实验 1：内核模块编程

1. 实验目的

（1） 学习基本的内核模块编写及加载/卸载过程

（2） 学习编译内核

2. 实验内容与任务

本实验要求编写四个模块，分别实现以下功能：

（1） 模块一，加载和卸载模块时在系统日志输出信息。

1) module_init 和 module_exit 是两个特殊的函数，它们分别在模块加载和卸载时被调用

2) 请参考加载模块的相关代码，在卸载模块时打印“Hi, Module1 is removed.”

（2） 模块二，支持整型、字符串、数组参数，加载时读入并打印。

1) 在模块初始化函数中，添加一个循环来遍历整型数组，并打印每个元素的值

2) 使用 printk 函数来输出每个数组元素，格式为："int_array[%d] = %d\n"

（3） 模块三，在/proc 下创建只读文件。

1) 在模块的初始化函数中，添加 proc_create 函数来创建一个只读的 /proc 文件。此文件将使用 hello_proc_fops 文件操作结构，该结构定义了文件的操作方法

2) 在模块的卸载函数中，使用 remove_proc_entry 函数来删除之前创建的 /proc 文件

（4） 模块四，在/proc 下创建文件夹，并创建一个可读可写的文件。

1) 补全 hello_write 函数，以允许从用户空间接收字符串并保存到内核空间

2) 在写入函数中，使用 kzalloc 为字符串分配内存，使用 copy_from_user 安全地从用户空间复制数据

（5） 学习编译内核，并分析 Linux 内核模块的基本结构是怎样？

参考实验步骤完成上述任务，其中，部分代码已经给出。提交需要包含补充的代码和上文/实验步骤中要求打印内容的截图。

需要注意以下问题：

1. 模块可以写在一个 c 文件中，模块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）。

3. 准备知识

（1）Linux 内核模块

内核模块是一种可以扩展运行时内核功能的目标文件（Object File）。大多数类 Unix 及 Windows 系统均使用模块，这种机制使得允许内核在运行过程中动态地插入或者移除代码。

Linux 是一个跨平台的操作系统，支持众多的设备，在 Linux 内核源码中有超过 50%的代码都与设备驱动相关。 Linux 为宏内核架构，如果开启所有的功能，内核就会变得十分臃肿。 内核模块就是实现了某个功能的一段内核代码，在内核运行过程，可以加载这部分代码到内核中， 从而动态地增加了内核的功能。基于这种特性，我们进行设备驱动开发时，以内核模块的形式编写设备驱动， 只需要编译相关的驱动代码即可，无需对整个内核进行编译。

内核模块的引入不仅提高了系统的灵活性，对于开发人员来说更是提供了极大的方便。 在设备驱动的开发过程中，我们可以随意将正在测试的驱动程序添加到内核中或者从内核中移除， 每次修改内核模块的代码不需要重新启动内核。 在资源有限的开发板上，我们也不需要将内核模块程序，或者说设备驱动程序的 ELF 文件存放在开发板中， 免去占用不必要的存储空间。当需要加载内核模块的时候，可以通过挂载诸如 NFS 服务器，将存放在其他设备中的内核模块，加载到开发板上。 在某些特定的场合，我们可以按照需要加载/卸载系统的内核模块，从而更好的为当前环境提供服务。

了解了内核模块引入以及带来的诸多好处，我们可以建立起对内核模块的初步认识，下面给出内核模块的具体的定义：内核模块全称 Loadable Kernel Module(LKM)，是一种在内核运行时加载一组目标代码来实现某个特定功能的机制。

模块是具有独立功能的程序，它可以被单独编译，但不能独立运行， 在运行时它被链接到内核作为内核的一部分在内核空间运行，这与运行在用户空间的进程是不一样的。 模块由一组函数和数据结构组成，用来实现一种文件系统、一个驱动程序和其他内核上层功能。 因此内核模块具备如下特点：

- 模块本身不被编译入内核映像，这控制了内核的大小。
- 模块一旦被加载，它就和内核中的其它部分完全一样。

ko 文件在数据组织形式上是 ELF(Executable and Linking Format)格式，是一种普通的可重定位目标文件。 这类文件包含了代码和数据，可以被用来链接成可执行文件或共享目标文件，静态链接库也可以归为这一类。

（2）内核模块加载过程

执行 insmod 命令时，必须指定要加载模块的位置；对需求加载的内核模块，通常保存在 /lib/modules/kernel-version。和系统的其他程序一样，内核模块实际是经连接的目标文件，但模块是可重定位的，也就是说，为了让装入的模块和已有的内核组件之间可以互相访问，模块不能连接为从特定地址执行的映像文件。模块可以是 a.out 或 elf 格式的目标文件。insmod 利用一个特权系统调用，可找到内核的导出符号表，符号成对出现，一个是符号名称，另外一个是符号的值，例如符号的地址。内核维护一个由 module_list 指针指向的 module 链表，其中第一个 module 数据结构保存有内核的导出符号表。并不是所有的内核符号均在符号表中导出，而只有一些特殊的符号才被添加到符号表中。

