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) ： 信息安全

学 号 ：

姓 名 ：

任课教师 ： 王鹃

2023 年 10 月 23 日

目录

[实验二 Linux字符设备驱动实验 1](#_Toc150027914)

[1、实验名称 1](#_Toc150027915)

[2、实验目的 1](#_Toc150027916)

[3、实验步骤及内容 1](#_Toc150027917)

[4 实验关键过程及其分析 3](#_Toc150027918)

[4.1 学习Linux内核模块的开发和加载步骤 3](#_Toc150027919)

[4.2 学习字符设备创建的步骤并编写内核模块创建字符设备 4](#_Toc150027920)

[4.3编写一个程序从创建的字符设备中读取数据 5](#_Toc150027921)

[5、问题及思考 8](#_Toc150027922)

[5.1 字符设备驱动是如何访问用户空间的数据的？ 8](#_Toc150027923)

[5.2分析struct cdev和struct file\_operations结构体 10](#_Toc150027924)

[5.3分析用于编译内核模块编译的Makefile文件 12](#_Toc150027925)

[5.4 主/次设备号 13](#_Toc150027926)

[5.5 内核模块与用户程序 13](#_Toc150027927)

# 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：

|  |
| --- |
| #include <linux/module.h>  /\*模块初始化函数,名称可改，通过modue\_init函数注册就可以了\*/  static int module\_init\_function(void) {  /\*初始化操作\*/  return 0;  }  /\*模块卸载函数，名称可改，通过modue\_exit函数注册就可以了\*/  static void module\_exit\_function(void) {  /\*资源释放操作\*/  }  MODULE\_LICENSE("GPL");  module\_init(module\_init\_function);  module\_exit(module\_exit\_function); |

对应的Makefile文件基本格式如下(请不要直接粘贴，因为缩进问题会无法执行成功)

|  |
| --- |
| MODULE\_NAME := 模块的名字（和源代码名字一致）  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：

|  |
| --- |
| linux/fs.h  linux/cdev.h  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。

