

# LẬP TRÌNH KERNEL LINUX MODULES

Phần 1. Giới Thiệu Linux Kernel .....	2
Phần 2. Giới Thiệu Linux Kernel Module.....	5
Phần 3. Cách viết một Linux Kernel Module .....	7
Phần 4. Cơ bản về driver trong Linux.....	11
Phần 5. Character Device Driver .....	14
Phần 6. Các thao tác với file device .....	26

## Phần 1. Giới Thiệu Linux Kernel

Năm 1991, dựa trên UNIX kernel, Linus Torvalds đã tạo ra Linux kernel chạy trên máy tính của ông ấy. Dựa vào chức năng của hệ điều hành, Linux kernel được chia làm 6 thành phần (hình 3):

- **Process management:** có nhiệm vụ quản lý các tiến trình, bao gồm các công việc:
  - Tạo/hủy các tiến trình.
  - Lập lịch cho các tiến trình. Đây thực chất là lên kế hoạch: CPU sẽ thực thi chương trình khi nào, thực thi trong bao lâu, tiếp theo là chương trình nào.
  - Hỗ trợ các tiến trình giao tiếp với nhau.
  - Đồng bộ hoạt động của các tiến trình để tránh xảy ra tranh chấp tài nguyên.
- **Memory management:** có nhiệm vụ quản lý bộ nhớ, bao gồm các công việc:
  - Cấp phát bộ nhớ trước khi đưa chương trình vào, thu hồi bộ nhớ khi tiến trình kết thúc.
  - Đảm bảo chương trình nào cũng có cơ hội được đưa vào bộ nhớ.
  - Bảo vệ vùng nhớ của mỗi tiến trình.
- **Device management:** có nhiệm vụ quản lý thiết bị, bao gồm các công việc:
  - Điều khiển hoạt động của các thiết bị.
  - Giám sát trạng thái của các thiết bị.
  - Trao đổi dữ liệu với các thiết bị.
  - Lập lịch sử dụng các thiết bị, đặc biệt là thiết bị lưu trữ (ví dụ ổ cứng).
- **File system management:** có nhiệm vụ quản lý dữ liệu trên thiết bị lưu trữ (như ổ cứng, thẻ nhớ). Quản lý dữ liệu gồm các công việc: thêm, tìm kiếm, sửa, xóa dữ liệu.
- **Networking management:** có nhiệm vụ quản lý các gói tin (packet) theo mô hình TCP/IP.
- **System call Interface:** có nhiệm vụ cung cấp các dịch vụ sử dụng phần cứng cho các tiến trình. Mỗi dịch vụ được gọi là một **system call**.

Khi triển khai thực tế, mã nguồn của Linux kernel gồm các thư mục sau:

Thư mục	Vai trò
/arch	Chứa mã nguồn giúp Linux kernel có thể thực thi được trên nhiều kiến trúc CPU khác nhau như x86, alpha, arm, mips, mk68, powerpc, sparc,...
/block	Chứa mã nguồn triển khai nhiệm vụ lập lịch cho các thiết bị lưu trữ.
/drivers	Chứa mã nguồn để triển khai nhiệm vụ điều khiển, giám sát, trao đổi dữ liệu với các thiết bị.
/fs	Chứa mã nguồn triển khai nhiệm vụ quản lý dữ liệu trên các thiết bị lưu trữ.
/ipc	Chứa mã nguồn triển khai nhiệm vụ giao tiếp giữa các tiến trình
/kernel	Chứa mã nguồn triển khai nhiệm vụ lập lịch và đồng bộ hoạt động của các tiến trình.
/mm	Chứa mã nguồn triển khai nhiệm vụ quản lý bộ nhớ
/net	Chứa mã nguồn triển khai nhiệm vụ xử lý các gói tin theo mô hình TCP/IP.

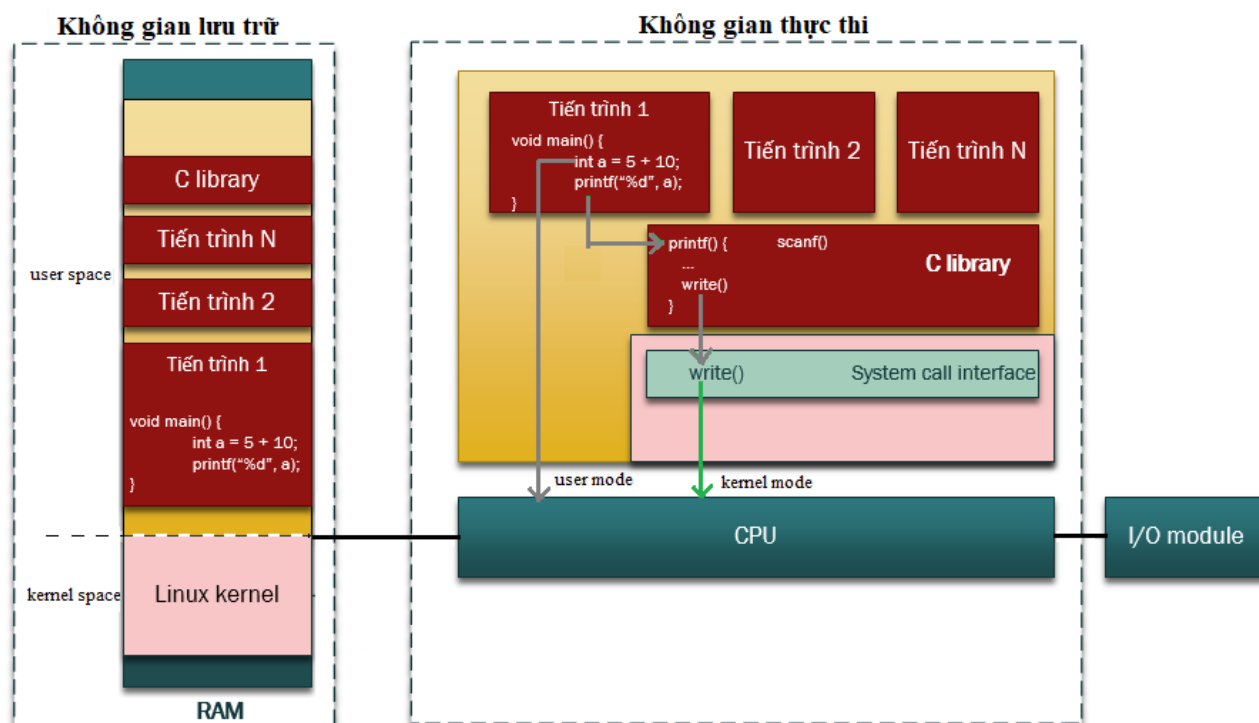
## Các khái niệm thường dùng.

Trong phần này, chúng ta sẽ tìm hiểu về một số khái niệm rất hay được sử dụng:

- User space và kernel space.
- User mode và kernel mode.
- System call và ngắt.
- Process context và interrupt context.

Bộ nhớ RAM chứa các lệnh/dữ liệu dạng nhị phân của Linux kernel và các tiến trình. RAM được chia làm 2 miền (hình 4):

- **Kernel space** là vùng không gian chứa các lệnh và dữ liệu của kernel.
- **User space** là vùng không gian chứa các lệnh và dữ liệu của các tiến trình.

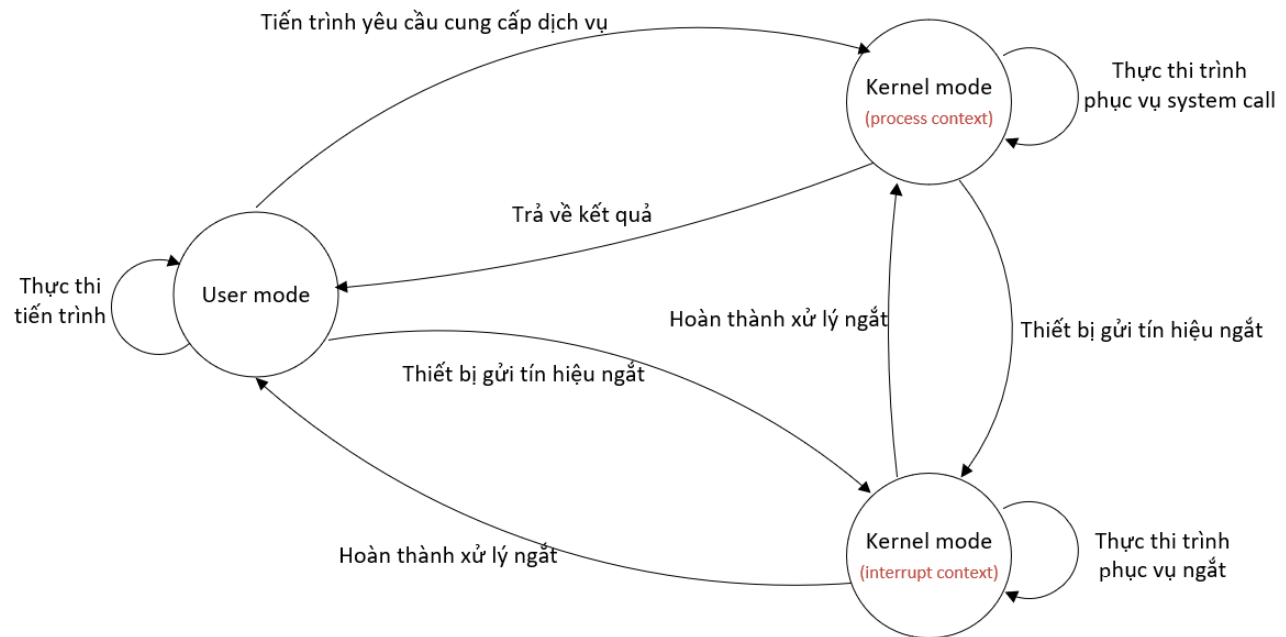


Hình 4 Kiến trúc Linux kernel đứng ở góc độ thực thi của CPU (execution point of view)

CPU có 2 chế độ thực thi (hình 4):

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

Để hiểu rõ hơn, ta xét tiến trình 1 trong hình 4. Tiến trình này gồm nhiều lệnh nhị phân, tương ứng với 2 lệnh C. CPU sẽ lần lượt lấy các lệnh này ra và thực thi. Lệnh thứ nhất, "a = 5 + 10", là một lệnh tính toán, sẽ được CPU thực thi ở chế độ user mode. Lệnh thứ hai, "printf("%d", a)", là một lệnh vào/ra. Hàm "printf" sẽ gọi system call "write" để yêu cầu Linux kernel in thông tin ra màn hình. Khi đó, CPU sẽ chuyển sang chế độ kernel mode để thực thi các lệnh của Linux kernel.



