BAN CƠ YẾU CHÍNH PHỦ

**HỌC VIỆN KỸ THUẬT MẬT MÃ**

**LẬP TRÌNH DRIVER**

**ĐỀ TÀI: CAESAR CIPHER**

**Khoa: Công nghệ thông tin.**

**Giảng viên hướng dẫn: Triệu Văn Vũ Quân**

**Sinh viên thực hiện: Cao Thị Thùy Linh**

**Hồ Sỹ Phi**

**Trương Việt Hoàng**

**HÀ NỘI – 2021**



Mục lục

[1. TỔNG QUAN VÀ PHÂN TÍCH THUẬT TOÁN 3](#_Toc69049326)

[1.1. Giới thiệu Character Device 3](#_Toc69049327)

[1.2. Tổng quan mã hóa và giải mã dữ liệu 3](#_Toc69049328)

[1.3. Định nghĩa mật mã Caesar(Theo wikipedia) 3](#_Toc69049329)

[1.4. Phân tích thuật toán Caesar Cipher(Theo wikipedia) 3](#_Toc69049330)

[2. PHƯƠNG PHÁP VÀ QUY TRÌNH VIẾT CHARACTER DEVICE DRIVER(STATIC VÀ DYNAMIC METHOD) 5](#_Toc69049331)

[2.1. Quy trình thao tác với character device driver 5](#_Toc69049332)

[2.1.1. Quy trình thao tác với character device driver 5](#_Toc69049333)

[2.1.2. Ánh xạ giữa device file- driver- device 6](#_Toc69049334)

[2.2. Cấp phát tĩnh driver number 6](#_Toc69049335)

[2.2.1. Biểu diễn device number 6](#_Toc69049336)

[2.2.2. Cấp phát tĩnh device number 7](#_Toc69049337)

[2.2.3. Hủy đăng ký device number với kernel 8](#_Toc69049338)

[2.2.4. Ví dụ 8](#_Toc69049339)

[2.2.5. Tổng kết về cấp phát tĩnh 10](#_Toc69049340)

[2.3. Cấp phát động driver number 11](#_Toc69049341)

[2.4. Tạo file thiết bị 11](#_Toc69049342)

[2.4.1. Tạo file thiết bị thủ công 11](#_Toc69049343)

[2.4.2. Tạo file thiết bị tự động 11](#_Toc69049344)

[2.4.3. Cấp phát bộ nhớ và khởi tạo thiết bị 15](#_Toc69049345)

[2.5. Đăng ký các điểm vào (Entry point) 15](#_Toc69049346)

[2.5.1. Đăng ký entry point open và release 15](#_Toc69049347)

[2.5.2. Đăng ký entry point read và write 16](#_Toc69049348)

[3. PHẦN CODE 19](#_Toc69049349)

[3.1. Makefile 19](#_Toc69049350)

[3.2. Kbuild 19](#_Toc69049351)

[3.3. driver\_module.c 19](#_Toc69049352)

[3.4. user\_test.c 24](#_Toc69049353)

[4. DEMO & KẾT QUẢ THỰC HIỆN 26](#_Toc69049354)

## 

## 1. TỔNG QUAN VÀ PHÂN TÍCH THUẬT TOÁN

### 1.1. Giới thiệu Character Device

Character Device (chardev) là các device được truy cập như một luồng nhị phân (byte stream - tương tự như các file trong máy tính), và character device driver có nhiệm vụ thực hiện những thao tác đọc ghi này. Đối với một character device driver, ít nhất 4 system call: open, close, read và write cần được implement để chardev có thể hoạt động một cách bình thường.

Một số ví dụ về character device là: Serial console và console.

Do Linux về cơ bản là file, file và file nên các character device cũng không phải là ngoại lệ, chúng được truy cập bằng các file system node (trong /dev directory). Tuy nhiên, không giống như các file thông thường khác (có thể truy cập tiến, lùi tùy ý), các file dev này chỉ được truy cập theo một cách tuần tự

### 1.2. Tổng quan mã hóa và giải mã dữ liệu

Mã hóa dữ liệu là tiến trình che dấu dữ liệu thật (plaintext), nghĩa là chuyển dữ liệu thật thành dữ liệu không có ý nghĩa hoặc có ý nghĩa khác xa với dữ liệu thật. Tiến trình đó gọi là mã hóa (encryption). Kết quả của tiến trình gọi là bản mã (ciphertext). Từ “encryption” được tạo ra từ “cryptography” (mật mã) xuất phát từ tiếng Hy Lạp cổ xưa “Kryptos” (Che dấu) và từ “graphia” (viết). Tiến trình mã hóa dữ liệu có thế được thực hiện bằng cách hoán vị dữ liệu thật hoặc thay thế chúng bằng dữ liệu khác.

Tiến trình ngược với tiến trình mã hóa tức là chuyển từ bản mã thành dữ liệu ban đầu gọi là giải mã.

### 1.3. Định nghĩa mật mã Caesar(Theo wikipedia)

Trong mật mã học, **Mật mã Caesar** (hay còn được gọi là **Mật mã của Caesar**, **Mật mã chuyển vị**, **Mã của Caesar** hay **Chuyển vị Caesar**) là một trong những kỹ thuật mã hóa đơn giản và phổ biến nhất. Đây là một dạng mật mã thay thế, trong đó mỗi ký tự trên văn bản thô sẽ được thay bằng một ký tự khác, có vị trí cách nó một khoảng xác định trong bảng chữ cái. Ví dụ với độ dịch chuyển là 3, D sẽ trở thành A, E sẽ trở thành B, v.v. Tên của kỹ thuật mã hóa này được đặt theo tên của Julius Caesar, người đã sử dụng nó trong các thư từ bí mật của mình.

### 1.4. Phân tích thuật toán Caesar Cipher(Theo wikipedia)

Mô tả cách thay thế các ký tự trong một bộ mật mã Caesar có thể thực hiện bằng cách sắp xếp hai bảng chữ cái trên hai hàng song song với nhau; bảng chữ cái mật mã sẽ là bảng chữ cái thô đã được dịch sang trái hoặc sang phải một số vị trí. Ví dụ, dưới đây là một bộ mật mã Caesar được thiết lập bằng phép dịch sang trái 3 vị trí, tương đương với phép dịch sang phải 23 vị trí (con số vị trí dịch này được sử dụng làm khóa mã):

|  |  |
| --- | --- |
| Thô | *A B C D E F G H I J K L M N O P Q R S T U V W X Y Z* |
| Mật Mã | *D E F G H I J K L M N O P Q R S T U V W X Y Z A B C* |

Khi tiến hành mã hóa, người gửi mật mã sẽ tra cứu từng ký tự của tin nhắn gốc trên dòng "thô" và sau đó viết ra ký tự tương ứng lấy từ dòng "mật mã".

|  |  |
| --- | --- |
| Văn bản thô | *THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG* |
| Văn bản mật mã | *QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD* |

Quá trình giải mã của người nhận mật mã được thực hiện ngược lại, với thao tác dịch sang phải 3 vị trí.

