**字符设备驱动与添加系统调用实验实验**

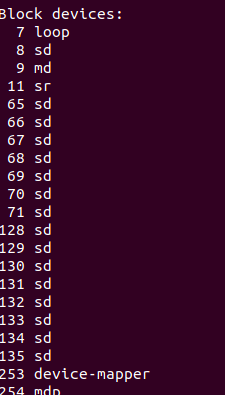
1. **实验目的**

* 通过一个简单的设备驱动的实现过程，学会Linux中设备驱动程序的编写
* 通过实现一个简单的系统调用函数，加深对Linux系统调用原理的理解

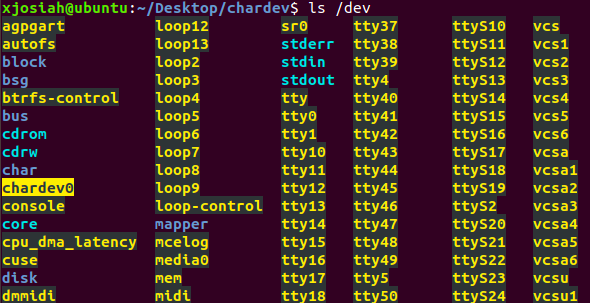
1. **实验环境**
2. macOS Mojave 10.14.1 (x86\_64-apple-darwin18.2.0)  
    Apple LLVM version 10.0.1 (clang-1001.0.46.4)
3. Ubuntu 18.04 (Linux 5.4.0-53-generic) x64  
    gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)   
    GNU Make 4.1
4. **实验内容**
5. 编写一个字符设备驱动，并利用对字符设备的同步操作，设计实现一个聊天程序。可以有一个读、一个写进程共享该字符设备，进行聊天；也可以由多个读程序和写程序共享该字符设备进行聊天。
6. 添加一个系统调用，编写一个用户程序来调用该系统调用，并通过系统调用实现一些功能，要做得可以分辨调用是否已经成功，可以采用编译内核的方法或者编写内核模块的方法来实现系统调用的服务例程。
7. **实验原理 实验中用到的系统调用函数（包括实验原理中介绍的和自己采用的），实验步骤，**
8. 结构体file\_operations在头文件linux/fs.h中定义，用来存取驱动内核模块提供的对设备进行各种操作的函数的指针，该结构体的每个域都对应着驱动内核模块用来处理某个被请求的事务的函数的地址，设备“globalvar”的基本入口点结构变量是“globalvar\_fops”
9. 注册字符设备函数int register\_chrdev(unsigned int major, unsigned int baseminor, unsigned int count, const char \*name, const struct file\_operations \*fos)中major参数是被请求的主设备号，name是设备的名称，fops是指向函数指针数组的指针，这些函数是调用驱动程序的入口点
10. MKDEV()函数，其宏定义为  
    #define MKDEV(major,minor)（（（major）<<MINORBITS）|(minor)）  
    成功执行返回dev\_t类型的设备编号，dev\_t类型的是unsigned int 类型，32位，用于在驱动程序中定义设备号，高12位是主设备号，低20位为次设备号，可以通过MAJO和MINOR来获得主设备号和次设备号，在module\_init宏调用的函数中去注册字符设备驱动。
11. 添加系统调用的步骤：添加系统调用号，系统调用号在unistd.h文件中被定义，以\_\_NR\_\_开头，在系统调用表中添加自己的系统调用服务例程sys\_mysyscall，把sys\_mysyscall加到kernel目录下的系统调用文件sys.c中，重新编译内核并编写用户程序来调用系统调用
12. **实验结果分析（截屏的实验结果，与实验结果对应的实验分析）**

**1.编写一个字符设备驱动，并利用对字符设备的同步操作**

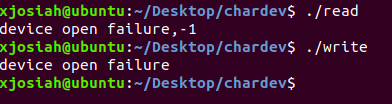
先查看系统中设备号的使用情况，选择一个空闲设备号，实验中选择了256号：



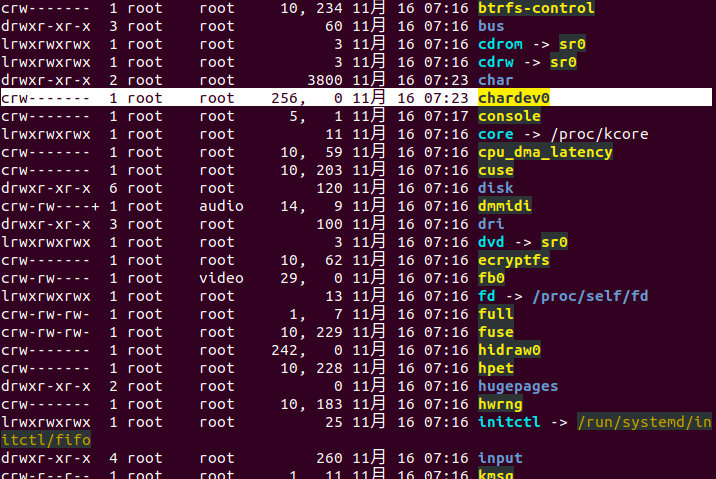
编写内核模块并编译，使用ls /dev 查找当前系统中所有设备的信息，找到我们装入模块后创建出的虚拟字符设备chardev0是否已经存在：

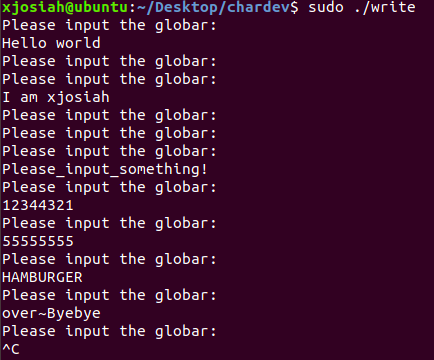


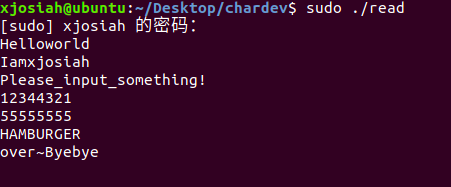
编译读写程序，程序中用open（）函数像打开一个文件一样打开这个字符设备，并利用内核模块中实现的读写命令来对字符进行操作：



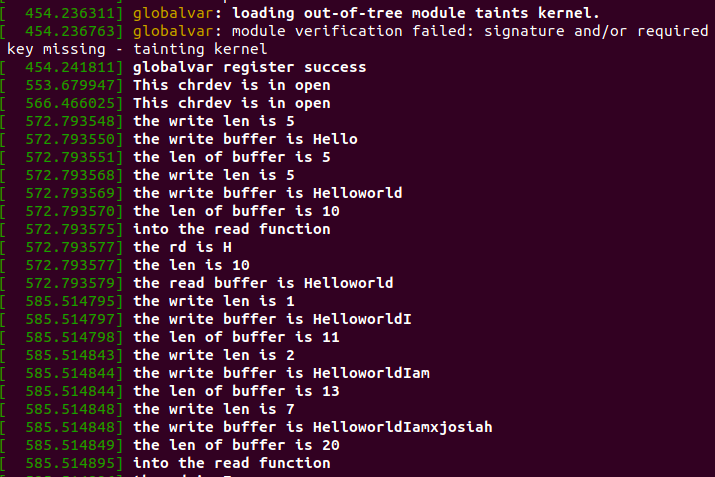
这里出现无法打开程序的情况，但内核和相关系统信息都表明了字符设备已经被成功注册了，于是回顾注册时使用的权限符并使用ls -l /dev 命令来查看chardev0的操作权限，发现该设备的操作权限是700，即必须使用系统管理员身份才可以正常读写执行操作该设备，于是使用sudo ./ 程序成功执行：

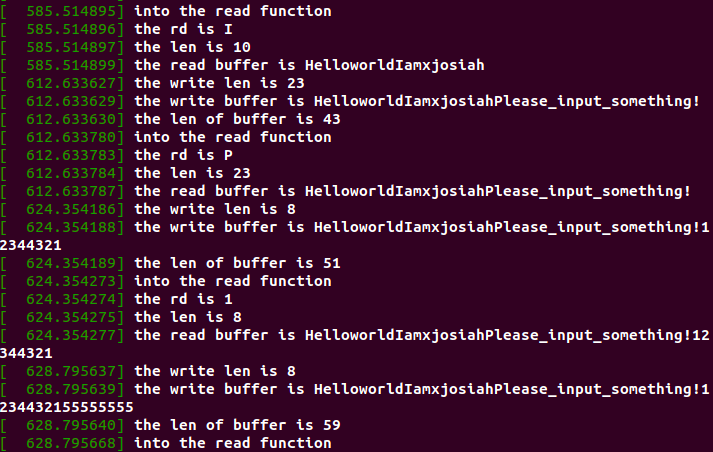




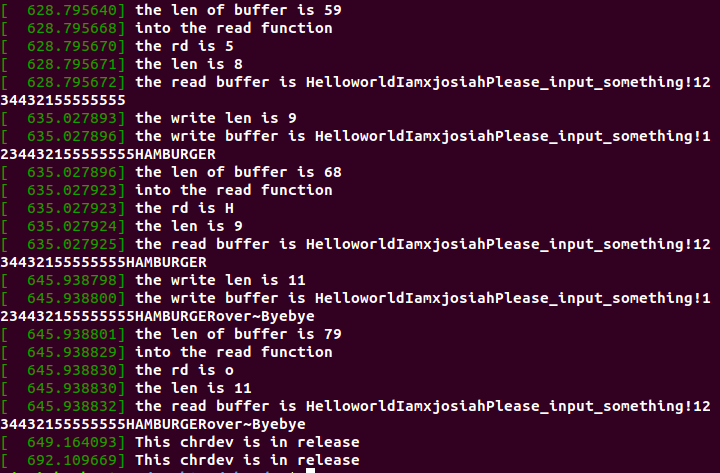


最后使用dmesg命令查看内核模块在系统中输出的信息：





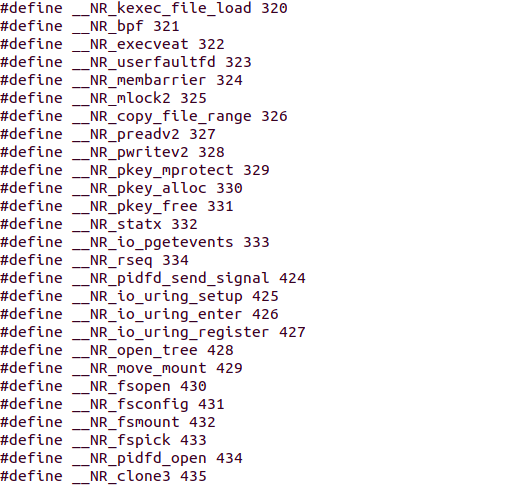
卸载内核模块后，再次查看系统中的信息



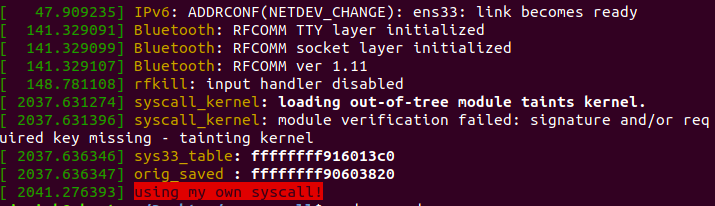
可以发现运行结果和预期一致。

**2.添加一个自己的系统调用**

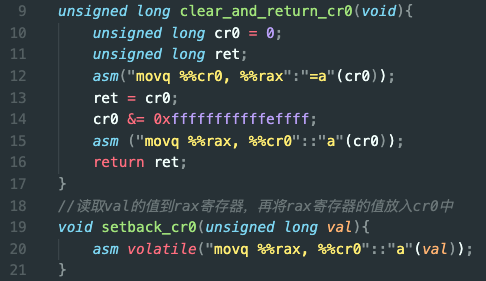
查看系统中的系统调用的使用情况，选择一个空闲的系统调用号来使用，本次实验程序中使用了335号系统调用



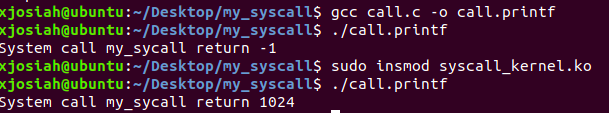
使用sudo cat /proc/kallsyms | grep sys\_call\_table 来查看系统调用表的地址，方便我们后面编写内核模块来引入系统调用，在本台机器中的地址为：0xffffffff916013c0



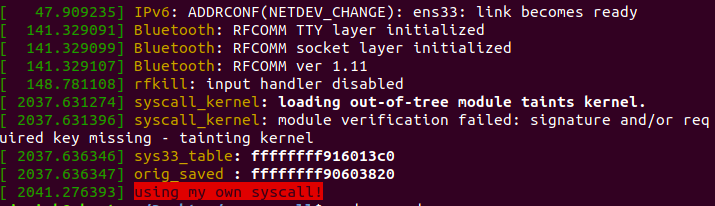
编写内核模块，在编写系统调用服务例程的时候，将系统调用函数的返回值设为1024，这样我们在编写用户程序的时候，可以通过系统调用访问的值查看到调用是否成功，同时使用内嵌汇编代码时要注意和本机相结合，64位机使用的寄存器是rax而32位机使用的寄存器是eax，mov的用法也有一些差别: movb（8位）、movw（16位）、movl（32位）、movq（64位）



在未装入内核模块前，编写用户进程，使用系统调用，并观察程序的输出结果，装入内核模块，再次调用用户进程，观察程序的输出结果，对比验证是否正常使用了系统调用



卸载内核模块，使用dmesg命令查看系统中内核模块输出的信息



程序和系统调用都执行正常，达到了预期的效果

1. **实验总结**
2. 使用相关函数前必须认证检查程序要使用的头文件是否引用正确，本次实验中，使用copy\_from\_user等函数时，由于用错了头文件，找不到相关函数导致gcc编译异常，我以为是函数由于内核版本不同而发生变动，于是根据gcc的警告提醒改用了raw\_copy\_from\_user()函数，结果在编译内核的时候发送了严重的错误，后来查看相关资料才发现copy\_from\_user（）是由linux/uaccess.h定义的，而不是asm/uaccess.h
3. 在编写虚拟字符设备驱动的时候，由于使用的是内存区的映射，并没有真正创建一个物理设备，因此在写相关驱动的时候只需要进行简单操作就可以了，但应当注意的一点在退出的模块的时候要对设备进行卸载，具体操作为退出设备，使用卸载函数int unregister\_chrdev(unsigned int major, const char \*name)
4. 查看系统调用号的方法在Linux内核版本为Linux 5.4.0-XX-generic下的命令为sudo cat /usr/src/linux-headers-5.4.0-XX-generic/arch/x86/include  
   /generated/uapi/asm/unistd\_64.h 而查看当前使用的内核版本号的命令是 uname -r 我们还可以通过find()函数来查找系统中所有unistd\_64.h的文件存放位置，具体命令为：sudo find / -name unistd\_64.h
5. 在实验的最后我要感谢这8个星期来老师的细心教导，我在这门实验课中得到了很多实践的机会，让我对Linux内核中的更多细节了解得更深入了，对我而言这是一门很有意义的课程，我很开心能和老师共同度过这门课，在此，我再次对老师表达我的感谢！