Write a Simple Linux Driver

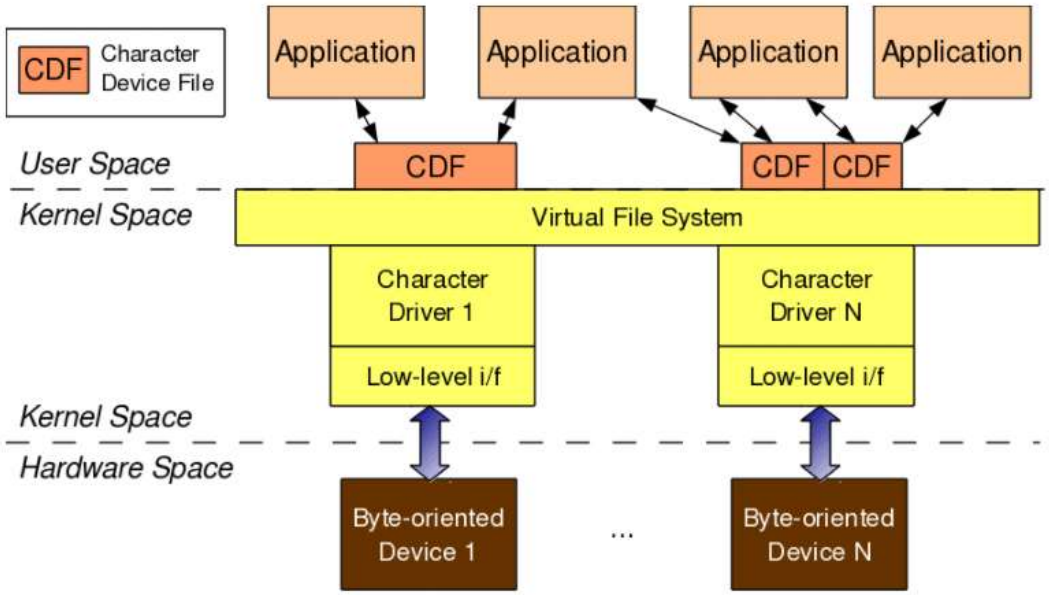
一、簡介

1. Kernel 的工作:
處理 System Call，將硬體回應回傳給 user process (System Call : OS interface, call to the OS service)

2. Device Driver 扮演的角色:
一種軟體，能讓 OS 認識某種硬體，並讓應用程式能利用這個硬體

3. Device Driver 授權形式會受連結方式所影響
Static link -> GPL
Dynamic link -> driver 開發者自訂

以 kernel module 形式提供的 device driver，開發者可透過 MODULE_LICENSE 這個 macro 將授權方式設定為以下七種之一：
GPL, GPL v2, GPL and additional rights, Dual BSD/GPL, Dual MIT/GPL, Dual MPL/GPL, Proprietary



二、Linux 驅動程式三類型

1. Linux device driver 可分成 3 種類型：
   (1) character device driver
   (2) block device driver
   (3) network device driver

2. 驅動程式本身可分成 2 個層面來討論：
   (1) virtual device driver
   往上層支援 Linux kernel 所提供的 Virtual File System 層，並藉此實作 system calls，Virtual device driver 的目的在於善用 Linux 的 APIs 來設計機制 (mechanism) 與行為 (behavior) 良好的驅動程式。
   (2) physical device driver
   往下層使用 Linux kernel 所提供的 device interface 來存取並控制實體硬體裝置，Physical device driver 則是討論「如何透過 I/O port 或 I/O memory」來控制裝置，也就是與晶片組的溝通。

三、System Call 與驅動程式的關係

System call 是 user application 與 Linux device driver 的溝通介面。

User application 透過呼叫 system call 來「叫起」driver 的 task，user application 要呼叫 system call 必須呼叫 GNU C 所提供的「wrapper function」，每個 system call 都會對應到 driver 內的一個 task，此 task 即是 file_operation 函數指標所指的函數。

Linux 驅動程式與 user application 間的溝通方式是透過 system call，實際上 user application 是以 device file 與裝置驅動程式溝通。要達成此目的，驅動程式必須建構在此「file」之上，因此 Linux 驅動程式必須透過 VFS（virtual file system）層來實作 system call。