Mã hóa cũng có thể được biểu diễn thông qua số học mô đun, bằng cách gán các ký tự bằng các con số, theo tuần tự, A → 0, B → 1,..., Z → 25. Mã hóa một chữ cái *x* bằng phép dịch chuyển *n* vị trí có thể mô tả bằng biểu thức toán học sau:

|  |
| --- |
| En(x)=(x+n)\mod {26}. |

Giải mã được mô tả tương tự,

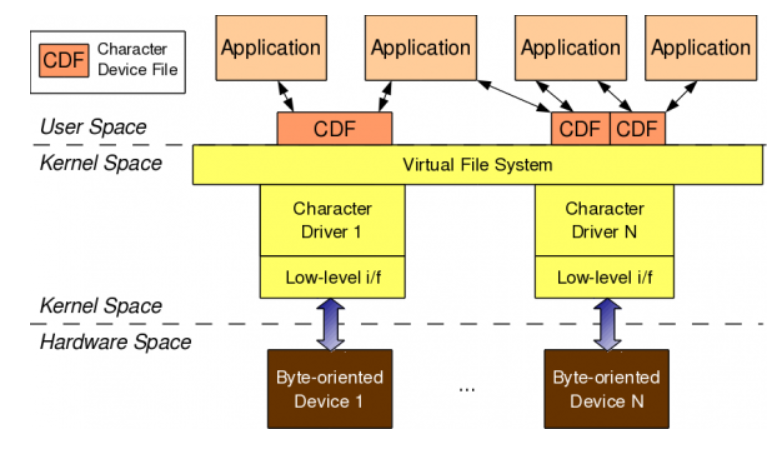
|  |
| --- |
| Dn(x)=(x-n)\mod {26} |

(Có nhiều định nghĩa cho phép toán Modulo. Trong trường hợp trên, kết quả phải nằm trong khoảng từ 0 đến 25. Do đó nếu *x+n* hoặc *x-n* không nằm trong đoạn 0...25, ta phải cộng hoặc trừ nó với 26.)

Loại mã hóa này có các giải pháp thay thế của từng ký tự là không đổi trong suốt quá trình mã hóa tin nhắn, vì vậy nó được xếp vào dạng *thay thế một bảng chữ cá*i, khác với *thay thế nhiều bảng chữ cái*.

## 2. PHƯƠNG PHÁP VÀ QUY TRÌNH VIẾT CHARACTER DEVICE DRIVER(STATIC VÀ DYNAMIC METHOD)

### 2.1. Quy trình thao tác với character device driver



*Tổng quan về thao tác với charcter device driver*

#### 2.1.1. Quy trình thao tác với character device driver

Bất kỳ một ứng dụng nào ở tầng người dùng (user space) muốn thao tác với một thiết bị kiểu character device trong tầng phần cứng (hardware space) sẽ sử dụng character device driver tương ứng trong tầng nhân (kernel space)

Việc sử dụng các character driver được thực hiện thông qua các file thiết bị (device files) tương ứng, được liên kết với driver thông qua hệ thống file ảo (virtual file system – VFS). Điều này có nghĩa là các ứng dụng có thể thực hiện các thao tác file thông thường trên các file thiết bị. Các thao tác file này sẽ được VFS diễn giải ra các hàm tương ứng trong driver liên kết với nó. Các hàm này sau đó sẽ thực hiện các truy cập ở mức thấp đến các thiết bị thật sự để đạt được kết quả mong muốn.

Tuy nhiên, cần lưu ý rằng mặc dù các ứng dụng truy cập đến các file thiết bị sử dụng các thao tác file thông thường như đối với file dữ liệu (mở, đọc, ghi, đóng, …), nhưng hiệu quả là khác so với các thao tác thông thường trên file dữ liệu. Ví dụ việc đọc dữ liệu từ thiết bị sẽ không là dữ liệu đã được ghi ra và ngược lại …

Việc kết nối từ ứng dụng đến thiết bị được thực hiện hoàn chỉnh thông qua 4 thực thể chính liên quan gồm:

* Application (ứng dụng)- đọc ghi thiết bị
  + Tạo file thiết bị
  + Làm việc với file thiết bị thông qua system call: open, close,

read, write, v.v.

* Character device file (File thiết bị) làm việc với driver thông qua các Entry point
  + Entry point: là các thao tác driver đã đăng ký để làm việc với nhân hệ điều hành
* Character device driver (Driver thiết bị)
  + Các hàm mức thấp (device specific) của driver được Entry point yêu cầu, và sẽ làm việc với device
* Character device (Thiết bị)

#### 2.1.2. Ánh xạ giữa device file- driver- device

* Với việc tạo ra device file:
  + Cho các tiến trình biết các char/block device cũng chỉ là các file
  + Đọc/ghi dữ liệu từ thiết bị giống như đọc/ghi dữ liệu từ file thông thường
* Device file ánh xạ với driver thông qua Device number
* Device number là một thuộc tính quan trọng của device file; gồm 2 số:
  + 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.
* Mã nguồn của char driver được tổ chức thành 2 phần chính:
  + Phần OS specific: gồm các hàm khởi tạo/kết thúc driver và các hàm entry point của driver.
  + Phần device specific: gồm 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.

