- **LDDP:零、Linux Device Driver Programming**

LDDP:一、Linux 裝置驅動程式的概要

LDDP:二、Linux 的授權

LDDP:三、準備開發裝置驅動程式

LDDP:四、開發驅動程式的第一步

LDDP:五、開發驅動程式需要的基礎知識

LDDP:六、實際撰寫驅動程式

LDDP:七、控制硬體

LDDP:八、記憶體

LDDP:九、計時器

LDDP:十、同步與鎖定

LDDP:十一、中斷

LDDP:十六、測試與除錯

LDDP:十七、驅動程式設計與實作實務

一、Linux 裝置驅動程式的概要

1-2、kernel 的運作

kernel 的工作：

- 處理 user process 的要求(系統呼叫)
- 將硬體的回應回傳系 user process
-
-
- 1-5、驅動程式扮演的角色
- 驅動程式(device driver)：

一種軟體，能讓 OS 認識某種硬體(裝置)，並讓應用程式能夠利用這個硬體。

二、Linux 的授權

2-5、device driver 的授權

device driver 的授權會受到連結方式影響：

1. static link -> GPL
2. 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：

1. GPL
2. GPL v2
3. GPL and additional rights
4. Dual BSD/GPL
5. Dual MIT/GPL

6. Dual MPL/GPL
7. Proprietary

沒有定義此 macro 時，將自動視為 Proprietary，這個 macro 可讓 kernel 知道 device driver 的授權方式。

三、準備開發裝置驅動程式

3-1、開發驅動程式的要點

建構(build)

將原始碼編譯之後只會得到目的檔(.o)，這不是可以直接執行的形式。
開發應用程式的時候，會在編譯好之後「連結」，建立執行格式(ELF)的檔案。
編譯與連結動作加起來稱為「建構(build)」。
而驅動程式沒有連結需求，以預先準備好的 Makefile 建立驅動程式專用的物件檔(.ko)即可。
.ko 檔也是 ELF(Executable and Linking Format)格式，但不能直接執行，只能以 insmod 指令載入並嵌入 kernel。

能否使用 printf

驅動程式是在與 kernel 的連結狀態下運作的，而這個環境下沒有「標準輸出」的觀念，所以 Linux 準備了「printk()」這個與 printf 十分類似的函式。
printk() 會向稱為「kernel buffer」的一小塊資料空間(128KB)寫入訊息，其中 kernel buffer 為 ring buffer。
想檢查 kernel buffer 的內容時，可以執行 dmesg 指令，或是直接查詢 syslog 訊息紀錄(/var/log/messages)，或是直接以 cat /proc/kmsg 得知即時訊息。

Kernel Module 相關指令介紹：

驅動程式的載入與卸載

透過 insmod 指令可以載入 module。
lsmod 指令可以列出現在載入的所有 kernel module。
卸載驅動程式的工作可由 rmmod 指令完成。
「cat /proc/modules」會列出所有已載入的驅動程式。

modprobe

modprobe 同樣可用來載入驅動程式，但有一些不同：
引數不是檔名，而是「module name」。
會自動到 /lib/modules/'uname -r' 搜尋檔案。
會參考 /lib/modules/'uname -r'/modules.dep ，如果有需要用到其它的 modules，就會自動一起載入。

modinfo
如果要檢視 Linux 下 kernel module 的版本，可以透過 modinfo 指令實現，引數同樣是「module name」。

有關 module name 可以在 driver 的 makefile 中，搜尋 "MODNAME" 即可找到。

四、開發驅動程式的第一步
裝置驅動程式分成動態連結、靜態連結兩種。

4-1、建立動態裝置驅動程式
最初的程式
hello.c

```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
printk(KERN_ALERT "driver loaded\n");
return 0;
}
static void hello_exit(void) {
printk(KERN_ALERT "driver unloaded\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

建構驅動程式

Makefile:

```
obj-m := hello.o

all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

「obj-m」指定最後要建立的 .ko 檔是以哪些 .o 檔構成的。

想建立「hello.ko」，這邊指定的就是「hello.o」，「:=」為 GNU make 的擴充指派運算子。

make 指令的 「-C」選項，是指定 Makefile 的位置，此範例是以 uname 取得 kernel 版本，如果編譯給其它 kernel 使用，需修改路徑。

執行 make 即會得到所需的 「hello.ko」。

以 file hello.ko 檢查，會發現它是個 ELF binary 檔案。

以 modinfo hello.ko 檢查，可以取得驅動程式的授權與版本標記。

如果想看 pre-processor (預處理;cpp) 之後的結果，並將這個結果存成檔案的話，可以在 Makefile 中多加這一行：

```
CFLAGS += -E
```

不需要行號的話，可以再加「-P」。

make 之後，就會在 gcc 的 -o 指定檔案存入 pre-process 結果。

驅動程式的載入與卸載

透過 insmod 指令可以載入 module。

lsmod 指令可以列出現在載入的所有 kernel module。

卸載驅動程式的工作可由 rmmod 指令完成。

「/proc/modules」會列出所有已載入的驅動程式。

modprobe

modprobe 同樣可用來載入驅動程式，但有一些不同：

引數不是檔名，而是「module名」。

會自動到 /lib/modules/'uname -r' 搜尋檔案

會參考 /lib/modules/'uname -r'/modules.dep，如果有需要用到其它的 modules，就會自動一起載入

開機時載入驅動程式

Linux 在開機時自動載入驅動程式的功能，是在 rc script 之內以 insmod 與modprobe 指令完成的。
常用的 rc script 如下：
/etc/rc.local
/etc/rc.sysinit
/etc/rc.d/rc?.d

自訂 Makefile

加上「V=1」就可以看到詳細的建構過程。

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) V=1 modules
```

支援許多個原始檔：

```
CFILES := main.c sub.c

obj-m := hello.o


hello-objs := $(CFILES:.c=.o)


all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules


clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

這邊要注意的是 CFILES 中的檔案名稱，不能與 obj-m 的名稱有相同的情況，否則 make 會產生 confusion。

補充說明

對多個 .c 檔情況做一個說明：希望創建一個名叫 hello 的 module，有三個 .c 檔，分別為 hello.c, file1.c 和 file2.c。
這樣是有問題的，因為在 Makefile 中 obj-m := hello.o，這是指定模塊的名稱, hello-objs := file1.o file2.o hello.o，這裡是說 hello module 包括的 .obj 檔，如果在裡面不填寫 hello.o，那麼實際並沒有編譯hello.c，而是在 CC[M] file1.o 和 file2.o，通過 LD[M] 得到模塊 hello.o，如果在這裡填寫了 hello.o，那麼在 obj-m 和 hello-objs 中都含有 hello.o，對 make 來講會產生循環和混淆，因此不能這樣寫。

如果我們由多個 .c 檔來構造一個模塊，那麼 .c 檔的名字不能和 module 名字一樣， 在這個例子中我們可以將 hello.c 改名為 main.c，在Makefile中obj-m := hello.o ，hello-objs = file1.o file2.o hello_main.o。

main.c

```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

extern void sub(void);

static int hello_init(void) {
printk(KERN_ALERT "driver loaded\n");
sub();
return 0;
}
static void hello_exit(void) {
printk(KERN_ALERT "driver unloaded\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

sub.c

```
#include
#include

void sub(void) {
printk("%s: sub() called\n", __func__);
}
```

加到 kernel 配置選單

可以將自己寫的驅動程式加到 kernel 配置選單內，這邊試著把它加到「Character devices」分類底下。
首先，到 Linux kernel 的原始碼目錄(/usr/src/kernels/2.6.23.1-42.fc8-i686)內的 devices/char 建立子目錄，然後將原始檔放到裡面。

```
cd drivers/char
mkdir hello
cd hello
ls
```

Makefile main.c sub.c
並修改此 Makefile 成如下：

```
obj-$(CONFIG_HELLO) += hello.o
hello-objs := main.o sub.o
```

接著修改 Character devices 本身的 Makefile 與 Kconfig。
在 Makefile 中，新增一行：

```
obj-$(CONFIG_HELLO) += hello/
```

在 Kconfig 中，新增四行：

```
config HELLO
tristate "hello driver"
---help---
This is a sample driver.
```

做好上述修正後，在配置 kernel 的選單中，就會看到「hello」驅動程式了。

```
make menuconfig
```

驅動程式的依賴關係

當驅動程式需要呼叫另一個驅動程式提供的函式時，就是「這兩個驅動程式彼此依賴」。

要查閱 kernel 內含的 symbol name ，可以查閱 /proc/kallsyms。

4-2、建立靜態裝置驅動程式

直接連結到 vmlinux 的驅動程式類型，其建立方式與 kernel module 幾乎相同，只有連結的方式不一樣。
在從 kernel 配置選單建構驅動程式時，需要
在 kernel 配置選單中，將驅動程式的類型設為「*」
設成「*」後，就會靜態連結到 kernel 之內，不會產生獨立的 .ko 檔。

以此種 kernel 開機，會在 runlevel 1 的階段就載入驅動程式。
啟動後，用 lsmod 看不到此驅動程式，因為它不是 kernel module。

4-3、原始碼解說

檔頭檔位於 /lib/modules/`uname -r`/build/include 目錄內。
接著透過 MODULE_LICENSE 巨集定義驅動程式的授權方式。
載入裝置驅動程式的進入點，函式名稱可透過 module_init 巨集指定，基本上，函式都要加上 static，讓命名空間階制在檔案內。
printk() 輸出的資料會跑到 kernel buffer，可以用 dmesg 指令查閱。
int printk(const char *fmt, ...);
格式化字串(fmt) 的開頭加上「<編號>」的話，就會指定訊息的等級，在 include/linux/kernel.h 之內定義有相關巨集。
以 rmmod 卸除驅動程式時，呼叫函式的進入點是透過 module_exit 巨集指定。
hello.ko在建立時，原始碼裡不過只有兩個函式，在建構時，會加入一些其它符號，用 readelf 指定列出檔案中的符號就可以看到。
readelf -s hello.ko

4-4、進入點

呼叫 main 函式的流程

一般應用程式中，都覺得啟動程式後馬上會呼叫 main() 是理所當然的，但實際上在呼叫之前需要做不少事情。
用 strace 與 ltrace 之類的指令，可以看到系統呼叫、函式庫呼叫的內容。
file - determine file type
ldd - print shared library dependencies
strace - trace system calls and signals
ltrace - A library call tracer
gcc test.c
file a.out
ldd ./a.out
strace ./a.out
ltrace ./a.out

這樣就能發現 C語言的 main() 其實是由名為 glibc 的 C 函式庫呼叫的，而在 main() 執行 return 之後也是回到 glibc。

對應用程式來說，main()可以說是要讓 glibc 呼叫的入口，這個入口就稱為「進入點(entry point)」。

裝置驅動程式的進入點

驅動程式的進入點與一般應用程式不同，必須準備很多個，其中至少需要兩個進入點：

insmod 與 modprobe 呼叫的初始化函式

rmmod 呼叫的結束函式

以上述範例來說，分為是 hello_init() 及 hello_exit()。

其它的進入點還包含：

系統呼叫

中斷服務程序

計時器程序…等等。

「系統呼叫」是應用程式與 kernel 交換資訊用的介面，產生系統呼叫時，會在 CPU 引發中斷 (INT 80h)，此時執行環境會從 user space 轉到 kernel space。

系統呼叫會在使用 open()與 read()、close()、ioctl() 之類的函式時自動產生。

驅動程式必須為每個系統呼叫分別準備進入點。

「中斷服務程序(ISR: Interrupt Service Routine)」是驅動程式在所控制的硬體產生「中斷」時，被呼叫用來處理這類信號(signal)的函式。

中斷服務程序負責處理中斷，但不知何時會發生中斷，所以是「非同步(asynchronous;ASYNC)」，系統呼叫則是「同步(synchronous:SYNC)」。

「計時器程序」是由 kernel 的計時器處理程序在經過特定的時間、或是以固定間隔執行特定工作時回呼的進入點。

由於計時器處理程序本身也是硬體「計時器中斷」的處理程式，所以被它呼叫的計程器程序也算中斷服務程序的一種。

執行環境

要理解驅動程式的運作，就必須先理解「context(執行環境)」。

裝置驅動程式的 context 的意義是

從進入點開始，到最後一個函式的整個處理流程。

對驅動程式而言，只有下面這2種 context：

Process context(一般行程執行環境)

Interrupt context(中斷執行環境)

驅動程式的 Process context 個數為應用程式進入點的個數總和。

同樣地， interrupt context 的個數是核心進入點的個數總和。

重要的是：

所有的 context 都可能同時發生

Context 何時都可能發生(非同步)

五、開發 driver 需要的基礎知識

以 Character device driver 為例，學會 user process 將如何透過 device file 與 driver 作溝通。

5-1、執行環境

裝置驅程程式屬於 kernel 程式碼的一部分， driver 在 kernel space 運作。
kernel 的程式分成兩種 context：
Process context (一般行程執行環境)
Interrupt context (中斷執行環境)
Process context 也稱為 user context 或 thread context。
另外還有「kernel context(核心執行環境)」這個稱呼，主要代表在 kernel 內執行的程式碼，可能是 user context 也可能是 interrupt context。
Process context 可以 sleep，可以被 preempt，處理時間可以拖長沒關係。
Interrupt context 不可以 sleep，也不可以被 preempt，處理時間要極力縮短。
但就算是 process context，如果控制權回到 kernel 之內的話，在 kernel 內的 process context 是不能被 preempt 的。
在 Interrupt context 之內，必須確定用到的 kernel 函式都不會 sleep。
雖然前面提到 kernel space 不能被 preempt，但只要啟動 kernel 配置選項的「CONFIG_PREEMPT」，kernel space 也會變成可以 preempt，切換執行內容的時機僅限於 spin lock 鎖定的地方。
Process type and features -> Preemption Model -> Preemptible Kernel

5-2、資料模型

Data model(資料模型) 指的是 C 語言的 int、long 等等資料型別分別佔用多少 bytes。
在 ANSI C 的規格中， int、long 等資料型別的大小都沒明確定義，完全視執行環境而定。
如果要無視平台差異、使用固定大小的資料型別的話，就要用 u8 或 u32 之類的 typedef。

```
typedef signed char s8;
typedef unsigned char u8;
typedef signed short s16;
typedef unsigned short u16;
typedef signed int s32;
```

typedef unsigned int u32;

typedef signed long s64;

typedef unsigned long u64;

5-3、Endian

有 endian 差異的部分主要包含：

CPU

Bus(PCI 或 USB 等等)

網路封包

EEPROM 等的資料內容

現在由於 Intel 與 Microsoft 的成功，因此世界的主流是 little endian。

Intel 主導訂定的 PCI bus 規格也採用 little endian。

網路封包由於歷史因素，多半使用 big endian。

EEPROM 之類的 NVRAM(Non Volatile Ram) 儲存資料的格式會在儲存時固定下來，所以 driver 在讀寫 NVRAM 時必須注意 endian 的不同。

在 driver 需要自己應對 endian 差異的，可以用事先定義好的巨集來交換 byte 順序，這些巨集定義在「include/linux/byteorder/generic.h」標頭檔內：

htonl()：將 4 bytes 資料從 host endian 轉成 network byte order(big endian)。

ntohl()：將 4 bytes 資料從 network byte order(big endian) 轉成 host endian。

htons()：將 2 bytes 資料從 host endian 轉成 network byte order(big endian)。

ntohs()：將 2 bytes 資料從 network byte order(big endian) 轉成 host endian。

little endian:

int i = 0x12345678

Copy i to char array[4]:

array[0]: 0x7fff929d1c70 = 0x78

array[1]: 0x7fff929d1c71 = 0x56

array[2]: 0x7fff929d1c72 = 0x34

array[3]: 0x7fff929d1c73 = 0x12

5-4、對齊

Page 邊界

有些硬體要求指標一定要指在 page 的開頭，這時 driver 就必須對齊邊界：

新指標 = (記憶體指標 + PAGE_SIZE -1) & ~(PAGE_SIZE-1)

struct 的成員邊界

5-5、連結串列

Stack(堆疊) 類型的 Linked List

Queue(佇列) 類型的 Linked List

5-6、虛擬記憶體

driver 的開發需要瞭解 OS 是如何管理記憶體的，因為 driver 需要與一般應用程式(user space) 交換資料，所以必須知道 kernel space 與 user space 的記憶體有什麼差別才行。

實體記憶體

電腦安裝的記憶體模組稱為「實體記憶體」(physical memory;或稱「物理記憶體」)。
有一些 OS 會直接在物理記憶體內執行 kernel 與一般應用程式。

虛擬記憶體

隨著應用程式功能不斷擴充，直接在物理記憶體上執行應用程式越來越容易影響系統穩定，因此「虛擬記憶體」(Virtual Memory) 的想法就誕生了。

在應用程式載入記憶體時，把它配置在虛擬記憶體內，如此，應用程式執行時，就不會直接影響到物理記憶體了。
Process 在讀寫記憶體時，CPU 會參考 page table 將虛擬位址轉換成物理位址，接著讀寫物理記憶體空間。

如果物理記憶體已經沒有空間的話，將沒辦法再分配記憶體來使用，為了應對這情況，linux 引進了「swap」的觀念。
要為新的 process 分配虛扯記憶體空間時，先將物理記憶體內沒用到的資料移到硬碟內(swap out)，等到舊 process 開始再次執行時，再將硬碟裡的資料讀回物理記憶體(swap in)。

driver 之間的通訊

driver 要呼叫 kernel 之中其它 driver 提供的函式、或是讀寫相關變數時，可以直接透過指標進行。
Kernel space 之間的記憶體空間是共用的，可以把 kernel 想成一個獨立的 process 來看待。
在 Linux kernel 之內想複製記憶體內容時，也可以用 ANSI C 提供的 memcy() 函式，其相關函式都實作在「lib/string.c」檔案裡。

User Process 與 driver 的通訊

User process 可透過 read() 與 write() 系統呼叫與 driver 交換資料。
User process 與 kernel 分別在不同的(虛擬)記憶體空間內運作，因此兩者之間無法直接讀寫記憶體。
IA-32版的 Linux 在 user context 之內，是把 4GB 虛擬記憶體空間的下半部 3GB 分配給 user process 使用，上半部 1GB 給 kernel 使用，如此就能從 kernel 直接讀寫 user process 的記憶體空間了，但在讀寫之前，須確認目的地：
是不是合法位址
有沒有被 swap out
因此，在 user process 與驅動程式之間搬移資料的時候，光呼叫 memcpy() 是不夠的。
所以，Linux kernel 準備了在 user process 與驅動程式之間搬移資料的 kernel 函式與巨集，讓裝置驅動程式使用：

```
int get_usr(x, ptr)、int put_usr(x, ptr)
int access_ok(int type, void *addr, unsigned long size)
unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);
unsigned long copy_to_user(void *to, const void __user *from, unsigned long n);
unsigned long clear_user(void __user *to, unsigned long n)
```

使用這些函式時，需要 include「asm/uaccess.h」
這些函式都可能 sleep(在 user process 的記憶體被 swap out 的時候)，所以只能在 user context 使用。

5-7、交換資料

說明 user process 與 character 類型的裝置驅動程式交換資料的方法。

device file 的角色

User process 要向 driver 送資料、或是從 driver 取得資料時，需要有「某種東西」把 user process 與 driver 連起來。
扮演這個角色的就是「device file」以及「device special file」。

裝置檔通常放在 /dev 目錄內。
原則上一個裝置就要準備一個 device file。
前面說是「特殊」的檔案，用 ls 就能看出與一般檔案有點差異，最左邊的字元代表裝置檔的種類：
ls -l /dev/ttyS0
b：區塊裝置(block device)
c 或 u：字元裝置(character/unbuffered device)
p：具名管線，或稱 FIFO(named pipe)
平常顯示檔案大小的地方變成兩個數值，分別是 major number 及 minor number。

建立與刪除 device file

裝置檔可透過 mknod 指令建立。
刪除的時候，直接用 rm 即可。
目前 Linux 的 major number 是 12-bit，minor number 是 20-bit，Major/minor number 可以使用的 bit 數定義在「/include/linux/kdev_t.h」裡。

Major / Minor Number 扮演的角色

User process 打開裝置檔時，kernel 看到開檔的是系統呼叫，就算 user process 是呼叫 fopen() 這類函式庫函式，最終還是由 glibc 對 kernel 送出 open() 系統呼叫。
Kernel 的 open() 系統呼叫介面是 sys_open() 函式，但是只知道開啟 device file 的話，就不知道要找哪個驅動程式處理了。
這邊就要用到「major number」，驅動程式載入時，可以向 kernel 登記 major number，每個驅動程式都有獨一無二的 major number。
Major number 與指向驅動程式的指標存放在 kobj_map 對應表內。
Kernel 的 sys_open() 函式會在 user process 開啟裝置檔時得知 major number，然後由此查詢 kobj_map，如此，透過 major number 即可把裝置檔與對應的驅動程式結合起來。
Minor number 是由驅動程式自己管理的編號。
驅動程式在同時控制許多個裝置時，通常會用 minor number 來區分各個裝置。
Major/minor number 在系統內必須獨一無二，所以 Linux kernel 在原始檔的「Documentation/devices.txt」列出所有 major/minor number。
但現在的主流是動態分配 major number。
Linux kernel 是以「dev_t」這個型別來表現 major/minor number，它被 typedef 成 unsigned int：
typedef __u32 __kernel_dev_t; typedef __kernel_dev_t dev_t;
想從 dev_t 變數取出 major / minor number 時，可以使用 MAJOR() 與 MINOR() 巨集。
想結合 major / minor number 建立 dev_t 時，則是用 MKDEV() 巨集。

5-8、User Process 與 driver 的通訊管道

User process 以 open() 系統呼叫打開 device file 之後，可以：
透過 write() 系統呼叫將資料傳給驅動程式。
透過 read() 系統呼叫從驅動程式取得資料。
處理完畢後，再以 close() 系統呼叫關閉 device file。
除了上述幾個系統呼叫外，還有下面這些可以用：
seek()
poll() 或 select()
ioctl()
mmap()
fcntl()
但如果驅動程式沒有為這些系統呼叫準備對應的 handler，user process 就無法使用。

5-9、Major Number 的登記方式

為了將 device file 與驅動程式連起來，驅動程式必須向 kernel 登記「major number」，先前 major number 是靜態的，最近改採動態管理方式為主流。

靜態(傳統)登記法

靜態登記為 Linux 2.4 的標準作法， Linux 2.6 已不推薦這個作法。
是以使用 register_chrdev() 登記：

```
int register_chrdev(unsigned int major, const char *name, const struct file_operations *fops);
```

這個函式是在「fs/char_dev.c」檔案內實作的。
major 引數是要使用的 major number，name 引數則是驅動程式名稱，fops 引數則是驅動程式提供的系統呼叫處理介面。
登錄成功後，就會在 /proc/devices 顯示驅動程式名稱及 major number。
major 引數也可傳入「0」，如此 kernel 會自動分配一個還沒用到的 major number。
卸載驅動程式時，要用 unregister_chrdev() 函式跟 kernel 刪除 major number。

```
void unregister_chrdev(unsigned int major, const char *name);
```

第一個引數(major) 是 register_chrdev() 函式分配的「major number」。
第二個引數(name) 是驅動程式的名稱。

動態登記法

1、以 alloc_chrdev_region() 動態取得 major number。
2、以 cdev_init() 登記系統呼叫 handler。
3、以 cdev_add() 向 kernel 登記驅動程式。

在卸載驅動程式時，則依相反步驟解除登記：
1、以 cdev_del() 向 kernel 釋放驅動程式。
2、以 unregister_chrdev_region() 釋放 major number。
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name);
alloc_chrdev_region() 是依驅動程式名稱用來取得 major number，並從指定的起點開始預留指定數目的 minor number。
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
cdev_init() 負責初始化 cdev 結構，並登記系統呼叫 handler(fops)，另外，其 cdev 結果變數在卸除驅動程式時還要用到，需定義為全域變數。
在呼叫 cdev_init() 後，還要將 cdev 結構變數的 owner 成員設為「THIS_MODULE」。
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
cdev_add 向 kernel 登記 cdev_init() 設定好的裝置資訊。
void dcdev_del(struct cdev *p)
cdev_del() 的功能與 cdev_add() 相反，會從 kernel 釋放驅動程式。
void unregister_chrdev_region(dev_t from, unsigned count)
unregister_chrdev_region() 的功能跟 alloc_chrdev_region() 相反，會釋放之前拿到的 major number。

5-10、系統呼叫處理函式的登記方式

說明系統呼叫 handler 的相關細節。

驅動程式的 handler

想讓 user process 能對驅動程式發出 open() 與 read() 之類系統呼叫的話，驅動程式必須先向 kernel 登記相應的 handler 才行。
驅程程式的 handler 是透過 file_operations 結構指定的，其宣告在 include/linux/fs.h 內。
C 語言在初始化 struct 的時候需要記得成員的定義位置，但如果用上 gcc 擴充語法的話，就不必記得這些位置，比如說，可以寫成下面這個樣子：

