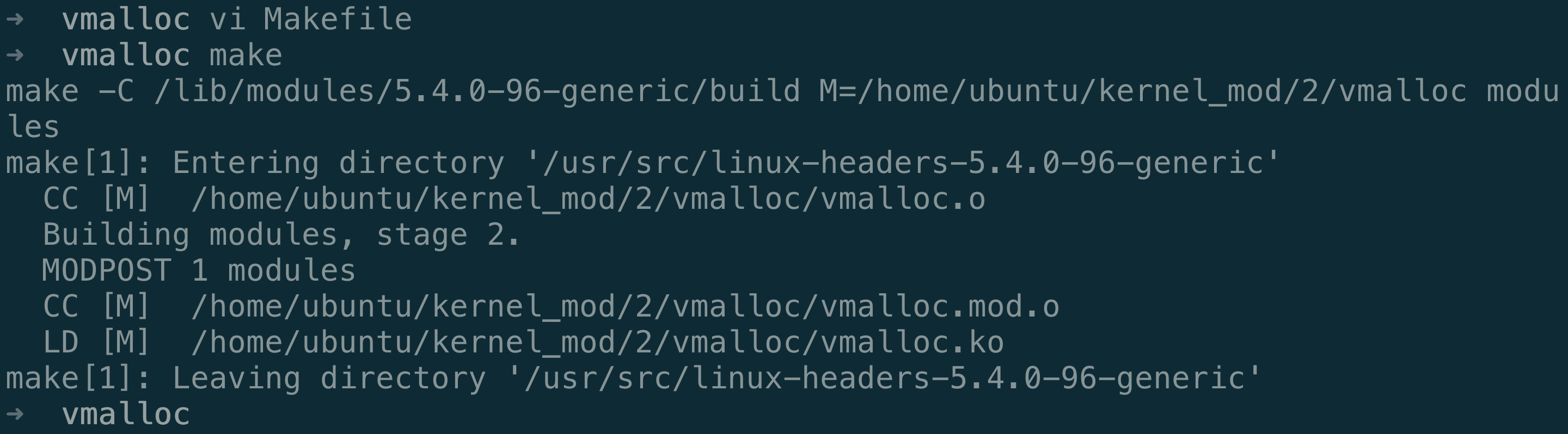
}

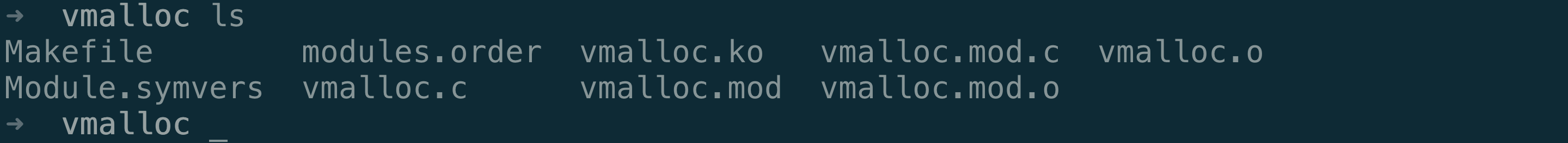
module\_init(alloc\_init);

module\_exit(alloc\_exit);

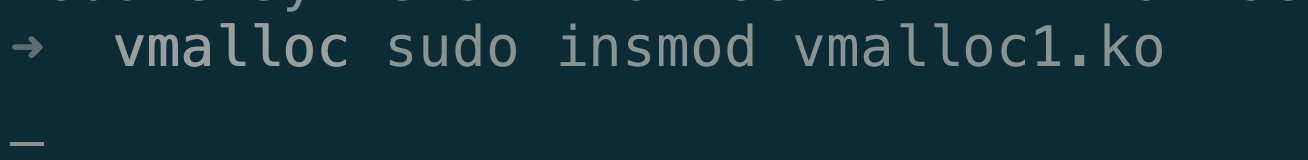
### vmalloc运行结果

使用Makefile编译模块





安装模块



## 实验三

### 源代码

**创建并运行内核线程**

**kthread.c**

#include <linux/kthread.h>

#include <linux/module.h>

#include <linux/delay.h>

​

MODULE\_LICENSE("GPL");

