Linux 網路卡驅動程式 ： 追蹤與效能分析

投稿領域：網路軟體

陳一瑋　林盈達

國立交通大學資訊科學系

新竹市大學路 1001 號

TEL：(03) 5712121 EXT. 56667

E-MAIL：iwchen@cis.nctu.edu.tw , ydlin@cis.nctu.edu.tw

主要聯絡人：陳一瑋 TEL：0927308032

摘要

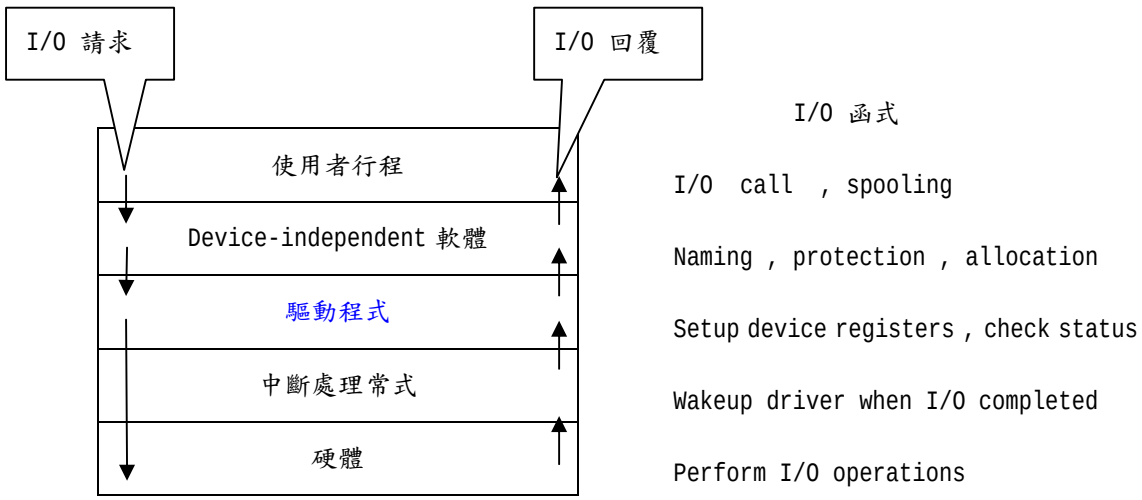
一部電腦的網路元件可分成硬體和軟體兩部分，硬體方面有網路卡、網路線等，軟體方面有網路卡驅動程式(Adapter driver)、協定驅動程式(protocol driver)及應用軟體，其中網路卡驅動程式負責銜接硬體與軟體，本文解說裝置驅動程式運作的流程，找出撰寫裝置驅動程式的重點與要領，以 Linux 網路卡驅動程式為例說明，並對它做一些測試與分析。分析之後發現到在不考慮網路傳輸時間(propagation time)的條件下，封包處理上較花時間的順序為: (1) 網路卡發送(transmit)、接受(receive)封包 ,(2) DMA(直接存取記憶體)封包，(3) 網路卡驅動程式(CPU 執行時間)，三者所花的時間比例大約為 250 ： 30 ： 1，這是以封包大小 1028byte 來測試的結果，封包愈大三者的比例會愈懸殊，可見發送（或接收）及 DMA 封包是時間瓶頸所在；除此之外，也了解在中斷處理時“新封包來到”的情形比“封包傳送完畢”要花多一點時間。

關鍵字：Linux、協定驅動程式、網路卡驅動程式、ISR、IRQ、I/O port、DMA。

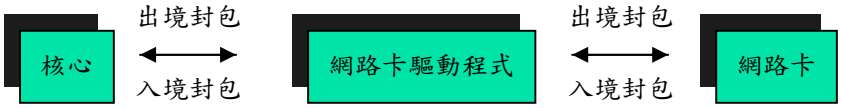
1. 簡介

作業系統的主要功能之一就是控制電腦周邊裝置的輸入、輸出。而用來控制的軟體主要可以分為四個部分，請參考圖一。我們可以看出驅動程式(Driver)在現今的電腦實作架構中處於一個相當重要的地位，它要能 “吃軟” 也要能 “吃硬”。通常中斷處理常式可以視為是驅動程式的一部分。

相對於網路卡驅動程式(Adapter driver)而言，它所要吃的 “軟” 就是上層的協定驅動程式(Protocol driver)，也就 TCP/IP protocol stack；而它所要吃的 ” 硬”就是下層的網路卡，說得更詳細點，便是要和網路卡上的 MAC Controller 溝通，透過 MAC Controller 上的 registers 來交換訊息，以達到網路卡驅動程式的主要工作之一：將封包傳送給網路卡或者是從網路卡上抓取已經到達的封包；而另一項主要的工作，是將從網路卡上抓取到的封包傳遞給上層的協定驅動程式，或者是將從協定驅動程式收到的封包傳遞給網路卡，請參考圖二。



