编号：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |

武汉大学国家网络安全学院

课程实验(设计)报告

课程名称 ： Linux架构分析与安全设计

实验内容 ： 实验2 Linux字符设备驱动实验

专业(班) ：

学 号 ：

姓 名 ：

任课教师 ： 王娟

2020 年 11 月 1 日

目 录

[实验二 Linux字符设备驱动实验 1](#_Toc55650837)

[1、实验名称 1](#_Toc55650838)

[2、实验目的 1](#_Toc55650839)

[3、实验步骤及内容 1](#_Toc55650840)

[4、实验关键过程及其分析 3](#_Toc55650841)

[5、问题及思考 12](#_Toc55650842)

## 实验二 Linux字符设备驱动实验

### 1、实验名称

《Linux字符设备驱动实验》

### 2、实验目的

1、熟悉Linux内核模块的开发步骤

2、熟悉Linux字符设备驱动的实现原理

### 3、实验步骤及内容

#### 第一阶段：学习Linux内核模块的开发和加载步骤

开发编译一个linux模块需要模块源代码文件和对应的Makefile文件，linux内核模块的基本格式如下：

//*引入相关内核头文件*

#include <linux/module.h>

//*内核模块信息，包括许可证、作者、描述和版本等*

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("lyl");

MODULE\_DESCRIPTION("An hello worlk module for demonstration");

MODULE\_VERSION("1.0");

//*内核模块参数，加载时指定或者动态指定，以此控制模块行为*

static char \*name = "world";

module\_param(name, charp, S\_IRUGO);

MODULE\_PARM\_DESC(name, "Whom this module say hello to");

//*初始化函数，在加载时调用，分配资源准备执行环境*

//*这里只是往内核日志输出一行记录*

static int \_\_init hello\_init(void)

{

    printk(KERN\_INFO "HELLO: Hello %s, this is hello module speaking\n", name);

    return 0;

}

//*清理函数，在卸载时调用，回收资源销毁执行环境*

static void \_\_exit hello\_exit(void)

{

    printk(KERN\_INFO "HELLO: Goodbye %s", name);

}

//*登记初始化函数及清理函数*

module\_init(hello\_init);

module\_exit(hello\_exit);

对应的Makefile文件基本格式如下

MODULE\_NAME := kernel\_test.c

obj-m := $(MODULE\_NAME).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

将源代码文件和makefile放在同一目录执行make就可以得到内核模块ko文件，使用insmod命令就可以安装.ko文件。

#### 第二阶段：学习字符设备创建的步骤并编写内核模块创建字符设备

字符设备创建相关的内核头文件如下：

#include<linux/fs.h>

#include<linux/cdev.h>

#include<linux/uaccess.h>

字符设备的创建一般包括以下步骤：

1. 为字符设备分配设备号(alloc\_chrdev\_region)
2. 初始化struct cdev结构体（cdev\_init）
3. 给结构体添加设备（cdev\_add）

对应资源释放

1. 注销struct cdev中的设备（cdev\_del）
2. 释放设备号（unregister\_chrdev\_region）

还需要为字符设备提供读写，打开和释放等操作的功能函数，具体功能可以自己实现，比如写入时记录数据，读取时将上次写入的数据输出等等，这些功能函数分别赋给struct file\_operations结构体中对应函数指针，在cdev\_init函数中和struct cdev结构体绑定。

字符设备驱动内核模块编译完成后，使用insmod加载，记得在模块初始化函数中打印出设备号，之后使用mknod命令在/dev文件夹下使用对应的设备号创建一个设备文件即可。

#### 第三阶段：编写一个程序从创建的字符设备中读取数据

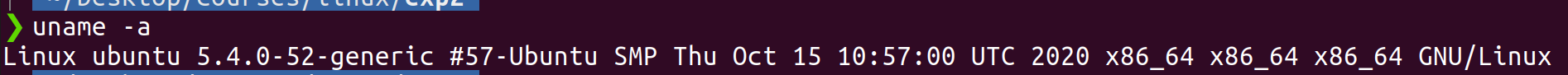
使用读写普通文件的的API编写用户程序测试读写刚刚创建的设备文件。

### 4、实验关键过程及其分析

#### 实验环境

Linux发行版本：Ubuntu20.04 64bit

Linux内核版本：



#### 实验一

###### 编写模块源代码文件hello.c

//*引入相关内核头文件*

#include <linux/module.h>

//*内核模块信息，包括许可证、作者、描述和版本等*

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("lyl");

