

BÁO CÁO ĐỒ ÁN HỆ ĐIỀU HÀNH



Tên đồ án: TÌM HIỂU VÀ LẬP TRÌNH LINUX KERNEL MODULE

I. Giới thiệu sơ lược đồ án

- **Danh sách thành viên:**

1. Nguyễn Tất Hưng – 18127104
2. Nguyễn Minh Đức – 18127081
3. Nguyễn Hoàng Anh Tú - 18127243

- **Mô tả sơ lược:**

- Mục tiêu hiểu về Linux kernel module và hệ thống quản lý file và device trong linux, giao tiếp giữa tiến trình ở user space và xây dựng chương trình là một driver trong kernel.
- ✓ Viết một module dùng để tạo ra số ngẫu nhiên (yêu cầu là 4 byte – Integer).
- ✓ Module này sẽ tạo một character device để cho phép các tiến trình ở userspace có thể open device file, đọc số ngẫu nhiên được lưu trong entry point của driver trong device file và thao tác đóng file này lại ở menu được thiết kế riêng cho user.

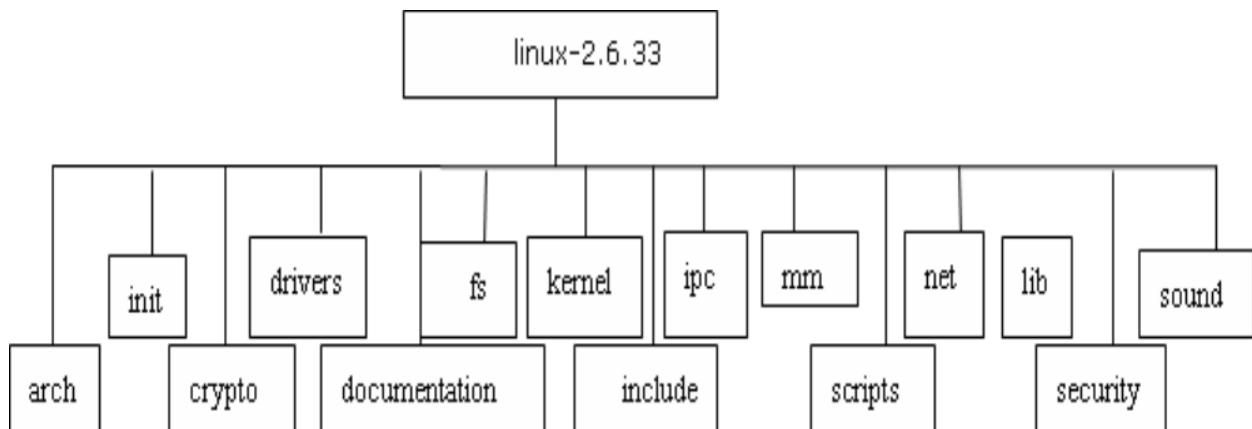
II. Tìm hiểu các định nghĩa và giải thích source code kernel linux

1. Tìm hiểu các định nghĩa:

a) Giới thiệu sơ lược về Linux Kernel:

- **Process management** có nhiệm vụ quản lý các tiến trình, bao gồm các công việc:
 - Tạo/hủy các tiến trình.
 - Lập lịch cho các tiến trình. Đây thực chất là lên kế hoạch: CPU sẽ thực thi chương trình khi nào, thực thi trong bao lâu, tiếp theo là chương trình nào.
 - Hỗ trợ các tiến trình giao tiếp với nhau.
 - Đồng bộ hoạt động của các tiến trình để tránh xảy ra tranh chấp tài nguyên.
- **Memory management** có nhiệm vụ quản lý bộ nhớ, bao gồm các công việc:
 - Cấp phát bộ nhớ trước khi đưa chương trình vào, thu hồi bộ nhớ khi tiến trình kết thúc.
 - Đảm bảo chương trình nào cũng có cơ hội được đưa vào bộ nhớ.
 - Bảo vệ vùng nhớ của mỗi tiến trình.
- **Device management** có nhiệm vụ quản lý thiết bị, bao gồm các công việc:
 - Điều khiển hoạt động của các thiết bị.

- Giám sát trạng thái của các thiết bị.
- Trao đổi dữ liệu với các thiết bị.
- Lập lịch sử dụng các thiết bị, đặc biệt là thiết bị lưu trữ (ví dụ ổ cứng).
- **File system management** có nhiệm vụ quản lý dữ liệu trên thiết bị lưu trữ (như ổ cứng, thẻ nhớ). Quản lý dữ liệu gồm các công việc: thêm, tìm kiếm, sửa, xóa dữ liệu.
- **Networking management** có nhiệm vụ quản lý các gói tin (packet) theo mô hình TCP/IP.
- **System call Interface** có nhiệm vụ cung cấp các dịch vụ sử dụng phần cứng cho các tiến trình. Mỗi dịch vụ được gọi là một **system call**.



- **Arch/:** Kernel linux có thể được cài đặt bằng công cụ cho các server lớn. Nó hỗ trợ intel, alpha, mips, arm, cấu trúc bộ xử lý sparc. Danh mục 'arch' có thể chứa các thư mục nhỏ cho một bộ xử lý cụ thể. Mỗi thư mục nhỏ chứa 1 mã cấu trúc độc lập. Ví dụ, đối với một PC, mã sẽ là thư mục arch/i386, đối với bộ xử lý arm, mã sẽ là thư mục arch/arm/arm64.
- **fs/:** Linux được hỗ trợ từ nhiều hệ thống file như ext2, ext3, fat, vfat, ntfs, nfs, jffs và nhiều hơn. Tất cả mã nguồn cho những file khác nhau này được hỗ trợ trong thư mục này theo những danh mục con như fs/ext2, fs/ext3 etc. Và linux cung cấp hệ thống file 16 ảo giống như lớp bọc cho những file khác. Hệ thống file ảo tương tác có thể giúp người dùng sử dụng hệ thống file khác với gốc ('/'). Mã này cho vfs cũng nằm ở đây. Cơ cấu dữ liệu liên quan vfs được xác định trong include/linux/fs.h. Điều này rất quan trọng để phát triển file cho kernel.
- **mm/:** Thư mục này rất quan trọng là thư mục cho phát triển kernel. Nó chứa mã chung quản lý bộ nhớ và hệ thống bộ nhớ ảo. Mã kiến trúc cụ thể là trong kiến trúc thư mục / * / mm /. Điều này một phần của mã kernel có trách nhiệm yêu cầu / giải phóng bộ nhớ, tin nhắn, xử lý lỗi trang, lập bản đồ bộ nhớ, lưu trữ khác nhau.