圖一：I/O 系統中的分層以及各層的主要函式



圖二：網路卡驅動程式的主要工作

2. 撰寫驅動程式

如何正確地在 Linux 作業系統中撰寫一個驅動程式呢？以下我們將分為*探測硬體* (probe hardware)、*中斷處理* (interrupt handling)、*使用 I/O 埠* (Using I/O ports) 讀取及寫入資料這三大部分來一一解說。

2.1 探測硬體(probe hardware)

在一個驅動程式可以和裝置做溝通前，必須要做的一些初始化工作包括：探測裝置的 I/O ports 和 IRQ number。有了 I/O ports 才可以和裝置上的某些暫存器(register)溝通，有了 IRQ number 才能正確地將中斷處理常式註冊到核心之中。探測硬體的方法和匯流排(bus)的種類有關，PCI 介面的裝置是在開機時(boot)就自動會將所用的 I/O ports 及 IRQ number 存在其某些暫存器上，而驅動程式只需去讀取這些暫存器就可以了，並不需要真正的去探測裝置，但 ISA 介面的裝置驅動程式就必須要實際地去做探測的工作。

在 PCI 介面方面，Linux 核心 2.4 版更進一步地將 I/O ports 及 IRQ number 整合在系統資源中 (***pci_dev*** structure)，因此在驅動程式中就不必直接由裝置上的暫存器來獲得 I/O ports 和 IRQ number，而只需呼叫下列函式即可由系統資源取得:

```
int pci_enable_device (struct pci_dev *dev);
unsigned long pci_resource_start(struct pci_dev *dev, int bar);
struct resource *request_region (unsigned long start , unsigned long len , char*
name);
void release_region (unsigned long start , unsigned long len);
```

想要取得 I/O ports 和 IRQ number 以前，一定要呼叫 *pci_enable_device( )*來啟動這個 PCI 裝置，接下來若是要 IRQ number，就可從 *dev->irq* 中獲得，若是要 I/O ports 就呼叫 *pci_resource_start( )* 來取得基底位址(base address)，然後呼叫 *request_region( )*向系統宣告要用某一段的 I/O ports 即可使用，最後若沒有要用時要記得呼叫 *release_region( )*來釋放這段 I/O ports。

至於 ISA 介面的裝置就麻煩許多，在探測 I/O ports 方面所用的機制(mechanism)是去“掃瞄”(scan)所有可能的 I/O ports，只有與裝置連接的 I/O ports 才會有正確的回應，

以下是大概的流程:

1. `int check_region (unsigned long start , unsigned long len );`

這個函式是用來檢查這一段 I/O ports 是否可以拿來使用。

2. 實際去探測硬體，察看此裝置是否存在，在此必須儘量必免"寫入"的動作，因為不小心的寫入會讓系統出現很多問題。

3. 呼叫 *request_region( )* 來向核心要求所要用的 I/O ports。

4. 當沒有要用到某段 I/O ports 時，要呼叫 *release_region( )*來釋放出系統資源。

在探測 IRQ number 方面所用到的機制是讓裝置產生一個中斷(interrupt)，然後去查看一些相關資訊以得知裝置所用的 IRQ 是多少，這裡要注意的是：驅動程式是要知道裝置所用的 IRQ 是多少，而不是要分配 IRQ 給裝置，以下是可以用到的函式:

unsigned long probe_irq_on(void);

int probe_irq_off (unsigned long);

*probe_irq_on( )*會回傳一個位元遮罩(bitmask)，此遮罩可以用來判斷還有哪些中斷還沒被用到，而驅動程式必須儲存此遮罩，因為等一下必須將此遮罩當成參數傳給 *probe_irq_off( )*。在 *probe_irq_on( )*執行完後，驅動程式應該要促使裝置產生一個中斷，之後再呼叫 probe_irq_off(bitmask)，如此一來 probe_irq_off( ) 就會傳回此裝置所用的 IRQ number 了。

2.2 中斷處理(interrupt handling)

當資料在核心與裝置間傳輸時難免會有一些的延遲(delay)，因此驅動程式就必須將資料先暫存起來，放到緩衝區中等待資料的完整與正確傳送時刻的到

來，而我們常見的一個較好的緩衝機制為"interrupt-driver I/O"，此方法中輸入緩衝區(input buffer)在中斷處理時被填入資料，再被需要這些資料的程序(process)所取出; 輸出緩衝區(output buffer)被某個程序填入資料後，在中斷處