```
struct file_operations devone_fops = {
.open = devone_open,
.release = devone_close, }
```

這樣就會將 open 及 release 成員指定為驅動程式提供的函式指標，並自動把其它成員設為 NULL。

open handler

User process 在操作 device file 時，第一個動作一定是「開啟」，結束操作之後則會將它「關閉」。
因此，驅動程式至少要提供 open 與 close 這兩個 handler。
int (*open) (struct inode *inode, struct file *file);
inode 引數是內含「inode」資料的結構指標，開發驅動程式時會用到的成員如下：
bdev_t i_rdev： Major/minor number
void *i_private： 驅動程式私有指標
驅動程式可以透過 i_rdev 成員判別驅動程式內部使用的 minor number，想改變控制硬體對象時，可以使用。
取出 minor number 的方法有兩種，如下：
ret = MINOR(inode->i_rdev);
ret = iminor(inode)
考慮到可攜性，還是使用 iminor() 比較好，想得到 major number，亦是使用同樣的方法：
unsigned iminor(const struct inode *inode); unsigned imajor(const struct inode *inode);
i_private 成員是驅動程式可以自由使用的指標。
file 也是個巨大的資料結構，開發驅動程式常用到的成員如下：
struct file_operations *f_op：系統呼叫 handlers
unsigned int f_flags：open 函式第二引數傳入的旗標
void *private_data：驅動程式私有資料指標
f_op 成員是驅動程式載入時，登記的系統呼叫 handles 指標照對表。
f_flags 成員是 user process 呼叫 open() 時指定的旗標。
private_data 成員是驅動程式可以自由使用的指標。

close Handler
close handler 的 prototype 如下，這邊要注意它的成員名稱是「release」：
int (*release) (struct inode *inode, struct file *file);
引數與 open handler 一樣，被開啟時如果有配置資源的話，一定要在 close handler 之內釋放掉，否則會導致 memory leak 的問題，只要 OS 沒有重開，就無法釋放資源。
就算 user process 忘記明確呼叫 close()，OS 也會在 user process 結束的時候呼叫 close()。
一個或多個 user process 可能同時重複開啟同一個 device file，想在 close handler 被呼叫時，得知目前是對應哪個 open() 開啟的話，可以透過 inode 或 file 結構的驅動程式私有指標來判斷，下個主題會談到。

inode 結構與 file 結構的關係
open 與 close handler 都會從 kernel 接到 inode 結構與 file 結構的指標，而這兩個指標都有給驅動程式的私用指標。
inode 結構在 mknod 指令建立 device file 時由 kernel 建立，而 file 結構是每次開檔時由 kernel 建立。

也就是說，驅動程式想區分 open 系統呼叫的話，應該透過 file 結構的驅動程式私有指標來進行，open handler 配置資源時將之給「file->private_data」，close handler 釋放資源時也是從「file->private_data」將之釋放。
inode 結構的驅動程式私有指標，可以在存放 device file 相關資訊時使用。
父 process 透過 fork() 系統呼叫產生子 process 時，檔案 handle 會被共用，所以 file 結構只會有一個，但此時因為 file 會被兩個 process 共同使用，所以 reference count 改成「2」，在這兩個 process 呼叫 close() 時，reference count 都會減一，直到 reference 變成「0」時，才會呼叫驅動程式的 close handler，之後 kernel 會釋放不再需要用到的 file 結構。

使用 open 與 close 的範例程式

這隻驅動程式會登記 major number，只支援 open 與 close handler。
devone.c:

```
#include
#include
#include
#include
#include
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

#define DRIVER_NAME "devone"

static unsigned int devone_major = 0;
module_param(devone_major, uint, 0);

static int devone_open(struct inode *inode, struct file *file) {
printk("%s: major %d minor %d (pid %d)\n", __func__, imajor(inode), iminor(inode),
current->pid);
```

```
inode->i_private = inode;
file->private_data = file;

printk("i_private=%p private_data=%p\n", inode->i_private, file->private_data);

return 0;
}

static int devone_close(struct inode *inode, struct file *file) {
printk("%s: major %d minor %d (pid %d)\n", __func__, imajor(inode), iminor(inode),
current->pid);
printk("i_private=%p private_data=%p\n", inode->i_private, file->private_data);

return 0;
}

struct file_operations devone_fops = {
.open = devone_open,
.release = devone_close,
};

static int devone_init(void) {
int major;
int ret = 0;

major = register_chrdev(devone_major, DRIVER_NAME, &devone_fops);
if ((devone_major > 0 && major != 0) || // static allocation
(devone_major == 0 && major < 0) || // dynamic allocation
major < 0) // else
{
printk("%s driver registeration error\n", DRIVER_NAME);
ret = major;
```

```
goto error;
}
if (devone_major == 0) {
devone_major = major;
}
printk("%s driver(major %d) installed.\n", DRIVER_NAME, devone_major);
error:
return (ret);
}

static void devone_exit(void) {
unregister_chrdev(devone_major, DRIVER_NAME);
printk("%s driver removed.\n", DRIVER_NAME);
}

module_init(devone_init);
module_exit(devone_exit);
```

Major number 是以 register_chrdev() 登記的，第一個引數傳入0的話，就會動態分配，但是為了在 insmod 時可以透過引數指定，所以用了 module_param 巨集：

```
static unsigned int devone_major = 0;
module_param(devone_major, uint, 0);
```

module_param 巨集的第二引數是變數的資料型別，用了這個巨集後，就可以下列方式透過引數改變全域變數的初始值：

```
/sbin/insmod ./sample.ko devone_major=261
```

open 與 close handler 是在 process context 內運作的，所以可以透過 current 巨集取得呼叫 open 或 close 的 process 資訊，使用 current 巨集，要引入兩個標頭檔：

```
#include
#include
```

open handler 會把 inode 結構與 file 結構的指標分別放到各自驅動程式私有成員內。

Makefile:

```
CFLAGS += -Wall
CFILES := devone.c
```

```
obj-m += sample.o
sample-objs := $(CFILES:.c=.o)

all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

make 之後得到 sample.ko，接著載入驅動程式，瀏覽「/proc/devices」可以看到驅動程式分配的 major number，再用這個號碼透過 mknod 建立 device file：

```
sudo /sbin/insmod ./sample.ko
/sbin/lsmod | grep sample
cat /proc/devices | grep devone
sudo /bin/mknod /dev/devone c 250 0
ls -l /dev/devone
```

可以看到 device file 的最初權限是「644」，如果要讓一般使用者也可以使用，就必須修改權限：

```
sudo /bin/insmod --mode=666 /dev/devone c `grep devone /proc/devices | awk '{print $1}'` 0
```

接著是 user process 的應用程式，只會打開、接著關閉 device file。

simple.c

```
#include
#include
#include
#include
#include
#include
#include
#include
#include

#define DEVFILE "/dev/devone"
```

```
int open_file(void) {
int fd;

fd = open(DEVFILE, O_RDWR);
if (fd == -1) {
perror("open");
}
return fd;
}

int close_file(int fd) {
if (close(fd) != 0) {
perror("close");
}
}

int main(void) {
int fd;
fd = open_file();
sleep(20);
close_file(fd);
return 0;
}
```

連續執行這個程式兩次，kernel buffer 會顯示驅動程式的訊息，可以發現 inode 結構的指標都是同一個，但 file 結果指標則不同。

接著透過 fork() 複製 process 看看，子 process 關閉之前打開的 file handle，父 process 也會關閉 file handle。

fork.c

```
#include
#include
#include
```

```
#include
#include
#include
#include
#include
#include

#define DEVFILE "/dev/devone"

int open_file(void) {
int fd;

fd = open(DEVFILE, O_RDWR);
if (fd == -1) {
perror("open");
}
return fd;
}

int close_file(int fd) {
printf("%s called\n", __func__);
if (close(fd) != 0) {
perror("close");
}
}

int main(void) {
int fd;
int status;

fd = open_file();
```

```
if (fork() == 0) { // child process
sleep(3);
close_file(fd);
exit(1);
}
wait(&status);
sleep(10);
close_file(fd);
return 0;
}
```

編譯執行後，比對 kernel buffer 的訊息，可以發現最初 close() 時，不會呼叫驅動程式的 close handler。
使用 register_chrdev() 時，驅動程式可以自由使用 minor number，就算 mknod隨意指定 minor number，user process 還是可以開啟成功，但是可以用到的 minor number 範圍僅限 0 ~ 255。
使用 cdev_add() 時，驅動程式需要明確指出想使用的 minor number 範圍。

使用 open 與 close 的範例程式 - cdev_add() [kernel 2.6 推薦]

把上一節的程式改寫成使用 cdev_add() 的形式，不同的地方只有驅動程式的載入、卸除部分，open 與 close handler 直接延用：
devone.c:

```
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

#define DRIVER_NAME "devone"
```

```
static int devone_devs=2;
static unsigned int devone_major = 0;
module_param(devone_major, uint, 0);
static struct cdev devone_cdev;

static int devone_open(struct inode *inode, struct file *file) {
printk("%s: major %d minor %d (pid %d)\n", __func__, imajor(inode), iminor(inode),
current->pid);

inode->i_private = inode;
file->private_data = file;

printk("i_private=%p private_data=%p\n", inode->i_private, file->private_data);

return 0;
}

static int devone_close(struct inode *inode, struct file *file) {
printk("%s: major %d minor %d (pid %d)\n", __func__, imajor(inode), iminor(inode),
current->pid);
printk("i_private=%p private_data=%p\n", inode->i_private, file->private_data);

return 0;
}

struct file_operations devone_fops = {
.open = devone_open,
.release = devone_close,
};

static int devone_init(void) {
```

```
dev_t dev = MKDEV(devone_major, 0);
int alloc_ret = 0;
int major;
int cdev_err = 0;

alloc_ret = alloc_chrdev_region(&dev, 0, devone_devs, DRIVER_NAME);
if (alloc_ret)
goto error;
devone_major = major = MAJOR(dev);

cdev_init(&devone_cdev, &devone_fops);
devone_cdev.owner = THIS_MODULE;

cdev_err = cdev_add(&devone_cdev, MKDEV(devone_major, 0), devone_devs);
if (cdev_err)
goto error;

printk(KERN_ALERT "%s driver(major %d) installed.\n", DRIVER_NAME, major);
return 0;
error:
if (cdev_err == 0)
cdev_del(&devone_cdev);
if (alloc_ret == 0)
unregister_chrdev_region(dev, devone_devs);
return -1;
}

static void devone_exit(void) {
dev_t dev = MKDEV(devone_major, 0);
cdev_del(&devone_cdev);
unregister_chrdev_region(dev, devone_devs);
printk("%s driver removed.\n", DRIVER_NAME);
```

```
}

module_init(devone_init);
module_exit(devone_exit);
```

Makefile 同上例

devone_devs 全域變數為使用的 minor number 個數，此程式以 2 為初始值，所以只有「0」與「1」這兩個 minor number 可以使用。
即使空用 mknod 建立了 minor number 0 ~ 3 的裝置檔，user process 也只能 open 「0」或「1」的裝置檔，對於「2」及「3」的裝置檔會產生 No such device or address 的錯誤訊息。
user process 的範例程式碼如下(simple.c)：

```
#include
#include
#include
#include
#include
#include
#include
#include
#include

#define DEVFILE "/dev/devone"
#define DEVCOUNT 4

int open_file(char *filename) {
int fd;

fd = open(filename, O_RDWR);
if (fd == -1) {
perror("open");
}
```

```
return fd;
}

int close_file(int fd) {
if (close(fd) != 0) {
perror("close");
}
}

int main(void) {
int fd[DEVCOUNT];
int i;
char file[BUFSIZ];

for (i=0; i snprintf(file, sizeof(file), "%s%d", DEVFILE, i);
printf("%s\n", file);
fd[i] = open_file(file);
}
sleep(3);

for (i=0; i printf("closing fd[%d]\n", i);
close_file(fd[i]);
}
return 0;
}
```

read Handler 與 write Handler

這邊要介紹 read 與 write handler 的實作方式：

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

只要這樣定義 handler，驅動程式就能把資料傳給 user process。

filp 引數指向開啟裝置檔時 kernel 建立的 file 結構，跟 open 收到的指標是同一個，因此，open handler 設給「filp->private_data」成員的指標，在 read handler 裡也可以拿來用。

buf 引數是 user process 呼叫 read() 時指定的緩衝區指標，驅動程式無法直接取用 buf 指標，必須透過 copy_to_user() 這個 kernel 提供的函式把資料複製過去，原因是之前所提有關「kernel space」與「user space」的不同。
count 引數是 user process 呼叫 read() 時提供的緩衝區空間。
f_pos 引數是 offset，驅動程式可以視需要將之更新。

ssize_t *write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos);
write handler 的 prototype 與 read handler 大致相同。
write handler 是在 user process 要傳資料給驅動程式時使用的，驅動程式需透過 kernel 提供的 copy_from_user() 函式從 buf 引數讀入資料，緩衝區的大小是 count bytes。
用 read 與 write 寫個驅動程式測試看看，User Process 在一開始在讀檔時，會全讀到 0xff，之後再寫入 1 byte 資料，再讀出寫入後的資料。
devone.c

```
#include
#include
#include
#include
#include
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

#define DRIVER_NAME "devone"

static int devone_devs=1;
static unsigned int devone_major = 0;
module_param(devone_major, uint, 0);
static struct cdev devone_cdev;

struct devone_data {
unsigned char val;
```

```
rwlock_t lock;
};

ssize_t devone_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos) {
struct devone_data *p = filp->private_data;
unsigned char val;
int retval = 0;

printk("%s: count %d pos %11d\n", __func__, count, *f_pos);

if (count >= 1) {
if (copy_from_user(&val, &buf[0], 1)) {
retval = -EFAULT;
goto out;
}

write_lock(&p->lock);
p->val = val;
write_unlock(&p->lock);
retval = count;
}
out:
return (retval);
}

ssize_t devone_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos) {
struct devone_data *p = filp->private_data;
int i;
unsigned char val;
int retval = 0;

read_lock(&p->lock);
```

```
val = p->val;
read_unlock(&p->lock);

printk("%s: count %d pos %11d\n", __func__, count, *f_pos);

for ( i=0; i if (copy_to_user(&buf[i], &val, 1)) {
retval = -EFAULT;
goto out;
}
}
retval = count;
out:
return (retval);
}

static int devone_open(struct inode *inode, struct file *file) {
struct devone_data *p;
printk("%s: major %d minor %d (pid %d)\n", __func__, imajor(inode), iminor(inode),
current->pid);

p = kmalloc(sizeof(struct devone_data), GFP_KERNEL);
if (p == NULL) {
printk("%s: No memory\n", __func__);
return -ENOMEM;
}

p->val = 0xff;
rwlock_init(&p->lock);

file->private_data = p;

return 0;
```

```
}

static int devone_close(struct inode *inode, struct file *file) {
printk("%s: major %d minor %d (pid %d)\n", __func__, imajor(inode), iminor(inode),
current->pid);

if (file->private_data) {
kfree(file->private_data);
file->private_data = NULL;
}
return 0;
}

struct file_operations devone_fops = {
.open = devone_open,
.release = devone_close,
.read = devone_read,
.write = devone_write,
};

static int devone_init(void) {
dev_t dev = MKDEV(devone_major, 0);
int alloc_ret = 0;
int major;
int cdev_err = 0;

alloc_ret = alloc_chrdev_region(&dev, 0, devone_devs, DRIVER_NAME);
if (alloc_ret)
goto error;
devone_major = major = MAJOR(dev);

cdev_init(&devone_cdev, &devone_fops);
```

```
devone_cdev.owner = THIS_MODULE;

cdev_err = cdev_add(&devone_cdev, MKDEV(devone_major, 0), devone_devs);
if (cdev_err)
goto error;

printk(KERN_ALERT "%s driver(major %d) installed.\n", DRIVER_NAME, major);
return 0;
error:
if (cdev_err == 0)
cdev_del(&devone_cdev);
if (alloc_ret == 0)
unregister_chrdev_region(dev, devone_devs);
return -1;
}

static void devone_exit(void) {
dev_t dev = MKDEV(devone_major, 0);
cdev_del(&devone_cdev);
unregister_chrdev_region(dev, devone_devs);
printk("%s driver removed.\n", DRIVER_NAME);
}

module_init(devone_init);
module_exit(devone_exit);
```

Makefile 同上例

user process program
sample.c

```
#include
#include
#include
```

```
#include
#include
#include
#include
#include
#include

#define DEVFILE "/dev/devone"

int open_file(char *filename) {
int fd;

fd = open(filename, O_RDWR);
if (fd == -1) {
perror("open");
}
return fd;
}

int close_file(int fd) {
if (close(fd) != 0) {
perror("close");
}
}

void read_file(int fd) {
unsigned char buf[8], *p;
ssize_t ret;

ret = read(fd, buf, sizeof(buf));
if (ret > 0) {
p = buf;
```

```
while (ret--)
printf("%02x", *p++);
} else {
perror("read");
}
printf("\n");
}

void write_file(int fd, unsigned char val) {
ssize_t ret;

ret = write(fd, &val, 1);
if (ret <= 0) {
perror("write");
}
}

int main(void) {
int fd;
int i;

for (i=0; i<2; ++i) {
printf("NO. %d\n", i+1);
fd = open_file(DEVFILE);

read_file(fd);
write_file(fd, 0x00);
read_file(fd);
write_file(fd, 0xC0);
read_file(fd);
close_file(fd);
}
```

```
return 0;
}
```

此範此的精髓在於 struct devone_data 的應用，包含記憶體的配置、指檔的運用…等。
open handler 及 close handler 因為使用了 BKL(Big Kernel Lock)，所以目前的版本無法同時執行多個呼叫。
read handler 與 write handler 會從 file 結構參考 private_date 成員，從而得到數值。
不管是 read handler 或 write handler 都可能同時呼叫多次，所以需要寫成 reentrant(可重進入) 的形式，有需要的話，就得用上 semaphore 做鎖定。
此範例透過 reader/writer lock 來控制 devone_data 結構成員的修改動作，就是為了預防同時執行 p->val = val; 時會發生衝突的問題。

透過 Minor Number 切換 Handler

Minor number 是驅動程式自己管理的，如果希望依不同的 minor number 提供不同功能時，若在 handler 內以 if 判斷 minor number 的話，會寫出不易閱讀的套疊程式碼。
open handler 會拿到 file 結構指標，如果能在驅動程式這邊更換 f_op 成員的話，也能依照 minor number 切換 handler，程式碼較為易讀。

```
static int devone_open(struct inode *inode, struct file *file) {
swtich (iminor(inode)) {
case 0:
file->f_op = &zero_fops;
break;

case 1:
file->f_op = &one_fops;
break;
default:
return -ENXIO;
}

if (file->f_op && file->f_op->open)
return file->f_op->open(inode, file);

return 0;
}
```

其中 zero_fops, one_fops 是 struct file_operations 的變數，各自定義自己的 handler。

5-11、udev

將 character 或 block 類型的裝置驅動程式載入 kernel 後，要在 /dev 目錄下建立對應的裝置檔，隨著硬體種類不斷增加，這個方法的破綻也逐漸顯現。
因此出現了希望在 OS 偵測到新硬體時，可以自動建立裝置檔，透過 hotplug 之類的機制，也希望能做到在移除裝置時，自動刪除裝置檔。
/dev 是在 RAM 裡面的檔案系統，所以用 mknod 建立的裝置檔，重開機之後就會消失。
在 OS 啟動時，因為要動態產生 /dev 內容的關係，所以需要稍花時間，kernel 在進入 runlevel 3 之後的「Starting udev」就是建立裝置檔的地方。
在 init daemon 開始執行之前，/dev 是磁碟上的目錄，要在 RAM 裡頭建立 udev 之前，須先 unmount 才行。
啟動時 init daemon 的 rc script(/etc/rc.d/rc.sysinit) 會呼叫「/sbin/start_udev」script 建立裝置檔。
為了讓 udev 能夠執行，/dev 目錄下還是要有一些最基本的裝置檔，這些裝置檔要用 MAKEDEV 手動建立。
手動建立的裝置檔清單可在 /etc/udev/makedev.d 目錄內找到。
啟動 OS 後載入驅動程式的話，會對系統新增機置，udevd daemon 會偵側到這個事件，而後去檢查 /sys 目錄。
如果驅動程式建立了「dev」標案的話，檔案裡會有 major/minor number，如此 udevd 就能以它建立裝置檔。
建立裝置檔的規則設定位於 /etc/udev/rules.d/ 內。

驅動程式的 udev 支援

要讓驅動程式支援 udev 的話，必須登錄驅動程式的 class，並在 /sys/class 目錄下建立驅動程式資訊，此外，還要配合需求提供規則檔。
舉例要新建「/sys/class/devone/devone0/dev」。
首先是登記 class，登記 class 是使用 class_create() 這個 kernel 函式，其 prototype 定義在 linux/device.h 內：

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

第二引數 name 用來指定 class 的名稱，按上面的例子來說，就是要指定成 "devone"。
函式呼叫的成功與否，可透過 IS_ERR() 來判斷，不能直接判斷是否為 NULL。
卸載驅程程式時，必須一併刪除先前登記的 class，這時用的是 class_destroy()：

```
void class_destroy(struct class *cls);
```

建立完 /sys/class/devone 資料夾後，接著要建立「/sys/class/devone/devone0」這個裝置名稱，用的是 class_device_create()：

```
struct device *device_create(struct class *cls, struct class_device *parent, dev_t devt,
struct device *device, const char *fmt, ...);
```
第一引數 cls 是 class_create() 傳回的 class。
第二引數 parent 是指定上層 class 時使用的，也可以指定成 NULL。
第三引數 devt 是要在 dev 檔顯示的 major/minor number，用 MKDEV 巨集指定即可。
第四引數 device 在想為 class 驅動程式連結「struct device」時可以指定，傳入 NULL 也可。
第五引數 fmt 是裝置名稱，以上例來說就是 devone0 ，可以使用可變引數。
卸載驅動程式時，須刪除先前登記的裝置名，可以使用 class_device_destroy()：
```
void class_device_destroy(struct class *cls, dev_t devt);
```

支援 udev 的驅動程式範例：
devone.c
```
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static int devone_devs = 1;
static unsigned int devone_major = 0;
static unsigned int devone_minor = 0;
static struct cdev devone_cdev;
static struct class *devone_class = NULL;
static dev_t devone_dev;

ssize_t devone_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos) {
```

```
int i;
unsigned char val = 0xff;
int retval;

for ( i=0; i if (copy_to_user(&buf[i], &val, 1)) {
retval = -EFAULT;
goto out;
}
}
retval = count;
out:
return (retval);
}

struct file_operations devone_fops = {
.read = devone_read,
};

static int devone_init(void) {
dev_t dev = MKDEV(devone_major, 0);
int alloc_ret = 0;
int major;
int cdev_err = 0;
struct device *class_dev = NULL;

alloc_ret = alloc_chrdev_region(&dev, 0, devone_devs, "devone");
if (alloc_ret)
goto error;
devone_major = major = MAJOR(dev);

cdev_init(&devone_cdev, &devone_fops);
devone_cdev.owner = THIS_MODULE;
```

```
devone_cdev.ops = &devone_fops;
cdev_err = cdev_add(&devone_cdev, MKDEV(devone_major, devone_minor), 1);
if (cdev_err)
goto error;

devone_class = class_create(THIS_MODULE, "devone");
if (IS_ERR(devone_class)) {
goto error;
}
devone_dev = MKDEV(devone_major, devone_minor);
class_dev = device_create(devone_class, NULL, devone_dev, NULL, "devone%d",
devone_minor);

printk(KERN_ALERT "devone driver(major %d) installed.\n", major);
return 0;
error:
if (cdev_err == 0)
cdev_del(&devone_cdev);
if (alloc_ret == 0)
unregister_chrdev_region(dev, devone_devs);
return -1;
}

static void devone_exit(void) {
dev_t dev = MKDEV(devone_major, 0);

class_device_destroy(devone_class, devone_dev);
class_destroy(devone_class);

cdev_del(&devone_cdev);
unregister_chrdev_region(dev, devone_devs);
```

```
printk(KERN_ALERT "devone driver removed.\n");
}

module_init(devone_init);
module_exit(devone_exit);
```

範例的規則檔：
/etc/udev/rules.d/51-devone.rules
```
KERNEL=="devone[0-9]*", GROUP="root", MODE="0644"
```
這邊設定的權限讓一般使用者也能打開裝置檔。
ps. 1, 檔名似乎是隨意取的，只要不與同資料夾的其它檔案產生衝突即可。
2, 試著移除後，發現以 root 的權限還是能打開裝置檔，但以 user 的權限就不行。

Makefile 同上例
編譯、測試過程：
```
insmod ./sample.ko
lsmod | grep sample
dmesg | tail
cat /proc/devices | grep devone
ls -l /dev/devone0
ls -l /sys/class/devone/devone0/
cat /sys/class/devone/devone0/dev
hexdump -C -v -n 32 /dev/devone0
rmmod sample
ls -l /dev/devone*
```

5-12、結語
驅動程式多半要與 user process 通訊才有辦法實現功能，而其通訊方式多半遵循 UNIX 的傳統手法，透過檔案來進行。

六、實際撰寫驅動程式
延續上一章的驅動程式基礎知識，繼續介紹實際應用技巧。

6-1、IOCTL

驅動程式準備了 read 與 write 介面就能與 user process 交換資料，但仍有許多事情辦不到，例如，控制硬體暫存器、或是改變驅動程式本身運作模式的情形等。
Linux 為了實現這些無法透過 read 或 write 完成的「資料交換」工作，另外準備「IOCTL」介面。

IOCTL 的機制

IOCTL(I/O Control) 是一種系統呼叫，user process 呼叫 ioctl() 即可對驅動程式送出系統呼叫，而呼叫到驅動程式的 IOCTL 處理函式。
與 read() 跟 write() 一樣，可跟驅動程式交換資料，但 IOCTL 交換資料的格式可由驅動程式開發者自行決定，應用較為靈活。

user process 呼叫 ioctl() 的 prototype：

```
int ioctl(int d, int request, ...);
```

第一引數(d) 是裝置置的 handle。
第二引數(request) 是驅動程式定義的指令代碼(32 bit)，一般透過驅動程式的巨集指定。
第三引數(…) 代表可變引數。
要支援 ioctl()，驅動程式必須提供 IOCTL 方法，並將 file_operations 結構的 ioctl 成員設定函數指標：

```
int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
```

第一引數(inode) 是 user process 開啟的裝置檔相關資訊。
第二引數(filp) 通常在取出驅動程式私有資料時，會透過 filp 進行。
第三引數(cmd) 是 IOCTL 的指令，透過此引數，驅動程式可以判斷 user process 究竟想做什麼。
第四引數(arg) 相當於 ioctl() 可變引數(...) 的參數，內含 user process 指標。
ioctl() 是 user process 呼叫的，所以驅動程式的 IOCTL 方法是在「user context」底下運行，因此可以 sleep，若當 IOCTL 方法內有 critical section 時，可以使用「semaphore」來鎖定。

IOCTL 指令

IOCTL 指令是 unsigned int 型別的變數，它的格式是固定的，以巨集定義。
| dir(2bit) | Size(14bit) | type(8bit) | nr(8bit) |
IOCTL 指令格式定義在 include/asm-generic/ioctl.h。
這些巨集可以用來在標頭檔下定義，讓驅動程式及 user process 可以 include 進來共用。
dev_ioctl.h

```
#ifndef _DEVONE_IOCTL_H
#define _DEVONE_IOCTL_H
```

```
#include

struct ioctl_cmd {
unsigned int reg;
unsigned int offset;
unsigned int val;
};

#define IOC_MAGIC 'd'

#define IOCTL_VALSET _IOW(IOC_MAGIC, 1, struct ioctl_cmd)
#define IOCTL_VALGET _IOR(IOC_MAGIC, 2, struct ioctl_cmd)

#endif
```