- **Giao diện mạng (Network Interface - NET):** Linux dựng sẵn TCP/IP trong kernel. Thành phần này của Linux Kernel cung cấp truy cập và kiểm soát các thiết bị mạng khác nhau.
- **Bộ truyền thông nội bộ (Inter-process communication IPC):** Cung cấp các phương tiện truyền thông giữa các tiến trình trong cùng hệ thống. Hệ thống phụ IPC cho phép các tiến trình khác nhau có thể chia sẻ dữ liệu với nhau.

b) Các lệnh trong Linux Kernel Module:

- User space và kernel space:

- Kernel space: Mã thực thi có quyền truy cập không hạn chế vào bất kỳ không gian địa chỉ nào của memory và tới bất kỳ phần cứng nào. Nó được dành riêng cho các chức năng có độ tin cậy cao nhất bên trong hệ thống. Kernel mode thường được dành riêng cho các chức năng hoạt động ở cấp độ thấp nhất, đáng tin cậy nhất của hệ điều hành. Do số lượng truy cập mà kernel có, bất kỳ sự không ổn định nào bên trong mã thực thi kernel cũng có thể dẫn đến lỗi hệ thống hoàn toàn.

- User space: Mã thực thi bị giới hạn truy cập. Nó là không gian địa chỉ mà các process user thông thường chạy. Những processes này không thể truy cập trực tiếp tới kernel space được. Khi đó lời gọi API được sử dụng tới kernel để truy vấn memory và truy cập thiết bị phần cứng. Bởi truy cập bị hạn chế, các trục trặc hay vấn đề gì xảy ra trong user mode chỉ bị giới hạn trong không gian hệ thống mà chúng đang hoạt động và luôn có thể khôi phục được. Hầu hết các đoạn mã đang chạy trên máy tính của bạn sẽ thực thi trong user mode.

- User mode và kernel mode:

- Khi CPU thực thi các lệnh của kernel, thì nó hoạt động ở chế độ **kernel mode**. Khi ở chế độ này, CPU sẽ thực hiện bất cứ lệnh nào trong tập lệnh của nó, và CPU có thể truy cập bất cứ địa chỉ nào trong không gian địa chỉ.

- Khi CPU thực thi các lệnh của tiến trình, thì nó hoạt động ở chế độ **user mode**. Khi ở chế độ này, CPU chỉ thực hiện một phần tập lệnh của nó, và CPU cũng chỉ được phép truy cập một phần không gian địa chỉ.

- System call và ngắt:

- System call là một cửa ngõ vào kernel, cho phép tiến trình trên tầng user yêu cầu kernel thực thi một vài tác vụ cho mình. Những dịch vụ này có thể là tạo một tiến trình mới (fork), thực thi I/O (read, write), hoặc tạo ra một pipe cho giao tiếp liên tiến trình (IPC).

- Ngắt là một sự kiện làm gián đoạn hoạt động bình thường của CPU, buộc CPU phải chuyển sang thực thi một đoạn mã lệnh đặc biệt để xử lý sự kiện đó. Thực chất, sự kiện này là một tín hiệu điện, do một mạch điện tử nằm bên trong hoặc bên ngoài CPU phát ra. Dựa vào điều này, ta chia ngắt thành 2 loại:

- Interrupt:

- Interrupt được phát ra bởi các I/O module nằm bên ngoài CPU, do đó nó còn được gọi là hardware interrupt.

- Interrupt được phát ra ở bất kì thời điểm nào, không phụ thuộc vào xung nhịp của CPU, nên ta còn gọi là asynchronous interrupt.

- Dựa vào tính chất của Interrupt, ta chia Interrupt làm hai loại:

+ Non-maskable Interrupt là các tín hiệu khẩn cấp. CPU luôn tiếp nhận Interrupt thuộc loại này.

+ Maskable Interrupt là các tín hiệu không thật sự khẩn cấp. CPU có thể tiếp nhận các Interrupt thuộc loại này hoặc không.

- Process context và interrupt context:

- Process context: Đây là một trong những phần quan trọng nhất trong việc thực thi chương trình. Các câu lệnh được đọc từ các file có thể thực thi và được thực thi trong không gian địa chỉ của chương trình. Việc thực thi các chương trình bình thường diễn ra ở User-space. Khi 1 chương trình thực thi 1 system call hay bắt đầu 1 ngoại lệ. Nó đi vào Kernel-space. Tại đây, Kernel dc xem như là đang thực thi chương trình thay cho process và ở trong process context

- Interrupt context: Khi thực thi một trình phục vụ ngắt (interrupt handler), đơn giản là kernel đang ở trong interrupt context. Trái ngược với process context, interrupt context không liên kết gì với một process, vô hiệu hóa bộ lập lịch, và nó tồn tại một cách độc lập với process context với mục đích giúp trình phục vụ ngắt phản hồi thật nhanh một ngắt và exit. Loại ngữ cảnh đặc biệt này thỉnh thoảng còn được gọi là atomic context, vì code thực thi ở ngữ cảnh này không được phép block (hay ngủ). (Nonatomic context thì ngược lại, như vậy có vẻ nonatomic ám chỉ process context). Interrupt context không thể ngủ, do đó không thể sử dụng một số hàm ở đây (ví dụ như sleep function). Interrupt context vô cùng khẩn khe về mặt thời gian (time-critical) vì trình phục vụ ngắt đã ngắt một chương trình khác, vậy nên trình phục vụ phải nhanh và đơn giản.

