



湖南大学

HUNAN UNIVERSITY

操作系统实验报告

课 程 名 称： 操作系统

实验项目名称： 操作系统内核编程实验

专 业 班 级： 软件 2203

姓 名： 白旭

学 号： 202226010306

指 导 教 师： 周军海

完 成 时 间： 2024 年 4 月 28 日

信息科学与工程学院

实验题目：实验二 内核模块编程

实验目的：

本实验要求编写四个模块，分别实现以下功能：

- 模块一，加载和卸载模块时在系统日志输出信息。
- 模块二，支持整型、字符串、数组参数，加载时读入并打印。
- 模块三，在/proc 下创建只读文件。
- 模块四，在/proc 下创建文件夹，并创建一个可读可写的文件。

需要注意以下问题：

1. 模块写在一个 c 文件中，参数传递参考宏定义 module_param(name, type, perm), 需要用到头文件 linux/moduleparam.h
2. 编写 Makefile 文件将 c 源码编译成 .ko 的模块
3. 模块下 proc 目录和文件的创建参考 proc_make() 和 proc_create 函数
4. 写入 proc 文件时，可以考虑解决写缓冲溢出的问题（可选）

实验环境：

- 华为 ESC 弹性云服务器
- WinScp

实验内容及操作步骤：

1. 打开 ESC 弹性云服务器：



2. 使用 powershell 连接服务器与主机：

```
Welcome to Huawei Cloud Service

Welcome to 4.19.90-2110.8.0.0119.oe1.aarch64

System information as of time: Sun Apr 21 01:58:26 CST 2024

System load:      0.93
Processes:        164
Memory used:      4.9%
Swap used:        0.0%
Usage On:         8%
IP address:       192.168.1.122
Users online:     1

[root@openeuler ~]# |
```

3. 编写代码

1) module1.c

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

// KERN_INFO是用于打印一般信息的日志级别
// 模块加载时调用的初始化函数
static int __init hello_init(void)
{
    printk(KERN_INFO "Hi, Module1 is loaded.\n");
    return 0; // 返回0表示模块加载成功
}

// 模块卸载时调用的退出函数
static void __exit hello_exit(void)
{
    printk(KERN_INFO "Hi, Module1 is removed.\n");
}

// 声明初始化和退出函数
module_init(hello_init);
module_exit(hello_exit);

// 模块许可证信息
MODULE_LICENSE("GPL");
```

2) module2.c

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>

#define MAX_ARRAY 6 // 定义整数数组的最大长度为6

static int int_var = 0; // 定义一个整数变量并初始化为0
static char *str_var = "default"; // 定义一个字符串指针变量并初始化为"default"
static int int_array[MAX_ARRAY]; // 定义一个整数数组

int narr; // 定义一个变量用于存储整数数组的大小

// 使用module_param宏将int_var注册为模块参数，类型为int，访问权限为0644，同时添加描述信息
module_param(int_var, int, 0644);
MODULE_PARAM_DESC(int_var, "An integer variable");

// 使用module_param宏将str_var注册为模块参数，类型为charp（字符指针），访问权限为0644，同时添加描述信息
module_param(str_var, charp, 0644);
MODULE_PARAM_DESC(str_var, "A string variable");

// 使用module_param_array宏将int_array注册为模块参数，类型为int，访问权限为0644。
// 传入指向narr的指针作为参数大小，并添加描述信息
module_param_array(int_array, int, &narr, 0644);
MODULE_PARAM_DESC(int_array, "An integer array");

// 模块加载时调用的初始化函数
static int __init hello_init(void)
{
    int i;
    printk("Module2 is loaded!\n"); // 打印模块加载信息
    printk("int_var is %d.\n", int_var); // 打印整数参数的值
    printk("str_var is %s.\n", str_var); // 打印字符串参数的值
    for (i = 0; i < narr; i++)
    { // 遍历整数数组并打印每个元素的值
        printk("int_array[%d] = %d\n", i, int_array[i]);
    }
    return 0; // 返回0表示模块加载成功
}

// 模块卸载时调用的退出函数
static void __exit hello_exit(void)
{
    printk("Module2 is removed!\n"); // 打印模块卸载信息
}

// 注册模块的初始化和退出函数
module_init(hello_init);
module_exit(hello_exit);

// 指定模块的许可证信息为GPL
MODULE_LICENSE("GPL");
```

