BAN CƠ YẾU CHÍNH PHỦ

HỌC VIỆN KỸ THUẬT MẬT MÃ

PHÁT TRIỂN TRÌNH ĐIỀU KHIỂN CHO THIẾT BỊ TRUYỀN DỮ LIỆU DẠNG KÝ TỰ

Ngành: Công nghệ thông tin

Chuyên ngành: Kỹ thuật phần mềm nhúng và di động

Mã số: 52.48.02.01

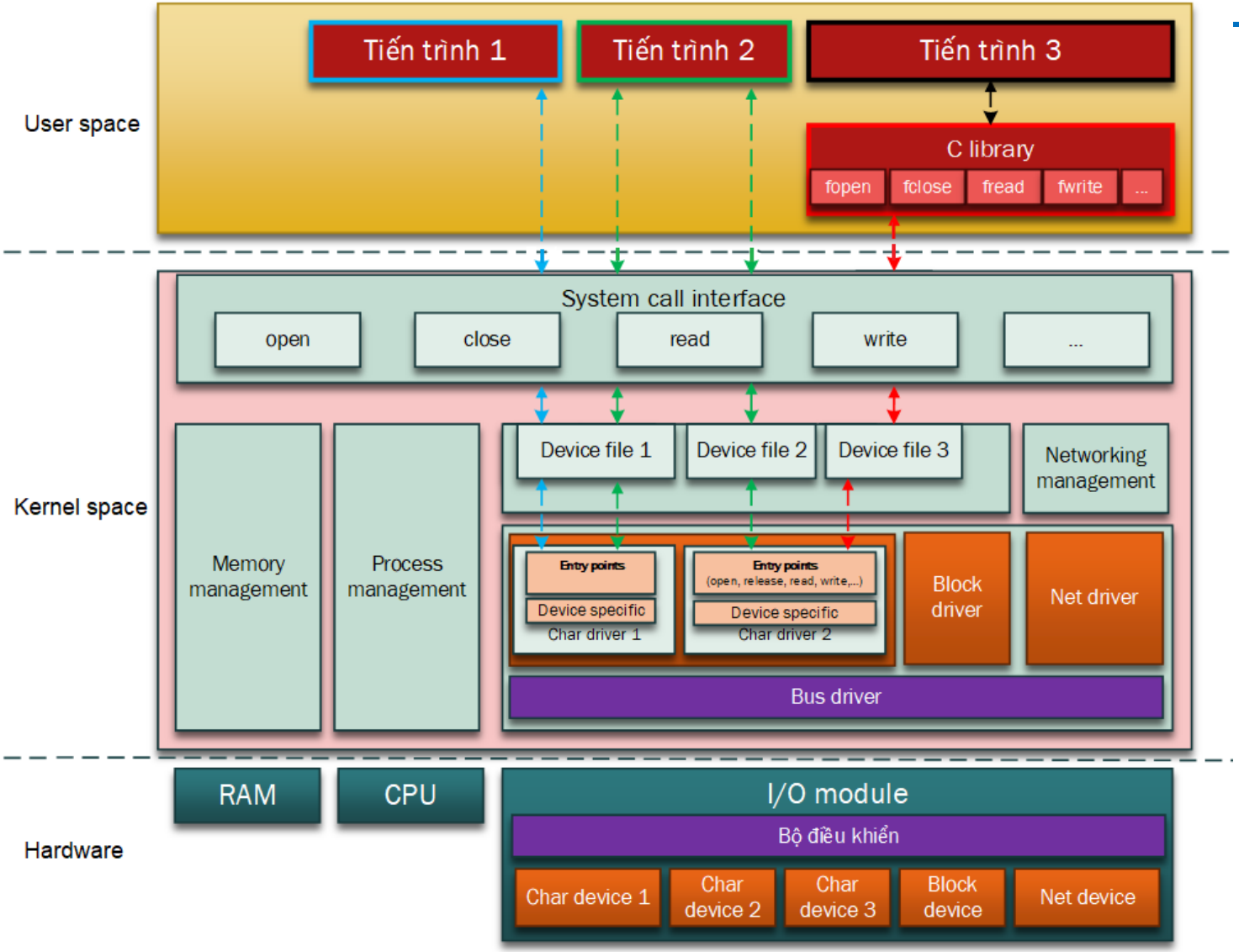
Nội dung

- Review bài trước: Quy trình thao tác với character driver
- Cấp phát tĩnh Device number
- Cấp phát động Device number
- Tạo file thiết bị
- Đăng ký các điểm vào (Entry point)
- Ví dụ minh họa

Quy trình thao tác Character driver

- Hoạt động:
  - 1. Application (ứng dụng) đọc/ghi file thiết bị
    - ✓ Tạo file thiết bị
    - ✓ Làm việc với file thiết bị thông qua system call: open, close, read, write, v.v.
  - 2. Character device file (CDF - 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
  - 3. 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
  - 4. Character device (Thiết bị)

Quy trình thao tác Character driver



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.

Cấp phát tĩnh Device number

- Biểu diễn device number
- Cấp phát tĩnh device number
- Hủy đăng ký device number với kernel
- Ví dụ minh họa

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

Hàm hoặc macro	Ý nghĩa
MAJOR(dev_t ***dev***)	Trả về major number từ device number
MINOR(dev_t ***dev***)	Trả về minor number từ device number
MKDEV(int ***major***, int ***minor***)	Gộp số major và minor để tạo thành device number
