***A close up of a logo

Description automatically generated***

**PROJECT 1: RESEARCHING AND PROGRAMMING LINUX KERNEL MODULE**

**Phạm Nguyễn Huy Hoàng – 1751068**

**Nguyễn Trường Giang – 1751065**

|  |
| --- |
|  |

1. **Linux model kernel**

Linux kernel module is a file with extension name called (.ko). It can be used to plugin or unplug out of kernel when necessary. Linux kernel module has another name called that loadable kernel module.

Driver is the most use loadable kernel module.

Driver using loadable module has 3 advantage:

* Decrease the size of kernel. Because of this it can decrease the waste memory and system boot time.
* Don’t need to retranslate the kernel when a new driver add-on or when change another driver.
* Don’t need to restart system when add a new driver. In windows, when you install a new driver, you need to restart the system, this is not suitable for a sever.

When a module needs to be used but it’s not in the kernel space, the kernel will put it in. This process can be done automatically:

* Step 1: kernel activate the modprobe process with the variable is the name of the module (example: abc.ko).
* Step 2: Modprobe process check file “/lib/modules/<kernel version>/modules.dep to see if abc.ko dependence to any other module. Example the abc.ko dependent to aaa.ko
* Step 3: modprope process will activate the insmode to put all the dependent module in first, then it will add-in the necessary module.

There are 2 way to activate the modprobe process:

* Using kmod. This a part of Linux kernel, it works in kernel space. When any part of the kernel needs to put in a module into the kernel space, it will transmit the name of module into request\_module of kmod. Request\_module will use call\_usermodehelper\_setup to create a modprobe process.
* Using udevd. This a process work in the user space. If a device plugin computer system, the PCI bus or USB bus will change, and controller will know this. Bus driver will send a call to udevd. This call has information about the device. Udevd will check file /lib/modules/<kernel-version>/modules.alias to file the drive that compatible with the device. Then udevd create modprobe process.

1. **Character driver**

Like other device driver, the character driver (char driver) has 2 part:

* OS specific has these functions:
  + Create function. This function responsible for:
    - Request kernel to allocate device number.
    - Request kernel to create a device file.
    - Request kernel to allocate memory for the data structure of the driver of initialize them.
    - Request initialize a physical device.
    - Register entry point function to kernel.
    - Register stop function.
  + Stop function: This function does opposite thing that the create function has done.
  + Entry point function. Example: open(), release(), read(), …
* Device specific part has these functions:
  + Group of function create/release device
  + Group of function read/write into the memory of the device.
    - Read/write data memory.
    - Get the information in the memory
    - Setup the command for control memory
  + Group of stop function.

We create vchar\_driver.c. This file contains source code to initial driver for a virtual character device. File vchar\_driver.c divided into 2 part: OS specific and device specific:

* + OS specific content initialize function, stop function and entry point for driver.
  + Device specific content create/release function for physical device, read/write memory and handle stop signal from physical device

A screenshot of text

Description automatically generated

We create 2 file Makefile and Kbuild to make thing easier for the translation:

