BÀI THỰC HÀNH 3: TRÌNH ĐIỀU KHIỂN KÝ TỰ

I. MỤC TIÊU

Sau khi hoàn thành buổi thực hành này, sinh viên có thể:

- Hiểu khái niệm *Character Device Driver (char driver)* và vai trò của nó trong Linux.
- Giải thích được *major number, minor number* và cách kernel sử dụng chúng để ánh xạ thiết bị.
- Sử dụng *struct file_operations* để định nghĩa các hàm *open, release, read, write* cho driver.
- Biên dịch và nạp một char driver đơn giản (scull), tạo device file trong /dev.
- Kiểm tra hoạt động của driver bằng các lệnh *echo, cat, cp, dd…*
- Quản lý bộ nhớ trong driver bằng *kmalloc/kfree, copy_to_user, copy_from_user*.

II. LÝ THUYẾT CƠ BẢN

1. GIỚI THIỆU VỀ THIẾT BỊ KÝ TỰ VÀ TRÌNH ĐIỀU KHIỂN THIẾT BỊ KÝ TỰ

Trong hệ điều hành Linux, phần cứng được trừu tượng hóa dưới dạng thiết bị. Người dùng và chương trình có thể truy cập thiết bị thông qua các tệp đặc biệt (device file), thường được đặt trong thư mục */dev*. Khi liệt kê bằng lệnh *ls -l /dev*, ta có thể phân biệt các loại thiết bị nhờ ký tự đầu tiên trong cột quyền truy cập:

- **Thiết bị ký tự (character device)**: ký hiệu c. Đây là những thiết bị trao đổi dữ liệu theo dòng byte (byte stream). Mỗi lần đọc/ghi có thể chỉ vài byte, tương tự như thao tác trên một tệp văn bản. Ví dụ: cổng nối tiếp (/dev/ttyS0), bàn phím, chuột, hay các thiết bị đặc biệt như /dev/null và /dev/zero.
- **Thiết bị khối (block device)**: ký hiệu b. Dữ liệu được trao đổi theo khối có kích thước cố định (block), thích hợp cho việc lưu trữ và quản lý hệ thống tệp. Ví dụ: ổ cứng HDD/SSD (/dev/sda), ổ USB, hay thẻ nhớ SD (/dev/mmcblk0).
- **Thiết bị mạng (network device)**: không xuất hiện dưới dạng tệp trong /dev, mà được quản lý như giao diện mạng (network interface). Ví dụ: card Ethernet (eth0), card Wi-Fi (wlan0).

Trong ba loại trên, thiết bị ký tự thường đơn giản hơn cả về cơ chế hoạt động và lập trình điều khiển và thường chọn char driver làm nội dung đầu tiên để làm quen với lập trình driver.

Trình điều khiển thiết bị ký tự (character driver)

Char driver là phần mềm trong kernel quản lý thiết bị ký tự (char device), chịu trách nhiệm kết nối lời gọi hệ thống (system call) từ user space (như open, read, write, llseek) tới thiết bị ký tự thực tế. Nhờ có char driver, ứng dụng người dùng có thể thao tác với thiết bị thông qua các lời gọi quen thuộc như với tệp tin bình thường.

Đặc điểm nổi bật của char driver:

- Ngữ nghĩa theo luồng byte: dữ liệu được truyền tải liên tục từng byte, có thể đọc/ghi với độ dài tùy ý.
- Hỗ trợ truy cập tuần tự hoặc ngẫu nhiên thông qua cơ chế con trỏ vị trí (file offset).
- Đơn giản, trực quan: dễ hình dung và dễ kiểm thử bằng các lệnh cơ bản trong shell (cat, echo, dd).
- Cấu trúc chuẩn hóa: mọi char driver đều phải cung cấp một tập hợp hàm cơ bản (open, release, read, write, llseek) và đăng ký chúng với kernel thông qua cấu trúc file_operations.

scull driver – mô hình học tập

Trong khóa học này, chúng ta không làm việc trực tiếp với phần cứng phức tạp mà sử dụng một driver mô phỏng tên là scull (*Simple Character Utility for Loading Localities*). Scull là một char driver giả lập: thay vì điều khiển thiết bị phần cứng, nó thao tác với vùng nhớ động trong RAM và coi đó là thiết bị. Nhờ vậy:

- Sinh viên có thể thực hành viết driver mà không cần phần cứng thật.
- Môi trường an toàn để thử nghiệm, không lo gây lỗi cho các thiết bị quan trọng của hệ thống.
- Minh họa rõ ràng cơ chế giao tiếp giữa kernel, driver và user space, vốn là trọng tâm cần nắm vững trước khi làm việc với thiết bị thực.
- Có thể dễ dàng kiểm tra bằng những lệnh thông thường như echo "abc" > /dev/scull0 và cat /dev/scull0.

2. MAJOR VÀ MINOR NUMBER

Trong hệ điều hành Linux, mỗi thiết bị (device) được kernel định danh bằng một cặp số hiệu gồm major number và minor number. Hai số này giúp kernel xác định driver nào chịu

trách nhiệm xử lý các thao tác I/O, và thiết bị con nào (device instance) trong số các thiết bị mà driver đó quản lý đang được truy cập.

Nhờ cơ chế này, Linux có thể quản lý hàng nghìn thiết bị khác nhau thông qua cùng một giao diện file thống nhất trong thư mục /dev. Khi hiểu rõ về major/minor, ta sẽ nắm được cách kernel "liên kết" một tệp thiết bị trong /dev với driver trong kernel.

2.1. Device file trong /dev và hiển thị major/minor

Mỗi thiết bị trong Linux đều có một tệp thiết bị (device file) trong thư mục **/dev**. Khi liệt kê bằng lệnh **ls -l**, ta sẽ thấy loại thiết bị (c cho character, b cho block) và hai con số thể hiện major và minor number:

```
$ ls -l /dev/null
crw-rw-rw- 1 root root 1, 3 Oct  5 11:45 /dev/null
```

Ở đây:

- Chữ cái đầu c cho biết đây là character device.
- Số 1 là major number – đại diện cho driver chịu trách nhiệm (driver quản lý nhóm thiết bị có major này).
- Số 3 là minor number – xác định thiết bị con cụ thể trong nhóm driver đó.

Ví dụ, tất cả thiết bị có major = 1 (như */dev/null, /dev/zero, /dev/full*) đều do cùng driver quản lý, chỉ khác nhau ở minor để phân biệt chức năng. Như vậy, kernel không nhìn */dev/null* như một "tên file", mà như một thiết bị số (1,3) — chính hai con số này cho phép kernel tra ra driver tương ứng trong bảng thiết bị hệ thống.

2.2. Device number (dev_t) và các macro hỗ trợ

Trong kernel, cặp (major, minor) được gói gọn trong một kiểu dữ liệu duy nhất là **dev_t**. Thay vì làm việc trực tiếp với số nguyên, lập trình viên dùng các macro tiện ích để tách hoặc ghép hai giá trị này.

Cấu trúc khái niệm:

```
typedef unsigned long dev_t;
```

dev_t là một số nguyên không dấu, trong đó kernel lưu trữ cả major và minor bằng cách chia nhỏ các bit bên trong. Ta không nên truy cập bit trực tiếp, mà luôn dùng macro chính thức sau:

1. MAJOR(dev_t dev)

→ Trích major number từ một giá trị dev_t.

Cú pháp:

```
unsigned int MAJOR(dev_t dev);
```

Ví dụ:

```
dev_t devno = MKDEV(250, 0);  // tạo dev_t
unsigned int major = MAJOR(devno);  // major = 250
```

2. MINOR(dev_t dev)

→ Trích minor number từ một dev_t.

Cú pháp:

```
unsigned int MINOR(dev_t dev);
```

Ví dụ:

```
dev_t devno = MKDEV(250, 5);
unsigned int minor = MINOR(devno);  // minor = 5
```

3. MKDEV(int major, int minor)

→ Tạo một giá trị dev_t từ cặp số hiệu.

Cú pháp:

```
dev_t MKDEV(unsigned int major, unsigned int minor);
```

Ví dụ:

```
dev_t devno = MKDEV(250, 3);
printk(KERN_INFO "Device number: major=%d minor=%d\n",
       MAJOR(devno), MINOR(devno));
```

Kết quả log:

```
Device number: major=250 minor=3
```

Các macro này giúp mã nguồn của driver trở nên rõ ràng, dễ đọc và tương thích với mọi phiên bản kernel (do layout bit của dev_t có thể thay đổi giữa các phiên bản).

2.3. Quản lý số hiệu thiết bị – Cấp phát và giải phóng

Trong Linux, tất cả các thiết bị (kể cả thiết bị ảo) đều cần được đăng ký số hiệu thiết bị với kernel.

Có hai cách: **đăng ký tĩnh (static)** và **đăng ký động (dynamic)**.

a) Đăng ký tĩnh với register_chrdev_region()

Hàm này dùng khi ta đã biết trước major number và muốn kernel dành riêng dải số hiệu này cho driver của ta.