/dev 目錄下的檔案稱為 device file，是 user application 用來與硬體裝置溝通的介面。

四、Device File

在 UNIX 系統底下我們把外部的周邊裝置均視為一個檔案，並透過此檔案與實體硬體溝通，這樣的檔案就叫做 device files，或 special files。

檔案屬性的第一個位元如果顯示為 “c” 表示這是一個字元型裝置的 device file、若為 “b” 表示這是一個區塊型裝置的 Device file。

```
chenging@chenging-VirtualBox:~$ sudo ls -la /dev/sda? /dev/ttyS?
brw-rw---- 1 root disk     8,  1 6月 23 20:13 /dev/sda1
brw-rw---- 1 root disk     8,  2 6月 23 20:13 /dev/sda2
brw-rw---- 1 root disk     8,  5 6月 23 20:13 /dev/sda5
crw-rw---- 1 root dialout 4, 64 6月 23 20:13 /dev/ttyS0
crw-rw---- 1 root dialout 4, 65 6月 23 20:13 /dev/ttyS1
crw-rw---- 1 root dialout 4, 66 6月 23 20:13 /dev/ttyS2
crw-rw---- 1 root dialout 4, 67 6月 23 20:13 /dev/ttyS3
crw-rw---- 1 root dialout 4, 68 6月 23 20:13 /dev/ttyS4
crw-rw---- 1 root dialout 4, 69 6月 23 20:13 /dev/ttyS5
crw-rw---- 1 root dialout 4, 70 6月 23 20:13 /dev/ttyS6
crw-rw---- 1 root dialout 4, 71 6月 23 20:13 /dev/ttyS7
crw-rw---- 1 root dialout 4, 72 6月 23 20:13 /dev/ttyS8
crw-rw---- 1 root dialout 4, 73 6月 23 20:13 /dev/ttyS9
```

Device file 的 major number 代表一個特定的裝置，例如 major number 為 1 為 null 虛擬裝置，major number 定義於 kernel 文件目錄 Documentation/devices.txt 。Minor number 代表裝置上的子裝置，例如同一個硬碟上的分割區就用不同的 minor number 來代表，但其 major number 相同。

Device File 與 Driver 的關係

我們在設計 device driver 時，會先透過一個 “註冊” (register) 的動作將自己註冊到 kernel 裡，註冊時，我們會指定一個 major number 參數，以指定此驅動程式所要實作的週邊裝置。當 user 開啟 device file 時，kernel 便會根據 device file 的 major number 找到對應的驅動程式來回應使用者。Minor number 則是 device driver 內部所使用，kernel 並不會處理不同的 minor number。

設計 device driver 的第一個步驟就是要定義 driver 所要提供的功能 (capabilities)，當 user application 呼叫 open() system call 時，kernel 就會連繫相對應的 driver 來回應使用者。

file_operations 是學習 device driver 最重要的一個資料結構，file_operations 內的成員為函數指標，指向 “system call 的實作函數”。file_operations 即是圖中的 VFS 層。換句話說，Linux 驅動程式是透過 file_operations 來建構 VFS 層的支援。而 file_operation 裡的函數指標，即是指向每一個 system call 的實作函數。

五、Linux 驅動程式一般化設計流程

Virtual device driver 往上是為了連結 Linux kernel 的 VFS 層，physical device drvier 往下是為了存取實體硬體。

Virtual Device Driver

Virtual device driver 的目的在於設計一個「機制」良好的 kernel mode 驅動程式，virtual device driver 也必須考慮與 user application 的互動。實作上，則是需要善用 kernel 所提供的介面（interface），即 kernel APIs。

Virtual device driver 再分為 3 階段的觀念實作：

1. 定義 file_operations
2. 實作 system calls
3. 註冊 driver (VFS)

fops 是指向 file_operations 結構的指標，驅動程式呼叫 register_chrdev() 將 fops 註冊到 kernel 裡後，fops 便成為該 device driver 所實作的 system call 進入點。實作 system call 的函數便是透過 file_operations 結構來定義，我們稱實作 system call 的函數為 driver method。

kernel 會在需要時回呼 (callback) 我們所註冊的 driver method。因此，當 driver 裡的 method 被呼叫時，kernel 便將傳遞參數（parameters）給 driver method，driver method 可由 kernel 所傳遞進來的參數取得驅動程式資訊。