3) module3.c

```
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/kernel.h>
#include <linux/uaccess.h>

#define PROC_NAME "hello_proc" // 定义/proc目录下的文件名为hello

static struct proc_dir_entry *proc_entry;

// 读取/proc/hello文件的回调函数
static ssize_t hello_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    char message[100]; // 存储消息的缓冲区
    int len;

    len = snprintf(message, sizeof(message), "Hello from /proc/%s!\n", PROC_NAME); // 构造消息内容
    if (*pos > 0 || count < len)
        return 0;
    if (copy_to_user(buf, message, len) != 0)
        return -EFAULT;
    *pos = len;
    return len;
}

// 定义文件操作结构体, 指定回调函数
static const struct file_operations hello_fops = {
    .owner = THIS_MODULE,
    .read = hello_read,
};

// 模块初始化函数
static int __init hello_init(void)
{
    // 在/proc目录下创建hello文件, 权限为只读
    proc_entry = proc_create(PROC_NAME, 0444, NULL, &hello_fops);
    printk(KERN_INFO "/proc/%s created\n", PROC_NAME);
    return 0;
}

// 模块退出函数
static void __exit hello_exit(void)
{
    // 移除/proc/hello文件
    proc_remove(proc_entry);
    printk(KERN_INFO "/proc/%s removed\n", PROC_NAME);
}

// 注册模块的初始化和退出函数
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

4) module4.c

```
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/kernel.h>
#include <linux/uaccess.h>
#include <linux/slab.h>

#define PROC_DIR_NAME "hello_dir" // 定义/proc目录下的文件夹名为myprocdir
#define PROC_FILE_NAME "hello" // 定义文件夹中的文件名为myprocfile

static struct proc_dir_entry *proc_dir; // 指向创建的文件夹的指针
static struct proc_dir_entry *proc_file; // 指向创建的文件的指针
static char *proc_buffer; // 文件的缓冲区
static size_t proc_buffer_size = 256; // 缓冲区大小为256字节
```

```

// 读取文件的回调函数
static ssize_t proc_file_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    if (*pos >= proc_buffer_size) // 如果读取位置超过了缓冲区大小, 返回0表示读取结束
        return 0;

    if (count > proc_buffer_size - *pos) // 如果要读取的字节数超过了剩余的字节数, 调整count为剩余的字节数
        count = proc_buffer_size - *pos;

    if (copy_to_user(buf, proc_buffer + *pos, count)) // 将缓冲区的数据复制到用户空间
        return -EFAULT;

    *pos += count; // 更新读取位置
    return count; // 返回实际读取的字节数
}

// 写入文件的回调函数
static ssize_t proc_file_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)
{
    if (*pos + count > proc_buffer_size) // 如果写入位置加上要写入的字节数超过了缓冲区大小, 返回错误
        return -ENOMEM;

    if (copy_from_user(proc_buffer + *pos, buf, count)) // 将用户空间的数据复制到缓冲区
        return -EFAULT;

    *pos += count; // 更新写入位置
    return count; // 返回实际写入的字节数
}

// 定义文件操作结构体, 指定读写回调函数
static const struct file_operations proc_file_fops = {
    .owner = THIS_MODULE,
    .read = proc_file_read,
    .write = proc_file_write,
};

// 模块初始化函数
static int __init proc_init(void)
{
    // 分配缓冲区内存
    proc_buffer = kmalloc(proc_buffer_size, GFP_KERNEL);
    if (!proc_buffer)
        return -ENOMEM;

    // 在/proc目录下创建myprocdir文件夹
    proc_dir = proc_mkdir(PROC_DIR_NAME, NULL);
    if (!proc_dir)
    {
        kfree(proc_buffer);
        return -ENOMEM;
    }