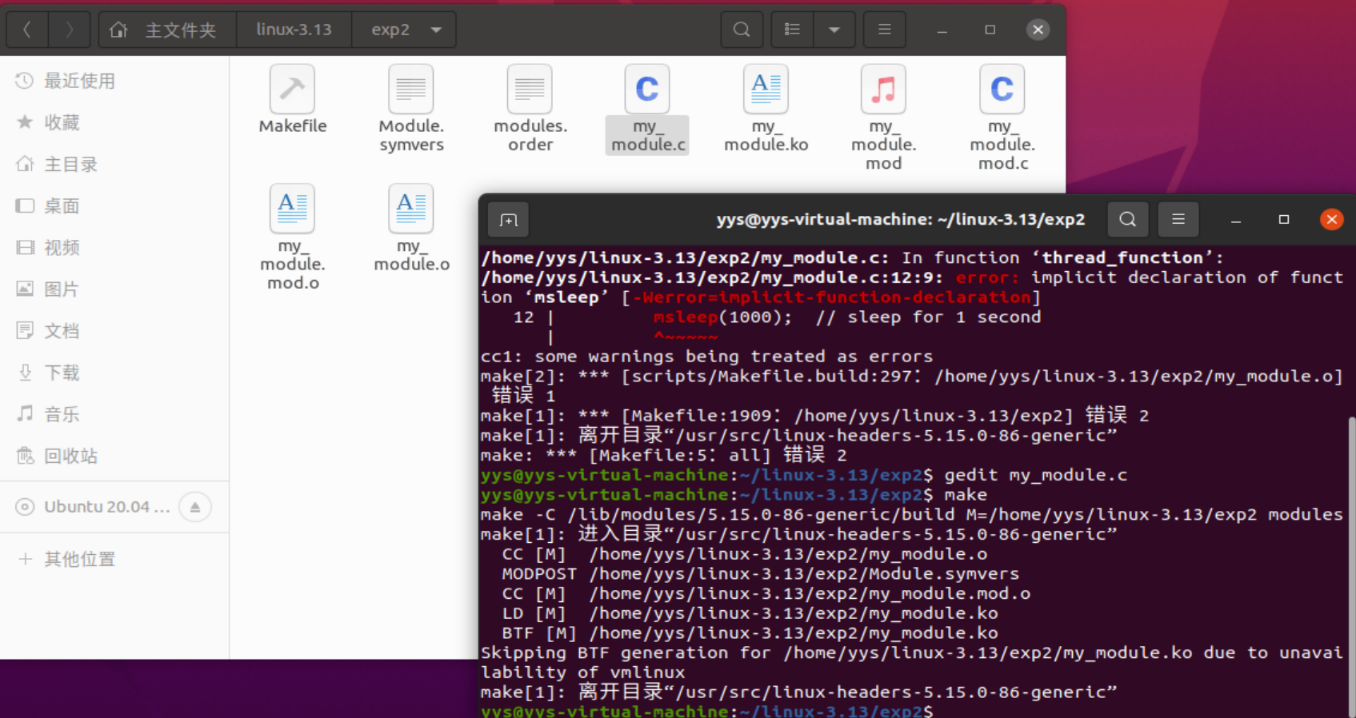
# 4 实验关键过程及其分析

## 4.1 学习Linux内核模块的开发和加载步骤

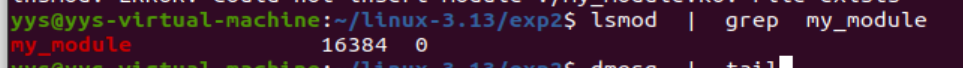
写一个my\_module.c如下：



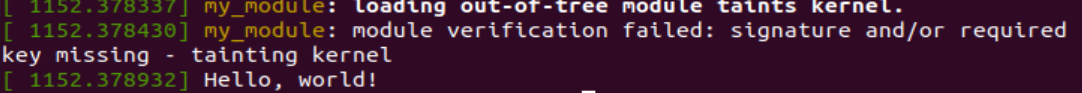
Make



检查是否已经加载



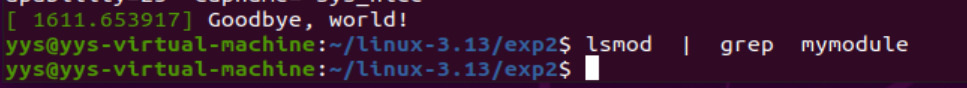
这里已经显示我们编译的内核模块已经加载到当前内核中

使用dmesg查看内核打印信息，一般的新加载的内核模块信息会在最后显示出来

卸载内核模块

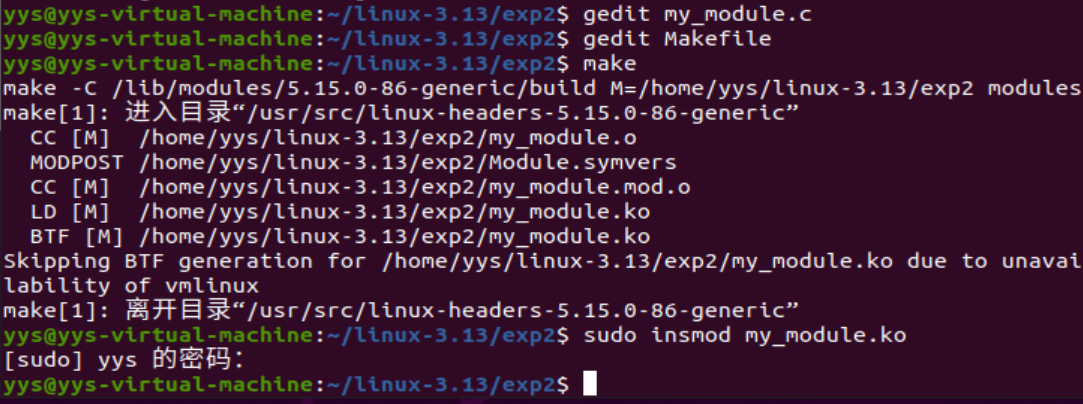


我们在末尾查看到内核模块退出的输出信息



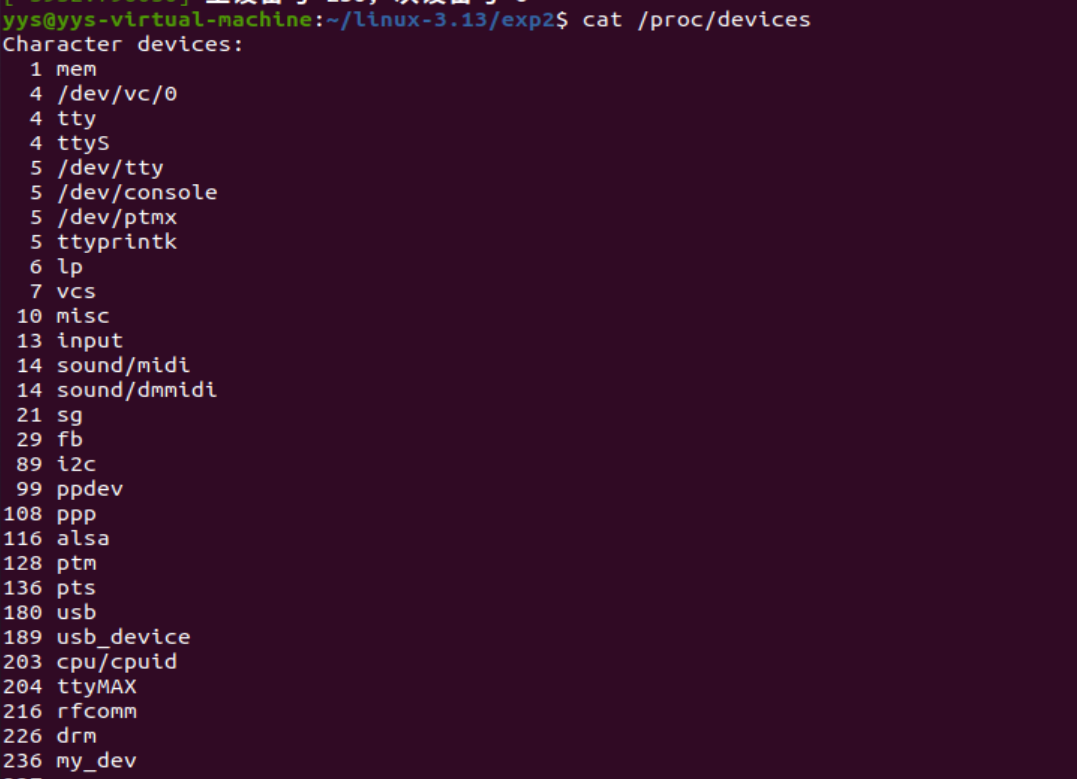
再次检查，显示我们的内核模块已经被卸载

## 4.2 学习字符设备创建的步骤并编写内核模块创建字符设备

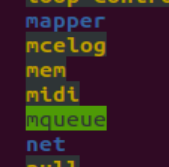