理時再將資料送給裝置，而所謂中斷處理的部分就是交給中斷處理常式來負責。通常的情況下，一個驅動程式只會為其裝置向核心註冊一個中斷處理常式，以下的函式可以用來註冊及註銷中斷處理常式:

```
int request_irq(unsigned int irq, void(*handler) (int, void *, struct pt_regs *) ,
unsigned long flags, const char *dev_name ,void *dev_id);
void free_irq (unsigned int irq, void *dev_id);
```

request_irq 可以用來註冊中斷處理常式，而 free_irq 則用來註銷它。

2.2.1 中斷處理大綱

當有中斷產生時系統其實是 “軟硬兼施” 的，請參考表一。

1. 硬體儲存程式計數器(programm conuter)等
2. 硬體載入新的程式計數器
3. 組語程式儲存暫儲器值
4. 組語程式設定新的堆疊
5. C 程式處理實際中斷事務、喚醒行程、可能呼叫 schedule( )，最後跳回到組語程式
6. 組語程式啟動目前該執行的行程

表一：中斷處理大綱

Interrupt service routine(ISR)的整個過程為 3 ~ 6，而驅動程式就是實作於第 5 項.

2.2.2 "快速"與"慢速"處理常式

在舊版的 Linux 核心中花了很大的工夫來區別”快速”和”慢速”中斷，所謂 “快” 、” 慢” 的中斷其差別就在於所需的處理時間長短，若一中斷只需很短的時間就可處理完畢即為快速中斷，相反地，若一中斷需要很長的時間來完成就是慢速中斷。相對於中斷處理常式而言，在處理中斷時，處理器是否還會接受其它的中斷回報(interrupt reporting)，會的話則此中斷處置常

式就為 Slow (因為處理器必須兼顧其它中斷)，反之則為 Fast (因處理器完全被此中斷處理常式占住)。兩者有一個相同的地方，無論是 Fast 或是 Slow 的中斷處置式，當其在為某個中斷服務期間，此中斷在中斷控制器 (interrupt controller)中是失效(disable)的。在 *request_irq( )*中的 flags 參數若被設成 *SA_INTERRUPT* 便是代表要註冊一快速中斷處理常式，而在現代的核心中，快速和慢速中斷已經被視為幾乎是同樣的了，表二為兩者的比較:

	在中斷期間使其它中斷失效	在中斷期間使目前服務的中斷失效	服務結束後是否會呼叫 ret_from_sys_call
快速處置常式	是	是	否
慢速處置常式	否	是	是

表二：快速與慢速中斷處置常式的比較

2.2.3 撰寫中斷處理常式

在製作中斷處置常式時所必須要考慮到的工作如表四所列。

判斷中斷的原因，e.g. 網路卡中斷處理常式要判斷中斷是因封包的到來、離去或是錯誤的發生。
叫醒在
若需要較長的執行時間，則使用後續常式(bottom half)來完成。

表四：中斷處置常式所要做的工作

表四中最後一項所寫到的後續常式我們將在下一小節中作介紹。

中斷處理常式由於是在中斷期間所執行，因此其執行時受到了許多的限制，

例如: 它不能和使用者空間互相傳遞資料、它不能做任何會讓自己進人睡眠狀態的事情。在實作中斷處理常式過程中我們可以使用的資源即為所傳入的三個參數，這三個參數分別為 *int irq*、*void* dev_id*、*struct pt_regs* regs*。irq 可以用在記錄檔中，dev_id 是此裝置的指標，當我們使用共享中斷時(e.g. 兩個中斷處理常式共用一個 IRQ number)，中斷處理常式可以利用 dev_id 辨別某個中斷是否屬於自己要處理的，而最後一個參數 regs 很少被使用到，regs 會儲存著處理器在進入中斷處理常式前一時刻的內容，因此 regs 便可做為監控、除錯之用。

2.2.4 後續處理常式(bottom half)