    // 在myprocdir文件夹下创建myprocfile文件
    proc_file = proc_create_data(PROC_FILE_NAME, 0666, proc_dir, &proc_file_fops, NULL);
    if (!proc_file)
    {
        remove_proc_entry(PROC_DIR_NAME, NULL);
        kfree(proc_buffer);
        return -ENOMEM;
    }

    printk(KERN_INFO "/proc/%s/%s created\n", PROC_DIR_NAME, PROC_FILE_NAME);
    return 0;
}

// 模块退出函数
static void __exit proc_exit(void)
{
    // 移除myprocfile文件
    if (proc_file)
        remove_proc_entry(PROC_FILE_NAME, proc_dir);

    // 移除myprocdir文件夹
    if (proc_dir)
        remove_proc_entry(PROC_DIR_NAME, NULL);

    // 释放缓冲区内存
    if (proc_buffer)
        kfree(proc_buffer);

    printk(KERN_INFO "/proc/%s/%s removed\n", PROC_DIR_NAME, PROC_FILE_NAME);
}

// 注册模块的初始化和退出函数
module_init(proc_init);
module_exit(proc_exit);
MODULE_LICENSE("GPL");

```

4. 模块功能验证

- 1) 正确编写满足功能的源文件，包括.c 源文件和 Makefile 文件

```
[root@openeuler module]# ls
cmdline.sh Makefile module1.c module2.c module3.c module4.c
```

- 2) 编译源文件

```
[root@openeuler module]# make
make -C /lib/modules/4.19.90-2110.8.0.0119.oe1.aarch64/build M=/root/module modules
make[1]: Entering directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarch64'
CC [M] /root/module/module1.o
CC [M] /root/module/module2.o
CC [M] /root/module/module3.o
CC [M] /root/module/module4.o
Building modules, stage 2.
MODPOST 4 modules
CC /root/module/module1.mod.o
LD [M] /root/module/module1.ko
CC /root/module/module2.mod.o
LD [M] /root/module/module2.ko
CC /root/module/module3.mod.o
LD [M] /root/module/module3.ko
CC /root/module/module4.mod.o
LD [M] /root/module/module4.ko
```

- 3) 加载编译完成的内核模块，并查看加载结果

```
[root@openeuler module]# insmod module1.ko
[root@openeuler module]# insmod module2.ko int_var=666 str_var=hello int_array=10,20,30,40
[root@openeuler module]# insmod module3.ko
[root@openeuler module]# insmod module4.ko
[root@openeuler module]# lsmod | grep module
module4                262144  0
module3                262144  0
module2                262144  0
module1                262144  0
```

```
[root@openeuler module]# cat /proc/hello
Hello from /proc/hello!
[root@openeuler module]# cat /proc/myprocdir/myprocfile
Va++++,a++++,a++++(,a++++(,a++++@,a++++@,a++++P,a++++P,a+++++5  ++
a++++[
root@openeuler module]# iecho nice > /proc/myprocdir/myprocfile
[root@openeuler module]# cat /proc/myprocdir/myprocfile
nice
+++ ,a++++,a++++(,a++++(,a++++@,a++++@,a++++P,a++++P,a+++++5  ++
a++++[
```

- 4) 卸载内核模块，并查看结果

```
root@openeuler module]# rmmod module4.ko
[root@openeuler module]# rmmod module3.ko
[root@openeuler module]# rmmod module2.ko
[root@openeuler module]# rmmod module1.ko
```