unsigned **imajor**(struct inode* ***inode***)	Trả về major number từ cấu trúc inode (cấu trúc inode dùng để mô tả device file)
unsigned **iminor**(struct inode* ***inode***)	Trả về minor number từ inode mô tả device file

Biểu diễn device number

- Xem device number
  - *Lệnh ls -l*

```
crw-rw-rw-    1 root     root       1,   3 Apr 11  2002 null
crw-------    1 root     root      10,   1 Apr 11  2002 psaux
crw-------    1 root     root       4,   1 Oct 28 03:04 tty1
crw-rw-rw-    1 root     tty        4,  64 Apr 11  2002 ttys0
crw-rw----    1 root     uucp       4,  65 Apr 11  2002 ttyS1
crw--w----    1 vcsa     tty        7,   1 Apr 11  2002 vcs1
crw--w----    1 vcsa     tty        7, 129 Apr 11  2002 vcsa1
crw-rw-rw-    1 root     root       1,   5 Apr 11  2002 zero
```

  - Trong: ***/proc/devices***

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
180 usb
```

```
Block devices:
  2 fd
  8 sd
 11 sr
 65 sd
 66 sd
```

Cấp phát tĩnh

- Chọn bất cứ số nào trong khoảng từ 0 -> 2^{12} - 1 làm major number trừ các số đã cấp phát
  - Cần xem device number (slide trước)
- Đăng ký
  - gọi hàm **register_chrdev_region**
  - *int register_chrdev_region (dev_t first, unsigned count, const char *name);*
- Hủy đăng ký
  - *void unregister_chrdev_region(dev_t first, unsigned int cnt)*

Cấp phát tĩnh