處理中斷時所面對的主要問題之一，就是如何讓處置常式順利執行較為耗時的工作而又不會遺漏掉接下來的中斷，Linux 對此所提出的解決之道，就是將中斷處置常式分為兩個部分，一為 “先行常式”(top half)，另一個部分是 “後續常式”(bottom half)。先行常式就是之前我們提到過，用 request_irq 所註冊的處置常式，其負責將後續常式排班到作業系統中，作業系統等到可以執行的安全時間(所謂安全是指對執行時間的要求不會嚴苛)便會去執行後續常式。Linux 核心中有兩種機制來實作後續常式: BH(也被稱作 bottom half)和 tasklets。BH 為較舊的方法，在核心 2.4 後實際上也是用 tasklets 來實作，而 tasklets 是從 2.3 系列才發展出來的，其為目前較常被使用的後續常式實作方法，但由於 tasklets 無法在較舊的核心上使用，因此若考慮 portability 的問題話，用 BH 的方法較為妥當。先來看看若要以 tasklets 的方法實作後續常式，有哪些函式可以使用:

```
DECLARE_TASKLET(name, function, data);
tasklet_schedule(struct tasklet_struct *t);
```

要如何使用這些函式呢? 舉例而言，若你寫了一個函式 *func( )*要用來作為後續常式，第一步是要向系統宣告一個 tasklet : DECLARE_TASKLET(task,

func, 0)，task 是代表此 tasklet 的名字，接下來再呼叫 tasklet_schedule(&task) 來將此 task 排班到作業系統中，等到作業系統方便時就會去執行它。而在 BH 的方法中，若是想要將一個後續常式排班到系統中，可用下列函式:

void mark_bh(int nr);

其中 nr 是代表某個 bottom-half routine，在舊版的核心中 mark_bh( )會在一個 bitmask 中設定某位元，讓這位元所對應到的中斷處理常式可以順利地被執行通常核心都會提供幾個後續常式讓程式設計者使用，表三列出驅動程式能使用的部分:

IMMEDIATE_BH	TQUEUE_BH	TIMER_BH

表三：核心宣告的後續常式

然而在新版的核心(2.4)中，*mark_bh( )*實際上是去呼叫 *tasklet_hi_schedule( )* (類似 *tasklet_schedule( )*)來將所要執行的 bottom-half routine 排班到核心之中。

2.2.5 競爭狀態 (race condition)

由於 interrupt-driver I/O 的原因，因而引進了某一類的問題: 對於一群行程間若有共享的資料，該如何解決好幾個行程同時要存取該共享資料的同步問題，而這類問題我們稱之為競爭狀態 (race condition)。由於在在驅動程式的任何一個執行點都有可能會發生中斷，因此只要是和中斷有互動的驅動程式(大部分都會有)都必須要注意競爭狀態。在 Linux 核心裡支援好幾種技術來必避資料的敗壞(data corruption)，而我們將介紹最常使用的方式: 使用 spinlocks 來達到行程之間兩兩互斥(mutual exclusion)，如此即可避免競爭狀態的發生，說明白點，就是不論中斷處理常式或驅動程式要存

取共享資料時，要先能拿到(lock)一把鎖，然後才進行共享資料的存取，最後不用時再把這把鎖歸還(unlock)，讓別的行程能夠也藉由同樣的方法來存取共享資料。

Spinlocks 在 Linux 中的型別為 spinlock_t，而以下是一些可以運作在 spinlocks 上的函式:

void spin_lock(spinlock_t *lock);

void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);

void spin_lock_irq(spinlock_t *lock);

void spin_lock_bh(spinlock_t *lock);

void spin_unlock(spinlock_t *lock);

void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);

void spin_unlock_irq(spinlock_t *lock);

void spin_unlock_bh(spinlock_t *lock);

*spin_lock( )*以 busy-wait 的方式取得所想要的一把鎖 (lock)，等到 *spin_lock( )*函式返回時，呼叫它的行程便可以獲得此把鎖。 *spin_lock_irqsave( )*也是以同樣的方法取得一把鎖，除此之外它會使中斷無效，並且把目前的中斷情況記錄到 *flags* 參數中，而 *spin_lock_irq( )*除了不會將目前中斷的情形記錄下來以外，其餘的動作都 *spin_lock_irqsave( )*一樣，至於 *spin_lock_bh( )*，它和前三者都是一樣會用 busy-wait 的方式來取得一把鎖，然而它還不准後續常式被執行。以上四個函式可以獲得一把鎖，相反地，接下來的四個相對應的函式會將鎖歸還，spin_unlock( )就只是單純地把之前所獲得的鎖歸還，spin_unlock_irqrestore( )會把鎖歸還並且根據 flags 的值來使得一些中斷變有效，spin_unlock_irq( )除了把鎖歸還，還會把所有的中斷變得有效，而 spin_unlock_bh( )會將鎖歸還，且讓後續常式可以被執行。在使用這些函式時有兩點要特別注意，第一是要確