2. Giải thích source code Linux Kernel Module:

→ Để thuận tiện cho quá trình biên dịch, thay vì tự mình viết những câu lệnh dài thì ta tạo ra 2 file Makefile và Kbuild như sau

- **Make file:**

- Để chương trình có thể thực thi trong linux, ta cần các câu lệnh và make là một trong những câu lệnh quan trọng trong quá trình này. Đầu tiên ta vào Terminal gõ câu lệnh gedit Makefile , cửa sổ file Makefile hiện ra và ta có những câu lệnh cơ bản như sau:



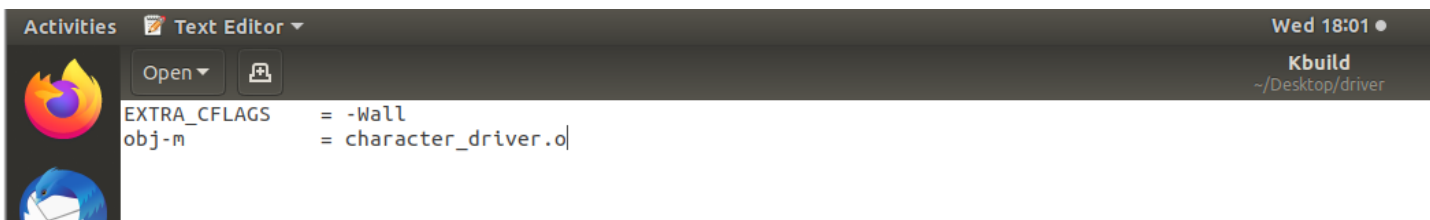
The screenshot shows a Linux Text Editor window titled 'Text Editor' with a file named 'Makefile' located at '~/Desktop/driver'. The editor contains the following content:

```
KDIR = /lib/modules/`uname -r`/build
all:
    make -C $(KDIR) M=`pwd`

clean:
    make -C $(KDIR) M=`pwd` clean
```

- **Kbuild:**

- Ở đây, ta tạo ra một driver ảo có tên là character_driver
- Cách tạo cũng tương tự như Makefile bằng cách gõ gedit Kbuild và trong file ta viết như sau:



The screenshot shows a Linux Text Editor window titled 'Text Editor' with a file named 'Kbuild' located at '~/Desktop/driver'. The editor contains the following content:

```
EXTRA_CFLAGS = -Wall
obj-m        = character_driver.o
```

→Sau khi thiết lập xong 2 file Makefile và Kbuild, Ta tạo file cấu hình module có tên là character_driver.c gồm các biến và thư viện sau:

```
//Head
#include<linux/module.h>
#include<linux/fs.h>
//Device init
#include<linux/device.h>

// cập nhật bộ nhớ cấu trúc dữ liệu của driver và khởi tạo//
#include<linux/slab.h>
#include"character_driver.h"
#include <linux/random.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include<linux/cdev.h>
#include <linux/sched.h>
#include <linux/workqueue.h>
#include <linux/interrupt.h>
#define DRIVER_AUTHOR "NGUYEN TAT HUNG - NGUYEN MINH DUC - NGUYEN HOANG ANH TU"
#define DRIVER_DESC "TEST MODULE LINUX"
#define DRIVER_VERSION "Released"
```

- Phần đầu file chứa các thư viện, các thông tin cơ bản về module như là tên tác giả, mô tả và phiên bản module.

- Về phần sau sẽ là phần OS specific

- **OS specific:** chứa các hàm khởi tạo (`init_character_driver`), hàm kết thúc (`exit_character_driver`) và các hàm entry point (`dev_open`, `dev_read`, `dev_release`).

➔ Thông thường sẽ có thêm phần device specific chứa các hàm khởi tạo/giải phóng thiết bị, đọc/ghi các thanh ghi và các hàm xử lý tín hiệu ngắt đến từ thiết bị nhưng lần này ta sẽ gom chung các câu lệnh này và các hàm khởi tạo để khiến cấu trúc file đỡ rối rắm hơn.

- **Các lệnh trong hàm khởi tạo (`init_character_driver`) và hàm hủy (`exit_character_driver`):**

A. Đăng ký driver (`character_driver`)

- Để khởi tạo một module trong Linux Kernel, Linux cung cấp cho chúng ta 2 phương pháp khởi tạo đó là phương pháp cấp phát tĩnh và cấp phát động để cấp phát device number cho driver. Trong đồ án, ta sử dụng phương pháp cấp phát động vì nó rất linh hoạt và có thể sử dụng trên nhiều thiết bị khác nhau mà không phát sinh vấn đề.

```
static int __init character_driver_init(void)
{
    //cấp phát device number//
    int ret=0;
    vchar_drv.dev_number = 0;
    ret = alloc_chrdev_region(&vchar_drv.dev_number, 0, 1, "character_device");
    if(ret<0)
    {
        printk("Failed to register device number. \n");
        return ret;
    }
    printk("Allocated device number (%d,%d) \n", MAJOR(vchar_drv.dev_number), MINOR(vchar_drv.dev_number));
}
```

- Để thực hiện cấp phát động, Linux kernel cung cấp một hàm là **`alloc_chrdev_region`**. Nhiệm vụ của hàm này là tìm ra một giá trị có thể dùng làm device number.

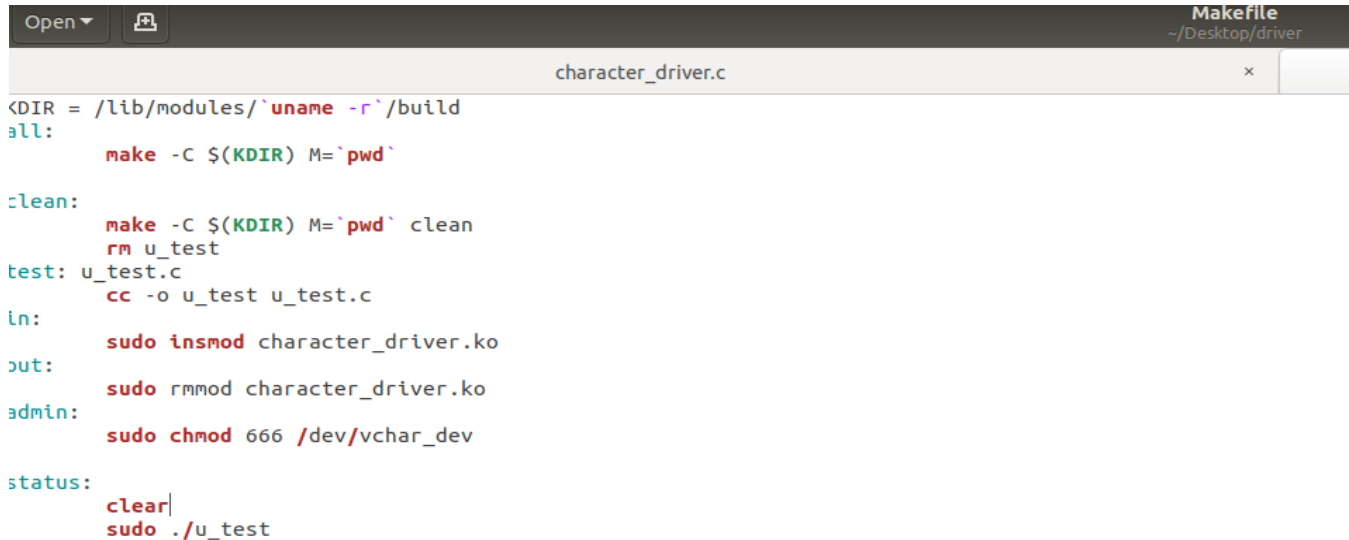
- Major number giúp kernel xác định device driver nào tương ứng với device file.
- Minor number giúp driver biết nó cần tương tác với thiết bị nào.

