**Linux操作系统**

**实验报告**

**学 号\_\_\_21071003\_\_\_\_\_\_\_**

**姓 名\_\_\_高立扬\_\_\_\_\_\_\_\_\_**

**提交日期\_\_\_2024/4/\_\_\_\_\_\_**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **出勤情况** | **表现情况** | **考核情况** | **程序执行情况** | **问题回答情况** |
| **□按时出勤**  **□有缺席**  **□从未出现** | **□积极提问**  **□认真设计**  **□与同学一起商量**  **□主要靠别人讲解**  **□完全不参与** | **□演示实验代码**  **□进行答辩**  **□源代码分析** | **□顺畅**  **□有问题，经过老师指出之后改正**  **□有问题，无法改正** | **□立即正确回答**  **□经思考后正确回答**  **□回答有部分错误**  **□回答完全错误**  **□不能回答问题** |
| **报告结构** | **报告内容** | **报告图表** | **最终成绩** | |
| **□完全符合要求**  **□基本符合要求**  **□有比较多的缺陷**  **□完全不符合要求** | **□充实正确**  **□基本正确**  **□有一些问题**  **□问题很大** | **□符合规范**  **□基本符合规范**  **□有一定错误**  **□完全不正确** |  | |

**教师签字:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

目录

**[实验一 添加一个新的系统调用 1](#_Toc32176)**

[一、 实验目的 1](#_Toc24129)

[二、 实验内容介绍 1](#_Toc30606)

[三、 实验要求 1](#_Toc32728)

[四、 实验设计 1](#_Toc6871)

[五、 实验结果及分析 2](#_Toc23798)

[六、 运行结果展示 2](#_Toc4406)

[七、 实验解剖体会 2](#_Toc10452)

**[实验二 内核模块的添加和卸载 3](#_Toc6222)**

[一、 实验目的 3](#_Toc27790)

[二、 实验内容介绍 3](#_Toc30387)

[三、 实验要求 4](#_Toc14027)

[四、 实验设计 4](#_Toc16734)

[五、 实验结果及分析 4](#_Toc19908)

[六、 运行结果展示 4](#_Toc15912)

[七、 实验解剖体会 5](#_Toc7782)

**[实验三 编写系统时钟模块 6](#_Toc21211)**

[一、 实验目的 6](#_Toc14755)

[二、 实验内容介绍 6](#_Toc11714)

[三、 实验要求 8](#_Toc31024)

[四、 实验设计 8](#_Toc31729)

[五、 实验结果及分析 8](#_Toc28478)

[六、 运行结果展示 8](#_Toc25241)

[七、 实验解剖体会 9](#_Toc8432)

**实验一 添加一个新的系统调用**

1. **实验目的**

理解操作系统内核与应用程序的接口关系；加深对内核空间和用户空间的理解；学会增加新的系统调用。

1. **实验内容介绍**

2.1 数据结构及变量说明

无。

2.2 完成功能的主要函数说明

主要由main函数完成。main函数用syscall()对自行添加的系统调用进行调用。

2.3 主要流程图

图1

图1 实验一主要流程图

2.4 主要程序段的功能描述和代码注释

main函数的功能是调用本实验中自行添加的系统调用。代码（含注释）如下：

#include <stdio.h>

#include <linux/kernel.h>

#include <unistd.h>

#include <sys/syscall.h>

int main(){

long a;

a = syscall(335, 666); // 系统调用

printf("The number from syscall is %ld\n", a); // 输出调用返回值

return 0;

}

1. **实验要求**

首先增加一个系统调用函数，然后连接新的系统调用，重建新的Linux内核

（4.20.13），用新的内核启动系统，使用新的系统调用。

1. **实验设计**

4.1 功能设计

添加一个系统调用，主要更改的内核文件有：sys.c、syscall.h、syscall\_64.tbl，修改完毕安装内核，编写上文提到的main函数进行调试，最后观察输出结果从而确定添加是否成功。