MODULE\_DESCRIPTION("An hello worlk module for demonstration");

MODULE\_VERSION("1.0");

//*初始化函数，在加载时调用，分配资源准备执行环境*

//*这里只是往内核日志输出一行记录*

static int \_\_init hello\_init(void)

{

        printk(KERN\_INFO "Hello, world\n");

        return 0;

}

//*清理函数，在卸载时调用，回收资源销毁执行环境*

static void \_\_exit hello\_exit(void)

{

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

}

//*登记初始化函数及清理函数*

module\_init(hello\_init);

module\_exit(hello\_exit);

###### 编写相应的Makefile

obj-m+=hello.o

all:

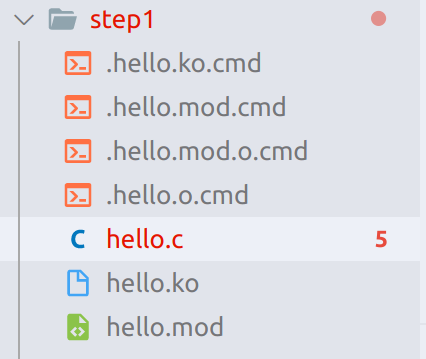
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules

clean:

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

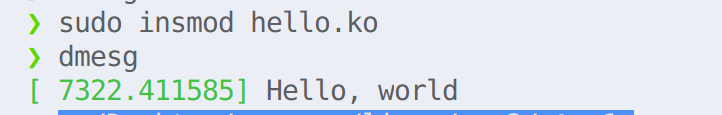
###### 进行编译

可以发现编译后出现了.ko文件

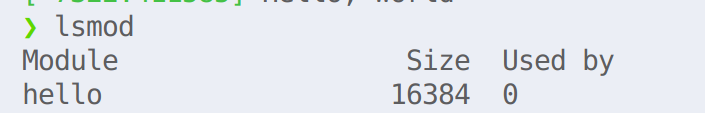


###### 模块加载和卸载

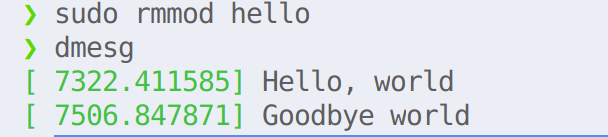
加载模块并使用dmesg查看日志信息



使用lsmod可以看到模块被成功加载



卸载模块并查看日志信息