- Để lưu giá trị device number, ta sẽ tạo ra một struct **`vchar_drv`** chứa trường **`dev_number`** (kiểu **`dev_t`**).

```
struct _vchar_drv
{
    dev_t dev_number;
```

- Sau khi gọi hàm để kernel driver tìm kiếm một giá trị device number phù hợp. Nếu không có giá trị phù hợp thì hàm sẽ trả về giá trị âm và thông báo rằng không thể lắp driver này vào kernel được.

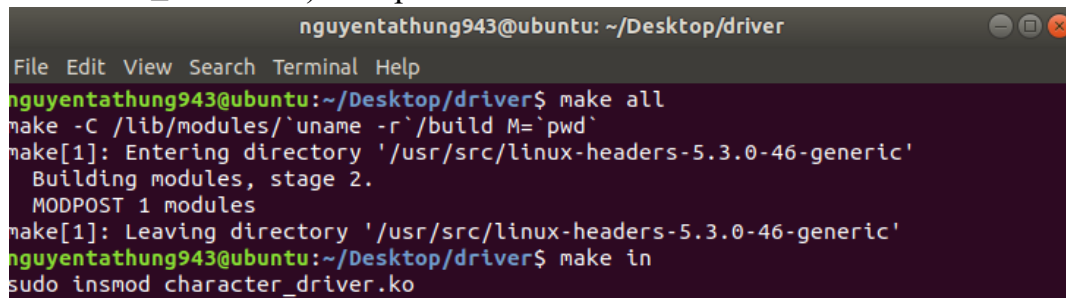
*Để tránh gõ những câu lệnh dài dòng, ta định nghĩa file Makefile lại như sau:



```
Open Makefile ~/Desktop/driver
character_driver.c
<DIR = /lib/modules/`uname -r`/build
all:
    make -C $(KDIR) M=`pwd`

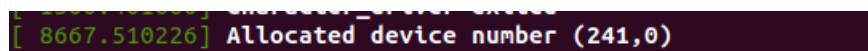
clean:
    make -C $(KDIR) M=`pwd` clean
    rm u_test
test: u_test.c
    cc -o u_test u_test.c
in:
    sudo insmod character_driver.ko
out:
    sudo rmmod character_driver.ko
admin:
    sudo chmod 666 /dev/vchar_dev
status:
    clear
    sudo ./u_test
```

- Ta mở Terminal, gõ make all để thực thi, sau đó gõ lệnh make in (**sudo insmod character_driver.ko**) để ráp driver vào kernel.



```
nguyentathung943@ubuntu: ~/Desktop/driver
File Edit View Search Terminal Help
nguyentathung943@ubuntu:~/Desktop/driver$ make all
make -C /lib/modules/`uname -r`/build M=`pwd`
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-46-generic'
Building modules, stage 2.
MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-46-generic'
nguyentathung943@ubuntu:~/Desktop/driver$ make in
sudo insmod character_driver.ko
```

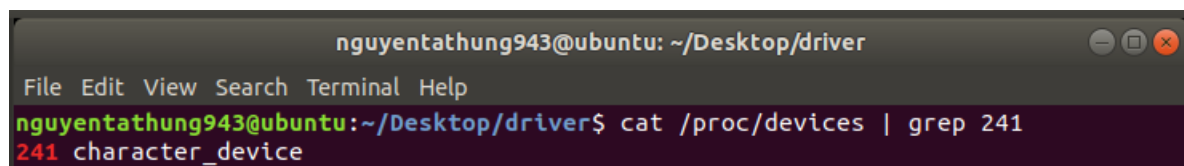
- Sau đó, ta gõ **dmesg** để kiểm tra:



```
[ 8667.510226] Allocated device number (241,0)
```

→ Ta thấy character_driver đã được đánh số Major là 241 và Minor là 0.

- Recheck thư mục bằng lệnh **cat /proc/devices | grep 241** để kiểm tra lại lần nữa:



```
nguyentathung943@ubuntu: ~/Desktop/driver
File Edit View Search Terminal Help
nguyentathung943@ubuntu:~/Desktop/driver$ cat /proc/devices | grep 241
241 character_device
```

- Trong hàm **character_driver_exit**, ta đăng ký hàm hủy tương ứng:

```
//giai phong device number//
unregister_chrdev_region(vchar_drv.dev_number, 1);
```

→ Ta đã hoàn tất đăng ký cho driver

B. Tạo device file:

- Tạo device file thực chất là đánh lừa các tiến trình rằng các char/block device cũng chỉ là các file thông thường. Do đó, các tiến trình sẽ nghĩ rằng, đọc/ghi dữ liệu từ thiết bị cũng giống như đọc/ghi dữ liệu từ file thông thường.

- Có 2 cách để tạo device file trong Kernel Linux đó là cách **thủ công** và **tự động**. Cách thủ công bằng câu lệnh

```
mknod -m <quyền truy cập> <tên device file> <kiểu device> <major> <minor>
```

- Nhưng thay vì sử dụng phương pháp **thủ công** thì ta có thể sử dụng phương pháp tự động bằng cách tham chiếu tới thư viện **<linux/device.h>** và sử dụng 2 hàm tạo và hủy sau đây:

```
struct device* device_create(struct class* cls, struct device *parent,
                             dev_t devt, void *drvdata, const char *name)

/* Hàm hủy tương ứng với device_create */
void device_destroy(struct class * cls, dev_t devt)
```

- Ở đây ta gọi hàm trong hàm **init_character_driver**

```
//tao device file//
vchar_drv.dev_class = class_create(THIS_MODULE, "class_vchar_dev");
if(vchar_drv.dev_class==NULL)
{
    printk("Failed to create device file");
    unregister_chrdev_region(vchar_drv.dev_number, 1);
    return 0;
}
vchar_drv.dev = device_create(vchar_drv.dev_class, NULL, vchar_drv.dev_number, NULL, "vchar_dev");
if(IS_ERR(vchar_drv.dev))
{
    printk("Failed to create device file");
    class_destroy(vchar_drv.dev_class);
    return 0;
}
printk("Device file has been created");
```