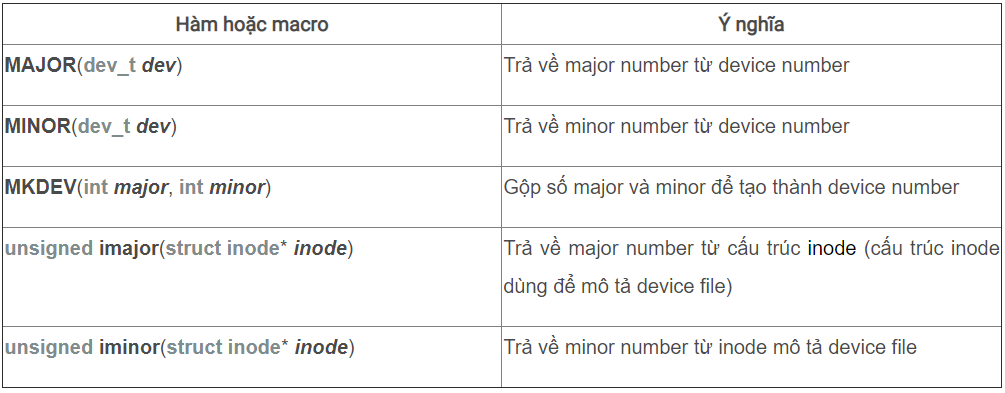
### 2.2. Cấp phát tĩnh driver number

#### 2.2.1. Biểu diễn device number

* Linux kernel sử dụng cấu trúc dev\_t để biểu diễn

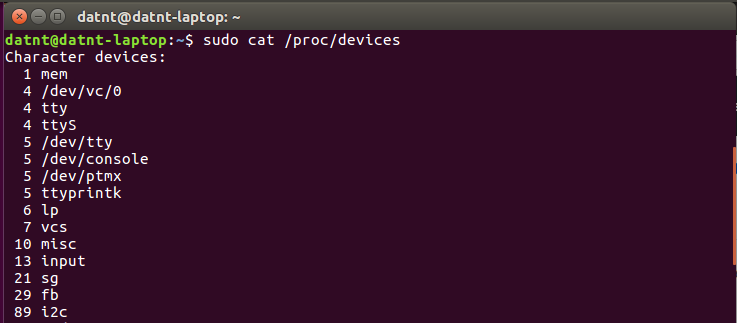
device number với kích thước 32 bit

* major number chiếm 12 bits
* minor number chiếm 20 bits



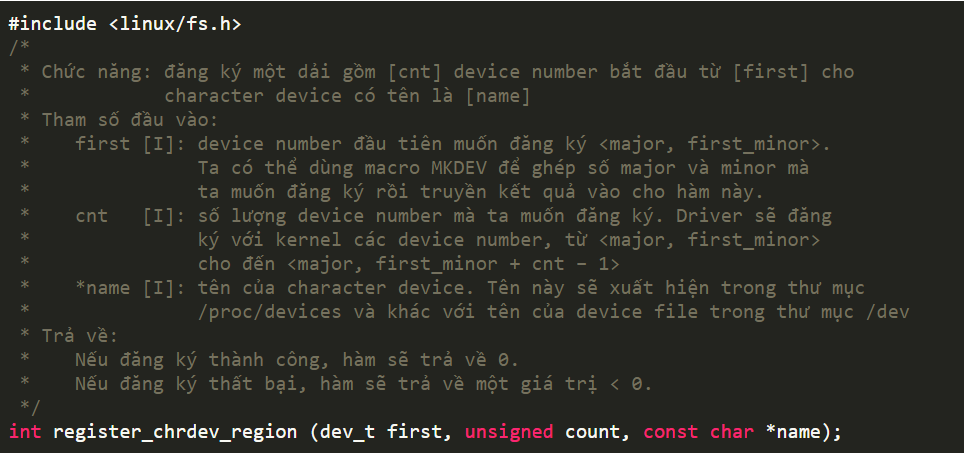
#### 2.2.2. Cấp phát tĩnh device number

Trong phương pháp này, lập trình viên sẽ chọn trước một giá trị làm major number. Để biết được nên chọn giá trị nào, ta gõ lệnh “cat /proc/devices” (hình dưới). Cột đầu tiên chứa các major number đã được sử dụng bởi các driver khác. Vì thế, ta không chọn những số này. Ngoại trừ những số này, ta có thể chọn bất cứ số nào trong khoảng từ 0 đến (2^12 - 1) làm major number.



*Danh sách các thiết bị đang hoạt động trong hệ thống*

Sau khi đã chọn được major number, chúng ta sẽ gọi hàm *register\_chrdev\_region* để đăng ký device number với kernel. Nếu giá trị này vẫn chưa được sử dụng bởi driver nào, thì kernel sẽ cấp phát giá trị đó cho driver và quá trình đăng ký sẽ thành công. Ngược lại, nếu đã có driver sử dụng giá trị đó rồi, thì kernel sẽ từ chối cấp phát và quá trình đăng ký thất bại.



#### 2.2.3. Hủy đăng ký device number với kernel

Khi không còn dùng đến device number nữa, chúng ta cũng cần giải phóng nó. Hàm giải phóng device number thường được đặt trong hàm kết thúc của char driver. Hàm giải phóng device number có nguyên mẫu như sau:



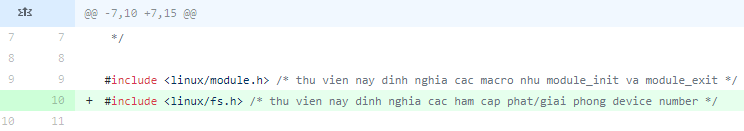
#### 2.2.4. Ví dụ

Ví dụ này sẽ giúp các bạn hiểu được cách thức cấp phát tĩnh device number trong quá trình khởi tạo một char driver. Đầu tiên, ta tạo thư mục cho bài học hôm nay:

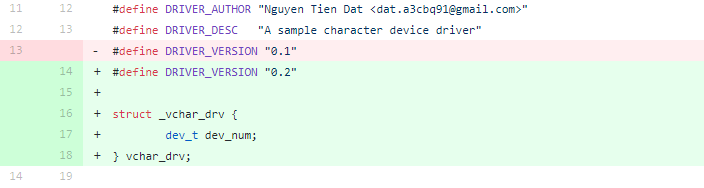


Ví dụ này dựa trên ví dụ trong bài Giới thiệu về Character driver. Thư mục này vẫn gồm 3 file. Hai file Makefile và Kbuild không có gì thay đổi. Riêng file *vchar\_driver.c,* ta sửa lại một chút.

Để sử dụng được các hàm cấp phát/giải phóng device number, ta cần tham chiếu tới thư viện *<linux/fs.h>.*



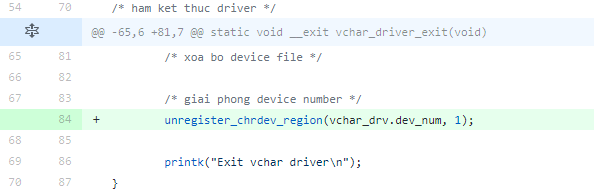
Sau khi kiểm tra /proc/devices, ta thấy rằng chưa có driver nào sử dụng số 235 làm major number, nên ta chọn bộ số <235, 0> làm device number. Để lưu giá trị device number này, ta sẽ tạo ra một cấu trúc *vchar\_drv* chứa trường *dev\_num.*



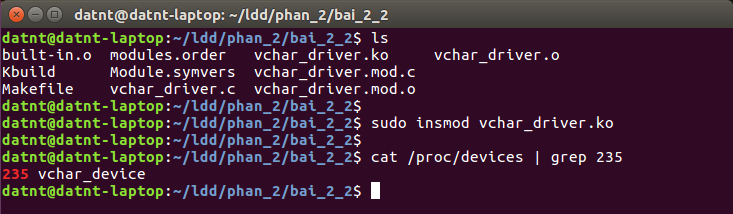
Trong hàm *vchar\_driver\_init* của driver này, ta sẽ sử dụng macro MKDEV để khởi tạo giá trị cho trường *dev\_num* của cấu trúc *vchar\_drv*. Sau đó, ta gọi hàm *register\_chrdev\_region* để đăng ký device number với Linux kernel. Nếu quá trình đăng ký thất bại, hàm này trả về một giá trị âm. Khi đó, ta sẽ trả về một số âm để thông báo với Linux kernel rằng: không thể lắp char driver này vào kernel được.



Cuối cùng, trong hàm *vchar\_driver\_exit* của driver này, ta gọi hàm *unregister\_chrdev\_region* để giải phóng device number đó.