4.2 数据结构

无。

4.3 程序框图

图2

图2 实验一调试程序main函数流程图

1. **实验结果及分析**

编译并运行调试程序，得到输出结果如图3所示。可见系统调用以及返回值均正确，证明系统调用添加成功。

1. **运行结果展示**

**b314c212bb1c5c7f7350b906d04a817**

图3 运行结果

1. **实验解剖体会**

本次实验，我按照指导书的步骤提示，成功添加了一个系统调用。这背后的原理是我应当自行思考探究的。根据观察，注意到系统调用的类型是asmlinkage，这是系统调用中的一个重要gcc tag，使得本调用直接从栈中取参数，而不是从寄存器取，以确保正确的堆栈操作和参数传递。

通过本次实验，我学会了编译并安装内核的技巧，学会了添加新的系统调用。

**实验二 内核模块的添加和卸载**

1. **实验目的**

了解模块的编程方法。

1. **实验内容介绍**

2.1 数据结构及变量说明

无。

2.2 完成功能的主要函数说明

完成模块添加和卸载的函数主要是module\_init和module\_exit，其实他们不是函数，是宏。init负责注册驱动、申请资源、创建节点，exit则为前者的逆过程。

2.3 主要流程图

图3

图4 实验二主要流程图

2.4 主要程序段的功能描述和代码注释

编写HelloWorld.c，代码如下

#include <linux/module.h>

#include <linux/init.h>

/\* 模块加载时运⾏的代码，可⾃定义函数名 \*/

static int moduletest\_init(void){

printk("HelloWorld!\n");

return 0;

}

/\* 模块卸载时运⾏的代码 \*/

static void moduletest\_exit(void){

printk("HelloWorld exit!\n");

}

/\* 模块加载宏和卸载宏，括号⾥填写上⾯两个函数名 \*/

module\_init(moduletest\_init);

module\_exit(moduletest\_exit);

/\* 模块遵循的公共许可证 \*/

MODULE\_LICENSE("GPL");

代码编写完毕，还需要进行make 编译，Makefile如下

obj-m := HelloWorld.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean

1. **实验要求**

编写HelloWorld模块，实现编译、加载和卸载模块。

1. **实验设计**

4.1 功能设计

自定义一个模块，通过linux的模块加载宏和卸载宏，完成模块的添加和卸载。

4.2 数据结构

无。

4.3 程序框图

图5

图5 实验二

1. **实验结果及分析**

make编译HelloWorld.c，得到对应的.ko内核模块，insmod进行模块添加，lsmod查看模块添加成功，同时dmesg可以看到模块添加时printk的内容。rmmod进行模块卸载，通过lsmod和dmesg观察到模块卸载完毕。