此例的 IOCTL_VALSET 巨集是用來傳資料給驅動程式， IOCTL_GET 則是從驅動程式讀回資料。
巨集的 type 引數長度為 1byte，所以通常用字元常數指定，驅動程式名稱的第一個字元是很常見的選擇。
引數 nr 是數值(number) 的意思，通常為 IOCTL 指令連續編號。
引數 size 是傳給 ioctl() 可變引數(...) 的介面型別。
驅動程式可以使用 _IOC_SIZE 巨集得知傳給 ioctl() 可變引數的結構大小。

```
#define _IOC_SIZE(nr) (((nr) >> _IOC_SIZESHIFT) & _IOC_SIZEMASK)
```

在讀寫 user process 指標時，可以先呼叫 access_ok() 巨集判斷指標可否讀寫。

```
#define access_ok(type, addr, size) (likely(__range_ok(addr, size) == 0))
```

權限

IOCTL 指令如果要限制只有 root 可以使用，此時 capable() 就很方便：

```
int capable(int cap);
```

這個函式的引數可以指定的巨集，定義在 include/linux/capability.h 內。
要檢查是否擁有 root 權限，可以寫成：

```
if (!capable(CAP_SYS_ADMIN)) { retval = -EPERM goto done; }
```

範例程式

使用 IOCTL 的範例驅動程式，read() 傳回的資料可透過 ioctl() 改變，但需要 root 的權限才能修改。
而驅動程式內的資料讀寫動作，也透過讀寫鎖定(read-write lock)來保護。

dev_ioctl.h

```
#ifndef _DEVONE_IOCTL_H
#define _DEVONE_IOCTL_H

#include

struct ioctl_cmd {
unsigned int reg;
unsigned int offset;
unsigned int val;
};

#define IOC_MAGIC 'd'

#define IOCTL_VALSET _IOW(IOC_MAGIC, 1, struct ioctl_cmd)
#define IOCTL_VALGET _IOR(IOC_MAGIC, 2, struct ioctl_cmd)

#endif
```

devone.c

```
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include "devone_ioctl.h"
```

```
MODULE_LICENSE("Dual BSD/GPL");

static int devone_devs = 1;
static unsigned int devone_major = 0;
static unsigned int devone_minor = 0;
static struct cdev devone_cdev;
static struct class *devone_class = NULL;
static dev_t devone_dev;

struct devone_data {
rwlock_t lock;
unsigned char val;
};

int devone_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
{
struct devone_data *dev = filp->private_data;
int retval = 0;
unsigned char val;
struct ioctl_cmd data;

memset(&data, 0, sizeof(data));

switch (cmd) {
case IOCTL_VALSET:
if (!capable(CAP_SYS_ADMIN)) {
retval = -EPERM;
goto done;
}
if (!access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd))) {
retval = -EFAULT;
```

```
goto done;
}
if ( copy_from_user(&data, (int __user *)arg, sizeof(data)) ) {
retval = -EFAULT;
goto done;
}

printk("IOCTL_cmd.val %u (%s)\n", data.val, __func__);

write_lock(&dev->lock);
dev->val = data.val;
write_unlock(&dev->lock);
break;
case IOCTL_VALGET:
if (!access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd))) {
retval = -EFAULT;
goto done;
}
read_lock(&dev->lock);
val = dev->val;
read_unlock(&dev->lock);
data.val = val;
if ( copy_to_user((int __user *)arg, &data, sizeof(data)) ) {
retval = -EFAULT;
goto done;
}
break;
default:
retval = -ENOTTY;
break;
}
done:
```

```
return (retval);
}

ssize_t devone_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos) {
struct devone_data *dev = filp->private_data;
unsigned char val;
int retval;
int i;

read_lock(&dev->lock);
val = dev->val;
read_unlock(&dev->lock);
for ( i=0; i if (copy_to_user(&buf[i], &val, 1)) {
retval = -EFAULT;
goto out;
}
}
retval = count;
out:
return (retval);
}

int devone_close(struct inode *inode, struct file *filp) {
struct devone_data *dev = filp->private_data;

if (dev) {

kfree(dev);
}
return 0;
}
```

```
int devone_open(struct inode *inode, struct file *filp) {
struct devone_data *dev;

dev = kmalloc(sizeof(struct devone_data), GFP_KERNEL);
if (dev == NULL) {
return -ENOMEM;
}

rwlock_init(&dev->lock);
dev->val = 0xFF;

filp->private_data = dev;

return 0;
}

struct file_operations devone_fops = {
.owner = THIS_MODULE,
.open = devone_open,
.release = devone_close,
.read = devone_read,
.ioctl = devone_ioctl,
};

static int devone_init(void) {
dev_t dev = MKDEV(devone_major, 0);
int alloc_ret = 0;
int major;
int cdev_err = 0;
struct class_device *class_dev = NULL;

alloc_ret = alloc_chrdev_region(&dev, 0, devone_devs, "devone");
```

```
if (alloc_ret)
goto error;
devone_major = major = MAJOR(dev);

cdev_init(&devone_cdev, &devone_fops);
devone_cdev.owner = THIS_MODULE;
devone_cdev.ops = &devone_fops;
cdev_err = cdev_add(&devone_cdev, MKDEV(devone_major, devone_minor), 1);
if (cdev_err)
goto error;

devone_class = class_create(THIS_MODULE, "devone");
if (IS_ERR(devone_class)) {
goto error;
}
devone_dev = MKDEV(devone_major, devone_minor);
class_dev = class_device_create(devone_class, NULL, devone_dev, NULL,
"devone%d", devone_minor);

printk(KERN_ALERT "devone driver(major %d) installed.\n", major);
return 0;
error:
if (cdev_err == 0)
cdev_del(&devone_cdev);
if (alloc_ret == 0)
unregister_chrdev_region(dev, devone_devs);
return -1;
}

static void devone_exit(void) {
dev_t dev = MKDEV(devone_major, 0);
```

```
class_device_destroy(devone_class, devone_dev);
class_destroy(devone_class);

cdev_del(&devone_cdev);
unregister_chrdev_region(dev, devone_devs);

printk(KERN_ALERT "devone driver removed.\n");
}

module_init(devone_init);
module_exit(devone_exit);
```

Makefile 同上。

sample.c

```
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include "devone_ioctl.h"

#define DEVFILE "/dev/devone0"

void read_buffer(int fd) {
unsigned char buf[64];
int ret;
int i;
```

```
ret = read(fd, buf, sizeof(buf));
if (ret == -1) {
perror("read");
}
for ( i=0; i printf("%02x ", buf[i]);
}
printf("\n");
}

int main(void) {
struct ioctl_cmd cmd;
int ret;
int fd;

fd = open(DEVFILE, O_RDWR);
if (fd == -1) {
perror("open");
exit(1);
}

memset(&cmd, 0, sizeof(cmd));
ret = ioctl(fd, IOCTL_VALGET, &cmd);
if (ret == -1) {
printf("errno %d\n", errno);
perror("ioctl");
}
printf("val %d\n", cmd.val);

read_buffer(fd);

memset(&cmd, 0, sizeof(cmd));
```

```
cmd.val = 0xCC;
ret = ioctl(fd, IOCTL_VALSET, &cmd);
if (ret == -1) {
printf("errno %d\n", errno);
perror("ioctl");
}

read_buffer(fd);

memset(&cmd, 0, sizeof(cmd));
ret = ioctl(fd, IOCTL_VALGET, &cmd);
if (ret == -1) {
printf("errno %d\n", errno);
perror("ioctl");
}
printf("val %d\n", cmd.val);

close(fd);
return 0;
}
```

執行範例：

```
insmod sample.ko
lsmod | grep sample
ls -l /dev/devone0
hexdump -C -n 16 /dev/devone0
./a.out
```

6-2、64-bit kernel

在 64-bit Linux 的 kernel space 可以執行 64-bit 與 32-bit 的應用程式，所以驅動程式也必須注意應用程式有 32-bit 與 64-bit 兩種情形。

Data Model(資料模型)

OS 從32-bit 轉移到 64-bit 帶來的開發差異，在於資料型別的大小改變了，稱為「資料模型的差異」。

轉移到 64-bit 環境的要點

sizeof

32-bit linux 與 64-bit linux 不同之處，除了 pointer 的大小由 4 bytes 改變為 8 bytes 之外，long 的長度也由 4 bytes 改為 8bytes，所以使用 sizeof(long) 所傳回的值會不一樣，在 32-bit linux 可以順利執行的程式，在 64-bit linux 則可能造成緩衝區溢位。

數字型別轉換

在比較兩個數值時，如果彼此資料型別不同，可能會造成邏輯錯誤的現象，也可能會造成 32-bit linux 與 64-bit linux 執行結果不同的情況，算術型別轉換機制如下：

幾乎在所有的情況下，都會把 signed 數字 轉換成 unsigned 數字的型別，之後再作比較，除非 unsigned 數字這端的級別比較低，而且 signed 的資料型別的 bit 數目比 unsigned 的資料型別的 bit 數目多。

printf

顯示指標的位址通常使用「%x」，但它的寬度是 32-bit，無法在 64-bit 環境下顯示正確結果，需改用「%p」。

整數的上限

在 C 語言中，將整數視為 int 來計算，如果在 64-bit 環境下，想用 64-bit 來計算，要在數值後面加上「L」使用成為 long long。

指標的差值

指標可以相減，但在 64-bit 環境下指標是 8 bytes 長，必須將相減結果放到 long 變數內。

「-1」的處理方式

在 32-bit 環境下，-1是「0xffffffff」，若是在 64-bit 環境下，-1 轉成 long 的話是「0xffffffffffffffff」，但是如果把 -1 儲存到 unsigned int 之類的 32-bit 變數時，就會維持「0xffffffff」的形式。

NULL 與 0

空指標由「NULL」巨集定義的，在 C 語言通常以指標實作：

```
#define NULL ((void*)0)
```

實作 IOCTL 介面

實作驅動程式的 IOCTL 介面時，需考慮 64-bit 環境帶來的影響，特別是傳遞 ioctl() 引數結構的動作。

比如說，下面這個結構由於內含指標成員，所以在 32-bit 與 64-bit 環境下的大小、變數偏移量並不相同。

```
struct ioctl_com {
int cmd;
unsigned int size;
unsigned char *buf;
int flag;
}
```

如果沒有用 pack pragma(#pragma pack()) 指定變數打包方式的話，編譯器會為了提高記憶體讀寫效率、應對處理器的限制，而自動把變數配置在適當的偏移量上。
在 32-bit 環境是以 4 bytes 為一個單位，而在 64-bit 環境則是以 8 bytes 為配置單位，為了讓結構的大小能貼齊邊界，結構最後也可能自動補上空白。
這種結構大小的差異會影響在 64-bit linux 環境下執行的 32-bit 應用程式，如果驅動程式跟應用程式對記憶體的佈局認知不同，就會參考到不對的地方，進而引發錯誤。

gcc 定義的巨集
32-bit 應用程式發出的 IOCTL 結構必須與 64-bit 應用程式一致才行，如此，驅動程式就不必區分這兩種結構佈局差異。
要達到這個目的，可以使用編譯器定義的巨集，如此，在編譯應用程式時，判斷是 32-bit 還是 64-bit，讓結構的偏移量能一致。
64-bit gccc 會定義下面這類巨集，代表編譯的是 64-bit 應用程式：

```
*__LP64__
*__ia64__
```

下面這兩個指令，都可以用來列出巨集的定義：

```
echo | gcc -v -E -dM -
echo | cpp -dM
```

IOCTL 結構的寫法
將新改寫前面的範例結構，如下：

```
struct ioctl_com {
int cmd;
unsigned int size;
#ifdef __LP64__
unsigned char *buf; /* 8 bytes */
#else
unsigned char *buf; /* 4 bytes */
```

```
unsigned int mbz; /* 4 bytes padding*/
#endif /* __LP64__ */
int flag;
int dummy_pad
}
```

身為指標的 buf 成員位置之後，在以 32-bit 編譯時明確插入平移(mbz 成員)
為了讓結構大小是 8 bytes 的倍數，所以尾端也加上平移(dummy_pad 成員)。
如此，就算以 32-bit 編譯，結構的記憶體配置也對與 64-bit 時相同。
驅動程式讀出 buf 成員時會讀到 8 bytes 資料，但只要事先將 mbz 設為 0，就能得到正確的 32-bit 應用程式指標(4 bytes)。
目前的結構是專屬於「little endian」，如果要再加上支援「big endian」的話，需再作修改：

```
struct ioctl_com {
int cmd;
unsigned int size;
#ifdef __LP64__
unsigned char *buf; /* 8 bytes */
#else

#ifdef __LITTLE_ENDIAN_BITFIELD
unsigned char *buf; /* 4 bytes */
unsigned int mbz; /* 4 bytes padding*/
#elif __BIG_ENDIAN_BITFIELD
unsigned int mbz; /* 4 bytes padding*/
unsigned char *buf; /* 4 bytes */
#else
#error "Please fix "
#endif

#endif /* __LP64__ */
int flag;
int dummy_pad
}
```

DMA

DMA (Direct Memory Access) 傳送資料時必須注意的問題之一，是 DMA 暫存區的位址變成 64 bits 時，會不會出現問題。
執行 DMA 的是硬體本身，如果硬體不支援 64 bits 位址，就必須限制在 32-bit 位址範圍內。

6-3、select/poll

以 open 系統呼叫開啟檔案時，預設會以「blocking mode」開啟，block 指的是 process 為了等待某件事情發生，而進入 sleep 狀態的情形。
Linux 應用程式多以 blocking mode 開發，而 windows 的主流則是 non-blocking mode。
對應用程式來說，如果 read 系統呼叫會被 block 的話，有時可能會引發設計上的問題，所以 linux 準備了以下的措施：

- Non-block 模式
- 同時執行多個同步 I/O 工作

啟用 non-blocking 模式後，read() 與 write() 就不會被 block 了，而是傳回「EAGAIN」錯誤碼(errno)。
在應用程式中，如果要使用 non-blocking 模式的話，就要在 open() 開檔時，以 OR 指定「O_NONBLOCK」選項。

```
fd = open(DEVFILE, O_RDWR | O_NONBLOCK);
if ( fd == -1 ) {
perror("open");
exit(1);
}
```

在驅動程式的部分，如果要支援 non-blocking 模式的話，驅動程式也需要改寫才行，會在下面章節中提到。

同時執行多個同步 I/O 工作

同時執行多個同步 I/O 工作(synchronous I/O multiplexing) 指的是使用 select 系統呼叫。
透過 select 系統呼叫，可以監視多個 file handle 在三種狀態之的變化(可讀取、可寫入、發生錯誤)。
select 系統呼叫本身會被 block，但是可以指定 timeout。
read() 呼叫後，會一直等到資料抵達為止，所以可以在呼叫 read() 之前先呼叫 select() 判斷資料抵達了沒有(能不能讀取)，如果可以讀取，再呼叫 read()。

```
retval = select(fd+1, &rfds, NULL, NULL, &tv);
if ( retval ) {
```

```
read(fd, buf, sizeof(buf));
}
```

使用 select() 時，有幾個經常用到的函式巨集，請情可以參考「man 2」。

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval
*timeout);
void FD_CLR(int fd, fd_set *set);
void FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

另外還有個跟 select() 很類似的「poll()」系統呼叫，以基本功能來說 select() 與 poll() 都一樣，但指定 file handle 的方式不同。

而且，在指定多個 file handle 時，poll() 走訪所有 file handle 的速度較快，請細用法可以參考「man 2 poll」。

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
struct pollfd {
int fd; /* file descriptor */
short events; /* requested events */
short revents; /* returned events */
};
```

驅動程式的實作

應用程式想同時執行多個同步 I/O 工作時，可以使用的系統呼叫有好幾個，包含 select 與 poll 等，但驅動程式只需要準備一個函式就夠了，kernel 只會呼叫驅動程式提供的這個函式。

Character 類型的裝置想支援同時執行多個同步 I/O 工作的話，只要在驅動程式準備 poll 方法即可。

```
unsigned int (*poll) (struct file *, struct poll_table_struct *);
```

poll 方法會在 kernel 處理 select 與 poll 之類的系統呼叫時用到，它必須執行的工作很簡單，包含：

- 在 wait queue 登記
- 傳回目前可以操作的狀態

在呼叫執行多個同步 I/O 工作的系統呼叫時，「block 直到狀態變化」的動作，指的是在 kernel 裡面 sleep。

要 sleep，需要先準備 wait queue (wait_queue_head_t)，由驅動程式負責提供。

Wait queue 可以參考以下 6.6 節 -「Sleep 與 Wake Up」。

登記 wait queue 的工作可透過 poll_wait() 完成，定義在「linux/poll.h」：
void poll_wait(struct file *filp, wait_queue_head_t *wait_address, poll_table *p);
在呼叫同時執行多個同步 I/O 工作的系統呼叫時，解除 block 的時機，是驅動程式透過傳給 poll_wait() 的 wait queue 被喚醒的時候。
Kernel 在被 wait queue 喚醒之後，會再次呼叫驅動程式的 poll() 確認是否已成為等待中的狀態 (可寫入或可讀出)。
如果是的話 FD_ISSET 巨集就會成立，要判斷是否為這種狀態，當然還是需要驅動程式提供資訊才行，這個資訊就是透過 poll() 方法的傳回值來表示。
傳回值要透過 linux/poll.h 定義的巨集 OR 起來表示，以下是常用到的組合：

Bit OR	意義
POLLIN | POLLRDNORM	可讀取
POLLOUT | POLLWRNORM	可寫入
POLLIN | POLLRDNORM | POLLOUT | POLLWRNORM	可讀寫
POLLERR	發生錯誤
POLLHUP	裝置離線(EOF)

下列是 poll 方法的程式片段，poll 方法在向 wait queue 登記完成之後必須立刻返回，不能在 poll 方法裡 sleep：

```
unsigned int devone_poll(struct file *filp, poll_table *wait) {
struct devone_data *dev = filp->private_data;
unsigned int mask = 0;

if (dev == NULL)
return -EBADFD;
poll_wait(filp, &dev->read_wait, wait);
if (dev->timeout_done == 1) { /* readable */
mask |= POLLIN | POLLRDNORM;
}
printk("%s returned (mask 0x%x)\n", __func__, mask);
return mask;
}
```

範例驅動程式

接著來寫滿足下列條件的驅動程式：

- 支援 user process 發出 read() 系統呼叫
- 有資料時，read() 會立刻反回
- 無資料時，read() 會 sleep (block)
- 支援 poll() 系統呼叫
- 發出 write() 會導致錯誤

模擬有無資料的方式，是在收到 open() 系統呼叫時啟動 kernel timer，經過一段時間後 wake up，來模擬收到新資料。
此範例是假想應用程式為了從硬體取得資料，所以送出 read() 系統呼叫的情形。
沒有資料時，驅動程式內部需要 sleep，但是要能在 user process 收到 signal 的時候解除 sleep 才行。
這種用途可以使用 wait_event_interruptible() 這個 kernel 函式，如果要避免錯過 wake up 的時機，可以改用 wait_event_interruptible_timeout() 這個附帶 time out 功能的函式。
順道一提，在收網路封包時，處理的方式也十分類似，是透過實作於 net/core/datagram.c 的 wiat_for_packet() 來等待的。
另一個要考慮的地方是 non-blocking mode 的支援，User process 以 non-blocking mode 開啟裝置檔時，read() 就不能 block，也就是驅動程式內不能 sleep，沒有資料的話必須傳回錯誤碼才行。

main.c ，產生 .ko 檔的驅程程式程式碼：

```
#include
#include
#include
#include
#include
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static unsigned int timeout_value = 10;
```

```
module_param(timeout_value, uint, 0);

static int devone_devs = 1;
static int devone_major = 0;
static int devone_minor = 0;
static struct cdev devone_cdev;
static struct class *devone_class = NULL;
static dev_t devone_dev;

struct devone_data {
struct timer_list timeout;
spinlock_t lock;
wait_queue_head_t read_wait;
int timeout_done;
struct semaphore sem;
};

static void devone_timeout(unsigned long arg) {
struct devone_data *dev = (struct devone_data *)arg;
unsigned long flags;

printk("%s called\n", __func__);

spin_lock_irqsave(&dev->lock, flags);

dev->timeout_done = 1;
wake_up_interruptible(&dev->read_wait);
spin_unlock_irqrestore(&dev->lock, flags);
}

unsigned int devone_poll(struct file *filp, poll_table *wait) {
struct devone_data *dev = filp->private_data;
```

```
unsigned int mask = POLLOUT | POLLWRNORM;
printk("%s called\n", __func__);

if (dev == NULL)
return -EBADFD;

down(&dev->sem);
poll_wait(filp, &dev->read_wait, wait);
if (dev->timeout_done == 1) {
mask |= POLLIN | POLLRDNORM;
}
up(&dev->sem);

printk("%s returned (mask 0x%x)\n", __func__, mask);
return (mask);
}

ssize_t devone_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos) {
return -EFAULT;
}

ssize_t devone_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos) {
struct devone_data *dev = filp->private_data;
int i;
unsigned char val;
int retval;

if (down_interruptible(&dev->sem))
return -ERESTARTSYS;

if (dev->timeout_done == 0) {
up(&dev->sem);
```

```
if (filp->f_flags & O_NONBLOCK)
return -EAGAIN;
do {
retval = wait_event_interruptible_timeout(dev->read_wait, dev->timeout_done == 1,
1*HZ);
if (retval == -ERESTARTSYS)
return -ERESTARTSYS;
} while (retval == 0);
if (down_interruptible(&dev->sem))
return -ERESTARTSYS;
}

val = 0xff;
for (i=0; i if (copy_to_user(&buf[i], &val, 1)) {
retval = -EFAULT;
goto out;
}
}
retval = count;
out:
dev->timeout_done = 0;
mod_timer(&dev->timeout, jiffies+timeout_value*HZ);
up(&dev->sem);
return (retval);
}

int devone_close(struct inode *inode, struct file *filp) {
struct devone_data *dev = filp->private_data;
if (dev) {
del_timer_sync(&dev->timeout);
kfree(dev);
}
```

```
return 0;
}

int devone_open(struct inode *inode, struct file *filp) {
struct devone_data *dev;
dev = kmalloc(sizeof(struct devone_data), GFP_KERNEL);
if (dev == NULL)
return -ENOMEM;

spin_lock_init(&dev->lock);
init_waitqueue_head(&dev->read_wait);
init_MUTEX(&dev->sem);
init_timer(&dev->timeout);
dev->timeout.function = devone_timeout;
dev->timeout.data = (unsigned long)dev;

filp->private_data = dev;
dev->timeout_done = 0;
mod_timer(&dev->timeout, jiffies+timeout_value*HZ);

return 0;
}

struct file_operations devone_fops = {
.owner = THIS_MODULE,
.open = devone_open,
.release = devone_close,
.read = devone_read,
.write = devone_write,
.poll = devone_poll,
};
```

```
static int devone_init(void) {
dev_t dev = MKDEV(devone_major, 0);
int alloc_ret = 0;
int major;
int cdev_err = 0;
struct class_device *class_dev = NULL;

alloc_ret = alloc_chrdev_region(&dev, 0, devone_devs, "devone");
if (alloc_ret)
goto error;
devone_major = major = MAJOR(dev);

cdev_init(&devone_cdev, &devone_fops);
devone_cdev.owner = THIS_MODULE;
devone_cdev.ops = &devone_fops;
cdev_err = cdev_add(&devone_cdev, MKDEV(devone_major, devone_minor), 1);
if (cdev_err)
goto error;

devone_class = class_create(THIS_MODULE, "devone");
if (IS_ERR(devone_class))
goto error;

devone_dev = MKDEV(devone_major, devone_minor);
class_dev = class_device_create(devone_class, NULL, devone_dev, NULL,
"devone%d", devone_minor);

printk(KERN_ALERT "devone driver(major %d) installed.\n", major);
return 0;

error:
if (cdev_err == 0)
```

```
cdev_del(&devone_cdev);
if (alloc_ret == 0)
unregister_chrdev_region(dev, devone_devs);
return -1;
}

static void devone_exit(void) {
dev_t dev = MKDEV(devone_major, 0);
class_device_destroy(devone_class, devone_dev);
class_destroy(devone_class);
cdev_del(&devone_cdev);
unregister_chrdev_region(dev, devone_devs);
printk(KERN_ALERT "devone driver removed\n");
}

module_init(devone_init);
module_exit(devone_exit);
```

read.c - 依照 read() 的 flag 的設定，此程式可以寫成 block mode 或 non-block mode，依 blocking 這個參數來選擇。

```
#include
#include
#include
#include
#include
#include
#include
#include
#include

#define DEVFILE "/dev/devone0"
```

```
int main(void) {
int fd;
unsigned char buf[64];
ssize_t sz;
int i;
int blocking = 0;

if ( blocking )
fd = open(DEVFILE, O_RDWR);
else
fd = open(DEVFILE, O_RDWR|O_NONBLOCK);
if (fd == -1) {
perror("open");
exit(1);
}

printf("read() ...\n");
sz = read(fd, buf, sizeof(buf));
printf("read() %d\n", sz);

if ( sz > 0 ) {
for ( i=0; i printf("%02x ", buf[i]);
}
printf("\n");
} else {
printf("errno %d\n", errno);
perror("read");
}

close(fd);
return 0;
}
```

select.c - 透過 select() 等待，直到可以讀取資料時才送出 read()：

```
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include

#define DEVFILE "/dev/devone0"

int main(void) {
int fd;
fd_set rfds;
struct timeval tv;
int retval;
unsigned char buf[64];
ssize_t sz;
int i;

fd = open(DEVFILE, O_RDWR);
if (fd == -1) {
perror("open");
exit(1);
}

do {
FD_ZERO(&rfds);
FD_SET(fd, &rfds);
```

```
tv.tv_sec = 5;
tv.tv_usec = 0;

printf("select() ...\n");
retval = select(fd+1, &rfds, NULL, NULL, &tv);
if (retval == -1) {
perror("select");
break;
}
if (retval) {
break;
}
} while (retval == 0);

if (FD_ISSET(fd, &rfds)) {
printf("read() ...\n");
sz = read(fd, buf, sizeof(buf));
printf("read() %d\n", sz);
for ( i=0; i printf("%02x ", buf[i]);
}
printf("\n");
}

close(fd);
return 0;
}
```

poll.c - 透過 poll() 等待，直到可以讀取資料時才送出 read()：

```
#include
#include
#include
#include
#include
```

```
#include
#include
#include
#include
#include
#include

#define DEVFILE "/dev/devone0"

int main(void) {
int fd;
struct pollfd fds;
int retval;
unsigned char buf[64];
ssize_t sz;
int i;

fd = open(DEVFILE, O_RDWR);
if (fd == -1) {
perror("open");
exit(1);
}

do {
fds.fd = fd;
fds.events = POLLIN;

printf("poll() ...\n");
retval = poll(&fds, 1, 5*1000);
if (retval == -1) {
perror("poll");
break;
```