Bây giờ, ta gõ lệnh *make* để biên dịch char driver. Sau khi biên dịch thành công, ta sử dụng lệnh insmod để lắp driver vào trong kernel. Lúc này, ta thấy xuất hiện một dòng chứa “235 vchar\_device” trong file /proc/devices. Tuy nhiên, chú ý rằng, ta vẫn không thấy có thêm device file trong thư mục /dev. Lý do là vì chúng ta mới chỉ xin Linux kernel cấp phát device number, chứ chưa tạo device file tương ứng với char driver này.



*Xuất hiện thêm thông tin trong /proc/devices sau khi lắp vchar\_driver.ko vào kernel*

#### 2.2.5. Tổng kết về cấp phát tĩnh

Linux kernel sử dụng cấu trúc dev\_t để biểu diễn device number. Cấu trúc này có kích thước 32 bit. Trong đó, major number chiếm 12 bit, còn lại là minor number.

Có 2 cách thức cấp phát device number, là cấp phát tĩnh và cấp phát động. Bài học này hướng dẫn các bạn cách cấp phát tĩnh. Các bước làm như sau:

* Bước 1: chọn một số không có trong /proc/devices làm major number.
* Bước 2: sử dụng macro MKDEV để tạo ra số device number.
* Bước 3: gọi hàm *register\_chrdev\_region* để đăng ký số device number với kernel.

Khi tháo char driver ra khỏi kernel, thì device number gắn với driver đó không còn được sử dụng nữa. Do đó, ta cần gọi hàm *unregister\_chrdev\_region* để giải phóng device number đó. Ta nên đặt hàm *unregister\_chrdev\_region* bên trong hàm kết thúc của char driver.

### 2.3. Cấp phát động driver number

Thông thường, lập trình viên sẽ tạo ra char driver trên một máy tính, và char driver đó sẽ được sử dụng trên nhiều máy tính khác. Nếu lựa chọn phương pháp cấp phát tĩnh device number, thì device number đó có thể đã được sử dụng trên những máy tính khác, dẫn tới char driver không hoạt động được trên các máy tính ấy. Để giải quyết vấn đề này, lập trình viên nên sử dụng phương pháp cấp phát động device number.

Trong phương pháp này, 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. Ta thường gọi hàm này trong hàm khởi tạo của char driver.

### 2.4. Tạo file thiết bị

* Tiến trình ứng dụng làm việc với driver thông qua

các file thiết bị

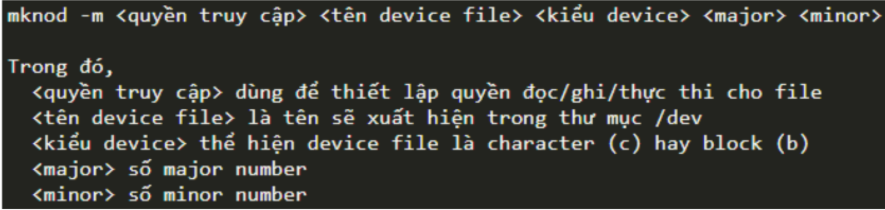
* Các tiến trình sẽ đọc/ghi dữ liệu từ thiết bị cũng

giống như đọc/ghi dữ liệu từ file thông thường

* File thiết bị nằm trong thư mục /dev
* 2 cách tạo file thiết bị:
* Tạo device file một cách thủ công
* Tạo device file một cách tự động

#### 2.4.1. Tạo file thiết bị thủ công

* Sử dụng công cụ **mknod**



* Ví dụ



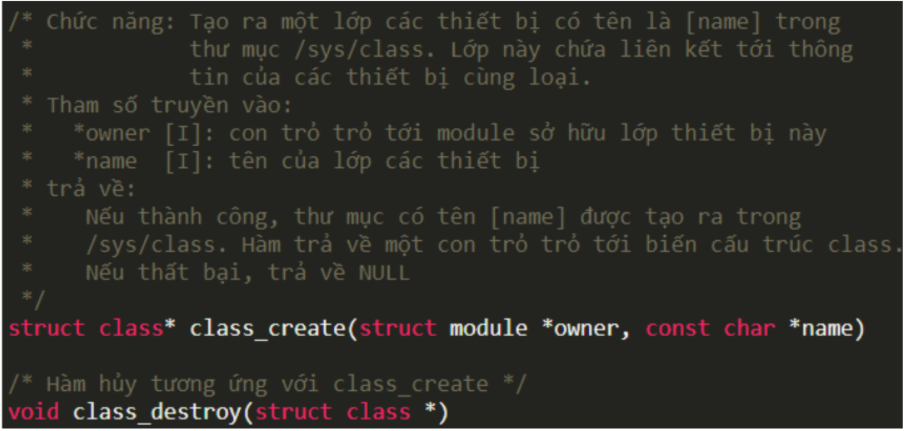
* Chú ý: Tên file thiết bị sẽ được các tiến trình ứng dụng sử dụng để truy xuất

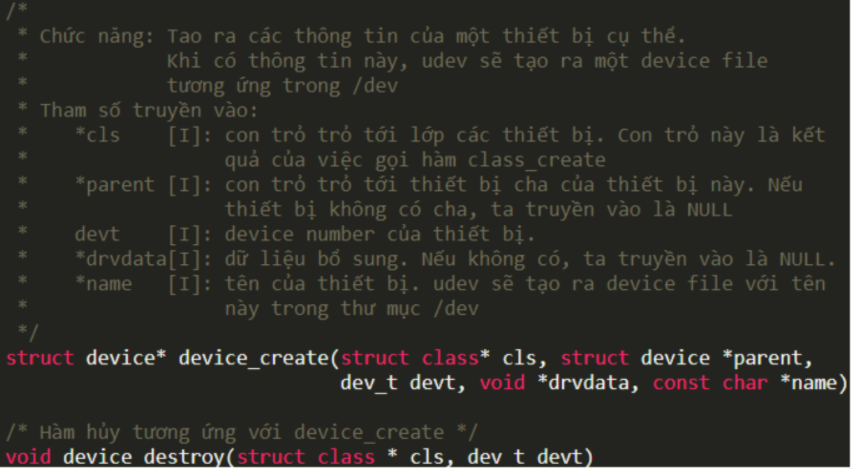
#### 2.4.2. Tạo file thiết bị tự động

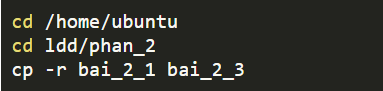
* Dựa vào tiến trình **udevd** để tạo/hủy các device file trong thư mục /dev
* Khi viết char driver, sẽ sử dụng một số hàm của Linux kernel để gửi sự kiện lên cho udevd
  + Các sự kiện được gọi là uevent (user event)
  + Sau khi nhận được uevent, udevd sẽ tạo ra một device file trong

thư mục /dev.