Cú pháp:

```
int register_chrdev_region(dev_t first, unsigned count,
                           const char *name);
```

- first: giá trị dev_t đầu tiên trong dải (tạo bằng MKDEV(major, minor_start)).
- count: số lượng thiết bị con ta muốn đăng ký.
- name: tên nhận diện driver (hiển thị trong /proc/devices).

Trả về: 0 nếu thành công, hoặc giá trị âm nếu lỗi (ví dụ major đã bị chiếm dụng).

Ví dụ:

```
dev_t devno = MKDEV(250, 0);
register_chrdev_region(devno, 2, "mydriver");
```

→ Đăng ký major 250, minor 0–1 cho hai thiết bị /dev/mydev0, /dev/mydev1.

Cách này phù hợp khi ta cố định môi trường, chẳng hạn hệ thống nhúng. Tuy nhiên, trong môi trường chung, major 250 có thể đã bị driver khác chiếm dụng, khiến hàm trả lỗi. Vì vậy, Linux khuyến khích dùng cách cấp phát động.

b) Cấp phát động với alloc_chrdev_region()

Đây là phương thức được dùng phổ biến hiện nay (và là cách mà scull áp dụng). Kernel sẽ tự chọn major còn trống, giúp tránh xung đột với các driver khác.

Cú pháp:

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor,
                        unsigned count, const char *name);
```

- dev: con trỏ đến biến dev_t – nơi kernel sẽ ghi giá trị số hiệu được cấp.
- baseminor: minor đầu tiên (thường là 0).

- count: số lượng thiết bị con cần đăng ký.
- name: tên hiển thị trong /proc/devices.

Trả về: 0 nếu thành công, hoặc giá trị âm nếu lỗi.

Ví dụ:

```
dev_t devno;
int ret;

ret = alloc_chrdev_region(&devno, 0, 4, "mydriver");
if (ret < 0) {
    printk(KERN_ERR "Cannot allocate device number\n");
    return ret;
}
printk(KERN_INFO "Registered device: major=%d minor=%d\n",
       MAJOR(devno), MINOR(devno));
```

Ý nghĩa: kernel sẽ chọn một major chưa dùng (ví dụ 240) và dành 4 minor (0–3). Khi kiểm tra bằng cat /proc/devices, ta sẽ thấy:

```
240 mydriver
```

c) Giải phóng số hiệu thiết bị với unregister_chrdev_region()

Sau khi driver được gỡ bỏ, ta phải trả lại số hiệu thiết bị đã đăng ký để kernel có thể tái sử dụng.

Cú pháp:

```
void unregister_chrdev_region(dev_t first, unsigned count);
```

- first: giá trị dev_t đầu tiên trong dải.
- count: số lượng thiết bị con.

Ví dụ:

```
unregister_chrdev_region(devno, 4);
```

→ Giải phóng 4 thiết bị có major và minor tương ứng đã đăng ký.

Cơ chế cấp phát động major number phản ánh triết lý thiết kế mở và an toàn của Linux:

- Tránh xung đột: mỗi driver chỉ được cấp major còn trống; kernel đảm bảo duy nhất.

- Linh hoạt: cùng một driver có thể được nạp ở nhiều máy khác nhau mà không cần chỉnh sửa mã nguồn.
- Thuận tiện kiểm tra: sau khi nạp module, có thể xem major number được cấp bằng lệnh: `dmesg | tail`

(Driver thường in log hiển thị major bằng printk() để người dùng tiện theo dõi.)

2.4. Ví dụ trong scull.c

(a) Xin số hiệu thiết bị động cho N thiết bị con

Trong hàm khởi tạo ***scull_init()***, driver scull xin dải số hiệu thiết bị động cho *scull_nr_devs* thiết bị (thường mặc định là 4).

```
/* Cấp phát số hiệu thiết bị động cho scull_nr_devs thiết bị */
err = alloc_chrdev_region(&scull_devno, 0, scull_nr_devs, "scull");
if (err) return err;

pr_info("scull: loaded (major=%d) devices=%d\n",
        MAJOR(scull_devno), scull_nr_devs);
```

Giải thích:

- scull_devno: biến kiểu dev_t để kernel ghi vào số hiệu được cấp.
- 0: minor bắt đầu là 0.
- scull_nr_devs: số lượng thiết bị con cần quản lý (4).
- "scull": tên hiển thị trong /proc/devices.

Khi lệnh *alloc_chrdev_region()* thành công, kernel sẽ tự chọn một major chưa sử dụng, ví dụ 240, và cấp 4 minor liên tiếp (0–3). Driver sau đó in ra log, để người dùng biết major number đã được cấp.

(b) Gỡ bỏ – Giải phóng dải số hiệu thiết bị

Khi module được gỡ khỏi kernel, scull_exit() phải giải phóng toàn bộ số hiệu đã đăng ký để tránh rò rỉ tài nguyên.

```
/* Giải phóng số hiệu thiết bị */
unregister_chrdev_region(scull_devno, scull_nr_devs);
```

```
pr_info("scull: unloaded\n");
```

Giải thích:

- scull_devno: giá trị dev_t gốc đã được cấp phát.
- scull_nr_devs: số thiết bị con tương ứng cần giải phóng.

Lời gọi này trả lại dải số hiệu cho kernel, đảm bảo rằng driver khác có thể sử dụng lại nếu cần.Việc giải phóng đúng cách giúp hệ thống duy trì tính ổn định và tránh lỗi khi nạp/gỡ module nhiều lần.

3. Một số cấu trúc dữ liệu quan trọng

Khi một chương trình người dùng (user-space) truy cập vào thiết bị ký tự bằng các lời gọi hệ thống như *open(), read(), write()* hay *llseek()*, kernel sẽ cần những cấu trúc dữ liệu đặc biệt để quản lý thông tin về tệp thiết bị và định tuyến lời gọi đến đúng hàm trong trình điều khiển (driver). Ba cấu trúc quan trọng nhất trong cơ chế này là:

- **struct file_operations** – Bảng ánh xạ giữa các lời gọi hệ thống và các hàm xử lý trong driver.
- **struct file** – Đại diện cho *một phiên làm việc* (một lần mở tệp hoặc thiết bị).
- **struct inode** – Đại diện cho *bản thân tệp thiết bị* trong hệ thống tệp ảo của Linux (Virtual File System – VFS).

Ba cấu trúc này tạo thành chuỗi liên kết **"file → inode → cdev → fops"**, là xương sống của mọi driver ký tự trong Linux. Hiểu rõ vai trò và mối quan hệ của chúng là nền tảng để triển khai được các hàm *open(), read(), write()* trong các phần sau.

3.1. struct file_operations – Bảng phương thức thao tác (Operations Table)

Cấu trúc *file_operations* định nghĩa những hành động mà driver hỗ trợ khi người dùng thao tác với thiết bị.

Mỗi phần tử của cấu trúc này là một con trỏ hàm, trỏ đến hàm xử lý tương ứng trong driver. Khi ứng dụng gọi read() hoặc write(), kernel sẽ tìm đến hàm tương ứng trong bảng file_operations và gọi hàm đó trong ngữ cảnh kernel space.

Cấu trúc tổng quát:

```
struct file_operations {
    struct module *owner;         /* con trỏ đến module chủ sở hữu */
```

```
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read)  (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t,
loff_t *);
    int (*open)      (struct inode *, struct file *);
    int (*release)   (struct inode *, struct file *);
    /* ... có thể có thêm poll, ioctl, mmap, v.v... */
};
```

Ý nghĩa các trường quan trọng:

- owner: Trỏ tới module sở hữu driver. Thường gán bằng THIS_MODULE để kernel biết rằng module này đang được sử dụng và ngăn việc gỡ bỏ khi thiết bị còn đang mở.
- llseek(): Di chuyển vị trí đọc/ghi trong thiết bị (giống như con trỏ file trong chương trình C).
- read() / write(): Trao đổi dữ liệu giữa user space và kernel space thông qua các hàm copy_to_user() và copy_from_user().
- open() / release(): Được gọi khi thiết bị được mở hoặc đóng, dùng để cấp phát hoặc giải phóng tài nguyên.

Nếu một phương thức không được hỗ trợ, ta có thể gán con trỏ hàm đó bằng NULL — kernel sẽ có cơ chế xử lý mặc định (hoặc báo lỗi "operation not supported").

Ví dụ trong mã nguồn scull.c:

```
static const struct file_operations scull_fops = {
    .owner   = THIS_MODULE,
    .open    = scull_open,
    .release = scull_release,
    .llseek  = scull_llseek,
    .read    = scull_read,
    .write   = scull_write,
};
```

Bảng này khai báo rằng: khi người dùng gọi *open(), read(), write()...* trên /dev/scull0, kernel sẽ gọi các hàm scull_open, scull_read, scull_write... tương ứng trong driver.

3.2. struct file – Đại diện cho một phiên mở thiết bị

Khi người dùng gọi *`open("/dev/scull0", O_RDWR)`* trong chương trình, kernel sẽ tạo ra một cấu trúc struct file để đại diện cho phiên làm việc này. Mỗi lần mở tệp sẽ có một bản sao struct file riêng biệt, ngay cả khi nhiều tiến trình cùng mở chung một tệp thiết bị.

Cấu trúc rút gọn:

```
struct file {
    loff_t f_pos;    /* vị trí đọc/ghi hiện tại */
    struct file_operations *f_op;   /* bảng phương thức của driver */
    void  *private_data;  /* dữ liệu riêng cho mỗi phiên mở */
    unsigned int f_flags;  /* cờ điều khiển (O_NONBLOCK, O_RDWR...) */
   /* ... các trường khác không thường dùng trong driver char cơ bản
... */
};
```

Các trường thường được sử dụng trong driver:

- f_pos: Vị trí đọc/ghi hiện tại trong thiết bị (offset). Khi đọc hoặc ghi, kernel sẽ tự động cập nhật f_pos, và ta có thể thay đổi nó trong hàm llseek().
- f_op: Con trỏ trỏ đến bảng file_operations mà driver đã đăng ký. Từ đây, kernel biết được cần gọi hàm nào khi thực hiện các thao tác như read, write.
- private_data: Là con trỏ mà driver có thể sử dụng để lưu dữ liệu riêng cho từng phiên mở — thường là con trỏ trỏ tới cấu trúc thiết bị (scull_dev). Điều này giúp các hàm read() và write() truy cập nhanh đến vùng nhớ thiết bị mà không cần tra cứu lại thông tin từ inode.
- f_flags: Cờ mở file. Dùng để kiểm tra chế độ mở (O_RDONLY, O_WRONLY, O_RDWR) hoặc các cờ như O_NONBLOCK.

Ví dụ trong hàm scull_open() của driver scull:

```
static int scull_open(struct inode *inode, struct file *filp)
```

```
{
    struct scull_dev *dev;

    /* Lấy con trỏ struct thiết bị từ inode->i_cdev */
    dev = container_of(inode->i_cdev, struct scull_dev, cdev);

    /* Gán vào private_data để các hàm read/write có thể truy cập */
    filp->private_data = dev;

    /* Nếu mở ở chế độ ghi (O_WRONLY) thì xóa dữ liệu cũ trong thiết
bị */
    if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
        scull_trim(dev);

    return 0;
}
```

Ở đây, *filp->private_data* giữ địa chỉ của thiết bị scull_dev tương ứng với minor device đang mở. Nhờ đó, các hàm read() và write() sau này có thể truy cập vào đúng vùng nhớ thiết bị.

3.3. struct inode – Đại diện cho tệp thiết bị trong hệ thống tệp ảo (VFS)

Trong hệ thống tệp của Linux, mỗi tệp (dù là tệp thường hay tệp thiết bị) đều được đại diện bởi một inode – viết tắt của *index node*. Cấu trúc struct inode chứa thông tin mô tả *đối tượng tệp* trong hệ thống, như loại tệp, quyền truy cập, và đặc biệt đối với thiết bị, là số hiệu thiết bị và con trỏ tới struct cdev mà driver đã đăng ký.

Cấu trúc rút gọn:

```
struct inode {
    dev_t         i_rdev;   /* số hiệu thiết bị (major, minor) */
    struct cdev  *i_cdev;   /* con trỏ đến cấu trúc cdev của driver */
    umode_t       i_mode;   /* loại tệp và quyền truy cập */
    /* ... các trường khác ... */
};
```

Một số trường quan trọng:

- i_rdev: Chứa cặp số (major, minor) của thiết bị mà inode đại diện.
- i_cdev: Con trỏ tới cấu trúc cdev mà driver đã đăng ký bằng cdev_add(). Thông qua inode->i_cdev, driver có thể truy cập ngược trở lại tới đối tượng thiết bị (scull_dev) của mình.
- i_mode: Xác định loại tệp (file thường, thư mục, thiết bị, v.v.) và quyền truy cập.

Trong các hàm của driver, struct inode thường chỉ được sử dụng trong hàm *open()*, nhằm xác định thiết bị cụ thể đang được mở (thông qua *inode->i_cdev* hoặc *imajor(inode)* và *iminor(inode)*).

3.4. Mối quan hệ giữa file, inode và file_operations

Ba cấu trúc này tạo nên chuỗi liên kết logic giữa kernel và driver khi người dùng thao tác với thiết bị:

1. Khi người dùng mở thiết bị (ví dụ *open("/dev/scull0", O_RDWR)*):
    - Kernel tìm *inode* tương ứng với tệp */dev/scull0* trong hệ thống tệp */dev*.
    - *inode->i_cdev* cho biết thiết bị này thuộc về driver nào.
    - Kernel tạo mới struct file để đại diện cho phiên mở và gán *filp->f_op = inode->i_cdev->ops* (tức là bảng *file_operations* của driver).
2. Khi driver thực thi hàm *open()*:
    - Hàm *open(inode, file)* được gọi.
    - Driver thường dùng macro *container_of(inode->i_cdev, struct scull_dev, cdev)* để lấy con trỏ tới thiết bị cụ thể.
    - Con trỏ này được gán vào *file->private_data* để các hàm *read()* và *write()* có thể truy cập lại thiết bị.
3. Khi thực hiện đọc/ghi *(read()* hoặc *write()*):
    - Kernel gọi *file->f_op->read()* hoặc *file->f_op->write()*.
    - Driver truy xuất thiết bị thông qua *file->private_data* và cập nhật vị trí *f_pos*

4. ĐĂNG KÝ CHAR DEVICE

Sau khi đã có major và minor number, cùng bảng phương thức *file_operations*, bước tiếp theo trong quá trình xây dựng driver là đăng ký thiết bị ký tự với Kernel.

Việc "đăng ký" này giúp kernel biết rằng một dải số hiệu thiết bị nào đó (ví dụ (240, 0–3)) sẽ được xử lý bởi driver của ta, và mỗi khi người dùng mở /dev/scull0, kernel có thể định tuyến lời gọi đến các hàm trong driver tương ứng (open(), read(), write()...).

Để thực hiện việc đăng ký này, Linux sử dụng cấu trúc dữ liệu trung tâm là ***struct cdev***.

4.1. struct cdev – Đại diện cho thiết bị ký tự trong kernel

Mỗi thiết bị ký tự trong hệ thống đều được kernel biểu diễn bởi một đối tượng struct cdev. Cấu trúc này lưu giữ thông tin liên kết giữa:

- Dải số hiệu thiết bị (major, minor);
- Bảng phương thức (file_operations);
- Và module driver tương ứng.

Khi người dùng thao tác với thiết bị, kernel sẽ sử dụng cdev để tìm đúng hàm xử lý mà driver đã đăng ký.

Cấu trúc tổng quát (được định nghĩa trong <linux/cdev.h>):

```
struct cdev {
    struct kobject kobj;     /* đối tượng thuộc hệ thống kobject */
    struct module *owner;          /* module sở hữu thiết bị */
    const struct file_operations *ops;  /* bảng phương thức thao tác */
    dev_t dev;                  /* số hiệu thiết bị (major, minor) */
    unsigned int count;      /* số lượng minor mà cdev quản lý */
};
```

Trong thực tế, ta không cần truy cập trực tiếp vào các trường này. Kernel cung cấp một bộ API chuẩn để khởi tạo, thêm, và xóa thiết bị char thông qua ***cdev_init(), cdev_add(),*** và ***cdev_del().***

4.2. Các API đăng ký và gỡ cdev

4.2.1. Khởi tạo cdev với cdev_init()

Trước khi thêm thiết bị vào hệ thống, ta cần khởi tạo cấu trúc cdev và liên kết nó với bảng *file_operations*.

Cú pháp:

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

Giải thích tham số:

- cdev: con trỏ tới cấu trúc struct cdev mà ta muốn khởi tạo (thường là biến thành viên trong cấu trúc thiết bị, ví dụ struct scull_dev).
- fops: con trỏ tới bảng phương thức của driver (struct file_operations), nơi chứa các hàm xử lý thao tác (open, read, write...).

Ví dụ:

```
cdev_init(&dev->cdev, &scull_fops);
dev->cdev.owner = THIS_MODULE;
```

Ở đây:

- cdev_init() liên kết thiết bị dev với bảng hàm scull_fops.
- Trường owner được gán bằng THIS_MODULE để kernel biết module nào đang sở hữu thiết bị này, giúp ngăn việc gỡ module trong khi thiết bị còn đang được sử dụng.

4.2.2. Thêm thiết bị vào hệ thống với cdev_add()

Sau khi khởi tạo, cdev vẫn chưa tồn tại trong hệ thống. Để kernel thật sự nhận biết, ta cần thực hiện bước tiếp theo là cdev_add().Khi cdev đã được khởi tạo, ta cần đăng ký nó với kernel để chính thức kích hoạt thiết bị ký tự.

Cú pháp:

```
int cdev_add(struct cdev *cdev, dev_t first, unsigned int count);
```

Giải thích tham số:

- cdev: con trỏ tới cấu trúc cdev đã được khởi tạo bằng cdev_init().
- first: số hiệu thiết bị đầu tiên trong dải mà cdev quản lý (thường tạo bằng MKDEV(major, minor)).
- count: số lượng minor liên tiếp mà thiết bị sẽ quản lý (thường là 1 với mỗi thiết bị con).

Giá trị trả về:

- 0 nếu thành công.
- Mã lỗi âm nếu thất bại (ví dụ -EBUSY nếu số hiệu đã được dùng).

Ví dụ:

```
dev_t devno = MKDEV(major, minor);
int err;

err = cdev_add(&dev->cdev, devno, 1);
if (err)
    pr_err("Error %d adding cdev for device %d\n", err, minor);
```

Lưu ý:

Sau khi cdev_add() thành công, thiết bị đã được kernel nhận biết. Từ thời điểm này, driver có thể nhận các lời gọi open(), read(),... từ user-space. Vì vậy, hãy đảm bảo rằng toàn bộ dữ liệu nội bộ và đồng bộ hóa (mutex, buffer,...) đã sẵn sàng trước khi gọi cdev_add().

4.2.3. Gỡ bỏ thiết bị với cdev_del()

Khi driver được gỡ khỏi kernel, hoặc khi ta muốn hủy thiết bị, cần xóa cdev khỏi hệ thống.

Cú pháp:

```
void cdev_del(struct cdev *cdev);
```

Ý nghĩa:

- Gỡ cdev khỏi hệ thống, ngăn kernel tiếp tục gọi vào bảng file_operations của thiết bị.
- Sau khi gọi *cdev_del()*, ta có thể giải phóng bộ nhớ hoặc dọn dữ liệu thiết bị.

Ví dụ:

```
cdev_del(&dev->cdev);
```

4.3. Tạo node trong /dev (device file)

4.3.1. Khái niệm

Sau khi thiết bị ký tự (char device) được đăng ký thành công với kernel bằng *cdev_add()*, kernel đã biết rằng cặp số hiệu (major, minor) đó thuộc về driver của bạn.

Tuy nhiên, lúc này người dùng vẫn chưa thể truy cập thiết bị – vì chưa có một tệp tương ứng trong thư mục /dev.

"Tạo node trong /dev" chính là bước tạo tệp thiết bị (device file) để ứng dụng người dùng có thể sử dụng các lời gọi quen thuộc như *open(), read(), write()* trên thiết bị.

Node trong /dev đóng vai trò như một cửa ngõ giao tiếp giữa user space và kernel space:

- Khi người dùng mở */dev/scull0*, kernel tra trong node này để biết driver nào (theo major number) và thiết bị con nào (theo minor number) cần xử lý.
- Sau đó, kernel định tuyến yêu cầu tới đúng hàm trong bảng *file_operations* của driver (đã đăng ký trước đó).

4.3.2. Thứ tự tạo node trong quy trình khởi tạo

Trong hàm khởi tạo (init) của driver, bước tạo node phải được thực hiện ngay sau khi *cdev_add()* thành công. Vì chỉ khi *cdev_add()* hoàn tất, kernel mới "biết" thiết bị này tồn tại và có thể xử lý lời gọi từ user-space.

Trình tự khởi tạo chuẩn trong một char driver gồm:

1. Cấp phát số hiệu thiết bị: *alloc_chrdev_region()*
2. Khởi tạo cấu trúc cdev: *cdev_init()*
3. Đưa thiết bị vào hệ thống: *cdev_add()*
4. Tạo node trong */dev*: cho phép chương trình người dùng mở thiết bị
5. *(Tuỳ chọn)* In thông tin log và kiểm tra */proc/devices*

Nếu tạo node trước *cdev_add(), node /dev/...* sẽ xuất hiện nhưng kernel chưa biết driver nào xử lý thiết bị đó, dẫn đến lỗi khi người dùng thao tác (*No such device* hoặc treo hệ thống).

4.3.3. Tạo node thủ công bằng mknod

Trong môi trường Linux cơ bản hoặc khi hệ thống chưa có *udev*, bạn có thể tạo node thủ công bằng lệnh *mknod*.

Cú pháp:

```
sudo mknod /dev/<tên_node> c <major> <minor>
```

Giải thích tham số:

- /dev/<tên_node> – tên node bạn muốn tạo, ví dụ /dev/scull0.

- c – chỉ định đây là character device.
- <major> – số hiệu major mà driver đã đăng ký (in trong dmesg hoặc /proc/devices).
- <minor> – số hiệu minor của thiết bị con.

Ví dụ:

```
sudo mknod /dev/scull0 c 240 0
sudo mknod /dev/scull1 c 240 1
sudo chmod 666 /dev/scull*
```

Kiểm tra node vừa tạo:

```
ls -l /dev/scull*
crw-rw-rw- 1 root root 240, 0 Oct 7 14:30 /dev/scull0
crw-rw-rw- 1 root root 240, 1 Oct 7 14:30 /dev/scull1
```

Phương pháp này giúp hiểu rõ cơ chế major/minor nhưng cần biết chính xác số hiệu, phải tự phân quyền, và phải làm thủ công lại mỗi khi driver được nạp lại (vì major có thể thay đổi khi cấp phát động).

4.3.4. Tạo node tự động bằng class_create() và device_create()

Hầu hết các hệ thống Linux hiện nay đều sử dụng *udev* hoặc *devtmpfs*, cho phép driver tự động tạo node trong */dev*. Phương pháp này vừa an toàn, vừa tiện lợi, không cần can thiệp thủ công.

a) Tạo lớp thiết bị bằng class_create()

Đầu tiên, bạn cần tạo một lớp thiết bị (struct class) – đại diện cho một nhóm thiết bị cùng loại (ví dụ tất cả thiết bị scull).

Cú pháp:

```
struct class *class_create(struct module *owner, const char *name);
```

Giải thích tham số:

- owner – con trỏ tới module sở hữu lớp thiết bị (thường là THIS_MODULE).
- name – tên lớp thiết bị, cũng sẽ xuất hiện trong */sys/class/*.

Giá trị trả về:

- Con trỏ *struct class* * nếu thành công.
- Con trỏ lỗi (*ERR_PTR()*) nếu thất bại, kiểm tra bằng *IS_ERR()*.

Ví dụ:

```
scull_class = class_create(THIS_MODULE, "scull");
if (IS_ERR(scull_class)) {
    pr_err("Failed to create class scull\n");
    return PTR_ERR(scull_class);
}
```

Sau lời gọi này, kernel sẽ tạo thư mục */sys/class/scull/*, đại diện cho nhóm thiết bị "scull".

b) Tạo node thiết bị bằng device_create()

Khi bạn đã có class, bước tiếp theo là tạo từng thiết bị (device node) trong /dev. Hàm device_create() yêu cầu kernel và udev tạo node tương ứng với dev_t mà bạn truyền vào.

Cú pháp:

```
struct device *device_create(struct class *class,
                             struct device *parent,
                             dev_t devt,
                             void *drvdata,
                             const char *fmt, ...);
```

Giải thích tham số:

- class – lớp thiết bị được tạo bằng class_create().
- parent – thiết bị cha (thường để NULL cho thiết bị đơn giản).
- devt – số hiệu thiết bị (dev_t, gồm major và minor).
- drvdata – dữ liệu tuỳ chọn, thường để NULL.
- fmt – chuỗi định dạng tên node, giống như printf (ví dụ "scull%d").
- Các tham số sau fmt là đối số tương ứng (ví dụ i trong "scull%d").

Giá trị trả về:

- Con trỏ struct device * nếu thành công.
- Con trỏ lỗi (ERR_PTR()) nếu thất bại.

Ví dụ:

```
device_create(scull_class, NULL,
              MKDEV(MAJOR(scull_devno), MINOR(scull_devno) + i),
              NULL, "scull%d", i);
```

Giải thích:

- scull_class – lớp "scull" vừa tạo.

- NULL – không có thiết bị cha.
- MKDEV(...) – tạo số hiệu (major, minor) cho thiết bị thứ i.
- "scull%d" – đặt tên node /dev/scull0, /dev/scull1, ...

Ngay sau lời gọi này, node */dev/scull<i>* sẽ xuất hiện tự động, nhờ cơ chế của *devtmpfs/udev*.

c) Xoá node khi gỡ module

Trong hàm exit, bạn **cần xoá node và lớp thiết bị theo thứ tự ngược**:

```
device_destroy(scull_class, MKDEV(MAJOR(scull_devno), i));
class_destroy(scull_class);
```

Nếu có nhiều thiết bị con, hãy lặp vòng for để gọi *device_destroy()* cho từng minor.

4.3.5. So sánh hai phương pháp tạo node

Tiêu chí	**mknod thủ công**	**class_create() + device_create() (tự động)**
Cách tạo	Người dùng tạo thủ công qua dòng lệnh mknod	Driver tự tạo node qua API kernel
Yêu cầu biết trước	Cần biết major, minor (từ /proc/devices)	Không cần, kernel/udev tự xử lý
Tự động khi nạp/gỡ module	Không tự động, phải chạy lại lệnh	Có tự động, node xuất hiện và biến mất tự động
Khả năng mở rộng	Khó quản lý khi nhiều thiết bị con	Dễ dàng – chỉ cần lặp device_create()
Phân quyền, tên node	Người dùng tự đặt thủ công	udev/devtmpfs tự quản lý
Phù hợp với	Hệ thống nhúng tối giản, học lệnh cơ bản	Hệ thống Linux hiện đại (chuẩn khuyến nghị)

Trong các bản kernel hiện nay, class_create() + device_create() là phương pháp chuẩn và khuyến nghị để tạo node trong /dev. Nó giúp driver hoạt động ổn định, tương thích với udev/devtmpfs, và giảm đáng kể thao tác thủ công khi thử nghiệm hoặc phân phối module.

4.5. Ví dụ trong scull – Đưa cdev vào hệ thống và tháo/gỡ thiết bị

4.5.1. Cấu trúc scull_dev

Trong driver scull, mỗi thiết bị con (minor) được mô tả bởi một cấu trúc dữ liệu có tên *struct scull_dev*.

Cấu trúc này lưu trữ toàn bộ thông tin cần thiết để quản lý một thiết bị ký tự riêng biệt, bao gồm vùng nhớ, các tham số cấu hình, khóa đồng bộ, và đặc biệt là một cấu trúc struct cdev dùng để đăng ký thiết bị với kernel.

*Định nghĩa rút gọn trong mã **scull.c**:*

```
struct scull_dev {
    struct scull_qset *data;  /* vùng dữ liệu (bộ nhớ thiết bị) */
    int quantum; /* kích thước mỗi khối dữ liệu (mặc định 4 KB) */
    int qset;      /* số phần tử trong một qset */
    unsigned long size;    /* tổng số byte đã ghi vào thiết bị */
    struct cdev cdev;     /* cấu trúc đại diện cho thiết bị ký tự
*/
    struct mutex lock;    /* khóa đồng bộ truy cập thiết bị */
};
```

Giải thích:

- data: con trỏ tới danh sách vùng nhớ (qset) mà driver sử dụng để lưu dữ liệu của thiết bị.
- quantum và qset: định nghĩa cách tổ chức bộ nhớ (chi tiết ở phần sau).
- size: tổng kích thước dữ liệu hiện có trong thiết bị.
- cdev: thành phần trung tâm giúp kernel liên kết thiết bị này với các hàm trong file_operations.
- lock: khóa bảo vệ chống truy cập đồng thời từ nhiều tiến trình.

Như vậy, mỗi phần tử *scull_dev* trong mảng *scull_devices[]* chính là một thiết bị ký tự độc lập – tương ứng với một node */dev/scull<i>* trong hệ thống.

4.5.2. Đưa cdev vào hệ thống qua scull_setup_cdev() (per-minor)

Sau khi đã xin dải số hiệu thiết bị (major/minor) và khởi tạo các tham số cơ bản, scull sẽ đưa từng thiết bị con vào hệ thống thông qua hàm scull_setup_cdev(). Mỗi thiết bị con tương ứng với một minor number riêng, có struct cdev và node /dev/scull<i> riêng biệt.

*Mã nguồn rút gọn của hàm **scull_setup_cdev()**:*

```
static void scull_setup_cdev(struct scull_dev *dev, int index, dev_t
base)
{
    int err;
    dev_t devno = MKDEV(MAJOR(base), MINOR(base) + index);

    /* Khởi tạo cấu trúc cdev và liên kết với bảng file_operations */
    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;

    /* Đưa thiết bị vào hệ thống kernel */
    err = cdev_add(&dev->cdev, devno, 1);
    if (err)
        pr_err("scull: error %d adding cdev %d\n", err, index);
}
```

Giải thích chi tiết:

- devno = MKDEV(MAJOR(base), MINOR(base) + index)
  → Tạo số hiệu thiết bị (major, minor) cho từng thiết bị con. Ví dụ: nếu base = (240, 0) và index = 2 → devno = (240, 2) tương ứng /dev/scull2.
- cdev_init(&dev->cdev, &scull_fops)
  → Gắn bảng phương thức scull_fops cho thiết bị dev. Đây là "bản đồ" để kernel biết driver xử lý các lời gọi open(), read(), write() như thế nào.
- dev->cdev.owner = THIS_MODULE

→ Cho kernel biết module nào sở hữu thiết bị, giúp ngăn việc gỡ module khi thiết bị vẫn đang mở.

- cdev_add(&dev->cdev, devno, 1)
  → Đăng ký thiết bị với kernel. Từ thời điểm này, thiết bị được xem là đang hoạt động (live), có thể nhận các lời gọi từ user-space.

Tạo node tự động /dev/scull*

Ngay sau khi *cdev_add()* thành công, scull gọi hàm *device_create()* để yêu cầu kernel (và udev/devtmpfs) tự động tạo file thiết bị tương ứng trong /dev.

Ví dụ trong scull_init():

```
device_create(scull_class, NULL,
              MKDEV(MAJOR(scull_devno), MINOR(scull_devno) + i),
              NULL, "scull%d", i);
```

Giải thích cú pháp:

- scull_class: lớp thiết bị được tạo trước bằng class_create(THIS_MODULE, "scull").
- NULL: không có thiết bị cha.
- MKDEV(...): số hiệu thiết bị (major, minor) cho thiết bị con thứ i.
- "scull%d": tên node trong /dev, ví dụ /dev/scull0, /dev/scull1, ...

Ngay sau lời gọi này, node */dev/scull<i>* xuất hiện tự động và sẵn sàng cho người dùng thao tác.

Tóm tắt:

- cdev_add() đăng ký thiết bị với kernel.
- device_create() tạo node tương ứng trong /dev.

Hai bước này luôn đi liền nhau trong mỗi lần khởi tạo thiết bị con.

4.5.3. Tháo/gỡ thiết bị khi gỡ module

Khi module được gỡ khỏi kernel (*rmmod scull*), tất cả các thiết bị và node /dev phải được dọn sạch theo thứ tự ngược với quá trình khởi tạo để tránh lỗi hoặc rò rỉ tài nguyên.

*Mã nguồn rút gọn trong hàm **scull_exit()**:*

```
static void __exit scull_exit(void)
{
```

```
    int i;

    /* Gỡ từng thiết bị con */
    for (i = 0; i < scull_nr_devs; i++) {
        device_destroy(scull_class,
                MKDEV(MAJOR(scull_devno), MINOR(scull_devno) + i));
        cdev_del(&scull_devices[i].cdev);
    }

    /* Hủy lớp thiết bị và giải phóng số hiệu */
    class_destroy(scull_class);
    unregister_chrdev_region(scull_devno, scull_nr_devs);