```
[ 1228.857497] Hi, Module1 is loaded.
[ 1228.859746] Module2 is loaded!
[ 1228.859965] int_var is 666.
[ 1228.860168] str_var is hello.
[ 1228.860394] int_array[0] = 10
[ 1228.860624] int_array[1] = 20
[ 1228.860914] int_array[2] = 30
[ 1228.861101] int_array[3] = 40
[ 1228.863333] /proc/hello created
[ 1235.026872] /proc/myprocdir/myprocfile created
[ 1504.235447] /proc/myprocdir/myprocfile removed
[ 1504.256225] /proc/hello removed
[ 1504.272060] Module2 is removed!
[ 1505.677772] Hi, Module1 is removed.
```

- 5) 退出登录

实验结果及分析：

5. 实验结果与预期相符，第二次实验学习了编写内核模块相关操作，与文件权限相关知识，完成得比较顺利，源代码可以正常运行

收获与体会：

1. cat /proc/hello_dir/hello 时会出现乱码，应该是编码格式出现问题
2. 做完实验要及时将 ESC 关机以节约经费
3. 掌握了如何进行内核模块编程

思考题：

1. Linux 内核模块的基本结构是什么？

1) 头文件包含：

#include <linux/module.h>：用于 Linux 模块编程的必要头文件
其他需要的内核头文件，如<linux/kernel.h>、<linux/init.h>等

2) 模块初始化和退出函数：

``static int __init init_module(void)``：模块初始化函数，在加载模块时被调用

``static void __exit cleanup_module(void)``：模块退出函数，在卸载模块时被调用

3) 模块参数：

- 使用``module_param``宏定义模块参数，并使用``MODULE_PARM_DESC``宏添加描述信息

4) 模块许可证信息：

使用``MODULE_LICENSE``宏指定模块的许可证信息，如“GPL”或“MIT”等

5) 其他函数或数据结构：

根据模块的功能，可能会包含其他函数或数据结构的定义

6) 模块初始化和退出函数的注册：

使用``module_init``宏注册模块初始化函数。

使用``module_exit``宏注册模块退出函数。

7) 模块的编译和安装：

使用``make``等工具编译模块

使用``insmod``命令加载模块

使用``rmmod``命令卸载模块

8) 模块的日志输出:

使用 `printk` 函数向内核日志输出信息, 以便在调试时查看

9) 模块的 Makefile:

包含编译模块所需的规则和指令, 通常使用 `obj-m` 变量指定模块的目标文件

2. `proc` 文件系统的解释

`proc` 文件系统是一种特殊的文件系统, 它不存储在硬盘上, 而是由内核在内存中创建的, 用于向用户空间提供内核和系统信息的接口。`proc` 文件系统下的文件和目录并不是真实的文件和目录, 而是内核中各种数据结构的映射, 通过读取这些文件, 用户可以获取有关系统状态和配置的信息, 甚至可以修改一些内核参数。

3. Linux 的内核空间和用户空间的区分

1) 内存访问权限

内核空间可以访问系统的所有内存地址, 并且拥有对硬件的直接访问权限; 用户空间只能访问其自己的内存地址空间, 并且无法直接访问硬件

2) 数据共享

内核空间中的数据可以被所有进程访问和共享, 因为内核是系统的核心组件; 用户空间的数据只能被拥有相应权限的进程访问

3) 运行环境

内核空间中运行的代码是操作系统内核本身的组成部分, 用于管理系统资源和提供服务; 用户空间中运行的代码是用户应用程序, 用于实现各种功能和服务

4. `file_operations` 结构体的作用

`file_operations` 结构体是 Linux 内核中用于表示文件操作的结构体, 它定义了一组函数指针, 这些函数指针指向了一组操作文件的函数。当用户空间的进程对文件进行操作时 (如打开、读取、写入、关闭等操作), 内核会根据文件类型和文件系统调用相应的 `file_operations` 结构体中的函数来执行对应的操作。主要作用包括定义文件操作函数, 实现文件操作逻辑, 连接文件系统和 VFS 层, 提供对用户空间的接口, 实现文件的具体行为。总的来说, `file_operations` 结构体定义了文件操作的接口和实现, 是文件系统与内核之间交互的重要接口之一。

实验
成绩

实验报告撰写说明

1. 实验题目和目的

请从实验指导资料中获取。

2. 实验步骤和内容

包括：

（1）本次实验的要求；

（2）源程序清单或者主要伪代码；

（3）预期结果；

（4）上机执行或调试结果：包括原始数据、相应的运行结果和必要的说明（截图）；

3. 实验体会

调试中遇到的问题及解决办法；若最终未完成调试，要试着分析原因；调试程序的心得与体会；对课程及实验的建议等。