- Để lưu kết quả trả về của **class_create** và **device_create**, ta thêm trường ***dev_class** và ***dev** vào trong cấu trúc **vchar_drv**.

```
struct _vchar_drv
{
    dev_t dev_number;
    struct class *dev_class;
    struct device *dev;
};
```

- Trong hàm **character_driver_init**, ta gọi hàm **class_create** để tạo ra 1 class thiết bị có tên “**class_vchar_dev**”. Nếu hàm này không thực hiện thành công, hàm **unregister_chrdev_region** sẽ giải phóng device number đã được cấp phát trước đó và hiện thông báo. Còn nếu thành công, ta tiếp tục gọi hàm **device_create** để tạo 1 device có tên là **vchar_dev** trong thư mục **/dev**. Nếu hàm này thất bại, ta gọi **destroy_class** để giải phóng class thiết bị.

- Trong hàm **character_driver_exit**, ta đăng ký hàm huỷ tương ứng:

```
//xoa bo device file//
device_destroy(vchar_drv.dev_class, vchar_drv.dev_number);
class_destroy(vchar_drv.dev_class);
```

- Ta gọi 2 hàm huỷ theo thứ tự như trên. 2 hàm này có nhiệm vụ giải phóng tất cả những **class device** và **class** đã tạo ra trước đó.

- Để kiểm tra device file đã được tạo hay chưa, cách đơn giản nhất là gõ lệnh **ls -la /dev/vchar_dev** và **cat /sys/class/class_vchar_dev/vchar_dev/uevent**. Nếu hiện như sau thì thư mục đã được tạo thành công

```
nguyentathung943@ubuntu:~/Desktop/driver$ ls -la /dev/vchar_dev
crw-rw-rw- 1 root root 241, 0 Apr 22 19:54 /dev/vchar_dev
nguyentathung943@ubuntu:~/Desktop/driver$ cat /sys/class/class_vchar_dev/vchar_dev/uevent
MAJOR=241
MINOR=0
DEVNAME=vchar_dev
nguyentathung943@ubuntu:~/Desktop/driver$
```

- Trong quá trình khởi tạo **character_driver**, hàm **class_create** sẽ giúp tạo thư mục **class_vchar_dev** trong thư mục **/sys/class**, và hàm **device_create** giúp tạo thư mục **vchar_dev** bên trong thư mục **/sys/class/class_vchar_dev**. Bên trong thư mục **vchar_dev**, có một file tên là **uevent**. Ta truy vấn file thì sẽ có thông tin hiện như trên

- Ta kiểm tra trong bằng lệnh **dmesg**

```
[ 8667.510226] Allocated device number (241,0)
[ 8667.510319] Device file has been created
```

→ Ta đã thành công tạo device file

C. Cấp phát bộ nhớ và khởi tạo thiết bị vật lý:

○ Cấp phát bộ nhớ:

- Để thực hiện cấp phát bộ nhớ, ta sử dụng hàm **kmalloc** để cấp phát N byte có cú pháp như sau, tham chiếu sử dụng thư viện **<linux/slab.h>**

```
#include <linux/slab.h>
kmalloc(N, GFP_KERNEL);
```

- Nếu cấp phát thành công, hàm sẽ trả về địa chỉ của vùng nhớ được cấp phát. Nếu lỗi sẽ trả về NULL.

- Có 1 hàm khác được sử dụng rộng rãi hơn đó là hàm **kzalloc**. Không chỉ cấp phát bộ nhớ mà còn set các byte của vùng nhớ bằng 0. Cách hoạt động và giá trị trả về tương tự **kmalloc**.

- Để giải phóng vùng nhớ, ta dùng hàm **kfree** với tham số truyền vào là địa chỉ của ô nhớ.

- Trong hàm **character_driver_init**, khai triển như sau

```
//cấp phát bộ nhớ cấu trúc dữ liệu của driver và khởi tạo//
vchar_drv.vchar_hw=kzalloc(sizeof(vchar_dev_t), GFP_KERNEL);
if(!vchar_drv.vchar_hw){
    printk("Failed to allocate data structure of the device\n");
    ret=-ENOMEM;
    device_destroy(vchar_drv.dev_class, vchar_drv.dev_number);
    return 0;
}
printk("Data structure allocated successfully");
```

- Tiếp theo sẽ sang bước khởi tạo thiết bị vật lý.

○ Khởi tạo thiết bị vật lý:

- Trong cấp độ Kernel Linux, các thiết bị vật lý bao gồm các thanh ghi. Các thanh ghi được phân làm 3 loại:

+ Thanh ghi điều khiển (control register)

+ Thanh ghi trạng thái (status register)

+ Thanh ghi dữ liệu (data register)

- Đầu tiên, ta tạo 1 file có tên **character_driver.h** chứa các thông tin của 1 thanh ghi điều khiển, 5 thanh ghi trạng thái và 256 thanh ghi dữ liệu, mỗi thanh có kích thước 1 byte. Thiết bị giả lập có tên **character_driver**

- Ta viết hàm đăng ký thiết bị và xóa thiết bị vật lý

```

//khởi tạo thiết bị vật lý//
int vchar_hw_init(vchar_dev_t*hw)
{
    char*buf;
    buf=kzalloc(NUM_DEV_REGS*REG_SIZE, GFP_KERNEL);
    if(!buf)
    {
        return -ENOMEM;
    }
    hw->control_regs=buf;
    hw->status_regs=hw->control_regs + NUM_CTRL_REGS;
    hw->data_regs=hw->status_regs + NUM_STS_REGS;

    //KHỞI TẠO GIA TRI THANH GHI//
    hw->control_regs[CONTROL_ACCESS_REG] = 0x03;
    hw->status_regs[DEVICE_STATUS_REG]=0x03;
    return 0;
}

//XÓA THIẾT BỊ VẬT LÝ
void vchar_hw_exit(vchar_dev_t*hw)
{
    kfree(hw->control_regs);
}