```
}
if (retval) {
break;
}
} while (retval == 0);

if (fds.revents & POLLIN) {
printf("read() ...\n");
sz = read(fd, buf, sizeof(buf));
printf("read() %d\n", sz);
for ( i=0; i printf("%02x ", buf[i]);
}
printf("\n");
}

close(fd);
return 0;
}
```

6-4、procfs

profs (/proc file system) 指的是位於 /proc 的虛擬檔案系統 (實體在 RAM 之內)，內含 kernel 與各驅動程式的設定選項。
/proc 裡面的大多數檔案都是唯讀的，但也有些檔案可讓使用者寫入新設定。

建立唯讀的 procfs

在驅動程式初始化時，呼叫 create_proc_read_entry() 即可建立唯讀的 procfs 檔案，定義的標頭檔是 linux/proc_fs.h：

```
struct proc_dir_entry *create_proc_read_entry(const char *name, mode_t mode, struct proc_dir_entry *base, read_proc_t *read_proc, void *data);
```

各引數的意義如下：

name	在 /proc 之內的檔案名稱
mode	權限 (可 NULL)
base	上層目錄 (可 NULL)

read_proc	讀取處理函式
data	私有資料 (可 NULL)

其中 read_proc 引數要傳入讀取時呼叫的處理函式指標，這個函式的 prototype 已 typedef 為 read_proc_t：

```
typedef int (read_proc_t) (char *page, char **start, off_t off, int count, int *eof, void *data);
```

各引數的意義為：

page	Kernel 配置的記憶體空間
start	驅動程式寫入資料的開始位址指標(由驅動程式回傳)
off	驅動程式傳回資料的偏移量
count	User process 一次讀回的量
eof	通知已達資料終點(由驅動程式回傳)
data	create_proc_read_entry() 設定的私有資料

page 引數是 kernel 分配的記憶體，所以驅動程式可以直接寫入，但是因為只分配了一個 page 的大小(PAGE_SIZE)，所以寫入的資料量超過 PAGE_SIZE 就會破壞記憶體內容。
start 引數在呼叫讀取處理函式之後的值是「*start = NULL」，start 指向 page 這塊記憶體空間裡開始寫入資料的位置，由驅動程式負責設定，如果資料量沒超過 PAGE_SIZE 的話，驅動程式就不必動到這個引數。
off 引數是 kernel 負責更新的偏移量，代表驅動程式回傳資料的偏移量，如果資料量沒超過 PAGE_SIZE 的話，就不必特地處理這個引數。
count 引數是 user process 一次讀回的資料量，通常比 PAGE_SIZE 小，資料量沒超過 PAGE_SIZE 的話，也不必特地處理這個引數。
讀取處理函式的工作內容，實作於 fs/proc/generic 的 proc_file_read()。
驅動程式卸載時，要呼叫 remove_proc_entry() 清除 procfs 的內容：

```
void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
```

procfs.c：

```
#include
#include
#include
#include
```

```
MODULE_LICENSE("Dual BSD/GPL");

#define PROCNAME "driver/sample"
#define DUMMY_BUFSIZ 4096
static char dummy[DUMMY_BUFSIZ];

static int sample_read_proc(char *page, char **start, off_t off, int count, int *eof, void
*data) {
int len;
printk("page=%p *start=%p off=%d count=%d *eof=%d data=%p\n", page, *start, (int)off,
count, *eof, data);

len = DUMMY_BUFSIZ;
if (len > PAGE_SIZE)
return -ENOBUFS;

memcpy(page, dummy, len);
*eof = 1;

printk("len=%d (%s)\n", len, __func__);
return (len);
}

static int sample_init(void) {
struct proc_dir_entry *entry;
int i;

entry = create_proc_read_entry(PROCNAME, S_IRUGO | S_IWUGO, NULL,
sample_read_proc, NULL);

if (entry == NULL)
```

```
printk(KERN_WARNING "sample: unable to create /proc entry");

for (i=0; i dummy[i] = 'A' + (i%26);

printk("driver loaded\n");
return 0;
}

static void sample_exit(void) {
remove_proc_entry(PROCNAME, NULL);
printk(KERN_ALERT "devone unloaded\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

執行範例：

```
# sudo /sbin/insmod ./sample.ko
# cat /proc/driver/sample | wc -c
#dmesg
```

建立可讀寫的 procfs

要建立可讀寫的 procfs 項目時，可透過 create_proc_entry() 指記到 /proc 之內：

```
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode, struct
proc_dir_entry *parent);
```

各引數的意義為：

name	/proc 之內的項目名稱
mode	權限 (可NULL)
base	上層目錄 (可NULL)

登記處理函式的方式，是為 create_proc_entry() 傳回的 proc_dir_entry 結構設定 read_proc 與 write_proc 成員。 read_proc 成員的 prototype 與上節相同，已被 typedef 為 read_proc_t：

typedef int (read_proc_t) (char *page, char **start, off_t off, int count, int *eof, void *data); write_proc 成員的 prototype 也已被 typedef 為 write_proc_t：
typedef int (write_proc_t) (struct file *file, const char __user **buffer, unsigned long count, void *data);
這兩個 prototype 並不相同，且 write_proc 會收到 user process 傳來的資料，所以必須透過 copy_from_user() 接收這些資料。

procfs.c：

```
#include
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

#define PROCNAME "driver/sample"
static int sample_flag = 0;

static int sample_proc_read(char *page, char **start, off_t off, int count, int *eof, void
*data) {
int len;
printk(KERN_INFO "%s called\n", __func__);

if (off>0)
len = 0;
else
len = sprintf(page, "%d\n", sample_flag);

return (len);
}
```

```
static int sample_proc_write(struct file *file, const char *buffer, unsigned long count, void
*data) {
char buf[16];
unsigned long len = count;
int n;

printk(KERN_INFO "%d (%s)\n", (int)len, __func__);

if (len >= sizeof(buf))
len = sizeof(buf)-1;

if (copy_from_user(buf, buffer, len))
return -EFAULT;
buf[len] = '\0';

n = simple_strtol(buf, NULL, 10);
if (n==0)
sample_flag = 0;
else
sample_flag = 1;

return (len);
}

static int sample_init(void) {
struct proc_dir_entry *entry;

entry = create_proc_entry(PROCNAME, 0666, NULL);

if (entry == NULL) {
printk(KERN_WARNING "sample: unable to create /proc entry");
return -ENOMEM;
```

```
}

entry->read_proc = sample_proc_read;
entry->write_proc = sample_proc_write;
entry->owner = THIS_MODULE;

printk("driver loaded\n");
return 0;
}

static void sample_exit(void) {
remove_proc_entry(PROCNAME, NULL);
printk(KERN_ALERT "devone unloaded\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

執行範例：

```
# sudo /sbin/insmod ./sample.ko
# ls -l /proc/driver/sample
# cat /proc/driver/sample
# echo 1 > /proc/driver/sample
# cat /proc/driver/sample
# echo 0 > /proc/driver/sample
# cat /proc/driver/sample
```

其它功能

使用 proc_mkdir() 即可在 /proc 建立新目錄：

```
struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent);
```

要在新建立的目錄中放入檔案的話，只要將 proc_mkdir() 的傳回值當成「上層目錄」透過 create_proc_entry() 的第三引數傳入即可。
則除目錄時，可以呼叫 remove_proc_entry()。
呼叫 proc_symlink() 即可在 /proc 之內建立 symbolic link：

```
struct proc_dir_entry *proc_symlink(const char *name, struct proc_dir_entry *parent,
const char *dest);
```

6-5、seq_file

/proc 在處高小資料時很方便，但一旦想一次交換的資料超過 page size 就會變得很複雜，另外如果 /proc 的檔案同時被許多 process 讀取時，也可能造成問題。
為了解決這些困擾，Linux 定義了新的「seq_file」介面。

seq_file 的位階

seq_file 不是用來取代 /proc 的東西，而是停用 procfs 的 read_proc 與 write_proc 改用 proc_fops 的作法。
將 proc_fops 的成員登記為 seq_file 提供的函式群，即可改用「seq_file」介面。
使用 seq_file 介面的話，就不必煩惱 page size，可以輕易輸出大量資料。

seq_file 的用法

驅動程式需要把 linux/seq_file.h 及 linux/proc_fs.h 這兩個標頭檔一起 include 進來。
seq_file 的基本用法為：

```
static struct seq_operations sample_seq_op = {
.start = as_start,
.next = as_next,
.stop = as_stop,
.show = as_show,
};

static int sample_proc_open(struct inode *inode, struct file *file)
{
return seq_open(file, &sample_seq_op);
}

static struct file_operations sample_operations = {
.open = sample_proc_open,
.read = seq_read,
.llseek = seq_lseek,
.release = seq_release,
};
```

```
static int sample_init(void) {
struct proc_dir_entry *entry;

entry = create_proc_entry("file name inside procfs", S_IRUGO | S_IWUGO, NULL);
if (entry)
entry->proc_fops = &sample_operations;
}
```

重點在 seq_operations 結構定義的四個處理函式：

- as_start()
- as_next()
- as_stop()
- as_show()

驅動程式開發者只要設計這四個處理函式即可，seq_file 介面原則上只支援「讀取」，沒辦法寫入。
各成員意義如下：

as_start	開始讀取時最先呼叫的函式。傳回最初資料(記錄)的索引(從1起算)，沒有資料時傳回「0」
as_next	有下一筆資料時，傳回下一筆資料的索引，否則傳回「0」
as_stop	在讀完所有資料之後呼叫的函式
as_show	顯示資料

seq_file 的觀念是從 C++ 與 Java 的迭代器(iterator) 學來的，這邊就不附上流程圖。
各成員函式的 prototype 如下：

```
void * (*start) (struct seq_file *m, loff_t *pos);
void (*stop) (struct seq_file *m, void *v);
void * (*next) (struct seq_file *m, void *v, loff_t *pos);
int (*show) (struct seq_file *m, void *v);
```

start 與 next 接到的「*pos」變數會從 0 開始，隨著逐項處理資料的過程都加，增加這個變數的動作是由驅動程式進行的，kernel 不會修改。

start 與 next 如果有要顯示的資料，就要以 void* 的型別傳回「1以上」的索引，實際上，void * 的使用方式是由驅動程式自行決定，而 kernel 只會把以下兩種傳回值當成已經沒有資料：

- NULL
- IS_ERR 巨集成立的值(負數)

也就是說，如果把它當資料索引的話，須注意要「從1起算」。
stop 如果不需要做額外處理的話，不實作也無妨，但如果在取用資料時必須鎖定的話，可以在 start 裡面鎖定、在 stop 釋放鎖定。
show 負責將資料顯示給使用者，採用下列 seq_file 專用的介面：

```
int seq_printf(struct seq_file *m, const char *f, ...);
int seq_putc(struct seq_file *m, char c);
int seq_puts(struct seq_file *m, const char *s);
```

seq_printf() 接受可變引數，seq_putc() 可輸出一個字元，seq_puts() 則輸出一個字串。

seq_file.c - 範例程式

```
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

#define PROCNAME "sample"

static char *data_message[] = {
"Fedora",
"Red Hat Enterprise Server",
"Debian",
"Vine",
"Ubuntu",
0
};
```

```
static void *as_start(struct seq_file *m, loff_t *pos) {
loff_t n = *pos;
char **p;
int i;

printk("%lld (%s)\n", n, __func__);

if (n==0)
seq_printf(m, "=== seq_file header ===\n");

p = data_message;
for (i=0; p[i]; ++i) {
n--;
if (n<0)
return (void*)(i+1);
}
return 0;
}

static void *as_show(struct seq_file *m, void *p) {
int n = (int)p-1;

printk("%u (%s)\n", (int)p, __func__);
seq_printf(m, "[%d] %s\n", n, data_message[n]);
return 0;
}

static void *as_next(struct seq_file *m, void *p, loff_t *pos) {
int n = (int)p;
char **ptr;

printk("%u (%s)\n", n, __func__);
```

```
(*pos)++;

ptr = data_message;
if (ptr[n]) {
return (void*)(n+1);
}
return 0;
}

static void as_stop(struct seq_file *m, void *p) {
int n = (int)p;
printk("%u (%s)\n", n, __func__);
}

static struct seq_operations sample_seq_op = {
.start = as_start,
.next = as_next,
.stop = as_stop,
.show = as_show,
};

static int sample_proc_open(struct inode *inode, struct file *file)
{
return seq_open(file, &sample_seq_op);
}

static struct file_operations sample_operations = {
.open = sample_proc_open,
.read = seq_read,
.llseek = seq_lseek,
.release = seq_release,
```

```
};

static int sample_init(void) {
struct proc_dir_entry *entry;

entry = create_proc_entry(PROCNAME, S_IRUGO | S_IWUGO, NULL);
if (entry)
entry->proc_fops = &sample_operations;
printk("driver loaded\n");
return 0;
}

static void sample_exit(void) {
remove_proc_entry(PROCNAME, NULL);
printk(KERN_ALERT "driver unloaded\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

執行範例：

```
# sudo /sbin/insmod ./sample.ko
# ls -l /proc/sample
# cat /proc/sample
# dmesg
```

seq_file.c - 範例程式 - 2

透過 kernel 的 task_init 變數，列出目前系統中執行的所有 process。

在參考 task list 時，Linux 2.6.18 之前的版面需要透過讀取專用的 tasklist_lock 進行鎖定，但 Linux 2.6.18 之後的版本不再匯出 tasklist_lock 符號，改成需要取得 RCU (Read Copy Update) 鎖定。

```
#include
#include
```

```
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

#define PROCNAME "sample2"

static void *as_start(struct seq_file *m, loff_t *pos) {
loff_t n = *pos;
struct task_struct *p = NULL;

printk("%lld (%s)\n", n, __func__);

if (n==0)
seq_printf(m, "=== seq_file header ===\n");

#if ( LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,18) )
read_lock(&tasklist_lock);
#else
rcu_read_lock();
#endif
if (n==0)
return (&init_task);

for_each_process (p) {
n--;
if (n<=0)
return (p);
}
return 0;
```

```
}

static void *as_show(struct seq_file *m, void *p) {
struct task_struct *tp = (struct task_struct*)p;

printk("%p (%s)\n", tp, __func__);

seq_printf(m, "[%s] pid=%d\n", tp->comm, tp->pid);
seq_printf(m, " tgid=%d\n", tp->tgid);
seq_printf(m, " state=%ld\n", tp->state);
seq_printf(m, " mm=0x%p\n", tp->mm);
seq_printf(m, " utime=%lu\n", tp->utime);
seq_printf(m, " stime=%lu\n", tp->stime);
seq_printf(m, " oomkilladj=%d\n", tp->oomkilladj);
seq_puts(m, "\n");
return 0;
}

static void *as_next(struct seq_file *m, void *p, loff_t *pos) {
struct task_struct *tp = (struct task_struct*)p;

printk("%lld (%s)\n", *pos, __func__);

(*pos)++;

tp = next_task(tp);
if (tp == &init_task) {
return NULL;
}
return (tp);
}
```

```
static void as_stop(struct seq_file *m, void *p) {
printk("%p (%s)\n", p, __func__);

#if ( LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,18) )
read_unlock(&tasklist_lock);
#else
rcu_read_unlock();
#endif
}

static struct seq_operations sample_seq_op = {
.start = as_start,
.next = as_next,
.stop = as_stop,
.show = as_show,
};

static int sample_proc_open(struct inode *inode, struct file *file)
{
return seq_open(file, &sample_seq_op);
}

static struct file_operations sample_operations = {
.open = sample_proc_open,
.read = seq_read,
.llseek = seq_lseek,
.release = seq_release,
};

static int sample_init(void) {
struct proc_dir_entry *entry;
```

```
entry = create_proc_entry(PROCNAME, S_IRUGO | S_IWUGO, NULL);
if (entry)
entry->proc_fops = &sample_operations;
printk("driver loaded\n");
return 0;
}

static void sample_exit(void) {
remove_proc_entry(PROCNAME, NULL);
printk(KERN_ALERT "driver unloaded\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

執行範例:

```
# sudo /sbin/insmod ./sample.ko
# ls -l /proc/sample2
# cat /proc/sample2
# dmesg
```

6-6、Sleep 與 Wake Up

介紹 sleep 與 wake up 觀念。

Preemptive (可被搶佔)

執行 user process 時，會根據 kernel 內建的排程演算法，由 OS 定期強制切換執行的程式，因此 user process 稱為「preemptive(可被搶佔)」。
切換 user process 的是 OS 本身 (Linux kernel)。
為 user process 排程的演算法有許多種，但負責切換的都是 kernel/sched.c 裡面實作的「schedule()」這個 kernel 函式。
schedule() 切換 process 的動作稱為「context switch (環境切換)」。
schedule() 會被 kernel 許多地方呼叫，主要呼叫的地方包含:

- User context 開始 sleep 的時候
- 嘗試鎖定 spin lock 的時候(啟用 CONFIG_PREEMPT 的情形)

- Timer scheduler 延遲呼叫的時候

而 Linux 的 kernel space 並不是 preemptive 的，包含驅動程式在內的 kernel 工作內容都不是 preemptive。
但是有個例外的情形，在發生硬體中斷時，中斷函式會強制取得 CPU 控制權。
負責在 user process 之間切換的就是 kernel，而 kernel 內部並沒有其它角色負責排程，但是 kernel 內部可以明確放出 CPU 使用權，也就是直接呼叫 schedule()。
特別是驅動程式，有時需等待硬體完成工作，此時就需要暫時放下自己的工作，讓 CPU 去處理其它事情。
為了讓驅動程式能在進行 DMA 之類工作時，可以明確釋放 CPU 使用權，所以 kernel 就提供了 sleep 與 wake up 的功能。

Sleep 有危險

呼叫 sleep_on() 函式後，就能在 kernel 之內進入 sleep，「進入sleep」具體來說是：

- 中斷 user context 要求 kernel 做的事情，明確把 CPU 使用權交給另一個 user context

這個函式無法在 interrupt context 中使用：

```
void sleep_on(wait_queue_head_t *q);
```

引數要傳入 wait_queue_head_t 的指標，這個指標需先以 init_waitqueue_head() 初始化：

```
void init_waitqueue_head(wait_queue_head_t *q);
```

sleep.c - 這段程式碼很危險，只要 sleep ，就會醒不來，因此無法卸載驅動程式，要復原只要重新開機一途，嚴格來說，這個程式連 insmod 都無法執行完成。

```
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static wait_queue_head_t wait;

static int sample_init(void) {
printk("driver loaded\n");
```

```
init_waitqueue_head(&wait);
sleep_on(&wait);

return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "driver unloaded\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

執行範例：

```
# sudo /sbin/insmod ./sleep.ko
# ps aux | grep insmod
# kill -9 #pid
# ps aux | grep insmod
```

Wake up

要讓睡著的人起來，一定要別人叫它，負責個工作的是 wake_up()。
wake_up() 的引數跟傳給 sleep_on() 的是同一個指標，但是如果在呼叫 sleep_on() 之前就呼叫 wake_up() 的話，則什麼都不會做。
以下範例是 kernel module 呼叫 sleep_on()，然後由另一個 kernel module 呼叫 wake_up() 把它叫醒的例子，其中為了讓其它模組也看得到 wait queue，所以用 EXPORT_SYMBOL 巨集將之匯出。

devone.c

```
#include
#include
#include
#include
#include
```

```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static int devone_devs = 1;
static int devone_major = 0;
static struct cdev devone_cdev;

wait_queue_head_t wait;
EXPORT_SYMBOL(wait);

ssize_t devone_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos) {
int i;
unsigned char val = 0xff;
int retval;

init_waitqueue_head(&wait);
sleep_on(&wait);

printk("Wakeup! (%s)\n", __func__);

for (i=0; i if (copy_to_user(&buf[i], &val, 1)) {
retval = -EFAULT;
goto out;
}
}
retval = count;
out:
return (retval);
}
```

```
struct file_operations devone_fops = {
.read = devone_read,
};

static int devone_init(void) {
dev_t dev = MKDEV(devone_major, 0);
int ret;
int major;
int err;

ret = alloc_chrdev_region(&dev, 0, devone_devs, "devone");
if (ret < 0)
return ret;
devone_major = major = MAJOR(dev);

cdev_init(&devone_cdev, &devone_fops);
devone_cdev.owner = THIS_MODULE;
devone_cdev.ops = &devone_fops;
err = cdev_add(&devone_cdev, MKDEV(devone_major, 0), 1);
if (err)
return err;

printk(KERN_ALERT "devone driver (major %d) installed.\n", major);

return 0;
}

static void devone_exit(void) {
dev_t dev = MKDEV(devone_major, 0);
printk(KERN_ALERT "devone driver removed.\n");
cdev_del(&devone_cdev);
unregister_chrdev_region(dev, devone_devs);
```

```
}

module_init(devone_init);
module_exit(devone_exit);
```

wakeup.c

```
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

extern wait_queue_head_t wait;

static int sample_init(void) {
wait_queue_head_t *wq = &wait;
printk(KERN_ALERT "driver loaded\n");
wake_up(wq);
return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "driver unloaded\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

執行範例：

```
# sudo /sbin/insmod ./devone.ko
# mknod --mode=644 /dev/devone c `grep devone /proc/devices | awk '{print $1;}'` 0
# ls -l /dev/devone
# hexdump -C -n 16 /dev/devone
```

sudo /sbin/insmod ./wakeup.ko

禁止使用 sleep_on()

從前面的例子可以發現，如果能保証在 sleep_on() 執行之後再呼叫 wake_up() 的話，就不會有問題，但如果先執行 wake_up() 再執行 sleep_on()，則就沒有人會去叫醒 sleep_on()，就算是送 KILL 信號給 process 也無法解除睡眠，所以目前 sleep_on() 正在朝著廢除的方向前進。

正確的 Sleep 方式

sleep 與 wake up 的正確作法已經定型：

- 以 wait_event_timeout() 入睡
- 以 wake_up() 喚醒

wait_event_timeout() 會：

- 在滿足 wake up 條件的時候解除睡眠
- 未滿足 wake up 條件的時候保持睡眠
- 經過一段時間後，自動解除睡眠

使用這個函式，可以避免「錯過呼叫時間」而造成的「永久睡眠」問題，也可以在不動用 kernel timer 的情況下，達成 timeout 的效果：

long wait_event_timeout(wait_queue_head_t wq, condition, long timeout);

wait_event_timeout巨集定義在 linux/wait.h，第一引數是 wait_queue_head_t 變數，第二引數則是 process 喚醒的條件，第三引數是時間(以 jiffies 為單位).

wait_event_timeout() 會在滿足條件或是被喚醒之前讓 process 保持 sleep 狀態，但如果被喚醒之後，條件還是沒成立的話，就會再次落入 sleep 狀態。

另外，也有可被 signal 解除 sleep 的 wait_event_interruptiable_timeout() 巨集可供使用：

long wait_event_interruptible_timeout(wait_queue_head_t wq, condition, long timeout);

main.c - wait_event_timeout() 的範例程式

```
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

#define SLEEP_TIMEOUT (6*HZ)
```

```
#define WAKEUP_TIMEOUT (6*HZ)

static struct timer_list tickfn;
static wait_queue_head_t wait;
static int condition = 0;

static void sample_timeout(unsigned long arg) {
wait_queue_head_t *wq = (wait_queue_head_t *)arg;

printk("wq %p (%s)\n", wq, __func__);
condition = 1;
wake_up(wq);
}

static int devone_init(void) {
long val;
printk(KERN_ALERT "devone loaded: %p\n", &tickfn);

init_waitqueue_head(&wait);

init_timer(&tickfn);
tickfn.function = sample_timeout;
tickfn.data = (unsigned long) &wait;
tickfn.expires = jiffies + WAKEUP_TIMEOUT;
add_timer(&tickfn);

condition = 0;
val = wait_event_timeout(wait, condition, SLEEP_TIMEOUT);
if (val == 0)
printk(KERN_ALERT "Timeout occurred. (%s)\n", __func__);

return 0;
```

```
}

static void devone_exit(void) {
del_timer_sync(&tickfn);
printk(KERN_ALERT "devone unloaded.\n");
}

module_init(devone_init);
module_exit(devone_exit);
```

驗証呼叫時機

wait_event_timeout() 與 wake_up() 的呼叫時機有各種變化，不管是以哪種順序呼叫，驅機程式都不能出問題。
如果是因為硬體故障之類的原因，導致不停產生中斷，這種情形下，驅動程式也要有適當的應對才行。

I2C驅動程式

看不懂…目前。

6-7、向 TTY 主控台輸出訊息

使用 prink() 可以把訊息送到 kernel buffer，但沒辦法顯示到使用者的終端機上。
而應用程式在呼叫 printf() 把資料顯示到畫面上時，也是因為應用程式透過 shell 執行，而 shell 控制著 TTY(Tele-Typewriter) 的緣故。
所以當程式切離 shell 成為 daemon (父程序變成 init process) 的時候，呼叫 printf() 也會失去作用。
但如果限定在這種情形下的話：

- 驅動程式在 user context 運作

就可以讓驅動程式向使用者的終端機送出訊息，取得 user context 的方式是 current 巨集 (linux/sched.h)，TTY 的指標取得方式如下：