定取得(lock)鎖的函式比歸還(unlock)鎖的函式早出現，第二是要這些函式是兩兩成對來使用的。

2.3 使用 I/O 埠讀取及寫入資料

在探測硬體階段之後，驅動程式便可獲得硬體所使用的 I/O ports。大部分的硬體會將 I/O ports 的寬度分成 8 位元、16 位元、以及 32 位元，因此驅動程

式就必須針對不同寬度的 I/O ports 來呼叫不同的函式存取 I/O ports，在 Linux 核心中定義以下的函式可以用來存取 I/O ports。

unsigned inb (unsigned port);

void outb (unsigned char bye, unsigned port);

*inb( )*讀取寬度 8 位元的 ports，而 *outb( )*寫入寬度 8 位元的 ports。

unsigned inw (unsigned port);

void outw (unsigned char bye, unsigned port);

*inw( )*讀取寬度 16 位元的 ports，而 *outw( )*寫入寬度 16 位元的 ports。

unsigned inl (unsigned port);

void outl (unsigned char bye, unsigned port);

*inl( )*讀取寬度 32 位元的 ports，而 *outl( )*寫入寬度 32 位元的 ports。

除了以上單一讀取的方式外，Linux 還支援了字串的操作:

void insb (unsigned port, void *addr, unsigned long count);

void outsb (unsigned port, void *addr, unsigned long count);

*insb( )*從 8 位元的 ports 讀取 *count* 位元組的資料, 然後將這些資料儲存到記憶體位址為 *addr* 的地方，而 *outsb( )*將記憶體位址為 *addr* 的資料，寫入 *count* 位元組到 8 位元的 ports。

void insw (unsigned port, void *addr, unsigned long count);

void outsw (unsigned port, void *addr, unsigned long count);

這兩個函作的運作和以上兩個很類似，除了說它們是針對寬度為 16 位元的 ports 所設計的。

void insl (unsigned port, void *addr, unsigned long count);

void outsl (unsigned port, void *addr, unsigned long count);

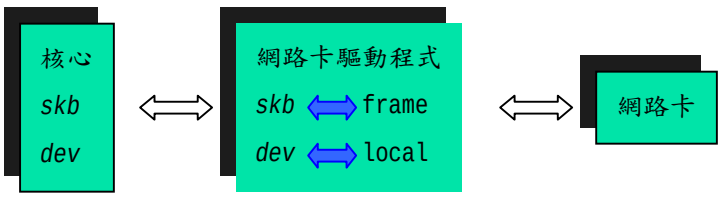
這兩個函作的運作和以上兩個很類似，除了說它們是針對寬度為 32 位元的 ports 所設計的。

3. Linux 網路卡驅動程式實例

在眾家網路卡驅動程式中，我們挑選其中相容性最高的 NE2000、PCI 介面網路卡的 Linux 版驅動程式來進行追蹤，如此更有實際應用的意義。

3.1 重要資料結構(data structure)

在 Linux 系統中，網路卡驅動程式常用到的資料結構有兩種：sk_buff 以及 net_device。圖四顯示出這兩種資料結構是位於系統的哪些位置，也就是有誰會去存取它們。



圖四：skb_buff 及 net_device 於系統中的位置

簡單說明一下圖四：圖中 skb 是指 sk_buff 形態的指標變數，dev 是指 net_device 形態的指標變數。網路卡驅動程式從網路卡上抓取到封包（此時稱作 frame）後，會去配置一塊 sk_buff 記憶體，將 frame 塞進 skb 中的 data 欄位，以後封包在系統中就是以 sk_buff 的形式表現。而 net_device 中的許多欄位值都是在網路卡驅動程式裡填好的，圖中 local 是指 dev 中的大部分的欄位值都是在網路卡驅動程式裡產生然後填入 net_device 欄位的，以後網路卡在系統中就是以 net_device 的形式表現，接著就來詳細介紹這兩個資料結構的主要欄位。

3.1.1 sk_buff

此資料結構定義在<linux/skbuff.h>中，表五列出其主要欄位，與欄位所代表的意義。

欄位名	意義	欄位名	意義
head	指向 sk_buff 的起點	len	資料本身的長度
data	指向“真正資料”的起點	pkt_type	幫包的類型
tail	指向“真正資料”的終點	saddr	來源位址
end	指向 sk_buff 的終點	daddr	目的位址
dev	封包到達或離開的裝置	raddr	路由器位址