```
#include <linux/fs.h>
/*
 * Chức năng: đăng ký một dải gồm [cnt] device number bắt đầu từ [first] cho
 *            character device có tên là [name]
 * Tham số đầu vào:
 *    first [I]: device number đầu tiên muốn đăng ký <major, first_minor>.
 *               Ta có thể dùng macro MKDEV để ghép số major và minor mà
 *               ta muốn đăng ký rồi truyền kết quả vào cho hàm này.
 *    cnt   [I]: số lượng device number mà ta muốn đăng ký. Driver sẽ đăng
 *               ký với kernel các device number, từ <major, first_minor>
 *               cho đến <major, first_minor + cnt - 1>
 *    *name [I]: tên của character device. Tên này sẽ xuất hiện trong thư mục
 *               /proc/devices và khác với tên của device file trong thư mục /dev
 * Trả về:
 *    Nếu đăng ký thành công, hàm sẽ trả về 0.
 *    Nếu đăng ký thất bại, hàm sẽ trả về một giá trị < 0.
 */
int register_chrdev_region (dev_t first, unsigned count, const char *name);
```

Ví dụ minh họa

- Tạo project với 3 file: Makefile, Kbuild và *vchar_driver.c*
- Phân tích các thành phần trong vchar_driver.c:
  - Khai báo thư viện

```
#include <linux/module.h> /* thu vien nay dinh nghia cac macro nhu module_init va module_exit */
#include <linux/fs.h> /* thu vien nay dinh nghia cac ham cap phat/giai phong device number */
```

  - Kiểm tra trong */proc/devices* để lấy số chưa sử dụng làm Device number

Ví dụ minh họa

- Để lưu giá trị device number này, ta sẽ tạo ra một cấu trúc ***vchar_drv*** chứa trường ***dev_num***

```
struct _vchar_drv {
        dev_t dev_num;
} vchar_drv;
```

- Trong hàm **vchar_driver_init** của driver này, sẽ sử dụng macro **MKDEV** để khởi tạo giá trị cho trường ***dev_num*** của cấu trúc ***vchar_drv***
- Sau đó, gọi hàm **register_chrdev_region** để đăng ký device number với Linux kernel

Ví dụ minh họa

```
/* ham khoi tao driver */
static int __init vchar_driver_init(void)
{
+       int ret = 0;
+
        /* cap phat device number */
+       vchar_drv.dev_num = MKDEV(235,0);
+       ret = register_chrdev_region(vchar_drv.dev_num, 1, "vchar_device");
+       if (ret < 0) {
+               printk("failed to register device number statically\n");
+               goto failed_register_devnum;
+       }
        printk("Initialize vchar driver successfully\n");
        return 0;
+
+ failed_register_devnum:
+       return ret;
}
```

Ví dụ minh họa

- Trong hàm **vchar_driver_exit** của driver này, ta gọi hàm **unregister_chrdev_region** để giải phóng device number

```
    /* ham ket thuc driver */
@@ -65,6 +81,7 @@ static void __exit vchar_driver_exit(void)
            /* xoa bo device file */

            /* giai phong device number */
+           unregister_chrdev_region(vchar_drv.dev_num, 1);

            printk("Exit vchar driver\n");
    }
```

- Sử dụng các lệnh: make, insmod để dịch và load module
  - Xuất hiện một dòng chứa “235 vchar_device” trong */proc/devices*

Ví dụ minh họa

- Chú ý
  - Sau các thao tác này, không thấy có thêm device file trong thư mục */dev*
  - Vì mới chỉ xin Linux kernel cấp phát device number, chứ chưa tạo device file tương ứng với char driver này

Tổng kết về cấp phát tĩnh

- **3 bước thực hiện**
  - 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
- Sau khi không sử dụng thì hủy đăng ký
  - Gọi hàm **unregister_chrdev_region** để giải phóng device number
  - Nên đặt hàm **unregister_chrdev_region** bên trong hàm kết thúc của char driver

Cấp phát động Device number

- Hạn chế của cấp phát tĩnh
  - Cần phải kiểm tra device number trong /proc/devices
  - Khi chạy driver trên **máy tính khác** có thể lỗi do device number có thể đã được sử dụng => lỗi
- Khắc phục:
  - Sử dụng 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
  - Thường gọi hàm này trong hàm khởi tạo của char driver

Cấp phát động Device number

