实 验 报 告

课程名称： 操作系统实验

学 院： 计算机科学与工程学院

专 业： 软件工程 班 级： 软件18-1班

姓 名：KAFLE SAMRAT学 号： 201801060933

山 东 科 技 大 学

实 验 报 告

页

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 组 别 |  | 姓 名 | KAFLE SAMRAT | 同组实验者 |  |
| 实验项目  名称 | Linux驱动程序 | | 实验日期 |  | |
| 教师评语 |  | | | | |
| 实验成绩： | | | 指导教师（签名）：  年 月 日 | | |

实验目的:

在linux系统中，一个硬件设备想要运行同样需要提供设备驱动程序，底层的原理和MCU中的设备驱动程序一样：收发数据以及处理数据，只是由于桌面操作系统的特殊性，设备驱动程序的流程会复杂很多。

linux将内核与用户分离，驱动模块运行在内核空间中，而应用程序运行在用户空间，内核主要对公共且有限的资源进行管理、调度，比如硬件外设资源、内存资源等。

当用户需要使用系统资源时，通过系统调用进入内核，由内核基于某种调度算法对这部分资源进行调度。

实验要求:

通过阅读、执行driver.c及其对应的Makefile文件，理解Linux内核模块LKM的基本框架和运行方式、原理。

结合操作系统知识，通过实验深入理解计算机程序在操作系统支持下的运行方式。

实验内容:

从教材提供的电子资源中找到或者按教材提示自己编写简单的Linux内核模块driver.c及其对应的Makefile文件

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/kthread.h>

#include <linux/delay.h>

#include <linux/kobject.h>

#include <linux/sysfs.h>

#include <linux/slab.h>

#include <linux/string.h>

#include <linux/gpio.h>

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Downey");

MODULE\_DESCRIPTION("Kobject test!");

MODULE\_VERSION("0.1");

static int led\_status = 0;

#define LED\_PIN 26

static struct kobject \*kob;

static ssize\_t led\_show(struct kobject\* kobjs,struct kobj\_attribute \*attr,char \*buf)

{

printk(KERN\_INFO "Read led\n");

return sprintf(buf,"The led\_status status = %d\n",led\_status);

}

static ssize\_t led\_status\_show(struct kobject\* kobjs,struct kobj\_attribute \*attr,char \*buf)

{

printk(KERN\_INFO "led status show\n");

return sprintf(buf,"led status : \n%d\n",led\_status);

}

static ssize\_t led\_status\_store(struct kobject \*kobj, struct kobj\_attribute \*attr,const char \*buf, size\_t count)

{

printk(KERN\_INFO "led status store\n");

if(0 == memcmp(buf,"on",2))

{

gpio\_set\_value(LED\_PIN,1);

led\_status = 1;

}

else if(0 == memcmp(buf,"off",3))

{

gpio\_set\_value(LED\_PIN,0);

led\_status = 0;

}

else

{

printk(KERN\_INFO "Not support cmd\n");

}

return count;

}

static struct kobj\_attribute status\_attr = \_\_ATTR\_RO(led);

static struct kobj\_attribute led\_attr = \_\_ATTR(led\_status,0660,led\_status\_show,led\_status\_store); //Doesn't support 0666 in new version.

static struct attribute \*led\_attrs[] = {

&status\_attr.attr,

&led\_attr.attr,

NULL,

};

static struct attribute\_group attr\_g = {

.name = "kobject\_test",

.attrs = led\_attrs,

};

int create\_kobject(void)

{

kob = kobject\_create\_and\_add("obj\_test",kernel\_kobj->parent);

return 0;

}

static void gpio\_config(void)

{

if(!gpio\_is\_valid(LED\_PIN)){

printk(KERN\_ALERT "Error wrong gpio number\n");

return ;

}

gpio\_request(LED\_PIN,"led\_ctr");

gpio\_direction\_output(LED\_PIN,1);

gpio\_set\_value(LED\_PIN,1);

led\_status = 1;

}

static void gpio\_deconfig(void)

{

gpio\_free(LED\_PIN);

}

static int \_\_init sysfs\_ctrl\_init(void){

printk(KERN\_INFO "Kobject test!\n");

gpio\_config();

create\_kobject();

sysfs\_create\_group(kob, &attr\_g);

return 0;