表五：sk_buff 的重要欄位

3.1.2 net_device

此資料結構定義在<linux/netdevice.h>中，表六列出其主要欄位，與欄位所代表的意義。

欄位名	意義
Name	裝置名稱
base_addr	I/O 埠的基底位址
Irq	中斷編號
dev_addr	網路卡硬體位址
Mtu	最大傳輸單位
hard_start_xmit	裝置功能函式之一

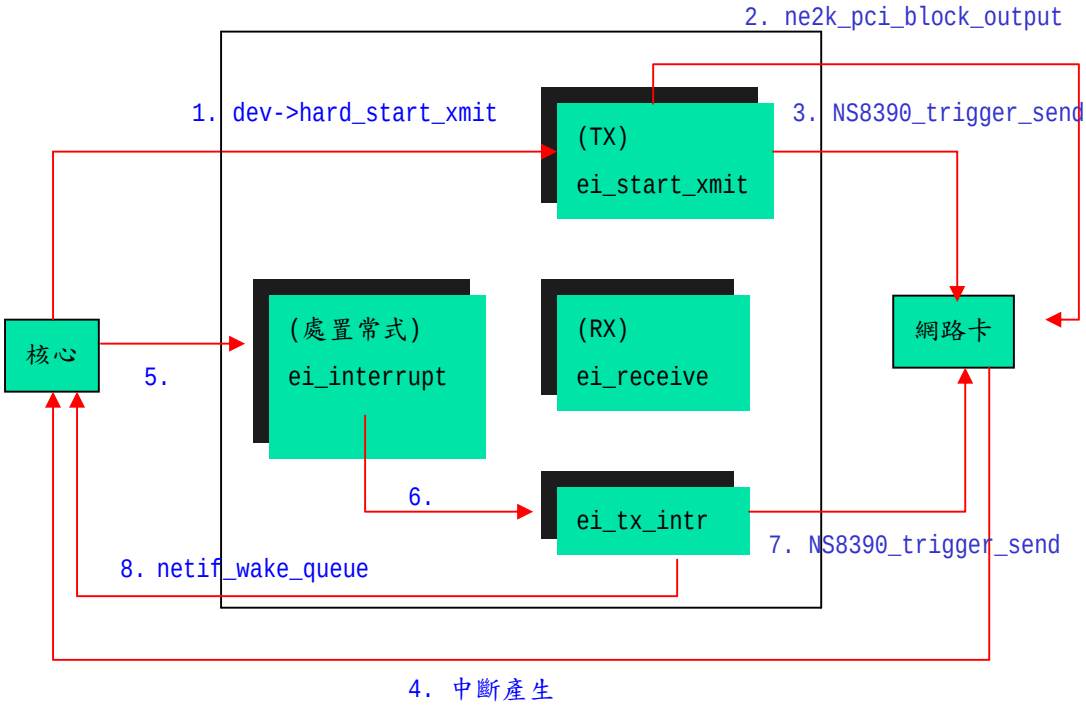
表六：net_device 的重要欄位

3.2 初始化

初始化的工作可分為“註冊中斷處理常式”以及“探測硬體”，註冊中斷處理常就是用之前說過的 *request_irq( )*函式來完成，而探測硬體方面由於是 PCI 介面的網路卡，因此只需用我們之前所教的方法就可取得 I/O ports 及 IRQ number，而不用實際地去探測網路卡。