**查看**

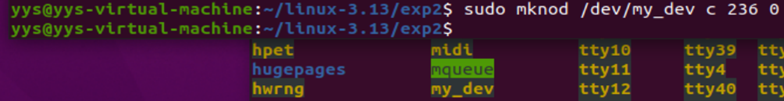
****

****

**我们再查看/dev下**

****

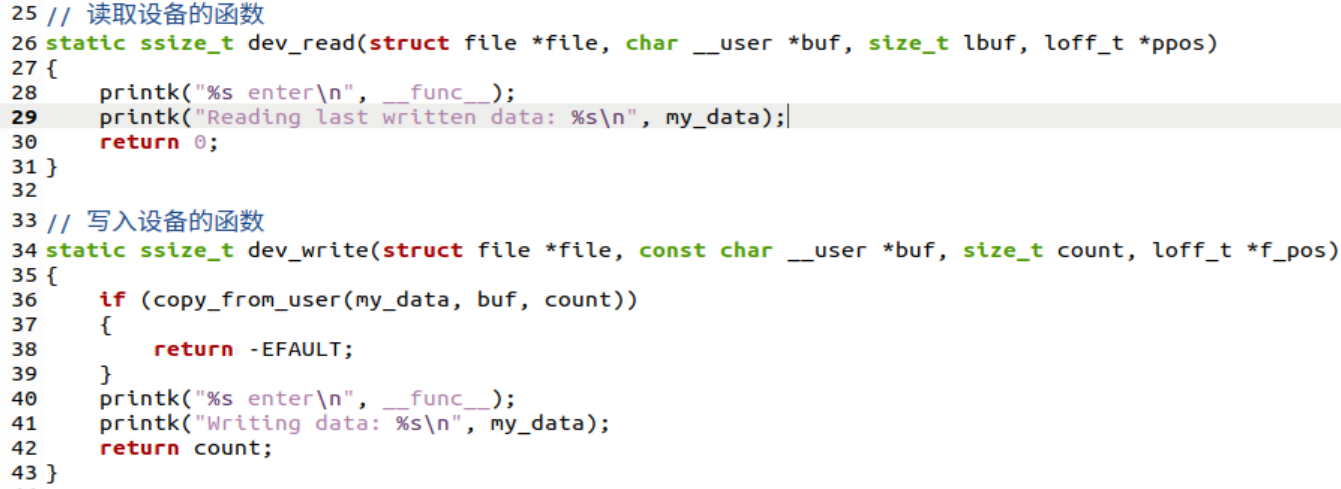
**Mknod一下**



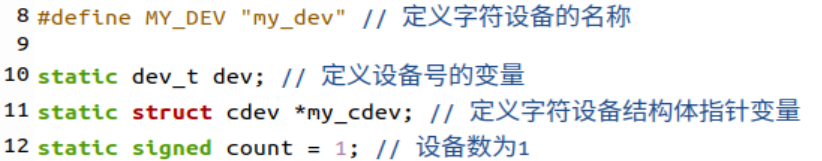
查看到我们的设备出现了

## 4.3编写一个程序从创建的字符设备中读取数据

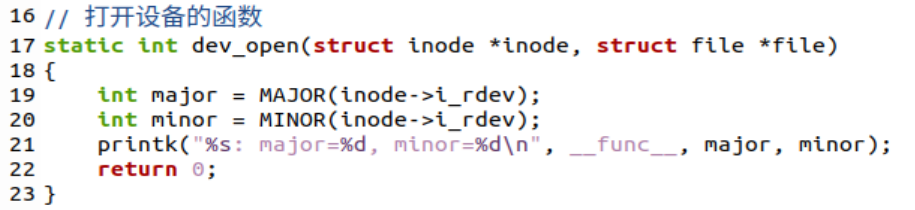
编写my\_module.c实现字符设备的创建及数据的读取和写入



上面的函数实现了数据读写

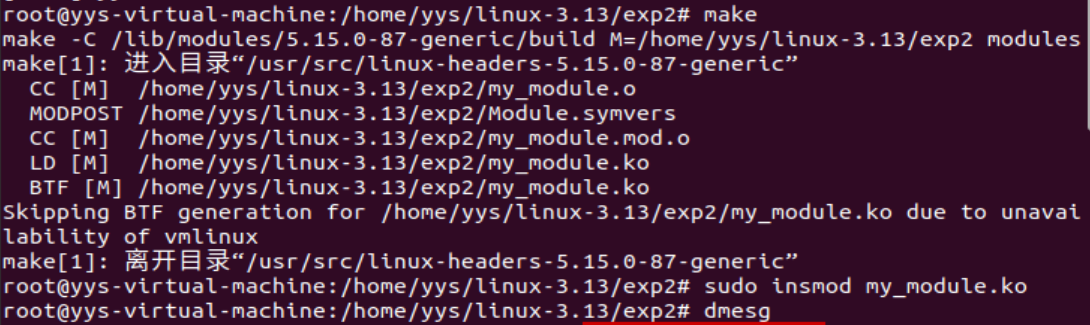


自定义的设备名为my\_dev



上述为设备打开函数的实现

编译模块并插入到内核中

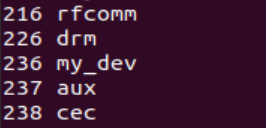


查看日志内容



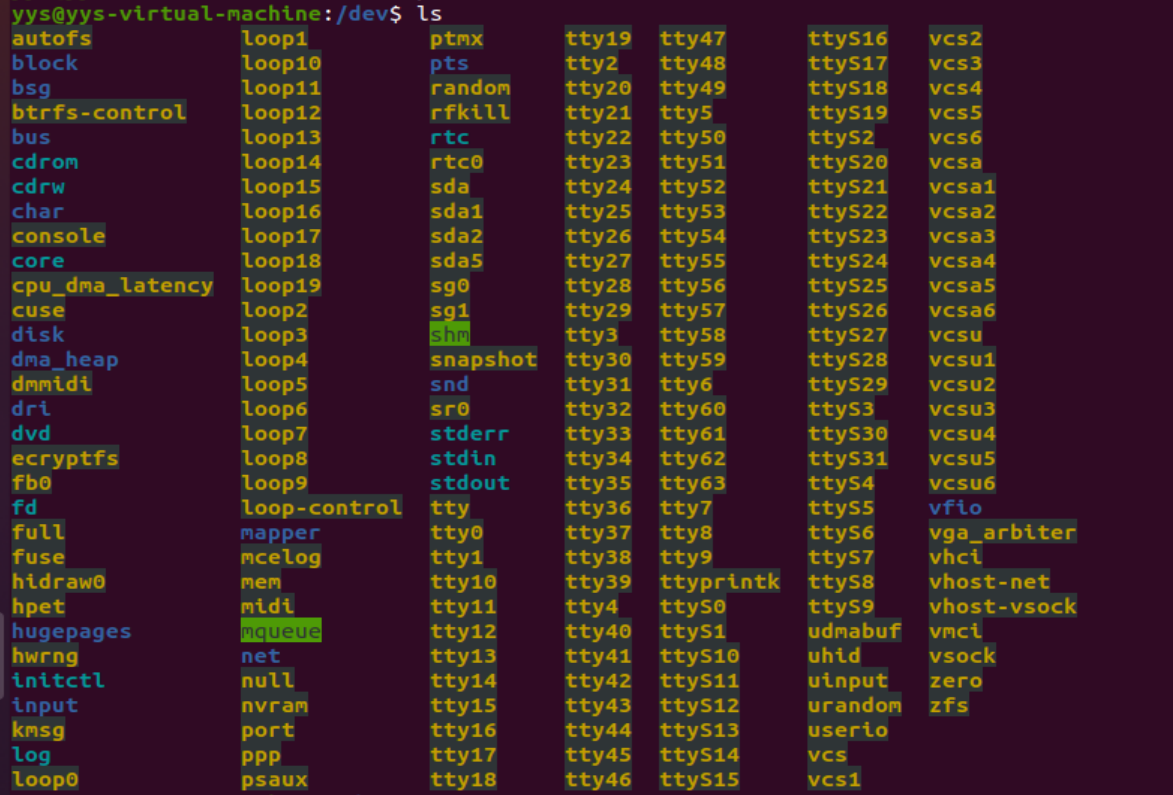
可见我们的设备已经成功创建，并分配了主设备号236及次设备号0