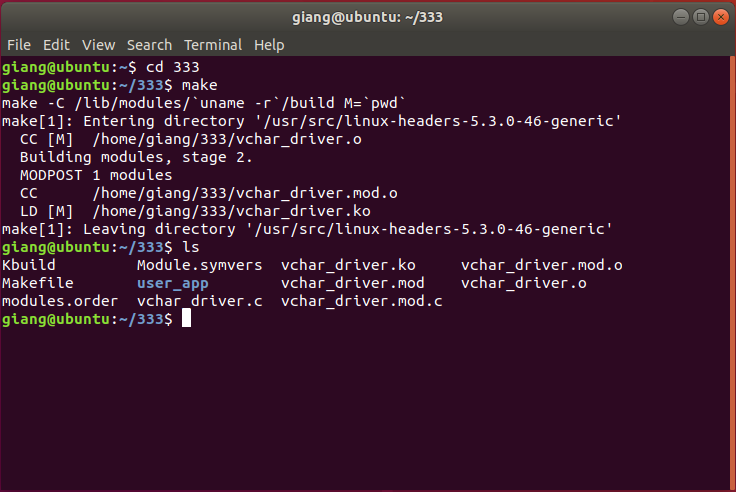
A picture containing drawing

Description automatically generated

A picture containing drawing

Description automatically generated

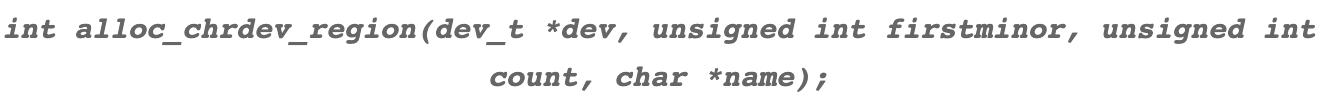
And here is the result



1. **Dynamic allocated device number**

Normally, programmer will create a character driver on a computer and that char driver will use in many other computers. If we use the static allocation device number, that device may use in other computer, this can lead to the char driver can’t be used in that computer. To solve this problem, we use the dynamic allocation device number.

In this method, Linux kernel will provide a function called alloc\_chrdev\_region. This function will find a value that could be used for device number. We used to call this function in the create function of char driver.



***dev***is an output-only parameter that will, on successful completion, hold the first number in your allocated range.

***firstminor***should be the requested first minor number to use; it is usually 0.

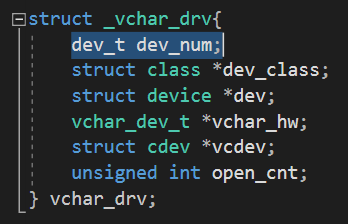
***count***is the total number of contiguous device numbers you are requesting.

***name***is the name of the device that should be associated with this number range; it will appear in */proc/devices* and *sysfs*.

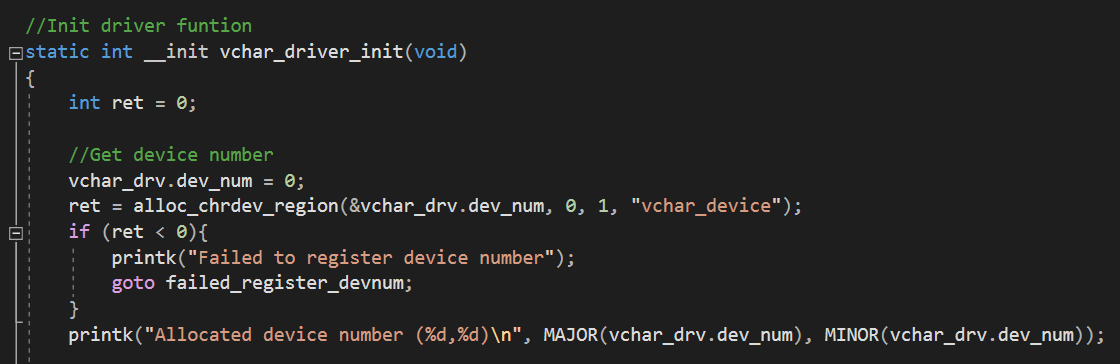
To use the function to init/release device number, we need to add <linux/fs.h> library



To save the value of device number, we create a vchar\_drv structure that stores dev\_num

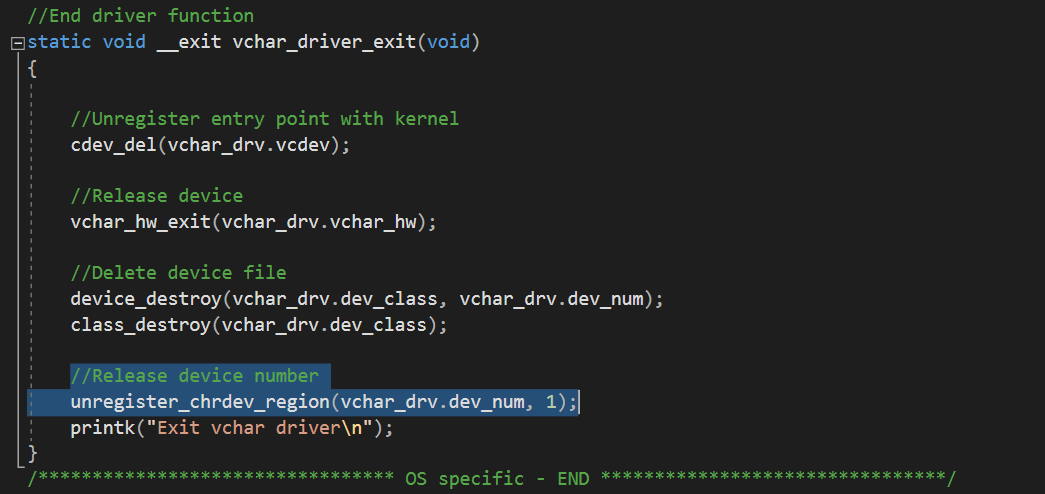


In the vchar\_driver\_ninit function of this driver, we call alloc\_chrdev\_region to request kernel to find a suitable device number. If there is no suitable device number, this function returns a negative value. So that, we return a negative number to inform Linux kernel that this char driver can’t put into kernel.