#### 实验二

##### 概览

###### 设备驱动

**设备驱动是计算机和设备进行相互通信的特殊程序**。相当于硬件的接口，操作系统只有通过这个接口，才能控制硬件设备的工作。

**驱动程序和硬件通信的方式**：驱动程序通过特殊指定来读写寄存器或者I/O空间，从而实现驱动程序和硬件的通信。

Linux设备驱动可以分为如下几类：

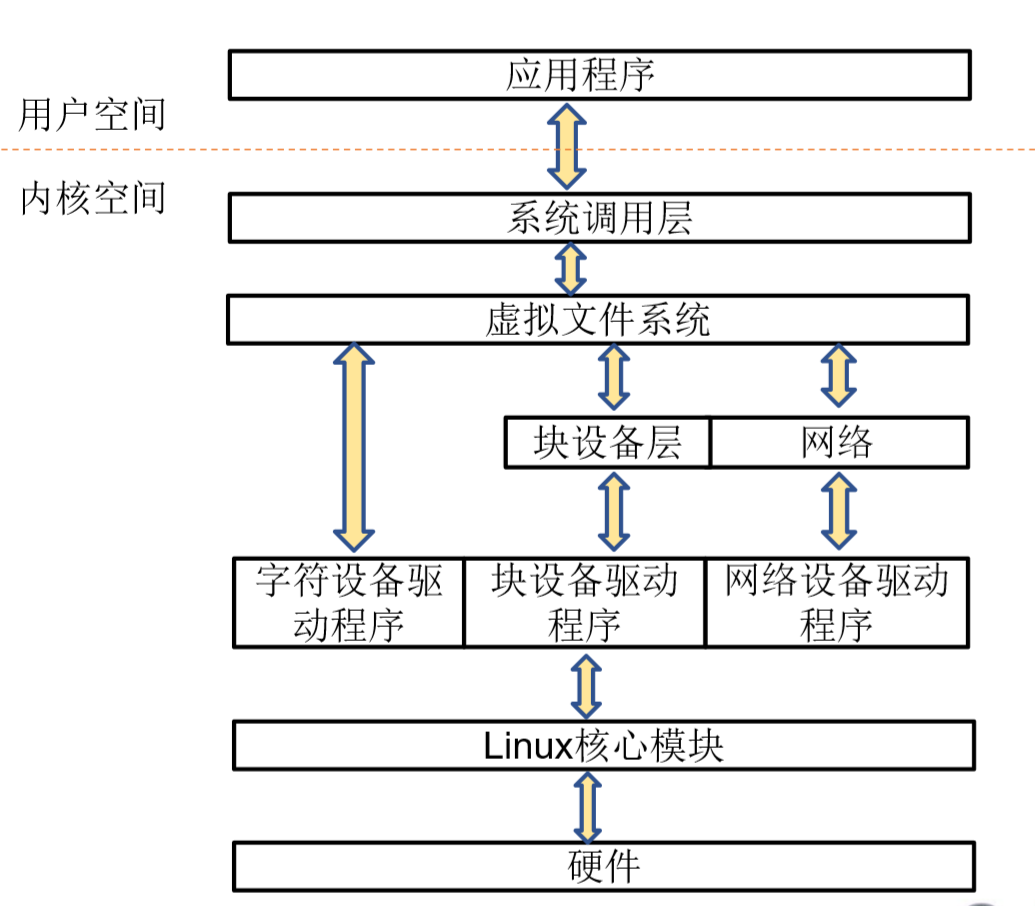
1. **字符设备**：以字节为单位的I/O传输，这种字符流的传输率通常比较低，常见的字符设备有鼠标、键盘、触摸屏等。
2. **块设备**：以块为单位进行传输的，常见的块设备是磁盘。
3. **网络设备**：一种比较特殊的设备，涉及网络协议层。

我们可以用**ls -al /dev**指令查看设备文件，其中b开头的代表**块设备文件**，c开头的代表**字符设备文件。**



缘于linux***一切皆文件***的设计哲学，其设备都已文件的形式存放在/dev目录下，该目录是一个**动态生成的、使用devtmpfs虚拟文件**系统挂载的、基于RAM的虚拟文件系统。应用程序可以打开、关闭和读写这些设备文件，完成对设备的操作，就像**操作普通的数据文件 一样**。

Linux的设备驱动框架可以用下图来表示



###### 设备标识符

为了便于管理linux下的设备，系统为每个设备编了号，设备号由**主设备号**和**次设备号**组成，他们的含义如下：

* **主设备号**：用于标识设备的类型，例如IDE disk、SCSI disk、Serial port等。
* **次设备号**：用来标区分同一类型的不同设备，例如第一个硬盘、第二个串行端口等。

为新的设备选择设备号时有两种方法：**静态分配(static)**和**动态分配**(dynamic)。

本次实验中，我们将编写一个字符设备驱动，并利用对字符设备的同步操作，设计实现一个聊天程序。可以有一个读，一个写进程共享该字符设备，进行聊天。

##### 创建字符设备

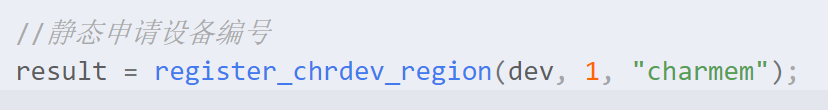
字符设备驱动管理的核心对象是以**字节**为数据流的设备。字符设备的创建一般包含如下步骤：