Hình 5 Các chế độ hoạt động của CPU

Khi một tiến trình cần sử dụng một dịch vụ nào đó của kernel, tiến trình sẽ gọi một **system call**. System call cũng tương tự như các hàm bình thường khác (library call). Chỉ có điều, các library call được cung cấp bởi các thư viện trong user space, còn các system call được cung cấp bởi kernel. Do đó, khi tiến trình gọi các library call, CPU vẫn giữ nguyên chế độ thực thi user mode. Còn khi tiến trình gọi các system call, CPU phải chuyển sang chế độ kernel mode để thực thi các lệnh của kernel (hình 5). Lúc này, ta nói rằng, CPU đang thực thi ở chế độ kernel mode, trong ngữ cảnh **process context**. Sau khi kernel thực hiện xong yêu cầu, kernel gửi trả kết quả cho tiến trình. Lúc này, CPU lại chuyển sang chế độ user mode để thực thi tiếp các lệnh của tiến trình.

Ngoài system call, ngắt cũng là một nguyên nhân khiến CPU chuyển chế độ thực thi sang kernel mode (hình 5). Khi có một thiết bị muốn trao đổi dữ liệu với CPU, nó sẽ gửi một tín hiệu ngắt tới CPU bằng cách nâng điện áp trên chân INT của CPU. Khi đó, CPU sẽ ngừng thực thi các lệnh của tiến trình lại, chuyển sang chế độ kernel mode rồi thực thi một chương trình đặc biệt của kernel để xử lý tín hiệu ngắt đó. Lúc này, ta nói CPU đang thực thi ở chế độ kernel mode, trong ngữ cảnh **interrupt context**. Sau khi xử lý xong, CPU trở lại chế độ user mode và tiếp tục thực hiện các lệnh tiếp theo của tiến trình.

## Phần 2. Giới Thiệu Linux Kernel Module

Linux được thiết kế để làm việc với hàng tỉ thiết bị. Nhưng ta không thể đưa tất cả các driver vào trong kernel được, vì sẽ làm cho kích thước kernel rất lớn. Giải pháp cho vấn đề này đó là: thiết kế các driver dưới dạng module tách rời với kernel. Trong quá trình hoạt động, driver nào cần thiết sẽ được lắp vào kernel, còn driver nào không cần thiết sẽ bị tháo ra khỏi kernel (dynamic loading)

Linux kernel module là một file với tên mở rộng là (.ko). Nó sẽ được lắp vào hoặc tháo ra khỏi kernel khi cần thiết. Chính vì vậy, nó còn có một tên gọi khác là loadable kernel module. Một trong những kiểu loadable kernel module phổ biến đó là driver. Việc thiết kế driver theo kiểu loadable module mang lại 3 lợi ích:

- Giúp giảm kích thước kernel. Do đó, giảm sự lãng phí bộ nhớ và giảm thời gian khởi động hệ thống.
- Không phải biên dịch lại kernel khi thêm mới driver hoặc khi thay đổi driver.
- Không cần phải khởi động lại hệ thống khi thêm mới driver. Trong khi đối với Windows, mỗi khi cài thêm driver, ta phải khởi động lại hệ thống, điều này không thích hợp với các máy server.

Phần lớn các driver đều là các loadable kernel module, nhưng không phải là tất cả. Vẫn có một số driver được tích hợp luôn vào trong kernel, đặc biệt là các bus driver. Chúng được gọi là built-in driver. Các device driver thường sẽ là các loadable kernel module.

Ngược lại, không phải loadable kernel module nào cũng là driver, ví dụ kvm.ko là loadable kernel module nhưng không phải là driver. Trên thực tế, loadable kernel module được chia làm 3 loại chính: device driver, system call và file system.

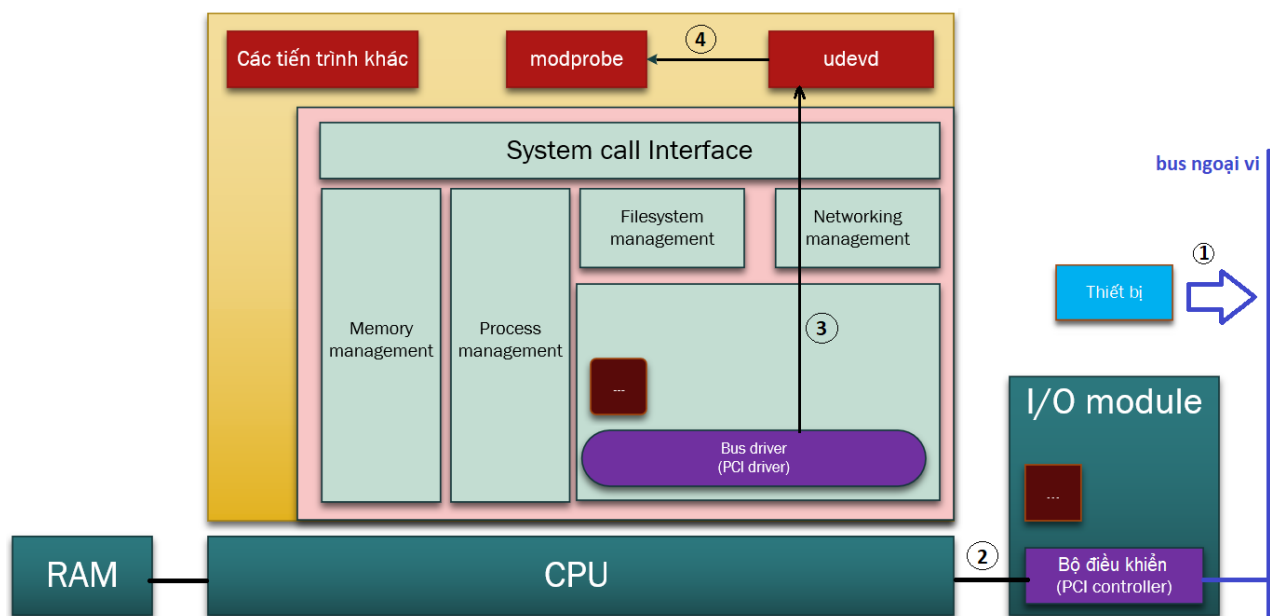
Trong khóa học này, các thuật ngữ module, device driver, loadable kernel module, Linux kernel module, loadable module, kernel module đều được hiểu là một.

Khi cần một module nhưng nó lại chưa có trong kernel space, kernel sẽ đưa module ấy vào. Quá trình này có thể diễn ra một cách tự động, với trình tự sau:

- Bước 1: Kernel kích hoạt tiến trình **modprobe** cùng với tham số truyền vào là tên của module (ví dụ xxx.ko).
- Bước 2: Tiến trình modprobe kiểm tra file `/lib/modules/<kernel-version>/modules.dep` xem xxx.ko có phụ thuộc vào module nào khác không. Giả sử xxx.ko phụ thuộc vào module yyy.ko.
- Bước 3: Tiến trình modprobe sẽ kích hoạt tiến trình **insmod** để đưa các module phụ thuộc vào trước (yyy.ko), rồi mới tới module cần thiết (xxx.ko).

Như vậy, các module được đưa vào kernel space dưới sự giúp đỡ của tiến trình modprobe. Câu hỏi đặt ra là: kernel kích hoạt tiến trình modprobe bằng cách nào?

- Cách 1 là sử dụng **kmod**. Đây là một thành phần của Linux kernel, hoạt động trong kernel space. Khi một thành phần nào đó của kernel cần đưa một module vào trong kernel space, nó sẽ truyền tên module cho hàm **request\_module** của kmod. Hàm **request\_module** sẽ gọi hàm **call\_usermodehelper\_setup** để sinh ra tiến trình **modprobe**. Các bạn có thể tham khảo mã nguồn của kmod tại [/kernel/kmod.c](https://kernel.org/pub/linux/kernel/v4.x/source/kmod.c).
- Cách 2 là sử dụng **udev** (hình 1). Đây là một tiến trình hoạt động trong use space. Nếu một thiết bị cắm vào hệ thống máy tính, thì điện trở trên bus ngoại vi (ví dụ PCI bus hoặc USB bus) sẽ thay đổi và bộ điều khiển (controller) sẽ biết điều này. Khi đó, bus driver sẽ gửi một bản tin lên cho tiến trình **udev**. Bản tin này chứa thông tin về thiết bị. Tiến trình **udev** sẽ tra cứu file `/lib/modules/<kernel-version>/modules.alias` để tìm ra driver nào tương thích với thiết bị. Sau đó, **udev** sinh ra tiến trình **modprobe**.