​

#define BUF\_SIZE 20

​

static struct task\_struct \*myThread = NULL;

​

static int print(void \*data)

{

while(!kthread\_should\_stop()){//接收kthread\_stop传递的结束线程信号，如果内核线程中未用此函数，则kthread\_stop使其结束。

printk("New kthread is running.");

msleep(2000);

}

return 0;

}

​

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

{

printk("Create kernel thread!\n");

myThread = kthread\_run(print, NULL, "new\_kthread");//创建并启动一个线程

return 0;

}

​

static void \_\_exit kthread\_exit(void)

{

printk("Kill new kthread.\n");

if(myThread)

kthread\_stop(myThread);//发送信号给myThread指向的线程，使之退出

}

​

module\_init(kthread\_init);

module\_exit(kthread\_exit);

**打印输出当前系统 CPU 负载情况**

**cpu\_loadavg.c**

#include <linux/module.h>

#include <linux/fs.h>

​

MODULE\_LICENSE("GPL");

​

char tmp\_cpu\_load[5] = {'\0'};

​

static int get\_loadavg(void)

{

struct file \*fp\_cpu;

loff\_t pos = 0;

char buf\_cpu[10];

fp\_cpu = filp\_open("/proc/loadavg", O\_RDONLY, 0);

if (IS\_ERR(fp\_cpu)){

printk("Failed to open loadavg file!\n");

return -1;

}

kernel\_read(fp\_cpu, buf\_cpu, sizeof(buf\_cpu), &pos);

strncpy(tmp\_cpu\_load, buf\_cpu, 4);

filp\_close(fp\_cpu, NULL);

return 0;

}

​

static int \_\_init cpu\_loadavg\_init(void)

{

printk("Start cpu\_loadavg!\n");

if(0 != get\_loadavg()){

printk("Failed to read loadarvg file!\n");

return -1;

}

printk("The cpu loadavg in one minute is: %s\n", tmp\_cpu\_load);

return 0;

}

​

static void \_\_exit cpu\_loadavg\_exit(void)

{

printk("Exit cpu\_loadavg!\n");

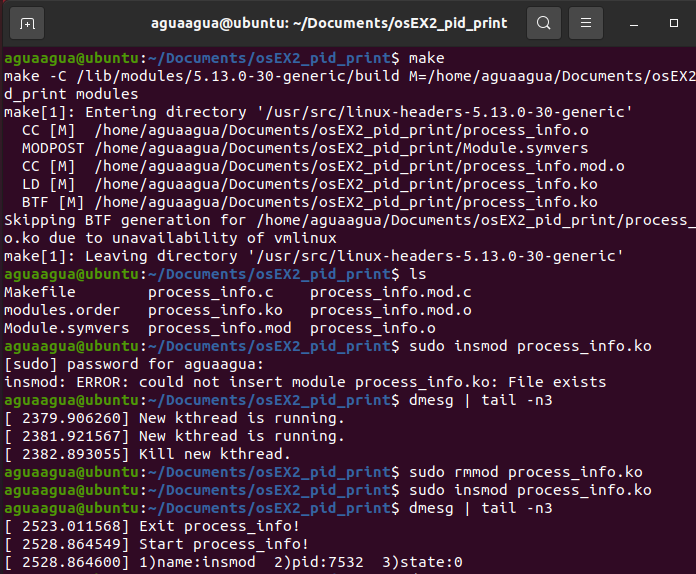
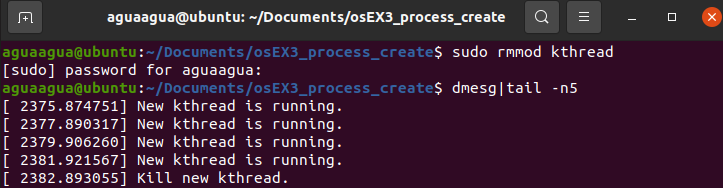
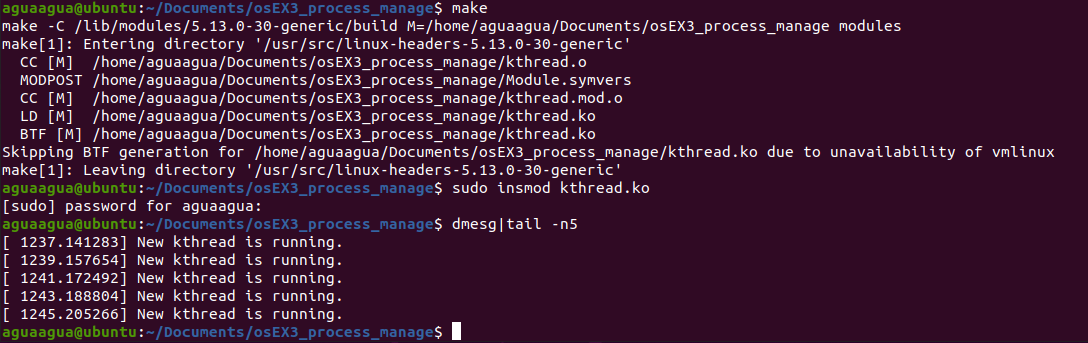
}