```
#include <linux/fs.h>
/*
 * Chức năng: yêu cầu kernel cấp phát một dải gồm [cnt] device number cho
 *            char device có tên là [name].
 * Tham số đầu vào:
 *    *dev        [O]: con trỏ này chứa giá trị trả về của hàm. Device number
 *                     đầu tiên của dải sẽ được trả về thông qua biến này.
 *     firstminor [I]: giá trị minor của số device number đầu tiên trong dải.
 *     cnt        [I]: là số lượng device number mà hàm này yêu cầu cấp phát.
 *     *name      [I]: tên của character device. Tên này sẽ xuất hiện trong
 *                     thư mục /proc/devices
 * Trả về:
 *     Nếu tìm được một device number, hàm này sẽ trả về 0. Device number đầu
 *     tiên trong dải sẽ được trả qua tham số *dev.
 *     Nếu không thể tìm được một device number nào, hàm sẽ trả về số nguyên âm.
 */
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int cnt, char
 *name)
```

Cấp phát động Device number

- Ví dụ minh họa: sửa đổi ví dụ phần cấp phát tĩnh
  - Thay cấp phát tĩnh thành động trong hàm khởi tạo
  - Các thành phần khác tương tự

```
/* ham khoi tao driver */
static int __init vchar_driver_init(void)
{
+       int ret = 0;
+
        /* cap phat device number */
+       vchar_drv.dev_num = 0;
+       ret = alloc_chrdev_region(&vchar_drv.dev_num, 0, 1, "vchar_device");
+       if (ret < 0) {
+               printk("failed to register device number dynamically\n");
+               goto failed_register_devnum;
+       }
+       printk("allocated device number (%d,%d)\n", MAJOR(vchar_drv.dev_num), MINOR(vchar_drv.dev_num));
```

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

Tạo file thiết bị cách thủ công

- Sử dụng công cụ ***mknod***

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

Trong đó,
  <quyền truy cập> dùng để thiết lập quyền đọc/ghi/thực thi cho file
  <tên device file> là tên sẽ xuất hiện trong thư mục /dev
  <kiểu device> thể hiện device file là character (c) hay block (b)
  <major> số major number
  <minor> số minor number
```

- Ví dụ

```
sudo mknod -m 666 /dev/vchar_dev c 246 0
```

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

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

Tạo file thiết bị tự động

- Để tạo một lớp các thiết bị, sử dụng hàm **class_create** của Linux kernel

```
/* Chức năng: Tạo ra một lớp các thiết bị có tên là [name] trong
 *            thư mục /sys/class. Lớp này chứa liên kết tới thông
 *            tin của các thiết bị cùng loại.
 * Tham số truyền vào:
 *   *owner [I]: con trỏ trỏ tới module sở hữu lớp thiết bị này
 *   *name  [I]: tên của lớp các thiết bị
 * trả về:
 *    Nếu thành công, thư mục có tên [name] được tạo ra trong
 *    /sys/class. Hàm trả về một con trỏ trỏ tới biến cấu trúc class.
 *    Nếu thất bại, trả về NULL
 */
struct class* class_create(struct module *owner, const char *name)

/* Hàm hủy tương ứng với class_create */
void class_destroy(struct class *)
```

Tạo file thiết bị tự động

- Để tạo thiết bị trong lớp trên, sử dụng hàm **device_create** của Linux kernel

```
/*
 * Chức năng: Tao ra các thông tin của một thiết bị cụ thể.
 *            Khi có thông tin này, udev sẽ tạo ra một device file
 *            tương ứng trong /dev
 * Tham số truyền vào:
 *    *cls    [I]: con trỏ trỏ tới lớp các thiết bị. Con trỏ này là kết
 *                 quả của việc gọi hàm class_create
 *    *parent [I]: con trỏ trỏ tới thiết bị cha của thiết bị này. Nếu
 *                 thiết bị không có cha, ta truyền vào là NULL
 *    devt    [I]: device number của thiết bị.
 *    *drvdata[I]: dữ liệu bổ sung. Nếu không có, ta truyền vào là NULL.
 *    *name   [I]: tên của thiết bị. udev sẽ tạo ra device file với tên
 *                 này trong thư mục /dev
 */
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)
```