Hình 1 Minh họa quá trình kích hoạt modprobe bằng udev

## Phần 3. Cách viết một Linux Kernel Module

Bước 1: tạo thư mục cho bài học hôm nay

```
cd /home/ubuntu
mkdir ldd
cd ldd
mkdir phan_1
mkdir phan_1/bai_1_3
```

Bước 2: dùng lệnh "cd" di chuyển vào thư mục phan\_1/bai\_1\_3, rồi dùng lệnh "vim hello.c" để mở một file có tên hello.c. Sau đó, sao chép đoạn mã sau vào file hello.c

```
/*
 * hello.c - vi du ve linux kernel module
 */

#include <linux/module.h> /* thu vien nay dinh nghia cac macro nhu module_init
va module_exit */

#define DRIVER_AUTHOR "Nguyen Tien Dat <dat.a3cbq91@gmail.com>"
#define DRIVER_DESC "A sample loadable kernel module"

static int __init init_hello(void)
{
    printk("Hello Vietnam\n");
    return 0;
}

static void __exit exit_hello(void)
{
    printk("Goodbye Vietnam\n");
}

module_init(init_hello);
module_exit(exit_hello);

MODULE_LICENSE("GPL"); /* giay phep su dung cua module */
MODULE_AUTHOR(DRIVER_AUTHOR); /* tac gia cua module */
MODULE_DESCRIPTION(DRIVER_DESC); /* mo ta chuc nang cua module */
MODULE_SUPPORTED_DEVICE("testdevice"); /* kieu device ma module ho tro */
```

Do module hello cần dùng một số hàm hoặc macro của Linux kernel, nên chúng ta sẽ sử dụng từ khóa **#include** để chỉ rõ các file cần dùng. Điều này cũng tương tự như khi viết các chương trình ứng dụng trên user space, chúng ta cũng dùng **#include** cùng với tên của các thư viện. Do đó, có thể nói rằng, Linux kernel chính là một thư viện, cung cấp các hàm, các macro để chúng ta phát triển kernel module.

Trong ví dụ trên, ta chỉ cần tham chiếu tới file của Linux kernel là **<linux/module.h>**. File này chứa 2 macro quan trọng, là **module\_init()** và **module\_exit()**. Do đó, dù viết bất cứ kernel module nào, ta cũng cần tham chiếu tới **<linux/module.h>**.

- `module_init` giúp xác định hàm nào sẽ được thực thi ngay sau khi lắp module vào kernel.
- `module_exit` giúp xác định hàm nào được thực thi ngay trước khi tháo module ra khỏi kernel.

Trong ví dụ trên, `init_hello()` là hàm được gọi ngay sau khi module hello được lắp vào, và `exit_hello()` là hàm được gọi ngay trước khi module hello bị tháo ra khỏi kernel.

Macro `__init` thường đi kèm với hàm khởi tạo. Trong ví dụ trên, macro `__init` xuất hiện trước tên hàm `init_hello`. Macro này giúp kernel biết rằng, hàm `init_hello()` chỉ phải thực thi lúc khởi tạo, nên vùng nhớ chứa hàm này có thể được giải phóng sau khi nó thực thi xong mà không ảnh hưởng gì.

Tương tự, macro `__exit` thường đi kèm với hàm kết thúc. Trong ví dụ trên, `__exit` xuất hiện trước tên hàm `exit_hello`. Macro này cho kernel biết, khi lắp module vào kernel thì chưa cần đưa hàm `exit_hello` vào trong bộ nhớ RAM. Chỉ khi chuẩn bị tháo module ra khỏi kernel, hàm `exit_hello` này mới cần được đưa vào RAM và thực thi.

Trong quá trình viết kernel module, các lập trình viên thường sử dụng hàm [printk\(\)](#) để ghi lại quá trình hoạt động của module. Việc này được gọi là logging. Mục đích của việc logging là để phục vụ quá trình gỡ lỗi sau này (debug). Ta có thể sử dụng lệnh **dmesg** để xem quá trình hoạt động của kernel kể từ lúc nó khởi động. Chúng ta sẽ tìm hiểu kỹ hơn về hàm `printk` trong bài 3\_1.

Các macro nằm ở cuối ví dụ trên cung cấp các thông tin về module. Ta có thể sử dụng lệnh **modinfo** để xem các thông tin của một module:

- Macro `MODULE_AUTHOR` cho biết ai là người tạo ra module.
- Macro `MODULE_DESCRIPTION()` cho biết module làm được những gì.
- Macro `MODULE_SUPPORTED_DEVICE()` cho biết module này hỗ trợ làm việc với những thiết bị nào.
- `MODULE_LICENSE` cho biết người dùng có cần phải trả phí nếu sử dụng module hay không. Trong ví dụ trên, giấy phép sử dụng module thuộc loại GPL. Với giấy phép sử dụng GPL, người dùng có thể sử dụng module miễn phí. Ngoài GPL, còn có các loại license như GPL v2, BSD/GPL, MIT/GPL, MPL/GPL.

## Biên dịch kernel module

Để biên dịch kernel module, ta sử dụng phương pháp Kbuild. Theo phương pháp này, chúng ta cần tạo ra 2 file: một file có tên là **Makefile**, file còn lại có tên là **Kbuild**. Đầu tiên, ta sẽ tạo ra Makefile.

```
#cd /home/ubuntu/ldd/phan_1/bai_1_3
#vim Makefile

KDIR = /lib/modules/`uname -r`/build
```



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

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

Trong Makefile trên:

- Thẻ **all** chứa câu lệnh để biên dịch các module trong thư mục hiện tại.
- Thẻ **clean** chứa lệnh xóa tất cả các object file có trong thư mục hiện tại.

Tiếp theo, ta tạo ra file Kbuild nằm trong cùng thư mục với Makefile:

```
#cd /home/ubuntu/ldd/phan_1/bai_1_3
#vim Kbuild

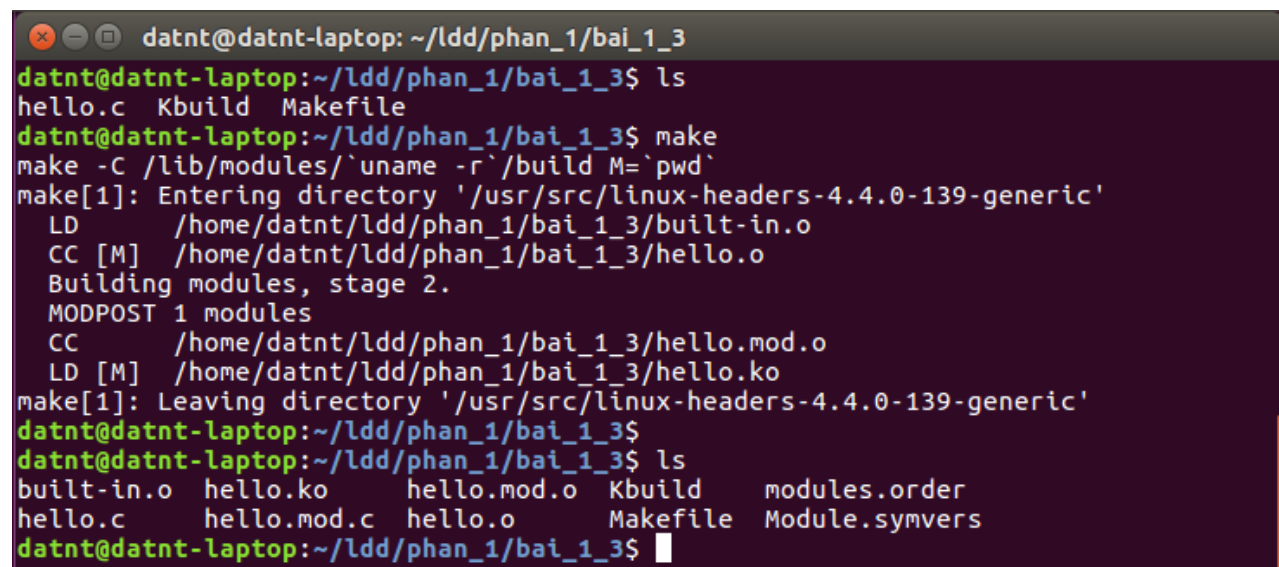
EXTRA_CFLAGS = -Wall

obj-m      = hello.o
```

Trong file Kbuild trên:

- Biến **obj-m** chỉ ra rằng: object file sẽ được biên dịch theo kiểu kernel module.
- Cờ **-Wall** cho phép trình biên dịch hiển thị tất cả các bản tin cảnh báo trong quá trình biên dịch.