1. **运行结果展示**

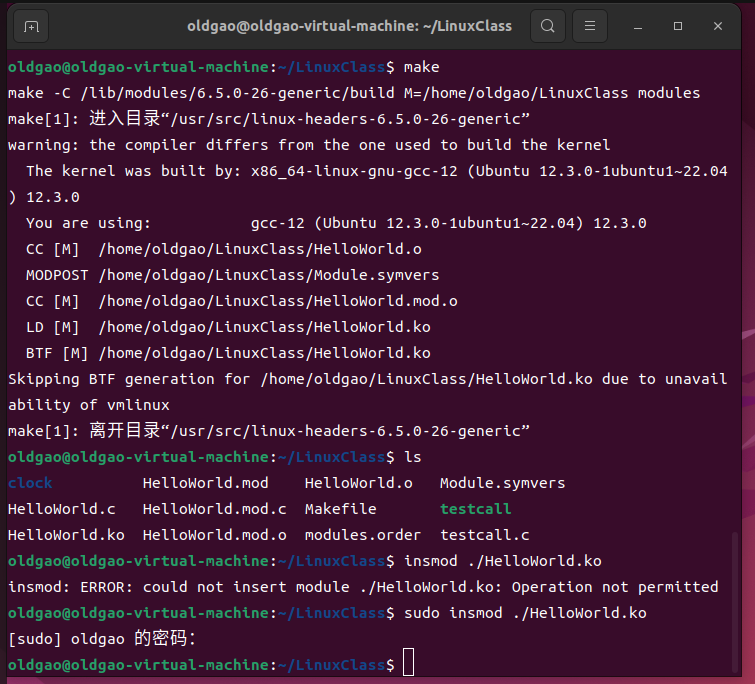
****

图6 make编译&insmod

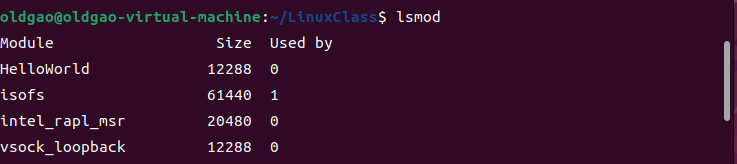


图7 lsmod看到模块加载成功

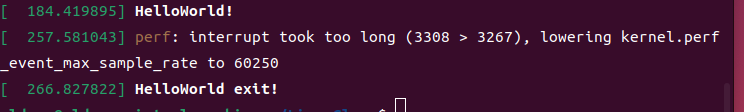


图8 dmesg查看模块添加和卸载时的输出语句

1. **实验解剖体会**

本次实验，我学会了编译模块，学会了添加和卸载内核模块。

编译用到了make，Makefile的内容由⼀组规则组成，每个规则都具有以下形式：

target: requirements

target build instructions

其中 target build instructions 前必须使⽤TAB缩进，否则会出现 Makefile:2: \*\*\* missing separator. Stop.等错误信息。obj-m 的⽬的是让内核中的 kbuild 系统通过 mod\_name.c ⽂件编译出 mod\_name.o，连接这些⽂件后得到内核模块 mod\_name.ko。

而加载和卸载宏，调用的时机就是添加和卸载模块的时候。

**实验三 编写系统时钟模块**

1. **实验目的**

理解/proc⽂件系统的作⽤、原理和机制

1. **实验内容介绍**

2.1 数据结构及变量说明

struct file\_operations my\_clock\_fops，是一个简单的fops结构体，除open方法以外直接使用seq\_file接口已实现的⽅法，open方法为自行定义的int proc\_clock\_open函数，通过\*show⽅法⼀次性输出虚拟⽂件中的内容。

自定义/proc父目录结构体如下

struct proc\_dir\_entry \*my\_clock\_dir;

struct proc\_dir\_entry \*my\_clock\_file;

获取系统时钟的结构体struct timeval tv;

2.2 完成功能的主要函数说明

时钟模块代码写在clock.c文件内。clock.c定义了模块信息、自定义了/proc父目录、定义了简单的fops结构体及其自定义的open方法，使得虚拟文件能一次性输出、定义了输出系统当前时间函数，最后编写了加载模块和卸载模块时的函数及其语句：加载模块时创建/proc下的虚拟文件，卸载模块时移除这些文件。

2.3 主要流程图

图9

图9 实验三主要流程图

2.4 主要程序段的功能描述和代码注释

/\* clock.c \*/

#include <linux/module.h>

#include <linux/proc\_fs.h>

#include <linux/seq\_file.h>

#include <linux/time.h>

/\* prototypes \*/

static int proc\_clock\_show(struct seq\_file\*m, void\*v);

static int proc\_clock\_open(struct inode\*inode, struct file\*file);

static int init\_clock(void);

static void cleanup\_clock(void);

/\* 定义模块信息 \*/

#define USER\_MODULE\_VERSION "1.0"

#define USER\_ROOT\_DIR "clock"

#define MODULE\_NAME "my\_clock"

/\*⾃定义/proc⽗⽬录\*/

struct proc\_dir\_entry \*my\_clock\_dir;

struct proc\_dir\_entry \*my\_clock\_file;

/\*定义⼀个简单的fops结构体，除open⽅法以外直接使⽤seq\_file接⼝已实现的⽅法\*/

static const struct file\_operations my\_clock\_fops =

{

.owner = THIS\_MODULE,

.open = proc\_clock\_open,

.read = seq\_read,

.llseek = seq\_lseek,

.release = single\_release,

};

/\*输出读取的当前系统时间\*/

static int proc\_clock\_show(struct seq\_file \*m, void\*v){

struct timeval tv;

do\_gettimeofday(&tv);

seq\_printf(m, "%ld %ld\n", tv.tv\_sec, tv.tv\_usec);

return 0;

