**Báo Cáo Training linux**

1. **PART 1**

+ Một driver sẽ baogồm 2 phần quan trọng

a) giao tiếp với thiết bị (Device-specific)

b) giao tiếp với hệ điều hành (OS-specific)

+ Driver trên Linux được phân chia thành 3 loại (phân cấp theo chiều dọc):

- Packet-oriented or the network vertical (driver hướng góidữ liệu)

- Block-oriented or the storage vertical (driver hướng khối dữ liệu)

- Byte-oriented or the character vertical (driver hướng byte/kýtự)

* **Packet-oriented hay network driver** gồm 2 phần:

+ network protocol stack

+ 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:

+ File-system drivers để giải mã các định dạng khác nhau trên các phân vùng lưu trữ khác nhau (FAT, ext, …)

+ 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)

VD: tty driver, input driver, console driver, frame-buffer drivers, sound driver,...

(tương ứng với các giao tiếp như RS232, PS/2, VGA, I2C, SPI,...)

**II. PART 2**

Để build 1 chương trình linux driver sử dụng kernel’s makefile.

Để insert, remove, xemthông tin của 1 module drive, dùngtậplệnhsau:

* lsmod : xem list các driver cótronghệthống.
* sudo insmod <module\_file> : Nạpmột module vàohệthống

(yêu cầu modules phụ thuộc nếu có phải được nạp trước)

* sudo modpobe <module> : Nạp một module vào hệthống

(Tự động tìm các module phụ thuộc nếu có để nạp)

* sudo rmmod <module> : removes/unloads module
* uname –r : lấy phiên bản nhân hệ thống Linux đang sử dụng.

+ Nơi để chứa các modules đã được biên dịch có sẵn, thường được tổ chức trong thư mục : */lib/modules/<kernel\_version>/kernel.*

(trongđó, <kernel\_version>: là phiên bản nhân hệ thống Linux đang sử dụng.)

+ Sau khi dung *lệnh insmod* thì chương trình driver *sẽ gọi hàm module\_init()* trong thư viện <linux/module.h>.

+ Tương tự nếu dùng *lệnh rmmod* thì chương trình sẽ gọi hàm module\_exit() trong thư viện <linux/module.h> . Hai hàm này dung để tạo, hoặc xóa 1 driver.

+ **dmesg:** để xem thông tin ghi log từ tầng nhân, kết hợp với lệnh tail để giới hạn số dòng muốn xem (do quá nhiều thông tin được log trên hệ thống).

***$ dmesg | tail -7***

+ **Hàm printk()** : Ghi log thông điệp ra một bộ đệm ở trong tầng nhân.

* Thực tế, hàm printk được chạy ngầm, nó thực thi giống như một thư viện, chỉ được kích hoạt từ tầng phần cứng hoặc từ tầng ứng dụng.
* Tất cả các lời gọi hàm printk sẽ xuất các thông điệp (ghi log) ra một bộ đệm ở tầng nhân. Sau đó, tiến trình syslog chạy ở tầng ứng dụng sẽ thu thập các thong điệp này và điều hướng nó đến một vài đầu ra xác định, ví dụ như đến file cấu hình */etc/syslog.conf.*
* Hàm printk sử dụng các macro để ghép với thông điệp được truyền vào theo sau nó tạo thành một xâu.

Các macro này được định nghĩa trong *linux/kernel.h*

#define KERN\_EMERG "<0>" /\* system is unusable \*/

#define KERN\_ALERT "<1>" /\* action must be taken immediately \*/

#define KERN\_CRIT "<2>" /\* critical conditions \*/

#define KERN\_ERR "<3>" /\* error conditions \*/

#define KERN\_WARNING "<4>" /\* warning conditions \*/

#define KERN\_NOTICE "<5>" /\* normal but significant condition \*/

#define KERN\_INFO "<6>" /\* informational \*/

#define KERN\_DEBUG "<7>" /\* debug-level messages \*/

* Tất cả các lời gọi hàm printk mặc định sẽ ghi thông điệp ra log file*: /var/log/messages*.
* Cũng có thể cấu hình để xuất thong điệp ra một đầu ra khác, ví dụ ra cổng nối tiếp (/dev/ttyS0), hoặc ra màn hình console, thường được thực hiện với thong điệp loại *KERN\_EMERG*

1. **PART 4**

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.*

Vídụ: serial drivers, audio drivers, video drivers, camera drivers, vàcác I/O drivers.

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 lien kết với driver thông qua hệ thống file ảo (virtual file system – VFS)

Việc kết nối từ ứng dụng (apps) 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)

+ **Số hiệu file thiết bị (Major và minor number)**

Để kết nối giữa app với device file( file thiết bị) có thể sử dụng tên của device file ( file thiết bị) nhưng *để kết nối giữa device file và device driver thì phải dùng số hiệu của device file không phải dùng tên. Số đấy được gọi là<major, minor>.*

Vì vậy trước khi tạo 1 device driver cần biết <major, minor> cần sử dụng.

$ cat /proc/devices # Liệt kê tên và các số hiệu major đã được đăng ký.

$ ls -l /dev/ | grep "^c" # Liệt kê các file thiết bị kiểu character khác nhau trên hệ thống.

+ **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 (device file).

Sử dụng một trong hai hàm API sau (được định nghĩa trong *linux/fs.h*):

+ int register\_chrdev\_region (dev\_t first, unsigned intcnt, char \*name);

+ int alloc\_chrdev\_region (dev\_t \*first, unsigned intfirstminor, unsigned intcnt, char \*name);

1. Kết nối các thao tác file thiết bị (device file) với các hàm tương ứng trong driver.

Dùng lệnh

+ int ***register\_chrdev\_region***(dev\_t first, unsigned int cnt, char \* name);

*Để đăng kí major cho device files khi biết chắc chắn số major chưa có trong /etc/devices, và muốn biết trước major của device file.*

**Arguments**

*from*

the first in the desired range of device numbers; must include the major number.

*count*

the number of consecutive device numbers required

*name*

the name of the device or driver.

+ **int alloc\_chrdev\_region** (dev\_t \*first, unsigned int firstminor, unsigned int *count*, char \*name);

*Để đăng kí major, minor cho device file mà hệ thống sẽ tự động đăng ký với số hiệu device file chưa có trong hệ thống.*

**Arguments**

*dev*

output parameter for first assigned number

*firstminor*

first of the requested range of minor numbers

*count*

the number of minor numbers required

*name*

the name of the associated device or driver

Allocates a range of char device numbers. The major number will be chosen dynamically, and returned (along with the first minor number) in *dev*. Returns zero or a negative error code.

Tạo ra device file bằng command line:

**Sudo mknod <name> <type of device c:- character b:- block device etc> <major> <minor>**

VD: mknod datvv c 250 0

1. **PART 5**

* **Tạo device file tự động.**

+ 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.

* **Hàm class\_create:** Lớp thiết bị (device class) được tạo ra như sau:

struct class \*cl = class\_create( struct module \* own, const char \* name );

*owner*

pointer to the module that is to “own” this struct class

*name*

pointer to a string for the name of this class.

**Description**

This is used to create a struct class pointer that can then be used in calls to class\_device\_create.

Note, the pointer created here is to be destroyed when finished by making a call to class\_destroy.

* **Hàm device\_create:**

device\_create - creates a device and registers it with sysfs

struct device \* device\_create = { struct class \* class,

struct device \* parent,

dev\_t devt,

const \* char \* fmt,

………….

};

***Argument:***

*class*

pointer to the struct class that this device should be registered to

*parent*

pointer to the parent struct device of this new device, if any

*devt*

the dev\_t for the char device to be added.

( tương ứng với cặp giá trị <major, minor>.)

*fmt*

string for the device's name

*...*

variable arguments

**Description**

* This function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class.
* A “dev” file will be created, showing the dev\_t for the device, if the dev\_t is not 0,0.
* If a pointer to a parent struct device is passed in, the newly created struct device will be a child of that device in sysfs. The pointer to the struct device will be returned from the call. Any further sysfs files that might be required can be created using this pointer.

**Note:**

* The struct class passed to this function must have previously been created with a call to class\_create.
* Khi hủy lớp thiết bị, thứ tự thực hiện là ngược lại:

*device\_destroy(cl, first);*

*class\_destroy(cl)*

* **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

**+ B1**: Tạo ra biến cấu trúc struct file\_operations pugs\_fops:

Điền vào cấu trúc này  các thao tác xử lý muốn dùng đối với file thiết bị đang viết driver

VD: các thao tác thông thường như *my\_open, my\_close, my\_read, my\_write*

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().*

**+ B2**: Điều khiển cấu trúc này đến hệ thống file ảo VFS bằng cách gọi hàm *cdev\_add()*

* **Hàm cdev\_add: thêm character device vào hệ thống.**

|  |  |  |
| --- | --- | --- |
| int **cdev\_add**( | struct cdev \* | *p*, |
|  | dev\_t | *dev*, |
|  | unsigned | *count*); |

**Arguments**

*p*

the cdev structure for the device

*dev*

the first device number for which this device is responsible

*count*

the number of consecutive minor numbers corresponding to this device

**Description:** cdev\_add adds the device represented by *p* to the system, making it live immediately. A negative error code is returned on failure.

* **Hàm cdev\_init: khởi tạo 1 cdev structure**

|  |  |  |
| --- | --- | --- |
| void **cdev\_init**( | struct cdev \* | *cdev*, |
|  | const struct file\_operations \* | *fops*); |

**Arguments**

*cdev*

the structure to initialize

*fops*

the file\_operations for this device

**Description**

Initializes *cdev*, remembering *fops*, making it ready to add to the system with cdev\_add.

**PART 6**

Việc trao đổi dữ liệu qua bộ đệm giữa tầng ứng dụng (app) và driver được thực hiện thông qua 2 hàm APIs quan trọng là copy\_to\_user() và copy\_from\_user()

* **Copy to user:** Đề đọc data của device file

|  |  |  |
| --- | --- | --- |
| unsigned long **copy\_to\_user**( | void \_\_user \* | *to*, |
|  | const void \* | *from*, |
|  | unsigned long | *n*); |

**Arguments**

*to*

Destination address, in user space.

*from*

Source address, in kernel space.

*n*

Number of bytes to copy.

**Context**

User context only. This function may sleep.

**Description**

Copy data from kernel space to user space.

Returns number of bytes that could not be copied. On success, this will be zero.

* **Copy from user:** Để ghi data vào device file

|  |  |  |
| --- | --- | --- |
| unsigned long **copy\_from\_user**( | Void \* | *to*, |
|  | const void \_\_user \* | *from*, |
|  | unsigned long | *n*); |

**Arguments**

*to*

Destination address, in kernel space.

*from*

Source address, in user space.

*n*

Number of bytes to copy.

**Context**

User context only. This function may sleep.

**Description**

Copy data from user space to kernel space.

Returns number of bytes that could not be copied. On success, this will be zero.

If some data could not be copied, this function will pad the copied data to the requested size using zero bytes.

**PART 7 Generic Hardware Access in Linux**

Run cat /proc/iomem : to list the memory map on your system.

Runcat /proc/meminfo **:** to get the approximate RAM size on your system.

Để truy cập vào generic hardware sử dụng thư viện <asm/io.h>

Sử dụng 2 hàm để mapping và unmapping giữa device bus addresses với virtual addresses

void \*ioremap(unsigned long device\_bus\_address, unsigned long device\_region\_size);

void iounmap(void \*virt\_addr);

* Để đọc và ghi giữ liệu vào device file sử dụng hàm :

unsigned int ioread8 (void \*virt\_addr);

unsigned int ioread16 (void \*virt\_addr);

unsigned int ioread32 (void \*virt\_addr);

unsigned int iowrite8 (u8 value, void \*virt\_addr);

unsigned int iowrite16 (u16 value, void \*virt\_addr);

unsigned int iowrite32 (u32 value, void \*virt\_addr);

Với virt\_addr là virtual address là giá trị trả về của hàm ioremap .

**PART 9 : I/O Control in Linux**

* Giới thiệu về ioctl()

Input - output control (ioctl) giống như các system call như open(), close(), read(),… có thể dùng với hầu hết các loại driver.

Ví dụ như valume control for audio device, display configuration for a video device, reading device registers…

Syntax:

*long ioctl (struct file \*f, unsigned int cmd, unsigned long arg);*

Điểm đặc biệt của hàm ioctl() là có thể truyền vào argument cmd để thực hiện công việc theo yêu cầu.

Define các argument cmd như sau:

#define CMD\_NAME \_IO (int type, int number)

* **Type**: an 8-bit integer selected to be specific to the device driver. type should be chosen so as not to conflict with other drivers that might be `listening'' to the same file descriptor. (Inside the kernel, for example, the TCP and IP stacks use distinct numbers since an ioctl sent to a socket file descriptor might be examined by both stacks.)
* **Number**: an 8-bit integer command number. Within a driver, distinct numbers should be chosen for each different kind of ioctl command that the driver services
* **data\_type**, the name of a type used to compute how many bytes are exchanged between the client and the driver. This argument is, for example, the name of a structure.

The macros used to generate command numbers are:

***\_IO( int type, int number):*** used for a simple ioctl that sends nothing but the type and number, and receives back nothing but an (integer) retval

***\_IOR( int type, int number, data\_type)****:* used for an ioctl that reads data from the device driver. The driver will be allowed to return sizeof(data\_type) bytes to the user

***\_IOW( int type, int number, data\_type)*** :similar to \_IOR, but used to write data to the driver

***\_IORW (int type, int number, data\_type)***: a combination of \_IOR and \_IOW. That is, data is both written to the driver and then read back from the driver by the client