1. 为字符设备分配设备号，可使用如下两种方式
   1. 动态分配
   2. 静态分配
2. 初始化struct\_cdev结构体
3. 给结构体添加设备

###### 分配设备号

在这里我们使用静态分配的方式对设备号进行分配。

#define MAJOR\_NUM 456 //*主设备号 ,没有被使用*



###### 初始化struct cdev结构体

设备dev结构体定义如下

struct dev

{

    struct cdev devm; //*字符设备*

    struct semaphore sem;

    wait\_queue\_head\_t outq;  //*等待队列,实现阻塞操作*

    int flag;                //*阻塞唤醒标志*

    char buffer[MAXNUM + 1]; //*字符缓冲区*

    char \*rd, \*wr, \*end;     //*读,写,尾指针*

} globalvar;

使用cdev\_init初始化struct cdev结构体



###### 给结构体添加设备

使用cdev\_add为结构体添加设备



##### 定义功能函数

在上述使用cdev\_init初始化结构体的过程中，除了字符设备之外，我们还需要提供字符设备对应的file\_operations结构体指针，该结构体是把**系统调用和驱动程序关联**起来的关键数据结构。这个结构的每一个成员都对应着一个系统调用。当外部应用程序调用read、write时，内核会调用该设备对应驱动程序所绑定的file\_operations中相应的函数。其实file\_opeations中的成员可以看成是**回调函数**，他们将作为参数在系统调用中被使用。

###### read

读操作定义如下图所示。其主要功能为将内核中缓冲区中的数据读入到用户缓冲区中。在这里值得注意的是：**内核空间与用户空间的内存不能直接互访**。这是因为在 Linux 中，**用户内存和内核内存是独立的，在各自的地址空间实现**。事实上，内核本身驻留在一个地址空间中，每个进程驻留在自己的地址空间。因此我们需要借助copy\_to\_user()函数将一个数据块从内核复制到用户空间，copy\_from\_user()函数将一个数据块从用户空间复制到内核。



###### write

写操作定义如下图所示。其主要功能为将用户缓冲区中的数据写入内核缓冲区中。这里和读操作一样，我们不能直接进行内核空间和用户空间的数据互访，需要借助copy\_from\_user()函数完成从用户空间到内核空间的复制



###### open

在大部分的驱动程序中，open应该完成如下操作：

* 通过try\_module\_get递增使用计数
* 检查设备特定的错误
* 如果是首次打开，则需要进行初始化操作
* 识别次设备号
* 分配并填写被置于filp->private\_data里的数据结构