```
courrnt->signal->tty
```

tty.c - 傳送訊息到 tty 的範例

```
#include
#include
#include
```

```
#include

MODULE_LICENSE("Dual BSD/GPL");

static void hook_tty_console(char *msg) {
struct tty_struct *tty;
tty = current->signal->tty;
if (tty != NULL) {
((tty->driver)->write)(tty, msg, strlen(msg));
((tty->driver)->write)(tty, "\r\n", 2);
}
}

static int devone_init(void) {
printk(KERN_ALERT "devone loaded.\n");
hook_tty_console("This message is written by sample driver");
return 0;
}

static void devone_exit(void) {
printk(KERN_ALERT "devone unloaded.\n");
}

module_init(devone_init);
module_exit(devone_exit);
```

6-8、Interrupt Hook

在硬體產生中斷時，就會呼叫這個 IRQ (Interrupt Request) 對應的驅動程式，IRQ 的分配情形可到 /proc/interrupts 查閱。
IRQ 可以同時分配給多個不同的驅動程式，IRQ 在驅動程式之間是可以共享的。

中斷處理程序的登記工作可透過 request_irq() 完成，在 kernel 收到 IRQ 時，會呼叫所有對應的中斷處理程式。
舉個例子，來對網路卡驅動程式的 IRQ16 登記範例驅動程式內的中斷處理程序看看：

irq.c

```
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

#define IRQ_NUM 16

static irqreturn_t sample_intr(int irq, void *dev_id) {
printk(KERN_ALERT "irq %d dev_id %p (%s)\n", irq, dev_id, __func__);
return IRQ_NONE;
}

static int devone_init(void) {
printk(KERN_ALERT "devone loaded.\n");
if (request_irq(IRQ_NUM, sample_intr, IRQF_SHARED, "sample_intr", (void
*)sample_intr)) {
}
return 0;
}

static void devone_exit(void) {
free_irq(IRQ_NUM, (void *)sample_intr);
printk(KERN_ALERT "devone unloaded.\n");
}

module_init(devone_init);
module_exit(devone_exit);
```

執行範例：

```
# cat /proc/interrupts
# sudo /sbin/insmod ./devone.ko
# sudo tail -f /var/log/messages
# dmesg | tail
# sudo /sbin/rmmod devone
# cat /proc/interrupts
```

6-9、結語

如果 user process 透過 IOCTL 控制程式使用 select() 與 poll() 之類的系統呼叫的話，驅動程式這邊也必須配合實作，另外如果要在 /proc 提供資料時，也必須配合修改驅動程式。接著是 sleep 與 wake up 的手法，如果沒有慎重地進行設計，很可能會讓 user process 完全無法運作。

七、控制硬體

驅動程式最主要的任務 - 控制硬體。

7-1、控制硬體

驅動程式的工作，就是幫助 kernel 控制硬體，kernel 將不認識的硬體交給驅動程式控制，而驅動程式再與 kernel 溝通後，即可讓 kernel 認識硬體，如此， user process 即可透過系統呼叫介面來操作硬體。

各種硬體都有獨特的控制方式，但大多數硬體都可透過驅動程式讀寫「暫存器」來操控。暫存器是硬體(裝置)內部的記憶空間，但屬於斷電後內容就會消失的揮發性記憶體。

驅動程式這類軟體去讀寫裝置的暫存器，也就是 CPU 去讀寫裝置，因為軟體是在 CPU 上面運行的。
具體的讀寫方式隨著裝置而有不同，一般來說可歸為「I/O mapped I/O」與「Memory mapped I/O」這兩類。

7-2、I/O Mapped I/O

「I/O mapped I/O」指的是經由通訊埠讀寫，只是為了與「Memory mapped I/O」對應，才如此描述。
很久以前的裝置，多半都各自擁有 I/O port(通訊埠)，這是用來將資料寫給硬體、或是從硬體讀出資料的介面，代表只有硬體擁有 I/O port，就可以由驅動程式透過 I/O port 進行控制。

為了通作 I/O port，系統準備了「I/O space」這個特別的空間，它與 memory space(記憶體空間) 是各自獨立的，I/O space 裡面的位址稱為「I/O port space」或直稱「port」。
各個裝置都擁有自己的 I/O port address 範圍，硬體設計者必須確保系統內的硬體不會重複用到同一段位址。
Linux 的 I/O port address 分配狀況可透過「cat /proc/ioports」指令來查閱。

I/O port address 的讀寫方式有下面幾種：

- 以 1 byte、2 bytes 或 4 bytes 為單位執行讀寫

因此大量資料不適合經由 I/O port 傳送，另外硬體如果要提供 I/O port 的話，會佔用寶貴的電路板空間，所以最近的硬體多半完全不透過 I/O port 來通訊。

為了讓驅動程式之類的軟體可以操作 I/O port，CPU 提供了 I/O port 專用的 mnemonic(方便記憶的組合語言指令寫法)，而 linux kernel 為了讓驅動程式不必動用組合語言指令，所以提供了 C 語言可以取用的函式，這些函式定義在「asm/io.h」檔案內。

函式名稱	功能
inb()	從 port 讀出 1 byte
inw()	從 port 讀出 2 byte
inl()	從 port 讀出 4 byte
outb()	從 port 寫入 1 byte
outw()	從 port 寫入 2 byte

outl()	從 port 寫入 4 byte

在讀寫 I/O port 的時候需要注意的是，近來 CPU 的性能愈來愈高，因此裝置有可能跟不上 CPU 的處理速度，因此 Linux 準備了會等待的函式，末尾的「p」就代表「pause」的意思：

```
u8 inb_p(unsigned long port);
u16 inw_p(unsigned long port);
u32 inl_p(unsigned long port);
void outb_p(u8 b, unsigned long port);
void outw_p(u16 w, unsigned long port);
void outl_p(u32 l, unsigned long port);
```

如果 kernel 與好幾個驅動程式同時使用同一段 I/O port 的話，硬體運作可能會出問題，因此，要讀寫 I/O port 的 module 要先用 request_region() 把一塊位址空間保留下來，若保留成功，就會在 /proc/ioports 顯示驅動程式的名稱：

```
struct resource *request_region(resource_size_t start, resource_size_t n, const char *name);
```

I/O port 的開始位址透過 start 引數傳入，想讀寫的 port 個數則透過 n 引數傳入，name 是在 /proc/ioports 顯示的名字，不過 name 引數的指標會被 kernel 留著使用，所以不能傳入區域變數的指標。

另外，保留 memory mapped I/O (MMIO) 位址空間的函式叫做 request_mem_region()。

在不需要用到 I/O port 之後，一定要呼叫 release_region() 釋放先前保留的空間，否則在卸載驅動程式後，直到重開機之後，都無法再使用這塊 I/O port：

```
void release_region(resource_size_t start, resource_size_t n);
```

7-3、Memory Mapped I/O

以記憶體讀寫動作代替 I/O port 在硬體層級操縱裝置的手法就稱為「Memory Mapped I/O」(記憶體映射讀寫)，經常縮寫為「MMIO」。

驅動程式只要向預先定義的位址範圍 (MMIO範圍) 進行讀寫動作，就會變成對特定硬體讀寫暫存器的動作。
也因為只要以 C 語言的指標就能簡單辦到，所以成為近代的標準作法。

一般常見的 IA-32 個人電腦架構中，是把緊鄰 4GB 以下的 1GB 範圍當成 MMIO 範圍。

讀寫 MMIO 範圍時，只要用指標就可以了，但是不能直接把記憶體位址的數值當成指標來用，否則可能會造成「kernel panic」。
因為驅動程式是在虛擬記憶體空間運作的緣故，這跟 user space 的應用程式讀寫不正確的記憶體位址而導致「segmentation fault」是一樣的。

驅動程式想讀寫 MMIO 範圍時，必須透過 ioremap() 把物理位址映射到 kernel 的虛擬記憶體空間才行，定義在 「asm/io.h」：
void __iomem * ioremap(unsigned long offset, unsigned long size);
offset 引數是物理記憶體位址，size 引數是以 byte 單位指定的映射範圍大小，Linux kernel 也把物理記憶體位址稱為「bus address」。
函式執行成功的話會傳回 kernel 的虛擬記憶體位址，否則傳回 NULL。

透過 ioremap() 映射的動作，在某些架構上可能會導致問題，這是因為被 cache 影響的緣故。
驅動程式在讀寫位於 MMIO 範圍內的暫存器時，有時會從 cache 讀回先前的資料，而不會真得去讀硬體，寫入資料時也會發生類似的情形，因此擾亂 memory mapped I/O 的工作。
這時就要把用到 ioremap() 的地方改用 ioremap_nocache()：
void __iomem * ioremap_nocache(unsigned long phys_addr, unsigned long size);

在不需要 memory mapped I/O 時，驅動程式必須呼叫 iounmap() 明確取消映射：
void iounmap(volatile void __iomem *addr);

驅動程式範例：

```
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static int sample_init(void) {
char *reg;
printk(KERN_ALERT "sample driver installed.\n");
```

```
reg = ioremap_nocache(0xFEC00000, 4);
if (reg != NULL) {
printk(KERN_ALERT "%x\n", *reg);
iounmap(reg);
}
return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed.\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

讀寫 MMIO 範圍是可以直接 ioremap()，但為了避免衝突，還是建議事先利用 request_mem_region() 保留位址範圍：

```
struct resource *request_mem_region(resource_size_t start, resource_size_t n, const char *name);
```

start 引數是 MMIO 範圍開頭的物理記憶體位址，n引數則是 byte 單位的範圍大小，name 是引數裝置名稱，函式呼叫成功後，會回傳 resource 結構的指標，否則傳回 NULL。
保留成功後，就能在 /proc/iomem 看到裝置名稱。
在不需要讀寫 MMIO 範圍後，呼叫 iounmap() 解除映射後，還需要呼叫 release_mem_region 取消保留：

```
void release_mem_region(resource_size_t start, resource_size_t n);
```

start 引數跟 request_mem_region() 同樣是 MMIO 範圍開頭的物理記憶體位址，n引數則是 byte 單位的大小。

7-4、PCI 裝置的泛用介面

讀寫 PCI 裝置的暫存器時，可以透過 I/O port 也可以透過 MIMO，每個裝置都可能使用不同的讀寫方式，控制這些裝置的驅動程式必須根據裝置來切換讀寫方式。

如此會讓驅動程式的工作變得更複雜，所以 Linux 提供了不必區分 I/O port 與 MMIO 的抽象介面 pci_iomap()，定義在「asm/iomap.h」：

```
void __iomem *pci_iomap(struct pci_dev *dev, int bar, unsigned long maxlen);
```

呼叫後會自動切換 I/O port 或 MMIO 讀寫方式，並傳回指定 BAR(Base Address Register) 的 kernel 虛擬位址。
需要透田 I/O port 讀寫時，內部會用 ioport_map() 進行映射，使它能跟 MMIO 透過同樣的方式讀寫。

不需要讀寫後，要呼叫 pci_iounmap() 解除映射：

```
void pci_iounmap(struct pci_dev *dev, void __iomem *addr);
```

而透過 pci_iomap() 映射的位址讀寫暫存器時，必須使用專用的函式，一樣是定義在「asm/io.h」：

```
unsigned int ioread8(void __iomem *addr);
unsigned int ioread16(void __iomem *addr);
unsigned int ioread32(void __iomem *addr);
unsigned int iowrite8(u8 b, void __iomem *addr);
unsigned int iowrite16(u16 b, void __iomem *addr);
unsigned int iowrite32(u32 b, void __iomem *addr);
```

這些函式會判斷對象是 I/O port 還是 MMIO，並配合做適當的處理。

7-5、以 sparse 做靜態程式碼檢查

前面介紹的函式 prototype 多半都有「__iomem」的標記，查看標頭檔定義：

```
#ifdef __CHECKER__
# define __iomem __attribute__((noderef, address_space(2)))
#else
# define __iomem
#endif
```

平常編譯驅動程式時，沒有定義 __CHECKER__，因此 __iomem 這個巨集的內容是空的，這個巨集要在透過 sparse 這個靜態程式碼檢查工具檢查程式碼時才會起作用。
__iomem 巨集會透過 gcc 擴充功能指定下列屬性：

- 不能透過 *(解參考運算子)存取指標(noderef)
- 位於 I/O 位址空間 (I/O mapped I/O 或 memory mapped I/O; address_space(2))

address_space() 括弧內的數字意義：

數值	意義
0	Kernel space
1	User space
2	I/O space

sparse 安裝完成後，就可以建構驅動程式，以往都是在命令列執行「make」，現在要多指定「C=1」或「C=2」，「C=1」會檢查需要重新編譯的檔案，「C=2」則會檢查所有的檔案。

在執行「make C=2」編譯 7-3 的範例程式後，會出現些許 warning，需修改程式碼如下：
char *reg; => char __iomem *reg;
printk(KERN_ALERT "%x\n", *reg); => printk(KERN_ALERT "%x\n", ioread8(reg));

7-6、記憶體屏障

C語言的程式被編譯器轉成組合語言，最後轉成機械語言才能在 CPU 上執行，而編譯器為了提昇程式執行的效率，有時會交換指令的執行順序。

但進這種調換法對驅動程式來說卻很有問題，在使用 MMIO 讀寫暫存器時，可能會造成意外的行為，如同之前提過，MMIO 是透過指標去存取裝置的暫存器，但編譯器只看指標存取動作而已。

比如說在準備執行 DMA (Direct Memory Access) 時，驅動程式常會用到下面這種寫法：

```
*a = ptr; /* 把 buffer 的位址寫入暫存器 */
*b = len; /* 把 buffer 容量寫入暫存器 */
*c = 1; /* 開始 DMA */
```

先把 buffer 準備好後，再透過暫存器啟動 DMA，一定要到最後才能寫入「*c」，但如果經過編譯器的調換：

```
*b = len; /* 把 buffer 容量寫入暫存器 */
```

```
*c = 1; /* 開始 DMA */
*a = ptr; /* 把 buffer 的位址寫入暫存器 */
```
則會造成在 buffer 還沒準備好的情形下開始 DMA，有可能讓系統當掉。

為了應用這種情形， kernel 提供了「 Memory Barrier」(記憶體屏障) 的機制。

barrier()
barrier() 巨集會插入 memory barrier，gcc 不會把指令移過 memory barrier。

wmb()
wmb (Write Memory Barrier) 是使記憶體寫入順序維持與 C 原始碼一致的巨集。

rmb()
rmb (Read Memory Barrier) 能保証在這行之前的「記憶體讀取工作」已經全部執行完成。

mb()
mb (Memory Barrier) 這個巨集同時具備 wmb() 與 rmb() 的功能。

7-7、volatile

C語言的資料型態有「volatile(揮發性)」這麼一個修飾詞，為變數加上 volatile 修飾詞，可以：

- 防止記憶體存取動作的最佳化
- 防止刪除記憶體存取動作

具體例子：

```
#include

int main(void) {
int val, n;
int *p;

val = 2008;
p = &val;
```

```
n = *p;

printf("%d\n", val);
return 0;
}
```

這個例子把 *p 指標向向的值存到 n 之內，但是卻沒用到 n 變數，因些編譯器可能會把存入 n 的動作刪除，但如果 *p 指標是 MMIO 位址的話，或許這代表「從裝置讀取暫存器內容」的動作，於是少掉這個動作的話，硬體可能就不會一如預想地運作，所以要在 MMIO 的指檔變數要加上「volatile」才行：

```
volatile int *p
```

另一個例子：

```
#include

int main(void) {
int val, n;
int *p;

val = 2008;
p = &val;
n = *p;
n = *p;
n = *p;

printf("%d\n", val);
return 0;
}
```

這個例子連續三次寫入 n 變數，編譯器最佳化時，可能把這些寫入動作整合成一個，但如果 *p 是 MMIO 位址的話，就可能造成問題，因此 *p 指標變數應該要加上 volatile 才對。

編譯器有沒有產生適切的組合語言碼，可以透過「gcc -S」來檢視組合語言碼(*.s)來判斷。

7-8、行內組譯

有時光靠 C 語言無法控制硬體，還是需要讓組合語言上場，本節就在說明 C 語言原始碼內撰寫組合語言的方式。

asm 關鍵字

在 C 語言的原始碼內插入組合語言原始碼的動作稱為「Inline assembler(行內組譯)」，而行內組譯的關鍵字是「asm」。
asm 關鍵字的語言如下：

```
asm("組合語言指令碼"
: 輸出暫存器
: 輸入暫存器
: 會變更的暫存器
```

檢查組合語言碼

要檢查產生的組合語言碼正確與否的話，可修改驅動程式的 Makefile，在 CFLAGS 選項加上「-S」。

稍微複雜的 asm 關鍵字

7-9、結語

控制硬體是驅動程式的主要工作，基本上透過指標讀寫硬體的暫存器就行了，但是編譯器在最佳化時有一些陷阱，開發時必須特別注意這些情形。

八、記憶體

在 Kernel 內管理記憶體的方式與一般應用程式不同，所以驅動程式需要取用記憶體時，必須要知道 kernel 管理記憶體的方式及其特性。
本章會解說 kernel 記憶體的用法，以及驅動程式經常用到的 DMA 功能。

8-1、Linux kernel 的記憶體管理

最近的 OS 都以「paging」的方式來管理記憶體， Linux 也是其中之一。
Paging 是利用 CPU 與 MMU(Memory Management Unit) 的硬體機制實現的。
Paging 可以讓作業系統實現下列功能：

- Virtual Memory (虛擬記憶體)

- Swap (置換檔)
- Demand Paging (按需求分頁；用到時才載入資料)
- Copy on Write (只有資料有更動時才實際複製)
- Shared Memory (共享資料)
- mmap (將磁碟內容映射到記憶體內)

特別是 virtual memory 可以讓 kernel 與一般應用程式在虛擬位址空間內執行。
虛擬位址轉成物理位址(實體位址) 的工作由硬體完成，轉換速度夠快。

Paging 指的是把物理記憶體切割成長度固定的「page」，並以此作為管理單位。
一個 page 的大小隨著硬體架構而不同， IA-32 的 page size 是 4 KB。
物理記憶體固定以 page size 當成管理單位，也就是說，就算只要求分配 1 byte 記憶體，還是會分配到最小單位，也就是 1 page。
系統啟動時，OS 會從 BIOS 得知系統安裝了多少物理記憶體。
OS 以 page 為管理物理記憶體的單位，實作了「buddy system(buddy allocator)」。
而 Buddy system 當然就是以 page size 為單位管理物理記憶體，以 2 的冪次方控制單位數量。
例如，當 OS 內的 module 要求 128 bytes 記憶體時，會回傳 1 page 的記憶體，如果要求 4097 bytes 記憶體時，就會回傳 2 pages 記憶體，因為是以 page 當成管理單位，所以有著出現無謂浪費的缺點。
在分配 2 pages 以上的記憶體時，得到的物理記憶體空間會是連續的，適合拿來進行 DMA 傳送。

Buddy system 的空間使用狀態可以透過以下指令查詢：
cat /proc/buddyinfo
Buddy system 因為只能以 page size 當成單位，所以在驅動程式需備記憶體時，是個不怎麼好用的介面，因此在 buddy system 上層，還準備了「Slab allocator」介面。
slab allocator 是以稱為「cache」的單位來管理記憶體，slab allocator 是比較容易使用的介面，驅動程式常用的 kmalloc() 及 kfree() 就是它所提供的。

8-2、Stack
Stack 是存放暫時性資料的記憶體空間，這邊會說明 kernel 的 stack 管理機制。

8-2.1、Kernel Stack

一般應用程式使用區域變數時，會在 user space 的 stack 範單取用記憶體，而驅動程式也是一樣，使用區域變數、管理返回位址的時候，就需要用到 kernel 內的 stack 範圍。
kernel stack 是透過 thread_info 結構管理的，緊鄰分配於 thread_info 結構 (include/linux/sched.h)：

```
union thread_union {
struct thread_info thread_info
unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

THREAD_SIZE 是 thread_info 結構與 stack 合在一起的大小，在 IA-32 環境下預設是「8KB」。
kernel stack 是從記憶體的高位址側往低位址側使用的，kernel 內部如果發生 stack overflow 的話，就會破壞 process 結構。
Stack 大小設為 8KB 乍看之下或許很小，但其實已經十分足夠，一般來說 kernel 之內的程式碼，為了抑制記憶體使用量，會極力避免使用 stack 才對。

user process 呼叫 system call，讓驅動程式在 user context 運作的時候，就會使用 user process 定義的 kernel stack。
那 interrupt context 又是怎樣的情況呢？發生中斷時，驅動程式所在的 interrupt context 沒有對應的 user process，這時中斷處理程序會拿發生中斷時「正在執行」的 user process 擁有的 kernel stack 來用。
呼叫函式會使用 kernel stack 空間，因此了函式都是 reentrant(可重進入) 的。

8-2.2、4K Stacks

8KB 的 kernel stack 有個問題：

- 8192 bytes 的大小無法放入 1 page

所以 kernel 配直選項能讓它變成 4096 bytes，剛好能放進一個 page，稱為「4K Stacks」機制：

- Kernel hacking -> Use 4Kb for kernel stacks instead of 8Kb

在開發驅動程式時，為了避免因為 stack overflow 而造成 kernel panic，必須特別注意：

- 盡量不要使用區域變數
- 不要套疊呼叫太多層函式

8-3、全域變數

驅動程式可以一如往常地宣告、使用全域變數，全域變數放在 RAM 之內，只要不加上 const 就可自由讀寫，需注意的有：

- 全域變數會在整個驅動程式之內共享
- 對編譯成 module 的驅動程式而言，其它 module 看不到它的全域變數；但可用 EXPORT_SYMBOL 對外公開
- 對靜態連結到 kernel 的驅動程式而言，全域變數是整個 kernel 都看得到的

以下面這個驅動程式為例，假設它是 kernel module 類型的驅動程式：

```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

int g_sample_counter __attribute__ ((unused));
const int g_sample_counter2 __attribute__ ((unused));
static int g_sample_counter3 __attribute__ ((unused));
int g_sample_counter4 __attribute__ ((unused));
EXPORT_SYMBOL (g_sample_counter4);

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed.\n");
return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

定義了四個全域變數，接著來看外部 module 會看到哪些東西。
因為這些個變數都沒在程式碼內用到，為了不讓 gcc 自動把它們刪除，所以加上「unused」屬性。

Makefile 如下，因為希望建立 link map，所以在 LDFLAGS 環境變數加上「-Map」選項：

```
CFLAGS += -Wall
LDFLAGS += -Map $(PWD)/sample.map
CFILES := main.c

obj-m += sample.o
sample-objs := $(CFILES:.c=.o)

all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

建構驅動程式後，會產生映射檔(sample.map)。
因為 g_sample_counter3 變數宣告為 static，所以命名範圍限定在 main.c 檔案內，因此配置在「BSS(Block Started by Symbol」區段內的變數只有三個：

```
cat sample.map | grep g_sample_counter
```

建構時產生的「Module.symvers」包含被 EXPORT_SYMBOL 的符號清單，這些符號可被其它 module 取用：

```
cat Module.symvers | grep g_sample_counter
```

最後把驅動程式掛進 kernel，再看看 kernel 的符號表：

```
cat /proc/kallsyms | grep sample
```

全域變數是一個驅動程式的 module 內共用的，如果在許多不同的 context 之內存取的話，就必須作好鎖定才行。

8-4、動態取得記憶體

linux kernel 提供了許多種不同的記憶體管理機制，所以在取用記憶體時，要先弄清楚它們之間的差異。

8-4.1、kmalloc

取用記憶體時，驅動程式最常用的方式應該是 kmalloc()，定義在 linux/slab.h 內，但不必特地 include。

```
void *kmalloc(size_t size, gfp_t flags);
```
kmalloc() 會取得 size bytes 的 kernel 記憶體，適合用來取得 128 KB 左右的小 buffer。
取用記憶體失敗時，會傳回 NULL 指標，所以一定要檢查傳回值。
記憶體用完後，一定要呼叫 kfree() 釋放，否則會在 kernel 內引發 memory leak，驅動程式的 memory leak 並不會在卸除驅動程式後由 kernel 代為釋放。
```
void *kfree(const void *block);
```

使用 kmalloc() 的範例：
```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

void *memptr;

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed.\n");
memptr = ptr = kmalloc(2, GFP_KERNEL):
printk("ptr %p\n", ptr);
return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed\n");
kfree(memptr);
}

module_init(sample_init);
module_exit(sample_exit);
```

8-4.2、直接從 Cache 取用記憶體

kmalloc() 與 kfree() 都是以「slab allocator」這個 cache 機制實作出來的，這邊的「cache」指的是「一塊記憶體」的意思。

slab allocator 的下層是之前提過的 buddy system 記憶體管理機制，slab allocator 利用 buddy system 提供更靈活的記憶體管理功能。

8-4.3、直接從 Cache 取用記憶體

若要直接從 cache 取得記憶體，首先要以 kmem_cache_create() 建立最初的 cache 容器：

struck kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align, unsigned long flags, void (*ctor)(void *, struct kmem_cache *, unsigned long));

引數 name 指的是在 /proc/slabinfo 或 /sys/slab 中讓其它人看到的名稱。
引數 size 指的是想取用的記憶體量(bytes)。
其它的引數不常用到，填 0 或 NULL 都無所謂。
呼叫 kmem_cache_create() 後，並不會實際配置記憶體，這個 cache 對 OS 來說還是「空」的，若呼叫失敗時，則會傳回 NULL值。
不用要用到後，需須叫 kmem_cache_destroy() 將之釋放：

void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);

如果要在 cache 內配置記憶體，可以呼叫 kmem_cache_alloc()：

void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);

函式呼叫成功的話，代表成功配置到 kmem_cache_create() 要求的記憶體空間，並傳回指向這塊記憶體開頭的指標，若呼叫失敗時，則會傳回 NULL 。
要釋放記憶體，可以呼叫 kmem_cache_free()：

void kmem_cache_free(struct kmem_cache *cachep, void *disp);

使用 kmem_cache_alloc() 的範例：
略

8-4.4、Buddy System

Linux 記憶體管理的最低層是「buddy system」，這是 slab allocator 的下層。
驅動程式在需要取用剛好等於 page size 的記憶體時，可以直接使用這個介面。
唯 buddy system 只能以 page size 為單位管理記憶體，並以 order 作為 page 的計量單位，取得 2 的 order 次方個 page。
要從 buddy system 取用記憶體時，是呼叫 __get_free_pages()：

unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order);

order 引數是要取用的數量，若 order 為 3，則會取用 2 的 3 次方，也就是 8 pages 的記憶體，page size 隨系統架構而不同。
函式呼叫成功的話，會傳回記憶體指標，傳回值的型別是 unsigned long，必須明確轉成「void *」或「char *」之類的型別。
釋放記憶體時，則是呼叫 free_pages()：

void free_pages(unsigned long addr, unsigned int order);

使用 __get_free_pages() 的範例：
略

8-4.5、vmalloc
slab allocator 保證取到的記憶體範圍是連續的物理記憶體，可以用來進行 DMA 傳送工作。但在某些情況下，並不需要連續的物理記憶體，只要 kernel 虛擬記憶體空間內的位址連續就好，此時可以使用 vmalloc() 及 vfree() 函式：