Tạo file thiết bị tự động

- **Ví dụ minh họa**: sửa file *vchar_driver.c* trong phần trước
  - Khai báo thư viện

```
#include <linux/module.h> /* thu vien nay dinh nghia cac macro nhu module_init va module_exit */
#include <linux/fs.h> /* thu vien nay dinh nghia cac ham cap phat/giai phong device number */
#include <linux/device.h> /* thu vien nay chua cac ham phuc vu viec tao device file */
```

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

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

Tạo file thiết bị tự động

- Trong hàm **vchar_driver_init**, ta gọi hàm **class_create** để tạo ra một lớp thiết bị có tên là “class_vchar_dev”
  - Nếu không thực hiện thành công, cần gọi hàm **unregister_chrdev_region** để giải phóng device number đã được cấp phát trước đó
- Tiếp tục gọi hàm **device_create** để tạo ra một thiết bị có tên là “vchar_dev” trong thư mục */dev*
  - Nếu hàm **device_create** không thực hiện thành công, ta cần gọi hàm **destroy_class** để giải phóng lớp thiết bị đã được tạo ra trước đó

Tạo file thiết bị tự động

```
printk("allocated device number (%d,%d)\n", MAJOR(vchar_drv.dev_num), MINOR(vchar_drv.dev_num));
/* tao device file */
vchar_drv.dev_class = class_create(THIS_MODULE, "class_vchar_dev");
if(vchar_drv.dev_class == NULL) {
        printk("failed to create a device class\n");
        goto failed_create_class;
}
vchar_drv.dev = device_create(vchar_drv.dev_class, NULL, vchar_drv.dev_num, NULL, "vchar_dev");
if(IS_ERR(vchar_drv.dev)) {
        printk("failed to create a device\n");
        goto failed_create_device;
}
printk("Initialize vchar driver successfully\n");
return 0;

failed_create_device:
        class_destroy(vchar_drv.dev_class);
failed_create_class:
        unregister_chrdev_region(vchar_drv.dev_num, 1);
failed_register_devnum:
        return ret;
}
```

Tạo file thiết bị tự động

- Cuối cùng, trong hàm **vchar_driver_exit** của driver này, gọi hàm **device_destroy** và **class_destroy** để giải phóng những gì đã tạo ra
- Chú ý: thứ tự gọi **device_destroy** trước, gọi **class_destroy** sau

```
/* giai phong bo nho da cap phat cau truc du lieu cua driver */


/* xoa bo device file */
device_destroy(vchar_drv.dev_class, vchar_drv.dev_num);
class_destroy(vchar_drv.dev_class);


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

Tạo file thiết bị tự động

- Sử dụng lệnh ***make*** để biên dịch char driver
- sử dụng lệnh ***insmod*** để load driver vào trong kernel
- Trong quá trình khởi tạo char driver
  - Hàm **class_create** sẽ tạo thư mục *class_vchar_dev* trong thư mục */sys/class*, và hàm **device_create** tạo thư mục *vchar_dev* bên trong thư mục */sys/class/class_vchar_dev*
  - Trong thư mục *vchar_dev*, có một file tên là *uevent* chứa thông tin mà char driver gửi cho tiến trình **udevd**
    - ✓ Dựa vào các thông tin này, **udevd** sẽ tạo ra device file *vchar_dev* trong thư mục */dev*

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

Đăng ký entry point open và release

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

```
#include <linux/cdev.h>.
struct cdev {
      ...
      struct file_operations *ops; /* các entry points */
      dev_t dev; /* device number của character device */
      unsigned int count; /* số lượng các thiết bị có cùng major number */
      ...
}
```

- 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

Hàm thao tác cdev phổ biến