查看/proc/devices文件：

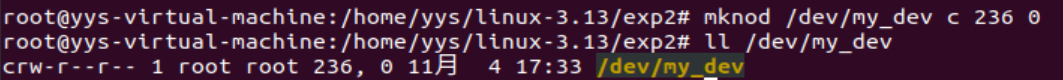


可以看到主设备号236的字符设备就是我们自己创建的设备

我们再查看/dev/目录



发现我们添加的设备并没有被写入其中，那么我们要为此增加我们的设备文件

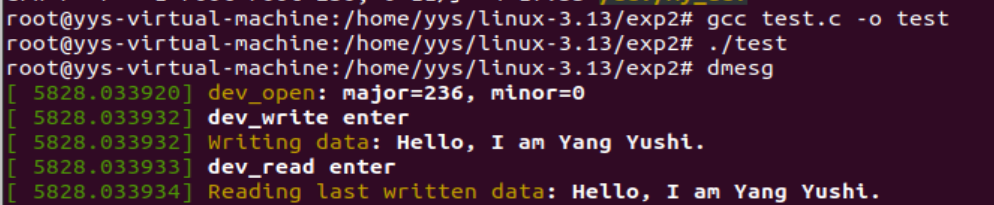


使用mknod命令加入我们的设备，这时在查看/dev/目录下，就有我们的设备了

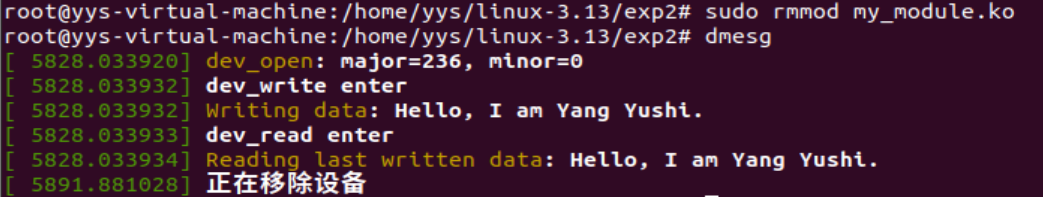
编写的测试程序如下：



我们要把自己定义的字符串写入设备文件，并且读取刚才写入的数据



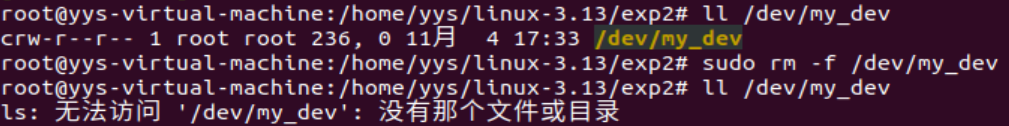
可以看到我们的数据被正确写入，并且能从设备中读取刚才写入的字符串



最后移除模块



可以看到我们的模块被移除



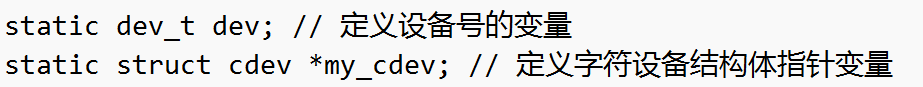
最后不要忘记把我们添加到/dev/目录下的设备移除