}

static void \_\_exit sysfs\_ctrl\_exit(void){

gpio\_deconfig();

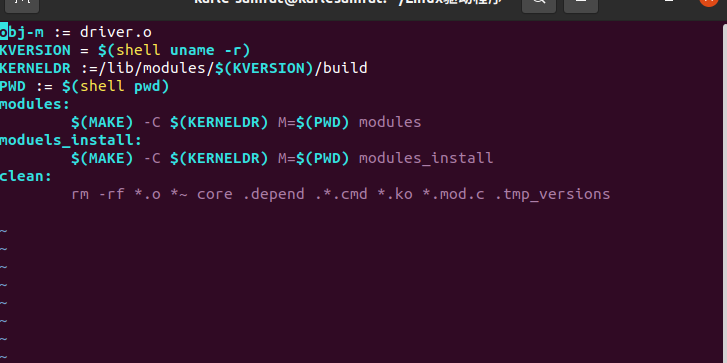
kobject\_put(kob);

printk(KERN\_INFO "Goodbye!\n");

}

module\_init(sysfs\_ctrl\_init);

module\_exit(sysfs\_ctrl\_exit);



obj-m := driver.o

KVERSION = $(shell uname -r)

KERNELDR :=/lib/modules/$(KVERSION)/build

PWD := $(shell pwd)

modules:

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

moduels\_install:

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

clean:

rm -rf \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c .tmp\_versions



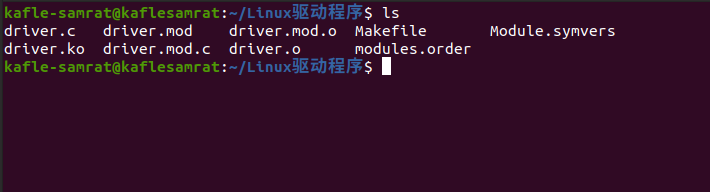
编译、安装、删除该模块，查看该模块的安装位置、运行情况

本次采用单独编译、动态插入内核；把将开发的内核代码文件直接进行编译，然后使用命令动态插入内核或者从内核卸载。

优点：编译速度快；单独调试代码

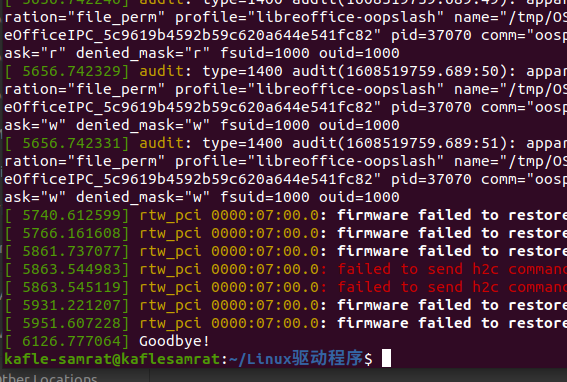
缺点：每次系统启动后都需要再加载代码

.



删除模块:

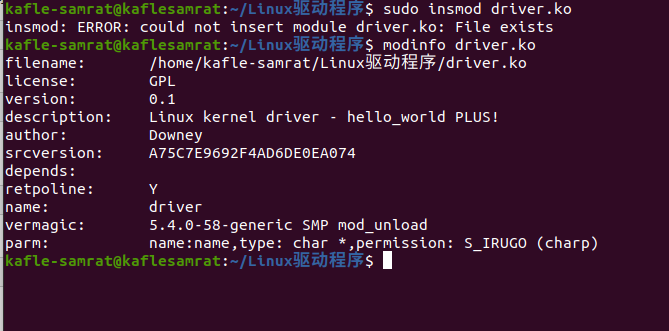
sudo rmmod driver.ko

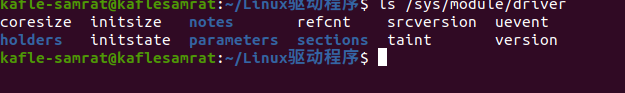


实验心得:

通过这次实验，我学习了基本的Linux内核模块开发框架和编译方法，熟悉了添加Linux内核模块的过程，了解了程序的工作原理。

.

.

.