```
/*
 * hàm: cdev_alloc
 * chức năng: cấp phát bộ nhớ cho cấu trúc cdev.
 * tham số đầu vào: không có.
 * giá trị trả về:
 *    Nếu thành công, hàm này trả về địa chỉ của vùng nhớ chứa cấu trúc cdev
 *    Nếu thất bại, hàm này trả về NULL.
 */
struct cdev *cdev_alloc(void)

/*
 * hàm: cdev_init
 * chức năng: khởi tạo các trường của cấu trúc cdev.
 * tham số đầu vào:
 *    *p    [I]: là địa chỉ của vùng nhớ chứa cấu trúc cdev.
 *    *fops [I]: là địa chỉ của vùng nhớ chứa cấu trúc file_operations.
 *               Cấu trúc file_operations mô tả hàm nào của char driver
 *               sẽ trở thành entry point. Tham số này sẽ được gán cho
 *               trường *ops của cấu trúc cdev.
 * giá trị trả về: không có.
 * chú ý: hàm này được gọi sau hàm cdev_alloc.
 */
void cdev_init(struct cdev *p, struct file_operations *fops)
```

Hàm thao tác cdev phổ biến

```
/*
 * hàm cdev_add
 * chức năng: đăng ký cấu trúc cdev với Linux kernel
 *            và liên kết cấu trúc này với device file.
 * tham số đầu vào:
 *    *p    [I]: là địa chỉ của vùng nhớ chứa cấu trúc cdev.
 *    first [I]: là device number của device file đầu tiên trong
 *               chuỗi các device file liên kết với cấu trúc cdev.
 *               Các device number này có cùng major number.
 *    count [I]: là số lượng các device file sẽ liên kết với cấu trúc cdev
 * giá trị trả về:
 *    Nếu thành công, hàm này trả về 0.
 *    Nếu thất bại, hàm này trả về một số âm.
 * chú ý: hàm này được gọi sau hàm cdev_alloc và cdev_init.
 */
int cdev_add(struct cdev *p, dev_t first, unsigned count)
/*
 * hàm cdev_del
 * chức năng: hủy bỏ vùng nhớ của cấu trúc cdev.
 * tham số đầu vào:
 *     *p [I]: địa chỉ của vùng nhớ chứa cấu trúc cdev.
 * giá trị trả về: không có.
 * chú ý: hàm này thường được gọi trong hàm kết thúc của
 *        của char driver.
 */
void cdev_del(struct cdev *p)
```

Cấu trúc file_operations

- Cấu trúc **file_operations** gồm các con trỏ hàm
  - Nhiệm vụ của char driver là gán các hàm của char driver cho các con trỏ ấy
  - Sau khi gán xong, các hàm này trở thành entry point của char driver
- Khi char driver gọi hàm **cdev_add**, Linux kernel sẽ thiết lập mối liên hệ 1-1 giữa system call và entry point
  - 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

Cấu trúc file_operations

```
/*
 * con trỏ hàm open
 * chức năng: khi một tiến trình trên user space gọi system call open
 *            để mở device file tương ứng với char driver này, thì hàm
 *            này sẽ được gọi để thực hiện một số việc như kiểm tra
 *            thiết bị đã sẵn sàng chưa, khởi tạo thiết bị nếu cần,
 *            lưu lại minor number…
 * tham số đầu vào:
 *     *inode [I]: địa chỉ của cấu trúc inode. Cấu trúc này dùng để mô tả
 *                 các file trong hệ thống.
 *     *filp  [I]: địa chỉ của cấu trúc file. Cấu trúc file dùng để mô tả
 *                 một file đang mở.
 * giá trị trả về:
 *      Hàm này trả về 0 để thông báo mở device file thành công.
 *      Ngược lại, trả về một số khác 0 để thông báo mở device file
 *      thất bại.
 */
int (*open)(struct inode *inode, struct file *filp)
```

Cấu trúc file_operations