# 5、问题及思考

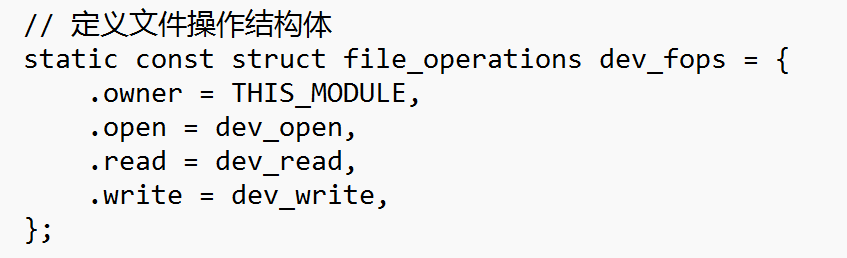
## 5.1 字符设备驱动是如何访问用户空间的数据的？



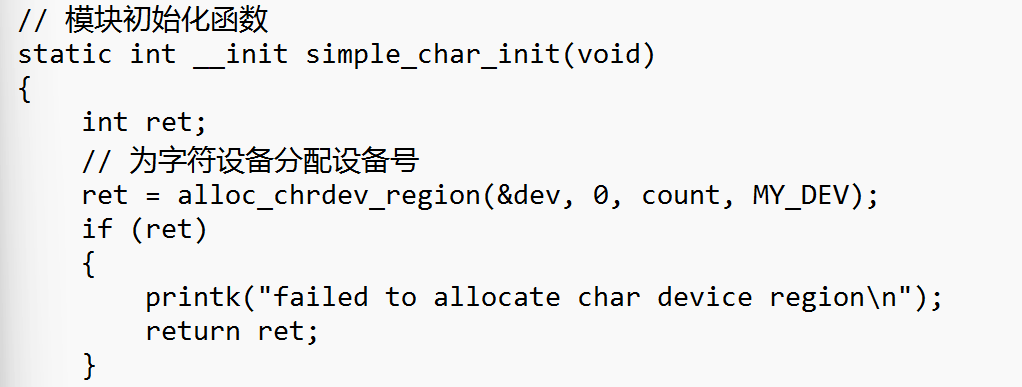
在linux内核中使用cdev结构体来描述字符设备，通过其成员dev\_t来定义设备号，以确定字符设备的唯一性。



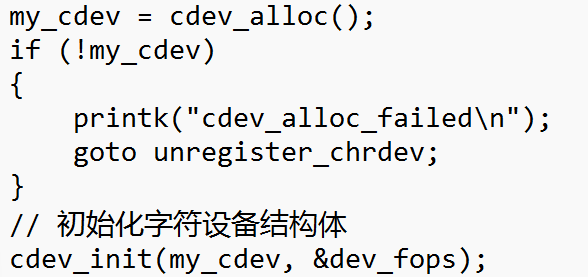
除此之外，cdev结构体还包含了对该字符设备的操作，这些操作被定义在file\_operations成员中。这些操作函数包括了常见的操作，如open()、read()、write()等。通过file\_operations成员，字符设备驱动可以向虚拟文件系统（VFS）提供接口函数，使得用户空间可以通过系统调用来对字符设备进行访问和操作。



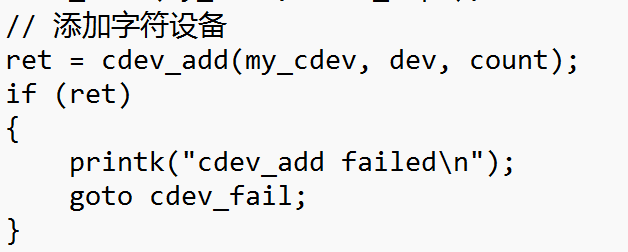
在Linux字符设备驱动中，模块加载函数使用register\_chrdev\_region()或alloc\_chrdev\_region()来获取设备号。这些函数可以在静态或者动态情况下分配设备号。