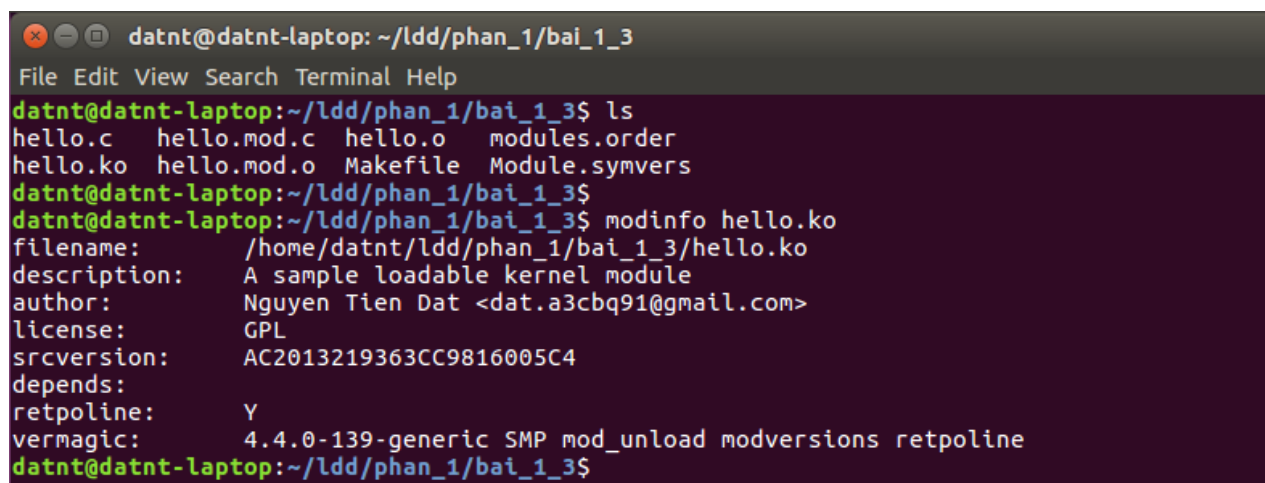
Để tạo ra kernel module, ta gõ lệnh **make** hoặc **make all** (hình 2). Khi ta gõ lệnh "make", tiến trình **make** sẽ dựa vào Makefile và Kbuild để biên dịch mã nguồn, tạo ra kernel module.



```
datnt@datnt-laptop: ~/ldd/phan_1/bai_1_3
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ ls
hello.c Kbuild Makefile
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ make
make -C /lib/modules/`uname -r`/build M=`pwd`
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-139-generic'
LD      /home/datnt/ldd/phan_1/bai_1_3/built-in.o
CC [M]  /home/datnt/ldd/phan_1/bai_1_3/hello.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/datnt/ldd/phan_1/bai_1_3/hello.mod.o
LD [M]  /home/datnt/ldd/phan_1/bai_1_3/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-139-generic'
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ ls
built-in.o hello.ko hello.mod.o Kbuild modules.order
hello.c hello.mod.c hello.o Makefile Module.symvers
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$
```

Hình 2. sử dụng công cụ make để biên dịch kernel module

Sau khi biên dịch xong, ta sẽ thấy xuất hiện một file có tên mở rộng là **.ko** (ko là viết tắt của kernel object). Đây chính là kernel module. Để biết được các thông tin về module, ta sử dụng lệnh **modinfo** (hình 3).

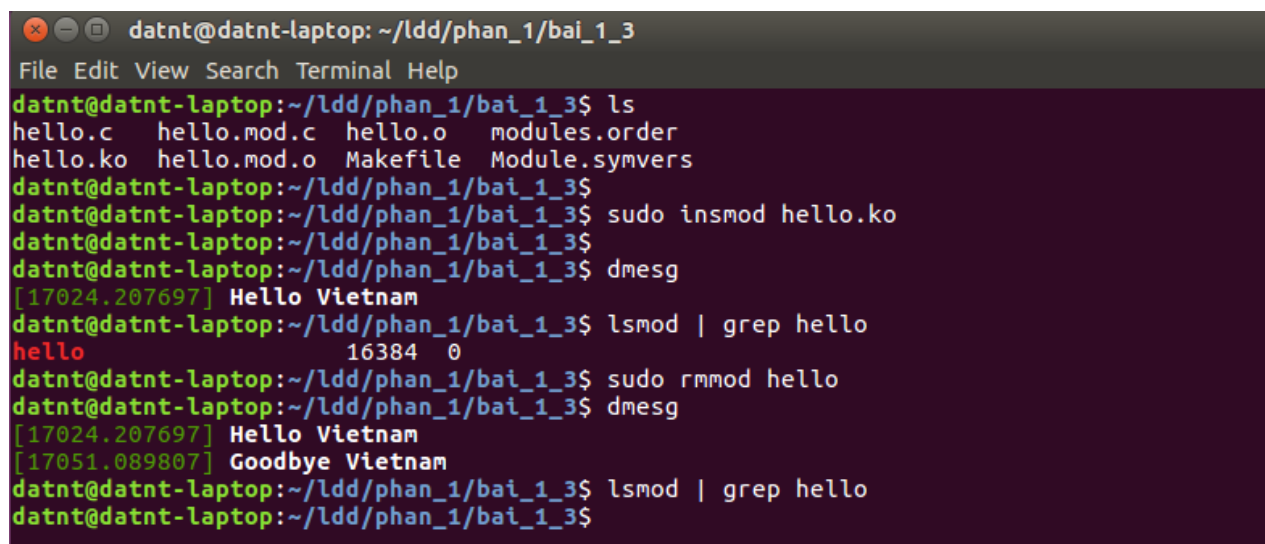


```
datnt@datnt-laptop: ~/ldd/phan_1/bai_1_3
File Edit View Search Terminal Help
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ ls
hello.c  hello.mod.c  hello.o  modules.order
hello.ko  hello.mod.o  Makefile  Module.symvers
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ modinfo hello.ko
filename:       /home/datnt/ldd/phan_1/bai_1_3/hello.ko
description:    A sample loadable kernel module
author:         Nguyen Tien Dat <dat.a3cbq91@gmail.com>
license:        GPL
srcversion:     AC2013219363CC9816005C4
depends:
retpoline:      Y
vermagic:       4.4.0-139-generic SMP mod_unload modversions retpoline
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$
```

Hình 3. sử dụng công cụ modinfo để biết thông tin về module

## Lắp/tháo kernel module

Để lắp module vào trong kernel, ta có thể thực hiện thủ công bằng cách gõ lệnh **insmod**. Sau khi lắp xong, ta sẽ dùng lệnh **lsmod** để kiểm tra xem module đã được load thành công chưa. Tiếp theo, ta sẽ dùng lệnh **dmesg** để theo dõi quá trình hoạt động của module (hình 4). Cuối cùng, chúng ta sẽ dùng lệnh **rmmod** để tháo module ra khỏi kernel.



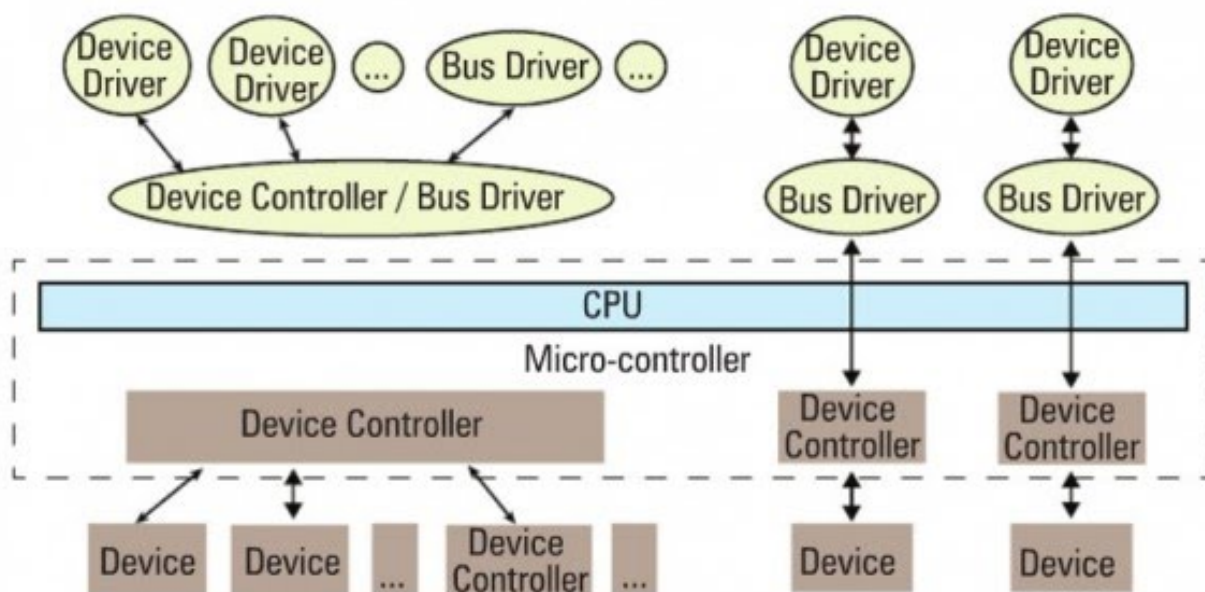
```
datnt@datnt-laptop: ~/ldd/phan_1/bai_1_3
File Edit View Search Terminal Help
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ ls
hello.c  hello.mod.c  hello.o  modules.order
hello.ko  hello.mod.o  Makefile  Module.symvers
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ sudo insmod hello.ko
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ dmesg
[17024.207697] Hello Vietnam
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ lsmod | grep hello
hello                16384  0
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ sudo rmmod hello
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ dmesg
[17024.207697] Hello Vietnam
[17051.089807] Goodbye Vietnam
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$ lsmod | grep hello
datnt@datnt-laptop:~/ldd/phan_1/bai_1_3$
```

Hình 4. Đưa module vào/ra khỏi kernel

## Phần 4. Cơ bản về driver trong Linux

### Vai trò của driver

Driver là một trình điều khiển có vai trò điều khiển, quản lý, giám sát một thực thể nào đó dưới quyền của nó. Bus driver làm việc với một đường bus, device driver làm việc với một thiết bị (chuột, bàn phím, màn hình, đĩa cứng, camera, ...). Có thể lấy ví dụ tương tự như vai trò của một phi công hoặc một hệ thống bay tự động được giám sát bởi phi công, một thành phần phần cứng có thể được điều khiển bởi một driver hoặc được điều khiển bởi một phần cứng khác mà được quản lý bởi một driver. Trường hợp này, phần cứng có vai trò điều khiển được gọi là một device controller. Bản thân các controller cũng cần driver. Ví dụ: hard disk controller, display controller, audio controller, ... quản lý các thiết bị kết nối với chúng, mà nói một cách kỹ thuật hơn đó là các IDE controller, PCI controller, USB controller, SPI controller, I2C controller, ... Các khái niệm này được minh họa tổng quan như hình sau:



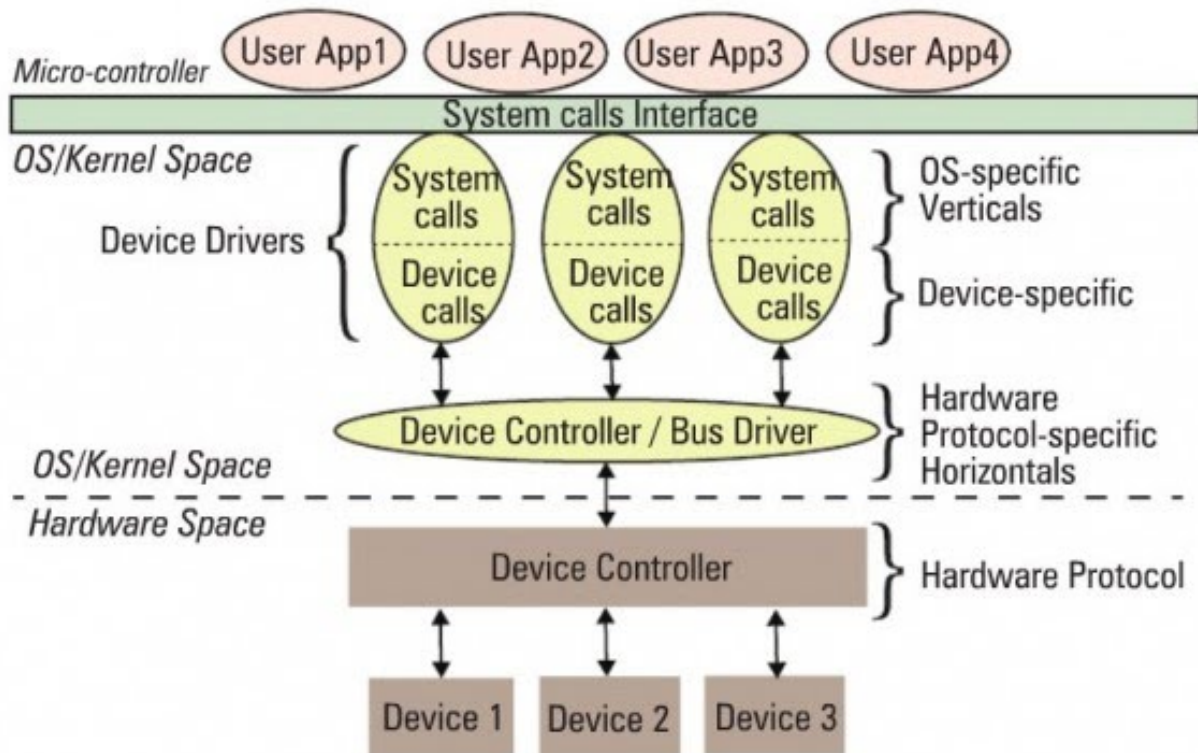
Hình 1. Tương tác giữa thiết bị và driver

Các device controller thông thường được kết nối với CPU thông qua đường bus (PCI, IDE, USB, SPI, ...). Trong vi điều khiển, CPU và các device controller thường được thiết kế trên một chip. Điều này cho phép giảm kích thước và giá thành, phù hợp với phát triển hệ thống nhúng. Mà về mặt nguyên tắc, sẽ không có gì khác biệt đối lớn đối với các driver trên các hệ thống máy tính cá nhân.

### Hai nhiệm vụ của driver

Các bus driver cung cấp giao diện đặc tả cho các giao thức phần cứng tương ứng. Nó nằm ở tầng dưới cùng trong mô hình phân lớp phần mềm của hệ điều hành. Nằm trên nó là các device driver thực sự để vận hành các thiết bị, mang đặc trưng của từng thiết bị xác định. Ngoài ra, mục đích quan trọng

của các driver thiết bị là cung cấp một giao diện trừu tượng hóa cho người sử dụng, tức là cung cấp một giao diện lên tầng trên của hệ điều hành. Một cách tổng quan, một driver sẽ bao gồm 2 phần quan trọng: a) giao tiếp với thiết bị (Device-specific) b) giao tiếp với hệ điều hành (OS-specific)



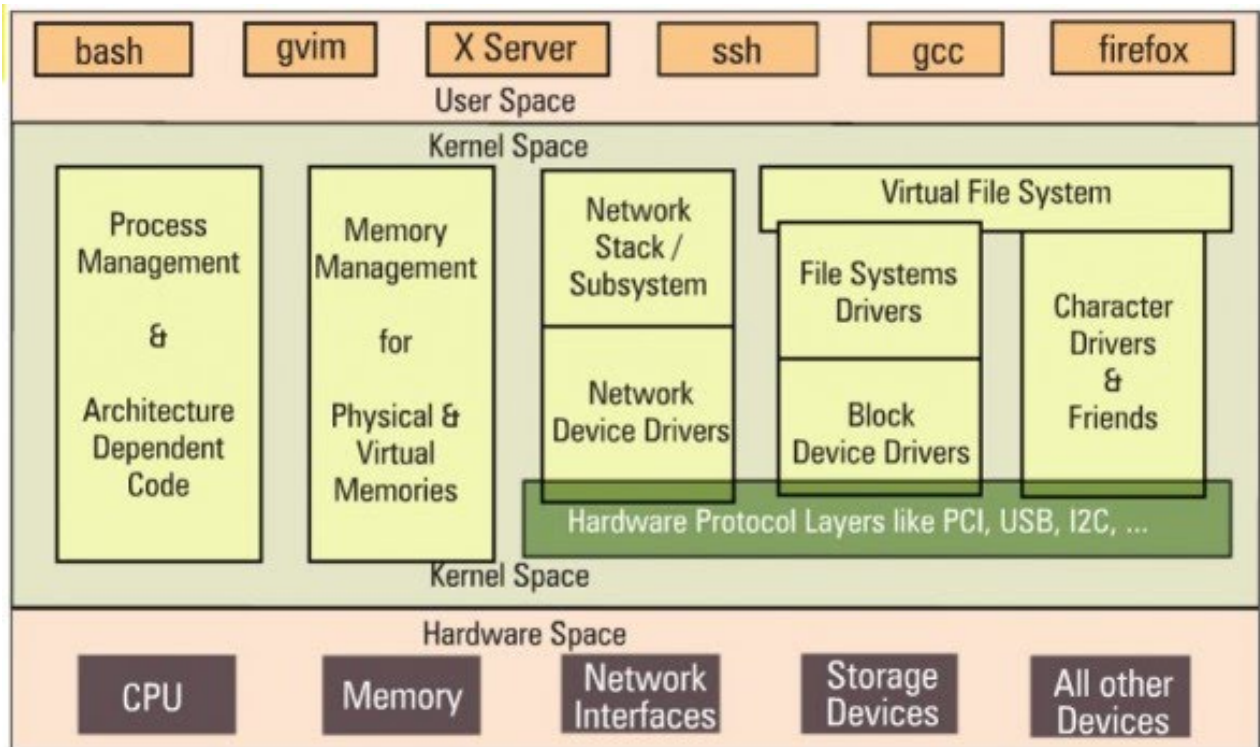
Hình 2. Các thành phần của driver trên Linux

Thành phần giao tiếp với thiết bị (device-specific) của một driver là giống nhau đối với tất cả các hệ điều hành. Nó có thể hiểu và giải mã các thông tin về thiết bị (chi tiết kỹ thuật, kiểu thao tác, hiệu năng, cách lập trình giao tiếp với thiết bị, ...)

Thành phần giao tiếp với hệ điều hành (OS-specific) gắn kết chặt chẽ với các cơ chế của hệ điều hành, và do vậy sẽ là khác nhau giữa một driver trên Linux và một driver trên Windows, hoặc MacOS, ...

## Mô hình phân lớp theo chiều dọc

Trên Linux, device driver cung cấp một giao diện “system call” (giao diện gọi các hàm hệ thống) đến tầng ứng dụng cho người dùng; đây được coi là một ranh giới giữa tầng nhân (kernel space) và tầng người dùng (user space) của Linux. Mô hình phân tầng được chỉ ra như hình vẽ.



Hình 3. Kiến trúc tổng quan nhân Linux

Tùy thuộc vào đặc trưng của của driver với hệ điều hành, driver trên Linux được phân chia thành 3 loại (phân cấp theo chiều dọc):

- Packet-oriented or the network vertical (driver hướng gói dữ liệu)
- Block-oriented or the storage vertical (driver hướng khối dữ liệu)
- Byte-oriented or the character vertical (driver hướng byte/ký tự)

Packet-oriented hay network driver gồm 2 phần: a) network protocol stack và b) network interface card (NIC) device drivers, hoặc đơn giản là network device driver (có thể là Ethernet, Wi-Fi, hoặc bất kỳ các giao tiếp mạng nào khác,...)