```
/*
 * con trỏ hàm release
 * chức năng: khi một tiến trình trên user space gọi system call close
 *            để đóng device file tương ứng với char driver này, thì
 *            hàm này sẽ được gọi để thực hiện một số việc như tắt
 *            thiết bị, hoặc làm cho thiết bị ngừng hoạt động…
 * tham số đầu vào: giống như hàm open
 * giá trị trả về:
 *      Hàm này trả về 0 để thông báo đóng device file thành công.
 *      Ngược lại, trả về một số khác 0 để thông báo đóng device file
 *      thất bại.
 */
int (*release)(struct inode *inode, struct file *filp)
```

Ví dụ minh họa

- Yêu cầu
  - Đăng ký và triển khai các entry point **open** và entry point **release**
  - Xác định số lần mở device file kể từ lúc char driver được lắp vào trong Linux kernel

Ví dụ minh họa

- Tham chiếu tới thư viện <*linux/cdev.h*> để có thể sử dụng được cấu trúc **cdev** cùng các hàm liên quan

```
#include <linux/fs.h> /* thu vien nay dinh nghia cac ham cap phat/giai phong device number */
#include <linux/device.h> /* thu vien nay chua cac ham phuc vu viec tao device file */
#include <linux/slab.h> /* thu vien nay chua cac ham kmalloc va kfree */
#include <linux/cdev.h> /* thu vien nay chua cac ham lam viec voi cdev */
```

- Để lưu địa chỉ của cấu trúc **cdev**, thêm trường ***vcdev** vào cấu trúc ***vchar_drv***
- Để lưu được số lần đã mở device file, thêm trường ***open_cnt*** vào trong cấu trúc ***vchar_drv***

```
typedef struct vchar_dev {
        unsigned char * control_regs;
@@ -28,6 +29,8 @@ struct _vchar_drv {
        struct class *dev_class;
        struct device *dev;
        vchar_dev_t * vchar_hw;
+       struct cdev *vcdev;
+       unsigned int open_cnt;
} vchar_drv;
```

Ví dụ minh họa

- Triển khai các hàm **vchar_driver_open** và **vchar_driver_release**
- Gán các hàm này cho các con trỏ ***open*** và ***release*** của cấu trúc **file_operation**

Ví dụ minh họa

```
/******************************** OS specific - START ********************************/
/* cac ham entry points */
+ static int vchar_driver_open(struct inode *inode, struct file *filp)
+ {
+         vchar_drv.open_cnt++;
+         printk("Handle opened event (%d)\n", vchar_drv.open_cnt);
+         return 0;
+ }
+
+ static int vchar_driver_release(struct inode *inode, struct file *filp)
+ {
+         printk("Handle closed event\n");
+         return 0;
+ }
+
+ static struct file_operations fops =
+ {
+         .owner    = THIS_MODULE,
+         .open     = vchar_driver_open,
+         .release = vchar_driver_release,
+ };
```

Ví dụ minh họa

- Trong hàm **vchar_driver_init**
  - Đăng ký cấu trúc **cdev** với Linux kernel
    - ✓Gọi hàm **cdev_alloc** để yêu cầu kernel "cấp cho một bộ hồ sơ" cdev mới
    - ✓Nếu quá trình cấp phát thất bại, 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, tiếp tục gọi hàm **cdev_init** để "điền các thông tin vào bộ hồ sơ" cdev
    - ✓Sau đó, gọi hàm **cdev_add** để gửi “bộ hồ sơ” này tới kernel

Ví dụ minh họa

```
        /* dang ky cac entry point voi kernel */
+       vchar_drv.vcdev = cdev_alloc();
+       if(vchar_drv.vcdev == NULL) {
+               printk("failed to allocate cdev structure\n");
+               goto failed_allocate_cdev;
+       }
+       cdev_init(vchar_drv.vcdev, &fops);
+       ret = cdev_add(vchar_drv.vcdev, vchar_drv.dev_num, 1);
+       if(ret < 0) {
+               printk("failed to add a char device to the system\n");
+               goto failed_allocate_cdev;
+       }
        printk("Initialize vchar driver successfully\n");
        return 0;