3.3 封包的發送

圖五顯示出此網路卡驅動程式的封包發送流程，有陰影的部分為驅動程式。

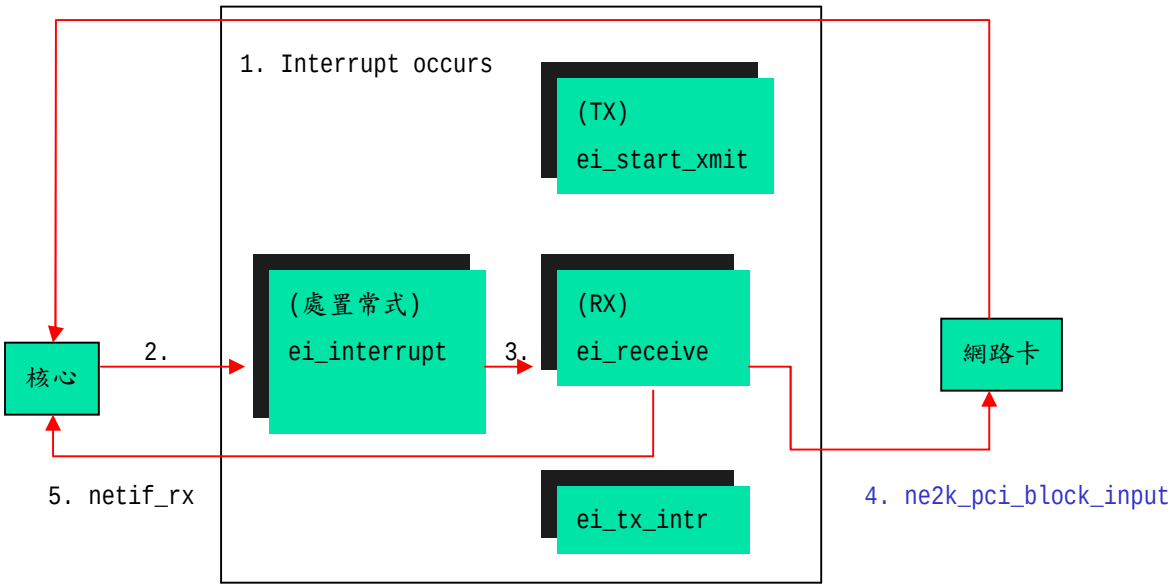


圖五：封包發送流程

在此以文字簡單解說圖五：核心要傳送封包而呼 *dev->hard_start_xmit( )*，在此驅動程式中實作出的函式為 *ei_start_xmit( )*，在 ei_start_xmit( )中會呼 *ne2k_pci_block_output( )*將封包搬到網路卡上，接著呼叫 *NS8390_trigger_send( )*以觸發(trigger)網路卡將封包送出，等到封包送完，網路卡會以中斷來通知核心，於是核心就去呼叫對應的中斷處理常式，在此驅動程式中是 *ei_interrupt( )*，ei_interrupt( )會先判斷是屬於哪一種中斷，發現是封包傳送完所發出的中斷後，它會呼叫 *ei_tx_intr( )*來處理，ei_tx_intr( )會先呼叫 NS8390_trigger_send( )來觸發網路卡上的另一個封包傳送（若有的話），接著呼叫 *netif_wake_queue( )*讓核心知道封包已傳遞完畢可以進行下一項工作。

3.4 封包的接收

圖六顯示出此網路卡驅動程式的封包接收流程。



圖六：封包接受流程

在此以文字簡單解說圖六：當網路卡接收到封包時它會以中斷來通知核心，於是核心就去呼叫對應的中斷處理常式，在此驅動程式中是 *ei_interrupt( )*，ei_interrupt( )會先判斷是屬於哪一種中斷，發現是有封包到來因而發出的中斷後，它會呼叫 *ei_receive( )*來處理，在 ei_receive( )中會呼叫 *ne2k_pci_block_input( )*將封包從網路卡搬到系統記憶體中，然後將封包塞入 sk_buff 的結構中並且以 *netif_rx( )*函式將此封包丟往上層處理，而 netif_rx( ) 所做的事情即是我們之前所討論的後續常式所該做的事。

4. 效能分析

在此我們將會做兩個有趣的效能分析，一個名為“封包的一生”，另一個名為“中斷處理的時間”。第一個測試可以讓我們了解，一個出境以及入境封包在系統的各個部分所停留的時間，如此便可知道封包處理（packet processing）將時間花在哪些地方；第二個測試可以讓我們知道，網路卡驅動程式的不同中斷處理常式所花的時間是完全不同的，而且中斷的類別不同（傳送完畢、接收到新的封包）也有不同的處理方式，我們可依據這個測試結果來做為判斷網路卡和驅路卡驅動程式搭配的好壞與否的因素之一，因為網路卡與網路卡驅動程式搭配的愈好，在中斷處理所花的平均時

間就愈短。

4.1 封包的一生

我們將封包的一生分成三個階段：網路卡階段、DMA 階段，以及網路卡驅動程式階段(不包括 DMA 部分)，會去測量出境及入境封包分別在這三個階段待了多久的時間。量測的方法是先定出每個階段的起訖點然後再使用 rdtscll(定義在<asm/msr.h>)來抓取時間，表七是我們做此測試所用到的硬體、軟體：

CPU：Celeron 567.007 MHz
網路卡：PCI Realtek 10Mbps 乙太網路卡
作業系統核心：Linux kernel 2.2.19
網路卡驅動程式：ne2k-pci

表七：測試一之軟、硬體設備

接下來的表八就是這個測試所得到的結果，時間單位為 CPU clock cycle 數，而由於我們的 CPU 速度為 567.007MHz，所以一個 clock = 1.8 nanosecond。