这里为了简单，我们只打印递增应用计数和打印信息：

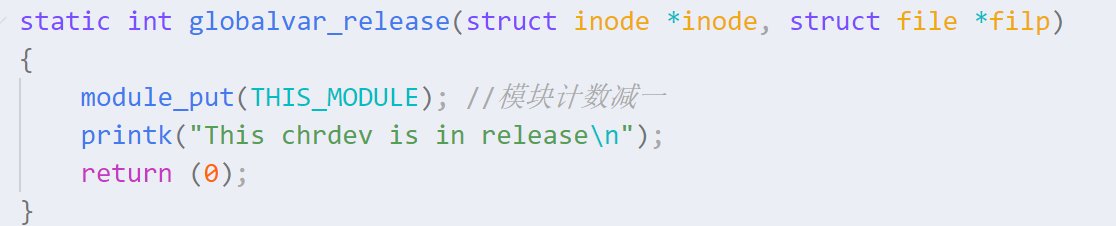


###### release

在大部分的驱动程序中，release应该完成如下的操作：

* 释放由open分配的，保存在flip->private\_data中的内容
* 通过module\_put递减使用计数
* 在最后一次关闭操作时关闭设备

这里由于open中没有进行内存的分配，我们只是简单的将模块计数减1并打印信息



##### 释放字符设备

###### 注销struct cdev中的设备

使用cdev\_del进行设备的注销



###### 释放设备号

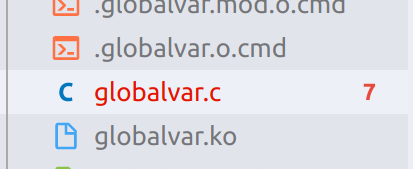
使用unregister\_chrdev\_region释放设备号



##### 模块的编译和加载

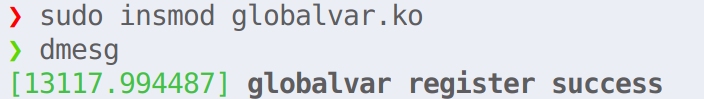
###### 编译

编译出的.ko内核文件如下

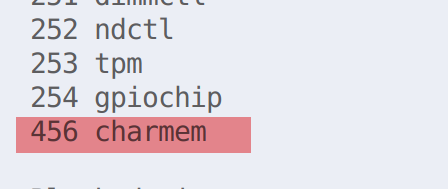


###### 加载

使用insmod加载内核模块，并通过dmesg查看内核日志



使用cat /proc/devices查看注册的设备列表，可以发现主设备号为456，设备名为charmem的设备被成功注册，这和我们在模块初始化中定义的数值一样

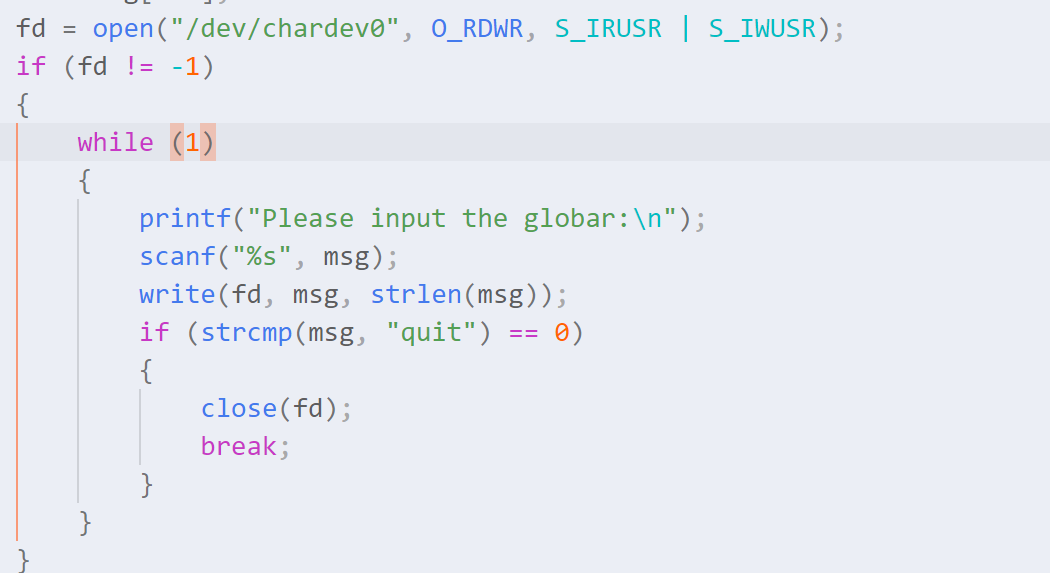


#### 实验三

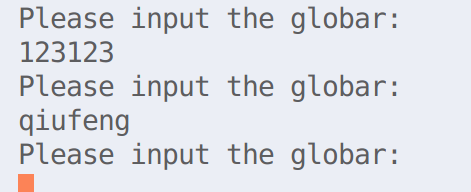
我们在read.c中调用了读api，每当设备缓冲区有新数据被写入时，该程序能够读出被写入的数据