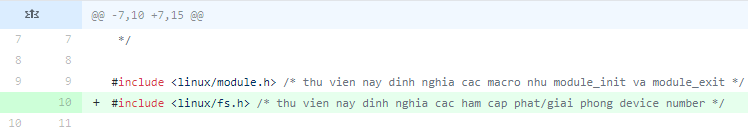
* Để triển khai phương pháp này, ta thực hiện 3 bước sau:
  + Tham chiếu tới thư viện *<linux/device.h>*
  + Tạo một lớp các thiết bị
  + Tạo thiết bị trong lớp
* Để tạo một lớp các thiết bị, sử dụng hàm *class\_create* của Linux Kernel



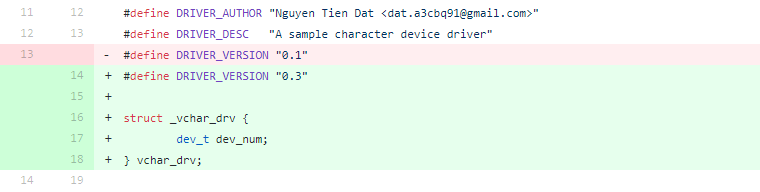
* Để tạo thiết bị trong lớp trên, sử dụng hàm *device\_create* của LinuxKernel
* Ví dụ này sẽ trình bày cách triển khai kỹ thuật cấp phát động device number. Đầu tiên, ta tạo thư mục cho bài học hôm nay:



Ví dụ này dựa trên ví dụ trong bài Giới thiệu character driver. Thư mục này vẫn gồm 3 file. Hai file Makefile và Kbuild không có gì thay đổi so với bài trước. Riêng file *vchar\_driver.c*, ta sửa lại một chút. Như đã trình bày, để sử dụng được các hàm cấp phát/giải phóng device number, ta cần tham chiếu tới thư viện *<linux/fs.h>.*

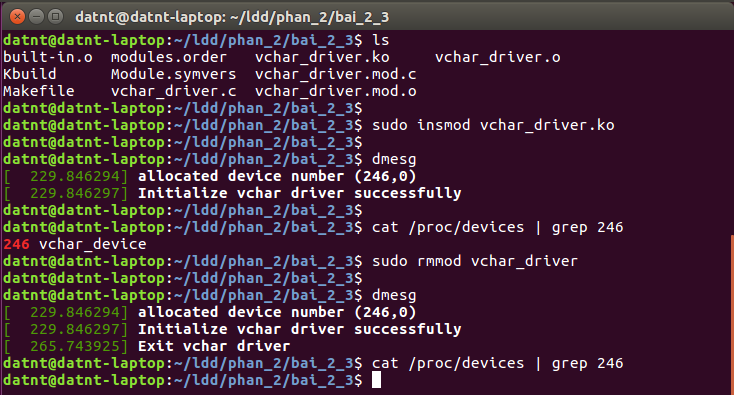


Để lưu giá trị device number, ta sẽ tạo ra một cấu trúc vchar\_drv chứa trường *dev\_num (kiểu dev\_t).*



Trong hàm *vchar\_driver\_init* của driver này, ta sẽ gọi hàm *alloc\_chrdev\_region* để yêu cầu kernel tìm kiếm một device number phù hợp. Nếu không thể tìm thấy một device number nào, hàm này trả về một giá trị âm. Khi đó, ta sẽ trả về một số âm để thông báo với Linux kernel rằng:không thể lắp char driver này vào kernel được.

Bây giờ, ta gõ lệnh make để biên dịch char driver. Sau khi biên dịch thành công, ta sử dụng lệnh insmod để lắp driver vào trong kernel. Lúc này, ta thấy xuất hiện một *dòng chứa “246 vchar\_device” trong file /proc/devices*

*Xuất hiện thêm thông tin trong /proc/devices sau khi lắp vchar\_driver.ko vào kernel* 

* Kết luận:

Trong quá trình viết char driver, lập trình viên nên sử dụng phương pháp cấp phát động device number. Mục đích là để char driver đó có thể chạy được trên nhiều máy tính khác nhau. Để thực hiện cấp phát động device number, ta sẽ gọi hàm *alloc\_chrdev\_region* bên trong hàm khởi tạo của char driver.

#### 2.4.3. Cấp phát bộ nhớ và khởi tạo thiết bị

* 2 thao tác khác cần phải thực hiện trong quá trình

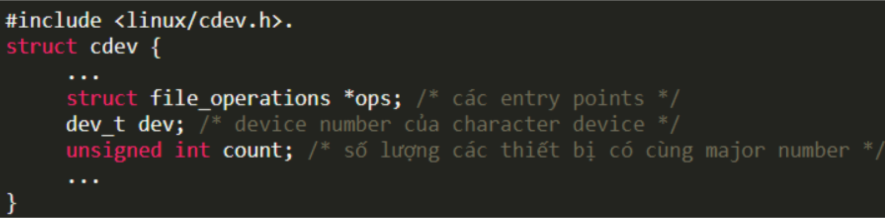
khởi tạo char driver

* + Cấp phát bộ nhớ cho các cấu trúc của char driver.
  + Khởi tạo thiết bị
* Cấp phát bộ nhớ
  + cấp phát bộ nhớ dưới kernel space, ta thường dùng hàm kmalloc
  + muốn cấp phát N byte dưới kernel space, thực hiện:
    - *#include <linux/slab.h>*
    - *kmalloc(N, GFP\_KERNEL);*
  + Một hàm khác thường dùng hơn là kzalloc
    - Cấp phát và khởi tạo 0
  + Giải phóng bộ nhớ (cấp phát theo *kmalloc/kzalloc*)
    - Gọi hàm kfree với tham số truyền vào là địa chỉ của vùng nhớ

### 2.5. Đăng ký các điểm vào (Entry point)

#### 2.5.1. Đăng ký entry point open và release

* Cấu trúc **cdev**
  + Để mô tả một character device



* Linux kernel cung cấp các hàm để đọc hoặc ghi

giá trị vào các trường của cdev

* + Nên sử dụng các hàm này, thay vì đọc/ghi trực tiếp

**2.5.1. Đăng ký entry point open và release**

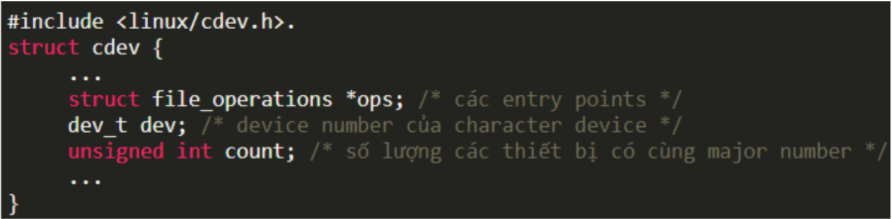
open(): thực hiện các khởi tạo cơ bản để giúp các tác vụ khác có thể hoạt động sau đó. Thông thường, hàm open() sẽ thực hiện các nhiệm vụ sau:

* Kiểm tra xem device đã sẵn sàng chưa? Hardware có vấn đề gì không?…
* Khởi tạo device nếu nó được mở lần đầu.
* Cập nhật f\_op nếu cần tiếp.
* Cấp phát và gán các thông tin cần thiết vào filp->private\_data.