註冊 driver 的動作呼叫 register_chrdev()函數完成，此函數接受 3 個參數如下：

major：要註冊的裝置 major number
name：device 名稱
fops：driver 的 file operation

「註冊」這個動作觀念上是將 fops 加到 kernel 的 VFS 層，因此 user application 必須透過「device file」才能呼叫到 driver method。註冊這個動作的另一層涵意則是將 driver method 與不同的 system call 做「正確的對應」，當 user application 呼叫 system call 時，才能執行正確的 driver method。

六、A Simple Example : Hello World

hello.c

```
chenging@chenging-VirtualBox: ~/test
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void){
    printk("<1> Hello world!\n");
    return 0;
}

static void hello_exit(void){
    printk("<1> Bye, cruel world\n");
}

module_init(hello_init);

module_exit(hello_exit);
```

Makefile

```
chenging@chenging-VirtualBox: ~/test
PWD := $(shell pwd)
KVERSION := $(shell uname -r)
KERNEL_DIR = /usr/src/linux-headers-$(KVERSION)/

MODULE_NAME = hello
obj-m := $(MODULE_NAME).o

all:
        make -C $(KERNEL_DIR) M=$(PWD) modules
clean:
        make -C $(KERNEL_DIR) M=$(PWD) clean
```

sudo insmod hello.ko

dmesg # 觀察 kernel

```
[   33.854569] ISO 9660 Extensions: Microsoft Joliet Level 3
[   33.878109] ISO 9660 Extensions: RRIP_1991A
[ 1630.332335] <1> Hello world!
[ 1675.720248] <1> Bye, cruel world
[ 1871.134111] <1> Hello world!
```

lsmod | grep "hello"

```
chenging@chenging-VirtualBox:~/test$ lsmod | grep "hello"
hello                  16384  0
```

sudo rmmod hello.ko

dmesg

```
[ 1871.134111] <1> Hello world!
[ 1935.067125] <1> Bye, cruel world
```

七、重點整理

將 driver 自己「註冊」到 kernel 的 VFS 層，註冊時所要呼叫的函數根據裝置類型的不同而不同。

將驅動程式「註冊」(registration) 至 kernel 的動作必須在 init_module()函數裡實作。根據裝置類型的不同，所呼叫的函數也不同，以下是幾個基本的裝置註冊函數：

<table>
<tr><td>int register_chrdev(unsigned int major, const char * name, struct file_operations *fops)：註冊字元型驅動程式</td><td rowspan="2">第 1 個參數：為 device file 的 major number。該 device file 應在 Linux 系統底下以 root 身份手動建立。
第 2 個參數：
第 3 個參數：為驅動程式的 fops。</td></tr>
<tr><td>int register_blkdev(unsigned int major, const char *name, struct file_operations *fops)：註冊區塊型驅動程式</td></tr>
<tr><td colspan="2">int usb_register(struct usb_driver *new_driver)：註冊 USB 驅動程式</td></tr>
<tr><td colspan="2">int pci_register_driver(struct pci_driver *)：註冊 PCI 驅動程式</td></tr>
</table>

註冊的動作是寫在 init_module()裡，因此當使用者執行 insmod 載入驅動程式時，register_chrdev()便會執行。由此可知，註冊驅動程式的時機為 insmod 時。相對的，在 rmmod 時，必須執行解除註冊的動作，此動作必須實作在 cleanup_module()函數裡。

<table>
<tr><td>int unregister_chrdev(unsigned int major, const char * name) ：解除註冊字元型驅動程式</td><td rowspan="2">第 1 個參數：為 device file 的 major number。該 device file 應在 Linux 系統底下以 root 身份手動建立。
第 2 個參數：
第 3 個參數：為驅動程式的 fops。</td></tr>
<tr><td>int unregister_blkdev(unsigned int major, const char *name) ：解除註冊區塊型驅動程式</td></tr>
<tr><td colspan="2">void usb_deregister(struct usb_driver *driver)：解除註冊 USB 驅動程式</td></tr>
<tr><td colspan="2">pci_unregister_driver(struct pci_driver *drv) ：解除註冊 PCI 驅動程式。</td></tr>
</table>