```

- Trong hàm cấp phát bộ nhớ ở mục đầu, nếu thất bại thì sẽ bị thu hồi tất cả bao gồm hủy class device, hủy class và thu hồi device number.

```

//cấp phát bộ nhớ cấu trúc dữ liệu của driver và khởi tạo//
vchar_drv.vchar_hw=kzalloc(sizeof(vchar_dev_t), GFP_KERNEL);
if(!vchar_drv.vchar_hw){
    printk("Failed to allocate data structure of the device\n");
    ret=-ENOMEM;
    device_destroy(vchar_drv.dev_class, vchar_drv.dev_number);
    return 0;
}
printk("Data structure allocated successfully");

//khởi tạo thiết bị vật lý
ret=vchar_hw_init(vchar_drv.vchar_hw);
if(ret<0)
{
    printk("Failed to initialize a virtual character device");
    kfree(vchar_drv.vchar_hw);
    return 0;
}
printk("Virtual character device initialized successfully");

```

- Để char driver tương tác được với thiết bị “character_driver”, ta định nghĩa cấu trúc `vchar_dev_t`, đồng thời thêm trường `*vchar_hw` vào trong cấu trúc `vchar_drv`

```

struct _vchar_drv
{
    dev_t dev_number;
    struct class *dev_class;
    struct device *dev;
    vchar_dev_t* vchar_hw;
}

```

- Việc còn lại đó là gọi hàm hủy trong **exit_character_driver** để giải phóng bộ nhớ và thu hồi thiết bị

```
//giai phong thiet bi vat ly//  
vchar_hw_exit(vchar_drv.vchar_hw);
```

```
//giai phong bu nho da cap phat cau truc du lieu cua driver//  
kfree(vchar_drv.vchar_hw);
```

- Để biết chắc hàm hoạt động tốt, ta kiểm tra trong **dmesg**

```
19625.538529] Character_driver exited  
19634.098138] Allocated device number (241,0)  
19634.098291] Device file has been created  
19634.098291] Data structure allocated successfully  
19634.098292] Virtual character device initialized successfully
```

→ Ta hoàn thành task tại đây

D. Đăng ký các entry point (bao gồm random number module):

- Entry point tương ứng với function được gọi khi module được load (insmod, modprobe) và exit point tương ứng với function được gọi khi unload module (rmmod, modprobe -r).

- Đầu tiên, ta gọi hàm **cdev_alloc** để yêu cầu kernel "cấp cho một bộ hồ sơ" cdev mới. Nếu quá trình cấp phát thất bại, thì ta cần hủy những gì đã làm được trước đó (bao gồm giải phóng thiết bị, thu hồi bộ nhớ đã cấp phát cho các cấu trúc của char driver, hủy device file, hủy lớp, thu hồi device number). Còn nếu thành công, ta tiếp tục gọi hàm **cdev_init** để "điền các thông tin vào bộ hồ sơ" cdev. Sau đó, ta gọi hàm **cdev_add** để gửi "bộ hồ sơ" này tới kernel.

```
//dang ky cac entry point//  
vchar_drv.vcdev=cdev_alloc();  
if(vchar_drv.vcdev==NULL){  
    printk("Failed to allocate cdev structure");  
    vchar_hw_exit(vchar_drv.vchar_hw);  
    return 0;  
}  
cdev_init(vchar_drv.vcdev, &fops);  
ret=cdev_add(vchar_drv.vcdev, vchar_drv.dev_number, 1);  
if(ret<0){  
    printk("Failed to add entry point!");  
    vchar_hw_exit(vchar_drv.vchar_hw);  
    return 0;  
}  
printk("Entry points added succesfully");
```

- Ví dụ, system call **open** tương ứng với entry point **open**, system call **close** tương ứng với entry point **release**, system call **read** tương ứng với entry point **read**, system call **write** tương ứng với entry point **write**... Nhờ vậy, các tiến trình có thể phát đi các system call **open**, **read**, **write**, **close** trên device file để tương tác với thiết bị vật lý. Các hoạt động tương tác này bao gồm mở một phiên làm việc với thiết bị (được triển khai bởi entry point **open**), đọc/ghi dữ liệu từ thiết bị (được triển khai bởi entry point **read/write**), kết thúc phiên làm việc với thiết bị (được triển khai bởi entry point **release**).

- Entry point open:

```
static int dev_open(struct inode *inode, struct file *filp)
{
    printk(KERN_INFO "character_driver: Device is opening....\n");
    return 0;
}
```

- Entry point release:

```
static int dev_release(struct inode *inode, struct file *filp)
{
    printk(KERN_INFO "character_driver: Device successfully closed\n");
    return 0;
}
```

-Entry point read:

- Khi tiến trình cần đọc dữ liệu từ một char device nào đó, các bước diễn ra như sau:

- Đầu tiên, tiến trình gọi system call open để mở device file tương ứng của char device này.
- Sau đó, tiến trình gọi system call read để đọc dữ liệu từ device file này. Khi đó, Linux kernel sẽ kích hoạt entry point read của char driver tương ứng.
- Cuối cùng, entry point read đọc dữ liệu từ char device ra rồi trả về cho tiến trình trên user space.

```
static ssize_t dev_read(struct file *filp, char *buffer, size_t len, loff_t *offset)
{
    unsigned int randomnumber;
    get_random_bytes(&randomnumber, sizeof(randomnumber));
    printk("character_driver: Random number is : %u\n", randomnumber);
    return randomnumber;
}
```

- Hàm get_random_bytes tạo ra 1 số ngẫu nhiên để trả về cho user space.

- Để đăng ký các hàm này trở thành entry point của char driver, ta tiến hành gán địa chỉ của các hàm **dev_open**, **dev_read** và **dev_release** cho các trường **open**, **read** và **release** của cấu trúc **file_operations**.

```
static struct file_operations fops =
{
    .open = dev_open,
    .read = dev_read,
    .release = dev_release,
};
```

- Hủy đăng ký entry point:

```
//huy dang ky entry point vs kernel//
cdev_del(vchar_drv.vcdev);
```

→ Ta hoàn thành đăng ký entry point (open, read, release)

E. Hàm xử lý ngắt

- Để đăng ký một hàm xử lý một ngắt của hệ thống, chúng ta cần thực hiện 2 bước sau:

- Viết hàm xử lý ngắt – hàm chứa những hành động chúng ta cần thực thi khi ngắt xảy ra.
- Sử dụng hàm `request_irq()` để đăng ký hàm xử lý ngắt. Kết quả của việc đăng ký xử lý ngắt là hàm xử lý ngắt sẽ được gọi – thực thi khi ngắt xảy ra

- Hàm đăng ký xử lý ngắt:

```
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char * name, void * dev)
```