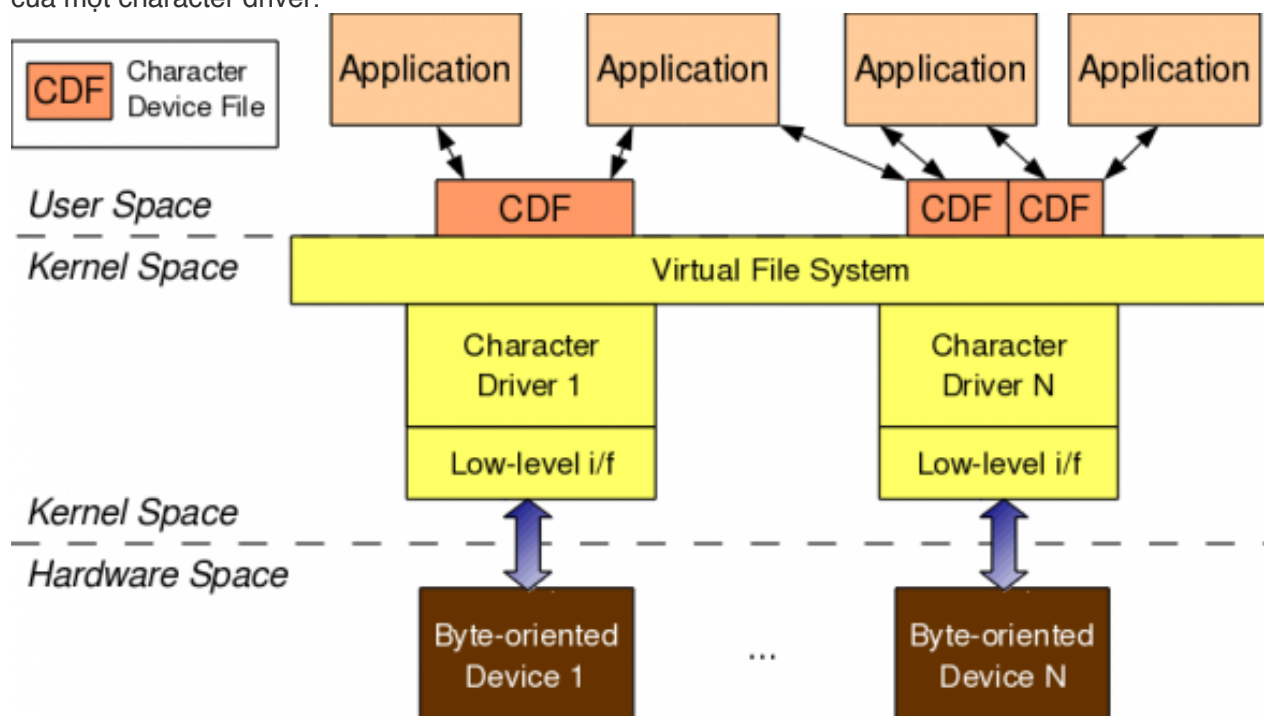
Block-oriented hay storage driver gồm 2 phần: a) File-system drivers để giải mã các định dạng khác nhau trên các phân vùng lưu trữ khác nhau (FAT, ext, ...) và b) Block device drivers cho các giao thức phần cứng ứng với các thiết bị lưu trữ khác nhau (IDE, SCSI, MTD, ...)

Các Byte-oriented hay character driver lại tiếp tục được phân chia thành các lớp con (sub-classified) như tty driver, input driver, console driver, frame-buffer drivers, sound driver, ... (tương ứng với các giao tiếp như RS232, PS/2, VGA, I2C, SPI, ...)



## Phần 5. Character Device Driver

Hầu hết các thiết bị đều thuộc kiểu thiết bị hướng byte (byte-oriented), do vậy hầu hết các device driver là các character device drivers. Ví dụ: serial drivers, audio drivers, video drivers, camera drivers, và các I/O drivers. Trong thực tế, các device driver của cả storage và network device drivers cũng là các dạng của một character driver.



Hình 1. Tổng quan về character driver trên Linux

Như minh họa trong hình vẽ trên, 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:

1. Application (ứng dụng)
2. Character device file (File thiết bị)
3. Character device driver (Driver thiết bị)
4. Character device (Thiết bị)

Một điểm thú vị là các thực thể trên có thể tồn tại một cách độc lập trên 1 hệ thống mà không cần sự tham gia của các thực thể khác. Việc tồn tại của mỗi thực thể không có nghĩa là chúng đã được liên kết đến các thực thể khác để tạo ra kết nối hoàn chỉnh. Các kết nối giữa chúng cần được thực hiện một cách tường minh. Một ứng dụng kết nối đến file thiết bị bằng cách gọi một hàm mở file thiết bị đó. Các file thiết bị liên kết đến driver của nó bằng thao tác đăng ký được thực hiện trong driver. Một driver được kết nối đến một thiết bị thông qua các thao tác mức thấp với thiết bị phần cứng đặc trưng. Như vậy, chúng ta đã tạo ra một kết nối hoàn chỉnh từ ứng dụng đến thiết bị phần cứng. Lưu ý rằng, file thiết bị không phải là thiết bị thực sự, nó chỉ là một thực thể nắm giữ thiết bị thực sự.

## Số hiệu file thiết bị (Major và minor number)

Việc kết nối giữa ứng dụng và file thiết bị được thực hiện thông qua tên file thiết bị. Tuy nhiên, kết nối giữa file thiết bị và device driver được thực hiện dựa trên số hiệu của file thiết bị chứ không thông qua tên file. Điều này cho phép các ứng dụng ở không gian người dùng có thể sử dụng bất kỳ tên file thiết bị nào và cho phép không gian nhân có thể sử dụng các kết nối thông qua số hiệu thông thường giữa file thiết bị và driver. Các số hiệu file thiết bị này thường được biết đến như là một cặp số <major, minor> hoặc số hiệu major, minor của file thiết bị.

Trong các phiên bản Linux cũ (trước 2.4), một số hiệu major được sử dụng cho một driver và số hiệu minor dùng để biểu diễn cho các chức năng con của một driver. Từ phiên bản 2.6, sự phân biệt này không còn quan trọng nữa, nhiều driver có thể sử dụng cùng số hiệu major nhưng số hiệu minor khác nhau.

Có thể sử dụng câu lệnh như sau để liệt kê các file thiết bị kiểu character khác nhau trên hệ thống:

```
$ ls -l /dev/ | grep "^c"
```

## Các hỗ trợ liên quan đến <major, minor> trong kernel 2.6

Sử dụng kiểu dữ liệu (định nghĩa trong `linux/types.h`):

- `dev_t` chứa cả số hiệu major và minor

Sử dụng các Macros (định nghĩa trong `linux/kdev_t.h`):

- `MAJOR(dev_t dev)` lấy số hiệu major từ tham số `dev`
- `MINOR(dev_t dev)` lấy số hiệu minor từ tham số `dev`
- `MKDEV(int major, int minor)` tạo ra dữ liệu `dev` từ cặp số hiệu major và minor.

Việc kết nối giữa file thiết bị và driver được thực hiện thông qua 2 bước sau:

1. Đăng ký số hiệu <major, minor> cho file thiết bị

2. Kết nối các thao tác file thiết bị với các hàm tương ứng trong driver.

Bước đầu tiên được thực hiện bằng cách sử dụng một trong 2 hàm API sau: (được định nghĩa trong `linux/fs.h`):

```
+ int register_chrdev_region(dev_t first, unsigned int cnt, char *name);

+ int alloc_chrdev_region(dev_t *first, unsigned int firstminor, unsigned int cnt,
char *name);
```

Trong cả 2 trường hợp, xem nội dung file `/proc/devices` sẽ liệt kê tên và các số hiệu major đã được đăng ký.

Mã nguồn cho driver với các thao tác thực hiện đăng ký số hiệu và tên module như sau:

```
#include <linux/types.h>

#include <linux/kdev_t.h>

#include <linux/fs.h>

static dev_t first; // Global variable for the first device number
```

Trong hàm tạo:

```
if (alloc_chrdev_region(&first, 0, 3, "Shweta") < 0)

{

    return -1;

}

printk(KERN_INFO "<Major, Minor>: <%d, %d>\n", MAJOR(first), MINOR(first));
```

Trong hàm hủy:

```
unregister_chrdev_region(first, 3);
```

Mã nguồn hoàn chỉnh:

```
#include <linux/module.h>

#include <linux/version.h>

#include <linux/kernel.h>

#include <linux/types.h>

#include <linux/kdev_t.h>
```



```

#include <linux/fs.h>

static dev_t first; // Global variable for the first device number

static int __init ofcd_init(void) /* Constructor */
{
    printk(KERN_INFO "Namaskar: ofcd registered");

    if (alloc_chrdev_region(&first, 0, 3, "Shweta") < 0)
    {
        return -1;
    }

    printk(KERN_INFO "<Major, Minor>: <%d, %d>\n", MAJOR(first), MINOR(first));

    return 0;
}

static void __exit ofcd_exit(void) /* Destructor */
{
    unregister_chrdev_region(first, 3);

    printk(KERN_INFO "Alvida: ofcd unregistered");
}

module_init(ofcd_init);

module_exit(ofcd_exit);

MODULE_LICENSE("GPL");

MODULE_AUTHOR("Anil Kumar Pugalia");

MODULE_DESCRIPTION("Our First Character Driver");

```

Biên dịch mã nguồn driver trên và nạp vào hệ thống:

- Build bằng Makefile để tạo ra file driver (.ko)
- Nạp vào hệ thống sử dụng lệnh insmod

- Liệt kê các module đã nạp sử dụng lệnh `lsmod`
- Gỡ module sử dụng `rmmmod`

Sử dụng lệnh `cat /proc/devices` cho phép mở nội dung xem các thông tin về số hiệu `major` và tên module đã đăng ký.