+ failed_allocate_cdev:
+       vchar_hw_exit(vchar_drv.vchar_hw);
  failed_init_hw:
        kfree(vchar_drv.vchar_hw);
```

Ví dụ minh họa

- Trong hàm **vchar_driver_exit**, gọi hàm **cdev_del** để hủy đăng ký entry point với kernel

```
@@ -139,6 +174,7 @@ static void __exit vchar_driver_exit(void)
        /* huy dang ky xu ly ngat */

        /* huy dang ky entry point voi kernel */
+       cdev_del(vchar_drv.vcdev);

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

Đăng ký entry point ioctl

- Mục đích
  - Điều chỉnh cấu hình, hoặc lấy thông tin cấu hình của char device
- Khi tiến trình cần điều chỉnh cấu hình, hoặc lấy thông tin cấu hình của char device, các bước diễn ra như sau:
  - Đầu tiên, tiến trình gọi system call open để mở device file tương ứng của char device
  - Tiếp theo, tiến trình gọi system call ioctl để đọc/ghi thông tin vào device file. Khi đó, Linux kernel sẽ kích hoạt entry point ioctl của char driver tương ứng.
  - Sau đó, tùy vào yêu cầu, entry point ioctl sẽ điều chỉnh cấu hình hoặc lấy thông tin cấu hình của char device.
  - Cuối cùng, entry point ioctl trả kết quả về cho tiến trình trên user space

System call ioctl

```c
#include <sys/ioctl.h>
/*
 * system call ioctl
 * chức năng: truyền đạt mong muốn của tiến trình cho kernel về việc
 *            điều chỉnh cấu hình hoặc lấy thông tin cấu hình của char device
 * tham số đầu vào:
 *        fd      [I]: là định danh mô tả device file của char device đang mở
 *        request [I]: là mã lệnh được gửi tới char driver.
 *        ...   [I/O]: tham số này có thể có hoặc không. Nếu có, thì đây
 *                     là địa chỉ của vùng nhớ chứa giá trị điều chỉnh hoặc
 *                     chứa thông tin trả về từ char driver.
 * giá trị trả về:
 *        Trả về 0 nếu yêu cầu điều chỉnh hoặc lấy thông tin thành công.
 *        Trả về -1 nếu xảy ra lỗi.
 */
int ioctl(int fd, unsigned long request, ...)
```

System call ioctl

- Để tạo ra được mã lệnh, ta sử dụng các macro **__IO**, **__IOW**, **__IOR**, **__IOWR**

```
/*
 * IOX có thể là:
 *     IO  : thể hiện đây là một ioctl không tham số.
 *     IOW : thể hiện đây là một ioctl với các tham số ghi.
 *     IOR : thể hiện đây là một ioctl với các tham số đọc.
 *     IOWR: thể hiện đây là một ioctl với các tham số đọc và ghi.
 * magic number là một số định danh để xác định yêu cầu sẽ được
 *              gửi tới char driver mong muốn nào đó.
 * command number là một số định danh để phân biệt các lệnh với nhau.
 * argument type là kiểu dữ liệu muốn trao đổi giữa tiến trình và char driver
 */
#define “ioctl name” __IOX(“magic number”, “command number”, “argument type”)
```

- **Ví dụ:** muốn điều chỉnh độ to của loa là 69, các việc cần làm:

```
int vol = 69;
int fd = open("/dev/speaker", O_RDWR);
#define SET_VOLUMN_SPEAKER _IOW(96, 1, int *)
ioctl(fd, SET_VOLUMN_SPEAKER, &vol);
```

HỎI - ĐÁP