- Trong đó tham số *irq* là số thứ tự của ngắt
- Tham số *irq_handler_t handler* là hàm xử lý ngắt cần đăng ký vào ngắt thứ *irq*.
- Tham số *flags* có thể bằng 0 hoặc là bit mask của 1 hay nhiều cờ được định nghĩa trong **<linux/interrupt.h>**.
- Tham số thứ 4 *name* là tên liên kết với ngắt. Tên này được sử dụng bởi các procfile `/proc/irq` và `/proc/interrupts`.
- Tham số thứ 5, *dev*, được sử dụng cho các đường ngắt chia sẻ chung (shared interrupt lines). Như đã giải thích ở chương trước, có thể đăng ký nhiều hàm xử lý ngắt trên cùng một đường ngắt chia sẻ chung. Khi một hàm xử lý ngắt của đường ngắt chia sẻ chung được giải phóng (freed), kernel cần biết chính xác nó cần giải phóng hàm xử lý ngắt nào. Tham số thứ 5 *dev* giúp kernel thực hiện việc đó – điều này sẽ được làm rõ ở phần “Giải phóng một hàm xử lý ngắt”. Tham số thứ 5 có thể truyền vào với giá trị là NULL nếu như đường ngắt không phải là đường ngắt chia sẻ chung.

- Cờ của hàm xử lý ngắt – tham số truyền vào hàm xử lý ngắt:

- Các cờ của hàm xử lý ngắt được định nghĩa trong **<linux/interrupt.h>**. Một vài cờ quan trọng nhất được mô tả trong bảng dưới đây:

Cờ	Chức năng
IRQF_DISABLED	Khi cờ này được set cho một hàm xử lý ngắt, kernel sẽ disable tất cả các ngắt còn lại khi thực thi hàm xử lý ngắt này.
IRQF_TIMER	Cờ này chỉ rõ hàm xử lý ngắt này xử lý ngắt của timer của hệ thống.
IRQF_SHARED	Cờ này chỉ ra đường ngắt này được chia sẻ sử dụng chung cho nhiều hàm xử lý ngắt.

- Giải phóng một hàm xử lý ngắt:

```
void free_irq(unsigned int irq, void * dev)
```

Trong đó:

- Tham số đầu tiên *irq* là số thứ tự của ngắt (giá trị của vector ngắt *n* đã đề cập ở chương 7).
- Tham số thứ 2 *dev* giống như tham số thứ 5 *dev* trong hàm đăng kí hàm xử lý ngắt. Ở đây ta có thể hiểu vai trò của tham số *dev*.

- Tạo một hàm xử lý ngắt:

- Đây là mã nguồn kernel module đăng kí một hàm xử lý ngắt cho bàn phím. Kernel module sẽ in log khi người dùng ấn nút ESC, F1 hoặc F2

```
/* Đây là hàm xử lý ngắt cần được đăng kí vào đường ngắt dành cho bàn phím */
irqreturn_t irq_handler(int irq, void * dev_id, struct pt_regs * regs)
{
    /* ** This variables are static because they need to be **
    accessible (through pointers) to the bottom half routine. ** */
    static unsigned char scancode;
    unsigned char status;
    /* ** Read keyboard status ** */
    status = inb(0x64);
    scancode = inb(0x60);
    switch (scancode) {
        case 0x01:
            printk(KERN_EMERG "! You pressed Esc ...\n");
            break;
        case 0x3B:
            printk(KERN_EMERG "! You pressed F1 ...\n");
            break;
        case 0x3C:
            printk(KERN_EMERG "! You pressed F2 ...\n");
            break;
        default:
            break;
    }
    return IRQ_HANDLED;
}

//đăng ký hàm xử lý ngắt//

ret = request_irq(1,
(irq_handler_t)irq_handler, IRQF_SHARED, "character_driver_interrupt", (void * )
(irq_handler));

if (ret<0){
    printk(KERN_EMERG "Can't initialize character_driver_interrupt\n");
}

//Hủy đăng ký xử lý ngắt//
free_irq(1, (void * )(irq_handler));
```

- Trong phần mã nguồn, hàm đăng kí xử lý ngắt đăng kí vào đường ngắt số 1 – đường ngắt dành cho bàn phím. Còn đăng kí **IRQF_SHARED** mục đích là đăng kí thêm một hàm xử lý ngắt vào đường ngắt. Tên liên kết với hàm xử lý ngắt là “**character_driver_interrupt**”, và tham số thứ 5 **irq_handler** được đăng kí và giải phóng hàm xử lý ngắt.

```
anhtu@ubuntu:~/Desktop$ cat /proc/interrupts
```

	CPU0	CPU1			
0:	4	0	IO-APIC	2-edge	timer
1:	0	272	IO-APIC	1-edge	i8042, character_driver_in
8:	1	0	IO-APIC	8-edge	rtc0
9:	0	0	IO-APIC	9-fasteoi	acpi
12:	5547	0	IO-APIC	12-edge	i8042
14:	0	0	IO-APIC	14-edge	ata_piix
15:	0	0	IO-APIC	15-edge	ata_piix
16:	2746	698	IO-APIC	16-fasteoi	vmwgfx, snd_ens1371
17:	22248	0	IO-APIC	17-fasteoi	ehci_hcd:usb1, ioc0
18:	0	168	IO-APIC	18-fasteoi	uhci_hcd:usb2
19:	0	16552	IO-APIC	19-fasteoi	ens33

- Kết quả hệ thống sau khi insmod: Ngắt có giá trị 1 có hai hàm xử lý ngắt được đăng kí là có tên là **i8042** và **character_driver_interrupt**.


```
[ 962.046182] ! You pressed F1 ...
[ 963.410697] ! You pressed F2 ...
[ 963.410776] ! You pressed F2 ...
[ 963.411901] ! You pressed Esc ...
anhthu@ubuntu:~/Desktop$
```

- Kết quả sau khi nhấn thử các nút bấm ESC, F1 và F2

```
anhthu@ubuntu:~/Desktop$ make out
sudo rmmod character_driver.ko
[sudo] password for anhthu:
anhthu@ubuntu:~/Desktop$ cat /proc/interrupts
```

	CPU0	CPU1			
0:	4	0	IO-APIC	2-edge	timer
1:	0	458	IO-APIC	1-edge	i8042
8:	1	0	IO-APIC	8-edge	rtc0
9:	0	0	IO-APIC	9-fasteoi	acpi
12:	14472	0	IO-APIC	12-edge	i8042
14:	0	0	IO-APIC	14-edge	ata_piix
15:	0	0	IO-APIC	15-edge	ata_piix
16:	9625	698	IO-APIC	16-fasteoi	vmwgfx, snd_ens1371
17:	24238	0	IO-APIC	17-fasteoi	ehci_hcd:usb1, ioc0
18:	0	168	IO-APIC	18-fasteoi	uhci_hcd:usb2

- Kết quả hệ thống sau khi unload kernel module – giải phóng hàm xử lý ngắt “character_driver_interrupt”.