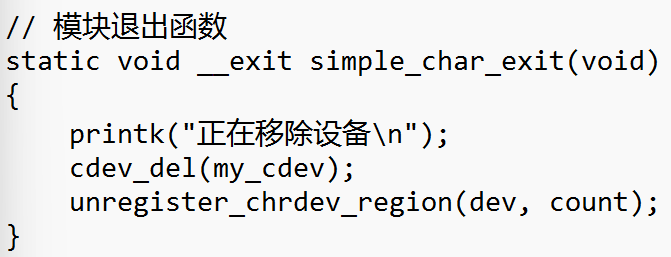
接着，使用cdev\_alloc函数分配一个 cdev 结构，该结构用于描述字符设备驱动程序的行为。然后通过cdev\_init()函数建立cdev结构体与file\_operations之间的连接，确保字符设备的操作函数可以被正确调用。



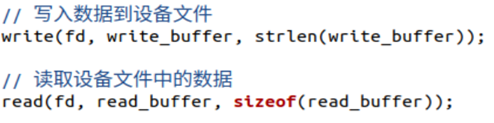
然后，使用cdev\_add()函数将cdev结构体添加到系统中，完成字符设备的注册。



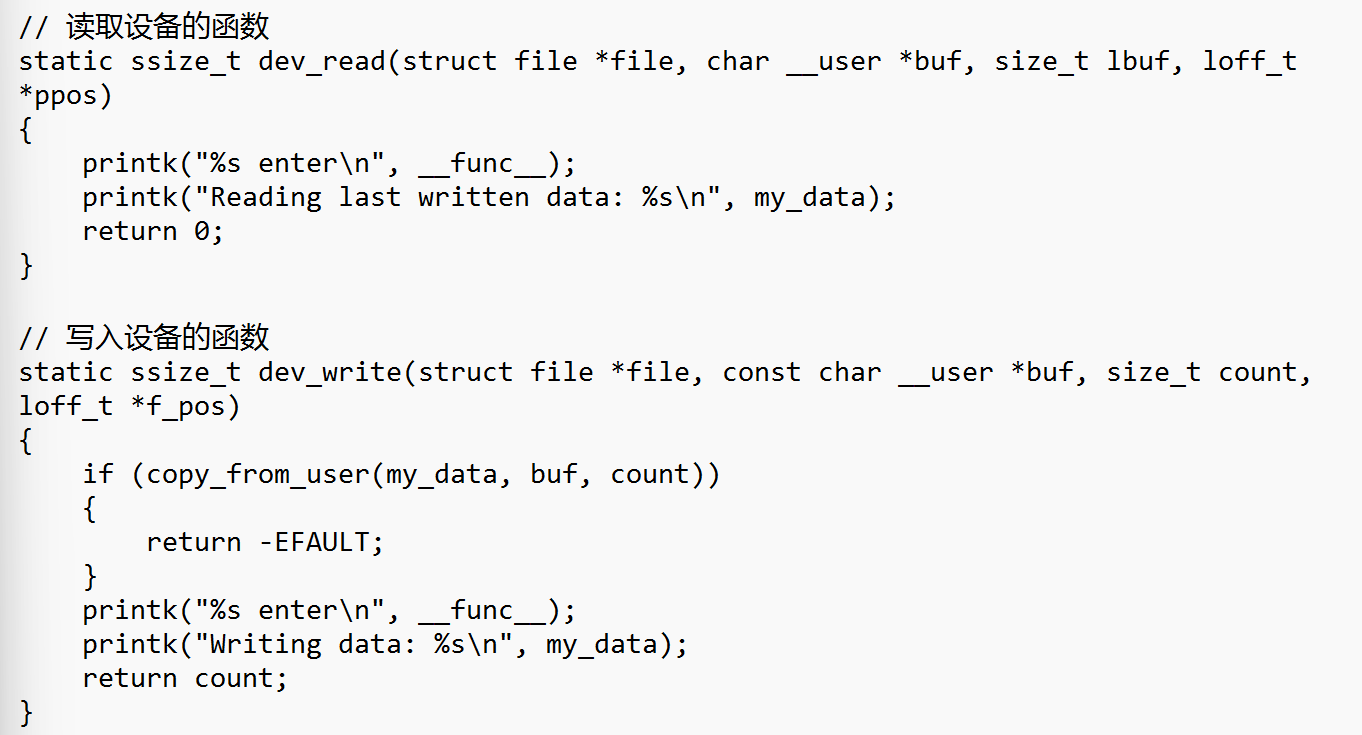
在模块卸载时，可以使用cdev\_del()函数来注销cdev，确保其在系统中被正确移除。同时，使用unregister\_chrdev\_region()函数来释放之前分配的设备号，以保证资源的正确释放。



用户空间的程序可以通过系统调用（例如open()、read()、write()）来访问字符设备。这些系统调用实际上会调用file\_operations结构体中定义的字符设备驱动提供的接口函数。