​

module\_init(cpu\_loadavg\_init);

module\_exit(cpu\_loadavg\_exit);

### 运行结果



## 实验四

### 源代码

**Delay.c**

#include <linux/module.h>

#include <linux/workqueue.h>

#include <linux/delay.h>

MODULE\_LICENSE("GPL");

static struct workqueue\_struct \*queue = NULL;

static struct delayed\_work mywork;

static int i = 0;

//work handle

void work\_handle(struct work\_struct \*work)

{

printk(KERN\_ALERT "Hello World!\n");

}

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

{

printk(KERN\_ALERT "Start workqueue\_test module.");

queue = create\_singlethread\_workqueue("workqueue\_test");

if(queue == NULL)

{

printk(KERN\_ALERT "Failed to create workqueue\_test!\n");

return -1;

}

INIT\_DELAYED\_WORK(&mywork, work\_handle);

for(;i <= 3; i++) {

queue\_delayed\_work(queue, &mywork, 5 \* HZ);

ssleep(15);

}

return 0;

}

static void \_\_exit timewq\_exit(void)

{

flush\_workqueue(queue);

destroy\_workqueue(queue);

printk(KERN\_ALERT "Exit workqueue\_test module.");

}

module\_init(timewq\_init);

module\_exit(timewq\_exit);

**Makefile**

MOD = delay

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

PWD :=$(shell pwd)

obj-m = $(MOD).o

all:

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

clean:

$(MAKE) -C $(KPATH) M=$(PWD) clean

insmod: all

sudo rmmod $(MOD).ko; true

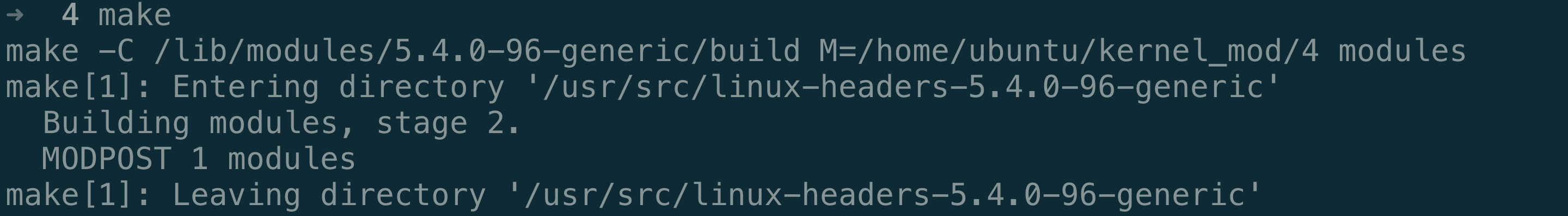
sudo insmod $(MOD).ko

log:

tail -f /var/log/messages

### 运行结果

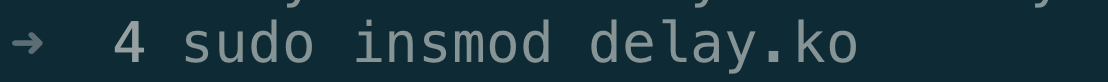
编译内核模块



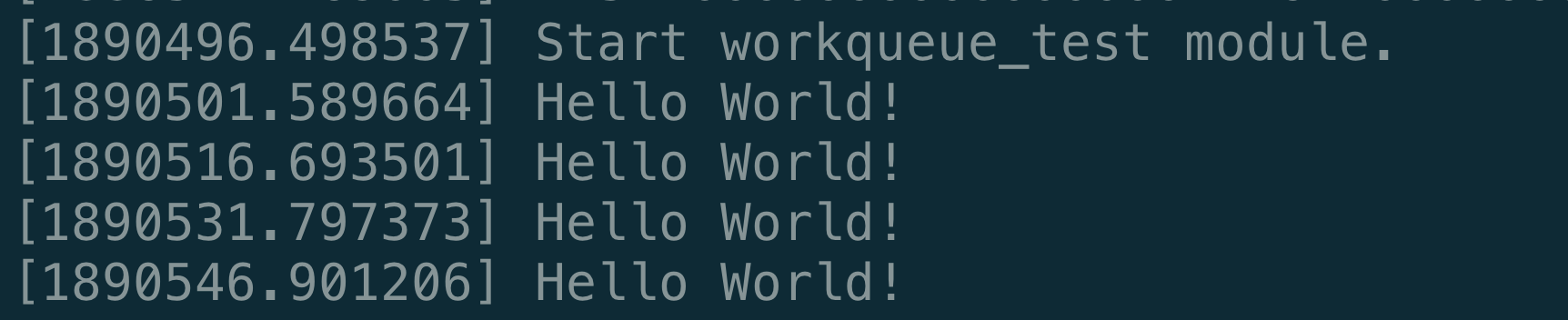
查看模块文件



安装内核模块

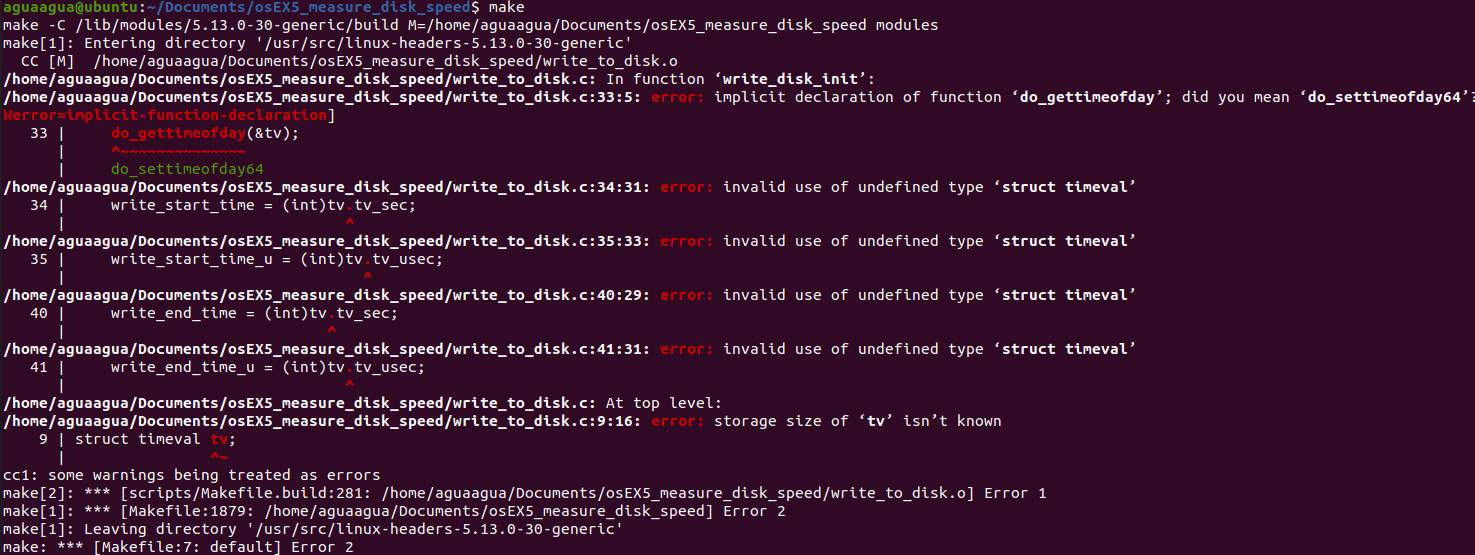


结果



## 实验五

因为版本问题，两位同学的电脑都无法正常运行





所以主要采取学习代码编思想的思路，进行代码编写

write\_to\_disk.c:

#include <linux/module.h>

#include <linux/fs.h>

#include <linux/rtc.h>

​

#define buf\_size 1024

#define write\_times 524288

​

MODULE\_LICENSE("GPL");

​

struct timeval tv;

​

static int \_\_init write\_disk\_init(void)

{

struct file \*fp\_write;

char buf[buf\_size];

int i;

int write\_start\_time;

int write\_start\_time\_u;

int write\_end\_time;

int write\_end\_time\_u;

int write\_time;

loff\_t pos;

printk("Start write\_to\_disk module...\n");

for(i = 0; i < buf\_size; i++)

{

buf[i] = i + '0';

}

fp\_write = filp\_open("/home/tmp\_file", O\_RDWR | O\_CREAT,0644);

if (IS\_ERR(fp\_write)) {

printk("Failed to open file...\n");

return -1;

}

pos = 0;

do\_gettimeofday(&tv);

write\_start\_time = (int)tv.tv\_sec;

write\_start\_time\_u = (int)tv.tv\_usec;

for(i = 0; i < write\_times; i++) {

kernel\_write(fp\_write, buf, buf\_size, &pos);

}

do\_gettimeofday(&tv);

write\_end\_time = (int)tv.tv\_sec;

write\_end\_time\_u = (int)tv.tv\_usec;

filp\_close(fp\_write, NULL);

write\_time = (write\_end\_time - write\_start\_time) \* 1000000 + (write\_end\_time\_u - write\_start\_time\_u);

printk(KERN\_ALERT "Writing to file costs %d us\n", write\_time);

printk("Writing speed is %d M/s\n", buf\_size \* write\_times / write\_time);

return 0;

}

​

static void \_\_exit write\_disk\_exit(void)

{

printk("Exit write\_to\_disk module...\n");

}

​

module\_init(write\_disk\_init);

module\_exit(write\_disk\_exit);

测试写速度

read\_from\_disk.c:

#include <linux/module.h>

#include <linux/fs.h>

#include <linux/rtc.h>

​

#define buf\_size 1024

#define read\_times 524288

​

MODULE\_LICENSE("GPL");

​

struct timeval tv;

​

static int \_\_init read\_disk\_init(void)

{

struct file \*fp\_read;

char buf[buf\_size];

int i;

int read\_start\_time;

int read\_start\_time\_u;

int read\_end\_time;

int read\_end\_time\_u;

int read\_time;

loff\_t pos;

printk("Start read\_from\_disk module...\n");

fp\_read = filp\_open("/home/tmp\_file", O\_RDONLY, 0);

if (IS\_ERR(fp\_read)) {

printk("Failed to open file...\n");

return -1;

}

​

do\_gettimeofday(&tv);

read\_start\_time = (int)tv.tv\_sec;

read\_start\_time\_u = (int)tv.tv\_usec;

pos = 0;

for(i = 0; i < read\_times; i++) {

kernel\_read(fp\_read, buf, buf\_size, &pos);

}

do\_gettimeofday(&tv);

read\_end\_time = (int)tv.tv\_sec;

read\_end\_time\_u = (int)tv.tv\_usec;

filp\_close(fp\_read, NULL);

read\_time = (read\_end\_time - read\_start\_time) \* 1000000 + (read\_end\_time\_u - read\_start\_time\_u);

printk(KERN\_ALERT "Read file costs %d us\n", read\_time);

printk("Reading speed is %d M/s\n", buf\_size \* read\_times / read\_time);

return 0;

}

​

static void \_\_exit read\_disk\_exit(void)

{

printk("Exit read\_from\_disk module...\n");

}

​

module\_init(read\_disk\_init);

module\_exit(read\_disk\_exit);