Tuy nhiên, mục tiêu hàng đầu là xác định xem device nào sẽ được mở. Hiện tại chưa cần đến hàm này, chỉ cần định nghĩa 1 hàm thân rỗng là được.

release(): Hàm này dùng để phá hoại hết những gì đã làm trong hàm open. Đầu tiên là phải thu deallocate filp->private\_data. Poweroff device trong lần dùng cuối. trong scull hàm này không làm gì cả vì không có gì để giải phóng hay power off hết. Trong kernel, có một counter dùng để đếm xem một *file* structure có bao nhiêu đối tượng đang sử dụng nó. Khi counter bằng này có giá trị bằng 0 thì đó được xem là lần sử dụng cuối của device và nó sẽ bị poweroff. Ngoài ra counter cũng đảm bảo là mỗi lời gọi đến open() sẽ chỉ có một lời gọi đến release() đi kèm (tránh release 1 file 2 lần).  
Hiện tại chúng ta chưa cần đến hàm này, nên chỉ cần định nghĩa 1 hàm thân rỗng là được.

* Cấu trúc **cdev**
  + Để mô tả một character device



#### 2.5.2. Đăng ký entry point read và write

**ssize\_t read(struct file \*filp, char \_\_user \*buff, ssize\_t count, loff\_t \*offp);**

**ssize\_t write(struct file \*filp, const char \_\_user \*buff, ssize\_t count, loff\_t \*offp);**

Hàm này sẽ gửi một buffer có kích thước count bytes tới user space bắt đầu tự vị trí offp của file. Do buff là user-space pointer nên nó không thể được truy vấn một cách trực tiếp từ kernel code, sau đây là một số hạn chế:

* Phụ thuộc vào kiến trúc của hệ thống và các cài đặt cấu hình của kernel, user-space pointer có thể là không hợp lệ đối với kernel mode. (Do kernel memory là direct mapping, trong khi ở user-space không phải là direct mapping nên cùng một địa chỉ nhưng vị trí sẽ khác nhau).
* User-mem được paged (paging) nên nó không tồn tại lâu dài trong RAM. Việc tham chiếu đến vùng nhớ user-space một cách trực tiếp sẽ gây ra page fault (không phải lúc nào cũng xảy ra nhưng xác suất cao) kể cả nếu pointer trong kernel-space và user-space có cách mapping giống nhau.
* Về vấn đề bảo mật, việc tham chiếu trực tiếp đến pointer của user-space cũng không tốt vì nó tạo ra nguy cơ bảo mật cao.

Mặc dù có những hạn chế ở trên, nhưng rõ ràng là chúng ta vẫn cần truy cập đến user-space buffer để người dùng có thể tương tác với device. Kernel cung cấp các hàm để thực hiện điều này một cách an toàn. Những hàm này được định nghĩa trong header **linux/uaccess.h.** Những hàm này đã sử dụng ma thuật hắc ám của kẻ mà ai cũng biệt là ai để truyền dữ liệu giữa kernel và user space một cách an toàn và im lặng. Trong phần read(), write() chúng ta cần đến phép thuật sau: **usigned long copy\_to\_user(void \_\_user \*to, const void \*from, usinged long count);** **usigned long copy\_from\_user(void \_\_user \*to, const void \_\_user \*from, usinged long count);** Lưu ý là do user-space sử dụng cơ chế paging/swapping nên tại thời điểm bất kỳ, có thể page cần dùng để copy/send data không nằm trong bộ nhớ, do đó cần có thời gian để transfer các page này vào mem, điều này đồng nghĩa với việc các hàm read/write phải sleepable, nên các hàm này sẽ thực hiện một cách đồng thời với các hàm khác của driver.  
Hai hàm này không phải là atomic, tức là có thể chỉ một phần khối lượng công việc được hoàn thành. Cụ thể hơn, việc copy sẽ không được thực hiện nếu như user-space pointer không hợp lệ, trường hợp ngược lại, việc copy sẽ được thực hiện, nhưng giả dụ trong lúc đang copy nó phát hiện ra một địa chỉ không hợp lệ, quá trình copy sẽ bị break và phần data chưa copy sẽ không được xử lý, phần đã copy thì vẫn giữ nguyên. Giá trị trả về của các hàm này đều là lượng data đã copy (bytes). [Atomic nghĩ là chỉ có 2 trường hợp: chạy hết thành công, trường hợp 2 là chạy thất bại ở một bước nào đấy thì toàn bộ sẽ bị roll back, giống trong SQL].

* Cần update \*offp sau khi thực hiện read/write để đảm bảo rằng vị trí hiện tại là đúng.
* Nếu thao tác đọc/ghi không thành công thì giá trị trả về là 1 số ÂM. b1. read() Với mỗi giá trị trả về của hàm read(), có một tác động tương ứng lên chương trình (app space) có lời gọi hàm đến nó.
* Nếu giá trị trả về bằng với count thì toàn bộ dữ liệu yêu cầu đã được truyền thành công. Đây là trường hợp tối ưu.
* Nếu giá trị là dương, nhưng nhỏ hơn count, chỉ một phần của dữ liệu đã được truyền thành công. Điều này có thể xảy ra do một số thế lực hắc ám phụ thuộc vào pháp sư sử dụng nó (device). Trường hợp này thường xảy ra khi user-space program gọi đến read().
* Nếu giá trị là 0 thì không có data để truyền đi nữa (chakra cạn kiệt).
* Nếu giá trị trả về là 0, thì tức là nó đã bị phong ấn ở đâu đấy.
* Trường hợp cá biệt, chakra vẫn còn nhưng bị bakugan phong tỏa huyệt đạo, shinobi sẽ rơi vào trang thái block. Mặc dù ở trên có đề cập việc thay đổi file offset, tuy nhiên ví dụ của chúng ta mong muốn là đọc ghi từ đầu file, nên không cần phải update nó làm gì cả, (cả read và write).

|  |
| --- |
| **static ssize\_t oni\_read(struct file \*filp, char \_\_user \*buffer, size\_t count, loff\_t \*offset)**  **{**  **int err\_count = 0;**  **err\_count = copy\_to\_user(buffer, msg, size\_of\_msg);**  **if( err\_count == 0 )**  **{**  **pr\_info( "Oni Chrdev: Sent %lu chars to the user\n", size\_of\_msg);**  **size\_of\_msg = 0;**  **return 0;**  **}else**  **{**  **pr\_info( "Oni Chrdev: Failed to send %d chars to the user\n", err\_count);**  **return –EFAULT;**  **}}** |

*Hàm read()*

**write()** giống read, write có thể truyền ít hơn dữ liệu được yêu cầu, sau đây là các giá trị trả về ở user-space calling tương ứng.