Tuy nhiên, đến đây vẫn chưa có tên file thiết bị được tạo ra dưới thư mục `/dev` của hệ thống, chúng ta sẽ tạo bằng tay, sử dụng lệnh `mknod`, thao tác như minh họa sau:

```
sudo mknod /dev/ofcd0 c 250 0
sudo mknod /dev/ofcd1 c 250 1
sudo mknod /dev/ofcd2 c 250 2
```

```
[anil@shrishti CharDriver]$ cat /proc/devices | head -28 | tail -10
136 pts
171 ieee1394
180 usb
189 usb_device
226 drm
250 Shweta
251 heci
252 hidraw
253 bsg
254 rtc
[anil@shrishti CharDriver]$ ls -l /dev | grep "250,"
[anil@shrishti CharDriver]$
[anil@shrishti CharDriver]$ sudo mknod /dev/ofcd0 c 250 0
[anil@shrishti CharDriver]$ sudo mknod /dev/ofcd1 c 250 1
[anil@shrishti CharDriver]$ sudo mknod /dev/ofcd2 c 250 2
[anil@shrishti CharDriver]$ sudo chmod a+w /dev/ofcd*
[anil@shrishti CharDriver]$ ls -l /dev/ofcd*
crw-rw-rw- 1 root root 250, 0 2011-01-05 18:45 /dev/ofcd0
crw-rw-rw- 1 root root 250, 1 2011-01-05 18:45 /dev/ofcd1
crw-rw-rw- 1 root root 250, 2 2011-01-05 18:45 /dev/ofcd2
[anil@shrishti CharDriver]$ cat /dev/ofcd0
cat: /dev/ofcd0: No such device or address
[anil@shrishti CharDriver]$ echo Hi > /dev/ofcd0
bash: /dev/ofcd0: No such device or address
[anil@shrishti CharDriver]$
```

Hình 2. Thử nghiệm file thiết bị tạo ra từ driver

Lưu ý rằng số hiệu 250 là số hiệu mặc định có thể khác nhau tùy từng hệ thống, số hiệu này có thể nhận được từ kết quả trả về lấy từ hàm đăng ký trong driver (xem thông qua log file).

Trong phần trước, chúng ta đã đề cập đến việc đăng ký số hiệu file thiết bị <major, minor> trong driver, tuy nhiên, file thiết bị chưa được tự động tạo ra trong thư mục /dev của hệ thống mà cần thực hiện tạo bằng tay sử dụng lệnh mknod. Phần này sẽ tiếp tục đề cập đến việc tạo file thiết bị tự động sử dụng tiến trình udev của hệ thống và vấn đề quan trọng tiếp theo trong driver là kết nối các thao tác file thiết bị với các hàm driver cung cấp.

## Tạo file thiết bị tự động

Trong các phiên bản nhân Linux trước đây (trước 2.4), việc tạo các file thiết bị tự động được thực hiện bởi chính nhân hệ điều hành, bằng cách sử dụng các hàm API thích hợp trong devfs. Tuy nhiên, các nhà phát triển nhân hệ điều hành nhận thấy rằng file thiết bị liên quan nhiều hơn đến không gian người dùng và do đó nó nên được giải quyết trên không gian người dùng thay vì làm ở tầng nhân. Xuất phát từ ý tưởng này, tầng nhân sẽ chỉ đưa ra lớp thiết bị thích hợp (device class) và thông tin thiết bị trong thư mục /sys trong đó thiết bị sẽ được xem xét. Sau đó, không gian người dùng cần thông dịch nó và đưa ra các hành động thích hợp. Trong hầu hết các hệ thống Linux, tiến trình udev được sử dụng để thu thập các thông tin và tạo ra file thiết bị. udev có thể cấu hình thông qua file cấu hình của nó để điều chỉnh tên file thiết bị, quyền truy xuất, kiểu của file thiết bị đó, ... Do vậy, đối với driver, các mục trong thư mục /sys cần được đưa vào sử dụng các hàm API liên quan đến mô hình thiết bị được khai báo trong <linux/device.h>. Các công việc còn lại sẽ được thực hiện bởi udev.

Lớp thiết bị (device class) được tạo ra như sau:

```
struct class *cl = class_create(THIS_MODULE, "<device class name>");
```

Sau đó, thông tin (<major, minor>) của thiết bị trong lớp này được tạo ra bởi hàm API

```
device_create(cl, NULL, first, NULL, "<device name format>", ...);
```

Trong đó, biến first có kiểu `dev_t` tương ứng với cặp giá trị <major, minor>. Khi hủy lớp thiết bị, thứ tự thực hiện là ngược lại:

```
device_destroy(cl, first);
```

```
class_destroy(cl);
```

Hình dưới minh họa các mục trong /sys được tạo ra sử dụng tên lớp thiết bị là `chardrv` và tên file thiết bị là `mynull`. (Sau khi driver được nạp vào hệ thống) Kết quả là file thiết bị được tạo ra bởi udev, dựa trên mục <major>:<minor> nằm trong file `dev`, trong thư mục `mynull`

```
[anil@shrishti CharDriver]$ ls /sys/class/
backlight/  firmware/  ieee1394_host/  pktcdvd/  sound/
bdi/        gpio/      ieee1394_node/  power_supply/  thermal/
block/      graphics/  ieee1394_protocol/  ppdev/  tty/
bsg/        hidraw/    input/          rtc/      vc/
chardrv/    hwmon/     mem/            scsi_device/  video4linux/
dma/        i2c-adapter/  misc/          scsi_disk/  video_output/
dmi/        ide_port/   net/           scsi_generic/  vtconsole/
drm/        ieee1394/   pci_bus/        scsi_host/

[anil@shrishti CharDriver]$ ls /sys/class/chardrv/
mynull@
[anil@shrishti CharDriver]$ ls -l /sys/class/chardrv/
total 0
lrwxrwxrwx 1 root root 0 2011-02-10 11:42 mynull -> ../../devices/virtual/chardrv/mynull/
[anil@shrishti CharDriver]$ ls -l /sys/class/chardrv/mynull/
total 0
-r--r--r-- 1 root root 4096 2011-02-10 11:41 dev
drwxr-xr-x 2 root root 0 2011-02-10 11:42 power/
lrwxrwxrwx 1 root root 0 2011-02-10 11:41 subsystem -> ../../../../class/chardrv/
-rw-r--r-- 1 root root 4096 2011-02-10 11:41 uevent
[anil@shrishti CharDriver]$ cat /sys/class/chardrv/mynull/dev
250:0
[anil@shrishti CharDriver]$ ls -l /dev/mynull
crw-rw---- 1 root root 250, 0 2011-02-10 11:41 /dev/mynull
[anil@shrishti CharDriver]$
```

Hình 1. Tạo file thiết bị tự động

Trong trường hợp có nhiều số hiệu minors, các hàm API `device_create()` và `device_destroy()` có thể đặt trong vòng lặp, và có thể sử dụng chuỗi `<device name format>`. Ví dụ minh họa:

```
device_create(cl, NULL, MKNOD(MAJOR(first), MINOR(first) + i), NULL,
"mynull%d", i);
```

## Các thao tác với file thiết bị (File operations)

Tất cả những lời gọi hệ thống (hay thao tác file) đối với một file thông thường đều có thể áp dụng cho file thiết bị. Đứng trên phương diện không gian người dùng, hầu hết mọi thứ đều là một file. Điều khác biệt nằm ở tầng nhân hệ điều hành, trong đó hệ thống file ảo (virtual file system – VFS) sẽ giải mã các loại file và diễn giải các thao tác file truyền đến driver thiết bị tương ứng đối với file thiết bị.

Để VFS truyền các thao tác file thiết bị (file operations) vào driver, nó phải được thông báo về các thao tác đó. Việc làm này chính là đăng ký các thao tác file với VFS được thực hiện trong mã nguồn driver. Quá trình này bao gồm 2 bước (chi tiết được thể hiện trong mã nguồn của “null driver” bên dưới).

Đầu tiên, cần tạo ra biến cấu trúc `struct file_operations pugs_fops` và điền vào cấu trúc này các thao tác xử lý muốn dùng đối với file thiết bị đang viết driver, các thao tác thông thường như

`my_open`, `my_close`, `my_read`, `my_write`, ... và khởi tạo một cấu trúc thiết bị kiểu character bằng cách khai báo biến cấu trúc `struct cdev c_dev` và gọi hàm `cdev_init()`.

Bước 2, điều khiển cấu trúc này đến hệ thống file ảo VFS bằng cách gọi hàm `cdev_add()`.

Các hàm `cdev_init()` và `cdev_add()` được khai báo trong `<linux/cdev.h>`. Để tiện quan sát, các hàm xử lý tương ứng với các thao tác file (`my_open`, `my_close`, `my_read`, `my_write`) cũng được viết mã, tuy nhiên chỉ xét đơn giản in ra thông báo mà chưa cần làm điều gì phức tạp (như là giao tiếp với thiết bị phần cứng thực sự).

Toàn bộ các mô tả trên được viết chi tiết trong mã nguồn một driver đơn giản gọi là “null driver” như sau:

```
#include <linux/module.h>

#include <linux/version.h>

#include <linux/kernel.h>

#include <linux/types.h>

#include <linux/kdev_t.h>

#include <linux/fs.h>

#include <linux/device.h>

#include <linux/cdev.h>

static dev_t first; // Global variable for the first device number

static struct cdev c_dev; // Global variable for the character device structure

static struct class *cl; // Global variable for the device class

static int my_open(struct inode *i, struct file *f)

{

    printk(KERN_INFO "Driver: open()\n");

    return 0;

}
```

```

static int my_close(struct inode *i, struct file *f)
{
    printk(KERN_INFO "Driver: close()\n");

    return 0;
}

static ssize_t my_read(struct file *f, char __user *buf, size_t
len, loff_t *off)
{
    printk(KERN_INFO "Driver: read()\n");

    return 0;
}

static ssize_t my_write(struct file *f, const char __user *buf,
size_t len, loff_t *off)
{
    printk(KERN_INFO "Driver: write()\n");

    return len;
}

static struct file_operations pugs_fops =
{
    .owner = THIS_MODULE,

    .open = my_open,

    .release = my_close,

    .read = my_read,

    .write = my_write
};

```

```

static int __init ofcd_init(void) /* Constructor */
{
    printk(KERN_INFO "Hello: ofcd registered");

    if (alloc_chrdev_region(&first, 0, 1, "Shweta") < 0)
    {
        return -1;
    }

    if ((cl = class_create(THIS_MODULE, "chardrv")) == NULL)
    {
        unregister_chrdev_region(first, 1);
        return -1;
    }

    if (device_create(cl, NULL, first, NULL, "mynull") == NULL)
    {
        class_destroy(cl);
        unregister_chrdev_region(first, 1);
        return -1;
    }

    cdev_init(&c_dev, &pugs_fops);

    if (cdev_add(&c_dev, first, 1) == -1)
    {
        device_destroy(cl, first);
        class_destroy(cl);
        unregister_chrdev_region(first, 1);
        return -1;
    }
}