测试读速度：

## 实验六

## Register\_newfs.c源代码

**Register\_newfs.c**

#include <linux/module.h>

#include <linux/fs.h>

MODULE\_LICENSE("GPL");

static struct file\_system\_type myfs\_type =

{ .name = "myfs", .owner = THIS\_MODULE, };

MODULE\_ALIAS\_FS("myfs");

static int \_\_init register\_newfs\_init(void)

{

printk("Start register\_newfs module...");

return register\_filesystem(&myfs\_type);

}

static void \_\_exit register\_newfs\_exit(void)

{

printk("Exit register\_newfs module...");

unregister\_filesystem(&myfs\_type);

}

module\_init(register\_newfs\_init);

module\_exit(register\_newfs\_exit);

**Makefile**

**MOD = register\_newfs**

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

**PWD :=$(shell pwd)**

**obj-m = $(MOD).o**

**all:**

**$(MAKE) -C $(KPATH) M=$(PWD) modules**

**clean:**

**$(MAKE) -C $(KPATH) M=$(PWD) clean**

**insmod: all**

**sudo rmmod $(MOD).ko; true**

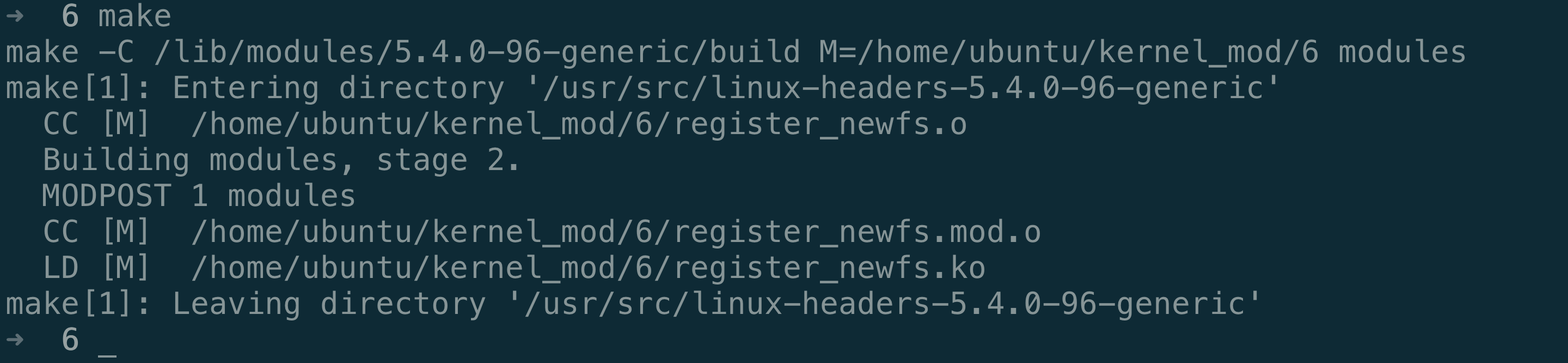