    pr_info("scull: unloaded successfully\n");
}
```

Giải thích chi tiết:

1. device_destroy()
    - Xóa node /dev/scull<i> ra khỏi hệ thống (udev/devtmpfs sẽ tự cập nhật lại /dev).
    - Thao tác này được thực hiện trước để đảm bảo người dùng không thể mở thiết bị trong khi đang gỡ.
2. cdev_del()
    - Gỡ thiết bị khỏi hệ thống kernel, xóa liên kết giữa (major, minor) và driver.
3. class_destroy()
    - Hủy lớp thiết bị đã tạo bằng class_create().
    - Giải phóng thư mục tương ứng trong /sys/class/scull/.
4. unregister_chrdev_region()
    - Trả lại dải số hiệu (major, minor) cho kernel.
    - Nếu bỏ qua bước này, các số hiệu cũ sẽ bị "chiếm dụng", gây lỗi khi nạp lại module.
5. pr_info()

- Ghi log thông báo việc gỡ module thành công, tiện kiểm tra trong dmesg.

Thứ tự ***device_destroy → cdev_del → class_destroy → unregister_chrdev_region*** là thứ tự chuẩn và an toàn nhất để đảm bảo quá trình tháo/gỡ driver không để lại node mồ côi hoặc vùng nhớ chưa giải phóng.

4.5.4. Trình tự chuẩn khi khởi tạo và gỡ thiết bị

Bảng dưới đây tóm tắt quy trình chuẩn mà driver scull (và hầu hết char driver hiện đại) tuân theo:

Thao tác	Hàm API	Mục đích
Khởi tạo (init)		
1	alloc_chrdev_region()	Cấp phát số hiệu thiết bị (major/minor).
2	class_create()	Tạo lớp thiết bị đại diện cho nhóm driver.
3	cdev_init()	Khởi tạo cấu trúc cdev, gắn với file_operations.
4	cdev_add()	Đưa thiết bị vào hệ thống kernel.
5	device_create()	Tạo node /dev/scull<i> để user-space truy cập.
Tháo/gỡ (exit)		
1	device_destroy()	Xóa node /dev/scull<i> khỏi /dev.
2	cdev_del()	Gỡ thiết bị khỏi kernel.
3	class_destroy()	Hủy lớp thiết bị.
4	unregister_chrdev_region()	Trả lại số hiệu thiết bị cho hệ thống.

5. CÁC PHƯƠNG THỨC OPEN VÀ RELEASE

Trong lập trình thiết bị ký tự, hai phương thức *open()* và *release()* đóng vai trò tương tự như việc mở và đóng tệp trong không gian người dùng. Đây là hai điểm vào (entry point) cơ bản mà kernel gọi mỗi khi một tiến trình bắt đầu hoặc kết thúc làm việc với thiết bị.

Các phương thức này giúp driver:

- Xác định *thiết bị cụ thể* đang được truy cập.
- Chuẩn bị và dọn dẹp *trạng thái phiên làm việc* (per-open state).

- Thực hiện các thao tác khởi tạo hoặc kết thúc đặc thù cho thiết bị.

Cả hai được đăng ký trong cấu trúc *file_operations*, và sẽ được gọi tự động khi người dùng thực hiện các thao tác tương ứng.

5.1. Phương thức open()

5.1.1. Vai trò và cơ chế hoạt động

Khi một tiến trình gọi open("/dev/scull0", O_RDWR) hoặc sử dụng một lệnh tương đương (ví dụ cat /dev/scull0), hệ thống tệp ảo (VFS) của Linux sẽ:

1. Tìm node thiết bị tương ứng trong /dev.
2. Tra số hiệu (major, minor) để xác định driver nào xử lý thiết bị đó.
3. Tạo cấu trúc struct file đại diện cho phiên làm việc mới.
4. Gọi hàm *open()* mà driver đã đăng ký trong file_operations.

Mục tiêu của hàm *open()* trong driver là:

- *Xác định thiết bị cụ thể* đang được mở (theo minor number).
- *Gắn con trỏ thiết bị* vào *filp->private_data* để các hàm *read(), write(), llseek()* có thể truy cập nhanh vào đúng vùng dữ liệu của thiết bị.
- (Nếu cần) *khởi tạo hoặc làm sạch dữ liệu* khi mở thiết bị ở chế độ ghi *(O_WRONLY).*

Cú pháp:

```
int open(struct inode *inode, struct file *filp);
```

Giải thích tham số:

- inode: đại diện cho tệp thiết bị trong VFS; chứa thông tin về số hiệu (major, minor) và con trỏ tới struct cdev của driver.
- filp: đại diện cho phiên làm việc (lần mở tệp) của tiến trình; driver có thể lưu thông tin phiên làm việc vào đây (qua trường private_data).

Cơ chế xác định thiết bị:

Khi có nhiều thiết bị con (nhiều minor), driver cần xác định *thiết bị nào* đang được mở. Trong scull, điều này thực hiện bằng cách dùng macro *container_of* để truy ngược từ *inode->i_cdev* (do kernel lưu sẵn) về cấu trúc thiết bị *struct scull_dev*:

```
struct scull_dev *dev;
dev = container_of(inode->i_cdev, struct scull_dev, cdev);
```

Ví dụ cài đặt trong scull.c

Hàm *scull_open()* trong mã nguồn *scull.c* minh họa đầy đủ các thao tác cần thiết trong quá trình mở thiết bị:

```
static int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev;

    /* 1. Lấy con trỏ tới thiết bị cụ thể từ inode */
    dev = container_of(inode->i_cdev, struct scull_dev, cdev);

    /* 2. Gắn con trỏ thiết bị vào file->private_data */
    filp->private_data = dev;

    /* 3. Nếu mở ở chế độ ghi, xóa dữ liệu cũ (truncate) */
    if ((filp->f_flags & O_ACCMODE) == O_WRONLY) {
        mutex_lock(&dev->lock);
        scull_trim(dev);  // đặt kích thước dữ liệu về 0
        mutex_unlock(&dev->lock);
    }
    return 0;
}
```

Giải thích:

- Dòng *container_of(...)* cho phép driver tìm ra đúng đối tượng thiết bị tương ứng với minor của file được mở.
- Việc gắn *filp->private_data = dev* giúp các phương thức khác (*read/write/llseek*) truy cập thiết bị nhanh hơn mà không cần tính toán lại.
- Nếu thiết bị được mở với cờ O_WRONLY, dữ liệu cũ sẽ bị xóa bằng cách gọi *scull_trim()*, đảm bảo thiết bị bắt đầu ở trạng thái rỗng (giống như khi mở file thường để ghi đè).

- Các thao tác xóa dữ liệu được bảo vệ bằng *mutex (dev->lock)* để tránh xung đột nếu nhiều tiến trình cùng truy cập.

Lưu ý:

- Hàm *open()* có thể được gọi *nhiều lần* nếu nhiều tiến trình cùng mở cùng một node */dev/...*. Mỗi tiến trình sẽ có một bản sao *struct file* riêng, với *private_data* riêng.
- Nếu cần thực hiện thao tác khởi tạo chỉ một lần cho "thiết bị toàn cục", bạn có thể dùng bộ đếm mở (open count) trong *struct scull_dev* để biết liệu đây có phải là lần mở đầu tiên hay không.
- Trong các thiết bị phần cứng thật, *open()* còn có thể kích hoạt thiết bị hoặc bật nguồn. Trong scull (thiết bị mô phỏng bằng bộ nhớ RAM), việc này không cần thiết.

5.2. Phương thức release()

5.2.1. Vai trò và cơ chế hoạt động

Phương thức *release() (*đôi khi gọi là *close()*) là đối nghịch với *open()*. Nó được kernel gọi khi tiến trình đóng thiết bị hoặc khi phiên làm việc cuối cùng với thiết bị kết thúc.

Lưu ý:

- Mỗi lần tiến trình gọi *open()* không nhất thiết dẫn đến lời gọi *release()*.
- Kernel chỉ gọi release() khi *bộ đếm tham chiếu (reference count)* của struct file giảm về 0 — nghĩa là không còn tiến trình nào giữ file descriptor này (sau dup() hoặc fork()).
- Điều này đảm bảo release() chỉ được thực thi *một lần* cho mỗi open() thực sự.

Cú pháp:

```
int release(struct inode *inode, struct file *filp);
```

Giải thích tham số:

- inode: trỏ tới đối tượng tệp trong VFS, tương tự như trong open().
- filp: con trỏ tới phiên làm việc (struct file) đang được đóng.

Driver thường dùng hàm này để:

- Giải phóng bộ nhớ hoặc tài nguyên được cấp phát trong open().
- Ngắt kết nối hoặc tắt thiết bị phần cứng khi không còn người dùng.
- Ghi log hoặc thực hiện các thao tác dọn dẹp.
-

Ví dụ: cài đặt trong scull.c

Trong scull, thiết bị chỉ dùng bộ nhớ trong RAM, không có tài nguyên đặc biệt, nên *release()* rất đơn giản:

```
static int scull_release(struct inode *inode, struct file *filp)
{
    return 0;
}
```

Ở đây:

- Không cần giải phóng private_data vì nó chỉ trỏ tới scull_dev — tồn tại trong suốt vòng đời module.
- Không có tài nguyên động nào cấp phát trong open(), nên việc dọn dẹp là không cần thiết.
- Nếu driver khác có cấp phát vùng nhớ tạm thời trong open(), thì đây sẽ là nơi giải phóng nó.

6. PHƯƠNG THỨC READ VÀ WRITE

6.1 Cấu trúc bộ nhớ trong scul

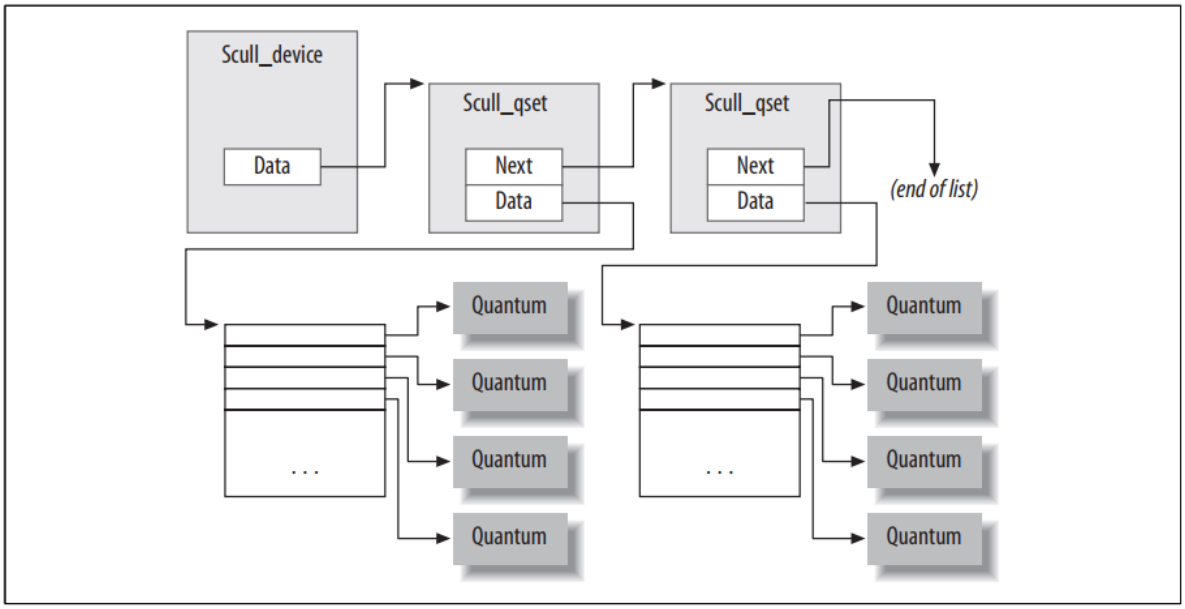
Trong các driver thiết bị ký tự thông thường, dữ liệu có thể được lưu trữ trực tiếp trên phần cứng (như RAM, EEPROM, hay buffer FIFO). Tuy nhiên, *scull* là một thiết bị *mô phỏng bằng phần mềm*, không có phần cứng thật. Thay vào đó, scull sử dụng *vùng nhớ động trong RAM* để biểu diễn dữ liệu của thiết bị.Mục tiêu của cơ chế này là giúp người học hiểu cách quản lý bộ nhớ trong kernel, cách cấp phát, thu hồi, và tổ chức dữ liệu sao cho có thể mô phỏng hành vi của một thiết bị thật – tức là có thể *đọc/ghi* dữ liệu tuần tự, có kích thước giới hạn, và có thể *tăng kích thước* khi cần. Vì vậy, bộ nhớ của mỗi thiết bị scull được quản lý dưới dạng *cấu trúc động*, gồm nhiều phần tử nhỏ được cấp phát khi cần thiết và giải phóng khi không còn sử dụng.

Mô hình tổ chức bộ nhớ của scull

Bộ nhớ của scull được chia thành các khối logic, tổ chức theo ba cấp độ chính:

1. **Quantum**: là đơn vị nhỏ nhất của dữ liệu trong scull. Mỗi quantum là một khối nhớ có kích thước cố định (mặc định là *4 KB*). Đây là vùng nhớ thực tế chứa dữ liệu mà người dùng ghi vào thiết bị.
2. **Quantum set (qset)**: là một *mảng con trỏ*, trong đó mỗi phần tử trỏ tới một quantum. Mỗi qset có thể chứa nhiều quantum (ví dụ *100 quantum*). Cấu trúc này giúp driver dễ dàng cấp phát thêm quantum mới khi dữ liệu tăng dần.
3. **Danh sách qset**: nhiều qset được *liên kết với nhau* bằng con trỏ next để tạo thành một *danh sách liên kết đơn*, mô phỏng một thiết bị có khả năng lưu trữ dung lượng lớn hơn nhiều lần so với chỉ một qset.

Nói cách khác, bộ nhớ trong scull được tổ chức như một ***danh sách các qset***, mỗi qset là một "hộp chứa" gồm nhiều "ngăn nhỏ" (quantum) chứa dữ liệu thực tế.



Các cấu trúc dữ liệu chính

Để quản lý bộ nhớ theo mô hình trên, scull định nghĩa hai cấu trúc dữ liệu cơ bản trong mã nguồn:

struct scull_qset

Đại diện cho ***một qset***, bao gồm:

- Mảng con trỏ data, trỏ tới các quantum thực tế.
- Con trỏ next, trỏ tới qset kế tiếp trong danh sách.

```
struct scull_qset {
    void **data;                /* Mảng con trỏ tới các quantum */
    struct scull_qset *next;    /* Liên kết tới qset tiếp theo */
};
```

Mỗi phần tử data[i] là một con trỏ trỏ tới vùng nhớ thật (kmalloc) có kích thước quantum byte.

Khi cần mở rộng bộ nhớ, driver chỉ việc cấp phát thêm một struct scull_qset mới và nối vào danh sách.

struct scull_dev

Đây là *cấu trúc trung tâm* của mỗi thiết bị scull.Mỗi thiết bị con (minor) có một struct scull_dev riêng, chứa toàn bộ thông tin quản lý bộ nhớ, thông tin thiết bị và đồng bộ hóa.

```
struct scull_dev {
    struct scull_qset *data;    /* Con trỏ tới qset đầu tiên (danh sách liên
kết) */
    int quantum;                /* Kích thước mỗi khối dữ liệu (quantum) */
    int qset;                   /* Số lượng quantum trong mỗi qset */
    unsigned long size;         /* Tổng số byte dữ liệu hiện có */
    struct cdev cdev;          /* Cấu trúc thiết bị ký tự (đăng ký với kernel)
*/
    struct mutex lock;          /* Khóa bảo vệ đồng bộ truy cập thiết bị */
};
```

Các giá trị quantum và qset có thể được đặt thông qua *module parameter*, giúp người dùng thay đổi kích thước khối dữ liệu mà không cần biên dịch lại driver.

Ví dụ:

static int scull_quantum = 4000;

static int scull_qset = 1000;

module_param(scull_quantum, int, 0);

module_param(scull_qset, int, 0);

6.2 Trao đổi dữ liệu giữa user space và kernel space

Trong hệ điều hành Linux, không gian bộ nhớ được chia tách rõ ràng thành hai phần:

- ***User space*** – nơi các chương trình người dùng (như cat, echo, hay ứng dụng C) hoạt động.
- ***Kernel space*** – nơi nhân hệ điều hành và các driver như scull thực thi.

Sự phân tách này mang ý nghĩa cực kỳ quan trọng nhằm đảm bảo *tính ổn định và an toàn của hệ thống*: một chương trình lỗi hoặc độc hại ở user space *không thể trực tiếp truy cập bộ nhớ kernel*, tránh việc ghi đè hay phá vỡ dữ liệu nội bộ của nhân. Khi một tiến trình user gọi các lời gọi hệ thống như read() hoặc write() lên tệp /dev/scull0, dữ liệu cần được *sao chép qua lại giữa hai không gian nhớ riêng biệt* này. Do đó, một trong những nhiệm vụ cốt lõi của lập trình driver là *thực hiện việc truyền dữ liệu an toàn và đúng quy tắc* giữa *user space* ↔ *kernel space*.

Trong driver, tham số buf của các hàm read() hoặc write() mà kernel truyền vào *không phải là con trỏ bình thườn*g. Nó trỏ tới một địa chỉ trong *user space*, mà kernel *không được phép truy cập trực tiếp* vì:

1. Không gian nhớ tách biệt: User space và kernel space được ánh xạ trong các vùng địa chỉ khác nhau. Kernel không thể (và không nên) truy cập trực tiếp địa chỉ user-space, vì địa chỉ đó có thể không hợp lệ trong ngữ cảnh kernel.
2. Bảo vệ bộ nhớ: Kernel hoạt động ở đặc quyền cao (ring 0). Nếu driver dereference trực tiếp con trỏ từ user, một lỗi nhỏ (ví dụ con trỏ rỗng hoặc sai địa chỉ) có thể làm hệ thống crash toàn bộ (kernel panic).
3. Quản lý trang (paging): Bộ nhớ user-space có thể đã bị swap ra đĩa. Nếu kernel truy cập trực tiếp, sẽ gây lỗi page fault trong ngữ cảnh mà kernel không thể phục hồi.
4. Bảo mật: Kernel không thể tin tưởng con trỏ do user gửi — một tiến trình có thể cố tình gửi địa chỉ trỏ vào vùng nhạy cảm của kernel nhằm chiếm quyền điều khiển hệ thống.

Linux cung cấp hai hàm API trong *<asm/uaccess.h>* để truyền dữ liệu giữa user-space và kernel-space một cách an toàn. Đây là những hàm nền tảng mà mọi driver ký tự (bao gồm scull) đều sử dụng khi cài đặt các phương thức read() và write().

6.2.1. Hàm copy_to_user()

Được dùng để **gửi dữ liệu từ kernel sang user space** – thường dùng trong hàm read().

```
unsigned long copy_to_user(void __user *to, const void *from, unsigned
                           long count);
```

Giải thích tham số:

- to – địa chỉ đích trong user space (vùng nhớ mà chương trình người dùng truyền vào).
- from – địa chỉ nguồn trong kernel space (nơi driver lưu dữ liệu).
- count – số byte cần sao chép.

Giá trị trả về:

- Trả về 0 nếu sao chép thành công toàn bộ dữ liệu.
- Trả về số byte chưa sao chép được nếu gặp lỗi (ví dụ con trỏ không hợp lệ).

Ví dụ sử dụng trong scull_read():

```
if (copy_to_user(buf, dev->data + offset, count))
    return -EFAULT;  // lỗi truy cập vùng nhớ người dùng
```

6.2.2. Hàm copy_from_user()

Được dùng để *nhận dữ liệu từ user space vào kernel* – thường xuất hiện trong hàm write().