* Nếu giá trị trả về bằng count thì toàn bộ các bytes được yêu cầu đã truyền thành công.
* Nếu giá trị trả về là giá trị dương lớn hơn count, thì chỉ một phần chakra được truyền từ cửu vĩ sang naruto. Chương trình (user-space) gần như ngay lập tức cố gắng write phần data còn lại.
* Nếu giá trị trả về là 0 thì tức là không có ghì để write.
* Nếu giá trị trả về là âm thì đã có lỗi.

|  |
| --- |
| **static ssize\_t oni\_write(struct file \*filp, const char \_\_user \*buffer, size\_t count, loff\_t \*offset)**  **{**  **if(copy\_from\_user(msg, buffer, count))**  **{**  **return -EACCES;**  **}**  **size\_of\_msg = strlen(msg);**  **pr\_info( "Oni Chrdev: receive %zu charaters for the user: %s\n",count,msg);**  **return count;**  **}** |

*Hàm write()*

## 3. PHẦN CODE

Bắt đầu thực hiện code, cần cài :

* Cài đặt make: *$ sudo apt install make -y*
* Cài đặt gcc: *$ sudo apt install build-essential*
* Check gcc: *$ gcc --version*
* *$ sudo apt install linux-headers-$(uname -r)*
* *$ sudo apt-get install manpages-dev*

Đầu tiên chúng ta sẽ kể đến là 2 file là Makefile và Kbuild, ta sẽ dùng chúng như một câu lệnh nhúng dùng cho việc compile ra file .o và sau đó ra file .ko

### 3.1. Makefile

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

### 3.2. Kbuild

|  |
| --- |
| EXTRA\_CFLAGS = -Wall  obj-m = driver\_module.o |

### 3.3. driver\_module.c

|  |
| --- |
| #include <linux/init.h>  #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/device.h>  #include <linux/slab.h>  #include <linux/string.h>  #include <linux/ctype.h>  #include <linux/cdev.h>  #include <linux/uaccess.h>  #include <linux/fs.h>  #define MEM\_SIZE 1024  char \*kernel\_buffer;  char XauRo[MEM\_SIZE], XauMa[MEM\_SIZE], XauGiaiMa[MEM\_SIZE];  int i, key, choice;  char x; |

|  |
| --- |
| struct vchar\_drv  {  dev\_t dev\_num;  struct class \*dev\_class;  struct device \*dev;  struct cdev \*vcdev;  }driver\_module; |

|  |
| --- |
| static int my\_open(struct inode \*inode, struct file \*filp);  static int my\_release(struct inode \*inode, struct file \*filp);  static ssize\_t my\_read(struct file \*filp, char \_\_user \*user\_buf,size\_t len, loff\_t \*off);  static ssize\_t my\_write(struct file \*filp, const char \_\_user \*user\_buf, size\_t len, loff\_t \*off);  void MaHoa(char XauRo[MEM\_SIZE],char XauMa[MEM\_SIZE], int key);  void GiaiMa(char XauMa[MEM\_SIZE],char XauGiaiMa[MEM\_SIZE], int key); |

|  |
| --- |
| static struct file\_operations fops = {  .owner = THIS\_MODULE,  .read = my\_read,  .write = my\_write,  .open = my\_open,  .release= my\_release,  }; |

|  |
| --- |
| int my\_open(struct inode \*inode, struct file \*filp)  {  pr\_info("Driver: open\n");  return 0;  }  int my\_release(struct inode \*inode, struct file \*filp)  {  pr\_info("Driver: close\n");  return 0;  } |

|  |
| --- |
| ssize\_t my\_read(struct file \*filp, char \_\_user \*user\_buf, size\_t len, loff\_t \*off)  {  if(choice == 2) {  copy\_to\_user(user\_buf, XauMa, MEM\_SIZE);  }  if(choice == 3 )  {  copy\_to\_user(user\_buf, XauGiaiMa, MEM\_SIZE);  }  return MEM\_SIZE;  }  ssize\_t my\_write(struct file \*filp, const char \_\_user \*user\_buf, size\_t len, loff\_t \*off)  {  copy\_from\_user(kernel\_buffer, user\_buf, len);  choice = \*kernel\_buffer;  if(choice == 1)  {  i = 0;  kernel\_buffer++;  while(\*kernel\_buffer != '\0') {  XauRo[i++] = \*kernel\_buffer;  kernel\_buffer++;  }  XauRo[i] = '\0';  }    if(choice == 2)//encrypt  {  kernel\_buffer++;  key = \*kernel\_buffer;  key = key -22;  MaHoa(XauRo,XauMa,key);    }  if(choice == 3)//decrypt  {  kernel\_buffer++;  key = \*kernel\_buffer;  key = key -22;  GiaiMa(XauMa,XauGiaiMa,key);    }  return len;  } |

|  |
| --- |
| void MaHoa(char XauRo[MEM\_SIZE],char XauMa[MEM\_SIZE], int key)  {  len = strlen(XauRo);// do dai xau  for(i = 0; i < len; i++)  {  if(isalpha(XauRo[i]))//check chu  {  if(isupper(XauRo[i]))// la chu hoa  {  XauMa[i] = ((((XauRo[i] - 65) + key) % 26) + 65);  }  else if (islower(XauRo[i]))//la chu thuong  {  XauMa[i] = ((((XauRo[i] - 97) + key) % 26) + 97);  }  }  else  {  XauMa[i] = XauRo[i];  }  }  }  void GiaiMa(char XauMa[MEM\_SIZE],char XauGiaiMa[MEM\_SIZE], int key)  {  len = strlen(XauMa);// do dai xau  for(i = 0; i < len; i++)  {  if(isalpha(XauMa[i]))//check chu  {  if(isupper(XauMa[i]))// la chu hoa  {  XauGiaiMa[i] = (((((XauMa[i] - 65) + 26) - key) % 26) + 65);  }  else if (islower(XauRo[i]))//la chu thuong  {  XauGiaiMa[i] = (((((XauMa[i] - 97) + 26) - key) % 26) + 97);  }  }  else  {  XauGiaiMa[i] = XauMa[i];  }  }  } |