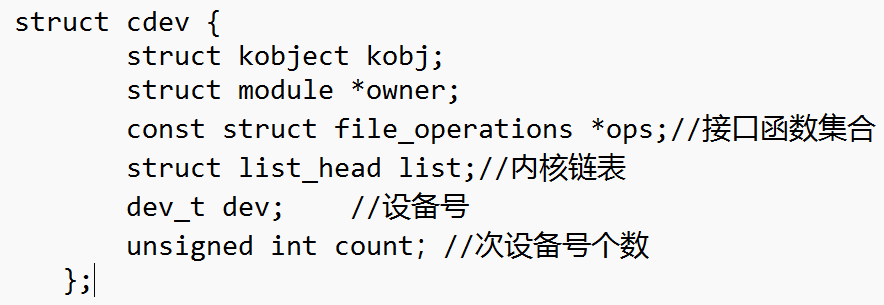


在本实验的代码中，dev\_read()函数用于读取设备的数据，而dev\_write()函数用于向设备中写入数据。为了从用户空间读取数据或向用户空间写入数据，使用了copy\_from\_user()和copy\_to\_user()函数。这两个函数负责将数据从内核空间复制到用户空间或者将数据从用户空间复制到内核空间。



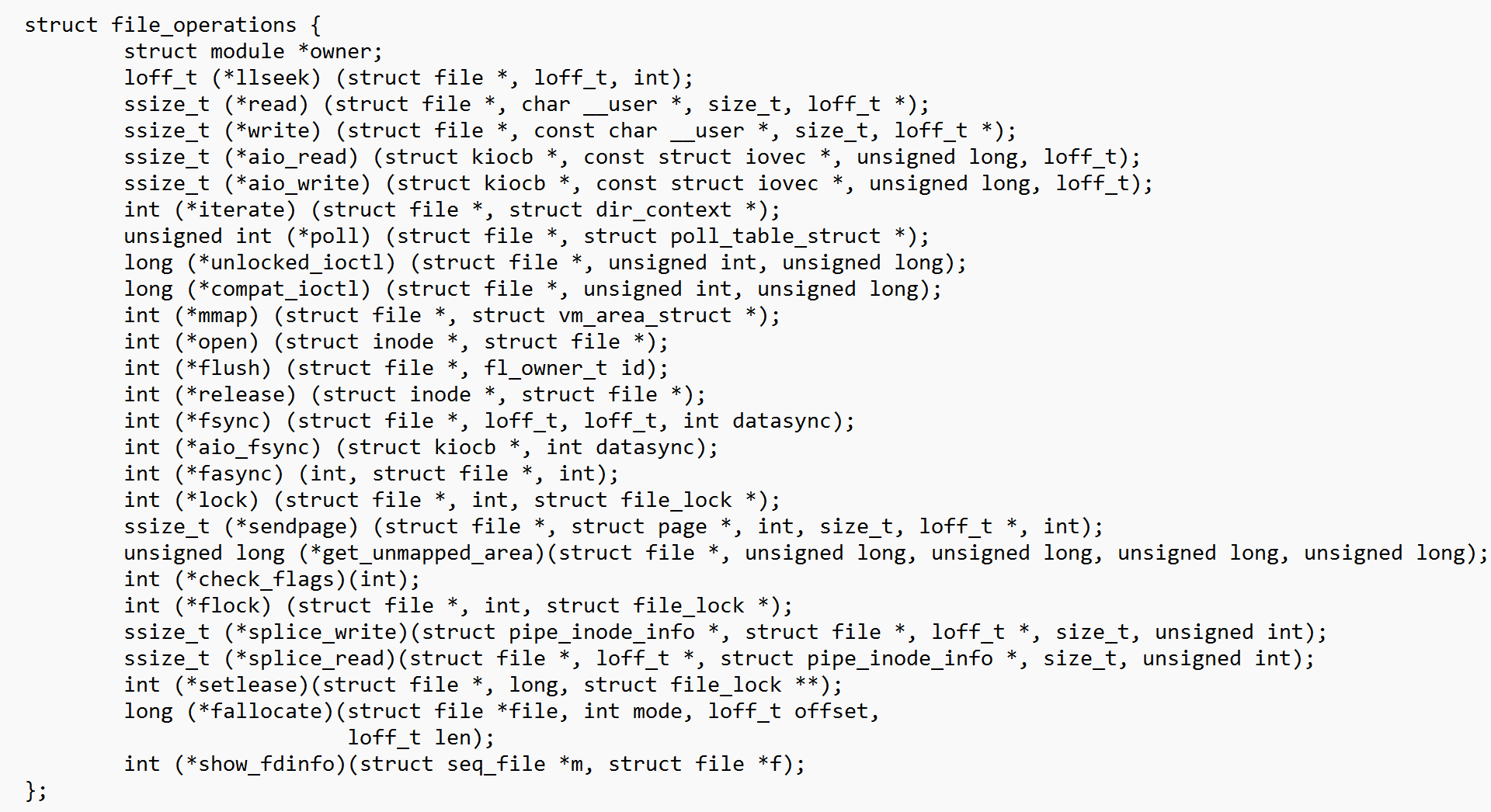
## 5.2分析struct cdev和struct file\_operations结构体

1.struct cdev



|  |  |
| --- | --- |
| 字段 | 描述 |
| kobj | 用于表示内核对象。 |
| owner | 指向拥有此结构的模块的指针。 |
| ops | 用于指向字符设备驱动提供给VFS的接口函数集合。 |
| list | 用于表示内核链表中的链表头。 |
| dev | 用于表示设备号。 |
| count | 用于表示次设备号的个数。 |