}

/\*open⽅法通过\*show⽅法⼀次性输出虚拟⽂件中的内容\*/

static int proc\_clock\_open(struct inode \*inode, struct file \*file){

return single\_open(file, proc\_clock\_show, NULL);

}

/\*加载模块时创建/proc下的虚拟⽂件\*/

static int \_\_init init\_clock(void){

printk("clock: init\_module()\n");

my\_clock\_dir=proc\_mkdir(USER\_ROOT\_DIR, NULL);

my\_clock\_file=proc\_create(MODULE\_NAME, 0, my\_clock\_dir, &my\_clock\_fops);

printk(KERN\_INFO"%s %s has initialized.\n", MODULE\_NAME, USER\_MODULE\_VERSION);

return 0;

}

/\*卸载模块时将proc entry移除\*/

static void \_\_exit cleanup\_clock(void){

printk("clock: cleanup\_module()\n");

remove\_proc\_entry(MODULE\_NAME, my\_clock\_dir);

remove\_proc\_entry(USER\_ROOT\_DIR, NULL);

printk(KERN\_INFO"%s %s has removed.\n", MODULE\_NAME, USER\_MODULE\_VERSION);

}

module\_init(init\_clock);

module\_exit(cleanup\_clock);

MODULE\_DESCRIPTION("clock module for gettimeofday of proc.");

MODULE\_LICENSE("GPL");

1. **实验要求**

编写⼀个时钟模块读取系统当前时间，并在/proc⽬录下创建⽤于交换数据的虚拟⽂件，使⽤户可以通过读取该⽂件获得模块在内核模式读取到的时间数据

1. **实验设计**

4.1 功能设计

编写testclock.c文件，对/proc/clock/my\_clock文件进行读取，并输出读取到的系统时间。

4.2 数据结构

无。

4.3 程序框图

图10

图10 实验三testclock.c程序框图

1. **实验结果及分析**

编译clock.c后，加载模块，用testclock.c编译后的文件，或cat指令，对/proc/clock/my\_clock文件进行读取，看到了系统时间证明模块加载成功。dmesg查看printk输出信息同样证明模块的加载和卸载是成功的。

4月3日，由于在实验课做实验时没有下载Ubuntu 18.04 server，而校园网下载起来比较缓慢，因此我借用杨嘉鹏同学的电脑完成了实验，当时他已经检查完三个实验。下面的运行结果均是在杨嘉鹏同学的虚拟机上,通过我的自主实验，运行后的截图。

1. **运行结果展示**

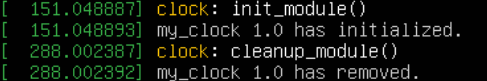


图11 dmesg查看加载模块和卸载模块的结果

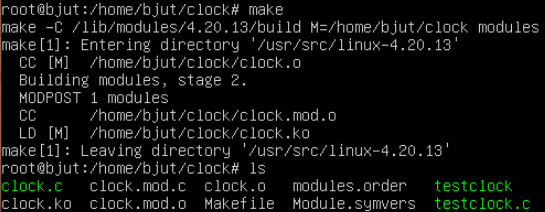


图12 make编译结果

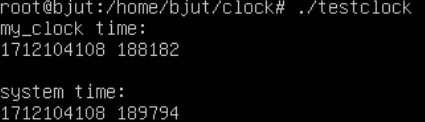


图13 testclock运行结果



图14 insmod查看模块加载情况

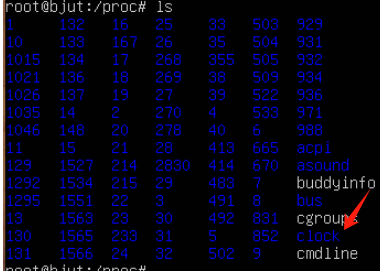


图15 查看proc下是否生成clock文件夹

1. **实验解剖体会**

本次实验，我学会了添加系统时钟调用的流程。添加系统时钟首先需要创建文件的fops结构体，用来在/proc目录一次性输出含有系统时钟的文件内容。最后编写init和exit函数，分别进行文件输出和删除，并输出对应的语句。

感谢杨嘉鹏同学的帮助，最终我能按时完成实验和检查。