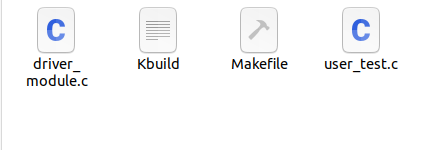
|  |
| --- |
| static int driver\_module\_init(void)  {  int ret = 0;  driver\_module.dev\_num = 0;    pr\_info("Bat dau driver\_module\n");  ret = alloc\_chrdev\_region(&driver\_module.dev\_num,0,1,"driver\_module");  if(ret < 0) {  printk("Can't allocate character driver\n");  goto failed\_register\_devnum;  }  printk("Insert character driver successfully. major(%d), minor(%d)\n", MAJOR(driver\_module.dev\_num), MINOR(driver\_module.dev\_num));    driver\_module.dev\_class = class\_create(THIS\_MODULE, "driver\_module\_class");  if(IS\_ERR(driver\_module.dev\_class)) {  printk("Can't create class\n");  goto failed\_create\_class;  }    driver\_module.dev = device\_create(driver\_module.dev\_class, NULL, driver\_module.dev\_num, NULL, "driver\_module\_device");  if(IS\_ERR(driver\_module.dev)) {  printk("Can't create device file\n");  goto failed\_create\_device;  }    kernel\_buffer = kmalloc(MEM\_SIZE, GFP\_KERNEL);    driver\_module.vcdev = cdev\_alloc();  cdev\_init(driver\_module.vcdev, &fops);  cdev\_add(driver\_module.vcdev, driver\_module.dev\_num,1);  return 0;    failed\_create\_device:  class\_destroy(driver\_module.dev\_class);  failed\_create\_class:  unregister\_chrdev\_region(driver\_module.dev\_num, 1);  failed\_register\_devnum:  return ret;  }  /\* ham giai phong thiet bi \*/  static void driver\_module\_exit(void)  {  cdev\_del(driver\_module.vcdev);/\* huy dang ky entry point voi kernel \*/  kfree(kernel\_buffer);/\* giai phong bo nho da cap phat \*/  /\* xoa bo device file \*/  device\_destroy(driver\_module.dev\_class, driver\_module.dev\_num);  class\_destroy(driver\_module.dev\_class);  unregister\_chrdev\_region(driver\_module.dev\_num, 1);  pr\_info("ket thu driver\_module \n");  } |

### 3.4. user\_test.c

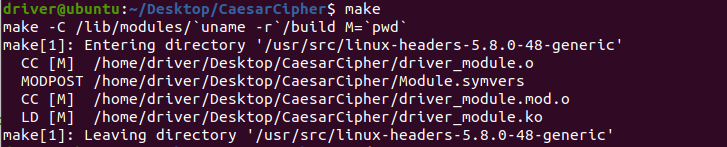
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  #define DEVICE\_NODE "/dev/driver\_module\_device"  #define MEM\_SIZE 1024  char write\_buf[MEM\_SIZE];  char read\_buf[MEM\_SIZE];  int key, i ;  int main()  {  int fd;  char opt;  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  printf("\*\*\*\*\*\*\*user application to test char driver\*\*\*\*\*\*\*\n");    fd = open(DEVICE\_NODE, O\_RDWR);    if(fd < 0)  {  printf("Cannot open device file ... \n");  return 0;  }      while(1)  {  printf("\n\*\*\*\*\*\*\*\*\*Please Enter the Option\*\*\*\*\*\*\*\n");  printf("\* 1. Nhap vao mot xau. \*\n");  printf("\* 2. Encrypt Caesar and save. \*\n");  printf("\* 3. Read and decrypt Caesar. \*\n");  printf("\* 4. Exit. \*\n");  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");    scanf("%c", &opt);  printf("\nYou Option: %c\n", opt);    switch(opt)  {  case '1':  \*write\_buf = 1;  printf("\nNhap xau ro: ");  scanf(" %[^\t\n]s", write\_buf + 1);// viết đc dấu cách  write(fd, write\_buf, strlen(write\_buf) + 1);  break;  case '2':  \*write\_buf = 2;  printf("\nNhap key de ma hoa: ");  scanf(" %s", write\_buf + 1);  write(fd, write\_buf, strlen(write\_buf)+1);  read(fd, read\_buf, MEM\_SIZE);  printf("\nXau sau khi ma hoa ---->> %s\n",read\_buf);  break;  case '3':  \*write\_buf = 3;  printf("\nNhap key de giai ma: ");  scanf("%s", write\_buf + 1);  write(fd, write\_buf, strlen(write\_buf) + 1);  read(fd, read\_buf, MEM\_SIZE);  printf("\nXau sau khi giai ma ---->> %s\n", read\_buf);  break;  case '4':  close(fd);  exit(1);  break;  default:  printf("Enter Valid option = %c\n",opt);  break;  }  }    return 0;  } |

## 4. DEMO & KẾT QUẢ THỰC HIỆN

* Sau khi tạo các file ở “Phần code”



* Mở terminal -> make



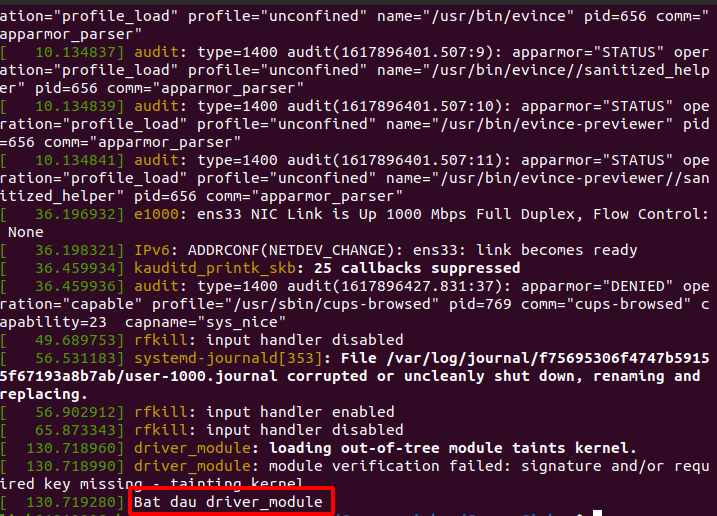
* Vào quyền root :$sudo su, gõ password
* Gõ: #insmod driver\_module.ko



* Gõ: #dmesg



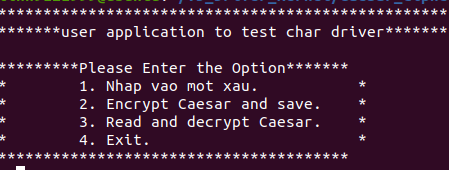
Lục đó sẽ hiển thị nội dung của file driver\_module.c



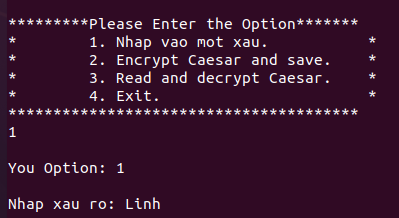
* Gõ: #user\_test.c -o test



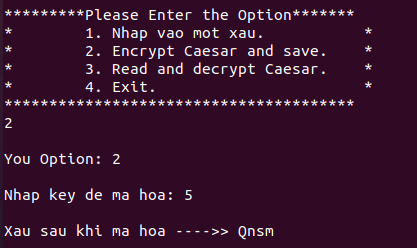
* Gõ: # ./test



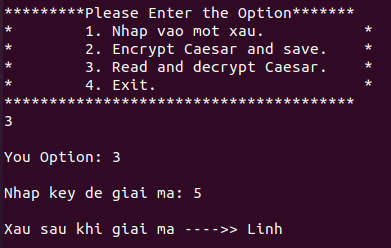
* Xuất hiện menu:
* Chọn ‘1’: nhập xâu rõ: vd :”Linh”



* Chọn ‘2’: Chọn key = 5, ta được xâu sau khi mã hóa



* Chọn ‘3’: Giải mã trên với key = 5



### 