```

```

    return 0;
}

static void __exit ofcd_exit(void) /* Destructor */
{
    cdev_del(&c_dev);

    device_destroy(cl, first);

    class_destroy(cl);

    unregister_chrdev_region(first, 1);

    printk(KERN_INFO "Alvida: ofcd unregistered");
}

module_init(ofcd_init);

module_exit(ofcd_exit);

MODULE_LICENSE("GPL");

MODULE_AUTHOR("Anil Kumar Pugalia");

MODULE_DESCRIPTION("Our First Character Driver");

```

Biên dịch driver trên và nạp vào nhân hệ thống theo các bước như sau:

1. Viết Makefile và chạy make để biên dịch driver tạo ra file `.ko`.
2. Nạp driver vào nhân hệ thống sử dụng lệnh `insmod`.
3. Liệt kê danh sách các module đã nạp sử dụng lệnh `lsmod`.
4. Xem thông tin số hiệu major và module đã nạp sử dụng lệnh `cat /proc/devices`.
5. Xem thông tin về “null driver” đã nạp bằng cách dùng lệnh `dmesg`
6. Gỡ bỏ driver sử dụng lệnh `rmmmod`



```
[anil@shrishti CharDriver]$ dmesg
Namaskar: ofcd registered
[anil@shrishti CharDriver]$ ls -l /dev/mynull
crw-rw---- 1 root root 250, 0 2011-02-10 14:54 /dev/mynull
[anil@shrishti CharDriver]$ su
शब्दकटः
[root@shrishti CharDriver]# echo "Hello Universe" > /dev/mynull
[root@shrishti CharDriver]# dmesg | tail -10

Namaskar: ofcd registered
Driver: open()
Driver: write()
Driver: close()
[root@shrishti CharDriver]# cat /dev/mynull
[root@shrishti CharDriver]# dmesg | tail -10

Namaskar: ofcd registered
Driver: open()
Driver: write()
Driver: close()
Driver: open()
Driver: read()
Driver: close()
[root@shrishti CharDriver]#
```

Hình 2. Thực nghiệm với null driver

Như vậy, đến đây chúng ta đã viết một driver và đăng ký tự động file thiết bị kết nối với nó để tăng ứng dụng có thể truy xuất như một file thông thường (có thể sử dụng lệnh echo, cat để ghi, đọc file thiết bị đó). Tuy nhiên, sẽ chưa có hiệu quả gì xảy ra khi sử dụng các thao tác đọc, ghi đối với file thiết bị này là bởi vì mặc dù đã có các hàm thao tác file thiết bị trong driver nhưng các hàm này chưa cung cấp thao tác xử lý dữ liệu nào cả. Phần tiếp theo, chúng ta sẽ tiếp tục phát triển các hàm này trong driver để cung cấp đầy đủ các thao tác cho file thiết bị tạo ra.

## Phần 6. Các thao tác với file device

### Thao tác đọc file device

Khi người sử dụng muốn đọc dữ liệu từ file device `/dev/mynull`, lời gọi hàm đọc (là một system call) sẽ truyền đến hệ thống file ảo VFS ở tầng nhân. VFS sẽ giải mã cặp số hiệu `<major, minor>` và tìm ra driver cần thiết để triệu gọi hàm `my_read()` của driver này (hàm đã được đăng ký với thao tác read của file thiết bị). Hàm này trả về giá trị là bao nhiêu byte dữ liệu nhận được đối với yêu cầu đọc từ người sử dụng file thiết bị. Trong ví dụ null driver, giá trị trả về là 0, điều đó có nghĩa là không có byte dữ liệu nào được cung cấp cho thao tác đọc trên tầng ứng dụng. Do đó, đến đây chúng ta sẽ tiếp tục cải tiến hàm này để cung cấp dữ liệu cho người sử dụng khi gọi thao tác đọc file thiết bị từ tầng ứng dụng. Quan sát các tham số của hàm `my_read()`

```
static ssize_t my_read(struct file *f, char __user *buf, size_t len, loff_t *off)
{
    printk(KERN_INFO "Driver: read()\n");

    return 0;
}
```

Từ các tham số của hàm này cho biết rằng dữ liệu nên được ghi vào bộ đệm buf để đưa lên tầng ứng dụng, tham số len cho biết kích thước (tính theo byte) của bộ đệm dữ liệu được yêu cầu bởi người dùng khi đọc. Thông thường, nên cung cấp số lượng ít hơn hoặc bằng len bytes dữ liệu vào bộ đệm buf, và số lượng byte đưa vào thật sự cũng được dùng làm giá trị trả về cho hàm.

Trong thao tác đọc, người lập trình driver sẽ điền dữ liệu lấy từ thiết bị thật vào bộ đệm dữ liệu được yêu cầu bởi người sử dụng để người sử dụng có thể nhận được chúng từ trên tầng ứng dụng.

### Thao tác ghi file device

```
static ssize_t my_write(struct file *f, const char __user *buf, size_t len, loff_t *off)
{
    printk(KERN_INFO "Driver: write()\n");

    return len;
}
```

Ngược lại với thao tác đọc là thao tác ghi dữ liệu vào file thiết bị. Người sử dụng sẽ cung cấp một kích thước len byte dữ liệu (tham số thứ 3 của hàm my\_write()) để ghi nằm trong bộ đệm buf (tham số thứ 2 của hàm my\_write()). Hàm my\_write() sẽ nhận dữ liệu đó từ bộ đệm mà người sử dụng cung cấp để ghi nó ra thiết bị thật, số byte thực sự ghi thành công cũng sẽ được dùng làm giá trị trả về của hàm.

Như vậy, hàm my\_read(), my\_write() cần được bổ sung thêm để có thể thực hiện được như các mô tả trên. Để minh họa đơn giản, chúng ta bổ sung trong mã nguồn driver một biến tĩnh toàn cục để cung cấp 1 byte dữ liệu cho cả tình huống đọc và ghi.

```
static char c;

static ssize_t my_read(struct file *f, char __user *buf, size_t len, loff_t *off)
{
    printk(KERN_INFO "Driver: read()\n");

    buf[0] = c;

    return 1;
}

static ssize_t my_write(struct file *f, const char __user *buf, size_t len, loff_t *off)
{
    printk(KERN_INFO "Driver: write()\n");

    c = buf[len - 1];

    return len;
}
```

Việc trao đổi dữ liệu qua bộ đệm giữa tầng ứng dụng và driver cần được thực hiện an toàn hơn thông qua 2 hàm APIs quan trọng là copy\_to\_user() và copy\_from\_user(). Áp dụng 2 hàm này trong mã nguồn minh họa như sau:

```
static char c;

static ssize_t my_read(struct file *f, char __user *buf, size_t len, loff_t *off)
{

```

```

    printk(KERN_INFO "Driver: read()\n");

    if (copy_to_user(buf, &c, 1) != 0)

        return -EFAULT;

    else

        return 1;

}

static ssize_t my_write(struct file *f, const char __user *buf, size_t len,
loff_t *off)

{

    printk(KERN_INFO "Driver: write()\n");

    if (copy_from_user(&c, buf + len - 1, 1) != 0)

        return -EFAULT;

    else

        return len;

}

```

Đến đây, chúng ta có một driver hoàn chỉnh với các hàm đọc ghi cung cấp dữ liệu thật. Thử nghiệm driver này theo các bước:

1. Biên dịch null driver sử dụng Makefile, tạo ra file .ko
2. Nạp driver sử dụng lệnh insmod, file thiết bị /dev/mynull được tạo ra
3. Ghi dữ liệu vào file /dev/mynull, bằng cách sử dụng `echo -n "Pugs" > /dev/ mynull`
4. Đọc dữ liệu từ file /dev/mynull bằng cách sử dụng `cat /dev/mynull` (Ctrl+C để dừng đọc)
5. Gỡ driver bằng cách sử dụng lệnh `rmmmod`.

Khi đọc file thiết bị /dev/mynull bằng lệnh cat, dữ liệu đọc được sẽ là chuỗi liên tục ký tự s bởi vì hàm đọc trả về ký tự cuối cùng ghi vào (lưu trữ trong biến tĩnh toàn cục c). Nếu muốn hàm đọc my\_read() chỉ trả về ký tự cuối cùng được ghi một lần thì cần sử dụng đến tham số thứ 4 của hàm này off (giá trị offset liên quan đến số lần đọc) để hàm trả về 1 byte đọc được trong lần đầu và 0 byte đọc được trong lần tiếp theo. Khi đó mã nguồn hàm read có thể viết như sau:

```

static ssize_t my_read(struct file *f, char __user
*buf, size_t len, loff_t *off)

{

```

```
printk(KERN_INFO "Read()\n");

/* You have just a single char in your buffer, so only 0 offset is valid */
if(*off > 0)

    return 0; /* End of file */

if (copy_to_user(buf, &c, 1))

    return -EFAULT;

*off++;

return 1;

}
```

## NGUỒN TỔNG HỢP

### Tiếng Việt

1. <https://quantrimang.com/tim-hieu-ve-linux-kernel-va-nhung-chuc-nang-chinh-cua-chung-72129>
2. <https://vimentor.com/vi/course/linux-device-driver?type=lesson>
3. <https://sites.google.com/site/embedded247/ddcourse/device-drivers-phan-6-cac-thao-tac-doi-voi-file-thiet-bi>

### Tiếng Anh

1. <https://www.howtogeek.com/howto/31632/what-is-the-linux-kernel-and-what-does-it-do/>
2. <http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/>
3. <https://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf>