ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

HỆ ĐIỀU HÀNH

ĐỒ ÁN 3: LINUX KERNEL MODULE

Giảng viên hướng dẫn: **thầy Phạm Tuấn Sơn**
Nhóm thực hiện: **18120078 – Ngô Phù Hữu Đại Sơn**
18120211 – Võ Thế Minh
18120227 – Phạm Văn Minh Phương
Lớp: **HĐH 18_4**
Khóa: **2018**

MỤC LỤC

A.THÔNG TIN KHÁI QUÁT

I. Thông tin nhóm

MSSV	HỌ TÊN	VAI TRÒ
18120211	Võ Thế Minh	Trưởng nhóm
18120078	Ngô Phù Hữu Đại Sơn	Thành viên
18120227	Phạm Văn Minh Phương	Thành viên

II. Bảng phân công công việc

MSSV	CÔNG VIỆC PHỤ TRÁCH
18120078	*random_module.c* (character device để tạo số ngẫu nhiên)
18120211	Viết chương trình ở user space(*test.c*).
18120227	Kiểm lỗi
18120078	Báo cáo phần 1, 2
18120211	Báo cáo phần 3, 4
18120227	Báo cáo phần 5, 6

B. NỘI DUNG

I. Mục tiêu của đồ án

Mục tiêu hiểu về Linux kernel module và hệ thống quản lý file và device trong linux, giao tiếp giữa tiến trình ở user space và code kernel space

II. Nội dung

1. Giới thiệu Linux Kernel

Linux Kernel được chia làm 6 thành phần:

- **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:

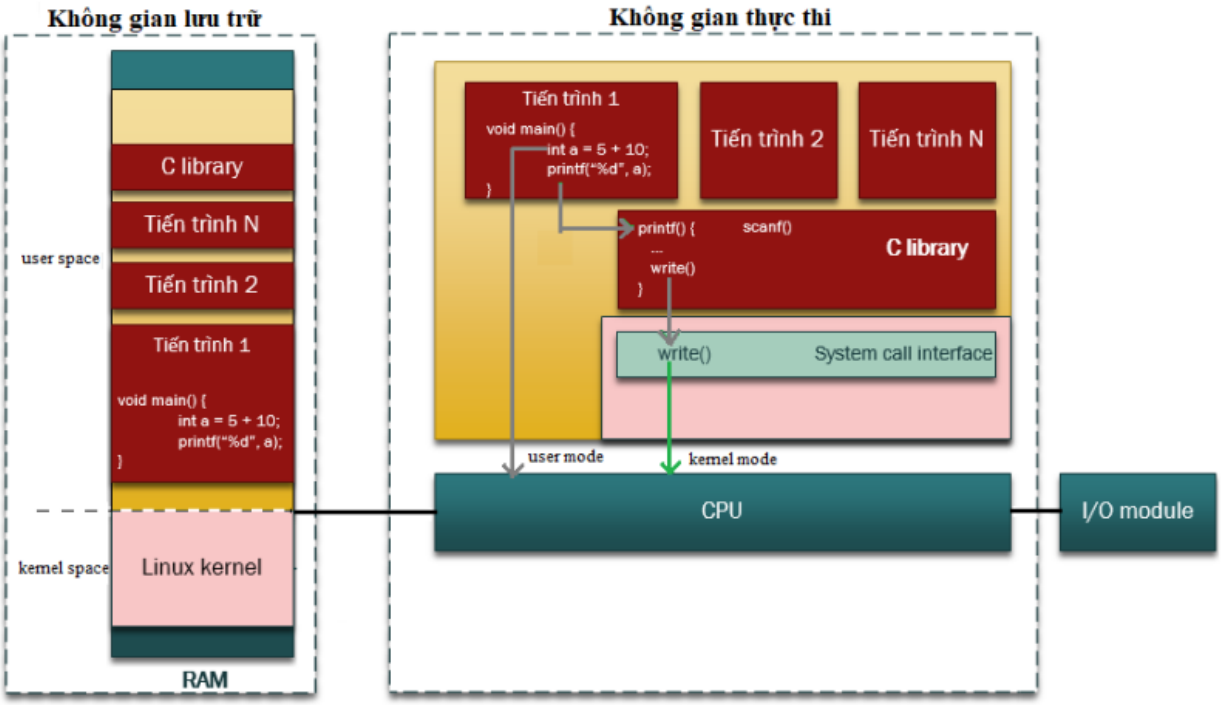
Thư mục	Vai trò
/arch	Chứa mã nguồn giúp Linux kernel có thể thực thi được trên nhiều kiến trúc CPU khác nhau như x86, alpha, arm, mips, mk68, powerpc, sparc,…
/block	Chứa mã nguồn triển khai nhiệm vụ lập lịch cho các thiết bị lưu trữ.
/drivers	Chứa mã nguồn để triển khai nhiệm vụ điều khiển, giám sát, 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:

- 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:

- **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



CPU có 2 chế độ thực thi:

- 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.

Khi một tiến trình cần sử dụng dịch vụ nào đó của kernel, tiến trình sẽ gọi một **system call** (Được cung cấp bởi kernel) nên CPU phải chuyển sang kernel mode để thực thi. Sau khi thực hiện xong yêu cầu của tiến trình thì kernel sẽ gửi trả lại kết quả cho tiền trình và chuyển lại về user mode để thực thi các lệnh tiếp theo. Đây là ngữ cảnh **process context.**

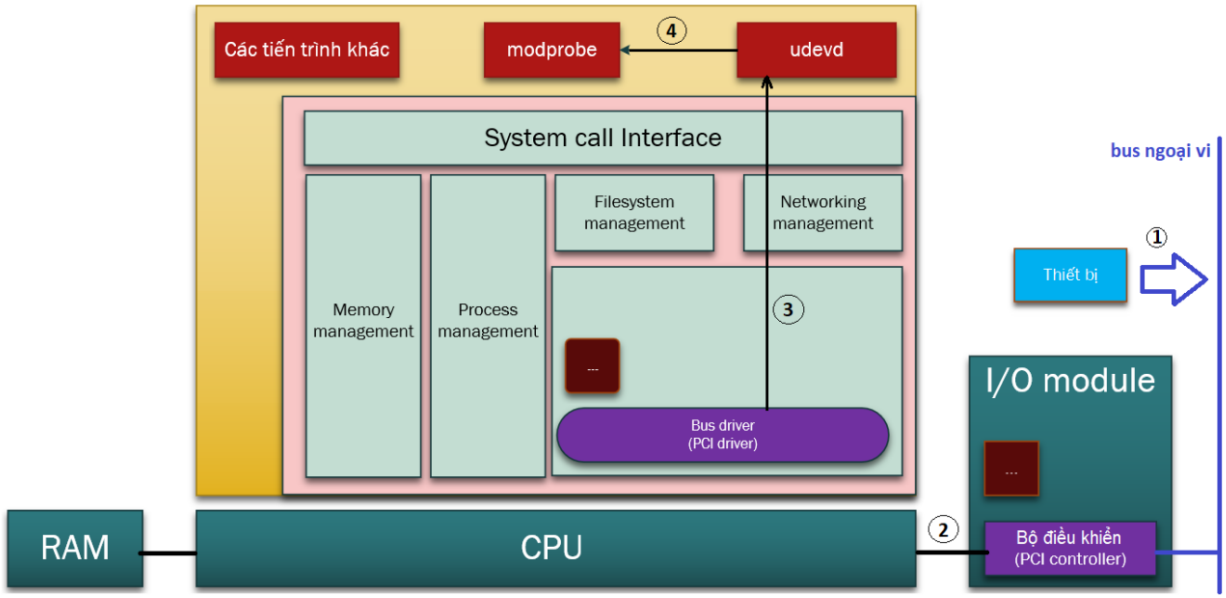
Trong ngữ cảnh interrupt context thì CPU phải ngừng thực thi các lệnh của tiến trình lại và chuyển chế độ thực thi sang kernel mode khi có một tín hiệu ngắt được gửi tới CPU và xử lý tín hiệu ngắt đó bằng một chương trình đặc biệt của kernel. Sau khi xử lý xong thì CPU trở lại user mode và tiếp tục thực hiện các lệnh tiếp theo của tiến trình.

2. Giới thiệu Linux Kernel Module

Để có thể làm việc với các thiết bị khác nhau thì ta cần driver nhưng lại không thể để quá nhiều driver trong kernel vì sẽ làm cho kích thước kernel lớn nên cần phải thiết kế driver dưới dạng module tách rời với kernel, chỉ được nạp vào kernel khi cần sử dụng.

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). Có 2 cách kích hoạt
  - *Cách 1*: Sử dụng **kmod** – một thành phần của Linux kernel hoạt động trong kernel space. Khi một thành phần của kernel cần đưa một module vào trong kernel space thì nó sẽ truyền tên cho *kmod* và *kmod* sẽ sinh ra tiến trình modprobe.
  - *Cách 2*: Sử dụng **udevd** – một tiến trình hoạt động trong user space. Khi có một thiết bị cắm vào hệ thống máy tính thì nó sẽ thay đổi điện trở trên bus ngoại vi để thông báo cho bộ điều khiển, rồi bus driver sẽ gửi bản tin chứa thông tin về thiết bị cho tiến trình *udevd*, udevd sau đó tìm driver tương thích với thiết bị và sinh tiến trình modprobe.
- Bước 2: Tiến trình modprobe kiểm tra file */lib/modules/<kernelversion>/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).



3. Cách viết một Linux Kernel Module

Tạo file *hello.c* chứa mã nguồn của Kernel Module

```
/*
 *  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 */
```

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 .

- *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.

Trong quá trình viết kernel module, các lập trình viên thường sử dụng hàm *printk()* để ghi lai 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.

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:

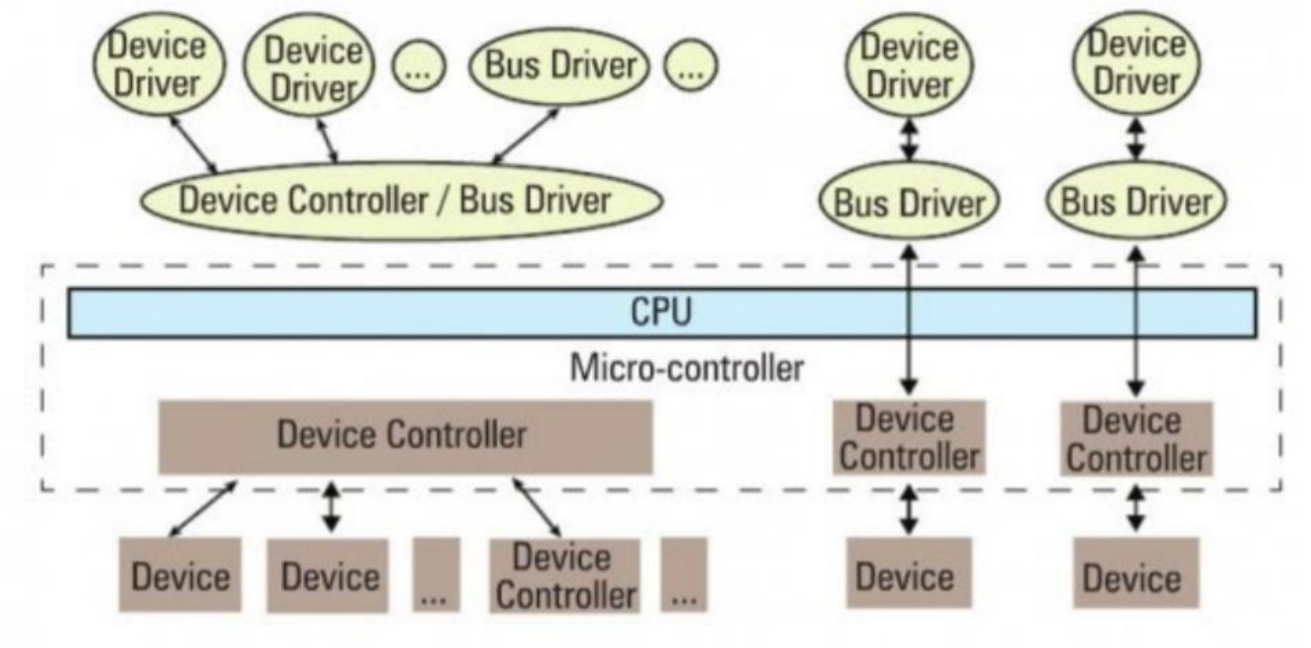
```
#cd /home/ubuntu/ldd/phan_1/bai_1_3
#vim Kbuild

EXTRA_CFLAGS = -Wall

obj-m        = hello.o
```

Để tạo ra kernel module, ta gõ lệnh *make* hoặc *make all*. Khi ta gõ lệnh "*make*", tiến trình make sẽ dựa vào *Makefile* và *Kbuild* để biên dịch mã nguồn, tạo ra 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. Cuối cùng, chúng ta sẽ dùng lệnh *rmmod* để tháo module ra khỏi kernel.

4. Cơ bản về driver trong Linux

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, làm việc với một thiết bị (chuột, bàn phím, màn hình, đĩa cứng, camera, …). một thành phần phần *device driver* 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.



Hình 1. Tương tác giữa thiết bị và driver

Một cách tổng quan, một driver sẽ bao gồm 2 phần quan trọng:

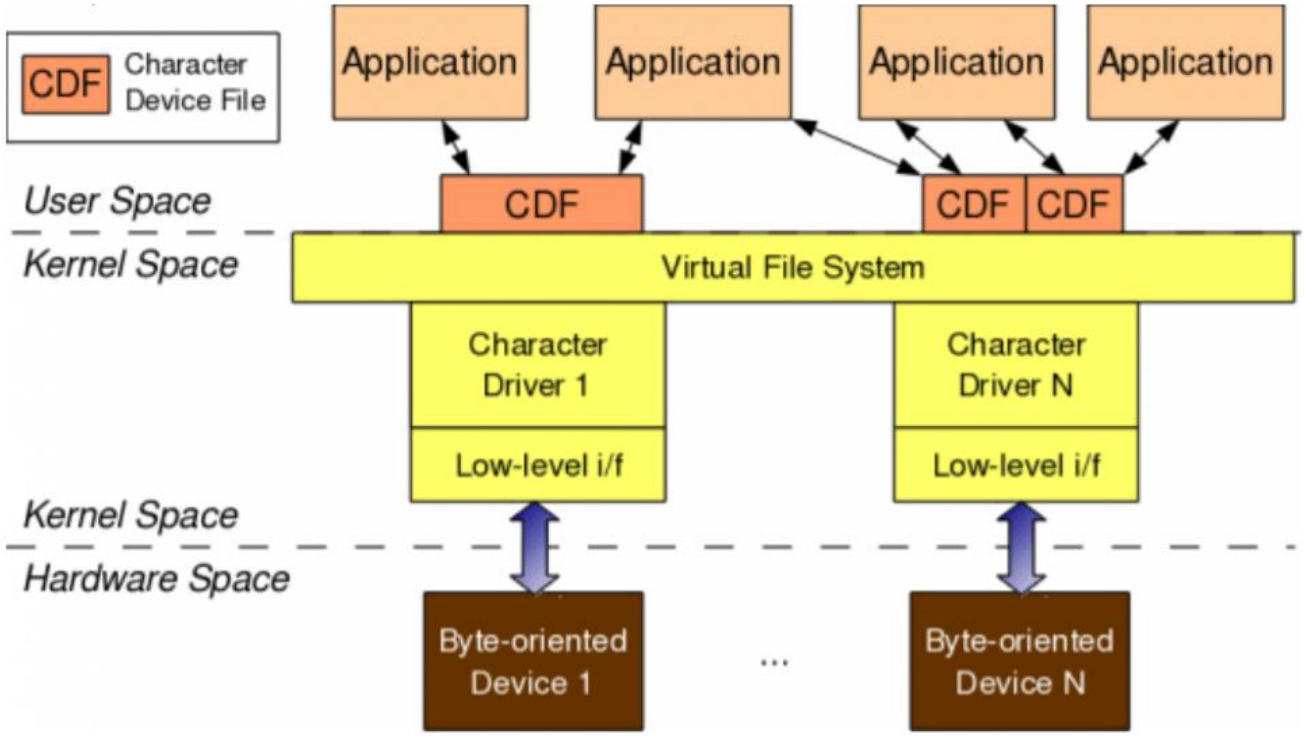
a) giao tiếp với thiết bị (*Device-specific*)
b) giao tiếp với hệ điều hành (*OS-specific*).

Thành phần giao tiếp với thiết bị (*device-specific*) của một driver là giống nhau đối với tất cả các hệ điều hành. Thành phần giao tiếp với hệ điều hành (*OS-specific*) gắn kết chặt chẽ với các cơ chế của hệ điều hành.

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)

5. Character Device Driver



Hình 1. Tổng quan về character driver trên Linux

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)

Các kết nối giữa chúng cần được thực hiên 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. 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.

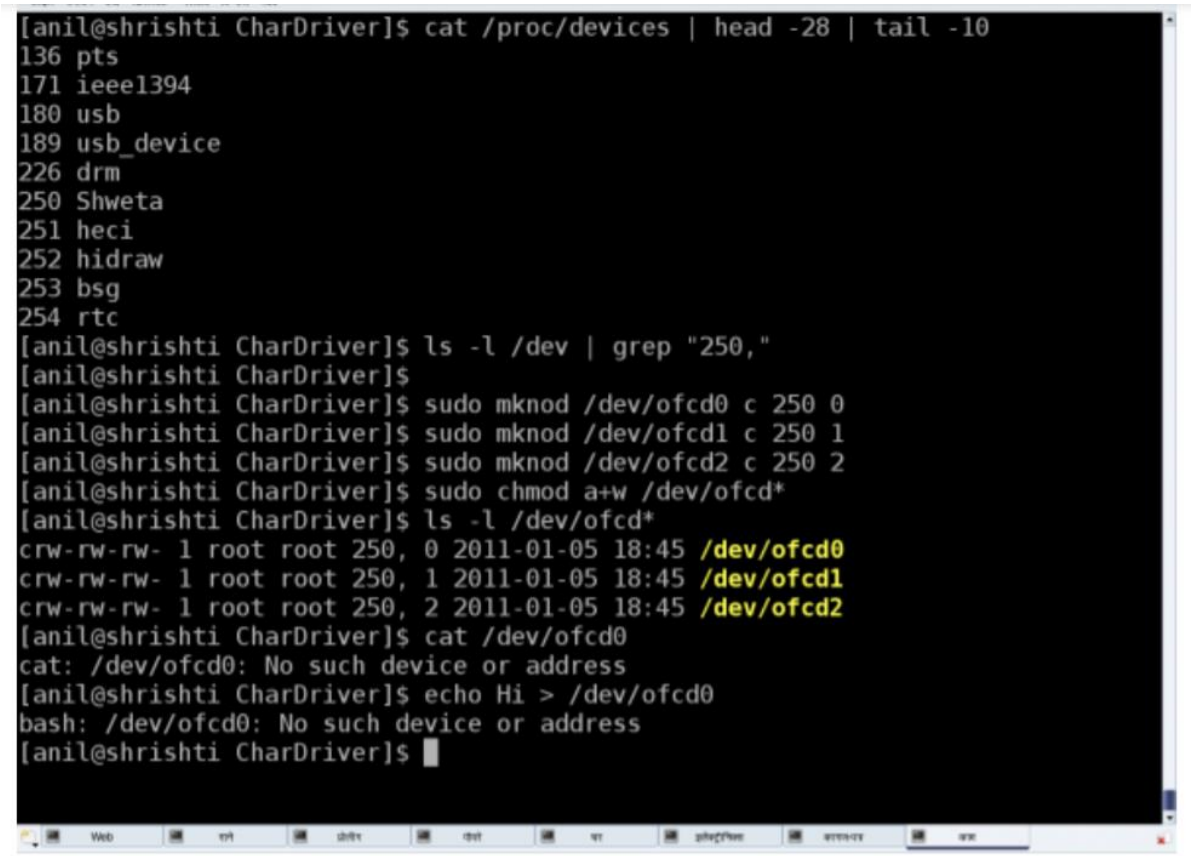
Việc kết nối giữa file thiết bị và driver được thực hiện thông qua 2 bước sau:

1. Đăng ký số hiệu cho file thiết bị
2. Kết nối các thao tác file thiết bị với các hàm tương ứng trong 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 dung *rmmod*

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*.



Tạo file thiết bị tự động

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.

Các thao tác với file thiết bị (File operations)

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.

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().*

6. Các thao tác với file device

Thao tác đọc file device

Khi người sử dụng muốn đọc dữ liệu từ file device tại */dev*, 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 và tìm ra driver cần thiết để triệu gọi hàm *my_read()* của driver này. 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 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

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

Khi đọc file thiết bị tại */dev* 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). 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.

C.TỔNG KẾT

I. Đánh giá đồ án

MSSV	Mức độ hoàn thành công việc	Đóng góp
18120078	100	40
18120211	100	30
18120227	100	30

II. Nguồn tham khảo

- https://github.com/84436/OS-Project-02
- https://vimentor.com/vi/lesson/linux-kernel-module