- 模块加载依赖关系

以 OpenEuler 为例，模块依赖关系在/lib/modules/{kernel-version}/modules.dep 文件中

在这里，{kernel-version}可以使用 uname -r 来查询当前正在运行的 linux 内核版本。

- 系统需要加载的模块信息

1) /etc/modules-load.d 中配置系统中要加载哪些模块
2) /etc/modprobe.d 文件夹中给出了系统模块的配置参数文件
3) /lib/modules/{kernel-version} 目录包含系统核心可加载各种模块，尤其是那些在恢复损坏的系统时重新引 导系统所需的模块(例如网络和文件系统驱动)。
4) /proc/modules 存放当前加载了哪些核心模块信息。

- Linux 相关命令

1) lsmod :列出已经被内核调入的模块
2) insmod:将某个 module 插入到内核中
3) modprobe: modprobe 命令是根据 depmod -a 的输出/lib/modules/{kernel version}/modules.dep 来加载全部的所需要模块
4) rmmod：将某个 module 从内核中卸载
5) depmod: 生成依赖文件，告诉将来的 insmod 要从哪儿调入 modules
6) lshw：列出硬件模块情况，通常可以配合 grep 来使用，查询某个具体硬件模块信息

（3）内核编程的基本框架

现在的内核模块一般具有如下形式的基本结构：

```
#include <linux/init.h>  //包含了模块初始化的宏定义和一些初始化函数
#include <linux/module.h> //包含了对模块的结构定义
#include <linux/kernel.h> //内核头文件，含有一些内核常用函数的原形定义

MODULE_LICENSE("GPL"); //声明版权

static __init my_module_init(void) {
   //加载模块
}

static __exit my_module_exit(void) {
   //卸载模块
}

module_init(my_module_init); //指定初始化函数
module_exit(my_module_exit); //指定卸载函数
```

其中，

module_init：用来指定加载驱动时要执行的模块加载函数；

module_eixt：用来指定卸载驱动时要执行的模块卸载函数。

为了添加模块的描述信息，我们通常会用到以下的这些宏

```
MODULE_LICENSE("GPL");              // 描述模块的许可证
MODULE_AUTHOR("Zhang Sun");             // 描述模块的作者
MODULE_DESCRIPTION("module test");  // 描述模块的介绍信息
MODULE_ALIAS("alias test");         // 描述模块的别名信息
```

为了在内核中记录日志信息，我们会用到 printk 函数，形如如下形式：

```
printk(KERN_INFO "Message: %s\n", message);
```

其中 KERN_INFO 是日志级别（注意，它与格式字符串连在一起，日志级别不是一个单独的参数）。日志级别指定了一条消息的重要性，系统根据此决定是否立即显示消息。可用的日志级别包括：

```
#define    KERN_EMERG     "<0>"       /* 系统不可用*/
#define    KERN_ALERT     "<1>"       /* 必须立即采取行为*/
```

```
#define    KERN_CRIT      "<2>"      /* 紧急情况*/
#define    KERN_ERR       "<3>"      /*错误情况*/
#define    KERN_WARNING   "<4>"      /*警告情况*/
#define    KERN_NOTICE    "<5>"      /*普通但是需要注意的情况*/
#define    KERN_INFO      "<6>"      /*普通信息*/
#define    KERN_DEBUG     "<7>"      /*调试级别信息*/
```

如果省略了日志级别，则以 KERN_DEFAULT 级别打印消息。

如果你想在系统启动或者模块加载时候，为参数指定相应值，传参给模块，则需要在内核模块中使用 module_param( )函数。

```
module_param(name, type, perm)
其中 name：参数名，是用户和内核模块内接受参数的变量
type：参数类型
perm：指定在 sysfs 中的相应文件的访问权限
```

（4）编译 Linux 可加载内核模块

在 linux 下编译可加载内核模块形成.ko 文件的 makefile 中的核心语句是：

```
$(MAKE) -C $(KERNEL_DIR) M=(PWD) modules
```

其中：

- $(MAKE) 是 make 工具，
- “-C”选项的作用是指将当前工作目录转移到你所指定的位置，-C $(KERNEL_DIR) 代表切换到内核编译的工作目录，因为内核源码顶层的 Makefile 文件定义了伪目标 modules，所以要先将工作目录切换到内核源码顶层 Makefile 所在位置；
- “M=”选项的作用是，当用户需要以某个内核为基础编译一个外部模块的话，需要在 make modules 命令中加入“M=dir”，程序会自动到你所指定的 dir 目录中查找模块源码，将其编译，生成 KO 文件。