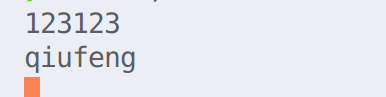
我们在write.c中调用了写api，该程序能够向设备缓冲区写入数据



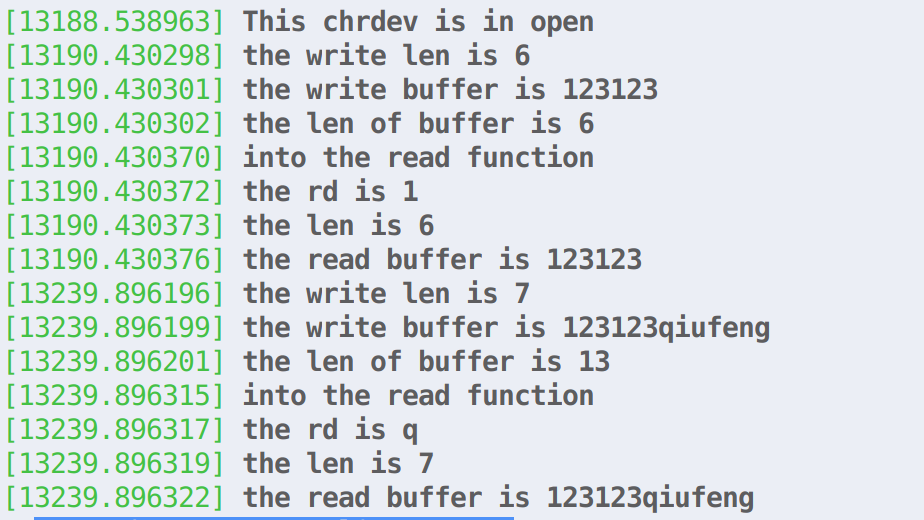
在实际测试中，我们首先向设备中写入数据，如下图



在另一端我们同时进行数据的读取，其结果如下



使用dmesg命令查看日志，可以看到数据被成功的写入和读取



### 5、问题及思考

1. ***Linux内核模块和普通的用户程序有什么区别，需要注意哪些安全问题？***

内核模块与普通程序的差异：

* **运行空间不同**：内核模块是操作系统内核的一部分，运行在内核空间；而应用程序运行在用户空间。
* **出入口不同**：应用程序的入口是main()函数，出口是return；内核模块的入口是init\_module()，出口是cleanup\_module()。
* **库函数不同**：应用程序可以调用C标准库中的函数，如printf()。这些函数被编译到libc中，因此在**链接**时才会和实际的函数地址绑定。因此在内核中无法使用这类的函数。
* **特权级不同**：普通的用户程序处于ring3，内核模块处于ring0。
* **错误危害程度不同**：内核模块如果产生了非法访问等错误，有可能导致整个系统的崩溃，而应用程序出现错误通常只会影响自己。
* **内核支持异步中断、抢占和SMP**：内核及驱动程序在开发时必须时刻注意同步和并发。

需要注意的安全问题：

由于linux可以动态的加载内核模块，并且内核模块具有ring0级别的特权，因此我们必须要保证模块的安全性。如果攻击者恶意加载内核模块，将会使内核变得极其危险。我们可以通过给**内核模块签名**的方式来保证内核只加载正确的模块。

1. ***struct cdev结构体和struct file\_operations结构体中有哪些内容，有什么作用？***

struct cdev结构体**用来描述一个字符设备**，其定义如下(作用见注释)

struct cdev {

    //*内嵌的内核对象*

    struct kobject kobj;

    //*该字符设备所在的内核模块的对象指针*

    struct module \*owner;

    //*该结构描述了字符设备所能实现的方法，是极为关键的一个结构体*

    const struct file\_operations \*ops;

    //*用来将已经向内核注册的所有字符设备形成链表*

    struct list\_head list;

    //*字符设备的设备号，由主设备号和次设备号构成*

    dev\_t dev;

    //*隶属于同一主设备号的次设备号的个数*

    unsigned int count;

};