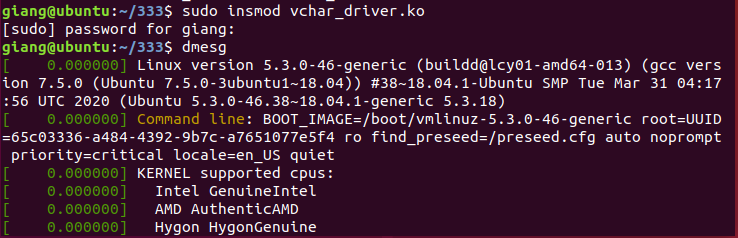




Finally, in the vchar\_driver\_exit of this driver, we call unregister\_chrdev\_region function to release device number.



After loading module to kernel, we get the device number by using *dmesg.*





1. **Creating device file**

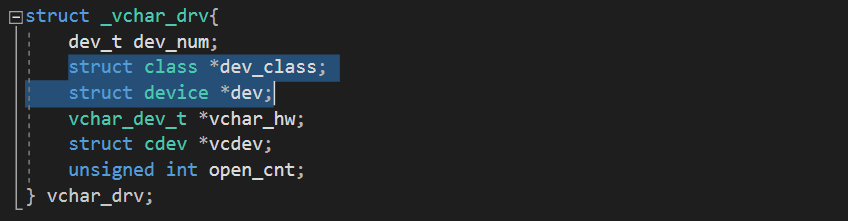
The automatic creation of device files can be handled with udev. Udev is the device manager for the Linux kernel that creates/removes device nodes in the /dev directory dynamically. Just follow the below steps.

1. Include the library **linux/device.h**, which contains library to create device file.
2. Create the struct Class
3. Create Device with the class which is created by the above step

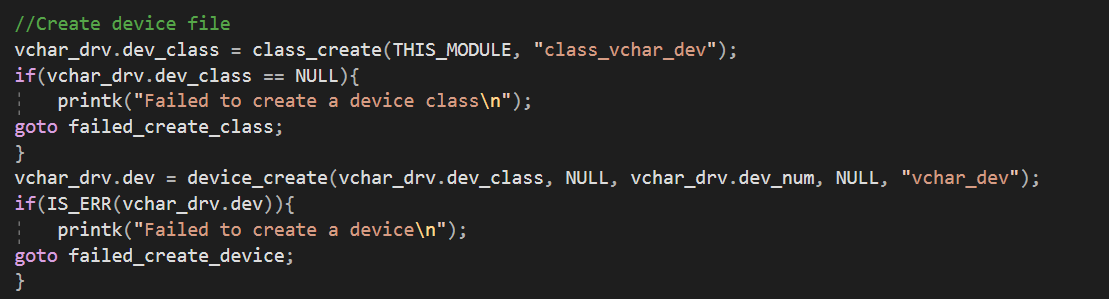
To create a device file automatically, we need to include the <linux/device.h>

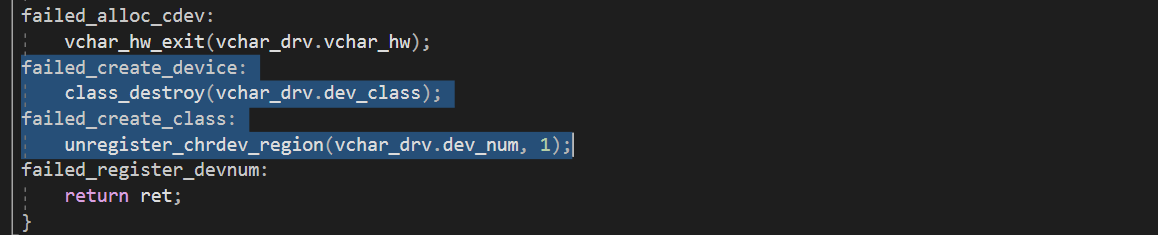


To store the return value of functions class\_create and device\_create, we need to add \*dev\_class and \*dev to the vchar\_drv struct.

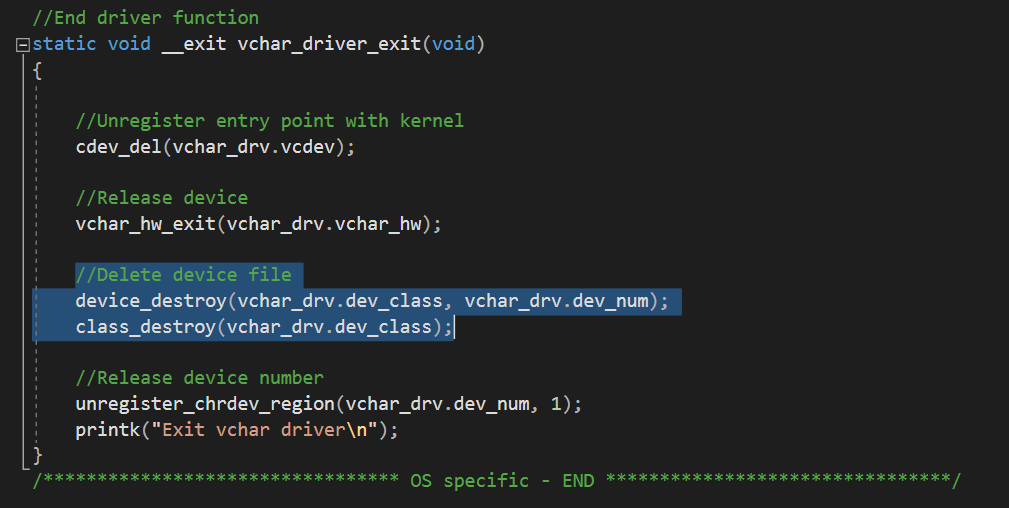


In the vchar\_driver\_init function, we call class\_create to create a device class name class\_vchar\_dev. If this function can’t be done successfully, we need to call unregister\_chrdev\_region function to release device number which has been initialized before. If success, we need to continue to call device\_create to create a device name vchar\_dev in the folder /dev. If device\_create is failed, we will call class\_destroy to release the device class.

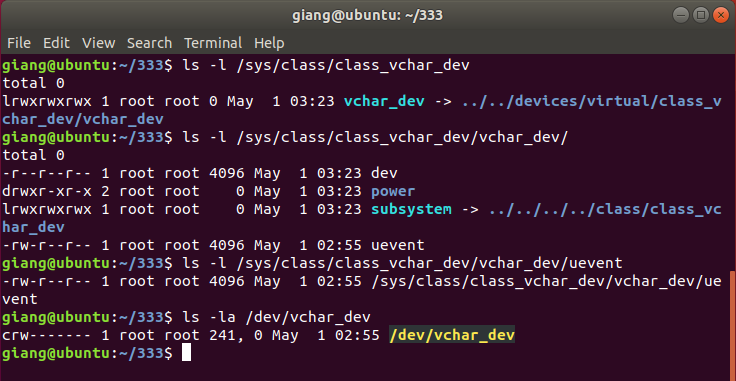




Finally, in function vchar\_driver\_exit, we call functions device\_destroy and class\_destroy to release what have been created.



While initializing character driver, function class\_create will creates folder class\_vchar\_dev in folder /sys/class, and function device\_create will create folder vchar\_dev in folder /class\_vchar\_dev. Inside folder vchar\_dev there is a file name uevent. Uevent store datas that character driver sends to process udevd. Using theses data, udevd will create a device file vchar\_dev in folder /dev.



1. **Register entry points open and release.**

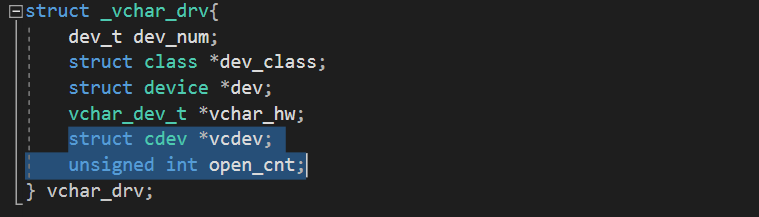
* **cdev structure**

In Linux kernel **struct inode** structure is used to represent files. Therefore, it is different from the file structure that represents an open file descriptor. There can be numerous file structures representing multiple open descriptors on a single file, but they all point to a single **inode**structure.

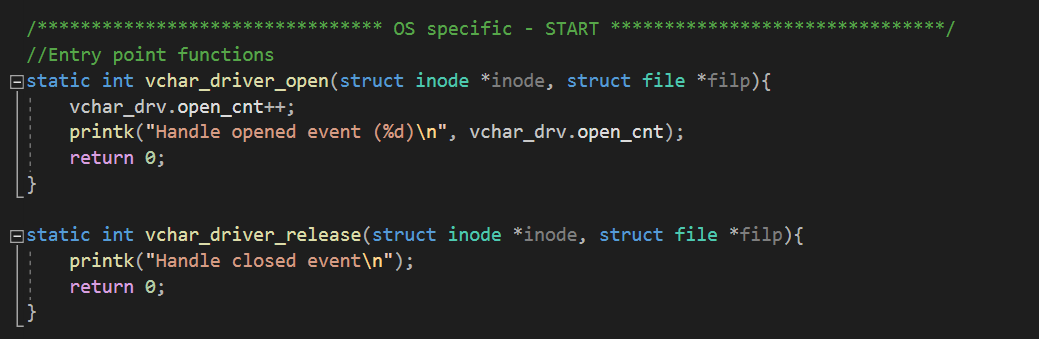
First, we include <linux/cdev.h> library to use the cdev structure and relevant function/

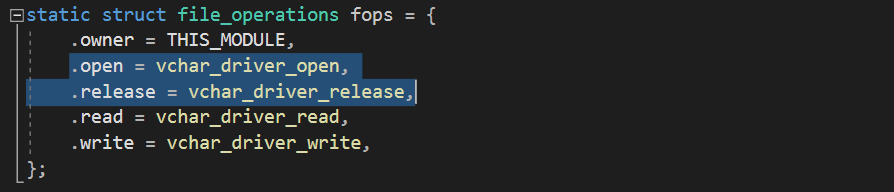


Next, we save the address of cdev structure, we add \*vcdev into vchar\_drv. To save how many time device file had been open, we add open\_cnt into vchar\_drv structure.

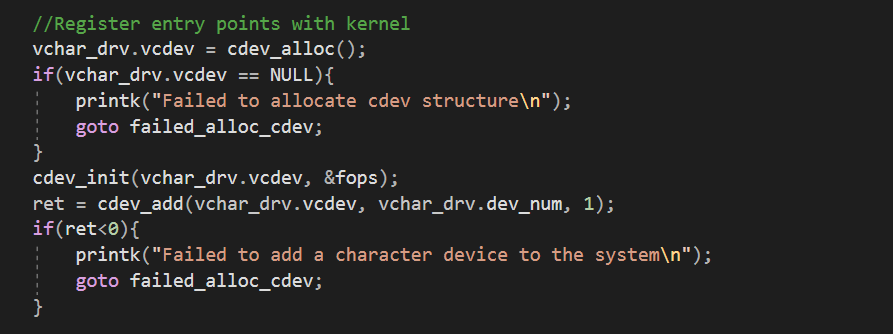


Next, to deploy vchar\_driver\_open and vchar\_driver\_release, we assign these functions to pointer open and release of file\_operation structure.

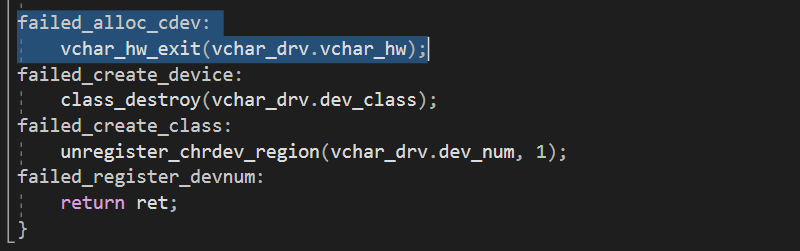




Inside vchar\_driver\_init function, we register cdev structure to Linux kernel. First, call the cdev\_alloc function to request the kernel create a case for new cdev. If this process failed, we destroy all the thing had been done before. If success, we continued to call cdev\_init to fill the information of cdev. Then call cdev\_add to send these information to kernel.

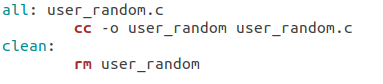


Inside vchar\_driver\_exit, we call cdev\_del to unregister entry point with kernel.

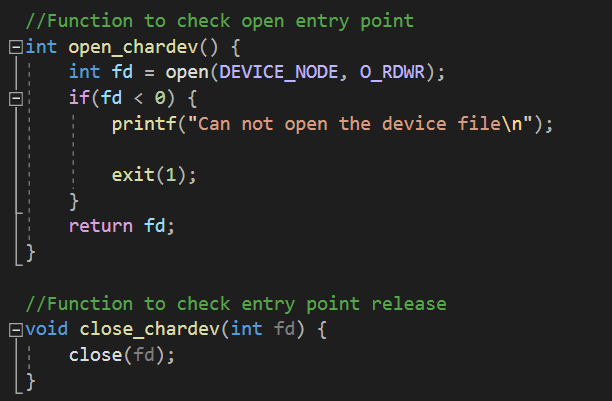


From here, I created a user app in user space that contact with vchar driver by system call open and close to check entry points close and open. First I created a folder name user\_app, and inside that folder, I created user\_random.c file and a Makefile to run this file.

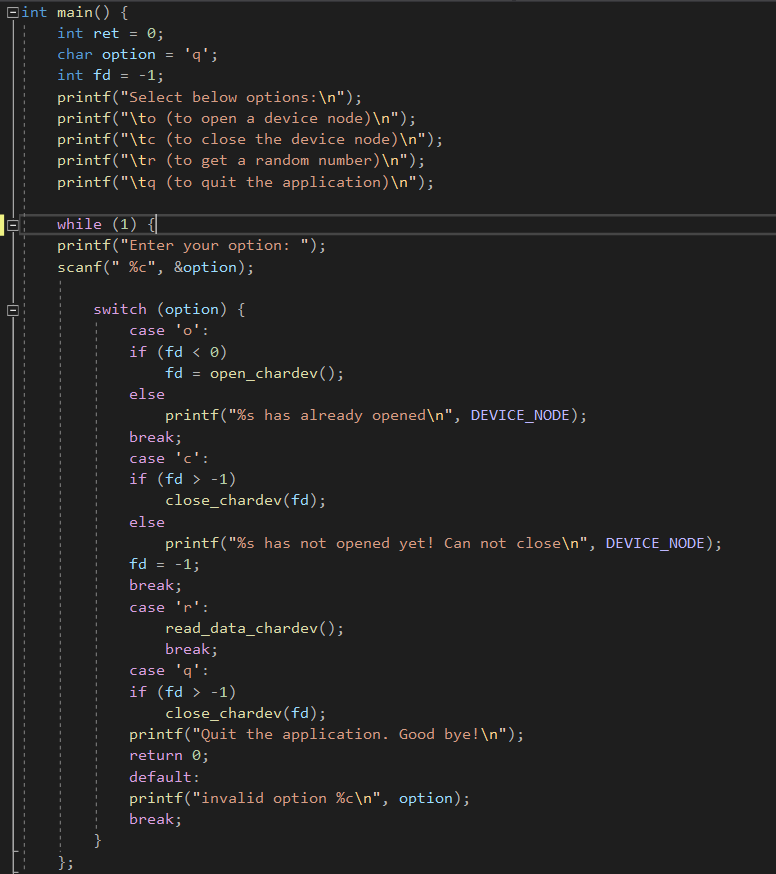
This is the Makefile.



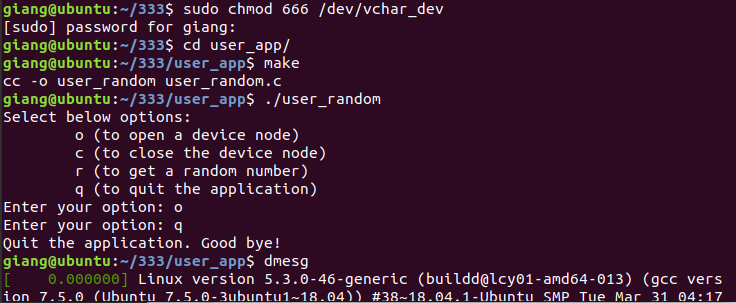
Inside user\_random.c, I created functions to check entry points open and release.



And using switch case to give options



I will run this app and check dmesg to see if open and release entry points is working or not. This app is in user space and is not allowed to access to /dev/vchar\_dev so we have to use chmod to grant access.