2.struct file\_operations



对结构体分析如下

|  |  |
| --- | --- |
| 字段 | 描述 |
| owner | 指向拥有此结构的模块的指针。 |
| llseek | 用于执行文件定位操作。 |
| read | 用于从设备中读取数据。 |
| write | 用于向设备中写入数据。 |
| aio\_read | 用于执行异步I/O读取操作。 |
| aio\_write | 用于执行异步I/O写入操作。 |
| iterate | 用于遍历目录文件。 |
| poll | 用于检查设备是否可读或可写。 |
| unlocked\_ioctl | 用于执行未加锁的输入输出控制操作。 |
| compat\_ioctl | 用于执行兼容输入输出控制操作。 |
| mmap | 用于映射设备的内存。 |
| open | 用于打开设备文件。 |
| flush | 用于刷新设备缓冲区。 |
| release | 用于释放设备资源。 |
| fsync | 用于将设备上的文件系统数据同步到存储设备。 |
| aio\_fsync | 用于执行异步文件同步操作。 |
| fasync | 用于对异步通知操作进行设置。 |
| lock | 用于对文件进行加锁。 |
| sendpage | 用于向另一个文件传输页面。 |
| get\_unmapped\_area | 用于获取未映射区域。 |
| check\_flags | 用于检查标志。 |
| flock | 用于对文件进行加锁。 |
| splice\_write | 用于在管道中进行写操作。 |
| splice\_read | 用于在管道中进行读操作。 |
| setlease | 用于设置文件租约。 |
| fallocate | 用于分配文件空间。 |
| show\_fdinfo | 用于显示文件描述符信息。 |

## 5.3分析用于编译内核模块编译的Makefile文件

Makefile里主要包含了五个东西：显式规则、隐晦规则、变量定义、文件指示和注释。

1.显式规则。显式规则说明了，如何生成一个或多的的目标文件。这是由Makefile的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。

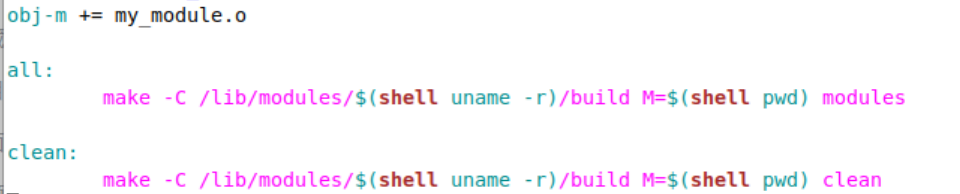
2.隐晦规则。由于我们的make有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写Makefile，这是由make所支持的。

3.变量的定义。在Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点你C语言中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。

4.文件指示。其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。

5.注释。Makefile中只有行注释，和UNIX的Shell脚本一样，其注释是用“#”字符，这个就像C/C++中的“//”一样。如果你要在你的Makefile中使用“#”字符，可以用反斜框进行转义，如：“\#”。

本实验中的Makefile文件如下：



1. 变量定义： obj-m += my\_module.o，指定要编译为模块的源文件，my\_module.c是指定模块的源文件名。

2. 显式规则：all和clean部分，定义了编译和清理的规则。

3. 文件指示：make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules和make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean，这些指示告诉make如何执行编译和清理操作。

## 5.4 主/次设备号

在Linux系统中，设备文件被用于与设备进行通信。每个设备文件都有一个主设备号和一个次设备号。主设备号用于标识设备驱动程序，而次设备号用于标识特定的设备实例。主设备号指定了哪个驱动程序将处理设备的请求，而次设备号则允许内核区分同一驱动程序下的不同设备。

主设备号：主设备号用于识别设备所使用的驱动程序。内核使用主设备号来确定要调用哪个驱动程序来处理设备的请求。

次设备号：次设备号用于标识特定的设备实例。它允许内核在同一驱动程序下区分多个设备实例。

对于同一类设备，它们通常会共享一个主设备号，但拥有不同的次设备号来区分各自的实例。这种机制使得内核能够根据设备号确定设备的驱动程序，并将设备文件连接到相应的驱动程序。这种机制允许内核将设备访问抽象为文件访问，使得用户空间应用程序可以像访问文件一样访问设备。

## 5.5 内核模块与用户程序

1.内核模块和应用程序运行在不同的空间，分别是内核空间和用户空间。这种区分是操作系统设计的基础，旨在确保系统的安全性和稳定性。

2.内核模块在操作系统内核空间运行，因此具有更高的特权级别。这意味着它们可以直接访问系统的硬件和敏感数据，并且具有更高的系统权限。相比之下，普通用户程序在用户空间运行，其权限受到操作系统的严格限制。内核模块可以直接访问和修改操作系统内核数据结构和函数，因此需要小心处理以免对系统的稳定性和安全性造成危害，确保只有受信任的代码可以访问特权操作。

3.内核模块是扩展内核功能的一种方式，可以为系统添加新的驱动程序或功能。它负责处理系统调用以及中断处理等任务。内核模块的编程与应用程序编程有一些显著区别，其中最重要的是并发问题。由于内核模块运行在多任务的环境中，开发者需要仔细考虑并发访问共享内存可能带来的问题，以避免系统不稳定甚至错误的发生。

4.内核空间的堆栈区通常比用户空间的堆栈区要小得多。因此，在内核编程或编写驱动程序时，需要注意避免声明过多的大型自动变量和递归调用等可能占用过多堆栈空间的操作，以防止堆栈溢出导致的系统崩溃。