struct file\_operations结构体绑定了系统调用对应的驱动程序的函数，其内容如下

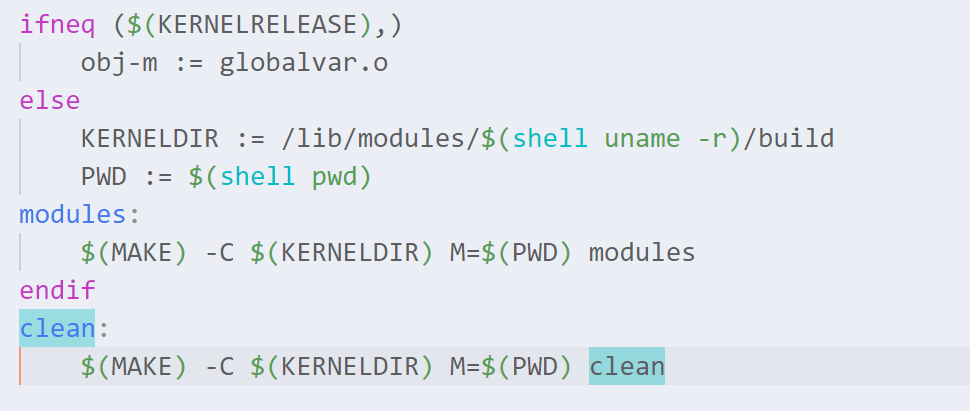


一些字段的解释：

* owner：指向拥有这个结构的模块的指针。
* llseek：改变文件中的当前读/写位置, 并且新位置作为(正的)返回值。
* read：从设备中获取数据。
* aio\_read：异步获取数据。
* write：发送数据给设备。
* aio\_write：异步写数据。
* readdir：读取目录, 仅对文件系统有用。
* poll：返回设备资源的可获取状态。
* ioctl：提供了发出设备特定命令的方法。
* mmap：请求将设备内存映射到进程的地址空间。
* open：打开设备文件。
* flush：在进程关闭它的设备文件描述符的拷贝时调用。
* release：清理未结束的输入输出操作，释放资源，用户自定义排他标志的复位等。
* fsync：刷新待处理的数据,允许进程把所有的脏缓冲区刷新到磁盘。
* aio\_fsync：异步刷新待处理的数据。
* fasync：异步通知设备它的 FASYNC 标志的改变。
* lock：文件加锁。
* readv：发散/汇聚读和写操作。
* sendfile：实现 sendfile 系统调用的读, 使用最少的拷贝从一个文件描述符搬移数据到另一个。
* sendpage：由内核调用来发送数据, 一次一页, 到对应的文件。
* get\_unmapped\_area：在进程的地址空间找一个合适的位置来映射在底层设备上的内存段中。
* check\_flags：允许模块检查传递给 fnctl(F\_SETFL...) 调用的标志。
* dir\_notify：在应用程序使用 fcntl 来请求目录改变通知时调用. 只对文件系统有用。

1. ***用于编译内核模块编译的Makefile文件中各个部分有什么作用？***

Makefile 的内容如下

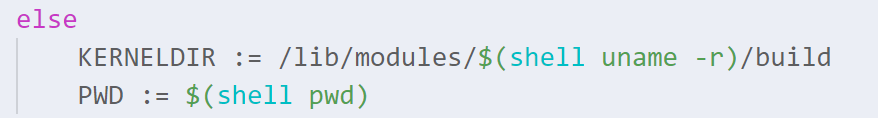


第一部分：



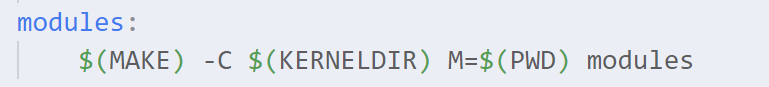
*KERNELRELEASE*是在内核源码的顶层Makefile中定义的一个变量，在第一次读取执行此Makefile时，*KERNELRELEASE*没有被定义，所以make将读取执行else之后的内容。

第二部分：



其中*KERNELDIR*用来标识内核源码所在的目录，*PWD*用来表示当前目录。

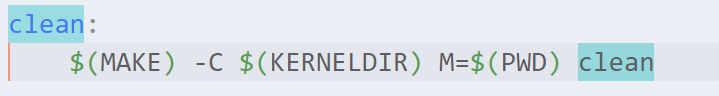
第三部分：



*-C $(KERNELDIR)*表示跳转到内核源码目录下执行Makefile。

*M=$(PWD)*表示返回当前目录继续并执行Makefile。

第四部分：



清除之前编译的可执行文件及配置文件。

1. ***什么是主设备号和次设备号，有什么作用？***

**主设备号和次设备号用来确定具体的设备，其作用可以概括如下：**

* 主设备号：**区分不同种类的设备**。
* 次设备号：**区分同一类型的多种设备**。

1. ***字符设备驱动是如何访问用户空间的数据的？***

**内核空间与用户空间的内存不能直接互访**。这是因为在 Linux 中，**用户内存和内核内存是独立的，在各自的地址空间实现**。因此，我们需要通过调用copy\_from\_user()实现设备驱动对用户空间数据的复制。同理，如果需要将内核空间的内容复制到用户空间，我们需要使用copy\_to\_user()函数。