（5）编译 OpenEuler 操作系统内核

1） 安装工具，构建开发环境：

```
# yum group install -y "Development Tools"
# yum install -y bc
# yum install -y openssl-devel
```

在这里可能会遇到 gpg check failed 的问题，直接加上参数—nogpgcheck 即可。如果你不想把"Development Tools"都安装上。那么你也可以执行以下语句

```
# yum groupinstall "Development Tools" --
setopt=group_package_types=mandatory --nogpgcheck
```

2) 备份 boot 目录以防后续步骤更新内核失败

```
# tar czvf boot.origin.tgz /boot/
```

保存当前内核版本信息

```
# uname -r > uname_r.log
```

3) 获取内核源代码并解压，在这里 openeuler 社区，将 openEuler-20.03-LTS-SP3，

命名为 openEuler-1.0-LTS。当然，你也可以用 yum 安装当前 openEuler 对应版

本的源码，这样就可以编译出一个当前的版本了。

```
# wget
https://gitee.com/openeuler/kernel/repository/archive/openEuler-
1.0-LTS.zip
# unzip openEuler-1.0-LTS.zip
```

注意：您可能需要在 openEuler 代码仓（https://gitee.com/openeuler/kernel）获取到正确的内核代码 URL 地址并更新其源代码包的具体文件名称。

4) 编译内核并重启系统

```
1. cd kernel-openEuler-1.0-LTS
2.
3. make clean
4. make mrproper
5.
6. make openeuler_defconfig
```

在这里，我们按源代码文件 arch/arm64/configs/openeuler_defconfig 的配置来配置内核。如果只想在原来内核配置的基础上修改一部分，可以使用 make oldconfig，如果是基于文本选择的配置界面，可以使用 make menuconfig。接下来先查看一下可编译的 Image。

```
1. make help | grep Image
```

```
* Image.gz          - Compressed kernel image (arch/arm64/boot/Image.gz)
  Image             - Uncompressed kernel image (arch/arm64/boot/Image)
```

接下来就可以编译了。

```
1. make -j4 Image modules dtbs
```

在这里，如果云系统超时断开会话，编译会导致失败，

这样可以进行后台编译

```
make -j4 Image modules dtbs > make.log 2>&1 &
```

使用以下命令，查看编译的情况。

```
tail -n 10 make.log
```

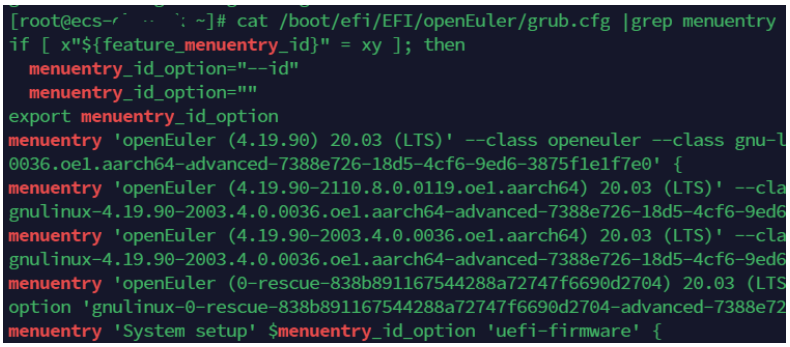
再接下来就是新内核安装

```
make modules_install
make install
reboot
```

编译后，设置我们的内核作为默认启动项

```
cat /boot/efi/EFI/openEuler/grub.cfg |grep menuentry
```

在这里面找到我们的相应项目，例如如下图这样，第一个'openEuler (4.19.90) 20.03 (LTS)'是我们刚编译的内核。




```
grub2-set-default 'openEuler (4.19.90) 20.03 (LTS)'
reboot
```

这样就会默认启动我们新编译好的内核了。

4. 实验步骤

参考 1-module-programming/exp1/task1 文件夹下源代码（注意在 Makefile 中指定新编译的 kernel 源码路径）。

参考 shell 脚本（非 root 用户部分命令需用 sudo 执行）以及以下步骤。

（1）编译源代码

```
make clean
make
```

（2）安装模块并测试

```
insmod module1.ko
insmod module2.ko int_var=666 str_var=hello int_array=10,20,30,40
insmod module3.ko
insmod module4.ko
lsmod | grep module

cat /proc/hello_proc
cat /proc/hello_dir/hello
```

```
9. echo nice > /proc/hello_dir/hello
10.cat /proc/hello_dir/hello
```

（3）卸载模块

```
1. rmmod module4.ko
2. rmmod module3.ko
3. rmmod module2.ko
4. rmmod module1.ko
```

（4）查看系统日志信息

```
1. clear
2. dmesg | tail -n80
```

（5）编译 Linux 操作系统内核

具体方法见准备知识部分