Linux 驅動程式的「註冊」是一個非常重要的動作，這個動作代表 Linux 驅動程式是一個嚴謹的分層式架構；換句話說，Linux 驅動程式的分層（layered）關係可透過「註冊」的程序來分析。

八、完整裝置驅動 memory.c

An assignment to build a memory driver which can convert lowercase to uppercase in Linux

memory.c

```c
/* Necessary includes for device drivers */
#include <linux/init.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>       /* printk() */
#include <linux/slab.h>         /* kmalloc() */
#include <linux/fs.h>
#include <linux/errno.h>        /* error codes */
#include <linux/types.h>        /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h>        /* O_ACCMODE */
#include <asm/switch_to.h>      /* cli(), *_flags */
#include <asm/uaccess.h>        /* copy_from/to_user */

MODULE_LICENSE("Dual BSD/GPL");

/* Declaration of memory.c functions */

int memory_open(struct inode *inode, struct file *filp);
int memory_release(struct inode *inode, struct file *filp);
ssize_t memory_read(struct file *filp, char *buf, size_t count, loff_t *f_pos);
ssize_t memory_write(struct file *filp, char *buf, size_t count, loff_t *f_pos);
int memory_init(void);
void memory_exit(void);

/* Declaration of the init and exit functions */

module_init(memory_init);
module_exit(memory_exit);

/* Global variables of the driver */

/* Major number */

int memory_major = 60;
```

```c
/* Buffer to store data */

char *memory_buffer;

/* Structure that declares the usual file */

/* access functions */

struct file_operations memory_fops = {

    read: memory_read,
    write: memory_write,
    open: memory_open,
    release: memory_release
};

int memory_open(struct inode *inode, struct file *filp) {

    /* Sucess */

    return 0;
}

int memory_release(struct inode *inode, struct file *filp) {

    /* Success */

    return 0;
}

ssize_t memory_read(struct file *filp, char *buf, size_t count, loff_t *f_pos) {

    /* Transfering data to user space */

    copy_to_user(buf, memory_buffer, 1);

    /* Changing reading position as best suits */

    if (*f_pos == 0) {
        *f_pos += 1;
        return 1;
    } else {
        return 0;
    }
}

ssize_t memory_write(struct file *filp, char *buf, size_t count, loff_t *f_pos) {

    char *tmp;
    char to_upper = memory_buffer[0];

    tmp = buf + count - 1;
    copy_from_user(memory_buffer, tmp, 1);

    /* Conver lower to upper case */
    if (to_upper >= 97 && to_upper <= 122) {
        to_upper = to_upper - 32;
        memory_buffer[0] = to_upper;
    }
    printk(KERN_ALERT "memory_write\n");
    return 1;
}

void memory_exit(void) {

    /* Freeing the major number */
```

```
    if (memory_buffer) {
        kfree(memory_buffer);
    }

    printk("<1>Removing memory module\n");
}

int memory_init(void) {

    int result;

    /* Registering device */

    result = register_chrdev(memory_major, "memory", &memory_fops);

    if (result < 0) {
        printk("<1>memory: cannot obtain major number %d\n", memory_major);
        return result;
    }

    /* Allocating memory for the buffer */

    memory_buffer = kmalloc(1, GFP_KERNEL);

    if (!memory_buffer) {
        result = -ENOMEM;
        goto fail;
    }

    memset(memory_buffer, 0, 1);


    printk("<1>Inserting memory module\n");

    return 0;

    fail:
        memory_exit();

        return result;
}
```

```
chenging@chenging-VirtualBox: ~/test
chenging@chenging-VirtualBox:~/test$ sudo insmod memory.ko
chenging@chenging-VirtualBox:~/test$ echo -n abcdef >/dev/memory
chenging@chenging-VirtualBox:~/test$ cat /dev/memory
fchenging@chenging-VirtualBox:~/test$
```

dmesg

```
[50982.634110] <1>Inserting memory module
[50985.573187] memory_write
[50985.573192] memory_write
[50985.573193] memory_write
[50985.573194] memory_write
[50985.573195] memory_write
[50985.573196] memory_write
[51034.519724] <1>Removing memory module
```