1. **Register entry points read and write**

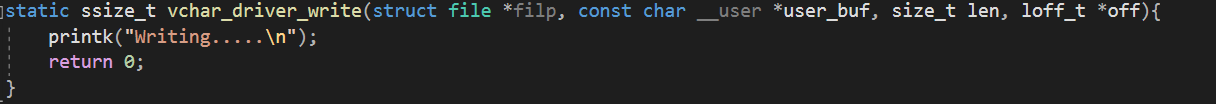
We add the <linux/uaccess.h> library to use the copy\_to\_user function and copy\_from\_user.



I put the function to create a random integer number in entry point read. First, initializing rand as an integer variable, then using get\_random\_bytes function to get a random integer number and that value is assigned into rand. Size of an integer number is 4 bytes. Call copy\_to\_user to copy rand to user buffer and return size of (int).



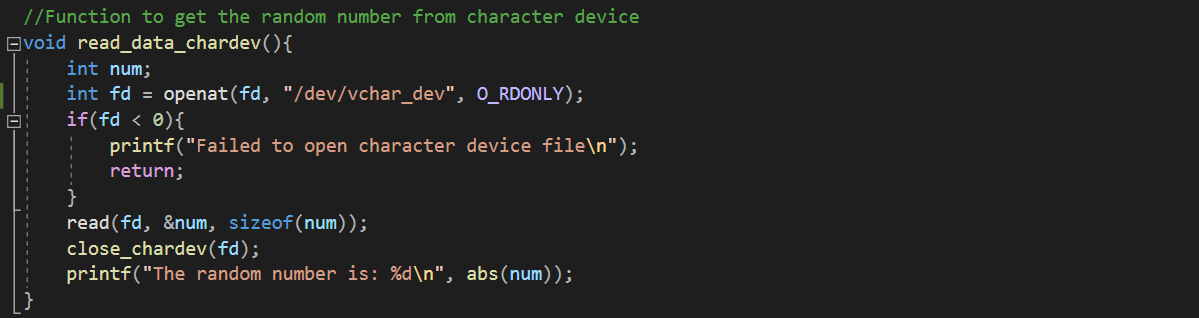
We write vchar\_driver\_write to deploy entry point write. We don’t use entry point write in this project.



Finally, to register these functions into entry point read and write of char driver, we need to assign address of vchar\_driver\_read and vhcar\_driver\_write to read and write of file\_operations structure.



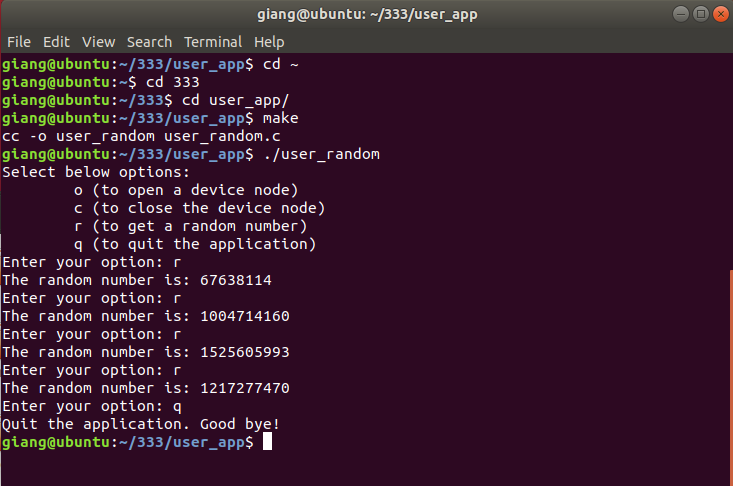
In user\_random.c, I created a function to get random integers from character device through entry point read. First opening device file at read-only mode and assigning the return value of the function to variable fd. If the device file can not be opened, quit the process. Read the random integer number to variable num and close the device file. Then print out the value of the random variable with abs() function to get positive number.



Add option “r” as showing a random integer to the switch case.



Let’s run this app to see the result.



1. **How to run**

First, go to the folder /333 and run make.



Insert module to kernel.



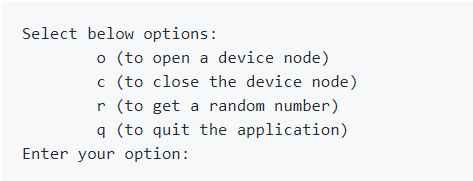
Get right to read device file.



Go to user\_app and run.



A selection board show up. Press “r” to get a random number.



To detach module from kernel.



To clean in /333 and /333/user\_app.