```
void *vmalloc(unsigned long size);
void vfree(void *addr)
```

8-5、檢查 Slab Allocator 的洩漏情形
略

8-6、kmalloc 檢查偵測 patch
略

8-7、DMA
驅動程式在裝置與記憶體之間轉移資料時，經常用到 DMA(Direct Memory Access) 的機制，它代表：

- 在裝置與物理記憶體之間傳送資料時，不必 CPU 插手

的機制。
也就是說，只要用了 DMA， CPU 就可以去做其它的事，可以提昇系統效能。

8-7.1、DMA Buffer 的限制
因為裝置需要直接讀寫系統記憶體，所以一般來說 DMA Buffer 的限制有：

- DMA buffer 的記憶體範圍必須是連續的
- DMA buffer 的物理開頭位址必須位於 page 的邊界起點
- DMA buffer 的記憶體空間必須在開頭 4GB 之內
- DMA buffer 的記憶體不能被 swap

8-7.2、取得 DMA Buffer 的 Kernel 函式
略

8-8、快取一致性
執行 DMA 傳送工作時，有時會遇到「cache coherency(快取一致性)」的問題。
一般來說，電腦架構為了提高記憶體讀寫效率，會利用 CPU 內的 L1/L2 cache，所以驅動程式對記憶體寫入資料時，不見得會立刻反應到記憶體內容上，而是會放在 CUP 的 cache 裡，等到適當時機才寫入記憶體。
相同的情況下，由裝置寫入資料到記憶體時，CPU 的 cache 也未必能及時更新，而可能導致驅動程式讀取 cache 時，讀到舊的資料。

linux kernel 為了應用這種問題，提供了控制 cache 的函式：
void dma_cache_sync(struct device *dev, void *vaddr, size_t size, enum dma_data_direction direction);
引數 vaddr 是 DMA buffer 的指標(kernel 虛擬位址)。
引數 size 是想 flush/purge 的 DMA buffer 大小。
dma_data_direction 是列舉常數，指定 DMA_TO_DEVICE 代表「要把資料從記憶體傳給裝置」，也就是執行 flush cache 的動作。
指定 DMA_FROM_DEVICE 代表「要讀取裝置傳到記憶體的資料」，也就是執行 purge cache 的動作。

IA-32 在硬體層及保証 cache coherency，所以驅動程式不必特別控制 cache，但若要移植到其它平台時，則需特別注意。

8-9、結語
開發驅動程式要應對各種不同的記憶體管理機制，依用途選最適合的方式。
而 DMA buffer 有諸多限制，開發時需注意。

九、計時器
介紹驅動程式特有的時間管理機制。

9-1、Kernel 的計時器管理

kernel 管理的 timer 大致分成兩類：

- 現在的日期與時間
- jiffies

其中 jiffies 指的就是 kernel 的 timer。

9-2、現在的日期與時間

「現在的日期與時間」指的是 1970 年 1月1日0時0分0秒至今為止的秒數，進而換算成日期與時間。

9-2.1、do_gettimeofday()

在 kernel 內想取得目前時刻，可以呼叫 do_gettimeofday()：

```
void do_gettimeofday(struct timeval *tv);
```

timeval 結構定義在「linux/time.h」標頭檔內：

```
struct timeval {
time_t tv_sec; /* seconds */
suseconds_t tv_usec; /*microseconds */
};
```

do_gettimeofday 的範例：

```
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static int sample_init(void) {
struct timeval now;
suseconds_t diff;
volatile int i;
printk(KERN_ALERT "devone driver installed: %s\n", __func__);
```

```
do_gettimeofday(&now);
diff = now.tv_usec; /* microseconds */

printk("Current UTC: %lu (%lu)\n", now.tv_sec, now.tv_usec);
for (i = 0; i < 999; i++)
;

do_gettimeofday(&now);
diff = now.tv_usec - diff;
printk("Elapsed time: %lu\n", diff);

return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "devone driver removed.\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

9-2.2、2038 年問題

如果以 32-bit 變數存放經過的秒數，則會在 2038年1月19日 產生溢位，有可能導致程式運作出錯，稱為「2038年問題(Year 2038 problem; Y2K38)」。
解決的方式之一，是把 time_t 改成 64-bit，Linux 也是採用這個解法。

9-3、jiffies

OS 內部會使用 timer interrupt 定期執行某些 kernel 的工作。
timer interrupt 是使用硬體的計時功能，在指定的時間間隔向 OS 發出 interrupt。
linux 在收到 timer interrupt 時，會遞增 jiffies 變數。

9-3.1、HZ 巨集

前面提到 jiffies 是 timer interrupt 的次數紀錄。
timer interrupt 的間隔是由 HZ 巨集決定的，HZ 的意義就是「hertz」。

```
#ifdef __KERNEL__
define HZ CONFIG_HZ /* Internal kernel timer frequency */
define USER_HZ 100 /* .. some suer interfaces are in "ticks" */
define CLOCKS_PER_SEC /* like times() */
#endif
```

假設 HZ 的值是 100，這樣代表每秒 timer interrupt 會發生 100 次，也就是每 0.01 秒發生一次：

- Timer interrupt 發生間隔 = 1/HZ (秒)

增加 HZ 的值時，就會增加 timer interrupt 的次數，也就是增加 kernel 內部需要處理的工作量。
kernel 內部需要定期處理各種工作，這些工作就是由 timer interrupt 定期引發的。
要縮短 kernel 的反應間隔，方法只有增加 timer interrupt 一途，但這樣會讓 kernel 消耗更多 CPU 資源。

9-3.2、遞增計數

實際增加 jiffies 的工作是由 do_timer() 完成的。

```
void do_timer(unsigned long ticks) {
jiffies_64 += ticks;
update_times(ticks);
}
```

9-3.3、驅動程式的取用方式

驅動程式想取用「jiffies」或「jiffies_64」，必須先 include 進「linux/jiffies.h」。
不過只要 include 進 linux/module.h 就會一併載入 jiffies.h ，所以不必明確 include 進 jiffies.h。

使用 jiffies 的範例：

```
#include
#include
```

```
MODULE_LICENSE("Dual BSD/GPL");

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed: %lu\n", jiffies);
return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed.\n");
}

module_init(sample_init);
module_exit(sample_exit);
}
```

9-3.4、497日問題

jiffies 會隨會時間不斷向上遞增，如果 HZ 巨集的時間設為「100」的話，每 10 毫秒就會有一次 timer interrupt，如果 jiffies 的值增加了「200」的話，就知道是經過「2秒」。
jiffies 定義為 unsigned long，在 IA-32 上是 32-bit 整數，而32-bit 整數的最大值是 4294967295(2^32-1)，所以

- 4294967295/100 => 42949672(秒)/86400 => 497(日)

這就稱為「497日問題」。

驅動程式如果用到 jiffies 的話，一定要特別注意這個問題，應對方式為：

- 以 unsigned long 變數儲存 jiffies
- 計算 timeout 時，只從 jiffies 減去先前的值

程式碼範例為：

```
#define OVER_TICKS (3*HZ)
unsigned long start_time = jiffies;
/* .... */
if (jiffies - start_time >= OVER_TICKS) {
... /* 經過三秒後要做的事 */
```

```
}
```

9-3.5、jiffies 初始值

linux 2.4 的 jiffies 是以 0 作為初始值，也知道 jiffies 會在第 497 天 overflow。
但為了徹底篩選出沒有處理 overflow 的程式碼， linux 2.6 開始把 jiffies 的初始值設為：「-300秒」。
也就是說，只要五分鐘就會 overflow(變成 0 的時候就會 overflow)。

9-4、等待

有時為了讓硬體完成工作，驅動程式需要等待一段固定時間，此時可以使用定義在 linux/delay.h 的函式：

```
void mdelay(unsigned long msecs);
void udelay(unsigned long usecs);
void ndelay(unsigned long nsecs);
```

延遲的單位分別為毫秒(10^-3)、微秒(10^-6)、奈秒(10^-9)。

這些等待函式，在等待的過程中都會佔住 CPU，不會讓 CPU 去處理其它工作，這樣子的方式，稱為「busy waiting(忙錄等待)」。
系統如果只有一個 CPU 的話，等待時，會讓整個 OS 停住(但是可以處理 interrupt)，所以一般的認知為：

- 不要長時間 busy waiting

busy waiting 的好處是可以在 interrupt context 使用。

busy waiting 的範例：

```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed: %lu\n", jiffies);

mdelay(100);
```

```
udelay(1000);
ndelay(1000);

printk(KERN_ALERT "%lu (HZ %d)\n", jiffies, HZ);
return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed.\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

9-4.1、不忙碌的等待

如果在等待過程中，希望 CPU 去做其它事的話，可以用定義在「linux/delay.h」中 sleep 系列的等待函式：

```
void ssleep(unsigned int seconds);
void msleep(unsigned int msecs);
unsigned long msleep_interruptible(unsigned int msecs);
```

延遲的單位分別為秒、毫秒(10^-3)、毫秒(10^-3)。
msleep_interruptible() 與 msleep() 不同的地方在於等待過程可能被中斷，中斷指的是 process 可以接收 signal。
如果在 msleep_interruptible() 中收到 signal 而中斷等待的話，則會回傳距離原始時限的時間(正值)，否則的話傳回「0」。

這幾個函式都會使 driver context 進入 sleep，所以不能在 interrupt context 中使用，因為 interrupt context 位於最高級，如果在 interrupt 處理時式之內 sleep 的話，就不會執行其它 process，也不會有其它人來叫醒 interrupt 處理程式，會造成全系統鎖死。

sleep() 的範例：

```
#include
#include
```

```
MODULE_LICENSE("Dual BSD/GPL");

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed: %lu\n", jiffies);
#if ssleep
ssleep(10);
#else
ret = msleep_interruptible(10*1000); /* sleep for 10 seconds */
#endif

printk(KERN_ALERT "%lu (HZ %d)\n", jiffies, HZ);
return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed.\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

在執行 ssleep() 時，終端就就算按「Ctrl+C」鍵送出訊號也不會造成中斷。
以 ps 來看 insmod 的狀態，會發現是「D(uninterruptible sleep)」。
而在執行 msleep_interruptible() 時，若按「Ctrl+C」鍵送出中斷訊號，則會發現 process 被中斷了，並回傳正值。
以 ps 來看 insmod 的狀態，會發現是「S(interruptible sleep)」。

9-5、Kernel Timer

驅動程式在對硬體送出 DMA 指令後，為了監視 DMA 執行完成，有時需要在一定時間過後呼叫某些函式，linux kernel 有為這類計時需求提供機制，這個機制就是「kernel timer」。
舉例來說，驅動程式對 I/O 裝置下令開始 DMA 後，通常在 I/O 裝置完工後，都會送出 interrupt 讓驅動程式繼續作處理。

不過驅動程式有個鐵則，那就是：

- 不能假定硬體一定能正確完成工作

I/O 裝置有可能故障或發生其它問題，這時也不能讓 OS 當掉，必須重新，讓應用程式可以繼續執行，或是通知應用程式硬體發生問題。
應用程式對驅動程式發出請求的情況下，驅動程式必須對這個請求送出回應，如果沒有在時限內收到 interrupt，必須能偵測出這種情形，並將錯誤碼回報給應用程式。

9-5.1、Kernel Timer 的使用方式

kernel timer 的使用方式為：

- 準備 timer_list 結構變數
- 撰寫 timeout 處理函式
- 設定時限並啟動 timer

其中 timer_list 結構變數不能是區域變數，必須是全域變數或配置在 heap 之內，timer_list 結構定義於「linux/timer.h」，主要成員包含：

- expires
- function
- data

timer_list 的初始化工作必須使用 kernel 提供的 init_timer() 函式：

```
void init_timer(struct timer_list *timer);
```

Timeout 處理函式的 prototype 如下：

```
void (*function) (unsigned long);
```

它會收到一個 unsigned long 型別的引數，如果要傳遞許多引數時，使用結構指標即可。

「expires」成員是時限(單位是 jiffies)，也就是呼叫 timeout 處理函式的時間，舉例來說，如果要在五秒後呼叫 timeout 函式的話：

```
expires = jiffies + 5*HZ
```

雖然沒有考慮到 overflow 的情形，不過 kernel 會做適當的處理。
呼叫 add_timer() 就會開始計時：

```
void add_timer(struct timer_list *timer);
```

Timer 範例程式：

```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");
```

```
#define TIMEOUT_VALUE (10*HZ)

static struct timer_list tickfn;

static void sample_timeout(unsigned long arg) {
struct timer_list *ptr = (struct timer_list *)arg;
printk("ptr %p (%s) \n", ptr, __func__);
}

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed.\n");

init_timer(&tickfn);
tickfn.function = sample_timeout;
tickfn.data = (unsigned long)&tickfn;
tickfn.expires = jiffies + TIMEOUT_VALUE;
add_timer(&tickfn);

return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed.\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

試著編譯、載入後，注意 syslog，可以看到剛好過了 10 秒後呼叫 timeout 函式。
通常在經過指定時間後，不會剛好在「那一刻」呼叫 timeout 函式，因為此時， kernel 可能在忙著處理其它事情，就沒辦法立刻轉來呼叫 timeout 函式，因為 kernel 並不是 preemptive。

9-5.2、卸除驅動程式時必須注意

上述範例，有個致命的問題，如果在 kernel 呼叫 timeout 函式之前，卸除驅動程式的話，會發生什麼事呢？
kernel panic。

因此在卸除驅動程式時，如果有以下情形的話：

- 還有 timeout 函式沒被呼叫
- 正在執行 timeout 函式

就必須先拿掉 timeout 函式、或將之停止才行。

為此，kernel 提供了 del_timer_sync() 函式：
int del_timer_sync(struct timer_list *timer);
回傳值有兩種形式：

傳回值	意義
0	成功卸除函式，而 timeout 函式已被呼叫並執行完畢
正值	在呼叫 timeout 函式之前已將之卸除

使用這個函式時，須注意：

- 不能在 interrupt context 之內呼叫
- 不能在拿著 spin lock 時呼叫

因此，結論是，在驅動程式使用 kernel timer 時，一定要記得讓 add_timer() 與 del_timer_sync() 成對出現。

修正過的 Timer 範例程式：

```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

#define TIMEOUT_VALUE (10*HZ)

static struct timer_list tickfn;
```

```
static void sample_timeout(unsigned long arg) {
struct timer_list *ptr = (struct timer_list *)arg;
printk("ptr %p (%s) \n", ptr, __func__);
}

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed.\n");

init_timer(&tickfn);
tickfn.function = sample_timeout;
tickfn.data = (unsigned long)&tickfn;
tickfn.expires = jiffies + TIMEOUT_VALUE;
add_timer(&tickfn);

return 0;
}

static void sample_exit(void) {
int ret;
ret = del_timer_sync(&tickfn);
printk(KERN_ALERT "sample driver removed (%d)\n", ret);
}

module_init(sample_init);
module_exit(sample_exit);
```

9-5.3、Kernel Timer 的 Context

timeout 函式會在 interrupt context 執行，所以 timeout 函式需注意：

- 不能呼叫會 sleep 的函式
- 不能使用 user context 的 spin lock
- 如果有多個 CPU，則可能同時執行多個 timeout 函式

9-5.4、Interval Timer

kernel timer 是 one-shot timer，也就是發生一次 timeout 之後就不會再呼叫第二次 timeout 函式，如果要達到每隔一段時間連續呼叫的話，可以：

- 在 timeout 函式的最後，呼叫 mod_timer() 重設 timer：
  -
  - int mod_timer(struct timer_list *timer, unsigned long expires); 函式回傳值有兩種形式，不過都不代表失敗情形：

	○ 意義
	○ timer 是「inactive」，要重設的 timer 還沒加進 kernel
	○ timer 是「active」，要重設的 timer 已經加進 kernel

  -
  - 事實上， mod_timer() 的處理內容其實與下面這樣程式碼相等：
  - del_timer(timer);
  - timer->expires = expires;
  - add_timer(timer);
  -
  - Interval timer 範例程式：
  - #include
  - #include
  -
  - MODULE_LICENSE("Dual BSD/GPL");
  -
  - #define TIMEOUT_VALUE (10*HZ)
  -
  - static struct timer_list tickfn;
  -
  - static void sample_timeout(unsigned long arg) {
  - struct timer_list *ptr = (struct timer_list *)arg;
  - int ret;
  - ret = mod_timer(tick, jiffies + TIMEOUT_VALUE);
  - printk("ptr %p (%s) \n", ptr, __func__);
  - }
  -
  - static int sample_init(void) {
  - printk(KERN_ALERT "sample driver installed.\n");

```
init_timer(&tickfn);
tickfn.function = sample_timeout;
tickfn.data = (unsigned long)&tickfn;
tickfn.expires = jiffies + TIMEOUT_VALUE;
add_timer(&tickfn);

return 0;
}

static void sample_exit(void) {
int ret;
ret = del_timer_sync(&tickfn);
printk(KERN_ALERT "sample driver removed (%d)\n", ret);
}

module_init(sample_init);
module_exit(sample_exit);
```

可以利用 cmd: cat /proc/kmsg 或 cat /var/log/messages 來查看 printk 的 log 訊息。

9-5.5、注意「錯過」情形

使用 kernel timer 時，必須特別注意「錯過」的情形，也就是「race condition」或簡稱「racing」。

舉例來說，在 user process 送出 read 請求，準備讀取資料的情形後，驅動程式收到 read 請求，會接著向硬體(I/O裝置) 送出 DMA 指令，並進入 sleep 狀態，等到 DMA 完成，硬體會發出 interrupt 通知驅動程式，接著驅動程式開始讀取資料，並把資料送給 user process。

- 在驅動程式進入 sleep 狀態的同時，會有一個 timer ，如果超過一定的時間，還沒被硬體發出的 interrupt 喚醒，則會在 timeout 的情況下被喚醒。
- 這個例子硬體所發出的 interrupt 來喚醒驅動程式，和驅動程式因為 timeout 情況下被喚醒，就是造成 racing 的兩個原因。
- 由於並不知道 interrupt 何時會發生，所以設計時，必須考慮不同的時機，避免競爭狀態才行。

舉一個 character 類型的驅動程式，讀取裝置檔時會不斷傳回「0xFF」，透過 timer interrupt 來模擬。

- 載入驅動程式的步驟如下：
    - 以 insmod 載入：# insmod ./devone.ko
    - 在 syslog 查閱 major number 並紀錄
    - 建立裝置檔：# mknod /dev/devone c 0
    - 讀取裝置檔：# hexdump -C -n 16 /dev/devone
    - 卸除驅動程式：# rmmod devone
-
- 驅動程式可以接收引數，修改 interrupt 的時機與等待時限，比如說，這樣執行的話，會將 timeout 的值設為3 秒、interrupt 發生的時間設為 10 秒：
- # insmod ./devone.ko timeout_value=3 irq_value=10
-
- 以預設的情形執行，就是將 timeout 設為 10 秒，IRQ 在 3 秒時發生；如果要模擬讀取資料超過時限的情形，就是更改 timer 與 IRQ 設定的方式來執行：
- #sudo /sbin/insmod ./devone.ko
- #sudo /sbin/insmod ./devone.ko timeout_value=3 irq_value=10
-
- 以下列指令來確認讀取資料時情形：
- sudo tail /var/log/messages
- sudo mknod /dev/devone c 250 0
- hexdump -C -n 16 /dev/devone
- sudo tail /var/log/messages
- sudo /sbin/rmmod devone
- sudo tail /var/log/messages
-
- 考慮到 racing 的 timer 範例程式：
- #include
- #include
- #include
- #include
- #include
- #include
- #include
- #include
- #include
-
- MODULE_LICENSE("Dual BSD/GPL");
-
- static unsigned int timeout_value = 10;
- static unsigned int irq_value = 3;

```
module_param(timeout_value, uint, 0);
module_param(irq_value, uint, 0);

static int devone_devs = 1;
static int devone_major = 0; /* 0=dynamic allocation */
static struct cdev devone_cdev;

struct devone_data {
struct timer_list timeout;
struct timer_list irq;
spinlock_t lock;
wait_queue_head_t wait;
int dma_done;
int timeout_done;
};

static void devone_timeout(unsigned long arg) {
struct devone_data *dev = (struct devone_data *)arg;
unsigned long flags;

spin_lock_irqsave(&dev->lock, flags);
printk("%s called\n", __func__);
dev->dma_done = 1;
wake_up(&dev->wait);
spin_unlock_irqrestore(&dev->lock, flags);
}

static void devone_irq(unsigned long arg) {
struct devone_data *dev = (struct devone_data *)arg;
unsigned long flags;

spin_lock_irqsave(&dev->lock, flags);
printk("%s called\n", __func__);
dev->dma_done = 1;
wake_up(&dev->wait);
spin_unlock_irqrestore(&dev->lock, flags);
}

```

```
static void devone_dma_transfer(struct devone_data *dev) {
dev->dma_done = 0;
mod_timer(&dev->irq, jiffies + irq_value*HZ);
}

ssize_t devone_read(struct file *filp, char __user *buf, size_t count, loff_t
*f_pos) {
printk(KERN_ALERT "%s called\n", __func__);
struct devone_data *dev = filp->private_data;
int i;
unsigned char val = 0xff;
int retval;

/* start timer */
dev->timeout_done = 0;
mod_timer(&dev->timeout, jiffies + timeout_value*HZ);

/* kick DMA */
devone_dma_transfer(dev);

/* sleep process with condition */
wait_event(dev->wait, (dev->dma_done == 1) || (dev->timeout_done ==
1));

/* cancel timer */
del_timer_sync(&dev->timeout);

if (dev->timeout_done == 1)
return -EIO;

/* store read data */
for (i = 0; i < count; i++) {
if (copy_to_user(&buf[i], &val, 1)) {
retval = -EFAULT;
goto out;
}
}
retval = count;
out:
```

```
return (retval);
}

int devone_open(struct inode *inode, struct file *filp) {
struct devone_data *dev;

dev = kmalloc(sizeof(struct devone_data), GFP_KERNEL);
if (dev == NULL)
return -ENOMEM;

/* initialize members */
spin_lock_init(&dev->lock);
init_waitqueue_head(&dev->wait);
dev->dma_done = 0;
dev->timeout_done = 0;

init_timer(&dev->timeout);
dev->timeout.function = devone_timeout;
dev->timeout.data = (unsigned long)dev;

init_timer(&dev->irq);
dev->irq.function = devone_irq;
dev->irq.data = (unsigned long)dev;

filp->private_data = dev;

return 0;
}

int devone_close(struct inode *inode, struct file *filp) {
struct devone_data *dev = filp->private_data;

if (dev) {
del_timer_sync(&dev->timeout);
del_timer_sync(&dev->irq);
kfree(dev);
}
return 0;
}
```

```
struct file_operations devone_fops = {
.open = devone_open,
.release = devone_close,
.read = devone_read,
};

static int sample_init(void) {
dev_t dev = MKDEV(devone_major, 0);
int ret;
int major;
int err;

ret = alloc_chrdev_region(&dev, 0, devone_devs, "devone");
if (ret < 0)
return ret;
devone_major = major = MAJOR(dev);

cdev_init(&devone_cdev, &devone_fops);
devone_cdev.owner = THIS_MODULE;
devone_cdev.ops = &devone_fops;
err = cdev_add(&devone_cdev, MKDEV(devone_major, 0), 1);
if (err)
return err;

printk(KERN_ALERT "devone driver(major %d) installed.\n", major);
printk(KERN_ALERT "timeout %u irq %u timer (%s)\n", timeout_value,
irq_value, __func__);

return 0;
}

static void sample_exit(void) {
dev_t dev = MKDEV(devone_major, 0);
printk(KERN_ALERT "devone driver removed.\n");
cdev_del(&devone_cdev);
unregister_chrdev_region(dev, devone_devs);
}
```

- module_init(sample_init);
- module_exit(sample_exit);
- 目前執行起來， read() 是無法 work 的，不確定原因…，過陣子再回過頭來解。
-
- 上述範例準備了兩個 timer 函式，devone_interrupt() 是偵測時限使用，devone_irq() 則是模擬 interrupt 使用，在 process 打開裝置檔(/dev/devone)時，會初始化這兩個 timer 函式，但當 process 實際 read 時，才會真正啟動 timer。
- 透過 wait_event() 讓 process 進入 sleep 狀態。
- process 被喚醒後，首先會取消 timer 函式，接著檢查收到的是 interrupt 還是 timeout，來決定回傳的狀態碼。
- 最後是執行 close() 處理函式中釋放資源的動作。
-
-
-
- 9-6、結語
- 應用程式如果需要每隔一段時間作什麼事的話，只要建立 thread 反覆等待執行即可，但驅動程式不能用這種作法。
- 在 kernel space 內，如果不平行執行工作的話，就無法提供 OS 功能了。

kernel timer 使用時雖然有些限制、麻煩，但卻能讓驅動程式更加靈活、提供更多功能。

十、同步與鎖定

前一章的計時器有「錯過」的問題，但被這個問題困擾的不只計時器，本章要繼續介紹「同步與鎖定」。

10-1、同步與鎖定的必要性

驅動程式要能夠強固的執行，則必須考慮到「同步(synchronization)」與「鎖定(exclusion)」的相關知識，另一種說法為「競爭狀態(race condition; racing)」，指的是相同的東西。

驅動程式分成 user context 與 interrupt context ，而這兩種 context 十分可能同時並列執行，因此在開發驅動程式時，必須隨時記得「現在寫的程式碼，必須能毫無問題地同時並列執行」。

10-2、read-modify-write

要理解為何需要鎖定，常見的描述是「ead-modify-write」，這指的是三個連續動作：

- 讀取資料
- 變更資料
- 寫入資料

如果這段動作有兩個 thread 同時執行的話，則難保執行結果會是正確的。

10-3、Atomic 操作

要讓遞增、遞減變動的動作「atomic」時，可以使用 kernel 提供的機制：
將變數定義為 atomic_t 型別，並使用 ATOMIC_INIT 巨集為 atomic 變數設定初始值，如：
atomic_t counter = ATOMIC_INIT(0);
操作 atomic 變數的函式，在 IA-32 版本是定義在「include/asm-x86/atomic_32.h」檔案內。

Atomic 操作的範例：