```
unsigned long copy_from_user(void *to, const void __user *from,
                             unsigned long count);
```

Giải thích tham số:

- to – địa chỉ đích trong kernel (buffer nội bộ của driver).
- from – địa chỉ nguồn trong user space (vùng nhớ chứa dữ liệu từ chương trình người dùng).
- count – số byte cần sao chép.

Giá trị trả về:

- Trả về 0 nếu sao chép thành công.

- Trả về số byte còn lại *chưa được sao chép* nếu lỗi.

Ví dụ sử dụng trong scull_write():

```
if (copy_from_user(dev->data + offset, buf, count))
    return -EFAULT;  // lỗi: địa chỉ từ user-space không hợp lệ
```

6.3. Phương thức read và write

Cú pháp

```
ssize_t read (struct file *filp, char __user *buf, size_t count,
                          loff_t *ppos);
ssize_t write(struct file *filp, const char __user *buf, size_t count,
                          loff_t *ppos);
```

- *filp*: con trỏ tới phiên làm việc (struct file). Trong open(), ta đã gán filp->private_data = dev để các hàm read/write truy cập thiết bị cụ thể (minor) nhanh chóng.
- *buf*: bộ đệm ở user space.
    - read: driver ghi dữ liệu vào buf (dùng copy_to_user).
    - write: driver đọc dữ liệu từ buf (dùng copy_from_user).
- *count*: số byte mà user yêu cầu chuyển.
- *ppos:* vị trí đọc/ghi trong "tệp thiết bị". Driver phải cập nhật *ppos khi chuyển dữ liệu thành công (trừ các lời gọi kiểu pread/pwrite ở user-space, vốn không thay đổi vị trí chung).

Giá trị trả về:

- ≥ 0: số byte đã chuyển thành công (có thể ít hơn count – *partial transfer* là hợp lệ).
- 0 trong read: *EOF* (không còn dữ liệu).
- < 0: mã lỗi âm (thường gặp -EFAULT khi lỗi truy cập vùng nhớ user).

Lưu ý: buf là con trỏ user space, không dereference trực tiếp trong kernel; luôn dùng copy_to_user / copy_from_user.

Chức năng của hai phương thức

- **read**: lấy dữ liệu từ thiết bị đưa lên user-space. Trong scull, "thiết bị" là vùng nhớ động tổ chức theo *quantum/qset*; vì vậy read cần:

1. ánh xạ **ppos* → tìm đúng *qset/quantum/offset*,
2. sao chép tối đa *đến hết quantum hiện tại* sang buf,
3. cập nhật **ppos*, trả về số byte đã đọc (0 nếu EOF).

- **write**: nhận dữ liệu từ user-space ghi xuống thiết bị. Với scull, write có thể *mở rộng bộ nhớ*:
  1. ánh xạ *ppos → xác định vị trí cần ghi,
  2. *theo/cấp phát* qset, mảng con trỏ và quantum nếu chưa có,
  3. sao chép tối đa *đến hết quantum hiện tại*,
  4. cập nhật *ppos và *kích thước logi*c dev->size.

Cả hai phương thức đều *bảo vệ đồng bộ* (mutex) vì copy_*_user() có thể "ngủ", và các thao tác cấp phát/giải phóng bộ nhớ cần an toàn khi truy cập cạnh tranh.

CODE MẪU TRONG SCULL DRIVER

```
static ssize_t scull_read(struct file *filp, char __user *buf,
                          size_t count, loff_t *ppos)
{
    struct scull_dev   *dev = filp->private_data;
    struct scull_qset  *qs;
    int itemsize, item, rest, s_pos, q_pos;
    size_t to_copy;
    ssize_t ret = 0;

    /* (1) Đồng bộ: có thể bị tín hiệu ngắt → trả -ERESTARTSYS */
    if (mutex_lock_interruptible(&dev->lock))
        return -ERESTARTSYS;

    /* (2) EOF / rút gọn count nếu đọc vượt size hiện có */
    if (*ppos >= dev->size)
        goto out;                           // trả 0
    if (*ppos + count > dev->size)
        count = dev->size - *ppos;
```

```
    /* (3) Ánh xạ offset → (item, s_pos, q_pos) */
    itemsize = dev->quantum * dev->qset;   // dữ liệu mà 1 qset quản
lý
    item  = *ppos / itemsize;              // qset thứ mấy
    rest  = *ppos % itemsize;
    s_pos = rest / dev->quantum;             // chỉ số quantum trong
qset
    q_pos = rest % dev->quantum;           // offset trong 1 quantum

    /* (4) Lần theo danh sách tới qset cần đọc (không cấp phát mới) */
    qs = scull_follow(dev, item, false);
    if (!qs || !qs->data || !qs->data[s_pos])
        goto out;                              // "lỗ hổng" (chưa từng
ghi) → kết thúc

    /* (5) Chỉ đọc tối đa đến hết quantum hiện tại (partial read hợp
lệ) */
    to_copy = dev->quantum - q_pos;
    if (count < to_copy) to_copy = count;

    /* (6) Sao chép sang user space, lỗi → -EFAULT */
    if (copy_to_user(buf, (char*)qs->data[s_pos] + q_pos, to_copy)) {
        ret = -EFAULT;
        goto out;
    }

    /* (7) Cập nhật vị trí đọc và số byte trả về */
    *ppos += to_copy;
    ret = to_copy;

out:
    mutex_unlock(&dev->lock);
```

```
    return ret;                                  // 0 nếu EOF, >0 nếu đọc
được, âm nếu lỗi
}
```

Luồng xử lý trong scull_read

1. Khóa *dev->lock*.
2. Nếu **ppos* >= *size* → EOF (0).
3. Rút gọn count nếu vượt *size* - **ppos*.
4. Tính (*item, s_pos, q_pos*).
5. Theo qset (không cấp phát mới). Nếu rỗng → kết thúc.
6. Copy tối đa *đến hết quantum hiện tại* từ kernel → user (copy_to_user).
7. Cập nhật **ppos*, trả số byte đã đọc, mở khóa.

CODE MẪU – SCULL_WRITE

```
static ssize_t scull_write(struct file *filp, const char __user *buf,
                           size_t count, loff_t *ppos)
{
    struct scull_dev   *dev = filp->private_data;
    struct scull_qset  *qs;
    int itemsize, item, rest, s_pos, q_pos;
    size_t to_copy;
    ssize_t ret = -ENOMEM;

    /* (1) Đồng bộ: có thể bị tín hiệu ngắt → -ERESTARTSYS */
    if (mutex_lock_interruptible(&dev->lock))
        return -ERESTARTSYS;

    /* (2) Ánh xạ offset → (item, s_pos, q_pos) */
    itemsize = dev->quantum * dev->qset;
    item  = *ppos / itemsize;
    rest  = *ppos % itemsize;
```

```
    s_pos = rest / dev->quantum;
    q_pos = rest % dev->quantum;

    /* (3) Theo/cấp phát qset cần thiết (ghi có quyền mở rộng bộ nhớ)
*/
    qs = scull_follow(dev, item, true);
    if (!qs) goto out;

    /* (4) Bảo đảm mảng con trỏ quantum tồn tại */
    if (!qs->data) {
        qs->data = kcalloc(dev->qset, sizeof(void *), GFP_KERNEL);
        if (!qs->data) goto out;
    }

    /* (5) Bảo đảm quantum đích tồn tại */
    if (!qs->data[s_pos]) {
        qs->data[s_pos] = kzalloc(dev->quantum, GFP_KERNEL);
        if (!qs->data[s_pos]) goto out;
    }

    /* (6) Chỉ ghi tối đa đến hết quantum hiện tại (partial write hợp
lệ) */
    to_copy = dev->quantum - q_pos;
    if (count < to_copy) to_copy = count;

    /* (7) Sao chép từ user xuống kernel, lỗi → -EFAULT */
    if (copy_from_user((char*)qs->data[s_pos] + q_pos, buf, to_copy))
{
        ret = -EFAULT;
        goto out;
    }
```

```
    /* (8) Cập nhật vị trí và kích thước thiết bị */
    *ppos += to_copy;
    if (dev->size < *ppos) dev->size = *ppos;

    ret = to_copy;

out:
    mutex_unlock(&dev->lock);
    return ret;      // >0 số byte đã ghi; 0: chưa ghi; âm: lỗi
}
```

Luồng xử lý trong scull_write (tóm tắt)

1. Khóa dev->lock.
2. Tính *(item, s_pos, q_pos)* theo *ppos.
3. scull_follow(dev, item, true) để *theo/cấp phát* qset.
4. Bảo đảm có *mảng con trỏ quantum* và *quantum đích*.
5. Ghi tối đa *đến hết quantum hiện tại* (partial write).
6. copy_from_user user → kernel; lỗi -EFAULT nếu copy thất bại.
7. Cập nhật **ppos*, cập nhật *dev->size* nếu cần; mở khóa, trả số byte đã ghi.