**sudo insmod $(MOD).ko**

**log:**

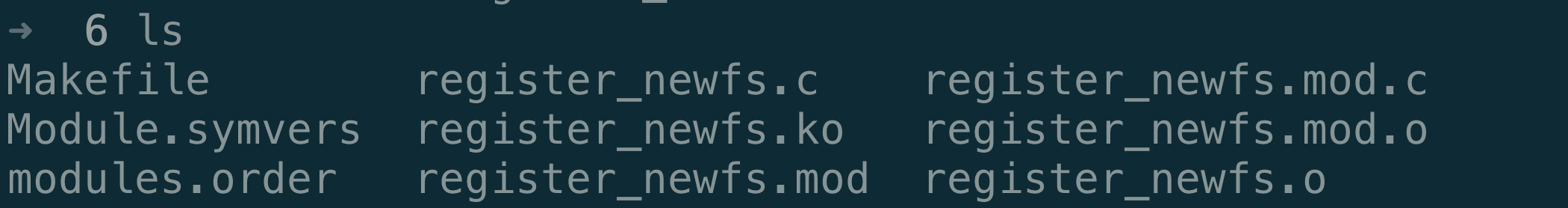
**tail -f /var/log/messages**

### Register\_newfs.c运行结果

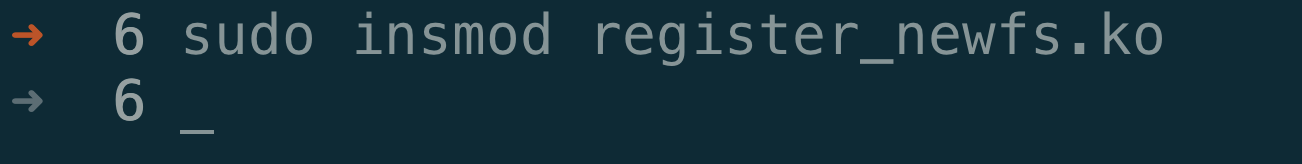
编译模块



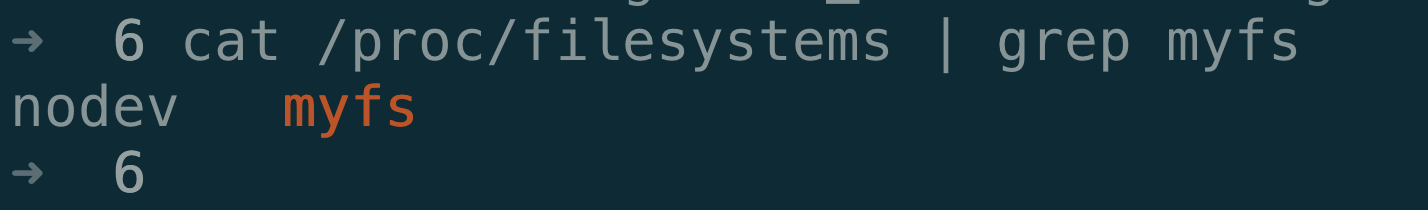
查看编译后的文件



安装模块



查看创建的文件系统



卸载模块



### Proc\_mkdir.c源代码

#include <linux/module.h>

#include <linux/proc\_fs.h>

MODULE\_LICENSE("GPL");

static struct proc\_dir\_entry \*myproc\_dir;

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

{

int ret = 0; printk("Start proc\_mkdir module...");

myproc\_dir = proc\_mkdir("myproc",NULL);

if(myproc\_dir == NULL)

return -ENOMEM;

return ret;

}

static void \_\_exit myproc\_exit(void)

{

printk("Exit proc\_mkdir module...");

proc\_remove(myproc\_dir);

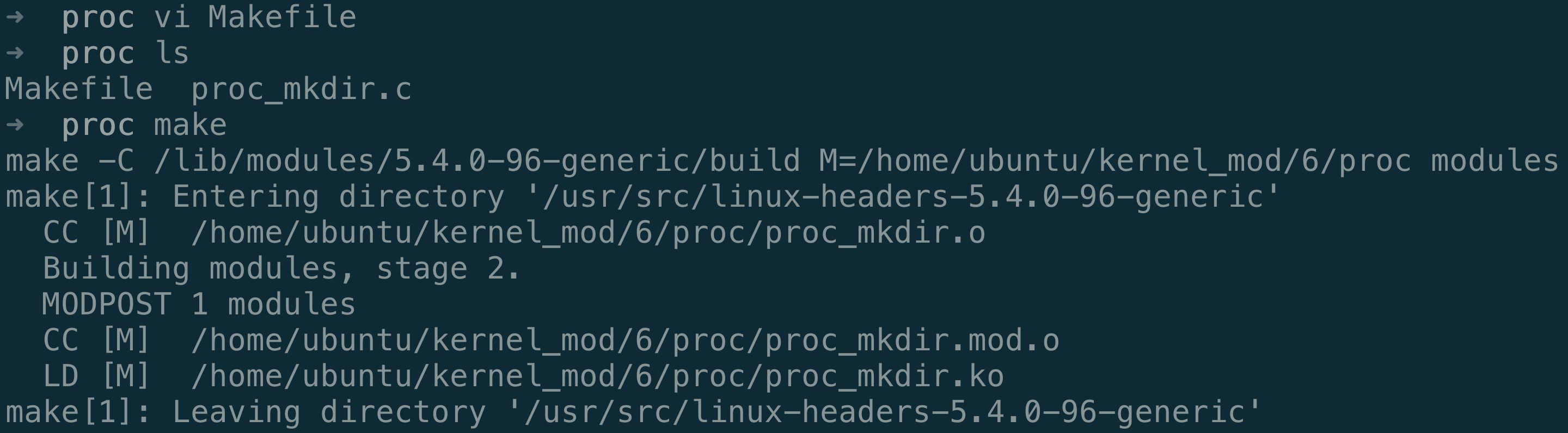
}

module\_init(myproc\_init);