```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

atomic_t counter = ATOMIC_INIT(0);

static int sample_init(void) {
int n;

atomic_inc(&counter);
n = atomic_read(&counter);
printk("n %d\n", n);

if (atomic_dec_and_test(&counter))
printk("atomic_dec_and_test() TRUE\n");
```

```
else
printk("atomic_dec_and_test() FALSE\n");

return 0;
}

static void sample_exit(void) {
printk("sample driver removed.\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

10-4、旗標

程式處理的過程中，有一些不適合同時並行執行的地方稱為「critical section」，要保護 critical section 的話，必須：

- 一次只讓一個 thread 執行 critical section

也因為這個作法，可能導致系統的效能降低，所以有另一個鐵則：

- critical section 的程式碼極力縮短

linux kernel 提供了「semaphore(旗標)」作為 user context 進行同步與鎖定的機制，用法只要在 critical section 前、後，執行「P操作」、「V操作」即可。
執行「P操作」，就是取得進入 critical section 的許可，並拿到 semaphore，此時若有其它 context 要求「P操作」，就會進入等待狀態。
離開 critical section 時，作做「V操作」，釋放 semaphore，此時如果有其它 context 正在等待，就會喚醒其中一個，讓它獲得 semaphore。
也因為 semaphore 會「sleep」的緣故，所以：

- 不能在 interrupt context 使用 semaphore

驅動程式要使用 semaphore 時，必須先定義 semaphore 結構變數：

```
struct semaphore sem;
```

要初始化 semaphore 變數，要呼叫 init_MUTEX()：

```
void init_MUTEX(struct semaphore *sem);
```

取得 semaphore 的動作(P操作)可透過 down() 或 down_interruptible() 完成：

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
```

釋放 semaphore 的動作(V操作)則是 up()：

```
void up(struct semaphore *sem);
```

取得 semaphore 的函式有兩種，差異在於 user context(process) 進入 sleep 後：

- 能不能被 signal 解除 sleep

關於 down() 與 down_interruptible() 的選用方式，大多數情況下可直接使用 down()，力求讓驅動程式簡化。
而使用 down_interruptible() 的情況包括 socket interface 收封包資料等情況，在收到封包前都無法處理系統呼叫的話很麻煩，所以會希望能透過 signal 中斷等待。

使用 semaphore 的範例：

```
#include
#include
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static int devone_devs = 1; /* device count */
static int devone_major = 0; /* MAJOR: dynamic allocation */
static int devone_minor = 0; /* MINOR: static allocation */
static struct cdev devone_cdev;
static struct class *devone_class = NULL;
static dev_t devone_dev;

struct devone_data {
struct semaphore sem;
};
```

```
#define DATA_MAX 64
static unsigned char data[DATA_MAX];

int devone_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos) {
struct devone_data *dev = filp->private_data;
int len;
int retval;

if (down_interruptible(&dev->sem))
return -ERESTARTSYS;
if (count > DATA_MAX)
len = DATA_MAX;
else
len = count;

if (copy_from_user(data, buf, len)) {
retval = -EFAULT;
goto out;
}
retval = len;
out:
up(&dev->sem);
return (retval);

}

int devone_read(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos) {
struct devone_data *dev = filp->private_data;
int len;
int retval;
```

```
if (down_interruptible(&dev->sem))
return -ERESTARTSYS;
if (count > DATA_MAX)
len = DATA_MAX;
else
len = count;
if (copy_to_user(data, buf, len)) {
retval = -EFAULT;
goto out;
}
retval = len;
out:
up(&dev->sem);
return (retval);
}

int devone_close(struct inode *inode, struct file *filp) {
struct devone_data *dev = filp->private_data;
if (dev)
kfree(dev);
return 0; /* success */
}

int devone_open(struct inode *inode, struct file *filp) {
struct devone_data *dev;

dev = kmalloc(sizeof(struct devone_data), GFP_KERNEL);
if (dev == NULL)
return -ENOMEM;
/* initialize members */
init_MUTEX(&dev->sem);
filp->private_data = dev;
```

```
return 0;
}

struct file_operations devone_fops = {
.owner = THIS_MODULE,
.open = devone_open,
.release = devone_close,
.read = devone_read,
.write = devone_write,
};

static int devone_init(void) {
dev_t dev = MKDEV(devone_major, 0);
int alloc_ret = 0;
int major;
int cdev_err = 0;
struct class_device *class_dev = NULL;

alloc_ret = alloc_chrdev_region(&dev, 0, devone_devs, "devone");
if (alloc_ret)
goto error;
devone_major = major = MAJOR(dev);

cdev_init(&devone_cdev, &devone_fops);
devone_cdev.owner = THIS_MODULE;
devone_cdev.ops = &devone_fops;
cdev_err = cdev_add(&devone_cdev, MKDEV(devone_major, devone_minor), 1);

if (cdev_err)
goto error;

/* register class */
```

```
devone_class = class_create(THIS_MODULE, "devone");
if (IS_ERR(devone_class))
goto error;
devone_dev = MKDEV(devone_major, devone_minor);
class_dev = class_device_create(devone_class, NULL, devone_dev, NULL,
"devone%d", devone_minor);

printk(KERN_ALERT "devone driver(major %d) installed.\n", major);
return 0;

error:
if (cdev_err == 0)
cdev_del(&devone_cdev);
if (alloc_ret == 0)
unregister_chrdev_region(dev, devone_devs);
return -1;
}

static void devone_exit(void) {
dev_t dev = MKDEV(devone_major, 0);
class_device_destroy(devone_class, devone_dev);
class_destroy(devone_class);

cdev_del(&devone_cdev);
unregister_chrdev_region(dev, devone_devs);
printk("KERN_ALERT devone driver removed.\n");
}

module_init(devone_init);
module_exit(devone_exit);
```

對 linux kernel 來說，locked 的 semaphore 有些問題，所以通常使用「等待完成(completion)」機制。

10-5、等待完成

要等待某件處理中的工作完成，可以使用「等待完成(completion)」機制，只要先準備好 completion 變數，再呼叫要等待的函式，之前此函式處理結束後，會自動呼叫通報完成的函式，並返回等待的函式，也由於過程中會 sleep，所以無法在 interrupt context 中使用。相關函式定義在 linux/completion.h 檔案內。

init_completion() 負責初始化「completion變數」：

```
void init_completion(struct completion *x);
```

呼叫 wait_for_completion() 之後，會讓 user context(process) 進入 sleep，在等待對象完成前都不會醒來：

```
void fastcall __wait_for_completion(struct completion *x);
```

其它的變化還包括可以用 signal 喚醒的 wait_for_completion_interruptible() 及可指定等待時限的 wait_for_completion_timeout() 等等：

```
int fastcall __wait_for_completion_interruptible(struct completion *x);
unsigned long fastcall __wait_for_completion_interruptible(struct completion *x,
unsigned long timeout);
```

用來喚醒等待者的是 complete()，就算錯過時機，在 wait_for_completion() 進入 sleep 之前呼叫了 complete() 也不會造成鎖死，因為內部有用 spin lock 保護 sleep 動作：

```
void fastcall complete(struct completion *x);
```

另外也有通報完同，同時終止 kernel thread 的 complete_and exit() 函式：

```
void fastcall complete_and_exit(struct completion *x, long code);
```

Completion 的範例程式，它會在載入驅動程式時，建立 kernel thread，並在卸除驅動程式時，會先結束 thread 的執行：

```
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");
```

```
static wait_queue_head_t wait;
static struct completion comp;
static pid_t thr_pid;

static int sample_thread(void *num) {
printk("%s called\n", __func__);

daemonize("sample_thread");
allow_signal(SIGTERM);

for (;;) {
sleep_on_timeout(&wait, 3 * HZ);
if (signal_pending(current))
break;
}
printk("%s leaved\n", __func__);
complete_and_exit(&comp, 0);
return 0;
}

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed.\n");

init_completion(&comp);
init_waitqueue_head(&wait);

thr_pid = kernel_thread(sample_thread, NULL, CLONE_FS | CLONE_FILES);
if (thr_pid < 0)
return -EINVAL;
printk("kernel_thread %d PID\n", thr_pid);
return 0;
}
```

```
static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed\n");

kill_proc(thr_pid, SIGTERM, 1);
wait_for_completion(&comp);
}

module_init(sample_init);
module_exit(sample_exit);
```

載入此驅動程式後，可下 ps 來看 sample_thread 已被執行：

```
ps aux | grep sample_thread
```

在卸除驅動程式時，會先向 kernel thread 送出 SIGTERM，並以 wait_for_completion() 等待 thread 結束，而 kernel thread 會在脫離無窮迴圈後，會以 complete_and_exit() 通知完成。

10-6、Kernel Thread

上述範例使用 kernel_thread() 函式來建立 kernel thread，但現在的 Linux kernel 推薦的是使用 kthread_create() 函式，需 include linux/kthread.h：

```
struct task_struct *kthread_create(int (*threadfn) (void *data), void *data, const char namefmt[], ...);
```

與 kernel_thread() 不同的地方是 kthread_create() 在呼叫之後，只是建立 thread，並不會開始執行。

namefmt[] 引數是 kernel daemon 的名稱字串(最多16 bytes)，之後可透過 wake_up_process() 來啟動 kernel thread：

```
int fastcall wake_up_process(struct task_struct *p);
```

結束 kernel thread 時，是呼叫 kthread_stop() 函式：

```
int kthread_stop(struct task_struct *k)
```

在呼叫 kthread_stop() 之後，kernel thread 如果再呼叫 kthread_should_stop() 函式，就會得到「true」值，此時即可讓 kernel thread 結束：

```
int kthread_should_stop(void)
```

而 kthread_stop() 內部會自動透過 completion 等待 thread 結束。

kthread_create() 範例：

```
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static struct task_struct *kmain_task = NULL;
static wait_queue_head_t wait;

static int sample_thread(void *num) {
printk("%s called\n", __func__);

for (;;) {
sleep_on_timeout(&wait, 3 * HZ);
if (kthread_should_stop())
break;
}
printk("%s leaved\n", __func__);
return 0;
}

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed.\n");

init_waitqueue_head(&wait);

kmain_task = kthread_create(sample_thread, NULL, "sample_kthread");
if (IS_ERR(kmain_task))
return PTR_ERR(kmain_task);
wake_up_process(kmain_task);
```

```
return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed\n");
kthread_stop(kmain_task);
}

module_init(sample_init);
module_exit(sample_exit);
```

10-7、禁止中斷

如果在 user context 進入 critical section 時，有 interrupt 進來的話，就會切到驅動程式的處理函式，而如果 interrupt 處理函式會進入到相同的 critical section 時，就必須作好鎖定的工作，否則將會造成 deadlock，因為此時 critical section 已被 user context 佔住，所以 interrupt 處理函式無法進去，又因為 interrupt 為最優先，所以 user context 也無法退出 critical section。

解決的方法之一是在 user context 進入 critical section 之前暫時「禁止中斷」，直到脫離 critical section。
但是這是假定只有一顆 CPU 的情形下，禁止、許可中斷的函式如下：

```
void local_irq_save(unsigned long flags);
void local_irq_restore(unsigned long flags);
```

如果系統是多 CPU 的話，則單單禁止中斷是不夠的，因為 critical section 還是有可能被 interrupt context 優先使用。

10-8、自旋鎖

要保護 interrupt context 之內的 critical section 時，不能用 semaphore，而是要使用「spinlock(自旋鎖)」，semaphore 在等待的時候會 sleep，而 spinlock 則是對在原地 busy-loop，也就是會佔住一顆 CPU，直到等到為止，因此 spinlock 在使用時有個鐵則：

- 極力縮短 critical section 的處理時間

使用 spinlock 時，如果有 critical section 同時出現在 interrupt context 與 user context 的話，則兩邊都必須使用 spinlock，若是只有在 user context 出現的 critical section 則應該用 semaphore 來鎖定。

10-8.1、限制

在使用 spinlock 時，擁有鎖定時可以做的事是有限制的：

- 極力縮短 critical section 的處理時間
- 擁有鎖定時不能 sleep

因為在使用 spinlock 時，鎖定只能自己解除，所以如果 sleep 的話，會造成死結。

在獲得鎖定時能不能呼叫某些 kernel 函式，需看 kernel 函式原始碼的註解判斷，如：

- printk() 的註解：This is printk(). It can be called from any context. ...
- copy_to_user() 的註解： User context only. This function may sleep. ...

10-8.2、spinlock 相關函式

使用 spinlock 時，要在全域空間配置 spinlock 變數，變數型別是 spinlock_t：

```
spinlock_t lock;
```

初始化 spinlock 變數時是使用 spin_lock_init() 巨集：

```
spin_lock_init(spinlock_t *lock);
```

要取得 spinlock 多半只要呼叫 spin_lock_irqsave()：

```
spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
```

spin_lock_irqsave() 會將呼叫的程式碼所在的處理器中斷設定存到 flags 變數中，禁止處理中斷並取得鎖定。

解鎖時，則是呼叫 spin_unlock_irqrestore()：

```
spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
```

spin_unlock_irqrestore() 會解除鎖定，並復原本地處理器的中斷設定。

儘管 spin_lock_irqsave() 與 spin_unlock_irqrestore() 只要成對出現即可，但如果需要放在不同函式的話，則可能代表著驅動程式的設計有問題。

10-8.3、死結問題

如果使用 spin_lock() 取代 spin_lock_irqsave() 的話，可能發生什麼樣的問題？

下例是使用 spinlock 的範例，在 user context 的 read 處理函式與 interrupt context 的 timer 處理函式讀寫 sample_data[] 時，會呼叫 spin_lock() 來鎖定：

```
#include
```

```
#include
#include
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static int devone_devs = 1; /* device count */
static int devone_major = 0; /* MAJOR: dynamic allocation */
static int devone_minor = 0; /* MINOR: static allocation */
static struct cdev devone_cdev;
static struct class *devone_class = NULL;
static dev_t devone_dev;

struct devone_data {
spinlock_t lock;
struct timer_list tickfn;
};

#define TIMEOUT_VALUE (5*HZ)
#define DATA_MAX 64
static unsigned char sample_data[DATA_MAX];

static void sample_timeout(unsigned long arg) {
struct devone_data *dev = (struct devone_data *)arg;
int i;
printk("%s entered\n", __func__);

spin_lock(&dev->lock);
for (i = 0; i < DATA_MAX; i++)
sample_data[i]++;
```

```
spin_unlock(&dev->lock);
mod_timer(&dev->tickfn, jiffies + TIMEOUT_VALUE);
}

int devone_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos) {
return 0;

}

int devone_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos) {
struct devone_data *dev = filp->private_data;
int i, len;
int retval;
unsigned char tmpbuf[DATA_MAX];

/* Copying the global data to the temporary buffer while holding a spinlock. */
spin_lock(&dev->lock);
for (i = 0; i < DATA_MAX; i++)
tmpbuf[i] = sample_data[i];
mdelay(3000);
spin_unlock(&dev->lock);

if (count > DATA_MAX)
len = DATA_MAX;
else
len = count;
if (copy_to_user(buf, tmpbuf, len)) {
retval = -EFAULT;
goto out;
}
retval = len;
out:
```

```
return (retval);
}

int devone_close(struct inode *inode, struct file *filp) {
struct devone_data *dev = filp->private_data;
if (dev) {
del_timer_sync(&dev->tickfn);
kfree(dev);
}
return 0; /* success */
}

int devone_open(struct inode *inode, struct file *filp) {
struct devone_data *dev;

dev = kmalloc(sizeof(struct devone_data), GFP_KERNEL);
if (dev == NULL)
return -ENOMEM;
/* initialize members */
spin_lock_init(&dev->lock);

init_timer(&dev->tickfn);
dev->tickfn.function = sample_timeout;
dev->tickfn.data = (unsigned long)dev;
dev->tickfn.expires = jiffies + TIMEOUT_VALUE;
add_timer(&dev->tickfn);

filp->private_data = dev;
return 0;
}

struct file_operations devone_fops = {
```

```
.owner = THIS_MODULE,
.open = devone_open,
.release = devone_close,
.read = devone_read,
.write = devone_write,
};

static int devone_init(void) {
dev_t dev = MKDEV(devone_major, 0);
int alloc_ret = 0;
int major;
int cdev_err = 0;
struct class_device *class_dev = NULL;

alloc_ret = alloc_chrdev_region(&dev, 0, devone_devs, "devone");
if (alloc_ret)
goto error;
devone_major = major = MAJOR(dev);

cdev_init(&devone_cdev, &devone_fops);
devone_cdev.owner = THIS_MODULE;
devone_cdev.ops = &devone_fops;
cdev_err = cdev_add(&devone_cdev, MKDEV(devone_major, devone_minor), 1);

if (cdev_err)
goto error;

/* register class */
devone_class = class_create(THIS_MODULE, "devone");
if (IS_ERR(devone_class))
goto error;
devone_dev = MKDEV(devone_major, devone_minor);
```

```
class_dev = class_device_create(devone_class, NULL, devone_dev, NULL,
"devone%d", devone_minor);

printk(KERN_ALERT "devone driver(major %d) installed.\n", major);
return 0;

error:
if (cdev_err == 0)
cdev_del(&devone_cdev);
if (alloc_ret == 0)
unregister_chrdev_region(dev, devone_devs);
return -1;
}

static void devone_exit(void) {
dev_t dev = MKDEV(devone_major, 0);

/* unregister class */
class_device_destroy(devone_class, devone_dev);
class_destroy(devone_class);

cdev_del(&devone_cdev);
unregister_chrdev_region(dev, devone_devs);
printk("KERN_ALERT devone driver removed.\n");
}

module_init(devone_init);
module_exit(devone_exit);
```

spinlock 的測試程式：

```
#include
#include
```

```
#include
#include
#include
#include
#include

#define DEVFILE "/dev/devone0"

int main(void) {
int fd;
int i;
ssize_t size;
char buf[32];

fd = open(DEVFILE, O_RDWR);
if (fd == -1) {
perror("open");
exit(1);
}

for ( ;; ) {
size = read(fd, buf, sizeof(buf));
for (i = 0; i < size; i++)
printf("%02x ", buf[i] & 0xff);
printf("\n");
sleep(3);
}

close(fd);
return 0;
}
```

實際執行這個程式後，會發現 kernel 沒過多久就當了，原因是驅動程式陷入死結，持續佔用 100% CPU 等待解鎖的後果就是 OS 失去反應，要復原只要重開機。
當掉的原因是 user context 取得 spinlock 時沒有禁止本地處理器處理 interrupt，鎖定期間收到 interrupt 的話，在 timer 處理函式又會嘗試取得鎖定，但解除鎖定需要離開 busy-loop，切換到 user process 繼續執行才行，於是產生了死結。
要修正上述的範例，可以把 spin_lock() 換成 spin_lock_irqsave()，spin_unlock() 換成 spin_unlock_irqrestore()

10-8.4、套疊 spinlock
linux kernel 的 spinlock 允許套疊使用，也就是拿到某個鎖定之後，可以再拿其它的鎖定，減少鎖定的範圍可以提昇程式的效率，但也可能造成死結。
同時並列執行的死結問題，有個很有名的例子是「dining philosophers」套疊取得 spinlock 也可能造成類似的情形。

10-9、其它
linux kernel 內部提供各式各樣的同步機制，舉例如下：

10-9.1、核心搶佔執行(Kernel preemption)
說 linux 是 preemptive 系統，其實只對了一半，因為可以被 preempt 的只有應用程式而已，而 linux kernel 並不是 preemptive，驅機程式也包含在內。
但新版的 linux kernel 支援先進的 kernel preemption 功能(需啟動對應的 kernel 配置選項)，這個想法基於取得 spinlock 時可以釋出執行資源，如果在等待鎖定的過程中不執行 busy-loop，而是讓其它程式使用 CPU 的話，就可以提高效率。

10-9.2、大核心鎖(Big Kernel Lock)
「Big Kernel Lock」是鎖定整個 linux kernel 的機制，可透過 lock_kernel() 與 unlock_kernel() 函式使用，這是先前為了讓 linux kernel 支援多處理器時，暫時引進的鎖定機制，將來會希望把 BKL 徹底清除。

10-9.3、讀寫鎖定(Reader-Writer Lock)
某些 critical section 希望讀取時不必鎖定，而是在寫入時才鎖定，所以準備了「Reader-Writer Lock」機制，可以透過 read_lock() 與 write_lock() 函式取用。

10-9.4、計數讀寫鎖定(Seqlock)
這是一種 spinlock，透過 write_seqlock() 與 read_seqbegin() 等函式提供服務，linux kernel 主要是用此機制來維護系統計時器。

10-9.5、RCU

「RCU (Read Copy Update)」是種較特殊的鎖定方式，透過 rcu_read_lock() 與 rcu_dereference() 等函式提供服務。

10-10、結語

在單 CPU 的年，通常不必煩惱「同步與鎖定」的問題，但目前是多核處理器當道的年代，在設計驅動程式時，一定要把「同步與鎖定」納入考量。

十一、中斷

控制週邊裝置時，與同步、鎖定等功能同樣不可或缺的是「中斷」機制。

11-1、中斷的概念

在 kernel 中，所謂的「中斷」是讓軟體(kernel) 可以接收到來自硬體的緊急報告，並做出反應的機制。
例如，網路卡在收到外面傳來的封包時，就會以中斷向 CPU 報告，CPU 偵側到裝置傳來的中斷後，就會立刻執行 OS 的「中斷處理程序(interrupt handler)」。
「中斷處理程序(interrupt handler)」涵蓋了「中斷程序(interrupt routine)」以及「中斷服務程序(ISR: Interrupt Service Routine)」，各種 OS 用的稱呼都不太一樣，但概念都是相同的。

linux kernel 則常常以「IRQ (Interrupt ReQuest」來代表「中斷」或「中斷處理程序」，IRQ最初是中斷控制晶片的用詞，與 CPU 的中斷號碼(中斷向量、位址)不同，但 linux 也把 CPU 的中斷號碼稱為「IRQ」。

系統內，各個裝置的中斷訊號透過「可程式化中斷控制器」整合在一起，裝置與中斷控制器之間有著稱為「IRQ line」的中斷線路，如果要知道是哪個裝置要求中斷的話，只要看是哪條 IRQ line 送來的訊號即可。

可程式化中斷控制器中的「可程式化」指的是 kernel 可以修改它的設定，用以決定某些 IRQ line 是否接收中斷。
中斷控制器在收到中斷之後會把 IRQ line 轉換成對應的中斷編號(向量)，並向 CPU 的 INTR 腳位送出訊號，通知發生中斷。
「EOI(End Of Interrupt」是 CPU 通知中斷控制器，已處理好中斷情形的訊號，中斷控制器收到這個訊號後，就可以把 IRQ line 放掉，準備接收下一次中斷訊號。
從 /proc/interrupts 可以看到 IRQ 的狀態，最左邊的數值即是 IRQ 編號：

```
cat /proc/interrupts
```

如果裝置不支援中斷，則 kernel 就要用「輪詢(polling)」的機制來監視裝置的狀態，這會耗費 CPU 的處理資源，可且處理可能不夠及時。

11-2、驅動程式的中斷問題

驅動程式如果要收到裝置送出的中斷訊號，必須先登記中斷處理程序，中斷處理程序的登記有時隨裝置而異，大多為下列兩者之一：

- 驅動程式載入後
- 裝置初始化完成後

I2C 驅動程式屬於前者，而網卡、序列埠驅動程式則屬於後者。
網卡、序列埠驅動程式必須先設定好硬體才能開始使用，所以在裝置初始化完成後才登記中斷處理程序，反過來說，如果不先關閉裝置(中斷連線)，就不能卸除中斷處理程序，在卸除驅動程式時需注意，舉例來說，網卡要透過 ifconfig 的 up 以及 down 來啟動、關閉。
而不管是哪種類型，在卸除驅動程式之前，都必須卸除中斷處理程序。

登錄中斷處理程序用的是 request_irq() 這個 kernel 函式：
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long irqflags, const char *devname, void *dev_id);
irq 引數指定的是 IRQ 編號。
handler 引數是中斷處理程序的指標， prototype 如下：
typedef irqreturn_t (*irq_handler_t) (int, void *);
第一個引數是 IRQ 編號，第二個引數則會傳入 dev_id 的內容。
回傳值是 irqreturn_t 型別(實際上是 int 型別)，需要傳回適當的數值：

傳回值	意義
IRQ_NONE	沒有處理這個中斷
IRQ_HANDLED	已經處理這個中斷

irqflags 引數是選用的，不指定時會傳入「0」。
devname 引數是要在 /proc/interrupts 顯示的名稱，以字串字面常數指定。
dev_id 引數是在共享 IRQ 時讓 kernel 區分中斷處理程序用的，傳入驅動程式的私有位址即可，如果沒有共享 IRQ時，則設為 NULL。

驅動程式的中斷處理程序很可能同時並列執行，可以需寫成可以重複進入(reentrant)的形式。
linux kernel 會依序執行同一個 IRQ 的中斷處理程序，不會同時並列呼叫中斷處理程序。
另外如果安裝了許多個相同的裝置，而這些裝置又分配到不同的 IRQ 時，呼叫的中斷處理程序仍然是同一份。
如果是使用 APIC 中斷控制器的多處理器環境下，則中斷處理程序就十分有機會同時並列執行，

中斷處理程序的範例，這個驅動程式會共享網路介面(eth0) 的 IRQ，接下網路裝置的中斷訊號：