→ Ta hoàn thành hàm xử lý ngắt

F. User_space của menu người dùng:

- Để người dùng giao tiếp dễ dàng hơn driver trong Kernel Linux, ta xây dựng menu người dùng gồm 3 chức năng

- Mở device file của driver

- Đọc dữ liệu từ file (entry point read chứa số ngẫu nhiên nằm trong file)

- Đóng device file

Ta xây dựng file **u_test.c** tương trưng cho menu người dùng như sau:

```

int main()
{
    unsigned int ret, fd;
    unsigned int result;
    char option;

    printf("WELCOME TO THE DRIVER DEVICE INTERFACE \n");
    printf("Make your own choices: \n");
    printf("-----\n");
    printf("*Press 1 to open the device file\n");
    printf("*Press 2 to read the random 4 bytes number in the device file\n");
    printf("*Press 3 to exit device file\n");

    while(1){
        printf("You choose: ");
        scanf(" %c", &option);
        switch(option){
            case '1':
                fd = open(DEVICE_NODE, O_RDWR); // Open the device with read/write access
                if (fd < 0)
                {
                    printf("Failed to open device file!.....\n");
                    return 0;
                }
                else{
                    printf("Device file opened!\n");
                }
                break;

            case '2':
                printf("Reading from the device...\n");
                ret = read(fd, &result, sizeof(BUFFER_LENGTH)); // Read the response from the LKM
                printf("Random number is: %u\n", ret);
                break;

            case '3':
                close(fd); //CLOSE the device
                printf("Good bye!\n");
                return 0;

            default:
                printf("Invalid input %c \n", option);
                break;
        }
    }
};
}

```

- Ta gọi tham số **fd** là tham số trả về của hàm open, nếu **fd** là giá trị âm có nghĩa là device file không mở được và sẽ thoát chương trình.

- Tham số **ret** sẽ là tham số đọc các byte của số ngẫu nhiên 4 byte trong device file, đây sẽ là kết quả trả về của số ngẫu nhiên và in ra console menu của user.

- Ta vào terminal và biên dịch file menu bằng cách gõ **make test**

```

nguyentathung943@ubuntu:~/Desktop/driver$ make test
cc -o u_test u_test.c

```

- Sau khi chương trình biên dịch thành công, ta gõ **make status** (hay **sudo ./u_test**) để vào menu

```

nguyentathung943@ubuntu:~/Desktop/driver$ make status
clear
sudo ./u_test
WELCOME TO THE DRIVER DEVICE INTERFACE
Make your own choices:
-----
*Press 1 to open the device file
*Press 2 to read the random 4 bytes number in the device file
*Press 3 to exit device file
You choose:

```

- Ta chọn option 1:

```
WELCOME TO THE DRIVER DEVICE INTERFACE
Make your own choices:
-----
*Press 1 to open the device file
*Press 2 to read the random 4 bytes number in the device file
*Press 3 to exit device file
You choose: 1
Device file opened!
You choose: █
```

→ Thông báo mở device file thành công.

- Tiếp đến ta chọn option 2 để đọc số ngẫu nhiên trong file và có kết quả trả về của biến **ret** sau khi đọc những byte được lưu trong **buffer** của entry point read

```
WELCOME TO THE DRIVER DEVICE INTERFACE
Make your own choices:
-----
*Press 1 to open the device file
*Press 2 to read the random 4 bytes number in the device file
*Press 3 to exit device file
You choose: 1
Device file opened!
You choose: 2
Reading from the device...
Random number is: 1282957116
You choose:
```

- Ta chọn option 3 để thoát khỏi device files

```
WELCOME TO THE DRIVER DEVICE INTERFACE
Make your own choices:
-----
*Press 1 to open the device file
*Press 2 to read the random 4 bytes number in the device file
*Press 3 to exit device file
You choose: 1
Device file opened!
You choose: 2
Reading from the device...
Random number is: 1282957116
You choose: 3
Good bye!
nguyentathung943@ubuntu:~/Desktop/driver$ █
```

- Bây giờ, ta vào **dmesg** để kiểm tra toàn bộ tiến trình và số nguyên 4 byte được in trong đây

```
2294.315352] Allocated device number (241,0)
2294.315633] Device file has been created
2294.315634] Data structure allocated successfully
2294.315634] Virtual character device initialized successfully
2294.315635] Entry points added successfully
2294.315639] Itnitialized character driver successfully
2318.848663] character_driver: Device is opening....
2340.840114] character_driver: Random number is : 1282957116
2366.428860] character_driver: Device successfully closed
nguyentathung943@ubuntu:~/Desktop/driver$
```

- Ta có thể thấy toàn bộ tiến trình driver trong đây bao gồm cả open và release device file, bao gồm cả số ngẫu nhiên 4 byte. Điều này giúp cho giao tiếp giữa user và driver trở nên đơn giản hơn

→ Ta hoàn thành toàn bộ quá trình xây dựng driver và module đọc và xuất các số ngẫu nhiên ra user-space

III. Tài liệu tham khảo:

<https://vimentor.com/vi/lesson/gioi-thieu-character-driver>

<https://vimentor.com/vi/lesson/cap-phat-dong-device-number-1>

<https://vimentor.com/vi/lesson/device-file>

<https://vimentor.com/vi/lesson/cap-phat-bo-nho-va-khoi-tao-thiet-bi>

<https://vimentor.com/vi/lesson/entry-point-cua-device-driver>

<https://vimentor.com/vi/lesson/ham-xu-ly-ngat>

https://github.com/nhathuy13598/Kernel_Random_Numbers

<https://xemtailieu.com/tai-lieu/tim-hieu-linux-kernel-cac-thanh-phan-dich-vu-49441.html?fbclid=IwAR0xnAr2SbZqfK-h31Nb4OgypWxZoa1kF585BTxhMLraeDjCHxIfsRSbiyg>

<https://stackoverflow.com/questions/57987140/difference-between-interrupt-context-and-process-context>