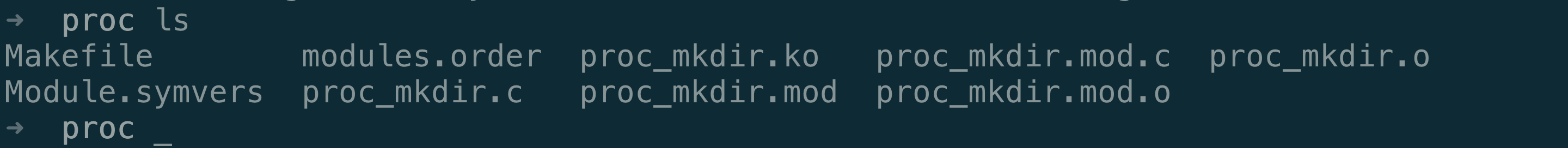
module\_exit(myproc\_exit);

### Proc\_mkdir.c运行结果

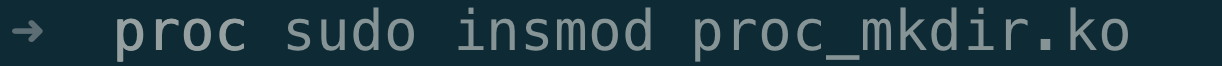
编译内核模块



列出内核模块



安装内核模块



验证结果



## 实验七

### 问题分析

编写 C 源码，基于 socket 的 UDP 发送接收程序，实现客户端与服务端的简单通信。客户端从命令行输入中读取要发送的内容，服务端接收后实时显示。

### 源代码实现

服务器端源代码如下（server.c）：

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <unistd.h>#define PORT 40000

#define BUF\_SIZE 1024

int main(void)

{

int sock\_fd;

int len;

char buffer[BUF\_SIZE];

struct sockaddr\_in server\_addr, client\_addr;

if(-1 == (sock\_fd = socket(AF\_INET, SOCK\_DGRAM, 0)) )

{

printf("Failed to create a socket!\n");

return 0;

}

//server information

memset(&server\_addr, 0, sizeof(server\_addr));

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(PORT);

server\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

if(-1 == bind(sock\_fd, (struct sockaddr\*)&server\_addr, sizeof(server\_addr)))

{

printf("Failed to bind the socket!\n");

return 0;

}

len = sizeof(client\_addr);

//rec and print

while(1)

{

bzero(buffer, BUF\_SIZE);

if(-1 != (recvfrom(sock\_fd, buffer, BUF\_SIZE, 0, (struct

sockaddr\*)&client\_addr, &len)) )

{

printf("The message received is: %s", buffer);}

}

return 0;

}

客户端源代码如下（client.c）：

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <unistd.h>

#define PORT 40000

#define BUF\_SIZE 1024

int main(void)

{

int sock\_fd;

char buffer[BUF\_SIZE];

int size;

int len;

int ret;

struct sockaddr\_in server\_addr;

if(-1 == (sock\_fd = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_IP)) ){

printf("Failed to create a socket!\n");

return 0;

}

//server infomation

memset(&server\_addr, 0, sizeof(server\_addr));

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(PORT);

server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

bzero(buffer, BUF\_SIZE);len = sizeof(server\_addr);

//read from stdin and send to server

while(1){

printf("Please enter the content to be sent:\n");

size = read(0, buffer, BUF\_SIZE);

if(size){

sendto(sock\_fd, buffer, size, 0, (struct sockaddr\*)&server\_addr, len);

bzero(buffer, BUF\_SIZE);

}

}

close(sock\_fd);

return 0;

}

### 运行结果