```
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

const int IRQ_NUM = 17;
void *irq_dev_id = (void *)&IRQ_NUM;

irqreturn_t sample_isr(int irq, void *dev_instance) {
if (printk_ratelimit()) {
printk("%s: irq %d dev_instance %p\n", __func__, irq, dev_instance);
}
return IRQ_NONE;
}

static int sample_init(void) {
int ret;
printk(KERN_ALERT "sample driver installed.\n");

ret = request_irq(IRQ_NUM, sample_isr, IRQF_SHARED, "sample", irq_dev_id);

if (ret) {
```

```
printk("request_irq() failed (%d)\n", ret);
return (ret);
}

return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed\n");
free_irq(IRQ_NUM, irq_dev_id);
}

module_init(sample_init);
module_exit(sample_exit);
```

載入驅動程式後，會發現 IRQ 17 最後多了「sample」：
```
cat /proc/interrupts
```

中斷處理程序會在 interrupt context 運作，所以可以運用的 kernel 函式有限制，同時也要極力縮短處理時間。
中斷處理程序的內容隨裝置而異，大體工作如下：

- (有需要的話)取得 spinlock
- 檢查中斷狀態
- 解決造成中斷的原因
- 處理中斷工作
- 傳回 IRQ_HANDLED

11-3、Tasklet

中斷處理工作須極力縮短時間，同時在中斷控制器的 IRQ line 送出訊號期間，無法進一步處高其它中斷，而 CPU 更會優先處理中斷，而沒辦法作其它事。

因此 linux kernel 提供了「tasklet」機制，讓中斷處理程序只需要完成最重要的部分即可，稍後再執行剩下的工作。

一般用語常為「延遲中斷」或「軟體中斷」，Windows 稱為「DPC(Deferred Procedue Call)」，HP-UX 則稱為「software trigger」，各平台的稱呼都不盡相同。
使用 tasklet 機制，可以讓中斷處理程序儘快結束，以便中斷控制器接收下一個中斷，提高系統整體效能。
在中斷處理程序呼叫 tasklet 時，tasklet 並不會立刻開始執行，它會等到系統有空閒時才開始執行。

驅動程式想使用 tasklet，必須準備好「tasklet 變數(tasklet_struct 結構)」，並填入資料、做好初始化，初始化工作由 tasklet_init() 完成：

```
void tasklet_init(struct tasklet_struct *t, void (*func) (unsigned long), unsigned long data);
```

func 引數是要以 tasklet 形式執行的驅動程式自有函式。
data 引數則是要傳給這個函式的資料。
驅動程式卸除時，必須以 tasklet_kill() 清掉不需要的 tasklet：

```
void tasklet_kill(struct tasklet_struct *t);
```

啟動 tasklet 的工作由 tasklet_schedule() 負責，由中斷處理程序負責呼叫：

```
void tasklet_schedule(struct tasklet_struct *t);
```

若要暫時取消 tasklet 或重新啟動的話，可以使用：

```
void tasklet_disable(struct tasklet_struct *t);
void tasklet_enable(struct tasklet_struct *t);
```

tasklet 的範例程式：

```
#include
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

const int IRQ_NUM = 17;
void *irq_dev_id = (void *)&IRQ_NUM;
struct tasklet_struct tasklet;

void sample_tasklet(unsigned long data) {
printk("%s called (%ld, %ld, %ld)\n", __func__, in_irq(), in_softirq(), in_interrupt());
}
```

```
irqreturn_t sample_isr(int irq, void *dev_instance) {
if (printk_ratelimit()) {
printk("%s: irq %d (%ld, %ld, %ld)\n", __func__, irq, in_irq(), in_softirq(), in_interrupt());
}
return IRQ_NONE;
}

static int sample_init(void) {
int ret = 0;
printk(KERN_ALERT "sample driver installed.\n");

tasklet_init(&tasklet, sample_tasklet, 0);
ret = request_irq(IRQ_NUM, sample_isr, IRQF_SHARED, "sample", irq_dev_id);

if (ret) {
printk("request_irq() failed (%d)\n", ret);
return (ret);
}

return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed\n");
tasklet_kill(&tasklet);
free_irq(IRQ_NUM, irq_dev_id);
}

module_init(sample_init);
module_exit(sample_exit);
```

11-4、Work queue

tasklet 雖然可以用來延後處理一些中斷情形留下的工作，但 tasklet 還是只能在 interrupt context 中執行。
因此 linux 還提供了「work queue」機制，允許中斷情形的後續處理工作在 user space 執行。

work queu 的原理是延後處理工作時，排進 wait queue 的等待隊伍之中，一個 work queue 有一個對應的 kernel daemon，會在適當時機以 kernel daemon 處理這些延後的工作，而 kernel daemon 是一隻在 user context 執行的程式。

驅動程式要使用 work queu，要先準備「work queu 變數 (work_struct 結構)」並以 INIT_WORK 巨集初始化：

```
void INIT_WORK(struct work_struct *work, work_func_t func);
```

work_func_t 的定義如下：

```
typedef void (*work_func_t) (struct work_struct *work);
```

在 func 引數登記的函式是要延後處理的工作，稍後由 work queue 執行。

中斷處理程序呼叫 schedule_work() 後，會在適當的時機執行指定的工作、呼叫 INIT_WORK 巨集指定的函式：

```
int schedule_work(struct work_struct *work);
```

這邊使用的 work queue 是「keventd_wq」，負責執行的 kernel daemon 是「events」，「keventd_wq」是泛用的 work queue，許多驅動程式都有用到：

```
ps aux | grep events
```

卸除驅動程式時，如果 work queue 之內還有工作要處理的話，須以 flush_scheduled_work() 把它們執行完成：

```
void flush_scheduled_work(void);
```

如果要取消 work queue 之內剩餘的工作，可以改用 cancel_work_sync()：

```
int cancel_work_sync(struct work_struct *work);
```

使用 work queue 的範例程式：

```
#include
#include
#include
```

```
MODULE_LICENSE("Dual BSD/GPL");

const int IRQ_NUM = 17;
void *irq_dev_id = (void *)&IRQ_NUM;
struct work_struct workq;

void sample_workqueue(struct work_struct *work) {
printk("%s called (%ld, %ld, %ld) pid %d\n", __func__, in_irq(), in_softirq(), in_interrupt(),
current->pid);
msleep(3000); /* sleep */
}

irqreturn_t sample_isr(int irq, void *dev_instance) {
if (printk_ratelimit()) {
printk("%s: irq %d (%ld, %ld, %ld)\n", __func__, irq, in_irq(), in_softirq(), in_interrupt());
schedule_work(&workq);
}
return IRQ_NONE;
}

static int sample_init(void) {
int ret = 0;
printk(KERN_ALERT "sample driver installed.\n");

INIT_WORK(&workq, sample_workqueue);
ret = request_irq(IRQ_NUM, sample_isr, IRQF_SHARED, "sample", irq_dev_id);

if (ret) {
printk("request_irq() failed (%d)\n", ret);
return (ret);
}
```

```
return 0;
}

static void sample_exit(void) {
flush_scheduled_work();
free_irq(IRQ_NUM, irq_dev_id);
printk(KERN_ALERT "sample driver removed\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

使用 work queu 時，函式只會拿到 struct work_struct 的指標，從這個指標要讀寫驅動程式的私有空間需要花點功夫：

- 在驅動程式的 struct 之內儲存 work_struct 變數
- 使用 container_of 巨集

11-5、結語

驅動程式控制裝置時，幾乎都必須用到「中斷」的概念，但處理中斷時，有不少限制，在開發驅動程式時須特別留意。

十六、測試與除錯

開發驅動程式的測試與除錯方法，與開發應用程式時有著很大的差異。

16-1、裝置驅動程式的測試與除錯

應用程式遇到 segmentation fault 時，可以用 gdb 解析 core dump，就能還原當機當時的堆疊，而且，應用程式也很容易在磁碟上留下運作記錄，分析起來較為方便。
但驅動程式是跟 kernel 一起運作的，如果驅動程式碰了不對的記憶體位址，則可能跟 kernel 一起當掉，當然最近的 linux kernel 變得更強固，驅動程式用了 NULL 指標並不會讓整個 kernel 當掉，但這個驅動程式還是無法繼續運作。

如果要留下驅動程式的運作記錄，在 kernel 之內也沒有能直接取用的檔案系統，也就是說驅動程式只能在記憶體上運作，不保證能確實把記錄內容存成檔案。
linux kernel 本身因為 kernel panic 當掉時，只會在螢幕上顯示 oops 訊息，無法像商用作業系統可以把整塊記憶體內容存到磁碟。
因此驅動程式的開發者須看著稀少的資料，一邊閱讀原始碼，尋找導致當機的原因。

16-2、建構工程

linux 2.6 是以 make 建構驅動程式，以下列的驅動程式為例：

```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed.\n");
return 0;
}

static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

make 時只需要一般使用者身份，要清除建構出來的檔案時，只需要執行：

```
make clean
```

如果要得知詳細的建構過程，可以在 make 時加上「V=1」引數：

```
make V=1
```

或是設定 KBUILD_VERBOSE 環境變數，也能開「verbose」模式：

```
export KBUILD_VERBOSE=1
```

另外在編譯驅動程式時，gcc 一定會接到「-Os」打開最佳化，這是無法關掉的，因為關掉 gcc 最佳化的話，就不會展開 inline 函式，會導致 kernel 無法正常運作。
所以 kernel 開發者須特別留意：

- gcc 的最佳化臭蟲

16-3、編譯選項

想看驅動程式原始碼時，可在 Makefile 的 CFAGS 指定「-S」，如此就會把「-S」傳給 gcc，使得編譯工作在中途停止，並把產生的組合語言碼存到 -o 指定的檔案：

```
CFLAGS += -S
CFILES := main.c

obj-m += sample.o
sample-objs := $(CFILES:.c=.o)

all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

要把許多原始碼編譯成同一個 kernel module 時，所有的 .c 檔都會以同樣的編譯選項編譯，如果要為某個特定的 .c 檔指定特別的 CFLAGS，可以在 makefile 寫「CFLAGS_$F.o」來支援：

```
CFILES := main.c sub.c
CFILES_sub.o := -DDEBUG

obj-m += sample.o
sample-objs := $(CFILES:.c=.o)

all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

```
clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

如此，就會只有在編譯「sub.c」時，會傳「-DDEBUG」給 gcc。

16-4、前置處理器

gcc 在編譯時，會從指定的目錄內尋找 include 標頭檔載入，預設的路徑可以這個指令列出：

```
cpp -v /dev/null
```

可以看出 #include<...> 的搜尋路徑包含這三個：

```
/usr/local/include
/usr/lib/gcc/i386-redhat-linux/4.1.2/include
/usr/include
```

加上 -nostdinc 選項的話，就會略過這三個目錄，而要明確指定 include 標頭檔搜尋目錄的話，可以透過 -I 選項達成。

gcc 預先定義的巨集可透過這個指令全部顯示出來：

```
cpp -dM /dev/null
```

驅動程式所使用的 kernel 函式，有不少都是巨集，如果要查閱前置處理器怎麼展開 C 原始碼巨集的話，可以為 gcc 加上「-E -P」選項：

```
CFLAGS += -E -P
CFILES := main.c

obj-m += sample.o
sample-objs := $(CFILES:.c=.o)

all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

編譯驅動程式後，就會在 -o 指定的檔案下儲存前置處理器結果。

16-5、printk 除錯

printk() 可以在任何 context 中使用，因此也能用來追縱中斷處理程序。
只是 printk() 的輸出內容只會存到 kernel buffer(128KB)，如果滿了，會不斷蓋掉舊資料。
user spacce 的 klogd daemon 會定期讀取 /proc/kmsg ，並將內容傳給 syslogd，而 syslogd 再把這些訊息當成系統記錄存到 /var/log/messages。
所以 kernel buffer 有可能會溢出，導致 daemon 會來不及把這些訊息存到檔案裡。

以 printk() 除錯時，可以用條件式來切換顯示與否，而要取得 C 語言的函式名的話，可以用「__FUNCTION__」或「__func__」：

```
#ifdef DRIVER_DEBUG
# define DPRINTK(fmt, args...) printk("%s: " fmt, __FUNCTION__, ## args)
#else
# define DPRINTK(fmt, args...)
#endif
```

有時希望硬體故障時，可以留下記錄，但若硬體持續故障的話，就會留下一大堆訊息，要抑制大量的輸出訊息，可以透過 printk_ratelimit()：

```
#include
#include

MODULE_LICENSE("Dual BSD/GPL");

static int sample_init(void) {
printk(KERN_ALERT "sample driver installed.\n");

for (i = 0; i < 300; i++) {
if (printk_ratelimit())
printk("I2C access error\n");
}
return 0;
}
```

```
static void sample_exit(void) {
printk(KERN_ALERT "sample driver removed\n");
}

module_init(sample_init);
module_exit(sample_exit);
```

執行這個範例的話，就只會在 kernel buffer 看到 10 個訊息。

開始抑制之前的顯示次數可透過 /proc/sys/kernel/printk_ratelimit_burst 設定，而解除抑制的時間可透過 /proc/sys/kernel/printk_ratelimit 設定。
這些位於 /proc/sys/kernel/* 的設定選項，可在 Documentation/sysctl/kernel.txt 找到詳細說明。

16-6、Magic SysRq 鍵

驅動程式發生問題，而讓整個 kernel 行為怪異的話，可以使用「Magic SysRq 鍵」，Magic SysRq 是透過特定按鍵方式題示暫存器內容、 process 狀態的機制。
啟動 Magic SysRq 的方法為：

```
# echo 1 > /proc/sys/kernel/sysrq
```

或是修改 /et/sysctl.conf，將 「kernel.sysrq」設為「1」：

```
kernel.sysrq = 1
```

Magic SysRq 的呼叫方法：

主控台種類	按鍵方式
VGA	Alt+SysRq(Print Screen) + 指令
Serial console	Break訊號 + 指令

Serial console 傳送 break 訊號的方式隨終端機軟體而有所不同， TeraTerm 是透過 Control 選單選擇 send break，minicom 則是按「Ctrl+A」鍵再按「F」。
Magic SysRq 可用的輔助指令隨架構而異，說明文件位於 Documentation/sysrq.txt。

16-7、oops 訊息

linux kernel 內部發生某些矛盾時，會在主控台顯示 oops 訊息(panic 訊息)。
linux kernel 在發現自身矛盾時，會時會判斷無法保證後續正常運作，這時就會明確呼叫 panic()：

```
NORET_TYPE void panic(const char * fmt, ...);
```

呼叫 panic() 會讓系統停止運作，如果希望系統能自動重開的話，可以在開機時加上「panic=」選項：

```
# cat /proc/cmdline
ro root=/dev/VolGroup00/LogVol00 rhgb quiet panic=10
```

上述的例子，會在 panic() 之後10 秒，重新啟動。

16-8、分析 kernel panic 情形

linux kernel 在 panic 後，要從當時的指令指標(EIP) 識別對應的指令時，objdump 可以派上用場。
執行 objdump 即可將指令指標附近的指令反組譯，這時需要建構 kernel 時產生的「vmlinux」檔，不能用「vmlinuz」：

```
# objdump -d --start-address=0xc00f9e9c --stop-address=0xc00f9eec ./vmlinux
```

16-9、結語

linux 解析問題的機制還不夠充實，因此在開發驅動程式時，須不斷摸索解決問題的方式。

十七、驅動程式設計與實作實務

解析實際的網路裝置驅動程式如何實作，並解說原始碼的原理。

17-1、網路裝置驅動程式的實作實務

linux 的網路裝置驅動程式不屬於 character 也不屬於 block 類型，是自成一格的特殊類型。
user process 不是直接與網路驅動程式溝通，中間一定要經過 kernel 的 TCP/IP stack。

網路程式不是使用裝置檔，而是使用「socket interface」與驅動程式溝通，對驅動程式送出 IOCTL 時，也是透過 socket interface 執行的， ifconfig 與 ethtool 等指令的操作也是。

TCP/IP stack 與網路驅動程式之間的介面稱為「網路裝置」，驅動程式開發者必須知道網路裝置的用法。
網路裝置驅動程式的原始碼位於 drivers/net 目錄下，接著會以 8139too 驅動程式為例 (drivers/net/8139too.c)，解說網路機置驅動程式的設計與實作。

17-1.1、網路裝置的規格書

要取得網路裝置控制對象的規格書，要先知道晶片的廠商與種類，在 8139too.c 可以看到：
8139too.c: A RealTek RTL-8139 Fast Ethernet driver for Linux.
得知這個驅動程式可控制 RealTek 公司生產的「RTL8139」晶片。
在 www.realtek.com.tw 以「RTL8139」為關鍵字搜尋，即可取得 datasheet。

17-1.2、網路裝置

TCP/IP stack 與驅動程式之間的介面是 Linux kernel 定義的「網路裝置」。
網路裝置是 net_device 結構，定義在 include/linux/netdevie.h 檔案內。

17-1.3、驅動程式的種類

網路驅動程式大多是 kernel module，但也可以靜態連結到 kernel 內部，嵌入式 linux 有時會在開機時透過 NFS 掛載 root filesystem，這時需要在 runlevel 1 就連上網路，因此只能把驅動程式靜態連結到 kernel 內部。

以 kernel module 的形式載入驅動程式的話，要先寫到 /etc/modprobe.conf 檔案：

```
alias eth0 e1000
alias eth1 tulip
```

以 alias 指定 module 名稱。

modprobe.conf 會被 rc script 載入、交給 modprobe 指定載入驅動程式。
而驅動程式檔需要事先複製到 /lib/modules/`uname -r`/kernel/net 目錄內。

17-2、辨識 PCI 裝置

網路裝置是 PCI 裝置的話，需要先將網路裝置驅動程式登記為 kernel 的 PCI 裝置驅動程式，透過 pci_register_driver() 將 PCI ID table、probe 程序、remove 程序登記至 kernel：

```
int pci_register_driver(struct pci_driver *driver);
```

void pci_unregister_driver(struct pci_driver *);

OS 開機時或裝置 hotplug 時，會在偵測到新的 PCI 裝置時搜尋 id_table(rtl8139_pci_tbl)，表格內有符合的 PCI 裝置時，就會呼叫 probe 程序(rtl8139_init_one)。

驅動程式卸載或是移除裝置時，則會依序呼叫清理(rtl8139_cleanup_module)與移除(rtl8139_remove_one)處理程序。

17-2.1、登錄 probe 程序

probe 程序會收到 pci_dev 結構與 pci_device_id 結構，這是 kernel 準備好的資料空間。
probe 程序被呼叫前會先取得 semaphore 進行鎖定，但將來為了縮短 OS 開機時間，還是有可能同時呼叫多個 probe 程序，所以在設計驅動程式時先寫成可重進入的形式會比較好。

probe 成功時會傳回 0，否則傳回非 0 值，如果裝置故障，則驅動程式不僅要回報錯誤，還要呼叫 pci_disable_device() 禁止進一步使用此裝置：
void pci_disable_device(struct pci_dev *dev);

probe 程序在呼叫 rtl8139_init_board() 將 PCI 裝置初始化後，接著從裝置裡讀出 MAC address，以讓 kernel 進行 ARP 處理。
並將此 MAC address 登錄到 dev->perm_addr[]，主要是 ethtool 等指令會用到。

就像 character 類型的驅動程式登記各種處理函式，網路驅動程式也要透過 net_device 結構登記處理程序。

17-2.2、登記 remove 處理程序

呼叫 remove 處理程序的時機為：

- 卸載驅動程式時(rmmod)
- 透過 hotplug 移除裝置時

卸除驅動程式時，必須讓裝置完全停止運作，如果時機抓錯，很容易導致 kernel panic 停止工作。
以 rmmod 來卸除網路驅動程式時，如果網路裝置仍然 up，就會導致錯誤，所以在卸除前，一定要先將裝置 down 下來：
ifconfig eth0 down
把網路裝置 down 下來時，會呼叫驅動程式的 stop 程序。

所以在呼叫 remove 程序時，網路裝置會在停止狀態，remove 程序該做的是切離網路裝置並關閉 PCI 裝置。

8139too驅動程式因為有用到 work queue，所以在網路裝置 down 之後到卸除驅動程式之前，可能還會有 work queue 運作，因此要呼叫 flush_scheduled_work() 等待 work queue 執行完畢。

17-3、IOCTL

網路驅動程式並不具有裝置檔，所以 linux 是透過「socket」來呼叫 IOCTL ，也因為是經由 TCP/IP stack，所以 IOCTL 指令名稱無法讓驅動程式開發者自行定義。

網路驅動程式內可以自行定義的 IOCTL 指令定義在 linux/sockios.h 檔案內，最多為16個。

網路驅動程式 IOCTL 方法定義為 do_ioctl：

```
int (*do_ioctl) (struct net_device *dev, struct ifreq *ifr, int cmd);
```

ioctl 範例：

```
static int sample_ioctl(struct net_device *dev, struct ifreq *ifr, int cmd) {
int rc = 0; /* success return */
switch (cmd) {
case SIOCDEVPRIVATE:
break;
case SIOCDEVPRIVATE+1:
break;
default:
rc = -EOPNOTSUPP;
}
return (rc);
}
```

user process 想送出 IOCTL 時，必須打開 RAW socket 透過 raw socket 送出 IOCTL，並以 ifreq 結構的 name 指定要控制的網路介面。

從 user process 送出 IOCTL 的範例：

```
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include

int main(void) {
int sock;
struct ifreq f;
int ret;

sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
if (sock == -1) {
perror("Can't create socket\n");
exit(1);
}

strcpy(f.ifr_name, "eth0");
ret = ioctl(sock, SIOCDEVPRIVATE, &f);
if (ret == -1)
perror("ioctl error\n");

close(sock);
return 0;
```

```
}
```

17-4、Media Independent Interface

網路驅動程式實作 IOCTL 方法時，相較於上一節自行實作的方式，透過「MII(Media Independent Interface)」提供介面的例子也不少。

MII 是 IEEE 802.ea 定義，介於 MAC(Media Access Control) 與 PHY(Physical) 兩層之間的介面，透過 MII 即可讀寫網路裝置的 PHY 暫存器，讀取 PHY 暫存器即可得知自動協調狀態以及物理連線資訊。

如果驅動程式有支援 MII 的話，即可用 ethtool 指令取得、設定網路裝置的資訊：

```
# ethtool eth0
```

在 8139too 驅動程式的實作 IOCTL 方法程式碼中，只呼叫了 kernel 的 generic_mii_ioctl() 而已：

```
int generic_mii_ioctl(struct mii_if_info *mii_if, struct mii_ioctl_data *mii_data, int cmd,
unsigned int *duplex_chg_out);
```

實際讀寫 MII 暫存器的是驅動程式，因此 MII 介面要在驅動程式初始化時設定好，而此介面是透過 MDC/MDIO 腳位控制，這是類似 I2C 的兩線式簡單介面。
MDC 是 Management Data Clock。
MDIO 是 Management Data Input/Output。

MII 資訊定義為 mii_if_info 結構，其中設有讀寫 PHY 暫存器用的函式：

```
int (*mdio_read) (struct net_device *dev, int phy_id, int location);
```

```
void (*mdio_write) (struct net_device *dev, int phy_id, int location, int val);
```

這兩個函式由驅動程式負責，因為 MII 的控制方式隨裝置而異。
phy_id引數是「PHYID」或「PHYAD」所指的「Physical Address」，這個值是裝置固定的值。
location 引數是「PHYREG」所指的「Physical Register」。

17-5、open/stop 處理程序

17-5.1、start 處理程序

用 ifconfig 將網路介面 up 時，會呼叫驅動程式的 open 處理程序，這時驅動程式該做的事情包括：

- 登記中斷程序
- 配置 DMA 暫存區
- 將裝置初始化
- 指示啟動封包傳送佇列

中斷程序可透過 request_irq() 登錄，IRQ 編號 kernel 會事先設定好，直接使用這個值即可 (dev->irq)。
裝置如果支援 DMA ，就必須配置 DMA 暫存區，可透過 pci_alloc_consistent() 完成。
裝置做好準備後，須告知上層可以開始傳送封包，這時呼叫的是 netif_start_queue()：
void netif_start_queue(struct net_device *dev);

17-5.2、stop 處理程序

用 ifconfig 將網路介面 down 時，會呼叫驅動程式的 stop 處理程序，這個在 character 類型的驅動程式是稱為 release 程序。
這時驅動程式該做的事情包括：

- 指示停止封包傳送佇列
- 停止裝置
- 卸除中斷處理程序
- 釋放 DMA 暫存區

處理順序與 open 處理程序相反。

停止封包傳送佇列的工作可透過 netif_stop_queue() 完成：
void netif_stop_queue(struct net_device *dev);
卸除中斷處理程序可由 free_irq() 完成，但此時中斷處理程序可能還在執行，所以要先以 synchronize_irq() 等待。
釋放 DMA 暫存區可透過 pci_free_consistent() 完成。

17-6、傳送與接收處理程序

17-6.1、傳送處理程序

網路介面 up 之後，應用程式即可透過 socket 送出封包，這時會呼叫驅動程式的傳送程序 (dev->hard_start_xmit)。
其中 xmit 是 transmit 的縮寫。

在 hard_start_xmit() 中，封包會以 socket buffer (struct sk_buff *skb) 的形式傳入，封包大小可透過「skb->len」取得，這是包含 Ethernet frame header 的大小，此時已經解讀 ARP、填好接收端的 MAC address 了。

socket buffer 寫入 DMA 暫存區的動作由 skb_copy_and_csum_dev() 完成，並透過 dev_kfree_sdk() 釋放。

寫入暫存器啟動 DMA 後，將 dev->trans_start 設為目前時間(jiffies)，這是用來檢測傳送時限的值。
如果經過一段時間後仍未完成傳送，就是傳送逾時，此時會呼叫 rtl8139_tx_timeout() ，透過 work queue 處理逾時情形。

封包傳送成功後，則結束中斷，呼叫中斷處理程序 rtl8139_interrupt() ，並轉交 rtl8139_tx_interrupt() 處理傳送結束的情形，此時如果有暫停傳送佇列的話，會呼叫 netif_wake_queue() 將之啟動。

17-6.2、接收處理程序

網路裝置從外部收到封包後會引發中斷，呼叫中斷處理程序 rtl8139_interrupt()。
實際的接收工作是以 polling 的方式進行，以減輕 kernel 的負擔，所以只需要呼叫 __netif_rx_schedule() 。

實際的接收工作由 rtl8139_poll() 來執行，並檢查中斷原因，如果是要接收封包的話，就呼叫 rtl8139_rx()。

17-7、結語

網路裝置需由網路裝置驅動程式來控制，linux 的網路裝置驅動程式既不屬於 characher 類型，也不屬於 block 類型，則讀現有的驅動程式原始碼，即可理解其處理機制。