封包類別　大小(byte)	網路卡階段		DMA 階段	網路卡驅動程式階段	
	入境	出境	入境及出境	入境	出境
ICMP (28 + 1)	62000	12000	7000	1300	3000
ICMP (28 + 100)	102000	52000	11000	1300	3000
ICMP (28 + 1000)	512000	462000	66000	1300	3000
UDP (28 + 1)	62000	12000	7000	1300	3000
UDP (28 + 100)	102000	52000	11000	1300	3000
UDP (28 + 1000)	512000	462000	66000	1300	3000
ARP (28)	62000	12000	7000	1300	3000

在網路卡階段，入境封包和出境封包所存在的時間和封包大小成正比，這是因

為網路卡在傳送封包到網路上，以及從網路上接收封包所花費的時間是和封包的大小成正比；而入境封包會比出境封包在網路卡階段待得久的原因是:

入境封包：接收時間 ＋ 組合語言中斷處理時間 ＋ C 語言中斷處理時間。

出境封包：傳送時間 ＋ 部份 C 語言中斷處理時間。

由於入境封包多花了組合語言中斷處理的時間，再加上其 C 語言中斷處理的部分較出境封包久，所以入境封包在網路卡上所待的時間較出境封包久。在 DMA 階段中，封包所待的時間很明顯的只和封包大小有關，不論是要入境還是要出境的封包。而封包在網路卡驅動程式階段中所待的時間是最短的，尤其是入境封包，這是因為當入境封包 DMA 上網路卡驅動程式後，網路卡驅動程式會立刻呼中 netif_rx 將封包交給後續常式來處理。

4.2 中斷處理的時間

在這個測試中，我們找了兩張網路卡及其對應的網路卡驅動程式，測試兩個驅動程式在中斷處理上所花的時間，當然這也是分成封包傳送完畢以及新封包來到這兩種類型的中斷，表九是此測試所用到的軟、硬體。

CPU：Celeron 567.007 MHz
網路卡：PCI Realtek RTL-8029 (AS) 10Mbps 乙太網路卡 網路卡驅動程式：ne2k-pci
網路卡：PCI Intel GD82559 100Mbps 乙太網路卡 網路卡驅動程式：eepro100
作業系統核心：Linux kernel 2.2.19

表九：測試二之軟、硬體設備

而表十就是這個測試的結果，時間的單位為 CPU clock 數。

中斷類型 封包大小(byte)	新封包來到		封包傳送完畢	
	eepro100	ne2k-pci	eepro100	ne2k-pci
28 + 1	8500	29000	6000	8500
28 + 100	8500	34000	6000	8500
28 + 1000	8500	90000	6000	8500

表十：測試二之結果

由此結果可看出 eepro100 與 Intel 網路卡的組合在中斷處理上所花的平均時間較 ne2k-pci 與 Realtek 網路卡的組合來得少，所以可以說 eepro100 與 Intel 網路卡的組合較 ne2k-pci 與 Realtek 網路卡的組合來得好；另外我們可發現“新封包到來”所需要的時間比“封包傳送完畢”要來得多，這是因為處理新封包到來時必須先將封包從網路卡 DMA 上系統記憶體，之後再塞入 sk_buff 中，並且將其往上層傳遞，而處理“封包傳送完畢”時只是在做些統計工作，例如將傳送的封包數目加一等。

5. 結論

網路卡驅動程式在系統中扮演著重要的角色，由於它必須要與下層網路卡及上層核心溝通，所以要做的事情就變得很多樣化、很複雜，若想要學習寫網路卡驅動程式最好的方式就是找一個範例來改寫。在第二節所介紹的方法是很通用的，不只是網路卡驅動程式適用而已，其它裝置驅動程式的作法也是大同小異。第四節中我們做了兩個很有趣的測試，從這些測試裡面，可以更了解封包傳遞與網路卡驅動程式間的關係，從“封包的一生”測試中可以知道在處理封包時，DMA 及發送（或接收）封包是個時間上的瓶頸，如此一來或許就可以設計某種方式來加快處理封包的速度。

6. 參考文獻

[1] Alessandro Rubini, “Linux Device Drivers”, first edition, O’Reilly & Associates, 1998.

[2] Alessandro Rubini & Jonathan Corbet, “Linux Device Drivers”, 2nd edition, O’Reilly & Associates, June 2001.

[3] Andrew S.Tanenbaum, ” Modern Operating Systems”, Prentice Hall, 1996

[4] W. Richard Stevens, “TCP/IP Illustrated, Volume1&2”, Addison Wesley , 1994

[5] 林盈達, "計算機網路實驗", 維科出版社, 1999。